Modular Capacitive Touchpads

Daniel Sheen
Aaron Pfitzenmaier
12/14/17

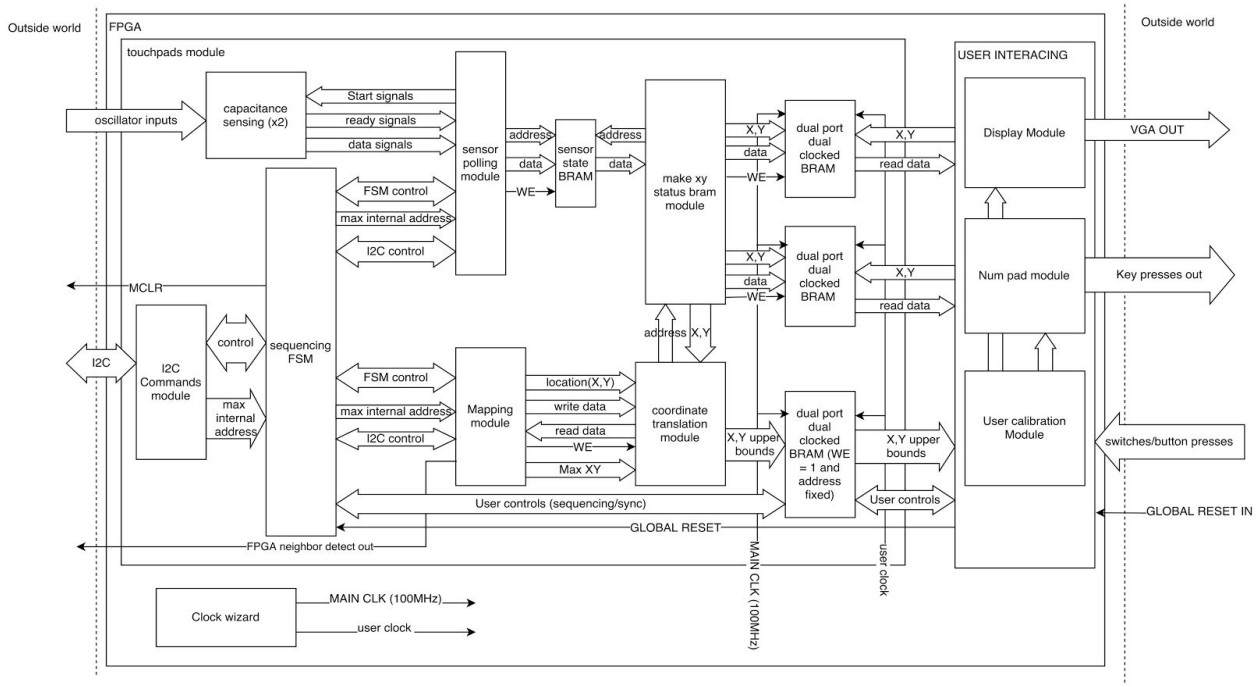
1 Project Overview

For our final project our goal was to create a modular capacitive touch sensor system that could be interfaced with the Nexys 4 as a versatile tool for creating general purpose user interfaces for projects and tinkering. Ideally, the sensor system could be used for custom control systems, for instance, for embedded touch sensing in work surfaces, or for basic projects without requiring complicated or expensive capacitive touch sensor hardware. Essentially, we were hoping to create a nice tool for FPGA based tinkering projects going forward, since it could provide a simple modular platform for creating and using relatively arbitrary size and shape arrays of sensors for user interfaces.

The system consists of square blocks of 16 sensors each which each have one port on each edge that can interface to other sensors or to the FPGA. Each block contains a PIC microcontroller programmed with a unique I2C address that can communicate with the FPGA. Our goal was for the FPGA to be able to automatically scan and identify all blocks connected in an array, and then determine the position of each individual sensor, allowing us to snap together arbitrary configurations of the touch sensor blocks and read the sensors directly by their XY coordinates. This would allow a user of the touch sensors to be able to quickly and easily throw together an interface without worrying overly much about the physical hardware.

We were partially successful in achieving our goal. The FPGA does successfully scan and identify connected sensor blocks as well as correctly sensing pad capacitance to determine if a touch has occurred, however, we suffered setbacks involving touchpad block assembly which delayed the mapping module testing and prevented us from fully achieving our goals.

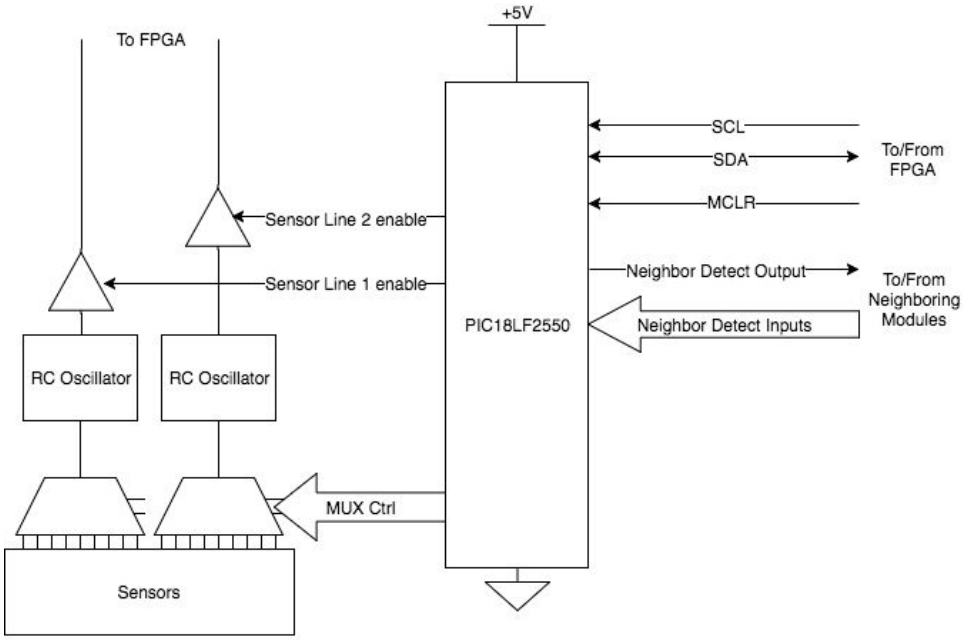
2 FPGA Block diagram



Shown above is a simplified block diagram of the touchpads module and associated user logic modules implemented on the FPGA.

3 Physical Hardware (Aaron and Daniel)

3.1 Physical Hardware Overview



Each block has 4 connectors on each edge. These connectors are used to connect adjacent blocks as well as to the FPGA. Each connector has the following I/O pins:

- SDA/SCL for I2C communication with the FPGA
- 2 sensor data lines
- MCLR (active low PIC reset)
- Neighbor detect pins

The sensor data, MCLR, and I2C communication pins on one edge connector are electrically connected to their corresponding pins on the other 3 edge connectors so that the entire system will all share the same sensor data lines, MCLR, SDA, and SCL. In order to prevent contentions on the sensor data lines, tristate buffers have benn placed between the sensor outputs on each block and the sensor data lines themselves. Each block is controlled by a PIC18LF2550 controller which communicates with the FPGA over I2C and handles switching sensors to and from the data lines in response to commands from the FPGA. This PIC is clocked by the 8MHz internal oscillator and instructions take 0.5us to execute. (one instruction cycle = 4 clock cycles)

The sensors themselves are RC square wave oscillators with a no touch frequency of approximately 100kHz. When the user touches one of the touchpads, their parasitic capacitance to the circuit is added in parallel with the capacitor inside the RC oscillator, causing the output frequency to decrease. This change in frequency is measured by the capacitance sensing module on the FPGA described below. Since each block has two sensor data lines, there are 2 RC oscillators per block, each of which can be connected to 8 of the 16 sensors on a block. 8 input analog multiplexers control which sensors are connected to the oscillators and tristate buffers are used to switch the oscillators on each block to and from the sense lines.

The neighbor detect pins are used by the mapping module on the FPGA to determine the layout in which all of the blocks are connected. The neighbor detect output pin on the PIC connects to each of the 4 edge connectors on a block. There is one neighbor detect input coming from each side of the block, making 4 neighbor detect inputs in total.

The external components (all of which are relatively inexpensive) needed to make a single block are (excluding passive components such as resistors and capacitors, which are available in lab):

- 2 Analog Multiplexers
- 1 18LF2550 PIC controller
- 2 tristate buffers
- 2 comparators (used in the RC oscillator)
- 2 tristate buffers
- 16 metallic touchpads (which we manufactured using the EDS PCB mill)
- 4 DB-9 connectors

3.2 PIC Controller Pinout and Command Set

PIC I/O Pinout:

- Pin RA0 = Mux 1 select bit 0
- Pin RA1 = Mux 1 select bit 1
- Pin RA2 = Mux 1 select bit 2
- Pin RA3 = Mux 2 select bit 0
- Pin RA4 = Mux 2 select bit 1
- Pin RA5 = Mux 2 select bit 2

- Pin RC0 = Tristate 1 Enable
- Pin RC1 = Tristate 2 Enable 2
- Pin RC2 = Neighbor Detect Out

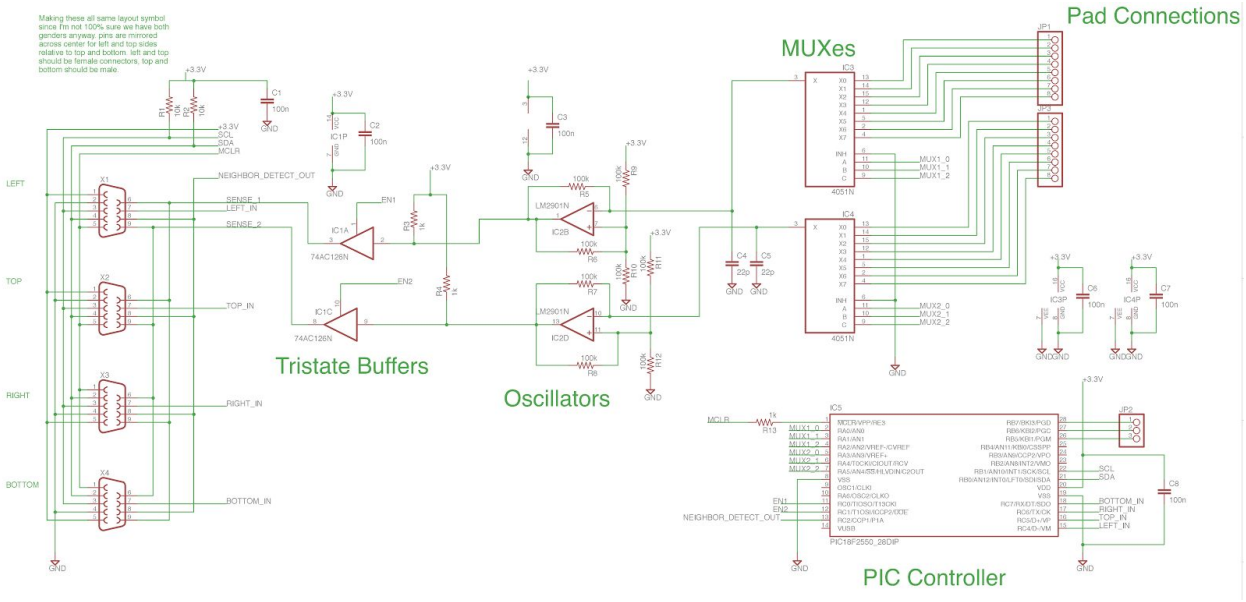
- Pin RC4 = Neighbor Detect In 0
- Pin RC5 = Neighbor Detect In 1
- Pin RC6 = Neighbor Detect In 2
- Pin RC7 = Neighbor Detect In 3

PIC I2C Command Set

- Set Multiplexers -- 11(Mux2[2:0])(Mux1[2:0])
- Set Tristate Enables -- 000000(en2)(en1)
- Neighbor Detect Out Off -- 01000000

- Neighbor Detect Out On -- 01100000
- Response to Read -- (Neighbor Detect In[3:0])0000

3.3 Circuit Diagram



4 Verilog Module details

4.1 Touchpads module (Daniel and Aaron) (top level IP block)

4.1.1 Summary

The touchpads module is the top level block which implements all of the necessary functionality to support an array of capacitive touch sensors and return data for User IO functions. All the logic for communications to the touch sensor blocks including implementation of the I2C interface tristate pins as well as control interfaces for the PICs are handled internal to the module and simply need to be wired to external pins of the FPGA when the module is instantiated. The module also uses two separate clock domains to make user interfacing simpler. The main clock for the module should be 100MHz, and all the internal logic assumes this is the

case. A second user clock can be run at lower frequencies for convenient interfacing to other IO logic (ie. VGA modules).

4.1.2 User Parameters

The module implements the internal address length as a user parameter (N) which can be between 1 and 7. This sets the maximum number of sensor blocks that can be attached to the I2C bus as 2^N (the limit of 7 is set by the fact that I2C uses 7 bit addresses). The user can also set the I2C clock speed (the default is 400kHz) and a parameter called XY quadrant, which places the origin of the XY coordinates used by the sensors at an appropriate corner of the array (for instance, if one wants the XY axis to be oriented the same was as on a VGA display XY_QUADRANT should be 4.

The current design of the touchpads module also provides two separately addressable copies of the capacitive sensor data to allow for multiple user IO functions. It might make sense for a future version to have a parameterized number of user readable ports instead.

4.1.3 Sequencing FSM

The sequencing for the all the logic needed to control the touch sensor blocks is managed by a single FSM implemented in the touchpads module. On reset or on a command to rescan the sensor block configuration, the sequencing FSM first initiates a scan of the I2C bus to detect all attached modules by sending the command SCAN_ADDRESSES to the I2C commands module. This triggers the I2C commands module to generate a translation table between the addresses of all blocks attached to the bus and an internal N bit address space.

Once this is completed, the sequencing FSM signals the mapping module to start, and acts as a passthrough for the commands and data transferred between the mapping module ad the I2C commands module. When the mapping module signals done, the sequencing FSM then triggers a poll of the sensors to establish an initial reading, and then waits on the user to issue a command to poll the capacitances again.

When polling is complete, the sequencing FSM stays in a wait state. If the mode input is written high, this triggers it to poll the sensors. It does so by calling the polling module, giving it

control of the I2C commands module, and then waiting for it to finish before writing the new data to the user accessible status brams and returning to the idle state. If the user holds mode high, it allows the polling module to free run, starting a new polling scan as soon as the previous has finished.

4.1.4 Clock domain crossing

To make interfacing simpler, the touch pads module uses two separate clock domains. All internal logic is operated on a fixed 100MHz clock frequency. All interfaces to user logic operate on a separate user clock domain which permits controlling logic to run at a different frequency.

To cross between the two clock domains, we use dual port brams to create a memory mapped interface. Inputs from user logic are written to the bram on the user clock and read on the 100MHz clock. Outputs are written on the internal 100MHz clock and read on the user clock. The internal logic is arranged so that despite differences in the clock frequencies, information should never be lost during the transition from one clock domain to the other. Provided the user clock is reasonably fast (faster than the I2C clock is sufficient and is required to be able to effectively use the sensors module regardless).

4.2 I2C modules and command handling (Daniel)

4.2.1 summary

The I2C Commands module instantiates an open I2C interface IP, and provides handling for the set of commands used by the touchpads core to communicate with the PICs. It also abstracts the I2C bus addresses to a set of internal addresses shared by all modules, and handles the initial bus boundary scan to detect connected sensor blocks.

4.2.2 commands

The I2C module implements the following commands that are called during operation of the capacitive touchpads.

SCAN_ADDRESSES:
Triggers a scan of the I2C bus to detect attached sensor blocks, and generates a lookup table to convert the blocks' addresses to a set of internal addresses. Returns the maximum internal address used. This command is called on a reset, or when new modules are connected and the user logic requests that attached modules be remapped.

NEIGHBOR_DETECT_HIGH and NEIGHBOR_DETECT_LOW:
Send the I2C commands to enable and disable the neighbor detect outputs of the sensor block at a specified internal address. These commands are used during mapping.

SENSE_NEIGHBORS: performs an I2C read of the PIC on the sensor block at the specified internal address. Reads f any edge of the block read is connected to a block asserting its neighbor detect pin and returns this as 8 bits in the format {4'b0000, left, top, right, bottom}.

SET_MUX_OUTPUTS: (coded as 5'b01xxx) used during polling to select which sensor pad on a block is being read. tells the block at a specified internal address to switch its mux control outputs to the value carried in the lower 3 bits of the command.

SET_SENSE_RAILS_ON and SET_TRISTATE_BUFFERS_OFF: Signals the specified sensor block to enable or disable its tristate buffer outputs. This is used during polling to select which block is being read.

4.3 Mapping module and coordinate translation (Daniel)

4.3.1 Summary

The mapping module determines which sensor blocks are next to which, and then use this information to construct a table which stores the internal address and sensor position for a sensor at a given position in (x,y) coordinates. This allows the make_xy_status_bram module to translate sensor readings to an x,y addressed form readable by user logic modules.

4.3.2 Neighbor Detection

When it receives a start signal from the control module, the mapping module executes the following procedure for each sensor block (by repeating for address 0 through the max internal address). Note that we need to handle the FPGA location separately.

1. Tell the sensor block at "address" to write it's neighbor detect output pin high.
2. Request each sensor block in the rest of the address space to return which if any of its neighbor detect inputs are currently high.
3. If one of the polled block's pins is high, store that the sensor at "address" is abutting it on the detected edge, otherwise just move on to the next one.
4. Repeat this procedure for all currently valid addresses.
5. Finally, write the FPGA neighbor detect high, use the same procedure to check which block is connected to the FPGA and which side the FPGA is attached to. This will be saved in a special register as {internal_address[N-1:0], side}. Note that side only needs to be one bit, because since we aren't using hermaphrodite connectors, the FPGA can only connect to one of two sides of the board.

The data is stored in a BRAM with a depth the same as the internal address space. At the location corresponding to each used address, the adjacency information is stored as follows (where each address is N bits and the valid bit is a one bit flag to clarify whether there is in fact a neighbor)

{left_valid, left_address, top_valid, top_address, right_valid, right_address, bottom_valid, bottom_address}

4.3.3 XY position calculation

Once this information is stored, we will construct a second table of signed X,Y coordinates based on the adjacency information. The values for X and Y are initially N+1 bits each to accommodate the sign bit. Data is stored in a BRAM as {valid, rotation[1:0], X, Y}, where valid indicates that the entry is current, and rotated is a 2 bit number that indicates the orientation of the block. From this we then construct the conversion table available to the user IO and display modules. The procedure to generate the addresses to location map is as follows.

1. Start by assigning the block the FPGA is connected to the coordinates (0,0). If the side the FPGA is connected to is the top rotation is 0, if it is attached to the left edge 1, bottom 2, and right 3. Ie. the rotation number maps to clockwise physical rotations as (0 degrees => 0, 90 =>1, 180=>2, and 270 =>3)

2. Read the adjacency data for this block (the base block for this step). In turn, for each neighboring block which doesn't yet have an assigned location, assign a pair of coordinates X,Y based on which edge of the base block it is attached to and the known rotation of the base block. Assign the block a rotation based on which of it's edges is attached to the base block edge and the rotation of the base block. Finally, mark the location data for the neighbor block as valid.

3. Mark the data for this base block as used (stored in the neighbor_data_used register) to prevent us from going back to it.

4. Check if all addresses now have a known location (are marked valid), if not, go to step 5 if they do, we're done with this part.

5. For all addresses less than the max internal address check if there is now a defined location. Repeat step 2 for each address that has a known location and is marked as unused.

4.3.4 XY position storage

The next step is to store the XY position information in a lookup table. The BRAM we actually write to is actually part of the coordinate translation block, so I'll simply refer to it as the coordinate translation BRAM

1. First ind the minimum X and Y values (this should require one more pass over the address space) and save them in another register as Xmin Ymin.

2. Write the valid bits for every location in the coordinate translation BRAM to zero.

8. For each address retrieve {valid, rotation, X, Y}, and store the string {valid, rotated, address} to the location {{X - Xmin}[N-1:0], {Y -Ymin}[N-1:0]} in the coordinate translation BRAM. This makes everything positive and drops the sign bit from the location space.

9. We use a Frame buffer to allow us to update the coordinate translation module outputs all at once, this requires we shift in the new data by asserting a “set” signal which tells the coordinate translation module to start using the new data. Finally, we assert done to tell the touchpads sequencing FSM that the process has finished.

4.3.5 XY coordinate translation

The actual conversion between XY coordinates is performed by the coordinate translation module. Upon a request from the make_y_status_bram module, the coordinate translation takes in a pair of XY coordinates and returns the address corresponding to the pad at those coordinates as used by the polling module. It does this by using the upper part of the coordinates to retrieve the address of the block at that location (previously written by the mapping module). This is returned to the make_y_status_bram module as the upper N bits of the address. The lower 4 bits are generated from a lookup table, which returns the pad on the block corresponding to the lower two bits of X and Y for each of the four possible rotations. The correct pad address is selected based on the stored rotation of the block, and returned as the lower four bits of the address to the make_y_status_bram.

4.4 Polling and Capacitance sensing (Aaron)

4.4.1 Capacitance sensing module

The capacitance sensing module counts the number of rising edges seen on one of the sense rails over a period of time defined as a parameter (default 750us) and outputs it. When a person touches a pad that is being sensed, the added capacitance causes the frequency of its RC oscillator decrease, thus reducing the number of rising edges on the sense rail in a given amount of time. By comparing this number to the threshold determined by the User Calibration Module, one can determine if a pad has been touched.

This module takes a start signal as input from the polling module and begins counting rising edges as soon as the start signal transitions from low to high. Once the sensing time has elapsed, a done output is pulsed for one clock cycle, signaling that the data output is valid.

4.4.2 Polling Module

The polling module cycles through all available touchpads, samples them, and stores the data in a BRAM. This module instantiates two capacitance sensing modules, one for each sense rail. The inputs to this module are

1. A start input.
2. An I2C done input, which goes high once an I2C command has finished executing.
3. An input from the upper level touchpads module telling the polling module what the maximum internal block address is.
4. The two sense rails.

The outputs of this module are

1. An I2C start output telling the I2C module to send a command, an output telling which I2C command we want to send, and an output telling which block to send it to.
2. BRAM write enable, data in, and address outputs used to write to the polling BRAM

The polling process occurs as follows:

1. Send I2C commands to a block to enable its tristate buffers and connect its RC oscillators to the sense rails, and wait for the command to execute.
2. Send an I2C command to set the sensor multiplexer select lines to the new sensor, and wait for the command to execute.
3. Pulse the start input on both capacitance sensing modules, and wait for the modules to asset their done signal.
4. Store the data in a BRAM. The address of a certain touchpad's data has the following format {Internal block address, Sensor number on block}
5. Update the sensor to read. If we've finished scanning an entire block, send and I2C command to disable its tristate buffers and increment the internal address of the block we are currently polling.
6. If we have sampled all touchpads, pulse done for one clock cycle and halt until the start signal is asserted again, otherwise return to step 1.

4.5 XY status translation (Aaron)

The polling module creates a BRAM that maps a block internal address and touchpad number on the block (the touchpad number is a number 0-15 corresponding to the MUX select inputs that select the sensor and which sense rail it is on) to the data read from the touchpad. The mapping module creates a BRAM that maps an xy coordinate of the touchpad in the block configuration to an internal address and touchpad number of the block.

For writing user IO modules, however, it is more useful to have and easier to use a BRAM that maps an xy coordinate of a touchpad in the block configuration directly to the data read from that touchpad. The XY Status Translation module generates this BRAM. It does this by cycling through all possible xy coordinates (the maximum x and y coordinates in the block layout are given as inputs to this module) and reading the mapping BRAM, which stores the address corresponding to the touchpad in the polling BRAM. The polling BRAM is then read to get the touchpad's data. This data is then stored in the new XY Status BRAM at the address {x coordinate, y coordinate}. If there is no touchpad located at the xy coordinate that is currently being looked at, a value of 0 is stored in the XY status BRAM, signaling the coordinate is not a valid sensor location.

This module begins generating the BRAM once the start input is asserted and asserts the done output once the BRAM is fully generated.

4.6 User demo modules (Aaron)

There are two user demo modules in this project. The first is simply a display module, which displays the layout of the blocks and status of the sensors on a 1024x768 VGA display. The size, in pixels, of a single touchpad on the screen and the threshold for determining whether or not a sensor is touched are given as inputs to the display module.

The other user demo module utilizes 10 touchpads and a Teensy microcontroller to create a number pad that can be interfaced to a computer and used to type. This module repeatedly queries 10 touchpads arranged in a number pad and sends their statuses over serial to the Teensy.

