THE FLORIDA STATE UNIVERSITY

FAMU-FSU COLLEGE OF ENGINEERING

SYSTEM-ON-PROGRAMMABLE CHIP (SOPC)

IMPLEMENTATION OF THE SILICON TRACK CARD

By

ARVINDH-KUMAR LALAM

A Thesis submitted to the to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2002

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. Reginald J. Perry for his guidance and support throughout my graduate study at FSU. I would like to thank the members of my thesis committee, Dr. Simon Y. Foo and Dr. Uwe Meyer-Baese, for their valuable advice and guidance. I would also like to thank Dr. Horst D. Wahl from the Physics Department for his support throughout my work as a Research Assistant. I wish to thank the academic and administrative staff at the Department of Electrical and Computer Engineering for their kind support. I wish to thank the researchers from the Physics Department, Florida State University and the Physics Department, Boston University for their guidance. I wish to thank my family for their continuous support and confidence in me. I also wish to thank my friends for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Silicon Track Card (STC) is a digital circuit used as a part of the Silicon Track Trigger (STT) for the DZERO (D0) experiment at the Fermi National Accelerator Laboratory (FermiLab) in Batavia, Illinois. The preliminary implementation (Version 1.0) of the STC uses Altera's Flexible Logic Element MatriX (FLEX) programmable devices. In this implementation, each STC requires three to five FLEX devices. Usage of multiple programmable devices consumes more board space and increases the complexity of the board-design. In addition, splitting the STC to fit into multiple devices results in unpredictable programmable delays between various modules of the STC.

The current thesis work focuses on upgrading the STC and implementing it as a System-on-Programmable-Chip (SOPC). As part of the SOPC implementation, the STC is modified to fit into a single Altera's Advanced Programmable Embedded MatriX (APEX) device. The performance of this implementation has been validated at an experimental setup in Boston University. In order to upgrade the STC, a new buffer module (L3 module) is incorporated to handle debugging information. Out of the total time taken by the STC to process an event, typically 40% of the time is consumed only by the hit-filter, one of the STC components. Two new schemes have been developed to improve the performance of the hit-filter module, and thus the STC. These schemes use

APEX Content Addressable Memory (CAM) and are discussed in detail along with the previous hit-filter scheme.

CHAPTER 1

INTRODUCTION

Programmable devices are Integrated Circuits (ICs), which can be programmed "in-house" to implement digital logic designs. Though programmable devices are not mask programmable, they can be reconfigured to implement a particular circuit and thus are considered to be a part of the Application Specific Integrated Circuits (ASIC) family [1]. The building blocks of these devices are universal function generators, which can generate all logic functions for a given set of inputs. A simple example of a universal function generator is a 2-input NAND gate which can be used to implement any 2-input logic function. The design and implementation of the digital circuits in programmable devices requires an understanding of the software programming tools. The circuits can be designed using schematic capture or by using Hardware Description Languages (HDLs) like the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) [2] or Verilog. The design files written in VHDL or Verilog can be synthesized by either third party Electronic Design Automation (EDA) tools or by the software provided by the programmable device vendor. The vendor software then uses the synthesized file to generate a "configuration file" that can be used to configure the programmable device.

The developments in VLSI technology have enabled the chipmakers to place many of the important modules like on-board memory, processor core and Phase Locked Loop (PLL), on a single Integrated Circuit (IC). A “mask programmable” device that contains these essential modules is called a System-on-Chip (SOC). The SOCs have the required resources for building a digital system on the same IC and thus provide full functionality for an application with minimum number of components. These SOC devices typically have millions of gates, which were not available in programmable devices. But with huge strides in lithography techniques and fabrication processes, 0.11-mircon and 0.13-micron processes are now realizable. The corresponding increase in gate count has resulted in a new breed of programmable devices that are suited for System-On-Programmable Chip (SOPC) solutions. These programmable devices can accommodate most of the system functionality on a single IC like an SOC. Altera’s Advanced Programmable Embedded MatriX (APEX) device is an example of Programmable Logic Devices (PLDs) that offer SOPC integration [3].

Fast electronics called a ‘trigger’, associated with the D0 detector at Fermi National Accelerator Laboratory (FermiLab), performs the task of digitally sieving events for particular occurrences that are of interest to physicists. This system is divided into various levels each of which performs event selection to some extent. Effectively, data rate at the input of first level is 7MHZ, while data output rate at the last level of the trigger is 50Hz. The Silicon Track Card (STC) [2] is part of the Level2 trigger. The primary function of this module is to identify the charges collected in the detector that fall in particle paths.

The current project is based on the Version 1.0 of the STC discussed in [2]. This implementation of the STC required multiple ICs of the Flexible Logic Element MatriX (FLEX) family of PLDs [2]. As part of the current thesis work, STC has been implemented as an SOPC in a single APEX high-density device. The functionality of the SOPC implementation has been validated in hardware by using a custom-built STC prototype board at the experimental setup in Boston University (BU). The current work also includes incorporating a buffer module (L3 module) to store the intermediate information for debugging purposes. In addition, various schemes have been devised to use the Content Addressable Memory (CAM) functionality of the Altera's APEX devices to optimize the STC. The "hit-filter" module [2] and the "hit-format" module [2] have been designed to use the on-chip CAM resources. The "hit-filter" module using CAM was found to be utilizing more resources than the current implementation. However, the "hit-format" module using CAM blocks has improved the performance of the STC by a considerable factor.

In this thesis, Chapter 2 describes the programmable devices in more detail and discusses various architectures and their attributes. Chapter 3 introduces the field of High Energy Physics (HEP) and shows the functioning of the D0 Trigger. Chapter 4 describes the STC and its various modules. This chapter also describes the implementations of STC with FLEX and APEX devices. Chapter 5 explains the implementation of various "hit-filter" modules using the CAM blocks. Chapter 6 contains the conclusions and future work.

CHAPTER 2

PROGRAMMABLE DEVICE ARCHITECUTURES

The programmable devices have gradually grown in prominence in the IC market. The first programmable devices implemented Sum of Products (SOP) representation of the logic functions with a limited number of inputs. These devices have ever since grown in magnitude and technology to include the SOC functionality in a programmable device, the SOPC. Though they are associated with higher cost, programmable devices have gained popularity due to in-house programmability.

The following section details the evolution of the programmable device architectures. The products of leading vendors, Altera Corporation and Xilinx Incorporation, are compared in the following discussion.

2.1 Programmable Logic Array

A PLA is a combinational AND-OR programmable circuit arranged in two levels [4]. The PLA can be programmed to implement any logic function with a given number of inputs. However, the minterms required to represent the logic function in a Sum of Products (SOP) expression should not exceed the number of AND gates present in the device.

2.2 Programmable Array Logic (PAL) device

A PAL device is an extension of PLA introduced by Monolithic Memories, now part of Advanced Micro Devices (AMD) [4]. As opposed to PLAs, where arrays of both the AND and OR gates are programmable, in PAL devices, only the AND gate arrays are programmable. Each of the OR gates is permanently connected to a group of AND gates. Thus, the maximum number of minterms allowed for an OR gate is equal to the number of inputs to the OR gate. The logic functions with more minterms can be implemented by routing the output of one OR gate to input of another minterm set as shown in

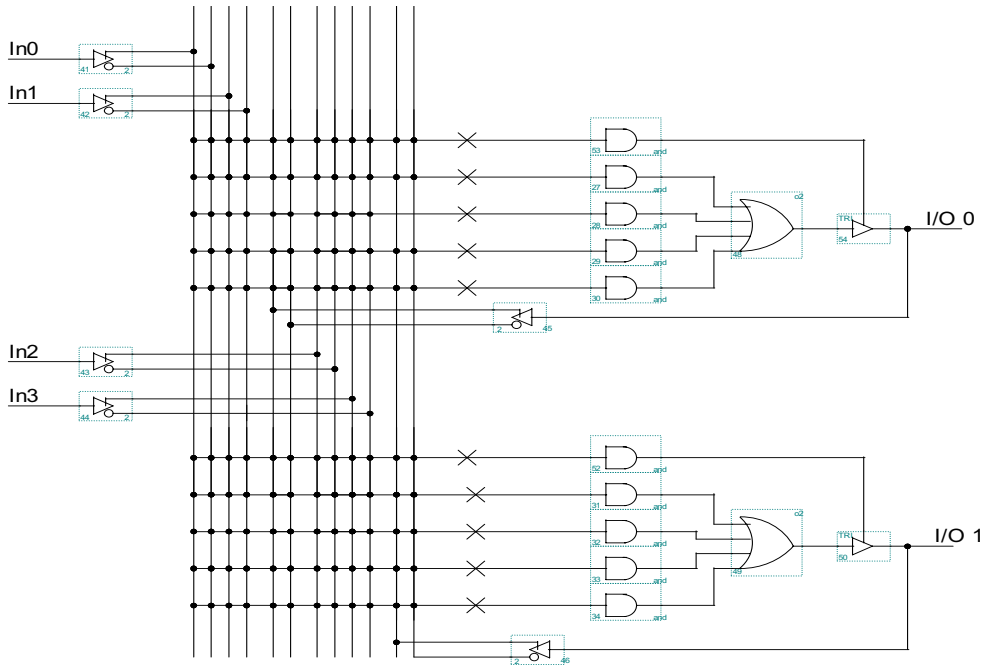


Figure 2.1 Programmable Array Logic (PAL) Device

2.3 Complex Programmable Logic Device (CPLD)

CPLDs are more complex than the programmable devices considered in previous sections. The CPLDs consist of groups of arrays of logic elements or logic cells which are connected through an interconnect, as shown in Figure 2.2 [5]. In these devices the datapath is not unidirectional from input to output of the IC. Instead, outputs of all the arrays are fed back to the common interconnect lines as shown in Figure 2.2 [5]. Output of a logic cell that is required to be fed as an input to another logic cell is first routed back to the common interconnect lines and then connected to the destination logic. While most of the first generation devices released by Altera Corp. belonged to the category of CPLDs, few first generation devices released by Xilinx Inc. were based on CPLD architecture.

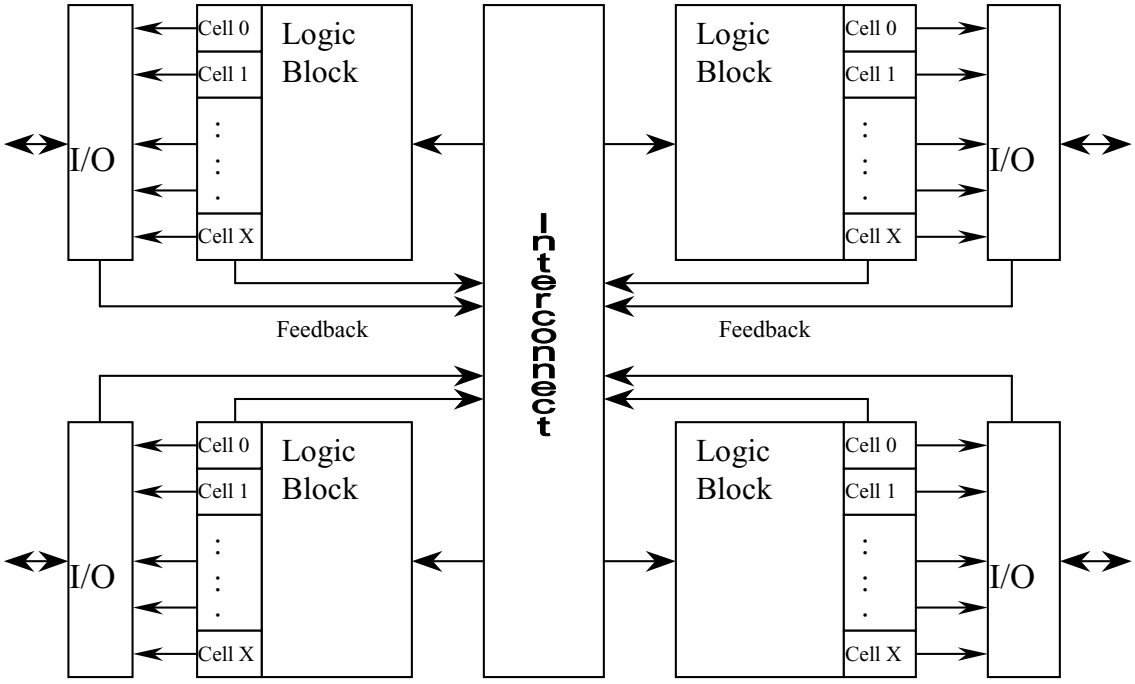


Figure 2.2 Complex Programmable Logic Device Structure (CPLD)

Altera Corporation released the Multiple Array Matrix (MAX) devices as part of the CPLD family. These devices comprised of MAX 5000, MAX 3000A, MAX 7000 and MAX 9000. While MAX 5000 uses Erasable PROM (EPROM) technology, other devices use Electrically Erasable PROMs (EEPROM) technology [6]. Xilinx Incorporation released XPLA2, 'Cool Runner XPLA3' and XC9500 as part of the CPLD family. All the above devices released by Xinlix Inc. utilized Flash memory technology [7]. Both the EEPROM and the Flash memory are electrically erasable. They however differ in the way data is erased from the memory. In an EEPROM, one bit is erased at a time, while in Flash memory a block of memory bits or the entire chip is erased at a time.

2.4 Mid-Density Families

Traditionally, "gate arrays" contain a number of building blocks or primitive cells [1] etched on the silicon throughout the chip area. The permanent connections between various terminals of the primitive cells are made later. These write-once devices can hold high-density circuits of the order of 5 million gates. FPGAs are similar to "gate arrays" in structure, as shown in Figure 2.3 [8]. However, FPGAs contain groups of programmable logic elements or basic cells instead of primitive cells found in "gate arrays". The programmable cells used in Altera's devices are called Logic Elements (LEs) [9] while the programmable cells used in Xilinx's devices are called the Configurable Logic Blocks (CLBs) [10]. The FPGAs are based on the Complementary Metal Oxide Semiconductor (CMOS) SRAM technology and thus are reset on power off.

The competing families of the second-generation mid-density programmable logic devices are the FLEX devices of Altera Corporation and XC3000, XC 4000 and XC5200

devices of Xilinx Incorporation. This generation of devices has a drastic improvement over the previous CPLD families in terms of gate count.

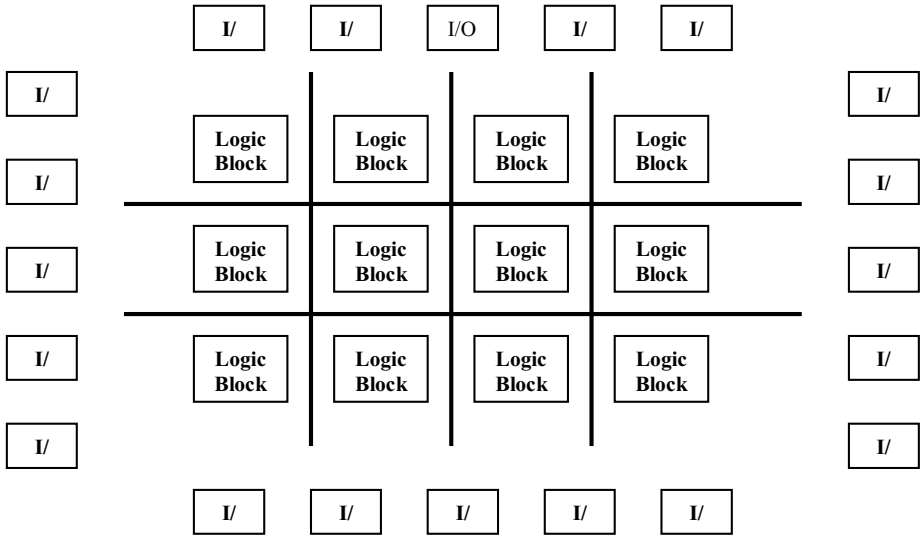


Figure 2.3 Field Programmable Gate Array (FPGA)

An important breakthrough achieved with these devices is the on-chip memory. Since, almost all the digital circuits need memory, external memories were extensively used. This limits the operating speed of the devices due to the delays associated with external interconnects across the PCB. Thus, the usage of on-chip memory drastically improves operating speed of the ICs. Another aspect of this generation of devices is the inclusion of embedded Phase Locked Loops (PLL) or Delay Locked Loops (DLL). In order to avoid timing hazards in the device, all the clocks have to be synchronized by a Phase Locked Loop externally. However, implementation of the PLL on the chip itself saves board space and improves the operating speed of the circuit.

2.5 The High density Families

The high-density programmable devices are the next generation devices with a capacity as high as 8 million gates. These PLDs are low-power devices that contain on-chip memory, additional clock management circuitry like PLL blocks and built-in low-skew clock trees [11]. Some devices also contain specialized blocks to implement arithmetic functions like multipliers. These devices provide a comprehensive "System-on-Programmable-Chip" (SOPC) solution for digital applications.

Figure 2.4 [11] shows the MegaLAB structure of Altera's APEX chips. The MultiCore architecture of APEX 20K devices integrates product-term logic, the Lookup Up Table (LUT) logic and the embedded memory [3]. The Figure 2.5 [12] shows the arithmetic module integrated into the Xilinx Virtex II device. The properties of devices from these contemporary device families are compared in the Table 2.1. [11][12].

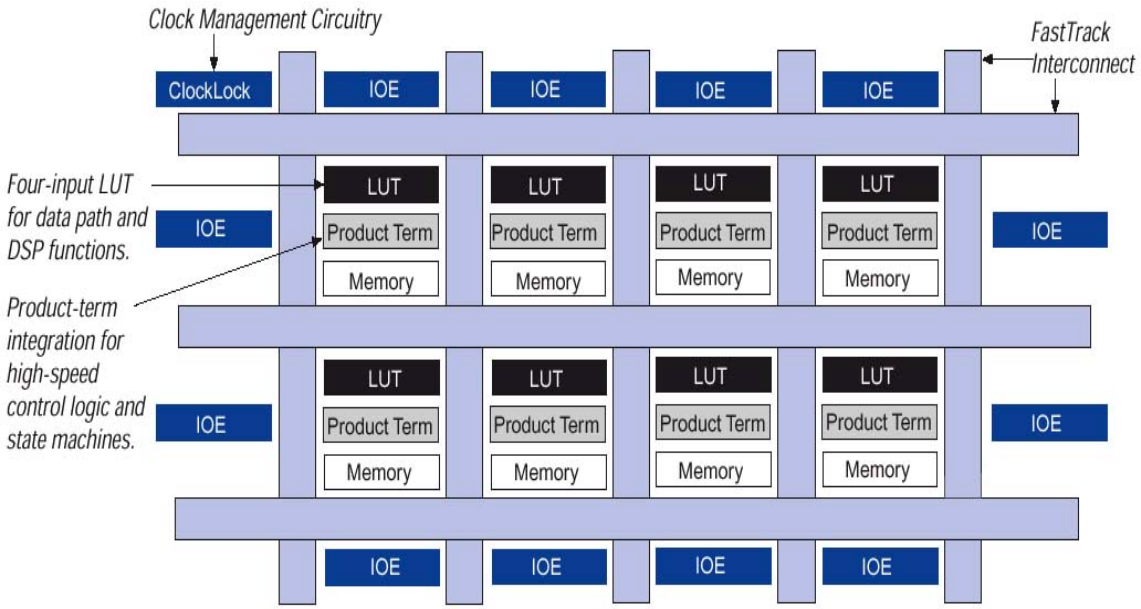


Figure 2.4 MegaLAB in Altera's APEX

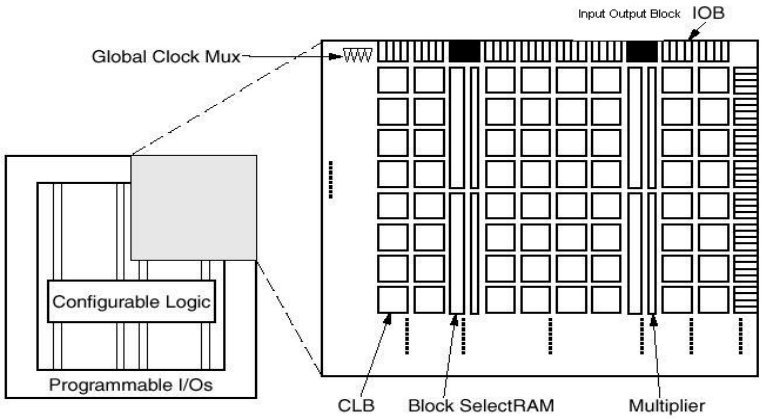


Figure 2.5 FPGA Architecture of Xilinx Virtex

Table 2.1 Comparison of High-density FPGA families

	Altera	**Xilinx**
Device families	APEX 10KE	Virtex II
Architecture	Uses both CPLD and gate array techniques	FPGA
Process technology	.22 micron	.15 / .12 micron (Virtex II)
Usable typical gates (max*)	5.25 Million	8 Million
Salient feature	CAM	Dedicated Multiplier blocks
Memory Bits	1.15 Mb	3 Mb of select RAM 1.5 Mb of CLB
Dual port RAM	Yes	Yes
Phase Locked Loop	Yes (clcoklock, clockboost and clockshift)	Yes
I/O pins	1060	1,108
Software support	Quartus II	3.2i Alliance Series and Foundation Series™ Integrated Synthesis Environment (ISE™)

2.6 Stratix

The latest device family released by Altera Corp. is the Stratix. The tri-matrix feature [13] of Stratix uses dedicated memory blocks of various sizes, unlike the previous device families, which had memory blocks of fixed size. The Stratix devices for the first time implement dedicated arithmetic blocks in Altera's devices. They contain several DSP blocks, each of which can be configured as eight 9 × 9-bit multipliers or four 18 × 18-bit multipliers or One 36 × 36-bit multiplier. Table 2.2 shows a comparison of APEX and Stratix devices [11] [13].

Table 2.2 Comparison of the APEX and Stratix devices of Altera Corp

	APEX	**Stratix**
Process technology	.15 micron	.13 micron
Usable typical gates (max*)	5.25 Million	1.1 Million
Architecture	MultiCore architecture	Trimatrix memory
Salient feature	ESB that can be used as CAM	Dedicated Multiplier blocks
Memory capacity	1.15 Mb	10 Mb
Memory block size	fixed	variable
Number of PLLs	4	12
I/O pins	1060	1310
Software support	Quartus II version 1.0	Quartus II version 2.0

In order to bridge the gap between the programmable devices and ASICs, Altera Corp. also introduced “hardcopy” devices. The “hardcopy” devices offer an economical alternative to the migration of the circuits from an SOPC prototype to high-volume ASICs [14].

The hardcopy devices for APEX contain the same basic functional blocks except for the programmable interconnects. The configurable routing resources in APEX devices are replaced by custom interconnects that use small die area in comparison with the actual APEX devices [14].

The APEX 20KE was chosen for the current implementation of the project. Altera’s SOPC development board, containing a EP20K400EBC652-1X is used for debugging individual modules. A custom-designed board containing two EP20K600EBC652-1X ICs was used for validating the performance of two STCs functioning simultaneously. The device specifications for EP20K400EBC652-1X and EP20K600EBC652-1X are given in the Table 2.3. [3]

Table 2.3 Device specifications of APEX20KE devices used to implement STC.

	Typical gates	**Logic Elements**	**Maximum RAM bits**	**Maximum ESBs**	**Maximum User Pins**
EP20K400EBC652-1X	400,000	16,640	212,992	104	488
EP20K600EBC652-1X	600,000	24,320	311,296	152	588

CHAPTER 3

HIGH ENERGY PHYSICS AND THE D0 EXPERIMENT

By the middle of 1930s, protons, neutrons and electrons were considered to form the core of matter and thus were considered to be the fundamental particles constituting matter. The atom was envisioned as a heavy nucleus that is comprised of heavy protons and neutrons with a number of electrons revolving around the nucleus in large orbits. The heavy nucleus was found to be bearing a net positive charge and occupying a relatively minute volume in the atom while being predominantly responsible for the atom's mass. Electrons however were found to have minute mass but equal and opposite charge to that of protons. This theory could explain most of the properties exhibited by matter. However, questions concerning the particles themselves, like 'Why protons and neutrons stay together?' baffled researchers. Many such exceptions were soon found and search for a model that identifies actual fundamental particles and better explains the inconsistencies was underway [15].

Accelerators have increasingly found use in next generation of experiments studying fundamental particles and their interactions. These devices accelerate particles producing particle beams of very high energy. Two such beams traveling in opposite directions are allowed to meet in a collision chamber of an accelerator, resulting in

collisions. After a collision between two high energy particles, tracks of generated particles and their decay is studied to identify a particular sequence of events called 'signature', to identify the particles. The field of High Energy Physics (HEP) deals with particle experiments studying these collisions [15]. Layers of detectors, each of which measure a particular parameter, surround the collision chamber. Information from all the detectors is analyzed to identify patterns associated with the particles and hopefully new particles. These accelerators at various locations around the world led to the discovery of around two hundred particles till date, though a very small fraction of these are considered to be fundamental particles. These discoveries helped develop "The Standard Model of Fundamental Particles and Interaction".

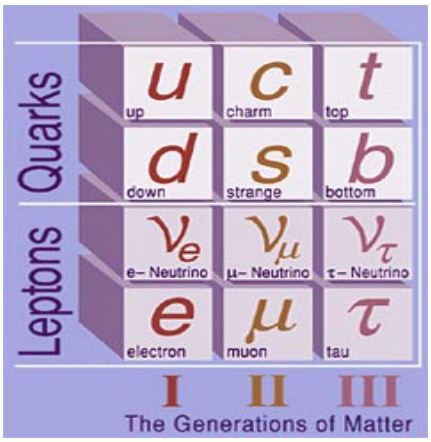


Figure 3.1 Generations of matter in The Standard Model.

