To the Graduate Council:

I am submitting herewith a thesis written by Mark Douglas Allen entitled "Development of an FPGA-Based Heap Manager for the PHENIX Multiplicity Vertex Detector." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Robert E. Bodenheimer, Major Professor

We have read this thesis and
recommend its acceptance:

Dr. Donald W. Bouldin

Dr. Daniel B. Koch

Dr. Charles L. Britton, Jr.

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

DEVELOPMENT OF AN FPGA-BASED HEAP MANAGER FOR THE PHENIX MULTIPLICITY VERTEX DETECTOR

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Mark Douglas Allen

December 1996

Dedicated to my loving parents,

Dr. Lynn Allen and Jo Allen,

who have supported me through the best and worst

of times and refused to think less of me even when

I thought less of myself.

Acknowledgments

Many people have provided me with aid in the preparation of this thesis. Although so many have helped me in ways that I cannot repay, Dr. Robert Bodenheimer stands out as the one without whom I would not have attempted graduate school; and his support, advice, and friendship have meant more to me than I can put into words. Also, I would like to thank Dr. Don Bouldin, under whose contract I completed my research for this thesis and who has provided me with numerous opportunities to expand my knowledge in the field of electrical engineering; and Dr. Daniel Koch, who has always been willing to take time out to help me and answer my questions. A very special extension of gratitude goes to Dr. Chuck Britton, who has gone far above any expectations I might have had in providing me with opportunities, understanding, insight, and general craziness on and off the job and whose character will always serve as inspiration to me as I go through life.

I would like to extend my thanks to all of the people with whom I have worked at Oak Ridge National Laboratory. In particular, Nance Ericson's friendship and patience are deeply appreciated, as are his suggestions, help, and attempts to make me a more responsible person. Others I would like to thank include Dr. Alan Wintenberg, Dr. Mike Simpson, Mike Emery, Bill Bryan, Gary Alley, Mark Musrock, Jim Walker, and Lloyd Clonts, whom I bugged to death when I got stuck on a UNIX problem.

Without my friends, I would have been totally lost. Among the friends who have stuck by me through the past several years are Shane Frank, whose insane interest in the unusual helped keep me from losing my mind; Tom Turner, who has always been a true friend; Tarra and Paul McCall, whose wisdom and friendship have been invaluable to me; Kathleen Wage, whose nutty e-mails sustained me through some pretty tough times; Martin Holder, a great friend and tremendous lab partner; and the stable of student work horses at ORNL: Randall Smith (Go Vols !!), John Writt, Ryan Lind, "General" Gentry Jackson, David French, and Rich Palmer. Also, I would like to thank secretaries Daisy Bolton and Joann Loy at The University of Tennessee, Knoxville, and Melissa Byrd and Linda Anthony at ORNL for all their help and guidance. Lastly, but perhaps most importantly, I would like to thank Ann Lacava of the graduate school for guiding me through the process of writing this thesis and providing suggestions and hints on how to make my thesis the best it could be.

Of course, I would not have made it this far, academically or otherwise, without the aid and support of my parents, Lynn and Jo Allen. By sticking by me, they ensured my success in writing this thesis; and I especially want to thank them for reading and editing this work and providing me with encouragement and advice. Thank You.

Most of all, I thank God, Who gave me the two greatest Gifts: Life and His Son, Jesus Christ, the One True Master and Teacher.

Further Up and Further In.

Abstract

In this thesis, a technique for implementation of a heap manager for management and control of instrumentation for the PHENIX Multiplicity Vertex Detector (MVD) is discussed. In the PHENIX MVD, an analog memory is utilized as an analog storage device so that analog data may be acquired from a charge-sensitive preamp, examined, and, if an event threshold is realized, converted by an analog-to-digital converter into digital data. A field programmable gate array (FPGA) solution for control and data management is developed as a means of maintaining the front-end electronics, which includes preamps, analog memories, and ADCs as well as a number of support logic devices. The heap manager, as a system controller, provides all timing and digital data acquisition functions needed by the MVD system; and a series of interfaces contained in the heap manager provides the link between an external CPU and the front-end electronics. Results from a prototype of the MVD system are examined and used to expose the strengths and weaknesses of the heap manager; and further refinement of the heap manager architecture is enacted to take advantage of the knowledge gleaned from the prototype and yields the power, speed, and resource allocation specifications for the system as configured for implementation in a multi-chip module. The heap manager is

shown to be fully functional with a clock of 60 MHz and has a power dissipation of approximately 700 mW at 40 MHz.

Table of Contents

List of Figures

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

List of Tables

Table Page

Table Page

Table Page

Chapter 1

Introduction to PHENIX

In the field of physics, theory must give way to experience. Because an experiment must be devised to support or disprove any postulate, scientists are charged with the responsibility of creating a test case that will promote or exclude a particular theory. Among the most fundamental physical phenomena is the "Big Bang," the idea that the universe existed as a singularity and all matter and natural laws evolved from a fixed moment in the past; therefore, scientists, searching for clues about the relationships between energy and matter in the early universe, develop experiments which point toward the emerging forms of matter as the universe expanded in the moments after the "Big Bang." According to the theory of the "Big Bang," as the universe cooled and expanded, the particles erupting from the singularity lost energy and bonded together to form atoms and complex molecules. Hence, because the particles in the first moments of the universe are the fundamental building blocks of matter, an enormous amount of energy must be expended to separate the particles from their assumed atomic structures. In high energy physics experiments, colliders are used to stage forced interactions between high-momentum particle beams; and data is collected by surrounding the collisions with

multiple detectors and instrumentation to measure the results. The data collected from the collisions provides evidence as to whether the observed energy/matter relationships of the experiments fit into the theory of the "Big Bang" model of creation.

1.1. Quark Gluon Plasma and PHENIX

The quark gluon plasma is thought to be the state of the universe a few microseconds after the "Big Bang." A time line of events, for the first moments after the universe started expanding, is illustrated in Figure 1.1.1. Before t=0, time has no meaning; and between t=0 and $t=10^{-43}$ seconds, the physical laws of nature were so intertwined that no theories may explain exactly what the universe may have looked like. The Planck Time ($t=10^{-43}$ seconds) determines the moment at which the diameter of the universe finally exceeded its Compton wavelength; therefore, before the Planck Time, the Heisenberg uncertainty principle says that any attempt to quantify the relationship between matter and energy is futile. As the universe expanded and cooled and the four fundamental forces (gravity, strong nuclear force, weak nuclear force, and electromagnetic force) broke symmetry, matter and energy began to take on forms which can be explained theoretically; therefore, scientists have deduced that a few microseconds beyond t=0 the universe assumed the structure of a high energy plasma known as the quark gluon plasma [1].

The nature of the quark gluon plasma (QGP) is such that matter is at its most fundamental form of existence. Quarks are thought to be the simplest state of matter, and

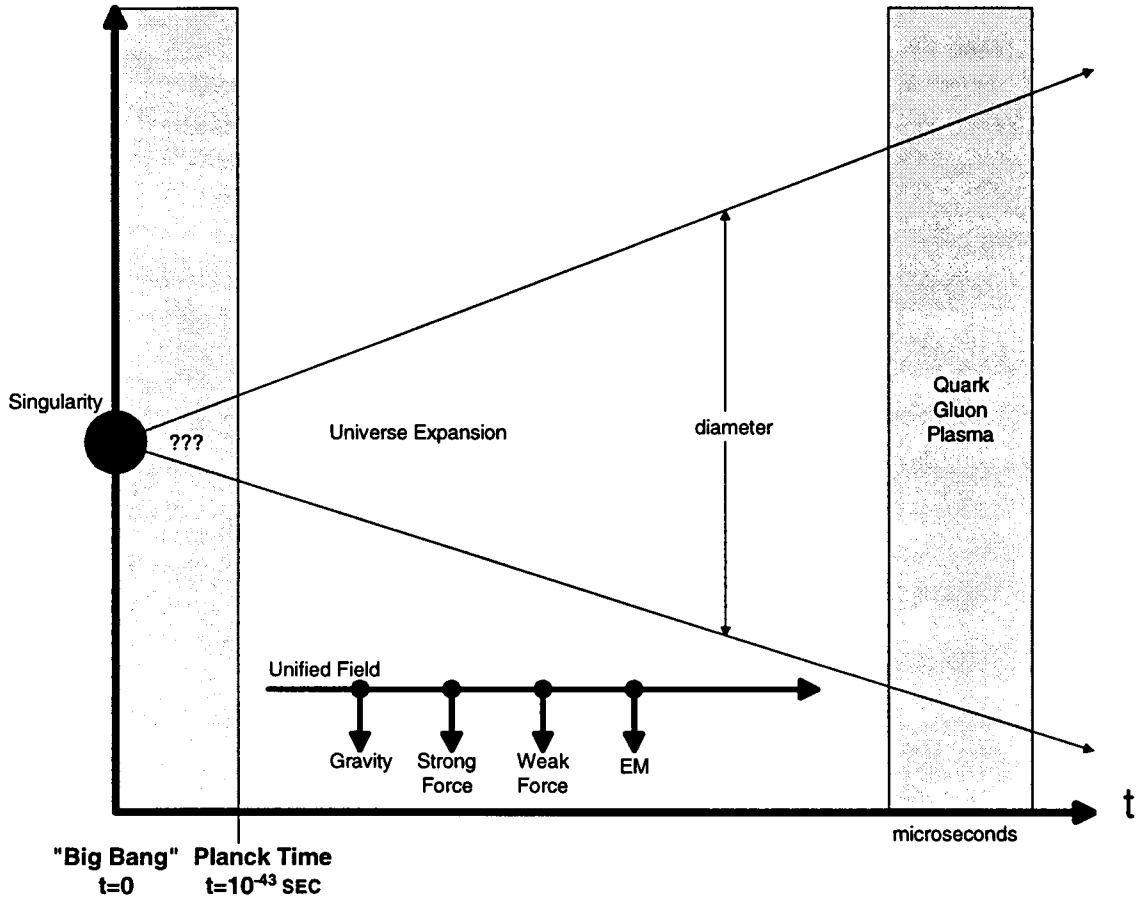


Figure 1.1.1. Universe Expansion after the "Big Bang"

gluons are believed to be the force carriers that bind the quarks into particles. In the QGP, the energy levels present create enough momentum in the quarks to prevent the gluons from maintaining bonds between the particles; and all quarks are "free," with gluons forming and breaking bonds continuously [2]. As the universe cooled and the energy of the quarks grew less, gluons drew the quarks together to form matter such as hadrons, which serve as building blocks for atoms and molecules.

Hadrons are particles made up of quarks and gluons and serve as the prototypes for heavier particles such as neutrons and protons. A hadron, as demonstrated in Figure 1.1.2, is composed of quarks bonded together by gluons; therefore, the energy required to create free quarks must be enough to break the bonds formed by the gluons. Obviously, not enough energy may be input into an experiment to free all quarks from a particle beam; but the possibility exists that some hadrons may be energized enough to free a few quarks. To date, no free quarks have been observed in the laboratory [2]; therefore, to study the QGP, a state of the universe in which only free quarks and gluons existed, the accelerator utilized must employ high energy collisions between heavy ion beams to produce a few quarks and study the return of the quarks to their less excited state of existence in the form of hadrons. Toward realization of the quark gluon plasma, the PHENIX experiment proposes to study the number and movement of particles as they are emitted from a high energy beam collision in an effort to isolate hadrons and, hopefully, the quarks of which they are made.

In the PHENIX experiment, to re-create and understand the quark gluon plasma, beams made up of many individual particles will be guided and accelerated at high speeds,

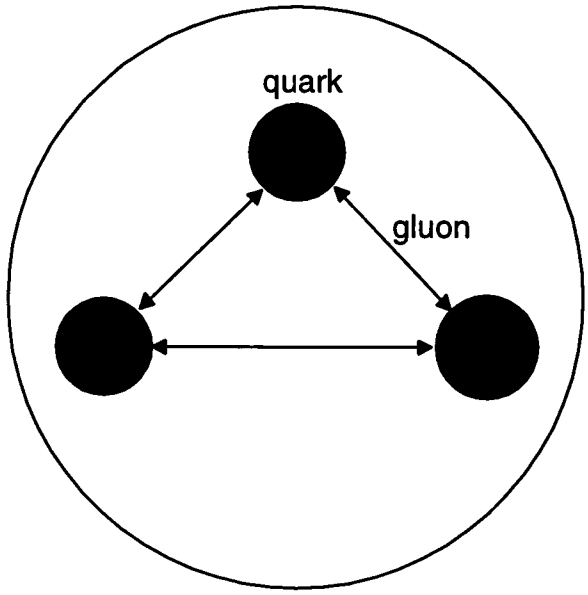


Figure 1.1.2. Quark-Gluon Model of a Hadron

through the use of powerful magnets, to a point at which the beams will collide. In theory, the colliding particles will send off smaller, daughter, particles which are more fundamental in nature than the heavy ions; and the PHENIX detectors are conceptualized to study the quark gluon plasma signatures associated with the detection of real and virtual photons and hadrons [3]. Thus, the determination of the existence of the quark gluon plasma will be made by studying the types of particles emitted by each beam collision. In the PHENIX experiment, high-energy collisions of beams of energies approaching 250 GeV [4] are expected to produce large numbers of particles and, hence, data. By tracing the paths taken by the particles and quantifying the energy resulting from the interactions of the particles with the detector, PHENIX will attempt to study the mode by which the quark gluon plasma is created.

To capture the particles created in a beam collision, the PHENIX detector will employ a number of subsystems which will be placed at several points around the beam trajectory. A diagram of the PHENIX detector and its subsystems is shown in Figure 1.1.3. In PHENIX, each detector subsystem will be situated in such a way as to capture events that are thought to occur at certain points in the experiment. Within the detector subsystems, the interaction of charged particles will vary with the magnetic field emitted by the magnets. Because some particles, such as photons, carry no charge, the path taken by the uncharged particles will be straight away from the beam collision; however, charged particles, deviating from the beam path in relation to the magnetic field, will cause a disturbance in the silicon detectors which surround the beam collision. The data resulting from the PHENIX experiment should provide clues as to the nature and paths of the

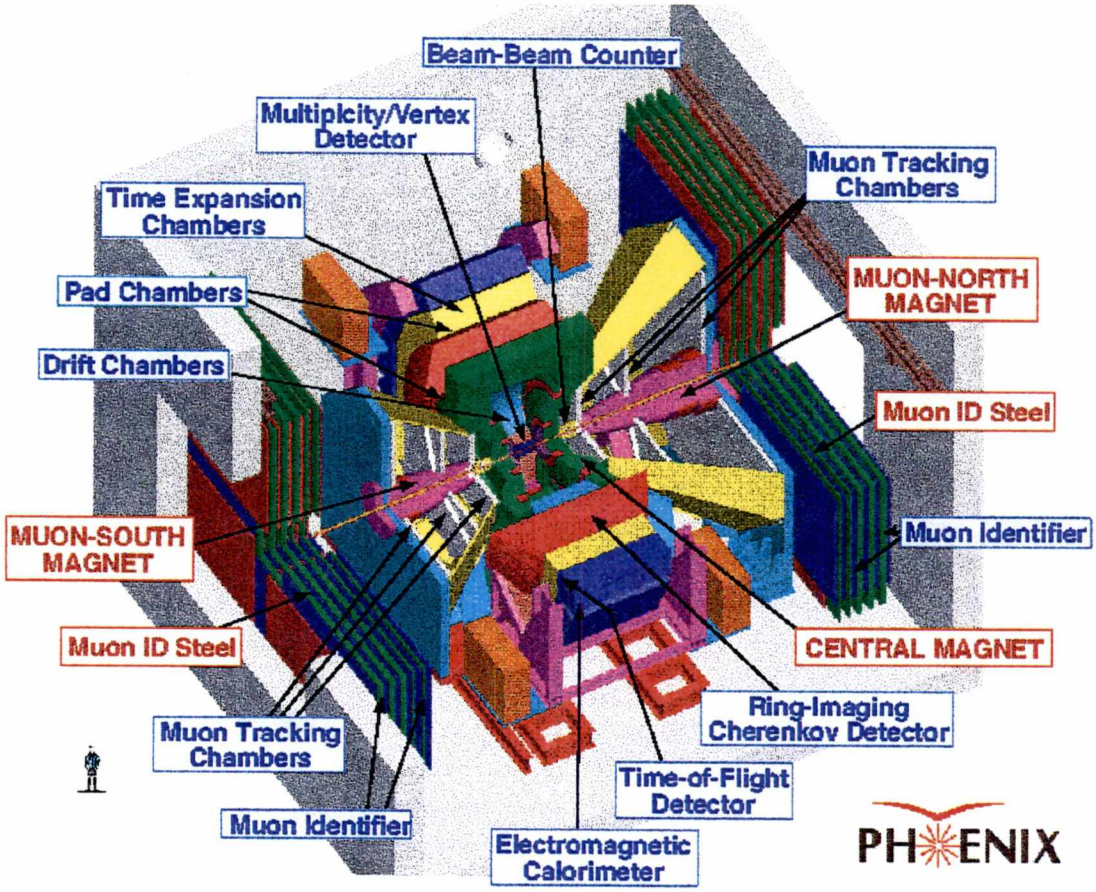


Figure 1.1.3. PHENIX Detector and Subsystems

Source: [http://www.rhic.bnl.gov/~phenix/GIF].

particles encountered from the high-energy beam collisions.

1.2. Multiplicity Vertex Detector (MVD)

The central detector in the PHENIX apparatus is the Multiplicity Vertex Detector (MVD). As the beams collide and particles are radiated outward from the beam collision, the cylindrical detector of MVD will collect data in the form of electrical signals as the particles interact with the silicon strips in the detector shell. Measurements to be made include multiplicity, the number of daughter particles interacting with the surface of the detector, and vertex angle, the angle at which daughter particles separate from the beam; therefore, the central detector, placed in the region of the beam collision, is called the Multiplicity Vertex Detector (MVD) and is represented, in cross-section, in Figure 1.2.1 as two concentric cylinders that are made up of silicon strips that will detect the particle interactions. When particles diverge from the beam path due to a collision, the smaller, charged, daughter particles will emanate toward the detector as dictated by the magnetic fields surrounding the beam path; and, if they decay further, the daughter particles will emit more particles which will make their way, along secondary paths, toward the silicon strips. In Figure 1.2.2, the creation of daughter particles is shown. Between the time when they emerge from the beam collision until the moment when they interact with the detector surface, daughter particles may decay several times; and each diverging path of the particles must be retraced to find the point of origin. Thus, to determine the position of the

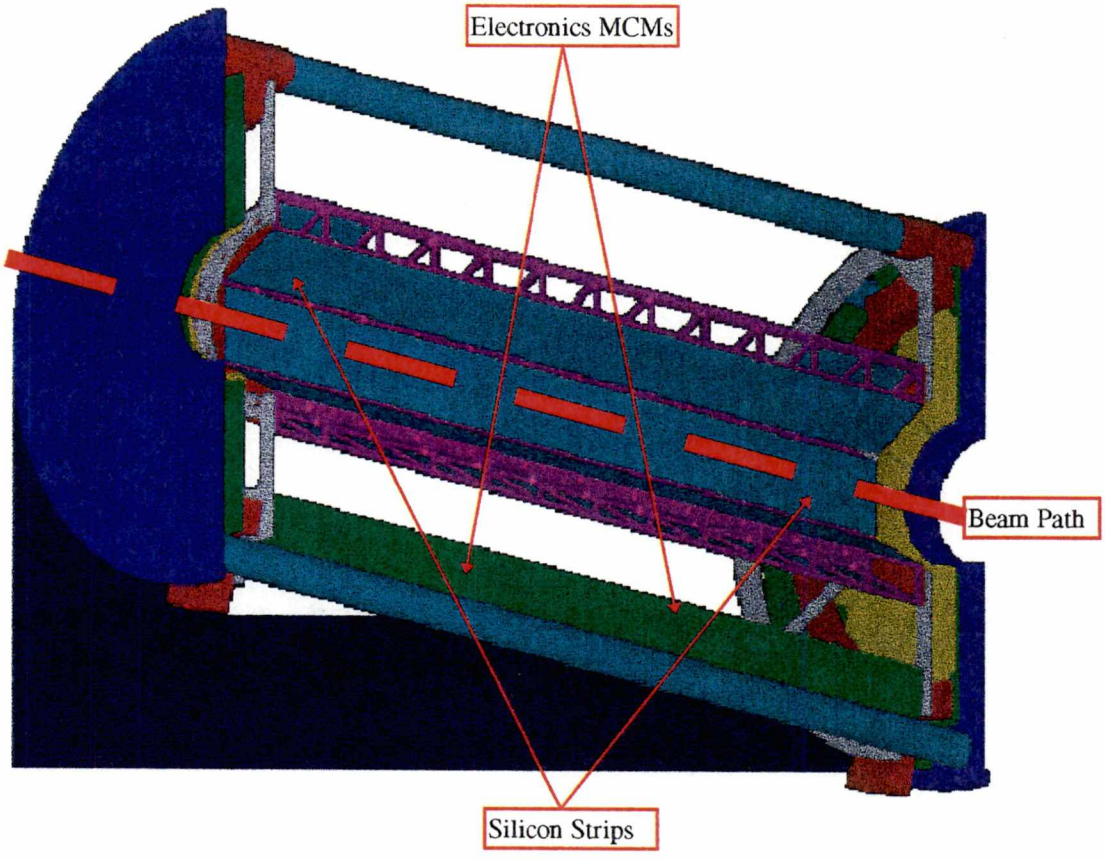


Figure 1.2.1. MVD Detector Structure

Source: [http://www.rhic.bnl.gov/~phenix/GIF].

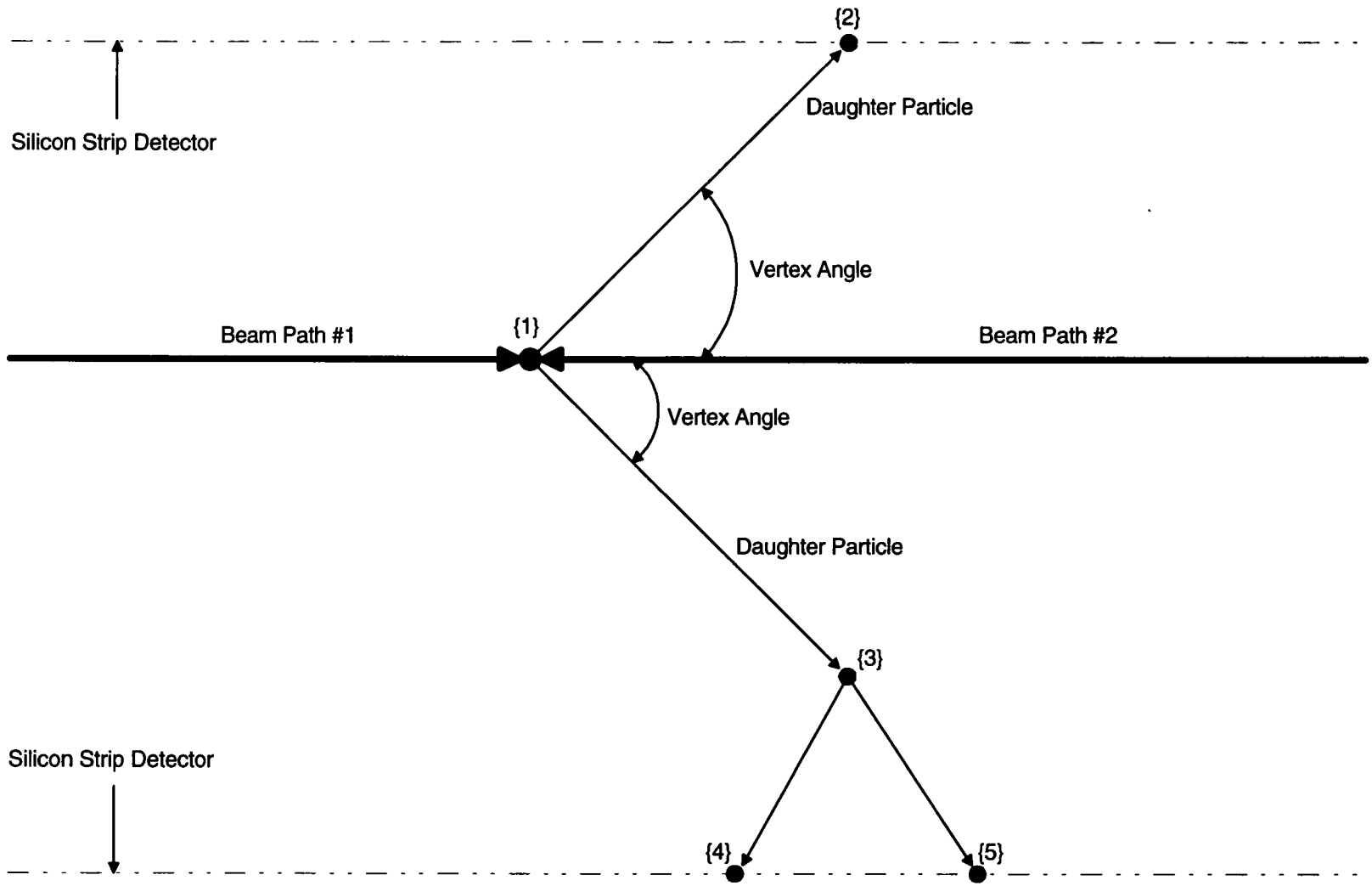


Figure 1.2.2. Splitting of Particle into Daughter Particles

first decay of daughter particles at {1}, a trace must be made which includes the vertex angle of {3}, as it decays into particles {4} and {5}, as well as the path of particle {2}. To perform its task, the PHENIX MVD subsystem must be able to map the daughter particles as they enter the detector from the beam and measure the energy that is contained in the particles. For MVD, the substantive information to be recorded includes the multiplicity, or the number of particles interacting with the detector, and the location of the particle/detector interactions.

For a detector the size of the one needed to chart and quantify the large number of particles produced in the PHENIX experiment, a multi-channel electronics system must be put in place to handle the signals as they occur in the detectors. Approximately 34,000 channels of electronics, divided into 256-channel modules with their corresponding silicon detector strips, will be utilized in MVD to measure events. The resolution of particle collisions for MVD requires that the electronics handle events that may occur at intervals of 105 ns, which is the period between beam collisions; therefore, the timing of the electronics will be produced using a beam clock rate of 9.5 MHz. Additionally, a clock with a period of 26.25 ns (four times the speed of the beam clock) will be introduced into the system so that the digital portion of the system may carry out multiple operations each beam cycle. The front-end electronics will have a resolution of eight bits, and a preamplifier shaping time of 25-30 ns will allow the system to achieve a signal-to-noise ratio of 20:1, or 2500 electrons rms [4]. The low-noise nature of the system requires that the electronics be placed in close proximity to the detectors which they service; therefore, the size of the electronics modules must be minimized so that the instrumentation may be

bonded directly to the detector shell; and the electronics used must be radiation hardened to avoid any degradation in performance in the high-radiation environment near the beam path.

As particles interact with the detector, the instrumentation must sum the charge over several adjacent detector strips and relay multiplicity information to the system controller; and, when an event threshold is reached, a LVL-1 trigger will be produced to tell the electronics that a significant event has occurred and to begin processing data. Interpretation of the LVL-1 trigger and the control and timing of the system is left to a digital controller; and, to facilitate changes in the timing and data readout of the system, the digital portion of the electronics must be implemented in a re-programmable field programmable gate array (FPGA) technology. The MVD analog electronics consist of a preamplifier, an analog memory unit, and an ADC, all of which must be designed as custom application specific integrated circuits (ASICs) to maximize speed and reduce power requirements and cost.

To allow sufficient time for the electronics to process and evaluate whether an event has taken place, the instrumentation will utilize the analog memory, which is the analog version of a digital RAM, to store the analog inputs from the detector and preamp. In previous experiments, miles of cable were used as delay lines to provide a timing interval for the processing of event thresholds. For PHENIX and MVD, the analog memory precludes the necessity for bulky cables and will provide the delay line in the form of a forty-pin ASIC in which analog values may be stored and retrieved from an addressable array of switched capacitors [5]. During normal operation of the MVD

system, data from the detectors will be stored, in one of sixty-four memory locations of an analog memory unit, each beam clock. Implementation of a repeating address loop will allow data to be stored in the analog memory until a LVL-1 trigger is issued, at which time the memory will be read and the analog data will be digitized.

The data collected by the electronics of the MVD system will include a pre-trigger sum (Pre) value and a post-trigger sum (Post) value. Each trigger sum will be related to an event that has been realized as a result of the evaluation of an event threshold. In Figure 1.2.3, the concept of Pre and Post event timing is illustrated. Each time a LVL-1 is generated, a Pre and a Post analog value will be retrieved from the analog memory; and the amplitude of the Pre will be subtracted from the amplitude of the Post to give a difference voltage that will yield the energy in the event.

In the final MVD system, two means of processing data will be available: raw mode and correlator mode. In raw mode, both Pre and Post values will be digitized by the ADC; and two sets of data packets will be issued to a central computer for storage. Correlator mode, however, will take advantage of a circuit resident in the analog memory that will allow the Pre analog value to be subtracted from the Post before a conversion is made, thus saving time and storage space. Consequently, in correlator mode, a single ADC conversion will take place for each LVL-1 trigger; and, after it is digitized by the ADC, the data will be relayed, in the form of a single packet, to the central processing unit, which is responsible for storing the information for analysis. Effective use of raw and correlated data will allow the MVD system to characterize events based on the energy suggested by the stored Pre and Post analog data and will permit system control of the

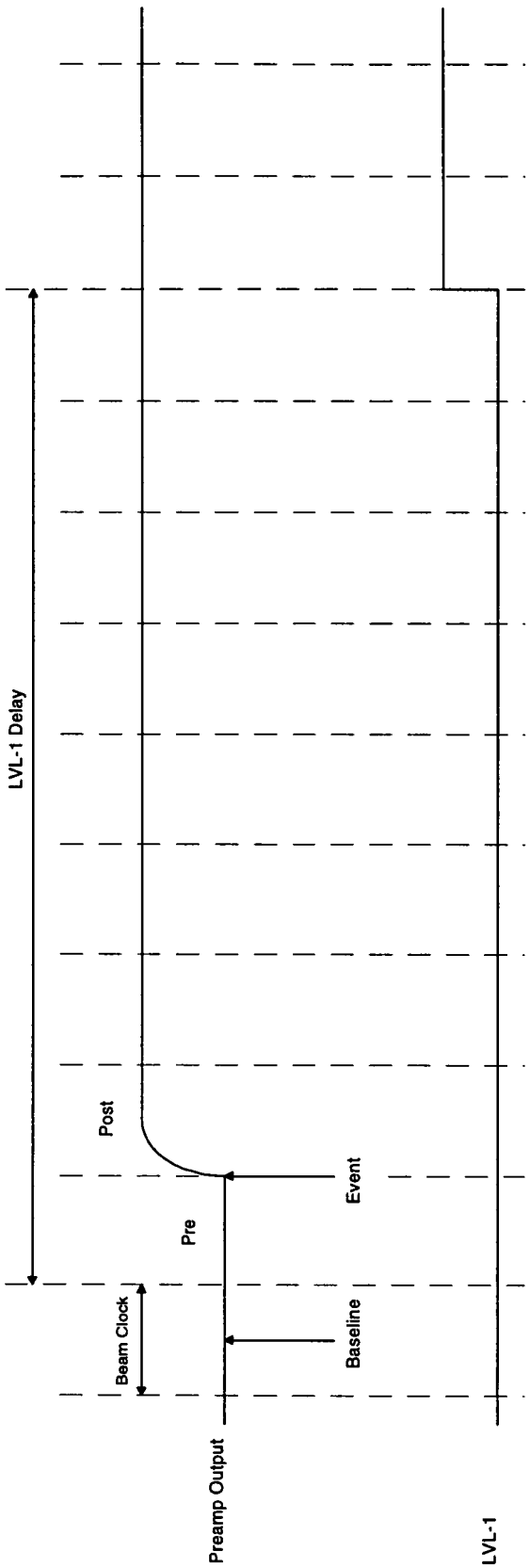


Figure 1.2.3. LVL-1 Delay Timing

amount of data logged by the system.

Since time is needed to determine whether or not an event has occurred, a delay between the storage of analog values and the decision to produce a LVL-1 is necessary. As data is processed and stored, a decision, based on the event threshold, must be made as to whether a particular piece of data represents a valid event. Thus, a constant LVL-1 delay, as shown in Figure 1.2.3, exists between the time when data is stored and when the data, if tagged by a LVL-1, is read and digitized. Because the total charge associated with an event is related to both Pre and Post analog values, the LVL-1 delay is measured from the beam clock preceding the event, or Pre. When a LVL-1 is issued, all channels of the instrumentation module involved in the calculation of the current sum predicating a LVL-1 trigger will be digitized. Thus, the data sent to the central computer will allow the system to extract the multiplicity information from all detector channels involved in the event.

The final implementation of the MVD subsystem requires that multiple instrumentation modules be placed in close proximity to the detectors which they service. To maximize the instrumentation relative to the small space around the detectors, the 256-channel electronics modules are to be developed into multi-chip modules (MCMs) which will be coupled with the detectors. Due to the large number of channels in the MVD system and the high cost of water cooling, the power specifications for the system are instituted to allow for an air cooling system. Thus, although the FPGA technology to be used is unspecified, the per-channel power requirement of 15 mW/channel is calculated with the ASICs and the FPGA included.

The Multiplicity Vertex Detector is a self-contained subsystem of PHENIX that utilizes advanced ASIC technology and re-programmable FPGAs to measure the energy signatures produced by high momentum beam collisions. The instrumentation is miniaturized through the use of a multi-chip module architecture to minimize the distance between the detector and the electronics, and the power consumption is reduced by developing custom electronics modules which are specialized for the experiment. With its fellow subsystems in the PHENIX apparatus, MVD will help determine the existence of the quark gluon plasma.

1.3. The MVD Beam Test As a Basis for Study

The development of the MVD system is a multi-stage project, of which the first milestone is the beam test [3]. Conducted at Brookhaven National Laboratory, the beam test is meant as a functional test during which a sixty-four channel prototype of the electronics and detector is mated. Since it is intended as a precursor to the final MVD system, the beam test does not have stringent speed and power requirements; but the design of the beam test unit will provide a benchmark for further system development.

The storage matrix of the analog portion of the electronics is required to handle sixteen separate beam-clock-delineated analog values instead of the sixty-four required by the full MVD system; therefore, the delay available to the beam test system between the time when it stores an analog value and when it must retrieve the value is significantly

shorter than for the full MVD system. In the beam test system, a constant delay of nine beam clocks exists between the Pre beam clock period and the LVL-1 trigger; however, in the final implementation of the MVD subsystem, the LVL-1 delay will be programmable over a range of sixteen beam clocks, thus allowing the LVL-1 delay to be between thirty-two and forty-eight beam clocks wide.

Although it is a miniaturized version of the final MVD system, the sixty-four-channel unit is designed with significantly more test and diagnostics options than the 256-channel system; and special emphasis is placed on flexibility so that the limits of MVD can be evaluated. Thus, the digital logic required in the beam test system is enhanced from the architecture proposed for the final MVD electronics. Exploring the evolution of the MVD system, including changes made to the heap manager during and after the beam test, this thesis focuses on the development of the PC interface and digital electronics used for control and readout for MVD and defines and expands on the system characteristics from the view point of the heap manager.

Throughout this thesis, the details of forging a working relationship between the heap manager, front-end electronics, and CPU are posited and tested for merit. Since the electronics are pre-designed and fabricated, the proposals and methods expounded in this work assume the qualification of the analog components of the FEE and concentrates on the trade-offs associated with control and readout of the electronics. By providing a study of the development of the heap manager and CPU interface for use in the Multiplicity Vertex Detector, this thesis dissects techniques of design associated with digital control

and interfacing of discrete and continuous signal components as they interact to form a cohesive MVD beam test design.

To encompass the bulk of work done in preparation of the beam test, this thesis is divided into chapters that cover the design of the heap manager and PC interface from conception to conclusion and presents the first steps toward the realization of a complete MVD system for fabrication in an MCM. Thus, Chapter 1 of this thesis serves as a basis for PHENIX development and presents the physics behind the experiment. Chapter 2, then, studies the suppositions made to define the system in terms of the interfaces between the various system components and proposes a framework for creation of a cooperative heap manager. The core of the thesis centers on Chapter 3 and the design process which goes into the implementation of the logic and production of a heap manager which conforms to the specifications laid out in Chapter 1 and Chapter 2; and Chapter 4 illustrates the system integration which produces a working beam test platform and introduces the CPU interface and software, which is created to control the heap manager. Next, Chapter 5 presents the changes enacted in the heap manager as a result of decisions made based on the beam test results and defines the first steps toward construction of a multi-chip module. Finally, Chapter 6 reflects on the methodology used for design of the heap manager and CPU interface and compares and contrasts the design elements of the preliminary beam test architecture with the post-beam-test system to provide insights on the pros and cons of the assumptions made at the beginning of the design process. In the final analysis, this thesis verifies the achievement of a working digital platform for control and readout of the Multiplicity Vertex Detector and uses the MVD beam test as an

example of one method for achieving a re-programmable solution for control of detector electronics.

Chapter 2

Heap Manager Responsibilities in the MVD Beam Test

Because it is the first hurdle for MVD, the beam test unit is developed as a prototype for the final MVD system. The controller structure of the beam test Multiplicity Vertex Detector must be able to handle the timing requirements associated with system-level components as well as service the needs of the external computer that will accept data and issue system commands via a human interface. Since multiple channels of the front-end electronics (FEE) must be handled by a single heap manager (HM), a standard unit model, called a front-end module (FEM), is created to define the system. The FEM contains a heap manager to provide digital controls and a series of analog components that accept electrical impulses from the detector, store the inputs in an analog memory, and provide a means of converting the analog signals into digitized values. The data path for the system includes inputs from the computer and FEE into the heap manager, and the control functions are implemented inside the heap manager to manage the flow of data and return the information of interest to the computer. As a self-contained control and readout structure, the front-end module serves as a paradigm for the beam test electronics.

2.1. Front-End Module Architecture

The front-end module for the MVD system is comprised of a number of data and timing paths to and from the heap manger, which communicates with an external interface to receive instructions and return data. As seen in Figure 2.1.1, the heap manager is the central controller around which the FEE and external computer interface are built. The timing path for the analog portion of the FEM is from right to left in the FEE portion of the diagram. Thus, the detector emits a signal that is amplified by the preamp; and the analog amplification of the signal is stored in the AMU. To determine whether a signal represents an event of interest, a trigger sum of several channels is inspected; and, if the trigger sum exceeds a predetermined threshold, a Level-1 is sent to the controller, which reads the appropriate memory location in the AMU and begins a conversion with the ADC. Subsequently, the digital readout from the ADC is fed into the heap manager, which returns a packet of information to the computer via a FIFO that acts as a buffer between the heap manager and the CPU. The front-end module will be duplicated in the final MVD system and implemented as a multi-chip module to offer a complete data acquisition system for the multi-channel detector subsystem.

External to the FEM are the programming and data collection modules (DCMs) which are proposed as the human interface. For the beam test, the heap manager control unit provides separate outlets to two external human interfaces: a PC or a CAMAC ECLine Model 4302 Triple Port Fast Memory Unit; and, for data storage, the PC interface reads data from an on-board FIFO, while the CAMAC module has the data synchronized

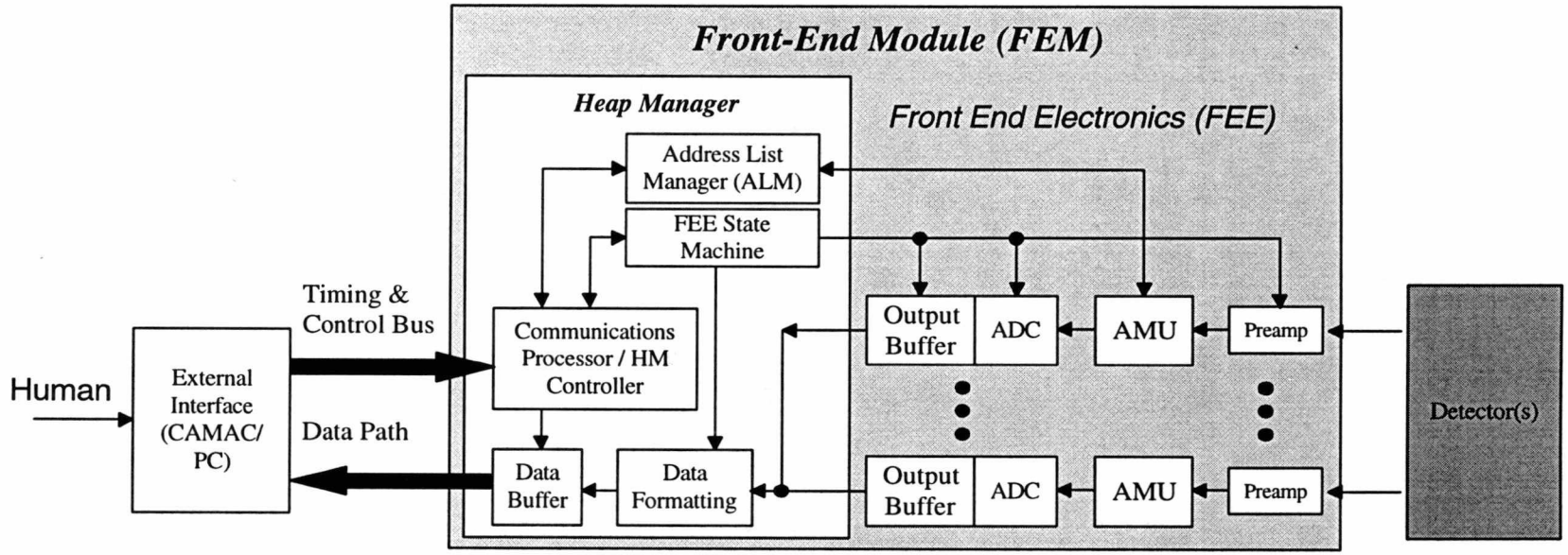


Figure 2.1.1. Front-End Module for MVD

with a clock generated by the heap manager. The control signals for the heap manager remain constant throughout the acquisition process but can be channeled through the PC or CAMAC ports, although only one of the two external interfaces may be used at a time; and the programming and control interface allows in-circuit reconfiguration of the heap manager. Because it is set up to service two separate external human interfaces, the heap manager must be manually configured to talk to the PC or CAMAC.