The Teensy then parses the serial data it receives and presents itself as a USB Keyboard to the computer using the touchpads as a number pad. Whenever it detects a touchpad has been pressed, it tells the computer to type the corresponding number on screen.

4.6.1 Calibration Module

The calibration module determines the threshold used to determine whether a touchpad is pressed or not by looking at the minimum value of all sensors when the calibration button is pressed. Once this threshold is determined, a sensor is considered on when the sensor value is less than the threshold.

5 Troubles and Lessons Learned

What really kept us from achieving our was that we didn't have multiple modules built until a couple days before the checkoff. We got a single working sensor block breadboarded early on, and then simply sat on it trusting it would work reliably with more of them. We developed the mapping module mostly using verilog test benches for verification and basically trusted that it would work correctly since we were able to verify that communication with the PIC worked the way we expected it to, so the fact that it failed in the end implies that we failed to fully account for the behavior interacting with multiple blocks.

Trying to build more sensor blocks on perfboard also turned out to be a huge mistake since they were sufficiently complex so as to be nearly impossible to troubleshoot when something went wrong with them. This led to us second guessing a lot of our observations when we first put together more blocks, and we wound up spending a lot of time questioning whether the mapping failure was a result of the hardware or the verilog code being faulty which severely hampered troubleshooting.

In hindsight we should have finalized the hardware design much earlier on and ordered PCBs so that we could have physical references to test against early on in development. If we had we would have had plenty of time to troubleshoot and fix the problems with the mapping FSM during design and would have had a fully working system for checkoff.

6 Conclusion

Overall, we were able to use the Nexys 4 to implement functionality for reading capacitive sensors, displaying their status on a VGA display, and interfacing them with a computer by having a group of ten touchpads act as a number pad for typing. This involved constructing an I2C communications protocol between the Nexys4 and PIC microcontrollers on the touch sensor blocks, utilizing multiple BRAMs, using different clock domains, and interfacing with multiple external devices. We were also able to manufacture the 4x4 grids of touchpads using a PCB mill.

Unfortunately, due to unforeseen construction difficulties and sub-optimal use of time on our part, we did not have time to fully debug the part of the mapping module that correctly orients the blocks with respect to each other. Although we can join together multiple blocks and read and display the status of the sensors on all of them, we were unable to correctly display the rotation of the touchpad blocks on the VGA.

We were able to successfully achieve all of our baseline goals, part of our expected goals, and two of our stretch goals. In the future, we plan on fixing the above mentioned issues with rotation determination at which point, the entire system should work as expected.

7 Source Code Files

7.1 PIC Code (Aaron)

- Unless otherwise stated, all instructions written below can be assumed to use the Access RAM as the RAM Bank for reading memory, that the destination for the result of any instruction is the working register WREG, and that all returns do not use fast call/return mode.
- For bit oriented operation, the bit that is being acted on is identified at the end of the instruction in the format ,b=<bit>.
- See PIC18LF2550 datasheet, Section 26 (Instruction Set Summary) for details: http://ww1.microchip.com/downloads/en/devicedoc/39632c.pdf

7.1.1 Programming instructions:

You can use an arduino to program your PIC. See the following link (make sure low voltage programming is enabled to use this): https://sites.google.com/site/thehighspark/arduino-pic18f

If you choose to use this programmer, the code below can be written to the PIC by first sending the erase command 'EX' through the Arduino IDE serial monitor, and then sending the following instructions (creates a PIC with i2c address 85 in decimal):

```
C108XC21EXC31EXC581XW0000EF80F000FFFFFFFF0E08149EE002EC00F0010010FFFFFFFFFFFFFFFFFFFFF
FFFXW01000E4C6ED368936A926A890EF06E946A8B966D866F6A006A016A026A036AC70E36XW01206EC60E8
16EC50EAA6EC80EC06EF2969E869D869F8ED0D7FFFFFFFFFFFFFFFFFFFXW0200CFC9F0009CC69EC60E2414C
76E010E201801E102EC80F0010E041801E102EC00XW0220F002969E88C60012FFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFXW03000E401800E102948B00120E601800E102848B00120EC014006E020EC
01802E108XW03200EC014896E030E3F140010036E8900120EFC1400E1080EFC148B6E030E031400XW03401
0036E8B00120012FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXW04000EF014826EC90012F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFX
```

Use the following to make a PIC with arbitrary I2C address by replacing the ++ with the I2C address byte in hex (bits [7:1] are the address itself, bit 0 is always 0)

```
C108XC21EXC31EXC581XW0000EF80F000FFFFFFFF0E08149EE002EC00F0010010FFFFFFFFFFFFFFFFFFFFF
FFFXW01000E4C6ED368936A926A890EF06E946A8B966D866F6A006A016A026A036AC70E36XW01206EC60E8
16EC50E++6EC80EC06EF2969E869D869F8ED0D7FFFFFFFFFFFFFFFFFFFXW0200CFC9F0009CC69EC60E2414C
76E010E201801E102EC80F0010E041801E102EC00XW0220F002969E88C60012FFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFXW03000E401800E102948B00120E601800E102848B00120EC014006E020EC
01802E108XW03200EC014896E030E3F140010036E8900120EFC1400E1080EFC148B6E030E031400XW03401
0036E8B00120012FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFXW04000EF014826EC90012F
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFX
```

Ensure the following configuration bytes are set for proper operation, the rest should take on their default values:

```
0x300001: 00001000
0x300002: 00011110
0x300003: 00011110
0x300005: 10000001
```

7.1.2 Assembly Code:

```
;;;Beginning of the program, store the line starting at program memory location
0x000000;;;
start:
GOTO initial


;;;Program initialization;;;
initial:
MOVLW 01001100 ;oscillator configuration
MOVWF OSCCON
SETF TRISB ;config sda/scl as inputs
CLRF TRISA ;config portA as outputs
```

```
CLRF LATA
MOVLW 11110000
MOVWF TRISC ;config PORTC [3:0] as outputs, [7:4] as inputs
CLRF LATC

BCF UCON, b=011 ;diasble USB port on RC4 and RC5
BSF UCFG, b=011

;Clear the 4 variables we use in RAM;
CLRF data
CLRF adrw
CLRF cmd_part
CLRF lat_part

;init i2c bus;
CLRF SSPSTAT
MOVLW 00110110
MOVWF SSPCON1
MOVLW 10000001
MOVWF SSPCON2
MOVLW 10101010 ;put the i2c address of your PIC here, this code corresponds to an
address of 85 in decimal (bits [7:1] are the address itself, bit 0 is always 0)
MOVWF SSPADD

;init interrupts;
MOVLW 11000000
MOVWF INTCON
BCF PIR1, b=011
BSF PIE1, b=011
BSF IPR1, b=011
BSF RCON, b=111

BRA -1 ;Halt


;;;Store the following 5 lines starting at the High Priority Interrupt Vector (located
in memory at 0x000008);;;
MOVLW 00001000
ANDWF PIR1 ;not zero if synchronous serial port interrupt flag is set
BZ 2 ;branch if the SSP interrupt flag is not set, we didn't receive an I2C command so
don't parse anything
CALL i2c_interrupt ;CALL instruction take up two bytes of program memory
RETFIE


;;;Handle any I2C command received and call i2c_w or i2c_r depending on whether it was
a write or read command;;;
i2c_prog:
MOVFF SSPBUF,data ;read synchronous serial port buffer and store its contents in data
BCF SSPCON1, b=110 ;clear WCOL and SSPOV bits
BCF SSPCON1, b=111
MOVLW 00100100 ;filter SSPSTAT to get just data/address and read write bits
ANDWF SSPSTAT
MOVWF adrw ;store result in adrw

MOVLW 00100000
XORWF adrw ;0 if data+write
BNZ 2 ;skip if not a write command
CALL i2c_w ;CALL instruction take up two bytes of program memory

MOVLW 00000100
```

```
XORWF adrw ;0 if address+read
BNZ 2 ;skip if not a read command
CALL i2c_r ;CALL instruction take up two bytes of program memory

BCF PIR1 b=011 ;clear Synchronous Serial Port interrupt flag
BSF SSPCON1, b=100 ;release SCL
RETURN


;;;Subroutine to parse I2C reads, reads the 4 neighbor detect inputs;;;
i2c_r:
MOVLW 11110000
ANDWF PORTC ;mask the lower 4 bits of port C since these pins are not used as inputs
MOVWF SSPBUF ;Send the data over I2C
RETURN

;;;Subroutine to parse and execute I2C write commands;;;
i2c_w:

;Neighbor detect off command;
MOVLW 01000000
XORWF data ;zero if neighbor detect off command
BNZ 2 ;skip if not neighbor detect off command
BCF LATC, b=010
RETURN

;Neighbor detect on command;
MOVLW 01100000
XORWF data ;zero if neighbor detect on command
BNZ 2 ;skip if not neighbor detect off command
BSF LATC, b=010
RETURN

;Set MUX select lines command;
MOVLW 11000000
ANDWF data
MOVWF cmd_part ;Store the result of the above operation in cmd_part
MOVLW 11000000
XORWF cmd_part ;zero if mux select command
BNZ 8 ;skip if not mux select command
MOVLW 11000000
ANDWF LATA
MOVWF lat_part ;Store the result of the above operation in lat_part
MOVLW 00111111
ANDWF data
IORWF lat_part
MOVWF LATA
RETURN


;Set Tristate buffer enables command;
MOVLW 11111100
ANDWF data ;zero if tristate command
BNZ 8 ;skip if not tristate command
MOVLW 11111100
ANDWF LATC
MOVLW 00000011
ANDWF data
IORWF lat_part
MOVWR LATC
RETURN
```

RETURN

7.2 Verilog Modules

7.2.1 Touchpads Top Module

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////
//
//   NEW TOP level Block for the Touchpads verilog module.
//
//   started on 12/3/17 based on single_clock design
//      attempt to integrate support for a separate user clock domain
//      we use dual clock brams as a memory map for trasitioning between clock domains. This seems to work quite nicely
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////

module touchpads #(
      parameter    N          = 4,          //number of bits to describe touchpad blocks (sets a maximum of 2^N blocks that the synthesized module can support)
      parameter    I2C_SPEED    = 400000,      //I2C speed assuming a 100MHz clock needs to be set proportionally higher if the main clock is lower
      parameter    XY_QUADRANT = 4            //coordinate system to be used (defines positive XY direction) 4 is appropriate for VGA and places the origin at
                                 //the top left corner
      )
      (

      //global signals
      input wire    clk_100mhz,
      input wire    clk_user,               //this cannot be arbitrarily slow. we assume that the user is running with a clock at least as fast as the I2C,
                                 //otherwise this can never work reliably. handling of user clocked signals assumes it's fast enough that a mode
                                 //pulse doesn't last longer than 10ms (time to scan one block of sensors)

                                 //also MUST be less than or equal to 100MHz
      input wire    reset,

      //control communications for user
      input wire    mode,         //write 1 for free running continuous scan, or pulse this to trigger start of a scan of the pad capacitances
      input wire  rescan_trig,   //pulse this for one clock cycle to request a rescan of the I2C bus
      output wire done_out,          //pulses high for one clock cycle on completion of a full scan. first pulse will be after the boundary scan and figuring things out is
done
      output wire config_in_progress_out,         //hold high while we're figuring out what we're attached to. lets user logic know we aren't ready yet.
                                    //also reassert this if we need to reinitialize the boundary scan

      //user data comms

      output wire [N+1:0]    x_upper_bound_out,      //these tell the user the maximum bounds of the XY coordinates occupied by sensors
      output wire [N+1:0]    y_upper_bound_out,

      input wire    [N+1:0]      x_coordinate,      //write these to query a sensor at a given XY coordinate
      input wire    [N+1:0]      y_coordinate,
      input wire    [N+1:0]      x_coordinate2,      //write these to query a sensor at a given XY coordinate
      input wire    [N+1:0]      y_coordinate2,

      output wire [11:0]       sensor_data_current,   //current and last readings from the sensor at the given XY coordinate. output two clock cycles after the data is
requested
      output wire [11:0]       sensor_data_current2,   //current and last readings from the sensor at the given XY coordinate. output two clock cycles after the data is
requested
      //output wire [11:0]       sensor_data_previous,

      output wire          user_coordinate_valid,   //assert this if sensor is in fact present at requested location and data is ready
      output wire          user_coordinate_valid2,   //assert this if sensor is in fact present at requested location and data is ready

      output [15:0] mapping_status,


       //I2C signals

       inout          scl,   //SCL tristate signal. connect this directly to an external pin, call it an inout, and don't try to do anything else clever with it!
       inout          sda,   //SDA tristate signal. same goes for this.

       output wire    fpga_neighbor_detect,   //pin for figuring out who is next to the FPGA
       output reg       sensor_blocks_reset,   //global clear for the PICs active low


       //capacitance sense signals

       input wire sensor_1,
       input wire sensor_2,

       //troubleshooting access (user probably needn't ever bother instantiating this)

       output wire [4:0]   master_fsm_state, //this is kept on the 100MHz clock because it's never going to be able to sync to the slower one

       output [N-1:0] polling_int_addr,
       output [N-1:0] command_int_addr
      );

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
   //
   //   SUBMODULE INSTANTIATIONS
   //
```

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


   ////////////////////////////
   //   I2C module instantiation
   ////////////////////////////

   reg       i2c_start;   //assert start to initiate a response to inputted command (doesn't effect control signals)
   reg       i2c_stop;   //current implementation doesn't actually do anything with this, will be gotten rid of in syntheseis
   reg       i2c_hold;   //halts control FSM in current state while high, resumes when released.
   wire       i2c_done;   //indicates that the FSM has finished executing a command. held high when inactive so we don't have to remember
   wire [4:0]    i2c_state;   //state output for debus and syncing with system control FSM

   //I2C command handling
   reg    [COMMAND_LEN-1:0]    i2c_command;
   reg      [N-1:0]          i2c_address_int;    //internal address used by other modules
   wire    [7:0]             i2c_return_data;   //multipurpose return data output, different commands may use this differently
   wire                    i2c_bus_error;      ////flag for something went wrong. error code will be contained in return data.

   //intenal address upper bound output after I2C has finished
   wire [N-1:0] max_internal_address;

   //I2C tristate control signals
   wire    scl_i;
    wire    scl_o;
    wire    scl_t;
    wire    sda_i;
    wire    sda_o;
    wire    sda_t;

    //instantiate module
    i2c_commands
          #(
          .N(N),
          .COMMAND_LEN(COMMAND_LEN),
          .I2C_SPEED(I2C_SPEED)
          )
      commands_1(
         .clk_100mhz(clk_100mhz),
         .reset(reset_internal),
         .start(i2c_start),
         .stop(i2c_stop),
         .hold(i2c_hold),
         .done(i2c_done),
         .state(i2c_state),
         .command(i2c_command),
         .address_int(i2c_address_int),
         .return_data(i2c_return_data),
         .bus_error(i2c_bus_error),
         .max_internal_address(max_internal_address),
         .scl_i(scl_i),
         .scl_o(scl_o),
         .scl_t(scl_t),
         .sda_i(sda_i),
         .sda_o(sda_o),
         .sda_t(sda_t)
         );

      assign command_int_addr = i2c_address_int;
   ////////////////////////////////
   //   mapping module instantiation
   ////////////////////////////////

   //signals to/from the control FSM
   reg          mapping_start;
   reg          mapping_stop;
   reg          mapping_hold;
   wire          mapping_done;
   wire    [4:0]    mapping_state;

   //i2c handling IO (processed by control FSM before going to I2C commands module)
   wire                  mapping_i2c_send_req;         //this is one clock pulse, needs to be detected immediately by control FSM, but can take a while to act on it.
   wire    [N-1:0]          mapping_i2c_address_int;
   wire    [COMMAND_LEN-1: 0]   mapping_i2c_command;
   reg                   mapping_i2c_command_processed;   //should be pulsed high for one clock cycle as soon as the I2C module has finished running the command and any
return data is ready


   //communications to the XY coordinant to address conversion memory and logic
   wire         mapping_set_output;      //tells map frame buffer to shoift out newly written data
   wire         mapping_write_enable;   //enable to write to the lookup BRAM
   wire    [N-1:0]   mapping_x_address;      //X position of block to be written
   wire    [N-1:0]   mapping_y_address;      //Y position of block to be written
   wire   [N+2:0]   mapping_data_out;      //data formatted as {valid,rotated,internal_address}
   wire    [N+2:0] mapping_data_in;      //readback to make sure things are actually written correctly

   wire    [N-1:0] mapping_max_x_bound_out;
   wire    [N-1:0] mapping_max_y_bound_out;

   //instantiate mapping module
   xy_mapping #(.N(N), .COMMAND_LEN(COMMAND_LEN), .XY_QUADRANT(XY_QUADRANT))
      mapping_1(
         .clk(clk_100mhz),
         .reset(reset_internal),
         .max_internal_address(max_internal_address),
         .start(mapping_start),
         .stop(mapping_stop),
         .hold(mapping_hold),
         .done(mapping_done),
         .state(mapping_state),
         .i2c_send_req(mapping_i2c_send_req),
         .i2c_address_int(mapping_i2c_address_int),
         .i2c_command(mapping_i2c_command),
         .i2c_command_processed(mapping_i2c_command_processed),
         .i2c_data_in(i2c_return_data),
         .fpga_neighbor_detect_out(fpga_neighbor_detect),
```

```
        .set_output(mapping_set_output),
        .write_enable(mapping_write_enable),
        .x_address(mapping_x_address),
        .y_address(mapping_y_address),
        .data_out(mapping_data_out),
        .data_in(mapping_data_in),
        .max_x_bound_out(mapping_max_x_bound_out),
        .max_y_bound_out(mapping_max_y_bound_out),
        .mapping_status(mapping_status)
        );


    ////////////////////////////////////////////
    //   coordinate translation module instantiation
    ////////////////////////////////////////////////

    wire          coordinate_error;  //we'll assert this if the xy request from user is not within bounds
     wire    [N+3:0] request_data_address;      //address to read sensor data from
     wire    [N+1:0] xRead, yRead;
     wire          coordinate_valid;

     wire [N+1:0]    x_upper_bound;      //these tell the user the maximum bounds of the XY coordinates occupied by sensors
    wire [N+1:0]    y_upper_bound;

    coordinate_translation #(.N(N), .XY_QUADRANT(XY_QUADRANT))
       translation_1(
        .clk(clk_100mhz),
        .reset(reset_internal),
        .max_internal_address(max_internal_address),
        .set_output(mapping_set_output),
        .write_enable(mapping_write_enable),
        .x_address(mapping_x_address),
        .y_address(mapping_y_address),
        .data_in(mapping_data_out),
        .data_out(mapping_data_in),
        .max_x_bound_in(mapping_max_x_bound_out),
        .max_y_bound_in(mapping_max_y_bound_out),
        .x_coordinate(xRead),
        .y_coordinate(yRead),
        .max_x_bound_out(x_upper_bound),
          .max_y_bound_out(y_upper_bound),
        .valid(coordinate_valid),
        .error(coordinate_error),
        .address(request_data_address)
        );

    ////////////////////////////////////////////
    // polling module instantiation
    ////////////////////////////////////////////

    wire    [4:0]             polling_state;
    reg                   polling_start;
    wire                   polling_done;

    wire                   polling_i2c_send_req;         //this is one clock pulse, needs to be detected immediately by control FSM, but can take a while to act on it.
    wire    [N-1:0]          polling_i2c_address_int;
    wire    [COMMAND_LEN-1: 0]   polling_i2c_command;
    reg                   polling_i2c_command_processed;   //should be pulsed high for one clock cycle as soon as the I2C module has finished running the command and any
return data is ready

    wire    [N+3:0]          bram1_address;
    wire                   bram1_we;
    wire    [11:0]             bram1_din;

    polling #(.N(N),
              .COMMAND_LEN(COMMAND_LEN))
            polling_1(.clock(clk_100mhz),
                 .reset(reset_internal),
                 .start(polling_start),
                 .i2c_done(polling_i2c_command_processed),
                 .max_internal_address(max_internal_address),
                 .sensor_in1(sensor1_clean),
                 .sensor_in2(sensor2_clean),
                 .done(polling_done),
                 .i2c_start(polling_i2c_send_req),
                 .i2c_command(polling_i2c_command),
                 .internal_address(polling_i2c_address_int),
                 .bram1_address(bram1_address),
                .bram1_we(bram1_we),
                .bram1_din(bram1_din),
                .state(polling_state)
                );

    assign polling_int_addr = polling_i2c_address_int;


    //polling bram instantiation
    wire [N+3:0] polling_read_addr;
    wire [11:0] polling_data_out;
    //in theory never risks a collision, so trying the rw1 version here to get vivado to behave more nicely
    //otherwise vivado insists on building lutram instead of BRAM, which is silly
    mybram_rw1 #(.LOGSIZE(N+4),
                 .WIDTH(12))
          polling_data(.r_addr(polling_read_addr),
                    .w_addr(bram1_address),
                    .clk(clk_100mhz),
                    .din(bram1_din),
                    .dout(polling_data_out),
                    .we(bram1_we)
                    );

    //assign sensor_data_previous = bram1_din;
    //assign sensor_data_current[3:0] = bram1_address[3:0];
    //assign sensor_data_current[4] = bram1_we;

    ////////////////////////////////////////////
    // xy to status bram module instantiation
    ////////////////////////////////////////////
```

```
   reg xy_status_bram_generation_start;
   wire xy_status_bram_generation_done;
   wire status_we;
   wire [2*N+3:0] status_bram_w_addr;
   wire [11:0] status_data_in;

   wire [2*N+3:0] status_bram_r_addr;
   wire [11:0] status_bram_data_out;

   make_xy_status_bram #(.N(N))

      make_xy_status_bram
      (.clk(clk_100mhz),
      .xy_addr(request_data_address), //address to read sensor data from coordinate translation module
      .valid(coordinate_valid), //is sensor at xy location valid, from coordinate translation module
      .error(coordinate_error), //is the xy location within bounds, from coordinate translation module
      .polling_data(polling_data_out), //polling bram data
      .maxX(x_upper_bound),
      .maxY(y_upper_bound),
      .reset(reset_internal),
      .start(xy_status_bram_generation_start), //start generating the bram, this should be initiated by the control fsm
      //.threshold(threshold),

      .xRead(xRead), //x coordinate of pad to check
      .yRead(yRead), //y coordinate of pad to check
      .polling_addr(polling_read_addr), //polling bram address
      .status_we(status_we), //write enable for the xy status bram
      .status_bram_addr(status_bram_w_addr),
      .status_data(status_data_in), //data to write to xy status bram (00=invalid, 01 = off, 11 = on)
      .done(xy_status_bram_generation_done) //we're done generating the bram
   );


   //use 2 brams with the write inputs tied together
   //one is for the VGA display, the other is for User IO implementations
   wire [11:0] xy_bram_dout;
   wire [11:0] xy_bram_dout2;

   //bram1
   mybram_dualclock #(.LOGSIZE(2*N+4),
               .WIDTH(12))
         xy_status_bram(.r_addr({x_coordinate,y_coordinate}),
                     .w_addr(status_bram_w_addr),
                     .clk_100mhz(clk_100mhz),
                     .clk_user(clk_user),
                     .din(status_data_in),
                     .dout(xy_bram_dout),
                     .we(status_we)
                     );
   assign sensor_data_current = xy_bram_dout;

   assign user_coordinate_valid = |sensor_data_current;

   //wire [11:0] xy_bram_dout;

   //bram2
   mybram_dualclock #(.LOGSIZE(2*N+4),
               .WIDTH(12))
         xy_status_bram2(.r_addr({x_coordinate2,y_coordinate2}),
                     .w_addr(status_bram_w_addr),
                     .clk_100mhz(clk_100mhz),
                     .clk_user(clk_user),
                     .din(status_data_in),
                     .dout(xy_bram_dout2),
                     .we(status_we)
                     );
   assign sensor_data_current2 = xy_bram_dout2;

   assign user_coordinate_valid2 = |sensor_data_current2;




////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
   //
   //   module internal parameters
   //

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/

   // fixed global values
   parameter COMMAND_LEN = 5;

   //I2C command codes
   parameter SCAN_ADDRESSES       = 5'b00000; //this triggers the boundary scan FSM

   parameter NEIGHBOR_DETECT_HIGH    = 5'b00001; //write neighbor detect output high
   parameter NEIGHBOR_DETECT_LOW    = 5'b00010; //write neighbor detect output low
   parameter SENSE_NEIGHBORS       = 5'b00011; //needs to trigger a read and return suitable data

   parameter SET_MUX_OUTPUTS      = 5'b11xxx; //where xxx will contain the info to tell us what the exact command should be for the mux selects
   parameter SET_SENSE_RAILS_ON   = 5'b10100; //turns on the tristate buffers on a given board
   parameter SET_SENSE_RAILS_OFF   = 5'b10000; //just shuts all the sense rails on the adressed board off

   //I2C error flags encodings   //either of these probably should trigger a bus rescan.
   parameter MISSED_ACK_ON_READ    = 1; //indicates that the beastie we want to read isn't responding
   parameter READ_NEIGHBOR_INVALID   = 2; //indicates we got different data than expected on a neighbor detect read


   //FSM STATES
   parameter MAIN_FSM_START      = 0;
   parameter BOUNDARY_SCAN_START_1 = 1;
   parameter BOUNDARY_SCAN_START_2 = 2;
   parameter BOUNDARY_SCAN_WAIT   = 3;
```

```
   parameter LAYOUT_MAPPING_START   = 4;
   parameter LAYOUT_MAPPING_1      = 5;
   parameter LAYOUT_MAPPING_2      = 6;
   parameter LAYOUT_MAPPING_3      = 7;
   parameter POLLING_START         = 8;
   parameter POLLING_1            = 9;
   parameter POLLING_2            = 10;
   parameter POLLING_3            = 11;

   parameter I2C_ERROR_HANDLING_1   = 12;
   //parameter I2C_ERROR_HANDLING_2   = 10;

   parameter XY_STATUS_BRAM_START    = 13;
   parameter XY_STATUS_BRAM1       = 14;



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
   //
   //   Combinational logic assignments
   //

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/

   ////////////////////////////////////////////////////
   //   clock domain crossing logic (and a few modules)
   ////////////////////////////////////////////////////

   //we assume reset is asynchronous and don't bother doing anything with it

   //sense signals
   wire sensor1_clean, sensor2_clean;
   synchronize SYNC2(clk_100mhz, sensor_1, sensor1_clean);
    synchronize SYNC3(clk_100mhz, sensor_2, sensor2_clean);


   //input clock domain transition

   wire mode_sync, rescan_trig_sync;
   reg rescan_trig_last;

   // always @(posedge clk_100mhz) begin
   //    rescan_trig_last    <= rescan_trig_sync;
   //    rescan_trig_pulse    <= rescan_trig_sync && ~ rescan_trig_last;
   // end

   //transpose read and write clocks

   mybram_dualclock #(.LOGSIZE(1),
               .WIDTH(2))
         input_buffer1(.r_addr(1'b0),
                     .w_addr(1'b0),
                     .clk_100mhz(clk_user),
                     .clk_user(clk_100mhz),
                     .din({mode,rescan_trig}),
                     .dout({mode_sync,rescan_trig_sync}),
                     .we(1'b1)
                     );



   //output clock domain transition

   reg done, config_in_progress;

      mybram_dualclock #(.LOGSIZE(1),
               .WIDTH(24))
         output_buffer1(.r_addr(1'b0),
                     .w_addr(1'b0),
                     .clk_100mhz(clk_100mhz),
                     .clk_user(clk_user),
                     .din({done,config_in_progress,x_upper_bound,y_upper_bound}),
                     .dout({done_out,config_in_progress_out,x_upper_bound_out,y_upper_bound_out}),
                     .we(1'b1)
                     );



   ////////////////////////////
   //   submodule reset signal
   ////////////////////////////

   wire reset_internal;

   assign reset_internal = reset || force_reset;

   ////////////////////////////
   //   I2C tristate signals
   ////////////////////////////
   assign scl_i = scl;
    assign scl = scl_t ? 1'bz : scl_o;
    assign sda_i = sda;
    assign sda = sda_t ? 1'bz : sda_o;

    ////////////////////////////
    //   MASTER FSM REGISTERS
    ////////////////////////////

    reg [4:0]   state;
    reg       force_reset;
    reg [23:0]   count;      //counter register

    assign master_fsm_state = state;

    reg rescan_requested;
```

```
    reg mapping_i2c_send_requested;
    reg polling_i2c_send_requested;

    reg init_scan;         //internal override for trigger, lets us do a startup scan without user asking for it.


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/
   //
   //   MASTER SEQUENCING FSM (and any other sequential logic that winds up being added on this layer)
   //

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/


   //this will start immediately after reset is deasserted, as opposed to all the submodules, which require start signals.
   always @(posedge clk_100mhz) begin

      if (reset) begin
         state             <= MAIN_FSM_START;
         force_reset       <= 1;
         sensor_blocks_reset <= 0;
         done              <= 0;
         config_in_progress    <= 1;
         count             <= 0;

         rescan_requested       <= 0;
         rescan_trig_last      <= 0;
         mapping_i2c_send_requested    <= 0;
         polling_i2c_send_requested    <= 0;

         i2c_start      <= 0;
          i2c_stop      <= 0;
         i2c_hold      <= 0;
         i2c_command    <= 0;
         i2c_address_int <= 0;

         mapping_start    <= 0;
         mapping_stop    <= 0;
         mapping_hold    <= 0;
         mapping_i2c_command_processed <= 0;

         polling_start    <= 0;
         polling_i2c_command_processed <= 0;

         init_scan      <= 0;

      end

      else begin
         case (state)

            ////////////////////////////////////////////////////////////////////////////////////////////////////
            //
            //   MAIN_FSM_START:
            //   wait a while. be damned sure the pics are reset and powered up before asking them to do things.
            //
            ////////////////////////////////////////////////////////////////////////////////////////////////////

            MAIN_FSM_START: begin
               done              <= 0;
               config_in_progress    <= 1;
               force_reset       <= 0;   //let FPGA stuff come up right away
               //count             <= (count < 200000) ? count + 1 : 0;                  //increment for two milliseconds (if we're running at 100MHz)
               count             <= (count < 6600000) ? count + 1 : 0;                  //wait for 66ms for PIC power on reset (if we're running at 100MHz) , datasheet
section 4.5
               sensor_blocks_reset <= (count < 6600000) ? 0 : 1;                       //deassert block reset after 1ms
               state             <= (count < 6600000) ? MAIN_FSM_START : BOUNDARY_SCAN_START_1; //delay another ms to be safe, then jump to next state
            end

            ////////////////////////////////////////////////////////////////////////////////////////////////////
            //
            //   BOUNDARY_SCAN_START_1:
            //   waits to make sure we figure things out before starting then initiates the I2C boundary scan as
            //   soon as we know the bus is available
            //
            ////////////////////////////////////////////////////////////////////////////////////////////////////

            BOUNDARY_SCAN_START_1: begin
               if (i2c_done) begin
                   init_scan         <= 0;
                  config_in_progress    <= 1;
                  done              <= 0;
                  force_reset       <= 0;

                  i2c_command       <= SCAN_ADDRESSES;   //boundary scan command
                  i2c_start         <= 1; //assert start and deassert as soon as it is acknowledged

                  state             <= BOUNDARY_SCAN_START_2; //holdoff 'til it has acknowledged that it got the request
               end
               else state <= BOUNDARY_SCAN_START_1;
            end

            ////////////////////////////////////////////////////////////////////////////////////////////////////
            //
            //   BOUNDARY_SCAN_START_2:
            //
            ////////////////////////////////////////////////////////////////////////////////////////////////////

            BOUNDARY_SCAN_START_2: begin
               i2c_start         <= 0; //assert start and deassert as soon as it is acknowledged
               state             <= BOUNDARY_SCAN_WAIT; //holdoff 'til it has acknowledged that it got the request
            end

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

