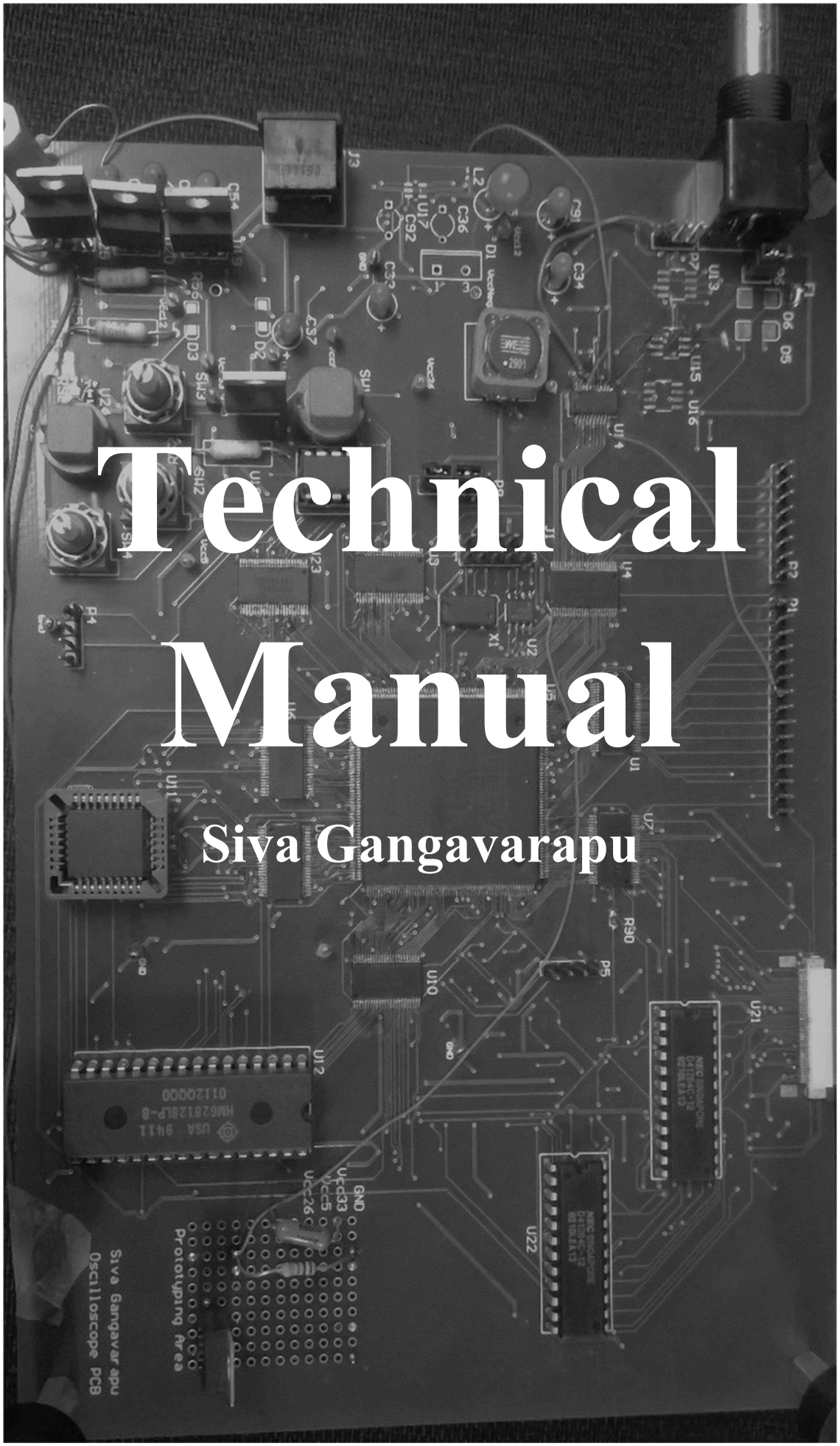


Technical Manual

Siva Gangavarapu

Table of Contents

Hardware Technical Manual

General System Overview……3
Power Circuitry……7
FPGA and Related Circuitry……...9
Reset…….....13
Serial Configuration Device……...14
ROM and SRAM……14
ROM……...17
SRAM……18
Keypad……..21
VRAM, VRAM Controller, Display Controller, External Display……24
VRAM Block……25
Signal Timing and Function for U21 and U22……...27
Display Block……38
Signal Timing for Display Block……..40
Analog……...43

Software Technical Manual

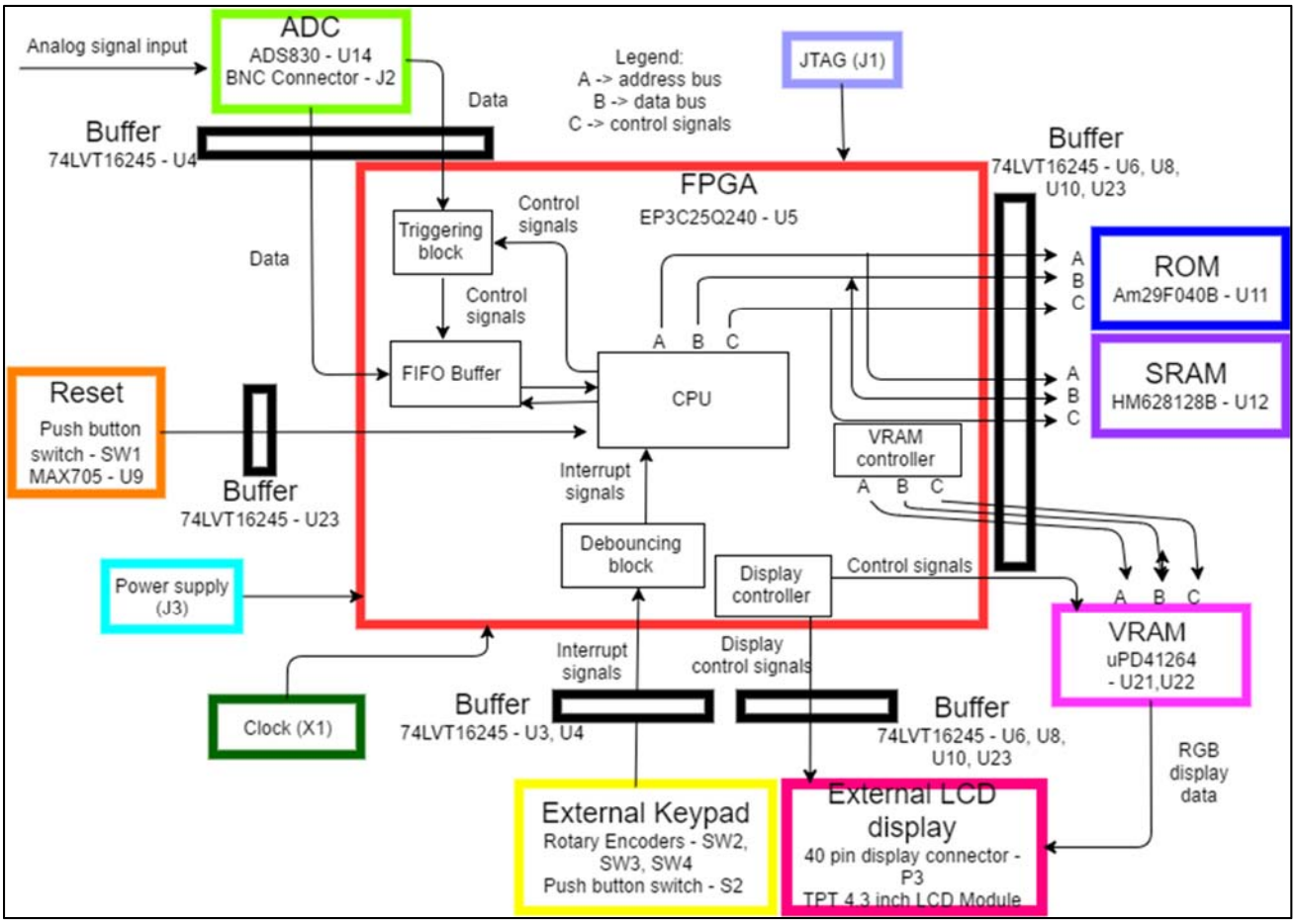
System Overview……47
Altera© Nios II Soft Processor Design……..51
Clock, Reset, ROM/SRAM……53
Buffer Software……...53
Keypad Logic and Software……...55
VRAM Controller……...64
Display Controller and Display Software……71
Display Signal Generation……71
Display Software Control……75
Analog Logic and Software……78

Design Changes……95

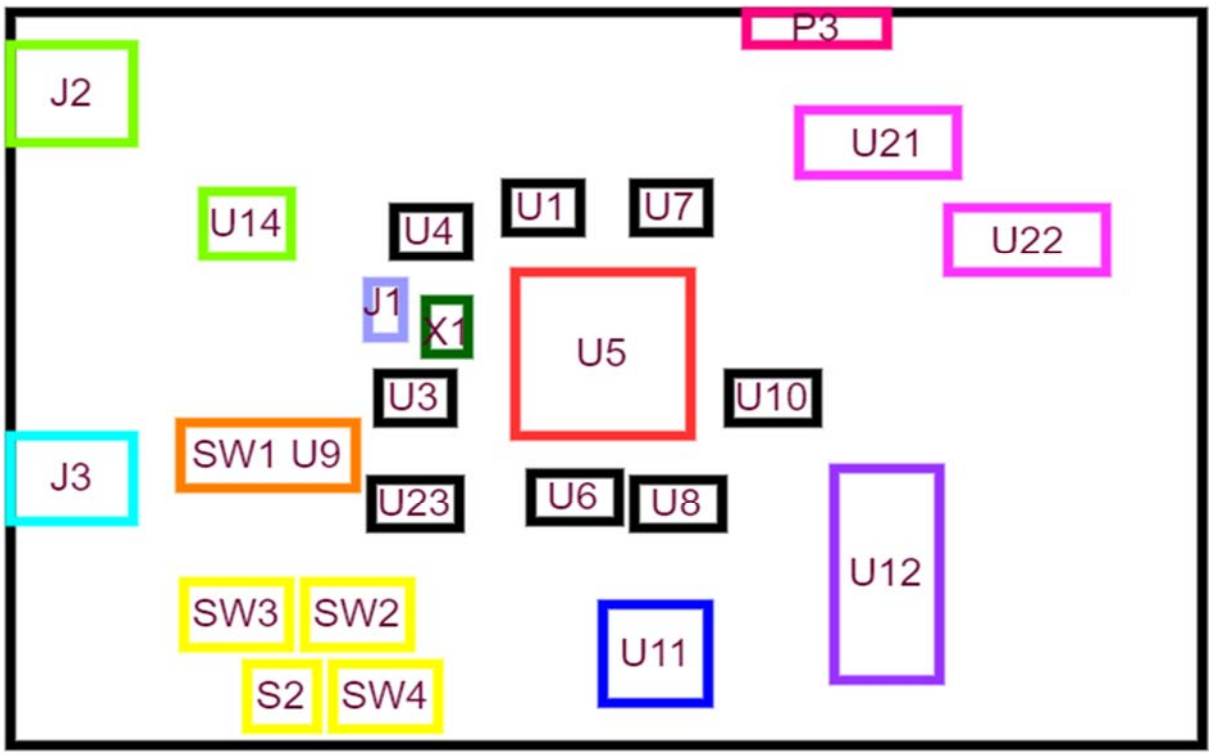
HARDWARE

General System Overview

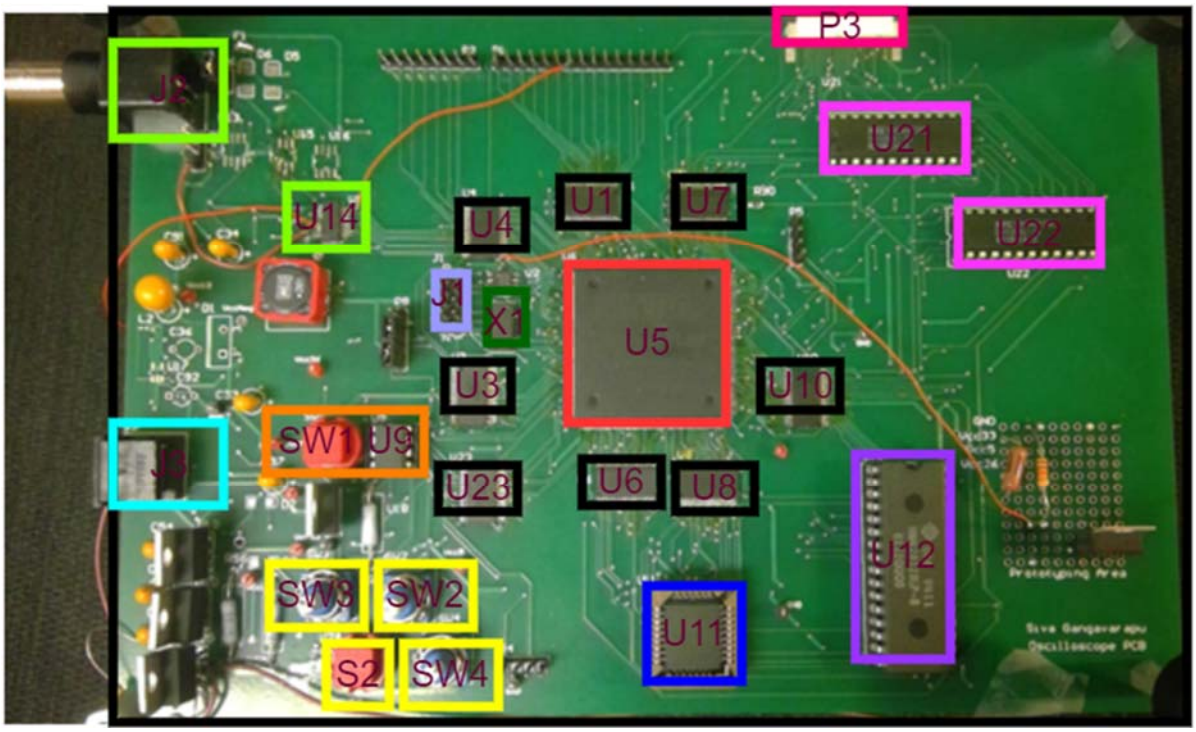
The Dual-ViewOscope is a System on Programmable Chip (SoPC) FPGA/microprocessor based oscilloscope system that runs on the 240 pin EP3C25 Cyclone III FPGA by Altera. The following system block diagram and board layout highlights the major components of the system



General system block diagram*: This block diagram is not an accurate representation of component layout on the PCB. The following is a depiction of the board layout. The color coding of the block diagram corresponds to the board layout on the following page. An image of the board layout juxtaposed on top of the actual PCB, as well as an image of the actual PCB, is also included in order to clarify component placement.*



Color coded board layout: *General board layout (measurements not exact, but roughly to scale) with a color coding corresponding to the aforementioned system block diagram. The board layout highlights only major board components and ignores the smaller, passive components.*



Color coded PCB layout: *Color-coded board layout juxtaposed on actual PCB to allow for easy reference.*

OVERVIEW OF SYSTEM FUNCTIONS

Analog input signal, which is intended to be displayed to the user, is received and converted to a digital signal by component U14 (ADS830 ADC) before being buffered by U4 (74LVT16245 16-bit bus transceiver) and inputted into component U5 (FPGA). This analog data input bus is processed by the FPGA's triggering logic block and transferred to the FIFO block for temporary storage as sample data.

Another set of major inputs to the FPGA is the keypad signals (including both rotary encoders and push buttons for keypresses). The rotary encoders are components SW3, SW2, and SW4, and the push button is S2. Keypress and encoder signals generated by user operation of these switches are buffered and processed by the FPGA for decoding into interrupts that are handled by the CPU (an embedded Nios II soft processor). The CPU processes these signals and responds by modifying parameters (such as trigger level, sweeping rate, etc.) integral to the triggering and FIFO logic. After the analog data is sent through the triggering logic and stored in the FIFO as sampled data, it is sent out for display.

Components U21 and U22, two uPD41264 NEC 64K x 5 Dual-Port Graphic DRAM chips, comprise the system's VRAM block, which communicates with logic programmed in the FPGA as the VRAM controller using control signals, an 8-bit address bus, and an 8-bit data bus. The sample graphics data, as well as other display data such as the axis and menu, is stored in components U21 and U22 prior to being displayed on the external LCD.

The external display, a 480RGBx272 TFT LCD, is connected to the PCB by component P3, a 40-pin SMD horizontal-type ZIF connector (part number ER-CON40HT-1)**.** It is driven by control signals originating logic programmed in the FPGA as part of the display controller. The display also serially receives the RGB graphics data stored in U21 and U22, which is then processed for display by the onboard OTA5180A System-On-Chip Driver

Aside from U21 and U22, memory for the oscilloscope is provided by components U11, an Am29F040B 512K x 8 EEROM (ROM), and U12, a HM628128B 128K x 8 CMOS Static RAM (SRAM). Both U11 and U12 share the same unidirectional 19-bit address bus, 8-bit data bus (different from the bus communicated with U21 and U22), write-enable signal, and output-enable signals. Chip select for each component is different.

Reset functionality for the oscilloscope is provided by components SW1, a push-button switch, and U9, a MAX705 supervisory circuit. The buffered reset set provides system-wide reset for every component block, including the ADC, VRAM controller, display controller, ROM, SRAM, and CPU.

Finally, component X1 provides the system with a 36 MHz clock. The signal is inputted into U5 without any buffering.

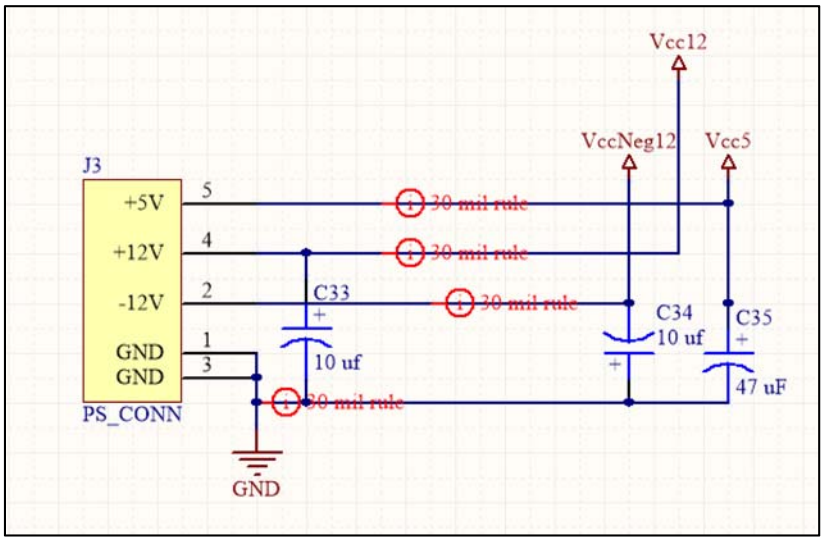
Special Note about Buffers:

Every input and output signal associated with U5, including unused I/O pins and signals associated with the ROM, SRAM, VRAM, display, keypad, reset, and analog block, is buffered.

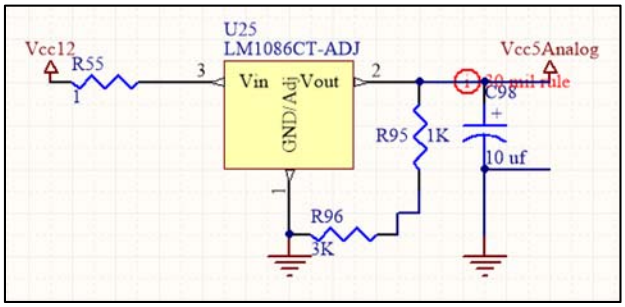
These buffers, which are 74LVT16245 16-bit bus transceivers, are marked as components with designators U1, U7, U10, U8, U6, U23, U3, and U4. Signals passing through theses buffers are also affected by a buffer delay (minimum 500ps, maximum 6.5ns). These delays are accounted for in the subsequent signal timing diagrams in this manual

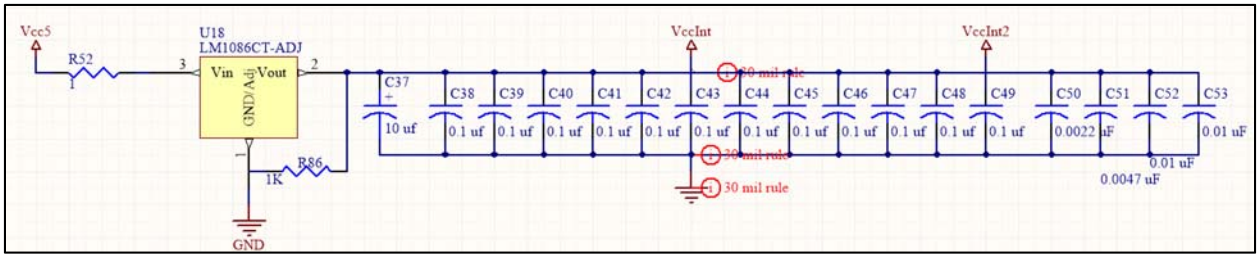
Power Circuitry

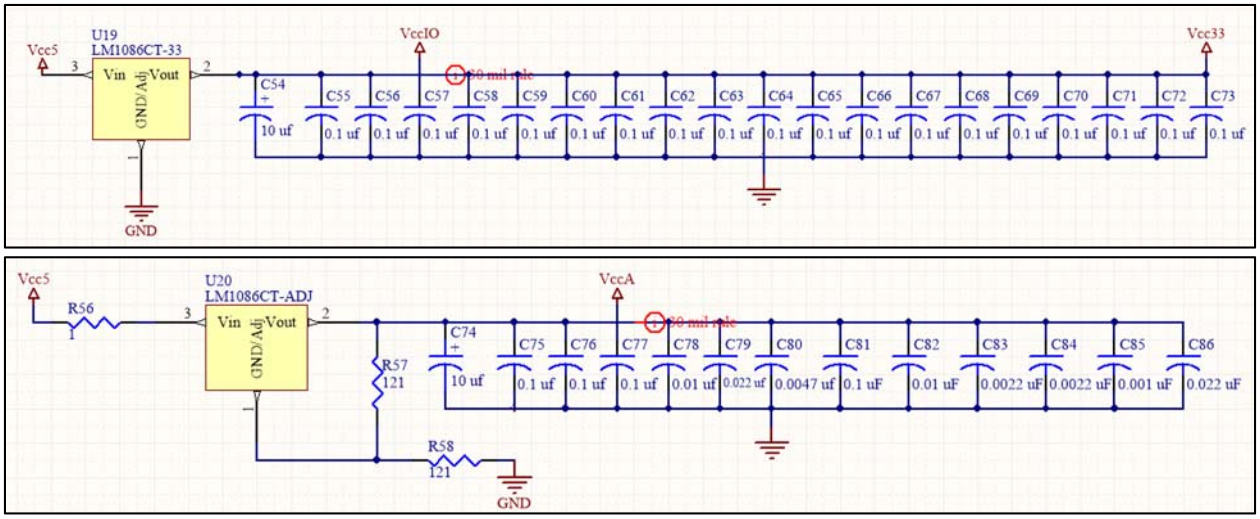
The primary power input for the system comes through the power connector component J3, the schematic for which is shown below.



Generation of various other power buses is provided through the use of LM1086 voltage regulators (components U18, U19, and U25). The schematics for the power buses that are generated using these components is shown below







Power Bus Circuitry Schematics: *Schematics for the generation of power buses Vcc5Analog, VccInt, VccInt2, VccIO, Vcc33, and VccA from the power sources directly offered by J3. All voltage regulator circuits other than U19 use the ADJ form of the LM1086, while U19 uses the fixed 3.3 voltage variety.*

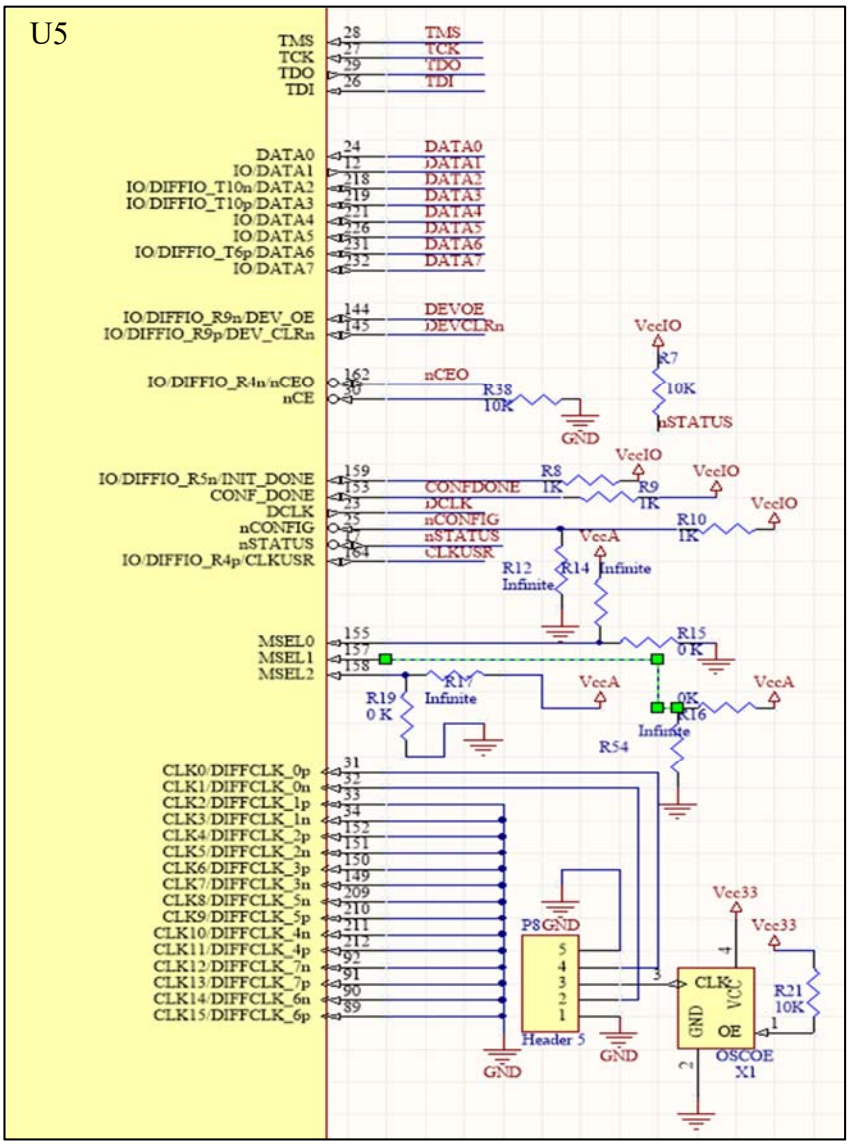
The following chart shows the actual voltage values of the various power buses shown in the schematics above.

Power Bus	**Voltage (V)**
Vcc5	5
Vcc12	12
VccNeg12	-12
VccInt	1.25
VccInt2	1.25
VccIO	3.3
Vcc33	3.3
VccA	2.5

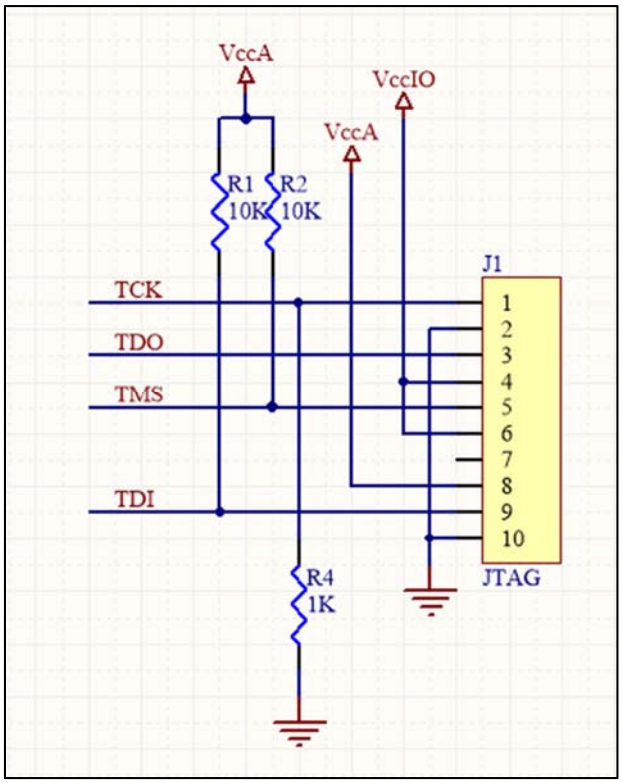
FPGA and Related Circuitry

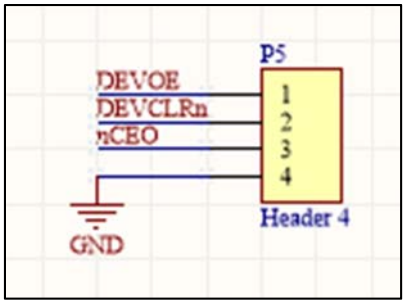
The FPGA and related circuitry block includes components U5, the EP3C25Q240 FPGA, J1, the JTAG Connector, U2, the EPCS16 serial configuration device, and X1, the external oscillator. The following schematic shows the signal connections for these components and associated passive components.

The schematic for U5 will be presented in portions, since the chip is very large. The following schematic illustrates the component's configurational, clock, and JTAG connections (non-input/output or power), as well as other related componenents.

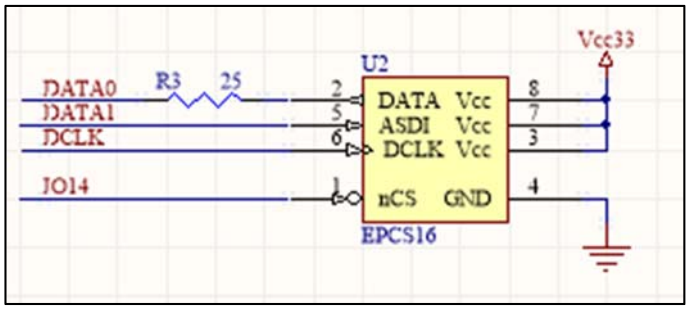


Right Side U5 Schematic: *This schematic excerpt highlights JTAG, configurational, and clock pin connections*





Schematic for P5 (Pinout header)



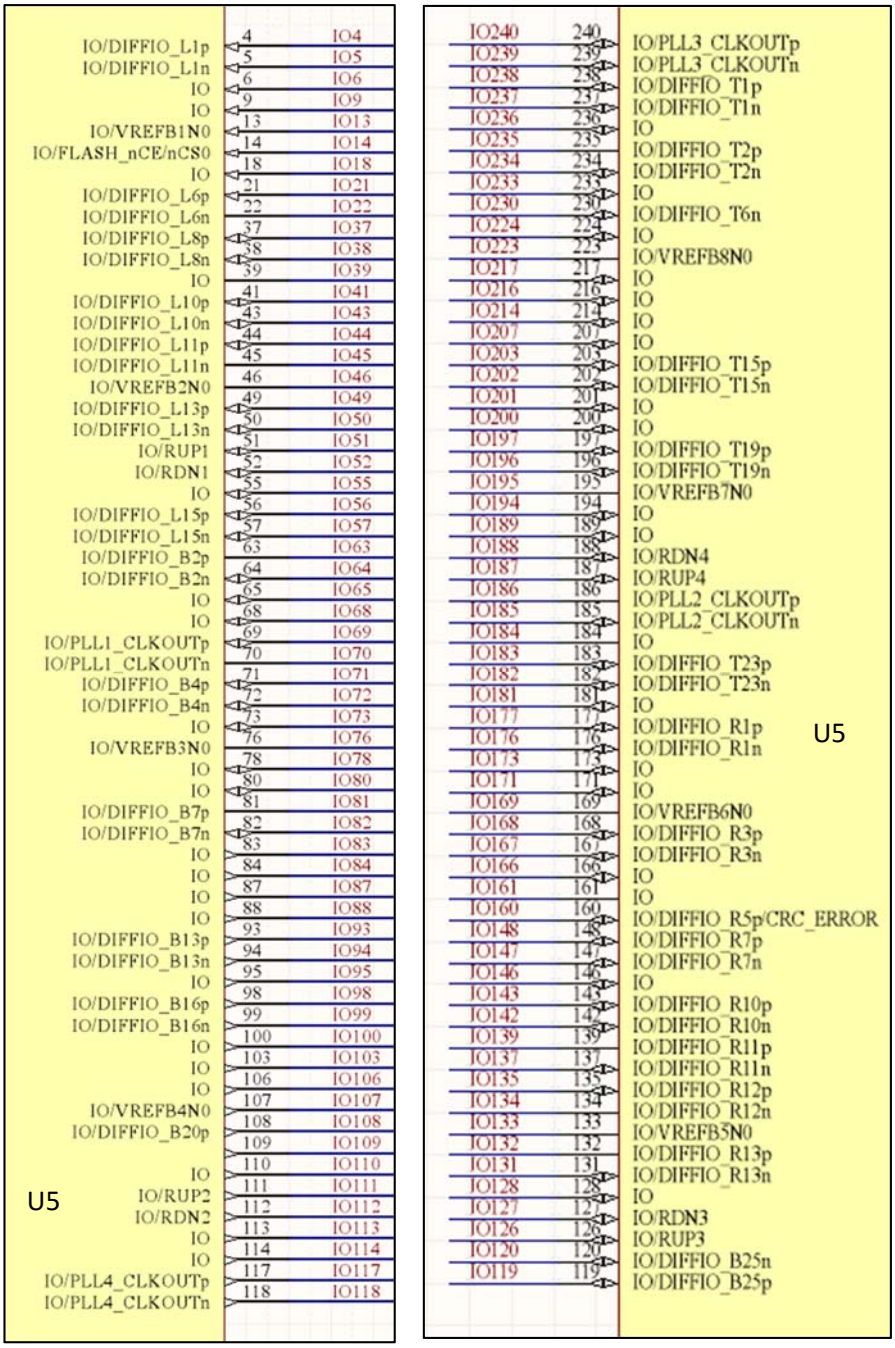
Schematic for J1 (JTAG Connector)

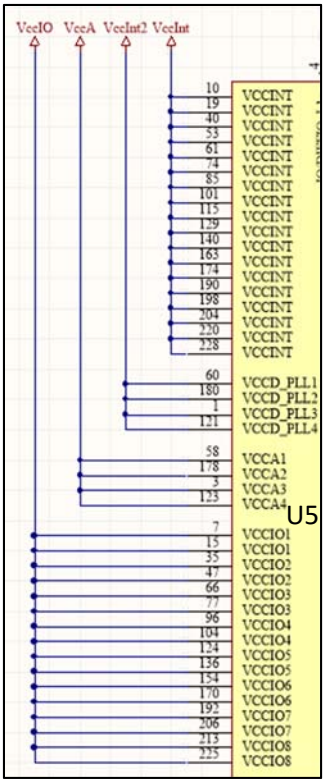
Schematic for U2 (serial configuration device)

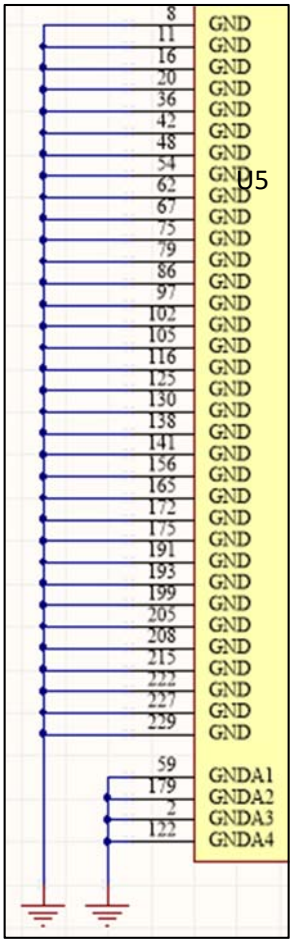
Component P8 allows for the user to configure the system to run on the clock source provided by component X1 or an external clock source, if desired. The board comes with two different shunts on P8 so pins 3 and 4 in component P8 share a connection, as do pins 1 and 2, allowing the system to run on the 36 MHz frequency oscillator from X1.

