A Low Cost Timing Generation Unit

Senior Project

Chris Vochoska

Professor Lynne Slivovsky

June 2016
# Table of Contents

Overview ........................................................................................................................................... 4  
Introduction......................................................................................................................................... 4  
Clients and Community Partners ......................................................................................................... 4  
Project Goal .......................................................................................................................................... 4  
Project Outcomes and Deliverables ..................................................................................................... 4  
Background .......................................................................................................................................... 5  
Functional Requirements .................................................................................................................... 6  
  Output Specifications ............................................................................................................................ 6  
  TGU Control Specifications .................................................................................................................. 8  
Final Detailed Design ........................................................................................................................... 8  
Hardware ............................................................................................................................................. 9  
  TGU Top Level ................................................................................................................................... 10  
  Clock Generation ............................................................................................................................... 10  
  TOD Generation ............................................................................................................................... 11  
  Delay Component ............................................................................................................................. 11  
  SPI Receiver ...................................................................................................................................... 12  
  FGPA Constraints ............................................................................................................................. 13  
Other Hardware Designs ...................................................................................................................... 13  
Software .............................................................................................................................................. 14  
  Arduino .............................................................................................................................................. 14  
  PC/Python Communication ................................................................................................................. 15  
Results .................................................................................................................................................. 15  
  Frequency Measurements .................................................................................................................. 15  
  TOD Signals ...................................................................................................................................... 19  
  Delay Groups .................................................................................................................................... 19  
Conclusion ............................................................................................................................................ 20  
Appendices .......................................................................................................................................... 20  
  Appendix A: FPGA VHDL Source Code .............................................................................................. 20  
  TGU_Top.vhd: ................................................................................................................................... 20  
  Tod_generator.vhd: ............................................................................................................................ 25
<table>
<thead>
<tr>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay_block.vhd</td>
<td>27</td>
</tr>
<tr>
<td>Clock_generator.vhd</td>
<td>30</td>
</tr>
<tr>
<td>SPI_receiver.vhd</td>
<td>32</td>
</tr>
<tr>
<td>Appendix B: FPGA Signal Locations</td>
<td>34</td>
</tr>
<tr>
<td>Appendix C: Arduino Code</td>
<td>35</td>
</tr>
<tr>
<td>Appendix D: Python Code</td>
<td>36</td>
</tr>
</tbody>
</table>
Overview

Introduction
A Timing Generation Unit (TGU) provides multiple clock and data signals to many individual units within a greater system where synchronized timing is critical. Its primary goal is to control of “time” and provide synchronization for larger systems where different units might physically be many meters apart. It does this by ensuring timing data is received by each unit, each unit processes said timing data within 15 ns of each other, and the required clocks are stable at each of the designated frequencies. In order for the timing and synchronization data to be processed at nearly the same time, the TGU delays the required signals by a predetermined amount of time based on an electrical distance calculation. This project designs, implements, and tests a low cost, proof of concept TGU on a Digilent Nexys4 DDR evaluation board. The design and results will be presented in the following report.

Clients and Community Partners
Northrop Grumman (NG) uses full system engineering models (EM) of satellites to verify designs, test software updates, and troubleshoot any anomalies that may occur throughout the life of a satellite. These EMs have the full “primary” capabilities of their flight versions, without the extreme specifications like radiation tolerant ASICs, thermal requirements, and the arrays of backups via redundancies.

Because Northrop Grumman contracted out the TGU’s original design, a replacement EM version is of great interest to NG for future testing. The current EM TGU is over 10 years old, reaching the end of its designed 15 year lifespan. Northrop Grumman plans on needing a fully functional EM satellite for at least 10 more years.

Project Goal
The goal of this project is to physically demonstrate the design and architecture of an EM TGU on an FPGA evaluation board to keep cost to a minimum. It will provide much of the legwork for a true EM design and implementation, should the need to build one arise. The different outputs of this design will include six different system clocks, 200MHz, 50MHz, 40MHz, 36MHz, 20MHz, and 9.6MHz, four time of day (TOD) signals, TOD clock, TOD enable, TOD data, and TOD step, and two synchronization signals, system synchronization and the system epoch. The TGU will provide these various outputs to multiple simulated receiving units and each of those will be validated with an oscilloscope.

A user will be able to send commands to the TGU to vary the delay of a group of outputs to each of the different units and change the TOD to a predetermined value.

Project Outcomes and Deliverables
The final outcomes of this project will produce a verified design template for an EM TGU design. This design will have been verified on a physical system where each of the different
Synchronization clocks and time of day step signals can all be independently delayed per receiving unit. Each signal, whether it is a system clock, time of day signal, or synchronization clock, can be independently fanned out to numerous outputs on the Nexy4 board. All of the software files, VHDL, Arduino C, and Python, will be provided with clear comments to help future designers understand what needs to be changed for the future implementation.

**Background**

A timing generation unit is one of the most critical units within a satellite. Not only does it provide numerous clocks to many different units, it provides synchronization pulses where all units receive said pulses within 28ns of each other. In this application, timing is critical because the data that is processed throughout the satellite is time sensitive as it travels through the payload. Because there are physical length differences between the different cables, the TGU needs precise control of when to send out the signals to each unit. In an effort to minimize extreme clock skew and delay values, a TGU is centrally normally located within the satellite. Figure 1 shows a cartoon drawing of what a TGU’s harness connections might look like. The differencing lengths of the lines refer to variable lengths of cables.

![Figure 1: Simulated Physical Layout of System](image)

If we assume the lengths from Figure 1 above and only Units 0, 10, and 11 have timing critical signals, the TGU would first output the critical timing signals to the longest path unit, Unit 11, delay by a predetermined amount, then output to Unit 10, delay again, and finally send the signal to Unit 0. Figure 2 below shows what the signals might look like from the different perspectives. Note that the signals will not be perfectly aligned at the different units because each signal is shifted by discrete steps.
**Functional Requirements**

The overall goal is to provide a low-cost proof-of-concept EM TGU design for future implementation, should Northrop Grumman require one. This means that this project’s specifications shall mimic the specifications of the real TGU used on the satellite.

**Output Specifications**

The TGU shall provide multiple clocks, timing signals, and synchronization pulses to multiple receiving units. Table 1 shows the output summary of the TGU including the number of signals and their corresponding destination group; the destination groups refer to different units within the system that receive each type of signal.

<table>
<thead>
<tr>
<th>TGU Output Signals</th>
<th>Number of Signals</th>
<th>Destination Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOD (clk, en, data, step)</td>
<td>5 of each (20 total)</td>
<td>0, 1, 2, 6, 11</td>
</tr>
<tr>
<td>Synchronization</td>
<td>7</td>
<td>1, 3, 4, 6, 7, 8, 11</td>
</tr>
<tr>
<td>Epoch</td>
<td>2</td>
<td>10, 11</td>
</tr>
<tr>
<td>200 MHz</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50 MHz</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40 MHz</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>36 MHz</td>
<td>2</td>
<td>0, 2</td>
</tr>
<tr>
<td>20 MHz</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9.6 MHz</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Debug signals</td>
<td>15</td>
<td>On-board LEDs</td>
</tr>
</tbody>
</table>

**Output Clocks**

The five different clocks shall have frequencies including 200MHz, 50MHz, 40MHz, 36MHz, and 9.6MHz. Each of the different clocks shall be a square wave with a 50% duty cycle.
None of the clocks are required to be delayed by any amount.

**Time of Day Signals**
The Time of Day signals provide the different units with the current system time. These shall include a TOD clock, TOD enable, TOD data, and TOD step and each are defined below.

