

FPGA-Based Trigger for LArIAT

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science degree in Physics from the College of William and Mary

by

Matthew James Stephens

Advisor: Mike Kordosky

Senior Research Coordinator: Henry Krakauer

Date: April 23, 2014

Introduction

My senior project has focused on the development and testing of a trigger system that will be used for data acquisition in the Liquid Argon In A Test Beam (LArIAT) experiment at Fermilab. The purpose of LArIAT is to characterize and calibrate liquid argon time projection chambers (LArTPC) as they are becoming popular detectors for conducting neutrino physics research due to their ability to show interactions in fine detail. Testing will consist of using Fermilab's accelerator to shoot several different species of particles with well-known energies into the LArTCP. The TPC will be used to identify the particles and any secondaries created when they interact. It will also be used to measure the energy of the incoming particles and determine the cross-section for inelastic interactions.

Liquid Argon Time Projection Chambers

The TPC consists of a liquid Argon filled chamber whose dimensions are approximately one-half meter in both the x and y directions and one meter in the z direction (see Figure 1). A particle travelling along the z direction enters the chamber and ionizes. An electric field of 500 V/cm applied in the negative y direction prevents recombination and causes electrons to drift in the positive y direction towards induction and collection planes, which are planes of crisscrossed wires. The induced signals on the induction and collection planes provide measurements of the x and z positions. The drift speed of the electrons in the chamber is known, so a measurement of the y position can be determined from the time it takes an electron to drift to the collection plane. This is why such a detector is known as a time projection chamber (1).

The outputs of the induction and collection planes are stored in circular buffers, so the detector needs to know when a particle of interest enters the chamber and when to read out data from the buffer. The maximum drift time is known, so it can be used to delay the read out from

the time that a particle enters the chamber. This is what the trigger system accomplishes. Its core functionality is comprised of detecting when a particle of interest enters the chamber, waiting a programmable length of time, and then generating a trigger signal.

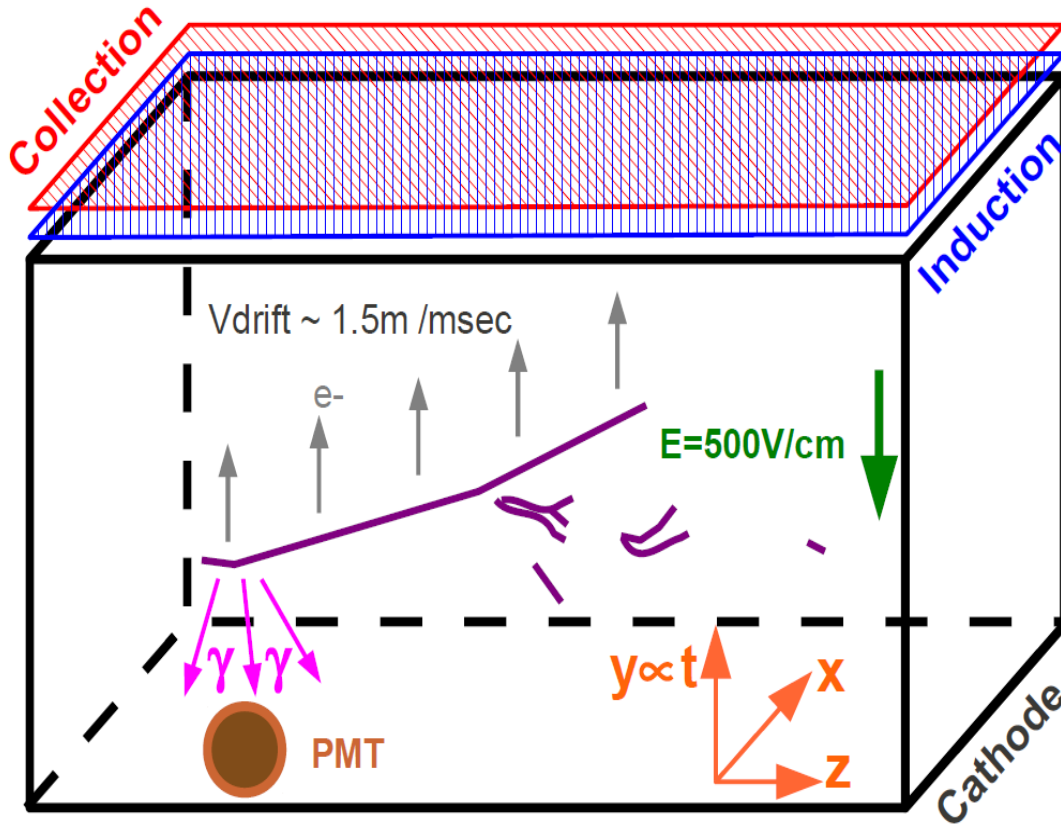


Figure 1. Time Projection Chamber. Courtesy of M. Kordosky, W&M.

Field-Programmable Gate Arrays

The trigger system for LArIAT was developed in VHDL (Very High Speed Integrated Circuit Hardware Description Language) for use on the user field-programmable gate array (FPGA) of a CAEN V1495 board. An FPGA is a configurable device that can realize nearly any digital circuit. Their flexibility allows for rapid prototyping, which has led to their popularity. An FPGA device generally consists of a two-dimensional array of logic elements that are interconnected via programmable switches (2). When an FPGA is programmed, each logic

element can implement some piece of combinatorial or sequential logic, and the switches route their inputs and outputs between them to implement the design.

The FPGA used to implement the trigger is in the Altera Cyclone family of FPGAs. Each of its logic elements consist of a four input lookup table, providing it with the capability of implementing any four input Boolean function, and a single flip-flop for sequential logic. The specific model used for the trigger has approximately 20,000 logic elements (3).

Prototyping the Trigger System

The original design of the trigger system monitored eight channels for four different user-programmed bit patterns, three of which represented a particle of interest entering the chamber with the fourth representing an undesirable condition. When a desired bit pattern is detected, a user-programmable delay time is initiated, after which a trigger signal is generated. In the case of detecting the undesirable bit pattern, or in the event of pileup (a condition where multiple particles enter the chamber in rapid succession such that the detector cannot distinguish between them), the system issues a veto, which aborts any pending trigger signals and initiates a dead time of user-programmable length. The system also acts as a scalar, counting pulses on individual channels and counting the number of occurrences of each bit pattern. The original design also included maintaining a record of detected patterns with accompanying time stamps. All communication with the trigger system (i.e. programming a bit pattern, setting the delay time, or reading out channel counts, etc.) occurs via VME, which is a standard for connecting and facilitating communication between modular electronic components.

Figure 2 shows the block diagram of the prototype trigger system. Component names prefaced with “R_” are registers that are either readable or writable through VME, indicating that

the value it contains is either a user-programmable input or user-readable output of the particular module to which the register is connected.

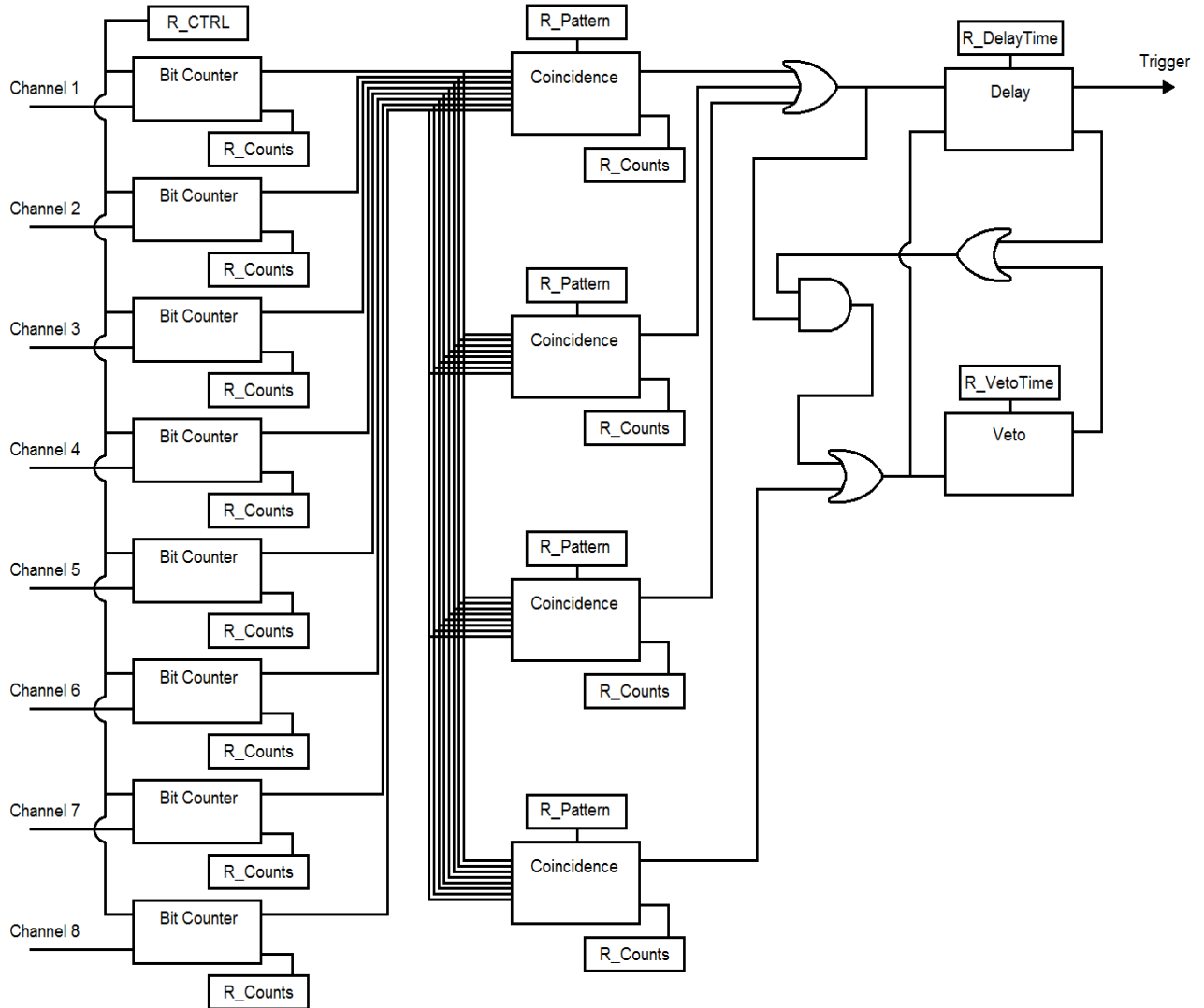


Figure 2. Block diagram of prototype trigger system.

The first module that I developed was the *bitcounter*. It monitors a single channel, detects the rising edge of incoming pulses, counts these pulses and outputs a pulse of user-programmable length. The prototype contained eight instances of the module in the system. The interface of the *bitcounter* includes single-bit inputs for the clock, an asynchronous reset, and the

channel to monitor, and a multi-bit input that specifies the desired width of the output pulse in number of clock cycles. The outputs consist of a single-bit output pulse and a 32-bit wide bus that indicates the number of incoming pulses counted.

An incoming pulse on the channel is asynchronous, so a VHDL process is used to detect its rising edge, which is approximated as the next rising edge of the clock so that the rest of the system may operate synchronously. Two important limitations arise from the manner in which incoming signals are detected; first, there is an uncertainty in the time of arrival of the pulse equal to the clock period, and second, in order to guarantee detection, an incoming pulse must have a duration at least as great as the clock period. The prototype originally used a 40 MHz clock but was upgraded to use a 100 MHz clock during development.

When a pulse is detected, three events occur; a 32-bit internal register is incremented, a counter is enabled, and the output signal goes high. The register is used to count pulses and its contents are available through the 32-bit output bus and can ultimately be read out through the VME. The counter is used to regulate the output pulse width. On each clock cycle it increments until the value, supplied through the multi-bit input, is reached, at which point, the timer is reset and disabled. The output pulse coincides with the signal that enables the timer.

The individual outputs of the eight *bitcounter* modules are grouped together in an 8-bit wide bus that is fed into four separate instances of the *coincidence* module. The *coincidence* module looks for a given bit pattern on its input bus, counts the number of occurrences of the pattern, and outputs a single bit signal, lasting a single clock cycle, when the pattern is found. The module's interface consists of single bit inputs for the clock and an asynchronous reset, eight-bit inputs for the channels to monitor, and the desired bit pattern, a single bit output indicating when the pattern has been found, and a 32-bit wide output bus for the pattern counts.

On each rising edge of the clock, the *coincidence* module compares the bit pattern on its input bus to the programmed desired bit pattern, and sets an internal signal high if they are the same and low if they are not. The rising edge of this internal signal is used to generate a pulse with pulse width equal to one clock cycle that accomplishes two tasks. First, it propagates to the single-bit output of the *coincidence* module that indicates that a bit pattern has been matched. Second, it increments a 32-bit internal register that counts the number of times the given pattern has been identified. This register drives the 32-bit wide output bus, which is available for readout through the VME.

