Digital Audio Processor

EE4G Assignment

Yousef Amar
(1095307)

2014-01-20

Abstract

This report details the design, implementation, and simulation of a digital stereo audio processing unit that receives and transmits data in the AES3 domestic format (aka S/PDIF). The final design can asynchronously receive an AE3 signal at a sampling rate anywhere between 8 kHz and 192 kHz, attenuate the volume and balance of the left and right channels on button presses, and output the processed AES3 signal. The design is also able to adapt to any change in input data frequency and handle errors appropriately.

Contents

1 Introduction **1**

1.1 Project Specifications 1

2 Design, Implementation, and Verification **2**

2.1 Top Level Design 2

2.1.1 Test Bench and Simulation 3

2.2 Receiver 4

2.2.1 Test Bench and Simulation 4

2.3 Processor 4

2.3.1 Test Bench and Simulation 6

2.4 Transmitter 6

2.4.1 Packet Generator 7

2.4.2 Parity Generator 7

2.4.3 Test Bench and Simulation 7

2.5 Clock Recovery 8

2.5.1 Phase Detector 10

2.5.2 Test Bench and Simulation 11

2.6 Miscellaneous 11

3 Conclusion **12**

Appendices **13**

A Appended Figures **13**

B Referenced VHDL Code **21**

B.1 nx3_top.vhd 21

B.2 top_tb.vhd 24

B.3 receiver.vhd 27

B.4 rx_tb.vhd 30

B.5 processor.vhd 33
B.6 pc_tb.vhd 36
B.7 transmitter.vhd 38
B.8 packetgen.vhd 40
B.9 paritygen.vhd 42
B.10 tx_tb.vhd 43
B.11 clkRecovery.vhd 45
B.12 phaseDetector.vhd 47
B.13 cr_tb.vhd 49
B.14 counter.vhd 51
B.15 counterR.vhd 52
B.16 display.vhd 53
B.17 display4.vhd 55

List of Figures

1 Nx3_top RTL Schematic 2

2 Most Common Frequencies 8

3 $f(x) = \frac{10^8}{x}$ 9

4 Nx3_top Detailed RTL Schematic 13

5 Xilinx Design Summary 1 14

6 Xilinx Design Summary 2 15

7 Top Simulation 1 16

8 Top Simulation 2 16

9 Receiver Simulation 1 16

10 Receiver Simulation 2 16

11 Processor Simulation 1 16

12 Processor Simulation 2 17

13 Processor Simulation 3 17

14 Processor Simulation 4 17

15 Processor Simulation 5 17

16 Processor Simulation 6 17

17 Transmitter Simulation 1 18

18 Transmitter Simulation 2 18

19 Transmitter Simulation 3 18

20 Dynamic Phase Shift Simulations 19

21 Clock Recovery Simulation 1 19

22 Clock Recovery Simulation 2 19

23 Clock Recovery Simulation 3 19

24 Clock Recovery Simulation 4 20

25 Clock Recovery Simulation 5 20

26 Clock Recovery Simulation 6 20

27 Clock Recovery Simulation 7 20

1 Introduction

Contained within this report is the full, working design and implementation of AES3 digital stereo audio processing unit in VHDL, as well as complete and rigorous simulation report. Sections are divided into modules; each section explains the reasoning behind a module's design, how it was implemented in VHDL, and how it was verified. All commented source code, simulation waveforms, figures, and diagrams are attached as appendices.

1.1 Project Specifications

The main requirements of the assignment are:

- The final design must consist of four main modules:
  - A clock recovery module capable of recovering the data clock from an input of an unknown sample rate
  - An AES3 receiver capable of decoding left and right audio channels and performing rudimentary error checking via the parity bit
  - A signal processor capable of attenuating each channel independently according to volume and balance user inputs
  - An AES3 transmitter outputting the result of the signal processing operation
- The clock recovery module must be able to support arbitrary sample rates between 8 and 192 kHz with minimal timing jitter.
- Hardware resources required for the processing module should be minimised as much as possible.

2 Design, Implementation, and Verification

The final design is in some places identical to the assignment notes and lab exercises, while it differs significantly in others. A detailed RTL schematic (figure 4) is appended on page 13. Additionally, for the generated design summary, see figure 5 and 6 on pages 14 and 15 respectively. Incidentally, most of the warnings that were given were reminders that the evaluation version of Xilinx ISE is about to run out and are as such completely meaningless. Each of the following sub-section headings have an equivalent VHDL file in the appendix.

2.1 Top Level Design

As modules were designed from the bottom up as opposed to the top down, the structure of this module is emergent. The final entity only adds two new I/O pins to the top level Nexys 3 Spartan-6 FPGA board: `aes3In` and `aes3Out`.

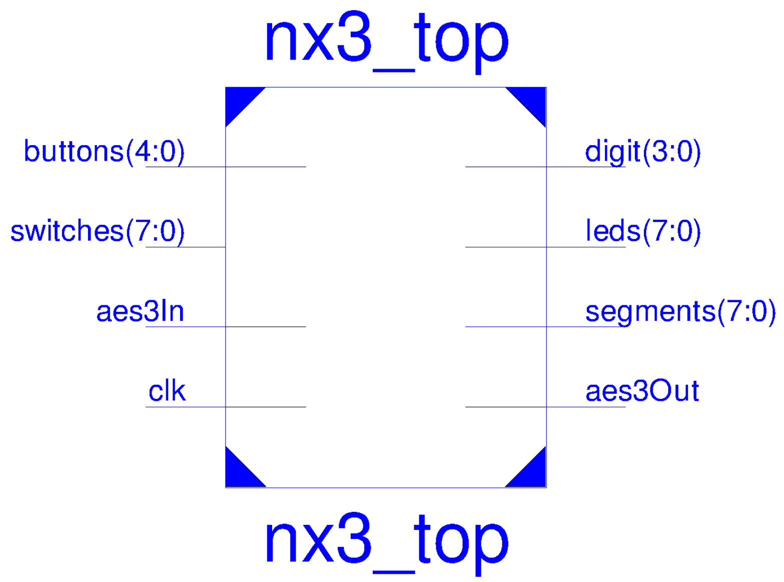


Figure 1: Nx3_top RTL Schematic

All that is done in *nx3_top* is driving the board's LEDs to 0 and connecting all the modules discussed in the rest of the report together. Some differences to the assignment slides design include the fact that only a single bus is used to carry the audio data from module to module. Doing this reduces latency significantly as the processor module does not have to wait until two channels have been received to start outputting but can instead use an extra channel signal – which incidentally doubles as a word clock – to indicate which channel the input data is coming from. Furthermore, since the processor module is asynchronous, the channel line from receiver can be directly fed into the transmitter rather than being routed through the processor module and being registered therein. Both channel and blockStart can be inferred purely from timing, but you cannot always trust the device that the AES3 signal is coming form. For this reason, two extra flags are used to indicate the channel and the start of an audio block.

2.1.1 Test Bench and Simulation

To generate a test AES3 signal, a dummy transmitter is used. Inputs to the dummy transmitter can be modified freely before simulation. A range of sample rates were tested, the fastest supportable rate sample rate being 192 kHz. At that frequency, new data is fed into the dummy transmitter every 333312 ns, which is also the expected output word clock period. This value comes from the a simple calculation. A frequency of $\approx$ 192 kHz means a period of 5208 ns. Since there are 64 bits to sample, the data rate must be 64 times the period, thus 333312 ns. Of course the dummy transmitter must also be clocked at a rate of 192 kHz. Finally, the *nx3_top* module is also fed a 100 MHz clock to simulate the on-board oscillator.

Screen shots of the simulations are appended on pages 16 and 16. Make sure you are viewing this PDF file on a PC and not printed out on paper or you will not be able to zoom in to further inspect the simulation waveforms.

Figure 7 is a demonstration of the pipeline filling. The red lines range the latency and show the correct data travelling from pipeline stage to pipeline stage. The cyan waveforms are all inside the top level block and serve only to help visualise the pipeline stages.

Figure 8 shows a measurement of the latency between a raw AES3 signal coming in and the correct, processed AES3 signal coming out of the top level block. Simulations can be augmented with assert statements for correct outputs but fulfil their purpose even without them.

2.2 Receiver

The receiver is slightly more complicated. The heart of this module is a 64-bit-long shift register. It is fed the AES3 signal and clocked on the falling edge of the data clock. This is done to further protect from inevitable data clock jitter that can come from the clock recovery module. The first 8 bits of this shift register are then continuously compared to the preamble `constants` and their inverses. If the first 8 bits ever match a preamble, a flag is raised. Once that flag is raised, the contents of the shift register are copied into another signal.

Information is then concurrently extracted from that signal. Essentially, the opposite of the packet generator happens here; a bunch of XOR operations to translate biphase mark encoding into usable data. The only additional bit of logic is that a parity generator instance is used to calculate the data parity bit. The result of this is then compared to the received parity bit and if they do not match, the output validity bit is overridden to mute the packet as the signal must have been corrupted somehow. The only time this could happen is if the source of the AES3 signal outputs the corrupt data or while the clock recovery module is still synchronising. Once the clock recovery module has synchronised however – which it does dynamically to both lower and higher frequencies as is discussed later – then the data can only have been corrupted by an outside source.

2.2.1 Test Bench and Simulation

To test the receiver, a method similar to that in the top level design was employed for the same reasons. A dummy transmitter was used to provide the receiver with an input signal and clocked at 192 kHz with its input data changing every 333312 ns.

Screen shots of the simulations are appended on pages 16 and 16. Figure 9 demonstrates expected behaviour with uninitialised signals to aid visualisation. The output matches the input exactly. Figure 10 shows further correct behaviour under the impossible `channel = '1'` and `blockStart = '1'` case that is further discussed in the packet generator section.

2.3 Processor

The main challenges in designing the processor are two-fold; debouncing the button inputs and minimising required hardware resources. Although this design is meant for FPGA synthesis and hardware resources are "free" a different device may be more limiting and as such one must never be wasteful. For this application, it was decided that a centred balance value at 100% volume outputs left and right channels at 50% volume to avoid clipping when panning left and right.

Since attenuating volume and balance are by their very nature scaling operations, it is inevitable that there needs to be some sort of higher level arithmetic operation such as multiplication or division. As it turns out, the minimum are two multiplications. As the module was designed with minimal complexity in mind, it was decided that the channel volumes are to be adjusted in segments of 256ths. This was derived from the decision to allow volume and balance to have a 16 step resolution.

$$\begin{aligned}
dataOut =& \quad dataIn \times volume[0.0 - 1.0] \times balance[0.0 - 1.0] \\
dataOut =& \quad dataIn \times \frac{volume[0 - 16]}{16} \times \frac{balance[0 - 16]}{16} \\
dataOut =& \quad \frac{dataIn}{256} \times volume[0 - 16] \times balance[0 - 16]
\end{aligned}$$

This makes things significantly easier as dataIn can be shifted 8 bits to the right to divide it by 256. Equally the same could have been done with the final result. One potential problem is that the bit shift operation would lose precision as it "rounds" down to the nearest step. This will rarely occur however as dataIn has a range of 0 to 16777215 inclusive and will only do so at 65535 making this hardly an issue.

