Intro

For our project, we planned to develop a digital system which projects a 1.5 octave piano, of fixed dimensions, onto the floor, and is played using one's feet. This system interfaced a projector, a line array of distance sensors, and an FPGA to accomplish this task. We were inspired to work on a project like this because of the movie "Big." In one particular scene of the movie, Robert Losure and Tom Hanks play *Heart and Soul* on a giant floor piano in the famous NYC toy store, F.A.O Schwarz. We wanted to create a miniature version of this for ourselves, and for our own enjoyment. I mean, who *doesn't* want to play with a giant piano?

Overview

There are four main modules to this device: the *visual module*, the *sensor module*, the *sound module* and the *control module*. Each module was designed such that they minimally rely on each other so that parallel development is possible. This will be made apparent in the block diagram.

The first module - the visual module - required the FPGA to interface with a VGA projector. The main responsibility of this module was to display a dynamic image of a 1.5 octave keyboard. The image was updated in response to information from the sound module indicating which key was pressed. The module also indicated which mode the user was in via a colored dot.

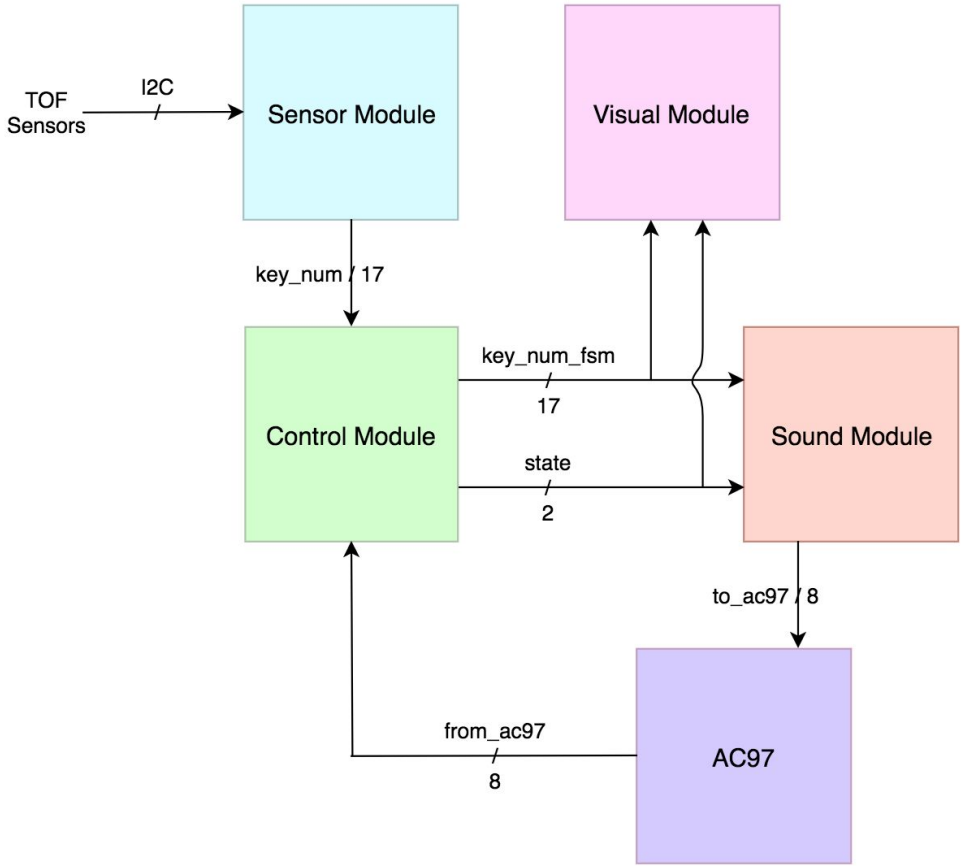
The second module - the sensor module - interfaced with an array of time-of-flight distance sensors. The main responsibility of this module was to use the information from the TOFs and have the FPGA generate a finite-state-machine which converts the given information to a number indicating which key was pressed. This information is updated periodically and made available to the other two modules. Originally, the module was meant to be entirely implemented on the FPGA, the only physical interface being a two-wire I2C bus. However, due to time constraints, the sensor information had to be preprocessed on an Arduino nano which, provided the keypress input to the other three modules.

The third module - the control module - acted as an FSM that allowed for communication between the different modules. It allowed for 3 primary modes: user mode, record mode, and playback mode. Transitions between these modes were controlled by buttons on the FPGA. The module received a 17 bit key number from the sensor module indicating keys currently being pressed. Then depending on the current mode it outputted the key number that the visual module and sound module should use at that time.

The fourth module - the sound module - took input from the control module the 17 bit key number to play. This module will then use the FPGA to interface with the AC97 codec to generate a tone corresponding to the right note for the given key press.

This represents the basic functionality of our project. The nuances of each module, will be given in further detail in the following sections.

High Level Implementation

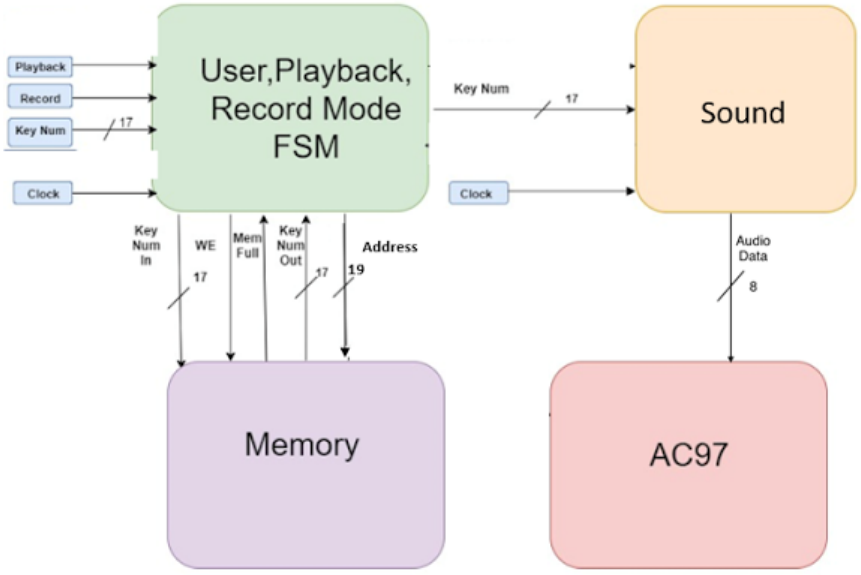


Module: Control (Sarah Flanagan)

Introduction:

The control module functioned as an FSM that determined which mode the piano should be in and outputted the appropriate state and key number to be used by the sound and sensor module. There were 3 main modes: User Mode, Record Mode, and Playback Mode.

The main inputs are the 27mHz clock and the key number from the sensor module. The main outputs to other modules are the key number generated from the FSM and the current state, which indicates the mode. Other important signals include we_ZBT0, data_in0, data_out0, and address0, which controlled writing to ZBT memory.



Specifics of Implementation:

User Mode is the typical functionality of a piano. When the user places their foot on the key, the sensor module sends it a key number and that same key number was also outputted from

the module. This was represented as a 17 bit array called key_number where 1's indicated that a key was pressed and 0's indicated that a key was not pressed. Due to time constraints the sensor module was only able to display data for the last 5 keys instead of all 17 keys. As a result, the left, right, up, down, and numbered buttons and switches 7 through 5 were used to represent a key press for the 12 most significant bits and the data from the sensor module was used for the 5 least significant bits.

Record mode is activated by pressing button 0 on the FPGA. In this mode, the key number indicating current notes being pressed is both outputted to the user every clock cycle and stored into ZBT memory. The user can leave record mode and go back to user mode by pressing the record button a second time or the control module will automatically go back to user mode when the memory is full. The key numbers were stored in SRAM0 after every 4 clock cycles. Writing to memory only every 4 clock cycles allowed for longer recordings while not creating a significant decline in recording accuracy. In order to further increase the length of recordings I could have stored 2 key_numbers per address since each line in ZBT memory was 36 bits and each key number was only 17 bits.

Playback mode is activated by pressing button 1 on the FPGA. In this mode, the module starts at address 0 and reads the key number stored in ZBT memory after every 4 clock cycles (the same amount of cycles used to write the notes into memory).

As a stretch goal in an attempt to load instrument sounds, I created a 4th mode in the control module called calibration mode. The user can connect their phone to the FPGA using an audio cable. In this mode, the note that should be played on the phone lights up on the the keyboard. The user then presses a button to start recording audio data from the phone corresponding to that note. This data is stored into ZBT memory for the first 2^14 addresses. Since key numbers were stored in SRAM0, the audio data was stored in SRAM1. After this time the next key lights up and the user can store this phone audio data in the next 2^14 addresses when they have the appropriate button pressed. This process repeated until audio data for all 17 notes had been recorded. I was able to properly implement all of the Verilog to write this audio data into ZBT memory. However, I ran out of time to debug a synthesize error that prevented

proper playback of this data in the sound module. In the future I would like to work on further developing playback of instrument sounds.

Module: Sound (Sarah Flanagan)

The sound module is responsible for generating an 8 bit tone corresponding to the 17 bit key_number that it received from the control module. This was done through modifying the tone 750 module from lab 5. This module produced a 750 Hz tone. It had 64 points on a sine wave that it stepped through based on a ready signal that occured with a frequency of 48 KHz. It then sent the point on the sine wave to the ac97 as an 8 bit bus.

Every note has a naturally occurring frequency in Hz. In order to be able to produce tones at the appropriate frequency for all notes on the piano, I needed to change the frequency of this ready signal. To do this, I created a ready generator. The ready generator was a clock divider that would set the “new_ready signal” to high for one cycle. I created a lookup table that determined how many 27 mHz cycles should occur prior to making the “new_ready” signal be set to high.

In order to determine the frequency of the new ready signal in KHz that would step through the 64 points of the sine wave to create the appropriate note frequency, I multiplied the note frequency by 64 and divided by 1000. To figure out the number of 27mHz clock cycles to count prior to setting “new_ready” to 1, I did 27,000/frequency of the new ready signal. I then created a look-up table using a case statement to use the appropriate clockDividerNum depending on the key numbers pressed. These calculations are represented by the two equations below.

$$noteFrequency(Hz) * 64 * \frac{1KHz}{1000Hz} = readyFrequency(KHz)$$

$$\frac{27MHz}{readyFrequency(KHz)} * \frac{1000KHz}{1MHz} = clockDividerNum$$

The table below shows the calculated values based on the unique frequency of each note. The values in the final row (27mHz divider (Rounder)) were the values used in the look-up table of the ready signal generator.

Note	C	C#	D	Eb	E	F	F#	G	G#	A	Bb	B
Frequency	261.65	277.2	293.65	311.15	329.65	349.25	370	392	415.3	440	466.15	493.9
Ready Frequency (kHz)	16.7456	17.7408	18.7936	19.9136	21.0976	22.352	23.68	25.088	26.5792	28.16	29.8336	31.6096
27mHz divider	1612.364	1521.916	1436.659	1355.857	1279.766	1207.946	1140.203	1076.212	1015.832	958.8068	905.0198	854.1709
27mHz divider (Rounded)	1612	1522	1437	1356	1280	1208	1140	1076	1016	959	905	854

Note	C(high)	C#	D	Eb	E
Frequency	523.5	554.4	587.3	622.3	659.3
Ready Frequency (kHz)	33.504	35.4816	37.5872	39.8272	42.1952
27mHz divider	805.8739	760.9578	718.3296	677.9287	639.8832
27mHz divider (Rounded)	806	761	718	678	640

One interesting observation I discovered was that the frequency of a note an octave above was exactly double the frequency of the original note. This also meant that the number the ready_generator clock divider needed to count up to should be cut in half or shifted to the right by 1 bit in order to make the note go up one octave. In order to make the frequency of all notes an octave up, I was able to simply right shift by one bit in the look-up table. Similarly if I had wanted to make the notes go down an octave I could have had a left shift by one bit.

I started off by only allowing one note to be played at a time. If multiple notes were pressed at the same time I would choose only the most significant bit to be played. As a stretch goal I then attempted to play 2 notes at the same time. To do this, I produced 2 sine waves at frequencies corresponding to the different notes on the piano. I then attempted to add up the 2 sine waves, by adding the data points at each positive edge of the clock and right shifted by one bit to divide by 2. Although this made sense in theory, in practice the combination of the two notes was very displeasing to hear. In an attempt to debug this I made sure that both sine waves would start at the same time, based on a restart signal that was set to high every time the key number changes. However, this did not fix the overall sound. I also attempted increasing the number of bits of sine wave data sent to the ac97, but this did not fix the sound quality either so I went back down to 8 bits.

At this point, I decided to attempt storing instrument sounds instead. However, had I pursued sine wave addition further I would have attempted calculating more points on the sine wave than the current 64 or filtered the data in an attempt to further smoothen out the sine wave.

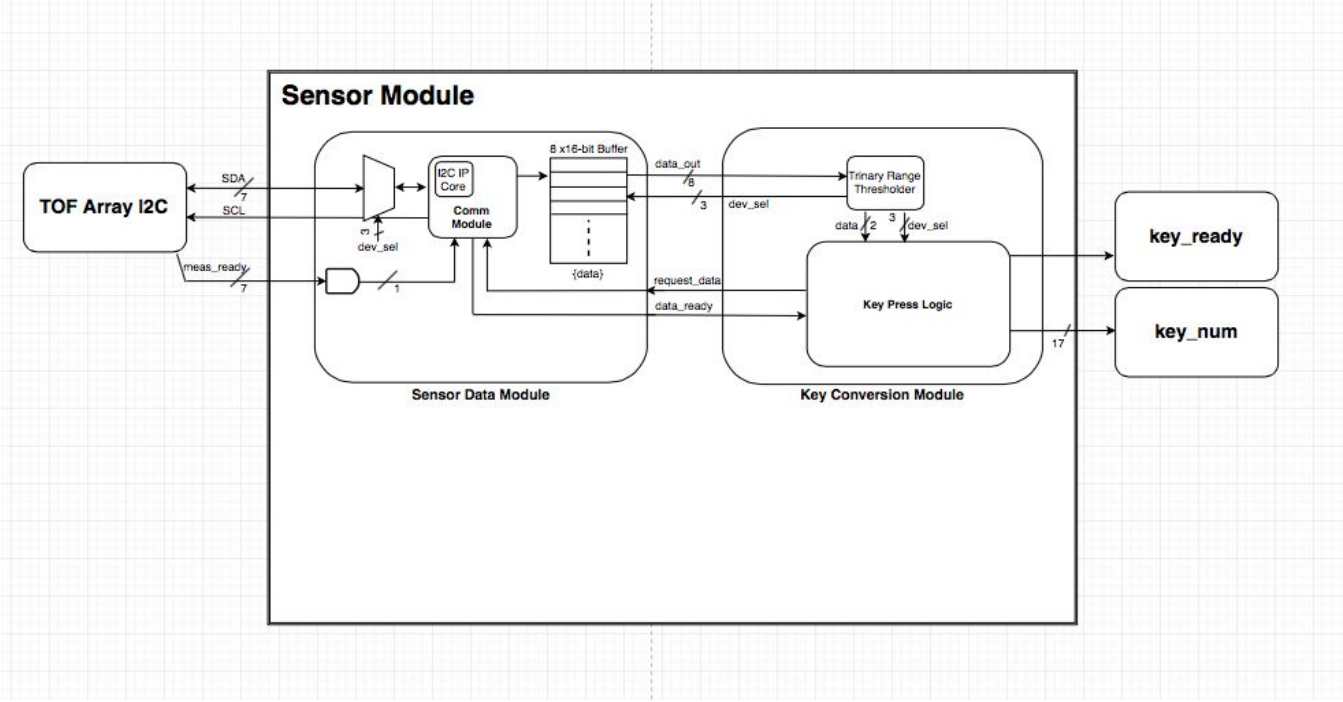
In an attempt to playback instrument sounds I attempted to playback the data stored in ZBT memory in SRAM1. The key number would be sent into an address generator that would determine the proper location in memory where a note was stored. It did this by calculating the note number, a number between 0 and 16 and then left shifted it 14 bits. This is the equivalent of multiplying the key number by 2^14, which was the size of each note stored in memory. It would then loop through that starting address up until the starting address + 2^14-1. For example, if the key number bus was 0_0100_0000_0000, this would indicate that a D needed to be played. (The most significant bit corresponds to a C, the next most significant was a C#, and third most significant was a D and so on). In this situation the system would start with address 2*2^14 and loop back to this start address once it reached 2*2^14+2^14-1. This all should have theoretically worked and I came very close. However, unfortunately due to time constraints a single synthesize error held me back and I was unable to fully implement playback of instrument sounds. I regret not starting on using instrument sounds sooner because I feel that if I had given myself one more day, I would have been able to fully implement the system. Part of my setback was misunderstanding the proper cable that I need. While I originally thought I need an RCA cable, I realized after spending sometime confused on how to use it, that what I actually needed was a male to male stereo audio aux cable. I highly recommend that future groups check in earlier with the lab staff to make sure that they are using the proper cable.

Module: Sensors (Liam Cohen)

Introduction:

The sensor module's purpose was to take data from an array of time-of-flight distance sensors (VL53L0X), and convert it into a 17-bit bus describing the power-set of every possible combination of key presses on a 1.5 octave piano. Originally, we believed that we would be able

to uniquely determine any combination of key presses using only seven T.O.F sensors, however later in the project we realized that we would actually need eleven sensors to generate the full 17 key superset. We decided to use T.O.F sensors because for short distances they are incredibly accurate, and in even in a long range limit (about 1 meter away from the sensor) they are still far more accurate than other traditional distance sensors. A high level block diagram of the sensor module's system architecture can be seen in the figure at the top of the next page. Furthermore, more details on the physical implementation and our development process will follow as well.



As can be seen in the high level implementation, the sensor module originally had two sub-modules. One module would have been responsible for gathering the data from the T.O.F sensors via I2C, and the other would have been responsible for converting that data into a 17-bit bus, representing which keys were pressed, that would be sent off to the sound and visual modules for processing. However, due to time constraints, we were unable to fully implement the sensor data module purely on the FPGA, and we had to result to piping out only 5 bits out of

the 17 bits in the key-number bus to the sound and visual modules by using an Arduino Nano as a conduit between the T.O.F sensors and the FPGA.

Ultimately, interfacing with the VL53L0X was far too complicated, and although over 100 hours was spent attempting to get the communication to work, it was unsuccessful. The reason why communicating with the VL53L0X was so complicated is because the original manufacturer, ST, has not released a register map for the sensor. Instead they have only released a C based API that can modified and then uploaded to a microcontroller to control the sensor. The library we attempted to base our Verilog code off of was written by Pololu for their hardware implementation of the VL53L0X. The required setup for the VL53L0X, before taking any sensor data, required over 100ms straight of I2C register writes; this can be interpreted as a measure of the sensors complexity.

In addition to the sensor complexity, many global and local variables needed to be stored, and complex calculations on that data needed to be performed as well. This required instantiating block RAM and a related memory controller. Much of the Verilog that was written for the sensor module ended up as a large amount of hard coded instructions that were required to interface with the VL53L0X. Effectively, the Verilog presented for the sensor module represents that of a highly task specific processor. This was a grave misuse of the FPGA, and we should have realized that it was more appropriate to use a microcontroller to interface with the T.O.F. sensor array. This way, we could have used the FPGA for signal processing, and most likely would have had a fully working piano that was far more accurate and responsive than the demonstrated version.

Despite not being able to completely extract sensor readings from the VL53L0X with the FPGA, we did have a great amount of success in setting up the I2C back bone for communicating with the sensor. There are 4 main modules which contribute to the I2C backbone, three are serializers, and one is an I2C master that was taken from an open-source project by Alex Forencich. The serializers are finite-state machines which are designed to interface with the I2C master and take a register address byte and optionally a data byte in parallel, and send them to the master at the appropriate time such that data can be read or written to from a specific register on the slave. The operation of these serializers are completely

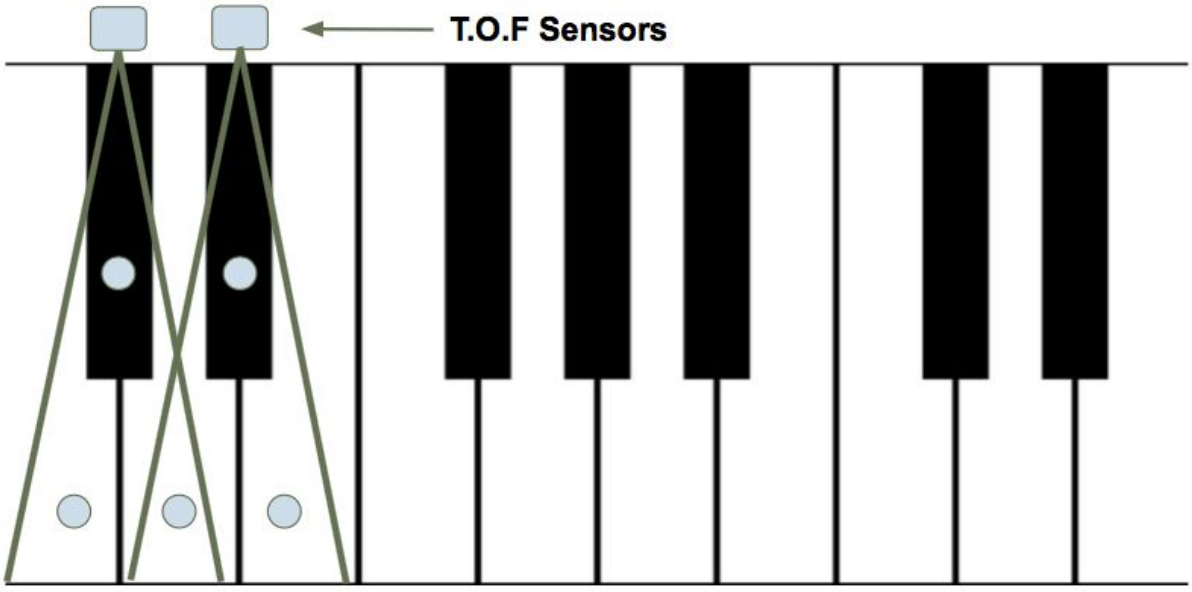
verified, and in conjunction with the I2C master allow for incredibly simple FSMs to be generated which can perform register writes and reads to an I2C slave. This code may be extremely useful for future 6.111 projects that utilize I2C communication protocols.

Another merit to our system, is that although much of the Verilog written to have the FPGA communicate with the VL53L0X directly ended up not being used in the presented version of the piano, everything that was written is functional. The only reason we were not able to retrieve sensor data from the VL53L0X was that the complexity needed for addressing the sensor was too great to finish the required code in time (not for lack of trying on our end). However, what is currently written, demonstrates our teams' understanding of the complex interplay of multiple digital systems. The sensor module's verilog interfaces multiple IP cores, both RAM for storing the results of register reads and performing calculations on them, and ROM for storing the vast swath of necessary register writes. Furthermore, we interface all of these with our own written FSMs for communicating with the VL53L0X. Even though we didn't meet our full system goal, what is currently employed, demonstrates an advanced understanding of digital system integration on an FPGA.

Specifics of Implementation:

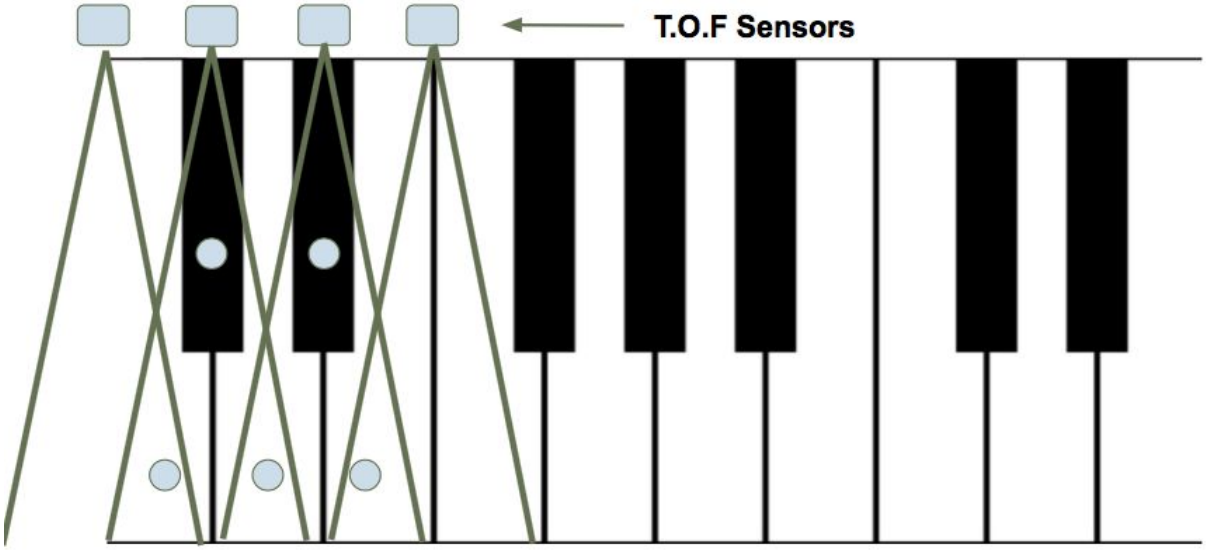
Hardware:

First we will discuss the hardware implementation for the sensor module. As mentioned in the introduction, the physical hardware setup is a line array of time-of-flight sensors, specifically an array of VL53L0Xs. The sensors are arranged such that by using the overlap between the sensors, one can distinguish key presses for all 17 keys by using fewer than 17 T.O.F sensors. The key realization that makes this work is that the T.O.F sensors are not perfectly collimated, and have a 25˚ cone of sight. For example, if you look at the figure below, you will see that for all of the keys between the C and the E key, by only placing two sensors on the accidentals, we were able to generate states for all 5 keys.



Since each sensor will pick up a reading for any object within its cone of sight, looking at the point where the two T.O.F sensors in the figure above would both pick up readings, we see this corresponds to the depression of the D key. Furthermore, in the example above, both T.O.F sensors by themselves are sufficient enough to distinguish the C, C#, D#, and E keys.

However, we noticed a problem with this implementation that would need to be resolved if other 6.111 teams were to try and re-implement our design. In the figure above, having two T.O.F sensors is not quite enough to fully distinguish between every combination of key presses. To do this correctly, we would need to add two additional T.O.F sensors for the C through E key, and 4 T.O.F sensors for the piano over all, bringing the total number of required sensors to 11. The main issue, is that the sensor array as presented could not distinguish between simultaneously pressing the C and E keys, and just pressing the D key by itself. However, if you see the figure below, if we add two more T.O.F sensors, these would give us enough information to discriminate between cases like these, allowing all combinations of keys to be simultaneously pressed.



This remedies the problem, as one can see, by introducing more sensor information. For example, if one were to press the D key, this would uniquely respond to solely readings from the inner two T.O.F sensors. However, if one were to press the C and E keys simultaneously, this would also pick up readings from the outer two sensors as well. Extrapolating the same pattern to the rest of the piano, a total of 11 T.O.F. sensors would be required, 4 more than we originally expected, to distinguish simultaneous key presses.

Something that is worth mentioning is a reflection on resource allocation. We spent an enormous amount of time attempting to get the FPGA to communicate with the sensor array, which, as mentioned, due to the complexity of the VL53L0X, was not successful. If we had used a microcontroller to interface with the sensor array sooner we could have focused on more interesting problems, that would have been more worthwhile to work on. One example is that data from the T.O.F's is noisy. On the Arduino Nano when we set up our demo version of project, we used a simple rolling average filter to reduce noise, however, since the VL53L0X has a long integration time (20ms) at minimum. Taking 10 averages, although cleaning up the signals, introduced a small but noticeable delay of 200ms between a physical key press and the note actually playing. Something that would have been extremely interesting and useful to implement on the FPGA would have been some kind of Bayesian estimation, or Kalman filtering, which uses minimal samples and a probabilistic model of the sensor to generate the most likely key press. This would have been a great use of the FPGA, and would have greatly

improved the piano's responsiveness. It also would have added a layer of deep complexity to the project that would probably have been a great stretch goal: ultimately, this would be enabled by reducing the project complexity by not having the FPGA physically interface with the VL53L0X directly, and instead have a microcontroller as a conduit, and UART the relevant data into the FPGA for processing.

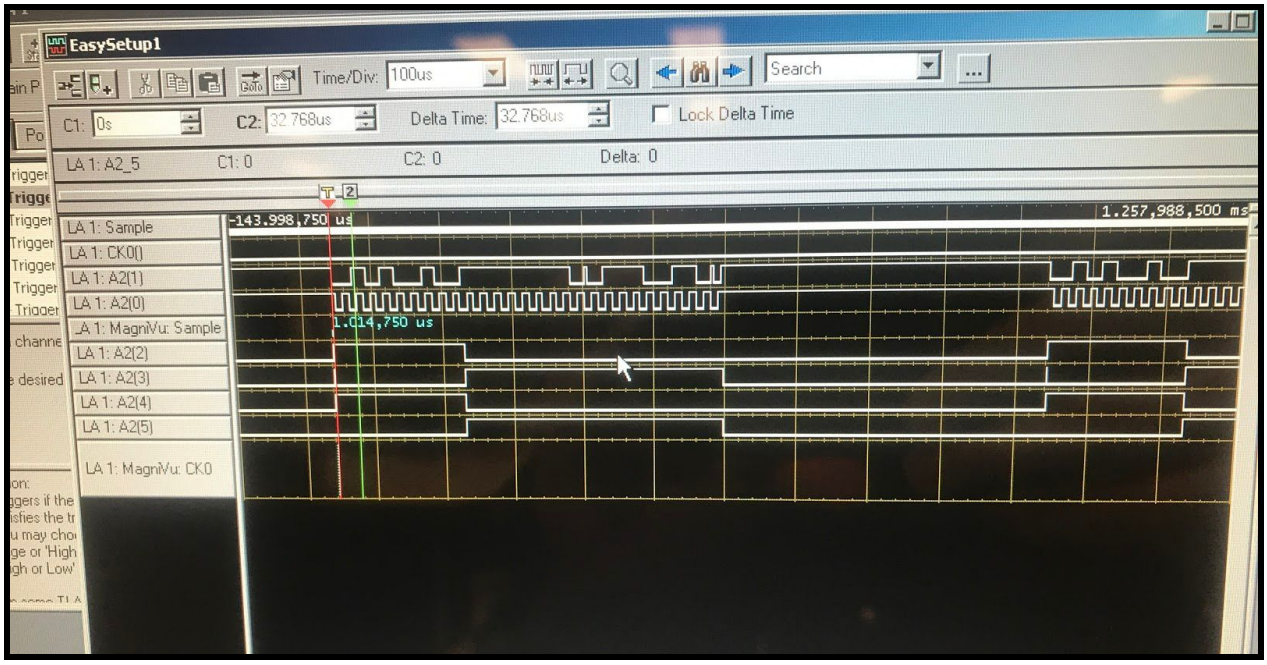
Another comment on possible clues for further implementation. We never got to fully implement simultaneous key pressing. However, keeping the idea that the best use of the FPGA, in this case, is signal processing in mind. Consider, after cleaning up the raw data from the sensor, acquired via a microcontroller like the Arduino, in order to figure out which keys were pressed, it would be intelligent to implement some type of SAT solver on the microcontroller itself. So, raw data comes out of the microcontroller, the FPGA processes it, and it returns a binary representation of which sensors read a ternary value of close, far, or out of range. Then whether or not each key individually is pressed can be represented as some logical combination of these boolean variables. The microcontroller can then run an SAT algorithm on this data and determine which keys are pressed or not, and return this data back to the FPGA as a 17-bit number. This is just a suggestion for future teams who may want to implement a similar project.

I2C Framework:

Moving on, I want to touch on some of the I2C back bone. Although this did not make it into the demo version of our project, a lot of effort went into developing a framework for the FPGA to communicate with the VL53L0X via I2C. Communicating with the VL53L0X time-of-flight sensor required developing a framework for performing register writes via I2C to the slave. In order to actually communicate with an I2C slave, one has to first send over, from the master, the I2C device address, then the value of the byte wide register address, and then the actual data one wants to write. In the case of I2C reads, it's much the same, except instead of sending in a data byte, the master waits to read in a data byte from the slave.

For this project we wrote three main I2C modules that communicate with a pre-built I2C master by Alex Forencich. The three modules are: i2c_register_read.v, i2c_register_write.v, and write_multi.v (along with i2c_master.v). The first, i2c_read_reg.v, reads any specified number

of data bytes from a particular register on the slave: it stores these data bytes into a fifo. It is then the FPGA designers responsibility to clear out said fifo before the next use. The second, i2c_reg_write.v, simply writes a single data byte to a specified register on an I2C slave. The third, write_multi.v, writes a specified number of bytes to an I2C slave. It does this by taking all of the data bytes stored in a fifo, and sending them out to a given register address on an I2C slave one by one. These three modules are enough to generate an FSM that communicates over I2C to a slave in any type of foreseeable fashion. Furthermore, all of these FSMs have built in timeouts so that they are capable of dealing with failed communications and NACK bits, without disrupting system execution, instead they throw an error flag. I believe this code will be of great use to future 6.111 students who would like to communicate to an I2C slave. (See below for logic analyzer image of verified I2C register writes).



A particularly challenging aspect of developing these modules was getting i2c_read_reg.v to work correctly. The module required both write and read functionality from the I2C master, and it also required communication with its built-in fifo. The I2C master was not particularly well documented, and it wasn't entirely clear how to take advantage of its full functionality. A lot of designing the I2C serializers went into figuring out what the correct signals were to send to the I2C master, and at what time. This probably could have been made

less tedious, again by avoiding having the FPGA communicate with I2C all together, however, other than that, there was probably no way to avoid the tedium since we had no I2C communication IP cores available to us, and I2C master we had was not very well documented. However, future students who want to write to communicate to a slave via I2C can use our modules and the master provided very simply. Once they instantiate the FSMs, all they have to do is provide the I2C device address, the byte wide register address, and the data bytes if performing a write, and then flash the start signal. The serializer FSMs we wrote will automatically send out the information in the correct order, and return the requested information if necessary.

Additional Implementation Details:

Briefly, there are other aspects to the sensor module implementation that were not previously mentioned. These include the hardware modules used to initiate sensor readings from the VL53L0X. Ultimately, this code ended up unfinished, so not as much attention will be given to its implementation details. However, some are worth mentioning due to their complexity and the design challenges they presented. These modules include, VL53L0X_INIT.v, get_measurement_timing_budget.v, getVcselPulsePeriod.v, get_spad_info.v, RAM.v, ROM.v, timeoutMclkcsToMicroseconds.v, timeoutMicrosecondsToMclks.v, and mem_ctl.v. Most of these modules are designed to mimic C functions as presented in the Pololu VL53L0X library (https://github.com/pololu/vl53l0x-arduino). As such, they interface with RAM to store global and local variables for later computation. They also all have instruction counters which allow them to move between hard-coded instructions to execute the various commands required for setup of the VL53L0X. They also interface with a ROM which holds all of the specific register writes needed to setup the sensor. Again, we will not go into too much detail into how these work because they are not of use to future 6.111 students, they were not utilized in our demonstrated design due to incompleteness, and their functionality is apparent from the Verilog and from the C functions they are attempting to emulate.

I will, however, make a comment how designing a memory controller to interface with the RAM was a really useful design trick. Writing and reading to the RAM requires somewhat tricky timing, and is challenging to integrate into an FSM. Having a sub-FSM that allows one to abstract this timing away and simply provides a done signal when writing or reading from the RAM is complete was very helpful. For any future 6.111 students who plan on using a RAM for their project, writing a quick memory controller will greatly improve your lives.

For a semi-low level block diagram depicting the sensor module's detailed implementation see the following link. This depicts both implemented and unimplemented modules and should act as a guide for reimplementation if desired.

https://www.draw.io/?lightbox=1&highlight=0000ff&edit=_blank&layers=1&nav=1&title=Write_I2C_Reg.xml#Uhttps%3A%2F%2Fraw.githubusercontent.com%2Fliamcohen%2FProjectedPiano%2FSensorModule2%2FWrite_I2C_Reg.xml

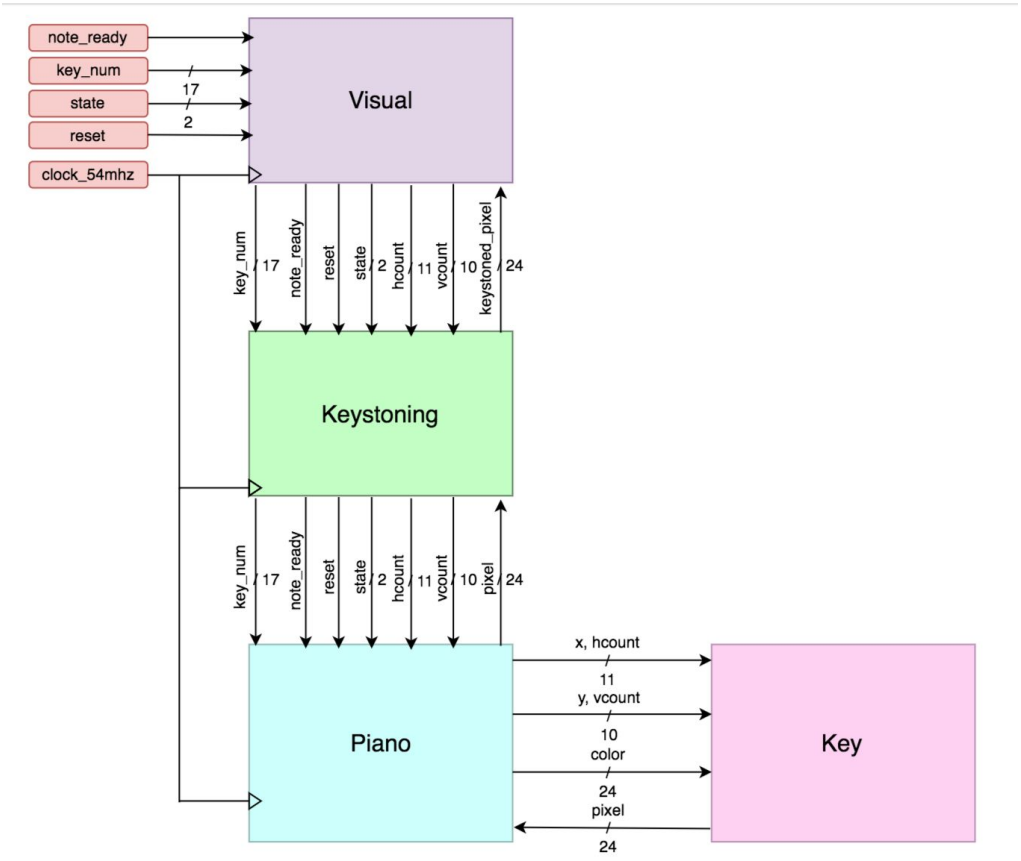
Conclusion:

The main takeaway from this section is that, although we were not successful in directly interfacing with the VL53L0X with the FPGA, in retrospect we learned a lot on how our implementation could have been improved, and how we could have done better. Despite this, even though our original goal in terms of sensor interfacing was unsuccessful, it still bore useful fruit for future 6.111 students. So ultimately, even in failure, we were still successful.

Module: Visual (Zoe Klawans)

Introduction:

The main purpose of the visual module was to display the image of the piano on the floor using a projector, highlighting keys as they were pressed or played back. At first we intended to build our hardware setup so that the projector's built-in keystoning would be sufficient for our purposes, but after making adjustments to the physical setup, we needed additional software keystoning. Instead of using images of a real piano stored in memory for the projected image, we focused on keystoning the original image made of 17 different sprites, one for each key in 1.5 octave piano.



The visual module receives the note_ready signal and the key_num containing the key press information from the sound module. It receives the mode state information from the control

FSM. The visual module passes on note_ready, key_num, and state to the keystoning module, and creates the VGA signals using the resulting keystoned_pixel signal. The piano module creates the pixels for the un-keystoned piano, and it contains 17 of the key module, using the key_num signal to determine which keys to highlight.

Specifics of Implementation:

Hardware:

The projector was mounted on a ladder, pointing towards the floor at a fixed angle. The FPGA was placed close to the TOF sensors, and the projector was attached via a long VGA cable. The fixed angle caused the piano to be at a constant distance from the projector, allowing for fixed keystone calculations.

Sprites:

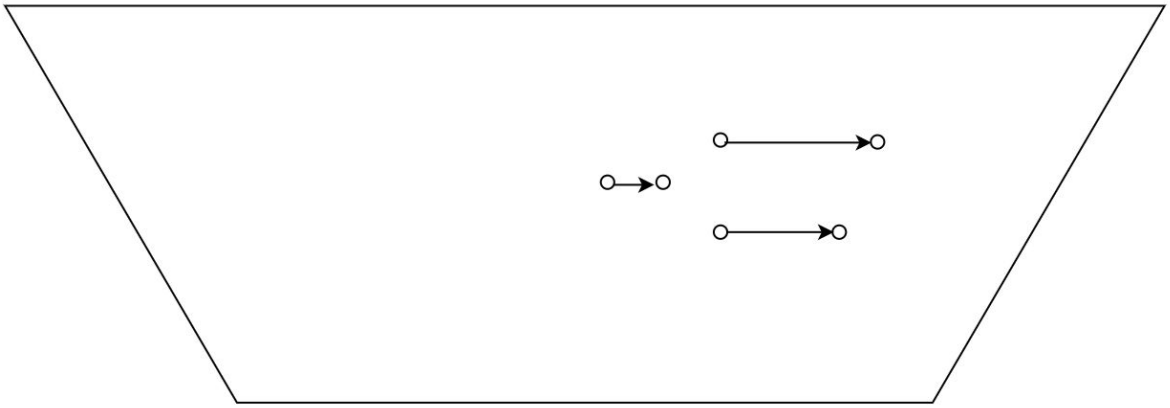
The keys were created using 17 different instances of various sprite modules. There are separate modules for the black keys, the white keys between two black keys, the white keys to the left of one black key, and the white keys to the right of one black key. The bits of key_num[16:0] corresponded to each key on the piano, with key_num[16] being the low C, and key_num[0] being the high E. If any of the bits were a 1, then the corresponding keys would light up in red. Given the starting hcount and vcount position for the low C and the width of the white keys and black keys, the piano module placed the key sprites in position.