In addition to the JTAG and clock connections, the ***Right Side U5 Schematic*** shows the connections for the configurational pins as well. These pins are connected for U5 to operate in AS (active serial) configurational mode. Therefore, MSEL pins 0 and 1 are shorted to VccA, and pin 2 to GND. In addition, nSTATUS, nCONFIG, CONF_DONE, and INIT_DONE are pulled high to VccA using a 10 kOhms resistor. Also, pin nCE is pulled low to GND to enable the device.

The following schematics show I/O pin connections on U5. In addition to several unused pins and direction control for the buffers, these I/O pins are involved in all signaling to the analog, ROM, SRAM, VRAM, display, keypad, and reset blocks.





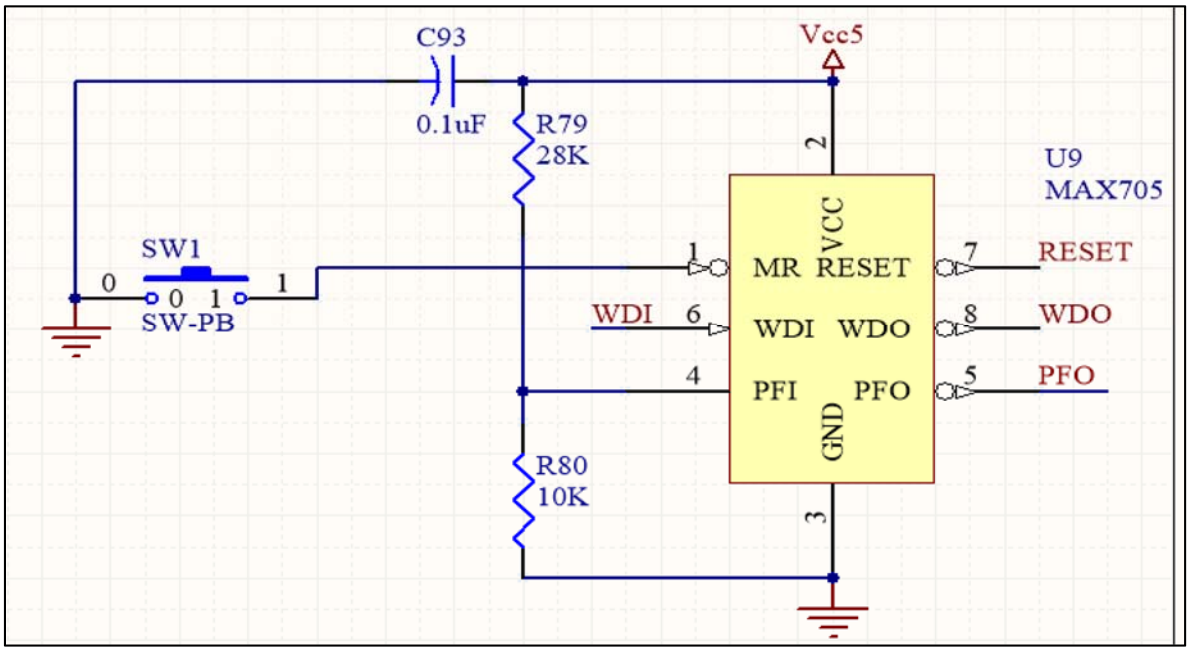


I/O and Power Pin Configuration Schematic for U5*. Note: To see relationship between these I/O pins and the software setup, refer to* ***PinList Table*** at the end of the document**.** The following diagram illustrates the location of U5 in the PCB, as well as the pin numbers corresponding to the schematic.

PCB Location and Pin Numbers for Component U5: *Marked pin numbers on the diagram correspond to the pin numbers in U5 schematic figures shown above.*

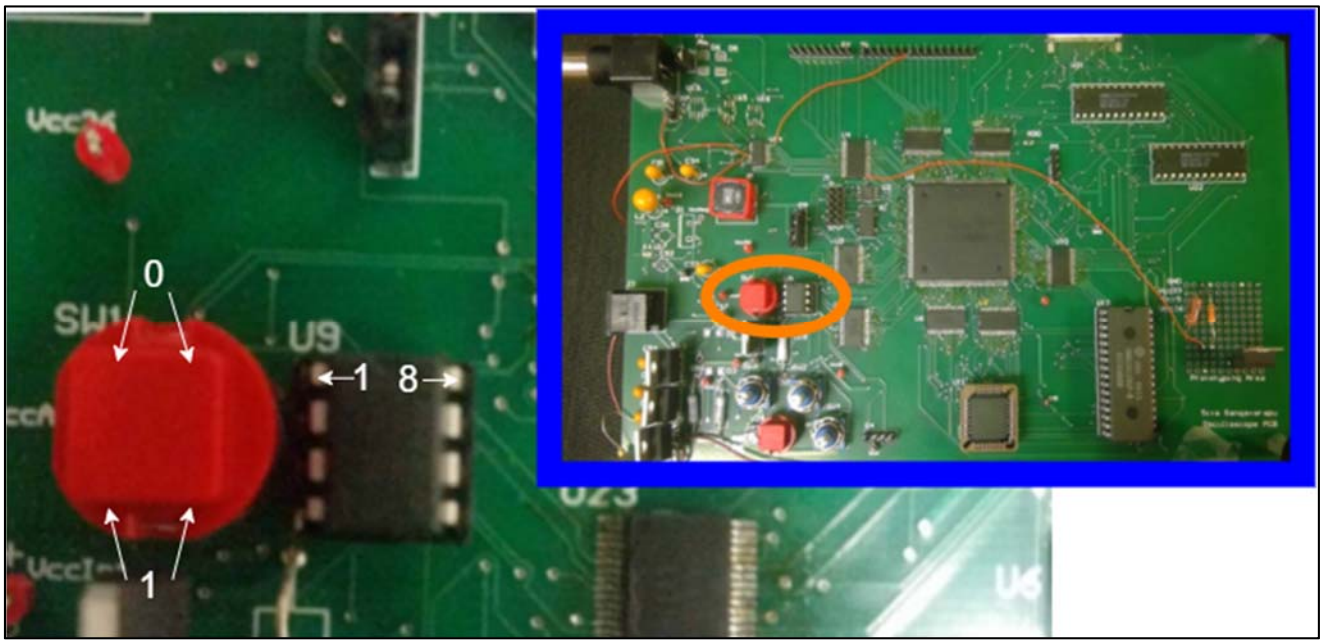
Reset

System-wide reset is provided to the system through components SW1, a push button switch, and U9, a MAX705 supervisory circuit component. The schematics for these components are shown below.



Schematics for Components U9 and SW1: *Schematics for the system reset circuitry.*

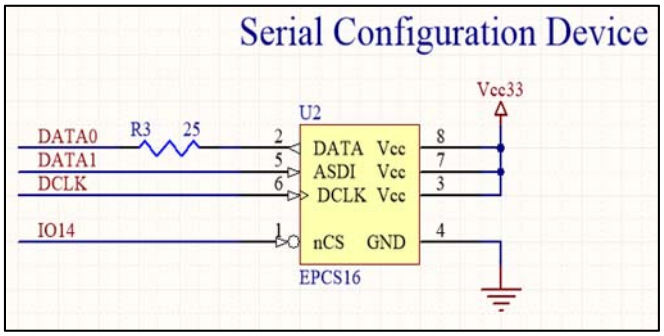
The following diagram illustrates the locations of components U9 and SW1 the PCB, as well as the pin numbers corresponding to the schematic.



PCB Location and Pin Numbers for Components U9 and SW1: *The marked pin numbers on the diagram correspond to the schematic pin numbers in* ***Schematic for U9 and SW1***. Pins on SW1 cannot be seen in this picture, as they are only visible on the bottom layer of this switch.

These signals are buffered through component U23, the schematic for which is shown in diagram ***Buffer Schematics for U11 and U12.*** Once the RESET signal is buffered and received by component U5, it is used as the system reset (see software technical manual and ***Pin List Table.***

Serial Configuration Device



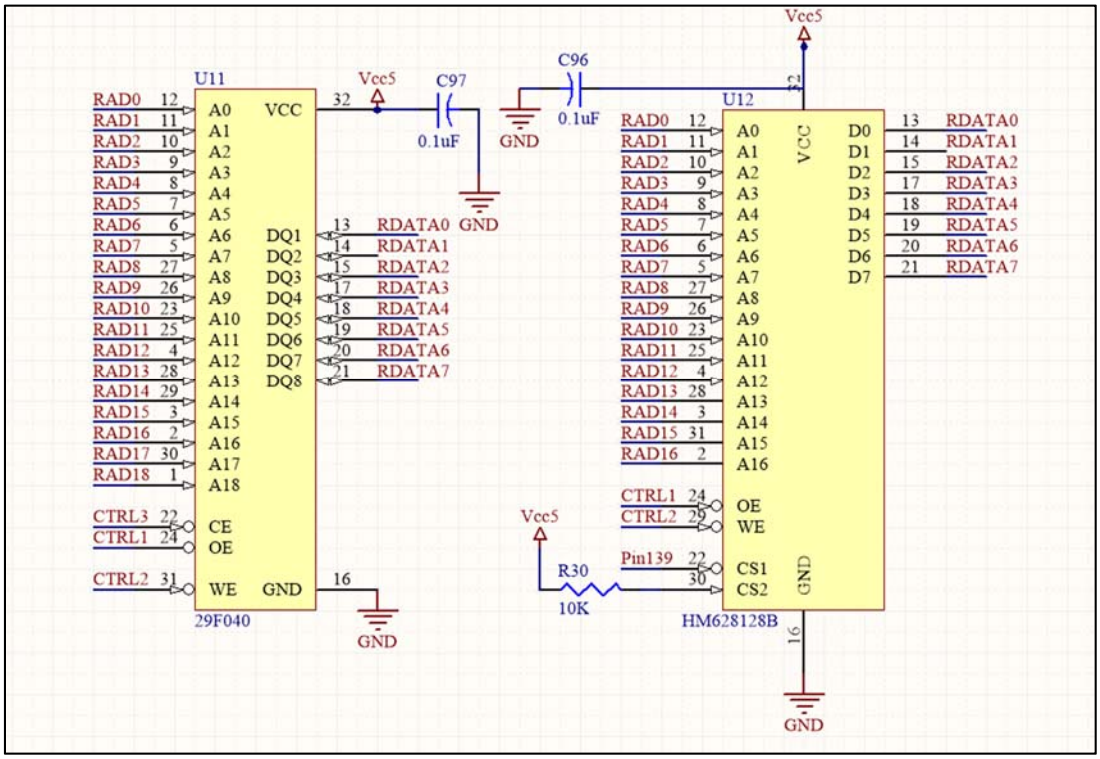
The Dual-ViewOscope® utilizes the EPCS16 flash memory device (component U2) to store configurational data for the FPGA. The device supports the active serial (AS) configurational scheme, corresponding to the MSEL pin configuration described in the **FPGA and Related Circuitry** section. The schematic for U2 is shown to the left.

ROM and SRAM

The ROM and SRAM blocks provide non-volatile read-only and volatile, respectively, memory for the oscilloscope. The following memory map highlights the location of these components in system memory

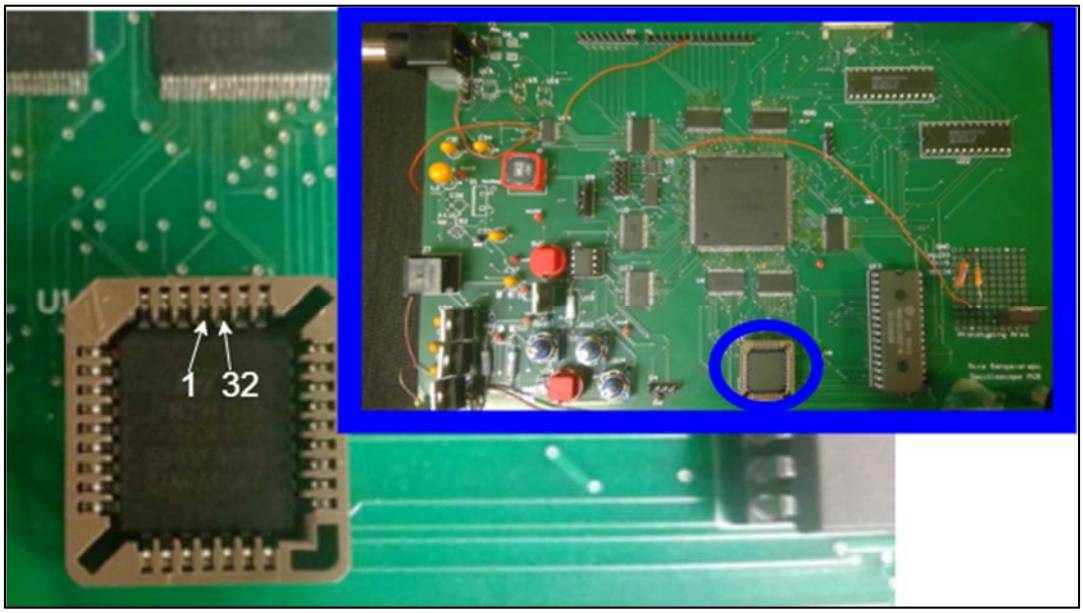
Component	**Memory Address (Start – End)**
ROM	0x0008000 – 0x000FFFF
SRAM	0x00120000-0x0013FFFF
Trigger Enable	0x00181170-0x0018117F
Sample Rate	0x00181160-0x0018116F
Auto Trigger	0x00181150-0x0018115F
Trigger Delay	0x00181140-0x0018114F
Trigger Slope	0x00181130-0x0018113F
Trigger Level	0x00181120-0x0018112F
FIFO Request	0x00181110-0x0018111F
On-chip memory	0x00170000-0x00179FFF
Keypress	0x00181100-0x001810FF
VRAM	0x00150000-0x0015FFFF
FIFO Data	0x001810F0-0x001810FF
FIFO Full	0x001810E0-0x001810EF

The schematic for U11 (Am29F040B) and U12 (HM628128B) is shown below.

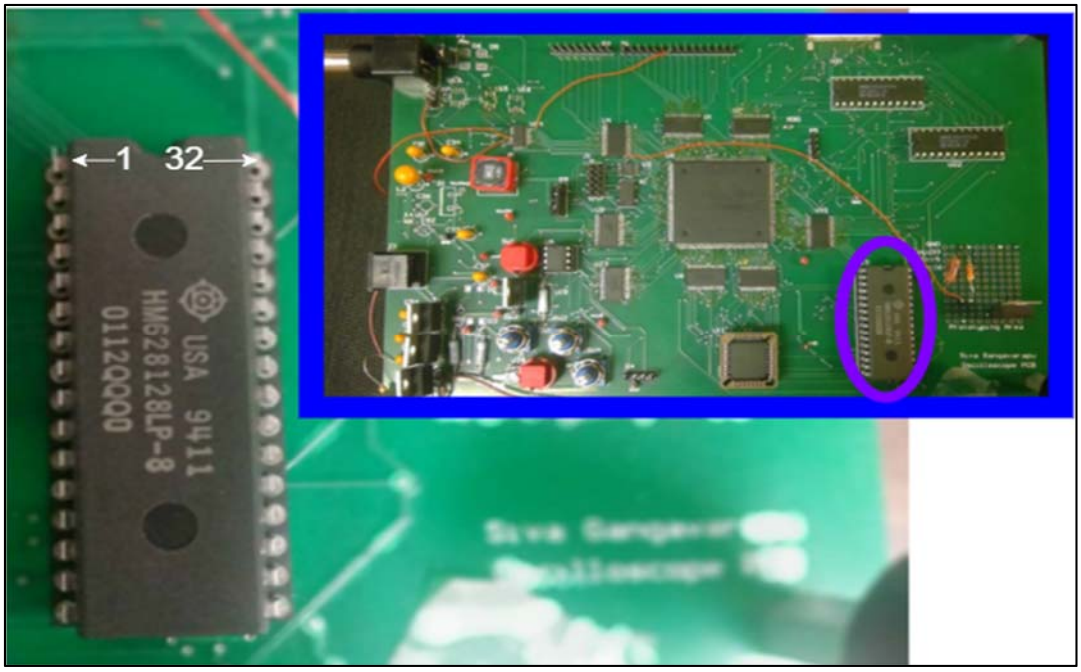


Schematic for U11 and U12: *Schematics for the ROM and SRAM circuitry*

The following diagrams illustrates the locations of component U11 and U12 in the PCB, as well as the pin numbers corresponding to the schematic.

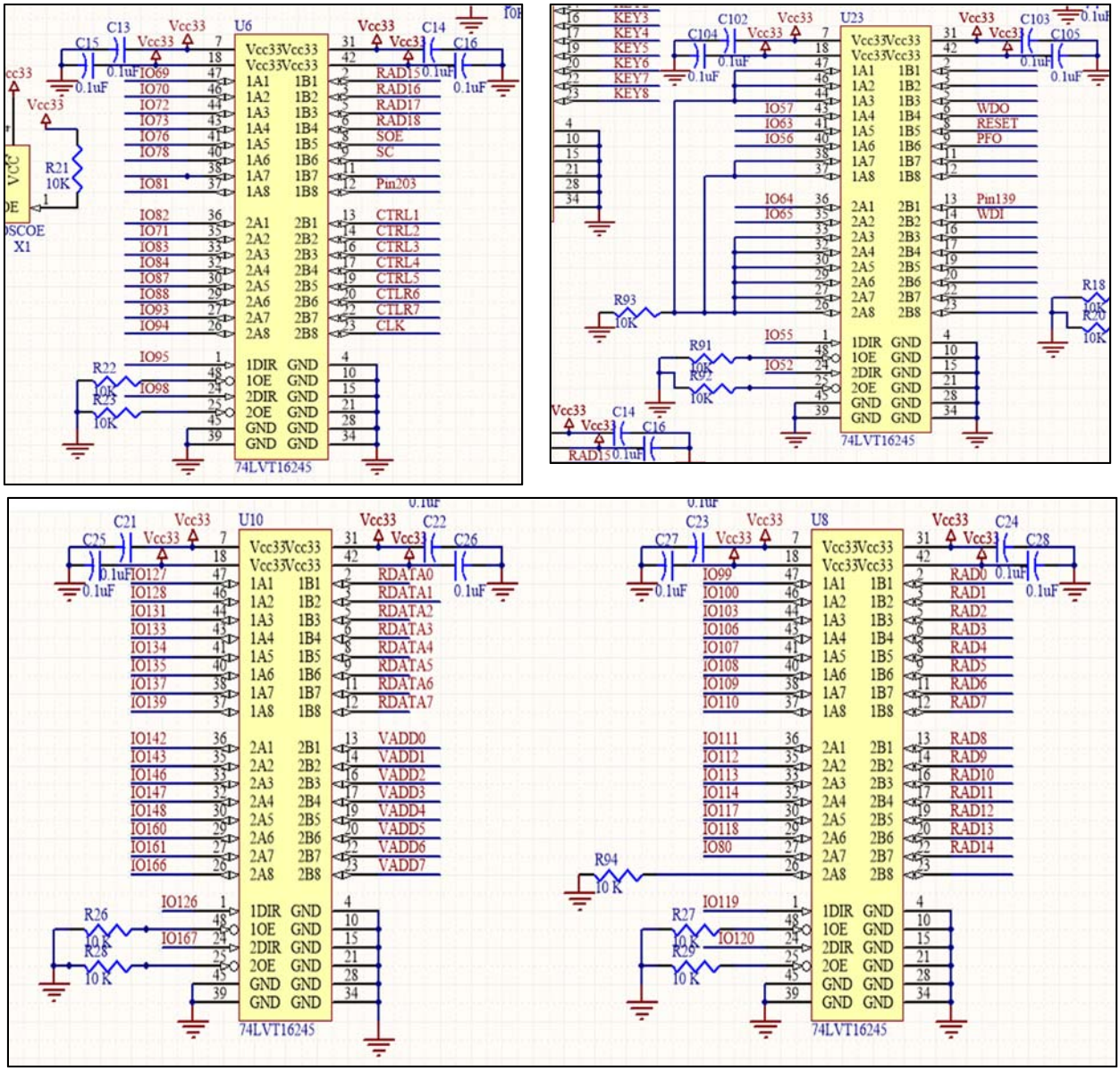


PCB Location and Pin Numbers for Component U11: *The marked pin numbers on the diagram correspond to the schematic pin numbers in* ***Schematic for U11 and U12*** for component U11.



PCB Location and Pin Numbers for Component U12: *The marked pin numbers on the diagram correspond to the schematic pin numbers in* ***Schematic for U11 and U12*** for component U12.

Both U11 and U12 share a unidirectional 19-bit address bus (RAD0-18), a bidirectional 8-bit data bus (RDATA0-7), and write-enable (CTRL2) and output-enable (CTRL1) signals. U11 and U12 are controlled by different chip-select signals (CTRL3 for U11, Pin139 for CS1). The address bus is buffered through buffers U6 and U8, data bus through U10, WE/OE through U6, and chip-selects through U6 and U23 for U11 and U12, respectively. The buffer control for U11 and U12 is highlighted in the following representation of the buffers' schematic and board layout.



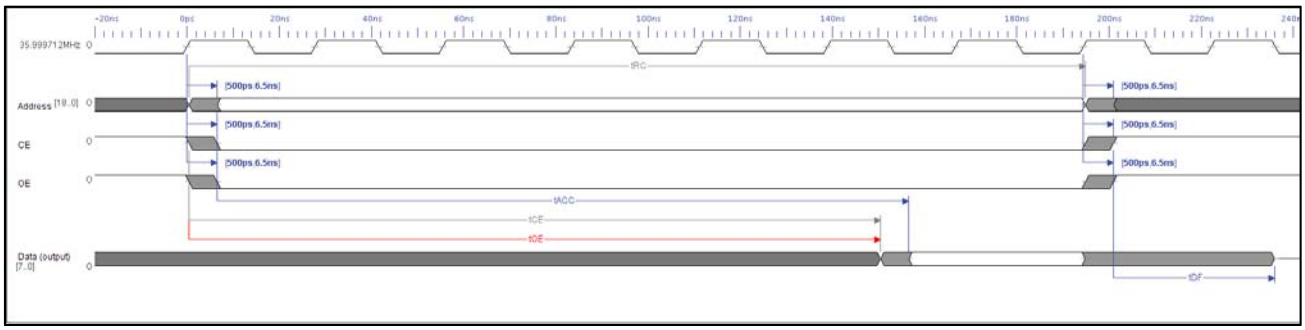
Buffer Schematics for U11 and U12: *Starting clockwise from top-left: U6 (chip select for U11, WE, OE, bits 19-16 of address bus), U23 (chip select for U12), U8 (bits 15-1 of address bus), U10 (data bus). See the* ***Buffer Direction Control*** *section of the software technical manual to see the direction pin configurations for these buffers.*

The ***Buffer Schematics for U11 and U12*** diagram also shows the connections to the FPGA for the buses and control signals before they are buffered. The corresponding IO pin connections for U5 can be found in the diagram ***I/O and Power Pin Configuration Schematic for U5***

ROM

U11 is powered off of a single 5 V power source, which is connected to a single 0.1 uF bypass capacitor (C97).

The timing diagram for signal control of U11 is depicted below.



U11 Read Cycle Timing.

A read operation on U11 is executed by bringing chip-select (CE), OE, and WE, all active low signals, low. The address to output delay (tACC) and chip enable to output delay (tOE) dictate that data becomes valid to be read six clock cycles after the address bus becomes valid. Data remains valid for one full clock cycle. After seven clock cycles, CE and OE are asserted high and inactive. Data output then becomes invalid, then enter high impedance after a delay. The following table illustrates the timing parameters and requirements for a read cycle.

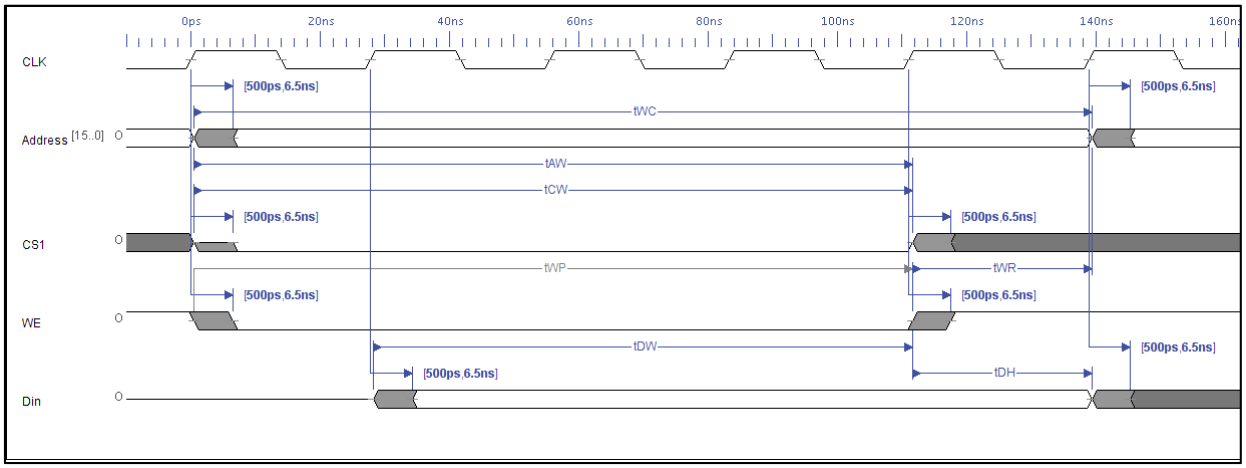
Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>150	188
T_{ACC} (address to output delay)	Max 150	-
T_{CE} (chip enable to output delay)	Max 150	-
T_{OE} (output enable to output delay)	Max 55	-
T_{DF} (chip enable to output high Z)	Max 35	-
T_{OH} (output hold time from address, CE, or OE, whichever comes first)	>0	0
T_{OEH} (output enable hold time)	>0	Met (WE not shown on diagram)

SRAM

U12 is also powered off a single 5 V power source, which is bypassed to ground by a 0.1 uF capacitor (C96). Its addressing is provided by bits 16-0 (discarding the upper two bits) of the shared RAM/ROM address bus (RAD16-0), and it data line requires the full 8 bits of the shared data bus (RDATA7-0).

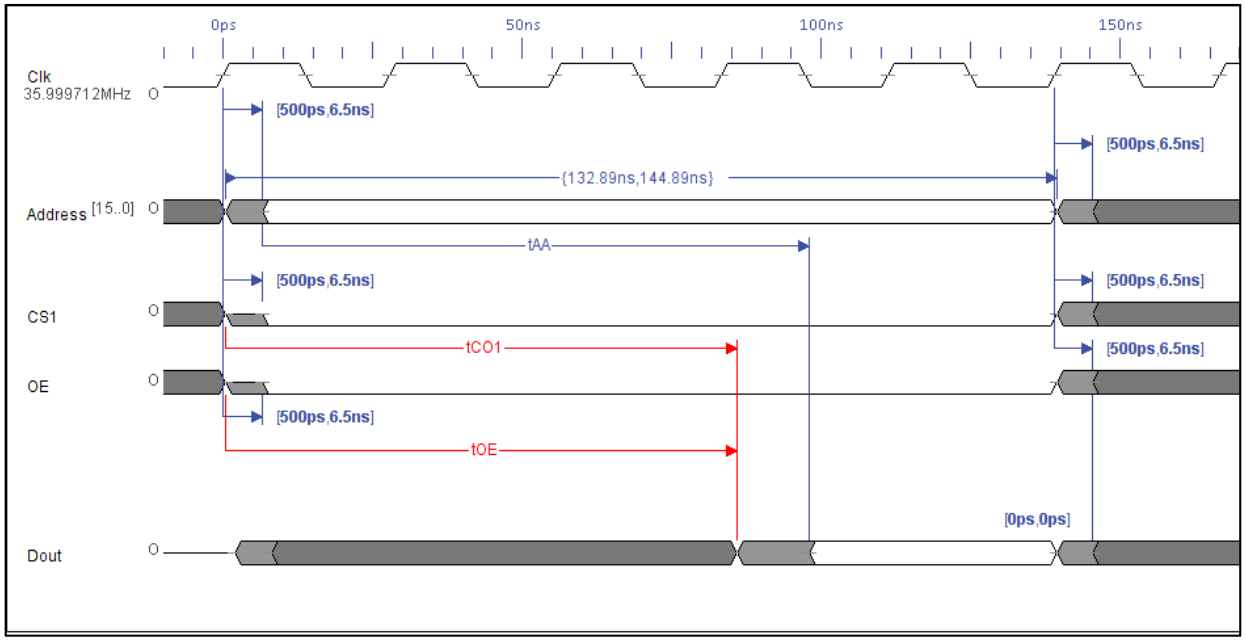
Operation of U12 requires CS1 to by logic high when on standby, and logic low during read and write cycles. However, while CS2 is required to be high during read and write cycles, its status on standby can be either high or low. Therefore, CS2 is tied logic high to V_{IH} (5 V) by a 10 K series resistor (R30).

The following diagrams illustrate the signal timing during write and read cycles for U12. For the aforementioned reason, CS2 is not shown on either diagram.



U12 Write Cycle Timing*. The write cycle begins with CS1 and WE going low while CS2 is high. The cycle ends with CS1 and WE going high. In between, data entering U12 on the data bus is valid and written to the device.*

Timing Parameter	Required (ns)	Actual (minimum value in ns)
T_{WC} (write cycle time)	>85	132.89
T_{CW} (chip selection to end of write)	>75	105.112
T_{AS} (address setup time)	>0	0.5
T_{AW} (address valid to end of write)	>75	105.112
T_{WP} (write pulse width)	>55	105.112
T_{WR} (write recovery time)	>0	21.778
T_{DW} (data to write time overlap)	>35	77.334
T_{DH} (data hold from write time)	>0	21.778



U11 Read Cycle Timing. *The time at which valid data appears on the data bus is given by three different access times: t_{AA} (address access time), t_{C1} (chip selection to output), and t_{OE} (output enable to output valid). Since t_{AA} and t_{C1} are both longer than t_{OE}, the timing diagram shows that data output becomes valid four clock cycles after the beginning of the cycle.*

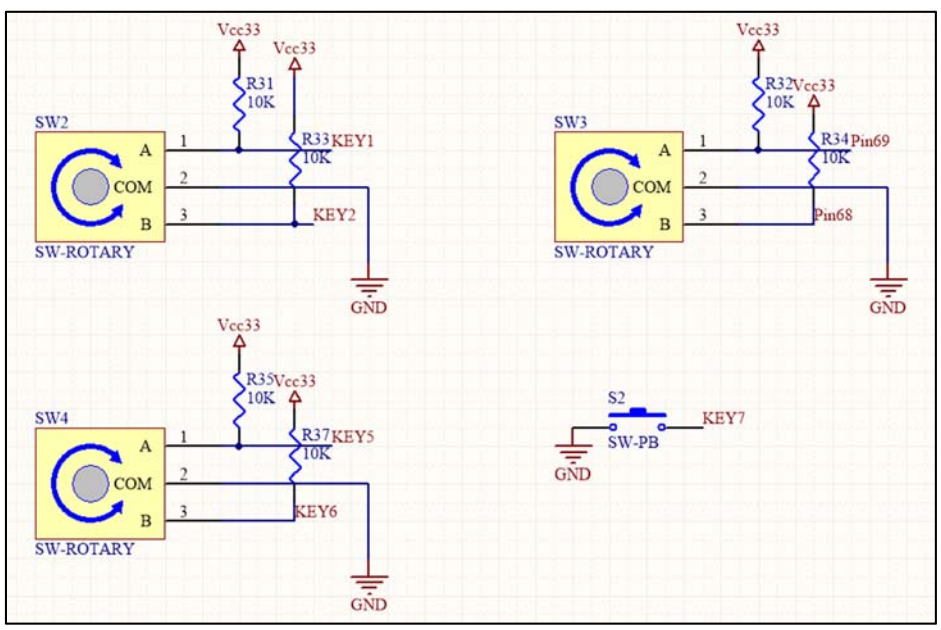
Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>85	132.89
T_{AA} (address access time)	<85	-
T_{C01} (chip selection to output valid)	<85	-
T_{HZ1} (chip deselection to output in high-Z)	<30	-
T_{C02} (chip enable to output high Z)	<85	Irrelevant (CS2 tied high)
T_{LZ2} (chip selection to output in low-Z)	>10	Irrelevant (CS2 tied high)
T_{HZ2} (chip deselection to output in high-Z)	<30	Irrelevant (CS2 tied high)
T_{OE} (output enable to output valid)	<45	-
T_{OHZ} (output disable to output in high-Z)	<30	-
T_{OLZ} (output enable to output in low-Z)	>5	-
T_{OH} (output hold from address change)	>10	-

Keypad

The keypad block for the oscilloscope consists of SW2, SW3, and SW4 (three mechanical quadrature encoders, Bourns Inc. part number 3315P-001-006L), and S2 (press button switch). The segment of memory devoted to keypad software control is as follows

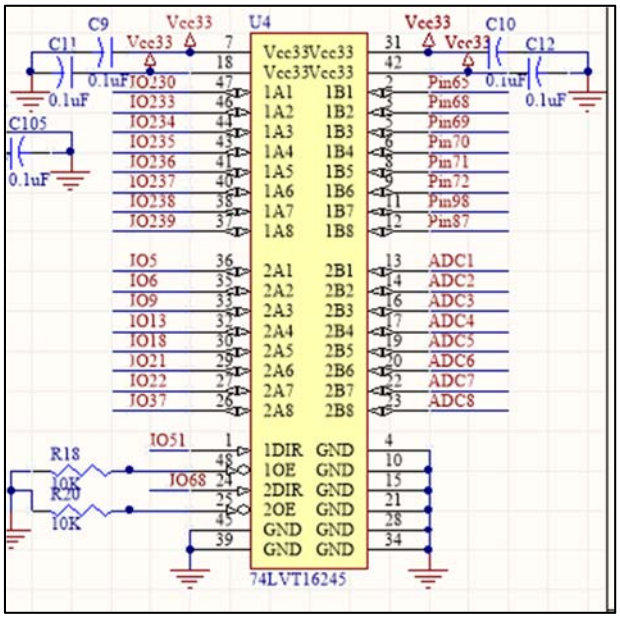
Component	Memory Address (Start – End)
ROM	0x0008000 – 0x000FFFF
SRAM	0x00120000-0x0013FFFF
Trigger Enable	0x00181170-0x0018117F
Sample Rate	0x00181160-0x0018116F
Auto Trigger	0x00181150-0x0018115F
Trigger Delay	0x00181140-0x0018114F
Trigger Slope	0x00181130-0x0018113F
Trigger Level	0x00181120-0x0018112F
FIFO Request	0x00181110-0x0018111F
On-chip memory	0x00170000-0x00179FFF
Keypress	0x00181100-0x001810FF
VRAM	0x00150000-0x0015FFFF
FIFO Data	0x001810F0-0x001810FF
FIFO Full	0x001810E0-0x001810EF

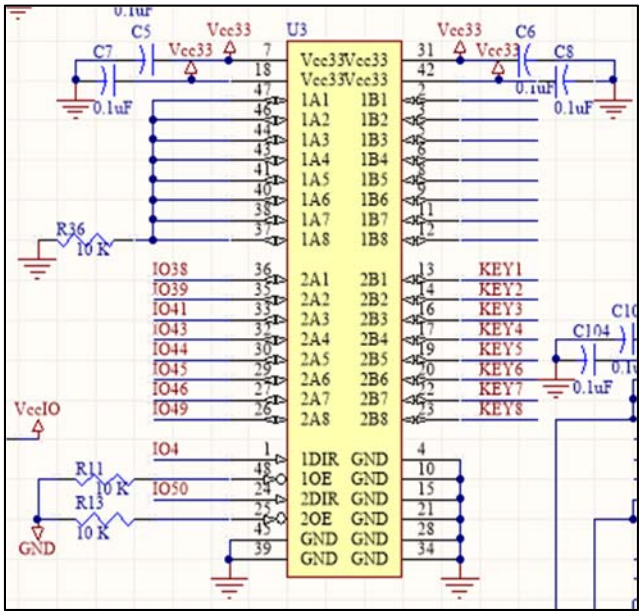
An external debouncer is not included in the design for S2 since debouncing is done in software. The following is the schematic design for the three encoders and single press button switch. Although SW2 is present in the hardware, it does not serve any actual functionality for the oscilloscope



Schematic for Components SW2, SW3, SW4, and S2: *Rotary encoder and push button schematics.*

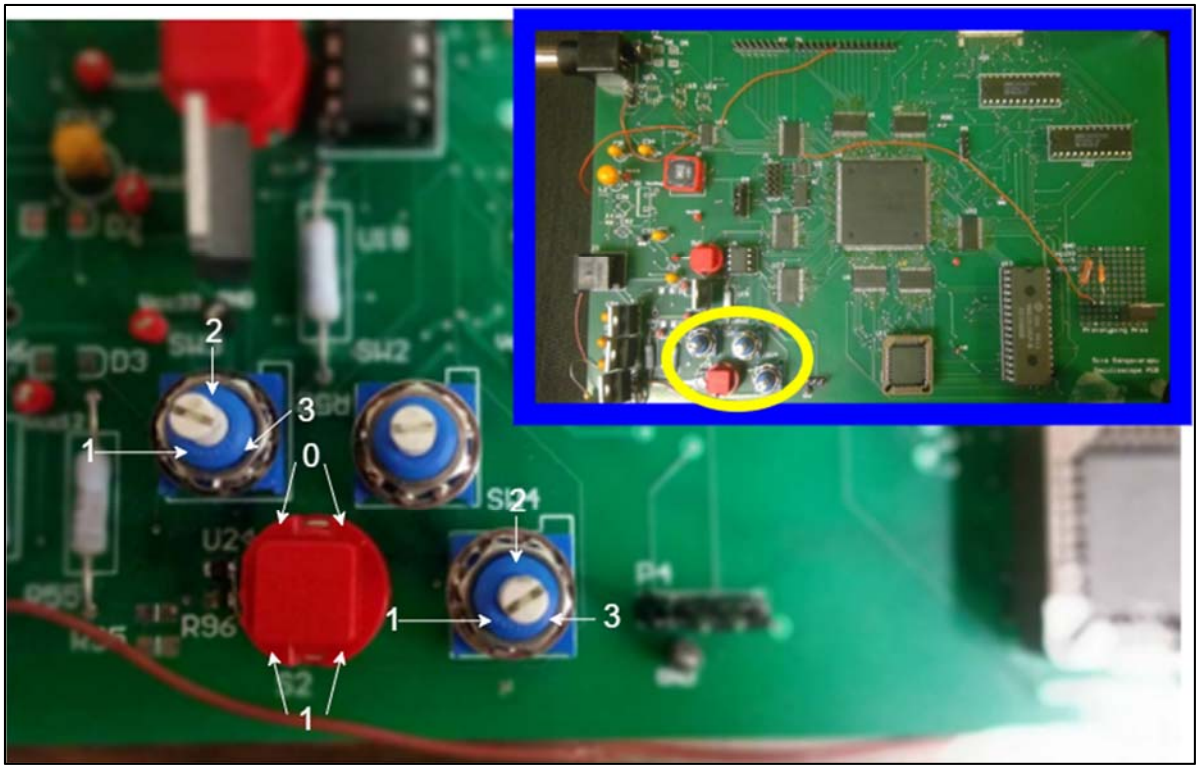
As with all signals going into the FPGA, all keypress signals are passed through buffers U3 and U4, the schematics for which are displayed below.





