EE 52

Spring 2017

Digital Oscilloscope Documentation

Sophia Liu

Contents

1 User Manual

1.1 System Description

The System on Programmable Chip (SoPC) Digital Oscilloscope is an FPGA/microprocessor-based system capable of capturing and displaying up to 5 MHz signals. The analog input can range from 0 V to 3.8 V. The system has all the features of a standard oscilloscope with the exception of input signal scaling. Two rotary encoders are used to control the settings, and a LCD display is used to display the captured signal.

1.2 How to Use the System

The system will begin immediately upon powering up. [1] The system starts in one-shot trace mode with a sampling rate of 100 ns, a mid-level trigger (halfway between the minimum and maximum trigger levels, at 1.9 V) with no delay and positive slope, and with the menu displayed and scale set to axes. The probe can be attached to the desired input source. The oscilloscope settings are described in section 1.3. The reset button can be used to restart the system.

1.3 System Settings

1.3.1 Menu

The scope parameter menu is located in the upper right corner of the LCD and contains the following entries in order:

```
Mode
Scale
Sweep
Trigger
    Level
    Slope
    Delay
```

The menu display can be toggled on and off by pressing the menu rotary encoder. The user can cursor to any of the entries by turning the menu rotary encoder and can change a setting by turning the secondary rotary encoder. Changes take effect

[1] The serial configuration device currently does not work, so the FPGA must first be programmed through the JTAG debugger.

immediately. More rotary details can be found in section 1.3.2. The menu entries are described in more detail below.

Mode : The Mode menu entry can be set to Mode Normal, Mode Automatic, or Mode One-Shot. In Mode Normal the scope waits for another trigger after every retrace. In this mode, new traces are captured as fast as the scope can redraw the screen. Mode Automatic works the same as Mode Normal if there are trigger events. But if no trigger event occurs after a specified delay (typically significantly longer than the time represented by a screen of data) the scope triggers automatically without a trigger event occurring. In Mode One-Shot the scope triggers only once and then holds that trace on the screen. It does not look for another trigger event until the Trigger menu item is selected and secondary rotary encoder is turned.

Scale : The Scale menu entry can be set to Scale Axes, Scale Grid, or Scale Off. If the scale is set to Scale Axes, the x and y axes are displayed along with the trace. If the scale is set to Scale Grid, an x-y grid is displayed along with the trace. If the scale is set to Scale Off, no axes or grid are displayed.

Sweep : The Sweep menu entry sets the sweep rate (in time per sample) for the scope. Possible settings are: 100, 200, and 500 nanoseconds, and 1, 2, 5, 10, 20, 50, 100, 200, and 500 microseconds, and 1, 2, 5, 10, and 20 milliseconds (per sample).

Trigger : The Trigger menu entry re-arms the trigger for the scope in one-shot mode. Any time it is selected and the secondary rotary encoder is turned the scope trigger is re-armed and a new trace will then be captured once the trigger conditions (level and slope) are met.

Level : The Level menu entry sets the trigger level. It can be set to any value from the most negative input voltage to the most positive in 128 steps. Additionally, the trigger level is displayed as a line on the screen when the trigger level is being changed.

Slope : The Slope menu entry is either Slope + or Slope - and determines whether the scope is triggered on a positive or negative slope respectively.

Delay : The Delay menu entry determines the trigger delay. It sets the time after the trigger event at which the trace will start. It may be set to any value from the minimum delay to the maximum delay times the sample rate and it is displayed as a time.

1.3.2 Rotary Encoder Functionality

The user can change the scope configuration via two rotary encoders (seen in **??**). All of the scope parameters are set via an on-screen menu. The rotary actions are detailed below:

Press encoder 1 : Turns the menu on/off. If the menu is off it is not displayed and turning the rotary encoders have no effect on the settings.

Turn encoder 1 CW : Moves the cursor down, if not already at the bottom menu item. If at the bottom, the cursor does not move.

Turn encoder 1 CCW : Moves the cursor up, if not already at the top menu item. If at the top, the cursor does not move.

Turn encoder 2 CW : Changes the currently selected (with cursor) menu item. Goes "forward" through the list of possible settings. If at the "end" of the list, doesn't change the current selection.

Turn encoder 2 CCW : Changes the currently selected (with cursor) menu item. Goes "backward" through the list of possible settings. If at the "beginning" of the list, doesn't change the current selection.

2 Technical Documentation

2.1 Hardware

2.1.1 Hardware System Overview

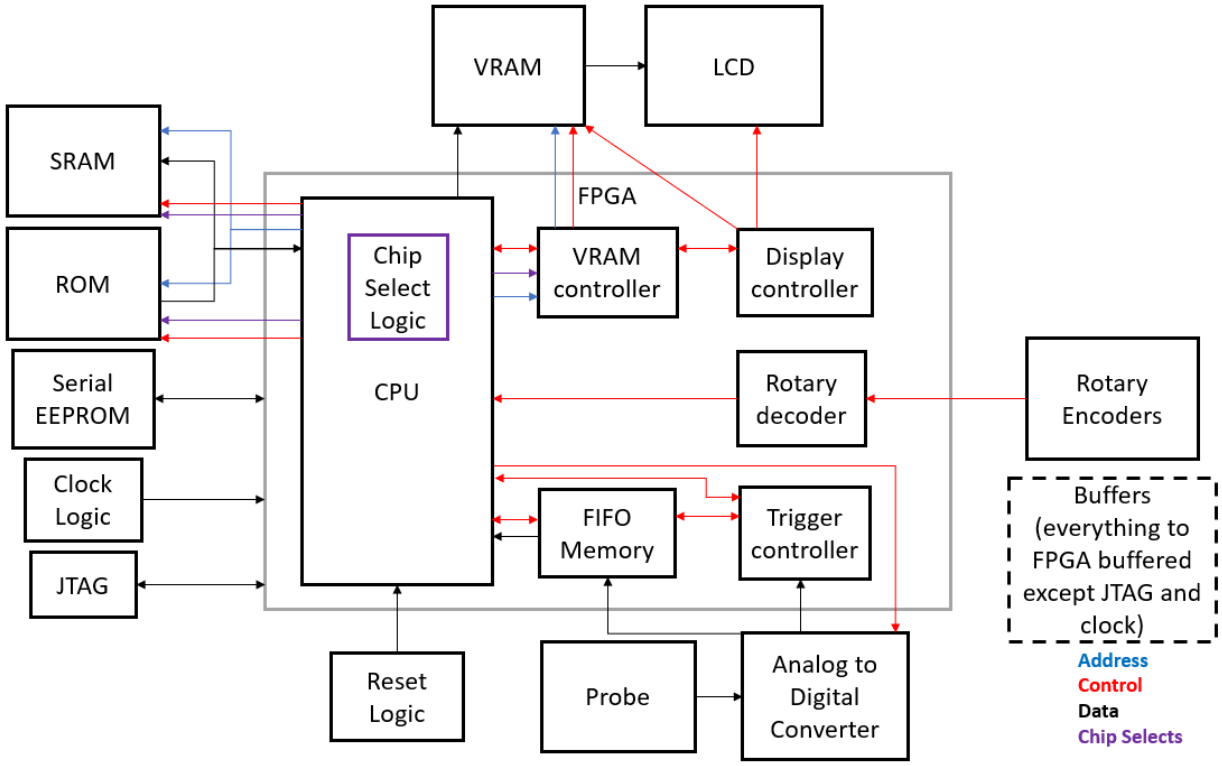

Figure 1: Block diagram overview of FPGA and peripheral chips.

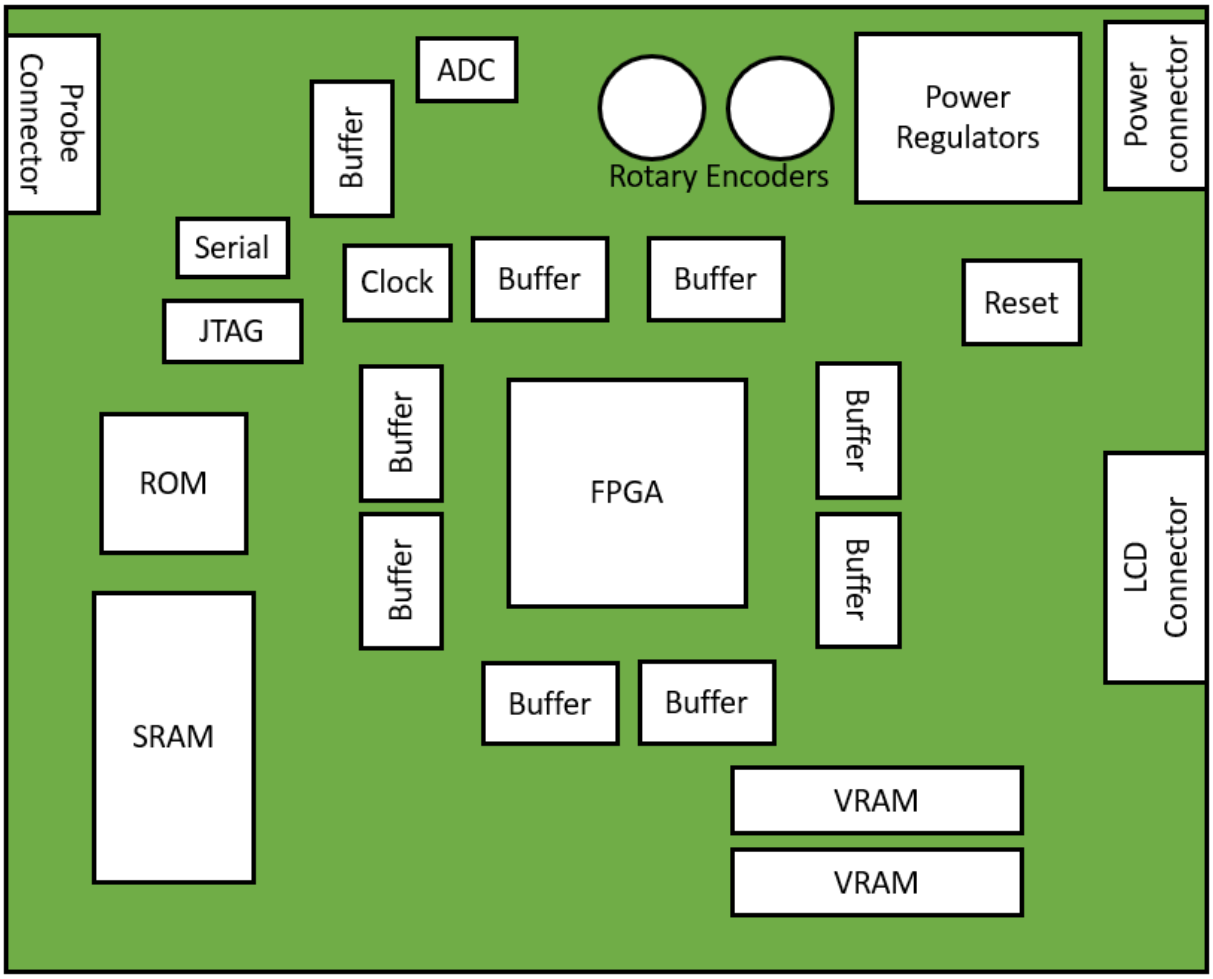

Figure 2: Board Layout.

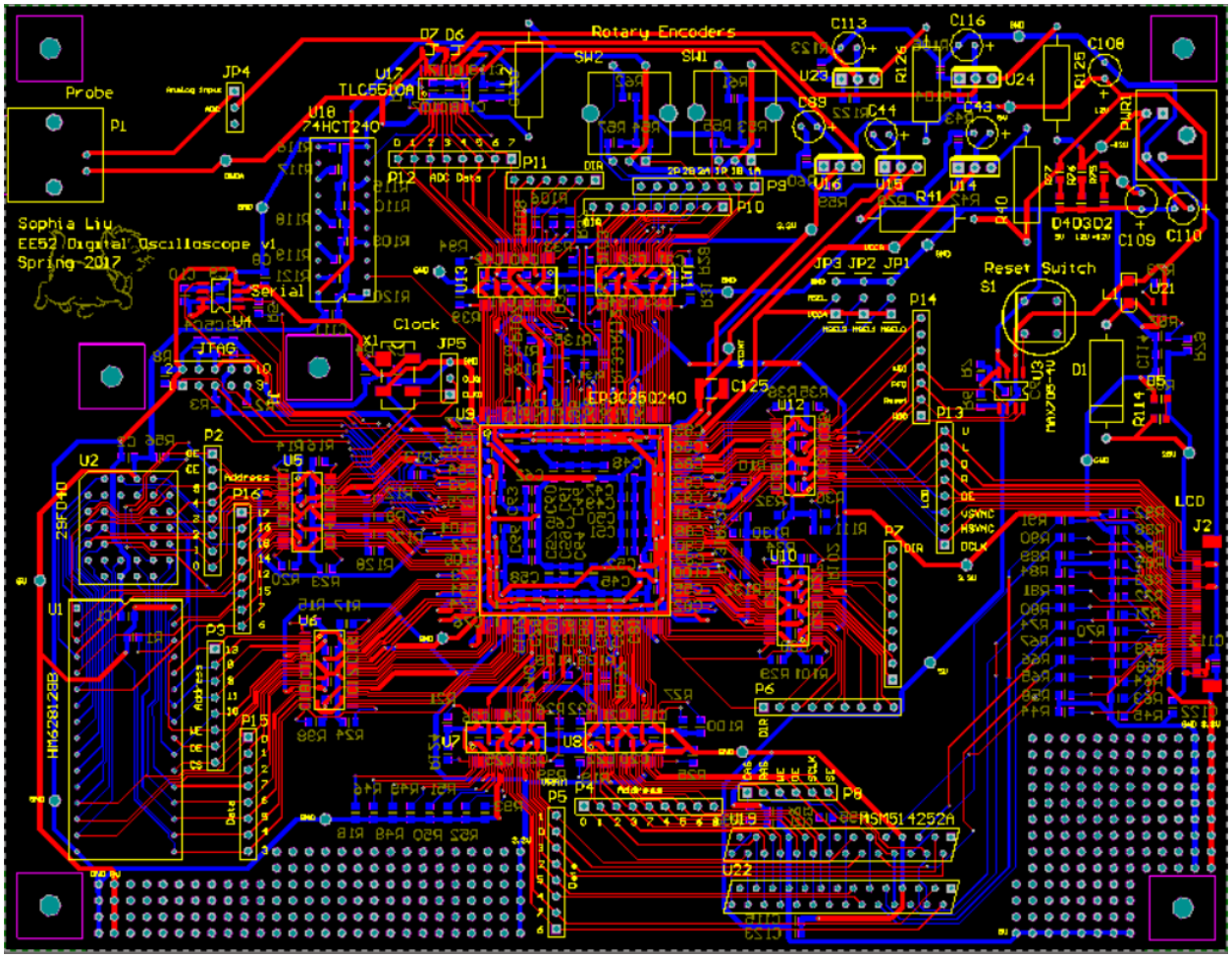

Figure 3: Image of PCB.

The oscilloscope system is centered around a Cyclone III FPGA, which includes a Nios II CPU and controllers for the VRAM, LCD, rotary encoders, and analog input and triggering. Upon startup, the bitmap is loaded from the serial EEPROM. The is then loaded from the ROM.

Receiving a Signal: When the probe is connected to a desired input, the analog signal is sent to an Analog to Digital Converter chip. 8 bits of data are then sent in parallel to the FPGA to an analog controller. The various trigger setting inputs (auto triggering, trigger enable, trigger slope, trigger level) are used along with the 8 bit signal to determine if the system should begin sampling. Once triggering has occurred, the signal is written to a FIFO buffer at the sampling rate. Once the FIFO is full, the CPU clocks out and stores the buffer data to be displayed.

Displaying a Signal: In order to display the signal on the LCD, data is written to the VRAM, which is then sent out to the LCD. Several control signals (write enable, chip select, address, wait) are sent from the CPU to the VRAM controller to perform read and write operations on the VRAM. An LCD controller sends out the necessary control signals to display the signal.

2.1.2 FPGA

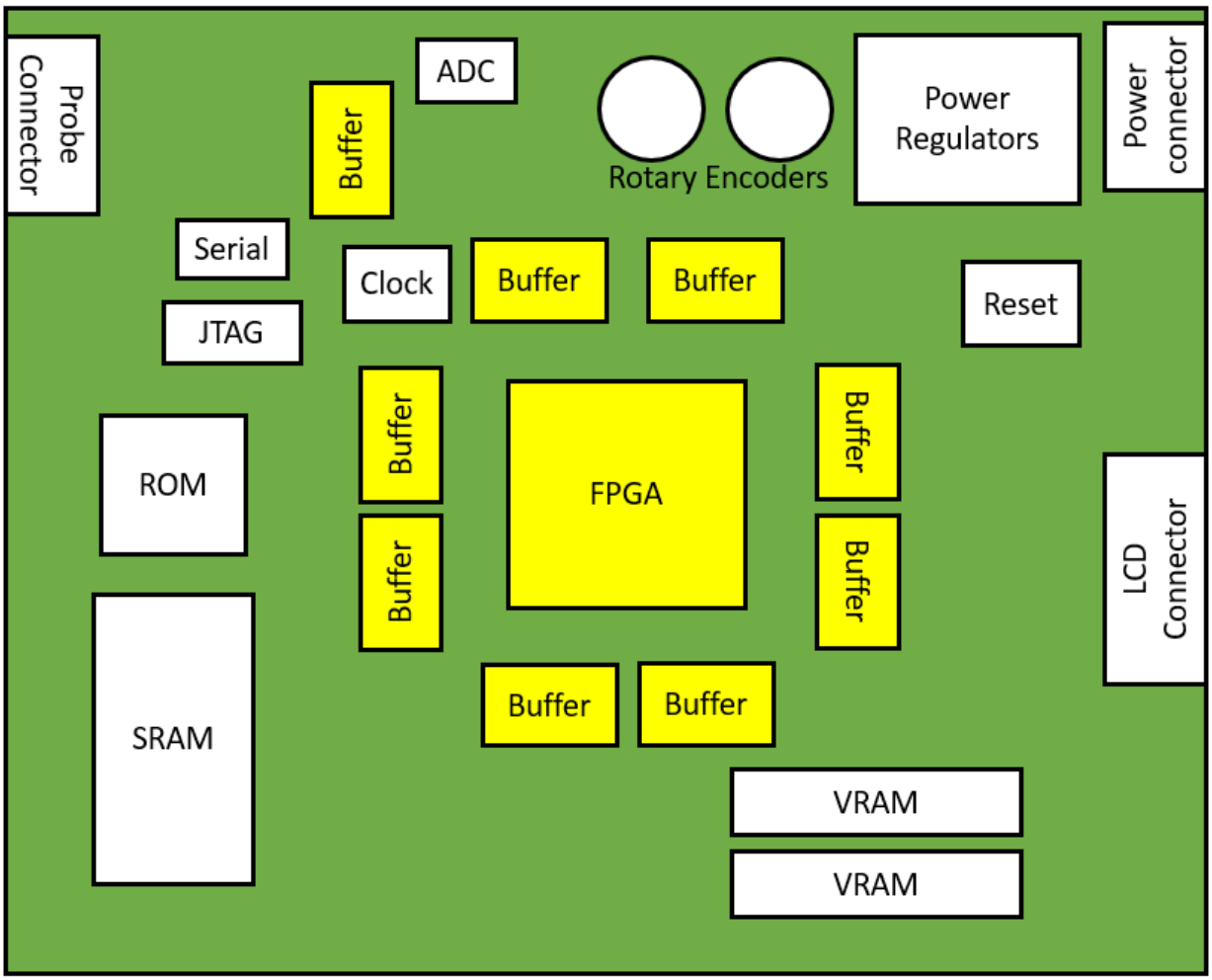

Figure 4: Board Layout: FPGA and buffers.

An Altera Cyclone III FPGA was used for all the logic and processing required (U9, EP3C25Q240). The three required voltage levels were generated with regulators, and each power pin was locally bypassed. The schematics can be seen in Figure **??**.

Buffers, or level shifters, were used to connect the buses and control signals from the peripheral chips to the FPGA to go between the low FPGA voltage and mainly 5V environment (U5, U6, U7, U8, U10, U12; 74LVT16245). A separate buffer was used to interface with the ADC (U17, TLC5510A), which had a larger input voltage range requirement (U18, SN74HCT240).

Pin Assignments: The pins were connected on the PCB as listed in Figure **??**.

The final product included several changes. Pin 162, an output for device configuration, was left floating, and the reset input was changed to Pin 187. One buffer was also left unusable, resulting in rewiring to extra I/O pins.

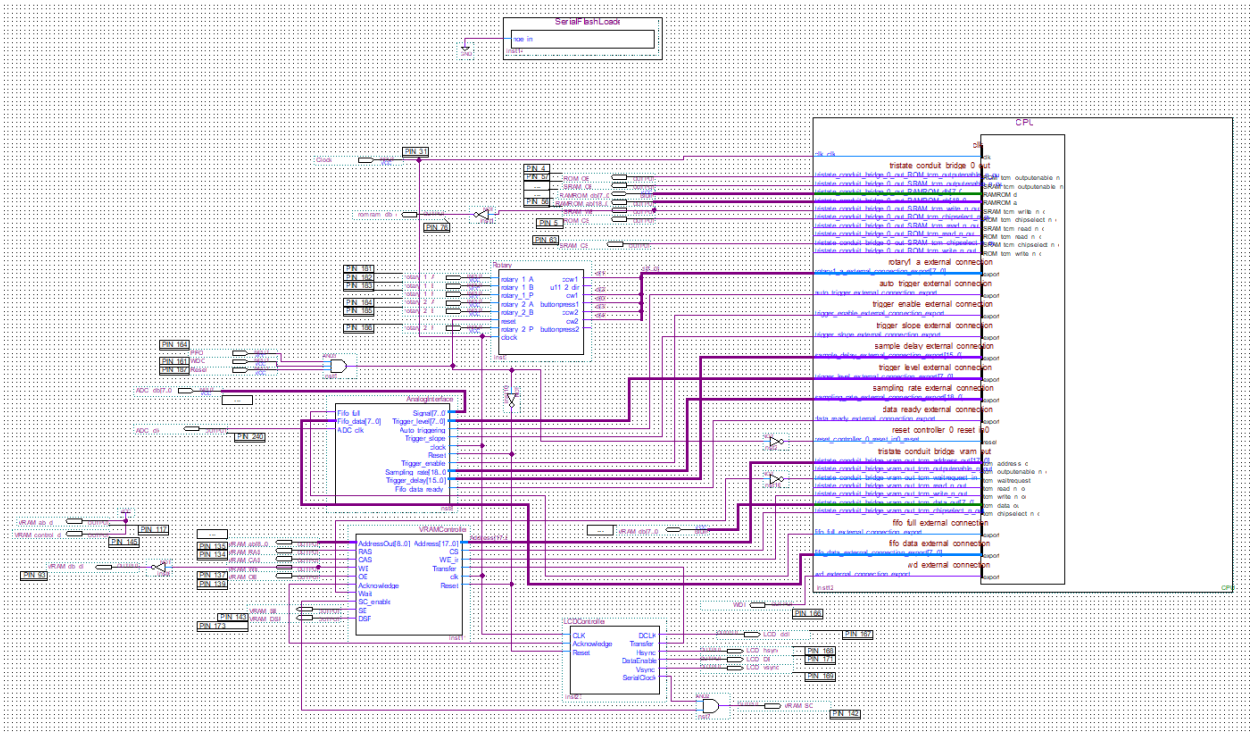

Figure 5: The main FPGA block diagram with the CPU, inputs and outputs, and controllers.