In addition to the external interfaces, the functions of the front-end module can be separated into two divisions: heap manager and front-end electronics. A further break down of the FEE produces three more divisions: preamp, AMU, and ADC; and the heap manager can be realized in terms of its main components: heap manager controller and address list manager (ALM). The segregation of the FEE components is based solely on the fact that each of the three units is an individual ASIC and has its own separate control and timing structure, whereas the heap manager structure is chosen to take advantage of the re-programmability and resource management associated with the FPGA technology. Because the methodology needed to produce the functionality of MVD is nontrivial and the links between the system components form a complex network of interactions, the design process may be expedited by breaking down the system into its constituent components, each of which may be designed as self-contained modules.

2.2. Front-End Electronics

For the beam test, two FEE mother boards of thirty-two channels each are combined with one heap manager into a sixty-four channel unit. Each FEE board of the MVD system includes a set of four eight-channel charge-sensitive preamps, four eight-channel AMUs, and four eight-channel ADCs for a total of thirty-two channels of each species. Thus, the HM functions need to be doubled in some instances to allow independent control of a specific mother board; and, consequently, the heap manager is able to work with one or both FEE boards at a time and is able to maintain maximum control of FEE functionality under the specifications. The FEE and the relevant inputs and outputs of the heap manager are shown in Figure 2.2.1. By accepting charge from the detector and providing memory and a conversion route, the FEE provides all the analog functionality of the MVD system. The job of the heap manager is to coordinate with the FEE and provide a data path from the detector to a central computer which stores the data permanently. Since it is an ASIC with unique timing requirements, each of the three components of the FEE must be dealt with separately when constructing the timing scheme of the heap manager. The pre-determined status of the FEE means that the heap manager must be built around the timing requirements of the FEE to provide a complete path through which data may pass.

To provide a foundation for the signal and timing requirements posed for the development of the heap manager, a study of the individual ASICs that make up the front-end electronics is required. As dictated by the specifications of the beam test system,

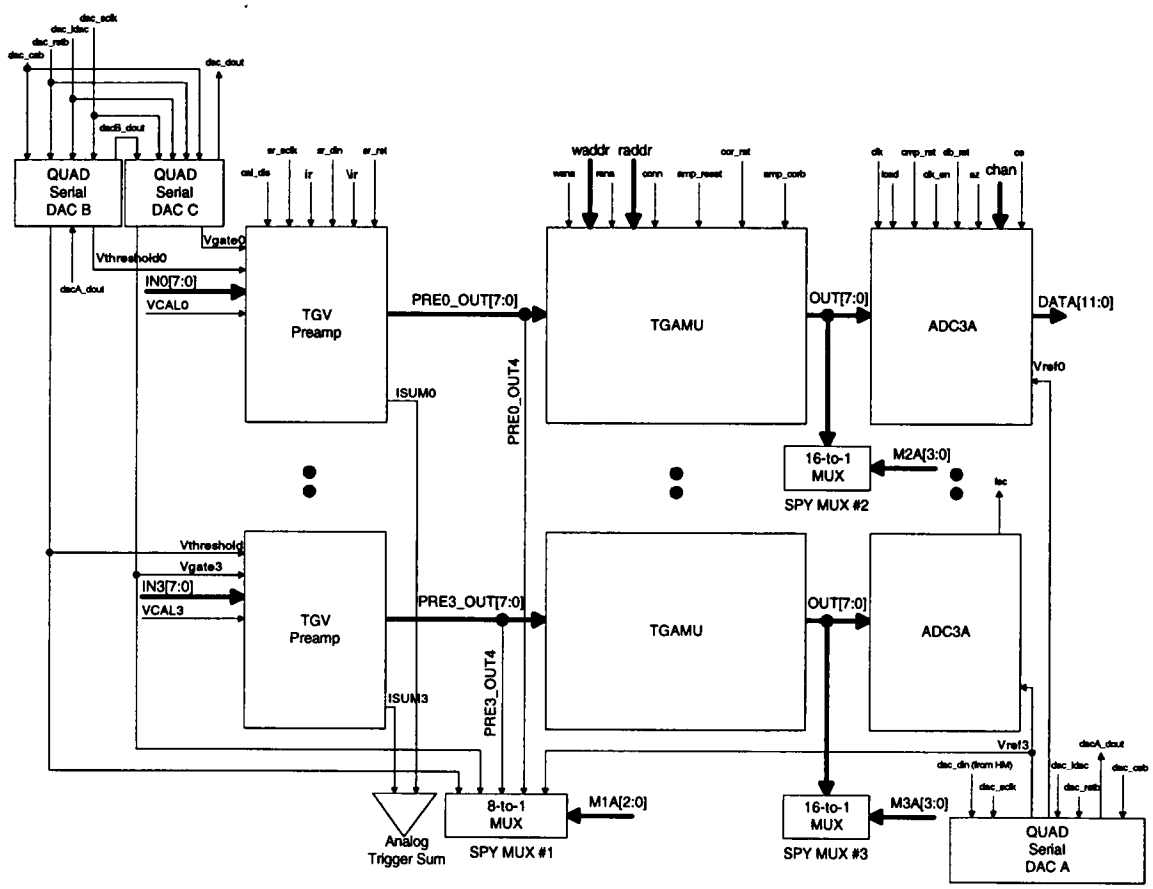


Figure 2.2.1. FEE Block Diagram for the 32-Channel FEE Mother Board

the heap manager provides the impetus for all data transfers in the system; therefore, a complete understanding of the heap manager cannot be accomplished without an analysis of the capabilities of the preamp, analog memory, and ADC.

TGV Preamp

The TGV preamp is a charge-injection amplifier which, when coupled with the detector, is capable of accepting an AC-coupled signal and amplifying it to levels which may be stored in the analog memory. The block diagram for the eight-channel TGV preamp is provided in Figure 2.2.2, and Table 2.2.1 contains a summary of the signals associated with the preamp. Of the critical signals for the preamp, the **Vgate** bias voltage is used to set the gain of the discriminator, and the **Vthreshold** voltage controls the trigger level for each discriminator, which outputs one of the currents used in the **Isum** trigger sum. In the MVD beam test system, eight **Isum** trigger sums, one for each ASIC, determine the existence of a LVL-1 condition. Since the production of LVL-1 triggers in the MVD system relies heavily on the correlation between the gain and threshold of the discriminator and the sensitivity of the preamp to changes in input signals, careful maintenance of the bias voltages for the TGV circuit is required.

For the heap manager, the signals of interest are the resets and the calibration functions. In its role as the link between the detector and the front-end electronics, the preamp continually stores charge on its feedback capacitor. As seen in Figure 2.2.3, charge is accumulated on the output of the preamp with each event. Because added charge will eventually cause the output of the preamp to reach the rail, the preamp must be reset

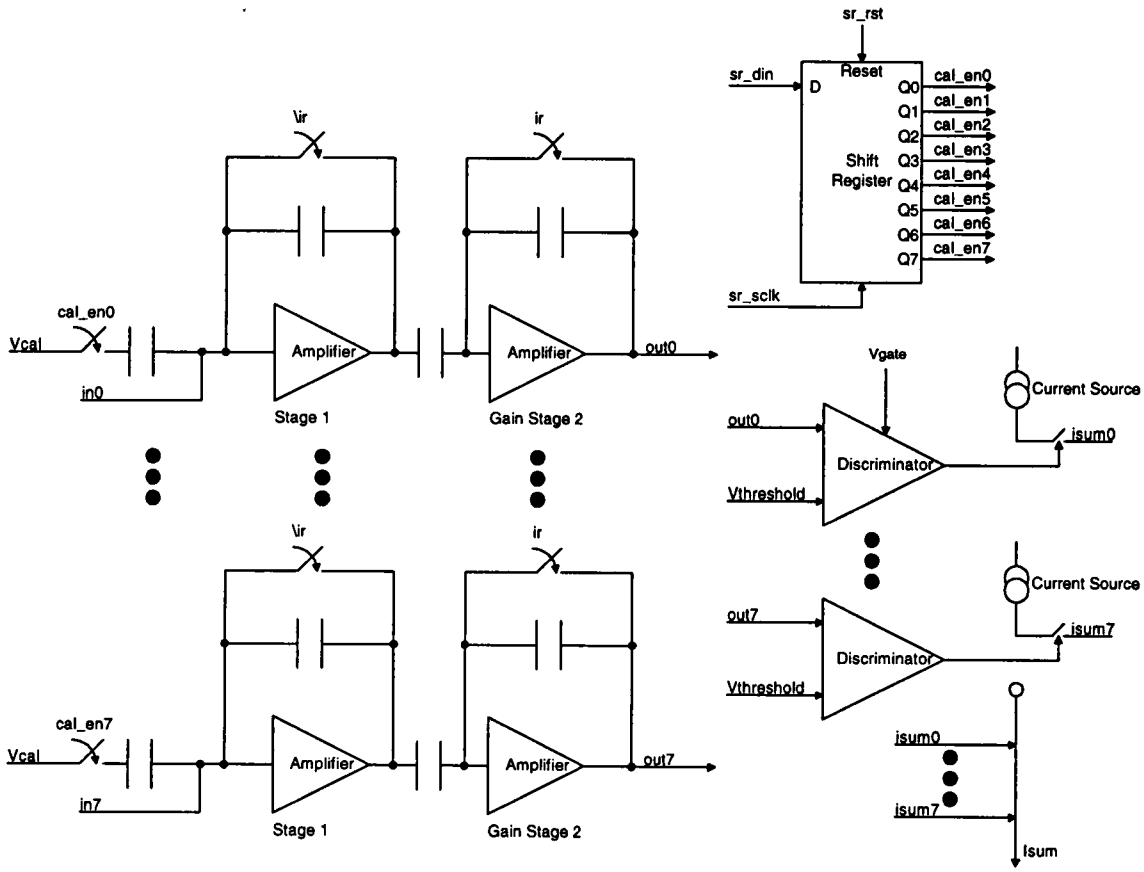


Figure 2.2.2. TGV Preamp Block Diagram

Table 2.2.1. TGV Preamp Signal Descriptions

Pin #	Name	I/O/P	Description
1	OUT4	O	preamp output (one of eight)
2	OUT5	O	preamp output (one of eight)
3	OUT6	O	preamp output (one of eight)
4	OUT7	O	preamp output (one of eight)
5	SR_SCLK	I	clock for serial data (active on rising edge)
6	VDD_DISC	P	+5V power supply for discriminator
7	GND_DISC	P	analog ground for discriminator
8	NC	I	no connection
9	VDD_S1	P	+5V power supply for 1st stage of preamp
10	GND_S1	P	analog ground for 1st stage of preamp
11	GND	P	analog ground (chip ground)
12	VDD	P	+5V analog supply
13	VCAL	I	calibration voltage
14	\IR	I	amplifier reset bar (active high)
15	IR	I	amplifier reset (active low)
16	SR_DIN	I	serial data in for calibration
17	IN7	I	analog input (one of eight)
18	IN6	I	analog input (one of eight)
19	IN5	I	analog input (one of eight)
20	IN4	I	analog input (one of eight)
21	IN3	I	analog input (one of eight)
22	IN2	I	analog input (one of eight)
23	IN1	I	analog input (one of eight)
24	IN0	I	analog input (one of eight)
25	BIAS_PREAMP	I	preamp bias (R to gnd)
26	CAL_DIS	I	calibration discharge
27	BIAS_TGS2	I	second stage preamp bias
28	Vgate	I	discriminator differentiator gate voltage (sets gain)
29	VDD_PRE	P	+5V power supply for preamp
30	GND_PRE	P	analog ground for preamp
31	BIAS_DSCRM	I	discriminator bias (R to 5V)
32	BIAS_SUM	I	current switch bias
33	SR_RST	I	shift register reset (active HIGH)
34	VMID	I	discriminator vmid
35	Vthreshold	I	discriminator threshold voltage (1V to 2V)
36	ISUM	O	current sum output
37	OUT0	O	preamp output (one of eight)
38	OUT1	O	preamp output (one of eight)
39	OUT2	O	preamp output (one of eight)
40	OUT3	O	preamp output (one of eight)

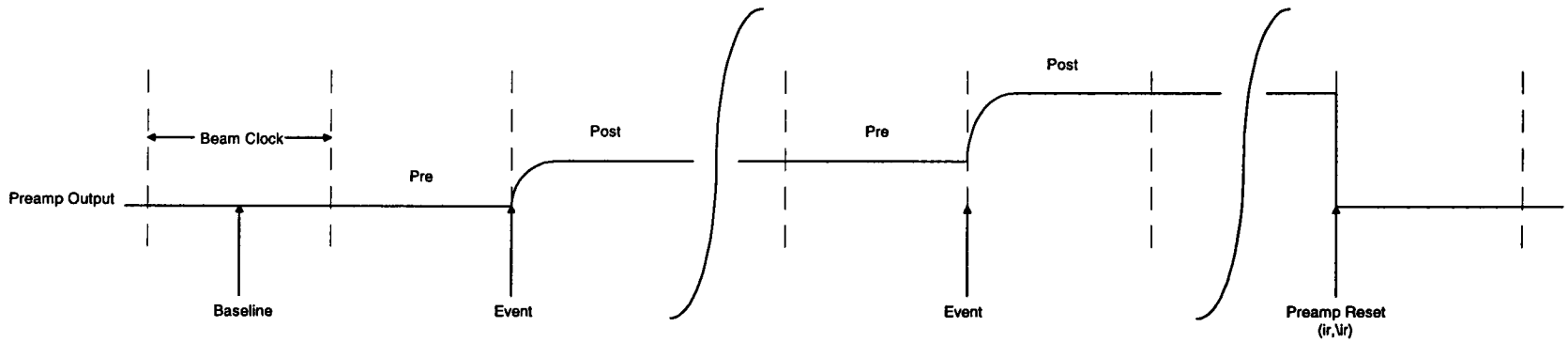


Figure 2.2.3. TGV Preamp Timing

so that the output may return to the initial baseline voltage; therefore, to avoid the overload condition, the resets, **ir** and **\ir**, are employed to short the feedback capacitors and return the preamp output to baseline, thus allowing charge accumulated during normal operation of the circuit to be discharged. The frequency of the preamp reset is determined by the maximum expected rate with which events are predicted to occur. Timing for the integrator reset function is plotted in Figure 2.2.4, and the heap manager implementation must approximate the integrator reset using the **4xclk** timing; therefore, a counter circuit is included in the heap manager to provide the delays required for the operation. Since the beam test system runs continuously from power-up, the integrator reset is enacted periodically to ensure that the preamp output never reaches the rail. Ultimately, the human interface dictates the frequency of the integrator reset function; so the heap manager interfaces include a means by which the integrator reset frequency may be programmed. Because the preamp reset precludes any real analog data from being input into the system while the integrator capacitors are shorted, timing of the reset function must be carefully monitored so that no erroneous data is manufactured during the periods of preamp latency.

In addition to the integrator resets, a calibration circuit is included in the preamp. By shifting in the eight calibration enables with **sr_din** and **sr_sclk,** as shown in Figure 2.2.5, any channel, or channels, may be selected for calibration mode, during which **Vcal** is used as the analog input voltage that is amplified and presented to the output of the preamp; and **cal_dis** supplies a control which discharges the preamplifier calibration circuit. In the beam test, all four preamps on a given FEE board share the **sr_din**, **sr_clk**,

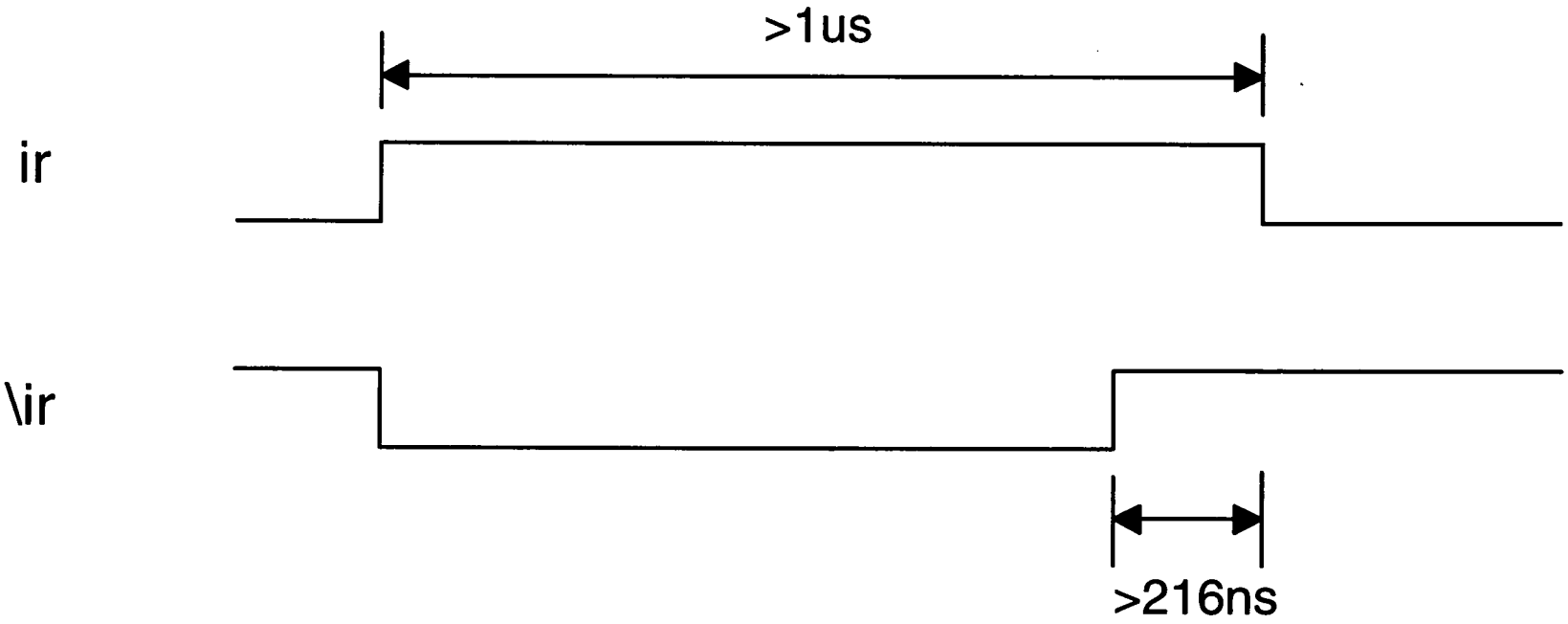


Figure 2.2.4. Integrator Reset Timing for the TGV Preamp

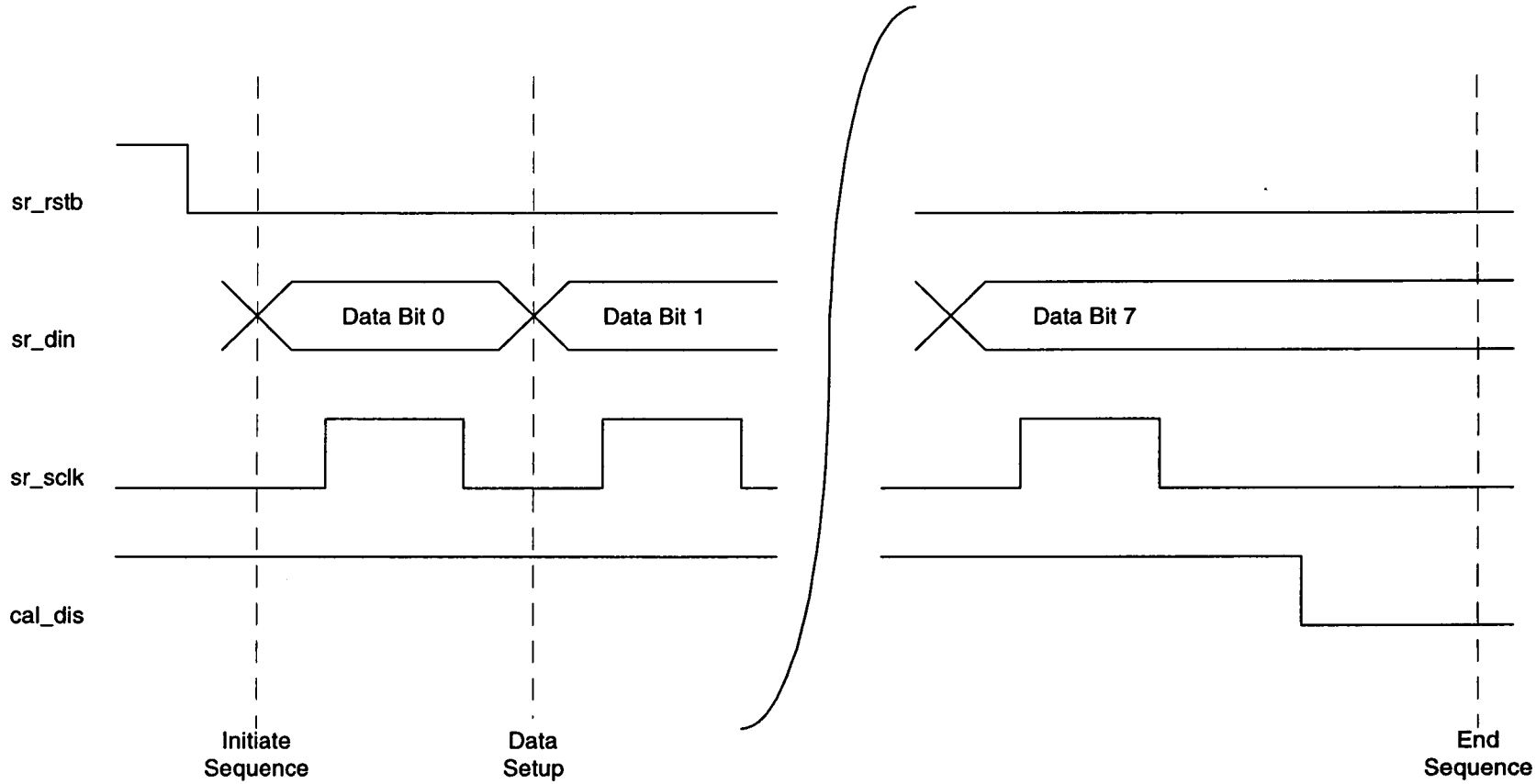


Figure 2.2.5. Calibration Timing for the TGV Preamp

and **sr_reset** lines; therefore, the status of the calibration shift registers is identical for each of the TGV chips on a specific FEE mother board. Since it is utilized only in the absence of a signal on the main analog input, the calibration circuit is useful in examining the input/output relationship of the amplifier as well as the true trigger level of the discriminator and may be utilized as a diagnostic to assure that the preamp is functioning as expected, thus facilitating testing of the beam test system in the absence of the detector.

For MVD, the preamp provides the primary link between the detector and the front-end electronics. Impulses picked up by the silicon strips are amplified by the preamp and stored in the analog memory each beam clock. For it to successfully control the preamp, the heap manager must provide a reliable way of resetting the preamp as well as manipulating the calibration circuitry; therefore, the interface between the computer and the heap manager includes logic which provides programmability for all critical signals of the TGV preamp. The successful retrieval of analog data produced by the preamp is left to the heap manager and the analog memory.

Analog Memory Unit (AMU)

An analog memory is the primary unit for storing analog data in the MVD system. The TGAMU is a sixteen-cell, eight-channel analog memory which can be controlled to act as a delay line in the MVD prototype [5]. Figure 2.2.6 and Table 2.2.2 contain the block diagram and signal definitions, respectively, for the TGAMU chip used in the MVD beam test. From Figure 2.2.6 and Table 2.2.2, the AMU can be seen to be made up of three distinct parts: memory cells, output amplifier, and correlator circuit. Each analog

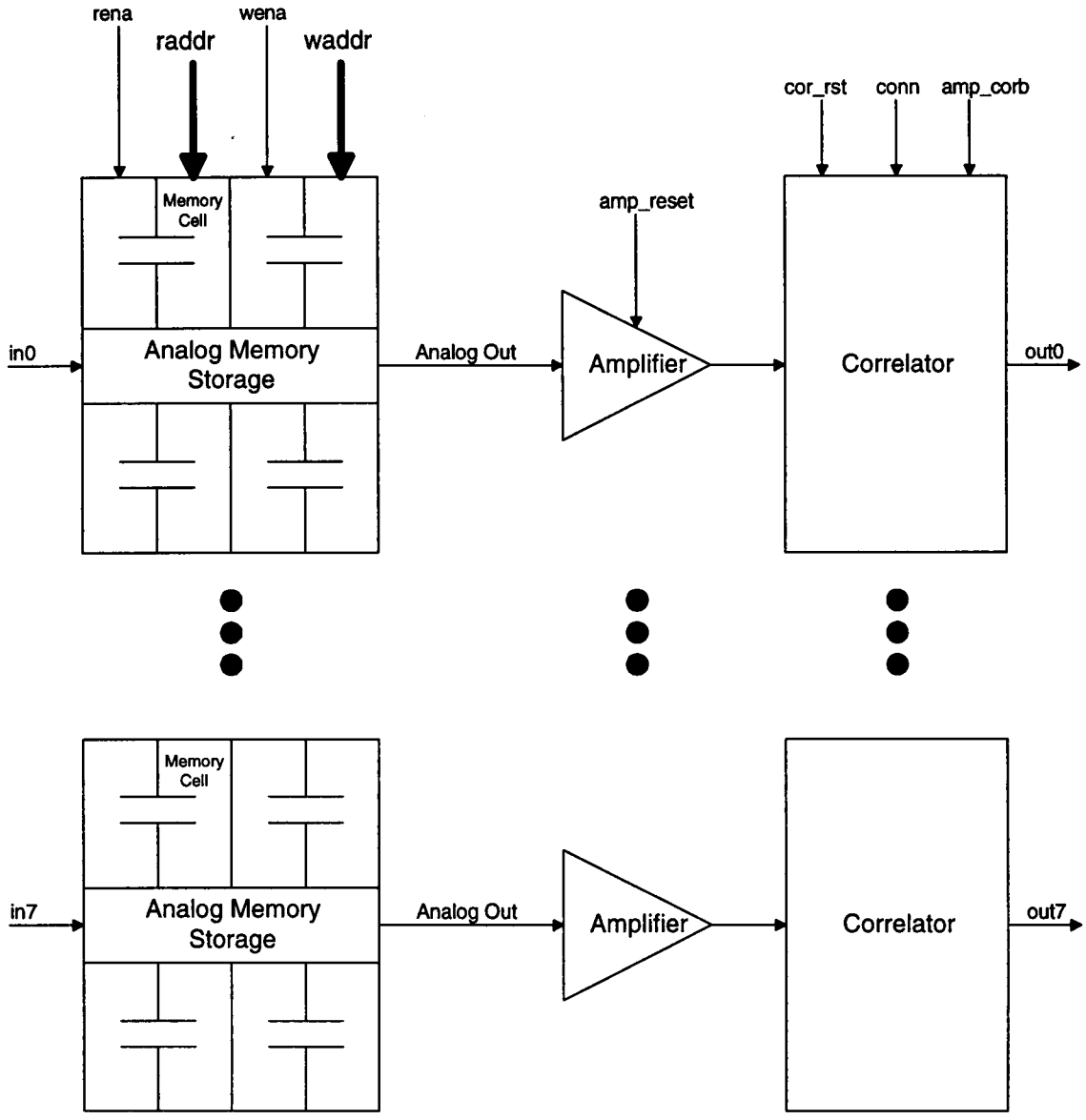


Figure 2.2.6. Block Diagram of the TGAMU Analog Memory

Table 2.2.2. TGAMU Signal Descriptions

Pin #	Name	I/O/P	Description
1	OUT5	O	analog output (6 of 8)
2	OUT6	O	analog output (7 of 8)
3	OUT7	O	analog output (8 of 8)
4	AMPOFF	I	connect to COMMON
5	COR_RST	I	correlator reset (active HIGH)
6	NC		no connection
7	CONN	I	connects correlator output to analog output
8	CORROFF	I	correlator offset
9	AMP_RESET	I	amplifier reset
10	RADDR3	I	read address (4 of 4)
11	RADDR2	I	read address (3 of 4)
12	RADDR1	I	read address (2 of 4)
13	RADDR0	I	read address (1 of 4)
14	RENA	I	read enable
15	GND	P	ground
16	VDD	P	+5V
17	NC		no connection
18	AMP_CORB	I	selects correlator function
19	IN7	I	analog input (8 of 8)
20	IN6	I	analog input (7 of 8)
21	IN5	I	analog input (6 of 8)
22	IN4	I	analog input (5 of 8)
23	COMMON	I	bias (R to 5V)
24	IN3	I	analog input (4 of 8)
25	IN2	I	analog input (3 of 8)
26	IN1	I	analog input (2 of 8)
27	IN0	I	analog input (1 of 8)
28	NC		no connection
29	VDD	P	+5V
30	GND	P	ground
31	WENA	I	write enable
32	WADDR0	I	write address (1 of 4)
33	WADDR1	I	write address (2 of 4)
34	WADDR2	I	write address (3 of 4)
35	WADDR3	I	write address (4 of 4)
36	AMP_VDD	P	amplifier VDD (+5V)
37	AMP_BIAS	I	amplifier bias
38	AMP_VSS	P	amplifier ground
39	NC		no connection
40	OUT0	O	analog output (1 of 8)
41	OUT1	O	analog output (2 of 8)
42	OUT2	O	analog output (3 of 8)
43	OUT3	O	analog output (4 of 8)
44	OUT4	O	analog output (5 of 8)

input runs to a memory cell which is selected by the write address (**waddr**). The memory cell is the analog version of a RAM cell in digital circuits, and each cell consists of a storage capacitor which is charged by the input signal when the write enable (**wena**), which controls an analog switch, is asserted. Similarly, readout of the AMU is accomplished by applying the address of the cell (**raddr**) and raising the read enable (**rena**); and a correlator circuit may be exercised to combine the outputs of two cells into a single output. Although it is not used in the beam test, the correlator, in the final version of the MVD system, is used as an option to raw mode. Whereas raw mode data includes Pre and Post components for each LVL-1, correlator mode data makes use of the correlator circuit, which permits the subtraction of the Pre (baseline) analog voltage from the Post analog value to provide a single difference voltage, thus saving space for data storage and improving system throughput. Finally, for all readout operations, the amplifier reset (**amp_reset**) is utilized in the output amplifier circuit to keep the amplifier inactive until needed, thus precluding effects of charge injection which could manifest itself in the form of a pedestal voltage. By manipulating the signals of the analog memory, the heap manager has the capability to store, temporarily, the outputs of the preamp in the AMU and preserve the data while a decision is made as to the relevancy of the information to the experiment.

The ability of the heap manager to use the AMU as an analog RAM has the effect of producing a delay in the passage of the detector signals through the FEE. Since an instantaneous decision on whether data is valid cannot be made, the AMU allows the analog preamp output to be stored for each beam clock; and, when a valid event is

detected, via the trigger sums (**Isum**) provided by the preamps, the heap manager reads the corresponding cell(s) in the AMU and makes the necessary conversions. The address list manager controls the addressing for the AMU so that a delay of nine beam clocks is available between the time when an analog value is stored and when a decision must be made to preserve the cell for a conversion by the ADC. Consequently, the system is not hampered by having to stop while a decision is made on a specific detector signal. By working together to manage the read and write functions of the AMU, the address list manager and heap manager controller govern a dead-timeless system which continuously circulates the stimuli from the detectors and can shuttle data to an ADC for conversion without affecting the acquisition process.

Analog-to-Digital Converter (ADC)

As the link between the analog AMU and digital storage unit, the analog-to-digital converter in the MVD beam test system is the ADC3A, a Wilkinson-type ADC. The Wilkinson architecture employs a ramp circuit which provides one of two inputs to a comparator, the second input of which is supplied with the analog voltage to be converted. When a conversion is initiated, the ramp begins its descent; and a twelve-bit Gray code counter is enabled. At the instant when the ramp voltage equals the voltage of the analog input, a comparator fires; and the counter value is latched. If the ramp is linear, the value latched by the firing of the comparator is directly proportional to the input analog voltage. Because the architecture takes advantage of a ramp circuit which is common to all channels, the Wilkinson ADC is used in applications where the per-channel

area is to be minimized [6] and is ideal for implementation in systems where power is at a premium.

The timing of ADC3A includes signals which apply to both digital and analog portions of the circuit. In Figure 2.2.7 and Table 2.2.3, the signals applicable to ADC3A, an eight-channel analog-to-digital converter, are shown. At the beginning of a conversion, the auto-zero (**az**) signal initializes the ADC so that all analog values return to baseline; and, subsequently, the comparator reset (**cmp_rst**) is lowered to bring the comparator to a state of readiness. To minimize power consumption, the ADC is left in an inactive state, with all resets active, until a conversion is begun.

Once the ADC is initialized and activated, the analog voltage to be converted is input into the ADC; and, when a conversion is ready to begin, the analog input is switched onto an internal capacitor with the **ics** signal, thus maintaining a steady voltage for the ADC conversion circuit and allowing the external input voltage to change while the conversion is taking place. To start the counter and ramp at exactly the same moment, the clock enable (**clk_en**) signal is utilized as an inverted ramp reset as well as the clock enable for the counter. Since the comparator triggers when the ramp voltage and the voltage on the internal capacitor are the same, the moment of a successful conversion is indeterminate from the view point of the external circuit; therefore, a full-scale count (**fsc**) signal, which goes HIGH when the counter reaches its final count of Hex FFF, is utilized as the end-of-conversion indicator. When the conversion is complete, the twelve-bit digital representation of the voltage is loaded to the output drivers via the **load** signal. At the end of the conversion cycle, the data stored on the latch driven by the **load** impetus is the

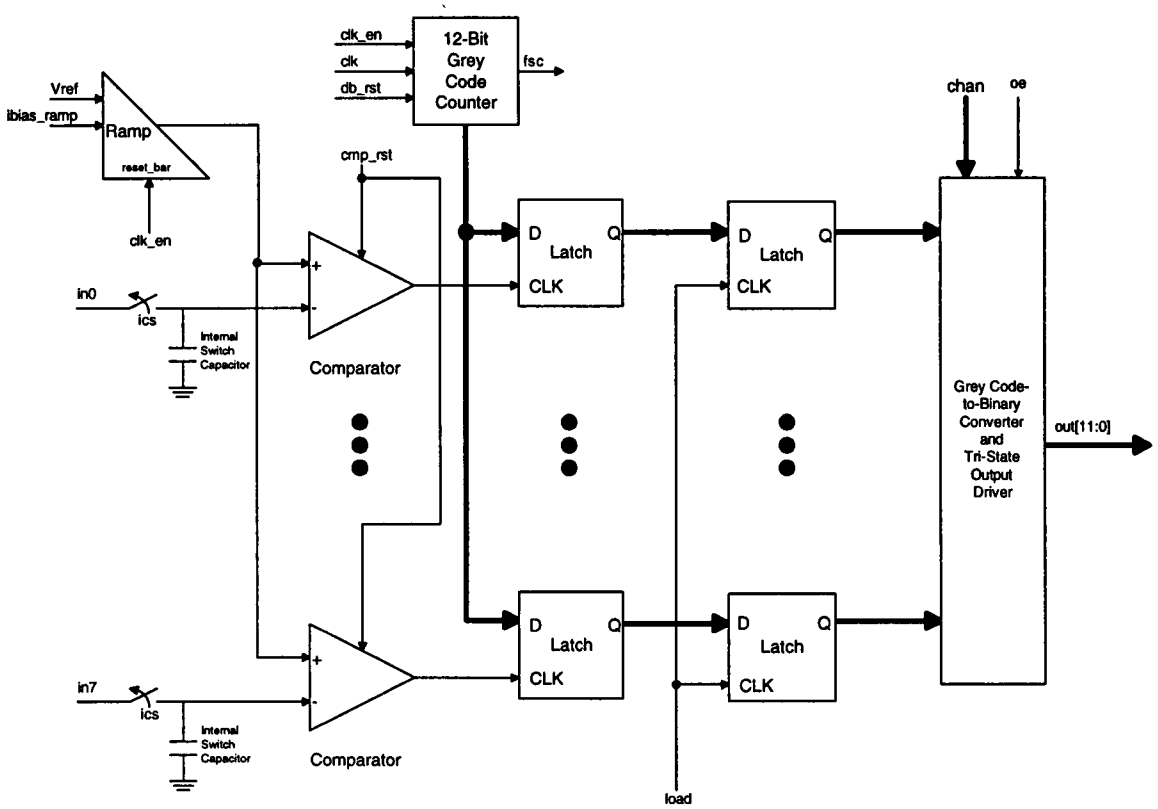


Figure 2.2.7. ADC3A Wilkinson ADC Block Diagram

Table 2.2.3. ADC3A Signal Descriptions

Pin #	Name	I/O/P	Description
1	OUT10	O	output data bit (one of twelve)
2	OUT11	O	output data bit (one of twelve)
3	OE	I	output enable for tri-state output data bits
4	CHAN0	I	adc address bit for channels 0-7
5	CHAN1	I	adc address bit for channels 0-7
6	D_GND	P	digital ground
7	CHAN2	I	adc address bit for channels 0-7
8	CLK-	I	positive end of differential clock input
9	CLK+	I	negative end of differential clock input
10	IBIAS_CLK	I	bias current for clock input buffer
11	CLK_EN	I	digital control signal to enable clock
12	FSC	O	full scale count - triggers when counter reaches HEX FFF
13	SEL_11_BIT	I	shuts down LSB of counter to make adc 11 bits
14	A_VDD	P	+5V analog supply
15	A_GND	P	analog ground
16	RAMP	O	ramp output
17	D_VDD	P	+5V digital supply
18	Iref	I	reference current
19	Vref	I	reference voltage - ramp begins here and ramps to 0V
20	IBIAS_RAMP	I	ramp bias (adjusts slope of ramp)
21	IN7	I	channel 7 analog input
22	IN6	I	channel 6 analog input
23	IN5	I	channel 5 analog input
24	IN4	I	channel 4 analog input
25	IN3	I	channel 3 analog input
26	IN1	I	channel 2 analog input
27	IN0	I	channel 1 analog input
28	D_GND	P	digital ground
29	ICS	I	channel 0 analog input
30	AZ	I	auto-zero
31	CMP_RST	I	comparator reset
32	DB_RST	I	debounce reset
33	LOAD	I	load - used to latch data into output buffer
34	OUT0	O	output data bit (one of twelve)
35	OUT1	O	output data bit (one of twelve)
36	OUT2	O	output data bit (one of twelve)
37	OUT3	O	output data bit (one of twelve)
38	OUT4	O	output data bit (one of twelve)
39	D_VDD	P	+5V digital supply
40	OUT5	O	output data bit (one of twelve)
41	OUT6	O	output data bit (one of twelve)
42	OUT7	O	output data bit (one of twelve)
43	OUT8	O	output data bit (one of twelve)
44	OUT9	O	output data bit (one of twelve)

Gray-coded representation of the input voltage.

For the conversion, a Gray code counter is used so that only one bit of the counter is changing at a time, thus precluding glitches which may result if the counter is making a transition when the comparator fires. For output, a Gray-to-binary converter is combined with tri-stated output drivers to return data to a usable binary stream. Hence, binary, not Gray, encoded data is read from the output buffer by addressing the ADC with the channel (**chan**) bus and by asserting the output enable (**oe**) for the tri-state buffers. Since the outputs are tri-stated, several ADCs may be used to drive the same data bus and preclude the need for external tri-state drivers in the beam test system and, in the future, in the multi-chip module. Because it cannot distinguish between the data that may be present on any given ADC channel and correlate specific data with the reception of a LVL-1 trigger, the heap manager, after a conversion cycle, enables each output of each ADC, one at a time, and packages the twelve-bit words for storage in the CPU.

Because the analog data from the detectors is circulated continuously in the preamp and AMU portion of the electronics, the throughput of the system is determined by the time required to produce digital data from the analog values stored in the AMU. From the moment at which a LVL-1 trigger is issued to the time when data is returned to the CAMAC or PC FIFO, the delay is measured in terms of the ability of the heap manager to read the AMU, initialize the ADC, and address the ADC for converted data. The throughput of the MVD beam test prototype is proportional to the time needed by the address list manager to read the AMU, the time required by the ADC state machine in the heap manager controller, the conversion time of the ADC, which is set by the frequency of

the clock applied to the ADC counter circuit, and the time needed by the heap manager controller to read the digitized data from the ADC and return a packet of information to the CAMAC or PC FIFO.

Finally, a successful ADC conversion relies heavily on the ramp circuit. Not only must the ramp be linear, but also the slope and range of the ramp must coincide with the dynamic range of the data to be represented. Thus, the ramp circuit is provided with **Ibias**, the current used to charge the capacitor in the ramp circuit, and **Vref**, the voltage which sets the upper bound of the ramp and at which point the ramp begins its descent. For the beam test prototype, **Ibias** is set on the FEE mother board by means of a potentiometer, and **Vref** is provided by a programmable serial DAC which is controlled by the heap manager via a serial interface. The ability to control the ramp and bias current is imperative since minute deviations in the circuits caused by imperfections in the silicon process are reflected in variances in the performance of the ramp and, explicitly, in the data produced by the ADC.

Serial Digital-to-Analog Converters (DACs)

In the beam test prototype, several key bias voltages for the FEE are regulated via digital-to-analog converters to allow changes in the system parameters and facilitate calibration of the system. Two quad serial DACs are utilized to permit the setup of specific gate (**Vgate**) and threshold (**Vthreshold**) voltages for the preamp discriminator as well as **Vref** for the ADC. A single serial string is used to set up all three DACs, and the heap manager provides an interface with the CPU to permit the programming of the

voltages to be implemented through the DACs. From the heap manager, the serial data line is fed to DAC A, as seen in Figure 2.2.1; and each subsequent DAC receives its data via the **dac_dout** line from the previous DAC. Since each chip contains four DACs, all four **Vthreshold**, **Vgate**, and **Vref** voltages for the individual ASICs on a given FEE mother board are independently controllable.

The serial string for each DAC is made up of twelve bits, eight bits to set the voltage and four control bits. A graphical representation of the DAC register is presented in Figure 2.2.8, and Table 2.2.4 defines the commands used in the MVD system. Each time an update to one or more of the DACs is initiated, the DAC is selected by lowering the chip select (**dac_csb**); and a serial string of thirty-six bits is issued from the CPU, as indicated by the timing evinced in Figure 2.2.9. The command bits allow any or all DACs to be set for a specific voltage within eight-bit accuracy. To set a particular DAC channel or channels, the serial string is embedded with the eight-bit digital representation of the data as well as the four command bits: A1, A2, C1, and C2; and the data is shifted into the DACs via the data line, **dac_din**, and clock line, **dac_sclk**. A successful update to the DACs is initiated when the **dac_ldacb**, or load line, is pulsed. Since the **dac_din**, **dac_sclk**, **dac_csb**, and **dac_ldacb** lines are shared by all DACs on a given FEE board, each update includes all three DACs; therefore, the inclusion of the NOOP command provides a means by which any DAC may be bypassed when an update is made to another digital-to-analog converter chip. Since it contains an interface by which the serial string may be manipulated, the heap manager allows discrete control over the bias voltages in the beam test system.