In addition to all the keys, there was a colored dot that indicated which mode the user was in. No dot meant user mode, a red dot meant record mode, a green dot meant playback mode, and a blue dot meant instrument calibration mode.

Keystoning:

Keystoning was the biggest challenge of the visual module. The piano image made of sprites had to be linear transformed into a trapezoid, with the top side longer than the bottom side, in order to appear as a normal rectangular piano on the floor. This involved spreading out the piano like a fan from the middle, so that the middle remained untransformed, and the transformed lines of the piano got less steep as you move outward from the middle.

In order to choose the correct pixel color for the transformed image, we had to calculate where that pixel would correspond to on the original piano image, and then use that color. This calculation depends on both the vertical height of the pixel with respect to the piano, as well as the distance of the pixel from the center of the piano.



```
x <= hcount - PIANO_MIDDLE;
phcount <= hcount - (((vcount - KEY_START_VERTICAL) * (x * -1)) >> 10);
```

The above verilog code first calculates the change in horizontal distance from the center. Then to find the pixel needed from the piano, it takes the keystoned image hcount and subtracts the pixel's distance from the top of the piano. However not every pixel can be moved by the same amount since they need to fan out, so that subtraction needs to be scaled by a value based on hcount. We found that a good approximate scaling factor is the distance of hcount from the center of the piano over the entire size of the screen (1024 pixels). Since x is a signed value, multiplying it my negative one accounted for both left and right shifts on either side of the piano. However, when doing the same calculation using addition and subtraction, the keystoning didn't work. Given more time, we would have investigated this phenomena more and tried to implement keystoning using the addition/subtraction solution to further understand how it works. We also would have made the keystoning more flexible for change by controlling the scaling factor with buttons on the FPGA. This could be done using a lookup table to calculate the numerator and denominator of the scaling factor.

The same keystoning algorithm based on the vertical position of the piano was used for the colored dot for mode indication. Since the dot was above the piano, it got stretched a little and looked more like an oval. With more time, we would have implemented a second algorithm for the dot, or have generalized the existing one more.

Clock domain:

Different clock domains were needed for the visual and sound modules. We started with the visual module using the 65mhz clock from lab 3, and the sound module using the 27mhz clock from lab 5, but this caused issues when we synthesized the two modules together. Since the piano visuals do not require video game levels of visual updating, we used the DCM to synthesize a 54mhz clock for the visual module. Then we used a clock divider to recreate the 27mhz clock for the sound module to ensure the two clock domains were synchronized.

Discussion

Although we were able to get a minimal version of our project up and running on time, we did so just barely. We learned an enormous amount about project management throughout the duration of our project, and we wish to pass this information onto the next generation of 6.111 students. In general, we ran into issues both with understanding the scope of a problem presented to us and with standard time management procedures. We learned that even large amounts of time invested can lead to failure modes if one is incorrect in their estimate of how much of the project still remains. These problems manifested themselves in all four modules, though most severely in the sensor module. In this section, we will briefly conclude our report by discussing these issues and how future 6.111 students can avoid them in the future.

First, both prepare and be ready to accept slippage: one should think carefully about the amount of time a task will take, and then plan ahead assuming that task will take double the amount of time predicted. Furthermore, when thinking about the overall goal of a task, think of the simplest implementation possible, and then attempt to implement that first, even if as a proof of concept. Oftentimes, one will find that there are many difficulties associated even with the simplest way of doing things, and project complexity will simply arise from solving these issues.

Specifically, in our case, it would have made the most sense to develop our our system from the beginning using an Arduino Nano as the conduit for extracting out sensor data from the VL53L0X time-of-flight sensor array. This way, we could have piped out all of the necessary sensor data to the FPGA right away. What we would have realized if we had done this early on, instead of focusing on the Verilog code to extract out sensor readings via the FPGA only, is that the input data is noisy, and needed to be filtered. This would have been an excellent use of the FPGA, and could have been expanded in complexity throughout the project if need be. Most likely, if we had done this, we would have had a fully synthesized system up and running within the first two weeks, which was our original goal in the first place.

If one decides to try a more challenging task first, set a *hard* deadline for when that challenge should be met by, and if it's not, make sure that you fall back on something. For us, we found that in trying to get the FPGA to read sensor data on its own, we kept having small amounts of success, which convinced us that we should continue forward. However, progress although successful, was slow, and our personal deadlines for certain tasks slipped away. Having continued small successes convinced us that it was possible to accomplish what we wanted the hard way, but in reality, by the first missed deadline, it should have been apparent that switching to the back-up plan, i.e., using an Arduino for sensor input in our case, was the correct choice.

This leads to a comment on team dynamics. When we planned out the project, we were intelligent in thinking about slippage and generating backup plans, our issue lied in the overall decision to execute those backup plans. This could have been greatly improved with more open communication between team members. For the most part, we mostly left each of our team members to their own devices. However, for each section of the project, we let our personal deadlines slip by, and we didn't hold each other accountable. It can be challenging to balance bugging your team members to get things done while still fostering a positive and encouraging work environment. Additionally, it can be difficult to hold a team member accountable for a deadline if you are also behind on your deadline.

Before starting the project, creating a team contract or creating a plan of action of what to do when teammates miss deadlines would have been very valuable. Then when the situation

occurred teammates could point to the contract and hold their teammates accountable without feeling guilty. Additionally, having more frequent team meetings would establish the precedent that each member had an important voice in the project, and could without guilt make other members accountable for their work.Knowing when to give up on something and fall back is a challenge, especially if continued progress is made. Although personal introspection and overall perspective on the project is key for knowing this, having team members that are comfortable reminding each other about deadlines, and holding each other to their word, is an invaluable asset and can greatly improve project efficiency.

From a more practical standpoint, when working with an FPGA, it's important to give some thought to what a good use of the FPGA is as well. Ultimately, if you're working with the FPGA, and you find that the Verilog that you're writing effectively seems like writing a processor with hard-coded instructions, then most likely this is a bad use of the FPGA. This was the case when working with the sensor module. Communication to the VL53L0X was an incredibly complicated beast, and much of the Verilog that ended up being written were I2C write or read instructions, along with memory storage, and other components, that again ended up being like hard-coded instructions in a processor. Ultimately, a microcontroller was just the best task for this job, and an FPGA was the wrong way to go about it. In the beginning it seemed like using an Arduino was "cheating" because it was too easy, but in reality, even things that seem easy are more complex than you originally scope them out to be. Understanding what is the right equipment for a job is an important part of being an engineer.

Again, from an engineering standpoint, if one is working with physical hardware that interfaces with the FPGA, it's best to get the hardware setup in its full form as quickly as possible. There are a lot of unforeseen issues that arise when interfacing with external hardware that ultimately may require solutions that don't even really have much to do with the hardware itself. For us, since we didn't set up our projector until about half way through the project, we didn't realize that our main piano image would need massive keystoning to make the image look reasonable. This added an entire layer of complexity for the visual module halfway through the project. This could have easily been avoided if we had setup the projector within the first week.

In the same vein, we could have had the basic sensor hardware setup within a week, and we would have been able to deduce all of the issues surrounding that as well.

Ultimately, we learned a lot over the course of the semester, both in terms of digital systems, and in terms of project management. Both experiences have been invaluable and I believe will greatly improve our ability to complete complex projects in the future: both for digital systems and in general. All in all the projected ended up good, we did get the piano to work, albeit to a minimal extent. Yet, the most important thing was that despite all the difficulty we ran into, we still had fun.