Four instances of the *coincidence* module were incorporated into the prototype trigger, allowing it to detect four different bit patterns. In addition to bit patterns that should induce a trigger signal, the detection of certain patterns should veto the trigger if one were scheduled to be transmitted. The *coincidence* module need not be modified to accommodate these patterns; the difference in behavior is accomplished by the combinatorial logic that is driven by the module's output.

The *delay* module manages implementing the delay time between detecting an incoming desirable bit pattern and transmitting the trigger signal. The module's interface includes single-bit inputs for the clock, an asynchronous reset, and a pulse indicating that a desirable pattern has been detected, and multi-bit input buses to specify the length of the delay and the width of the trigger pulse in number of clock cycles. Its output is comprised of two single-bit outputs. One is a signal that is high while a delay is in progress and low otherwise. The other is the trigger signal.

When the *delay* module receives a pulse on its 'pattern detected' input, an internal counter is enabled, which increments on each rising edge of the clock. When the counter reaches

the number specified by the 'delay length' input, it generates a trigger signal. The pulse width of the trigger is regulated in the same manner as the output pulse of the *bitcounter* module. The output that indicates that a delay is in progress is driven by the internal signal that enables the counter.

The termination of a delay and its subsequent trigger signal when a veto condition occurs can be accomplished by using the reset input since it sets all of the internal signals and registers to a default value, which is typically zero. In the case of the *delay* module, receiving a reset signal resets and disables the delay length counter by setting the internal register that acts as the counter, and the signal that enables it, to zero.

A veto signal with pulse width equal to one clock cycle is generated whenever any of the following conditions are met: an undesirable bit pattern is detected; a desirable bit pattern is detected while the delay from a previous desirable bit pattern is in progress; a desirable bit pattern is detected during the dead time following a veto. The veto conditions are implemented through combinatorial logic driven by the signals interfacing between the various modules.

The *veto* module is similar to the *delay* module with the major differences being that it does not generate a trigger signal and that instead of managing a time delay, it manages a dead time window. Its interface includes single-bit inputs for the clock, an asynchronous reset, and a veto signal, and a 32-bit bus input that specifies the length of the dead time window in number of clock cycles. It has one single-bit output, which indicates whether or not the veto dead time window is currently active.

Upon detecting the rising edge of a veto signal, the *veto* module enables a counter that counts rising clock edges until it reaches the number indicated by the 32-bit wide input bus, after which the counter is reset and disabled. The dead time window is active while the counter is

enabled. If an additional veto signal is received while the veto window is active, the counter is reset but remains enabled, thus lengthening the dead time.

The *FIFO* module implements a first-in first-out protocol for storing and transmitting data. Its interface is comprised of single-bit inputs for the clock, an asynchronous reset, a signal to store data, and a signal to increment the read position, and a 32-bit wide input bus for data. Its sole output is a 32-bit wide bus that carries data from the FIFO to be read out from a register through the VME. The important internal signals of the FIFO module are multi-bit registers for read and write positions, and an array to store data. When the module receives a ‘store data’ pulse, it stores the current value on its 32-bit input bus to the array at the index indicated by the write position signal. The write position signal is then incremented. The 32-bit output bus is driven by the array position indexed by the current value of the read position signal. When the register containing the FIFO output is read, the module controlling VME communication transmits a pulse to the *FIFO* module’s ‘increment read position’ input. A small delay is initiated by use of an internal counter to allow the VME to complete its read cycle before incrementing the read position signal.

The prototype uses two instances of the *FIFO* module. One stores each bit pattern that is detected by the *coincidence* modules, and the other stores a timestamp indicating when the pattern was detected. The two FIFOs are synchronized such that each timestamp entry is associated with the correct bit pattern entry. Each FIFO can store up to 32 entries, each consisting of 32 bits, for a total of 128 bytes. Although it was known that a much larger sized FIFO was desired for the final version of the trigger, the size was kept small on the prototype to facilitate testing.

The *lb_int* (local bus interface) module was written by CAEN. It provides the user FPGA of their V1495 board with the ability to communicate via VME. The trigger system uses this module, but I have made the modifications necessary to incorporate more readable and writable registers.

Expanding the Design

The prototype monitors eight channels for four different bit patterns, but the final version of the system monitors 16 channels for 32 different bit patterns, 16 of which correspond to a particle of interest entering the chamber and the remaining 16 corresponding to a veto condition. In addition to expanding the number of channels and identifiable patterns, the final version also implements a few new features but loses the FIFO.

The reason the FIFO is not used in the final design is due to a conflict between the desired size of the FIFO and the capacity of the FPGA. Essentially, the *FIFO* module uses the logic elements of the FPGA to synthesize random access memory. Generating a single bit of memory requires a flip-flop, and each logic element only has a single flip-flop, so with approximately 20,000 logic elements, one could synthesize a maximum of about 2.5 kilobytes of memory. The desired size of the FIFO was 4096 entries. Two FIFOs with 4096 32-bit entries each require a total of 32 kilobytes of memory, which far exceeds the capacity of our FPGA model.

One of the additional features is the *to_digitizer* module. It outputs a 16-bit signal whenever a trigger signal is generated. This output is directed through the C port of the V1495 board, where it can be fed to an input on the digitizer, which is another electronics board produced by CAEN that resides in the same crate as the V1495. The digitizer will capture data

on this input port when it receives a trigger signal, so the 16-bit output of the *to_digitizer* module is transmitted during a window beginning five clock cycles prior to trigger generation and lasting until five clock cycles afterwards to guarantee that the digitizer will see this data on its input when it receives a trigger. The 16-bit signal is comprised of four bits to identify which pattern was detected, and 12 bits to indicate the current count of detected patterns.

The *M_of_N* module implements m of n logic such that out of n signals, m signals must be active in order for the group of n signals to be considered active. For the trigger system, n is fixed at four, but m is programmable by the user. The module takes in the outputs from the *bitcounters* of channels one through four, and produces a single output based on the value m specified by the user. The single output of this module is propagated to the input of the *coincidence* module instances instead of the individual outputs of the first four *bitcounter* instances.

Another method of increasing the number of identifiable patterns without increasing the number of instances of the coincidence module is the addition of the ability to specify wildcard bits for a given coincidence pattern. When a particular bit is set as a wildcard, its value plays no role in determining pattern detection. Only non-wildcard bits need match in order for the *coincidence* module to identify the pattern. This feature was implemented by adding an additional 16-bit input to the coincidence module that masks the channel input before comparing it to the specified pattern. Setting a bit to '0' in the bit mask marks that bit as a wildcard. Since all the registers used in the trigger system are 32-bits wide, both the coincidence pattern and its associated bit mask are specified in a single register.

Individual *coincidence* module instances may be disabled and re-enabled on the fly. The implementation uses a bit mask to mask the outputs of the *coincidence* module instances. This

bit mask can be written to a single 32-bit register, with the lower two bytes corresponding to the 16 desirable patterns, and the upper two bytes corresponding to the 16 undesirable patterns.

Testing

I developed the trigger system one module at a time. As each module was completed, I tested it on its own using the ModelSim simulation software package. The simulation is driven by scripts that specify which inputs, outputs, and internal signals to display, the value and timing of input signals, and the duration of the simulation. Figure 3 shows a sample of ModelSim's display output from simulating the *veto* module. Signals with a green arrow to the left of the blue diamond are inputs, those with green arrows to the right of the blue diamond are outputs, and those with no green arrows are internal signals. If the simulation demonstrated expected behavior for the module, I would incorporate the module into the overall design and test it on the actual v1495 hardware; otherwise, I would troubleshoot the problem and revise the code.

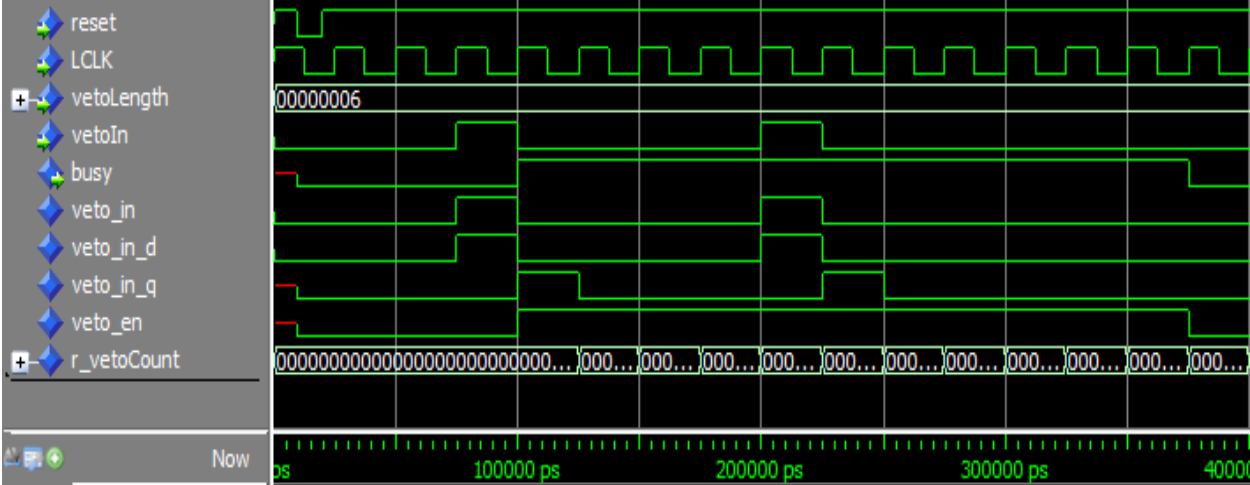


Figure 3. ModelSim display output for simulation of the *veto* module.

Most of the testing of the trigger system on the actual hardware has been relatively straightforward. Given a specific set of inputs, including the programmed patterns and delay

time in addition to the channel inputs, the system is checked for expected outputs, which typically entails verifying the characteristics of the trigger signal on an oscilloscope and reading registers through VME to ensure that they contain the correct data. Figure 4 shows an annotated sample oscilloscope trace. A NIM signal was fed into a translator to convert it to an ECL signal, which is the type accepted by the A port of the V1495 board. One input to the scope, which is annotated on Figure 3, was a signal indicating when the translator received the NIM signal, which was used as an approximation of the time that the trigger received the input. The 40 MHz clock was in use at the time of this picture, and the delay was set to 10 clock cycles. The elapsed time from receiving the input signal to generating a trigger is seen to be approximately 15 clock cycles. The additional five clock cycles are due to the fact that as the various internal signals propagate through the system, each time rising edge detection is used, it adds about a clock cycle worth of delay. Registers were written and read via software that utilizes the CAEN library.

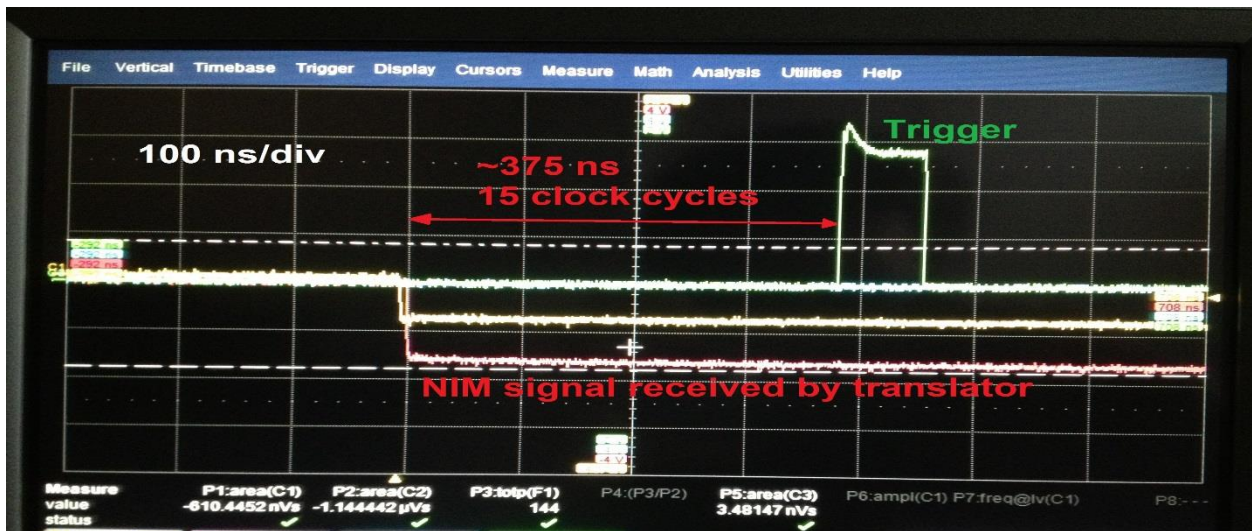


Figure 4. Oscilloscope trace from testing the trigger system.