3.1 The Standard Model

The Standard Model identifies ‘quarks’ and ‘leptons’ as the fundamental particles and explains particle interactions in terms of ‘gravitational’, ‘electromagnetic’, ‘weak’ and ‘strong’ forces [15]. ‘Quarks’ and ‘leptons’ are of six types each and in turn have an equal number of anti-particles. The Standard Model categorizes these particles into three sets, each consisting of two quarks and two leptons as shown in Figure 3.1 [15]. Each of these sets is called a generation of matter. Generations of matter are arranged in increasing order of mass. The heaviest particles fall under third generation of particles and are the most unstable, thus very hard to detect. For example, top quark, considered to be the third generation particle is exceptionally heavy with its mass equal to that of a gold atom and with an occurrence of once in several billion collisions [15]. The Standard Model describes protons, neutrons and electrons, previously considered fundamental, in terms of ‘quarks’ and ‘leptons’. Protons and Neutrons are made up of three first generation quarks while electrons are first generation charged leptons. For example a proton is made of two ‘UP’ quarks and one ‘DOWN’ quark as shown in Figure 3.2. The fact that protons have high mass, in spite of low mass of its constituent quarks, is explained by the kinetic and potential energies of constituent particles [15].

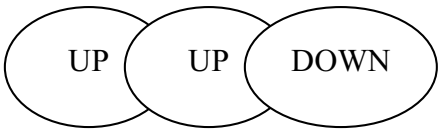


Figure 3.2 Constituents of a proton.

3.2 Fermilab

The Fermi National Accelerator Laboratory (FNAL), also called Fermilab, was commissioned in November 21, 1967, under the name of National Accelerator Laboratory by the United States Atomic Commission [16]. It was renamed to the present name on May 11, 1974, in honor of Nobel laureate Enrico Fermi. Fermilab has since been in the forefront of research in High Energy Physics helping researchers understand fundamental nature of matter and energy. It is credited with the discovery of the two third generation quarks, 'bottom' and 'top' quarks. The 'bottom' quark was discovered in 1977 suggesting existence of the 'top' quark, the last of the six quarks. The 'top' quark was finally discovered in 1995 at the TeVatron[16] accelerator situated in Fermilab.

3.3 D0 trigger

TeVatron accelerator has two detectors, DZero (D0) and Collider Detector at Fermilab (CDF). The D0 detector is a general-purpose collider detector that uses beams of proton and anti-protons. This is being upgraded to study more about the 'top' quark and look for previously undetected phenomena. Though particle beams with high luminosity of $2x10^{32}$ particles per square centimeter per second ($2x10^{32}cm^{2}s^{-1}$) are used in TeVatron [17], a very small fraction of the proton anti-proton pairs actually collide and a still smaller fraction of these collisions result in events that are of interest to physicists. The number of rare events that are of interest, like generation of the top quark, are in the order of one in 10 billion collisions. The objective of the detector is to identify these rare events among billions of events occurring every second during the course of collisions

between protons and anti-protons. This depends on how well the trigger eliminates unwanted events. In Run I of D0 collider that was carried between 1992-1996 [17], events were recorded at a rate of 3.5 Hz from a total collision rate of 0.5 to 1.0 MHz. For Run II D0 is being upgraded to operate with a ten-fold improvement in beam intensity (luminosity) [17] and twenty-fold improvement in the amount of data [18]. The decision electronics used in the detector, also called a 'trigger', is divided into three levels. The Figure 3.3 shows Level 1 and Level 2 of the upgraded D0 trigger.

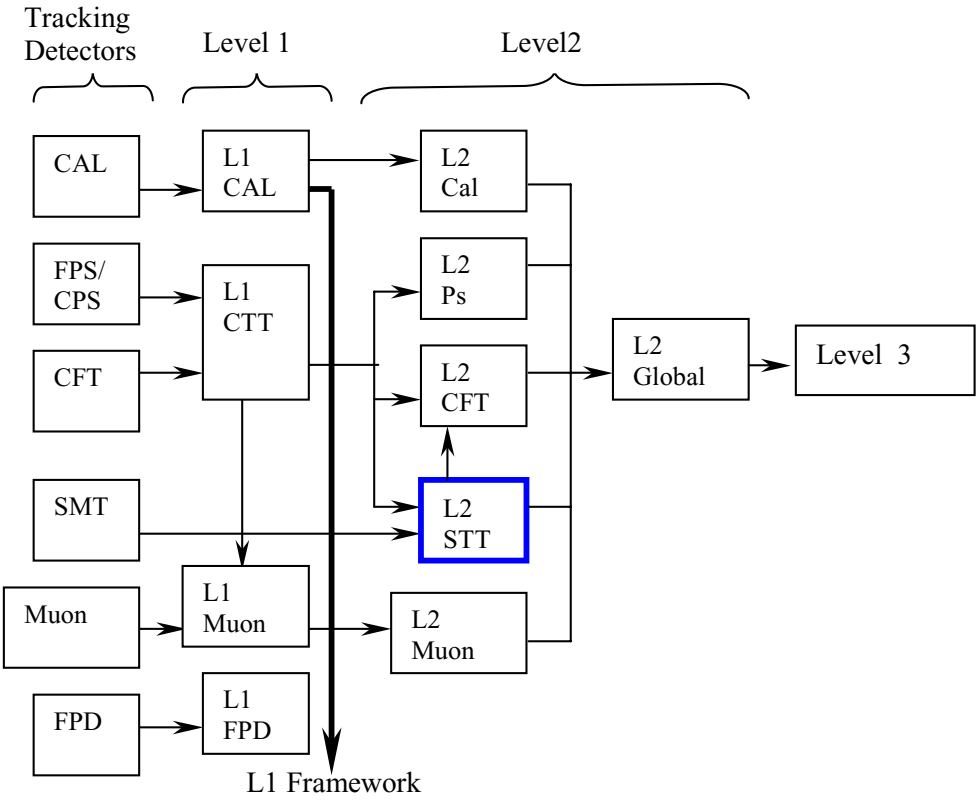


Figure 3.3 Level 1 and Level 2 of D0 Trigger

The tracking detectors of the upgraded D0 detector are Central Fiber Tracker (CFT), silicon tracker, calorimeter, muon scintillators, central and forward preshower detectors (CPS and FPS) [17]. In addition, D0 detector also contains Silicon Micro-strip Tracker (SMT), which directly sends the captured data to the Level 2 [19]. The SMT consists of layers of rectangular silicon wafers acting as p-n junctions, which are in depletion mode over the whole length of the wafer. The passage of the charged particles through the wafers results in generation of an electron-hole pair. This charge is collected by aluminum electrodes called "strips" and deposited on chips that contain 32 deep capacitor arrays. Analog-to-Digital Converters (ADCs) [2] are used to digitize the deposited charge. The digitized data from the ADC is then sent to the Level 2 through an optical link. However, Level 2 will not process this event data until the Level 1 issues a corresponding trigger. The various levels of the trigger are briefly described.

3.3.1 Level 1

Level 1 analyzes detector data, locates clusters of energy in calorimeter (CAL) and identifies hit patterns in Central Fiber Tracker (CFT) and Muon chambers that follow a pre-programmed format [18]. Framework in Level 1 has 128 trigger bits, each of which is set when specific combinations of trigger terms are found [17]. Various combinations of trigger terms are used to set the bits, setting any of which sends a trigger and the corresponding event data to Level 2. An example of triggering combination is a track candidate in CFT having energy more than a particular threshold [2]. Output rate from Level 1 to next stage is 10 KHz.

3.3.2 Level 2

Level 2 improves accept rate of events by a factor of ten. This has access to more refined information than Level 1 and processes data in two stages. First stage consists of preprocessors that analyze data sent by corresponding modules in Level 1. All preprocessors send data to Level 2 global processor (second stage), which makes a decision of selecting or rejecting events. Data from various modules is combined for the first time in this processor. The Level 2 Silicon Track Trigger (L2STT) is one of the preprocessors and is organized into fiber road card (FRC), STC and track fit card (TFC) [2] as shown in Figure 3.4.

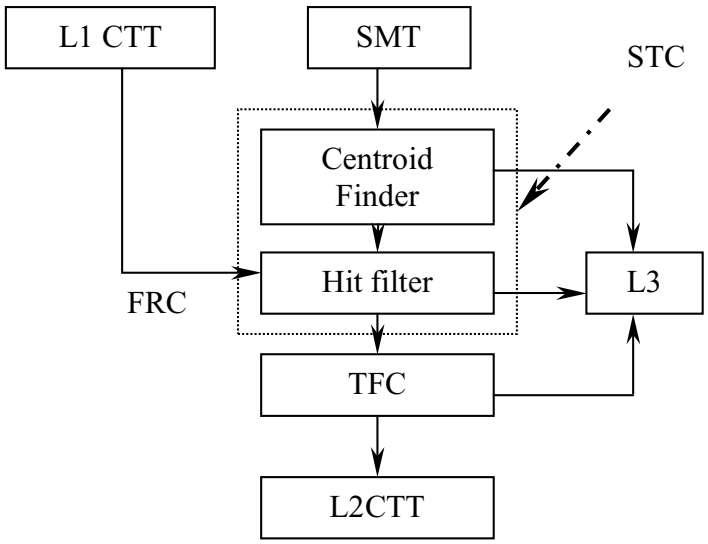


Figure 3.4 Block diagram of the Level 2

FRC receives information from the Level 1 CFT and generates particle trajectory information understandable to the STC (roads), as shown in Fig 3.5 [20]. The data from the optical fiber layers A-H of the CFT are used to define a "road", which passes through the SMT layers as shown in Figure 3.5. The detector layers are divided into various segments, each of which is connected to a group of STCs. Each STC receives SMT data (charge information) directly from one segment of the detector [18] and finds the clusters of charges. It then calculates cluster centroids and compares them with roads received from FRC. The centroids that fall within a road are called 'hits' and are shown in Figure 3.5. STC sends this hit information to the TFC and Level 3 [2]. TFC uses track-fitting algorithms to find the path taken by a newly generated particle.

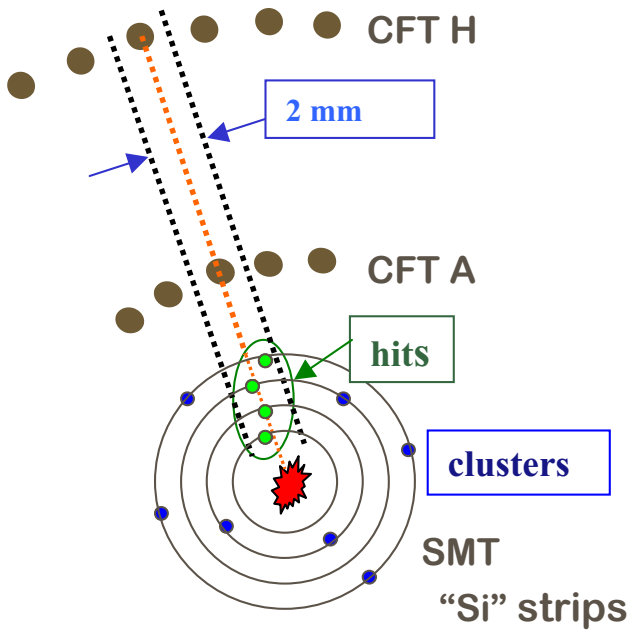


Figure 3.5 Functional diagram of the D0 trigger and Level 2

3.3.3 Level 3

Level 3 is final level of the D0 trigger. Upon receipt Level_2 accept, Level 3 receives data from Level 1 and Level 2 modules for final selection of events. This stage is implemented in software unlike other levels and uses parallel fast processors to achieve the processing rate required [17]. Output rate of this final stage is 50 Hz. Events are written onto disk after Level 3 for further examination.

CHAPTER 4

SILICON TRACK CARD

The charges found in the SMT layers are sent to the STC in digitized form, called "strip" information. The information sent by Level 1 CFT is used by the FRC to define "roads", each of which represents a path 2-mm wide. The function of each STC is to organize the strip information into groups called "clusters" and to find the centers of these clusters. In addition, STC identifies "hits", the cluster centers that fall in the "roads" received from FRC. The identified "hits" are sent to the TFC for further processing. The "control logic" designed by engineers at BU acts as an interface between the STC channels and the rest of the STT. Instead of taking the live SMT data from the D0 detector, STC uses an internal test-FIFO during the test phase. The "control logic" downloads the test vectors into the test-FIFO before starting the processing of the event.

4.1 Main Datapath

The STC constitutes a main data path, miscellaneous memory block [2] and L3 buffers as shown in Figure 4.1. Since, several STCs function in parallel, the data stored in the miscellaneous memory block is used to distinguish various STCs. The Main Data

Path is indicated in Figure 4.1 as shaded regions. This has three major parts, the “strip reader”, “cluster finder” and “hit filter.” Each of these modules will be briefly described.

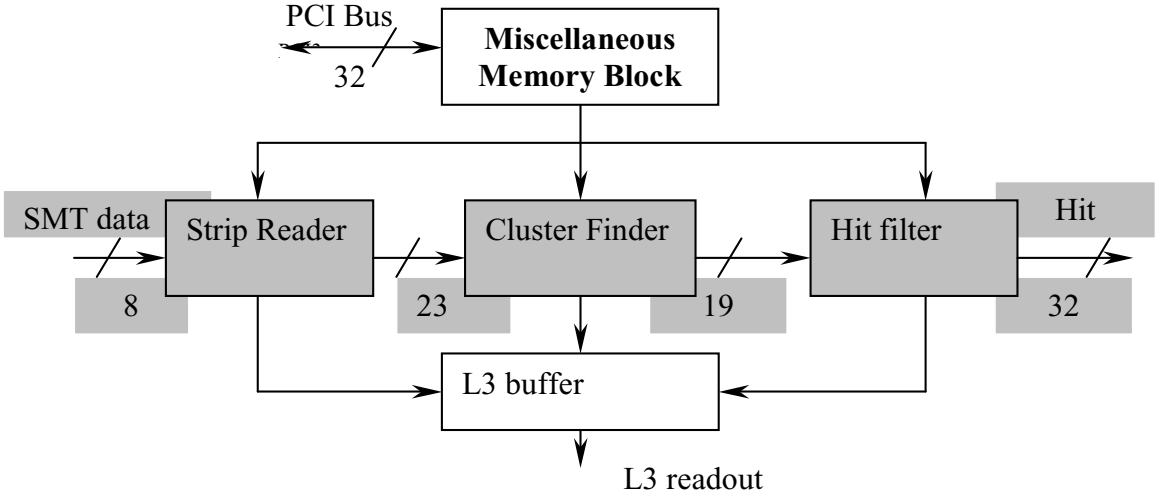


Figure 4.1 STC and Main data path.

4.1.1 Strip Reader Module

The strip reader module accepts the SMT strip information in the form of a byte stream arriving at a rate of 53MHz and formats it into an 18bit word [2]. Look Up Tables (LUTs) are used to identify bad strips and to perform gain and offset compensation for good strips. The valid data words thus obtained are stored in a FIFO for later use by the cluster finder module.

4.1.2 Cluster Finder Module

The cluster-finder module contains a clustering algorithm and a centroid calculator. The clustering module organizes the strips into "clusters", consisting of either three or five strips [2], while the centroid calculator finds the cluster's center. The clustering module organizes strips such that the strip with the highest value is placed in the center while the strips immediately before and after this are arranged on either side in the same order. The centroid calculator is an asynchronous module that takes the strip data from the clustering module. The centroid calculation in this module is centered on the second strip. This module generates the centroids by adding an offset value to the second strip in the cluster. The expressions used to find the offset for both the five-strip and three-strip clusters are shown, with D1, D2, D3, D4 and D5 representing strip data:

$$\text{Five strip centroid} = \frac{-\text{D1} + \text{D3} + 2\text{D4} + 3\text{D5}}{\text{D1} + \text{D2} + \text{D3} + \text{D4} + \text{D5}}$$

$$\text{Three strip centroid} = \frac{\text{D3} + 2\text{D4}}{\text{D2} + \text{D3} + \text{D4}}$$

The calculated centroid offset values are represented in three bits in the centroid-calculator. This allows the range of numbers between 0 and 2 to be categorized into 8 quarters, a 3-bit word representing all the values falling in a particular quarter, as shown in Table 4.1. The minimum and maximum offsets possible for the five-strip cluster are 0 (0.00) and 2 (1.11), while the values for three-strip cluster are 0.5 (0.10) and 1.5 (1.10). The maximum quantization error introduced in this process is 0.25. The calculated centroid effectively has a precision of two bits. The generated centroids, with format as shown in Table 4.2, are stored in the centroid FIFO for further readout.

Table 4.1 3-bit representation of the Centroid offset

Offset Range	0.00 to 0.24	0.25 to 0.49	0.50 to 0.74	0.75 to 0.99	1.00 to 1.24	1.25 to 1.49	1.50 to 1.74	1.75 to 2.00
Binary Equivalent	0.00	0.01	0.10	0.11	1.00	1.01	1.10	1.11

Table 4.2 Distribution of bits in the 13-bit Centroid word

12 .. 9	**8 .. 2**	**1..0**
Chip ID	Strip address	Precision Bits

4.1.3 Hit Filter

The hit-filter receives centroids from the centroid-FIFO and roads from the memory associated with FRC. Each of the roads received by the hit-filter has 22 bits, of which the first 11 bits are called “upper-address”, while the last 11 bits are called the “lower-address.” The upper-address and lower-address represent the strips on either sides of a road and thus define the road boundaries. The two precision bits of the centroid are discarded while checking for “hits”, thus the centroids used in the hit-filter have only 11 bits. The hit-filter functions in two phases. In the first phase, it internally stores all the received roads. In the second stage, for each of the centroids, hit-filter identifies the roads whose boundaries satisfy the following condition.

$$\text{lower - address} \leq \text{centroid} \leq \text{upper - address}$$

The track numbers of the identified roads are used to generate “hit-words.” For example, if a centroid falls in the fifth and seventh roads, the associated track numbers

will be “000101” and “000111”. Each centroid can fall in more than one road, thus each centroid may result in multiple hits. After hits of all the centroids are stored in the hit-FIFO, hit-filter also writes a “hit-trailer.”

In the Version 1.0 of STC, hit-filter contains a “comparator” module and a hit-format module as shown in Figure 4.2. The comparator module contains several “hit-match” modules in parallel. Each of these modules is designed to contain the upper-address and lower-address of a road. When a hit-match module receives a centroid, it checks to see if the centroid results in a hit. The output of this module is a ‘1’ in case of a hit and a ‘0’ otherwise. Since only one road can be stored in a hit-match block, 46 of these blocks are required in the “comparator” module to store the maximum number of 46 roads as determined in the design specifications [21]. Thus, the output from the comparator-module is a 46-bit word, each bit representing presence or absence of a centroid in that particular road.

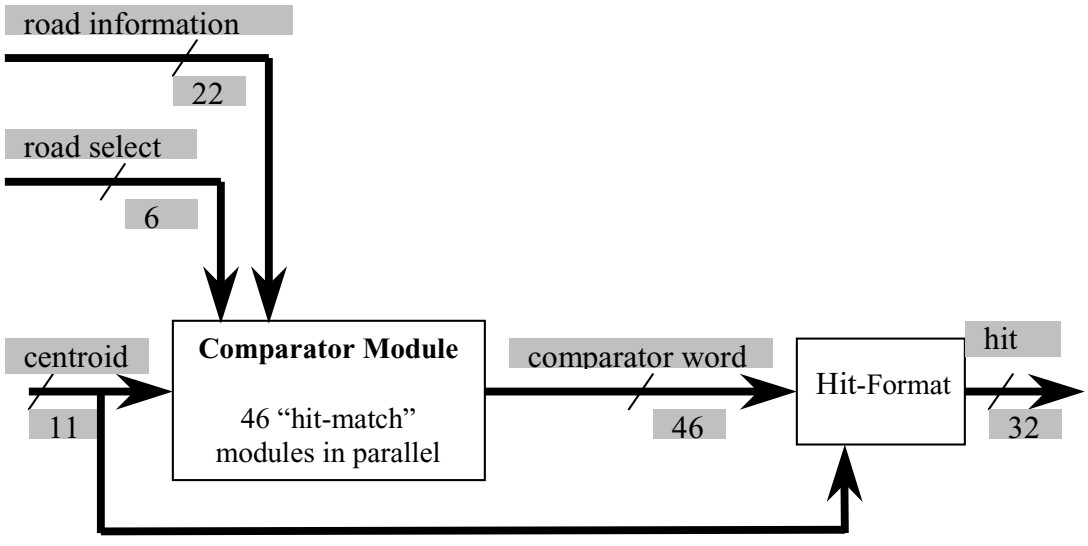


Figure 4.2 The Hit Filter Block in the previous STC

The hit-format module encodes the locations of '1's in the 46-bit comparator word to determine the track numbers. Hit-format module designed using VHDL employs a Finite State Machine (FSM) to perform sequential search of the comparator word for '1's. A counter is used to assign the track number to the detected '1's. The hit-filter uses handshaking signals to find if hit-format module is busy, before reading the next centroid. After hit-format block writes all the hit-words for the centroid, hit-filter reads the next centroid and this process continues until the centroid-FIFO is empty.

In the upgraded STC, the hit-format module is replaced by a hit-word generator. While the former uses sequential search, the latter use APEX CAM to encode the locations of '1's in comparator word. The functionality of hit-word generator is discussed in Chapter 5. Since the STC card contains eight individual STC channels, a common data bus is used by the control logic to read the hits from the hit-FIFOs from each channel. To avoid contention between various STC blocks, a "data transfer protocol" is adopted. The Table 4.3 and Table 4.4 [2] show the format of hits and hit-trailers.

Table 4.3 Data format for the 32-bit Hit Word

31....26	25..24	22..16	15..13	12.. 0
TRACK	DE/DX	SEQ ID	HDI	CENTROID

Table 4.4 Data format for the 32-bit Hit Trailer

31…27	26..24	23…16	15.. 8	7…4	3..0
11110	-	EVENT	No. of Hits	Misc	-

4.1.4 L3 Buffers

In addition to clustering and finding centroids, the STC also buffers intermediate information throughout the processing of an event. L3_config is a 13-bit word that is used to selectively activate L3 buffering for required channels. Every event is initiated by an "event_start" signal upon which l3_config is latched. The sequence of steps involved in storing the data in the L3 buffer is shown as a flowchart in APPENDIX A.1. The L3 buffer module also allows data to be read independently from the L3 buffers through a 'start_l3' word. Start_l3 is a 10-bit word that can be used to read out data from the selected FIFO buffers. Since there are a total of eight channels that process the data, a "data transfer protocol" very similar to the one used for hit readout is used to control data transfer from L3 buffers. When an L3 buffer is ready for readout, the corresponding STC pulls up its l3_busy signal and waits for data bus to become available. This signal acts like a bus-request. When the bus becomes available, l3_block signal is set high. This signal is used to block the bus from being used by other channels until the whole block of data is read. The sequence of steps involved in putting the content of L3 buffer onto an external data bus is shown in a flowchart in APPENDIX A.2. Types of data that each of the channels can store in the FIFO buffers are hits, raw data, corrected data, strips of the cluster and bad strips. The priority of the channels for L3 data transfer is set externally by using the channel number.

4.2 Implementation of STC in CPLD devices

The preliminary implementation of the STC uses Altera's FLEX20KE PLDs [2] and Altera's Maxplus II design software. This implementation requires three to five FLEX PLDs for fitting the STC. Some of the memory modules are implemented using logic cells instead of the memory elements of the Embedded Array Blocks (EAB) to attain an optimum utilization of available recourses [2]. Using this approach, the design software fits the STC into three FLEX devices. The utilization of the resources among the FLEX devices is shown in Table 4.5 [2]. The usage of multiple FLEX devices in the above approach requires more board space. The total IC pins used in this approach is 829, while the number of pins required for the SOPC implementation is 262, as discussed in Section 4.3. The redundant pins required in the FLEX devices increase the complexity of the board design interconnects. Since several internal connections of the STC run on the PCB, additional propagation delays are also introduced.

Table 4.5 Utilization of the FLEX resources.

Module	Chip	Inputs	Outputs	Memory Bits	Logic cells	EABs
Hitfilter _Schematic	EPF10K100 EBC356-1	77	144	10532 (21%)	4012 (80%)	12 (100%)
L3 _Schematic	EPF10K130 EFC484-1	183	175	40960 (62%)	1576 (23%)	13 (81%)
Strip_reader_ Chip_schematic	EPF10K200 SBC356-1	76	174	45120 (45%)	4773 (47%)	17 (70%)
Total		**336**	**493**	**96612**	**10361**	**42**

4.3 Implementation of STC as an SOPC

This implementation of the STC uses Altera's Quartus II design software and an APEX20KE SOPC device. The STC is modified to fit into the Altera's EP20K600EBC652-1X device. The STC uses Embedded System Blocks (ESB) in the above device to implement memory functions. Table 4.6 shows the APEX resources used by the STC along with the total FLEX resources used for previous implementation. Since only one APEX device is used, the STC consumes less board space and is not affected by the on-board propagation delays. As shown in Table 4.6, the number of pins required in APEX implementation is far less than that required for FLEX implementation. Fewer pins in APEX implementation means that the board design interconnects are less complex.

Table 4.6 Resources utilized by the STC.

Chip Family	Number of Chips	Logic Elements	Memory Bits	Total I/O Pins
FLEX 10KE	3	10,361	96,612	829
APEX 20KE	1	6,744	105,828	262

4.3.1 Validation of SOPC Implementation

The hardware STC card used in the D0 detector consists of one "control logic" and eight STC modules, called "channels", as shown in Figure 4.3. The control and feedback signals between the "control logic" and each of the channels are dedicated,

while a “common data bus” is used for the data transfer (hits) from the channels to the “control logic”.

