ORNL/LTR-2014/424

Radiation-Hardened Circuitry Using Mask-Programmable Analog Arrays

C. L. Britton
J. Shelton
M. N. Ericson
M. Bobrek
B. Blalock

Approved for public release:
distribution is unlimited.

September 2014

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

CONTENTS

Page

LIST OF FIGURES v
ACRONYMS vii
ABSTRACT 1
1. PROJECT STATUS 1
2. FUNCTIONAL BLOCK DIAGRAM 1
3. HIGH-LEVEL ELECTRICAL BLOCK DIAGRAM 2
4. SYSTEM DEVELOPMENT 3
4.1 RAD-HARD BOARD 4
4.1.1 Voltage regulator and frequency synthesizer 6
4.1.2 ASIC on the RHB 6
4.2 SENSOR BOARD 7
4.3 INTERFACE BOARD 9
4.4 NEXYS 3 BOARD FIRMWARE/COMPUTER INTERFACE 10
4.5 LABVIEW INTERFACE PROGRAM 13
4.6 PRELIMINARY TEST DATA 14
5. CONCLUSION 16
6. REFERENCES 17

LIST OF FIGURES

Figures **Page**

Fig. 1. System functional description. 2
Fig. 3. System partition and function. 4
Fig. 4. Detailed radiation-hardened block diagram. 5
Fig. 5. The completed rad-hard board. 7
Fig. 6. Detailed sensor board block diagram. 8
Fig. 7. The Sensor board with insulation coatings. 9
Fig. 8. Detailed interface board data flow block diagram. 10
Fig. 9. The interface board (right) connected to the Nexys 3 board. 11
Fig. 10. Decimation filter mathematical structure. The "N" represents the filter order, and the "R" is the filter decimation ratio. In this application N = 3, and R = 256. 12
Fig. 11. Decimation filter frequency response. 13
Fig. 12. Screen shot of the LabView interface program. 14
Fig. 13. ADC preliminary testing data. 15
Fig. A1. Top-level instantiation of the VHDL code. 18

ACRONYMS

ADC	analog-digital converter
ASIC	application-specific integrated circuit
CIC	cascade, integrate and comb
FPGA	field-programmable gate array
G-M	Geiger-Mueller
IB	interface board
IC	integrated circuit
NEET	Nuclear Energy Enabling Technologies
PC	personal computer
PCB	printed circuit board
RHB	rad-hard board
UART	universal asynchronous receive transmitter
USB	universal serial bus
VCA	Via-Configured Array

ABSTRACT

As the recent accident at Fukushima Daiichi so vividly demonstrated, telerobotic technologies capable of withstanding high radiation environments need to be readily available to enable operations, repair, and recovery under severe accident scenarios when human entry is extremely dangerous or not possible. Telerobotic technologies that enable remote operation in high dose rate environments have undergone revolutionary improvement over the past few decades. However, much of this technology cannot be employed in nuclear power environments because of the radiation sensitivity of the electronics and the organic insulator materials currently in use.

This is a report of the activities involving Task 2 of the Nuclear Energy Enabling Technologies (NEET) 2 project "Radiation Hardened Circuitry Using Mask-Programmable Analog Arrays" [1]. Using the analog blocks available in our currently preconfigured via-configured array (VCA), we will perform a detailed schematic design of our system to include the signal-processing blocks for temperature, radiation and pressure. Control and data acquisition will be implemented with the Spartan-6 field-programmable gate array (FPGA), as well as with wired serial communications using a remote computer. In addition, batteries and associated voltage regulators will be selected for powering the system. Fabrication may include a polyimide, printed circuit board (PCB) for improved radiation and temperature tolerance. The PCB will go through a layout process by one of the organizations we use for this function and be fabricated by another external vendor. Population of the system board will be performed by one of our in-house technicians or by an outside vendor we commonly use. Five prototype systems [each system consisting of sensors, electronics board, battery power supply, and personal computer serial communications port] will be constructed to support the testing objectives of this work. The submitters and the vendors will perform quality assurance at each step.

1. PROJECT STATUS

We are currently on schedule or slightly ahead of schedule. Our published schedule shown in the project plan [1] states that fabrication of all five systems should be complete by the middle of the first quarter of FY 2015, and we expect to be ready on or before that time. We presently have the complete system operating end-to-end and are able to take data. In addition, we currently have two fully-functioning systems and are having the remaining boards fabricated to complete the five plus spares. We will be refining the personal computer–based data-acquisition software to simplify data presentation.

2. FUNCTIONAL BLOCK DIAGRAM

The final goal of the data acquisition system is to provide intelligible data to the end users so that they can assess a reactor environment operating condition. In order to transform the

output signals from the environmental sensors, a few steps must be taken. First, each sensor will output a specific type of signal, and these signals must be converted to a digital structure so that a computer processor or controller can process and relay this information to the user in a useful format, as depicted in Fig. 1. The conversion methods for each signal will differ slightly, and those variations can be considered on a lower sublevel. The controller will convert the data into a format that the user will find easy to understand (for example, temperature, pressure, and radiation count in standard units). Ultimately, each conversion will result in a binary output value, or DN, that the controller will prepare and display for the user.

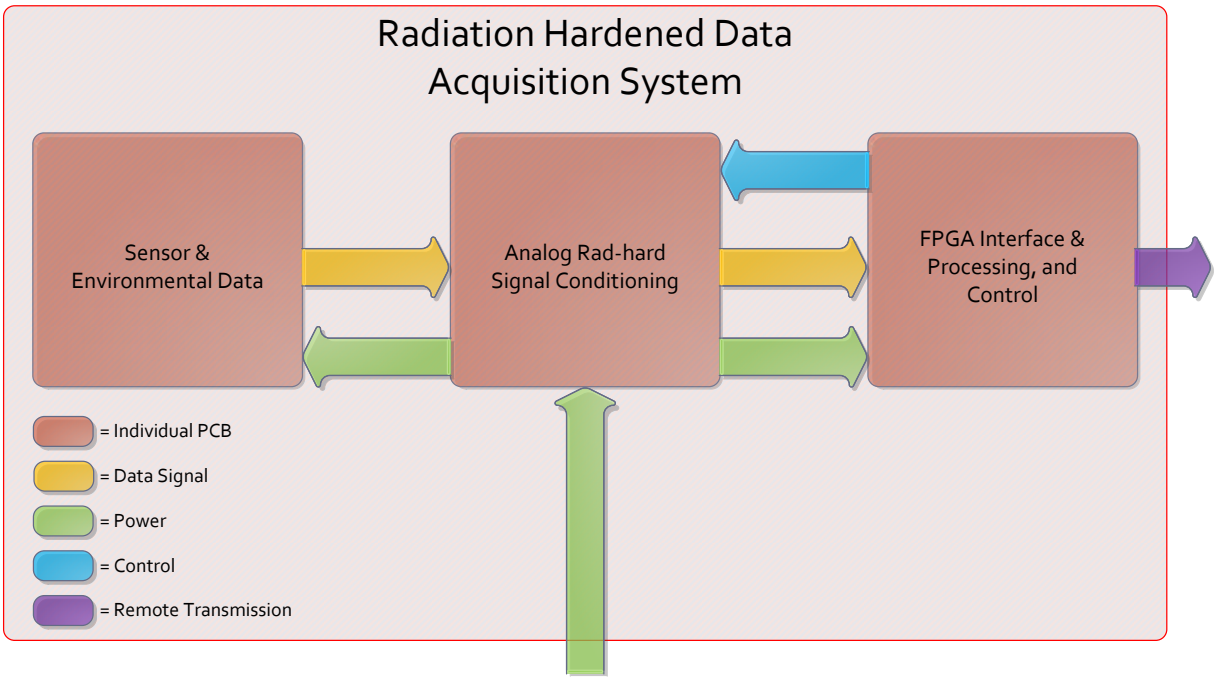


Fig. 1. System functional description.

3. HIGH-LEVEL ELECTRICAL BLOCK DIAGRAM

The high-level block diagram, shown in Fig. 2, illustrates the electrical signal-processing paths for three distinct types of sensors. Each sensor will require some level of input gain and filtering and an analog-digital converter (ADC) to convert the signals to a digital format. The sensor and electronics signal flow will be presented in the following sections. The three channels of signal conditioning/signal processing will be implemented using the circuitry present on the Triad via-configured array (VCA). Each VCA application-specific integrated circuit (ASIC) contains multiple single-ended operational amplifiers, biquad filters designed as input anti-aliasing filters for the sigma-delta modulators which are also on the ASIC, and a bandgap voltage reference.

ASICs are very important to electronic systems for multiple reasons. First, integrated circuits (ICs) have the inherent capability of performing numerous electrical functions and operations within one area efficient space. Not only do ICs optimize circuit density, but they

also offer improved matching behavior performance when compared to using individual functional blocks to accomplish the same operation. This is because the IC fabrication process, although precise, is not perfect, and silicon substrate variations from chip to chip are inevitable. Combining multiple functions into one IC effectively reduces matching errors and provides minimal variation from expected results.

Secondly, as their name implies, ASICs are application specific, which means they can be designed to perform precisely to the exact requirements of an application. On the other hand, general ICs must be designed to satisfy a wide variety of applications, and nothing comes free with circuit design. Tradeoffs must be made; thus certain performance characteristics of the IC will diminish relative to an ASIC. This advantage over general ICs makes ASICs almost necessary, especially when stringent conditions such as extreme environment operation is required.

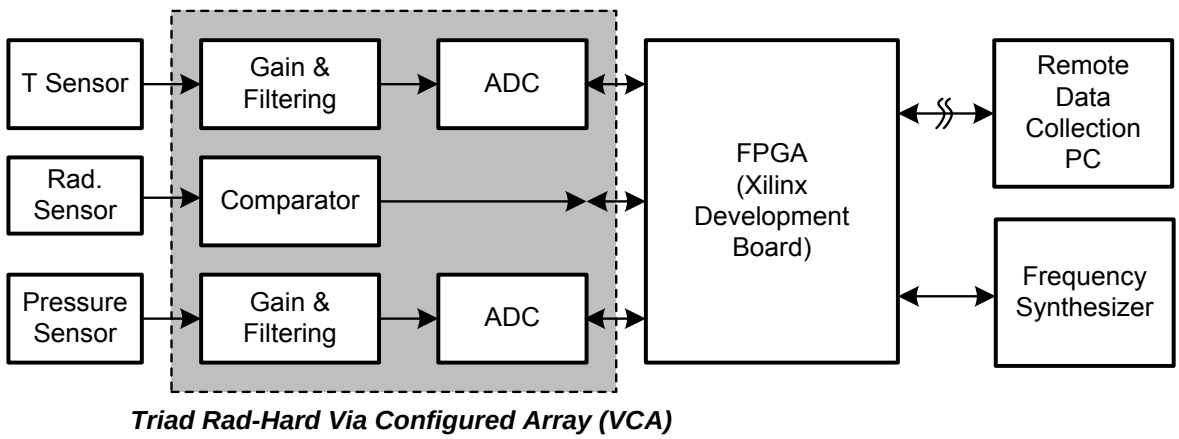


Fig. 2. Data acquisition system based on commercial and near-commercial rad-hard circuits.

4. SYSTEM DEVELOPMENT

The system was developed and partitioned as shown in Fig. 3. The boards labeled sensor board, rad-hard board, and interface board were designed at Oak Ridge National Laboratory (ORNL). The Nexys 3 board is the commercial board developed and sold by Digilent, Inc., that contains the Spartan 6 field-programmable gate array (FPGA), along with the communications hardware interface. Of all the boards shown, only the rad-hard board (the RHB) has hardened components. The other boards were not designed to be exposed to radiation. The system, with the exception of the commercial Nexys 3 board, is designed to operate off 9–12 V DC so that either NiMH batteries or lithium batteries can be used. The Nexys 3 board actually uses a 110 V to 5 V DC universal adapter and was not redesigned to be used with a battery, but this could be done if required. If, after surveying the radiation test facilities, it is determined that a battery for

this board is necessary, we will provide one. The board developments and associated software will be individually presented. Schematics of each board can be found in Appendix B.

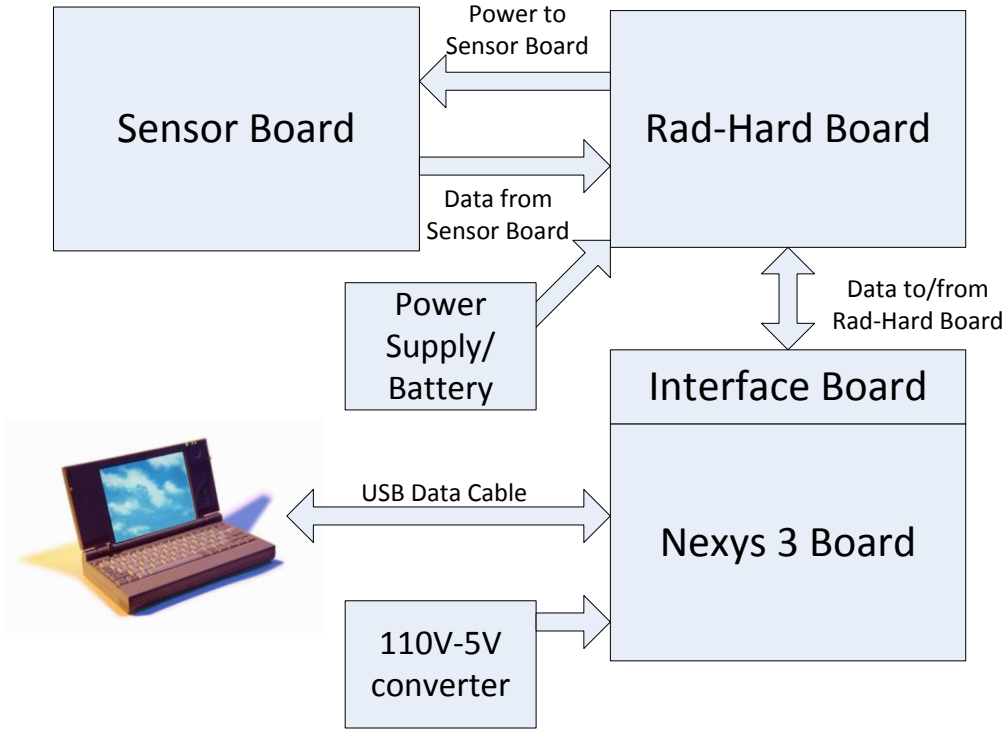


Fig. 3. System partition and function.

