EE/CS 52 SoPC Digital Oscilloscope

Technical Manual

Santiago Navonne

June 2014

Contents

1. Introduction

This document describes the system workings and details of the EE/CS 52 System-on-Programmable-Chip (SoPC) Digital Oscilloscope.

The guide describes first the hardware, and then the software, giving for both a system overview, followed by a detailed description of every part of the system. In Appendix A, the schematics and printed circuit board used in the original prototype can be found.

2. Hardware

This section explains how the system's hardware works, from the system overview to the detailed description of each element. The interactions of components are described, and detailed schematics, timing diagrams, and board layouts are provided.

2.1. System Overview

The highest level illustration of the structure of the system is provided in the block diagram of Figure 1. The parts colored blue are created within the FPGA component (*U2*), while the parts colored red are outside components.

The central component of the system is the NIOS II CPU, a soft-core device generated within the FPGA. A NIOS II/f processor is used upon power-up, and can be upgraded to the faster NIOS II/s by connecting the device to a computer. Within the NIOS CPU are included the chip select decoding and interrupt control logic sections. The chip select decoding logic uses the address bus to activate the chip select control line of the device being accessed, if it requires one. The interrupt controller processes interrupt control signals from hardware devices and makes them available to software procedures.

The display controller, also included within the FPGA, controls the VRAM serial clock and all of the display timing signals, updating the VRAM serial data bus as needed to ensure that data is correctly shown on the display. The debouncers and decoders take signals from the user input sections of the system (i.e. rotary encoders and push-button switches), and process them to translate them into interrupt signals for the processor. These signals are accessed through a Parallel IO (PIO) interface. The triggering logic is configured through a PIO interface, and reads the signal output by the ADC, determining the correct moment to trigger based on triggering mode, level, slope, and delay parameters. The component then instructs the FIFO to start writing samples as necessary. The First-In First-Out (FIFO) data structure stores samples to be processed by the CPU. The FIFO starts being filled when instructed by the trigger, and once full transmits this signal to both the trigger, which disables the trigger signal, and the CPU, which goes ahead and processes the samples.

Outside the FPGA but closely related is the reset logic, which generates a reset signal for the NIOS CPU on power-up, power failure, and when requested by the user. Similarly, the clock logic is an outside component that generates a constant, 38 MHz clock signal used throughout the system. A JTAG connector and interface is used to program and debug the FPGA and NIOS CPU.

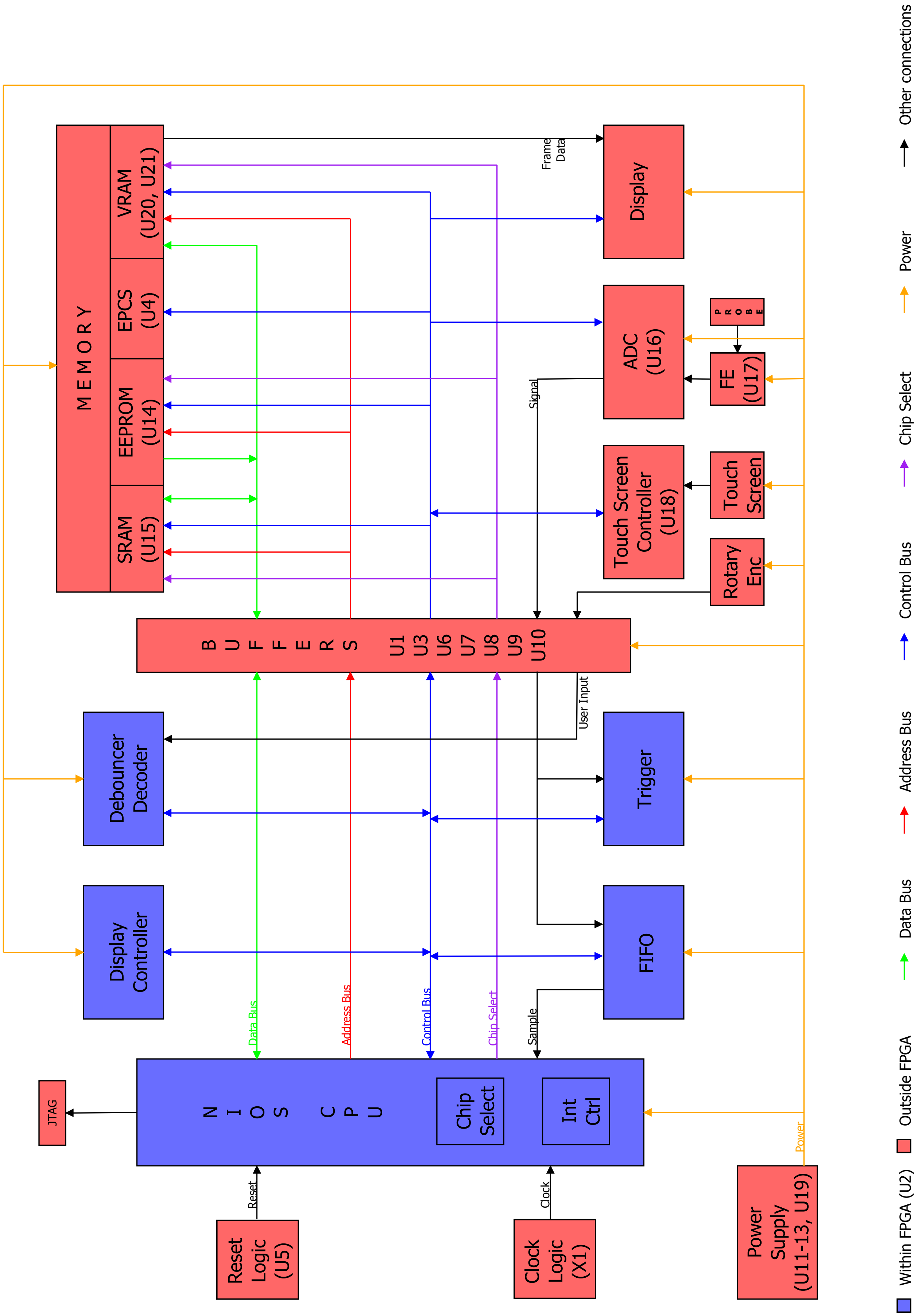


Figure 1: Highest level block diagram of the SoPC Oscilloscope system. The diagram is described in Section 2.1.

Every signal exiting the FPGA, with the sole exception of the I^{2}C bus, is buffered. These chips provide a layer of protection for the FPGA, as well as more driving power and voltage flexibility for the components being operated.

Four memory devices are used by the system. The Serial Configuration device (EPCS) is used to store the FPGA that is loaded upon start-up. The device uses dedicated serial control signals to communicate with the FPGA. The Static Random Access Memory (SRAM) device is the system's volatile memory, used to store the software's variables, stack, and other uninitialized memory. The device shares the data and address buses with the other memory devices, and is selected and controlled by a small set of exclusive signals from the CPU. The Electrically Erasable Programmable Read-Only Memory (EEPROM) device is the system's non-volatile memory, used to store code and constants. This device also shares the data and address buses with the other memory devices, and it is, too, selected and controlled by a small set of exclusive signals from the CPU. Two Video RAM (VRAM) devices are used as a buffer for the frames being shown on the display. Data is put there by the CPU via the VRAM controller, and subsequently extracted serially and shown on the LCD by the display controller. This device shares the data bus with other memory devices, while the address bus is exclusive from the controller to the device. A set of VRAM specific control signals is also exclusive to this device.

The display is controlled by timing signals from the display controller, which, synchronized with the VRAM controller, ensure that pixels are output over a dedicated bus between display and VRAM at the right moment to be shown in the correct region of the display.

An Analog Front-End (AFE) scales and shifts signals from the oscilloscope probe as needed to prepare them for input into the Analog-to-Digital Converter (ADC). This device, in turn, reads the samples and converts them into digital values, which are then directly relayed to the triggering logic and FIFO within the FPGA. The ADC is also clocked by a control signal routed through the FPGA.

Two rotary encoders with momentary push-button switches provide the user-input interface of the system: the devices are connected to the FPGA and then the decoders and debouncers, which filter process the signals before making them available to the CPU. Finally, a touch screen controller, currently unimplemented, communicates with the CPU over an I^{2}C bus and a dedicated interrupt line used to identify touch screen events. This line is made available to the CPU through a PIO interface. The controller uses an analog interface to drive and read the touch screen.

All these elements, after being designed in detail and their connections finalized, are physically placed on a printed circuit board. The front of the board is illustrated in Figure 2, where each section is highlighted in a different color: the FPGA and related components are colored red; the buffers are green; the memory devices are yellow; the power supply circuitry is blue; the analog interface is orange; the display connector is pink; and the rotary encoders are brown. The components without any highlighting are prototyping and debugging holes and pins, unused in the final design. The back of the board is shown in figure 3. Note that no components other than capacitors and resistors are placed on the back of the board, and thus nothing is highlighted.

The memory map of the system is shown in figure 4. The *JTAG* device is used for debugging purposes the; *trig_period*, *trig_level*, *fifo_data*, *fifo_full*, *fifo_delay*, and *trig_ctrl* are parallel IO devices used to interface with the triggering logic. Each of these locations in memory contains several registers necessary for the interface to function. *pio_0* is another parallel IO device, this time used to interface with the rotary encoders and push buttons. Finally, *ram*, *rom*, and *vram* are the memory devices described above. Note that only memory devices have assigned chip select signals, since the other components do not require them; furthermore, note that the VRAM chip select signal does not exist in hardware external to the FPGA.

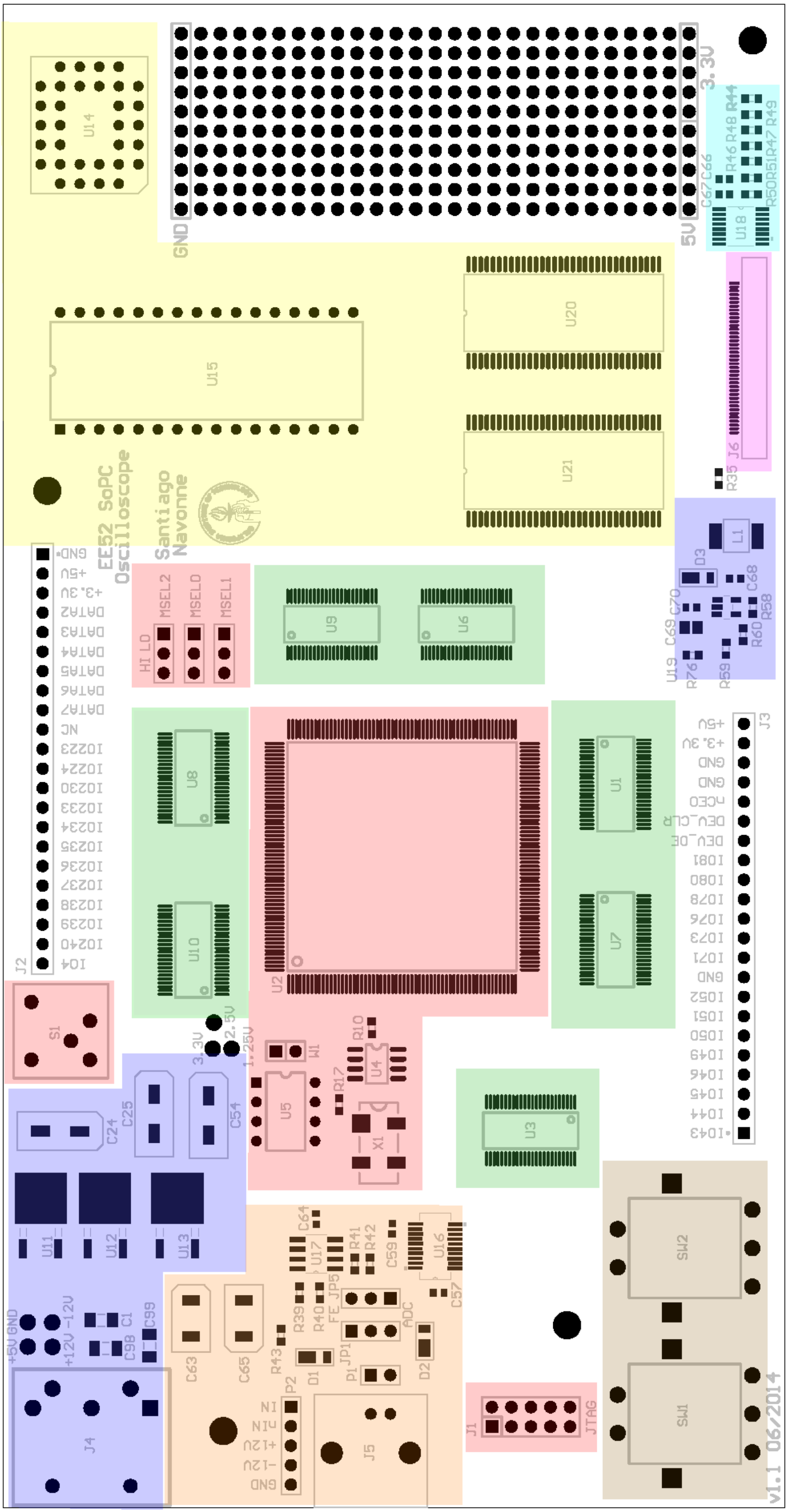


Figure 2: Front side of the system's Printed Circuit Board (PCB). The color each section is highlighted and identifies the corresponding block.

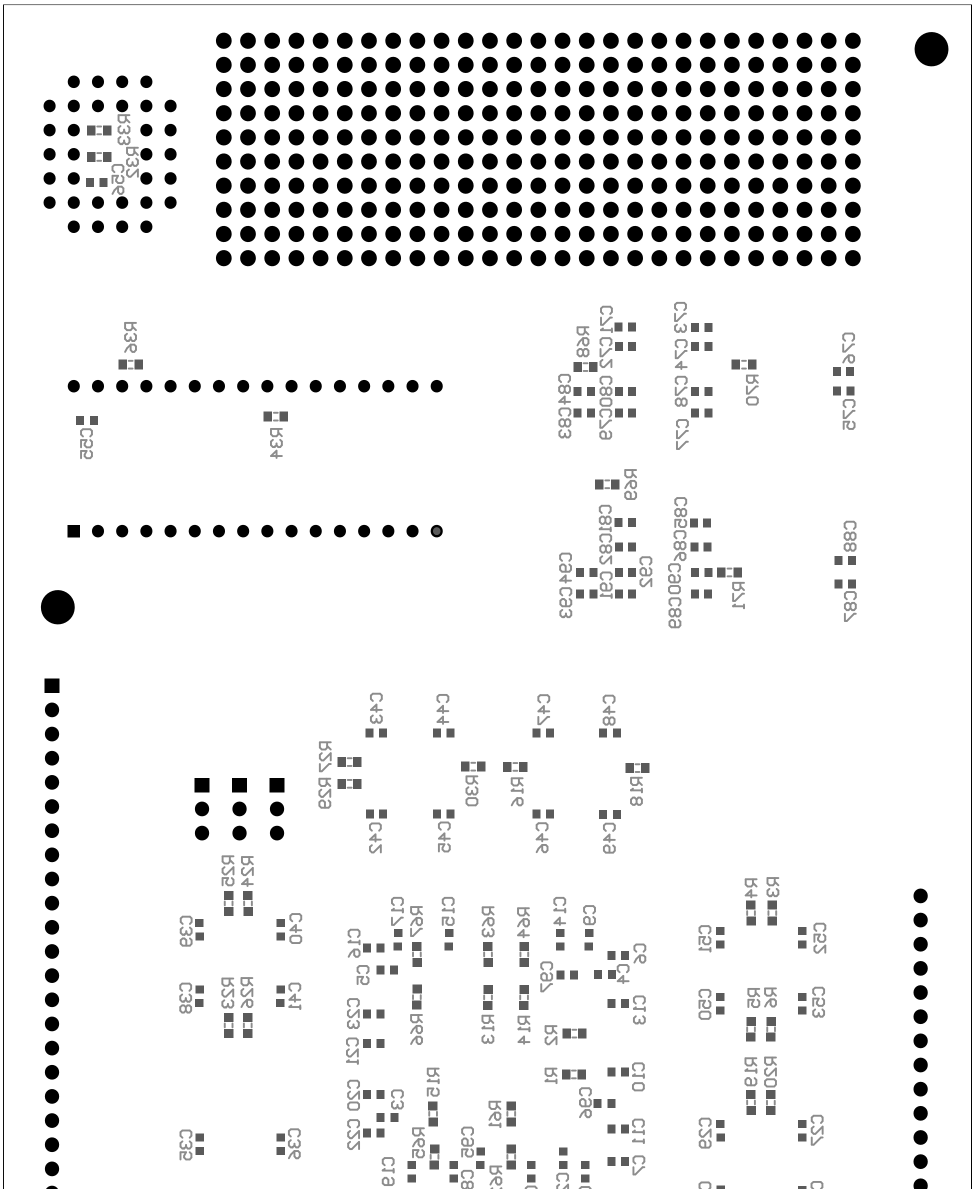
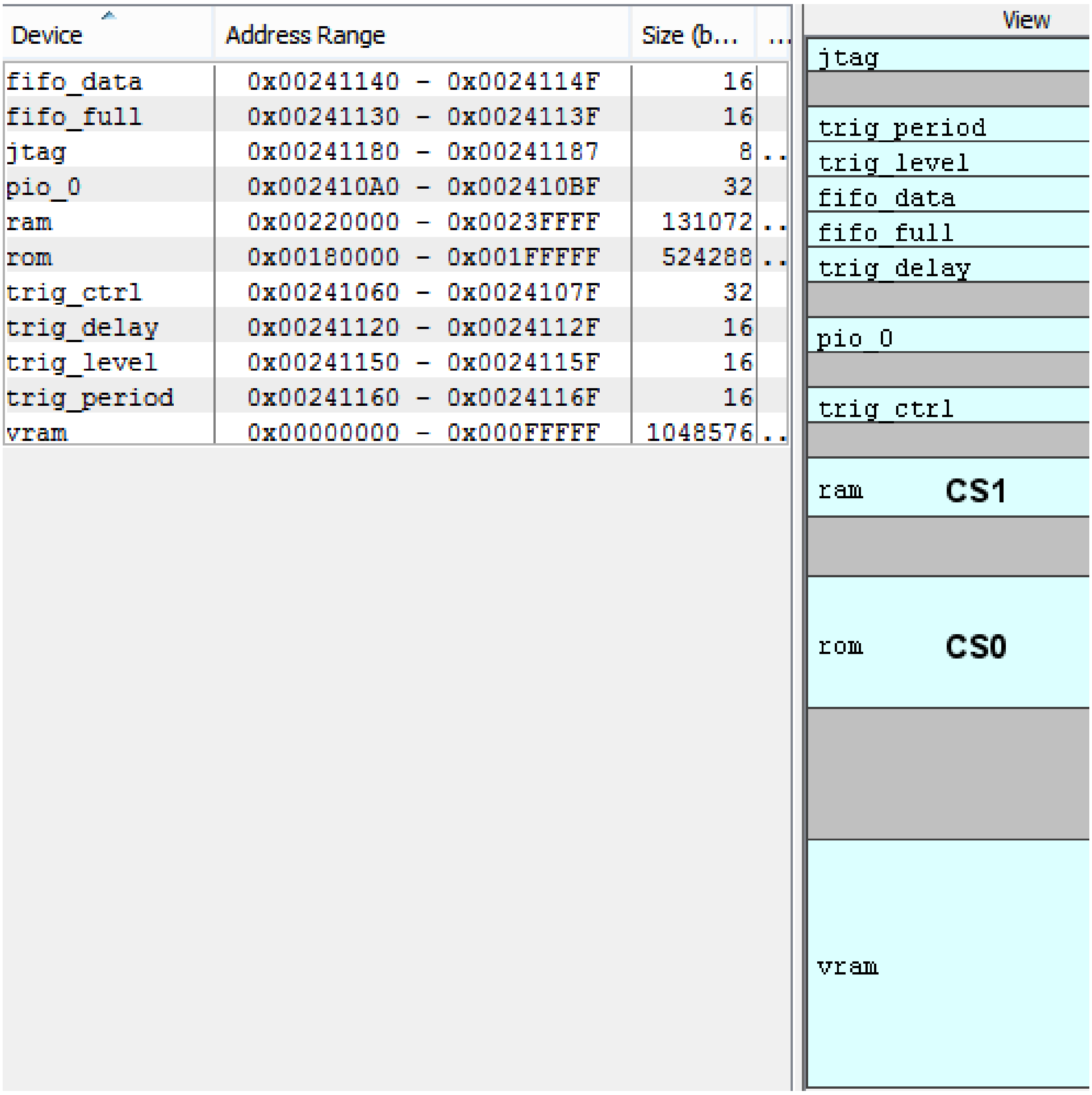


Figure 3: Back side of the system's Printed Circuit Board (PCB). Note that this side is only used for routing and placement of passive parts, such as resistors and capacitors.



Device	Address Range	Size (b...	...
fifo_data	0x00241140 - 0x0024114F	16	
fifo_full	0x00241130 - 0x0024113F	16	
jtag	0x00241180 - 0x00241187	8	..
pio_0	0x002410A0 - 0x002410BF	32	
ram	0x00220000 - 0x0023FFFF	131072	..
rom	0x00180000 - 0x001FFFFF	524288	..
trig_ctrl	0x00241060 - 0x0024107F	32	
trig_delay	0x00241120 - 0x0024112F	16	
trig_level	0x00241150 - 0x0024115F	16	
trig_period	0x00241160 - 0x0024116F	16	
vram	0x00000000 - 0x000FFFFF	1048576	..

Figure 4: Memory map of the system within the NIOS processor, with associated chip select lines and addresses.

2.2. FPGA

An Altera EP3C25Q240 Cyclone III Field Programmable Gate Array (FPGA), *U2*, is the central unit of the system. The component and its associated parts are at the center of the PCB, highlighted in red in Figure 2. Its connections at a system's level are illustrated in the schematic of Figure 5. The device is programmed and debugged through a JTAG interface. After debugging, the final design is loaded upon power-on from the serial memory device *U4* through lines *DATA0* and *DATA1*, clocked by *DCLK* and configured through *nCSO*; the debugging and design loading configuration is determined by the *MSEL2..0* jumpers *JP2*, *JP3*, and *JP4*; for the correct operation of the device, *MSEL2* and *MSEL0* should be jumped on the HI position, while *MSEL1* should be configured to the LO position. Pins 100 and 103 are pulled high through *R1* and *R2* to act as the I^{2}C bus lines *SDA* and *SCL*. Configuration lines *INIT_DONE*, *CONF_DONE*, and *nSTATUS* are pulled high, while *nCE* and *CLKUSR* are tied low. Configuration lines *DEV_OE*, *DEV_CLRn*, and *nCEO* can be left floating, and are therefore connected to break-out pins in *J3*.

The device contains the system's CPU, as well as all of the logic needed to process analog signals, debounce keys, decode rotary encoders, control the VRAM, and control the display. The logic was designed using the Altera tool chain: Quartus and QSys.

Within the FPGA, several components interact with one another. Figure 6 illustrates these components and their interactions.

The top left section of the diagram constitutes the user input section. Rotary encoder channels A (*ROT1A*, *ROT2A*) and B (*ROT1B*, *ROT2B*), as well as the push-button lines (*PUSH1*, *PUSH2*), for both devices, are input on pins 57, 63-65, 68, 69. The inputs are processed through debouncers (*DBC1*, *DBC2*) and decoders (*DEC1*, *DEC2*), which filter the signals, generating events as appropriate at their outputs. These events are collected into a bus, that is input to the NIOS processor (*CPU1*) at the *switches_in* port of *PIO_0*.

The middle left section of the diagram is the devices triggering logic. The signal from the ADC (*SIG7..0*) is input on pins 9, 13, 18, 21, 37-39, 41; and then connected to the *DATA7..0* input of the triggering block (*TRIG1*). The trigger's general clock (*GCLK*) is connected to the system clock input at pin 31 (*CLK*). All the other lines of the trigger block are connected to their PIO counterpart at the processor. Note that all the control lines (*SLOPE*, *AUTO_TRIG*, *FIFO_WE*, *READ*, *RESET*) are collected into a single bus that is then connected to the processor's *trig_ctrl* PIO interface. The ADC's sample clock on pin 72 (*ACLK*) is clocked at a constant 38 MHz through the system clock (*CLK*).

The bottom left block, *SR1*, is a necessary component for the functioning of the system's serial memory device (*U4*).

In the bottom right, we recognize the VRAM (*VRAM1*) and display (*DISP1*) controllers. These components' inputs are connected to the processor (*CPU1*), with the exception of the interconnected serial row update request (*UREQ*) and acknowledge (*UACK*) signals. The VRAM receives its own dedicated address bus shifted right twice to turn 4-byte addressing into single-word addressing (*vaddr_out19..2* to *A17..9* and *A8..0*). The dedicated chip select (*CS*) and shared write enable (*WE*) signals are operated as for any other generic controller. All clocks are connected to the system clock. Finally, these two components output all the necessary timing signals for the display and VRAM devices on pins 82, 83, 87, 88, 93-95, 98. Additionally, the VRAM controller outputs a *RDY* signal, converted to an active-high *WAIT* signal for the processor through inverter *NOT*, to regulate the VRAM access cycles, which use a variable number of wait states.

In the top right, we see the system's NIOS processor (*CPU1*). Apart from the previously described connections, the devices shared address (*ADDR_BUS18..0*) and data (*DATA_BUS23..0*) buses for the ROM and RAM devices are output on pins 114, 117-120, 126-128, 131-135, 137, 139, 142, 143, 146-148, 160, 161, 166, 167, 181-189, 194-197, 200-203, 217. The write enable signal (*WE*) is output to the RAM on pin 168, and is also used as the direction signal for the data bus buffers (*U6*, *U9*) on pin 207 (*DATA_DIR*). The RAM and ROM's dedicated chip select signals (*CS1*, *CS0*) are output on pins 171 and 173, respectively.

Finally note that I^{2}C bus lines *I2C_SDA* and *I2C_SCL* on pins 103 and 100, as well as the *PENIRQ* input on pin 70, are left unconnected since the touch screen interface remains unimplemented. Also note that the display enable line (*DISP*) is tied high to permanently enable the device, and that buffer *U10* is configured as always enabled, output by pulling *U10_DIR* on pin 214 and *U10_OE* on pin 216 low.

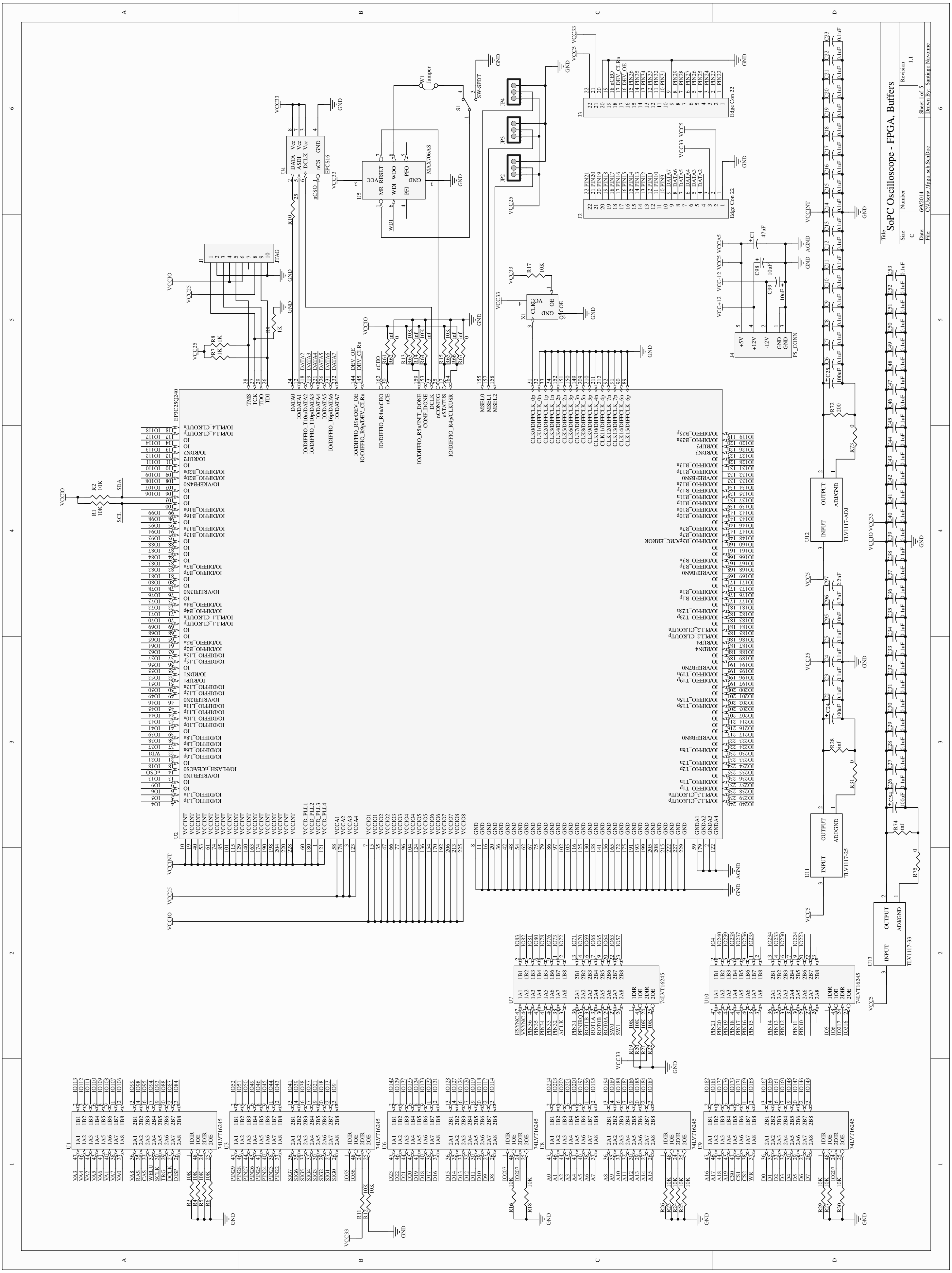


Figure 5: Schematic of the FPGA and related components. Also included are the buffers and power supply. The document is described in Section 2.2.

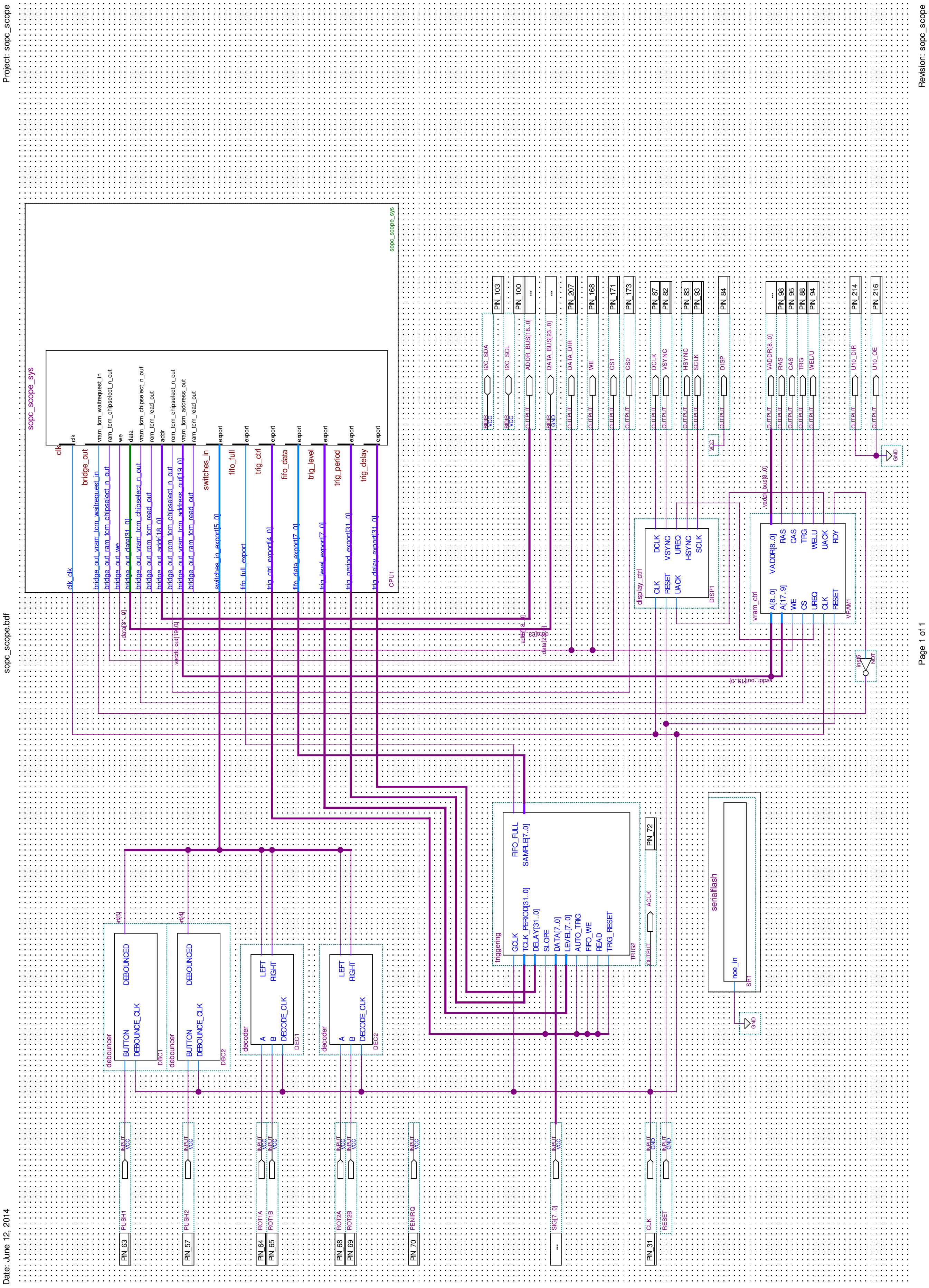


Figure 6: Main block diagram of the FPGA's internal design. The document is described in Section 2.2.

2.2.1. NIOS Processor

The NIOS processor used in the project is generated with Altera's QSys. Table 1 summarizes all the components included within the synthesized device.

Note that the Interrupt Controller and Chip Select blocks are automatically implemented by the Altera tool chain, and are therefore not described in this document; these are compiled together with the SoPC design in QSys.

Device	**Type**	**Description**
clk_0	Clock Source	System's main clock, generated at 38 MHz.
nios	NIOS II Processor	NIOS II/f Central Processing Unit, host of all of the system's software.
ram	Generic Tri-State Controller	RAM device controller, used to store volatile data such as variables and the stack.
rom	Generic Tri-State Controller	ROM device controller, used to store non-volatile data such as code and constants.
vram	Generic Tri-State Controller	VRAM device controller, used as a frame buffer to output data to the display.
pin_sharer	Tri-State Conduit Pin Sharer	Component that combines all data and address lines into a single bus.
bridge	Tri-State Conduit Bridge	Device that outputs the memory devices' shared and exclusive lines to FPGA pins.
jtag	JTAG UART	Debugging interface for standard output.
pio_0	PIO (Parallel I/O)	6-bit debounced and decoded rotary encoder and push button inputs. Bits 0 and 1 are the left and right rotation events of rotary encoder 2. Bits 2 and 3 are the left and right rotation events of rotary encoder 1. Bits 4 and 5 are the push-button press events of rotary encoder push-buttons 2 and 1, respectively. All of these bits generate interrupts on their falling edges.
trig_period	PIO (Parallel I/O)	Output for the 32-bit sampling period of the analog interface (time between samples, in number of 38 MHz clock cycles). Individual bit set and clear registers are disabled for this component.
trig_level	PIO (Parallel I/O)	Output for the configured level, as an 8-bit value where 0 is the most negative value, and 255 the most positive value. Individual bit set and clear registers are disabled for this component.
fifo_data	PIO (Parallel I/O)	Input for the current 8-bit sample being output by the FIFO data structure.
trig_ctrl	PIO (Parallel I/O)	Output for the triggering logic control values. Bit 0 is an active-high auto trigger enable signal; bit 1 is the slope bit, 1 for negative slope and 0 for negative slope; bit 2 is the active-low write enable signal for the FIFO, which is enable when a sample is started and disabled when it is completed; bit 3 is the FIFO's read clock; bit 4 is an active-high reset signal for the triggering logic. Individual bit set and clear registers are enabled for this component.
fifo_full	PIO (Parallel I/O)	1-bit input for the interrupt signal indicating that the FIFO is full. Interrupts are generated on rising edges of the line.
trig_delay	PIO (Parallel I/O)	Output for the 32-bit trigger delay value, in number of sample times where the minimum valid value is 1. Note that the value to be output is actually the desired value minus one. Individual bit set and clear registers are disabled for this component.

Table 1: Central Processing Unit and related devices configured within the QSys part of the FPGA design.

2.2.2. Triggering and FIFO

The triggering logic and FIFO of component *TRIG1* acquire samples from the analog interface as instructed by the processor, and then make them available on a serial data structure until the CPU is ready to process them. The block structure of this component is illustrated in figure 7.

The very top section of the diagram divides the system clock (*GCLK*) down to the sample clock requested by the CPU. The requested value is input as the 32-bit duration of the sample clock in number of system clock cycles (*CLK_PERIOD31..0*). A counter (*CT1*) counts system clocks, and the output count is then compared in *CMP1* to the required period divided by two (shifted right twice in *W1*): when the count is less than the compare value, the output of *CMP1* is high; when the count is greater, the output is low. The counter is cleared by *CMP4* when the count reaches the clock period or when the triggering logic is reset (*G8*), effectively generating a 50

Manual trigger events are generated in the middle section of the block diagram. Here, the sample from the ADC (*DATA7..0*) is sent through three chained delayed flip-flops (TDFF2) to remove any glitches, and then compared to the desired trigger level (input at *LEVEL7..0*). *CMP2* thus generates the *TL* and *TEQ* signals required by the ScopeTrigger state machine, instantiated in *TRIG1*. This component, which also takes the slope (*SLOPE*), sample clock, and reset signal (*RESET*), generates a trigger event (*TrigEvent*) when the input signal intersects the desired trigger level with the requested slope. The component is part of the EE/CS 52 library, and its code is provided in the next few pages for reference.

```vhdl
----------------------------------------------------------------------------
--
--  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.
--     01 Mar 14  Santiago Navonne  Removed unnecessary 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;

```

```
end  assign_statebits;
```

Trigger events generated by *TRIG1* are then sent through the delay logic. Since the events only last one clock (i.e. only a pulse is generated), *TrigEvent* is used to set J/K flip flop *FF2*. The output of *FF2* enables a counter (*CT2*) that counts sample clocks. When the number of sample clocks from the trigger event reaches the requested 32-bit delay (*DELAY31..0*), *CMP3*'s output goes high. This output clears *FF2*, disabling the counter until the next trigger event, and *CT2*, resetting the counter. Note that *CT2* is also reset on *RESET* events. *CMP3*'s output is therefore a single sample-clock long pulse that is low at all times, except for *DELAY31..0* sample clocks after a *TrigEvent*.

The bottom section of the diagram uses the so far generated delayed trigger events and other settings from the CPU to correctly fill the FIFO (*FIFO1*). When automatic triggering is disabled (*AUTO_TRIG* low), counter *CT8* gets constantly cleared and is therefore "bypassed." If writing to the FIFO is disabled because no sample has been started (*FIFO_WE* high), J/K flip-flop *FF1* will be cleared, and its output, *FIFO1*'s *wrreq*, will be disabled; no data will thus be written to the FIFO. If writing to the FIFO is enabled (*FIFO_WE* low), *FF1* will be set through *G4* whenever a delayed trigger event (output of *CMP3*) is received. Once *FF1* is set, *wrreq* becomes enabled, and samples from the ADC (*DATA7..0*), sent through three additional delayed flip-flops (*TDFF1*) for pipelining, are written to the FIFO. When the FIFO becomes full, *G5* clears *FF1*, disabling writing to *FIFO1*. The FIFO full signal also acts as an interrupt for the processor through output *FIFO_FULL*, prompting it to read the completed sample. The sample is read by first disabling writing (sending *FIFO_WE* low), and then bit-banging the read clock (*READ*). Since the read enable line (*rdreq*) is permanently enabled, every time the *READ* line transitions from low to high a new sample is output from *FIFO1* onto *SAMPLE7..0*. *RESET* signals clear *FIFO1*.

When automatic triggering is enabled (*AUTO_TRIG* high), the logic described above still applies with one addition: counter *CT8* is enabled as long as writing to the FIFO is enabled too (*FIFO_WE* low), and the FIFO is not currently being written to (input to *wrreq* low). When the counter reaches 380,000, a timeout designed to count 10 ms on the 38 MHz system clock, comparator *CMP12* transitions to high, causing the FIFO to start being written to (*wrreq* sent high). This in turns disables counting in *CT8*, which causes the output of *CMP12* to stay high until writing is disabled (that is, until the samples are read). This mechanism effectively forces the generation of a trigger 10 ms after a sample is started, if no regular trigger was generated before then.

In a typical interaction, the processor will configure all triggering settings, and send the *FIFO_WE* line low to start the sample. When a sample is completed, the *FIFO_FULL* line will exhibit a rising edge. The processor must thus disable the *FIFO_WE* line (send it high), and clock the *READ* line appropriately to extract all 512 samples from *FIFO1*. Any time settings are changed, the processor should reset the triggering logic by pulsing the *RESET* line high. The line may be maintained high while setting are being changed.

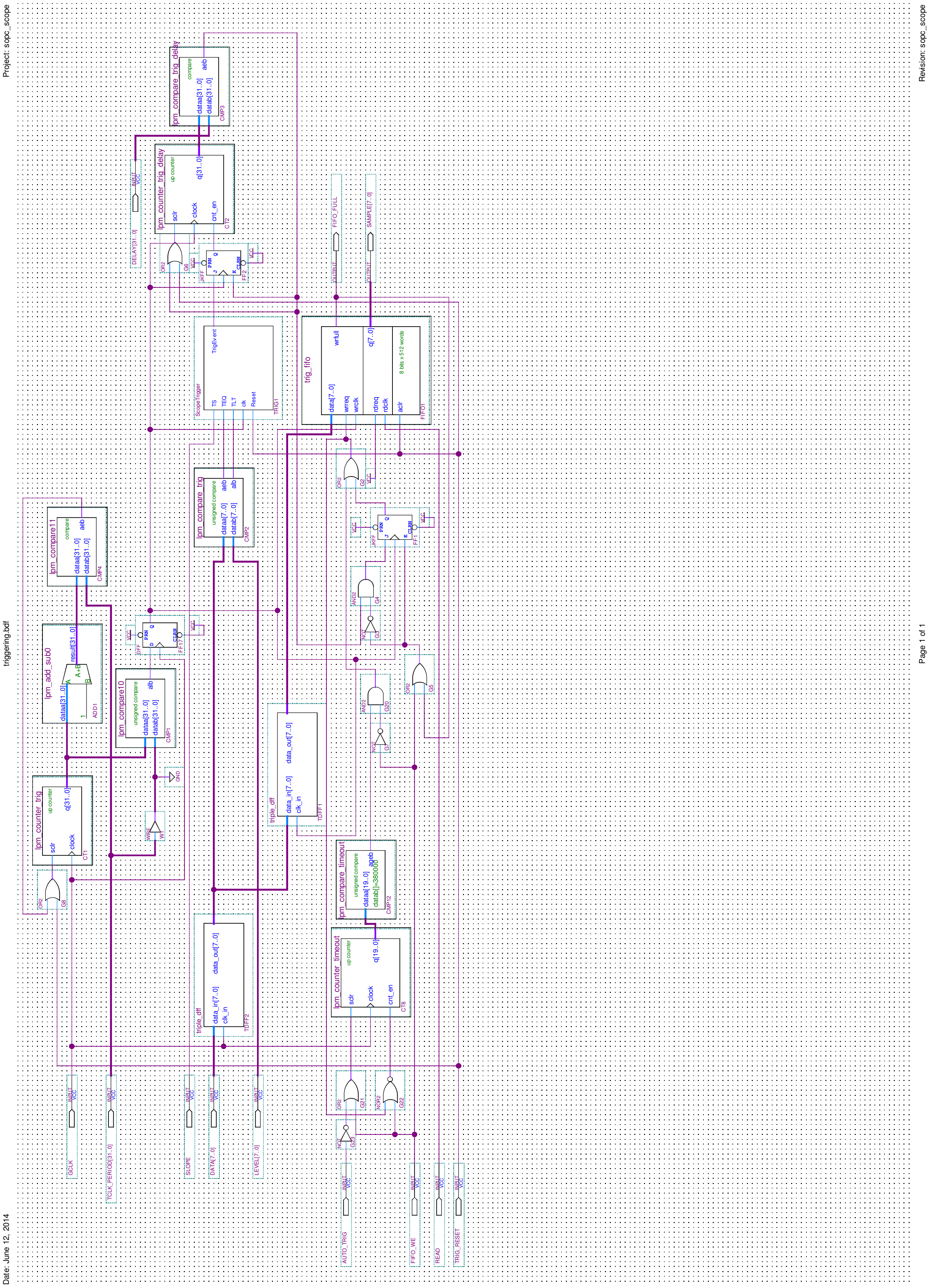


Figure 7: Block diagram of the triggering logic within the FPGA. The document is described in Section 2.2.2.

2.2.3. Debouncer

Two debouncers (*DBC1*, *DBC2*) are used to filter the input from the two rotary encoder push-buttons. Figure 8 illustrates the structure of the debouncer component.

The component takes the signal from the push-button, *BUTTON*, and a debouncing clock, *DEBOUNCE_CLK*, as inputs. *BUTTON* is assumed to be active-low to support pulled-up switches that are grounded upon activation.

As long as *BUTTON* is high, counter *CT3* will keep getting cleared, and its count enable line will be active; the counter will therefore not count. When *BUTTON* goes low, the counter will start counting *DEBOUNCE_CLK*. When this value reaches 380,000, the output of *CMP4* will go high. The compare constant was chosen to generate a 10 ms delay on the 38 MHz clock used in the system, and input into *DEBOUNCE_CLK*. *CMP4*'s output is then inverted to create an active-low signal that is used to prevent *CT3* from counting (and therefore wrapping around and deboucing the signal again) and output on line *DEBOUNCED*.