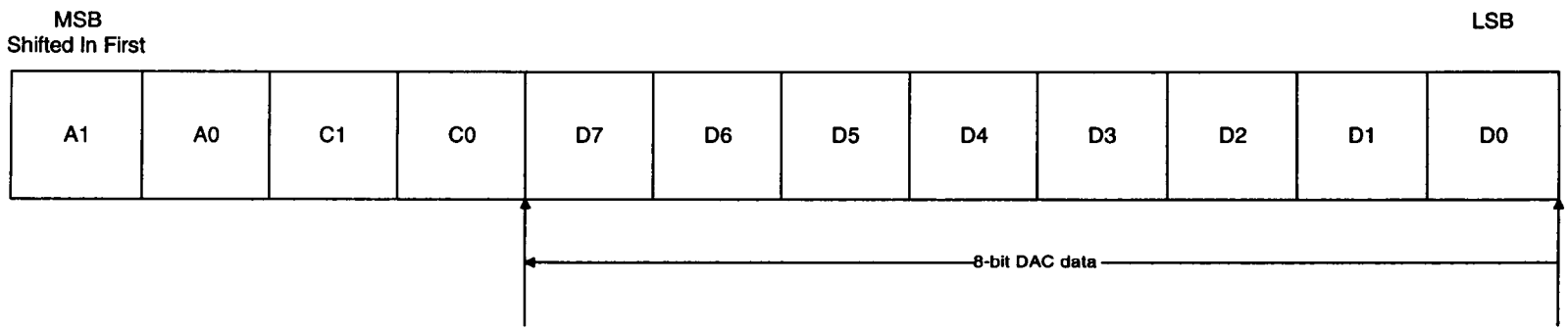


Figure 2.2.8. DAC Register Structure

Table 2.2.4. DAC Serial Commands

A1 A0	C1 C0	Description
Address	1 1	change single channel
X 1	0 0	NOOP (no operation)
X 0	0 0	update all DAC channels

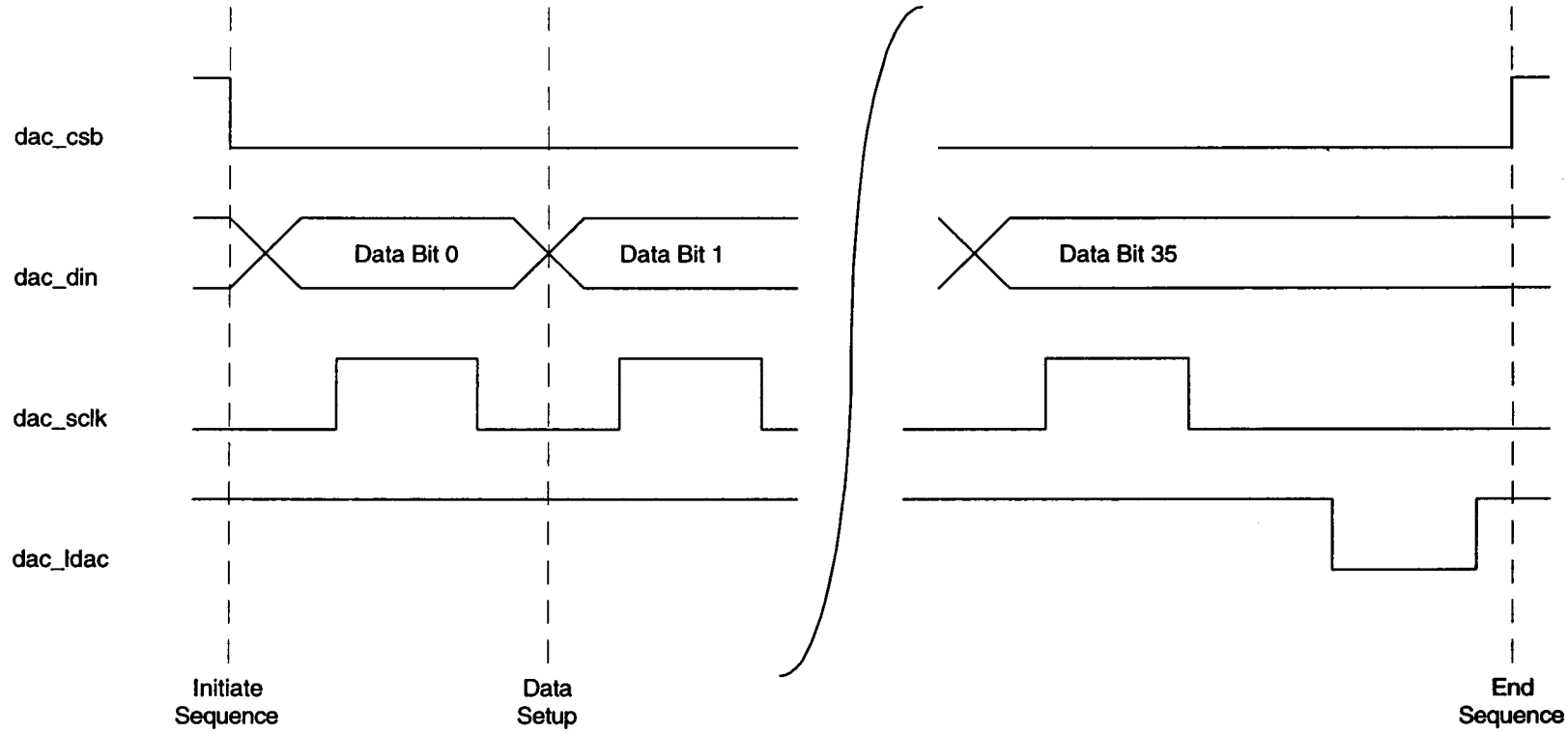


Figure 2.2.9. DAC Serial String Timing

SPY Multiplexers

The SPY Multiplexers provide test points for certain voltages from the ASICs on the FEE board. Because so many different components, each of a sensitive nature, are included on the FEE mother board, a means of sensing critical signals is needed. The SPY multiplexers are conceived to make it easy to view three critical signals at a time by addressing the multiplexers. One 8-to-1 and two 16-to-1 multiplexers are available for use as SPYs. Control of the multiplexers is left to the heap manager, which accepts inputs from an interface with the CPU and converts the commands into addresses for the multiplexers. The outputs of the SPY multiplexers are channeled to test points designed for oscilloscope probes so that the buffered signals may be monitored during system operation.

2.3. Heap Manager PC FIFO

Although not a part of an FPGA, the PC FIFO, an ITD72205 4096x18 CMOS SyncFIFO, is on the heap manager board to provide a temporary storage unit for data from the heap manager controller. Obviously, the FIFO is needed for reasons of resource management since the FPGAs are not capable of supplying the number of RAM cells needed for such a large memory; and, accounting for the disparity between the ability of the heap manager to produce data and the relative slowness of the PC, the amount of data

which the system produces requires some means of buffering to maintain highest possible system throughput.

The layout of the PC FIFO is simplified to maximize speed, yet includes enough input and output signals to make the FIFO easy to use while protecting data from accidental overwrites. A block diagram of the FIFO is presented in Figure 2.3.1, and the definitions of the relevant signals for the FIFO are provided in Table 2.3.1. For the SyncFIFO, a 4096x18 RAM provides the main digital memory; and the incoming and outgoing data is received and transmitted via eighteen-bit parallel buses [7]. A read and write pointer keep track of the read and write addresses, respectively; and data is clocked into and out of the FIFO with the **pcfifo_wclk** and **pcfifo_rclk** signals, respectively. To preclude erroneous reads or writes from occurring with glitches on the read clock or write clock lines, a pair of enables, **pcfifo_r** for the read pointer and **pcfifo_w** for the write circuit, must be driven HIGH prior to the enactment of the corresponding operation [7]. Thus, the integrity of the FIFO is preserved as long as the read enable and write enable are inactive. Furthermore, since the address pointers of the FIFO are unavailable to an external circuit, flags are provided to show the current status of the memory in relation to the amount of data stored. The flags include the standard full flag (**\ff**) and empty flag (**\ef**), as well as a half full flag (**\hf**), which becomes active when 2048 words of data are stored in the memory. Also, the almost full flag (**\paf**) and almost empty flag (**\pae**) may be checked to determine whether the write pointer and read pointer are separated by thirty-one addresses [7]. The interdependence of the flag logic with the read and write operations of the FIFO allows independent control of the data buffer from the perspectives

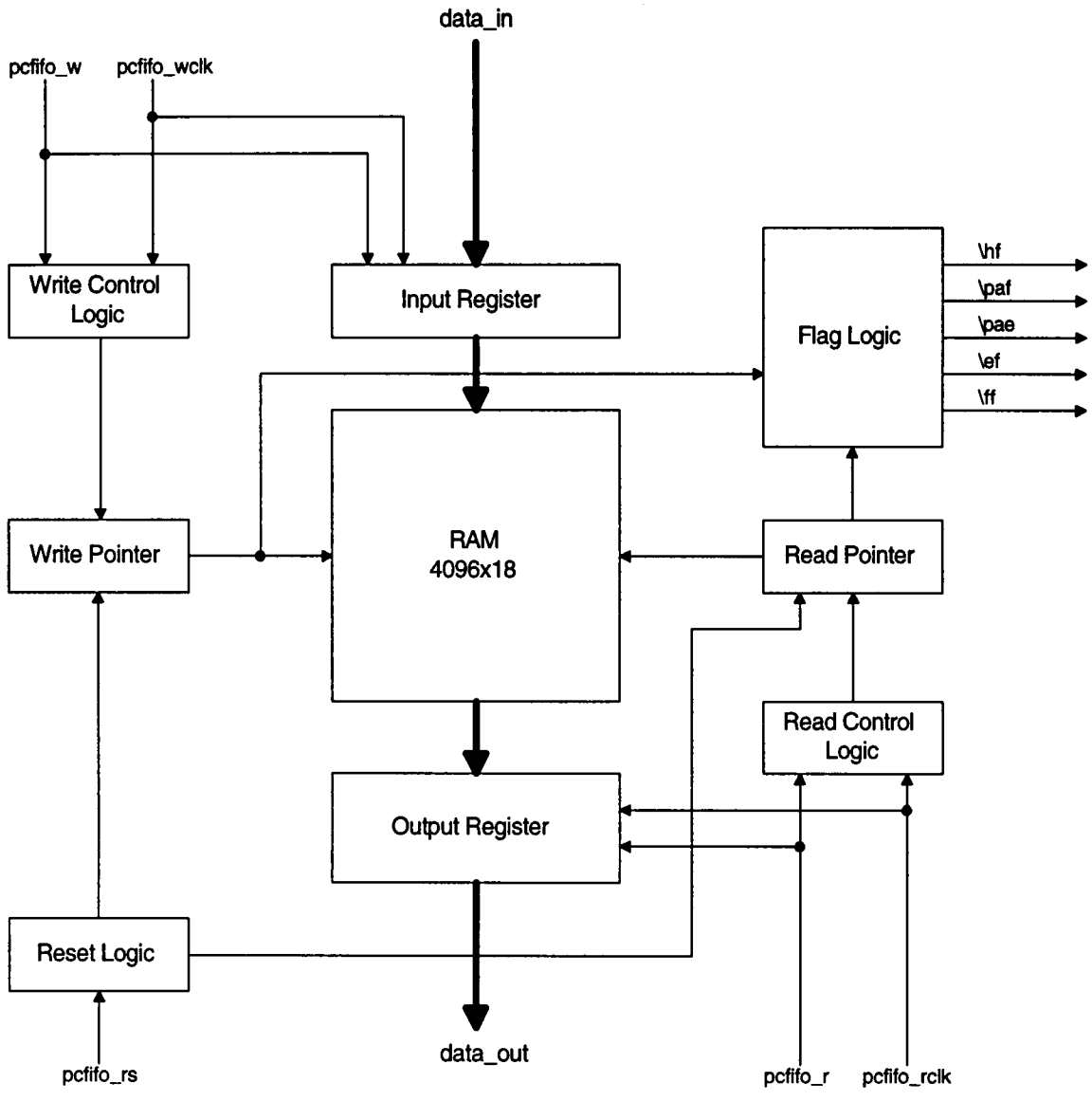


Figure 2.3.1. Block Diagram of the ITD72205 4096x18 SyncFIFO

Table 2.3.1. Signal Definitions for the PC FIFO

Signal Name	Type	Controller	Description
pcfifo_w	I	HM	write enable (active LOW)
pcfifo_wclk	I	HM	write clock (active HIGH)
pcfifo_r	I	CPU	read enable (active LOW)
pcfifo_rclk	I	CPU	read clock (active HIGH)
\hf	O	CPU	half full flag (active LOW)
\ff	O	CPU	full flag (active LOW)
\paf	O	CPU	almost full flag (active LOW) - 31 RAM locations available
\ef	O	CPU	empty flag (active LOW)
\pae	O	CPU	almost empty flag (active LOW) - 31 RAM locations filled
data_in	I	HM	input data
data_out	O	CPU	output data
pcfifo_rs	I	CPU	FIFO reset (active LOW) - resets read and write pointers to 0

of the PC and heap manager and facilitates maximum throughput, which is influenced only by the rate at which the heap manager and ADC can process data.

Because the MVD front-end module can work much faster and store data at higher clock speeds than the computer interface can accept them, the PC FIFO is used as a buffer between the heap manager and the PC interface. The heap manager controls the write enable (**pcfifo_w**) and write clock (**pcfifo_wclk**) for the FIFO; and the PC reads the flag logic signals and manipulates the read enable (**pcfifo_r**) and read clock (**pcfifo_rclk**) as well as the FIFO reset (**pcfifo_rs**). In the beam test system, the PC FIFO is used only for the PC interface, since the CAMAC interface contains its own input data buffer which allows data to be written directly to the CAMAC via the **WSI** clock. In terms of the architecture for the heap manager and the data readout function of the PC, the PC FIFO relaxes the resource requirements pertaining to logic allocation and routing in the FPGA and permits the PC to run at a much slower rate than the heap manager without affecting system throughput.

2.4. FEM Interfaces

The interfaces for the front-end module are comprised of control, timing, readout, and programming functions, which are implemented in the heap manager. To determine the number of resources needed by the heap manager to implement the necessary functions, the interfaces are developed and separated by functions. An analysis of the

number of I/O and estimated gate count required yields a better understanding of the FPGAs needed by the design.

Obviously, the primary concern for the FPGAs is programmability. For Xilinx parts, four signals are of supreme import. The programming signals used for the Xilinx family are tabulated in Table 2.4.1. The **sdin** and **sclk** signals are used to clock the configuration data into the FPGA, and the **prog** signal is used to clear the configuration memory. Upon completion of a configuration cycle, the **done** bit may be used to determine whether the FPGA is configured correctly. Since two FPGAs are utilized for the heap manager, two sets (or eight) serial lines are needed to perform the programming function.

The control and timing interface of the heap manager consists of the system clocks, the LVL-1 trigger, and five mode bits used to determine the overall functioning of the heap manager controller and the manner in which the heap manager collects and processes data. Of the elements of Table 2.4.2, which holds the definitions of the signals that make up the control and timing interface, the beam clock and the fast clock (**4xclk**) are brought in from external clock sources and are used by the controller and ALM to synchronize functions with the detector and FEE; and the LVL-1 accept is used to indicate that an interesting physics event has occurred and data is to be generated. The system requirements specify that AMU addressing must be congruent with the beam clock to ensure that data can be retraced to a specific time interval in relation to the events of the PHENIX system, but the **4xclk** may be used to coordinate functions that must occur at a faster rate. The relationship between the **4xclk** and the beam clock is such that any

Table 2.4.1. Programming Interface for the Xilinx Family

Signal Name	Type	Description
din	Input	Serial Data Input
cclk	Input	Serial Clock
prog	Input	FPGA Configuration Reset
done	Output	FPGA Configuration Done Bit

Table 2.4.2. Control and Timing Interface for the Heap Manager

Pin Name	Type	Description
Beam Clock (BC)	Input	9.5 MHz Beam Clock Input
4xclk (4xBC)	Input	38 MHz Clock Input
LVL-1	Input	LVL-1 Accept
Mode 4	Input	Mode Control Bit 4
Mode 3	Input	Mode Control Bit 3
Mode 2	Input	Mode Control Bit 2
Mode 1	Input	Mode Control Bit 1
Mode 0	Input	Mode Control Bit 0

operation synchronized with the beam clock is also synchronized with the **4xclk**. Because the I/O needed by the heap manager to implement all of its programmable functions in the form of parallel interfaces is prohibitive, the heap manager is designed so that only the functions which necessitate minimum response time are provided with a parallel command bus. The single command bus in the control interface is made up of five mode bits which are synchronized with the beam clock. Consequently, the mode bits are utilized to define a subset of commands which may be used to specify operations that are to be implemented as quickly as possible. Of the command interfaces, the control and timing interface defines the signals which are most fundamental to the operation of the heap manager.

In addition to the control and timing interface, a series of three serial functions are included in the serial interface of the heap manager. The serial interface requirements for the heap manager controller are displayed in Table 2.4.3. Since two FEE boards of thirty-two channels each are to be used in the beam test, two sets of DAC channels and FEE serial data lines are needed to provide independent control of the FEE mother boards and CPU access to the digital-to-analog converters and calibration circuit of the preamps, respectively. Furthermore, for the operation of the heap manager, secondary functions not specified by the mode bits in the control and timing interface are programmed with a serial string which is shifted into the heap manager via the **hm_sclk**, **hm_din**, and **hm_sloadb** signals. Because the secondary functions are mutable and are set up before the system is put into operation, the latency required for the shifting process does not influence the speed of the heap manager; therefore, the heap manager serial string may be as long as needed since the length of the string does not affect the I/O allocation of the heap

Table 2.4.3. Serial Interface for the Heap Manager

Signal Name	Type	Description
dac1_csb	Input	FEE Board 1 - DAC Chip Select - Active Low
dac1_ldacb	Input	FEE Board 1 - DAC Serial Load Control - Active Low
dac1_din	Input	FEE Board 1 - DAC Serial Data Input - Rising Edge Clocked
dac1_sclk	Input	FEE Board 1 - DAC Serial Clock
dac1_dout	Output	FEE Board 1 - DAC Serial Data Out - Falling Edge Clocked
dac1_rstb	Input	FEE Board 1 - DAC Serial Reset - Active Low
sr1_rst	Input	FEE Board 1 - Preamp Calibration Serial Reset - Active High
sr1_din	Input	FEE Board 1 - Preamp Calibration Serial Data Input
sr1_sclk	Input	FEE Board 1 - Preamp Calibration Serial Clock
dac2_csb	Input	FEE Board 2 - DAC Chip Select - Active Low
dac2_ldacb	Input	FEE Board 2 - DAC Serial Load Control - Active Low
dac2_din	Input	FEE Board 2 - DAC Serial Data Input - Rising Edge Clocked
dac2_sclk	Input	FEE Board 2 - DAC Serial Clock
dac2_dout	Output	FEE Board 2 - DAC Serial Data Out - Falling Edge Clocked
dac2_rstb	Input	FEE Board 2 - DAC Serial Reset - Active Low
sr2_rst	Input	FEE Board 2 - Preamp Calibration Serial Reset - Active High
sr2_din	Input	FEE Board 2 - Preamp Calibration Data Input
sr2_sclk	Input	FEE Board 2 - Preamp Calibration Serial Clock
hm_sloadb	Input	HM Serial Load - Active Low
hm_din	Input	HM Serial Data Input - LSB shifted in first
hm_sclk	Input	HM Serial Clock
hm_dout	Output	HM Serial Data Out - Taken from LSB of Shift Register

manager. Although they cannot be combined, the DAC, calibration, and heap manager serial strings relieve some of the I/O requirements of the heap manager while providing the functionality necessitated by the beam test prototype specifications.

Finally, the data collection module (DCM) provides all data readout operations for the heap manager. Whether a computer or the CAMAC interface is used, the data is sent out in parallel, twelve bits at a time; and signals for the control of the PC FIFO (computer case) or the CAMAC data latch are needed. Of the DCM interface requirements, shown in Table 2.4.4, the **WSI** signal is the latch clock for the CAMAC module; and the **pcfifo_w** and **pcfifo_wclk** lines are for the on-board FIFO buffer from which a PC may read data. Although the data read from the ADC is only twelve bits wide, a sixteen-bit data bus is implemented for the readout to enhance the robustness of the function and provide extra bits for the header and trailer words of the data packet, thus creating a distinction between data and the words used as packet borders. Because all signals for the DCM are exercised during a data packet readout operation regardless of the destination of the data, changes are made to the heap manager printed circuit board to dictate whether the data is to be sent to the PC or CAMAC module; and, since the data acquisition function of the CAMAC module is predetermined, the I/O requirements cannot be eased by sending the data out in the form of a serial string.

Along with the DCM interface, the programming interface, control and timing interface, and serial interface of the heap manager define the I/O required by the heap manager to interact with the external CPU. Thus, the partitioning of the heap manager hinges on the combination of I/O requirements and gates needed to fulfill the functionality

Table 2.4.4. DCM Interface for the Heap Manager

Signal Name	Signal Type	Description
Packet Data[15..0]	Output - Differential ECL or TTL	16-Bit ECL Data Bus
WSI	Output - Differential ECL	Write Strobe For CAMAC 4302 Module
pcfifo_w	Output - TTL	Write Enable for PC FIFO
pcfifo_wclk	Output - TTL	Write Clock for PC FIFO

requirements of the beam test system. By defining the CPU interfaces to be implemented by the heap manager and realizing the signal constraints applied to the front-end electronics, the heap manager may be divided into two optimized designs which will maximize the efficiency with which resources are allocated.

2.5. Heap Manager Partitioning

Because it supplies all digital functionality for the MVD beam test system, the heap manager must have access to sufficient I/O and logic resources to control the front-end electronics and service the CPU. Made up of two FPGAs, one for the address list manager and one for the heap manager controller, the heap manager is included on a printed circuit board which contains the PC FIFO. The decision to divide the FPGA resources into two parts derives from the conclusion that an optimized design is best served by an address list manager which is self-contained and the Xilinx family of FPGAs provides the floorplan which is most conducive to the heap manager design; and a study of the I/O, gate count, and speed specifications for the various Xilinx parts available suggests that two FPGAs, with differing characteristics, might elicit the best response from the heap manager design in the MVD beam test environment.

The development of a cohesive heap manager design depends on the successful integration of the address list manager with the heap manager controller. The layout of the heap manager, revealed in Figure 2.5.1, relies on the relationship of the individual

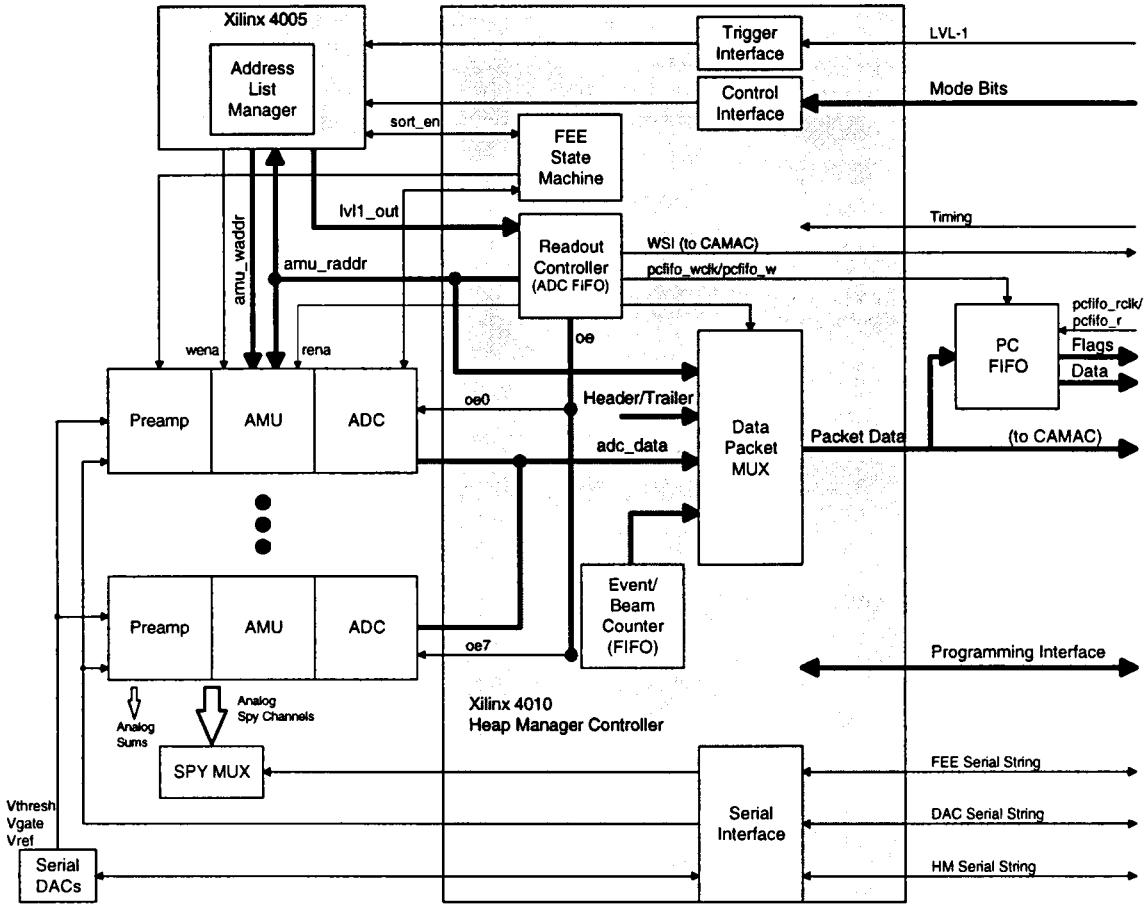


Figure 2.5.1. Block Diagram of the Partitioned Heap Manager

components and the critical signal paths between the digital components, PC FIFO, and front-end electronics. A study of the speed constraints dictated by the addressing function for the AMU indicates that a single FPGA should be allocated expressly for use in setting up an address loop for the sixteen-cell analog memory. Known as the address list manager, the logic which sustains the address loop for the AMU is responsible for the storage and retrieval of addresses in the presence of a LVL-1 trigger. Because the system is acquiring data each beam clock, the address list manager must provide a write address (**amu_waddr**) and write enable (**wena**) each beam cycle; therefore, the ALM is required to run continuously while the system is in operation. When a significant event is realized and a LVL-1 is issued, the ALM removes an address and stores it in an internal FIFO (ADC FIFO), thus preserving the analog value stored in the analog memory location until it can be read from the AMU, converted by an ADC, and passed to the external CPU. The fact that the beam clock frequency is immutable dictates that the address list manager must be capable of maintaining strict logic integrity throughout the operation of the beam test system. In light of the large I/O requirement of the heap manager interfaces, the decision to place the address list manager alone in a single FPGA relieves some of the routing problems accrued during optimization of an FPGA that has limited resources and a static floorplan and thus favors the speed prospects of a self-contained ALM design.

Handling the state machine and interface portions of the heap manager, the heap manager controller works cooperatively with the ALM to provide support functions and control of the FEE. All data must eventually pass through the HM controller, and the PC or CAMAC interface allows variations in the manner in which the electronics are utilized

to collect data. As data is released to the PC FIFO from the HM controller, the PC drains the data and stores the packets in a file. When a LVL-1 is sensed by the system, the heap manager controller begins a conversion by reading the ADC FIFO contained in the address list manager to produce a read address (**amu_raddr**) for the AMU and asserts the AMU read enable (**rena**). After the ADC is finished converting the analog value from the AMU, the heap manager controller reads out the digitized value from the ADC and stores a formatted data packet in the PC FIFO for the computer, or else writes the packet into the CAMAC.

Because it contains the logic necessary for the CPU interface in addition to the logic for the data acquisition function, the heap manager controller commands more I/O and digital resources than the address list manager and is relegated to an FPGA which has significantly more I/O and logic blocks than the FPGA for the ALM. The benefit of placing the heap manager controller in an FPGA separate from the ALM is that, although it is more complex than the address list manager, the heap manager controller does not face the same speed constraints as the ALM; and, by relieving the I/O and logic burden on the ALM FPGA, the heap manager controller assumes some of the routing difficulties which could threaten the performance of the AMU write operation. To further alleviate the demands on the ALM, the ADC FIFO, which stores the addresses removed from the AMU address loop by the LVL-1 trigger and operates synchronously with the slower beam clock, is placed in the same FPGA as the heap manager controller; therefore, although the translation of the ADC FIFO does not affect its functionality, the ALM FPGA attains more flexibility in routing and logic placement. The concessions made in the

proposal of the two-FPGA heap manager culminate in the foundation of a cooperative design which manifests itself in the form of a system controller and high performance address list manager.

Due to the division of the heap manager into two separate FPGAs and the removal of the ADC FIFO into the controller FPGA, a symbiotic relationship exists between the address list manager and heap manager controller. Thus, the controller and ALM utilize flags and mutual address busses to maintain the AMU functions and facilitate re-sorting of AMU addresses after a conversion is completed. In Figure 2.5.1, the **lvl1_out** bus denotes the addresses which are circulated in the ALM during normal operation of the beam test system. With the advent of a LVL-1 trigger, which is also communicated to the ALM, the controller utilizes the **lvl1_out** address bus from the ALM to store the corresponding address in the ADC FIFO while the ALM removes the address from the loop. Subsequently, because it is the component which implements the readout function, the heap manager controller institutes a sort enable (**sort_en**) signal to tell the ALM when to re-sort the address pulled from the ADC FIFO (**amu_raddr**); and the address list manager resets the **sort_en** when the re-sorting operation is complete. By sharing the LVL-1 and addressing information and by synchronizing the readout function with the re-sort operation, the heap manager controller and address list manager maintain system operation from separate FPGAs.

To fulfill the vision of the symbiotic heap manager designs, the FPGA technology selected for the heap manager is the Xilinx 4000 series. Not only does it provide RAM cells for the creation of FIFOs, but the Xilinx 4000 family also has excellent resources for

in-circuit re-programmability. Due to the large number of I/O needed by the heap manager controller, a Xilinx 4010, 10000-gate, 191-pin pin grid array part is chosen to fulfill the primary state machine and interface management requirements. A smaller, 5000-gate, 84-pin PLCC part is chosen as the platform for the address list manager. With the decision to implement the heap manager in the Xilinx technology, the success of the heap manager and allocation of logic and I/O resources for the two FPGAs hinge on the ability of the designer and Xilinx software to optimize the architecture in light of the worst-case gate delays which are pre-determined by the Xilinx silicon process.

The combination of factors considered in the implementation of the heap manager points to the molding of a cooperative design in which two FPGAs are used to work in tandem to carry out the addressing and state machine requirements entailed in the MVD beam system specifications. From a review of I/O and logic demands, separation of the heap manager promises to be the best method to preserve functional integrity while meeting the speed issues that arise from the addressing expectations placed on the address list manager. Consequently, the selection of Xilinx as the FPGA technology takes advantage of imbedded RAM cells and in-circuit programmable logic to provide the necessary resources and I/O for the FEE and CPU interfaces. Because the system requirements are matched to a well conceived methodology and a platform conducive to the structuring of the design, the heap manager layout is begun with a firm foundation on which to construct the design.

Chapter 3

Heap Manager Implementation

The implementation of the heap manager requires two independent designs: address list manager and heap manager controller. The address list manager is designed for a Xilinx 4005, 5000-gate, 84-pin PLCC part; and the heap manager controller is designed for a Xilinx 4010, 10000-gate, 191-pin pin grid array part. Using the criteria set forth by the evaluation of FPGA technologies and following the decision to separate the heap manager into two sections, the design of the address list manager and heap manager controller is performed utilizing the Xilinx ProSeries software which allows schematic and VHDL entry of logic into a comprehensive architecture that is compiled and downloaded into the FPGAs via the computer interface. Although the designs are constructed separately, the heap manager and controller are imbued with the flexibility inherent in the Xilinx technology; and the structuring of the design takes advantage of the re-usable Xilinx platform to maximize the facility by which the CPU receives data and controls the system parameters.

3.1. Address List Manager

In physics experiments like PHENIX, the control of the analog memory is paramount for the successful management of the analog data [8]. Two means of addressing the AMU are the bit-enabling scheme and direct addressing. Bit-enabling schemes utilize address-specific enables to maintain the read and write operations for a specific AMU cell, while direct addressing embraces a mapping technique wherein specific addresses are provided to the AMU and addresses to memory cells are stored and retrieved from an address map in the ALM. Because the bit-enabling scheme requires logic to look ahead for unprotected analog memory cells, large delays may result, thus impeding the ability of the ALM to correctly identify an available memory location before a write is issued [9]. Conversely, direct addressing requires sufficient memory to store the addresses of all memory cells of an AMU as well as logic to control the memory blocks; however, for PHENIX, where multiple cells may be protected by a LVL-1 and sample spacing is required, direct addressing provides flexibility and ultimate control over the AMU [10]. In MVD, where the built-in RAM cells of the Xilinx FPGA may be used for address storage, the address list manager employs the direct addressing scheme to manipulate the read and write operations for the AMU.

For the direct addressing scheme of PHENIX, the architecture of the address list manager is required to provide a continuous, incremental string of sixteen addresses and sustain addressing while an address is removed or inserted into the address loop. The choice of Xilinx as the platform for the ALM permits implementation of RAM in

synchronous FIFOs which, when used in tandem, provide a storage and address retrieval matrix that is transparent to the addressing function of the AMU. Since the primary requirement for the ALM is that it be able to provide addresses to the AMU at beam clock speed, the FIFO controls and decision logic in the address list manager must operate at the faster, **4xclk** speed. Since speed and resource usage in an FPGA are closely related, a compact FIFO design is implemented to allow as many operations as possible to take place in as short a time as possible, thus negating delays which accrue from routing through multiple configurable logic blocks (CLBs) in the Xilinx part.

To achieve the highest throughput for the FIFO-based ALM, the design of the address list manager employs a minimum-delay FIFO. The high performance FIFO utilized in the address list manager is shown in Figure 3.1.1 and is developed from a design proposed by Xilinx [11]. The ALM FIFO uses linear feedback shift registers (LFSRs) for internal addressing operations and half-clock-cycle push/pop logic, advantages which show a gain in speed and decrease in area when the LFSR FIFO is compared to linear or Gray-code counter FIFOs. In the ALM FIFO, a read (**read_addr**) or write address (**write_addr**) is supplied to the static RAM (SRAM) each clock cycle; and a compare function determines the **\full** and **\empty** status of the FIFO by contrasting the relationship of the addresses. **Push** and **pop** signals are provided by the external ALM circuit to specify whether a read (**pop**) or write (**push**) is to be made to the SRAM, and the **SRAM Address** points to the corresponding memory address. The write enable of the SRAM is fed by the FIFO clock, so a read and a write are performed each clock cycle, the write on the HIGH level of the clock and the read during the LOW portion of the clock cycle.

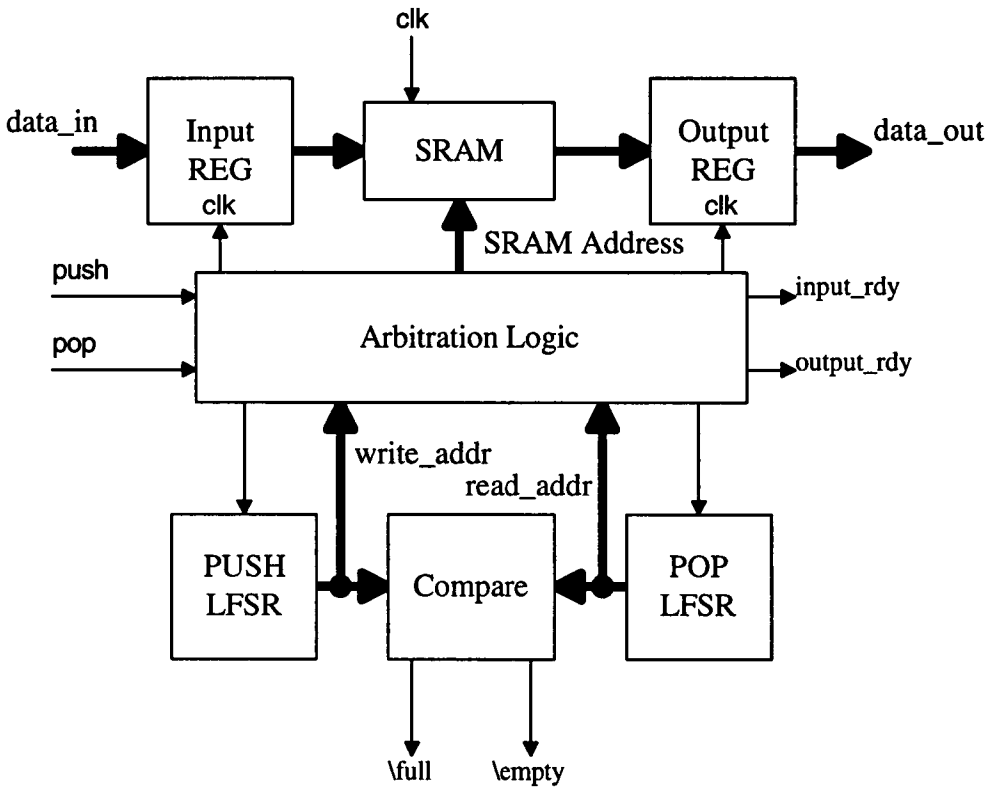


Figure 3.1.1. High Performance FIFO for the Address List Manager

Although the FIFO SRAM is written to and read from each clock cycle regardless of the function, the input and output registers are updated only when a **push** or **pop** is provided to the FIFO. Thus, during a **pop**, the Output REG will be updated; and, conversely, during a **push**, the Input REG will be updated. To ensure that no data is lost, the Arbitration Logic monitors **push** and **pop**. In the event that a simultaneous **push/pop** occurs, the FIFO delays the **pop** until the subsequent clock cycle unless the FIFO is full, in which case the **pop** is carried out first; and the **input_rdy** and **output_rdy** signals are set to show the status of the FIFO to preclude a **push** when the FIFO is full or a **pop** when the FIFO is empty. To gain maximum benefit from the LFSR FIFO, the address list manager must be configured to separate read and write operations by at least one clock cycle to avoid timing problems associated with the delaying of a **pop** coincident with a simultaneous **push/pop** occurrence.

By itself, the ability of the high performance FIFO to handle simultaneous **push/pop** operations does not significantly increase the speed of the FIFO; however, by employing linear feedback shift registers to address the SRAM, the FIFO may show a marked speed improvement over a similar counter-based FIFO during individual **push** or **pop** operations [11]. The LFSR addressing logic for the high performance FIFO is provided in Figure 3.1.2. As seen in the LSFR diagram, the shift registers for the write address (**waddr**) and read address (**raddr**) are connected to a four-input AND, a two-input XNOR, and a two-input XOR gate. By utilizing feedback from the shift register, the LFSR generates a pseudo-random sequence, which means that the SRAM is not addressed in an incremental fashion. Of course, since the FIFO is self-contained, mode of addressing

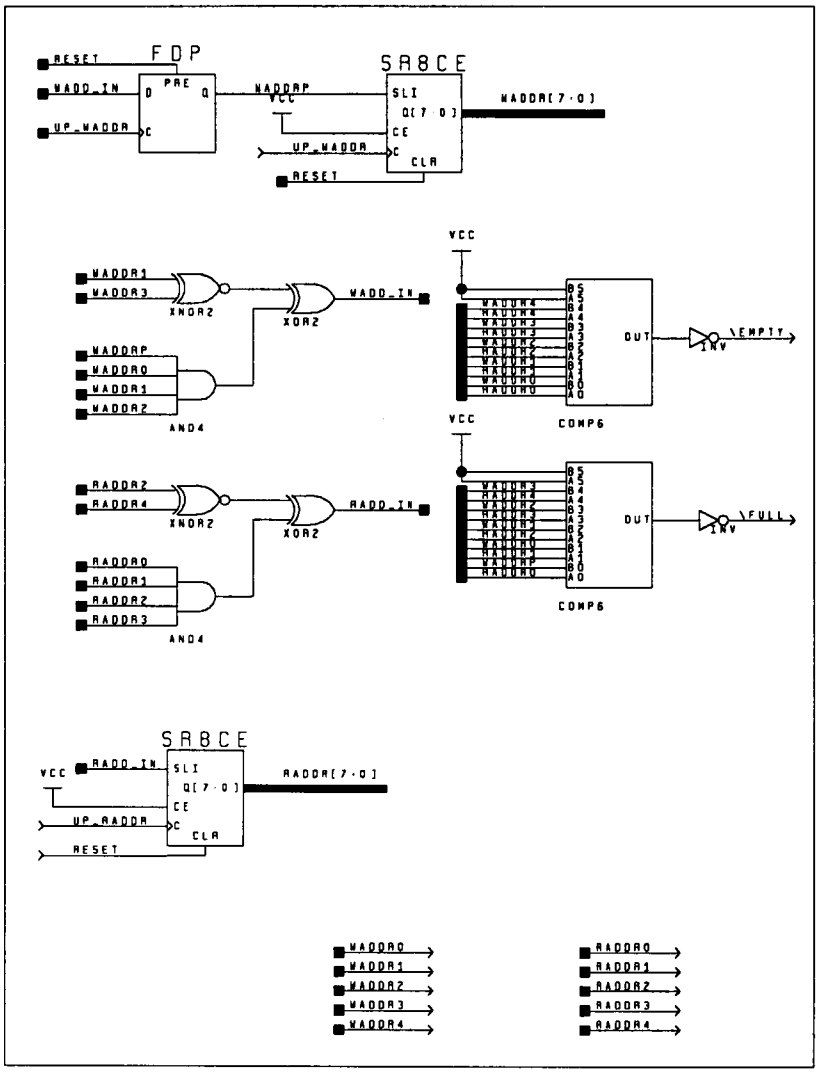


Figure 3.1.2. Five-Bit Linear Feedback Shift Register Logic

does not affect the external ALM circuit; however, the fact that adjacent addresses generated by the LFSR are identical in all but the most significant bit can be a distinct design advantage. In fact, the main benefit of using LFSR counters in the design is that two adjacent addresses may be accessed by the shifting of one extra bit into the shift register, which means that the flag logic may be easily generated, thus saving space and increasing speed.