The result of the multiplication is then stored in another 8 bit larger signal, `idataOut`, for the sake of synthesis. The 8 MSBs of `idataOut` will never contain any data however and as such the signal is truncated before being output without losing any information.

Furthermore, the `balance` value must be inverted depending on the channel. This is done by simply concurrently subtracting it from 16 if need be. Finally, the volume value is also output to the display module to give the user a visual indication of the volume level.

Additionally, the button inputs are debounced. The user can press the button a maximum of ≈6 times a second (≈12 ups and downs). Any more than that is considered bouncing. ≈12 ups and downs a second means 8388607 clock tick intervals between button events. 8388607 (0x7FFFFF) clock tick intervals means a 23 bit counter. Thus, a timer of that size was used to prevent additional button events from registering in the cool-down period. Once the cool-down period is over, a flag is reset and button events can be registered again. Changes in the `buttons` signal and the `buttonsDebounced` signal (i.e. button events) are detected by using a register that stores the previous button states and the current button states are compared against those to detect a button event as well as a button press followed by a release to trigger volume or balance adjustment.

2.3.1 Test Bench and Simulation

The test bench is simple; the input data starts at 10000, steps to 20000 in increments of 2500, then repeats. A step happens after 666624 ns. Meanwhile the channel signal toggles every 333312 ns and therefore the input data is tested on both channels one after the other. At the same time, a madman who can press buttons at a rate of 1.5 kHz is also being simulated. These simulated button presses are expected to only ever be effective every ≈83 ms (≈12 button events per second) due to debouncing.

Screen shots of the simulations are appended on pages 16 to 17. Figure 11 demonstrates correct calculation before:

$$\left\lfloor \frac{10000}{256} \right\rfloor \times 15 \times 15 = 8775$$

and after:

$$\left\lfloor \frac{12500}{256} \right\rfloor \times 14 \times 15 = 10080$$

a button is pressed to decrease the volume and the debounce flag is raised. Figure 12 shows how the same holds true even after the debouce cool-down is over. Figure 13 shows the correct output when the global volume is 0, figure 14 shows the correct output when the balance is all the way to the left, figure 15 shows the correct output when the balance is being adjusted after the first debounce cool-down period, and figure 16 shows the correct output when the balance is all the way to the right as can be verified using the same calculation as above.

2.4 Transmitter

The transmitter is the sub-module with the most nested modules. A counter module is used to count from 0 to 63 repeatedly and is clocked by the data clock. A packet generator module is used to encode the data into BMC and the word clock output is driven by the inverse of the fifth counter bit which transitions after 32 clock cycles making an accurate word clock. Data is sampled near the middle of the signal to be safer. The exact position at which the data is sampled depends on how long the clock recovery module takes to lock on to the AES3 input signal frequency. However, the positions at which the data is sampled are always 64 clock cycles apart so the correct data is always sampled.

The output of the counter module is used to determine which bit should be output to `aes3Out`. This could have been done using a bit on a shift register instead, like with the receiver module, however, unlike the receiver module, not only is this not necessary but would also use exponentially more hardware resources especially if one decided to scale up. Additionally, the parity bit is generated a second time to make absolutely certain that it is uncorrupted.

2.4.1 Packet Generator

The packet generator contains only concurrent statement. The verification of this module and the encapsulated parity generator module is inherent in the verification of the transmitter module. It determines the preamble sequence needed based on the inputs to the module and sets the first 8 bits of the packet to that preamble sequence. If by some glitch `channel = '1'` **and** `blockStart = '1'`, it is assumed that `blockStart` is incorrect. Finally, the rest of the bits are determined using XOR operations to adhere to BMC. The final bit depends on the output of the parity generator instance in this module.

2.4.2 Parity Generator

The parity generator module does what it says on the tin; it uses XOR operations to determine whether there are an even or odd number of 1s in the input word and outputs the according result.

2.4.3 Test Bench and Simulation

This test bench works the exact same way as the processes that drove the inputs to the dummy transmitters in the previous test benches. The timing of the stimuli is likewise exactly the same.

Screen shots of the simulations are appended on pages 18 to 18. Figure 17 demonstrates expected behaviour, figure 18 demonstrates an overridden validity bit, and figure 19 depicts correct behaviour after a long period of time with a different preamble sequence and similarly correct behaviour.

2.5 Clock Recovery

This designing of this module was quite formidable, so much so that an alternative, likewise synthesisable version was made (discussed later) that is easier to follow and predict how the resulting hardware will look from albeit with a lot more code. For example, the alternative treats the input signal like a synchronous reset while the main does not.

The design of the clock recovery module was approached from many different angles. First the most common sampling frequencies were explored. Figure 2 is a plot of the most common frequencies from highest to lowest in the range of 8 kHz to 192 kHz sorted from highest to lowest.

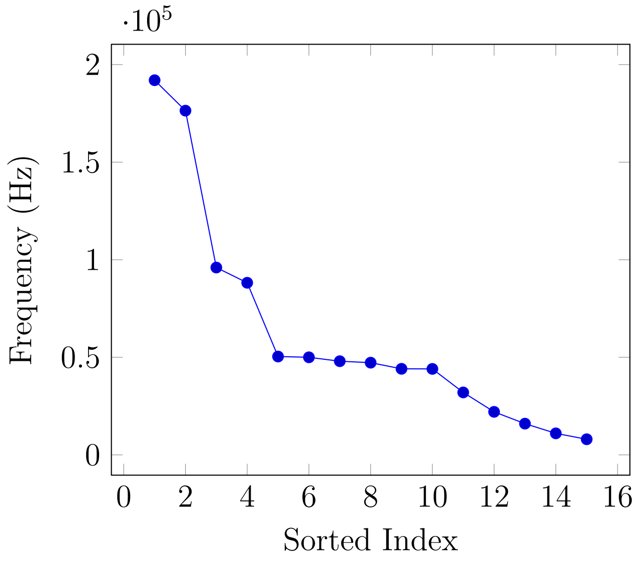


Figure 2: Most Common Frequencies

It seems to almost follow a trend line that would be equivalent to a division by 2 for each step. If that were the case, the implementation could be a counter with a suitable number of bits that would half on-board clock frequency by 2 for each additional bit in the counter. This would allow the clock recovery module to simply adjust the output frequency by changing the bit that is being used as the data clock. The range of frequencies for a 32-bit counter is depicted in figure 3.

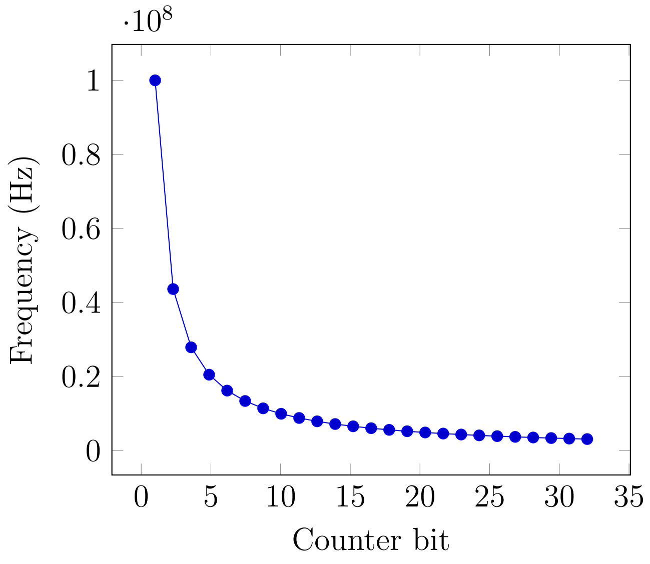


Figure 3: $f(x) = \frac{10^8}{x}$

Due to the small number of common frequencies however, as well as constant "plateauing" of frequencies visible in figure 2, where a few potential sample frequencies cluster very closely together (often for historical reasons), it was decided that this is not a suitable design for the clock recovery module.

Instead, a more versatile algorithm is used. When the AES3 input signal is high, a timer starts counting on the rising edge of the 100 MHz clock. This timer resets on the falling edge of the signal, but before that it is registered into a signal that stores the pulse width. This signal is only updated under specific conditions. These are:

- If signal timer is greater than 150% the current registered pulse width, ignore it because it must be more than one bit.
- If signal timer is less than the minimum 519 (520-1 to account for phase correction) increments (100 MHz clock cycles in 192 kHz pulses; the fastest frequency), ignore it because it is noise.
- The exception is the initial state (all 1s).

If the signal timer goes over 350% the current registered pulse width, then the frequency is clearly wrong because you cannot have four of the same bits in a row in an AES3 signal. If this happens, then the pulse width register is reset (all 1s) and the clock recovery module can resynchronise.

Then another 100 MHz counter is used, that is limited by the registered pulse width, to drive the output clock. It is high for half the limit and low for the other half. It would have been very elegant to use the DCM to output clock frequencies that the most common sampling rates divide nicely into, such as 4.8 MHz (the lowest common multiple of 8, 16, 32, 48, 50, 96, and 192 kHz), 5292 kHz (the lowest common multiple of 11.025, 22.05, 44.1, 47.25, 50.4, 88.2, and 176.4 kHz), and 44.056 kHz (a fringe case), however the DCM is not able to output any of these frequencies so one has to make do with the fastest possible frequency, 100 MHz, to get close enough.

Finally, an Alexander (bang-bang) phase detector is used to correct the output clock phase to match the input signal one clock cycle at a time. This is done by boosting the counter by incrementing it once more on a given clock rising edge or pausing it by decrementing it. This may seem slow at first sight but even in the worst case (180°phase shift) at most a bit will be lost. Once the phase is corrected however, it locks on tight together with the output clock frequency.

2.5.1 Phase Detector

The phase detector contains two architectures, one for a Hogge P.D. and one for an Alexander (bang-bang) P.D. The clock recovery module uses the Alexander P.D. The design of this module is exactly how you would expect it to be; 4 D-types and two XORs. The only change that was made after simulation was splitting the inverted clock (rising_edge else) into its own process (falling_edge) so that the module can synthesise.

An alternative would have been to use the Dynamic Phase Shift option in the DCM module and indeed the DCM module was initially encapsulated in the clock recovery module but eventually became redundant and was removed. In fact, the current design is 100% DCM-free. Another reason it was removed was because you cannot use the DCM Phase Alignment feature and without it, only CLKFX or CLKFX180 output clocks are allowed which only provide limited frequencies. Furthermore the PLL_BASE is not available when Dynamic Phase Shift is being used.