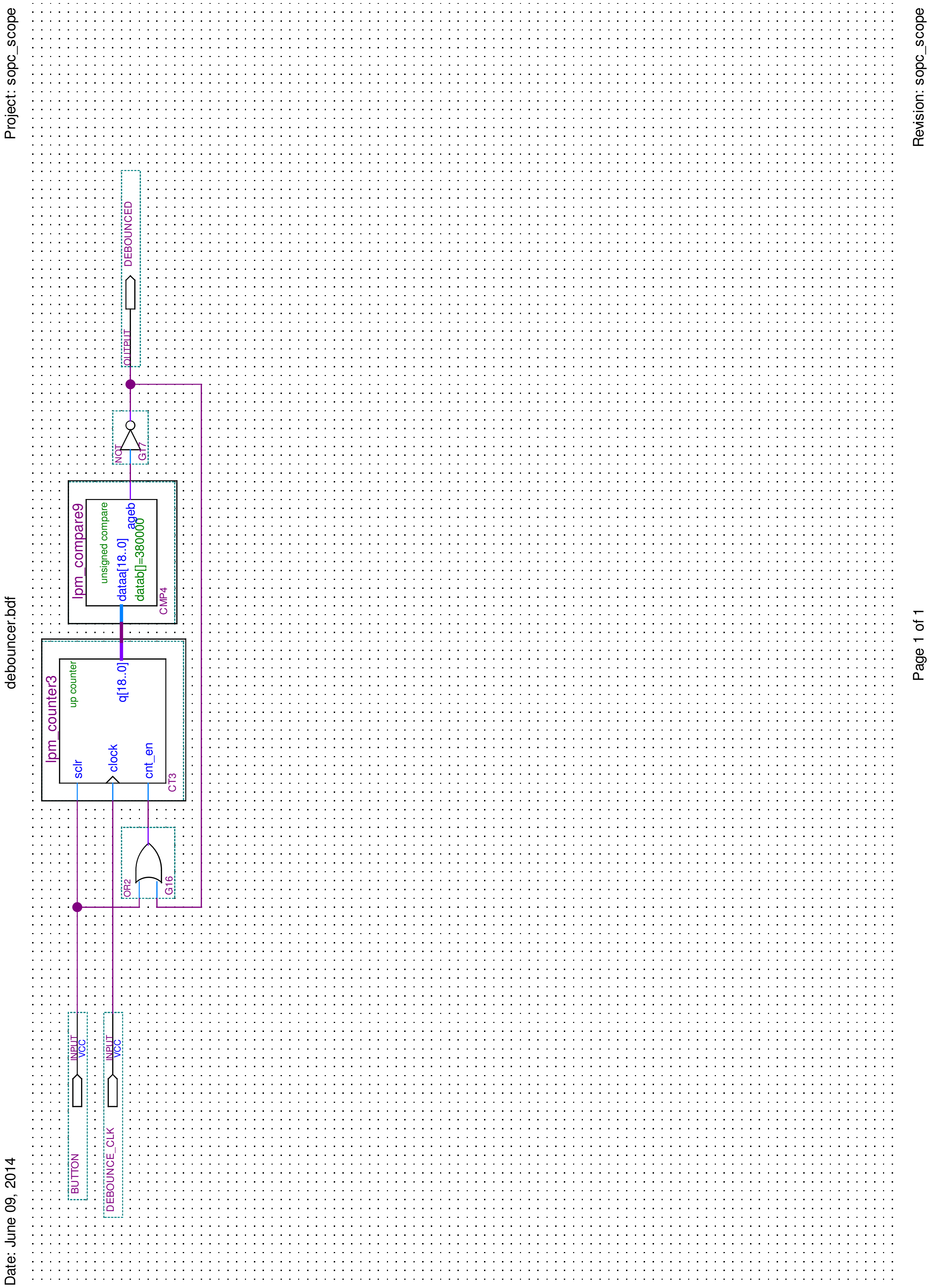


Figure 8: Block diagram of the push-button debouncer component within the FPGA. The document is described in Section 2.2.3.

2.2.4. Decoder

Two decoders (*DEC1*, *DEC2*)are used to decode the input from the two rotary encoders' rotation. Figure 9 illustrates the structure of the decoder component.

The component takes the *A* and *B* signals from the rotary encoder, which is assumed to have detents only on *A* and *B* active (high), and a decoding clock, *DECODE_CLK*, as inputs. The bottom part of the block diagram generates an enable signal, while the top part determines the direction of rotation.

To determine whether the encoder was turned (i.e. to generate an enable signal "clock"), an S/R flip-flop, *FF4*, is used. *FF4* is set when both *A* and *B* are high, that is when the encoder finds itself at a detent. *FF4* is reset when both *A* and *B* are low, that is when the encoder is between detents. Since rotary encoders only bounce between adjacent positions, the set and reset signals on *FF4* will not bounce. The output of *FF4* is an active-high enable signal.

To determine the direction of rotation, *A* is XOR'd in *G9* with the previous clock's *B*, saved through delayed flip-flop *FF3*. The output of *G9* will be high if the encoder was turned clockwise, and low if the encoder was turned counter-clockwise, due to the order in which positions occur within the encoder. The output is then fed to delayed flip-flop *FF5* for pipelining, and to delayed flip-flop *FF6*, which is clocked on the above described enable signal, to latch the direction only when a detent is reached. The output of *FF6* is thus directly NAND'd with the enable signal in *G13* to generate the active-low clockwise rotation interrupt, *RIGHT*, and inverted through *G12* and then NAND'd with the enable signal in *G15* to generate the active-low counter-clockwise rotation interrupt, *LEFT*. Note that *RIGHT* and *LEFT* will be high most of the time, and exhibit a falling edge when the encoder is turned in the corresponding direction.

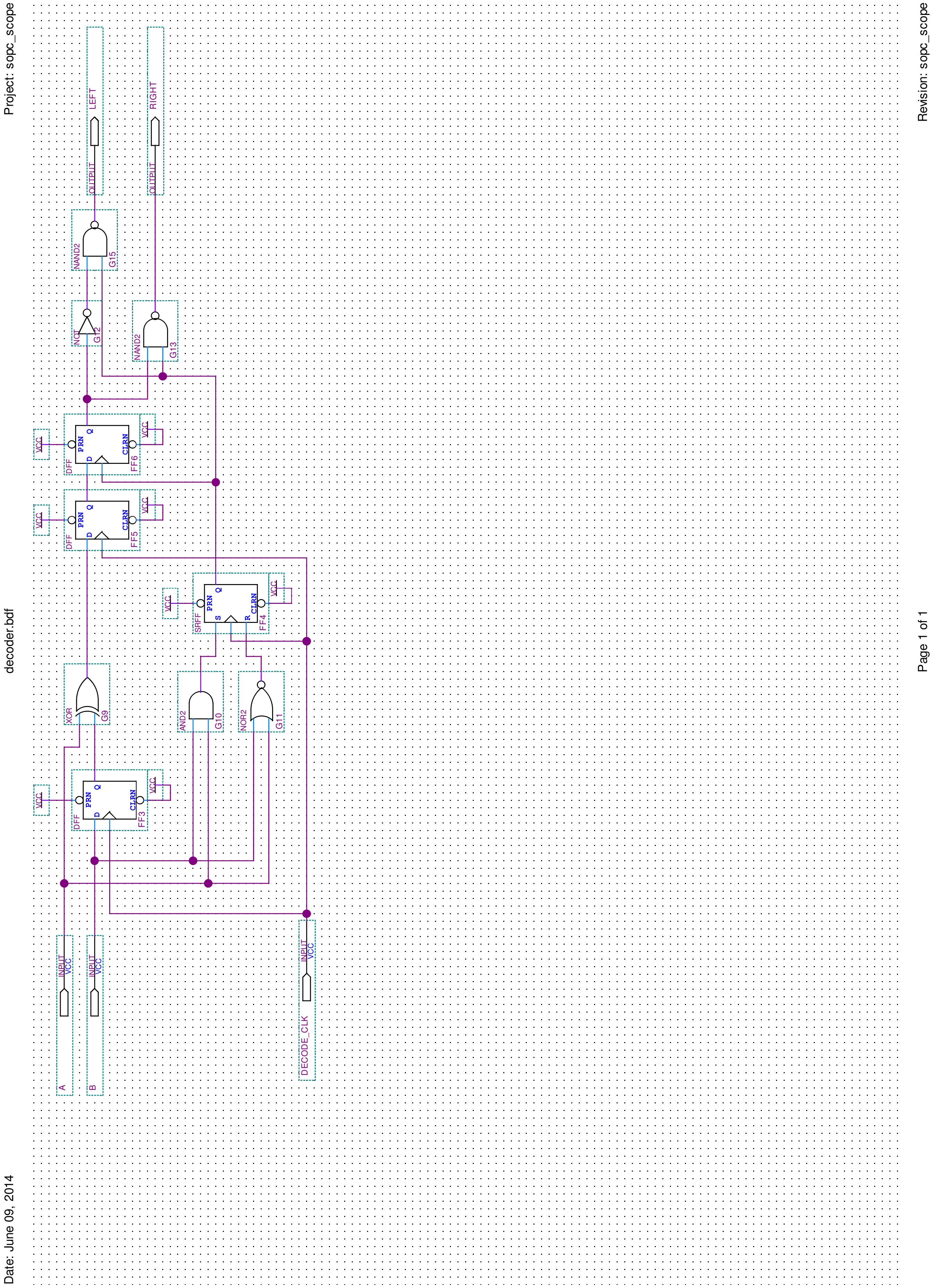


Figure 9: Block diagram of the rotary encoder decoder component within the FPGA. The document is described in Section 2.2.

2.2.5. VRAM Controller

The VRAM controller mediates interactions between the processor and the Video RAM. The processor can thus interact with the controller as if it were a regular memory device with variable wait states (i.e., an access is completed when the *WAIT* line goes low), and the controller generate the signals actually needed by the VRAM chips. Additionally, the VRAM controller performs row updates when requested by the display controller. A generic block diagram of the interactions between processor, VRAM controller, and display controller can be seen in Figure 10. The VRAM controller is illustrated in more detail in the block diagram of Figure 11.

The component takes a 18-bit address bus, *A[17..0]*, an active-low write enable signal, *WE*, an active low chip select signal, *CS*, an active high serial update request signal, *UREQ*, an active high reset signal, *RESET*, and a clock, *CLK*.

The address is divided into a row address, *A[17..9]*, and a column address, *A[8..0]*. These are muxed, together with a row address generated by *CT4* and a blank column address, in *MUX1*, allowing the VRAM control to output the correct address onto the VRAM address bus, *VADDR[8..0]*, as needed. *CT4* counts every time a new row transfer is requested, and wraps around the number of rows in the display, 272: this effectively causes the row address used for row updates to sequentially go through the whole display.

The bulk of the controller's logic is implemented in VHDL, using a Moore state machine. The next few pages provide the code of this state machine component. The state machine starts in the idle state (*IDLE*), where all the output signals are maintained at their neutral levels. If a row update is being requested (*UREQ* active), the state machine transitions into a row update cycle (*SERIAL1..6*) to ensure that the display controller is provided with a new row of data before it needs to start outputting at the end of its row porches. If no row update is being requested, but a read or write is being requested (*CS* active), the state machine transitions into the corresponding read (*READ1..6*) or write (*WRITE1..5*) cycle. If no memory access is being requested either, the controller performs a memory refresh by transitioning into the refresh (*REFRESH1..6*) cycle. Note that after each cycle, the state machine transitions back into the idle state.

```
----------------------------------------------------------------------------
--
--  SoPC Oscilloscope VRAM Controller
--
--  Implementation of the VRAM Controller for the SoPC Oscilloscope project.
--  The state machine generates the necessary timing signals for the VRAM
--  based on the needs of the CPU and display controller. Additionally,
--  it refreshes the VRAM as necessary whenever no other cycle is being
--  performed.
--  The inputs to the system determine what action needs to be performed:
--  cs+we requests a read or a write, while ureq requests a SAM row update.
--  The system then outputs the necessary timing signals, and a rdy/uack
--  signal to notify the sender of the end of the cycle.
--  The state machine is implemented using a Moore state machine and a state
--  assignment architecture.
--
--
--  Revision History:
--     13 Apr 04  Glen George       Initial template.
--     20 Feb 14  Santiago Navonne  Initial revision.
--
----------------------------------------------------------------------------


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


--
--  Oscilloscope VRAM Controller entity declaration
--

entity  VRAMCtrl  is
    port (
        we       :  in  std_logic;       -- read / not write
        cs       :  in  std_logic;       -- chip select
        ureq     :  in  std_logic;       -- serial row update request
        clk      :  in  std_logic;       -- clock
        Reset    :  in  std_logic;       -- reset the system
        ras      :  out  std_logic;      -- RAS timing signal
        cas      :  out  std_logic;      -- CAS timing signal
        trg      :  out  std_logic;      -- transfer/read signal
        welu     :  out  std_logic;      -- write signal
        asrc     :  out  std_logic;      -- address source selection
        arow     :  out  std_logic;      -- address row/column selection
        uack     :  out  std_logic;      -- serial row update acknowledge
        rdy      :  out  std_logic       -- read/write acknowledge
    );
end  VRAMCtrl;



--
--  Oscilloscope VRAM Controller Moore State Machine
--

architecture  assign_statebits  of  VRAMCtrl  is

    subtype  states  is  std_logic_vector(10 downto 0);     -- state type

    -- define the actual states as constants

    -- bits are: RAS CAS TRG WE ASRC AROW UACK RDY ID[2..0]
    constant  IDLE      : states := "11111100000";  -- waiting for events

    constant  READ1     : states := "01111100000";  -- read state 1
    constant  READ2     : states := "01011000000";  -- read state 2
    constant  READ3     : states := "00011000000";  -- read state 3
    constant  READ4     : states := "00011101000";  -- read state 4
    constant  READ5     : states := "11111100001";  -- read state 5
    constant  READ6     : states := "11111100010";  -- read state 6

    constant  WRITE1    : states := "01111100001";  -- write state 1
    constant  WRITE2    : states := "01101000000";  -- write state 2
    constant  WRITE3    : states := "00101001000";  -- write state 3
    constant  WRITE4    : states := "11111100011";  -- write state 4
    constant  WRITE5    : states := "11111100100";  -- write state 5

    constant  SERIAL1   : states := "11010100000";  -- serial transfer state 1
    constant  SERIAL2   : states := "01010100000";  -- serial transfer state 2
    constant  SERIAL3   : states := "01110000000";  -- serial transfer state 3
    constant  SERIAL4   : states := "00110000000";  -- serial transfer state 4
    constant  SERIAL5   : states := "11111110000";  -- serial transfer state 5
    constant  SERIAL6   : states := "11111100101";  -- serial transfer state 6

    constant  REFRESH1  : states := "10111100000";  -- refresh state 1
    constant  REFRESH2  : states := "00111100001";  -- refresh state 2
    constant  REFRESH3  : states := "00111100010";  -- refresh state 3
    constant  REFRESH4  : states := "00111100011";  -- refresh state 4
    constant  REFRESH5  : states := "11111100110";  -- refresh state 5
    constant  REFRESH6  : states := "11111100111";  -- refresh state 6

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

begin


    -- the output is always the 8 highest bits of the encoding
    ras <= CurrentState(10);
     cas <= CurrentState(9);
     trg <= CurrentState(8);
     welu <= CurrentState(7);
     asrc <= CurrentState(6);
     arow <= CurrentState(5);
     uack <= CurrentState(4);
     rdy <= CurrentState(3);


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

    transition:  process (Reset, ureq, we, cs, CurrentState)
    begin

        case  CurrentState  is          -- do the state transition/output

        -- transition from idle
            when  IDLE =>               -- in idle state, do transition
                if  (ureq = '1')  then
                    NextState <= SERIAL1;      -- serial update request has priority
                elsif  (cs = '0' and we = '1')  then
                    NextState <= READ1;        -- read request
                elsif  (cs = '0' and we = '0')  then
                    NextState <= WRITE1;        -- write request
                else
                    NextState <= REFRESH1;     -- nothing to do; refresh
                end if;

        -- read cycle
            when  READ1 =>           -- continue read cycle
                NextState <= READ2;

            when  READ2 =>           -- continue read cycle
                NextState <= READ3;

            when  READ3 =>            -- continue read cycle
                NextState <= READ4;

            when  READ4 =>            -- continue read cycle
                NextState <= READ5;

            when  READ5 =>            -- continue read cycle
                NextState <= READ6;

            when  READ6 =>            -- end read cycle
                NextState <= IDLE;

        -- write cycle
            when  WRITE1 =>           -- continue write cycle
                NextState <= WRITE2;

            when  WRITE2 =>           -- continue write cycle
                NextState <= WRITE3;

            when  WRITE3 =>           -- continue write cycle
                NextState <= WRITE4;

            when  WRITE4 =>           -- continue write cycle
                NextState <= WRITE5;

            when  WRITE5 =>           -- end write cycle
                NextState <= IDLE;

        -- serial update cycle
            when  SERIAL1 =>          -- continue serial cycle
                NextState <= SERIAL2;

            when  SERIAL2 =>          -- continue serial cycle
                NextState <= SERIAL3;

            when  SERIAL3 =>          -- continue serial cycle
                NextState <= SERIAL4;

            when  SERIAL4 =>          -- continue serial cycle
                NextState <= SERIAL5;

            when  SERIAL5 =>          -- continue serial cycle
                NextState <= SERIAL6;

            when  SERIAL6 =>          -- end serial cycle
                NextState <= IDLE;

            -- refresh cycle
            when  REFRESH1 =>          -- continue refresh cycle
                NextState <= REFRESH2;

            when  REFRESH2 =>          -- continue refresh cycle
                NextState <= REFRESH3;

            when  REFRESH3 =>          -- continue refresh cycle
                NextState <= REFRESH4;

            when  REFRESH4 =>          -- continue refresh cycle
                NextState <= REFRESH5;

            when  REFRESH5 =>          -- continue refresh cycle
                NextState <= REFRESH6;

            when  REFRESH6 =>          -- end refresh cycle
                NextState <= IDLE;

                when  OTHERS   =>          -- default; needed for compilation
                   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;

```

The transitions within the read, write, serial, and refresh cycles are those of typical RAS/CAS DRAM access cycles, shown in the timing diagrams of Figures 12-15 and Tables 2-5, and output the necessary signals for the VRAM controller (RAS, CAS, TRG, WEL/U), an active-high row update acknowledge signal ($UACK$) to the display controller when the row update has been completed, an active-high ready signal (RDY) to the processor when a read or write cycle has been completed (and, in case of the read cycle, valid data will be present on the data bus for 1 clock following the activation of the signal), and the control signals for the address multiplexer (MUX) that determine what section of which address bus should be output: $ASRC$ is high when the processor's address bus should be output, and low when the row update address should be output, $AROW$ is high when the row address should be output, and low when the column address should.

Note that the controller does not specify a number of wait states, and requires the processor to wait for the RDY signal to go high before completing the access cycle instead.

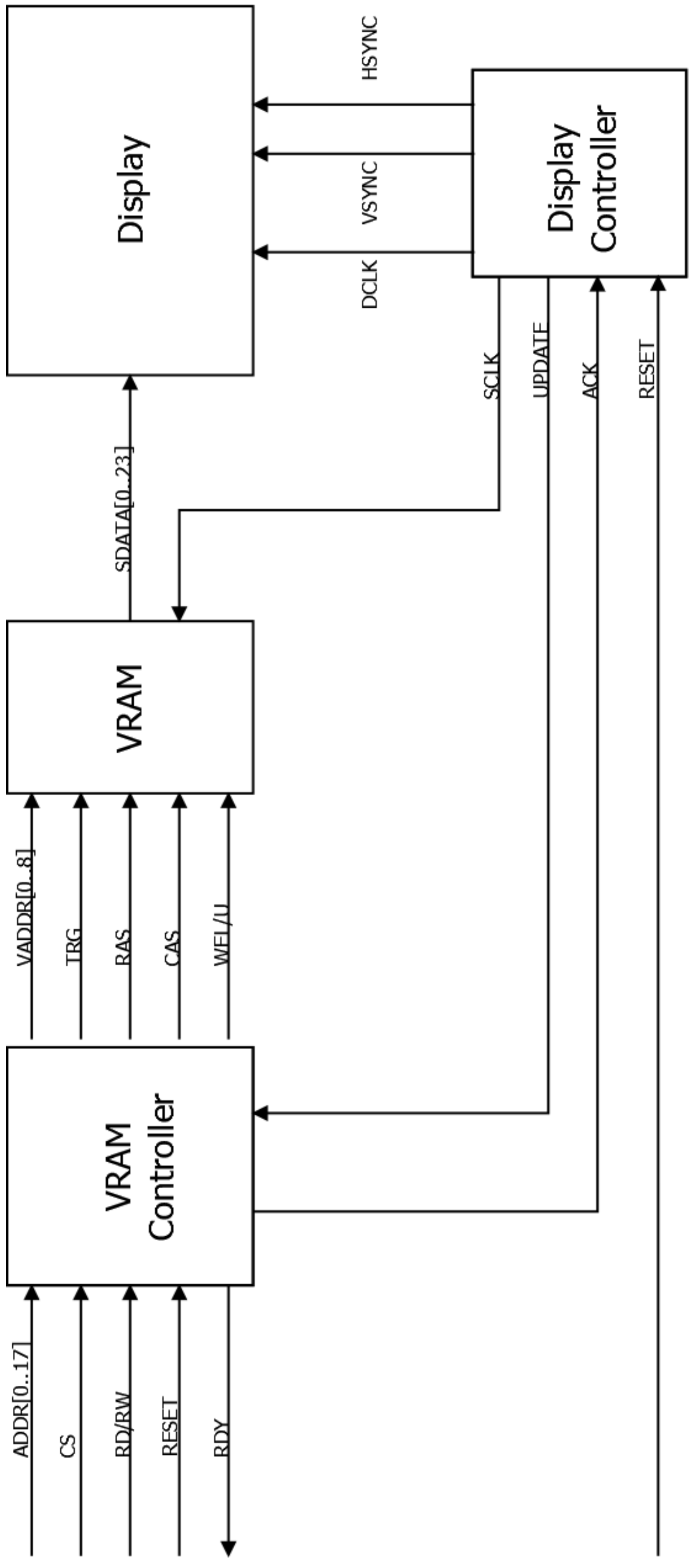


Figure 10: High level, summarizing block diagram of the VRAM and display interface. The VRAM controller listens to the CPU's generic memory controller commands, and generates the necessary timing lines for the VRAM device. The display controller periodically updates the serial row on the VRAM device, outputting new data to the display, while simultaneously generating the necessary timing signals.

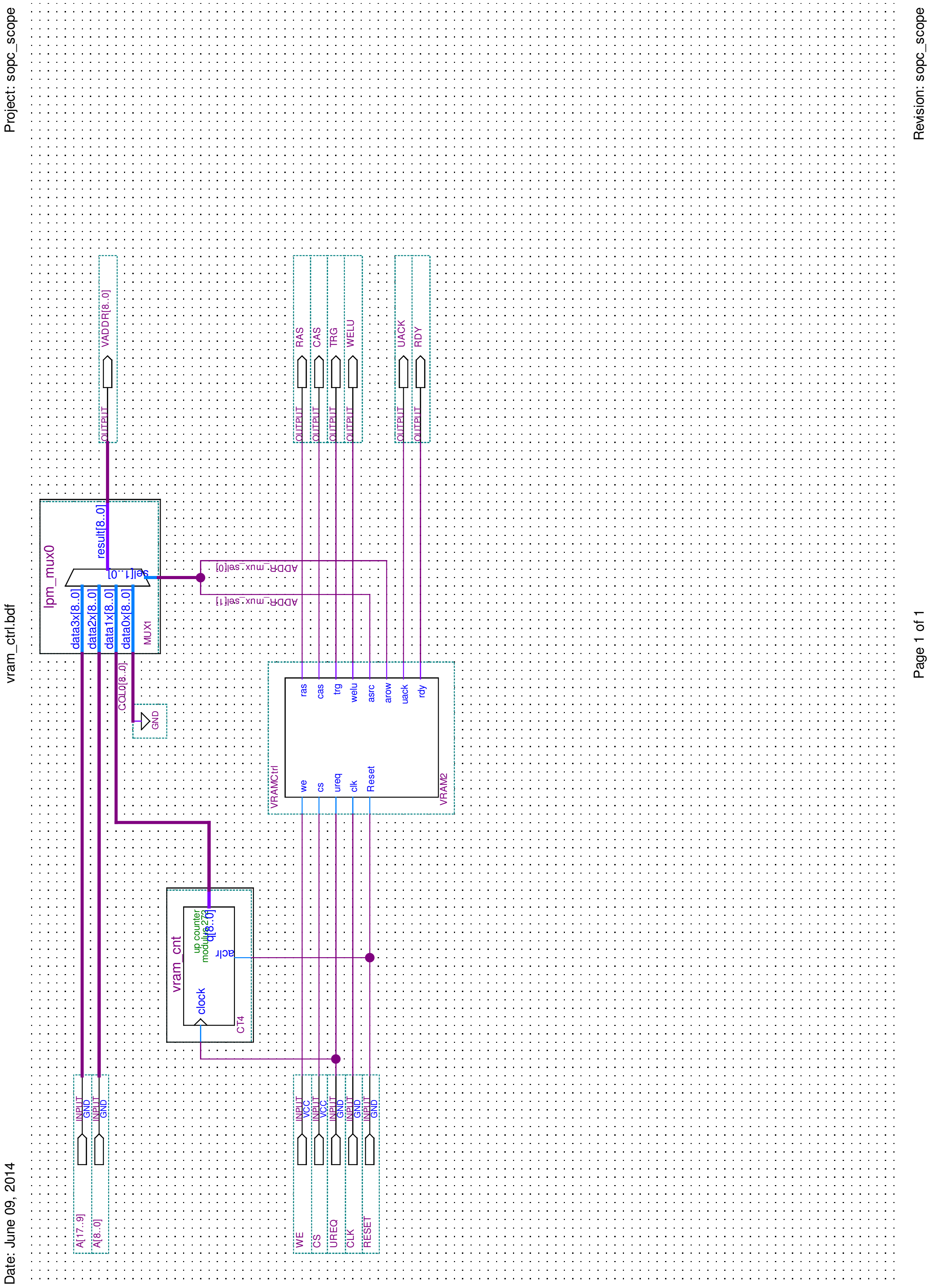


Figure 11: Block diagram of the VRAM controller component within the FPGA. The document is described in Section 2.2.5.

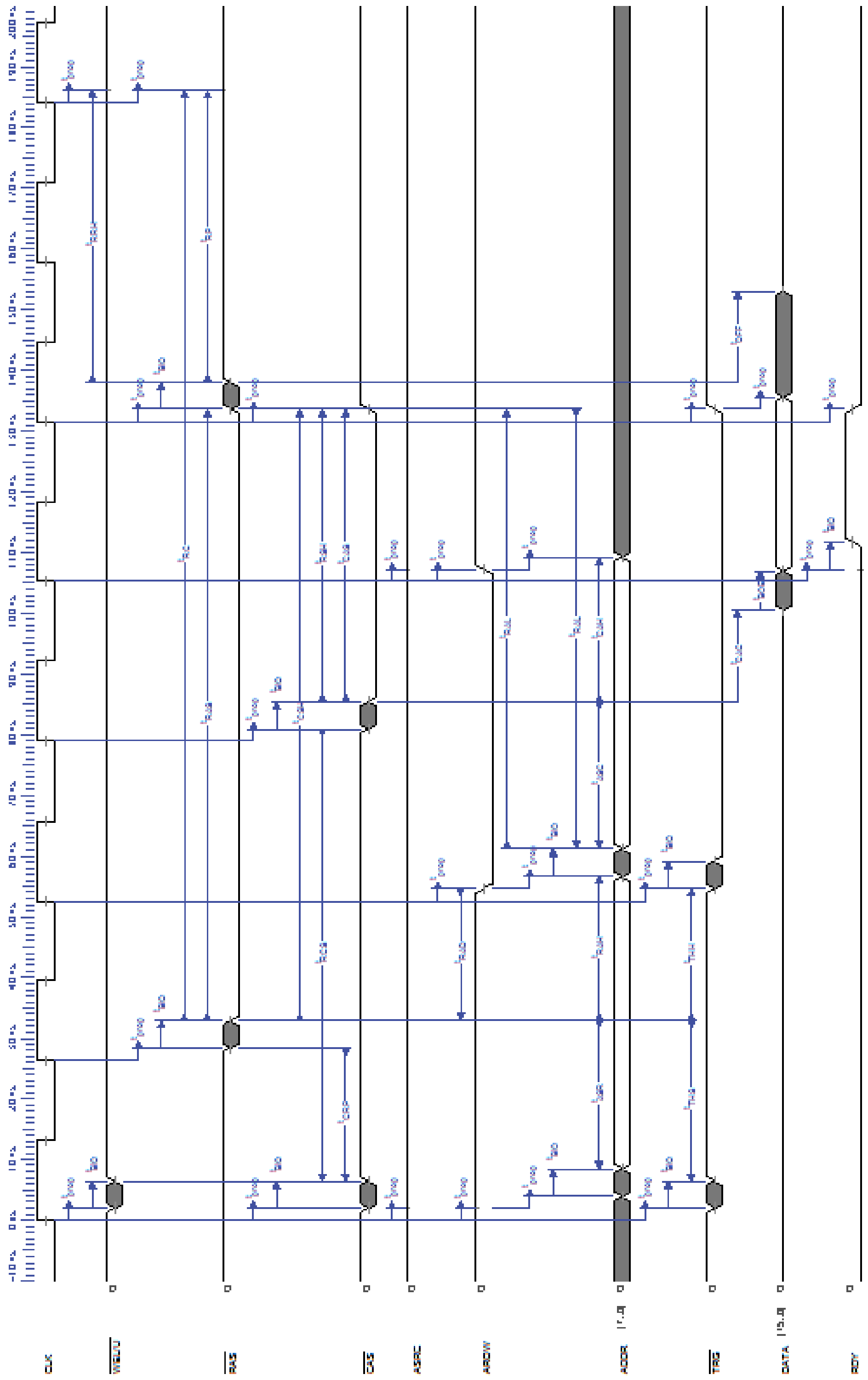


Figure 12: Timing diagram of the read cycle of the VRAM device, described in Section 2.2.5.

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation Delay		2ns	
t_{BIO}		Buffer I/O Delay		4.5ns	
t_{BOE}		Buffer output enable		6.5ns	
t_{RAS}		RAS Pulse Width	50ns		
t_{CAS}		CAS Pulse Width	10ns		
t_{CSH}		CAS Hold Time	45ns		
t_{RC}		Read/Write Cycle Time	104ns		
t_{RP}		RAS Precharge Time	40ns		
t_{RSH}		RAS Hold Time	15ns		
t_{RAH}		Row Address Hold Time	10ns		
t_{ASR}		Row Address Setup	0ns		
t_{CAH}		Column Address Hold Time	10ns		
t_{ASC}		Column Address Setup Time	0ns		
t_{THH}		TRG High Hold Time	10ns		
t_{THS}		TRG High Setup Time	0ns		
t_{CAC}		CAS Output Time		15ns	
t_{OFF}		Data Bus Off Time		15ns	
t_{RRH}		Read Command Hold Time (from RAS)	0ns		
t_{RAL}		Column Address to RAS Lead Time	30ns		

Table 2: Table of constraints of the read cycle of the VRAM device, shown in Figure 12 and described in Section 2.2.5.

General Data continued...

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{RCH}		Read Command Hold Time (from CAS)	0ns		
t_{RCS}		Read Command Setup Time	0ns		
t_{CRP}		CAS to RAS Precharge Time	5ns		
t_{RAL}		Column Address to RAS Lead Time	30ns		
t_{RAD}		RAS to Column Address Delay Time	12ns		

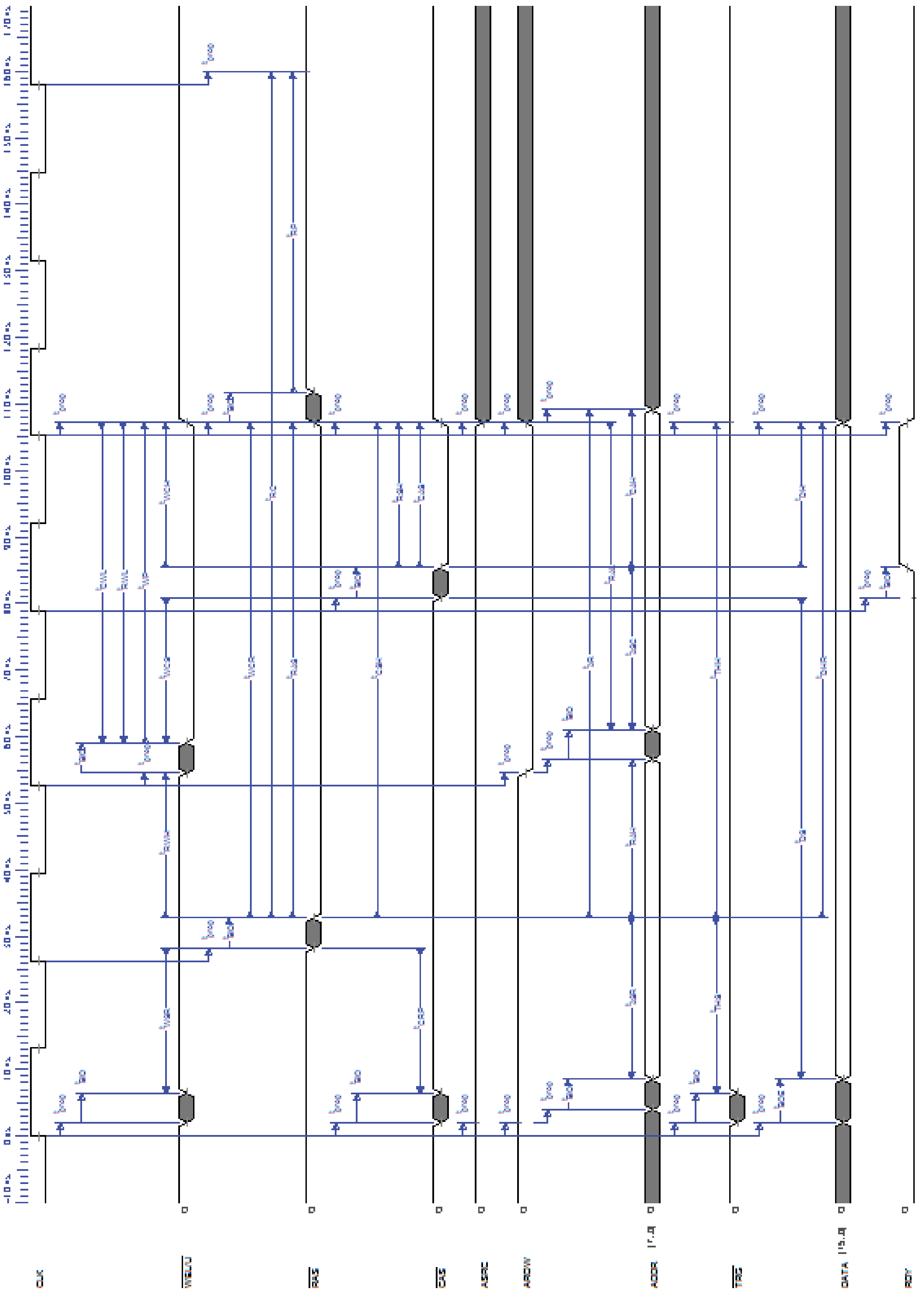


Figure 13: Timing diagram of the write cycle of the VRAM device, described in Section 2.2.5.

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation Delay		2ns	
t_{BIO}		Buffer I/O Delay		4.5ns	
t_{BOE}		Buffer output enable		6.5ns	
t_{RAS}		RAS Pulse Width	50ns		
t_{CAS}		CAS Pulse Width	10ns		
t_{CSH}		CAS Hold Time	45ns		
t_{RC}		Read/Write Cycle Time	104ns		
t_{RP}		RAS Precharge Time	40ns		
t_{RSH}		RAS Hold Time	15ns		
t_{RAH}		Row Address Hold Time	10ns		
t_{ASR}		Row Address Setup	0ns		
t_{ASC}		Column Address Setup Time	0ns		
t_{CAH}		Column Address Hold Time	10ns		
t_{THS}		TRG High Setup Time	0ns		
t_{THH}		TRG High Hold Time	10ns		
t_{CRP}		CAS to RAS Precharge Time	5ns		
t_{WSR}		WE Setup Time	0ns		
t_{RWH}		WE Hold Time	10ns		
t_{WCS}		Write Command Setup Time	0ns		

Table 3: Table of constraints of the write cycle of the VRAM device, shown in Figure 13 and described in Section 2.2.5.

General Data continued...

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{AR}		Column Address Hold Time (from RAS)	50ns		
t_{WCH}		Write Command Hold Time	10ns		
t_{RAL}		Column Address to RAS Lead Time	30ns		
t_{WP}		Write Command Pulse Width	10ns		
t_{WCR}		Write Command Hold Time (from RAS)	50ns		
t_{RWL}		Write Command to RAS Lead Time	15ns		
t_{CWL}		Write Command to CAS Lead Time	15ns		
t_{DS}		Data Setup TIme	0ns		
t_{DH}		Data Hold Time	10ns		
t_{DHR}		Data Hold Time (from RAS)	50ns		

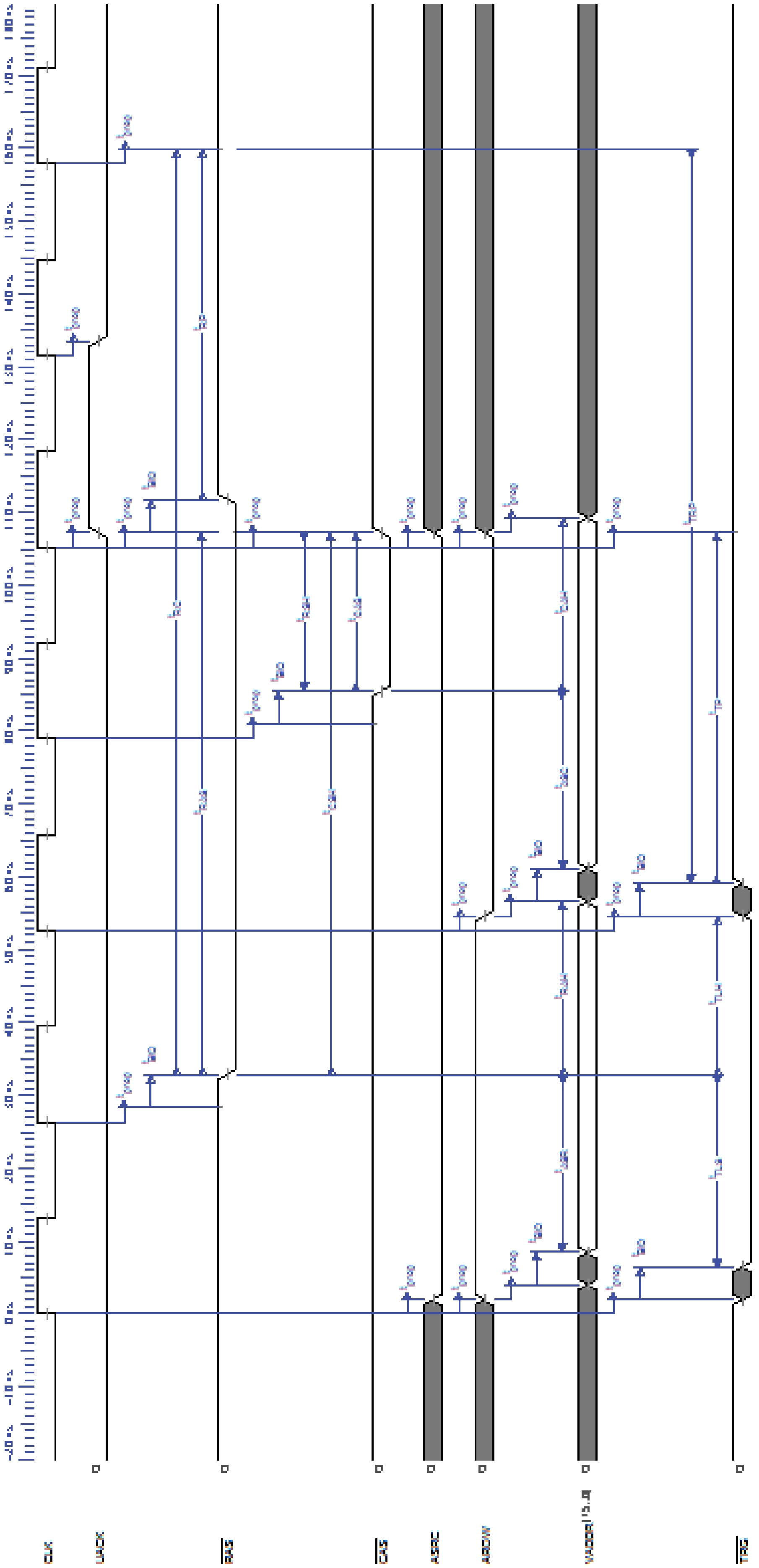


Figure 14: Timing diagram of the serial row transfer cycle of the VRAM device, described in Section 2.2.5.

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation Delay		2ns	
t_{BIO}		Buffer Input/Output Delay		4.5ns	
t_{RAS}		RAS Pulse Width	60ns		
t_{RC}		Read/Write Cycle Time	104ns		
t_{RP}		RAS Precharge Time	40ns		
t_{CSH}		CAS Hold Time	45ns		
t_{RSH}		RAS Hold Time	15ns		
t_{RCD}		RAS to CAS Delay Time	15ns	42ns	
t_{CAS}		CAS Pulse Width	10ns	10000ns	
t_{AR}		Column Address Hold Time (from RAS)	50ns		
t_{RAD}		RAS to Column Address Delay Time	12ns		
t_{RAL}		Column Address to RAS Lead Time	30ns		
t_{ASR}		Row Address Setup	0ns		
t_{RAH}		Row Address Hold Time	10ns		
t_{ASC}		Column Address Setup Time	0ns		
t_{CAH}		Column Address Hold Time	10ns		
t_{TLS}		TRG Low Setup Time	0ns		
t_{TLH}		TRG Low Hold Time	10ns		

Table 4: Table of constraints of the serial row transfer cycle of the VRAM device, shown in Figure 14 and described in Section 2.2.5.

General Data continued...

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{RSD}		RAS to First SC Delay Time	60ns		
t_{TRP}		TRG to RAS Precharge Time	40ns		
t_{TP}		TRG Precharge Time	20ns		
t_{TSD}		TRG to First SC Delay Time	10ns		

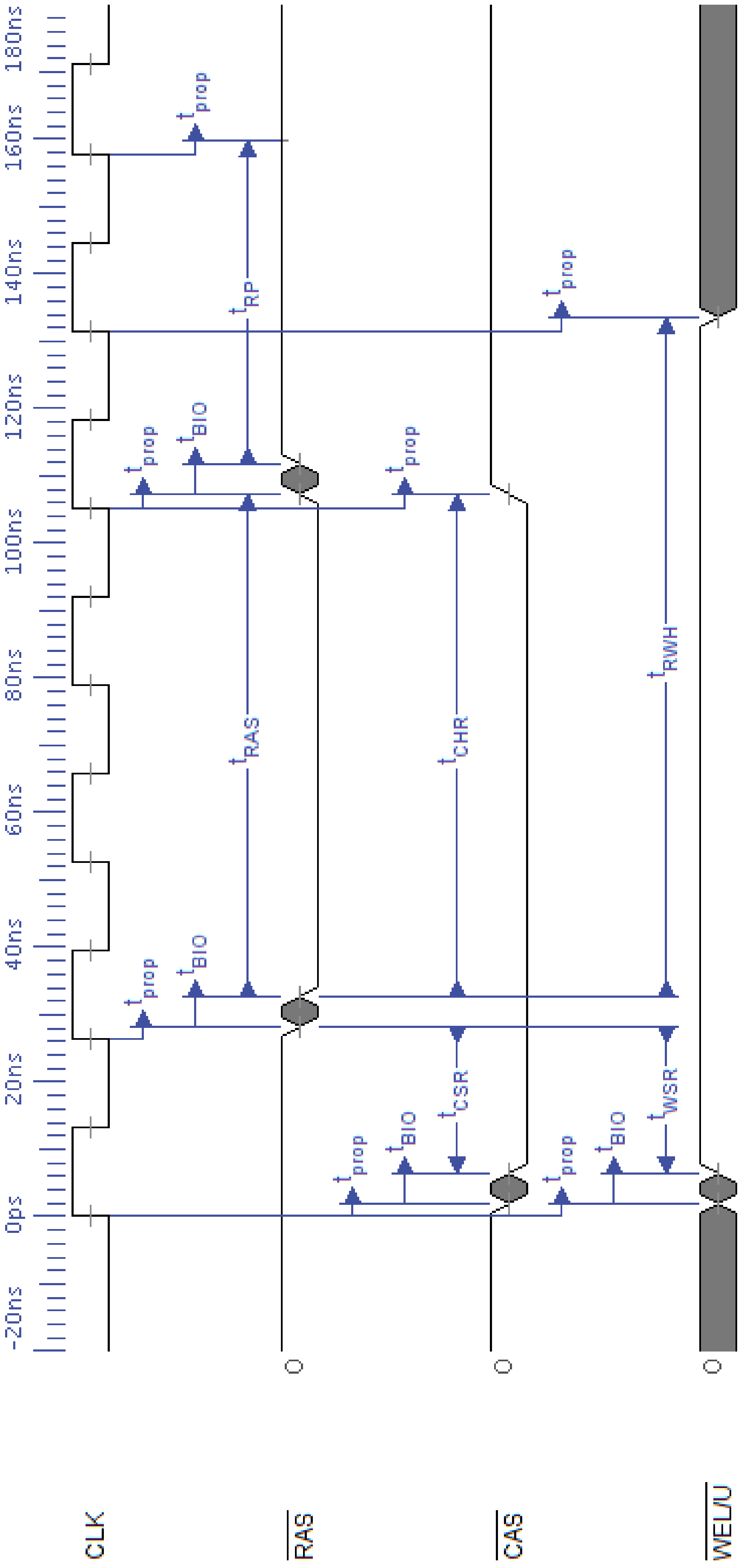


Figure 15: Timing diagram of the refresh cycle of the VRAM device, described in Section 2.2.5.