4.1 RAD-HARD BOARD

The RHB consists of the Triad ASIC; the FMI synthesizer ASIC; and various passive resistors, capacitors, and connectors used for board-board or board-power connections. As previously mentioned, the RHB implements the interface functions between the sensors and the ADCs, and the ADCs themselves. It also contains the rad-hard voltage regulators which supply 5 V and 3.3 V to the on-board circuitry.

The temperature, pressure, and gamma radiation sensors will supply voltage waveforms to radiation-hardened analog circuits on the Triad chip to begin analog-to-digital conversion. Since these measurements are inherently low frequency in nature, low-frequency single-ended op amps, and thus low-frequency single-ended sigma-delta modulators will be chosen from the Triad chip selection in order to increase noise filtering and reduce power consumption.

Both the temperature and pressure sensor analog data will undergo sigma-delta modulation to generate a digital pulse density modulated bit stream. This bit stream will then be converted into a digital number (DN) through proper filtering. The sigma-delta modulator works by increasing the output bit stream pulse density as the input analog waveform voltage increases, and decreasing the bit stream pulse density as the input analog waveform voltage decreases. Given a stable loop, the output bit streams are passed to the FPGA digital sinc filter for final translation into a DN, as seen in Fig. 4. Different values of external offsets for sigma-delta

modulation are needed for their respective temperature and pressure data, considering these waveforms exhibit differing peak voltages and DC offsets.

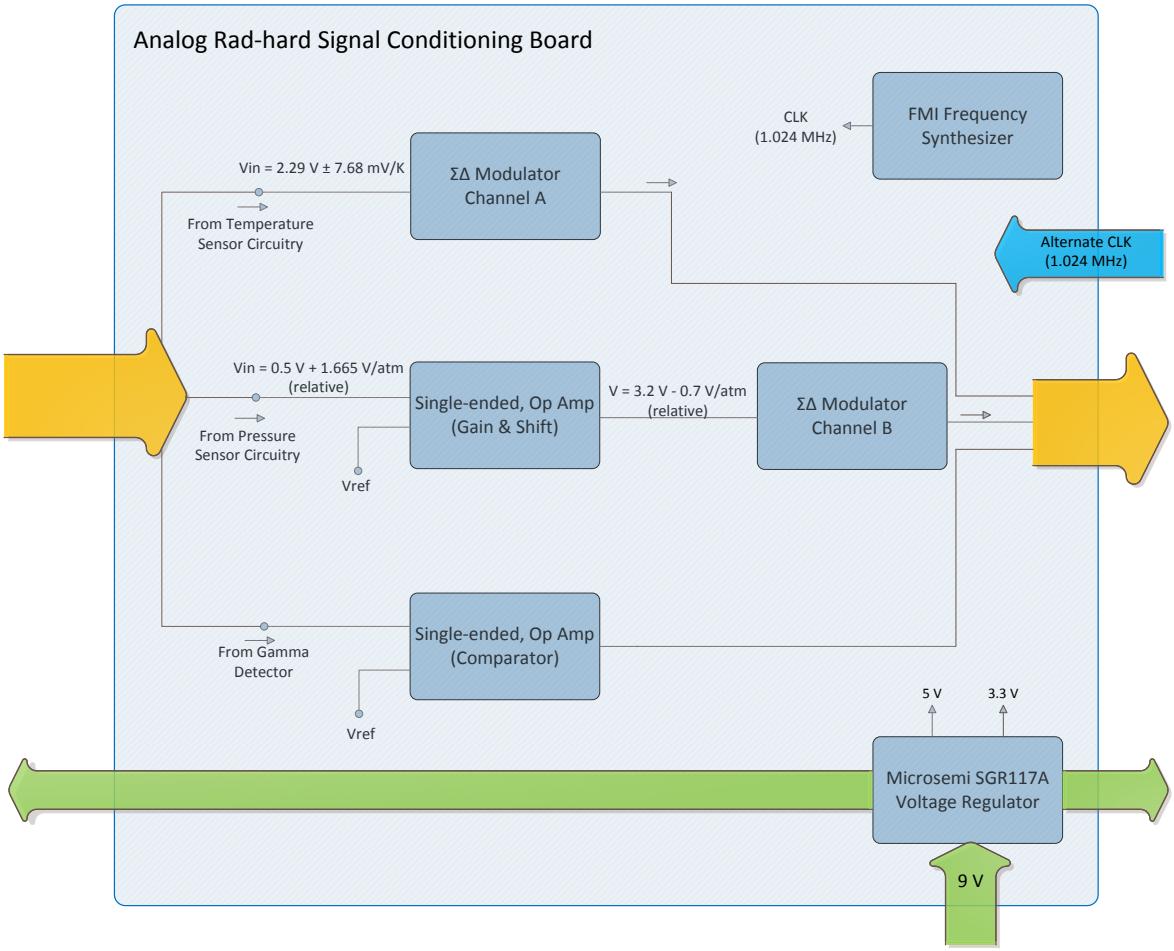


Fig. 4. Detailed radiation-hardened block diagram.

Digitizing radiation events is much different from digitizing analog waveforms, in that the charge generated by the event is only present for a short period of time, and it needs to be classified. The current spike output from the Geiger-Mueller (G-M) tube is converted to voltage through a resistor, and depending on the peak level of current, the voltage will also reach a certain peak value. In order to discern whether a voltage spike is in fact a radiation event, a certain voltage threshold must be met. This can be achieved by comparing the voltage spike with a reference voltage that is unachievable by any excitation other than a radiation event. If the voltage spike surpasses this threshold, the comparator will output a digital 'high' value, representing one event, and subsequently return to a digital low in preparation for the next event. These events can be summed with an FPGA digitally implemented counter, which will notify users of the total ionizing dose, as well as the dose rate, which is derived through the rate of change of the event count. Because there is no comparator circuit on the Triad chip, the output

of an op amp will need to be buffered so as to replicate a comparator and produce a full-swing digital output that can be counted.

4.1.1 Voltage regulator and frequency synthesizer

Two other components will be present on the radiation-hardened printed circuit board: the Microsemi voltage regulator and the FMI frequency synthesizer. The rad-hard voltage regulator, which is radiation-hardened beyond 1 Mrad total integrated dose (TID), will supply the voltage and current necessary for all circuits to operate. It will be powered by an external battery. We have designed the board to be used with battery packs from 9 V to 12 V, which encompasses chemistries of NiCd, NiMH, lithium ion, and lead-acid. We presently have NiMH batteries ordered for this application since they are easily chargeable and readily available. Multiple copies of the voltage regulator will be present for circuits with varying voltage requirements. The rad-hard frequency synthesizer is capable of precise operation to at least 300krad TID and will serve as the clock generator for the FPGA controller. It is mounted on a so-called interposer board, which allows quick changes of the FMI ASIC.

4.1.2 ASIC on the RHB

Some additional components are necessary to provide the power required by the system and enable full circuit operation. All circuits within the system will need specific levels of voltage and current supply generated from a voltage regulator, and for radiation rich environments, this supply must be very robust across a large range of dose rates, potentially up to 200 krad/h. Microsemi Corporation produces a variety of radiation-hardened voltage regulators with differing total dose capabilities, but the SGR117A model stood out with the highest TID capability of all, claiming total doses exceeding 1 Mrad. This voltage regulator can produce output voltages ranging from 1.25 to 34 V, supply at least 1.5 A of current across all operating conditions, and maintain a 0.3% load regulation specification. The SGR117A is available in a 3-pin K package with a thermal dependence of only 3ºC/W and operating temperatures up to 150ºC. These voltage regulator specifications can satisfy power requirements for every circuit within the system, with the exception of the G-M tube which will require a high-voltage DC converter to reach a 500 V potential. A photograph of the RHB is shown in Fig. 5.

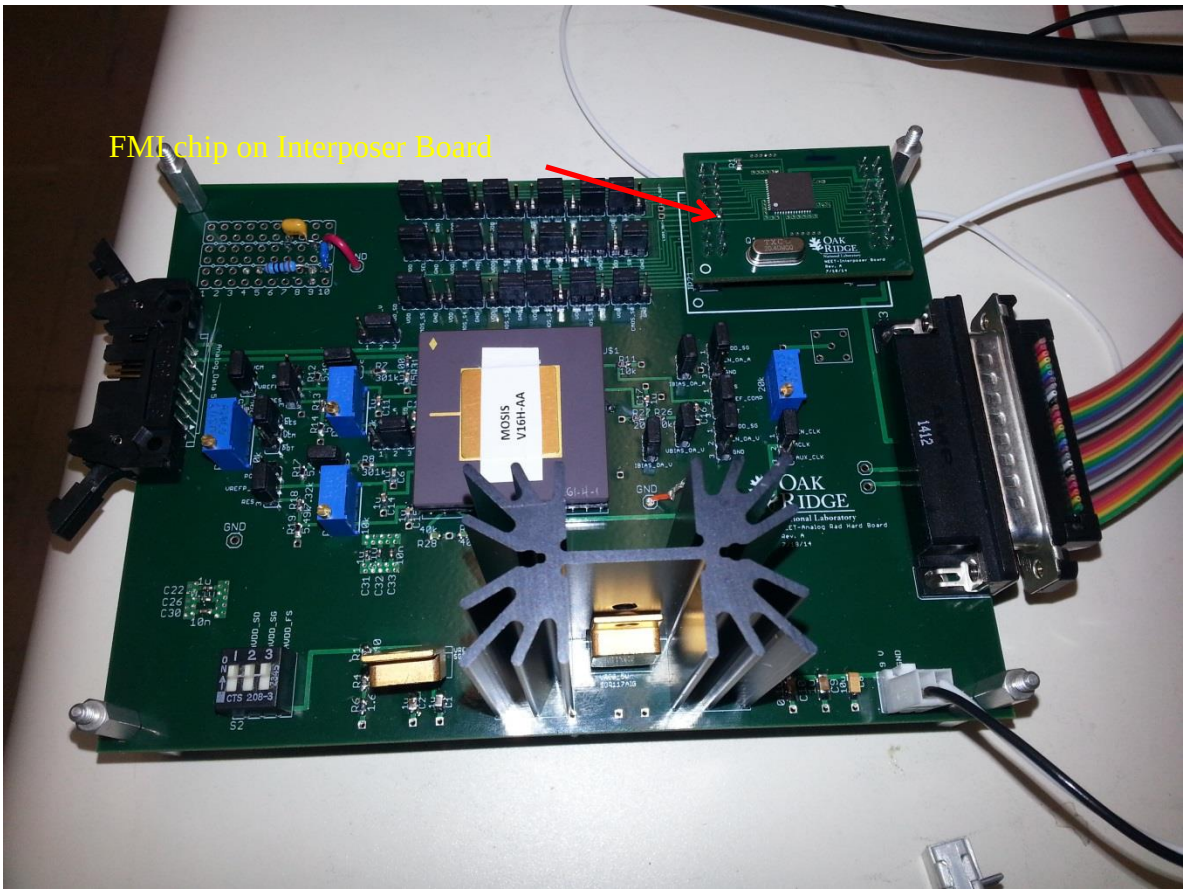


Fig. 5. The completed rad-hard board.

4.2 SENSOR BOARD

As previously mentioned, the sensor board (block diagram in Fig. 6) provides temperature, pressure, and gamma radiation detection elements. The analog device AD592 temperature sensor outputs a nominal current at room temperature (~25 ºC) of 298.1 µA and will vary at 1 µA/K. The sensor also requires a supply voltage between 4 and 33 V, which will be provided by the selected Microsemi rad-hard voltage regulator. Placing a 10 kΩ resistor at the sensor output converts this current into a DC voltage of 2.981 V with a sensitivity of 10 mV/K. This resistance value optimizes the $V_{out,DC}$ and ΔV components of the temperature sensing circuit output voltage V_{out} so that the common voltage of the ADC is not exceeded and temperature change resolution is measurable. This voltage is input to rad hard block sigma-delta modulator for digitization.