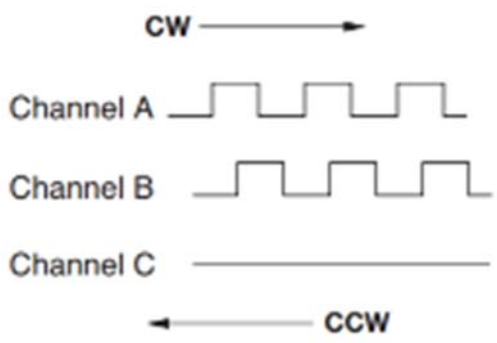
Buffer Control for Keypad Block. *The location for the destination of these signals on U5 can be found in the diagram* ***I/O and Power Pin Configuration Schematic for U5.*** *See the* ***Buffer Direction Control*** *section of the software technical manual to see the direction pin configurations for these buffers.*

PCB location for SW2, SW3, SW4, and S2 are depicted below.



PCB Location and Pin Numbers for Components SW2, SW3, SW4, and S2: *The marked pin numbers on the diagram correspond to the schematic pin numbers in* ***Schematic for Components SW2, SW3, SW4, and S2***

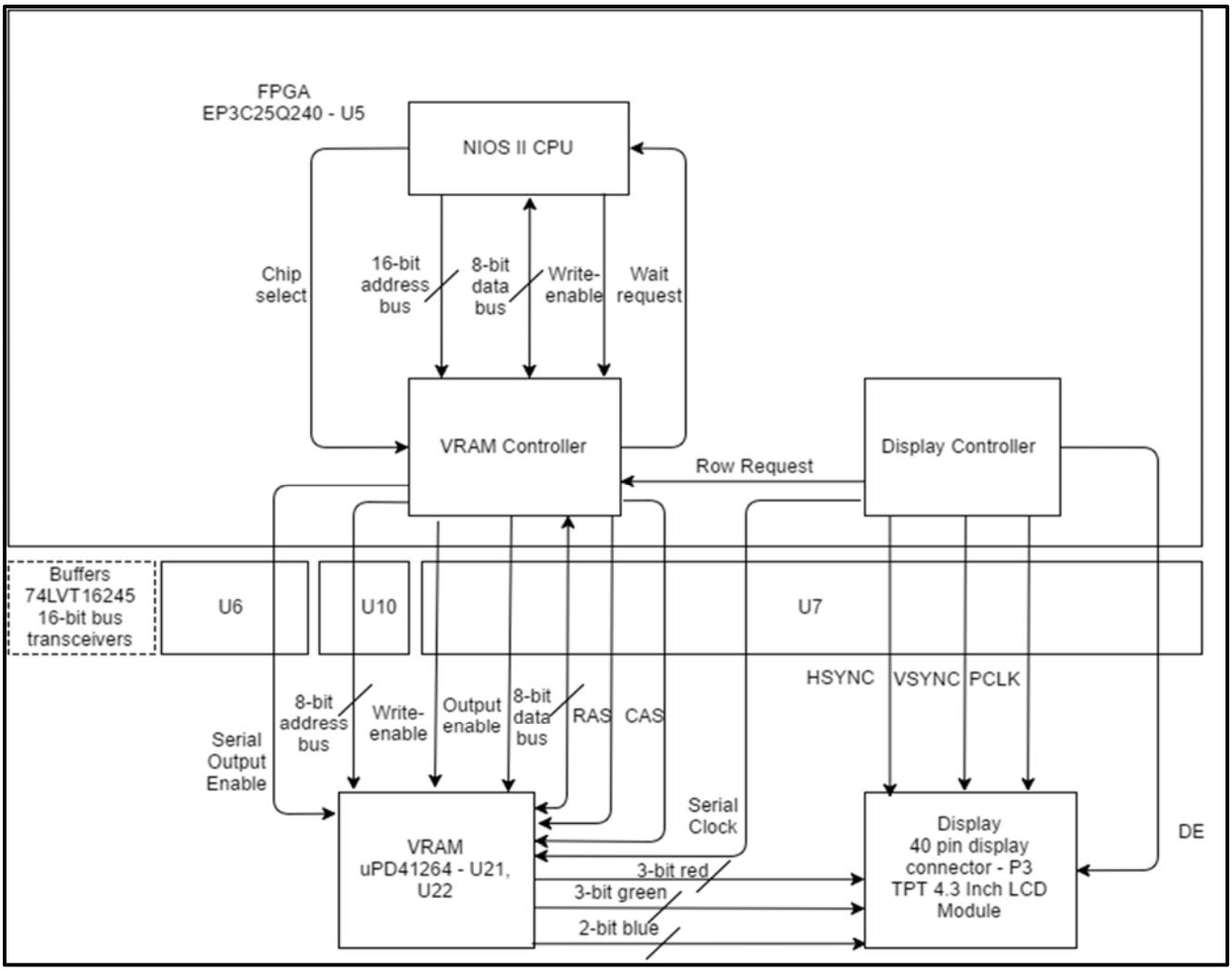
Encoders SW2, SW3, and SW4 have their COM pin tied to ground and their A and B pins tied to 3.3 voltage in order to produce the following quadrature output table (taken from the 3315 series datasheet). Since the encoders are not incremental, one successful turn is defined as a turn of roughly 60 degrees in either direction.



These signals are buffered and inputted into the FPGA through IO pins, where they are processed by internal CPU debouncing and decoding logic **(**see **Keypad** section in software technical manual**)**

VRAM, VRAM Controller, Display Controller, and External Display

The operation and functionality of the oscilloscope's VRAM, VRAM Controller, display controller, and LCD display block can be understood through the following signal block diagram.



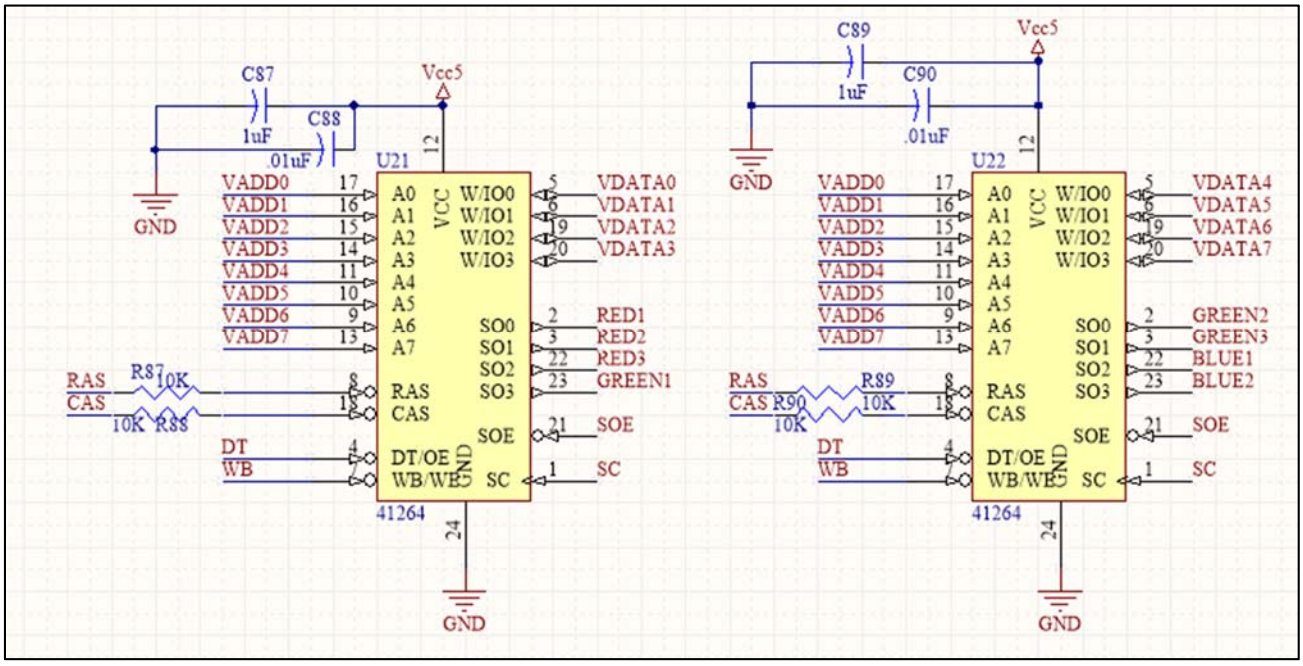
As demonstrated the block diagram, the primary function of the VRAM block is to temporarily store display data so that it can be read by the external LCD and displayed. The VRAM controller and display controller house logic to provide the VRAM and display blocks with control signals in order to do so while adhering to the components' signal timing requirements. The segment in system memory devoted to the operation of this block is as follows.

Component	Memory Address (Start – End)
ROM	0x0008000 – 0x000FFFF

SRAM	0x00120000-0x0013FFFF
Trigger Enable	0x00181170-0x0018117F
Sample Rate	0x00181160-0x0018116F
Auto Trigger	0x00181150-0x0018115F
Trigger Delay	0x00181140-0x0018114F
Trigger Slope	0x00181130-0x0018113F
Trigger Level	0x00181120-0x0018112F
FIFO Request	0x00181110-0x0018111F
On-chip memory	0x00170000-0x00179FFF
Keypress	0x00181100-0x001810FF
VRAM	0x00150000-0x0015FFFF
FIFO Data	0x001810F0-0x001810FF
FIFO Full	0x001810E0-0x001810EF

VRAM BLOCK

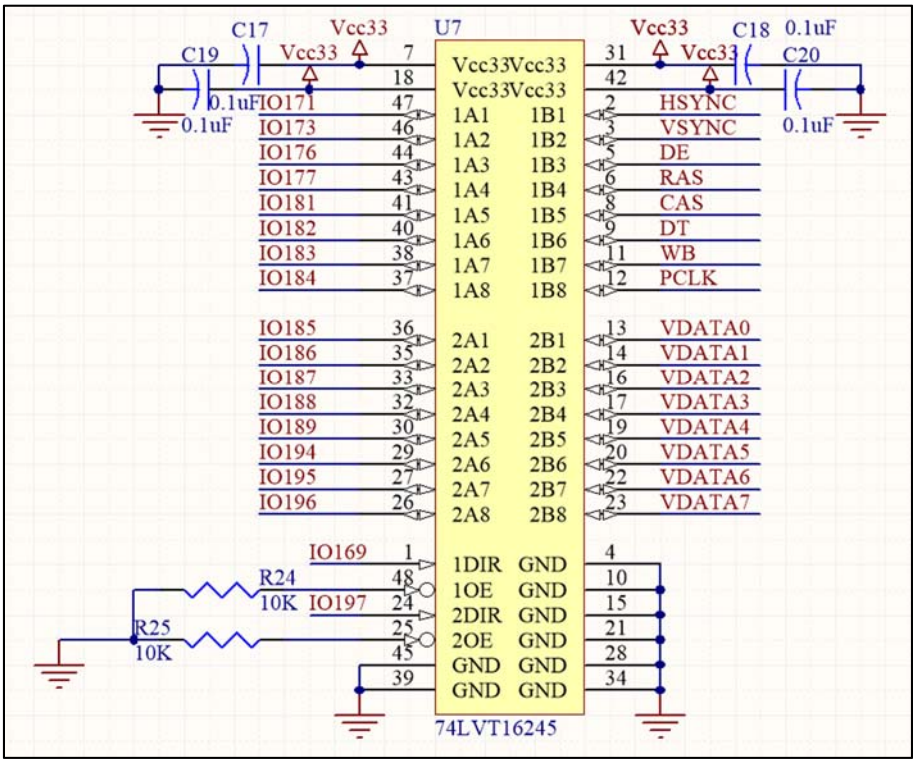
Components U21 and U22, two identical uPD41264 DRAMs, share an 8-bit data bus and 8-bit address bus, as well as write-enable, output enable, RAS, CAS, serial output enable (SOE), and serial clock (SC) signals. In addition, the two chips are powered by a 5 V power source, and each of them utilize .01 uF (C90 and C88) and 1 uF (C89 and C87) bypass capacitors. The following schematic depicts these components and their signals, and the following diagrams show the schematic and PCB design for U10 and U7, the buffers for these signals.



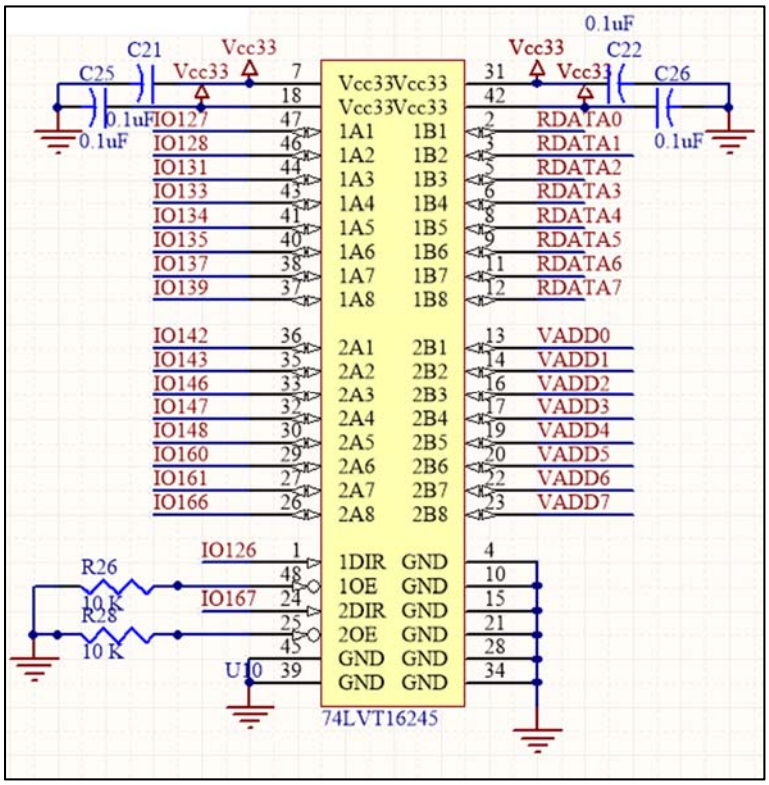
Schematic for U21 and U22: *Depicts all signals entering U21 and U22, as well as the connections for bypass capacitors C87, C88, C89, and C90*

The following diagram illustrates the location of components U21 and U22 in the PCB, as well as the pin numbers corresponding to the schematic.

PCB Location and Pin Numbers for Components U21 and U22: *The marked pin numbers on the diagram correspond to the schematic pin numbers in* ***Schematic for U21 and U22****.*



Schematic for U7*: Buffer for VRAM control signals as well as the bidirectional data bus****.*** *See the* ***Buffer Direction Control*** *section of the software technical manual to see the direction pin configurations for these buffers.*



Schematic for U10*. Buffer for unidirectional VRAM address bus (as well as ROM/SRAM data bus). See the* ***Buffer Direction Control*** *section of the software technical manual to see the direction pin configurations for these buffers.*

PCB locations and pin number information for U10 and U7 can be found in the ***Color Coded PCB Layout*** and ***Pin Numbers for Buffer Components*** diagrams.

SIGNAL TIMING AND FUNCTION FOR U21 AND U22

U21 and U22 serve to buffer graphics data sent out from the CPU intended for display by the external LCD. As part of their dual-port functionality, each contain two types of data ports – a random access port (W/IO0-3) and a serial read port (SO0-3). The random access port uses RAS and CAS signaling in order to execute read, write, and data transfer cycles. Both ports can operate asynchronously, allowing the display data to be read from the serial port while the CPU is communicating with the random access port for read and write operations. The data transfer cycle serves to transfer graphics data in each storage cell at a particular row address to the corresponding location on an internal 1024-bit data register, the contents of which are outputted by the 4 channels of the serial port (total of 8 bits from each U21 and U22) as part of the serial read cycle. During the data transfer cycle, the two ports cannot operate asynchronously.

For proper addressing, U21 and U22 rely on the VRAM controller in order to properly latch 8 bits of the 16-bit address bus into the VRAM controller onto the chip using the RAS signal, and then 8 bits of the 16-bit bus using the CAS signal. In order to do this, the VRAM controller also employs a ROW/COL signal to indicate whether the address bus input to the VRAM controller must be used to decipher a row or column. Although this is an intermediate signal in the system's logic, and will be further documented in the **VRAM Controller** section of the software technical manual it is still shown in the subsequent signal timing diagrams for the appropriate cycles.

U21 and U22 utilize a 256-row by 1024-column matrix as a storage array. During the data transfer cycle, one of the 256 possible rows are selected using 8 row address bits, after which 8 column address bits are used to select one of 256 possible serial decoders corresponding to the location of the next serial read cycle and transfer the data to the 1024-bit data register. The 4 data bits decoded by each of these 256 decoders are then transferred from the data register to be outputted through the 4 serial data buses (SO0-3).

Read Cycle

The ***Timing Diagram for U21 and U22 Read Cycle*** diagram on page 39 shows the timing for the signals generated by the VRAM controller that act upon components U21 and U22 in order to execute a read cycle. Since the read cycle entails a random access cycle, signal OE on the DT/OE (pin 4) is held high as RAS falls. The cycle is executed by bringing RAS, CAS, and OE low while WE is held high (now shown). The output on pins 5, 6, 19, and 20 (IO0, IO1, IO2, and IO3) appears as valid data after device access time, t_{ACC}, has elapsed, which is given by t_{RAC} (150 ns), access time from RAS becoming active. The output data then remains valid until CAS and OE return high, at which point it returns to high impedance.

Early Write Cycle

The ***Timing Diagram for U21 and U22 Early Write Cycle*** diagram on page 31 depicts the signal timing required to execute an early write cycle on components U21 and U22. To execute this cycle, signal WE on the (WB/)WE port is brought low before CAS. Since the write cycle involves the random access part, it sees RAS fall while maintaining DT(/OE) high. Output data on ports IO0-3 remains in high impedance during the entire cycle (not shown in diagram). DT/OE is held high during the entire cycle in order to meet high DT setup and hold times (not shown). Based on the diagram, input data becomes valid between the second and third clock cycles. All other setup and hold times can be seen in the timing diagram.

CAS Before RAS Refresh Cycle

Internal refreshing in U21 and U22 allows chip circuitry to automatically refresh the row addresses specified by the internal address counter and increment it before the next refresh cycle. This cycle is executed maintaining CAS low while bringing RAS low. The ***Timing for CAS Before RAS Refresh cycle*** diagram on page 33 depicts the timing for this cycle in U21 and U22.

Data Transfer Cycle

The data transfer cycle is executed by holding DT(/OE) low when RAS is brought low. For this cycle, the address input specifies which of the 256 possible rows' 1024-bits of data is being transferred to the internal data register. When the cycle's hold and setup times are met, as shown in the timing diagram, graphics data for the specified row is transferred to the data register. In addition, at least one serial clock cycle is required to hold the data in the register so it is not thrown out during the subsequent transfer cycle, as shown in the timing diagram. Moreover, the IO0-3 ports remain on high impedance, since U21 and U22 cease asynchronous port behavior during the data transfer cycle.

In addition, the timing diagram for the data transfer cycle includes the ROW_T/LC signal. This signal's purpose is to act as a "latch clear" for the row transfer request input signal into the VRAM controller (for a more detailed explanation, please see **VRAM Controller** section of software technical manual).

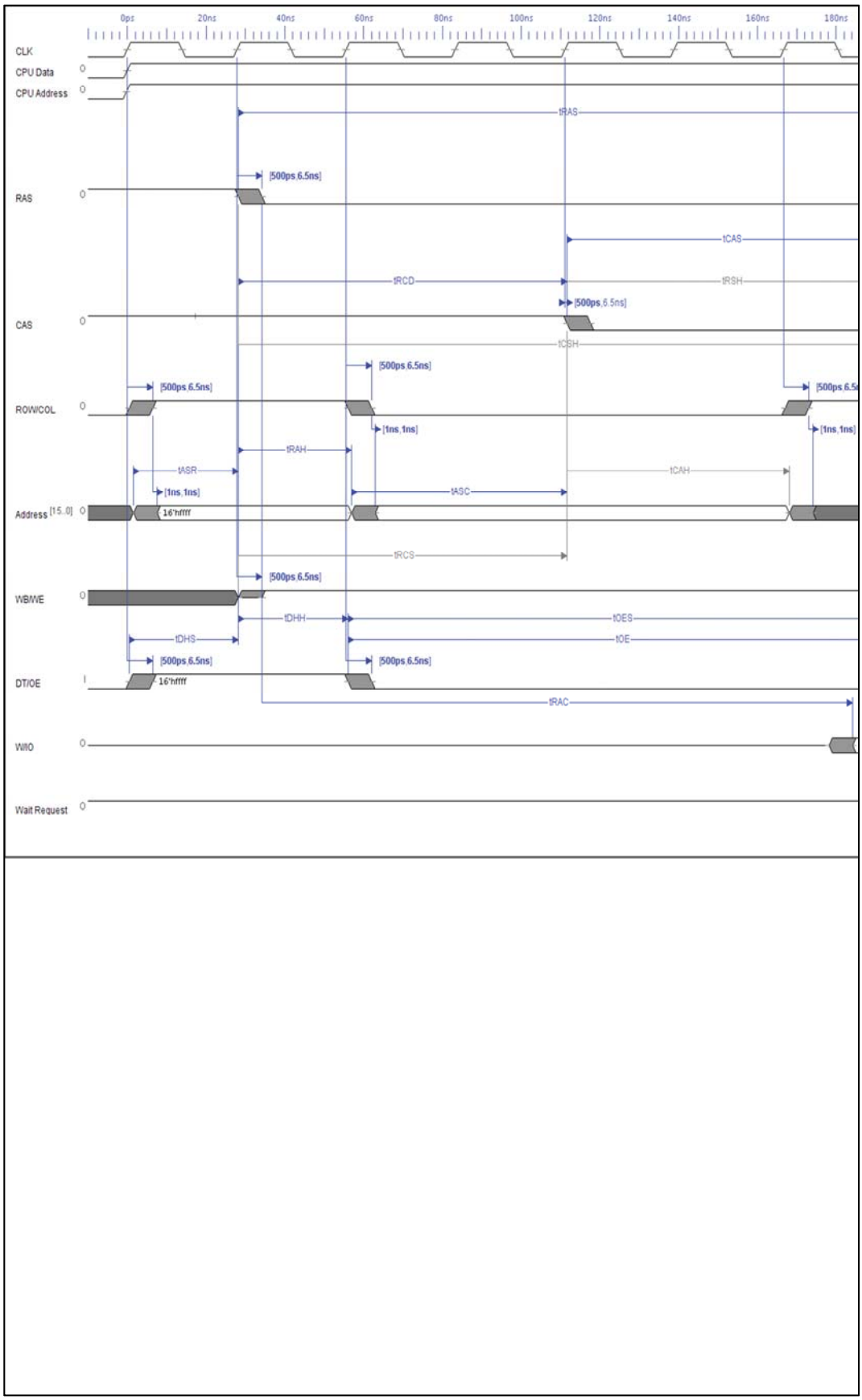
Lastly, the data transfer timing diagram illustrates the cycle's operation while the serial port is inactive. The behavior of an active serial port mimics the behavior shown in ***Timing Diagram for Serial Port Read Cycle*** on page 33.

Serial Read Cycle

Timing for the serial read port is independent and operates asynchronously (except during data transfer cycles) relative to operations performed using the random access port. The serial read cycle occurs after a data transfer cycle in order to output the graphics data stored in the data register corresponding to the row address used in the data transfer cycle.

Signaling for the SC port (pin 1) of U21 and U22 is provided by the display controller, as opposed to the VRAM controller. The **Display Controller** section of the software technical manual will explain the generation of the SC signal.

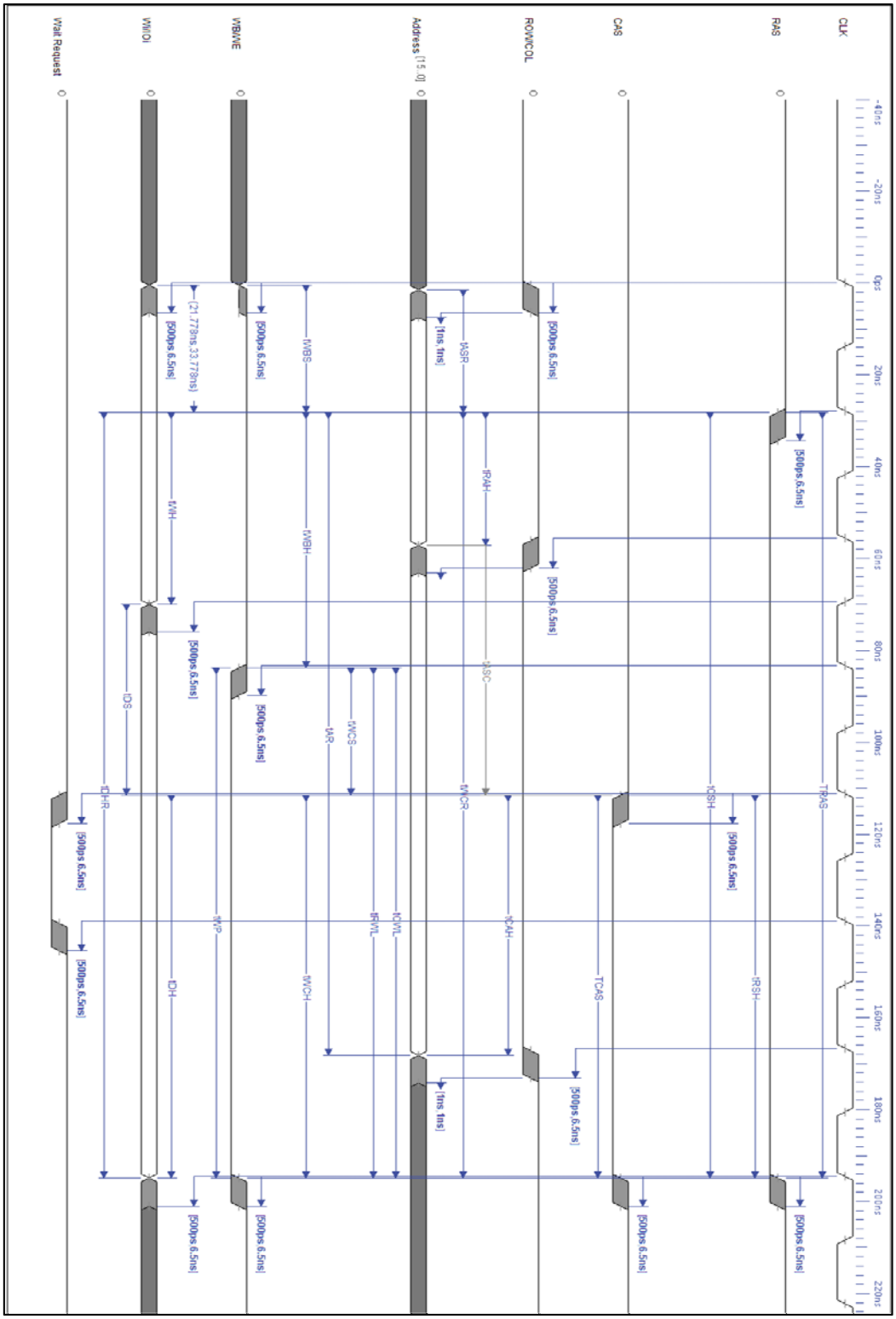
Output data on pins SO0-SO3 (pins 2, 3, 22, 23) appears after an access time of t_{SCA} (max 60 ns) after SC goes high, and remains valid until the subsequent SC cycle. Since the serial output enable (SOE) is active low, SOE is simply always driven low.