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation Delay		2ns	
t_{BIO}		Buffer Input/Output delay		4.5ns	
t_{RP}		RAS Precharge Time	40ns		
t_{RAS}		RAS Pulse Width	50ns	10000ns	
t_{RC}		Read/Write Cycle Time	104ns		
t_{RPC}		RAS Precharge to CAS Active	0ns		
t_{CSR}		CAS Setup Time	5ns		
t_{CHR}		CAS Hold Time	10ns		
t_{WSR}		WE Setup Time	0ns		
t_{RWH}		WE Hold Time	10ns		

Table 5: Table of constraints of the refresh cycle of the VRAM device, shown in Figure 15 and described in Section 2.2.5.

2.2.6. Display Controller

The display controller outputs data from the VRAM onto the display at a very fast rate, ensuring that the most up-to-date version of the video data is constantly being displayed. A generic block diagram of the interactions between processor, VRAM controller, and display controller can be seen in Figure 10. The display controller is illustrated in more detail in the block diagram of Figure 16.

The controller takes a clock, *CLK*, a reset signal, *RESET*, and a VRAM controller row update acknowledge signal, *UACK*, as inputs. The clock is immediately divided by two by clocking a mod-2 counter, *CT5*, with it, and taking the high bit of the output. The bit is sent through delayed flip-flop *FF10* to remove any glitches.

All the necessary timing signals required by the display are generated using a combination of counters, comparators, and flip-flops. The outline of an interaction is shown in Figure 17; the detailed timing of every transition is shown in the timing diagram of Figure 18 and Table 6.

The display pixel clock, *DCLK*, must always run. It is thus generated from the divided clock, pipelined through delayed flip-flop *FF11* for synchronization.

The frame clock, *VSYNC*, is also always running, going low at the beginning of every frame, staying low for the VSYNC pulse period of 10 *HSYNC*s, and then going high and staying high for the remainder of the frame. Each cycle lasts a total of 286 *HSYNC*s. This structure is achieved with counter *CT6*, which counts the number of clocks per frame, and comparator *CMP5*, which determines the moment *VSYNC* should transition. The signal is then pipielined through delayed flip-flop *FF12* to synchronize it with the rest of the controller.

The row clock (*HSYNC*) is also constantly running, going low at the beginning of a row, staying low for the HSYNC pulse period of *DCLK*s, and then going high and staying high for the remainder of the row. Each cycle lasts a total of 525 *SCLK*s. This structure is achieved with counter *CT7*, which counts the number of clocks per row, and comparator *CMP9*, which determines the moment *HSYNC* should transition. The signal is then pipelined through two delayed flip-flops, *FF14* and *FF15*, to ensure synchronization with the other signals in the controller.

The serial clock, *SCLK*, is used to shift pixels out of the VRAM. The signal must only run during the display period of each row in order to output the correct region of the VRAM. To achieve this result, *CT7* is used in combination with comparators *CMP10* and *CMP11*, which determine the bounds of within a row where the clock should run (effectively excluding the horizontal front porch of 2 *DCLK*s and the horizontal back porch of 2 *DCLKs*). These conditions are then ANDed in *G20* with the output of comparators *CMP67* and *CMP7*, which use row-counter *CT6* to only enable *SCLK* during the display part of the frame, effectively excluding the the vertical front porch of 2 *SCLKs* and the vertical back porch of 2 *SCLKs*.

The outputs of *CMP6* and *CMP7* are also used to generate a row update request signal for the VRAM controller at the end of the display period within each row. As long as the we're within the display portion of the frame (i.e. between row porches), *G18* will set S/R flip-flop *FF13* when the row clock reaches the end of the display portion of a row, indicated by comparator *CMP8* based on the row clock of *CT7*. The output *FF13* is used as a row update signal ($UREQ$); the flip-flop is reset when the VRAM controller confirms the completion of the row update ($UACK$).

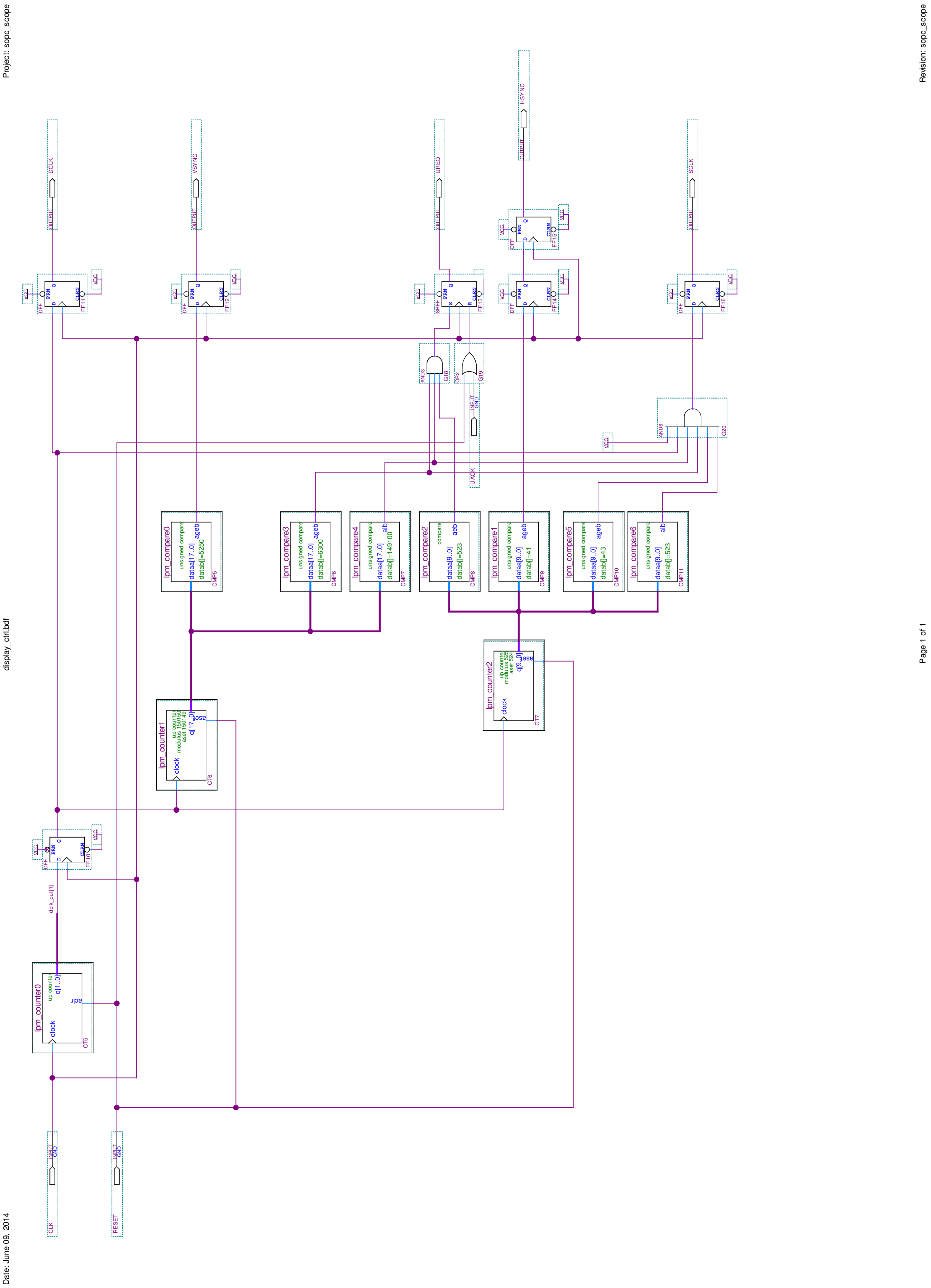
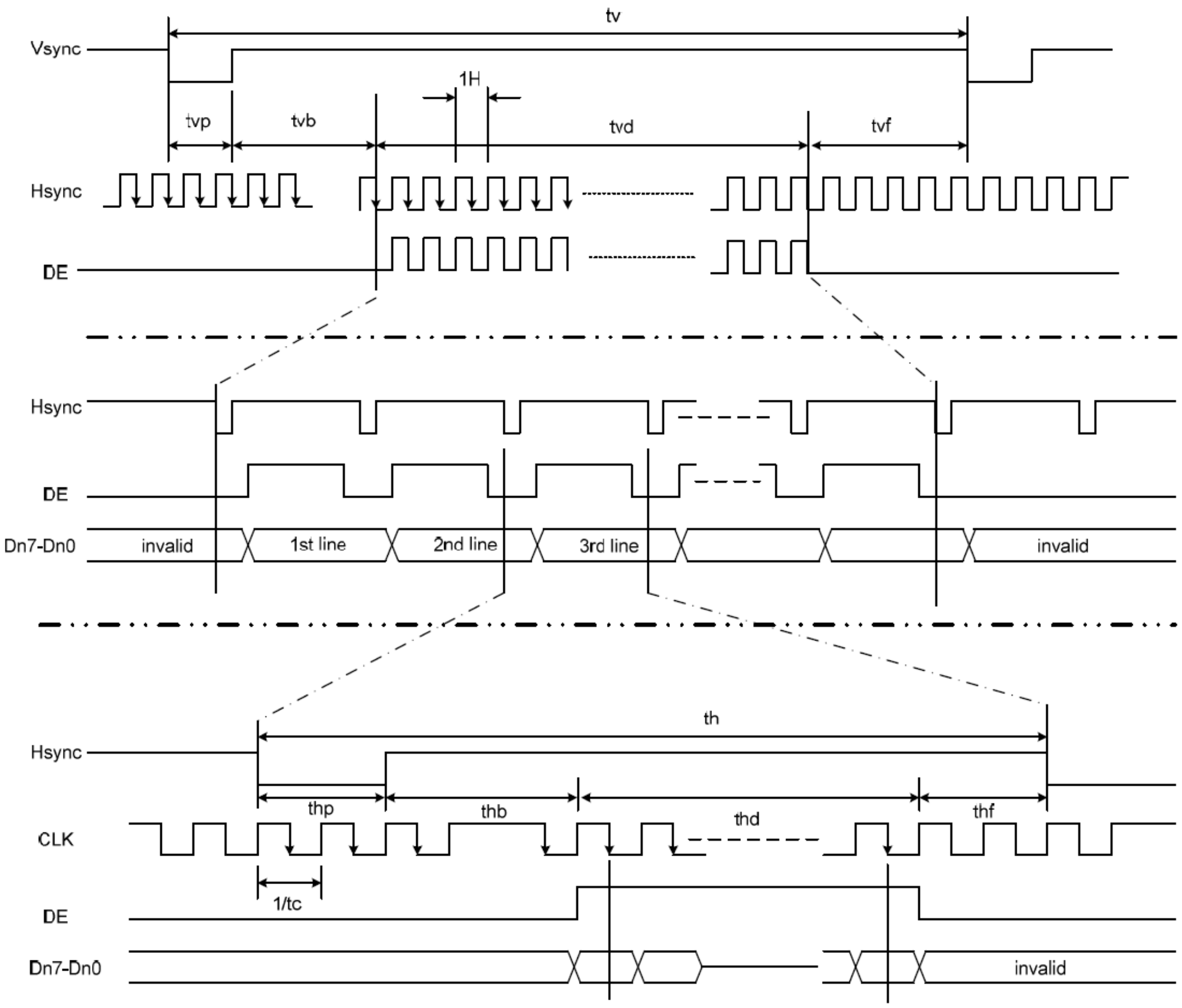


Figure 16: Block diagram of the display controller component within the FPGA. The document is described in Section 2.2.6.



[7]

Figure 17: Summary of the display's frame cycle structure. Source: HX8257 LCD driver datasheet.

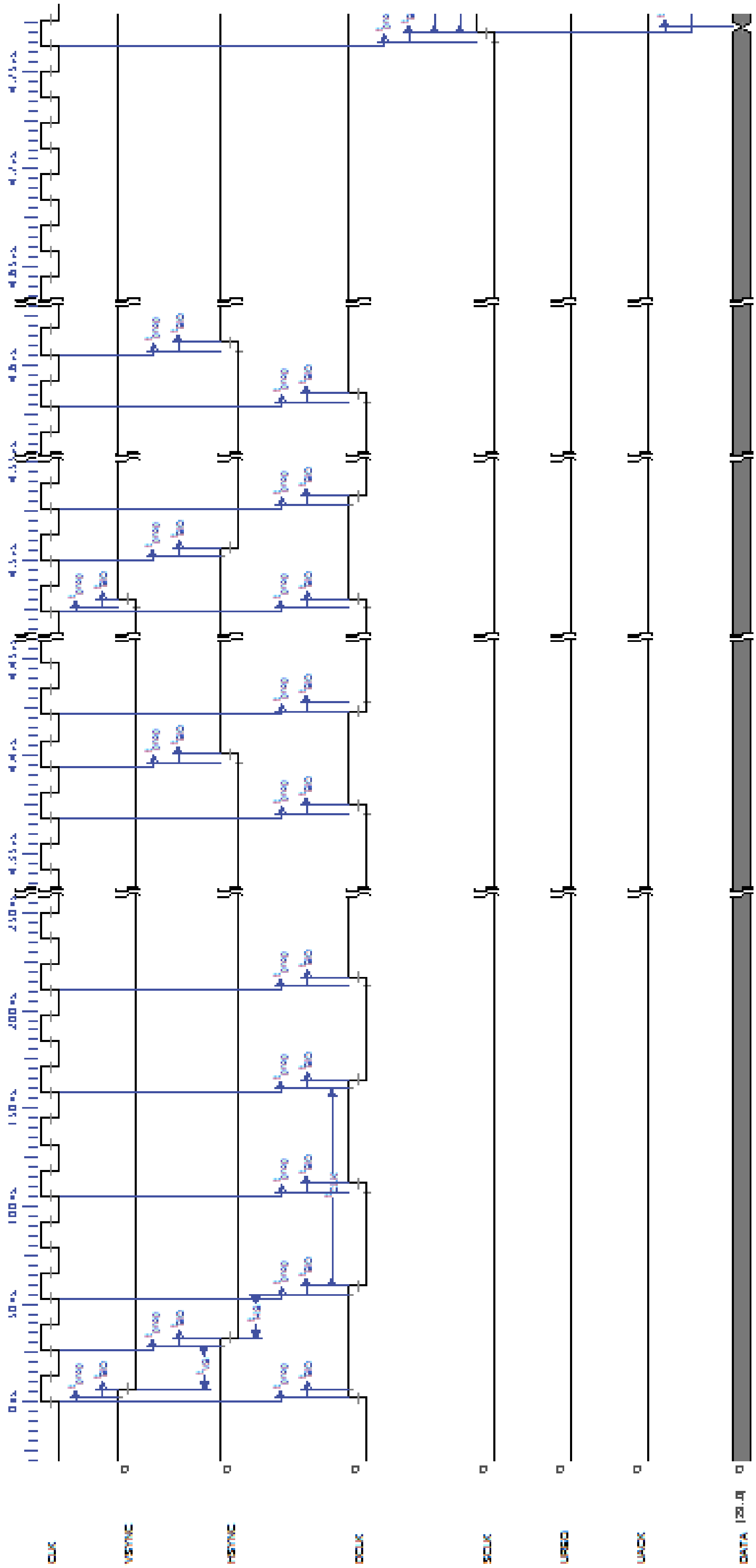
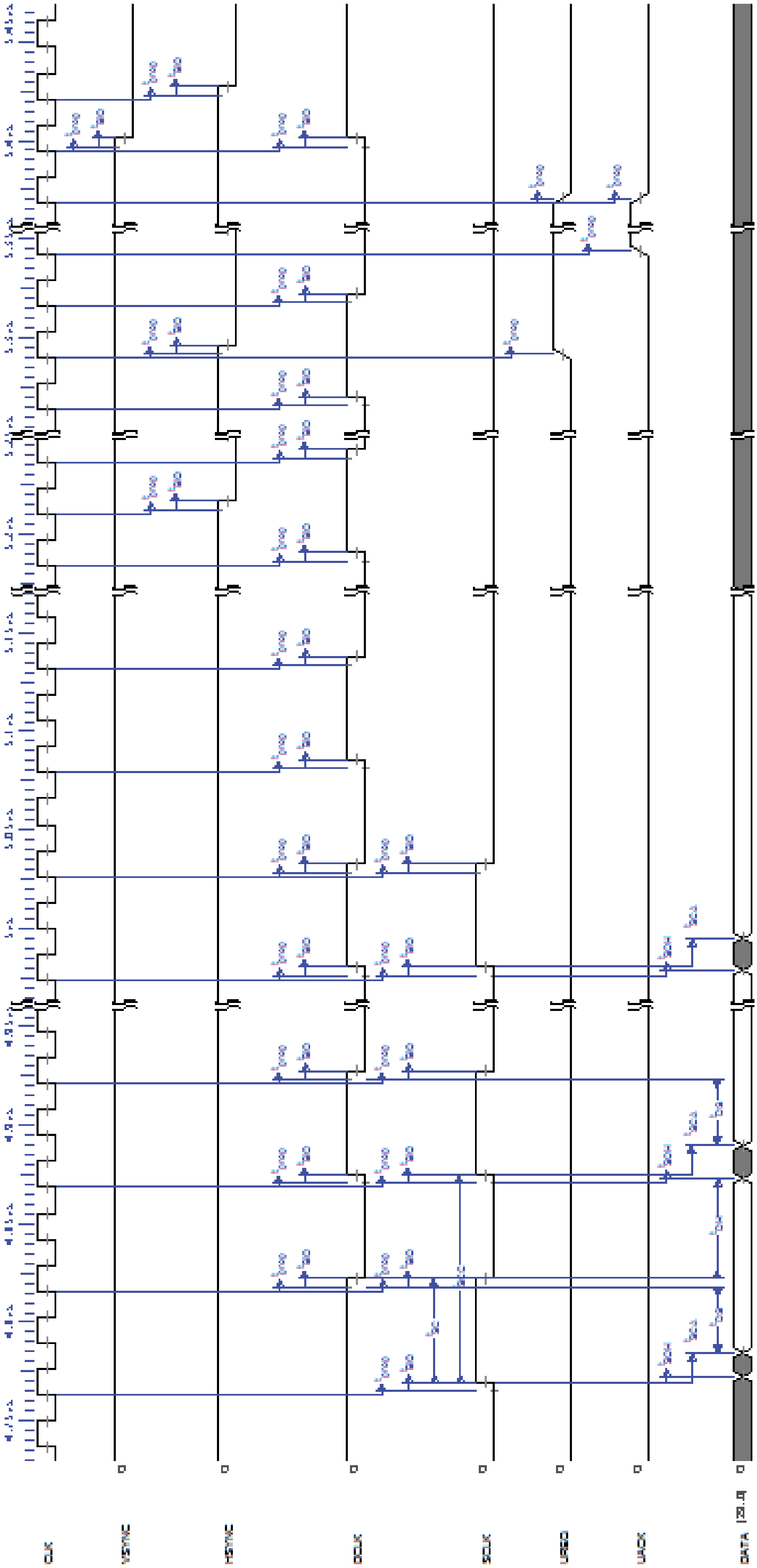


Figure 18: Timing diagram of the display cycle, described in Section 2.2.6.


CLK
VSYNC
HSYNC
DCLK
SCLK
UREQ
UACK
DATA[23..0]
4.75 us
4.8 us
4.85 us
4.9 us
4.95 us
5 us
5.05 us
5.1 us
5.15 us
5.2 us
5.25 us
5.3 us
5.35 us
5.4 us
5.45 us

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{BIO}		Buffer I/O Delay		4.5ns	
t_{prop}		FPGA Propagation Delay		2ns	
t_{CLK}		DCLK Period	66.7ns		
H		HSync Cycle - 2100			
t_{VP}		VSync Pulse Width - 10 H			
t_{VB}		VSync Back Porch - 2 H			
t_{VD}		VSync Display Period - 272 H			
t_{VF}		VSync Front Porch - 2 H			
t_{HP}		HSync Pulse Width - 164 CLKS	260ns	4.33us	
t_{HB}		HSync Back Porch - 8 CLKS			
t_{HD}		HSync Display Period - 1920 CLKS			
t_{HF}		HSync Front Porch - 8 CLKS			
t_{SCA}		VRAM Access Time (from SC)		15ns	
t_{SOH}		VRAM Serial Output Hold Time (rom SC)	3ns		
t_{DS}		Data Setup Time	10ns		
t_{DH}		Data Hold Time	10ns		
t_{VS}		VSync Setup Time	10ns		
t_{HS}		HSync Setup Time	10ns		

Table 6: Table of constraints of a display cycle, shown in Figure 18 and described in Section 2.2.6.

General Data continued...

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{SC}		Serial Clock Pulse Width	5ns		
t_{SCC}		Serial Clock Period	18ns		

2.3. Reset Logic

A MAX706AS reset chip, *U5* (illustrated in Figure 5), is used to provide power-on reset and manual reset functionality. The chip is closely connected to the FPGA, and therefore highlighted in red in Figure 2. The device natively provides the power-on and power-loss reset functionality, while switch *S1* adds the manual reset functionality by allowing the manual reset pin *MR* to be pulled low: users can thus reset the system by pressing and releasing *S1*. Jumper *W1* can be used to enable watchdog timer expiration reset when shorted: in this configuration, FPGA pin 22 (line *WDI*) would need to transition from low to high or high to low every 1.6s at most to avoid reset. *U5*'s reset output line is connected to *U2*'s *nCONFIG* line, which is an active-low reset input to the FPGA. For the correct operation of the system, *W1* should be left unjumped.

2.4. Clock Logic

The FPGA is clocked using a 38 MHz SG363 oscillator, *X1*, illustrated in Figure 5. The device is closely connected to the FPGA, and therefore highlighted in red in Figure 2. Its output enable line is tied high through *R17* to always enable the clock. Its output is connected to the *U2*'s *CLK0* input, and therefore acts as the FPGA's main clock. Every other clock line, *CLK1* through *CLK15*, is tied low and therefore disabled.

2.5. JTAG Interface

The FPGA can be programmed and debugged using a JTAG interface, through JTAG connector *J1*, illustrated in Figure 5. Lines *TMS*, *TCK*, *TDO*, and *TDI* control the interface. Pull-up and pull-down resistors *R7*, *R8*, and *R9* are installed as needed by the JTAG interface. The interface is closely related to the FPGA, and therefore highlighted in red in Figure 2.

2.6. Power Supply

A +5 V / +12 V / -12 V external power supply is expected to be used with this system. The different power lines are connected through DIN-5 connector *J4*, illustrated in Figure 5. Each one of them is immediately filtered using capacitors *C1*, *C98*, and *C99*. The 5 V line, capable of providing the most current, is then regulated to create the various rails. The 3.3 V source is regulated by a TLV1117-33 regulator, *U13*. The 2.5 V source is regulated by a TLV1117-25 regulator, *U11*. The 1.25 V source is regulated by a TLV1117-ADJ adjustable regulator, *U12*: the output selection resistors *R72* and *R73* are chosen to output the minimum possible voltage, 1.25 V. Each one of the regulators has a 100 μF capacitor at the output (*C24*, *C25*, *C54*). The other listed bypass capacitors (*C2*-*C23*, *C26*-*C53*, *C95*-*C97*) are placed as close as possible to every power pin of *U1*-*U3* and *U6*-*U10*that uses that regulator, and are sized as required by the component itself.

Additionally, the +5 V line is also regulated to +20 V to power the display's backlight LEDs, as illustrated in Figure 23. This is performed using an LMR62014 boost regulator. Resistors *R59* and *R76* are chosen to select the correct output voltage, while inductor *L1* and capacitor

C69 are selected to obtain the best output waveform characteristics.

The power supply circuitry is highlighted in blue in Figure 2. Note that all bypass capacitors are placed on the backside of the board, drawn in Figure 3.

2.7. Buffers

Every FPGA pin that can be sent through a buffer is; thus, every buffer is placed between the FPGA, *U2*, and some other chip(s). This provides protection against over-voltage, converts every voltage into 3.3 V, and provides better current characteristics, allowing more devices to be driven with the each line. Seven 74LVT16245 buffers, *U1*, *U3*, and *U6*-*U10* (illustrated in Figure 5), are used to this goal. The delay introduced by these devices is assumed throughout the project to be less than 2 ns.

Buffer *U8* relays the low 16 bits of the address bus (*A15..0*) output from *U2* on both ports; therefore, both of its ports are configured as output (B→A) by pulling the direction pins *1DIR* and *2DIR* low through *R26* and *R24*. The outputs are always enabled, and thus *1OE* and *2OE* are also pulled low through *R23* and *R25*.

Buffer *U9* relays the remaining 4 bits of the address bus (*A19..16*), as well as chip select signals *CS2..0*, write enable signal *WR*, and the bottom 8 bits of the data bus *D0..7*. Port A contains only output signals (B→A), and is therefore configured as always-enabled, output-only by pulling *1DIR* and *1OE* low through *R29* and *R27*. The data bus is always enabled by tying *2OE* low through *R30*; however, it is bidirectional, and its direction (*2DIR*) is therefore controlled by an FPGA output line that mediates the data bus, *IO207*.

Buffer *U6* connects the remaining 16 bits of the data bus (*D23..D8*). These are always-enabled, bidirectional as described above. Its output enable lines, *1OE* and *2OE* are therefore tied low through *R16* and*R18*, while the direction lines are controller by FPGA output line *IO207*.

Buffer *U1* relays VRAM and display signals. It transmits the video address bus (*VA8..0*), VRAM control signals (*RAS*, *CAS*, *WEL/U*, *SCLK*, *TRG*, and *DCLK*), and the display enable signal (*DISP*). Since all the signals are always-enabled, output-only (B→A), the direction and output enable control lines (*1DIR*, *2DIR*, *1OE*, *2OE*) are pulled low through resistors *R3*-*R6*.

Buffer *U3* carries the ADC output *SIG7..0* to the FPGA on port 2: the relative direction and output enable lines are therefore tied high and low respectively, through *R11* and *R12*. Port 1 connects eight unused FPGA pins. These pins are made available on break-out header *J3*. To allow the configuration of the direction of these lines, the direction and output enable pins are made available to *U2* on pins *IO55* and *IO56*, respectively.

Buffer *U7* bridges the user input lines from the rotary encoders (*SW0,1*; *ROT0A,B*; *ROT1A,B*) and the interrupt line from the touch screen controller (*PENIRQ*), as well as an unused pin made available on *J3*, on port 2. These lines are configured as always-enabled, input-only (A→B) by tying port 2 direction line *2DIR* high through *R21* and output enable line *2OE* low through *R22*. Port 1 connects display timing lines *HSYNC* and *VSYNC*, ADC clock *ACLK*, and five unused pins; all of these lines are configured always-enabled, output-only (B→A) by tying both port 1 control lines low through *R19,20*.

Buffer *U10* connects 12 unused FPGA pins to break-out pins in *J2*. The pins are divided between port 1, with seven connections, and port 2, with five. The direction and output of both ports can be configured by using FPGA pins *IO5,6* and *216,217*.

The buffers are highlighted in green in Figure 2. Note that all bypass capacitors are placed on the back side of the board, drawn in Figure 3.

2.8. Memory

The system uses three memory devices, in addition to the previously mentioned serial ROM used by the FPGA. A Random Access Memory (RAM) chip is used as volatile memory for the NIOS processor. A Read-Only Memory (ROM) device is used for the storage of non-volatile constants and code for the NIOS processor. Two Video RAM (VRAM) chips are used as a frame buffer for the display: the NIOS processor loads frame data into the memory device, which is subsequently read by the display controller and output to the display.

The memory devices are highlighted in yellow in Figure 2. Note that all bypass capacitors are placed on the backside of the board, drawn in Figure 3.

2.8.1. RAM

A HM628128B 128 Kword x 8-bit RAM chip, *U15*, constitutes the system's volatile storage. The device's connections are illustrated in Figure 19. The device is accessible by the processor at addresses 0x220000-0x23FFFF, as shown in Figure 4.

The chip is connected to the bottom 17 bits of the address bus (*A16..0*) and the bottom 8 bits of the data bus (*D7..0*), which are then routed through buffers *U8* and *U9* to *U2*, the FPGA. Note that both buses are shared between multiple devices. Also note that since this is a volatile memory device, the alignment of the address and data lines does not matter; therefore, both buses are "shuffled" on their interface with the chip to simplify routing on the PCB.

U15 is selected by using active-low line *CS1* uniquely, which is then routed through buffer *U9* and into *U2*; the active-high counterpart is tied high through *R36* to render it unnecessary. The chip is configured as always-enabled by pulling the output-enable line low through *R34*. The processor selects whether it's reading or writing to the chip by modulating the active-low write-enable line, *WR*. This line is bridged by buffer *U9* and then routed into the FPGA, *U2*.

The interactions between the processor and the memory device, which follow a generic memory controller interaction model, are illustrated in the timing diagrams of Figures 20-21 and Tables 7-8. Note from the diagrams that the device requires 3 wait states when reading, and 3 wait states when writing.

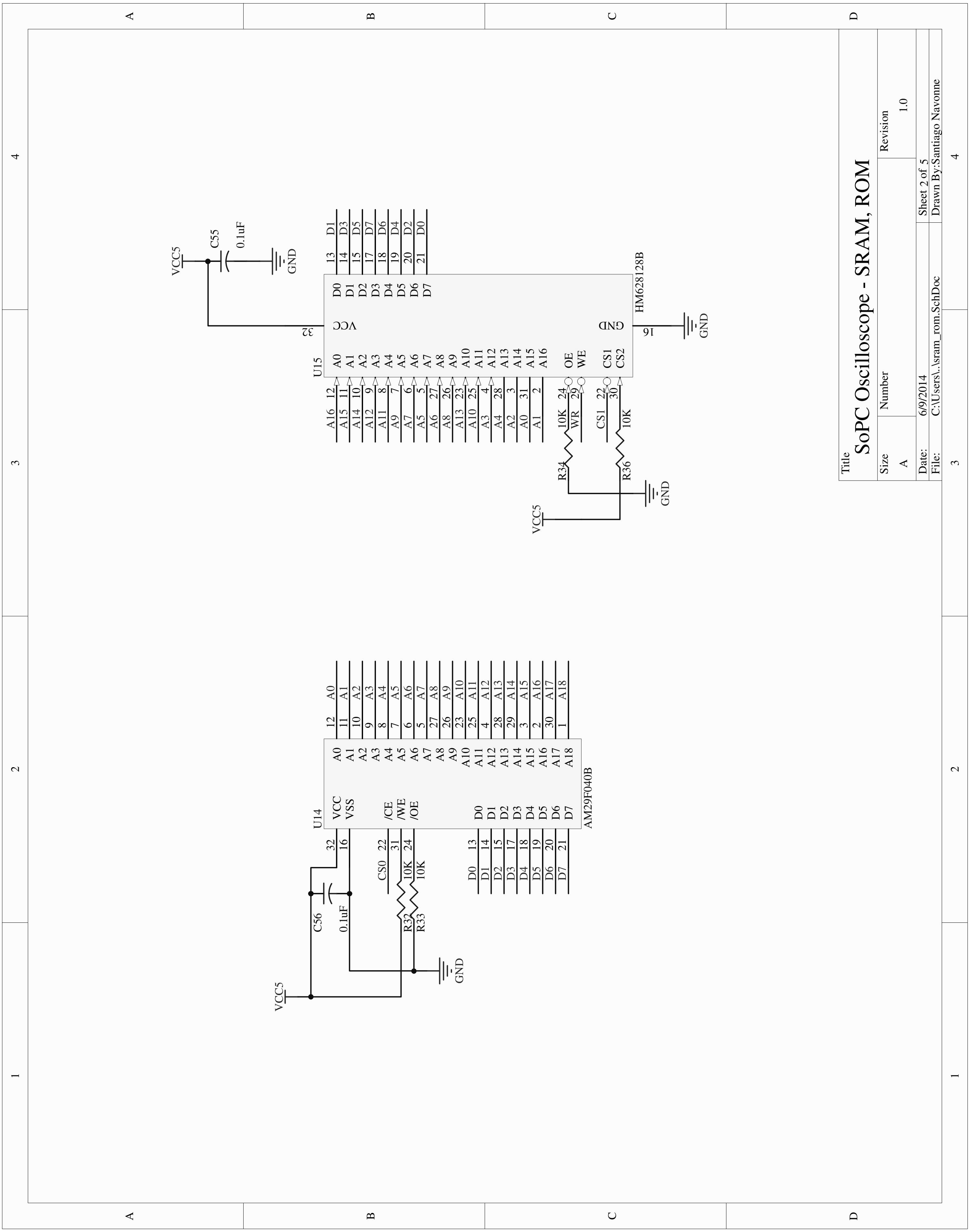


Figure 19: Schematic of the RAM and ROM memory devices. Further details are provided in Sections 2.8.1 and 2.8.2.

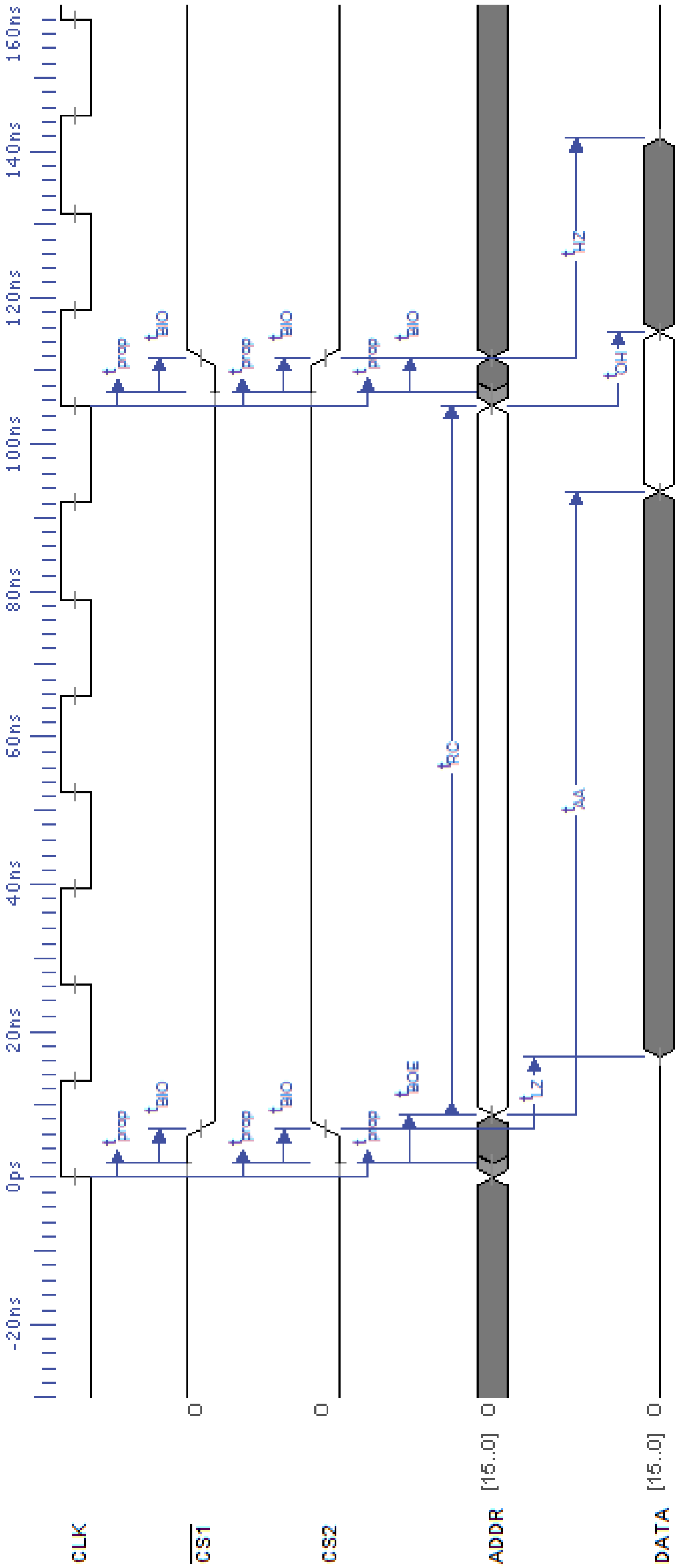
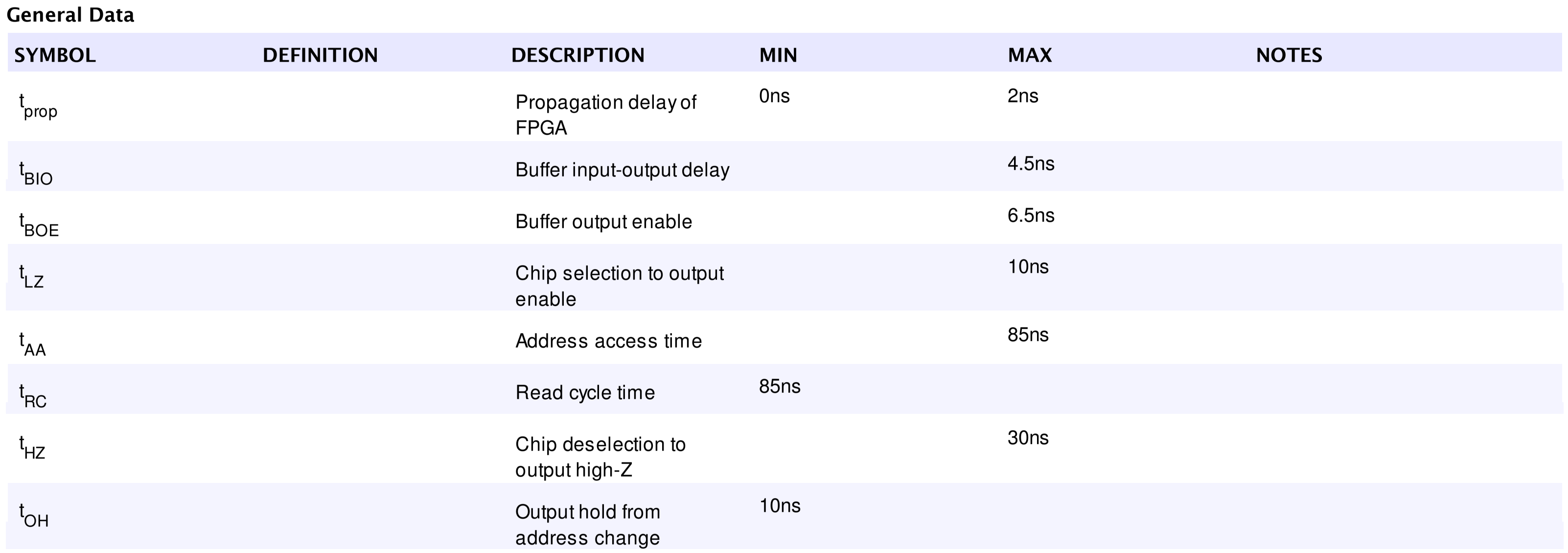


Figure 20: Timing diagram of the read cycle of the RAM device, described in Section 2.8.1.



General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation delay of FPGA	0ns	2ns	
t_{BIO}		Buffer input-output delay		4.5ns	
t_{BOE}		Buffer output enable		6.5ns	
t_{LZ}		Chip selection to output enable		10ns	
t_{AA}		Address access time		85ns	
t_{RC}		Read cycle time	85ns		
t_{HZ}		Chip deselection to output high-Z		30ns	
t_{OH}		Output hold from address change	10ns		

Table 7: Table of constraints of a display cycle, shown in Figure 20 and described in Section 2.8.1.

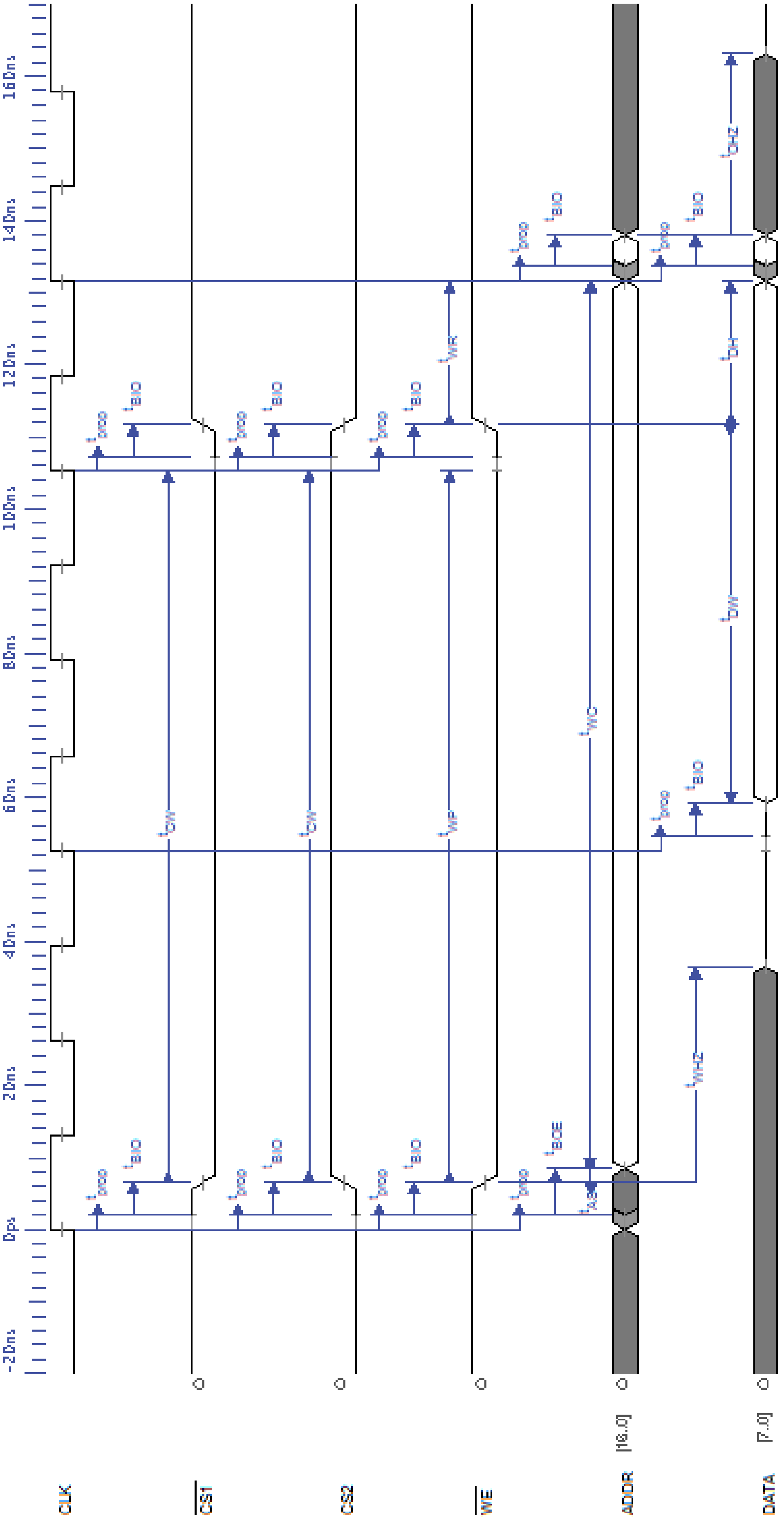


Figure 21: Timing diagram of the write cycle of the RAM device, described in Section 2.8.2.

General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		FPGA Propagation Delay	0	2ns	
t_{BIO}		Buffer Input-Output Delay		4.5ns	
t_{BOE}		Buffer Output Enable Delay		6.5ns	
t_{CW}		Chip selection to end of write ($\overline{\text{CS1}}$, CS2 hold time)	75ns		
t_{WC}		Write cycle time	85ns		
t_{WP}		Write pulse width ($\overline{\text{WE}}$ hold time)	55ns		
t_{AS}		Address setup time	0ns		
t_{WHZ}		Write output to high-Z (don't drive data during this delay)		30ns	
t_{DW}		Data to write setup time	30ns		
t_{DH}		Data hold from write time	0ns		
t_{WR}		Write recovery time	0ns		
t_{OHZ}		Output disable to output high-Z delay		25ns	

Table 8: Table of constraints of the write cycle of the RAM device, shown in Figure 21 and described in Section 2.8.1.

2.8.2. ROM

A AM29F040B 512 Kword x 8-bit ROM chip, *U14*, constitutes the system's non-volatile storage. The device's connections are shown in Figure 19. The device is accessible by the processor at addresses 0x180000-0x1FFFFF, as shown in Figure 4.

The chip is connected to the bottom 18 bits of the address bus (*A17..0*) and the bottom 8 bits of the data bus (*D7..0*), which are then routed through buffers *U8* and *U9* to *U2*, the FPGA. Note that both buses are shared between multiple memory devices.

U14 is selected by using active-low signal *CS0* uniquely, which is then routed through buffer *U9* and into *U2*. The chip is configured as always-enabled by pulling the active-low output-enable line low through *R33*. Writing to the device is always disabled, since the device is read-only, by pulling the active-low write-enable line high through *R32*.

The interactions between the processor and the memory device, which follow a generic memory controller interaction model, are illustrated in the timing diagram of Figure 22 and Table 9. Note from the diagram that the device requires 5 wait states when reading.