The use of DPS was figured out with the aid of Xilinx documentation. To use it PSEN was pulsed high to shift the phase by one PSCLK cycle. Then you wait for PSDONE to pulse high and you are free to shift the phase again. To see the simulation waveform proving that this method also works, see figure 20.

2.5.2 Test Bench and Simulation

The test bench simply generates a random signal that is 1, 2, or 3 consecutive bits at a time to input as a signal into the clock recovery instance. The bit width is controlled by a variable. This in essence controls the clock frequency that needs to be "recovered".

Screen shots of the simulations are appended on pages 19 to 20. Figure 21 demonstrates correct behaviour at a signal frequency of 192 kHz (maximum). Figure 22 depicts an early error in detected frequency which is then shown to be fixed later in figure 23. Both those simulations ran with a signal frequency of 8 kHz (minimum). Figure 24 shows an example of phase lag which is detected and decreased in figure 25 before finally locking on tight in figure 26. Finally, figure 27 shows how the output of the phase detector cause the phase to sync before eventually oscillating between 01 and 10 once the phase is synchronised and the recovered clock locked onto the signal.

2.6 Miscellaneous

Additional modules were also created for this project that are very self-explanatory. These are all appended to this report and include *counter.vhd*, *counterR.vhd* (a resettable accumulator), *clkRecovery2.vhd* (the alternative clock recovery module), *display.vhd* (a 7-segment display driver), and *display4.vhd* (a driver for four 7-segment displays).

3 Conclusion

Ultimately, a fully working design was completed. All that would have been left to do would be to test it out with a actual AES3 audio signal and an AES3 to analogue converter to use with a speaker. Additionally, further additions could be made to user-friendliness such as a more readable display (decimal instead of hexadecimal) or more input options and button combinations. That is for another time though.

Appendices

A Appended Figures

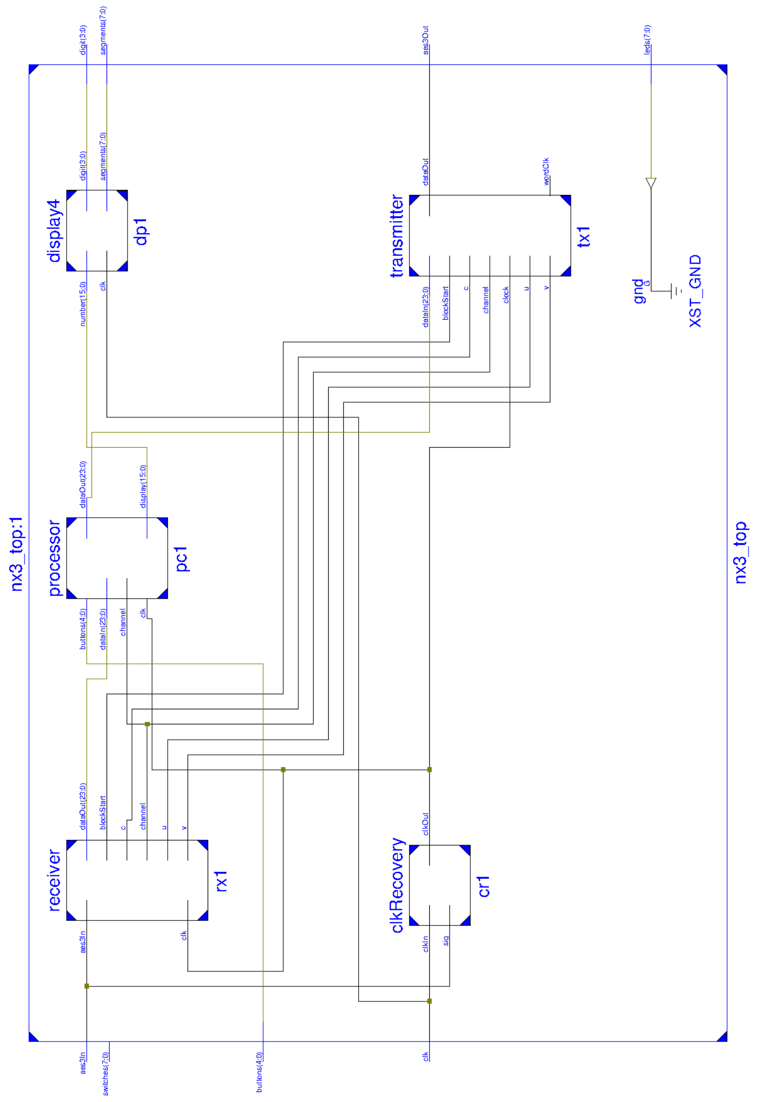
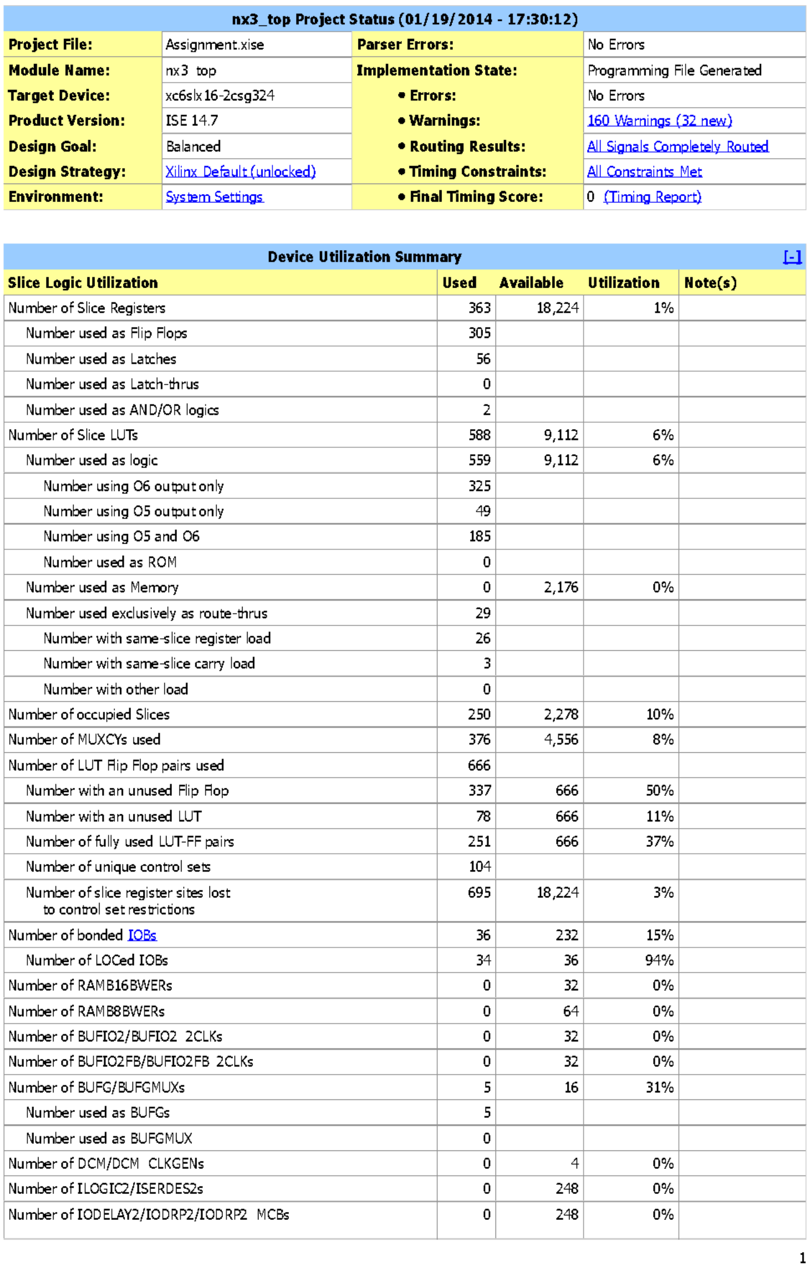


Figure 4: Nx3_top Detailed RTL Schematic



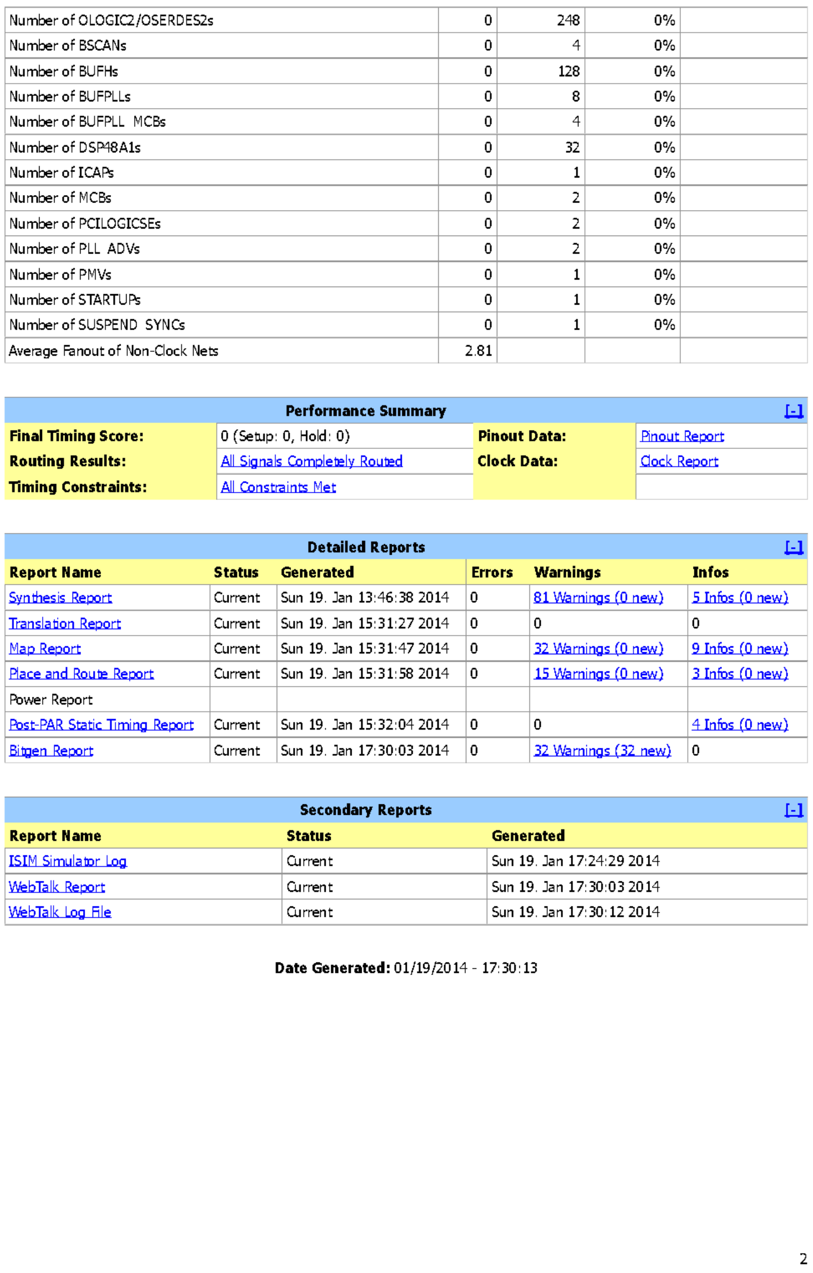
nx3_top Project Status (01/19/2014 - 17:30:12)			
Project File:	Assignment.xise	**Parser Errors:**	No Errors
Module Name:	nx3 top	**Implementation State:**	Programming File Generated
Target Device:	xc6slx16-2csg324	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	160 Warnings (32 new)
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary [-]

Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	363	18,224	1%	
Number used as Flip Flops	305			
Number used as Latches	56			
Number used as Latch-thrus	0			
Number used as AND/OR logics	2			
Number of Slice LUTs	588	9,112	6%	
Number used as logic	559	9,112	6%	
Number using O6 output only	325			
Number using O5 output only	49			
Number using O5 and O6	185			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number used exclusively as route-thrus	29			
Number with same-slice register load	26			
Number with same-slice carry load	3			
Number with other load	0			
Number of occupied Slices	250	2,278	10%	
Number of MUXCYs used	376	4,556	8%	
Number of LUT Flip Flop pairs used	666			
Number with an unused Flip Flop	337	666	50%	
Number with an unused LUT	78	666	11%	
Number of fully used LUT-FF pairs	251	666	37%	
Number of unique control sets	104			
Number of slice register sites lost to control set restrictions	695	18,224	3%	
Number of bonded IOBs	36	232	15%	
Number of LOCed IOBs	34	36	94%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2 2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB 2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	5	16	31%	
Number used as BUFGs	5			
Number used as BUFGMUX	0			
Number of DCM/DCM CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2 MCBs	0	248	0%	

1

Figure 5: Xilinx Design Summary 1



Number of OLOGIC2/OSERDES2s	0	248	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL MCBs	0	4	0%	
Number of DSP48A1s	0	32	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	2.81			

Performance Summary [-]

Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports [-]

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Sun 19. Jan 13:46:38 2014	0	81 Warnings (0 new)	5 Infos (0 new)
Translation Report	Current	Sun 19. Jan 15:31:27 2014	0	0	0
Map Report	Current	Sun 19. Jan 15:31:47 2014	0	32 Warnings (0 new)	9 Infos (0 new)
Place and Route Report	Current	Sun 19. Jan 15:31:58 2014	0	15 Warnings (0 new)	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Sun 19. Jan 15:32:04 2014	0	0	4 Infos (0 new)
Bitgen Report	Current	Sun 19. Jan 17:30:03 2014	0	32 Warnings (32 new)	0

Secondary Reports [-]

Report Name	Status	Generated
ISIM Simulator Log	Current	Sun 19. Jan 17:24:29 2014
WebTalk Report	Current	Sun 19. Jan 17:30:03 2014
WebTalk Log File	Current	Sun 19. Jan 17:30:12 2014