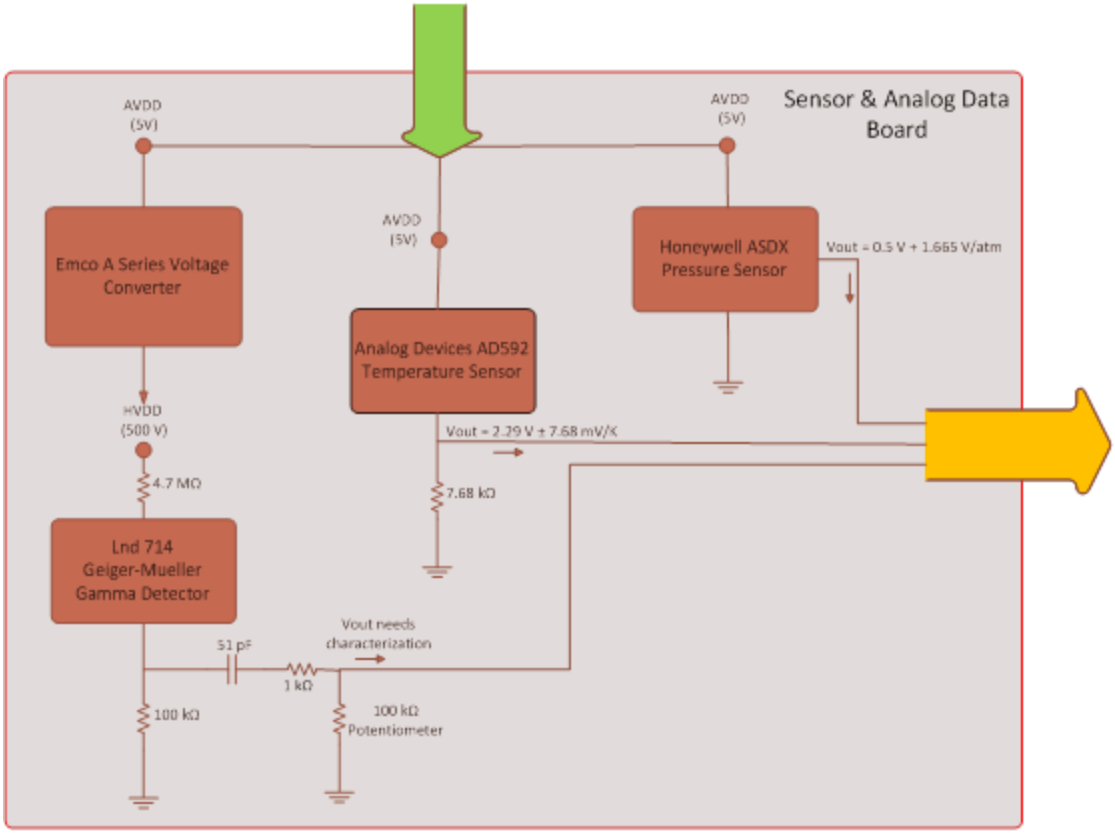


Fig. 6. Detailed sensor board block diagram.

The Honeywell ASDX pressure sensor outputs a voltage that is dependent on gauge pressure and proportional to supply voltage. Based on the sensor characteristics, such as 5 V supply and 10–90 % calibration, the output voltage will be 0.5 V minimum plus 1.66 V/atm above the sealed 1 atm reference pressure, up to 2 atm (Fig. 6). These voltages ranges are sufficient for analog-to-digital conversion without signal conditioning; therefore, this voltage is connected directly to the Triad sigma-delta modulator circuit input.

The LND 714 G-M tube is a purely gamma radiation detection device; it requires a large electric field, in the range of 500 V, in order to gather quickly moving electron-hole pairs freed by high energy incident radiation. When a radiation event occurs, the G-M tube essentially "shorts" as a result of numerous charge carriers traveling toward their respective electrodes. Sizeable current limiting resistors are required to prevent any significant current spikes as a consequence of shorting a 500 V electric field. As the G-M tube shorts, electrons move toward the positive supply, producing a fast positive current spike at the cathode. Since this waveform has a very high frequency AC characteristic, it will take the low impedance path through the capacitor and be converted to voltage across the potentiometer. This voltage is then input to the rad-hard comparator for conversion to a digital logic signal. We have set up the GM-tube voltage and comparator thresholds using a 5-μC ^{137}Cs source.

The manufacturer of the LND 714 G-M tube recommends a 500 V potential for operation in order to capture high-speed freed electrons and holes, but a significant current supply is not

required because the charge produced comes directly from the tube itself. The EMCO A05P5 1 W, positive adjustable DC converter can output up to 500 V at a maximum current of 2 mA, with only a 5 V and less than 200 mA input at no load. This EMCO DC converter is not radiation-hardened and will be placed on the shielded sensors board to minimize high power transmission lengths and optimize board density. The 5 V, 200 mA converter input will be supplied by the Microsemi voltage regulator from the radiation-hardened board. A heat sink will be added to the 5 V regulator to allow operation from both 9.6 V NiMH batteries and lithium ion 11.1 V devices. A photograph of the finished sensor board is shown in Fig. 7. The black coating is a high-voltage insulation material.

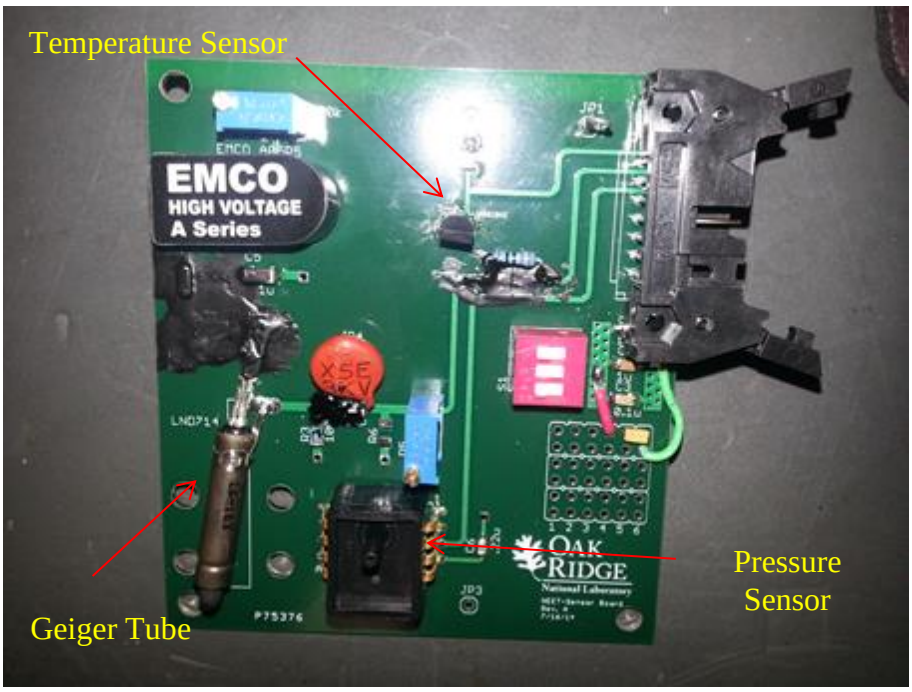


Fig. 7. The Sensor board with insulation coatings.

4.3 INTERFACE BOARD

The function of the interface board (IB, shown in Fig. 8) is to translate the 5 V signals generated by the RHB and buffer and convert them to 3.3 V signals for the Nexys 3 board. In addition, the IB is used to perform the physical connections between the Nexys 3 board and the cable which connects the IB to the RHB and provides an ADC sample-clock/clock loopback from the Nexys 3 board if needed. The isolators are from the Texas Instruments ISO isolator family of interface devices. These circuits allow different supply voltages (in this case, 5 V and 3.3 V) to be used on either side of the devices; they can transmit or receive digital pulses of the appropriate amplitude.

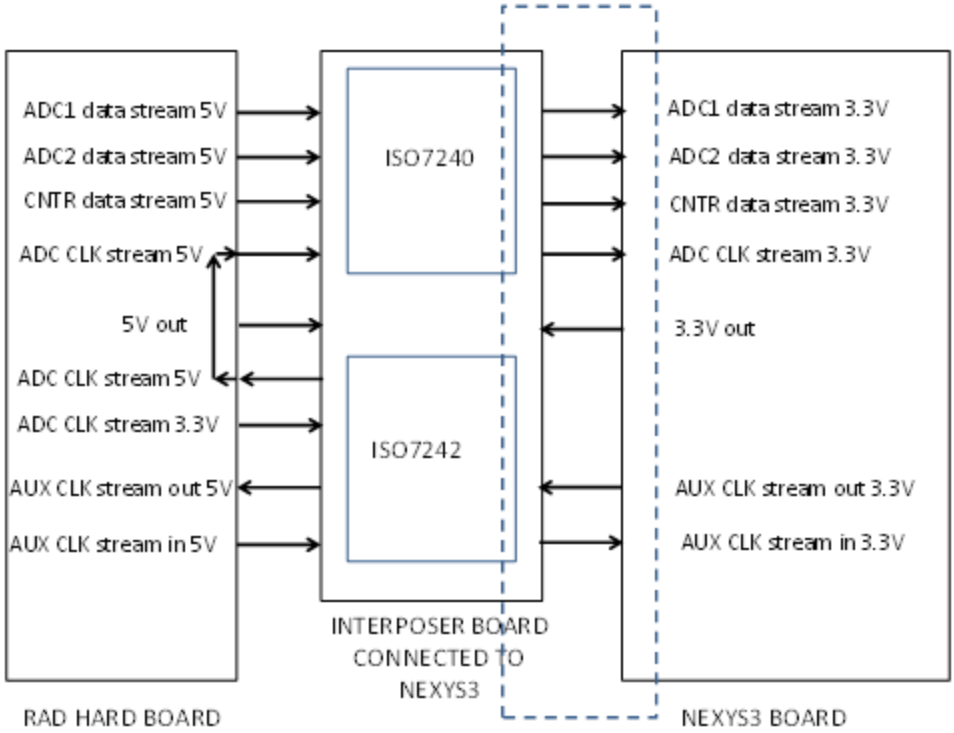


Fig. 8. Detailed interface board data flow block diagram.

4.4 NEXYS 3 BOARD FIRMWARE/COMPUTER INTERFACE

The program created for the FPGA controller was written as a state machine implementation of a universal asynchronous receive transmitter (UART) communications interface. The UART provides a serial data interface to a standard universal serial bus (USB) found on all modern computers. The USB interface provides fast data transfer between the data-collection computer and the rad-hard system. The electrical hardware portion of the interface, made by FTD International, is already resident on the Nexys 3 board (shown in Fig. 9) and requires code to implement the software functionality. The UART interfaces to a standard personal computer USB port and is configured to operate at 9600 baud. Figure A1 illustrates the functional flow and will be described below. All VHDL code can be found in Appendix A.

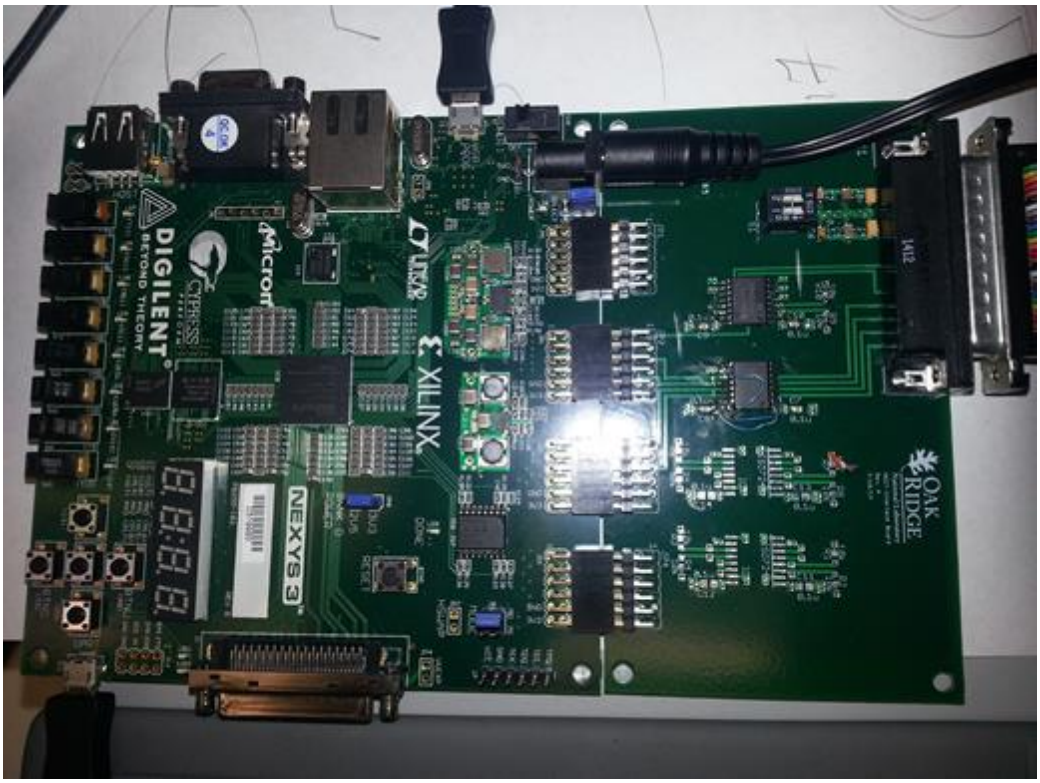


Fig. 9. The interface board (right) connected to the Nexys 3 board.

<u>UART (Block t_serial)</u> is the UART state machine code. It was taken from open-source VHDL code developed by Bainville [2] and modified for this project's purpose. The code implements a single-byte receive and retransmit UART which was minimally modified to receive a command byte and then retransmit either a status byte or a data byte, depending on the code sent. The UART is programmed to receive any of X different ASCII bytes and then parse them in a state machine. If the byte is a valid command, there is a predefined action that takes place. A list of the commands is as follows:

COMMAND	FUNCTION	RETURN CHARACTER
hex 20 (space)	system-wide reset	hex 20 (space)
hex 21 (!)	counter reset	hex 21 (!)
hex 22 (")	system-wide data load	hex 22 (")
hex 30 (0)	counter high-byte load	(data)
hex 31 (1)	counter low-byte load	(data)
hex 32 (2)	adc1(temp) low-byte load	(data)
hex 33 (3)	adc1(temp) high-byte load	(data)
hex 34 (4)	adc2(press) low-byte load	(data)
hex 35 (5)	adc2(press) high-byte load	(data)

For these commands, "temp" means the temperature ADC, and "press" means the

pressure ADC. The command architecture is designed such that combinations of these commands can be issued to perform a greater overall function than any single command. For example, issuing system-wide reset, system wide data load, adc1(temp) low-byte load, adc1(temp) high-byte load would read the entire data from the temperature ADC for a single reading.