Timing Diagram for U21 and U22 Read Cycle*: Parameters given in this diagram correspond to the values given in the table below*

Timing Parameters for U21 and U22 Read Cycle

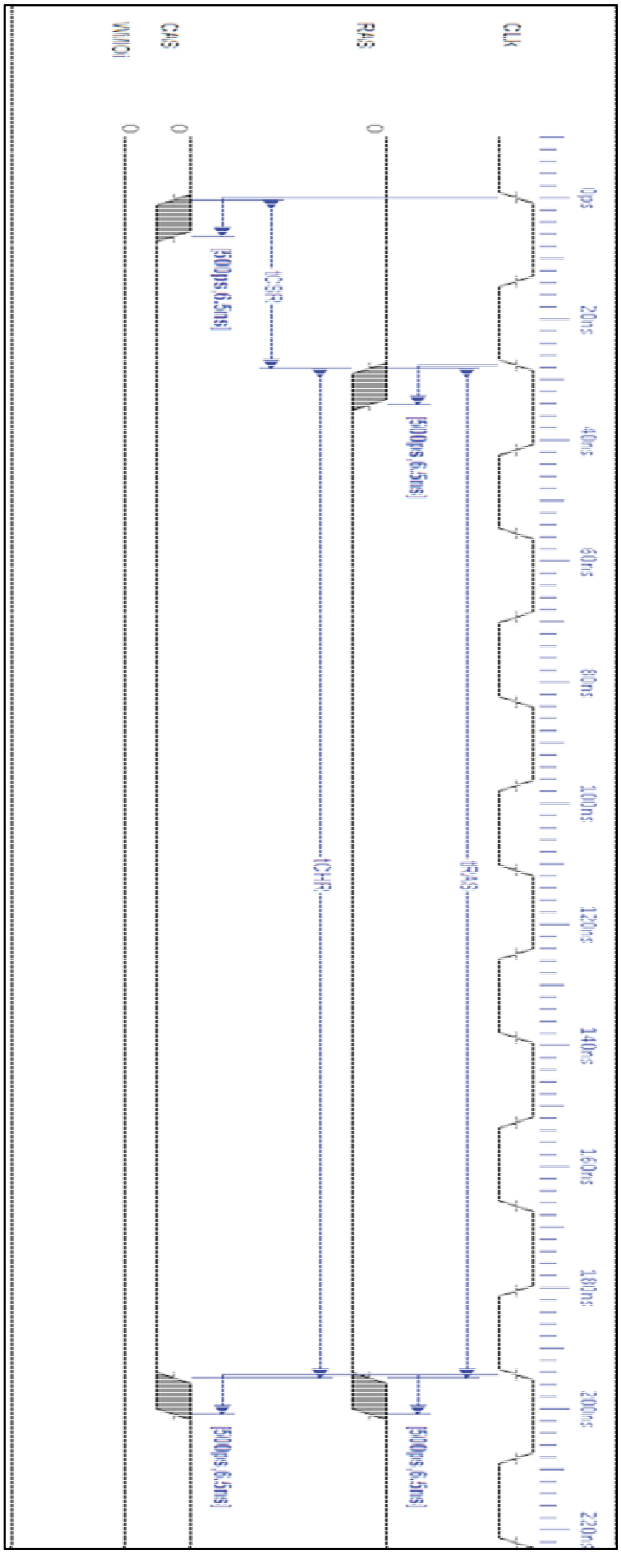
Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>270	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{RAS}	>150, <10000	160.668
T_{RP}	>100	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{CRP}	>10	Met (since CAS is held high going into the cycle)
T_{RCD}	>30, <75	77.334 (since tRCD is greater than the max, access time is controlled exclusively by tCAC
T_{RSH}	>75	77.334
T_{CAS}	>75, <10000	77.334
T_{CPN}	>30	Met (since CAS is held high going into the cycle where it is brought low)
T_{CSH}	>150	160.668
T_{ASR}	>0	20.778
T_{RAH}	>20	22.778
T_{ASC}	>0	48.556
T_{CAH}	>25	50.556
T_{RCH}	>0	49.556
T_{RCS}	>0	77.334
T_{RRH}	>20	49.556
T_{OES}	>10	132.89
T_{AR}	>100	133.89
T_{DHS}	>0	21.778
T_{OEA}	40 max	-
T_{DHH}	>0	21.778
T_{CAC}	75 max	-
T_{OE}	>40	132.89
T_{OEZ}	40 max	-
T_{RAC}	150 max	-
T_{OFF}	40 max	-



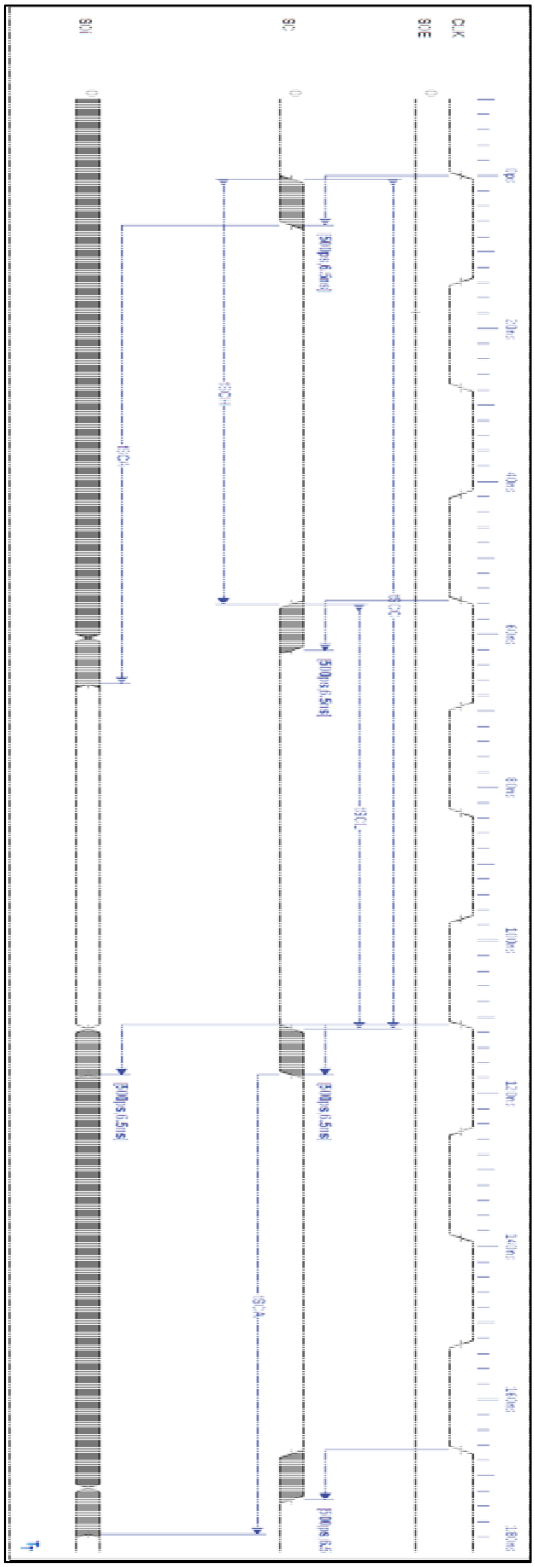
Timing Diagram for U21 and U22 Early Write Cycle

Timing Parameters for U21 and U22 Early Write Cycle

Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>270	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{RAS}	>150, <10000	160.668
T_{RP}	>100	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{CRP}	>10	Met (since CAS is held high going into the cycle)
T_{RCD}	>30, <75	77.334 (since tRCD is greater than the max, access time is controlled exclusively by tCAC
T_{RSH}	>75	77.334
T_{CAS}	>75, <10000	77.334
T_{CPN}	>30	Met (since CAS is held high going into the cycle where it is brought low)
T_{CSH}	>150	160.668
T_{ASR}	>0	20.778
T_{RAH}	>20	22.778
T_{ASC}	>0	48.556
T_{CAH}	>25	50.556
T_{WCS}	>0	21.778
T_{WP}	>0	105.112
T_{WBS}	>0	Met (since WE is held high going into the cycle)
T_{WBH}	>25	49.556
T_{AR}	>100	133.89
T_{CWL}	>45	105.112
T_{DHS}	>0	-
T_{WCR}	>120	160.668
T_{WS}	>0	-
T_{DHH}	>25	-
T_{RWL}	>45	105.112
T_{WCH}	>45	77.334
T_{WH}	>25	35.667
T_{DS}	>0	35.667
T_{DH}	>45	77.334
T_{DHR}	>120	160.668



Timing for CAS Before RAS Refresh cycle



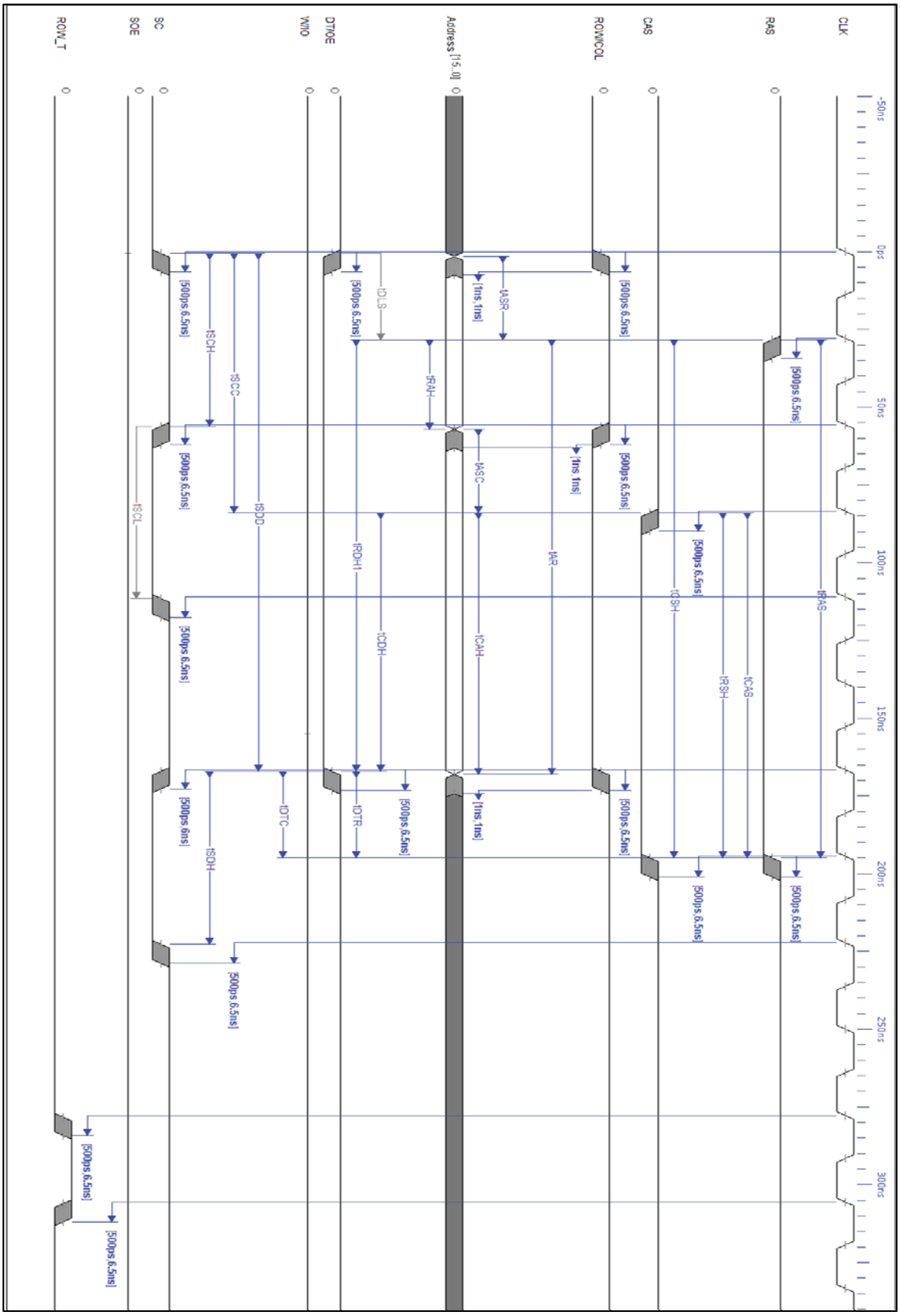
Timing Diagram for Serial Port Read Cycle

Timing Parameters for CAS Before RAS Refresh Cycle

Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>270	Met (since RAS is not brought low again after tRAS until the next cycle
T_{RAS}	>150, <10000	160.668
T_{RP}	>100	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{CSR}	>10	21.778
T_{CHR}	>30	160.668
T_{RP}	>100	Met (since RAS is not brought low again after tRAS until the next cycle
T_{RPC}	>0	Met (since CAS is not brought low again after tRAS until the next cycle

Timing Parameters for Serial Port Read Cycle

Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{SOE}	>20	Met since SOE is tied active (low)
T_{SOP}	>20	Met since SOE is tied active (low)
T_{SOZ}	Max 40	Not shown since SOE tied active (low)
T_{SCC}	>60, < 50000	105.112
T_{SCL}	>20	49.556
T_{SCH}	>20	49.556
T_{SOO}	>5	Met since SOE is tied active (low)
T_{SOA}	Max 50	Not shown since SOE tied active (low)
T_{SCA}	Max 60	-



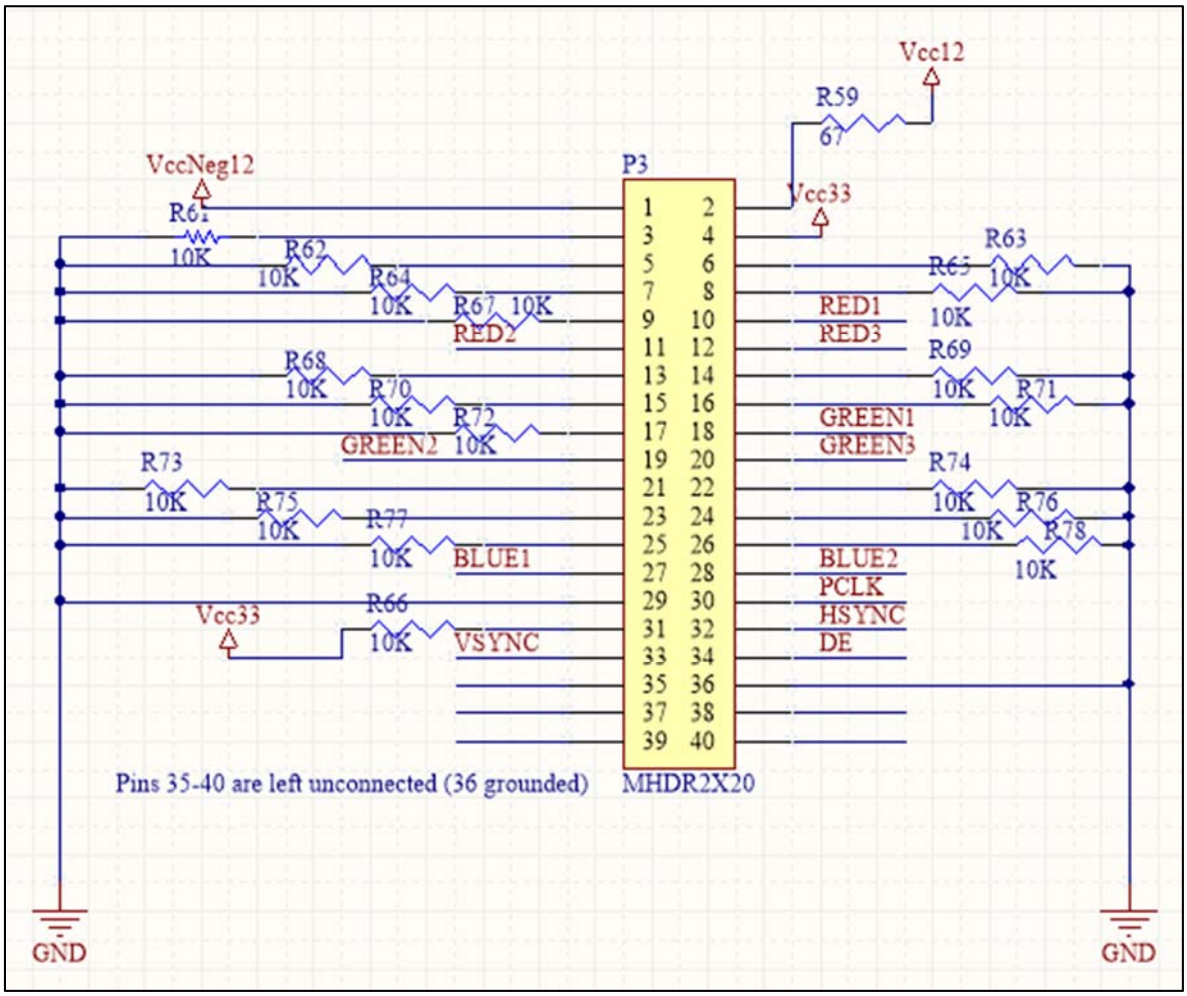
Timing Diagram for Data Transfer Cycle

Timing Parameters for Data Transfer Cycle

Timing Parameter	Required (ns)	Actual (minimum value ns)
T_{RC} (read cycle time)	>270	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{RAS}	>150, <10000	160.668
T_{RP}	>100	Met (since RAS is not brought low again after tRAS until the nex cycle
T_{CRP}	>10	Met (since CAS is held high going into the cycle)
T_{RCD}	>30, <75	77.334 (since tRCD is greater than the max, access time is controlled exclusively by tCAC
T_{RSH}	>75	105.112
T_{CAS}	>75, <10000	105.112
T_{CPN}	>30	Met (since CAS is held high going into the cycle where it is brought low)
T_{CSH}	>150	160.668
T_{AR}		133.89
T_{ASR}	>0	20.778
T_{RAH}	>20	22.778
T_{ASC}	>0	20.778
T_{CAH}	>25	78.334
T_{RDH1}	>130	132.89
T_{DTR}	>10	21.778
T_{DLS}	>0	21.778
T_{CDH}	>55	77.334
T_{DTC}	>10	21.778
T_{SCC}	>60, <50000	77.334
T_{SDD}		160.668
T_{SDH}	>20	49.556
T_{SOZ}	40 max	Not shown since SOE is always low
T_{SCL}	>20	49.556
T_{SCH}	>20	49.556

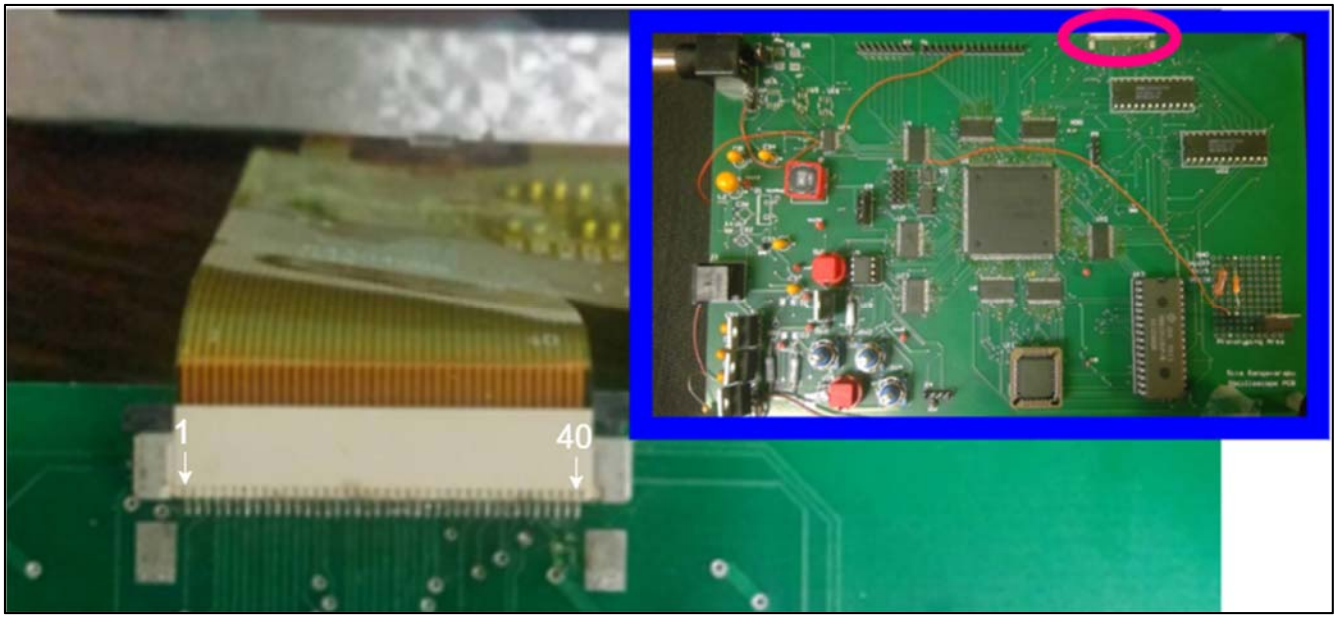
DISPLAY BLOCK

The display block for the system consists of the TFT 4.3 Inch LCD external display module and component P3, a 40-pin 0.50 mm ZIF connector that connects the external display to the rest of the board. The LCD (part number ER-TFT043-3) comes with an onboard OTA5180A System-On-Chip driver. The following is the schematic for P3.



Schematic for Component P3

The location on the PCB as well as the corresponding pin connections for P3 are the subject of the following diagram, ***PCB Location and Pin Numbers for Component P3***



PCB Location and Pin Numbers for Component P3: *The pin numbers for P3 correspond to the numbers shown in the schematic in the diagram* ***Schematic for Component P3.***

The display requires a backlight forward voltage (VLED) of 24 V, which is provided by a 12 V (pin 2) and -12 V (pin 1) power line. In order to provide an appropriate amount of backlight current for the display's LEDs, a current-limiting resistor (R59) is included. In addition, the display requires a power supply voltage (VCC) of 3.3 V (pin 4).

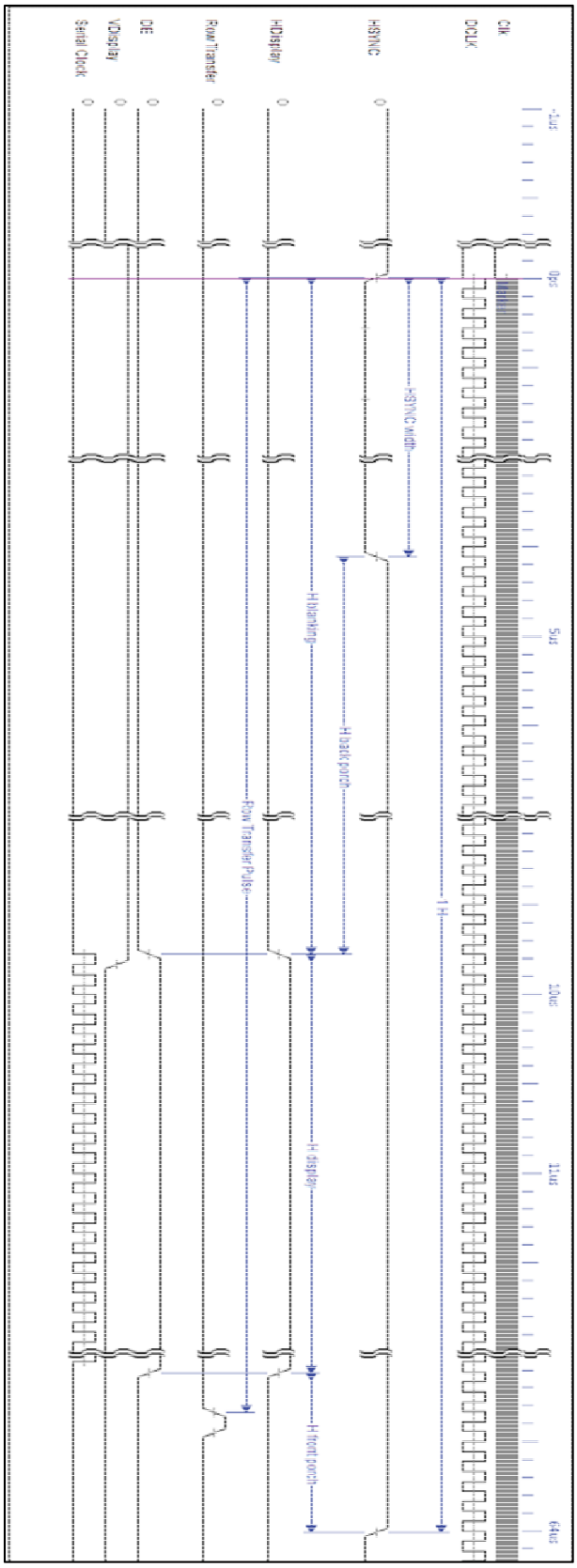
The pins numbered 5-12 correspond to pins R0-R7 (red data). Since only 3 bits of red are outputted from the VRAM block, specifically component U21, pins 5-9 on P3 are grounded using 10 kΩ resistors (R62, R63, R64, R65, R67). The three bits of red data are then connected to the upper 3 data lines (R5-R7) of the red data bus (pin numbers 10-12). Similarly, the 3 bits of green data (one from U21, two from U22), are connected to the upper 3 data lines (pins 18-20) of the G0-G7 data bus (pins 13-20). The unused pins are also pulled low by grounding using 10 kΩ resistors. Finally, two blue data lines originating from component U22 comprise the two couple bits (B6-B7) of the B0-B7 data bus and are connected to pins 27-28 on P3, and the unused blue bus lines are pulled low to ground using 10 kΩ resistors.

Pin number 31 corresponds to the DISP pin, which must be pulled high to VCC (3.3 V) using a 10 kΩ resistor in order to turn on the display. This allows the display to automatically be turned when the power supply is connected to the oscilloscope.

Pins 30, 32, 33, and 34 correspond to the PCLK (display clock), HSYNC (horizontal sync input), VSYNC (vertical sync input), and DE (data enable) control signals received by component P3 after being buffered and outputted by VRAM controller logic in U5. The schematic and PCB diagram for the buffer component for these signals, U7, can be found on page 26.

Lastly, pins 3, 29, and 36 are grounded using a 10 kΩ resistor and pins 35 and 37-40 are left unconnected.

SIGNAL TIMING FOR DISPLAY BLOCK – show link to software part



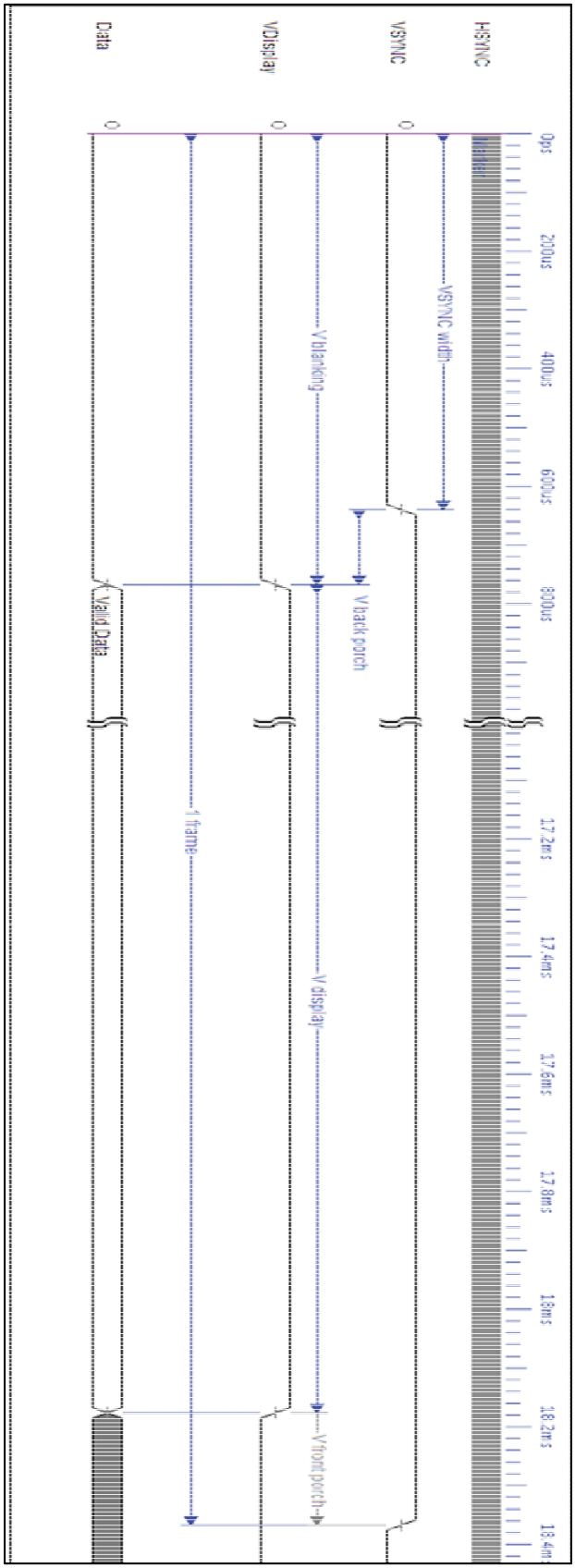
Since the oscilloscope uses the color functionality provided by the LCD, HSYNC and VSYNC must be included and adhere to timing requirements. The timing diagram and timing parameter chart for HSYNC, Row Transfer, and DE signals is shown below.

Display Timing –HSYNC, Row Transfer, Hdisplay, Serial Clock, and DE. This timing diagram is valid only when VDisplay is high (see diagram ***Display Timing-VSYNC*** on next page). When VDisplay is not high, DE and Row Transfer cycles do not go high, and serial clock does not pulse, at any point during the cycle; the rest of the timing diagrams is the same. The following chart shows the timing parameter values for the diagram.

Parameter Name	Length in DCLK cycles
HSYNC Width	41
H blanking	88
H back porch	47
Row Transfer Pulse	570
H display	480
H front porch	8
1 H	525

Display Timing Parameters – HSYNC, Row Transfer, Serial Clock, and DE.

The DCLK (shown as PCLK in ***Schematic for Component P3)*** signal has a frequency of 9 MHz, with once cycle equivalent to four system clock cycles. In addition, the DE signal is equivalent to the Hdisplay in this diagram (where VDisplay is also assumed to be high). The subsequent diagram, ***Display Timing – VSYNC and VDisplay***, and its corresponding parameter chart, shows the timing information for the vertical display signaling.



Display Timing –VSYNC and VDisplay

The following table gives the values for the timing parameters for this diagram, in the units of HSYNC cycles (see previous page).

Parameter Name	Length in HSYNC cycles
VSYNC Width	10
V blanking	12
V back porch	2
V display	272
V front porch	3
1 frame	287

This diagram also shows that data is read by the display when VDisplay is asserted. However, it should be noted that data is only read when VDisplay is asserted and HDisplay is asserted, along with DE, as shown in the diagram on the previous page. When this is the case, each byte of RGB data is clocked in (parallel on each of the three serial data lines) on each DCLK cycle by the display driver. This timing for data to be read by the display driver is communicated to components U21 and U22 as the serial clock signal, shown in the ***Display Timing –HSYNC, Row Transfer, Hdisplay, Serial Clock, and DE*** *diagram.*

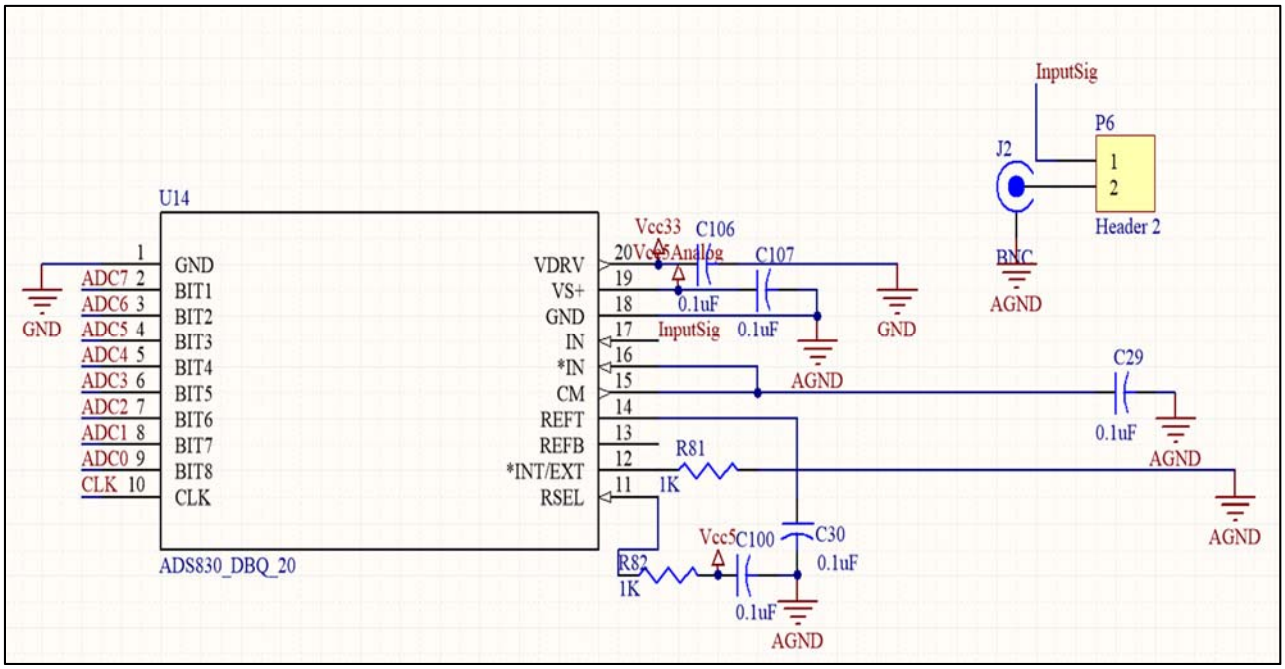
The generation for these signals –HSYNC, VSYNC, VDisplay, HDisplay, DE, Row Transfer, DCLK, and Serial Clock –is a result of component U5's display controller. The logic that composes this block is the subject of the **Display Controller and Display Software** section in the software technical manual.

DISCREPANCY BETWEEN VRAM SIZE AND DISPLAY SIZE

Components U21 and U22 each have a row size of 256 and column size of 256. The LCD follows a display format of 480 x 272 (width by height). However, since the counter for the data transfer cycle addressing is set to repeat after reaching 256 (see ***VRAM Controller Signaling Diagram)*** rows, the display repeats itself with the same information stored in U21 and U22. This gives rise to the oscilloscope's trademark Dual-ViewOscope functionality.

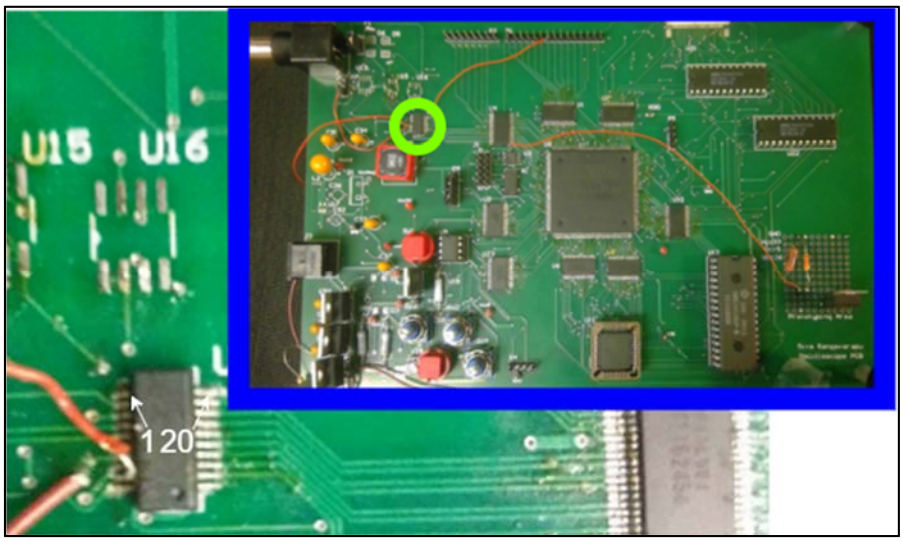
Analog

The analog block for the oscilloscope is responsible for receiving an external analog signal and converting it to a digital data bus that is buffered and inputted into U5. The block consists of components J2, a female connector, and U14, an ADS830 analog-to-digital converter. The schematic and PCB locations for these components is shown in the figure ***Schematic for U14 and J2***



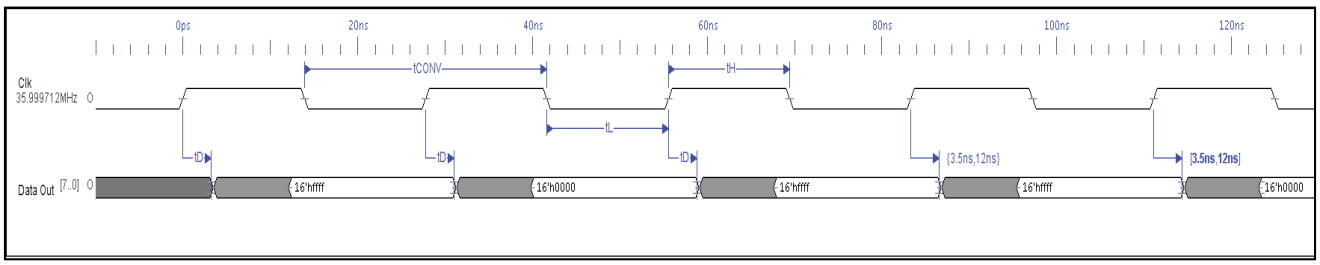
Schematic for U14 and J2

The following diagram illustrates the location of U14 in the PCB, as well as the pin numbers corresponding to the schematic.



PCB Location and Pin Numbers for Component U14: *Marked pin numbers on the diagram correspond to the pin numbers in* U14 schematic figure shown above.

When the user supplies the oscilloscope with an input waveform by connecting J2 with a male BNC cable carrying an analog signal, U14 converts the waveform to digital data that is carried on the ADC data bus (ADC0-7). The input clock for U14 is the same system clock for U5 that is provided by X1 (frequency of 36 MHz). This timing for this analog-to-digital conversion follows the diagram shown below.

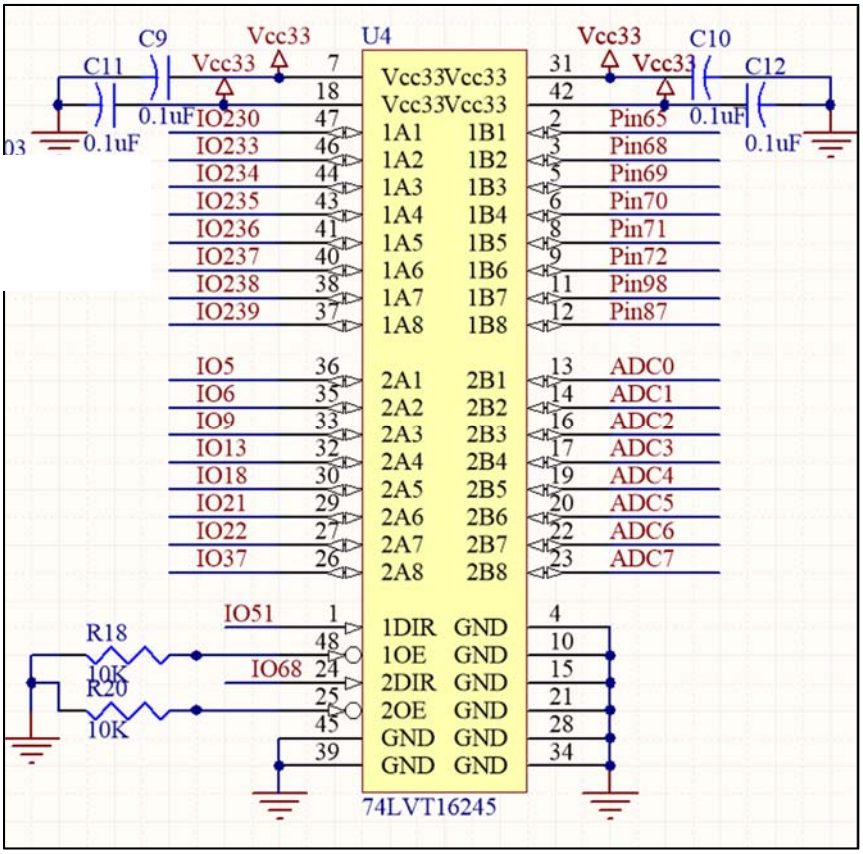


Analog to Digital Conversion Timing Diagram*: The diagram only depicts the full conversion of 4 bits of data; 8 clock cycles are required to obtain all 8 bits of data for the bus.*

Timing Parameter	Requirement or delay (ns)	Value (ns)
T_D	3	3
T_{CONV}	>16.6, <100us	**27.778**
T_L	**>7.3**	**13.889**
T_H	**>7.3**	**13.889**

Analog to Digital Conversion Timing Parameters: *Timing parameters for the* ***Analog to Digital Conversion Timing Diagram*** with the requirements and actual values.

The ADC0-7 data bus is then buffered through component U4, the schematic for which is below.



***ADC0-7 data bus buffer schematic**: Refer back to the **I/O and Power Pin Configuration Schematic for U5** diagram to see where the buffered ADC buffer data is inputted into U5 and the location of the I/O pins seen above. See the **Buffer Direction Control** section of the software technical manual to see the direction pin configurations for these buffers.*

Finally, the area in memory devoted to the operation of this block is as follows.

