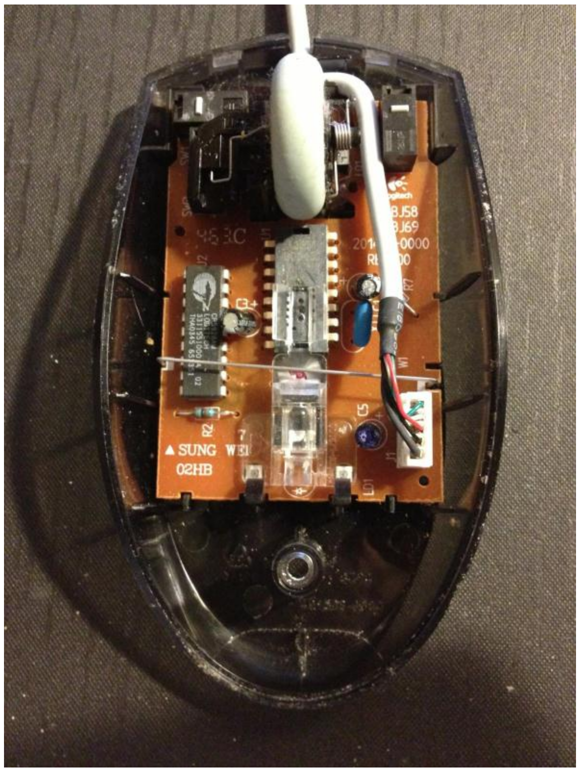
Optical Mouse Scanner



Embedded System Design - Prof. Stephen Edwards
CSEE 4840, Spring 2013

Group Name: optical-mouse-scanner
David Calhoun (dmc2202)
Kishore Padmaraju (kp2362)
Serge Yegiazarov (sy2464)

I. Overview

In our project, “Optical Mouse Scanner,” we will be implementing a system in which a user can create low resolution scans of a document using an ordinary optical mouse. The mouse is operated using a normal configuration, with the user running it over the portion of the document he is interested in scanning. The aggregated results of the current scan will be displayed on a computer monitor. By observing the aggregated scan on the monitor, the user can note which areas of the scan are missing or erroneous, and rescan the document at those locations.

The user will be able to use the left-click and right-click functionality of the mouse to scan and reset the scan, respectively. The current image being read by the optical mouse will always be displayed in a small inset box on the monitor display, next to the larger image of the aggregate scan up to that point. The position of the mouse will also be indicated on the aggregate scan. The figure below depicts the described visual output.

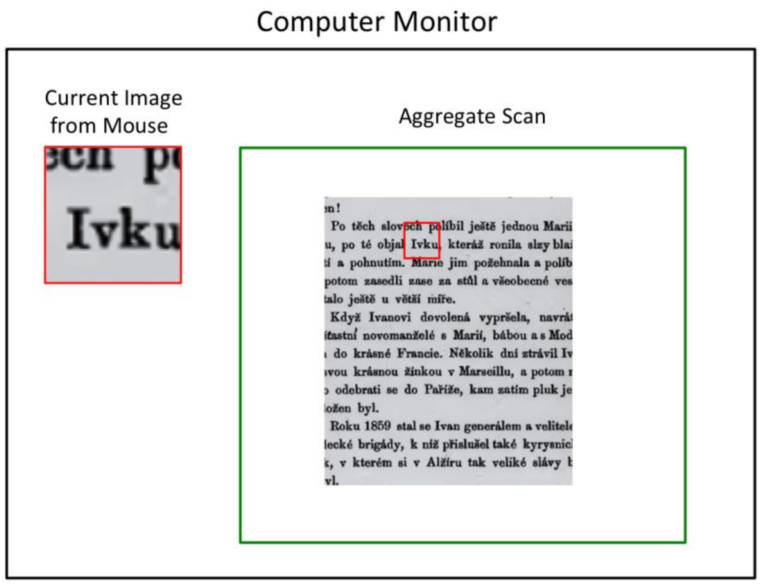


Figure 1: GUI

II. Description

To implement our low-resolution scanner, we will be interfacing with two main peripherals: the optical mouse and the VGA monitor.

Input - Optical Mouse

A standard optical mouse transmits information about the location of the mouse in a serial, packetized format. This information can be acquired using USB, PS/2, or direct connection to a serial pin on the processor—for the purposes of this project, the serial pin will be used. We will acquire image data from the optical mouse via the ADNS-2051 optical processor, which is one of a series of common optical processors distributed by Agilent for use in optical mice. This processor includes a 16x16 pixel CCD; the image data is acquired via a synchronous, half-duplex serial port on the processor. By soldering connections to the optical processor’s clock, serial I/O, and power status pins, we can communicate and exchange data with the optical processor. The ADNS-2051 data sheet outlines communication/request protocols that we must establish to access the CCD and other information. All connections to the optical processor will require soldering ribbon cables to the optical processor to establish custom GPIO (to connect to the Altera DE2).

Output - VGA Monitor

The output will be an image on a VGA monitor as described in the figure on the previous page.

Hardware

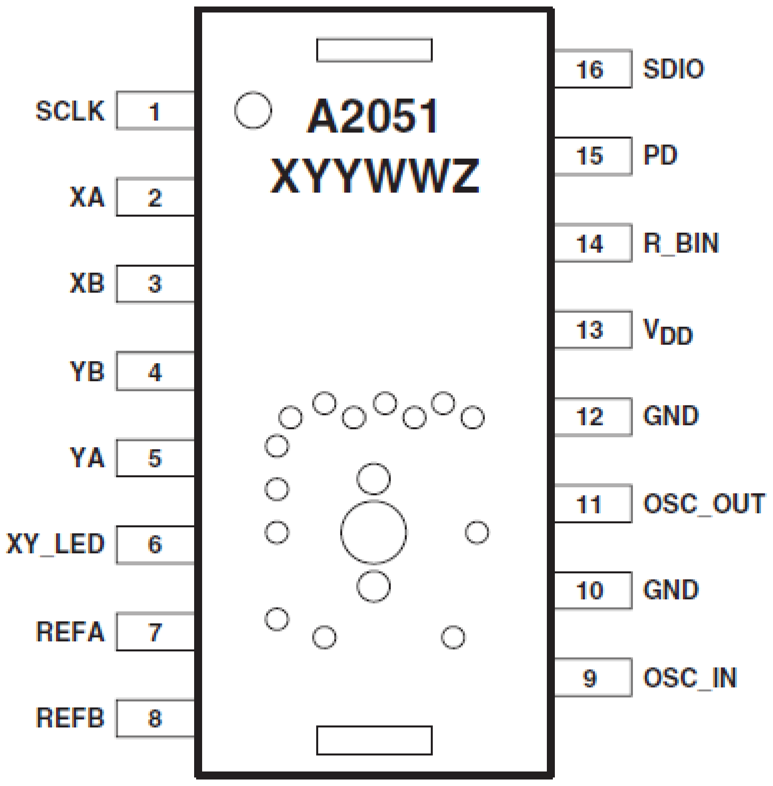
Hardware components will consist of the serial pin interface of the optical mouse, a GPIO connection to ADNS-2051 pins, and the interface of the VGA monitor. The parallel inputs and video processing (VGA) capabilities of the FPGA will be leveraged.

Software & Algorithms

There are several aspects of this project that require hardware and software functionality. We will map the serialized coordinate data to image data acquired from the mouse using our own custom “sorting” algorithm. This algorithm will spatially position the scanned image onto an external VGA monitor based on the coordinate data. Our first approach to this algorithm will be centering image data based on the change of X and Y coordinates received over the serial port.

Other software will consist of additional bookkeeping and image processing needed to compose the scan from the images retrieved from the optical mouse. The bookkeeping consists of making sure that the coordinates of the moving mouse correspond with the coordinates on the monitor to generate a cohesive image. Image processing will provide further refinement of the image, as necessary, from the raw data. This can be done using simple filtering techniques, such as via a low- order moving average filter, or a small median filter. Such filters work with the attenuation of high-frequency image noise and removal of arbitrary incorrect pixels.

Optical Mouse Scanner Project Design



Embedded System Design - Prof. Stephen Edwards
CSEE 4840, Spring 2013

Group Name: optical-mouse-scanner
David Calhoun (dmc2202)
Kishore Padmaraju (kp2362)
Serge Yegiazarov (sy2464)

Overview:

In our project, “Optical Mouse Scanner,” we will be implementing a system in which a user can create low resolution scans of a document using an ordinary optical mouse. The mouse is operated as per usual, with the user running it over the portion of the document he/she is interested in scanning. The aggregated results of the current scan will be displayed on a computer monitor. By observing the aggregated scan on the monitor, the user will be able to note which areas of the scan are missing or erroneous, and rescan the document at those locations.

The user will be able to use the left-click and right-click functionality of the mouse to begin and reset the scan, respectively. The current image being read by the optical mouse will always be displayed in a small inset box on the monitor display, next to the larger image of the aggregate scan up to that point. The position of the mouse will also be indicated on the aggregate scan.

A standard optical mouse transmits information about the location of the mouse in a serial, packetized format. This information can be acquired using USB, PS/2, or direct connection to a serial pin on the processor—for the purposes of this project, the serial pin will be used. We will acquire image data from the optical mouse via the ADNS-2051 optical processor, which is one of a series of common optical processors distributed by Agilent for use in optical mice - this processor includes a 16x16 pixel CCD. The image data and XY positional data is acquired via a synchronous, half-duplex serial port on the processor. By soldering connections to the optical processor’s clock, serial I/O, and power status pins, we can communicate with and exchange data with the optical processor. The ADNS-2051 data sheet outlines communication/request protocols that we must establish to access CCD and other information. All connections to the optical processor will require soldering ribbon cables to the optical processor to establish custom GPIO (to connect to the Altera DE2).

The following image is a high-level block diagram of all the hardware components and how they interact:

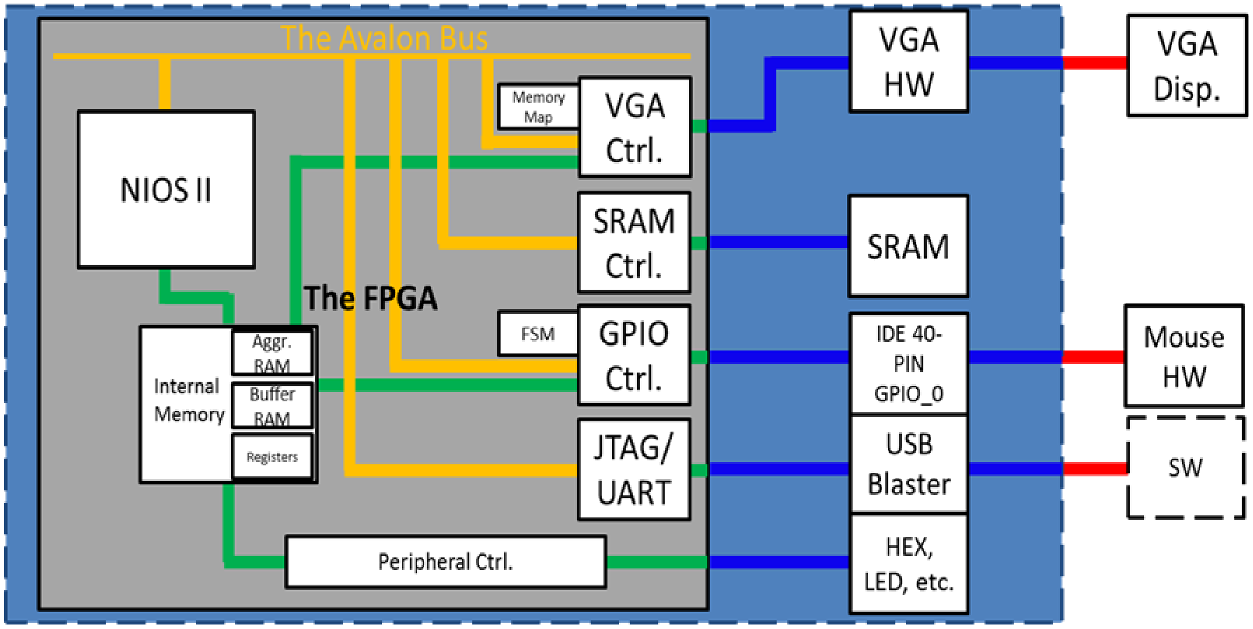


Figure 2: System architecture and hardware interconnection.

- The VGA Ctrl. block contains a controller for storing image samples to the aggregate image. This controller also handles the entire VGA raster scan. This raster scan includes 2 active regions, one for the larger aggregate image, and one for the smaller active scan (the smaller active scan is a live feed of the camera).
- The SRAM Ctrl. Block handles the interfacing with the SRAM. This is a key component because the software is stored on the SRAM.
- JTAG/UART is the standard interface on the DE2 for using the USB Blaster to download software onto the board.
- The GPIO Ctrl. Block is used to actually obtain the serialized data from the mouse.
- The internal memory stores all the image and location data of our samples. It contains the aggregate RAM, which corresponds to the 128x128 screen in our UI. It also contains the buffer RAM, which corresponds to the smaller 16x16 live feed samples. Other registers are responsible for holding the x and y coordinate to which the sample will be written.
- The peripheral controller is the standard interface by which to communicate with the hex and LED. We used these two in our project to keep some track of deltaX and deltaY

Interconnections within the DE2 block in Figure 2 indicate communication using the Avalon Bus (also known as the Avalon Switch Fabric).

Critical Path

Potential timing failure might arise when communicating between the ADNS-2051 optical processor and the DE2 board. Specifically, a polling FSM is required to communicate on the serial data line of the ADNS-2051, which will control how registers on the ADNS-2051 are accessed. The timing of how information is collected from these registers, how it is stored, and how it is used, is critical to this project design.

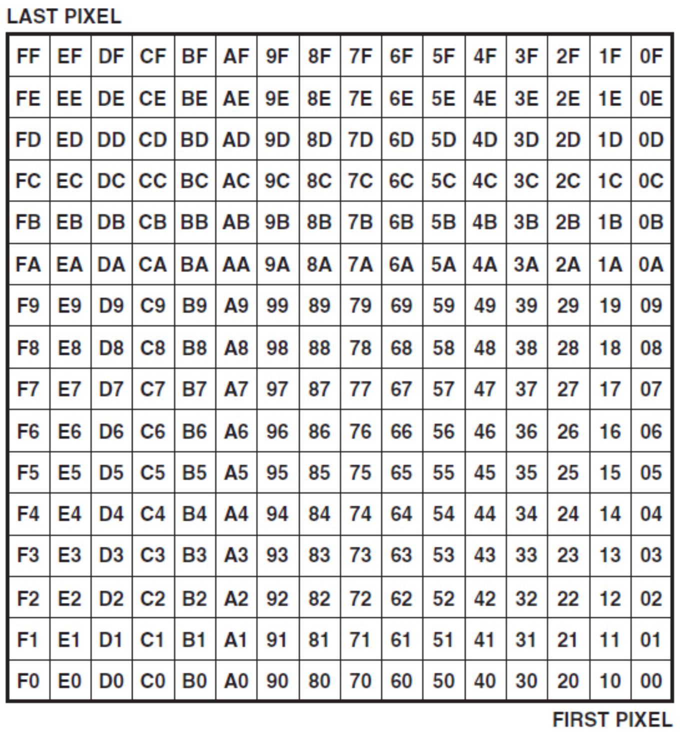
Specifically, the gray scale values of each pixel in a complete 16x16 scanned image sample are sequentially accessed and stored. When accessing registers that store the 6-bit gray scale values of each pixel in the 16x16 CCD image matrix, it is necessary to store all 256 pixels of the image matrix to associate with a given XY location. The given XY location itself must also be stored in a timely fashion, corresponding with the image matrix. Both of these pieces of information are stored in SRAM and accessed by software. The image and location information, stored in SRAM, is accessed via software for sorting and aggregation. This sorting and aggregation consists of displaying the current image (256 pixels) according to the current location information on the VGA display.

The exact critical path comes into play when associating the timing between polling the ADNS-2051 registers, storing/updating that information in SRAM, and accessing that information to aggregate on the VGA display. Concern rises when considering how quickly information will be collected from the mouse and updated in SRAM on the DE2, because that limits how quickly this information can be accessed and sent to the VGA display. These timing issues can be remedied with the use of buffers.

<u>Memory Management</u>:

Pixel Address Map

The 16x16 CCD image pixel mapping corresponds to how data is read from the ADNS-2051, and how it physically corresponds to the captured image. The captured image is addressed from bottom right (first pixel), proceeding upwards with increasing address, eventually ending at the top left (last pixel) of the matrix shown below. Eight bits are required to represent all 256 pixels.


LAST PIXEL

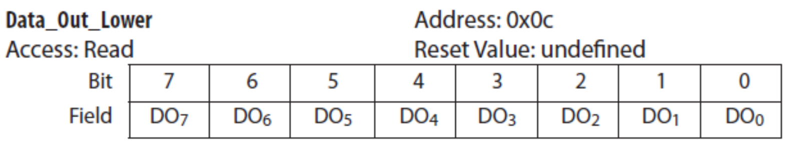
FF	EF	DF	CF	BF	AF	9F	8F	7F	6F	5F	4F	3F	2F	1F	0F
FE	EE	DE	CE	BE	AE	9E	8E	7E	6E	5E	4E	3E	2E	1E	0E
FD	ED	DD	CD	BD	AD	9D	8D	7D	6D	5D	4D	3D	2D	1D	0D
FC	EC	DC	CC	BC	AC	9C	8C	7C	6C	5C	4C	3C	2C	1C	0C
FB	EB	DB	CB	BB	AB	9B	8B	7B	6B	5B	4B	3B	2B	1B	0B
FA	EA	DA	CA	BA	AA	9A	8A	7A	6A	5A	4A	3A	2A	1A	0A
F9	E9	D9	C9	B9	A9	99	89	79	69	59	49	39	29	19	09
F8	E8	D8	C8	B8	A8	98	88	78	68	58	48	38	28	18	08
F7	E7	D7	C7	B7	A7	97	87	77	67	57	47	37	27	17	07
F6	E6	D6	C6	B6	A6	96	86	76	66	56	46	36	26	16	06
F5	E5	D5	C5	B5	A5	95	85	75	65	55	45	35	25	15	05
F4	E4	D4	C4	B4	A4	94	84	74	64	54	44	34	24	14	04
F3	E3	D3	C3	B3	A3	93	83	73	63	53	43	33	23	13	03
F2	E2	D2	C2	B2	A2	92	82	72	62	52	42	32	22	12	02
F1	E1	D1	C1	B1	A1	91	81	71	61	51	41	31	21	11	01
F0	E0	D0	C0	B0	A0	90	80	70	60	50	40	30	20	10	00

FIRST PIXEL

Figure 3: 16x16 CCD image, physical pixel map and addresses.

Registers

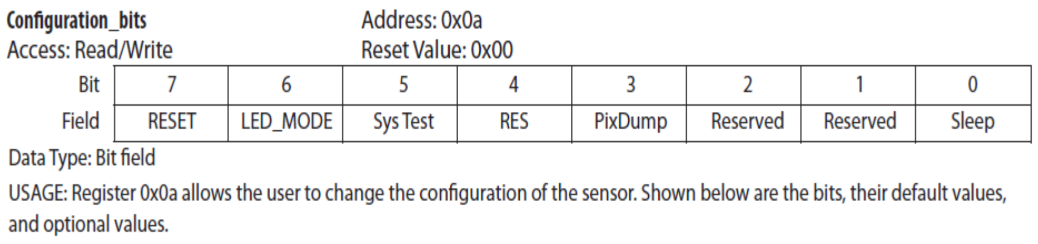
The Data_Out_Lower register holds the values associated with the current pixel address being read. The most significant bit (MSB) of this register holds a flag that indicates whether the data is valid (meaning if it’s currently being read or not) - it is high when invalid. Once a read is completed, the register is loaded up with the next pixel value and the most significant bit is set back to low, indicating that the first six bits are ready to be read once again. This cycle continues until the entire pixel map is handled:


Data_Out_Lower Address: 0x0c
Access: Read Reset Value: undefined

Bit	7	6	5	4	3	2	1	0
Field	DO_7	DO_6	DO_5	DO_4	DO_3	DO_2	DO_1	DO_0

Figure 4: Data_Out_Lower Register.

The Configuration_bits register allows us to trigger the pixel dump of the pixel array map:



Configuration_bits Address: 0x0a

Access: Read/Write Reset Value: 0x00

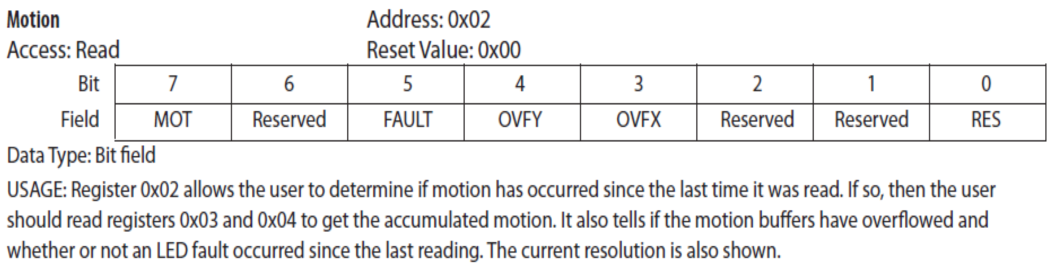
Bit	7	6	5	4	3	2	1	0
Field	RESET	LED_MODE	Sys Test	RES	PixDump	Reserved	Reserved	Sleep

Data Type: Bit field

USAGE: Register 0x0a allows the user to change the configuration of the sensor. Shown below are the bits, their default values, and optional values.

Figure 5: Configuration_bits register.

Because we don't want to waste buffer memory on pixel samples which match exactly previous pixel samples (if the mouse has not moved since last polling), we will need to use the motion register to first determine if any motion has occurred before reading:



Motion Address: 0x02

Access: Read Reset Value: 0x00

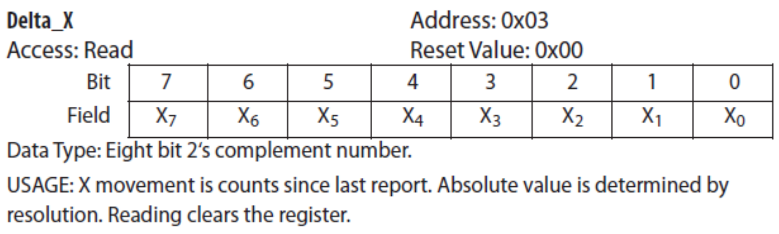
Bit	7	6	5	4	3	2	1	0
Field	MOT	Reserved	FAULT	OVFY	OVFX	Reserved	Reserved	RES

Data Type: Bit field

USAGE: Register 0x02 allows the user to determine if motion has occurred since the last time it was read. If so, then the user should read registers 0x03 and 0x04 to get the accumulated motion. It also tells if the motion buffers have overflowed and whether or not an LED fault occurred since the last reading. The current resolution is also shown.

Figure 6: Motion register.

The Delta_X register provides a signed representation of the amount of x-axis movement that the mouse experienced since last polling:



Delta_X Address: 0x03

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0

Data Type: Eight bit 2's complement number.

USAGE: X movement is counts since last report. Absolute value is determined by resolution. Reading clears the register.

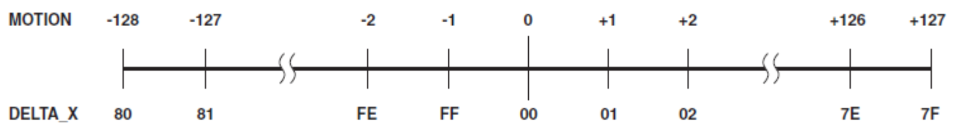
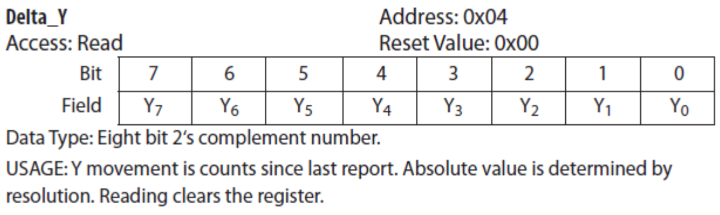


Figure 7: Delta_X Register.

The Delta_Y register provides a signed representation of the amount of y-axis movement that the mouse experienced since last polling:



Delta_Y Address: 0x04

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

Data Type: Eight bit 2's complement number.

USAGE: Y movement is counts since last report. Absolute value is determined by resolution. Reading clears the register.

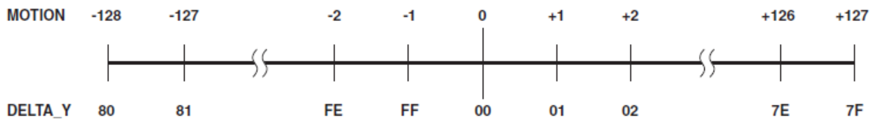


Figure 8: Delta_Y Register.

Pixel Sample Buffer Queue

In order to deal with the timing issues inherent in continual dumping of the pixels (the rest of the system may not be able to keep up), some sort of backlog or history of pixel samples will be necessary so that they may be referenced later. For this purpose, we have designed the concept of a pixel sample buffer queue. One can picture this by thinking of each pixel sample as a piece of a jigsaw puzzle, and the buffer queue as being a constantly shifting collection of these jigsaw pieces placed one atop the other. As the system continues to poll the mouse, additional jigsaw pieces are placed on top of this collection (assuming the mouse has moved). Simultaneously, jigsaw pieces are being pulled from the bottom of the collection and being placed one by one into the main puzzle, or the aggregate image of the scan. This is basically a stack.

Numerical Analysis

Because the image being pulled from the CCD is in gray scale, we will only require 6 bits per pixel for each of our pixel samples. The video frame buffer, or the aggregate image, is 128 by 128 pixels, meaning it will support 98,304 bits and 12,288 bytes. Since we are also displaying a small inlet image corresponding to the current pixel sample, we will need an additional 16 x 16 pixels = 1536 bits = 192 bytes. This comes out to a total of 12,288 + 192 = 12,480 bytes. We now need to add to this the memory required by the pixel sample buffer queue. We intend for the queue to hold at most five pixel samples - this corresponds to 192 x 5 = 960 bytes, plus one more for the 8 bits which correspond to the serial data input from the mouse peripheral. Adding this to our previous total, we get a grand total of 13,441 bytes.