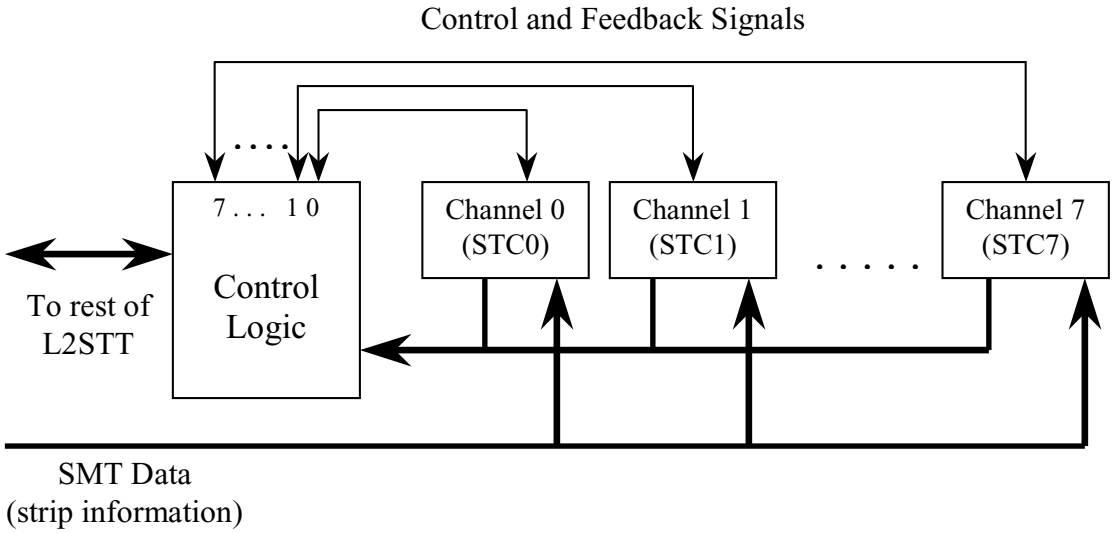


Figure 4.3 The various modules of the STC card

As part of this thesis, the STC was tested at an experimental setup in the HEP, BU, using an STC prototype board. The STC prototype board, shown in Figure 4.4, is designed in the Electronic Design Facility, BU. It contains two STC channels (channel 0 and channel 1) and one “control logic” module. The feedback signals from channels 0 and 1 are connected to the “control logic”. The other inputs of the control logic intended for feedback from the channels 2 through 7 are connected to a common ground. The two STC channels on the prototype board are used to test the data processing and the “data transfer protocol” being used in the “common data bus.”

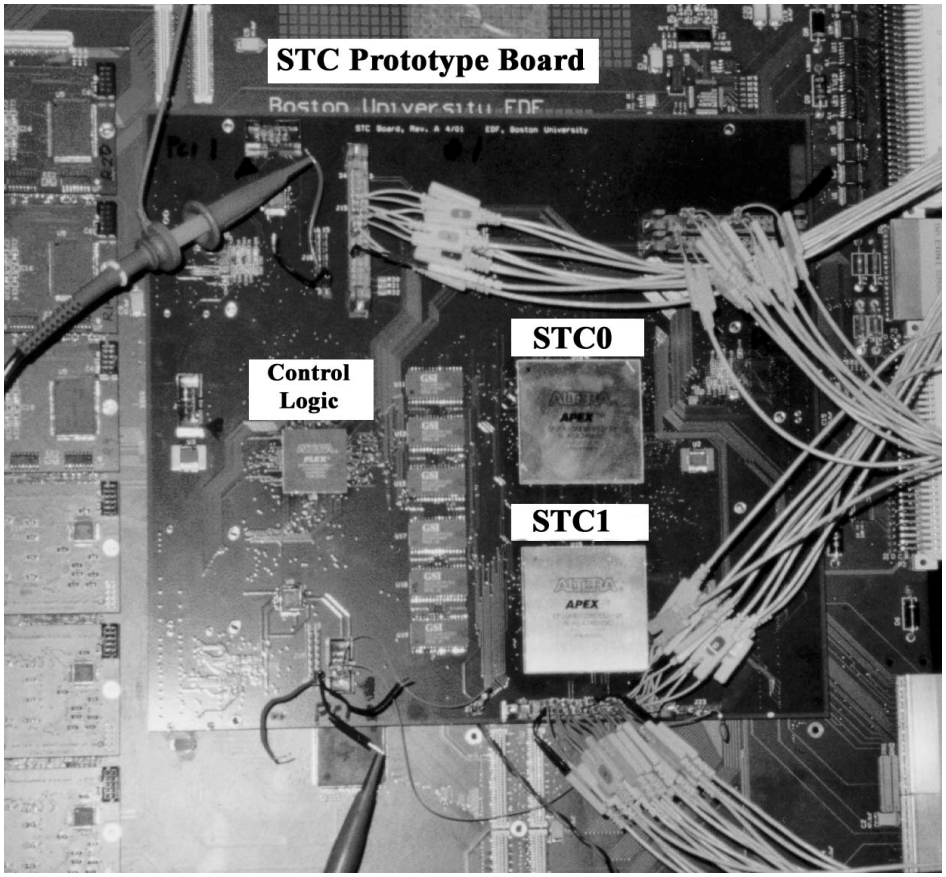


Figure 4.4 The STC prototype board used to validate STC.

The STC channels and the control logic are configured in-circuit into the corresponding devices on the prototype board. The data required for initialization of the event processing is downloaded into the various memory blocks in the “control logic”. All the LUTs in the two STC channels are sequentially loaded. The vector files generated by the researchers at The Florida State University are used to provide input to the channels. The various prototype board signals used to observe the functioning of the STC are shown in Table 4.7 along with their description.

Table 4.7 Signals observed in the Logic Analyzer.

Signal Name		Signal Active Level	Source Module	Description
In Figure	On the board			
RD_WR	road_write	High	Control Logic	Stores roads in the hit-filters of STC0 and STC1.
RD_END	road_end	High	Control Logic	Indicates end of roads.
EV_STA	event_start	High	Control Logic	Starts the event-processing in the channels.
EV_BSY	event_busy	High	STC0 & STC1	High when either of the channels are processing data.
HC_WR	hc_wr	High	STC0 & STC1	High when either of the channels give a write pulse.
ST_HIT	start_hits	High	Control Logic	A pulse in this signal starts hit readout from the channels.
HC_BY0	hc_busy0	High	STC0	This signal acts like "bus-request" for STC0.
HC_WR0	hc_wr0	High	STC0	A write signal from the STC0 after putting data onto the common data bus.
HC_WR1	hc_wr1	High	STC1	A write pulse issued by STC1 after putting data onto the common data bus.
HC_BY1	hc_busy1	High	STC1	This signal acts like "bus-request" for STC0.

The testing of the STC was done using test vectors generated by the researchers at the HEP, FSU. The test-vector of a simple event is used to show the various stages of STC operation, while the test-vector of a complex event is used to show the hit-readout in more detail. Figure 4.5 shows an instance of the test with simple event, captured through the logic analyzer. The encircled parts '1' and '2' in the Figure 4.5 are the event-initiation sequence and the hit-readout sequence respectively. The test-vector is downloaded into

the test-FIFO initiating the event processing. The event-initiation sequence seen in encircled part 1 is briefly described.

1. The ‘EV_STA’ pulse initiates the event processing. In return, the channels pull up ‘EV_BSY’ signals to indicate the busy state. This signal remains ‘HIGH’ until all the channels have processed the strip data.

2. RD_WR and RD_END are used to write the roads into the hit-filters of the two channels.

3. ST_HIT signal initiates the transfer of hits from the STC channels. However, hit-readout sequence doesn’t start until the hits are stored in the hit-FIFO.

After the initial steps, hit-filter waits until the first centroid is calculated. The hit-filter then finds hits for each of the centroids and stores in hit-FIFO. As soon as the first hit is stored in the hit-FIFO, hit-readout sequence commences, as shown in encircled part-2 of Figure 4.5. When multiple channels report “hits” in the same clock cycle, channel with lowest number is given priority. Thus, channel 0 is not affected by any other channels, while channel 1 is affected by channel 0 only. This sequence is explained in more details using a complex event. In the Figure 4.5, four pulses in HC_WR0 indicate that STC0 has four hit-words (three hits and one hit-trailer). Similarly, two pulses in HC_WR1 indicate that STC1 has two hit-words (one hit and one hit-trailer).

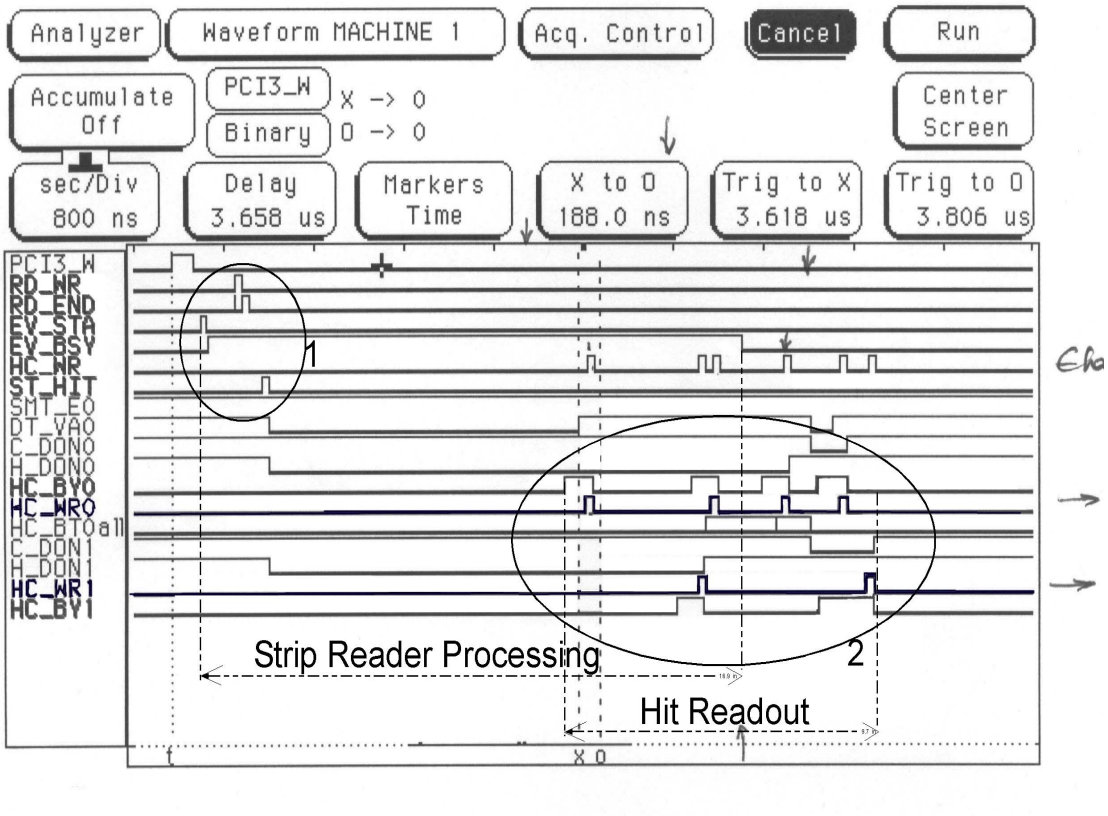


Figure 4.5 Logic analyzer display showing the prototype board signals for a simple event.

Figure 4.6 shows the hit-readout sequence during the test with a complex event. The highlighted signals, also described in Table 4.7, are used to verify the "data transfer protocol." The hit-readout sequence as seen in Figure 4.6 is briefly described.

1. The STC1 is the first to report a "hit", thus it pulls up the HC_BY1 signal first to request access to the common data bus. Since the STC0 doesn't have a "hit" at this instant, STC1 is granted the bus control.

2. The STC0 reports a “hit” in the next clock cycle. Since STC0 has the priority, it prepares to upload the hit. However, STC0 is lagging behind the STC1 by a clock cycle and thus does not contend at the same time.

3. After STC1 uploads the “hit” onto the data bus, it sends a pulse of HC_WR1 for the “control logic” to latch on the data. In the very next clock cycle STC0 uploads its “hit” onto the data bus and sends a pulse of HC_WR0.

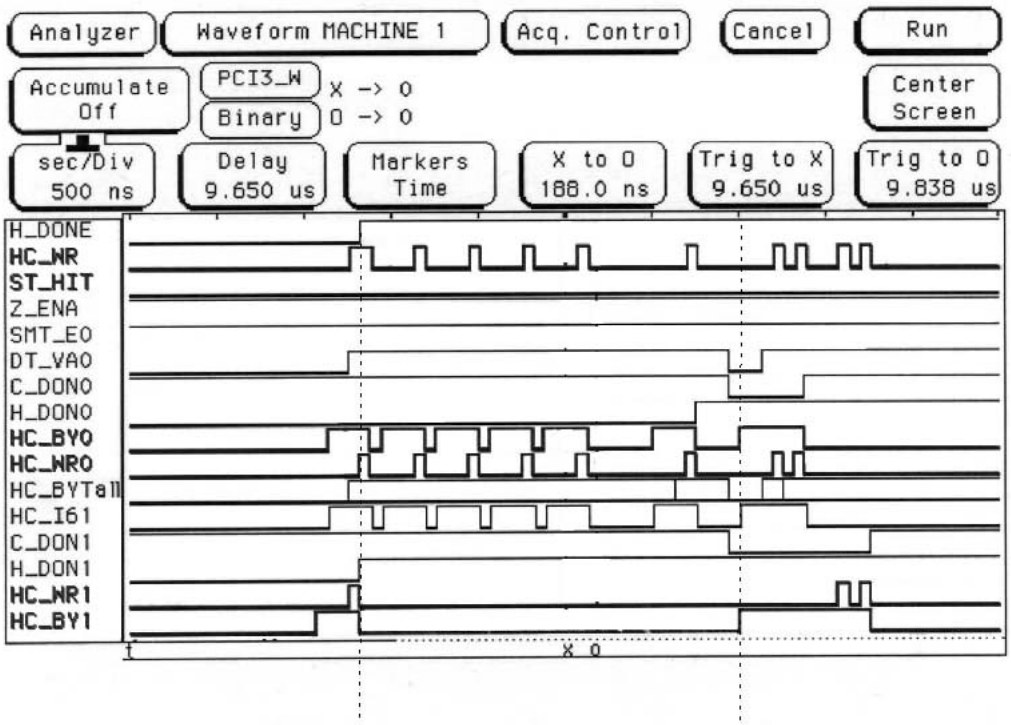


Figure 4.6 Logic Analyzer display showing the hit-data transfer

4. In this instance, other “hits” in STC0 are immediately available while STC1 takes more clock cycles to find remaining “hits”. STC0 thus keeps uploading the hits and sending pulses of HC_WR0.

5. The seventh hit of STC0 and second hit of STC1 are reported at the same time by pulling up the bus-request signals (HC_BY0 and HC_BY1). Since STC0 has higher priority, STC1 waits with the HC_BY1 high until STC0 uploads the seventh hit and hit-trailer.

6. STC1 now takes control of the data bus and uploads the second hit and hit-trailer.

This particular test-vector yields eight “hit words” in STC0, seven of which are the “hits” while the last word is a “hit-trailer”. Similarly STC1 yields three “hit words”, two of which are the “hits” while the third word is a “hit trailer”. It can also be observed that the “data transfer protocol” successfully resolves contention between the two STC channels. The functionality of the STC has thus been successfully tested.

CHAPTER 5

IMPLEMENTATION WITH CONTENT ADDRESSABLE MEMORY

A Random Access Memory (RAM) memory accepts an address of the data and returns the data. In a RAM, given the location of the data, retrieving the data takes the same time irrespective of the location. However, given the data itself, finding the location of the data requires sequential search through all the locations until the data is found. This search operation thus takes a number of clock cycles in a conventional memory block. The Content Addressable Memory (CAM) is a type of memory that accepts data and returns the corresponding location. The time required to search for the data in the CAM is same for data present anywhere in the memory block, while the time required for searching a RAM is proportional to the number of memory words stored. CAMs are extensively used for applications that require reverse-lookup, fast searching and matching of the data.

Figures 5.1 and 5.2 show a "4 X 3 CAM" containing 4, 7, 1 and 0 in binary format. The output "found" of the CAM goes to '1' when the given data is present in the memory block. The CAM blocks provide a valid location of the data word when "found" signal is '1'. Given a binary word as input, the CAM can return either the unencoded or encoded location of the data. Figure 5.1 shows a CAM returning the unencoded location

while Figure 5.2 shows a CAM returning encoded location of the data. It can also be observed that both the blocks return a valid location, accompanied by a '1' in "found" signal, for the data words "001" and "100". They return an invalid location, represented as "X" and accompanied by '0' in "found" signal, for the other words.

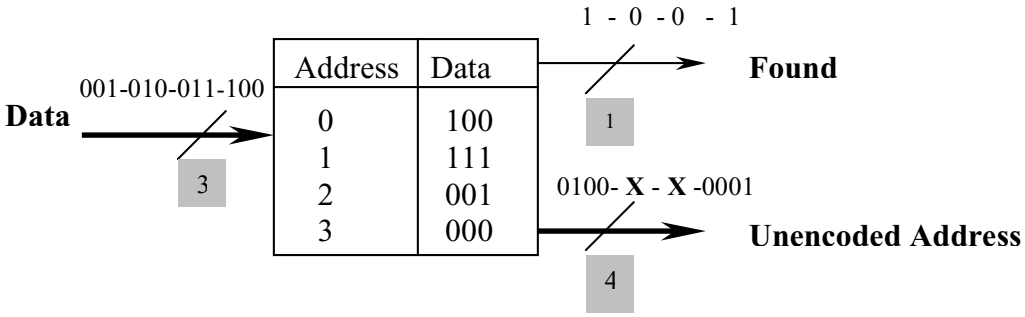


Figure 5.1 A Simple CAM block returning unencoded output

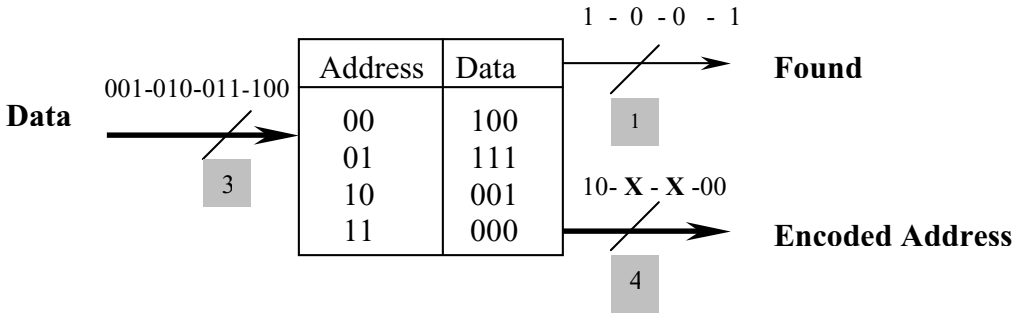


Figure 5.2 A Simple CAM block returning encoded output

While a simple CAM can hold logic levels of '0' and '1', Ternary CAMs can also hold "don't care" (***d***) values. A CAM containing "don't cares" in a particular bit location, also represented with a '***d***', returns a match for both the logic levels. Multiple data words

can be represented by fewer data words by using the “don’t cares”. For example, numbers from 1 through 7 can be represented by three words containing “don’t cares”, as shown in Table 5.1. The table also shows representation of the multiples of 4 as a single word. The data discussed above can be stored in a Ternary CAM, so that a search can be performed in minimal time. In addition, a Ternary CAM needs fewer entries for applications involving searching and matching of data.

Table 5.1 Data stored in the Ternary CAM shown in Figure 5.3

Address (binary)	**Data represented in the CAM**		**Equivalent Word**
	decimal	**binary**	
00	1	0001	0 0 0 1
01	2, 3	001**0** 001**1**	0 0 1 ***d***
10	4, 5, 6, 7	01**00** 01**01** 01**10** 01**11**	0 1 ***d d***
11	0, 4, 8, 12	**00**00 **01**00 **10**00 **11**00	***d d*** 0 0

Figure 5.3 shows a “4 X 4 Ternary CAM” that can provide an encoded location of the given data. The CAM contains equivalent words shown in Table 5.1. An input of “1100” to the CAM fetches a ‘1’ in “found” signal and an encoded address of “11” in the address bus. The input “1001” finds no match, while input “0100” finds two matches in “10” and “11” respectively. Since the CAM uses the “found” signal and the “address” bus, it can be said to be operating in “search mode” as well as “reverse-lookup mode”.

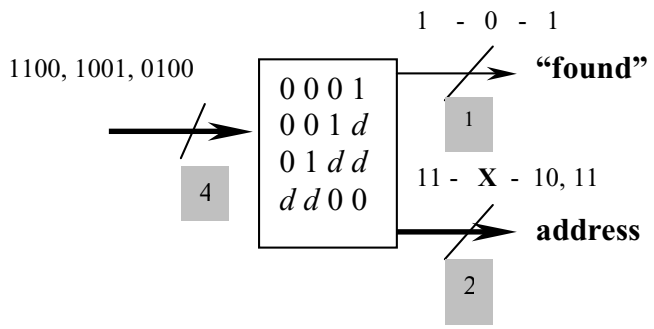


Figure 5.3 Encoded output of a Ternary CAM containing "don't cares".

5.1 APEX CAM

The CAM blocks are available as discrete components that can be externally connected to the logic module. Since the external signals travel on the PCB, they have an associated time delay. However, integration of the CAMs into the PLDs drastically reduces the time delay and saves the board space on the PCB. In Altera's Quartus II, the APEX CAM is implemented by using the Altera's "altcam" megafunction [22] and the ESBs of the APEX devices. The APEX CAMs can be configured to accommodate any configuration between 32 X 4096 and 4096 X 32. The Quartus II software cascades ESBs to implement wider and deeper CAMs, however, wider CAMs cannot provide encoded output.

The APEX CAM can support "don't cares" [22] and thus allows designer to efficiently use the memory resources. The contents of the CAM can be written either during power-up or during the normal operation of the CAM. A memory initialization file(.mif) or a intel hex file can be used to initialize the memory during power-up. "Don't

cares" can also be written into the CAM using the initialization files. Writing the data into the CAM after power-up requires two clock cycles for words not containing "don't cares" and three clock cycles for words containing "don't cares." The APEX CAM can be used in three modes depending on the application.

5.1.1 Single-Match Mode

In the single-match mode, the APEX CAM requires only one clock cycle to return the data location [22]. However, this CAM can be used only when the stored data is unique. When same data word is stored in multiple locations, the CAM returns the last location that contains the data. In this mode, each ESB in the CAM can accommodate as many as 32 words with 32 bits each [22].

5.1.2 Multiple-Match Mode

In the multiple-match mode, CAM can contain same data words in multiple locations. In this mode, all the locations containing a data word can be readout sequentially. For each data word, the CAM takes two clock cycles to return the first location and one clock cycle for the subsequent locations. Each ESB of a CAM in multiple-match mode can accommodate 32 words with only 31 bits in each word [22].

5.1.3 Fast Multiple-Match Mode

In fast multiple-match mode, the CAM can contain the same data in multiple locations like in multiple-match mode. In addition, for each data word, it takes only one clock cycle to return the first location and one clock cycle each for subsequent locations.

However, in this mode, each ESB of the CAM can accommodate only 16 words with 32 bits in each word [22].

5.2 Implementation of Hit-Filter

As discussed in the Section 4.1.3, hit-filter takes a centroid and finds if it falls between the two road boundaries, the upper-address and the lower-address. This can be implemented either by using a comparator and an encoder logic, like in previous implementation, or by using a ternary CAM module alone for the whole hit-filter functionality. Section 5.2.1 in this chapter discusses the CAM-only implementation, while Section 5.2.2 discusses usage of CAM as an encoder in the hit-filter

5.2.1 Hit-filter containing only a CAM

Instead of using a combinational logic to check if a centroid falls within boundaries of given roads, the current approach uses memory to store the whole set of words occurring between the upper-address and lower-address of a road. The upper-address and lower-address are two strips that fall on either sides of a road. Thus, the set of all the words falling between the two digital words represent each and every strip falling in the given road. This set of digital words representing all the strips of a road is called a “road-set”. Each word of the road-set has the same format as that of the upper-address and the lower-address, shown in Table 5.2

Table 5.2 Distribution of bits in the 11-bit upper address and lower address

10 .. 7	**6 ... 0**
Chip ID	Strip address

A “road” can span across two adjacent chips [18] though it is mostly restricted to the same chip. In order to simplify the road-sets, roads spanning across chips are represented by different road-sets, each road-set representing the road in a particular chip. Thus, 11-bit road-set words contain a constant 4-bit chip ID and a variable 7-bit strip address. Since, only the chip ID and strip-address of the centroid are used in the hit-filter, centroid is effectively 11 bits wide in this module.

Since the number of words in a road-set can reach a maximum of 2^7 (128) words, a scheme is devised to represent the road-set in as few words as possible. This scheme uses “don’t cares” to represent the road-set in a maximum of 12 words. The flowchart shown in APPENDIX A.3.details the sequence of steps used to generate the minimized road-set for each road. As a first step, the highest changing bit, called “highest-bit”, in the whole road-set is calculated. The example in Table 5.3 shows the whole road-set for a set of road boundaries. As seen in this table, bits 0 through 3 are variable, while bits 4 through 10 are constant. Thus the “highest-bit” is bit3.

Table 5.3 Road-set showing the variable and constant bits of a road

	Chip ID (10 9 8 7)	**Strip address** (6 5 4 3 2 1 0)
Lower-address	1 0 0 0	1 0 1 **0 0 1 0**
	1 0 0 0	1 0 1 **0 0 1 1**
	1 0 0 0	1 0 1 **0 1 0 0**
	1 0 0 0	1 0 1 **0 1 0 1**
	1 0 0 0	1 0 1 **0 1 1 0**
	1 0 0 0	1 0 1 **0 1 1 1**
Upper-address	1 0 0 0	1 0 1 **1 0 0 0**

The lower-address and the upper-address are XORed as shown below. The highest bit containing '1' is the "highest-bit" for the given road-set. Thus, for the road-set shown in Table 5.3, highest-bit is found to be bit3.

$$\begin{array}{rl} 101\mathbf{1000} & \text{- Upper word} \\ \oplus\, 101\mathbf{0010} & \text{- Lower word} \\ \hline 000\mathbf{1010} & \text{Highest - bit is BIT3} \end{array}$$

After finding the "highest-bit", the road-set generator generates the minimized road-set. In a worst-case situation, the seven variable bits of the lower-address and the upper-address will be "0000001" and "1111110". Table 5.4 shows the minimized road-set for this situation.