Appendix

```
`default_nettype none

//////////////////////////////////////////////////////////////////////
////////
//
// Switch Debounce Module
//
//////////////////////////////////////////////////////////////////////
////////

module debounce (
  input wire reset, clock, noisy,
  output reg clean
);
  reg [18:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
    else if (noisy != new) begin
      // noisy input changed, restart the .01 sec clock
      new <= noisy;
      count <= 0;
    end
    else if (count == 270000)
      // noisy input stable for .01 secs, pass it along!
      clean <= new;
    else
      // waiting for .01 sec to pass
      count <= count+1;

endmodule

//////////////////////////////////////////////////////////////////////
////////
//
// bi-directional monaural interface to AC97
//
//////////////////////////////////////////////////////////////////////
////////

module lab5audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [7:0] audio_in_data,
```

```
  input wire [7:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);

  wire [7:0] command_address;
  wire [15:0] command_data;
  wire command_valid;
  wire [19:0] left_in_data, right_in_data;
  wire [19:0] left_out_data, right_out_data;

  // wait a little before enabling the AC97 codec
  reg [9:0] reset_count;
  always @(posedge clock_27mhz) begin
    if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
    end else if (reset_count == 1023)
      audio_reset_b = 1'b1;
    else
      reset_count = reset_count+1;
  end

  wire ac97_ready;
  ac97 ac97(.ready(ac97_ready),
            .command_address(command_address),
            .command_data(command_data),
            .command_valid(command_valid),
            .left_data(left_out_data), .left_valid(1'b1),
            .right_data(right_out_data), .right_valid(1'b1),
            .left_in_data(left_in_data), .right_in_data(right_in_data)
,
            .ac97_sdata_out(ac97_sdata_out),
            .ac97_sdata_in(ac97_sdata_in),
            .ac97_synch(ac97_synch),
            .ac97_bit_clock(ac97_bit_clock));

  // ready: one cycle pulse synchronous with clock_27mhz
  reg [2:0] ready_sync;
  always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0],
ac97_ready};
  assign ready = ready_sync[1] & ~ready_sync[2];

  reg [7:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
```

```
  assign audio_in_data = left_in_data[19:12];
  assign left_out_data = {out_data, 12'b000000000000};
  assign right_out_data = left_out_data;

  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
  reg [7:0] bit_count;

  reg [19:0] l_cmd_addr;
  reg [19:0] l_cmd_data;
  reg [19:0] l_left_data, l_right_data;
  reg l_cmd_v, l_left_v, l_right_v;

  initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";

    bit_count <= 8'h00;
    // synthesis attribute init of bit_count is "0000";
    l_cmd_v <= 1'b0;
    // synthesis attribute init of l_cmd_v is "0";
    l_left_v <= 1'b0;
    // synthesis attribute init of l_left_v is "0";
```

```
  l_right_v <= 1'b0;
  // synthesis attribute init of l_right_v is "0";

  left_in_data <= 20'h00000;
  // synthesis attribute init of left_in_data is "00000";
  right_in_data <= 20'h00000;
  // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
  // Generate the sync signal
  if (bit_count == 255)
    ac97_synch <= 1'b1;
  if (bit_count == 15)
    ac97_synch <= 1'b0;

  // Generate the ready signal
  if (bit_count == 128)
    ready <= 1'b1;
  if (bit_count == 2)
    ready <= 1'b0;

  // Latch user data at the end of each frame. This ensures that the
  // first frame after reset will be empty.
  if (bit_count == 255) begin
    l_cmd_addr <= {command_address, 12'h000};
    l_cmd_data <= {command_data, 4'h0};
    l_cmd_v <= command_valid;
    l_left_data <= left_data;
    l_left_v <= left_valid;
    l_right_data <= right_data;
    l_right_v <= right_valid;
  end

  if ((bit_count >= 0) && (bit_count <= 15))
    // Slot 0: Tags
    case (bit_count[3:0])
      4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
      4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
      4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
      4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
      4'h4: ac97_sdata_out <= l_right_v; // Right data valid
      default: ac97_sdata_out <= 1'b0;
    endcase
  else if ((bit_count >= 16) && (bit_count <= 35))
    // Slot 1: Command address (8-bits, left justified)
    ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
  else if ((bit_count >= 36) && (bit_count <= 55))
    // Slot 2: Command data (16-bits, left justified)
    ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
```

```
    else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;
  end // always @ (posedge ac97_bit_clock)

  always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
  end
endmodule

// issue initialization commands to AC97
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
  input wire [4:0] volume,
  input wire [2:0] source
);
  reg [23:0] command;

  reg [3:0] state;
  initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
  end

  assign command_address = command[23:16];
  assign command_data = command[15:0];

  wire [4:0] vol;
```

```
  assign vol = 31-volume;  // convert to attenuation

  always @(posedge clock) begin
    if (ready) state <= state+1;

    case (state)
      4'h0: // Read ID
        begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
        end
      4'h1: // Read ID
        command <= 24'h80_0000;
      4'h3: // headphone volume
        command <= { 8'h04, 3'b000, vol, 3'b000, vol };
      4'h5: // PCM volume
        command <= 24'h18_0808;
      4'h6: // Record source select
        command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
      4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
      4'h9: // set +20db mic gain
        command <= 24'h0E_8048;
      4'hA: // Set beep volume
        command <= 24'h0A_0000;
      4'hB: // PCM out bypass mix1
        command <= 24'h20_8000;
      default:
        command <= 24'h80_0000;
    endcase // case(state)
  end // always @ (posedge clock)
endmodule // ac97commands

////////////////////////////////////////////////////////////////////////
////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
////////////////////////////////////////////////////////////////////////
////////

module tone750hz (
  input wire clock,
  input wire ready,
  input wire restart,
  output reg [19:0] pcm_data
);
   reg [8:0] index;

   initial begin
```

```
   index <= 8'h00;
   // synthesis attribute init of index is "00";
   pcm_data <= 20'h00000;
   // synthesis attribute init of pcm_data is "00000";
end

always @(posedge clock) begin
    if (restart) index <= 0;
    else if (ready) index <= index+1;
end

// one cycle of a sinewave in 64 20-bit samples
always @(index) begin
   case (index[5:0])
     6'h00: pcm_data <= 20'h00000;
     6'h01: pcm_data <= 20'h0C8BD;
     6'h02: pcm_data <= 20'h18F8B;
     6'h03: pcm_data <= 20'h25280;
     6'h04: pcm_data <= 20'h30FBC;
     6'h05: pcm_data <= 20'h3C56B;
     6'h06: pcm_data <= 20'h471CE;
     6'h07: pcm_data <= 20'h5133C;
     6'h08: pcm_data <= 20'h5A827;
     6'h09: pcm_data <= 20'h62F20;
     6'h0A: pcm_data <= 20'h6A6D9;
     6'h0B: pcm_data <= 20'h70E2C;
     6'h0C: pcm_data <= 20'h7641A;
     6'h0D: pcm_data <= 20'h7A7D0;
     6'h0E: pcm_data <= 20'h7D8A5;
     6'h0F: pcm_data <= 20'h7F623;
     6'h10: pcm_data <= 20'h7FFFF;
     6'h11: pcm_data <= 20'h7F623;
     6'h12: pcm_data <= 20'h7D8A5;
     6'h13: pcm_data <= 20'h7A7D0;
     6'h14: pcm_data <= 20'h7641A;
     6'h15: pcm_data <= 20'h70E2C;
     6'h16: pcm_data <= 20'h6A6D9;
     6'h17: pcm_data <= 20'h62F20;
     6'h18: pcm_data <= 20'h5A827;
     6'h19: pcm_data <= 20'h5133C;
     6'h1A: pcm_data <= 20'h471CE;
     6'h1B: pcm_data <= 20'h3C56B;
     6'h1C: pcm_data <= 20'h30FBC;
     6'h1D: pcm_data <= 20'h25280;
     6'h1E: pcm_data <= 20'h18F8B;
     6'h1F: pcm_data <= 20'h0C8BD;
     6'h20: pcm_data <= 20'h00000;
     6'h21: pcm_data <= 20'hF3743;
     6'h22: pcm_data <= 20'hE7075;
     6'h23: pcm_data <= 20'hDAD80;
```

```
        6'h24: pcm_data <= 20'hCF044;
        6'h25: pcm_data <= 20'hC3A95;
        6'h26: pcm_data <= 20'hB8E32;
        6'h27: pcm_data <= 20'hAECC4;
        6'h28: pcm_data <= 20'hA57D9;
        6'h29: pcm_data <= 20'h9D0E0;
        6'h2A: pcm_data <= 20'h95927;
        6'h2B: pcm_data <= 20'h8F1D4;
        6'h2C: pcm_data <= 20'h89BE6;
        6'h2D: pcm_data <= 20'h85830;
        6'h2E: pcm_data <= 20'h8275B;
        6'h2F: pcm_data <= 20'h809DD;
        6'h30: pcm_data <= 20'h80000;
        6'h31: pcm_data <= 20'h809DD;
        6'h32: pcm_data <= 20'h8275B;
        6'h33: pcm_data <= 20'h85830;
        6'h34: pcm_data <= 20'h89BE6;
        6'h35: pcm_data <= 20'h8F1D4;
        6'h36: pcm_data <= 20'h95927;
        6'h37: pcm_data <= 20'h9D0E0;
        6'h38: pcm_data <= 20'hA57D9;
        6'h39: pcm_data <= 20'hAECC4;
        6'h3A: pcm_data <= 20'hB8E32;
        6'h3B: pcm_data <= 20'hC3A95;
        6'h3C: pcm_data <= 20'hCF044;
        6'h3D: pcm_data <= 20'hDAD80;
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule

//////////////////////////////////////////////////////////////////////
//////////
////
//// 6.111 FPGA Labkit -- Template Toplevel Module
////
//// For Labkit Revision 004
//// Created: October 31, 2004, from revision 003 file
//// Author: Nathan Ickes, 6.111 staff
////
//////////////////////////////////////////////////////////////////////
//////////

module lab5   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
          ac97_bit_clock,

          vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
          vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
```

```
          vga_out_vsync,

          tv_out_ycrcb, tv_out_reset_b, tv_out_clock,
tv_out_i2c_clock,
          tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
          tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

          tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
          tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
          tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
          tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

          ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
          ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

          ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
          ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

          clock_feedback_out, clock_feedback_in,

          flash_data, flash_address, flash_ce_b, flash_oe_b,
flash_we_b,
          flash_reset_b, flash_sts, flash_byte_b,

          rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

          mouse_clock, mouse_data, keyboard_clock, keyboard_data,

          clock_27mhz, clock1, clock2,

          disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
          disp_reset_b, disp_data_in,

          button0, button1, button2, button3, button_enter,
button_right,
          button_left, button_down, button_up,

          switch,

          led,

          user1, user2, user3, user4,

          daughtercard,

          systemace_data, systemace_address, systemace_ce_b,
          systemace_we_b, systemace_oe_b, systemace_irq,
systemace_mpbrdy,

          analyzer1_data, analyzer1_clock,
```

```
          analyzer2_data, analyzer2_clock,
          analyzer3_data, analyzer3_clock,
          analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
     vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data,
     tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
     tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
tv_in_aef,
     tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock,
tv_in_iso,
     tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;
```

```
   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter,
button_right,
     button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
       analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock,
analyzer4_clock;

   ////////////////////////////////////////////////////////////////
////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////
////////


   // Audio Input and Output
   assign beep= 1'b0;
   //lab5 assign audio_reset_b = 1'b0;
   //lab5 assign ac97_synch = 1'b0;
   //lab5 assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // VGA Output
   /*
   assign vga_out_red = 10'h0;
   assign vga_out_green = 10'h0;
   assign vga_out_blue = 10'h0;
   assign vga_out_sync_b = 1'b1;
```

```
   assign vga_out_blank_b = 1'b1;
   assign vga_out_pixel_clock = 1'b0;
   assign vga_out_hsync = 1'b0;
   assign vga_out_vsync = 1'b0;
   */

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
   assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs
//   assign ram0_data = 36'hZ;
//   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
//   assign ram0_clk = 1'b0;
   assign ram0_clk = clock_27mhz;
//   assign ram0_cen_b = 1'b1;
   assign ram0_cen_b = 1'b0;
//   assign ram0_ce_b = 1'b1;
   assign ram0_ce_b = 1'b0;
//   assign ram0_oe_b = 1'b1;
   assign ram0_oe_b = 1'b0;
//   assign ram0_we_b = 1'b1;
//   assign ram0_bwe_b = 4'hF;
   assign ram0_bwe_b = 4'h0;

   //assign ram1_data = 36'hZ;
   //assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = clock_27mhz;
```

```
   assign ram1_cen_b = 1'b0;
   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
   //assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'h0;
   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are
inputs

   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab5 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   //assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;
```

```
   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   //lab5 assign analyzer1_data = 16'h0;
   //lab5 assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   //lab5 assign analyzer3_data = 16'h0;
   //lab5 assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

//   wire [7:0] from_ac97_data, to_ac97_data;
//   wire ready;

   ////////////////////////////////////////////////////////////////////
////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high
reset
   // signal that remains high for 16 clock cycles after configuration
finishes
   // and the FPGA's internal clocks begin toggling.
   //
   ////////////////////////////////////////////////////////////////////
////////
   wire reset;
   //SRL16 #(.INIT(16'hFFFF))
reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                                      //.A0(1'b1), .A1(1'b1), .A2(1'b1)
, .A3(1'b1));

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce
bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),.clean(vup));
   debounce
bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),.clean(vd
```

```
own));
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
   if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
   if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end

wire clock_54mhz_unbuf,clock_54mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_54mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 20
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_54mhz),.I(clock_54mhz_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_54mhz), .Q(power_on_reset),
         .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

  wire clk_27mhz;
   // AC97 driver
   lab5audio a(clock_27mhz, reset, volume, from_ac97_data,
to_ac97_data, ready,
          audio_reset_b, ac97_sdata_out, ac97_sdata_in,
          ac97_synch, ac97_bit_clock);

   // push ENTER button to record, release to playback
   wire playback;
   debounce
benter(.reset(reset),.clock(clock_27mhz),.noisy(button_enter),.clean(p
layback));

   // switch 0 up for filtering, down for no filtering
   wire filter;
   debounce
sw0(.reset(reset),.clock(clk_27mhz),.noisy(switch[0]),.clean(filter));

   // light up LEDs when recording, show volume during playback.
   // led is active low
   //assign led = playback ? ~{filter,2'b00, volume} : ~{filter,
7'hFF};
   //assign led[7:2] = 6'b111111;
   wire button0c, button1c, button2c, button3c, button_enterc,
```

```
button_rightc,
     button_leftc, button_downc, button_upc;

   debounce
debouncing8(.clock(clk_27mhz),.reset(reset),.noisy(~button_left),.clea
n(button_leftc));
   debounce
debouncing7(.clock(clk_27mhz),.reset(reset),.noisy(~button_up),.clean(
button_upc));
   debounce
debouncing6(.clock(clk_27mhz),.reset(reset),.noisy(~button_down),.clea
n(button_downc));
   debounce
debouncing5(.clock(clk_27mhz),.reset(reset),.noisy(~button_right),.cle
an(button_rightc));

   debounce
debouncing4(.clock(clk_27mhz),.reset(reset),.noisy(~button_enter),.cle
an(button_enterc));
   debounce
debouncing3(.clock(clk_27mhz),.reset(reset),.noisy(~button3),.clean(bu
tton3c));
   debounce
debouncing2(.clock(clk_27mhz),.reset(reset),.noisy(~button2),.clean(bu
tton2c));
   debounce
debouncing1(.clock(clk_27mhz),.reset(reset),.noisy(~button1),.clean(bu
tton1c));
   debounce
debouncing(.clock(clk_27mhz),.reset(reset),.noisy(~button0),.clean(but
ton0c));

   wire [16:0] key_num;
   //replace with Liam's sensor data
   assign key_num[16:0] =
{button_leftc,button_upc,button_downc,button_rightc,button_enterc,butt
on3c&~button2c,
button2c&~button3c,button3c&button2c,button0c,switch[7:5],user3[4:0]};
   // record module

   wire [16:0] key_num_fsm;
   wire [1:0] state;

   /*
   sound sound_mod(.clock(clock_27mhz), .reset(reset), .ready(ready),
             .playback(button_enterc), .record(button3c),
             .we_ZBT(ram0_we_b), .data_in(ram0_data),
             .data_out(ram0_data), .address(ram0_address),
             .key_num(key_num[16:0]),.key_num_fsm(key_num_fsm[16:0]),
             .state(state[1:0]),.to_ac97_data(to_ac97_data));
```

```
   */
   wire calibrating;
   assign calibrating = switch[0];
   wire [7:0]to_ac97_data2;

   fsm r(.clock(clk_27mhz), .reset(reset), .ready(ready),
              .playback(button1c), .record(button0c),
              .we_ZBT0(ram0_we_b), .data_in0(ram0_data),.calibrating(c
alibrating&switch[6]),
              .data_out0(ram0_data), .address0(ram0_address),
              .we_ZBT1(ram1_we_b), .data_in1(ram1_data),
              .data_out1(ram1_data), .address1(ram1_address),
              .key_num(key_num[16:0]),.key_num_fsm(key_num_fsm[16:0]),
              .from_ac97_data(from_ac97_data),.to_ac97_data(to_ac97_da
ta2),
              .state(state[1:0]),.led(led[7:0]));
   wire b0, b1, b2, b3, enter, right, left, down, up;

   // use FPGA's digital clock manager to produce a
   // 54MHz clock (actually 64.8MHz)

  clock_divider
c(.clk_54mhz(clock_54mhz),.restart(reset),.clk_27mhz(clk_27mhz));


   wire note_ready = 1;
   visual
vmod(.clock_54mhz(clock_54mhz), .key_num(key_num_fsm), .note_ready(not
e_ready),
               .state(state[1:0]),.reset(reset), .switch(switch), .vga
_out_red(vga_out_red),
               .vga_out_green(vga_out_green), .vga_out_blue(vga_out_bl
ue),
               .vga_out_sync_b(vga_out_sync_b), .vga_out_blank_b(vga_o
ut_blank_b),
               .vga_out_pixel_clock(vga_out_pixel_clock), .vga_out_hsy
nc(vga_out_hsync),
               .vga_out_vsync(vga_out_vsync));

   //assign led[1:0] = ~state[1:0];

   assign analyzer3_clock = clock_27mhz;
   assign analyzer3_data = {8'b0,to_ac97_data};




   play_sound
player(.clock(clk_27mhz),.key_num(key_num_fsm[16:0]),.shift(switch[1]&
switch[0]),.state(state[1:0]),
```

```
   .to_ac97_data2(to_ac97_data2),.to_ac97_data(to_ac97_data));

   // output useful things to the logic analyzer connectors
   assign analyzer1_clock = ac97_bit_clock;
   assign analyzer1_data[0] = audio_reset_b;
   assign analyzer1_data[1] = ac97_sdata_out;
   assign analyzer1_data[2] = ac97_sdata_in;
   assign analyzer1_data[3] = ac97_synch;
   assign analyzer1_data[15:4] = 0;

   //assign analyzer3_clock = ready;
   //assign analyzer3_data = {from_ac97_data, to_ac97_data};
endmodule

//////////////////////////////////////////////////////////////////////
////////
//
// Record/playback
//
//////////////////////////////////////////////////////////////////////
////////

module fsm #(parameter LOGSIZE = 19,WIDTH = 36,MAX_MEM_ADDR =
2**LOGSIZE - 1,

CALIBRATION = 3, PLAYBACK = 2, RECORD = 1, USER = 0, RECORDING_LEN =
32768)

(
  input wire clock,             // 27mhz system clock
  input wire reset,                // 1 to reset to initial state
  input wire playback,
  input wire record,
  input wire ready,                // 1 when AC97 data is available
  input wire calibrating,
  output wire [7:0] led,
  //input wire filter,               // 1 when using low-pass filter
  input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
  output wire [7:0] to_ac97_data,    // 8-bit PCM data to headphone
  // ZBT related outputs
  output reg we_ZBT0,               // ~WE for ZBT
  input wire [35:0] data_in0,    // audio data for writing to ZBT
  output reg [35:0] data_out0,      // audio data from reading ZBT
  output reg [18:0] address0,
  output wire we_ZBT1,              // ~WE for ZBT
  input wire [35:0] data_in1,    // audio data for writing to ZBT
  output wire [35:0] data_out1,     // audio data from reading ZBT
  output wire [18:0] address1,    // ZBT address reading from &
writing to
  input wire [16:0] key_num,
```

```
  output reg [16:0] key_num_fsm,
  output reg [1:0] state
  //output reg done
);

   reg [LOGSIZE-1:0] max_used_mem;
   reg [2:0] counter;
   reg old_playback;
   reg [7:0] filter_input;
   wire [17:0] filter_output;
   reg [4:0] note_num;

   //help with pressing the record button twice to switch into user
and record mode
   reg switched_to_record;
   reg switched_to_user;
   reg [18:0] address_in;
   wire recording_done;
   wire done_prev;

   recorder
r(.clock(clock),.reset(reset),.record(record),.ready(ready&state==2'b1
1),.address_in(address_in),
               .from_ac97_data(from_ac97_data),.to_ac97_data(to_ac97_d
ata),.we_ZBT(we_ZBT1),.data_in(data_in1),
               .data_out(data_out1),.address(address1),.led(led),.reco
rding_done(recording_done),
               .done_prev(done_prev));

   always @ (posedge clock) begin
      if (ready) begin
         case (state)
            USER: begin
               if (calibrating) begin
                  state <= CALIBRATION;
                  address_in <= 0;
                  note_num <= 0;
               end
               if (~record & ~playback) switched_to_user <= 1;
               if (record & switched_to_user) begin
                  state <= RECORD;
                  address0 <= 0;
                  counter <= 0;
                  switched_to_record <= 0;
                  switched_to_user <= 0;
               end
               else if (playback) begin
                  state <= PLAYBACK;
                  address0 <= 0;
                  counter <= 0;
```

```
                switched_to_user <= 0;
            end
            //else if (key_num[16:0] == 0) begin
            //data_out <= 8'b0;
            //end
            // eventually make an else if note ready
            else begin
            key_num_fsm[16:0] <= key_num[16:0];
            end

        end
        RECORD: begin
            if (calibrating) begin
                state <= CALIBRATION;
                address_in <= 0;
                note_num <= 0;
            end
            if (playback) begin
                state <= PLAYBACK;
                address0 <= 0;
                counter <= 0;
            end
            //else if (key_num[16:0] == 0) begin
            //data_out <= 8'b0;
            //data_out[16:0] <= key_num[16:0];
            //end
            // eventually make an else if note ready
            else begin
                key_num_fsm[16:0] <= key_num[16:0];
                data_out0[16:0] <= key_num[16:0];
            end
            if (~record) begin
                switched_to_record <= 1;
                max_used_mem <= 0;
            end
            // for ZBT
            max_used_mem <= address0;
            if (we_ZBT0 == 0) address0 <= address0 + 1;
            //if memory full or rewgo back to user mode
            if (address0 == MAX_MEM_ADDR|(switched_to_record &
record)) begin
                we_ZBT0 <= 1;
                state <= USER;
                address0 <= 0;
                counter <= 0;
            end
            else if (counter == 0) we_ZBT0 <= 0;
            else we_ZBT0 <= 1;
        end
```

```
            PLAYBACK: begin
               // for ZBT
               //if (address == max_used_mem) address <= 0;
               if (calibrating) begin
                  state <= CALIBRATION;
                  address_in <= 0;
                  note_num <= 0;
               end
               if (address0 >= max_used_mem) state <= USER;
               else if (record) begin
                  state <= RECORD;
                  address0 <= 0;
                  counter <= 0;
                  switched_to_record <= 0;
               end
               else if (counter == 0) address0 <= address0 + 1;

               key_num_fsm[16:0] <= data_in0[16:0];

               data_out0 <= 36'hZ;
               we_ZBT0 <= 1;

            end

            CALIBRATION: begin
               if (~calibrating) state <= USER;

               //display 1st note by setting key_num equal to it
               //start with 1_0000_0000_0000_0000 and do a bit shift
for whatever note you're on
               //starting address is 0
               //instantiate recorder
               //when done shift to new note number and make that the
new address. 0 + 32768*note_num
               //once note number is greater than 16 switch back to
user mode

               key_num_fsm[16:0] <= 17'b1_0000_0000_0000_0000 >>
note_num;


               if (note_num >= 16) state <= USER;

               else if (recording_done & ~done_prev) begin
                  //key_num_fsm[16:0] <= 17'b0_0000_0000_0000_0001 >>
note_num;
                   //once you've saved all the notes go back to user
mode
                  note_num <= note_num + 1;
                  address_in[18:0] <= (note_num+1)<< 14;
```

```
               end



            end
          default: state <= 0;
          endcase
          counter <= counter + 1;
       end
   end
endmodule

////////////////////////////////////////////////////////////////////////
////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
////////////////////////////////////////////////////////////////////////
////////

module mybram #(parameter LOGSIZE=14, WIDTH=1)
               (input wire [LOGSIZE-1:0] addr,
                input wire clk,
                input wire [WIDTH-1:0] din,
                output reg [WIDTH-1:0] dout,
                input wire we);
   // let the tools infer the right number of BRAMs
   (* ram_style = "block" *)
   reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
   always @(posedge clk) begin
     if (we) mem[addr] <= din;
     dout <= mem[addr];
   end
endmodule

////////////////////////////////////////////////////////////////////////
////////
//
// 31-tap FIR filter, 8-bit signed data, 10-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
//
////////////////////////////////////////////////////////////////////////
////////
```

```
module fir31(
  input wire clock,reset,ready,
  input wire signed [7:0] x,
  output reg signed [17:0] y
   );

   reg signed [7:0] sample [31:0];
   reg [4:0] offset = 0;
   reg signed [17:0] accumulator;
   reg [4:0] index;
   wire signed [9:0] coeff;
   integer k = 0;
   wire [4:0] index2 = offset - index;
   reg calculating = 0;

   coeffs31 c31(.index(index), .coeff(coeff));

   initial
   begin
      for (k = 0; k < 32; k = k + 1)
      begin
         sample[k] = 0;
      end
   end

   always @(posedge clock)
   begin
      // ready logic
      if (ready)
      begin
         accumulator <= 0;
         index <= 0;
         offset <= offset + 1;
         sample[offset] <= x;
         calculating <= 1;
      end

      // reset logic
      if (reset)
      begin
         accumulator <= 0;
         index <= 0;
         calculating <= 0;
      end

      if (calculating)
      begin
         index <= index + 1;
         // calculating
```

```
         if (index < 31) accumulator <= accumulator + (coeff *
sample[index2]);
         // finished calculating
         else
         begin
            y <= accumulator;
            calculating <= 0;
         end
      end
   end
endmodule

//////////////////////////////////////////////////////////////////////
////////
//
// Coefficients for a 31-tap low-pass FIR filter with Wn=.125 (eg,
3kHz for a
// 48kHz sample rate).  Since we're doing integer arithmetic, we've
scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,.125)*1024)
//
//////////////////////////////////////////////////////////////////////
////////

module coeffs31(
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -10'sd1;
      5'd1:  coeff = -10'sd1;
      5'd2:  coeff = -10'sd3;
      5'd3:  coeff = -10'sd5;
      5'd4:  coeff = -10'sd6;
      5'd5:  coeff = -10'sd7;
      5'd6:  coeff = -10'sd5;
      5'd7:  coeff = 10'sd0;
      5'd8:  coeff = 10'sd10;
      5'd9:  coeff = 10'sd26;
      5'd10: coeff = 10'sd46;
      5'd11: coeff = 10'sd69;
      5'd12: coeff = 10'sd91;
      5'd13: coeff = 10'sd110;
      5'd14: coeff = 10'sd123;
      5'd15: coeff = 10'sd128;
      5'd16: coeff = 10'sd123;
      5'd17: coeff = 10'sd110;
```

```
      5'd18: coeff = 10'sd91;
      5'd19: coeff = 10'sd69;
      5'd20: coeff = 10'sd46;
      5'd21: coeff = 10'sd26;
      5'd22: coeff = 10'sd10;
      5'd23: coeff = 10'sd0;
      5'd24: coeff = -10'sd5;
      5'd25: coeff = -10'sd7;
      5'd26: coeff = -10'sd6;
      5'd27: coeff = -10'sd5;
      5'd28: coeff = -10'sd3;
      5'd29: coeff = -10'sd1;
      5'd30: coeff = -10'sd1;
      default: coeff = 10'hXXX;
    endcase
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:01:12 12/11/2017
// Design Name:
// Module Name:    clock_divider
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module clock_divider(
    input clk_54mhz,
    input restart,
    output reg clk_27mhz
    );
    reg counter = 0;
        always @(posedge clk_54mhz) begin
            counter <= counter + 1;
            if (restart) counter <= 0;
            else if (counter == 1 ) begin
                clk_27mhz <= 1;
            end
            else clk_27mhz <= 0;
         end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    13:17:07 12/10/2017
// Design Name:
// Module Name:    play_instrument
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module play_instrument#(RECORDING_LEN = 32768)

(
   input wire clock,            // 27mhz system clock
   input wire reset,               // 1 to reset to initial state
   input wire playback,            // 1 for playback, 0 for record
   input wire ready,               // 1 when AC97 data is available
   input wire [18:0] address_in,
   input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
   output reg [7:0] to_ac97_data,  // 8-bit PCM data to headphone
   output reg we_ZBT,
   input wire [35:0] data_in, //audio data for writing to ZBT
   output reg [35:0] data_out, //audio data from reading ZBT
   output reg [18:0] address,   //ZBT address
   output reg [7:0] led);

   reg started = 0;
   //wire [19:0] tone;
   reg [1:0] counter = 0;
   //reg send = 0;

   reg [LOGSIZE-1:0] highest_address;

   always @ (posedge clock) begin
      //led<= 8'b11111111;
      mode <= playback;
      if (~started) begin
          address <= start_address;
```

```
            counter <= 0;
        end
        we_ZBT <= 1; //don't write to memory just read it
        //led[1:0]<= ~2'b10;
         if (ready)begin
            counter <= counter + 1;
            if (address == address_in + RECORDING_LEN) begin
               counter <= 0;
               address <= address_in;
            end

            else if (counter == 3) begin
               //if (counter == 1) begin
               address <= address + 1;
               to_ac97_data <= data_in;
            end
            data_out <= 36'hZ;
         end
    end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:42:22 11/20/2017
// Design Name:
// Module Name:    play_sound
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module play_sound(
    input [16:0] key_num,
    input clock,
    input [7:0] to_ac97_data2,
    input [1:0] state,
    input shift,
    output reg [7:0] to_ac97_data
    );
    wire new_ready;
    wire [19:0] tone;
    wire [19:0] tone2;
    reg restart;
    reg [16:0] old_key_num;

    ready_generator
generator(.clk(clock),.restart(restart),.shift(shift),.key_num(key_num
),.ready(new_ready));
    //ready_generator
generator2(.clk(clock),.restart(restart),.key_num(17'b0_0000_0000_0000
_1000),.ready(new_ready2));
    tone750hz
xxx(.clock(clock),.ready(new_ready),.restart(restart),.pcm_data(tone))
;
    //tone750hz
xxx2(.clock(clock),.ready(new_ready2),.restart(restart),.pcm_data(tone
2));

    always @(posedge clock) begin
```

```
        if (key_num[16:0] != old_key_num[16:0]) restart <= 1;
        else restart <= 0;
        old_key_num <= key_num[16:0];

        if (state == 3) to_ac97_data[7:0] <= to_ac97_data2;
        else if (key_num[16:0] == 0) to_ac97_data[7:0] <= 8'b0;
        else to_ac97_data[7:0] <= tone[19:12];
        //else to_ac97_data[7:0] <= {0,tone[19:13]}+{0,tone2[19:13]};
    end

    //assign restart = (key_num[16:0] != old_key_num[16:0] ) ? 1:0;

    //assign old_key_num = key_num[16:0];

    //assign to_ac97_data = (key_num[16:0] == 0) ? 8'b0:
(8'b0+tone[19:13]+tone2[19:13]);

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:43:35 11/20/2017
// Design Name:
// Module Name:    ready_generator
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ready_generator

(input clk,
    input restart,
    input shift,
    input [16:0] key_num,
    output reg ready
    );
    reg [11:0] divider_num = 0;

    reg [11:0] counter = 0;
        always @(posedge clk) begin
           //casex allows x's to be don't cares
           casex (key_num)
               /*
               17'b1_xxxx_xxxx_xxxx_xxxx: divider_num <= 1612; //C
               17'b0_1xxx_xxxx_xxxx_xxxx: divider_num <= 1522; //C#
               17'b0_01xx_xxxx_xxxx_xxxx: divider_num <= 1437; //D
               17'b0_001x_xxxx_xxxx_xxxx: divider_num <= 1356; //Eb
               17'b0_0001_xxxx_xxxx_xxxx: divider_num <= 1280; //E
               17'b0_0000_1xxx_xxxx_xxxx: divider_num <= 1208; //F

               17'b0_0000_01xx_xxxx_xxxx: divider_num <= 1140; //F#
               17'b0_0000_001x_xxxx_xxxx: divider_num <= 1076; //G
               17'b0_0000_0001_xxxx_xxxx: divider_num <= 1016; //G#

               17'b0_0000_0000_1xxx_xxxx: divider_num <= 959; //A
               17'b0_0000_0000_01xx_xxxx: divider_num <= 905; //Bb
```

```
                17'b0_0000_0000_001x_xxxx: divider_num <= 854; //B
                17'b0_0000_0000_0001_xxxx: divider_num <= 806; //C high

                17'b0_0000_0000_0000_1xxx: divider_num <= 761; //C#
                17'b0_0000_0000_0000_01xx: divider_num <= 718; //D
                17'b0_0000_0000_0000_001x: divider_num <= 678; //Eb
                17'b0_0000_0000_0000_0001: divider_num <= 640; //E
                */
                17'b1_xxxx_xxxx_xxxx_xxxx: divider_num <= 1612 >>
shift; //C
                17'b0_1xxx_xxxx_xxxx_xxxx: divider_num <= 1522 >>
shift; //C#
                17'b0_01xx_xxxx_xxxx_xxxx: divider_num <= 1437 >>
shift; //D
                17'b0_001x_xxxx_xxxx_xxxx: divider_num <= 1356 >>
shift; //Eb
                17'b0_0001_xxxx_xxxx_xxxx: divider_num <= 1280 >>
shift; //E
                17'b0_0000_1xxx_xxxx_xxxx: divider_num <= 1208 >>
shift; //F

                17'b0_0000_01xx_xxxx_xxxx: divider_num <= 1140 >>
shift; //F#
                17'b0_0000_001x_xxxx_xxxx: divider_num <= 1076 >>
shift; //G
                17'b0_0000_0001_xxxx_xxxx: divider_num <= 1016 >>
shift; //G#

                17'b0_0000_0000_1xxx_xxxx: divider_num <= 959 >>
shift; //A
                17'b0_0000_0000_01xx_xxxx: divider_num <= 905 >>
shift; //Bb
                17'b0_0000_0000_001x_xxxx: divider_num <= 854 >>
shift; //B
                17'b0_0000_0000_0001_xxxx: divider_num <= 806 >>
shift; //C high

                17'b0_0000_0000_0000_1xxx: divider_num <= 761 >>
shift; //C#
                17'b0_0000_0000_0000_01xx: divider_num <= 718 >>
shift; //D
                17'b0_0000_0000_0000_001x: divider_num <= 678 >>
shift; //Eb
                17'b0_0000_0000_0000_0001: divider_num <= 640 >>
shift; //E
            endcase
            counter <= counter + 1;
            if (restart) counter <= 0;
            else if (counter >= divider_num) begin
                ready <= 1;
```

```
                counter <= 0;
            end
            else ready <= 0;
        end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:41:29 12/09/2017
// Design Name:
// Module Name:    recorder
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module recorder#(parameter LOGSIZE=19, WIDTH=36,MAX_MEM_ADDR =
2**LOGSIZE-1,RECORDING_LEN = 32768)

   (input wire clock,              // 27mhz system clock
   input wire reset,                // 1 to reset to initial state
   input wire record,
   input wire ready,                // 1 when AC97 data is available
   input wire [18:0] address_in,
   input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
   output reg [7:0] to_ac97_data,  // 8-bit PCM data to headphone
   output reg we_ZBT,
   input wire [35:0] data_in, //audio data for writing to ZBT
   output reg [35:0] data_out, //audio data from reading ZBT
   output reg [18:0] address,   //ZBT address
   output reg [7:0] led,
   output reg done_prev,
   output reg recording_done);

   // test: playback 750hz tone, or loopback using incoming data
   //reg [LOGSIZE-1:0] address;
   reg mode;
   //wire [19:0] tone;
   reg [1:0] counter = 0;
   //reg [10:3] counter2 = 0;
   reg send = 0;

   reg [LOGSIZE-1:0] highest_address;
   reg we;
```

```
   //reg [7:0] din = 8'b0;
   //wire [7:0] dout;
   //reg [7:0] filter_input = 8'b0;
   //wire [17:0] filter_output;

   //tone750hz xxx(.clock(clock),.ready(ready),.pcm_data(tone));


   //mybram #(.LOGSIZE(LOGSIZE),.WIDTH(WIDTH))
   //
record_samples(.addr(address),.clk(clock),.we(we),.din(din),.dout(dout
));

   //fir31
filter2(.clock(clock),.reset(reset),.ready(ready),.x(filter_input),.y(
filter_output));

   reg started = 0;

   always @ (posedge clock) begin
      //led<= 8'b11111111;
      done_prev <= recording_done;
      if (~started) begin
          address <= address_in;
          counter <= 0;
          started <= 1;
      end
   //Record Mode
     if ((address >= (address_in + RECORDING_LEN)) & ~recording_done)
begin
        recording_done <= 1;
         we_ZBT <= 1; //don't write to memory
      end
      else recording_done <= 0;
      if (ready & record) begin
         to_ac97_data <= from_ac97_data;
         data_out <= from_ac97_data;
         counter <= counter + 1;
         if (counter == 3) begin
            we_ZBT <= 0; //write to memory
            address <= address + 1;
            led[7:0] <= address[18:12];
            highest_address <= address + 1;
         end
         else we_ZBT <= 1; // don't write to memory if not 8th sample
      end
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:37:46 12/05/2017
// Design Name:
// Module Name:    debounce
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
//////////////////////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:37:46 12/05/2017
// Design Name:
// Module Name:    debounce
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
//////////////////////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:17:57 12/05/2017
// Design Name:
// Module Name:    visual
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module visual(
   input clock_54mhz,
   input [16:0] key_num,
   input [1:0] state,
   input note_ready, reset,
   input [7:0] switch,

   output [7:0] vga_out_red, vga_out_green, vga_out_blue,
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
         vga_out_hsync, vga_out_vsync
   );

   ////////////////////////////////////////////////////////////////////
   //
   // visual module
   //
   ////////////////////////////////////////////////////////////////////

//   // use FPGA's digital clock manager to produce a
//   // 65MHz clock (actually 64.8MHz)
//   wire clock_65mhz_unbuf,clock_65mhz;
//   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
//   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
//   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 20
//   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
//   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
```

```
//   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
//   wire power_on_reset;    // remain high for first 16 clocks
//   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
//         .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
//   defparam reset_sr.INIT = 16'hFFFF;
//
//   // RESET button is user reset
//   wire reset = power_on_reset;

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(.vclock(clock_54mhz),.hcount(hcount),.vcount(vcount),
              .hsync(hsync),.vsync(vsync),.blank(blank));

   // debounce all buttons
// wire b0, b1, b2, b3, enter, right, left, down, up;
// debounce
db0(.reset(reset), .clock(clock_65mhz), .noisy(~button0), .clean(b0));
// debounce
db1(.reset(reset), .clock(clock_65mhz), .noisy(~button1), .clean(b1));
// debounce
db2(.reset(reset), .clock(clock_65mhz), .noisy(~button2), .clean(b2));
// debounce
db3(.reset(reset), .clock(clock_65mhz), .noisy(~button3), .clean(b3));
// debounce
db_enter(.reset(reset), .clock(clock_65mhz), .noisy(~button_enter), .c
lean(enter));
// debounce
db_right(.reset(reset), .clock(clock_65mhz), .noisy(~button_right), .c
lean(right));
// debounce
db_left(.reset(reset), .clock(clock_65mhz), .noisy(~button_left), .cle
an(left));
// debounce
db_down(.reset(reset), .clock(clock_65mhz), .noisy(~button_down), .cle
an(down));
// debounce
db_up(.reset(reset), .clock(clock_65mhz), .noisy(~button_up), .clean(u
p));

   // feed XVGA signals to piano
   wire [23:0] pixel;
   wire phsync,pvsync,pblank;
// wire [16:0] key_num = {left, up, down, right, enter, b3, b2, b1,
b0, switch};
// wire note_ready = 1;
```

```
   keystoning
      ks(.clk(clock_54mhz),.reset(reset),
      .hcount(hcount),.vcount(vcount),
      .hsync(hsync),.vsync(vsync),.blank(blank),
      .key_num(key_num), .note_ready(note_ready), .state(state),
      .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.keystoned_pixel
(pixel));

   // switch[1:0] selects which video generator to use:
   //  00: piano
   //  01: 1 pixel outline of active video area (adjust screen
controls)
   //  10: color bars
   //  11: piano
   reg [23:0] rgb;
   wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

   reg b,hs,vs;
   always @(posedge clock_54mhz) begin
//      if (switch[1:0] == 2'b01) begin
//    // 1 pixel outline of visible area (white)
//    hs <= hsync;
//    vs <= vsync;
//    b <= blank;
//    rgb <= {24{border}};
//      end else if (switch[1:0] == 2'b10) begin
//    // color bars
//    hs <= hsync;
//    vs <= vsync;
//    b <= blank;
//    rgb <= {{8{hcount[8]}}, {8{hcount[7]}}, {8{hcount[6]}}} ;
//      end else begin
         // default: piano
    hs <= phsync;
    vs <= pvsync;
    b <= pblank;
    rgb <= pixel;
//      end
   end

   assign vga_out_red = rgb[23:16];
   assign vga_out_green = rgb[15:8];
   assign vga_out_blue = rgb[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_54mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:59:10 12/05/2017
// Design Name:
// Module Name:    xvga
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
//////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current
line
            output reg [9:0] vcount,    // line number
            output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);
   assign hsyncon = (hcount == 1047);
   assign hsyncoff = (hcount == 1183);
   assign hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);
   assign vsyncon = hreset & (vcount == 776);
```

```
   assign vsyncoff = hreset & (vcount == 782);
   assign vreset = hreset & (vcount == 805);

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Zoe Klawans
//
// Create Date:    15:24:29 11/28/2017
// Design Name:
// Module Name:    keystoning
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module keystoning
   #(parameter WHITE_KEY_WIDTH = 99,
               SPACE = 4,
               PIANO_MIDDLE = (5 * WHITE_KEY_WIDTH) + (4 * SPACE) + 2,
               BOARD_HEIGHT = 10'd768,
               KEY_START_VERTICAL = BOARD_HEIGHT >> 2)
    (input clk,
    input reset,
    input [10:0] hcount,
    input [9:0] vcount,
    input hsync,
    input vsync,
    input blank,
    input [16:0] key_num,
    input note_ready,
    input [1:0] state,

    output phsync,
    output pvsync,
    output pblank,
    output [23:0] keystoned_pixel
    );

   parameter BLACK = 24'h00_00_00;
   parameter RED = 24'hFF_00_00;
   parameter GREEN = 24'h00_FF_00;

   reg [10:0] phcount;
```

```
   reg [9:0] pvcount;
   reg signed [10:0] x;
   wire [23:0] temp_pixel;

   always @(posedge clk)
   begin

      x <= hcount - PIANO_MIDDLE;
      phcount <= hcount - (((vcount - KEY_START_VERTICAL) * (x * -1))
>> 10);

   end

   piano #(.BOARD_WIDTH(11'd1024), .BOARD_HEIGHT(BOARD_HEIGHT),
         .WHITE_KEY_HEIGHT(BOARD_HEIGHT >>
2), .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH),
         .WHITE_KEY_START_HORIZONTAL(0), .KEY_START_VERTICAL(KEY_START
_VERTICAL),
         .BLACK_KEY_HEIGHT(BOARD_HEIGHT >> 3), .BLACK_KEY_WIDTH(60),
         .BLACK_KEY_START_HORIZONTAL(69), .SPACING(SPACE +
WHITE_KEY_WIDTH))

p(.vclock(clk), .reset(reset), .hcount(phcount), .vcount(vcount),
         .hsync(hsync), .vsync(vsync), .blank(blank), .key_num(key_num
),
         .note_ready(note_ready), .state(state), .phsync(phsync), .pvs
ync(pvsync),
         .pblank(pblank), .pixel(temp_pixel));

   assign keystoned_pixel = (hcount < 2) ? BLACK : temp_pixel;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Zoe Klawans
//
// Create Date:    19:49:38 11/21/2017
// Design Name:
// Module Name:    piano
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// generate piano
//
//////////////////////////////////////////////////////////////////////////////////

module piano
   #(parameter BOARD_WIDTH = 11'd1024,
               BOARD_HEIGHT = 10'd768,
               WHITE_KEY_HEIGHT = BOARD_HEIGHT >> 2,
               WHITE_KEY_WIDTH = 99,
               WHITE_KEY_START_HORIZONTAL = 0,
               KEY_START_VERTICAL = BOARD_HEIGHT >> 2,
               BLACK_KEY_HEIGHT = BOARD_HEIGHT >> 3,
               BLACK_KEY_WIDTH = 60,
               BLACK_KEY_START_HORIZONTAL = 69,
               SPACING = 103)
   (input vclock,  // 65MHz clock
   input reset,      // 1 to initialize module
   input [10:0] hcount, // horizontal index of current pixel (0..1023)
   input [9:0]    vcount, // vertical index of current pixel (0..767)
   input hsync,      // XVGA horizontal sync signal (active low)
   input vsync,      // XVGA vertical sync signal (active low)
   input blank,      // XVGA blanking (1 means output black pixel)
   input [16:0] key_num,   // keys pressed on piano
   input note_ready,    // key_num valid note(s) on piano
```

```
   input [1:0] state, // 00: user, 01: record, 10: playback

   output phsync, // piano's horizontal sync
   output pvsync, // piano's vertical sync
   output pblank, // piano's blanking
   output [23:0] pixel  // piano's pixel  // r=23:16, g=15:8, b=7:0
   );

   parameter PRESSED = 24'hFF_00_00;
   parameter WHITE = 24'hFF_FF_FF;
   parameter BLACK = 24'h00_00_00;
   parameter GREEN = 24'h00_FF_00;
   parameter BLUE = 24'h07_F8_FF;

   wire [23:0] c_pixel, db_pixel, d_pixel, eb_pixel, e_pixel, f_pixel,
gb_pixel,
               g_pixel, ab_pixel, a_pixel, bb_pixel, b_pixel,
high_c_pixel,
               high_db_pixel, high_d_pixel, high_eb_pixel,
high_e_pixel;

   wire [23:0] c_color, db_color, d_color, eb_color, e_color, f_color,
gb_color,
               g_color, ab_color, a_color, bb_color, b_color,
high_c_color,
               high_db_color, high_d_color, high_eb_color,
high_e_color;

   key_colors
kc(.reset(reset), .clk(vclock), .note_ready(note_ready), .key_num(key_
num),
         .c_color(c_color), .db_color(db_color), .d_color(d_color), .e
b_color(eb_color),
         .e_color(e_color), .f_color(f_color), .gb_color(gb_color), .g
_color(g_color),
         .ab_color(ab_color), .a_color(a_color), .bb_color(bb_color),
.b_color(b_color),
         .high_c_color(high_c_color), .high_db_color(high_db_color), .
high_d_color(high_d_color),
         .high_eb_color(high_eb_color), .high_e_color(high_e_color));

   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;

   reg [23:0] mode_color;
   wire [23:0] mode_pixel;

   always @(posedge vclock)
   begin
```

```
      case (state)
         0: mode_color <= BLACK;
         1: mode_color <= PRESSED;
         2: mode_color <= GREEN;
         default: mode_color <= BLUE;
      endcase

   end

   // Mode indicator blob
   circle #(.RADIUS(35))
mode(.x(800), .y(70), .hcount(hcount), .vcount(vcount), .color(mode_co
lor), .pixel(mode_pixel));

   // C key
   left_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      C(.x(WHITE_KEY_START_HORIZONTAL + (0 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(c_color), .pixel(c_pixe
l));

   // Db key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      Db(.x(BLACK_KEY_START_HORIZONTAL + (0 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(db_color), .pixel(db_pi
xel));

   // D key
   middle_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
                  .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDT
H(BLACK_KEY_WIDTH >> 1),
                  .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      D(.x(WHITE_KEY_START_HORIZONTAL + (1 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(d_color), .pixel(d_pixe
l));

   // Eb key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      Eb(.x(BLACK_KEY_START_HORIZONTAL + (1 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(eb_color), .pixel(eb_pi