The implementation of the LFSR FIFO in the address list manager is based on the **push**, **pop**, and **\empty** signals. For the simulation presented in Figure 3.1.3, which shows the addressing behavior for the LFSR FIFO, the push and pop signals are maintained in the active HIGH state; and the input data (**data_in**) is supplied by a counter to demonstrate the performance of the FIFO in the presence of changing data. In the simulation, when reset goes LOW, since both **push** and **pop** are HIGH and the FIFO is not full, **push** has the priority; and the write address (**write_addr**) changes as **data_in** is written to the SRAM. Notably, the **\empty** flag immediately goes LOW as the first data word is written into the FIFO. For the heap manager controller, the **\empty** flag of the ADC FIFO is monitored to gauge whether an AMU cell has been marked by a LVL-1. Since the LVL-1 trigger is not contingent on the completion of a previous conversion, the ADC FIFO may contain multiple addresses, which means that the heap manager controller must have access to the **\empty** flag to determine whether a new conversion is to be initiated.

When the FIFO is full, as evidenced by the **\full** flag going LOW, the **pop** function becomes the priority since no more data may be written to the FIFO until a space becomes

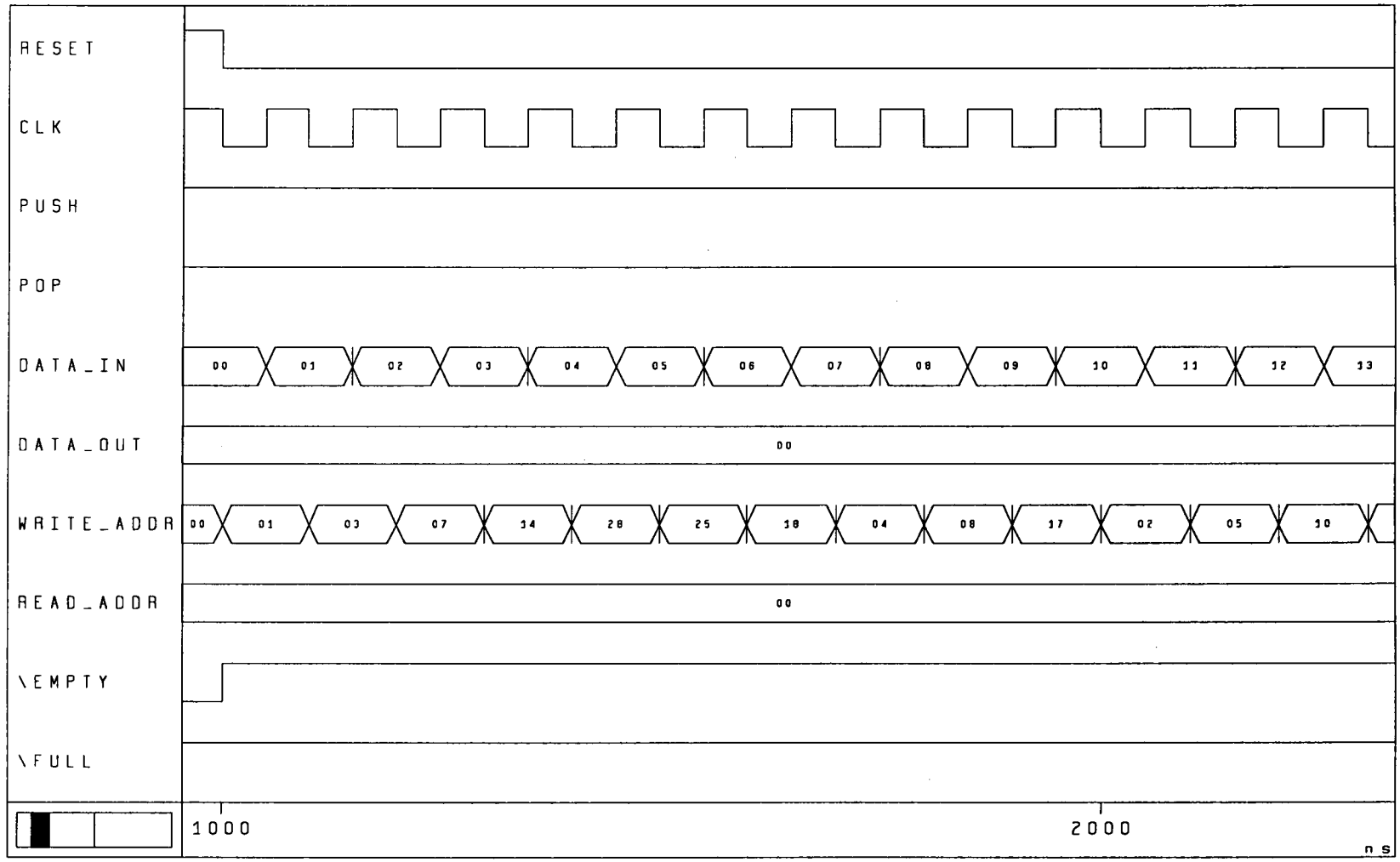


Figure 3.1.3. Address Sequencing in the LFSR FIFO

available. The movement of the read address (**read_addr**) as the **pop** command becomes active is demonstrated in Figure 3.1.4, a continuation of the simulation shown in Figure 3.1.3. As should be observed from Figure 3.1.4, the **\full** flag goes LOW one address before **write_addr** wraps around to address 00000; and, since the FIFO is filled, the data presented on the following clock cycle is not written to the FIFO. Due to the fact that data is read and written each clock cycle independent of the **push** or **pop** operation performed, one space in the SRAM is always reserved for writing. Thus, for the n=32-word FIFO, the SRAM stores only n-1 of n possible words in the SRAM; but the output latch allows storage of the n^{th} word to permit a total of n words to be stored in the FIFO. Because it is the precursor to the sixty-four-address version of MVD, the beam test LFSR FIFOs implemented are thirty-two words deep; therefore, the delays associated with the beam test FIFOs may be studied to determine the possibility of accomplishing the same speeds in the final version of MVD.

Three high performance LFSR FIFOs are integrated into the architecture of the address list manager. As determined in the planning stage of the heap manager, the ADC FIFO actually resides in the same FPGA as the heap manager controller; therefore, the ADC FIFO does not significantly affect the resource allocation or speed of the address list manager FPGA. As observed from the ALM block diagram of Figure 3.1.5, three FIFOs (ADC FIFO, Available Address FIFO, and LVL1 FIFO) are available for storage and retrieval of the AMU addresses; and the control logic is made up of a number of multiplexers (MUX blocks) and glue logic which accept inputs from the heap manager controller to provide a means of regulating the read and write (rd/wr) functions of the

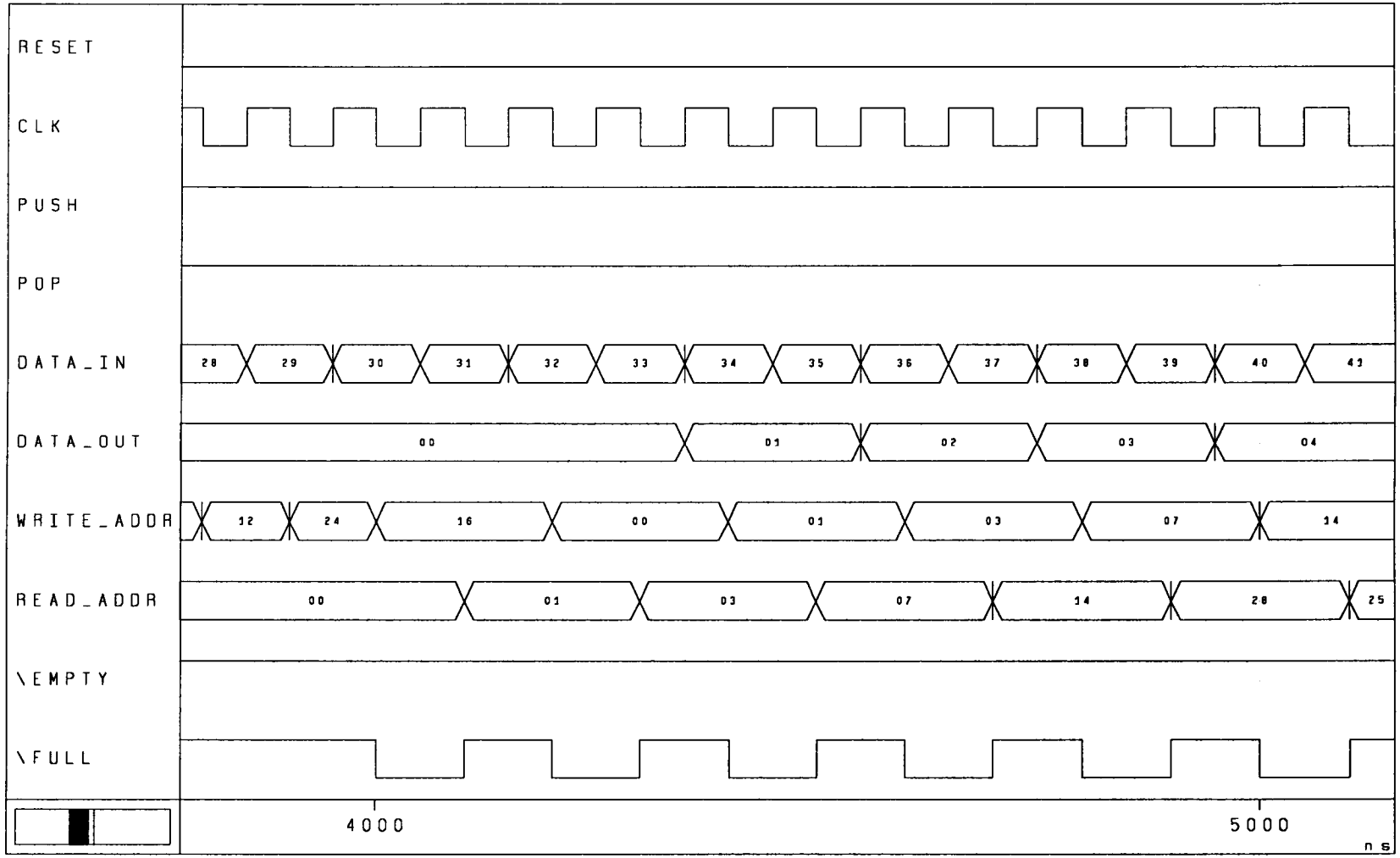


Figure 3.1.4. Read Vs. Write Operations in the LFSR FIFO

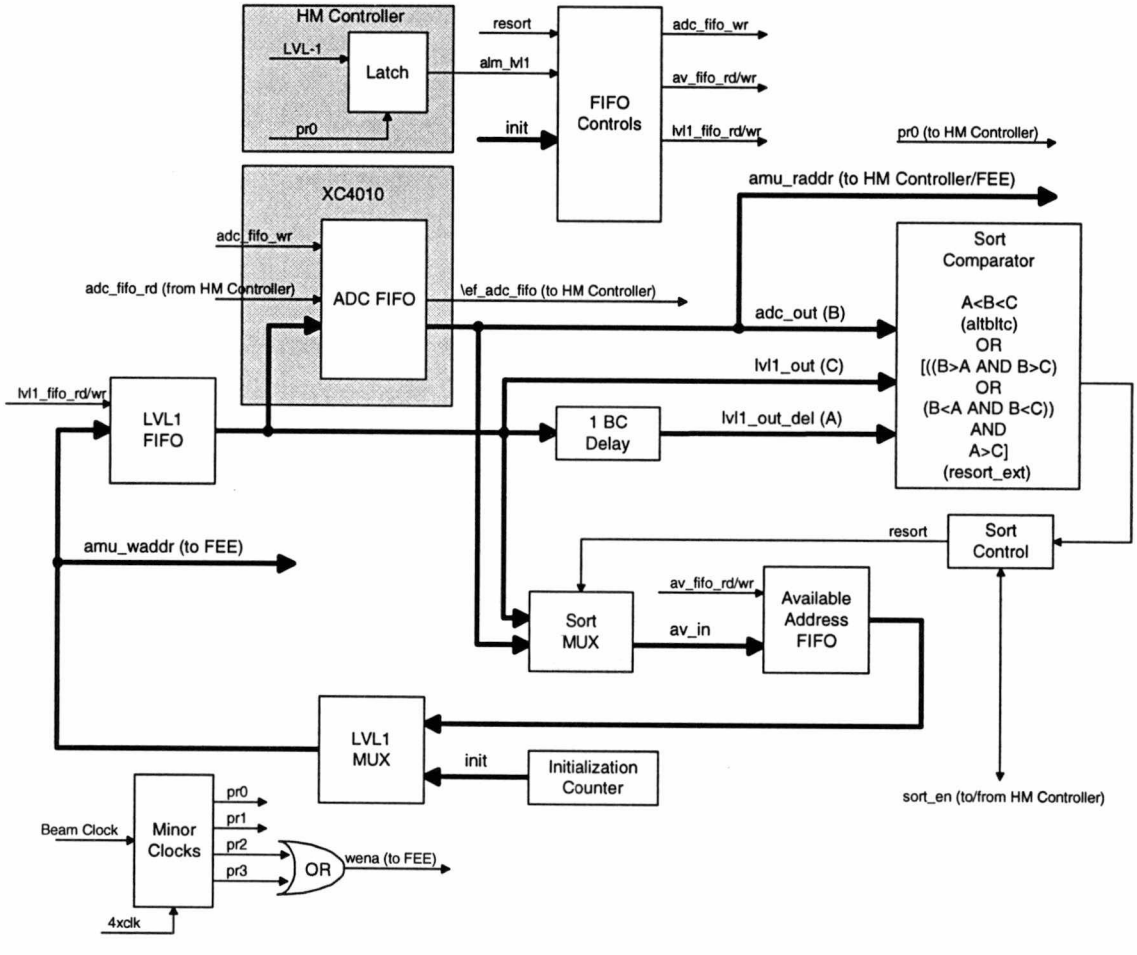


Figure 3.1.5. Address List Manager Block Diagram

FIFOs. To coordinate its functions, the address list manager partitions events into intermediate commands which are synchronized with the **4xclk**.

The address timing of the ALM is based on the beam clock; but, because several operations need to take place for an address to be processed, the beam clock is broken down into four minor clock cycles, each of which conform to the timing provided by the **4xclk**. As indicated in Figure 3.1.6, the four minor clocks are labeled **pr0**, **pr1**, **pr2**, and **pr3**, each of which delineate a specific subset of functions for ALM control. Because the address list manager must maintain a carefully regulated address loop for the AMU, the relevant information to be gleaned from the FIFO simulations is that a simultaneous **push/pop** should be avoided so that the data flow does not become divided across subsequent clock cycles. Based on the philosophy that only one FIFO operation is to be used on any given clock cycle, the address list manager performs, at most, one read and two writes to a FIFO during any given beam clock cycle. During normal (non-re-sorting) operation, all FIFO writes begin on the rising edge of **pr2**, and all reads are synchronized with **pr0** to provide the maximum delay between when one read occurs and the next write is initiated and thus allowing the output of one FIFO to settle before the address is written into the next FIFO. For the re-sorting operation, two writes must be enacted during a beam clock cycle to assure that the address loop is not broken; therefore, an additional write, synchronized to **pr1**, is made. By staggering commands with the **4xclk**, the address list manager is able to avoid a simultaneous **push/pop** while performing the operations which it needs to function properly.

The operation of the ALM is centered around three functions: initialization, LVL-1

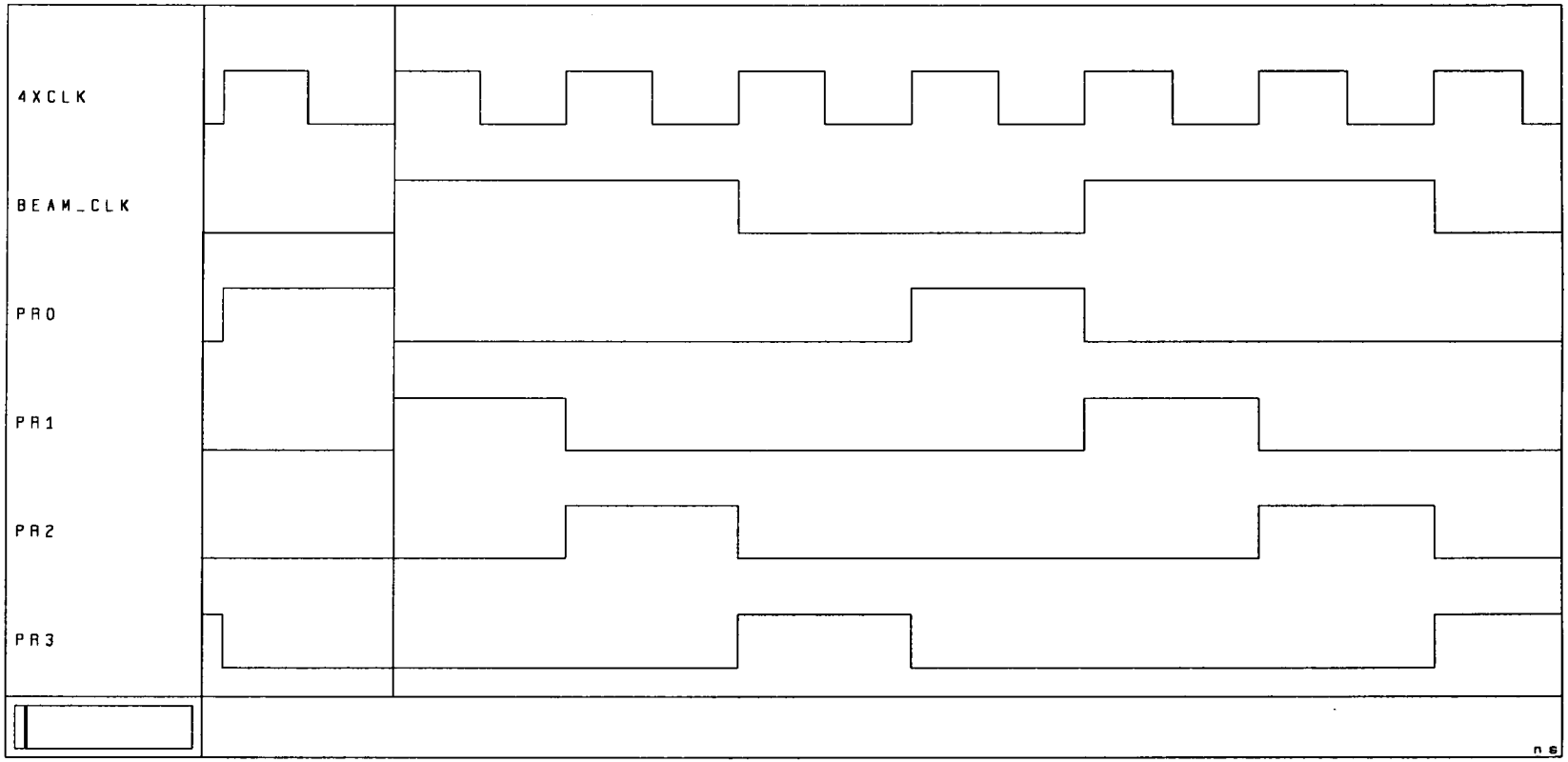


Figure 3.1.6. Clock Phases in the Address List Manager

handling, and re-sorting. For initialization, the LVL1 FIFO is loaded with the primary sixteen addresses (0 through 15) through the initialization counter. After the LVL1 FIFO is provided with the first few addresses and while the initialization process is going on, the control logic starts emptying the LVL1 FIFO and begins pushing the addresses into the Available Address FIFO. Then, when the initialization process is complete, the control logic begins taking addresses out of the Available Address FIFO and placing them back into the LVL1 FIFO. Because neither the LVL1 FIFO nor the Available Address FIFO contains all sixteen addresses at any given moment, the synchronization of the FIFOs is the mechanism that produces the direct addressing function. The result of the symbiotic relationship between the Available Address FIFO and the LVL1 FIFO is that a loop is initiated in which all sixteen addresses will be present in one of the two FIFOs.

Because it is required to strictly adhere to the beam clock timing of the rest of the MVD system, the address list manager designates specific FIFOs to hold different classes of addresses to facilitate the multiplexing of the address loop and gain the full potential from the time allotted to store and retrieve addresses for the AMU writing function pending a LVL-1 trigger. As its name implies, the Available Address FIFO contains all addresses which have not been tagged by a LVL-1 and may be overwritten in the AMU, whereas the LVL1 FIFO stores the addresses that are in the nine-beam-clock delay period and which may yet be tagged by a LVL-1 trigger. Consequently, the AMU write address (**amu_waddr**) is the multiplexed address from either the initialization counter (during initialization) or the output of the Available Address FIFO (during normal operation), and the AMU write enable (**wena**), which is centered on the **amu_waddr**, is merely the OR of

two minor clocks, **pr2** and **pr3**. In the simulation contained in Figure 3.1.7, in which no addresses are available for re-sort, the input to the Available Address FIFO (**av_in**) is provided with the output of the ADC FIFO, which holds the addresses tagged by a LVL-1, until a write occurs with **pr2**, at which time the Sort MUX selects the LVL1 FIFO output (**lvl1_out**). Since initialization is complete, the output of the Available Address FIFO (**amu_waddr**) is passed to the LVL1 FIFO; and the address loop continues. In Figure 3.1.8, a close-up of the addressing loop is displayed and shows the inputs to the Available Address FIFO as the addresses are shuffled between the LVL1 FIFO and the Available Address FIFO. As can be noted in the simulations, the distribution of commands across the four minor clock cycles facilitates the inclusion of the sort function for the Available Address FIFO; and the output of the ADC FIFO, which is physically in the heap manager FPGA, can be presented to the Available Address FIFO during most of the beam clock cycle until another address from the LVL1 FIFO is to be written, thereby maximizing the settling time of the outputs of the ADC FIFO.

As it circulates addresses to and from the Available Address FIFO, the address list manager must be able to remove from the loop any address tagged by a LVL-1 and store the address in the ADC FIFO. Because there is a delay of nine beam clocks between the time when a particular address is output by the Available Address FIFO and when it emerges from the LVL1 FIFO, the decision to take an address out of the loop must be made within nine beam clocks after the address is written to the AMU and deposited in the LVL1 FIFO. To synchronize the LVL-1 with the read and write functions of the FIFOs, the address list manager samples the LVL-1 at the beginning of each beam clock. If a

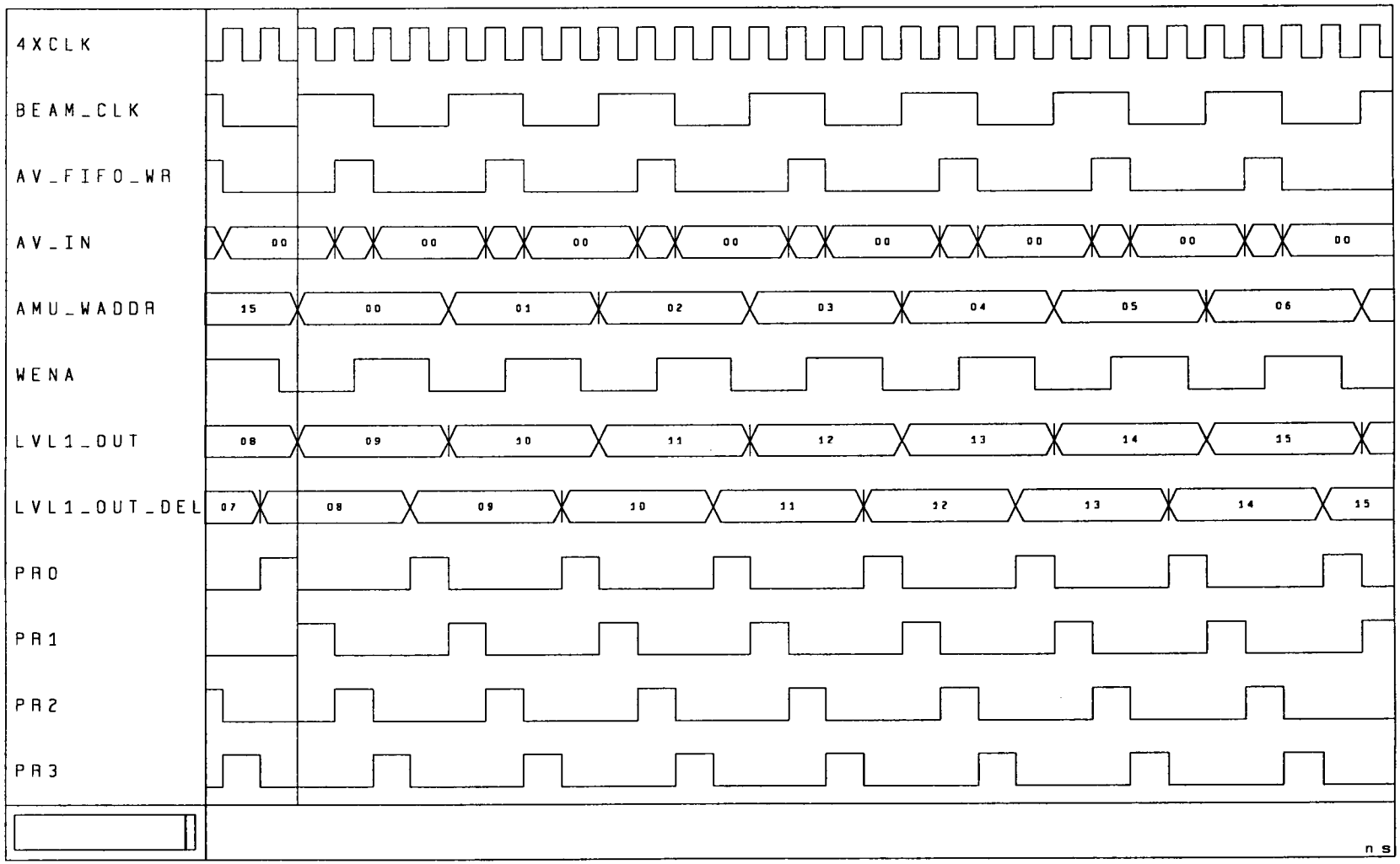


Figure 3.1.7. FIFO Addressing in the Address List Manager

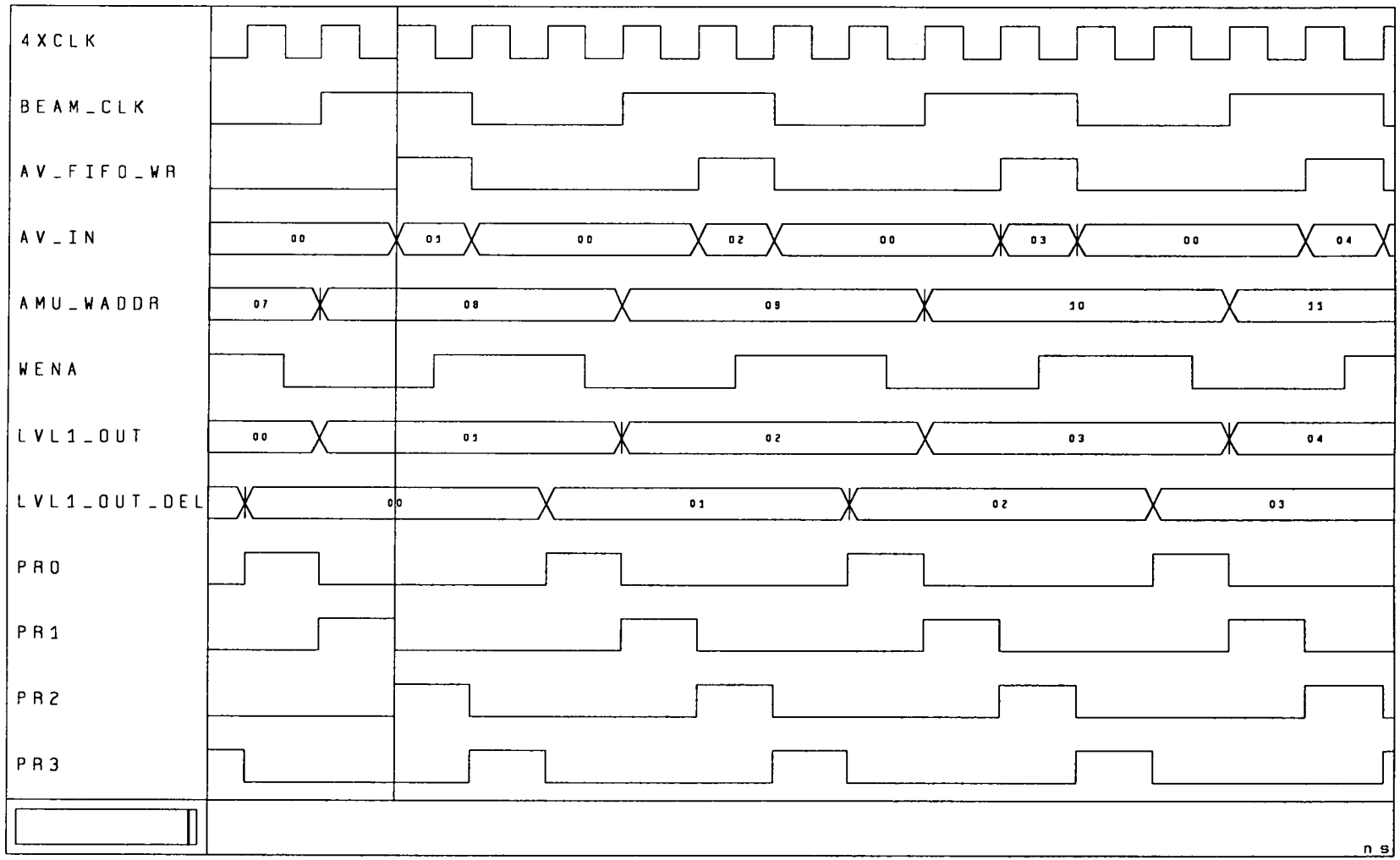


Figure 3.1.8. Clocking of Addresses in the Address List Manager

LVL-1 is detected, the corresponding address (**lvl1_out**) from the LVL1 FIFO is taken out of the loop by masking the write to the Available Address FIFO (**av_fifo_wr**); and a write to the ADC FIFO (**adc_fifo_wr**) is initiated. Thus, not only does it effectively remove an address, but a LVL-1 also causes the address to be saved so that the analog value stored in the AMU may be read at a later time, converted by an ADC, and stored as digitized data through the heap manager controller and PC interface.

As should be observed in the simulations of the addressing scheme of the address list manager, **lvl1_out** and **amu_waddr** are separated by nine addresses, or nine beam clocks; and the LVL-1 removes an address which was written to the AMU nine beam clocks in the past. The address list manager is capable of handling a maximum of seven LVL-1s without a re-sort. The limit on the number of LVL-1s corresponds to the number of times the Available Address FIFO may be read before it is drained completely, at which time no more addresses are available and an additional read will produce a non-unique address. A LVL-1 condition for the address list manager is demonstrated in Figure 3.1.9. In the case displayed in Figure 3.1.9, a pair of consecutive LVL-1s (a Pre and a Post LVL-1) are generated; and each LVL-1 is synchronized with **pr0** and is presented to the ALM as **alm_lvl1**. As per the simulation, the LVL-1 masks addresses 03 and 04 by suppressing the **av_fifo_wr** signal and masking **lvl1_out** address from the input to the Available Address FIFO, thus effectively taking the two addresses out of the loop; and, during the same beam clock cycle as the active **alm_lvl1**, the **adc_fifo_wr** signal is activated to push both **lvl1_out** addresses into the ADC FIFO. Thus, the address list manager is able to remove addresses from the loop by simply masking the write signal to the Available

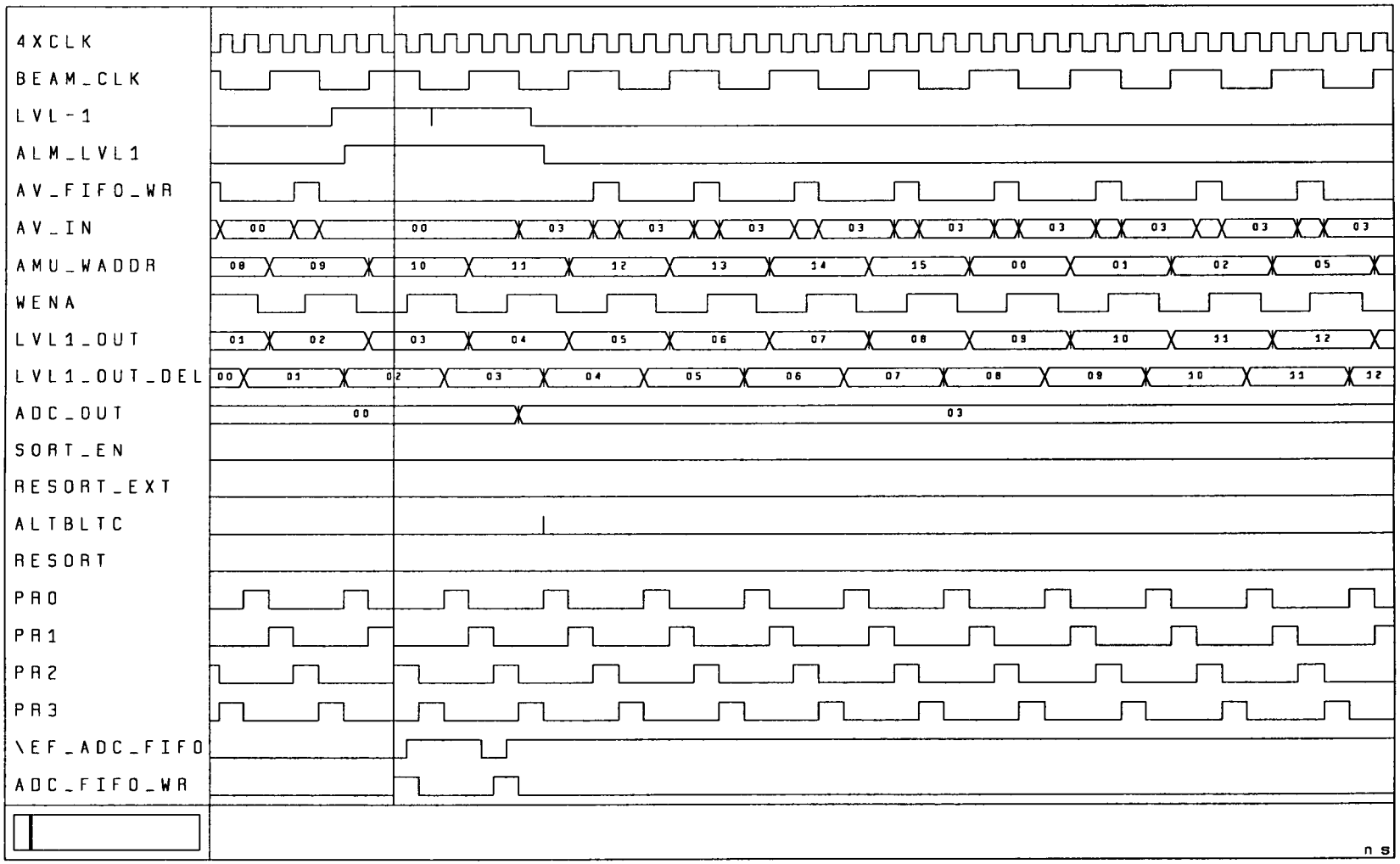


Figure 3.1.9. LVL-1 Handling in the Address List Manager

Address FIFO; and the fact that the AMU write address is supplied by the Available Address FIFO allows the addressing of the AMU to continue unabated.

After they are removed from the loop and placed in the ADC FIFO, the addresses tagged by a LVL1 are used by the heap manager controller to read the AMU and are then returned to the address list manager for re-sort. Since the readout function of the heap manager is performed by the heap manager controller and is transparent to the address list manager, some means of synchronizing the re-sorting of an address must be contrived. Once it is read from the ADC FIFO, the AMU read address (**adc_out**) is immediately available to the ALM; however, to prevent the address from being re-sorted before the conversion process is finished, a sort enable (**sort_en**) signal must set so that the AMU address is not overwritten prematurely. When the conversion is finished and the **sort_en** signal is set, the ALM places the address back into the address loop at its proper location by using a delayed version of the LVL1 FIFO output (**lvl1_out_del**), which is the **lvl_out** address from the previous beam clock cycle, and the current LVL1 FIFO output (**lvl1_out**) to compare with the output of the ADC FIFO. A successful re-sort occurs when the ADC FIFO output rests between two adjacent addresses, or **lvl_out** and **lvl1_out_del**; however, due to the fact that the **adc_out** address may be either at the border of the address loop or in the middle of the address loop, two permutations of the re-sort operation are possible.

One occurrence of a re-sort is when the ADC FIFO address lies between two incremental addresses. For instance, in the case shown in Figure 3.1.10, address 03 is re-sorted. In Figure 3.1.10, the **altbltc** (A<B<C) flag of the Sort Comparator indicates that

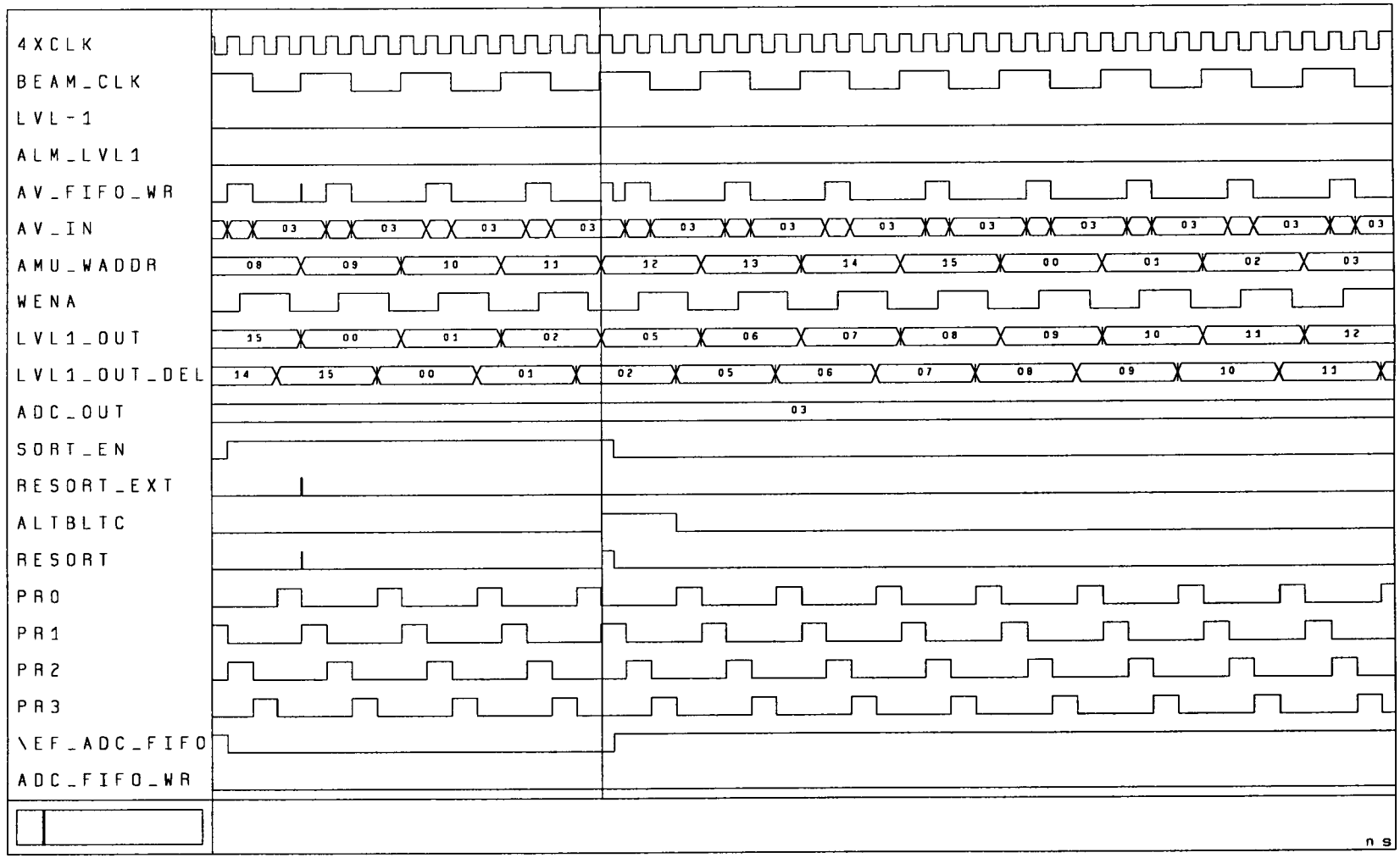


Figure 3.1.10. A<B<C Re-Sort Timing for the Address List Manager

the **adc_out** address rests between two incremental addresses (**lvl1_out**=02 and **lvl1_out_del**=05); therefore an additional write, which is synchronized with **pr1**, to the Available Address FIFO (**av_fifo_wr**) is made to push address 03 into the FIFO. Because the write does not overlap the normal **av_fifo_wr**, which is synchronous with **pr2**, both the **adc_out** address (address to be re-sorted) and the **lvl1_out** address are written to the Available Address FIFO during the same beam clock period. Because the ADC FIFO output is present from the beginning of the beam clock cycle until the normal **av_fifo_wr** is made, **adc_out** has three **4xclk** cycles to settle; and, because they are separated by two **4xclk** cycles, the normal read and write functions of the ALM do not impinge on the write needed to re-sort the ADC FIFO address back into the loop.