The Figure 5.4 shows the hit-filter module using only CAM blocks. The "road-set generator" is designed in VHDL to generate the minimized road-set. The CAM module functions in multiple-match mode so that each ESB can accommodate 32 words of width 31 bits. For optimal usage of the resources, a block of 16 locations is assigned to each

road-set and two road-sets are designed to fit into a single ESB. Sixteen memory locations are allotted to each road-set, so that the lowest four bits of the CAM addresses can represent locations within the same road-set. When a centroid is given as input to the CAM containing all the road-sets, the lower four bits of the output are removed to find the road-set in which the centroid falls. The actual road number and thus the track number can be identified by keeping track of the number of road-sets used to represent each road.

Table 5.4 Minimized road-set for the worst-case situation

	Actual road-set	**Minimized road-set**
1	0 0 0 0 0 0 1	0 0 0 0 0 0 1
2	0 0 0 0 0 1 0 0 0 0 0 0 1 1	0 0 0 0 0 1 ***d***
3	0 0 0 0 1 0 0 : 0 0 0 0 1 1 1	0 0 0 0 1 ***d d***
4	0 0 0 1 0 0 0 : 0 0 0 1 1 1 1	0 0 0 1 ***d d d***
5	0 0 1 0 0 0 0 : 0 0 1 1 1 1 1	0 0 1 ***d d d d***
6	0 1 0 0 0 0 0 : 0 1 1 1 1 1 1	0 1 ***d d d d d***
7	1 0 0 0 0 0 0 : 1 0 1 1 1 1 1	1 0 ***d d d d d***
8	1 1 0 0 0 0 0 : 1 1 0 1 1 1 1	1 1 0 ***d d d d***
9	1 1 1 0 0 0 0 : 1 1 1 0 1 1 1	1 1 1 0 ***d d d***
10	1 1 1 1 0 0 0 : 1 1 1 1 0 1 1	1 1 1 1 0 ***d d***
11	1 1 1 1 1 0 0 1 1 1 1 1 0 1	1 1 1 1 1 0 ***d***
12	1 1 1 1 1 1 1	1 1 1 1 1 1 0

Since the APEX CAM requires three clock cycles to write each of the words, storing the whole road-set may require up to 50 clock cycles, including the cycles required for the state machine of the "road-set generator." Repeating this scheme for all the 46 roads requires 2070 clock cycles as discussed in Section 5.3.

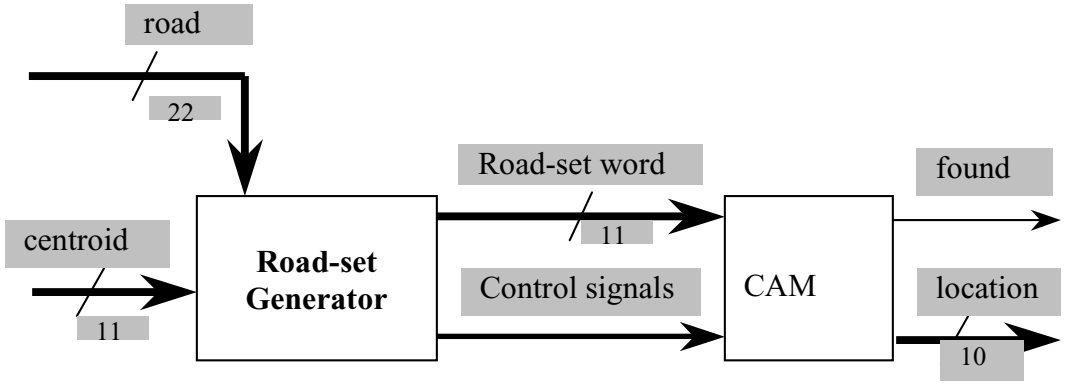


Figure 5.4 The hit-filter containing a CAM and road-set generator.

While checking for hits, the CAM gives out a 10-bit location of the centroid, if present. The upper six bits indicate the road-set number while the lower four bits indicate the exact position of the centroid in a road-set, as shown in Table 5.5. The 6-bit road-set number can be used to find the track number for generating the hit-word. Thus, the CAM itself acts an encoder by providing the road-set number. The CAM in this implementation takes two clock cycles to give the first location, and takes one clock cycle each for the remaining locations.

Table 5.5 Distribution of bits in the CAM output

9……4	3 … 0
Road-set Number	Location in the road-set

5.2.2 Implementation of hit-filter with CAM as Encoder

In this implementation, the hit-filter uses a similar setup as in the preliminary STC. It uses the "comparator" along with a "hit-word generator" which contains CAM blocks as shown in the Figure 5.5. The locations of '1's in the 46-bit comparator word are encoded to find the track-numbers associated with the give centroids. The APEX Ternary CAM with encoded output is used for this purpose.

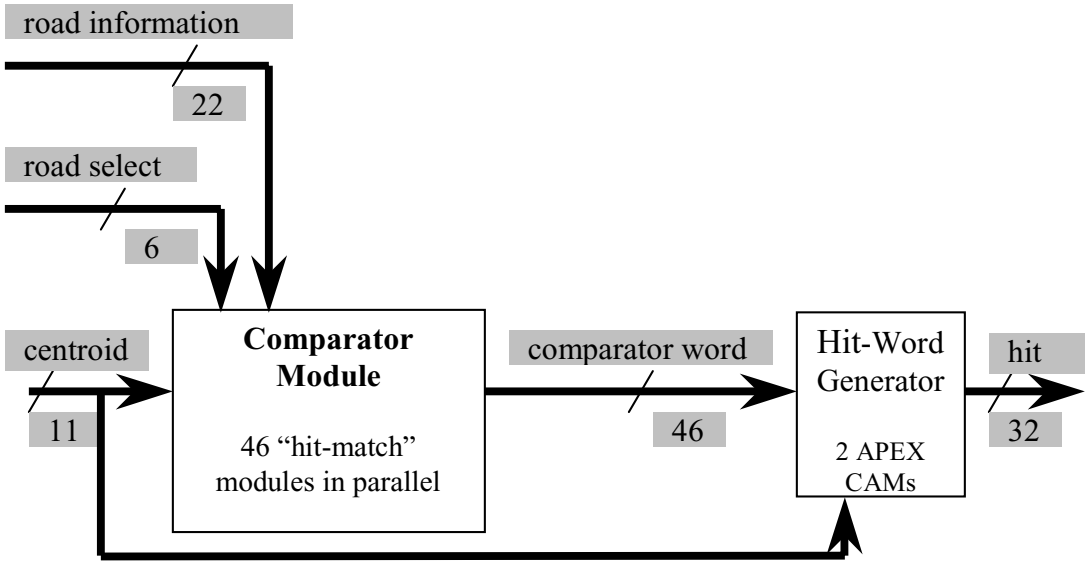


Figure 5.5 New hit-filter module using the "hit-word generator."

The data content of the "4 X 4 CAM" shown in Figure 5.6 is chosen such that location of '1's in each of the words is same as the location of the data word itself in the CAM. Rest of the bits in each of the data words are filled with "don't cares" (***d***). For example, the data word in location 0 is "***d d d*** 1", where only the bit0 has a '1'. Since rest of the bits are "don't cares", the CAM returns a match whenever there is a '1' in bit0, irrespective of the other bits in the input word. The CAM can also return the encoded location of the data word. In case of an input with multiple active bits, like "1 0 0 1", CAM in Figure 5.6 returns encoded locations of all the '1's sequentially. This set of data words stored in the CAM is called a 4-bit "encoder-map."

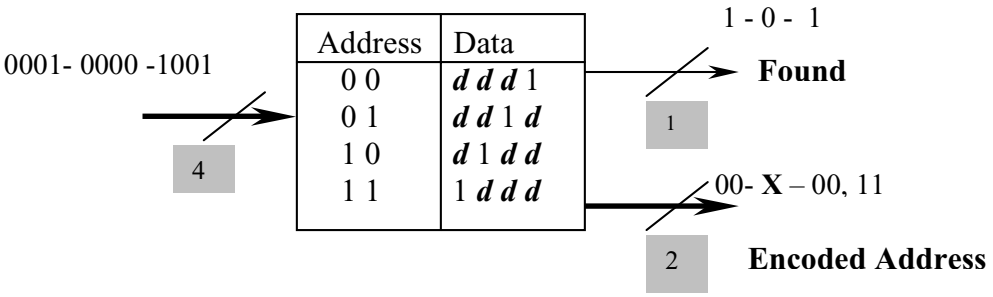


Figure 5.6 A "4 X 4 Ternary CAM" and its Encoder-map

The encoder-map can be extended to accommodate all the 46 bits of the comparator word. However, APEX CAM cannot provide an encoded output for CAMs wider than 31 bits [22] owing to the limitations on the ESB blocks. Thus, the 46-bit encoder-map is broken into two smaller maps of 31 and 15 bits respectively. The block diagram of this implementation is shown in the Figure 5.7.

The two APEX CAMs with configurations “31 X 31” and “15 X 15”, are used in multiple-match mode for this purpose. A “hit-generator” block combines the encoded addresses from the two CAMs and generates the actual track-number. The output from “31 X 31 CAM” is directly used to generate a hit-word, while 31 (011111) is added to the output from “15 X 15 CAM”, before using it to generate a hit-word. The Table 5.5 shows the 46-bit encoder-map used in the CAM blocks. As shown, the actual 46-bit encoder-map is broken into two smaller encoder-maps. The two smaller maps are highlighted in the table below. Two “.mif” files are used to store these encoder-maps during device power-up.

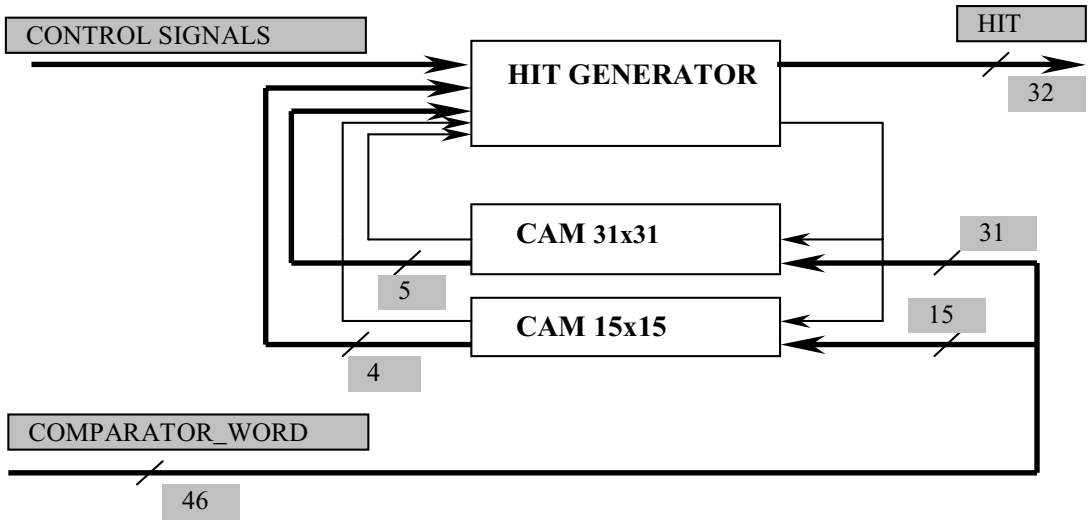


Figure 5.7 Hit-word generator using two CAM blocks.

Table 5.6 Distribution of 46 bit word across two CAMs

		45 44 43 42 41……...28 29 30 31	30 29 28 27 26 ……..4 3 2 1 0
			30----0
0	**0**	*d d d d d . d d d d d*	*d d d d d . d d d d d* **1**
1	**1**	*d d d d d . d d d d d*	*d d d d d . d d d d* **1** *d*
2	**2**	*d d d d d . d d d d d*	*d d d d d . d d d* **1** *d d*
		d d d d d . d d d d d	*d d d* **CAM 31** *d d d*
		d d d d d . d d d d d	*d d d* 30…0 *d d d*
		. . . .	*. . .* **CAM 31** *. . .*
		d d d d d . d d d d d	*d d d* 30…0 *d d d*
		d d d d d . d d d d d	*d d d* **CAM 31** *d d d*
28	**28**	*d d d d d . d d d d d*	*d d* **1** *d d . d d d d d d*
29	**29**	*d d d d d . d d d d d*	*d* **1** *d d d . d d d d d d*
30	**30**	*d d d d d . d d d d d*	**1** *d d d d . d d d d d d*
		14 ------0	
3	**0**	*d d d d d . d d d d* **1**	*d d d d d . d d d d d d*
1	**1**	*d d d d d . d d d* **1** *d*	*d d d d d . d d d d d d*
3		*d d d* **CAM 15** *d d*	*d d d d d . d d d d d d*
2		*d d d* 14….0 *D d*	*d d d d d . d d d d d d*
		d d d **CAM 15** *D d*	*d d d d d . d d d d d d*
		. . . **CAM 15** *. .*	*. . . . .*
		d d d 14….0 *D d*	*d d d d d . d d d d d d*
		d d d **CAM 15** *D d*	*d d d d d . d d d d d d*
43	**12**	*d d* **1** *d d . d d d D d*	*d d d d d . d d d d d d*
44	**13**	*d* **1** *d d d . d d d D d*	*d d d d d . d d d d d d*
45	**14**	**1** *d d d d . d d d D d*	*d d d d d . d d d d d d*

5.3 Results

The various hit-filter implementations are tested with a test-vector representing SMT data of a simple event. The number of roads used for this event is varied to compare the performance for various cases.

1 Six Consecutive Hits: This case results when the STC finds “hits” in the first six stored roads.

2 Six Distributed Hits: This case results when the STC finds "hits" in the non-adjacent roads. In the worst-case situation that is being considered, the last hit is found in the 46^{th} road.

3 Forty Six Hits: This case results when STC finds "hits" in all the 46 roads.

The Tables 5.7 and 5.8 show the clock cycles required for storing the roads and identifying the roads that contain "hits". This does not include the rest of the STC processing. The three hit-filter implementations compared in the tables are

1. Hit-filter using comparator and a sequential search module (old implementation).

2. Hit-filter completely implemented with CAM

3. Hit-filter using comparator from previous hit-filter and a new hit-word generator containing CAM.

Table 5.7 Number of clock cycles required for storing the roads.

No.of Hits / **Hit-filter Implementation**	**6 (consecutive)**	**6 (distributed)**	**46**
Sequential search (contains comparator)	6	46	46
CAM only	270 *	310 *	2070*
With CAM block in hit-word generator (contains comparator)	**6**	**46**	**46**

* This depends on the upper and lower words of the road. The quoted figures correspond to the worst possible case.

Table 5.8 Number of clock cycles required for finding the hits

No.of Hits / **Hit-filter Implementation**	**6 (consecutive)**	**6 (distributed)**	**46 roads**
Sequential search (contains comparator)	32	150	232
CAM only	6	6	46
With CAM block in hit-word generator (contains comparator)	**10**	**10**	**50**

As seen in the Table 5.7 and Table 5.8, hit-filter block using a CAM-only implementation takes a very long time to store the roads, while the sequential-search implementation takes a long time to find the "hits."

Two trial events provided by the researchers from HEP group, are used for a more realistic testing of the complete STC module. The events "event1" and "event2" represent the SMT data for simple and complex cases respectively. The two implementations tested are the previous STC with sequential search and the upgraded STC using a comparator in conjunction with a CAM. Table 5.9 shows the number of clock cycles required for the "event1" and "event2" and also shows the improvement in performance of the upgraded STC over the previous implementation. The performance is measured in terms of the number of clock cycles taken for the STC to process the incoming SMT data and to store the last road-word in the hit-FIFO. Table 5.10 shows the performance in terms of the time taken with the system clock of 33 MHz.

Table 5.9 Performance of STC module in terms of number of clock cycles

Block	STC	6 consecutive hits		46 hits		6 distributed hits	
		Event1	Event2	Event1	Event2	Event1	Event2
STC	Previous	161	497	544	2509	384	1709
	Upgraded	133	228	173	629	133	229
% Improvement		**121%**	**217%**	**314.4%**	**398.9%**	**288.7%**	**749.5%**

Table 5.10 Performance of the STC modules in terms of time taken (µs)

Block	STC	6 consecutive hits		46 hits		6 distributed hits	
		Event1	Event2	Event1	Event2	Event1	Event2
STC	Previous	4.878µs	15µs	16.48µs	76.03µs	11.636µs	51.78µs
	Upgraded	4.03µs	6.909µs	5.242µs	19.06µs	4.03µs	6.909µs
% Improvement		**121%**	**217%**	**314.4%**	**398.9%**	**288.7%**	**749.5%**

In the preliminary implementation, in order to encode the active bits of the comparator word, the hit-filter sequentially searches all the used comparator bits. Thus, the time required for a finding “hits” is approximately the same even when there are no “hits”. This situation is aggravated when the hits associated with the event are distributed. However, in the new implementation, before encoding the active bits, the hit-filter can find if there are any active bits (‘1’s) in the comparator word. Thus, the time required to identify the hits is proportional to the number of “hits”.

CHAPTER 6

CONCLUSIONS

6.1 Conclustions

The STC has been successfully implemented as a System-on-Programmable-Chip. The SOPC implementation extensively uses the Embedded System Blocks of the Altera's APEX device for memory and requires only one APEX device. This implementation uses a smaller area on the Printed Circuit Board and requires a fraction of the user pins required in the previous implementation. This makes the board-design interconnects less complex. The hardware validation in Boston University has shown that the STC meets the specified design requirements. In addition, the STC validation has shown that the data-transfer protocol successfully resolves the contention between the STC modules.

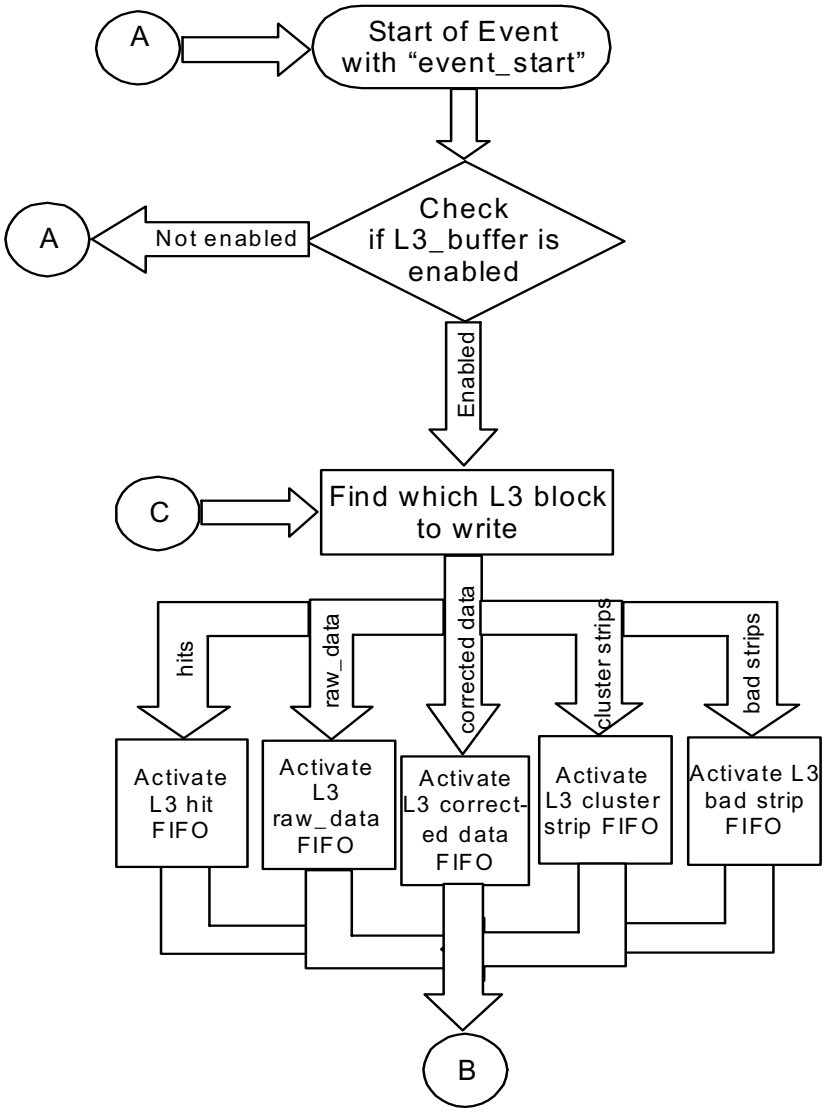
Though, the CAM-only implementation of the hit-filter module was found to be taking less time to find "hits", the prohibitively long time required to store the roads makes this implementation unsuitable for the STC. The alternative implementation of the hit-filter module uses the comparator and a new "hit-word generator." In this implementation, the time taken to find the hits is proportional to the number of hits, while in Version 1.0 of the STC the time taken for finding the hits, when present, is same irrespective of the number of the hits present. The timing simulations of the STC with this hit-filter implementation have shown considerable improvement in the time required

for processing the events. An improvement of up to 87% has been observed in the time taken to find the hits.

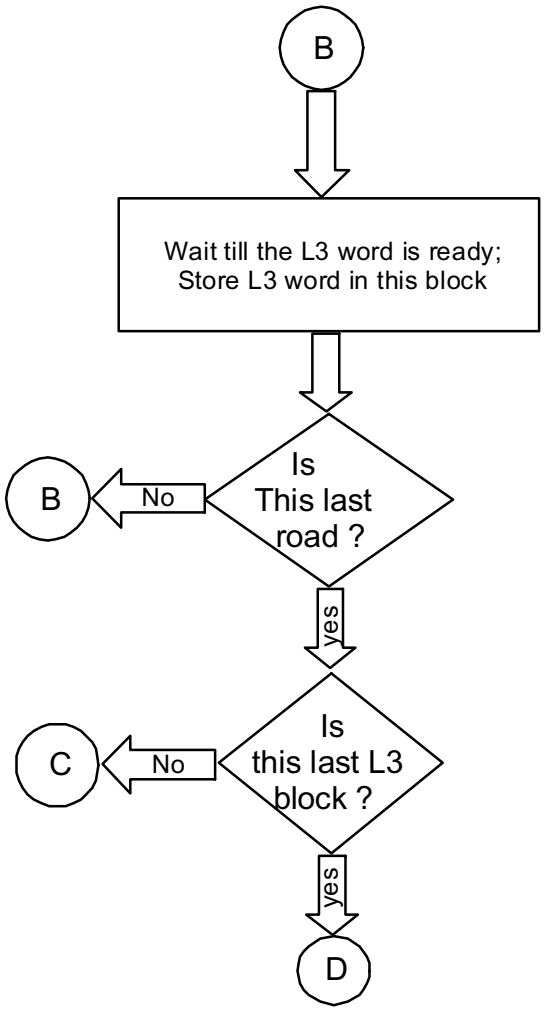
APPENDIX A

FLOWCHARTS OF STC MODULES

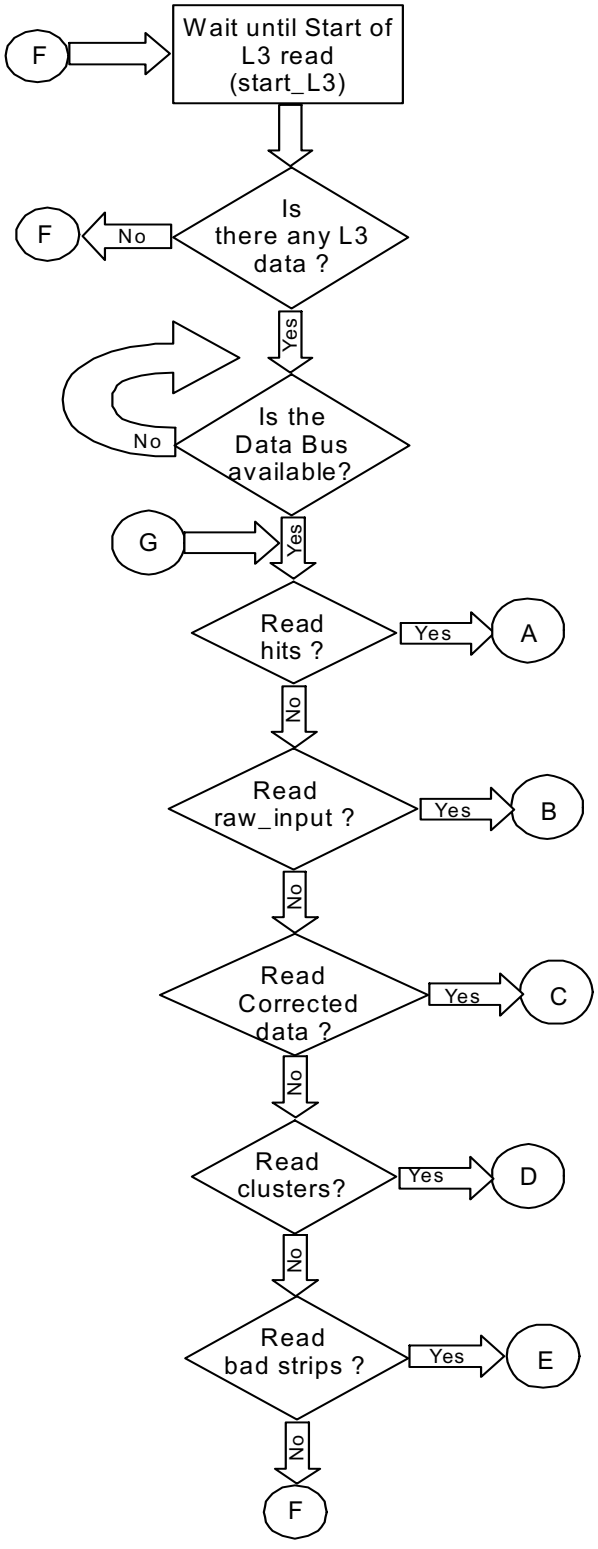
A.1 L3 module while storing data in the buffer.