Counter (Ctr_16_dp). The counter maintains a running count of the individual events received from the Geiger-tube comparator. After an interval of time determined by the computer software, the counter value is transferred 8 bits at a time (2 bytes) to the computer. Each data transfer requires a transfer command. The counter is then reset and allowed to begin counting again.

Decimation Filter (Decimation_filter_top). The purpose of the decimation filter is to filter out the out-of-band digitization noise from the sigma-delta modulator data. Input to the filter is a single-bit data with variable pulse density that corresponds to the signal level at the modulator/ADC input. In this particular case, the input data represent slowly changing temperature and pressure measurements. For all practical purposes, the modulator input is considered a DC signal.

The decimation filter is implemented as a third-order Cascade, Integrate and Comb (CIC) low-pass filter (shown in Fig. 10). The "N" in Fig. 10 represents the filter order, and the "R" is the filter decimation ratio. In this application, N = 3, and R = 256.

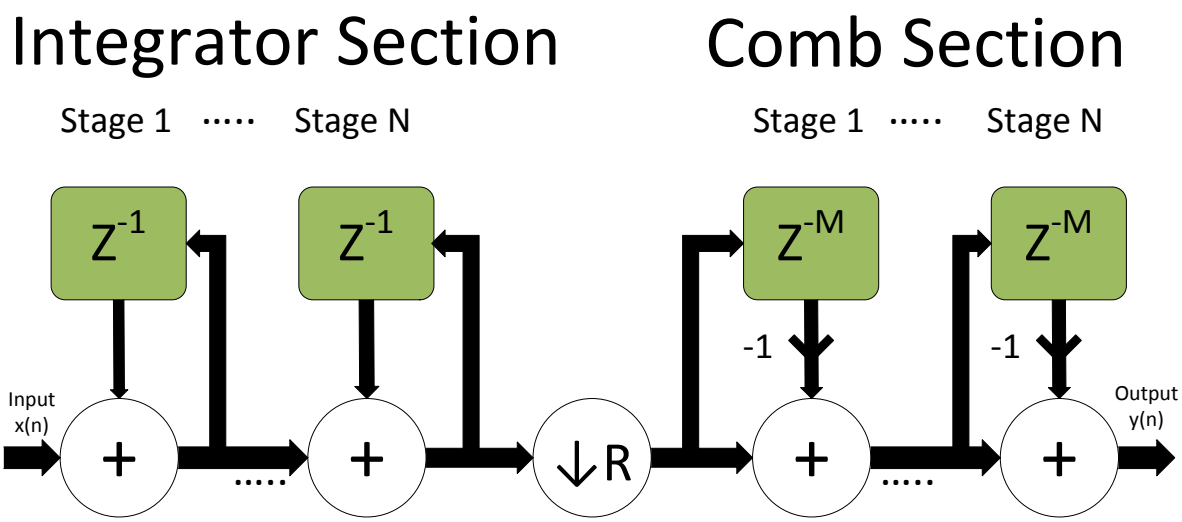


Fig. 10. Decimation filter mathematical structure. The "N" represents the filter order, and the "R" is the filter decimation ratio. In this application N = 3, and R = 256.

The size of the filter is 1 bit, and the output size is 16 bits. The filter's magnitude frequency response without the decimation is shown in Fig. 11.

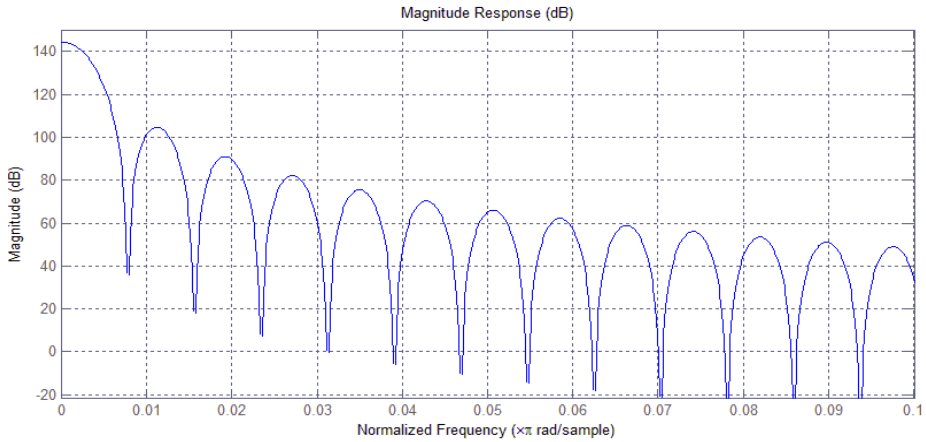


Fig. 11. Decimation filter frequency response.

The performance of the decimation filter was simulated using Simulink. When using the first-order modulator to feed the decimation filter, the dynamic range of the filter's output was around 80, which corresponded to more than 13 bits of resolution. The decimation filter structure from Fig. 10 was implemented in FPGA, and its proper behavior was confirmed in hardware testing.

Clock Generator (ClkGen). The clock generator is a backup clock generator implemented as a frequency divider off of the 100 MHz Nexys 3 system clock. The output value of the clock is 100 MHz/2^6 or 1.5625 MHz.

4.5 LABVIEW INTERFACE PROGRAM

The LabView interface program collects data from the Nexys 3 board programming and displays the value of the data. It transmits the selected command outlined in the UART section and displays the return value of the requested command. It also configures the COM port on the computer to the appropriate settings. A screen shot of the program is shown in Fig. 12.

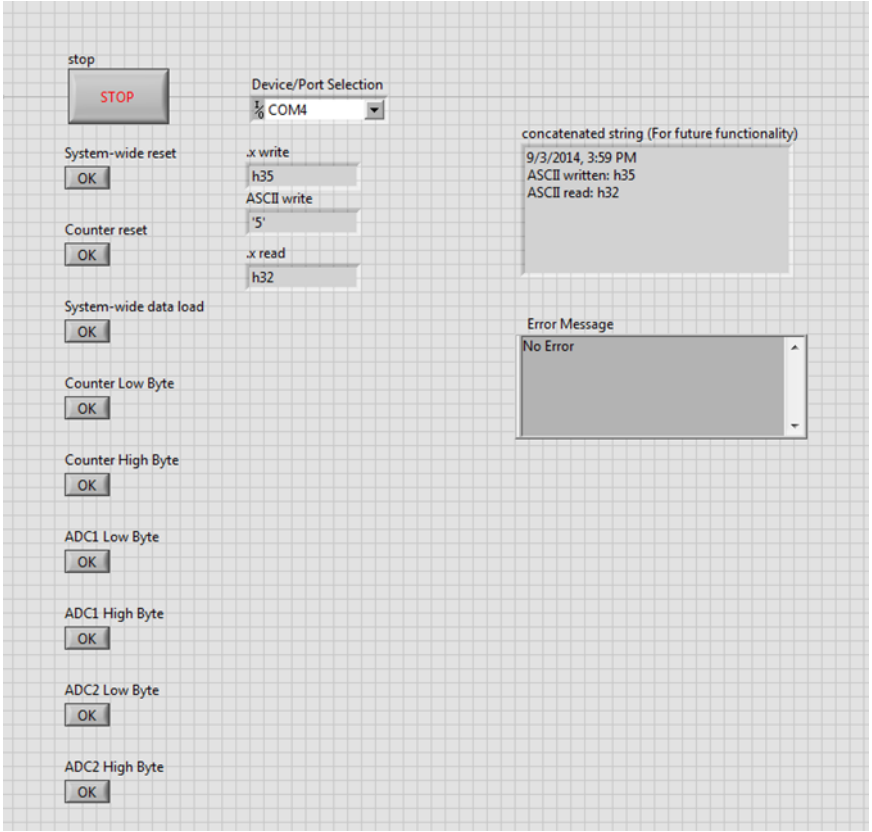
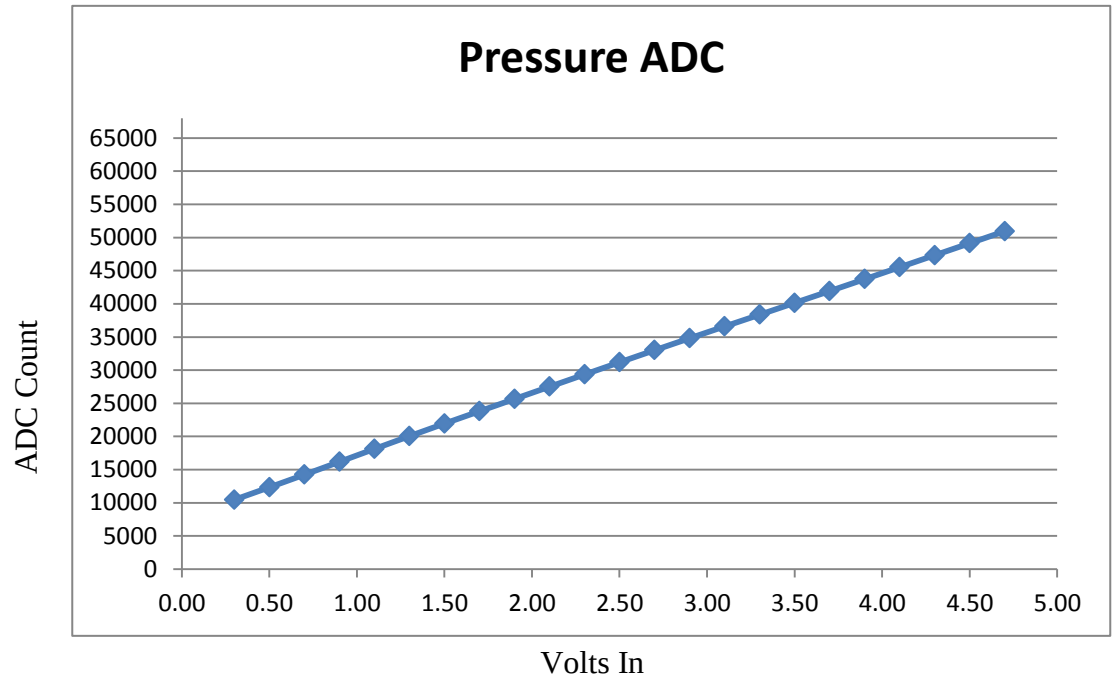


Fig. 12. Screen shot of the LabView interface program.

4.6 PRELIMINARY TEST DATA

We have set up the boards and are now testing each section. Final temperature and pressure calibration will be performed after all boards are fabricated, but preliminary measured data for Board 1 is available (Fig. 13). The input range is adjusted for the particular sensor; it is the actual range that will be used. The output data are in ADC counts; these data will be used to convert the input voltage to the actual temperature and/or pressure. The major feature of interest is that the data are linear and continuous, as expected. There is an additional signal inversion in the temperature ADC chain, and that is the reason that increasing input results in a decreasing ADC code.



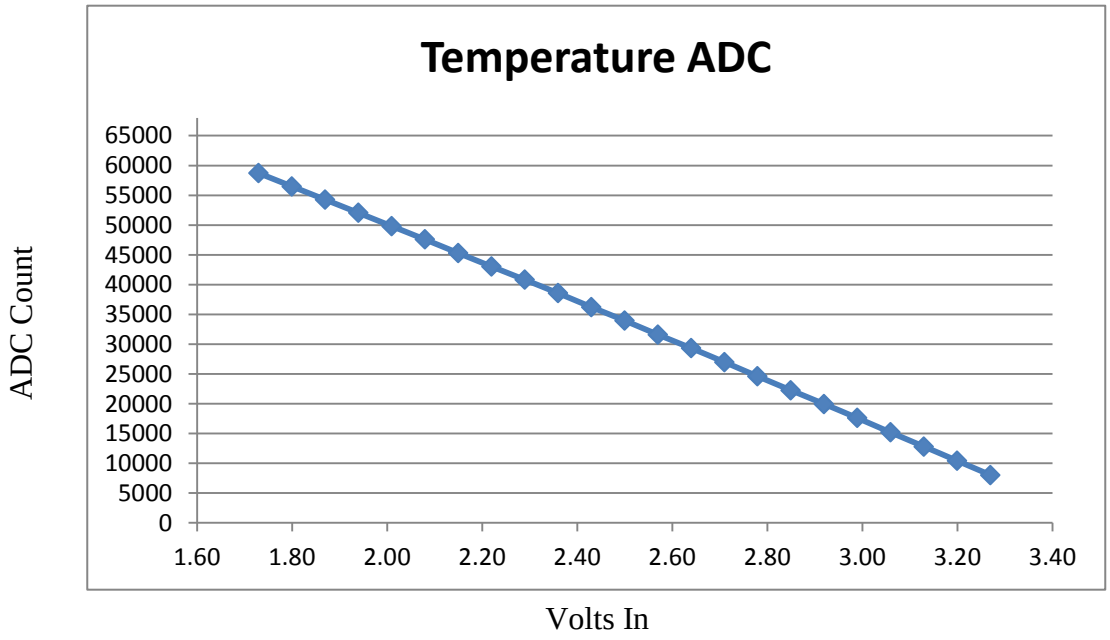


Fig. 13. ADC preliminary testing data.

5. CONCLUSION

We have presented the results of Task 2 of the NEET 2 project "Radiation Hardened Circuitry Using Mask-Programmable Analog Arrays" [1]. This task included design of new boards and implementation of digital filters and communications on the Xilinx Spartan 6 FPGA board. Bench testing was also performed to ensure correct functionality.

The system was subdivided into three component boards. These are a rad-hard analog board, a non-rad-hard sensor board, and a non-rad-hard digital controller board. The digital controller board was chosen to be non-rad-hard because of the excessive cost (many tens of thousands of $ each) for a truly rad-hard digital part. The sensors are off-the-shelf parts that are readily available.