I tested the *bitcounter* module by writing to the control register to specify the width of the output pulse, sending in a number of input pulses, observing the output pulse of appropriate

width on an oscilloscope, and then reading the output register to ensure that all pulses were detected and recorded.

I tested the addition of the *coincidence* module in a like manner. I wrote a test pattern to the appropriate register, sent in various patterns of pulses, observed the oscilloscope for an output pulse indicating that the test pattern had been identified, and then read out the registers for pulse counts on each *bitcounter* channel as well the coincidence pattern counts.

Testing the design that resulted from incorporating the *delay* module was similar, but with the addition of writing the delay time length to the appropriate register, and monitoring the oscilloscope for a trigger signal delayed by the proper length upon sending in the correct pattern of input pulses.

Adding the *veto* module necessitated that the programmed delay time be on the order of a few seconds rather than nanoseconds in order to allow me to create veto conditions after sending in a good pattern. I tested each veto condition and ensured that no trigger signal was generated. I also re-performed the tests of the system that I had performed for the delay module to ensure that the operation of the *delay* module had not been altered or interfered with by the addition of the *veto* module.

One of the more rigorous tests performed on the system was a bandwidth test to assess the rate at which it could reliably detect incoming pulses. The bandwidth test uses a program written by Dr. Kordosky, which generates a train of 200 pulses on one channel followed by a readout every 20 seconds. The pulse width and periodicity of the pulses are variable. Two variations of the test were performed; the first method fixes the period of the incoming pulses while allowing the pulse width to vary, and the second method fixes the pulse width while allowing the period to vary. For each change in the independent variable, ten trials were

performed and the results were averaged. With the 100 MHz clock, the results show a maximum bandwidth of 30 MHz and that for a pulse to be guaranteed to be detected, its width must be at least 15 ns.

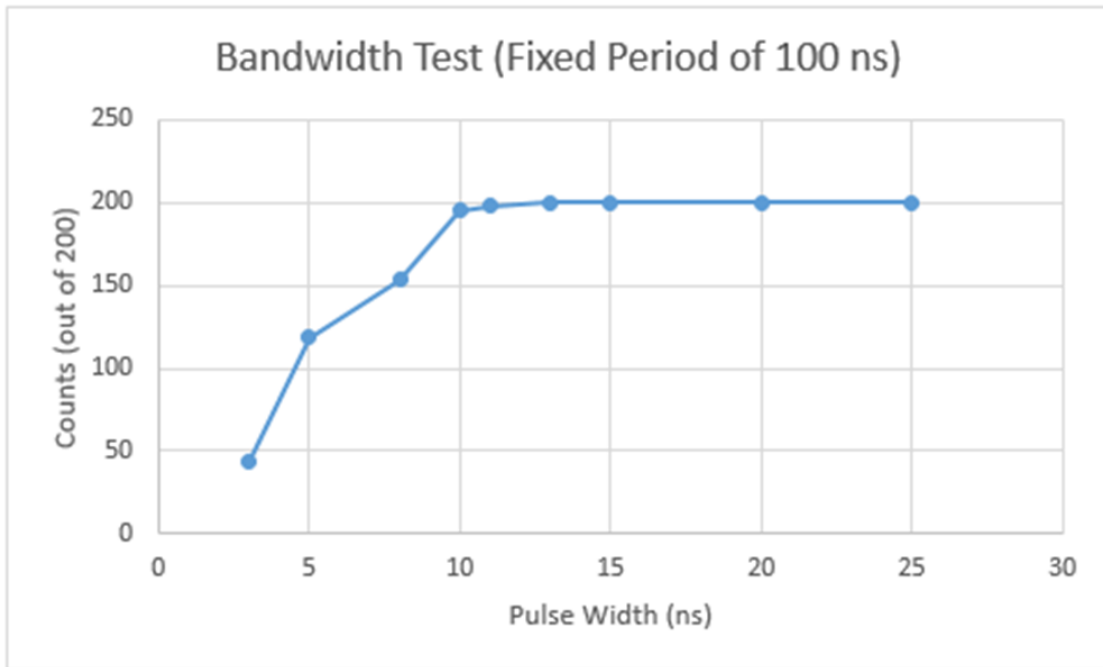


Figure 5. Bandwidth test results for pulses of fixed period with varying pulse width.

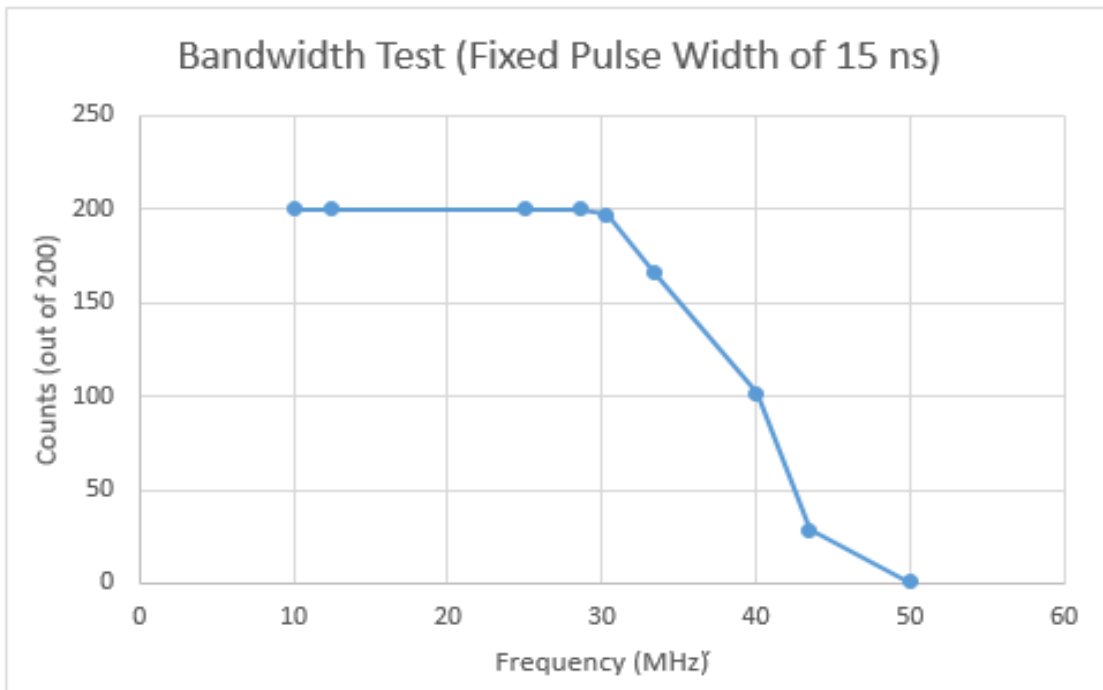


Figure 6. Bandwidth test results for pulses of fixed width with varying period.

Conclusion

The development and testing of the trigger system took place at William and Mary, but Dr. Kordosky has already loaded the trigger firmware onto a V1495 board onsite at Fermilab and done some preliminary testing of it in preparation for use in the LArIAT experiment. Beam testing of the trigger system is anticipated to occur at Fermilab this summer.

References

1. Anderson, Antonello, Baller, Bolton, Bromberg, Cavanna, Church, Edmunds, Ereditato, Farooq, Fleming, Greenlee, Guenette, Haug, Horton-Smith, James, Klein, Lang, Lathrop, Laurens, Linden, McKee, Mehdiyev, Page, Palamara, Partyka, Pordes, Rameika, Rebel, Rossi, Sanders, Soderberg, Spitz, Szec, Weber, Yang, Wongjirad, Zeller. “The ArgoNeuT Detector in the NuMI Low-Energy Beam Line at Fermilab”
arXiv:1205.6747v2
2. Chu, Pong P. *FPGA Prototyping by VHDL Examples*. John Wiley & Sons, Inc. 2008.
3. *Cyclone Device Handbook*. www.altera.com/literature/hb/cyc/cyc_c5v1.pdf

Appendix A: Using the Quartus II Software

This appendix gives step-by-step instructions for using the Quartus II software to generate firmware in raw binary format from VHDL files and a pin assignment contained in a Quartus settings file (extension .qsf).

1. Create a directory to use as the project directory and move the VHDL files and pin assignment file into it.
2. Open Quartus II.
3. Select *File* → *New Project Wizard*.
4. On page one of the new project wizard, select the project working directory, enter a name for the project, and name the top-level VHDL entity (Must match the top-level entity named in the VHDL code exactly. For the trigger, the top-level entity is “trigger_top”).
5. On page two of the new project wizard, add all of the VHDL files to the project.
6. On page three of the new project wizard, select the target FPGA device family and name. For the trigger, the device family is Cyclone, and the name is EP1C20F400C6.
7. Page four of the new project wizard can be skipped.
8. Page five of the new project wizard gives a summary of the options selected for the new project. Click *Finish* to create the Quartus project.
9. Select *Assignments* → *Import Assignments*. Select the pin assignment (.qsf) file.
10. Select *Processing* → *Start Compilation*. The *Tasks* pane on the left side of the main Quartus window will show the progress of each step involved in compilation and synthesis of the firmware.
11. Upon successful compilation and synthesis, Quartus creates a directory named *output_files* in the working directory of the project. This directory contains an SRAM object file (extension .sof), which can be converted to the raw binary file format required by the user FPGA of the v1495 board.
12. Select *File* → *Convert Programming Files*.
13. In the *Convert Programming Files* window, select *Raw Binary File* from the *Programming file type* drop-down menu and enter a name for the file you will generate.
14. In the *Input files to convert* pane of the *Convert Programming Files* window, highlight *SOF Data*, and click *Add File*.
15. Select the SRAM object file from the *output_files* directory. Click *Generate* to produce the raw binary file of the firmware.

Appendix B: Full Code Listing for Final Version of Trigger Firmware

```
-- trigger_top.vhd
-- Written by: Matthew Stephens
-- Provides the interface to the v1495, which is specified by CAEN, so many inputs
-- and outputs are unused. Interconnects all of the sub-modules to implement veto
-- logic.

library IEEE;
use IEEE.std_Logic_1164.all;
use IEEE.std_Logic_arith.all;
use IEEE.std_Logic_unsigned.all;
use work.components.all;

entity trigger_top is
    port(
        -- Front Panel Ports
        A      : IN  std_logic_vector (31 DOWNT0 0); -- In A (32 x LVDS/ECL)
        B      : IN  std_logic_vector (31 DOWNT0 0); -- In B (32 x LVDS/ECL)
        C      : OUT std_logic_vector (31 DOWNT0 0); -- Out C (32 x LVDS)
        D      : INOUT std_logic_vector (31 DOWNT0 0); -- In/Out D (I/O Expansion)
        E      : INOUT std_logic_vector (31 DOWNT0 0); -- In/Out E (I/O Expansion)
        F      : INOUT std_logic_vector (31 DOWNT0 0); -- In/Out F (I/O Expansion)
        GIN    : IN  std_logic_vector ( 1 DOWNT0 0); -- In G - LEMO (2 x NIM/TTL)
        GOUT   : OUT std_logic_vector ( 1 DOWNT0 0); -- Out G - LEMO (2 x NIM/TTL)
        -- Port Output Enable (0=Output, 1=Input)
        nOED   : OUT  std_logic;           -- Output Enable Port D (only for A395D)
        nOEE   : OUT  std_logic;           -- Output Enable Port E (only for A395D)
        nOEF   : OUT  std_logic;           -- Output Enable Port F (only for A395D)
        nOEG   : OUT  std_logic;           -- Output Enable Port G
        -- Port Level Select (0=NIM, 1=TTL)
        SELD   : OUT  std_logic;           -- Output Level Select Port D (only for A395D)
        SELE   : OUT  std_logic;           -- Output Level Select Port E (only for A395D)
        SELF   : OUT  std_logic;           -- Output Level Select Port F (only for A395D)
        SELG   : OUT  std_logic;           -- Output Level Select Port G

        -- Expansion Mezzanine Identifier:
        -- 000 : A395A (32 x IN LVDS/ECL)
        -- 001 : A395B (32 x OUT LVDS)
        -- 010 : A395C (32 x OUT ECL)
        -- 011 : A395D (8 x IN/OUT NIM/TTL)
        IDD    : IN  std_logic_vector (2 DOWNT0 0); -- Slot D
        IDE    : IN  std_logic_vector (2 DOWNT0 0); -- Slot E
        IDF    : IN  std_logic_vector (2 DOWNT0 0); -- Slot F

        -- Delay Lines
        -- 0:1 => PDL (Programmable Delay Line): Step = 0.25ns / FSR = 64ns
        -- 2:3 => FDL (Free Running Delay Line with fixed delay)
        PULSE   : IN  std_logic_vector (3 DOWNT0 0); -- Output of the delay line (0:1 => PDL; 2:3 => FDL)
        nSTART  : OUT  std_logic_vector (3 DOWNT0 2); -- Start of FDL (active low)
        START   : OUT  std_logic_vector (1 DOWNT0 0); -- Input of PDL (active high)
        DDLY    : INOUT std_logic_vector (7 DOWNT0 0); -- R/W Data for the PDL
        WR_DLY0 : OUT  std_logic;           -- Write signal for the PDL0
        WR_DLY1 : OUT  std_logic;           -- Write signal for the PDL1
        DIRDDL  : OUT  std_logic;           -- Direction of PDL data (0 => Read Dip Switches)
        -- (1 => Write from FPGA)
        nOEDDLY0 : OUT  std_logic;         -- Output Enable for PDL0 (active low)
        nOEDDLY1 : OUT  std_logic;         -- Output Enable for PDL1 (active low)

        -- LED drivers
        nLEDG   : OUT  std_logic;           -- Green (active low)
        nLEDR   : OUT  std_logic;           -- Red (active low)

        -- Spare
        --SPARE  : INOUT std_logic_vector (11 DOWNT0 0);

        -- Local Bus in/out signals
        nLBRES  : IN  std_logic;
        nBLAST  : IN  std_logic;
    );
end trigger_top;
```

```

WnR      : IN   std_logic;
nADS     : IN   std_logic;
LCLK     : IN   std_logic;
nREADY  : OUT  std_logic;
nINT     : OUT  std_logic;
LAD      : INOUT std_logic_vector (15 DOWNT0 0)
);