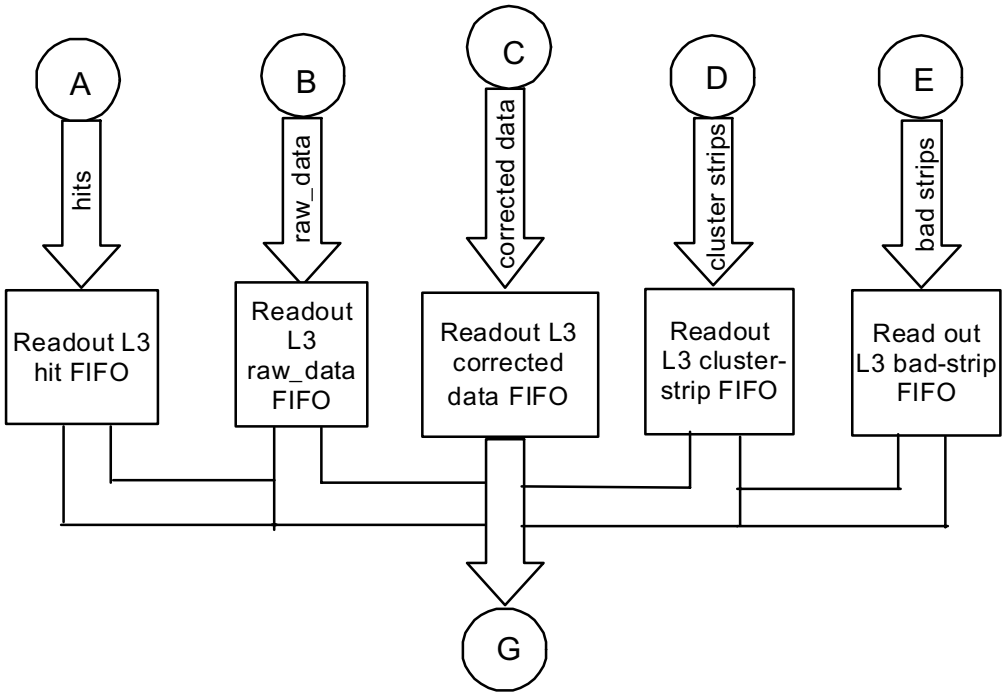
A.1 L3 module while storing data in the buffer. (continued..)



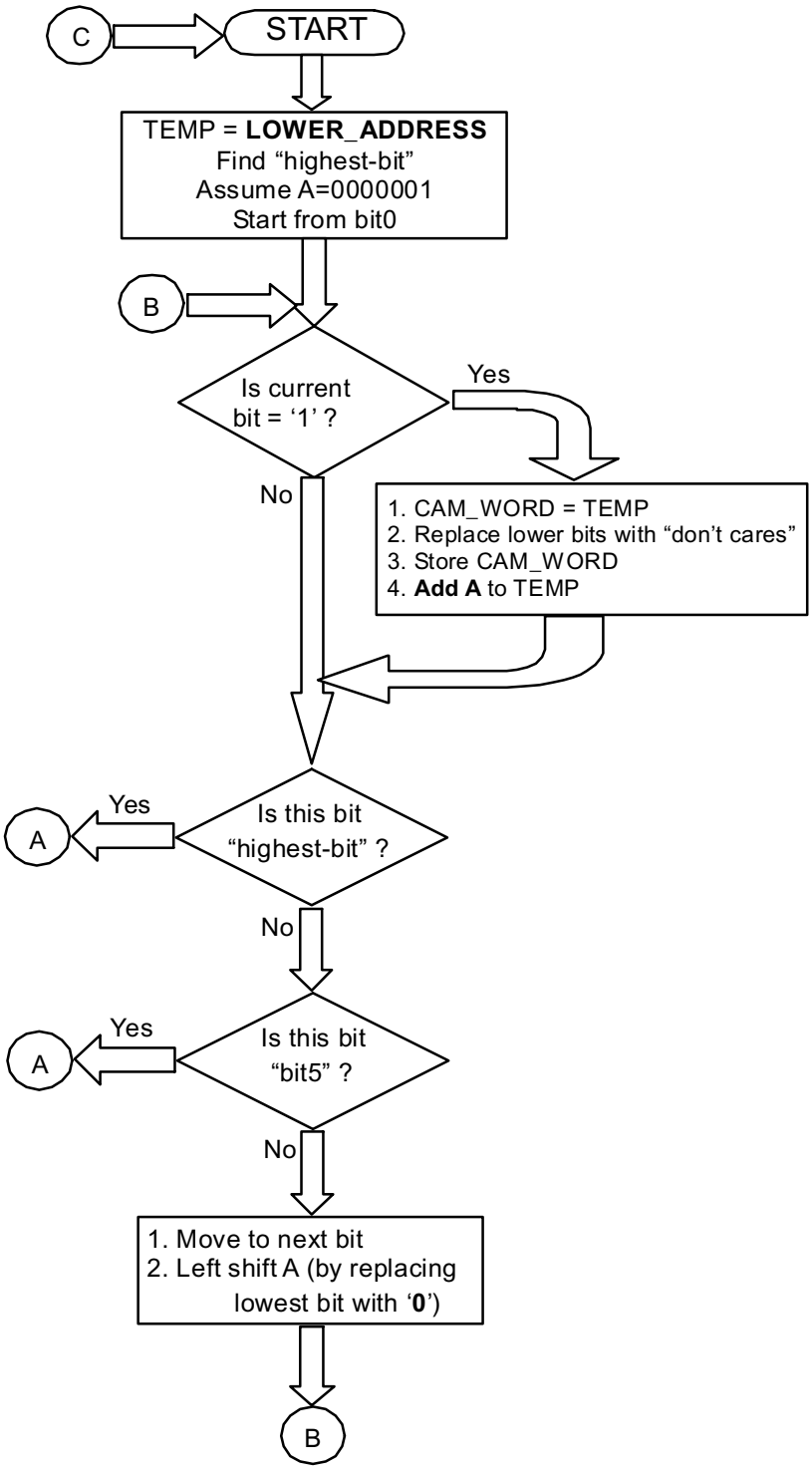
A.2 L3 module while reading out data to an external bus.



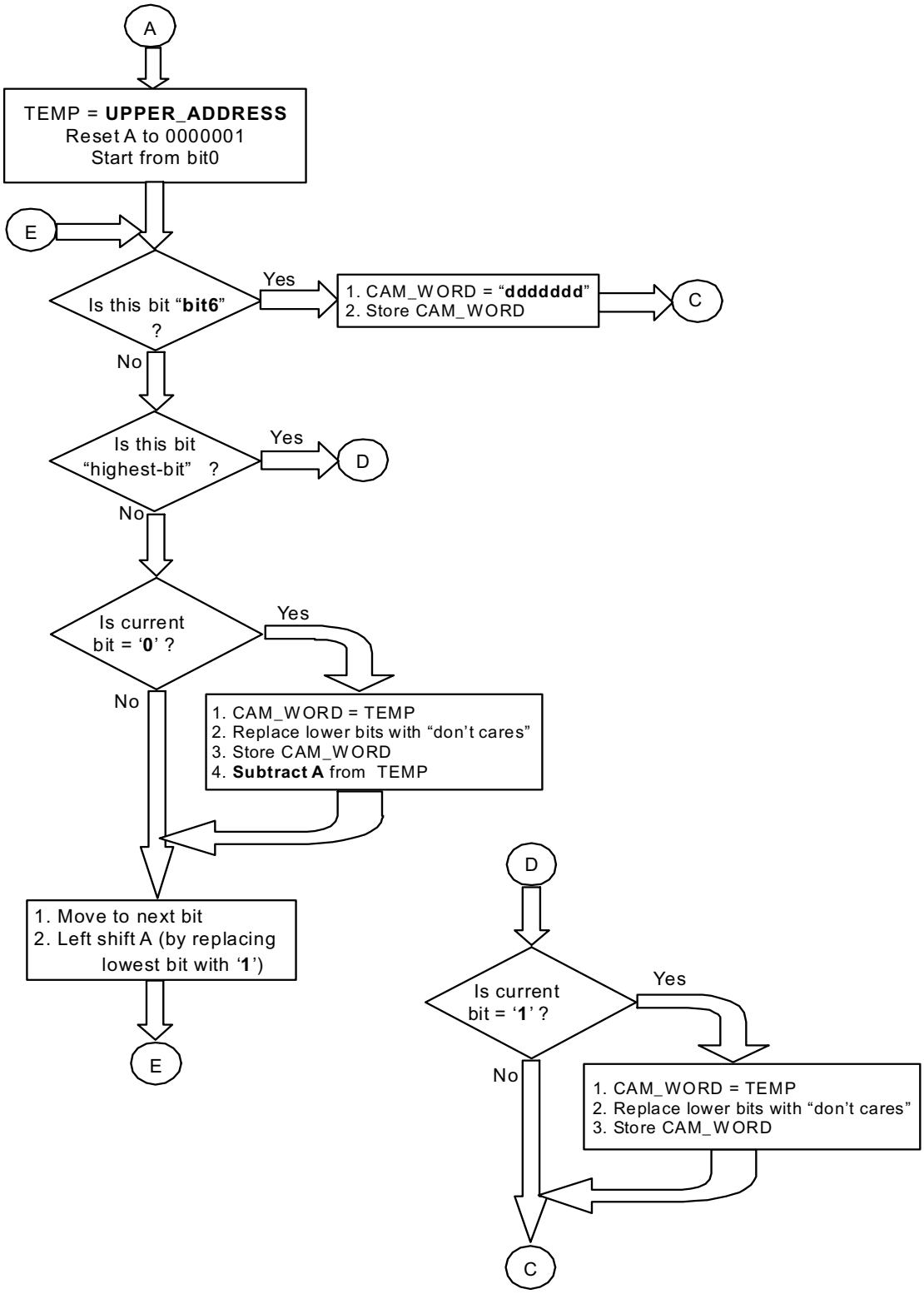
A.2 L3 module while reading out data to an external bus (continued.)



A.3: Road-Word Generator Block



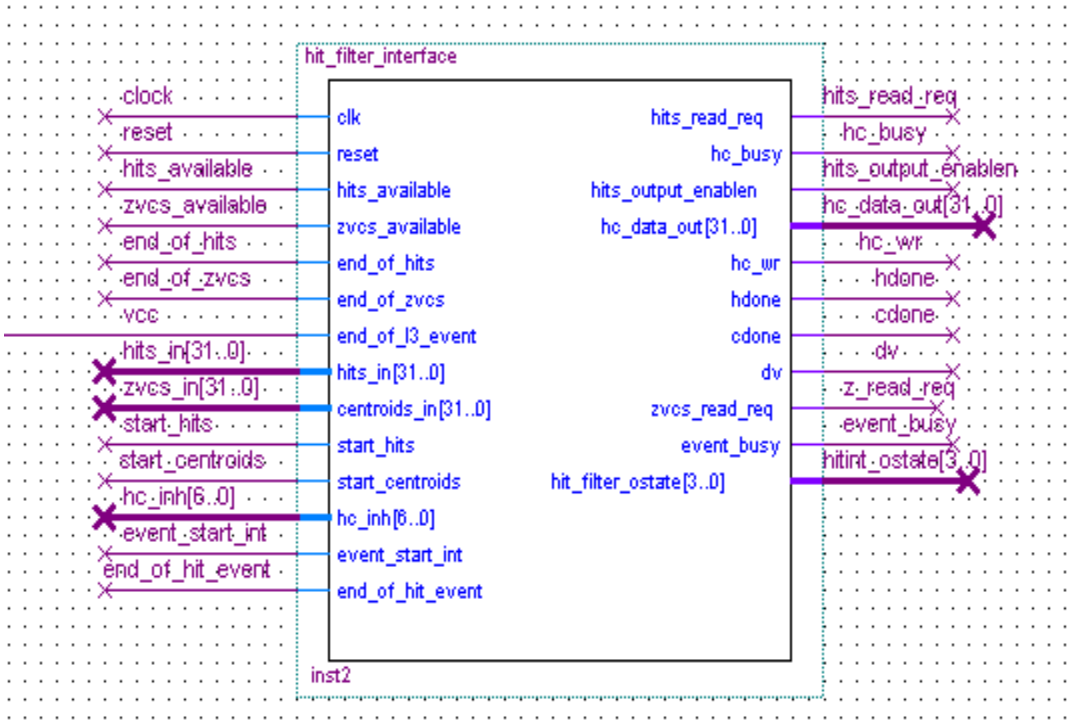
A.3: Road-Word Generator Block (continued)



APPENDIX B

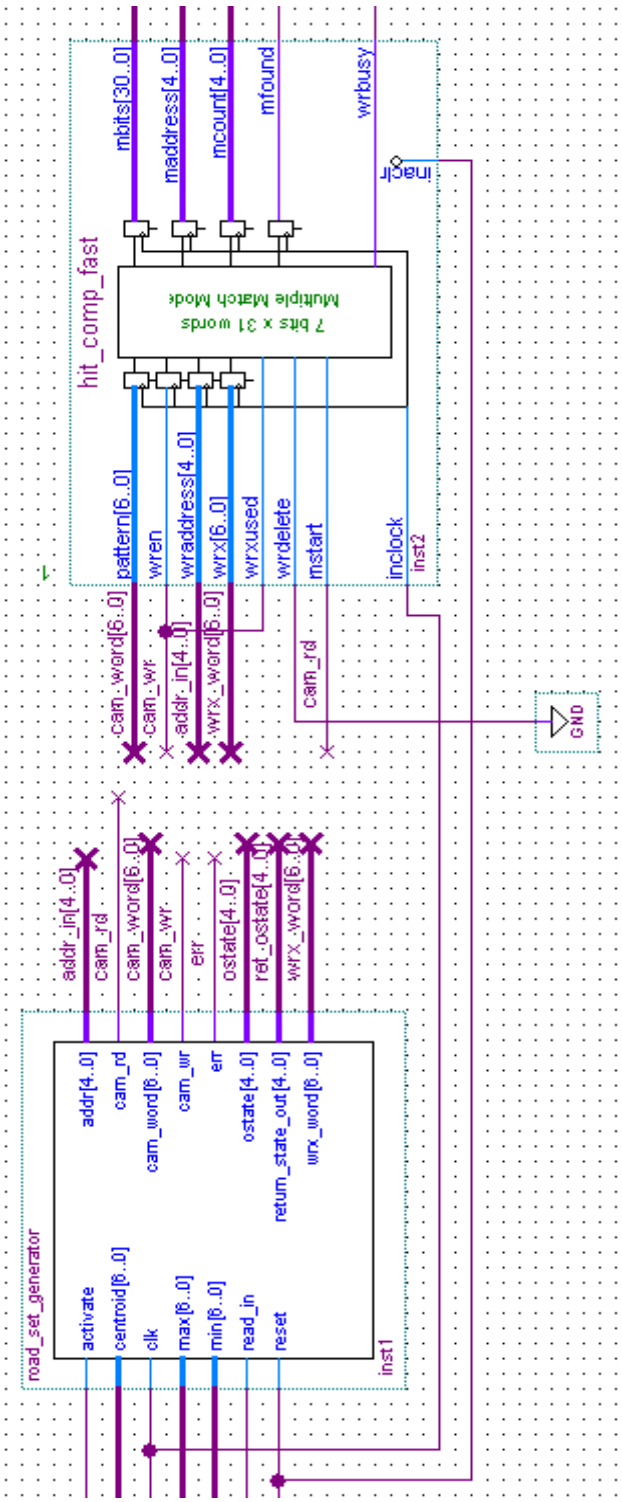
SCHEMATICS OF THE STC MODULES

Hit Filter Interface:



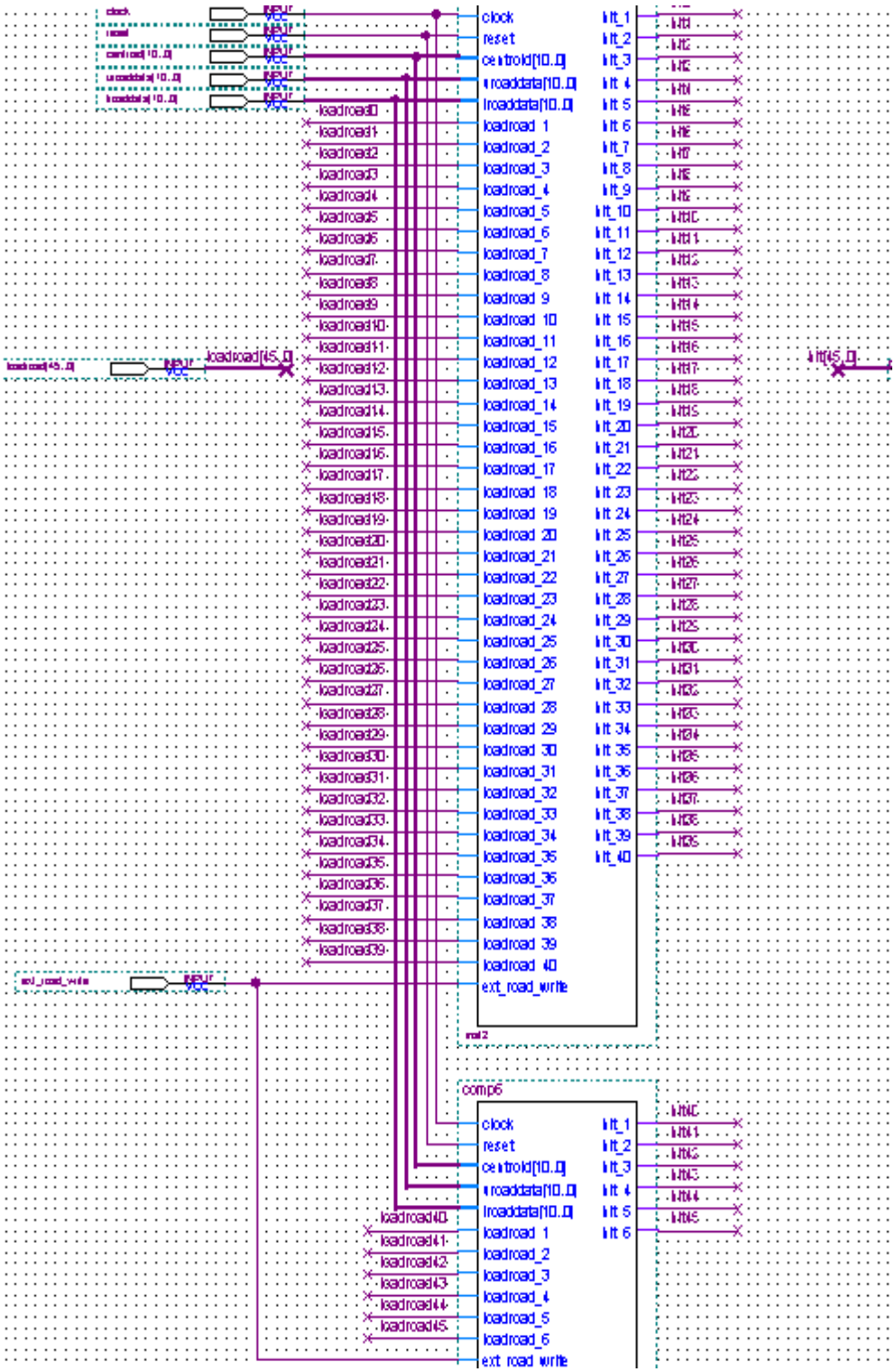
Hit-Filter Implemented with only a CAM

Road-set Generator and the CAM for a single road.

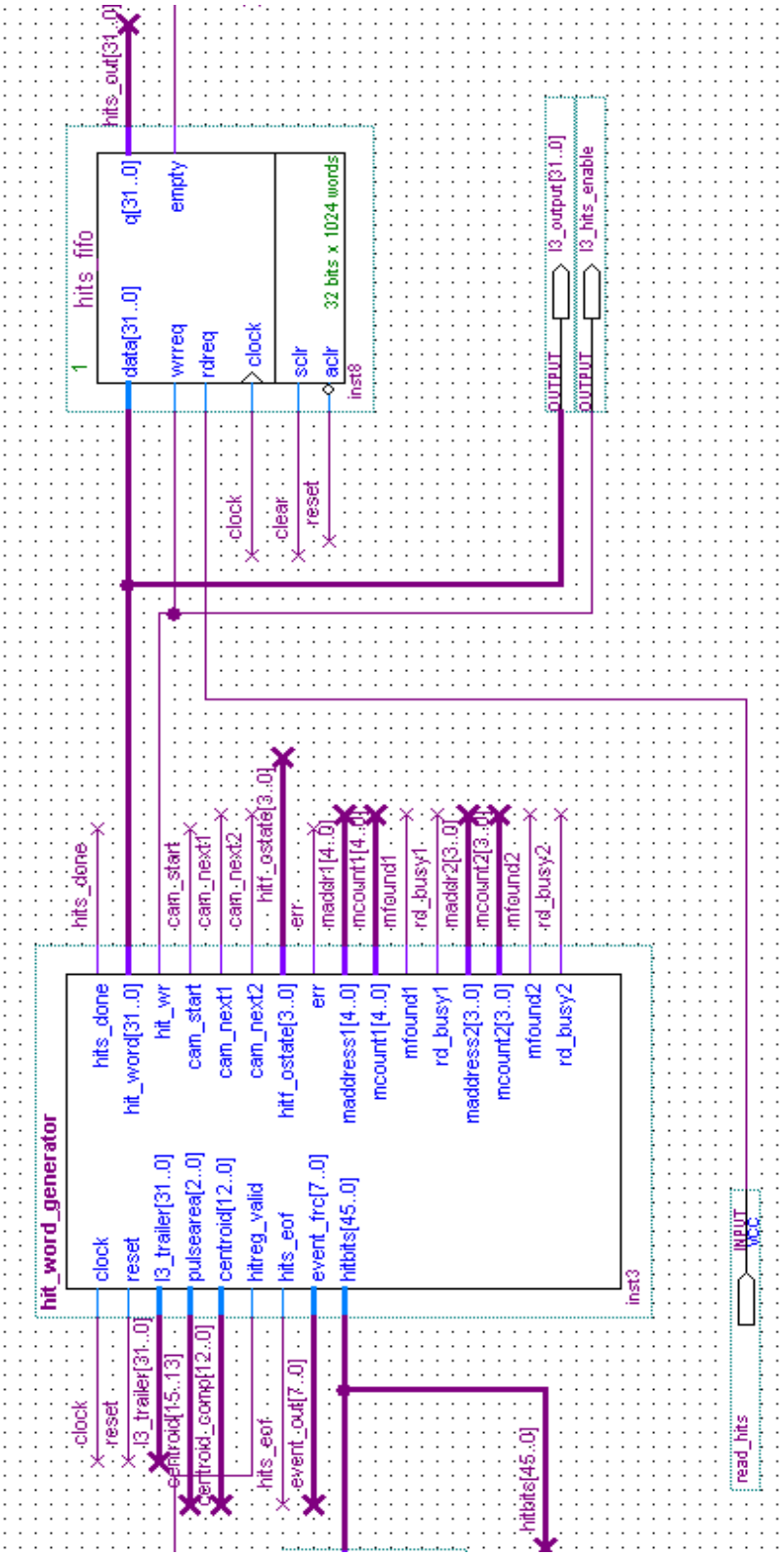


Hit-Filter Implemented with a comparator and Hit-Word Generator Block:

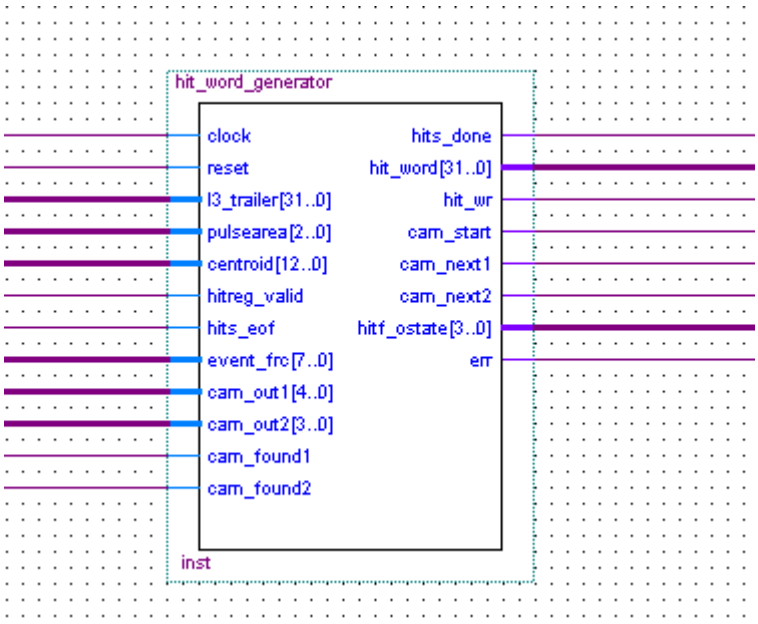
Comparator Module for 46 roads



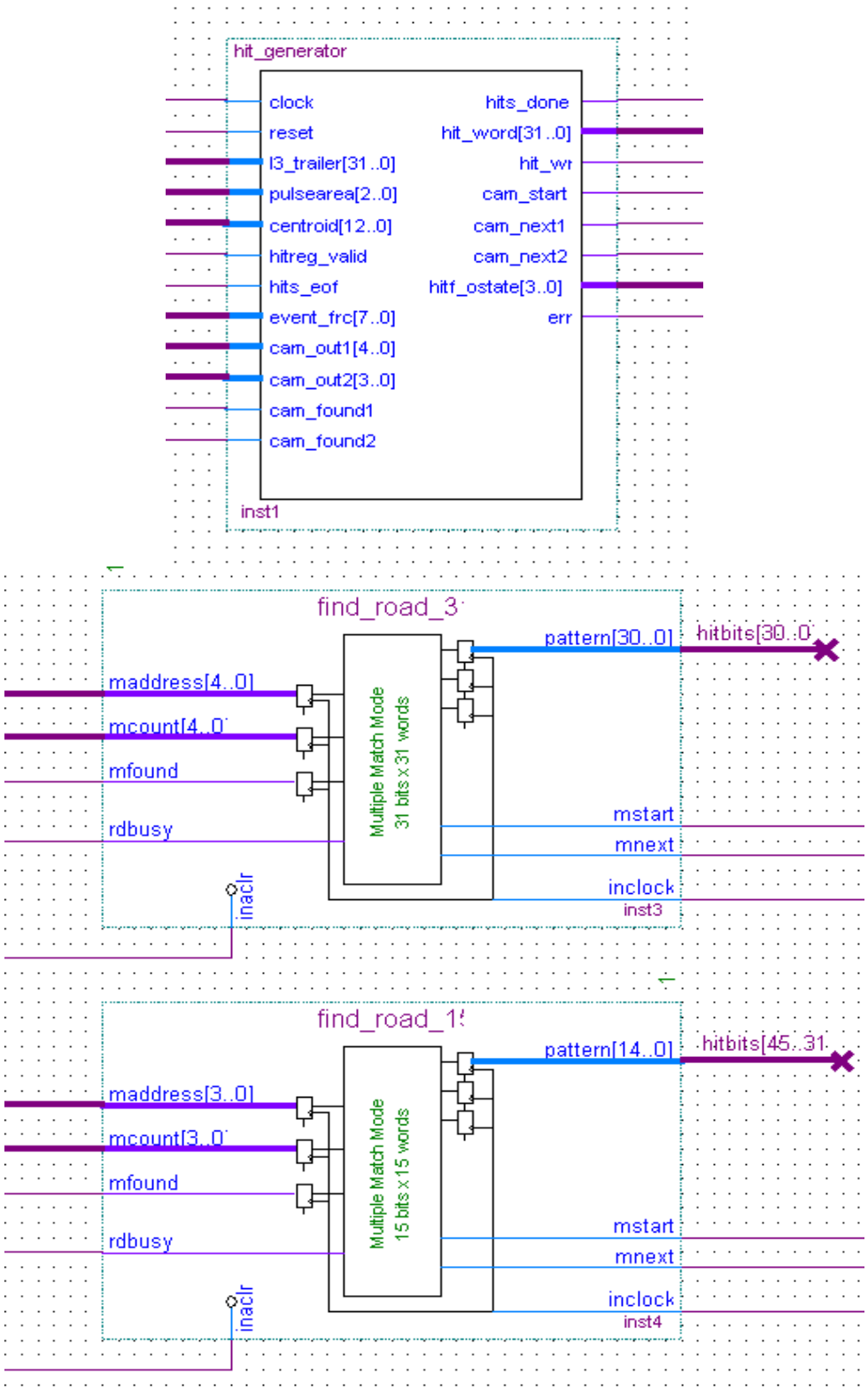
Hit Word Generator and the Hit-FIFO



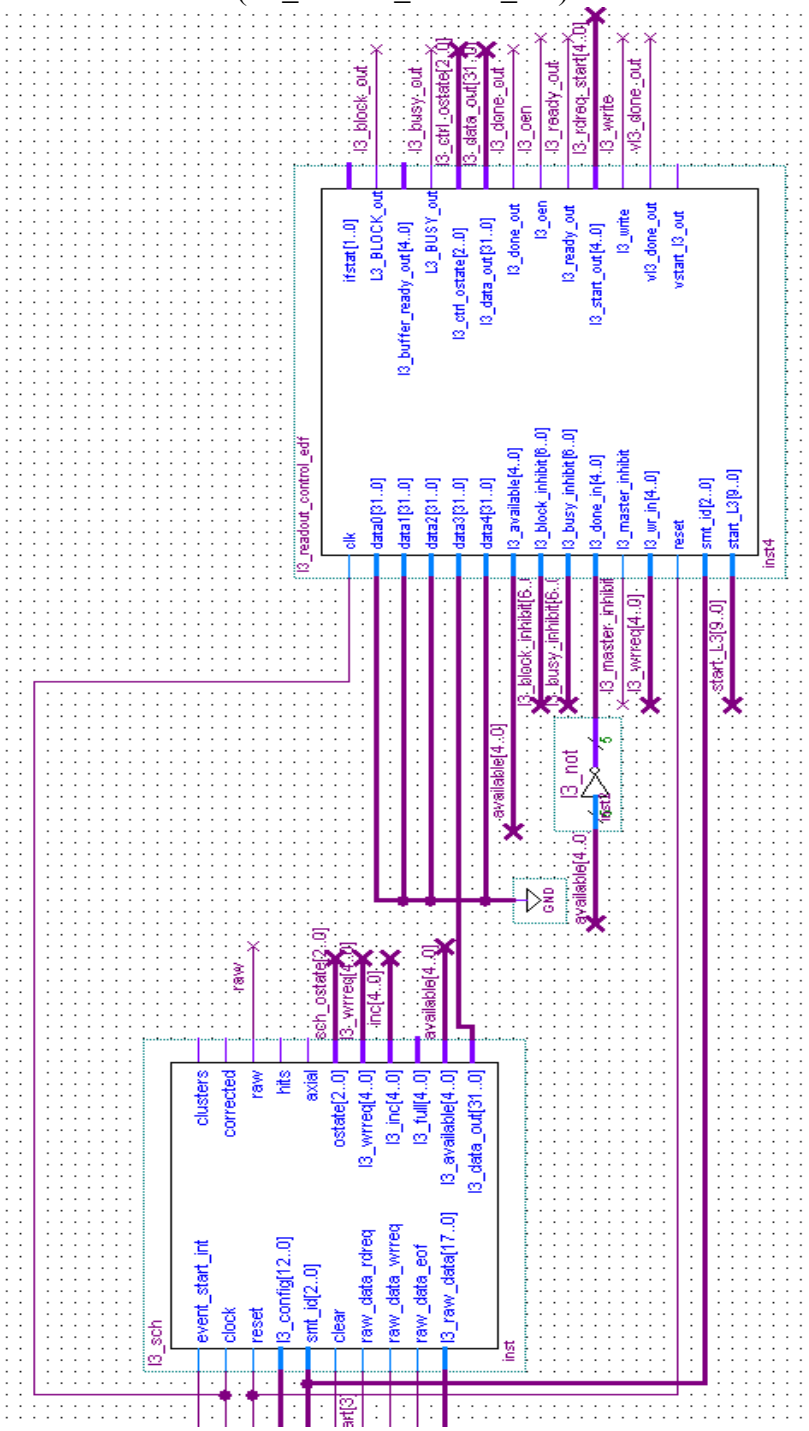
Hit-Word Generator Module



Hit-word Generator Module:
Hit_generator and the two CAMs



The L3 write control module (L3_sch) and L3 read control module (L3_readout_control_edf)



APPENDIX C

VHDL CODE OF THE STC MODULES

Hit-Filter Interface

```
-- Version 0 This block is used to interface to the Hit Filter
--------------------------------------------------------------------------------
-- Initial Design: Reginald Perry  (12/15/2000)
--------------------------------------------------------------------------------
--  Modified
-- 7/28/2001  Fix start_centroids
-- 7/30/2001 Arvindh Lalam    Making changes to get proper Data_valid waveform.
-- 6/10/2002 Arvindh Lalam    Forcing this module to check for bus availability before
--                                                 reading out each word.
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
------------------------------------------------------------------------------------------------
---
-- Entity Declaration : Defines Inputs and Outputs of the device.
------------------------------------------------------------------------------------------------
---
entity hit_filter_interface is

       port (clk,reset,hits_available,zvcs_available:in std_logic;
             end_of_hits,end_of_zvcs,end_of_l3_event: in std_logic;
             hits_in,centroids_in:in std_logic_vector(31 downto 0);
             start_hits,start_centroids: in std_logic;
             hc_inh: in std_logic_vector(6 downto 0);
             event_start_int: in std_logic;
             end_of_hit_event: in std_logic;
       --
       --       Hits busy is actually hits bus request
       --
             hits_read_req,hc_busy,hits_output_enablen: out std_logic;
             hc_data_out: out std_logic_vector(31 downto 0);
             hc_wr, hdone, cdone,dv,zvcs_read_req,event_busy: out std_logic;
             hit_filter_ostate: out std_logic_vector(3 downto 0)
        );

end entity hit_filter_interface;

------------------------------------------------------------------------------------------------
-----
--Architecture body
------------------------------------------------------------------------------------------------
------
architecture logic of hit_filter_interface is
  type mystates is
(sreset,swait_for_start,swait_for_hits,swait_for_centroids,swait_for_bus,
                    swrite_hits,swrite,sdummy_wait);
  signal ndv,pdv,nhdone,phdone,ncdone,pcdone: std_logic;
  signal nhc_busy,phc_busy: std_logic;
  signal pevent_busy, nevent_busy: std_logic;
  signal nhits_output_enablen, phits_output_enablen,nhwr,phwr: std_logic;
  signal bus_available,nctype,pctype: std_logic;
  signal ns,ps: mystates;
  constant hits_type: std_logic := '0';
  constant zvcs_type: std_logic := '1';
```

```
 begin

-------------------------------------------------------------------------------
-- This WITH SELECT is used to extract the current state of the Finite State Machine
-------------------------------------------------------------------------------
      with ps select
         hit_filter_ostate <= "0000" when sreset,
                              "0001" when swait_for_start,
                              "0010" when swait_for_hits,
                              "0011" when swait_for_centroids,
                              "0100" when swait_for_bus,
                              "0101" when swrite_hits,
                              "0110" when swrite,
                              "0111" when sdummy_wait,
                              "1111" when others;

-------------------------------------------------------------------------------
--   This process block ORs all the "inhibit"s to determine if the bus is
---  available
-------------------------------------------------------------------------------
    process(hc_inh)
       variable i: integer;
       variable temp: std_logic;
    begin
           temp := '0';
          for i in 0 to 6 loop
               temp := temp or hc_inh(i);
          end loop;
          bus_available <= not temp;
     end process;

-------------------------------------------------------------------------------
--  This0 process block controls the interface lines
-------------------------------------------------------------------------------

process(ps,start_hits,start_centroids,bus_available,pctype,zvcs_available,hits_available,

phdone,pcdone,pdv,phwr,phc_busy,phits_output_enablen,end_of_hits,end_of_zvcs)
      begin

       nhdone <= phdone;
       ncdone <= pcdone;
       ndv <= pdv;
       nhwr <= phwr;
       nhc_busy <= phc_busy;
       hits_read_req <= '0';
       zvcs_read_req <= '0';
       nhits_output_enablen <=phits_output_enablen;
       nctype <= pctype;

-----------------------------------------------------
--      This is the main FSM
-----------------------------------------------------
       case ps is
-----------------------------------------------------
--  SRESET : This state sets the correct values for all outputs after the initial reset
-----------------------------------------------------
          when sreset =>
             ns <= swait_for_start;
             nhdone <= '1';
             ncdone <= '1';
             ndv <= '1';
             nhwr <= '0';
             nhc_busy <= '0';
             nhits_output_enablen <= '1';
             nctype <= pctype;

-----------------------------------------------------
--      Wait for hits or centroids to be available
-----------------------------------------------------
          when swait_for_start =>
           ns <= swait_for_start;
           nhdone <= phdone;
           ncdone <= pcdone;
           ndv <= pdv;
           nhwr <= phwr;
           nhc_busy <= phc_busy;
           hits_read_req <= '0';
           zvcs_read_req <= '0';
```

```
           nhits_output_enablen <=phits_output_enablen;

-----------------------------------------------------
-- Looking for start_hits or start_centroids.  Assuming that can't occur
-- simulataneously
-----------------------------------------------------
            if(start_hits = '1') then
              nhdone <= '0';
              ncdone <= pcdone;
              ndv <= '0';
              ns <= swait_for_hits;
              nctype <= hits_type;

            elsif(start_centroids = '1') then
              nhdone <= phdone;
              ncdone <= '0';
              ndv <= '0';
              ns <= swait_for_centroids;
              nctype <= zvcs_type;

            else
              nhdone <= '1';
              ncdone <= '1';
              ndv <= '1';
              ns <= swait_for_start;
            end if;

-----------------------------------------------------
--  When Control logic wants to read hits,
--  This state Checks if hits are available
-----------------------------------------------------
          when swait_for_hits =>

           nhits_output_enablen <= '1';
           nhwr <= '0';
           nhc_busy <= '0';
           nhdone <= '0';
           ncdone <= pcdone;
           hits_read_req <= '0';
           zvcs_read_req <= '0';
           nctype <= hits_type;

-----------------------------------------------------
-- IF hits are available -THEN request the bus
--               ELSE wait here until we have hits
-----------------------------------------------------
          if(hits_available = '1') then

              nhc_busy <= '1';
              ns <= swait_for_bus;
            else
              nhc_busy <= '0';
              ns <= swait_for_hits;
            end if;

-----------------------------------------------------
--    Check if z centroids are available
-----------------------------------------------------
          when swait_for_centroids  =>

           nhits_output_enablen <= '1';
           nhwr <= '0';
           nhc_busy <= '0';
           ncdone <= '0';
           nhdone <= phdone;
           zvcs_read_req <= '0';
           hits_read_req <= '0';
           nctype <= zvcs_type;

-----------------------------------------------------
-- IF zvcs_centroids  are available THEN request the bus
--                    ELSE wait in this state
-----------------------------------------------------
            if(zvcs_available = '1') then
              nhc_busy <= '1';
              ns <= swait_for_bus;
            else
              nhc_busy <= '0';
              ns <= swait_for_centroids;
```

```
          end if;


-------------------------------------------------
--   We have hits or zvcs, now we are waiting for the bus
-------------------------------------------------
          when swait_for_bus=>
            ncdone <= pcdone;
            nhdone <= phdone;
            nhits_output_enablen <= '1';
            hits_read_req <= '0';
            zvcs_read_req <= '0';
            nhc_busy <= '1';    --  keep requesting the bus

-------------------------------------------------
--   IF the bus is available THEN upload hits ?
--             ELSE wait in this state
-------------------------------------------------
            if(bus_available = '1') then
              nhc_busy <= '1';
              ns <= swrite_hits;
            end if;


-------------------------------------------------
-- Reads hits from the internal HIT-FIFOs
-------------------------------------------------
            when swrite_hits =>
        ndv <= '1';           -- Setting Data_valid signal to indicate valid hit word.
                              -- Valid data word appears before end of this clock cycle.
             if(pctype = hits_type) then
               hits_read_req <= '1';
                              -- asynchronous, thus Data should appear on the next cycle
               zvcs_read_req <= '0';
             else
               hits_read_req <= '0';
               zvcs_read_req <= '1';  -- asynchronous, thus Data should appear on the
next cycle
             end if;
             nhits_output_enablen <= '0'; --  Turn on output bus.
             nhwr <= '0';
             ns <= sdummy_wait;

-------------------------------------------------
-- dummy state to allow data to appear on the bus
-------------------------------------------------
            when sdummy_wait =>
              ns <= swrite;
              hits_read_req <= '0';   -- bring read request low
              zvcs_read_req <= '0';
              nhits_output_enablen <= '0'; --   keep buffer on
              nhwr  <= '1';    -- the data will appear written into control logic during
next cycle

-------------------------------------------------
-- Data is available on the bus, and being written
-------------------------------------------------
            when swrite =>   --- write into control logic during this state
              hits_read_req <= '0';
              zvcs_read_req <= '0';
              nhits_output_enablen <= '0';
              nhwr <= '0';    --  turn off hc wr line
              nhc_busy <= '1'; --   keep bus

-------------------------------------------------
--                    Checking for more hits/Centroids
-------------------------------------------------
             case pctype is
              when hits_type =>

                if(hits_available = '1') then
                        hits_read_req <= '1';    --  bring read request line high
                        zvcs_read_req <= '0';
                        ns <=swait_for_bus;  -- go write the next word
                else

-------------------------------------------------
--             No more hits, Give up the bus
--                    Turn off output buffers and then release bus on next cycle
```

```
-------------------------------------------------
                      nhits_output_enablen <= '1';
                      nhc_busy <= '0';

-------------------------------------------------
--    If at the end of hits THEN, reset all values
--             ELSE wait for more hits
-------------------------------------------------
                     if(end_of_hits = '1') then
                      ns <= swait_for_start;
                      nhdone <= '1';
                      nhwr <= '0';
                      nhc_busy <= '0';
                      nhits_output_enablen <= '1';
                     else
                      ns <= swait_for_hits;
                      nhdone <= '0';
                      nhwr <= '0';
                      nhc_busy <= '0';
                      nhits_output_enablen <= '1';
                     end if;
            end if;

             when zvcs_type =>

               if(zvcs_available = '1') then
                       hits_read_req <= '0';    --  bring read request line high
                       zvcs_read_req <= '1';
                       ns <=sdummy_wait;  -- go write the next word
               else

-------------------------------------------------
--             No more Centroids, Give up the bus
--                    Turn off output buffers and then release bus on next cycle
-------------------------------------------------
                      nhits_output_enablen <= '1';
                      nhc_busy <= '0';

-------------------------------------------------
--    If at the end of centroids THEN, reset all values
--                    ELSE wait for more centroids
-------------------------------------------------
                     if(end_of_zvcs = '1') then
                      ns <= swait_for_start;
                      ncdone <= '1';
                      nhwr <= '0';
                      nhc_busy <= '0';
                      nhits_output_enablen <= '1';
                     else
                      ns <= swait_for_centroids;
                      ncdone <= '0';
       --              ndv <= '0';
                      nhwr <= '0';
                      nhc_busy <= '0';
                      nhits_output_enablen <= '1';

                     end if;
            end if;
            when others =>
              null;

           end case;  -- case pctype

          when others =>
             null;

          end case;   --- case ps

     end process;

-------------------------------------------------------------------------------------------
-
--  Register process block
-------------------------------------------------------------------------------------------
-
    process(clk,reset,ns, ndv,nhdone,ncdone,nhits_output_enablen,nhwr,nhc_busy,nctype)
     begin
       if(reset = '0') then
         ps <= sreset;
```

```
        pdv <= '0';
        phdone <= '0';
        pcdone <= '0';
        phits_output_enablen <= '0';
        phwr <= '0';
        pevent_busy <= '0';
        phc_busy <= '0';
        pctype <= '0';
      elsif(clk'event and clk = '1') then
        ps <= ns;
        pdv <= ndv;
        phdone <= nhdone;
        pcdone <= ncdone;
        phits_output_enablen <= nhits_output_enablen;
        phwr <= nhwr;
        pevent_busy <= nevent_busy;
        phc_busy <= nhc_busy;
        pctype <= nctype;
      end if;
    end process;

-------------------------------------------------------------------------------------------
-
--    This process block determines event_busy
--    It goes high when event_start_int goes high
--    It goes low with end of hits
-------------------------------------------------------------------------------------------
-
       process(event_start_int,end_of_hit_event,end_of_l3_event,pevent_busy)
         variable end_of_event: std_logic;

        begin

-----------------------------------------
--    Check for end_of_event, if we are currently in an event otherwise assume default
--     condition is we have completed the previous event.
-----------------------------------------
          end_of_event := end_of_hit_event and end_of_l3_event;
         if(pevent_busy = '1') then

-----------------------------------------
--   we are current in an event.  Have we finished?
--   Are we at end_of_event?
-----------------------------------------
              if(end_of_event = '1') then
                nevent_busy <= '0';     -- yes.Reset, event_busy
              else
                nevent_busy <= '1'; -- no. Keep event_busy high
              end if;
         else

-----------------------------------------
--                     ck for event start
-----------------------------------------
           if(event_start_int = '1') then
              nevent_busy <= '1';    --  yes, set event_busy
           else
              nevent_busy <= '0';     -- no, reset event busy
           end if;

         end if;
  end process;


-------------------------------------------------------------------------------------------
-
--    Use multiplexer to select between hits and centroids
-------------------------------------------------------------------------------------------
-
      with pctype select
        hc_data_out <=  hits_in when hits_type,
                        centroids_in when zvcs_type,
                        hits_in when others;

-- attach registers to outputs
   hits_output_enablen <= phits_output_enablen;
   dv <= pdv;
   hc_busy <= phc_busy;
   hdone <= phdone;
```

```
cdone <= pcdone;
hc_wr <= phwr;
event_busy <= pevent_busy;
end architecture;
```