6. REFERENCES

1. C. L. Britton, M. N. Ericson, and B. Blalock, "Radiation Hardened Circuitry Using Mask-Programmable Analog Arrays," proposal submitted under NEET-2: Advanced Sensors and Instrumentation, 2012.
2. Eric Bainville, http://www.bealto.com/home.html.

APPENDIX A. VHDL Code

Top-Level VHDL Schematic

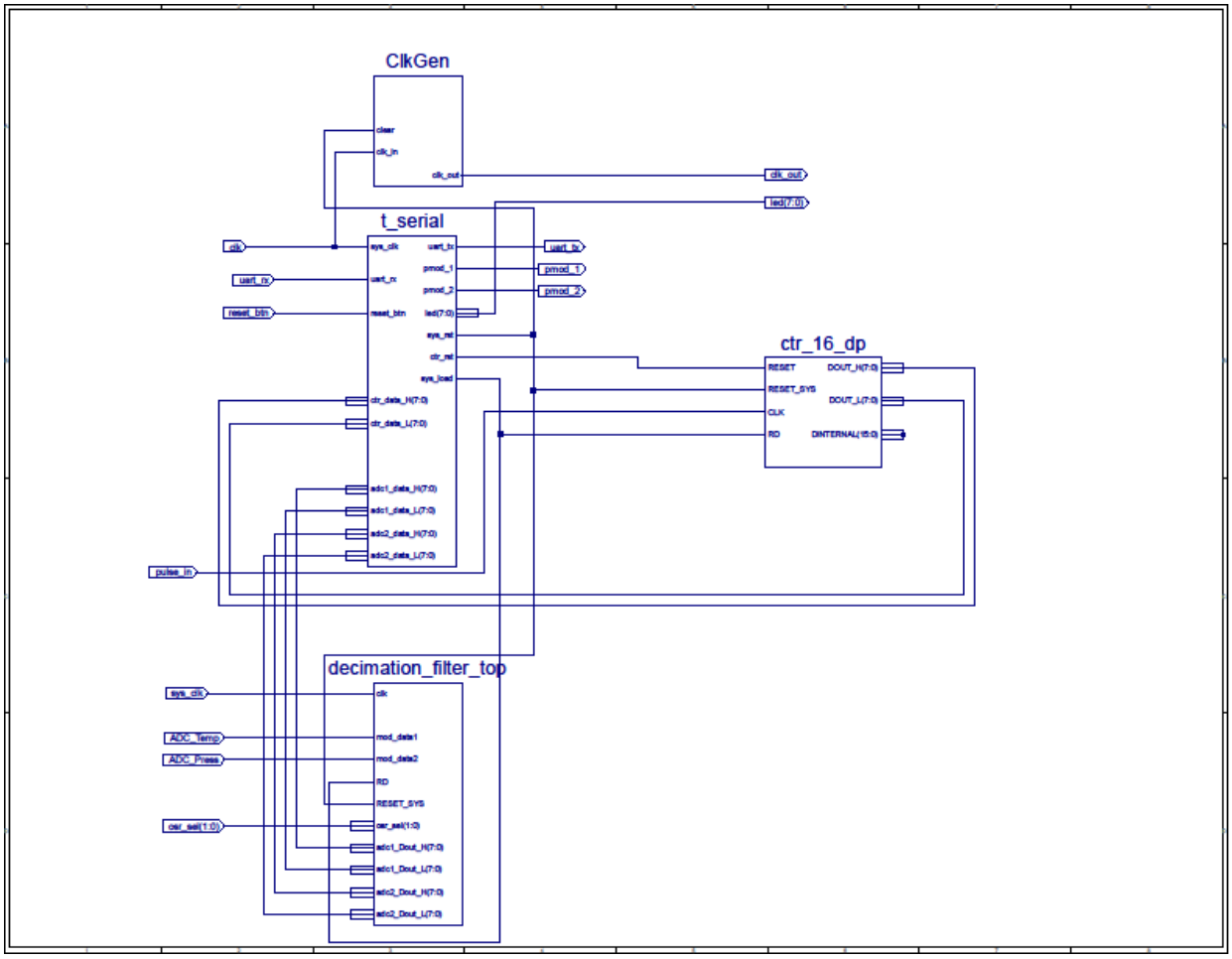


Fig. A1. Top-level instantiation of the VHDL code.

T_serial VHDL Code

****
T_SERIAL
****

```
-- EB Mar 2013
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity t_serial is
port(
  sys_clk: in std_logic; -- 100 MHz system clock

  led: out std_logic_vector(7 downto 0);
  uart_rx: in std_logic;
  uart_tx: out std_logic;

  pmod_1: out std_logic; -- debug outputs
  pmod_2: out std_logic;

  reset_btn: in std_logic;

 -------New ports for register control
  sys_rst: out std_logic:= '0'; -- Generates the system-wide reset (includes all resets)
  ctr_rst: out std_logic:= '0'; -- Counter reset generates the counter ctr_16, NOT the register
  sys_load: out std_logic:= '0'; -- Loads the data from all registers
  ctr_data_H: in std_logic_vector(7 downto 0); --Inputs high byte counter data
  ctr_data_L: in std_logic_vector(7 downto 0); --Inputs low byte counter data
  --ctr_rate_H: in std_logic_vector(7 downto 0); --Inputs high byte counter rate data
  --ctr_rate_L: in std_logic_vector(7 downto 0); --Inputs low byte counter rate data
  adc1_data_H: in std_logic_vector (7 downto 0); --Inputs high byte temperature adc data
  adc1_data_L: in std_logic_vector (7 downto 0); --Inputs low byte temperature adc data
  adc2_data_H: in std_logic_vector (7 downto 0); --Inputs high byte pressure adc data
  adc2_data_L: in std_logic_vector (7 downto 0) --Inputs low byte pressure adc data
 -------End of new signals for register control
);
end t_serial;

architecture Behavioral of t_serial is

component basic_uart is
generic (
  DIVISOR: natural
);
port (
  clk: in std_logic;   -- system clock
  reset: in std_logic;

  -- Client interface
  rx_data: out std_logic_vector(7 downto 0);  -- received byte
  rx_enable: out std_logic;  -- validates received byte (1 system clock spike)
  tx_data: in std_logic_vector(7 downto 0);  -- byte to send
```

```
  tx_enable: in std_logic;  -- validates byte to send if tx_ready is '1'
  tx_ready: out std_logic;  -- if '1', we can send a new byte, otherwise we won't take it

  -- Physical interface
  rx: in std_logic;
  tx: out std_logic
);
end component;

type fsm_state_t is (idle, received, emitting);
type state_t is
record
  fsm_state: fsm_state_t; -- FSM state
  tx_data: std_logic_vector(7 downto 0);
  tx_enable: std_logic;
end record;

signal reset: std_logic;
signal uart_rx_data: std_logic_vector(7 downto 0);
signal uart_rx_enable: std_logic;
signal uart_tx_data: std_logic_vector(7 downto 0);
signal uart_tx_enable: std_logic;
signal uart_tx_ready: std_logic;

signal state,state_next: state_t;
signal sys_rst_state, ctr_rst_state, sys_load_state: std_logic;

begin

  basic_uart_inst: basic_uart
  generic map (DIVISOR => 651) -- 9600
  port map (
    clk => sys_clk, reset => reset,
    rx_data => uart_rx_data, rx_enable => uart_rx_enable,
    tx_data => uart_tx_data, tx_enable => uart_tx_enable, tx_ready => uart_tx_ready,
    rx => uart_rx,
    tx => uart_tx
  );

  reset_control: process (reset_btn) is
  begin



    if reset_btn = '0' then --modified
      reset <= '0';
    else
      reset <= '1';
    end if;
  end process;

  pmod_1 <= uart_tx_enable;
  pmod_2 <= uart_tx_ready;


  fsm_clk: process (sys_clk,reset) is
```

```
  begin
    if reset = '1' then
      state.fsm_state <= idle;
      state.tx_data <= (others => '0');
      state.tx_enable <= '0';
    else
      if rising_edge(sys_clk) then
        state <= state_next;
      end if;
    end if;
  end process;

  fsm_next: process (state,uart_rx_enable,uart_rx_data,uart_tx_ready, ctr_data_H, ctr_data_L,
                                                                    adc1_data_H,        adc1_data_L,        adc2_data_H,
adc2_data_L, sys_clk) is
  begin

      --sys_rst_state <= '0';
                        --ctr_rst_state <= '0';
                        --sys_load_state <= '0';

    state_next <= state;
    case state.fsm_state is

    when idle =>

                        sys_rst_state <= '0';
                        ctr_rst_state <= '0';
                        sys_load_state <= '0';

              --Parses the input and loads an alternative value for output to the screen
      if uart_rx_enable = '1' then


-- Defines a hex 20 (space) as the character for a system-wide reset
                                if uart_rx_data = X"20" then
                                        sys_rst_state <= '1';
                                        state_next.tx_data <= X"20";


-- Defines a hex 21 (!) as the character for a counter reset
                                elsif uart_rx_data = X"21" then
                                        ctr_rst_state <= '1';
                                        state_next.tx_data <= X"21";


-- Defines a hex 22 (") as the character for a sytem-wide load
                                elsif uart_rx_data = X"22" then
                                        sys_load_state <= '1';
                                        state_next.tx_data <= X"22";


-- Defines a hex 30 (0) as the character for a counter high-byte load
                                elsif uart_rx_data = X"30" then
                                        state_next.tx_data <= ctr_data_L (7 downto 0);
```

```
-- Defines a hex 31 (1) as the character for a counter low-byte load
                                elsif uart_rx_data = X"31" then
                                        state_next.tx_data <= ctr_data_H (7 downto 0);



-- Defines a hex 32 (2) as the character for adc1(temp) low-byte load
                                elsif uart_rx_data = X"32" then
                                        state_next.tx_data <= adc1_data_L (7 downto 0);


-- Defines a hex 33 (3) as the character for adc1(temp) high-byte load
                                elsif uart_rx_data = X"33" then
                                        state_next.tx_data <= adc1_data_H (7 downto 0);


-- Defines a hex 34 (4) as the character for adc2(press) low-byte load
                                elsif uart_rx_data = X"34" then
                                        state_next.tx_data <= adc2_data_L (7 downto 0);


-- Defines a hex 35 (5) as the character for adc2(press) high-byte load
                                elsif uart_rx_data = X"35" then
                                        state_next.tx_data <= adc2_data_H (7 downto 0);


                                else
                                        state_next.tx_data <= X"00"; --Outputs NULL if one of the preferred keys is not
pressed.
                                end if;

                        --Sets up for next state
      state_next.tx_enable <= '0';
      state_next.fsm_state <= received;
    end if;

  when received =>
    if uart_tx_ready = '1' then
      state_next.tx_enable <= '1';
      state_next.fsm_state <= emitting;
    end if;

  when emitting =>
    if uart_tx_ready = '0' then
      state_next.tx_enable <= '0';
      state_next.fsm_state <= idle;
    end if;

  end case;
end process;

fsm_output: process (state, uart_rx_data, ctr_rst_state, sys_load_state, sys_rst_state) is
begin

  uart_tx_enable <= state.tx_enable;
```

```
            uart_tx_data <= state.tx_data;
            led <= state.tx_data;
            sys_rst <= sys_rst_state;
            ctr_rst <= ctr_rst_state;
            sys_load <= sys_load_state;


  end process;


end Behavioral;
```