```
//
//   BOUNDARY_SCAN_WAIT:
//   wait for boundary scan to finish
//
////////////////////////////////////////////////////////////////////////////////////////////////////

BOUNDARY_SCAN_WAIT: begin
   state    <=  i2c_done ? LAYOUT_MAPPING_START : BOUNDARY_SCAN_WAIT;
end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   LAYOUT_MAPPING_START:
 //   initiates layout mapping
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 LAYOUT_MAPPING_START: begin
    mapping_start     <= mapping_done ? 1 : 0; //hold start until it is acknowledged
    state          <= mapping_done ? LAYOUT_MAPPING_START : LAYOUT_MAPPING_1;   //wait until start before transition
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   LAYOUT_MAPPING_1:
 //   wait for a command from the mapping module and execute it
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 LAYOUT_MAPPING_1: begin
    mapping_i2c_command_processed    <= 0;

    if (mapping_i2c_send_requested) begin
       //i2c must be idle to get here, so don't check
       mapping_i2c_send_requested    <= 0;

       i2c_command       <= mapping_i2c_command;
       i2c_address_int    <= mapping_i2c_address_int;
       i2c_start          <= 1;

       state             <= LAYOUT_MAPPING_2;
    end
    else if (mapping_done) begin //if mapping is finished, initiate polling cycle
       count <= (count < 2000) ? count + 1 : 0;
       config_in_progress    <= 0;
       init_scan          <= 1; //scan capacitances immediately without waiting for trig
       state             <= (count < 2000) ? LAYOUT_MAPPING_1 : POLLING_START;
    end
    else state <= LAYOUT_MAPPING_1;
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   LAYOUT_MAPPING_2:
 //   delay for i2c to get command
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 LAYOUT_MAPPING_2: begin
    i2c_start <= 0;
    if (~i2c_done)    state <= LAYOUT_MAPPING_3;
    else          state <= LAYOUT_MAPPING_2;
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   LAYOUT_MAPPING_3:
 //   wait for command to process and return result
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 LAYOUT_MAPPING_3: begin
    if (i2c_bus_error) state <= I2C_ERROR_HANDLING_1;

    else if (i2c_done) begin
       //we let I2C data get returned directly to all modules. The command processed flag determines who uses it
       mapping_i2c_command_processed    <= 1;
       state                      <= LAYOUT_MAPPING_1;
    end

    else state    <= LAYOUT_MAPPING_3;
 end


////////////////////////////////////////////////////////////////////////////////////////////////////
//
//   POLLING_START:
//   when mode is high (either held or pulsed) let the polling module take over
//   mode and rescan should be checked here
//
////////////////////////////////////////////////////////////////////////////////////////////////////

POLLING_START: begin
   done <= 0;

   if (rescan_requested) begin
      rescan_requested <= 0;
      state    <= BOUNDARY_SCAN_START_1; //don't really need to go through start again. Odds are everything is ready by now
      //also, all the submodules are in an okay state if we do this here, so we don't need to reset them
   end

   else if (mode_sync || init_scan) begin
       polling_start    <= 1;
       init_scan      <= 0;
       state          <= POLLING_1;
   end

   else state    <= POLLING_START; //wait
```

```
end

////////////////////////////////////////////////////////////////////////////////////////////////////
//
//   POLLING_1:
//   wait for an I2C send request from the polling module and begin execution
//
////////////////////////////////////////////////////////////////////////////////////////////////////

POLLING_1: begin
    polling_i2c_command_processed    <= 0;


   if (polling_i2c_send_requested) begin
      polling_start <= 0;          //just an easy way to make sure this actually worked
       //i2c must be idle to get here, so don't check
       polling_i2c_send_requested    <= 0;

       i2c_command       <= polling_i2c_command;
       i2c_address_int    <= polling_i2c_address_int;
       i2c_start          <= 1;

       state             <= POLLING_2;
    end

    else if (polling_done) begin
       //done    <= 1;
       //state    <= POLLING_START;    // return to start state, and wait for a new trigger
       state <= XY_STATUS_BRAM_START;
    end

   else state    <= POLLING_1; //wait

end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   POLLING_2:
 //   delay for i2c to get command
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 POLLING_2: begin
    i2c_start <= 0;
    if (~i2c_done)    state <= POLLING_3;
    else          state <= POLLING_2;
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   POLLING_3:
 //   wait for command to process and return result
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////

 POLLING_3: begin
    if (i2c_bus_error) state <= I2C_ERROR_HANDLING_1;

    else if (i2c_done) begin
       //we let I2C data get returned directly to all modules. The command processed flag determines who uses it
       polling_i2c_command_processed    <= 1;
       state                       <= POLLING_1;
    end

    else state    <= POLLING_3;
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   XY_STATUS_BRAM_START:
 //   start generation of xy coordinate to status bram
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////
 XY_STATUS_BRAM_START: begin
    xy_status_bram_generation_start <= 1;
    state <= XY_STATUS_BRAM1;
 end

 ////////////////////////////////////////////////////////////////////////////////////////////////////
 //
 //   XY_STATUS_BRAM1:
 //   wait for completion of xy coordinate to status bram generation
 //
 ////////////////////////////////////////////////////////////////////////////////////////////////////
 XY_STATUS_BRAM1: begin
    xy_status_bram_generation_start <= 0;
    if (xy_status_bram_generation_done) begin
       done  <= 1;
       state <= POLLING_START;
    end
    else begin
       state <= XY_STATUS_BRAM1;
    end
 end


////////////////////////////////////////////////////////////////////////////////////////////////////
//
//   I2C_ERROR_HANDLING_1:
//   I2C has returned garbage or failed to respond. either way, something really bad has happened.
//   easiest solution which should always work is to restart the I2C bus and redo exverything
//
////////////////////////////////////////////////////////////////////////////////////////////////////

I2C_ERROR_HANDLING_1: begin
   force_reset   <= 1;            //reset all sub blocks. This won't actually do anything
                          //to the buffered map or data, so hopefull can't crash user processes
```

```
                                    //since they can choose to stop after a full cycle over already
                                    //recorded readings
                  state       <= MAIN_FSM_START;   //restart everything
               end

                   //default: state    <= state;
               default: state    <= MAIN_FSM_START; //if we wind up here, we've had an error and there's no way to reover except to start over anyway.

            endcase


         ////////////////////////////////////////////////////////////////////////////////////////////////////
         //
         //   control registers for handling requests/interrupts
         //
         ////////////////////////////////////////////////////////////////////////////////////////////////////

         // the way these are done is not great since nothing here explicitly prevents collision with a write to zero in the FSM. However,
         // in practice the way the modules that communicate with these are implemented prevents this, and Vivado seems to be okay with it
         // because of that.

         if (mapping_i2c_send_req)    mapping_i2c_send_requested         <= 1;
         if (polling_i2c_send_req)    polling_i2c_send_requested         <= 1;

         if (rescan_trig_sync && ~rescan_trig_last)    rescan_requested    <= 1;    // this one is scarier, this is only OK if input is a pulse

         rescan_trig_last    <= rescan_trig_sync;

         end
      end

endmodule

//dual clock domain BRAM
//dsheen 12/3/17
//only the 100MHz clock domain is capable of writing
//only the user clk domain can read
//this should work ok as long as the user clock is slower

module mybram_dualclock #(parameter LOGSIZE=14, WIDTH=1)
                  (input wire [LOGSIZE-1:0] r_addr,
                   input wire [LOGSIZE-1:0] w_addr,
                   input wire clk_100mhz,
                   input wire clk_user,
                   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_100mhz) begin
        if (we) begin
           mem[w_addr] <= din;
        end
      end

      always @(posedge clk_user) begin
        dout <= mem[r_addr];
    end
endmodule


//bram with separate read/write address
//apfitzen 12/2/17
//trying an experiment with this dsheen 12/3/17
//killed the dout <= din bypass to stop vivado from building LUTrams and wasting a ton of space
module mybram_rw1 #(parameter LOGSIZE=14, WIDTH=1)
                  (input wire [LOGSIZE-1:0] r_addr,
                   input wire [LOGSIZE-1:0] w_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) begin
        mem[w_addr] <= din;
        //if (r_addr == w_addr) begin
        //   dout <= din;
        //nd
        //else begin
           dout <= mem[r_addr];
        //end
     end
     else begin
        dout <= mem[r_addr];
     end

   end
endmodule
```

7.2.2 Demo Module (top block)

```
////////////////////////////////////////////////////////////////////////////////
//
//  hardware testbed for integrated module
//  started by dsheen 11/30/17
```

```
//  updated by apfitzen and dsheen 12/2/17
//      tweaked touchpads io and test functions,
//      eliminated coordinate error and sensor_data_previous outputs
//  this revision started 12/4/17 to try a few new things
//
//////////////////////////////////////////////////////////////////////////////////

module hardware_tb_multiclock_v3(
    input CLK100MHZ,
    input CPU_RESETN,
    inout [7:0] JA,
    input [15:0] SW,
    input BTNC, BTNU, BTNL, BTNR, BTND,
    output [15:0] LED,
    output [7:0] SEG,  // segments A-G (0-6), DP (7)
    output [7:0] AN,    // Display 0-7

    //VGA monitor things
    output [3:0] VGA_R,
    output [3:0] VGA_G,
    output [3:0] VGA_B,
    output VGA_HS,
    output VGA_VS
    );

    //assign LED[15:2] = 0;

    //clocking stuffs

    wire clk_100mhz;
    wire clk_65mhz;
    wire clk_10mhz;
    wire reset;
    wire locked;

    assign reset = ~CPU_RESETN;


    //this should not be reset on reset
    clk_wiz_0 CLK1(.clk_out1(clk_100mhz), .clk_out2(clk_65mhz), .clk_out3(clk_10mhz), .reset(1'b0), .locked(locked), .clk_in1(CLK100MHZ));

    //control
    reg  rescan_trig;
    //make a useable start signal

    wire button_center_bouncy, button_left_bouncy, button_right_bouncy;
    wire sig_in;
    reg sig_in_previous, trig_in_previous;
    synchronize SYNC1(clk_65mhz, BTNC, button_center_bouncy);
    synchronize SYNC4(clk_65mhz, BTNL, button_left_bouncy);
    synchronize SYNC5(clk_65mhz, BTNR, button_right_bouncy);
    //debounce DB1(reset,clk_100mhz, button_center_bouncy, sig_in);


    // always @(posedge clk_100mhz) begin
    //     sig_in_previous <= button_center_bouncy;
    //     mode <= ((sig_in_previous == 0) && button_center_bouncy); //generates a one clock cycle long pulse
    // end

    //assign mode = button_center_bouncy;
    assign mode = 1;

    always @(posedge clk_65mhz) begin
        trig_in_previous <= button_left_bouncy;
        rescan_trig <= ((trig_in_previous == 0) && button_left_bouncy); //generates a one clock cycle long pulse
    end

        //instantiate the touchpads module

    parameter   N           = 4;
    parameter   I2C_SPEED   = 100000;
    parameter   XY_QUADRANT = 4;

    wire mode;

    wire done;
    wire config_in_progress;

    wire [N+1:0]     x_upper_bound;      //these tell the user the maximum bounds of the XY coordinates occupied by sensors
    wire [N+1:0]     y_upper_bound;

    wire  [N+1:0]     x_coordinate;       //write these to query a sensor at a given XY coordinate
    wire  [N+1:0]     y_coordinate;
    wire  [N+1:0]     x_coordinate2;       //write these to query a sensor at a given XY coordinate
    wire  [N+1:0]     y_coordinate2;
```

```
    //assign x_coordinate = SW[2*N+3:N+2];
    //assign y_coordinate = SW[N+1:0];

    wire [11:0]      sensor_data_current;
    wire [11:0]      sensor_data_current2;
    wire [11:0]      sensor_data_previous;

    wire             user_coordinate_valid;
    wire             user_coordinate_valid2;
    wire             user_coordinate_error;

    wire [15:0] mapping_status;

    wire command_int_addr;
    wire polling_int_addr;



   // wire [4:0]      master_fsm_state;

    wire [11:0]     threshold;

    touchpads #(.N(N), .I2C_SPEED(I2C_SPEED), .XY_QUADRANT(XY_QUADRANT))
        touchpads1(
                    .clk_100mhz(clk_100mhz),
                    .clk_user(clk_65mhz),
                    .reset(reset),
                    .mode(mode),
                    .rescan_trig(rescan_trig),
                    .done_out(done),
                    .config_in_progress_out(config_in_progress),

                    .x_upper_bound_out(x_upper_bound),
                    .y_upper_bound_out(y_upper_bound),

                    .x_coordinate(x_coordinate),
                    .y_coordinate(y_coordinate),
                    .x_coordinate2(x_coordinate2),
                    .y_coordinate2(y_coordinate2),

                    .sensor_data_current(sensor_data_current),
                    .sensor_data_current2(sensor_data_current2),
                    //.sensor_data_previous(sensor_data_previous),

                    .user_coordinate_valid(user_coordinate_valid),
                    .user_coordinate_valid2(user_coordinate_valid2),
                    .mapping_status(mapping_status),
                    //.user_coordinate_error(user_coordinate_error),
                    .scl(JA[0]),    //SCL tristate signal. connect this directly to an external pin, call it an inout, and don't try to do anything
else clever with it!
                    .sda(JA[1]),   //SDA tristate signal. same goes for this.

                    .fpga_neighbor_detect(JA[2]),   //pin for figuring out who is next to the FPGA
                    .sensor_blocks_reset(JA[5]),    //global clear for the PICs active low
                    .sensor_1(JA[3]),
                    .sensor_2(JA[4]),

                    .polling_int_addr(polling_int_addr),
                    .command_int_addr(command_int_addr)

                    //.master_fsm_state(master_fsm_state)
                    );


    ////////////////////////////////////////
    //Display initialization
    ////////////////////////////////////////
    //xvga module generates the hsync, vsync, and blank signals
    //same as in lab 3

    wire [10:0] hcount;
    wire [9:0] vcount;
    wire [23:0] pixel_out;
    wire hsync, vsync, at_display_area, p_at_display_area;
    xvga xvga1(.vga_clock(clk_65mhz),.hcount(hcount),.vcount(vcount),
          .hsync(hsync),.vsync(vsync),.at_display_area(at_display_area));

    display #(.N(N))
            vga_display(.vclock(clk_65mhz),
                        .reset(reset),
                        .pad_size(75), //calculate based on block layout dimensions later
                        .pad_data(sensor_data_current),
                        .pad_valid(user_coordinate_valid),
                        .threshold(threshold),
                        .hcount(hcount),
                        .vcount(vcount),
```

```
                    .hsync(hsync),
                    .vsync(vsync),
                    .at_display_area(at_display_area),

                    .xCoord(x_coordinate),
                    .yCoord(y_coordinate),
                    .phsync(hsync_out), // display's horizontal sync
                    .pvsync(vsync_out), // display's vertical sync
                    .p_at_display_area(p_at_display_area), // display's blanking
                    .pixel(pixel_out)   // pixel color  // r=23:16, g=15:8, b=7:0
                    );

/*assign VGA_R = at_display_area ? {4{hcount[7]}} : 0;
assign VGA_G = at_display_area ? {4{hcount[6]}} : 0;
assign VGA_B = at_display_area ? {4{hcount[5]}} : 0;*/
assign VGA_R = p_at_display_area ? {pixel_out[23:20]} : 0;
assign VGA_G = p_at_display_area ? {pixel_out[15:12]} : 0;
assign VGA_B = p_at_display_area ? {pixel_out[7:4]} : 0;
assign VGA_HS = ~hsync_out;
assign VGA_VS = ~vsync_out;



num_pad num_pad1(
           .clk(clk_65mhz),
           .reset(reset),
           .pad_data(sensor_data_current2), //data from xy status bram for a pad
           .threshold(threshold),

           .xCoord(x_coordinate2), //the x coordinate of the pad we are reading
           .yCoord(y_coordinate2), //the y coordinate of the pad we are reading
           .xmit_data(JA[7])
          );

/*wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clk_65mhz),.hcount(hcount),.vcount(vcount),
          .hsync(hsync),.vsync(vsync),.blank(blank));

wire hsync_out, vsync_out, blank_out;
wire [23:0] pixel_out;

display #(.N(N))
        vga_display(.vclock(clk_65mhz),
                    .reset(reset),
                    .pad_size(150), //calculate based on block layout dimensions later
                    .pad_data(sensor_data_current),
                    .pad_valid(user_coordinate_valid),
                    .threshold(threshold),
                    .hcount(hcount),
                    .vcount(vcount),
                    .hsync(hsync),
                    .vsync(vsync),
                    .blank(blank),

                    // .padX(x_coordinate),
                    // .padY(y_coordinate),
                    .xCoord(x_coordinate),
                    .yCoord(y_coordinate),
                    .phsync(hsync_out), // display's horizontal sync
                    .pvsync(vsync_out), // display's vertical sync
                    .pblank(blank_out), // display's blanking
                    .pixel(pixel_out)   // pixel color  // r=23:16, g=15:8, b=7:0
                    );

//output assignments for VGA

assign VGA_HS = ~hsync_out;
assign VGA_VS = ~vsync_out;
assign VGA_R = pixel_out[23:20];
assign VGA_G = pixel_out[15:12];
assign VGA_B = pixel_out[7:4];*/



//display useful things
wire [31:0] display_data;
//assign display_data = {2'b00,sensor_data_previous,master_fsm_state[3:0],{2'b00,y_upper_bound},{2'b00,x_upper_bound}};
assign display_data = {6'b0000_00,threshold,{2'b00,y_upper_bound},{2'b00,x_upper_bound}};
display_8hex displ(clk_65mhz, display_data, SEG, AN);