- TOD Clock shall be a continuous 1MHz 50% duty cycle square wave.
- TOD Enable is an active low pulse that has a frequency of 1 second with a high pulse width of 19.68ms. This signal is low during serial transmission of TOD data.
- TOD Data is an NRZ encoded bits used to transmit a 32 bit word to different units. This word gives the current time to every unit.
- TOD Step shall be a continuous 50Hz 50% duty cycle square wave. This signal shall be able to be delayed by up to 127 steps of 27.7ns (up to 3.53us).

The TOD clock, data, and enable lines are used to transfer the system time a unit. The enable line goes low to signal the TOD data bits are being transmitted and remains low until the last bit is sent. The TOD step’s rising edge activates the word previously sent. Only one TOD transfer can occur within a step period. See Figure 3 for timing relationship during transmission.

![Figure 3: TOD Transfer Timing Diagram](image)

TOD data is a 32 bit word that defines the current system time. Each bit represents 20ms. This means that the system can count up to just over 2.73 years (or 85,899,345.92 seconds).

**Synchronization Signals**
There are two different synchronization signals the TGU is required to output, system synchronization and the system epoch. Both of these signals are 50% duty cycle square waves, with the system synchronization having a frequency of 5kHz and the system epoch having a frequency of 1Hz.
These two synchronization signals shall be able to be delayed by up to 127 steps of 27.7ns (up to 3.53us).

**Adjustable Delayed Signals**
The TOD step, system synchronization, and system epoch shall be able to be delayed per individual group number by 128 total steps, counting 0 delay as step 1, with each step less than 28ns. All signals within a group shall be delayed by the same amount.

**TGU Control Specifications**
The TGU shall be capable of receiving commands from an external source. The minimum commands shall allow alterations of the current system time, down to the second, and change the delay value of the individual groups, which is adjustable in 27.8ns steps, for a total of 128 steps.

The system shall provide an output of the command received so the user can verify the correct command data was received by the TGU.

**Final Detailed Design**
The TGU can be broken down into three different components: input commanding, clock/signal generation, signal fan-out and delay. The best choice to achieve the above specifications was to implement all of these using a hardware design on an FPGA. Only the user commanding will need to have external hardware and software to interact with the FGPA. The overall system, shown in Figure 4, includes a Nexys4 DDR, which is the TGU, an Arduino Uno, and any computer running Python 2.7 with the PySerial package. The switches and LED’s on the evaluation board are used for debugging and SPI command verification.

![Figure 4: System Overview](image)
Hardware

Most of the project was designed in hardware on the Nexy4 DDR Evaluation Board which houses a Xilinx Artix-7 FPGA. Figure 5 shows the top level schematic produced by Xilinx’s Vivado. There are four components inside the top module, an SPI receiver, a clock generator, a TOD generator, and a delay block. Each of these will be discussed further in this section.

Figure 5: TGU Top Schematic
**TGU Top Level**

The only real logic in the top module is a mux, named ‘data_received’. A more appropriate name might be ‘LED output’ because its purpose is to output a directed signal to the board’s sixteen LEDs. Because there are only 16 LEDs on the board, if the data bus going in is longer than 16 bits, like the TOD or the SPI input, they are split between switch 0 being in position 0, for the lower bits (15 : 0), and position 1 for the upper bits (31 : 16). Currently, only two telemetries are integrated, however it is very easy to add more should the need arise. The select lines for the mux are driving by the 16 switches on the board. There is a possibility of housing 65,536 sixteen bit outputs, or 32,768 32 bit outputs, more than enough for any telemetries required.

**Clock Generation**

The clock generation component’s goal is to generate all of the different clocks and synchronization signals for the entire system. These signals include: 200MHz, 50MHz, 40MHz, 36MHz, 20MHz, 9.6MHz, system synchronization (5kHz), and system epoch (1Hz). The schematic for the component is shown in Figure 6 below.

![Figure 6: Schematic of Clock Generation Component](image)

To generate the higher frequency clock signals, Xilinx IP core “Clocking Wizard v5.1” sets up different phase locked loops to generate all of the clocks. A simple clock dividing circuit wasn’t used to generate the lower frequency clocks because some of them were not divided evenly into the 100MHz input. Because of this, the design is to use three PLLs to generate all of the different clocks. The first PLL outputs 200MHz, 50MHz, 40MHz, and 20MHz. The 36MHz and the 9.6MHz couldn’t be added to this IP without degradation of some of the frequencies. Those two clocks are generated via two more PLLs. After synthesis, FPGA PLL usage is at 50%.

The other signals are built via simple counters because the PLL’s minimum output frequency is 5MHz. As they are both divisible by the 100MHz input clock, this was not a problem and they are able to retain the 50% duty cycle criteria.
TOD Generation
The TOD generation component took in the system 100MHz clock and generated all of the TOD signals and their transfer protocol. Generating the TOD clock and the TOD step were simple counters, the clock being 1MHz and the step being 50Hz.

The bulk of the design was in the transmission waveform logic based on the 1MHz clock. A counter was used to count the number of clock cycles from 0 to 20,000, 20,000 steps of 1MHz clocks gives 20ms, the TOD Step frequency. This decision was done to ensure the rising edge of the TOD Step was at least 10μs away from the TOD enable’s rising or falling edge. The logic for the transmission can be found below in Table 2.

<table>
<thead>
<tr>
<th>TOD Counter Value</th>
<th>Actions Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TOD Step = 0&lt;br&gt;TOD Enable = 1&lt;br&gt;TOD Data = TOD_time[TOD Counter Value]</td>
</tr>
<tr>
<td>1 - 31</td>
<td>TOD Data = TOD_time[TOD Counter Value]</td>
</tr>
<tr>
<td>32</td>
<td>TOD Enable = 0</td>
</tr>
<tr>
<td>10,000</td>
<td>TOD Step = 1&lt;br&gt;TOD Time = TOD Time + 1</td>
</tr>
<tr>
<td>19,999</td>
<td>TOD Counter = 0</td>
</tr>
</tbody>
</table>

This is the basics of the TOD Generator, however one of the requirements is to allow for the user to set the TOD Time to anything. Therefore, there is a 32 bit input for a new TOD Time and a 1 bit, 1 clock length flag to tell the component if the 32 bit word has been updated. As a precaution, the hardware won’t stop this transmission or update the TOD Time until all of the bits have been sent. To enable this functionality, when the input is now valid flag tells the component that the data is valid, it in turn sets its own internal ‘update TOD’ flag to update the TOD time at the beginning of a transmission, but before the enable pin is set high. At the same time the TOD time bus is updated, set another flag to tell the previous process that the new TOD has been written and that the ‘update TOD’ flag can be set back to zero.

Delay Component
The delay component was the main reason an FPGA was selected for the job. It delays all of the critical signals using a 128 bit shift register, and the different output groups pick which bit they want to pull from. Table 3 shows a code snippet of how this is achieved with relative ease.

```
-- Registers for the different values of the different groups
signal reg_step : STD_LOGIC_VECTOR (127 downto 0) := (others => '1');
delays: process(i_clk_36MHz) begin
  if (rising_edge(i_clk_36MHz)) then
    reg_step(127 downto 0) <= reg_step (126 downto 0) & s_tod_step(2);
```
Before each of the data falls into the shift register, however, the signals need to cross clock domains from 100MHz, to 36MHz to follow the timing specification. To ensure no signals go into a metastable state, three synchronization registers are used as shown in Figure 7. By the time the third register outputs the signal there is almost no chance the signal will be metastable.

And finally, one of the constraints is to allow the user to send a command to change the delay of different output groups. Again, two of the inputs of the component are the outputs of the SPI Receiver component. When the data is valid flag pulses high, the header of the word is checked for a delay command. If a delay command is seen, then the delay group is selected and the new delay value is stored in the appropriate register. In the code example above in table 3, the o_0_del register would be updated to a new value and everything in the group would pull from the new input of the muxed shift register.