****
BASIC UART
****

```
-- Eric Bainville
-- Mar 2013

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.math_real.all;

entity basic_uart is
  generic (
    DIVISOR: natural  -- DIVISOR = 100,000,000 / (16 x BAUD_RATE)
    -- 2400 -> 2604
    -- 9600 -> 651
    -- 115200 -> 54
    -- 1562500 -> 4
    -- 2083333 -> 3
  );
  port (
    clk: in std_logic;                    -- clock
    reset: in std_logic;                  -- reset

    -- Client interface
    rx_data: out std_logic_vector(7 downto 0); -- received byte
    rx_enable: out std_logic;             -- validates received byte (1 system clock spike)
    tx_data: in std_logic_vector(7 downto 0);  -- byte to send
    tx_enable: in std_logic;              -- validates byte to send if tx_ready is '1'
    tx_ready: out std_logic;              -- if '1', we can send a new byte, otherwise we won't take it

    -- Physical interface
    rx: in std_logic;
    tx: out std_logic
  );
end basic_uart;

architecture Behavioral of basic_uart is
  constant COUNTER_BITS : natural := integer(ceil(log2(real(DIVISOR))));
  type fsm_state_t is (idle, active); -- common to both RX and TX FSM
  type rx_state_t is
  record
    fsm_state: fsm_state_t;               -- FSM state
    counter: std_logic_vector(3 downto 0); -- tick count
```

```
    bits: std_logic_vector(7 downto 0);    -- received bits
    nbits: std_logic_vector(3 downto 0);   -- number of received bits (includes start bit)
    enable: std_logic;                     -- signal we received a new byte
  end record;
  type tx_state_t is
  record
    fsm_state: fsm_state_t; -- FSM state
    counter: std_logic_vector(3 downto 0); -- tick count
    bits: std_logic_vector(8 downto 0); -- bits to emit, includes start bit
    nbits: std_logic_vector(3 downto 0); -- number of bits left to send
    ready: std_logic; -- signal we are accepting a new byte
  end record;

  signal rx_state,rx_state_next: rx_state_t;
  signal tx_state,tx_state_next: tx_state_t;
  signal sample: std_logic; -- 1 clk spike at 16x baud rate
  signal sample_counter: std_logic_vector(COUNTER_BITS-1 downto 0); -- should fit values in 0..DIVISOR-1

begin

  -- sample signal at 16x baud rate, 1 CLK spikes
  sample_process: process (clk,reset) is
  begin
    if reset = '1' then
      sample_counter <= (others => '0');
      sample <= '0';
    elsif rising_edge(clk) then
      if sample_counter = DIVISOR-1 then
        sample <= '1';
        sample_counter <= (others => '0');
      else
        sample <= '0';
        sample_counter <= sample_counter + 1;
      end if;
    end if;
  end process;

  -- RX, TX state registers update at each CLK, and RESET
  reg_process: process (clk,reset) is
  begin
    if reset = '1' then
      rx_state.fsm_state <= idle;
      rx_state.bits <= (others => '0');
      rx_state.nbits <= (others => '0');
      rx_state.enable <= '0';
      tx_state.fsm_state <= idle;
      tx_state.bits <= (others => '1');
      tx_state.nbits <= (others => '0');
      tx_state.ready <= '1';
    elsif rising_edge(clk) then
      rx_state <= rx_state_next;
      tx_state <= tx_state_next;
    end if;
  end process;

  -- RX FSM
```

```
rx_process: process (rx_state,sample,rx) is
begin
  case rx_state.fsm_state is

  when idle =>
    rx_state_next.counter <= (others => '0');
    rx_state_next.bits <= (others => '0');
    rx_state_next.nbits <= (others => '0');
    rx_state_next.enable <= '0';
    if rx = '0' then
      -- start a new byte
      rx_state_next.fsm_state <= active;
    else
      -- keep idle
      rx_state_next.fsm_state <= idle;
    end if;

  when active =>
    rx_state_next <= rx_state;
    if sample = '1' then
      if rx_state.counter = 8 then
        -- sample next RX bit (at the middle of the counter cycle)
        if rx_state.nbits = 9 then
          rx_state_next.fsm_state <= idle; -- back to idle state to wait for next start bit
          rx_state_next.enable <= rx; -- OK if stop bit is '1'
        else
          rx_state_next.bits <= rx & rx_state.bits(7 downto 1);
          rx_state_next.nbits <= rx_state.nbits + 1;
        end if;
      end if;
      rx_state_next.counter <= rx_state.counter + 1;
    end if;

  end case;
end process;

-- RX output
rx_output: process (rx_state) is
begin
  rx_enable <= rx_state.enable;
  rx_data <= rx_state.bits;
end process;

-- TX FSM
tx_process: process (tx_state,sample,tx_enable,tx_data) is
begin
  case tx_state.fsm_state is

  when idle =>
    if tx_enable = '1' then
      -- start a new bit
      tx_state_next.bits <= tx_data & '0';  -- data & start
      tx_state_next.nbits <= "0000" + 10; -- send 10 bits (includes '1' stop bit)
      tx_state_next.counter <= (others => '0');
      tx_state_next.fsm_state <= active;
      tx_state_next.ready <= '0';
```

```
      else
        -- keep idle
        tx_state_next.bits <= (others => '1');
        tx_state_next.nbits <= (others => '0');
        tx_state_next.counter <= (others => '0');
        tx_state_next.fsm_state <= idle;
        tx_state_next.ready <= '1';
      end if;

    when active =>
      tx_state_next <= tx_state;
      if sample = '1' then
        if tx_state.counter = 15 then
          -- send next bit
          if tx_state.nbits = 0 then
            -- turn idle
            tx_state_next.bits <= (others => '1');
            tx_state_next.nbits <= (others => '0');
            tx_state_next.counter <= (others => '0');
            tx_state_next.fsm_state <= idle;
            tx_state_next.ready <= '1';
          else
            tx_state_next.bits <= '1' & tx_state.bits(8 downto 1);
            tx_state_next.nbits <= tx_state.nbits - 1;
          end if;
        end if;
        tx_state_next.counter <= tx_state.counter + 1;
      end if;

    end case;
  end process;

  -- TX output
  tx_output: process (tx_state) is
  begin
    tx_ready <= tx_state.ready;
    tx <= tx_state.bits(0);
  end process;

end Behavioral;
```

Decimation_filter VHDL Code

```
--------------------------------------------------------------------------------
-- Company: ORNL
-- Engineer: Miljko Bobrek
--
-- Create Date:    15:23:08 07/31/2014
-- Design Name:
-- Module Name:    decimation_filter_top - Behavioral
-- Project Name:  NEET Controller
-- Target Devices:
-- Tool versions: ISE 14.7
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity decimation_filter_top is
	Port ( clk 	: in STD_LOGIC; -- 1 MHz
		osr_sel 	: in STD_LOGIC_VECTOR(1 downto 0);
		--osr_sel2 	: in STD_LOGIC_VECTOR(1 downto 0);
		mod_data1 	: in STD_LOGIC;
		mod_data2 	: in STD_LOGIC;
		adc1_Dout_H 	: out STD_LOGIC_VECTOR(7 downto 0);
		adc1_Dout_L 	: out STD_LOGIC_VECTOR(7 downto 0);
		adc2_Dout_H 	: out STD_LOGIC_VECTOR(7 downto 0);
		adc2_Dout_L 	: out STD_LOGIC_VECTOR(7 downto 0);
		RD 	: in STD_LOGIC; 	-- Transfers data from adc(s) to registers.
		RESET_SYS 	: in STD_LOGIC 	-- Overall system reset. Does reset the
output registers.

		);
end decimation_filter_top;

architecture Inside of decimation_filter_top is

	component decimation_filter is
	Port(
		clk 	: in STD_LOGIC; -- 1 MHz
```

```
                    rst                                     : in STD_LOGIC;
                    osr_sel             : in STD_LOGIC_VECTOR(1 downto 0);
                    --osr_sel2                              : in STD_LOGIC_VECTOR(1 downto 0);
                    mod_data1                               : in STD_LOGIC;
                    mod_data2                               : in STD_LOGIC;
                    adc1                                    : out STD_LOGIC_VECTOR(15 downto 0); -- 2's
complement
                    adc2                    : out STD_LOGIC_VECTOR(15 downto 0)  -- 2's complement
                );
        end component;

        component reg_16 is
        Port(
                    RD :    in STD_LOGIC;           -- Moves input data to output.
                    RST2    :       in STD_LOGIC;           -- Register reset.
                    CTR_DIN :       in STD_LOGIC_VECTOR (15 downto 0);  --  Data  intput  from
counter data output. 15-0 MSB-LSB.
                    REG_DOUT_H : out  STD_LOGIC_VECTOR (7 downto 0);
                    REG_DOUT_L : out  STD_LOGIC_VECTOR (7 downto 0)
                );
        end component;

signal adc1_int : STD_LOGIC_VECTOR (15 downto 0); -- Internal lines from adc1 output to register
signal adc2_int : STD_LOGIC_VECTOR (15 downto 0); -- Internal lines from adc2 output to register

begin

decimation_filter_inst : decimation_filter
port map (
        clk => clk,
        rst => RESET_SYS,
        osr_sel => osr_sel,
        mod_data1 => mod_data1,
        mod_data2 => mod_data2,
        adc1 => adc1_int,
        adc2 => adc2_int
);

reg_16_inst : reg_16
port map (
        RD => RD,
        RST2 => RESET_SYS,
        CTR_DIN => adc1_int,
        REG_DOUT_H => adc1_Dout_H,
        REG_DOUT_L => adc1_Dout_L
);

reg_16_inst2 : reg_16
port map (
        RD => RD,
        RST2 => RESET_SYS,
        CTR_DIN => adc2_int,
        REG_DOUT_H => adc2_Dout_H,
        REG_DOUT_L => adc2_Dout_L
);
```

```
end Inside;



--
==========================================================================================
==============================--
-- This file implements two 3rd order sinc decimation filters with selectable decimation/oversampling ratio (OSR)
--  osr_sel    "00"    "01"   "10"   "11"
--  OSR        128     256    512    1024
--
==========================================================================================
==============================--

library IEEE;
library UNISIM;
use IEEE.STD_LOGIC_1164.all;

use IEEE.STD_LOGIC_arith.all;
use IEEE.STD_LOGIC_unsigned.all;

use UNISIM.Vcomponents.ALL;

entity decimation_filter is

port (
                clk                                     : in STD_LOGIC; -- 1 MHz
                rst                                     : in STD_LOGIC;
    osr_sel                     : in STD_LOGIC_VECTOR(1 downto 0);
    --osr_sel2                          : in STD_LOGIC_VECTOR(1 downto 0);
                mod_data1                           : in STD_LOGIC;
    mod_data2                         : in STD_LOGIC;
                adc1                                  :  out  STD_LOGIC_VECTOR(15  downto  0);  --  2's
complement
                adc2                        : out STD_LOGIC_VECTOR(15 downto 0)  -- 2's complement
                --osr_en                              :  out  STD_LOGIC  --  Possibly  use  for  making  sure  no
metastability state (RD and adc data change at same time)
                );
end;

architecture decimation_filter_arch of decimation_filter is

component integrator is

port (
                clk                                     : in STD_LOGIC;
                rst                                     : in STD_LOGIC;
                mod_data                              : in STD_LOGIC;
                out_data                            : out STD_LOGIC_VECTOR(31 downto 0)
                );
end component;

component differentiator is

port (
```

```
                clk                                     : in STD_LOGIC;
                rst                                     : in STD_LOGIC;
                en                                      : in STD_LOGIC;
                in_data                 : in STD_LOGIC_VECTOR(31 downto 0);
                out_data                        : out STD_LOGIC_VECTOR(31 downto 0)
                );
end component;

signal cnt1                     : STD_LOGIC_VECTOR(15 downto 0);
signal cnt2                     : STD_LOGIC_VECTOR(15 downto 0);
signal int_out1     : STD_LOGIC_VECTOR(31 downto 0);
signal int_out2     : STD_LOGIC_VECTOR(31 downto 0);
signal out_data1    : STD_LOGIC_VECTOR(31 downto 0);
signal out_data2    : STD_LOGIC_VECTOR(31 downto 0);
signal osr1b        : STD_LOGIC_VECTOR(15 downto 0);
signal osr2b        : STD_LOGIC_VECTOR(15 downto 0);
signal osr_en1              : STD_LOGIC;
signal osr_en2      : STD_LOGIC;
signal clkb         : STD_LOGIC;
signal adc1_2s     : STD_LOGIC_VECTOR(15 downto 0);
signal adc2_2s     : STD_LOGIC_VECTOR(15 downto 0);


begin

buff1: BUFG port map (I => clk, O => clkb);

osr1b <= X"007F" when osr_sel = "00" else -- OSR = 128
        X"00FF" when osr_sel = "01" else -- OSR = 256
                                X"01FF" when osr_sel = "10" else -- OSR = 512
                                X"03FF";                -- OSR = 1024
osr2b <= X"007F" when osr_sel = "00" else -- OSR = 128
        X"00FF" when osr_sel = "01" else -- OSR = 256
                                X"01FF" when osr_sel = "10" else -- OSR = 512
                                X"03FF";                -- OSR = 1024

--adc1 <= (15 => not out_data1(21), OTHERS => out_data1(20 downto 6)) when osr_sel = "00" else -- Changed
adc1_2s to adc1
--                      (15 => not out_data1(24), OTHERS => out_data1(23 downto 9)) when osr_sel = "01" else
--                      (15 => not out_data1(27), OTHERS => out_data1(26 downto 12)) when osr_sel = "10" else
--                      (15 => not out_data1(30), OTHERS => out_data1(29 downto 15));

adc1_2s <= out_data1(21 downto 6)  when osr_sel = "00" else -- Changed adc1_2s to adc1
                                out_data1(24 downto 9)  when osr_sel = "01" else
                                out_data1(27 downto 12) when osr_sel = "10" else
                                out_data1(30 downto 15);

--adc2 <= (15 => not out_data2(21), OTHERS => out_data2(20 downto 6)) when osr_sel = "00" else -- Changed
adc1_2s to adc1
--                      (15 => not out_data2(24), OTHERS => out_data2(23 downto 9)) when osr_sel = "01" else
--                      (15 => not out_data2(27), OTHERS => out_data2(26 downto 12)) when osr_sel = "10" else
--                      (15 => not out_data2(30), OTHERS => out_data2(29 downto 15));

adc2_2s <= out_data2(21 downto 6)  when osr_sel = "00" else -- Changed adc2_2s to adc2
                                out_data2(24 downto 9)  when osr_sel = "01" else
                                out_data2(27 downto 12) when osr_sel = "10" else
```

```
                                        out_data2(30 downto 15);


--osr_en <= osr_en1;

process (clkb,rst)

begin

if rst = '1' then
            cnt1 <= X"0000";
            cnt2 <= X"0000";
            osr_en1 <= '0';
            osr_en2 <= '0';
elsif clkb'event and clkb = '1' then
   if cnt1 = osr1b then
      cnt1 <= X"0000";
                        osr_en1 <= '1';
            else
      cnt1 <= cnt1 + '1';
                        osr_en1 <= '0';
   end if;
   if cnt2 = osr2b then
      cnt2 <= X"0000";
                        osr_en2 <= '1';
            else
      cnt2 <= cnt2 + '1';
                        osr_en2 <= '0';
   end if;
end if;

end process;

INTGR1: integrator
  port map (
           rst      => rst,
                                        clk      => clkb,
                                        mod_data => mod_data1,
                                        out_data => int_out1
                                        );

INTGR2: integrator
  port map (
           rst      => rst,
                                        clk      => clkb,
                                        mod_data => mod_data2,
                                        out_data => int_out2
                                        );

DIFF1: differentiator
  port map (
           rst      => rst,
                                        clk      => clkb,
                                        en       => osr_en1,
                                        in_data  => int_out1,
```

```
                                        out_data => out_data1
                                        );

DIFF2: differentiator
  port map (
          rst      => rst,
                                        clk      => clkb,
                                        en       => osr_en2,
                                        in_data  => int_out2,
                                        out_data => out_data2
                                        );

-- Converting 2's compliment to binary
adc1(15) <= not adc1_2s(15);
adc1(14 downto 0) <= adc1_2s(14 downto 0);
adc2(15) <= not adc2_2s(15);
adc2(14 downto 0) <= adc2_2s(14 downto 0);

end decimation_filter_arch;

--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    10:11:06 06/25/2014
-- Design Name:
-- Module Name:    integrator - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
library UNISIM;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.all;
use UNISIM.VComponents.all;

entity differentiator is

port (
                clk                                     : in STD_LOGIC;
                rst                                     : in STD_LOGIC;
                en                                        : in STD_LOGIC;
                in_data                 : in STD_LOGIC_VECTOR(31 downto 0);
                out_data                        : out STD_LOGIC_VECTOR(31 downto 0)
                );
end;
```

```
architecture Behavioral of differentiator is

signal diff1                    : STD_LOGIC_VECTOR(31 downto 0);
signal diff2                    : STD_LOGIC_VECTOR(31 downto 0);
signal diff3                    : STD_LOGIC_VECTOR(31 downto 0);
signal diffd1                   : STD_LOGIC_VECTOR(31 downto 0);
signal diffd2                   : STD_LOGIC_VECTOR(31 downto 0);
signal diffd3                   : STD_LOGIC_VECTOR(31 downto 0);
signal in_data1         : STD_LOGIC_VECTOR(31 downto 0);

begin

out_data <= diff3;

diff1 <= in_data1 - diffd1;
diff2 <= diff1 - diffd2;
diff3 <= diff2 - diffd3;

process (clk,rst)

begin

if rst = '1' then
        diffd1 <= X"00000000";
        diffd2 <= X"00000000";
        diffd3 <= X"00000000";
        in_data1 <= X"00000000";
elsif clk'event and clk = '1' then
  if en = '1' then
          in_data1 <= in_data;
    diffd1 <= in_data1;
    diffd2 <= diff1;
          diffd3 <= diff2;
        end if;
end if;

end process;


end Behavioral;

--------------------------------------------------------------------------------
-- Company: ORNL
-- Engineer: Miljko Bobrek
--
-- Create Date:    10:11:06 06/25/2014
-- Design Name:
-- Module Name:    integrator - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
```

```
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
library UNISIM;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.all;
use UNISIM.VComponents.all;

entity integrator is

port (
                clk                                     : in STD_LOGIC;
                rst                                     : in STD_LOGIC;
                mod_data                        : in STD_LOGIC;
                out_data                        : out STD_LOGIC_VECTOR(31 downto 0)
                );
end;

architecture Behavioral of integrator is

signal acc1                     : STD_LOGIC_VECTOR(31 downto 0);
signal acc2                     : STD_LOGIC_VECTOR(31 downto 0);
signal acc3                     : STD_LOGIC_VECTOR(31 downto 0);
signal mod_data1  : STD_LOGIC_VECTOR(31 downto 0);

begin

mod_data1 <= X"00000001" when mod_data = '1' else
        X"FFFFFFFF";

out_data <= acc3;

process (clk,rst)

begin

if rst = '1' then
        acc1 <= X"00000000";
        acc2 <= X"00000000";
        acc3 <= X"00000000";
elsif clk'event and clk = '1' then
  acc1 <= acc1 + mod_data1;
  acc2 <= acc2 + acc1;
        acc3 <= acc3 + acc2;
end if;

end process;


end Behavioral;


----------------------------------------------------------------------------------
-- Company: ORNL
```

```
-- Engineer: Chuck Britton
--
-- Create Date:    10:24:43 06/02/2014
-- Design Name:
-- Module Name:    reg_16 - Behavioral
-- Project Name: NEET Controller
-- Target Devices:
-- Tool versions: ISE 14.7
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity reg_16 is
   Port ( RD : in  STD_LOGIC;
        RST2 : in  STD_LOGIC;
        CTR_DIN : in  STD_LOGIC_VECTOR (15 downto 0);
        REG_DOUT_H : out  STD_LOGIC_VECTOR (7 downto 0);
                                REG_DOUT_L : out  STD_LOGIC_VECTOR (7 downto 0));
end reg_16;

architecture Behavioral of reg_16 is

begin

register_process: process (RST2, RD)
begin
          if (RST2 = '1') then
                    REG_DOUT_H <=(others => '0');
                    REG_DOUT_L <=(others => '0');
          else
                    if (rising_edge (RD)) then
                              REG_DOUT_H(7 downto 0) <= CTR_DIN(15 downto 8);
                              REG_DOUT_L(7 downto 0) <= CTR_DIN(7 downto 0);
                    end if;
          end if;
end process ;


end Behavioral;
```