```

END trigger_top ;

architecture rtl of trigger_top is

```

-- =====
-- 100 MHz Clock
-- =====
signal clk100                : std_logic;

-- =====
-- Input Signals
-- =====
signal raw_input   : std_logic_vector(15 downto 0);
signal sync_input  : std_logic_vector(15 downto 0);
signal channels    : std_logic_vector(15 downto 0);

-- =====
-- Veto Logic Signals
-- =====
signal goodPatterns : std_logic_vector(15 downto 0);
signal goodMasked   : std_logic_vector(15 downto 0);
signal badPatterns  : std_logic_vector(15 downto 0);
signal badMasked    : std_logic_vector(15 downto 0);
signal goodPattIn   : std_logic;
signal badPattIn    : std_logic;
signal delayBusy    : std_logic;
signal vetoBusy     : std_logic;
signal pileup       : std_logic;
signal vetoCondition : std_logic;
signal resetDelay   : std_logic;

-- =====
-- Trigger Signal
-- =====
signal trigger_out : std_logic;

-- =====
-- To Digitizer Signal
-- =====
signal toDigitizer : std_logic_vector(15 downto 0);

-- =====
-- Readable Register Signals
-- =====
signal REG_R1 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R2 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R3 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R4 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R5 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R6 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R7 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R8 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R9 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R10 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R11 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R12 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R13 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R14 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R15 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R16 : std_logic_vector(31 downto 0) := (others => 'Z');
signal REG_R17 : std_logic_vector(31 downto 0) := (others => 'Z');

```



```

signal REG_RW36          : std_logic_vector(31 downto 0) := (others => 'Z');

-- =====
-- Reset
-- =====
signal reset_in          : std_logic;
signal reset_ext         : std_logic;

-- =====
-- Control Register Component Aliases
-- =====
alias reset_ctrl         : std_logic is REG_RW1(0);
alias outWidth_in       : std_logic_vector(1 downto 0) is REG_RW1(2 downto 1);
alias trigWidth_in      : std_logic_vector(1 downto 0) is REG_RW1(4 downto 3);
alias M                  : std_logic_vector(2 downto 0) is REG_RW1(7 downto 5);

-- =====
-- Pattern Masks Aliases
-- =====
alias goodPattMask      : std_logic_vector(15 downto 0) is REG_RW36(15 downto 0);
alias badPattMask       : std_logic_vector(15 downto 0) is REG_RW36(31 downto 16);

begin

--green LED on, red LED off
nLEDR <= not vetoBusy;
nLEDG <= not delayBusy;
--D = input, G = output
nOED <= '1';
nOEG <= '0';
--set ports D and G to NIM
SELD <= '0';
SELG <= '0';
--set the 100 MHz clock
nSTART(2) <= '0';
clk100 <= PULSE(2);

-- Resets
reset_ext <= D(2);
reset_in <= reset_ctrl and reset_ext;

-- Raw input comes directly from the A port
raw_input <= A(31 downto 16);
-- Channels 0 to 2 are set to '0' since they are not used due to MofN
-- Channel 3 is the output of the MofN module
-- Channels 4 to 15 are just the synchronous outputs of the bit counters
channels(2 downto 0) <= (others => '0');
channels(15 downto 4) <= sync_input(15 downto 4);

goodMasked <= goodPatterns and goodPattMask;
badMasked <= badPatterns and badPattMask;

goodPattIn <= goodMasked(0) or goodMasked(1) or goodMasked(2) or goodMasked(3) or
goodMasked(4) or goodMasked(5) or goodMasked(6) or goodMasked(7) or
goodMasked(8) or goodMasked(9) or goodMasked(10) or goodMasked(11) or
goodMasked(12) or goodMasked(13) or goodMasked(14) or goodMasked(15);

badPattIn <= badMasked(0) or badMasked(1) or badMasked(2) or badMasked(3) or
badMasked(4) or badMasked(5) or badMasked(6) or badMasked(7) or
badMasked(8) or badMasked(9) or badMasked(10) or badMasked(11) or
badMasked(12) or badMasked(13) or badMasked(14) or badMasked(15);

-- Veto logic
pileup <= goodPattIn and (delayBusy or vetoBusy);
vetoCondition <= badPattIn or pileup;
resetDelay <= vetoCondition or not reset_in; --Other modules use active low reset, delay needs active high reset

-- Output trigger signal via port G(0)
-- Output fast trigger via portG(1)

```

```

GOUT(0) <= trigger_out;
GOUT(1) <= goodPatIn;

-- Output to digitizer via port C
C(15 downto 0) <= toDigitizer;
C(31 downto 16) <= (others => '0');

```

```

-- =====
-- Module Instances
-- =====
-- =====
-- Channels
-- =====
i_bitcounter0 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(0),
  outWidth => outWidth_in,
  bitcount => REG_R1,
  outPulse => sync_input(0)
);

i_bitcounter1 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(1),
  outWidth => outWidth_in,
  bitcount => REG_R2,
  outPulse => sync_input(1)
);

i_bitcounter2 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(2),
  outWidth => outWidth_in,
  bitcount => REG_R3,
  outPulse => sync_input(2)
);

i_bitcounter3 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(3),
  outWidth => outWidth_in,
  bitcount => REG_R4,
  outPulse => sync_input(3)
);

i_bitcounter4 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(4),
  outWidth => outWidth_in,
  bitcount => REG_R5,
  outPulse => sync_input(4)
);

i_bitcounter5 : bitcounter
port map(
  LCLK    => clk100,
  reset   => reset_in,
  bitIn   => raw_input(5),
  outWidth => outWidth_in,
  bitcount => REG_R6,

```

```

    outPulse => sync_input(5)
);

i_bitcounter6 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(6),
    outWidth => outWidth_in,
    bitcount => REG_R7,
    outPulse => sync_input(6)
);

i_bitcounter7 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(7),
    outWidth => outWidth_in,
    bitcount => REG_R8,
    outPulse => sync_input(7)
);

i_bitcounter8 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(8),
    outWidth => outWidth_in,
    bitcount => REG_R9,
    outPulse => sync_input(8)
);

i_bitcounter9 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(9),
    outWidth => outWidth_in,
    bitcount => REG_R10,
    outPulse => sync_input(9)
);

i_bitcounter10 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(10),
    outWidth => outWidth_in,
    bitcount => REG_R11,
    outPulse => sync_input(10)
);

i_bitcounter11 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(11),
    outWidth => outWidth_in,
    bitcount => REG_R12,
    outPulse => sync_input(11)
);

i_bitcounter12 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(12),
    outWidth => outWidth_in,
    bitcount => REG_R13,

```



```

    outPulse => sync_input(12)
);

i_bitcounter13 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(13),
    outWidth => outWidth_in,
    bitcount => REG_R14,
    outPulse => sync_input(13)
);

i_bitcounter14 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(14),
    outWidth => outWidth_in,
    bitcount => REG_R15,
    outPulse => sync_input(14)
);

i_bitcounter15 : bitcounter
port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitIn   => raw_input(15),
    outWidth => outWidth_in,
    bitcount => REG_R16,
    outPulse => sync_input(15)
);

-- =====
-- M of N
-- =====
i_MofN : MofN
port map(
    a => sync_input(0),
    b => sync_input(1),
    c => sync_input(2),
    d => sync_input(3),
    M => M,
    O => channels(3)
);

-- =====
-- Good Patterns
-- =====
i_good_coincidence1 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW2(31 downto 16),
        testPattern => REG_RW2(15 downto 0),
        patternCnt => REG_R17,
        patternIn => goodPatterns(0)
    );

i_good_coincidence2 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW3(31 downto 16),
        testPattern => REG_RW3(15 downto 0),
        patternCnt => REG_R18,
        patternIn => goodPatterns(1)
    );

```

```

i_good_coincidence3 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW4(31 downto 16),
    testPattern => REG_RW4(15 downto 0),
    patternCnt => REG_R19,
    patternIn => goodPatterns(2)
  );

```

```

i_good_coincidence4 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW5(31 downto 16),
    testPattern => REG_RW5(15 downto 0),
    patternCnt => REG_R20,
    patternIn => goodPatterns(3)
  );

```

```

i_good_coincidence5 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW6(31 downto 16),
    testPattern => REG_RW6(15 downto 0),
    patternCnt => REG_R21,
    patternIn => goodPatterns(4)
  );

```

```

i_good_coincidence6 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW7(31 downto 16),
    testPattern => REG_RW7(15 downto 0),
    patternCnt => REG_R22,
    patternIn => goodPatterns(5)
  );

```

```

i_good_coincidence7 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW8(31 downto 16),
    testPattern => REG_RW8(15 downto 0),
    patternCnt => REG_R23,
    patternIn => goodPatterns(6)
  );

```

```

i_good_coincidence8 : coincidence
  port map(
    LCLK    => clk100,
    reset   => reset_in,
    bitsIn  => channels,
    bitMask => REG_RW9(31 downto 16),
    testPattern => REG_RW9(15 downto 0),
    patternCnt => REG_R24,
    patternIn => goodPatterns(7)
  );

```

```

i_good_coincidence9 : coincidence
  port map(
    LCLK    => clk100,

```

```

reset    => reset_in,
bitsIn   => channels,
bitMask  => REG_RW10(31 downto 16),
testPattern => REG_RW10(15 downto 0),
patternCnt => REG_R25,
patternIn => goodPatterns(8)
);

i_good_coincidence10 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW11(31 downto 16),
        testPattern => REG_RW11(15 downto 0),
        patternCnt => REG_R26,
        patternIn => goodPatterns(9)
    );

i_good_coincidence11 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW12(31 downto 16),
        testPattern => REG_RW12(15 downto 0),
        patternCnt => REG_R27,
        patternIn => goodPatterns(10)
    );

i_good_coincidence12 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW13(31 downto 16),
        testPattern => REG_RW13(15 downto 0),
        patternCnt => REG_R28,
        patternIn => goodPatterns(11)
    );

i_good_coincidence13 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW14(31 downto 16),
        testPattern => REG_RW14(15 downto 0),
        patternCnt => REG_R29,
        patternIn => goodPatterns(12)
    );

i_good_coincidence14 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW15(31 downto 16),
        testPattern => REG_RW15(15 downto 0),
        patternCnt => REG_R30,
        patternIn => goodPatterns(13)
    );

i_good_coincidence15 : coincidence
    port map(
        LCLK    => clk100,
        reset   => reset_in,
        bitsIn  => channels,
        bitMask => REG_RW16(31 downto 16),
        testPattern => REG_RW16(15 downto 0),