Logic: A CPU and several controllers were programmed into the FPGA, as seen in Figure 1. These will be discussed in greater detail in their respective subsections. As seen in the main block diagram, the peripheral memory, user interface elements, and other elements all interact with the FPGA and its various controllers. The input signal and settings are sent to the CPU and the appropriate controllers, and the output data is sent from the CPU to be displayed on the LCD.

2.1.3 Power

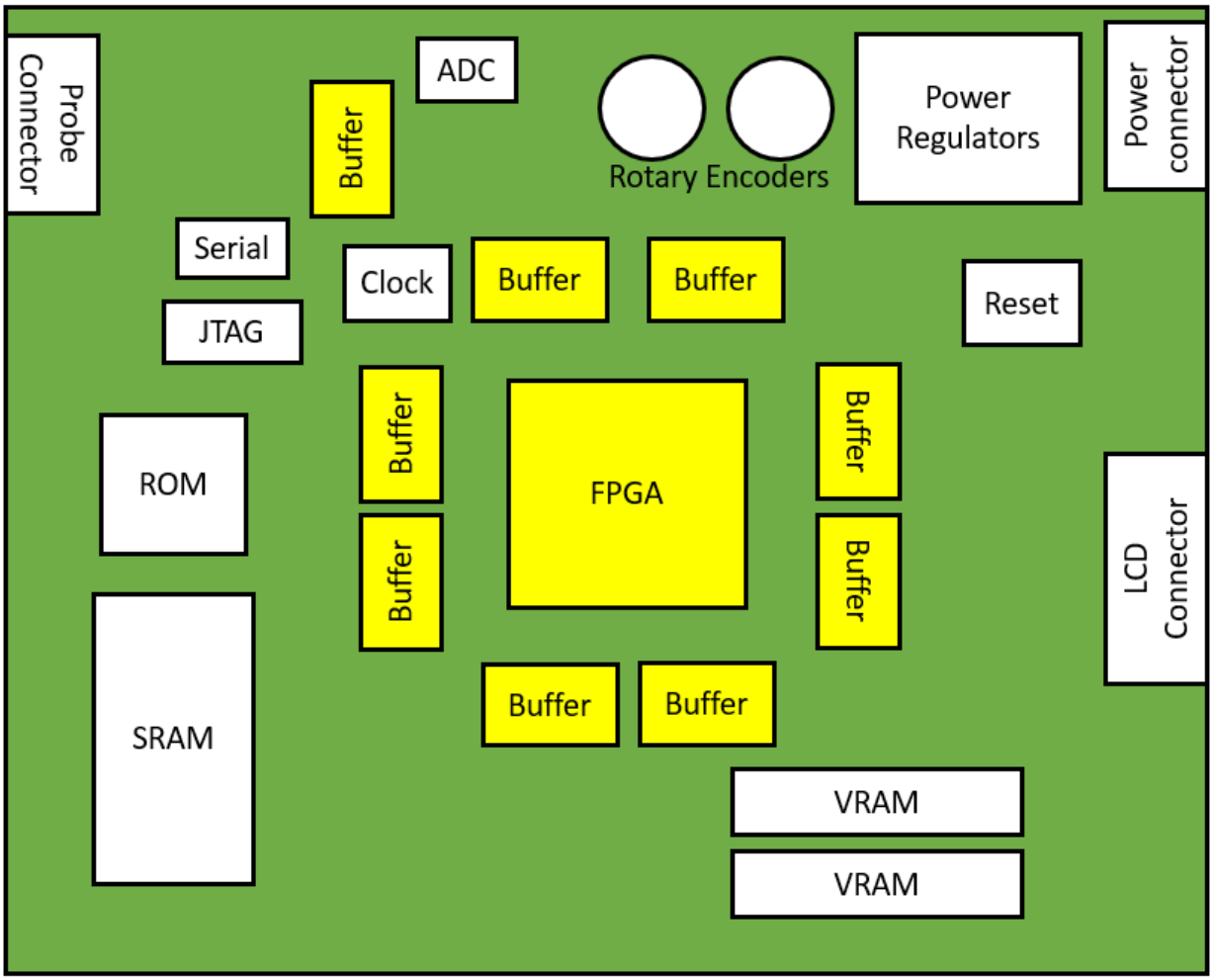

Figure 6: Board Layout: Power circuitry.

A din4 connector (PWR1) is used to supply the board with +/- 12 V and 5 V. All chips are locally bypassed.

5 V is used to power the reset, VRAM, SRAM, ROM, and ADC chips. Several LM1086 regulators are used to generate the other required voltages. Specifically, U14 uses the 5 V supply to generate 2.5 V for analog power to the FPGA PLLs, and U15 uses the 5 V supply to generate 1.2 V for FPGA power supply and digital power to the FPGA PLLs. U16 generates 3.3 V from 5 V, which is used to power the FPGA I/O supply pins, buffers, JTAG, serial configuration device, clock, and for pulling high the rotary encoder outputs.

Regulators also generate 5 V from 12 V (U24) and 4 V from 5 V (U23) for ADC reference voltages. The main power schematics can be seen in Figure **??**.

A boost converter is used to generate the 25 V necessary for the LCD backlight. U21 (AP3012) steps up from 5 V and outputs 25 V (Figure 7).

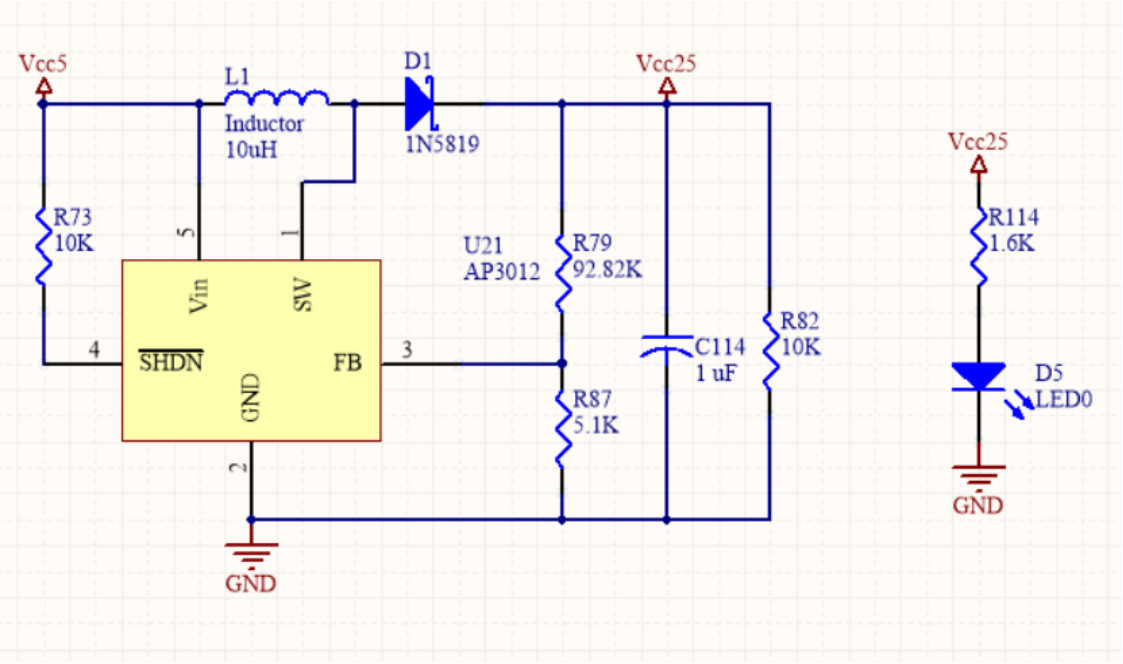

Figure 7: Boost converter circuit for LCD backlight power supply.

2.1.4 Memory

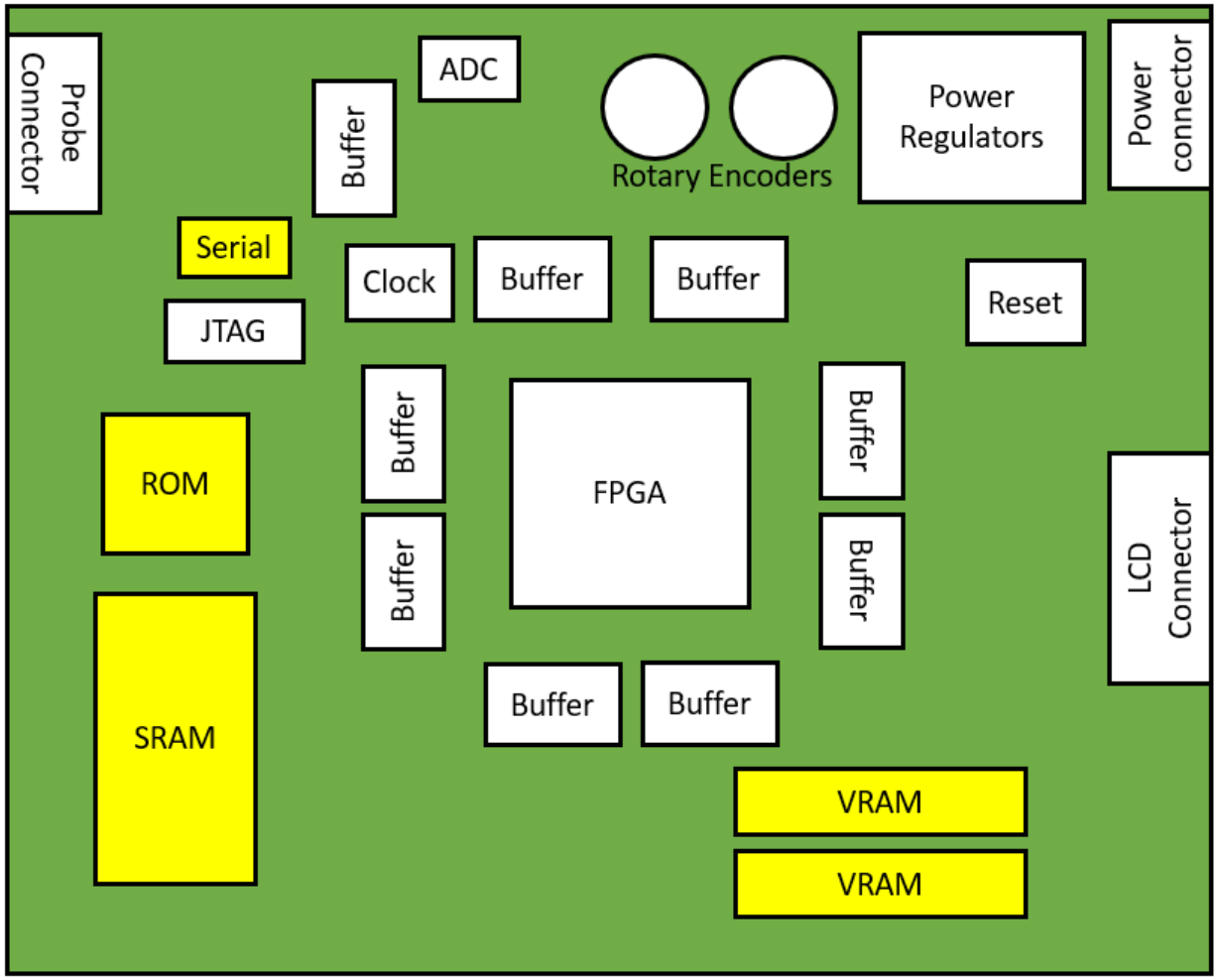

Figure 8: Board Layout: Memory.

Memory Map:

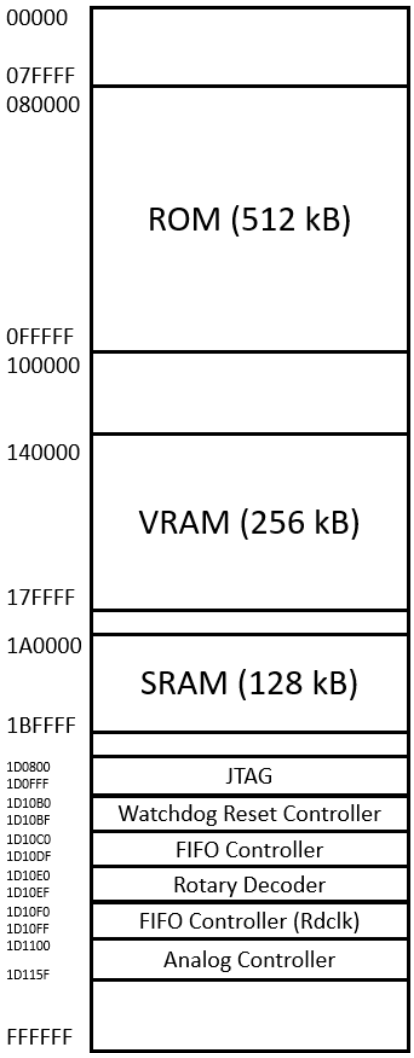

Figure 9: Memory map diagram

Several types of storage were used, including ROM (U2) for permanent program storage, and a smaller amount of static RAM (U1) for storage including the data segment, stack, and heap. Video RAM (U19, U22), with dynamic RAM, which is cheaper and smaller than static RAM, and serial memory, was used for the display memory. A serial device (U4) was also used to store configuration data for the FPGA after powering on.

ROM: 1 512 K x 8 bit EEROM (U2, Am29F040) was used for program storage. 8 data bits are connected in parallel through the buffers to the CPU, along with 19 address bits, an active-low chip enable (CE), and output enable (OE) signal. The write enable (WE)

signal was pulled up, as the ROM was written to with a dedicated ROM programmer.

The ROM timing diagram for the read cycle can be seen at Appendix A, Figure 42. 3 read wait states were used.

The executable file from the Nios II IDE was loaded into the ROM using a ROM programmer.

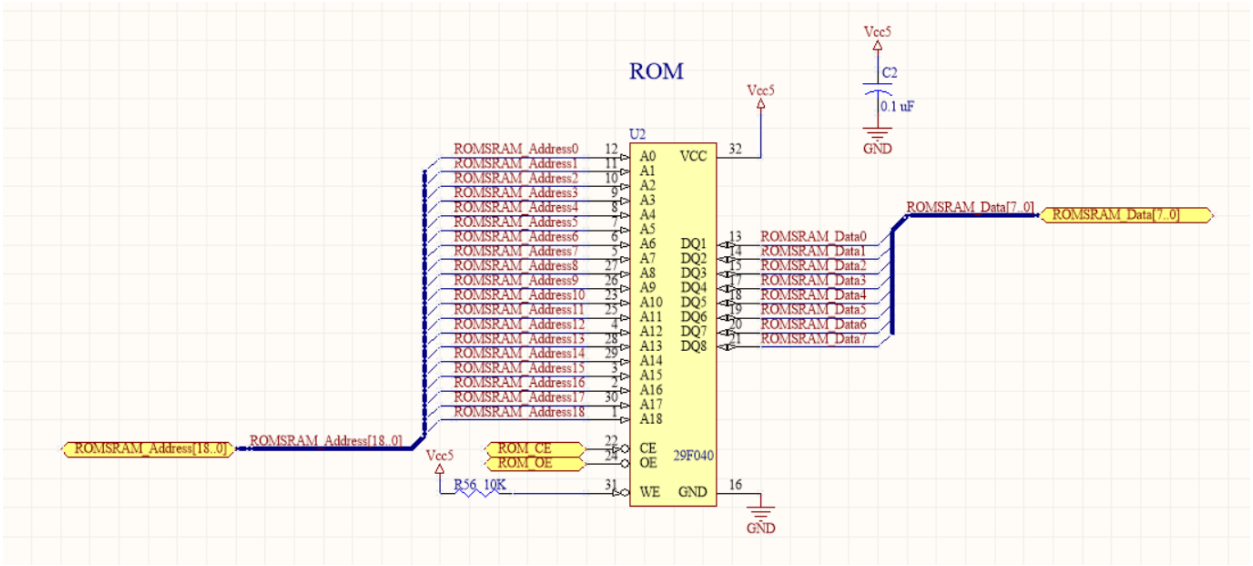

Figure 10: ROM Schematic.

SRAM: 1 128 K x 8 bit CMOS static RAM was used (U1, HM628128B)

The address line is shared between the ROM and SRAM, with only the 17 least significant bits used for RAM addressing. The 8-bit data bus is also shared between the ROM and SRAM. The active-low output enable (OE), write enable (WE), and chip select (CS1) are also connected between the SRAM and CPU, with the second active high chip select (CS2) pulled high as required by the read and write cycles.

The SRAM timing diagrams can be seen at Appendix A, Figures 43 and 44. 2 read wait states and 1 write wait state was used.

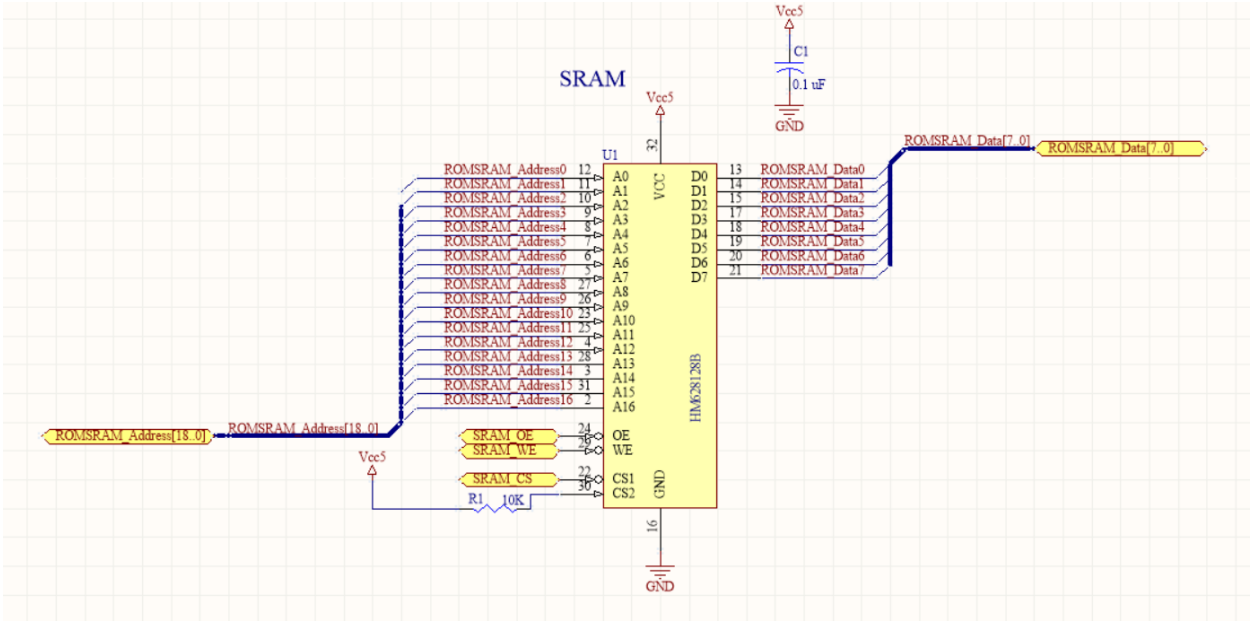

Figure 11: SRAM Schematic.

VRAM: See Section 2.1.7 for details on the VRAM.

Serial Device: A serial configuration device (U4, EPCS16), a 2 MB flash memory device, was used to store FPGA configuration data [2].

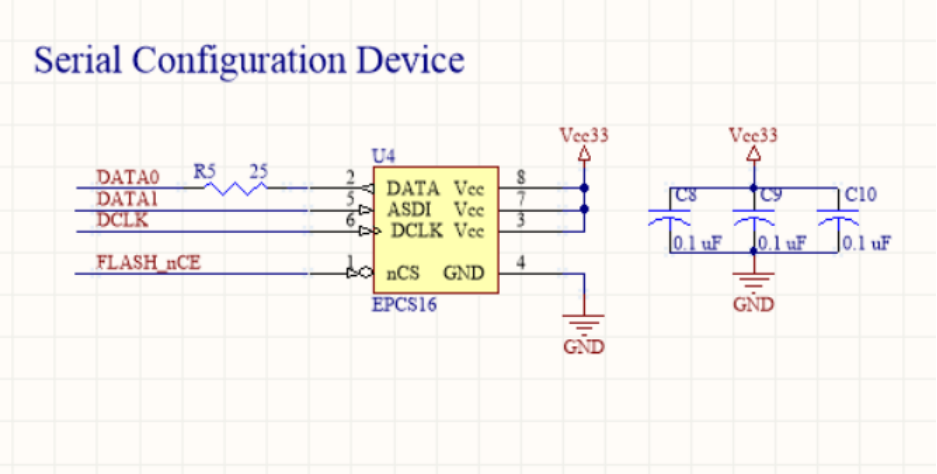

Figure 12: Serial Device Schematic.

[2] Was unable to successfully use serial device for unknown reasons, possibly due to incorrect FPGA configurations.

2.1.5 Analog System

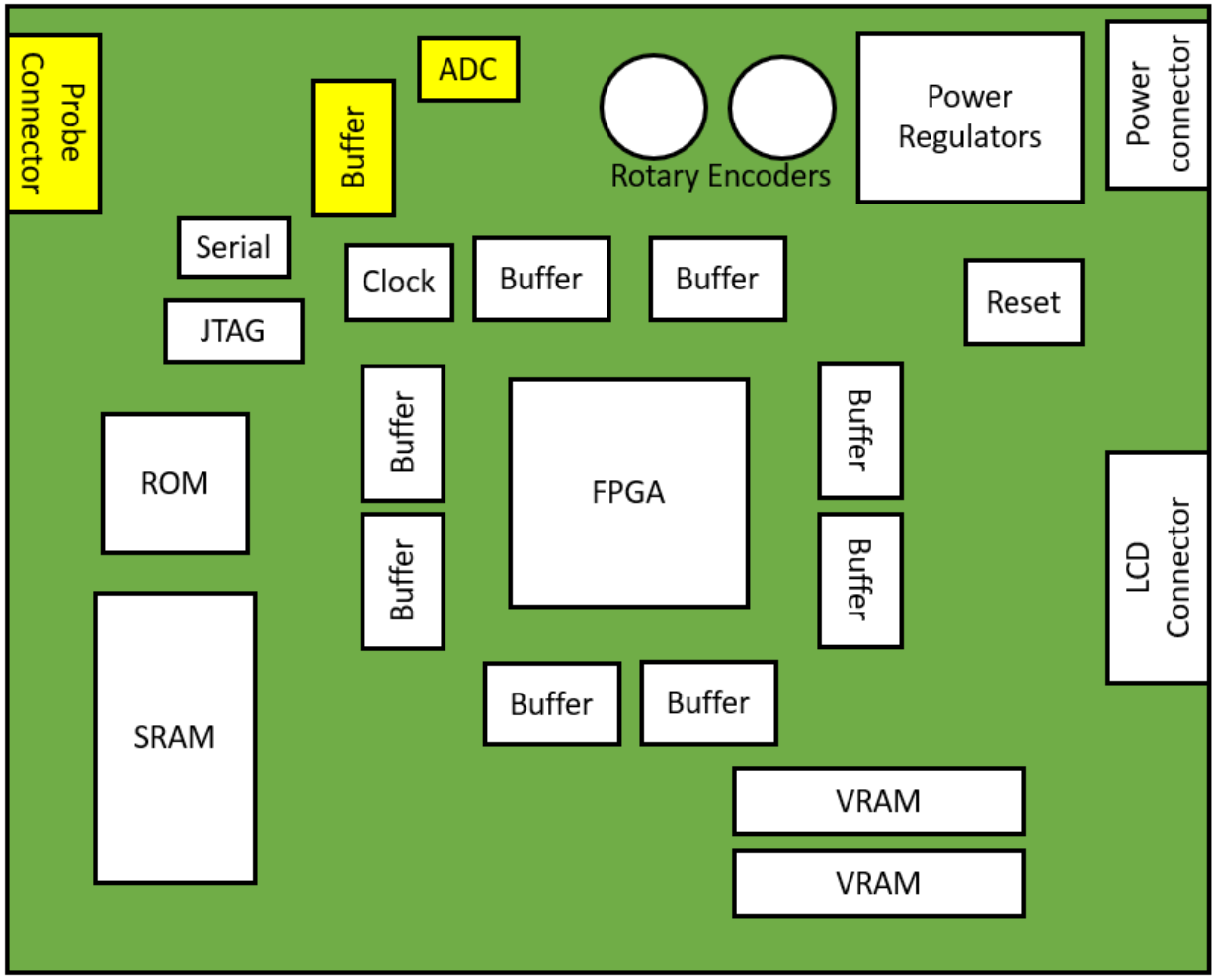

Figure 13: Board Layout: Analog circuitry.