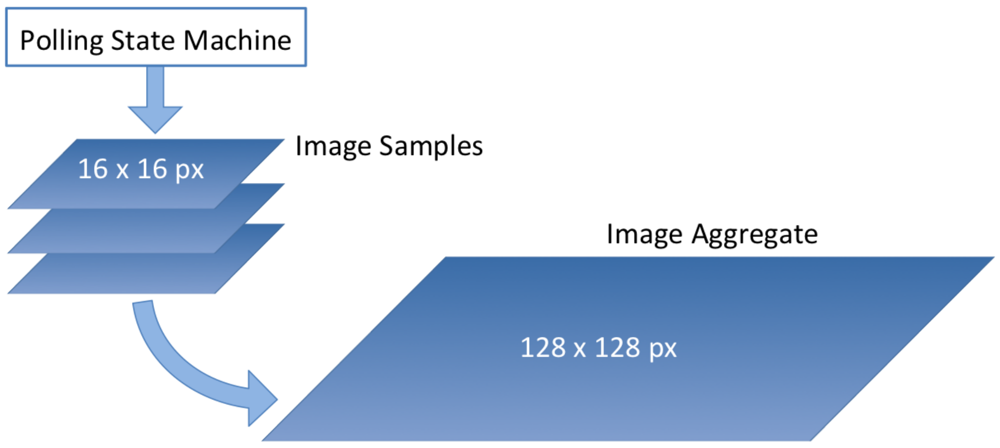


Figure 9: The polling state machine generates image samples, which are then processed into the image aggregate.

Pixel Writing

Pixel dumps from the mouse hardware yield 6-bit gray scale, which does not match the 10-bit, 3-channel (red, green, blue) VGA display driver. To remedy converting 6-bit gray scale to 10-bit RBG, it is possible to write all 3 channels with the same gray scale information. To extend the 6-bit gray scale to 10-bits, we consider which bits of the gray scale have the most control of the overall gray scale pixel. Considering that the most-significant bits cause the greatest variation in the gray scale pixel, it is reasonable to duplicate lower significant bits to extend 6-bit gray scale to 10-bit gray scale.

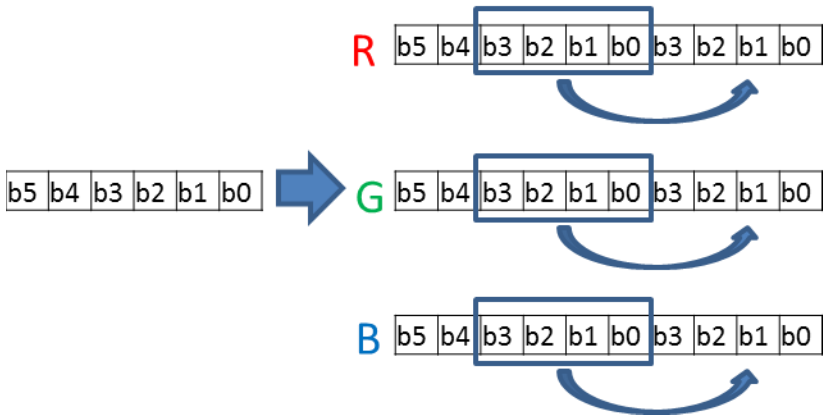

R
G
B
b5 b4 b3 b2 b1 b0
b5 b4 b3 b2 b1 b0 b3 b2 b1 b0
b5 b4 b3 b2 b1 b0 b3 b2 b1 b0
b5 b4 b3 b2 b1 b0 b3 b2 b1 b0

Figure 10: Gray scale operations

Aggregate and Inset Image Dynamics

There are essentially two groups of registers which handle all the imaging data required for display. The first group of registers handles the 16 x 16 pixel inset screen which displays the current pixel dump of whatever image the mouse is sitting on. The second group of registers handles the 128 x 128 pixel accumulation of all these pixel dumps, which is built up as the user progresses over the to-be-scanned image.

This procedure is handled with a relatively simple algorithm:

```
while (true){
    //we first fetch the front most pixel sample in the buffer queue
    pixelDump = fetchNewPixelDump();

    //we get the x coordinate corresponding to the center of the pixel sample
    pixelDumpCenterXCoordinate = getXCoordinatePixelDumpCenter(pixelDump);

    //we get the y coordinate corresponding to the center of the pixel sample
    pixelDumpCenterYCoordinate = getYCoordinatePixelDumpCenter(pixelDump);

    //we add the pixel sample to the aggregate in the location corresponding to the center of the
    sample
    aggregateImage.addToAggregateImage(pixelDumpCenterXCoordinate,
    pixelDumpCenterYCoordinate, pixelDump);
}
```

The image below essentially graphically illustrates what the algorithm above does. The pixel doubling is essentially a magnification of the 128 x 128 for a better UI experience.

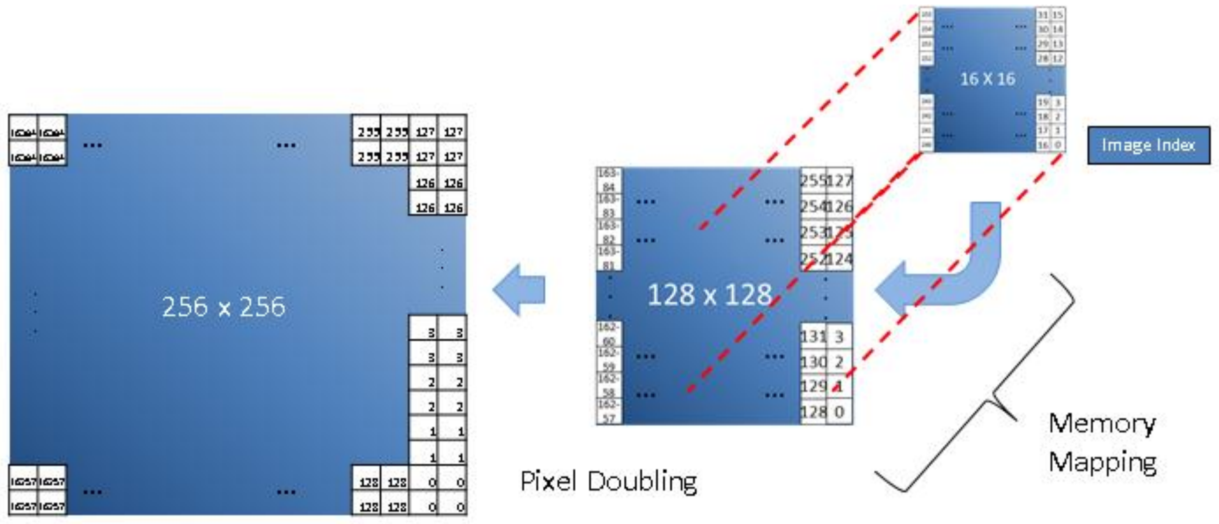


Figure 11: Memory Mapping

Peripherals

Optical Mouse Input Peripheral and GPIO

To connect to the optical mouse hardware peripheral shown in Figure 11, we must use the general purpose input output (GPIO) peripheral. The GPIO peripheral provides several configurable analog and digital IO pins for use with external hardware. These pins are configurable as inputs, outputs, or inouts, and can be mapped to memory. To connect the image, position, and set/reset information for capturing and displaying images to the VGA display, we required connecting several pins from the optical mouse's ADNS-2051 processor to the Altera DE2 Board via GPIO.

Five GPIO pins are required to connect to SCLK, PD, and SDIO on the ADNS-2051, and the left and right click buttons (L/R) of the mouse. These serial pins are addressed according to their physical location on a 40-pin connector on the DE2. The exact address mapping—which pins we will use—is still under consideration.

Each pin is a serial connection—SCLK and SDIO are part of a serial peripheral interface (SPI) protocol connection/communication, and PD, L, and R are enabled for the polling state machine. PD is a device enable that is always set high—some optical mice can use this enable to enable low power mode. L and R will be low/high depending on if the left and right mouse buttons are pressed/not pressed (active low). The GPIO will communicate all values obtained/written to its pins via the Avalon Bus to/with other peripherals. These communications include sending PD, L, and R to the polling state machine, and writing/reading data to and from SRAM over SDIO. The typical operating frequency for the serial clock port on the ADNS-2051 is 4.5 MHz, which means that a PLL will be used to communicate a 4.5MHz clock over SCLK on the GPIO to the ADNS-2051. Figure 12 shows the interconnection of GPIO to the mouse hardware.

Left and right click controls three states of operation for the system: idle, scanning, and reset. When the left button is pressed on the mouse, the mouse will enter a scanning state, where image and position information is relayed through the GPIO to SRAM. When the right mouse button is pressed, the system resets, clearing the VGA display—this takes precedence over left click. When neither button is pressed, the system is in an idle state, where is does not poll the ADNS-2051 for new image or position data. These states are shown in Figure 13.

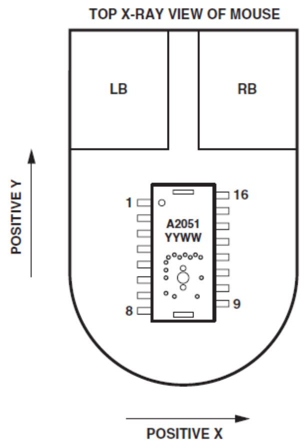


Figure 12: Mouse peripheral with embedded ADNS-2051 optical processor.

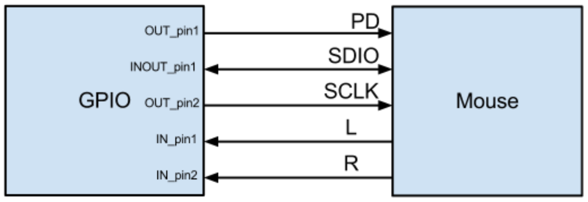


Figure 13: Mouse peripheral hardware to GPIO interconnection.

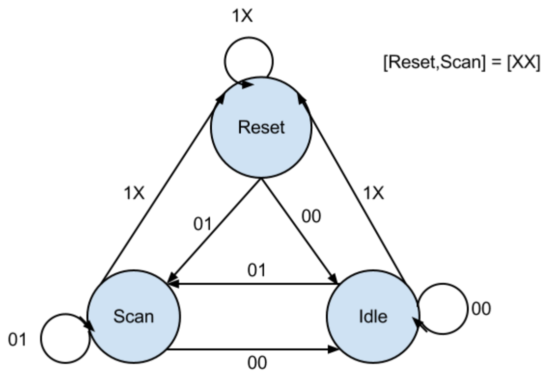


Figure 14: FSM for left and right click.

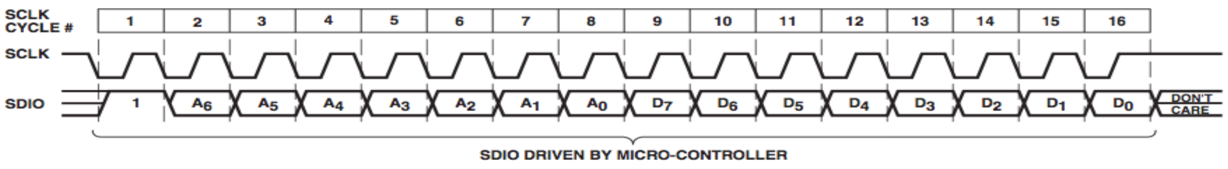


Figure 15: Datasheet Waveform

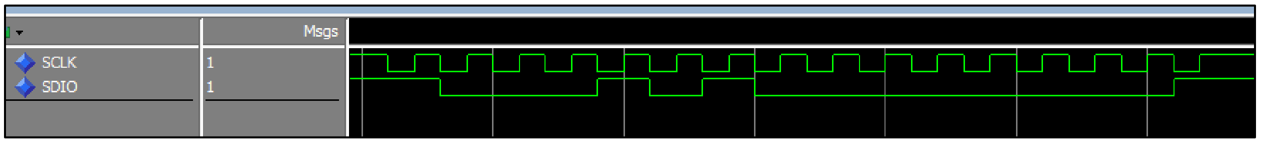


Figure 16: Simulated Waveforms

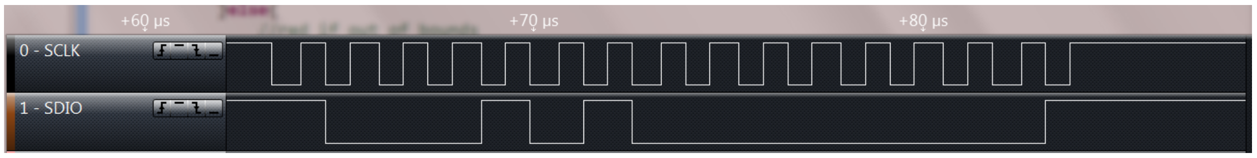


Figure 17: Captured Waveforms using Logic Analyzer

The logic analyzer used to capture the above waveform is the Saleae Logic8 analyzer. While debugging, to ensure that the FSM was operating as required, we needed to probe all connections with the logic analyzer. The Saleae enabled real time sampling of all hardware connections. We used this sampling to determine the function of our synthesized code and whether or not it was operating correctly. This debugging was crucial in ensuring the proper operation of our FSM.

Polling State Machine

To interface with the mouse peripheral a polling state machine (PSM) is implemented in hardware on the DE2 board. The details of the PSM are given in Figure 14. As described before, the PSM communicates with the ADNS-2051 using the GPIO peripheral of the DE2. Implemented in hardware will be a serial peripheral interface (SPI) protocol for writing to and reading the registers of the ADNS-2051. The registers of interest have already been described in the prior text. As shown in Figure 14, the main loop of the PSM continually polls the Motion register of the mouse. Within the Motion register, the MOT bit is raised high to indicate that the mouse has moved. The PSM then reads the relative X and Y movement of the mouse and transcribes the information to a new image sample stored in the SRAM. The PixDump bit in the Pixel Dump register is then set high to initiate the Pixel Dump from the ADNS-2051. As described before, the Data_Out_Lower register will continually feed the progressing pixel values for the pixel map (Figure 3). These pixel values are stored in the appropriate location of the image sample in the SRAM. Once the full pixel map has been read the PixDump bit is reset and the loop reiterates.

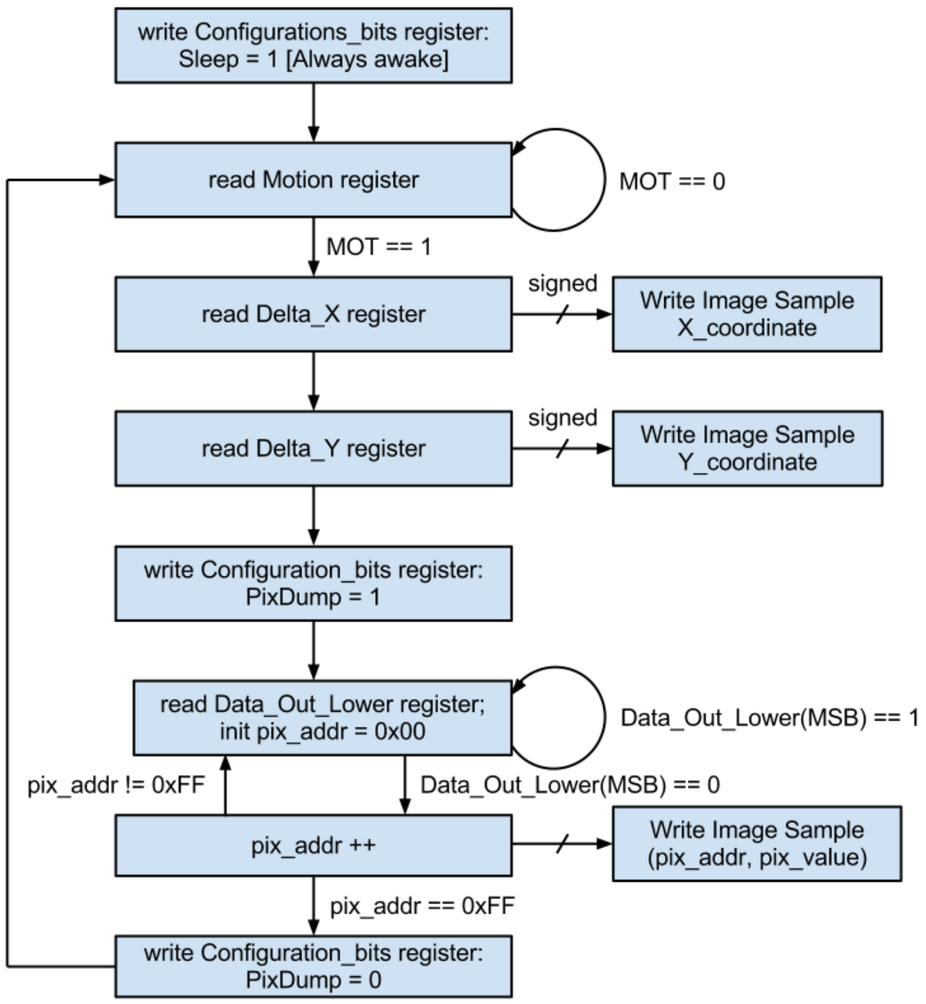

write Configurations_bits register: Sleep = 1 [Always awake]
read Motion register
MOT == 0
MOT == 1
read Delta_X register
signed
Write Image Sample X_coordinate
read Delta_Y register
signed
Write Image Sample Y_coordinate
write Configuration_bits register: PixDump = 1
read Data_Out_Lower register; init pix_addr = 0x00
Data_Out_Lower(MSB) == 1
pix_addr != 0xFF
Data_Out_Lower(MSB) == 0
pix_addr ++
Write Image Sample (pix_addr, pix_value)
pix_addr == 0xFF
write Configuration_bits register: PixDump = 0

Figure 18: Algorithm for acquiring image samples from ADNS-2051

VGA Monitor Output

To display the image information on the VGA display, we require a VGA raster controller similar to the one implemented in Lab 3. The controller will update pertinent VGA signals, including the clock (~25 MHz, H_SYNC, V_SYNC, BLANK, SYNC, and the RGB level of the current VGA pixel. Once implemented in hardware (via VHDL), this peripheral will be controlled with software that updates the VGA signals. The VGA signals are stored and accessed as registers and/or counters. This peripheral communicates to the SRAM peripheral via the Avalon Bus.

Input obtained from the SRAM includes XY location and the current gray scale pixel value. Software is required to convert the gray scale pixel to 10-bit RGB (as required by the VGA hardware controller) and associate this pixel with its XY location according to the VGA signals. XY location is translated to VGA signals via counters for horizontal and vertical position. These counters, and other internal signals required for translating XY position to VGA display position are internal to the VGA controller. The following diagram outlines the basic VGA controller structure:

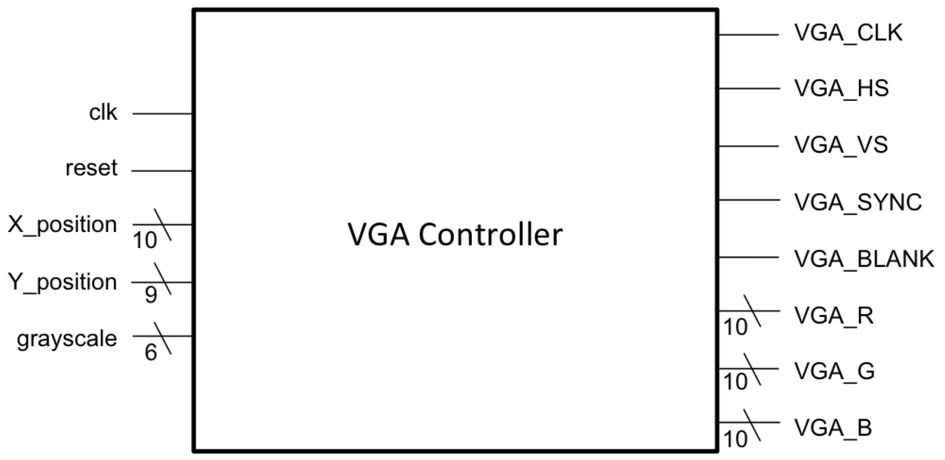


Figure 19: Block diagram of the VGA module.

Finite State Machines

The state diagrams for the FSM are detailed in the next figure. The first figure features a condensed version of the full FSM. Subsequent images break up the sub-components into their full form. There are 75 states in total controlling the SCLK, SDIO, and PD lines interfacing with the optical processor, and writing retrieved image data out to the main hardware controller.