**SPI Receiver**

Finally, the SPI receiver circuit takes in an SPI signal from an external source. For this project, the external source was an Arduino board because it is easily able to create a program to take in a PC’s input from a user and translate it into a data format for an SPI interface.
The trick to the SPI receiver circuit in this instance is that the FPGA is the slave in the SPI world. In many ways, this makes the programming much easier, if the variable external clock can be figured out. The workaround for this was to create a signal that mimicked the “rising edge” of the inbound external clock. It assumes that the input clock is less than one sixth of the master clock. For this design, the transfer rate was set to 250kHz, which is obviously much less than 100MHz. By over sampling the input clock, the circuit can have a 3-bit shift register that continually samples at the 100MHz rate. When bits 1 and 0 are high, and bit 2 is low, then it pulses a rising edge flag. The same can be done with a falling edge if required. With this idea, the receive circuit implements two flags for rising edge of the clock and the rising edge of the enable pin. While the enable pin is held low and the rising edge of the clock is triggered, the 32 bit data received register takes in the new sample and shifts everything left by one bit.

This ends when the circuit detects a rising edge of the enable pin. When this occurs, a single pulse on the 100MHz clock is sent out to tell all the other components that the data is now valid.

**FGPA Constraints**

Defining the constraints file for this project was a new experience. Not only did the inputs and outputs have to get assigned pins, but the slew rates needed to be set for the clocks, and in order to meet the timing requirement of the DRC, the inputs needed to set hold times the correct hold times and the cross-over of the two used asynchronous clocks, 100MHz and 36MHz, needed to be ignored. In Xilinx’s Vivado, the constraints file is now defined by Tcl commands, unlike the older ISE Project Navigator form. This opens up many different possibilities, however also creates a steep learning curve. See Appendix A for the constraints file and a summary of final inputs and outputs.

**Other Hardware Designs**

As stated in the SPI section, an Arduino Uno was used to translate between the PC commands.
and the SPI receiving interface. The Arduino uses a USB connection to the computer and three output pins which drive the SPI_clk, SPI_SS (enable), and SPI_data signals.

The pin mapping for the Arduino to Nexys4 DDR is shown in Table 4 below.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Arduino Pin</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock (250kHz)</td>
<td>13</td>
<td>Pmod JD Pin 1</td>
</tr>
<tr>
<td>Data</td>
<td>11</td>
<td>Pmod JD Pin 2</td>
</tr>
<tr>
<td>Enable</td>
<td>10</td>
<td>Pmod JD Pin 3</td>
</tr>
</tbody>
</table>

**Software**

Most of the software in the project is in the external components used to communicate to the TGU. The Arduino has a simple code and the python script used to communicate with the Arduino board is a simple connection to a communication port on the PC. Below are described in detail below.

**Arduino**

The Arduino code for the project was used to send an SPI signal to the FPGA translated from a numeric string. Below is a flow diagram of the code.

![Arduino Software Flow Diagram](image-url)

*Figure 9: Arduino Software Flow Diagram*
PC/Python Communication
Python was chosen for the end user for two main reasons: because it is very easy to connect to an Arduino using the PySerial library, and it is extremely easy to convert between integers, strings, hex strings, etc. If a larger initialization script will be used in the future, python is the perfect tool to do such a thing.

Python 2.7 is needed to run the code used in Table 5 below.

Table 5: Python Program to Send Data to the Arduino

```python
import serial
import time

# My computer (Windows) uses 'COM4' for Arduino
ser = serial.Serial('COM4', 9600, timeout=0)

# Continue until the user enters 'exit'
while True:
    input = raw_input('>> ')
    if input == 'exit':
        ser.close()
        exit()
    else:
        print 'Sending: %d (decimal) ' % int(input, 16)
        ser.write('%d % int(input, 16))

        # Give time for device to process data
        time.sleep(0.5)

        # Read output from device
        out = ''
        while ser.inWaiting() > 0: out += ser.read(1)
        if out != '': print out
```

Results
Frequency Measurements
After the system was fully synthesized and compiled, it was time to test all of the different functionalities from the requirements section above.

The first test was to measure the system’s synchronization/epoch frequencies on an oscilloscope. Again, they matched as expected. The system synchronization signals were right at 5kHz and the epoch showed 1Hz.

Measuring the clocks resulted in less than ideal results. On an oscilloscope, it was apparent that the FPGA was trying to output the correct clocks at the expected frequencies, however the faster the frequency, the smaller the “waveform”. Ultimately, at the 200MHz signal, it looked like some 5mV noise might have been the 200MHz signal coming through. Even more interesting,
when a 50 ohm resistor was placed across the correct output and the ground pin, the signal showed a slightly worst signal. Obviously this wasn’t acceptable and would mean that a redesign would be needed.

After looking through the schematics of the FGPA board itself, the problem became apparent. Figure 10 below shows the schematic from an output of the FPGA to the final output of the board. This schematic is pulled from the Digilent website. There is a 200 ohm safety resistor in series with the Pmod output. The signal simply wasn’t being terminated correctly.

The new test to validate the suspicion was to use an oscilloscope’s internal 50 ohm impedance as the parallel resistor. For every clock signal that was being measured, a shorting wire was placed between the corresponding pin at the Pmod connector and the ground pin. The new circuit is shown in Figure 11.
Now, after probing across the resistor a valid sine wave at 200MHz, ~1V_{pk-pk} was present on the oscilloscope. See Figure 12 below.

![Figure 12: Oscilloscope Capture of the 200MHz Clock](image)

The rest of the clock signals were tested and all showed similar results, meaning the clock signals appeared to be being generated as expected. Figure 13 and Figure 14 below show the oscilloscope screen prints of the 50MHz and 40MHz clocks.

![Figure 13: 50 MHz Clock Oscilloscope Capture](image)
Other than the 200MHz signal, there was a ton of ringing on the signals, especially as the frequency slowed down, 9.6MHz showing the most. This is due to the test setup and the grossly mismatched impedances. Figure 15 shows the consequence of the mismatched impedance on the 20MHz clock.

From the above data, I can conclude that the FPGA is sending out the correct frequencies and signal levels, however the Nexys4 DDR board layout isn’t built for high frequency outputs. If
this project is going to be put to use, an new printed circuit board would need to be designed with the correct impedance in mind.

**TOD Signals**

After the verification of the clocks, next was to verify all of the different outputs expected of the TGU. The TOD clock is indeed a 1MHz square-wave and the rest of the signals match the transmission requirements. The TOD Step rising edge was measured to be ~10ms after the rising edge of the enable signal. The data was incrementally counting once per step. This was measured over a 5 hour period and the bits matched accordingly. To verify the system doesn’t crash during rollover (when all bits transition from all ‘1’s to all ‘0’s), a jam set command was sent to the end of the 2.7 year period. All bits functioned as expected.

**Delay Groups**

The other component was to be able to delay the system synchronization, system epoch, and TOD step signals by up to a total of 128 total 27.8 ns steps.

Groups 0, 1, and 2 all have a TOD Step signal being output. All three signals were shown on an oscilloscope and with zero delay, they were lined up almost exactly (within picoseconds). Figure 16 below shows the three signals with different delay values: group 0 having no delay, group 2 having the max delay, and group 1 in between. The delay between the rising edge of group 0 and the rising edge of group 2 can be seen as 3.528us, which is as expected (2.77 x 127 = 3.5277us).