Counter VHDL Code

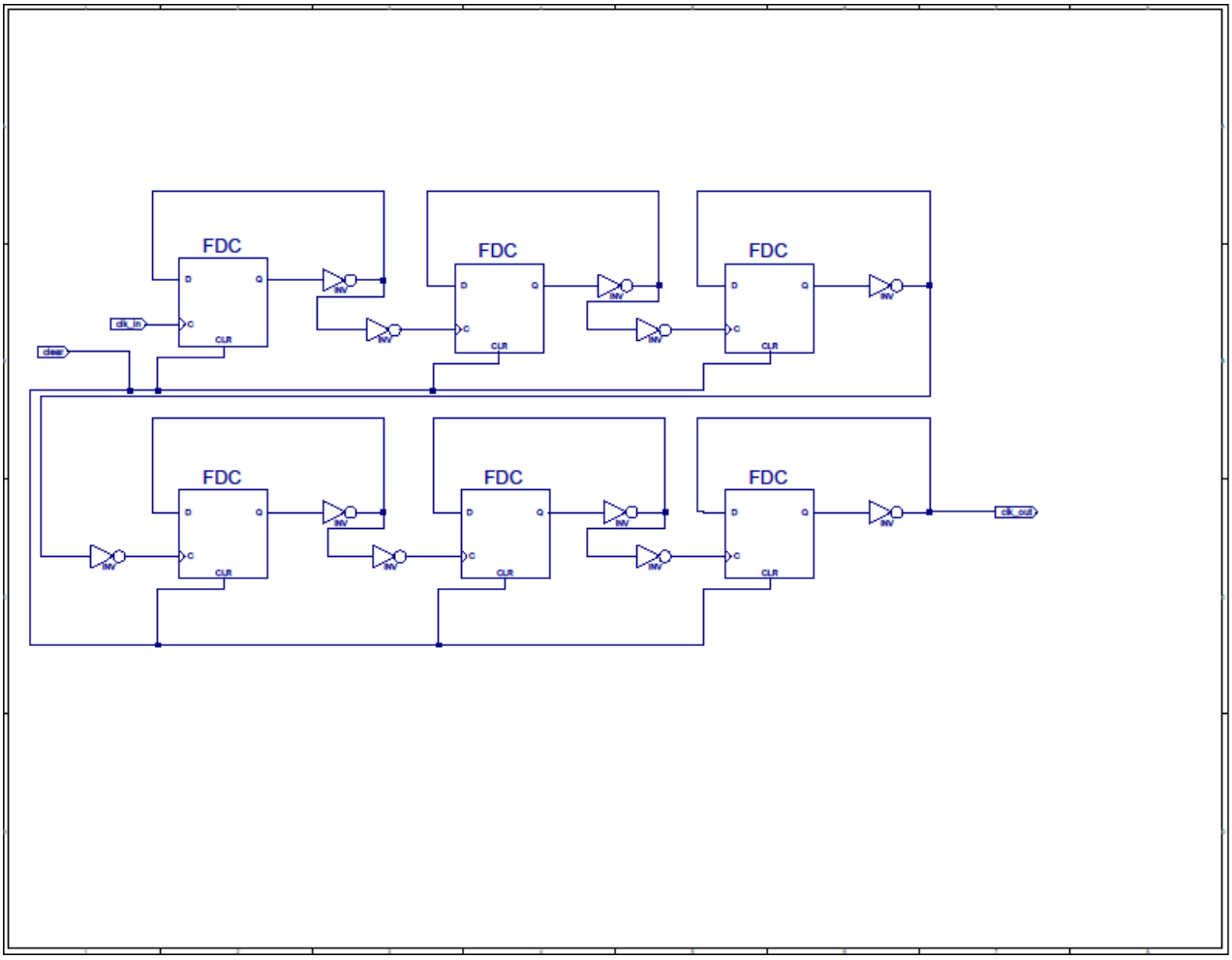
```
--------------------------------------------------------------------------------
-- Company: ORNL
-- Engineer: Chuck Britton
--
-- Create Date:    13:04:28 05/30/2014
-- Design Name:
-- Module Name:    ctr-16-dp - Behavioral
-- Project Name: NEET Controller
-- Target Devices:
-- Tool versions: ISE 14.7
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity ctr_16_dp is
   Port ( RESET : in  STD_LOGIC;            -- Counter reset after each read cycle.  Does not reset the output
registers.
                              RESET_SYS : in STD_LOGIC;   -- Overall system reset. Does reset the output
registers.
         CLK : in  STD_LOGIC;                -- Events to be counted input.
         RD : in  STD_LOGIC;                 -- Transfers data from counter to registers.
         DOUT_H : out  STD_LOGIC_VECTOR (7 downto 0);-- Data output from registers.
                              DOUT_L : out  STD_LOGIC_VECTOR (7 downto 0);-- Data output from registers.
                              DINTERNAL : out STD_LOGIC_VECTOR (15 downto 0) -- Output from the counter
that goes to the buffer.
                              );
end ctr_16_dp;

architecture inside of ctr_16_dp is

            component ctr_16 is
            Port(
                              CTR_CLK : in STD_LOGIC;  -- Event inputs.  Not really a clock.
                              CTR_RST : in STD_LOGIC;        -- Resets the output of the counter but not the
register.
                              SYS_RST : in STD_LOGIC; -- Also resets the output of the counter
                              CTR_DOUT: out STD_LOGIC_VECTOR (15 downto 0) -- Output of the counter.
```

```
                        );
        end component;

        component reg_16 is
        Port(
                        RD :    in STD_LOGIC;           -- Moves input data to output.
                        RST2    :       in STD_LOGIC;           -- Register reset.
                        CTR_DIN :       in STD_LOGIC_VECTOR (15 downto 0);  --  Data  intput  from
counter data output. 15-0 MSB-LSB.
                        REG_DOUT_H : out  STD_LOGIC_VECTOR (7 downto 0);
                        REG_DOUT_L : out  STD_LOGIC_VECTOR (7 downto 0)
                        );
        end component;


signal DINT : STD_LOGIC_VECTOR (15 downto 0);  --Internal lines from the counter output.

begin

ctr_16_inst : ctr_16
port map (
        CTR_CLK => CLK,
        CTR_RST => RESET,
        SYS_RST => RESET_SYS,
        CTR_DOUT => DINT
);


reg_16_inst : reg_16
port map (
        RD => RD,
        RST2 => RESET_SYS,
        CTR_DIN => DINT,
        REG_DOUT_H => DOUT_H,
        REG_DOUT_L => DOUT_L
);



DINTERNAL <= DINT;  -- Moves the counter data to the output.  Just a diagnostic.

end inside;



--------------------------------------------------------------------------------
-- Company: ORNL
-- Engineer: Chuck Britton
--
-- Create Date:    10:13:48 06/02/2014
-- Design Name:
-- Module Name:    ctr_16 - Behavioral
-- Project Name: NEET Controller
-- Target Devices:
-- Tool versions: ISE 14.7
-- Description:
```

```
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity ctr_16 is
   Port ( CTR_CLK : in  STD_LOGIC;
          CTR_RST : in  STD_LOGIC;
                                SYS_RST : in  STD_LOGIC;
          CTR_DOUT : out  STD_LOGIC_VECTOR (15 downto 0)
                                );
end ctr_16;

architecture Behavioral of ctr_16 is

signal CTR_DOUT_INCR : STD_LOGIC_VECTOR (15 downto 0);

begin

counter_process: process (CTR_CLK, CTR_RST, SYS_RST)
begin
          if CTR_RST = '1' then
                    CTR_DOUT_INCR <=(others => '0');

          elsif SYS_RST = '1' then
                    CTR_DOUT_INCR <=(others => '0');
          else
                    if (rising_edge (CTR_CLK)) then
                              CTR_DOUT_INCR <= std_logic_vector(unsigned(CTR_DOUT_INCR) + 1);
                    end if;
          end if;
end process ;

CTR_DOUT <= CTR_DOUT_INCR;

end Behavioral;


----------------------------------------------------------------------------------
-- Company: ORNL
-- Engineer: Chuck Britton
```