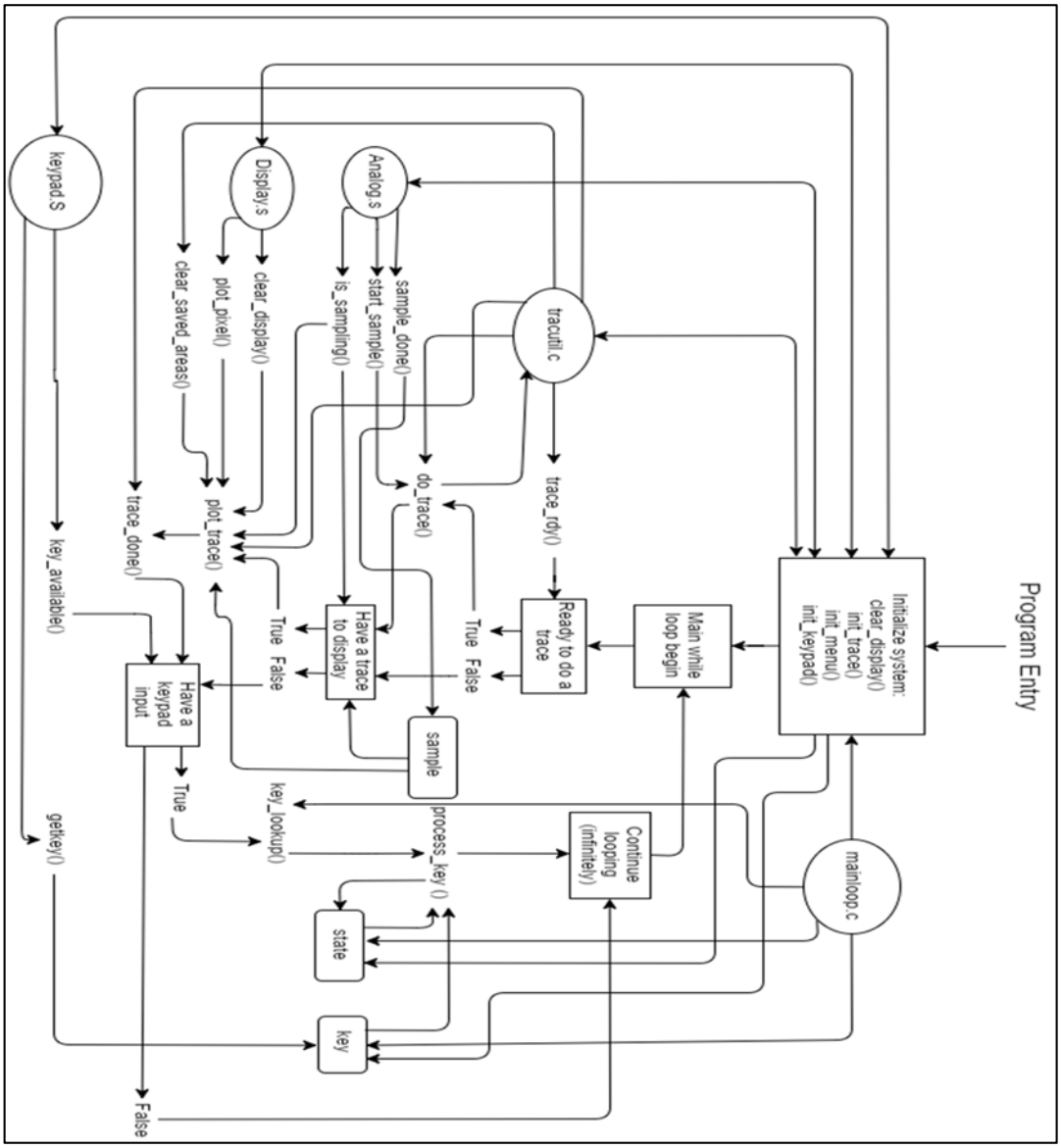
Component	**Memory Address (Start – End)**
ROM	0x0008000 – 0x000FFFF
SRAM	0x00120000-0x0013FFFF
Trigger Enable	0x00181170-0x0018117F
Sample Rate	0x00181160-0x0018116F
Auto Trigger	0x00181150-0x0018115F
Trigger Delay	0x00181140-0x0018114F
Trigger Slope	0x00181130-0x0018113F
Trigger Level	0x00181120-0x0018112F
FIFO Request	0x00181110-0x0018111F
On-chip memory	0x00170000-0x00179FFF
Keypress	0x00181100-0x001810FF
VRAM	0x00150000-0x0015FFFF
FIFO Data	0x001810F0-0x001810FF
FIFO Full	0x001810E0-0x001810EF

After buffering analog data converted by U14 is then received by the analog logic block, which is programmed logic in the FPGA. The design of this logic is the subject of the **Analog Logic and Software** section in the subsequent software technical manual.

SOFTWARE

General System Overview

The software technical manual documents two key facets of the oscilloscope system: the design for the logic programmed onto the FPGA, as well as the code required to run the system software. Programmed logic is stored in the serial configurational device and loaded serial onto the FPGA upon power-on (see **Serial Configuration Device** section in hardware manual), and the system software is stored in ROM to be executed upon power-on. The following flowchart provides a general overview of the operation of the system software.



System software flowchart*: Simple overview of the program flow and main loop required to run the system software.*

In addition, the programmed logic includes a design and implementation of an Altera© Nios II soft processor, which is responsible for the execution of the program code. This logic is also responsible for the proper functioning and interactions of the logic block components, including buffer direction control, keypress decoding logic, VRAM controller, display controller, and analog logic.

As shown in ***System Software Flowchart,*** the system software centers around the primary execution loop, which is found in the main() function in the file "mainloop.c." Prior to entering the infinite main loop, the program initializes the display, menu, keypad, and traces by making the appropriate function calls. Code for the main() function follows below.

```
/*
   main

   Description:      This procedure is the main program loop for the Digital
                     Oscilloscope.  It loops getting keys from the keypad,
                     processing those keys as is    appropriate. It also
                  initializes the display and routines to handle keypress
                  events. It also handles
                     starting scope sample collection and updating the LCD
                     screen.
   Arguments:        None.
   Return Value:     (int) - return code, always 0 (never returns).

   Input:            Keys from the keypad.
   Output:           Traces and menus to the display.

   Error Handling:   Invalid input is ignored.

   Algorithms:       The function is table-driven.  The processing routines
                     for each input are given in tables which are selected
                     based on the context (state) the program is operating
in.
   Data Structures:  Array (process_key) to associate keys with actions
             (functions to call).

   Global Variables: None.

   Author:           Glen George
   Last Modified:    8/14/2016

*/

int  main()
{
    /* variables */
    enum keycode        key;            /* an input key */

    enum status         state = MENU_ON;    /* current program state */

    unsigned char far  *sample;         /* a captured trace */

    /* key processing functions (one for each system state type and key) */
```

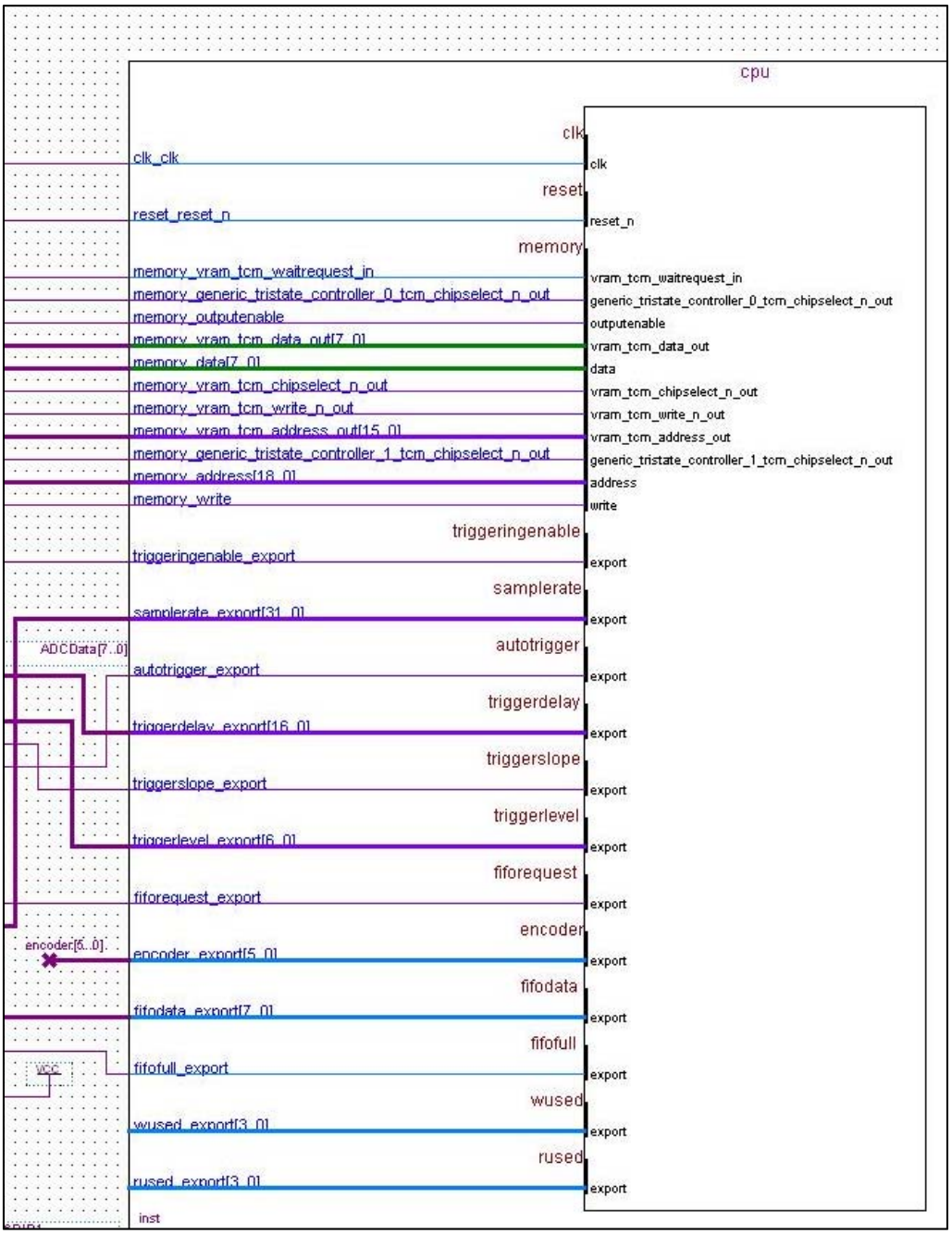
```
    static enum status  (* const process_key[NUM_KEYCODES][NUM_STATES])(enum
status) =
       /*   Current System State                          */
       /*  MENU_ON      MENU_OFF             Input Key   */
      { {  menu_key,    menu_key    },   /* <Menu>      */
        {  menu_up,     no_action   },   /* <Up>        */
        {  menu_down,   no_action   },   /* <Down>      */
        {  menu_left,   no_action   },   /* <Left>      */
        {  menu_right,  no_action   },   /* <Right>     */
        {  no_action,   no_action   } }; /* illegal key */



    /* first initialize everything */
    clear_display();        /* clear the display */

    init_trace();       /* initialize the trace routines */
    init_menu();        /* initialize the menu system */
    init_keypad();  /* initialize the keypad routines, including the
interrupts */

    /* infinite loop processing input */
    while(TRUE)  {

        /* check if ready to do a trace */
    if (trace_rdy())
        /* ready for a trace - do it */
        do_trace();


    /* check if have a trace to display */
    if (is_sampling() && ((sample = sample_done()) != NULL))  {

        /* have a trace - output it */
        plot_trace(sample);
        /* done processing this trace */
        trace_done();
    }


    /* now check for keypad input */
    if (key_available())  {

        /* have keypad input - get the key */
        key = key_lookup();

        /* execute processing routine for that key */
        state = process_key[key][state](state);
    }
    }


    /* done with main (never should get here), return 0 */
    return  0;
}
```

In addition, the code excerpts presented in the following sections of the software technical manual, particularly those for keypad, display, and analog, make extensive references to constants found in the “interface.h” file, which facilitates integration between the code that will be presented, which is mostly written in assembly, and the main software loop shown above. The contents of “interface.h” file are found below.

```
/*************************************************************************
/*                                INTERFAC.H
/*                            Interface Definitions
/*                                Include File
/*                        Digital Oscilloscope Project
/*                                 EE/CS 52
/*************************************************************************
/
/*
   This file contains the constants for interfacing between the C code and
   the assembly code/hardware for the Digital Oscilloscope project.

   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Added constant MAX_SAMPLE_SIZE and removed
                            KEY_UNUSED.
        6/13/16    Siva Gangavarapu  Revised for project

*/
#ifndef  __INTERFAC_H__
    #define  __INTERFAC_H__

/* library include files */
  /* none */
/* local include files */
  /* none */
/* constants */
/* keypad constants */
#define  KEY_MENU       5   /* <Menu>      */
#define  KEY_UP         1   /* <Up>        */
#define  KEY_DOWN       2   /* <Down>      */
#define  KEY_LEFT       4   /* <Left>      */
#define  KEY_RIGHT      3   /* <Right>     */
#define  KEY_ILLEGAL    0   /* illegal key */
/* display constants */
#define  SIZE_X         256         /* size in the x dimension */
#define  SIZE_Y         256              /* size in the y dimension */
#define  PIXEL_WHITE      0xE0     /* pixel off */
#define  PIXEL_BLACK      0xFF     /* pixel on */
/* scope parameters */
#define  MIN_DELAY     0    /* minimum trigger delay */
#define  MAX_DELAY     50000    /* maximum trigger delay */
#define  MIN_LEVEL         0    /* minimum trigger level (in mV) */
#define  MAX_LEVEL      5000    /* maximum trigger level (in mV) */
/* sampling parameters */
#define  MAX_SAMPLE_SIZE   1024 /* maximum size of a sample (in samples) */

#endif
```

Altera © Nios II Soft Processor Design

The CPU for the Dual-ViewOscope® is a Nios II soft processor implemented in the FPGA. The following diagram shows the design and connections for the Nios II in the system's main block diagram.



***Main Block – CPU**: Input and output ports for the system soft processor*

Subsequent sections in the software technical manual will detail the various connections seen above. The following table provides an outline for the ports, respective components, and target blocks (which will the subsequent section titles for this portion of the technical manual.

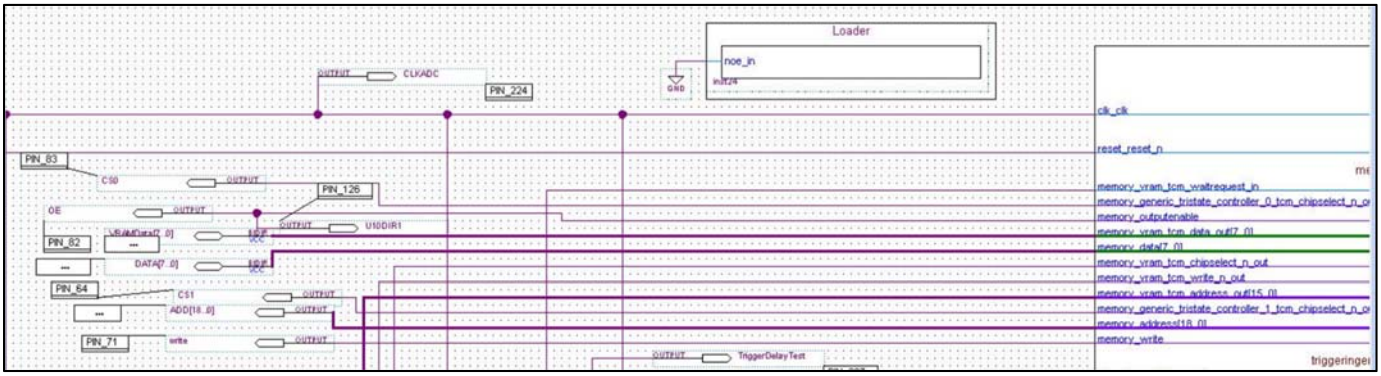
CPU Port	**CPU Component Name/ PIO (Parallel I/O) Name**	**Block**
clk	Clock Input	System
reset	Reset Input	System
vram_tcm_waitrequest_in	VRAM wait request	VRAM Controller
generic_tristate_controller_0_tcm_chipselect_n_out	ROM chip select	ROM/SRAM
outputenable	ROM/SRAM output enable	ROM/SRAM
vram_tcm_data_out	VRAM data bus	VRAM Controller
data	ROM/SRAM data bus	ROM/SRAM
Vram_tcm_chipselect_n_out	VRAM chip select	VRAM Controller
Vram_tcm_write_n_out	VRAM write enable	VRAM Controller
Vram_tcm_address_out	VRAM address bus	VRAM Controller
generic_tristate_controller_1_tcm_chipselect_n_out	ROM chip select	ROM/SRAM
address	ROM/SRAM address bus	ROM/SRAM
write	ROM/SRAM write enable	ROM/SRAM
triggeringenable	PIO_0	Analog Logic
samplerate	PIO_1	Analog Logic
autotrigger	PIO_2	Analog Logic
triggerdelay	PIO_3	Analog Logic
triggerslope	PIO_4	Analog Logic
triggerlevel	PIO_5	Analog Logic
fiforequest	PIO_6	Analog Logic
encoder	PIO_7	Keypad Logic
fifodata	PIO_8	Analog Logic
fifofull	PIO_9	Analog Logic

CPU Port, Component, and Target Block Table

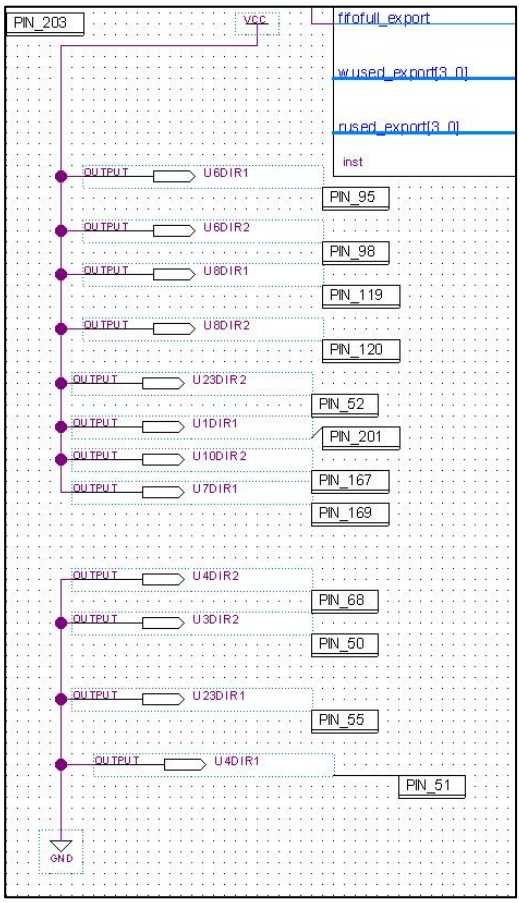
Note: the comments in the code shown in subsequent sections of the software portion of this technical manual refers to system address values by the PIO values very frequently.

Clock, Reset, ROM/SRAM

The connections to the CPU for the clock, reset, and ROM/SRAM components are quite straightforward and can be understood by examining the following figure: ***Main Block –Clock, Reset, ROM/SRAM***



Main Block –Clock, Reset, ROM/SRAM: *Connections for the Clock, Reset, and ROM/SRAM ports in the CPU block.*

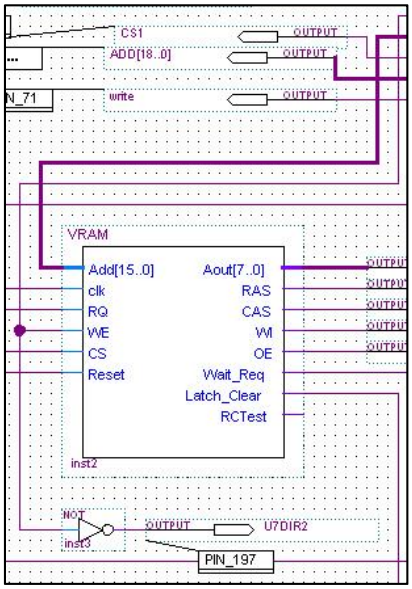


Basic Buffer Direction Control.

Buffer Software

Since all buffer banks' direction pins are controlled by FPGA I/O pins, programmed logic is used to determine if the buffers operate in the A→B or B→A directions (output or input). The majority of the buffer banks operate in only one direction at all times, as shown in the diagram ***Basic Buffer Direction Control***. Direction pins tied high to VCC operate in the A→B direction (as outputs), while pins tied low to GND operate in the B→direction (as inputs). In addition, ***Pin List Table*** on page 89 gives a pin list showing each input/output pin in the ***Main Block – Clock, Reset, ROM/SRAM*** and ***Basic Buffer Direction Control*** diagrams and its corresponding I/O pin on the FPGA (component U5).

Two of the direction pins, U10DIR1 and U7DIR2, operate in accordance with another signal that is not always held high or low. The direction control of component U10 bank 1 is controlled by the output enable signal for ROM/SRAM, and direction control of component U7 bank 2 is controlled by the write enable signal that is fed into the VRAM controller. Pin U10DIR1 can be seen in the ***Main Block –Clock, Reset, ROM/SRAM*** above, and the connections for pin U7DIR2 are depicted below. For a full description of the WE signal that goes into the VRAM Controller, refer to the VRAM Controller section in the software technical manual.



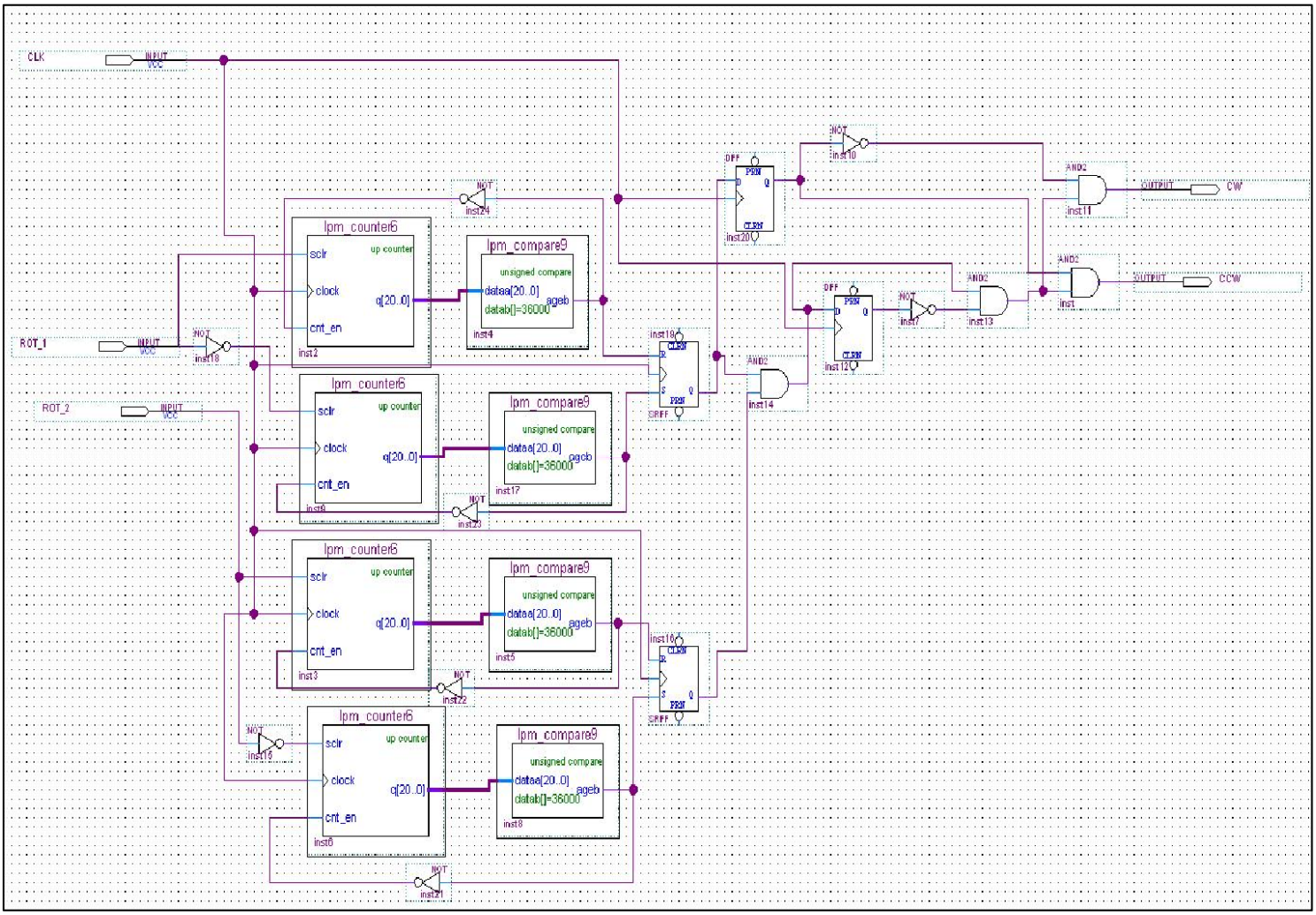
Component U7 bank 2 direction control in relation to VRAM write enable signal. U7DIR2 pin is seen at the bottom of the diagram.

Keypad Logic and Software

Software control of the keypad hardware, including the two rotary encoders and one push button switch, occurs in primarily two phases. See section Buffered keypress signals are inputted into the I/O pins of U5 and converted into interrupt signals by logic design implemented in the FPGA. These interrupts are then handled by the CPU using the functions found in the file "keypad.S." Once the keypresses are appropriately handled, the main loop found in "mainloop.c" updates the core software logic and display appropriately.

Generation of Interrupt Signals from Raw Hardware Input – Rotary Encoders

As seen in the diagram in the **Keypad** section of the hardware technical manual, operation of the rotary encoders generates a waveform that is first debounced then decoded in order to generate different output signals for clockwise and counterclockwise turns. The logic design responsible for doing so is the subject of the following diagram.



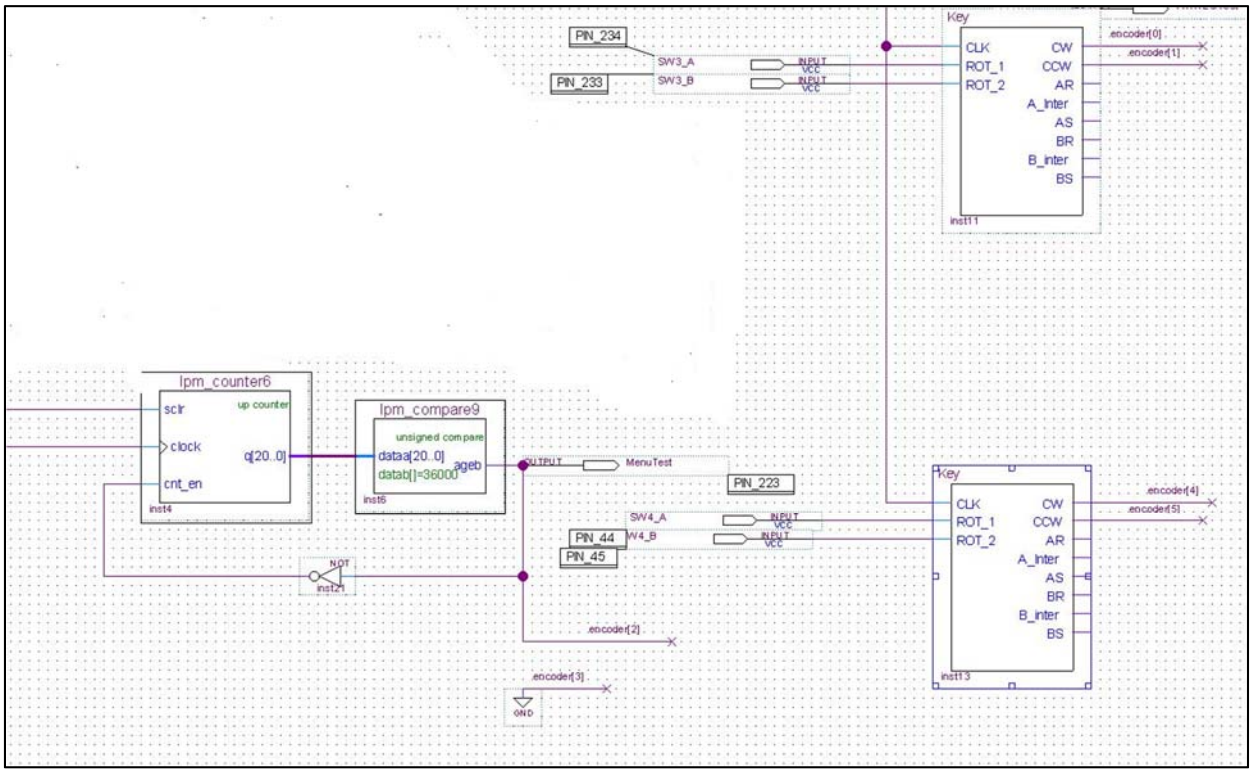
Decoding Rotary Encoder Quadrature Output into Clockwise and Counterclockwise Interrupts: *This logic design is implemented in the FPGA.*

The inputs into this logic block diagram are the system clock signal and the two outputs of each rotary encoder. The signals are first debounced for 36000 clock cycles (roughly 1 ms). The logic design then ensures that a turn in the clockwise direction drives the CW output high for one clock cycle and leaves the CCW output low; the opposite occurs for a counterclockwise turn**.** Thus, two different signals are generated based on the direction of the rotary switch's rotation by the user

Generation of Interrupt Signals from Raw Hardware Input – Push Button Switch

A similar logic block allows for debouncing of the push button switch, shown in the diagram below (***Main Block -Keypad Logic)***

Interrupt Handling

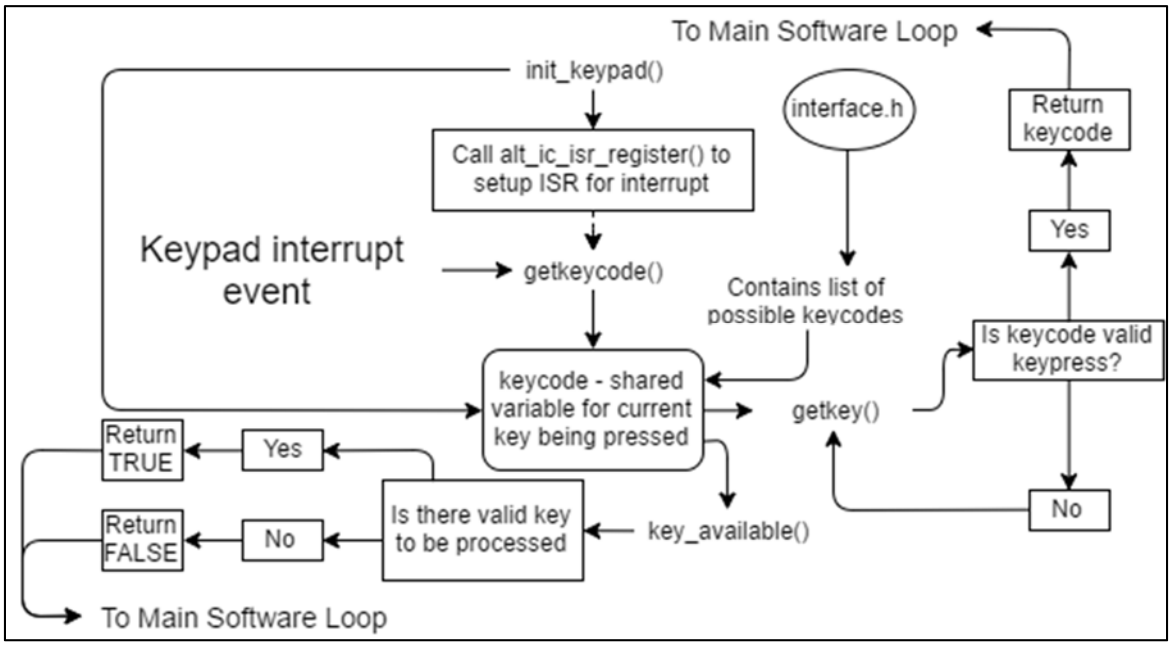


Main Block -Keypad Logic: Diagram showing the connections between the logic to decode each rotary switch/push button switch into interrupts and the rest of the main block, including CPU

Each rotary encoder and its two outputs are decoded by two different logic blocks (inst 13 and inst11). The interrupt outputs from each logic block comprises the encoder[0…5] bus that is received by the CPU block's PIO 7 (see diagram ***Main Block – CPU***).

The encoder PIO is configured in the CPU as edge-type IRQ generating interrupt with a synchronously captured rising-edge type edge capture register. Software found in "keypad.S,"

in addition to initializing the interrupt handler for this PIO to respond to keypress events, is responsible for integrating them into the operation of the rest of the software. A block diagram outlining the keypress software flow is the subject of the following figure.