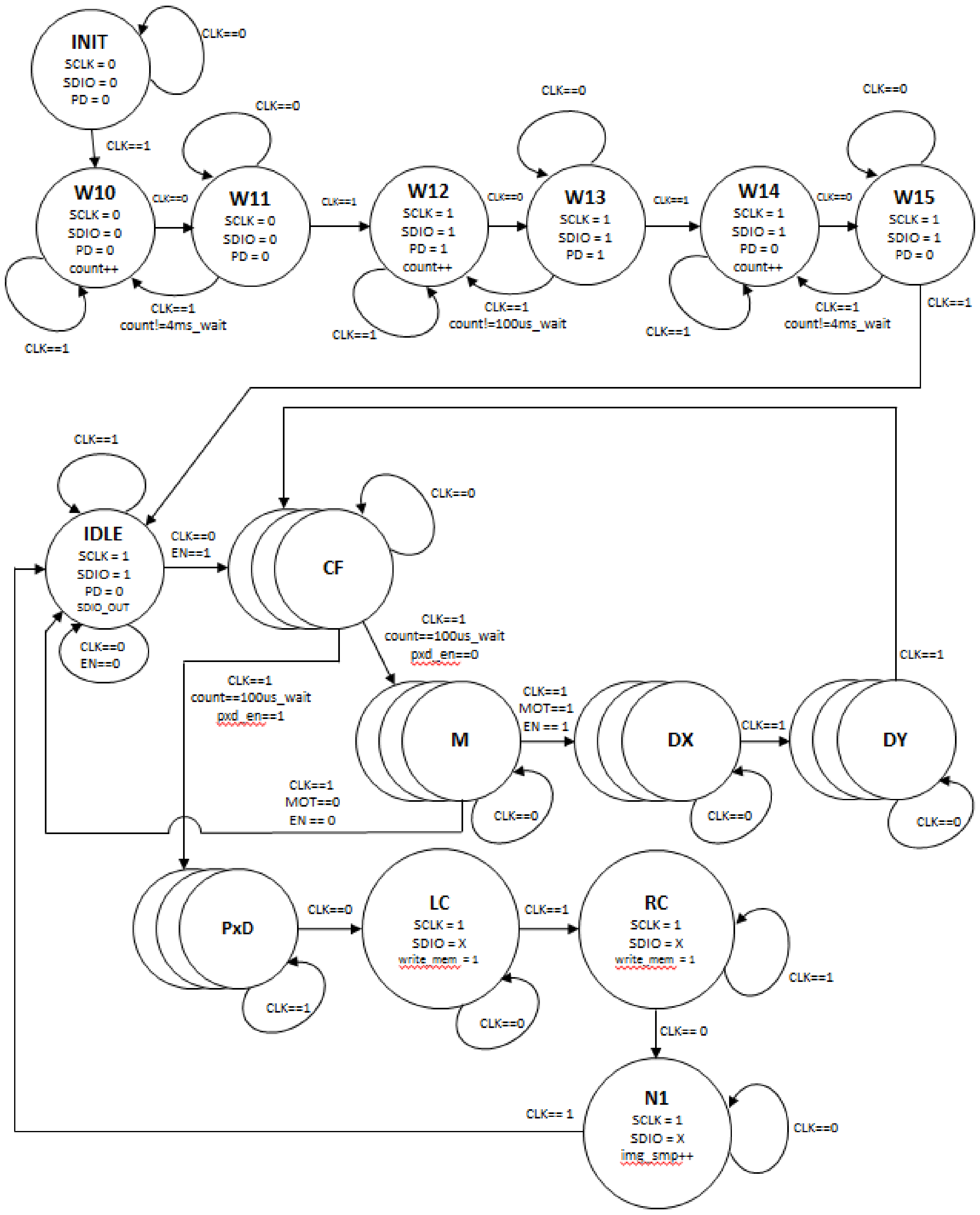


Figure 20: Finite State Machine 1

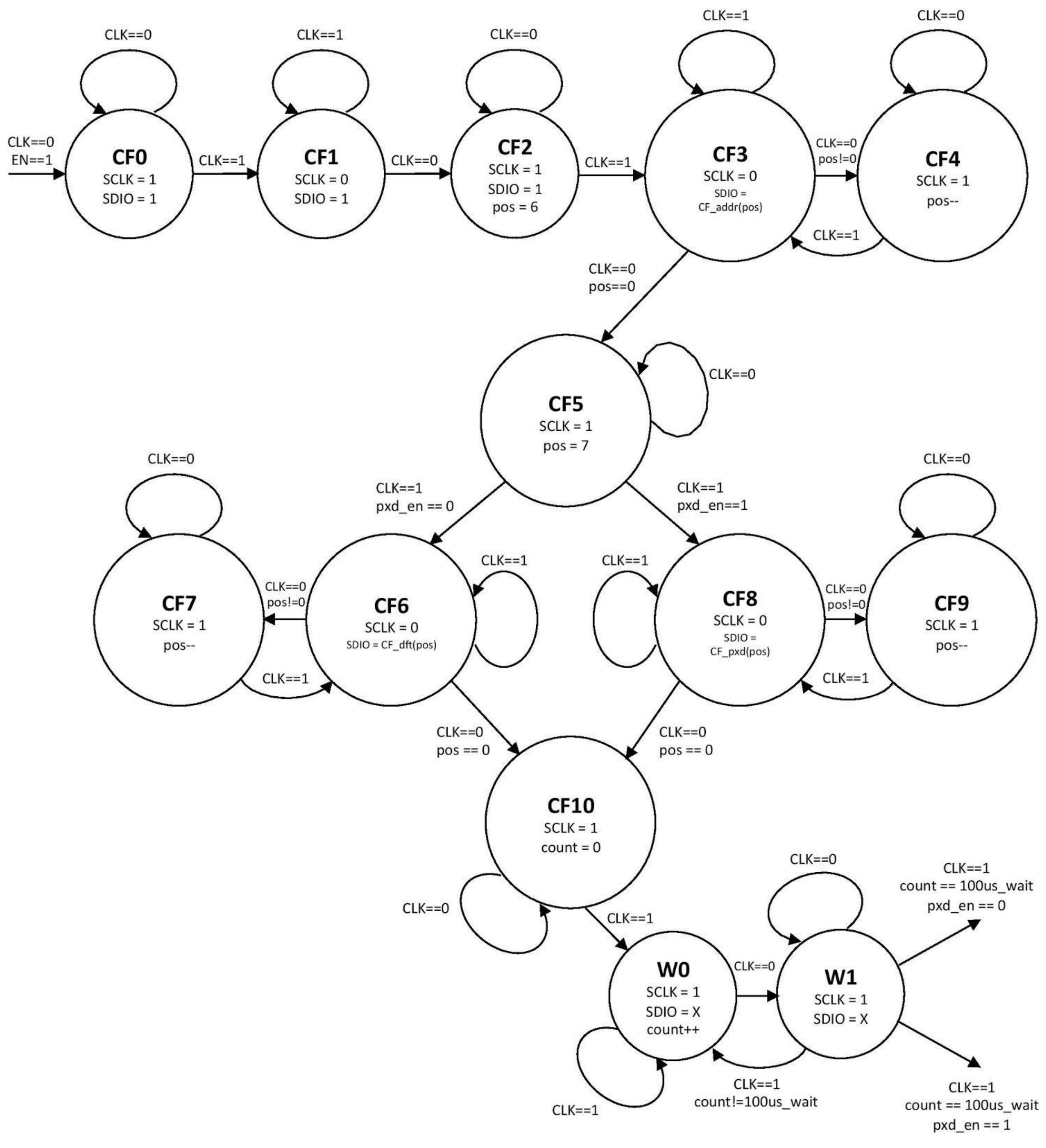


Figure 21: Finite State Machine 2

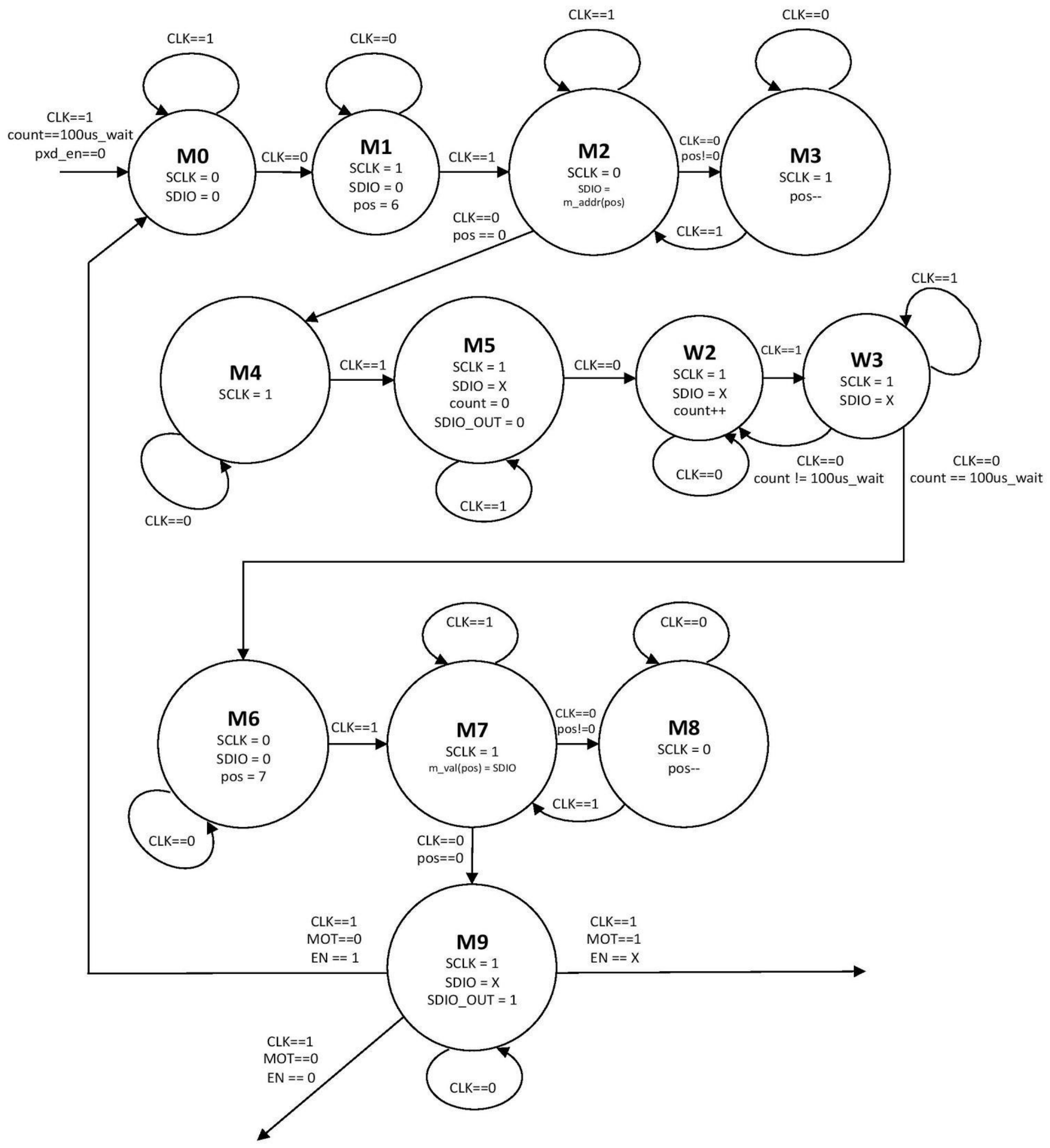


Figure 22: Finite State Machine 3

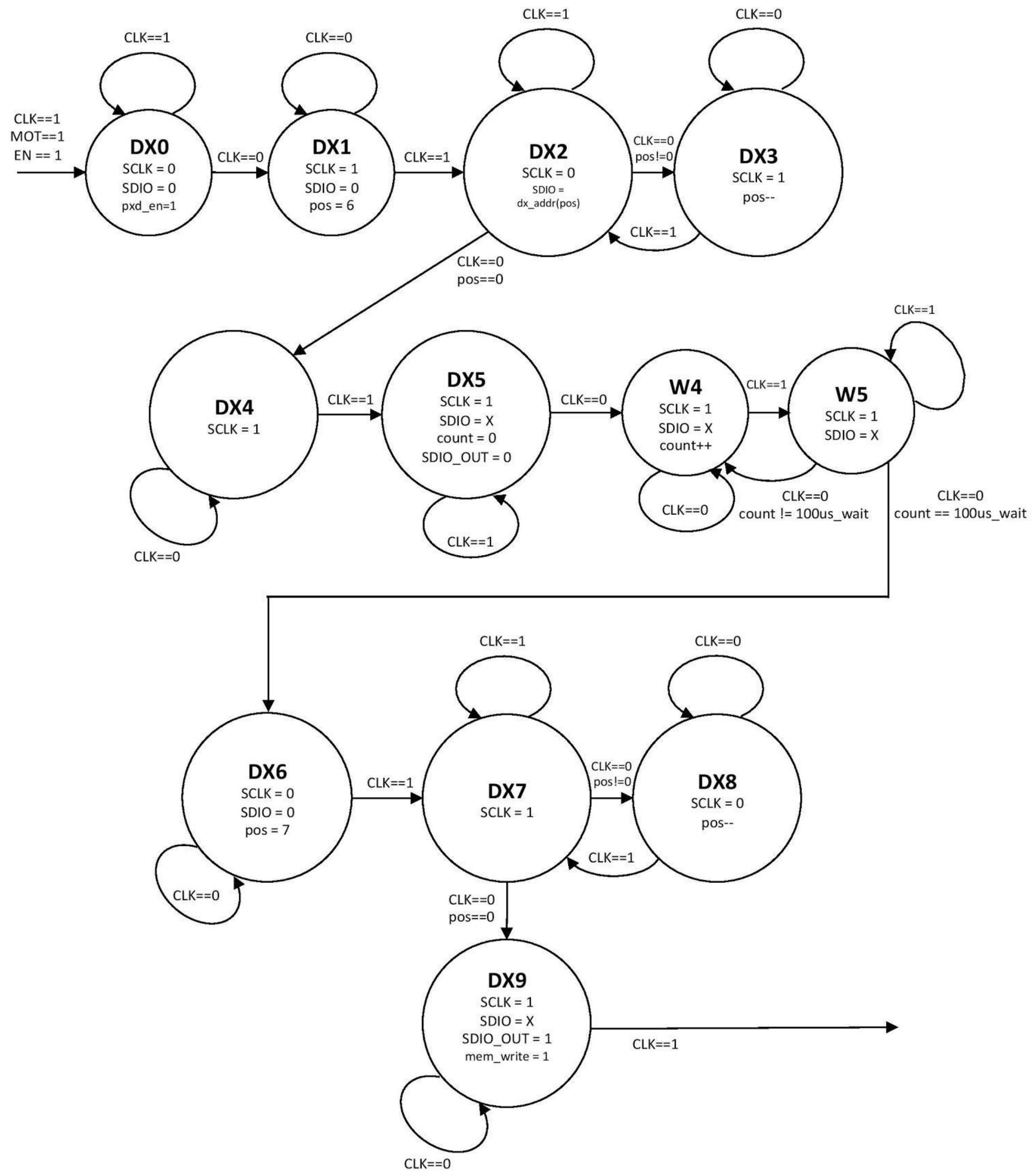


Figure 23: Finite State Machine 4

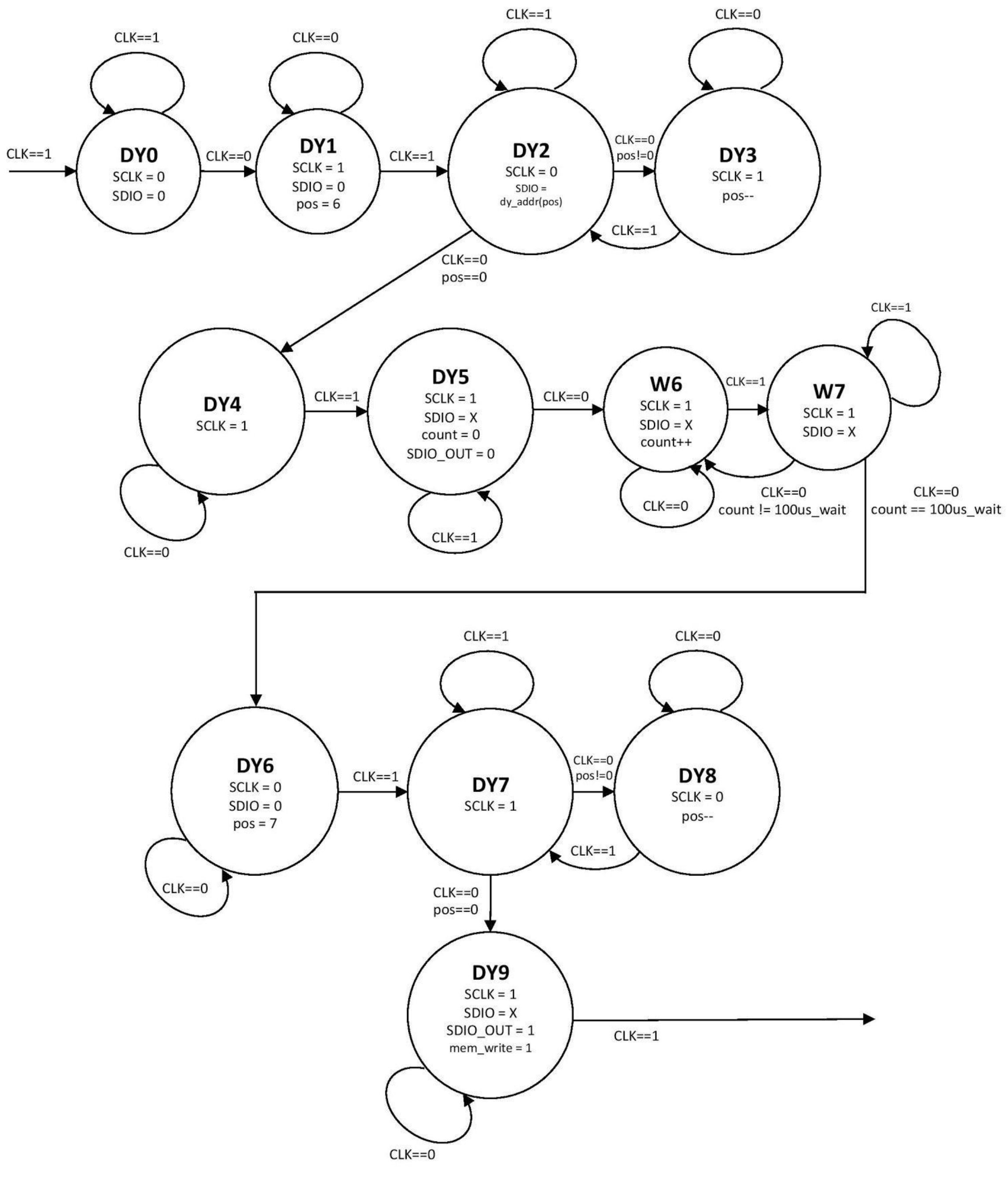


Figure 24: Finite State Machine 5

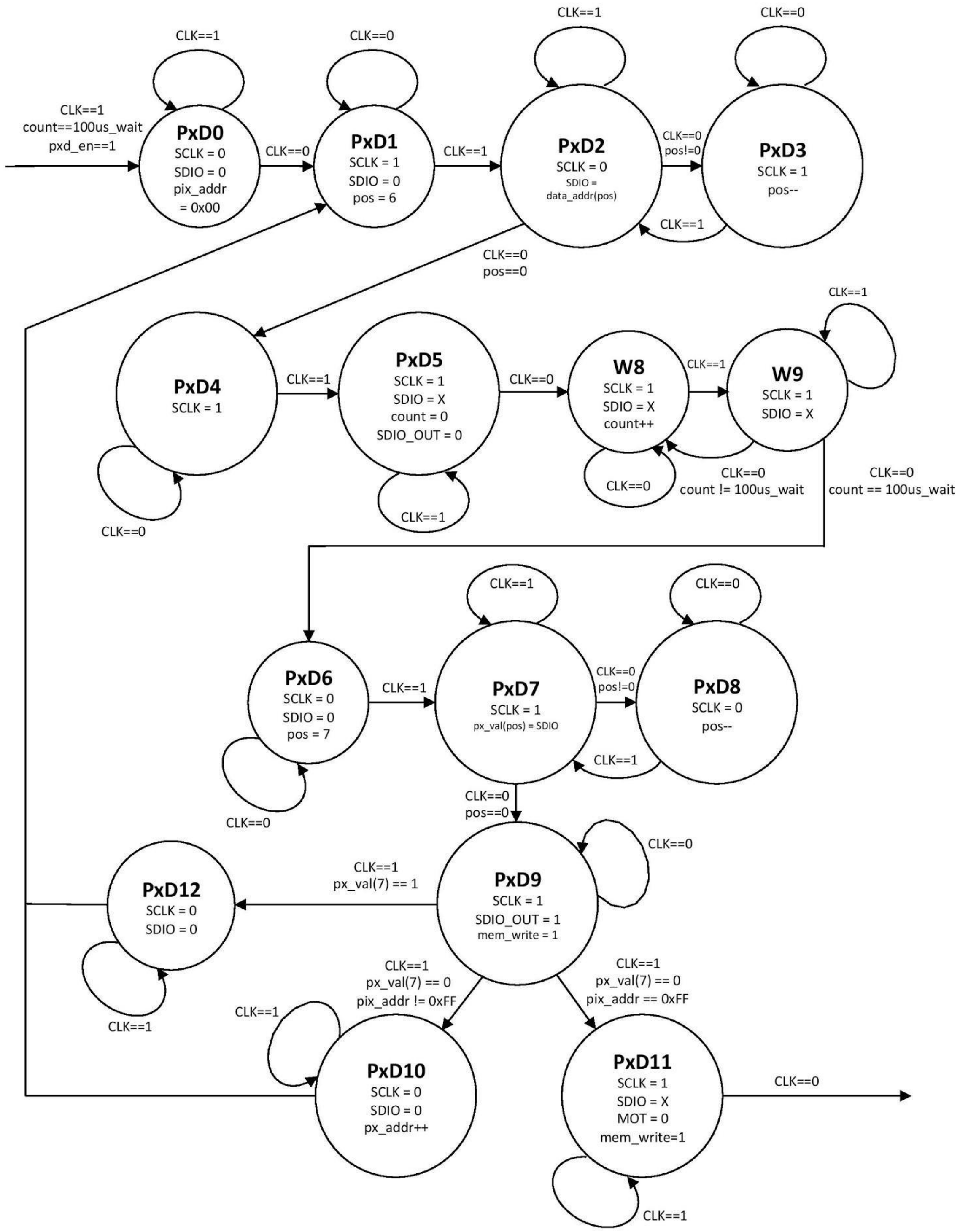


Figure 25: Finite State Machine 6

Inside the FPGA: RTL Viewer

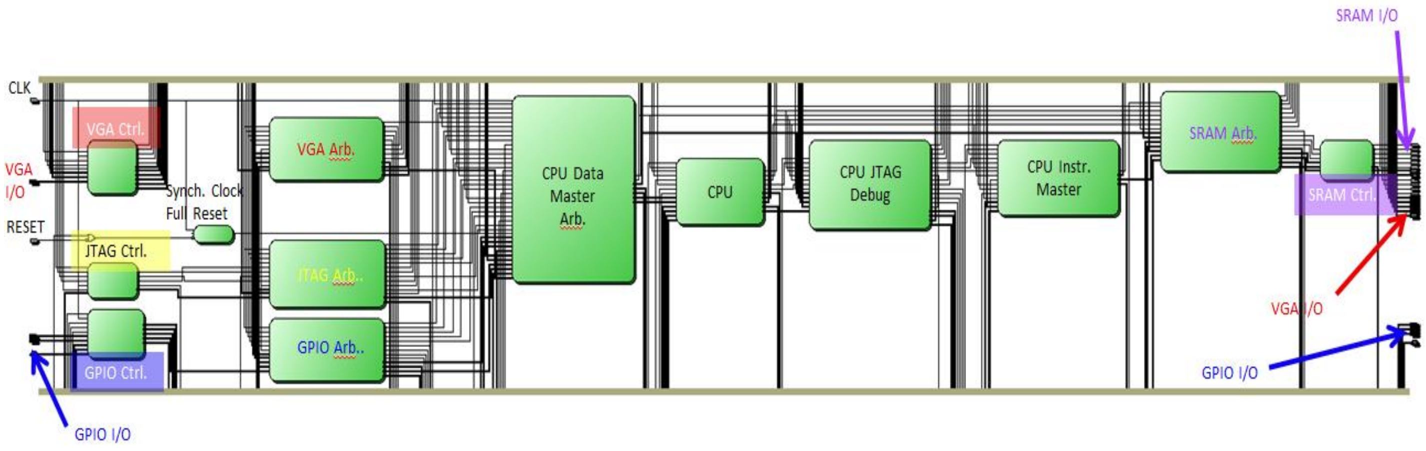


Figure 26: Top Level RTL View

VGA Controller

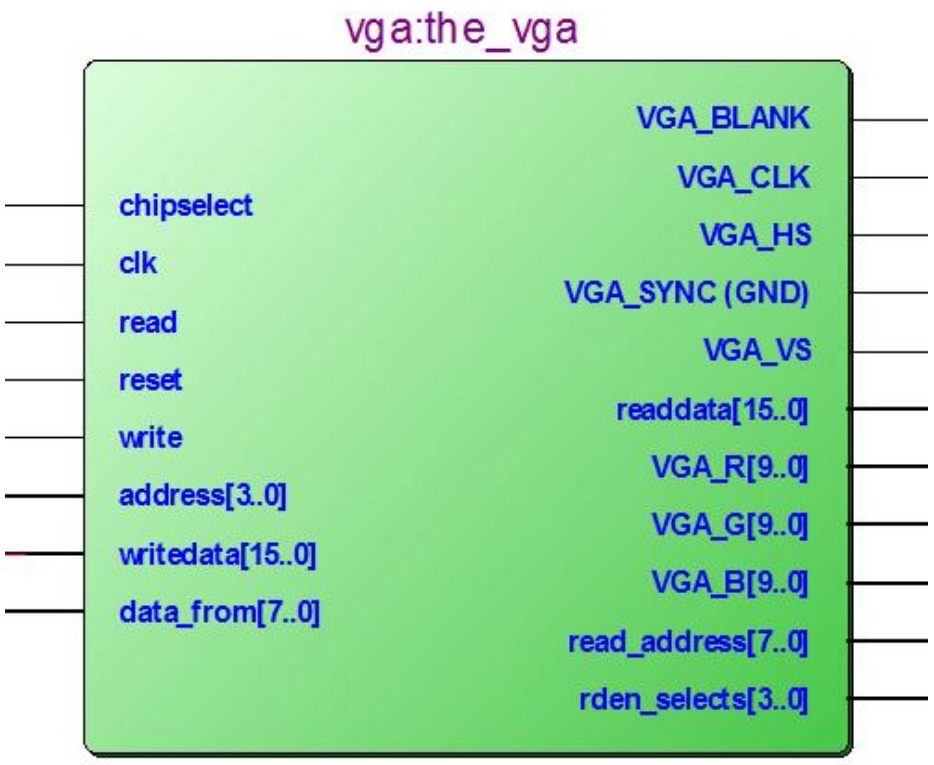


Figure 27: VGA Controller

JTAG Controller

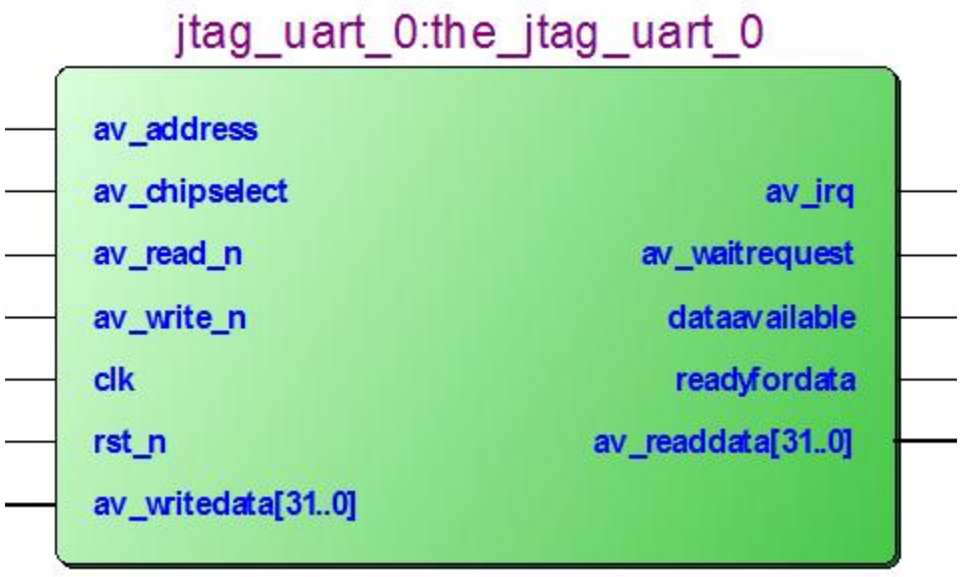


Figure 28: JTAG Controller

GPIO Controller

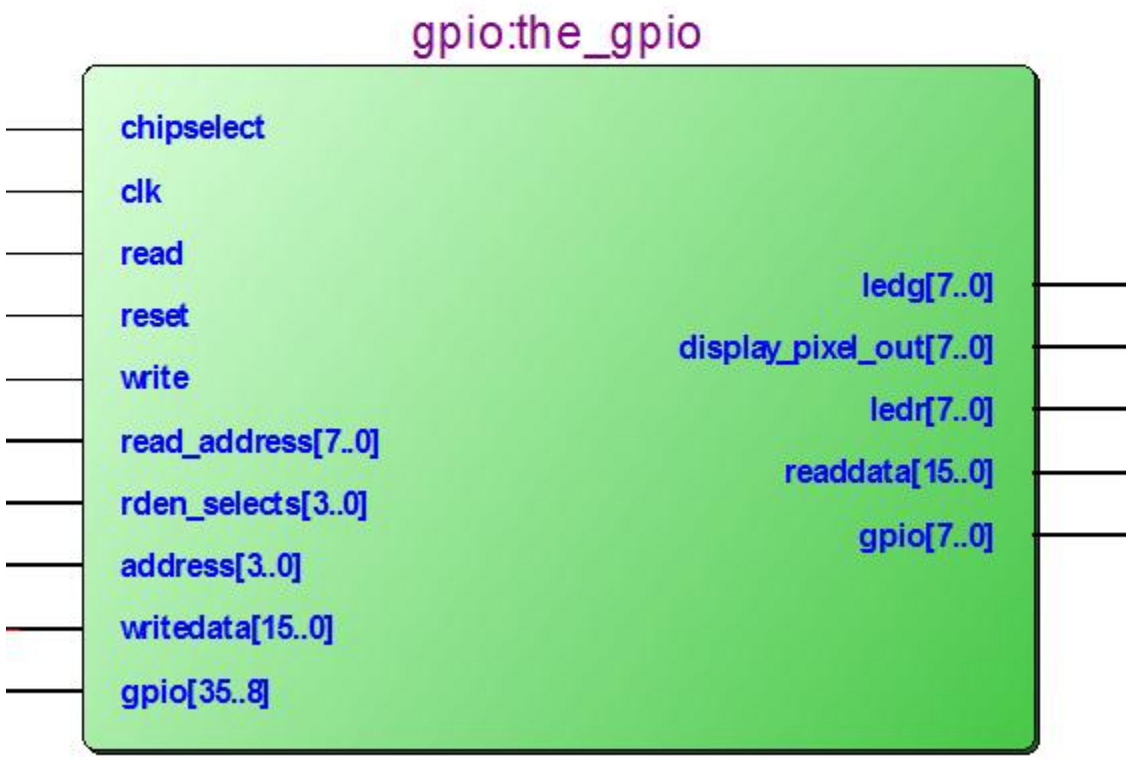


Figure 29: GPIO Controller

Synchronization Clock Full Reset

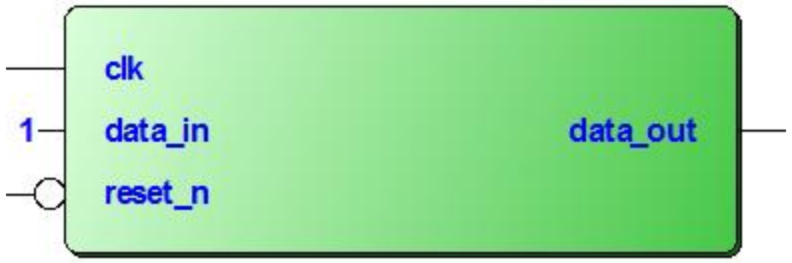


Figure 30: Sync Clock Full Reset

CPU Instruction Master Arbitrator

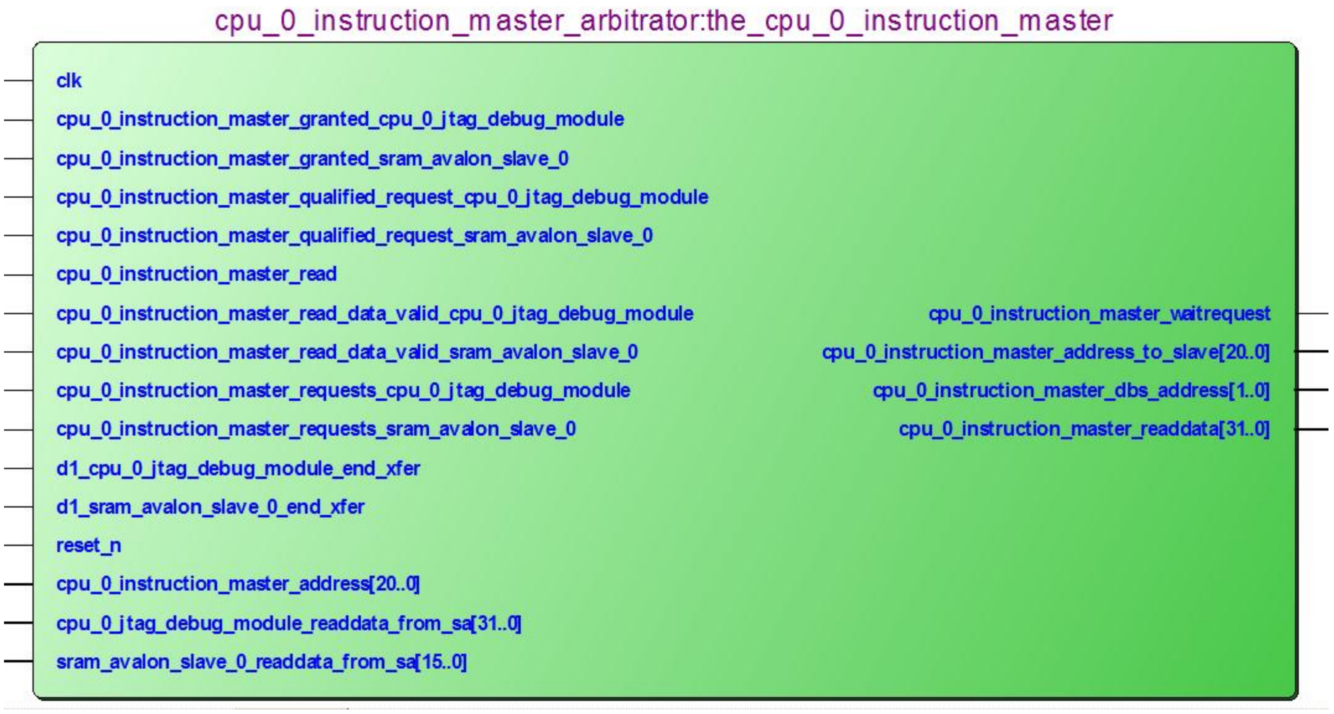

cpu_0_instruction_master_arbitrator:the_cpu_0_instruction_master
clk
cpu_0_instruction_master_granted_cpu_0_jtag_debug_module
cpu_0_instruction_master_granted_sram_avalon_slave_0
cpu_0_instruction_master_qualified_request_cpu_0_jtag_debug_module
cpu_0_instruction_master_qualified_request_sram_avalon_slave_0
cpu_0_instruction_master_read
cpu_0_instruction_master_read_data_valid_cpu_0_jtag_debug_module
cpu_0_instruction_master_read_data_valid_sram_avalon_slave_0
cpu_0_instruction_master_requests_cpu_0_jtag_debug_module
cpu_0_instruction_master_requests_sram_avalon_slave_0
d1_cpu_0_jtag_debug_module_end_xfer
d1_sram_avalon_slave_0_end_xfer
reset_n
cpu_0_instruction_master_address[20..0]
cpu_0_jtag_debug_module_readdata_from_sa[31..0]
sram_avalon_slave_0_readdata_from_sa[15..0]
cpu_0_instruction_master_waitrequest
cpu_0_instruction_master_address_to_slave[20..0]
cpu_0_instruction_master_dbs_address[1..0]
cpu_0_instruction_master_readdata[31..0]