```

```

patternCnt => REG_R31,
patternIn => goodPatterns(14)
);

i_good_coincidence16 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW17(31 downto 16),
  testPattern => REG_RW17(15 downto 0),
    patternCnt => REG_R32,
    patternIn => goodPatterns(15)
  );

-- =====
-- Bad Patterns
-- =====

i_bad_coincidence1 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW18(31 downto 16),
  testPattern => REG_RW18(15 downto 0),
    patternCnt => REG_R33,
    patternIn => badPatterns(0)
  );

i_bad_coincidence2 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW19(31 downto 16),
  testPattern => REG_RW19(15 downto 0),
    patternCnt => REG_R34,
    patternIn => badPatterns(1)
  );

i_bad_coincidence3 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW20(31 downto 16),
  testPattern => REG_RW20(15 downto 0),
    patternCnt => REG_R35,
    patternIn => badPatterns(2)
  );

i_bad_coincidence4 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW21(31 downto 16),
  testPattern => REG_RW21(15 downto 0),
    patternCnt => REG_R36,
    patternIn => badPatterns(3)
  );

i_bad_coincidence5 : coincidence
  port map(
    LCLK    => clk100,
  reset    => reset_in,
  bitsIn   => channels,
  bitMask  => REG_RW22(31 downto 16),
  testPattern => REG_RW22(15 downto 0),
    patternCnt => REG_R37,

```

```

        patternIn => badPatterns(4)
    );

i_bad_coincidence6 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW23(31 downto 16),
        testPattern => REG_RW23(15 downto 0),
        patternCnt => REG_R38,
        patternIn => badPatterns(5)
    );

i_bad_coincidence7 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW24(31 downto 16),
        testPattern => REG_RW24(15 downto 0),
        patternCnt => REG_R39,
        patternIn => badPatterns(6)
    );

i_bad_coincidence8 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW25(31 downto 16),
        testPattern => REG_RW25(15 downto 0),
        patternCnt => REG_R40,
        patternIn => badPatterns(7)
    );

i_bad_coincidence9 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW26(31 downto 16),
        testPattern => REG_RW26(15 downto 0),
        patternCnt => REG_R41,
        patternIn => badPatterns(8)
    );

i_bad_coincidence10 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW27(31 downto 16),
        testPattern => REG_RW27(15 downto 0),
        patternCnt => REG_R42,
        patternIn => badPatterns(9)
    );

i_bad_coincidence11 : coincidence
    port map(
        LCLK      => clk100,
        reset     => reset_in,
        bitsIn    => channels,
        bitMask   => REG_RW28(31 downto 16),
        testPattern => REG_RW28(15 downto 0),
        patternCnt => REG_R43,
        patternIn => badPatterns(10)
    );

i_bad_coincidence12 : coincidence

```

```

        port map(
            LCLK      => clk100,
reset      => reset_in,
bitsIn     => channels,
bitMask    => REG_RW29(31 downto 16),
testPattern => REG_RW29(15 downto 0),
            patternCnt => REG_R44,
            patternIn  => badPatterns(11)
        );

i_bad_coincidence13 : coincidence
    port map(
        LCLK      => clk100,
reset      => reset_in,
bitsIn     => channels,
bitMask    => REG_RW30(31 downto 16),
testPattern => REG_RW30(15 downto 0),
            patternCnt => REG_R45,
            patternIn  => badPatterns(12)
    );

i_bad_coincidence14 : coincidence
    port map(
        LCLK      => clk100,
reset      => reset_in,
bitsIn     => channels,
bitMask    => REG_RW31(31 downto 16),
testPattern => REG_RW31(15 downto 0),
            patternCnt => REG_R46,
            patternIn  => badPatterns(13)
    );

i_bad_coincidence15 : coincidence
    port map(
        LCLK      => clk100,
reset      => reset_in,
bitsIn     => channels,
bitMask    => REG_RW32(31 downto 16),
testPattern => REG_RW32(15 downto 0),
            patternCnt => REG_R47,
            patternIn  => badPatterns(14)
    );

i_bad_coincidence16 : coincidence
    port map(
        LCLK      => clk100,
reset      => reset_in,
bitsIn     => channels,
bitMask    => REG_RW33(31 downto 16),
testPattern => REG_RW33(15 downto 0),
            patternCnt => REG_R48,
            patternIn  => badPatterns(15)
    );

-- =====
-- Delay
-- =====
    i_delay : delay
        port map(
            LCLK      => clk100,
reset      => resetDelay,
delayLength => REG_RW34,
triggerWidth => trigWidth_in,
goodPattn  => goodPattn,
busy       => delayBusy,
triggerOut  => trigger_out
        );

-- =====
-- Veto

```

```

-- =====
i_veto : veto
port map(
  LCLK      => clk100,
  reset     => reset_in,
  vetoLength => REG_RW35,
  vetoIn    => vetoCondition,
  busy      => vetoBusy
);

-- =====
-- To Digitizer
-- =====
i_to_digitizer : to_digitizer
port map(
  LCLK      => clk100,
  reset     => reset_in,
  patternVector => goodPatterns,
  delayLength => REG_RW34,
  output    => toDigitizer
);

-- =====
-- Local Bus Interface
-- =====
i_LB_INT: LB_INT
      port map (
        -- Local Bus in/out signals
        nLBRES => nLBRES,
        nBLAST => nBLAST,
        WnR    => WnR,
        nADS   => nADS,
        LCLK   => LCLK,
        nREADY => nREADY,
        nINT   => nINT,
        LAD    => LAD,

        -- Internal Registers
        REG_R1  => REG_R1,
        REG_R2  => REG_R2,
        REG_R3  => REG_R3,
        REG_R4  => REG_R4,
        REG_R5  => REG_R5,
        REG_R6  => REG_R6,
        REG_R7  => REG_R7,
        REG_R8  => REG_R8,
        REG_R9  => REG_R9,
        REG_R10 => REG_R10,
        REG_R11 => REG_R11,
        REG_R12 => REG_R12,
        REG_R13 => REG_R13,
        REG_R14 => REG_R14,
        REG_R15 => REG_R15,
        REG_R16 => REG_R16,
        REG_R17 => REG_R17,
        REG_R18 => REG_R18,
        REG_R19 => REG_R19,
        REG_R20 => REG_R20,
        REG_R21 => REG_R21,
        REG_R22 => REG_R22,
        REG_R23 => REG_R23,
        REG_R24 => REG_R24,
        REG_R25 => REG_R25,
        REG_R26 => REG_R26,
        REG_R27 => REG_R27,
        REG_R28 => REG_R28,
        REG_R29 => REG_R29,
        REG_R30 => REG_R30,
        REG_R31 => REG_R31,
        REG_R32 => REG_R32,

```

```

REG_R33 => REG_R33,
REG_R34 => REG_R34,
REG_R35 => REG_R35,
REG_R36 => REG_R36,
REG_R37 => REG_R37,
REG_R38 => REG_R38,
REG_R39 => REG_R39,
REG_R40 => REG_R40,
REG_R41 => REG_R41,
REG_R42 => REG_R42,
REG_R43 => REG_R43,
REG_R44 => REG_R44,
REG_R45 => REG_R45,
REG_R46 => REG_R46,
REG_R47 => REG_R47,
REG_R48 => REG_R48,

REG_RW1 => REG_RW1,
REG_RW2 => REG_RW2,
REG_RW3 => REG_RW3,
REG_RW4 => REG_RW4,
REG_RW5 => REG_RW5,
REG_RW6 => REG_RW6,
REG_RW7 => REG_RW7,
REG_RW8 => REG_RW8,
REG_RW9 => REG_RW9,
REG_RW10 => REG_RW10,
REG_RW11 => REG_RW11,
REG_RW12 => REG_RW12,
REG_RW13 => REG_RW13,
REG_RW14 => REG_RW14,
REG_RW15 => REG_RW15,
REG_RW16 => REG_RW16,
REG_RW17 => REG_RW17,
REG_RW18 => REG_RW18,
REG_RW19 => REG_RW19,
REG_RW20 => REG_RW20,
REG_RW21 => REG_RW21,
REG_RW22 => REG_RW22,
REG_RW23 => REG_RW23,
REG_RW24 => REG_RW24,
REG_RW25 => REG_RW25,
REG_RW26 => REG_RW26,
REG_RW27 => REG_RW27,
REG_RW28 => REG_RW28,
REG_RW29 => REG_RW29,
REG_RW30 => REG_RW30,
REG_RW31 => REG_RW31,
REG_RW32 => REG_RW32,
REG_RW33 => REG_RW33,
REG_RW34 => REG_RW34,
REG_RW35 => REG_RW35,
REG_RW36 => REG_RW36

```

```
);
```

```
end rtl;
```

```

-- bitcounter.vhd
-- Written by: Matthew Stephens
-- This module looks at a single channel and accomplishes the following
-- two tasks: it counts the pulses on the channel, and it outputs a
-- pulse of programmable width for every incoming pulse.

```

```

library ieee;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;

```



```

ENTITY bitcounter is
port(
  LCLK   : in    std_logic;
  reset  : in    std_logic;
  bitIn  : in    std_logic;
  outWidth : in  std_logic_vector(1 downto 0);
         bitcount : out  std_logic_vector(31 downto 0);
         outPulse : out  std_logic
);
END bitcounter;

```

ARCHITECTURE rtl of bitcounter is

```

signal bitIn_q    : std_logic_vector(1 downto 0);
signal bitpulse   : std_logic;
signal r_outPulse : std_logic;
signal r_outPulseCnt : std_logic_vector(1 downto 0);
signal r_bitcount : std_logic_vector(31 downto 0);

```

BEGIN

```

bitpulse <= bitIn_q(0) and (not bitIn_q(1)); -- detects rising edge of bitIn
outPulse <= r_outPulse;
bitcount <= r_bitcount;

```

```

-- This process implements a D flip-flop using the bitIn input to
-- aid in detecting the rising edge of bitIn
bitIn_DFF : process(reset, LCLK)

```

```

BEGIN
  if (reset = '0') then
    bitIn_q <= "00";
  elsif (LCLK'event and LCLK = '1') then
    bitIn_q(0) <= bitIn;
    bitIn_q(1) <= bitIn_q(0);
  end if;
END process bitIn_DFF;

```

```

-- This process counts the rising edges of the bitIn input and
-- stores the number in the register r_bitcount

```

```

bitIn_counter : process(reset, LCLK)
BEGIN
  if (reset = '0') then
    r_bitcount <= (others => '0');
  elsif (LCLK'event and LCLK = '1') then
    if (bitpulse = '1') then
      r_bitcount <= r_bitcount + 1;
    end if;
  end if;
END process bitIn_counter;

```

```

-- This process controls the output pulse. It is set high on the
-- rising edge of the clock following the detection of a pulse on
-- the input channel, and it lasts for the number of clock cycles
-- specified in the outWidth input.

```

```

outPulse_ctrl : process(reset, LCLK)
BEGIN
  if (reset = '0') then
    r_outPulse <= '0';
  elsif (LCLK'event and LCLK = '1') then
    if (bitpulse = '1') then
      r_outPulse <= '1';
    elsif (r_outPulseCnt = outWidth - 1) then
      r_outPulse <= '0';
    end if;
  end if;
END process outPulse_ctrl;

```

```

-- This controls the width of the output pulse by counting clock
-- cycles and storing the count in the register r_outPulseCnt, which

```

```

-- is used by the outPulse_ctrl process. Counting is enabled only
-- when r_outPulse is high.
outWidth_ctrl : process(reset, LCLK)
BEGIN
  if (reset = '0') then
    r_outPulseCnt <= (others => '0');
  elsif (LCLK'event and LCLK = '1') then
    if (r_outPulse = '1') then
      r_outPulseCnt <= r_outPulseCnt + 1;
    else
      r_outPulseCnt <= (others => '0');
    end if;
  end if;
END process outWidth_ctrl;

END rtl;

-- mofn.vhd
-- Written by: Matthew Stephens

library IEEE;
use IEEE.std_Logic_1164.all;
use IEEE.std_Logic_arith.all;
use IEEE.std_Logic_unsigned.all;

ENTITY MofN is
  port(
    a : in std_logic;
    b : in std_logic;
    c : in std_logic;
    d : in std_logic;
    M : in std_logic_vector(2 downto 0);
    O : out std_logic
  );
END MofN;

architecture rtl of MofN is

  signal one_of_four : std_logic;
  signal two_of_four : std_logic;
  signal three_of_four : std_logic;
  signal four_of_four : std_logic;

begin

  one_of_four <= a or b or c or d;
  two_of_four <= (a and b) or (a and c) or (a and d) or
    (b and c) or (b and d) or (c and d);
  three_of_four <= (a and b and c) or (a and b and d) or (a and c and d) or (b and c and d);
  four_of_four <= a and b and c and d;

  O <= one_of_four when M = "001" else
    two_of_four when M = "010" else
    three_of_four when M = "011" else
    four_of_four when M = "100" else
    '0';

end rtl;

-- coincidence.vhd
-- Written by: Matthew Stephens
-- This module receives the synchronous outputs from 16 instances of the
-- bit counter module. This 16-bit signal is masked before being compared
-- to the 16-bit user-defined pattern. The purpose of the mask is to mask
-- any channel which is designated as "don't care." The module outputs a
-- signal lasting one clock cycle when the masked incoming bits match
-- the coincidence pattern. The number of pattern detections is counted.

library ieee;