//wire [11:0] threshold;

// use this for now just to check scanning and threshold
/*led_test led_test(.clock(clk_65mhz),
```

```
                .reset(reset),
                .we(user_coordinate_valid),
                .addr({x_coordinate[1:0],y_coordinate[1:0]}),
                .data(sensor_data_current),
                .leds(LED[15:0]),
                .threshold(threshold)
                );*/

    //use this to test mapping module, mapping_status still on 100mhz clock
    mapping_led_test mapping_led_test1(.clock(clk_100mhz),
                                       .reset(reset),
                                       .status(mapping_status),
                                       .leds(LED[15:0]));

    //scanning should be provided by the VGA module


    // scanner #(.N(N))
    //         scanner1(
    //                 .clk(clk_65mhz),
    //                 .reset(reset),
    //                 .x_upper_bound(x_upper_bound),
    //                 .y_upper_bound(y_upper_bound),
    //                 .x_coordinate(x_coordinate),
    //                 .y_coordinate(y_coordinate)
    // );

    wire [11:0] cal_threshold;
    assign threshold = SW[0] ? (cal_threshold - SW[4:1]) : (cal_threshold + SW[4:1]);

    simple_calibration #(.N(N))
                call(
                    .clk(clk_65mhz),
                    .reset(reset),
                    .start(button_right_bouncy),
                    //    done,           //could equivalently be called valid
                    .x_upper_bound(x_upper_bound),
                    .y_upper_bound(y_upper_bound),
                    .x_coordinate(x_coordinate),
                    .y_coordinate(y_coordinate),
                    .valid(user_coordinate_valid),
                    .data(sensor_data_current),
                    .threshold(cal_threshold)
                    );

endmodule


////////////////////////
// io test module
////////////////////////

// module user_io_test
//     #(parameter N = 4)
//     (
//         input wire clk,
//         input wire reset,
//         input wire [11:0] threshold,
//         input wire [11:0] data,

//         output wire [N+1:0] x_out,
//         output wire [N+1:0] y_out,
//         output reg [15:0] led_out
//     );


//     reg [3:0] count;

//     //assign status_bram_addr[3:0] = {count[1:0], count[3:2]};
//     assign x_out[1:0]   = count[1:0];
//     assign x_out[N+1:2] = 0;
//     assign y_out[1:0]   = count[3:2];
//     assign y_out[N+1:2] = 0;


//     always @(posedge clk) begin
//         if (reset) begin
//             led_out <= 0;
//             count <= 0;
//         end
//         else begin
//             count <= count + 1;

//             if (data < threshold) begin
//                 led_out[count-1] <= 1;
//             end
```

```
//             else led_out[count-1] <= 0;
//         end
//     end

// endmodule

module led_test(input clock,
                 input reset,
                 input we,
                 input [3:0] addr,
                 input [11:0] data,
                 output reg [15:0] leds,
                 input [11:0] threshold);

        reg [3:0] address;

        always @(posedge clock) begin
            address <= addr;
            if (reset) begin
                leds <= 0;
            end
            else begin
                if (we) begin
                    if (data < threshold) begin
                        leds[address] <= 1;
                    end
                    else begin
                        leds[address] <= 0;
                    end
                end
            end
        end

endmodule

module mapping_led_test(input clock,
                 input reset,
                 input [15:0] status,
                 output reg [15:0] leds
                 );

        parameter delay = 25000000;
        reg [31:0] delay_count;

        always @(posedge clock) begin
            if (reset) begin
                leds <= 0;
            end
            else begin
                if (delay_count == delay) begin
                    leds <= (~leds) & status; //blink leds if mapping bit is 1
                end
                else begin
                    delay_count <= delay_count + 1;
                end
            end
        end

endmodule
```

7.2.3 User Calibration Module

```
////////////////////////////////////////////////////////////////////////////////////////////////
//
//   dsheen 12/4/17
//   because I'm tired of figuring out the threshold for the modules and this seems easy to implement
//
//   assumes the modules are all probably pretty similar (no huge variations in behavior between blocks)
//   playing a bit of a game here. this doesn't do the actual scanning, allows it to be alongside
//   anything else running
//
////////////////////////////////////////////////////////////////////////////////////////////////

module simple_calibration #(parameter N = 4)
   (
   input             clk,
   input             reset,
   input            start,
   output reg         done,         //could equivalently be called valid
   input       [N+1:0] x_upper_bound,
```

```
   input       [N+1:0] y_upper_bound,
   input       [N+1:0]   x_coordinate,
   input       [N+1:0]   y_coordinate,
   input             valid,
   input       [11:0]   data,
   output reg    [11:0]   threshold
   );

   reg [11:0]    min;
   reg [1:0]   state;

   parameter    IDLE    = 0;
   parameter    WAIT    = 1;
   parameter    RECORD   = 2;
   parameter    FINISH   = 3;

   always @(posedge clk) begin
      if (reset) begin
         state       <= IDLE;
         threshold    <= 0; //keeps anything using it from triggering while it's doing this
         min       <= 11'b111_1111_1111_1111;
      end
      else begin
         case (state)
            ////////////////////////////////////////////
            //
            //   wait for a start command
            //
            ////////////////////////////////////////////
            IDLE: begin
               done    <= start ? 0 : 1;
               state    <= start ? WAIT : IDLE;
               min    <= 11'b111_1111_1111_1111;

               //note that we don't change the threshold here so this never disturbs
               //anyhing running until it has the new threhold ready
            end

            ////////////////////////////////////////////
            //
            //   wait for the scan to cycle to zero
            //
            ////////////////////////////////////////////

            WAIT: begin
               state <= (|{x_coordinate,y_coordinate}) ? WAIT : RECORD; //reduction on, so only if both are zero does the state change
            end

            ////////////////////////////////////////////
            //
            //   record highest values
            //
            ////////////////////////////////////////////

            RECORD: begin
               state <= (|{x_coordinate,y_coordinate}) ? RECORD : FINISH; //reduction on, so only if both are zero does the state change
               if (valid) min <= (data < min) ? data : min;
            end

            ////////////////////////////////////////////
            //
            //   finish this thing
            //
            ////////////////////////////////////////////

            FINISH: begin
               state       <= IDLE;
               done       <= 1;
               threshold    <= min - 5; //oughtta be enough for stability
            end
         endcase
      end
   end


endmodule


////////////////////////////////////////////////////////////////////////////////
//
//   scanner module
//   just scans across the XY coordinate space
//
////////////////////////////////////////////////////////////////////////////////

module scanner #(parameter N = 4)
```

```
   (
   input             clk,
   input             reset,
   input       [N+1:0] x_upper_bound,
   input       [N+1:0] y_upper_bound,
   output reg    [N+1:0]   x_coordinate,
   output reg    [N+1:0]   y_coordinate
   );



   always @(posedge clk) begin
      if (reset) begin
         x_coordinate <= 0;
         y_coordinate <= 0;
      end

      else begin
         if (y_coordinate < y_upper_bound) y_coordinate <= y_coordinate + 1;
         else if (x_coordinate < x_upper_bound) begin
            y_coordinate <= 0;
            x_coordinate <= x_coordinate + 1;
         end
         else begin
            x_coordinate <= 0;
            y_coordinate <= 0;
         end
      end
   end


endmodule
```

7.2.4 Clocking Modules Generated In Vivado

```
// file: clk_wiz_0.v
//
// (c) Copyright 2008 - 2013 Xilinx, Inc. All rights reserved.
//
// This file contains confidential and proprietary information
// of Xilinx, Inc. and is protected under U.S. and
// international copyright and other intellectual property
// laws.
//
// DISCLAIMER
// This disclaimer is not a license and does not grant any
// rights to the materials distributed herewith. Except as
// otherwise provided in a valid license issued to you by
// Xilinx, and to the maximum extent permitted by applicable
// law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
// WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
// AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
// BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
// INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
// (2) Xilinx shall not be liable (whether in contract or tort,
// including negligence, or under any other theory of
// liability) for any loss or damage of any kind or nature
// related to, arising under or in connection with these
// materials, including for any direct, or any indirect,
// special, incidental, or consequential loss or damage
// (including loss of data, profits, goodwill, or any type of
// loss or damage suffered as a result of any action brought
// by a third party) even if such damage or loss was
// reasonably foreseeable or Xilinx had been advised of the
// possibility of the same.
//
// CRITICAL APPLICATIONS
// Xilinx products are not designed or intended to be fail-
// safe, or for use in any application requiring fail-safe
// performance, such as life-support or safety devices or
// systems, Class III medical devices, nuclear facilities,
// applications related to the deployment of airbags, or any
// other applications that could lead to death, personal
// injury, or severe property or environmental damage
// (individually and collectively, "Critical
// Applications"). Customer assumes the sole risk and
// liability of any use of Xilinx products in Critical
// Applications, subject only to applicable laws and
// regulations governing limitations on product liability.
//
// THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
```

```
// PART OF THIS FILE AT ALL TIMES.
//
//------------------------------------------------------------------------------
// User entered comments
//------------------------------------------------------------------------------
// None
//
//------------------------------------------------------------------------------
//  Output     Output      Phase    Duty Cycle   Pk-to-Pk     Phase
//   Clock     Freq (MHz)  (degrees)    (%)     Jitter (ps)  Error (ps)
//------------------------------------------------------------------------------
// clk_out1___100.000______0.000______50.0______145.553____124.502
// clk_out2____65.000______0.000______50.0______159.200____124.502
// clk_out3____10.000______0.000______50.0______231.613____124.502
//
//------------------------------------------------------------------------------
// Input Clock   Freq (MHz)    Input Jitter (UI)
//------------------------------------------------------------------------------
// __primary_________100.000____________0.010

`timescale 1ps/1ps

module clk_wiz_0_clk_wiz


 (// Clock in ports
  // Clock out ports
  output        clk_out1,
  output        clk_out2,
  output        clk_out3,
  // Status and control signals
  input         reset,
  output        locked,
  input         clk_in1
 );
  // Input buffering
  //------------------------------------
wire clk_in1_clk_wiz_0;
wire clk_in2_clk_wiz_0;
  IBUF clkin1_ibufg
   (.O (clk_in1_clk_wiz_0),
    .I (clk_in1));


  // Clocking PRIMITIVE
  //------------------------------------

  // Instantiation of the MMCM PRIMITIVE
  //    * Unused inputs are tied off
  //    * Unused outputs are labeled unused

  wire        clk_out1_clk_wiz_0;
  wire        clk_out2_clk_wiz_0;
  wire        clk_out3_clk_wiz_0;
  wire        clk_out4_clk_wiz_0;
  wire        clk_out5_clk_wiz_0;
  wire        clk_out6_clk_wiz_0;
  wire        clk_out7_clk_wiz_0;

  wire [15:0] do_unused;
  wire        drdy_unused;
  wire        psdone_unused;
  wire        locked_int;
  wire        clkfbout_clk_wiz_0;
  wire        clkfbout_buf_clk_wiz_0;
  wire        clkfboutb_unused;
    wire clkout0b_unused;
   wire clkout1b_unused;
   wire clkout2b_unused;
   wire clkout3_unused;
   wire clkout3b_unused;
   wire clkout4_unused;
  wire        clkout5_unused;
  wire        clkout6_unused;
  wire        clkfbstopped_unused;
  wire        clkinstopped_unused;
  wire        reset_high;

  MMCME2_ADV
  #(.BANDWIDTH            ("OPTIMIZED"),
    .CLKOUT4_CASCADE      ("FALSE"),
    .COMPENSATION         ("ZHOLD"),
    .STARTUP_WAIT         ("FALSE"),
    .DIVCLK_DIVIDE        (1),
    .CLKFBOUT_MULT_F      (6.500),
    .CLKFBOUT_PHASE       (0.000),
    .CLKFBOUT_USE_FINE_PS ("FALSE"),
```

```
    .CLKOUT0_DIVIDE_F     (6.500),
    .CLKOUT0_PHASE        (0.000),
    .CLKOUT0_DUTY_CYCLE   (0.500),
    .CLKOUT0_USE_FINE_PS  ("FALSE"),
    .CLKOUT1_DIVIDE       (10),
    .CLKOUT1_PHASE        (0.000),
    .CLKOUT1_DUTY_CYCLE   (0.500),
    .CLKOUT1_USE_FINE_PS  ("FALSE"),
    .CLKOUT2_DIVIDE       (65),
    .CLKOUT2_PHASE        (0.000),
    .CLKOUT2_DUTY_CYCLE   (0.500),
    .CLKOUT2_USE_FINE_PS  ("FALSE"),
    .CLKIN1_PERIOD        (10.000))
  mmcm_adv_inst
    // Output clocks
   (
    .CLKFBOUT            (clkfbout_clk_wiz_0),
    .CLKFBOUTB           (clkfboutb_unused),
    .CLKOUT0             (clk_out1_clk_wiz_0),
    .CLKOUT0B            (clkout0b_unused),
    .CLKOUT1             (clk_out2_clk_wiz_0),
    .CLKOUT1B            (clkout1b_unused),
    .CLKOUT2             (clk_out3_clk_wiz_0),
    .CLKOUT2B            (clkout2b_unused),
    .CLKOUT3             (clkout3_unused),
    .CLKOUT3B            (clkout3b_unused),
    .CLKOUT4             (clkout4_unused),
    .CLKOUT5             (clkout5_unused),
    .CLKOUT6             (clkout6_unused),
     // Input clock control
    .CLKFBIN             (clkfbout_buf_clk_wiz_0),
    .CLKIN1              (clk_in1_clk_wiz_0),
    .CLKIN2              (1'b0),
     // Tied to always select the primary input clock
    .CLKINSEL            (1'b1),
    // Ports for dynamic reconfiguration
    .DADDR               (7'h0),
    .DCLK                (1'b0),
    .DEN                 (1'b0),
    .DI                  (16'h0),
    .DO                  (do_unused),
    .DRDY                (drdy_unused),
    .DWE                 (1'b0),
    // Ports for dynamic phase shift
    .PSCLK               (1'b0),
    .PSEN                (1'b0),
    .PSINCDEC            (1'b0),
    .PSDONE              (psdone_unused),
    // Other control and status signals
    .LOCKED              (locked_int),
    .CLKINSTOPPED        (clkinstopped_unused),
    .CLKFBSTOPPED        (clkfbstopped_unused),
    .PWRDWN              (1'b0),
    .RST                 (reset_high));
  assign reset_high = reset;

  assign locked = locked_int;
// Clock Monitor clock assigning
//--------------------------------------
 // Output buffering
  //-----------------------------------

  BUFG clkf_buf
   (.O (clkfbout_buf_clk_wiz_0),
    .I (clkfbout_clk_wiz_0));



  BUFG clkout1_buf
   (.O   (clk_out1),
    .I   (clk_out1_clk_wiz_0));


  BUFG clkout2_buf
   (.O   (clk_out2),
    .I   (clk_out2_clk_wiz_0));

  BUFG clkout3_buf
   (.O   (clk_out3),
    .I   (clk_out3_clk_wiz_0));



endmodule
```

```
// file: clk_wiz_0.v
//
// (c) Copyright 2008 - 2013 Xilinx, Inc. All rights reserved.
//
// This file contains confidential and proprietary information
// of Xilinx, Inc. and is protected under U.S. and
// international copyright and other intellectual property
// laws.
//
// DISCLAIMER
// This disclaimer is not a license and does not grant any
// rights to the materials distributed herewith. Except as
// otherwise provided in a valid license issued to you by
// Xilinx, and to the maximum extent permitted by applicable
// law: (1) THESE MATERIALS ARE MADE AVAILABLE "AS IS" AND
// WITH ALL FAULTS, AND XILINX HEREBY DISCLAIMS ALL WARRANTIES
// AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING
// BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-
// INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and
// (2) Xilinx shall not be liable (whether in contract or tort,
// including negligence, or under any other theory of
// liability) for any loss or damage of any kind or nature
// related to, arising under or in connection with these
// materials, including for any direct, or any indirect,
// special, incidental, or consequential loss or damage
// (including loss of data, profits, goodwill, or any type of
// loss or damage suffered as a result of any action brought
// by a third party) even if such damage or loss was
// reasonably foreseeable or Xilinx had been advised of the
// possibility of the same.
//
// CRITICAL APPLICATIONS
// Xilinx products are not designed or intended to be fail-
// safe, or for use in any application requiring fail-safe
// performance, such as life-support or safety devices or
// systems, Class III medical devices, nuclear facilities,
// applications related to the deployment of airbags, or any
// other applications that could lead to death, personal
// injury, or severe property or environmental damage
// (individually and collectively, "Critical
// Applications"). Customer assumes the sole risk and
// liability of any use of Xilinx products in Critical
// Applications, subject only to applicable laws and
// regulations governing limitations on product liability.
//
// THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
// PART OF THIS FILE AT ALL TIMES.
//
//------------------------------------------------------------------------------
// User entered comments
//------------------------------------------------------------------------------
// None
//
//------------------------------------------------------------------------------
//  Output     Output      Phase    Duty Cycle   Pk-to-Pk     Phase
//   Clock     Freq (MHz)  (degrees)    (%)     Jitter (ps)  Error (ps)
//------------------------------------------------------------------------------
// clk_out1___100.000______0.000______50.0______145.553___124.502
// clk_out2____65.000______0.000______50.0______159.200___124.502
// clk_out3____10.000______0.000______50.0______231.613___124.502
//
//------------------------------------------------------------------------------
// Input Clock   Freq (MHz)    Input Jitter (UI)
//------------------------------------------------------------------------------
// __primary_________100.000____________0.010

`timescale 1ps/1ps

(* CORE_GENERATION_INFO =
"clk_wiz_0,clk_wiz_v5_4_1_0,{component_name=clk_wiz_0,use_phase_alignment=true,use_min_o_jitter=false,use_max_i_jitter=false,use_dyn_phase_shift=fal
se,use_inclk_switchover=false,use_dyn_reconfig=false,enable_axi=0,feedback_source=FDBK_AUTO,PRIMITIVE=MMCM,num_out_clk=3,clkin1_period=10.000,clkin2
_period=10.000,use_power_down=false,use_reset=true,use_locked=true,use_inclk_stopped=false,feedback_type=SINGLE,CLOCK_MGR_TYPE=NA,manual_override=fa
lse}" *)

module clk_wiz_0
 (
  // Clock out ports
  output        clk_out1,
  output        clk_out2,
  output        clk_out3,
  // Status and control signals
  input         reset,
  output        locked,
 // Clock in ports
  input         clk_in1
```

```
 );

  clk_wiz_0_clk_wiz inst
  (
  // Clock out ports
  .clk_out1(clk_out1),
  .clk_out2(clk_out2),
  .clk_out3(clk_out3),
  // Status and control signals
  .reset(reset),
  .locked(locked),
 // Clock in ports
  .clk_in1(clk_in1)
  );

endmodule
```

7.2.5 Debounce

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=1000000)   // .01 sec with a 100Mhz clock
            (input 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
       new <= noisy;
       count <= 0;
     end
     else if (count == DELAY)
       clean <= new;
     else
       count <= count+1;

endmodule
```

7.2.6 8 bit display module

```
//`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:   g.p.hom
// Engineer:
//
// Create Date:    18:18:59 04/21/2013
// Module Name:    display_8hex

// Description:  Display 8 hex numbers on 7 segment display
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module display_8hex(
    input clk,                 // system clock
    input [31:0] data,         // 8 hex numbers, msb first
    output reg [6:0] seg,      // seven segment display output
    output reg [7:0] strobe    // digit strobe
    );

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;
    assign segments[1]  = 7'b111_1001;
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
```

```
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
    assign segments[10] = 7'b000_1000;
    assign segments[11] = 7'b000_0011;
    assign segments[12] = 7'b010_0111;
    assign segments[13] = 7'b010_0001;
    assign segments[14] = 7'b000_0110;
    assign segments[15] = 7'b000_1110;

    always @(posedge clk) begin
      counter <= counter + 1;
      case (counter[bits:bits-2])
          3'b000: begin
                  seg <= segments[data[31:28]];
                  strobe <= 8'b0111_1111 ;
                 end

          3'b001: begin
                  seg <= segments[data[27:24]];
                  strobe <= 8'b1011_1111 ;
                 end

          3'b010: begin
                   seg <= segments[data[23:20]];
                   strobe <= 8'b1101_1111 ;
                  end
          3'b011: begin
                  seg <= segments[data[19:16]];
                  strobe <= 8'b1110_1111;
                 end
          3'b100: begin
                  seg <= segments[data[15:12]];
                  strobe <= 8'b1111_0111;
                 end

          3'b101: begin
                  seg <= segments[data[11:8]];
                  strobe <= 8'b1111_1011;
                 end

          3'b110: begin
                   seg <= segments[data[7:4]];
                   strobe <= 8'b1111_1101;
                  end
          3'b111: begin
                  seg <= segments[data[3:0]];
                  strobe <= 8'b1111_1110;
                 end




       endcase
      end

endmodule
```

7.2.8 Bram Modules

```
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

//bram with separate read/write address
//apfitzen 12/2/17
module mybram_rw #(parameter LOGSIZE=14, WIDTH=1)
              (input wire [LOGSIZE-1:0] r_addr,
```

```
              input wire [LOGSIZE-1:0] w_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) begin
        mem[w_addr] <= din;
        if (r_addr == w_addr) begin
           dout <= din;
        end
        else begin
           dout <= mem[r_addr];
        end
     end
     else begin
        dout <= mem[r_addr];
     end

   end
endmodule
```

7.2.9 Synchronize Module

```
// pulse synchronizer
module synchronize #(parameter NSYNC = 2)  // number of sync flops.  must be >= 2
                   (input clk,in,
                    output reg out);

  reg [NSYNC-2:0] sync;

  always @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule
```