A second instance of a re-sort condition occurs when **adc_out** lies at the border of the addressing loop as the loop wraps around from the final available address to the initial address. If the **adc_out** address rests on the border of the address loop and not between two incremental addresses, the **altbltc** signal is not triggered. For instance, as can be observed in Figure 3.1.11, address 15, which has been removed by a LVL-1, is presented to the ALM to be re-sorted; however, because its position in the loop lies between address 14 (**lvl1_out_del**) and address 01 (**lvl1_out**), address 15 does not trigger the **altbltc** condition signal. Thus, to accomplish the border address re-sort condition which exists in the simulation shown in Figure 3.1.11, the **resort_ext** condition ((B<A AND B<C) OR (B>C AND B>A)) is tested, which indicates that address 15 is a border address; and, since it rests between address 14 and address 01 in the loop, address 15 is written into the Available Address FIFO using an additional **av_fifo_wr**, which is synchronized with **pr1**

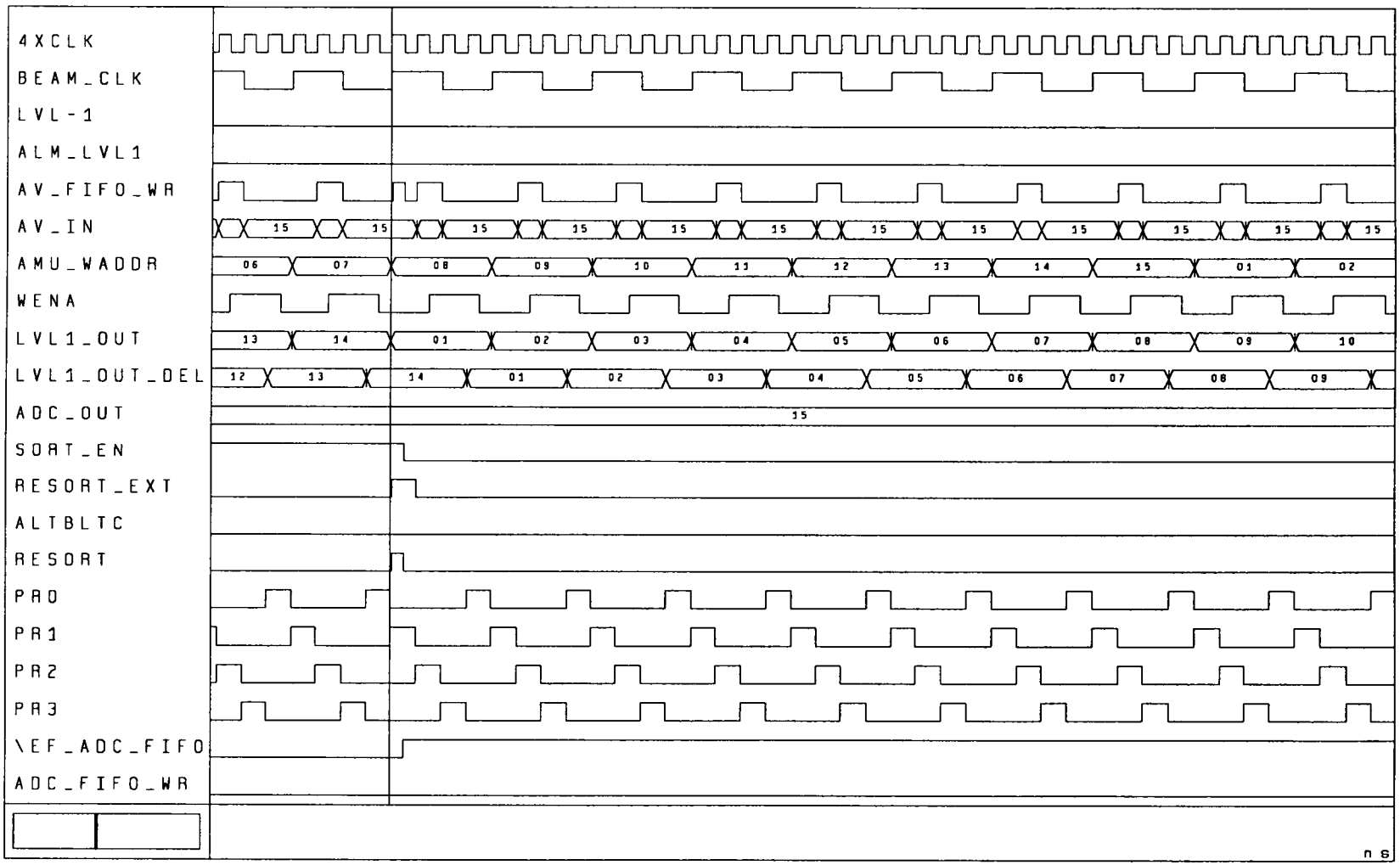


Figure 3.1.11. Border Re-Sort Timing for the Address List Manager

as is the case in the simulation of Figure 3.1.10.

The ability of the address list manager to handle addressing of the AMU during system operation is paramount to the success of the MVD beam test. The address list manager must be able to set up a sixteen-address loop and be able to take out one or more addresses and, in turn, re-sort the addresses, in order, while complying with the timing provided by the **4xclk** and the beam clock; and the heap manager controller must have a way of communicating to the ALM the need to remove and re-sort a particular address. Construction of the address list manager in a four-clock-cycle architecture imbues the ALM with the ability to mask and write addresses into any one of three FIFOs while maintaining a steady stream of addresses to the AMU and providing the heap manager controller with the ability to affect changes in the address loop without impeding the write functions to the analog memory.

3.2. Heap Manager Controller

The heap manager controller, in contrast to the address list manager, is a combination of disparate blocks which perform a wide range of functions. To perform its task, the controller must be able to provide as much flexibility as possible and handle a large number of I/O. Because of the large I/O requirement, an XC4010, 10000-gate, 191-pin pin grid array (PGA) Xilinx part is needed to implement the logic which encompasses all of the support functions needed by the ALM and FEE and the data collection, control,

and serial string interfaces which are required by the external PC/CAMAC interfaces. A block diagram of the heap manager controller is given in Figure 3.2.1. Ultimately, the responsibility of the heap manager controller is to return a formatted data packet to the PC or CAMAC; however, a number of functions are programmable within the controller to alter the manner in which data is collected and processed. The control interface, trigger interface, and serial interface are used to delineate functions which manipulate the mode of operation of the heap manager controller and, consequently, the beam test system as a whole. To yield access to human control of the system, management of the heap manager controller is left to the CPU (PC or CAMAC), which provides the commands from a visual software interface.

One of the command environments in the heap manager controller is the control interface, which is defined by a four-bit bus that sets one of six possible commands for the controller. The functions provided in the control interface are preemptive commands that affect the fundamental operation of the controller and permit the human user of the system access to the primary readout and clocking functions involved in system behavior. Synchronized with the beam clock to avoid a race in the controller state machine, the mode bits and the commands associated with the permutations of the five-bit codes are organized in Table 3.2.1. For the basic command structure, only the first three mode bits are utilized; and the fourth mode bit and fifth mode bit are reserved for future use. Command 000 is a passive command that keeps the controller in its current mode and is used as a transition command when the heap manager is in operation. The Integrator Reset instruction causes a single integrator reset, **ir**, and its corresponding **\ir**, to be issued

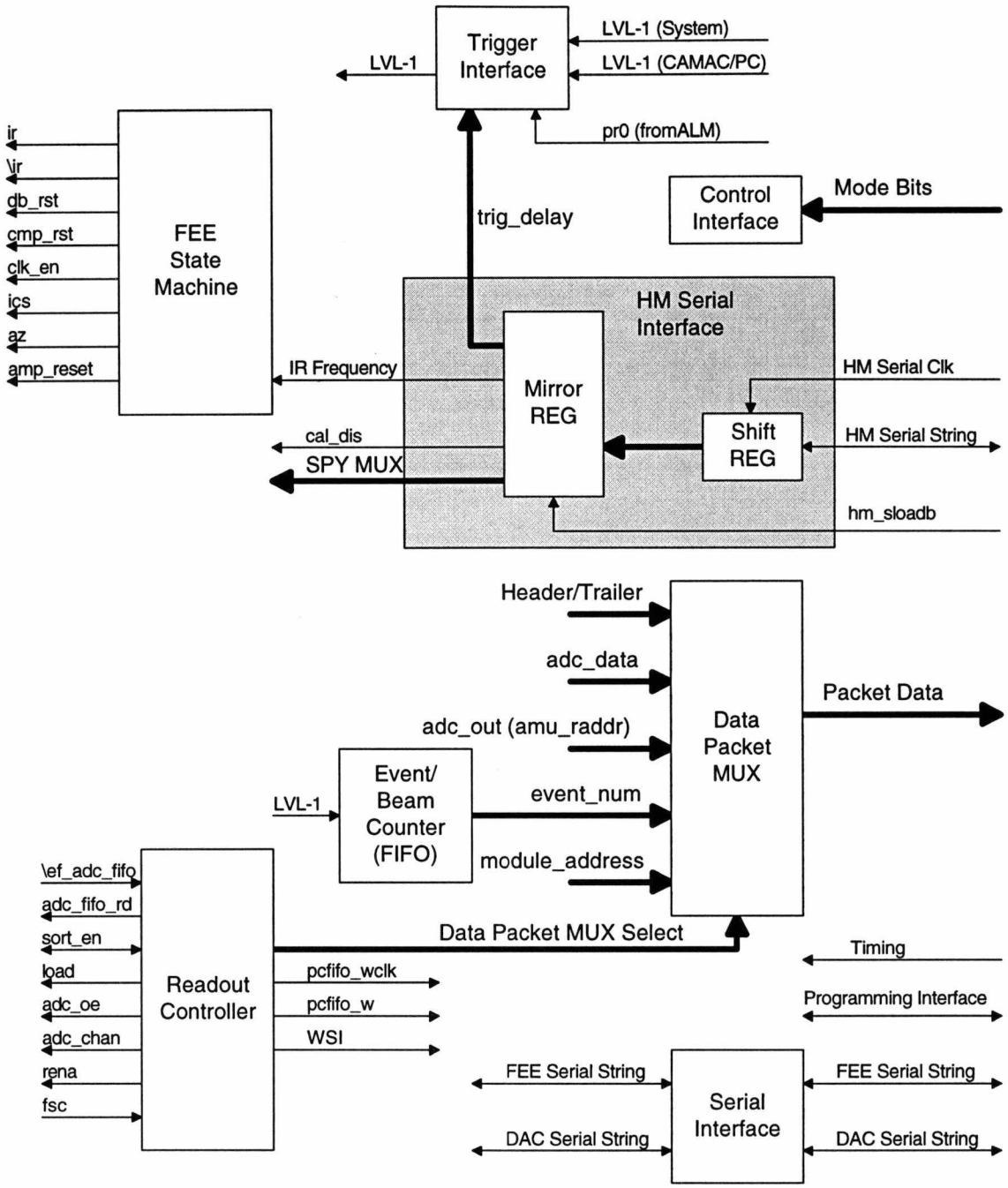


Figure 3.2.1. Heap Manager Controller Block Diagram

Table 3.2.1. Mode Bit Commands for the HM Controller

MB[2:0]	Command
000	No-op (No Operation)
001	Integrator Reset
010	Pipeline Reset
011	Run
100	Halt All
101	Next SL Command
110	Halt Acquisition
111	Undefined

to the preamp and allows the feedback capacitor of the preamp to be discharged. Pipeline Reset initializes the ADC FIFO and external data collection FIFO and effectively resets the FEE state machine without affecting any FEE serial functions or the heap manager serial interface. The Halt All function disables the clocks for the entire heap manager; but the Halt Acquisition is included to allow the ALM to continue addressing while only the LVL-1 related functions are disabled. Finally, because the heap manager is expected to start up in the disabled mode, the Run command is issued to begin the acquisition process. Since it describes the commands which affect the primary state machines of the heap manager controller, the control interface is critical to the success of the beam test unit and affords the human user the ability to manipulate the system on a very basic level.

In contrast to the control interface, the serial interface of the heap manager controller is designed to provide secondary functions to the front-end electronics. Whereas the control interface is separated into individual bits to allow commands to change each beam clock, the serial interface consists of a serial string eighteen bits in length, which is shifted in by a clock provided by the CAMAC or PC using the timing from Figure 3.2.2. To make sure commands are not misinterpreted and to keep the integrator reset frequency from changing while bits are shifted into the control register, a mirror register must be clocked, with **hm_sloadb**, by the computer to load and execute each command. Furthermore, to provide the computer with feedback, a return bit is derived from the final bit of the shift register; and the computer may make additional shifts to read out the serial string while the Mirror REG holds the command for the heap manager. Distribution of the serial string among several instructions enhances the robustness of the heap manager

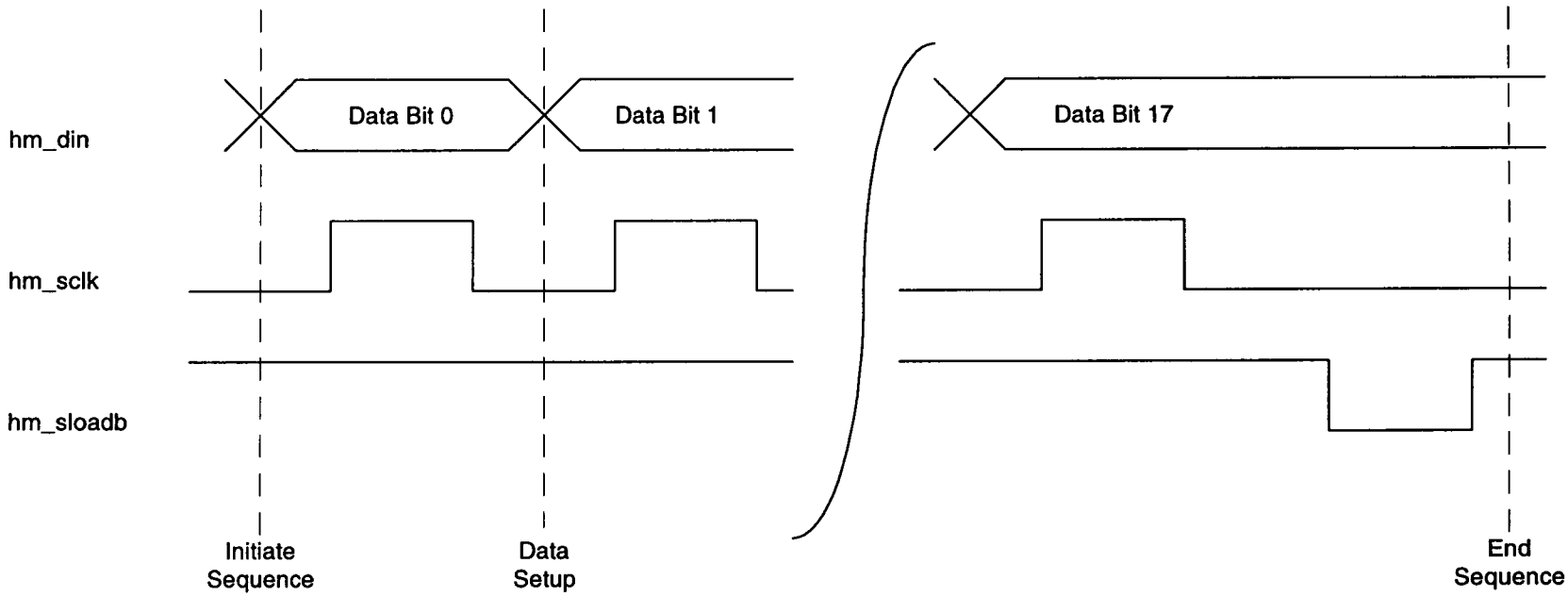


Figure 3.2.2. Heap Manager Serial Stream Timing

controller and increases the controllability of the system while limiting the I/O needed by the PC interface to issue a wide range of commands.

The command bit definitions for the eighteen-bit serial interface string are shown in Table 3.2.2 and include five subsets of controls. In the heap manager serial interface command structure, the initial two bits, IR1 and IR0, of the serial string set the integrator reset (IR) frequency, which determines how often the preamplifier is re-initialized. Four permutations of the IR frequency command bits, as presented in Table 3.2.3, are defined in terms of the number of beam clocks between resets. When in reset mode, the preamp yields no valid data; therefore, the IR frequency must be correlated with the system LVL-1 status so that data stored during the reset is not mistaken as valid data. Thus, the frequency of **ir** and **\ir** is critical in the calculation of the dead time of the system; and, with an integrator reset frequency ranging from 1000 to 8000 beam clock cycles, the MVD system can be adjusted to guarantee valid data within several temporal windows of operation.

Bits two through thirteen of the heap manager serial string set the addresses of the SPY multiplexers. Each of the SPY multiplexers produces an output which may be read at a test point on the FEE mother board and precludes the use of probing techniques on the sensitive areas of the board. The MUXEN signal enables all SPY multiplexers, and the remaining bits signify specific addresses for the multiplexers, each of which functions independently of the other two. As shown in Table 3.2.4, M1A2, M1A1, and M1A0 address Multiplexer #1 and allow the outputs from the preamp chips and the reference voltages, **Vgate**, **Vref**, and **Vthreshold** from the serial DACs, to be seen. Similarly,

Table 3.2.2. Heap Manager Serial String Commands

Bit Name	Bit Position
IR1	0
IR0	1
M1A2	2
M1A1	3
M1A0	4
M2A3	5
M2A2	6
M2A1	7
M2A0	8
M3A3	9
M3A2	10
M3A1	11
M3A0	12
MUXEN	13
RAW_MODE	14
CAL_DIS	15
TRIG_DELAY0	16
TRIG_DELAY1	17

Table 3.2.3. Integrator Reset Frequency Commands

IR[1:0]	IR Frequency
00	1000 Beam Clocks
01	2000 Beam Clocks
10	4000 Beam Clocks
11	8000 Beam Clocks

Table 3.2.4. SPY Multiplexer #1 (M1) Output Vs. Address

Multiplexer Address M2 M1 M0	Output Function
000	Preamp Chip #1 Output #4
001	Preamp Chip #2 Output #4
010	Preamp Chip #3 Output #4
011	Preamp Chip #4 Output #4
100	VGATE DAC Diagnostic
101	VTH DAC Diagnostic
110	VREF DAC Diagnostic
111	NC

M2A3-M2A0 and M3A3-M3A0 address Multiplexer #2 and Multiplexer #3, respectively; and the outputs derived from the multiplexers are shown in Table 3.2.5 and Table 3.2.6. Thus, all of the AMU outputs may be read easily by sending the proper commands to through the serial interface. Consequently, use of the SPY Multiplexers increases the testability of the beam test system by providing a safe and easy way of examining system parameters without interfering with normal system operations.

Also included in the heap manager serial string are bits which set internal functions for the heap manager controller. If it is set, **raw_mode** forces the heap manager controller to generate two LVL-1s, a Pre and a Post, for each LVL-1 received; and, because the beam test is to be run in raw mode only, the **raw_mode** bit will always be set by the CPU. The presence of the **raw_mode** command bit indicates that the final MVD system will be able to run in either raw mode or correlator mode. In conjunction with the **raw_mode** control, the **trig_delay** bits set the delay between the Pre and Post LVL-1s as indicated in Table 3.2.7. The programmable delay between Pre and Post LVL-1s provides the flexibility to sense data that would otherwise be ignored; and, because the system is not yet proven, the need to see data from several perspectives can prove useful to the debugging effort.

Finally, the **cal_dis** signal is added to the heap manager serial string to provide the discharge for the calibration of the preamp. During the ORNL test phase, the detectors are not present; therefore, the **cal_dis** signal permits use of the **Vcal** inputs to the preamps to be used as test input voltages to measure data which is circulated through the system. By packaging the **cal_dis** signal with the other four commands imbedded in the heap manager

Table 3.2.5. SPY Multiplexer #2 (M2) Output Vs. Address

Multiplexer Address M3 M2 M1 M0	Output Function
0000	AMU Chip #1 Output #0
0001	AMU Chip #1 Output #1
0010	AMU Chip #1 Output #2
0011	AMU Chip #1 Output #3
0100	AMU Chip #1 Output #4
0101	AMU Chip #1 Output #5
0110	AMU Chip #1 Output #6
0111	AMU Chip #1 Output #7
1000	AMU Chip #2 Output #0
1001	AMU Chip #2 Output #1
1010	AMU Chip #2 Output #2
1011	AMU Chip #2 Output #3
1100	AMU Chip #2 Output #4
1101	AMU Chip #2 Output #5
1110	AMU Chip #2 Output #6
1111	AMU Chip #2 Output #7

Table 3.2.6. SPY Multiplexer #3 (M3) Output Vs. Address

Multiplexer Address M3 M2 M1 M0	Output Function
0000	AMU Chip #3 Output #0
0001	AMU Chip #3 Output #1
0010	AMU Chip #3 Output #2
0011	AMU Chip #3 Output #3
0100	AMU Chip #3 Output #4
0101	AMU Chip #3 Output #5
0110	AMU Chip #3 Output #6
0111	AMU Chip #3 Output #7
1000	AMU Chip #4 Output #0
1001	AMU Chip #4 Output #1
1010	AMU Chip #4 Output #2
1011	AMU Chip #4 Output #3
1100	AMU Chip #4 Output #4
1101	AMU Chip #4 Output #5
1110	AMU Chip #4 Output #6
1111	AMU Chip #4 Output #7

Table 3.2.7. Trigger Delay Between Pre and Post LVL-1s

TRIG_DELAY[1:0]	Delay (Beam Clocks)
00	1
01	2
10	4
11	8

serial string into a four-signal interface, the heap manager controller is imbued with the ability to receive multiple commands while minimizing the routing and I/O requirements necessitated by eighteen separate inputs.

To perform its primary data acquisition functions, the heap manager controller combines the commands inherent in the control and serial interfaces with the DCM and trigger interfaces and coordinates the individual logic functions with the address list manager and the PC. The timing of the heap manager controller state machines is divided into the areas of LVL-1 handling with the ALM, FEE state machine, and data packet disbursement with the PC, each of which is related, in some way, to the other two. The trigger interface, for instance, is set up to accept a LVL-1, either from the computer or from the system, as determined by the trigger sum from the preamps; and because it coordinates the readout and re-sorting functions with the ALM, the heap manager must synchronize and relay an incoming LVL-1 so that the proper address is taken out of the AMU write loop and placed in the ADC FIFO. Furthermore, because it is designed to run in raw mode, the heap manager controller has to generate two LVL-1s, a Pre and a Post, while maintaining the spacing of the LVL-1s dictated by the **trig_delay** serial string bits received from the PC. A typical LVL-1 event, where the trigger delay is set to one, is illustrated in Figure 3.2.3; and the synchronized LVL-1 is denoted by the **LVL-1** signal. In the simulation presented in Figure 3.2.3, a Pre and Post LVL-1 are generated consecutively; and the analog values stored in two AMU addresses are tagged and written to the ADC FIFO. Due to the fact that the ADC FIFO is in the heap manager controller FPGA and the LVL1 FIFO is contained in the address list manager FPGA, the output of

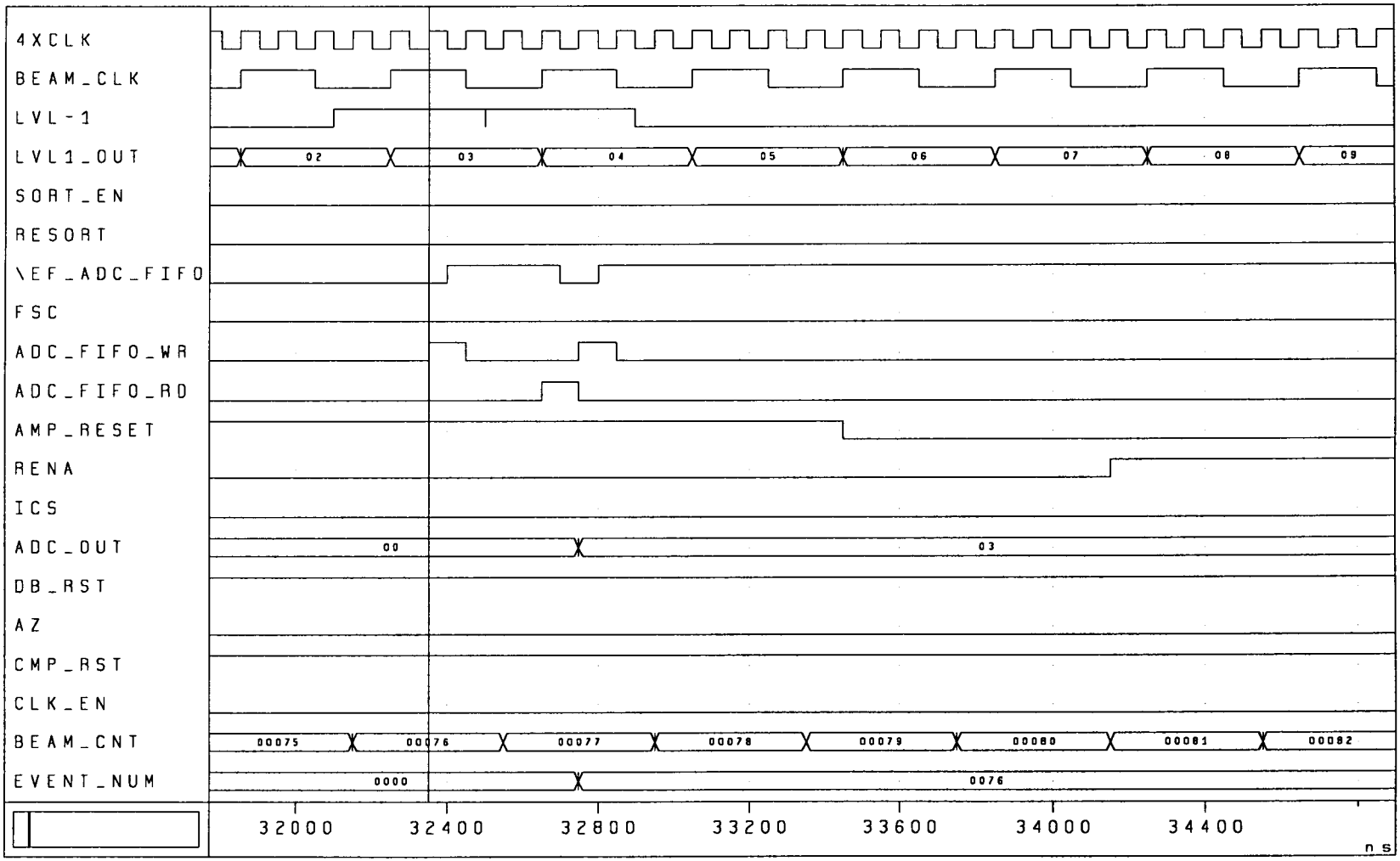


Figure 3.2.3. LVL-1 Condition Timing for the Heap Manager Controller

the LVL1 FIFO is presented to the heap manager controller as **lvl1_out**; and address 03 and address 04, the **lvl1_out** addresses present during the LVL-1, are written to the ADC FIFO. In conjunction with the LVL-1 event, the twelve-bit Event/Beam Counter, which is clocked by the beam clock, is utilized to store a place marker for the system so that a relative account of the trigger delay can be measured; and the values of the counter (**beam_cnt**) present during the LVL-1 are written into consecutive memory addresses of the Event FIFO, a 32x16 LFSR FIFO. Further, as seen in Figure 3.2.3, as soon as **\ef_adc_fifo** goes low, indicating that a value is stored in the ADC FIFO, the ADC FIFO is read via the **adc_fifo_rd** signal; and **adc_out** indicates that address 03 of the AMU is to be converted. The FEE state machine, upon the read of the ADC FIFO, causes the **amp_reset** of the AMU to go LOW, thus putting the AMU amplifier in working mode, and asserts the AMU read enable, **rena**, to read out the analog value of the AMU as addressed by **adc_out**. With the activation of rena, the analog value corresponding to the LVL-1 is presented to the ADC; and a conversion may be initiated.

The conversion sequence is implemented to minimize power and maximize efficiency of the AMU addressing architecture. During a conversion, all resets are asserted as soon as their corresponding functions are completed; and the **amu_raddr** is provided to the ALM immediately after the analog value of the AMU is stored on the internal switch capacitor of the ADC. The conversion of the AMU analog value is carried out as detailed in Figure 3.2.4. First, the comparator reset, **cmp_rst**, of the ADC is lowered to enable the ramp comparator; and the auto-zero, **az**, of the ADC is pulsed to bring the ADC amplifier to baseline. As soon as the analog value of the AMU is switched onto the

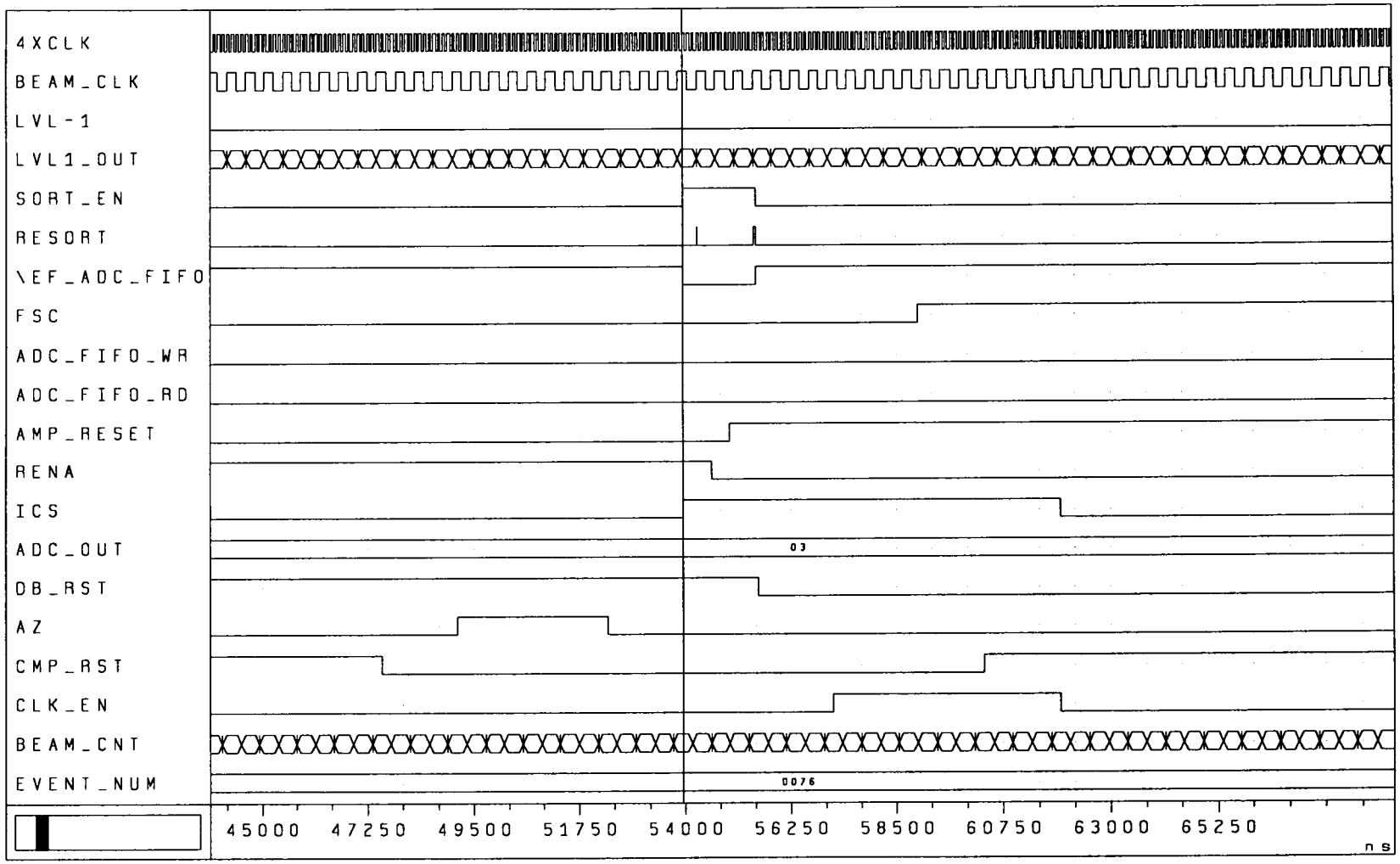


Figure 3.2.4. FEE State Machine Timing for the Heap Manager Controller

internal switch capacitor of the ADC via **ics**, the **rena** signal and **amp_reset** of the AMU can be switched to turn off the analog memory read circuit; and, since all circuits consume more power when logic levels are switching, resetting the circuit minimizes the power consumption of the comparator. With the analog value from the AMU safely held by the internal switch capacitor of the ADC, a conversion is initiated by bringing the debounce reset, **db_rst**, LOW and starting the ADC clock and ramp with the **clk_en** signal; and, at the same time, the heap manager controller sets the sort enable (**sort_en**), as set forth in Figure 3.2.5, so that the ALM may re-sort the address as soon as possible, thus providing another address for analog data storage and maximizing AMU resources. Timing of the conversion process so that all resources are used most effectively is critical, especially in the beam test where only sixteen AMU addresses are available and power consumption is to be measured as a benchmark for further development of the MVD system.

As soon as an ADC end-of-conversion (**fsc**) is detected, the heap manager controller begins a readout of the data and submits a packet of information to the external FIFO and the CAMAC. The formatting of a data packet is carried out by the heap manager controller through the use of a multiplexer, the Data Packet MUX. Each data packet contains seventy-one twelve-bit words as delineated, in order of appearance, in Table 3.2.8. Because two FEE mother boards are used in the beam test, two data packets are sent for each conversion, thus yielding 142 pieces of data for each individual LVL-1 encountered in raw mode. The first two words and last two words in each packet are static headers/trailers, HEX AAA followed by HEX 555 (decimal 2730 and 1365, respectively), which serve as packet boundaries. The event number (**event_num**), which is

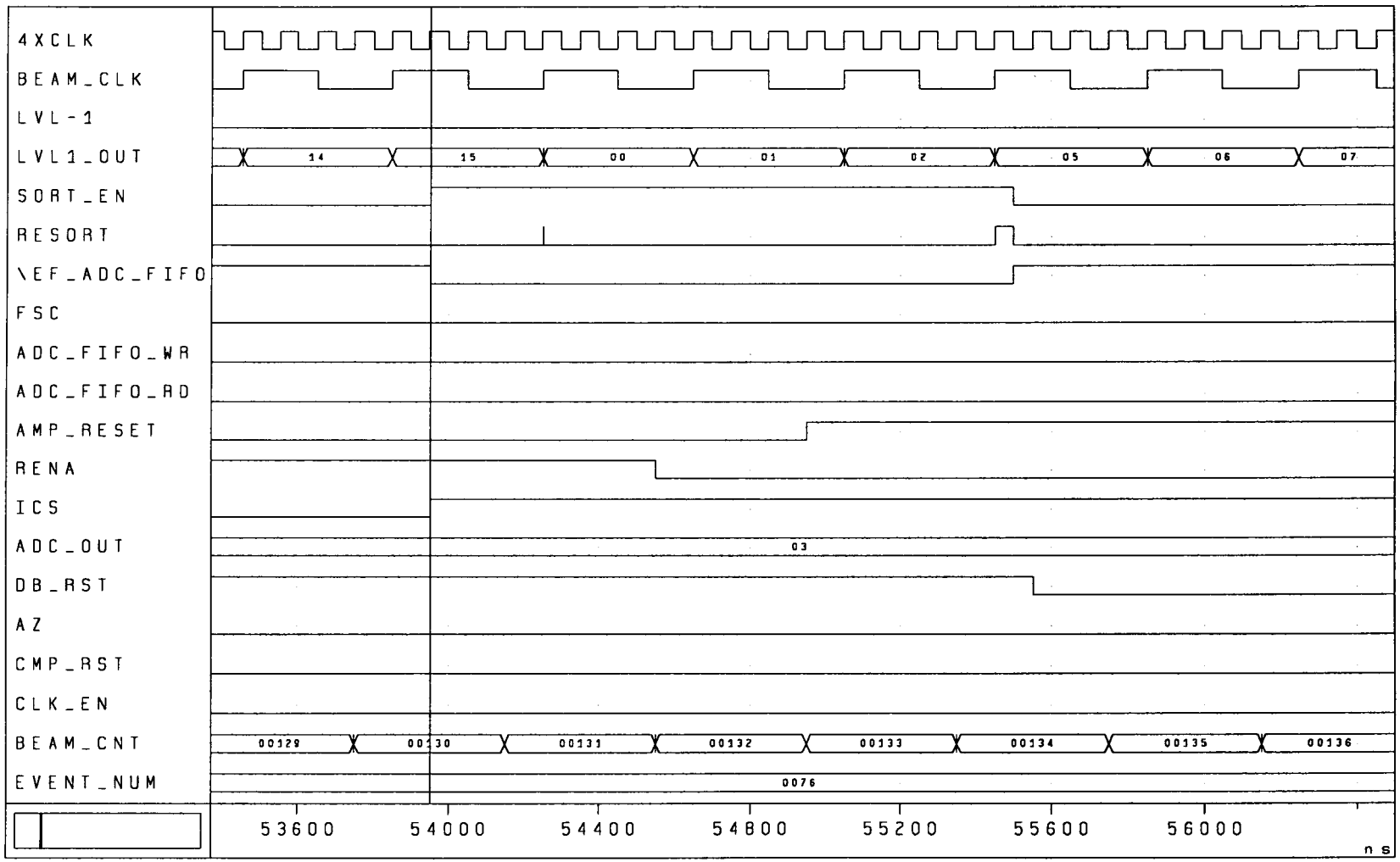


Figure 3.2.5. Re-sort Timing for the Heap Manager Controller

Table 3.2.8. Data Packet Format

Designation	Number Of 12-bit Words	Description
Header	2	Hex AAA Followed by Hex 555 Indicates Start of Packet
Event Number	1	12-bit Event Counter
Module Address	1	Contains Unique Module Address For Identification
AMU Cell #	1	Pre OR Post AMU Cell Addresses
Data Block	64	Raw Mode ((32 Pre OR 32 Post) X2)
Trailer	2	Hex AAA Followed by Hex 555 Indicates End Of Packet

read from the Beam Counter FIFO at the same time as the AMU read address, is used to indicate the relative position of the LVL-1s. Although it is not needed in the beam test, the Module Address specifies, for the final MVD system, one of the 256-channel detector modules; and the AMU Cell # gives the AMU address (**amu_raddr**) from which the ADC data is derived. Knowledge of the AMU Cell # permits the use of corrective measures which may be taken, based on the characteristics of a particular AMU, to remedy any pedestals or other errors which may be present for a given cell. The packet timing for the beginning of the packet readout sequence is expressed in Figure 3.2.6. When the readout is initiated, the write enable for the external FIFO (**pcfifo_w**) is asserted in the active LOW position; and each twelve-bit word is clocked into the PC FIFO (with **pcfifo_wclk**) or the CAMAC (with **WSI**).

The data block is constructed from both of the FEE mother boards and consists of the Pre or Post data read from the ADC as selected by the channel bus (**adc_chan**), output enable bus (**adc_oe**), and board enable bus (**oe_bd1** and **oe_bd2**). To coordinate the readout of each ADC on the FEE boards, the heap manager controller makes use of the tri-stated ADC output drivers and asserts only one of the eight output enables at a time; and, to select a particular board, the controller utilizes local tri-state drivers which are driven by the ADC data busses from each board and can be selected with the board enables, **oe_bd1** and **oe_bd2**. In Figure 3.2.6, the packet header, **event_num**, module address, and **amu_raddr** (**adc_out**) are written to the PC FIFO or CAMAC; and the output of the ADCs from the first FEE board are switched onto the **adc_data** bus with the **oe_bd1** signal. As the first FEE board is chosen, the heap manager controller engages the

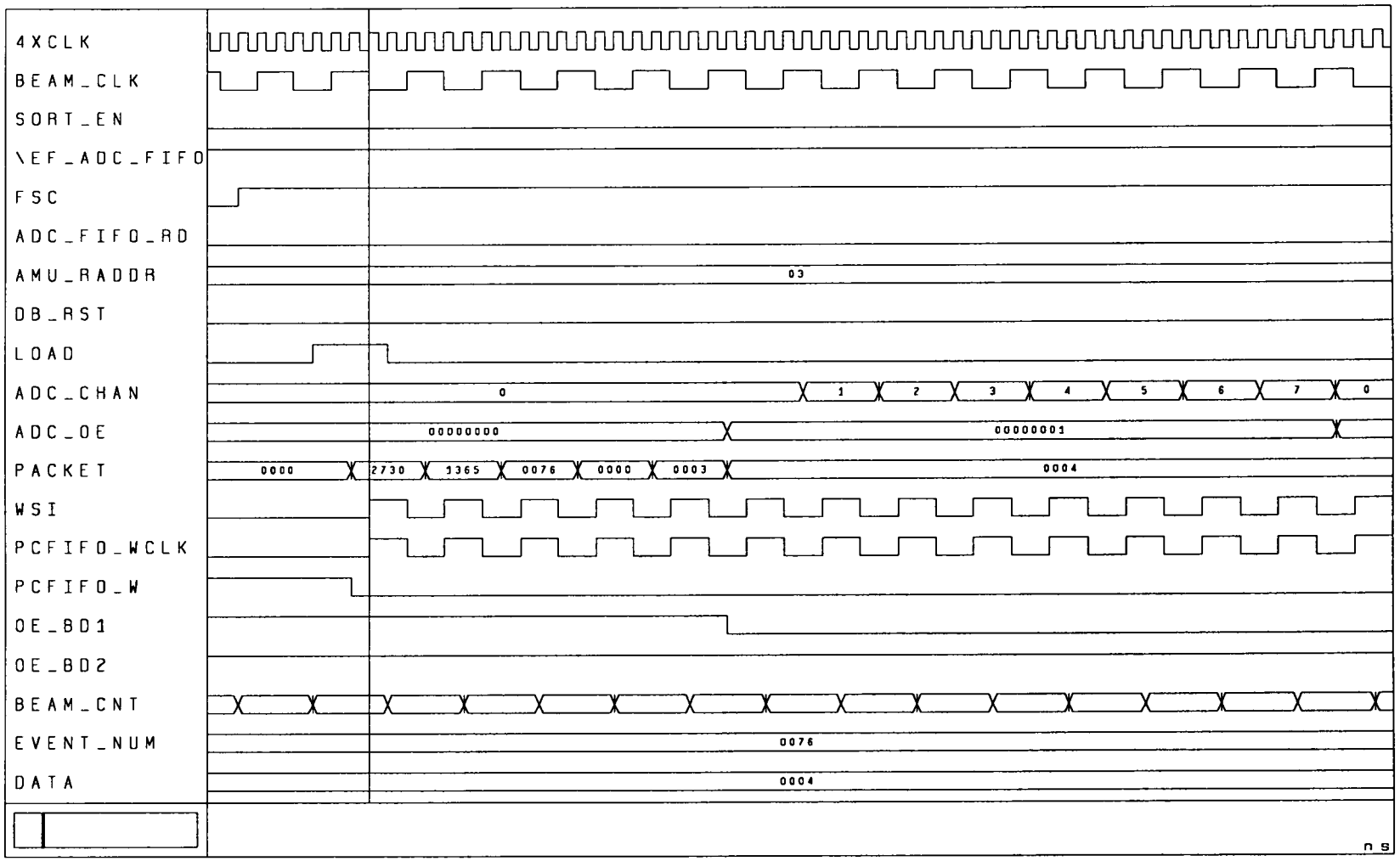


Figure 3.2.6. Header Timing for the Heap Manager Controller

ADC readout process by cycling through the eight channels of each ADC and shifts the **adc_oe** to another analog-to-digital converter when it finishes the last channel of a particular ADC. When it finishes reading the fourth ADC on the first FEE board, the heap manager controller disables **oe_bd1** and enables **oe_bd2** as seen in Figure 3.2.7. Finally, in Figure 3.2.8, the trailer data is written to the PC FIFO; and, when the readout ends, the digital circuitry of the ADC is reset by bringing the **db_rst** signal HIGH. For the simulations in Figure 3.2.6, Figure 3.2.7, and Figure 3.2.8, a static data word, 04, is used as the simulated ADC **data** input for the Data Packet MUX; therefore, in the testing phase of the MVD beam test system, the settling times of the ADCs must be measured to ensure that the correct data is stored in the PC FIFO.