```

```

use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;

```

ENTITY coincidence is

```

port(
  LCLK      : in    std_logic;
  reset     : in    std_logic;
  bitsIn    : in    std_logic_vector(15 downto 0);
  bitMask   : in    std_logic_vector(15 downto 0);
  testPattern : in   std_logic_vector(15 downto 0);
  patternCnt : out   std_logic_vector(31 downto 0);
  patternIn : out   std_logic
);
END coincidence;

```

ARCHITECTURE rtl of coincidence is

```

signal maskedBitsIn : std_logic_vector(15 downto 0);
signal pattIn_d      : std_logic;
signal pattIn_q      : std_logic;
signal r_patternIn   : std_logic;
signal r_patternCnt  : std_logic_vector(31 downto 0);

```

BEGIN

```

maskedBitsIn <= bitsIn and bitMask;
patternIn <= r_patternIn;
patternCnt <= r_patternCnt;
r_patternIn <= pattIn_d and not pattIn_q; -- detects rising edge of pattern

```

-- Checks if the incoming pattern matches the programmable test

-- pattern

checkPattern : process(reset, LCLK)

BEGIN

```

if (reset = '0') then
  pattIn_d <= '0';
elsif (LCLK'event and LCLK = '1') then
  if (maskedBitsIn = testPattern) then
    pattIn_d <= '1';
  else
    pattIn_d <= '0';
  end if;
end if;

```

END process checkPattern;

-- This process implements a D flip-flop to aid in detecting the

-- rising edge of an incoming pattern that matches the test pattern

pattIn_DFF : process(reset, LCLK)

BEGIN

```

if (reset = '0') then
  pattIn_q <= '0';
elsif (LCLK'event and LCLK = '1') then
  pattIn_q <= pattIn_d;
end if;

```

END process pattIn_DFF;

-- This process counts the rising edges of the r_patternIn signal

-- and stores the number in the register r_patternCnt

pattIn_counter : process(reset, LCLK, r_patternIn)

BEGIN

```

if (reset = '0') then
  r_patternCnt <= (others => '0');
elsif (LCLK'event and LCLK = '1') then
  if (r_patternIn = '1') then
    r_patternCnt <= r_patternCnt + 1;
  end if;
end if;

```

END process pattIn_counter;

```
END rtl;
```

```
-- delay.vhd
-- Written by: Matthew Stephens
-- This module implements a programmable delay between an incoming
-- recognized pattern and the output of a trigger signal. The width
-- of the trigger signal is also programmable. If a reset occurs
-- during the delay time, the delay is terminated, and an output
-- trigger signal is not generated. The logic handling the reset
-- conditions occurs outside of this module.
-- Modified: (August 6, 2013) Implemented a flip flop of the reset
-- signal to ensure that the reset signal lasts at least one clock
-- cycle. This was done to fix the problem that a good pattern
-- coming in during the delay time of another good pattern did not
-- cause the delay to terminate.
```

```
library ieee;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;
```

```
ENTITY delay is
port(
  LCLK      : in  std_logic;
  reset     : in  std_logic;
  delayLength : in  std_logic_vector(31 downto 0);
  triggerWidth : in  std_logic_vector(1 downto 0);
  goodPattIn  : in  std_logic;
  busy       : out std_logic;
  triggerOut  : out std_logic
);
END delay;
```

```
ARCHITECTURE rtl of delay is
```

```
  signal delay_en      : std_logic;
  signal delayComplete : std_logic;
  signal r_delayCount  : std_logic_vector(31 downto 0);
  signal r_trigger     : std_logic;
  signal r_trigCount   : std_logic_vector(1 downto 0);
  signal resetDelay    : std_logic;
  signal resetDelay_q  : std_logic;
```

```
BEGIN
```

```
  busy <= delay_en or delayComplete or r_trigger;
  triggerOut <= r_trigger;
  resetDelay <= resetDelay_q;
```

```
-- This process controls enabling and disabling the delay timer count.
enableDelay : process(resetDelay, LCLK)
```

```
BEGIN
  if (resetDelay = '1') then
    delay_en <= '0';
    delayComplete <= '0';
  elsif (LCLK'event and LCLK = '1') then
    if (goodPattIn = '1') then
      delay_en <= '1';
    elsif (r_delayCount = delayLength - 1) then
      delay_en <= '0';
      delayComplete <= '1';
    else
      delayComplete <= '0';
    end if;
  end if;
END process enableDelay;
```

```

-- This process ensures that the resetDelay signal lasts at least
-- one clock cycle.
delayResetDFF : process(resetDelay, LCLK)
BEGIN
    if (resetDelay = '1') then
        resetDelay_q <= '0';
    elsif (LCLK'event and LCLK = '1') then
        resetDelay_q <= reset;
    end if;
END process delayResetDFF;

-- This process increments the delay counter on each rising edge of
-- the clock if the delay timer is enabled. The timer is set to 0
-- on reset.
delayTimer : process(resetDelay, LCLK)
BEGIN
    if (resetDelay = '1') then
        r_delayCount <= (others => '0');
    elsif (LCLK'event and LCLK = '1') then
        if (delay_en = '1') then
            r_delayCount <= r_delayCount + 1;
        else
            r_delayCount <= (others => '0');
        end if;
    end if;
END process delayTimer;

-- This process controls the output of the trigger signal. A
-- trigger is generated only if the delay timer completed its
-- count successfully.
trigOutCtrl : process(resetDelay, LCLK)
BEGIN
    if (resetDelay = '1') then
        r_trigger <= '0';
    elsif (LCLK'event and LCLK = '1') then
        if (delayComplete = '1') then
            r_trigger <= '1';
        elsif (r_trigCount = triggerWidth - 1) then
            r_trigger <= '0';
        end if;
    end if;
END process trigOutCtrl;

-- This process controls the width of the output trigger.
trigTimer : process(resetDelay, LCLK)
BEGIN
    if (resetDelay = '1') then
        r_trigCount <= (others => '0');
    elsif (LCLK'event and LCLK = '1') then
        if (r_trigger = '1') then
            r_trigCount <= r_trigCount + 1;
        else
            r_trigCount <= (others => '0');
        end if;
    end if;
END process trigTimer;

END rtl;

-- veto.vhd
-- Written by: Matthew Stephens
-- This module handles the delay time associated with veto signals. The
-- conditions for a veto signal are the same as those for a reset in the
-- delay module.

library ieee;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;

```

```
use IEEE.Std_Logic_unsigned.all;
```

```
ENTITY veto is
port(
  LCLK      : in  std_logic;
  reset     : in  std_logic;
  vetoLength : in  std_logic_vector(31 downto 0);
  vetoIn    : in  std_logic;
  busy      : out std_logic
);
END veto;
```

```
ARCHITECTURE rtl of veto is
```

```
signal veto_in      : std_logic;
signal veto_in_d    : std_logic;
signal veto_in_q    : std_logic;
signal veto_en      : std_logic;
signal r_vetoCount  : std_logic_vector(31 downto 0);
```

```
BEGIN
```

```
veto_in <= veto_in_d and (not veto_in_q);
busy <= veto_en;
```

```
-- This process aids in detecting the rising edge of an incoming veto
-- signal.
```

```
vetoInDFF : process(reset, LCLK)
```

```
BEGIN
```

```
  if (reset = '0') then
    veto_in_d <= '0';
    veto_in_q <= '0';
  elsif (LCLK'event and LCLK = '1') then
    veto_in_d <= vetoIn;
    veto_in_q <= veto_in_d;
  end if;
END process vetoInDFF;
```

```
-- This process controls enabling and disabling the veto timer count.
```

```
enableVeto : process(reset, LCLK)
```

```
BEGIN
```

```
  if (reset = '0') then
    veto_en <= '0';
  elsif (LCLK'event and LCLK = '1') then
    if (veto_in = '1') then
      veto_en <= '1';
    elsif (r_vetoCount = vetoLength - 1) then
      veto_en <= '0';
    end if;
  end if;
END process enableVeto;
```

```
-- This process increments the veto counter on each rising edge of
-- the clock if the veto timer is enabled. If a new veto pulse is
-- detected, the timer is reset to 0 but counting is still enabled.
```

```
vetoTimer : process(reset, LCLK)
```

```
BEGIN
```

```
  if (reset = '0') then
    r_vetoCount <= (others => '0');
  elsif (LCLK'event and LCLK = '1') then
    if (veto_in = '1') then
      r_vetoCount <= (others => '0');
    elsif (veto_en = '1') then
      r_vetoCount <= r_vetoCount + 1;
    else
      r_vetoCount <= (others => '0');
    end if;
  end if;
END process vetoTimer;
```

```

END process vetoTimer;

END rtl;

-- todigitizer.vhd
-- Written by: Matthew Stephens
-- This module transmits a 16-bit signal to the digitizer when a trigger is
-- generated. The 16 bits of the output consist of: 4 bits indicating which
-- pattern caused the trigger, and 12 bits that gives the current count of
-- patterns detected.

library ieee;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use IEEE.numeric_std.all;

ENTITY to_digitizer is
port(
  LCLK      : in  std_logic;
  reset     : in  std_logic;
  patternVector : in std_logic_vector(15 downto 0);
  delayLength : in std_logic_vector(31 downto 0);
  output    : out std_logic_vector(15 downto 0)
);
END to_digitizer;

ARCHITECTURE rtl of to_digitizer is

  signal patternID    : std_logic_vector(3 downto 0);
  signal patternCnt   : std_logic_vector(11 downto 0);
  signal transmitting : std_logic;
  signal anyPatternIn : std_logic;
  signal r_output     : std_logic_vector(15 downto 0);
  signal delayCounter : std_logic_vector(31 downto 0);
  signal delayCntEn   : std_logic;

BEGIN

  r_output(11 downto 0) <= patternCnt;
  r_output(15 downto 12) <= patternID;

  output <= r_output when transmitting = '1' else (others => '0');

  anyPatternIn <= patternVector(0) or patternVector(1) or
    patternVector(2) or patternVector(3) or
    patternVector(4) or patternVector(5) or
    patternVector(6) or patternVector(7) or
    patternVector(8) or patternVector(9) or
    patternVector(10) or patternVector(11) or
    patternVector(12) or patternVector(13) or
    patternVector(14) or patternVector(15);

  setPatternID : process(reset, LCLK)
  BEGIN
    if (reset = '0') then
      patternID <= "ZZZZ";
    elsif (LCLK'event and LCLK = '1') then
      if (patternVector(0) = '1') then
        patternID <= "0000";
      elsif (patternVector(1) = '1') then
        patternID <= "0001";
      elsif (patternVector(2) = '1') then
        patternID <= "0010";
      elsif (patternVector(3) = '1') then
        patternID <= "0011";
      elsif (patternVector(4) = '1') then
        patternID <= "0100";
      end if;
    end if;
  end process;

```

```

elseif (patternVector(5) = '1') then
    patternID <= "0101";
elseif (patternVector(6) = '1') then
    patternID <= "0110";
elseif (patternVector(7) = '1') then
    patternID <= "0111";
elseif (patternVector(8) = '1') then
    patternID <= "1000";
elseif (patternVector(9) = '1') then
    patternID <= "1001";
elseif (patternVector(10) = '1') then
    patternID <= "1010";
elseif (patternVector(11) = '1') then
    patternID <= "1011";
elseif (patternVector(12) = '1') then
    patternID <= "1100";
elseif (patternVector(13) = '1') then
    patternID <= "1101";
elseif (patternVector(14) = '1') then
    patternID <= "1110";
elseif (patternVector(15) = '1') then
    patternID <= "1111";
end if;
end if;
END process setPatternID;

countPatterns : process(reset, LCLK)
BEGIN
    if (reset = '0') then
        patternCnt <= (others => '0');
    elseif (LCLK'event and LCLK = '1') then
        if (anyPatternIn = '1') then
            patternCnt <= patternCnt + 1;
        end if;
    end if;
END process countPatterns;

enableDelayTimer : process(reset, LCLK)
BEGIN
    if (reset = '0') then
        delayCntEn <= '0';
    elseif (LCLK'event and LCLK = '1') then
        if (anyPatternIn = '1') then
            delayCntEn <= '1';
        elseif (delayCounter = delayLength + 5) then
            delayCntEn <= '0';
        end if;
    end if;
END process enableDelayTimer;

delayTimer : process(reset, LCLK)
BEGIN
    if (reset = '0') then
        delayCounter <= (others => '0');
    elseif (LCLK'event and LCLK = '1') then
        if (delayCntEn = '1') then
            delayCounter <= delayCounter + 1;
        end if;
        if (delayCounter = delayLength + 5) then
            delayCounter <= (others => '0');
        end if;
    end if;
END process delayTimer;

transmit : process(reset, LCLK)
BEGIN
    if (reset = '0') then
        transmitting <= '0';
    elseif (LCLK'event and LCLK = '1') then
        if (delayCounter = delayLength - 5) then