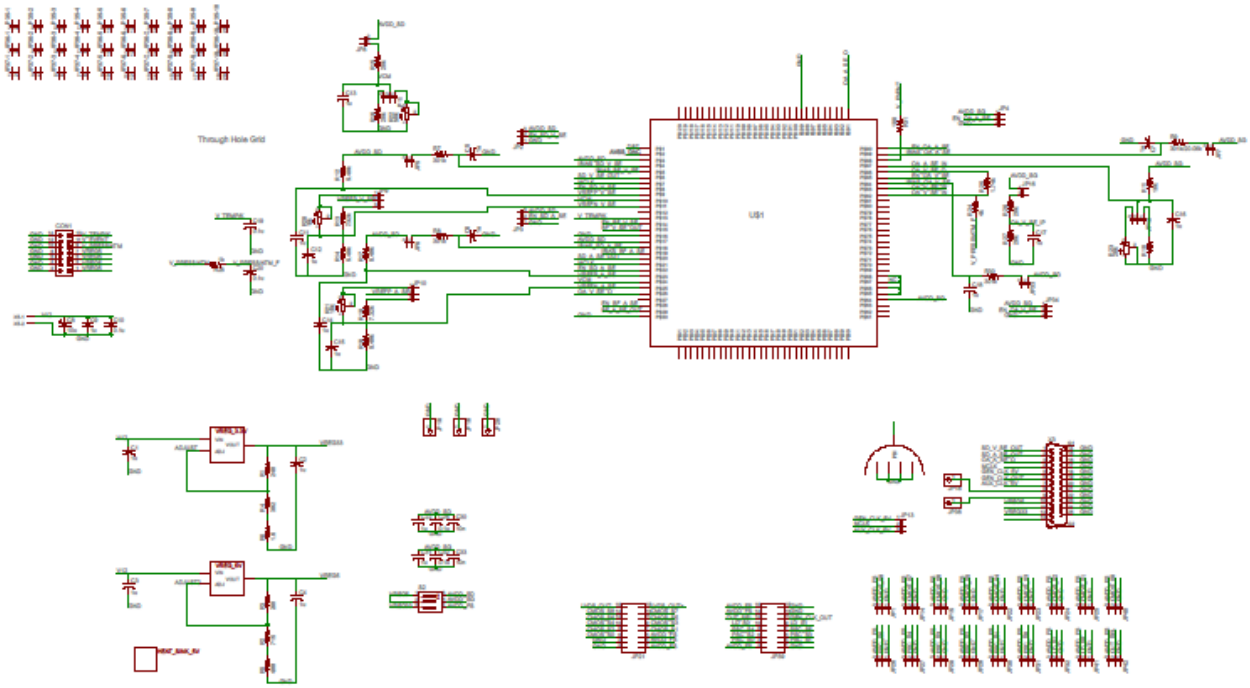
```
--
-- Create Date:    10:24:43 06/02/2014
-- Design Name:
-- Module Name:    reg_16 - Behavioral
-- Project Name: NEET Controller
-- Target Devices:
-- Tool versions: ISE 14.7
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity reg_16 is
   Port ( RD : in  STD_LOGIC;
          RST2 : in  STD_LOGIC;
          CTR_DIN : in  STD_LOGIC_VECTOR (15 downto 0);
          REG_DOUT_H : out  STD_LOGIC_VECTOR (7 downto 0);
                                REG_DOUT_L : out  STD_LOGIC_VECTOR (7 downto 0));
end reg_16;

architecture Behavioral of reg_16 is

begin

register_process: process (RST2, RD)
begin
            if (RST2 = '1') then
                        REG_DOUT_H <=(others => '0');
                        REG_DOUT_L <=(others => '0');
            else
                        if (rising_edge (RD)) then
                                    REG_DOUT_H(7 downto 0) <= CTR_DIN(15 downto 8);
                                    REG_DOUT_L(7 downto 0) <= CTR_DIN(7 downto 0);
                        end if;
            end if;
end process ;

end Behavioral;
```

Clock Generator VHDL Schematic



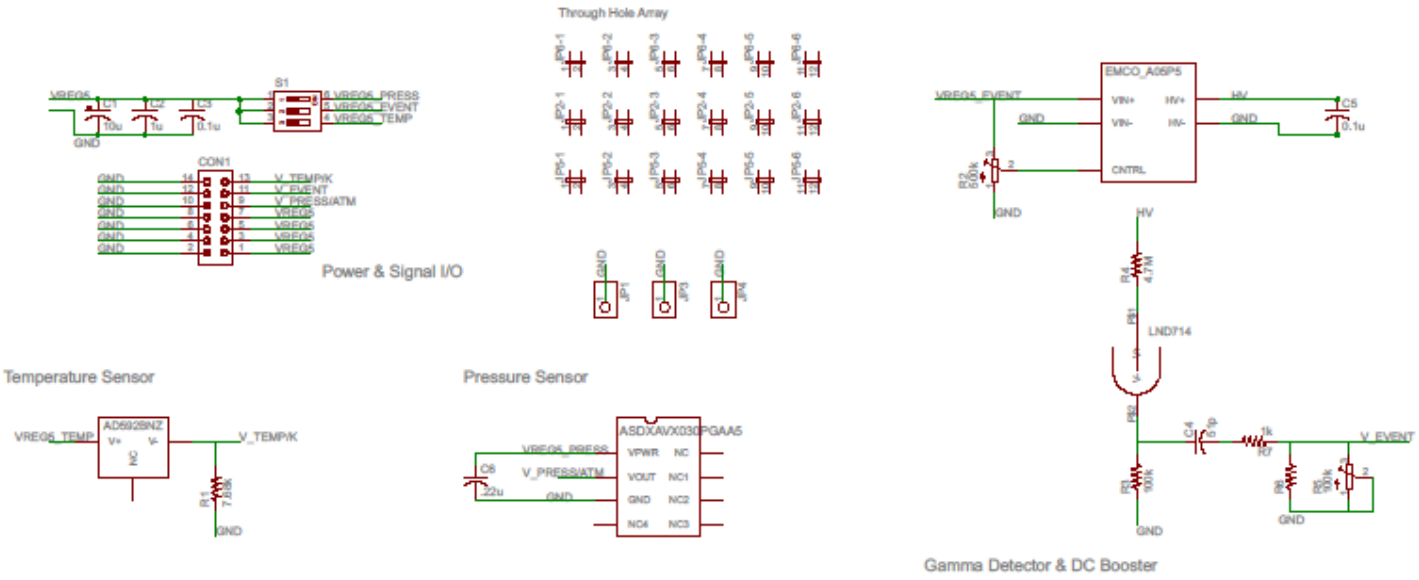
APPENDIX B. BOARD SCHEMATICS

Rad-Hard Board Schematic



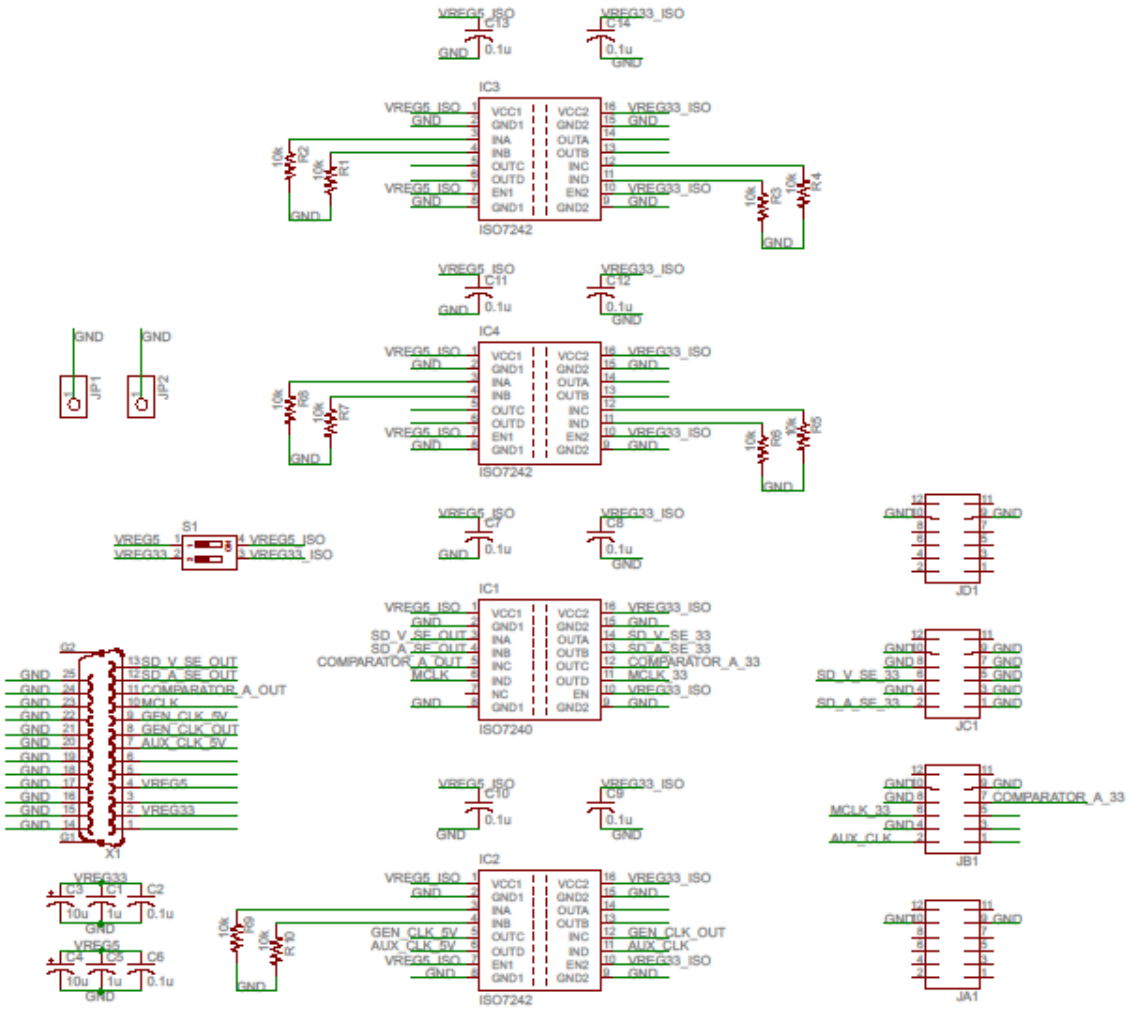
9/3/2014 4:16:28 PM f=0.48 C:\Users\hsv\Documents\eagle\RH System\TriadBoard.sch (Sheet: 1/1)

Sensor Board Schematic



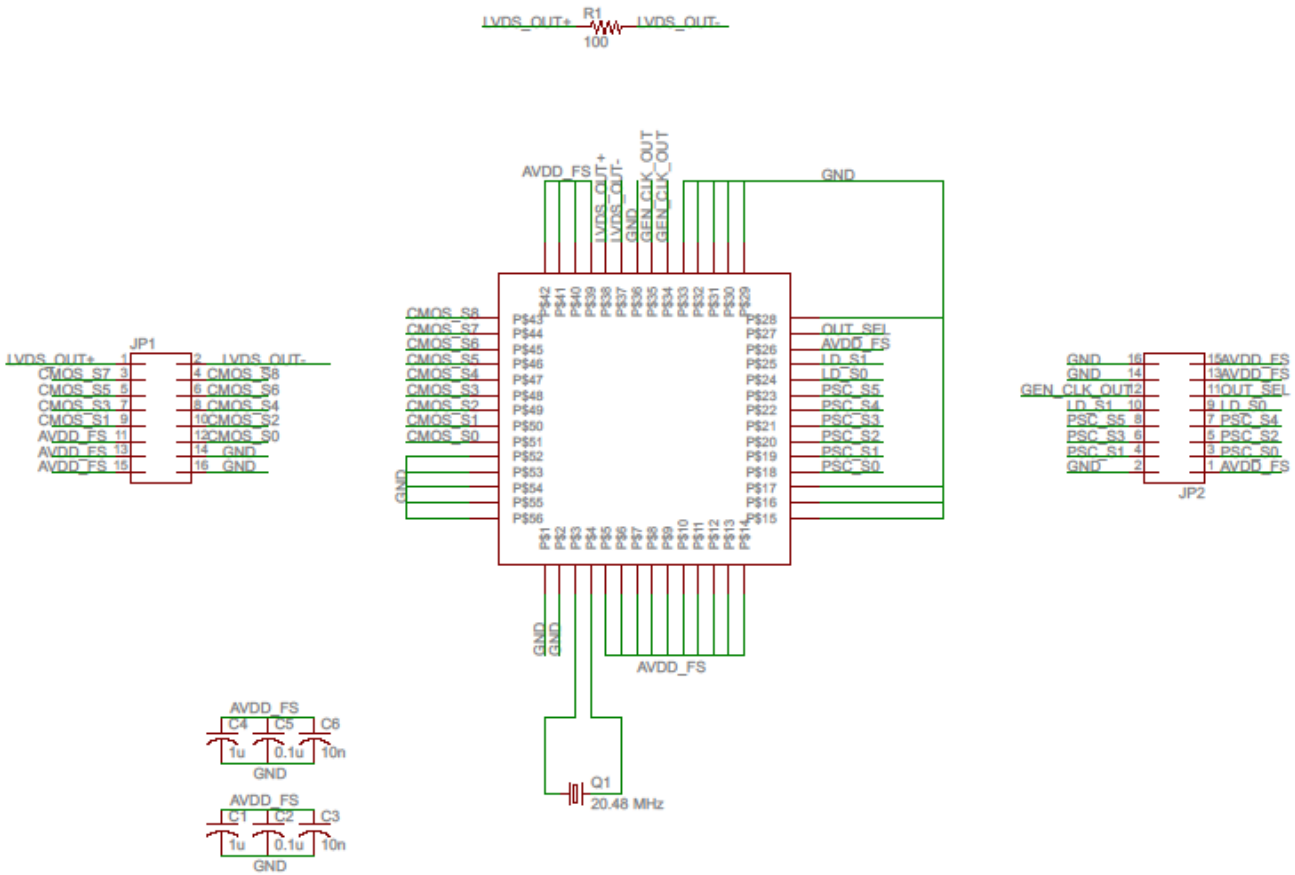
9/3/2014 4:17:17 PM f=0.87 C:\Users\hsv\Documents\eagle\RH System\SensorBoard.sch (Sheet: 1/1)

Interface Board Schematic



9/3/2014 4:15:05 PM C:\Users\hsv\Documents\eagle\RH System\InterfaceBoard.sch (Sheet: 1/1)

Interposer Board Schematic



9/3/2014 4:14:00 PM f=1.20 C:\Users\hsv\Documents\eagle\RH System\InterposerBoard.sch (Sheet: 1/1)