Date Generated: 01/19/2014 - 17:30:13

2

Figure 6: Xilinx Design Summary 2

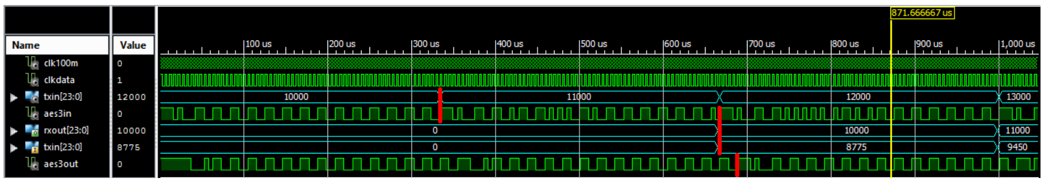


Figure 7: Top Simulation 1

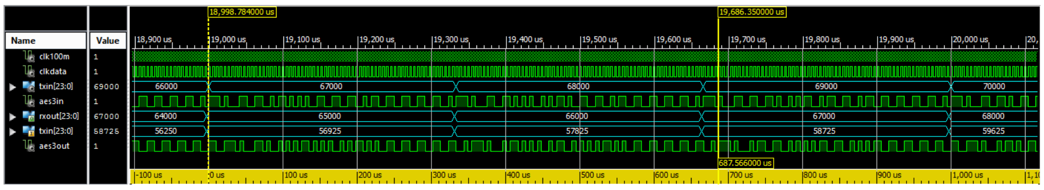


Figure 8: Top Simulation 2

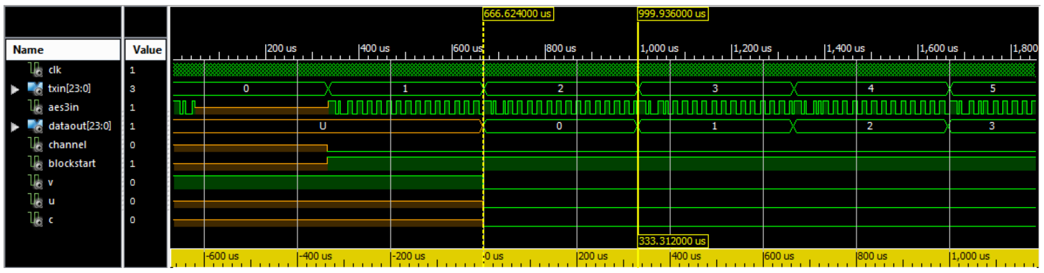


Figure 9: Receiver Simulation 1

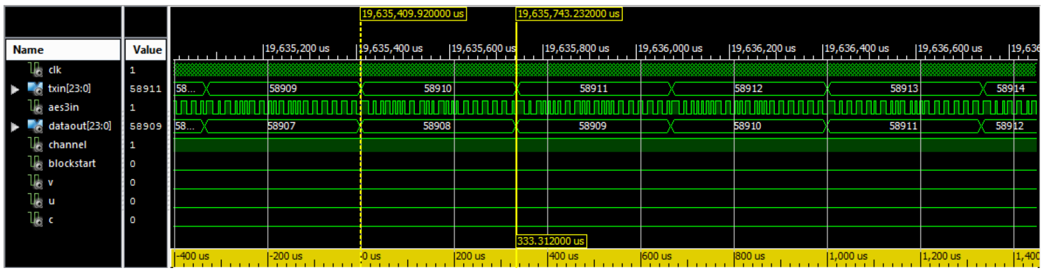


Figure 10: Receiver Simulation 2

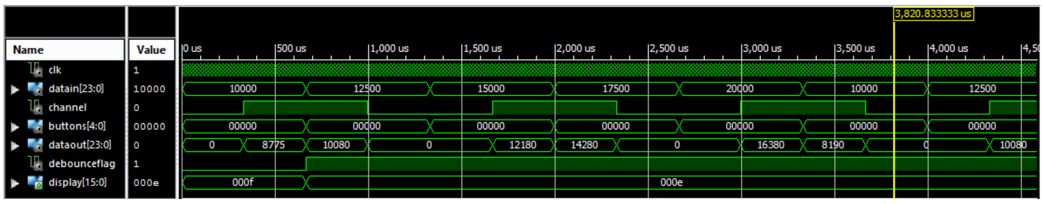


Figure 11: Processor Simulation 1

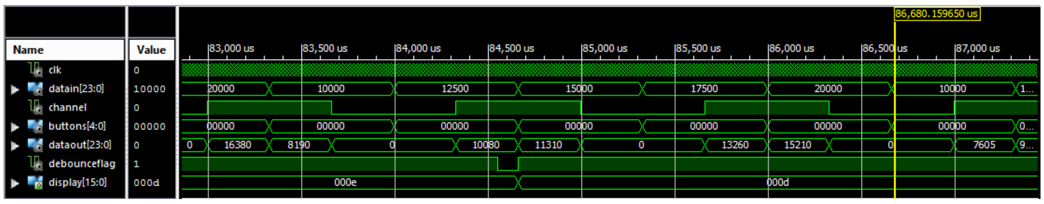


Figure 12: Processor Simulation 2

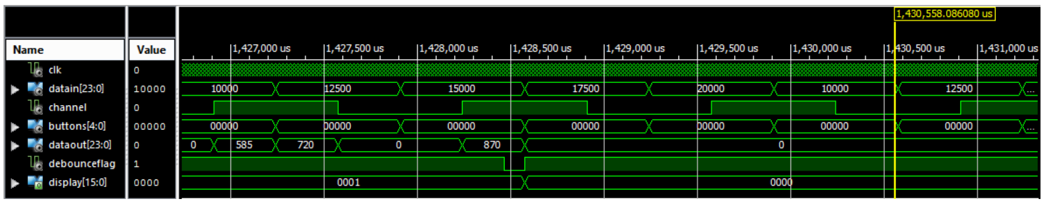


Figure 13: Processor Simulation 3

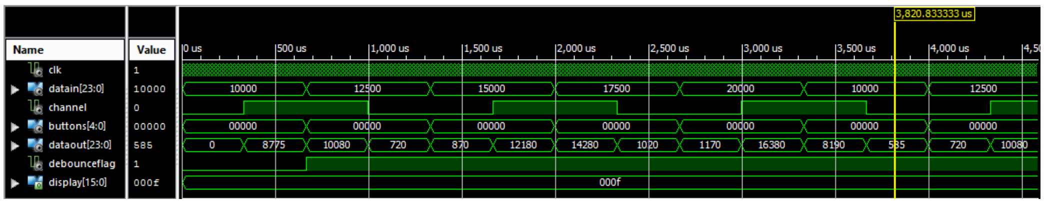


Figure 14: Processor Simulation 4

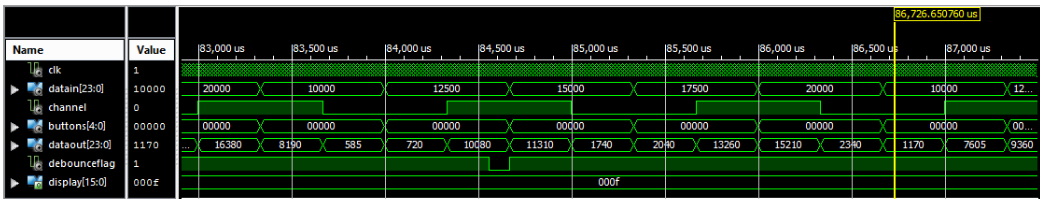


Figure 15: Processor Simulation 5

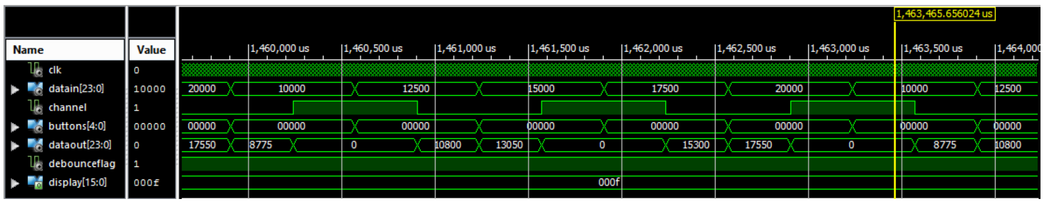


Figure 16: Processor Simulation 6

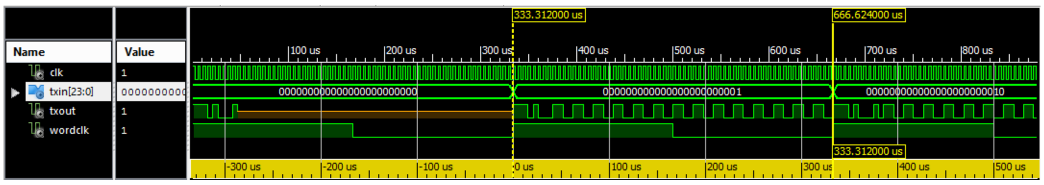


Figure 17: Transmitter Simulation 1

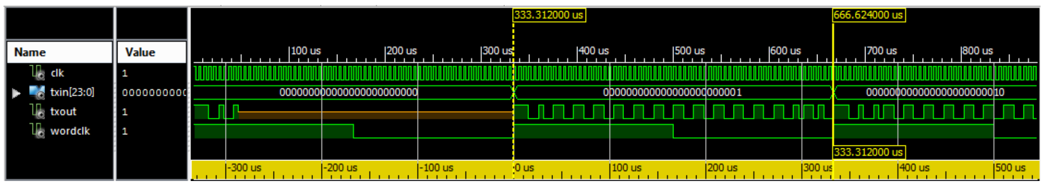


Figure 18: Transmitter Simulation 2

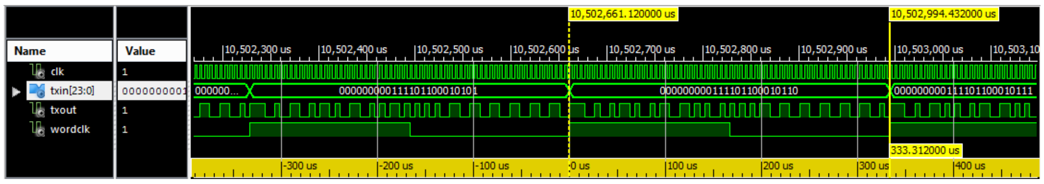


Figure 19: Transmitter Simulation 3

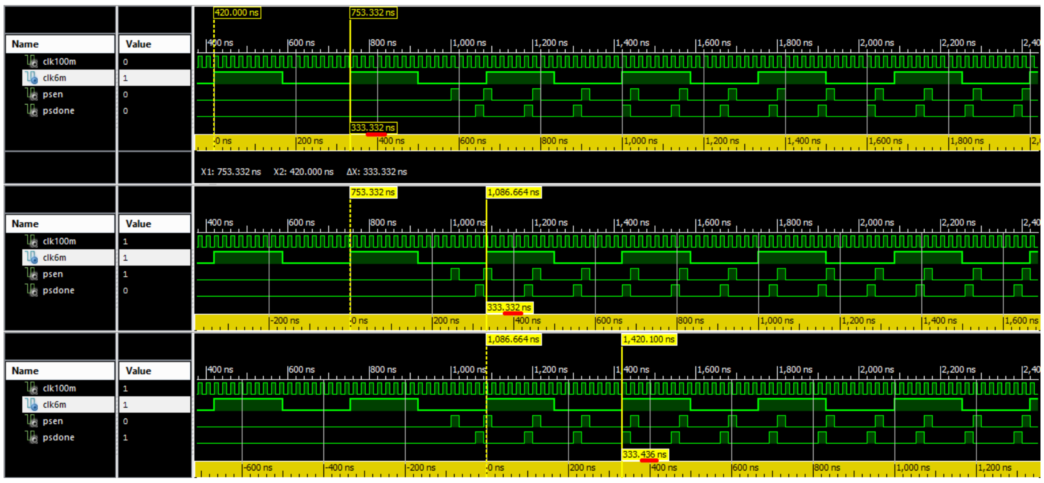


Figure 20: Dynamic Phase Shift Simulations

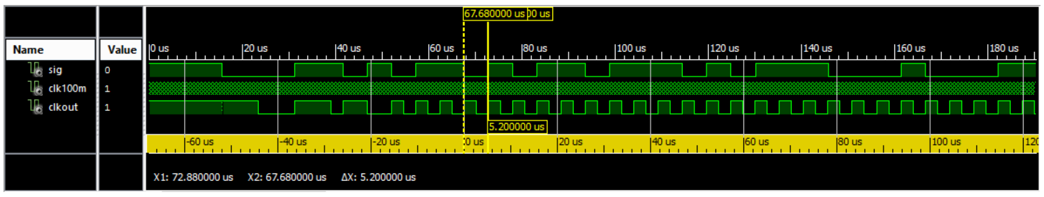


Figure 21: Clock Recovery Simulation 1

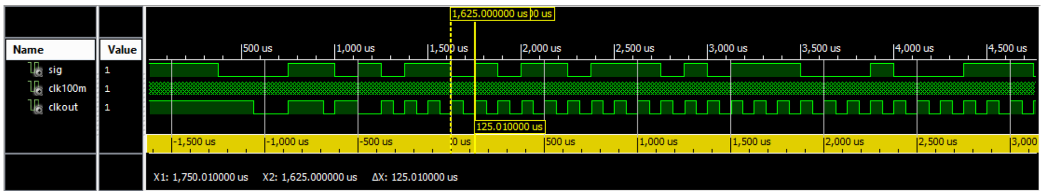


Figure 22: Clock Recovery Simulation 2

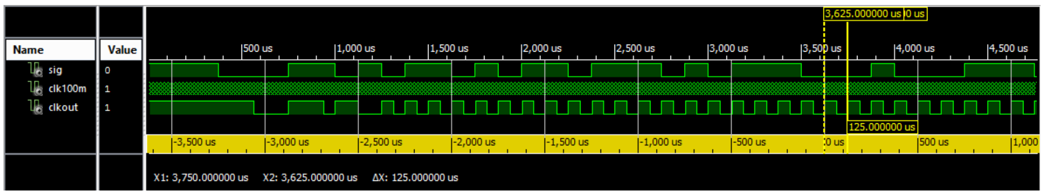


Figure 23: Clock Recovery Simulation 3

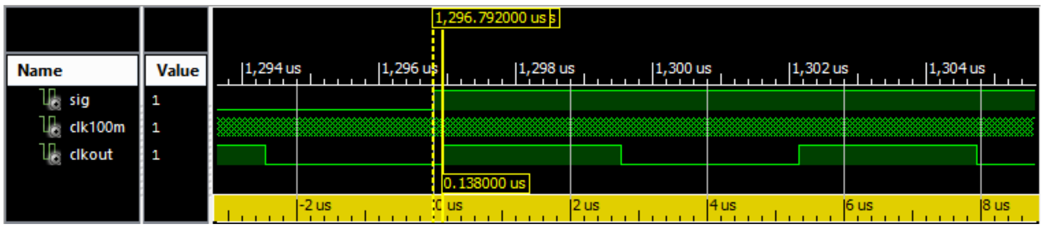


Figure 24: Clock Recovery Simulation 4

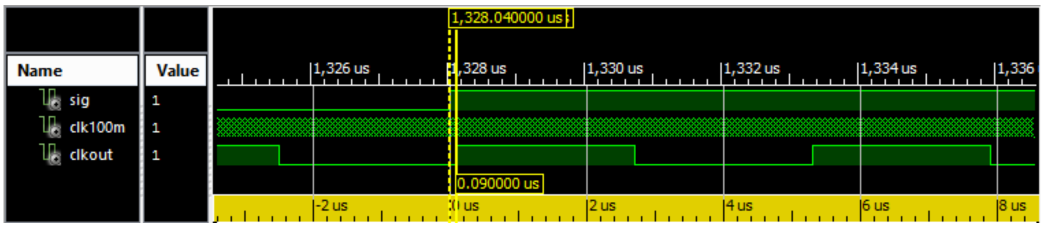


Figure 25: Clock Recovery Simulation 5

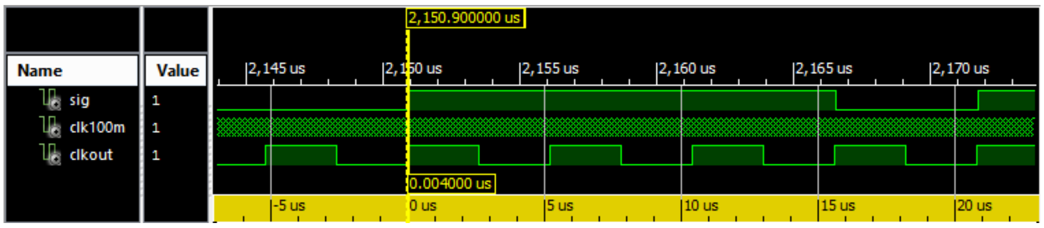


Figure 26: Clock Recovery Simulation 6

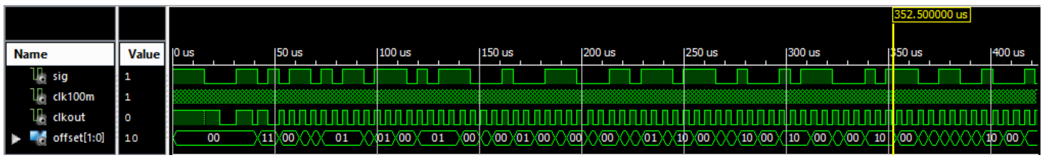


Figure 27: Clock Recovery Simulation 7

B Referenced VHDL Code

B.1 nx3_top.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   nx3_top - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity nx3_top is
  port (
    clk : in std_logic;
    buttons : in std_logic_vector(4 downto 0); -- centre,
       left, up, right, down
    switches : in std_logic_vector(7 downto 0);
    leds : out std_logic_vector(7 downto 0);
    digit : out std_logic_vector(3 downto 0);
    segments : out  std_logic_vector(7 downto 0);

    aes3In : in std_logic;
    aes3Out : out std_logic
  );
end nx3_top;