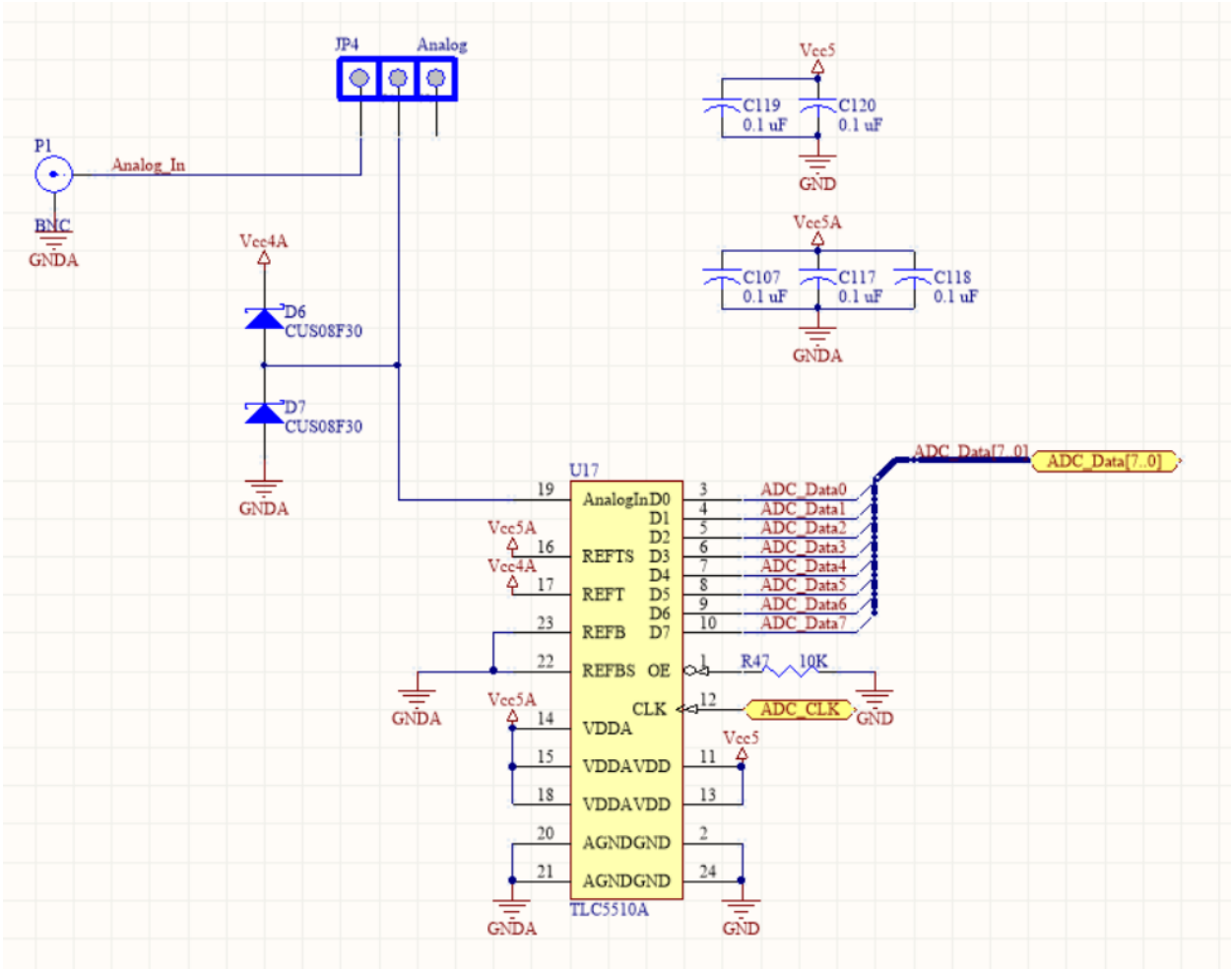

Figure 14: ADC circuit schematic.

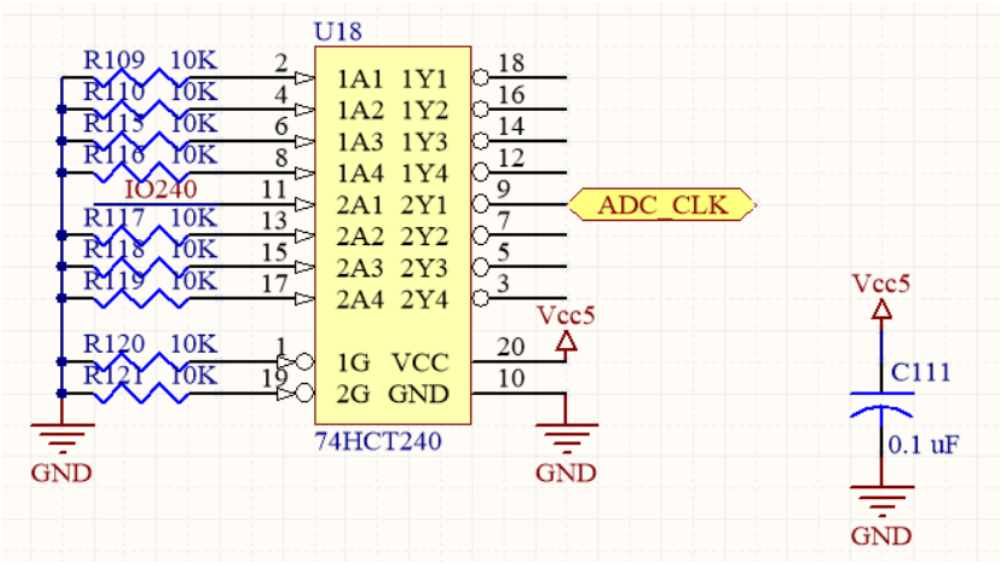

Figure 15: Buffer for FPGA to ADC signals.

Summary: An analog signal is inputted through the BNC connector (P1). A jumper connector was used in case additional analog front end circuitry was used. The signal then passes through protection diodes and is sent to the ADC. The data is then sent through the buffers to the FPGA and analog controller, which stores the signal at the sampling rate once it has been triggered.

ADC: The analog input can range from 0 to 4 V. An analog-to-digital converter (U17, TLC5510A) was used to digitize the input signal to 8 bits. This specific ADC was chosen because of the simplicity of the circuit needed in order to meet the 0-3 V requirement.

The 8 bit data bus was connected to the CPU in parallel, and a clock input was generated from the analog controller. A separate buffer was required to meet the ADC input voltage requirements (U18 in Figure 15). Protection diodes (D6, D7 in Figure 14) with a low forward voltage (about 0.2 V) were used to keep the analog input within the allowed input range.

Several analog supply voltages were used to isolate the analog and digital circuitry. An external 4 V analog reference was generated and used for a 0-4 V input signal range. A 5 V analog power supply was also required, along with an analog ground reference connected to digital ground through a power inductor (Part L2, Figure **??**).

Analog logic: The analog controller consists of a trigger controller to determine

when a trigger has occurred, along with logic for storing the signal at the specified sampling rate. If a trigger occurs, or if the auto triggering times out, the signal is stored in a FIFO buffer to be clocked out and displayed.

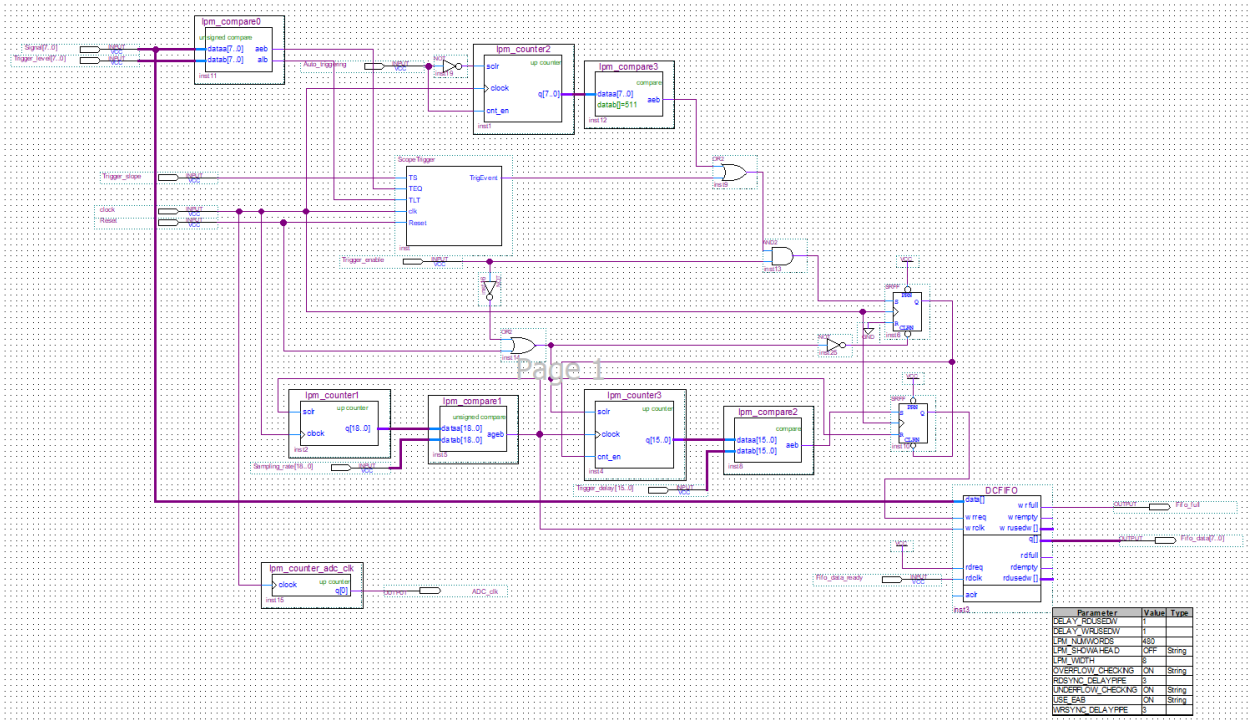

Figure 16: Analog controller block diagram.

Analog input settings: Several values are taken as outputs from the CPU and are used in the hardware logic. Specifically, an 8 bit trigger level, 1 bit auto triggering, 1 bit trigger slope, 1 bit trigger enable, 19 bit sampling rate, and 16 bit trigger delay signal are used as set by the user.

Trigger logic: First, the system determines whether or not a trigger has occurred. A Moore state machine (the ScopeTrigger block in Figure 16) is used to determine whether or not a trigger has occurred. The trigger event output is set high when the trigger slope is positive and the signal has transitioned from below to above the trigger level, or if the trigger slope is negative and the signal has transitioned from above to below the trigger level. The logic description can be found at Appendix **??**.

In order to do this, the 8-bit signal is compared to the trigger level in lpm_compare0 in Figure 16, and the outputs are sent to

the trigger state machine.

Autotriggering: The system also generates a trigger event after 512 clocks if auto trigger is enabled, which was arbitrarily chosen as a short amount of time. Future designs may be improved by correlating the auto trigger countdown to the number of samples and the sampling rate.

An SRFF is set once a trigger event has occurred, and the controller is reset to wait for another trigger event when the trigger enable signal is disabled and re-enabled (set low then high).

FIFO buffer storage: A FIFO buffer is used to store the signal once a trigger event has occurred (DCFIFO in Figure 16). The FIFO has a size of 480 (the width of the LCD) x 8 bits of data. Once a trigger event has occurred, a counter and comparator are used to create the trigger delay, and an SRFF is set once the delay has been reached to enable writing to the FIFO. A counter and comparator are used to create a clock at the given sampling rate, which is sent to the write clock of the FIFO. The signal is then written into a FIFO buffer at the sampling rate once the trigger delay has finished, and is clocked out by the CPU when it is full.

Analog outputs: As seen in Figure 16, the Fifo_full signal indicates when the FIFO is full and the sample has been stored, the Fifo_data bus contains the stored signal data. The Fifo_data_ready input is controlled from the CPU, which sends the clock necessary to read out the signal from the data bus.

The ADC clock (ADC_clk) is also outputted from the controller to the ADC. It is is half the 24 MHz system clock (or a 12 MHz clock) because of the maximum ADC conversion rate, which is 20 M samples per second.

2.1.6 Rotary Encoders

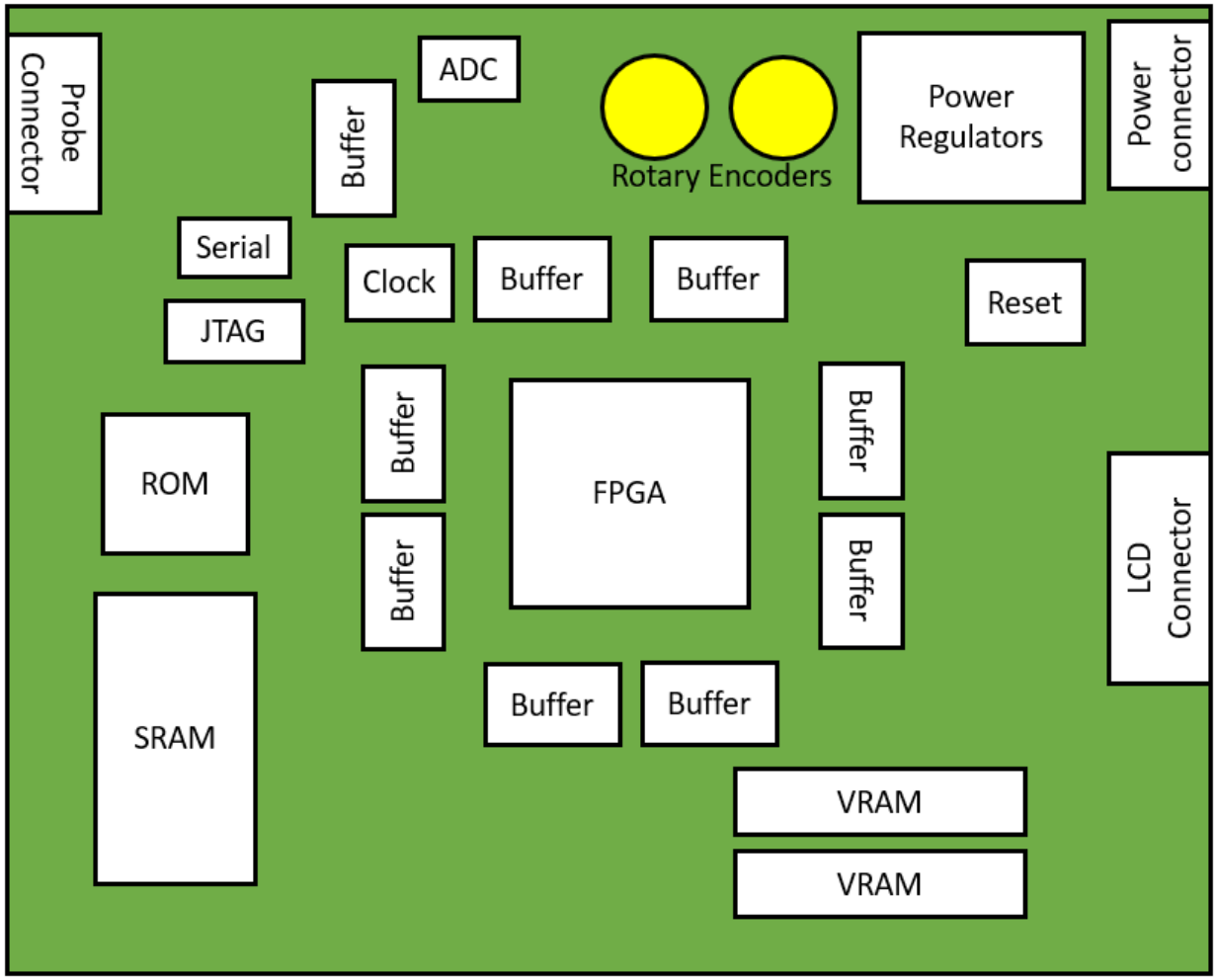

Figure 17: Board Layout: Rotary Encoders.

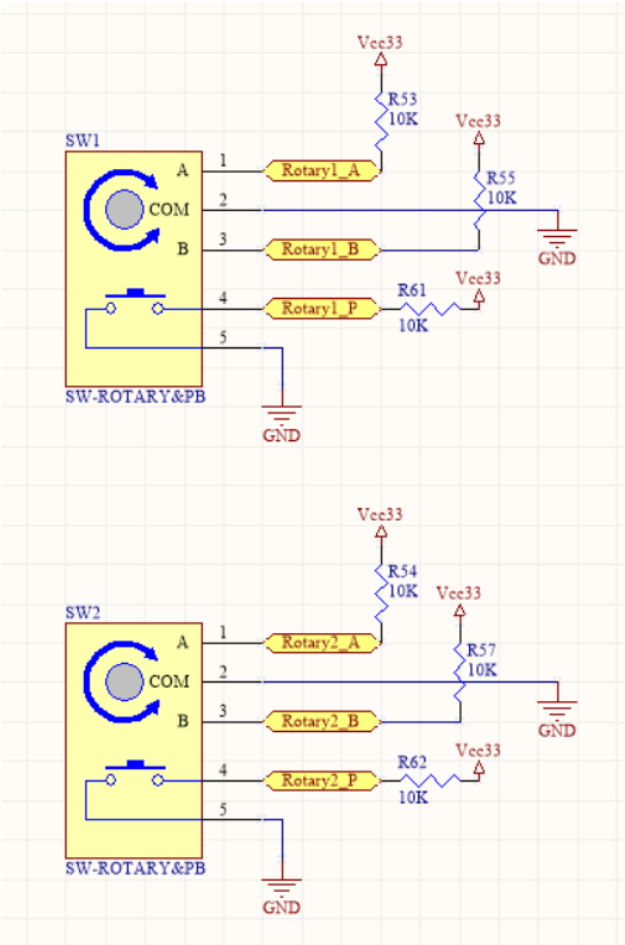

Figure 18: Rotary encoders schematics.

Two rotary encoders are used for the user input (SW1, SW2). These include and A and B output for the encoder and a single pull single throw push button switch.

Pins A and B are pulled high[3], while the common pin was grounded to create out-of-phase pulses from outputs A and B.

Rotary decoder logic:

[3] 1 K resistors were used in place of the 10 K resistors on the schematic because the voltage high output was too low for the buffers.

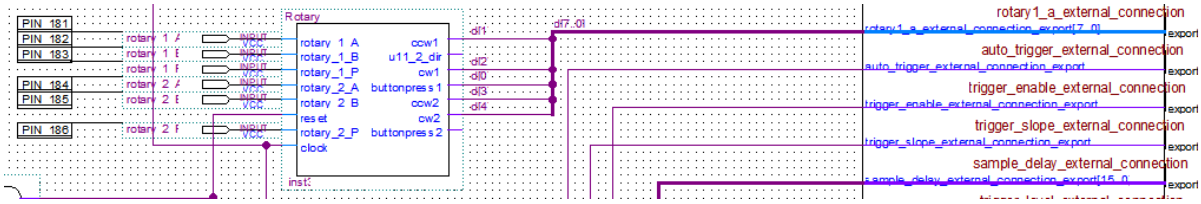

Figure 19: Connections between CPU and rotary controller.

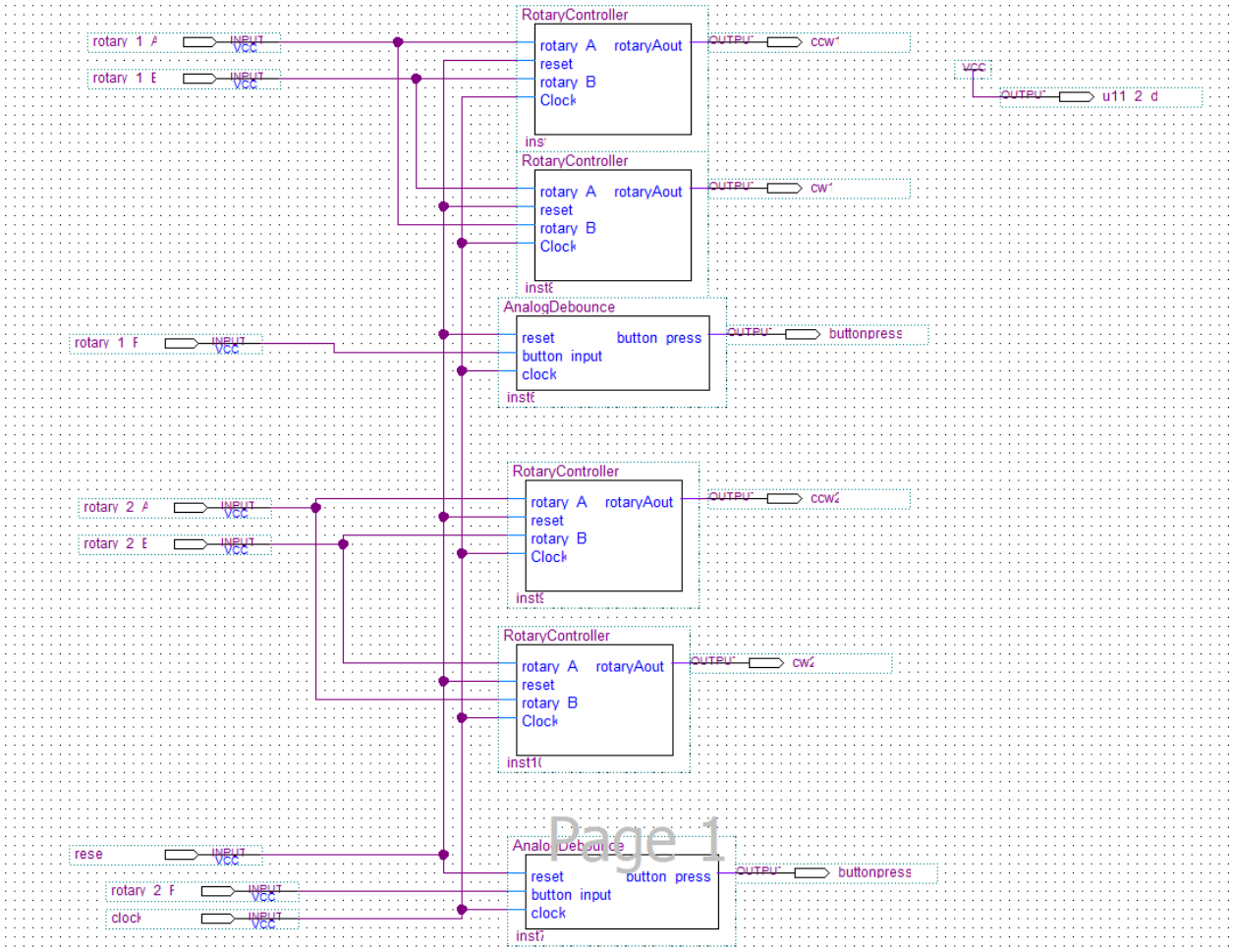

Figure 20: Rotary controller.