The data readout function of the heap manager controller is implemented to read all ADCs on both FEE boards for each **amu_raddr** converted by the FEE state machine. Because a LVL-1 is produced as the result of a trigger sum calculated using all preamp summing outputs of a given module, the individual analog values stored in the AMUs are indistinguishable; therefore, all analog values stored in the eight AMUs at the address tagged by a LVL-1 must be converted. Separation of the data harvested from the ADCs takes place after the PC logs and stores the information to a file; and the data produced by the heap manager may then be analyzed to determine the detector strips affected by a particular physics event.

By coordinating the functions of the FEE and ALM with the commands expressed by the PC serial and control interfaces and the system LVL-1 trigger interface, the heap manager controller is able to return to the PC a packet of data which reflects the physics

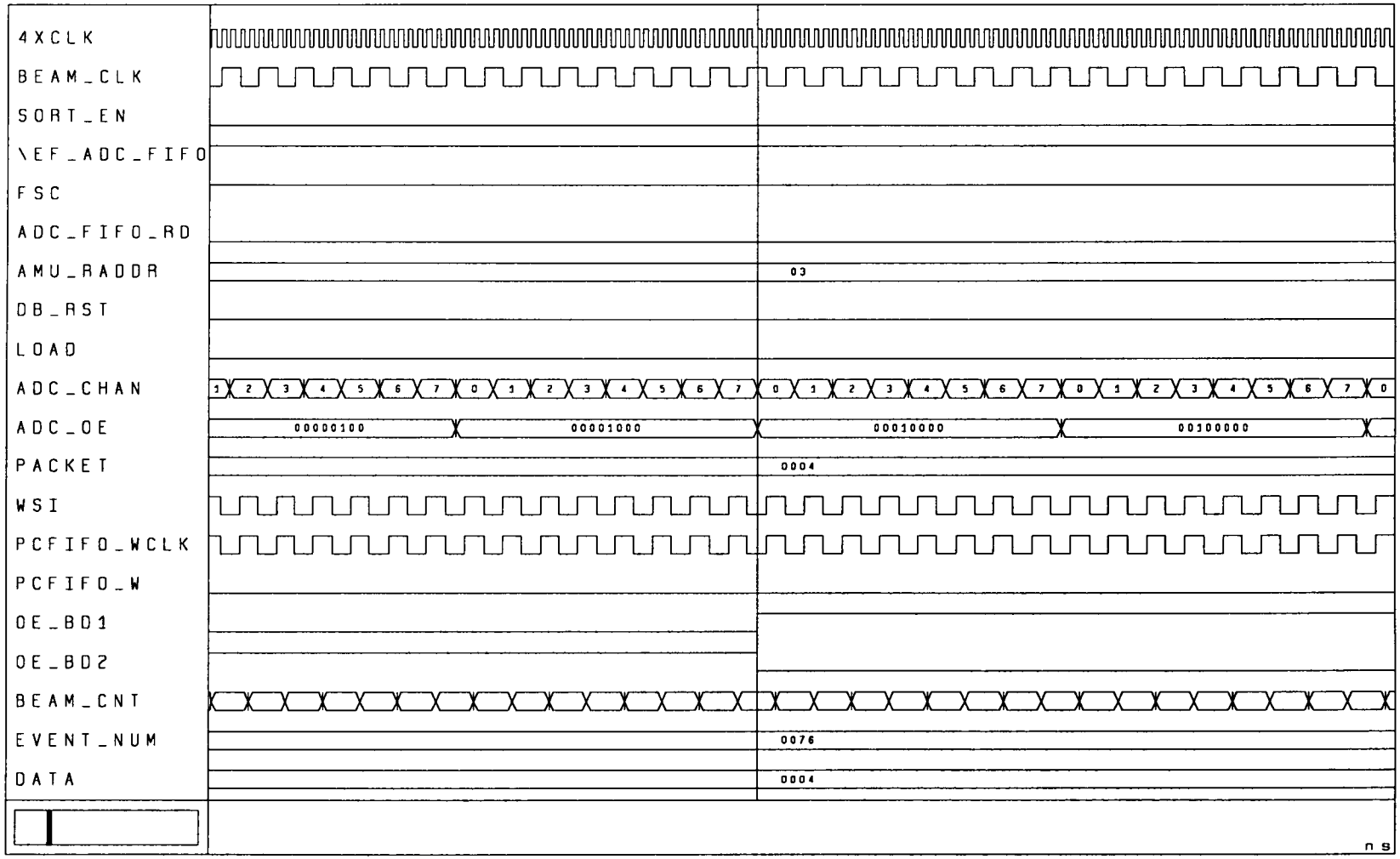


Figure 3.2.7. FEE Board Enable Timing for the Heap Manager Controller

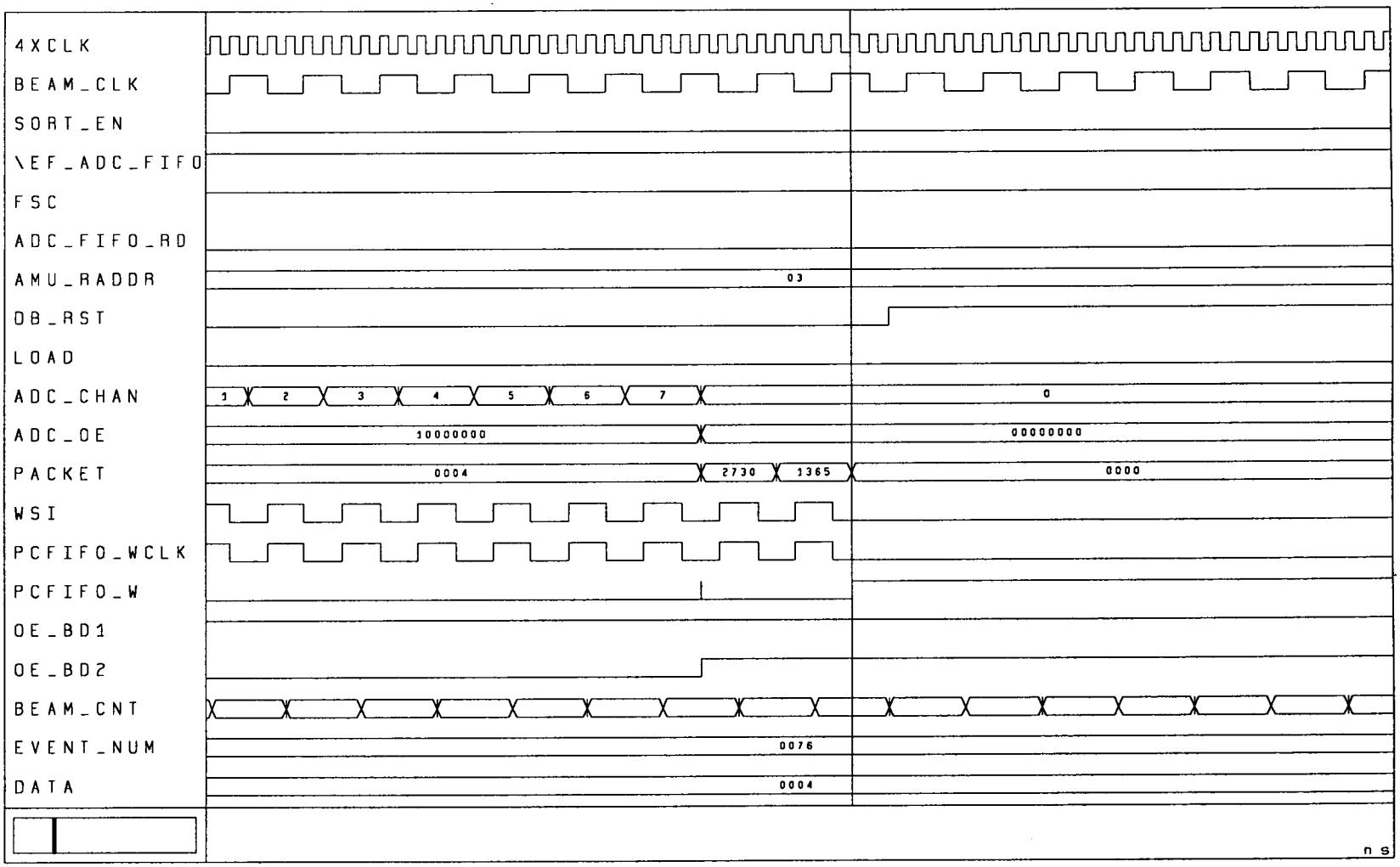


Figure 3.2.8. End-of-Packet Timing for the Heap Manager Controller

events taking place in the MVD beam test system while simultaneously managing the preamp, AMU, and ADC functions to minimize power consumption and increase the testability of the system. Utilizing the control interface, the computer instructs the heap manager controller in the fundamental operations of the system, such as Run and Halt. From the serial interface, the heap manager can define test points for the FEE, manipulate the preamp reset function, change the spacing of LVL-1 triggers, and control the discharge of the calibration circuit in the preamp. Furthermore, the heap manager controller coordinates its readout function with the front-end electronics and address list manager. With the aid of internal state machines that are optimized with special attention to the resets of the FEE and the re-sorting of addresses in the ALM, the heap manager exercises the digital controls of the FEE to affect the generation of digital data; and the packet formatting which the controller institutes allows the PC to filter the data and retrieve the cogent information from each event. Combining the wide variety of functions into a cohesive architecture, the heap manager controller brings the different components of the MVD beam test together to collect data which may be used to study the quark gluon plasma.

3.3. FPGA Implementation Issues

Implementation of the heap manager controller and address list manager is based on the need to balance resources available versus the resources required for constructing

an architecture which meets the speed and power constraints of the Multiplicity Vertex Detector beam test. Development of the heap manager, therefore, begins with the selection of the FPGA technology and specific Xilinx family. With the decision to separate the heap manager into a 5000-gate address list manager and a 10000-gate heap manager controller, the resources available are static; and the speed and power requirements of the designs are determined by the gate delays inherent in the Xilinx 4000 series and the clock frequency supplied to the parts. To achieve the best possible design, the Xilinx ProSeries software is utilized to optimize and compile the designs so that they meet the requirements laid out in the MVD specifications.

The speed of the address list manager corresponds to the maximum delay path for the addresses as they pass through the sorting multiplexers and the latches within the FIFOs. Since a re-sort condition produces an extra write cycle to the Available Address FIFO, setup times for the RAM and latches are critical; and floorplanning becomes a top priority. The Xilinx ProSeries software provides means to specify critical paths in the schematic layout of the design; and a hardware description language, VHDL, is employed to create the multiplexers utilized for sorting and initialization. A combination of the optimization options imbedded in the Xilinx software and the VHDL-based multiplexer blocks produces the minimized delay paths for the address list manager. Additionally, the Xilinx software provides a floorplanner which permits manual placement of the configurable logic blocks (CLBs) in the Xilinx architecture. In instances where the routing portion of the Xilinx software is unable to comply with design specifications, the floorplanner allows the designer to re-route portions of the layout which may coincide

with specific I/O pins. For the address list manager design, all possible avenues for improving the speed are explored and used in a variety of circumstances to produce the architecture with the best characteristics achievable under the fiscal constraints of the project.

For the heap manager controller, the only speed issue involves the readout of the data packet. Since throughput is a concern in MVD, the DCM interface must operate at the highest possible frequency. Although it provides a buffer between the data readout and the PC, the PC FIFO is not available to the CAMAC module; and the delays associated with data readout are translated into greater temporal delays between conversions in the presence of multiple LVL-1s. Consequently, the readout controller in the heap manager design is singled out for optimization using the Time Spec element in the XC4000 library. The Time Spec permits the designer to input the maximum acceptable delays for any number of elements in relation to specific inputs and outputs; therefore, the readout controller may be tagged so that all gates and flip-flops will operate at the **4xclk** speed. If the Xilinx design compiler is unable to meet the specifications of the Time Spec element, the logic is flagged; and the compiler does the best job possible in trying to come close to the specifications. For the heap manager controller, the readout function, which is relatively small compared to the number of gates available, can be closely regulated to meet the speed requirements of the beam test system.

Finally, of the components in the MVD system, the address list manager and heap manager controller present the greatest threat to the power specification of 15 mW per channel. Due to the programming elements and the enhanced transistors in the output

pads, the Xilinx parts require more power than the customized ASICs, which are designed to perform only a certain number of functions. Thus, the tradeoffs in the design of the heap manager involve the judicious usage of the clocks. Since the frequency with which gates and flip-flops switch is proportional to the power needed to sustain the functions, power can be reduced only by distributing the clocks in such a manner as to limit the speed for functions that do not require to be driven at higher rates. Unfortunately, the address list manager is speed intensive; so any attempt to curtail power via the slower clock is out of the question. For the heap manager controller, however, most functions do not demand the **4xclk** for synchronization. Hence, since the heap manager controller is optimized for the readout function only and the optimization process is easier if the **4xclk** drives a minimum number of gates, the use of the slower beam clock in secondary functions actually improves the performance of the controller. Fortunately, for the heap manager, the per channel power consumption is divided over sixty-four channels; and calculations made toward the power requirement are alleviated somewhat by the distribution of the heap manager functions over a wide array of FEE elements.

In the end, the power and speed of the heap manager are dictated by the selection of the Xilinx technology as the platform for the design. Although optimization of the architectures and clock distribution play roles in its successful implementation, the heap manager is constrained by the delays inherent in the floorplan of the individual FPGAs and the power required by the programming elements, CLBs, and I/O pads. The only reprieve for the heap manager in light of shortcomings in the areas of speed in power is the possibility of advancements in the Xilinx family of parts or the redesign of the heap

manager. Even though the simulations show that the heap manager is qualified to be implemented in the MVD beam test system, only through testing may the heap manager be recognized as successful.

Chapter 4

MVD System Prototype and Testing

The prototype testing of the beam test unit takes place in three phases. The ORNL phase of the test involves the quantification of the accuracy and functionality of the front-end electronics and the qualification of the heap manager, which is controlled by an IBM PC. The data logged during the ORNL test phase is meaningless from the standpoint of physics but does provide insight into the ability of the system to act as a cohesive unit. The second phase of the beam test takes place at Los Alamos National Laboratory (LANL) and involves the integration of the beam test unit with the CAMAC crate, which is unavailable at ORNL; and the LANL phase includes a preliminary trial with the system in the presence of a detector to make sure that the FEE can be coupled correctly with the silicon strips. Finally, the beam test takes place at Brookhaven National Laboratory (BNL) and integrates the prototype with the detector in the presence of an energy beam. Since the results of the BNL beam test are unpublished, the focus of this thesis is on the conclusions reached during the ORNL and LANL phases of development.

4.1. MVD Beam Test Electronics

The prototype for the MVD beam test consists of a heap manager board and two FEE mother boards, each of which contains a preamp baby board. The heap manager board consists of the two Xilinx FPGAs, a Xilinx XC4010-5PG191 for the heap manager controller and a Xilinx XC4005E-3PC84 for the address list manager, a ITD72205LB Integrated Device Technology 4096x18 CMOS SyncFIFO, and an array of drivers for the computer interfaces and FEE mother board interfaces. Each FEE mother board contains four eight-channel ADCs, four eight-channel sixteen-cell AMUs, and drivers for the interface to the heap manager board. Each preamp baby board has four eight-channel preamps, which are wire bonded to the printed circuit board. The system is controlled by a Gateway, IBM-compatible, PC 486 which is provided access to the board via two Digital I/O "cheesy" cards that are programmable with the C++ language. For the MVD beam test system, the only clock provided is the **4xclk**; and the beam clock is derived inside the heap manager FPGAs. The test inputs, in the absence of the detectors, are fed by a Lecroy waveform generator; and the ADC clock is generated by an 80 MHz crystal. Testing of the system involves both functional and quantitative tests; but only the functional tests are examined in this document since power, speed, and controllability are the important factors in the qualification of the heap manager.

The electronics for the beam test system are divided between a single heap manager board and two 32-channel FEE boards. System development at ORNL demands that an IBM PC 486 be used as a human interface for the issuance of commands to the

heap manager and as a means of data logging. A block diagram of the two FEE boards and the main control and data buses, relative to the heap manager and computer interface, is portrayed in Figure 4.1.1. The single line inputs to the MVD system include the two clocks, a 9.5 MHz beam clock, which is not used, and the 38 MHz **4xclk**, the LVL-1 trigger, the ADC clock, the two trigger sum outputs from the preamplifiers, the **Vcal** analog input for calibration of the preamplifiers, and the analog inputs from the detectors, all of which are entered into the system via Lemo sockets that accept shielded cable connectors. Connections between the FEE mother board and the heap manager board are implemented using 50-conductor ribbon cables. Testing of the system is made possible through the use of a logic analyzer and digital and analog oscilloscopes, which are used to probe the SPY MUX test points and the test points distributed on the heap manager board. As the boards are integrated into a unit, test trials of individual ASICs and the heap manager are made to ensure that each component is functional.

The computer interface between the IBM PC and the heap manager is arranged utilizing ribbon cables and two Digital I/O "cheesy" cards. The interface is structured to conform to the I/O requirements laid out for the heap manager, and management of the individual commands is left to the software. Since it is limited by the number of I/O it can source and sink, each Digital I/O card is configured to perform a specific subset of functions in tandem with the heap manager. Each Digital I/O card contains two ports, each with an input and output connector, as revealed in Figure 4.1.2. The ground signals of the Digital I/O cards are coupled with the digital grounds of the heap manger board through ten ohm resistors, and the inputs and outputs are used to drive and sink signals through

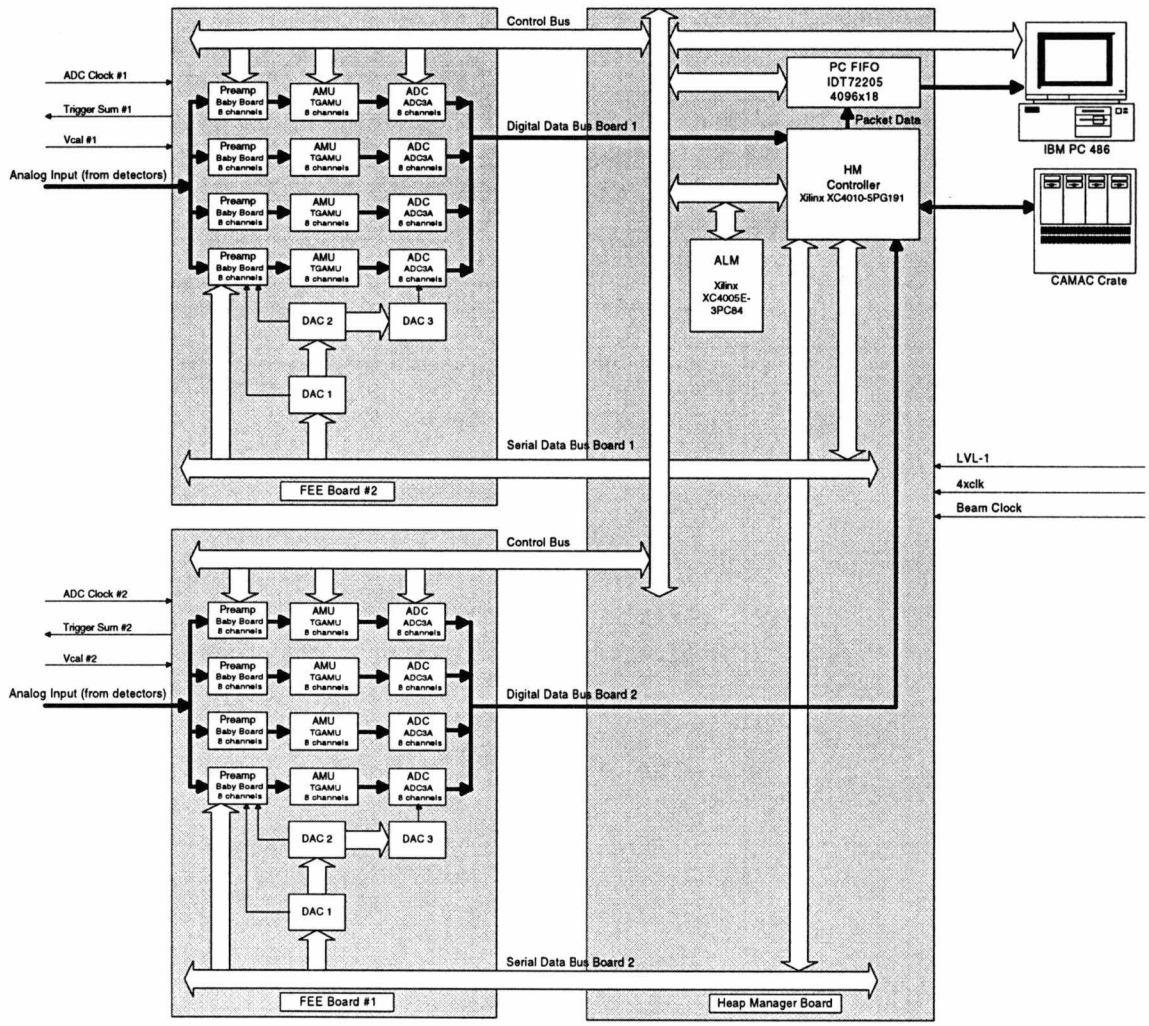


Figure 4.1.1. Block Diagram of the MVD Beam Test Prototype System

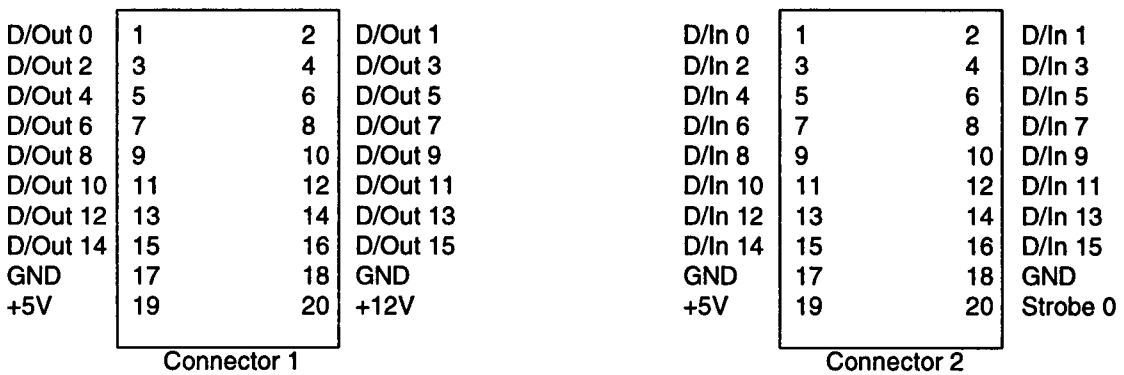


Figure 4.1.2. Digital I/O "Cheesy Card" Interface Ports

74ACT541 octal drivers which isolate the heap manager from the I/O cards to prevent a power surge from destroying the FPGAs. When the system is mated with the CAMAC crate, all of the signals, except the ones associated with the PC FIFO, are driven and received via MC10H125 ECL drivers, which are interfaced through four specialized, thirty-four-pin I/O ports and one ten-pin I/O port, and can be added or removed depending on the whether the crate is present. All computer interface signals are programmed through the PC or CAMAC to interact with the heap manager and provide the programming, serial, and data collection functions required to make the system work. As per the interface definitions for the heap manager controller, the signals of the PC interface are divided among the two Digital I/O cards; and the port and connector definitions are given in Table 4.1.1, Table 4.1.2, Table 4.1.3, Table 4.1.4, Table 4.1.5, and Table 4.1.6. The signals for the CAMAC are similar in composition to the PC interface, but are distributed as shown in Table 4.1.7, Table 4.1.8, Table 4.1.9, Table 4.1.10, and Table 4.1.11. Because it is driven by differential ECL, the CAMAC interface is defined in terms of differential pairs of signals, which require two connector pins for every individual control. As intimated by the connector definition tables, the I/O ports are divided into groups to facilitate the ingress of signals onto the heap manager printed circuit board (PCB) and to minimize crosstalk between different groups of signals.

Due to the large number of signals present in the beam test system and the reputation of digital signals for being noisy, precautions are taken to make sure that signal interference is kept at a minimum. As observed in Figure 4.1.3, the proximity of the FEE boards (on the right) to the heap manager board is minimized; and all signals intended to

Table 4.1.1. Output Port 1 of Digital I/O Card 1 for the PC Interface

Pin	Pin Name	Description
1	hm_din	HM shift register data
2	hm_sclk	HM shift register clock
3	hm_sloadb	HM mirror register load
4	xin1	extra pin (reserved)
5	pc_din	Xilinx programming data - HM controller
6	pc_cclk	Xilinx programming clock - HM controller
7	pc_prog	Xilinx programming configuration clear
8	cpu_rst	HM reset from PC
9	sr1_din	Preamp FEE Board #1 calibration data
10	sr1_sclk	Preamp FEE Board #1 calibration data clock
11	sr1_rst	Preamp FEE Board #1 calibration reset
12	xin2	extra pin (reserved)
13	sr2_din	Preamp FEE Board #2 calibration data
14	sr2_sclk	Preamp FEE Board #2 calibration data clock
15	sr2_rst	Preamp FEE Board #1 calibration reset
16	xin3	extra pin (reserved)

Table 4.1.2. Input Port 1 of Digital I/O Card 1 for the PC Interface

Pin	Pin Name	Description
1	dac1_dout	serial out of DAC C for board 1
2	dac2_dout	serial out of DAC C for board 2
3	hm_dout	serial out of HM shift register
4	xout1	extra pin (reserved)
5	xout2	extra pin (reserved)
6	xout3	extra pin (reserved)
7	xout4	extra pin (reserved)
8	xout5	extra pin (reserved)
9	pc_init	Xilinx configuration error flag - HM controller
10	pc_init2	Xilinx configuration error flag - ALM
11	done	Xilinx DONE bit - HM controller
12	done2	Xilinx DONE bit - ALM
13	NC	no connection
14	NC	no connection
15	NC	no connection
16	NC	no connection

Table 4.1.3. Input Port 2 of Digital I/O Card 1 for the PC Interface

Pin	Pin Name	Description
1	dac1_ldacb	DAC load for Board 1
2	dac1_din	DAC serial data for Board 1
3	dac1_sclk	DAC serial clock for Board 1
4	dac1_rstb	DAC reset for Board 1
5	dac2_ldacb	DAC load for Board 2
6	dac2_din	DAC serial data for Board 2
7	dac2_sclk	DAC serial clock for Board 2
8	dac2_rstb	DAC reset for Board 2
9	mb0	Mode Bit 0 - HM controller
10	mb1	Mode Bit 1 - HM controller
11	mb2	Mode Bit 2 - HM controller
12	mb3	Mode Bit 3 - HM controller
13	mb4	Mode Bit 4 - HM controller
14	xo4	extra pin (reserved)
15	xo5	extra pin (reserved)
16	xo6	extra pin (reserved)

Table 4.1.4. Output Port 1 of Digital I/O Card 2 for the PC Interface

Pin	Pin Name	Description
1	pcfifo_r	PC FIFO read enable
2	pcfifo_rclk	PC FIFO read clock
3	pcfifo_rs	PC FIFO reset
4	pcfifo_load	PC FIFO load
5	dac1_csb	DAC FEE Board #1 chip selects
6	dac2_csb	DAC FEE Board #2 chip selects
7	pc_din2	Xilinx programming data - ALM
8	pc_cclk2	Xilinx programming clock - ALM
9	NC	no connection
10	NC	no connection
11	NC	no connection
12	NC	no connection
13	NC	no connection
14	NC	no connection
15	NC	no connection
16	NC	no connection

Table 4.1.5. Input Port 1 of Digital I/O Card 2 for the PC Interface

Pin	Pin Name	Description
1	Packet Data 0	data from PC FIFO (bit 1 of 16)
2	Packet Data 1	data from PC FIFO (bit 2 of 16)
3	Packet Data 2	data from PC FIFO (bit 3 of 16)
4	Packet Data 3	data from PC FIFO (bit 4 of 16)
5	Packet Data 4	data from PC FIFO (bit 5 of 16)
6	Packet Data 5	data from PC FIFO (bit 6 of 16)
7	Packet Data 6	data from PC FIFO (bit 7 of 16)
8	Packet Data 7	data from PC FIFO (bit 8 of 16)
9	Packet Data 8	data from PC FIFO (bit 9 of 16)
10	Packet Data 9	data from PC FIFO (bit 10 of 16)
11	Packet Data 10	data from PC FIFO (bit 11 of 16)
12	Packet Data 11	data from PC FIFO (bit 12 of 16)
13	Packet Data 12	data from PC FIFO (bit 13 of 16)
14	Packet Data 13	data from PC FIFO (bit 14 of 16)
15	Packet Data 14	data from PC FIFO (bit 15 of 16)
16	Packet Data 15	data from PC FIFO (bit 16 of 16)

Table 4.1.6. Input Port 2 of Digital I/O Card 2 for the PC Interface

Pin	Pin Name	Description
1	pcfifo_ff	PC FIFO full flag
2	pcfifo_paf	PC FIFO almost full flag
3	pcfifo_hf	PC FIFO half full flag
4	pcfifo_pae	PC FIFO almost empty flag
5	pcfifo_ef	PC FIFO empty flag
6	NC	no connection
7	NC	no connection
8	NC	no connection
9	NC	no connection
10	NC	no connection
11	NC	no connection
12	NC	no connection
13	NC	no connection
14	NC	no connection
15	NC	no connection
16	NC	no connection

Table 4.1.7. 34-Bit I/O Port 1 for the CAMAC CPU Interface

Pins	Pin Name	Description
1,2	hm_din	HM shift register data
3,4	hm_sclk	HM shift register clock
5,6	hm_sloadb	HM mirror register load
7,8	xin1	reserved input
9,10	sr1_din	Preamp FEE Board #1 calibration data
11,12	sr1_sclk	Preamp FEE Board #1 calibration data clock
13,14	sr1_rst	Preamp FEE Board #1 calibration reset
15,16	xin2	reserved input
17,18	sr2_din	Preamp FEE Board #2 calibration data
19,20	sr2_sclk	Preamp FEE Board #2 calibration data clock
21,22	sr2_rst	Preamp FEE Board #2 calibration reset
23,24	xin3	reserved input
25,26	N/C	no connection
27,28	N/C	no connection
29,30	N/C	no connection
31,32	N/C	no connection
33,34	N/C	no connection

Table 4.1.8. 34-Bit I/O Port 2 for the CAMAC CPU Interface

Pins	Pin Name	Description
1,2	dac1_ldacb	DAC load for Board 1
3,4	dac1_din	DAC serial data for Board 1
5,6	dac1_sclk	DAC serial clock for Board 1
7,8	dac1_rstb	DAC reset for Board 1
9,10	dac2_ldacb	DAC load for Board 2
11,12	dac2_din	DAC serial data for Board 2
13,14	dac2_sclk	DAC serial clock for Board 2
15,16	dac2_rstb	DAC reset for Board 2
17,18	dac1_dout	serial out of DAC3 for Board 1
19,20	dac2_dout	serial out of DAC3 for Board 2
21,22	hm_dout	serial out of HM shift register
23,24	done	Xilinx DONE bit - HM controller
25,26	done2	Xilinx DONE bit - ALM
27,28	dac1_csb	DAC chip select for Board 1
29,30	dac2_csb	DAC chip select for Board 2
31,32	xout5	reserved output
33,34	N/C	no connection

Table 4.1.9. 34-Bit I/O Port 3 for the CAMAC CPU Interface

Pins	Pin Name	Description
1,2	mb0	Mode Bit 0 - HM controller
3,4	mb1	Mode Bit 1 - HM controller
5,6	mb2	Mode Bit 2 - HM controller
7,8	mb3	Mode Bit 3 - HM controller
9,10	mb4	Mode Bit 4 - HM controller
11,12	pc_din	Xilinx programming data - HM controller
13,14	pc_cclk	Xilinx programming clock - HM controller
15,16	pc_prog	Xilinx programming configuration clear
17,18	pc_din2	Xilinx programming data - ALM
19,20	pc_cclk2	Xilinx programming clock - ALM
21,22	N/C	no connection
23,24	N/C	no connection
25,26	N/C	no connection
27,28	N/C	no connection
29,30	N/C	no connection
31,32	N/C	no connection
33,34	N/C	no connection

Table 4.1.10. 34-Bit I/O Port 4 for the CAMAC CPU Interface

Pins	Pin Name	Description
1,2	Packet Data 0	data from PC FIFO (bit 1 of 16)
3,4	Packet Data 1	data from PC FIFO (bit 2 of 16)
5,6	Packet Data 2	data from PC FIFO (bit 3 of 16)
7,8	Packet Data 3	data from PC FIFO (bit 4 of 16)
9,10	Packet Data 4	data from PC FIFO (bit 5 of 16)
11,12	Packet Data 5	data from PC FIFO (bit 6 of 16)
13,14	Packet Data 6	data from PC FIFO (bit 7 of 16)
15,16	Packet Data 7	data from PC FIFO (bit 8 of 16)
17,18	Packet Data 8	data from PC FIFO (bit 9 of 16)
19,20	Packet Data 9	data from PC FIFO (bit 10 of 16)
21,22	Packet Data 10	data from PC FIFO (bit 11 of 16)
23,24	Packet Data 11	data from PC FIFO (bit 12 of 16)
25,26	Packet Data 12	data from PC FIFO (bit 13 of 16)
27,28	Packet Data 13	data from PC FIFO (bit 14 of 16)
29,30	Packet Data 14	data from PC FIFO (bit 15 of 16)
31,32	Packet Data 15	data from PC FIFO (bit 16 of 16)
33,34	N/C	no connection

Table 4.1.11. 10-Bit I/O Port 5 for the CAMAC CPU Interface

Pins	Pin Name	Description
1,2	WSI	data packet clock
3,4	N/C	no connection
5,6	N/C	no connection
7,8	N/C	no connection
9,10	N/C	no connection

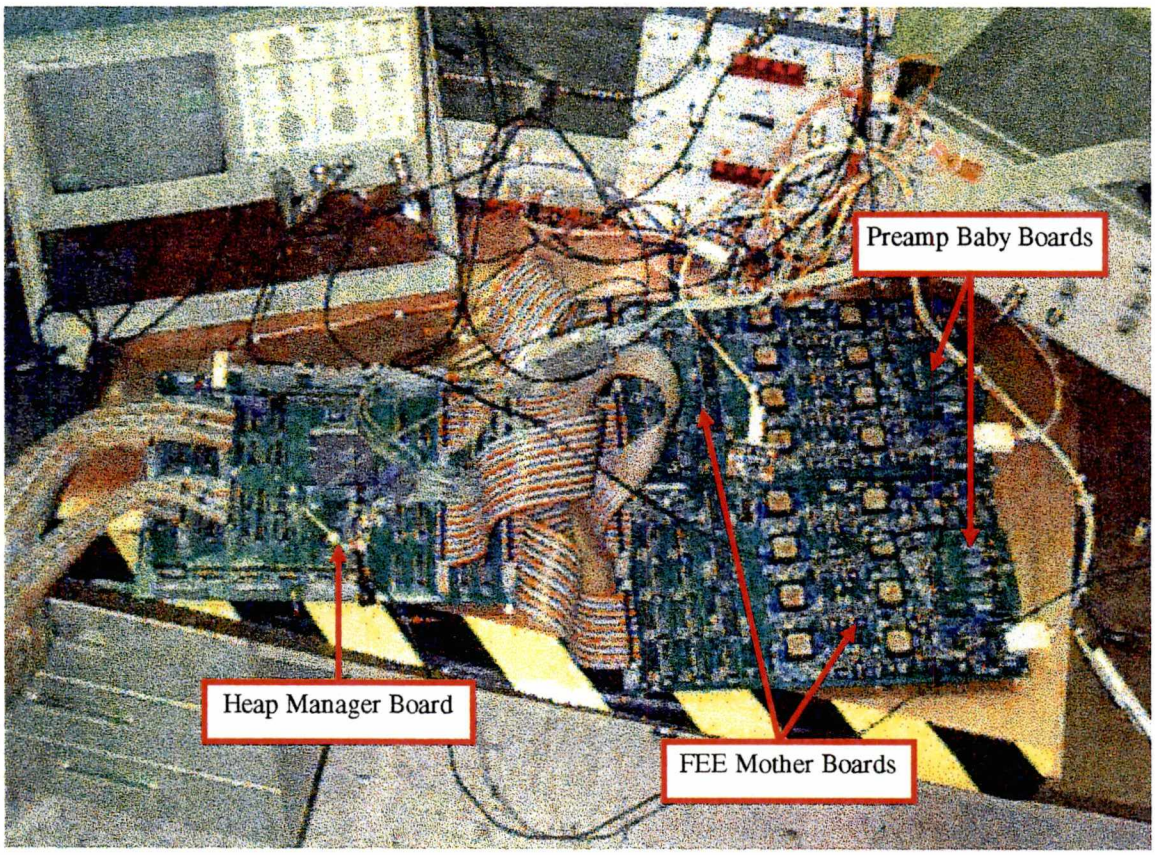


Figure 4.1.3. Photograph of the 64-channel MVD Beam Test Prototype

Source: [http://phnxmu.lanl.gov/Files/MVD/beamtest/setup_to.jpg].

pass through the ribbon cables are first strengthened with 74ACT541 octal drivers which maintain signal-level integrity. The transmitting and receiving drivers between the FEE mother board and heap manager board are terminated with 1000-ohm resistors, and the ribbon cables are cut as short as possible to prevent crosstalk. On critical lines, such as the **fsc** signal from the ADC, 100-ohm resistors are put in series with the signals to slow down the response time of the switching edges and prevent glitches from influencing operation of the heap manager; and, inside the heap manager controller, latches are used for signals such as the **fsc** and serial interface clocks and data lines to prevent glitches from prematurely triggering operations in the heap manager. Since the signal traces on the printed circuit boards are close together and the passage of signals through ribbon cable tends to produce crosstalk, coordination of termination and filtering techniques is critical to ensure that the correct operations are performed by the heap manager.

The management of noise is enhanced further by the inclusion of voltage regulators on the front-end electronics boards. Each FEE board contains a set of regulators which maintain the voltage levels that supply power to the analog components of the preamps, analog memory units, and analog-to-digital converters. Although they are specified as having low noise characteristics, the primary power supplies used in the beam test unit must entered onto the board through twisted wires which act as antennas; and the broad distribution of power on the board results in noise being coupled between analog and digital traces which, in turn, influence the power and ground lines. Although the FEE boards and the heap manager board are six-layer printed circuit boards which have separate ground and power planes for analog and digital signals, the on-board regulators

help reduce the coupling of analog and digital supply lines and the noise associated with each. Lastly, to get rid of high-frequency noise on the power supply lines, 0.1 microFarad by-pass capacitors are used on all boards for each individual chip to aid the filtering effects of the regulators and to get rid of noise coupled onto the network of traces between the power supplies and the chips.

In the end, noise cannot be eliminated. Regulation of power supplies, filtering, termination, and management of PCB resources alleviate some of the problems associated with unwanted signals in the MVD beam test system; but the analysis of measurements for the system must take into account the presence of noise. Unfortunately, due to the large number of analog and digital components and the distances which the signals must travel between boards and through the closely-packed traces on the printed circuit boards, crosstalk and noise from external instrumentation, such as the PC, is omnipresent; and testing of the integrated beam test unit must be enacted to gauge the severity of the noise as well as the validity of the data produced by the system. Ultimately, to be successful, the Multiplicity Vertex Detector will have to take advantage of the proximity of the electronics to the detector and to each other, in the MCM, to rid the system of the most detrimental noise components.

4.2. Software Development

The software for the MVD beam test prototype is developed for an IBM PC 486 computer which is used during the Oak Ridge National Laboratory (ORNL) test phase and serves as the paradigm for code development at Los Alamos National Laboratory (LANL) for the final CAMAC-based beam test system. Written in the C++ language, the code for the MVD beam test, provided in Appendix A, embodies as many permutations of the system commands as possible to enhance the flexibility of the system and provide subroutines which may be used for the CAMAC code.

The main menu for the MVD beam test software provides access to the major central processing unit (CPU) interfaces: Mode Control Interface (heap manager control interface), DCM (data collection module) Interface, Serial Interface (heap manager serial interface), and Xilinx Configuration Interface. The main menu, illustrated in Figure 4.2.1, is the first interface which the user encounters upon execution of the beam test program. Besides providing a path to the interface menus, the main menu evinces the ability to reset the Digital I/0 "cheesy" cards (option 5), which results in all digital outputs to the heap manager board being returned to ground. Used for shut down, the "Reset Cheesy Cards" option is exercised before the heap manger board is powered down to prevent the interface drivers from being harmed by voltage and current output from the computer. Also, the main menu offers option 6 to return the system to the initialization state, in which the outputs of the Digital I/O cards are set so the heap manager is inactive. From the main menu, the hierarchical format of the program branches to permit the user to

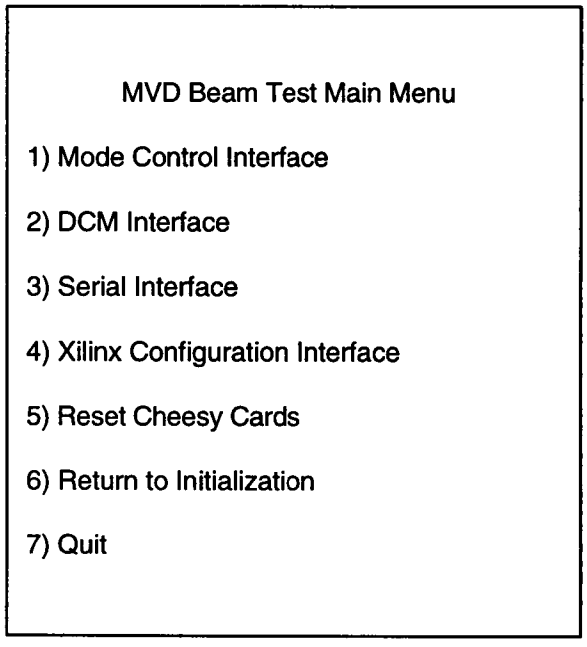


Figure 4.2.1. MVD Beam Test Main Menu

quickly find any of the commands in the heap manager vocabulary.