xel));

   // E key
```

```
   right_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      E(.x(WHITE_KEY_START_HORIZONTAL + (2 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(e_color), .pixel(e_pixe
l));

   // F key
   left_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      F(.x(WHITE_KEY_START_HORIZONTAL + (3 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(f_color), .pixel(f_pixe
l));

   // Gb key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      Gb(.x(BLACK_KEY_START_HORIZONTAL + (3 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(gb_color), .pixel(gb_pi
xel));

   // G key
   middle_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
                .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDT
H(BLACK_KEY_WIDTH >> 1),
                .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      G(.x(WHITE_KEY_START_HORIZONTAL + (4 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(g_color), .pixel(g_pixe
l));

   // Ab key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      Ab(.x(BLACK_KEY_START_HORIZONTAL + (4 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(ab_color), .pixel(ab_pi
xel));

   // A key
   middle_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
                .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDT
H(BLACK_KEY_WIDTH >> 1),
                .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      A(.x(WHITE_KEY_START_HORIZONTAL + (5 *
SPACING)), .y(KEY_START_VERTICAL),
```

```
      .hcount(hcount), .vcount(vcount), .color(a_color), .pixel(a_pixe
l));

   // Bb key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      Bb(.x(BLACK_KEY_START_HORIZONTAL + (5 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(bb_color), .pixel(bb_pi
xel));

   // B key
   right_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      B(.x(WHITE_KEY_START_HORIZONTAL + (6 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(b_color), .pixel(b_pixe
l));

   // high C key
   left_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      high_C(.x(WHITE_KEY_START_HORIZONTAL + (7 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(high_c_color), .pixel(h
igh_c_pixel));

   // high Db key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      high_Db(.x(BLACK_KEY_START_HORIZONTAL + (7 * SPACING +
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(high_db_color), .pixel(
high_db_pixel));

   // high D key
   middle_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
                .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDT
H(BLACK_KEY_WIDTH >> 1),
                .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      high_D(.x(WHITE_KEY_START_HORIZONTAL + (8 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(high_d_color), .pixel(h
igh_d_pixel));

   // high Eb key
   blob #(.WIDTH(BLACK_KEY_WIDTH), .HEIGHT(BLACK_KEY_HEIGHT - 1))
      high_Eb(.x(BLACK_KEY_START_HORIZONTAL + (8 * SPACING +
```

```
2)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(high_eb_color), .pixel(
high_eb_pixel));

   // high E key
   right_key #(.WIDTH(WHITE_KEY_WIDTH), .HEIGHT(WHITE_KEY_HEIGHT),
               .BLACK_KEY_HEIGHT(BLACK_KEY_HEIGHT), .BLACK_KEY_WIDTH(B
LACK_KEY_WIDTH >> 1),
               .WHITE_KEY_WIDTH(WHITE_KEY_WIDTH))
      high_E(.x(WHITE_KEY_START_HORIZONTAL + (9 *
SPACING)), .y(KEY_START_VERTICAL),
      .hcount(hcount), .vcount(vcount), .color(high_e_color), .pixel(h
igh_e_pixel));

   assign pixel = c_pixel | db_pixel | d_pixel | eb_pixel | e_pixel |
f_pixel
                  | gb_pixel | g_pixel | ab_pixel | a_pixel | bb_pixel
| b_pixel
                  | high_c_pixel | high_db_pixel | high_d_pixel |
high_eb_pixel
                  | high_e_pixel | mode_pixel;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:03:35 11/21/2017
// Design Name:
// Module Name:    key_colors
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module key_colors(
    input reset,
    input note_ready,
    input [16:0] key_num,
    input clk,
    output reg [23:0] c_color, db_color, d_color, eb_color, e_color,
f_color,
            gb_color, g_color, ab_color, a_color, bb_color, b_color,
            high_c_color, high_db_color, high_d_color, high_eb_color,
high_e_color
    );

   parameter BLACK = 24'h00_00_00;
   parameter RED = 24'hFF_00_00;
   parameter WHITE = 24'hFF_FF_FF;

   always @(posedge clk)
   begin
      if (note_ready)
      begin

         if (key_num[16]) c_color <= RED;
         else c_color <= WHITE;

         if (key_num[15]) db_color <= RED;
         else db_color <= BLACK;

         if (key_num[14]) d_color <= RED;
```

```
            else d_color <= WHITE;

            if (key_num[13]) eb_color <= RED;
            else eb_color <= BLACK;

            if (key_num[12]) e_color <= RED;
            else e_color <= WHITE;

            if (key_num[11]) f_color <= RED;
            else f_color <= WHITE;

            if (key_num[10]) gb_color <= RED;
            else gb_color <= BLACK;

            if (key_num[9]) g_color <= RED;
            else g_color <= WHITE;

            if (key_num[8]) ab_color <= RED;
            else ab_color <= BLACK;

            if (key_num[7]) a_color <= RED;
            else a_color <= WHITE;

            if (key_num[6]) bb_color <= RED;
            else bb_color <= BLACK;

            if (key_num[5]) b_color <= RED;
            else b_color <= WHITE;

            if (key_num[4]) high_c_color <= RED;
            else high_c_color <= WHITE;

            if (key_num[3]) high_db_color <= RED;
            else high_db_color <= BLACK;

            if (key_num[2]) high_d_color <= RED;
            else high_d_color <= WHITE;

            if (key_num[1]) high_eb_color <= RED;
            else high_eb_color <= BLACK;

            if (key_num[0]) high_e_color <= RED;
            else high_e_color <= WHITE;

        end
    end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:38:22 10/03/2017
// Design Name:
// Module Name:    blob
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
//////////////////////////////////////////////////////////////////////////////////
module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64)           // default height: 64 pixels
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    input [23:0] color,
    output reg [23:0] pixel);

   always @ * begin
      if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y &&
vcount < (y+HEIGHT))) pixel = color;
      else pixel = 0;
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:43:24 11/09/2017
// Design Name:
// Module Name:    left_key
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module left_key
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               BLACK_KEY_HEIGHT = 64,  // default black key height: 64
pixels
               BLACK_KEY_WIDTH = 15,   // default black key width: 15
pixels
               WHITE_KEY_WIDTH = 90)   // default white key width: 90
pixels
   (input [10:0] x, hcount,
    input [9:0] y, vcount,
    input [23:0] color,
    output reg [23:0] pixel
    );

    always @ * begin
      if ((hcount >= (x+WHITE_KEY_WIDTH-BLACK_KEY_WIDTH)) && (vcount <
(y+BLACK_KEY_HEIGHT))) pixel = 0;
      else if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y &&
vcount < (y+HEIGHT))) pixel = color;
      else pixel = 0;
   end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    22:01:50 11/09/2017
// Design Name:
// Module Name:    middle_key
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module middle_key
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               BLACK_KEY_HEIGHT = 64,  // default black key height: 64
pixels
               BLACK_KEY_WIDTH = 15,   // default black key width: 15
pixels
               WHITE_KEY_WIDTH = 90)   // default white key width: 90
pixels
   (input [10:0] x, hcount,
    input [9:0] y, vcount,
    input [23:0] color,
    output reg [23:0] pixel
    );

    always @ * begin
      if (((hcount >= (x+WHITE_KEY_WIDTH-BLACK_KEY_WIDTH)) && (vcount
< (y+BLACK_KEY_HEIGHT))) ||
           ((hcount < (x+BLACK_KEY_WIDTH)) && (vcount < (y
+BLACK_KEY_HEIGHT)))) pixel = 0;
      else if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y &&
vcount < (y+HEIGHT))) pixel = color;
      else pixel = 0;
   end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    22:16:32 11/09/2017
// Design Name:
// Module Name:    right_key
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module right_key
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               BLACK_KEY_HEIGHT = 64,  // default black key height: 64
pixels
               BLACK_KEY_WIDTH = 15,   // default black key width: 15
pixels
               WHITE_KEY_WIDTH = 90)   // default white key width: 90
pixels
   (input [10:0] x, hcount,
    input [9:0] y, vcount,
    input [23:0] color,
    output reg [23:0] pixel
    );

    always @ * begin
      if ((hcount < (x+BLACK_KEY_WIDTH)) && (vcount < (y
+BLACK_KEY_HEIGHT))) pixel = 0;
      else if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y &&
vcount < (y+HEIGHT))) pixel = color;
      else pixel = 0;
   end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:11:06 12/08/2017
// Design Name:
// Module Name:    circle
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module circle
   #(parameter RADIUS = 35)            // default radius: 35 pixels
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    input [23:0] color,
    output reg [23:0] pixel);

   reg [10:0] x_length;
   reg [9:0] y_length;

   always @ * begin
      x_length <= hcount - x;
      y_length <= vcount - y;

      if ((((x_length - RADIUS) * (x_length - RADIUS)) + ((y_length -
RADIUS) * (y_length - RADIUS))) <= (RADIUS * RADIUS)) pixel <= color;
      else pixel <= 0;

   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    16:40:34 09/22/2016
// Design Name:
// Module Name:    debounce
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module debounce #(parameter DELAY=270000-1) (
    input reset,
    input clock,
    input bouncey,
    output steady
    );

    reg [18:0] count;
    reg old;
    reg steady_reg;

    always @(posedge clock) begin
        if(reset) begin
            count <= 0;
            old <= bouncey;
            steady_reg <= bouncey;
        end
        else if(bouncey != old) begin
            old <= bouncey;
            count <= 0;
        end
        else if(count == DELAY) begin
            steady_reg <= old;
        end
        else count <= count + 1;
    end
    assign steady = steady_reg;
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    23:57:34 10/13/2016
// Design Name:
// Module Name:    divider
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module divider(
    input clk,
     input reset,
    output clk_1000Hz
    ); //1000Hz enable --> on for a pulse every millisecond
     parameter CLOCK_PERIOD = 27000; //27Mhz clock

    reg [16:0] counter; //32-bit counter to reduce clock period
    reg clk_1000Hz_reg;

    always @(posedge clk) begin
        if(counter < CLOCK_PERIOD - 1) begin
            counter <= counter + 1;
            clk_1000Hz_reg <= 0;
        end
        else if(counter >= CLOCK_PERIOD - 1) begin
            clk_1000Hz_reg <= 1;
            counter <= 0;
        end
        else if(reset) begin
            clk_1000Hz_reg <= 0;
            counter <= 0;
        end
        else begin
            clk_1000Hz_reg <= 0;
            counter <= 0;
        end
    end

    assign clk_1000Hz = clk_1000Hz_reg;

endmodule
```

```
/*****************************************************************************
*     This file is owned and controlled by Xilinx and must be used           *
*     solely for design, simulation, implementation and creation of          *
*     design files limited to Xilinx devices or technologies. Use            *
*     with non-Xilinx devices or technologies is expressly prohibited        *
*     and immediately terminates your license.                               *
*                                                                            *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"          *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION        *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION            *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS              *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE       *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY               *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR         *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF        *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS        *
*     FOR A PARTICULAR PURPOSE.                                              *
*                                                                            *
*     Xilinx products are not intended for use in life support               *
*     appliances, devices, or systems. Use in such applications are          *
*     expressly prohibited.                                                  *
*                                                                            *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                   *
*     All rights reserved.                                                   *
*****************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file FIFO.v when simulating
// the core, FIFO. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module FIFO(
    clk,
    din,
    rd_en,
    rst,
    wr_en,
    data_count,
    dout,
    empty,
    full,
    overflow,
    valid,
    underflow,
    wr_ack);


input clk;
```

```
input [7 : 0] din;
input rd_en;
input rst;
input wr_en;
output [3 : 0] data_count;
output [7 : 0] dout;
output empty;
output full;
output overflow;
output valid;
output underflow;
output wr_ack;

// synthesis translate_off

      FIFO_GENERATOR_V4_4 #(
        .C_COMMON_CLOCK(1),
        .C_COUNT_TYPE(0),
        .C_DATA_COUNT_WIDTH(4),
        .C_DEFAULT_VALUE("BlankString"),
        .C_DIN_WIDTH(8),
        .C_DOUT_RST_VAL("0"),
        .C_DOUT_WIDTH(8),
        .C_ENABLE_RLOCS(0),
        .C_FAMILY("virtex2"),
        .C_FULL_FLAGS_RST_VAL(0),
        .C_HAS_ALMOST_EMPTY(0),
        .C_HAS_ALMOST_FULL(0),
        .C_HAS_BACKUP(0),
        .C_HAS_DATA_COUNT(1),
        .C_HAS_INT_CLK(0),
        .C_HAS_MEMINIT_FILE(0),
        .C_HAS_OVERFLOW(1),
        .C_HAS_RD_DATA_COUNT(0),
        .C_HAS_RD_RST(0),
        .C_HAS_RST(1),
        .C_HAS_SRST(0),
        .C_HAS_UNDERFLOW(1),
        .C_HAS_VALID(1),
        .C_HAS_WR_ACK(1),
        .C_HAS_WR_DATA_COUNT(0),
        .C_HAS_WR_RST(0),
        .C_IMPLEMENTATION_TYPE(1),
        .C_INIT_WR_PNTR_VAL(0),
        .C_MEMORY_TYPE(3),
        .C_MIF_FILE_NAME("BlankString"),
        .C_MSGON_VAL(1),
        .C_OPTIMIZATION_MODE(0),
        .C_OVERFLOW_LOW(0),
        .C_PRELOAD_LATENCY(1),
        .C_PRELOAD_REGS(0),
        .C_PRIM_FIFO_TYPE("512x36"),
        .C_PROG_EMPTY_THRESH_ASSERT_VAL(2),
        .C_PROG_EMPTY_THRESH_NEGATE_VAL(3),
        .C_PROG_EMPTY_TYPE(0),
        .C_PROG_FULL_THRESH_ASSERT_VAL(14),
```

```
        .C_PROG_FULL_THRESH_NEGATE_VAL(13),
        .C_PROG_FULL_TYPE(0),
        .C_RD_DATA_COUNT_WIDTH(4),
        .C_RD_DEPTH(16),
        .C_RD_FREQ(1),
        .C_RD_PNTR_WIDTH(4),
        .C_UNDERFLOW_LOW(0),
        .C_USE_DOUT_RST(1),
        .C_USE_ECC(0),
        .C_USE_EMBEDDED_REG(0),
        .C_USE_FIFO16_FLAGS(0),
        .C_USE_FWFT_DATA_COUNT(0),
        .C_VALID_LOW(0),
        .C_WR_ACK_LOW(0),
        .C_WR_DATA_COUNT_WIDTH(4),
        .C_WR_DEPTH(16),
        .C_WR_FREQ(1),
        .C_WR_PNTR_WIDTH(4),
        .C_WR_RESPONSE_LATENCY(1))
    inst (
        .CLK(clk),
        .DIN(din),
        .RD_EN(rd_en),
        .RST(rst),
        .WR_EN(wr_en),
        .DATA_COUNT(data_count),
        .DOUT(dout),
        .EMPTY(empty),
        .FULL(full),
        .OVERFLOW(overflow),
        .VALID(valid),
        .UNDERFLOW(underflow),
        .WR_ACK(wr_ack),
        .INT_CLK(),
        .BACKUP(),
        .BACKUP_MARKER(),
        .PROG_EMPTY_THRESH(),
        .PROG_EMPTY_THRESH_ASSERT(),
        .PROG_EMPTY_THRESH_NEGATE(),
        .PROG_FULL_THRESH(),
        .PROG_FULL_THRESH_ASSERT(),
        .PROG_FULL_THRESH_NEGATE(),
        .RD_CLK(),
        .RD_RST(),
        .SRST(),
        .WR_CLK(),
        .WR_RST(),
        .ALMOST_EMPTY(),
        .ALMOST_FULL(),
        .PROG_EMPTY(),
        .PROG_FULL(),
        .RD_DATA_COUNT(),
        .WR_DATA_COUNT(),
        .SBITERR(),
        .DBITERR());
```

```
// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of FIFO is "black_box"

endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    15:52:51 11/29/2017
// Design Name:
// Module Name:    get_measurement_timing_budget
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module getMeasurementTimingBudget (
    //FSM start and done
     input clk,
     input reset,
     input start,
    output done,
    output [31:0] timing_budget,

     //start and done for read, write, and write_multi FSMs
    output write_start,
    input write_done,
    output write_multi_start,
    input write_multi_done,
    output read_start,
    input read_done,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //data input and output
     output [7:0] reg_address_out,
    output [7:0] data_out,
    output [3:0] n_bytes,

     //read/write/write_multi select
     output [1:0] fnc_sel,

     //FIFO inputs and outputs
     //read fifo
     input [7:0] fifo_data_in,
```

```
 output fifo_read_en,
 input fifo_empty,
 input fifo_read_valid,
 input fifo_underflow,

 //write fifo
 output [7:0] fifo_data_out,
 output fifo_wr_en,
 output fifo_ext_reset,
 input fifo_full,
 input fifo_write_ack,
 input fifo_overflow,

 //inputs and outputs to read RAM
 //ram instantiated in labkit so that gobal variables can be stored and
     updated
 output [7:0] mem_addr,
 output [7:0] mem_data_out,
 input [7:0] mem_data_in,
 output mem_start,
 input mem_done,
 output mem_rw,

 //status
 output timing_budget_error,

 //debug
 output [5:0] instruction_count_debug,
 output [31:0] timeout_period_us
);

////////////////////////////////////////////////////////////////////////
    ////
// Register map taken from Pololu VL53L0X open-source library on
    github        //
//
                                    //
////////////////////////////////////////////////////////////////////////
    ////
parameter SYSRANGE_START                              = 8'h00;

parameter SYSTEM_THRESH_HIGH                          = 8'h0C;
parameter SYSTEM_THRESH_LOW                           = 8'h0E;

parameter SYSTEM_SEQUENCE_CONFIG                      = 8'h01;
parameter SYSTEM_RANGE_CONFIG                         = 8'h09;
parameter SYSTEM_INTERMEASUREMENT_PERIOD              = 8'h04;

parameter SYSTEM_INTERRUPT_CONFIG_GPIO                = 8'h0A;

parameter GPIO_HV_MUX_ACTIVE_HIGH                     = 8'h84;

parameter SYSTEM_INTERRUPT_CLEAR                      = 8'h0B;

parameter RESULT_INTERRUPT_STATUS                     = 8'h13;
parameter RESULT_RANGE_STATUS                         = 8'h14;
```

```
parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_RTN       = 8'hBC;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_RTN        = 8'hC0;
parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_REF       = 8'hD0;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_REF        = 8'hD4;
parameter RESULT_PEAK_SIGNAL_RATE_REF                 = 8'hB6;

parameter ALGO_PART_TO_PART_RANGE_OFFSET_MM           = 8'h28;

parameter I2C_SLAVE_DEVICE_ADDRESS                    = 8'h8A;

parameter MSRC_CONFIG_CONTROL                         = 8'h60;

parameter PRE_RANGE_CONFIG_MIN_SNR                    = 8'h27;
parameter PRE_RANGE_CONFIG_VALID_PHASE_LOW            = 8'h56;
parameter PRE_RANGE_CONFIG_VALID_PHASE_HIGH           = 8'h57;
parameter PRE_RANGE_MIN_COUNT_RATE_RTN_LIMIT          = 8'h64;

parameter FINAL_RANGE_CONFIG_MIN_SNR                  = 8'h67;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_LOW          = 8'h47;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_HIGH         = 8'h48;
parameter FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT = 8'h44;

parameter PRE_RANGE_CONFIG_SIGMA_THRESH_HI            = 8'h61;
parameter PRE_RANGE_CONFIG_SIGMA_THRESH_LO            = 8'h62;
parameter PRE_RANGE_CONFIG_VCSEL_PERIOD               = 8'h50;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI          = 8'h51;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_LO          = 8'h52;
parameter SYSTEM_HISTOGRAM_BIN                        = 8'h81;
parameter HISTOGRAM_CONFIG_INITIAL_PHASE_SELECT       = 8'h33;
parameter HISTOGRAM_CONFIG_READOUT_CTRL               = 8'h55;
parameter FINAL_RANGE_CONFIG_VCSEL_PERIOD             = 8'h70;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI        = 8'h71;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_LO        = 8'h72;
parameter CROSSTALK_COMPENSATION_PEAK_RATE_MCPS       = 8'h20;
parameter MSRC_CONFIG_TIMEOUT_MACROP                  = 8'h46;

parameter SOFT_RESET_GO2_SOFT_RESET_N                 = 8'hBF;
parameter IDENTIFICATION_MODEL_ID                     = 8'hC0;
parameter IDENTIFICATION_REVISION_ID                  = 8'hC2;
parameter OSC_CALIBRATE_VAL                           = 8'hF8;
parameter GLOBAL_CONFIG_VCSEL_WIDTH                   = 8'h32;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_0            = 8'hB0;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_1            = 8'hB1;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_2            = 8'hB2;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_3            = 8'hB3;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_4            = 8'hB4;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_5            = 8'hB5;
parameter GLOBAL_CONFIG_REF_EN_START_SELECT           = 8'hB6;
parameter DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD         = 8'h4E;
parameter DYNAMIC_SPAD_REF_EN_START_OFFSET            = 8'h4F;
parameter POWER_MANAGEMENT_GO1_POWER_FORCE            = 8'h80;
parameter VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV           = 8'h89;
parameter ALGO_PHASECAL_LIM                           = 8'h30;
parameter ALGO_PHASECAL_CONFIG_TIMEOUT                = 8'h30;
```

```
////////////////////////////////////////////////////////////////////////////
    ////
////////////////////////////////////////////////////////////////////////////
    ////

////////////////////////////////////////////////////////////////////////////
    ////
// Memory map for variables: ram is global
// Multi-byte memory locations are stacked LSB, then MSB in order of
    increasing
// addresses
////////////////////////////////////////////////////////////////////////////
    ////
parameter STOP_VARIABLE = 8'h00;//8-bit
parameter SPAD_COUNT = 8'h01;//8-bit
parameter SPAD_TYPE_IS_APERTURE = 8'h02;//bool
parameter REF_SPAD_MAP = 8'h03; //array
parameter SEQUENCE_STEP_ENABLE_TCC = 8'h09;//bool
parameter SEQUENCE_STEP_ENABLE_MSRC = 8'h0A;//bool
parameter SEQUENCE_STEP_ENABLE_DSS = 8'h0B;//bool
parameter SEQUENCE_STEP_ENABLE_PRE_RANGE = 8'h0C;//bool
parameter SEQUENCE_STEP_ENABLE_FINAL_RANGE = 8'h0D;//bool
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP = 8'h0E;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP = 8'h10;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTM = 8'h12; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS = 8'h14; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS = 8'h16; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTU = 8'h18; //32-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US = 8'h22; //32-bIt
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US = 8'h26; //32 bit
parameter MEASUREMENT_TIMING_BUDGET = 8'h30; //32-bit

//define data_types
parameter VcselPeriodPreRange = 8'h00;
parameter VcselPeriodFinalRange = 8'h01;


////////////////////////////////////////////////////////////////////////////
    ////
// Program Constants
////////////////////////////////////////////////////////////////////////////
    ////
parameter StartOverhead      = 32'd1910;
parameter EndOverhead        = 32'd960;
parameter MsrcOverhead       = 32'd660;
parameter TccOverhead        = 32'd590;
parameter DssOverhead        = 32'd690;
parameter PreRangeOverhead   = 32'd660;
parameter FinalRangeOverhead = 32'd550;

////////////////////////////////////////////////////////////////////////////
    ////
// Some temporary registers
////////////////////////////////////////////////////////////////////////////
    ////
reg [7:0] tmp0_8, tmp1_8, tmp2_8, tmp3_8;
```

```
 reg [15:0] tmp0_16, tmp1_16, tmp2_16, tmp3_16;
 reg [31:0] tmp0_32, tmp1_32, tmp2_32, tmp3_32;

 //define state parameters
 parameter S_RESET = 2'b00;
 parameter S_GET_ENABLES = 2'b01;
 parameter S_GET_TIMEOUTS = 2'b10;
 parameter S_DONE = 2'b11;

 //define state register and counters
 reg [1:0] state = 2'b00;
 reg [5:0] instruction_count = 6'b000000;

 //define output registers
 reg done_reg = 1'b0;
 reg [31:0] timing_budget_reg = 32'b0;

 reg write_start_reg = 1'b0;
reg write_multi_start_reg = 1'b0;
reg read_start_reg = 1'b0;

 reg timer_start_reg = 1'b0;
 reg [3:0] timer_param_reg = 4'b0101;
 reg timer_reset_reg = 1'b1;

 reg [7:0] data_out_reg = 1'b0;
 reg [3:0] n_bytes_reg = 4'b0000;
 reg [7:0] reg_address_out_reg = 8'h00;

 reg [1:0] fnc_sel_reg = 2'b00;

 reg fifo_read_en_reg = 1'b0;
 reg [7:0] fifo_data_out_reg = 8'h00;
 reg fifo_wr_en_reg = 1'b0;
 reg fifo_ext_reset_reg = 1'b1;

 reg [7:0] mem_addr_reg = 8'h00;
 reg [7:0] mem_data_out_reg = 8'h00;
 reg mem_start_reg = 1'b0;
 reg mem_rw_reg = 1'b0;

 reg timing_budget_error_reg = 1'b0;

 ////////////////////////////////////////////////////////////////////////////
 // Sub-FSM getVcselPulsePeriod
 ////////////////////////////////////////////////////////////////////////////
 reg pulse_start = 1'b0;
 wire pulse_done;

 wire write_start_pulse;
 wire write_multi_start_pulse;
 wire read_start_pulse;

 reg [7:0] vcsel_period_type = 8'h00;
 wire [7:0] vcsel_pulse_period;
```

```
        wire timer_start_pulse;
        wire [3:0] timer_param_pulse;
        wire timer_reset_pulse;

        wire [7:0] reg_address_out_pulse;
        wire [7:0] data_out_pulse;
        wire [3:0] n_bytes_pulse;

        wire [1:0] fnc_sel_pulse;

        wire [7:0] fifo_data_out_pulse;
        wire fifo_wr_en_pulse;
        wire fifo_ext_reset_pulse;
        wire fifo_read_en_pulse;

        wire [7:0] mem_addr_pulse;
        wire [7:0] mem_data_out_pulse;
        wire mem_start_pulse;
        wire mem_rw_pulse;

        wire pulse_error;

        getVcselPulsePeriod getVcselPulsePeriod(
            .reset(reset),
            .clk(clk),
            .start(pulse_start),
            .done(pulse_done),

            .vcsel_period_type(vcsel_period_type),
            .vcsel_pulse_period(vcsel_pulse_period),

            .write_start(write_start_pulse),
            .write_done(write_done),
            .write_multi_start(write_multi_start_pulse),
            .write_multi_done(write_multi_done),
            .read_start(read_start_pulse),
            .read_done(read_done),

            .timer_exp(timer_exp),
            .timer_start(timer_start_pulse),
            .timer_param(timer_param_pulse),
            .timer_reset(timer_reset_pulse),

            .reg_address_out(reg_address_out_pulse),
            .data_out(data_out_pulse),
            .n_bytes(n_bytes_pulse),

            .fnc_sel(fnc_sel_pulse),

            .fifo_data_out(fifo_data_out_pulse),
            .fifo_wr_en(fifo_wr_en_pulse),
           .fifo_ext_reset(fifo_ext_reset_pulse),
           .fifo_full(fifo_full),
           .fifo_write_ack(fifo_write_ack),
           .fifo_overflow(fifo_overflow),
```

```
    .fifo_data_in(fifo_data_in),
   .fifo_read_en(fifo_read_en_pulse),
   .fifo_empty(fifo_empty),
   .fifo_read_valid(fifo_read_valid),
   .fifo_underflow(fifo_underflow),

    .mem_addr(mem_addr_pulse),
   .mem_data_out(mem_data_out_pulse),
   .mem_data_in(mem_data_in),
   .mem_start(mem_start_pulse),
    .mem_done(mem_done),
    .mem_rw(mem_rw_pulse),

    .error(pulse_error)
);

////////////////////////////////////////////////////////////////////////////
// Sub-FSM timeoutMclksToMicroseconds
////////////////////////////////////////////////////////////////////////////
reg timeout_convert_start = 1'b0;
wire timeout_convert_done;
reg [15:0] timeout_period_mclks= 16'h0000;
reg [7:0] vcsel_period_pclks = 8'h00;
//wire [31:0] timeout_period_us;

timeoutMclksToMicroseconds timeoutMclksToMicroseconds(
    .clk(clk),
    .reset(reset),
    .start(timeout_convert_start),
    .done(timeout_convert_done),

    .timeout_period_mclks(timeout_period_mclks),
    .vcsel_period_pclks(vcsel_period_pclks),
    .timeout_period_us(timeout_period_us)
);

//main FSM implementation
always @(posedge clk) begin
    if(reset) state <= S_RESET;
    else begin
        case(state)
            S_RESET: begin
                if(start) state <= S_GET_ENABLES;
                else state <= S_RESET;

                done_reg <= 1'b0;

                write_start_reg <= 1'b0;
                write_multi_start_reg <= 1'b0;
                read_start_reg <= 1'b0;

                timer_start_reg <= 1'b0;
                timer_param_reg <= 4'b0101;
                timer_reset_reg <= 1'b1;

                data_out_reg <= 8'h00;
```

```
            n_bytes_reg <= 4'b0000;
            reg_address_out_reg <= 8'h00;

            fnc_sel_reg <= 2'b00;

            fifo_read_en_reg <= 1'b0;
            fifo_data_out_reg <= 8'h00;
            fifo_wr_en_reg <= 1'b0;
            fifo_ext_reset_reg <= 1'b1;

            mem_addr_reg <= 8'h00;
            mem_data_out_reg <= 8'h00;
            mem_rw_reg <= 1'b0;
            mem_start_reg <= 1'b0;

            timing_budget_error_reg <= 1'b0;

            tmp0_8 <= 8'h00;
            tmp1_8 <= 8'h00;
            tmp2_8 <= 8'h00;
            tmp3_8 <= 8'h00;

            tmp0_16 <= 16'h0000;
            tmp1_16 <= 16'h0000;
            tmp2_16 <= 16'h0000;
            tmp3_16 <= 16'h0000;

            tmp0_32 <= 32'h00000000;
            tmp1_32 <= 32'h00000000;
            tmp2_32 <= 32'h00000000;
            tmp3_32 <= 32'h00000000;

            pulse_start <= 1'b0;
            vcsel_period_type <= 8'h00;

            timeout_convert_start <= 1'b0;
            timeout_period_mclks <= 16'h0000;
            vcsel_period_pclks <= 8'h00;

            instruction_count <= 6'b000000;
        end
        S_GET_ENABLES: begin
            //getSequenceStepEnables(&enables);
            case(instruction_count)
                //uint8_t sequence_config =
                    readReg(SYSTEM_SEQUENCE_CONFIG);
                6'b000000: begin
                    tmp0_32 <= StartOverhead + EndOverhead;
                    instruction_count <= instruction_count + 1;
                    state <= S_GET_ENABLES;
                end
                6'b000001: begin
                    read_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b01; //sel read
                    n_bytes_reg <= 4'b0001;
                    reg_address_out_reg <= SYSTEM_SEQUENCE_CONFIG;
```

```
            instruction_count <= instruction_count + 1;
            state <= S_GET_ENABLES;
        end
        6'b000010: begin
            if(!read_done) begin
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_ENABLES;
        end
        6'b000011: begin
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_ENABLE_TCC;
                mem_data_out_reg <= (fifo_data_in >> 4) & 8'h01;
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_ENABLES;
        end
        6'b000100: begin
            if(!mem_done) begin
                instruction_count <= instruction_count;
                mem_start_reg <= 1'b0;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_ENABLE_DSS;
                mem_data_out_reg <= (fifo_data_in >> 3) & 8'h01;
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_ENABLES;
        end
        6'b000101: begin
            if(!mem_done) begin
                instruction_count <= instruction_count;
                mem_start_reg <= 1'b0;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_ENABLE_MSRC;
                mem_data_out_reg <= (fifo_data_in >> 2) & 8'h01;
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;
```

```
                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_GET_ENABLES;
                end
                6'b000110: begin
                    if(!mem_done) begin
                        instruction_count <= instruction_count;
                        mem_start_reg <= 1'b0;
                    end
                    else begin
                        mem_addr_reg <= SEQUENCE_STEP_ENABLE_PRE_RANGE;
                        mem_data_out_reg <= (fifo_data_in >> 6) & 8'h01;
                        mem_rw_reg <= 1'b1;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_GET_ENABLES;
                end
                6'b000111: begin
                    if(!mem_done) begin
                        instruction_count <= instruction_count;
                        mem_start_reg <= 1'b0;
                    end
                    else begin
                        mem_addr_reg <= SEQUENCE_STEP_ENABLE_FINAL_RANGE;
                        mem_data_out_reg <= (fifo_data_in >> 7) & 8'h01;
                        mem_rw_reg <= 1'b1;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_GET_ENABLES;
                end
                6'b001000: begin
                    if(!mem_done) begin
                        instruction_count <= instruction_count;
                        mem_start_reg <= 1'b0;
                        state <= S_GET_ENABLES;
                    end
                    else begin
                        instruction_count <= 6'b000000;
                        state <= S_GET_TIMEOUTS;
                    end
                end
            endcase
        end
        S_GET_TIMEOUTS: begin
            ////////////////////////////////////////////////////////////
                ////////////////////////////////
            // getSequenceStepTimeouts
            ////////////////////////////////////////////////////////////
                ////////////////////////////////
            case(instruction_count)
                ////////////////////////////////////////////////////////
                    ////////////////////////////////////
```

```
//timeouts->pre_range_vcsel_period_pclks =
    getVcselPulsePeriod(VcselPeriodPreRange);
////////////////////////////////////////////////////
    ////////////////////////////////////
6'b000000: begin
    //shift FSM Control to getVcselPulsePeriod
    pulse_start <= 1'b1;
    vcsel_period_type <= VcselPeriodPreRange;

    write_start_reg <= write_start_pulse;
    write_multi_start_reg <= write_multi_start_pulse;
    read_start_reg <= read_start_pulse;

    timer_start_reg <= timer_start_pulse;
    timer_param_reg <= timer_param_pulse;
    timer_reset_reg <= timer_reset_pulse;

    reg_address_out_reg <= reg_address_out_pulse;
    data_out_reg <= data_out_pulse;
    n_bytes_reg <= n_bytes_pulse;

    fnc_sel_reg <= fnc_sel_pulse;

    fifo_data_out_reg <= fifo_data_out_pulse;
    fifo_wr_en_reg <= fifo_wr_en_pulse;
    fifo_ext_reset_reg <= fifo_ext_reset_pulse;
    fifo_read_en_reg <= fifo_read_en_pulse;

    mem_addr_reg <= mem_addr_pulse;
    mem_data_out_reg <= mem_data_out_pulse;
    mem_start_reg <= mem_start_pulse;
    mem_rw_reg <= mem_rw_pulse;

    timing_budget_error_reg <= pulse_error;

    instruction_count <= instruction_count + 1;
    state <= S_GET_TIMEOUTS;
end
6'b000001: begin
    //only relinquish control back to normals FSM until
        VcselPulsePeriod is finished
    if(!pulse_done) begin
        pulse_start <= 1'b0;
        vcsel_period_type <= VcselPeriodPreRange;

        write_start_reg <= write_start_pulse;
        write_multi_start_reg <= write_multi_start_pulse;
        read_start_reg <= read_start_pulse;

        timer_start_reg <= timer_start_pulse;
        timer_param_reg <= timer_param_pulse;
        timer_reset_reg <= timer_reset_pulse;

        reg_address_out_reg <= reg_address_out_pulse;
        data_out_reg <= data_out_pulse;
        n_bytes_reg <= n_bytes_pulse;
```

```
                fnc_sel_reg <= fnc_sel_pulse;

                fifo_data_out_reg <= fifo_data_out_pulse;
                fifo_wr_en_reg <= fifo_wr_en_pulse;
                fifo_ext_reset_reg <= fifo_ext_reset_pulse;
                fifo_read_en_reg <= fifo_read_en_pulse;

                mem_addr_reg <= mem_addr_pulse;
                mem_data_out_reg <= mem_data_out_pulse;
                mem_start_reg <= mem_start_pulse;
                mem_rw_reg <= mem_rw_pulse;

                timing_budget_error_reg <= pulse_error;

                instruction_count <= instruction_count;
            end
            else begin
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b000010: begin
            mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP;
            mem_data_out_reg <= vcsel_pulse_period;
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b000011: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP + 1;
                mem_data_out_reg <= 8'h00;
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        ////////////////////////////////////////////////////////
            ////////////////////////////////////
        //timeouts->msrc_dss_tcc_mclks =
            readReg(MSRC_CONFIG_TIMEOUT_MACROP) + 1;
        ////////////////////////////////////////////////////////
            ////////////////////////////////////
        6'b000100: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
```

```
                instruction_count <= instruction_count;
            end
            else begin
                read_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b01; //sel read
                n_bytes_reg <= 4'b0001;
                reg_address_out_reg <=
                    MSRC_CONFIG_TIMEOUT_MACROP;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b000101: begin
            if(!read_done) begin
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b000110: begin
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                tmp0_16 <= fifo_data_in + 1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b000111: begin
            mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTM;
            mem_data_out_reg <= tmp0_16[7:0];
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b001000: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTM +
                    1;
                mem_data_out_reg <= tmp0_16[15:8];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;
```

```
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        ////////////////////////////////////////////////////////
            ////////////////////////////////////////////////////
            ////////////////////
        //timeouts->msrc_dss_tcc_us =
            timeoutMclksToMicroseconds(timeouts-
            >msrc_dss_tcc_mclks, timeouts-
            >pre_range_vcsel_period_pclks);
        ////////////////////////////////////////////////////////
            ////////////////////////////////////////////////////
            ////////////////////
        6'b001001: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                timeout_convert_start <= 1'b1;
                timeout_period_mclks <= tmp0_16; //mrsc_dtm
                vcsel_period_pclks <= vcsel_pulse_period;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b001010: begin
            if(!timeout_convert_done) begin
                instruction_count <= instruction_count;
                timeout_convert_start <= 1'b0;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU;
                mem_data_out_reg <= timeout_period_us[7:0];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b001011: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                    1;
                mem_data_out_reg <= timeout_period_us[15:8];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
```

```
                end
                state <= S_GET_TIMEOUTS;
            end
            6'b001100: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                        2;
                    mem_data_out_reg <= timeout_period_us[23:16];
                    mem_rw_reg <= 1'b1;
                    mem_start_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_GET_TIMEOUTS;
            end
            6'b001101: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                        3;
                    mem_data_out_reg <= timeout_period_us[31:24];
                    mem_rw_reg <= 1'b1;
                    mem_start_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_GET_TIMEOUTS;
            end
            ////////////////////////////////////////////////////////
                ////////////////////////////////////
            //timeouts->pre_range_mclks =
                decodeTimeout(readReg16Bit(PRE_RANGE_CONFIG_TIMEOUT_M
                ACROP_HI));
            ////////////////////////////////////////////////////////
                ////////////////////////////////////
            6'b001110: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    read_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b01; //sel read
                    n_bytes_reg <= 4'b0010;
                    reg_address_out_reg <=
                        PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI;

                    instruction_count <= instruction_count + 1;
                end
```

```
            state <= S_GET_TIMEOUTS;
        end
        6'b001111: begin
            //Read MSB of byte
            if(!read_done) begin
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b010000: begin
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                tmp0_8 <= fifo_data_in;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b010001: begin
            //Read LSB of byte
            fifo_read_en_reg <= 1'b1;
            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b010010: begin
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                tmp1_8 <= fifo_data_in;
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b010011: begin
            tmp0_16 <= tmp1_8 * (1 << tmp0_8) + 1; //LS_BYTE *
                2^(MS_BYTE) + 1 - I2C sends msb then lsb
            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b010100: begin
            mem_addr_reg <=
                SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS;
            mem_data_out_reg <= tmp0_16[7:0]; //lsb
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
```

```
                state <= S_GET_TIMEOUTS;
            end
            6'b010101: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    mem_addr_reg <=
                        SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS + 1;
                    mem_data_out_reg <= tmp0_16[15:8]; //msb
                    mem_rw_reg <= 1'b1;
                    mem_start_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_GET_TIMEOUTS;
            end
            ////////////////////////////////////////////////////////
                ////////////////////////////////////////////////////
                ////////////
            //timeouts->pre_range_us =
                timeoutMclksToMicroseconds(timeouts->pre_range_mclks,
                timeouts->pre_range_vcsel_period_pclks);
            ////////////////////////////////////////////////////////
                ////////////////////////////////////////////////////
                ////////////
            6'b010110: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    timeout_convert_start <= 1'b1;
                    timeout_period_mclks <= {tmp0_8, tmp1_8}; //
                        pre_range_mclks
                    vcsel_period_pclks <= vcsel_pulse_period; //same
                        period from before

                    instruction_count <= instruction_count + 1;
                end
                state <= S_GET_TIMEOUTS;
            end
            6'b010111: begin
                if(!timeout_convert_done) begin
                    timeout_convert_start <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    mem_addr_reg <=
                        SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US;
                    mem_data_out_reg <= timeout_period_us[7:0];
                    mem_rw_reg <= 1'b1;
                    mem_start_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
```

```
                state <= S_GET_TIMEOUTS;
            end
        end
        6'b011000: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 1;
                mem_data_out_reg <= timeout_period_us[15:8];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b011001: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 2;
                mem_data_out_reg <= timeout_period_us[23:16];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b011010: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 3;
                mem_data_out_reg <= timeout_period_us[31:24];
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        ////////////////////////////////////////////////////////
            ///////////////////////////////////
        // timeouts->final_range_vcsel_period_pclks =
            getVcselPulsePeriod(VcselPeriodFinalRange);
        ////////////////////////////////////////////////////////
```

```
                    ////////////////////////////////
                6'b011011: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        //shift FSM Control to getVcselPulsePeriod
                        pulse_start <= 1'b1;
                        vcsel_period_type <= VcselPeriodFinalRange;

                        write_start_reg <= write_start_pulse;
                        write_multi_start_reg <= write_multi_start_pulse;
                        read_start_reg <= read_start_pulse;

                        timer_start_reg <= timer_start_pulse;
                        timer_param_reg <= timer_param_pulse;
                        timer_reset_reg <= timer_reset_pulse;

                        reg_address_out_reg <= reg_address_out_pulse;
                        data_out_reg <= data_out_pulse;
                        n_bytes_reg <= n_bytes_pulse;

                        fnc_sel_reg <= fnc_sel_pulse;

                        fifo_data_out_reg <= fifo_data_out_pulse;
                        fifo_wr_en_reg <= fifo_wr_en_pulse;
                        fifo_ext_reset_reg <= fifo_ext_reset_pulse;
                        fifo_read_en_reg <= fifo_read_en_pulse;

                        mem_addr_reg <= mem_addr_pulse;
                        mem_data_out_reg <= mem_data_out_pulse;
                        mem_start_reg <= mem_start_pulse;
                        mem_rw_reg <= mem_rw_pulse;

                        timing_budget_error_reg <= pulse_error;

                        instruction_count <= instruction_count + 1;
                        state <= S_GET_TIMEOUTS;
                    end
                end
                6'b011100: begin
                    //only relinquish control back to normals FSM until
                        VcselPulsePeriod is finished
                    if(!pulse_done) begin
                        pulse_start <= 1'b0;
                        vcsel_period_type <= VcselPeriodFinalRange;

                        write_start_reg <= write_start_pulse;
                        write_multi_start_reg <= write_multi_start_pulse;
                        read_start_reg <= read_start_pulse;

                        timer_start_reg <= timer_start_pulse;
                        timer_param_reg <= timer_param_pulse;
                        timer_reset_reg <= timer_reset_pulse;
```

```
                reg_address_out_reg <= reg_address_out_pulse;
                data_out_reg <= data_out_pulse;
                n_bytes_reg <= n_bytes_pulse;

                fnc_sel_reg <= fnc_sel_pulse;

                fifo_data_out_reg <= fifo_data_out_pulse;
                fifo_wr_en_reg <= fifo_wr_en_pulse;
                fifo_ext_reset_reg <= fifo_ext_reset_pulse;
                fifo_read_en_reg <= fifo_read_en_pulse;

                mem_addr_reg <= mem_addr_pulse;
                mem_data_out_reg <= mem_data_out_pulse;
                mem_start_reg <= mem_start_pulse;
                mem_rw_reg <= mem_rw_pulse;

                timing_budget_error_reg <= pulse_error;

                instruction_count <= instruction_count;
            end
            else begin
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b011101: begin
            mem_addr_reg <=
                SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP;
            mem_data_out_reg <= vcsel_pulse_period;
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b011110: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP + 1;
                mem_data_out_reg <= 8'h00;
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        ////////////////////////////////////////////////////////
            ////////////////////////////////////////
        //timeouts->final_range_mclks =
            decodeTimeout(readReg16Bit(FINAL_RANGE_CONFIG_TIMEOUT
            _MACROP_HI));
```

```
                    ////////////////////////////////////////////////////
                        ////////////////////////////////////////
                    6'b011111: begin
                        if(!mem_done) begin
                            mem_start_reg <= instruction_count;
                        end
                        else begin
                            read_start_reg <= 1'b1;
                            fnc_sel_reg <= 2'b01; //sel read
                            n_bytes_reg <= 4'b0010;
                            reg_address_out_reg <=
                                FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI;

                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_GET_TIMEOUTS;
                    end
                    6'b100000: begin
                        //Read MSB of byte
                        if(!read_done) begin
                            instruction_count <= instruction_count;
                            read_start_reg <= 1'b0;
                        end