architecture Behavioral of nx3_top is

  component display4
    port (
      clk : in std_logic;
      number : in std_logic_vector(15 downto 0);
      digit : out std_logic_vector(3 downto 0);
      segments : out std_logic_vector(7 downto 0)
    );
  end component;

  component clkRecovery
    port(
      sig : in std_logic;
      clkIn : in std_logic;
      clkOut : out std_logic
    );
  end component;

  component receiver
    port (
      clk : in std_logic;
      aes3In : in std_logic;

      dataOut : out std_logic_vector (23 downto 0);
      channel : out std_logic; -- Left/A = 0, right/B = 1
      blockStart : out std_logic; -- 0 = false, 1 = true
      v, u, c : out std_logic
    );
  end component;

  component processor
    port (
      clk : in std_logic;
      dataIn : in std_logic_vector(23 downto 0);
      channel: in std_logic; -- Left/A = 0, right/B = 1
      buttons : in std_logic_vector(4 downto 0); -- centre,
         left, up, right, down

      dataOut : out std_logic_vector(23 downto 0);
      display : out std_logic_vector(15 downto 0) -- 0000 =
         0%, 000F = 100%
    );
  end component;

  component transmitter
    port (
      clock: in std_logic;
      dataIn: in std_logic_vector(23 downto 0);
      channel : in std_logic; -- Left/A = 0, right/B = 1
      blockStart : in std_logic; -- 0 = false, 1 = true
      v, u, c : in std_logic;

      dataOut: out std_logic;
      wordClk: out std_logic
    );
  end component;

  signal display : std_logic_vector(15 downto 0);
  signal dataRaw, dataProc : std_logic_vector(23 downto 0);
  signal clkData, channel, blockStart, v, u, c : std_logic;
begin
```

```
  leds <= (others => '0');

  dp1: display4 port map (
      clk => clk,
      number => display,
      digit => digit,
      segments => segments
    );

  cr1: clkRecovery port map (
      sig => aes3In,
      clkIn => clk,
      clkOut => clkData
    );

  rx1: receiver port map (
      clk => clkData,
      aes3In => aes3In,
      dataOut => dataRaw,
      channel => channel,
      blockStart => blockStart,
      v => v, u => u, c => c
    );

  pc1: processor port map (
      clk => clkData,
      dataIn => dataRaw,
      channel => channel,
      buttons => buttons,
      dataOut => dataProc,
      display => display
    );

  tx1: transmitter port map (
      clock => clkData,
      dataIn => dataProc,
      channel => channel,
      blockStart => blockStart,
      v => v, u => u, c => c,
      dataOut => aes3Out,
      wordClk => open
    );

end Behavioral;
```

B.2 top_tb.vhd

```
---------------------------------------------
-- Company:      University of Birmingham
-- Engineer:     Yousef Amar
-- Create Date:    2013-10-28 20:00:00
-- Design Name:    Digital Audio Processor
-- Module Name:    top_tb - behavior
-- Project Name:   EE4G
-- Target Devices:   xc6slx16
-- Description:    EE4G Assignment
---------------------------------------------

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


entity top_tb is
end top_tb;

architecture behavior of top_tb is

  -- UUT
  component nx3_top
    port(
      clk : in std_logic;
      buttons : in std_logic_vector(4 downto 0);
      switches : in std_logic_vector(7 downto 0);
      leds : out std_logic_vector(7 downto 0);
      digit : out std_logic_vector(3 downto 0);
      segments : out std_logic_vector(7 downto 0);

      aes3In : in std_logic;
      aes3Out : out std_logic
    );
  end component;

  component transmitter
    port (
      clock: in std_logic;
      dataIn: in std_logic_vector(23 downto 0);
      channel : in std_logic; -- Left/A = 0, right/B = 1
      blockStart : in std_logic; -- 0 = false, 1 = true
      v, u, c : in std_logic;

      dataOut: out std_logic;
      wordClk: out std_logic
    );
  end component;

  --Inputs
  signal clk100M : std_logic := '0';
  signal buttons : std_logic_vector(4 downto 0) := (others =>
      '0');
  signal switches : std_logic_vector(7 downto 0) := (others
     => '0');
  signal aes3In, aes3Out : std_logic := '0';

  --Outputs
  signal leds : std_logic_vector(7 downto 0);
  signal digit : std_logic_vector(3 downto 0);
  signal segments : std_logic_vector(7 downto 0);

  signal clkData : std_logic := '1';
  signal txIn : std_logic_vector(23 downto 0) := X"002710";
     -- Start at 10000 to be loud enough
begin

  uut: nx3_top
    port map (
      clk => clk100M,
      buttons => buttons,
      switches => switches,
      leds => leds,
      digit => digit,
      segments => segments,

      aes3In => aes3In,
      aes3out => aes3Out
    );

  -- Use a dummy transmitter to generate test data
  tx1: transmitter port map (
      clock => clkData,
      dataIn => txIn,
      channel => '0',
      blockStart => '1',
      v => '0', u => '0', c => '0',

      dataOut => aes3In,
      wordClk => open
    );

  -- Continuously give the dummy transmitter test data to
     generate an AES3 signal
  txDataProc: process
  begin
```

```
    -- 5208 ns (~192 kHz period) * 64 (bits to sample) =
       333312 ns (== word clock period)
    wait for 333312 ns;
    txIn <= txIn + 1000;
  end process;

  -- 192 kHz clock for dummy transmitter
  clkDataProc: process
  begin
    wait for 2604 ns;
    clkData <= not clkData;
  end process;

  -- 100 MHz clock for nx3_top to simulate on-board
     oscillator
  clk100MProc: process
  begin
    clk100M <= '1';
    wait for 5 ns;
    clk100M <= '0';
    wait for 5 ns;
  end process;

end;
```

B.3 receiver.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   receiver - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity receiver is
  port (
    clk : in std_logic;
    aes3In : in std_logic;

    dataOut : out std_logic_vector (23 downto 0);
    channel : out std_logic; -- Left/A = 0, right/B = 1
    blockStart : out std_logic; -- 0 = false, 1 = true
    v, u, c : out std_logic
  );
end receiver;

architecture Behavioral of receiver is
  signal shiftReg : std_logic_vector(63 downto 0) := (others
     => '0');
  signal packet : std_logic_vector(63 downto 0) := (others =>
      '0');
  signal preambleFlag, parityPacket, parityCalcd : std_logic
     := '0';
  signal idataOut: std_logic_vector(23 downto 0);

  component paritygen
    generic (N : integer);
    port (
      word_in : in std_logic_vector(N-1 downto 0);
      parity : out std_logic
    );
  end component;

  constant preambleX : std_logic_vector(7 downto 0) :=
     "01000111"; -- Start of A
  constant preambleY : std_logic_vector(7 downto 0) :=
     "00100111"; -- Start of B
  constant preambleZ : std_logic_vector(7 downto 0) :=
     "00010111"; -- Start of Audio Block and A
begin

  -- Extract information form packet

  dataOut <= idataOut;

  dataGen: for i in 0 to 23 generate
    idataOut(i) <= packet(2*i + 8) xor packet(2*i + 9);
  end generate;

  channel <= packet(3) xor packet(5);
  blockStart <= packet(3) xor packet(4);

  -- 0 = valid, 1 = muted. If the parity bit is incorrect,
     the validity bit is overridden and muted.
  v <= '0' when (packet(56) xor packet(57)) = '0' and
     parityCalcd = parityPacket else '1';
  u <= packet(58) xor packet(59);
  c <= packet(60) xor packet(61);

  -- TODO: Consider separating into own module
  parityPacket <= packet(62) xor packet(63);
  pGen: paritygen generic map (N=>24) port map (idataOut,
     parityCalcd);

  preambleFlag <= '1' when shiftReg(7 downto 0) = preambleX
     or shiftReg(7 downto 0) = not preambleX
           or shiftReg(7 downto 0) = preambleY or shiftReg(7
               downto 0) = not preambleY
           or shiftReg(7 downto 0) = preambleZ or shiftReg(7
               downto 0) = not preambleZ else '0';

  -- Register packet on preamble detection
  -- TODO: Consider making concurrent
  packetProc: process (preambleFlag)
  begin
    if rising_edge(preambleFlag) then
      packet <= shiftReg;
    end if;
  end process;

  -- Clock shift register on falling edge to hit the middle
     of a bit just to be safe from clock jitter.
  shiftRegProc: process (clk)
  begin
    if falling_edge(clk) then
      shiftReg(62 downto 0) <= shiftReg(63 downto 1);
      shiftReg(63) <= aes3In;
    end if;
  end process;

end Behavioral;
```

B.4 rx_tb.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   rx_tb - behavior
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity rx_tb is
end rx_tb;

architecture behavior of rx_tb is

  component transmitter
    port (
      clock: in std_logic;
      dataIn: in std_logic_vector(23 downto 0);
      channel : in std_logic; -- Left/A = 0, right/B = 1
      blockStart : in std_logic; -- 0 = false, 1 = true
      v, u, c : in std_logic;

      dataOut: out std_logic;
      wordClk: out std_logic
    );
  end component;

  component receiver
    port (
      clk : in std_logic;
      aes3In : in std_logic;

      dataOut : out std_logic_vector (23 downto 0);
      channel : out std_logic; -- Left/A = 0, right/B = 1
      blockStart : out std_logic; -- 0 = false, 1 = true
      v, u, c : out std_logic
    );
  end component;

  -- UUT Inputs
  signal clk : std_logic := '1';
  signal aes3In : std_logic := '0';

  -- UUT Outputs
  signal dataOut : std_logic_vector(23 downto 0);
  signal channel : std_logic; -- Left/A = 0, right/B = 1
  signal blockStart : std_logic; -- 0 = false, 1 = true
  signal v, u, c : std_logic;

  signal txIn : std_logic_vector(23 downto 0) := (others =>
     '0');
begin

  -- Dummy transmitter instance
  tx1: transmitter port map (
      clock => clk,
      dataIn => txIn,
      channel => '1',
      blockStart => '0',
      v => '0', u => '0', c => '0',

      dataOut => aes3In,
      wordClk => open
    );

  uut: receiver port map (
      clk => clk,
      aes3In => aes3In,

      dataOut => dataOut,
      channel => channel,
      blockStart => blockStart,
      v => v, u => u, c => c
    );

  -- Dummy transmitter input process
  stimProc: process
  begin
    -- 5208 ns (~192 kHz period) * 64 (bits to sample) =
        333312 ns (== word clock period)
    wait for 333312 ns;
    txIn <= txIn + 1;
  end process;

  -- Clock frequency ~192 kHz (maximum)
  clkProc: process
  begin
    clk <= '1';
    wait for 2604 ns;
    clk <= '0';
    wait for 2604 ns;
  end process;
end;
```

B.5 processor.vhd