Keypad Software Flow: *Blcok diagram showing the interactions in the software found in the "keypad.S" file, which is responsible for registering user keypress events and interfacing them with the main software loop.*

The init_keypad() function, which is called at the beginning of the main() loop in "mainloop.c," is responsible for initializing the aforementioned interrupt service routine. The interrupt handler for this ISR, getkeycode(), updates the keycode shared variable with every keypad interrupt event. The key_available() and getkey() functions then interface with the rest of the software (see ***System Software Flowchart***) using the value of the keycode variable. The contents of "keypad.S," including the code for these functions, are shown below.

```
#include "interfac.h"
#include "system.h"
.data
keycode:  .long     0

.text
keyTable:   .word CaseIllegal, CaseUp, CaseDown, CaseFill, CaseFill2,
                  CaseRight, CaseLeft, CaseMenu, Case6, Case7

.global getkey, init_keypad, key_available

# void init_keypad()
# Description:      Function initializes the interrupt handler and shared
#                   variables used by the keypad routines. Function also
#                   registers the ISR for the keypress interrupt signals.
# Operation:        The function first initializes the keycode shared
```

```
#                   variable by setting it to KEY_ILLEGAL. The function
#                   then makes the function call to alt_ic_isr_register()
#                   in order to setup the system to register the ISR for
#                   the keypress input (located at PIO 7). The function
#                   call is made with the following argument name and values:
#                   Argument 1: ic_id - 0
#                   Argument 2: irq - 0
#                   Argument 3: isr - getkeycode
#                   Argument 4: isr_context - 0
#                   Argument 5: flags - 0
#                   The function then clears the value of the edge capture
#                   register for this PIO. Next, it writes 0x3F to the PIOs
#                   interrupt mask register to enable interrupts for all
#                   the necessary ports in the PIO.
# Arguments:        None
# Return Value:     None
# Local Variables:  None
# Shared Variables: keycode-Current key that is being pressed (write only)
# Global Variables: None
# Input:            Keypress input, which is setup as an interrupt, located
#                   at PIO 7
# Output:           None
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: sp, ra, r4, r5, r6, r7, r8, r9, r10
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16

init_keypad:
    subi    sp, sp, 4               #Set up stack pointer in preparation for
                                    #call to alt_ic_isr_register() function
    stwio   ra, 0(sp)               #store stack pointer in return address
                                    #register to return back to this frame
                                    #after function call
    movia   r4, keycode             #store keycode variable location in r4
    movi    r5, KEY_ILLEGAL         #store keycode for KEY_ILLEGAL in r5
    stwio   r5, 0(r4)               #set keycode shared variable to
                                    #KEY_ILLEGAL
    mov     r4, r0                  #setup first argument of
                                    #alt_ic_isr_register() function call to
                                    #0. First argument is interrupt
                                    #controller identifier
    mov     r5, r0                  #Setup second argument for
                                    #alt_ic_isr_register() call to 0. Second
                                    #argument is irq for  hardware interrupt
    movia   r6, getkeycode          #Setup third argument for function call
                                    #Third argument is the address of the
                                    #function to jump to when this interrupt
                                    #is triggered. The function in this case
                                    #is the interrupt handler getkeycode()
    mov     r7, r0                  #Setup fourth argument (isr_context) to
                                    #function to be 0
    mov     r8, r0                  #Setup for fifth argument (flags) to be
                                    #0
    subi    sp, sp, 4               #Setup stack pointer since 5th argument
```

```
                                        #in function call will be first value on
                                        #the stack
    stwio   r8, 0(sp)                   #Store argument value on stack
    call    alt_ic_isr_register         #Make the aactual function call
    addi    sp, sp, 4                   #Restore the original stack state
    movhi   r10, %hi(PIO_7_BASE)        #Get the high bits of the address for
                                        #the interrupt PIO (PIO_7)
    ori     r10, r10, %lo(PIO_7_BASE) #Or in the low bits of the address
    movi    r9, 0x3F                     #Prepare to clear edge capture register
                                         #and enable interrupts by writing to
                                         #the interruptmask register
    stwio   r9, 12(r10)                 #The edge capture register is located at
                                        #offset of 12 bytes from start of PIO_7
                                        #Writing 0x3F to edge capture register
                                        #clears the bits for all six bits in
                                        #edge capture register
    stwio   r9, 8(r10)                  #Writing 0x3F to interruptmask register
                                        #(which is located offset 8 bytes from
                                        #start of PIO_7 base address) enables
                                        #interrupts for all corresponding PIO
                                        #input ports.
    ldwio   ra, 0(sp)                   #restore the function state before
                                        #function call
    addi    sp, sp, 4                   #restore original stack pointer frame
    ret

# void getkeycode()
# Description:      Interrupt handler for keypresses. Upon detecting
interrupt,
#                   the program jumps to this function, which deciphers the
#                   value of the edge capture register and determines which
#                   key has been pressed, updating the value of the keycode
#                   shared variable accordingly
# Operation:        The function first obtains the value of the edge capture
#                   register by finding its offset from PIO 7 base address,
#                   which is the location of the keypress interrupt. The
#                   function then uses the value of the edge capture register
#                   as the index for a jump table, where the value of the
#                   edge capture register, which corresponds to a particular
#                   keypress and keycode, will allow the function to use the
#                   table to jump to the appropriate subroutine to update
#                   the keycode shared variable value with the corresponding
#                   keycode. Once the keycode variable has been updated,
#                   the function ends by clearing the edge capture register.
# Arguments:        None
# Return Value:     None
# Local Variables:  None
# Shared Variables: keycode-Current key that is being pressed (write only)
# Global Variables: None
# Input:            None
# Output:           None
# Error Handling:   None
# Algorithms:       None
# Data Structures:  Keytable (jumptable)
# Registers Changed: r9, r10, r8, r12, r7, r13, r15, r14
# Limitations:      None
# Author:           Siva Gangavarapu
```

```
# Last Modified:    8/8/16
getkeycode:
    movia   r9, keyTable                #store address of beginning of
                                        #keyTable jump table in r9
    movhi   r10, %hi(PIO_7_BASE)        #get the address for the keypress
                                        #interrupt PIO (PIO_7) high bits
    ori     r10, r10, %lo(PIO_7_BASE)   #get the low bits for the complete
                                        #address
    ldwio   r8, 12(r10)                 #store edge capture register in r8
    mov     r12, r8                     #store the edge capture register in
                                        #r12 as well
    movi    r7, 1                       #move while loop counter into r7
                                        #This will also be index for jump
                                        #table

getkeycodeloopstart:
    movi    r13, 1                      #Prepare to increment jump table
    beq     r8, r13, getkeycodeloopend  #Check if edge capture register value
                                        #is one, which means r7 contains the
                                        #correct index for the jump table
    add     r7, r7, r13                 #increment jump table offset
    movi    r10, 1                      #prepare to right shift the edge
                                        #capture register
    sra     r8, r8, r10                 #right shift edge capture register
    jmpi    getkeycodeloopstart         #keep looping until value of edge
                                        #capture register is 0

getkeycodeloopend:                      #The formula for deciphering the
                                        #proper address in the jump table
                                        #to jump to is given by (4 * r7 +
                                        # address_of_start_of_keyTable)
    add     r7, r7, r7                  #r7 (jump table index) = 2 * r7
    add     r7, r7, r7                  #r7 = 4 * r7 (since jump table has
                                        #elements of size words)
    add     r7, r7, r9                  #r7 = r7 (offest) + r9(start of
                                        #keyTable)
    ldw     r7, 0(r7)                   #get the address of the label to
                                        #jump to in r7
    movia   r15, keycode                #store the address of the keycode
                                        #shared variable in r15 so that the
                                        #value maybe be updated later
    jmp     r7                          #jump to the appropriate label
                                        #based on the calculated index/offset
CaseIllegal:                            #if the deciphered edge capture
                                        #register value represented none
                                        #of the predetermined keypress
                                        #values, then jump here
    movi        r14, KEY_ILLEGAL        #store the appropriate keycode in r14
    jmpi        getkeycodeend           #jump to function end
CaseUp:
    movi        r14, KEY_UP             #same thing as CaseIllegal, except
                                        #jump here when the the value
                                        #represented a KEY_UP keypress. Store
                                        #keycode in r14
    jmpi        getkeycodeend           #jump to function end
CaseDown:
    movi        r14, KEY_DOWN           #same as above, except for the
```

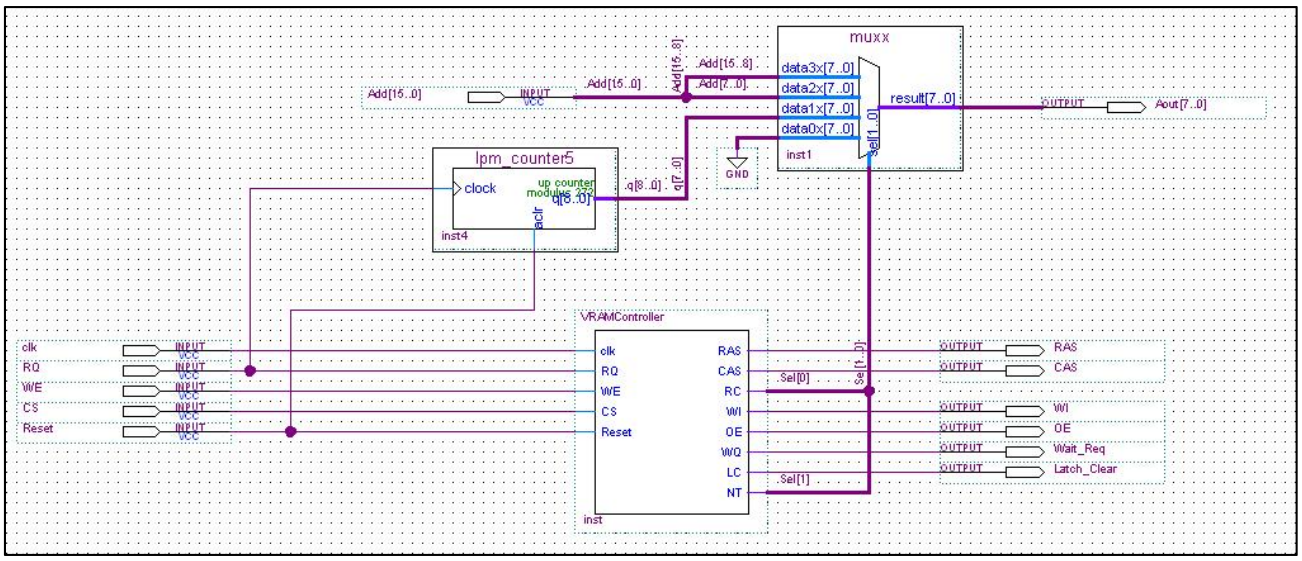
```
                                        #KEY_DOWN keypress
    jmpi        getkeycodeend
CaseFill:
    movi        r14, KEY_MENU           #same as above, except for the
                                        #KEY_MENU keypress
    jmpi        getkeycodeend
CaseFill2:
    movi        r14, KEY_MENU           #same as above, except for the
                                        #KEY_MENU keypress
    jmpi        getkeycodeend
CaseRight:
    movi        r14, KEY_RIGHT          #same as above, except for the
                                        #KEY_RIGHT keypress
    jmpi        getkeycodeend
CaseLeft:
    movi        r14, KEY_LEFT           #same as above, except for the
                                        #KEY_LEFT keypress
    jmpi        getkeycodeend
CaseMenu:
    movi        r14, KEY_MENU           #same as above, except for the
                                        #KEY_MENU keypress
    jmpi        getkeycodeend
Case6:                                  #do nothing for now
    jmpi    getkeycodeend
Case7:
getkeycodeend:
    stwio       r14, 0(r15)             #since the current keypressis in r14,
                                        #update the keycode shared variable
                                        #by writing to its address, which
                                        #is in r15
    movhi       r10, %hi(PIO_7_BASE)    #once again, in order to access
                                        #the edge capture register, we need
                                        #to get the PIO 7 base address,
                                        #starting with the high bits
    ori         r10, r10, %lo(PIO_7_BASE) #finish off the address by getting
                                              #the low bits and putting it
                                              #in r10
    stwio       r12, 12(r10)                  #clear the edge capture register
                                              #since the current keypress has
                                              #been dealt with, and the system
                                              #can be ready for the next one
    ret                                 #return

# unsigned char key_available()
# Description:      If there is a valid key to be processed, then the
#                   function returns TRUE. If not, then the function
#                   returns FALSE.
# Operation:        The function reads the value of the keycode shared
#                   variable, and compares it to KEY_ILLEGAL to see if
#                   there is a valid key. If the value is equal to
#                   KEY_ILLEGAL, then the function stores 0 in r2 for
#                   returning. If the value is not equal to KEY_ILLEGAL
#                   the function stores 1 in r2 for returning.
# Arguments:        None
# Return Value:     unsigned char - a zero or one, depending on if there
#                   is a valid key to be processed (0) or not (1)
# Local Variables:  None
```

```
# Shared Variables: keycode-Current key that is being pressed (ready only)
# Global Variables: None
# Input:            None
# Output:           None
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r10, r11, r2
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
key_available:
    movia       r10, keycode            #get the address where the keycode
                                        #shared variable is stored
    ldwio       r10, 0(r10)             #get the keycode value
    movi        r11, KEY_ILLEGAL        #prepare to check if keycode is equal
                                        #to KEY_ILLEGAL
    beq         r10, r11, keyIllegal    #if keycode equal to KEY_ILLEGAL,
                                        #then jump to keyIllegal label
keyNotIllegal:                          #if the keycode was not equal to
                                        #KEY_ILLEGAL, then there is a valid
                                        #keypress
    movi        r2, 1                   #store 1 in r2 so it can be returned
                                        #since there is valid keypress
    jmpi    key_available_end           #end the function
keyIllegal:
    movi        r2, 0                   #since there is no valid keypress,
                                        #store 0 in r2 so it can be returned
key_available_end:
    ret                                 #return

# int getkey(void)
# Description:      Function returns the keycode for the key currently
#                   being pressed. Function does not return until a valid
#                   keypress event occurs.
# Operation:        The function first reads the value of the keycode
#                   shared variable and checks if it has value KEY_ILLEGAL
#                   If it does, then the function keeps looping until the
#                   value is something different. Once it is, then the
#                   function moves the keycode value into r2 and returns it
# Arguments:        None
# Return Value:     int - keycode for current key being pressed. This value
#                   will never be KEY_ILLEGAL
# Local Variables:  None
# Shared Variables: keycode-Current key that is being pressed (read only)
# Global Variables: None
# Input:            None
# Output:           None
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed:r11, r10, r2
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
getkey:
```

```
    movia       r10, keycode                #get the address where the
keycode
                                            #shared variable is stored
    ldwio       r10, 0(r10)                 #get the keycode value
    movi        r11, KEY_ILLEGAL            #prepare to check if keycode
                                            #value is KEY_ILLEGAL
    beq         r10, r11, getKeyKeepLooping #if it is, then keep looping
    jmpi        returnKey                   #if not, then jump so function
                                            #can return
getKeyKeepLooping:
    jmpi        getkey                      #if the keycode value is KEY_
                                            #ILLEGAL, then keep looping
                                            #by jumping back to function
                                            #beginning
returnKey:
    movia       r10, keycode                #get the address where the
                                            # keycode shared variable is
                                            # stored
    ldwio       r2, 0(r10)                  #store the actualy keycode
                                            #variable in r2 so it can be
                                            #returned
    stwio       r0, 0(r10)                  #clear the keycode to prepare for
                                            #the next key to be processed
                                            #by the software
    ret                                     #return
```

VRAM Controller

The VRAM Controller is responsible for taking signals generated by the CPU and translating them into signals to be read by the VRAM block (components U21 and U22), CPU, and display controller. For reference, ***The VRAM Block Diagram*** in the hardware technical manual shows an outline of the signaling during this process.



VRAM Controller Signaling Diagram*: Generation of signals to be received by display controller and VRAM block in response to input signals.*

VRAM Controller Signaling Diagram illustrates the logic used to take the signals generated by the CPU (write enable-WE, 16-bit address bus-Add[15..0], and chip select-CS), the display controller (row request RQ), and standard system clock (clk) and reset (Reset) signals and generate signals that will be received by the VRAM block components U21 and U22 (an 8-bit address bus-Aout[7..0], RAS, CAS, write enable-WI, output enable-OE), by the CPU (wait request-Wait_Req), and an intermediate signal (latch clear-Latch_Clear)

The core of the logic diagram is the VRAMController block, which receives the input signals described above and generates the output signals RAS, CAS, RC, WI, OE, WQ, LC, and NT. As described in the **VRAM Block** portion of the hardware technical manual**,** the random access port of components U21 and U22 is always operating in four distinct cycles (read, write, refresh, and row transfer). The signaling for these cycles, which follows ***Timing Diagram for U21 and U22 Read/Early Write/Refresh/Data Transfer Cycle*** diagrams found in the hardware technical manual, is controlled by the VRAMController block. Timing for the LC and NT signals is not shown in these diagrams.

In addition to the signals already described, RC and NT are used to generate the proper 8-bit address bus from the 16-bit address bus that is generated by the CPU. The RC signal is held high when the address output represents the row address and low when the output represents the column address. This behavior, as well as the timing for the address signaling, can be found in

Timing Diagram for U21 and U22 Read/Early Write/Data Transfer Cycle diagrams from the hardware technical manual. The NT signal is held high for the entirety of the read, write, and refresh cycles, and low during the data transfer cycles. This allows the address output Aout[7..0] to represent the row address, which generated by counting the number of row requests to the VRAM controller using lpm_counter5, during a data transfer cycle. The counter uses a modulus of 272 since that is the size of the LCD's height in pixels. As a special note, the presence of the mux in the logic circuitry introduces a slight delay (of about 1 ns) to the correct address being output on the address bus following an assertion or dissertation of the RC signal in the VRAMController block. This delay is accounted for in the ***Timing Diagram for U21 and U22 Read/Early Write/Data Transfer Cycle*** diagrams.

The VRAMController block is written as a finite-state machine in the VHDL language. It uses statebits to represent each clock cycle during a particular cycle. The following code snippet shows the definitions for the VRAMController entity input and output ports.

```
entity VRAMStateMachine is
   port (
         clk      :  in std_logic;        -- system clock signal
         RQ       :  in std_logic;        -- row request signal (from display controller)
         WE       :  in std_logic;        -- write enable signal (from CPU)
         CS       :  in std_logic;        -- chip select signal (from CPU)
         Reset    :  in std_logic;        -- system reset signal
         RAS      :  out std_logic;       -- RAS signal (to VRAM)
         CAS      :  out std_logic;       -- CAS signal (to VRAM)
         RC       :  out std_logic;       -- ROW/COL signal (intermediate)
         WI       :  out std_logic;       -- write enable output signal (to VRAM)
         OE       :  out std_logic;       -- output enable signal (to VRAM)
         WQ       :  out std_logic;       -- wait request signal (to CPU)
         LC       :  out std_logic;       -- latch clear signal for row transfer (intermediate)
         NT       :  out std_logic        -- signal high when not in a row transfer cycle (intermediate)
         );
end   VRAMStateMachine;
```

VRAMController Port Definitions*: The input and output port definitions and descriptions for the VRAMController block in the VRAM Controller logic block diagram (seen in* ***VRAM Controller Signaling Diagram)***

As mentioned earlier, the VRAM Controller translates signals from the CPU and display controller into output signals. Based on the value of the RQ, WE, and CS inputs, the following code forces the state machine to enter one of the four cycles.

```
transition: process (Reset, RQ, WE, CS, CurrentState)
begin

   case CurrentState is              -- do the state transition/output
      -- input signals --> RQ, WE, CS
      WHEN  IDLE =>
         if (RQ = '1') then -- if row request signal input asserted
            NextState <= TONE;      -- then enter row transfer cycle
         elsif (RQ = '0' and CS ='1') then -- if no row request signal, and chip select inactive
            NextState <= FONE; -- then enter refresh cycle
         elsif (RQ = '0' and CS = '0' and WE ='0') then -- if no row request, but write-enable and chip select inputs active
            NextState <= WONE; -- then enter write cycle
         elsif (RQ = '0' and CS = '0' and WE = '1') then -- if no row request and write-enable, and chip select active
            NextState <= RONE; -- then enter read cycle
         else                  -- default state should be IDLE state
            NextState <= IDLE;
         end if;
```

VRAMController Cycle Entry Logic: *VHDL logic to determine which of the four cycles the state machine must enter based on the input values*

Once the state machine enters a particular cycle, the following code ensures that all twelve states of the cycle are completed in sequential order before the state machine returns to the IDLE states.

```
WHEN RONE =>              -- if a read cycle has been started, then go through all twelve read states
   NextState <= RTWO;
WHEN RTWO =>
   NextState <= RTHREE;
when RTHREE =>
   NextState <= RFOUR;
when RFOUR =>
   NextState <= RFIVE;
when RFIVE =>
   NextState <= RSIX;
when RSIX  =>
   NextState <= RSEVEN;
when RSEVEN =>
   NextState <= REIGHT;
when REIGHT =>
   NextState <= RNINE;
when RNINE =>
   NextState <= RTEN;
when RTEN =>
   NextState <= RELEVEN;
when RELEVEN =>
   NextState <= RTWELVE;
when RTWELVE =>          -- after finishing read cycle, return to idle state
   NextState <= IDLE;
```

```
when WONE =>             -- if a write cycle has been started, then go through all twelve write states
   NextState <= WTWO;
when WTWO =>
   NextState <= WTHREE;
when WTHREE =>
   NextState <= WFOUR;
when WFOUR =>
   NextState <= WFIVE;
when WFIVE =>
   NextState <= WSIX;
when WSIX  =>
   NextState <= WSEVEN;
when WSEVEN =>
   NextState <= WEIGHT;
when WEIGHT =>
   NextState <= WNINE;
when WNINE =>
   NextState <= WTEN;
when WTEN =>
   NextState <= WELEVEN;
when WELEVEN =>
   NextState <= WTWELVE;
when WTWELVE =>
   NextState <= IDLE;   -- after finishing write cycle, return to idle state
```

```
when FONE =>            -- if a refresh cycle has been started, then go through all twelve refresh states
   NextState <= FTWO;
when FTWO =>
   NextState <= FTHREE;
when FTHREE =>
   NextState <= FFOUR;
when FFOUR =>
   NextState <= FFIVE;
when FFIVE =>
   NextState <= FSIX;
when FSIX  =>
   NextState <= FSEVEN;
when FSEVEN =>
   NextState <= FEIGHT;
when FEIGHT =>
   NextState <= FNINE;
when FNINE =>
   NextState <= FTEN;
when FTEN =>
   NextState <= FELEVEN;
when FELEVEN =>
   NextState <= FTWELVE;
when FTWELVE =>        -- after finishing refresh cycle, return to idle state
   NextState <= IDLE;
```

```
when TONE =>           -- if a row transfer cycle has been started, then go through all twelve row transfer states
   NextState <= TTWO;
when TTWO =>
   NextState <= TTHREE;
when TTHREE =>
   NextState <= TFOUR;
when TFOUR =>
   NextState <= TFIVE;
when TFIVE =>
   NextState <= TSIX;
when TSIX  =>
   NextState <= TSEVEN;
when TSEVEN =>
   NextState <= TEIGHT;
when TEIGHT =>
   NextState <= TNINE;
when TNINE =>
   NextState <= TTEN;
when TTEN =>
   NextState <= TELEVEN;
when TELEVEN =>
   NextState <= TTWELVE;
when TTWELVE =>         -- after finishing row transfer cycle, return to idle state
   NextState <= IDLE;
when OTHERS  =>         -- in any other case, return to the idle state
   NextState <= IDLE;
```

VRAMController Cycle Progression Logic

The following code shows the statebit definition format as well as the statebit values for each state. Statebit definitions include 8 signal bits (12-5) and 5 ID bits (4-0) so that each of the four cycles is defined by twelve unique states (although the signal bits might not vary).

```
architecture   assign_statebits  of VRAMStateMachine  is
   subtype  states   is    std_logic_vector(12 downto 0);

   --define the actual states as constants
   -- state machine bit definitions  12  11  10  9   8  7  6  5  4  3  2  1  0
                                  --  NT  RAS CAS RC  WI OE LC WQ ID ID ID ID ID
                                  --  ID bits are used to make sure each state has a unique definition

   constant IDLE        :  states := "1111110100000"; -- idle state
-- Read Cycle Definitions
   constant RONE        :  states := "1111110100001"; -- initial read cycle state
   constant RTWO        :  states := "1011110100010"; -- bring RAS low while keeping other signals high
   constant RTHREE      :  states := "1010100110000"; -- bring OE low and switch from row to column
   constant RFOUR       :  states := "1010100111000"; -- keep previous state for another cycle
   constant RFIVE       :  states := "1000100111100"; -- bring CAS low
   constant RSIX        :  states := "1000100110000"; -- keep previous state
   constant RSEVEN      :  states := "1001100111000"; -- exit the column address output state
   constant REIGHT      :  states := "1111110000000"; -- simultaneously bring RAS and CAS high and pulse WQ
   constant RNINE       :  states := "1111110100110"; -- return wait request high (must pulse for only one clock cycle)
   constant RTEN        :  states := "1111110110001"; -- keep previous state
   constant RELEVEN     :  states := "1111110101000"; -- keep previous state
   constant RTWELVE     :  states := "1111110110000"; -- keep previous state
-- Write Cycle Definitions
   constant WONE        :  states := "1111110100100"; -- initial write cycle state (same as read cycle )
   constant WTWO        :  states := "1011110101000"; -- bring RAS low while keeping other signals high
   constant WTHREE      :  states := "1010110100000"; -- switch from row to column address output
   constant WFOUR       :  states := "1010010100001"; -- bring WE low
   constant WFIVE       :  states := "1000010000000"; -- bring CAS low and pulse WQ low
   constant WSIX        :  states := "1000010110000"; -- bring WQ high again since its pulsed only once cycle
   constant WSEVEN      :  states := "1001010111000"; -- stop outputing column address (return to row state)
   constant WEIGHT      :  states := "1111110111000"; -- bring RAS, CAS, and WI high simultaneously
   constant WNINE       :  states := "1111110110100"; -- keep previous state
   constant WTEN        :  states := "1111110101100"; -- keep previous state
   constant WELEVEN     :  states := "1111110111100"; -- keep previous state
   constant WTWELVE     :  states := "1111110100101"; -- keep previous state
```

```
-- Refresh Cycle Definitions
   constant FONE        :  states := "1101110110000"; -- initial refresh state (all signals in idle value, except CAS is low)
   constant FTWO        :  states := "1001110110010"; -- bring RAS low
   constant FTHREE      :  states := "1001110101000"; -- keep previous state
   constant FFOUR       :  states := "1001110100100"; -- keep previous state
   constant FFIVE       :  states := "1001110111000"; -- keep previous state
   constant FSIX        :  states := "1001110110100"; -- keep previous state
   constant FSEVEN      :  states := "1001110101100"; -- keep previous state
   constant FEIGHT      :  states := "1111110100011"; -- bring RAS and CAS high simultaneously
   constant FNINE       :  states := "1111110101001"; -- keep previous state
   constant FTEN        :  states := "1111110100111"; -- keep previous state
   constant FELEVEN     :  states := "1111110111101"; -- keep previous state
   constant FTWELVE     :  states := "1111110111111"; -- keep previous state
-- Row Transfer Cycle Definitions
   constant TONE        :  states := "0111000111100"; -- initial transfer state (NT, WI, OE, LC low , others high)
   constant TTWO        :  states := "0011000100000"; -- bring RAS low
   constant TTHREE      :  states := "0010000110000"; -- switch from row to column address output
   constant TFOUR       :  states := "0000000101110"; -- keep previous state
   constant TFIVE       :  states := "0000000100110"; -- keep previous state
   constant TSIX        :  states := "0001000100000"; -- stop outputing column address (return to row state)
   constant TSEVEN      :  states := "0001010100000"; -- bring OE high
   constant TEIGHT      :  states := "0111010100000"; -- bring RAS and CAS high simultaneously
   constant TNINE       :  states := "0111010110000"; -- stop outputing column address (return to row state)
   constant TTEN        :  states := "0111010110011"; -- stop outputing column address (return to row state)
   constant TELEVEN     :  states := "0111011101000"; -- pulse LC high
   constant TTWELVE     :  states := "0111010110001"; -- bring LC back low (so it pulses for only one clock cycle)
```

VRAMController State Definition Format and Definitions

The following code snippet corresponds each output bit in the statebit to the appropriate output pins seen in ***VRAMController Port Definitions*** and ***VRAM Controller Signaling Diagram.*** During each state transition, the VRAMController port output values change to match the outputs bits for the current state.

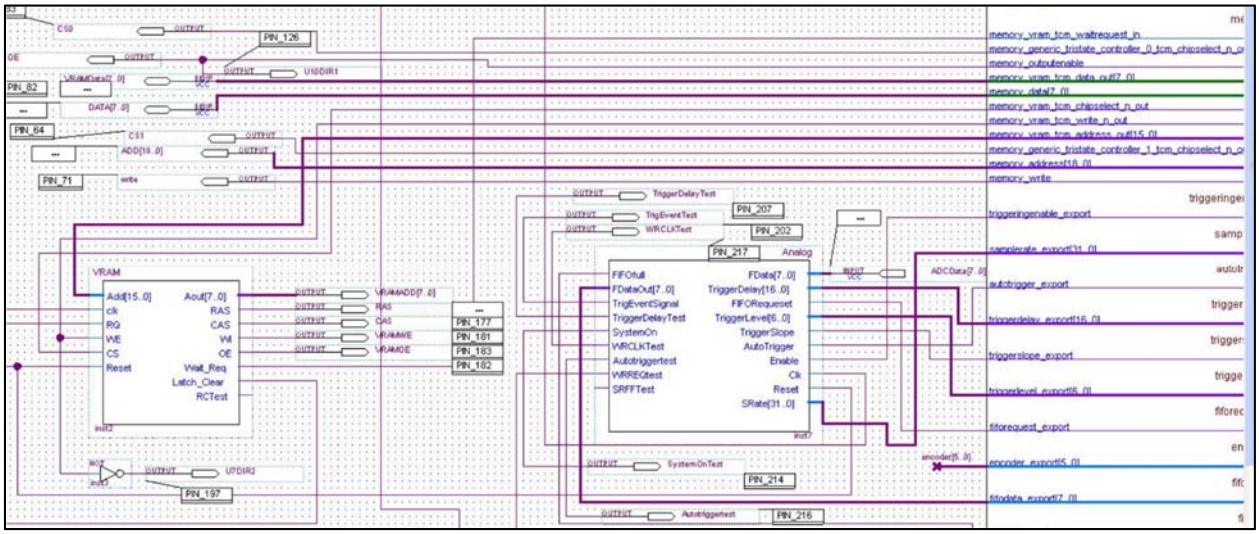
```
   --the output for each signal depends on the value of the state bit (minus the error bits)
         NT    <= CurrentState(12);        --Not in a row transfer cycle output signal (intermediate)
         RAS   <= CurrentState(11);        --RAS output signal
         CAS   <= CurrentState(10);        --CAS output signal
         RC    <= CurrentState(9);         -- ROW/COL signal (intermediate)
         WI    <= CurrentState(8);         -- write enable output signal
         OE    <= CurrentState(7);         -- output enable signal
         LC    <= CurrentState(6);         -- latch clear signal for row transfer
         WQ    <= CurrentState(5);         -- wait request signal
```

VRAMController Signal Outputs Corresponding to Statebits:

Finally, the state machine VHDL code dictates that each state transition occurs on a positive clock edge, since clk is an input to the state machine. Also, any assertion of the Reset input to logic high forces the state machine to return to the IDLE state, no matter what the current cycle the state machine is in. The following code snippets describes this behavior.

```
        if Reset = '1' then        -- if reset is asserted high (active high), immediately go to idle state no matter current state
              NextState <= IDLE;
        end if;
    end process transition;

    -- storage of current state (loads the next state on the clock)
    process (clk)
    begin

        if clk = '1' then
            CurrentState <= NextState;
        end if;

    end process;
```

The following logic in the diagram ***VRAM Controller and CPU*** shows the connections between the CPU and the VRAM Controller. The connections between the VRAM Controller and the Display Controller are the subject of the next section and the diagram ***Main Block-Display and VRAM Controllers.***



Main Block- VRAM Controller*: The relationship between the CPU and VRAM Controller blocks. The VRAM Controller is the block labeled VRAM.*

Although the diagram doesn't show it, the clk and Reset inputs to the VRAM Controller are connected to the system clock and system reset inputs, respectively. The 16 bit Add[15..0] input is connected to the 16-bit address output originating from the CPU's VRAM Generic Tri-State Controller, WE to the VRAM Generic Tri-State Controller's write enable output, and CS to the chip select output. The Tri-State Controller is configured to have write enable and chip select as active low signals, corresponding to the logic seen in the ***VRAMController Cycle Entry Logic*** diagram. The Aout[7..0] 8-bit address bus, RAS, CAS, WI, and OE are directly sent out through I/O pins (*see* ***Pin List Table*** *at end of document)* to be buffered and received by the U21 and U22. The buffer components involved in this signaling can be found in the **VRAM** section of the

hardware technical manual. Lastly, The Wait_Req output signal is sent to be received by the VRAM Generic Tri-State Controller's wait request input in the CPU.

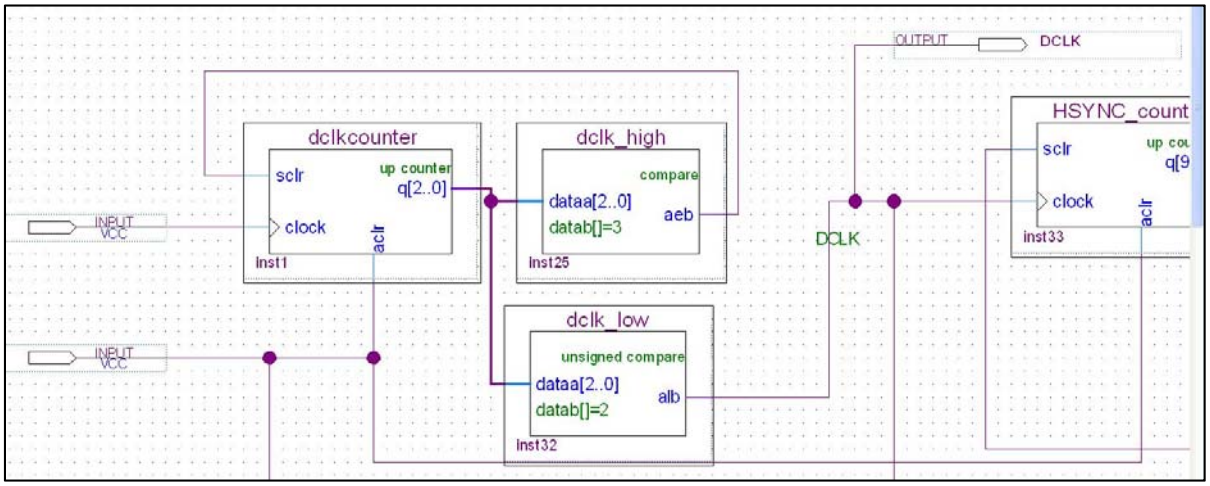
The complete pin list found on page 88 describes the I/O pins corresponding to each pin involved in the VRAM controller logic shown above in ***VRAM Controller and CPU***.

Display Controller and Display Software

Display Signal Generation

The design for the display controller implemented in the FPGA is responsible for generating the control signals (DCLK, HSYNC, VSYNC, and DE) that are received by the LCD, as well as the serial clock and row transfer signals that are received by the VRAM block and VRAM controller respectively. The timing and synchronization of these signals is shown in the **Display** section in the hardware technical manual on pages 39-40.

The following diagram illustrates the design for the FPGA logic used to generate the DCLK signal.

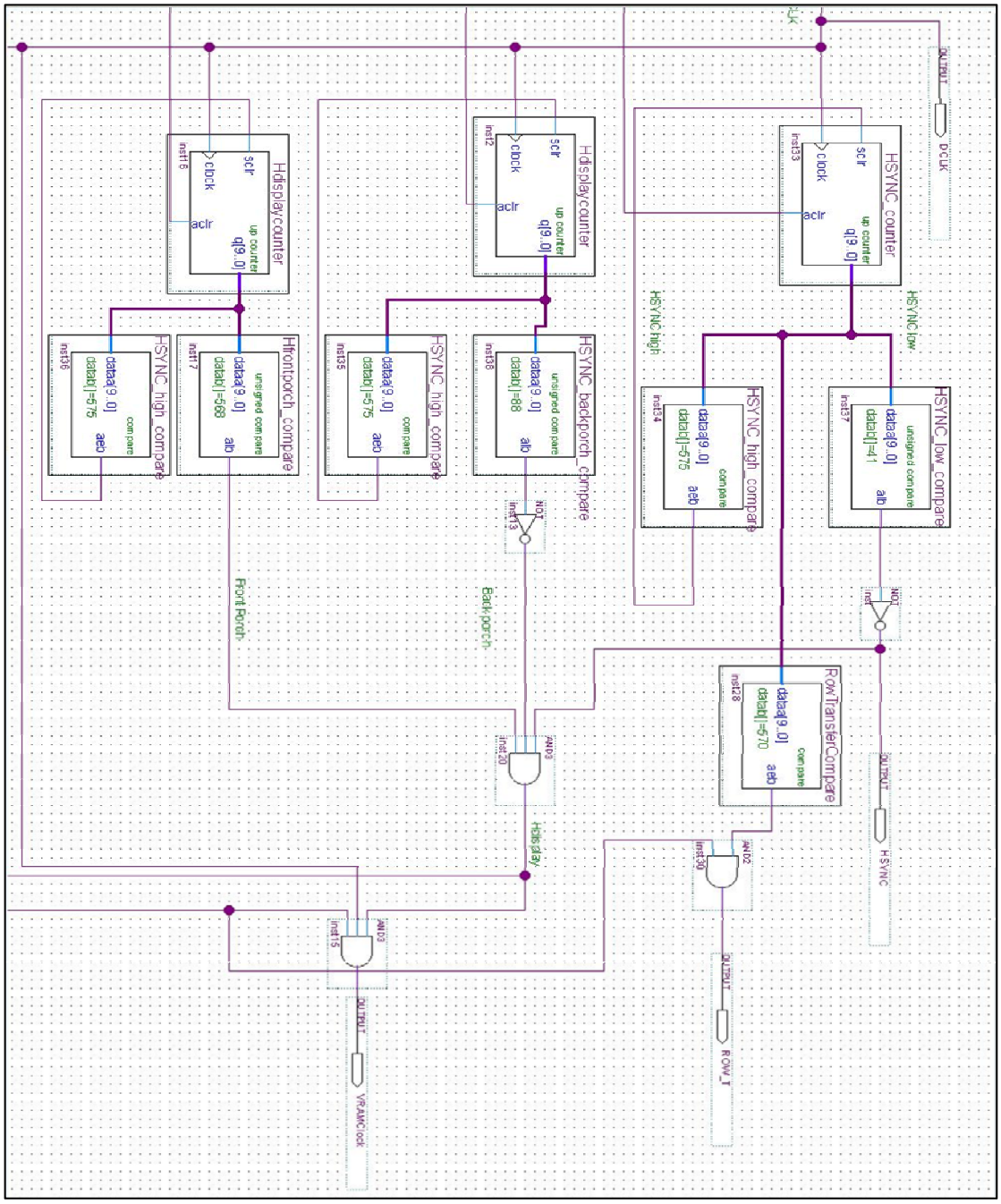


DCLK Signal Generation: *Generating DCLK signal from system clock input*

Since the DLCK frequency needs to be 9 MHz, and the system clock runs at 36 MHz, the design shown in ***DCLK Signal Generation*** simply combines four system clock cycles into one DCLK cycle. Using the counter dclk counter to count system clock cycles, dclk_high keeps the signal high for 2 cycles, while dclk_low keeps the signal low for 2 cycles. The generated DCLK signal is used as the clock input for the logic to generate HSYNC and VSYNC. The Reset input, which is the system reset, is used to asynchronously clear all blocks in the logic design.