Of the interface options, the Mode Control Interface menu provides a path to the commands specified by the mode bits of the heap manager controller. As guaranteed in Figure 4.2.2, the Mode Control Interface has an option for each mode command. In addition to setting the mode bits, the Mode Control Interface has permutations of the Run command which allow the user to initiate LVL-1 triggers from the CPU for diagnostic testing. Option (4) allows the user to generate a single LVL-1 trigger, and option (5) causes the program to generate LVL-1s, at intervals greater than sixteen beam clocks, until the user hits a key and returns the program to the Mode Control Interface menu. Option (8) controls the third mode bit and performs a single pulse which resets the heap manager controller and address list manager. Each option in the Mode Control Interface conforms to the timing specified in Chapter 3.

The data collection module, or DCM Interface, provides the user with the ability to read data from the PC FIFO on the heap manager board. While the heap manager controller has access to the write functions of the PC FIFO, the CPU interface provides the read functions to the PC FIFO. The screen print of the DCM Interface menu is presented in Figure 4.2.3, and each of the menu items represented in the figure is related to FIFO functions needed for data acquisition and data logging. Option (1) of the DCM Interface instructs the program to get a single packet. To read the FIFO, the computer first tests the empty flag of the FIFO; and, if the empty flag indicates that data is present, the computer enables the FIFO read by lowering the read enable (**pcfifo_r**) and pulses the read clock (**pcfifo_rclk**), thus popping the first data word onto the data bus. Since it is

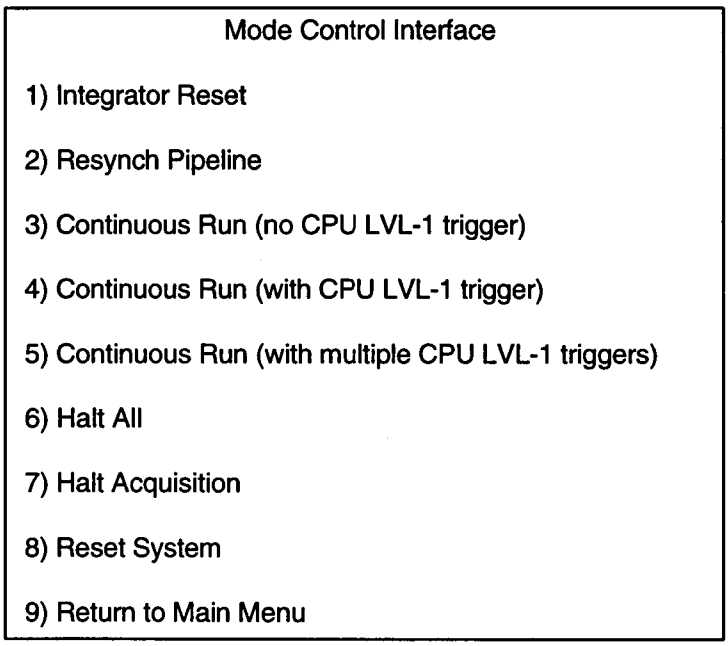


Figure 4.2.2. Mode Control Interface Menu

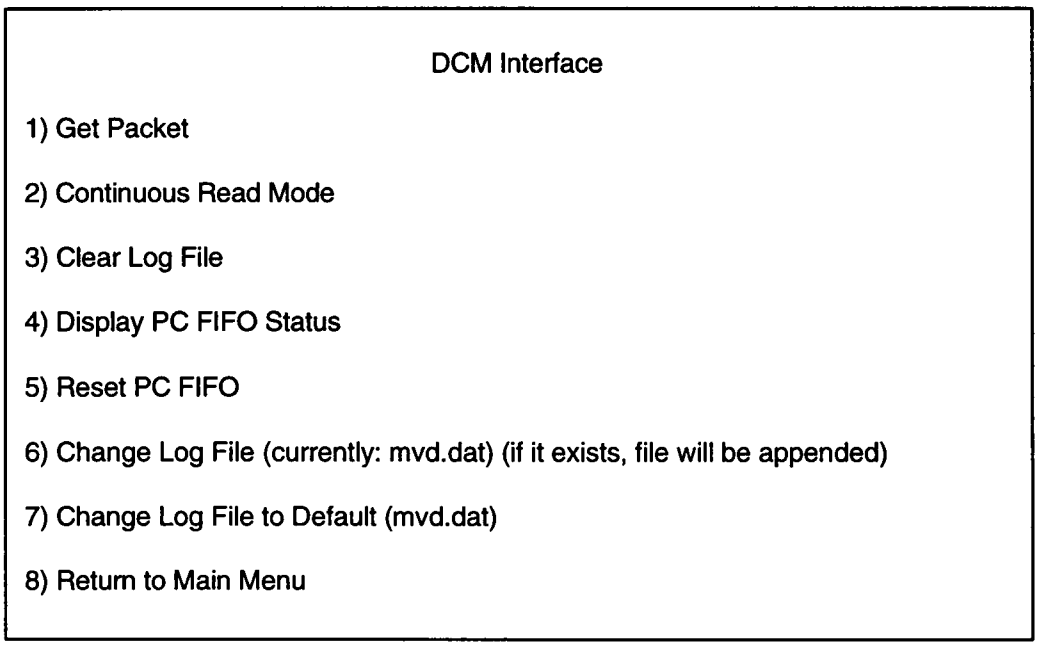
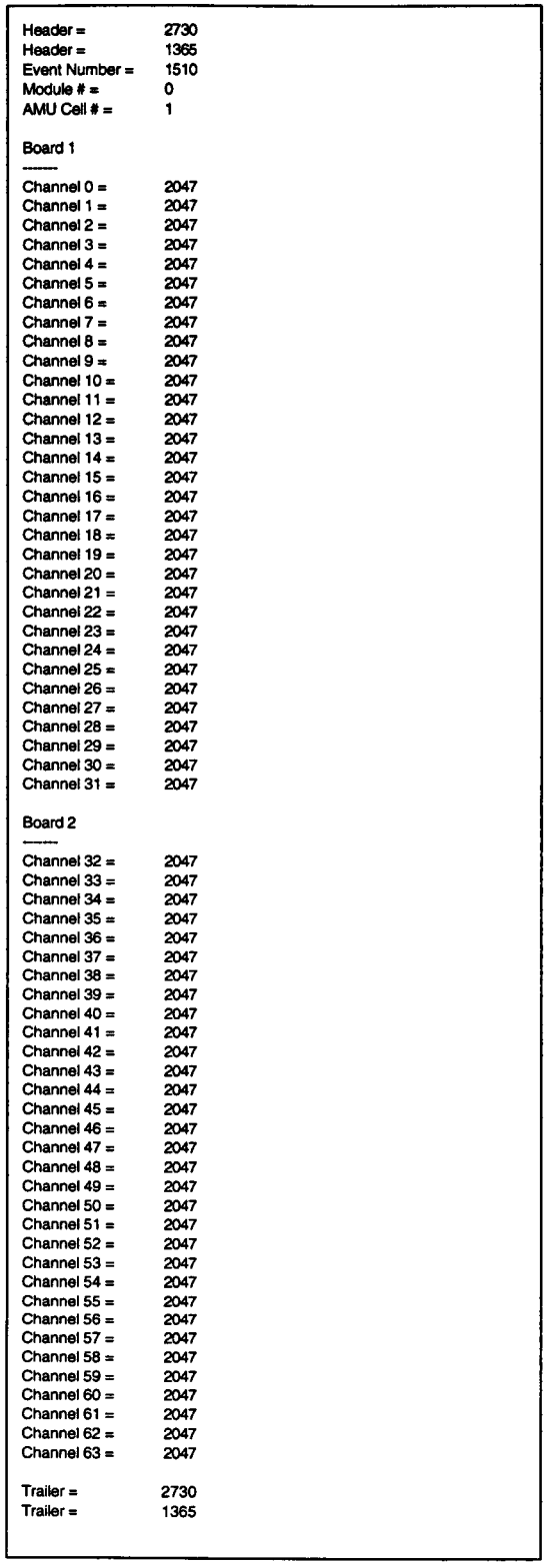


Figure 4.2.3. DCM Interface Menu

programmed to receive seventy-one words of data, or one packet, each time it reads the FIFO, the computer, if it does not sense that the empty flag is LOW for a particular piece of data, replaces the data with the decimal value 9999 to indicate that no data is present, thus precluding a lock-up, or infinite loop, situation. Because the controller is much faster at adding data to the FIFO than the CPU is at reading the FIFO, a null data value indicates that an error has occurred in the heap manager controller; however, a successful read of a data packet produces data which is written to the screen as well as to a log file. The screen print for a typical packet is given in Figure 4.2.4. Although irrelevant, the data represented in Figure 4.2.4 demonstrates the successful retrieval of a packet and the separation of the bundled data and data tagging words. When it is exercised, option (2), Continuous Read Mode, utilizes the empty flag of the FIFO to detect data; and, as data is collected, a printout of the packet is given every seventy-one words. By utilizing option (2), the user may set up an intermittent, external LVL-1 trigger and monitor the data on the screen to perform diagnostics on the system.

Options (3) through (7) of the DCM Interface provide FIFO status monitoring and log file modifications. Option (3) resets the log file pointer in the program and effectively erases all data in the current data log file. Option (4) displays, to the screen, the status of the empty flag, full flag, almost full flag, and almost empty flag of the PC FIFO so that the user may determine if data is being written successfully to the PC FIFO. Option (5) allows the user to clear all data from the PC FIFO and begin a new test run. Also included in the DCM Interface menu is the ability to change the log file name or return to the default (mvd.dat) file, operations which append data to the specified file, if it exists, or else create



```
Header =          2730
Header =          1365
Event Number =    1510
Module # =        0
AMU Cell # =      1

Board 1
-------
Channel 0 =       2047
Channel 1 =       2047
Channel 2 =       2047
Channel 3 =       2047
Channel 4 =       2047
Channel 5 =       2047
Channel 6 =       2047
Channel 7 =       2047
Channel 8 =       2047
Channel 9 =       2047
Channel 10 =      2047
Channel 11 =      2047
Channel 12 =      2047
Channel 13 =      2047
Channel 14 =      2047
Channel 15 =      2047
Channel 16 =      2047
Channel 17 =      2047
Channel 18 =      2047
Channel 19 =      2047
Channel 20 =      2047
Channel 21 =      2047
Channel 22 =      2047
Channel 23 =      2047
Channel 24 =      2047
Channel 25 =      2047
Channel 26 =      2047
Channel 27 =      2047
Channel 28 =      2047
Channel 29 =      2047
Channel 30 =      2047
Channel 31 =      2047

Board 2
-------
Channel 32 =      2047
Channel 33 =      2047
Channel 34 =      2047
Channel 35 =      2047
Channel 36 =      2047
Channel 37 =      2047
Channel 38 =      2047
Channel 39 =      2047
Channel 40 =      2047
Channel 41 =      2047
Channel 42 =      2047
Channel 43 =      2047
Channel 44 =      2047
Channel 45 =      2047
Channel 46 =      2047
Channel 47 =      2047
Channel 48 =      2047
Channel 49 =      2047
Channel 50 =      2047
Channel 51 =      2047
Channel 52 =      2047
Channel 53 =      2047
Channel 54 =      2047
Channel 55 =      2047
Channel 56 =      2047
Channel 57 =      2047
Channel 58 =      2047
Channel 59 =      2047
Channel 60 =      2047
Channel 61 =      2047
Channel 62 =      2047
Channel 63 =      2047

Trailer =         2730
Trailer =         1365
```

Figure 4.2.4. Data Packet Printout from the DCM Interface

a new log file. Of the menus in the MVD beam test program, the DCM Interface is the only menu which has no bearing on the operation of the heap manager.

The Serial Interface consists of the heap manager serial functions, the DAC serial interface, and the serial data for preamp calibration. The Serial Interface menu, shown in Figure 4.2.5, contains the options for editing the functions of the heap manager serial interfaces. Options (1) through (4) pertain to the definition of the eighteen heap manager serial string bits; and, when any of the first four options is executed, the entire serial string is modified and re-loaded into the heap manager controller. After they are clocked into the heap manager controller, all eighteen bits of the serial string are loaded into the mirror register via **hm_sloadb**. Option (1) of the Serial Interface allows the user to set the frequency of the integrator reset during Run mode, as evinced in Figure 4.2.6. Then, as shown in Figure 4.2.7, option (2) sources bits 2 through 13 of the heap manager serial string to permit change to any or all of the bits related to the SPY multiplexers. Option (3), used to set the Raw/Correlator Mode of the MVD system, is disabled for the beam test and defaults to Raw Mode. Option (4), the last of the heap manager serial string modifiers and presented in Figure 4.2.8, lets the user determine the spacing of the LVL-1 triggers in raw mode; therefore, for each LVL-1 which is received by the heap manager controller, two LVL-1s, spaced by the number of beam clocks specified with option (4), are generated. Thus, by exercising one of the first four options of the Serial Interface, the user can change all of the commands, with the exception of **cal_dis**, associated with the heap manager serial string.

For the sake of keeping similar commands together, option (5) of the Serial

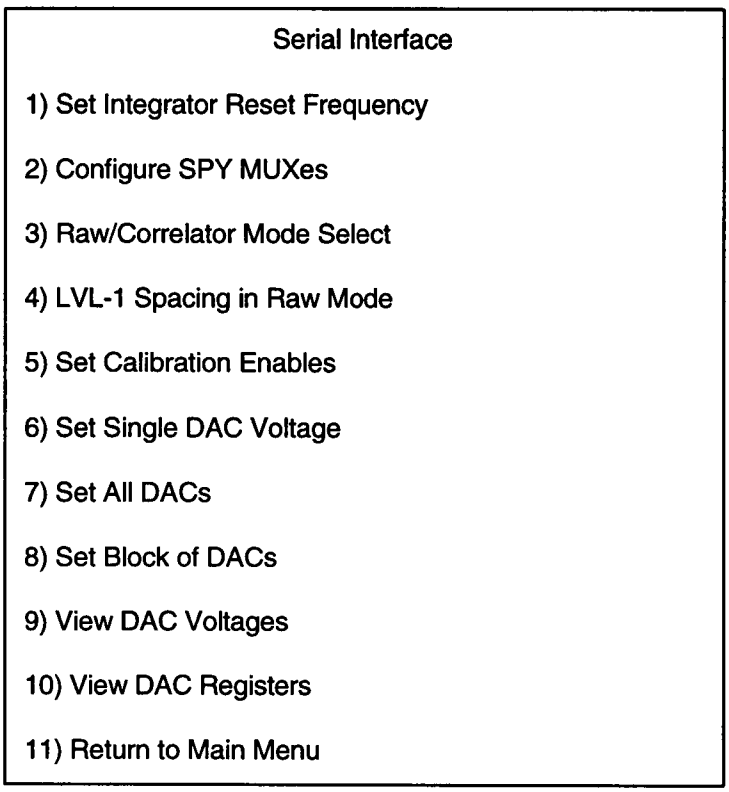


Figure 4.2.5. Serial Interface Menu

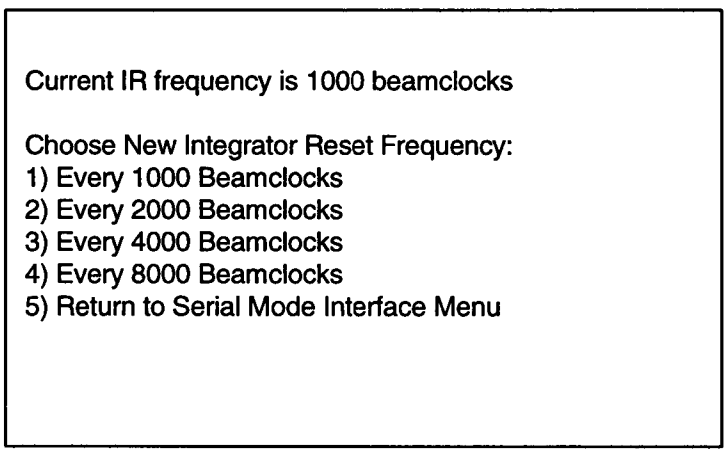


Figure 4.2.6. Option (1) of the Serial Interface Menu

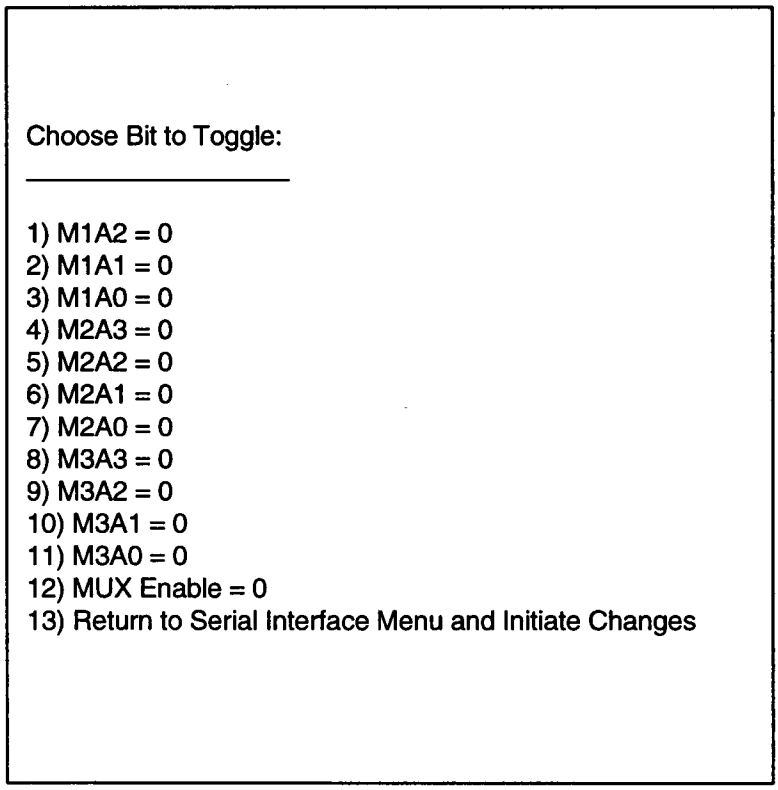


Figure 4.2.7. Option (2) of the Serial Interface Menu

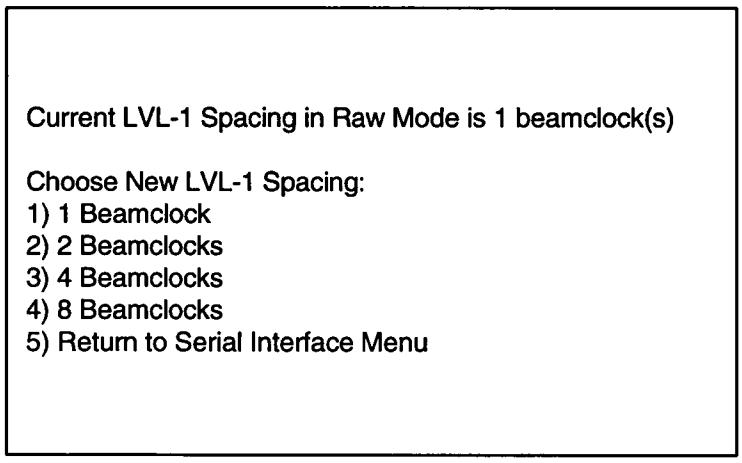


Figure 4.2.8. Option (4) of the Serial Interface Menu

Interface menu is designed to include **cal_dis**. Thus, the preamp serial options actually span two serial command streams since the **cal_dis** signal resides in the heap manager serial string. Option (5) of the Serial Interface menu, captured in Figure 4.2.9, allows the user to change any one of sixteen calibration enables for the preamps as well as toggle the calibration enable or **cal_dis**. Because the calibration serial strings are shared between all preamps on a single board, only eight calibration bits of data need to be set to specify the state of calibration for all channels on a given FEE board. When the calibration bits are set as desired, the calibration enable (option (19) in Figure 4.2.9) may be toggled to initiate the shifting of the calibration enables into the preamps; and **cal_dis** can used to discharge the preamp calibration voltage set by **Vcal** on the FEE boards. For the qualification of the beam test system, the calibration menu is utilized to set up the preamps so that known analog data, in lieu of the detector outputs, may be pumped into the electronics.

The DAC serial string option of the Serial Interface permits each of the twenty-four DAC channels to be set independently or concurrently as needed. Option (6) of the Serial Interface, presented in Figure 4.2.10, provides access to the menu choices which allow any one DAC channel to be set to a particular voltage by permitting the user to input a binary value, 0-255, for the eight-bit DAC. Also available are option (7) and option (8), shown in Figure 4.2.11 and Figure 4.2.12, respectively, which facilitate the modification of several DAC channels at one time, either with a common voltage level, such as half scale, or with a value selected by the user. After choosing one of the DAC modification options, the user may view either the DAC voltages or the actual binary serial string loaded in the DAC serial registers by utilizing option (7) or option (8), respectively.

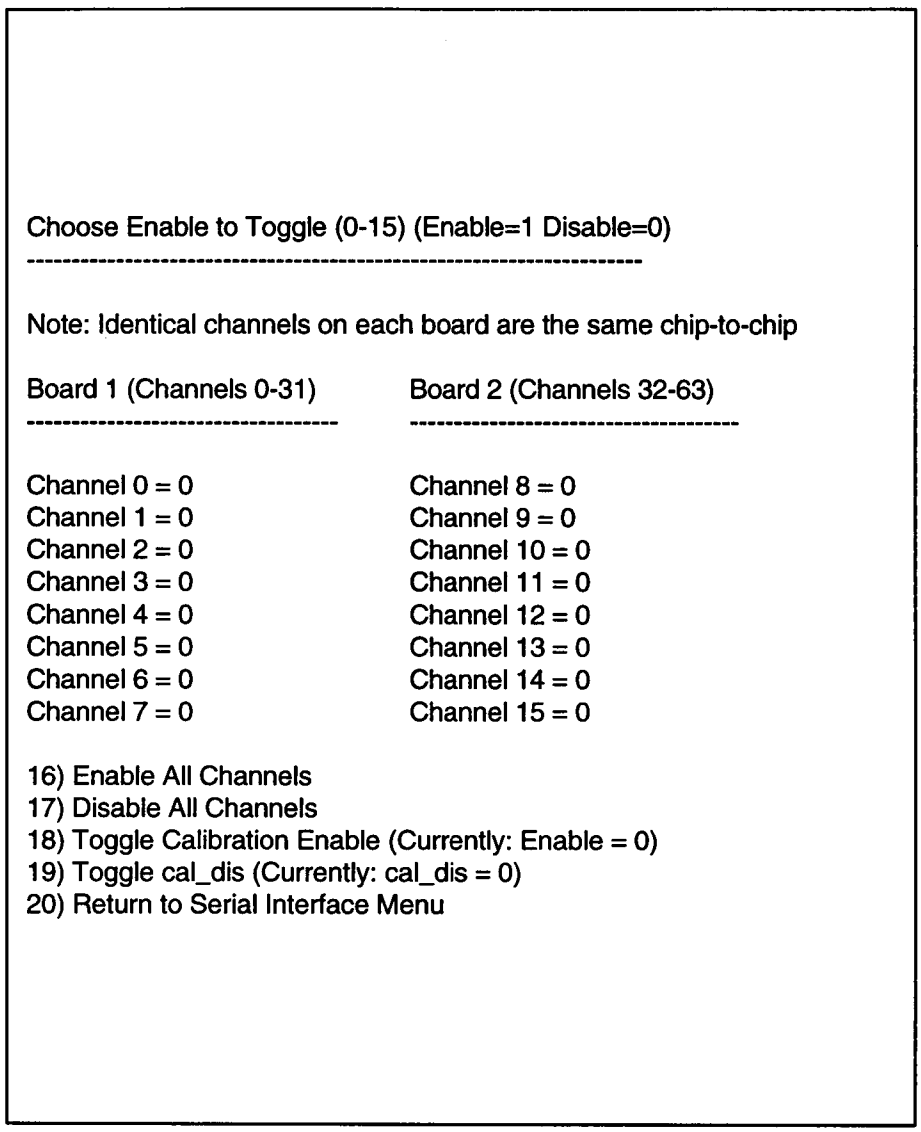


Figure 4.2.9. Option (5) of the Serial Interface Menu

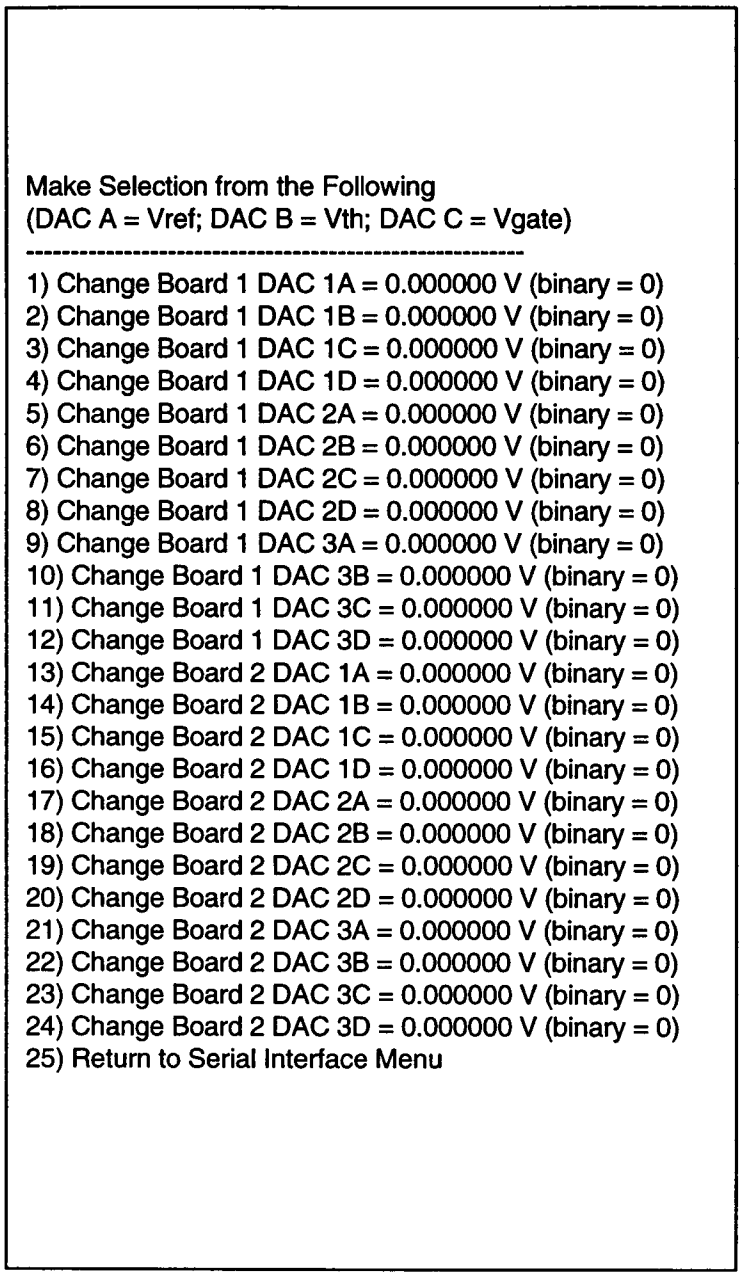


Figure 4.2.10. Option (6) of the Serial Interface Menu

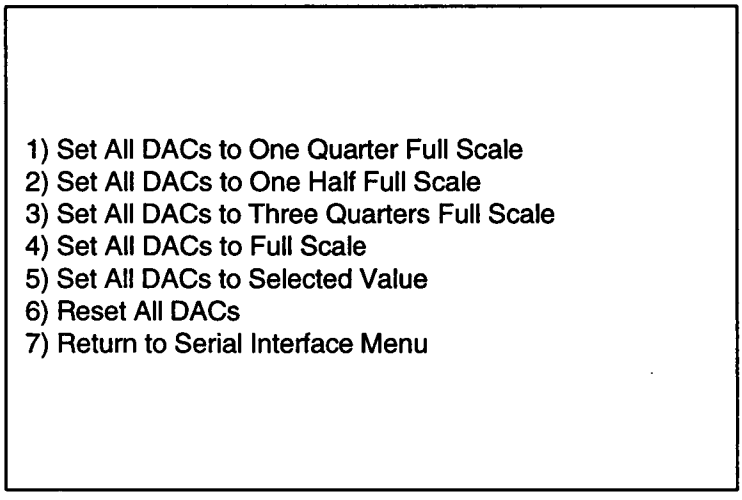


Figure 4.2.11. Option (7) of the Serial Interface Menu

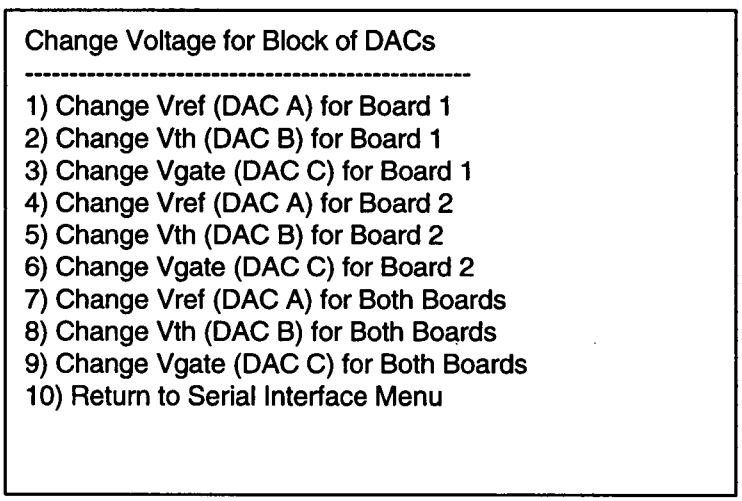


Figure 4.2.12. Option (8) of the Serial Interface Menu

Since the DACs supply the **Vgate** voltages, which sets the gain of the discriminators in the preamps, **Vthreshold**, which determines the discriminator threshold level, and **Vref**, which affects the ramp in the ADCs, the flexibility to set individual DAC voltages makes the system more robust and accounts for possible irregularities which may exist between separate preamps.

Finally, the Xilinx Configuration Interface provides a means of programming the address list manager FPGA and heap manager controller FPGA. As seen in Figure 4.2.13, the Xilinx Configuration Interface contains all functions which directly affect the configuration of the FPGAs. Option (1) programs both parts using ASCII configuration (.rbt) files generated by the Xilinx ProSeries software. To configure the heap manager FPGAs, the program requires that the ALM and heap manager controller files be in the same directory as the program. So the user can monitor the configuration status of the ALM and heap manager controller, option (2) may be selected to reveal the DONE flag and INIT flag for each of the heap manager FPGAs. Moreover, the user may switch between different designs by choosing option (3) to change the ALM configuration file, or option (4) to change the controller configuration file. The default configuration files are named "alm.rbt" and "mvdctrl.rbt" for the ALM and heap manager controller, respectively; and option (5) and option (6) provide an easy means to revert to the default files at any time during the execution of the main routine. Upon completion of any Xilinx reconfiguration sequence, the program reverts to the initialization state, thereby bringing the heap manager up in its dormant, Halt mode. Because it is one of the main reasons behind the selection of Xilinx as the FPGA architecture, the programmability of the heap

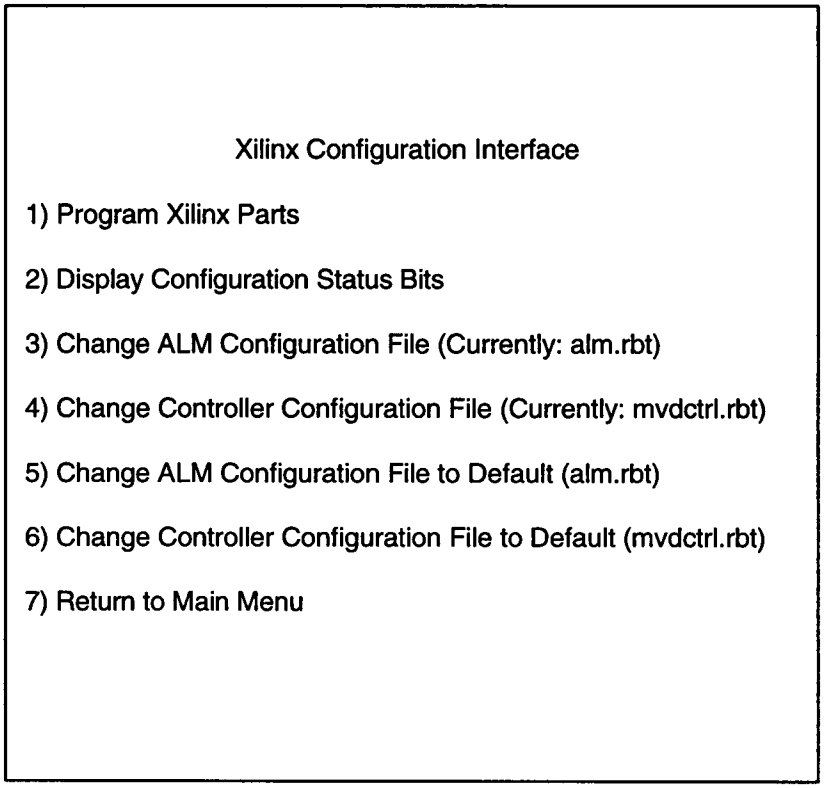


Figure 4.2.13. Xilinx Configuration Interface Menu

manager is exercised fully in the Xilinx Configuration Interface, which provides the user with options that can be utilized to load, rapidly, any number of designs with only a couple of menu choices.

The software for the ORNL development of the MVD beam test unit is written to satisfy the requirements of the beam test and encompass as many permutations of the command codes as possible. Outlets to all heap manager functions affected by the PC interfaces are provided to the user to facilitate testing and system diagnostics, and the subroutines developed for the ORNL test bed can be converted to code for use with the LANL development of software for the CAMAC by simply redirecting the internal variables of the program to coincide with the I/O architecture of the 34-pin and 10-pin ports. From the hierarchical design of the menu structure, the program is developed so that, at most, four consecutive menu choices are needed to access any particular function of the heap manager or PC FIFO.

4.3. Heap Manager Performance

From the beam test, results are studied to measure the robustness and conformity of the heap manager to the pre-determined MVD system specifications. For the heap manager, the points of interest are speed, power, and re-programmability. The speed and power tests devised for the heap manager are based on the assumption of a working FEE; and the re-programmability of the heap manager relies on the synthesis reports from the

Xilinx ProSeries software. Because it is imbedded in a multi-component architecture, the heap manager is isolated from the test fixture as much as possible so that the influence of the FEE and the PC is minimized.

The test setup used in the performance tests on the heap manager involves the use of the continuous readout function of the DCM Interface from the MVD beam test software. A waveform generator is utilized to supply a periodic LVL-1 trigger to the heap manager to mimic a continuous conversion cycle and provides a one-beam-clock-wide pulse at intervals which coincide with the maximum conversion time of the system. To set up the test, a logic analyzer is used to determine the time needed for a conversion to take place and the ADC FIFO address to be re-sorted for a particular **4xclk** frequency. The measurement of the power consumption of the heap manager controller or address list manager is conducted using a digital multimeter, which is placed in series with the connection between the power supply and the corresponding FPGA; and the current is measured and multiplied by the five volts of the supply. Of the problems discovered in the heap manager, the main cause for concern is the inability of the system to operate at the designated **4xclk** rate of 38 MHz.

During testing, the frequency of the square wave produced by the waveform generator is manipulated. As the frequency is increased, the logic analyzer is studied to determine the point of failure for the heap manager. In Table 4.3.1, the results of the heap manager tests for different **4xclk** frequencies are tabulated; and, as can be observed in the tabulated data, the heap manager functions cease to operate properly at 20 MHz, or half of the expected performance criterion. Although the power measurements for the sixty-

Table 4.3.1. Heap Manager Power Requirements

4xclk Frequency (MHz)	ALM Power (mW)	HM Controller Power (mW)	Per-Channel Power (Total Power/64) (mW)	Per-Channel Power (Total Power/256) (mW)
5	108	99	3.23	0.81
10	172	163	5.23	1.31
15	231	224	7.11	1.78
20	287	282	8.89	2.22

four channel system seem to indicate that the per-channel power may exceed the allowed maximum per-channel power (15 mW) specified for the system if the heap manager were able to run at 40 MHz, two FPGAs will service 256 channels in the final system; therefore, the power measurements, translated for the MCM case, are more conservative. Since the beam test version of the MVD system is only applied as a functional prototype, speed issues are not a factor in the execution of the beam test; however, the speed and power requirements must be addressed before the MCM development begins.

Besides the speed and power tests, the robustness of the heap manager is of major import. Since it is intended to be fully re-programmable, the heap manager must have sufficient resources to allow at least minimal changes to be made in the architecture. A study of the resource allocation demonstrates that some flexibility exists in the heap manager; and, From Table 4.3.2, the tally of the I/O and number of configurable logic blocks (CLBs) used by the ALM and heap manager controller indicates that more functionality may be added to the system. With almost ten percent of the CLBs in the heap manager controller FPGA and fifteen percent of the CLBs in the address list manager FPGA unused, the heap manager may be altered to include test circuits or more command routines to enhance the beam test system in route to the realization of the final MVD system.

Through the compilation of the power, speed, and resource allocation measurements for the heap manager, the strengths and weaknesses of the design are exposed. Obviously, the failure of the heap manager to attain the minimum speed strictures placed on the system is disturbing; however, the power measurements and

Table 4.3.2. Heap Manager Resource Allocation

Logic Blocks	ALM Allocation (Used/Total)	HM Controller Allocation (Used/Total)
Bonded I/O Pins	20/61 (32%)	136/160 (85%)
F and G Function Generators	184/392 (46%)	314/800 (39%)
H Function Generators	29/196 (14%)	67/400 (16%)
CLB Flip Flops	113/392 (28%)	276/800 (34%)
Total Occupied CLBs	168/196 (85%)	366/400 (91%)

availability of resources is encouraging. Thus, the main concern for the MVD system is whether the heap manager may be affected enough to gain the speed required by the final system. To reach the goals set forth in the planning phase of the MVD heap manager, the efficacy of the beam test heap manager must be re-examined and a new design methodology, plied with the understanding acquired from the testing data of the beam test prototype, enacted.

Chapter 5

Post-Beam-Test Design Alterations

The MVD beam test, which took place between April 29, 1996 and May 13, 1996, at the Relativistic Heavy Ion Collider (RHIC) at Brookhaven National Laboratory was deemed a success as verbally related by the physicists from Los Alamos National Laboratory. Despite the disappointing lack of speed capabilities exhibited by the heap manager, the system prototype, as a whole, functioned within the parameters specified by the beam test constraints and predicted by the test runs conducted at Oak Ridge National Laboratory; and the system, when hooked to the RHIC detector, provided data which closely approximated expected results. With the first hurdle cleared, the development of the MVD system must focus on enhancing the heap manager to run at speed and within power requirements. Thus, the next step in the MVD system development is to forge ahead with the construction of a sixty-four address heap manager and its sixteen-address sister for fabrication as a multi-chip module (MCM) in the new phase of the Multiplicity Vertex Detector.

5.1. An Improved Address List Manager Design

Obviously, the greatest hurdle to overcome in the redesign of the address list manager is the speed requirement. As noted in the beam test, the ALM, as developed for the beam test, is incapable of running at the beam clock rate of 9.5 MHz. Thus, a different approach is needed to ensure the future of the digital portion of the MVD project.

The direction taken in the post-beam test design of the address list manager is to overhaul completely the beam test heap manager. The fruit of the redesign process is an address list manager which utilizes six-bit AMU addressing at **4xclk** speeds above 60 MHz, well over the required **4xclk** of 38 MHz. The revitalized design of the ALM is achieved by employing a base-one approach much like the methodology used to create the beam test address list manager. In other words, for the sixty-four address ALM, the building elements are taken from the beam test address list manager and re-applied in a fresh architecture that takes advantage of new insights on the design.