7.2.10 I2C Commands Module

```
////////////////////////////////////////////////////////////////////////////////
//
//  This module implements the command and control for the I2C interface
//   also manages I2C boundary scan internal to the module.
//
//  dsheen 11/12/2017
//
////////////////////////////////////////////////////////////////////////////////

module i2c_commands #(parameter N=4, parameter COMMAND_LEN=5, parameter I2C_SPEED=400000)
   (
   //global signals
   input wire       clk_100mhz,
   input wire       reset,

   //control
   input wire       start,   //assert start to initiate a response to inputted command (doesn't effect control signals)
                     //also assert to clear an erro state (makes sure controling hardware acknowledges it)
   input wire       stop,   //haven't decided what to do with this, may need to make it generate a jump to a state which figures out
                     //what the I2C FSM is doing and then ends the transaction appropriately
   input wire       hold,   //halts control FSM in current state while high, resumes when released.
   output reg       done,   //indicates that the FSM has finished executing a command.
   output reg       [4:0] state,   //state output for debus and syncing with system control module

   //command handling
   input wire       [COMMAND_LEN-1:0] command,
   input wire       [N-1:0] address_int,       //internal address used by other modules
   output reg       [7:0] return_data,         //multipurpose return data output, different commands may use this differently
   output reg       bus_error,               //flag for something went wrong. error code will be contained in return data.

   //intenal address upper bound
   output reg [N-1:0] max_internal_address,

   //passthrough for I2C
   input  wire       scl_i,
    output wire       scl_o,
    output wire       scl_t,
    input  wire       sda_i,
```

```
 output wire       sda_o,
 output wire       sda_t
);

//some general system parameters
parameter INTERNAL_ADDRESS_BOUND = 2**N-1;
parameter MAX_I2C_ADDRESS = 127;


//instantiate wires needed to connect to the I2C module

//command handling
reg [6:0]     cmd_address;
 reg           cmd_start;
 reg           cmd_read;
 reg           cmd_write;
 reg           cmd_write_multiple;
 reg           cmd_stop;
 reg           cmd_valid;
 wire        cmd_ready;

 //send data handling
 reg [7:0]     data_in;
 reg             data_in_valid;
 wire          data_in_ready;
 reg           data_in_last;

 //recieve data handling
 wire [7:0]  data_out;
 wire          data_out_valid;
 reg           data_out_ready;
 wire        data_out_last;

 //Status
 wire        busy;
 wire        bus_control;
 wire        bus_active;
 wire        missed_ack;

 //Configuration
 wire [15:0] prescale;
 wire        stop_on_idle;


 assign prescale = 100000000/(4*I2C_SPEED);
 assign stop_on_idle = 0;

 reg i2c_force_reset;
 wire reset_i2c;
 assign reset_i2c = i2c_force_reset | reset; //putting this in to allow me to forcibly end failed bus transactions

 //instantiate master I2C FSM
 i2c_master
   I2C(
       .clk(clk_100mhz),
       .rst(reset_i2c),
       .cmd_address(cmd_address),
       .cmd_start(cmd_start),
       .cmd_read(cmd_read),
       .cmd_write(cmd_write),
       .cmd_write_multiple(cmd_write_multiple),
       .cmd_stop(cmd_stop),
       .cmd_valid(cmd_valid),
       .cmd_ready(cmd_ready),
       .data_in(data_in),
       .data_in_valid(data_in_valid),
       .data_in_ready(data_in_ready),
       .data_in_last(data_in_last),
       .data_out(data_out),
       .data_out_valid(data_out_valid),
       .data_out_ready(data_out_ready),
       .data_out_last(data_out_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(busy),
       .bus_control(bus_control),
       .bus_active(bus_active),
       .missed_ack(missed_ack),
       .prescale(prescale),
       .stop_on_idle(stop_on_idle)
      );
```

```
//commands (for now just placeholders here, though maybe we'll keep some of these encodings)
parameter SCAN_ADDRESSES       = 5'b00000; //this triggers the boundary scan FSM

parameter NEIGHBOR_DETECT_HIGH    = 5'b00001; //write neighbor detect output high
parameter NEIGHBOR_DETECT_LOW    = 5'b00010; //write neighbor detect output low
parameter SENSE_NEIGHBORS       = 5'b00011; //needs to trigger a read and return suitable data

parameter SET_MUX_OUTPUTS      = 5'b11xxx; //where xxx will contain the info to tell us what the exact command should be for the mux selects
parameter SET_SENSE_RAILS_ON   = 5'b10100; //turns on the tristate buffers on a given board
parameter SET_SENSE_RAILS_OFF   = 5'b10000; //just shuts all the sense rails on the adressed board off


//I2C command structures
parameter RESET_ALL_OUTPUTS                = 8'b0000_0000; //actually just shuts off tristate buffers
parameter SET_NEIGHBOR_DETECT_OUTPUT_LOW      = 8'b0100_0000; //detect output off
parameter SET_NEIGHBOR_DETECT_OUTPUT_HIGH      = 8'b0110_0000; //detect output on
parameter SET_TRISTATE_BUFFERS_ON            = 8'b0000_0011; //turns on tristate buffers
parameter SET_TRISTATE_BUFFERS_OFF            = 8'b0000_0000; //shuts off tristate buffers


//STATE NAMES (I'll be tacking more on as we implement commands)
//idle state
parameter IDLE = 0;
//address lookup states
parameter ADDRESS_FETCH_1 = 1;
parameter ADDRESS_FETCH_2 = 2;
//boundary scan states
parameter ADDRESS_SCAN_1 = 3;
parameter ADDRESS_SCAN_2 = 4;
parameter ADDRESS_SCAN_3 = 5;
parameter ADDRESS_SCAN_4 = 6;
parameter ADDRESS_SCAN_5 = 7;
//i2c write handling states
parameter WRITE_CYCLE_1 = 8;
parameter WRITE_CYCLE_2 = 9;
parameter WRITE_CYCLE_3 = 10;
//i2c read handling states
parameter READ_CYCLE_1 = 11;
parameter READ_CYCLE_2 = 12;
parameter READ_CYCLE_3 = 13;
//command entry states (these set control registes, call read/write, and handle returns)
parameter NEIGHBOR_DETECT_SET    = 14;
parameter SENSE_RAILS_SET      = 15;
parameter MUX_OUTPUTS_SET      = 16;
parameter READ_NEIGHBORS_1      = 17; //this one sets things up and call read
parameter READ_NEIGHBORS_2      = 18; //this one handles the value returned on a successfull read
//what if something goes wrong that other modules nead to know about (for example, a failed read)?
parameter BUS_ERROR_HANDLER_1   = 19; //if something goes wrong, his will handle it. probably need to add module IO for this.
parameter BUS_ERROR_HANDLER_2   = 20; //if something goes wrong, his will handle it. probably need to add module IO for this.



//signals used by boundary scan and address lookup bram
reg [N-1:0] internal_address; //address pointing to real address in the translation BRAM
reg internal_address_overflow;
reg address_lookup_we;
reg [6:0] address_lookup_din;
wire [6:0] address_lookup_dout;

//storage register for commands during address lookup
reg [COMMAND_LEN-1:0] next_i2c_command;

// other handy registers
reg [4:0] return_state; //allows FSM to return to a state other than IDLE after a write if needed (ie, for more complex I2C transactions)
reg [7:0] i2c_return_data;
reg [2:0] error_flag;

//error flag encodings (don't use zero, we want to reserve an invalid option)
parameter MISSED_ACK_ON_READ       = 1; //indicates that the beastie we want to read isn't responding
parameter READ_NEIGHBOR_INVALID      = 2; //indicates we got different data than expected on a neighbor detect read



//bram that will store the address lookup information for most operations
mybram #(.LOGSIZE(N),.WIDTH(7))
    address_abstract_1(.addr(internal_address),.clk(clk_100mhz),.we(address_lookup_we),.din(address_lookup_din),.dout(address_lookup_dout));

 //main FSM for command handling
 //this whole thing really behaves like a bunch of separate little FSMs with simple FSM that calls them,
 //but it's a lot easier to implement as one big one FSM

 always @(posedge clk_100mhz) begin
```

```
//response for a reset
if (reset) begin
   state             <= IDLE;
   done            <= 1;

   cmd_address       <= 0;
   cmd_start          <= 0;
     cmd_read          <= 0;
   cmd_write          <= 0;
   cmd_write_multiple    <= 0;
   cmd_stop         <= 0;
   cmd_valid         <= 0;

   data_in          <= 0;
   data_in_valid       <= 0;
   data_in_last      <= 0;

   data_out_ready      <= 0;

   internal_address          <= 0;
   internal_address_overflow    <= 0;
   address_lookup_we         <= 0;
   address_lookup_din         <= 0;
   max_internal_address      <= 0;

   next_i2c_command   <= 0;
   return_state      <= 0;
   i2c_force_reset      <= 0;
   bus_error          <= 0;
   error_flag         <= 0;
   i2c_return_data      <= 0;

end
else if (hold) state <= state; //just keep the FSM from doing anything until hold is released

else begin
   case (state)

      ////////////////////////////////////////////////////////////////////////////////////////
      //
      //   IDLE STATE
      //
      ////////////////////////////////////////////////////////////////////////////////////////

      IDLE: begin
         done    <= 1; //need to set this here because I can't always set it during the return
         if (start) begin
            //command recieved to start boundary scan
            if (command == SCAN_ADDRESSES) begin
               state <= ADDRESS_SCAN_1;
               //cmd_address <= 0;
               //to keep it from picking up global address 0 during this
               cmd_address <= 1;
               internal_address <=0;
               max_internal_address <=0;
               done <= 0;
            end
            //command to write neighbor detect high.
            else if ((command == NEIGHBOR_DETECT_HIGH)|(command == NEIGHBOR_DETECT_LOW)
                  |(command == SENSE_NEIGHBORS)|(command == SET_SENSE_RAILS_ON)
                  |(command == SET_SENSE_RAILS_OFF)|(command[4:3] == SET_MUX_OUTPUTS[4:3])
               )begin

               state <= ADDRESS_FETCH_1;
               next_i2c_command <= command;
               cmd_address <= 0;
               internal_address <= address_int;
               done <= 0;
            end
         end
         //if no command is recieved
         else state <= IDLE;

      end

      ////////////////////////////////////////////////////////////////////////////////////////
      //
      //   ADDRESS LOOKUP STATES
      //    these just exist to make it wait untill we have the correct I2C address
      //
      ////////////////////////////////////////////////////////////////////////////////////////

      ADDRESS_FETCH_1: begin
         state <= ADDRESS_FETCH_2;
      end
```

```
            ADDRESS_FETCH_2: begin
               cmd_address <= address_lookup_dout; //we wrote the input address to it two cycles ago, so this should be the address we want now.

               //jump to appropriate FSM state once we have the address
               if ((next_i2c_command == NEIGHBOR_DETECT_HIGH)|(next_i2c_command == NEIGHBOR_DETECT_LOW))
                  state <= NEIGHBOR_DETECT_SET;

               else if ((next_i2c_command == SET_SENSE_RAILS_ON)|(next_i2c_command == SET_SENSE_RAILS_OFF))
                  state <= SENSE_RAILS_SET;

               else if (next_i2c_command[4:3] == SET_MUX_OUTPUTS[4:3])
                  state <= MUX_OUTPUTS_SET;

               else if (next_i2c_command == SENSE_NEIGHBORS)
                  state <= READ_NEIGHBORS_1;

               else begin //something went horribly wrong
                  state <= IDLE;
                  done <= 1;
               end
            end


            ////////////////////////////////////////////////////////////////////////////////
            //
            //   boundary scan states
            //
            ////////////////////////////////////////////////////////////////////////////////

            ADDRESS_SCAN_1: begin
               if (cmd_ready) begin
                  cmd_valid    <= 1;
                  cmd_write    <= 1;
                  cmd_start    <= 1;
                  cmd_stop   <= 1;
                  address_lookup_din <= cmd_address;
                  state <= ADDRESS_SCAN_2;
               end
               else state <= ADDRESS_SCAN_1;
            end

            ADDRESS_SCAN_2: begin
                  cmd_valid    <= 0;
                  cmd_write    <= 0;
                  cmd_start    <= 0;
               if (missed_ack) begin
                  state <= ADDRESS_SCAN_4;
               end
               else if (data_in_ready) begin
                  address_lookup_we <= 1;
                  state <= ADDRESS_SCAN_3;
               end
               //else if (cmd_ready) state <= ADDRESS_SCAN_5;
               else state <= ADDRESS_SCAN_2;
            end

            ADDRESS_SCAN_3: begin
               address_lookup_we <= 0;
               if (internal_address <= INTERNAL_ADDRESS_BOUND) begin
                  internal_address <= internal_address + 1;
                  max_internal_address <= internal_address;
               end
               else internal_address_overflow <= 1;
               //jump to state 4
               state <= ADDRESS_SCAN_4;
            end

            ADDRESS_SCAN_4: begin
               data_in <= RESET_ALL_OUTPUTS;
               data_in_valid <= 1;
               cmd_valid <= 1;
               cmd_stop <= 1;
               if (data_in_ready) state <= ADDRESS_SCAN_4; //makes sure that this doesn't transion until I2C FSM gets a command
               //if that doesn't hold us up, this can only have transitioned to a state in the I2C FSM where jumping to the next state is okay
               else state <= ADDRESS_SCAN_5;
            end

            ADDRESS_SCAN_5: begin
               data_in_valid <= 0;
               cmd_valid <= 0;
               cmd_stop <= 0;

               if (cmd_ready) begin

                  if ((max_internal_address < INTERNAL_ADDRESS_BOUND) && (cmd_address < MAX_I2C_ADDRESS) && (~internal_address_overflow)) begin
//then we aren't done yet.
```

```
            cmd_address <= cmd_address+1;
            state <= ADDRESS_SCAN_1;
         end

         else begin
            cmd_address <= 0;
            done <= 1; //this way we don't waste any of our (admittely pretty plentiful) clock cycles
            state <= IDLE;
         end
      end
   end

   ////////////////////////////////////////////////////////////////////////////////////
   //
   //   Write cycle states
   //   these handle the write operations on the the I2C bus
   //
   ////////////////////////////////////////////////////////////////////////////////////

   WRITE_CYCLE_1: begin
      if (cmd_ready) begin
         //for now setting all of this here, if we need to get more complicated I may want
         data_in_valid       <= 1;
         cmd_valid          <= 1;
         cmd_write          <= 1;
         cmd_start          <= 1;
         cmd_stop         <= 1;
         cmd_write_multiple   <= 0;
         cmd_read          <= 0;
         state    <= WRITE_CYCLE_2;
      end
      else state    <= WRITE_CYCLE_1;
   end

   WRITE_CYCLE_2: begin
      cmd_write    <= 0;
      cmd_start    <= 0;
      state <= data_in_ready ? WRITE_CYCLE_3 : WRITE_CYCLE_2;
   end

   WRITE_CYCLE_3: begin
      data_in_valid    <= 0;
      cmd_valid       <= 0;
      cmd_stop      <= 0;

      if (cmd_ready) begin
         done    <= (return_state==IDLE) ? 1 : 0; //if we're returning to the idle state, don't waste a clock cycle before writind done
high
         state    <= return_state;
      end

      else state    <= WRITE_CYCLE_3;
   end

   ////////////////////////////////////////////////////////////////////////////////////
   //
   //   Read cycle states
   //   these handle the read operations on the the I2C bus
   //
   ////////////////////////////////////////////////////////////////////////////////////

   READ_CYCLE_1: begin
      if (cmd_ready) begin
         //for now setting all of this here, if we need to get more complicated I may want
         data_in_valid       <= 0;
         cmd_valid          <= 1;
         cmd_write          <= 0;
         cmd_start          <= 1;
         cmd_stop         <= 1;
         cmd_write_multiple   <= 0;
         cmd_read          <= 1;
         state    <= READ_CYCLE_2;
      end
      else state    <= READ_CYCLE_1;
   end

   READ_CYCLE_2: begin
      cmd_stop      <= 0;
      cmd_read       <= 0;
      data_out_ready   <= 1; //may or may not be necessary

      if (missed_ack) begin //this is bad
         state       <= BUS_ERROR_HANDLER_1; //break out of this now, whatever we get back is garbage
         error_flag <= MISSED_ACK_ON_READ;
      end
```

```
   else if (data_out_valid) begin
         i2c_return_data <= data_out;
         state <= READ_CYCLE_3;
   end

   else state <= READ_CYCLE_2;
end

READ_CYCLE_3: begin       //this state may just be a waste of a clock cycle, but I feel like something will probably come up to make it
needed
   data_out_ready   <= 0;
   state <= return_state;
end

////////////////////////////////////////////////////////////////////////////////////////////
//
//   Neighbor detect output set state
//   this handles writing the detect outputs both low and high
//
////////////////////////////////////////////////////////////////////////////////////////////

NEIGHBOR_DETECT_SET: begin
   if (next_i2c_command == NEIGHBOR_DETECT_HIGH)
            data_in   <= SET_NEIGHBOR_DETECT_OUTPUT_HIGH;
   else
            data_in   <= SET_NEIGHBOR_DETECT_OUTPUT_LOW;
   state <= WRITE_CYCLE_1;
   return_state <= IDLE;
end

////////////////////////////////////////////////////////////////////////////////////////////
//
//   tristate buffer set state
//   this handles telling the buffers whether to be on or off
//
////////////////////////////////////////////////////////////////////////////////////////////

SENSE_RAILS_SET: begin
   if (next_i2c_command == SET_SENSE_RAILS_ON)
         data_in   <= SET_TRISTATE_BUFFERS_ON;
   else
         data_in <= SET_TRISTATE_BUFFERS_OFF;
   state <= WRITE_CYCLE_1;
   return_state <= IDLE;
end

////////////////////////////////////////////////////////////////////////////////////////////
//
//   mux output set state
//   this handles generating the commands to set the mux control bits.
//
////////////////////////////////////////////////////////////////////////////////////////////

MUX_OUTPUTS_SET: begin
   //i2c command is {11,mux1[2:0],mux2[2:0]}, simplify internal commands by always having them the same
   data_in <= {2'b11,next_i2c_command[2:0],next_i2c_command[2:0]};
   state <= WRITE_CYCLE_1;
   return_state <= IDLE;
end

////////////////////////////////////////////////////////////////////////////////////////////
//
//   Read neighbors states
//   these states handle reading the neighbor detect and processing the returned data
//
////////////////////////////////////////////////////////////////////////////////////////////

READ_NEIGHBORS_1: begin
   state <= READ_CYCLE_1;
   return_state <= READ_NEIGHBORS_2;
end

READ_NEIGHBORS_2: begin
   if (i2c_return_data[3:0]==0) begin //this makes sure we're reading something meaningful
      return_data <= {4'b0000,i2c_return_data[7:4]}; //zeros followed by neighbor detect readings
      state <= IDLE;
      done <= 1;
   end
   else begin
      state <= BUS_ERROR_HANDLER_1;
      error_flag <= READ_NEIGHBOR_INVALID;
   end
end

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

```
            //   error handling states
            //   these states handle cases where something goes horribly wrong
            //
            ////////////////////////////////////////////////////////////////////////////////

            BUS_ERROR_HANDLER_1: begin
               return_data <= error_flag;

               //if we miss an ack on the read cycle
               if ((error_flag == MISSED_ACK_ON_READ) && (data_out_valid)) begin //wait until the bus transaction has finished even though it's
garbage
                  bus_error <= 1; //let controller know
                  state <= BUS_ERROR_HANDLER_2;
               end

               else if (error_flag == READ_NEIGHBOR_INVALID) begin
                  bus_error <= 1; //let controller know
                  state <= BUS_ERROR_HANDLER_2;
               end

               else state <= BUS_ERROR_HANDLER_1;
            end

            BUS_ERROR_HANDLER_2: begin
               if (start) begin
                  bus_error <= 0; //reset error flag
                  return_data <= 0;
                  done <=1;
                  state <= IDLE;
               end
               else state <= BUS_ERROR_HANDLER_2;
            end

            default: begin
                   state    <= IDLE;
                   done    <= 1;
               end
         endcase
      end
   end

endmodule
```

7.2.11 I2C Interface Module

```
/*

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,
```

```
    input  wire        cmd_start,
    input  wire        cmd_read,
    input  wire        cmd_write,
    input  wire        cmd_write_multiple,
    input  wire        cmd_stop,
    input  wire        cmd_valid,
    output wire        cmd_ready,

    input  wire [7:0]  data_in,
    input  wire        data_in_valid,
    output wire        data_in_ready,
    input  wire        data_in_last,

    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
```

7.2.12 Mapping Module

```
////////////////////////////////////////////////////////////////////////////////////
//
//     module to handle the XY mapping of the sensor blocks
//     dsheen 11/12/17
//        altered to make the mapping algorithm less dependent on the actual hardware implementation
//        dsheen 12/7/17
//
////////////////////////////////////////////////////////////////////////////////////

module xy_mapping
              #(parameter N = 4,          //sets number of bits used for internal address.
              parameter COMMAND_LEN = 5,    //sets length of the internal command words used for I2C management
              parameter XY_QUADRANT = 4   //sets which corner the XY origin is in, (X and Y are always positive relative to this)
              )
              (
              //global signals
              input wire           clk,
              input wire           reset,
              input wire [N-1:0]   max_internal_address,

              //signals to/from the control FSM
              input wire           start,
              input wire           stop,
              input wire           hold,
              output reg           done,
              output reg    [4:0]  state,

              //i2c handling IO (processed by control FSM before going to I2C commands module)
              output reg                          i2c_send_req,              //this is one clock pulse, needs to be detected immediately by control FSM, but can take a while
to act on it.
              output reg    [N-1:0]               i2c_address_int,
              output reg    [COMMAND_LEN-1: 0]    i2c_command,
              input wire                          i2c_command_processed,      //should pulse high for one clock cycle as soon as the I2C module has finished running the
command and any return data is ready
              input wire    [7:0]                 i2c_data_in,

              //fpga neighbor detect output line
              output reg    fpga_neighbor_detect_out,

              //communications to the XY coordinant to address conversion memory and logic
              output reg                  set_output,         //tells map frame buffer to shoift out newly written data
              output reg                  write_enable, //enable to write to the lookup BRAM
              output reg    [N-1:0]       x_address,          //X position of block to be written
              output reg    [N-1:0]       y_address,          //Y position of block to be written
              output reg    [N+2:0]       data_out,           //data formatted as {valid,rotated[1:0],internal_address}
              input wire    [N+2:0]       data_in,             //readback to make sure things are actually written correctly

              //
              output reg    [N-1:0]       max_x_bound_out,
              output reg    [N-1:0]       max_y_bound_out,

              output reg [15:0] mapping_status
              );

              integer i;
              integer j;
              //I2C commands
              parameter NEIGHBOR_DETECT_HIGH     = 5'b00001; //write neighbor detect output high
              parameter NEIGHBOR_DETECT_LOW      = 5'b00010; //write neighbor detect output low
              parameter SENSE_NEIGHBORS          = 5'b00011; //needs to trigger a read and return suitable data


              //states
              parameter IDLE                            = 0;
              parameter HOLDOFF                         = 1; //used to get 1 cycle delays

              parameter DETECT_NEIGHBORS_1              = 2;
              parameter DETECT_NEIGHBORS_2              = 3;
              parameter DETECT_NEIGHBORS_3              = 4;
              parameter DETECT_NEIGHBORS_4              = 5;
              parameter DETECT_NEIGHBORS_5              = 6;
              parameter DETECT_NEIGHBORS_6              = 7;

              parameter GET_FPGA_LOCATION               = 8;

              parameter COMPUTE_LOCATIONS_START         = 9;
              parameter COMPUTE_LOCATIONS_1             = 10;
              parameter COMPUTE_LOCATIONS_2             = 11;
              parameter COMPUTE_LOCATIONS_3             = 12;
              parameter COMPUTE_LOCATIONS_LEFT          = 13;
              parameter COMPUTE_LOCATIONS_TOP           = 14;
              parameter COMPUTE_LOCATIONS_RIGHT         = 15;
              parameter COMPUTE_LOCATIONS_BOTTOM        = 16;
              parameter COMPUTE_LOCATIONS_4             = 17;
              parameter COMPUTE_LOCATIONS_5             = 18;
              parameter WRITEBACK_1                     = 19;
              parameter WRITEBACK_2                     = 20;
              parameter WRITEBACK_3                     = 21;
              parameter WRITEBACK_4                     = 24;

              reg [N:0]     count; //generally useful register


              //instantiate BRAM for storing neighbor data
              reg                 neighbor_table_we;
              reg    [N-1:0]      neighbor_table_index;
              reg [4*N+3:0]       neighbor_table_data_in;
              wire[4*N+3:0]       neighbor_table_data_out;


              //data format {left_valid, left_address, top_valid, top_address, right_valid, right_address, bottom_valid, bottom_address}

              mybram #(.LOGSIZE(N),.WIDTH(4*N+4))
                     neighbor_table_1(.addr(neighbor_table_index),.clk(clk),.we(neighbor_table_we),.din(neighbor_table_data_in),.dout(neighbor_table_data_out));

              //instantiate a register to store the information about which module the FPGA is next to.
              reg [N+2:0]   fpga_neighbor; //should be stored as {valid,rotated[1:0],internal_address}
```

```
//registers needed for the I2C sweep
reg [N-1:0]   base_address;
reg           loop_flag; //when need
reg [4:0]     return_state;
reg [4:0]     holdoff_return_state;
reg           send_req_next;

//signal breakouts for bits from the returned data
wire left;
wire right;
wire top;
wire bottom;

parameter zero              = 0;
parameter ninety            = 1;
parameter one_eighty = 2;
parameter two_seventy = 3;

//temporary assignments, will need to match actual hardware
assign left   = i2c_data_in[3];
assign top    = i2c_data_in[2];
assign right  = i2c_data_in[1];
assign bottom = i2c_data_in[0];



//instantiate BRAM for XY coordinate mapping store data as {rotated[1:0], X[N:0], Y[N:0]} note these are signed coordinates
reg           location_table_we;
reg [N-1:0]   location_table_index;
reg [2*N+3:0] location_table_data_in;
wire[2*N+3:0] location_table_data_out;

mybram #(.LOGSIZE(N),.WIDTH(2*N+4))
       location_table_1(.addr(location_table_index),.clk(clk),.we(location_table_we),.din(location_table_data_in),.dout(location_table_data_out));

//register to store which data in the location table is valid
reg [(1<<N)-1:0] location_table_data_valid;
//register to store which neighbor tabe entries have already been used in a computation
reg [(1<<N)-1:0] neighbor_table_data_used;
//register used for computing if we're done yet or not
reg [N-1:0] num_valid_entries;

assign fpga_neighbor_address = fpga_neighbor[N-1:0];


//registers for the XY coordinate computation operations
reg    finished;
reg    signed        [N:0] base_x;
reg    signed        [N:0] base_y;
reg    [1:0]  base_rotated;
// wire   right_valid;
// wire   left_valid;
// wire   top_valid;
// wire   bottom_valid;
// wire   [N-1:0] left_address;
// wire   [N-1:0] top_address;
// wire   [N-1:0] right_address;
// wire   [N-1:0] bottom_address;

reg   cached_right_valid;
reg   cached_left_valid;
reg   cached_top_valid;
reg   cached_bottom_valid;
reg   [N-1:0] cached_left_address;
reg   [N-1:0] cached_top_address;
reg   [N-1:0] cached_right_address;
reg   [N-1:0] cached_bottom_address;