The diagram ***HSYNC and Row Transfer Signal Generation*** on **page x** illustrates the logic used to generate the HSYNC signal. The counters for the generation of these signals run on the DCLK signal generated earlier. Since the LCD uses color, the display controller logic generates both the actual HSYNC signal (one that is held high during the HSYNC back and front porches) that is sent out to the actual display (given by the output pin HSYNC), as well as variant of the HSYNC signal that is low during the porches that is used to help generate the serial clock signal (marked Hdisplay in the diagram). A counter (HSYNC_counter) and two comparators (HSYNC_low_compare and HSYNC_high_compare) are used to generate the actual HSYNC signal. Using HSYNC_counter to count number of DCLK cycles, HSYNC_low_compare and a NOT gate keep HSYNC low for 41 DCLK cycles, while HSYNC_high_compare is used to keep HSYNC high for 576 DCLK cycles.

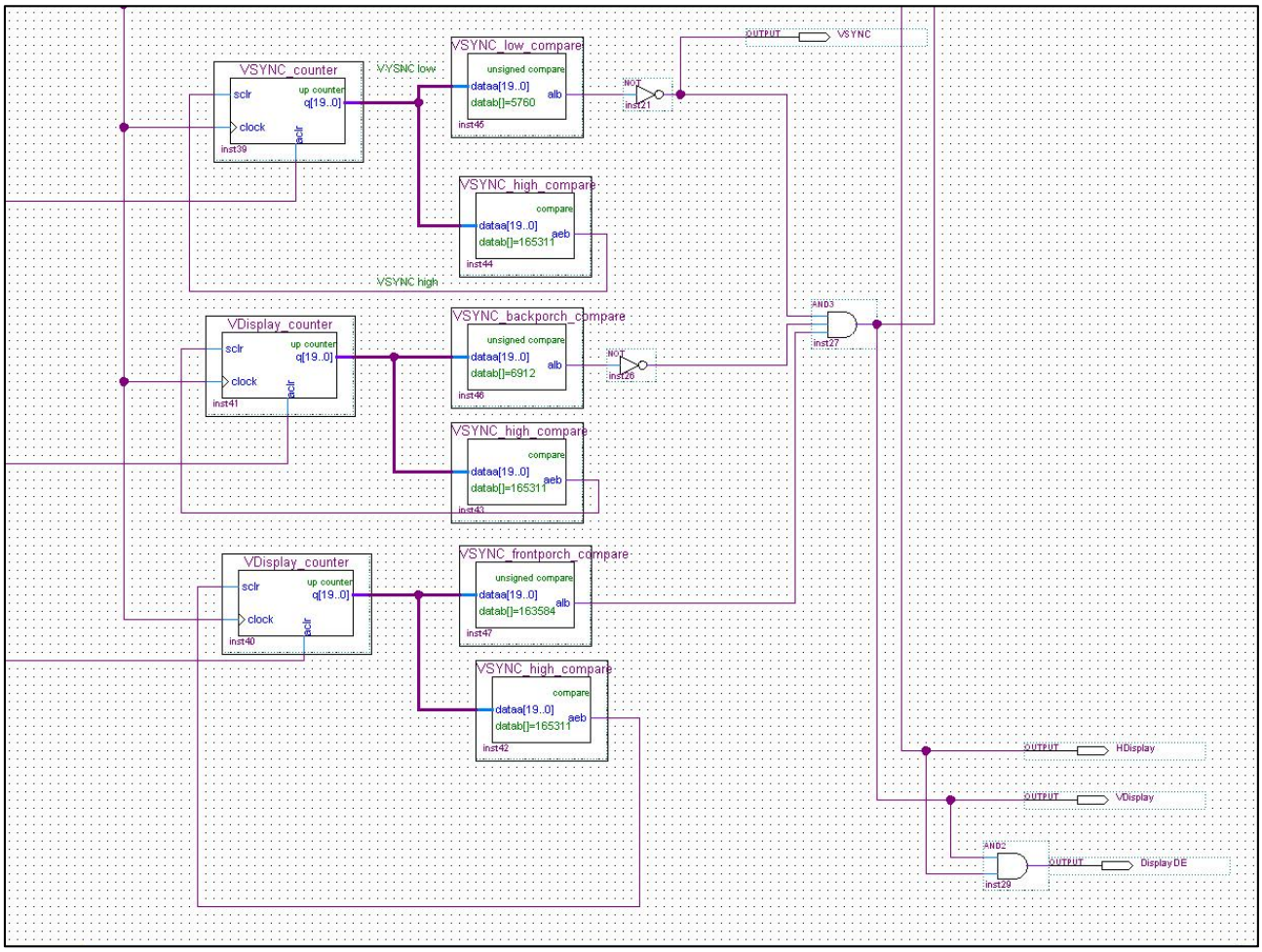
To generate the Hdisplay signal, two counters are each combined with two sets of two comparators. Using the first instance of the counter Hdisplaycounter (inst2), comparator HSYNC_backporch_compare and a NOT gate (inst13) keep the signal low for 88 DCLK cycles, and comparator HSYNC_high_compare (inst35) keeps the signal high for 576 DCLK cycles. This step, which accounts for the HSYNC back porch, is AND'd with the signal generated by another instance of counter Hdisplaycounter (inst16), comparator Hfrontporch_compare, and



comparator HSYNC_high_compare (inst36). This ensures that Hdisplay signal goes low after 568 DCLK cycles to account for the front porch.

HSYNC, Hdisplay, and Row Transfer Signal Generation*: Generation of row transfer, HSYNC, and Hdisplay signals.*

The generation of the VSYNC and the VDisplay signal (where VSYNC is low during the VSYNC back and front porches) follows an identical pattern to the one shown above. However, the length and timing of the cycles is different. The diagram **VSYNC and DE Signal Generation** illustrates the generation of these signals (note: VDisplay is the VSYNC equivalent of the HDisplay signal for HSYNC). Once again, the clock inputs to the diagram's counters are DCLK.



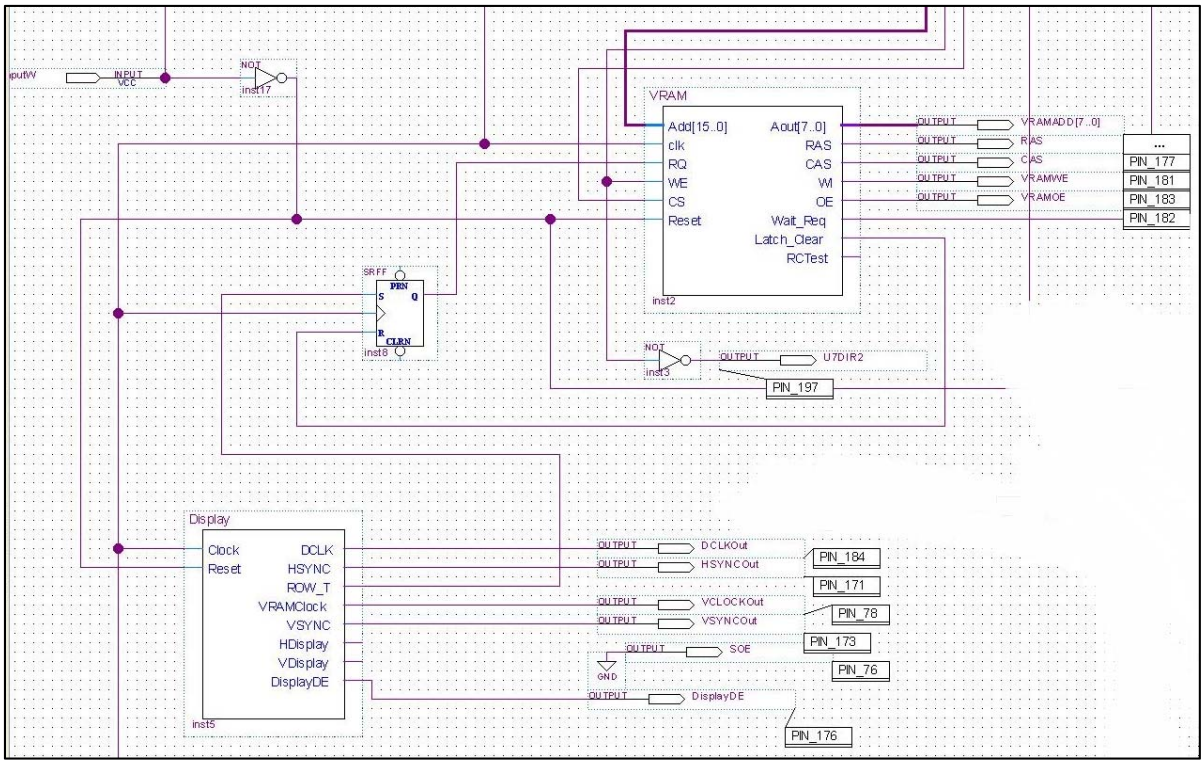
VSYNC and DE Signal Generation: *Generation of DE, VSYNC, and VDisplay signals*

VDisplay and HDisplay are both fed into an AND gate (AND2), the output of which gives DisplayDE, the DE signal that is sent out to the display. VDisplay, HDisplay, and DCLK are AND'd (AND3 inst 15) together to give VRAMClock (shown in ***HSYNC, Hintermediate, and Row Transfer Signal Generation***), which is the serial clock signal that is received by the VRAM block.

HSYNC, Hdisplay, and Row Transfer Signal Generation also illustrates the generation of ROW_T, the row transfer signal. Due to the AND gate (AND2) and comparator

RowTransferCompare, the signal is asserted high every time VDisplay is high and HSYNC is on the 570th DCLK cycle.

The ***VSYNC and DE Signal Generation***, the ***HSYNC, Hdisplay, and Row Transfer Signal Generation ,*** and the ***DCLK Signal Generation*** diagrams comprise the system's display controller block. The ***Main Block-Display and VRAM Controllers*** diagram shows the relationship between the display controller block and the VRAM controller block.



Main Block-Display and VRAM Controllers*: Relationship between the display and VRAM controller blocks.*

As seen in the diagram, the clock input into the display controller is the system clock signal, and the reset input is the system reset signal. The HDisplay and VDisplay outputs are left unconnected, since they are intermediate signals used in the generation of other signals (left for debugging purposes)

The row transfer output from the display controller (ROW_T) is first passed through an SR Flip Flop (SRFF), the output of which is received by the VRAM controller as the row request signal (RQ). The SRFF is cleared by the Latch_Clear output of the VRAM controller. This is to ensure that the row request signal is seen by the VRAM controller and cleared once the row transfer cycle is complete, since the ROW_T output from the display controller is only pulsed high for one DCLK cycle.

The DCLKOut, HSYNCOut, VCLOCKOut, VSYNCOut, and DisplayDE output pins correspond to the DCLK, HSYNC, SC, VSYNC, and DE outputs that are buffered and sent to the appropriate components. The buffer control for these signals is found in the hardware technical

manual. Finally, the complete pin list found on at the end of this document describes the I/O pins corresponding to each pin involved in the display controller logic shown above in ***Main Block-Display and VRAM Controller***

Display Software Control

The ***Software Main Loop Block Diagram*** includes two functions in the file Display.S, clear_display() and plot_pixel(), that are called by the software main loop. The following is the contents of the Display.S file.

```
#;;;
```

```
    movi    r9, PIXEL_WHITE             #color bit value that we want to
write
                                        #to each VRAM address (PIXEL_WHITE)
    stwio   r9, (r10)                   #write this value to the VRAM address
    movi    r8, 1                       #prepare the jump to next VRAM
address
    add     r10, r10, r8                #increment VRAM address
    beq     r10, r11, clear_display_done #check if we covered the entire VRAM
    jmpi    clear_display_loop          #if not, then keep looping
clear_display_done:                     #if we did, then end the function
    ret                                 #return

# void plot_pixel(unsigned int x, unsigned int y, int p)
# Description:      Plot the passed-in pixel value to the x and y coordinate
#                   specified in the argument on the LCD
# Operation:
# Arguments:        unsigned int x: x coordinate where pixel will be plotted
#                       will be stored in r4 upon function entry
#                   unsigned int y: y coordinate where pixel will be plotted
#                       will be stored in r5 upon function entry
#                   int p: passed pixel value
#                       will be stored in r6 upon function entry
# Return Value:     None
# Local Variables:  None
# Shared Variables: None
# Global Variables: None
# Input:            None
# Output:           VRAM will be updated and display as a result
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r5, r4, r10, r6
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
plot_pixel:
    slli    r5, r5, 8                   #shift the y coordinate value by 1
                                        # byte
    or      r5, r5, r4                  #or in the x coordinate so r5 now
                                        #contains the offset within the VRAM
                                        #which corresponds to the coordinate
                                        #that needs to plotted with the pixel
    movhi   r10, %hi(VRAM_BASE)         #get the high bits of the VRAM base
                                        #address
    ori     r10, r10, %lo(VRAM_BASE)    #get the complete VRAM base address
    or      r10, r10, r5                #OR in the offset to the VRAM base
                                        #address to get the correct
                                        # coordinate on the display
    stbio   r6, 0(r10)                  #write the pixel value to the VRAM
                                        #address (base + offset) that is
                                        #stored in r6
    ret                                 #return
```

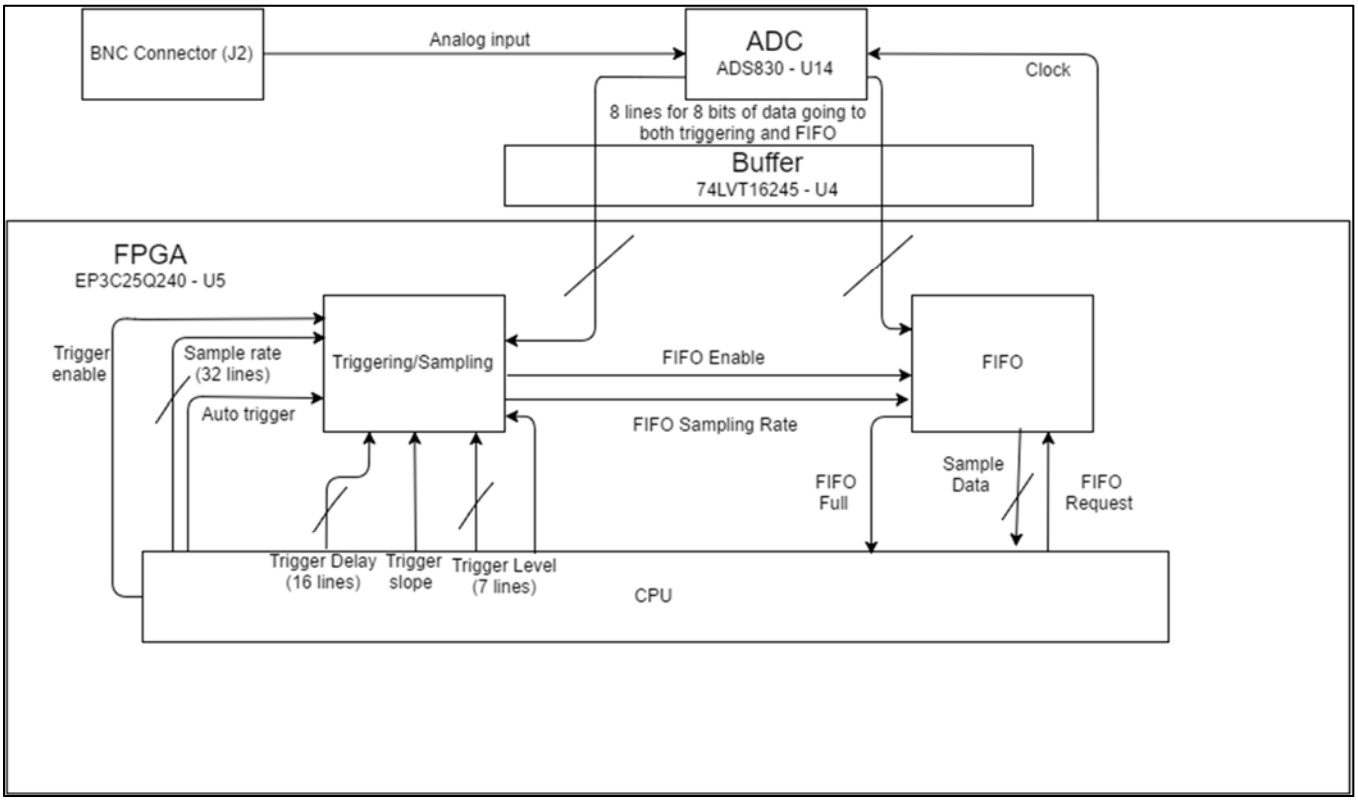
The actual values for the PIXEL_WHITE and PIXEL_BLACK constants, which are the only two colors used in the display, can be found in the interface.h file (shown on page 48). Since U21 and

U22 are connected in serial, pixel values that are sent to them for plotting are configured so that the following bit configuration is followed:

Bit Number	**Bit Function**
0 (MSB)	Red bit 1
1	Red bit 2
2	Red bit 3
3	Green bit 1
4	Green bit 2
5	Green bit 3
6	Blue bit 1
7 (LSB)	Blue bit 2

Analog Logic and Software

Proper functioning of the analog portion of the system involves the logic implemented in the FPGA as the analog block, which includes the triggering block and internal FIFO, as well as functions found in the "Analog.S" file that interface with the main software loop. The logic for the triggering block and internal FIFO (the analog block) is responsible for receiving raw digital data from the ADC block, processing it, and supplying the CPU with sample data to be displayed at an appropriate rate. An outline of this process, and the signals involved, is the subject of the following diagram, ***Analog Block Signaling.***



Analog Signaling: *A block diagram showing the signaling in the hardware and software sides of the analog block.*

As seen in ***Analog Signaling***, the buffered 8-bit digital bus from the ADC is sent to both the internal FIFO and triggering/sampling block. Using the trigger delay, slope, level, enable, and auto trigger inputs from the CPU, the triggering/sampling block determines whether or not the FIFO should start storing ADC data as sample data, as well as the rate to do so. The FIFO enable line enables the triggering block to tell the FIFO to start storing sample data, while the FIFO Sampling Rate line is a clock line that tells the FIFO how fast to collect the data.

The FIFO Request signal is used by the CPU to receive sample data from the FIFO when it is ready to do so. Finally, when the FIFO is filled with sample data, the FIFO Full signal is used to indicate this to the CPU. The internal FIFO used in the software has a width of 8 bits and a depth of 1024 words. Therefore, a full FIFO can hold 1024 different samples, each with a resolution of 8 bits.

As described in the **CPU Software** section, Trigger Enable, Sample Rate, Auto Trigger, Trigger Delay, Trigger Slope, Trigger Level, FIFO Request, FIFO Full, and Sample Data are setup as PIOs in the CPU. The locations of these input/outputs in memory is given by the following memory map.

Component	**Memory Address (Start – End)**
ROM	0x0008000 – 0x000FFFF
SRAM	0x00120000-0x0013FFFF
Trigger Enable	0x00181170-0x0018117F
Sample Rate	0x00181160-0x0018116F
Auto Trigger	0x00181150-0x0018115F
Trigger Delay	0x00181140-0x0018114F
Trigger Slope	0x00181130-0x0018113F
Trigger Level	0x00181120-0x0018112F
FIFO Request	0x00181110-0x0018111F
On-chip memory	0x00170000-0x00179FFF
Keypress	0x00181100-0x001810FF
VRAM	0x00150000-0x0015FFFF
FIFO Data	0x001810F0-0x001810FF
FIFO Full	0x001810E0-0x001810EF

All PIOs other than FIFO Full and FIFO Data, which are outputs, are setup as inputs into the CPU. The setter functions found in Analog.S, which are set_trigger(int level, int slope), set_delay(long int delay), and set_sample_rate(long int samples_per_sec), write the appropriate value to the memory location for the corresponding parameter. The code for these functions is given below.

```
# int set_sample_rate(long int samples_per_sec)
# Description:      Function sets the oscillscope sample rate to the value
#                   passed in the argument
# Operation:        The function checks the passed sample rate (in samples
per
#                   in second) against 17 possible sample rates. It then
writes
#                   the corresponding sample rate (in units of number of
system
#                   clock cycles per sample) to the sample rate address in
#                   the CPU (PIO 1). Finally, it returns the number of
samples
#                   that will be sampled, which has a value of 1024 no matter
#                   what the sample rate is
# Arguments:        None
# Return Value:     None
# Local Variables:  None
# Shared Variables: None
# Global Variables: None
# Input:            None
# Output:           Sample Rate will be updated
```

```
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r10, r5, r4, r2
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
set_sample_rate:
    movhi   r10, %hi(PIO_1_BASE)        #get the high bits of the PIO 1 base
                                        #address (where the sample rate is
                                        #stored in memory)
    ori     r10, r10, %lo(PIO_1_BASE)   #get the full PIO 1 base address by
                                        #ORing in the low bits as well
    movhi   r5, %hi(11990407)           #check the first sample rate case and
                                        #store the high bits of the case
                                        #value
                                        #in r5
    ori     r5, r5, %lo(11990407)       #add in the low bits to store the
                                        # whole
                                       #value in r5 so that it can be checked
                                        #with the passed argument
    beq     r5, r4, sample_rate_case_1  #if the passed argument matches the
                                        #first case, then jump accordintly
    movhi   r5, %hi(5142446)            #do the same for  the second
                                        # sample rate case
    ori     r5, r5, %lo(5142446)
    beq     r5, r4, sample_rate_case_2
    movhi   r5, %hi(2000000)
    ori     r5, r5, %lo(2000000)
    beq     r5, r4, sample_rate_case_3  #do the same for  the third
                                        # sample rate case
    ori     r5, r5, %lo(5142446)
    movhi   r5, %hi(1000000)
    ori     r5, r5, %lo(1000000)
    beq     r5, r4, sample_rate_case_4 #do the same for  the 4th
                                        # sample rate case
    movhi   r5, %hi(500000)
    ori     r5, r5, %lo(500000)
    beq     r5, r4, sample_rate_case_5 #do the same for  the 5th
                                        # sample rate case
    movhi   r5, %hi(200000)
    ori     r5, r5, %lo(200000)
    beq     r5, r4, sample_rate_case_6 #do the same for  the 6th
                                        # sample rate case
    movhi   r5, %hi(100000)
    ori     r5, r5, %lo(100000)
    beq     r5, r4, sample_rate_case_7 #do the same for  the 7th
                                        # sample rate case
    movhi   r5, %hi(50000)
    ori     r5, r5, %lo(50000)
    beq     r5, r4, sample_rate_case_8 #do the same for  the 8th
                                        # sample rate case
    movhi   r5, %hi(20000)
    ori     r5, r5, %lo(20000)
    beq     r5, r4, sample_rate_case_9 #do the same for  the 9th
                                        # sample rate case
    movhi   r5, %hi(10000)
```

```
    ori     r5, r5, %lo(10000)
    beq     r5, r4, sample_rate_case_10 #do the same for  the 10th
                                        # sample rate case
    movhi   r5, %hi(5000)
    ori     r5, r5, %lo(5000)
    beq     r5, r4, sample_rate_case_11 #do the same for  the 11th
                                        # sample rate case
    movhi   r5, %hi(2000)
    ori     r5, r5, %lo(2000)
    beq     r5, r4, sample_rate_case_12 #do the same for  the 12th
                                        # sample rate case
    movhi   r5, %hi(1000)
    ori     r5, r5, %lo(1000)
    beq     r5, r4, sample_rate_case_13 #do the same for  the 13th
                                        # sample rate case
    movhi   r5, %hi(500)
    ori     r5, r5, %lo(500)
    beq     r5, r4, sample_rate_case_14 #do the same for  the 14th
                                        # sample rate case
    movhi   r5, %hi(200)
    ori     r5, r5, %lo(200)
    beq     r5, r4, sample_rate_case_15 #do the same for  the 15th
                                        # sample rate case
    movhi   r5, %hi(100)
    ori     r5, r5, %lo(100)
    beq     r5, r4, sample_rate_case_16 #do the same for  the 16th
                                        # sample rate case
    movhi   r5, %hi(50)
    ori     r5, r5, %lo(50)
    beq     r5, r4, sample_rate_case_17 #do the same for  the 17th
                                        # sample rate case

sample_rate_case_illegal:               #if the passed sample rate doesn't
                                        #match any of the stored case values
                                        #then jump to the end of the function
    jmpi    sample_rate_end

sample_rate_case_1:                     #if the passed sample rate matches
                                        #case
                                        #1, store the appropriate value (in
                                        #units of number of system clock
                                        #cycles per sample in r4)
    movi    r4, 3
    jmpi    sample_rate_end

sample_rate_case_2:
    movi    r4, 7                       #do the same for case 2
    jmpi    sample_rate_end
sample_rate_case_3:                     #do the same for case 3
    movi    r4, 18
    jmpi    sample_rate_end
sample_rate_case_4:                     #do the same for case 4
    movi    r4, 36
    jmpi    sample_rate_end
sample_rate_case_5:                     #do the same for case 5
    movi    r4, 72
    jmpi    sample_rate_end
```

```
sample_rate_case_6:                     #do the same for case 6
    movi    r4, 180
    jmpi    sample_rate_end
sample_rate_case_7:                     #do the same for case 7
    movi    r4, 360
    jmpi    sample_rate_end
sample_rate_case_8:                     #do the same for case 8
    movi    r4, 720
    jmpi    sample_rate_end
sample_rate_case_9:                     #do the same for case 9
    movi    r4, 1800
    jmpi    sample_rate_end
sample_rate_case_10:                    #do the same for case 10
    movi    r4, 3600
    jmpi    sample_rate_end
sample_rate_case_11:                    #do the same for case 11
    movi    r4, 7200
    jmpi    sample_rate_end
sample_rate_case_12:                    #do the same for case 12
    movi    r4, 18000
    jmpi    sample_rate_end
sample_rate_case_13:                    #do the same for case 13
    movhi   r4, %hi(36000)
    ori     r4, r4, %lo(36000)
    jmpi    sample_rate_end
sample_rate_case_14:                    #do the same for case 14
    movhi   r4, %hi(72000)
    ori     r4, r4, %lo(72000)
    jmpi    sample_rate_end
sample_rate_case_15:                    #do the same for case 15
    movhi   r4, %hi(180000)
    ori     r4, r4, %lo(180000)
    jmpi    sample_rate_end
sample_rate_case_16:                    #do the same for case 16
    movhi   r4, %hi(360000)
    ori     r4, r4, %lo(360000)
    jmpi    sample_rate_end
sample_rate_case_17:                    #do the same for case 17
    movhi   r4, %hi(720000)
    ori     r4, r4, %lo(720000)
    jmpi    sample_rate_end

sample_rate_end:
    stwio   r4, 0(r10)                  #since the sample rate value (inunits
                                        #of number of system clk cycles per
                                        #sample) in stored in r4, it can be
                                        #stored in the PIO 1 based address,
                                        #which will update the system's
                                        #Sample Rate value
    movi    r2, 1024                   #return the number of samples that will
                                        #be returned by storing the value in
                                        #r2. As explained before, this value
                                        #is a constant
    ret


# void set_trigger(int level, int slope)
```

```
# Description:      Function sets the oscillscope trigger level to the first
#                   argument and trigger slope to the second argument.
# Operation:        The function writes the value passed in the first
argument
#                   to the address for the trigger level in memory, which is
#                   PIO 5 in the CPU. It does the same to the value in the
#                   second argument to the address for trigger slope in
memory
#                   which is PIO 4 in the CPU
# Arguments:        Int level - trigger level, a value between 0 and 127
#                   inclusiev
#                   int slope - trigger slope, with 1 for negative slope and
0
#                   for positive slope
# Return Value:     None
# Local Variables:  None
# Shared Variables: None
# Global Variables: None
# Input:            None
# Output:           Trigger level and trigger slope will be updated in memory
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r10
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
set_trigger:
    movhi   r10, %hi(PIO_5_BASE)        #get the address for PIO 5 base,which
                                        #is where trigger level is stored, in
                                        #r10 high bits first
    ori     r10, r10, %lo(PIO_5_BASE)   #get the low bits to complete thebase
                                        #address value in r10
    stwio   r4, 0(r10)                  #write the passed trigger level value
                                        #to memory
    movhi   r10, %hi(PIO_4_BASE)        #get the address for PIO 4 base,which
                                        #is where trigger slope is stored, in
                                        #r10 high bits first
    ori     r10, r10, %lo(PIO_4_BASE)   #get the low bits to complete thebase
                                        #address value in r10
    stwio   r5, 0(r10)                  #write the passed trigger slope value
                                        #to memory
    ret                                 #return

# void set_delay(long int delay)
# Description:      Function sets the oscillscope trigger delay to the
#                   argument passed
# Operation:        The function writes the value passed in the argument
#                   to the address for the trigger delay in memory, which is
#                   given by PIO 3 in the CPU
# Arguments:        long int delay - trigger delay (in samples)
# Return Value:     None
# Local Variables:  None
# Shared Variables: None
# Global Variables: None
# Input:            None
# Output:           Trigger delay will be updated in memory
```

```
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r10
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
set_delay:
    movhi   r10, %hi(PIO_3_BASE)        #get the address for PIO 3 base,which
                                        #is where trigger delay value is
                                        # stored, in r10 (high bits first)
    ori     r10, r10, %lo(PIO_3_BASE)   #get the low bits to complete thebase
                                        #address value in r10
    stwio   r4, 0(r10)                  #write the passed trigger delay value
                                        #to the appropraite location inmemory
    ret                                 #return
```

These functions serve to update the trigger level, slope, delay, and sample rate, which are inputs to the analog logic block, which includes the triggering block and internal FIFO. The following figures ***Analog Block Logic Diagram (1 and 2)*** on pages 84-85 show the design for the logic that comprises the analog block.

The first figure in the series depicts the ScopeTrigger component, the inputs of which include the trigger slope value, an input that indicates if the input data is equal to the trigger value, and a an input that indicates if the input data value is less than the trigger value, as well as the clock and reset inputs. The triggerlevel_compare comparator is used to determine if the input data value is equal to, or less than, the trigger level value. The component has only one output, TrigEvent, which goes high if a trigger event has occurred based on the given trigger parameters. This TrigEvent output is OR'd (using the OR2 gate) with the autotrigger signal (which is debounced using the autotrigger_counter and compare3 comparators for 1000 clock cycles). This allows the system to overrule the trigger conditions if the user wishes to use the autotrigger feature.

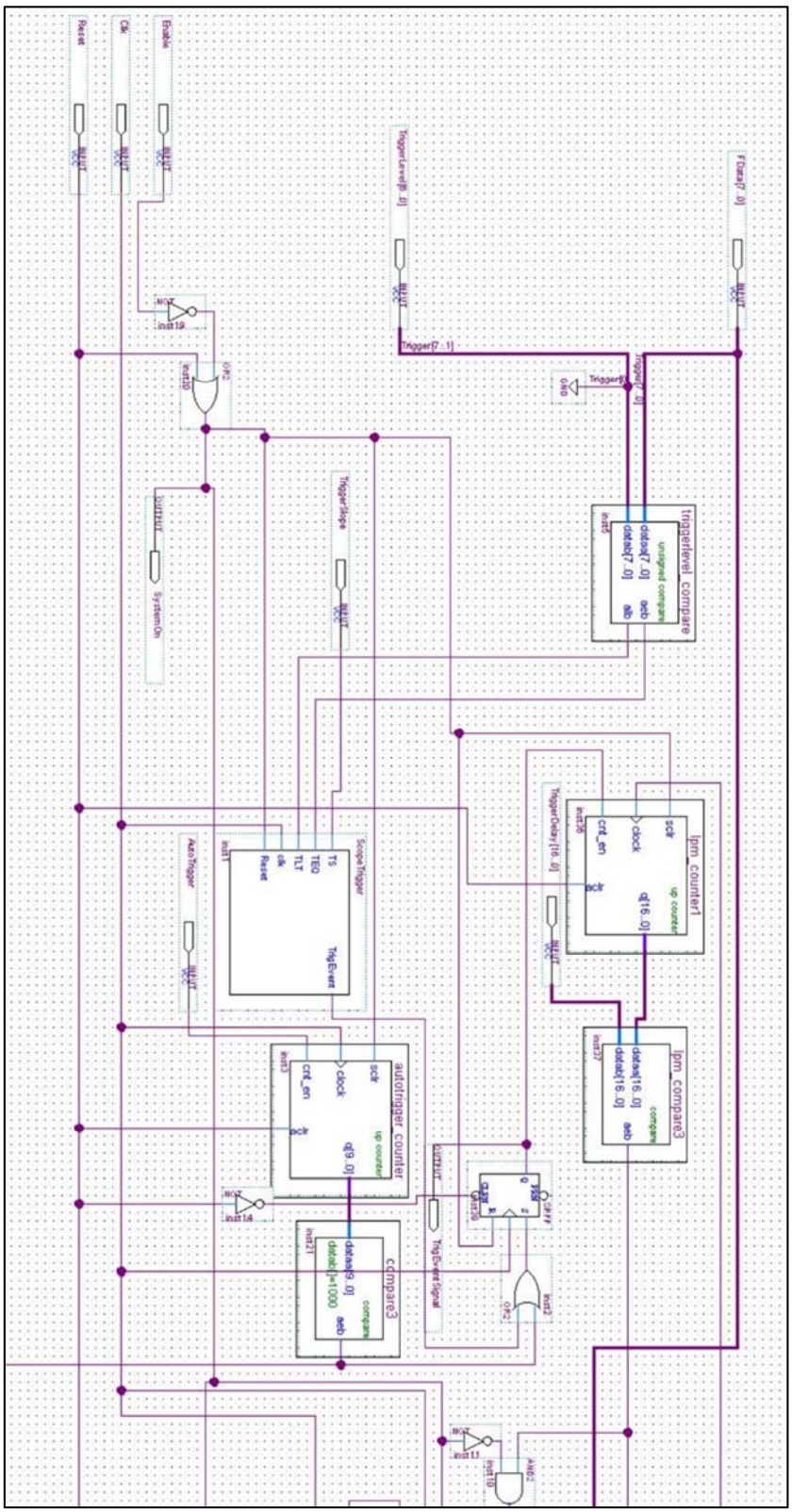
Control and activation of the triggering block and FIFO in the analog block is offered by the reset and trigger enable inputs (Reset and Enable pins on the diagram). While the reset is powered by the system reset, the trigger enable input allows software control of when to activate the analog logic block through the start_sample() function, the code for which is as follows.

```
 # void start_sample(int auto_trigger)

# Description:      Function starts sampling data. If the passed argument
#                   is false, then this sampling only starts when there is a
#                   triggering event. If its true, then sampling starts
#                   immediately regardless.
# Operation:        The function asserts the trigger enable signal by writing
#                   a 1 to its location in memory (given by PIO 0). Function
#                   also writes the passed argument to the system autotrigger
#                   value location in memory (given by PIO 2)
# Arguments:        int auto_trigger -
# Return Value:     None
# Local Variables:  None
# Shared Variables: None
```

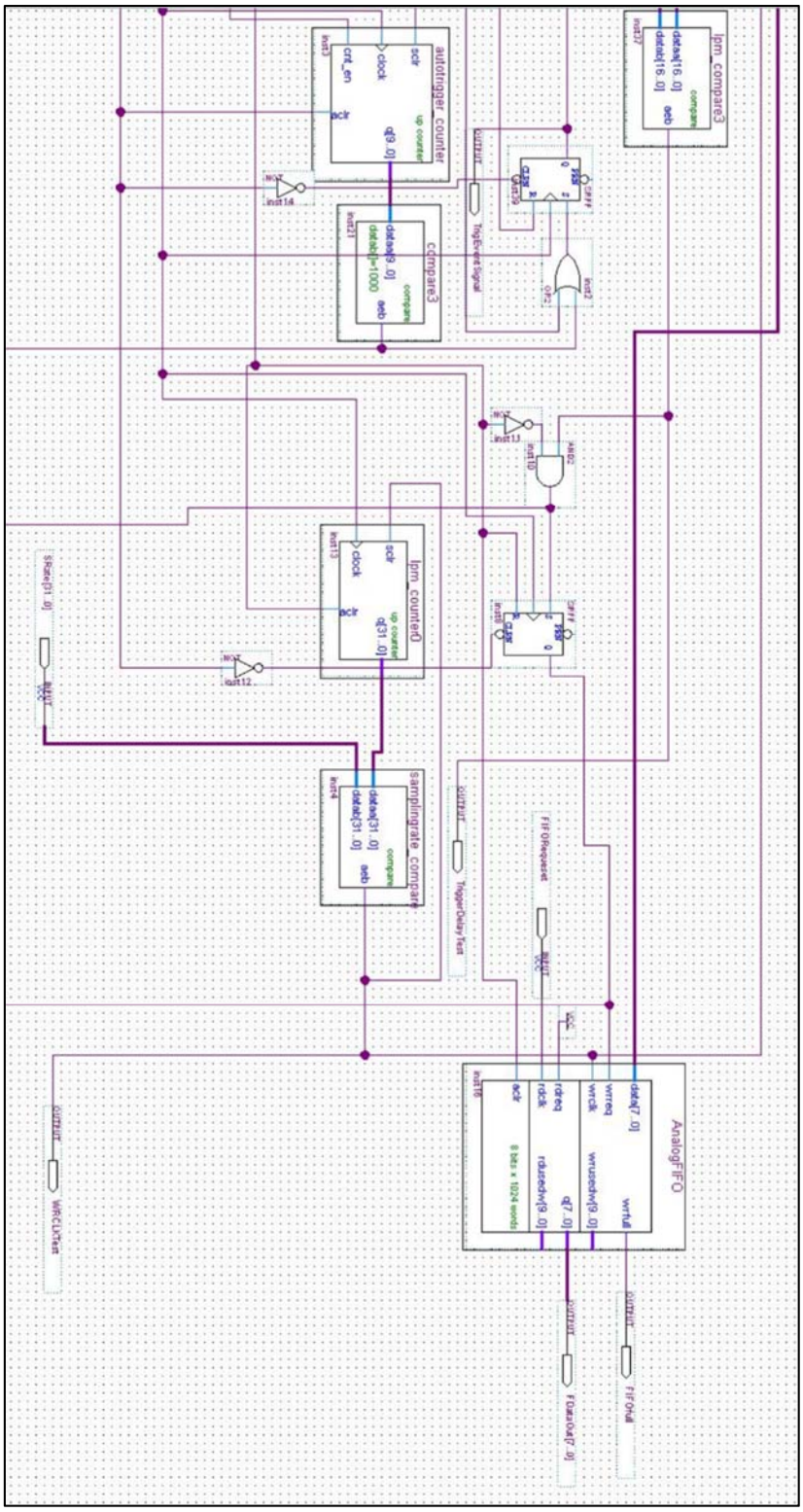
```
# Global Variables: None
# Input:            None
# Output:
# Error Handling:   None
# Algorithms:       None
# Data Structures:  None
# Registers Changed: r10, r11, r4, r5
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
start_sample:
    movhi   r10, %hi(PIO_0_BASE)        #get the triggerenable signal address
                                        #high bits in r10
    ori     r10, r10, %lo(PIO_0_BASE)   #add in the low bits to get the full
                                        #address
    movhi   r11, %hi(PIO_2_BASE)        #get the autotrigger signal address
                                        #high bits in r11
    ori     r11, r11, %lo(PIO_2_BASE)   #add in the low bits to get the full
                                        #address
    stwio   r4, 0(r11)                  #write the auto trigger value given
                                        #in the argument to the correct
                                        #location in memory
    movi    r5, 1                       #prepare to activate the trigger
                                        #enable signal
    stwio   r5, 0(r10)                  #activate trigger enable by
                                        #asserting it high
    ret                                 #return
```

Due to the logic circuitry shown below, simply activating the trigger enable signal allows the logic to take care of the trigger event signal generation while accounting for autotrigger.