Road-Set Generator：

```
 ------------------------------------------------------------------------------
--  File       : Road_Set_Generator.vhd
--  Created on 01/22/02 by Arvindh Lalam.
--     Modified on 01/26/02 by Arvindh Lalam.
--                    Adding estimation of 'words with don't cares' from UPPER ROAD
------------------------------------------------------------------------------
--     Author  : Vindi Lalam
--     1. Based on algorithm discussed in the Thesis work.
--     2. Generates road-sets for given roads and stores them in a CAM
--  3. Data words belonging to the road-set may contain 'DON'T CARES'
--     4. It takes a maximum of 50 Cycles to generate coded words for a given road.
------------------------------------------------------------------------
--     clk                            IN      : Clock
--     reset                  IN      : Reset
--     min[6..0]              IN      : Lower road
--     max[6..0]              IN      : Upper road
--     activate               IN      : Activates this module
--     read_in                IN      : Read signal for checking if data exits in CAM
--     centroid[6..0] IN      : centroid calculated by Centroid finder.

--
--     cam_wr                 OUT     : Cam write signal
--     cam_rd                 OUT     : Latched Read signal (required to sync with
centroid)
--     cam_word[6..0] OUT     : output word
--     wrx_word[6..0] OUT     : wrx for don't care.
--     adrr[4..0]             OUT     : CAM address.
--     ostate[4..0]   OUT     : FSM state.
--     err                            OUT     : Indicates err in 'CASE STATEMENT'
--     return_state_out[4..0] OUT     : Brings out internal 'return_state'
------------------------------------------------------------------------
--
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

------------------------------------------------------------------------------
-- Device ports
------------------------------------------------------------------------------
entity road_set_generator is
port    (       clk             : in std_logic;
                        reset   : in std_logic;
                        min             : in unsigned(6 downto 0);
                        max             : in unsigned(6 downto 0);
                        activate        : in std_logic;
                        read_in         : in std_logic;
                        centroid        : in unsigned(6 downto 0);

                        cam_wr          : out std_logic;
                        cam_rd          : out std_logic;
                        cam_word        : out unsigned(6 downto 0);
                        wrx_word        : out unsigned(6 downto 0);
                        addr            : out unsigned(4 downto 0);
                        ostate          : out std_logic_vector(4 downto 0);
                        return_state_out        : out std_logic_vector(4 downto 0);
                        err                     : out std_logic);
end road_set_generator;

------------------------------------------------------------------------------
-- Architecture Body Begins
------------------------------------------------------------------------------
architecture behaviour of road_set_generator is
        type mystates is (sreset, sinitialize, sbit0, sbit1, sbit2, sbit3, sbit4,
                                                sbit5, sbit6, sbit0_rev, sbit1_rev,
sbit2_rev, sbit3_rev,
                                                sbit4_rev, sbit5_rev, sbit6_rev, sread,
swrite, sdummy1,
                                                sdecide);
        signal ns, ps, nreturn_state, preturn_state  : mystates;
        signal ncam_wr, pcam_wr, ncam_rd, pcam_rd    : std_logic;
        signal ncam_word, pcam_word, nmin, pmin, nmax, pmax, nwrx_word,
                        pwrx_word : unsigned (6 downto 0);
        signal ntemp, ptemp             : unsigned(6 downto 0);
        signal nstatus,pstatus : unsigned(6 downto 0);
```

```
        signal naddr, paddr           : unsigned(4 downto 0);
        constant zero7: unsigned(6 downto 0) := "0000000";
        constant zero8: unsigned(7 downto 0) := "00000000";
begin

-------------------------------------------------------------------------------
-- Starting the Process
-------------------------------------------------------------------------------
        process (pmin, pmax, pcam_word, pwrx_word, activate, ntemp, ptemp, pcam_wr,
                        pcam_rd, clk, pstatus, ps, max, min, preturn_state, paddr, read_in,
                        centroid)
                variable diff          :unsigned(7 downto 0);
                variable diff_xor      :unsigned(6 downto 0);
                variable final_state, final_state_rev:mystates;
                variable i                      : integer;
        begin

-------------------------------------
-- Initialization in the FSM
-------------------------------------
        ns                      <= ps;
        ncam_word       <= pcam_word;
        nmin            <= pmin;
        nmax            <= pmax;
        nwrx_word       <= pwrx_word;
        ncam_wr         <= pcam_wr;
        ncam_rd         <= pcam_rd;
        nstatus         <= pstatus;
        nreturn_state <= preturn_state;
        ntemp           <= ptemp;
        naddr           <= paddr;
        err <= '0';

        diff(7) := '0';

-------------------------------------
-- Finds the range of words. diff_xor <= MAX (xor) MIN
-- starting at bit 0, all bits after the first occurrence of 1 are made '1'

-- 0010101 <= 0010010 (xor) 0000111    ; 0011111 ;
-------------------------------------
        for i in 0 to 6 loop
                diff_xor(6-i) := min(6-i) xor max(6-i);
                diff(6-i) := diff(7-i) or diff_xor(6-i);
        end loop;

-------------------------------------
-- Find the highest-bit and thus the last state to go into:
-- a value of 0011111 indicates final_state = sbit4 AND
--                                                         final_state_rev = sbit4_rev
-------------------------------------
        case pstatus is
        when "0000001" =>
                final_state := sbit0;
                final_state_rev := sbit0_rev;
        when "0000011" =>
                final_state := sbit1;
                final_state_rev := sbit1_rev;
        when "0000111" =>
                final_state := sbit2;
                final_state_rev := sbit2_rev;
        when "0001111" =>
                final_state := sbit3;
                final_state_rev := sbit3_rev;
        when "0011111" =>
                final_state := sbit4;
                final_state_rev := sbit4_rev;
        when "0111111" =>
                final_state := sbit5;
                final_state_rev := sbit5_rev;
        when "1111111" =>
                final_state := sbit6;
                final_state_rev := sbit6_rev;
        when others    =>
                final_state := sbit6;
                final_state_rev := sbit6_rev;
        end case;

-------------------------------------
```

```
-- Redirect :  If ps = final_state,
--                             THEN start from UPPER_ADDRESS.
--                             ELSE continue
-------------------------------------
       if(ps = final_state) then
              ns <= sbit0_rev;
              ntemp <= pmax;
              naddr <= "00110";
       else
              case ps is

-------------------------------------
       -- RESET State: Initializes all the values
-------------------------------------
              when sreset =>
                     nmin <= zero7;
                     nmax <= zero7;
                     ntemp <= zero7;
                     ncam_word <= zero7;
                     nwrx_word <= zero7;
                     ncam_wr <= '0';
                     ncam_rd <= '0';
                     ns       <= sinitialize;
                     nstatus <= zero7;
                     err <= '0';
                     nreturn_state <= sreset;
                     naddr   <= "00000";

-------------------------------------
       -- SINITIALIZE : Stays in this state until activate = '1',
       --                                                  or read_in = '1'.
-------------------------------------
              when sinitialize =>
                     if (activate='1') then               -- Goto state SBIT0
                            nmin <= min;
                            ntemp <= min;
                            nmax <= max;
                            ncam_word      <= zero7;
                            nwrx_word      <= zero7;
                            ns <= sbit0;
                            ncam_wr <= '0';
                            ncam_rd <= '0';
                            nstatus <= diff(6 downto 0);
                     else
                            nmin <= pmin;
                            nmax <= pmax;
                            ntemp <= ptemp;
                            nwrx_word      <= pwrx_word;
                            ns <= sinitialize;
                            ncam_wr <= pcam_wr;
                            nstatus <= pstatus;
                            if(read_in='1') then          -- Goto state SREAD
                                   ncam_word <= centroid;
                                   ns <= sread;
                                   ncam_rd <= '1';
                            else                    -- Wait for change in
'activate'/'read_in'
                                   ncam_word <= pcam_word;
                                   ns <= sinitialize;
                                   ncam_rd <= '0';
                            end if;
                     end if;

-------------------------------------
       -- Estimate CAM words starting from LOWER_ADDRESS,
       -- to represent the given range DONT CARES are used to represent ranges.
-------------------------------------
       -- BIT0 : Checks bit'0'.
-------------------------------------
              when sbit0 =>
                     ns <= swrite;
                     nreturn_state <= sbit1;
                     if (ptemp(0) = '1') then
                            ntemp <= ptemp + "0000001";
                            ncam_word <= ptemp;
                            nwrx_word <= zero7;
                            ncam_wr <= '1';
                     else
                            ns <= sbit1;
```

```
                        end if;

-------------------------------------
        -- BIT1        : Checks bit 1.
-------------------------------------
                when sbit1 =>
                        ns <= swrite;
                        nreturn_state <= sbit2;
                        if (ptemp(1) = '1') then
                                ntemp <= ptemp + "0000010";
                                ncam_word <= ptemp;
                                nwrx_word <= "0000001";
                                ncam_wr <= '1';
                        else
                                ns <= sbit2;
                        end if;

-------------------------------------
        -- BIT2        : Check bit2
-------------------------------------
                when sbit2 =>
                        ns <= swrite;
                        nreturn_state <= sbit3;
                        if (ptemp(2) = '1') then
                                ntemp <= ptemp + "0000100";
                                ncam_word <= ptemp;
                                nwrx_word <= "0000011";
                                ncam_wr <= '1';
                        else
                                ns <= sbit3;
                        end if;

-------------------------------------
        -- BIT3 : Check bit3
-------------------------------------
                when sbit3 =>
                        ns <= swrite;
                        nreturn_state <= sbit4;
                        if (ptemp(3) = '1') then
                                ntemp <= ptemp + "0001000";
                                ncam_word <= ptemp;
                                nwrx_word <= "0000111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit4;
                        end if;

-------------------------------------
        -- BIT4        : Check bit4
-------------------------------------
                when sbit4 =>
                        ns <= swrite;
                        nreturn_state <= sbit5;
                        if (ptemp(4) = '1') then
                                ntemp <= ptemp + "0010000";
                                ncam_word <= ptemp;
                                nwrx_word <= "0001111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit5;
                        end if;


-------------------------------------
        -- BIT5        : Check bit5
--      This state behaves differently from others SBIT_ states.:
--      Going through next state(sbit6) is redundant, thus..
--      Redirect to SBIT0_REV; Reinitialize PTEMP to UPPER_ADDRESS (pmax).
-------------------------------------
                when sbit5 =>
                        ns <= swrite;
                        nreturn_state <= sbit0_rev;
                        ntemp <= pmax;
                        if (ptemp(5) = '1') then
                                ncam_word <= ptemp;
                                nwrx_word <= "0011111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit0_rev;
```

```
                        end if;

-------------------------------------
        -- SBIT0_REV   : Start from Bit'0'.
-------------------------------------
                when sbit0_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit1_rev;
                        ncam_word <= ptemp;
                        if (ptemp(0) = '0') then
                                ntemp <= ptemp - "0000001";
                                nwrx_word <= zero7;
                                ncam_wr <= '1';
                        else
                                ns <= sbit1_rev;
                        end if;


-------------------------------------
                        -- In the last state, then check if bit ='1'
-------------------------------------
                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(0) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0000001";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;

-------------------------------------
        -- SBIT1_REV   : Start from Bit'1'.
-------------------------------------
                when sbit1_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit2_rev;
                        ncam_word <= ptemp;
                        if (ptemp(1) = '0') then
                                ntemp <= ptemp - "0000010";
                                nwrx_word <= "0000001";
                                ncam_wr <= '1';
                        else
                                ns <= sbit2_rev;
                        end if;

                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(1) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0000011";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;

-------------------------------------
        -- SBIT2_REV   : Start from Bit'2'.
-------------------------------------
                when sbit2_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit3_rev;
                        ncam_word <= ptemp;
                        if (ptemp(2) = '0') then
                                ntemp <= ptemp - "0000100";
                                nwrx_word <= "0000011";
                                ncam_wr <= '1';
                        else
                                ns <= sbit3_rev;
                        end if;
```

```
                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(2) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0000111";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;

-------------------------------------
        -- SBIT3_REV   : Start from Bit'3'.
-------------------------------------
                when sbit3_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit4_rev;
                        ncam_word <= ptemp;
                        if (ptemp(3) = '0') then
                                ntemp <= ptemp - "0001000";
                                nwrx_word <= "0000111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit4_rev;
                        end if;

                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(3) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0001111";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;


-------------------------------------
        -- SBIT4_REV   : Start from Bit'4'.
-------------------------------------
                when sbit4_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit5_rev;
                        ncam_word <= ptemp;
                        if (ptemp(4) = '0') then
                                ntemp <= ptemp - "0010000";
                                nwrx_word <= "0001111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit5_rev;
                        end if;

                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(4) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0011111";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;

-------------------------------------
        -- SBIT5_REV   : Start from Bit'5'.
-------------------------------------
                when sbit5_rev =>
                        ns <= swrite;
                        nreturn_state <= sbit6_rev;
```

```
                        ncam_word <= ptemp;
                        if (ptemp(5) = '0') then
                                ntemp <= ptemp - "0100000";
                                nwrx_word <= "0011111";
                                ncam_wr <= '1';
                        else
                                ns <= sbit6_rev;
                        end if;

                        if (ps = final_state_rev) then
                                nreturn_state <= sreset;
                                if(ptemp(5) = '1') then
                                        ntemp <= ptemp;
                                        nwrx_word <= "0111111";
                                        ncam_wr <= '1';
                                        ns <= swrite;
                                else
                                        ntemp <= ptemp;
                                        ncam_wr <= '0';
                                        ns <= sdecide;
                                end if;
                        end if;

-------------------------------------
        -- SBIT6_REV   : Start from Bit'6'
        -- If the FSM comes to this state, it may mean that either all
        --             the bits are DON'T CARES (if this bit = '1')
-------------------------------------
                when sbit6_rev =>
                        nreturn_state <= sreset;
                        ncam_word <= ptemp;
                        if(ptemp(6) = '1') then
                                ntemp <= ptemp;
                                nwrx_word <= "1111111";
                                ncam_wr <= '1';
                                ns <= swrite;
                        else
                                ntemp <= ptemp;
                                ncam_wr <= '0';
                                ns <= sdecide;
                        end if;

-------------------------------------
        -- SREAD       : Start from Bit'0'.
-------------------------------------
                when sread =>
                        ns <= sdummy1;
                        ncam_rd <= '0';
                        nreturn_state <= preturn_state;

-------------------------------------
        -- SWRITE      : Store the ROAD-SET word.
-------------------------------------
                when swrite    =>
                        ns <= sdummy1;
                        nreturn_state <= preturn_state;

                when sdummy1 =>
                        ns <= sdecide;
                        nreturn_state <= preturn_state;

-------------------------------------
        -- SDECIDE     : find if this is the first or the second of the
        ---                    two road-sets that can be stored in each ESB
-------------------------------------
                when sdecide =>
                        if (preturn_state = sreset and paddr(4)='0') then
                                ns <= sinitialize;
                                naddr <= "10000";
                        else
                                ns <= preturn_state;
                                naddr <= paddr + 1;
                        end if;
                        ncam_wr <= '0';

                when others=>
                        err <= '1';
                        ns <= sreset;
                end case;
```

```
        end if;
end process;


-------------------------------------------------------------------
-- Register Block
-------------------------------------------------------------------

process (clk, reset, nmin, nmax, ntemp, ncam_word, ncam_wr, ncam_rd,
                ns, nreturn_state, naddr)
begin
        if (reset='0') then
                pmin <= zero7;
                pmax <= zero7;
                ptemp <= zero7;
                pcam_word <= zero7;
                pwrx_word <= zero7;
                pcam_wr <= '0';
                pcam_rd <= '0';
                ps       <= sreset;
                pstatus <= zero7;
                preturn_state <= sreset;
                paddr   <= "00000";
        elsif(clk='1' and clk'event) then
                pmin <= nmin;
                pmax <= nmax;
                ptemp <= ntemp;
                pcam_word      <= ncam_word;
                pwrx_word      <= nwrx_word;
                pcam_wr <= ncam_wr;
                pcam_rd <= ncam_rd;
                ps <= ns;
                pstatus <= nstatus;
                preturn_state <= nreturn_state;
                paddr   <= naddr;
        end if;
end process;

-------------------------------------
-- Connect Output ports to the register outuputs.
-------------------------------------
cam_wr <= pcam_wr;
cam_rd <= pcam_rd;
cam_word <= pcam_word;
wrx_word <= pwrx_word;
addr    <= paddr;

-------------------------------------
-- The encoded state for debugging purposes
-------------------------------------
with ps select
ostate <=       "00000" when sreset,
                        "00001" when sinitialize,
                        "00010" when sbit0,
                        "00011" when sbit1,
                        "00100" when sbit2,
                        "00101" when sbit3,
                        "00110" when sbit4,
                        "00111" when sbit5,
                        "01000" when sbit6,
                        "01001" when sbit0_rev,
                        "01010" when sbit1_rev,
                        "01011" when sbit2_rev,
                        "01100" when sbit3_rev,
                        "01101" when sbit4_rev,
                        "01110" when sbit5_rev,
                        "01111" when sbit6_rev,
                        "10000" when sread,
                        "10001" when swrite,
                        "10010" when sdummy1,
                        "10011" when sdecide,
                        "11111" when others;

-------------------------------------
-- The encoded "return state" for debugging purposes
-------------------------------------
with preturn_state select
return_state_out <="00000" when sreset,
                        "00001" when sinitialize,
```

```
                        "00010" when sbit0,
                        "00011" when sbit1,
                        "00100" when sbit2,
                        "00101" when sbit3,
                        "00110" when sbit4,
                        "00111" when sbit5,
                        "01000" when sbit6,
                        "01001" when sbit0_rev,
                        "01010" when sbit1_rev,
                        "01011" when sbit2_rev,
                        "01100" when sbit3_rev,
                        "01101" when sbit4_rev,
                        "01110" when sbit5_rev,
                        "01111" when sbit6_rev,
                        "10000" when swrite,
                        "10001" when sdummy1,
                        "10010" when sdecide,
                        "11111" when others;
end behaviour;
```

Comparator Block

```
--------------------------------------------------------------------------
--File:comparator.vhd
-- To compare the strip pointer with the data value from the
-- road data
-- Originally created by : Shweta Lolage
-- 05/15/2000 Shweta Lolage on  : Changes made according to the new approach
-- 7/7/2001 rjp   Add road_select signal to allow road_write signal to
--                                  directly load comparators
-- 7/31/2001 Arvindh Lalam: Using loadroad instead of road_write
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------------
--  Entity Declaration
-------------------------------------------------------------------------------
       entity comparator is
       port (clock,reset : in std_logic;
                centroid : in unsigned (10 downto 0);  -- the centroid
                uroaddata : in unsigned (10 downto 0); -- the upper road_data address
                lroaddata : in unsigned (10 downto 0); -- the lower road)data address
                road_select: in std_logic;  --  This signal is used to select the
comparator.
                loadroad: in std_logic;  -- This signal will load the current road data
                                                                    -- into the
comparator, if selected
                hit : out std_logic      -- Used to return the comparator output
               );
       end entity comparator;

-------------------------------------------------------------------------------
--  Architecture Body
-------------------------------------------------------------------------------
       architecture behavior of comparator is
-- internal signals for the process blocks
       signal ndata,pdata, compare1, compare2 : std_logic;
       signal nroaddatau,proaddatau,nroaddatal,proaddatal : unsigned (10 downto 0);
       signal  write_enable : std_logic;
       constant zero11 : unsigned ( 10 downto 0) := "00000000000";
       begin

-------------------------------------------
-- this block is used for the comparison of the road-data with the given centroid
-------------------------------------------
       process(proaddatau,proaddatal,centroid)
       begin
              nroaddatau<= proaddatau;
              nroaddatal<= proaddatal;
              compare1 <= '0';
              compare2 <= '0';
        if (proaddatau >= centroid ) then
                     compare1 <= '1';
              else
                     compare1 <= '0';
              end if;
              if (proaddatal <= centroid) then
                     compare2 <= '1';
              else
                     compare2 <= '0';
              end if;
       end process;

-------------------------------------------
-- this block gives the result of the comparison
-------------------------------------------
       process (compare1,compare2)
       begin
--  output is logical and of compare1 and compare2
              ndata <= compare1 and compare2;
       end process;

-------------------------------------------
-- the process block to register the road-data signals
-------------------------------------------
       process(ndata,clock,reset)
```

```
        begin
                if ( reset ='0') then
                        pdata <= '0';
                elsif ( clock'event and clock='1') then
                        pdata <= ndata;
                end if;
        end process;

-------------------------------------------
-- the process block to register the road-data signals
-------------------------------------------
        process( nroaddatau,nroaddatal,reset,road_select,loadroad,lroaddata,uroaddata)
        begin
                write_enable <= loadroad and road_select;
                if ( reset ='0') then
                        proaddatau <= zero11;
                        proaddatal <= zero11;
                elsif (clock'event and clock='1') then
                        if(write_enable = '0') then
                        proaddatau <= nroaddatau;
                        proaddatal <= nroaddatal;
                        else
                        proaddatau <= uroaddata;
                        proaddatal <= lroaddata;
                end if;
                end if;
        end process;

-------------------------------------------
-- final output from the comparator
-------------------------------------------
        hit <= pdata;
end architecture behavior;
```

Hit-Word generator

```
--      File    : hit_generator.vhd
--  Author      : Arvindh Lalam.
--  Dated       : March 06, 2002.
-------------------------------------------------------------------
--      This module works in conjunction with two CAM blocks. CAM1 (31
--              locations) and CAM2 (15 locations) are used to store information
--              of all the 46 roads.
--
--      This module adds 31 to the address location of CAM2 to find the actual
--              ROAD number.
--
--      ** Using async signals for DONE (HITS_DONE & VDONE), START, NEXT1, NEXT2.
-------------------------------------------------------------------
--  1. Encodes the "road number" that the centroid falls in.
--      2. Adds other information to generate the complete HIT to be stored.
--      3. Controls two external CAMs.
-------------------------------------------------------------------
--  Modified:
--  03/12/02 (Vindi)   : Completed debugging, transfering to Quartus.
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

-------------------------------------------------------------------------------------
-- The Device PORTS
-------------------------------------------------------------------------------------
entity hit_generator is
PORT (
--** INPUTS
                clock                                   : in std_logic;
                reset                                   : in std_logic;
                l3_trailer                              : in unsigned(31 downto 0);
                 -- Contains EVENT_ID ;
                 -- Received from strip_reader_control_edf_final thru Monitor blcok.
                pulsearea                               : in unsigned(2 downto 0);
                centroid                                : in unsigned(12 downto 0);
                 -- Above two values are read from Centroid RAM of Centroid finder block
                hitreg_valid                    : in std_logic;
                 -- Implies new centroid at input ; Generated by Hit_control_block ;
                hits_eof                                : in std_logic;
                 -- Momentary pulse indicating END OF EVENT.
                 -- This is found one clock period after DONE goes low.
                event_frc                               : in unsigned(7 downto 0);
                 -- Event latched at FRC_START
                cam_out1                                : in unsigned(4 downto 0);
                cam_out2                                : in unsigned(3 downto 0);
                 -- Encoded CAM output and count of number of matches.
                cam_found1, cam_found2 : in std_logic;

                 -- Goes HIGH if a match is found.

-- ** OUTPUTS
                hits_done       : out std_logic;
                 -- Pulse of HITS_DONE indiacates that hits for a centroids are stored;
                                --      Next centroid can be read
                hit_word                                : out unsigned(31 downto 0);
                hit_wr                                  : out std_logic;
                 -- Hit word and the async write signal
                cam_start                               : out std_logic;
                cam_next1,cam_next2             : out std_logic;
                 -- CAM control signals.

-- ** DEBUG SIGNALS
                hitf_ostate                             : out std_logic_vector(3 downto 0);
                err                                             : out std_logic
        );
end hit_generator;

-------------------------------------------------------------------------------------
-- Architecture Body
-------------------------------------------------------------------------------------
architecture behaviour of hit_generator is
        type mystates is (sreset, sinitialize, scam_start1, sdummy1, scam_next1,
sstore_hits1,
```

```
                                        sstore_hits2, sdummy2, scheck_eof,
strailer_write);
        signal nstate, pstate                : mystates;
        signal nhit_word, phit_word          : unsigned(31 downto 0);
        signal nhit_wr, phit_wr                     : std_logic;
        signal nhits, phits                         : unsigned(7 downto 0);
        constant zero5 : unsigned(4 downto 0) := "00000";
        constant zero6  : unsigned(5 downto 0) := "000000";
        constant zero8 : unsigned(7 downto 0) := "00000000";
        constant zero32 : unsigned(31 downto 0) := "00000000000000000000000000000000";

        constant trailer_head  : unsigned(4 downto 0) := "11110";
BEGIN

-----------------------------------------
-- The Process starts
-----------------------------------------
read_cam : process
(clock,reset,l3_trailer,pulsearea,centroid,hitreg_valid,hits_eof,event_frc,
                                    cam_out1,cam_out2,cam_found1, cam_found2, pstate,
phit_wr,phit_word,
                                    event_frc, phits)
                        variable vstart, vnext1, vnext2, vdone       : std_logic;
                          -- signals to control various CAMs.
                        variable veerr                                        :
std_logic;
                          -- Mismatch of "EVENT number" between FRC and SMT.
                        variable vevent, vevent_frc                  : unsigned(7 downto
0);
                          -- SMT and FRC event IDs
                        variable verr_bits                                    :
unsigned(3 downto 0);
                          -- SEERR, MM, RERR,EERR
                        variable vroad, vcam_temp1, vcam_temp2       : unsigned(5 downto
0);
                          -- temporary CAM address variables
                BEGIN

---------------------------------------
-- Variable Initializations
---------------------------------------
                vstart  := '0';
                vnext1  := '0';
                vnext2  := '0';
                vdone   := '0';
                vcam_temp1 := '0' & cam_out1;
                vcam_temp2 := "00" & cam_out2;
                nhit_wr        <= '0';
                nhit_word      <= phit_word;
                nhits          <= phits;
                vevent         := l3_trailer(7 downto 0);
                vevent_frc     := event_frc;
                veerr          := '0';
        --      Compares EVENT_IDs received from SMT & FRC and sets EERR in case of an
error.
                if(vevent = vevent_frc) then
                        veerr := '0';
                else
                        veerr := '1';
                end if;
        --      ERROR bits                      SERR + MM                       + RERR+ EERR
                verr_bits      := l3_trailer(26 downto 25) & '0' & veerr;

                case pstate is
                -- RESET state: Resets all the signals and waits for "GO" signal from
HIT_CONTROL_BLOCK.
                when sreset    =>
                        vstart  := '0';
                        vnext1  := '0';
                        vnext2  := '0';
                        vdone   := '0';
                        vroad   := zero6;
                        vcam_temp1 := zero6;
                        vcam_temp2 := zero6;
                        nhit_wr <= '0';
                        err             <= '0';
                        nhit_word <= zero32;
                        if(hitreg_valid = '1') then
                                nstate <= scam_start1;
```

```
                        else
                                nstate <= sreset;
                        end if;

---------------------------------------
        -- Initiates the CAM1
---------------------------------------
                when scam_start1 =>
                        vstart := '1';
                        nstate <= sdummy1;

---------------------------------------
                -- Waits for CAM iniatialization.
---------------------------------------
                when sdummy1 =>
                        vstart := '0';
                        nstate <= scam_next1;

---------------------------------------
        -- Activates reading of multiple CAM locations.
---------------------------------------
                when scam_next1         =>
                        vnext1 := '1';
                        nstate <= sstore_hits1;

---------------------------------------
                -- Reads CAM1 address and writes HIT word.
                -- If there are no matching contents, FSM goes to SSTORE_HITS2.
---------------------------------------
                when sstore_hits1 =>
                        vnext1 := '1';
                        vnext2 := '0';
                        vroad := vcam_temp1;
--                      6               3               <------------- [5              +
        2]      + 3      -------------->       13
--                      road            area            SEQ     (bit3 of 1-8 is left out)
        HDI                             centroid
nhit_word <= vroad & pulsearea & l3_trailer(18 downto 14) & l3_trailer(12 downto 8) &
centroid;
                        if (cam_found1 = '1') then
                                vnext1 := '1';
                                vnext2 := '0';
                                nhit_wr <= '1';
                                nhits   <= phits + 1;                   -- Number of hits
counted for an event
                                nstate <= sstore_hits1;
                        else
                                vnext1  := '0';
                                vnext2  := '1';
                                nhit_wr <= '0';
                                nhits   <= phits;
                                nstate  <= sstore_hits2;

                        end if;

---------------------------------------
        -- Reads CAM2 address, transforms into proper ROAD NUMBER and writes HIT word.
---------------------------------------
                when sstore_hits2 =>
                        vroad := vcam_temp2 + "011111";
--                              6               3               <------------- [5
        +       2]      + 3      -------------->        13
--                      road            area            SEQ     (bit3 of 1-8 is left out)
        HDI             centroid
hit_word <= vroad & pulsearea & l3_trailer(18 downto 14) & l3_trailer(12 downto 8) &
centroid;
                        if (cam_found2 = '1') then
                                vnext2  := '1';
                                nhit_wr <= '1';
                                vdone   := '0';
                                nhits   <= phits + 1;                   -- Number of hits
counted for an event.
                                nstate  <= sstore_hits2;
                        else
                                vnext2  := '0';
                                nhit_wr <= '0';
                                vdone   := '1';
                                nhits   <= phits;
                                nstate  <= scheck_eof;
```

```
                        end if;
---------------------------------------
        -- Checks for EOF and writes trailer if EOF.
---------------------------------------
                when scheck_eof =>
                        vdone := '0';
                        if (hits_eof = '1') then
                                nhit_wr         <= '1';
--                                              5               3               8
        8                                                       4                       4
--                      header          NULL    EVENT_ID        No. of hits of event      ERR
bits            NULL
nhit_word       <= trailer_head & "000" & vevent & phits
        & verr_bits & "0000";
                                nstate  <= strailer_write;
                        else
                                nhit_wr         <= '0';
                                nhit_word       <= phit_word;
                                nstate  <= sreset;
                        end if;

---------------------------------------
        -- DUMMY state that the FSM stays in while writing TRAILER.
---------------------------------------
                when strailer_write =>
                        nhit_wr <= '0';
                        nstate  <= sreset;
                        nhits   <= zero8;

                when others =>
                        nstate  <= sreset;
                        err     <= '1';
                end case;

---------------------------------------
        --sstore_hits1, sstrore_hits2, sstore_hits2, sdummy2, scheck_eof, strailer);

---------------------------------------
                cam_start       <= vstart;
                cam_next1       <= vnext1;
                cam_next2       <= vnext2;
                hits_done    <= vdone;
                END process read_cam;

---------------------------------------
-- REgister Block
---------------------------------------
latch   : process (clock, reset, nstate, nhit_word, nhit_wr, nhits)
                BEGIN
                        if(reset='0') then
                                pstate          <= sreset;
                                phit_wr         <= '0';
                                phit_word       <= zero32;

--                              phits_done      <= '1';
                                phits           <= zero8;
                        elsif (clock='1' and clock'event) then
                                pstate          <= nstate;
                                phit_wr         <= nhit_wr;
                                phit_word       <= nhit_word;
--                              phits_done      <= nhits_done;
                                phits           <= nhits;
                        end if;
                END process latch;

                hit_wr          <= phit_wr;
                hit_word        <= phit_word;

---------------------------------------
-- Encoded FSM states for debugging
---------------------------------------
                with pstate select
                hitf_ostate <=          "0000" when sreset,
                                "0001" when sinitialize,
                                "0010" when scam_start1,
                                "0011" when sdummy1,
                                "0100" when scam_next1,
                                "0101" when sstore_hits1,
                                "0110" when sstore_hits2,
```

```
                                "0111" when sdummy2,
                                "1000" when scheck_eof,
                                "1001" when strailer_write,
                                "1111" when others;
end behaviour;
```