```



```

    transmitting <= '1';
    elsif (delayCounter = delayLength + 5) then
        transmitting <= '0';
    end if;
end if;
END process transmit;

```

END rtl;

```

-- lb_int.vhd
-- Modified by: Matthew Stephens

```

```

-----
-- *****
-- Company:    CAEN SpA - Viareggio - Italy
-- Model:     V1495 - Multipurpose Programmable Trigger Unit
-- Device:    ALTERA EP1C4F400C6
-- Author:    Carlo Tintori
-- Date:     May 26th, 2010
-----
-- Module:    lb_int
-- Description: Local Bus interface
-- *****

```

```

library ieee;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_unsigned.all;

```

```

ENTITY LB_INT is
    port(
        -- Local Bus in/out signals
        nLBRES   : in    std_logic;
        nBLAST   : in    std_logic;
        WnR      : in    std_logic;
        nADS     : in    std_logic;
        LCLK     : in    std_logic;
        nREADY   : out   std_logic;
        nLNT     : out   std_logic;
        LAD      : inout std_logic_vector(15 DOWNTO 0);

        -- Internal Registers
        REG_R1   : in std_logic_vector(31 downto 0);
        REG_R2   : in std_logic_vector(31 downto 0);
        REG_R3   : in std_logic_vector(31 downto 0);
        REG_R4   : in std_logic_vector(31 downto 0);
        REG_R5   : in std_logic_vector(31 downto 0);
        REG_R6   : in std_logic_vector(31 downto 0);
        REG_R7   : in std_logic_vector(31 downto 0);
        REG_R8   : in std_logic_vector(31 downto 0);
        REG_R9   : in std_logic_vector(31 downto 0);
        REG_R10  : in std_logic_vector(31 downto 0);
        REG_R11  : in std_logic_vector(31 downto 0);
        REG_R12  : in std_logic_vector(31 downto 0);
        REG_R13  : in std_logic_vector(31 downto 0);
        REG_R14  : in std_logic_vector(31 downto 0);
        REG_R15  : in std_logic_vector(31 downto 0);
        REG_R16  : in std_logic_vector(31 downto 0);
        REG_R17  : in std_logic_vector(31 downto 0);
        REG_R18  : in std_logic_vector(31 downto 0);
        REG_R19  : in std_logic_vector(31 downto 0);
        REG_R20  : in std_logic_vector(31 downto 0);
        REG_R21  : in std_logic_vector(31 downto 0);
        REG_R22  : in std_logic_vector(31 downto 0);
        REG_R23  : in std_logic_vector(31 downto 0);
        REG_R24  : in std_logic_vector(31 downto 0);
        REG_R25  : in std_logic_vector(31 downto 0);
        REG_R26  : in std_logic_vector(31 downto 0);
        REG_R27  : in std_logic_vector(31 downto 0);
    );

```

```

REG_R28      : in std_logic_vector(31 downto 0);
REG_R29 : in std_logic_vector(31 downto 0);
REG_R30 : in std_logic_vector(31 downto 0);
REG_R31 : in std_logic_vector(31 downto 0);
REG_R32 : in std_logic_vector(31 downto 0);
REG_R33 : in std_logic_vector(31 downto 0);
REG_R34 : in std_logic_vector(31 downto 0);
REG_R35 : in std_logic_vector(31 downto 0);
REG_R36 : in std_logic_vector(31 downto 0);
REG_R37 : in std_logic_vector(31 downto 0);
REG_R38 : in std_logic_vector(31 downto 0);
REG_R39 : in std_logic_vector(31 downto 0);
REG_R40 : in std_logic_vector(31 downto 0);
REG_R41 : in std_logic_vector(31 downto 0);
REG_R42 : in std_logic_vector(31 downto 0);
REG_R43 : in std_logic_vector(31 downto 0);
REG_R44 : in std_logic_vector(31 downto 0);
REG_R45 : in std_logic_vector(31 downto 0);
REG_R46 : in std_logic_vector(31 downto 0);
REG_R47 : in std_logic_vector(31 downto 0);
REG_R48 : in std_logic_vector(31 downto 0);

```

```

REG_RW1 : buffer std_logic_vector(31 downto 0);
REG_RW2 : buffer std_logic_vector(31 downto 0);
REG_RW3 : buffer std_logic_vector(31 downto 0);
REG_RW4 : buffer std_logic_vector(31 downto 0);
REG_RW5 : buffer std_logic_vector(31 downto 0);
REG_RW6 : buffer std_logic_vector(31 downto 0);
REG_RW7 : buffer std_logic_vector(31 downto 0);
REG_RW8 : buffer std_logic_vector(31 downto 0);
REG_RW9 : buffer std_logic_vector(31 downto 0);
REG_RW10 : buffer std_logic_vector(31 downto 0);
REG_RW11 : buffer std_logic_vector(31 downto 0);
REG_RW12 : buffer std_logic_vector(31 downto 0);
REG_RW13 : buffer std_logic_vector(31 downto 0);
REG_RW14 : buffer std_logic_vector(31 downto 0);
REG_RW15 : buffer std_logic_vector(31 downto 0);
REG_RW16 : buffer std_logic_vector(31 downto 0);
REG_RW17 : buffer std_logic_vector(31 downto 0);
REG_RW18 : buffer std_logic_vector(31 downto 0);
REG_RW19 : buffer std_logic_vector(31 downto 0);
REG_RW20 : buffer std_logic_vector(31 downto 0);
REG_RW21 : buffer std_logic_vector(31 downto 0);
REG_RW22 : buffer std_logic_vector(31 downto 0);
REG_RW23 : buffer std_logic_vector(31 downto 0);
REG_RW24 : buffer std_logic_vector(31 downto 0);
REG_RW25 : buffer std_logic_vector(31 downto 0);
REG_RW26 : buffer std_logic_vector(31 downto 0);
REG_RW27 : buffer std_logic_vector(31 downto 0);
REG_RW28 : buffer std_logic_vector(31 downto 0);
REG_RW29 : buffer std_logic_vector(31 downto 0);
REG_RW30 : buffer std_logic_vector(31 downto 0);
REG_RW31 : buffer std_logic_vector(31 downto 0);
REG_RW32 : buffer std_logic_vector(31 downto 0);
REG_RW33 : buffer std_logic_vector(31 downto 0);
REG_RW34 : buffer std_logic_vector(31 downto 0);
REG_RW35 : buffer std_logic_vector(31 downto 0);
REG_RW36 : buffer std_logic_vector(31 downto 0)

```

```
);
```

```
END LB_INT;
```

ARCHITECTURE rtl of LB_INT is

```

-- States of the finite state machine
type LBSTATE_type is (LBIDLE, LBWRITEL, LBWRITEH, LBREADL, LBREADH);
signal LBSTATE : LBSTATE_type;

```

```

-- Output Enable of the LAD bus (from User to Vme)

```

```

signal LADoe    : std_logic;
-- Data Output to the local bus
signal LADout   : std_logic_vector(15 downto 0);
-- Lower 16 bits of the 32 bit data
signal DTL      : std_logic_vector(15 downto 0);
-- Address latched from the LAD bus
signal ADDR     : std_logic_vector(15 downto 0);

-- Register Address Map
constant A_REG_R1  : std_logic_vector(15 downto 0) := X"1030";
constant A_REG_R2  : std_logic_vector(15 downto 0) := X"1034";
constant A_REG_R3  : std_logic_vector(15 downto 0) := X"1038";
constant A_REG_R4  : std_logic_vector(15 downto 0) := X"103C";
constant A_REG_R5  : std_logic_vector(15 downto 0) := X"1040";
constant A_REG_R6  : std_logic_vector(15 downto 0) := X"1044";
constant A_REG_R7  : std_logic_vector(15 downto 0) := X"1048";
constant A_REG_R8  : std_logic_vector(15 downto 0) := X"104C";
constant A_REG_R9  : std_logic_vector(15 downto 0) := X"1050";
constant A_REG_R10 : std_logic_vector(15 downto 0) := X"1054";
constant A_REG_R11 : std_logic_vector(15 downto 0) := X"1058";
constant A_REG_R12 : std_logic_vector(15 downto 0) := X"105C";
constant A_REG_R13 : std_logic_vector(15 downto 0) := X"1060";
constant A_REG_R14 : std_logic_vector(15 downto 0) := X"1064";
constant A_REG_R15 : std_logic_vector(15 downto 0) := X"1068";
constant A_REG_R16 : std_logic_vector(15 downto 0) := X"106C";
constant A_REG_R17 : std_logic_vector(15 downto 0) := X"1070";
constant A_REG_R18 : std_logic_vector(15 downto 0) := X"1074";
constant A_REG_R19 : std_logic_vector(15 downto 0) := X"1078";
constant A_REG_R20 : std_logic_vector(15 downto 0) := X"107C";
constant A_REG_R21 : std_logic_vector(15 downto 0) := X"1080";
constant A_REG_R22 : std_logic_vector(15 downto 0) := X"1084";
constant A_REG_R23 : std_logic_vector(15 downto 0) := X"1088";
constant A_REG_R24 : std_logic_vector(15 downto 0) := X"108C";
constant A_REG_R25 : std_logic_vector(15 downto 0) := X"1090";
constant A_REG_R26 : std_logic_vector(15 downto 0) := X"1094";
constant A_REG_R27 : std_logic_vector(15 downto 0) := X"1098";
constant A_REG_R28 : std_logic_vector(15 downto 0) := X"109C";
constant A_REG_R29 : std_logic_vector(15 downto 0) := X"10A0";
constant A_REG_R30 : std_logic_vector(15 downto 0) := X"10A4";
constant A_REG_R31 : std_logic_vector(15 downto 0) := X"10A8";
constant A_REG_R32 : std_logic_vector(15 downto 0) := X"10AC";

constant A_REG_R33 : std_logic_vector(15 downto 0) := X"10B0";
constant A_REG_R34 : std_logic_vector(15 downto 0) := X"10B4";
constant A_REG_R35 : std_logic_vector(15 downto 0) := X"10B8";
constant A_REG_R36 : std_logic_vector(15 downto 0) := X"10BC";
constant A_REG_R37 : std_logic_vector(15 downto 0) := X"10C0";
constant A_REG_R38 : std_logic_vector(15 downto 0) := X"10C4";
constant A_REG_R39 : std_logic_vector(15 downto 0) := X"10C8";
constant A_REG_R40 : std_logic_vector(15 downto 0) := X"10CC";
constant A_REG_R41 : std_logic_vector(15 downto 0) := X"10D0";
constant A_REG_R42 : std_logic_vector(15 downto 0) := X"10D4";
constant A_REG_R43 : std_logic_vector(15 downto 0) := X"10D8";
constant A_REG_R44 : std_logic_vector(15 downto 0) := X"10DC";
constant A_REG_R45 : std_logic_vector(15 downto 0) := X"10E0";
constant A_REG_R46 : std_logic_vector(15 downto 0) := X"10E4";
constant A_REG_R47 : std_logic_vector(15 downto 0) := X"10E8";
constant A_REG_R48 : std_logic_vector(15 downto 0) := X"10EC";

constant A_REG_RW1 : std_logic_vector(15 downto 0) := X"1100";
constant A_REG_RW2 : std_logic_vector(15 downto 0) := X"1104";
constant A_REG_RW3 : std_logic_vector(15 downto 0) := X"1108";
constant A_REG_RW4 : std_logic_vector(15 downto 0) := X"110C";
constant A_REG_RW5 : std_logic_vector(15 downto 0) := X"1110";
constant A_REG_RW6 : std_logic_vector(15 downto 0) := X"1114";
constant A_REG_RW7 : std_logic_vector(15 downto 0) := X"1118";
constant A_REG_RW8 : std_logic_vector(15 downto 0) := X"111C";
constant A_REG_RW9 : std_logic_vector(15 downto 0) := X"1120";
constant A_REG_RW10 : std_logic_vector(15 downto 0) := X"1124";