![Figure 16: Three Different TOD Steps with Channel 3 Showing the Maximum Delay](image)

Page 19 of 37
Conclusion
With the provided design, Northrop Grumman has the foundation to quickly build an EM TGU if the need should arise. The specifications were met; however there was an impedance problem on the higher frequency signals. This will be engineered away with a new PCB when the new design is created. The board was able to be commanded via a scripting language, and thus could be autonomously set up should the need arise. The critical timing signals are able to be delayed appropriately. To alter the frequencies and number of output signals or what groups the output signals are sent to is trivial and should take minimal effort. Most of the future work will be designing a new PCB.

For under $200, a successful proof of concept TGU has been accomplished.

Appendices

Appendix A: FPGA VHDL Source Code

TGU_Top.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TGU_Top is
  Port ( i_clk_100MHz : in STD_LOGIC;
         SW : in STD_LOGIC VECTOR(15 downto 0);
         spi_clk : in STD_LOGIC; -- SPI input clock
         spi_data_in : in STD_LOGIC; -- SPI data input
         spi_en : in STD_LOGIC; -- SPI Enable (active low)
         spi_data_out : out STD_LOGIC; -- SPI data output
         data_received : out STD_LOGIC VECTOR(15 downto 0); -- SPI data
         o_0_tod_clk : out STD_LOGIC;
         o_0_tod_en : out STD_LOGIC;
         o_0_tod_data : out STD_LOGIC;
         o_0_tod_step : out STD_LOGIC;