Figure 31: CPU Instruction Master Arbitrator

CPU JTAG Debug

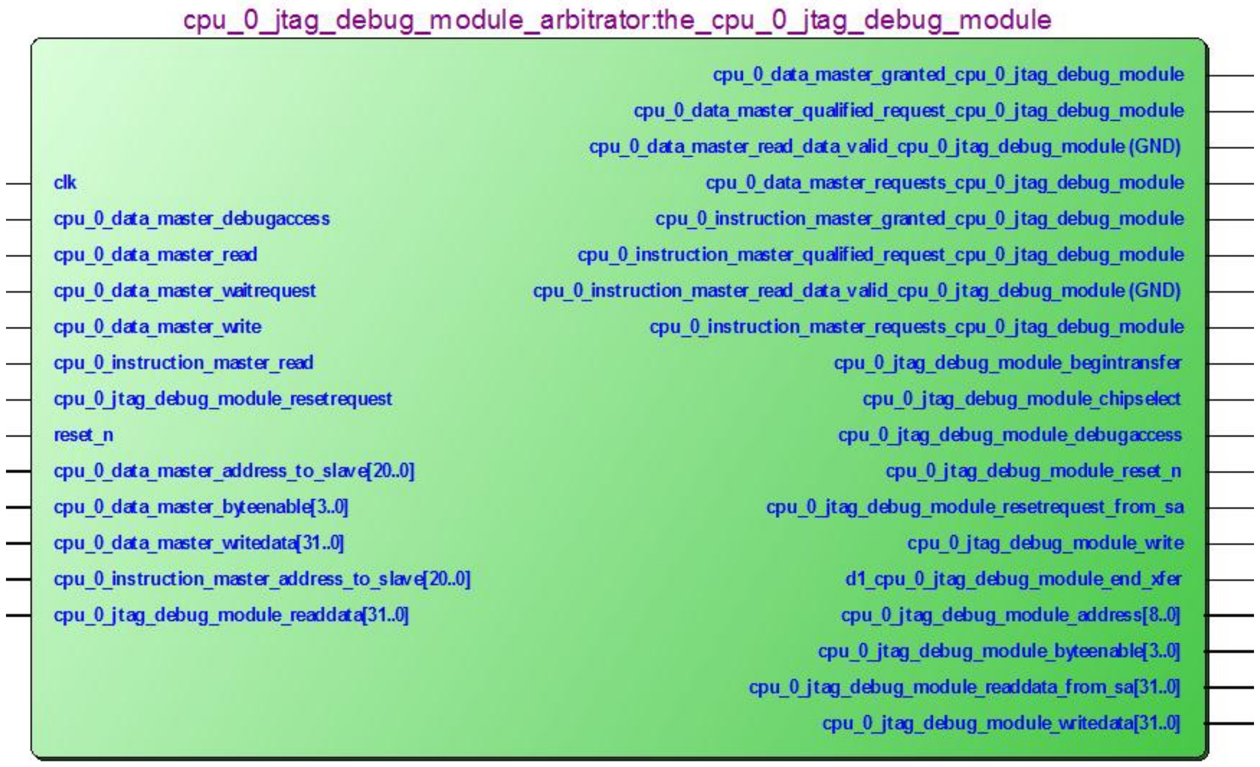

cpu_0_jtag_debug_module_arbitrator:the_cpu_0_jtag_debug_module
clk
cpu_0_data_master_debugaccess
cpu_0_data_master_read
cpu_0_data_master_waitrequest
cpu_0_data_master_write
cpu_0_instruction_master_read
cpu_0_jtag_debug_module_resetrequest
reset_n
cpu_0_data_master_address_to_slave[20..0]
cpu_0_data_master_byteenable[3..0]
cpu_0_data_master_writedata[31..0]
cpu_0_instruction_master_address_to_slave[20..0]
cpu_0_jtag_debug_module_readdata[31..0]
cpu_0_data_master_granted_cpu_0_jtag_debug_module
cpu_0_data_master_qualified_request_cpu_0_jtag_debug_module
cpu_0_data_master_read_data_valid_cpu_0_jtag_debug_module (GND)
cpu_0_data_master_requests_cpu_0_jtag_debug_module
cpu_0_instruction_master_granted_cpu_0_jtag_debug_module
cpu_0_instruction_master_qualified_request_cpu_0_jtag_debug_module
cpu_0_instruction_master_read_data_valid_cpu_0_jtag_debug_module (GND)
cpu_0_instruction_master_requests_cpu_0_jtag_debug_module
cpu_0_jtag_debug_module_begintransfer
cpu_0_jtag_debug_module_chipselect
cpu_0_jtag_debug_module_debugaccess
cpu_0_jtag_debug_module_reset_n
cpu_0_jtag_debug_module_resetrequest_from_sa
cpu_0_jtag_debug_module_write
d1_cpu_0_jtag_debug_module_end_xfer
cpu_0_jtag_debug_module_address[8..0]
cpu_0_jtag_debug_module_byteenable[3..0]
cpu_0_jtag_debug_module_readdata_from_sa[31..0]
cpu_0_jtag_debug_module_writedata[31..0]

Figure 32: CPU JTAG Debug

CPU

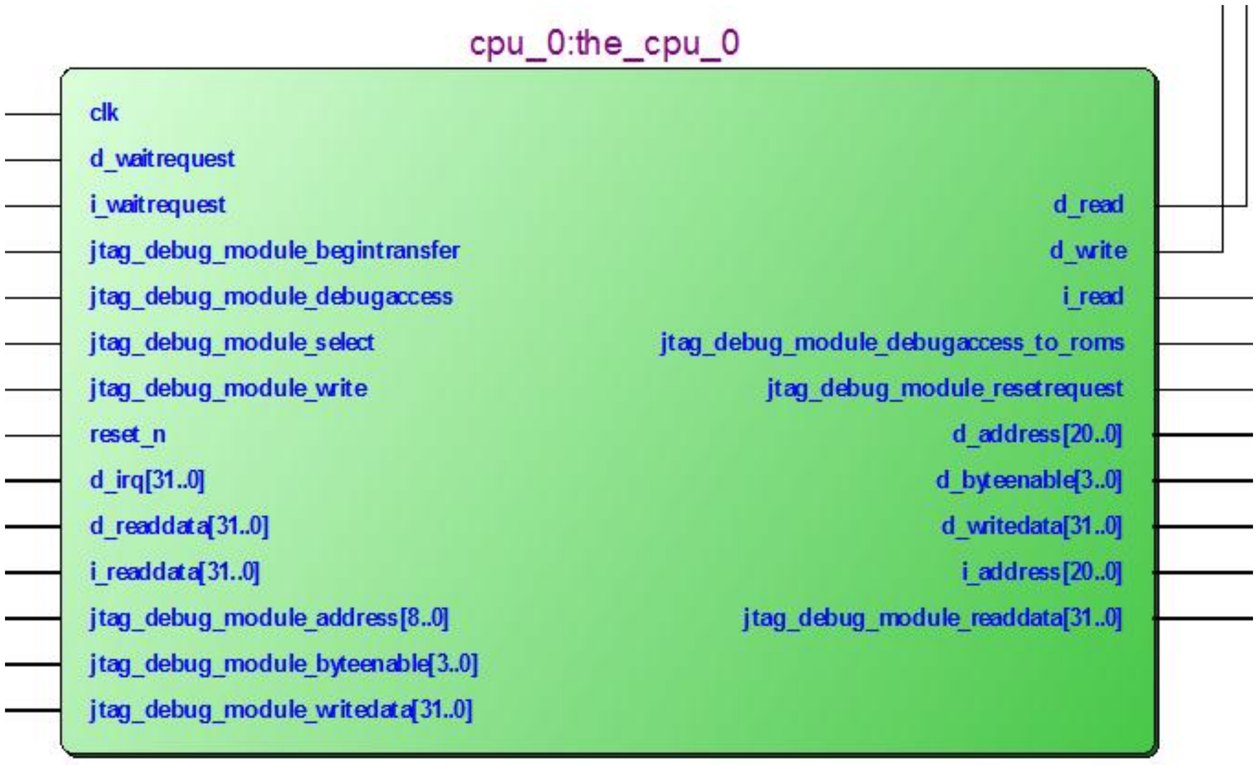

cpu_0:the_cpu_0
clk
d_waitrequest
i_waitrequest
jtag_debug_module_begintransfer
jtag_debug_module_debugaccess
jtag_debug_module_select
jtag_debug_module_write
reset_n
d_irq[31..0]
d_readdata[31..0]
i_readdata[31..0]
jtag_debug_module_address[8..0]
jtag_debug_module_byteenable[3..0]
jtag_debug_module_writedata[31..0]
d_read
d_write
i_read
jtag_debug_module_debugaccess_to_roms
jtag_debug_module_resetrequest
d_address[20..0]
d_byteenable[3..0]
d_writedata[31..0]
i_address[20..0]
jtag_debug_module_readdata[31..0]

Figure 33: CPU

VGA Arbitrator

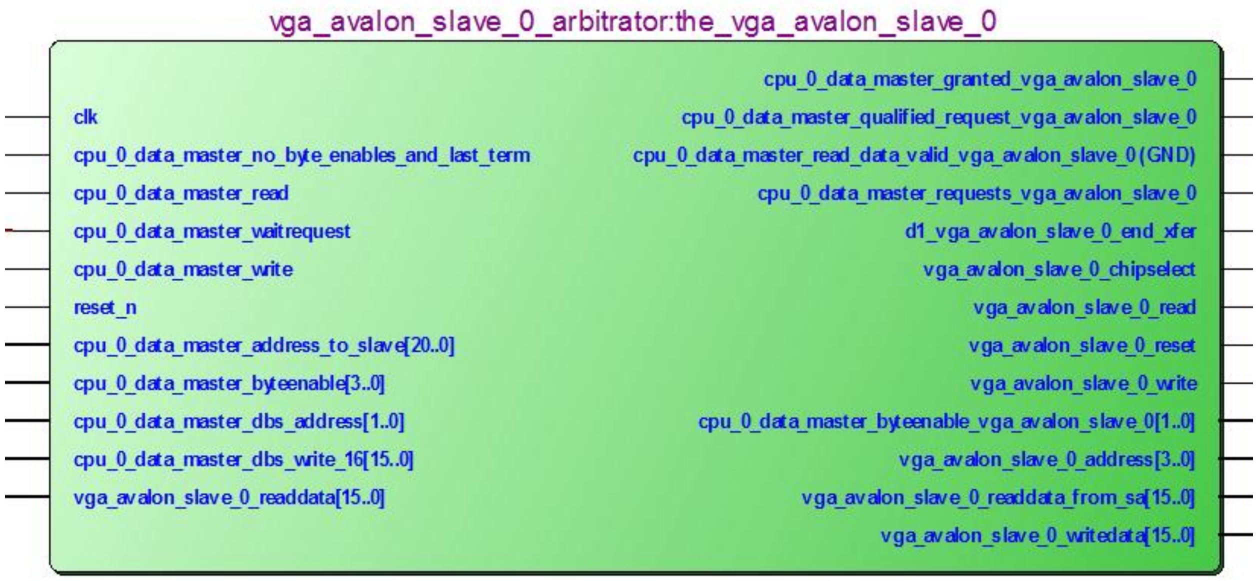

vga_avalon_slave_0_arbitrator:the_vga_avalon_slave_0
clk
cpu_0_data_master_no_byte_enables_and_last_term
cpu_0_data_master_read
cpu_0_data_master_waitrequest
cpu_0_data_master_write
reset_n
cpu_0_data_master_address_to_slave[20..0]
cpu_0_data_master_byteenable[3..0]
cpu_0_data_master_dbs_address[1..0]
cpu_0_data_master_dbs_write_16[15..0]
vga_avalon_slave_0_readdata[15..0]
cpu_0_data_master_granted_vga_avalon_slave_0
cpu_0_data_master_qualified_request_vga_avalon_slave_0
cpu_0_data_master_read_data_valid_vga_avalon_slave_0 (GND)
cpu_0_data_master_requests_vga_avalon_slave_0
d1_vga_avalon_slave_0_end_xfer
vga_avalon_slave_0_chipselect
vga_avalon_slave_0_read
vga_avalon_slave_0_reset
vga_avalon_slave_0_write
cpu_0_data_master_byteenable_vga_avalon_slave_0[1..0]
vga_avalon_slave_0_address[3..0]
vga_avalon_slave_0_readdata_from_sa[15..0]
vga_avalon_slave_0_writedata[15..0]

Figure 34: VGA Arbitrator

GPIO Arbitrator

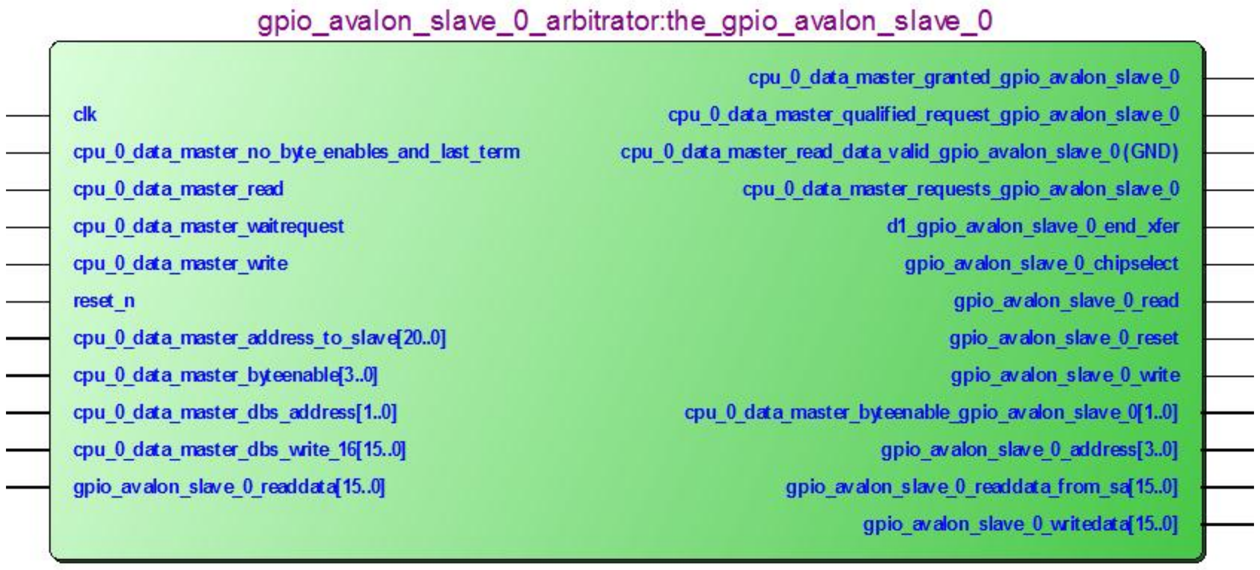

gpio_avalon_slave_0_arbitrator:the_gpio_avalon_slave_0
clk
cpu_0_data_master_no_byte_enables_and_last_term
cpu_0_data_master_read
cpu_0_data_master_waitrequest
cpu_0_data_master_write
reset_n
cpu_0_data_master_address_to_slave[20..0]
cpu_0_data_master_byteenable[3..0]
cpu_0_data_master_dbs_address[1..0]
cpu_0_data_master_dbs_write_16[15..0]
gpio_avalon_slave_0_readdata[15..0]
cpu_0_data_master_granted_gpio_avalon_slave_0
cpu_0_data_master_qualified_request_gpio_avalon_slave_0
cpu_0_data_master_read_data_valid_gpio_avalon_slave_0 (GND)
cpu_0_data_master_requests_gpio_avalon_slave_0
d1_gpio_avalon_slave_0_end_xfer
gpio_avalon_slave_0_chipselect
gpio_avalon_slave_0_read
gpio_avalon_slave_0_reset
gpio_avalon_slave_0_write
cpu_0_data_master_byteenable_gpio_avalon_slave_0[1..0]
gpio_avalon_slave_0_address[3..0]
gpio_avalon_slave_0_readdata_from_sa[15..0]
gpio_avalon_slave_0_writedata[15..0]

Figure 35: GPIO Arbitrator

CPU Data Master Arbitrator

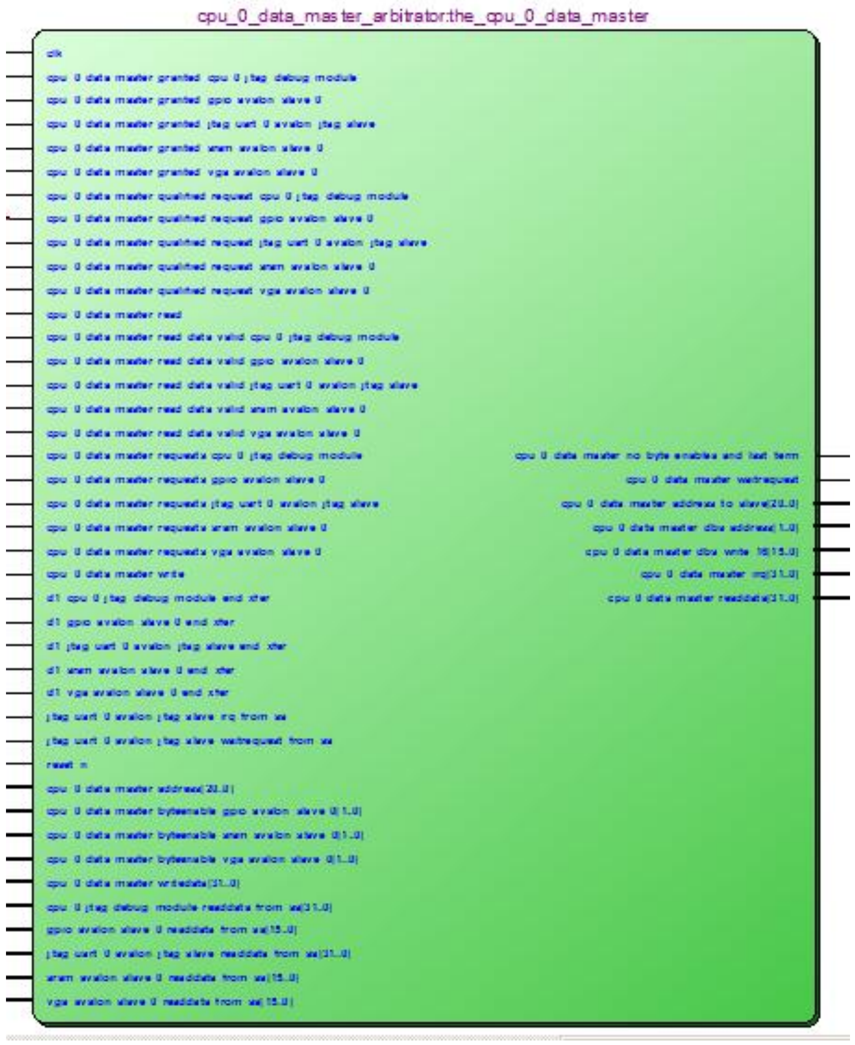

cpu_0_data_master_arbitrator:the_cpu_0_data_master
clk
cpu 0 data master granted cpu 0 jtag debug module
cpu 0 data master granted gpio avalon slave 0
cpu 0 data master granted jtag uart 0 avalon jtag slave
cpu 0 data master granted sram avalon slave 0
cpu 0 data master granted vga avalon slave 0
cpu 0 data master qualified request cpu 0 jtag debug module
cpu 0 data master qualified request gpio avalon slave 0
cpu 0 data master qualified request jtag uart 0 avalon jtag slave
cpu 0 data master qualified request sram avalon slave 0
cpu 0 data master qualified request vga avalon slave 0
cpu 0 data master read
cpu 0 data master read data valid cpu 0 jtag debug module
cpu 0 data master read data valid gpio avalon slave 0
cpu 0 data master read data valid jtag uart 0 avalon jtag slave
cpu 0 data master read data valid sram avalon slave 0
cpu 0 data master read data valid vga avalon slave 0
cpu 0 data master requests cpu 0 jtag debug module
cpu 0 data master requests gpio avalon slave 0
cpu 0 data master requests jtag uart 0 avalon jtag slave
cpu 0 data master requests sram avalon slave 0
cpu 0 data master requests vga avalon slave 0
cpu 0 data master write
d1 cpu 0 jtag debug module end xfer
d1 gpio avalon slave 0 end xfer
d1 jtag uart 0 avalon jtag slave end xfer
d1 sram avalon slave 0 end xfer
d1 vga avalon slave 0 end xfer
jtag uart 0 avalon jtag slave irq from sa
jtag uart 0 avalon jtag slave waitrequest from sa
reset n
cpu 0 data master address[20..0]
cpu 0 data master byteenable gpio avalon slave 0[1..0]
cpu 0 data master byteenable sram avalon slave 0[1..0]
cpu 0 data master byteenable vga avalon slave 0[1..0]
cpu 0 data master writedata[31..0]
cpu 0 jtag debug module readdata from sa[31..0]
gpio avalon slave 0 readdata from sa[15..0]
jtag uart 0 avalon jtag slave readdata from sa[31..0]
sram avalon slave 0 readdata from sa[15..0]
vga avalon slave 0 readdata from sa[15..0]
cpu 0 data master no byte enables and last term
cpu 0 data master waitrequest
cpu 0 data master address to slave[20..0]
cpu 0 data master dbs address[1..0]
cpu 0 data master dbs write 16[15..0]
cpu 0 data master irq[31..0]
cpu 0 data master readdata[31..0]

Figure 36: CPU Data Master Arbitrator

SRAM Arbitrator

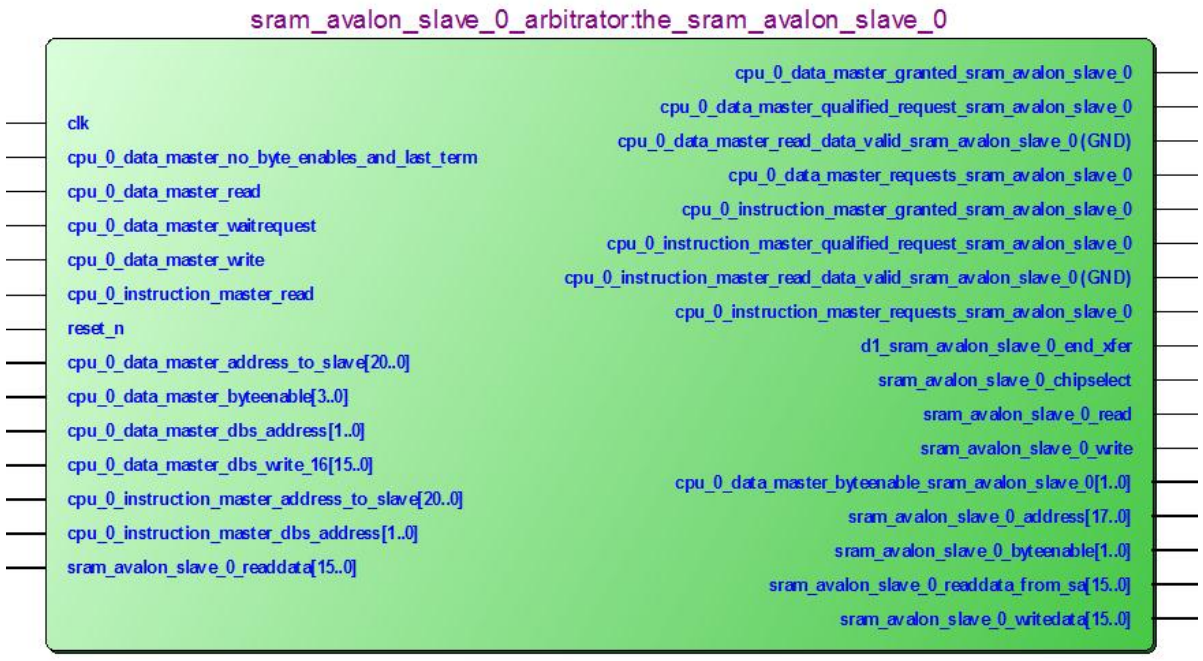

sram_avalon_slave_0_arbitrator:the_sram_avalon_slave_0
clk
cpu_0_data_master_no_byte_enables_and_last_term
cpu_0_data_master_read
cpu_0_data_master_waitrequest
cpu_0_data_master_write
cpu_0_instruction_master_read
reset_n
cpu_0_data_master_address_to_slave[20..0]
cpu_0_data_master_byteenable[3..0]
cpu_0_data_master_dbs_address[1..0]
cpu_0_data_master_dbs_write_16[15..0]
cpu_0_instruction_master_address_to_slave[20..0]
cpu_0_instruction_master_dbs_address[1..0]
sram_avalon_slave_0_readdata[15..0]
cpu_0_data_master_granted_sram_avalon_slave_0
cpu_0_data_master_qualified_request_sram_avalon_slave_0
cpu_0_data_master_read_data_valid_sram_avalon_slave_0(GND)
cpu_0_data_master_requests_sram_avalon_slave_0
cpu_0_instruction_master_granted_sram_avalon_slave_0
cpu_0_instruction_master_qualified_request_sram_avalon_slave_0
cpu_0_instruction_master_read_data_valid_sram_avalon_slave_0(GND)
cpu_0_instruction_master_requests_sram_avalon_slave_0
d1_sram_avalon_slave_0_end_xfer
sram_avalon_slave_0_chipselect
sram_avalon_slave_0_read
sram_avalon_slave_0_write
cpu_0_data_master_byteenable_sram_avalon_slave_0[1..0]
sram_avalon_slave_0_address[17..0]
sram_avalon_slave_0_byteenable[1..0]
sram_avalon_slave_0_readdata_from_sa[15..0]
sram_avalon_slave_0_writedata[15..0]