                        else begin
                            fifo_read_en_reg <= 1'b1;
                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_GET_TIMEOUTS;
                    end
                    6'b100001: begin
                        if(!fifo_read_valid) begin
                            instruction_count <= instruction_count;
                            fifo_read_en_reg <= 1'b0;
                        end
                        else begin
                            tmp0_8 <= fifo_data_in;
                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_GET_TIMEOUTS;
                    end
                    6'b100010: begin
                        //Read LSB of byte
                        fifo_read_en_reg <= 1'b1;
                        instruction_count <= instruction_count + 1;
                        state <= S_GET_TIMEOUTS;
                    end
                    6'b100011: begin
                        if(!fifo_read_valid) begin
                            instruction_count <= instruction_count;
                            fifo_read_en_reg <= 1'b0;
                        end
                        else begin
                            tmp1_8 <= fifo_data_in;
                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_GET_TIMEOUTS;
```

```
end
6'b100100: begin
    tmp0_16 <= tmp1_8 * (1 << tmp0_8) + 1; //LS_BYTE *
        2^(MS_BYTE) + 1 - I2C sends msb then lsb
    instruction_count <= instruction_count + 1;
    state <= S_GET_TIMEOUTS;
end
6'b100101: begin
    mem_addr_reg <=
        SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS;
    mem_data_out_reg <= tmp0_16[7:0]; //lsb
    mem_rw_reg <= 1'b1;
    mem_start_reg <= 1'b1;

    instruction_count <= instruction_count + 1;
    state <= S_GET_TIMEOUTS;
end
6'b100110: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        mem_addr_reg <=
            SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS + 1;
        mem_data_out_reg <= tmp0_16[15:8]; //msb
        mem_rw_reg <= 1'b1;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
    end
    state <= S_GET_TIMEOUTS;
end
/*
 if (enables->pre_range)
 {
    timeouts->final_range_mclks -= timeouts-
        >pre_range_mclks;
 }
*/
6'b100111: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        mem_addr_reg <= SEQUENCE_STEP_ENABLE_PRE_RANGE;
        mem_rw_reg <= 1'b0;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
    end
    state <= S_GET_TIMEOUTS;
end
6'b101000: begin
    if(!mem_done) begin
```

```
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp0_8 <= mem_data_in; //enables->pre_range
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b101001: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp0_16 <= {8'h00, mem_data_in} & 16'h00FF; //set
                    LSB of FINAL_RANGE_MCLKS
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS + 1;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b101010: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp0_16 <= tmp0_16 + ({mem_data_in, 8'h00} &
                    16'hFF00); //set MSB of FINAL_RANGE_MCLKS
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b101011: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_16 <= {8'h00, mem_data_in} & 16'h00FF;//set
                    LSB of PRE_RANGE_MCLKS
```

```
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS + 1;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b101100: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_16 <= tmp1_16 + ({mem_data_in, 8'h00} &
                    16'hFF00); //set MSB of PRE_RANGE_MCLKS
                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        6'b101101: begin
            if(tmp0_8[0]) begin //if enables->pre_range
                tmp0_16 <= tmp0_16 - tmp1_16;
            end
            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b101110: begin
            mem_addr_reg <=
                SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS;
            mem_data_out_reg <= tmp0_16[7:0]; //lsb
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_GET_TIMEOUTS;
        end
        6'b101111: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS + 1;
                mem_data_out_reg <= tmp0_16[15:8]; //msb
                mem_rw_reg <= 1'b1;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_GET_TIMEOUTS;
        end
        ////////////////////////////////////////////////////
            ////////////////////////////////////
```

```
// timeouts->final_range_us =
    timeoutMclksToMicroseconds(timeouts-
    >final_range_mclks, timeouts-
    >final_range_vcsel_period_pclks);
////////////////////////////////////////////////////
    ////////////////////////////////////
6'b110000: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        timeout_convert_start <= 1'b1;
        timeout_period_mclks <= tmp0_16; //
            final_range_mclks
        vcsel_period_pclks <= vcsel_pulse_period; //same
            period from before

        instruction_count <= instruction_count + 1;
    end
    state <= S_GET_TIMEOUTS;
end
6'b110001: begin
    if(!timeout_convert_done) begin
        timeout_convert_start <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        mem_addr_reg <=
            SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US;
        mem_data_out_reg <= timeout_period_us[7:0];
        mem_rw_reg <= 1'b1;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
        state <= S_GET_TIMEOUTS;
    end
end
6'b110010: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        mem_addr_reg <=
            SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 1;
        mem_data_out_reg <= timeout_period_us[15:8];
        mem_rw_reg <= 1'b1;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
    end
    state <= S_GET_TIMEOUTS;
end
6'b110011: begin
    if(!mem_done) begin
```

```
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        mem_addr_reg <=
                            SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 2;
                        mem_data_out_reg <= timeout_period_us[23:16];
                        mem_rw_reg <= 1'b1;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_GET_TIMEOUTS;
                end
                6'b110100: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                        state <= S_GET_TIMEOUTS;
                    end
                    else begin
                        mem_addr_reg <=
                            SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 3;
                        mem_data_out_reg <= timeout_period_us[31:24];
                        mem_rw_reg <= 1'b1;
                        mem_start_reg <= 1'b1;

                        instruction_count <= 6'b000000;
                        state <= S_DONE;
                    end
                end
            endcase
        end
        S_DONE: begin
            /*
            if (enables.tcc)
            {
                budget_us += (timeouts.msrc_dss_tcc_us + TccOverhead);
            }
            */
            case(instruction_count)
                6'b000000: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        mem_addr_reg <= SEQUENCE_STEP_ENABLE_TCC;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_DONE;
                end
                6'b000001: begin
```

```
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        tmp0_8 <= mem_data_in;
                        mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_DONE;
                end
                6'b000010: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        tmp1_32 <= {8'h00, 8'h00, 8'h00, mem_data_in} &
                            32'h000000FF;
                        mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                            1;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_DONE;
                end
                6'b000011: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        tmp1_32 <= tmp1_32 + ({8'h00, 8'h00, mem_data_in,
                            8'h00} & 32'h0000FF00);
                        mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                            2;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b1;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_DONE;
                end
                6'b000100: begin
                    if(!mem_done) begin
                        mem_start_reg <= 1'b0;
                        instruction_count <= instruction_count;
                    end
                    else begin
                        tmp1_32 <= tmp1_32 + ({8'h00, mem_data_in, 8'h00,
                            8'h00} & 32'h00FF0000);
```

```
                mem_addr_reg <= SEQUENCE_STEP_TIMEOUTS_MSRC_DTU +
                    3;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b000101: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({mem_data_in, 8'h00, 8'h00,
                    8'h00} & 32'hFF000000);
                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b000110: begin
            if(tmp0_8[0]) begin
                tmp0_32 <= tmp0_32 + (tmp1_32 + TccOverhead);
            end
            instruction_count <= instruction_count + 1;
            state <= S_DONE;
        end
        /*
         if (enables.dss)
         {
            budget_us += 2 * (timeouts.msrc_dss_tcc_us +
                DssOverhead);
         }
         else if (enables.msrc)
         {
            budget_us += (timeouts.msrc_dss_tcc_us +
                MsrcOverhead);
         }
        */
        6'b000111: begin
            mem_addr_reg <= SEQUENCE_STEP_ENABLE_DSS;
            mem_rw_reg <= 1'b0;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_DONE;
        end
        6'b001000: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp0_8 <= mem_data_in;
                instruction_count <= instruction_count + 1;
```

```
    end
    state <= S_DONE;
end
6'b001001: begin
    mem_addr_reg <= SEQUENCE_STEP_ENABLE_MSRC;
    mem_rw_reg <= 1'b0;
    mem_start_reg <= 1'b1;

    instruction_count <= instruction_count + 1;
    state <= S_DONE;
end
6'b001010: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        tmp1_8 <= mem_data_in;
        instruction_count <= instruction_count + 1;
    end
    state <= S_DONE;
end
6'b001011: begin
    if(tmp0_8[0]) begin
        tmp0_32 <= tmp0_32 + 2*(tmp1_32 + DssOverhead);
    end
    else if (tmp1_8[0]) begin
        tmp0_32 <= tmp0_32 + (tmp1_32 + MsrcOverhead);
    end
    instruction_count <= instruction_count + 1;
    state <= S_DONE;
end
/*
 if (enables.pre_range)
 {
  budget_us += (timeouts.pre_range_us +
      PreRangeOverhead);
 }
*/
6'b001100: begin
    mem_addr_reg <= SEQUENCE_STEP_ENABLE_PRE_RANGE;
    mem_rw_reg <= 1'b0;
    mem_start_reg <= 1'b1;

    instruction_count <= instruction_count + 1;
    state <= S_DONE;
end
6'b001101: begin
    if(!mem_done) begin
        mem_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        tmp0_8 <= mem_data_in;
        mem_addr_reg <=
            SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US;
```

```
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b001110: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= {8'h00, 8'h00, 8'h00, mem_data_in} &
                    32'h000000FF;
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 1;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b001111: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({8'h00, 8'h00, mem_data_in,
                    8'h00} & 32'h0000FF00);
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 2;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b010000: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({8'h00, mem_data_in, 8'h00,
                    8'h00} & 32'h00FF0000);
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US + 3;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
```

```
            state <= S_DONE;
        end
        6'b010001: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({mem_data_in, 8'h00, 8'h00,
                    8'h00} & 32'hFF000000);
                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b010010: begin
            if(tmp0_8[0]) begin
                tmp0_32 <= tmp0_32 + (tmp1_32 +
                    PreRangeOverhead);
            end
            instruction_count <= instruction_count + 1;
            state <= S_DONE;
        end
        /*
         if (enables.final_range)
         {
          budget_us += (timeouts.final_range_us +
              FinalRangeOverhead);
         }
        */
        6'b010011: begin
            mem_addr_reg <= SEQUENCE_STEP_ENABLE_FINAL_RANGE;
            mem_rw_reg <= 1'b0;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
            state <= S_DONE;
        end
        6'b010100: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp0_8 <= mem_data_in;
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b010101: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
```

```
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= {8'h00, 8'h00, 8'h00, mem_data_in} &
                    32'h000000FF;
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 1;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b010110: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({8'h00, 8'h00, mem_data_in,
                    8'h00} & 32'h0000FF00);
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 2;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b010111: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({8'h00, mem_data_in, 8'h00,
                    8'h00} & 32'h00FF0000);
                mem_addr_reg <=
                    SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US + 3;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count + 1;
            end
            state <= S_DONE;
        end
        6'b011000: begin
            if(!mem_done) begin
                mem_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                tmp1_32 <= tmp1_32 + ({mem_data_in, 8'h00, 8'h00,
                    8'h00} & 32'hFF000000);
```

```
            instruction_count <= instruction_count + 1;
        end
        state <= S_DONE;
    end
    6'b011001: begin
        if(tmp0_8[0]) begin
            tmp0_32 <= tmp0_32 + (tmp1_32 +
                FinalRangeOverhead);
        end
        instruction_count <= instruction_count + 1;
        state <= S_DONE;
    end
    6'b011010: begin
        mem_addr_reg <= MEASUREMENT_TIMING_BUDGET;
        mem_data_out_reg <= tmp0_32[7:0];
        mem_rw_reg <= 1'b1;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
        state <= S_DONE;
    end
    6'b011011: begin
        if(!mem_done) begin
            instruction_count <= instruction_count;
            mem_start_reg <= 1'b0;
        end
        else begin
            mem_addr_reg <= MEASUREMENT_TIMING_BUDGET + 1;
            mem_data_out_reg <= tmp0_32[15:8];
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
        end
        state <= S_DONE;
    end
    6'b011100: begin
        if(!mem_done) begin
            instruction_count <= instruction_count;
            mem_start_reg <= 1'b0;
        end
        else begin
            mem_addr_reg <= MEASUREMENT_TIMING_BUDGET + 2;
            mem_data_out_reg <= tmp0_32[23:16];
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
        end
        state <= S_DONE;
    end
    6'b011101: begin
        if(!mem_done) begin
            instruction_count <= instruction_count;
            mem_start_reg <= 1'b0;
        end
```

```
                            else begin
                                mem_addr_reg <= MEASUREMENT_TIMING_BUDGET + 3;
                                mem_data_out_reg <= tmp0_32[31:24];
                                mem_rw_reg <= 1'b1;
                                mem_start_reg <= 1'b1;

                                instruction_count <= instruction_count + 1;
                            end
                            state <= S_DONE;
                        end
                        6'b011110: begin
                            if(!mem_done) begin
                                mem_start_reg <= 1'b0;
                                instruction_count <= instruction_count;
                                state <= S_DONE;
                            end
                            else begin
                                timing_budget_reg <= tmp0_32;
                                done_reg <= 1'b1;
                                state <= S_RESET;
                            end
                        end
                    endcase
                end
            endcase
        end
    end

    //assign registers to outputs
    assign done = done_reg;
    assign timing_budget = timing_budget_reg;

    assign timer_start = timer_start_reg;
    assign timer_param = timer_param_reg;
    assign timer_reset = timer_reset_reg;

   assign write_start = write_start_reg;
   assign write_multi_start = write_multi_start_reg;
   assign read_start = read_start_reg;

    assign data_out = data_out_reg;
    assign n_bytes = n_bytes_reg;
    assign reg_address_out = reg_address_out_reg;

    assign fnc_sel = fnc_sel_reg;

    assign fifo_read_en = fifo_read_en_reg;
    assign fifo_data_out = fifo_data_out_reg;
    assign fifo_wr_en = fifo_wr_en_reg;
    assign fifo_ext_reset = fifo_ext_reset_reg;

    assign mem_addr = mem_addr_reg;
    assign mem_data_out = mem_data_out_reg;
    assign mem_start = mem_start_reg;
    assign mem_rw = mem_rw_reg;
```

```
        assign timing_budget_error = timing_budget_error_reg;
        assign instruction_count_debug = instruction_count;

    endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    15:52:51 11/29/2017
// Design Name:
// Module Name:    get_spad_info
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module get_spad_info(
    //FSM start and done
     input clk,
     input reset,
     input start,
    output done,

     //start and done for read, write, and write_multi FSMs
    output write_start,
    input write_done,
    output write_multi_start,
    input write_multi_done,
    output read_start,
    input read_done,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //data input and output
     output [7:0] reg_address_out,
    output [7:0] data_out,
    output [3:0] n_bytes,

     //read/write/write_multi select
     output [1:0] fnc_sel,

     //FIFO inputs and outputs
     //read fifo
     input [7:0] fifo_data_in,
     output fifo_read_en,
```

```
 input fifo_empty,
 input fifo_read_valid,
 input fifo_underflow,

 //write fifo
 output [7:0] fifo_data_out,
 output fifo_wr_en,
 output fifo_ext_reset,
 input fifo_full,
 input fifo_write_ack,
 input fifo_overflow,

 //inputs and outputs to read RAM
 //ram instantiated in labkit so that gobal variables can be stored and
     updated
 output [7:0] mem_addr,
 output [7:0] mem_data_out,
 input [7:0] mem_data_in,
 output mem_start,
 input mem_done,
 output mem_rw,

 //status
 output spad_error
);

////////////////////////////////////////////////////////////////////////////
    ////
// Register map taken from Pololu VL53L0X open-source library on
    github        //
//
                                    //
////////////////////////////////////////////////////////////////////////////
    ////
parameter SYSRANGE_START                                = 8'h00;

parameter SYSTEM_THRESH_HIGH                            = 8'h0C;
parameter SYSTEM_THRESH_LOW                             = 8'h0E;

parameter SYSTEM_SEQUENCE_CONFIG                        = 8'h01;
parameter SYSTEM_RANGE_CONFIG                           = 8'h09;
parameter SYSTEM_INTERMEASUREMENT_PERIOD                = 8'h04;

parameter SYSTEM_INTERRUPT_CONFIG_GPIO                  = 8'h0A;

parameter GPIO_HV_MUX_ACTIVE_HIGH                       = 8'h84;

parameter SYSTEM_INTERRUPT_CLEAR                        = 8'h0B;

parameter RESULT_INTERRUPT_STATUS                       = 8'h13;
parameter RESULT_RANGE_STATUS                           = 8'h14;

parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_RTN         = 8'hBC;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_RTN          = 8'hC0;
parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_REF         = 8'hD0;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_REF          = 8'hD4;
```

```
parameter RESULT_PEAK_SIGNAL_RATE_REF                = 8'hB6;

parameter ALGO_PART_TO_PART_RANGE_OFFSET_MM          = 8'h28;

parameter I2C_SLAVE_DEVICE_ADDRESS                   = 8'h8A;

parameter MSRC_CONFIG_CONTROL                        = 8'h60;

parameter PRE_RANGE_CONFIG_MIN_SNR                   = 8'h27;
parameter PRE_RANGE_CONFIG_VALID_PHASE_LOW           = 8'h56;
parameter PRE_RANGE_CONFIG_VALID_PHASE_HIGH          = 8'h57;
parameter PRE_RANGE_MIN_COUNT_RATE_RTN_LIMIT         = 8'h64;

parameter FINAL_RANGE_CONFIG_MIN_SNR                 = 8'h67;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_LOW         = 8'h47;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_HIGH        = 8'h48;
parameter FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT = 8'h44;

parameter PRE_RANGE_CONFIG_SIGMA_THRESH_HI           = 8'h61;
parameter PRE_RANGE_CONFIG_SIGMA_THRESH_LO           = 8'h62;
parameter PRE_RANGE_CONFIG_VCSEL_PERIOD              = 8'h50;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI         = 8'h51;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_LO         = 8'h52;
parameter SYSTEM_HISTOGRAM_BIN                       = 8'h81;
parameter HISTOGRAM_CONFIG_INITIAL_PHASE_SELECT      = 8'h33;
parameter HISTOGRAM_CONFIG_READOUT_CTRL              = 8'h55;
parameter FINAL_RANGE_CONFIG_VCSEL_PERIOD            = 8'h70;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI       = 8'h71;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_LO       = 8'h72;
parameter CROSSTALK_COMPENSATION_PEAK_RATE_MCPS      = 8'h20;
parameter MSRC_CONFIG_TIMEOUT_MACROP                 = 8'h46;

parameter SOFT_RESET_GO2_SOFT_RESET_N                = 8'hBF;
parameter IDENTIFICATION_MODEL_ID                    = 8'hC0;
parameter IDENTIFICATION_REVISION_ID                 = 8'hC2;
parameter OSC_CALIBRATE_VAL                          = 8'hF8;
parameter GLOBAL_CONFIG_VCSEL_WIDTH                  = 8'h32;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_0           = 8'hB0;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_1           = 8'hB1;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_2           = 8'hB2;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_3           = 8'hB3;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_4           = 8'hB4;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_5           = 8'hB5;
parameter GLOBAL_CONFIG_REF_EN_START_SELECT          = 8'hB6;
parameter DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD        = 8'h4E;
parameter DYNAMIC_SPAD_REF_EN_START_OFFSET           = 8'h4F;
parameter POWER_MANAGEMENT_GO1_POWER_FORCE           = 8'h80;
parameter VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV          = 8'h89;
parameter ALGO_PHASECAL_LIM                          = 8'h30;
parameter ALGO_PHASECAL_CONFIG_TIMEOUT               = 8'h30;

////////////////////////////////////////////////////////////////////////////////
    ////
////////////////////////////////////////////////////////////////////////////////
    ////
```

```
////////////////////////////////////////////////////////////////////////////
    ////
// Memory map for variables: ram is global
////////////////////////////////////////////////////////////////////////////
    ////
parameter STOP_VARIABLE = 8'h00;//8-bit
parameter SPAD_COUNT = 8'h01;//8-bit
parameter SPAD_TYPE_IS_APERTURE = 8'h02;//bool
parameter REF_SPAD_MAP = 8'h03; //array
parameter SEQUENCE_STEP_ENABLE_TCC = 8'h09;//bool
parameter SEQUENCE_STEP_ENABLE_MSRC = 8'h0A;//bool
parameter SEQUENCE_STEP_ENABLE_DSS = 8'h0B;//bool
parameter SEQUENCE_STEP_ENABLE_PRE_RANGE = 8'h0C;//bool
parameter SEQUENCE_STEP_ENABLE_FINAL_RANGE = 8'h0D;//bool
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP = 8'h0E;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP = 8'h10;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTM = 8'h12; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS = 8'h14; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS = 8'h16; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTU = 8'h18; //32-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US = 8'h22; //32-b-t
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US = 8'h26; //32 bit
parameter MEASUREMENT_TIMING_BUDGET = 8'h30; //32-bit

//define data_types
parameter VcselPeriodPreRange = 8'h00;
parameter VcselPeriodFinalRange = 8'h01;

//define state parameters
parameter S_RESET = 2'b00;
parameter S_TIMEOUT = 2'b01;
parameter S_WRITE = 2'b10;
parameter S_DONE = 2'b11;

//define state register and counters
reg [1:0] state = 2'b00;
reg [3:0] instruction_count = 4'b0000;

//define output registers
reg done_reg = 1'b0;

reg write_start_reg = 1'b0;
reg write_multi_start_reg = 1'b0;
reg read_start_reg = 1'b0;

reg timer_start_reg = 1'b0;
reg [3:0] timer_param_reg = 4'b0101;
reg timer_reset_reg = 1'b1;

reg [7:0] data_out_reg = 1'b0;
reg [3:0] n_bytes_reg = 4'b0000;
reg [7:0] reg_address_out_reg = 8'h00;

reg [1:0] fnc_sel_reg = 2'b00;

reg fifo_read_en_reg = 1'b0;
```

```
        reg [7:0] fifo_data_out_reg = 8'h00;
        reg fifo_wr_en_reg = 1'b0;
        reg fifo_ext_reset_reg = 1'b1;

        reg [7:0] mem_addr_reg = 8'h00;
        reg [7:0] mem_data_out_reg = 8'h00;
        reg mem_start_reg = 1'b0;
        reg mem_rw_reg = 1'b0;

        reg spad_error_reg = 1'b0;

        //main FSM implementation
        always @(posedge clk) begin
            if(reset) state <= S_RESET;
            else begin
                case(state)
                    S_RESET: begin
                        if(start) state <= S_TIMEOUT;
                        else state <= S_RESET;

                        done_reg <= 1'b0;

                        write_start_reg <= 1'b0;
                        write_multi_start_reg <= 1'b0;
                        read_start_reg <= 1'b0;

                        timer_start_reg <= 1'b0;
                        timer_param_reg <= 4'b0101;
                        timer_reset_reg <= 1'b1;

                        data_out_reg <= 8'h00;
                        n_bytes_reg <= 4'b0000;
                        reg_address_out_reg <= 8'h00;

                        fnc_sel_reg <= 2'b00;

                        fifo_read_en_reg <= 1'b0;
                        fifo_data_out_reg <= 8'h00;
                        fifo_wr_en_reg <= 1'b0;
                        fifo_ext_reset_reg <= 1'b1;

                        mem_addr_reg <= 8'h00;
                        mem_data_out_reg <= 8'h00;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b0;

                        spad_error_reg <= 1'b0;

                        instruction_count <= 4'b0000;
                    end
                    S_TIMEOUT: begin
                        case(instruction_count)
                            4'b0000: begin
                                write_start_reg <= 1'b1;
                                fnc_sel_reg <= 2'b10; //write
                                reg_address_out_reg <= 8'h80;
```

```
                    data_out_reg <= 8'h01; //set LSB

                    instruction_count <= instruction_count + 1;
                    state <= S_TIMEOUT;
                end
                4'b0001: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'hFF;
                        data_out_reg <= 8'h01;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_TIMEOUT;
                end
                4'b0010: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'h00;
                        data_out_reg <= 8'h00;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_TIMEOUT;
                end
                4'b0011: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'hFF;
                        data_out_reg <= 8'h06;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_TIMEOUT;
                end
                4'b0100: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
```

```
                    //setup register read
                    read_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b01; //sel read
                    n_bytes_reg <= 4'b0001;
                    reg_address_out_reg <= 8'h83;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_TIMEOUT;
            end
            4'b0101: begin
                if(!read_done) begin
                    instruction_count <= instruction_count;
                    read_start_reg <= 1'b0;
                end
                else begin
                    fifo_read_en_reg <= 1'b1;
                    instruction_count <= instruction_count + 1;
                end
                state <= S_TIMEOUT;
            end
            4'b0110: begin
                if(!fifo_read_valid) begin
                    instruction_count <= instruction_count;
                    fifo_read_en_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <= 8'h83;
                    data_out_reg <= fifo_data_in | 8'h04; //set LSB

                    instruction_count <= instruction_count + 1;
                end
                state <= S_TIMEOUT;
            end
            4'b0111: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <= 8'hFF;
                    data_out_reg <= 8'h07;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_TIMEOUT;
            end
            4'b1000: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
```

```
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h81;
                data_out_reg <= 8'h01;

                instruction_count <= instruction_count + 1;
            end
            state <= S_TIMEOUT;
        end
        4'b1001: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h80;
                data_out_reg <= 8'h01;

                instruction_count <= instruction_count + 1;
            end
            state <= S_TIMEOUT;
        end
        4'b1010: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h94;
                data_out_reg <= 8'h6B;

                instruction_count <= instruction_count + 1;
            end
            state <= S_TIMEOUT;
        end
        4'b1011: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h83;
                data_out_reg <= 8'h00;

                instruction_count <= instruction_count + 1;
            end
            state <= S_TIMEOUT;
        end
        4'b1100: begin
```

```
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        timer_start_reg <= 1'b1;
                        timer_reset_reg <= 1'b1;
                        timer_param_reg <= 3'b101; //500ms timeout

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_TIMEOUT;
                end
                4'b1101: begin
                    read_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b01; //sel read
                    n_bytes_reg <= 4'b0001;
                    reg_address_out_reg <= 8'h83;

                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;

                    instruction_count <= instruction_count + 1;
                    state <= S_TIMEOUT;
                end
                4'b1110: begin
                    if(!read_done) begin
                        instruction_count <= instruction_count;
                        read_start_reg <= 1'b0;
                    end
                    else begin
                        fifo_read_en_reg <= 1'b1;
                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_TIMEOUT;
                end
                4'b1111: begin
                    if(timer_exp) begin
                        state <= S_RESET;
                        spad_error_reg <= 1'b1;
                        instruction_count <= 4'b0000;
                    end
                    else if(fifo_data_in == 8'h00) begin //loop until
                        0x83 reg is not 0x00
                        instruction_count <= instruction_count - 2;
                        state <= S_TIMEOUT;
                    end
                    else begin
                        instruction_count <= 4'b0000;
                        state <= S_WRITE;
                    end
                    fifo_read_en_reg <= 1'b0;
                end
            endcase
        end
        S_WRITE: begin
```

```
case(instruction_count)
    4'b0000: begin
        write_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b10; //write
        reg_address_out_reg <= 8'h83;
        data_out_reg <= 8'h01; //set LSB

        instruction_count <= instruction_count + 1;
        state <= S_WRITE;
    end
    4'b0001: begin
        if(!write_done) begin
            instruction_count <= instruction_count;
            write_start_reg <= 1'b0;
        end
        else begin
            read_start_reg <= 1'b1;
            fnc_sel_reg <= 2'b01; //sel read
            n_bytes_reg <= 4'b0001;
            reg_address_out_reg <= 8'h92;

            instruction_count <= instruction_count + 1;
        end
        state <= S_WRITE;
    end
    4'b0010: begin
        if(!read_done) begin
            instruction_count <= instruction_count;
            read_start_reg <= 1'b0;
        end
        else begin
            fifo_read_en_reg <= 1'b1;
            instruction_count <= instruction_count + 1;
        end
        state <= S_WRITE;
    end
    4'b0011: begin
        if(!fifo_read_valid) begin
            instruction_count <= instruction_count;
            fifo_read_en_reg <= 1'b0;
        end
        else begin
            mem_addr_reg <= SPAD_COUNT;
            mem_data_out_reg <= fifo_data_in & 8'h7F;
            mem_rw_reg <= 1'b1;
            mem_start_reg <= 1'b1;

            instruction_count <= instruction_count + 1;
        end
        state <= S_WRITE;
    end
    4'b0100: begin
        if(!mem_done) begin
            instruction_count <= instruction_count;
            mem_start_reg <= 1'b0;
        end
```

```
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h81;
                data_out_reg <= 8'h00; //set LSB

                instruction_count <= instruction_count + 1;
            end
            state <= S_WRITE;
        end
        4'b0101: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                mem_addr_reg <= SPAD_TYPE_IS_APERTURE;
                mem_data_out_reg <= (fifo_data_in >> 7) & 8'h01;
                mem_start_reg <= 1'b1;
                mem_rw_reg <= 1'b1;

                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'hFF;
                data_out_reg <= 8'h06; //set LSB

                instruction_count <= instruction_count + 1;
            end
            state <= S_WRITE;
        end
        4'b0110: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
                mem_start_reg <= 1'b0;
            end
            else begin
                read_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b01; //sel read
                n_bytes_reg <= 4'b0001;
                reg_address_out_reg <= 8'h83;

                instruction_count <= instruction_count + 1;
            end
            state <= S_WRITE;
        end
        4'b0111: begin
            if(!read_done) begin
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_WRITE;
```

```
                end
                4'b1000: begin
                    if(!fifo_read_valid) begin
                        instruction_count <= instruction_count;
                        fifo_read_en_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'h83;
                        data_out_reg <= fifo_data_in & ~(8'h04); //set
                            LSB

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_WRITE;
                end
                4'b1001: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'hFF;
                        data_out_reg <= 8'h01; //set LSB

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_WRITE;
                end
                4'b1010: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'h00;
                        data_out_reg <= 8'h01; //set LSB

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_WRITE;
                end
                4'b1011: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                    end
                    else begin
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //write
                        reg_address_out_reg <= 8'hFF;
```

```
                                data_out_reg <= 8'h00; //set LSB

                                instruction_count <= instruction_count + 1;
                            end
                            state <= S_WRITE;
                        end
                        4'b1100: begin
                            if(!write_done) begin
                                instruction_count <= instruction_count;
                                write_start_reg <= 1'b0;
                                state <= S_WRITE;
                            end
                            else begin
                                write_start_reg <= 1'b1;
                                fnc_sel_reg <= 2'b10; //write
                                reg_address_out_reg <= 8'h80;
                                data_out_reg <= 8'h00; //set LSB

                                instruction_count <= 4'b0000;
                                state <= S_DONE;
                            end
                        end
                        default: state <= S_RESET;
                    endcase
                end
                S_DONE: begin
                    if(!write_done) begin
                        write_start_reg <= 1'b0;
                        state <= S_DONE;
                    end
                    else begin
                        state <= S_RESET;
                        write_start_reg <= 1'b0;
                        done_reg <= 1'b1;
                    end
                end
            endcase
        end
    end

    //assign registers to outputs
    assign done = done_reg;

    assign timer_start = timer_start_reg;
    assign timer_param = timer_param_reg;
    assign timer_reset = timer_reset_reg;

   assign write_start = write_start_reg;
   assign write_multi_start = write_multi_start_reg;
   assign read_start = read_start_reg;

    assign data_out = data_out_reg;
    assign n_bytes = n_bytes_reg;
    assign reg_address_out = reg_address_out_reg;

    assign fnc_sel = fnc_sel_reg;
```

```
    assign fifo_read_en = fifo_read_en_reg;
    assign fifo_data_out = fifo_data_out_reg;
    assign fifo_wr_en = fifo_wr_en_reg;
    assign fifo_ext_reset = fifo_ext_reset_reg;

    assign mem_addr = mem_addr_reg;
    assign mem_data_out = mem_data_out_reg;
    assign mem_start = mem_start_reg;
    assign mem_rw = mem_rw_reg;

    assign spad_error = spad_error_reg;

endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    15:52:51 11/29/2017
// Design Name:
// Module Name:    get_spad_info
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module getVcselPulsePeriod(
    //FSM start and done
     input clk,
     input reset,
     input start,
    output done,
     input [7:0] vcsel_period_type,
     output [7:0] vcsel_pulse_period,

     //start and done for read, write, and write_multi FSMs
    output write_start,
    input write_done,
    output write_multi_start,
    input write_multi_done,
    output read_start,
    input read_done,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //data input and output
     output [7:0] reg_address_out,
    output [7:0] data_out,
    output [3:0] n_bytes,

     //read/write/write_multi select
     output [1:0] fnc_sel,

     //FIFO inputs and outputs
     //read fifo
```

```
 input [7:0] fifo_data_in,
 output fifo_read_en,
 input fifo_empty,
 input fifo_read_valid,
 input fifo_underflow,

 //write fifo
 output [7:0] fifo_data_out,
 output fifo_wr_en,
 output fifo_ext_reset,
 input fifo_full,
 input fifo_write_ack,
 input fifo_overflow,

 //inputs and outputs to read RAM
 //ram instantiated in labkit so that gobal variables can be stored and
     updated
 output [7:0] mem_addr,
 output [7:0] mem_data_out,
 input [7:0] mem_data_in,
 output mem_start,
 input mem_done,
 output mem_rw,

 //status
 output error
);

////////////////////////////////////////////////////////////////////////////
    ////
// Register map taken from Pololu VL53L0X open-source library on
    github        //
//
                                    //
////////////////////////////////////////////////////////////////////////////
    ////
parameter SYSRANGE_START                                = 8'h00;

parameter SYSTEM_THRESH_HIGH                            = 8'h0C;
parameter SYSTEM_THRESH_LOW                             = 8'h0E;

parameter SYSTEM_SEQUENCE_CONFIG                        = 8'h01;
parameter SYSTEM_RANGE_CONFIG                           = 8'h09;
parameter SYSTEM_INTERMEASUREMENT_PERIOD                = 8'h04;

parameter SYSTEM_INTERRUPT_CONFIG_GPIO                  = 8'h0A;

parameter GPIO_HV_MUX_ACTIVE_HIGH                       = 8'h84;

parameter SYSTEM_INTERRUPT_CLEAR                        = 8'h0B;

parameter RESULT_INTERRUPT_STATUS                       = 8'h13;
parameter RESULT_RANGE_STATUS                           = 8'h14;

parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_RTN         = 8'hBC;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_RTN          = 8'hC0;
```

```
parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_REF       = 8'hD0;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_REF        = 8'hD4;
parameter RESULT_PEAK_SIGNAL_RATE_REF                 = 8'hB6;

parameter ALGO_PART_TO_PART_RANGE_OFFSET_MM           = 8'h28;

parameter I2C_SLAVE_DEVICE_ADDRESS                    = 8'h8A;

parameter MSRC_CONFIG_CONTROL                         = 8'h60;

parameter PRE_RANGE_CONFIG_MIN_SNR                    = 8'h27;
parameter PRE_RANGE_CONFIG_VALID_PHASE_LOW            = 8'h56;
parameter PRE_RANGE_CONFIG_VALID_PHASE_HIGH           = 8'h57;
parameter PRE_RANGE_MIN_COUNT_RATE_RTN_LIMIT          = 8'h64;

parameter FINAL_RANGE_CONFIG_MIN_SNR                  = 8'h67;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_LOW          = 8'h47;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_HIGH         = 8'h48;
parameter FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT = 8'h44;

parameter PRE_RANGE_CONFIG_SIGMA_THRESH_HI            = 8'h61;
parameter PRE_RANGE_CONFIG_SIGMA_THRESH_LO            = 8'h62;
parameter PRE_RANGE_CONFIG_VCSEL_PERIOD               = 8'h50;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI          = 8'h51;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_LO          = 8'h52;
parameter SYSTEM_HISTOGRAM_BIN                        = 8'h81;
parameter HISTOGRAM_CONFIG_INITIAL_PHASE_SELECT       = 8'h33;
parameter HISTOGRAM_CONFIG_READOUT_CTRL               = 8'h55;
parameter FINAL_RANGE_CONFIG_VCSEL_PERIOD             = 8'h70;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI        = 8'h71;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_LO        = 8'h72;
parameter CROSSTALK_COMPENSATION_PEAK_RATE_MCPS       = 8'h20;
parameter MSRC_CONFIG_TIMEOUT_MACROP                  = 8'h46;

parameter SOFT_RESET_GO2_SOFT_RESET_N                 = 8'hBF;
parameter IDENTIFICATION_MODEL_ID                     = 8'hC0;
parameter IDENTIFICATION_REVISION_ID                  = 8'hC2;
parameter OSC_CALIBRATE_VAL                           = 8'hF8;
parameter GLOBAL_CONFIG_VCSEL_WIDTH                   = 8'h32;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_0            = 8'hB0;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_1            = 8'hB1;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_2            = 8'hB2;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_3            = 8'hB3;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_4            = 8'hB4;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_5            = 8'hB5;
parameter GLOBAL_CONFIG_REF_EN_START_SELECT           = 8'hB6;
parameter DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD         = 8'h4E;
parameter DYNAMIC_SPAD_REF_EN_START_OFFSET            = 8'h4F;
parameter POWER_MANAGEMENT_GO1_POWER_FORCE            = 8'h80;
parameter VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV           = 8'h89;
parameter ALGO_PHASECAL_LIM                           = 8'h30;
parameter ALGO_PHASECAL_CONFIG_TIMEOUT                = 8'h30;

////////////////////////////////////////////////////////////////////////////////
    ////
////////////////////////////////////////////////////////////////////////////////
```

```
    ////

//////////////////////////////////////////////////////////////////////////////
    ////
// Memory map for variables: ram is global
//////////////////////////////////////////////////////////////////////////////
    ////
parameter STOP_VARIABLE = 8'h00;//8-bit
parameter SPAD_COUNT = 8'h01;//8-bit
parameter SPAD_TYPE_IS_APERTURE = 8'h02;//bool
parameter REF_SPAD_MAP = 8'h03; //array
parameter SEQUENCE_STEP_ENABLE_TCC = 8'h09;//bool
parameter SEQUENCE_STEP_ENABLE_MSRC = 8'h0A;//bool
parameter SEQUENCE_STEP_ENABLE_DSS = 8'h0B;//bool
parameter SEQUENCE_STEP_ENABLE_PRE_RANGE = 8'h0C;//bool
parameter SEQUENCE_STEP_ENABLE_FINAL_RANGE = 8'h0D;//bool
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP = 8'h0E;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP = 8'h10;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTM = 8'h12; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS = 8'h14; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS = 8'h16; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTU = 8'h18; //32-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US = 8'h22; //32-b-t
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US = 8'h26; //32 bit
parameter MEASUREMENT_TIMING_BUDGET = 8'h30; //32-bit

//define data_types
parameter VcselPeriodPreRange = 8'h00;
parameter VcselPeriodFinalRange = 8'h01;

//define state parameters
parameter S_RESET = 2'b00;
parameter S_SET_PULSE_PERIOD = 2'b01;
parameter S_DONE = 2'b10;

//define state register and counters
reg [1:0] state = 2'b00;
reg [3:0] instruction_count = 4'b0000;

//define output registers
reg done_reg = 1'b0;

reg write_start_reg = 1'b0;
reg write_multi_start_reg = 1'b0;
reg read_start_reg = 1'b0;

reg timer_start_reg = 1'b0;
reg [3:0] timer_param_reg = 4'b0101;
reg timer_reset_reg = 1'b1;

reg [7:0] data_out_reg = 1'b0;
reg [3:0] n_bytes_reg = 4'b0000;
reg [7:0] reg_address_out_reg = 8'h00;

reg [1:0] fnc_sel_reg = 2'b00;
```

```
        reg fifo_read_en_reg = 1'b0;
        reg [7:0] fifo_data_out_reg = 8'h00;
        reg fifo_wr_en_reg = 1'b0;
        reg fifo_ext_reset_reg = 1'b1;

        reg [7:0] mem_addr_reg = 8'h00;
        reg [7:0] mem_data_out_reg = 8'h00;
        reg mem_start_reg = 1'b0;
        reg mem_rw_reg = 1'b0;

        reg [7:0] vcsel_pulse_period_reg = 8'h00; //hold previous value until next
            iteration

        reg error_reg = 1'b0;

        //main FSM implementation
        always @(posedge clk) begin
            if(reset) state <= S_RESET;
            else begin
                case(state)
                    S_RESET: begin
                        if(start) state <= S_SET_PULSE_PERIOD;
                        else state <= S_RESET;

                        done_reg <= 1'b0;

                        write_start_reg <= 1'b0;
                        write_multi_start_reg <= 1'b0;
                        read_start_reg <= 1'b0;

                        timer_start_reg <= 1'b0;
                        timer_param_reg <= 4'b0101;
                        timer_reset_reg <= 1'b1;

                        data_out_reg <= 8'h00;
                        n_bytes_reg <= 4'b0000;
                        reg_address_out_reg <= 8'h00;

                        fnc_sel_reg <= 2'b00;

                        fifo_read_en_reg <= 1'b0;
                        fifo_data_out_reg <= 8'h00;
                        fifo_wr_en_reg <= 1'b0;
                        fifo_ext_reset_reg <= 1'b1;

                        mem_addr_reg <= 8'h00;
                        mem_data_out_reg <= 8'h00;
                        mem_rw_reg <= 1'b0;
                        mem_start_reg <= 1'b0;

                        error_reg <= 1'b0;

                        instruction_count <= 4'b0000;
                    end
                    S_SET_PULSE_PERIOD: begin
                        if(vcsel_period_type == VcselPeriodPreRange) begin
```

```
                case(instruction_count)
                    4'b0000: begin
                        read_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b01; //sel read
                        n_bytes_reg <= 4'b0001;
                        reg_address_out_reg <=
                            PRE_RANGE_CONFIG_VCSEL_PERIOD;

                        instruction_count <= instruction_count + 1;
                        state <= S_SET_PULSE_PERIOD;
                    end
                    4'b0001: begin
                        if(!read_done) begin
                            instruction_count <= instruction_count;
                            read_start_reg <= 1'b0;
                        end
                        else begin
                            fifo_read_en_reg <= 1'b1;
                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_SET_PULSE_PERIOD;
                    end
                    4'b0010: begin
                        if(!fifo_read_valid) begin
                            instruction_count <= instruction_count;
                            fifo_read_en_reg <= 1'b0;
                        end
                        else begin
                            vcsel_pulse_period_reg <= (fifo_data_in + 1)
                                << 1;
                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_DONE;
                    end
                endcase
            end
            else if(vcsel_period_type == VcselPeriodFinalRange) begin
                case(instruction_count)
                    4'b0000: begin
                        read_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b01; //sel read
                        n_bytes_reg <= 4'b0001;
                        reg_address_out_reg <=
                            FINAL_RANGE_CONFIG_VCSEL_PERIOD;