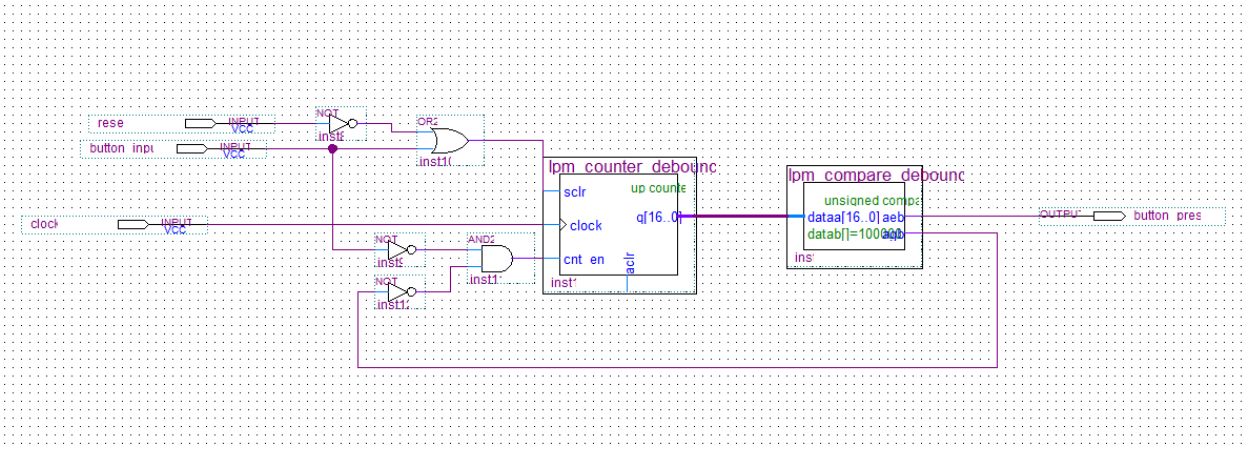

Figure 21: Debouncer for push button.

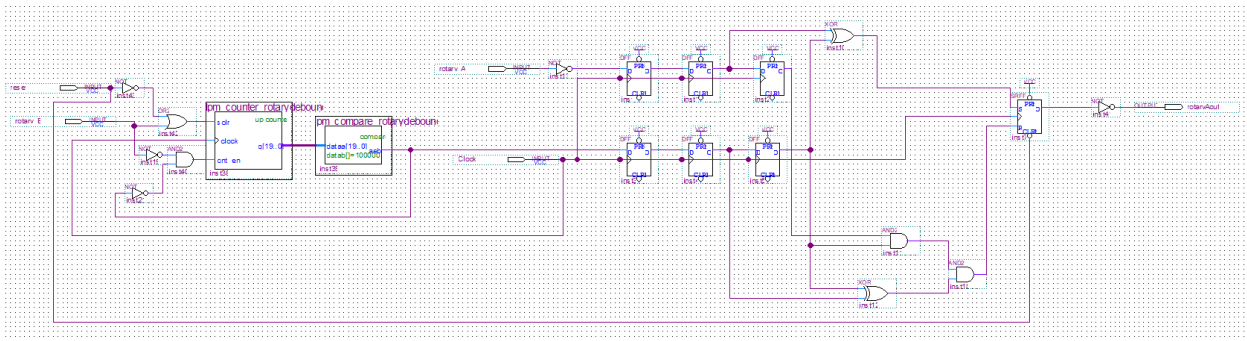

Figure 22: Rotary decoder.

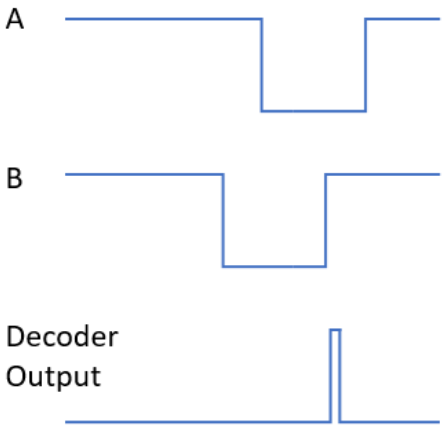

Figure 23: Output of rotary decoder, counterclockwise turn.

The switch input and the two out-of-phase signals from each rotary encoder are decoded with the rotary controller. The rotary logic block consists of a debouncer for the push button and a decoder for the rotary encoder.

The debouncer (Figure 21) takes as inputs the active-low button input (Rotary1_P and Rotary2_P on the schematic), clock, and active-low reset signal. It outputs a single active-high clock pulse whenever the button input signal is low for longer than 100000 clocks, which was chosen through testing the switches.

The decoder (Figure 22) takes as inputs the two active-low outputs from the encoder (Rotary1_A and Rotary1_B, and Rotary2_A and Rotary2_B on the schematic), and the reset and clock signals, and outputs a single active-high clock pulse for each counterclockwise turn of the rotary encoder, where output A leads B (Figure 23).

This is accomplished by first debouncing the input using a counter and comparator to eliminate glitches. Several DFFs are used in series in order to read inputs A and B from two consecutive clocks to determine when edges have occurred. An SRFF is set low when input B goes from low to high while input A is still low, and is set high otherwise. The inverse of this is returned as the output, resulting in a decoder for one direction. Two decoders are used for each rotary encoder to distinguish clockwise and counterclockwise turns (Figure 20).

2.1.7 VRAM and LCD

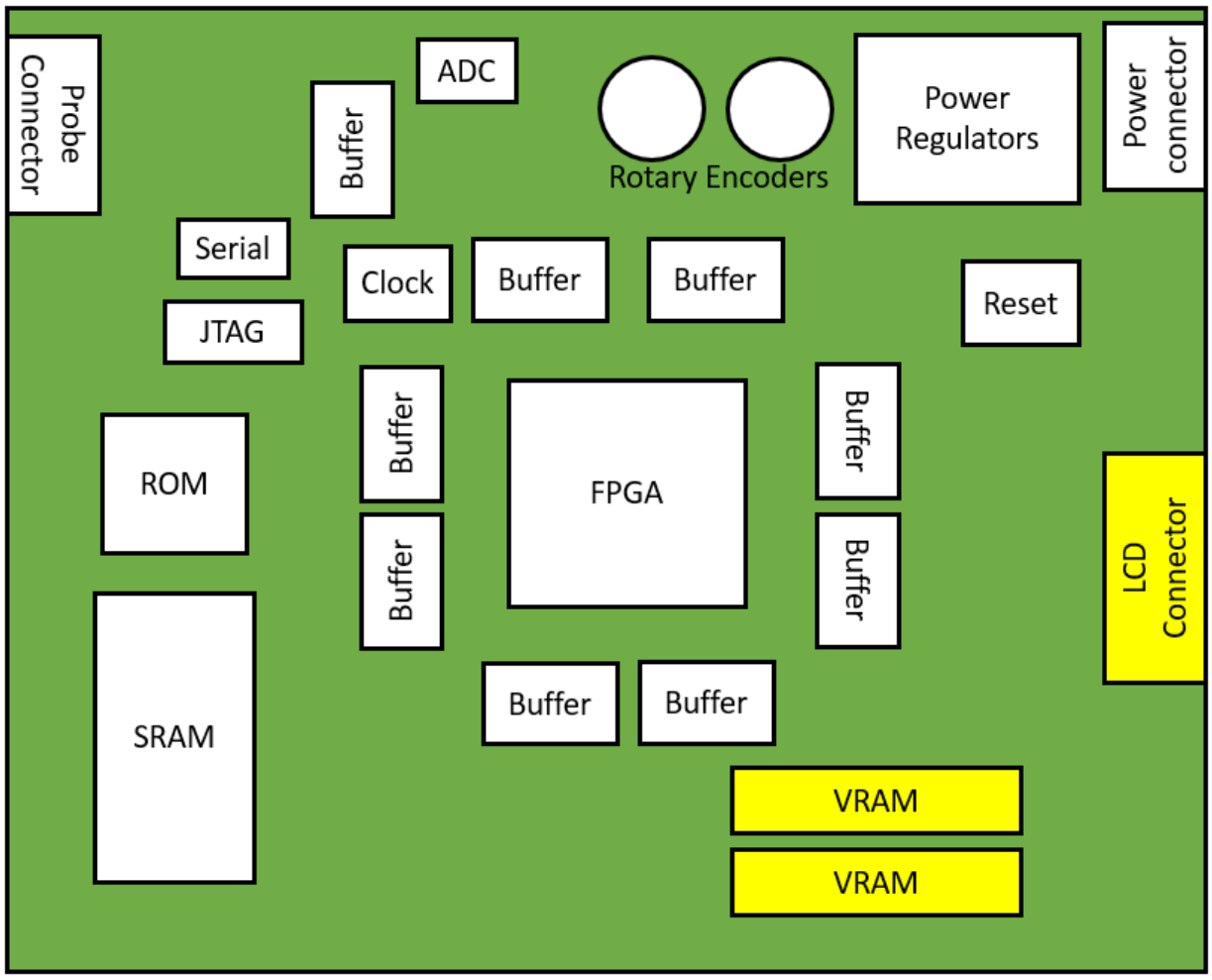

Figure 24: Board Layout: Display.

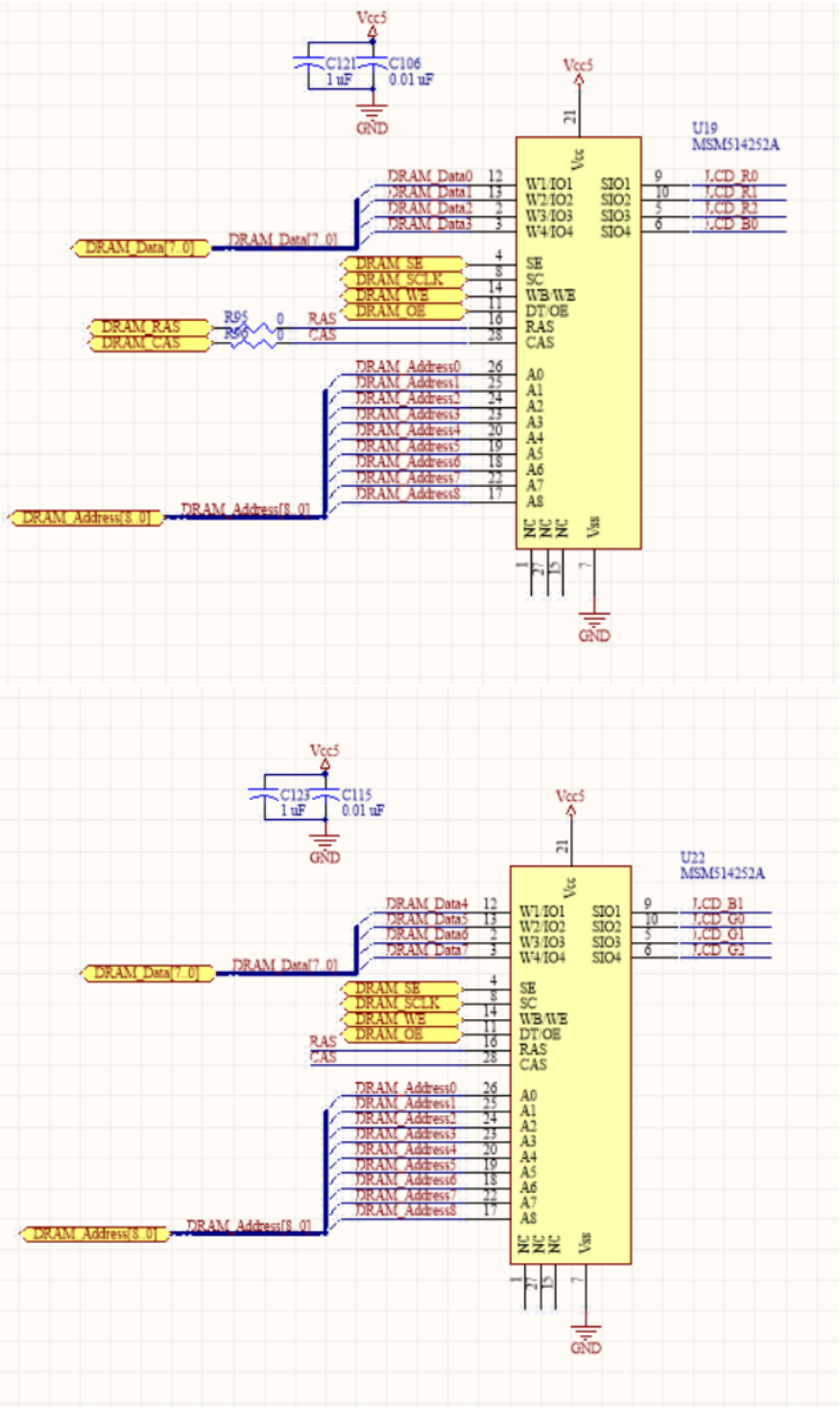

Figure 25: VRAM schematics.

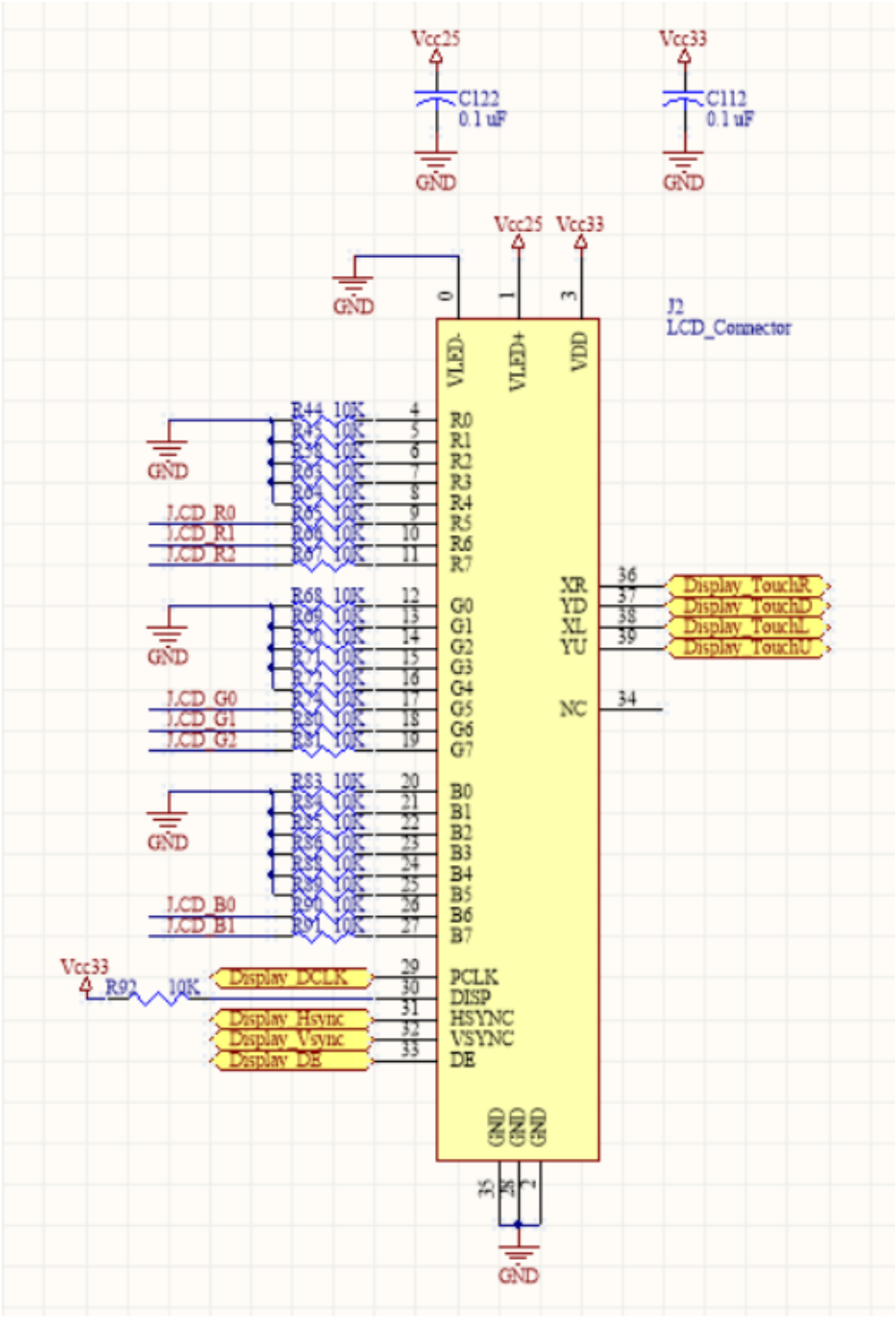

Figure 26: LCD connector schematics.

Two 256Kx4 DRAM with 512x4 SAM (Serial Access Memory) chips (U19, U22, MT42C4225) were used for a total of 8 bits of data, with a 480x272 RGB TFT LCD (Connector J2, ER-TFT043-3).

VRAM: Each VRAM sends 4 bits of serial data to the LCD, and receives 4 bits of data from the CPU, along with a 9 bit address bus, row address strobe (RAS), column address strobe (CAS), write enable (WE), and output enable (OE) signals from the VRAM controller. A serial clock signal is also sent to the VRAM. The LCD controller generates the various clock signals that are sent to synchronize the VRAM and LCD. The same address bus and control signals are used between the two VRAM, and each sends and receives 4 different data bits.

Resistors R95 and R96 are included in series in the RAS and CAS lines for termination.

The VRAM consists of the DRAM, transfer circuitry, and SAM. Read and write cycles are performed to read and write to the DRAM, and read transfer cycles are performed to transfer a row from the DRAM to SAM. Serial output cycles are performed to output the current row data from the SAM to the LCD. The VRAM is refreshed when no other operations are being performed in order to retain data.

LCD: The LCD receives 8 bits of data in parallel from the VRAM: 3 bits red, 3 bits green, and 2 bits blue, with the unused color bits pulled down. It also receives a horizontal sync, vertical sync, clock, and data enable signal from the LCD controller. The touch inputs were unused.

The LCD also required a 25V backlight supply (VLED on the schematic). A step-up converter (U21) was used, which is discussed in more detail in Section 2.1.3.

Logic overview: The VRAM-LCD system consists of a VRAM controller and LCD controller that interact with each other, the CPU, and the VRAM and LCD.

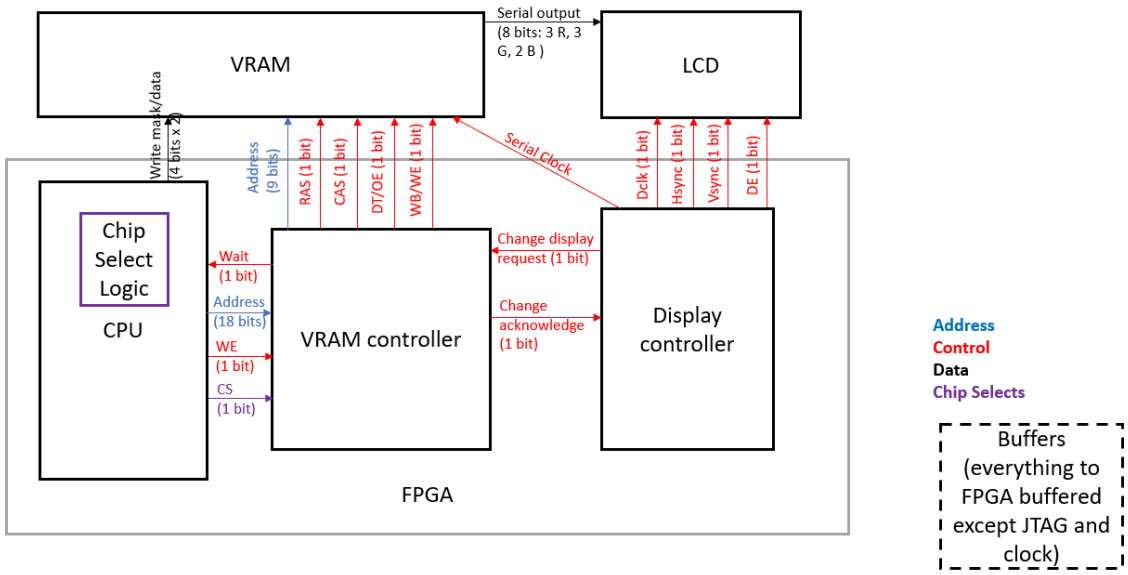

Figure 27: Block Diagram for VRAM and LCD.

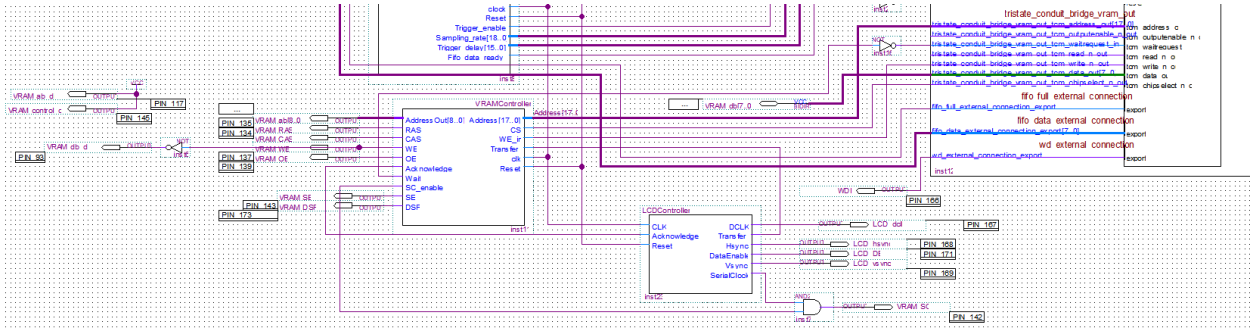

Figure 28: VRAM and LCD Controllers.

VRAM Controller: The VRAM controller takes several inputs from the CPU: 1-bit chip select and write enable signals, and an 18-bit address. It also takes as an input a transfer request signal from the display controller. A state machine is used to generate the output signals, the RAS, CAS, address selector, WE, OE, serial clock, CPU wait output, and transfer acknowledge signals.

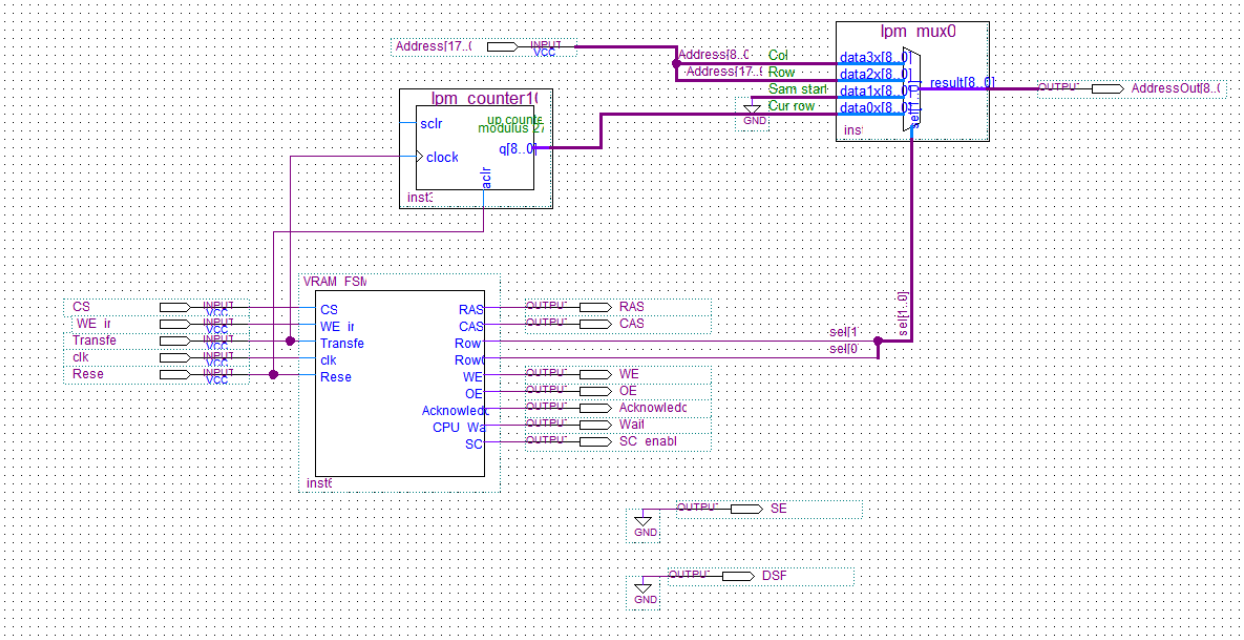

Figure 29: VRAM Controller.