```

```

constant A_REG_RW11 : std_logic_vector(15 downto 0) := X"1128";
constant A_REG_RW12 : std_logic_vector(15 downto 0) := X"112C";
constant A_REG_RW13 : std_logic_vector(15 downto 0) := X"1130";
constant A_REG_RW14 : std_logic_vector(15 downto 0) := X"1134";
constant A_REG_RW15 : std_logic_vector(15 downto 0) := X"1138";
constant A_REG_RW16 : std_logic_vector(15 downto 0) := X"113C";
constant A_REG_RW17 : std_logic_vector(15 downto 0) := X"1140";

constant A_REG_RW18 : std_logic_vector(15 downto 0) := X"1144";
constant A_REG_RW19 : std_logic_vector(15 downto 0) := X"1148";
constant A_REG_RW20 : std_logic_vector(15 downto 0) := X"114C";
constant A_REG_RW21 : std_logic_vector(15 downto 0) := X"1150";
constant A_REG_RW22 : std_logic_vector(15 downto 0) := X"1154";
constant A_REG_RW23 : std_logic_vector(15 downto 0) := X"1158";
constant A_REG_RW24 : std_logic_vector(15 downto 0) := X"115C";
constant A_REG_RW25 : std_logic_vector(15 downto 0) := X"1160";
constant A_REG_RW26 : std_logic_vector(15 downto 0) := X"1164";
constant A_REG_RW27 : std_logic_vector(15 downto 0) := X"1168";
constant A_REG_RW28 : std_logic_vector(15 downto 0) := X"116C";
constant A_REG_RW29 : std_logic_vector(15 downto 0) := X"1170";
constant A_REG_RW30 : std_logic_vector(15 downto 0) := X"1174";
constant A_REG_RW31 : std_logic_vector(15 downto 0) := X"1178";
constant A_REG_RW32 : std_logic_vector(15 downto 0) := X"117C";
constant A_REG_RW33 : std_logic_vector(15 downto 0) := X"1180";

constant A_REG_RW34 : std_logic_vector(15 downto 0) := X"1184";
constant A_REG_RW35 : std_logic_vector(15 downto 0) := X"1188";
constant A_REG_RW36 : std_logic_vector(15 downto 0) := X"118C";

```

```
begin
```

```
    LAD    <= LADout when LADoe = '1' else (others => 'Z');
```

```
-- Local bus FSM
```

```
process(LCLK, nLBRES)
```

```
    variable rreg, wreg : std_logic_vector(31 downto 0);
```

```
begin
```

```
    if (nLBRES = '0') then
```

```
        REG_RW1 <= X"00000003"; -- reset inactive, pulsewidth = 1 clock cycle
```

```
        REG_RW2 <= X"FFFFFFFF";
```

```
        REG_RW3 <= X"FFFFFFFF";
```

```
            REG_RW4 <= X"FFFFFFFF";
```

```
            REG_RW5 <= X"FFFFFFFF";
```

```
            REG_RW6 <= X"FFFFFFFF";
```

```
            REG_RW7 <= X"FFFFFFFF";
```

```
            REG_RW8 <= X"FFFFFFFF";
```

```
            REG_RW9 <= X"FFFFFFFF";
```

```
            REG_RW10 <= X"FFFFFFFF";
```

```
            REG_RW11 <= X"FFFFFFFF";
```

```
            REG_RW12 <= X"FFFFFFFF";
```

```
            REG_RW13 <= X"FFFFFFFF";
```

```
            REG_RW14 <= X"FFFFFFFF";
```

```
            REG_RW15 <= X"FFFFFFFF";
```

```
            REG_RW16 <= X"FFFFFFFF";
```

```
            REG_RW17 <= X"FFFFFFFF";
```

```
            REG_RW18 <= X"FFFFFFFF";
```

```
            REG_RW19 <= X"FFFFFFFF";
```

```
            REG_RW20 <= X"FFFFFFFF";
```

```
            REG_RW21 <= X"FFFFFFFF";
```

```
            REG_RW22 <= X"FFFFFFFF";
```

```
            REG_RW23 <= X"FFFFFFFF";
```

```
            REG_RW24 <= X"FFFFFFFF";
```

```
            REG_RW25 <= X"FFFFFFFF";
```

```
            REG_RW26 <= X"FFFFFFFF";
```

```
            REG_RW27 <= X"FFFFFFFF";
```

```
            REG_RW28 <= X"FFFFFFFF";
```

```
            REG_RW29 <= X"FFFFFFFF";
```

```
            REG_RW30 <= X"FFFFFFFF";
```

```
            REG_RW31 <= X"FFFFFFFF";
```

```

REG_RW32 <= X"FFFFFFFF";
REG_RW33 <= X"FFFFFFFF";
REG_RW34 <= X"FFFFFFFF";
REG_RW35 <= X"FFFFFFFF";
REG_RW36 <= X"FFFFFFFF";
nREADY <= '1';
LADoe <= '0';
ADDR <= (others => '0');
DTL <= (others => '0');
LADout <= (others => '0');
rreg := (others => '0');
wreg := (others => '0');
LBSTATE <= LBIDLE;
elsif rising_edge(LCLK) then

case LBSTATE is

when LBIDLE =>
LADoe <= '0';
nREADY <= '1';

if (nADS = '0') then -- start cycle
ADDR <= LAD; -- Address Sampling
if (WnR = '1') then -- Write Access to the registers
nREADY <= '0';
LBSTATE <= LBWRITEL;
else -- Read Access to the registers
nREADY <= '1';
LBSTATE <= LBREADL;
end if;
end if;

when LBWRITEL =>
DTL <= LAD; -- Save the lower 16 bits of the data
if (nBLAST = '0') then
LBSTATE <= LBIDLE;
nREADY <= '1';
else
LBSTATE <= LBWRITEH;
end if;

when LBWRITEH =>
wreg := LAD & DTL; -- Get the higher 16 bits and create the 32 bit data
case ADDR is
when A_REG_RW1 => REG_RW1 <= wreg;
when A_REG_RW2 => REG_RW2 <= wreg;
when A_REG_RW3 => REG_RW3 <= wreg;
when A_REG_RW4 => REG_RW4 <= wreg;
when A_REG_RW5 => REG_RW5 <= wreg;
when A_REG_RW6 => REG_RW6 <= wreg;
when A_REG_RW7 => REG_RW7 <= wreg;
when A_REG_RW8 => REG_RW8 <= wreg;
when A_REG_RW9 => REG_RW9 <= wreg;
when A_REG_RW10 => REG_RW10 <= wreg;
when A_REG_RW11 => REG_RW11 <= wreg;
when A_REG_RW12 => REG_RW12 <= wreg;
when A_REG_RW13 => REG_RW13 <= wreg;
when A_REG_RW14 => REG_RW14 <= wreg;
when A_REG_RW15 => REG_RW15 <= wreg;
when A_REG_RW16 => REG_RW16 <= wreg;
when A_REG_RW17 => REG_RW17 <= wreg;
when A_REG_RW18 => REG_RW18 <= wreg;
when A_REG_RW19 => REG_RW19 <= wreg;
when A_REG_RW20 => REG_RW20 <= wreg;
when A_REG_RW21 => REG_RW21 <= wreg;
when A_REG_RW22 => REG_RW22 <= wreg;
when A_REG_RW23 => REG_RW23 <= wreg;
when A_REG_RW24 => REG_RW24 <= wreg;
when A_REG_RW25 => REG_RW25 <= wreg;
when A_REG_RW26 => REG_RW26 <= wreg;
when A_REG_RW27 => REG_RW27 <= wreg;

```

```

        when A_REG_RW28    => REG_RW28    <= wreg;
        when A_REG_RW29    => REG_RW29    <= wreg;
        when A_REG_RW30    => REG_RW30    <= wreg;
        when A_REG_RW31    => REG_RW31    <= wreg;
        when A_REG_RW32    => REG_RW32    <= wreg;
        when A_REG_RW33    => REG_RW33    <= wreg;
        when A_REG_RW34    => REG_RW34    <= wreg;
        when A_REG_RW35    => REG_RW35    <= wreg;
        when A_REG_RW36    => REG_RW36    <= wreg;

    when others            => null;
end case;
nREADY <= '1';
LBSTATE <= LBIDLE;

when LBREADL =>
nREADY <= '0'; -- Assuming that the register is ready for reading
case ADDR is
    when A_REG_R1        => rreg := REG_R1;
    when A_REG_R2        => rreg := REG_R2;
    when A_REG_R3        => rreg := REG_R3;
    when A_REG_R4        => rreg := REG_R4;
    when A_REG_R5        => rreg := REG_R5;
    when A_REG_R6        => rreg := REG_R6;
    when A_REG_R7        => rreg := REG_R7;
    when A_REG_R8        => rreg := REG_R8;
    when A_REG_R9        => rreg := REG_R9;
    when A_REG_R10       => rreg := REG_R10;
    when A_REG_R11       => rreg := REG_R11;
    when A_REG_R12       => rreg := REG_R12;
    when A_REG_R13       => rreg := REG_R13;
    when A_REG_R14       => rreg := REG_R14;
    when A_REG_R15       => rreg := REG_R15;
    when A_REG_R16       => rreg := REG_R16;
    when A_REG_R17       => rreg := REG_R17;
    when A_REG_R18       => rreg := REG_R18;
    when A_REG_R19       => rreg := REG_R19;
    when A_REG_R20       => rreg := REG_R20;
    when A_REG_R21       => rreg := REG_R21;
    when A_REG_R22       => rreg := REG_R22;
    when A_REG_R23       => rreg := REG_R23;
    when A_REG_R24       => rreg := REG_R24;
    when A_REG_R25       => rreg := REG_R25;
    when A_REG_R26       => rreg := REG_R26;
    when A_REG_R27       => rreg := REG_R27;
    when A_REG_R28       => rreg := REG_R28;
    when A_REG_R29       => rreg := REG_R29;
    when A_REG_R30       => rreg := REG_R30;
    when A_REG_R31       => rreg := REG_R31;
    when A_REG_R32       => rreg := REG_R32;
    when A_REG_R33       => rreg := REG_R33;
    when A_REG_R34       => rreg := REG_R34;
    when A_REG_R35       => rreg := REG_R35;
    when A_REG_R36       => rreg := REG_R36;
    when A_REG_R37       => rreg := REG_R37;
    when A_REG_R38       => rreg := REG_R38;
    when A_REG_R39       => rreg := REG_R39;
    when A_REG_R40       => rreg := REG_R40;
    when A_REG_R41       => rreg := REG_R41;
    when A_REG_R42       => rreg := REG_R42;
    when A_REG_R43       => rreg := REG_R43;
    when A_REG_R44       => rreg := REG_R44;
    when A_REG_R45       => rreg := REG_R45;
    when A_REG_R46       => rreg := REG_R46;
    when A_REG_R47       => rreg := REG_R47;
    when A_REG_R48       => rreg := REG_R48;

    when A_REG_RW1       => rreg := REG_RW1;
    when A_REG_RW2       => rreg := REG_RW2;
    when A_REG_RW3       => rreg := REG_RW3;
    when A_REG_RW4       => rreg := REG_RW4;

```

```

when A_REG_RW5    => rreg := REG_RW5;
when A_REG_RW6    => rreg := REG_RW6;
when A_REG_RW7    => rreg := REG_RW7;
when A_REG_RW8    => rreg := REG_RW8;
when A_REG_RW9    => rreg := REG_RW9;
when A_REG_RW10   => rreg := REG_RW10;
when A_REG_RW11   => rreg := REG_RW11;
when A_REG_RW12   => rreg := REG_RW12;
when A_REG_RW13   => rreg := REG_RW13;
when A_REG_RW14   => rreg := REG_RW14;
when A_REG_RW15   => rreg := REG_RW15;
when A_REG_RW16   => rreg := REG_RW16;
when A_REG_RW17   => rreg := REG_RW17;
when A_REG_RW18   => rreg := REG_RW18;
when A_REG_RW19   => rreg := REG_RW19;
when A_REG_RW20   => rreg := REG_RW20;
when A_REG_RW21   => rreg := REG_RW21;
when A_REG_RW22   => rreg := REG_RW22;
when A_REG_RW23   => rreg := REG_RW23;
when A_REG_RW24   => rreg := REG_RW24;
when A_REG_RW25   => rreg := REG_RW25;
when A_REG_RW26   => rreg := REG_RW26;
when A_REG_RW27   => rreg := REG_RW27;
when A_REG_RW28   => rreg := REG_RW28;
when A_REG_RW29   => rreg := REG_RW29;
when A_REG_RW30   => rreg := REG_RW30;
when A_REG_RW31   => rreg := REG_RW31;
when A_REG_RW32   => rreg := REG_RW32;
when A_REG_RW33   => rreg := REG_RW33;
when A_REG_RW34   => rreg := REG_RW34;
when A_REG_RW35   => rreg := REG_RW35;
                    when A_REG_RW36 => rreg := REG_RW36;

    when others      => null;
end case;
LBSTATE <= LBREADH;
LADout <= rreg(15 downto 0); -- Save the lower 16 bits of the data
LADoe <= '1'; -- Enable the output on the Local Bus

when LBREADH =>
    LADout <= rreg(31 downto 16); -- Put the higher 16 bits
    LBSTATE <= LBIDLE;

end case;

end if;
end process;

END rtl;

```