                        instruction_count <= instruction_count + 1;
                        state <= S_SET_PULSE_PERIOD;
                    end
                    4'b0001: begin
                        if(!read_done) begin
                            instruction_count <= instruction_count;
                            read_start_reg <= 1'b0;
                        end
                        else begin
                            fifo_read_en_reg <= 1'b1;
                            instruction_count <= instruction_count + 1;
```

```
                                end
                                state <= S_SET_PULSE_PERIOD;
                            end
                            4'b0010: begin
                                if(!fifo_read_valid) begin
                                    instruction_count <= instruction_count;
                                    fifo_read_en_reg <= 1'b0;
                                end
                                else begin
                                    vcsel_pulse_period_reg <= (fifo_data_in + 1)
                                        << 1;
                                    instruction_count <= instruction_count + 1;
                                end
                                state <= S_DONE;
                            end
                        endcase
                    end
                    else begin
                        vcsel_pulse_period_reg <= 8'hFF;
                        state <= S_DONE;
                    end
                end
                S_DONE: begin
                    done_reg <= 1'b1;
                    state <= S_RESET;
                end
            endcase
        end
    end

    //assign registers to outputs
    assign done = done_reg;

    assign timer_start = timer_start_reg;
    assign timer_param = timer_param_reg;
    assign timer_reset = timer_reset_reg;

   assign write_start = write_start_reg;
   assign write_multi_start = write_multi_start_reg;
   assign read_start = read_start_reg;

    assign data_out = data_out_reg;
    assign n_bytes = n_bytes_reg;
    assign reg_address_out = reg_address_out_reg;

    assign fnc_sel = fnc_sel_reg;

    assign fifo_read_en = fifo_read_en_reg;
    assign fifo_data_out = fifo_data_out_reg;
    assign fifo_wr_en = fifo_wr_en_reg;
    assign fifo_ext_reset = fifo_ext_reset_reg;

    assign mem_addr = mem_addr_reg;
    assign mem_data_out = mem_data_out_reg;
    assign mem_start = mem_start_reg;
    assign mem_rw = mem_rw_reg;
```

```
    assign vcsel_pulse_period = vcsel_pulse_period_reg;

    assign error = error_reg;

endmodule
```

```
/*

Copyright (c) 2015-2017 Alex Forencich

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

*/

// Language: Verilog 2001

`timescale 1ns / 1ps

/*
 * I2C master
 */
module i2c_master (
    input  wire        clk,
    input  wire        rst,

    /*
     * Host interface
     */
    input  wire [6:0]  cmd_address,              //I2C address to talk to
    input  wire        cmd_start,                   //forces a start command to
        I2C bus upon read or write
    input  wire        cmd_read,                    //initiates I2C byte read
    input  wire        cmd_write,                  //initiates I2C byte write
    input  wire        cmd_write_multiple,        //initiate an I2C multiple byte
        write
    input  wire        cmd_stop,                    //forces a stop command to
        I2C bus upon read or write
    input  wire        cmd_valid,
    output wire        cmd_ready,

     //for writing
    input  wire [7:0]  data_in,
    input  wire        data_in_valid,
    output wire        data_in_ready,
    input  wire        data_in_last,
```

```
     //for reading
    output wire [7:0]  data_out,
    output wire        data_out_valid,
    input  wire        data_out_ready,
    output wire        data_out_last,

    /*
     * I2C interface
     */
    input  wire        scl_i,
    output wire        scl_o,
    output wire        scl_t,
    input  wire        sda_i,
    output wire        sda_o,
    output wire        sda_t,

    /*
     * Status
     */
    output wire        busy,
    output wire        bus_control,
    output wire        bus_active,
    output wire        missed_ack,

    /*
     * Configuration
     */
    input  wire [15:0] prescale,
    input  wire        stop_on_idle
);

/*

I2C

Read
    __    ___ ___ ___ ___ ___ ___ ___         ___ ___ ___ ___ ___ ___ ___ ___
      ___ ___ ___ ___ ___ ___ ___ ___         __
sda   \__/_6_X_5_X_4_X_3_X_2_X_1_X_0_\_R___A_/_7_X_6_X_5_X_4_X_3_X_2_X_1_X_0_
    \_A_/_7_X_6_X_5_X_4_X_3_X_2_X_1_X_0_\_A____/
    ____   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _
     _   _   _   _   _   _   _   _   _   _   ____
scl  ST \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/
    \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ SP

Write
    __    ___ ___ ___ ___ ___ ___ ___ ___     ___ ___ ___ ___ ___ ___ ___ ___
      ___ ___ ___ ___ ___ ___ ___ ___ ___     __
sda   \__/_6_X_5_X_4_X_3_X_2_X_1_X_0_/ W \_A_/_7_X_6_X_5_X_4_X_3_X_2_X_1_X_0_
    \_A_/_7_X_6_X_5_X_4_X_3_X_2_X_1_X_0_/ N \__/
    ____   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _
     _   _   _   _   _   _   _   _   _   _   ____
scl  ST \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/
    \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ SP

Commands:
```

```
read
    read data byte
    set start to force generation of a start condition
    start is implied when bus is inactive or active with write or different
        address
    set stop to issue a stop condition after reading current byte
    if stop is set with read command, then data_out_last will be set

write
    write data byte
    set start to force generation of a start condition
    start is implied when bus is inactive or active with read or different
        address
    set stop to issue a stop condition after writing current byte

write multiple
    write multiple data bytes (until data_in_last)
    set start to force generation of a start condition
    start is implied when bus is inactive or active with read or different
        address
    set stop to issue a stop condition after writing block

stop
    issue stop condition if bus is active

Status:

busy
    module is communicating over the bus

bus_control
    module has control of bus in active state

bus_active
    bus is active, not necessarily controlled by this module

missed_ack
    strobed when a slave ack is missed

Parameters:

prescale
    set prescale to 1/4 of the minimum clock period in units
    of input clk cycles (prescale = Fclk / (FI2Cclk * 4))

stop_on_idle
    automatically issue stop when command input is not valid

Example of interfacing with tristate pins:
(this will work for any tristate bus)

assign scl_i = scl_pin;
assign scl_pin = scl_t ? 1'bz : scl_o;
assign sda_i = sda_pin;
assign sda_pin = sda_t ? 1'bz : sda_o;
```

```
Equivalent code that does not use *_t connections:
(we can get away with this because I2C is open-drain)

assign scl_i = scl_pin;
assign scl_pin = scl_o ? 1'bz : 1'b0;
assign sda_i = sda_pin;
assign sda_pin = sda_o ? 1'bz : 1'b0;

Example of two interconnected I2C devices:

assign scl_1_i = scl_1_o & scl_2_o;
assign scl_2_i = scl_1_o & scl_2_o;
assign sda_1_i = sda_1_o & sda_2_o;
assign sda_2_i = sda_1_o & sda_2_o;

Example of two I2C devices sharing the same pins:

assign scl_1_i = scl_pin;
assign scl_2_i = scl_pin;
assign scl_pin = (scl_1_o & scl_2_o) ? 1'bz : 1'b0;
assign sda_1_i = sda_pin;
assign sda_2_i = sda_pin;
assign sda_pin = (sda_1_o & sda_2_o) ? 1'bz : 1'b0;

Notes:

scl_o should not be connected directly to scl_i, only via AND logic or a tristate
I/O pin.  This would prevent devices from stretching the clock period.

*/

localparam [4:0]
    STATE_IDLE = 4'd0,
    STATE_ACTIVE_WRITE = 4'd1,
    STATE_ACTIVE_READ = 4'd2,
    STATE_START_WAIT = 4'd3,
    STATE_START = 4'd4,
    STATE_ADDRESS_1 = 4'd5,
    STATE_ADDRESS_2 = 4'd6,
    STATE_WRITE_1 = 4'd7,
    STATE_WRITE_2 = 4'd8,
    STATE_WRITE_3 = 4'd9,
    STATE_READ = 4'd10,
    STATE_STOP = 4'd11;

reg [4:0] state_reg = STATE_IDLE, state_next;

localparam [4:0]
    PHY_STATE_IDLE = 5'd0,
    PHY_STATE_ACTIVE = 5'd1,
    PHY_STATE_REPEATED_START_1 = 5'd2,
    PHY_STATE_REPEATED_START_2 = 5'd3,
    PHY_STATE_START_1 = 5'd4,
    PHY_STATE_START_2 = 5'd5,
    PHY_STATE_WRITE_BIT_1 = 5'd6,
```

```
        PHY_STATE_WRITE_BIT_2 = 5'd7,
        PHY_STATE_WRITE_BIT_3 = 5'd8,
        PHY_STATE_READ_BIT_1 = 5'd9,
        PHY_STATE_READ_BIT_2 = 5'd10,
        PHY_STATE_READ_BIT_3 = 5'd11,
        PHY_STATE_READ_BIT_4 = 5'd12,
        PHY_STATE_STOP_1 = 5'd13,
        PHY_STATE_STOP_2 = 5'd14,
        PHY_STATE_STOP_3 = 5'd15;

    reg [4:0] phy_state_reg = STATE_IDLE, phy_state_next;

    reg phy_start_bit;
    reg phy_stop_bit;
    reg phy_write_bit;
    reg phy_read_bit;
    reg phy_release_bus;

    reg phy_tx_data;

    reg phy_rx_data_reg = 1'b0, phy_rx_data_next;

    reg [6:0] addr_reg = 7'd0, addr_next;
    reg [7:0] data_reg = 8'd0, data_next;
    reg last_reg = 1'b0, last_next;

    reg mode_read_reg = 1'b0, mode_read_next;
    reg mode_write_multiple_reg = 1'b0, mode_write_multiple_next;
    reg mode_stop_reg = 1'b0, mode_stop_next;

    reg [16:0] delay_reg = 16'd0, delay_next;
    reg delay_scl_reg = 1'b0, delay_scl_next;
    reg delay_sda_reg = 1'b0, delay_sda_next;

    reg [3:0] bit_count_reg = 4'd0, bit_count_next;

    reg cmd_ready_reg = 1'b0, cmd_ready_next;

    reg data_in_ready_reg = 1'b0, data_in_ready_next;

    reg [7:0] data_out_reg = 8'd0, data_out_next;
    reg data_out_valid_reg = 1'b0, data_out_valid_next;
    reg data_out_last_reg = 1'b0, data_out_last_next;

    reg scl_i_reg = 1'b1;
    reg sda_i_reg = 1'b1;

    reg scl_o_reg = 1'b1, scl_o_next;
    reg sda_o_reg = 1'b1, sda_o_next;

    reg last_scl_i_reg = 1'b1;
    reg last_sda_i_reg = 1'b1;

    reg busy_reg = 1'b0;
    reg bus_active_reg = 1'b0;
    reg bus_control_reg = 1'b0, bus_control_next;
```

```
    reg missed_ack_reg = 1'b0, missed_ack_next;

    assign cmd_ready = cmd_ready_reg;

    assign data_in_ready = data_in_ready_reg;

    assign data_out = data_out_reg;
    assign data_out_valid = data_out_valid_reg;
    assign data_out_last = data_out_last_reg;

    assign scl_o = scl_o_reg;
    assign scl_t = scl_o_reg;
    assign sda_o = sda_o_reg;
    assign sda_t = sda_o_reg;

    assign busy = busy_reg;
    assign bus_active = bus_active_reg;
    assign bus_control = bus_control_reg;
    assign missed_ack = missed_ack_reg;

    wire scl_posedge = scl_i_reg & ~last_scl_i_reg;
    wire scl_negedge = ~scl_i_reg & last_scl_i_reg;
    wire sda_posedge = sda_i_reg & ~last_sda_i_reg;
    wire sda_negedge = ~sda_i_reg & last_sda_i_reg;

    wire start_bit = sda_negedge & scl_i_reg;
    wire stop_bit = sda_posedge & scl_i_reg;

    always @* begin
        state_next = STATE_IDLE;

        phy_start_bit = 1'b0;
        phy_stop_bit = 1'b0;
        phy_write_bit = 1'b0;
        phy_read_bit = 1'b0;
        phy_tx_data = 1'b0;
        phy_release_bus = 1'b0;

        addr_next = addr_reg;
        data_next = data_reg;
        last_next = last_reg;

        mode_read_next = mode_read_reg;
        mode_write_multiple_next = mode_write_multiple_reg;
        mode_stop_next = mode_stop_reg;

        bit_count_next = bit_count_reg;

        cmd_ready_next = 1'b0;

        data_in_ready_next = 1'b0;

        data_out_next = data_out_reg;
        data_out_valid_next = data_out_valid_reg & ~data_out_ready;
        data_out_last_next = data_out_last_reg;
```

```
        missed_ack_next = 1'b0;

        // generate delays
        if (phy_state_reg != PHY_STATE_IDLE && phy_state_reg != PHY_STATE_ACTIVE)
            begin
            // wait for phy operation
            state_next = state_reg;
        end else begin
            // process states
            case (state_reg)
                STATE_IDLE: begin
                    // line idle
                    cmd_ready_next = 1'b1;

                    if (cmd_ready & cmd_valid) begin
                        // command valid
                        if (cmd_read ^ (cmd_write | cmd_write_multiple)) begin
                            // read or write command
                            addr_next = cmd_address;
                            mode_read_next = cmd_read;
                            mode_write_multiple_next = cmd_write_multiple;
                            mode_stop_next = cmd_stop;

                            cmd_ready_next = 1'b0;

                            // start bit
                            if (bus_active) begin
                                state_next = STATE_START_WAIT;
                            end else begin
                                phy_start_bit = 1'b1;
                                bit_count_next = 4'd8;
                                state_next = STATE_ADDRESS_1;
                            end
                        end else begin
                            // invalid or unspecified - ignore
                            state_next = STATE_IDLE;
                        end
                    end else begin
                        state_next = STATE_IDLE;
                    end
                end
                STATE_ACTIVE_WRITE: begin
                    // line active with current address and read/write mode
                    cmd_ready_next = 1'b1;

                    if (cmd_ready & cmd_valid) begin
                        // command valid
                        if (cmd_read ^ (cmd_write | cmd_write_multiple)) begin
                            // read or write command
                            addr_next = cmd_address;
                            mode_read_next = cmd_read;
                            mode_write_multiple_next = cmd_write_multiple;
                            mode_stop_next = cmd_stop;

                            cmd_ready_next = 1'b0;
```

```
                    if (cmd_start || cmd_address != addr_reg || cmd_read)
                        begin
                        // address or mode mismatch or forced start -
                            repeated start

                        // repeated start bit
                        phy_start_bit = 1'b1;
                        bit_count_next = 4'd8;
                        state_next = STATE_ADDRESS_1;
                    end else begin
                        // address and mode match

                        // start write
                        data_in_ready_next = 1'b1;
                        state_next = STATE_WRITE_1;
                    end
                end else if (cmd_stop && !(cmd_read || cmd_write ||
                    cmd_write_multiple)) begin
                    // stop command
                    phy_stop_bit = 1'b1;
                    state_next = STATE_IDLE;
                end else begin
                    // invalid or unspecified - ignore
                    state_next = STATE_ACTIVE_WRITE;
                end
            end else begin
                if (stop_on_idle & cmd_ready & ~cmd_valid) begin
                    // no waiting command and stop_on_idle selected, issue
                        stop condition
                    phy_stop_bit = 1'b1;
                    state_next = STATE_IDLE;
                end else begin
                    state_next = STATE_ACTIVE_WRITE;
                end
            end
        end
        STATE_ACTIVE_READ: begin
            // line active to current address
            cmd_ready_next = ~data_out_valid;

            if (cmd_ready & cmd_valid) begin
                // command valid
                if (cmd_read ^ (cmd_write | cmd_write_multiple)) begin
                    // read or write command
                    addr_next = cmd_address;
                    mode_read_next = cmd_read;
                    mode_write_multiple_next = cmd_write_multiple;
                    mode_stop_next = cmd_stop;

                    cmd_ready_next = 1'b0;

                    if (cmd_start || cmd_address != addr_reg || cmd_write)
                        begin
                        // address or mode mismatch or forced start -
                            repeated start
```

```
                    // write nack for previous read
                    phy_write_bit = 1'b1;
                    phy_tx_data = 1'b1;
                    // repeated start bit
                    state_next = STATE_START;
                end else begin
                    // address and mode match

                    // write ack for previous read
                    phy_write_bit = 1'b1;
                    phy_tx_data = 1'b0;
                    // start next read
                    bit_count_next = 4'd8;
                    data_next = 8'd0;
                    state_next = STATE_READ;
                end
            end else if (cmd_stop && !(cmd_read || cmd_write ||
                cmd_write_multiple)) begin
                // stop command
                // write nack for previous read
                phy_write_bit = 1'b1;
                phy_tx_data = 1'b1;
                // send stop bit
                state_next = STATE_STOP;
            end else begin
                // invalid or unspecified - ignore
                state_next = STATE_ACTIVE_READ;
            end
        end else begin
            if (stop_on_idle & cmd_ready & ~cmd_valid) begin
                // no waiting command and stop_on_idle selected, issue
                    stop condition
                // write ack for previous read
                phy_write_bit = 1'b1;
                phy_tx_data = 1'b1;
                // send stop bit
                state_next = STATE_STOP;
            end else begin
                state_next = STATE_ACTIVE_READ;
            end
        end
    end
    STATE_START_WAIT: begin
        // wait for bus idle

        if (bus_active) begin
            state_next = STATE_START_WAIT;
        end else begin
            // bus is idle, take control
            phy_start_bit = 1'b1;
            bit_count_next = 4'd8;
            state_next = STATE_ADDRESS_1;
        end
    end
    STATE_START: begin
        // send start bit
```

```
            phy_start_bit = 1'b1;
            bit_count_next = 4'd8;
            state_next = STATE_ADDRESS_1;
        end
        STATE_ADDRESS_1: begin
            // send address
            bit_count_next = bit_count_reg - 1;
            if (bit_count_reg > 1) begin
                // send address
                phy_write_bit = 1'b1;
                phy_tx_data = addr_reg[bit_count_reg-2];
                state_next = STATE_ADDRESS_1;
            end else if (bit_count_reg > 0) begin
                // send read/write bit
                phy_write_bit = 1'b1;
                phy_tx_data = mode_read_reg;
                state_next = STATE_ADDRESS_1;
            end else begin
                // read ack bit
                phy_read_bit = 1'b1;
                state_next = STATE_ADDRESS_2;
            end
        end
        STATE_ADDRESS_2: begin
            // read ack bit
            missed_ack_next = phy_rx_data_reg;

            if (mode_read_reg) begin
                // start read
                bit_count_next = 4'd8;
                data_next = 1'b0;
                state_next = STATE_READ;
            end else begin
                // start write
                data_in_ready_next = 1'b1;
                state_next = STATE_WRITE_1;
            end
        end
        STATE_WRITE_1: begin
            data_in_ready_next = 1'b1;

            if (data_in_ready & data_in_valid) begin
                // got data, start write
                data_next = data_in;
                last_next = data_in_last;
                bit_count_next = 4'd8;
                data_in_ready_next = 1'b0;
                state_next = STATE_WRITE_2;
            end else begin
                // wait for data
                state_next = STATE_WRITE_1;
            end
        end
        STATE_WRITE_2: begin
            // send data
```

```
            bit_count_next = bit_count_reg - 1;
            if (bit_count_reg > 0) begin
                // write data bit
                phy_write_bit = 1'b1;
                phy_tx_data = data_reg[bit_count_reg-1];
                state_next = STATE_WRITE_2;
            end else begin
                // read ack bit
                phy_read_bit = 1'b1;
                state_next = STATE_WRITE_3;
            end
        end
        STATE_WRITE_3: begin
            // read ack bit
            missed_ack_next = phy_rx_data_reg;

            if (mode_write_multiple_reg && !last_reg) begin
                // more to write
                state_next = STATE_WRITE_1;
            end else if (mode_stop_reg) begin
                // last cycle and stop selected
                phy_stop_bit = 1'b1;
                state_next = STATE_IDLE;
            end else begin
                // otherwise, return to bus active state
                state_next = STATE_ACTIVE_WRITE;
            end
        end
        STATE_READ: begin
            // read data

            bit_count_next = bit_count_reg - 1;
            data_next = {data_reg[6:0], phy_rx_data_reg};
            if (bit_count_reg > 0) begin
                // read next bit
                phy_read_bit = 1'b1;
                state_next = STATE_READ;
            end else begin
                // output data word
                data_out_next = data_next;
                data_out_valid_next = 1'b1;
                data_out_last_next = 1'b0;
                if (mode_stop_reg) begin
                    // send nack and stop
                    data_out_last_next = 1'b1;
                    phy_write_bit = 1'b1;
                    phy_tx_data = 1'b1;
                    state_next = STATE_STOP;
                end else begin
                    // return to bus active state
                    state_next = STATE_ACTIVE_READ;
                end
            end
        end
        STATE_STOP: begin
            // send stop bit
```

```
                phy_stop_bit = 1'b1;
                state_next = STATE_IDLE;
            end
        endcase
    end
end

always @* begin
    phy_state_next = PHY_STATE_IDLE;

    phy_rx_data_next = phy_rx_data_reg;

    delay_next = delay_reg;
    delay_scl_next = delay_scl_reg;
    delay_sda_next = delay_sda_reg;

    scl_o_next = scl_o_reg;
    sda_o_next = sda_o_reg;

    bus_control_next = bus_control_reg;

    if (phy_release_bus) begin
        // release bus and return to idle state
        sda_o_next = 1'b1;
        scl_o_next = 1'b1;
        delay_scl_next = 1'b0;
        delay_sda_next = 1'b0;
        delay_next = 1'b0;
        phy_state_next = PHY_STATE_IDLE;
    end else if (delay_scl_reg) begin
        // wait for SCL to match command
        delay_scl_next = scl_o_reg & ~scl_i_reg;
        phy_state_next = phy_state_reg;
    end else if (delay_sda_reg) begin
        // wait for SDA to match command
        delay_sda_next = sda_o_reg & ~sda_i_reg;
        phy_state_next = phy_state_reg;
    end else if (delay_reg > 0) begin
        // time delay
        delay_next = delay_reg - 1;
        phy_state_next = phy_state_reg;
    end else begin
        case (phy_state_reg)
            PHY_STATE_IDLE: begin
                // bus idle - wait for start command
                sda_o_next = 1'b1;
                scl_o_next = 1'b1;
                if (phy_start_bit) begin
                    sda_o_next = 1'b0;
                    delay_next = prescale;
                    phy_state_next = PHY_STATE_START_1;
                end else begin
                    phy_state_next = PHY_STATE_IDLE;
                end
            end
            PHY_STATE_ACTIVE: begin
```

```
            // bus active
            if (phy_start_bit) begin
                sda_o_next = 1'b1;
                delay_next = prescale;
                phy_state_next = PHY_STATE_REPEATED_START_1;
            end else if (phy_write_bit) begin
                sda_o_next = phy_tx_data;
                delay_next = prescale;
                phy_state_next = PHY_STATE_WRITE_BIT_1;
            end else if (phy_read_bit) begin
                sda_o_next = 1'b1;
                delay_next = prescale;
                phy_state_next = PHY_STATE_READ_BIT_1;
            end else if (phy_stop_bit) begin
                sda_o_next = 1'b0;
                delay_next = prescale;
                phy_state_next = PHY_STATE_STOP_1;
            end else begin
                phy_state_next = PHY_STATE_ACTIVE;
            end
        end
        PHY_STATE_REPEATED_START_1: begin
            // generate repeated start bit
            //           _______
            // sda XXX/      \________
            //             ________
            // scl _______/        \___
            //

            scl_o_next = 1'b1;
            delay_scl_next = 1'b1;
            delay_next = prescale;
            phy_state_next = PHY_STATE_REPEATED_START_2;
        end
        PHY_STATE_REPEATED_START_2: begin
            // generate repeated start bit
            //           _______
            // sda XXX/      \________
            //             ________
            // scl _______/        \___
            //

            sda_o_next = 1'b0;
            delay_next = prescale;
            phy_state_next = PHY_STATE_START_1;
        end
        PHY_STATE_START_1: begin
            // generate start bit
            //      ___
            // sda     \________
            //      ________
            // scl          \___
            //

            scl_o_next = 1'b0;
            delay_next = prescale;
```

```
            phy_state_next = PHY_STATE_START_2;
        end
        PHY_STATE_START_2: begin
            // generate start bit
            //     ___
            // sda    \_______
            //     _______
            // scl        \___
            //

            bus_control_next = 1'b1;
            phy_state_next = PHY_STATE_ACTIVE;
        end
        PHY_STATE_WRITE_BIT_1: begin
            // write bit
            //      ________
            // sda X________X
            //        ____
            // scl __/    \__

            scl_o_next = 1'b1;
            delay_scl_next = 1'b1;
            delay_next = prescale << 1;
            phy_state_next = PHY_STATE_WRITE_BIT_2;
        end
        PHY_STATE_WRITE_BIT_2: begin
            // write bit
            //      ________
            // sda X________X
            //        ____
            // scl __/    \__

            scl_o_next = 1'b0;
            delay_next = prescale;
            phy_state_next = PHY_STATE_WRITE_BIT_3;
        end
        PHY_STATE_WRITE_BIT_3: begin
            // write bit
            //      ________
            // sda X________X
            //        ____
            // scl __/    \__

            phy_state_next = PHY_STATE_ACTIVE;
        end
        PHY_STATE_READ_BIT_1: begin
            // read bit
            //      ________
            // sda X________X
            //        ____
            // scl __/    \__

            scl_o_next = 1'b1;
            delay_scl_next = 1'b1;
            delay_next = prescale;
            phy_state_next = PHY_STATE_READ_BIT_2;
```

```
end
PHY_STATE_READ_BIT_2: begin
    // read bit
    //      _________
    // sda X_________X
    //         ____
    // scl __/    \__

    phy_rx_data_next = sda_i_reg;
    delay_next = prescale;
    phy_state_next = PHY_STATE_READ_BIT_3;
end
PHY_STATE_READ_BIT_3: begin
    // read bit
    //      _________
    // sda X_________X
    //         ____
    // scl __/    \__

    scl_o_next = 1'b0;
    delay_next = prescale;
    phy_state_next = PHY_STATE_READ_BIT_4;
end
PHY_STATE_READ_BIT_4: begin
    // read bit
    //      _________
    // sda X_________X
    //         ____
    // scl __/    \__

    phy_state_next = PHY_STATE_ACTIVE;
end
PHY_STATE_STOP_1: begin
    // stop bit
    //                ___
    // sda XXX\_______/
    //             _______
    // scl _______/

    scl_o_next = 1'b1;
    delay_scl_next = 1'b1;
    delay_next = prescale;
    phy_state_next = PHY_STATE_STOP_2;
end
PHY_STATE_STOP_2: begin
    // stop bit
    //                ___
    // sda XXX\_______/
    //             _______
    // scl _______/

    sda_o_next = 1'b1;
    delay_next = prescale;
    phy_state_next = PHY_STATE_STOP_3;
end
PHY_STATE_STOP_3: begin
```

```
                // stop bit
                //                  ___
                // sda XXX\________/
                //              ________
                // scl ________/

                bus_control_next = 1'b0;
                phy_state_next = PHY_STATE_IDLE;
            end
        endcase
    end
end

always @(posedge clk) begin
    if (rst) begin
        state_reg <= STATE_IDLE;
        phy_state_reg <= PHY_STATE_IDLE;
        delay_reg <= 16'd0;
        delay_scl_reg <= 1'b0;
        delay_sda_reg <= 1'b0;
        cmd_ready_reg <= 1'b0;
        data_in_ready_reg <= 1'b0;
        data_out_valid_reg <= 1'b0;
        scl_o_reg <= 1'b1;
        sda_o_reg <= 1'b1;
        busy_reg <= 1'b0;
        bus_active_reg <= 1'b0;
        bus_control_reg <= 1'b0;
        missed_ack_reg <= 1'b0;
    end else begin
        state_reg <= state_next;
        phy_state_reg <= phy_state_next;

        delay_reg <= delay_next;
        delay_scl_reg <= delay_scl_next;
        delay_sda_reg <= delay_sda_next;

        cmd_ready_reg <= cmd_ready_next;
        data_in_ready_reg <= data_in_ready_next;
        data_out_valid_reg <= data_out_valid_next;

        scl_o_reg <= scl_o_next;
        sda_o_reg <= sda_o_next;

        busy_reg <= !(state_reg == STATE_IDLE || state_reg == STATE_ACTIVE_WRITE
            || state_reg == STATE_ACTIVE_READ) || !(phy_state_reg ==
            PHY_STATE_IDLE || phy_state_reg == PHY_STATE_ACTIVE);

        if (start_bit) begin
            bus_active_reg <= 1'b1;
        end else if (stop_bit) begin
            bus_active_reg <= 1'b0;
        end else begin
            bus_active_reg <= bus_active_reg;
        end
```

```
            bus_control_reg <= bus_control_next;
            missed_ack_reg <= missed_ack_next;
        end

        phy_rx_data_reg <= phy_rx_data_next;

        addr_reg <= addr_next;
        data_reg <= data_next;
        last_reg <= last_next;

        mode_read_reg <= mode_read_next;
        mode_write_multiple_reg <= mode_write_multiple_next;
        mode_stop_reg <= mode_stop_next;

        bit_count_reg <= bit_count_next;

        data_out_reg <= data_out_next;
        data_out_last_reg <= data_out_last_next;

        scl_i_reg <= scl_i;
        sda_i_reg <= sda_i;
        last_scl_i_reg <= scl_i_reg;
        last_sda_i_reg <= sda_i_reg;
    end

    endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    17:13:32 11/15/2017
// Design Name:
// Module Name:    i2c_read_reg
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
// This module is designed to read the contents of a register of an I2C
// connected device.  The module utilizes the I2C master module created by
// Alex Forencich.  The module requires as an input the 7 bit device I2C address
// and the 8-bit register address one wishes to read from. The end result is the
// specified number of bytes that are requested from the I2C slave are stored
    into
// a FIFO and it is up to the user to retrieve this information before the next
    use
// of this module since the FIFO is cleared upon reset.
//
//  Here's a description of the inputs and what their function is:
//
// [6:0] dev_address: I2C device address - 0x29 default for VL53L0X TOF Sensor
// [7:0] reg_address: byte wide register address for I2C device
//
// clk: 27mhz system clock
// reset: FSM reset
// start: FSM start
// done: FSM done - flashes high when all bytes are read
// byte_width: number of bytes to be read from I2C slave
//
// timer_exp: signal goes high when external timer module has expired its count
// timer_start: module sets signal high when it wishes to begin a timeout stage
// [3:0] timer_param: number of milliseconds each timeout should be
// timer_reset: resets timer for each timeout stage
//
// i2c_data_out_ready: data out to i2c master is ready
// i2c_cmd_ready: unused
// i2c_bus_busy: input from i2c master indicating if master module is
    communicating over bus
// i2c_bus_control: input from i2c master indicating if master has control over
    bus
// i2c_bus_active: input from i2c master indicating if i2c bus is active (not
    necessarily under master's control)
// i2c_missed_ack: input from i2c master indicating if acknoledge bit has been
    missed
```

```
//
// [7:0] i2c_data_out: data byte out to i2c master
// [7:0] i2c_data_in: data byte in from i2c master
// [6:0] i2c_dev_address: output to master providing i2c device address
//
//
//
////////////////////////////////////////////////////////////////////////////////
    /

module i2c_read_reg(
   //data inputs
     input [6:0] dev_address,
     input [7:0] reg_address,

    //FSM inputs for module
     input clk,
     input reset,
     input start,
     output done,
     input [3:0] byte_width,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //communication bus with I2C master module
     //combined with read module, all I2C master inputs should
     //be well defined
     input i2c_data_out_ready,
     input i2c_cmd_ready,
     input i2c_bus_busy,
     input i2c_bus_control,
     input i2c_bus_active,
     input i2c_missed_ack,

    input i2c_data_in_valid,
    output i2c_data_in_ready,
    input i2c_data_in_last,

     output [7:0] i2c_data_out,
     input  [7:0] i2c_data_in,
     output [6:0] i2c_dev_address,

     output i2c_cmd_start,
     output i2c_cmd_read,
     output i2c_cmd_write,
     output i2c_cmd_stop,
     output i2c_cmd_valid,
     output i2c_data_out_valid,
     output [3:0] state_out,

     //FIFO outputs
     output [7:0] data_out,
```

```
    input fifo_read_en,
    output fifo_empty,
    output fifo_read_valid,
    output fifo_underflow,

    //status
    output message_failure
   );
   //write_reg_i2c acts as a module which will, given that the I2C bus is
       available,
   //upon start, will take the data available at reg_address and data and upon
  //response from an i2C master send the register address and the corresponding
      data
   //in the appropriate manner to write to the given register.

   //define state parameters
   parameter S_RESET = 4'b0000;
   parameter S_VALIDATE_BUS = 4'b0001;
   parameter S_VALIDATE_TIMEOUT = 4'b0010;
   parameter S_WRITE_REG_ADDRESS_0 = 4'b0011;
   parameter S_WRITE_REG_ADDRESS_1 = 4'b0100;
   parameter S_WRITE_REG_ADDRESS_TIMEOUT = 4'b0101;
   parameter S_READ_DATA_0 = 4'b0110;
   parameter S_READ_DATA_1 = 4'b0111;
   parameter S_READ_DATA_2 = 4'b1000;
   parameter S_READ_DATA_TIMEOUT = 4'b1001;
   parameter S_FIFO_WRITE_ACK_TIMEOUT_0 = 4'b1010;
   parameter S_FIFO_WRITE_ACK_TIMEOUT_1 = 4'b1011;
   parameter S_CHECK_I2C_FREE = 4'b1100;
   parameter S_CHECK_I2C_FREE_TIMEOUT = 4'b1101;

   //define state registers and counters
   reg [3:0] state = 4'b0000;
   reg [3:0] data_read_count = 4'b0000;

   //define output registers -- outputs that are shared with
   //other I2C communication modules should be tristated as to prevent
   //bus contention -- looking to the future if I have a read/write module
   //for 16/32 bit I2C register writes, then any potentially shared connection
   //should be tristated.
   reg done_reg = 1'b0;
   reg timer_start_reg = 1'b0;
   reg [3:0] timer_param_reg = 3'b001;
   reg timer_reset_reg = 1'b0;

   reg [7:0] i2c_data_out_reg = 8'h00;
   reg [6:0] i2c_dev_address_reg = 7'b0000000;

   reg i2c_cmd_start_reg = 1'b0;
   reg i2c_cmd_write_reg = 1'b0;
   reg i2c_cmd_read_reg = 1'b0;
   reg i2c_cmd_stop_reg = 1'b0;
   reg i2c_cmd_valid_reg = 1'b0;

   reg i2c_data_in_ready_reg = 1'b0;
   reg i2c_data_out_valid_reg = 1'b0;
```

```
reg message_failure_reg = 1'b0;
reg i2c_control_reg = 1'b0;

//define combinational logic
wire bus_valid;
assign bus_valid = ~i2c_bus_busy & ~i2c_bus_active;
wire i2c_bus_free = ~i2c_bus_busy & ~i2c_bus_control;

//define I2C read FIFO
wire fifo_reset;
wire fifo_write_en;
wire fifo_full;
wire fifo_write_ack;
wire fifo_overflow;

reg fifo_reset_reg = 1'b1;
reg fifo_write_en_reg = 1'b0;

FIFO fifo(
    .din(i2c_data_in),
    .dout(data_out),
    .wr_en(fifo_write_en_reg),
    .rd_en(fifo_read_en),
    .clk(clk),
    .rst(fifo_reset_reg),
    .full(fifo_full),
    .empty(fifo_empty),
    .valid(fifo_read_valid),
    .wr_ack(fifo_write_ack),
    .overflow(fifo_overflow),
    .underflow(fifo_underflow)
);

assign fifo_reset = fifo_reset_reg;
assign fifo_write_en = fifo_write_en_reg;

//define state transition diagram
//and state outputs
always @(posedge clk) begin
    if(reset) state <= S_RESET;
    else if(i2c_missed_ack) begin
        state <= S_RESET;
        message_failure_reg <= 1'b1; //missed_ack --> pulse message_failure
    end
    else begin
        case(state)
            S_RESET: begin
                if(start) begin
                    state <= S_VALIDATE_BUS;
                end
                else begin
                    state <= S_RESET;
                end

                //reset values
```

```
            done_reg <= 1'b0;
            timer_start_reg <= 1'b0;
            timer_param_reg <= 3'b001;
            timer_reset_reg <= 1'b1;
            data_read_count <= byte_width;

            i2c_data_out_reg <= 8'h00;
            i2c_dev_address_reg <= dev_address;

            i2c_cmd_start_reg <= 1'b0;
            i2c_cmd_write_reg <= 1'b0;
            i2c_cmd_read_reg <= 1'b0;
            i2c_cmd_stop_reg <= 1'b0;
            i2c_cmd_valid_reg <= 1'b0;

            i2c_data_in_ready_reg <= 1'b0;
            i2c_data_out_valid_reg <= 1'b0;

            message_failure_reg <= 1'b0;
            i2c_control_reg <= 1'b0;

            fifo_reset_reg <= 1'b0;
        end
        S_VALIDATE_BUS: begin
            if(bus_valid) begin
                state <= S_WRITE_REG_ADDRESS_0;
            end
            else begin
                state <= S_VALIDATE_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            //outputs for S_VALIDATE_BUS state -- take ownership of the
            //communication channel:
            i2c_control_reg <= 1'b1;
            fifo_reset_reg <= 1'b0; //clear fifo before reading from I2C
                slave
            timer_reset_reg <= 1'b0; //reset timer
        end
        S_VALIDATE_TIMEOUT: begin
            if(timer_exp) begin
                state <= S_RESET;
                message_failure_reg <= 1'b1;
            end
            else if(bus_valid) begin
                state <= S_WRITE_REG_ADDRESS_0;
            end
            else begin
                state <= S_VALIDATE_TIMEOUT;
            end
            timer_start_reg <= 1'b0;
            timer_start_reg <= 1'b0;
            timer_param_reg <= 3'b001;
        end
        S_WRITE_REG_ADDRESS_0: begin
            if(i2c_data_out_ready) begin
```

```
                state <= S_WRITE_REG_ADDRESS_1;
            end
            else begin
                state <= S_WRITE_REG_ADDRESS_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            //This state is designed to validate whether or not
            //the I2C master is ready to accept data, so we need
            //to tell the master we're getting ready to write.
            i2c_data_out_reg <= reg_address;
            i2c_dev_address_reg <= dev_address;
            i2c_cmd_start_reg <= 1'b1;
            i2c_cmd_write_reg <= 1'b1;
            i2c_cmd_stop_reg <= 1'b1;
            i2c_cmd_valid_reg <= 1'b1;
            i2c_data_out_valid_reg <= 1'b0;
        end
        S_WRITE_REG_ADDRESS_1: begin
            state <= S_READ_DATA_0;
            i2c_data_out_valid_reg <= 1'b1;
            //used to bring down i2c_data_in_valid --> I ionno...the i2c
                master is bizarre
            i2c_data_in_ready_reg <= 1'b1;
        end
        S_WRITE_REG_ADDRESS_TIMEOUT: begin
            if(timer_exp) begin
                state <= S_RESET;
                message_failure_reg <= 1'b1;
            end
            else if(i2c_data_out_ready) begin
                state <= S_WRITE_REG_ADDRESS_1;
            end
            else begin
                state <= S_WRITE_REG_ADDRESS_TIMEOUT;
            end

            timer_start_reg <= 1'b0;
            timer_reset_reg <= 1'b0;
            timer_param_reg <= 3'b001;
        end
        S_READ_DATA_0: begin
            if(i2c_cmd_ready) begin
                state <= S_READ_DATA_TIMEOUT;
                i2c_cmd_read_reg <= 1'b1;
                i2c_cmd_start_reg <= 1'b0;
                i2c_cmd_stop_reg <= 1'b0;
                i2c_cmd_write_reg <= 1'b0;
                i2c_cmd_valid_reg <= 1'b0;
            end
            else begin
                state <= S_READ_DATA_0;
                i2c_cmd_valid_reg <= 1'b0;
            end
        end
        S_READ_DATA_1: begin
```

```
            if(i2c_data_in_valid) begin
                state <= S_READ_DATA_2;
            end
            else begin
                state <= S_READ_DATA_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            i2c_data_in_ready_reg <= 1'b1;
            data_read_count <= data_read_count - 1;
        end
        S_READ_DATA_2: begin
            //Now that we know that the master module is ready
            //to send data over I2C we need to clock the data
            //into a shift register and put it into the FIFO
            if(fifo_overflow) begin
                //too many data bytes were sent -- fifo is 16 bytes deep,
                //if more than 16 bytes were read this constitutes and
                    error
                state <= S_RESET;
                message_failure_reg <= 1'b1;
            end
            else begin
                if(data_read_count > 4'b0001) begin
                    state <= S_FIFO_WRITE_ACK_TIMEOUT_0;
                    timer_start_reg <= 1'b1;
                    timer_reset_reg <= 1'b1;
                end
                else begin
                    state <= S_FIFO_WRITE_ACK_TIMEOUT_1;
                    timer_start_reg <= 1'b1;
                    timer_reset_reg <= 1'b1;
                    i2c_cmd_stop_reg <= 1'b1;
                    i2c_cmd_valid_reg <= 1'b0;
                end
                fifo_write_en_reg <= 1'b1;
                i2c_data_in_ready_reg <= 1'b0;
            end
        end
        S_READ_DATA_TIMEOUT: begin
            if(timer_exp) begin
                state <= S_RESET;
                message_failure_reg <= 1'b1;
            end
            else if(i2c_data_in_valid) begin
                state <= S_READ_DATA_2;
            end
            else begin
                state <= S_READ_DATA_TIMEOUT;
            end
            i2c_cmd_valid_reg <= 1'b1;

            timer_start_reg <= 1'b0;
            timer_reset_reg <= 1'b0;
            timer_param_reg <= 3'b001;
        end
```

```
            S_FIFO_WRITE_ACK_TIMEOUT_0: begin
                if(timer_exp) begin
                    state <= S_RESET;
                    message_failure_reg <= 1'b1;
                end
                else if(fifo_write_ack) begin
                    state <= S_READ_DATA_1;
                end
                else begin
                    state <= S_FIFO_WRITE_ACK_TIMEOUT_0;
                end
                fifo_write_en_reg <= 1'b0;

                timer_start_reg <= 1'b0;
                timer_reset_reg <= 1'b0;
                timer_param_reg <= 3'b001;
            end
            S_FIFO_WRITE_ACK_TIMEOUT_1: begin
                if(timer_exp) begin
                    state <= S_RESET;
                    message_failure_reg <= 1'b1;
                end
                else if(fifo_write_ack) begin
                    state <= S_CHECK_I2C_FREE;
                end
                else begin
                    state <= S_FIFO_WRITE_ACK_TIMEOUT_1;
                end
                fifo_write_en_reg <= 1'b0;

                timer_start_reg <= 1'b0;
                timer_reset_reg <= 1'b0;
                timer_param_reg <= 3'b001;
            end
            S_CHECK_I2C_FREE: begin
                if(i2c_bus_free) begin
                    state <= S_RESET;
                    done_reg <= 1'b1;
                    i2c_cmd_valid_reg <= 1'b0;
                end
                else begin
                    state <= S_CHECK_I2C_FREE_TIMEOUT;
                    timer_start_reg <= 1'b1;
                    timer_reset_reg <= 1'b1;
                end
            end
            S_CHECK_I2C_FREE_TIMEOUT: begin
                if(timer_exp) begin
                    state <= S_RESET;
                    message_failure_reg <= 1'b1;
                end
                else if(i2c_bus_free) begin
                    state <= S_RESET;
                    done_reg <= 1'b1;
                end
                else begin
```

```
                        state <= S_CHECK_I2C_FREE_TIMEOUT;
                    end
                    i2c_cmd_valid_reg <= 1'b0;
                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;
                end
                default: state <= S_RESET;
            endcase
        end
    end

    //assign registers to outputs
    assign done = done_reg;
    assign timer_start = timer_start_reg;
    assign timer_param = timer_param_reg;
    assign timer_reset = timer_reset_reg;

    assign i2c_data_out = i2c_data_out_reg;
    assign i2c_data_out_valid = i2c_data_out_valid_reg;
    assign i2c_data_in_ready = i2c_data_in_ready_reg;
    assign i2c_dev_address = i2c_dev_address_reg;

    assign i2c_cmd_start = i2c_cmd_start_reg;
    assign i2c_cmd_read = i2c_cmd_read_reg;
    assign i2c_cmd_write = i2c_cmd_write_reg;
    assign i2c_cmd_stop = i2c_cmd_stop_reg;
    assign i2c_cmd_valid = i2c_cmd_valid_reg;

    assign message_failure = message_failure_reg;

    //debugging
    assign state_out = state;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    17:50:13 11/12/2017
// Design Name:
// Module Name:    i2c_write_reg
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
// MODULE USE: provide dev_address, reg_address, and data in parallel, then apply
// and active high start pulse.  This module will then time the FSM correctly
    such
// that it communicates with the I2C master and sends data, to reg_address, on
// dev_address.
//
//////////////////////////////////////////////////////////////////////////////////
    /
module i2c_write_reg(
   //data inputs
     input [6:0] dev_address,
     input [7:0] reg_address,
     input [7:0] data,

    //FSM inputs for module
     input clk,
     input reset,
     input start,
     output done,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //communication bus with I2C master module
     input i2c_data_out_ready,
     input i2c_cmd_ready,
     input i2c_bus_busy,
     input i2c_bus_control,
     input i2c_bus_active,
     input i2c_missed_ack,

     output [7:0] i2c_data_out,
     output [6:0] i2c_dev_address,
```

```
 output i2c_cmd_start,
 output i2c_cmd_write_multiple,
 output i2c_cmd_stop,
 output i2c_cmd_valid,
 output i2c_data_out_valid,
 output i2c_data_out_last,
 output [3:0] state_out,

 //status
 output message_failure
);

//define state parameters
parameter S_RESET = 4'b0000;
parameter S_VALIDATE_BUS = 4'b0001;
parameter S_VALIDATE_TIMEOUT = 4'b0010;
parameter S_WRITE_REG_ADDRESS_0 = 4'b0011;
parameter S_WRITE_REG_ADDRESS_1 = 4'b0100;
parameter S_WRITE_REG_ADDRESS_TIMEOUT = 4'b0101;
parameter S_WRITE_DATA_0 = 4'b0110;
parameter S_WRITE_DATA_1 = 4'b0111;
parameter S_WRITE_DATA_TIMEOUT = 4'b1000;
parameter S_CHECK_I2C_FREE = 4'b1001;
parameter S_CHECK_I2C_FREE_TIMEOUT = 4'b1010;

//define state registers and counters
reg [3:0] state = 4'b0000;

//define output registers
reg done_reg = 1'b0;
reg timer_start_reg = 1'b0;
reg [3:0] timer_param_reg = 4'b0001;
reg timer_reset_reg = 1'b1;

reg [7:0] i2c_data_out_reg = 8'h00;
reg [6:0] i2c_dev_address_reg = 7'b0000000;

reg i2c_cmd_start_reg = 1'b0;
reg i2c_cmd_write_multiple_reg = 1'b0;
reg i2c_cmd_stop_reg = 1'b0;
reg i2c_cmd_valid_reg = 1'b0;
reg i2c_data_out_valid_reg = 1'b0;
reg i2c_data_out_last_reg = 1'b0;

reg message_failure_reg = 1'b0;

//define combinational logic
wire bus_valid;
assign bus_valid = ~i2c_bus_busy & ~i2c_bus_active;
wire i2c_bus_free = ~i2c_bus_busy & ~i2c_bus_control;

//define state transition diagram
//and state outputs
always @(posedge clk) begin
    if(reset) state <= S_RESET;
```

```
        else if(i2c_missed_ack) begin
            state <= S_RESET;
            message_failure_reg <= 1'b1; //missed_ack --> pulse message_failure
        end
        else begin
            case(state)
                S_RESET: begin
                    if(start) begin
                        state <= S_VALIDATE_BUS;
                    end
                    else begin
                        state <= S_RESET;
                    end

                    //reset values
                    done_reg <= 1'b0;
                    timer_start_reg <= 1'b0;
                    timer_param_reg <= 4'b0001;
                    timer_reset_reg <= 1'b1;

                    i2c_data_out_reg <= 8'h00;
                    i2c_dev_address_reg <= 7'b0000000;

                    i2c_cmd_start_reg <= 1'b0;
                    i2c_cmd_write_multiple_reg <= 1'b0;
                    i2c_cmd_stop_reg <= 1'b0;
                    i2c_cmd_valid_reg <= 1'b0;
                    i2c_data_out_valid_reg <= 1'b0;
                    i2c_data_out_last_reg <= 1'b0;

                    message_failure_reg <= 1'b0;
                end
                S_VALIDATE_BUS: begin
                    if(bus_valid) begin
                        state <= S_WRITE_REG_ADDRESS_0;
                    end
                    else begin
                        state <= S_VALIDATE_TIMEOUT;
                        timer_start_reg <= 1'b1;
                        timer_reset_reg <= 1'b1;
                    end
                end
                S_VALIDATE_TIMEOUT: begin
                    if(timer_exp) begin
                        state <= S_RESET;
                        message_failure_reg <= 1'b1;
                    end
                    else if(bus_valid) begin
                        state <= S_WRITE_REG_ADDRESS_0;
                    end
                    else begin