end TGU_Top;
```
o_0_clk_40MHz : out STD_LOGIC;
o_0_clk_36MHz : out STD_LOGIC;
o_0_clk_20MHz : out STD_LOGIC;

-- Group 1
o_1_tod_clk : out STD_LOGIC;
o_1_tod_en : out STD_LOGIC;
o_1_tod_data : out STD_LOGIC;
o_1_tod_step : out STD_LOGIC;
o_1_sync : out STD_LOGIC;
o_1_clk_200MHz : out STD_LOGIC;
o_1_clk_36MHz : out STD_LOGIC;

-- Group 2
o_2_tod_clk : out STD_LOGIC;
o_2_tod_en : out STD_LOGIC;
o_2_tod_data : out STD_LOGIC;
o_2_tod_step : out STD_LOGIC;
o_2_clk_36MHz : out STD_LOGIC;
o_2_clk_9_6MHz : out STD_LOGIC;

-- Group 3
o_3_sync : out STD_LOGIC;

-- Group 4
o_4_sync : out STD_LOGIC;

-- Group 6
o_6_tod_clk : out STD_LOGIC;
o_6_tod_en : out STD_LOGIC;
o_6_tod_data : out STD_LOGIC;
o_6_tod_step : out STD_LOGIC;
o_6_sync : out STD_LOGIC;

-- Group 7
o_7_sync : out STD_LOGIC;

-- Group 8
o_8_sync : out STD_LOGIC;

-- Group 10
o_10_epoch : out STD_LOGIC;

-- Group 11
o_11_tod_clk : out STD_LOGIC;
o_11_tod_en : out STD_LOGIC;
o_11_tod_data : out STD_LOGIC;
o_11_tod_step : out STD_LOGIC;
o_11_sync : out STD_LOGIC;
o_11_epoch : out STD_LOGIC);
end TGU_Top;

architecture Behavioral of TGU_Top is

-- Component Declarations
-- TOD Generator Unit
component tod_generator

port (
  i_clk_100MHz : in STD_LOGIC;
  data_in : in STD_LOGIC_VECTOR(31 downto 0);
  data_valid : in STD_LOGIC;
  o_tod_clk_q : out STD_LOGIC; -- 1 MHz square wave
  o_tod_en_q : out STD_LOGIC; -- Pulse (f = 50 Hz, pulse width = 32 us)
  o_tod_step_q : out STD_LOGIC; -- 50 Hz square wave
  o_tod_data_q : out STD_LOGIC;
  tod_word : out STD_LOGIC_VECTOR(31 downto 0)); -- Data line, 32 bits of data
end component;

-- Clock Generation Unit
component clock_generator
port (
  i_clk_100MHz : in STD_LOGIC;
  clk_200MHz : out STD_LOGIC;
  clk_50MHz : out STD_LOGIC;
  clk_40MHz : out STD_LOGIC;
  clk_36MHz : out STD_LOGIC;
  clk_20MHz : out STD_LOGIC;
  clk_9_6MHz : out STD_LOGIC;
  sync : out STD_LOGIC;
  epoch : out STD_LOGIC);
end component;

component SPI_receiver
port (
  i_clk_100MHz : in STD_LOGIC; -- System clock
  sclk : in STD_LOGIC; -- SPI input clock
  data_in : in STD_LOGIC; -- SPI data input
  SS : in STD_LOGIC; -- SPI Enable (active low)
  data_out : out STD_LOGIC; -- SPI data output
  data_valid : out STD_LOGIC; -- Output data is valid
  data_received : out STD_LOGIC_VECTOR(31 downto 0));
end component;

component delay_block
port (i_clk_36MHz : in STD_LOGIC;
i_clk_100MHz : in STD_LOGIC;
i_tod_step : in STD_LOGIC;
i_sync : in STD_LOGIC;
i_epoch : in STD_LOGIC;
data_in : in STD_LOGIC_VECTOR(31 downto 0);
data_valid : in STD_LOGIC;
o_0_tod_step : out STD_LOGIC;
o_1_tod_step : out STD_LOGIC;
o_1_sync : out STD_LOGIC;
o_2_tod_step : out STD_LOGIC;
o_3_sync : out STD_LOGIC;
o_4_sync : out STD_LOGIC;
o_6_tod_step : out STD_LOGIC;
o_6_sync : out STD_LOGIC;
o_7_sync : out STD_LOGIC;
o_8_sync : out STD_LOGIC;)
end component;
o_10_epoch : out STD_LOGIC;
o_11_tod_step : out STD_LOGIC;
o_11_sync : out STD_LOGIC;
o_11_epoch : out STD_LOGIC);

end component;

signal s_data_rec : STD_LOGIC_VECTOR (31 downto 0);
signal s_data_valid : STD_LOGIC;
signal s_tod_word : STD_LOGIC_VECTOR(31 downto 0);

-- clock signals
signal s_clk_200MHz : STD_LOGIC;
signal s_clk_50MHz : STD_LOGIC;
signal s_clk_40MHz : STD_LOGIC;
signal s_clk_36MHz : STD_LOGIC;
signal s_clk_20MHz : STD_LOGIC;
signal s_clk_9_6MHz : STD_LOGIC;

-- Delay signals
signal s_tod_step : STD_LOGIC;
signal s_sync : STD_LOGIC;
signal s_epoch : STD_LOGIC;

-- TOD signals
signal s_tod_clk : STD_LOGIC;
signal s_tod_en : STD_LOGIC;
signal s_tod_data : STD_LOGIC;

begin

TOD_word: tod_generator PORT MAP ( 
  i_clk_100MHz => i_clk_100MHz, 
  data_in => s_data_rec, 
  data_valid => s_data_valid, 
  o_tod_clk_q => s_tod_clk, 
  o_tod_en_q => s_tod_en, 
  o_tod_step_q => s_tod_step, 
  o_tod_data_q => s_tod_data, 
  tod_word => s_tod_word);

clocks: clock_generator PORT MAP ( 
  i_clk_100MHz => i_clk_100MHz, 
  clk_200MHz => s_clk_200MHz, 
  clk_50MHz => s_clk_50MHz, 
  clk_40MHz => s_clk_40MHz, 
  clk_36MHz => s_clk_36MHz, 
  clk_20MHz => s_clk_20MHz, 
  clk_9_6MHz => s_clk_9_6MHz, 
  sync => s_sync, 
  epoch => s_epoch);

SPI_inputs: SPI_receiver PORT MAP ( 
  i_clk_100MHz => i_clk_100MHz, 
  sclk => spi_clk, 
  SS => spi_en, 
  data_in => spi_data_in, 
  data_out => spi_data_out,
data_received => s_data_rec,
data_valid => s_data_valid;

Delay: delay_block PORT MAP (  
i_clk_36MHz => s_clk_36MHz,
i_clk_100MHz => i_clk_100MHz,
i_tod_step => s_tod_step,
i_sync => s_sync,
i_epoch => s_epoch,
data_in => s_data_rec,
data_valid => s_data_valid,
o_0_tod_step => o_0_tod_step,
o_1_tod_step => o_1_tod_step,
o_1_sync => o_1_sync,
o_2_tod_step => o_2_tod_step,
o_3_sync => o_3_sync,
o_4_sync => o_4_sync,
o_6_tod_step => o_6_tod_step,
o_6_sync => o_6_sync,
o_7_sync => o_7_sync,
o_8_sync => o_8_sync,
o_10_epoch => o_10_epoch,
o_11_tod_step => o_11_tod_step,
o_11_sync => o_11_sync,
o_11_epoch => o_11_epoch);

o_0_tod_clk <= s_tod_clk; -- Group 0
o_0_tod_en <= s_tod_en;
o_0_tod_data <= s_tod_data;
o_0_clk_40MHz <= s_clk_40MHz;
o_0_clk_36MHz <= s_clk_36MHz;
o_0_clk_20MHz <= s_clk_20MHz;
o_1_tod_clk <= s_tod_clk; -- Group 1
o_1_tod_en <= s_tod_en;
o_1_tod_data <= s_tod_data;
o_1_clk_200MHz <= s_clk_200MHz;
o_1_clk_36MHz <= s_clk_36MHz;
o_2_tod_clk <= s_tod_clk; -- Group 2
o_2_tod_en <= s_tod_en;
o_2_tod_data <= s_tod_data;
o_2_clk_36MHz <= s_clk_36MHz;
o_2_clk_9_6MHz <= s_clk_9_6MHz;
o_6_tod_clk <= s_tod_clk; -- Group 6
o_6_tod_en <= s_tod_en;
o_6_tod_data <= s_tod_data;
o_11_tod_clk <= s_tod_clk; -- Group 11
o_11_tod_en <= s_tod_en;
o_11_tod_data <= s_tod_data;

-- Debug Section
-- Allow for 32 bits on the LED's
with SW select data_received <=
s_data_rec(15 downto 0) when x"0000",
s_data_rec(31 downto 16) when x"0001",
s_tod_word(15 downto 0) when x"0002",
s_tod_word(31 downto 16) when x"0003",
x"FFFF" when others;
end Behavioral;

Tod_generator.vhd:

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

entity tod_generator is
  Port ( i_clk_100MHz : in STD_LOGIC;
         data_in : in STD_LOGIC_VECTOR(31 downto 0);
         data_valid : in STD_LOGIC;
         o_tod_clk_q : out STD_LOGIC; -- 1 MHz square wave
         o_tod_step_q : out STD_LOGIC; -- 50 Hz square wave
         o_tod_en_q : out STD_LOGIC; -- Pulse (f = 50 Hz, pulse width = 32 us)
         o_tod_data_q : out STD_LOGIC; -- Data line, 32 bits of data
         tod_word : out STD_LOGIC_VECTOR(31 downto 0));
end tod_generator;

architecture Behavioral of tod_generator is
  -- TOD signals
  signal s_tod_clk : STD_LOGIC := '0';
  signal s_tod_data : STD_LOGIC := '0';
  signal s_tod_step : STD_LOGIC := '0';
  signal s_tod_en : STD_LOGIC := '1';

  signal s_tod_time : UNSIGNED(31 downto 0) := (others => '0');

  -- Constants
  constant clk_2MHz : integer := 100e6 / 2e6;
  constant step_period : integer := 20e3; -- In microseconds
signal s_tod_counter : integer := 0;

-- Create a 2MHz clock so we can get a 50% duty cycle 1MHz clock
-- using gated clock logic (takes 6 bits)
signal clk_2MHz_enable : STD_LOGIC := '0';
signal clk_2MHz_counter : UNSIGNED (5 downto 0) := (others => '0');

-- Data input signals
signal s_data_valid : STD_LOGIC := '0';
signal s_data_written : STD_LOGIC := '0';

begin

generate_tod_word:
  process (i_clk_100MHz)
  begin
    if (rising_edge(i_clk_100MHz)) then
      -- TOD 1 MHz clock
      if (clk_2MHz_enable = '1') then
        s_tod_clk <= not(s_tod_clk);
      end if;

      -- TOD Data sequence (based on 'gated' 1 MHz clock)
      if (clk_2MHz_enable = '1' and s_tod_clk = '0') then
        s_tod_counter <= s_tod_counter + 1;
      end if;

      s_tod_data <= s_tod_time(s_tod_counter);

      elsif (s_tod_counter = 0) then
        s_tod_data <= s_tod_time(s_tod_counter);
      elsif (s_tod_counter < 32) then
        s_tod_data <= s_tod_time(s_tod_counter);
      elsif (s_tod_counter = 32) then
        s_tod_en <= '0';
      elsif (s_tod_counter = (step_period / 2)) then
        s_tod_en <= '0';
      elsif (s_tod_counter = (step_period - 1)) then
        s_tod_counter <= 0;
      end if;

    end if;
  end process generate_tod_word;

clock_enables:
  process (i_clk_100MHz)
  begin
    if (rising_edge(i_clk_100MHz)) then
      -- 2 MHz signal