Figure 37: SRAM Arbitrator

SRAM Controller

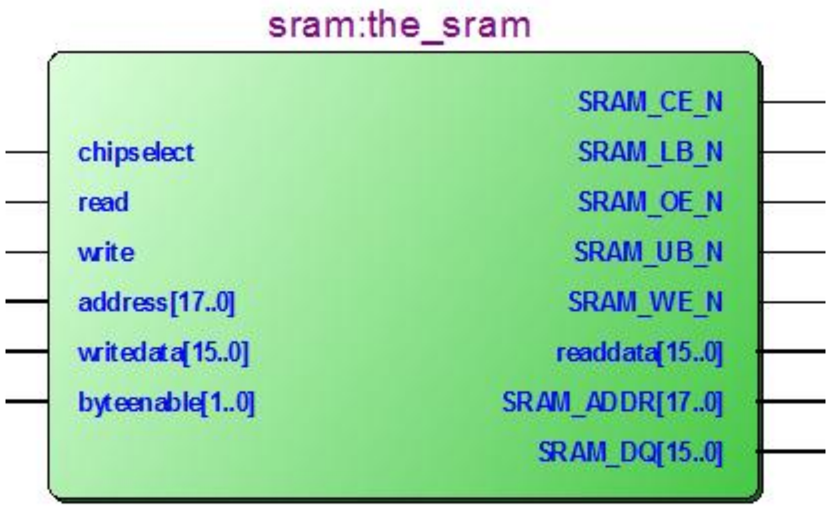


Figure 38: SRAM Controller

JTAG Arbitrator

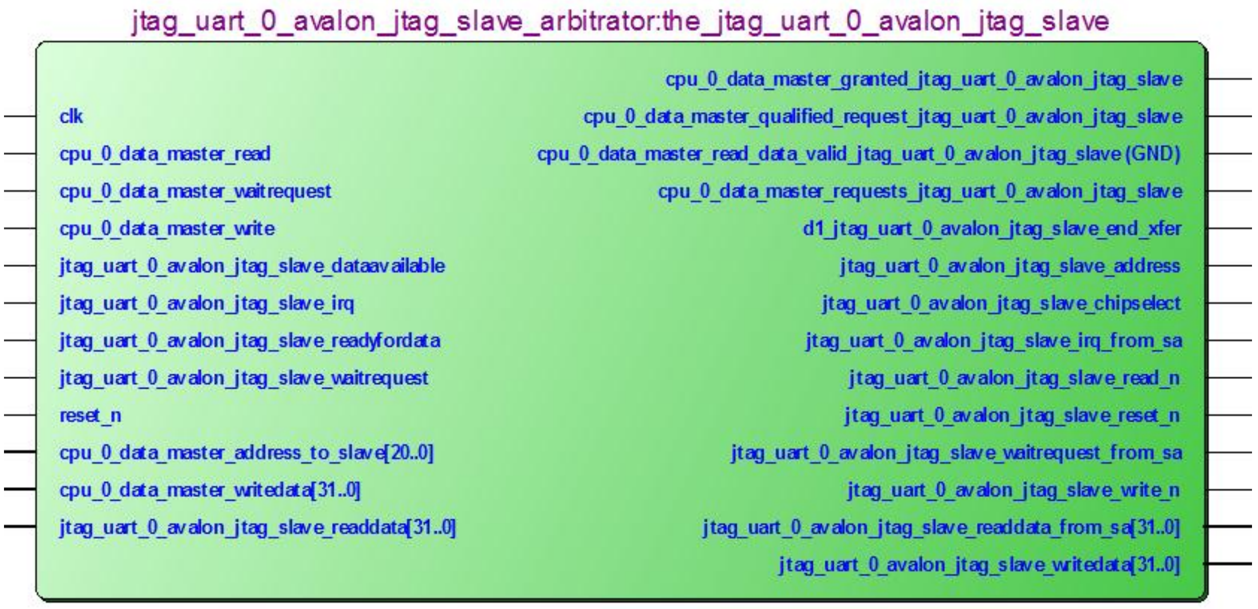
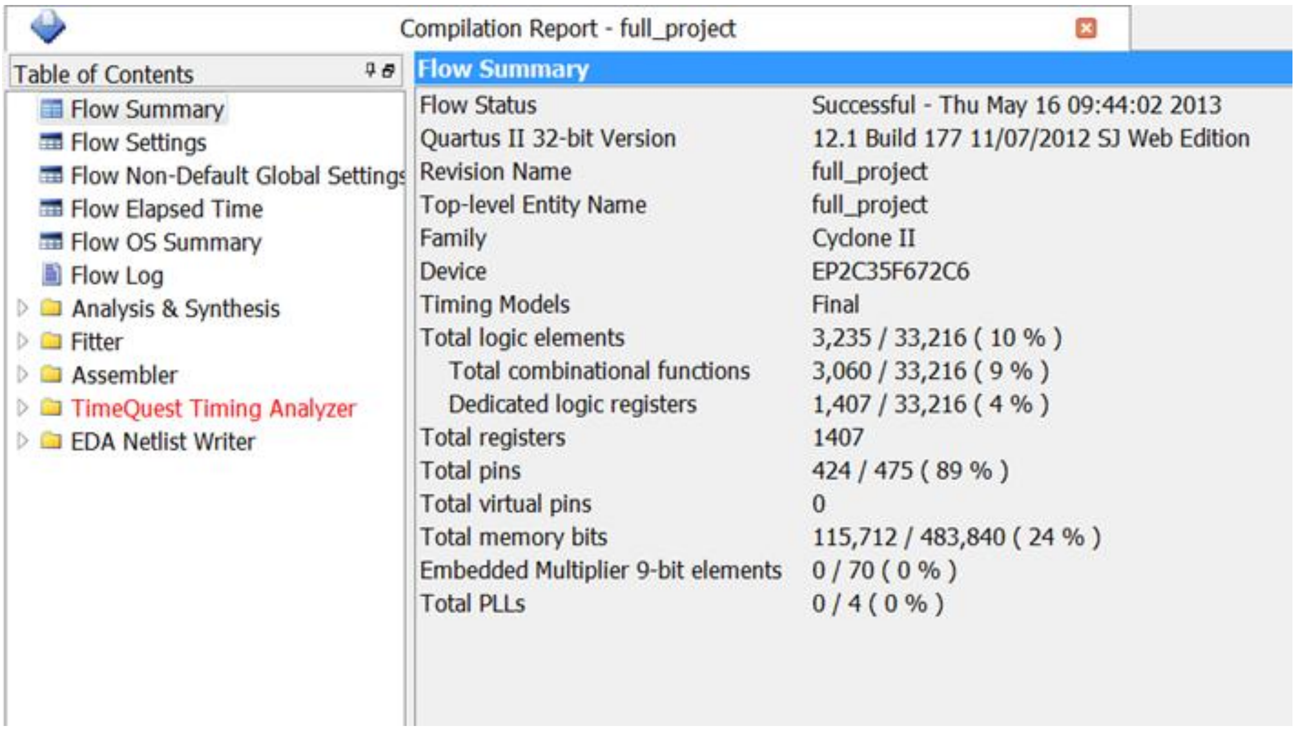

jtag_uart_0_avalon_jtag_slave_arbitrator:the_jtag_uart_0_avalon_jtag_slave
clk
cpu_0_data_master_read
cpu_0_data_master_waitrequest
cpu_0_data_master_write
jtag_uart_0_avalon_jtag_slave_dataavailable
jtag_uart_0_avalon_jtag_slave_irq
jtag_uart_0_avalon_jtag_slave_readyfordata
jtag_uart_0_avalon_jtag_slave_waitrequest
reset_n
cpu_0_data_master_address_to_slave[20..0]
cpu_0_data_master_writedata[31..0]
jtag_uart_0_avalon_jtag_slave_readdata[31..0]
cpu_0_data_master_granted_jtag_uart_0_avalon_jtag_slave
cpu_0_data_master_qualified_request_jtag_uart_0_avalon_jtag_slave
cpu_0_data_master_read_data_valid_jtag_uart_0_avalon_jtag_slave (GND)
cpu_0_data_master_requests_jtag_uart_0_avalon_jtag_slave
d1_jtag_uart_0_avalon_jtag_slave_end_xfer
jtag_uart_0_avalon_jtag_slave_address
jtag_uart_0_avalon_jtag_slave_chipselect
jtag_uart_0_avalon_jtag_slave_irq_from_sa
jtag_uart_0_avalon_jtag_slave_read_n
jtag_uart_0_avalon_jtag_slave_reset_n
jtag_uart_0_avalon_jtag_slave_waitrequest_from_sa
jtag_uart_0_avalon_jtag_slave_write_n
jtag_uart_0_avalon_jtag_slave_readdata_from_sa[31..0]
jtag_uart_0_avalon_jtag_slave_writedata[31..0]

Figure 39: JTAG Arbitrator

Final Compilation Report:



Flow Summary

Flow Status	Successful - Thu May 16 09:44:02 2013
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Web Edition
Revision Name	full_project
Top-level Entity Name	full_project
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	3,235 / 33,216 (10 %)
Total combinational functions	3,060 / 33,216 (9 %)
Dedicated logic registers	1,407 / 33,216 (4 %)
Total registers	1407
Total pins	424 / 475 (89 %)
Total virtual pins	0
Total memory bits	115,712 / 483,840 (24 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 40: Compilation Report

Image storage and aggregation memory:

- Internal to FPGA using ALTSYNCRAM megafunction

- Actual memory storage uses 128*128*6 + 16*16*6*4 = 104448 bits
- Compilation reports 115,712 bits, which means some other bits were used for other peripheral registers

Final Software Implementation:

The software performs/assists with the following tasks:

- Coordinates the left/right click functionality.
- Ensures new samples are unique.
- Performs aggregation by telling hardware where to write the next samples.
- Coordinates the location/color of highlight box.
- Checks various boundary conditions and allows for/tracks out of bounds traversal.

Ensuring Uniqueness

- Read 16 bit “select number” from hardware on every loop iteration.
- If this number is equivalent to the last such number, ignore and continue to next iteration.
- If it’s not equivalent, figure out which portion of the value differs, and write that to the hardware’s “read select”.
- This value determines the next sample.

The 16 bit register value consists of the four buffer entries – each entry contains the index of a particular sample. The software first checks if the 16 bit value is different from the last 16 bit value, and if it is, it breaks down the 16 bit value into its 4 subsections to determine which one is different. It then assigns that different as the next sample to be read – this is essentially the main function of the buffer.

Aggregation

- Since deltaX and deltaY are relative movement coordinates, software needs to keep track of absolute coordinates.
- Reads deltaX and deltaY, adds them to global position, checks boundaries, and writes back.
- The value written back is normalized to the following form for simplicity: ycoordinate+(xcoordinate*128)

Boundary Checking

- Firstly, checks when user is about to leave boundaries and warns with red box.
- Secondly, allows out of bounds traversal.
- Thirdly, tracks the out of bounds movement by moving red box along edge.
- Prevents strange bugs (such as splitting and syncing) with some corner case handling.

Splitting refers to an issue we had where if the box approaches a corner of the 128x128 with its corner, the box would display “split”, with half of it appearing in one corner, and half in another. Syncing refers to an issue we had where the red box is occasionally not able to respond in time when the user moves quickly off the in-bounds region – this would cause the box to remain static. To resolve this, we implemented a small algorithm that assumes a straight line from where the box began moving, to the edge where the user left the in-bounds region. We then just write the box to the position on the edge where the straight line from where the box started would have touched that edge.

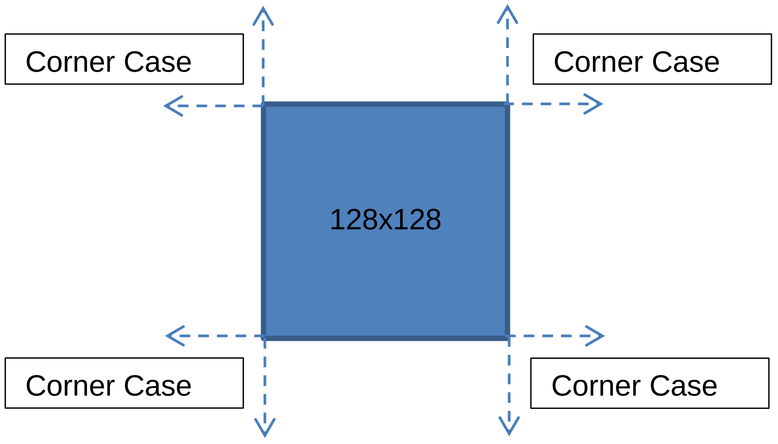


Figure 41: Software Glitchy Corner Cases

Tracking works as one would expect. As illustrated by the below picture, when the user is traversing the out-bounds, the red box follows the movement of the mouse by riding along the edge closest to it:

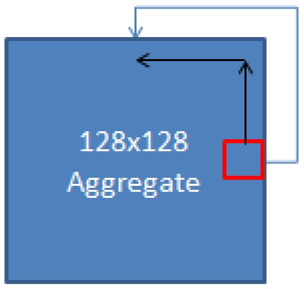


Figure 42: Edge Out-of-Bounds Tracking

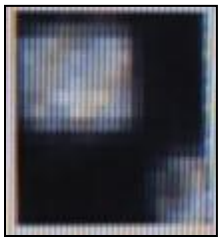
This makes it easier for the user to determine where exactly in the out bounds he is currently located.

Experiences:

Power of the ADNS-2051

- Or lack thereof…
- dx and dy are calculated based on an image gradient, but they are also rounded arbitrarily
    - Consider dx of 0.425 => 1
        - This skews the image, although it provides sensitivity for mouse movement.
- Image blurring adds skew
    - Quick movements are not supported

Slow Movement



Fast Movement

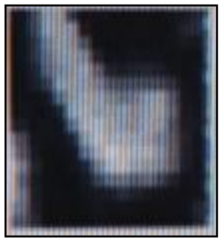


Figure 43: Slow and Fast Mouse Captures

- Hardware interfacing is simple using the DE2
    - Several ways to approach this project
        - Could have created our own microprocessor core
            - ✓ Set up digital I/O pins (GPIO) with buffers, multiplexers, etc. for communicating with the mouse
            - ✓ Same memory on FPGA still required
            - ✓ This would enable a “fully software-defined” implementation
- Timing diagrams are a good aid, however…
    - They do not always reflect what will happen in real time
    - Simulation vs. synthesizable
    - Heed the warnings given by Quartus II
        - Jitter
        - Latches
        - Timing concerns, etc.

Issues:

- Timing and synchronization
    - Image acquisition and software control are difficult to synchronize
        - Need to remove bottlenecks in software to get smooth acquisition and aggregation
    - It’s hard to determine the response time of software with respect to our clock speed on the FPGA (our queue system helped resolve any issues we would face from this issue)
- State machine
    - Specification
        - Need to consider all conditions outlined in ADNS-2051 datasheet
            - Timing required between sending and receiving commands
            - Timing required between different types of commands
            - Layout of the state machine in an efficient way

        - Toggling Power-Down pin in order to reset and synchronize the serial communication

- Timing
    - Data handling (outputs, changes) based on state changes vs. clock pulses
    - Simulations showed perfect behavior, actual communication generated by FPGA completely wrong [cannot trust simulation, had to use logic analyzer to verify what was going on]

Furthermore, the main limitations of our setup were as follows: The biggest limitation was probably the fidelity of the optical processor. Movements that are less than a pixel are converted to integers by the processor, so a lot of information is lost. What this manifests itself as is a “jagged” and “shifting” scan image, which changes a bit each time you scan over the same segment. Another limitation related to the CCD is the shutter speed – this causes blurring when the mouse is moved too fast. This, however, is understandable for a would be “scanner”. The final limitation was the intrinsic inaccuracy of the human hand. Because of the aforementioned issue of fidelity, the slight gyrations and inaccuracies caused by the movement of a human hand across a line of text can wreak major havoc on the quality of an image. The fix for this, as is the case in real scanner, is to have the scanning be done by a motorized component that moves slowly and a constant velocity.

Mentor Meeting Notes:

This section just serves to show several questions and concerns that were addressed during our meetings with our mentor, Luis.

Questions for Luis:

1) In the context of PLLs, are you familiar with the following warning:
Warning (15064): PLL "project_full:PROJECT|vga:the_vga|de2_vga_raster:vga|pll:pll_inst|altpll:altpll_component|pll" output port clk[0] feeds output pin "VGA_CLK" via non-dedicated routing -- jitter performance depends on switching rate of other design elements. Use PLL dedicated clock outputs to ensure jitter performance.

- If so, how do we fix it?

- Some sources say it is necessary to set up some pin to connect the PLL to, to which you may then connect any other logic that needs that PLL.

- We currently have it set up where we just do a clock division in a process statement to create 25MHz from the 50MHz drive clock. This seems to work fine.

There are 4 PLLs distributed around the FPGA - the closest one should be used. There are different settings for the synchronization and that could be a problem as well (in the megawizard setup...)

2) Do the megawizard functions ALTSYNCRAM guarantee we are connected to external SRAM?

3) Is there any easy way to port text displaying to our design so that we can have a status area on our VGA display?

- Looked through some of lab 2 source files, but not familiar enough with the design to extract this information

Some other notes:

- We are using internal FPGA RAM for images

- We are using SRAM for software

- Be sure to reference the RTL viewer for component connections in the design

- Get to this by Tool > Netlist Viewers > RTL view (or something like that)

- Might need to fix PLL to make design less jittery

Who Did What and Lessons Learned:

Since our group consisted of only three people, we were able to have pretty clear cut delineations as far as work goes. Dave served mainly the roles of system integrator and chief project designer. He did most of the tasks regarding getting data communications actually flowing between the mouse and FPGA, as well as providing the bigger picture of how the software and hardware would play together. Kishore skirted the line between software and hardware, designing and implementing in VHDL a large portion of the state diagrams, as well as assisting in translating portions of Arduino's provided mouse driver to C (for our purposes). I worked primarily on the software side, creating and implementing the mouse driver to get along with the FPGA.

We learned many valuable lessons from our experiences. Firstly, and perhaps most obviously, we learned a great deal more about the functionality of the FPGA as it relates to our project, especially with the GPIO interface and working with the on board memory. We also learned a lot about the surprising capabilities of the optical processors inside mice, and how to interface that mouse with the DE2. We learned what it takes to make hardware, peripherals, and software all work together in sync, particularly in regards to memory, drivers, and state machines. We learned better how to work together as a team, how to apportion work and maximize each other's strengths and weaknesses. We learned more about how to debug complex systems, and in that same regard, gained a better understanding of the tools/languages we used, like VHDL, GCC, C, and Quartus.

We managed to avoid many of the snags that have made victims out of past groups by heeding the advice of TAs and Professor Edwards. We started our project early on, and enabled constant teamwork and communication via shared Google Drive documents and files. We constructed careful state machines for all our protocols, and thanks to the successful attempts of previous individuals, had a clear idea of how our project would proceed. We stuck to the script set out in our initial proposal and project design relatively closely, and this allowed us to stay on track and coordinated as the project grew increasingly complicated. We also met twice a week just to make sure everybody was on the same page and to concretize our synchronization efforts. Because all of the above, in part, probably led to the success of our project, they would be on top of the list of our recommendations for future projects

Acknowledgements:

Firstly, a big thank you to our mentor Luis for his constant support and help throughout the lifetime of this project. Another big thank you to Professor Edwards for making this project possible (and for lending us his logic analyzer).

References:

[1] Avago ADNS-2051 Optical Mouse Sensor/Processor Data Sheet.
http://www.avagotech.com/docs/AV02-1364EN

Code:

RAM1, RAM2, RAM3, RAM4, and RAM5, were automatically generated using ALTSYNCRAM, which is a MegaWizard megafunction. This megafunction implements a dual-port ram of any size you want. RAM1 was 128x128 with 6 bits at each address (128*128 addresses with 6bits of data at each) while the others (RAM2-5) were 16x16 with 6 bits at each address (16*16 addresses with 6bits of data at each).

DE2_FSM_controller.vhd

```
-------------------------------------------------------------------------------
--
-- GPIO and FSM controller
--
-- Originally written by Kishore Padmaraju, kp2362@columbia.edu
--
-- Joint debugging and editing by David Calhoun, dmc2202@columbia.edu
-- and Kishore Padmaraju
--
-------------------------------------------------------------------------------

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


entity de2_FSM_controller is
  port(
          reset : in std_logic;
   clk   : in std_logic;
          --slowclock      : in std_logic;

          --connection to SW
          read                     :          in std_logic;
          write             :          in std_logic;
          chipselect        :          in std_logic;
          address           :          in std_logic_vector(3 downto 0);
		readdata		:		out std_logic_vector(15 downto 0);
		writedata		:		in std_logic_vector(15 downto 0);

		--address to get pixel buffer data from
		read_address : in std_logic_vector(7 downto 0);
		--current pixel data
		display_pixel_out : out std_logic_vector(7 downto 0);
		-- selects which image sample (2-5) should be written to
		--wren_selects	: in std_logic_vector(3 downto 0);
		-- selects which image sample (2-5) should be read from
		rden_selects	: in std_logic_vector(3 downto 0);
		--physical IO
		gpio	: inout std_logic_vector(35 downto 0);

		ledr	: out std_logic_vector(7 downto 0);
		ledg	: out std_logic_vector(7 downto 0)

  );
end de2_FSM_controller;

architecture layout of de2_FSM_controller is

		component Mouse_FSM is
			port(
				CLOCK	: in std_logic;						-- 50
MHz clock
				--SLOWCLOCK		:		in std_logic;
				RESET		: in std_logic;
				FSM_en			: in std_logic;
		-- FSM_en is to enable or halt the FSM
				GPIO		: inout std_logic_vector(35 downto 0);	-- GPIO pin connections

				MEM_wr			: out std_logic_vector(5 downto 0);
				data_out		: out std_logic_vector(7 downto 0);		-- data written out to
memory blocks
				wr_addr			: out std_logic_vector(7 downto 0)
			);
		end component;

		component RAM2 is
			port(
				clock		: in std_logic :='1';
				data		: in std_logic_vector (7 downto 0);
				rdaddress	: in std_logic_vector (7 downto 0);
				wraddress	: in std_logic_vector (7 downto 0);
				wren		: in std_logic :='0';
				q				: out std_logic_vector (7 downto 0)
			);
		end component;
```

```
component RAM3 is
	port(
		clock		: in std_logic :='1';
		data		: in std_logic_vector (7 downto 0);
		rdaddress	: in std_logic_vector (7 downto 0);
		wraddress	: in std_logic_vector (7 downto 0);
		wren		: in std_logic :='0';
		q			: out std_logic_vector (7 downto 0)
	);
end component;

component RAM4 is
	port(
		clock		: in std_logic :='1';
		data		: in std_logic_vector (7 downto 0);
		rdaddress	: in std_logic_vector (7 downto 0);
		wraddress	: in std_logic_vector (7 downto 0);
		wren		: in std_logic :='0';
		q			: out std_logic_vector (7 downto 0)
	);
end component;

component RAM5 is
	port(
		clock		: in std_logic :='1';
		data		: in std_logic_vector (7 downto 0);
		rdaddress	: in std_logic_vector (7 downto 0);
		wraddress	: in std_logic_vector (7 downto 0);
		wren		: in std_logic :='0';
		q			: out std_logic_vector (7 downto 0)
	);
end component;

signal dx : std_logic_vector(7 downto 0);
signal dx2 : std_logic_vector(7 downto 0);
signal dx3 : std_logic_vector(7 downto 0);
signal dx4 : std_logic_vector(7 downto 0);
signal dx5 : std_logic_vector(7 downto 0);

signal dy : std_logic_vector(7 downto 0);
signal dy2 : std_logic_vector(7 downto 0);
signal dy3 : std_logic_vector(7 downto 0);
signal dy4 : std_logic_vector(7 downto 0);
signal dy5 : std_logic_vector(7 downto 0);

signal lc : std_logic_vector(7 downto 0);
signal lc2 : std_logic_vector(7 downto 0);
signal lc3 : std_logic_vector(7 downto 0);
```

```
	signal lc4 : std_logic_vector(7 downto 0);
	signal lc5 : std_logic_vector(7 downto 0);

	signal rc : std_logic_vector(7 downto 0);
	signal rc2 : std_logic_vector(7 downto 0);
	signal rc3 : std_logic_vector(7 downto 0);
	signal rc4 : std_logic_vector(7 downto 0);
	signal rc5 : std_logic_vector(7 downto 0);


	signal img_smp: std_logic_vector(15 downto 0) := (others => '0'); -- stores the sequence numbers
for all the image samples

	signal FSM_enable			: std_logic :='1';
	signal mem_write			: std_logic_vector(5 downto 0); -- specifies whether RAM, dx, dy,
lc, or rc registers should be written to

	signal wren_selects		: std_logic_vector(3 downto 0) :="0001"; -- selects which image sample
(2-5) should be written to
	signal wren_RAM2, wren_RAM3, wren_RAM4, wren_RAM5: std_logic;
	signal data_line		: std_logic_vector(7 downto 0);
	--signal read_address			: std_logic_vector(7 downto 0) := (others => '0');
	signal write_address	: std_logic_vector(7 downto 0);
	signal display_pixel2	: std_logic_vector(7 downto 0) := (others => '0');
	signal display_pixel3	: std_logic_vector(7 downto 0) := (others => '0');
	signal display_pixel4	: std_logic_vector(7 downto 0) := (others => '0');
	signal display_pixel5	: std_logic_vector(7 downto 0) := (others => '0');


begin
--
	SW_Access : process (clk)
	begin

		if rising_edge(clk) then
			if reset = '1' then

			elsif chipselect = '1' then
				if read = '1' then

					--status of LC
					if address= "0000" then
						readdata <= "00000000" & lc;
					--status of RC
					elsif address= "0001" then
						readdata <= "00000000" & rc;
					--status of dx
					elsif address = "0010" then
						readdata <= "00000000" & dx;
```

```
                    --status of dy
                    elsif address = "0011" then
                        readdata <= "00000000" & dy;
                    elsif address = "0100" then
                        readdata <= img_smp;

                    end if;
                end if;
                if write = '1' then

--                  if address = "0100" then
--
--                  elsif address = "0101" then
--
--                  else
--
--
--                  end if;
                end if;
            end if;
        end if;
    end process SW_Access;




    process(clk)
    begin
        if(rising_edge(clk)) then

            case mem_write is
                when "000100" =>
                    wren_RAM2 <= wren_selects(0);
                    wren_RAM3 <= wren_selects(1);
                    wren_RAM4 <= wren_selects(2);
                    wren_RAM5 <= wren_selects(3);
                when others =>
                    wren_RAM2 <= '0';
                    wren_RAM3 <= '0';
                    wren_RAM4 <= '0';
                    wren_RAM5 <= '0';
            end case;

            case mem_write is
                when "000001" =>
                    case wren_selects is
                        when "0001" =>
                        dx2 <= data_line;
                when "0010" =>
                        dx3 <= data_line;
                when "0100" =>
                        dx4 <= data_line;
                when "1000" =>
                        dx5 <= data_line;
                when others =>
                        null;
        end case;
        --dx <= data_line;
when "000010" =>
        case wren_selects is
                when "0001" =>
                        dy2 <= data_line;
                when "0010" =>
                        dy3 <= data_line;
                when "0100" =>
                        dy4 <= data_line;
                when "1000" =>
                        dy5 <= data_line;
                when others =>
                        null;
        end case;
        --dy <= data_line;
when "001000" =>
        case wren_selects is
                when "0001" =>
                        lc2 <= data_line;
                when "0010" =>
                        lc3 <= data_line;
                when "0100" =>
                        lc4 <= data_line;
                when "1000" =>
                        lc5 <= data_line;
                when others =>
                        null;
        end case;
        --lc <= data_line;
when "010000" =>
        case wren_selects is
                when "0001" =>
                        rc2 <= data_line;
                when "0010" =>
                        rc3 <= data_line;
                when "0100" =>
                        rc4 <= data_line;
                when "1000" =>
                        rc5 <= data_line;
```

```
                                        when others =>
                                                null;
                                end case;
                                --rc <= data_line;
                        when "100000" =>
                                case wren_selects is
                                        when "0001" =>
                                                img_smp(3 downto 0) <= data_line(3 downto 0);
                                                wren_selects <= "0010";
                                        when "0010" =>
                                                img_smp(7 downto 4) <= data_line(3 downto 0);
                                                wren_selects <= "0100";
                                        when "0100" =>
                                                img_smp(11 downto 8) <= data_line(3 downto
0);
                                                wren_selects <= "1000";
                                        when "1000" =>
                                                img_smp(15 downto 12) <= data_line(3 downto
0);
                                                wren_selects <= "0001";
                                        when others =>
                                                null;
                                end case;
                        when others =>
                                null;
                        end case;
                end if;
        end process;

        Mouse_FSM_0: Mouse_FSM
        port map(
                CLOCK => clk,
                --SLOWCLOCK => slowclock,
                RESET => reset,
                FSM_en => FSM_enable,
                GPIO => gpio,
                MEM_wr => mem_write,
                data_out => data_line,
                wr_addr => write_address

        );

        RAM2_inst: RAM2
        port map(
                clock => clk,
                data => data_line,
                rdaddress => read_address,
                wraddress => write_address,
                wren => wren_RAM2,
		q => display_pixel2
	);

	RAM3_inst: RAM3
	port map(
		clock => clk,
		data => data_line,
		rdaddress => read_address,
		wraddress => write_address,
		wren => wren_RAM3,
		q => display_pixel3
	);

	RAM4_inst: RAM4
	port map(
		clock => clk,
		data => data_line,
		rdaddress => read_address,
		wraddress => write_address,
		wren => wren_RAM4,
		q => display_pixel4
	);

	RAM5_inst: RAM5
	port map(
		clock => clk,
		data => data_line,
		rdaddress => read_address,
		wraddress => write_address,
		wren => wren_RAM5,
		q => display_pixel5
	);


	Select_mem : process(clk)
	-- this could also end up being on the 25 MHz clock pulse we use for VGA
	begin
		if rising_edge(clk) then
			case wren_selects is
				when "0010" =>
					display_pixel_out <= display_pixel2;
					dx <= dx2;
					dy <= dy2;
					lc <= lc2;
					rc <= rc2;
				when "0100" =>
					display_pixel_out <= display_pixel3;
					dx <= dx3;
					dy <= dy3;
					lc <= lc3;
					rc <= rc3;
				when "1000" =>
					display_pixel_out <= display_pixel4;
					dx <= dx4;
					dy <= dy4;
					lc <= lc4;
					rc <= rc4;
				when "0001" =>
					display_pixel_out <= display_pixel5;
					dx <= dx5;
					dy <= dy5;
					lc <= lc5;
					rc <= rc5;
				when others =>
					display_pixel_out <= "00000000";
					dx <= "10101010";
					dy <= "10101010";
					lc <= "00000000";
					rc <= "00000000";
			end case;
		end if;
	end process Select_mem;

	ledg <= dx;
	ledr <= dy;

end layout;
```