                        state <= S_VALIDATE_TIMEOUT;
                    end
                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;
```

```
        end
        S_WRITE_REG_ADDRESS_0: begin
            if(i2c_data_out_ready) begin
                state <= S_WRITE_REG_ADDRESS_1;
            end
            else begin
                state <= S_WRITE_REG_ADDRESS_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            //This state is designed to validate whether or not
            //the I2C master is ready to accept data, so we need
            //to tell the master we're getting ready to write.
            i2c_data_out_reg <= reg_address;
            i2c_dev_address_reg <= dev_address;
            i2c_cmd_start_reg <= 1'b1;
            i2c_cmd_write_multiple_reg <= 1'b1;
            i2c_cmd_stop_reg <= 1'b1;
            i2c_cmd_valid_reg <= 1'b1;
            i2c_data_out_valid_reg <= 1'b1;
            i2c_data_out_last_reg <= 1'b0;
        end
        S_WRITE_REG_ADDRESS_1: begin
            state <= S_WRITE_DATA_0;
            i2c_data_out_valid_reg <= 1'b0;
        end
        S_WRITE_REG_ADDRESS_TIMEOUT: begin
            if(timer_exp) begin
                state <= S_RESET;
                message_failure_reg <= 1'b1;
            end
            else if(i2c_data_out_ready) begin
                state <= S_WRITE_REG_ADDRESS_1;
            end
            else begin
                state <= S_WRITE_REG_ADDRESS_TIMEOUT;
            end
            timer_start_reg <= 1'b0;
            timer_reset_reg <= 1'b0;
            timer_param_reg <= 3'b001;
        end
        S_WRITE_DATA_0: begin
            if(i2c_data_out_ready) begin
                state <= S_WRITE_DATA_1;
            end
            else begin
                state <= S_WRITE_DATA_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            i2c_data_out_reg <= data;
            i2c_data_out_valid_reg <= 1'b1;
            i2c_data_out_last_reg <= 1'b1;
        end
        S_WRITE_DATA_1: begin
            state <= S_CHECK_I2C_FREE;
```

```
                i2c_data_out_valid_reg <= 1'b0;
            end
            S_WRITE_DATA_TIMEOUT: begin
                if(timer_exp) begin
                    state <= S_RESET;
                    message_failure_reg <= 1'b1;
                end
                else if(i2c_data_out_ready) begin
                    state <= S_WRITE_DATA_1;
                end
                else begin
                    state <= S_WRITE_DATA_TIMEOUT;
                end
                timer_start_reg <= 1'b0;
                timer_reset_reg <= 1'b0;
                timer_param_reg <= 3'b001;
            end
            S_CHECK_I2C_FREE: begin
                if(i2c_bus_free) begin
                    state <= S_RESET;
                    done_reg <= 1'b1;
                    i2c_cmd_valid_reg <= 1'b0;
                end
                else begin
                    state <= S_CHECK_I2C_FREE_TIMEOUT;
                    timer_start_reg <= 1'b1;
                    timer_reset_reg <= 1'b1;
                end
            end
            S_CHECK_I2C_FREE_TIMEOUT: begin
                if(timer_exp) begin
                    state <= S_RESET;
                    message_failure_reg <= 1'b1;
                end
                else if(i2c_bus_free) begin
                    state <= S_RESET;
                    done_reg <= 1'b1;
                end
                else begin
                    state <= S_CHECK_I2C_FREE_TIMEOUT;
                end
                i2c_cmd_valid_reg <= 1'b0;
                timer_start_reg <= 1'b0;
                timer_reset_reg <= 1'b0;
                timer_param_reg <= 3'b001;
            end
            default: state <= S_RESET;
        endcase
    end
end

//assign registers to outputs
assign done = done_reg;
assign timer_start = timer_start_reg;
assign timer_param = timer_param_reg;
assign timer_reset = timer_reset_reg;
```

```
    assign i2c_data_out = i2c_data_out_reg;
    assign i2c_dev_address = i2c_dev_address_reg;

    assign i2c_cmd_start = i2c_cmd_start_reg;
    assign i2c_cmd_write_multiple = i2c_cmd_write_multiple_reg;
    assign i2c_cmd_stop = i2c_cmd_stop_reg;
    assign i2c_cmd_valid = i2c_cmd_valid_reg;
    assign i2c_data_out_valid = i2c_data_out_valid_reg;
    assign i2c_data_out_last = i2c_data_out_last_reg;

    assign message_failure = message_failure_reg;

    //debugging
    assign state_out = state;

endmodule
```

```
memory_initialization_radix=16;
memory_initialization_vector=
8800,
8001,
FF01,
0000,
0001,
FF00,
8000,
FF01,
0000,
FF00,
0900,
1000,
1100,
2401,
25FF,
7500,
FF01,
4E2C,
4800,
3020,
FF00,
3009,
5400,
3104,
3203,
4083,
4625,
6000,
2700,
5006,
5100,
5296,
5608,
5730,
6100,
6200,
6400,
6500,
66A0,
FF01,
2232,
4714,
49FF,
4A00,
FF00,
7A0A,
7B00,
7821,
FF01,
2334,
4200,
44FF,
4526,
4605,
```

```
4040,
0E06,
201A,
4340,
FF00,
3403,
3544,
FF01,
3104,
4B09,
4C05,
4D04,
FF00,
4400,
4520,
4708,
4828,
6700,
7004,
7101,
72FE,
7600,
7700,
FF01,
0D01,
FF00,
8001,
01F8,
FF01,
8E01,
0001,
FF00,
8000;
```

```
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
////////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
////////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
```

```
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
////////////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
           ac97_bit_clock,

           vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
           vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
           vga_out_vsync,

           tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
           tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
           tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

           tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
           tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
           tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
           tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

           ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
           ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

           ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
           ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

           clock_feedback_out, clock_feedback_in,

           flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
           flash_reset_b, flash_sts, flash_byte_b,

           rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

           mouse_clock, mouse_data, keyboard_clock, keyboard_data,

           clock_27mhz, clock1, clock2,

           disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
           disp_reset_b, disp_data_in,

           button0, button1, button2, button3, button_enter, button_right,
           button_left, button_down, button_up,

           switch,

           led,

           user1, user2, user3, user4,

           daughtercard,
```

```
        systemace_data, systemace_address, systemace_ce_b,
        systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

        analyzer1_data, analyzer1_clock,
        analyzer2_data, analyzer2_clock,
        analyzer3_data, analyzer3_clock,
        analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
   vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
   tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
   tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
   tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
   tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;
```

```
input  button0, button1, button2, button3, button_enter, button_right,
   button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
      analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
```

```
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;
// disp_data_in is an input
```

```
// Buttons, Switches, and Individual LEDs
//assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
//assign analyzer1_data = 16'h0;
//assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
//assign analyzer3_data = 16'h0;
//assign analyzer3_clock = 1'b1;
//assign analyzer4_data = 16'h0;
//assign analyzer4_clock = 1'b1;

 //Buttons for initiating i2c communications -- for testing (plus overall
     system reset)
 wire reset_button;
 wire clean_button1;
 wire clean_button2;

 //sel for i2c_write, i2c_read, or i2c_write_multi
 //2'b00: none
 //2'b01: read
 //2'b10: write
 //2'b11: write_multi
 wire [1:0] fnc_sel;

 //debouncers to clean up button presses
 debounce db_button0 (
     .clock(clock_27mhz),
     .reset(switch[0]),
     .bouncey(~button0),
     .steady(reset_button)
 );

 debounce db_button1 (
     .clock(clock_27mhz),
```

```
    .reset(switch[0]),
    .bouncey(~button1),
    .steady(clean_button1)
);

debounce db_button2 (
    .clock(clock_27mhz),
    .reset(switch[0]),
    .bouncey(~button2),
    .steady(clean_button2)
);

//pulse signals of varying lengths
wire clk_1000Hz;
wire clk_200Hz;

//signals for connecting to timer module
wire timer_start_read;
wire timer_start_write;
wire timer_start_write_multi;
wire timer_start;

wire [3:0] timer_param_read;
wire [3:0] timer_param_write;
wire [3:0] timer_param_write_multi;
wire [3:0] timer_param;

wire timer_exp;

wire timer_reset_read;
wire timer_reset_write;
wire timer_reset_write_multi;
wire timer_reset;

//muxes to timer module
mux4 m_timer_start (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(timer_start_read),
    .signal2(timer_start_write),
    .signal3(timer_start_write_multi),
    .out(timer_start)
);

mux4 #(.SIGNAL_WIDTH(4)) m_timer_param (
    .sel(fnc_sel),
    .signal0(4'b0000),
    .signal1(timer_param_read),
    .signal2(timer_param_write),
    .signal3(timer_param_write_multi),
    .out(timer_param)
);

mux4 m_timer_reset (
    .sel(fnc_sel),
    .signal0(1'b1),
```

```
        .signal1(timer_reset_read),
        .signal2(timer_reset_write),
        .signal3(timer_reset_write_multi),
        .out(timer_reset)
    );

    //declaration of timer module -- only one per i2c_xxx module to minimize
        hardware
    Timer timer (
        .clk(clock_27mhz),
        .reset(timer_reset),
        .startTimer(timer_start),
        .value(timer_param),
        .enable(clk_1000Hz),
        .time_expired(timer_exp)
    );

    //divider modules for producing single clock high pulses of varying frequency
    divider div (
        .clk(clock_27mhz),
        .reset(timer_start), //synchronized with timer_start signal
        .clk_1000Hz(clk_1000Hz)
    );

    divider #(130000) div2 (
        .clk(clock_27mhz),
        .reset(reset_button), //always on
        .clk_1000Hz(clk_200Hz)
    );


    ////////////////////////////////////////////////////////////////////////////
        ////////////////
    //
    // Signals for i2c_write, i2c_read, and i2c_write_multi modules
    // Way to read signal names:
    // lowercase --> signal type (or signal name)
    // uppercase --> signal origin (i.e., i2c_read, i2c_write, i2c_write_multi
        modules)
    // i2c_ prefix --> signal is between i2c_write, i2c_write, or
        i2c_write_multi, and the i2c master
    //
    ////////////////////////////////////////////////////////////////////////////
        ////////////////

    //signals essential to i2c module communication
    wire [6:0] dev_address;
    wire [7:0] reg_address;
   wire [7:0] data;
    wire [3:0] n_bytes;

    //FSM output status signals
    wire i2c_done_WRITE;
    wire i2c_done_READ;
    wire i2c_done_WRITEMULTI;
```

```
wire message_failure_READ;
wire message_failure_WRITE;
wire message_failure_WRITEMULTI;
wire message_failure; //system failure -- or of all three modules

//FSM input signals
wire i2c_data_out_ready;
wire i2c_cmd_ready;
wire i2c_bus_busy;
wire i2c_bus_control;
wire i2c_bus_active;
wire i2c_missed_ack;

//single byte parallel outputs to i2c master module
wire [7:0] i2c_data_out_READ;
wire [7:0] i2c_data_out_WRITE;
wire [7:0] i2c_data_out_WRITEMULTI;

//single byte parallel input to i2c_read module from i2c master
wire [7:0] i2c_data_in;

//output of device address to i2c master from i2c_xxxx modules
//haven't decided yet if device addresses are going to be static
//so I don't know if these signals are superfluous yet or not
wire [6:0] i2c_dev_address_WRITE;
wire [6:0] i2c_dev_address_READ;
wire [6:0] i2c_dev_address_WRITEMULTI;

//controls to access i2c_read FIFO
wire [7:0] read_data_out;
wire read_data_en;
wire read_data_empty;
wire read_data_valid;
wire read_data_underflow;

//controls for access to i2c_write_multi FIFO
wire [7:0] data_in_WRITEMULTI;
wire write_en_WRITEMULTI;
wire ext_reset_WRITEMULTI;
wire full_WRITEMULTI;
wire write_ack_WRITEMULTI;
wire overflow_WRITEMULTI;
wire test_write_WRITEMULTI;

//FSM input/output for i2c_read
wire i2c_data_in_valid;
wire i2c_data_in_ready;
wire i2c_data_in_last;

//cmd control for i2c_master
wire i2c_cmd_start_READ;
wire i2c_cmd_start_WRITE;
wire i2c_cmd_start_WRITEMULTI;

wire i2c_cmd_write_multiple_WRITE;
wire i2c_cmd_write_multiple_WRITEMULTI;
```

```
 wire i2c_cmd_write_READ;

 wire i2c_cmd_read_READ;

 wire i2c_cmd_stop_READ;
 wire i2c_cmd_stop_WRITE;
 wire i2c_cmd_stop_WRITEMULTI;

 wire i2c_cmd_valid_READ;
 wire i2c_cmd_valid_WRITE;
 wire i2c_cmd_valid_WRITEMULTI;

 //FSM output to i2c_master for controlling byte-wide data output
 wire i2c_data_out_valid_READ;
 wire i2c_data_out_valid_WRITE;
 wire i2c_data_out_valid_WRITEMULTI;

 wire i2c_data_out_last_WRITE;
 wire i2c_data_out_last_WRITEMULTI;

 wire [3:0] state_out_READ;
 wire [3:0] state_out_WRITE;
 wire [3:0] state_out_WRITEMULTI;

 //FSM inputs controled by VL53L0x Module that control start of I2C FSMs
 wire write_start;
wire write_done;

wire write_multi_start;
wire write_multi_done;

 wire read_start;
wire read_done;

 ////////////////////////////////////////////////////////////////////////////////
     ////////////////
 //
 // DECLARATION OF MAIN FSMs: read, write, and write_multi
 //
 // These modules have the main task of controlling the i2c master and
     providing it data
 // at the correct time such that one can read or write some number of bytes
     from the slave
 // device.
 //
 // i2c_write: writes one byte of data to the slave at a specified byte
     register
 // i2c_read: incrementally reads bytes into a FIFO from slave starting at a
     specified register -
 //           resets FIFO upon start of transmission
 // i2c_write_multi: incrementally writes multiple bytes from a FIFO to the
     slave starting at
 //                         a specified register - resets FIFO at the end of
     transmission
 //
```

```
    ////////////////////////////////////////////////////////////////////////////////
        ///////////////

    i2c_read_reg read (
        .dev_address(dev_address),
        .reg_address(reg_address),
        .byte_width(n_bytes),

        .clk(clock_27mhz),
        .reset(reset_button),
        .start(read_start),
        .done(read_done),

        .timer_exp(timer_exp),
        .timer_param(timer_param_read),
        .timer_start(timer_start_read),
        .timer_reset(timer_reset_read),

        .i2c_data_out_ready(i2c_data_out_ready),
        .i2c_cmd_ready(i2c_cmd_ready),
        .i2c_bus_busy(i2c_bus_busy),
        .i2c_bus_control(i2c_bus_control),
       .i2c_bus_active(i2c_bus_active),
       .i2c_missed_ack(i2c_missed_ack),

        .i2c_data_in_valid(i2c_data_in_valid),
        .i2c_data_in_ready(i2c_data_in_ready),
        .i2c_data_in_last(i2c_data_in_last),

        .i2c_data_out(i2c_data_out_READ),
        .i2c_data_in(i2c_data_in),
        .i2c_dev_address(i2c_dev_address_READ),

        .i2c_cmd_start(i2c_cmd_start_READ),
        .i2c_cmd_write(i2c_cmd_write_READ),
        .i2c_cmd_read(i2c_cmd_read_READ),
        .i2c_cmd_stop(i2c_cmd_stop_READ),
        .i2c_cmd_valid(i2c_cmd_valid_READ),
        .i2c_data_out_valid(i2c_data_out_valid_READ),

        .data_out(read_data_out),
        .fifo_read_en(read_data_en),
        .fifo_empty(read_data_empty),
        .fifo_read_valid(read_data_valid),
        .fifo_underflow(read_data_underflow),

        .message_failure(message_failure_READ)
    );

    i2c_write_reg write (
        .dev_address(dev_address),
        .reg_address(reg_address),
        .data(data),

        .clk(clock_27mhz),
        .reset(reset_button),
```

```
    .start(write_start),
    .done(write_done),

    .timer_exp(timer_exp),
    .timer_param(timer_param_write),
    .timer_start(timer_start_write),
    .timer_reset(timer_reset_write),

    .i2c_data_out_ready(i2c_data_out_ready),
    .i2c_cmd_ready(i2c_cmd_ready),
    .i2c_bus_busy(i2c_bus_busy),
    .i2c_bus_control(i2c_bus_control),
   .i2c_bus_active(i2c_bus_active),
   .i2c_missed_ack(i2c_missed_ack),

    .i2c_data_out(i2c_data_out_WRITE),
    .i2c_dev_address(i2c_dev_address_WRITE),

    .i2c_cmd_start(i2c_cmd_start_WRITE),
    .i2c_cmd_write_multiple(i2c_cmd_write_multiple_WRITE),
    .i2c_cmd_stop(i2c_cmd_stop_WRITE),
    .i2c_cmd_valid(i2c_cmd_valid_WRITE),

    .i2c_data_out_valid(i2c_data_out_valid_WRITE),
    .i2c_data_out_last(i2c_data_out_last_WRITE),
    .state_out(state_out_WRITE),

    .message_failure(message_failure_WRITE)
);

i2c_write_reg_multi write_multi (
    .dev_address(dev_address),
    .reg_address(reg_address),

    .clk(clock_27mhz),
    .reset(reset_button),
    .start(write_multi_start),
    .done(write_multi_done),
    .byte_width(n_bytes),

    .timer_exp(timer_exp),
    .timer_param(timer_param_write_multi),
    .timer_start(timer_start_write_multi),
    .timer_reset(timer_reset_write_multi),

    .i2c_data_out_ready(i2c_data_out_ready),
    .i2c_cmd_ready(i2c_cmd_ready),
    .i2c_bus_busy(i2c_bus_busy),
    .i2c_bus_control(i2c_bus_control),
   .i2c_bus_active(i2c_bus_active),
   .i2c_missed_ack(i2c_missed_ack),

    .i2c_data_out(i2c_data_out_WRITEMULTI),
    .i2c_dev_address(i2c_dev_address_WRITEMULTI),

    .i2c_cmd_start(i2c_cmd_start_WRITEMULTI),
```

```
    .i2c_cmd_write_multiple(i2c_cmd_write_multiple_WRITEMULTI),
    .i2c_cmd_stop(i2c_cmd_stop_WRITEMULTI),
    .i2c_cmd_valid(i2c_cmd_valid_WRITEMULTI),

    .i2c_data_out_valid(i2c_data_out_valid_WRITEMULTI),
    .i2c_data_out_last(i2c_data_out_last_WRITEMULTI),
    .state_out(state_out_WRITEMULTI),

    .data(data_in_WRITEMULTI),
    .fifo_wr_en(write_en_WRITEMULTI),
    .fifo_ext_reset(ext_reset_WRITEMULTI),
    .fifo_full(full_WRITEMULTI),
    .fifo_write_ack(write_ack_WRITEMULTI),
    .fifo_overflow(overflow_WRITEMULTI),

    .message_failure(message_failure_WRITEMULTI)
);

////////////////////////////////////////////////////////////////////////////////
    ////////////////
//
// DECLARATION OF SIGNALS AND SIGNAL MUXES FOR I2C MASTER MODULE
//
// One will notice that as defined, signals come out of the FSMs
// meaning that signals going into the i2c master should have opposite parity
    i.e., out/in
// not out/out or in/in.
//
////////////////////////////////////////////////////////////////////////////////
    ////////////////

wire [6:0] i2c_dev_address;
wire i2c_cmd_start;
wire i2c_cmd_read;
wire i2c_cmd_write;
wire i2c_cmd_write_multiple;
wire i2c_cmd_stop;
wire i2c_cmd_valid;

wire [7:0] i2c_data_out;
wire i2c_data_out_valid;
wire i2c_data_out_last;

mux4 #(.SIGNAL_WIDTH(7)) m_i2c_dev_address (
    .sel(fnc_sel),
    .signal0(7'b0000000),
    .signal1(i2c_dev_address_READ),
    .signal2(i2c_dev_address_WRITE),
    .signal3(i2c_dev_address_WRITEMULTI),
    .out(i2c_dev_address)
);

mux4 m_i2c_cmd_start (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_cmd_start_READ),
```

```
    .signal2(i2c_cmd_start_WRITE),
    .signal3(i2c_cmd_start_WRITEMULTI),
    .out(i2c_cmd_start)
);

mux4 m_i2c_cmd_read (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_cmd_read_READ),
    .signal2(1'b0),
    .signal3(1'b0),
    .out(i2c_cmd_read)
);

mux4 m_i2c_cmd_write (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_cmd_write_READ),
    .signal2(1'b0),
    .signal3(1'b0),
    .out(i2c_cmd_write)
);

mux4 m_i2c_cmd_write_multiple (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(1'b0),
    .signal2(i2c_cmd_write_multiple_WRITE),
    .signal3(i2c_cmd_write_multiple_WRITEMULTI),
    .out(i2c_cmd_write_multiple)
);

mux4 m_i2c_cmd_stop (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_cmd_stop_READ),
    .signal2(i2c_cmd_stop_WRITE),
    .signal3(i2c_cmd_stop_WRITEMULTI),
    .out(i2c_cmd_stop)
);

mux4 m_i2c_cmd_valid (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_cmd_valid_READ),
    .signal2(i2c_cmd_valid_WRITE),
    .signal3(i2c_cmd_valid_WRITEMULTI),
    .out(i2c_cmd_valid)
);


mux4 #(.SIGNAL_WIDTH(8)) m_i2c_data_out (
    .sel(fnc_sel),
    .signal0(8'h00),
    .signal1(i2c_data_out_READ),
    .signal2(i2c_data_out_WRITE),
```

```
    .signal3(i2c_data_out_WRITEMULTI),
    .out(i2c_data_out)
);

mux4 m_i2c_data_out_valid (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(i2c_data_out_valid_READ),
    .signal2(i2c_data_out_valid_WRITE),
    .signal3(i2c_data_out_valid_WRITEMULTI),
    .out(i2c_data_out_valid)
);

mux4 m_i2c_data_out_last (
    .sel(fnc_sel),
    .signal0(1'b0),
    .signal1(1'b0),
    .signal2(i2c_data_out_last_WRITE),
    .signal3(i2c_data_out_last_WRITEMULTI),
    .out(i2c_data_out_last)
);

wire cmd_ready;
wire [15:0] prescale;
wire stop_on_idle;

wire sda_i;
wire sda_o;
wire scl_i;
wire scl_o;
wire sda_t;
wire scl_t;

i2c_master master (
    .clk(clock_27mhz),
    .rst(reset_button),
    .cmd_address(i2c_dev_address),
    .cmd_start(i2c_cmd_start),
    .cmd_read(i2c_cmd_read),
    .cmd_write(i2c_cmd_write),
  .cmd_write_multiple(i2c_cmd_write_multiple),
    .cmd_stop(i2c_cmd_stop),
  .cmd_valid(i2c_cmd_valid),
    .cmd_ready(i2c_cmd_ready),

    //for writing
  .data_in(i2c_data_out),
    .data_in_valid(i2c_data_out_valid),
  .data_in_ready(i2c_data_out_ready),
  .data_in_last(i2c_data_out_last),

    //for reading
  .data_out(i2c_data_in),
    .data_out_valid(i2c_data_in_valid),
    .data_out_ready(i2c_data_in_ready),
    .data_out_last(i2c_data_in_last),
```

```
  .scl_i(scl_i),
  .scl_o(scl_o),
  .scl_t(scl_t),
  .sda_i(sda_i),
   .sda_o(sda_o),
  .sda_t(sda_t),

  .busy(i2c_bus_busy),
  .bus_control(i2c_bus_control),
  .bus_active(i2c_bus_active),
  .missed_ack(i2c_missed_ack),

  .prescale(prescale),
  .stop_on_idle(stop_on_idle)
);

//declaration of total failure logic
assign message_failure = message_failure_WRITE | message_failure_READ |
    message_failure_WRITEMULTI;

////////////////////////////////////////////////////////////////////////////////
    ////////////////
//
// Signals for initialization for VL53L0X
//
////////////////////////////////////////////////////////////////////////////////
    ////////////////

wire clk_10Hz;

wire timer_exp_init;
wire timer_start_init;
wire [3:0] timer_param_init;
wire timer_reset_init;

//declaration of timer module -- 100ms timer for counting VL53L0x timeout
Timer timer2 (
    .clk(clock_27mhz),
    .reset(timer_reset_init),
    .startTimer(timer_start_init),
    .value(timer_param_init),
    .enable(clk_10Hz),
    .time_expired(timer_exp_init)
);

//divider modules for producing single clock high pulses of varying frequency

divider #(2700000) div3 ( //pulse every 100ms
    .clk(clock_27mhz),
    .reset(timer_start_init),
    .clk_1000Hz(clk_10Hz)
);

wire clk_1Hz;
divider #(27000000) div4 ( //pulse every 1s
```

```
    .clk(clock_27mhz),
    .reset(reset_button),
    .clk_1000Hz(clk_1Hz)
);

wire init_done;

wire [7:0] log_mem_addr;
wire [7:0] log_mem_data_in;
wire [7:0] log_mem_data_out;

wire mem_start;
wire mem_done;
wire mem_rw;

wire init_error;

wire [4:0] instruction_count_debug;
wire [5:0] instruction_count_timeout_debug;
wire [31:0] timeout_period_us;
wire timing_start;

VL53L0X_INIT sensor_init(
    .reset(reset_button),
    .clk(clock_27mhz),
    .start(clk_1Hz),
    .done(init_done),
    .comm_error(message_failure),

    .write_start(write_start),
    .write_done(write_done),
    .write_multi_start(write_multi_start),
    .write_multi_done(write_multi_done),
    .read_start(read_start),
    .read_done(read_done),

    .timer_exp(timer_exp_init),
    .timer_start(timer_start_init),
    .timer_param(timer_param_init),
    .timer_reset(timer_reset_init),

    .reg_address_out(reg_address),
    .data_out(data),
    .n_bytes(n_bytes),

    .fnc_sel(fnc_sel),

    .fifo_data_out(data_in_WRITEMULTI),
    .fifo_wr_en(write_en_WRITEMULTI),
   .fifo_ext_reset(ext_reset_WRITEMULTI),
   .fifo_full(full_WRITEMULTI),
   .fifo_write_ack(write_ack_WRITEMULTI),
   .fifo_overflow(test_write_WRITEMULTI),

    .fifo_data_in(read_data_out),
   .fifo_read_en(read_data_en),
```

```
   .fifo_empty(read_data_empty),
   .fifo_read_valid(read_data_valid),
   .fifo_underflow(read_data_underflow),

    .mem_addr(log_mem_addr),
   .mem_data_out(log_mem_data_out),
   .mem_data_in(log_mem_data_in),
   .mem_done(mem_done),
    .mem_start(mem_start),
    .mem_rw(mem_rw),

    .init_error(init_error),

    .instruction_count_debug(instruction_count_debug),
    .instruction_count_timeout_debug(instruction_count_timeout_debug),
    .timeout_period_us(timeout_period_us),
    .timing_start_out(timing_start)
);

////////////////////////////////////////////////////////////////////////////////
    ////////////////
//
// Gobal Ram for data storage
//
////////////////////////////////////////////////////////////////////////////////
    ////////////////

wire [7:0] ram_addr;
wire [7:0] ram_data_out;
wire [7:0] ram_data_in;
wire ram_wr_en;

RAM ram(
    .addra(ram_addr),
    .dina(ram_data_out),
    .douta(ram_data_in),
    .wea(ram_wr_en),
    .clka(clock_27mhz)
);

mem_ctl ctl(
    .clk(clock_27mhz),
    .reset(reset_button),
    .start(mem_start),
    .done(mem_done),
    .rw(mem_rw),

    .mem_wr_en(ram_wr_en),
    .mem_addr(ram_addr),
    .mem_data_in(ram_data_in),
    .mem_data_out(ram_data_out),

    .log_mem_addr(log_mem_addr),
    .log_mem_data_in(log_mem_data_in),
    .log_mem_data_out(log_mem_data_out)
);
```

```
    //assigned constants for testing modules
    //assign reg_address = 8'hC0; // 8'b1111_1111
    assign dev_address = 7'h29; // 8'b0101_0010
    //assign data = 8'hFF; //8'b0111_0011
    //assign n_bytes = 4'b0001;

    assign prescale = 16'h80;
    assign stop_on_idle = 1'b1;

    //logic analyzer outputs for debugging
    assign user3[4] = message_failure;
    assign analyzer3_clock = clock_27mhz;
    assign analyzer3_data = {timeout_period_us[15:0]};
    assign analyzer1_data = {timeout_period_us[31:16]};
    assign analyzer4_data = {13'b0, timing_start, user3[1], user3[0]};
    assign analyzer4_clock = clock_27mhz;
   assign analyzer1_clock = clock_27mhz;
    //assign analyzer3_data = {fifo_out_debug, user3[1], user3[0],
        fifo_underflow_debug, state_out_WRITEMULTI, test_done_WRITEMULTI};
    //assign analyzer3_data = {i2c_data_out_WRITE, user3[1], user3[0], 1'b0,
        state_out_WRITE, clk_200Hz};
    assign led = {2'b11, ~instruction_count_timeout_debug};

    //physical pin delegation for i2c communication to slave
    assign scl_i = user3[0];
    assign user3[0] = scl_t ? 1'bz : scl_o;
    assign sda_i = user3[1];
    assign user3[1] = sda_t ? 1'bz : sda_o;

endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    19:31:47 12/07/2017
// Design Name:
// Module Name:    mem_ctl
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
// Memory controller FSM for easy addressing to RAM
// designed around a write-first timing scheme.
//
//////////////////////////////////////////////////////////////////////////////////
    /
module mem_ctl(
     //FSM inputs
    input clk,
    input reset,
    input start,
    output done,
     input rw, //set means write, not set means read

     //to RAM
     output mem_wr_en,
    output [7:0] mem_addr,
    input [7:0] mem_data_in,
    output [7:0] mem_data_out,

     //to logic
    input [7:0] log_mem_addr,
    output [7:0] log_mem_data_in,
    input [7:0] log_mem_data_out
    );

     //define states
     parameter S_RESET = 3'b000;
     parameter S_SET_ADDRESS = 3'b001;
     parameter S_DATA_IN = 3'b010;
     parameter S_WR_ENABLE = 3'b011;
     parameter S_DATA_OUT = 3'b100;
     parameter S_DONE = 3'b101;

     //define state register
     reg [2:0] state = 3'b000;
```

```
        //define output registers
        reg done_reg = 1'b0;
        reg mem_wr_en_reg = 1'b0;
        reg [7:0] mem_addr_reg = 8'h00;
        reg [7:0] mem_data_out_reg = 8'h00;
        reg [7:0] log_mem_data_in_reg = 8'h00;

        always @(posedge clk) begin
           if(reset) state <= S_RESET;
           else begin
               case(state)
                   S_RESET: begin
                       done_reg <= 1'b0;
                       mem_wr_en_reg <= 1'b0;
                       mem_addr_reg <= 8'h00;
                       mem_data_out_reg <= 8'h00;
                       log_mem_data_in_reg <= 8'h00;

                       if(start) state <= S_SET_ADDRESS;
                       else state <= S_RESET;
                   end
                   S_SET_ADDRESS: begin
                       mem_addr_reg <= log_mem_addr;
                       state <= S_DATA_IN;
                   end
                   S_DATA_IN: begin
                       mem_data_out_reg <= log_mem_data_out;
                       state <= S_WR_ENABLE;
                   end
                   S_WR_ENABLE: begin
                       if(rw) mem_wr_en_reg <= 1'b1;
                       else mem_wr_en_reg <= 1'b0;
                       state <= S_DATA_OUT;
                   end
                   S_DATA_OUT: begin
                       log_mem_data_in_reg <= mem_data_in;
                       state <= S_DONE;
                   end
                   S_DONE: begin
                       done_reg <= 1'b1;
                       mem_wr_en_reg <= 1'b0;
                       state <= S_RESET;
                   end
                   //default: state <= S_RESET;
               endcase
           end
        end

       assign done = done_reg;
       assign mem_wr_en = mem_wr_en_reg;
       assign mem_addr = mem_addr_reg;
       assign mem_data_out = mem_data_out_reg;
       assign log_mem_data_in = log_mem_data_in_reg;

    endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    13:07:33 11/20/2017
// Design Name:
// Module Name:    mux4
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module mux4(sel, signal0, signal1, signal2, signal3, out);
     parameter SIGNAL_WIDTH = 1;
    input [1:0] sel;
    input [SIGNAL_WIDTH-1:0] signal0;
    input [SIGNAL_WIDTH-1:0] signal1;
    input [SIGNAL_WIDTH-1:0] signal2;
    input [SIGNAL_WIDTH-1:0] signal3;
    output [SIGNAL_WIDTH-1:0] out;

     reg [SIGNAL_WIDTH-1:0] out_reg = 0;

     always @(*) begin
        case(sel)
            2'b00: out_reg = signal0;
            2'b01: out_reg = signal1;
            2'b10: out_reg = signal2;
            2'b11: out_reg = signal3;
            default: out_reg = signal0;
        endcase
     end

     assign out = out_reg;

endmodule
```

```
/*****************************************************************************
*     This file is owned and controlled by Xilinx and must be used           *
*     solely for design, simulation, implementation and creation of          *
*     design files limited to Xilinx devices or technologies. Use            *
*     with non-Xilinx devices or technologies is expressly prohibited        *
*     and immediately terminates your license.                               *
*                                                                            *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"          *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION        *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION            *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS              *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE       *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY               *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR         *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF        *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS        *
*     FOR A PARTICULAR PURPOSE.                                              *
*                                                                            *
*     Xilinx products are not intended for use in life support               *
*     appliances, devices, or systems. Use in such applications are          *
*     expressly prohibited.                                                  *
*                                                                            *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                   *
*     All rights reserved.                                                   *
*****************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file RAM.v when simulating
// the core, RAM. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module RAM(
    clka,
    dina,
    addra,
    wea,
    douta);


input clka;
input [7 : 0] dina;
input [7 : 0] addra;
input [0 : 0] wea;
output [7 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
```

```
    .C_ADDRA_WIDTH(8),
    .C_ADDRB_WIDTH(8),
    .C_ALGORITHM(1),
    .C_BYTE_SIZE(9),
    .C_COMMON_CLK(0),
    .C_DEFAULT_DATA("0"),
    .C_DISABLE_WARN_BHV_COLL(0),
    .C_DISABLE_WARN_BHV_RANGE(0),
    .C_FAMILY("virtex2"),
    .C_HAS_ENA(0),
    .C_HAS_ENB(0),
    .C_HAS_MEM_OUTPUT_REGS_A(0),
    .C_HAS_MEM_OUTPUT_REGS_B(0),
    .C_HAS_MUX_OUTPUT_REGS_A(0),
    .C_HAS_MUX_OUTPUT_REGS_B(0),
    .C_HAS_REGCEA(0),
    .C_HAS_REGCEB(0),
    .C_HAS_SSRA(0),
    .C_HAS_SSRB(0),
    .C_INIT_FILE_NAME("no_coe_file_loaded"),
    .C_LOAD_INIT_FILE(0),
    .C_MEM_TYPE(0),
    .C_MUX_PIPELINE_STAGES(0),
    .C_PRIM_TYPE(1),
    .C_READ_DEPTH_A(256),
    .C_READ_DEPTH_B(256),
    .C_READ_WIDTH_A(8),
    .C_READ_WIDTH_B(8),
    .C_SIM_COLLISION_CHECK("ALL"),
    .C_SINITA_VAL("0"),
    .C_SINITB_VAL("0"),
    .C_USE_BYTE_WEA(0),
    .C_USE_BYTE_WEB(0),
    .C_USE_DEFAULT_DATA(0),
    .C_USE_ECC(0),
    .C_USE_RAMB16BWER_RST_BHV(0),
    .C_WEA_WIDTH(1),
    .C_WEB_WIDTH(1),
    .C_WRITE_DEPTH_A(256),
    .C_WRITE_DEPTH_B(256),
    .C_WRITE_MODE_A("WRITE_FIRST"),
    .C_WRITE_MODE_B("WRITE_FIRST"),
    .C_WRITE_WIDTH_A(8),
    .C_WRITE_WIDTH_B(8),
    .C_XDEVICEFAMILY("virtex2"))
inst (
    .CLKA(clka),
    .DINA(dina),
    .ADDRA(addra),
    .WEA(wea),
    .DOUTA(douta),
    .ENA(),
    .REGCEA(),
    .SSRA(),
    .CLKB(),
    .DINB(),
```

```
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of RAM is "black_box"

endmodule
```

```
/*****************************************************************************
*     This file is owned and controlled by Xilinx and must be used           *
*     solely for design, simulation, implementation and creation of          *
*     design files limited to Xilinx devices or technologies. Use            *
*     with non-Xilinx devices or technologies is expressly prohibited        *
*     and immediately terminates your license.                               *
*                                                                            *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"          *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION        *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION            *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS              *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE       *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY               *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR         *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF        *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS        *
*     FOR A PARTICULAR PURPOSE.                                              *
*                                                                            *
*     Xilinx products are not intended for use in life support               *
*     appliances, devices, or systems. Use in such applications are          *
*     expressly prohibited.                                                  *
*                                                                            *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                   *
*     All rights reserved.                                                   *
*****************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file ROM.v when simulating
// the core, ROM. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module ROM(
    clka,
    addra,
    douta);


input clka;
input [7 : 0] addra;
output [15 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
        .C_ADDRA_WIDTH(8),
        .C_ADDRB_WIDTH(8),
        .C_ALGORITHM(1),
        .C_BYTE_SIZE(9),
```

```
        .C_COMMON_CLK(0),
        .C_DEFAULT_DATA("0"),
        .C_DISABLE_WARN_BHV_COLL(0),
        .C_DISABLE_WARN_BHV_RANGE(0),
        .C_FAMILY("virtex2"),
        .C_HAS_ENA(0),
        .C_HAS_ENB(0),
        .C_HAS_MEM_OUTPUT_REGS_A(0),
        .C_HAS_MEM_OUTPUT_REGS_B(0),
        .C_HAS_MUX_OUTPUT_REGS_A(0),
        .C_HAS_MUX_OUTPUT_REGS_B(0),
        .C_HAS_REGCEA(0),
        .C_HAS_REGCEB(0),
        .C_HAS_SSRA(0),
        .C_HAS_SSRB(0),
        .C_INIT_FILE_NAME("ROM.mif"),
        .C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(256),
        .C_READ_DEPTH_B(256),
        .C_READ_WIDTH_A(16),
        .C_READ_WIDTH_B(16),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(1),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(256),
        .C_WRITE_DEPTH_B(256),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(16),
        .C_WRITE_WIDTH_B(16),
        .C_XDEVICEFAMILY("virtex2"))
    inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
```

```
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of ROM is "black_box"

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    17:02:04 12/08/2017
// Design Name:
// Module Name:    timeoutMclksToMicroseconds
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module timeoutMclksToMicroseconds(
     input clk,
    input reset,
    input start,
    output done,

    input [15:0] timeout_period_mclks,
    input [7:0] vcsel_period_pclks,
    output [31:0] timeout_period_us
    );

     //define states
     parameter S_RESET = 2'b00;
     parameter S_CONVERT_0 = 2'b01;
     parameter S_CONVERT_1 = 2'b10;
     parameter S_DONE = 2'b11;

     //define tmp registers
     reg [31:0] macro_period_ns = 0;

     //define output registers
     reg done_reg = 1'b0;
     reg [31:0] timeout_period_us_reg = 0;

     //define state register
     reg [1:0] state = 2'b00;

     always @(posedge clk) begin
        case(state)
            S_RESET: begin
                if(start) state <= S_CONVERT_0;
                else state <= S_RESET;

                done_reg <= 1'b0;
```

```
                macro_period_ns <= 0;
            end
            S_CONVERT_0: begin
                macro_period_ns <= (((2304 * vcsel_period_pclks * 1655) + 500) *
                    66) >> 16;
                state <= S_CONVERT_1;
            end
            S_CONVERT_1: begin
                timeout_period_us_reg <= (((timeout_period_mclks *
                    macro_period_ns) + (macro_period_ns >> 1)) * 66) >> 16;
                state <= S_DONE;
            end
            S_DONE: begin
                done_reg <= 1'b1;
                state <= S_RESET;
            end
        endcase
    end

    assign done = done_reg;
    assign timeout_period_us = timeout_period_us_reg;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    17:27:30 12/08/2017
// Design Name:
// Module Name:    timeoutMicrosecondsToMclks
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module timeoutMicrosecondsToMclks(
     input clk,
    input reset,
    input start,
    output done,

    output [15:0] timeout_period_mclks,
    input [7:0] vcsel_period_pclks,
    input [31:0] timeout_period_us
    );

     //define states
     parameter S_RESET = 3'b000;
     parameter S_CONVERT_0 = 3'b001;
     parameter S_CONVERT_1 = 3'b010;
     parameter S_CONVERT_2 = 3'b011;
     parameter S_CONVERT_3 = 3'b100;
     parameter S_DONE = 3'b101;

     //define tmp registers
     reg [31:0] macro_period_ns = 0;
     reg [31:0] tmp0 = 0;

     //define output registers
     reg done_reg = 1'b0;
     reg [31:0] timeout_period_mclks_reg = 0;

     //define state register
     reg [2:0] state = 3'b000;

     //instantiate divider
     wire divider_ready_for_data;
     reg [31:0] dividend_reg;
     reg [31:0] divisor_reg;
```

```
      wire [31:0] quotient;
      wire [31:0] remainder;

      divider_module divider(
         .clk(clk),
         .rfd(divider_ready_for_data),

         .dividend(dividend_reg),
         .divisor(divisor_reg),
         .quotient(quotient),
         .remainder(remainder)
      );

      always @(posedge clk) begin
         case(state)
             S_RESET: begin
                 if(start) state <= S_CONVERT_0;
                 else state <= S_RESET;

                 timeout_period_mclks_reg <= 1'b0;
                 done_reg <= 1'b0;
                 macro_period_ns <= 0;
                 tmp0 <= 0;
             end
             S_CONVERT_0: begin
                 macro_period_ns <= (((2304 * vcsel_period_pclks * 1655) + 500) *
                     66) >> 16;
                 state <= S_CONVERT_1;
             end
             S_CONVERT_1: begin
                 tmp0 <= ((timeout_period_us * 1000) + (macro_period_ns >> 2));
                 state <= S_CONVERT_2;
             end
             S_CONVERT_2: begin
                 dividend_reg <= tmp0;
                 divisor_reg <= macro_period_ns;
                 state <= S_CONVERT_3;
             end
             S_CONVERT_3: begin
                 timeout_period_mclks_reg <= quotient;
                 state <= S_DONE;
             end
             S_DONE: begin
                 done_reg <= 1'b1;
                 state <= S_RESET;
             end
         endcase
      end

     assign done = done_reg;
     assign timeout_period_mclks = timeout_period_mclks_reg;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    00:17:45 10/14/2016
// Design Name:
// Module Name:    Timer
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module Timer(
     input clk,
     input reset,
    input startTimer,
    input [3:0] value,
    input enable,
    output time_expired
    );

     //define state parameters
     parameter S_STALL = 1'b0;
     parameter S_COUNT = 1'b1;

     reg state;
     reg [3:0] count = 4'b0000; //register to hold 4bit delay values
     reg time_expired_reg = 1'b0;

    always @(posedge clk) begin
        if(reset) state <= S_STALL;
        else begin
            case(state)
                S_STALL: begin
                    if(startTimer) state <= S_COUNT;
                    else state <= S_STALL;
                end
                S_COUNT: begin
                    if(count >= value) state <= S_STALL;
                    else state <= S_COUNT;
                end
                default state <= S_STALL;
            endcase
        end

        //define state outputs
```

```
            case(state)
                S_STALL: begin
                    time_expired_reg <= 1'b0;
                    count <= 4'b0000;
                end
                S_COUNT: begin
                    if(enable) begin
                        count <= count + 1;
                        time_expired_reg <= 1'b0;
                    end
                    else if(count >= value) begin
                        time_expired_reg <= 1'b1;
                        count <= 4'b0000;
                    end
                    else begin
                        time_expired_reg <= 1'b0;
                        count <= count;
                    end
                end
                default: begin
                    time_expired_reg <= 1'b0;
                    count <= 4'b0000;
                end
            endcase
        end

        assign time_expired = time_expired_reg;

    endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    14:02:43 11/28/2017
// Design Name:
// Module Name:    VL53L0X_INIT
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /


module VL53L0X_INIT(
     //FSM start and done
     input clk,
     input reset,
     input start,
    output done,
     input comm_error,

     //talk to timer
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //start and done for read, write, and write_multi FSMs
    output write_start,
    input write_done,
    output write_multi_start,
    input write_multi_done,
    output read_start,
    input read_done,

     //data input and output
     output [7:0] reg_address_out,
    output [7:0] data_out,
    output [3:0] n_bytes,

     //read/write/write_multi select
     output [1:0] fnc_sel,

     //FIFO inputs and outputs
```

```
 //read fifo
 input [7:0] fifo_data_in,
 output fifo_read_en,
 input fifo_empty,
 input fifo_read_valid,
 input fifo_underflow,

 //write fifo
 output [7:0] fifo_data_out,
 output fifo_wr_en,
 output fifo_ext_reset,
 input fifo_full,
 input fifo_write_ack,
 input fifo_overflow,

 //inputs and outputs to read RAM
 //ram instantiated in labkit so that gobal variables can be stored and
     updated
 output [7:0] mem_addr,
 output [7:0] mem_data_out,
 input [7:0] mem_data_in,
 output mem_start,
 input mem_done,
 output mem_rw,

 //status
 output init_error,

 //debug
 output [4:0] instruction_count_debug,
 output [5:0] instruction_count_timeout_debug,
 output [31:0] timeout_period_us,
 output timing_start_out
);

////////////////////////////////////////////////////////////////////////////
    ////
// Register map taken from Pololu VL53L0X open-source library on
    github        //
//
                                    //
////////////////////////////////////////////////////////////////////////////
    ////
parameter SYSRANGE_START                                = 8'h00;

parameter SYSTEM_THRESH_HIGH                            = 8'h0C;
parameter SYSTEM_THRESH_LOW                             = 8'h0E;

parameter SYSTEM_SEQUENCE_CONFIG                        = 8'h01;
parameter SYSTEM_RANGE_CONFIG                           = 8'h09;
parameter SYSTEM_INTERMEASUREMENT_PERIOD                = 8'h04;

parameter SYSTEM_INTERRUPT_CONFIG_GPIO                  = 8'h0A;

parameter GPIO_HV_MUX_ACTIVE_HIGH                       = 8'h84;
```

```
parameter SYSTEM_INTERRUPT_CLEAR                        = 8'h0B;

parameter RESULT_INTERRUPT_STATUS                       = 8'h13;
parameter RESULT_RANGE_STATUS                           = 8'h14;

parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_RTN         = 8'hBC;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_RTN          = 8'hC0;
parameter RESULT_CORE_AMBIENT_WINDOW_EVENTS_REF         = 8'hD0;
parameter RESULT_CORE_RANGING_TOTAL_EVENTS_REF          = 8'hD4;
parameter RESULT_PEAK_SIGNAL_RATE_REF                   = 8'hB6;

parameter ALGO_PART_TO_PART_RANGE_OFFSET_MM             = 8'h28;

parameter I2C_SLAVE_DEVICE_ADDRESS                      = 8'h8A;

parameter MSRC_CONFIG_CONTROL                           = 8'h60;

parameter PRE_RANGE_CONFIG_MIN_SNR                      = 8'h27;
parameter PRE_RANGE_CONFIG_VALID_PHASE_LOW              = 8'h56;
parameter PRE_RANGE_CONFIG_VALID_PHASE_HIGH             = 8'h57;
parameter PRE_RANGE_MIN_COUNT_RATE_RTN_LIMIT            = 8'h64;

parameter FINAL_RANGE_CONFIG_MIN_SNR                    = 8'h67;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_LOW            = 8'h47;
parameter FINAL_RANGE_CONFIG_VALID_PHASE_HIGH           = 8'h48;
parameter FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT = 8'h44;

parameter PRE_RANGE_CONFIG_SIGMA_THRESH_HI              = 8'h61;
parameter PRE_RANGE_CONFIG_SIGMA_THRESH_LO              = 8'h62;
parameter PRE_RANGE_CONFIG_VCSEL_PERIOD                 = 8'h50;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI            = 8'h51;
parameter PRE_RANGE_CONFIG_TIMEOUT_MACROP_LO            = 8'h52;
parameter SYSTEM_HISTOGRAM_BIN                          = 8'h81;
parameter HISTOGRAM_CONFIG_INITIAL_PHASE_SELECT         = 8'h33;
parameter HISTOGRAM_CONFIG_READOUT_CTRL                 = 8'h55;
parameter FINAL_RANGE_CONFIG_VCSEL_PERIOD               = 8'h70;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI          = 8'h71;
parameter FINAL_RANGE_CONFIG_TIMEOUT_MACROP_LO          = 8'h72;
parameter CROSSTALK_COMPENSATION_PEAK_RATE_MCPS         = 8'h20;
parameter MSRC_CONFIG_TIMEOUT_MACROP                    = 8'h46;

parameter SOFT_RESET_GO2_SOFT_RESET_N                   = 8'hBF;
parameter IDENTIFICATION_MODEL_ID                       = 8'hC0;
parameter IDENTIFICATION_REVISION_ID                    = 8'hC2;
parameter OSC_CALIBRATE_VAL                             = 8'hF8;
parameter GLOBAL_CONFIG_VCSEL_WIDTH                     = 8'h32;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_0              = 8'hB0;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_1              = 8'hB1;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_2              = 8'hB2;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_3              = 8'hB3;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_4              = 8'hB4;
parameter GLOBAL_CONFIG_SPAD_ENABLES_REF_5              = 8'hB5;
parameter GLOBAL_CONFIG_REF_EN_START_SELECT             = 8'hB6;
parameter DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD           = 8'h4E;
parameter DYNAMIC_SPAD_REF_EN_START_OFFSET              = 8'h4F;
parameter POWER_MANAGEMENT_GO1_POWER_FORCE              = 8'h80;
```

```
parameter VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV          = 8'h89;
parameter ALGO_PHASECAL_LIM                          = 8'h30;
parameter ALGO_PHASECAL_CONFIG_TIMEOUT               = 8'h30;

////////////////////////////////////////////////////////////////////////////////
    ////
////////////////////////////////////////////////////////////////////////////////
    ////

////////////////////////////////////////////////////////////////////////////////
    ////
// Memory map for variables: ram is global
////////////////////////////////////////////////////////////////////////////////
    ////
parameter STOP_VARIABLE = 8'h00;//8-bit
parameter SPAD_COUNT = 8'h01;//8-bit