      if (TO_INTEGER(clk_2MHz_counter) = (clk_2MHz - 1)) then
        clk_2MHz_counter <= (others => '0');
      end if;
    end if;
  end process clock_enables;
'0'
    clk_2MHz_enable <= '1';          -- Set clock pulse
else
    clk_2MHz_counter <= clk_2MHz_counter + 1;
    clk_2MHz_enable <= '0';          -- Reset clock pulse
end if;
end if;
end process clock_enables;

-- Now output registered TOD values
o_tod_clk_q <= s_tod_clk;
o_tod_en_q <= s_tod_en;
o_tod_step_q <= s_tod_step;
o_tod_data_q <= s_tod_data;

-- Data Input handling
process(i_clk_100MHz) begin
    if(rising_edge(i_clk_100MHz)) then
        if(data_valid = '1' and data_in(31 downto 27) = b"00010") then
            s_data_valid <= '1';          -- New TOD needs to get injected
        elsif(s_data_written = '1') then
            s_data_valid <= '0';          -- New data has been written, no
                                                -- longer flag
        end if;
    end if;
end process;

-- Debug stuff
tod_word <= STD_LOGIC_VECTOR(s_tod_time);
end Behavioral;

Delay_block.vhd
-----------------------------------------------------------------------------
-----
-- Company:       Cal Poly San Luis Obispo
-- Engineer:      Christopher Vochoska
--
-- Create Date:   05/20/2016 08:16:13 PM
-- Design Name:   TGU_Simulator
-- Module Name:   delay_block - Behavioral
-- Project Name:  TGU_Simulator
-- Target Devices: Nexys4 DDR
-- Tool Versions: Vivado 2015.3
-- Description:   This module fans out the different outputs to the
--                 corresponding output group, and delays the timing
--                 critical
--
-- Revision: 1.00
-- Revision 0.01 - File Created
-----------------------------------------------------------------------------
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- arithmetic functions with Signed or Unsigned values
use IEEE.Numeric_STD.ALL;

entity delay_block is
  Port ( i_clk_36MHz : in STD_LOGIC;
         i_clk_100MHz : in STD_LOGIC;
         i_tod_step : in STD_LOGIC;
         i_sync : in STD_LOGIC;
         i_epoch : in STD_LOGIC;
         data_in : in STD_LOGIC_VECTOR (31 downto 0);
         data_valid : in STD_LOGIC;
         o_0_tod_step : out STD_LOGIC;
         o_1_tod_step : out STD_LOGIC;
         o_1_sync : out STD_LOGIC;
         o_2_tod_step : out STD_LOGIC;
         o_3_sync : out STD_LOGIC;
         o_4_sync : out STD_LOGIC;
         o_6_tod_step : out STD_LOGIC;
         o_6_sync : out STD_LOGIC;
         o_7_sync : out STD_LOGIC;
         o_8_sync : out STD_LOGIC;
         o_10_epoch : out STD_LOGIC;
         o_11_tod_step : out STD_LOGIC;
         o_11_sync : out STD_LOGIC;
         o_11_epoch : out STD_LOGIC);
end delay_block;

architecture Behavioral of delay_block is

  -- Synchronization signals
  signal s_tod_step : STD_LOGIC_VECTOR (2 downto 0) := (others => '0');
  signal s_sync : STD_LOGIC_VECTOR (2 downto 0) := (others => '0');
  signal s_epoch : STD_LOGIC_VECTOR (2 downto 0) := (others => '0');

  -- Delays for the different groups
  signal o_0_del : INTEGER := 0;
  signal o_1_del : INTEGER := 0;
  signal o_2_del : INTEGER := 0;
  signal o_3_del : INTEGER := 0;
  signal o_4_del : INTEGER := 0;
  signal o_6_del : INTEGER := 0;
  signal o_7_del : INTEGER := 0;
  signal o_8_del : INTEGER := 0;
  signal o_10_del : INTEGER := 0;
  signal o_11_del : INTEGER := 0;

  -- Registers for the different values of the different groups
  signal reg_step : STD_LOGIC_VECTOR (127 downto 0) := (others => '1');
  signal reg_sync : STD_LOGIC_VECTOR (127 downto 0) := (others => '1');
  signal reg_epoch : STD_LOGIC_VECTOR (127 downto 0) := (others => '1');

  -- Data input
  signal del_group : INTEGER := 0;
  signal delay : INTEGER := 0;

begin
-- Cross the 100MHz clock domain to the 36MHz domain
sync: process(i_clk_36MHz) begin
  if (rising_edge(i_clk_36MHz)) then
    s_tod_step <= s_tod_step(1 downto 0) & i_tod_step;
    s_sync <= s_sync(1 downto 0) & i_sync;
    s_epoch <= s_epoch(1 downto 0) & i_epoch;
  end if;
end process sync;
-- Create the 128 different delay values to pull from
delays: process(i_clk_36MHz) begin
  if (rising_edge(i_clk_36MHz)) then
    reg_step(127 downto 0) <= reg_step(126 downto 0) & s_tod_step(2);
    reg_sync(127 downto 0) <= reg_sync(126 downto 0) & s_sync(2);
    reg_epoch(127 downto 0) <= reg_epoch(126 downto 0) & s_epoch(2);
  end if;
end process delays;
-- Each signal is getting it's corresponding delay value
o_0_tod_step <= reg_step(o_0_del);
o_1_tod_step <= reg_step(o_1_del);
o_1_sync <= reg_sync(o_1_del);
o_2_tod_step <= reg_step(o_2_del);
o_3_sync <= reg_sync(o_3_del);
o_4_sync <= reg_sync(o_4_del);
o_6_tod_step <= reg_step(o_6_del);
o_6_sync <= reg_sync(o_6_del);
o_7_sync <= reg_sync(o_7_del);
o_8_sync <= reg_sync(o_8_del);
o_10_epoch <= reg_epoch(o_10_del);
o_11_tod_step <= reg_step(o_11_del);
o_11_sync <= reg_sync(o_11_del);
o_11_epoch <= reg_epoch(o_11_del);
-- Data Input Handling
-- Word | 15 14 13 12 | 11 10 09 08 | 07 06 05 04 | 03 02 01 00
--------
-- 00 | Command Type: 1 | Spare | Group Number
--------
-- 01 | Spare | Delay Value
--------
del_group <= TO_INTEGER(UNSIGNED(data_in(23 downto 16)));
delay <= TO_INTEGER(UNSIGNED(data_in(7 downto 0)));

process(i_clk_100MHz) begin
  if(rising_edge(i_clk_100MHz)) then
    if(data_valid = '1' and data_in(31 downto 27) = b"00001") then
      if(del_group = 0) then
        o_0_del <= delay;
      elsif(del_group = 1) then
        o_1_del <= delay;
      end if;
    end if;
  end if;
end process;
o_1_del <= delay;
elsif(del_group = 2) then
  o_2_del <= delay;
elsif(del_group = 3) then
  o_3_del <= delay;
elsif(del_group = 4) then
  o_4_del <= delay;
elsif(del_group = 5) then
  o_5_del <= delay;
elsif(del_group = 6) then
  o_6_del <= delay;
elsif(del_group = 7) then
  o_7_del <= delay;
elsif(del_group = 8) then
  o_8_del <= delay;
elsif(del_group = 9) then
  o_9_del <= delay;
elsif(del_group = 10) then
  o_10_del <= delay;
elsif(del_group = 11) then
  o_11_del <= delay;
end if;
end if;
end if;
end process;
end Behavioral;

-- Company: Cal Poly San Luis Obispo
-- Engineer: Christopher Vochoska
-- Create Date: 04/12/2016 08:10:44 PM
-- Design Name: TGU_Simulator
-- Module Name: clock_generator - Behavioral
-- Project Name: TGU_Simulator
-- Target Devices: Nexys4 DDR
-- Tool Versions: Vivado 2015.3
-- Description: This module is used to generate all of the clocks for the TGU
-- Revision: 1.00
-- Revision 0.01 - File Created

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity clock_generator is
  Port ( i_clk_100MHz : in STD_LOGIC;
         clk_200MHz : out STD_LOGIC;
         clk_50MHz : out STD_LOGIC;
         clk_40MHz : out STD_LOGIC;
         clk_36MHz : out STD_LOGIC;
         clk_20MHz : out STD_LOGIC;
  );
clk_9_6MHz : out STD_LOGIC;
sync : out STD_LOGIC;
epoch : out STD_LOGIC);
end clock_generator;

architecture Behavioral of clock_generator is

-- Component Declarations
-- First set of clocks (200, 50, 40, 20)
component clk_wiz_0
port (i_clk_100MHz : in STD_LOGIC;
clk_200MHz : out STD_LOGIC;
clk_50MHz : out STD_LOGIC;
clk_40MHz : out STD_LOGIC;
clk_20MHz : out STD_LOGIC);
end component;

-- Second clock (36)
component clk_wiz_1
port (i_clk_100MHz : in STD_LOGIC;
clk_36MHz : out STD_LOGIC);
end component;

-- Third clocks (9.6)
component clk_wiz_2
port (i_clk_100MHz : in STD_LOGIC;
clk_9_6MHz : out STD_LOGIC);
end component;

signal s_sync, s_epoch : STD_LOGIC;

-- Constants
constant clk_sync : integer := 100e6 / 5e3;
constant clk_epoch : integer := 100e6 / 1e0;

-- Counters
signal sync_counter : UNSIGNED(14 downto 0) := (others => '0');
signal epoch_counter : UNSIGNED(27 downto 0) := (others => '0');

begin

first_clocks: clk_wiz_0 PORT MAP (
  i_clk_100MHz => i_clk_100MHz,
  clk_200MHz => clk_200MHz,
  clk_50MHz => clk_50MHz,
  clk_40MHz => clk_40MHz,
  clk_20MHz => clk_20MHz);

second_clock: clk_wiz_1 PORT MAP (
  i_clk_100MHz => i_clk_100MHz,
  clk_36MHz => clk_36MHz);

third_clock: clk_wiz_2 PORT MAP (
  i_clk_100MHz => i_clk_100MHz,
  clk_9_6MHz => clk_9_6MHz);
-- Sync and Epoch Timing

sync_epoch:
process (i_clk_100MHz) begin
  if(rising_edge(i_clk_100MHz)) then
    -- Sync