DE2_vga_raster.vhd

```
-------------------------------------------------------------------------------
--
-- VGA raster display of abritrary memory
--
-- David Calhoun
-- dmc2202@columbia.edu
--
-- Adapted from code written by Stephen A. Edwards, sedwards@cs.columbia.edu
--
-------------------------------------------------------------------------------

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

entity de2_vga_raster is
```

```
	port (
		reset : in std_logic;
		clk   : in std_logic;			-- Should be 25.125 MHz

		-- Read from memory to access position
		read				:		in std_logic;
		write		:		in std_logic;
		chipselect	:		in std_logic;
		address		:		in std_logic_vector(3 downto 0);
		readdata		:		out std_logic_vector(15 downto 0);
		writedata		:		in std_logic_vector(15 downto 0);

		-- address to write pixel obtained from external source
		read_address : out std_logic_vector(7 downto 0);
		-- pixel data from external source
		data_from : in std_logic_vector(7 downto 0);
		-- selects which image sample (2-5) should be read from
		rden_selects	: out std_logic_vector(3 downto 0);

		-- VGA connectivity
		VGA_CLK,			-- Clock
		VGA_HS,			-- H_SYNC
		VGA_VS,			-- V_SYNC
		VGA_BLANK,			-- BLANK
		VGA_SYNC : out std_logic;	-- SYNC
		VGA_R,			-- Red[9:0]
		VGA_G,			-- Green[9:0]
		VGA_B : out std_logic_vector(9 downto 0) -- Blue[9:0]
	);

end de2_vga_raster;

architecture rtl of de2_vga_raster is

	component RAM1 is
		port(
			clock		: in std_logic :='1';
			data		: in std_logic_vector (7 downto 0);
			rdaddress	: in std_logic_vector (13 downto 0);
			wraddress	: in std_logic_vector (13 downto 0);
			wren		: in std_logic :='0';
			q			: out std_logic_vector (7 downto 0)
		);
	end component;

	component map_memory is
	port(
		clock : in std_logic;
		addr_in : in unsigned(13 downto 0);
```

```
                wren : in std_logic;
                addr_aout : out unsigned(13 downto 0);
                addr_sout : out unsigned(7 downto 0)

        );
        end component;

        component pll is
        port(
                inclk0 : in std_logic;
                c0 : out std_logic
        );
        end component;

        -- Video parameters
constant HTOTAL       : integer := 800;
        constant HSYNC        : integer := 96;
        constant HBACK_PORCH  : integer := 48;
        constant HACTIVE      : integer := 640;
        constant HFRONT_PORCH : integer := 16;
        constant VTOTAL       : integer := 525;
        constant VSYNC        : integer := 2;
        constant VBACK_PORCH  : integer := 33;
        constant VACTIVE      : integer := 480;
        constant VFRONT_PORCH : integer := 10;

        constant BOX_SET_XSTART : integer := 100;
        constant BOX_SET_XEND : integer := 356;
        constant BOX_SET_YSTART : integer := 100;
        constant BOX_SET_YEND : integer := 356;
        constant BOX_SET_XSTART2 : integer := 498;
        constant BOX_SET_XEND2 : integer := 530;
        constant BOX_SET_YSTART2 : integer := 220;
        constant BOX_SET_YEND2 : integer := 252;

        signal ram_address : unsigned(13 downto 0);
        signal ram_address2 : unsigned(7 downto 0);
        signal display_address11 : unsigned(13 downto 0) := "00000000000000";
        signal display_address21 : unsigned(7 downto 0) := "00000000";

        -- Signals for the video controller
        signal Hcount : unsigned(9 downto 0);  -- Horizontal position (0-800)
        signal Vcount : unsigned(9 downto 0);  -- Vertical position (0-524)
        signal EndOfLine, EndOfField : std_logic;
        signal vga_hblank, vga_hsync,
vga_vblank, vga_vsync : std_logic;  -- Sync. signals

        signal area : std_logic := '0';  -- flag for within writable area
        signal area_x : std_logic := '0';  -- flag for within writable area
        signal area_y : std_logic := '0';  -- flag for within writable area
        signal area2 : std_logic := '0';  -- flag for within writable area
        signal area_x2 : std_logic := '0';  -- flag for within writable area
        signal area_y2 : std_logic := '0';  -- flag for within writable area
        signal both_areas : std_logic:= '0';

        signal display_pixel : std_logic_vector(7 downto 0) := "00000000";
        signal pixel : unsigned(7 downto 0);
        signal waitx : std_logic := '1';
        signal waity : std_logic := '1';
        signal waitx2 : std_logic := '1';
        signal waity2 : std_logic := '1';

        signal display_pixel2 : std_logic_vector(7 downto 0) := "00000000";
        signal pixel2 : unsigned(7 downto 0);
        signal rdaddress : std_logic_vector(13 downto 0);
        --signal data : std_logic_vector(7 downto 0);
        --signal wren : std_logic := '1';
        signal rdaddress2 : std_logic_vector(7 downto 0);
        --signal data2 : std_logic_vector(7 downto 0);
        signal wren2 : std_logic := '1';
        -- need to clock at about 25 MHz for NTSC VGA
        signal clk_25 : std_logic;

        signal start_ram : unsigned (13 downto 0) := "00011100000000";
        signal q1, q2, q3, q4, q5 : std_logic_vector(7 downto 0);
        --signal data_from : std_logic_vector(7 downto 0);
        signal addr_aout : unsigned(13 downto 0);
        signal addr_sout : unsigned(7 downto 0);
        --signal rden_selects : std_logic_vector(3 downto 0) := "0001";

        signal init1 : std_logic := '0';
        signal init2 : std_logic := '0';
        signal aggr_en : std_logic := '0';
        signal check_selects : std_logic_vector(3 downto 0) := "0001";
        signal async_reset : std_logic := '0';
        signal data_to_aggr : std_logic_vector(7 downto 0) := "00000000";
        signal write_to_address : unsigned(13 downto 0);
        signal clear_address : unsigned(13 downto 0) := "00000000000000";
        signal async_count : unsigned(3 downto 0) := x"0";
        signal box_status : std_logic_vector(5 downto 0) := "000000";

begin

        RAM1_inst : RAM1 PORT MAP (
                clock      => clk,
                data       => data_to_aggr,
                rdaddress          => rdaddress,
                wraddress          => std_logic_vector(write_to_address),
		--wren	=> (not area),
		wren	=> (not area) and aggr_en,
		q		=> display_pixel
	);

	-- Originally implemented PLL, but jitter issues were more prevalent using PLL vs. clock division
--	pll_inst : pll PORT MAP (
--		inclk0	=> clk,
--		c0		=> clk_25
--	);

	MAP_inst : map_memory PORT MAP (
		clock	=> clk_25,
		wren => both_areas,
		addr_in => start_ram,
		addr_aout => addr_aout,
		addr_sout => addr_sout
	);

	rden_selects <= check_selects;

	Mem_Wr : process (clk,async_reset)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				write_to_address <= addr_aout;
				data_to_aggr <= data_from;

			elsif async_reset = '1' then

				write_to_address <= clear_address;
				data_to_aggr <= (others => '0');

			else

				write_to_address <= addr_aout;
				data_to_aggr <= data_from;
			end if;
		end if;
	end process Mem_Wr;

	Clr_addr : process(clk)
	begin
		if rising_edge(clk) then
			clear_address <= clear_address+1;
		end if;

	end process Clr_addr;
```

```
--		-- set up 25 MHz clock
		process (clk)
		begin
			if rising_edge(clk) then
				clk_25 <= not clk_25;
			end if;
		end process;

		-- Write current location of writing area
		TL_Write : process (clk)
		begin

			if rising_edge(clk) then
				if reset = '1' then
					readdata <= (others => '0');
				elsif chipselect = '1' then
					if read = '1' then
						-- for purposes of checking blank
						if address= "0000" then
							readdata <=  "000000000000000" & (vga_vsync or 
vga_hsync);
						-- return bottom-right location of writing area
						elsif address= "0001" then
							readdata <=  "00" & std_logic_vector(start_RAM);
						-- check the image indices
						elsif address = "0010" then
							readdata <= "000000000000" & check_selects;
						else
							readdata <= "0000000000000000";
						end if;
					end if;
					if write = '1' then
						--write new bottom-right starting address
						if address = "0011" then
							start_ram <= unsigned(writedata(13 downto 0));
						--select memory buffer to read from
						elsif address = "0100" then
							check_selects <= std_logic_vector(unsigned(writedata(3 
downto 0)));
						--enable for writing to aggregate memory
						elsif address = "0101" then
							aggr_en <= writedata(0);
						--box color
						elsif address = "0110" then
							box_status <= writedata(5 downto 0);
						--reset of aggregate image
						elsif address = "0111" then
							async_reset <= writedata(0);
```

```
                        else
                            start_RAM <= start_RAM;
                            box_status <= box_status;
                        end if;
                    end if;
                end if;
            end if;
end process TL_Write;

  -- Horizontal and vertical counters

HCounter : process (clk_25)
begin
            if rising_edge(clk_25) then
                if reset = '1' then
                    Hcount <= (others => '0');
                elsif EndOfLine = '1' then
                    Hcount <= (others => '0');
                else
                    Hcount <= Hcount + 1;

                end if;
            end if;
end process HCounter;

EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

VCounter: process (clk_25)
begin
            if rising_edge(clk_25) then
                if reset = '1' then
                    Vcount <= (others => '0');
                elsif EndOfLine = '1' then
                    if EndOfField = '1' then
                        Vcount <= (others => '0');
                    else
                        Vcount <= Vcount + 1;
                    end if;
                end if;
            end if;
end process VCounter;

EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK

HSyncGen : process (clk_25)
begin
            if rising_edge(clk_25) then
```

```
                if reset = '1' or EndOfLine = '1' then
                        vga_hsync <= '1';
                elsif Hcount = HSYNC - 1 then
                        vga_hsync <= '0';
                end if;
        end if;
end process HSyncGen;

HBlankGen : process (clk_25)
begin
        if rising_edge(clk_25) then
                if reset = '1' then
                        vga_hblank <= '1';
                elsif Hcount = HSYNC + HBACK_PORCH then
                        vga_hblank <= '0';
                elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
                        vga_hblank <= '1';
                end if;
        end if;
end process HBlankGen;

VSyncGen : process (clk_25)
begin
        if rising_edge(clk_25) then
                if reset = '1' then
                        vga_vsync <= '1';
                elsif EndOfLine ='1' then
                        if EndOfField = '1' then
                                vga_vsync <= '1';
                        elsif Vcount = VSYNC - 1 then
                                vga_vsync <= '0';
                        end if;
                end if;
        end if;
end process VSyncGen;

VBlankGen : process (clk_25)
begin
        if rising_edge(clk_25) then
                if reset = '1' then
                        vga_vblank <= '1';
                elsif EndOfLine = '1' then
                        if Vcount = VSYNC + VBACK_PORCH - 1 then
                                vga_vblank <= '0';
                        elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
                                vga_vblank <= '1';
                        end if;
                end if;
        end if;
```

```
	end process VBlankGen;

	Area_Check_X : process(clk_25)
	begin
		if rising_edge(clk_25) then
			if reset = '1' or Hcount = HSYNC + HBACK_PORCH + BOX_SET_XSTART-1
then
				area_x <= '1';
			elsif Hcount = HSYNC + HBACK_PORCH + BOX_SET_XEND-1 then
				area_x <= '0';
			end if;

		end if;

	end process Area_Check_X;


	Area_Check_Y : process(clk_25)
	begin
		if rising_edge(clk_25) then
			if reset = '1' then
				area_y <= '0';
			elsif EndOfLine = '1' then
				if Vcount = VSYNC + VBACK_PORCH - 1 + BOX_SET_YSTART-1 then
					area_y <= '1';
				elsif Vcount = VSYNC + VBACK_PORCH - 1 + BOX_SET_YEND-1 then
					area_y <= '0';
				end if;
			end if;

		end if;

	end process Area_Check_Y;

	Area_Check_X2 : process(clk_25)
	begin
		if rising_edge(clk_25) then
			if reset = '1' or Hcount = HSYNC + HBACK_PORCH + BOX_SET_XSTART2-1
then
				area_x2 <= '1';
			elsif Hcount = HSYNC + HBACK_PORCH + BOX_SET_XEND2-1 then
				area_x2 <= '0';
			end if;

		end if;

	end process Area_Check_X2;
	Area_Check_Y2 : process(clk_25)
	begin
		if rising_edge(clk_25) then
			if reset = '1' then
				area_y2 <= '0';
			elsif EndOfLine = '1' then
				if Vcount = VSYNC + VBACK_PORCH - 1 + BOX_SET_YSTART2-1
then
					area_y2 <= '1';
				elsif Vcount = VSYNC + VBACK_PORCH - 1 + BOX_SET_YEND2-1
then
					area_y2 <= '0';
				end if;
			end if;

		end if;

	end process Area_Check_Y2;

	-- Performs counting and pixel doubling for first active region

	Display_from_memory : process(clk_25)
	begin
		if rising_edge(clk_25) then
			if reset = '1' then
				waitx <= '0';
				waity <= '0';
				display_address11 <= "11111111111111";
				init1 <= '0';
			elsif area = '1' then
					init1 <= '1';
					if waitx = '1' then
						if display_address11(13 downto 7) = "00000000" then

							if waity = '1' then
								if display_address11 =
"00000000000000" then
									display_address11 <=
display_address11 - 1;
								else
									display_address11 <=
display_address11 - "00000010000001";
								end if;
							else
								display_address11 <=
display_address11 - "00000010000000";
							end if;
							waity <= not waity;
						else
```

```
								if display_address11 = "00000000000000" then
									display_address11 <=
"11111111111111";
								else
									display_address11 <=
display_address11 - "00000010000000";
								end if;
							end if;
						end if;


						waitx <= not waitx;
				elsif area = '0' and init1 = '0' then
					waitx <= '0';
					waity <= '0';
					display_address11 <= "11111111111111";

				end if;
			end if;
		end process Display_from_memory;

		-- Performs counting and pixel doubling for second active region

		Display_from_memory2 : process(clk_25)
		begin
			if rising_edge(clk_25) then
				if reset = '1' then
					waitx2 <= '0';
					waity2 <= '0';
					display_address21 <= "11111111";
					init2 <= '0';

				elsif area2 = '1' then

					init2 <= '1';
					if waitx2 = '1' then
						if display_address21(7 downto 4) = "0000" then
							if waity2 = '1' then
								if display_address21 = "00000000" then
									display_address21 <=
display_address21 - 1;
								else
									display_address21 <=
display_address21 - "00010001";
								end if;
							else
								display_address21 <= display_address21 -
"00010000";
							end if;
						waity2 <= not waity2;
					else
						if display_address21 = "00000000" then
							display_address21 <= "11111111";
						else
							display_address21 <= display_address21 -
"00010000";
						end if;
					end if;
				end if;


				waitx2 <= not waitx2;
			elsif area2 = '0' and init2 = '0' then
				waitx2 <= '0';
				waity2 <= '0';
				display_address21 <= "11111111";

			end if;
		end if;
	end process Display_from_memory2;


	area <= area_x and area_y;
	area2 <= area_x2 and area_y2;
	both_areas <= area and area2;

	-- Maps full aggregate image to first active area

	Mem_map : process(clk)
	begin
		if area = '1' then
			rdaddress <= std_logic_vector(display_address11);
		else
			rdaddress <= "0000000000000";
		end if;
	end process Mem_map;

	-- Maps sample image to second active area

	Mem_map2 : process(clk)
	begin
		if area2 = '1' then
			read_address <= std_logic_vector(display_address21);
		elsif both_areas = '0' then
			read_address <= std_logic_vector(addr_sout);
		else
			read_address <= "00000000";
		end if;
	end process Mem_map2;


	display_pixel2 <= data_from;

	-- Registered video signals going to the video DAC

	VideoOut : process (clk_25, reset)
	begin
		if reset = '1' then
			VGA_R <= "0000000000";
			VGA_G <= "0000000000";
			VGA_B <= "0000000000";
		elsif clk_25'event and clk_25 = '1' then
			if area = '1' then
				-- Checks if the display address is within the stiching-sensitive boundary
				if ((display_address11 >= start_RAM+1920) and (display_address11 <=
start_RAM+1935)) or
					((display_address11 >= start_RAM) and (display_address11 <=
start_RAM+15)) or
					(display_address11 = start_RAM+128) or
					(display_address11 = start_RAM+256) or
					(display_address11 = start_RAM+384) or
					(display_address11 = start_RAM+512) or
					(display_address11 = start_RAM+640) or
					(display_address11 = start_RAM+768) or
					(display_address11 = start_RAM+896) or
					(display_address11 = start_RAM+1024) or
					(display_address11 = start_RAM+1152) or
					(display_address11 = start_RAM+1280) or
					(display_address11 = start_RAM+1408) or
					(display_address11 = start_RAM+1536) or
					(display_address11 = start_RAM+1664) or
					(display_address11 = start_RAM+1792) or
					(display_address11 = start_RAM+128+15) or
					(display_address11 = start_RAM+256+15) or
					(display_address11 = start_RAM+384+15) or
					(display_address11 = start_RAM+512+15) or
					(display_address11 = start_RAM+640+15) or
					(display_address11 = start_RAM+768+15) or
					(display_address11 = start_RAM+896+15) or
					(display_address11 = start_RAM+1024+15) or
					(display_address11 = start_RAM+1152+15) or
					(display_address11 = start_RAM+1280+15) or
					(display_address11 = start_RAM+1408+15) or
					(display_address11 = start_RAM+1536+15) or
					(display_address11 = start_RAM+1664+15) or
					(display_address11 = start_RAM+1792+15) then
					case box_status is
```

```
                                        -- Yellow display box
                                        when "000000" =>
                                                VGA_R <= "1111111111";
                                                VGA_G <= "1111111111";
                                                VGA_B <= "0000000000";
                                        -- Green display box
                                        when "000001" =>
                                                VGA_R <= "0000000000";
                                                VGA_G <= "1111111111";
                                                VGA_B <= "0000000000";
                                        -- Red display box
                                        when others =>
                                                VGA_R <= "1111111111";
                                                VGA_G <= "0000000000";
                                                VGA_B <= "0000000000";
                                end case;
                        elsif async_reset = '1' then
                                VGA_R <= "0000000000";
                                VGA_G <= "0000000000";
                                VGA_B <= "0000000000";
                        else
                                VGA_R <= display_pixel(5 downto 0) & display_pixel(3 downto
0);
                                VGA_G <= display_pixel(5 downto 0) & display_pixel(3 downto
0);
                                VGA_B <= display_pixel(5 downto 0) & display_pixel(3 downto
0);

                        end if;
                elsif area2 = '1' then
                        VGA_R <= display_pixel2(5 downto 0) & display_pixel2(3 downto 0);
                        VGA_G <= display_pixel2(5 downto 0) & display_pixel2(3 downto 0);
                        VGA_B <= display_pixel2(5 downto 0) & display_pixel2(3 downto 0);
                elsif vga_hblank = '0' and vga_vblank = '0' then
                        VGA_R <= "1111111111";
                        VGA_G <= "1111111111";
                        VGA_B <= "1111111111";
                else
                        VGA_R <= "0000000000";
                        VGA_G <= "0000000000";
                        VGA_B <= "0000000000";
                end if;
        end if;
    end process VideoOut;



    VGA_CLK <= clk_25;
    VGA_HS <= not vga_hsync;
```

```
        VGA_VS <= not vga_vsync;
        VGA_SYNC <= '0';
        VGA_BLANK <= not (vga_hsync or vga_vsync);

end rtl;
```