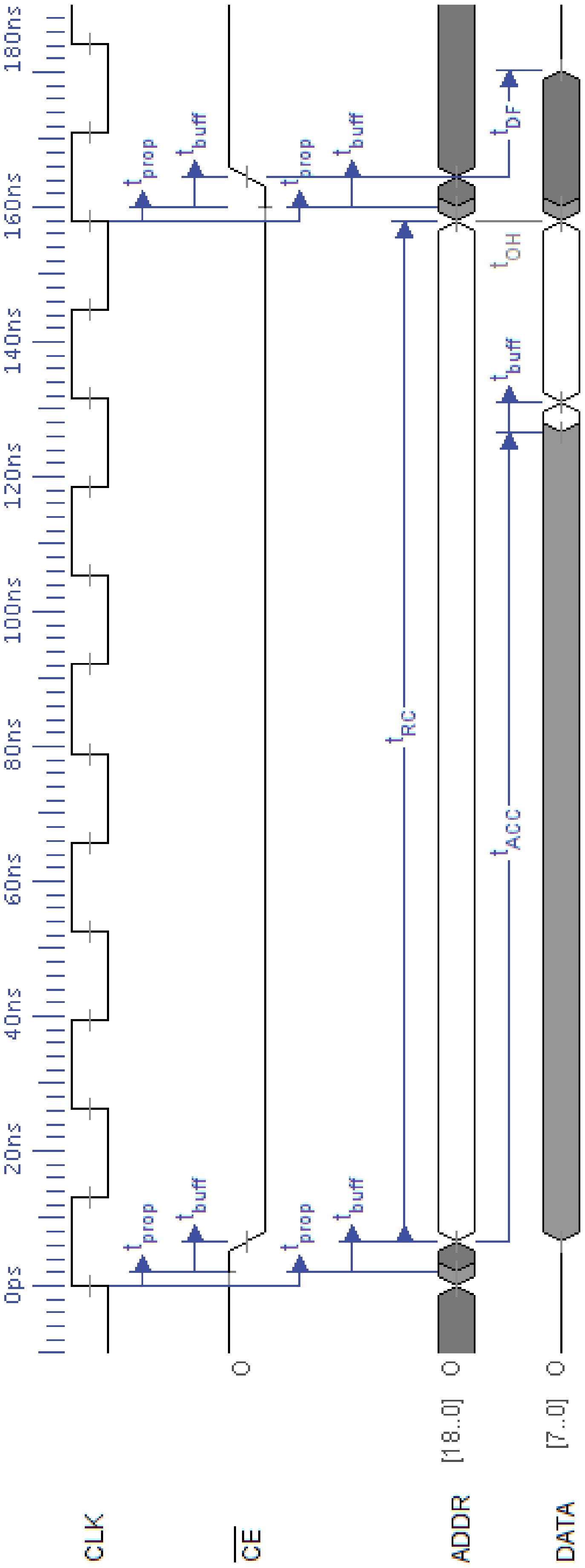
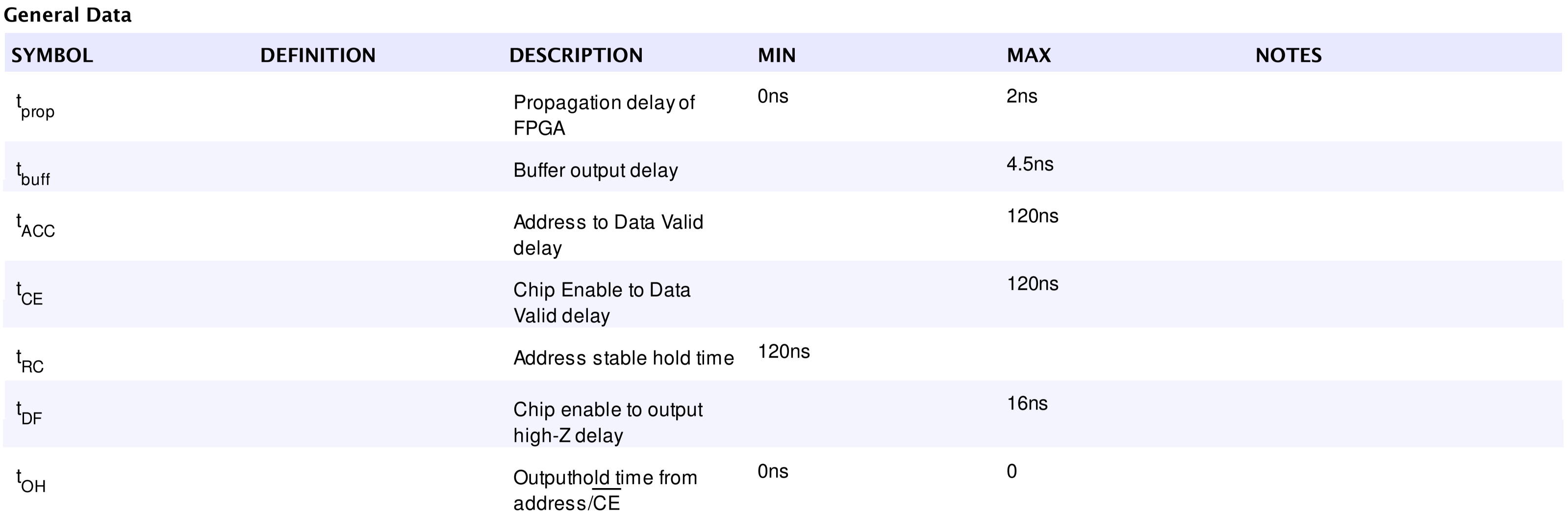


Figure 22: Timing diagram of the read cycle of the ROM device, described in Section 2.8.2.



General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_{prop}		Propagation delay of FPGA	0ns	2ns	
t_{buff}		Buffer output delay		4.5ns	
t_{ACC}		Address to Data Valid delay		120ns	
t_{CE}		Chip Enable to Data Valid delay		120ns	
t_{RC}		Address stable hold time	120ns		
t_{DF}		Chip enable to output high-Z delay		16ns	
t_{OH}		Outputhold time from address/$\overline{CE}$	0ns	0	

Table 9: Table of constraints of a display cycle, shown in Figure 22 and described in Section 2.8.2.

2.8.3. VRAM

Two MSM5416283 512 Kword x 16-bit VRAM chips, *U20* and *U21*, are connected "in parallel" to form a single virtual 512 Kword x 32-bit memory device. The devices' connections are illustrated in Figure 23. The device is accessible by the processor at addresses 0x00000-0xFFFFF, as shown in Figure 4.

In this configuration, the whole video address bus (*VA8..0*) is shared between the two devices, and bridged through buffer *U1* into the FPGA, *U2*. The data bus, on the other hand, is split: *U21* is connected to the bottom 16 bits of the data bus (*U15..0*), while *U20* is connected to the top 8 bits (*U23..16*). Note that eight of the data lines at *U20* are left unconnected, since only 24 bits of data are used. The data bus is shared with other memory devices, and relayed by buffers *U6* and *U9* into *U2*, the FPGA.

The serial outputs of both memory devices are output to the display on connector *J6*. The bottom byte of *U21* (*SDQ7..0*) is used as the red channel in the display (*R7..0*); the top byte of *U21* (*SDQ15..8*) is used as the green channel in the display (*G7..0*); the top byte in *U20* (*SDQ15..8*) is the blue channel in the display (*B7..0*).

With the parallel configuration of the two chips, *U20* and *U21* both share the same signals for *SCLK*, *TRG*, *CAS*, *RAS*, and the combination of *WEL* and *WEU*, *WEL/U*. These signals are routed through buffer *U1* into the FPGA, *U2*.

On both chips, *SOE* is tied low through *R68* and *R69* to permanently enable the serial interface of the devices. *DSF* is tied low through *R70* and *R71* to disable special functions. *QSF* is unused, and thus left floating.

The interactions between the processor and the VRAM are mediated by the VRAM controller, and are thus described in Section 2.2.5. As far as the processor is concerned, it can access the device following a generic memory controller interaction model with variable wait states and wait signal *!WAIT*.

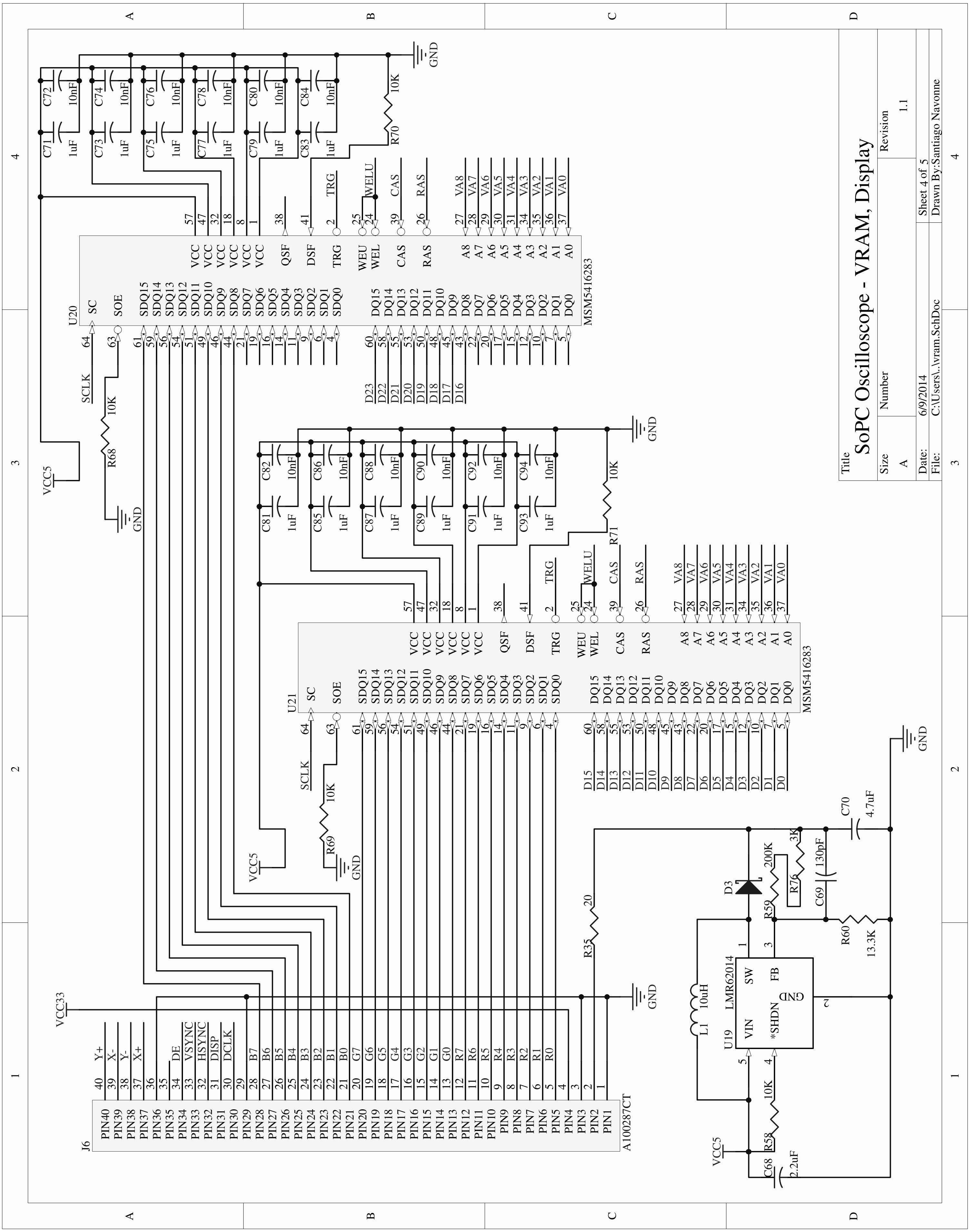


Figure 23: Schematic of the VRAM memory devices and display interface. Also shown is the section of the power supply used to power the display's LEDs. Further details are provided in Section 2.8.3.

2.9. Display

A NHD-4.3-480272-EF-ATXL#-T 4.3 inch, 480x270 pixel, color, touch screen display is used in the system. The display uses a HX8257 LCD driver, which is controlled by the display controller described in Section 2.2.6.

The display is connected through Molex connector *J6*, as shown in Figure 23. The 24 data lines for the three colors output by the VRAM memory devices (*R0..8*, *G0..8*, *B0..8*) are output on pins 5 through 28. The display control signals output by the display controller (*DCLK*, *DISP*, *HSYNC*, *VSYNC*), routed through buffers *U1* and *U7*, are output on pins 30 through 33. Pin 43 is used for the *DE* signal, which is unused and therefore left floating.

Pins 37 through 40 are connected to the display's touch screen, and are therefore used by the touch screen controller, *U18*. Pin 2 is used to drive the display backlight's LEDs, and is therefore tied to the +20 V power supply through a current-limiting 20 Ω resistor.

The display connector is highlighted in pink in Figure 2.

2.10. Touch Screen Controller

The display's touch screen is interfaced with using a TSC2003 touch screen controller. The connections of this device are shown in Figure 24. Note that the device remains unimplemented in the system's software, and therefore its hardware is not fully tested.

The *X+*, *Y+*, *X-*, *Y-* lines are connected to the corresponding pins on the display connector, *J6*. The I^2C bus lines, *SCL* and *SDA*, are connected directly to the FPGA and into a currently unimplemented I^2C controller. Lines *A1,0* are tied low to configure the address of the device on the I^2C bus. **PENIRQ* goes low when the screen it touched, signaling a touch screen event to the processor; this line is therefore relayed by buffer *U7* into the FPGA, *U2*, and then made accessible to the processor as PIO.

The full functionality of the chip is not used, and therefore monitoring pins *VBAT1,2* and *IN1,2* are simply tied low through resistors *R50,51* and *R46,47*.

The touch screen controller is highlighted in cyan in Figure 2. Note that all bypass capacitors are placed on the backside of the board, drawn in Figure 3.

2.11. Rotary Encoders

Two rotary encoders with temporary push-buttons are used to provide the main user input interface for the system. The devices' connections are illustrated in Figure 24. Once debounced and decoded within the FPGA, their signals are made available to the NIOS processor on the *PIO_0* interface, at addresses 0x2410A0-0x2410BF.

Each one of the signals (*ROT0A,B*; *ROT1A,B*; *SW0,1*) is pulled high through resistors *R52-57*. As the rotary encoders are turned, the rotation signals are shorted to the *COM* line, which is tied to ground. Similarly, as either push-button is pressed, the *SW* lines are shorted to ground. The signals are then debounced and decoded as described in Sections 2.2.3 and 2.2.4, making user input available to the processor.

The rotary encoders are highlighted in brown in Figure 2. Note that the pull-up resistors are placed on the backside of the board, drawn in Figure 3.

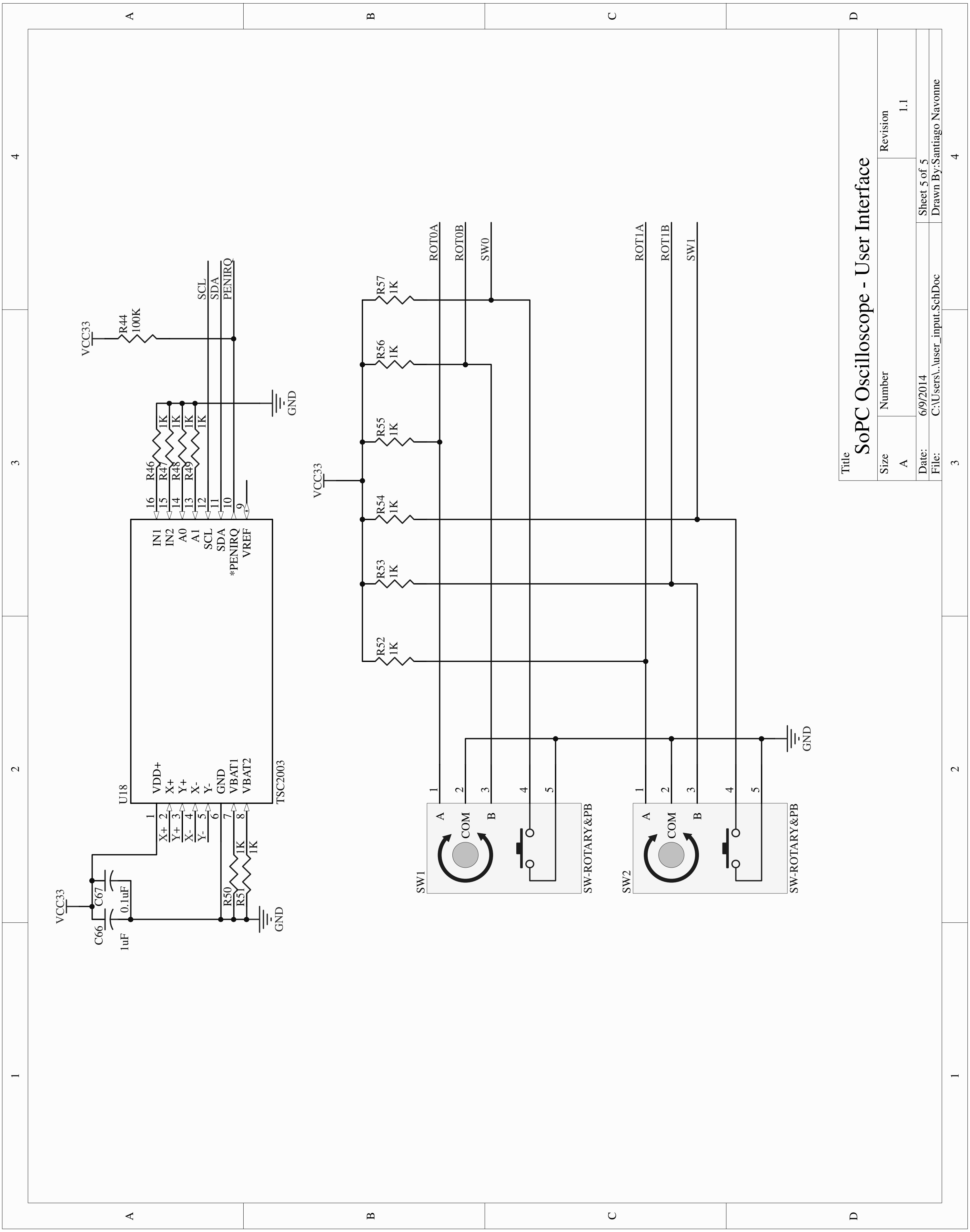


Figure 24: Schematic of the user input section of the system. This includes the rotary encoders, described in Section 2.11, and the touch screen controller, described in Section 2.10.

2.12. Analog Interface

Analog samples are acquired from the probe through the analog interface here described, and then made available to the processor via the triggering mechanism described in Section 2.2.2. The analog interface schematic is illustrated in Figure 25.

The signal is acquired through the probe connected to BNC connector *J5*. An alternative connector is provided through two-pin header *P1*. The positive line is then connected to +12 V and -12 V through Schottky diodes, ensuring that no voltages outside that range will ever reach components forward of this point. Note that this functionality remains untested.

JP1 provides an easy means of selecting whether to scale and shift the signal using the Analog Front-End (Section 2.12.1), or skip the section altogether and input the signal directly to the Analog-to-Digital Converter (Section 2.12.2). For the correct operation of the system, *JP1* should be configured to the FE position.

The analog interface is highlighted in orange in Figure 2. Note that all bypass capacitors are placed on the backside of the board, drawn in Figure 3. Ground planes are placed below the region in order to reduce noise.

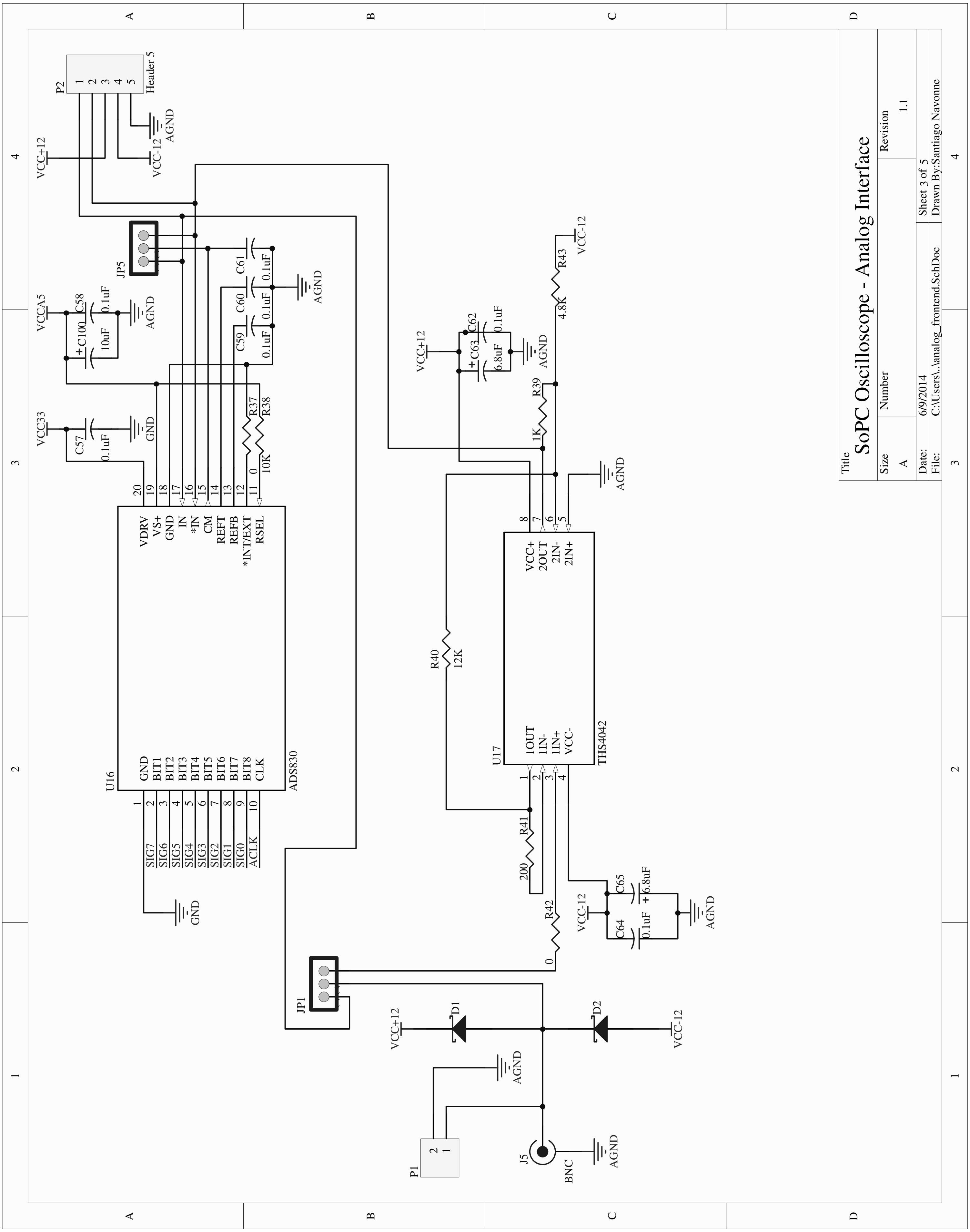


Figure 25: Schematic of the system's analog interface, described in Section 2.12. This includes the analog front-end and the analog-to-digital converter (ADC).

2.12.1. Analog Front-End

The system incorporates an analog front-end that scales and shifts the signal to allow for an increased voltage range. Thanks to this section of the circuit, the system is able to accept signals from -10 V to +10 V. These voltages are thus scaled to the Analog-to-Digital Converter's (ADC) voltage swing (±1 V), and shifted to its common mode voltage (+2.5 V).

To this end, the operational amplifier circuit of Figure 26 is used. A THS4042 165 MHz, dual op-amp chip was used. The first stage in this circuit, which corresponds to pins *1IN+*, *1IN-*, and *1OUT* at *U17*, and resistor *R41* in Figure 25, is a simply buffer, used to provide high-impedance to the input circuit: it is vital that the oscilloscope do not disturb the circuit being measured. The second stage, made up of pins *2IN-*, *2IN+*, and *2OUT* at *U17*, and resistors *R40*, *R39*, and *R43*, is a shifting-scaling stage. This part of the circuit adds one twelfth of the input signal (scaling it down from ±12 V to ± 1 V) to $-12V/4.8 = -2.5V$ (shifting it down to -2.5 V CM), and inverts the result. The output is the mirror image of the input signal ("negative" the signal), scaled down to one twelfth, and shifted to +2.5 V CM. This is exactly the input required by the inverted signal input of the ADC.

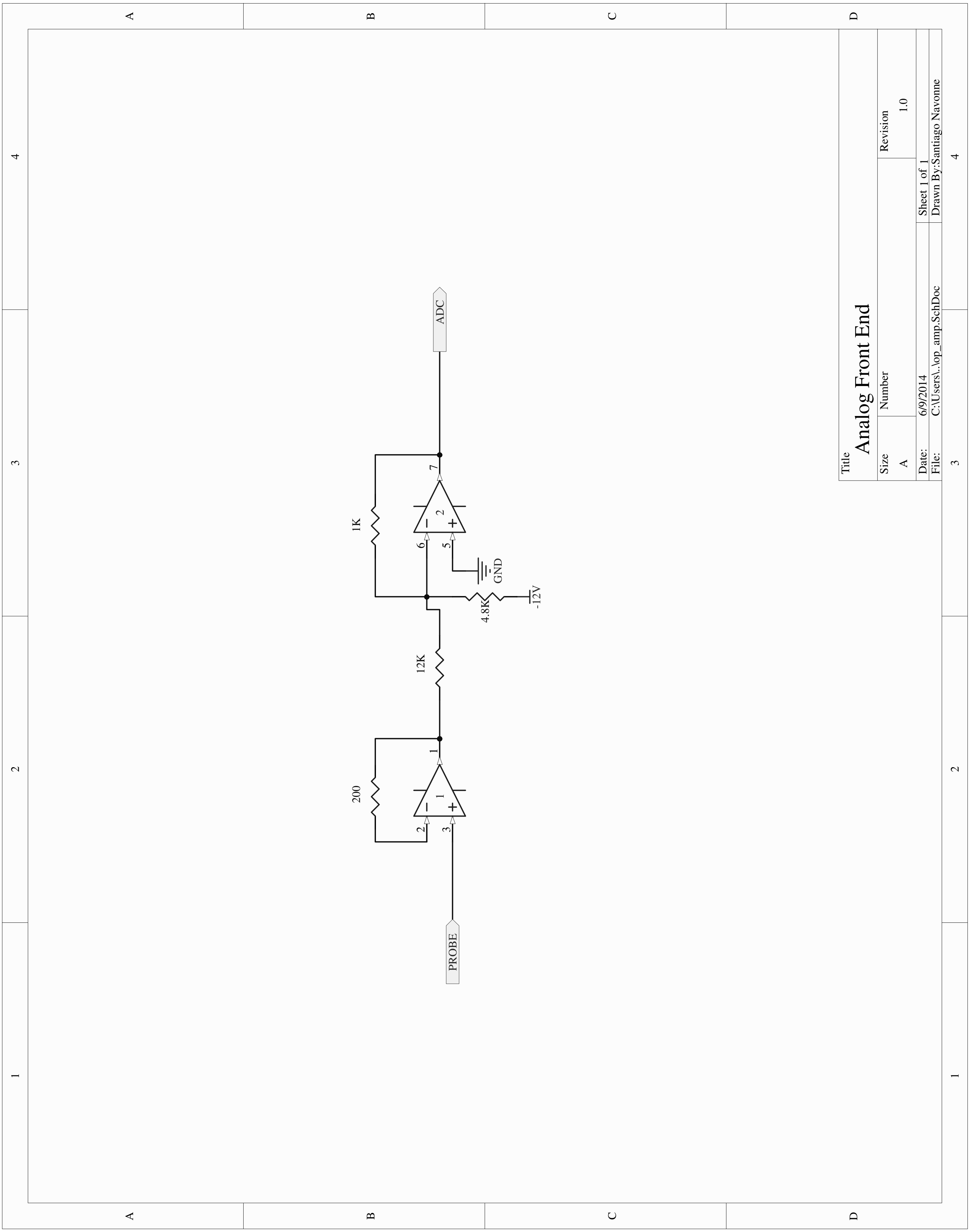


Figure 26: Symbol schematic of the operational amplifier scaling and shifting circuit used in the system's analog front-end, and described in Section 2.12.1.

2.12.2. Analog to Digital Converter

The analog to digital converter takes either the input signal or the scaled output of the op-amp circuit of Section 2.12.1, and converts it to a digital value. The digital value is output on lines *SIG7..0*, which are then bridged through buffer *U3* into the FPGA, *U2*. The digital conversion is clocked by *ACLK*, which is output by *U2* through buffer *U7*. The timing of a ADC clocking interaction is shown in Figure 27 and Table 10.

The input signal is placed on either the regular input pin *IN* or the inverted input line **IN*. The other pin must be tied to the common mode pin *CM* using jumper *JP5*. For the correct operation of the system, *JP5* must be configured to the FE position. A 5-pin header, *P2*, is provided to allow for the substitution of the analog front-end with an alternative circuit. The **INT/EXT* configuration line is tied low to select the internal reference, and *RSEL* is tied high through *R38*.

Note that the digital side of the chip is placed outside of the analog region on the PCB of Figure 2, and that the device is therefore placed at the edge of the analog region. The analog ground is separated from the digital ground.

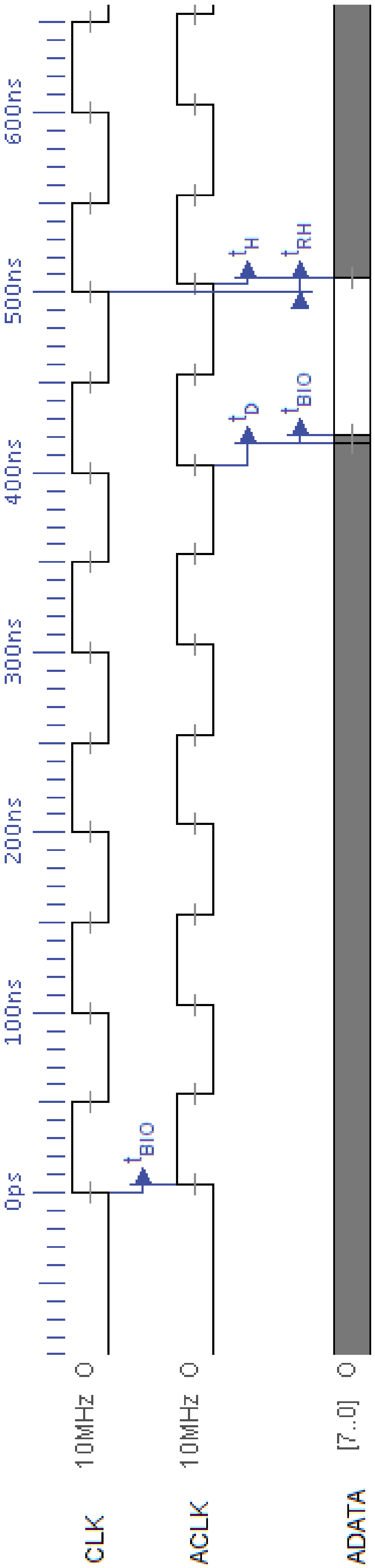
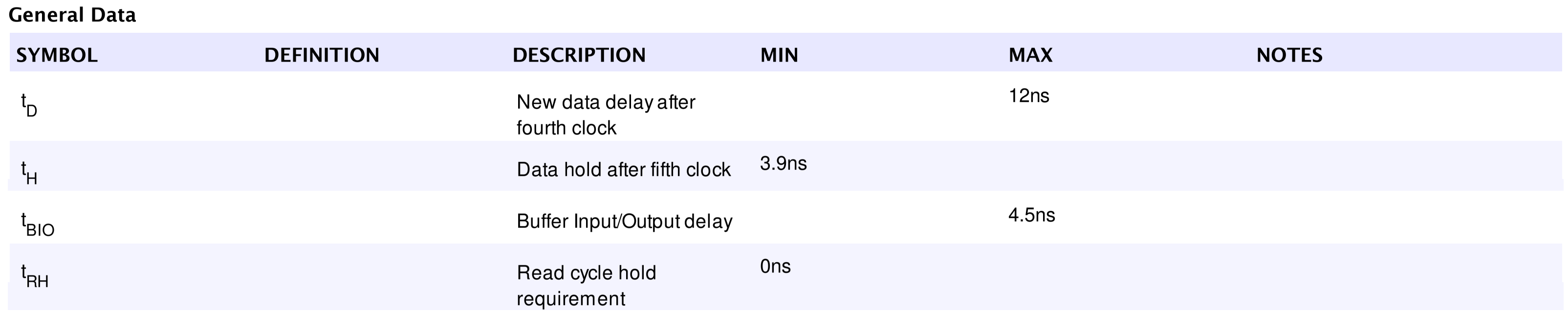


Figure 27: Timing diagram of an ADC sampling clock, described in Section 2.12.2.



General Data

SYMBOL	DEFINITION	DESCRIPTION	MIN	MAX	NOTES
t_D		New data delay after fourth clock		12ns	
t_H		Data hold after fifth clock	3.9ns		
t_{BIO}		Buffer Input/Output delay		4.5ns	
t_{RH}		Read cycle hold requirement	0ns		

Table 10: Table of constraint for the ADC sampling clock cycle, shown in Figure 27 and described in Section 2.12.2.

2.13. Revision History

Table 11 provides a summary of the revisions made to the original design.

Date	Revision	Document(s)	Changes
March 2014	1.0	All	Initial revision.
June 2014	1.1	User Input Schematic	Changed rotary encoder pull-up resistors to $1k\Omega$.
			Added pull-up resistor on PENIRQ line.
		FPGA Schematic	Corrected JTAG connector wiring.
			Fixed error in routing of A0.
		PCB	Changed voltage regulators' footprints.
		Analog Interface Schematic	Changed value of pull-down resistor at ADC.

Table 11: Revision history.

3. Software

This section describes how the system's software works, from the system overview to the detailed description of each element. The roles and interactions of the various part of the program are described. The actual code for the program is provided in Appendix B.

3.1. System Overview

The core of the system's software constitutes in the EE/CS 52 SoPC Oscilloscope software library, written in C, with minor modifications. The hardware interface procedures are written in NIOS assembly, and are specific to the hardware used in the system. The block and file structure of the project is shown in Figure 28.

When the system is started, the stack is automatically set up with other initialization, and *main* is run, within *mainloop.c*. This function, performs all required initialization and runs the system. The main loop makes use of several procedures: a set of user interface procedures found in files *lcdlout.c*, *keyproc.c*, *menu.c*, and *menuact.c* allow the system to interact with users through the UI; trace utility procedures in *tracutil.c* process acquired signals; key handling procedures in *keys.s* interface with the software to identify key presses and user actions; display controlling procedures in *display.s* provide a layer of abstraction for the specific LCD used; and analog interface procedures in *trigger.s* control the analog and triggering interface as needed.

Note that each file has a header file of the same name, but *.h* extension, associated with it, where functions are declared and constants defined. Additionally, *scopedefs.h* provides general project constants for C files, while *general.h* provides general constants for assembly files. *interfac.h* defines constants specific to the system's hardware interface. Finally, *system.h* contains memory map and IRQ number definitions, and is automatically generated by the Altera toolchain. The used version is also provided in Appendix B.

3.2. Initialization and Main Loop

When the system is started as after general initialization is performed by NIOS toolchain functions, *main* is called within *mainloop.c*, part of the Oscilloscope library. This function performs all additional initialization and runs the system. The procedure was modified to initialize the keys handler and the analog/triggering interface on start. For more details on the procedures within *mainloop.c*, please refer to the EE/CS 52 Oscilloscope library documentation.

3.3. User Interface

In order to interface with users, a number of functions are provided in the Oscilloscope library. *lcdout.c* provides procedures used to output data to the LCD; *keyproc.c* has functions that process the various available keys; *menu.c* constains the functions for processing menu entries; *menuact.c* includes the functions for carrying out menu actions. All of these functions utilize the hardware abstraction procedures provided in the assembly files. *lcdout.c* was modified to support user interface colors, and highlighted characters (used in the menu); *menuact.c* was modified to support a faster sweep rate, and to change the sweep rate values to the actual values obtained by dividing the system clock. For more details on the procedures within the user interface files, please refer to the EE/CS 52 Oscilloscope library documentation.

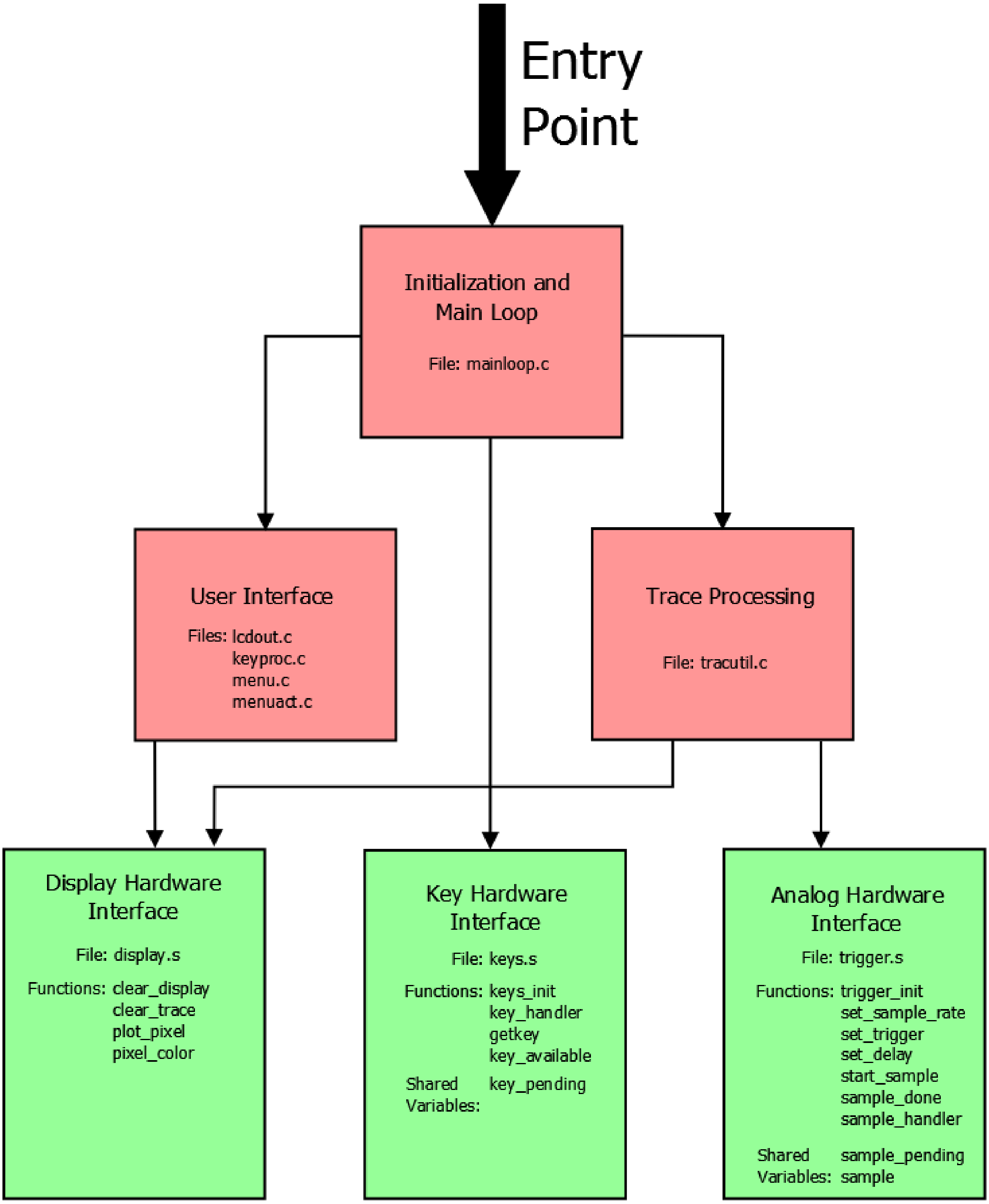


Figure 28: Block diagram of the SoPC Oscilloscope system's software. Blocks in red are part of the EE/CS 52 Oscilloscope library; blocks in green are hardware-specific and implemented for this system. The diagram is described in Section 3.1.

3.4. Trace Processing

In order to draw traces to the screen, capturing and outputting the acquired data, procedures within *tracutil.c*, provided by the Oscilloscope library, are used. This library file was modified to change the UI display colors (trace), and clear only the trace instead of the whole display when refreshing the sample, obtaining a better behavior of the display. These procedures employ analog and triggering hardware assembly procedures to perform the required actions. For more details on the procedures within *tracutil.c*, please refer to the EE/CS 52 Oscilloscope library documentation.

3.5. Key Hardware Interface

To interface with the system's push-buttons and rotary encoders, abstracted as keys, a set of procedures are provided in *keys.s*. These procedures translate user input actions into key values as needed, making them available to the system's software. The procedures use shared variable *curr_key* to store a pending key press, if any is available.

keys_init performs all the necessary initialization for the keys interface. It initializes shared variable *curr_key* and sets up interrupts as necessary, preparing the interface for use.

key_handler is executed whenever a key press interrupts occurs after initialization. The function identifies the key pressed, and saves it in buffer *curr_key* to make it available to *getkey* and *key_available*, and thus outside functions.

getkey returns the currently pending key press if one is available (i.e. present in the *curr_key* buffer). If none is available, the function blocks in a busy loop.

Finally, *key_available* checks whether a key press is pending (i.e. there's a valid value in *curr_key*). This procedure is normally called before *getkey* to avoid blocking.

For more details about the operation of these procedures, please refer to their definitions in *keys.s*.

3.6. Display Hardware Interface

To interface with the system's color display, a set of procedures are provided in *display.s*. These procedures communicate with the VRAM device as needed to control the display as requested. Note that the display requires no formal initialization; however, *clear_display* should be called at the start, since the image initially displayed is undefined.

clear_display completely clears the display, making every pixel in it black. This function should be called after initialization, since the image initially shown on the display is undefined.

clear_trace only clears the trace pixels on the display, that is pixels that are the color of the trace or of the cursor; the procedure turns these pixels black. The function is currently unused, but is still provided as it can simplify the implementation of additional features, such as a cursor.

plot_pixel changes the color of one pixel at a given location to an RGB value.

Finally, *pixel_color* accesses the color, as an RGB value, currently being displayed at a given location. This procedure is currently unused, but is still provided as it can simplify the implementation of additional features, such as a cursor.

For more details about the operation of these procedures, please refer to their definitions in *keys.s*.

3.7. Analog Hardware Interface

To control the analog and triggering interface, starting and acquiring samples and configuring the trigger parameters, a set of procedures are provided in *trigger.s*. These procedures output the necessary signals to the triggering logic, causing it to acquire samples as needed. Additionally, they transfer samples to the main code when they are requested after completion. The procedures use shared variable *sample_pending* to keep track of the currently started sample, as well as buffer *sample* to extract the samples from the FIFO and return them to the caller.

trigger_init performs all necessary initialization for the analog and triggering interface. it initializes shared variable *sample_pending*, sets up interrupts as needed, and resets the triggering logic, preparing the interface for use. The triggering logic is also reset.

set_sample_rate allows the caller to configure the sampling rate to any positive number of samples per second less than or equal to the system clock divided by two (19.5 Msamples per second). The frequency is translated to a number of system clocks per sample, and the value is then output to the triggering hardware block. The triggering logic is then reset. The number of samples that will be acquired at the configured rate is then returned, but note that this value is always the same, and equal to the width of the display (which must be less than or equal to 512 per hardware limitations).

set_trigger configures the trigger level and slope. The trigger level is translated to the corresponding level in the correct range, and then the its value is output together with the slope to the triggering hardware block. The triggering logic is then reset.

set_delay configures the trigger delay to any positive, unsigned 32-bit number of samples less than $2^32 - 2$. The value is corrected to take into account any hardware limitations (delay must be positive), and the value is then output to the triggering hardware block. The triggering logic is then reset.

start_sample starts a new sample with the previously configured settings. The sample can trigger automatically or manually, as configured via the procedure's argument. The sample is thus started in hardware.

sample_done checks whether the previously started data sample has been completed and is thus available by reading shared variable *sample_pending*. If the sample has been completed, it is at this point extracted from the FIFO and returned in a buffer, while variable *sample_pending* is reset. If no sample is available, a null pointer is provided.

Finally, *sample_handler* is executed any time the FIFO finished being filled. The procedure simply updates shared variable *sample_pending* to indicate that there is a sample that can be extracted by *sample_done*.

Appendices

A. Original Documents

Figures 29, 30, 31, 32, 33 show the schematic of the original design; Figures 34, 35 show the resulting PCB used in the prototype. Note that the RAM and ROM section is identical to the one described in Sections 2.8.1 and 2.8.2. The changes made and their reasons are summarized below.

After noticing that the footprints used for the voltage regulators (*U11-13*) in revision 1.0 were incorrect (power and ground pins were switched), these were corrected in revision 1.1. The bottom bit of the address bus (*A0*) but was incorrectly routed through a buffer direction pin; the line was thus re-routed in the newest revision. There was a mistake in the wiring of the JTAG connector (*J1*), where pins 9 and 10 were switched.

Additionally, the values of some resistors had to be changed after noticing too big voltage drops across them: the pull-up resistors at the rotary encoders (*R52-57*) had to be decreased from 10kΩ to 1kΩ, and the pull-down resistor at the ADC's **INT/EXT* pin (*R37*) had to be shorted.

Finally, a missing pull-up resistor had to be added to the touch screen controller's PENIRQ line in order to keep it from floating when not active (*R44*).

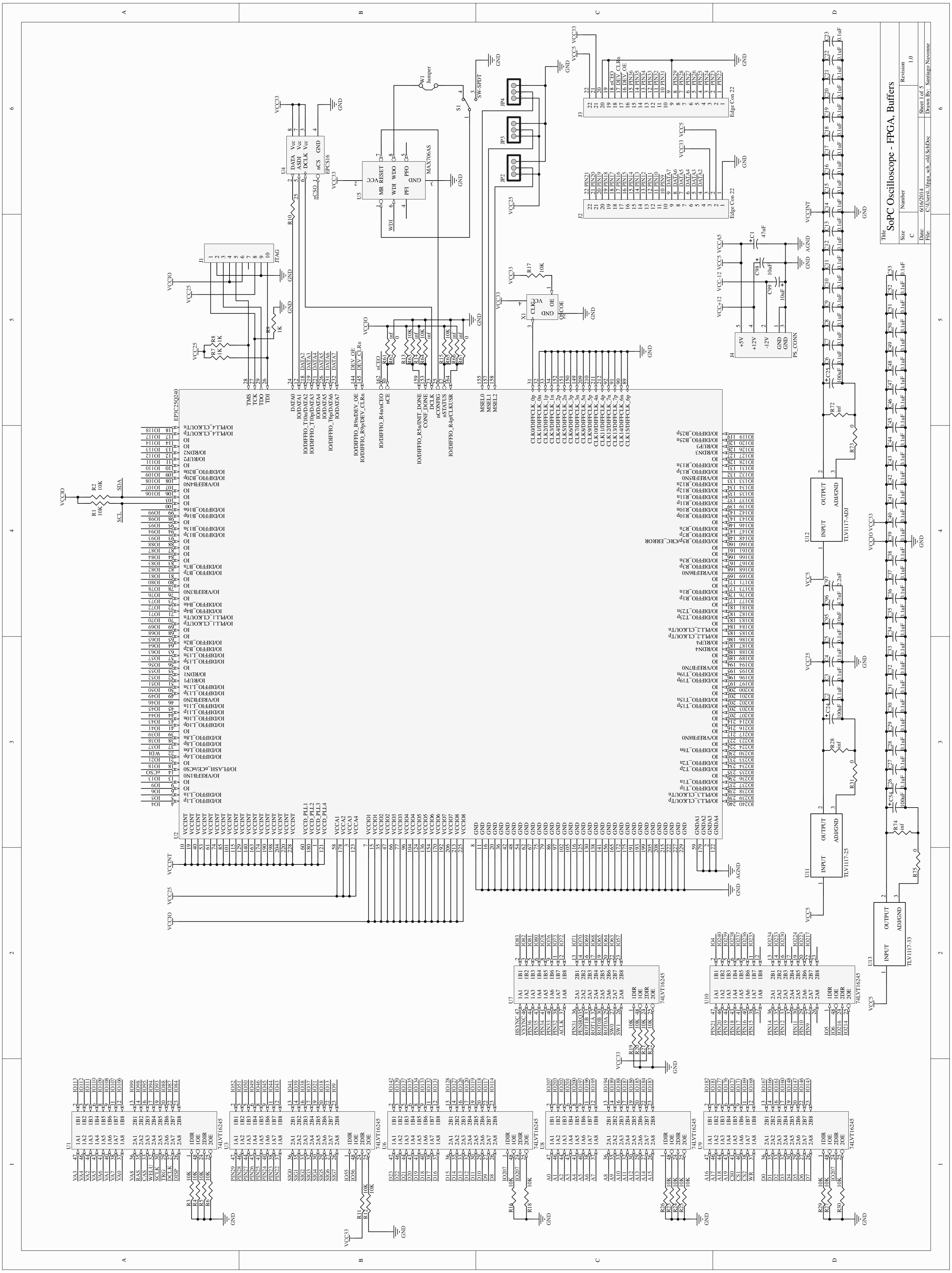


Figure 29: Original revision of the FPGA and related components schematic.