// assign        left_valid           = neighbor_table_data_out[4*N+3];
// assign        left_address         = neighbor_table_data_out[4*N+2:3*N+3];
// assign        top_valid            = neighbor_table_data_out[3*N+2];
// assign        top_address          = neighbor_table_data_out[3*N+1:2*N+2];
// assign        right_valid          = neighbor_table_data_out[2*N+1];
// assign        right_address        = neighbor_table_data_out[2*N:N+1];
// assign        bottom_valid         = neighbor_table_data_out[N];
// assign        bottom_address         = neighbor_table_data_out[N-1:0];

reg [N-1:0]   outer_loop_count;
reg           valid_locations_count;

//stuff for the final sweep to determine
reg signed [N:0] min_signed_x;
reg signed [N:0] min_signed_y;
reg signed [N:0] max_signed_x;
reg signed [N:0] max_signed_y;

wire signed [N:0] current_x;
wire signed [N:0] current_y;

assign current_x = location_table_data_out[2*N+1:N+1];
assign current_y = location_table_data_out[N:0];

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

              state  <= IDLE;
              done   <= 1;

              i2c_send_req         <= 0;
              i2c_address_int      <= 0;
              i2c_command          <= NEIGHBOR_DETECT_LOW;

              fpga_neighbor_detect_out    <= 0;

              set_output           <= 0;
              write_enable         <= 0;
              x_address            <= 0;
              y_address            <= 0;
```

```
        data_out            <= 0;

        neighbor_table_we           <= 0;
        neighbor_table_index        <= 0;
        neighbor_table_data_in      <= 0;

        location_table_we           <= 0;
        location_table_index        <= 0;
        location_table_data_in      <= 0;

        fpga_neighbor               <= 0;

        base_address  <= 0;
        count         <= 0;
        loop_flag     <= 0;
        send_req_next <= 0;
        finished      <= 0;

        location_table_data_valid   <= 0;
        neighbor_table_data_used    <= 0;
        num_valid_entries           <= 0;

        mapping_status <= 0;

end

else if (hold) state <= state; //do nothing if hold is asserted

else begin
        case(state)

                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                //
                //      IDLE
                //
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                IDLE: begin
                        if (start) begin
                                done <= 0; //we're busy now
                                //set address registers to appropriate values to start with before transitioning
                                neighbor_table_index        <= 0;
                                base_address  <= 0;

                                //set regs to erase first line in neighbor table
                                neighbor_table_data_in      <= 0;
                                neighbor_table_we           <= 1;
                                //jump to first state
                                state <= DETECT_NEIGHBORS_1;
                        end
                        else state <= state;
                end

                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                //
                //      HOLDOFF state
                //
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                HOLDOFF: begin
                        state <= holdoff_return_state;
                        i2c_send_req  <= send_req_next; //so we can use this during command sends
                        send_req_next <= 0;
                end


                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                //
                //      DETECT_NEIGHBORS_1
                //      start by erasing the existing neighbor table
                //
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                DETECT_NEIGHBORS_1: begin
                        if (neighbor_table_index < max_internal_address) begin
                                //we and data were set appropriately at entry, so don't need to change them
                                //just increment index
                                neighbor_table_index <= neighbor_table_index + 1;
                                state <= DETECT_NEIGHBORS_1;
                        end
                        else begin
                                neighbor_table_index <= 0;  //reset for scanning
                                neighbor_table_we           <= 0;
                                state <= DETECT_NEIGHBORS_2;
                                return_state <= DETECT_NEIGHBORS_4;
                        end
                end

                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                //
                //      DETECT_NEIGHBORS_2
                //      send the I2C command to set the neighbor detect line high
                //
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                DETECT_NEIGHBORS_2: begin
                        i2c_address_int      <= base_address;
                        i2c_command   <= NEIGHBOR_DETECT_HIGH;
                        i2c_send_req  <= 1;
                        state                <= DETECT_NEIGHBORS_3;
                end

                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                //
                //      DETECT_NEIGHBORS_3
                //      hold until I2C transaction has completed then send sense request or just do it without waiting if last state
                //      wasn't DETECT_NEIGHBORS_2
                //
                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                DETECT_NEIGHBORS_3: begin
```

```
        if (i2c_command_processed | loop_flag) begin
                i2c_address_int      <= neighbor_table_index; //point it at the first block we want to scan
                i2c_command   <= SENSE_NEIGHBORS;

                loop_flag            <= 0;
                holdoff_return_state <= return_state;
                send_req_next <= 1;
                state <= HOLDOFF;
        end

        else begin
                i2c_send_req  <= 0;
                state         <= DETECT_NEIGHBORS_3;
        end
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//      DETECT_NEIGHBORS_4
//      wait for returned data from the I2C bus, and store if appropriate
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
DETECT_NEIGHBORS_4: begin
        i2c_send_req  <= 0;
        if (i2c_command_processed) begin
                //store any detected adjacencies to BRAM
                neighbor_table_we <= (|i2c_data_in[3:0]); //if it's next to anything then we have data to store.
                count <= (|i2c_data_in[3:0]) ? count+1 : count;
                //we've already been pointing at the appropriate table index for a while, so output is the same line we want to write to
                neighbor_table_data_in[N:0]         <= bottom    ? {1'b1,base_address} : neighbor_table_data_out[N:0];
                neighbor_table_data_in[2*N+1:N+1]   <= right     ? {1'b1,base_address} : neighbor_table_data_out[2*N+1:N+1];
                neighbor_table_data_in[3*N+2:2*N+2] <= top       ? {1'b1,base_address} : neighbor_table_data_out[3*N+2:2*N+2];
                neighbor_table_data_in[4*N+3:3*N+3] <= left      ? {1'b1,base_address} : neighbor_table_data_out[4*N+3:3*N+3];

                state <= DETECT_NEIGHBORS_5;
        end
        else state <= DETECT_NEIGHBORS_4;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//      DETECT_NEIGHBORS_5
//      determine loopback in I2C scan. Are we done, or do we need to scan more blocks?
//      this could really be combined with DETECT_NEIGHBORS_4
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
DETECT_NEIGHBORS_5: begin
        neighbor_table_we <= 0;

        //if count is 4 we're definitely done with the inner sweep, so increment which block has the neighbor output high
        if (((count == 4)|(neighbor_table_index >= max_internal_address)) && (base_address < max_internal_address)) begin
                count <= 0;

                i2c_address_int      <= base_address;
                i2c_command   <= NEIGHBOR_DETECT_LOW;
                i2c_send_req  <= 1;

                state <= DETECT_NEIGHBORS_6;
        end

        //we aren't done with the inner loop
        else if (neighbor_table_index < max_internal_address) begin
                neighbor_table_index <= neighbor_table_index + 1;
                loop_flag <= 1;
                state <= DETECT_NEIGHBORS_3;
        end

        //else entire sweep finished go find where the FPGA is plugged in next
        else begin
                count <= 0;

                i2c_address_int      <= base_address;
                i2c_command   <= NEIGHBOR_DETECT_LOW;
                i2c_send_req  <= 1;
                neighbor_table_index <= 0;
                base_address <= 0;
                fpga_neighbor_detect_out    <= 1;
                fpga_neighbor                           <= 0;


                return_state <= GET_FPGA_LOCATION;
                holdoff_return_state <= DETECT_NEIGHBORS_3;
                send_req_next <= 0;
                state <= HOLDOFF;
        end
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//      DETECT_NEIGHBORS_6
//      makes sure the NEIGHBOR_DETECT_LOW command processes before returning to DETECT_NEIGHBORS_2
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
DETECT_NEIGHBORS_6: begin
        i2c_send_req  <= 0;
        if (i2c_command_processed) begin
                base_address <= base_address + 1; //actually increment this here
                neighbor_table_index <= 0;                                              //reset inner sweep to zero
                state <= DETECT_NEIGHBORS_2;
        end
        else state <= DETECT_NEIGHBORS_6;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//      GET_FPGA_LOCATION
//      finds the FPGA connection point. This can fail if the FPGA isn't connected to anything, but in that case
//      the I2C commands module will generate an error flag prior to us arriving at this state.
```

```
//     also initiallizes registers for the mapping computation
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
GET_FPGA_LOCATION: begin
       mapping_status[6] <= 1;
       i2c_send_req  <= 0;
       if(i2c_command_processed) begin
              if (|i2c_data_in[3:0]) begin
                     mapping_status[7] <= 1;
                     //fpga will always be assigned on top edge
                     fpga_neighbor[N+2]   <= 1;
                     fpga_neighbor[N+1:N] <= top ? zero : (left ? ninety : (bottom ? one_eighty : two_seventy)); //fixed for all rotation
angles
                     fpga_neighbor[N-1:0] <= neighbor_table_index;

                     neighbor_table_index <= 0;
                     fpga_neighbor_detect_out <= 0;

                     //we haven't used any of the neighbor location table entries, so set all used flags to zero
                     neighbor_table_data_used  <= 0;
                     //no valid entries in location table yet either, so reset these
                     location_table_data_valid <= 0;

                     state <= COMPUTE_LOCATIONS_START;
              end
              else begin
                     neighbor_table_index <= neighbor_table_index + 1;
                     loop_flag <= 1;
                     state <= DETECT_NEIGHBORS_3;
              end
       end
       else state <= GET_FPGA_LOCATION;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_START_1
//     stores the FPGA neighbor's position, then kicks off the hunt for neighboring positions
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

COMPUTE_LOCATIONS_START: begin

       location_table_index                     <= fpga_neighbor_address;   //address in location table
       location_table_data_in[2*N+3:2*N+2]      <= fpga_neighbor[N+1:N];    //set rotation bits
       location_table_data_in[2*N+1:0]    <= 0;                     //set location (x,y) = 0,0

       location_table_we <= 1;                                      //write all of this

       location_table_data_valid[fpga_neighbor_address] <= 1;       //this entry is now valid

       //set base address used by loop
       base_address  <= fpga_neighbor_address;

       //set the correct read address for the neighbor detect going into the loop
       neighbor_table_index <= fpga_neighbor_address;

       holdoff_return_state <= (max_internal_address > 0) ? COMPUTE_LOCATIONS_1 : COMPUTE_LOCATIONS_4;
       //if there's only one block, skip everything, we're already done here
       //otherwise, go through first cycle of loop
       state <= HOLDOFF;

end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_1
//     holdoff for read of the neighbor table BRAM
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

COMPUTE_LOCATIONS_1: begin
       location_table_we <= 0;
       state <= COMPUTE_LOCATIONS_2;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_2
//     part of the calling sequence for COMPUTE_LOCATIONS_3
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

COMPUTE_LOCATIONS_2: begin

       base_rotated        <= location_table_data_out[2*N+3:2*N+2];
       base_x              <= location_table_data_out[2*N+1:N+1];
       base_y              <= location_table_data_out[N:0];
       mapping_status[9:8]  <= location_table_data_out[2*N+3:2*N+2];

       cached_left_valid          <= neighbor_table_data_out[4*N+3];
       cached_left_address        <= neighbor_table_data_out[4*N+2:3*N+3];
       cached_top_valid           <= neighbor_table_data_out[3*N+2];
       cached_top_address         <= neighbor_table_data_out[3*N+1:2*N+2];
       cached_right_valid         <= neighbor_table_data_out[2*N+1];
       cached_right_address       <= neighbor_table_data_out[2*N:N+1];
       cached_bottom_valid        <= neighbor_table_data_out[N];
       cached_bottom_address      <= neighbor_table_data_out[N-1:0];

       state <= COMPUTE_LOCATIONS_3;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_3
//     handles transitions for state computation.
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
COMPUTE_LOCATIONS_3: begin
       mapping_status[5] <= 1;
       location_table_we <= 0;

       //don't waste time on blocks hat aren't there or that we already know the location of
       if (cached_left_valid && (~location_table_data_valid[cached_left_address])) begin
              location_table_index <= cached_left_address;
              neighbor_table_index <= cached_left_address; //we do need this for one operation in the next state
              holdoff_return_state <= COMPUTE_LOCATIONS_LEFT;
              state <= HOLDOFF;
       end

       else if (cached_top_valid && (~location_table_data_valid[cached_top_address])) begin
              location_table_index <= cached_top_address;
              neighbor_table_index <= cached_top_address;
              holdoff_return_state <= COMPUTE_LOCATIONS_TOP;
              state <= HOLDOFF;
       end

       else if (cached_right_valid && (~location_table_data_valid[cached_right_address])) begin
              location_table_index <= cached_right_address;
              neighbor_table_index <= cached_right_address;
              holdoff_return_state <= COMPUTE_LOCATIONS_RIGHT;
              state <= HOLDOFF;
       end

       else if (cached_bottom_valid && (~location_table_data_valid[cached_bottom_address])) begin
              location_table_index <= cached_bottom_address;
              neighbor_table_index <= cached_bottom_address;
              holdoff_return_state <= COMPUTE_LOCATIONS_BOTTOM;
              state <= HOLDOFF;
       end

       else begin
              neighbor_table_data_used[base_address]    <= 1;
              num_valid_entries <= 0; //reset this before using it in computation during next stage
              location_table_index <= 0; //doing this here saves a cycle on exit
              state  <= COMPUTE_LOCATIONS_4;

              mapping_status[0] <= 1;
       end

end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_LEFT
//     computes location of block to the left
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

COMPUTE_LOCATIONS_LEFT: begin
              //left/right
              mapping_status[1] <= 1;

              if ((base_rotated == zero) | (base_rotated == one_eighty)) begin
                     location_table_data_in[N:0]              <= base_y;    //y position doesn't change
                     location_table_data_in[2*N+1:N+1]        <= (base_rotated == one_eighty) ? (base_x + 1) : (base_x - 1);
//figure out x position
              end
              //up/down
              else /*if ((base_rotated == ninety) | (base_rotated == two_seventy))*/ begin
                     location_table_data_in[N:0]               <= (base_rotated == ninety) ? (base_y + 1 ) : (base_y - 1);
                     location_table_data_in[2*N+1:N+1]         <= base_x;  //figure out x position
              end

              //solve the rotation

              //left edge
              if (neighbor_table_data_out[4*N+2:3*N+3] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + one_eighty;
              //top edge
              else if (neighbor_table_data_out[3*N+1:2*N+2] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + ninety;
              //right edge
              else if (neighbor_table_data_out[2*N:N+1] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + zero;
              //bottom edge
              else if (neighbor_table_data_out[N-1:0] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + two_seventy;

     location_table_we    <= 1;     //write the data back

     location_table_data_valid[cached_left_address]   <= 1;  //this entry will now be valid
     cached_left_valid    <= 0;     //keeps it from returning to this state

     state <= COMPUTE_LOCATIONS_3;
end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
//     COMPUTE_LOCATIONS_TOP
//     computes location of block above
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

COMPUTE_LOCATIONS_TOP: begin

       mapping_status[2] <= 1;

              //up/down
              if ((base_rotated == zero) | (base_rotated == one_eighty)) begin
       location_table_data_in[N:0]             <= (base_rotated == zero) ? (base_y + 1 ) : (base_y - 1);
                     location_table_data_in[2*N+1:N+1]         <= base_x;  //figure out x position//figure out x position
              end
              //left/right
              else /*if ((base_rotated == ninety) | (base_rotated == two_seventy))*/ begin
                  location_table_data_in[N:0]           <= base_y;    //y position doesn't change
                     location_table_data_in[2*N+1:N+1]         <= (base_rotated == ninety) ? (base_x + 1) : (base_x - 1);
              end
```

```
                        //solve the rotation

                        //left edge
                        if (neighbor_table_data_out[4*N+2:3*N+3] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + two_seventy;
                        //top edge
                        else if (neighbor_table_data_out[3*N+1:2*N+2] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated +
one_eighty;
                        //right edge
                        else if (neighbor_table_data_out[2*N:N+1] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + ninety;
                        //bottom edge
                        else if (neighbor_table_data_out[N-1:0] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + zero;

      location_table_we    <= 1;     //write the data back

      location_table_data_valid[cached_top_address]   <= 1;  //this entry will now be valid
      cached_top_valid    <= 0;     //keeps it from returning to this state

      state <= COMPUTE_LOCATIONS_3;


        // location_table_data_in[N:0]        <= base_rotated ? (base_y-1) : (base_y+1);       //y position
        // location_table_data_in[2*N+1:N+1]  <= base_x;                    //figure out x position
        // location_table_data_in[2*N+2]      <= (neighbor_table_data_out[3*N+1:2*N+2] == base_address) ? ~base_rotated : base_rotated;
//figure out rotation

        // location_table_we    <= 1;     //write the data back

        // location_table_data_valid[cached_top_address]   <= 1;  //this entry will now be valid
        // cached_top_valid  <= 0;

        // state <= COMPUTE_LOCATIONS_3;
    end

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //
    //     COMPUTE_LOCATIONS_RIGHT
    //     computes location of block to the right
    //
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    COMPUTE_LOCATIONS_RIGHT: begin
                mapping_status[3] <= 1;
                //left/right
                if ((base_rotated == zero) | (base_rotated == one_eighty)) begin
                       location_table_data_in[N:0]              <= base_y;    //y position doesn't change
                       location_table_data_in[2*N+1:N+1]        <= (base_rotated == zero) ? (base_x + 1) : (base_x - 1);    //figure out x
position
                end
                //up/down
                else /*if ((base_rotated == ninety) | (base_rotated == two_seventy))*/ begin
                       location_table_data_in[N:0]               <= (base_rotated == two_seventy) ? (base_y + 1 ) : (base_y - 1);
                       location_table_data_in[2*N+1:N+1]         <= base_x;  //figure out x position
                end

                //solve the rotation

                //left edge
                if (neighbor_table_data_out[4*N+2:3*N+3] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + zero;
                //top edge
                else if (neighbor_table_data_out[3*N+1:2*N+2] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated +
two_seventy;
                //right edge
                else if (neighbor_table_data_out[2*N:N+1] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + one_eighty;
                //bottom edge
                else if (neighbor_table_data_out[N-1:0] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + ninety;

      location_table_we    <= 1;     //write the data back

      location_table_data_valid[cached_right_address]   <= 1;  //this entry will now be valid
      cached_right_valid    <= 0;     //keeps it from returning to this state

      state <= COMPUTE_LOCATIONS_3;

        // location_table_data_in[N:0]              <= base_y;    //y position
        // location_table_data_in[2*N+1:N+1]        <= base_rotated ? (base_x-1) : (base_x+1);                    //figure out x position
        // location_table_data_in[2*N+2]            <= (neighbor_table_data_out[2*N:N+1] == base_address) ? ~base_rotated : base_rotated;
//figure out rotation

        // location_table_we    <= 1;     //write the data back

        // location_table_data_valid[cached_right_address]   <= 1;  //this entry will now be valid
        // cached_right_valid   <= 0;

        // state <= COMPUTE_LOCATIONS_3;
    end

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //
    //     COMPUTE_LOCATIONS_BOTTOM
    //     computes location of block below
    //
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    COMPUTE_LOCATIONS_BOTTOM: begin
          mapping_status[4] <= 1;
      //up/down
        if ((base_rotated == zero) | (base_rotated == one_eighty)) begin
          location_table_data_in[N:0]            <= (base_rotated == one_eighty) ? (base_y + 1 ) : (base_y - 1);
                       location_table_data_in[2*N+1:N+1]        <= base_x;  //figure out x position//figure out x position
                end
                //left/right
                else /*if ((base_rotated == ninety) | (base_rotated == two_seventy))*/ begin
                    location_table_data_in[N:0]          <= base_y;    //y position doesn't change
                       location_table_data_in[2*N+1:N+1]        <= (base_rotated == two_seventy) ? (base_x + 1) : (base_x - 1);
                end

                //solve the rotation
```

```
                                        //left edge
                                        if (neighbor_table_data_out[4*N+2:3*N+3] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + ninety;
                                        //top edge
                                        else if (neighbor_table_data_out[3*N+1:2*N+2] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + zero;
                                        //right edge
                                        else if (neighbor_table_data_out[2*N:N+1] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated +
two_seventy;
                                        //bottom edge
                                        else if (neighbor_table_data_out[N-1:0] == base_address) location_table_data_in[2*N+3:2*N+2] <= base_rotated + one_eighty;

                                location_table_we    <= 1;     //write the data back

                                location_table_data_valid[cached_bottom_address]   <= 1;  //this entry will now be valid
                                cached_bottom_valid    <= 0;     //keeps it from returning to this state

                                state <= COMPUTE_LOCATIONS_3;


                                  // location_table_data_in[N:0]        <= base_rotated ? (base_y-1) : (base_y+1);       //y position
                                  // location_table_data_in[2*N+1:N+1]  <= base_x;                    //figure out x position
                                  // location_table_data_in[2*N+2]      <= (neighbor_table_data_out[3*N+1:2*N+2] == base_address) ? ~base_rotated : base_rotated;
//figure out rotation

                                  // location_table_we    <= 1;     //write the data back

                                  // location_table_data_valid[cached_top_address]   <= 1;  //this entry will now be valid
                                  // cached_top_valid  <= 0;

                                  // state <= COMPUTE_LOCATIONS_3;
                                  // location_table_data_in[N:0]              <= base_rotated ? (base_y+1) : (base_y-1);       //y position
                                  // location_table_data_in[2*N+1:N+1]        <= base_x;                    //figure out x position
                                  // location_table_data_in[2*N+2]            <= (neighbor_table_data_out[N-1:0] == base_address) ? ~base_rotated : base_rotated;
//figure out rotation

                                  // location_table_we     <= 1;     //write the data back

                                  // location_table_data_valid[cached_bottom_address]   <= 1;  //this entry will now be valid
                                  // cached_bottom_valid      <= 0;

                                  // state <= COMPUTE_LOCATIONS_3;
                                end

                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                                //
                                //     COMPUTE_LOCATIONS_4
                                //     after compute_locations 3. begins handling for first layer of looping
                                //
                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                                COMPUTE_LOCATIONS_4: begin
                                       location_table_we <= 0;
                                       for (i=0; i<2**N; i = i + 1) begin
                                              num_valid_entries = num_valid_entries + location_table_data_valid[i];
                                       end
                                       j = 0; //here for compliance
                                       if (num_valid_entries > max_internal_address) begin
                                              location_table_index <= 0;
                                              state                <= COMPUTE_LOCATIONS_5;
                                              min_signed_x         <= 2**(N-1) - 1;
                                              min_signed_y         <= 2**(N-1) - 1;
                                              max_signed_x         <= -2**(N-1);
                                              max_signed_y         <= -2**(N-1);
                                              count                <= 0;
                                       end
                                       else begin
                                              repeat (2**N)
                                              begin
                                                     if (location_table_data_valid[j] & ~neighbor_table_data_used[j]) begin
                                                            base_address       <=  j;
                                                            neighbor_table_index <= j;
                                                            location_table_index <= j;
                                                            state <= COMPUTE_LOCATIONS_1;
                                                     end
                                                     j = j + 1;
                                              end
                                       end
                                end

                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                                //
                                //     COMPUTE_LOCATIONS_5
                                //     go through and figure out what the minimum values of X and Y are
                                //
                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                                COMPUTE_LOCATIONS_5: begin
                                       min_signed_x <= (min_signed_x > current_x) ? current_x : min_signed_x;
                                       min_signed_y <= (min_signed_y > current_y) ? current_y : min_signed_y;
                                       max_signed_x <= (max_signed_x < current_x) ? current_x : max_signed_x;
                                       max_signed_y <= (max_signed_y < current_y) ? current_y : max_signed_y;

                                       location_table_index <= (count < max_internal_address) ? location_table_index + 1 : 0;
                                       count <= (count > max_internal_address) ? 0 : count+1; //greater than should be correct, I want this to break at
max_internal_address + 1 for BRAM timing
                                       state <= (count > max_internal_address) ? WRITEBACK_1 : COMPUTE_LOCATIONS_5;

                                       x_address <= 0;
                                       y_address <= 0;
                                end

                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                                //
                                //     WRITEBACK_1
                                //     go through and erase the coordinate to address lookup BRAM
                                //     since we have X and Y bounds we only need to erase the range which attached blocks can cover
                                //
                                ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
                        WRITEBACK_1: begin

                                x_address <= (x_address < max_internal_address) ? x_address + 1 : 0;
                                y_address <= (x_address < max_internal_address) ? y_address: ((y_address < max_internal_address) ? y_address + 1 : 0);
                                state <= ((y_address == max_internal_address) && (x_address == max_internal_address)) ? WRITEBACK_2 : WRITEBACK_1;
                                write_enable <= 1;
                                data_out <= 0;
                        end

                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                        //
                        //      WRITEBACK_2
                        //      writes data to the address lookup BRAM (also does transforms on addresses)
                        //
                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                        WRITEBACK_2: begin

                                write_enable         <= (location_table_index > 0) ? 1 : 0; //this should yield the correct delay
                                location_table_index <= (location_table_index > max_internal_address) ? 0 : location_table_index + 1;
                                state                <= (location_table_index > max_internal_address) ? WRITEBACK_3 : WRITEBACK_2;