full_project.vhd

```
-------------------------------------------------------------------------------
--
-- DE2 top-level module
--
-- David Calhoun
-- dmc2202@columbia.edu
--
-- Adapted from an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--
-------------------------------------------------------------------------------

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

entity full_project is
        port (
                -- Clocks

                CLOCK_27,                               -- 27 MHz
                CLOCK_50,                               -- 50 MHz
                EXT_CLOCK : in std_logic;                    -- External Clock

                -- Buttons and switches

                KEY : in std_logic_vector(3 downto 0);         -- Push buttons
                SW : in std_logic_vector(17 downto 0);         -- DPDT switches

                -- LED displays

                HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
                : out std_logic_vector(6 downto 0);
                LEDG : out std_logic_vector(8 downto 0);       -- Green LEDs
                LEDR : out std_logic_vector(17 downto 0);      -- Red LEDs

                -- RS-232 interface

                UART_TXD : out std_logic;                     -- UART transmitter
                UART_RXD : in std_logic;                     -- UART receiver
```

```
-- IRDA interface

--    IRDA_TXD : out std_logic;                      -- IRDA Transmitter
IRDA_RXD : in std_logic;                      -- IRDA Receiver

-- SDRAM

DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
DRAM_LDQM,                                  -- Low-byte Data Mask
DRAM_UDQM,                                  -- High-byte Data Mask
DRAM_WE_N,                                  -- Write Enable
DRAM_CAS_N,                                 -- Column Address Strobe
DRAM_RAS_N,                                 -- Row Address Strobe
DRAM_CS_N,                                  -- Chip Select
DRAM_BA_0,                                  -- Bank Address 0
DRAM_BA_1,                                  -- Bank Address 0
DRAM_CLK,                                   -- Clock
DRAM_CKE : out std_logic;                   -- Clock Enable

-- FLASH

FL_DQ : inout std_logic_vector(7 downto 0);      -- Data bus
FL_ADDR : out std_logic_vector(21 downto 0);  -- Address bus
FL_WE_N,                                    -- Write Enable
FL_RST_N,                                   -- Reset
FL_OE_N,                                    -- Output Enable
FL_CE_N : out std_logic;                    -- Chip Enable

-- SRAM

SRAM_DQ : inout std_logic_vector(15 downto 0); -- Data bus 16 Bits
SRAM_ADDR : out std_logic_vector(17 downto 0); -- Address bus 18 Bits
SRAM_UB_N,                                  -- High-byte Data Mask
SRAM_LB_N,                                  -- Low-byte Data Mask
SRAM_WE_N,                                  -- Write Enable
SRAM_CE_N,                                  -- Chip Enable
SRAM_OE_N : out std_logic;                  -- Output Enable

-- USB controller

OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
OTG_CS_N,                                   -- Chip Select
OTG_RD_N,                                   -- Write
OTG_WR_N,                                   -- Read
OTG_RST_N,                                  -- Reset
OTG_FSPEED,                   -- USB Full Speed, 0 = Enable, Z = Disable
```

```
OTG_LSPEED : out std_logic;     -- USB Low Speed, 0 = Enable, Z = Disable
OTG_INT0,                               -- Interrupt 0
OTG_INT1,                               -- Interrupt 1
OTG_DREQ0,                                -- DMA Request 0
OTG_DREQ1 : in std_logic;                    -- DMA Request 1
OTG_DACK0_N,                               -- DMA Acknowledge 0
OTG_DACK1_N : out std_logic;                   -- DMA Acknowledge 1

-- 16 X 2 LCD Module

LCD_ON,                  -- Power ON/OFF
LCD_BLON,                  -- Back Light ON/OFF
LCD_RW,                   -- Read/Write Select, 0 = Write, 1 = Read
LCD_EN,                  -- Enable
LCD_RS : out std_logic;     -- Command/Data Select, 0 = Command, 1 = Data
LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

-- SD card interface

SD_DAT,                  -- SD Card Data
SD_DAT3,                  -- SD Card Data 3
SD_CMD : inout std_logic;   -- SD Card Command Signal
SD_CLK : out std_logic;     -- SD Card Clock

-- USB JTAG link

TDI,                      -- CPLD -> FPGA (data in)
TCK,                      -- CPLD -> FPGA (clk)
TCS : in std_logic;         -- CPLD -> FPGA (CS)
TDO : out std_logic;        -- FPGA -> CPLD (data out)

-- I2C bus

I2C_SDAT : inout std_logic; -- I2C Data
I2C_SCLK : out std_logic;   -- I2C Clock

-- PS/2 port

PS2_DAT,                  -- Data
PS2_CLK : in std_logic;     -- Clock

-- VGA output

VGA_CLK,                                 -- Clock
VGA_HS,                                  -- H_SYNC
VGA_VS,                                  -- V_SYNC
VGA_BLANK,                                  -- BLANK
VGA_SYNC : out std_logic;                       -- SYNC
VGA_R,                                   -- Red[9:0]
```

```
            VGA_G,                                      -- Green[9:0]
            VGA_B : out std_logic_vector(9 downto 0);                 -- Blue[9:0]

            --  Ethernet Interface

            ENET_DATA : inout std_logic_vector(15 downto 0);    -- DATA bus 16Bits
            ENET_CMD,          -- Command/Data Select, 0 = Command, 1 = Data
            ENET_CS_N,                                     -- Chip Select
            ENET_WR_N,                                     -- Write
            ENET_RD_N,                                     -- Read
            ENET_RST_N,                                    -- Reset
            ENET_CLK : out std_logic;                          -- Clock 25 MHz
            ENET_INT : in std_logic;                           -- Interrupt

            -- Audio CODEC

            AUD_ADCLRCK : inout std_logic;                     -- ADC LR Clock
            AUD_ADCDAT : in std_logic;                         -- ADC Data
            AUD_DACLRCK : inout std_logic;                     -- DAC LR Clock
            AUD_DACDAT : out std_logic;                        -- DAC Data
            AUD_BCLK : inout std_logic;                        -- Bit-Stream Clock
            AUD_XCK : out std_logic;                           -- Chip Clock

            -- Video Decoder

            TD_DATA : in std_logic_vector(7 downto 0);  -- Data bus 8 bits
            TD_HS,                                  -- H_SYNC
            TD_VS : in std_logic;                       -- V_SYNC
            TD_RESET : out std_logic;                   -- Reset

            -- General-purpose I/O

            GPIO_0,                                 -- GPIO Connection 0
            GPIO_1 : inout std_logic_vector(35 downto 0) -- GPIO Connection 1
      );

end full_project;

architecture datapath of full_project is

      component seven_seg is
      port(
            inbits : in std_logic_vector(3 downto 0);
            outseg : out std_logic_vector(6 downto 0)

      );
      end component;

      -- signal clk25 : std_logic := '0';
```

```
	signal reset_n : std_logic;
	signal counter : unsigned(15 downto 0);
	signal read_address : std_logic_vector(7 downto 0);
	signal data_from : std_logic_vector(7 downto 0);
	signal rden_selects : std_logic_vector(3 downto 0);
	signal delx : std_logic_vector(7 downto 0);
	signal dely : std_logic_vector(7 downto 0);

	--signal wren_selects : std_logic_vector(3 downto 0);

begin

	process (CLOCK_50)
	begin
		if rising_edge(CLOCK_50) then
			if counter = x"FFFF" then
				reset_n <= '1';
			else
				reset_n <= '0';
				counter <= counter + 1;
			end if;
		end if;
	end process;

	PROJECT : entity work.project_full port map(
		clk_0 => CLOCK_50,
		reset_n => reset_n,

		SRAM_ADDR_from_the_sram => SRAM_ADDR,
		SRAM_CE_N_from_the_sram => SRAM_CE_N,
		SRAM_DQ_to_and_from_the_sram => SRAM_DQ,
		SRAM_LB_N_from_the_sram => SRAM_LB_N,
		SRAM_OE_N_from_the_sram => SRAM_OE_N,
		SRAM_UB_N_from_the_sram => SRAM_UB_N,
		SRAM_WE_N_from_the_sram => SRAM_WE_N,

		VGA_BLANK_from_the_vga => VGA_BLANK,
		VGA_B_from_the_vga => VGA_B,
		VGA_CLK_from_the_vga => VGA_CLK,
		VGA_G_from_the_vga => VGA_G,
		VGA_HS_from_the_vga => VGA_HS,
		VGA_R_from_the_vga => VGA_R,
		VGA_SYNC_from_the_vga => VGA_SYNC,
		VGA_VS_from_the_vga => VGA_VS,

		gpio_to_and_from_the_gpio => GPIO_0,
		ledg_from_the_gpio => delx,
		ledr_from_the_gpio => dely,
            display_pixel_out_from_the_gpio => data_from,
            read_address_to_the_gpio => read_address,
            rden_selects_to_the_gpio => rden_selects,
            --wren_selects_to_the_gpio => wren_selects,

            data_from_to_the_vga => data_from,
            read_address_from_the_vga => read_address,
            rden_selects_from_the_vga => rden_selects
            --wren_selects_from_the_vga => wren_selects

);

seg0 : seven_seg
port map(
            inbits => delx(3 downto 0),
            outseg => HEX0

);
seg1 : seven_seg
port map(
            inbits => delx(7 downto 4),
            outseg => HEX1

);
seg4 : seven_seg
port map(
            inbits => dely(3 downto 0),
            outseg => HEX4

);
seg5 : seven_seg
port map(
            inbits => dely(7 downto 4),
            outseg => HEX5

);

HEX7     <= (others => '1'); -- Leftmost
HEX6     <= (others => '1');
--HEX5     <= "1000111";
--HEX4     <= "1000111";
HEX3     <= (others => '1');
HEX2     <= (others => '1');
--HEX1     <= (others => '1');
--HEX0     <= (others => '1');          -- Rightmost
LEDG     <= (others => '1');
LEDR     <= (others => '1');
LCD_ON   <= '1';
LCD_BLON <= '1';
```

```
LCD_RW <= '1';
LCD_EN <= '0';
LCD_RS <= '0';

SD_DAT3 <= '1';
SD_CMD <= '1';
SD_CLK <= '1';

--  SRAM_DQ <= (others => 'Z');
--  SRAM_ADDR <= (others => '0');
--  SRAM_UB_N <= '1';
--  SRAM_LB_N <= '1';
--  SRAM_CE_N <= '1';
--  SRAM_WE_N <= '1';
--  SRAM_OE_N <= '1';

UART_TXD <= '0';
DRAM_ADDR <= (others => '0');
DRAM_LDQM <= '0';
DRAM_UDQM <= '0';
DRAM_WE_N <= '1';
DRAM_CAS_N <= '1';
DRAM_RAS_N <= '1';
DRAM_CS_N <= '1';
DRAM_BA_0 <= '0';
DRAM_BA_1 <= '0';
DRAM_CLK <= '0';
DRAM_CKE <= '0';
FL_ADDR <= (others => '0');
FL_WE_N <= '1';
FL_RST_N <= '0';
FL_OE_N <= '1';
FL_CE_N <= '1';
OTG_ADDR <= (others => '0');
OTG_CS_N <= '1';
OTG_RD_N <= '1';
OTG_RD_N <= '1';
OTG_WR_N <= '1';
OTG_RST_N <= '1';
OTG_FSPEED <= '1';
OTG_LSPEED <= '1';
OTG_DACK0_N <= '1';
OTG_DACK1_N <= '1';

TDO <= '0';

ENET_CMD <= '0';
ENET_CS_N <= '1';
ENET_WR_N <= '1';
```

```
        ENET_RD_N <= '1';
        ENET_RST_N <= '1';
        ENET_CLK <= '0';

        TD_RESET <= '0';

        I2C_SCLK <= '1';

        AUD_DACDAT <= '1';
        AUD_XCK <= '1';

        -- Set all bidirectional ports to tri-state
        DRAM_DQ     <= (others => 'Z');
        FL_DQ       <= (others => 'Z');
        SRAM_DQ     <= (others => 'Z');
        OTG_DATA    <= (others => 'Z');
        LCD_DATA    <= (others => 'Z');
        SD_DAT      <= 'Z';
        I2C_SDAT    <= 'Z';
        ENET_DATA   <= (others => 'Z');
        AUD_ADCLRCK <= 'Z';
        AUD_DACLRCK <= 'Z';
        AUD_BCLK    <= 'Z';
        --GPIO_0      <= (others => 'Z');
        GPIO_1      <= (others => 'Z');

end datapath;
```

map_memory.vhd

```
-------------------------------------------------------------------------------
--
-- Memory mapper
--
-- Currently configured for mapping a 16x16 block to an arbitrary location in
-- a 128x128 block
--
-- David Calhoun
-- dmc2202@columbia.edu
--
-------------------------------------------------------------------------------

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

entity map_memory is
```

```
	port(
		clock : in std_logic;
		addr_in : in unsigned(13 downto 0);
		wren : in std_logic;
		addr_aout : out unsigned(13 downto 0);
		addr_sout : out unsigned(7 downto 0)
		--rden_selects : in std_logic_vector(3 downto 0)
		--addr_out : out unsigned(13 downto 0)
	);

end map_memory;


architecture rtl of map_memory is
	--constant COL_HOP : unsigned := "00000000000001";
	--constant LIN_HOP : unsigned := "00000010000000";
	signal addr_map : unsigned(13 downto 0) := "11111111111111";
	signal addr_start : unsigned(13 downto 0);
	signal addr_cnt : unsigned(7 downto 0) := "11111110";
	signal counter : unsigned(3 downto 0) := "0000";


begin


	process(clock)

	begin
		if(rising_edge(clock)) then
				if addr_cnt = "11111111" then
					addr_start <= addr_in;
					addr_map <= addr_in;
					addr_cnt <= addr_cnt + 1;
					counter <= "0001";
				elsif addr_cnt(3 downto 0) = "1111" then
					-- move by 128
					addr_start <= addr_start + "00000010000000";
					addr_map <= addr_start + "00000010000000";
					addr_cnt <= addr_cnt + 1;
					counter <= "0001";
				else
					addr_map <= addr_start + counter;
					addr_cnt <= addr_cnt + 1;
					counter <= counter + 1;
				end if;
		end if;

	end process;
```

```
		addr_aout <= addr_map;
		addr_sout <= addr_cnt+1;





end rtl;
```