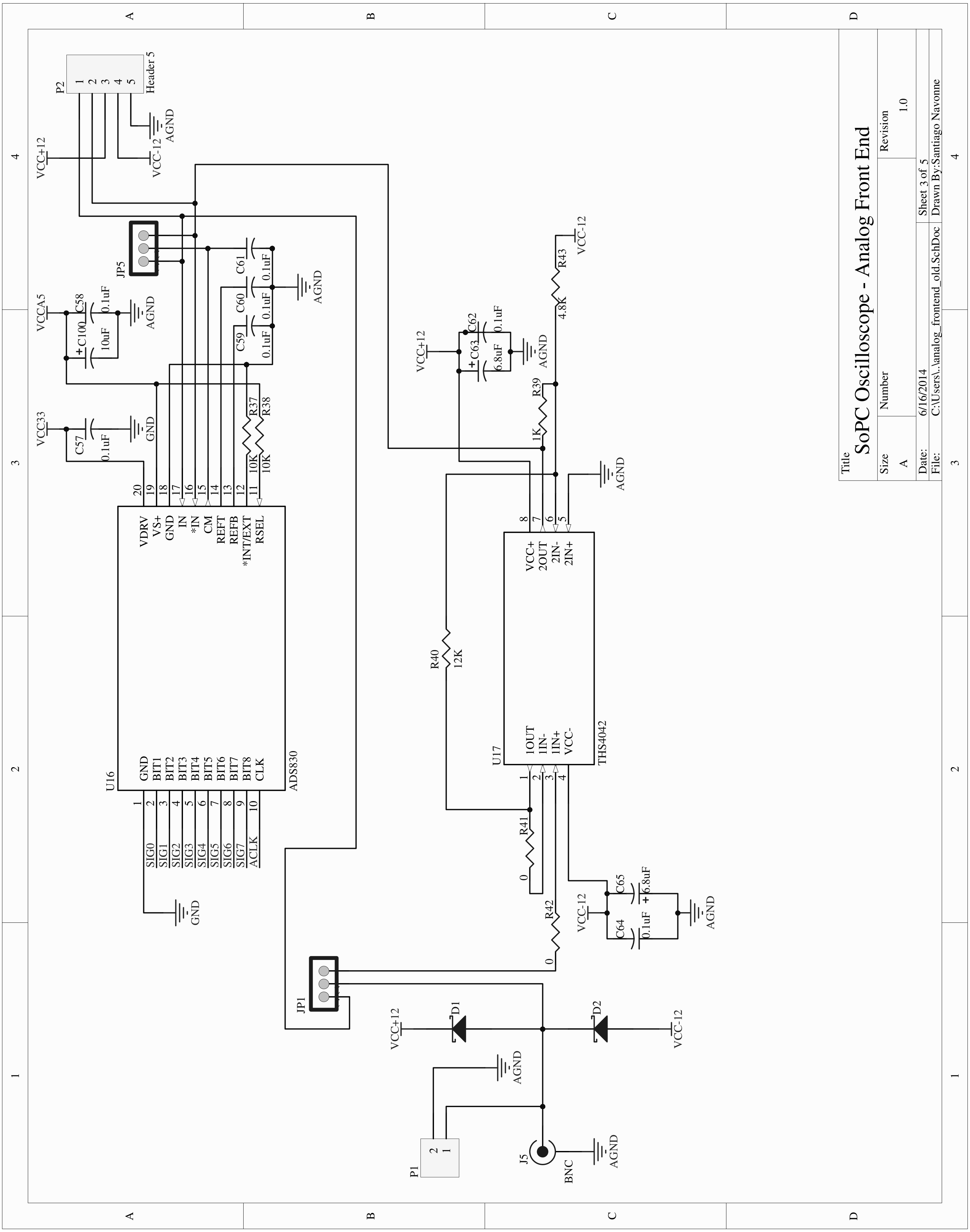


Figure 30: Original revision of the analog interface schematic.

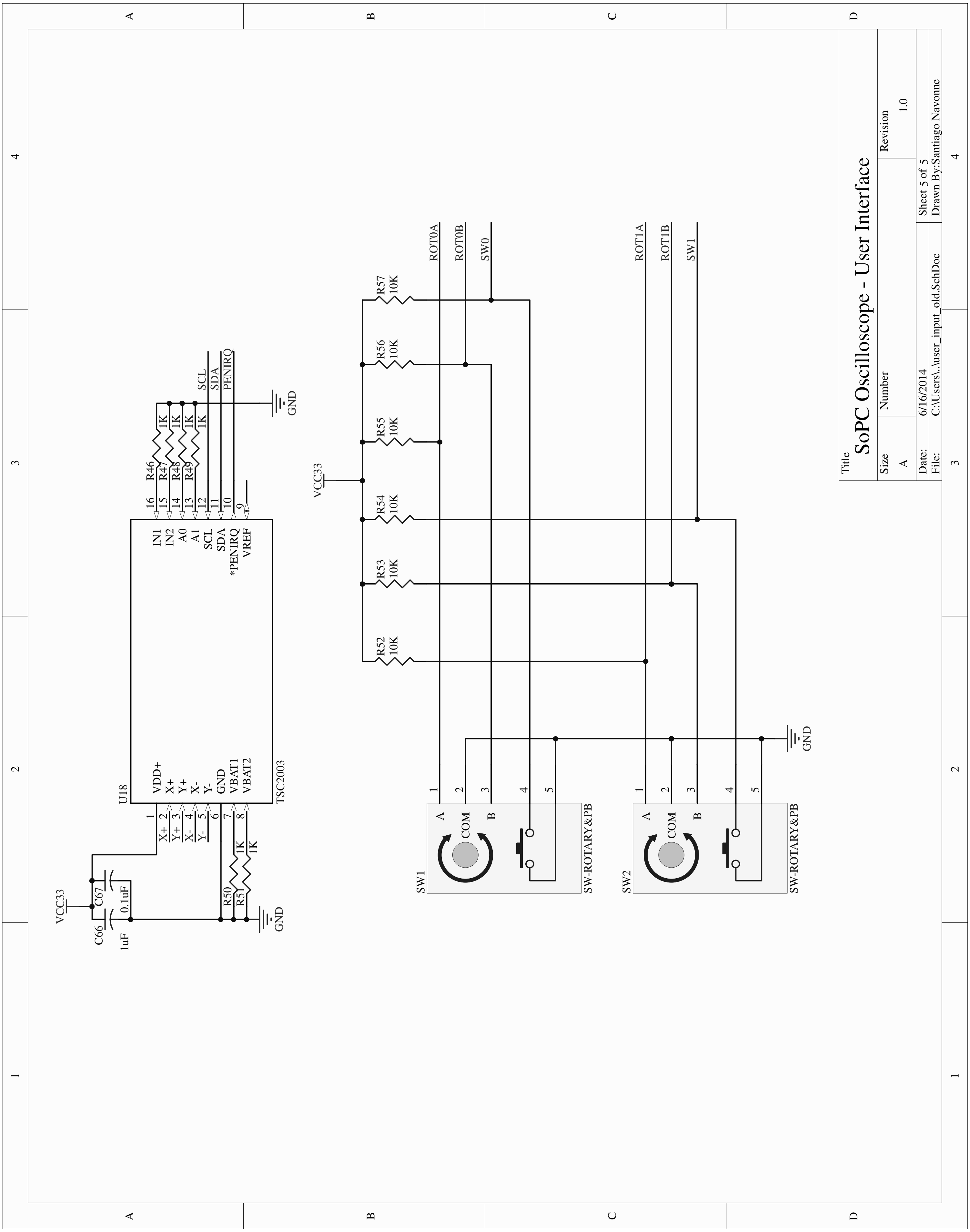


Figure 31: Original revision of the user input components schematic.

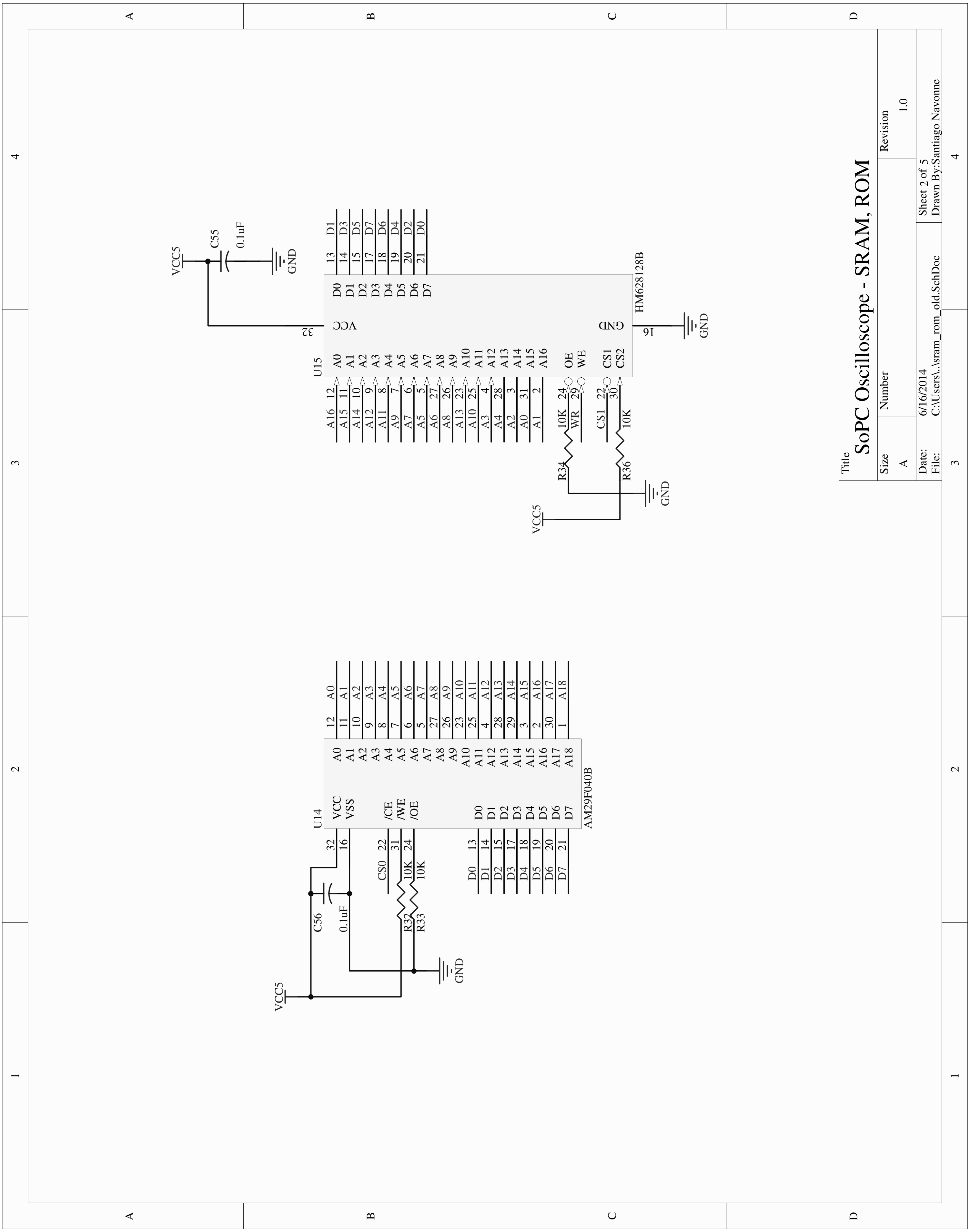


Figure 32: Original revision of the SRAM and ROM devices schematic.

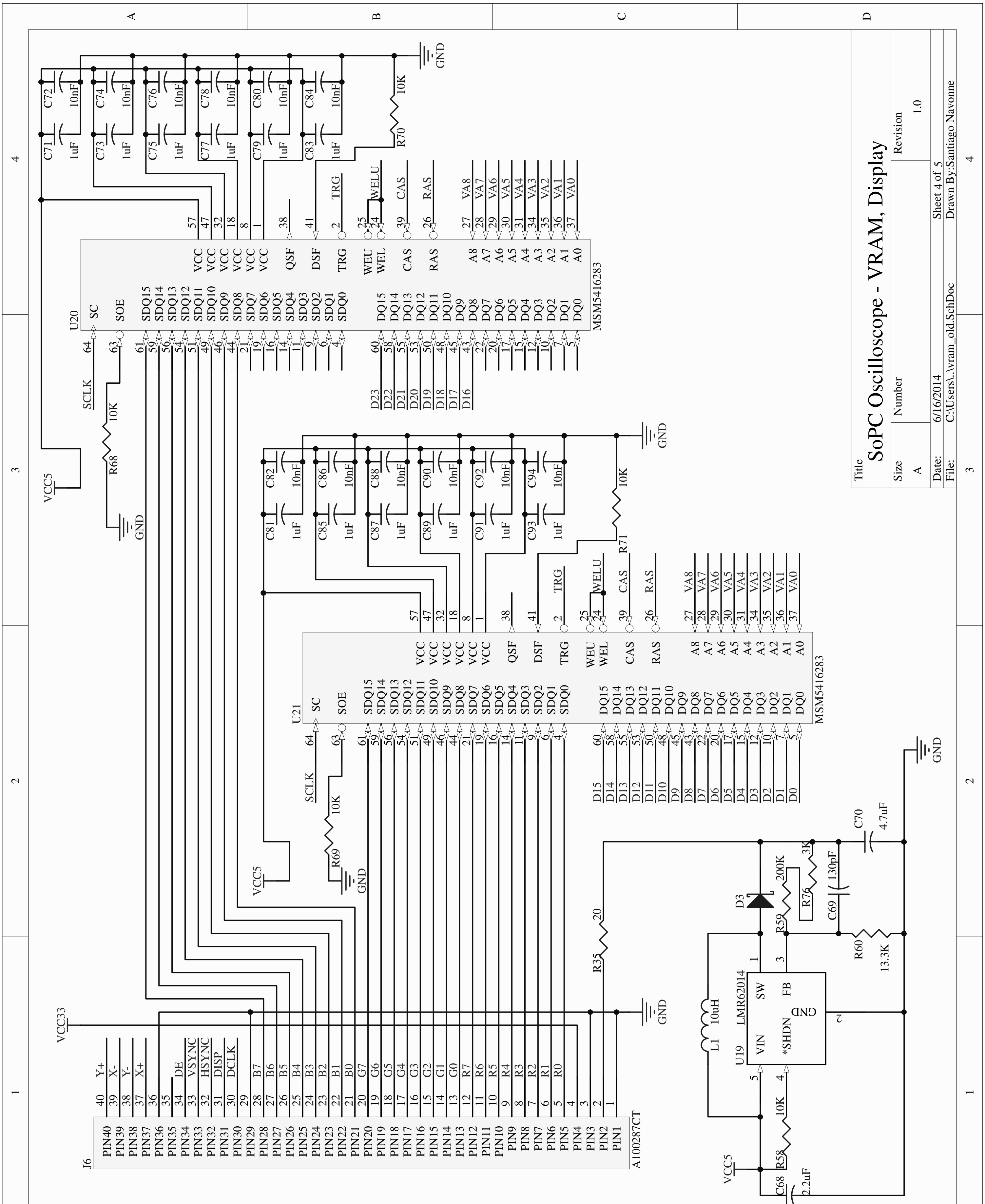


Figure 33: Original revision of the VRAM and display connector schematic.

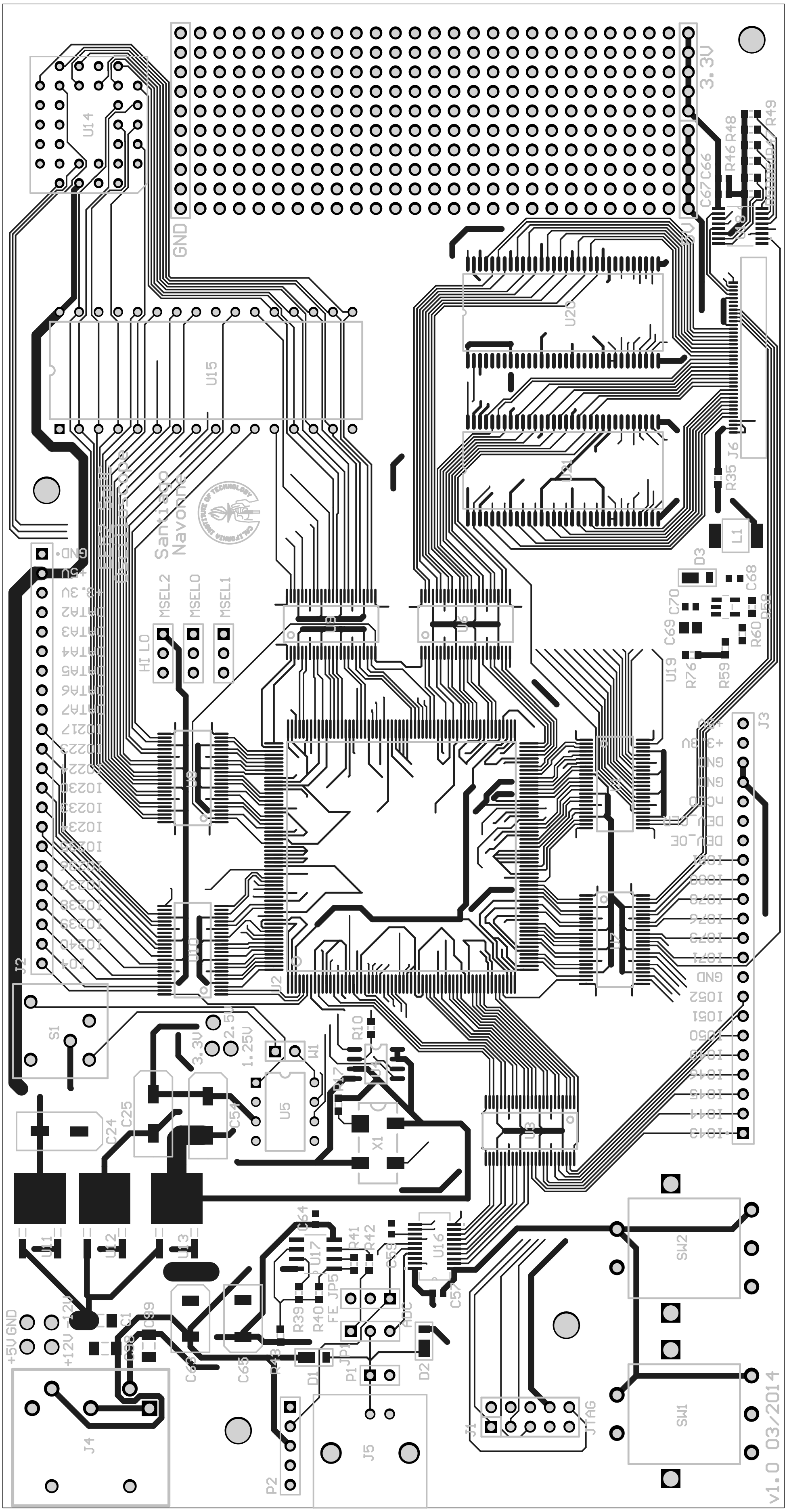


Figure 34: Front side of the original revision of the system's Printed Circuit Board (PCB).

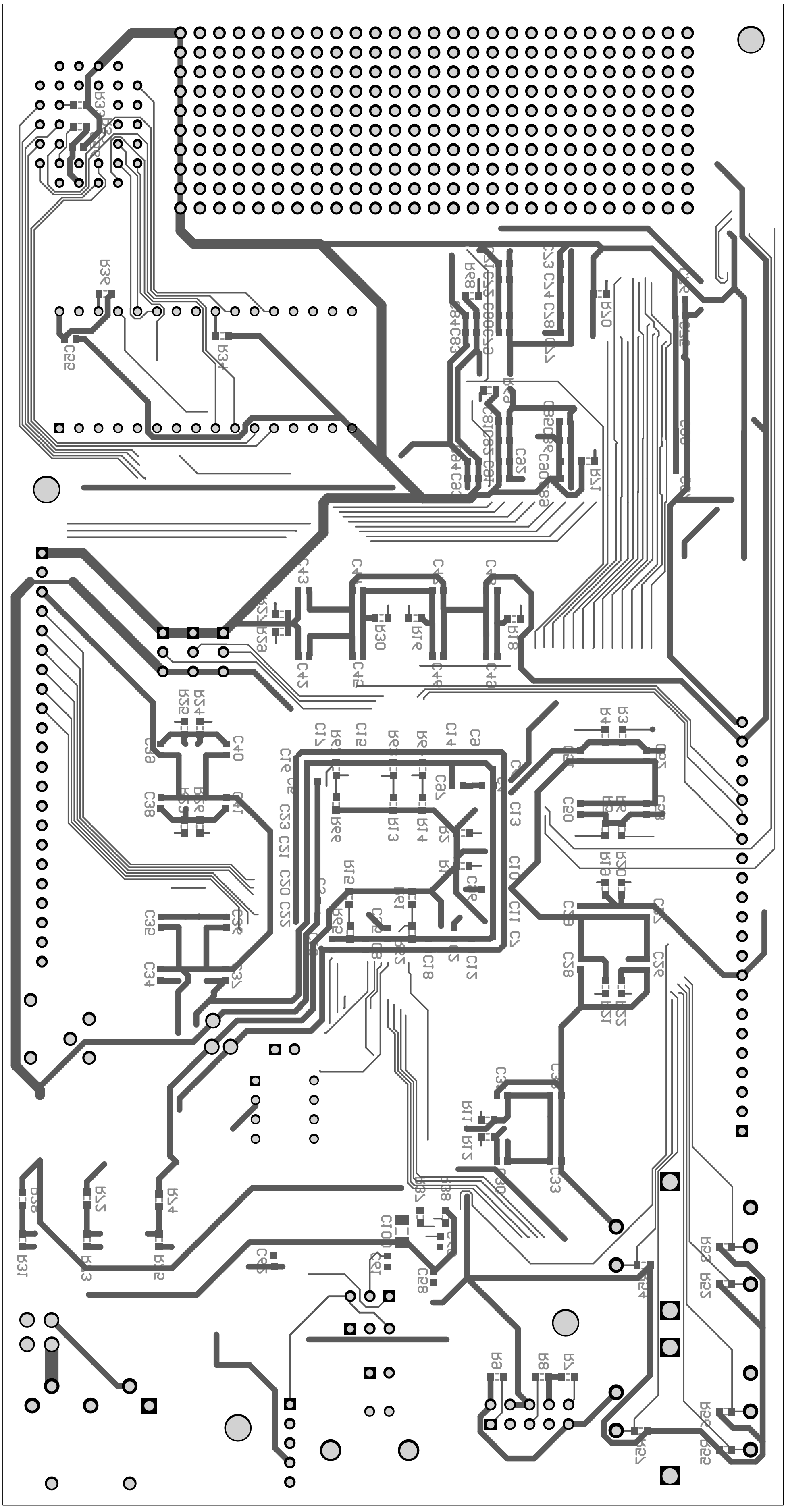


Figure 35: Back side of the original revision of the system's Printed Circuit Board (PCB).

B. Software Code

In this appendix, all the code contained within the program's software is provided. Table 12 shows a quick overview of the various files and their contents for quick reference, in alphabetical order.

Table 12: Table of Contents for system's software code.

```
/****************************************************************************/
/*                                                                          */
/*                                 DISPLAY.S                                */
/*                        Display Interface Functions                       */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                             Santiago Navonne                             */
/*                                                                          */
/****************************************************************************/

/*
   Display interface and control routines for the EE/CS 52 Digital Oscilloscope
   project. Function definitions are included in this file, and are laid out
   as follows:
    - clear_display: Completely clears the display;
    - clear_trace: Clears the pixels on the display that are the color of the
                   trace;
    - plot_pixel: Changes the color of the pixel at a given location;
    - pixel_color: Accesses the color of the pixel currently being displayed at
                   a given location.


   Revision History:
      6/3/14  Santiago Navonne  Initial revision.
*/

#include "general.h"
#include "system.h"
#include "interfac.h"
#include "display.h"


.section .text  /* Code starts here */


/*
 *  clear_display
 *
 *  Description:       This procedure clears the display, setting the color of every
 *                     pixel to black immediately.
 *
 *  Operation:         The procedure loops through every pixel in the display-mapped
 *                     region of the VRAM, storing 0 (black; clear pixel) into every
 *                     location.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            Clears every pixel on the display (changes color to black).
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r8, r9, r10, r11, r12.
 *
 *  Revision History:
 *      6/03/14   Santiago Navonne     Initial revision.
 *
 */
    .global clear_display
clear_display:                 /* clear the whole display */
    MOVHI   r8, %hi(VRAM_BASE) /* start at base of VRAM */
    ORI     r8, r8, %lo(VRAM_BASE)
    MOVI    r9, SIZE_X         /* and will loop through all columns */
    MOVI    r10, SIZE_Y        /*  and rows */
    MOV     r11, r0            /* starting at coordinates (0, 0) */
    MOV     r12, r0            /* (top left corner) */

row_loop:                      /* go through an entire row */
    STWIO   r0, (r8)           /* first clear the current pixel */
    ADDI    r8, r8, WORD_SIZE  /* then go to next column */
    ADDI    r11, r11, 1        /*  also incrementing the index */
    BLT     r11, r9, row_loop  /* and if we're still within display, repeat */

next_row:                      /* move to next row */
    ADDI    r8, r8, REMAINDER  /* add the remainder to finish up a VRAM row */
    MOV     r11, r0            /* reset the column index */
    ADDI    r12, r12, 1        /* and increment the row index */
    BLT     r12, r10, row_loop /* if we're still within display, repeat */

    RET                        /* all done, so return */

/*
 *  clear_trace
 *
 *  Description:       This procedure clears the trace from the display, changing the
 *                     color of every pixel that is currently the trace or cursor color
 *                     to black.
 *
 *  Operation:         The procedure loops through every pixel in the display-mapped
 *                     region of the VRAM. For every location, if the current value
 *                     matches either trace or cursor colors (both part of the trace)
 *                     the pixel is cleared by storing 0 into that memory location.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            Clears every trace pixel on the display (sets color to black).
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r8, r9, r10, r11, r12, r14, r15.
 *
 *  Revision History:
 *      6/03/14   Santiago Navonne     Initial revision.
 *
 */
    .global clear_trace_old
clear_trace_old:                   /* clear all trace pixels on display */
    MOVHI   r8, %hi(VRAM_BASE) /* start at base of VRAM */
    ORI     r8, r8, %lo(VRAM_BASE)
    MOVHI   r13, %hi(PIXEL_TRACE) /* load colors that will be cleared */
    ORI     r13, r13, %lo(PIXEL_TRACE)
    MOVHI   r14, %hi(PIXEL_CURSOR)/* which are trace and cursor */
    ORI     r14, r14, %lo(PIXEL_CURSOR)
    MOVI    r9, SIZE_X         /* will loop through all columns */
    MOVI    r10, SIZE_Y        /*  and all rows */
    MOV     r11, r0            /* starting at (0, 0) */
    MOV     r12, r0            /* (top left corner) */

trace_check:                   /* check if current pixel is part of trace */
    LDWIO   r15, (r8)          /* read value from VRAM */
    BEQ     r13, r15, trace_clear /* definitely clear if color is trace color */
```

```

cursor_check:                  /* check if current pixel is part of cursor */
    BNE     r14, r15, trace_row_loop /* also clear if part of cursor */

trace_clear:                   /* pixel is part of trace or cursor */
    STWIO   r0, (r8)           /*  so clear it */

trace_row_loop:                /* done with current pixel */
    ADDI    r8, r8, WORD_SIZE  /*  so go to next */
    ADDI    r11, r11, 1        /*  and also increment column index */
    BLT     r11, r9, trace_check /* if still within display, repeat */

trace_next_row:                /* done with current row */
    ADDI    r8, r8, REMAINDER  /* add remainder to finish up VRAM row */
    MOV     r11, r0            /* reset column index */
    ADDI    r12, r12, 1        /*  and increment row index */
    BLT     r12, r10, trace_check /* if still within display, repeat */

    RET                        /* all done, so return */


/*
 *  plot_pixel
 *
 *  Description:       This procedure changes the color to the pixel at the passed x, y
 *                     coordinates, where the top left corner is (0, 0), to the passed
 *                     color. Colors are specified with a 24-bit value, where the bottom
 *                     8 bits represent the amount of blue, the following 8 the amount
 *                     of green, and the next 8 the amount of red.
 *
 *  Operation:         The function simply translates the x and y coordinates into a VRAM
 *                     address by setting the top bits to the offset of the VRAM, and ORing
 *                     in the shifted row and column indeces. Then, it stores the passwed
 *                     color value at that address.
 *
 *  Arguments:         x - x coordinate of the pixel, where leftmost column is 0 (r4).
 *                     y - y coordinate of the pixel, where top row is 0 (r5).
 *                     color - 24-bit value with RGB color the pixel should change to (r6).
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            Changes the color of one pixel on the display.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r8, r9, r10.
 *
 *  Revision History:
 *      6/03/14   Santiago Navonne     Initial revision.
 *
 */
    .global plot_pixel
plot_pixel:                    /* draw a pixel of the specified color */
    MOVHI   r8, %hi(VRAM_BASE) /* find pixel location by first going to VRAM base */
    ORI     r8, r8, %lo(VRAM_BASE)
    MOVI    r9, ROW_ADDR_SHIFT /* shift the row to the row part of the address */
    SLL     r9, r5, r9
    MOVI    r10, COL_ADDR_SHIFT/* and the column to the column part */
    SLL     r10, r4, r10
    OR      r8, r8, r9         /* OR row, column, and VRAM base together */
    OR      r8, r8, r10        /*  to create final pixel address */
    STWIO   r6, (r8)           /* and finally save passed color value to that address */

    RET                        /* all done, so return */

/*
 *  pixel_color
 *
 *  Description:       This procedure returns the color of the pixel at the passed x, y
 *                     coordinates, where the top left corner is (0, 0). Colors are
 *                     specified with a 24-bit RGB value, where the bottom 8 bits
 *                     represent the amount of blue, the following 8 the amount of green,
 *                     and the next 8 the amount of red.
 *
 *  Operation:         The function simply translates the x and y coordinates into a VRAM
 *                     address by setting the top bits to the offset of the VRAM, and ORing
 *                     in the shifted row and column indeces. Then, it loads the color word
 *                     from VRAM and returns it in r2.
 *
 *  Arguments:         x - x coordinate of the pixel, where leftmost column is 0 (r4).
 *                     y - y coordinate of the pixel, where top row is 0 (r5).
 *
 *  Return Value:      color - 24-bit value with RGB color of requested pixel, or NO_TRACE
 *                             if no trace was found at the requested coordinate(r2).
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r8, r9, r10, r2.
 *
 *  Revision History:
 *      6/03/14   Santiago Navonne     Initial revision.
 *
 */
    .global pixel_color
pixel_color:                   /* read a pixel from display */
    MOVHI   r8, %hi(VRAM_BASE) /* find pixel location by first going to VRAM base */
    ORI     r8, r8, %lo(VRAM_BASE)
    MOVI    r9, ROW_ADDR_SHIFT /* shift the row to the row part of the address */
    SLL     r9, r5, r9
    MOVI    r10, COL_ADDR_SHIFT/* and the column to the column part */
    SLL     r10, r4, r10
    OR      r8, r8, r9         /* OR row, column, and VRAM base together */
    OR      r8, r8, r10        /*  to create final pixel address */
    LDWIO   r2, (r8)           /* and finally read color value from that address */

    RET                        /* storing it in return register */

```

```
/****************************************************************************/
/*                                                                          */
/*                              DISPLAY.H                                   */
/*                     Display Interface Definitions                        */
/*                             Include File                                 */
/*                     Digital Oscilloscope Project                         */
/*                               EE/CS 52                                   */
/*                           Santiago Navonne                               */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants for the display interface routines. The
   file includes hardware constants related to the memory layout of the display
   are in the VRAM.


   Revision History:
      6/3/14  Santiago Navonne  Initial revision.
*/

/* VRAM-related constants */
#define    ROW_SIZE     512
#define    REMAINDER    (ROW_SIZE-SIZE_X)*WORD_SIZE
#define    ROW_ADDR_SHIFT 11
#define    COL_ADDR_SHIFT 2

```

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

/*
   This file contains general constants for the assembly functions within the
   EE/CS 52 Digital Oscilloscope project.


   Revision History:
      5/30/14  Santiago Navonne  Initial revision.
*/

/* General constants */
#define    FALSE         0           /* Zero is false */
#define    TRUE          1           /* Non-zero is true */
#define    WORD_SIZE     4           /* A word is 4 bytes */
#define    NEG_WORD_SIZE -4          /* Include negative to facilitate subtraction */

/* PIO register constants */
#define    EDGE_CAP_OF   3*WORD_SIZE /* Offset of edge capture PIO register */
#define    INTMASK_OF    2*WORD_SIZE /* Offset of interrupt mask PIO register */
#define    ENABLE_ALL    0b00111111  /* Enable interrupts from all six sources */

```

```
/*****************************************************************************/
/*                                                                           */
/*                                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.
      5/14/14  Santiago Navonne  Changed keypad codes.
      6/01/14  Santiago Navonne  Changed scope and sampling parameters.
      6/03/14  Santiago Navonne  Changed and added display parameters.
*/



#ifndef  __INTERFAC_H__
    #define  __INTERFAC_H__


/* library include files */
  /* none */

/* local include files */
  /* none */




/* constants */

/* keypad constants */
#define  KEY_MENU       1   /* <Menu>       */
#define  KEY_UP         2   /* <Up>         */
#define  KEY_DOWN       3   /* <Down>       */
#define  KEY_LEFT       4   /* <Left>       */
#define  KEY_RIGHT      5   /* <Right>      */
#define  KEY_ILLEGAL    6   /* illegal key */

/* display constants */
#define  SIZE_X         480   /* size in the x dimension */
#define  SIZE_Y           272   /* size in the y dimension */
#define  PIXEL_CLEAR    0x00000000 /* pixel off is black */
#define  PIXEL_LINE     0x001B3830 /* lines are gray */
#define  PIXEL_TEXT_H   0x00FFFFFF /* highlighted text is white */
#define  PIXEL_TRACE    0x0000A000 /* trace is green */
#define  PIXEL_TEXT_N   0x001B3830 /* normal text is gray */
#define  PIXEL_CURSOR   0x00A00000 /* cursor is red */
#define  NO_TRACE       0xFFFFFFFF /* no trace found */

/* scope parameters */
#define  MIN_DELAY      0         /* minimum trigger delay */
#define  MAX_DELAY      0xFFFFFFFE/* maximum trigger delay */
#define  MIN_LEVEL      -12000    /* minimum trigger level (in mV) */
#define  MAX_LEVEL      12000     /* maximum trigger level (in mV) */

/* sampling parameters */
#define  MAX_SAMPLE_SIZE   512    /* maximum size of a sample (in samples) */


#endif
```

```
/****************************************************************************/
/*                                                                          */
/*                                 KEYPROC                                  */
/*                         Key Processing Functions                         */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the key processing functions for the Digital
   Oscilloscope project.  These functions are called by the main loop of the
   system.  The functions included are:
      menu_down  - process the <Down> key while in a menu
      menu_key   - process the <Menu> key
      menu_left  - process the <Left> key while in a menu
      menu_right - process the <Right> key while in a menu
      menu_up    - process the <Up> key while in a menu
      no_action  - nothing to do

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
*/



/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"
#include  "keyproc.h"
#include  "menu.h"




/*
   no_action

   Description:      This function handles a key when there is nothing to be
                     done.  It just returns.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  no_action(enum status cur_state)
{
    /* variables */
      /* none */



    /* return the current state */
    return  cur_state;

}




/*
   menu_key

   Description:      This function handles the <Menu> key.  If the passed
                     state is MENU_ON, the menu is turned off.  If the passed
             state is MENU_OFF, the menu is turned on.  The returned
             state is the "opposite" of the passed state.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state ("opposite" of the
             as current state).

   Input:            None.
   Output:           The menu is either turned on or off.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_key(enum status cur_state)
{
    /* variables */
      /* none */



    /* check if need to turn the menu on or off */
    if (cur_state == MENU_ON)
        /* currently the menu is on, turn it off */
    clear_menu();
    else
        /* currently the menu is off, turn it on */
    display_menu();


    /* all done, return the "opposite" of the current state */
    if (cur_state == MENU_ON)
        /* state was MENU_ON, change it to MENU_OFF */
        return  MENU_OFF;
    else
        /* state was MENU_OFF, change it to MENU_ON */
        return  MENU_ON;

}




/*
   menu_up

   Description:      This function handles the <Up> key when in a menu.  It
                     goes to the previous menu entry and leaves the system
             state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display is updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_up(enum status cur_state)
{
    /* variables */
      /* none */



    /* go to the previous menu entry */
    previous_entry();


    /* return the current state */
    return  cur_state;

}




/*
   menu_down

   Description:      This function handles the <Down> key when in a menu.  It
                     goes to the next menu entry and leaves the system state
             unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display is updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_down(enum status cur_state)
{
    /* variables */
      /* none */



    /* go to the next menu entry */
    next_entry();


    /* return the current state */
    return  cur_state;
```

```

}




/*
   menu_left

   Description:      This function handles the <Left> key when in a menu.  It
                     invokes the left function for the current menu entry and
             leaves the system state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display may be updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

enum status  menu_left(enum status cur_state)
{
    /* variables */
      /* none */



    /* invoke the <Left> key function for the current menu entry */
    menu_entry_left();


    /* return the current state */
    return  cur_state;

}




/*
   menu_right

   Description:      This function handles the <Right> key when in a menu.  It
                     invokes the right function for the current menu entry and
             leaves the system state unchanged.

   Arguments:        cur_state (enum status) - the current system state.
   Return Value:     (enum status) - the new system state (same as current
             state).

   Input:            None.
   Output:           The menu display may be updated.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

```

```
*/

enum status  menu_right(enum status cur_state)
{
    /* variables */
      /* none */



    /* invoke the <Right> key function for the current menu entry */
    menu_entry_right();


    /* return the current state */
    return  cur_state;

}

```

```
/****************************************************************************/
/*                                                                          */
/*                                KEYPROC.H                                 */
/*                         Key Processing Functions                         */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants and function prototypes for the key
   processing functions (defined in keyproc.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
*/



#ifndef  __KEYPROC_H__
    #define  __KEYPROC_H__


/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"




/* constants */
    /* none */




/* structures, unions, and typedefs */
    /* none */




/* function declarations */

enum status  no_action(enum status);      /* nothing to do */

enum status  menu_key(enum status);   /* process the <Menu> key */

enum status  menu_up(enum status);    /* <Up> key in a menu */
enum status  menu_down(enum status);      /* <Down> key in a menu */
enum status  menu_left(enum status);      /* <Left> key in a menu */
enum status  menu_right(enum status);     /* <Right> key in a menu */


#endif
```

```
/****************************************************************************/
/*                                                                          */
/*                                  KEYS.S                                  */
/*                               Key handlers                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                             Santiago Navonne                             */
/*                                                                          */
/****************************************************************************/

/*
   Key and rotary encoder control routines for the EE/CS 52 Digital Oscilloscope
   project. Function definitions are included in this file, and are laid out
   as follows:
    - keys_init: Initializes the key handler's shared variables, and enables
                 interrupts from the required sources, effectively preparing
                 the user input section for use;
    - keys_handler: Handles key press (and rotary encoder turn) interrupts;
    - getkey: Returns the currently pending user action, blocking if none is
              available.
    - key_available: Checks whether a user action is currently pending.


   Revision History:
      5/7/14  Santiago Navonne  Initial revision.
      5/14/14 Santiago Navonne  Added additional documentation.
      6/7/14  Santiago Navonne  Changed up/down rotation direction.
*/

/* Includes */
#include "general.h"  /* General constants */
#include "system.h"   /* Base addresses */
#include "interfac.h" /* Software interface definitions */
#include "keys.h"     /* Local constants */


/* Variables */
    .section .data  /* No alignment necessary: variables are bytes */
curr_key: .byte 0   /* Current pending key; 0 if no key available */

    .section .text  /* Code starts here */

/*
 *  keys_init
 *
 *  Description:       This procedure initializes the internal state of the key/
 *                     user input handling system, preparing any shared variables
 *                     for use and configuring interrupts. This function should be
 *                     called in order to start accepting user input.
 *
 *  Operation:         This procedure initializes any shared variables to their
 *                     default states:
 *                      - curr_key: value of the currently pending key (default: 0).
 *                     Additionally, the function registers the key press handler
 *                     as the default interrupt handler for key presses using the HAL
 *                     API alt_ic_isr_register, and finally unmasks all interrupts by
 *                     writing to the corresponding PIO register.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  - curr_key (write only).
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r4, r5, r6, r7, r8, r9.
 *
 *  Revision History:
 *      5/7/14    Santiago Navonne     Initial revision.
 *      5/14/14   Santiago Navonne     Added additional documentation.
 *
 */
    .global keys_init
keys_init:
    ADDI    sp, sp, NEG_WORD_SIZE  /* push return address */
    STW     ra, (sp)

    MOVIA   r9, curr_key           /* no key (r0) available at start */
    STB     r0, (r9)               /* so store it into variable curr_key */

    MOVHI   r8, %hi(PIO_0_BASE)    /* write to the PIO registers */
    ORI     r8, r8, %lo(PIO_0_BASE)
    MOVI    r9, ENABLE_ALL         /*  the ENABLE_ALL value */
    STBIO   r9, EDGE_CAP_OF(r8)    /* sending general EOI to clear ints */

    MOV     r4, r0                 /* argument ic_id is ignored */
    MOVI    r5, PIO_0_IRQ          /* second arg is IRQ num */
    MOVIA   r6, keys_handler       /* third arg is int handler */
    MOV     r7, r0                 /* fourth arg is data struct (null) */
    ADDI    sp, sp, NEG_WORD_SIZE  /* fifth arg goes on stack */
    STW     r0, (sp)               /*  and is ignored (so 0) */
    CALL    alt_ic_isr_register    /* finally, call setup function */
    ADDI    sp, sp, WORD_SIZE      /* clean up stack after call */

    LDW     ra, (sp)               /* pop return address */
    ADDI    sp, sp, WORD_SIZE

    STBIO   r9, INTMASK_OF(r8)     /* enable (unmask) interrupts */

    RET                            /* and finally return */



/*
 *  keys_handler
 *
 *  Description:       This procedure handles hardware interrupts generated by
 *                     key presses and rotary encoder steps. Every time one of
 *                     these fires, the shared variable containing the currently
 *                     pending key is updated to indicate a key press. Note that
 *                     previously pending key presses are overwritten by this
 *                     function.
 *                     The function is designed to support only one key press
 *                     at a time; its behavior in the event of simultaneous key
 *                     presses is undefined.
 *
 *  Operation:         When called, the function first reads the edge capture
 *                     register of the user input PIO interface to figure out
 *                     which interrupt fired. It compares the read value to all
 *                     the known constants, translating it into a key ID. Unknown
 *                     values, which are caused by simultaneous key presses,
 *                     are handled in the else case.
 *                     After the key press is decoded, the identification code is
 *                     saved to the shared variable curr_key.
 *                     Note that the procedure uses multiple comparisons and not
 *                     a jump table in order to save space; furthermore, the
 *                     interrupt register value is not simply used as a key
 *                     identifier to prevent simultaneous key presses from
 *                     breaking the system.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  - curr_key: currently pending key press code (read/write).
 *
 *  Global Variables:  None.
 *
 *  Input:             Key presses and rotary encoder turns from the user interface.
 *
 *  Output:            None.
 *
 *  Error Handling:    If multiple keys are pressed at once, the function's
 *                     behavior is undefined.
 *
 *  Limitations:       Only one simultaneous key press is accepted.
 *                     Any previously recognized but not yet polled key presses
 *                     are lost (overwritten) when a new event is received.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: et.
 *
 *  Revision History:
 *      5/7/14    Santiago Navonne     Initial revision.
 *      5/14/14   Santiago Navonne     Added additional documentation.
 *
 */
    .global keys_handler
keys_handler:
    ADDI    sp, sp, NEG_WORD_SIZE   /* save r8 */
    STW     r8, (sp)

    MOVHI   et, %hi(PIO_0_BASE)  /* fetch PIO edge capture register */
    ORI     et, et, %lo(PIO_0_BASE)
    LDBIO   r8, EDGE_CAP_OF(et)

    STBIO   r8, EDGE_CAP_OF(et)  /* and write back to send EOI */
                                 /* figure out what interrupt fired */
    MOVI    et, PUSH1_MASK       /* check if it was pushbutton 1 */
    BEQ     r8, et, keys_handler_push1
    MOVI    et, PUSH2_MASK       /* check if it was pushbutton 2 */
    BEQ     r8, et, keys_handler_push2
    MOVI    et, ROT1R_MASK       /* check if it was rotary enc 1 right */
    BEQ     r8, et, keys_handler_rot1r
    MOVI    et, ROT1L_MASK       /* check if it was rotary enc 1 left */
    BEQ     r8, et, keys_handler_rot1l
    MOVI    et, ROT2R_MASK       /* check if it was rotary enc 2 right */
    BEQ     r8, et, keys_handler_rot2r
    JMPI    keys_handler_rot2l    /* else it must be rotary enc 2 left */

keys_handler_push1:               /* handle pushbutton 1 ints */
    MOVI    et, KEY_MENU          /*  translates into menu key */
    JMPI    keys_handler_done

keys_handler_push2:               /* handle pushbutton 2 ints */
    MOVI    et, KEY_MENU          /*  translates into menu key */
    JMPI    keys_handler_done

keys_handler_rot1r:               /* handle rotary enc 1 right ints */
    MOVI    et, KEY_DOWN          /*  translates into down key */
    JMPI    keys_handler_done

keys_handler_rot1l:               /* handle rotary enc 1 left ints */
    MOVI    et, KEY_UP            /*  translates into up key */
    JMPI    keys_handler_done

keys_handler_rot2r:               /* handle rotary enc 2 right ints */
    MOVI    et, KEY_RIGHT         /*  translates into right key */
    JMPI    keys_handler_done

keys_handler_rot2l:               /* handle rotary enc 2 left ints */
    MOVI    et, KEY_LEFT           /*  translates into left key */
    JMPI    keys_handler_done

keys_handler_done:                /* handling completed */
    MOVIA   r8, curr_key          /* save to curr_key */
    STB     et, (r8)              /*  the processed key */

    LDW     r8, (sp)              /* restore r8 */
    ADDI    sp, sp, WORD_SIZE
    RET                           /* all done */



/*
 *  getkey
 *
 *  Description:       This procedure returns the identifier of the last pressed,
 *                     unpolled key, as described in interfac.h.
 *                     If no key press is pending, the function blocks.
 *                     (To ensure non-blocking behavior, getkey calls should be
 *                     preceded by key_available calls.)
 *
 *  Operation:         The function first fetches the value stored in curr_key and
 *                     compares it to 0, which would indicate that there isn't
 *                     actually any pending key press. In no key press is pending,
 *                     the function keeps fetching the value until it is not 0.
 *                     When the value is not 0, the function clears the value of
 *                     curr_key (to delete the now reported press) and returns
 *                     the retrieved value.
 *
 *  Arguments:         None.
 *
 *  Return Value:      key (r2) - ID code of the pending key, as defined in
 *                                interfac.h.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  - curr_key: currently pending key press code (read/write).
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    If no key is available, the funciton blocks until a key
 *                     is pressed.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r2, r8.
 *
 *  Revision History:
 *      5/7/14    Santiago Navonne     Initial revision.
 *      5/14/14   Santiago Navonne     Added additional documentation.
 *
 */
    .global getkey
getkey:
    MOVIA   r8, curr_key      /* return current pending key */
    LDB     r2, (r8)
    BEQ     r0, r2, getkey    /* if there is no key (curr_key == r0), block */

    STB     r0, (r8)          /* clear current key */
    RET                       /* return with current pending key in r2 */



/*
 *  key_available
 *
 *  Description:       This procedure checks whether a key has been pressed and
 *                     is available for polling. The function returns true
 *                     (non-zero) if there's a key available, and non-zero if no
 *                     key has been pressed.
 *                     This function should be called before using getkey to avoid
 *                     blocking.
 *
 *  Operation:          The function simply returns the value stored in the shared
 *                      variable curr_key, taking advantage of the fact that this
 *                      value is zero if no key is available, and non-zero otherwise.
 *
 *  Arguments:          None.
 *
 *  Return Value:       key_available (r2) - true (non-zero) if a key press is
 *                                           available, false (zero) otherwise.
 *
 *  Local Variables:    None.
 *
 *  Shared Variables:   - curr_key: currently pending key press code (read only).
 *
 *  Global Variables:   None.
 *
 *  Input:              Key presses and rotary encoder turns from the user interface.
 *
 *  Output:             None.
 *
 *  Error Handling:     None.
 *
 *  Limitations:        None.
 *
 *  Algorithms:         None.
 *  Data Structures:    None.
 *
 *  Registers Changed: r2, r8.
 *
 *  Revision History:
 *      5/7/14    Santiago Navonne     Initial revision.
 *      5/14/14   Santiago Navonne     Added additional documentation.
 *
 */
    .globl key_available
key_available:
    MOVIA   r8, curr_key       /* return current pending key */
    LDB     r2, (r8)           /* will be zero (FALSE) if no key is pending */

    RET                        /* return with boolean in r2 */
```

```
/****************************************************************************/
/*                                                                          */
/*                                 KEYS.H                                   */
/*                         Key Handlers Definitions                         */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                EE/CS 52                                  */
/*                             Santiago Navonne                             */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants for the key press and rotary encoder
   handler routines. The file includes interrupt masks used to determine the
   source of interrupts; offsets of the PIO registers.