    if(TO_INTEGER(sync_counter) = (clk_sync - 1)) then -- reset counter
      sync_counter <= (others => '0');
      s_sync <= not s_sync;
    else
      sync_counter <= sync_counter + 1;
    end if;

    -- Epoch
    if(TO_INTEGER(epoch_counter) = (clk_epoch - 1)) then
      epoch_counter <= (others => '0');
      s_epoch <= not s_epoch;
    else
      epoch_counter <= epoch_counter + 1;
    end if;
  end if;
end process sync_epoch;

sync <= s_sync;
epoch <= s_epoch;
end Behavioral;

-- SPI_receiver.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_MISC.ALL;

entity SPI_receiver is
  Port ( i_clk_100MHz : in STD_LOGIC; -- System clock
          sclk : in STD_LOGIC; -- SPI input clock
          data_in : in STD_LOGIC; -- SPI data input
          SS : in STD_LOGIC; -- SPI Enable (active low)
          sclk : in STD_LOGIC; -- SPI input clock
          data_in : in STD_LOGIC; -- SPI data input
          SS : in STD_LOGIC; -- SPI Enable (active low)
      );
end SPI_receiver;
data_out  : out STD_LOGIC;   -- SPI data output
data_valid : out STD_LOGIC;   -- Output data is valid
data_received : out STD_LOGIC_VECTOR (31 downto 0) :=
  x"00000000";
end SPI_receiver;

architecture Behavioral of SPI_receiver is

  signal data_buf : STD_LOGIC_VECTOR (31 downto 0) :=
    x"00000000";
signal sclk_sample : STD_LOGIC_VECTOR (2 downto 0) :=
  (others => '0');
signal sclk_rise, sclk_fall, SS_rise : STD_LOGIC;
signal SS_sample : STD_LOGIC_VECTOR (2 downto 0) :=
  (others => '0');
begin

receive_data:
  process (i_clk_100MHz) begin
    if (rising_edge (i_clk_100MHz)) then
      if (sclk_rise = '1' and SS = '0') then
        -- Shift data left one bit
        data_buf <= data_buf(30 downto 0) & data_in;
      end if;
    end if;
  end process receive_data;

register_output:
  process (i_clk_100MHz) begin
    if (rising_edge (i_clk_100MHz)) then
      if (SS_rise = '1') then
        data_valid <= '1';
        data_received <= data_buf;
      else
        data_valid <= '0';
      end if;
    end if;
  end process register_output;

interpret_sclk:
  process (i_clk_100MHz) begin
    if (rising_edge (i_clk_100MHz)) then
      sclk_sample <= sclk_sample(1 downto 0) & sclk;
      SS_sample <= SS_sample(1 downto 0) & SS;
    end if;
  end process interpret_sclk;

SS_rise <= AND_REDUCE (SS_sample(1 downto 0)) and not SS_sample(2);
sclk_rise <= AND_REDUCE (sclk_sample(1 downto 0)) and not sclk_sample(2);
sclk_fall <= NAND_REDUCE (sclk_sample(1 downto 0)) and sclk_sample(2);

--TODO: create output circuit for data
  data_out <= '0';
end Behavioral;
## Appendix B: FPGA Signal Locations

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Type</th>
<th>Input/Output</th>
<th>Pin Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_clk_100MHz</td>
<td>Clock</td>
<td>In</td>
<td>E3</td>
</tr>
<tr>
<td>SW[0]</td>
<td>Switch</td>
<td>In</td>
<td>J15</td>
</tr>
<tr>
<td>SW[1]</td>
<td>Switch</td>
<td>In</td>
<td>L16</td>
</tr>
<tr>
<td>SW[2]</td>
<td>Switch</td>
<td>In</td>
<td>M13</td>
</tr>
<tr>
<td>SW[3]</td>
<td>Switch</td>
<td>In</td>
<td>R15</td>
</tr>
<tr>
<td>SW[4]</td>
<td>Switch</td>
<td>In</td>
<td>R17</td>
</tr>
<tr>
<td>SW[5]</td>
<td>Switch</td>
<td>In</td>
<td>T18</td>
</tr>
<tr>
<td>SW[6]</td>
<td>Switch</td>
<td>In</td>
<td>U18</td>
</tr>
<tr>
<td>SW[7]</td>
<td>Switch</td>
<td>In</td>
<td>R13</td>
</tr>
<tr>
<td>SW[8]</td>
<td>Switch</td>
<td>In</td>
<td>T8</td>
</tr>
<tr>
<td>SW[9]</td>
<td>Switch</td>
<td>In</td>
<td>U8</td>
</tr>
<tr>
<td>SW[10]</td>
<td>Switch</td>
<td>In</td>
<td>R16</td>
</tr>
<tr>
<td>SW[12]</td>
<td>Switch</td>
<td>In</td>
<td>H6</td>
</tr>
<tr>
<td>SW[13]</td>
<td>Switch</td>
<td>In</td>
<td>U12</td>
</tr>
<tr>
<td>SW[14]</td>
<td>Switch</td>
<td>In</td>
<td>U11</td>
</tr>
<tr>
<td>SW[15]</td>
<td>Switch</td>
<td>In</td>
<td>V10</td>
</tr>
<tr>
<td>data_received[0]</td>
<td>LED</td>
<td>Out</td>
<td>H17</td>
</tr>
<tr>
<td>data_received[1]</td>
<td>LED</td>
<td>Out</td>
<td>K15</td>
</tr>
<tr>
<td>data_received[2]</td>
<td>LED</td>
<td>Out</td>
<td>J13</td>
</tr>
<tr>
<td>data_received[3]</td>
<td>LED</td>
<td>Out</td>
<td>N14</td>
</tr>
<tr>
<td>data_received[4]</td>
<td>LED</td>
<td>Out</td>
<td>R18</td>
</tr>
<tr>
<td>data_received[5]</td>
<td>LED</td>
<td>Out</td>
<td>V17</td>
</tr>
<tr>
<td>data_received[6]</td>
<td>LED</td>
<td>Out</td>
<td>U17</td>
</tr>
<tr>
<td>data_received[7]</td>
<td>LED</td>
<td>Out</td>
<td>U16</td>
</tr>
<tr>
<td>data_received[8]</td>
<td>LED</td>
<td>Out</td>
<td>V16</td>
</tr>
<tr>
<td>data_received[9]</td>
<td>LED</td>
<td>Out</td>
<td>T15</td>
</tr>
<tr>
<td>data_received[10]</td>
<td>LED</td>
<td>Out</td>
<td>U14</td>
</tr>
<tr>
<td>data_received[11]</td>
<td>LED</td>
<td>Out</td>
<td>T16</td>
</tr>
<tr>
<td>data_received[12]</td>
<td>LED</td>
<td>Out</td>
<td>V15</td>
</tr>
<tr>
<td>data_received[13]</td>
<td>LED</td>
<td>Out</td>
<td>V14</td>
</tr>
<tr>
<td>data_received[14]</td>
<td>LED</td>
<td>Out</td>
<td>V12</td>
</tr>
<tr>
<td>data_received[15]</td>
<td>LED</td>
<td>Out</td>
<td>V11</td>
</tr>
<tr>
<td>o_0_tod_clk</td>
<td>Group 0</td>
<td>Out</td>
<td>C17</td>
</tr>
<tr>
<td>o_0_tod_en</td>
<td>Group 0</td>
<td>Out</td>
<td>D18</td>
</tr>
<tr>
<td>o_0_tod_data</td>
<td>Group 0</td>
<td>Out</td>
<td>E18</td>
</tr>
<tr>
<td>o_0_tod_step</td>
<td>Group 0</td>
<td>Out</td>
<td>G17</td>
</tr>
<tr>
<td>o_0_clk_40MHz</td>
<td>Group 0</td>
<td>Out</td>
<td>D17</td>
</tr>
<tr>
<td>o_0_clk_36MHz</td>
<td>Group 0</td>
<td>Out</td>
<td>E17</td>
</tr>
<tr>
<td>o_0_clk_20MHz</td>
<td>Group 0</td>
<td>Out</td>
<td>F18</td>
</tr>
<tr>
<td>o_1_tod_clk</td>
<td>Group 1</td>
<td>Out</td>
<td>D14</td>
</tr>
<tr>
<td>signal</td>
<td>group</td>
<td>type</td>
<td>pin</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>o_1_tod_en</td>
<td>Group 1</td>
<td>Out</td>
<td>F16</td>
</tr>
<tr>
<td>o_1_tod_data</td>
<td>Group 1</td>
<td>Out</td>
<td>G16</td>
</tr>
<tr>
<td>o_1_tod_step</td>
<td>Group 1</td>
<td>Out</td>
<td>H16</td>
</tr>
<tr>
<td>o_1_clk_sync</td>
<td>Group 1</td>
<td>Out</td>
<td>E16</td>
</tr>
<tr>
<td>o_1_clk_200MHz</td>
<td>Group 1</td>
<td>Out</td>
<td>F13</td>
</tr>
<tr>
<td>o_1_clk_36MHz</td>
<td>Group 1</td>
<td>Out</td>
<td>G13</td>
</tr>
<tr>
<td>o_2_tod_clk</td>
<td>Group 2</td>
<td>Out</td>
<td>K1</td>
</tr>
<tr>
<td>o_2_tod_en</td>
<td>Group 2</td>
<td>Out</td>
<td>F6</td>
</tr>
<tr>
<td>o_2_tod_data</td>
<td>Group 2</td>
<td>Out</td>
<td>J2</td>
</tr>
<tr>
<td>o_2_tod_step</td>
<td>Group 2</td>
<td>Out</td>
<td>G6</td>
</tr>
<tr>
<td>o_2_clk_36MHz</td>
<td>Group 2</td>
<td>Out</td>
<td>E7</td>
</tr>
<tr>
<td>o_2_clk_9_6MHz</td>
<td>Group 2</td>
<td>Out</td>
<td>J3</td>
</tr>
<tr>
<td>o_3_sync</td>
<td>Group 3</td>
<td>Out</td>
<td>G18</td>
</tr>
<tr>
<td>o_4_sync</td>
<td>Group 4</td>
<td>Out</td>
<td>H16</td>
</tr>
<tr>
<td>spi_clk</td>
<td>SPI</td>
<td>In</td>
<td>H4</td>
</tr>
<tr>
<td>spi_data_in</td>
<td>SPI</td>
<td>In</td>
<td>H1</td>
</tr>
<tr>
<td>spi_en</td>
<td>SPI</td>
<td>In</td>
<td>G1</td>
</tr>
<tr>
<td>spi_data_out</td>
<td>SPI</td>
<td>In</td>
<td>G3</td>
</tr>
<tr>
<td>o_6_tod_clk</td>
<td>Group 6</td>
<td>Out</td>
<td>H2</td>
</tr>
<tr>
<td>o_6_tod_en</td>
<td>Group 6</td>
<td>Out</td>
<td>G4</td>
</tr>
<tr>
<td>o_6_tod_data</td>
<td>Group 6</td>
<td>Out</td>
<td>G2</td>
</tr>
<tr>
<td>o_6_tod_step</td>
<td>Group 6</td>
<td>Out</td>
<td>F3</td>
</tr>
<tr>
<td>o_6_sync</td>
<td>Group 6</td>
<td>Out</td>
<td>B18</td>
</tr>
<tr>
<td>o_7_sync</td>
<td>Group 7</td>
<td>Out</td>
<td>J4</td>
</tr>
<tr>
<td>o_8_sync</td>
<td>Group 8</td>
<td>Out</td>
<td>E6</td>
</tr>
<tr>
<td>o_10_epoch</td>
<td>Group 10</td>
<td>Out</td>
<td>A18</td>
</tr>
<tr>
<td>o_11_tod_clk</td>
<td>Group 11</td>
<td>Out</td>
<td>A14</td>
</tr>
<tr>
<td>o_11_tod_en</td>
<td>Group 11</td>
<td>Out</td>
<td>A13</td>
</tr>
<tr>
<td>o_11_tod_data</td>
<td>Group 11</td>
<td>Out</td>
<td>A16</td>
</tr>
<tr>
<td>o_11_tod_step</td>
<td>Group 11</td>
<td>Out</td>
<td>A15</td>
</tr>
<tr>
<td>o_11_sync</td>
<td>Group 11</td>
<td>Out</td>
<td>B17</td>
</tr>
<tr>
<td>o_11_epoch</td>
<td>Group 11</td>
<td>Out</td>
<td>B16</td>
</tr>
</tbody>
</table>

### Appendix C: Arduino Code