```
----------------------------------------------
-- Company:      University of Birmingham
-- Engineer:     Yousef Amar
-- Create Date:    2013-10-28 20:00:00
-- Design Name:    Digital Audio Processor
-- Module Name:    processor - Behavioral
-- Project Name:   EE4G
-- Target Devices:   xc6slx16
-- Description:    EE4G Assignment
----------------------------------------------

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

entity processor is
  port (
    clk : in std_logic;
    dataIn : in std_logic_vector(23 downto 0);
    channel: in std_logic; -- Left/A = 0, right/B = 1
    buttons : in std_logic_vector(4 downto 0); -- centre,
       left, up, right, down

    dataOut : out std_logic_vector(23 downto 0);
    display : out std_logic_vector(15 downto 0) -- 0000 = 0%,
        000F = 100%
  );
end processor;

architecture Behavioral of processor is
  signal buttonsReg, buttonsDebounced, buttonsDebouncedReg :
     std_logic_vector(4 downto 0) := (others => '0');
  signal debounceTimer : std_logic_vector(22 downto 0) := (
     others => '0');
  signal debounceFlag : std_logic := '0';

  signal dataIn256th : std_logic_vector(23 downto 0);
  signal idataOut : std_logic_vector(31 downto 0);

  signal volume : std_logic_vector(3 downto 0) := (others =>
     '1');
  signal balance : std_logic_vector(3 downto 0) := (others =>
      '0');
  signal balanceC : std_logic_vector(3 downto 0);
begin

  -- Invert balance depending on channel
```

```
balanceC <= 15 - balance when channel = '0' else balance;

-- Attenuate
-- NOTE: See report for detailed explanation.
dataIn256th <= X"00" & dataIn(23 downto 8);
idataOut <= dataIn256th * volume * balanceC;
dataOut <= idataOut(23 downto 0);

-- Output volume to Nexys 3 board display
display <= X"000" & volume;

-- Debounce button inputs (see report for timing details)
debounceProc: process (clk, buttons)
begin
  if rising_edge(clk) then
    buttonsReg <= buttons;

    -- If a button was pressed or released
    if buttons /= buttonsReg and debounceTimer = 0 then
      buttonsDebounced <= buttons;
      debounceFlag <= '1';
    end if;

    if debounceFlag = '1' then
      debounceTimer <= debounceTimer + 1;
      -- NOTE: 8388607 = 0x7FFFFF
      if (debounceTimer = 8388607) then
        debounceFlag <= '0';
      end if;
    end if;
  end if;
end process;

-- Adjust volume and balance levels depending on debounced
   button inputs
controlProc: process (buttonsDebounced)
begin
  buttonsDebouncedReg <= buttonsDebounced;

  -- Up (increase volume)
  if buttonsDebounced(2) = '1' and buttonsDebouncedReg(2) =
      '0' then
    if volume < X"F" then
      volume <= volume + 1;
    end if;
  end if;
  -- Down (decrease volume)
  if buttonsDebounced(0) = '1' and buttonsDebouncedReg(0) =
      '0' then
```

```
        if volume > X"0" then
          volume <= volume - 1;
        end if;
      end if;
      -- Left (pan left)
      if buttonsDebounced(3) = '1' and buttonsDebouncedReg(3) =
          '0' then
        if balance > X"0" then
          balance <= balance - 1;
        end if;
      end if;
      -- Right (pan right)
      if buttonsDebounced(1) = '1' and buttonsDebouncedReg(1) =
          '0' then
        if balance < X"F" then
          balance <= balance + 1;
        end if;
      end if;
    end process;

  end Behavioral;
```

B.6 pc_tb.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   pc_tb - behavior
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity pc_tb is
end pc_tb;

architecture behavior of pc_tb is
  component processor
    port (
      clk : in std_logic;
      dataIn : in std_logic_vector(23 downto 0);
      channel: in std_logic; -- Left/A = 0, right/B = 1
      buttons : in std_logic_vector(4 downto 0); -- centre,
         left, up, right, down

      dataOut : out std_logic_vector(23 downto 0);
      display : out std_logic_vector(15 downto 0) -- 0000 =
         0%, 000F = 100%
    );
  end component;

  -- UUT Inputs
  signal clk : std_logic := '1';
  signal dataIn : std_logic_vector(23 downto 0) := X"002710";
  signal channel : std_logic := '0';
  signal buttons : std_logic_vector(4 downto 0) := (others =>
      '0'); -- Try next combination after repeat.

  -- UUT Outputs
  signal dataOut : std_logic_vector(23 downto 0);
begin

  uut: processor port map (
      clk => clk,
      dataIn => dataIn,
      channel => channel,
      buttons => buttons,

      dataOut => dataOut,
      display => open
    );

  -- Simulate data
  -- NOTE: Starts at 10000, steps to 20000 in increments of
     2500 then repeats.
  stimProc: process
  begin
    wait for 333312 ns;
    channel <= not channel;
    wait for 333312 ns;

    if (dataIn < 20000) then
      dataIn <= dataIn + 2500;
    else
      dataIn <= X"002710";
    end if;
  end process;

  -- Simulate button presses
  buttonProc: process
  begin
    wait for 666620 ns;
    -- Volume Down
    buttons(0) <= '1';
    wait for 4 ns;
    buttons(0) <= '0';

    -- Balance Right
    --buttons(1) <= '1';
    --wait for 4 ns;
    --buttons(1) <= '0';
  end process;

  -- Simulate 100 MHz clock
  clkProc: process
  begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

end;
```

B.7 transmitter.vhd

```
----------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   transmitter - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
----------------------------------------------

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

entity transmitter is
  port (
    clock: in std_logic;
    dataIn: in std_logic_vector(23 downto 0);
    channel : in std_logic; -- Left/A = 0, right/B = 1
    blockStart : in std_logic; -- 0 = false, 1 = true
    v, u, c : in std_logic;

    dataOut: out std_logic;
    wordClk: out std_logic
  );
end transmitter;

architecture Behavioral of transmitter is
  component counter
    generic (N : integer);
    port (
      clock : in std_logic;
      count : out std_logic_vector(N-1 downto 0)
    );
  end component;

  component packetgen
    port (
      data : in std_logic_vector(23 downto 0);
      previousFinalBit : in std_logic;
      channel : in std_logic; -- Left/A = 0, right/B = 1
      blockStart : in std_logic; -- 0 = false, 1 = true
      v, u, c : in std_logic;

      txBuff : out std_logic_vector (63 downto 0)
    );
  end component;

  signal prevFinalBit : std_logic;
  signal packet : std_logic_vector(63 downto 0);
  signal sample, sampleReg : std_logic_vector(23 downto 0) :=
      (others => '0');
  signal count : std_logic_vector(5 downto 0);
begin

  cnt1: counter generic map (N => 6) port map (clock => clock
     , count => count);

  pkg1: packetgen port map (
      data => sample,
      previousFinalBit => prevFinalBit,
      channel => channel,
      blockStart => blockStart,
      v => v, u => u, c => c,
      txBuff => packet
    );

  wordClk <= not count(5);

  -- Sample data near the middle of the signal to be safer
  sampleReg <= dataIn when count = "100000";

  -- Drive AES output using count as a packet bit index
  dataOut <= packet(conv_integer(count));

  onWordProc: process (clock)
  begin
    if rising_edge(clock) then
      -- These have to be set one clock cycle before count
         reset as the counter is also clocked on the rising
         edge.
      if count = "111111" then
        -- TODO: Consider making some of this concurrent.
        prevFinalBit <= packet(63);
        sample <= sampleReg;
      end if;
    end if;
  end process;

end Behavioral;
```

B.8 packetgen.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   packetgen - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity packetgen is
  port (
    data : in std_logic_vector(23 downto 0);
    previousFinalBit : in std_logic;
    channel : in std_logic; -- Left/A = 0, right/B = 1
    blockStart : in std_logic; -- 0 = false, 1 = true
    v, u, c : in std_logic;

    txBuff : out std_logic_vector(63 downto 0)
  );
end packetgen;

architecture Behavioral of packetgen is
  signal itxBuff : std_logic_vector (63 downto 0) := (OTHERS
     => '0');
  signal p : std_logic := '0';
  signal preambleFlagX, preambleFlagY, preambleFlagZ :
     std_logic;

  component paritygen
    generic (N : integer);
    port (
      word_in : in std_logic_vector (N-1 downto 0);
      parity : out std_logic
    );
  end component;
begin

  txBuff <= itxBuff;

  pg1: paritygen generic map (N=>24) port map (word_in =>
     data, parity => p);
  -- Determine preamble sequence
  preambleFlagX <= channel nor blockStart;
  preambleFlagY <= channel;
  preambleFlagZ <= blockStart;--not channel and blockStart;

  -- Set preamble sequence
  itxBuff(7 downto 0) <= "10111000" when (preambleFlagX and
     previousFinalBit) = '1' else "01000111" when
     preambleFlagX = '1'
           else "11011000" when (preambleFlagY and
              previousFinalBit) = '1' else "00100111" when
              preambleFlagY = '1'
           else "11101000" when (preambleFlagZ and
              previousFinalBit) = '1' else "00010111" when
              preambleFlagZ = '1';

  -- Set first bits
  gen1: for i in 0 to 27 generate
    itxBuff(2*i + 8) <= not itxBuff(2*i + 7);
  end generate;

  -- Set second data bits
  gen2: for i in 0 to 23 generate
    itxBuff(2*i + 9) <= itxBuff(2*i + 8) xor data(i);
  end generate;

  -- Set second meta-data bits
  itxBuff(2*24 + 9) <= itxBuff(2*24 + 8) xor v;
  itxBuff(2*25 + 9) <= itxBuff(2*25 + 8) xor u;
  itxBuff(2*26 + 9) <= itxBuff(2*26 + 8) xor c;
  itxBuff(2*27 + 9) <= itxBuff(2*27 + 8) xor p;

end Behavioral;
```

B.9 paritygen.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   paritygen - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity paritygen is
  generic (N : integer);
  port (
    word_in : in std_logic_vector (N-1 downto 0);
    parity : out std_logic
  );
end paritygen;

architecture Behavioral of paritygen is
  signal p: std_logic_vector(N-1 downto 0);
begin
  p(0) <= word_in(0);

  gen1: for i in 1 to N-1  generate
    p(i) <= p(i-1) xor word_in(i);
  end generate;

  parity <= p(N-1);
end Behavioral;
```

B.10 tx_tb.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   tx_tb - behavior
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity tx_tb is
end tx_tb;

architecture behavior of tx_tb is
  -- UUT
  component transmitter
    port (
      clock: in std_logic;
      dataIn: in std_logic_vector(23 downto 0);
      channel : in std_logic; -- Left/A = 0, right/B = 1
      blockStart : in std_logic; -- 0 = false, 1 = true
      v, u, c : in std_logic;

      dataOut: out std_logic;
      wordClk: out std_logic
    );
  end component;