   Revision History:
      5/7/14  Santiago Navonne  Initial revision.
      5/14/14 Santiago Navonne  Added additional documentation.
*/

/* Interrupt masks */
#define    PUSH1_MASK    0b00100000  /* Pushbutton 1 mask */
#define    PUSH2_MASK    0b00010000  /* Pushbutton 2 mask */
#define    ROT1R_MASK    0b00000100  /* Rotary encoder 1, right mask */
#define    ROT1L_MASK    0b00001000  /* Rotary encoder 1, left mask */
#define    ROT2R_MASK    0b00000001  /* Rotary encoder 2, right mask */
#define    ROT2L_MASK    0b00000010  /* Rotary encoder 2, left mask */

```

```
/****************************************************************************/
/*                                                                          */
/*                                  LCDOUT                                  */
/*                           LCD Output Functions                           */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the functions for doing output to the LCD screen for the
   Digital Oscilloscope project.  The functions included are:
      clear_region - clear a region of the display
      plot_char    - output a character
      plot_hline   - draw a horizontal line
      plot_string  - output a string
      plot_vline   - draw a vertical line
      plot_cursor  - plot the cursor

   The local functions included are:
      none

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Simplified code in plot_string function.
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Change plot_char() and plot_string() to use
                     enum char_style instead of an int value.
      5/27/98  Glen George       Change plot_char() to explicitly declare the
                     size of the external array to avoid linker
                     errors.
      6/3/14   Santiago Navonne  Changed UI display colors, added support for
                     highlighted characters.
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"


extern int pixel_color(int, int);



/*
   clear_region

   Description:      This function clears the passed region of the display.
                     The region is described by its upper left corner pixel
                     coordinate and the size (in pixels) in each dimension.

   Arguments:        x_ul (int)   - x coordinate of upper left corner of the
                    region to be cleared.
             y_ul (int)   - y coordinate of upper left corner of the
                    region to be cleared.
             x_size (int) - horizontal size of the region.
             y_size (int) - vertical size of the region.
   Return Value:     None.

   Input:            None.
   Output:           A portion of the screen is cleared (set to PIXEL_CLEAR).

   Error Handling:   No error checking is done on the coordinates.

   Algorithms:          None.
   Data Structures:     None.

   Global Variables: None.

   Author:              Glen George
   Last Modified:       June 03, 2014

*/

void  clear_region(int x_ul, int y_ul, int x_size, int y_size)
{
    /* variables */
    int  x;     /* x coordinate to clear */
    int  y;     /* y coordinate to clear */



    /* loop, clearing the display */
    for (x = x_ul; x < (x_ul + x_size); x++)  {
        for (y = y_ul; y < (y_ul + y_size); y++)  {

        /* clear this pixel */
        plot_pixel(x, y, PIXEL_CLEAR);
        }
    }


    /* done clearing the display region - return */
    return;

}




/*
   plot_hline

   Description:         This function draws a horizontal line from the passed
                        position for the passed length.  The line is always drawn
                        with the color PIXEL_LINE.  The position (0,0) is the
             upper left corner of the screen.

   Arguments:           start_x (int) - starting x coordinate of the line.
             start_y (int) - starting y coordinate of the line.
             length (int)  - length of the line (positive for a line
                            to the "right" and negative for a line to
                        the "left").
   Return Value:        None.

   Input:               None.
   Output:              A horizontal line is drawn at the specified position.

   Error Handling:      No error checking is done on the coordinates.

   Algorithms:          None.
   Data Structures:     None.

   Global Variables: None.

   Author:              Glen George
   Last Modified:       June 03, 2014

*/

void  plot_hline(int start_x, int start_y, int length)
{
    /* variables */
    int  x;     /* x position while plotting */

    int  init_x;    /* starting x position to plot */
    int  end_x;     /* ending x position to plot */

    /* check if a line to the "right" or "left" */
    if (length > 0)  {

        /* line to the "right" - start at start_x, end at start_x + length */
    init_x = start_x;
    end_x = start_x + length;
    }
    else  {

        /* line to the "left" - start at start_x + length, end at start_x */
    init_x = start_x + length;
    end_x = start_x;
    }


    /* loop, outputting points for the line (always draw to the "right") */
    for (x = init_x; x < end_x; x++)
        /* plot a point of the line */
    plot_pixel(x, start_y, PIXEL_LINE);


    /* done plotting the line - return */
    return;

}




/*
   plot_vline

   Description:      This function draws a vertical line from the passed
                     position for the passed length.  The line is always drawn
                     with the color PIXEL_LINE.  The position (0,0) is the
             upper left corner of the screen.

   Arguments:        start_x (int) - starting x coordinate of the line.
             start_y (int) - starting y coordinate of the line.
             length (int)  - length of the line (positive for a line
                         going "down" and negative for a line
                     going "up").
   Return Value:     None.

   Input:            None.
   Output:           A vertical line is drawn at the specified position.

   Error Handling:   No error checking is done on the coordinates.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    June 03, 2014

*/

void  plot_vline(int start_x, int start_y, int length)
{
    /* variables */
    int  y;     /* y position while plotting */

    int  init_y;    /* starting y position to plot */
    int  end_y;     /* ending y position to plot */



    /* check if an "up" or "down" line */
    if (length > 0)  {

        /* line going "down" - start at start_y, end at start_y + length */
    init_y = start_y;
    end_y = start_y + length;
    }
    else  {

        /* line going "up" - start at start_y + length, end at start_y */
    init_y = start_y + length;
    end_y = start_y;
    }


    /* loop, outputting points for the line (always draw "down") */
    for (y = init_y; y < end_y; y++)
        /* plot a point of the line */
    plot_pixel(start_x, y, PIXEL_LINE);


    /* done plotting the line - return */
    return;

}




/*
   plot_char

   Description:      This function outputs the passed character to the LCD
                     screen at passed location.  The passed location is given
                     as a character position with (0,0) being the upper left
             corner of the screen.  The character can be drawn in
             "normal video" (gray on black), "reverse video" (black
             on gray), or highlighted (white on black).

   Arguments:        pos_x (int)             - x coordinate (in character
                               cells) of the character.
             pos_y (int)             - y coordinate (in character
                               cells) of the character.
             c (char)                  - the character to plot.
             style (enum char_style) - style with which to plot the
                                   character (NORMAL or REVERSE).
   Return Value:     None.

   Input:            None.
   Output:           A character is output to the LCD screen.

   Error Handling:   No error checking is done on the coordinates or the
             character (to ensure there is a bit pattern for it).

   Algorithms:       None.
   Data Structures:  The character bit patterns are stored in an external
             array.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    June 03, 2014

*/

void  plot_char(int pos_x, int pos_y, char c, enum char_style style)
{
    /* variables */

    /* pointer to array of character bit patterns */
    extern const unsigned char   char_patterns[(VERT_SIZE - 1) * 128];

    int  bits;          /* a character bit pattern */

    int  col;       /* column loop index */
    int  row;           /* character row loop index */

    int  x;     /* x pixel position for the character */
    int  y;     /* y pixel position for the character */

    int color = PIXEL_TEXT_N; /* pixel drawing color */



    /* setup the pixel positions for the character */
    x = pos_x * HORIZ_SIZE;
    y = pos_y * VERT_SIZE;


    /* loop outputting the bits to the screen */
    for (row = 0; row < VERT_SIZE; row++)  {

        /* get the character bits for this row from the character table */
    if (row == (VERT_SIZE - 1))
        /* last row - blank it */
        bits = 0;
    else
        /* in middle of character, get the row from the bit patterns */
            bits = char_patterns[(c * (VERT_SIZE - 1)) + row];

    /* take care of "normal/reverse video" */
    if (style == REVERSE)
        /* invert the bits for "reverse video" */
        bits = ~bits;
  if (style == HIGHLIGHTED)
      color = PIXEL_TEXT_H;

        /* get the bits "in position" (high bit is output first */
    bits <<= (8 - HORIZ_SIZE);


    /* now output the row of the character, pixel by pixel */
    for (col = 0; col < HORIZ_SIZE; col++)  {

            /* output this pixel in the appropriate color */
        if ((bits & 0x80) == 0)
            /* blank pixel - output in PIXEL_CLEAR */
        plot_pixel(x + col, y, PIXEL_CLEAR);
        else
            /* black pixel - output in PIXEL_TEXT */
        plot_pixel(x + col, y, color);

        /* shift the next bit into position */
        bits <<= 1;
        }


    /* next row - update the y position */
    y++;
    }


    /* all done, return */
    return;

}




/*
   plot_string

   Description:      This function outputs the passed string to the LCD screen
                     at passed location.  The passed location is given as a
                     character position with (0,0) being the upper left corner
             of the screen.  There is no line wrapping, so the entire
             string must fit on the passed line (pos_y).  The string
             can be drawn in "normal video" (black on white) or
             "reverse video" (white on black).

   Arguments:        pos_x (int)             - x coordinate (in character
                               cells) of the start of the
                           string.
             pos_y (int)             - y coordinate (in character
                               cells) of the start of the
                           string.
             s (const char *)        - the string to output.
             style (enum char style) - style with which to plot
                                   characters of the string.
   Return Value:     None.

   Input:            None.
   Output:           A string is output to the LCD screen.

   Error Handling:   No checking is done to insure the string is fully on the
             screen (the x and y coordinates and length of the string
             are not checked).

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

void  plot_string(int pos_x, int pos_y, const char *s, enum char_style style)
{
    /* variables */
      /* none */



    /* loop, outputting characters from string s */
    while (*s != '\0')

        /* output this character and move to the next character and screen position */
    plot_char(pos_x++, pos_y, *s++, style);


    /* all done, return */
    return;

}
```

```
/****************************************************************************/
/*                                                                          */
/*                                 LCDOUT.H                                 */
/*                           LCD Output Functions                           */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                EE/CS  52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants and function prototypes for the LCD output
   functions used in the Digital Oscilloscope project and defined in lcdout.c.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Added enumerated type char_style and updated
                                    function prototypes.
      6/3/14   Santiago Navonne  Added highlighted character style.
*/



#ifndef  __LCDOUT_H__
    #define  __LCDOUT_H__


/* library include files */
  /* none */

/* local include files */
  /* none */




/* constants */

/* character output styles */

/* size of a character (includes 1 pixel space to the left and below character) */
#define  VERT_SIZE    8         /* vertical size (in pixels -> 7+1) */
#define  HORIZ_SIZE   6         /* horizontal size (in pixels -> 5+1) */




/* structures, unions, and typedefs */

/* character output styles */
enum  char_style  {  NORMAL,     /* "normal video" */
                     REVERSE,      /* "reverse video" */
                     HIGHLIGHTED /* highlighted text */
              };




/* function declarations */

void  clear_region(int, int, int, int);       /* clear part of the display */

void  plot_hline(int, int, int);          /* draw a horizontal line */
void  plot_vline(int, int, int);          /* draw a vertical line */

void  plot_char(int, int, char, enum char_style); /* output a character */
void  plot_string(int, int, const char *, enum char_style);  /* output a string */

int   plot_cursor(int, int);          /* draws the cursor on the trace */


#endif
```

```
/****************************************************************************/
/*                                                                          */
/*                                 MAINLOOP                                 */
/*                             Main Program Loop                            */
/*                        Digital Oscilloscope Project                      */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the main processing loop (background) for the Digital
   Oscilloscope project.  The only global function included is:
      main - background processing loop

   The local functions included are:
      key_lookup - get a key and look up its keycode

   The locally global variable definitions included are:
      none


   Revision History
      3/8/94   Glen George       Initial revision.
      3/9/94   Glen George       Changed initialized const arrays to static
                 (in addition to const).
      3/9/94   Glen George       Moved the position of the const keyword in
                 declarations of arrays of pointers.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Removed display_menu call after plot_trace,
                 the plot function takes care of the menu.
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Made key_lookup function static to make it
                 truly local.
      3/17/97  Glen George       Removed KEY_UNUSED and KEYCODE_UNUSED
                 references (no longer used).
      5/27/08  Glen George       Changed code to only check for sample done if
                 it is currently sampling.
      6/03/14  Santiago Navonne  Added initialization code.
      6/11/14  Santiago Navonne  Added sleep time between draws.
*/



/* library include files */
#include  "unistd.h"

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "keyproc.h"
#include  "menu.h"
#include  "tracutil.h"




/* local function declarations */
static enum keycode  key_lookup(void);      /* translate key values into keycodes */




/*
   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 handles
                     starting scope sample collection and updating the LCD
                     screen. Additionally, it initializes the triggering logic
                     and key interface.

   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:    June 11, 2014

*/

int  main()
{
    /* initialize keys, triggering */
      keys_init();
      trigger_init();

    /* variables */
    enum keycode        key;            /* an input key */

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

    unsigned char *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 */


    /* 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);

        /* sleep for some time to reduce blinking of display */
        /*usleep(DRAW_INTERVAL);

        /* 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;

}




/*
   key_lookup

   Description:      This function gets a key from the keypad and translates
                     the raw keycode to an enumerated keycode for the main
                     loop.

   Arguments:        None.
   Return Value:     (enum keycode) - type of the key input on keypad.

   Input:            Keys from the keypad.
   Output:           None.

   Error Handling:   Invalid keys are returned as KEYCODE_ILLEGAL.

   Algorithms:       The function uses an array to lookup the key types.
   Data Structures:  Array of key types versus key codes.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

static  enum keycode  key_lookup()
{
    /* variables */

    const static enum keycode  keycodes[] = /* array of keycodes */
        {                           /* order must match keys array exactly */
           KEYCODE_MENU,      /* <Menu>     */   /* also need an extra element */
       KEYCODE_UP,        /* <Up>       */   /* for unknown key codes */
       KEYCODE_DOWN,      /* <Down>     */
       KEYCODE_LEFT,      /* <Left>     */
       KEYCODE_RIGHT,     /* <Right>    */
       KEYCODE_ILLEGAL    /* other keys */
        };

    const static int  keys[] =   /* array of key values */
        {            /* order must match keycodes array exactly */
           KEY_MENU,    /* <Menu>     */
       KEY_UP,      /* <Up>       */
       KEY_DOWN,    /* <Down>     */
       KEY_LEFT,    /* <Left>     */
       KEY_RIGHT,   /* <Right>    */
        };

    int  key;       /* an input key */

    int  i;             /* general loop index */



    /* get a key */
    key = getkey();


    /* lookup key in keys array */
    for (i = 0; ((i < (sizeof(keys)/sizeof(int))) && (key != keys[i])); i++);


    /* return the appropriate key type */
    return  keycodes[i];

}

```

```c
/****************************************************************************/
/*                                                                          */
/*                                   MENU                                   */
/*                              Menu Functions                              */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the functions for processing menu entries for the
   Digital Oscilloscope project.  These functions take care of maintaining the
   menus and handling menu updates for the system.  The functions included
   are:
      clear_menu       - remove the menu from the display
      display_menu     - display the menu
      init_menu        - initialize menus
      menu_entry_left  - take care of <Left> key for a menu entry
      menu_entry_right - take care of <Right> key for a menu entry
      next_entry       - next menu entry
      previous_entry   - previous menu entry
      refresh_menu     - re-display the menu if currently being displayed
      reset_menu       - reset the current selection to the top of the menu

   The local functions included are:
      display_entry    - display a menu entry (including option setting)

   The locally global variable definitions included are:
      menu             - the menu
      menu_display     - whether or not the menu is currently displayed
      menu_entry       - the currently selected menu entry


   Revision History
      3/8/94   Glen George       Initial revision.
      3/9/94   Glen George       Changed position of const keyword in array
                 declarations involving pointers.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Added display_entry function to output a menu
                 entry and option setting to the LCD (affects
                 many functions).
      3/13/94  Glen George       Changed calls to set_status due to changing
                     enum scale_status definition.
      3/13/94  Glen George       No longer clear the menu area before
                 restoring the trace in clear_menu() (not
                 needed).
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Fixed minor bug in reset_menu().
      3/17/97  Glen George       When initializing the menu in init_menu(),
                 set the delay to MIN_DELAY instead of 0 and
                 trigger to a middle value instead of
                 MIN_TRG_LEVEL_SET.
      5/3/06   Glen George       Changed to a more appropriate constant in
                                 display_entry().
      5/3/06   Glen George       Updated comments.
      5/9/06   Glen George       Changed menus to handle a list for mode and
                             scale (move up and down list), instead of
                     toggling values.
*/



/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menu.h"
#include  "menuact.h"
#include  "tracutil.h"
```

```
/* local function declarations */
static void  display_entry(int, int);      /* display a menu entry and its setting */




/* locally global variables */
static int  menu_display;           /* TRUE if menu is currently displayed */

const static struct menu_item  menu[] =     /* the menu */
    { { "Mode",    0, 4, display_mode      },
      { "Scale",   0, 5, display_scale     },
      { "Sweep",   0, 5, display_sweep     },
      { "Trigger", 0, 7, no_display        },
      { "Level",   2, 7, display_trg_level },
      { "Slope",   2, 7, display_trg_slope },
      { "Delay",   2, 7, display_trg_delay },
    };

static int  menu_entry;         /* currently selected menu entry */




/*
   init_menu

   Description:      This function initializes the menu routines.  It sets
                     the current menu entry to the first entry, indicates the
             display is off, and initializes the options (and
             hardware) to normal trigger mode, scale displayed, the
             fastest sweep rate, a middle trigger level, positive
             trigger slope, and minimum delay.  Finally, it displays
             the menu.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - reset to FALSE.
             menu_entry   - reset to first entry (0).

   Author:           Glen George
   Last Modified:    Mar. 17, 1997

*/

void  init_menu(void)
{
    /* variables */
      /* none */



    /* set the menu parameters */
    menu_entry = 0;     /* first menu entry */
    menu_display = FALSE;   /* menu is not currently displayed (but it will be shortly) */


    /* set the scope (option) parameters */
    set_trigger_mode(NORMAL_TRIGGER);   /* normal triggering */
    set_scale(SCALE_AXES);      /* scale is axes */
    set_sweep(0);           /* first sweep rate */
    set_trg_level((MIN_TRG_LEVEL_SET + MAX_TRG_LEVEL_SET) / 2); /* middle trigger level */
    set_trg_slope(SLOPE_POSITIVE);  /* positive slope */
    set_trg_delay(MIN_DELAY);       /* minimum delay */


    /* now display the menu */
    display_menu();


    /* done initializing, return */
    return;

}




/*
   clear_menu

   Description:      This function removes the menu from the display.  The
                     trace under the menu is restored.  The flag menu_display,
             is cleared, indicating the menu is no longer being
             displayed.  Note: if the menu is not currently being
             displayed this function does nothing.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu if displayed, is removed and the trace under it
             is rewritten.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked and set to FALSE.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  clear_menu(void)
{
    /* variables */
      /* none */



    /* check if the menu is currently being displayed */
    if (menu_display)  {

        /* menu is being displayed - turn it off and restore the trace in that area */
    restore_menu_trace();
    }


    /* no longer displaying the menu */
    menu_display = FALSE;


    /* all done, return */
    return;

}




/*
   display_menu

   Description:      This function displays the menu.  The trace under the
                     menu is overwritten (but it was saved).  The flag
             menu_display, is also set, indicating the menu is
             currently being displayed.  Note: if the menu is already
             being displayed this function does not redisplay it.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - set to TRUE.
             menu_entry   - used to highlight currently selected entry.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  display_menu(void)
{
    /* variables */
    int  i;     /* loop index */



    /* check if the menu is currently being displayed */
    if (!menu_display)  {

        /* menu is not being displayed - turn it on */
    /* display it entry by entry */
    for (i = 0; i < NO_MENU_ENTRIES; i++)  {

        /* display this entry - check if it should be highlighted */
        if (i == menu_entry)
            /* currently selected entry - highlight it */
            display_entry(i, TRUE);
        else
            /* not the currently selected entry - "normal video" */
            display_entry(i, FALSE);
        }
    }


    /* now are displaying the menu */
    menu_display = TRUE;


    /* all done, return */
    return;

}




/*
   refresh_menu

   Description:      This function displays the menu if it is currently being
             displayed.  The trace under the menu is overwritten (but
             it was already saved).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu is displayed.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

    Global Variables: menu_display - determines if menu should be displayed.

    Author:           Glen George
    Last Modified:    Mar. 8, 1994

 */

 void  refresh_menu(void)
 {
     /* variables */
       /* none */



     /* check if the menu is currently being displayed */
     if (menu_display)  {

         /* menu is currently being displayed - need to refresh it */
     /* do this by turning off the display, then forcing it back on */
     menu_display = FALSE;
     display_menu();
     }


     /* refreshed the menu if it was displayed, now return */
     return;

 }




 /*
    reset_menu

    Description:      This function resets the current menu selection to the
                      first menu entry.  If the menu is currently being
              displayed the display is updated.

    Arguments:        None.
    Return Value:     None.

    Input:            None.
    Output:           The menu display is updated if it is being displayed.

    Error Handling:   None.

    Algorithms:       None.
    Data Structures:  None.

    Global Variables: menu_display - checked to see if menu is displayed.
              menu_entry   - reset to 0 (first entry).

    Author:           Glen George
    Last Modified:    Mar. 17, 1997

 */

 void  reset_menu(void)
 {
     /* variables */
       /* none */



     /* check if the menu is currently being displayed */
     if (menu_display)  {

         /* menu is being displayed */
     /* remove highlight from currently selected entry */
     display_entry(menu_entry, FALSE);
     }


     /* reset the currently selected entry */
     menu_entry = 0;


     /* finally, highlight the first entry if the menu is being displayed */
     if (menu_display)
     display_entry(menu_entry, TRUE);



     /* all done, return */
     return;

 }




 /*
    next_entry

    Description:      This function changes the current menu selection to the
                      next menu entry.  If the current selection is the last
              entry in the menu, it is not changed.  If the menu is
              currently being displayed, the display is updated.

    Arguments:        None.
    Return Value:     None.

    Input:            None.
    Output:           The menu display is updated if it is being displayed and
              the entry selected changes.

    Error Handling:   None.

    Algorithms:       None.
    Data Structures:  None.

    Global Variables: menu_display - checked to see if menu is displayed.
              menu_entry   - updated to a new entry (if not at end).

    Author:           Glen George
    Last Modified:    Mar. 13, 1994

 */

 void  next_entry(void)
 {
     /* variables */
       /* none */



     /* only update if not at end of the menu */
     if (menu_entry < (NO_MENU_ENTRIES - 1))  {

         /* not at the end of the menu */

     /* turn off current entry if displaying */
     if (menu_display)
             /* displaying menu - turn off currently selected entry */
         display_entry(menu_entry, FALSE);

     /* update the menu entry to the next one */
     menu_entry++;

     /* now highlight this entry if displaying the menu */
     if (menu_display)
             /* displaying menu - highlight newly selected entry */
         display_entry(menu_entry, TRUE);
     }


     /* all done, return */
     return;

```

```
}




/*
   previous_entry

   Description:      This function changes the current menu selection to the
                     previous menu entry.  If the current selection is the
             first entry in the menu, it is not changed.  If the menu
             is currently being displayed, the display is updated.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the currently selected entry changes.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: menu_display - checked to see if menu is displayed.
             menu_entry   - updated to a new entry (if not at start).

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  previous_entry(void)
{
    /* variables */
      /* none */



    /* only update if not at the start of the menu */
    if (menu_entry > 0)  {

        /* not at the start of the menu */

    /* turn off current entry if displaying */
    if (menu_display)
            /* displaying menu - turn off currently selected entry */
        display_entry(menu_entry, FALSE);

    /* update the menu entry to the previous one */
    menu_entry--;

    /* now highlight this entry if displaying the menu */
    if (menu_display)
            /* displaying menu - highlight newly selected entry */
        display_entry(menu_entry, TRUE);

    }


    /* all done, return */
    return;

}




/*
   menu_entry_left

   Description:      This function handles the <Left> key for the current menu
                     selection.  It does this by doing a table lookup on the
             current menu selection.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the <Left> key causes a change to the display.

   Error Handling:   None.

   Algorithms:       Table lookup is used to determine what to do for the
             input key.
   Data Structures:  An array holds the table of key processing routines.

   Global Variables: menu_entry - used to select the processing function.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  menu_entry_left(void)
{
    /* variables */

    /* key processing functions */
    static void  (* const process[])(void) =
       /*  Mode            Scale             Sweep           Trigger      */
        {  mode_down,      scale_down,       sweep_down,     trace_rearm,
           trg_level_down, trg_slope_toggle, trg_delay_down               };
       /*  Level           Slope             Delay                        */



    /* invoke the appropriate <Left> key function */
    process[menu_entry]();

    /* if displaying menu entries, display the new value */
    /* note: since it is being changed - know this option is selected */
    if (menu_display)  {
        menu[menu_entry].display((MENU_X + menu[menu_entry].opt_off),
                                 (MENU_Y + menu_entry), OPTION_SELECTED);
    }


    /* all done, return */
    return;

}




/*
   menu_entry_right

   Description:      This function handles the <Right> key for the current
                     menu selection.  It does this by doing a table lookup on
             the current menu selection.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The menu display is updated if it is being displayed and
             the <Right> key causes a change to the display.

   Error Handling:   None.

   Algorithms:       Table lookup is used to determine what to do for the
             input key.
   Data Structures:  An array holds the table of key processing routines.

   Global Variables: menu       - used to display the new menu value.
             menu_entry - used to select the processing function.

   Author:            Glen George
   Last Modified:     May 9, 2006

*/

void  menu_entry_right(void)
{
    /* variables */

    /* key processing functions */
    static void  (* const process[])(void) =
       /*  Mode          Scale             Sweep          Trigger      */
        {  mode_up    ,  scale_up,     sweep_up,       trace_rearm,
           trg_level_up, trg_slope_toggle, trg_delay_up                };
       /*  Level         Slope             Delay                       */



    /* invoke the appropriate <Right> key function */
    process[menu_entry]();

    /* if displaying menu entries, display the new value */
    /* note: since it is being changed - know this option is selected */
    if (menu_display)  {
        menu[menu_entry].display((MENU_X + menu[menu_entry].opt_off),
                                 (MENU_Y + menu_entry), OPTION_SELECTED);
    }


    /* all done, return */
    return;

}




/*
   display_entry

   Description:       This function displays the passed menu entry and its
             current option setting.  If the second argument is TRUE
             it displays them with color SELECTED and OPTION_SELECTED
             respectively.  If the second argument is FALSE it
             displays the menu entry with color NORMAL and the option
             setting with color OPTION_NORMAL.

   Arguments:         entry (int)    - menu entry to be displayed.
             selected (int) - whether or not the menu entry is
                          currently selected (determines the color
                      with which the entry is output).
   Return Value:      None.

   Input:             None.
   Output:            The menu entry is output to the LCD.

   Error Handling:    None.

   Algorithms:        None.
   Data Structures:   None.

   Global Variables: menu - used to display the menu entry.

   Author:            Glen George
   Last Modified:     Aug. 13, 2004

*/

static void  display_entry(int entry, int selected)
{
    /* variables */
      /* none */



    /* output the menu entry with the appropriate color */
    plot_string((MENU_X + menu[entry].h_off), (MENU_Y + entry), menu[entry].s,
            (selected ? SELECTED : NORMAL));
    /* also output the menu option with the appropriate color */
    menu[entry].display((MENU_X + menu[entry].opt_off), (MENU_Y + entry),
                (selected ? OPTION_SELECTED : OPTION_NORMAL));


    /* all done outputting this menu entry - return */
    return;

}

```

```
/****************************************************************************/
/*                                                                          */
/*                                  MENU.H                                  */
/*                              Menu Functions                              */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants and function prototypes for the functions
   which deal with menus (defined in menu.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Added definitions for SELECTED,
                     OPTION_NORMAL, and OPTION_SELECTED.
      6/03/14  Santiago Navonne  Changed selected menu and option style to HIGHLIGHTED.
*/



#ifndef  __MENU_H__
    #define  __MENU_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"




/* constants */

/* menu size */
#define  MENU_WIDTH   16        /* menu width (in characters) */
#define  MENU_HEIGHT   7        /* menu height (in characters) */
#define  MENU_SIZE_X  (MENU_WIDTH * HORIZ_SIZE)  /* menu width (in pixels) */
#define  MENU_SIZE_Y  (MENU_HEIGHT * VERT_SIZE)  /* menu height (in pixels) */

/* menu position */
#define  MENU_X    (LCD_WIDTH - MENU_WIDTH - 1)  /* x position (in characters) */
#define  MENU_Y    0                                   /* y position (in characters) */
#define  MENU_UL_X (MENU_X * HORIZ_SIZE)         /* x position (in pixels) */
#define  MENU_UL_Y (MENU_Y * VERT_SIZE)          /* y position (in pixels) */

/* menu colors */
#define  SELECTED         HIGHLIGHTED     /* color for a selected menu entry */
#define  OPTION_SELECTED  HIGHLIGHTED   /* color for a selected menu entry option */
#define  OPTION_NORMAL    NORMAL        /* color for an unselected menu entry option */

/* number of menu entries */
#define  NO_MENU_ENTRIES  (sizeof(menu) / sizeof(struct menu_item))




/* structures, unions, and typedefs */

/* data for an item in a menu */
struct menu_item  {  const char  *s;        /* string for menu entry */
             int          h_off;    /* horizontal offset of entry */
             int          opt_off;  /* horizontal offset of option setting */
             void       (*display)(int, int, int);  /* option display function */
          };

```

```



/* function declarations */

/* menu initialization function */
void  init_menu(void);

/* menu display functions */
void  clear_menu(void);        /* clear the menu display */
void  display_menu(void);      /* display the menu */
void  refresh_menu(void);      /* refresh the menu */

/* menu update functions */
void  reset_menu(void);        /* reset the menu to first entry */
void  next_entry(void);        /* go to the next menu entry */
void  previous_entry(void);   /* go to the previous menu entry */

/* menu entry functions */
void  menu_entry_left(void);       /* do the <Left> key for the menu entry */
void  menu_entry_right(void);      /* do the <Right> key for the menu entry */


#endif

```

```
/****************************************************************************/
/*                                                                          */
/*                                 MENUACT                                  */
/*                          Menu Action Functions                           */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the functions for carrying out menu actions for the
   Digital Oscilloscope project.  These functions are invoked when the <Left>
   or <Right> key is pressed for a menu item.  Also included are the functions
   for displaying the current menu option selection.  The functions included
   are:
      display_mode      - display trigger mode
      display_scale     - display the scale type
      display_sweep     - display the sweep rate
      display_trg_delay - display the tigger delay
      display_trg_level - display the trigger level
      display_trg_slope - display the trigger slope
      get_trigger_mode  - get the current trigger mode
      mode_down         - go to the "next" trigger mode
      mode_up           - go to the "previous" trigger mode
      no_display        - nothing to display for option setting
      no_menu_action    - no action to perform for <Left> or <Right> key
      scale_down        - go to the "next" scale type
      scale_up          - go to the "previous" scale type
      set_scale         - set the scale type
      set_sweep         - set the sweep rate
      set_trg_delay     - set the tigger delay
      set_trg_level     - set the trigger level
      set_trg_slope     - set the trigger slope
      set_trigger_mode  - set the trigger mode
      sweep_down        - decrease the sweep rate
      sweep_up          - increase the sweep rate
      trg_delay_down    - decrease the trigger delay
      trg_delay_up      - increase the trigger delay
      trg_level_down    - decrease the trigger level
      trg_level_up      - increase the trigger level
      trg_slope_toggle  - toggle the trigger slope between "+" and "-"

   The local functions included are:
      adjust_trg_delay  - adjust the trigger delay for a new sweep rate
      cvt_num_field     - converts a numeric field value to a string

   The locally global variable definitions included are:
      delay         - current trigger delay
      level         - current trigger level
      scale         - current display scale type
      slope         - current trigger slope
      sweep         - current sweep rate
      sweep_rates   - table of information on possible sweep rates
      trigger_mode  - current triggering mode


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed all arrays of constant strings to be
                     static so compiler generates correct code.
      3/13/94  Glen George       Changed scale to type enum scale_type and
                     output the selection as "None" or "Axes".
                 This will allow for easier future expansion.
      3/13/94  Glen George       Changed name of set_axes function (in
                     tracutil.c) to set_display_scale.
      3/10/95  Glen George       Changed calculation of displayed trigger
                     level to use constants MIN_TRG_LEVEL_SET and
                 MAX_TRG_LEVEL_SET to get the trigger level
                 range.
      3/17/97  Glen George       Updated comments.
      5/3/06   Glen George       Changed sweep definitions to include new
                     sweep rates of 100 ns, 200 ns, 500 ns, and
                     1 us and updated functions to handle these
                 new rates.
      5/9/06   Glen George       Added new a triggering mode (automatic
                                 triggering) and a new scale (grid) and
                                 updated functions to implement these options.
      5/9/06   Glen George       Added functions for setting the triggering
                                 mode and scale by going up and down the list
                                 of possibilities instead of just toggling
                                 between one of two possibilities (since there
                 are more than two now).
      5/9/06   Glen George       Added accessor function (get_trigger_mode)
                                 to be able to get the current trigger mode.
      6/6/14   Santiago Navonne  Added fastest sweep rate and changed their
                                 values to reflect actual possible rates.
      6/11/14  Santiago Navonne  Modified delay set function to support faster
                                 sweep rates.
*/



/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menuact.h"
#include  "tracutil.h"




/* local function declarations */
static void  adjust_trg_delay(int, int);       /* adjust the trigger delay for new sweep */
static void  cvt_num_field(long int, char *);   /* convert a number to a string */




/* locally global variables

/* trace parameters */
static enum trigger_type   trigger_mode; /* current triggering mode */
static enum scale_type     scale;    /* current scale type */
static int         sweep;        /* sweep rate index */
static int         level;    /* current trigger level */
static enum slope_type     slope;    /* current trigger slope */
static long int            delay;    /* current trigger delay */

/* sweep rate information */
static const struct sweep_info  sweep_rates[] =
    { { 19000000L, " 52 ns " },
      {  9500000L, " 104 ns" },
      {  4750000L, " 208 ns" },
      {  2000000L, " 500 ns" },
      {  1000000L, " 1 \004s  " },
      {   500000L, " 2 \004s  " },
      {   200000L, " 5 \004s  " },
      {   100000L, " 10 \004s " },
      {    50000L, " 20 \004s " },
      {    20000L, " 50 \004s " },
      {    10000L, " 100 \004s" },
      {     5000L, " 200 \004s" },
      {     2000L, " 500 \004s" },
      {     1000L, " 1 ms  "    },
      {      500L, " 2 ms  "    },
      {      200L, " 5 ms  "    },
      {      100L, " 10 ms "    },
      {       50L, " 20 ms "    } };




/*
   no_menu_action

   Description:      This function handles a menu action when there is nothing
                     to be done.  It just returns.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  no_menu_action()
{
    /* variables */
      /* none */



    /* nothing to do - return */
    return;

}




/*
   no_display

   Description:      This function handles displaying a menu option's setting
                     when there is nothing to display.  It just returns,
             ignoring all arguments.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the menu option (not used).
             y_pos (int) - y position (in character cells) at which to
                   display the menu option (not used).
             style (int) - style with which to display the menu option
                       (not used).
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: None.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  no_display(int x_pos, int y_pos, int style)
{
    /* variables */
      /* none */



    /* nothing to do - return */
    return;

}




/*
   set_trigger_mode

   Description:      This function sets the triggering mode to the passed
                     value.

   Arguments:        m (enum trigger_type) - mode to which to set the
                         triggering mode.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trigger_mode(enum trigger_type m)
{
    /* variables */
      /* none */



    /* set the trigger mode */
    trigger_mode = m;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done setting the trigger mode - return */
    return;

}




/*
   get_trigger_mode

   Description:      This function returns the current triggering mode.

   Arguments:        None.
   Return Value:     (enum trigger_type) - current triggering mode.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - value is returned (not changed).

   Author:           Glen George
   Last Modified:    May 9, 2006

*/
```

```

enum trigger_type  get_trigger_mode()
{
    /* variables */
      /* none */



    /* return the current trigger mode */
    return  trigger_mode;

}




/*
   mode_down

   Description:      This function handles moving down the list of trigger
                     modes.  It changes to the "next" triggering mode and
                     sets that as the current mode.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - changed to "next" trigger mode.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  mode_down()
{
    /* variables */
      /* none */



    /* move to the "next" triggering mode */
    if (trigger_mode == NORMAL_TRIGGER)
        trigger_mode = AUTO_TRIGGER;
    else if (trigger_mode == AUTO_TRIGGER)
        trigger_mode = ONESHOT_TRIGGER;
    else
        trigger_mode = NORMAL_TRIGGER;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done with the trigger mode - return */
    return;

}




/*
   mode_up

   Description:      This function handles moving up the list of trigger
                     modes.  It changes to the "previous" triggering mode and
                     sets that as the current mode.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - changed to "previous" trigger mode.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  mode_up()
{
    /* variables */
      /* none */



    /* move to the "previous" triggering mode */
    if (trigger_mode == NORMAL_TRIGGER)
        trigger_mode = ONESHOT_TRIGGER;
    else if (trigger_mode == AUTO_TRIGGER)
        trigger_mode = NORMAL_TRIGGER;
    else
        trigger_mode = AUTO_TRIGGER;

    /* set the new mode */
    set_mode(trigger_mode);


    /* all done with the trigger mode - return */
    return;

}




/*
   display_mode

   Description:      This function displays the current triggering mode at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger mode.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger mode.
             style (int) - style with which to display the trigger
                       mode.
   Return Value:     None.

   Input:            None.
   Output:           The trigger mode is displayed at the passed position on
             the screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trigger_mode - determines which string is displayed.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/
```

```
void  display_mode(int x_pos, int y_pos, int style)
{
    /* variables */

    /* the mode strings (must match enumerated type) */
    const static char * const  modes[] =  {  " Normal    ",
                                             " Automatic",
                                             " One-Shot "  };



    /* display the trigger mode */
    plot_string(x_pos, y_pos, modes[trigger_mode], style);


    /* all done displaying the trigger mode - return */
    return;

}




/*
   set_scale

   Description:      This function sets the scale type to the passed value.

   Arguments:        s (enum scale_type) - scale type to which to initialize
                       the scale status.
   Return Value:     None.

   Input:            None.
   Output:           The new trace display is updated with the new scale.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  set_scale(enum scale_type s)
{
    /* variables */
      /* none */



    /* set the scale type */
    scale = s;

    /* output the scale appropriately */
    set_display_scale(scale);


    /* all done setting the scale type - return */
    return;

}




/*
   scale_down

   Description:      This function handles moving down the list of scale
                     types.  It changes to the "next" type of scale and sets
             this as the current scale type.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The new scale is output to the trace display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - changed to the "next" scale type.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  scale_down()
{
    /* variables */
      /* none */



    /* change to the "next" scale type */
    if (scale == SCALE_NONE)
        scale = SCALE_AXES;
    else if (scale == SCALE_AXES)
        scale = SCALE_GRID;
    else
        scale = SCALE_NONE;

    /* set the scale type */
    set_display_scale(scale);


    /* all done with toggling the scale type - return */
    return;

}




/*
   scale_up

   Description:      This function handles moving up the list of scale types.
                     It changes to the "previous" type of scale and sets this
             as the current scale type.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The new scale is output to the trace display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - changed to the "previous" scale type.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  scale_up()
{
    /* variables */
      /* none */
    /* change to the "previous" scale type */
    if (scale == SCALE_NONE)
        scale = SCALE_GRID;
    else if (scale == SCALE_AXES)
        scale = SCALE_NONE;
    else
        scale = SCALE_AXES;

    /* set the scale type */
    set_display_scale(scale);


    /* all done with toggling the scale type - return */
    return;

}




/*
   display_scale

   Description:      This function displays the current scale type at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the scale type.
             y_pos (int) - y position (in character cells) at which to
                   display the scale type.
             style (int) - style with which to display the scale type.
   Return Value:     None.

   Input:            None.
   Output:           The scale type is displayed at the passed position on the
             display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: scale - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  display_scale(int x_pos, int y_pos, int style)
{
    /* variables */

    /* the scale type strings (must match enumerated type) */
    const static char * const  scale_stat[] =  {  " None",
                                                  " Axes",
                                                  " Grid"  };



    /* display the scale status */
    plot_string(x_pos, y_pos, scale_stat[scale], style);


    /* all done displaying the scale status - return */
    return;

}
```

```
/*
   set_sweep

   Description:      This function sets the sweep rate to the passed value.
                     The passed value gives the sweep rate to choose from the
             list of sweep rates (it gives the list index).

   Arguments:        s (int) - index into the list of sweep rates to which to
                   set the current sweep rate.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed index is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_sweep(int s)
{
    /* variables */
    int  sample_size;       /* sample size for this sweep rate */



    /* set the new sweep rate */
    sweep = s;

    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done initializing the sweep rate - return */
    return;

}




/*
   sweep_down

   Description:      This function handles decreasing the current sweep rate.
             The new sweep rate (and sample size) is sent to the
             hardware (and trace routines).  If an attempt is made to
             lower the sweep rate below the minimum value it is not
             changed.  This routine also updates the sweep delay based
             on the new sweep rate (to keep the delay time constant).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - decremented if not already 0.
             delay - increased to keep delay time constant.