Address: The output address is selected and outputted using a multiplexer. For either the column or row address, the first or second half of the input address is returned. In the case of a row transfer, the SAM start address, 0 for the beginning of the memory, and the current row, are outputted at the appropriate times. A counter is used to increment the current row for after each row transfer cycle, from row 0 to 272, or the total height of the LCD.

VRAM State Machine: The read, write, row transfer, and refresh cycles are performed with a Moore state machine, Listing **??**.

The state machine begins at an idle state, and goes back to the idle state after each cycle is completed. If there is a row transfer request, when the transfer flag is active, the row transfer cycle is performed. If there is a write request, when the chip select and write enable inputs are active, then the write cycle is performed. If there is a read select, when the chip select is enabled but write enable is not, then the read cycle is performed. If none of these cycles are to be completed when in the idle state, a refresh cycle is performed.

Timing diagrams were used to create the state machine with the correct outputs, found in Appendix A.

LCD Controller: The LCD controller is used to generate the clock signals for the

LCD.

The Vertical Sync (VSYNC) signal is used for changing rows, and Horizontal Sync (HSYNC) for changing columns. The Data Enable (DE) signal is also generated for when input data is valid within the VSYNC AND HSYNC signals. A 12 MHz clock signal is also sent to the LCD, while a serial clock signal is generated for the VRAM clock input to the serial address counter for the SAM registers.

The row transfer request (Transfer signal in Figure 30) is set when the end of the row has been reached, and the flag is cleared once the VRAM controller has completed the row transfer and set the transfer acknowledge flag.

The timing diagrams can be found in Appendix A, Figures 40 and 41.

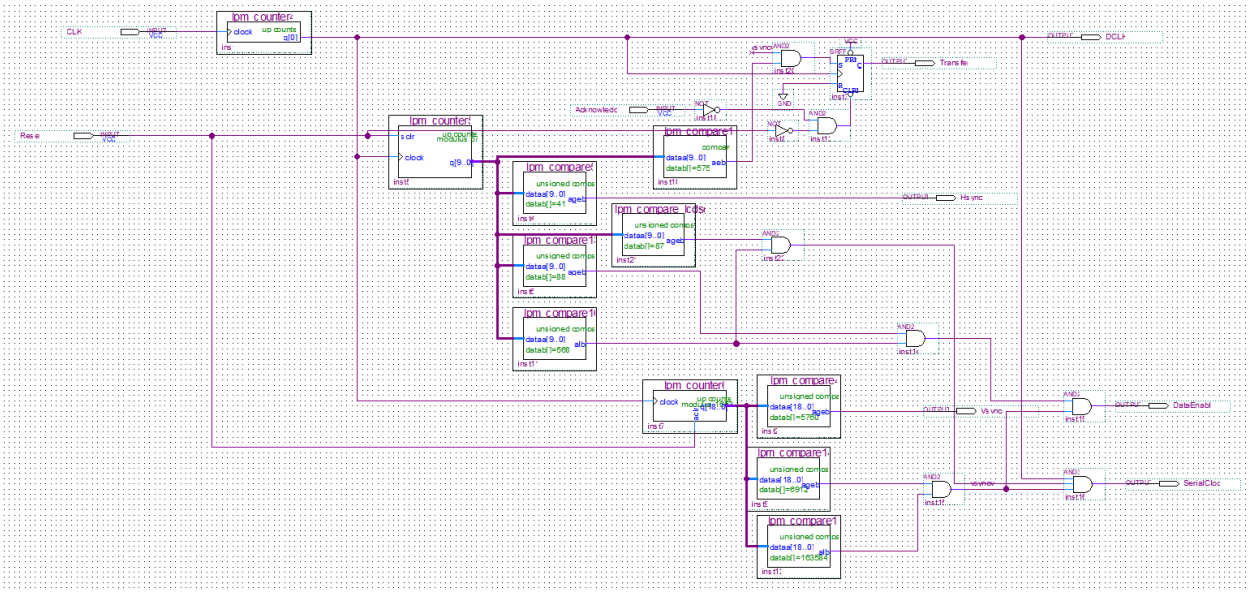

Figure 30: LCD Controller, with comparators for defining valid regions in the signals.

2.1.8 JTAG, Reset, and Clock

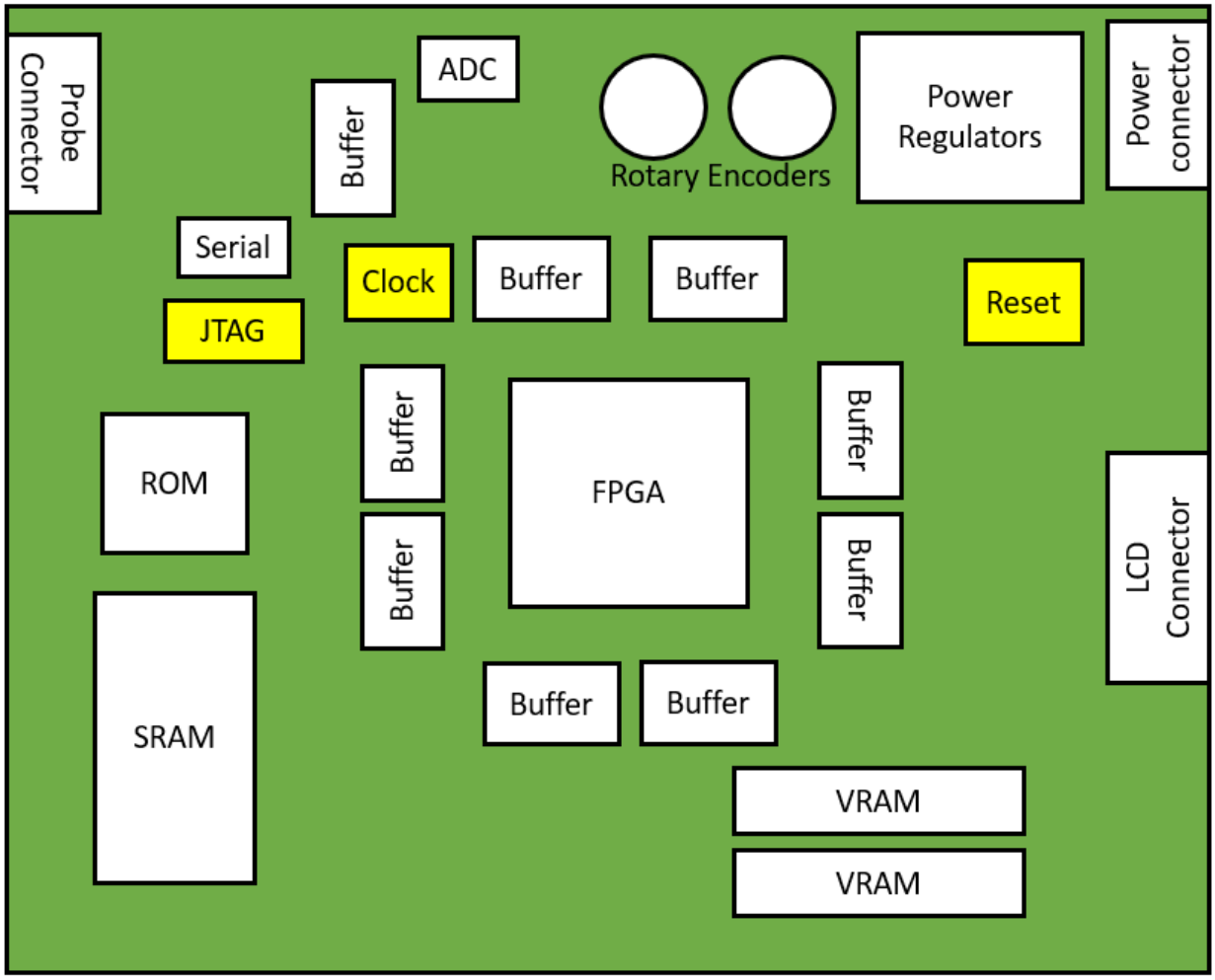

Figure 31: Board Layout: Clock, JTAG, Reset.

JTAG:

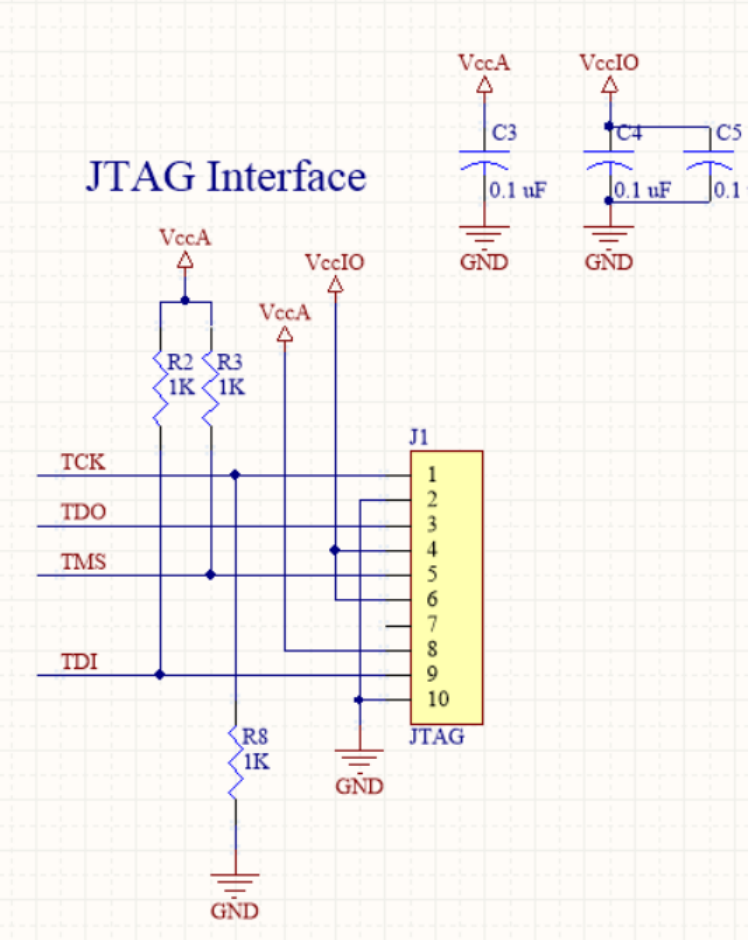

Figure 32: JTAG interface Schematic.

A JTAG interface was used for debugging the system, seen in Figure 32 and is connected directly to the appropriate FPGA pins.

Reset:

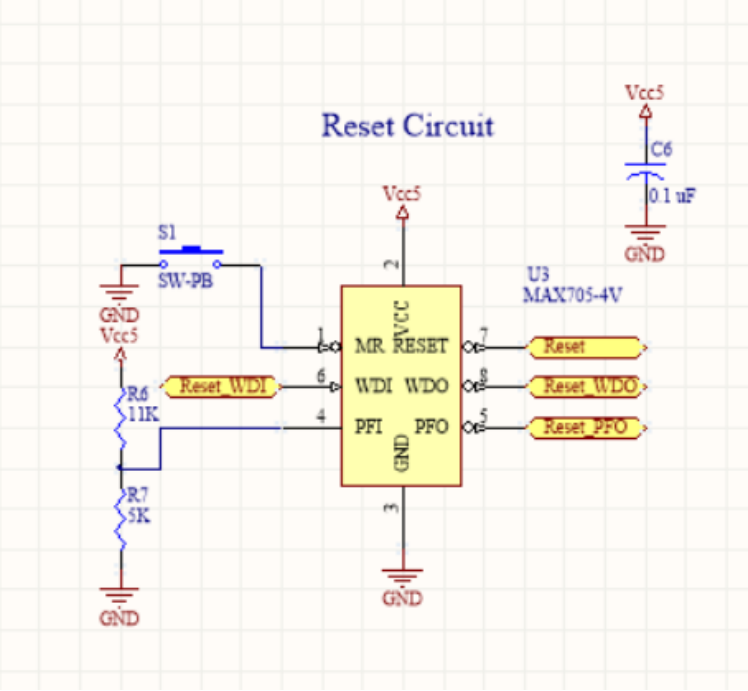

Figure 33: Reset Circuit Schematic.

A reset circuit is used, which includes a microprocessor supervisory device (U3, MAX705). The watchdog input (WDI) is implemented in the main loop, and a voltage divider is used for a power-fail warning at 4 V from the 5 V supply.

The active-low watchdog output, power-fail output, and reset output, and manual reset output are combined for an overall reset signal that is sent to the CPU and various controllers.

Clock:

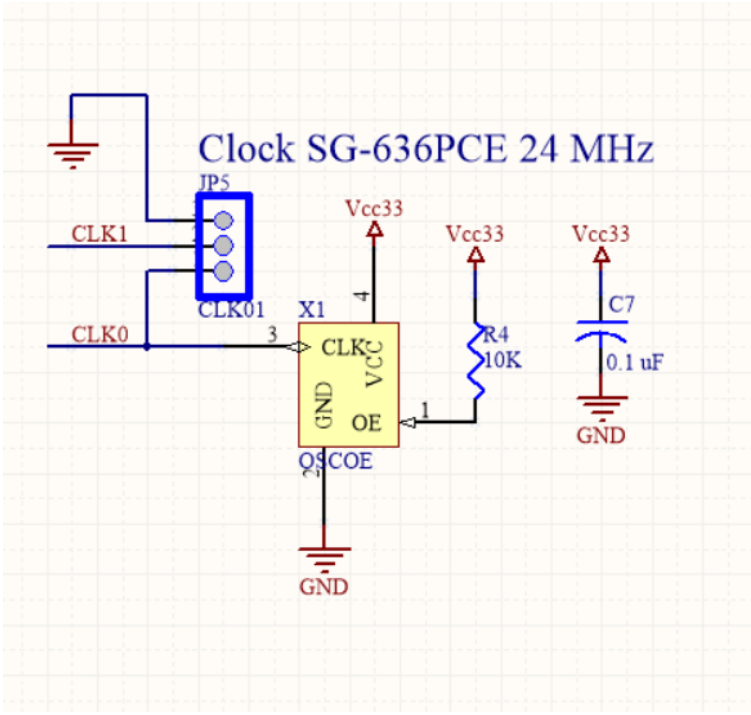

Figure 34: Clock Circuit Schematic.

A 24 MHz oscillator (X1) was used for the clock input for the device, with the CLK0 signal connected directly to the appropriate FPGA pin.

2.1.9 Fixes

wrong adc footprint, buffer soldering error, backwards LCD connector

2.2 Software

2.2.1 Software System Overview

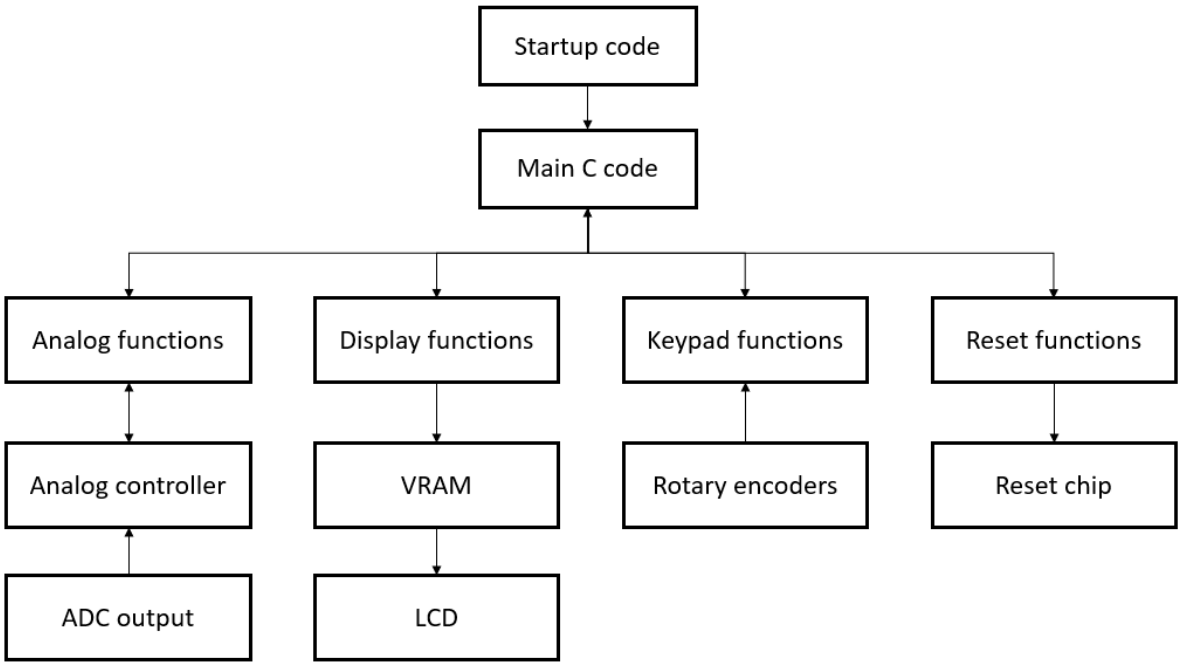

Figure 35: Software block diagram overview.

The main components that were written were routines for the analog settings, display, and keypad [4]. As shown in Figure 35, the main code handles the main loop and calls the various utility functions. The analog functions set the trigger settings used by the analog controller, and also clock out the signal data from the FIFO buffer into a buffer in the data section to be displayed. Display functions clear the display and plot a pixel by writing to the VRAM, whose data is outputted to the LCD. Keypad routines include an event handler and getting the key pressed. A reset function that toggles the watchdog input on the reset chip was also written. These will be discussed in more details below.

[4] rotary encoders were implemented instead of a keypad, so references to the keypad refer to the rotary encoders

2.2.2 Analog Software

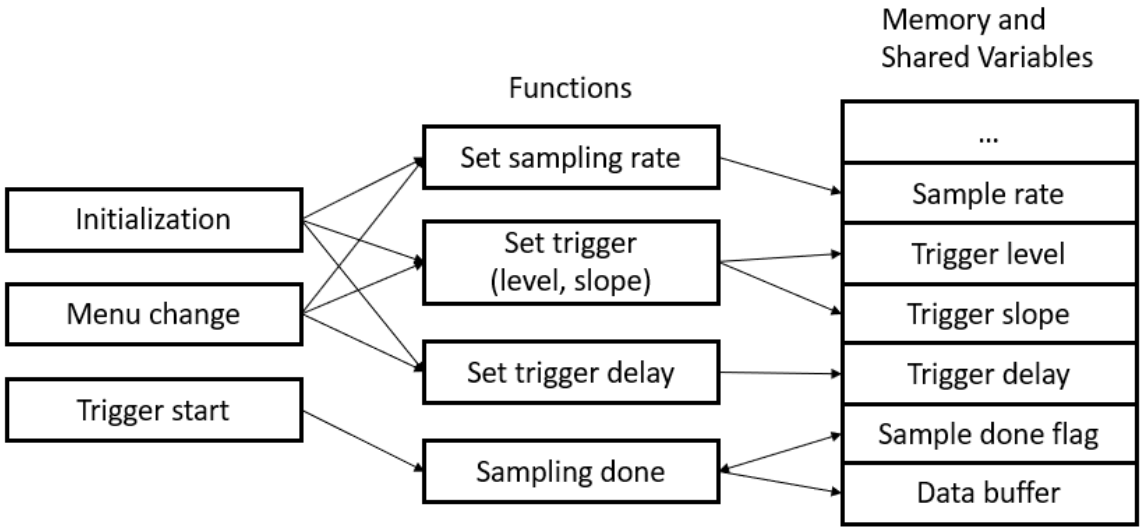

Figure 36: Analog software block diagram.