The output of the trigger event and autotrigger portion of the logic is then fed into the trigger delay circuitry, composed of the TriggerDelay[16..0] input, lpm_counter1, and lpm_compare3. After being latched by an SRFF (inst8), this signal is fed into the wrreq input port of the internal FIFO, labeled AnalogFIFO in the second figure in the series. The FIFO begins filling itself when the wrreq signal is asserted; the rate at which this occurs is determined by the input to the wrclk port. The wrclk comes from the system sample rate, which is determined by the sample rate input (SRate[31..0]) and a clock divider composed of lpm_counter0 and the comparator samplingrate_compare.



Analog Block Logic Diagram (1): *The logic that comprises the triggering/sampling block and the FIFO. The block communicates with the ADC and CPU.*



Analog Block Logic Diagram (2): *The logic that comprises the triggering/sampling block and the FIFO. The block communicates with the ADC and CPU.*

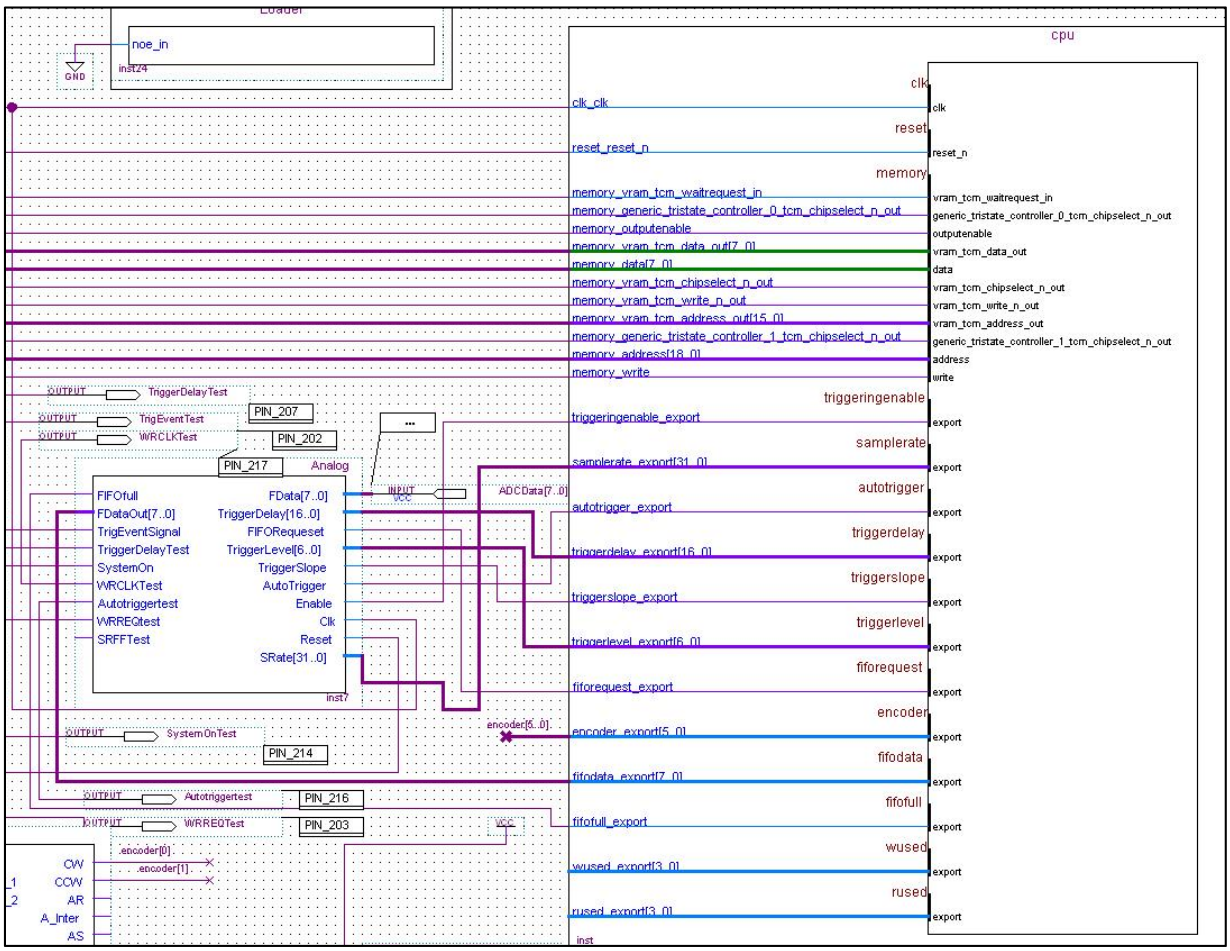
The analog logic block communicates sample data to the CPU through the FDataOut[7..0] and FIFOFull pins, which connect to the q[7..0] and wrfull output pins on the FIFO. The transfer of data out of the FIFO through the q[7..0] port is controlled through the rdclk input port to the FIFO, which is connected to the FIFORequest input pin. These features are exploited by the sample_done() function in the Analog.S file, the code for which is shown below.

```
# unsigned char far *sample_done(void)
# Description:      Function checks if the current sample is done being
sampled
#                   If it is, then the function returns a pointer to the
#                   sampled data. If not, then the function will return null
# Operation:        The function first reads the value of the FIFOFull output
#                   in the CPU (located at PIO 9). If the FIFOFull signal
#                   is active, then the function starts to loop through the
#                   FIFO data. For each loop, the function asserts the
#                   FIFORequest signal in the CPU (located at PIO 6) high and
#                   stores the value that is returned at the FDataOut[7..0]
#                   port in the CPU (located at PIO 8) in an array. Process
#                   is repeated until  array is full and FIFO is empty
#                   The function then returns the address of this array.
#                   The function then turns off the analog logic block by
#                   asserting the trigger enable signal low.
#                   If sample was not done being sampled, then function
#                   simply returns null.
# Arguments:        None
# Return Value:     unsigned char far* - pointer to the array containing
#                   the sample data (sample_data_array)
# Local Variables:  None
# Shared Variables: None
# Global Variables: None
# Input:            FIFOFull port on the CPU is read
# Output:           FDataOut[7..0] port and FIFORequest ports on the CPU are
#                   updated
# Error Handling:   None
# Algorithms:       None
# Data Structures:  sample_data_array - array that will hold the sample data
# Registers Changed: r10, r12, r11, r2, r9, r8, r13, r7
# Limitations:      None
# Author:           Siva Gangavarapu
# Last Modified:    8/8/16
sample_done:
    movhi   r10, %hi(PIO_9_BASE)        #get the address for PIO 9, which is
                                        #where the FIFOFull value is stored.
                                        #Get high bits first in r10
    ori     r10, r10, %lo(PIO_9_BASE)   #Get the full address by getting
                                        #the low bits as well
    ldwio   r11, 0(r10)                 #get the FIFOFull value in r11
    xor     r12, r12, r12               #make sure r12 is clear (especially
                                        #high bits)
    movi    r12, 1                      #prepare to check if FIFOFull value
                                        #is logic high (active)
    beq     r11, r12, sample_done_true  #if it is, then jump accordingly

sample_done_false:
    movi    r2, 0                       #if FIFOFull is low (sample is not
```

```
                                        #done sampling since the FIFO isn't
                                        #full yet, then return NULL)
    jmpi    sample_done_end             #end the function, with NULL being
                                        #returned

sample_done_true:                       #here we want to loop through FIFO
                                        #and collect all of our sample data
                                        #points, since sample is done being
                                        #sampled

    movhi   r10, %hi(PIO_6_BASE)        #Asserting the FIFORequest signal
                                        #means that the FIFO will empty
                                        #one data point and send it to the
                                        #CPU at a different PIO. Asserting it
                                        #low, then high again, will send out
                                        #the next data point. To exploit this
                                        #first we get the FIFO Request signal
                                        #address, located in PIO 6 in the
                                        #CPU, high bits first
    ori     r10, r10, %lo(PIO_6_BASE)   #Get the full FIFORequest signal
                                        #location address
    movi    r9, 0                       #initialize the array counter to the
                                        #beginning of the array. Counter
                                        #will be stored in r9

sample_done_loop:
    movia   r8, sample_data_array       #store the start of the array in r8
    movi    r12, 1                      #prepare to pulse FIFORequest so that
                                        #we can pull out a data point
    stwio   r12, 0(r10)                 #assert FIFORequest high (pulse it)
    movhi   r11, %hi(PIO_8_BASE)        #get the base address of PIO 8, where
                                        #data from the FIFO comes out. Get
                                        #the high bits of the address first
    ori     r11, r11, %lo(PIO_8_BASE)   #get the low bits of the address to
                                        #get the whole thing and store it in
                                        #r11
    add     r8, r8, r9                  #get the correct location in the
array
                                        #to store the data point
    ldbio   r13, 0(r11)                 #store the sample (8 bits) data in
r13
    stbio   r13, 0(r8)                  #store the data in r13 in the array
    addi    r9, r9, 1                   #increment the array counter to
                                        #prepare for the next data point
    movi    r7, 1024                    #prepare to check if we have finished
                                        #looking at the entire FIFO. The
                                        #size of the FIFO is 1024 data points
    stwio   r0, 0(r10)                  #pulse FIFORequest low to prepare for
                                        #the next data request
    beq     r9, r7, sample_done_return_array #check if we're done with all
                                             #the data points
    jmpi    sample_done_loop            #if not, then keep looping until
                                        #all the sample data is stored in
                                        #the array

sample_done_return_array:
    movhi   r10, %hi(PIO_0_BASE)        #get the address for PIO 0 high bits
```

```
                                        #in r10. PIO 0 gives the address
                                        #where trigger enable is located
    ori     r10, r10, %lo(PIO_0_BASE)   #complete the base address in r10
    movi    r9, 0                       #prepare to assert trigger enable
                                        #low
    stwio   r9, 0(r10)                  #assert trigger enable low since
                                        #we are done collecting sample done
    movia   r2, sample_data_array       #return the starting address of the
                                        #array containing the sampling data

sample_done_end:
    ret                                 #return
```

Finally, the relationship between the analog block and the rest of the system is shown in the diagram below, ***Main Block: Analog Block and CPU***. A full pin list describing the I/O pin on component U5 corresponding to each I/O pin in the block diagram is found in ***Pin List Table***



Main Block: Analog Block and CPU*: Pins ending in "Test" can be ignored, as they are only included for optional debugging purposes.*

Input and output pins on the Analog block are connected to their respective PIOs on the CPU (see ***CPU Port, Component, and Target Block Table***). Clk and Reset pins are connected to the system clock and system reset inputs, respectively.

Pin Name	Target Description	I/O Pin
ADD[18]	ROM/SRAM Address	PIN_73
ADD[17]	ROM/SRAM Address	PIN_72
ADD[16]	ROM/SRAM Address	PIN_70
ADD[15]	ROM/SRAM Address	PIN_69
ADD[14]	ROM/SRAM Address	PIN_80
ADD[13]	ROM/SRAM Address	PIN_118
ADD[12]	ROM/SRAM Address	PIN_117
ADD[11]	ROM/SRAM Address	PIN_114
ADD[10]	ROM/SRAM Address	PIN_113
ADD[9]	ROM/SRAM Address	PIN_112
ADD[8]	ROM/SRAM Address	PIN_111
ADD[7]	ROM/SRAM Address	PIN_110
ADD[6]	ROM/SRAM Address	PIN_109
ADD[5]	ROM/SRAM Address	PIN_108
ADD[4]	ROM/SRAM Address	PIN_107
ADD[3]	ROM/SRAM Address	PIN_106
ADD[2]	ROM/SRAM Address	PIN_103
ADD[1]	ROM/SRAM Address	PIN_100
ADD[0]	ROM/SRAM Address	PIN_99
CS1	ROM Chip select	PIN_64
CS0	SRAM Chip Select	PIN_83
DATA[7]	ROM/SRAM Data	PIN_139
DATA[6]	ROM/SRAM Data	PIN_137
DATA[5]	ROM/SRAM Data	PIN_135
DATA[4]	ROM/SRAM Data	PIN_134
DATA[3]	ROM/SRAM Data	PIN_133
DATA[2]	ROM/SRAM Data	PIN_131
DATA[1]	ROM/SRAM Data	PIN_128
DATA[0]	ROM/SRAM Data	PIN_127
OE	ROM/SRAM OE	PIN_82
write	ROM/SRAM WE	PIN_71
U6DIR1	U6 Bank 1 Direction Pin	PIN_95
U6DIR2	U6 Bank 2 Direction Pin	PIN_98
U8DIR1	U8 Bank 1 Direction Pin	PIN_119
U8DIR2	U8 Bank 2 Direction Pin	PIN_120
U10DIR1	U10 Bank 1 Direction Pin	PIN_126
U23DIR2	U23 Bank 2 Direction Pin	PIN_52
CLK	System Clock Input	PIN_31
U1DIR1	U1 Bank 1 Direction Pin	PIN_201
U3DIR2	U3 Bank 2 Direction Pin	PIN_50
SW3_A	SW3 Output 1	PIN_234
SW3_B	SW3 Output 2	PIN_233

SW2_A	SW2 Output 1	PIN_235
SW2_B	SW2 Output 2	PIN_39
SW4_A	SW4 Output 1	PIN_44
SW4_B	SW4 Output 2	PIN_45
U7DIR1	U7 Bank 1 Direction Pin	PIN_169
U7DIR2	U7 Bank 2 Direction Pin	PIN_197
U10DIR2	U10 Bank 2 Direction Pin	PIN_167
VRAMADD[7]	VRAM Address	PIN_166
VRAMADD[6]	VRAM Address	PIN_161
VRAMADD[5]	VRAM Address	PIN_160
VRAMADD[4]	VRAM Address	PIN_148
VRAMADD[3]	VRAM Address	PIN_147
VRAMADD[2]	VRAM Address	PIN_146
VRAMADD[1]	VRAM Address	PIN_143
VRAMADD[0]	VRAM Address	PIN_142
VRAMData[7]	VRAM Data	PIN_196
VRAMData[6]	VRAM Data	PIN_195
VRAMData[5]	VRAM Data	PIN_194
VRAMData[4]	VRAM Data	PIN_189
VRAMData[3]	VRAM Data	PIN_188
VRAMData[2]	VRAM Data	PIN_187
VRAMData[1]	VRAM Data	PIN_186
VRAMData[0]	VRAM Data	PIN_185
VRAMOE	VRAM Data	PIN_182
VRAMWE	VRAM Write Enable	PIN_183
RAS	VRAM RAS	PIN_177
CAS	VRAM CAS	PIN_181
VSYNCOut	VSYNC to LCD	PIN_173
VCLOCKOut	VRAM Serial Clock	PIN_78
SOE	VRAM Serial Output Enable	PIN_76
HSYNCOut	HSYNC to LCD	PIN_171
DCLKOut	DCLK to LCD	PIN_184
DisplayDE	Data Enable to LCD	PIN_176
ADCData[7]	ADC Data	PIN_37
ADCData[6]	ADC Data	PIN_22
ADCData[5]	ADC Data	PIN_21
ADCData[4]	ADC Data	PIN_18
ADCData[3]	ADC Data	PIN_13
ADCData[2]	ADC Data	PIN_9
ADCData[1]	ADC Data	PIN_6
ADCData[0]	ADC Data	PIN_5
U4DIR2	U4 Bank 2 Direction Pin	PIN_68
CLKADC	ADC Clock	PIN_224

U4DIR1	U4 Bank 1 Direction Pin	PIN_51
U23DIR1	U23 Bank 1 Direction Pin	PIN_55

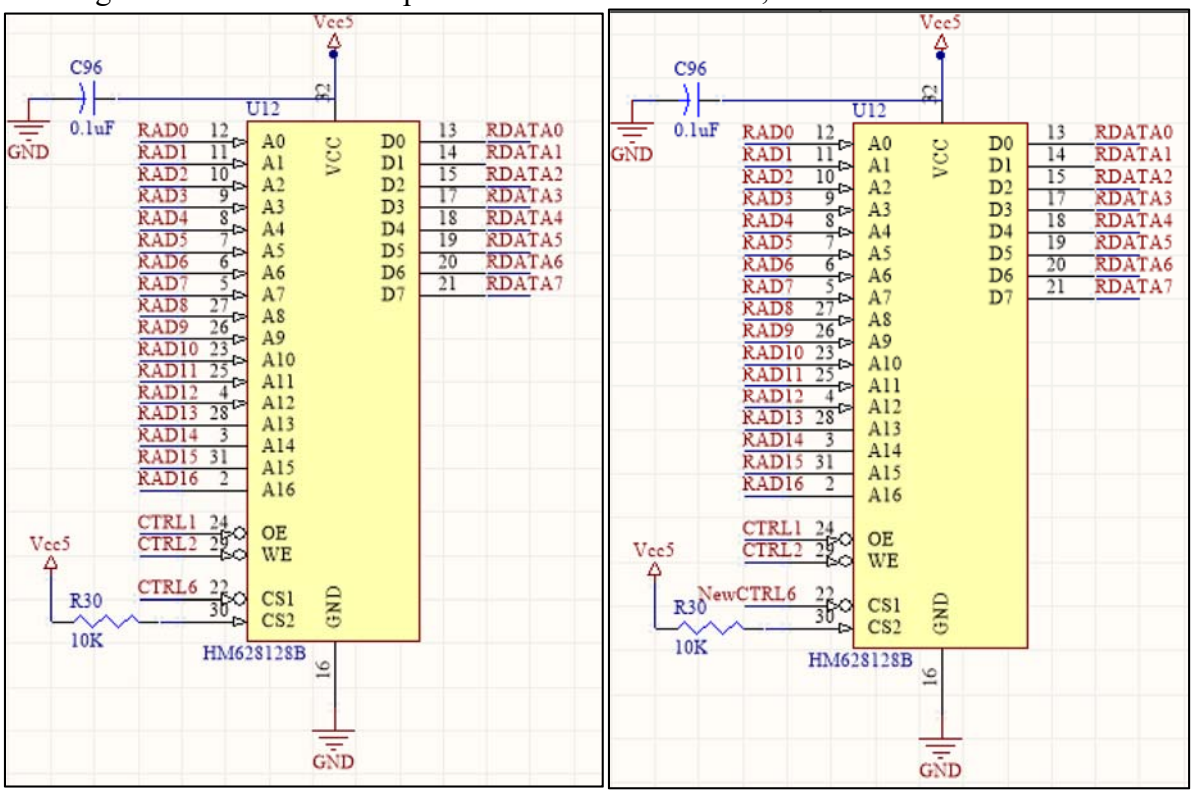
Pin List Table*: Table matching the pins found in the logic block diagrams (in the software technical manual) with the FPGA (component U5) I/O pins found in the hardware technical manual. Diagrams illustrating the location of these pins on U5 are found in the* ***FPGA and Related Circuitry*** *portion of the hardware technical manual*

Design Changes

This section of the documentation highlights changes between the original system design and the final product. The documentation thus far also references appropriate sections in this part of the documentation.

Design Changes for U12

The original schematic for component U12 is shown below, to the left of the finalized version:

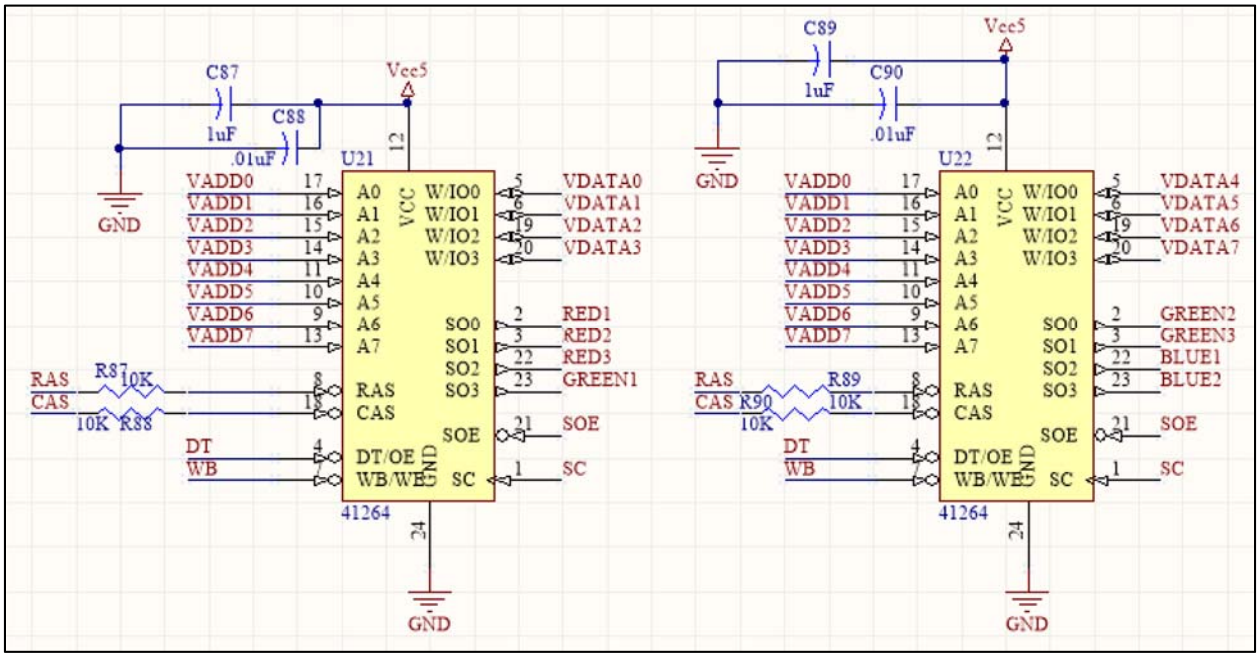


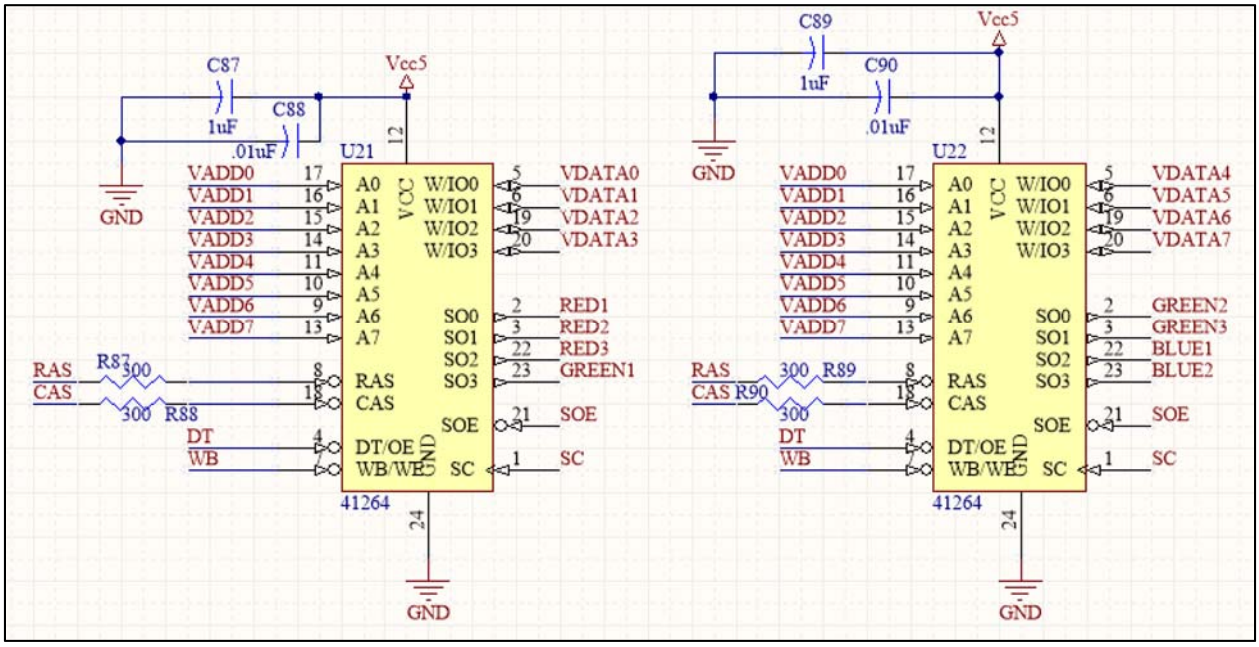
Schematic Changes for U12: *Original is shown on the left, updated on the right.*

Component U6, the buffer for the CTRL6 signal, the original CS1 signal for U12, experienced internal circuitry problems due to extensive shorting during board assembly. An unused buffered I/O signal, NewCTRL6, was then rerouted to act as the new CS1 signal for U12. The schematic for the buffer for NewCTRL6, component U23, can be found in the **ROM and SRAM** section in the hardware technical manual.

Design Changes for U21 and U22

The original schematic for U21 and U22 is shown below, on top of the finalized version:



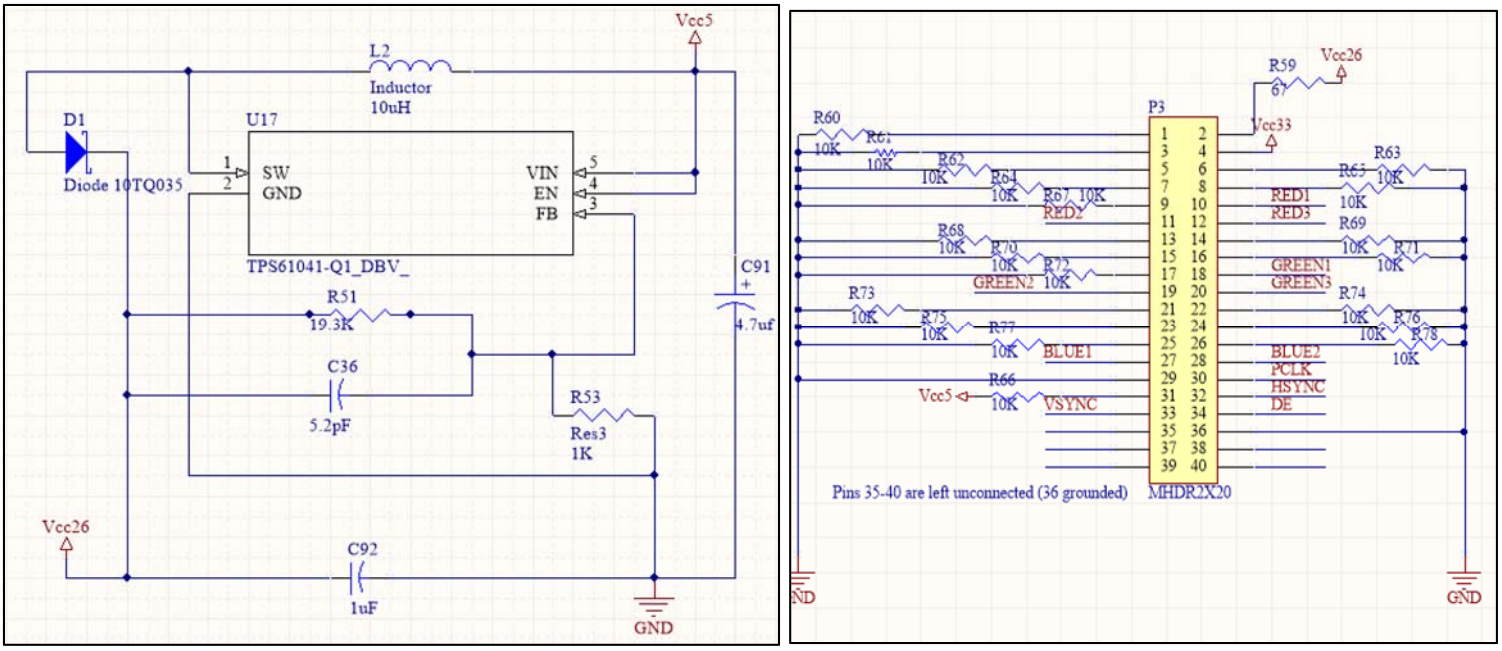


Schematic Changes for U21 and U22: *Original is shown on top, updated on the bottom.*

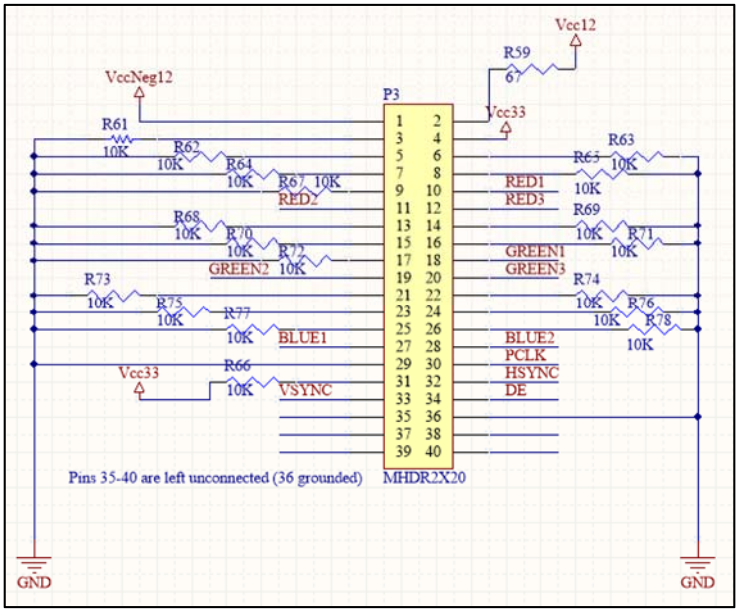
An error in the original design involved a 10 KOhm resistor in series with the RAS and CAS signals received by U21 and U22. Since this value is way too high to serve its purpose of providing signal integrity, the final design saw it being replaced by a much lower, 300 Ohm, resistor.

Design Changes for P3

The original design for component P3 used a separately generated 26-volt power line to provide power to the LCD. The schematic for P3, as well as the required boost converter power circuitry, is included below.



Original P3 and 26-volt generation circuitry schematics

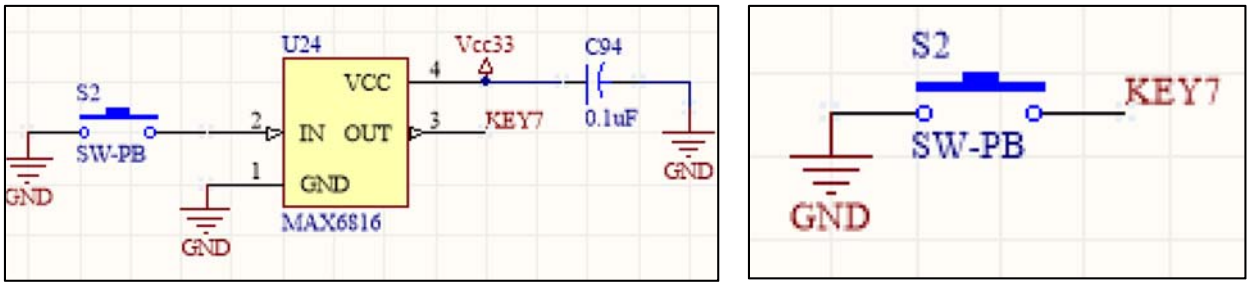


Updated P3 schematic

As seen in the schematic, the original design had pin 1 of P3 (the LED cathode pin) grounded while the LED anode pin received the power line. Due to a lack of parts, the boost converter circuitry was never constructed, and the -12V and 12V power lines where used to supply LED power for the LCD. The updated schematic is shown to the right. An updated boost converted schematic is not shown because those parts where simply left out of the board during assembly.

Design Changes for S2

The original schematic for component S2 is shown below, to the left of the finalized version



Schematic Changes for S2 and U24: *Original is shown to the left, updated to the right.*

As shown in the figure, the original design included an external MAX6816 debouncing chip (component U24) for S2, the menu button. Due to a lack of parts, debouncing was done in software, resulting the schematic shown on the right.