   Known Bugs:       The updated delay time is not displayed.  Since the time
```

```
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  sweep_down()
{
    /* variables */
    int  sample_size;       /* sample size for the new sweep rate */



    /* decrease the sweep rate, if not already the minimum */
    if (sweep > 0)  {
        /* not at minimum, adjust delay for new sweep */
    adjust_trg_delay(sweep, (sweep - 1));
    /* now set new sweep rate */
        sweep--;
    }

    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done with lowering the sweep rate - return */
    return;

}




/*
   sweep_up

   Description:      This function handles increasing the current sweep rate.
             The new sweep rate (and sample size) is sent to the
             hardware (and trace routines).  If an attempt is made to
             raise the sweep rate above the maximum value it is not
             changed.  This routine also updates the sweep delay based
             on the new sweep rate (to keep the delay time constant).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - incremented if not already the maximum value.
             delay - decreased to keep delay time constant.

   Known Bugs:       The updated delay time is not displayed.  Since the time
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  sweep_up()
{
    /* variables */
    int  sample_size;       /* sample size for the new sweep rate */



    /* increase the sweep rate, if not already the maximum */
    if (sweep < (NO_SWEEP_RATES - 1))  {
        /* not at maximum, adjust delay for new sweep */
    adjust_trg_delay(sweep, (sweep + 1));
    /* now set new sweep rate */
        sweep++;
    }

    /* set the sweep rate for the hardware */
    sample_size = set_sample_rate(sweep_rates[sweep].sample_rate);
    /* also set the sample size for the trace capture */
    set_trace_size(sample_size);


    /* all done with raising the sweep rate - return */
    return;

}




/*
   display_sweep

   Description:      This function displays the current sweep rate at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the sweep rate.
             y_pos (int) - y position (in character cells) at which to
                   display the sweep rate.
             style (int) - style with which to display the sweep rate.
   Return Value:     None.

   Input:            None.
   Output:           The sweep rate is displayed at the passed position on the
             display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sweep - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  display_sweep(int x_pos, int y_pos, int style)
{
    /* variables */
      /* none */



    /* display the sweep rate */
    plot_string(x_pos, y_pos, sweep_rates[sweep].s, style);


    /* all done displaying the sweep rate - return */
    return;

}




/*
   set_trg_level

   Description:      This function sets the trigger level to the passed value.

   Arguments:        l (int) - value to which to set the trigger level.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed value is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_level(int l)
{
    /* variables */
      /* none */



    /* set the trigger level */
    level = l;

    /* set the trigger level in hardware too */
    set_trigger(level, slope);


    /* all done initializing the trigger level - return */
    return;

}




/*
   trg_level_down

   Description:      This function handles decreasing the current trigger
             level.  The new trigger level is sent to the hardware.
             If an attempt is made to lower the trigger level below
             the minimum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - decremented if not already at the minimum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_level_down()
{
    /* variables */
      /* none */



    /* decrease the trigger level, if not already the minimum */
    if (level > MIN_TRG_LEVEL_SET)
        level--;

    /* set the trigger level for the hardware */
    set_trigger(level, slope);


    /* all done with lowering the trigger level - return */
    return;

}




/*
   trg_level_up

   Description:      This function handles increasing the current trigger
             level.  The new trigger level is sent to the hardware.
             If an attempt is made to raise the trigger level above
             the maximum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - incremented if not already the maximum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_level_up()
{
    /* variables */
      /* none */



    /* increase the trigger level, if not already the maximum */
    if (level < MAX_TRG_LEVEL_SET)
        level++;

    /* tell the hardware the new trigger level */
    set_trigger(level, slope);


    /* all done raising the trigger level - return */
    return;

}




/*
   display_trg_level

   Description:      This function displays the current trigger level at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger level.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger level.
             style (int) - style with which to display the trigger
                       level.
   Return Value:     None.

   Input:            None.
   Output:           The trigger level is displayed at the passed position on
             the display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: level - determines the value displayed.

   Author:           Glen George
   Last Modified:    Mar. 10, 1995

*/

void  display_trg_level(int x_pos, int y_pos, int style)
{
    /* variables */
    char      level_str[] = "        "; /* string containing the trigger level */
    long int  l;            /* trigger level in mV */



    /* compute the trigger level in millivolts */
    l = ((long int) MAX_LEVEL - MIN_LEVEL) * level / (MAX_TRG_LEVEL_SET - MIN_TRG_LEVEL_SET) + MIN_LEV

    /* convert the level to the string (leave first character blank) */
    cvt_num_field(l, &level_str[1]);

    /* add in the units */
    level_str[7] = 'V';


    /* now finally display the trigger level */
    plot_string(x_pos, y_pos, level_str, style);


    /* all done displaying the trigger level - return */
    return;

}




/*
   set_trg_slope

   Description:      This function sets the trigger slope to the passed value.

   Arguments:        s (enum slope_type) - trigger slope type to which to set
                       the locally global slope.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - set to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_slope(enum slope_type s)
```

```
{
    /* variables */
      /* none */



    /* set the slope type */
    slope = s;

    /* also tell the hardware what the slope is */
    set_trigger(level, slope);


    /* all done setting the trigger slope - return */
    return;

}




/*
   trg_slope_toggle

   Description:      This function handles toggling (and setting) the current
             trigger slope.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - toggled.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_slope_toggle()
{
    /* variables */
      /* none */



    /* toggle the trigger slope */
    if (slope == SLOPE_POSITIVE)
        slope = SLOPE_NEGATIVE;
    else
        slope = SLOPE_POSITIVE;

    /* set the new trigger slope */
    set_trigger(level, slope);


    /* all done with the trigger slope - return */
    return;

}




/*
   display_trg_slope

   Description:      This function displays the current trigger slope at the
                     passed position, in the passed style.
   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger slope.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger slope.
             style (int) - style with which to display the trigger
                       slope.
   Return Value:     None.

   Input:            None.
   Output:           The trigger slope is displayed at the passed position on
             the screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: slope - determines which string is displayed.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  display_trg_slope(int x_pos, int y_pos, int style)
{
    /* variables */

    /* the trigger slope strings (must match enumerated type) */
    const static char * const  slopes[] =  {  " +", " -"  };



    /* display the trigger slope */
    plot_string(x_pos, y_pos, slopes[slope], style);


    /* all done displaying the trigger slope - return */
    return;

}




/*
   set_trg_delay

   Description:      This function sets the trigger delay to the passed value.

   Arguments:        d (long int) - value to which to set the trigger delay.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   The passed value is not checked for validity.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - initialized to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trg_delay(long int d)
{
    /* variables */
      /* none */

    /* set the trigger delay */
    delay = d;

    /* set the trigger delay in hardware too */
    set_delay(delay);


    /* all done initializing the trigger delay - return */
    return;

}




/*
   trg_delay_down

   Description:      This function handles decreasing the current trigger
             delay.  The new trigger delay is sent to the hardware.
             If an attempt is made to lower the trigger delay below
             the minimum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - decremented if not already at the minimum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_delay_down()
{
    /* variables */
      /* none */



    /* decrease the trigger delay, if not already the minimum */
    if (delay > MIN_DELAY)
        delay--;

    /* set the trigger delay for the hardware */
    set_delay(delay);


    /* all done with lowering the trigger delay - return */
    return;

}




/*
   trg_delay_up

   Description:      This function handles increasing the current trigger
             delay.  The new trigger delay is sent to the hardware.
             If an attempt is made to raise the trigger delay above
             the maximum value it is not changed.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - incremented if not already the maximum value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trg_delay_up()
{
    /* variables */
      /* none */



    /* increase the trigger delay, if not already the maximum */
    if (delay < MAX_DELAY)
        delay++;

    /* tell the hardware the new trigger delay */
    set_delay(delay);


    /* all done raising the trigger delay - return */
    return;

}




/*
   adjust_trg_delay

   Description:      This function adjusts the trigger delay for a new sweep
             rate.  The factor to adjust the delay by is determined
             by looking up the sample rates in the sweep_rates array.
             If the delay goes out of range, due to the adjustment it
             is reset to the maximum or minimum valid value.

   Arguments:        old_sweep (int) - old sweep rate (index into sweep_rates
                       array).
             new_sweep (int) - new sweep rate (index into sweep_rates
                       array).
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       The delay is multiplied by 10 times the ratio of the
             sweep sample rates then divided by 10.  This is done to
             avoid floating point arithmetic and integer truncation
             problems.
   Data Structures:  None.

   Global Variables: delay - adjusted based on passed sweep rates.

   Known Bugs:       The updated delay time is not displayed.  Since the time
             is typically only rounded to the new sample rate, this is
             not a major problem.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/
```

```

static void  adjust_trg_delay(int old_sweep, int new_sweep)
{
    /* variables */
      /* none */



    /* multiply by 10 times the ratio of sweep rates */
    delay *= (10 * sweep_rates[new_sweep].sample_rate) / sweep_rates[old_sweep].sample_rate;
    /* now divide the factor of 10 back out */
    delay /= 10;

    /* make sure delay is not out of range */
    if (delay > MAX_DELAY)
        /* delay is too large - set to maximum */
        delay = MAX_DELAY;
    if (delay < MIN_DELAY)
        /* delay is too small - set to minimum */
    delay = MIN_DELAY;


    /* tell the hardware the new trigger delay */
    set_delay(delay);


    /* all done adjusting the trigger delay - return */
    return;

}




/*
   display_trg_delay

   Description:      This function displays the current trigger delay at the
                     passed position, in the passed style.

   Arguments:        x_pos (int) - x position (in character cells) at which to
                   display the trigger delay.
             y_pos (int) - y position (in character cells) at which to
                   display the trigger delay.
             style (int) - style with which to display the trigger
                       delay.
   Return Value:     None.

   Input:            None.
   Output:           The trigger delay is displayed at the passed position on
             the display.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: delay - determines the value displayed.

   Author:           Glen George
   Last Modified:    June 11, 2014

*/

void  display_trg_delay(int x_pos, int y_pos, int style)
{
    /* variables */
    char      delay_str[] = "         "; /* string containing the trigger delay */
    long int  units_adj;         /* adjustment to get to microseconds */

    long int  d;                         /* delay in appropriate units */
    float     temp_d;                    /* delay in float to avoid overflows */

    /* compute the delay in the appropriate units */
    /* have to watch out for overflow, so use float temp */
    if (sweep_rates[sweep].sample_rate > 1000000L)  {
        /* have a fast sweep rate  */
        /* first compute with float to avoid overflow */
        temp_d = delay * (1000000000L / sweep_rates[sweep].sample_rate);

    /* now convert to int */
    d = (int) temp_d;
    /* need to divide by 1000 to get to microseconds */
    units_adj = 1000;
    }
    else  {
        /* slow sweep rate, don't have to worry about overflow */
        d = delay * (1000000L / sweep_rates[sweep].sample_rate);
    /* already in microseconds, so adjustment is 1 */
    units_adj = 1;
    }

    /* convert it to the string (leave first character blank) */
    cvt_num_field(d, &delay_str[1]);

    /* add in the units */
    if (((d / units_adj) < 1000) && ((d / units_adj) > -1000) && (units_adj == 1000)) {
        /* delay is in microseconds */
    delay_str[7] = '\004';
    delay_str[8] = 's';
    }
    else if (((d / units_adj) < 1000000) && ((d / units_adj) > -1000000)) {
        /* delay is in milliseconds */
    delay_str[7] = 'm';
    delay_str[8] = 's';
    }
    else if (((d / units_adj) < 1000000000) && ((d / units_adj) > -1000000000))  {
        /* delay is in seconds */
    delay_str[7] = 's';
    delay_str[8] = ' ';
    }
    else  {
        /* delay is in kiloseconds */
    delay_str[7] = 'k';
    delay_str[8] = 's';
    }


    /* now actually display the trigger delay */
    plot_string(x_pos, y_pos, delay_str, style);


    /* all done displaying the trigger delay - return */
    return;

}




/*
   cvt_num_field

   Description:      This function converts the passed number (numeric field
                     value) to a string and returns that in the passed string
             reference.  The number may be signed, and a sign (+ or -)
             is always generated.  The number is assumed to have three
             digits to the right of the decimal point.  Only the four
             most significant digits of the number are displayed and
             the decimal point is shifted appropriately.  (Four digits
             are always generated by the function).

   Arguments:        n (long int) - numeric field value to convert.
             s (char *)    - pointer to string in which to return the
                        converted field value.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       The algorithm used assumes four (4) digits are being
             converted.
   Data Structures:  None.

   Global Variables: None.

   Known Bugs:       If the passed long int is the largest negative long int,
             the function will display garbage.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

static void  cvt_num_field(long int n, char *s)
{
    /* variables */
    int  dp = 3;        /* digits to right of decimal point */
    int  d;         /* digit weight (power of 10) */

    int  i = 0;         /* string index */



    /* first get the sign (and make n positive for conversion) */
    if (n < 0)  {
        /* n is negative, set sign and convert to positive */
    s[i++] = '-';
    n = -n;
    }
    else  {
        /* n is positive, set sign only */
    s[i++] = '+';
    }


    /* make sure there are no more than 4 significant digits */
    while (n > 9999)  {
        /* have more than 4 digits - get rid of one */
    n /= 10;
    /* adjust the decimal point */
    dp--;
    }

    /* if decimal point is non-positive, make positive */
    /* (assume will take care of adjustment with output units in this case) */
    while (dp <= 0)
       dp += 3;


    /* adjust dp to be digits to the right of the decimal point */
    /* (assuming 4 digits) */
    dp = 4 - dp;


    /* finally, loop getting and converting digits */
    for (d = 1000; d > 0; d /= 10)  {

        /* check if need decimal the decimal point now */
    if (dp-- == 0)
        /* time for decimal point */
        s[i++] = '.';

    /* get and convert this digit */
    s[i++] = (n / d) + '0';
    /* remove this digit from n */
    n %= d;
    }


    /* all done converting the number, return */
    return;

}

```

```
/****************************************************************************/
/*                                                                          */
/*                                MENUACT.H                                 */
/*                          Menu Action Functions                           */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants and function prototypes for the functions
   which carry out menu actions and display and initialize menu settings for
   the Digital Oscilloscope project (the functions are defined in menuact.c).


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed definition of enum scale_type (was
                     enum scale_status).
      3/10/95  Glen George       Changed MAX_TRG_LEVEL_SET (maximum trigger
                     level) to 127 to match specification.
      3/17/97  Glen George       Updated comments.
      5/3/06   Glen George       Updated comments.
      5/9/06   Glen George       Added a new mode (AUTO_TRIGGER) and a new
                                 scale (SCALE_GRID).
      5/9/06   Glen George       Added menu functions for mode and scale to
                                 move up and down a list instead of just
                 toggling the selection.
      5/9/06   Glen George       Added declaration for the accessor to the
                                 current trigger mode (get_trigger_mode).
*/



#ifndef  __MENUACT_H__
    #define  __MENUACT_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "lcdout.h"




/* constants */

/* min and max trigger level settings */
#define  MIN_TRG_LEVEL_SET   0
#define  MAX_TRG_LEVEL_SET   127

/* number of different sweep rates */
#define  NO_SWEEP_RATES     (sizeof(sweep_rates) / sizeof(struct sweep_info))




/* structures, unions, and typedefs */

/* types of triggering modes */
enum trigger_type  {  NORMAL_TRIGGER,       /* normal triggering */
              AUTO_TRIGGER,     /* automatic triggering */
              ONESHOT_TRIGGER       /* one-shot triggering */
           };

/* types of displayed scales */
enum scale_type    {  SCALE_NONE,       /* no scale is displayed */
              SCALE_AXES,       /* scale is a set of axes */
              SCALE_GRID        /* scale is a grid */
           };
```

```

/* types of trigger slopes */
enum slope_type    {  SLOPE_POSITIVE,       /* positive trigger slope */
              SLOPE_NEGATIVE        /* negative trigger slope */
           };

/* sweep rate information */
struct sweep_info  {  long int     sample_rate;    /* sample rate */
              const char  *s;          /* sweep rate string */
           };




/* function declarations */

/* menu option actions */
void  no_menu_action(void);    /* no action to perform */
void  mode_down(void);         /* change to the "next" trigger mode */
void  mode_up(void);           /* change to the "previous" trigger mode */
void  scale_down(void);        /* change to the "next" scale type */
void  scale_up(void);          /* change to the "previous" scale type */
void  sweep_down(void);        /* decrease the sweep rate */
void  sweep_up(void);          /* increase the sweep rate */
void  trg_level_down(void);    /* decrease the trigger level */
void  trg_level_up(void);      /* increase the trigger level */
void  trg_slope_toggle(void);  /* toggle the trigger slope */
void  trg_delay_down(void);    /* decrease the trigger delay */
void  trg_delay_up(void);      /* increase the trigger delay */

/* option accessor routines */
enum trigger_type  get_trigger_mode(void);  /* get the current trigger mode */

/* option initialization routines */
void  set_trigger_mode(enum trigger_type);  /* set the trigger mode */
void  set_scale(enum scale_type);           /* set the scale type */
void  set_sweep(int);                   /* set the sweep rate */
void  set_trg_level(int);               /* set the trigger level */
void  set_trg_slope(enum slope_type);       /* set the trigger slope */
void  set_trg_delay(long int);          /* set the tigger delay */

/* option display routines */
void  no_display(int, int, int);     /* no option setting to display */
void  display_mode(int, int, int);       /* display trigger mode */
void  display_scale(int, int, int);      /* display the scale type */
void  display_sweep(int, int, int);      /* display the sweep rate */
void  display_trg_level(int, int, int);  /* display the trigger level */
void  display_trg_slope(int, int, int);  /* display the trigger slope */
void  display_trg_delay(int, int, int);  /* display the tigger delay */


#endif

```

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

/*
   This file contains the general definitions for the Digital Oscilloscope
   project.  This includes constant and structure definitions along with the
   function declarations for the assembly language functions.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/17/97  Glen George       Removed KEYCODE_UNUSED (no longer used).
      5/3/06   Glen George       Added conditional definitions for handling
                                 different architectures.
      5/9/06   Glen George       Updated declaration of start_sample() to
                                 match the new specification.
      5/27/08  Glen George       Added check for __nios__ definition to also
                                 indicate the compilation is for an Altera
                     NIOS CPU.
      6/03/14  Santiago Navonne  Added cursor text area, and NO_TRACE value.
*/



#ifndef  __SCOPEDEF_H__
    #define  __SCOPEDEF_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "lcdout.h"




/* constants */

/* general constants */
#define  FALSE       0
#define  TRUE        !FALSE
#define  NULL        (void *) 0

/* display size (in characters) */
#define  LCD_WIDTH   (SIZE_X / HORIZ_SIZE)
#define  LCD_HEIGHT  (SIZE_Y / VERT_SIZE)

/* cursor area */
#define  CURSOR_STR_X      5
#define  CURSOR_STR_Y      5
#define  CURSOR_STR_W      100
#define  CURSOR_STR_H      7



/* macros */

/* let __nios__ also mean a NIOS compilation */
#ifdef  __nios__
  #define  NIOS         /* use the standard NIOS defintion */
#endif

/* add the definitions necessary for the Altera NIOS chip */
#ifdef  NIOS
  #define  FLAT_MEMORY      /* use the flat memory model */
```

```
#endif


/* if a flat memory model don't need far pointers */
#ifdef  FLAT_MEMORY
  #define  far
#endif




/* structures, unions, and typedefs */

/* program states */
enum status  {  MENU_ON,    /* menu is displayed with the cursor in it */
        MENU_OFF,   /* menu is not displayed - no cursor */
        NUM_STATES  /* number of states */
         };

/* key codes */
enum keycode  {  KEYCODE_MENU,      /* <Menu>     */
             KEYCODE_UP,        /* <Up>       */
             KEYCODE_DOWN,      /* <Down>     */
             KEYCODE_LEFT,      /* <Left>     */
             KEYCODE_RIGHT,     /* <Right>    */
             KEYCODE_ILLEGAL,   /* other keys */
         NUM_KEYCODES       /* number of key codes */
              };




/* function declarations */

/* keypad functions */
unsigned char  key_available(void);     /* key is available */
int            getkey(void);        /* get a key */

/* display functions  */
void  clear_display(void);                /* clear the display */
void  plot_pixel(unsigned int, unsigned int, int);    /* output a pixel */

/* sampling parameter functions */
int   set_sample_rate(long int);    /* set the sample rate */
void  set_trigger(int, int);        /* set trigger level and slope */
void  set_delay(long int);      /* set the trigger delay time */

/* sampling functions */
void                start_sample(int);  /* capture a sample */
unsigned char *sample_done(void);   /* sample captured status */


#endif
```

```
/***************************************************************************/
/*                                                                         */
/*                                TRACUTIL                                 */
/*                         Trace Utility Functions                         */
/*                       Digital Oscilloscope Project                      */
/*                                EE/CS 52                                 */
/*                                                                         */
/***************************************************************************/

/*
   This file contains the utility functions for handling traces (capturing
   and displaying data) for the Digital Oscilloscope project.  The functions
   included are:
      clear_saved_areas  - clear all the save areas
      do_trace           - start a trace
      init_trace         - initialize the trace routines
      plot_trace         - plot a trace (sampled data)
      restore_menu_trace - restore the saved area under the menus
      restore_trace      - restore the saved area of a trace
      set_display_scale  - set the type of displayed scale (and display it)
      set_mode           - set the triggering mode
      set_save_area      - determine an area of a trace to save
      set_trace_size     - set the number of samples in a trace
      trace_done         - inform this module that a trace has been completed
      trace_rdy          - determine if system is ready to start another trace
      trace_rearm        - re-enable tracing (in one-shot triggering mode)

   The local functions included are:
      none

   The locally global variable definitions included are:
      cur_scale    - current scale type
      sample_size  - the size of the sample for the trace
      sampling     - currently doing a sample
      saved_area   - saved trace under a specified area
      saved_axis_x - saved trace under the x lines (axes or grid)
      saved_axis_y - saved trace under the y lines (axes or grid)
      saved_menu   - saved trace under the menu
      saved_pos_x  - starting position (x coorindate) of area to save
      saved_pos_y  - starting position (y coorindate) of area to save
      saved_end_x  - ending position (x coorindate) of area to save
      saved_end_y  - ending position (y coorindate) of area to save
      trace_status - whether or not ready to start another trace


   Revision History
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Fixed inversion of signal in plot_trace.
      3/13/94  Glen George       Added sampling flag and changed the functions
                         init_trace, do_trace and trace_done to update
                 the flag.  Also the function trace_rdy now
                 uses it.  The function set_mode was updated
                 to always say a trace is ready for normal
                 triggering.
      3/13/94  Glen George       Fixed bug in trace restoring due to operator
                       misuse (&& instead of &) in the functions
                 set_axes, restore_menu_trace, and
                 restore_trace.
      3/13/94  Glen George       Fixed bug in trace restoring due to the clear
                     function (clear_saved_areas) not clearing all
                 of the menu area.
      3/13/94  Glen George       Fixed comparison bug when saving traces in
                     plot_trace.
      3/13/94  Glen George       Changed name of set_axes to set_display_scale
                     and the name of axes_state to cur_scale to
                 more accurately reflect the function/variable
                 use (especially if add scale display types).
      3/17/97  Glen George       Updated comments.
      3/17/97  Glen George       Changed set_display_scale to use plot_hline
                     and plot_vline functions to output axes.
      5/3/06   Glen George       Updated formatting.
      5/9/06   Glen George       Updated do_trace function to match the new
                                 definition of start_sample().
      5/9/06   Glen George       Removed normal_trg variable, its use is now
```

```
                                 handled by the get_trigger_mode() accessor.
      5/9/06   Glen George       Added tick marks to the axes display.
      5/9/06   Glen George       Added ability to display a grid.
      5/27/08  Glen George       Added is_sampling() function to be able to
                             tell if the system is currently taking a
                 sample.
      5/27/08  Glen George       Changed set_mode() to always turn off the
                             sampling flag so samples with the old mode
                                 setting are ignored.
      6/3/08   Glen George       Fixed problems with non-power of 2 display
                 sizes not working.
      6/3/14   Santiago Navonne  Changed UI display colors; changed plot_trace
                                 to clear just trace instead of whole display.
*/



/* library include files */
  /* none */

/* local include files */
#include  "scopedef.h"
#include  "lcdout.h"
#include  "menu.h"
#include  "menuact.h"
#include  "tracutil.h"




/* locally global variables */

static int  trace_status;   /* ready to start another trace */

static int  sampling;           /* currently sampling data */

static int  sample_size;    /* number of data points in a sample */

static int old_sample[SIZE_X]; /* sample currently being displayed */

static enum scale_type  cur_scale;  /* current display scale type */

/* traces (sampled data) saved under the axes */
static unsigned char  saved_axis_x[2 * Y_TICK_CNT + 1][PLOT_SIZE_X/8];  /* saved trace under x lines */
static unsigned char  saved_axis_y[2 * X_TICK_CNT + 1][PLOT_SIZE_Y/8];  /* saved trace under y lines */

/* traces (sampled data) saved under the menu */
static unsigned char  saved_menu[MENU_SIZE_Y][(MENU_SIZE_X + 7)/8];

/* traces (sampled data) saved under any area */
static unsigned char  saved_area[SAVE_SIZE_Y][SAVE_SIZE_X/8]; /* saved trace under any area */
static int        saved_pos_x;    /* starting x position of saved area */
static int        saved_pos_y;    /* starting y position of saved area */
static int        saved_end_x;    /* ending x position of saved area */
static int        saved_end_y;    /* ending y position of saved area */




/*
   init_trace

   Description:      This function initializes all of the locally global
                     variables used by these routines.  The saved areas are
             set to non-existant with cleared saved data.  Normal
             normal triggering is set, the system is ready for a
             trace, the scale is turned off and the sample size is set
             to the screen size.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to TRUE.
             sampling     - set to FALSE.
             cur_scale    - set to SCALE_NONE (no displayed scale).
             sample_size  - set to screen size (SIZE_X).
             saved_axis_x - cleared.
             saved_axis_y - cleared.
             saved_menu   - cleared.
             saved_area   - cleared.
             saved_pos_x  - set to off-screen.
             saved_pos_y  - set to off-screen.
             saved_end_x  - set to off-screen.
             saved_end_y  - set to off-screen.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  init_trace()
{
    /* variables */
      /* none */



    /* initialize system status variables */

    /* ready for a trace */
    trace_status = TRUE;

    /* not currently sampling data */
    sampling = FALSE;

    /* turn off the displayed scale */
    cur_scale = SCALE_NONE;

    /* sample size is the screen size */
    sample_size = SIZE_X;


    /* clear save areas */
    clear_saved_areas();

    /* also clear the general saved area location variables (off-screen) */
    saved_pos_x = SIZE_X + 1;
    saved_pos_y = SIZE_Y + 1;
    saved_end_x = SIZE_X + 1;
    saved_end_y = SIZE_Y + 1;


    /* done initializing, return */
    return;

}




/*
   set_mode

   Description:      This function sets the locally global triggering mode
                     based on the passed value (one of the possible enumerated
             values).  The triggering mode is used to determine when
             the system is ready for another trace.  The sampling flag
                     is also reset so a new sample will be started (if that is
                     appropriate).

   Arguments:        trigger_mode (enum trigger_type) - the mode with which to
                            set the triggering.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling     - set to FALSE to turn off sampling
                 trace_status - set to TRUE if not one-shot triggering.

   Author:           Glen George
   Last Modified:    May 27, 2008

*/

void  set_mode(enum trigger_type trigger_mode)
{
    /* variables */
      /* none */



    /* if not one-shot triggering - ready for trace too */
    trace_status = (trigger_mode != ONESHOT_TRIGGER);


    /* turn off the sampling flag so will start a new sample */
    sampling = FALSE;


    /* all done, return */
    return;

}




/*
   is_sampling

   Description:      This function determines whether the system is currently
                     taking a sample or not.  This is just the value of the
             sampling flag.

   Arguments:        None.
   Return Value:     (int) - the current sampling status (TRUE if currently
             trying to take a sample, FALSE otherwise).

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling - determines if taking a sample or not.

   Author:           Glen George
   Last Modified:    May 27, 2008

*/

int  is_sampling()
{
    /* variables */
      /* none */



    /* currently sampling if sampling flag is set */
    return  sampling;

}




/*
   trace_rdy

   Description:      This function determines whether the system is ready to
                     start another trace.  This is determined by whether or
             not the system is still sampling (sampling flag) and if
             it is ready for another trace (trace_status flag).

   Arguments:        None.
   Return Value:     (int) - the current trace status (TRUE if ready to do
             another trace, FALSE otherwise).

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sampling     - determines if ready for another trace.
             trace_status - determines if ready for another trace.

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

int  trace_rdy()
{
    /* variables */
      /* none */



    /* ready for another trace if not sampling and trace is ready */
    return  (!sampling && trace_status);

}




/*
   trace_done

   Description:      This function is called to indicate a trace has been
                     completed.  If in normal triggering mode this means the
             system is ready for another trace.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - may be set to TRUE.
             sampling     - set to FALSE.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/
```

```

void  trace_done()
{
    /* variables */
      /* none */



    /* done with a trace - if retriggering, ready for another one */
    if (get_trigger_mode() != ONESHOT_TRIGGER)
        /* in a retriggering mode - set trace_status to TRUE (ready) */
    trace_status = TRUE;

    /* no longer sampling data */
    sampling = FALSE;


    /* done so return */
    return;

}




/*
   trace_rearm

   Description:      This function is called to rearm the trace.  It sets the
                     trace status to ready (TRUE).  It is used to rearm the
             trigger in one-shot mode.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to TRUE.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  trace_rearm()
{
    /* variables */
      /* none */



    /* rearm the trace - set status to ready (TRUE) */
    trace_status = TRUE;


    /* all done - return */
    return;

}




/*
   set_trace_size

   Description:      This function sets the locally global sample size to the
                     passed value.  This is used to scale the data when
             plotting a trace.

   Arguments:        size (int) - the trace sample size.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: sample_size - set to the passed value.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_trace_size(int size)
{
    /* variables */
      /* none */



    /* set the locally global sample size */
    sample_size = size;


    /* all done, return */
    return;

}




/*
   set_display_scale

   Description:      This function sets the displayed scale type to the passed
             argument.  If the scale is turned on, it draws it.  If it
             is turned off (SCALE_NONE), it restores the saved trace
             under the scale.  Scales can be axes with tick marks
                     (SCALE_AXES) or a grid (SCALE_GRID).

   Arguments:        scale (scale_type) - new scale type.
   Return Value:     None.

   Input:            None.
   Output:           Either a scale is output or the trace under the old scale
             is restored.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: cur_scale    - set to the passed value.
             saved_axis_x - used to restore trace data under x-axis.
             saved_axis_y - used to restore trace data under y-axis.

   Author:           Glen George
   Last Modified:   June 03, 2014

*/

void  set_display_scale(enum scale_type scale)
{
    /* variables */
    int  p;             /* x or y coordinate */

    int  i;     /* loop indices */
    int  j;
    /* whenever change scale type, need to clear out previous scale */
    /* unnecessary if going to SCALE_GRID or from SCALE_NONE or not changing the scale */
    if ((scale != SCALE_GRID) && (cur_scale != SCALE_NONE) && (scale != cur_scale))  {

        /* need to restore the trace under the lines (tick, grid, or axis) */

    /* go through all points on horizontal lines */
    for (j = -Y_TICK_CNT; j <= Y_TICK_CNT; j++)  {

        /* get y position of the line */
        p = X_AXIS_POS + j * Y_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_Y)
            p = PLOT_SIZE_Y - 1;
        if (p < 0)
            p = 0;

        /* look at entire horizontal line */
        for (i = 0; i < PLOT_SIZE_X; i++)  {
            /* check if this point is on or off (need to look at bits) */
        if ((saved_axis_x[j + Y_TICK_CNT][i / 8] & (0x80 >> (i % 8))) == 0)
            /* saved pixel is off */
            plot_pixel(i, p, PIXEL_CLEAR);
        else
            /* saved pixel is on */
            plot_pixel(i, p, PIXEL_TRACE);
        }
    }

    /* go through all points on vertical lines */
    for (j = -X_TICK_CNT; j <= X_TICK_CNT; j++)  {

        /* get x position of the line */
        p = Y_AXIS_POS + j * X_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_X)
            p = PLOT_SIZE_X - 1;
        if (p < 0)
            p = 0;

        /* look at entire vertical line */
        for (i = 0; i < PLOT_SIZE_Y; i++)  {
            /* check if this point is on or off (need to look at bits) */
        if ((saved_axis_y[j + X_TICK_CNT][i / 8] & (0x80 >> (i % 8))) == 0)
            /* saved pixel is off */
            plot_pixel(p, i, PIXEL_CLEAR);
        else
            /* saved pixel is on */
            plot_pixel(p, i, PIXEL_TRACE);
        }
    }
    }


    /* now handle the scale type appropriately */
    switch (scale)  {

        case SCALE_AXES:    /* axes for the scale */
        case SCALE_GRID:    /* grid for the scale */

                    /* draw x lines (grid or tick marks) */
                for (i = -Y_TICK_CNT; i <= Y_TICK_CNT; i++)  {

                /* get y position of the line */
                p = X_AXIS_POS + i * Y_TICK_SIZE;
                /* make sure it is in range */
                if (p >= PLOT_SIZE_Y)
                    p = PLOT_SIZE_Y - 1;
                if (p < 0)
                    p = 0;

                /* should we draw a grid, an axis, or a tick mark */
                if (scale == SCALE_GRID)
                    /* drawing a grid line */
                        plot_hline(X_GRID_START, p, (X_GRID_END - X_GRID_START));
                else if (i == 0)
                    /* drawing the x axis */
                        plot_hline(X_AXIS_START, p, (X_AXIS_END - X_AXIS_START));
                else
                    /* must be drawing a tick mark */
                        plot_hline((Y_AXIS_POS - (TICK_LEN / 2)), p, TICK_LEN);
                }

                    /* draw y lines (grid or tick marks) */
                for (i = -X_TICK_CNT; i <= X_TICK_CNT; i++)  {

                /* get x position of the line */
                p = Y_AXIS_POS + i * X_TICK_SIZE;
                /* make sure it is in range */
                if (p >= PLOT_SIZE_X)
                    p = PLOT_SIZE_X - 1;
                    if (p < 0)
                    p = 0;

                /* should we draw a grid, an axis, or a tick mark */
                if (scale == SCALE_GRID)
                    /* drawing a grid line */
                        plot_vline(p, Y_GRID_START, (Y_GRID_END - Y_GRID_START));
                else if (i == 0)
                    /* drawing the y axis */
                        plot_vline(p, Y_AXIS_START, (Y_AXIS_END - Y_AXIS_START));
                else
                    /* must be drawing a tick mark */
                        plot_vline(p, (X_AXIS_POS - (TICK_LEN / 2)), TICK_LEN);
                }

                /* done with the axes */
                break;

        case SCALE_NONE:    /* there is no scale */
                /* already restored plot so nothing to do */
                break;

    }


    /* now remember the new (now current) scale type */
    cur_scale = scale;


    /* scale is taken care of, return */
    return;

}




/*
   clear_saved_areas

   Description:      This function clears all the saved areas (for saving the
                     trace under the axes, menus, and general areas).

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_axis_x - cleared.
             saved_axis_y - cleared.
```

```
                saved_menu   - cleared.
                saved_area   - cleared.

   Author:           Glen George
   Last Modified:    May 9, 2006

*/

void  clear_saved_areas()
{
    /* variables */
    int  i;     /* loop indices */
    int  j;



    /* clear x-axis and y-axis save areas */
    for (j = 0; j <= (2 * Y_TICK_CNT); j++)
        for (i = 0; i < (SIZE_X / 8); i++)
            saved_axis_x[j][i] = 0;
    for (j = 0; j <= (2 * X_TICK_CNT); j++)
        for (i = 0; i < (SIZE_Y / 8); i++)
            saved_axis_y[j][i] = 0;

    /* clear the menu save ares */
    for (i = 0; i < MENU_SIZE_Y; i++)
        for (j = 0; j < ((MENU_SIZE_X + 7) / 8); j++)
        saved_menu[i][j] = 0;

    /* clear general save area */
    for (i = 0; i < SAVE_SIZE_Y; i++)
        for (j = 0; j < (SAVE_SIZE_X / 8); j++)
        saved_area[i][j] = 0;


    /* done clearing the saved areas - return */
    return;

}




/*
   restore_menu_trace

   Description:      This function restores the trace under the menu when the
                     menus are turned off.  (The trace was previously saved.)

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The trace under the menu is restored to the LCD screen.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_menu - used to restore trace data under the menu.

   Author:           Glen George
   Last Modified:    June 03, 2014

*/

void  restore_menu_trace()
{
    /* variables */
    int  bit_position;  /* position of bit to restore (in saved data) */
    int  bit_offset;    /* offset (in bytes) of bit within saved row */

    int  x;     /* loop indices */
    int  y;
    /* loop, restoring the trace under the menu */
    for (y = MENU_UL_Y; y < (MENU_UL_Y + MENU_SIZE_Y); y++)  {

        /* starting a row - initialize bit position */
    bit_position = 0x80;    /* start at high-order bit in the byte */
    bit_offset = 0;     /* first byte of the row */

        for (x = MENU_UL_X; x < (MENU_UL_X + MENU_SIZE_X); x++)  {

        /* check if this point is on or off (need to look at bits) */
        if ((saved_menu[y - MENU_UL_Y][bit_offset] & bit_position) == 0)
            /* saved pixel is off */
        plot_pixel(x, y, PIXEL_CLEAR);
        else
            /* saved pixel is on */
        plot_pixel(x, y, PIXEL_TRACE);

        /* move to the next bit position */
        bit_position >>= 1;
        /* check if moving to next byte */
        if (bit_position == 0)  {
            /* now on high bit of next byte */
        bit_position = 0x80;
        bit_offset++;
        }
        }
    }


    /* restored menu area - return */
    return;

}




/*
   set_save_area

   Description:      This function sets the position and size of the area to
                     be saved when traces are drawn.  It also clears any data
             currently saved.

   Arguments:        pos_x (int)  - x position of upper left corner of the
                    saved area.
             pos_y (int)  - y position of upper left corner of the
                        saved area.
             size_x (int) - horizontal size of the saved area.
             size_y (int) - vertical size of the saved area.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_area  - cleared.
             saved_pos_x - set to passed value.
             saved_pos_y - set to passed value.
             saved_end_x - computed from passed values.
             saved_end_y - computed from passed values.

   Author:           Glen George
   Last Modified:    Mar. 8, 1994

*/

void  set_save_area(int pos_x, int pos_y, int size_x, int size_y)
```

```
{
    /* variables */
    int  x;     /* loop indices */
    int  y;



    /* just setup all the locally global variables from the passed values */
    saved_pos_x = pos_x;
    saved_pos_y = pos_y;
    saved_end_x = pos_x + size_x;
    saved_end_y = pos_y + size_y;


    /* clear the save area */
    for (y = 0; y < SAVE_SIZE_Y; y++)  {
        for (x = 0; x < (SAVE_SIZE_X / 8); x++)  {
        saved_area[y][x] = 0;
        }
    }


    /* setup the saved area - return */
    return;

}




/*
   restore_trace

   Description:      This function restores the trace under the set saved
                     area.  (The area was previously set and the trace was
             previously saved.)

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           The trace under the saved ares is restored to the LCD.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: saved_area  - used to restore trace data.
             saved_pos_x - gives starting x position of saved area.
             saved_pos_y - gives starting y position of saved area.
             saved_end_x - gives ending x position of saved area.
             saved_end_y - gives ending y position of saved area.

   Author:           Glen George
   Last Modified:    June 03, 2014

*/

void  restore_trace()
{
    /* variables */
    int  bit_position;  /* position of bit to restore (in saved data) */
    int  bit_offset;    /* offset (in bytes) of bit within saved row */

    int  x;     /* loop indices */
    int  y;



    /* loop, restoring the saved trace */
    for (y = saved_pos_y; y < saved_end_y; y++)  {

        /* starting a row - initialize bit position */
    bit_position = 0x80;    /* start at high-order bit in the byte */
    bit_offset = 0;     /* first byte of the row */

        for (x = saved_pos_x; x < saved_end_x; x++)  {

        /* check if this point is on or off (need to look at bits) */
        if ((saved_area[y - saved_pos_y][bit_offset] & bit_position) == 0)
            /* saved pixel is off */
        plot_pixel(x, y, PIXEL_CLEAR);
        else
            /* saved pixel is on */
        plot_pixel(x, y, PIXEL_TRACE);

        /* move to the next bit position */
        bit_position >>= 1;
        /* check if moving to next byte */
        if (bit_position == 0)  {
            /* now on high bit of next byte */
        bit_position = 0x80;
        bit_offset++;
        }
        }
    }


    /* restored the saved area - return */
    return;

}




/*
   do_trace

   Description:      This function starts a trace.  It starts the hardware
                     sampling data (via a function call) and sets the trace
             ready flag (trace_status) to FALSE and the sampling flag
             (sampling) to TRUE.

   Arguments:        None.
   Return Value:     None.

   Input:            None.
   Output:           None.

   Error Handling:   None.

   Algorithms:       None.
   Data Structures:  None.

   Global Variables: trace_status - set to FALSE (not ready for another trace).
             sampling     - set to TRUE (doing a sample now).

   Author:           Glen George
   Last Modified:    Mar. 13, 1994

*/

void  do_trace()
{
    /* variables */
      /* none */



    /* start up the trace */
    /* indicate whether using automatic triggering or not */
    start_sample(get_trigger_mode() == AUTO_TRIGGER);

    /* now not ready for another trace (currently doing one) */
    trace_status = FALSE;

    /* and are currently sampling data */
    sampling = TRUE;


    /* trace is going, return */
    return;

}


/*
   plot_trace

   Description:      This function plots the passed trace.  The trace is
                     assumed to contain sample_size points of sampled data.
                     Any points falling within any of the save areas are also
                     saved by this routine.  The data is also scaled to be
                     within the range of the entire screen.


   Arguments:        sample (unsigned char far *) - sample to plot.
   Return Value:     None.

   Input:            None.
   Output:           The sample is plotted on the screen.

   Error Handling:   None.

   Algorithms:       If there are more sample points than screen width the
             sample is plotted with multiple points per horizontal
             position.
   Data Structures:  None.

   Global Variables: cur_scale    - determines type of scale to plot.
             sample_size  - determines size of passed sample.
             saved_axis_x - stores trace under x-axis.
             saved_axis_y - stores trace under y-axis.
             saved_menu   - stores trace under the menu.
             saved_area   - stores trace under the saved area.
             saved_pos_x  - determines location of saved area.
             saved_pos_y  - determines location of saved area.
             saved_end_x  - determines location of saved area.
             saved_end_y  - determines location of saved area.

   Author:           Glen George
   Last Modified:    June 03, 2014

*/

void  plot_trace(unsigned char *sample)
{
    /* variables */
    int  x = 0;             /* current x position to plot */
    int  x_pos = (PLOT_SIZE_X / 2); /* "fine" x position for multiple point plotting */

    int  y;             /* y position of point to plot */

    int  p;                             /* an x or y coordinate */

    int  i;             /* loop indices */
    int  j;


    /* clear the saved areas too */
    clear_saved_areas();

    /* re-display the menu (if it was on) */
    refresh_menu();


    /* plot the sample */
    for (i = 0; i < sample_size; i++)  {

        /* determine y position of point (note: screen coordinates invert) */
    y = (PLOT_SIZE_Y - 1) - ((sample[i] * (PLOT_SIZE_Y - 1)) / 255);

    /* clear previous point on trace */
    plot_pixel(i, old_sample[i], PIXEL_CLEAR);

        /* plot this point */
    plot_pixel(x, y, PIXEL_TRACE);

    /* and save new value */
    old_sample[i] = y;


    /* check if the point is in a save area */

    /* check if in the menu area */
    if ((x >= MENU_UL_X) && (x < (MENU_UL_X + MENU_SIZE_X)) &&
        (y >= MENU_UL_Y) && (y < (MENU_UL_Y + MENU_SIZE_Y)))
        /* point is in the menu area - save it */
        saved_menu[y - MENU_UL_Y][(x - MENU_UL_X)/8] |= (0x80 >> ((x - MENU_UL_X) % 8));

    /* check if in the saved area */
    if ((x >= saved_pos_x) && (x <= saved_end_x) && (y >= saved_pos_y) && (y <= saved_end_y))
        /* point is in the save area - save it */
        saved_area[y - saved_pos_y][(x - saved_pos_x)/8] |= (0x80 >> ((x - saved_pos_x) % 8));

    /* check if on a grid line */
    /* go through all the horizontal lines */
    for (j = -Y_TICK_CNT; j <= Y_TICK_CNT; j++)  {

        /* get y position of the line */
        p = X_AXIS_POS + j * Y_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_Y)
            p = PLOT_SIZE_Y - 1;
        if (p < 0)
            p = 0;

        /* if the point is on this line, save it */
        if (y == p)
        saved_axis_x[j + Y_TICK_CNT][x / 8] |= (0x80 >> (x % 8));
    }

    /* go through all the vertical lines */
    for (j = -X_TICK_CNT; j <= X_TICK_CNT; j++)  {

        /* get x position of the line */
        p = Y_AXIS_POS + j * X_TICK_SIZE;
        /* make sure it is in range */
        if (p >= PLOT_SIZE_X)
            p = PLOT_SIZE_X - 1;
        if (p < 0)
            p = 0;

        /* if the point is on this line, save it */
        if (x == p)
        saved_axis_y[j + X_TICK_CNT][y / 8] |= (0x80 >> (y % 8));
    }


    /* update x position */
    x_pos += PLOT_SIZE_X;
    /* check if at next horizontal position */
    if (x_pos >= sample_size)  {
        /* at next position - update positions */
        x++;
        x_pos -= sample_size;
    }
    }


    /* finally, output the scale if need be */
    set_display_scale(cur_scale);


    /* done with plot, return */
    return;

}
```

```
/****************************************************************************/
/*                                                                          */
/*                                TRACUTIL.H                                */
/*                         Trace Utility Functions                          */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants and function prototypes for the trace
   utility functions (defined in tracutil.c) for the Digital Oscilloscope
   project.


   Revision History:
      3/8/94   Glen George       Initial revision.
      3/13/94  Glen George       Updated comments.
      3/13/94  Glen George       Changed name of set_axes function to
                     set_display_scale.
      5/9/06   Glen George       Added the constants for grids and tick marks.
      5/27/08  Glen George       Added is_sampling() function to be able to
                             tell if the system is currently taking a
                 sample.
      6/3/08   Glen George       Removed Y_SCALE_FACTOR - no longer used to
                             fix problems with non-power of 2 display
                 sizes.
*/



#ifndef  __TRACUTIL_H__
    #define  __TRACUTIL_H__


/* library include files */
  /* none */

/* local include files */
#include  "interfac.h"
#include  "menuact.h"




/* constants */

/* plot size */
#define  PLOT_SIZE_X    SIZE_X      /* plot takes entire screen width */
#define  PLOT_SIZE_Y    SIZE_Y      /* plot takes entire screen height */

/* axes position and size */
#define  X_AXIS_START   0          /* starting x position of x-axis */
#define  X_AXIS_END     (PLOT_SIZE_X - 1)  /* ending x position of x-axis */
#define  X_AXIS_POS (PLOT_SIZE_Y / 2)  /* y position of x-axis */
#define  Y_AXIS_START   0          /* starting y position of y-axis */
#define  Y_AXIS_END     (PLOT_SIZE_Y - 1)  /* ending y position of y-axis */
#define  Y_AXIS_POS (PLOT_SIZE_X / 2)  /* x position of y-axis */

/* tick mark and grid constants */
#define  TICK_LEN       5           /* length of axis tick mark */
/* tick mark counts are for a single quadrant, thus total number of tick */
/* marks or grids is twice this number */
#define  X_TICK_CNT     5       /* always 5 tick marks on x axis */
#define  X_TICK_SIZE    (PLOT_SIZE_X / (2 * X_TICK_CNT))   /* distance between tick marks */
#define  Y_TICK_SIZE    X_TICK_SIZE     /* same size as x */
#define  Y_TICK_CNT     (PLOT_SIZE_Y / (2 * Y_TICK_SIZE))  /* number of y tick marks */
#define  X_GRID_START   0          /* starting x position of x grid */
#define  X_GRID_END     (PLOT_SIZE_X - 1)  /* ending x position of x grid */
#define  Y_GRID_START   0          /* starting y position of y-axis */
#define  Y_GRID_END     (PLOT_SIZE_Y - 1)  /* ending y position of y-axis */

/* maximum size of the save area (in pixels) */
#define  SAVE_SIZE_X    120 /* maximum width */
```

```
#define  SAVE_SIZE_Y    16  /* maximum height */

/* sleep time between samples, designed to reduce blinking */
#define  DRAW_INTERVAL  50000




/* structures, unions, and typedefs */
    /* none */




/* function declarations */

/* initialize the trace utility routines */
void  init_trace(void);

/* trace status functions */
void  set_mode(enum trigger_type);  /* set the triggering mode */
int   is_sampling(void);        /* currently trying to take a sample */
int   trace_rdy(void);              /* determine if ready to start a trace */
void  trace_done(void);             /* signal a trace has been completed */
void  trace_rearm(void);            /* re-enable tracing */

/* trace save area functions */
void  clear_saved_areas(void);        /* clears all saved areas */
void  restore_menu_trace(void);           /* restore the trace under menus */
void  set_save_area(int, int, int, int);  /* set an area of a trace to save */
void  restore_trace(void);                /* restore saved area of a trace */

/* set the scale type */
void  set_display_scale(enum scale_type);

/* setup and plot a trace */
void  set_trace_size(int);              /* set the number of samples in a trace */
void  do_trace(void);                   /* start a trace */
void  plot_trace(unsigned char *);  /* plot a trace (sampled data) */


#endif