Listing 1: ../osc134/analog.S

```
# ##############################################################################
#                                                                              #
#                 Analog functions for digital oscilloscope                    #
#                                EE 52                                         #
#                                                                              #
#                                                                              #
# ##############################################################################
# Name of file: analog.s
# Description: Analog functions for digital oscilloscope
# Public functions:
#   int set_sample_rate(long int samples_per_sec): Sets the sample rate
#       to the passed value (in samples per second). The number of samples
#       that will be taken at that sample rate is returned.
#   void set_trigger(int level, int slope): Sets the trigger level to the
#       first argument and the slope to the second argument
#       (1 for negative slope, 0 for positive slope). The trigger level is
#       passed as a value between 0 and 127, with 0 indicating the lowest
#       (most negative for bipolar input) trigger input level and 127
#       indicating the highest (most positive) trigger input level.
#   void set_delay(long int delay): Sets the trigger delay to the passed value.
#   void start_sample(int auto_trigger): Immediately starts sampling data.
#       If the argument is FALSE, then the sampling should start when there
#       is a trigger event. If the argument is TRUE, then the sampling should
#       start when there is a trigger event or when the automatic trigger
#       timeout occurs, whichever is first.
#   unsigned char far *sample_done(void): Returns NULL if not done with the
#       current sample and a pointer to the sampled data otherwise. The
#       sampled data should contain the number of data points previously
#       returned by the set_sample_rate function. The function should only
#       return a non-NULL pointer once for each call to the start_sample
#       function.
# Local functions: None.
# Input:   None.
# Output:  None.
#
# Revision History: 06/24/17 Sophia Liu initial revision
#                   09/10/17 Sophia Liu   edited comments

# inc header files for constants
#include "general.h"

# set sample rate
#
#
```

```
# Description: Sets the sample rate to the passed argument, in samples per
#    second. Returns the number of samples that will be taken at that sample
#    rate.
#
# Operation: Divides the clock frequency by the rate to get the number of
#    clocks between samples. Writes the sample number to the sample rate pio.
#    Returns the sample number.
#
# Arguments:       Long int samples_per_sec (r4) - 19-bit value for the
#                  number of samples per second.
#
# Return Values:   int num_samples: number of samples that will be taken at
#                  the clock rate constant.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8. r9. r2.
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global set_sample_rate
.type set_sample_rate, @function
set_sample_rate:
movhi r8, %hi(CLOCK_FREQ) # store 32 bit clock frequency constant into register
ori r8, r8, %lo(CLOCK_FREQ)
divu r9, r8, r4         # divide clock frequency by sample rate for sample number

movia r8, SAMPLE_RATE_PIO # get address of sample rate pio
stwio r9, 0(r8)          # drive sample number on sample rate pio output

movi r2, NUM_SAMPLES     # return sample size constant (have a set fifo size)
ret
```

```
# set trigger
#
#
# Description: Takes two arguments for trigger level and slope. The trigger
#    level is between 0 (lowest trigger input level) and 127 (highest trigger
#    input level), and the slope is 1 (negative slope) or 0 (positive slope).
#    Sets the trigger level and slope pios to the passed arguments.
#
# Operation: Scales the trigger level argument to match the analog input
#   signal and writes the trigger level and slope arguments to the
#   corresponding pio cores.
#
# Arguments: int level (r4) - 7-bit value for the trigger level,
#              with 0 being the lowest trigger input level and 127 being the
#              highest trigger input level.
#          int slope (r5) - 1-bit value for the trigger slope, with
#              1 being a negative slope and 0 being the positive slope.
#
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r4, r8.
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global set_trigger
.type set_trigger, @function
set_trigger:
movia r8, TRIGGER_LEVEL_PIO # get trigger level pio address
muli r4, r4, 2           # multiply by 2 to get a range of 0 to 254,
```

```
                        #     to match 8 bit ADC data/signal range
stbio r4, 0(r8)         # write level argument to trigger level pio

movia r8, TRIGGER_SLOPE_PIO # get trigger slope pio address
stbio r5, 0(r8)         # write slope argument to trigger slope pio

ret

# set delay
#
#
# Description: Takes one argument, the delay in samples. Sets the trigger
#    delay to the passed value (in samples)
#
# Operation: Writes the sample delay argument to the corresponding pio.
#
# Arguments:       long int delay- 16-bit value for the delay (in samples).
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8.
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global set_delay
.type set_delay, @function
set_delay:
movia r8, SAMPLE_DELAY_PIO # get the sample delay pio address
stwio r4, 0(r8)            # write delay argument to delay pio core

ret
```

```
# start sample
#
#
# Description: Starts sampling data. Takes one argument for auto triggering.
#   If the argument is false, sampling starts when there is a trigger event.
#   If the argument is true, the sampling starts when there is a trigger event
#   or when the automatic trigger timeout occurs (whichever is first)
#
# Operation: Writes the auto trigger argument to the auto trigger pio, and
#    sets the trigger enable pio high to start sampling data. Sets the sample
#    done shared variable to false to indicate that no trigger has occured yet.
#
# Arguments:       int auto_trigger - 1-bit value for auto-triggering.
#                      FALSE if auto-triggering is not enabled, TRUE
#                      if it is enabled.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: sample_done_flag(W) - 1 byte boolean flag. TRUE if the
#                      sample has been taken, FALSE otherwise.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9.
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global start_sample
.type start_sample, @function
start_sample:
movia r8, AUTO_TRIGGER_PIO   # get auto trigger pio address
stbio r4, 0(r8)              # write auto triggering argument to pio core

# enable triggering- pulse trigger enable pio and set high
```

```
movia r8, TRIGGER_ENABLE_PIO # get trigger enable pio address
movi r9, FALSE
stbio r9, 0(r8)              # set trigger enable pio low

movi r9, TRUE
stbio r9, 0(r8)              # set trigger enable pio high

# reset sample done flag - only after each start_sample call
movia r8, sample_done_flag # get sample done flag shared variable address
movi r9, FALSE
stbio r9, 0(r8)            # set the sample done flag to false
ret

# sample done
#
#
# Description: Returns NULL if not done with the current sample and a pointer
#    to the sampled data otherwise. The sampled data contains the number of
#    data points previously returned by the set_sample_rate functions. Only
#    returns a non-NULL pointer once for each call to the start_sample function.
#
# Operation: Checks the sample done flag and returns null if the sample is
#    already completed. Checks the fifo full pio and returns null if the fifo
#    is not full. Otherwise, the fifo is clocked out into the data buffer
#    shared variable,the sample done flag is set to TRUE, trigger enabling is
#    set to FALSE, and a pointer to the data buffer is returned.
#
# Arguments:       None.
# Return Values:   sampled data - 32-bit pointer to the sampled data. Returns
#                     NULL if not done with the current sample. Only
#                     returns a non-NULL pointer once for each call to the
#                     start_sample function.
#
# Local Variables: None.
# Shared Variables: sample_done_flag(R/W) - 1 byte boolean flag. TRUE if the
#                     sample has been taken, FALSE otherwise.
#                  data_buffer(W) - NUM_SAMPLES (number of samples taken)
#                     bytes, buffer for sampled data from the fifo buffer.
#
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
```

```
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r2, r8, r9, r10, r11, r12, r13.
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global sample_done
.type sample_done, @function
sample_done:
movi r8, TRUE           # store true to compare to sample done flag

movia r9, sample_done_flag # get sample dong flag shared variable address
ldb r10, 0(r9)          # load sample done flag value from shared variable
beq r10, r8, return_null_ # If the flag is true, the sample is already done,
                        #    return null.
# bne r10, r8, check_fifo # Else flag is false, check if fifo is full

check_fifo:
movia r9, FIFO_FULL_PIO # get address of the fifo full pio
ldbio r10, 0(r9)        # check to see if the fifo is full
bne r10, r8, return_null_ # If the flag is not true, fifo is not full,
                        #    return null.
# beq r10, r8, return_pointer # Else flag is true, fifo is full, return pointer

return_pointer:         # copy fifo sample buffer into data buffer
movia r8, DATA_READY_PIO # get address for fifo read clock input pio
movia r10, FIFO_DATA_PIO # get address to read fifo data from
movia r11, data_buffer  # get address of data buffer shared variable
addi r13, r11, NUM_SAMPLES # get address of last sample in data buffer

# loop to clock out fifo buffer
read_fifo_loop:
ldbio r12, 0(r10)       # load current sample from fifo
stb r12, 0(r11)         # store sample in data buffer
addi r11, r11, 1        # go to next address in data buffer

bge r11, r13, got_sample # if at address of last sample in data buffer,
                        #    finished reading samples from fifo
# blt r11, r13, read_fifo_loop_next # otherwise, continue loop
```

```
read_fifo_loop_next:    # clock out next sample from fifo
movi r9, 0
stbio r9, 0(r8)
movi r9, 1
stbio r9, 0(r8)
movi r9, 0
stbio r9, 0(r8)         # pulse read clock for fifo
jmpi read_fifo_loop     # go back to top of loop to read the next sample

got_sample:
movia r8, sample_done_flag # get address of sample done flag shared variable
movi r9, TRUE
stb r9, 0(r8)             # set sample done flag to true - completed sample

movia r8, TRIGGER_ENABLE_PIO # get address of trigger enable pio
movi r9, FALSE
stbio r9, 0(r8)           # stop sampling, set trigger enable to false

movia r2, data_buffer     # return pointer to sampled data
ret

return_null_:
movi r2, PTRNL            # return null
ret

# init analog
#
#
# Description: Initializes variables used for analog functions.
#
# Operation: Initializes the sample done flag to false (sample has not been
#    completed) and sets the trigger enable pio to false (triggering not
#    enabled yet).
#
# Arguments:       None.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: sample_done_flag(W) - 1 byte boolean flag. TRUE if the
#                       sample has been taken, FALSE otherwise.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
```

```
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9.
# Stack depth:     2 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global init_analog
.type init_analog, @function
init_analog:
addi sp, sp, -8      # adjust stack pointer
stw ra, 0(sp)        # store return address
stw r28, 4(sp)       # store frame pointer

movia r8, sample_done_flag # get address of sample done flag shared variable
movi r9, FALSE
stb r9, 0(r8)              # set sample done flag to false

init_pios:
# disable triggering
movia r8, TRIGGER_ENABLE_PIO # get address of the trigger enable pio
movi r9, FALSE
stbio r9, 0(r8)           # set trigger enable pio to false


ldw ra, 0(sp)        # load return address
ldw r28, 4(sp)       # load frame pointer
addi sp, sp, 8       # adjust stack pointer
ret

# data section
.section .data
.align 4
sample_done_flag: .byte 0x0 # 1 byte boolean flag. TRUE if the sample
                            # has been taken, FALSE otherwise.

data_buffer: .skip NUM_SAMPLES # NUM_SAMPLES (number of samples taken) bytes,
                            # buffer for sampled data from the fifo buffer.
```

2.2.3 Display Software

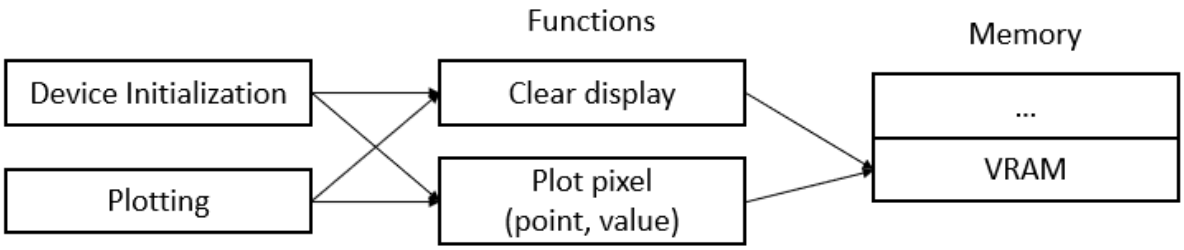

Figure 37: Display software block diagram.

Listing 2: ../osc134/displayFunc.S

```
# ##############################################################################
#                                                                              #
#               Display functions for digital oscilloscope                     #
#                                EE 52                                         #
#                                                                              #
#                                                                              #
# ##############################################################################
# Name of file: display.s
# Description: Display functions for digital oscilloscope
# Public functions:
#    void clear_display(void): Clears the display (turns black) by storing
#        the pixel off value (PIXEL_WHITE) in all VRAM locations.
#    void plot_pixel(unsigned int x, unsigned int y, int p): Sets the pixel at
#        the passed position (x, y) to the passed value p. The position is
#        passed in binary with (0,0) being the upper left corner.
#
# Local functions: None.
#
# Input:   None.
# Output:  None.
#
# Revision History: 06/06/17 Sophia Liu Initial revision
#                   09/04/17 Sophia Liu   Edited comments
#                   12/2017 Sophia Liu    Edited comments

# inc files for display constants
#include "interfac.h"
#include "general.h"

# Clear display
#
#
# Description: Clears the display (turns black) by storing the pixel off value
#             (PIXEL_WHITE) in all VRAM locations.
#
# Operation: Goes through each VRAM address and stores PIXEL_WHITE (pixel off)
#           at each location.
#
# Arguments:       None.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
```

```
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r6, r9, r11, r12, r13
# Stack depth: 0 words.
#
# Revision History: 06/22/17 Sophia Liu   initial revision
.section .text
.align 4
.global clear_display
.type clear_display, @function
clear_display:
movia r6, VRAM_ADDRESS   # store vram beginning address
movia r13, VRAM_ADDRESS_END # store vram ending address
movi  r9, PIXEL_WHITE    # store value for pixel off
movi  r12, TRUE          # store value for true, for comparisons

clear_display_start:
stb     r9, 0(r6)           # clear next vram byte
addi  r6, r6, 1          # move to next address

cmpeq r11, r6, r13              # check if reached end of vram
beq     r11, r12, clear_display_end # if reached end address, end function
jmpi  clear_display_start        # else move to next address

clear_display_end:
ret

# Plot_pixel
#
# Description: Sets the pixel at the passed position (x, y) to the passed
#             value p. The position is passed in binary with (0,0) being the
#             upper left corner.
#
# Operation: Checks to make sure the position (x, y) is with the bounds of the
#           display screen. If it is, the corresponding vram address is
#           calculated and the pixel value p is stored.
#
```

```
# Arguments:       unsigned int x (r4) - x value of pixel to plot, with (0,0)
#                      being the upper left corner.
#                  unsigned int y (r5)- y value of pixel to plot, with (0,0)
#                      being the upper left corner.
#                  int p (r6) - 8-bit value of the pixel to plot.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9, r10, r11.
# Stack depth:     0 words.
#
# Revision History: 06/22/17 Sophia Liu   initial revision
.section .text
.align 4
.global plot_pixel
.type plot_pixel, @function
plot_pixel:
check_pixel:
movi r9, TRUE
cmpgei r8, r4, SIZE_X    # check if x point is too large for LCD
beq r8, r9, invalid_point # if it is invalid, end

cmpgei r8, r5, SIZE_Y    # check if y point is too large for LCD
beq r8, r9, invalid_point # if it is invalid, end

movia r10, VRAM_ADDRESS  # get vram address
add r10, r4, r10         # add x offset to vram address

muli r11, r5, VRAM_WIDTH # multiply y by length of vram row for y offset
add r10, r11, r10        # add y offset to vram address

stb r6, 0(r10)           # store pixel byte value in vram address
jmpi end_plot
```

```
invalid_point:          # invalid point, return
ret

end_plot:               # plotted point, return
ret
```

2.2.4 Keypad Software

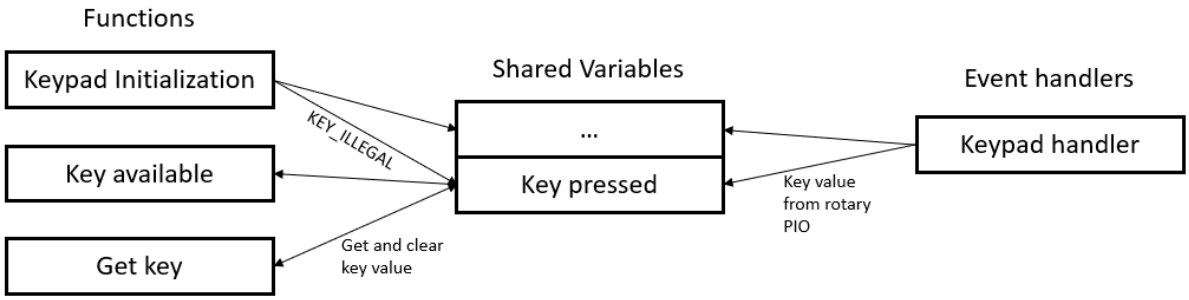

Figure 38: Keypad software block diagram.

Listing 3: ../osc134/keypad.S

```
# ##############################################################################
#                                                                              #
#                Keypad functions for digital oscilloscope                     #
#                              EE 52                                           #
#                                                                              #
#                                                                              #
# ##############################################################################
# Name of file: keypad.s
# Description: Keypad functions for digital oscilloscope. Used to get the
#             current key value and handle keypad interrupts. Rotary encoders
#             were implemented, so "keypad" often refers to the rotary encoders.
# Public functions:
#     unsigned char key_available(void): Whether or not there is a valid key
#         value. Returns TRUE (non-zero) if there is a valid key ready to be
#         processed and FALSE (zero) otherwise.
#     int getkey(void): Returns the keycode (found in interfac.h) for the
#         debounced key. This value is never KEY_ILLEGAL. The function does
#         not return until it has a valid key.
#     void initKeypad(void): Initializes keypad interrupts and variables.
#         Registers keypad event handler, initializes key_pressed with
#         KEY_ILLEGAL, initializes rotary PIO and enables interrupts.
#         Must be called before using the keypad.
#
# Local functions:
#     keypadInterruptHandler: Event handler for the keypad. Stores the current
#         valid key value in key_pressed.
#
# Input:   None
# Output:  None
#
# Revision History: 06/06/17 Sophia Liu initial revision
#                   12/2017 Sophia Liu    edited comments

#include "interfac.h"
#include "general.h"

.section .text

# Key_available
#
#
# Description: Whether or not there is a valid key value. Returns TRUE
#             (non-zero) if there is a valid key ready to be processed and
#             FALSE (zero) otherwise.
```

```
#
# Operation: Reads current key_pressed value. Returns TRUE (non-zero) if there
#           is a valid key ready to be processed and FALSE (zero) otherwise.
#
# Arguments:       None.
# Return Values:   key is available - unsigned char, TRUE (non-zero) if
#                      current key value is valid or FALSE (zero) if it
#                      is not; whether or not there is a valid key ready
#                      to be processed.
#
# Local Variables: None.
# Shared Variables: key_pressed (R) - 1 word unsigned int, contains current
#                      debounced key.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9, r2.
# Stack depth:     2 words.
#
# Revision History: 06/06/17 Sophia Liu   initial revision
.align 4
.global key_available
.type key_available, @function
key_available:
addi sp, sp, -8      # adjust stack pointer
stw ra, 0(sp)        # store return address
stw r28, 4(sp)       # store frame pointer

movia r8, key_pressed   # get address of key_pressed
ldbu r9, 0(r8)          # load value in key_pressed
cmpnei r2, r9, KEY_ILLEGAL # check if current key value is illegal key
                        # r2 = 0 (FALSE) if is illegal key, 1 (TRUE) if not

ldw ra, 0(sp)        # load return address
ldw r28, 4(sp)       # load frame pointer
addi sp, sp, 8       # adjust stack pointer
```

```
ret

# GetKey
#
#
# Description:     Returns the keycode (defined in interfac.h) for the debounced
#                  key, and does not return until there is a valid key.
#
# Operation:       Waits until there is a valid key from key_pressed, and
#                  returns the key value. Never returns KEY_ILLEGAL.
#
# Arguments:       None.
# Return Values:   key - int, the keycode for the debounced key.
#                     Always a valid key, never KEY_ILLEGAL.
#
# Local Variables: None.
# Shared Variables: key_pressed (R/W) - 1 word unsigned int, contains current
#                      debounced key.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r2, r9, r10.
# Stack depth:     2 words.
#
# Revision History: 06/06/17 Sophia Liu   initial revision
.align 4
.global getkey
.type getkey, @function
getkey:
addi sp, sp, -8      # adjust stack pointer
stw ra, 0(sp)        # store return address
stw r28, 4(sp)       # store frame pointer

movia r9, key_pressed # get address of key_pressed

get_key_loop:
ldw r2, 0(r9)            # load value in key_pressed
```

```
cmpeqi r10, r2, KEY_ILLEGAL # check if current key value is illegal key
bne r0, r10, get_key_loop # if current key is illegal key, loop back and
                          #    check again
# beq r0, r10, have_key  # else key is not illegal key, can return

have_key:
movi r10, KEY_ILLEGAL
stw r10, 0(r9)           # clear key_pressed

ldw ra, 0(sp)        # load return address
ldw r28, 4(sp)       # load frame pointer
addi sp, sp, 8       # adjust stack pointer

ret                  # have key, return

# keypadInterruptHandler
#
#
# Description:     Event handler for the keypad. Stores the current
#                  valid key value in key_pressed.
#
# Operation:       Reads in the keycode value and saves it in key_pressed if
#                  it is valid. Clears edge capture register and re-enables
#                  interrupts.
#
# Arguments:       None.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: key_pressed (W) - 1 word unsigned int, contains current
#                      debounced key.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9, r10.
# Stack depth:     2 words.
#
```

```
# Revision History: 06/06/17 Sophia Liu   initial revision
.align 4
.global keypadInterruptHandler
.type keypadInterruptHandler, @function
keypadInterruptHandler:
addi sp, sp, -8      # adjust stack pointer
stw ra, 4(sp)        # store return address
stw r28, 0(sp)       # store frame pointer

load_key:
movia r9, ROT_ADDRESS    # read in rotary pio core address
ldbio r10, 12(r9)

cmpeqi r8, r10, KEY_ILLEGAL    # check if current key value is illegal
bne r0, r8, endKeypadInterruptHandler # if current key is illegal key, end
                                  #    (all rotary actions are valid)

movia r8, key_pressed # get address of key_pressed
stw r10, 0(r8)       # store pio data value from edge register in key_pressed

movui r8, 0
stbio r8, 12(r9) # edge capture register, clear all

movi r8, 0x1f
stbio r8, 8(r9)  # interrupt mask register, enable interrupts

endKeypadInterruptHandler:
ldw r28, 0(sp)   # load return address
ldw ra, 4(sp)    # load frame pointer
addi sp, sp, 8   # adjust stack pointer

ret

# InitKeypad
#
#
# Description: Initializes keypad interrupts and variables. Registers keypad
#             event handler, initializes key_pressed with
#             KEY_ILLEGAL, initializes rotary PIO and enables interrupts.
#             Must be called before using the keypad.
#
# Operation: Registers keypad event handler, initializes key_pressed with
#           KEY_ILLEGAL, initializes rotary PIO and enables interrupts.
#
# Arguments:       None.
```

```
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: key_pressed (W) - 1 word unsigned int, contains current
#                      debounced key.
# Global Variables: None.
#
# Input:           None.
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r4, r5, r6, r8, r9.
# Stack depth:     2 words.
#
# Revision History: 06/06/17 Sophia Liu   initial revision
.align 4
.global initKeypad
.type initKeypad, @function
initKeypad:
addi sp, sp, -8      # adjust stack pointer
stw ra, 0(sp)        # store return address
stw r28, 4(sp)       # store frame pointer

movia r8, ROT_ADDRESS # get rotary pio core address
movi r9, 0
stbio r9, 8(r8)      # interrupt mask register, disable interrupts

movui r4, 0          # arguments for alt_irq_register
movui r5, 0
movia r6, keypadInterruptHandler
call alt_irq_register # register interrrupt handler for keypad (rotary) event

movia r8, key_pressed # get key_pressed address
movi r9, KEY_ILLEGAL
stw r9, 0(r8)        # initialize key_pressed with illegal key

initInterrupts:
movia r8, ROT_ADDRESS # get rotary pio core address
movi r9, 0x0
stbio r9, 4(r8)      # direction register for pio, set to input
```

```
movi r9, 0x0
stbio r9, 12(r8)     # edge capture register for pio, clear all

movi r9, 0x1f
stbio r9, 8(r8)      # interrupt mask register for pio, enable interrupts

ldw ra, 0(sp)        # load return address
ldw r28, 4(sp)       # load frame pointer
addi sp, sp, 8       # adjust stack pointer
ret

# data section
.section .data
.align 4
key_pressed: .word 0x0  # 1 word unsigned int, contains current debounced key
```

2.2.5 Watchdog Reset

A reset function to toggle the watchdog input on the reset circuit (Reset_WDI, Figure 33) was also written. It is called in the main loop (Listing **??**), which should never stall.