Mouse_FSM.vhd

```
-------------------------------------------------------------------------------
--
-- Finite state machine instantiation for mouse communications
--
-- Originally written by Kishore Padmaraju, kp2362@columbia.edu
--
-- Joint debugging and and editing by David Calhoun, dmc2202@columbia.edu
-- and Kishore Padmaraju
--
-------------------------------------------------------------------------------

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

entity Mouse_FSM is
	port(
		CLOCK, RESET			: in std_logic;
	-- 50 MHz clock
		FSM_en			: in std_logic;					--
FSM_en is to enable or halt the FSM
		GPIO		: inout std_logic_vector(35 downto 0);	-- GPIO pin connections
		MEM_wr			: out std_logic_vector(5 downto 0);		-- indicates
whether data is being written to RAM, dx, dy, lc, or rc registers
		data_out		: out std_logic_vector(7 downto 0);		-- data written out to memory
blocks
		wr_addr			: out std_logic_vector(7 downto 0)
	);
end Mouse_FSM;

architecture FSM of Mouse_FSM is
	type STATE_TYPE is (INIT, W10, W11, W12, W13, W14, W15,
	IDLE, CF0, CF1, CF2, CF3, CF4, CF5, CF6, CF7, CF8, CF9, CF10,
	M0, M1, M2, M3, M4, M5, M6, M7, M8, M9,
	DX0, DX1, DX2, DX3, DX4, DX5, DX6, DX7, DX8, DX9,
	DY0, DY1, DY2, DY3, DY4, DY5, DY6, DY7, DY8, DY9,
	PxD0, PxD1, PxD2, PxD3, PxD4, PxD5, PxD6, PxD7, PxD8, PxD9, PxD10, PxD11, PxD12,
	W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, LC, RC, N1);
	signal Y: STATE_TYPE := INIT;

	signal SDIO_en: std_logic := '1'; -- sets GPIO pin to read/write (set 0/1) for SDIO
	signal SCLK_en: std_logic := '1'; -- sets GPIO pin to read/write (set 0/1) for SCLK
	signal PD_en:	std_logic := '1'; -- sets GPIO pin to read/write (set 0/1) for PD

	signal SDIO:	std_logic := '0';
	signal SCLK:	std_logic := '0'; -- SCLK is clock driving communication with Optical processor
	signal PD:		std_logic := '0'; -- PD is power-down pin, set to 0 to keep Optical
processor constantly on

	constant counter_size_CLK:		integer := 7; -- 8 bits, counter will stop at "10000000",
yielding a clock rate of 97.65 kHz
	constant counter_size_100us:	integer := 13;  -- 5 bits, counter will stop at "10000", yielding a
wait of 163.8 us (> 100 us)
	signal CLK:				std_logic := '1'; -- Slow clock driving state
transitions (has to be < 4 MHz to work with optical processor chip)
	signal counter_CLK:			unsigned(counter_size_CLK downto 0) := (others => '0');
-- counter to generate slow clock from 50 MHz clock
	signal counter_approx5ms:			unsigned(19 downto 0) := (others => '0');
			-- counter to wait 4 ms
	signal counter_100us:			unsigned(counter_size_100us downto 0) := (others =>
'0'); --  counter to wait at least 100us
	signal counter_mem:				unsigned(counter_size_CLK downto 0) :=
(others => '0'); -- counter for waiting between memory writes

	constant CF_reg_addr:  std_logic_vector(7 downto 0) := X"0A"; -- address of configuration
register
	constant MOT_reg_addr:		std_logic_vector(7 downto 0) := X"02"; -- address of motion
register
	constant DX_reg_addr:  std_logic_vector(7 downto 0) := X"03"; -- address of dx coordinate
register
	constant DY_reg_addr:  std_logic_vector(7 downto 0) := X"04"; -- address of dy coordinate
register
	constant DATA_reg_addr: std_logic_vector(7 downto 0) := X"0C"; -- address of data_lower
register (register that holds the pixel value)
	constant CF_reg_dft:	std_logic_vector(7 downto 0) := X"01"; -- configuration register default
setting, sleep mode turned off
	constant CF_reg_pxd:	std_logic_vector(7 downto 0) := X"09"; -- configuration register pixel-
dump setting

	signal pxd_en:		std_logic						:= '0';
-- flag indicating pixel dump
	signal px_val:		std_logic_vector(7 downto 0) := (others => '0');
	signal px_addr:	unsigned(7 downto 0)		:= (others => '0');
	signal MOT_reg_val: std_logic_vector(7 downto 0) := (others => '0'); -- holds the value of the read
motion register, the MSB indicates motion occured
```

```
	signal DX_reg_val:	std_logic_vector(7 downto 0) := (others => '0'); -- holds the value of the
read dx coordinate register
	signal DY_reg_val:	std_logic_vector(7 downto 0) := (others => '0'); -- holds the value of the
read dy coordinate register
	signal bit_num:		unsigned(3 downto 0);
-- keeps track of bit numbers when reading or writing registers
	signal bit_num_prev: unsigned(3 downto 0);
	signal LC_val:		std_logic					:= '0';
-- high-value indicates left-click during image sample
	signal RC_val:		std_logic					:= '0';
-- high-value indicates right-click during image sample
	signal img_smp:		unsigned(3 downto 0) := "0000";   -- keeps track of image sequences

	signal mem_write:	std_logic_vector(5 downto 0) := "000000"; -- flags indicating memory
writes, "00001":dx coordinate, "010":dy coordinate, "100":pixel value
	signal mem_ff1  :	std_logic_vector(5 downto 0) := "000000";  -- flip-flops for detecting
changes in memory write conditions
	signal mem_ff2  :	std_logic_vector(5 downto 0) := "000000";

	--signal counter_approx5ms_test : unsigned(8 downto 0) := (others => '0');
	signal clk_ff : std_logic_vector(1 downto 0); -- flip flop to detect clock transitions
	signal pxd_en_ff : std_logic_vector(1 downto 0);
	signal pxd_addr_ff : std_logic_vector(1 downto 0);
	signal gpio_ff: std_logic_vector(1 downto 0);
	signal img_smp_ff: std_logic_vector(1 downto 0);
	signal click_ff: std_logic_vector(1 downto 0);
	signal mem_write_ff: std_logic_vector(1 downto 0);

	signal dump_skip : std_logic :='1';
	signal DX_reg_temp : std_logic_vector(7 downto 0) := (others => '0');
	signal DY_reg_temp : std_logic_vector(7 downto 0) := (others => '0');

begin

	process(CLOCK)
	begin
		if reset = '1' then
			GPIO(0) <= 'Z';
			GPIO(1) <= 'Z';
			GPIO(2) <= 'Z';
		elsif rising_edge(CLOCK) then
			if SDIO_en='1' then
				GPIO(0) <= SDIO;
			else
				GPIO(0) <= 'Z';
			end if;
			if SCLK_en='1' then
				GPIO(1) <= SCLK;
			else
```

```
                    GPIO(1) <= 'Z';
                end if;
                if PD_en='1' then
                    GPIO(2) <= PD;
                else
                    GPIO(2) <= 'Z';
                end if;
            end if;
        end process;

        GPIO(3) <= 'Z'; -- left-click
        GPIO(4) <= 'Z'; -- right-click

        GPIO(5) <= CLK;
        GPIO(7) <= reset;

        Write_to_Memory: process(CLOCK)
        begin
            if(rising_edge(CLOCK)) then

                mem_ff1 <= mem_write;
                mem_ff2 <= mem_ff1;

                MEM_wr <= mem_ff1 and (not mem_ff2);

                case mem_write is
                    when "000001" =>
                        wr_addr <= x"00";
                        -- convert 8-bit 2's complement to 16-bit 2's complement by
repeating the MSB
--                      data_out <= std_logic_vector(signed(DX_reg_val) +
signed(DX_reg_temp));
--                      DX_reg_temp <= DX_reg_val;
                        data_out <= DX_reg_val;
                    when "000010" =>
                        wr_addr <= x"00";
                        -- convert 8-bit 2's complement to 16-bit 2's complement by
repeating the MSB
--                      data_out <= std_logic_vector(signed(DY_reg_val) +
signed(DY_reg_temp));
--                      DY_reg_temp <= DY_reg_val;
                        data_out <= DY_reg_val;
                    when "000100" =>
                        wr_addr <= std_logic_vector(px_addr);
--                      if px_addr = "10001000" then
--                          data_out <= "11111111";
--                      else
--                          data_out <= "00000000";
--                      end if;
```

```
                        data_out <= px_val;
                when "001000" =>
                        wr_addr <= x"00";
                        data_out <= (7 downto 1 => '0') & LC_val;
                when "010000" =>
                        wr_addr <= x"00";
                        data_out <= (7 downto 1 => '0') & RC_val;
                when "100000" =>
                        wr_addr <= x"00";
                        data_out <= (7 downto 4 => '0') & std_logic_vector(img_smp);
                when others =>
                        wr_addr <= x"00";
                        data_out <= (others => '0');
            end case;
        end if;
end process Write_to_Memory;

Slow_Clock: process(CLOCK)
begin
        if reset = '1' then
            CLK <= '0';
            counter_CLK <= (others => '0');
        elsif(rising_edge(CLOCK)) then
            if(counter_CLK=x"20") then
                CLK <= NOT CLK;
                counter_CLK <= (others => '0');
            else
                counter_CLK <= counter_CLK + 1;
            end if;
        end if;
end process Slow_Clock;

process(CLOCK)
begin
        if reset = '1' then
            Y <= INIT;
        elsif(rising_edge(CLOCK)) then
        case Y is
            when INIT =>
                if(CLK='1') then
                        Y <= W10;
                end if;
            when W10 =>
                if(CLK='0') then
                        if (counter_approx5ms(18)='1') then
                            Y <= W11;
                        end if;
                end if;
            when W11 =>
```

```
		if(CLK='1') then

				Y <= W12;

		end if;
	when W12 =>
		if(CLK='0') then
			if counter_100us(13) = '1' then
				Y <= W13;
			end if;
		end if;
	when W13 =>
		if(CLK='1') then
			Y <= W14;
		end if;
	when W14 =>
		if(CLK='0') then
			if (counter_approx5ms(18)='1') then
				Y <= W15;
			end if;
		end if;
	when W15 =>
		if(CLK='1') then
				Y <= IDLE;
		end if;
	when IDLE =>
		if(CLK='0' AND FSM_en='1') then
			Y <= CF0;
		end if;
	when CF0 =>
		if(CLK='1') then
			Y <= CF1;
		end if;
	when CF1 =>
		if(CLK='0') then
			Y <= CF2;
		end if;
	when CF2 =>
		if(CLK='1') then
			Y <= CF3;
		end if;
	when CF3 =>
		if(CLK='0') then
			if(bit_num=X"0") then
				Y <= CF5;
			else
				Y <= CF4;
			end if;
		end if;
```

```
			when CF4 =>
				if(CLK='1') then
					Y <= CF3;
				end if;
			when CF5 =>
				if(CLK='1') then
					if(pxd_en='0') then
						Y <= CF6;
					elsif(pxd_en='1' and (DX_reg_val /= "00000000" or DY_reg_val
/= "00000000")) then

					--and (DX_reg_val /= "00000000" or DY_reg_val /= "00000000")
						Y <= CF8;
					end if;
				end if;
			when CF6 =>
				if(CLK='0') then
					if(bit_num=X"0") then
						Y <= CF10;
					else
						Y <= CF7;
					end if;
				end if;
			when CF7 =>
				if(CLK='1') then
					Y <= CF6;
				end if;
			when CF8 =>
				if(CLK='0') then
					if(bit_num=X"0") then
						Y <= CF10;
					else
						Y <= CF9;
					end if;
				end if;
			when CF9 =>
				if(CLK='1') then
					Y <= CF8;
				end if;
			when CF10 =>
				if(CLK='1') then
					Y <= W0;
				end if;
			when W0 =>
				if(CLK='0') then
					Y <= W1;
				end if;
			when W1 =>
				if(CLK='1') then
```

```
                                        if(counter_100us(13)='1') then
                                                if(pxd_en='0') then
                                                        Y <= M0;
                                                elsif(pxd_en='1' and (DX_reg_val /= "00000000" or
DY_reg_val /= "00000000")) then
                                                        Y <= PxD0;
                                                end if;
                                        else
                                                Y <= W0;
                                        end if;
                                end if;
                        when M0 =>
                                if(CLK='0') then
                                        Y <= M1;
                                end if;
                        when M1 =>
                                if(CLK='1') then
                                        Y <= M2;
                                end if;
                        when M2 =>
                                if(CLK='0') then
                                        if(bit_num=X"0") then
                                                Y <= M4;
                                        else
                                                Y <= M3;
                                        end if;
                                end if;
                        when M3 =>
                                if(CLK='1') then
                                        Y <= M2;
                                end if;
                        when M4 =>
                                if(CLK='1') then
                                        Y <= M5;
                                end if;
                        when M5 =>
                                if(CLK='0') then
                                        Y <= W2;
                                end if;
                        when W2 =>
                                if(CLK='1') then
                                        Y <= W3;
                                end if;
                        when W3 =>
                                if(CLK='0') then

                                        if(counter_100us(13)='1') then
                                                Y <= M6;
                                        else
                                                Y <= W2;
                                        end if;
                                end if;
                        when M6 =>
                                if(CLK='1') then
                                        Y <= M7;
                                end if;
                        when M7 =>
                                if(CLK='0') then
                                        if(bit_num=X"0") then
                                                Y <= M9;
                                        else
                                                Y <= M8;
                                        end if;
                                end if;
                        when M8 =>
                                if(CLK='1') then
                                        Y <= M7;
                                end if;
                        when M9 =>
                                if(CLK='1') then
                                        if(MOT_reg_val(7)='0') then
                                                Y <= IDLE;
                                        elsif(MOT_reg_val(7)='1') then
                                                Y <= DX0;
                                                --Y <= IDLE;
                                        end if;
                                end if;
                        when DX0 =>
                                if(CLK='0') then
                                        Y <= DX1;
                                end if;
                        when DX1 =>
                                if(CLK='1') then
                                        Y <= DX2;
                                end if;
                        when DX2 =>
                                if(CLK='0') then
                                        if(bit_num=X"0") then
                                                Y <= DX4;
                                        else
                                                Y <= DX3;
                                        end if;
                                end if;
                        when DX3 =>
                                if(CLK='1') then
                                        Y <= DX2;
                                end if;
                        when DX4 =>
```

```
		if(CLK='1') then
			Y <= DX5;
		end if;
	when DX5 =>
		if(CLK='0') then
			Y <= W4;
		end if;
	when W4 =>
		if(CLK='1') then
			Y <= W5;
		end if;
	when W5 =>
		if(CLK='0') then

			if(counter_100us(13)='1') then
				Y <= DX6;
			else
				Y <= W4;
			end if;
		end if;
	when DX6 =>
		if(CLK='1') then
			Y <= DX7;
		end if;
	when DX7 =>
		if(CLK='0') then
			if(bit_num=X"0") then
				Y <= DX9;
			else
				Y <= DX8;
			end if;
		end if;
	when DX8 =>
		if(CLK='1') then
			Y <= DX7;
		end if;
	when DX9 =>
		if(CLK='1') then
			Y <= DY0;
		end if;
	when DY0 =>
		if(CLK='0') then
			Y <= DY1;
		end if;
	when DY1 =>
		if(CLK='1') then
			Y <= DY2;
		end if;
	when DY2 =>
```

```
        if(CLK='0') then
            if(bit_num=X"0") then
                Y <= DY4;
            else
                Y <= DY3;
            end if;
        end if;
    when DY3 =>
        if(CLK='1') then
            Y <= DY2;
        end if;
    when DY4 =>
        if(CLK='1') then
            Y <= DY5;
        end if;
    when DY5 =>
        if(CLK='0') then
            Y <= W6;
        end if;
    when W6 =>
        if(CLK='1') then
            Y <= W7;
        end if;
    when W7 =>
        if(CLK='0') then

            if(counter_100us(13)='1') then
                Y <= DY6;
            else
                Y <= W6;
            end if;
        end if;
    when DY6 =>
        if(CLK='1') then
            Y <= DY7;
        end if;
    when DY7 =>
        if(CLK='0') then
            if(bit_num=X"0") then
                Y <= DY9;
            else
                Y <= DY8;
            end if;
        end if;
    when DY8 =>
        if(CLK='1') then
            Y <= DY7;
        end if;
    when DY9 =>
```

```
        if(CLK='1') then
                Y <= CF0;
        end if;
when PxD0 =>
        if(CLK='0') then
                Y <= PxD1;
        end if;
when PxD1 =>
        if(CLK='1') then
                Y <= PxD2;
        end if;
when PxD2 =>
        if(CLK='0') then
                if(bit_num=X"0") then
                        Y <= PxD4;
                else
                        Y <= PxD3;
                end if;
        end if;
when PxD3 =>
        if(CLK='1') then
                Y <= PxD2;
        end if;
when PxD4 =>
        if(CLK='1') then
                Y <= PxD5;
        end if;
when PxD5 =>
        if(CLK='0') then
                Y <= W8;
        end if;
when W8 =>
        if(CLK='1') then
                Y <= W9;
        end if;
when W9 =>
        if(CLK='0') then

                if(counter_100us(13)='1') then
                        Y <= PxD6;
                else
                        Y <= W8;
                end if;
        end if;
when PxD6 =>
        if(CLK='1') then
                Y <= PxD7;
        end if;
when PxD7 =>
```

```
                if(CLK='0') then
                        if(bit_num=X"0") then
                                Y <= PxD9;
                        else
                                Y <= PxD8;
                        end if;
                end if;
        when PxD8 =>
                if(CLK='1') then
                        Y <= PxD7;
                end if;
        when PxD9 =>
                if(CLK='1') then
                        if(px_val(7)='1') then
                                --Y <= PxD1;
                                Y <= PxD12;
                        elsif(px_val(7)='0') then
                                if(px_addr=X"FF") then
                                        Y <= PxD11;
                                else
                                        Y <= PxD10;
                                end if;
                        end if;
                end if;
        when PxD10 =>
                if(CLK='0') then
                        Y <= PxD1;
                end if;
        when PxD12 =>
                if(CLK='0') then
                        Y <= PxD1;
                end if;

        when PxD11 =>
                if(CLK='0') then
                        Y <= LC;
                end if;
        when LC =>
                if(CLK='1') then
                        Y <= RC;
                end if;
        when RC =>
                if(CLK='0') then
                        Y <= N1;
                end if;
        when N1 =>
                if(CLK='1') then
                        Y <= IDLE;
                end if;
```

```
				end case;
				end if;
		end process;

		-- Handle setting SDIO pin to write or read
		process(Y)
		begin
				case Y is
						when
M5|W2|W3|M6|M7|M8|DX5|W4|W5|DX6|DX7|DX8|DY5|W6|W7|DY6|DY7|DY8|PxD5|W8|W9|PxD6|PxD7|
PxD8 =>
								SDIO_en <= '0';
						when others =>
								SDIO_en <= '1';
				end case;
		end process;

		-- Handle SCLK generation, writing of SDIO pin
		process(Y)
		begin
				case Y is
						when INIT =>
								SCLK <= '0';
								SDIO <= '0';
						when W10 =>
								SCLK <= '0';
								SDIO <= '0';
						WHEN W11 =>
								SCLK <= '0';
								SDIO <= '0';
						WHEN W12 =>
								SCLK <= '1';
								SDIO <= '1';
						WHEN W13 =>
								SCLK <= '1';
								SDIO <= '1';
--						WHEN W16 =>
--								SCLK <= '1';
--								SDIO <= '1';
--						WHEN W17 =>
--								SCLK <= '1';
--								SDIO <= '1';
						WHEN W14 =>
								SCLK <= '1';
								SDIO <= '1';
						WHEN W15 =>
								SCLK <= '1';
								SDIO <= '1';
						when IDLE =>
```

```
        SCLK <= '1';
        SDIO <= '1';
when CF0 =>
        SCLK <= '1';
        SDIO <= '1';
when CF1 =>
        SCLK <= '0';
        SDIO <= '1';
when CF2 =>
        SCLK <= '1';
        SDIO <= '1';
when CF3 =>
        SCLK <= '0';
        SDIO <= CF_reg_addr(to_integer(bit_num));
when CF4 =>
        SCLK <= '1';
        SDIO <= CF_reg_addr(to_integer(bit_num_prev));
when CF5 =>
        SCLK <= '1';
        SDIO <= CF_reg_addr(to_integer(bit_num_prev));
when CF6 =>
        SCLK <= '0';
        SDIO <= CF_reg_dft(to_integer(bit_num));
when CF7 =>
        SCLK <= '1';
        SDIO <= CF_reg_dft(to_integer(bit_num_prev));
when CF8 =>
        SCLK <= '0';
        SDIO <= CF_reg_pxd(to_integer(bit_num));
when CF9 =>
        SCLK <= '1';
        SDIO <= CF_reg_pxd(to_integer(bit_num_prev));
when CF10 =>
        SCLK <= '1';
        SDIO <= '1';
when W0 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when W1 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when M0 =>
        SCLK <= '0';
        SDIO <= '0';
when M1 =>
        SCLK <= '1';
        SDIO <= '0';
when M2 =>
        SCLK <= '0';
        SDIO <= MOT_reg_addr(to_integer(bit_num));
when M3 =>
        SCLK <= '1';
        SDIO <= MOT_reg_addr(to_integer(bit_num_prev));
when M4 =>
        SCLK <= '1';
        SDIO <= MOT_reg_addr(to_integer(bit_num));
when M5 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when W2 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when W3 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when M6 =>
        SCLK <= '0';
        SDIO <= '1';      -- SDIO: Don't Care
when M7 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when M8 =>
        SCLK <= '0';
        SDIO <= '1';      -- SDIO: Don't Care
when M9 =>
        SCLK <= '1';
        SDIO <= '1';      -- SDIO: Don't Care
when DX0 =>
        SCLK <= '0';
        SDIO <= '0';
when DX1 =>
        SCLK <= '1';
        SDIO <= '0';
when DX2 =>
        SCLK <= '0';
        SDIO <= DX_reg_addr(to_integer(bit_num));
when DX3 =>
        SCLK <= '1';
        SDIO <= DX_reg_addr(to_integer(bit_num_prev));
when DX4 =>
        SCLK <= '1';
        SDIO <= DX_reg_addr(to_integer(bit_num));
when DX5 =>
        SCLK <= '1';
        SDIO <= '0';      -- SDIO: Don't Care
when W4 =>
        SCLK <= '1';
        SDIO <= '0';      -- SDIO: Don't Care
```

```
when W5 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when DX6 =>
        SCLK <= '0';
        SDIO <= '0';       -- SDIO: Don't Care
when DX7 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when DX8 =>
        SCLK <= '0';
        SDIO <= '0';       -- SDIO: Don't Care
when DX9 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when DY0 =>
        SCLK <= '0';
        SDIO <= '0';
when DY1 =>
        SCLK <= '1';
        SDIO <= '0';
when DY2 =>
        SCLK <= '0';
        SDIO <= DY_reg_addr(to_integer(bit_num));
when DY3 =>
        SCLK <= '1';
        SDIO <= DY_reg_addr(to_integer(bit_num_prev));
when DY4 =>
        SCLK <= '1';
        SDIO <= DY_reg_addr(to_integer(bit_num));
when DY5 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when W6 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when W7 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when DY6 =>
        SCLK <= '0';
        SDIO <= '0';       -- SDIO: Don't Care
when DY7 =>
        SCLK <= '1';
        SDIO <= '0';       -- SDIO: Don't Care
when DY8 =>
        SCLK <= '0';
        SDIO <= '0';       -- SDIO: Don't Care
when DY9 =>
```

```
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD0 =>
            SCLK <= '0';
            SDIO <= '0';
when PxD1 =>
            SCLK <= '1';
            SDIO <= '0';
when PxD2 =>
            SCLK <= '0';
            SDIO <= DATA_reg_addr(to_integer(bit_num));
when PxD3 =>
            SCLK <= '1';
            SDIO <= DATA_reg_addr(to_integer(bit_num_prev));
when PxD4 =>
            SCLK <= '1';
            SDIO <= DATA_reg_addr(to_integer(bit_num));
when PxD5 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when W8 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when W9 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD6 =>
            SCLK <= '0';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD7 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD8 =>
            SCLK <= '0';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD9 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD10 =>
            SCLK <= '0';
            SDIO <= '0';
when PxD11 =>
            SCLK <= '1';
            SDIO <= '0';      -- SDIO: Don't Care
when PxD12 =>
            SCLK <= '0';
            SDIO <= '0';
when LC =>
            SCLK <= '1';
```

```
                        SDIO <= '0';       -- SDIO: Don't Care
                    when RC =>
                        SCLK <= '1';
                        SDIO <= '0';       -- SDIO: Don't Care
                    when N1 =>
                        SCLK <= '1';
                        SDIO <= '0';       -- SDIO: Don't Care
                end case;
            end process;

            -- Handle reading of GPIO(2) [SDIO} pin
            process(CLOCK)
            begin
                if(rising_edge(CLOCK)) then
                    gpio_ff(0) <= CLK;
                    gpio_ff(1) <= gpio_ff(0);
                    if(gpio_ff(0) /= gpio_ff(1)) then
                        case Y is
                            when M7 =>
                                MOT_reg_val(to_integer(bit_num)) <= GPIO(0);
                            when DX7 =>
                                DX_reg_val(to_integer(bit_num)) <= GPIO(0);
                            when DY7 =>
                                DY_reg_val(to_integer(bit_num)) <= GPIO(0);
                            when PxD7 =>
                                px_val(to_integer(bit_num)) <= GPIO(0);
                            when others =>
                                null;
                        end case;
                    end if;
                end if;
            end process;



            -- Handle incrementing and resetting of 4 ms counter
--          process(Y,CLK)
--          begin
--
--              case Y is
--                  when W10|W14 =>
--                      counter_approx5ms <= counter_approx5ms + 1;
--
--                      when W16 =>
--                          counter_approx5ms <= (others => '0');
--                  when others =>
--                      counter_approx5ms_test <= counter_approx5ms;
--              end case;
--
```

```
--	end process;
	--counter_approx5ms_test <= counter_approx5ms;
	-- Handle decrementing and resetting of bit_num

	process(CLOCK)
	begin
		if(rising_edge(CLOCK)) then
			clk_ff(0) <= CLK;
			clk_ff(1) <= clk_ff(0);
				if(clk_ff(0) /= clk_ff(1)) then
					case Y is
						when
CF4|CF7|CF9|M3|M8|DX3|DX8|DY3|DY8|PxD3|PxD8 =>


									bit_num <= bit_num - 1;


						when CF2|M1|DX1|DY1|PxD1 =>
							bit_num <= x"6"; -- start at 7th bit-position
						when CF5|M6|DX6|DY6|PxD6 =>
							bit_num <= x"7"; -- start at 8th bit-position
						when CF6|CF8|CF3|M2|DX2|DY2|PxD2 =>
							bit_num_prev <= bit_num;
						when others =>
							null;
					end case;
				end if;
		end if;
	end process;


--	process(Y)
--	begin
--		case Y is
--			when CF4|CF7|CF9|M3|M8|DX3|DX8|DY3|DY8|PxD3|PxD8 =>
--				bit_num <= bit_num - 1;
--			when CF2|M1|DX1|DY1|PxD1 =>
--				bit_num <= x"6"; -- start at 7th bit-position
--			when CF5|M6|DX6|DY6|PxD6 =>
--				bit_num <= x"7"; -- start at 8th bit-position
--			when others =>
--				null;
--		end case;
--	end process;

	-- Handle flags for indicating memory write
	process(Y)
	begin
```

```
--          if rising_edge(CLOCK) then
--              mem_write_ff(0) <= CLK;
--              mem_write_ff(1) <= mem_write_ff(0);
--              if (mem_write_ff(0) /= mem_write_ff(1)) then
                    case Y is
                        when DX9 =>
                            mem_write <= "000001";
                        when DY9 =>
                            mem_write <= "000010";
                        when PxD9 =>
                            mem_write <= "000100";
                        when LC =>
                            mem_write <= "001000";
                        when RC =>
                            mem_write <= "010000";
                        when PxD11 =>
                            mem_write <= "100000";
                        when others =>
                            mem_write <= "000000";
                    end case;
--              end if;
--          end if;
        end process;

        -- Handle flags for indicated left and right clicks
        process(CLOCK)
        begin
            if rising_edge(CLOCK) then
                click_ff(0) <= CLK;
                click_ff(1) <= click_ff(0);
                if click_ff(0) /= click_ff(1) then
                    case Y is
                        when LC =>
                            lc_val <= GPIO(3);
                        when RC =>
                            rc_val <= GPIO(4);
                        when others =>
                            null;
                    end case;
                end if;
            end if;
        end process;

        -- Handle pixel dump flag
        process(CLOCK)
        begin
            if rising_edge(CLOCK) then
                pxd_en_ff(0) <= CLK;
                pxd_en_ff(1) <= pxd_en_ff(0);
```

```
                        if pxd_en_ff(0) /= pxd_en_ff(1) then
                                case Y is
                                        when PxD11 =>
                                                pxd_en <= '0';
                                        when Dx0 =>
--                                              if dump_skip = '0' then
                                                        pxd_en <= '1';
--                                                      dump_skip <= '1';
--                                              else
--                                                      dump_skip <= '0';
--                                              end if;
                                        when others =>
                                                null;
                                end case;
                        end if;
                end if;
        end process;

        -- Handle flags for px_addr incrementing
        process(CLOCK)
        begin
                if rising_edge(CLOCK) then
                        pxd_addr_ff(0) <= CLK;
                        pxd_addr_ff(1) <= pxd_addr_ff(0);
                        if pxd_addr_ff(0) /= pxd_addr_ff(1) then
                                case Y is
                                        when PxD0 =>
                                                px_addr <= X"00";
                                        when PxD10 =>
                                                px_addr <= px_addr + 1;
                                        when others =>
                                                null;
                                end case;
                        end if;
                end if;
        end process;

        --Change state of PD output
        process(Y)
        begin
                case Y is
                        when W12|W13 =>
                                PD <= '1';
                        when others =>
                                PD <= '0';
                end case;

        end process;
```

```
-- Handle incrementing and resetting of 100 us counter
process(CLOCK)
begin
        if reset = '1' then
                counter_100us <= (others => '0');
        elsif rising_edge(CLOCK) then
                case Y is
                        when W12|W0|W2|W4|W6|W8 =>
                                counter_100us <= counter_100us + 1;
                        when N1|CF10|M5|DX5|DY5|PxD5 =>
                                counter_100us <= (others => '0');
                        when others =>
                                counter_100us <= counter_100us;
                end case;
        end if;
end process;

-- Handle incrementing and resetting of 4 ms counter
process(CLOCK)
begin
        if reset = '1' then
                counter_approx5ms <= (others => '0');
        elsif rising_edge(CLOCK) then
                case Y is
                        when W10|W14 =>
                                counter_approx5ms <= counter_approx5ms + 1;
                                GPIO(6) <= counter_approx5ms(4);
                        when others =>
                                counter_approx5ms <= (others => '0');
                                GPIO(6) <= '0';

                end case;
        end if;
end process;

-- Handle incrementing of image sample
process(CLOCK)
begin
        if rising_edge(CLOCK) then
                img_smp_ff(0) <= CLK;
                img_smp_ff(1) <= img_smp_ff(0);
                if img_smp_ff(0) /= img_smp_ff(1) then
                        case Y is
                                when N1 =>
                                        img_smp <= img_smp + 1;
                                when others =>
                                        img_smp <= img_smp;
                        end case;
                end if;
```

```
			end if;
		end process;

end FSM;
```

seven_seg.vhd

```
-------------------------------------------------------------------------------
--
-- Seven segment display driver
--
-- David Calhoun
-- dmc2202@columbia.edu
--
-------------------------------------------------------------------------------

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


entity seven_seg is
port(
		inbits : in std_logic_vector(3 downto 0);
		outseg : out std_logic_vector(6 downto 0)

		);
end seven_seg;

architecture conv of seven_seg is

signal outsig : std_logic_vector(6 downto 0);

begin

		process (inbits)
		begin
			case inbits is

				when x"1" =>
					outsig <= "1111001";
				when x"2" =>
					outsig <= "0100100";
				when x"3" =>
					outsig <= "0110000";
				when x"4" =>
					outsig <= "0011001";
				when x"5" =>
					outsig <= "0010010";
```

```
                    when x"6" =>
                            outsig <= "0000010";
                    when x"7" =>
                            outsig <= "1111000";
                    when x"8" =>
                            outsig <= "0000000";
                    when x"9" =>
                            outsig <= "0011000";
                    when x"a" =>
                            outsig <= "0001000";
                    when x"b" =>
                            outsig <= "0000011";
                    when x"c" =>
                            outsig <= "1000110";
                    when x"d" =>
                            outsig <= "0100001";
                    when x"e" =>
                            outsig <= "0000110";
                    when x"f" =>
                            outsig <= "0001110";
                    when x"0" =>
                            outsig <= "1000000";
            end case;
    end process;

            outseg <= outsig;

end conv;
```

hello_world.c

```
#include <io.h>
#include <system.h>
#include <stdio.h>
//write data
#define IOWR_DATA(base, offset, data) \
        IOWR_16DIRECT(base, (offset) * 2, data)
//read data
#define IORD_DATA(base, offset) \
        IORD_16DIRECT(base, (offset) * 2)

#define RD_SYNC 0
#define RD_start 1
#define RD_readselects 2
#define WR_start 3
#define WR_readselects 4
#define WR_aggr 5
#define WR_box 6
#define WR_reset 7
```

```
#define RD_leftclick 0
#define RD_rightclick 1
#define RD_dx 2
#define RD_dy 3
#define RD_snum 4

/*void delay(int input){

   int i, j;
   for (i=0;i<input;i++){
      j=0;
   }

}*/

int main()
{
        //alt_u8 blank = 0;
        int i = 0;
        //bottom right corner of pixel dump
        alt_u16 xcoordinate = 0;
        alt_u16 ycoordinate = 0;

        int outOfBoundsX = 0;
        int outOfBoundsY= 0;
        alt_8 deltaX;
        alt_8 deltaY;

        int isOutOfBounds=0;
        //alt_8 lastDeltaX = IORD_DATA(GPIO_BASE, RD_dx);
        //alt_8 lastDeltaY = IORD_DATA(GPIO_BASE, RD_dy);
        //alt_u16 last_sample = IORD(GPIO_BASE, RD_snum);
        //alt_u16 this_sample;
        //alt_u16 writeable = 1;
        alt_u16 leftclick = 1;

        int xcoordinate_temp = 0;
        int ycoordinate_temp = 0;

        //alt_u16 start_RAM = 0;

        IOWR_DATA(VGA_BASE, WR_readselects, 1);
        IOWR_DATA(VGA_BASE,WR_start,0);
        IOWR_DATA(VGA_BASE, WR_reset, 0);
        IOWR_DATA(VGA_BASE, WR_box, 0);

        alt_u16 lastcurrentAllRAMS=0x0;

        for(;;){
```

```
//reset
if(IORD_DATA(GPIO_BASE, RD_rightclick)==0){
        IOWR_DATA(VGA_BASE, WR_aggr, 0);
        IOWR_DATA(VGA_BASE, WR_start, 0);
        xcoordinate = 0;
        ycoordinate = 0;
        outOfBoundsX = 0;
        outOfBoundsY = 0;
        isOutOfBounds=0;
        IOWR_DATA(VGA_BASE, WR_aggr, 1);
        for(i=0; i<15; i++){
                IOWR_DATA(VGA_BASE, WR_reset, 1);
        }
        IOWR_DATA(VGA_BASE, WR_aggr, 0);

        continue;
}
else {
        IOWR_DATA(VGA_BASE, WR_reset, 0);
        //read from address 1 for x position of sample
        deltaX = IORD_DATA(GPIO_BASE, RD_dx);

        //read from address 2 for y position of sample
        deltaY = IORD_DATA(GPIO_BASE, RD_dy);

        leftclick = IORD_DATA(GPIO_BASE, RD_leftclick);
        //IOWR_DATA(VGA_BASE, WR_reset, 0);



        alt_u16  currentAllRAMS = IORD_DATA(GPIO_BASE,RD_snum);
        //printf("%x\n",currentAllRAMS);

        if (currentAllRAMS == lastcurrentAllRAMS){
                continue;
        }
        alt_u16 currentRAM1 = currentAllRAMS & 0xF;
        alt_u16 currentRAM2 = (currentAllRAMS>>4) & 0xF;
        alt_u16 currentRAM3 = (currentAllRAMS>>8) & 0xF;
        alt_u16  currentRAM4 = (currentAllRAMS>>12) & 0xF;
        alt_u16  lastRAM1 = lastcurrentAllRAMS & 0xF;
        alt_u16  lastRAM2 = (lastcurrentAllRAMS>>4) & 0xF;
        alt_u16  lastRAM3 = (lastcurrentAllRAMS>>8) & 0xF;
        alt_u16  lastRAM4 = (lastcurrentAllRAMS>>12) & 0xF;
        alt_u16  ramWhichIsDifferent;
        if(currentRAM1!=lastRAM1){
                //printf("%x\n",currentAllRAMS);
```

```
				ramWhichIsDifferent = 0x1;
			}
			else if(currentRAM2!=lastRAM2){
				//printf("%x\n",currentAllRAMS);
				ramWhichIsDifferent=0x2;
			}
			else if(currentRAM3!=lastRAM3){
				//printf("%x\n",currentAllRAMS);
				ramWhichIsDifferent=0x4;
			}
			else{
				//printf("%x\n",currentAllRAMS);
				ramWhichIsDifferent=0x8;
			}
			IOWR(VGA_BASE, WR_readselects, ramWhichIsDifferent);

			lastcurrentAllRAMS = currentAllRAMS;


			if (!isOutOfBounds){
				xcoordinate_temp=xcoordinate-deltaX;
				ycoordinate_temp=ycoordinate+deltaY;

			}else{
				xcoordinate_temp=outOfBoundsX-deltaX;
				ycoordinate_temp=outOfBoundsY+deltaY;
			}

			//Boundary conditions
			//printf("xcoordinate_temp is %d\n", xcoordinate_temp);
			//printf("ycoordinate_temp is %d\n", ycoordinate_temp);
			//IOWR_DATA(VGA_BASE, WR_box, 0);
			//IOWR_DATA(VGA_BASE, WR_aggr, 0);
			if(xcoordinate_temp>=0 && ycoordinate_temp>=0 && xcoordinate_temp<=112
&& ycoordinate_temp<=112){
				xcoordinate-=deltaX;
				ycoordinate+=deltaY;

				outOfBoundsX = xcoordinate;
				outOfBoundsY = ycoordinate;

				if ((deltaX == 0) && (deltaY == 0)){
					printf("er\n");
				}

				//write back new position
				IOWR_DATA(VGA_BASE, WR_start, ycoordinate+xcoordinate*128);

				if((!leftclick)){
```

```
				//green if writing
				IOWR_DATA(VGA_BASE, WR_box, 1);
				for(i=0; i<15; i++){
					IOWR_DATA(VGA_BASE, WR_aggr, 1);
				}

				//printf("ag\n");
				IOWR_DATA(VGA_BASE, WR_aggr, 0);
			}else{
				//yellow if not writing
				IOWR_DATA(VGA_BASE, WR_box, 0);
				IOWR_DATA(VGA_BASE, WR_aggr, 0);
			}
			isOutOfBounds=0;
		}else{
			//printf("got here");
			outOfBoundsX-=deltaX;
			outOfBoundsY+=deltaY;

			//red if out of bounds
			IOWR_DATA(VGA_BASE, WR_box, 2);

			isOutOfBounds=1;

			if(xcoordinate_temp<0){
				//bottom right corner
				if(ycoordinate_temp<0){
					IOWR_DATA(VGA_BASE, WR_start, 0);
				//top right corner
				}else if(ycoordinate_temp>112){
					IOWR_DATA(VGA_BASE, WR_start, 112);
				}else{
					ycoordinate=ycoordinate_temp;
					IOWR_DATA(VGA_BASE, WR_start,
ycoordinate_temp);
				}
			}
			else if(ycoordinate_temp<0){
				//bottom right corner
				if(xcoordinate_temp<0){
					IOWR_DATA(VGA_BASE, WR_start, 0);
				//bottom left corner
				}else if(xcoordinate_temp>112){
					IOWR_DATA(VGA_BASE, WR_start, 112*128);
				}else{
					xcoordinate=xcoordinate_temp;
					IOWR_DATA(VGA_BASE, WR_start,
xcoordinate_temp*128);
				}
```

```
                                        }
                                        else if(xcoordinate_temp>112){
                                                  //bottom left corner
                                                  if(ycoordinate_temp<0){
                                                            IOWR_DATA(VGA_BASE, WR_start, 112*128);
                                                  //top left corner
                                                  }else if(ycoordinate_temp>112){
                                                            IOWR_DATA(VGA_BASE, WR_start, 112+112*128);
                                                  }else{
                                                            ycoordinate=ycoordinate_temp;
                                                            IOWR_DATA(VGA_BASE, WR_start, 112*128 +
ycoordinate_temp);
                                                  }
                                        }
                                        else if(ycoordinate_temp>112){
                                                  //top right corner
                                                  if(xcoordinate_temp<0){
                                                            IOWR_DATA(VGA_BASE, WR_start, 112);
                                                  //top left corner
                                                  }else if(xcoordinate_temp>112){
                                                            IOWR_DATA(VGA_BASE, WR_start, 112+112*128);
                                                  }else{
                                                            xcoordinate=xcoordinate_temp;
                                                            IOWR_DATA(VGA_BASE, WR_start,
xcoordinate_temp*128 + 112);
                                                  }
                                        }
                              }
                    }
          }
}
```