  --Inputs
  signal clk : std_logic := '1';
  signal txIn : std_logic_vector(23 downto 0) := (others =>
     '0');

  --Outputs
  signal txOut : std_logic := '0';
  signal wordClk : std_logic;
begin

  uut: transmitter port map (
      clock => clk,
      dataIn => txIn,
      channel => '1',
      blockStart => '0',
      v => '0', u => '0', c => '0',

      dataOut => txOut,
      wordClk => wordClk
    );

  -- Transmitter data input process
  stimProc: process
  begin
    -- 5208 ns (~192 kHz period) * 64 (bits to sample) =
       333312 ns (== word clock period)
    wait for 333312 ns;
    txIn <= txIn + 1;
  end process;

  -- Clock frequency ~192 kHz (maximum)
  clkProc: process
  begin
    clk <= '1';
    wait for 2604 ns;
    clk <= '0';
    wait for 2604 ns;
  end process;

end;
```

B.11 clkRecovery.vhd

```
---------------------------------------------
-- Company:      University of Birmingham
-- Engineer:     Yousef Amar
-- Create Date:    2013-10-28 20:00:00
-- Design Name:    Digital Audio Processor
-- Module Name:    clkRecovery - Behavioral
-- Project Name:   EE4G
-- Target Devices:   xc6slx16
-- Description:    EE4G Assignment
---------------------------------------------

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

entity clkRecovery is
  port (
    sig : in std_logic;
    clkIn : in std_logic;
    clkOut : out std_logic
  );
end clkRecovery;

architecture Behavioral of clkRecovery is
  component phaseDetector
      port (
      sig : in std_logic;
      clk : in std_logic;
      offset : out std_logic_vector(1 downto 0)
    );
  end component;

  signal counter : std_logic_vector(31 downto 0) := (others
     => '0');

  signal timer : std_logic_vector(31 downto 0) := (others =>
     '0');
  signal pulseW : std_logic_vector(31 downto 0) := (others =>
      '1');

  signal iclkOut : std_logic;
  signal offset : std_logic_vector(1 downto 0);
begin

  -- Drive output clock
  iclkOut <= '1' when counter < '0' & pulseW(31 downto 1) or
     counter > pulseW else '0';
  clkOut <= iclkOut;

  pd1: entity work.phaseDetector(alex) port map (sig => sig,
      clk => iclkOut, offset => offset);

  -- Recover data clock and correct phase
  crProc: process (clkIn, sig)
    -- NOTE: No extra bit needed for sum since counter will
        never get that high anyway at these frequencies.
    variable sum : std_logic_vector(31 downto 0);
  begin
    -- NOTE: Could potentially double resolution by using the
        falling edge too?
    if rising_edge(clkIn) then
      -- Drive pulse width timer
      if sig = '1' then
        timer <= timer + 1;
      else
        timer <= (others => '0');
      end if;

      -- Drive output clock counter
      sum := counter + 1;
      if sum >= pulseW then
        counter <= sum - pulseW;
      else
        counter <= sum;
      end if;
    end if;

    -- Correct phase once clock cycle at a time
    if rising_edge(sig) then
      if offset = "10" then
        counter <= counter + 1;
      elsif offset = "01" then
        counter <= counter - 1;
      end if;
    -- Register timed pulse width under the right conditions
        (see report for detailed explanation)
    elsif falling_edge(sig) and ((pulseW = X"FFFFFFFF") or (
        timer < pulseW + ('0'&pulseW(31 downto 1)) and timer
        >= 519)) then
      pulseW <= timer;
    end if;
  end process;

end Behavioral;
```

B.12 phaseDetector.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   phaseDetector - alex, hogge
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity phaseDetector is
  port (
    sig : in std_logic;
    clk : in std_logic;
    offset : out std_logic_vector (1 downto 0)
  );
end phaseDetector;

architecture alex of phaseDetector is
  signal d1, d2, d3, d4 : std_logic := '0';
begin

  -- NOTE: 01 means early, 10 means late
  offset(0) <= d2 xor d4; -- X
  offset(1) <= d1 xor d4; -- Y

  regProc: process (clk)
  begin
    if rising_edge(clk) then
      d1 <= sig;
      d2 <= d1;
      d4 <= d3;
    end if;
  end process;

  regNotProc: process (clk)
  begin
    if falling_edge(clk) then
      d3 <= sig;
    end if;
  end process;
end alex;
```

```
architecture hogge of phaseDetector is
  signal d1, d3 : std_logic := '0';
begin

  offset(0) <= d1 xor d3;
  offset(1) <= sig xor d1;

  regProc: process (clk)
  begin
    if rising_edge(clk) then
      d1 <= sig;
      d3 <= d1;
    end if;
  end process;
end hogge;
```

B.13 cr_tb.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   cr_tb - behavior
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
use ieee.numeric_std.all;

entity cr_tb is
end cr_tb;

architecture behavior of cr_tb is
  -- UUT
  component clkRecovery
    port(
      sig : in std_logic;
      clkIn : in std_logic;
      clkOut : out std_logic
    );
  end component;

  signal sig : std_logic := '0';
  signal clk100M : std_logic := '1';
  signal clkOut : std_logic;
begin

  uut: clkRecovery port map (sig => sig, clkIn => clk100M,
      clkOut => clkOut);

  -- Generate a random signal (1, 2, or 3 consecutive bits at
      a time).
  sigProc: process
    variable seed1, seed2: positive;
    variable rand : real;
    variable bitWidth : time := 5208 ns; -- ~192 kHz
    --variable bitWidth : time := 125 us; -- ~8 kHz
  begin
    uniform(seed1, seed2, rand);
    sig <= not sig;
    wait for (integer(trunc(rand*3.0))*bitWidth) + bitWidth;
  end process;

  clk100Mproc: process
  begin
    clk100M <= '1';
    wait for 5 ns;
    clk100M <= '0';
    wait for 5 ns;
  end process;

end;
```

B.14 counter.vhd

```vhdl
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   counter - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity counter is
  generic (N : integer);
  port (
    clock : in std_logic;
    count : out std_logic_vector(N-1 downto 0)
  );
end counter;

architecture Behavioral of counter is
  signal icount: std_logic_vector(N-1 downto 0) := (others =>
      '0');
begin
  count <= icount;

  incProc: process (clock)
  begin
    if rising_edge(clock) then
      icount <= icount + 1;
    end if;
  end process;
end Behavioral;
```

B.15 counterR.vhd

```
---------------------------------------------
-- Company:     University of Birmingham
-- Engineer:    Yousef Amar
-- Create Date:   2013-10-28 20:00:00
-- Design Name:   Digital Audio Processor
-- Module Name:   counterR - Behavioral
-- Project Name:  EE4G
-- Target Devices:  xc6slx16
-- Description:   EE4G Assignment
---------------------------------------------

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

entity counterR is
  generic (N : integer);
  port (
    clock : in std_logic;
    increment : in std_logic_vector(N-1 downto 0);
    limit : in std_logic_vector(N-1 downto 0);
    count : out std_logic_vector(N-1 downto 0)
  );
end counterR;

architecture Behavioral of counterR is
  signal icount: std_logic_vector(N-1 downto 0) := (others =>
      '0');
begin
  count <= icount;

  process (clock)
    -- NOTE: No extra bit needed for sum since counter will
       never get that high anyway at these frequencies.
    variable sum : std_logic_vector(N-1 downto 0);
  begin
    if rising_edge(clock) then
      sum := icount + increment;
      if sum >= limit then
        icount <= sum - limit; -- Pseudo-modulus operation
      else
        icount <= sum;
      end if;
    end if;
  end process;
end Behavioral;
```

B.16 display.vhd

```
---------------------------------------------
-- Company:          University of Birmingham
-- Engineer:         Yousef Amar
-- Create Date:      2013-10-28 20:00:00
-- Design Name:      Digital Audio Processor
-- Module Name:      display - Behavioral
-- Project Name:     EE4G
-- Target Devices:   xc6slx16
-- Description:      EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity display is
  port (
    number : in std_logic_vector (3 downto 0);
    segs : out std_logic_vector (7 downto 0)
  );
end display;

architecture Behavioral of display is
begin

  --       0
  --      ---
  --   5 |   | 1
  --      ---   <- 6
  --   4 |   | 2
  --      ---
  --       3

  segs(7) <= '1';
  with number SELect
  segs(6 downto 0) <= "1111001" when "0001", --1
     "0100100" when "0010", --2
     "0110000" when "0011", --3
     "0011001" when "0100", --4
     "0010010" when "0101", --5
     "0000010" when "0110", --6
     "1111000" when "0111", --7
     "0000000" when "1000", --8
     "0010000" when "1001", --9
     "0001000" when "1010", --A
     "0000011" when "1011", --b
     "1000110" when "1100", --C
     "0100001" when "1101", --d
```

```
      "0000110" when "1110", --E
      "0001110" when "1111", --F
      "1000000" when others; --0

end Behavioral;
```

B.17 display4.vhd

```
---------------------------------------------
-- Company:          University of Birmingham
-- Engineer:         Yousef Amar
-- Create Date:      2013-10-28 20:00:00
-- Design Name:      Digital Audio Processor
-- Module Name:      display4 - Behavioral
-- Project Name:     EE4G
-- Target Devices:   xc6slx16
-- Description:      EE4G Assignment
---------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity display4 is
  port (
    clk : in std_logic;
    number : in std_logic_vector(15 downto 0);
    digit : out std_logic_vector (3 downto 0);
    segments : out std_logic_vector(7 downto 0)
  );
end display4;

architecture Behavioral of display4 is
  component counter
    generic (N : integer);
    port (
      clock : in std_logic;
      count : out std_logic_vector (N-1 downto 0)
    );
  end component;
  component display
    port(
      number : in std_logic_vector(3 downto 0);
      segs : out std_logic_vector(7 downto 0)
    );
  end component;

  signal inumber : std_logic_vector(3 downto 0);
  signal count : std_logic_vector(31 downto 0);
  signal crtlBits : std_logic_vector (1 downto 0);
begin

  ct1: counter generic map (N=>32) port map (clock => clk,
     count => count);
  crtlBits <= count(19 downto 18);
  dd1: display port map (number => inumber, segs => segments)
     ;

  with crtlBits select
  digit <= "1110" when "00",
         "1101" when "01",
         "1011" when "10",
         "0111" when others;

  with crtlBits select
  inumber <= number(3 downto 0) when "00",
         number(7 downto 4) when "01",
         number(11 downto 8) when "10",
         number(15 downto 12) when others;

end Behavioral;
```