```

```
/****************************************************************************/
/*                                                                          */
/*                                 TRIGGER.S                                */
/*                       Data sampling and triggering                       */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                             Santiago Navonne                             */
/*                                                                          */
/****************************************************************************/

/*
   Data sampling and triggering control routines for the EE/CS 52 Digital
   Oscilloscope project. Function definitions are included in this file, and
   are laid out as follows:
    - set_sample_rate: Configures the sampling rate;
    - set_trigger: Configures the manual trigger level and slope;
    - set_delay: Configures the manual trigger delay;
    - start_sample: Starts a new data sample with the previously configured
                    settings and passed auto-trigger configuration;
    - sample_done: Checks whether a new data sample set is available, returning
                   a pointer to a buffer containing it if there is, or a NULL
                   pointer if there isn't;
    - sample_handler: Handles sampling FIFO full interrupts;
    - trigger_init: Initializes the environment's shared variables and the
                    triggering logic circuit (resetting it), effectively
                    preparing the sampling/triggering interface for use.


   Revision History:
      5/29/14 Santiago Navonne  Initial revision.
      6/01/14 Santiago Navonne  Minor fixes; updated documentation.
      6/11/14 Santiago Navonne  Changed division algorithm in set_sample_rate.
*/

/* Includes */
#include "general.h"  /* General assembly constants */
#include "system.h"   /* Base addresses */
#include "interfac.h" /* Software interface definitions */
#include "trigger.h"  /* Local constants */


/* Variables */
    .section .data        /* No alignment necessary: variables are bytes */
sample_pending: .byte 0   /* Logical value: whether a sample is pending */
sample: .skip FIFO_SIZE   /* Sample buffer */

    .section .text        /* Code starts here */

/*
 *  set_sample_rate
 *
 *  Description:       This procedure configures the sampling rate of the sampling
 *                     interface. After execution, the interface will start sampling
 *                     at the requested rate, rounded up to a multiple of the system
 *                     clock. The return value is how many samples will be acquired,
 *                     which is always the size of the FIFO.
 *                     If an argument of 0 is passed, the function has no effect, and
 *                     returns 0. The argument must however by less than or equal to
 *                     the system clock divided by two; no error checking is performed
 *                     on this.
 *
 *  Operation:         The procedure starts by error checking the value of the argument,
 *                     simply returning 0 if it is invalid. Then, it computes the
 *                     required clock period in system clock periods by dividing the
 *                     system clock frequency by the requested sample rate.
 *                     Finally, it saves the computed value to the trigger period
 *                     register, and pulses the reset bit in the control register to
 *                     reset the triggering logic. SIZE_X is ultimately moved into
 *                     r2 as constant return value.
 *
 *  Arguments:         samples_per_sec - positive integer indicating the sample rate
 *                                       in samples per second (r4). The value must
 *                                       be less than or equal to the system clock
 *                                       divided by two.
 *
 *  Return Value:      sample_num - positive integer, number of samples that will be
 *                                  acquired at the desired rate (r2).
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    If the argument is zero, the function has no effect, and returns 0.
 *                     No error checking is performed on the upper bound of the sampling
 *                     rate.
 *
 *  Limitations:       Resulting sample clock is an integer multiple of the system clock;
 *                     corresponding rate will be greater than or equal to the requested
 *                     rate, with a difference in period less than the system clock's.
 *                     Number of samples acquired must be <= FIFO_SIZE per hardware
 *                     limitations (size of FIFO).
 *
 *  Algorithms:        Division is performed using a repeated subtraction algorithm since
 *                     hardware division cannot be assumed to be available. This algorithm
 *                     is acceptable because generally very few iterations will be needed
 *                     to reach the result.
 *  Data Structures:   None.
 *
 *  Registers Changed: r2, r4, r8, r9.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Added error checking, expanded documentation.
 *      6/11/14   Santiago Navonne     Changed hardware divide instruction to division
 *                                     by repeated subtraction.
 *
 */
    .global set_sample_rate
set_sample_rate:
    MOV     r2, r0                 /* load return value of 0 in case of error */
    BEQ     r4, r0, set_sample_rate_done /* error if argument is 0 */

    MOVHI   r8, %hi(CLK_FREQ)      /* load system clock frequency to */
    ORI     r8, r8, %lo(CLK_FREQ)  /*  find number of system clocks that takes */
    /*DIVU    r9, r8, r4             /*  by dividing the sys clk by the requested rate */
    XOR     r9, r9, r9             /* prepare register for division: r9 is quotient */

div_check:                         /* check if the divisor fits in the dividend */
    BLT     r8, r4, div_done       /* we're done when it doesn't any more */

div_loop:                          /* need to keep subtracting: */
    SUB     r8, r8, r4             /* subtract divisor from dividend */
    ADDI    r9, r9, 1              /* and increment quotient */
    JMPI    div_check              /* thus repeat as needed */

div_done:
    MOVHI   r8, %hi(TRIG_PERIOD_BASE)     /* load period data register address to */
    ORI     r8, r8, %lo(TRIG_PERIOD_BASE) /*  finally save result to trigger period */
    STWIO   r9, (r8)               /*  data, effectively setting the sample rate */

    MOVHI   r8, %hi(TRIG_CTRL_SET) /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)   /* to reset trigger logic */
    MOVI    r9, FIFO_RESET_BIT     /* by sending reset bit high */
    STWIO   r9, (r8)
    ADDI    r8, r8, WORD_SIZE      /* and then move to bit clr reg */
    STWIO   r9, (r8)               /* to send it low */

    MOVI    r2, SIZE_X             /* number of samples acquired is always size of display */

set_sample_rate_done:              /* all done */
    RET                            /* return value is in r2 */
```

```
/*
 *  set_trigger
 *
 *  Description:       This function configures the triggering settings on the sampling
 *                     interface. After execution, triggering will occur as soon as the
 *                     input passes the value of <level>, in the direction indicated by
 *                     <slope>. Note that these settings are only used when a sample is
 *                     started with manual triggering enabled.
 *
 *  Operation:         The procedure first "corrects" the level, mapping it to the
 *                     right range ([0, 255]) and adding any necessary calibration
 *                     constants.
 *                     Then, it writes the slope bit to either the trigger control set
 *                     or clear register, depending on what action needs to be performed,
 *                     followed by the corrected level argument to the trigger level
 *                     register.
 *                     Finally, the reset bit within the trigger control register is
 *                     pulsed to reset the triggering logic.
 *
 *  Arguments:         level - trigger level to be configured, as a value between 0 and
 *                             127, where 0 is the most negative level, and 127 is the
 *                             most positive level (r4).
 *                     slope - desired trigger slope; 1 for positive slope, 0 for
 *                             negative slope (r5).
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r4, r8, r9, r10.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global set_trigger
set_trigger:
    MOVHI   r10, %hi(TRIG_LEVEL_BASE) /* load trigger level register address to update */
    ORI     r10, r10, %lo(TRIG_LEVEL_BASE) /* the desired trigger level */
    MOVI    r9, TRIG_LEVEL_SHIFT   /* shift the passed argument left as needed to */
    SLL     r4, r4, r9             /*  make sure we output a full byte */
    SUBI    r4, r4, CALIBRATION    /* and correct value with calibration data */

    MOVHI   r8, %hi(TRIG_CTRL_CLR) /* load control register bit clear address to */
    ORI     r8, r8, %lo(TRIG_CTRL_CLR) /*  initially assume that we want to set  */
    MOVI    r9, 2                  /*  slope to negative (clear the bit) */
    SLL     r5, r5, r9             /* subtract argument multiplied by word size */
    SUB     r8, r8, r5             /*  effectively moving to set bit register if enabling */
                                   /*  positive slope */

    MOVI    r9, SLOPE_BIT          /* finally write the appropriate bit to the register */
    STWIO   r9, (r8)               /* enabling or disabling the bit as needed */

    STWIO   r4, (r10)              /* and output desired trigger level */

    MOVHI   r8, %hi(TRIG_CTRL_SET) /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)     /* to reset trigger logic */
    MOVI    r9, FIFO_RESET_BIT     /* by sending reset bit high */
    STWIO   r9, (r8)
    ADDI    r8, r8, WORD_SIZE      /* and then move to bit clr reg */
    STWIO   r9, (r8)               /* to send it low */

    RET                            /* all done, so return */


/*
 *  set_delay
 *
 *  Description:       This procedure configures the sampling delay on manual triggers.
 *                     After execution, triggering will occur <delay> samples after the
 *                     configured level and slope settings are satisfied. Note that this
 *                     setting is only used when manual triggering is enabled.
 *                     Also note that delay must be less than MAX_DELAY.
 *
 *  Operation:         The function first corrects the argument by adding the necessary
 *                     hardware constant to it, and then outputs it to the trigger
 *                     delay register.
 *                     Finally, the reset bit within the trigger control register is
 *                     pulsed to reset the triggering logic.
 *
 *  Arguments:         delay - unsigned integer <= MAX_DELAY; trigger delay from
 *                             trigger event in number of samples (r4).
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  None.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       Only positive delays less than or equal to MAX_DELAY are valid.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r4, r10.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global set_delay
set_delay:
    MOVHI   r10, %hi(TRIG_DELAY_BASE) /* load trigger delay register address to update */
    ORI     r10, r10, %lo(TRIG_DELAY_BASE) /* the desired delay time */
    ADDI    r4, r4, DELAY_CONSTANT    /* add delay constant to correct argument */
    STWIO   r4, (r10)                 /* and output to delay register, effectively */
                                      /* configuring delay */

    MOVHI   r8, %hi(TRIG_CTRL_SET)    /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)     /* to reset trigger logic */
    MOVI    r9, FIFO_RESET_BIT        /* by sending reset bit high */
    STWIO   r9, (r8)
    ADDI    r8, r8, WORD_SIZE         /* and then move to bit clr reg */
    STWIO   r9, (r8)                  /* to send it low */

    RET                               /* all done, so return */


/*
 *  start_sample
 *
 *  Description:       This procedure immediately starts sampling data. If the argument
 *                     is FALSE, sampling starts upon a trigger event. If the argument
 *                     is TRUE, sampling starts immediately.
 *                     Any previously started but incomplete samples are cancelled and
 *                      replaced.
 *
 *  Operation:          The procedure sets or clears the auto trigger bit in the trigger
 *                      control register to enable or disable auto triggering.
 *                      Finally, it starts the sample by enabling writing to the FIFO
 *                      through the write enable bit in the control register, and resets
 *                      the triggering logic.
 *
 *  Arguments:          auto_trigger - TRUE if sampling should be started
 *                                     automatically (i.e. as soon as possible),
 *                                     FALSE if it should be started on a trigger
 *                                     event (r4).
 *
 *  Return Value:       None.
 *
 *  Local Variables:    None.
 *
 *  Shared Variables:   None.
 *
 *  Global Variables:   None.
 *
 *  Input:              None.
 *
 *  Output:             None.
 *
 *  Error Handling:     None.
 *
 *  Limitations:        None.
 *
 *  Algorithms:         None.
 *  Data Structures:    None.
 *
 *  Registers Changed: r8, r9.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global start_sample
start_sample:

    MOVHI   r8, %hi(TRIG_CTRL_CLR) /* load trigger control bit clear reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_CLR) /* assuming we'll clear auto trigger bit */
    MOVI    r9, 2                  /* subtract argument multiplied by word size */
    SLL     r4, r4, r9             /* effectively moving to set bit register if enabling */
    SUB     r8, r8, r4             /*  auto trigger*/

    MOVI    r9, AUTO_TRIG_BIT      /* store auto trigger bit in configured register */
    STWIO   r9, (r8)               /* enabling or disabling it as needed */

    MOVHI   r8, %hi(TRIG_CTRL_SET)    /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)     /* to reset trigger logic */
    MOVI    r9, FIFO_RESET_BIT        /* by sending reset bit high */
    STWIO   r9, (r8)
    ADDI    r8, r8, WORD_SIZE         /* and then move to bit clr reg */
    STWIO   r9, (r8)                  /* to send it low */

    MOVHI   r8, %hi(TRIG_CTRL_CLR) /* load trigger control bit clear reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_CLR) /* to clear fifo write enable (make active) */
    MOVI    r9, FIFO_WE_BIT        /* which allows the fifo to be filled with samples */
    STWIO   r9, (r8)               /* effectively starting a sample */

start_sample_done:
    RET                            /* all done, so return */


/*
 *  sample_done
 *
 *  Description:        This function checks whether the started sample was completed.
 *                      If the sample was completed, a pointer to the buffer containing the
 *                      sampled data is provided. If the sample was not completed, a NULL
 *                      pointer is returned.
 *                      Note that this function returns a non-NULL pointer once per call to
 *                     start_sample.
 *
 *  Operation:         The function first checks the value of sample_pending to
 *                     ensure that a sample is ready. If no sample is ready, it simply
 *                     returns with NULL in r2.
 *                     Then, it resets the values of the shared variable to indicate that
 *                     a sample was completed.
 *                     Finally, the function clocks the FIFO twice to account for its
 *                     latency, and then reads FIFO_SIZE bytes in a loop, storing them in
 *                     array <samples>. Note that at each iteration, reading is performed
 *                     by bit-banging the FIFO's read clock. Also note that a calibration
 *                     constant is added to each sample to account for the front end's DC
 *                     offset.
 *
 *  Arguments:         None.
 *
 *  Return Value:      *samples - pointer to bytes acquired in sample if any; NULL
 *                                otherwise (r2).
 *
 *  Local Variables:   r13 - pointer to current place in samples array.
 *                     r10 - number of sample currently being copied.
 *
 *  Shared Variables:  - sample_pending: logical value; zero if no sample is pending,
 *                                       non-zero otherwise. Read/Write.
 *
 *  Global Variables:  None.
 *
 *  Input:             Data samples from the FIFO.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   samples - array of size FIFO_SIZE where samples are stored and
 *                               whose pointer is returned.
 *
 *  Registers Changed: r2, r8, r9, r10, r11, r12, r13, r14.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global sample_done
sample_done:
    MOV     r2, r0                 /* assume no sample ready: null pointer return val */
    MOVIA   r8, sample_pending     /* fetch current pending value to see if this call */
    LDB     r9, (r8)               /* should be ignored */
    BEQ     r0, r9, sample_done_done   /*  which is when value is zero */

    MOVIA   r8, sample_pending     /* reset sample_pending to indicate  */
    STB     r0, (r8)               /* no sample is ready for processing */

    MOVHI   r12, %hi(FIFO_DATA_BASE) /* load fifo data register address */
    ORI     r12, r12, %lo(FIFO_DATA_BASE) /* to actually read data from fifo */
    MOVHI   r8, %hi(TRIG_CTRL_SET) /* load ctrl reg set bit addr for */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)    /* for bit banging */
    MOVIA   r13, sample            /* load array address to store samples */
    MOV     r2, r13                /* and also use it as return value (pointer) */
    MOV     r10, r0                /* and start a counter at 0 for looping */
    MOVI    r11, FIFO_SIZE         /* which will stop at FIFO_SIZE */
    MOVI    r9, FIFO_READ_BIT      /* finally load read clk bit for big banging */

                                   /* FIFO has 2 clocks latency */
    STWIO   r9, (r8)               /* send read clock high to output sample */
    ADDI    r8, r8, WORD_SIZE      /* and move to clear register: will send low next time */
    NOP                            /* wait for sample to actually come through */
    STWIO   r9, (r8)               /* send read clock low to prepare for next sample */
    ADDI    r8, r8, NEG_WORD_SIZE  /* and move to set register: will send high next time  */
    NOP                            /* wait for sample to actually come through */

    STWIO   r9, (r8)               /* send read clock high to output sample */
    ADDI    r8, r8, WORD_SIZE      /* and move to clear register: will send low next time */
    NOP                            /* wait for sample to actually come through */
    STWIO   r9, (r8)               /* send read clock low to prepare for next sample */
    ADDI    r8, r8, NEG_WORD_SIZE  /* and move to set register: will send high next time  */
    NOP                            /* wait for sample to actually come through */

get_data:
    STWIO   r9, (r8)               /* send read clock high to output sample */
    ADDI    r8, r8, WORD_SIZE      /* and move to clear register: will send low next time */
    NOP                            /* wait for sample to actually come through */

    LDBIO   r14, (r12)             /* read sample from fifo */
    ADDI    r14, r14, CALIBRATION  /* add calibration constant */
    STBIO   r14, (r13)             /* and store it in the sample array */

    STWIO   r9, (r8)               /* send read clock low to prepare for next sample */
    ADDI    r8, r8, NEG_WORD_SIZE  /* and move to set register: will send high next time  */

    ADDI    r10, r10, 1            /* increment counter */
    ADDI    r13, r13, 1            /* and sample pointer */
    BNE     r10, r11, get_data     /* and keep getting data until we reach end */

sample_done_done:                  /* all done */
    RET                            /* so return with pointer (or NULL) in r2 */


/*
 *  sample_handler
 *
 *  Description:       This function handles FIFO full hardware interrupts, notifying
 *                     the interface that a sample is ready to be read.
 *
 *  Operation:         The function changes the value of shared variable sample_pending
 *                     to indicate that a sample is now ready.
 *                     Then, it disables writing to the FIFO to make sure no data is
 *                     written as the FIFO is emptied.
 *                     Finally, it sends an EOI to reset the interrupt interface.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  - sample_pending: logical value; zero if no sample is pending,
 *                                       non-zero otherwise. Write only.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r8, r9.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global sample_handler
sample_handler:
    MOVIA   r8, sample_pending     /* mark sample_pending as true to indicate  */
    MOVI    r9, TRUE               /* a sample is ready for processing */
    STB     r9, (r8)

    MOVHI   r8, %hi(TRIG_CTRL_SET) /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)  /* to set fifo write enable (make inactive) */
    MOVI    r9, FIFO_WE_BIT        /* which prevents the fifo from being filled again */
    STWIO   r9, (r8)               /* effectively stopping a sample */

    MOVHI   r8, %hi(FIFO_FULL_BASE)/* write to edge capture register */
    ORI     r8, r8, %lo(FIFO_FULL_BASE) /* to send EOI */
    MOVI    r9, FIFO_INT
    STWIO   r9, EDGE_CAP_OF(r8)

    RET                            /* all done, so return */


/*
 *  trigger_init
 *
 *  Description:       This function performs all the necessary initialization of the
 *                     sampling and triggering interface, preparing shared variables
 *                     for use and configuring the triggering logic. It must be called
 *                     before using any of the other provided functions.
 *
 *  Operation:         The procedure first sets the shared variable sample_pending to
 *                     0, indicating that no sample is pending and no sample has been
 *                     started.
 *                     Then, it resets the triggering logic using the reset bit in the
 *                     control register, and configures the default triggering level,
 *                     delay, rate, and other settings.
 *                     Finally, it installs the interrupt handler by sending an EOI,
 *                     using the HAL API alt_ic_isr_register, and enabling interrupts
 *                     in the interrupt mask register.
 *
 *  Arguments:         None.
 *
 *  Return Value:      None.
 *
 *  Local Variables:   None.
 *
 *  Shared Variables:  - sample_pending: logical value; zero if no sample is pending,
 *                                       non-zero otherwise. Write only.
 *
 *  Global Variables:  None.
 *
 *  Input:             None.
 *
 *  Output:            None.
 *
 *  Error Handling:    None.
 *
 *  Limitations:       None.
 *
 *  Algorithms:        None.
 *  Data Structures:   None.
 *
 *  Registers Changed: r4, r5, r6, r7, r8, r9.
 *
 *  Revision History:
 *      5/29/14   Santiago Navonne     Initial revision.
 *      6/01/14   Santiago Navonne     Expanded documentation.
 *
 */
    .global trigger_init
trigger_init:
    MOVIA   r8, sample_pending     /* mark sample_pending as false to indicate  */
    STB     r0, (r8)               /* no sample is ready for processing */

    MOVHI   r8, %hi(TRIG_LEVEL_BASE)     /* load trigger level reg address */
    ORI     r8, r8, %lo(TRIG_LEVEL_BASE) /* to set default value */
    MOVI    r9, TRIG_LEVEL_DEF
    STWIO   r9, (r8)

    MOVHI   r8, %hi(TRIG_DELAY_BASE)     /* load trigger delay reg address */
    ORI     r8, r8, %lo(TRIG_DELAY_BASE) /* to set default value */
    MOVI    r9, TRIG_DELAY_DEF
    STWIO   r9, (r8)

    MOVHI   r8, %hi(TRIG_PERIOD_BASE)    /* load trigger period reg address */
    ORI     r8, r8, %lo(TRIG_PERIOD_BASE)/* to set default value for rate */
    MOVI    r9, TRIG_PERIOD_DEF
    STWIO   r9, (r8)

    MOVHI   r8, %hi(TRIG_CTRL_SET) /* load trigger control bit set reg address */
    ORI     r8, r8, %lo(TRIG_CTRL_SET)   /* to reset trigger logic */
    MOVI    r9, FIFO_RESET_BIT     /* by sending reset bit high */
    STWIO   r9, (r8)

    MOVI    r9, TRIG_CTRL_DEF      /* load default WE, read clock, auto */
    STWIO   r9, (r8)               /* trigger, and slope values */
    ADDI    r8, r8, WORD_SIZE      /* and move to clear register */
    MOVI    r9, FIFO_RESET_BIT     /* to send reset bit low */
    STWIO   r9, (r8)

    MOVHI   r8, %hi(FIFO_FULL_BASE)/* write to edge capture register to send */
    ORI     r8, r8, %lo(FIFO_FULL_BASE)  /* EOI to pending interrupts */
    MOVI    r9, FIFO_INT           /* and to edge capture register to send */
    STWIO   r9, EDGE_CAP_OF(r8)    /* EOI to pending interrupts */


    ADDI    sp, sp, NEG_WORD_SIZE  /* register interrupt handler */
    STW     ra, 0(sp)              /* push return address */
    MOV     r4, r0                 /* argument ic_id is ignored */
    MOVI    r5, FIFO_FULL_IRQ      /* second arg is IRQ num */
    MOVIA   r6, sample_handler     /* third arg is int handler */
    MOV     r7, r0                 /* fourth arg is data struct (null) */
    ADDI    sp, sp, NEG_WORD_SIZE  /* fifth arg goes on stack */
    STW     r0, 0(sp)              /*  and is ignored (so 0) */
    CALL    alt_ic_isr_register    /* finally, call setup function */
    ADDI    sp, sp, WORD_SIZE      /* clean up stack after call */
    LDW     ra, 0(sp)              /* pop return address */
    ADDI    sp, sp, WORD_SIZE

    MOVHI   r8, %hi(FIFO_FULL_BASE)/* write to interrupt mask register */
    ORI     r8, r8, %lo(FIFO_FULL_BASE)  /* to enable interrupts */
    MOVI    r9, FIFO_INT
    STWIO   r9, INTMASK_OF(r8)


    RET                            /* all done, so return */

```

```
/****************************************************************************/
/*                                                                          */
/*                                TRIGGER.H                                 */
/*                   Data Sampling and Triggering Definitions               */
/*                               Include File                               */
/*                       Digital Oscilloscope Project                       */
/*                                 EE/CS 52                                 */
/*                             Santiago Navonne                             */
/*                                                                          */
/****************************************************************************/

/*
   This file contains the constants for the data sampling and triggering
   routines. The file includes hardware constants used to interact with the
   triggering logic; masks used to access hardware registers; PIO register
   offsets; PIO register addresses; and default configuration values.


   Revision History:
      5/30/14  Santiago Navonne  Initial revision.
*/

/* Hardware constants */
#define    CLK_FREQ      38000000    /* System clock frequency in Hz */
#define    FIFO_SIZE     512         /* Size of sample FIFO in words */
#define    TRIG_LEVEL_SHIFT 1        /* Shift trig level left once to convert [0, 127] -> [0, 255] */
#define    CALIBRATION   13          /* DC offset of front end */
#define    DELAY_CONSTANT   1        /* Hardware delay offset */
#define    MAXDELAY      0xFFFFFFFF - 1 - DELAY_CONSTANT
                                     /* Maximum delay must take hardware delay offset into account */

/* Masks */
#define    FIFO_INT      1           /* FIFO interrupt bit */
#define    AUTO_TRIG_BIT 1<<0        /* Auto trigger bit is bit 0 in trigger control register */
#define    SLOPE_BIT     1<<1        /* Slope control bit is bit 1 in trigger control register */
#define    FIFO_WE_BIT   1<<2        /* FIFO write enable bit is bit 2 in trigger control register */
#define    FIFO_READ_BIT 1<<3        /* FIFO read clock bit is bit 3 in trigger control register */
#define    FIFO_RESET_BIT 1<<4       /* FIFO reset bit is bit 4 in trigger control register */

/* PIO register offsets */
#define    EDGE_CAP_OF   3*WORD_SIZE /* Offset of edge capture PIO register */
#define    INTMASK_OF    2*WORD_SIZE /* Offset of interrupt mask PIO register */
#define    SET_OF        4*WORD_SIZE /* Offset of bit set PIO register */
#define    CLR_OF        5*WORD_SIZE /* Offset of bit clear PIO register */

/* PIO offset locations */
#define    TRIG_CTRL_SET TRIG_CTRL_BASE+SET_OF /* Location of trigger control set bit register */
#define    TRIG_CTRL_CLR TRIG_CTRL_BASE+CLR_OF /* Location of trigger control clear bit register */

/* Default values */
#define    TRIG_CTRL_DEF 0b00000111 /* Initialize control register to: low read clock, inactive */
                                    /* (high) write enable, negative slope, auto trigger */
#define    TRIG_DELAY_DEF  0+DELAY_CONSTANT /* Default trigger delay (desired delay + DELAY_CONSTANT) *
#define    TRIG_LEVEL_DEF  128      /* Default trigger level */
#define    DEFAULT_SAMPLE_RATE 19000000     /* Default sample rate */
#define    TRIG_PERIOD_DEF CLK_FREQ/DEFAULT_SAMPLE_RATE  /* Translates into this trigger period */
```

```
/*
 * system.h - SOPC Builder system and BSP software package information
 *
 * Machine generated for CPU 'nios' in SOPC Builder design 'sopc_scope_sys'
 * SOPC Builder design path: C:/Users/tago/Dropbox/OUT/EE52/quartus/sopc_scope_sys.sopcinfo
 *
 * Generated: Wed Jun 11 15:26:36 PDT 2014
 */

/*
 * DO NOT MODIFY THIS FILE
 *
 * Changing this file will have subtle consequences
 * which will almost certainly lead to a nonfunctioning
 * system. If you do modify this file, be aware that your
 * changes will be overwritten and lost when this file
 * is generated again.
 *
 * DO NOT MODIFY THIS FILE
 */

/*
 * License Agreement
 *
 * Copyright (c) 2008
 * Altera Corporation, San Jose, California, USA.
 * All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 * DEALINGS IN THE SOFTWARE.
 *
 * This agreement shall be governed in all respects by the laws of the State
 * of California and by the laws of the United States of America.
 */

#ifndef __SYSTEM_H_
#define __SYSTEM_H_

/* Include definitions from linker script generator */
#include "linker.h"


/*
 * CPU configuration
 *
 */

#define ALT_CPU_ARCHITECTURE "altera_nios2_qsys"
#define ALT_CPU_BIG_ENDIAN 0
#define ALT_CPU_BREAK_ADDR 0x00240820
#define ALT_CPU_CPU_FREQ 50000000u
#define ALT_CPU_CPU_ID_SIZE 1
#define ALT_CPU_CPU_ID_VALUE 0x00000000
#define ALT_CPU_CPU_IMPLEMENTATION "tiny"
#define ALT_CPU_DATA_ADDR_WIDTH 0x16
#define ALT_CPU_DCACHE_LINE_SIZE 0
#define ALT_CPU_DCACHE_LINE_SIZE_LOG2 0
#define ALT_CPU_DCACHE_SIZE 0
#define ALT_CPU_EXCEPTION_ADDR 0x00180020
#define ALT_CPU_FLUSHDA_SUPPORTED
```

```
#define ALT_CPU_FREQ 50000000
#define ALT_CPU_HARDWARE_DIVIDE_PRESENT 0
#define ALT_CPU_HARDWARE_MULTIPLY_PRESENT 0
#define ALT_CPU_HARDWARE_MULX_PRESENT 0
#define ALT_CPU_HAS_DEBUG_CORE 1
#define ALT_CPU_HAS_DEBUG_STUB
#define ALT_CPU_HAS_JMPI_INSTRUCTION
#define ALT_CPU_ICACHE_LINE_SIZE 0
#define ALT_CPU_ICACHE_LINE_SIZE_LOG2 0
#define ALT_CPU_ICACHE_SIZE 0
#define ALT_CPU_INST_ADDR_WIDTH 0x16
#define ALT_CPU_NAME "nios"
#define ALT_CPU_RESET_ADDR 0x00180000


/*
 * CPU configuration (with legacy prefix - don't use these anymore)
 *
 */

#define NIOS2_BIG_ENDIAN 0
#define NIOS2_BREAK_ADDR 0x00240820
#define NIOS2_CPU_FREQ 50000000u
#define NIOS2_CPU_ID_SIZE 1
#define NIOS2_CPU_ID_VALUE 0x00000000
#define NIOS2_CPU_IMPLEMENTATION "tiny"
#define NIOS2_DATA_ADDR_WIDTH 0x16
#define NIOS2_DCACHE_LINE_SIZE 0
#define NIOS2_DCACHE_LINE_SIZE_LOG2 0
#define NIOS2_DCACHE_SIZE 0
#define NIOS2_EXCEPTION_ADDR 0x00180020
#define NIOS2_FLUSHDA_SUPPORTED
#define NIOS2_HARDWARE_DIVIDE_PRESENT 0
#define NIOS2_HARDWARE_MULTIPLY_PRESENT 0
#define NIOS2_HARDWARE_MULX_PRESENT 0
#define NIOS2_HAS_DEBUG_CORE 1
#define NIOS2_HAS_DEBUG_STUB
#define NIOS2_HAS_JMPI_INSTRUCTION
#define NIOS2_ICACHE_LINE_SIZE 0
#define NIOS2_ICACHE_LINE_SIZE_LOG2 0
#define NIOS2_ICACHE_SIZE 0
#define NIOS2_INST_ADDR_WIDTH 0x16
#define NIOS2_RESET_ADDR 0x00180000


/*
 * Define for each module class mastered by the CPU
 *
 */

#define __ALTERA_AVALON_JTAG_UART
#define __ALTERA_AVALON_PIO
#define __ALTERA_GENERIC_TRISTATE_CONTROLLER
#define __ALTERA_NIOS2_QSYS


/*
 * System configuration
 *
 */

#define ALT_DEVICE_FAMILY "Cyclone III"
#define ALT_ENHANCED_INTERRUPT_API_PRESENT
#define ALT_IRQ_BASE NULL
#define ALT_LOG_PORT "/dev/null"
#define ALT_LOG_PORT_BASE 0x0
#define ALT_LOG_PORT_DEV null
#define ALT_LOG_PORT_TYPE ""
#define ALT_NUM_EXTERNAL_INTERRUPT_CONTROLLERS 0
#define ALT_NUM_INTERNAL_INTERRUPT_CONTROLLERS 1
#define ALT_NUM_INTERRUPT_CONTROLLERS 1
#define ALT_STDERR "/dev/jtag"
#define ALT_STDERR_BASE 0x241180
#define ALT_STDERR_DEV jtag
#define ALT_STDERR_IS_JTAG_UART
```

```
#define ALT_STDERR_PRESENT
#define ALT_STDERR_TYPE "altera_avalon_jtag_uart"
#define ALT_STDIN "/dev/jtag"
#define ALT_STDIN_BASE 0x241180
#define ALT_STDIN_DEV jtag
#define ALT_STDIN_IS_JTAG_UART
#define ALT_STDIN_PRESENT
#define ALT_STDIN_TYPE "altera_avalon_jtag_uart"
#define ALT_STDOUT "/dev/jtag"
#define ALT_STDOUT_BASE 0x241180
#define ALT_STDOUT_DEV jtag
#define ALT_STDOUT_IS_JTAG_UART
#define ALT_STDOUT_PRESENT
#define ALT_STDOUT_TYPE "altera_avalon_jtag_uart"
#define ALT_SYSTEM_NAME "sopc_scope_sys"


/*
 * fifo_data configuration
 *
 */

#define ALT_MODULE_CLASS_fifo_data altera_avalon_pio
#define FIFO_DATA_BASE 0x241140
#define FIFO_DATA_BIT_CLEARING_EDGE_REGISTER 0
#define FIFO_DATA_BIT_MODIFYING_OUTPUT_REGISTER 0
#define FIFO_DATA_CAPTURE 0
#define FIFO_DATA_DATA_WIDTH 8
#define FIFO_DATA_DO_TEST_BENCH_WIRING 0
#define FIFO_DATA_DRIVEN_SIM_VALUE 0
#define FIFO_DATA_EDGE_TYPE "NONE"
#define FIFO_DATA_FREQ 50000000
#define FIFO_DATA_HAS_IN 1
#define FIFO_DATA_HAS_OUT 0
#define FIFO_DATA_HAS_TRI 0
#define FIFO_DATA_IRQ -1
#define FIFO_DATA_IRQ_INTERRUPT_CONTROLLER_ID -1
#define FIFO_DATA_IRQ_TYPE "NONE"
#define FIFO_DATA_NAME "/dev/fifo_data"
#define FIFO_DATA_RESET_VALUE 0
#define FIFO_DATA_SPAN 16
#define FIFO_DATA_TYPE "altera_avalon_pio"


/*
 * fifo_full configuration
 *
 */

#define ALT_MODULE_CLASS_fifo_full altera_avalon_pio
#define FIFO_FULL_BASE 0x241130
#define FIFO_FULL_BIT_CLEARING_EDGE_REGISTER 0
#define FIFO_FULL_BIT_MODIFYING_OUTPUT_REGISTER 0
#define FIFO_FULL_CAPTURE 1
#define FIFO_FULL_DATA_WIDTH 1
#define FIFO_FULL_DO_TEST_BENCH_WIRING 0
#define FIFO_FULL_DRIVEN_SIM_VALUE 0
#define FIFO_FULL_EDGE_TYPE "RISING"
#define FIFO_FULL_FREQ 50000000
#define FIFO_FULL_HAS_IN 1
#define FIFO_FULL_HAS_OUT 0
#define FIFO_FULL_HAS_TRI 0
#define FIFO_FULL_IRQ 4
#define FIFO_FULL_IRQ_INTERRUPT_CONTROLLER_ID 0
#define FIFO_FULL_IRQ_TYPE "EDGE"
#define FIFO_FULL_NAME "/dev/fifo_full"
#define FIFO_FULL_RESET_VALUE 0
#define FIFO_FULL_SPAN 16
#define FIFO_FULL_TYPE "altera_avalon_pio"


/*
 * hal configuration
 *
 */
```

```

#define ALT_MAX_FD 32
#define ALT_SYS_CLK none
#define ALT_TIMESTAMP_CLK none


/*
 * jtag configuration
 *
 */

#define ALT_MODULE_CLASS_jtag altera_avalon_jtag_uart
#define JTAG_BASE 0x241180
#define JTAG_IRQ 0
#define JTAG_IRQ_INTERRUPT_CONTROLLER_ID 0
#define JTAG_NAME "/dev/jtag"
#define JTAG_READ_DEPTH 64
#define JTAG_READ_THRESHOLD 8
#define JTAG_SPAN 8
#define JTAG_TYPE "altera_avalon_jtag_uart"
#define JTAG_WRITE_DEPTH 64
#define JTAG_WRITE_THRESHOLD 8


/*
 * pio_0 configuration
 *
 */

#define ALT_MODULE_CLASS_pio_0 altera_avalon_pio
#define PIO_0_BASE 0x2410a0
#define PIO_0_BIT_CLEARING_EDGE_REGISTER 1
#define PIO_0_BIT_MODIFYING_OUTPUT_REGISTER 1
#define PIO_0_CAPTURE 1
#define PIO_0_DATA_WIDTH 6
#define PIO_0_DO_TEST_BENCH_WIRING 0
#define PIO_0_DRIVEN_SIM_VALUE 0
#define PIO_0_EDGE_TYPE "FALLING"
#define PIO_0_FREQ 50000000
#define PIO_0_HAS_IN 1
#define PIO_0_HAS_OUT 0
#define PIO_0_HAS_TRI 0
#define PIO_0_IRQ 1
#define PIO_0_IRQ_INTERRUPT_CONTROLLER_ID 0
#define PIO_0_IRQ_TYPE "EDGE"
#define PIO_0_NAME "/dev/pio_0"
#define PIO_0_RESET_VALUE 0
#define PIO_0_SPAN 32
#define PIO_0_TYPE "altera_avalon_pio"


/*
 * ram configuration
 *
 */

#define ALT_MODULE_CLASS_ram altera_generic_tristate_controller
#define RAM_BASE 0x220000
#define RAM_IRQ -1
#define RAM_IRQ_INTERRUPT_CONTROLLER_ID -1
#define RAM_NAME "/dev/ram"
#define RAM_SPAN 131072
#define RAM_TYPE "altera_generic_tristate_controller"


/*
 * rom configuration
 *
 */

#define ALT_MODULE_CLASS_rom altera_generic_tristate_controller
#define ROM_BASE 0x180000
#define ROM_IRQ -1
#define ROM_IRQ_INTERRUPT_CONTROLLER_ID -1
#define ROM_NAME "/dev/rom"
```

```
#define ROM_SPAN 524288
#define ROM_TYPE "altera_generic_tristate_controller"


/*
 * trig_ctrl configuration
 *
 */

#define ALT_MODULE_CLASS_trig_ctrl altera_avalon_pio
#define TRIG_CTRL_BASE 0x241060
#define TRIG_CTRL_BIT_CLEARING_EDGE_REGISTER 0
#define TRIG_CTRL_BIT_MODIFYING_OUTPUT_REGISTER 1
#define TRIG_CTRL_CAPTURE 0
#define TRIG_CTRL_DATA_WIDTH 5
#define TRIG_CTRL_DO_TEST_BENCH_WIRING 0
#define TRIG_CTRL_DRIVEN_SIM_VALUE 0
#define TRIG_CTRL_EDGE_TYPE "NONE"
#define TRIG_CTRL_FREQ 50000000
#define TRIG_CTRL_HAS_IN 0
#define TRIG_CTRL_HAS_OUT 1
#define TRIG_CTRL_HAS_TRI 0
#define TRIG_CTRL_IRQ -1
#define TRIG_CTRL_IRQ_INTERRUPT_CONTROLLER_ID -1
#define TRIG_CTRL_IRQ_TYPE "NONE"
#define TRIG_CTRL_NAME "/dev/trig_ctrl"
#define TRIG_CTRL_RESET_VALUE 3
#define TRIG_CTRL_SPAN 32
#define TRIG_CTRL_TYPE "altera_avalon_pio"


/*
 * trig_delay configuration
 *
 */

#define ALT_MODULE_CLASS_trig_delay altera_avalon_pio
#define TRIG_DELAY_BASE 0x241120
#define TRIG_DELAY_BIT_CLEARING_EDGE_REGISTER 0
#define TRIG_DELAY_BIT_MODIFYING_OUTPUT_REGISTER 0
#define TRIG_DELAY_CAPTURE 0
#define TRIG_DELAY_DATA_WIDTH 32
#define TRIG_DELAY_DO_TEST_BENCH_WIRING 0
#define TRIG_DELAY_DRIVEN_SIM_VALUE 0
#define TRIG_DELAY_EDGE_TYPE "NONE"
#define TRIG_DELAY_FREQ 50000000
#define TRIG_DELAY_HAS_IN 0
#define TRIG_DELAY_HAS_OUT 1
#define TRIG_DELAY_HAS_TRI 0
#define TRIG_DELAY_IRQ -1
#define TRIG_DELAY_IRQ_INTERRUPT_CONTROLLER_ID -1
#define TRIG_DELAY_IRQ_TYPE "NONE"
#define TRIG_DELAY_NAME "/dev/trig_delay"
#define TRIG_DELAY_RESET_VALUE 1
#define TRIG_DELAY_SPAN 16
#define TRIG_DELAY_TYPE "altera_avalon_pio"


/*
 * trig_level configuration
 *
 */

#define ALT_MODULE_CLASS_trig_level altera_avalon_pio
#define TRIG_LEVEL_BASE 0x241150
#define TRIG_LEVEL_BIT_CLEARING_EDGE_REGISTER 0
#define TRIG_LEVEL_BIT_MODIFYING_OUTPUT_REGISTER 0
#define TRIG_LEVEL_CAPTURE 0
#define TRIG_LEVEL_DATA_WIDTH 8
#define TRIG_LEVEL_DO_TEST_BENCH_WIRING 0
#define TRIG_LEVEL_DRIVEN_SIM_VALUE 0
#define TRIG_LEVEL_EDGE_TYPE "NONE"
#define TRIG_LEVEL_FREQ 50000000
#define TRIG_LEVEL_HAS_IN 0
#define TRIG_LEVEL_HAS_OUT 1
```

```
#define TRIG_LEVEL_HAS_TRI 0
#define TRIG_LEVEL_IRQ -1
#define TRIG_LEVEL_IRQ_INTERRUPT_CONTROLLER_ID -1
#define TRIG_LEVEL_IRQ_TYPE "NONE"
#define TRIG_LEVEL_NAME "/dev/trig_level"
#define TRIG_LEVEL_RESET_VALUE 0
#define TRIG_LEVEL_SPAN 16
#define TRIG_LEVEL_TYPE "altera_avalon_pio"


/*
 * trig_period configuration
 *
 */

#define ALT_MODULE_CLASS_trig_period altera_avalon_pio
#define TRIG_PERIOD_BASE 0x241160
#define TRIG_PERIOD_BIT_CLEARING_EDGE_REGISTER 0
#define TRIG_PERIOD_BIT_MODIFYING_OUTPUT_REGISTER 0
#define TRIG_PERIOD_CAPTURE 0
#define TRIG_PERIOD_DATA_WIDTH 32
#define TRIG_PERIOD_DO_TEST_BENCH_WIRING 0
#define TRIG_PERIOD_DRIVEN_SIM_VALUE 0
#define TRIG_PERIOD_EDGE_TYPE "NONE"
#define TRIG_PERIOD_FREQ 50000000
#define TRIG_PERIOD_HAS_IN 0
#define TRIG_PERIOD_HAS_OUT 1
#define TRIG_PERIOD_HAS_TRI 0
#define TRIG_PERIOD_IRQ -1
#define TRIG_PERIOD_IRQ_INTERRUPT_CONTROLLER_ID -1
#define TRIG_PERIOD_IRQ_TYPE "NONE"
#define TRIG_PERIOD_NAME "/dev/trig_period"
#define TRIG_PERIOD_RESET_VALUE 1
#define TRIG_PERIOD_SPAN 16
#define TRIG_PERIOD_TYPE "altera_avalon_pio"


/*
 * vram configuration
 *
 */

#define ALT_MODULE_CLASS_vram altera_generic_tristate_controller
#define VRAM_BASE 0x0
#define VRAM_IRQ -1
#define VRAM_IRQ_INTERRUPT_CONTROLLER_ID -1
#define VRAM_NAME "/dev/vram"
#define VRAM_SPAN 1048576
#define VRAM_TYPE "altera_generic_tristate_controller"

#endif /* __SYSTEM_H_ */

```