```cpp
// SPI Library
#include <stdint.h>
#include <SPI.h>

// Set the clock rate to 250 kHz
// Send the MSB first
// The FPGA expects SPI_MODE0 to sample on rising edge
// ie: Clock polarity = 0
// Clock phase = 0
SPISettings SPI_settings(250000, MSBFIRST, SPI_MODE0);

const int SS_pin = 10;    // Slave select pin
```
uint32_t data = 0;
int rdy_bytes = 0;
char buf[32] = {0};
int i;

void setup() {
    Serial.begin(9600);
    pinMode(SS_pin, OUTPUT);
    SPI.begin();
}

void loop() {
    for (i = 0; i < 32; i++) buf[i] = 0;
    i = 0;
    while (Serial.available() > 0) {
        buf[i++] = Serial.read();
        delay(2);
    }
    if (i > 0) {
        Serial.print("i: "); Serial.println(i);
        Serial.print("buf: "); Serial.println(buf);
        data = (uint32_t) atol(buf);
        Serial.print("data: "); Serial.println(data, HEX);
        Serial.end();
        transmit_data(data);
        Serial.begin(9600);
    }
}

void transmit_data(uint32_t data) {
    SPI.beginTransaction(SPI_settings);
    digitalWrite(SS_pin, LOW);
    SPI.transfer16(data >> 16);
    SPI.transfer16(data & 0xFFFF);
    digitalWrite(SS_pin, HIGH);
    SPI.endTransaction(); // release the SPI bus
}

Appendix D: Python Code

```python
import serial
import time

# My computer (Windows) uses 'COM4' for Arduino
ser = serial.Serial('COM4', 9600, timeout=0)

# Continue until the user enters 'exit'
while True:
    input = raw_input('>> ')
    if input == 'exit':
        ser.close()
        exit()
    else:
        print 'Sending: %d (decimal)' % int(input, 16)
        ser.write('%d' % int(input, 16))
```
# Give time for device to process data
time.sleep(0.5)

# Read output from device
out = ''
while ser.inWaiting() > 0: out += ser.read(1)
if out != '': print out