Listing 4: ../osc134/reset.S

```
# ##############################################################################
#                                                                              #
#                  Reset functions for digital oscilloscope                    #
#                                EE 52                                         #
#                                                                              #
#                                                                              #
# ##############################################################################
# Name of file: reset.s
# Description: Reset functions for digital oscilloscope
# Public functions:
#   void pulse_wd(void): Pulses the watchdog reset output that is sent to the
#            reset chip. The reset input on the MAX705 used must be toggled
#            at least every 1.6 secs. If this function is not called within
#            that time, the system will reset.
# Local functions:
# Input:   None
# Output:  None
#
# Revision History:
#    06/24/17 Sophia Liu   initial revision
#    12/17   Sophia Liu    edited comments


# inc header files for constants
#include "general.h"

# pulse wd
#
#
# Description: Pulses the watchdog reset output that is sent to the reset
#            chip. The reset input on the MAX705 used must be toggled
#            at least every 1.6 secs. If this function is not called within
#            that time, the system will reset.
#
# Operation: Gets the reset PIO address and sets the output high then low.
#
# Arguments:       None.
# Return Values:   None.
#
# Local Variables: None.
# Shared Variables: None.
# Global Variables: None.
#
# Input:           None.
```

```
# Output:          None.
#
# Error Handling:  None.
# Algorithms:      None.
# Data Structures: None.
#
# Known Bugs:      None.
# Limitations:     None.
# Registers changed: r8, r9
# Stack depth:     0 words.
#
# Revision History: 06/24/17 Sophia Liu   initial revision
.section .text
.align 4
.global pulse_wd
.type pulse_wd, @function
pulse_wd:
movia r8, WD_PIO # get watchdog input pio address
movi r9, 1
stbio r9, 0(r8) # set watchdog input high

movi r9, 0
stbio r9, 0(r8) # set watchdog input low

ret
```

2.2.6 Header files

Listing 5: ../osc134/general.h

```
/*********************************************************************/
/*                                                                   */
/*                           GENERAL.H                               */
/*                      General Definitions                          */
/*                         Include File                              */
/*                  Digital Oscilloscope Project                     */
/*                           EE/CS 52                                */
/*                                                                   */
/*********************************************************************/

/*
   This file contains definitions for the Digital Oscilloscope
   project. This includes addresses and other constants specific to this scope.


   Revision History:
      07/16/17   Sophia Liu   Initial revision
      09/03/17   Sophia Liu   Edited comments
*/

#ifndef __GENERAL_H__
    #define __GENERAL_H__

// addresses
#define VRAM_ADDRESS    0x140000 // vram start address
#define VRAM_ADDRESS_END 0x17ffff // vram end address

// PIO addresses
#define ROT_ADDRESS     0x1d10e0 // rotary pio address

// analog PIO addresses
#define AUTO_TRIGGER_PIO 0x1d1150 // 1 bit data, 1 for auto triggering,
                                  //   0 for no auto triggering
#define TRIGGER_ENABLE_PIO 0x1d1140 // 1 bit data, 1 for trigger enable,
                                  //   0 if not enabled
#define TRIGGER_SLOPE_PIO 0x1d1130 // 1 bit data, 1 for negative slope,
                                  //   0 for positive slope
#define SAMPLE_DELAY_PIO 0x1d1120 // 16 bit data, sample delay (in # samples)
#define TRIGGER_LEVEL_PIO 0x1d1110 // 8 bit data for trigger level
#define SAMPLE_RATE_PIO 0x1d1100 // 19 bit data for sample rate (samples/sec)

// PIO addresses - FIFO
#define DATA_READY_PIO  0x1d10f0 // data ready signal, read clock input for fifo
#define FIFO_DATA_PIO   0x1d10c0 // 8 bit sample currently read from fifo
```

```
#define FIFO_FULL_PIO   0x1d10d0 // 1 bit, 1 if fifo full, 0 if not full

#define WD_PIO          0x1d10b0 // 1 bit output for watchdog timer

// other constants
#define VRAM_WIDTH  512         // length of VRAM row
#define NUM_SAMPLES 512         // number of samples in fifo
#define CLOCK_FREQ  24000000    // clock frequency, 24 MHz

#define ROT_IRQ     0           // rotary interrupt number

#define PTRNL      0            // null pointer reference
#define FALSE      0
#define TRUE       !FALSE

#endif
```