L3 Readout Control

```
--------------------------------------------------------------------------------
--      Program: l3_readout_control_edf.vhd
--      Created: by Arvindh Lalam (10/21/2001); used some of the code written
--                          by Shawn Roper
--      Description: This is the control block for the l3 readout
--      Remarks : Since the complexity of the program the FPGA express is used
--                          synthesize the program.
--
------------------------------------------------------------
--              Signal I/O
------------------------------------------------------------
--      signal name           direction             description
--              clk                     input                   clock line
--              reset             input                   reset line
--              done              input                   done processing signal from control
logic
--              Start_L3          input                   event start for readout
--              INH                     input                   inhibit signals
--              start             output                  event start for control logic
--              L3_DONE           output                  high if done processign
--              L3_BUSY           output                  high if busy processign
--              L3_BLOCK          output                  selector for final data mux
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--------------------------------------------------------------------------------
--      Declaration of the device ports
--------------------------------------------------------------------------------
entity l3_readout_control_edf is
port (clk,reset        : in std_logic;
          smt_id       : in std_logic_vector(2 downto 0);
                       -- Smt ID of the channel.
          l3_master_inhibit: in std_logic;
                       -- this is the master l3_busy bit
          start_L3     : in std_logic_vector(9 downto 0);
                       -- start_l3 bits from main logic
          l3_busy_inhibit      : in std_logic_vector(6 downto 0);
                       -- busy inhibit bits from other channels
          l3_block_inhibit  : in std_logic_vector(6 downto 0);
                       -- block inhibit bits from other channels
          l3_done_in: in std_logic_vector(4 downto 0);
                       --  indicates that a l3 buffer has finished block
          l3_available: in std_logic_vector(4 downto 0);
                       -- indicates that a l3 buffer has data available
          l3_wr_in:  in std_logic_vector(4 downto 0);
                       -- l3 write strobe out
          data0,data1,data2,data3,data4: in std_logic_vector(31 downto 0);
          l3_start_out      : out std_logic_vector(4 downto 0);
--        l3_write_strobe   : out std_logic;   -- or'ed l3_wr's from buffers
          l3_write                    : out std_logic;      -- EXTERNAL L3 write signal.
          l3_oen                      : out std_logic;      -- Enables output bus of
TRI_BUS module
          l3_data_out       : out std_logic_vector(31 downto 0);
          L3_BUSY_out         : out std_logic;
          L3_BLOCK_out        : out std_logic;
          l3_done_out       : out std_logic;

----------------------------------------
--      Debug pins.
----------------------------------------
          l3_ctrl_ostate      : out std_logic_vector(2 downto 0);
          vl3_done_out        : out std_logic;
          l3_ready_out        : out std_logic;
          vstart_l3_out               : out std_logic;
          ifstat                      : out std_logic_vector(1 downto 0);
          l3_buffer_ready_out  : out std_logic_vector(4 downto 0)
               );
end entity l3_readout_control_edf;

--------------------------------------------------------------------------------
--      Architecture Body Begins
--------------------------------------------------------------------------------
architecture behavior of l3_readout_control_edf is
       type mystates is (s0,s1,s2,s3);
```

```
        type myreadstates is (sinit,sdetermine,swait, sdone);
        signal nreadstate,preadstate: myreadstates;
        signal ns,ps : mystates;
        signal nstart,pstart: std_logic_vector(4 downto 0);
        signal nStart_L3, pStart_L3: std_logic_vector(9 downto 0);
        signal l3_channel_busy,l3_ready: std_logic;
        signal nl3_busy, pl3_busy,nreadout_l3,preadout_l3,pl3_block,nl3_block: std_logic;
        signal preadout_done,nreadout_done: std_logic;
        signal pl3_event,nl3_event,pend_l3,nend_l3: std_logic;
                                --  This bit is used to indicate a current l3 event
        signal pend_l3_event, nend_l3_event: std_logic;
        signal l3_buffer_ready:std_logic_vector(4 downto 0);
        signal nbuffer_select,pbuffer_select:std_logic_vector(2 downto 0);
--      signal nl3_done_out, pl3_done_out: std_logic;
        signal int_l3_data: std_logic_vector(31 downto 0);
        signal rotate_bit            :std_logic;
        signal channel               :std_logic_vector(2 downto 0);
        signal l3_buffers_done: std_logic;
        signal nl3_write, pl3_write   : std_logic;

        constant zero32:std_logic_vector(31 downto 0)
:="00000000000000000000000000000000";
        constant zero10: std_logic_vector(9 downto 0) := "0000000000";
        constant trailer_seq : std_logic_vector(4 downto 0) := "11110";
begin

----------------------------------------
--      This block checks to see if the channel is free.
--      Look at l3_master_inhibits and l3_channel_inhibits
----------------------------------------
        process(l3_master_inhibit,l3_busy_inhibit,l3_block_inhibit)
                variable i:integer;
                variable vtemp: std_logic;
        begin
                vtemp := l3_master_inhibit;
                for i in 0 to 6 loop
                   vtemp := (vtemp or (l3_busy_inhibit(i) or l3_block_inhibit(i)));
                end loop;
                l3_channel_busy <= vtemp;
        end process;

----------------------------------------
--      This process block will check if l3 data is available in buffers
----------------------------------------
        process(l3_available, l3_ready)
                variable vl3_ready: std_logic;
                variable i: integer;
        begin
                vL3_ready := '0';
                for i in 0 to 4 loop
                   vL3_ready := vL3_ready or l3_available(i);
                end loop;
                l3_ready <= vL3_ready;
                l3_ready_out <= l3_ready;
        end process;

----------------------------------------
--      This FSM determines if the bus is available.
----------------------------------------
        process(ps,preadout_l3, pStart_L3,l3_channel_busy, pl3_busy,l3_ready,
                        preadout_done,pl3_event)
        begin
----------------------------------------
--      set defaults
----------------------------------------
                ns <= ps;
                nl3_busy <= pl3_busy;
                nreadout_l3 <= preadout_l3;
                nl3_busy       <= pl3_busy;
                nl3_block      <= pl3_block;
                case ps is
                when  s0 =>
----------------------------------------
--      Are we in an L3 event with L3 data ready
--      Yes, request the channel by setting l3_busy high,  go to s1, else stay here
----------------------------------------

                        if((l3_ready = '1') and (pl3_event='1')) then
                           ns <= s1;
```

```
                        nl3_busy <= '1';
                     else
                        ns <= s0;
                        nl3_busy <= '0';
                     end if;
               when s1 =>
----------------------------------------
--      we have data and have requested the channel by setting l3_busy high
--      wait for l3_channel to go low indicating that l3 channel is free
----------------------------------------
                     if(l3_channel_busy = '1') then
                        nl3_busy <= '1';  -- channel is still busy, keep request high
                        nreadout_l3 <= '0';  -- this signal is for another FSM to
                                                                -- start the readout
process
                        ns <= s1;           -- stay in this state
                     else
                        nl3_block <= '1'; --   we have the channel, set nl3_block high
                        nreadout_l3 <= '1';    --  tell readout FSM to start
                        ns <= s2;
                     end if;
               when s2 =>
----------------------------------------
--      Channel is free.  Stay here until readout FSM is done
----------------------------------------
                     if(preadout_done = '1') then
----------------------------------------
--      There are no more channels with data ready to go..
--      So set nl3_busy and nl3_block low
----------------------------------------
                        nl3_busy <= '0';
                        nl3_block <= '0';
                        nreadout_l3 <= '0';   -- ok to set this to 0.
                        ns <= s0;   -- go back to s0 to wait for new data
                     else
----------------------------------------
--      More L3 data available, so stay here
----------------------------------------
                        nl3_busy <= '1';
                        nl3_block <= '1';
                        nreadout_l3 <= '0';   -- ok to set this to 0.
                        ns <= s2;
                     end if;
               when others =>   -- ERROR, goto RESET state
                        nl3_busy <= '0';
                        nl3_block <= '0';
                        nreadout_l3 <= '0';
                        ns <= s0;
               end case;

        end process;


----------------------------------------
--      This process block controls the individual l3 buffers.
--      It determines which one should have the internal l3 bus,
--      It waits for "readout_l3" signal before doing anything
----------------------------------------
        process(preadstate, pstart_l3,preadout_l3,l3_done_in,pl3_event,pend_l3_event,
               start_l3, rotate_bit, channel, smt_id, preadout_done, pbuffer_select,
               pstart, int_l3_data, l3_available, l3_buffer_ready)
               variable vl3_done,vendl3,vStart_L3,vpstart_l3: std_logic;
               variable vtemp, vptemp                  :std_logic;
               variable vl3_centroid_buffer, vl3_ext_wr                    : std_logic;
               variable i,j,iinc,iwr: integer;
        begin
----------------------------------------
--      Initialize the signals
----------------------------------------
               vStart_L3 := '0';
               vtemp  := '0';
               vpstart_l3 := '0';
               vptemp  := '0';
               ifstat <= "11";
               nl3_event <= pl3_event;
                                    -- indicates we are in a l3_event
               nend_l3_event <= pend_l3_event;
                                   -- indicates that current l3_event should end
               rotate_bit <= start_l3(6);
```

```
               channel <= start_l3(9 downto 7);
-----------------------------------------
--      This checks start l3 bits latched during
--      the current l3 event.  See ** below
-----------------------------------------
               for i in 0 to 5 loop
                  vtemp := vtemp or START_L3(i);
                  vptemp := vptemp or pStart_L3(i);
               end loop;
               if((rotate_bit = '1') and (channel = smt_id)) then
                  vStart_l3 := vtemp;
                  vpStart_l3 := vptemp;
                  ifstat <= "00";
               elsif (rotate_bit = '0') then
                  vStart_l3 := vtemp;
                  vpStart_l3 := vptemp;
                  ifstat <= "01";
               else
                  vStart_L3 := '0';
                  vpstart_l3 := '0';
                  ifstat <= "11";
               end if;
               vstart_l3_out <= vstart_l3;                          --** DEBUG pin.
-----------------------------------------
--      Checks if the l3 data transfer is complete.
-----------------------------------------
               if(preadout_done  = '1') then  --- should we end this l3 event;
                  nend_l3_event <= '0';  -- reset bit to prepare for the next l3 event.
                  nl3_event <= '0';
                  nStart_L3 <= zero10;
               elsif(pl3_event = '0') then
                                    -- are we currently waiting to start a l3 event
                  nl3_event <= vStart_L3;
                                   -- yes, if any bit is high, we enter a l3 event
                  nStart_L3 <= Start_L3;     -- register start_l3
                  nend_l3_event <= '0';      -- help synthesis tool
               else
                      nl3_event <= pl3_event;   -- we are in an l3 event.
                      nStart_L3 <= pstart_L3;   -- ** check pstart_l3 bits,
                                                -- if all zeros then leave this l3 event
                      nend_l3_event <= not vpStart_L3;
                                                                        --  on next cycle.
Use vpStart_l3 generated above
               end if;

-----------------------------------------
--      Generate l3_buffer_ready signals.   These are equal to
--      l3_available = '1' i.e. the buffer has data AND
--      start_l3 = '1' i.e. we need the l3 data for this event
--      If this is one, then output its content, need a separate signal for each buffer
--      L3_avaiable should be registered by individual l3 buffers
-----------------------------------------
               nreadstate <= preadstate;
               vendl3 := '0';
               vl3_centroid_buffer := '0';
-----------------------------------------
--      This calculates the parameters for the bits 1-3 of STRAT_L3
--       which stand for centroids.
-----------------------------------------
               for i in 0 to 4 loop
                  iinc := i + 1;
                  vendl3 := vendl3 or pstart_l3(iinc);
                  l3_buffer_ready(i) <= pstart_l3(iinc) and l3_available(i);
               end loop;
               l3_buffer_ready_out <= l3_buffer_ready;
-----------------------------------------
--      All of the done bits are ANDed to determine if
--       the last start has been completed
-----------------------------------------
               vl3_done := '1';
               for i in 0 to 4 loop
                  vl3_done := vl3_done and l3_done_in(i);
               end loop;
               l3_buffers_done<= vl3_done;
               vl3_done_out <= vl3_done;
               nreadout_done <= '0';
               nbuffer_select <= pbuffer_select;
-----------------------------------------
--      This logic is used to set l3_done_out
```

```
--      If we are in a l3_event, l3_done should be 0
--      If end_l3_event goes high, l3_done should be 1
--      L3_done should stay high, until reset by entering a new l3_event
----------------------------------------
--      nl3_done_out <= pl3_done_out;
               l3_done_out <= '1';
               if(pl3_event = '1') then
                  l3_done_out <= '0';
                  if(vl3_done = '1') then
                     l3_done_out <= '1';
                  end if;
               end if;
----------------------------------------
--      Generating the external L3 write signal to send out of STC.
--      pbuffer_select bits are ORed to generate WR and OEN.
----------------------------------------
               vl3_ext_wr := '0';
               for iwr in 0 to 2 loop
                  vl3_ext_wr := vl3_ext_wr or pbuffer_select(iwr);
               end loop;
               if( int_l3_data(31 downto 27) = trailer_seq) then
                  nl3_write <= '0';
               else
                  nl3_write   <= vl3_ext_wr;
               end if;
               l3_oen         <= not vl3_ext_wr;

----------------------------------------
--      The FSM starts here.
----------------------------------------
               case preadstate is
----------------------------------------
--      SINTI State
----------------------------------------
               when sinit =>
                      if(preadout_l3 = '1') then
                         nreadstate <= sdetermine;
                      else
                         nreadstate <= sinit;
                      end if;
----------------------------------------
--      SDETERMINE state
--  Need to determine which buffer gets to dump bus.
--  need to use if else to implement priority.
----------------------------------------
               when sdetermine =>
                      nreadstate <= swait;
                      nstart(0) <= pstart(0);
                      nstart(1) <= pstart(1);
                      nstart(2) <= pstart(2);
                      nstart(3) <= pstart(3);
                      nstart(4) <= pstart(4);
                      nstart_l3(0) <= pstart_l3(0);
                      nstart_l3(1) <= pstart_l3(1);
                      nstart_l3(2) <= pstart_l3(2);
                      nstart_l3(3) <= pstart_l3(3);
                      nstart_l3(4) <= pstart_l3(4);
----------------------------------------
--      Check to see if a buffer is ready.
--      Must use if then else since priority is assumed.
----------------------------------------
                      if(l3_buffer_ready(0) = '1') then
                         nstart(0) <= '1';   --  yes
                         nstart_l3(0) <= '0';  --  reset this start_l3 bit;
                         nreadstate <= swait;    -- go to wait state;
                         nbuffer_select <= "001";
                      elsif(l3_buffer_ready(1) = '1') then
                         nstart(1) <= '1';
                         nstart_l3(1) <= '0';  --  reset this start_l3 bit;
                         nreadstate <= swait;    -- go to wait state;
                         nbuffer_select <= "010";
                      elsif(l3_buffer_ready(2) = '1') then
                         nstart(2) <= '1';
                         nstart_l3(2) <= '0';  --  reset this start_l3 bit;
                         nreadstate <= swait;    -- go to wait state;
                         nbuffer_select <= "011";
                      elsif(l3_buffer_ready(3) = '1') then
                         nstart(3) <= '1';
                         nstart_l3(3) <= '0';  --  reset this start_l3 bit;
```

```
                        nreadstate <= swait;    -- go to wait state;
                        nbuffer_select <= "100";
                     elsif(l3_buffer_ready(4) = '1') then
                        nstart(4) <= '1';
                        nstart_l3(4) <= '0';  --  reset this start_l3 bit;
                        nreadstate <= swait;    -- go to wait state;
                        nbuffer_select <= "101";
                     else
-----------------------------------------
--      If we get here, then either all start_l3 bits have been reset
--      OR there is no more L3 content
-----------------------------------------
                        nreadstate <= sdone;  -- go to sdone to "clean up"
                        nbuffer_select <= "000";
                     end if;
               when swait =>  -- we started one of the l3 buffers,
                              --wait for done signal
                     nbuffer_select <= pbuffer_select;
                     if(vl3_done = '0') then
                        nreadstate <= swait;
                     else
-----------------------------------------
--  "done" bit has been returned.  Turn reset all nstart bits and
--  return to sdetermine to see if we have other buffers to output
-----------------------------------------
                        nstart(0) <= '0';
                        nstart(1) <= '0';
                        nstart(2) <= '0';
                        nstart(3) <= '0';
                        nstart(4) <= '0';
                        nreadstate <= sdetermine;
                     end if;

               when sdone =>
-----------------------------------------
--   Tell main FSM that we are done
-----------------------------------------
                     nreadout_done <= '1';
                     nreadstate <= sinit;  -- go wait for a new preadout signal
                     nbuffer_select <= "000";

               when others =>   -- should never come here
                     nreadout_done <= '1';
                     nbuffer_select <= "000";
                     nreadstate <=  sinit;
               end case;
        end process;

-------------------------------------------------------------------------------
-- Register Block
-------------------------------------------------------------------------------
reg: process (clk,reset,ns,nreadstate,nstart_l3,nl3_busy,nreadout_l3,nreadout_done,
                     nl3_event, nend_l3_event)
               begin
                     if(reset = '0') then
                        ps <= s0;
                        preadstate <= sinit;
                        pl3_busy <= '0';
                        pl3_block <= '0';
                        preadout_l3 <= '0';
                        preadout_done <= '0';
                        pl3_event <= '0';
                        pstart <= "00000";
                        pstart_l3 <= "0000000000";
                        pbuffer_select <= "000";
                        pl3_write <= '0';
               elsif(clk'event and clk = '1') then
                        ps <= ns;
                        preadstate <= nreadstate;
                        pl3_busy <= nl3_busy;
                        pl3_block <= nl3_block;
                        preadout_l3 <= nreadout_l3;
                        preadout_done <= nreadout_done;
                        pl3_event <= nl3_event;
                        pstart <= nstart;
                        pstart_l3 <= nstart_l3;
                        pbuffer_select <= nbuffer_select;
                        pl3_write <= nl3_write;
                     end if;
```

```
        end process reg;

-----------------------------------------
-- Assign outputs
-----------------------------------------
        l3_busy_out <= pl3_busy;
        l3_block_out <= pl3_block;
        l3_start_out <= pstart;
-----------------------------------------
--   use CASE to select one of the data_outs.
-----------------------------------------
        with pbuffer_select select
                int_l3_data <= data0 when "001",
                               data1 when "010",
                               data2 when "011",
                               data3 when "100",
                               data4 when "101",
                               zero32   when others;
        with preadstate select
                l3_ctrl_ostate <=  "000" when sinit,
                                   "001" when sdetermine,
                                   "010" when swait,
                                   "011" when sdone,
                                   "111" when others;
        l3_data_out <= int_l3_data;
        l3_write       <= pl3_write;
    --  send l3_write_strobe unregistered
--          l3_write_strobe <= l3_wr_in(0) or l3_wr_in(1) or l3_wr_in(2) or l3_wr_in(3)
or l3_wr_in(4);

end architecture behavior;
```

L3 Write Control

```
--------------------------------------------------------------------------------
-- File :       raw_data.vhd
-- Created by Arvindh Lalam (09/24/01)
-- Description : Loads data into the FIFO and controls the counter.
--------------------------------------------------------------------------------
--  Modified on 02/21/02 Arvindh Lalam.
--------------------------------------------------------------------------------


library altera;
use altera.maxplus2.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--------------------------------------------------------------------------------
--      Declaration of the Device ports
--------------------------------------------------------------------------------
entity raw_data is
port (  clock, reset   : in std_logic;
                                -- Clock and reset signals.
                event_start_int         : in std_logic;
                                -- Internal Event_start generated in Active channels.
                ctrl_wr                 : in std_logic;
                                -- WRREQ generated by data block sources.
                eof                             : in std_logic;
                                -- EOF generated by data block source.
                enable                  : in std_logic;
                                -- Enables RAW_DATA fifo depending on config bits.
                l3_data                 : in std_logic_vector(17 downto 0);
                                -- Input L3 data.

                l3_data_out             : out std_logic_vector(31 downto 0);
                                -- DATA that goes into FIFO.
                wrreq                   : out std_logic;
                                -- WRREQ for the actual FIFO.
                inc                             : out std_logic;
                                -- INC increments counter.
                ostate                  : out std_logic_vector(2 downto 0)
                                -- Encoded State for Debugging.
                );
end entity raw_data;

--------------------------------------------------------------------------------
--      Architecture Body Begins
--------------------------------------------------------------------------------
architecture behaviour of raw_data is
        type raw_states is (sreset, sdecide, swait_for_start, strailer);
        signal ns, ps  : raw_states;
        constant zero14 : std_logic_vector(13 downto 0) := "00000000000000";
        constant zero23 : std_logic_vector(22 downto 0) := "00000000000000000000000";
        constant zero32 : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";
begin

process(ctrl_wr,enable, eof,l3_data,event_start_int)
begin
-----------------------------------------
--      Initializing the asynchronous signals
-----------------------------------------
        wrreq <= '0';
        inc <= '0';
        --      done <= '1';
        l3_data_out <= zero32;
        case ps is
-----------------------------------------
--      Waits for event_start
-----------------------------------------
        when sreset =>
                --              done <= '1';
                if(event_start_int = '1') then
                   ns <= sdecide;
                else
                   ns <= sreset;
                end if;
-----------------------------------------
```

```
--      Checks if any of the FIFOs are enabled.
-----------------------------------------
        when sdecide =>
                --              done <= '1';
                if(enable='1') then                         -- Yes : Proceed with storing
data.
                   ns <= swait_for_start;
                   wrreq <= '1';
                   l3_data_out <= "10000" & "0000" & zero23;
                else                                        -- NO :  Go back to SRESET and wait
                                                                -- for next EVENT_START
                   ns <= sreset;
                   wrreq <= '0';
                   l3_data_out <= zero32;
                end if;
-----------------------------------------
--      Waits for WRREQ from the source block.
-----------------------------------------
        when swait_for_start =>
                --              done <= 0;
                ns <= swait_for_start;
                if(ctrl_wr = '1') then
                   wrreq <= '1';
                   l3_data_out <= zero14 & l3_data;
                   if(eof='0') then
                      ns <= swait_for_start;
                   else
                      ns <= strailer;
                   end if;
                end if;

-----------------------------------------
-- clock                |       |       |    |
-- EOF                ___|---|________
-- ctrl_wr            ____________|---|___________
-- data               ____________DDDDD_____________
-- WRREQ              ____________|----|_______
-----------------------------------------
--      Writes the trailer for current event.
-----------------------------------------
        when strailer =>
                --              done <= '0';
                wrreq <= '1';
                inc <= '1';
                l3_data_out <= "11110" & "000" & "0" & zero23;
                ns <= sreset;
        when others =>
                --              done <= '1';
                ns <= sreset;
        end case;
end process;

-------------------------------------------------------------------------------
-- Register Block
-------------------------------------------------------------------------------
process (clock, reset,ps)
begin
        if(reset='0') then
           ps <= sreset;
        elsif(clock='1' and clock'event) then
           ps <= ns;
        end if;
end process;

-----------------------------------------
-- Encode the State Information
-----------------------------------------
with ps select
        ostate  <=  "000" when sreset,
                            "001" when sdecide,
                            "010" when swait_for_start,
                            "011" when strailer,
                            "111" when others;
end architecture behaviour;
```

REFERENCES

[1] M.J. Sebastian smith, "*Applicatoin-specific Integrated Circuits*", Addison Wesley Longman.

[2] Shweta Lolage, "*VHDL design and FPLD implementation for silicon track card*", Masters Thesis, Fall 2000, Florida State University.

[3] Altera Corporation, "*APEX 20K: Programmable LogicDevice Family*", 101 Innovation Drive, San Jose, CA.

[4] John F. Wakerly, "*Digital Design Principles and Practices*", Prentice Hall, Englewood, NJ.

[5] Xilinx Incorporation., "*CoolRunner XPLA3 CPLD: Advanced Product Specification*".

[6] Altera Corporation. "*MAX 9000: Programmable LogicDevice Family*", 101 Innovation Drive, San Jose, CA.

[7] Xilinx Incorporation, "*CoolRunner XPLA3 CPLD Architecture Overview*".

[8] Stephen M. Trimberger, "*Field-Programmable Gate Array Technology*", Kluwar Academic Publishers.

[9] Altera Corporation, "*FLEX 10K Embedded Programmable Logic Device Family*", 101 Innovation Drive, San Jose, CA.

[10] Xilinx Incorporation, "*XC3000 SeriesField Programmable Gate Arrays: Product Description*".

[11] Altera Corporation, "*APEX II Programmable LogicDevice Family*", 101 Innovation Drive, San Jose, CA.

[12] Xilinx Incorporation, "*Virtex-II 1.5V Field-Programmable Gate Arrays*".

[13] Altera Incorporation, "*Stratix-Programmable Logic Device Family*".

[14] Altera Incorporation, "*Hardcopy Devices for APEX20K conversion*".

[15] www.particleadventure.com

[16] Fermi National Accelerator Laboratory profile at www.fnal.gov.

[17] G.C. Blazey, "*The D0 Run II Trigger*", Department of Physics, Northern Illinois University, DeKalb, Illinois.

[18] Wendy Taylor for D0 collaboration, "*An Impact Parameter Trigger for the D0 Experiment*", State University of New York, Stony Brook, NY.

[19] Wendy Taylor's talk at International Electrical and Electronics Engineers National Science Symposium, 2000, Lyon, France.

[20] Shweta Lolage, Reginald Perry "*The Silicon Track Card(STC) for the DZERO Experiment Upgrade – A Collaborative Physics and Engineering Design Project*", Proceedings of 2001 American Society for Engineers Education Annual Conference & Exposition.

[21] W.E.Earle, E. Hazen, M. Narain, U. Heintz, "*Silicon Track Card Specifications*", November 10, 2000.

[22] Altera Corporation, "*Implementing High-Speed Search Applications with Altera CAM*", 101 Innovation Drive, San Jose, CA.