parameter SPAD_TYPE_IS_APERTURE = 8'h02;//bool
parameter REF_SPAD_MAP = 8'h03; //array
parameter SEQUENCE_STEP_ENABLE_TCC = 8'h09;//bool
parameter SEQUENCE_STEP_ENABLE_MSRC = 8'h0A;//bool
parameter SEQUENCE_STEP_ENABLE_DSS = 8'h0B;//bool
parameter SEQUENCE_STEP_ENABLE_PRE_RANGE = 8'h0C;//bool
parameter SEQUENCE_STEP_ENABLE_FINAL_RANGE = 8'h0D;//bool
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_VPP = 8'h0E;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_VPP = 8'h10;//16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTM = 8'h12; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_MCLKS = 8'h14; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_MCLKS = 8'h16; //16-bit
parameter SEQUENCE_STEP_TIMEOUTS_MSRC_DTU = 8'h18; //32-bit
parameter SEQUENCE_STEP_TIMEOUTS_PRE_RANGE_US = 8'h22; //32-b-t
parameter SEQUENCE_STEP_TIMEOUTS_FINAL_RANGE_US = 8'h26; //32 bit
parameter MEASUREMENT_TIMING_BUDGET = 8'h30; //32-bit

//define data_types
parameter VcselPeriodPreRange = 8'h00;
parameter VcselPeriodFinalRange = 8'h01;

////////////////////////////////////////////////////////////////////////////////
    ////
// 8 byte wide registers for tmp-variables
////////////////////////////////////////////////////////////////////////////////
    ////
reg [7:0] tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;

//define state parameters
parameter S_RESET = 2'b00;
parameter S_DATA_INIT = 2'b01;
parameter S_STATIC_INIT = 2'b10;
parameter S_PERFORM_REF_CALIBRATION = 2'b11;

//define state register and counters
reg [1:0] state = 2'b00;
reg [4:0] instruction_count = 5'b00000;
reg [7:0] count = 8'h00;

//define output registers
```

```
 reg done_reg = 1'b0;

 reg timer_start_reg = 1'b0;
 reg [3:0] timer_param_reg = 4'b0001;
 reg timer_reset_reg = 1'b1;

reg write_start_reg = 1'b0;
reg write_multi_start_reg = 1'b0;
reg read_start_reg = 1'b0;

 reg [7:0] data_out_reg = 1'b0;
 reg [3:0] n_bytes_reg = 4'b0000;
 reg [7:0] reg_address_out_reg = 8'h00;

 reg [1:0] fnc_sel_reg = 2'b00;

 reg fifo_read_en_reg = 1'b0;
 reg [7:0] fifo_data_out_reg = 8'h00;
 reg fifo_wr_en_reg = 1'b0;
 reg fifo_ext_reset_reg = 1'b1;

 reg [7:0] mem_addr_reg = 8'h00;
 reg [7:0] mem_data_out_reg = 8'h00;
 reg mem_start_reg = 1'b0;
 reg mem_rw_reg = 1'b0;

 reg init_error_reg = 1'b0;

 ////////////////////////////////////////////////////////////////////////////
 // Local ROM to store large number of register writes
 ////////////////////////////////////////////////////////////////////////////
 wire [15:0] rom_data;
 reg [7:0] rom_addr_reg = 8'h00;

 ROM INIT_ROM (
     .clka(clk),
     .addra(rom_addr_reg),
     .douta(rom_data)
 );

 ////////////////////////////////////////////////////////////////////////////
 // Sub-FSM GetSPADInfo
 ////////////////////////////////////////////////////////////////////////////
 reg spad_start = 1'b0;
 wire spad_done;

 wire write_start_spad;
 wire write_multi_start_spad;
 wire read_start_spad;

 wire timer_start_spad;
 wire [3:0] timer_param_spad;
 wire timer_reset_spad;

 wire [7:0] reg_address_out_spad;
 wire [7:0] data_out_spad;
```

```
        wire [3:0] n_bytes_spad;

        wire [1:0] fnc_sel_spad;

        wire [7:0] fifo_data_out_spad;
        wire fifo_wr_en_spad;
        wire fifo_ext_reset_spad;
        wire fifo_read_en_spad;

        wire [7:0] mem_addr_spad;
        wire [7:0] mem_data_out_spad;
        wire mem_start_spad;
        wire mem_rw_spad;

        wire spad_error;

        get_spad_info get_spad_info(
            .reset(reset),
            .clk(clk),
            .start(spad_start),
            .done(spad_done),

            .write_start(write_start_spad),
            .write_done(write_done),
            .write_multi_start(write_multi_start_spad),
            .write_multi_done(write_multi_done),
            .read_start(read_start_spad),
            .read_done(read_done),

            .timer_exp(timer_exp),
            .timer_start(timer_start_spad),
            .timer_param(timer_param_spad),
            .timer_reset(timer_reset_spad),

            .reg_address_out(reg_address_out_spad),
            .data_out(data_out_spad),
            .n_bytes(n_bytes_spad),

            .fnc_sel(fnc_sel_spad),

            .fifo_data_out(fifo_data_out_spad),
            .fifo_wr_en(fifo_wr_en_spad),
           .fifo_ext_reset(fifo_ext_reset_spad),
           .fifo_full(fifo_full),
           .fifo_write_ack(fifo_write_ack),
           .fifo_overflow(fifo_overflow),

            .fifo_data_in(fifo_data_in),
           .fifo_read_en(fifo_read_en_spad),
           .fifo_empty(fifo_empty),
           .fifo_read_valid(fifo_read_valid),
           .fifo_underflow(fifo_underflow),

            .mem_addr(mem_addr_spad),
           .mem_data_out(mem_data_out_spad),
           .mem_data_in(mem_data_in),
```

```
    .mem_start(mem_start_spad),
     .mem_done(mem_done),
     .mem_rw(mem_rw_spad),

     .spad_error(spad_error)
);

////////////////////////////////////////////////////////////////////////////
// Sub-FSM getMeasurementTimingBudget
////////////////////////////////////////////////////////////////////////////
reg timing_start = 1'b0;
wire timing_done;
wire [31:0] timing_budget;

wire write_start_timing;
wire write_multi_start_timing;
wire read_start_timing;

wire timer_start_timing;
wire [3:0] timer_param_timing;
wire timer_reset_timing;

wire [7:0] reg_address_out_timing;
wire [7:0] data_out_timing;
wire [3:0] n_bytes_timing;

wire [1:0] fnc_sel_timing;

wire [7:0] fifo_data_out_timing;
wire fifo_wr_en_timing;
wire fifo_ext_reset_timing;
wire fifo_read_en_timing;

wire [7:0] mem_addr_timing;
wire [7:0] mem_data_out_timing;
wire mem_start_timing;
wire mem_rw_timing;

wire timing_error;
wire [5:0] instruction_count_timeout;

getMeasurementTimingBudget getMeasurementTimingBudget(
    .reset(reset),
    .clk(clk),
    .start(timing_start),
    .done(timing_done),
    .timing_budget(timing_budget),

    .write_start(write_start_timing),
    .write_done(write_done),
    .write_multi_start(write_multi_start_timing),
    .write_multi_done(write_multi_done),
    .read_start(read_start_timing),
    .read_done(read_done),

    .timer_exp(timer_exp),
```

```
    .timer_start(timer_start_timing),
    .timer_param(timer_param_timing),
    .timer_reset(timer_reset_timing),

    .reg_address_out(reg_address_out_timing),
    .data_out(data_out_timing),
    .n_bytes(n_bytes_timing),

    .fnc_sel(fnc_sel_timing),

    .fifo_data_out(fifo_data_out_timing),
    .fifo_wr_en(fifo_wr_en_timing),
   .fifo_ext_reset(fifo_ext_reset_timing),
   .fifo_full(fifo_full),
   .fifo_write_ack(fifo_write_ack),
   .fifo_overflow(fifo_overflow),

    .fifo_data_in(fifo_data_in),
   .fifo_read_en(fifo_read_en_timing),
   .fifo_empty(fifo_empty),
   .fifo_read_valid(fifo_read_valid),
   .fifo_underflow(fifo_underflow),

    .mem_addr(mem_addr_timing),
   .mem_data_out(mem_data_out_timing),
   .mem_data_in(mem_data_in),
   .mem_start(mem_start_timing),
    .mem_done(mem_done),
    .mem_rw(mem_rw_timing),

    .timing_budget_error(timing_error),
    .instruction_count_debug(instruction_count_timeout),
    .timeout_period_us(timeout_period_us)
);

////////////////////////////////////////////////////////////////////////////
// Main FSM implementation
////////////////////////////////////////////////////////////////////////////

always @(posedge clk) begin
    if(reset | comm_error) state <= S_RESET;
    else begin
        case(state)
            S_RESET: begin
                if(start) state <= S_DATA_INIT;
                else state <= S_RESET;

                done_reg <= 1'b0;

                timer_start_reg <= 1'b0;
                timer_param_reg <= 4'b0001;
                timer_reset_reg <= 1'b1;

                write_start_reg <= 1'b0;
                write_multi_start_reg <= 1'b0;
                read_start_reg <= 1'b0;
```

```
        data_out_reg <= 1'b0;
        n_bytes_reg <= 4'b0000;
        reg_address_out_reg <= 8'h00;

        fnc_sel_reg <= 2'b00;

        fifo_read_en_reg <= 1'b0;
        fifo_data_out_reg <= 8'h00;
        fifo_wr_en_reg <= 1'b0;
        fifo_ext_reset_reg <= 1'b1;

        rom_addr_reg <= 8'h00;
        mem_addr_reg <= 8'h00;
        mem_data_out_reg <= 8'h00;
        mem_start_reg <= 1'b0;
        mem_rw_reg <= 1'b0;

        init_error_reg <= 1'b0;

        spad_start <= 1'b0;
        timing_start <= 1'b0;

        instruction_count <= 5'b00000;
        count <= 8'h00;

        tmp0 <= 8'h00;
        tmp0 <= 8'h00;
        tmp2 <= 8'h00;
        tmp3 <= 8'h00;
        tmp4 <= 8'h00;
        tmp5 <= 8'h00;
        tmp6 <= 8'h00;
        tmp7 <= 8'h00;
    end
    S_DATA_INIT: begin
        case(instruction_count)
            ////////////////////////////////////////////////////////
                ////// 
            // Setting IO voltage to 2.8 V
            ////////////////////////////////////////////////////////
                ////// 
            5'b00000: begin
                //setup register read
                read_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b01; //sel read
                n_bytes_reg <= 4'b0001;
                reg_address_out_reg <=
                    VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV;

                state <= S_DATA_INIT;
                instruction_count <= instruction_count + 1;
            end
            5'b00001: begin
                //read data out of FIFO
                if(!read_done) begin
```

```
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_DATA_INIT;
        end
        5'b00010: begin
            //WRITE DATA USING VALUE FROM READ
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <=
                    VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV;
                data_out_reg <= fifo_data_in | 8'h01; //set LSB

                instruction_count <= instruction_count + 1;
            end
            state <= S_DATA_INIT;
        end
        ////////////////////////////////////////////////////////
            //////
        // Set I2C standard mode
        ////////////////////////////////////////////////////////
            //////
        5'b00011: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= rom_data[15:8];
                data_out_reg <= rom_data[7:0];

                if(rom_addr_reg < 3) instruction_count <=
                    instruction_count;
                else instruction_count <= instruction_count + 1;

                rom_addr_reg <= rom_addr_reg + 1;
            end
            state <= S_DATA_INIT;
        end
        5'b00100: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
```

```
                    //setup register read
                    read_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b01; //sel read
                    n_bytes_reg <= 4'b0001;
                    reg_address_out_reg <= 8'h91;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_DATA_INIT;
            end
            5'b00101: begin
                if(!read_done) begin
                    instruction_count <= instruction_count;
                    read_start_reg <= 1'b0;
                end
                else begin
                    fifo_read_en_reg <= 1'b1;
                    instruction_count <= instruction_count + 1;
                end
                state <= S_DATA_INIT;
            end
            5'b00110: begin
                if(!fifo_read_valid) begin
                    instruction_count <= instruction_count;
                    fifo_read_en_reg <= 1'b0;
                end
                else begin
                    //write STOP_VARIABLE to ram
                    mem_addr_reg <= STOP_VARIABLE;
                    mem_data_out_reg <= fifo_data_in;
                    mem_start_reg <= 1'b1;
                    mem_rw_reg <= 1'b1;

                    //start next write after storing data for ROM
                        addr 4
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <= rom_data[15:8];
                    data_out_reg <= rom_data[7:0];
                    rom_addr_reg <= rom_addr_reg + 1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_DATA_INIT;
            end
            5'b00111: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                    mem_start_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <= rom_data[15:8];
                    data_out_reg <= rom_data[7:0];
```

```
            if(rom_addr_reg < 6) instruction_count <=
                instruction_count;
            else instruction_count <= instruction_count + 1;

            rom_addr_reg <= rom_addr_reg + 1;
        end
        state <= S_DATA_INIT;
    end
    ////////////////////////////////////////////////////////
        //
    // disable SIGNAL_RATE_MSRC (bit 1)
    // and SIGNAL_RATE_PRE_RANGE (bit 4) limit checks
    ////////////////////////////////////////////////////////
        //
    5'b01000: begin
        //setup register read
        if(!write_done) begin
            instruction_count <= instruction_count;
            write_start_reg <= 1'b0;
        end
        else begin
            read_start_reg <= 1'b1;
            fnc_sel_reg <= 2'b01; //sel read
            n_bytes_reg <= 4'b0001;
            reg_address_out_reg <= MSRC_CONFIG_CONTROL;

            instruction_count <= instruction_count + 1;
        end
        state <= S_DATA_INIT;
    end
    5'b01001: begin
        if(!read_done) begin
            instruction_count <= instruction_count;
            read_start_reg <= 1'b0;
        end
        else begin
            fifo_read_en_reg <= 1'b1;
            instruction_count <= instruction_count + 1;
        end
        state <= S_DATA_INIT;
    end
    5'b01010: begin
        if(!fifo_read_valid) begin
            instruction_count <= instruction_count;
            fifo_read_en_reg <= 1'b0;
        end
        else begin
            write_start_reg <= 1'b1;
            fnc_sel_reg <= 2'b10; //write
            reg_address_out_reg <= MSRC_CONFIG_CONTROL;
            data_out_reg <= fifo_data_in | 8'h12; //set LSB

            instruction_count <= instruction_count + 1;
        end
        state <= S_DATA_INIT;
```

```
end
////////////////////////////////////////////////////////
    //
// SET SIGNAL RATE LIMIT = 0.25 MCPS
////////////////////////////////////////////////////////
    //
5'b01011: begin
    //write multiple bytes to write_multi FIFO
    if(!write_done) begin
        instruction_count <= instruction_count;
        write_start_reg <= 1'b0;
    end
    else begin
        fifo_wr_en_reg <= 1'b1;
        fifo_data_out_reg <= 8'h00; //32 in hex -->
            lim_MCPS = 0.25
        instruction_count <= instruction_count + 1;
    end
end
5'b01100: begin
    if(!fifo_write_ack) begin
        instruction_count <= instruction_count;
        fifo_wr_en_reg <= 1'b0;
    end
    else begin
        fifo_wr_en_reg <= 1'b1;
        fifo_data_out_reg <= 8'h20; //32 in hex -->
            lim_MCPS = 0.25
        instruction_count <= instruction_count + 1;
    end
end
5'b01101: begin
    if(!fifo_write_ack) begin
        instruction_count <= instruction_count;
        fifo_wr_en_reg <= 1'b0;
    end
    else begin
        //setup register write_multi
        write_multi_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b11; //sel write_multi
        n_bytes_reg <= 4'b0010;
        reg_address_out_reg <=
            FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT;

        instruction_count <= instruction_count + 1;
    end
    state <= S_DATA_INIT;
end
////////////////////////////////////////////////////////
    //////
// SET SYSTEM_SEQUENCE_CONFIG
////////////////////////////////////////////////////////
    //////
5'b01110: begin
    if(!write_multi_done) begin
        instruction_count <= instruction_count;
```

```
                        write_multi_start_reg <= 1'b0;
                    end
                    else begin
                        //setup register write
                        write_start_reg <= 1'b1;
                        fnc_sel_reg <= 2'b10; //sel write
                        reg_address_out_reg <= SYSTEM_SEQUENCE_CONFIG;
                        data_out_reg <= 8'hFF;

                        instruction_count <= instruction_count + 1;
                    end
                    state <= S_DATA_INIT;
                end
                5'b01111: begin
                    if(!write_done) begin
                        instruction_count <= instruction_count;
                        write_start_reg <= 1'b0;
                        state <= S_DATA_INIT;
                    end
                    else begin
                        state <= S_STATIC_INIT;
                        instruction_count <= 4'b0000;
                    end
                end
                default: state <= S_RESET;
            endcase
        end
        S_STATIC_INIT: begin
            case(instruction_count)
                ////////////////////////////////////////////////////////
                    //////
                // Set SPAD Map
                ////////////////////////////////////////////////////////
                    //////
                5'b00000: begin
                    //shift FSM Control to getSPADInfo
                    spad_start <= 1'b1;

                    write_start_reg <= write_start_spad;
                    write_multi_start_reg <= write_multi_start_spad;
                    read_start_reg <= read_start_spad;

                    timer_start_reg <= timer_start_spad;
                    timer_param_reg <= timer_param_spad;
                    timer_reset_reg <= timer_reset_spad;

                    reg_address_out_reg <= reg_address_out_spad;
                    data_out_reg <= data_out_spad;
                    n_bytes_reg <= n_bytes_spad;

                    fnc_sel_reg <= fnc_sel_spad;

                    fifo_data_out_reg <= fifo_data_out_spad;
                    fifo_wr_en_reg <= fifo_wr_en_spad;
                    fifo_ext_reset_reg <= fifo_ext_reset_spad;
                    fifo_read_en_reg <= fifo_read_en_spad;
```

```
                mem_addr_reg <= mem_addr_spad;
                mem_data_out_reg <= mem_data_out_spad;
                mem_start_reg <= mem_start_spad;
                mem_rw_reg <= mem_rw_spad;

                init_error_reg <= spad_error;

                instruction_count <= instruction_count + 1;
                state <= S_STATIC_INIT;
            end
            5'b00001: begin
                if(!spad_done) begin
                    //let spad-info fsm control outputs until it is
                        finished.
                    spad_start <= 1'b0;

                    write_start_reg <= write_start_spad;
                    write_multi_start_reg <= write_multi_start_spad;
                    read_start_reg <= read_start_spad;

                    timer_start_reg <= timer_start_spad;
                    timer_param_reg <= timer_param_spad;
                    timer_reset_reg <= timer_reset_spad;

                    reg_address_out_reg <= reg_address_out_spad;
                    data_out_reg <= data_out_spad;
                    n_bytes_reg <= n_bytes_spad;

                    fnc_sel_reg <= fnc_sel_spad;

                    fifo_data_out_reg <= fifo_data_out_spad;
                    fifo_wr_en_reg <= fifo_wr_en_spad;
                    fifo_ext_reset_reg <= fifo_ext_reset_spad;
                    fifo_read_en_reg <= fifo_read_en_spad;

                    mem_addr_reg <= mem_addr_spad;
                    mem_data_out_reg <= mem_data_out_spad;
                    mem_start_reg <= mem_start_spad;
                    mem_rw_reg <= mem_rw_spad;

                    init_error_reg <= spad_error;

                    state <= S_STATIC_INIT;
                    instruction_count <= instruction_count;
                end
                else begin
                    state <= S_STATIC_INIT;
                    instruction_count <= instruction_count + 1;
                end
            end
            5'b00010: begin
                //setup register read
                read_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b01; //sel read
                n_bytes_reg <= 4'b0110;
```

```
                reg_address_out_reg <=
                    GLOBAL_CONFIG_SPAD_ENABLES_REF_0;

                instruction_count <= instruction_count + 1;
                state <= S_STATIC_INIT;
            end
            5'b00011: begin
                if(!read_done) begin
                    instruction_count <= instruction_count;
                    read_start_reg <= 1'b0;
                end
                else begin
                    fifo_read_en_reg <= 1'b1;
                    instruction_count <= instruction_count + 1;
                end
                mem_addr_reg <= REF_SPAD_MAP;
                state <= S_STATIC_INIT;
            end
            5'b00100: begin
                if(!fifo_read_valid) begin
                    instruction_count <= instruction_count;
                    fifo_read_en_reg <= 1'b0;
                end
                else begin
                    //write ref_spad_map to ram (starting from
                        &REF_SPAD_MAP and ending on
                    // &REF_SPAD_MAP + 6)
                    mem_data_out_reg <= fifo_data_in;
                    mem_start_reg <= 1'b1;
                    mem_rw_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b00101: begin
                if(!mem_done) begin
                    instruction_count <= instruction_count;
                    mem_start_reg <= 1'b0;
                end
                else begin
                    if(mem_addr_reg < (REF_SPAD_MAP + 5)) begin
                        instruction_count <= instruction_count - 1;
                        mem_addr_reg <= mem_addr_reg + 1;
                        fifo_read_en_reg <= 1'b1;
                    end
                    else begin
                        instruction_count <= instruction_count + 1;
                    end
                end
                state <= S_STATIC_INIT;
            end
            5'b00110: begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'hFF;
```

```
                data_out_reg <= 8'h01;

                instruction_count <= instruction_count + 1;
                state <= S_STATIC_INIT;
            end
            5'b00111: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <=
                        DYNAMIC_SPAD_REF_EN_START_OFFSET;
                    data_out_reg <= 8'h00;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01000: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <=
                        DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD;
                    data_out_reg <= 8'h2C;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01001: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <= 8'hFF;
                    data_out_reg <= 8'h00;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01010: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
```

```
                end
                else begin
                    write_start_reg <= 1'b1;
                    fnc_sel_reg <= 2'b10; //write
                    reg_address_out_reg <=
                        GLOBAL_CONFIG_REF_EN_START_SELECT;
                    data_out_reg <= 8'hB4;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01011: begin
                if(!write_done) begin
                    instruction_count <= instruction_count;
                    write_start_reg <= 1'b0;
                end
                else begin
                    //read SPAD_TYPE_IS_APERTURE from RAM
                    //when mem_done goes high, data will be available
                        on
                    //mem_data_in line.
                    mem_rw_reg <= 1'b0;
                    mem_start_reg <= 1'b1;
                    mem_addr_reg <= SPAD_TYPE_IS_APERTURE;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01100: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
                    tmp0 <= mem_data_in[0] ? 8'd12 : 8'd0; //first
                        spad to enable
                    tmp1 <= 8'd0; //spads_enabled
                    count <= 8'h00;

                    //read SPAD_COUNT from RAM
                    mem_rw_reg <= 1'b0;
                    mem_start_reg <= 1'b1;
                    mem_addr_reg <= SPAD_COUNT;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01101: begin
                if(!mem_done) begin
                    mem_start_reg <= 1'b0;
                    instruction_count <= instruction_count;
                end
                else begin
```

```
                    tmp2 <= mem_data_in; //spad_count
                    instruction_count <= instruction_count + 1;

                    //dummy memory read to kick off next part
                    mem_addr_reg <= 8'h00;
                    mem_rw_reg <= 1'b0;
                    mem_start_reg <= 1'b1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01110: begin
                if(!mem_done) begin
                    instruction_count <= instruction_count;
                    mem_start_reg <= 1'b0;
                end
                else begin
                    mem_addr_reg <= REF_SPAD_MAP + (count >> 3);
                    mem_rw_reg <= 1'b0;
                    mem_start_reg <= 1'b1;

                    instruction_count <= instruction_count + 1;
                end
                state <= S_STATIC_INIT;
            end
            5'b01111: begin
                if(!mem_done) begin
                    instruction_count <= instruction_count;
                    mem_start_reg <= 1'b0;
                end
                else begin
                    if(count < 8'd48) begin
                        if(count < tmp0 || tmp1 == tmp2) begin
                            mem_rw_reg <= 1'b1;
                            mem_data_out_reg <= mem_data_in & ~(1 <<
                                (count % 8));
                            mem_start_reg <= 1'b1;
                        end
                        else if((mem_data_in >> (count % 8)) & 8'h01)
                            begin
                            tmp1 <= tmp1 + 1;

                            //dummy memory read
                            mem_rw_reg <= 1'b0;
                            mem_start_reg <= 1'b1;
                        end
                        else begin
                            //dummy memory read
                            mem_rw_reg <= 1'b0;
                            mem_start_reg <= 1'b1;
                        end
                        count <= count + 1;
                        instruction_count <= instruction_count - 1;
                    end
                    else begin
                        count <= 8'h00;
                        instruction_count <= instruction_count + 1;
```

```
            end
        end
        state <= S_STATIC_INIT;
    end
    5'b10000: begin
        //write out modified ref_spad_map
        mem_addr_reg <= REF_SPAD_MAP;
        mem_rw_reg <= 1'b0;
        mem_start_reg <= 1'b1;

        instruction_count <= instruction_count + 1;
        state <= S_STATIC_INIT;
    end
    5'b10001: begin
        if(!mem_done) begin
            instruction_count <= instruction_count;
            mem_start_reg <= 1'b0;
        end
        else begin
            //write multiple bytes to write_multi FIFO
            fifo_wr_en_reg <= 1'b1;
            fifo_data_out_reg <= mem_data_in;
            instruction_count <= instruction_count + 1;
        end
        state <= S_STATIC_INIT;
    end
    5'b10010: begin
        if(!fifo_write_ack) begin
            instruction_count <= instruction_count;
            fifo_wr_en_reg <= 1'b0;
        end
        else begin
            if(mem_addr_reg < REF_SPAD_MAP + 5) begin
                mem_addr_reg <= mem_addr_reg + 1;
                mem_rw_reg <= 1'b0;
                mem_start_reg <= 1'b1;

                instruction_count <= instruction_count - 1;
            end
            else begin
                instruction_count <= instruction_count + 1;
            end
        end
        state <= S_STATIC_INIT;
    end
    5'b10011: begin
        //setup register write_multi
        write_multi_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b11; //sel write_multi
        n_bytes_reg <= 4'b0110;
        reg_address_out_reg <=
            GLOBAL_CONFIG_SPAD_ENABLES_REF_0;

        instruction_count <= instruction_count + 1;
        state <= S_STATIC_INIT;
    end
```

```
////////////////////////////////////////////////////////
    //////
// LOAD DEFAULT TUNING SETTINGS FROM INIT ROM
////////////////////////////////////////////////////////
    //////
5'b10100: begin
    if(!write_multi_done) begin
        write_multi_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        write_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b10; //write
        reg_address_out_reg <= rom_data[15:8];
        data_out_reg <= rom_data[7:0];
        rom_addr_reg <= rom_addr_reg + 1;
        instruction_count <= instruction_count + 1;
    end
    state <= S_STATIC_INIT;
end
5'b10101: begin
    if(!write_done) begin
        write_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        write_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b10; //write
        reg_address_out_reg <= rom_data[15:8];
        data_out_reg <= rom_data[7:0];

        if(rom_addr_reg < 86) instruction_count <=
            instruction_count;
        else instruction_count <= instruction_count + 1;

        rom_addr_reg <= rom_addr_reg + 1;
    end
    state <= S_STATIC_INIT;
end
////////////////////////////////////////////////////////
    //////
// Set interrupt config to new sample ready
////////////////////////////////////////////////////////
    //////
5'b10110: begin
    if(!write_done) begin
        write_start_reg <= 1'b0;
        instruction_count <= instruction_count;
    end
    else begin
        write_start_reg <= 1'b1;
        fnc_sel_reg <= 2'b10; //write
        reg_address_out_reg <=
            SYSTEM_INTERRUPT_CONFIG_GPIO;
        data_out_reg <= 8'h04;
```

```
                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        5'b10111: begin
            if(!write_done) begin
                write_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                read_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b01; //sel read
                n_bytes_reg <= 4'b0001;
                reg_address_out_reg <= GPIO_HV_MUX_ACTIVE_HIGH;

                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        5'b11000: begin
            //read data out of FIFO
            if(!read_done) begin
                instruction_count <= instruction_count;
                read_start_reg <= 1'b0;
            end
            else begin
                fifo_read_en_reg <= 1'b1;
                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        5'b11001: begin
            if(!fifo_read_valid) begin
                instruction_count <= instruction_count;
                fifo_read_en_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= GPIO_HV_MUX_ACTIVE_HIGH;
                data_out_reg <= fifo_data_in & ~8'h10; //set LSB

                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        5'b11010: begin
            if(!write_done) begin
                write_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= SYSTEM_INTERRUPT_CLEAR;
                data_out_reg <= 8'h01;
```

```
                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        ////////////////////////////////////////////////////////
            //////
        // getMeasurementTimingBudget
        ////////////////////////////////////////////////////////
            //////
        5'b11011: begin
            if(!write_done) begin
                write_start_reg <= 1'b0;
                instruction_count <= instruction_count;
            end
            else begin
                //shift FSM Control to getMeasurementTimingBudget
                timing_start <= 1'b1;

                write_start_reg <= write_start_timing;
                write_multi_start_reg <=
                    write_multi_start_timing;
                read_start_reg <= read_start_timing;

                timer_start_reg <= timer_start_timing;
                timer_param_reg <= timer_param_timing;
                timer_reset_reg <= timer_reset_timing;

                reg_address_out_reg <= reg_address_out_timing;
                data_out_reg <= data_out_timing;
                n_bytes_reg <= n_bytes_timing;

                fnc_sel_reg <= fnc_sel_timing;

                fifo_data_out_reg <= fifo_data_out_timing;
                fifo_wr_en_reg <= fifo_wr_en_timing;
                fifo_ext_reset_reg <= fifo_ext_reset_timing;
                fifo_read_en_reg <= fifo_read_en_timing;

                mem_addr_reg <= mem_addr_timing;
                mem_data_out_reg <= mem_data_out_timing;
                mem_start_reg <= mem_start_timing;
                mem_rw_reg <= mem_rw_timing;

                init_error_reg <= timing_error;

                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
        5'b11100: begin
            if(!timing_done) begin
                //keep FSM Control under
                    getMeasurementTimingBudget until done
                timing_start <= 1'b0;
```

```
                write_start_reg <= write_start_timing;
                write_multi_start_reg <=
                    write_multi_start_timing;
                read_start_reg <= read_start_timing;

                timer_start_reg <= timer_start_timing;
                timer_param_reg <= timer_param_timing;
                timer_reset_reg <= timer_reset_timing;

                reg_address_out_reg <= reg_address_out_timing;
                data_out_reg <= data_out_timing;
                n_bytes_reg <= n_bytes_timing;

                fnc_sel_reg <= fnc_sel_timing;

                fifo_data_out_reg <= fifo_data_out_timing;
                fifo_wr_en_reg <= fifo_wr_en_timing;
                fifo_ext_reset_reg <= fifo_ext_reset_timing;
                fifo_read_en_reg <= fifo_read_en_timing;

                mem_addr_reg <= mem_addr_timing;
                mem_data_out_reg <= mem_data_out_timing;
                mem_start_reg <= mem_start_timing;
                mem_rw_reg <= mem_rw_timing;

                init_error_reg <= timing_error;

                instruction_count <= instruction_count;
                state <= S_STATIC_INIT;
            end
            else begin
                state <= S_STATIC_INIT;

                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h00;
                data_out_reg <= timing_budget[7:0];

                instruction_count <= instruction_count + 1;
            end
        end
        5'b11101: begin
            if(!write_done) begin
                instruction_count <= instruction_count;
                write_start_reg <= 1'b0;
            end
            else begin
                write_start_reg <= 1'b1;
                fnc_sel_reg <= 2'b10; //write
                reg_address_out_reg <= 8'h00;
                data_out_reg <= timing_budget[15:8];

                instruction_count <= instruction_count + 1;
            end
            state <= S_STATIC_INIT;
        end
```

```
                    5'b11110: begin
                        if(!write_done) begin
                            instruction_count <= instruction_count;
                            write_start_reg <= 1'b0;
                        end
                        else begin
                            write_start_reg <= 1'b1;
                            fnc_sel_reg <= 2'b10; //write
                            reg_address_out_reg <= 8'h00;
                            data_out_reg <= timing_budget[23:16];

                            instruction_count <= instruction_count + 1;
                        end
                        state <= S_STATIC_INIT;
                    end
                    5'b11111: begin
                        if(!write_done) begin
                            instruction_count <= instruction_count;
                            write_start_reg <= 1'b0;
                            state <= S_STATIC_INIT;
                        end
                        else begin
                            write_start_reg <= 1'b1;
                            fnc_sel_reg <= 2'b10; //write
                            reg_address_out_reg <= 8'h00;
                            data_out_reg <= timing_budget[31:24];

                            instruction_count <= instruction_count + 1;
                            state <= S_RESET;
                        end
                        //state <= S_STATIC_INIT;
                    end
                    default: state <= S_RESET;
                endcase
            end
            S_PERFORM_REF_CALIBRATION: begin
            end
        endcase
    end
end

//assign registers to outputs
assign done = done_reg;

assign timer_start = timer_start_reg;
assign timer_param = timer_param_reg;
assign timer_reset = timer_reset_reg;

assign write_start = write_start_reg;
assign write_multi_start = write_multi_start_reg;
assign read_start = read_start_reg;

assign data_out = data_out_reg;
assign n_bytes = n_bytes_reg;
assign reg_address_out = reg_address_out_reg;
```

```
    assign fnc_sel = fnc_sel_reg;

    assign fifo_read_en = fifo_read_en_reg;
    assign fifo_data_out = fifo_data_out_reg;
    assign fifo_wr_en = fifo_wr_en_reg;
    assign fifo_ext_reset = fifo_ext_reset_reg;

    assign mem_addr = mem_addr_reg;
    assign mem_data_out = mem_data_out_reg;
    assign mem_start = mem_start_reg;
    assign mem_rw = mem_rw_reg;

    assign init_error = init_error_reg;
    assign instruction_count_debug = instruction_count;
    assign instruction_count_timeout_debug = instruction_count_timeout;
    assign timing_start_out = timing_start;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
    /
// Company:
// Engineer:
//
// Create Date:    17:50:13 11/12/2017
// Design Name:
// Module Name:    i2c_write_reg_multi
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
    /
module i2c_write_reg_multi(
   //data inputs
     input [6:0] dev_address,
     input [7:0] reg_address,

    //FSM inputs for module
     input clk,
     input reset,
     input start,
     output done,
     input [3:0] byte_width,

     //timer inputs and outputs
     input timer_exp,
     output timer_start,
     output [3:0] timer_param,
     output timer_reset,

     //communication bus with I2C master module
     //combined with read module, all I2C master inputs should
     //be well defined
     input i2c_data_out_ready,
     input i2c_cmd_ready,
     input i2c_bus_busy,
     input i2c_bus_control,
     input i2c_bus_active,
     input i2c_missed_ack,

     output [7:0] i2c_data_out,
     output [6:0] i2c_dev_address,

     output i2c_cmd_start,
     output i2c_cmd_write_multiple,
     output i2c_cmd_stop,
```

```
 output i2c_cmd_valid,
 output i2c_data_out_valid,
 output i2c_data_out_last,
 output [3:0] state_out,

 //FIFO input
 input [7:0] data,
 input fifo_wr_en,
 input fifo_ext_reset,
 output fifo_full,
 output fifo_write_ack,
 output fifo_overflow,

 //status
 output message_failure
);
//write_reg_i2c acts as a module which will, given that the I2C bus is
    available,
//upon start, will take the data available at reg_address and data and upon
//response from an i2C master send the register address and the corresponding
   data
//in the appropriate manner to write to the given register.

//define state parameters
parameter S_RESET = 4'b0000;
parameter S_VALIDATE_BUS = 4'b0001;
parameter S_VALIDATE_TIMEOUT = 4'b0010;
parameter S_WRITE_REG_ADDRESS_0 = 4'b0011;
parameter S_WRITE_REG_ADDRESS_1 = 4'b0100;
parameter S_WRITE_REG_ADDRESS_TIMEOUT = 4'b0101;
parameter S_WRITE_DATA_0 = 4'b0110;
parameter S_WRITE_DATA_1 = 4'b0111;
parameter S_WRITE_DATA_1_LAST = 4'b1000;
parameter S_WRITE_DATA_2 = 4'b1001;
parameter S_WRITE_DATA_TIMEOUT = 4'b1010;
parameter S_WRITE_DATA_TIMEOUT_LAST = 4'b1011;
parameter S_FIFO_READ_VALID_TIMEOUT_0 = 4'b1100;
parameter S_FIFO_READ_VALID_TIMEOUT_1 = 4'b1101;
parameter S_CHECK_I2C_FREE = 4'b1110;
parameter S_CHECK_I2C_FREE_TIMEOUT = 4'b1111;

//define state registers and counters
reg [3:0] state = 4'b0000;
reg [3:0] data_read_count = 4'b0000;

//define output registers -- outputs that are shared with
//other I2C communication modules should be tristated as to prevent
//bus contention -- looking to the future if I have a read/write module
//for 16/32 bit I2C register writes, then any potentially shared connection
//should be tristated.
reg done_reg = 1'b0;
reg timer_start_reg = 1'b0;
reg [3:0] timer_param_reg = 4'b0001;
reg timer_reset_reg = 1'b1;

reg [7:0] i2c_data_out_reg = 8'h00;
```

```
reg [6:0] i2c_dev_address_reg = 7'b0000000;

reg i2c_cmd_start_reg = 1'b0;
reg i2c_cmd_write_multiple_reg = 1'b0;
reg i2c_cmd_stop_reg = 1'b0;
reg i2c_cmd_valid_reg = 1'b0;
reg i2c_data_out_valid_reg = 1'b0;
reg i2c_data_out_last_reg = 1'b0;

reg message_failure_reg = 1'b0;

//define combinational logic
wire bus_valid;
assign bus_valid = ~i2c_bus_busy & ~i2c_bus_active;
wire i2c_bus_free = ~i2c_bus_busy & ~i2c_bus_control;

//define I2C read FIFO
wire fifo_reset;
wire fifo_empty;
wire fifo_read_valid;
wire fifo_underflow;
wire [7:0] fifo_data_out;

reg fifo_reset_reg = 1'b1;
reg fifo_read_en_reg = 1'b0;
wire [3:0] data_count;

FIFO fifo_1(
    .din(data),
    .dout(fifo_data_out),
    .wr_en(fifo_wr_en),
    .rd_en(fifo_read_en_reg),
    .clk(clk),
    .rst(fifo_reset),
    .full(fifo_full),
    .empty(fifo_empty),
    .valid(fifo_read_valid),
    .wr_ack(fifo_write_ack),
    .overflow(fifo_overflow),
    .underflow(fifo_underflow),
    .data_count(data_count)
);

assign fifo_reset = fifo_reset_reg | fifo_ext_reset;

//define state transition diagram
//and state outputs
always @(posedge clk) begin
    if(reset) state <= S_RESET;
    else if(i2c_missed_ack) begin
        state <= S_RESET;
        message_failure_reg <= 1'b1; //missed_ack --> pulse message_failure
    end
    else begin
        case(state)
            S_RESET: begin
```

```
            if(start) begin
                state <= S_VALIDATE_BUS;
            end
            else begin
                state <= S_RESET;
            end

            //reset values
            done_reg <= 1'b0;
            timer_start_reg <= 1'b0;
            timer_param_reg <= 4'b0001;
            timer_reset_reg <= 1'b1;
            data_read_count <= byte_width;

            fifo_reset_reg <= 1'b0;
            fifo_read_en_reg <= 1'b0;

            i2c_data_out_reg <= 8'h00;
            i2c_dev_address_reg <= 7'b0000000;

            i2c_cmd_start_reg <= 1'b0;
            i2c_cmd_write_multiple_reg <= 1'b0;
            i2c_cmd_stop_reg <= 1'b0;
            i2c_cmd_valid_reg <= 1'b0;
            i2c_data_out_valid_reg <= 1'b0;
            i2c_data_out_last_reg <= 1'b0;

            message_failure_reg <= 1'b0;
        end
        S_VALIDATE_BUS: begin
            if(bus_valid) begin
                state <= S_WRITE_REG_ADDRESS_0;
            end
            else begin
                state <= S_VALIDATE_TIMEOUT;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
        end
        S_VALIDATE_TIMEOUT: begin
            if(timer_exp) begin
                state <= S_RESET;
            end
            else if(bus_valid) begin
                state <= S_WRITE_REG_ADDRESS_0;
            end
            else begin
                state <= S_VALIDATE_TIMEOUT;
            end
            timer_start_reg <= 1'b0;
            timer_reset_reg <= 1'b0;
            timer_param_reg <= 3'b001;
        end
        S_WRITE_REG_ADDRESS_0: begin
            if(i2c_data_out_ready) begin
                state <= S_WRITE_REG_ADDRESS_1;
```

```
        end
        else begin
            state <= S_WRITE_REG_ADDRESS_TIMEOUT;
            timer_start_reg <= 1'b1;
            timer_reset_reg <= 1'b1;
        end
        //This state is designed to validate whether or not
        //the I2C master is ready to accept data, so we need
        //to tell the master we're getting ready to write.
        i2c_data_out_reg <= reg_address;
        i2c_dev_address_reg <= dev_address;
        i2c_cmd_start_reg <= 1'b1;
        i2c_cmd_write_multiple_reg <= 1'b1;
        i2c_cmd_stop_reg <= 1'b1;
        i2c_cmd_valid_reg <= 1'b1;
        i2c_data_out_valid_reg <= 1'b1;
        i2c_data_out_last_reg <= 1'b0;
    end
    S_WRITE_REG_ADDRESS_1: begin
        state <= S_WRITE_DATA_0;
        i2c_data_out_valid_reg <= 1'b0;
    end
    S_WRITE_REG_ADDRESS_TIMEOUT: begin
        if(timer_exp) begin
            state <= S_RESET;
        end
        else if(i2c_data_out_ready) begin
            state <= S_WRITE_REG_ADDRESS_1;
        end
        else begin
            state <= S_WRITE_REG_ADDRESS_TIMEOUT;
        end
        timer_start_reg <= 1'b0;
        timer_reset_reg <= 1'b0;
        timer_param_reg <= 3'b001;
    end
    S_WRITE_DATA_0: begin
        if(fifo_underflow) begin
            state <= S_RESET;
            message_failure_reg <= 1'b1;
        end
        else begin
            if(data_read_count > 4'b0001) begin
                state <= S_FIFO_READ_VALID_TIMEOUT_0;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            else begin
                state <= S_FIFO_READ_VALID_TIMEOUT_1;
                timer_start_reg <= 1'b1;
                timer_reset_reg <= 1'b1;
            end
            data_read_count <= data_read_count - 1;
            fifo_read_en_reg <= 1'b1;
            i2c_data_out_valid_reg <= 1'b0;
        end
```

```
end
S_WRITE_DATA_1: begin
    if(i2c_data_out_ready) begin
        state <= S_WRITE_DATA_0;
    end
    else begin
        state <= S_WRITE_DATA_TIMEOUT;
        timer_start_reg <= 1'b1;
        timer_reset_reg <= 1'b1;
    end
    i2c_data_out_valid_reg <= 1'b1;
    i2c_data_out_reg <= fifo_data_out;
end
S_WRITE_DATA_1_LAST: begin
    if(i2c_data_out_ready) begin
        state <= S_WRITE_DATA_2;
    end
    else begin
        state <= S_WRITE_DATA_TIMEOUT_LAST;
        timer_start_reg <= 1'b1;
        timer_reset_reg <= 1'b1;
    end
    i2c_data_out_valid_reg <= 1'b1;
    i2c_data_out_reg <= fifo_data_out;
    i2c_data_out_last_reg <= 1'b1;
end
S_WRITE_DATA_2: begin
    state <= S_CHECK_I2C_FREE;
    i2c_data_out_valid_reg <= 1'b0;
end
S_WRITE_DATA_TIMEOUT: begin
    if(timer_exp) begin
        state <= S_RESET;
        message_failure_reg <= 1'b1;
    end
    else if(i2c_data_out_ready) begin
        state <= S_WRITE_DATA_0;
    end
    else begin
        state <= S_WRITE_DATA_TIMEOUT;
    end
    timer_start_reg <= 1'b0;
    timer_reset_reg <= 1'b0;
    timer_param_reg <= 3'b001;
end
S_WRITE_DATA_TIMEOUT_LAST: begin
    if(timer_exp) begin
        state <= S_RESET;
        message_failure_reg <= 1'b1;
    end
    else if(i2c_data_out_ready) begin
        state <= S_WRITE_DATA_2;
    end
    else begin
        state <= S_WRITE_DATA_TIMEOUT_LAST;
    end
```

```
                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;
                end
                S_FIFO_READ_VALID_TIMEOUT_0: begin
                    if(timer_exp) begin
                        state <= S_RESET;
                        message_failure_reg <= 1'b1;
                    end
                    else if(fifo_read_valid) begin
                        state <= S_WRITE_DATA_1;
                    end
                    else begin
                        state <= S_FIFO_READ_VALID_TIMEOUT_0;
                    end
                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;

                    fifo_read_en_reg <= 1'b0;
                end
                S_FIFO_READ_VALID_TIMEOUT_1: begin
                    if(timer_exp) begin
                        state <= S_RESET;
                        message_failure_reg <= 1'b1;
                    end
                    else if(fifo_read_valid) begin
                        state <= S_WRITE_DATA_1_LAST;
                    end
                    else begin
                        state <= S_FIFO_READ_VALID_TIMEOUT_1;
                    end
                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;

                    fifo_read_en_reg <= 1'b0;
                end
                S_CHECK_I2C_FREE: begin
                    if(i2c_bus_free) begin
                        state <= S_RESET;
                        done_reg <= 1'b1;
                        i2c_cmd_valid_reg <= 1'b0;
                    end
                    else begin
                        state <= S_CHECK_I2C_FREE_TIMEOUT;
                        timer_start_reg <= 1'b1;
                        timer_reset_reg <= 1'b1;
                    end
                end
                S_CHECK_I2C_FREE_TIMEOUT: begin
                    if(timer_exp) begin
                        state <= S_RESET;
                        message_failure_reg <= 1'b1;
                    end
                    else if(i2c_bus_free) begin
```

```
                        state <= S_RESET;
                        done_reg <= 1'b1;
                    end
                    else begin
                        state <= S_CHECK_I2C_FREE_TIMEOUT;
                    end
                    fifo_reset_reg <= 1'b1;

                    i2c_data_out_valid_reg <= 1'b0;
                    i2c_cmd_valid_reg <= 1'b0;

                    timer_start_reg <= 1'b0;
                    timer_reset_reg <= 1'b0;
                    timer_param_reg <= 3'b001;
                end
                default: state <= S_RESET;
            endcase
        end
    end

    //assign registers to outputs
    assign done = done_reg;
    assign timer_start = timer_start_reg;
    assign timer_param = timer_param_reg;
    assign timer_reset = timer_reset_reg;

    assign i2c_data_out = i2c_data_out_reg;
    assign i2c_dev_address = i2c_dev_address_reg;

    assign i2c_cmd_start = i2c_cmd_start_reg;
    assign i2c_cmd_write_multiple = i2c_cmd_write_multiple_reg;
    assign i2c_cmd_stop = i2c_cmd_stop_reg;
    assign i2c_cmd_valid = i2c_cmd_valid_reg;
    assign i2c_data_out_valid = i2c_data_out_valid_reg;
    assign i2c_data_out_last = i2c_data_out_last_reg;

    assign message_failure = message_failure_reg;

    //debugging
    assign state_out = state;

endmodule
```