                                max_x_bound_out <= max_signed_x - min_signed_x;
                                max_y_bound_out <= max_signed_y - min_signed_y;


                                // //if we want to place origin at bottom left corner
                                // if (XY_QUADRANT == 1) begin
                                //        x_address <= current_x - min_signed_x;    //always positive
                                //        y_address <= current_y - min_signed_y;    //always positive
                                // end
                                // //if we want to place origin at bottom right corner
                                // if (XY_QUADRANT == 2) begin
                                //        x_address <=   max_signed_x /*- min_signed_x*/ - current_x;       //initially negative, then made positive again
                                //        y_address <=   current_y - min_signed_y;        //always positive
                                // end

                                // //if we want to place origin at top right corner
                                // if (XY_QUADRANT == 3) begin
                                //        x_address <=  max_signed_x /*- min_signed_x*/ - current_x;       //initially negative, then made positive again
                                //        y_address <=  max_signed_y /*- min_signed_y*/ - current_y;       //initially negative, then made positive again
                                // end

                                // //if we want to place origin at top left corner
                                // if (XY_QUADRANT == 4) begin
                                //        x_address <=    current_x - min_signed_x;        //always positive
                                //        y_address <=    max_signed_y /*- min_signed_y*/ - current_y;       //initially negative, then made positive again
                                // end

                                x_address <= current_x - min_signed_x;    //always positive
                                y_address <= current_y - min_signed_y;    //always positive

                                data_out[N+2]        <= 1'b1;
                                data_out[N+1:N]      <= location_table_data_out[2*N+3:2*N+2];
                                data_out[N-1:0]      <= location_table_index-1;
                        end

                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                        //
                        //      WRITEBACK_3
                        //      shifts BRAM output frame, deasserts write.
                        //
                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                        WRITEBACK_3: begin
                                write_enable  <= 0;
                                set_output    <= 1;
                                state         <= WRITEBACK_4;
                        end

                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                        //
                        //      WRITEBACK_4
                        //      deasserts set, asserts DONE, returns to IDLE
                        //
                        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

                        WRITEBACK_4: begin
                                set_output    <= 0;
                                done          <= 1;
                                state         <= IDLE;
                        end

                        default: begin
                                state <= IDLE;
                                done  <= 1;
                        end
                endcase
        end
    end
endmodule
```

7.2.13 Coordinate Translation Module (theoretically correct version)

```
////////////////////////////////////////////////////////////////////////////////
//
//      module to translate XY coordinates to physical addresses
//      dsheen 11/19/17
//       changed 12/7/19 to allow universal rotations
//       this should be correct, doesn't work with hardware for some reason. Use V2, which
```

```
//      has a sort of kludgy solution that shouldn't work but does
//
//////////////////////////////////////////////////////////////////////////////////

module coordinate_translation #(parameter N = 4, XY_QUADRANT = 4)
      (

      //global signals
      input wire             clk,
      input wire             reset,
      input wire [N-1:0]      max_internal_address,

      //communication with the mapping module
      input wire                  set_output,         //tells map frame buffer to shoift out newly written data
        input wire                  write_enable,       //enable to write to the lookup BRAM
        input wire    [N-1:0]       x_address,          //X position of block to be written
        input wire    [N-1:0]       y_address,          //Y position of block to be written
        inout wire    [N+2:0]       data_in,            //data formatted as {valid,rotated,internal_address}
        output wire   [N+2:0]       data_out,           //return  data for mapping module in case anything is messed up.

        input wire    [N-1:0]       max_x_bound_in,      //these shall be forwarded on to the user along with the map data
        input wire    [N-1:0]       max_y_bound_in,      //these shall be forwarded on to the user along with the map data


        //user facing communications
        input wire [N+1:0]   x_coordinate,
        input wire [N+1:0]   y_coordinate,

        output reg [N+1:0]  max_x_bound_out,
        output reg [N+1:0]  max_y_bound_out,

        output wire       valid,         //assert this if sensor is in fact present at requested location and data is ready
        output reg          error,          //we'll assert this if the xy request from user is not within bounds
        output wire [N+3:0] address         //address to read sensor data
      );


   //instantiate frame BRAMS

   //first frame
   wire          frame1_we;
   wire [N-1:0]   frame1_x;
   wire [N-1:0]   frame1_y;
   wire [N+2:0]   frame1_data_in;
   wire [N+2:0]   frame1_data_out;

   mybram #(.LOGSIZE(2*N),.WIDTH(N+3))
                address_data_1(.addr({frame1_x,frame1_y}),.clk(clk),.we(frame1_we),.din(frame1_data_in),.dout(frame1_data_out));


   //second frame
   wire          frame2_we;
   wire [N-1:0]   frame2_x;
   wire [N-1:0]   frame2_y;
   wire [N+2:0]   frame2_data_in;
   wire [N+2:0]   frame2_data_out;

   mybram #(.LOGSIZE(2*N),.WIDTH(N+3))
               address_data_2(.addr({frame2_x,frame2_y}),.clk(clk),.we(frame2_we),.din(frame2_data_in),.dout(frame2_data_out));


    // frame output state control
    reg output_frame;

    always @(posedge clk) begin
       if (reset) begin
          output_frame    <= 0;
           max_x_bound_out   <= 0;
           max_y_bound_out   <= 0;
           error          <= 0;
          end

         else if (set_output) begin
            output_frame    <= ~output_frame;
            max_x_bound_out   <= ((max_x_bound_in + 1) << 2) -1;
            max_y_bound_out   <= ((max_y_bound_in + 1) << 2) -1;
            error          <= 0;
          end

          else error <= (x_coordinate > max_x_bound_out) || (y_coordinate > max_y_bound_out);
    end


    wire [N+1:0] x_coordinate_quadrant;
    wire [N+1:0] y_coordinate_quadrant;
    //if output frame ==0, make frame 1 outputed and frame 2 written to, otherwise do the reverse
```

```
    assign frame1_we    = output_frame ? write_enable : 0;
    assign frame2_we    = output_frame ? 0 : write_enable;

    assign frame1_x      = output_frame ? x_address : x_coordinate_quadrant[N+1:2];
    assign frame1_y      = output_frame ? y_address : y_coordinate_quadrant[N+1:2];
    assign frame2_x      = output_frame ? x_coordinate_quadrant[N+1:2] : x_address;
    assign frame2_y      = output_frame ? y_coordinate_quadrant[N+1:2] : y_address;

    assign frame1_data_in   = output_frame ? data_in : 0;
    assign frame2_data_in   = output_frame ? 0 : data_in;

    assign data_out    = output_frame ? frame1_data_out : frame2_data_out;


    wire [1:0]   rotation;
    wire [1:0]   rotation_corrected; //we need to be able to correct the rotation for the fact that it will be mirrored in different coordinate
systems
    wire [3:0]   xy;

    assign rotation = output_frame ? frame2_data_out[N+1:N] : frame1_data_out[N+1:N];

    //need to transform this

    assign xy[3:2]    = x_coordinate_quadrant[1:0];
    assign xy[1:0]  = y_coordinate_quadrant[1:0]; //because Q4 coordinates on the boards
    assign rotation_corrected = rotation;

    if (XY_QUADRANT == 1) begin
       assign x_coordinate_quadrant = x_coordinate;
       assign y_coordinate_quadrant  = y_coordinate;
       //assign rotation_corrected = rotation;
    end

    else if (XY_QUADRANT == 2) begin
       assign x_coordinate_quadrant = max_x_bound_out - x_coordinate;
       assign y_coordinate_quadrant = y_coordinate;
       //flip rotation angles
       /*reason this works
          zero       = 00
          ninety       = 01
          one_eighty  = 10
         two_seventy = 11
      */
       //assign rotation_corrected[1] = ^rotation;
       //assign rotation_corrected[0] = rotation[0];
    end

    if (XY_QUADRANT == 3) begin
       assign x_coordinate_quadrant = max_x_bound_out - x_coordinate;
       assign y_coordinate_quadrant = max_y_bound_out - y_coordinate;
       //assign rotation_corrected = rotation;
    end

    if (XY_QUADRANT == 4) begin
       assign x_coordinate_quadrant = x_coordinate;
       assign y_coordinate_quadrant = max_y_bound_out - y_coordinate;

       //assign rotation_corrected[1] = ^rotation;
       //assign rotation_corrected[0] = rotation[0];
    end

    //note on the above. the way I implemented this in the end was dumb. In hindsight, I sould have had the entirety of the address mapping
    //handled at once, since then the coordinate system would have been very easy to just nicely transform with a few matrix operations. the only
    //justification for this arrangement is it saves a lot of BRAM space (a factor of 16) when a large number of blocks are allowed.

    wire [3:0]   sensor_addr_zero;
    wire [3:0]   sensor_addr_ninety;
    wire [3:0]  sensor_addr_one_eighty;
    wire [3:0]    sensor_addr_two_seventy;

    //instantiate rotation lookup rom

    xy_to_sensor_addr_rom lookup1(.clk(clk),
                        .xy(xy),
                        .sensor_addr_zero(sensor_addr_zero),
                        .sensor_addr_ninety(sensor_addr_ninety),
                        .sensor_addr_one_eighty(sensor_addr_one_eighty),
                        .sensor_addr_two_seventy(sensor_addr_two_seventy)
                        );

     parameter zero          = 0;
      parameter ninety      = 1;
    parameter one_eighty    = 2;
    parameter two_seventy    = 3;
```

```
    assign address [3:0]   = (rotation_corrected == zero) ? sensor_addr_zero : ((rotation_corrected == ninety) ? sensor_addr_ninety :
((rotation_corrected == one_eighty) ? sensor_addr_one_eighty : sensor_addr_two_seventy));
    assign address [N+3:4]    = output_frame ? frame2_data_out[N-1:0] : frame1_data_out[N-1:0];
    assign valid         = output_frame ? frame2_data_out[N+2] : frame1_data_out[N+2];


endmodule


////////////////////////////////////////////////////////////////////////////////
///
/// this module translates the lower bits of the XY position into two possible
/// sensor numbers dependent on the block rotation
///
/// dsheen 11/2/2017
///
////////////////////////////////////////////////////////////////////////////////

module xy_to_sensor_addr_rom (
                     input wire        clk,
                     input wire [3:0] xy,
                     output reg [3:0] sensor_addr_zero,
                     output reg [3:0] sensor_addr_ninety,
                     output reg [3:0] sensor_addr_one_eighty,
                     output reg [3:0] sensor_addr_two_seventy
                     );

   //rom data may be different depending on how we do things.
   always @(posedge clk) begin
      case(xy)
         4'b0011: begin
            sensor_addr_zero       <= 4'b0000;
            sensor_addr_ninety      <= 4'b1100;
            sensor_addr_one_eighty   <= 4'b1111;
            sensor_addr_two_seventy   <= 4'b0011;
            end
         4'b0111: begin
            sensor_addr_zero       <= 4'b0001;
            sensor_addr_ninety      <= 4'b1000;
            sensor_addr_one_eighty   <= 4'b1110;
            sensor_addr_two_seventy   <= 4'b0111;
            end
         4'b1011: begin
            sensor_addr_zero       <= 4'b0010;
            sensor_addr_ninety      <= 4'b0100;
            sensor_addr_one_eighty   <= 4'b1101;
            sensor_addr_two_seventy   <= 4'b1011;
            end
         4'b1111: begin
            sensor_addr_zero       <= 4'b0011;
            sensor_addr_ninety      <= 4'b0000;
            sensor_addr_one_eighty   <= 4'b1100;
            sensor_addr_two_seventy   <= 4'b1111;
            end
         4'b0010: begin
            sensor_addr_zero       <= 4'b0100;
            sensor_addr_ninety      <= 4'b1101;
            sensor_addr_one_eighty   <= 4'b1011;
            sensor_addr_two_seventy   <= 4'b0010;
            end
         4'b0110: begin
            sensor_addr_zero       <= 4'b0101;
            sensor_addr_ninety      <= 4'b1001;
            sensor_addr_one_eighty   <= 4'b1010;
            sensor_addr_two_seventy   <= 4'b0110;
            end
         4'b1010: begin
            sensor_addr_zero       <= 4'b0110;
            sensor_addr_ninety      <= 4'b0101;
            sensor_addr_one_eighty   <= 4'b1001;
            sensor_addr_two_seventy   <= 4'b1010;
            end
         4'b1110: begin
            sensor_addr_zero       <= 4'b0111;
            sensor_addr_ninety      <= 4'b0001;
            sensor_addr_one_eighty   <= 4'b1000;
            sensor_addr_two_seventy   <= 4'b1110;
            end
         4'b0001: begin
            sensor_addr_zero       <= 4'b1000;
            sensor_addr_ninety      <= 4'b1110;
            sensor_addr_one_eighty   <= 4'b0111;
            sensor_addr_two_seventy   <= 4'b0001;
            end
         4'b0101: begin
            sensor_addr_zero       <= 4'b1001;
```

```
            sensor_addr_ninety      <= 4'b1010;
            sensor_addr_one_eighty   <= 4'b0110;
            sensor_addr_two_seventy   <= 4'b0101;
            end
          4'b1001: begin
            sensor_addr_zero       <= 4'b1010;
            sensor_addr_ninety      <= 4'b0110;
            sensor_addr_one_eighty   <= 4'b0101;
            sensor_addr_two_seventy   <= 4'b1001;
            end
          4'b1101: begin
            sensor_addr_zero       <= 4'b1011;
            sensor_addr_ninety      <= 4'b0010;
            sensor_addr_one_eighty   <= 4'b0100;
            sensor_addr_two_seventy   <= 4'b1101;
            end
           4'b0000: begin
            sensor_addr_zero       <= 4'b1100;
            sensor_addr_ninety      <= 4'b1111;
            sensor_addr_one_eighty   <= 4'b0011;
            sensor_addr_two_seventy   <= 4'b0000;
            end
          4'b0100: begin
            sensor_addr_zero       <= 4'b1101;
            sensor_addr_ninety      <= 4'b1011;
            sensor_addr_one_eighty   <= 4'b0010;
            sensor_addr_two_seventy   <= 4'b0100;
            end
          4'b1000: begin
            sensor_addr_zero       <= 4'b1110;
            sensor_addr_ninety      <= 4'b0111;
            sensor_addr_one_eighty   <= 4'b0001;
            sensor_addr_two_seventy   <= 4'b1000;
            end
          4'b1100: begin
            sensor_addr_zero       <= 4'b1111;
            sensor_addr_ninety      <= 4'b0011;
            sensor_addr_one_eighty   <= 4'b0000;
            sensor_addr_two_seventy   <= 4'b1100;
            end
        endcase
    end
endmodule
```

7.2.14 Coordinate Translation Module (demo version)

```
////////////////////////////////////////////////////////////////////////////////////
//
//     module to translate XY coordinates to physical addresses
//     dsheen 11/19/17
//      changed 12/7/19 to allow universal rotations
//      tweaked so that it works with the as built hardware, something kind of wierd
//      happened at some point there
//
////////////////////////////////////////////////////////////////////////////////////

module coordinate_translation #(parameter N = 4, XY_QUADRANT = 4)
      (

      //global signals
      input wire             clk,
      input wire             reset,
      input wire [N-1:0]      max_internal_address,

      //communication with the mapping module
      input wire                  set_output,         //tells map frame buffer to shoift out newly written data
        input wire                  write_enable,       //enable to write to the lookup BRAM
        input wire    [N-1:0]       x_address,          //X position of block to be written
        input wire    [N-1:0]       y_address,          //Y position of block to be written
        inout wire    [N+2:0]       data_in,            //data formatted as {valid,rotated,internal_address}
        output wire   [N+2:0]       data_out,           //return  data for mapping module in case anything is messed up.

        input wire    [N-1:0]       max_x_bound_in,      //these shall be forwarded on to the user along with the map data
        input wire    [N-1:0]       max_y_bound_in,      //these shall be forwarded on to the user along with the map data


        //user facing communications
        input wire [N+1:0]   x_coordinate,
        input wire [N+1:0]   y_coordinate,

        output reg [N+1:0]  max_x_bound_out,
        output reg [N+1:0]  max_y_bound_out,
```

```
     output wire       valid,         //assert this if sensor is in fact present at requested location and data is ready
     output reg          error,         //we'll assert this if the xy request from user is not within bounds
     output wire [N+3:0] address        //address to read sensor data
   );


//instantiate frame BRAMS

//first frame
wire          frame1_we;
wire [N-1:0]   frame1_x;
wire [N-1:0]   frame1_y;
wire [N+2:0]   frame1_data_in;
wire [N+2:0]   frame1_data_out;

mybram #(.LOGSIZE(2*N),.WIDTH(N+3))
               address_data_1(.addr({frame1_x,frame1_y}),.clk(clk),.we(frame1_we),.din(frame1_data_in),.dout(frame1_data_out));


//second frame
wire          frame2_we;
wire [N-1:0]   frame2_x;
wire [N-1:0]   frame2_y;
wire [N+2:0]   frame2_data_in;
wire [N+2:0]   frame2_data_out;

mybram #(.LOGSIZE(2*N),.WIDTH(N+3))
             address_data_2(.addr({frame2_x,frame2_y}),.clk(clk),.we(frame2_we),.din(frame2_data_in),.dout(frame2_data_out));


 // frame output state control
 reg output_frame;

 always @(posedge clk) begin
    if (reset) begin
       output_frame    <= 0;
        max_x_bound_out   <= 0;
        max_y_bound_out   <= 0;
        error          <= 0;
       end

      else if (set_output) begin
         output_frame    <= ~output_frame;
         max_x_bound_out   <= ((max_x_bound_in + 1) << 2) -1;
         max_y_bound_out   <= ((max_y_bound_in + 1) << 2) -1;
         error          <= 0;
       end

       else error <= (x_coordinate > max_x_bound_out) || (y_coordinate > max_y_bound_out);
 end


 wire [N+1:0] x_coordinate_quadrant;
 wire [N+1:0] y_coordinate_quadrant;
 //if output frame ==0, make frame 1 outputed and frame 2 written to, otherwise do the reverse

 assign frame1_we    = output_frame ? write_enable : 0;
 assign frame2_we    = output_frame ? 0 : write_enable;

 assign frame1_x      = output_frame ? x_address : x_coordinate_quadrant[N+1:2];
 assign frame1_y      = output_frame ? y_address : y_coordinate_quadrant[N+1:2];
 assign frame2_x      = output_frame ? x_coordinate_quadrant[N+1:2] : x_address;
 assign frame2_y      = output_frame ? y_coordinate_quadrant[N+1:2] : y_address;

 assign frame1_data_in   = output_frame ? data_in : 0;
 assign frame2_data_in   = output_frame ? 0 : data_in;

 assign data_out    = output_frame ? frame1_data_out : frame2_data_out;


 wire [1:0]   rotation;
 wire [1:0]   rotation_corrected; //we need to be able to correct the rotation for the fact that it will be mirrored in different coordinate
systems
 wire [3:0]   yx;

 assign rotation = output_frame ? frame2_data_out[N+1:N] : frame1_data_out[N+1:N];

 //need to transform this

 assign yx[1:0]    = x_coordinate_quadrant[1:0];
 assign yx[3:2]  = y_coordinate_quadrant[1:0];
 assign rotation_corrected = rotation;

 if (XY_QUADRANT == 1) begin
    assign x_coordinate_quadrant = x_coordinate;
```

```
       assign y_coordinate_quadrant  = y_coordinate;
       //assign rotation_corrected = rotation;
    end

    else if (XY_QUADRANT == 2) begin
       assign x_coordinate_quadrant = max_x_bound_out - x_coordinate;
       assign y_coordinate_quadrant = y_coordinate;
       //flip rotation angles (this is a kludge to deal with a bit of wierdness in how the hardware was wired in the end (x and y are mapped
wierdly))
       /*reason this works
          zero       = 00
          ninety       = 01
          one_eighty  = 10
         two_seventy = 11
      */
       //assign rotation_corrected[1] = ^rotation;
       //assign rotation_corrected[0] = rotation[0];
    end

    if (XY_QUADRANT == 3) begin
       assign x_coordinate_quadrant = max_x_bound_out - x_coordinate;
       assign y_coordinate_quadrant = max_y_bound_out - y_coordinate;
       //assign rotation_corrected = rotation;
    end

    if (XY_QUADRANT == 4) begin
       assign x_coordinate_quadrant = x_coordinate;
       assign y_coordinate_quadrant = max_y_bound_out - y_coordinate;

       //assign rotation_corrected[1] = ^rotation;
       //assign rotation_corrected[0] = rotation[0];
    end

    //note on the above. the way I implemented this in the end was dumb. In hindsight, I sould have had the entirety of the address mapping
    //handled at once, since then the coordinate system would have been very easy to just nicely transform with a few matrix operations. the only
    //justification for this arrangement is it saves a lot of BRAM space (a factor of 16) when a large number of blocks are allowed.

    wire [3:0]   sensor_addr_zero;
    wire [3:0]   sensor_addr_ninety;
    wire [3:0]  sensor_addr_one_eighty;
    wire [3:0]    sensor_addr_two_seventy;

    //instantiate rotation lookup rom

    xy_to_sensor_addr_rom lookup1(.clk(clk),
                         .xy(xy),
                         .sensor_addr_zero(sensor_addr_zero),
                         .sensor_addr_ninety(sensor_addr_ninety),
                         .sensor_addr_one_eighty(sensor_addr_one_eighty),
                         .sensor_addr_two_seventy(sensor_addr_two_seventy)
                         );

     parameter zero          = 0;
      parameter ninety      = 1;
    parameter one_eighty    = 2;
    parameter two_seventy    = 3;

    assign address [3:0]   = (rotation_corrected == zero) ? sensor_addr_zero : ((rotation_corrected == ninety) ? sensor_addr_ninety :
((rotation_corrected == one_eighty) ? sensor_addr_one_eighty : sensor_addr_two_seventy));
    assign address [N+3:4]    = output_frame ? frame2_data_out[N-1:0] : frame1_data_out[N-1:0];
    assign valid         = output_frame ? frame2_data_out[N+2] : frame1_data_out[N+2];


endmodule


//////////////////////////////////////////////////////////////////////////////////
///
/// this module translates the lower bits of the XY position into two possible
/// sensor numbers dependent on the block rotation
///
/// dsheen 11/2/2017
///
//////////////////////////////////////////////////////////////////////////////////

module xy_to_sensor_addr_rom (
                     input wire        clk,
                     input wire [3:0] yx,
                     output reg [3:0] sensor_addr_zero,
                     output reg [3:0] sensor_addr_ninety,
                     output reg [3:0] sensor_addr_one_eighty,
                     output reg [3:0] sensor_addr_two_seventy
                     );

   //rom data may be different depending on how we do things.
   always @(posedge clk) begin
```

```
case(yx)
   4'b0000: begin
      sensor_addr_zero       <= 4'b0000;
      sensor_addr_ninety      <= 4'b1100;
      sensor_addr_one_eighty   <= 4'b1111;
      sensor_addr_two_seventy   <= 4'b0011;
      end
   4'b0001: begin
      sensor_addr_zero       <= 4'b0001;
      sensor_addr_ninety      <= 4'b1000;
      sensor_addr_one_eighty   <= 4'b1110;
      sensor_addr_two_seventy   <= 4'b0111;
      end
   4'b0010: begin
      sensor_addr_zero       <= 4'b0010;
      sensor_addr_ninety      <= 4'b0100;
      sensor_addr_one_eighty   <= 4'b1101;
      sensor_addr_two_seventy   <= 4'b1011;
      end
   4'b0011: begin
      sensor_addr_zero       <= 4'b0011;
      sensor_addr_ninety      <= 4'b0000;
      sensor_addr_one_eighty   <= 4'b1100;
      sensor_addr_two_seventy   <= 4'b1111;
      end
   4'b0100: begin
      sensor_addr_zero       <= 4'b0100;
      sensor_addr_ninety      <= 4'b1101;
      sensor_addr_one_eighty   <= 4'b1011;
      sensor_addr_two_seventy   <= 4'b0010;
      end
   4'b0101: begin
      sensor_addr_zero       <= 4'b0101;
      sensor_addr_ninety      <= 4'b1001;
      sensor_addr_one_eighty   <= 4'b1010;
      sensor_addr_two_seventy   <= 4'b0110;
      end
   4'b0110: begin
      sensor_addr_zero       <= 4'b0110;
      sensor_addr_ninety      <= 4'b0101;
      sensor_addr_one_eighty   <= 4'b1001;
      sensor_addr_two_seventy   <= 4'b1010;
      end
   4'b0111: begin
      sensor_addr_zero       <= 4'b0111;
      sensor_addr_ninety      <= 4'b0001;
      sensor_addr_one_eighty   <= 4'b1000;
      sensor_addr_two_seventy   <= 4'b1110;
      end
   4'b1000: begin
      sensor_addr_zero       <= 4'b1000;
      sensor_addr_ninety      <= 4'b1110;
      sensor_addr_one_eighty   <= 4'b0111;
      sensor_addr_two_seventy   <= 4'b0001;
      end
   4'b1001: begin
      sensor_addr_zero       <= 4'b1001;
      sensor_addr_ninety      <= 4'b1010;
      sensor_addr_one_eighty   <= 4'b0110;
      sensor_addr_two_seventy   <= 4'b0101;
      end
   4'b1010: begin
      sensor_addr_zero       <= 4'b1010;
      sensor_addr_ninety      <= 4'b0110;
      sensor_addr_one_eighty   <= 4'b0101;
      sensor_addr_two_seventy   <= 4'b1001;
      end
   4'b1011: begin
      sensor_addr_zero       <= 4'b1011;
      sensor_addr_ninety      <= 4'b0010;
      sensor_addr_one_eighty   <= 4'b0100;
      sensor_addr_two_seventy   <= 4'b1101;
      end
    4'b1100: begin
      sensor_addr_zero       <= 4'b1100;
      sensor_addr_ninety      <= 4'b1111;
      sensor_addr_one_eighty   <= 4'b0011;
      sensor_addr_two_seventy   <= 4'b0000;
      end
   4'b1101: begin
      sensor_addr_zero       <= 4'b1101;
      sensor_addr_ninety      <= 4'b1011;
      sensor_addr_one_eighty   <= 4'b0010;
      sensor_addr_two_seventy   <= 4'b0100;
      end
   4'b1110: begin
```