A Timing Diagrams

A.1 ADC

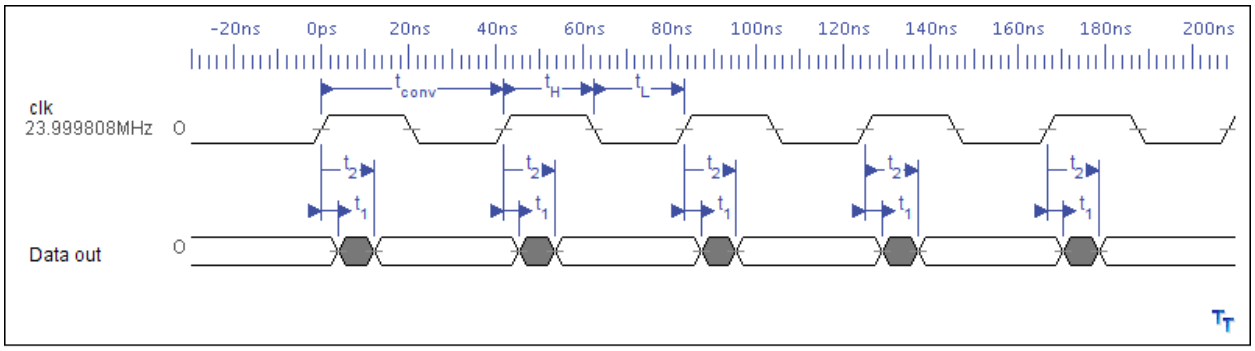

Figure 39: ADC Timing Diagram

A.2 LCD

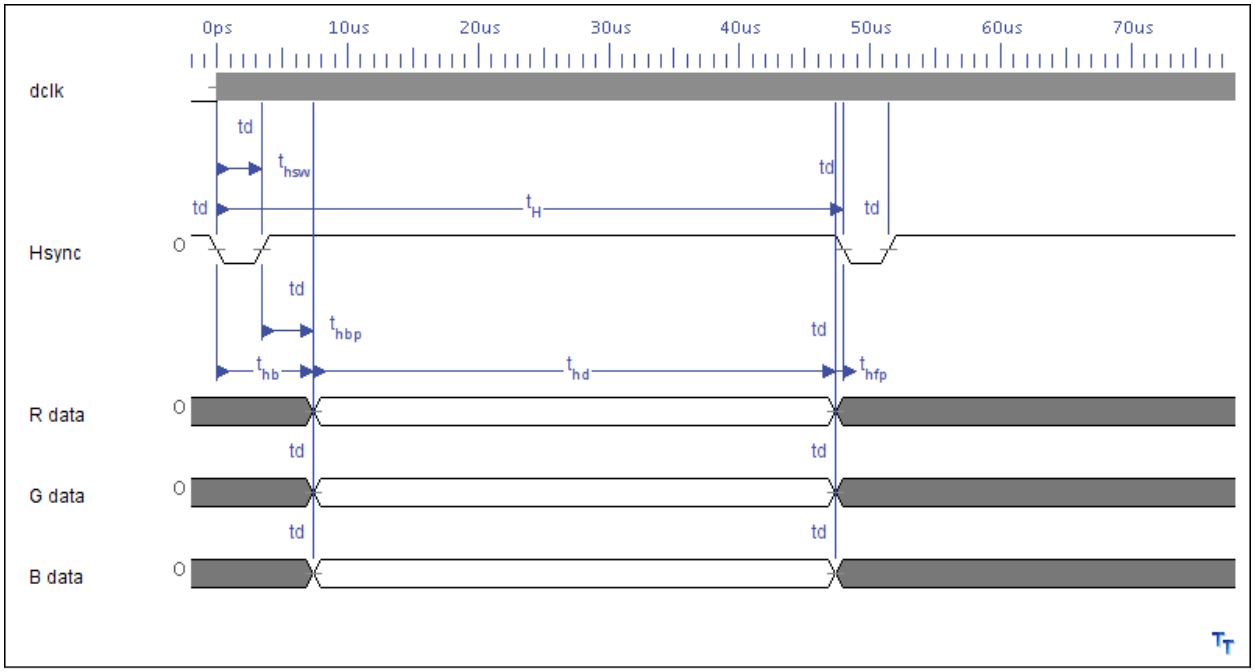

Figure 40: LCD Horizontal Timing Diagram

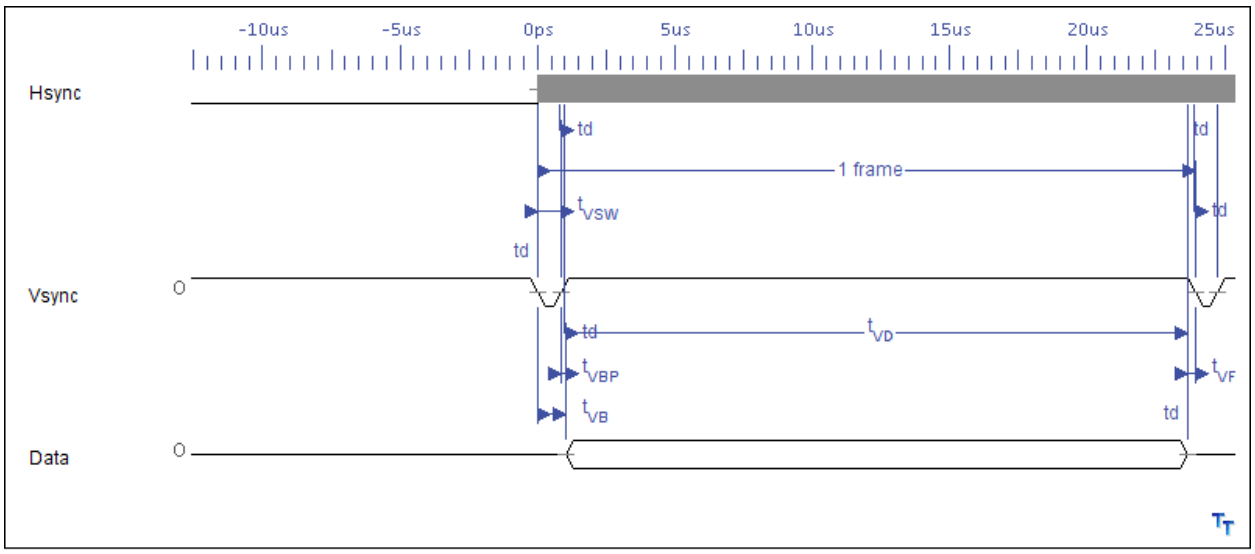

Figure 41: LCD Vertical Timing Diagram

A.3 ROM and RAM

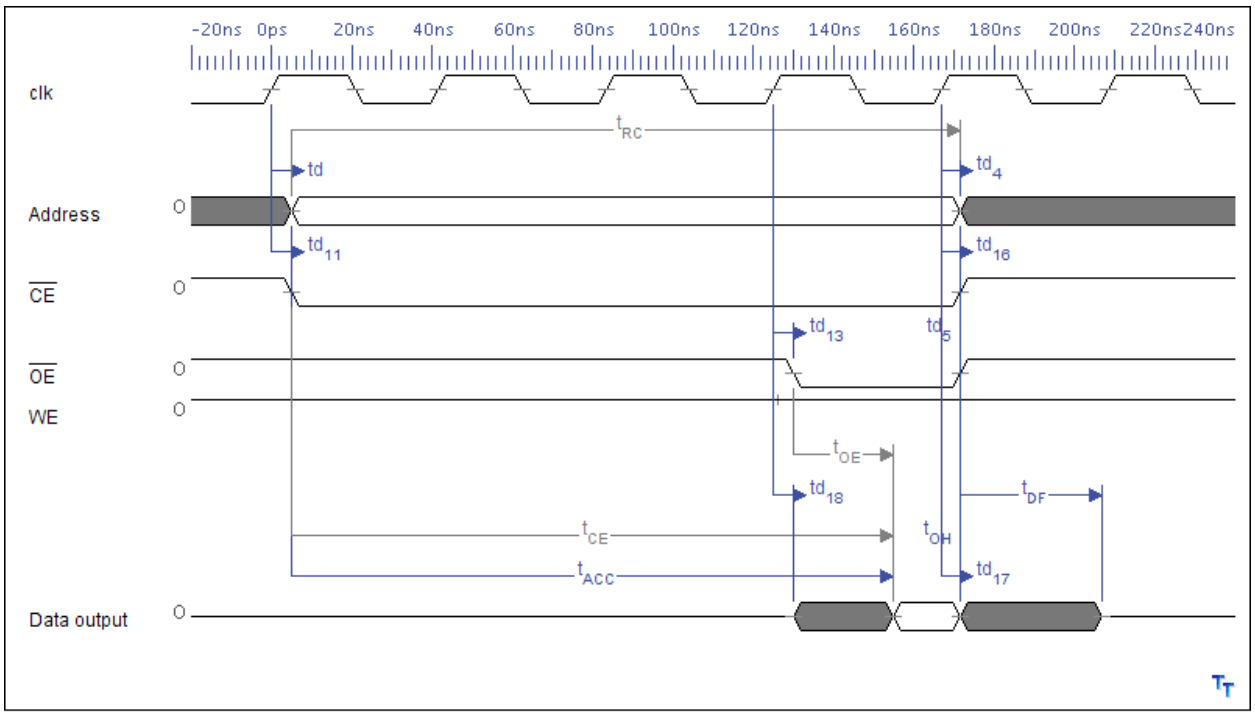

Figure 42: ROM Read Cycle Diagram

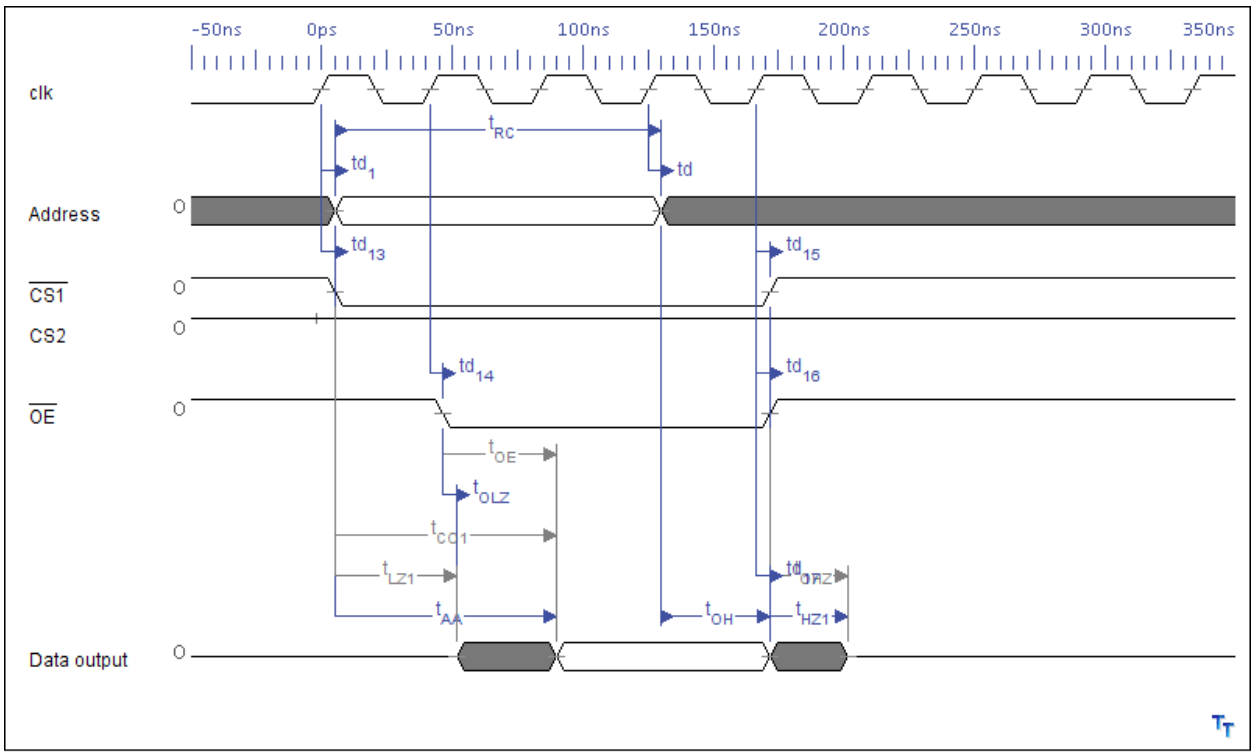

Figure 43: RAM Read Cycle Diagram

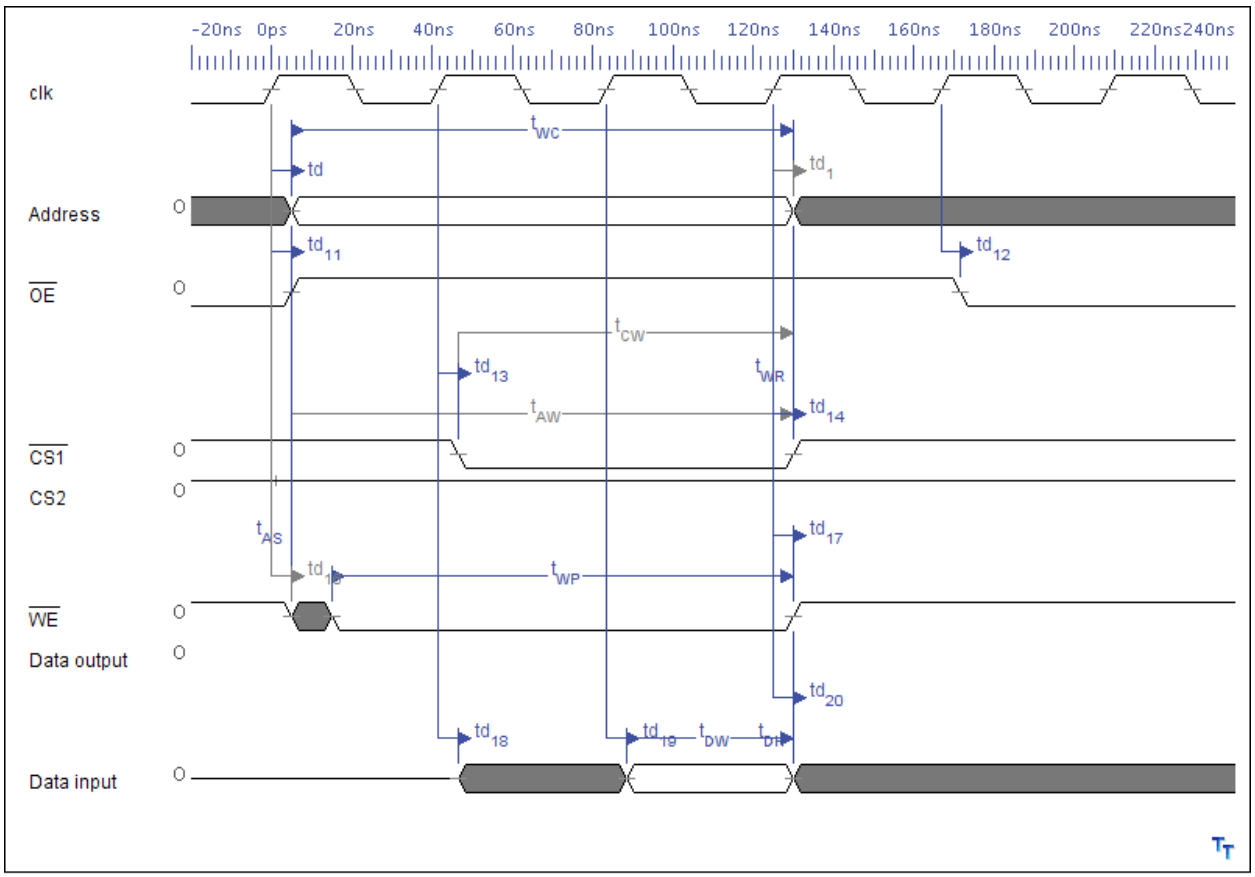

Figure 44: RAM Write Cycle Diagram

A.4 VRAM

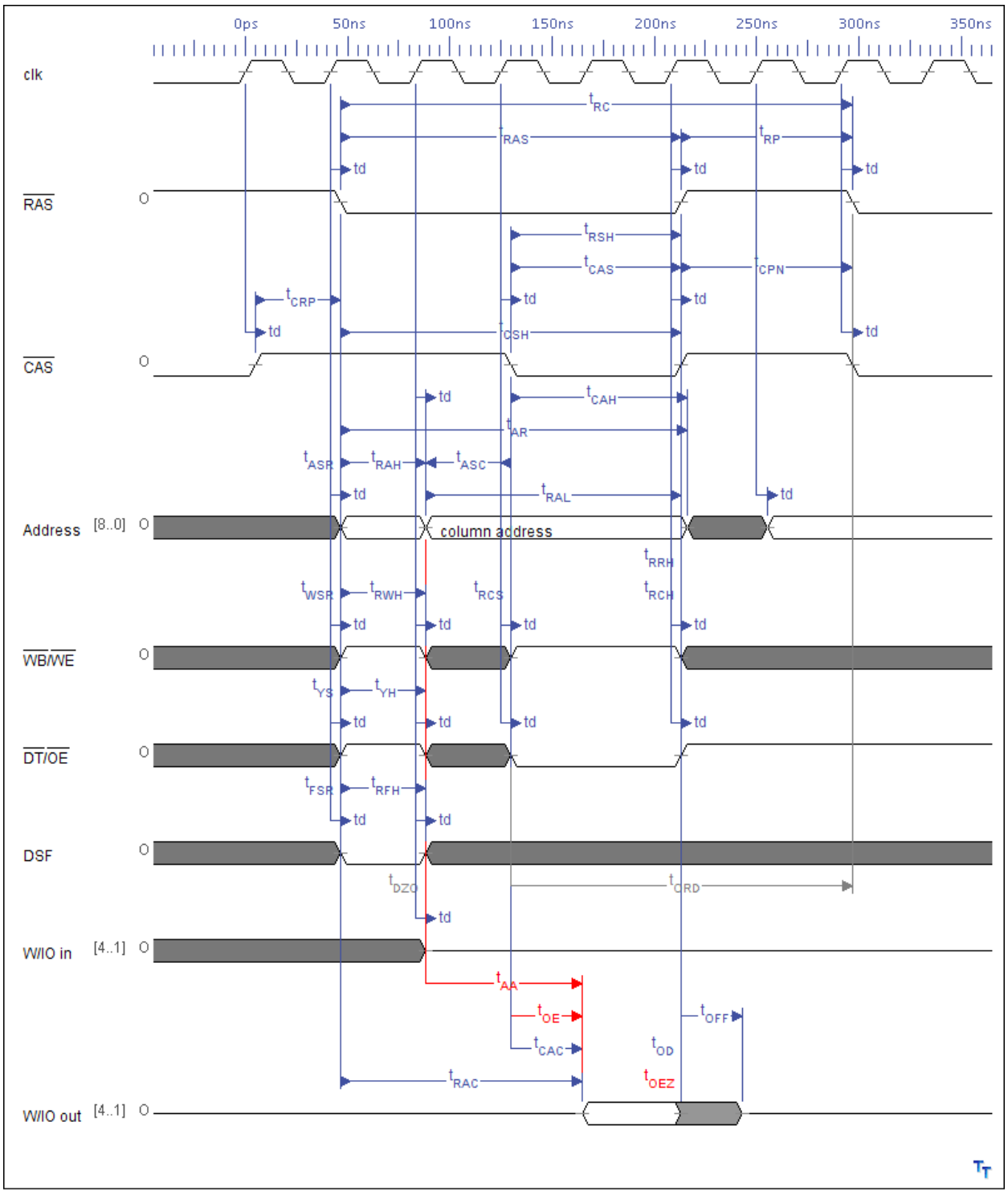

Figure 45: VRAM Read Cycle Diagram

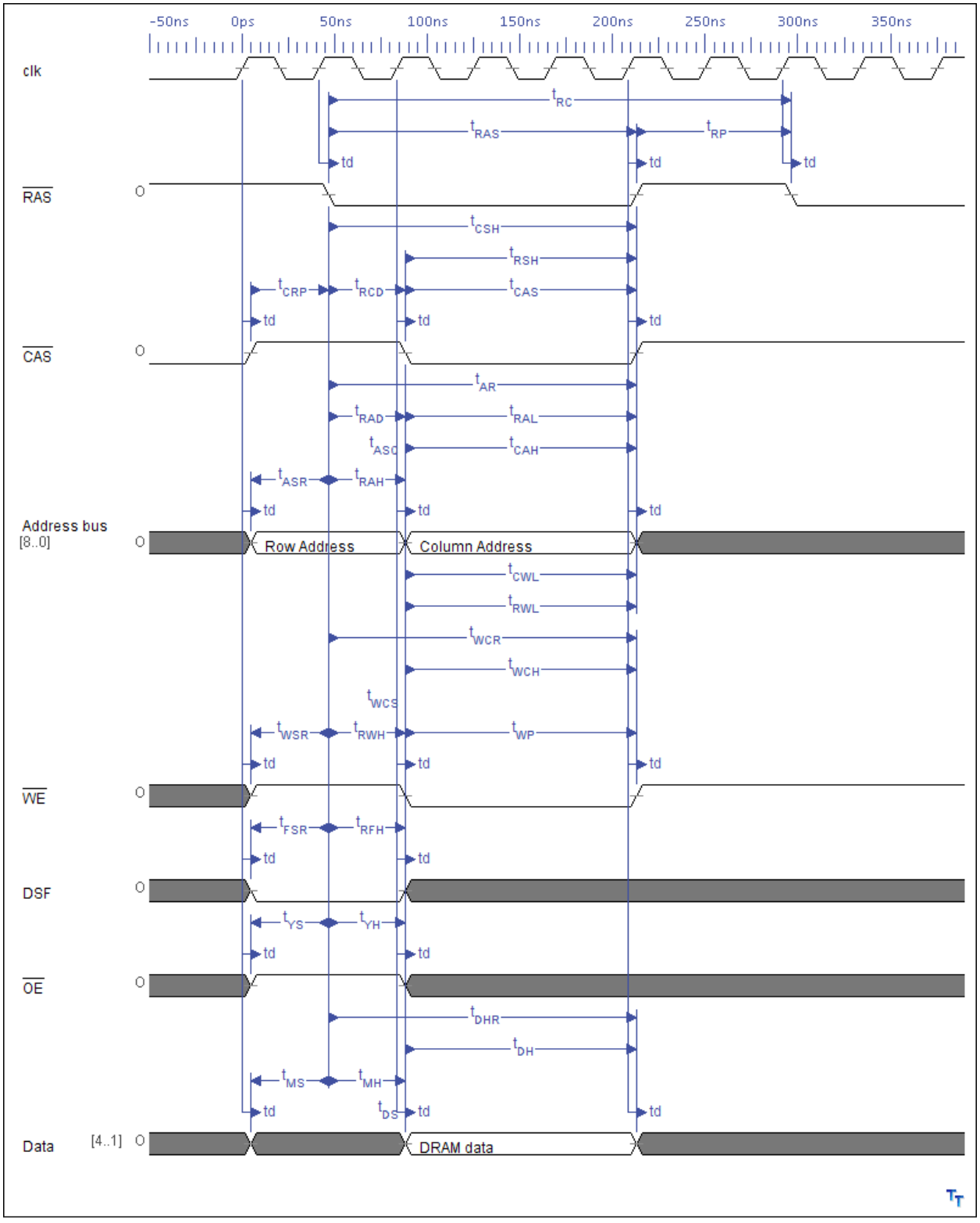

Figure 46: VRAM Write Cycle Diagram

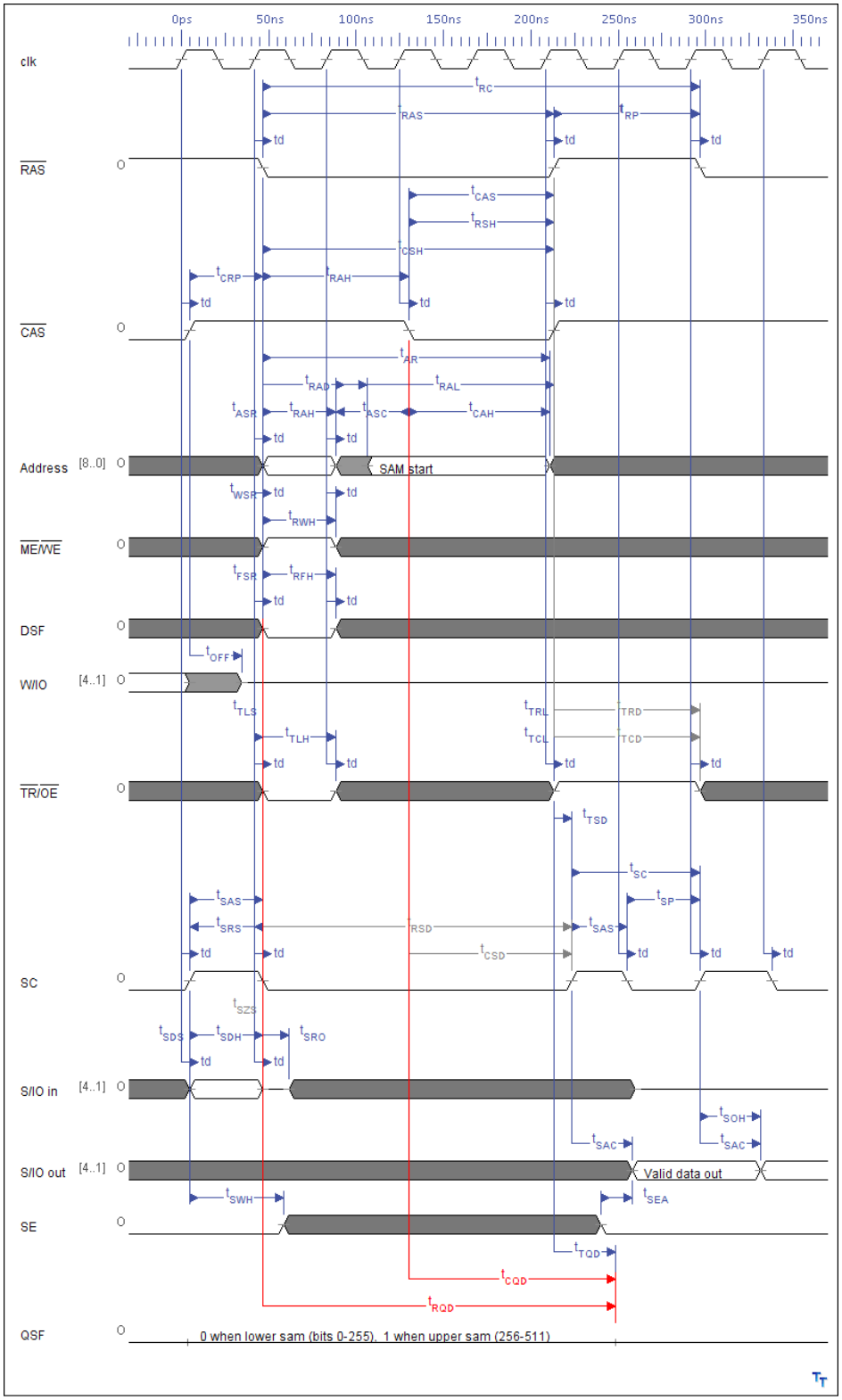

Figure 47: VRAM Row Transfer Cycle Diagram

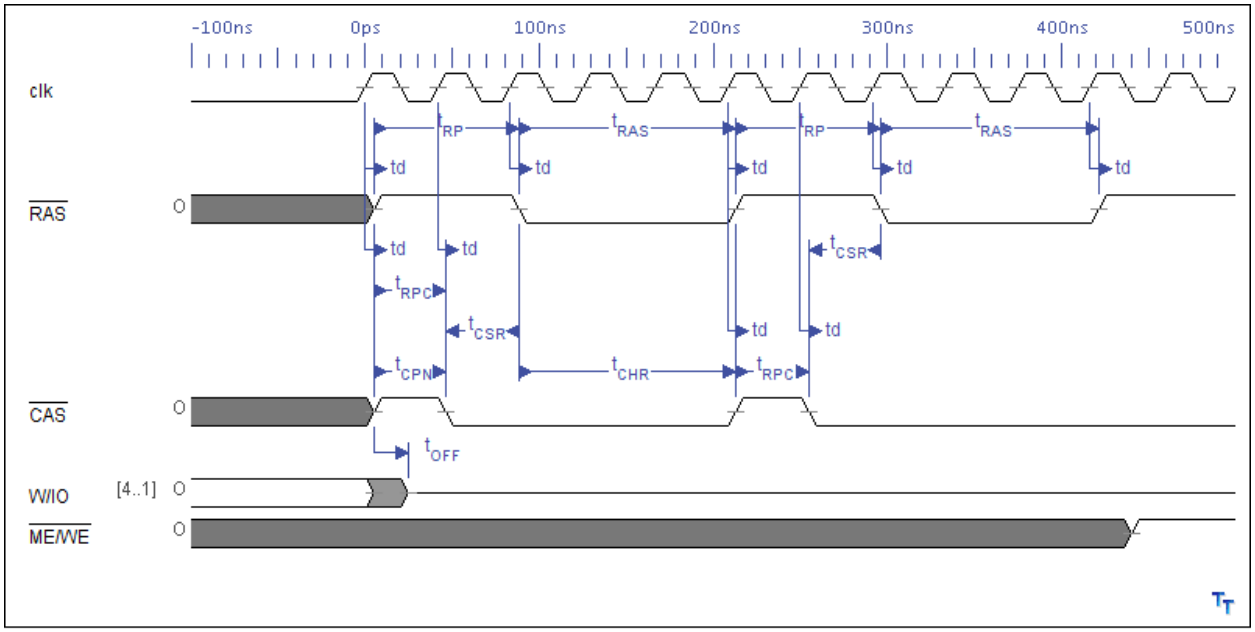

Figure 48: VRAM Refresh Cycle Diagram (CAS before RAS)

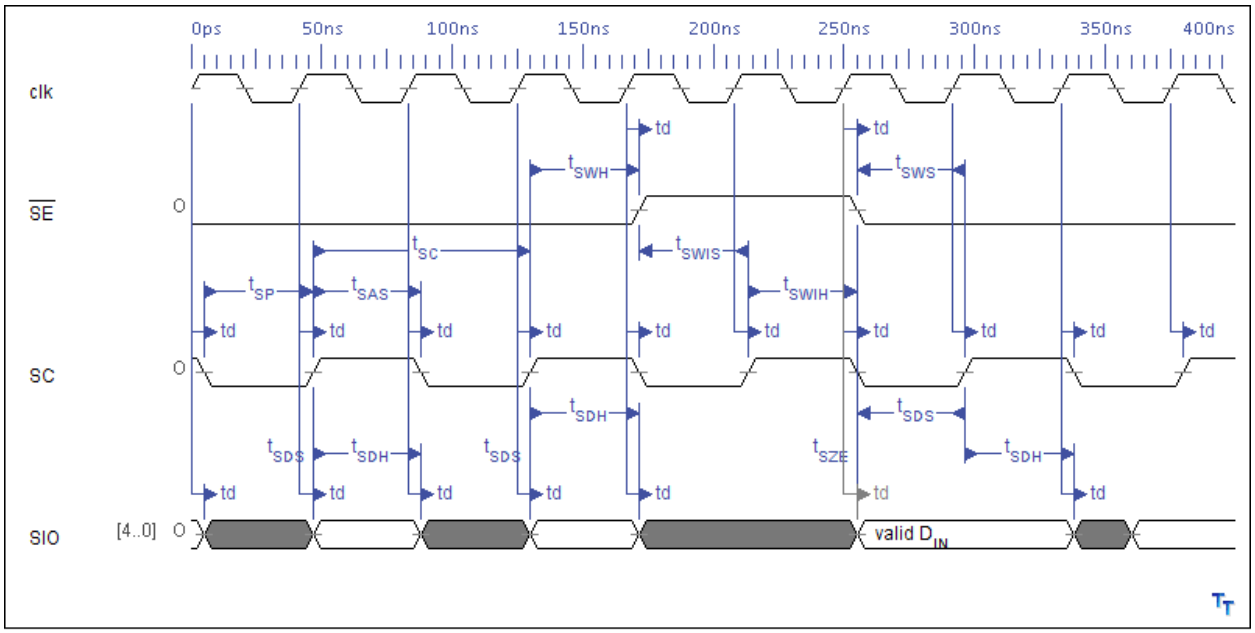

Figure 49: VRAM Serial Write Cycle Diagram

B Device Images

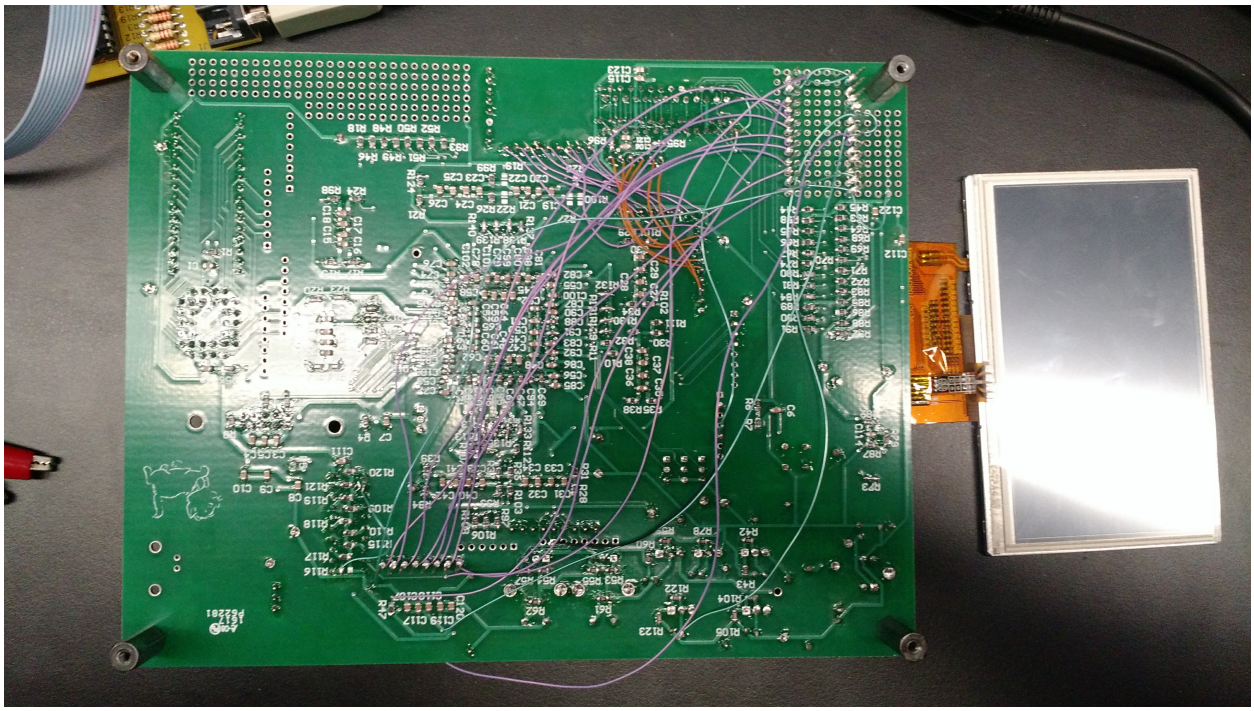

Figure 50: Front of device

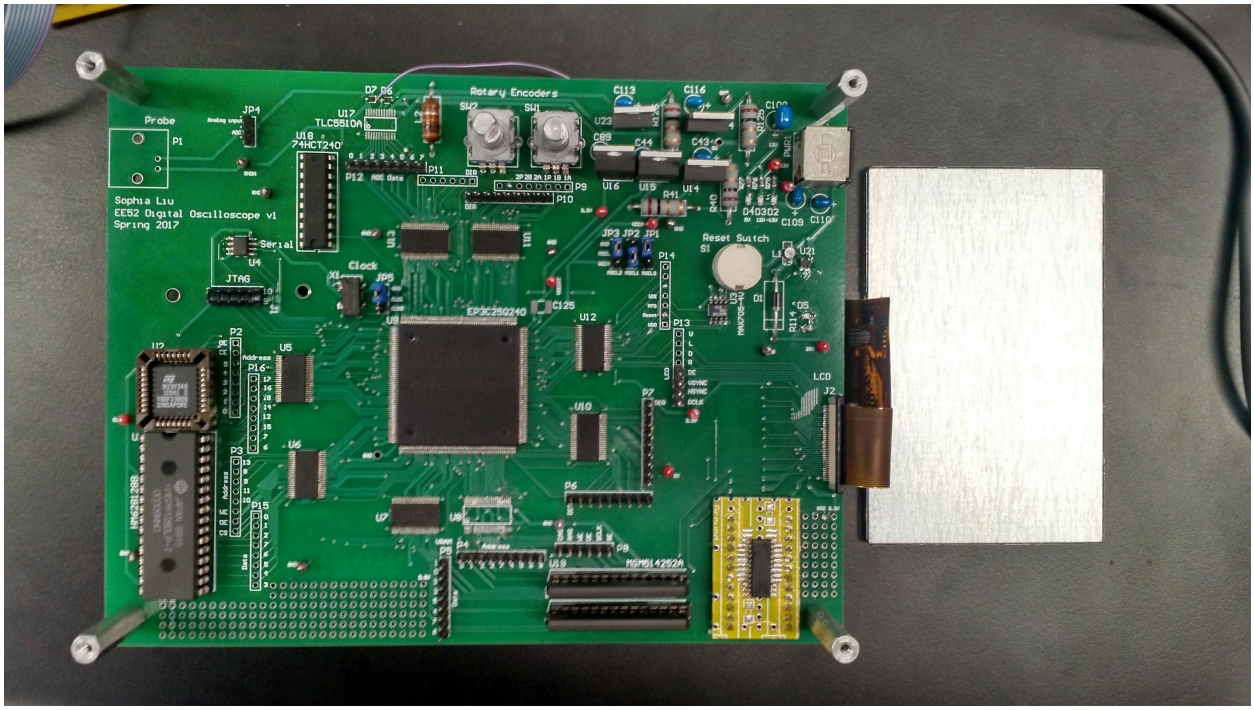

Figure 51: Back of device

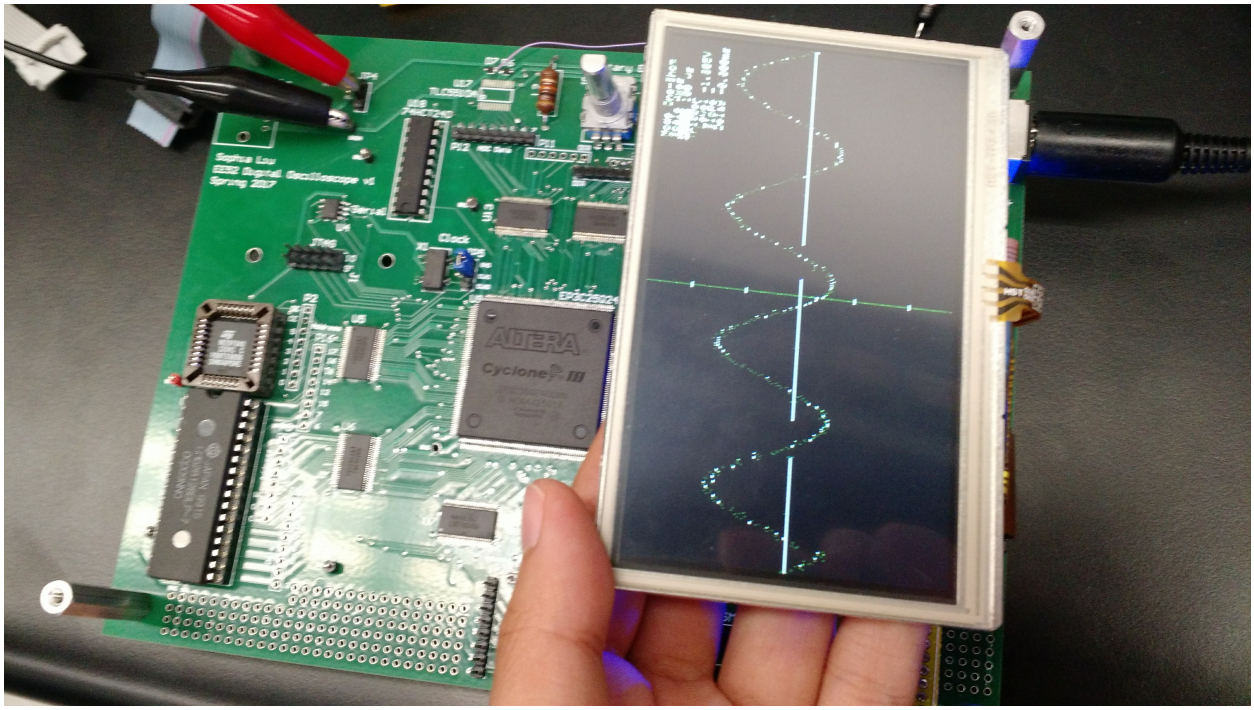

Figure 52: Device running with sinusoidal input

C Code

C.1 Hardware descriptions

Listing 6: data/scoptrig.vhd

```
--------------------------------------------------------------------------------
--
--  Oscilloscope Digital Trigger
--
--  This is an implementation of a trigger for a digital oscilloscope in
--  VHDL. There are three inputs to the system, one selects the trigger
--  slope and the other two determine the relationship between the trigger
--  level and the signal level. The only output is a trigger signal which
--  indicates a trigger event has occurred.
--
--  The file contains multiple architectures for a Moore state machine
--  implementation to demonstrate the different ways of building a state
--  machine.
--
--
--  Revision History:
--     13 Apr 04 Glen George     Initial revision.
--      4 Nov 05 Glen George     Updated comments.
--     17 Nov 07 Glen George     Updated comments.
--     13 Feb 10 Glen George     Added more example architectures.
--
--------------------------------------------------------------------------------


-- bring in the necessary packages
library ieee;
use  ieee.std_logic_1164.all;


--
--  Oscilloscope Digital Trigger entity declaration
--

entity ScopeTrigger is
    port (
        TS        :  in  std_logic;    -- trigger slope (1 -> negative, 0 ->
            positive)
        TEQ       :  in  std_logic;    -- signal and trigger levels equal
        TLT       :  in  std_logic;    -- signal level < trigger level
        clk       :  in  std_logic;    -- clock
        Reset     :  in  std_logic;    -- reset the system
        TrigEvent :  out  std_logic    -- a trigger event has occurred
    );
end  ScopeTrigger;
```

```
--
--  Oscilloscope Digital Trigger Moore State Machine
--     State Assignment Architecture
--
--  This architecture just shows the basic state machine syntax when the state
--  assignments are made manually. This is useful for minimizing output
--  decoding logic and avoiding glitches in the output (due to the decoding
--  logic).
--

architecture assign_statebits of ScopeTrigger is

    subtype states is  std_logic_vector(2 downto 0);  -- state type

    -- define the actual states as constants
    constant IDLE     : states := "000"; -- waiting for start of trigger event
    constant WAIT_POS : states := "001"; -- waiting for positive slope trigger
    constant WAIT_NEG : states := "010"; -- waiting for negative slope trigger
    constant TRIGGER  : states := "100"; -- got a trigger event


    signal CurrentState : states;   -- current state
    signal NextState    :  states;   -- next state

begin


    -- the output is always the high bit of the state encoding
    TrigEvent <= CurrentState(2);


    -- compute the next state (function of current state and inputs)

    transition: process (Reset, TS, TEQ, TLT, CurrentState)
    begin

        case  CurrentState is         -- do the state transition/output

            when  IDLE =>             -- in idle state, do transition
                if  (TS = '0' and TLT = '1' and TEQ = '0') then
                    NextState <= WAIT_POS;    -- below trigger and + slope
                elsif  (TS = '1' and TLT = '0' and TEQ = '0') then
```

```
            NextState <= WAIT_NEG;    -- above trigger and - slope
        else
            NextState <= IDLE;        -- trigger not possible yet
        end if;

    when WAIT_POS =>          -- waiting for positive slope trigger
        if (TS = '0' and TLT = '1') then
            NextState <= WAIT_POS;    -- no trigger yet
        elsif (TS = '0' and TLT = '0') then
            NextState <= TRIGGER;     -- got a trigger
        else
            NextState <= IDLE;        -- trigger slope changed
        end if;

    when WAIT_NEG =>          -- waiting for negative slope trigger
        if (TS = '1' and TLT = '0' and TEQ = '0') then
            NextState <= WAIT_NEG;    -- no trigger yet
        elsif (TS = '1' and (TLT = '1' or TEQ = '1')) then
            NextState <= TRIGGER;     -- got a trigger
        else
            NextState <= IDLE;        -- trigger slope changed
        end if;

    when TRIGGER =>           -- in the trigger state
        NextState <= IDLE;    -- always go back to idle

    when others =>
        NextState <= IDLE;

end case;

if Reset = '1' then           -- reset overrides everything
    NextState <= IDLE;        --   go to idle on reset
end if;

end process transition;


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

process (clk)
begin

if clk = '1' then             -- only change on rising edge of clock
    CurrentState <= NextState; -- save the new state information
```

```
        end if;

    end process;


end  assign_statebits;
```

C.2 Main software

Index