```
            sensor_addr_zero       <= 4'b1110;
            sensor_addr_ninety      <= 4'b0111;
            sensor_addr_one_eighty   <= 4'b0001;
            sensor_addr_two_seventy   <= 4'b1000;
            end
         4'b1111: begin
            sensor_addr_zero       <= 4'b1111;
            sensor_addr_ninety      <= 4'b0011;
            sensor_addr_one_eighty   <= 4'b0000;
            sensor_addr_two_seventy   <= 4'b1100;
            end
      endcase
   end
endmodule
```

7.2.15 Polling and Capacitance Sensing Modules

```
//////////////////////////////////////////////////////////////////////////////////
//
//  The cap_sense module handles reading the sensor data lines
//   The polling module iterates through all available sensors and reads them using
//  the cap_sense module
//
//  apfitzen 11/13/2017
//
//////////////////////////////////////////////////////////////////////////////////

module cap_sense #(parameter SENSE_TIME = 75000) //default 750us (assuming 100mhz clock)
            (input clock,
             input sensor_in, //input to FPGA from sensor blocks
              input start,     //the module begins reading a sensor data line on a
                               //rising edge of start

              output reg done, //done is pulsed high for one clock cycle once the read
                               //is complete
              output [11:0] data //output data from the read, valid after done is pulsed
              );

   reg [11:0] edge_count; //number of transitions counted on the sensor data line
   reg [19:0] sense_clocks; //number of clock cycles since the read began

   reg old_start;
   reg old_sensor_in;

   reg reading; //are we currently reading a sensor

   always @(posedge clock) begin
      old_start <= start;
       old_sensor_in <= sensor_in;

      if (start!=old_start && start) begin //rising edge of start
         edge_count <= 0;
         done <= 0;
         reading <= 1;
         sense_clocks <= 0;
      end

      else if (reading) begin
         sense_clocks <= sense_clocks + 1;
         if (sense_clocks == SENSE_TIME) begin
            //we're done sensing once SENSE_TIME clock cycles have elapsed
            done <= 1;
            reading <= 0;
         end
         else if (old_sensor_in!=sensor_in) begin //sensor data line has transitioned
            edge_count <= edge_count + 1;
         end
      //begin
      end
      else if (done) done <= 0; //pulse done for only one clock cycle

   end
   assign data = edge_count;

endmodule

module polling #(parameter N = 4, COMMAND_LEN = 5)
                (input clock,
            input reset,
            input start, //begins polling on a rising edge of start
            input i2c_done, //goes high once an i2c command is completed
            input [N-1:0] max_internal_address, //total number of boards in the system
            input sensor_in1,input sensor_in2, //sensor data line inputs from the blocks
```

```
      output reg done, //pulses high for one clock cycle once all sensors have been read

      output reg i2c_start, //assert to send an i2c command
      output reg [COMMAND_LEN-1:0] i2c_command, //command to be sent to the i2c command module
      output [N-1:0] internal_address, //the internal address of the board we are
                           //currently reading

      //add past state bram later
      output reg [N+3:0] bram1_address, //address in the sensor data bram to write to
      output reg bram1_we, //write enable for sensor data bram
      output reg [11:0] bram1_din,
      output reg [4:0] state //data to write into the sensor data bram
      );

//FSM states
parameter SENSE_RAILS_ON1 = 1;
parameter SENSE_RAILS_ON2 = 2;
parameter SET_MUXES1 = 3;
parameter SET_MUXES2 = 4;
parameter START_POLLING1 = 5;
parameter START_POLLING2 = 6;
parameter UPDATE_BRAM1 = 7;
parameter UPDATE_BRAM2 = 8;
parameter UPDATE_SENSOR_IN = 9;
parameter SENSE_RAILS_OFF1 = 10;
parameter SENSE_RAILS_OFF2 = 11;
parameter DONE = 12;

reg old_start;
reg [N-1:0] block_address; //internal address of the board we are currently reading
reg [2:0] sensor_num; //sensor mux select bits
//reg [4:0] state; //the FSM state we are currently in

//modules to read the two sensor data lines
reg sensor_start1;
reg sensor_start2;
wire sensor_done1;
wire sensor_done2;
wire [11:0] sensor_data1;
wire [11:0] sensor_data2;
cap_sense sensor1(.clock(clock), .sensor_in(sensor_in1), .start(sensor_start1),
              .done(sensor_done1), .data(sensor_data1));
cap_sense sensor2(.clock(clock), .sensor_in(sensor_in2), .start(sensor_start2),
              .done(sensor_done2), .data(sensor_data2));

always @(posedge clock) begin
   old_start <= start;
   if (reset) begin //reset condition
      i2c_start <= 0;
      sensor_start1 <= 0;
      sensor_start2 <= 0;

      block_address <= 0;
      sensor_num <= 0;

      bram1_we <= 0;
      bram1_address <= 0;
      bram1_din <= 0;

      state <= DONE;
      done <= 0;
   end
   else begin
      //we are done polling, wait for another start signal
      //also the idle state of the polling module
      case (state)
      DONE: begin
         //wait for start
         if (start && old_start!=start) begin
            i2c_start <= 0;
            sensor_start1 <= 0;
            sensor_start2 <= 0;

            block_address <= 0;
            sensor_num <= 0;

            bram1_we <= 0;
            bram1_address <= 0;
            bram1_din <= 0;

            //start polling by switching on the tristate buffers for the first board
            //Note: this assumes, all tristate buffers we shut off prior to starting
            //the polling module
            state <= SENSE_RAILS_ON1;
            done <= 0;
         end
```

```
end

   //send i2c command to turn on tristate buffers for the board we are sensing
   SENSE_RAILS_ON1: begin
      i2c_start <= 1;
      i2c_command <= 5'b10100;
      state <= SENSE_RAILS_ON2;
   end
   //wait for completion and update state
   SENSE_RAILS_ON2: begin
      i2c_start <= 0;
      if (i2c_done) state <= SET_MUXES1;
      else state <= SENSE_RAILS_ON2;
   end

   //send i2c command to set the sensor mux select lines
   SET_MUXES1: begin
      i2c_start <= 1;
      i2c_command <= {2'b11,sensor_num};
      state <= SET_MUXES2;
   end
   //wait for completion and update state
   SET_MUXES2: begin
      i2c_start <= 0;
      if (i2c_done) state <= START_POLLING1;
      else state <= SET_MUXES2;
   end

   //begin polling the sensor lines
   START_POLLING1: begin
      sensor_start1 <= 1;
      sensor_start2 <= 1;
      state <= START_POLLING2;
   end
   //wait for completion and update state
   START_POLLING2: begin
      sensor_start1 <= 0;
      sensor_start2 <= 0;
      if (sensor_done1 && sensor_done2) state <= UPDATE_BRAM1;
      else state <= START_POLLING2;
   end

   //update the bram entry corresponding to the sensor on the first sensor line
   UPDATE_BRAM1: begin
      //bram address: internal_address-0-mux select bits
      bram1_address <= {block_address,1'b0,sensor_num};
      bram1_din <= sensor_data1;
      bram1_we <= 1;
      state <= UPDATE_BRAM2;
   end
   //update the bram entry corresponding to the sensor on the second sensor line
   UPDATE_BRAM2: begin
       //bram address: internal_address-1-mux select bits
      bram1_address <= {block_address,1'b1,sensor_num};
      bram1_din <= sensor_data2;
      bram1_we <= 1;
      state <= UPDATE_SENSOR_IN;
   end

   //update the sensors to read
   UPDATE_SENSOR_IN: begin
      bram1_we <= 0;
      sensor_num <= sensor_num + 1; //this will overflow to zero if we finish a board

      //we've finished reading the current board,
      //need to shut off the previous one's tristate buffers
      if (sensor_num == 3'b111) state <= SENSE_RAILS_OFF1;

      //still reading the same board, jump straight to setting mux selects
      else state <= SET_MUXES1;
   end

   //send i2c command to turn off tristate buffers for the board we were sensing
   SENSE_RAILS_OFF1: begin
      i2c_start <= 1;
      i2c_command <= 5'b10000;
      state <= SENSE_RAILS_OFF2;
   end
   //wait for completion and update state
   SENSE_RAILS_OFF2: begin
      i2c_start <= 0;
      if (i2c_done) begin
         if (block_address < max_internal_address) begin
            //still more boards to read, begin reading next board
            block_address <= block_address + 1;
            state <= SENSE_RAILS_ON1;
```

```
                    end
                    else begin //all boards have been read, we're done
                       state <= DONE;
                       done <= 1;
                    end
                 end
                 else state <= SENSE_RAILS_OFF2;
              end
              default: state <= DONE;
           endcase
        end
     end

     assign internal_address = block_address;

endmodule
```

7.2.16 Display Modules

```
module display #(parameter N=4,
                    SCREEN_WIDTH = 1024,
                    SCREEN_HEIGHT = 768,
                    ON_COLOR = 24'hFF_00_00,
                    OFF_COLOR = 24'h00_FF_00,
                    INVALID_COLOR = 24'h00_00_FF)
  (input vclock,  // 65MHz clock
   input reset,      // 1 to initialize module
   input [10:0] pad_size, //dimension in pixels a single pad has on the screen
   input [11:0] pad_data, //data from xy status bram for a pad
   input pad_valid, //whether or not the pad is valid
   input [11:0] threshold, //threshold for turning a pad on
   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 at_display_area,      // XVGA blanking (1 means output black pixel)

   //output reg [N+1:0] padX, //the x coordinate of the pad we are drawing
   //output reg [N+1:0] padY, //the y coordinate of the pad we are drawing
   output reg [N+1:0] xCoord, //the x coordinate of the pad we are drawing
   output reg [N+1:0] yCoord, //the y coordinate of the pad we are drawing
   output reg phsync,   // pong game's horizontal sync
   output reg pvsync,   // pong game's vertical sync
   output reg p_at_display_area,   // pong game's blanking
   output [23:0] pixel // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );

   //xCoord and yCoord get incremented when one of these reaches pad_size
   reg [10:0] xCount;
   reg [10:0] yCount;

   //pipeline xcount and ycount for pixel generation
   reg [10:0] pxCount;
   reg [10:0] pyCount;
   reg [10:0] xCount_temp;
   reg [10:0] yCount_temp;

   reg [23:0] pixel_color;

   reg hsync_temp,vsync_temp,at_display_area_temp;

   //add a black border around each pad
   assign pixel = ((pxCount == 0 || pxCount == pad_size-1 || pyCount == 0 || pyCount == pad_size-1) && pad_valid) ? 24'h00_00_00 : pixel_color;

   //assign phsync = hsync;
   //assign pvsync = vsync;
   //assign p_at_display_area = at_display_area;

   always @(posedge vclock) begin
         // delay for two cycles due to pixel generation delay
         hsync_temp <= hsync;
         vsync_temp <= vsync;
         at_display_area_temp <= at_display_area;
         xCount_temp <= xCount;
         yCount_temp <= yCount;

         p_at_display_area <= at_display_area_temp;
         pvsync <= vsync_temp;
         phsync <= hsync_temp;
         pxCount <= xCount_temp;
         pyCount <= yCount_temp;


         if (reset) begin
```

```
            //padX <= 0;
            //padY <= 0;
            xCoord <= 0;
            yCoord <= 0;
            xCount <= 0;
            yCount <= 0;
         end

         else begin

            if (hcount == 0) begin
               //reset xCount once a line is complete
               xCount <= 0;
               xCoord <= 0;

               if (vcount == 0) begin
                  //reset yCount once a frame is complete
                  yCount <= 0;
                  yCoord <= 0;
               end
               else begin
                  if (yCount == pad_size - 1) begin
                     //finished displaying a pad, go to the next pad
                    yCount <= 0;
                    yCoord <= yCoord + 1;
                  end
                  else begin
                     yCount <= yCount + 1;
                  end
               end

            end
            else begin
               if (xCount == pad_size - 1) begin
                  //finished displaying a line for one pad, go to the next pad
                  xCount <= 0;
                  xCoord <= xCoord + 1;
               end
               else begin
                  xCount <= xCount + 1;
               end
            end

            /*if (vcount == 0) begin
               //reset yCount once a frame is complete
               yCount <= 0;
               yCoord <= 0;
            end
            else begin
               if (yCount == pad_size - 1) begin
                  //finished displaying a pad, go to the next pad
                  yCount <= 0;
                  yCoord <= yCoord + 1;
               end
               else begin
                  yCount <= yCount + 1;
               end
            end*/

            if (pad_valid) begin
               if (pad_data < threshold) begin
                  pixel_color <= ON_COLOR;
               end
               else begin
                  pixel_color <= OFF_COLOR;
               end
            end
            else begin
               pixel_color <= INVALID_COLOR;
            end

            //pixel <= ON_COLOR;

         end
   end

endmodule

//from lab 4
module xvga(input vga_clock,
            output reg [10:0] hcount = 0,    // pixel number on current line
            output reg [9:0] vcount = 0,    // line number
            output reg vsync, hsync,
            output at_display_area);

   // Comments applies to XVGA 1024x768, left in for reference
```

```
   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);    // active H  1023
   assign hsyncon = (hcount == 1047);     // active H + FP 1047
   assign hsyncoff = (hcount == 1183);    // active H + fp + sync  1183
   assign hreset = (hcount == 1343);      // active H + fp + sync + bp 1343

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);    // active V   767
   assign vsyncon = hreset & (vcount ==776 );     // active V + fp   776
   assign vsyncoff = hreset & (vcount == 783);    // active V + fp + sync  783
   assign vreset = hreset & (vcount == 805);      // active V + fp + sync + bp 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 vga_clock) 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

   end

   assign at_display_area = ((hcount >= 0) && (hcount < 1024) && (vcount >= 0) && (vcount < 768));

endmodule
```

7.2.17 Number Pad Modules

```
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:    Gim P. Hom
//
// Create Date:   13:26:09 05/07/06
// Module Name:   rs232send
//
// Revision:    08/11/2017
// Revision 0.01 - File Created  5/7/2006  GPH
// Additional Comments:  This file will transmit ascii characters to a RS-232
// interface. The baud rate for the example is preset to 115,200
//
// Character is sent when start_send goes high.
//
// The calcuations assume a clock of 27 mhz.
//
// Baud Rate     DIVISOR  Exact   Actual Baud   % error
//  230,400       117   117.1875 230,769.23     0.16%
//  115,200       234   234.375  115,384.62     0.16%
//   57,600       468   468.75   57,692.31      0.16%
//   38,400       703   703.125  38,406.83      0.02%
//   19,200       1,406  1406.25  19,203.41      0.02%
//    9,600       2,812  2812.5   9,601.71       0.02%
//    4,800       5,625  5625    4,800.00       0.00%
//    2,400      11,250  11250   2,400.00       0.00%
//    1,200      22,500  22500   1,200.00       0.00%
//////////////////////////////////////////////////////////////////////////////////

//borrowed from lab 2
module rs232send(
   input clk,
   input [7:0] data,   // 8 bit ascii data
   input start_send,   // one clock pulse width, enter pressed, data available
   output xmit_data   // serial data sent to RS232 output driver
   );

   // this section sets up the clk;
   parameter DIVISOR =  6671; // create 57600 baud rate clock from 65mhz clk, not exact, but should work.
   parameter MARK = 1'b1;
   parameter STOP_BIT = 1'b1;
   parameter START_BIT =1'b0;

   reg [15:0] count;
   reg xmit_clk;
```

```
  always @ (posedge clk)
  begin
    count <= xmit_clk ? 0 : count+1;
    xmit_clk <= count == DIVISOR-1;
  end

  ////////////////////////////////////////////////////////////////////////////
  //  insert your Verilog here

 reg [9:0] dataToSend;
 reg bitToSend;
 always @ (posedge clk) begin
  if (start_send) begin
    dataToSend <= {STOP_BIT,data,START_BIT};
  end
  else if (xmit_clk) begin
    dataToSend <= {STOP_BIT,dataToSend[9:1]};
    bitToSend <= dataToSend[0];
  end

 end


  assign xmit_data = bitToSend;  // change this line

  //
  ////////////////////////////////////////////////////////////////////////////

endmodule

module num_pad
          #(parameter N = 4)
          (
           input clk,
           input reset,
           input [11:0] pad_data, //data from xy status bram for a pad
           input [11:0] threshold,

           output reg [N+1:0] xCoord, //the x coordinate of the pad we are reading
           output reg [N+1:0] yCoord, //the y coordinate of the pad we are reading
           output xmit_data
          );

    reg [15:0] data;
    reg [7:0] send_this;
    reg [4:0] state;
    reg start_send;
    reg [31:0] delay_count;

    reg [7:0] count;

    parameter send_delay = 6671*10*20; //DIVISOR * 10bits * 20

    parameter START = 0;
    parameter DATA_DELAY = 1;
    parameter UPDATE_DATA = 2;
    parameter START_SEND = 3;
    parameter SEND_DELAY1 = 4;
    parameter START_SEND2 = 5;
    parameter SEND_DELAY2 = 6;

    rs232send serial_send(.clk(clk),.data(send_this),.start_send(start_send),.xmit_data(xmit_data));

    always @(posedge clk) begin
     if (reset) begin
      xCoord <= 0;
      yCoord <= 0;
      data <= 0;
      start_send <= 0;
      state <= 0;
      delay_count <= 0;
      count <= 0;
     end

     case (state)
      START: begin
       xCoord <= 0;
       yCoord <= 0;
       data <= 0;
       state <= DATA_DELAY;
       delay_count <= 0;
       //count <= 0;
      end

      DATA_DELAY: begin
       state <= UPDATE_DATA;
```

```
end

UPDATE_DATA: begin

  //update data
  if (pad_data < threshold) begin

    if (yCoord <= 2) begin
      data[xCoord + 3*yCoord] <= 1;
    end
    else begin //handle zero pad separately
      data[9] <= 1;
    end

  end

  //update pad coord
  if (xCoord == 2) begin

    if (yCoord == 2) begin

      //read 0 pad
      yCoord <= 3;
      xCoord <= 1;
      state <= DATA_DELAY;

    end

    else begin
      yCoord <= yCoord + 1;
      xCoord <= 0;
      state <= DATA_DELAY;
    end
  end

  else if (yCoord == 3) begin

      xCoord <= 0;
      yCoord <= 0;
      state <= START_SEND;

  end

  else begin
    xCoord <= xCoord + 1;
    state <= DATA_DELAY;
  end
end

START_SEND: begin
  //send off pads 1-8
  start_send <= 1;
  send_this <= {1'b0,data[5:0],1'b0};
  //send_this <= count;
  count <= count + 1;
  state <= SEND_DELAY1;

end

SEND_DELAY1: begin

  //wait for byte to send
  start_send <= 0;
  if (delay_count == send_delay) begin

    state <= START_SEND2;
    delay_count <= 0;

  end
  else begin
    delay_count <= delay_count + 1;
    state <= SEND_DELAY1;
  end

end

START_SEND2: begin

  start_send <= 1;
  send_this <= {1'b1,data[11:6],1'b1};
  //send_this <= count;
  count <= count + 1;
  state <= SEND_DELAY2;

end
```

```
      SEND_DELAY2: begin

       //wait for byte to send
       start_send <= 0;
       if (delay_count == send_delay) begin

         state <= START;
         delay_count <= 0;

       end
       else begin
         delay_count <= delay_count + 1;
         state <= SEND_DELAY2;
       end

      end

     endcase


    end

endmodule
```

7.2.18 make_xy_status_bram

```
//generate a bram that maps xy coordinates for touchpads to statuses (on,off,invalid)
//apfitzen 12/2/17
module make_xy_status_bram
    #(parameter N = 4)
   (
      input clk, //clock
      input [N+3:0] xy_addr, //address to read sensor data from coordinate translation module
      input valid, //is sensor at xy location valid, from coordinate translation module
      input error, //is the xy location within bounds, from coordinate translation module
      input [11:0] polling_data, //polling bram data
      input [N+1:0] maxX,
      input [N+1:0] maxY,
      input reset,
      input start, //start generating the bram, this should be initiated by the control fsm
      input [11:0] threshold,

      output reg [N+1:0] xRead, //x coordinate of pad to check
      output reg [N+1:0] yRead, //y coordinate of pad to check
      output [N+3:0] polling_addr, //polling bram address
      output reg status_we, //write enable for the xy status bram
      output wire [2*N+3:0] status_bram_addr,
      output reg [11:0] status_data, //data to write to xy status bram (00=invalid, 01 = off, 11 = on)
      output reg done //we're done generating the bram
   );

   parameter DONE = 0;
   parameter WAIT_FOR_TRANSLATION = 1;
   parameter IS_SENSOR_VALID = 2;
   parameter UPDATE_XY = 3;
   parameter CHECK_THRESHOLD = 4;
   parameter UPDATE_XY_STATUS_BRAM1 = 5;
   parameter UPDATE_XY_STATUS_BRAM2 = 6;

   reg [3:0] state;
   reg old_start;

   assign polling_addr = xy_addr;
   assign status_bram_addr = {xRead,yRead};

   always @(posedge clk) begin
      old_start <= start;
      if (reset) begin
         xRead <= 0;
         yRead <= 0;
         status_we <= 0;
         status_data <= 0;
         done <= 0;

         state <= DONE;
      end

      else begin
         case (state)
            DONE: begin
               //wait for start
               if (old_start!=start && start) begin
                  xRead <= 0;
                  yRead <= 0;
```

```
                status_we <= 0;
                status_data <= 0;
                done <= 0;

                state <= WAIT_FOR_TRANSLATION;
            end
        end
        WAIT_FOR_TRANSLATION: begin
            //xRead and yRead have been updated, wait one clock cycle for the coordinate translation module
            state <= IS_SENSOR_VALID;
        end
        IS_SENSOR_VALID: begin
            //check whether the sensor is valid
            //the address output from the coordinate transformation module is available in
            //polling_addr now
            //data from polling bram will be available on the next clock cycle
            if (valid && !error) begin
                //sensor is valid, check if it's on
                state <= CHECK_THRESHOLD;
            end
            else begin
                //sensor is invalid, write 00 to xy status bram
                //status_data <= 2'b00;
                status_data <= 0;
                state <= UPDATE_XY_STATUS_BRAM1;
            end
        end

        CHECK_THRESHOLD: begin
            /*if (polling_data < threshold) begin
                //sensor is on
                status_data <= 2'b11;
            end
            else begin
                //sensor is off
                status_data <= 2'b01;
            end*/
            status_data <= polling_data;
            state <= UPDATE_XY_STATUS_BRAM1;
        end

        UPDATE_XY_STATUS_BRAM1: begin
            status_we <= 1;
            state <= UPDATE_XY_STATUS_BRAM2;
        end
        UPDATE_XY_STATUS_BRAM2: begin
            status_we <= 0;
            state <= UPDATE_XY;
        end

        UPDATE_XY: begin
            //update xRead and yRead
            if (xRead < maxX) begin
                xRead <= xRead + 1;
                state <= WAIT_FOR_TRANSLATION;
            end
            else begin
                xRead <= 0;
                if (yRead < maxY) begin
                    yRead <= yRead + 1;
                    state <= WAIT_FOR_TRANSLATION;
                end
                else begin
                    yRead <= 0;
                    state <= DONE;
                    done <= 1;
                end
            end
        end

    endcase

  end

end

endmodule
```