The sixty-four-address ALM utilizes the same multiplexers and decision logic as the four-bit address list manager, with the exception that the decision logic for re-sorting is placed on the output side of the Available Address FIFO and is relieved of the necessity of performing writes on two consecutive **4xclk** cycles; and the LFSR FIFOs are modified for better performance. The block diagram for the six-bit ALM is shown in Figure 5.1.1; and, as can be seen from the architecture, the decision logic operates on the Available Address FIFO output. The advantage to regrouping the decision logic is that the AMU write address can be supplied by the LVL1 FIFO; and the re-sort is coupled with the

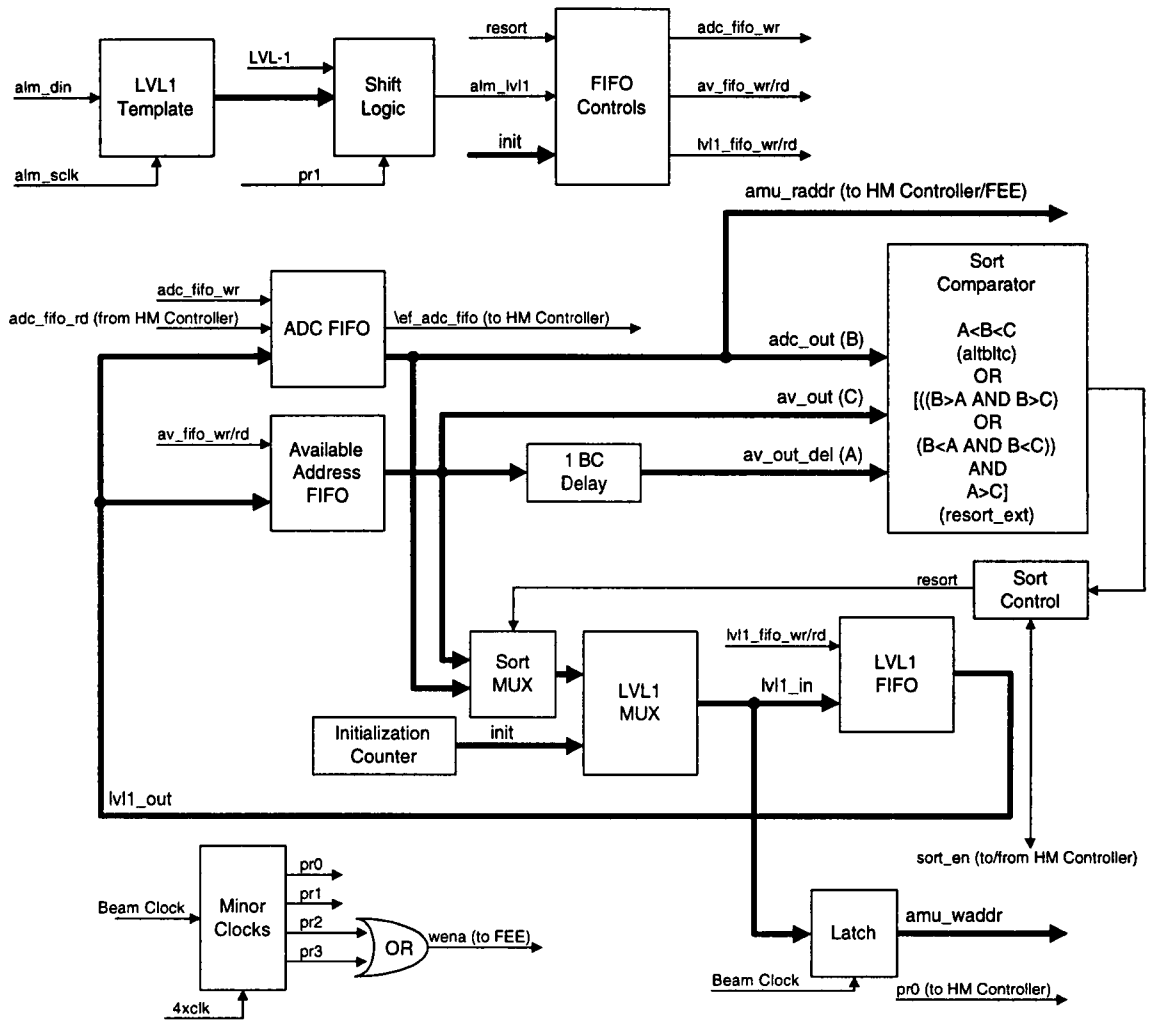


Figure 5.1.1. Block Diagram of the Six-Bit Address List Manager

initialization logic so that all multiplexers are in the input path of the LVL1 FIFO. The modification yields timing which does not require two writes to a single FIFO during any given beam clock cycle, thus relaxing the speed requirements of the FIFOs. Essentially, all writes and all reads can be made during one of the fast clock cycles (**pr0**, **pr1**, **pr2**, or **pr3**); and the data setup time for all of the FIFOs, including the inputs to the Available Address FIFO, can be lengthened to half of a beam clock, thus allowing each FIFO enough time to write data before a read command is initiated. Furthermore, the ADC FIFO is moved from the heap manager controller FPGA and placed in the ALM FPGA to take advantage of the new architecture and complete the data path from the Available Address FIFO to the re-sorting logic, thus allowing more strict control of critical paths and minimizing the delays between the ADC FIFO and the re-sort function.

A fundamental change in the architecture of the address list manager is that the FIFOs are reconstructed so that a read and a write can utilize a whole **4xclk** cycle. In the four-bit ALM design, the FIFOs conduct a write on the first half of the clock and a read on the last half of the **4xclk** cycle. The FIFOs for the six-bit address list manager, however, provide control of the internal write and read signals of the SRAM to the ALM logic. The functional blocks of the six-bit FIFO are outlined in figure 5.1.2, in which the interdependence of the control logic and the SRAM and **push/pop** inputs is mapped. As seen in Figure 5.1.2, the LFSR blocks and flag logic are left intact; however, the control of the static RAM is left to the external control signals which can make the read and write period as long as needed. Thus, the FIFO retains the speed inherent in the use of the linear feedback shift registers; but the added flexibility of the control and SRAM accessing logic

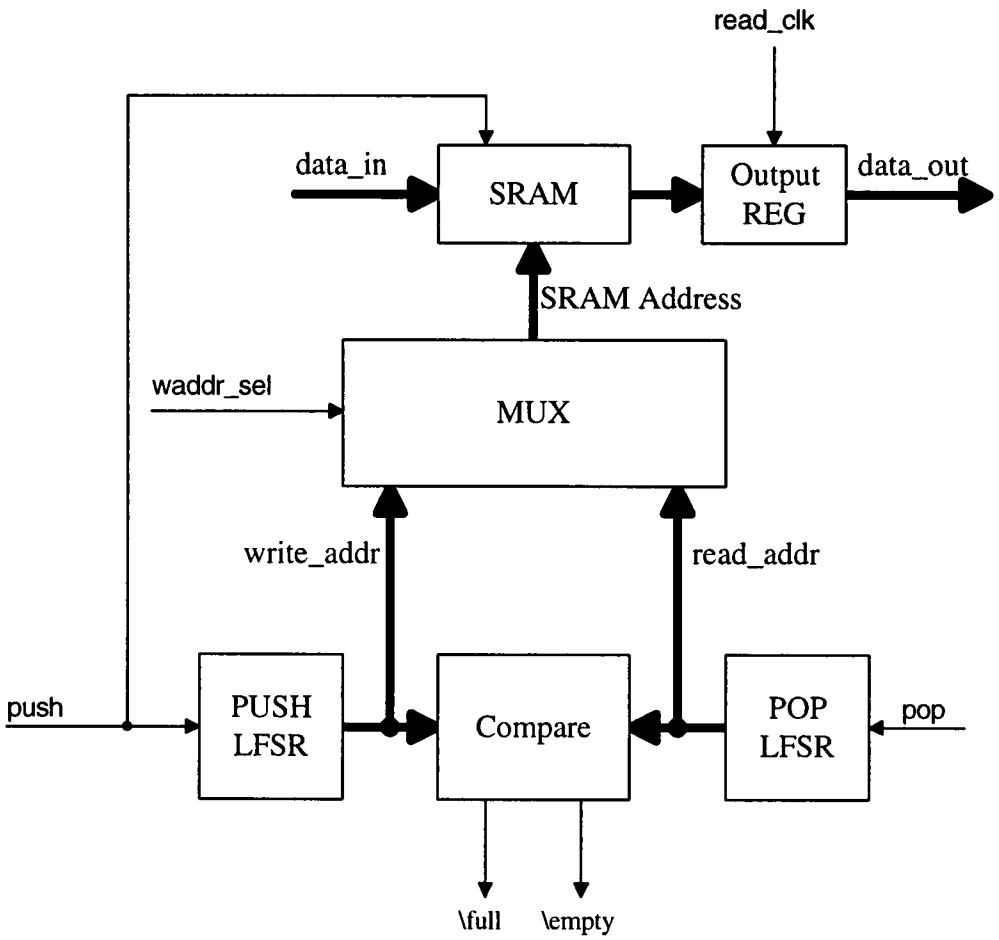


Figure 5.1.2. Updated FIFO Architecture for the Six-Bit Address List Manager

in the updated FIFO makes the address list manager less susceptible to timing delays in the multiplexer paths of the re-sort logic. For the new address list manager design, the improved FIFO architecture gives the address list manager an extra half **4xclk** cycle with which it may read or write to the SRAM; and, coupled with the ability of the sorting multiplexer to defer one of the two writes needed by the beam test ALM, the revitalized FIFO extends the delay constraints and increases the speed of the address list manager.

On the heels of the improvements made on the beam test address list manager, an improvement is made on the LVL-1 handling capability of the address list manager. Besides the restructuring of the re-sort logic and FIFOs, the six-bit ALM contains a new piece of logic, the LVL1 template. Conceived as a shift register which allows the controlled spacing of eight individual **alm_lvl1** triggers for each LVL-1 generated, the LVL1 template is a serial interface in which data is shifted via the **alm_din** and **alm_sclk** signals. Similar to the **trig_delay** command in the beam test heap manager, the LVL1 template data bits specify the LVL-1 trigger status for eight consecutive beam clocks after an initial **alm_lvl1** is received. The timing sequence for the LVL1 template serial string is portrayed in Figure 5.1.3. Each **alm_din** bit that is shifted into the template represents a single LVL-1 trigger; and, when an **alm_lvl1** is received, the template begins presenting LVL-1 triggers, based on the bits contained in the template, on consecutive beam clocks, for each of the bits represented in the shift register.

Simulations conducted for the six-bit address list manager are based on the simulations made for the beam test address list manager for comparison. The addressing for the six-bit address list manager includes an initialization sequence comparable to the

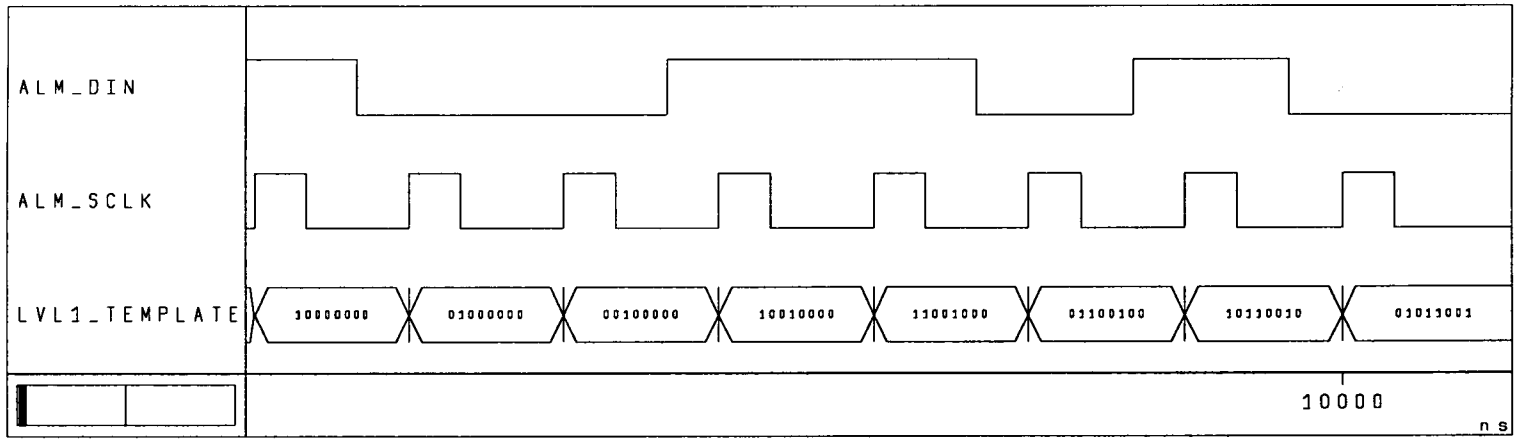


Figure 5.1.3. LVL1 Template Timing for the Six-Bit Address List Manager

one used for the four-bit ALM. As suggested in Figure 5.1.4, the LVL1 FIFO is filled before the first read (**lvl1_fifo_rd**) is made; therefore, a delay of thirty-one addresses exists between the output of the LVL1 FIFO (**lvl1_out**) and the Available Address FIFO (**av_out**). Thus, between the time when an analog value is stored in the AMU, via **amu_waddr** and **wena**, the system has thirty-one beam clocks in which to decide that a valid event has occurred and to issue a LVL-1 trigger (**alm_lvl1**). As is the case for the beam test address list manager, for the six-bit ALM, all reads occur with the **pr0** clock and all writes are synchronized with **pr2**, allowing a two-beam-clock delay between same-FIFO operations. To make the most effective use of the read and write cycles of its architecture and the natural setup times created by the separation of the read and write operations, the address list manager synchronizes the LVL-1 trigger with the **pr0** minor clock cycle so that the first bit of the LVL-1 template can mask the immediate LVL1 FIFO output.

In contrast to the beam test address list manager, the LVL-1 handling capability of the ALM is contingent on the contents of the LVL1 template, which is generated inside the FPGA of the ALM, thus by-passing the delays accrued in the routing network of the heap manager controller. An occurrence of a system LVL-1 (**alm_lvl1**) is injected into the simulation of Figure 5.1.5. For eight beam clocks after an event is tagged, the LVL1 template (**lvl1_template**) is shifted one bit at a time; and the LVL-1 status for a given beam clock cycle becomes the LSB of the template. In the case given in Figure 5.1.5, four separate LVL-1s are generated; and the corresponding **lvl1_out** addresses (60, 63, 00, and 02) are removed from the addressing loop. To replace the addresses, the address list

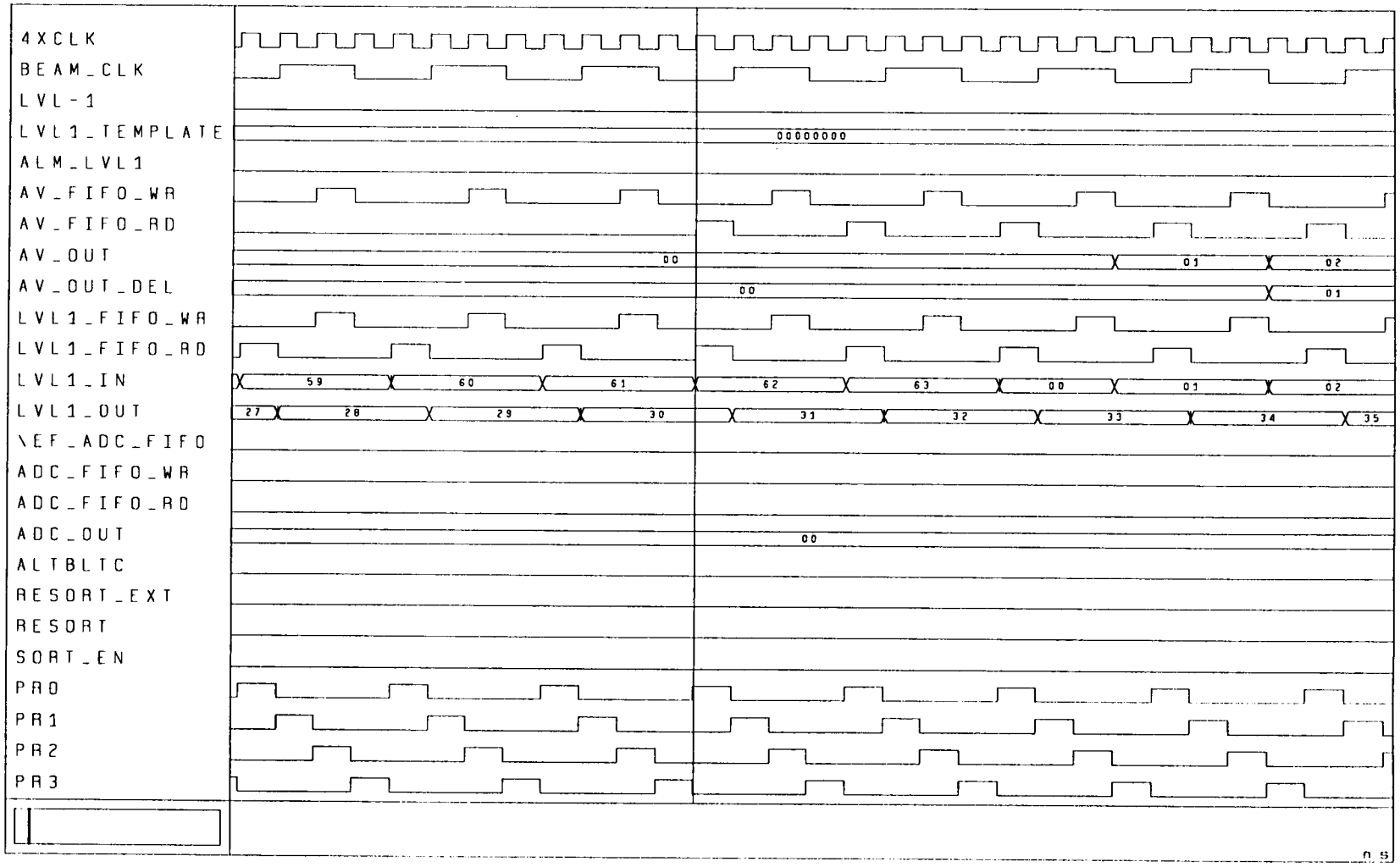


Figure 5.1.4. Initialization Sequence for the Six-Bit Address List Manager

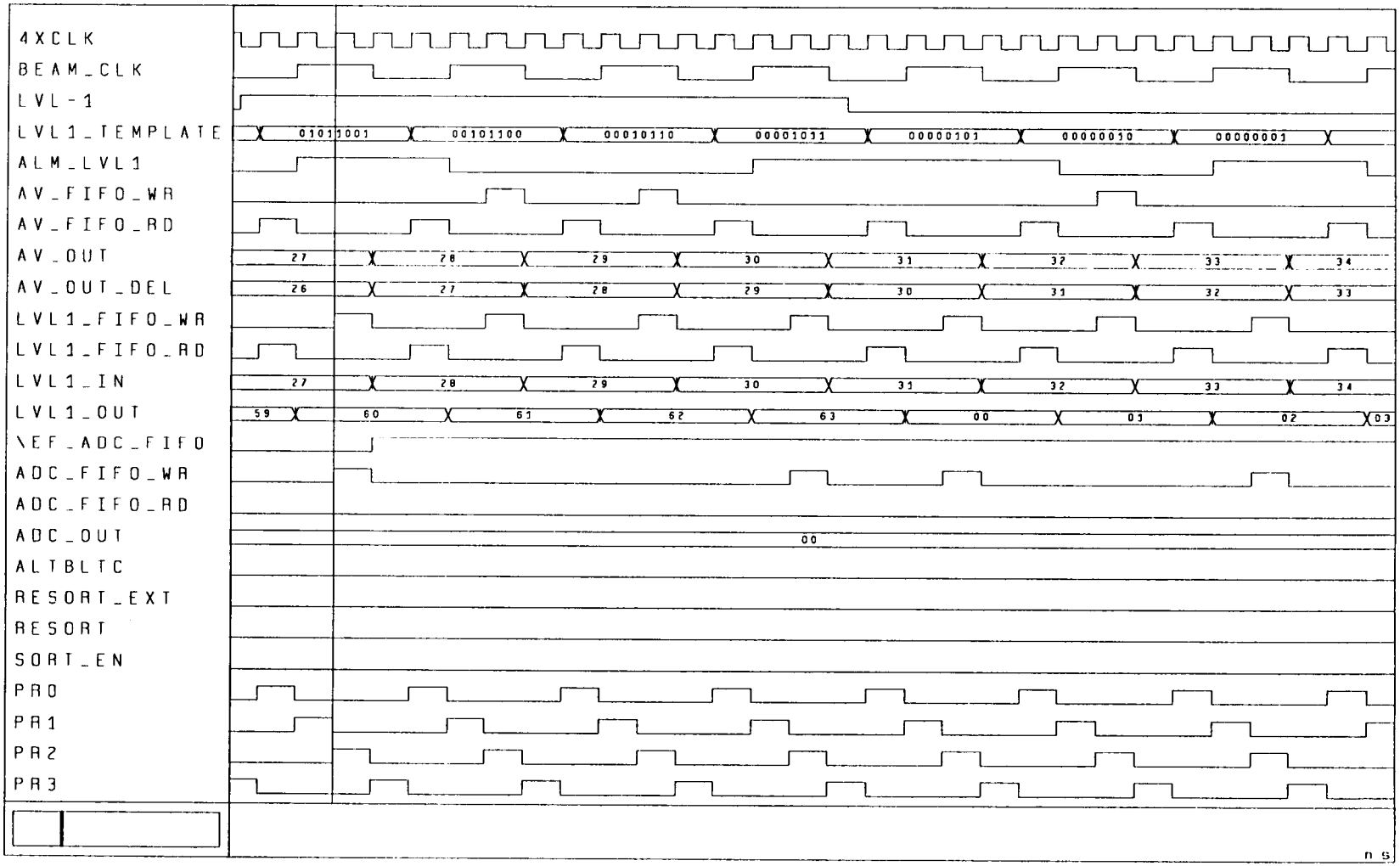


Figure 5.1.5. LVL-1 Handling in the Six-Bit Address List Manager

manager, as is the case in the beam test ALM, relies on the heap manager controller to monitor the ADC FIFO empty flag, read the ADC FIFO, make a conversion, and activate the sort enable.

With the receipt of a sort enable, the re-sorting logic for the six-bit ALM takes advantage of the one-write-per-beam-clock constraint. Because the same re-sort conditions apply to the six-bit address list manager that apply to the beam test ALM, the VHDL-generated sort multiplexers are left unchanged; however, the actions taken by the revitalized address list manager upon a valid re-sort condition deviate from the functions used in the beam test design by the four-bit ALM. For instance, the A<B<C (**altbltc**) re-sort condition, as presented in Figure 5.1.6, causes the read to the Available Address FIFO (**av_fifo_rd**) to be suppressed for one beam clock; and, instead of writing the output of the Available Address FIFO (**av_out**) into the LVL1 FIFO, the ALM writes the output of the ADC FIFO (**adc_out**), or address 60, to the LVL1 FIFO. Similarly, for a border re-sort, as shown in Figure 5.1.7, address 63 is written into the LVL1 FIFO in place of the **av_out** address, 01. Thus, both the border and A<B<C re-sort conditions are satisfied.

Whereas the beam test ALM, upon re-sorting an address, immediately performs a second FIFO write, the six-bit address list manager, which can make only one write per beam clock cycle per FIFO, cannot push the **av_out** address into the LVL1 FIFO during the same beam clock cycle as the re-sort and must, therefore, prolong the time during which the **av_out** address is presented to the sorting mechanism. Consequently, in the beam clock following a re-sort operation, the **av_fifo_rd** is masked; and the **av_out** address that was avoided during the re-sort condition is presented a second time to the re-

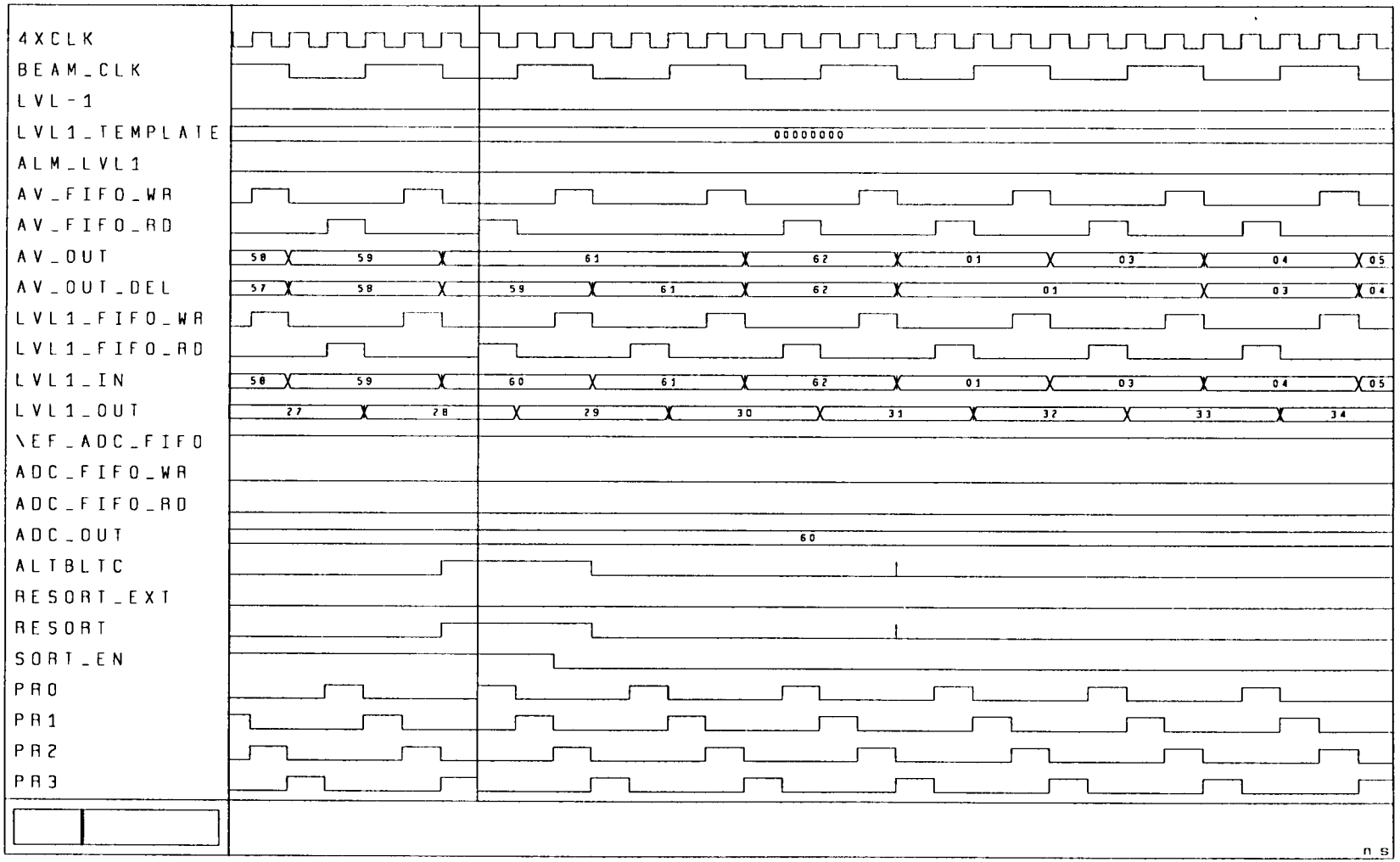


Figure 5.1.6. A<B<C Re-Sort Timing for the Six-Bit Address List Manager

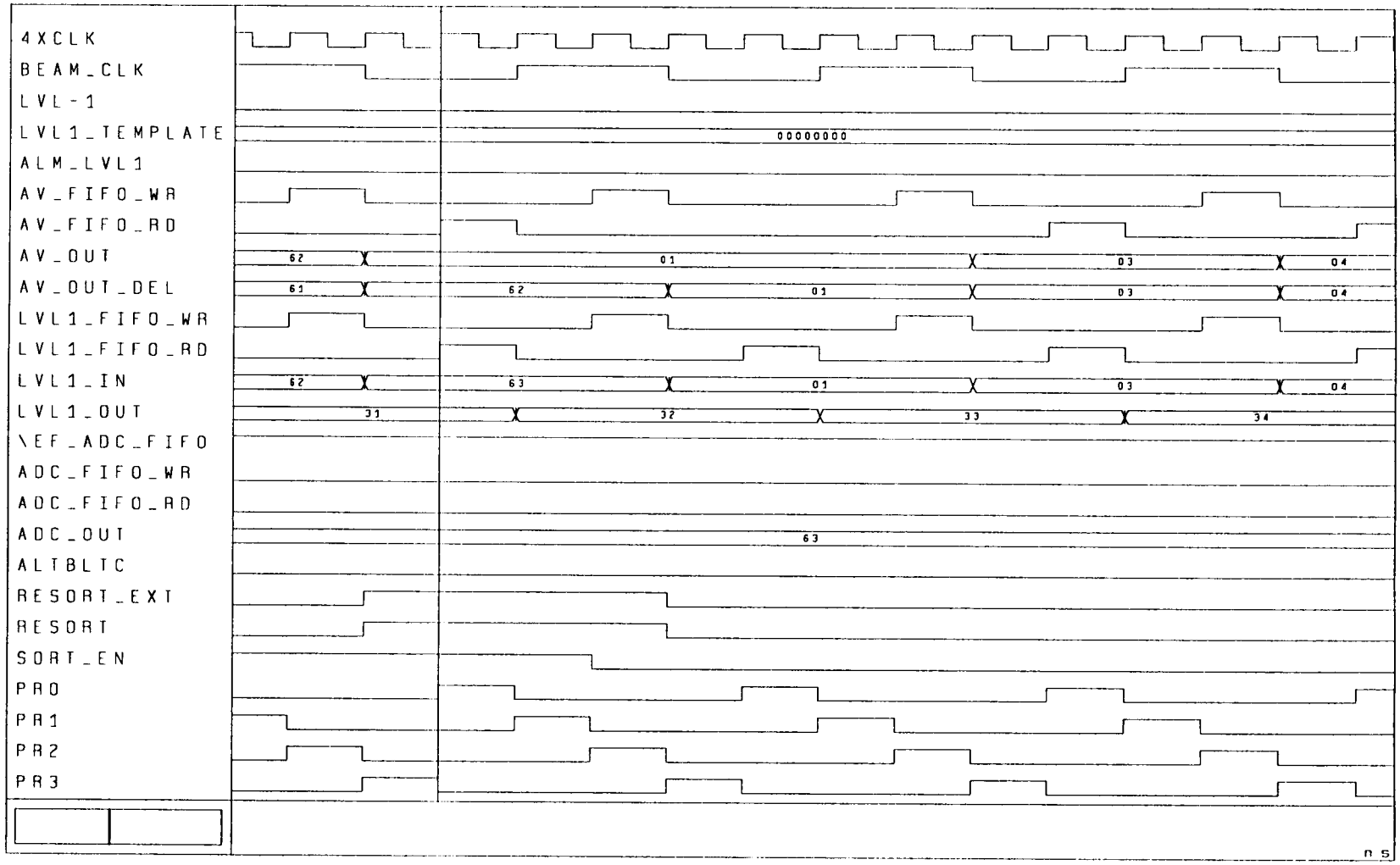


Figure 5.1.7. Border Re-Sort Timing for the Six-Bit Address List Manager

sort multiplexers, thus preventing the by-passed **av_out** address from being missed and summarily taken out of the loop. Because the re-sort condition occurs only when an address has been removed from the Available Address FIFO, suppression of the **av_fifo_rd** does not affect the operation of the circuit and never results in the overflow of the Available Address FIFO. Furthermore, because it is derived from a delayed version of the **lvl1_in** address, the AMU write address (**amu_waddr**) can reflect the status of the re-sorted address on the same beam clock in which **adc_out** is written to the LVL1 FIFO, thus maximizing the efficiency with which AMU resources are allocated. The ability of the six-bit address list manager to avoid disturbance of the timing for the AMU write function while decreasing the overhead associated with the timing of the re-sort, as compared to the functions of the beam test ALM, proves that improvements can be made to the address list manager to increase speed under the constraints of the Xilinx architecture.

To confirm that the improvements made in the design of the address list manager are genuine, the six-bit ALM is tested on the MVD heap manager board in conjunction with a test circuit that is constructed for use in the FPGA normally occupied by the heap manager controller. As measured in the laboratory, the six-bit address list manager, in its new form, is capable of functioning at speeds in excess of 60 MHz. Obviously, the efforts used in the restructuring of the LFSR FIFOs and the re-ordering of the re-sort logic are fruitful. Although more resources are needed due to the enlargement of the address bus and the inclusion of the LVL1 template and ADC FIFO in the ALM FPGA, the six-bit address list manager enjoys an increase in speed of 300 percent over the performance of the four-bit ALM and meets the 38 MHz requirement under the MVD specifications.

5.2. Multi-Chip Module (MCM) Development

Following the success of the beam test unit and the improved six-bit ALM design, the first step toward the final MVD system is the meshing of the FEE and the heap manager into a single multi-chip module which is the precursor to the 256-channel MCM of the final MVD subsystem. Consequently, the MVD beam test system is used as a model for the construction of a thirty-two channel, sixteen-cell prototype of the final MCM, whose tentative block diagram is presented in Figure 5.2.1. The first cut of the multi-chip module consists of the heap manager and the contents of one FEE board and one preamp baby board. Due to space considerations, only the bare essentials are included on the substrate blueprint; therefore, none of the DACs or SPY multiplexers are included with the new design. Although a serial fiber optics connection between the MCM and DCM Interface Card are conceived to carry all data from the multi-chip module to the external computer, the details of the 32-channel prototype are not finalized; however since all ASICs and the FPGAs are embedded, in both the prototype and 256-channel MCM, as bare die, the power requirement of the heap manager must be met before construction of the 32-channel model is complete to prevent overheating of the substrate. Although still early in its development, the first attempt at the MCM design requires that a new, four-bit address list manager be constructed which can meet the power and speed constraints of the final MVD system.

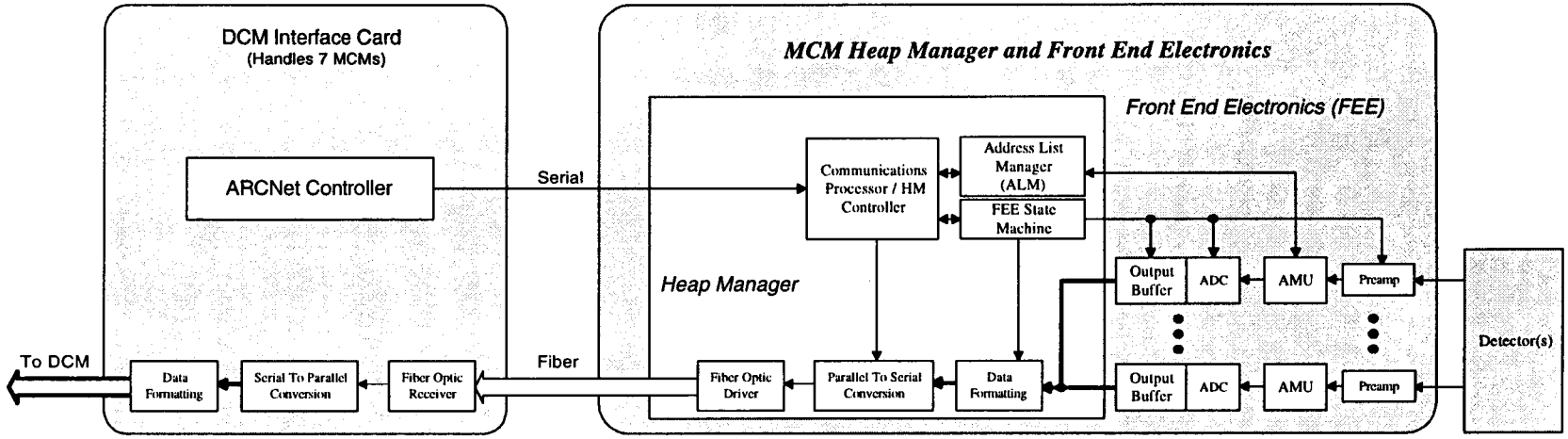


Figure 5.2.1. Block Diagram of the Multi-Chip Module (MCM) for MVD

5.3. Four-Bit Heap Manager Revision

To meet to the requirements demanded by the first cut of the multi-chip module, the six-bit address list manager is revised into a four-bit ALM that conforms to the functionality of the beam test system and is paired with a compatible heap manager controller. Since the architecture of the six-bit ALM is in place, the four-bit address list manager is derived directly from the six-bit version; and the functionality, speed, and power characteristics of the MCM heap manager are tested by mating the ALM with an updated version of the heap manager controller, which is re-sized to take advantage of the new architecture. With the advancement of the ALM and heap manager controller, a heap manager architecture is developed which parallels the design of the MCM and takes advantage of the lessons learned from the beam test prototype.

To obtain the system functionality realized in the beam test, the sixteen-address heap manager is structured to take advantage of the beam test prototype board and the existing PC code. As evident in Figure 5.3.1, the inclusion of the LVL1 template from the six-bit ALM is forgone in the four-bit ALM; and the heap manager controller is left the task of formatting incoming LVL-1 triggers for both components of the heap manager; however, due to the removal of the ADC FIFO from the heap manager controller, the empty flag (**\ef_adc_fifo**) must be shared between the ALM and the readout controller unit. Although they do not affect the overall functionality of the heap manager, the changes made to accommodate the new layout are instrumental in the attempt to increase the speed performance of the heap manager.

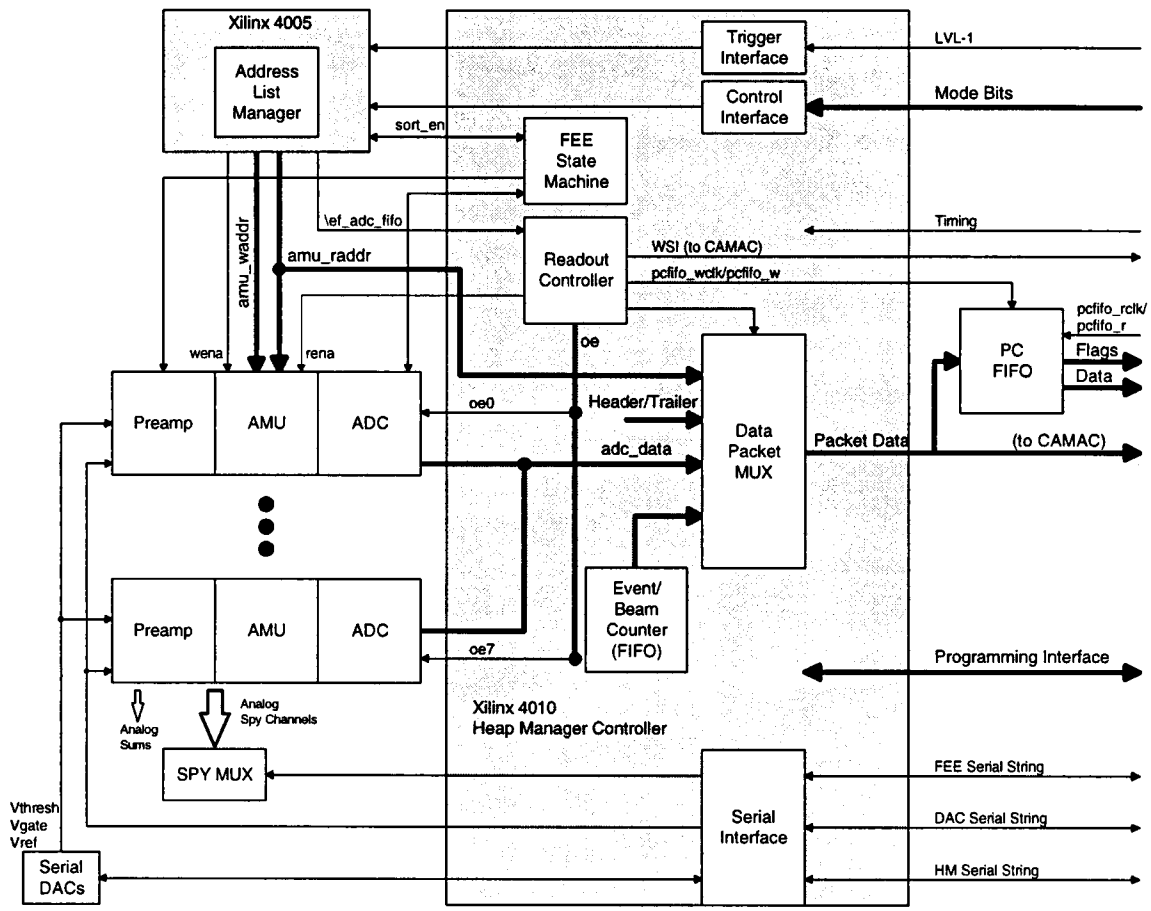


Figure 5.3.1. Block Diagram of the Revised Heap Manager Architecture

The integration of the revised four-bit ALM and the heap manager controller means that both components of the heap manager must be re-synthesized for use in the Xilinx FPGAs. As can be observed in Table 5.3.1, the logic requirements of the address list manager and heap manager controller benefit from the changes made in the post-beam-test development. A drop in allocation of configurable logic blocks (CLBs) is witnessed for both the ALM and heap manager controller; and the I/O strictures placed on the beam test heap manager controller by the inclusion of the ADC FIFO in the XC4010 part are removed. Consequently, the new design yields an opportunity for more flexibility in the MCM system and allows room for more functionality to be added to the heap manager, if needed.

The renovated heap manager is tested with the beam test system to ensure the functionality and enable comparisons to be made between the beam test heap manager and the post-beam-test heap manager. As its performance is deemed to be identical to the functional capabilities of the beam test prototype, the revised heap manager is measured for speed and power; and the results are presented in Table 5.3.2. As should be noted from the power and speed measurements, the top speed of the revised system is 59 MHz; and, without the FEE, the power, as measured in relation to a 64-channel or 256-channel system, consumed by the heap manger is below the 15 mW power requirement at 40 MHz. Considering that the routing of the MCM is on a substrate and the components of the heap manager and FEE are close together, the absolute speed of the heap manager should be sufficient to carry the design through the final phases of the Multiplicity Vertex Detector development.

Table 5.3.1. Revitalized Sixteen-Address Heap Manager Resource Allocation

Logic Blocks	ALM Allocation (Used/Total)	HM Controller Allocation (Used/Total)
Bonded I/O Pins	19/61 (31%)	126/160 (73%)
F and G Function Generators	162/392 (41%)	280/800 (35%)
H Function Generators	51/196 (26%)	61/400 (15%)
CLB Flip Flops	127/392 (32%)	237/800 (29%)
Total Occupied CLBs	166/196 (84%)	295/400 (73%)

Table 5.3.2. Revitalized Sixteen-Address Heap Manager Power Requirements

4xclk Frequency (MHz)	ALM Power (mW)	HM Controller Power (mW)	Per-Channel Power (Total Power/64) (mW)	Per-Channel Power (Total Power/256) (mW)
5	78	63	2.20	0.55
10	127	106	3.64	0.91
15	174	149	5.05	1.26
20	214	190	6.31	1.58
25	257	229	7.59	1.90
30	295	268	8.80	2.20
35	330	304	9.91	2.48
40	370	341	11.11	2.78
45	400	376	12.13	3.03
50	435	411	13.22	3.30
59	493	472	15.08	3.77

Chapter 6

Conclusions

By choosing a carefully constructed design methodology and adhering to the requirements laid out in the specifications for the MVD beam test system, development of a fully-functional 64-channel prototype of the Multiplicity Vertex Detector is achieved. From the structuring of the front-end module to the definition of the front-end electronics and CPU interfaces, the heap manager is founded on the prospect of the creation of a multi-dimensional control unit that is capable of managing analog components toward the production of digital data; and, from the specifications of MVD, the requirements on the heap manager include a per-channel power ceiling of 15 mW-per-channel and a minimum speed goal for operation with a 38 MHz **4xclk**, or 9.5 MHz beam clock. Consideration of the functional demands placed on the heap manager yields the realization that, for the AMU to be able to store data each beam clock, the address list manager must provide a continuous address loop, in which any address may be removed or re-sorted when tagged by a LVL-1 trigger; and, due to the large number of support functions and I/O needed to translate the CPU commands, a heap manager controller, partitioned in its own FPGA, is desirable. Integration of the heap manager and the FEE, and the subsequent coding for a

PC interface, is a task which embodies the definition of a data acquisition system as a group of mixed-signal components that can successfully retrieve and store information in a cogent fashion.

With the overhead created by the stringent I/O and speed requirements, the separation of the heap manager into an address list manager and heap manager controller is conceived as a means of accomplishing both strictures placed on the digital control portion of MVD. Since the system is reliant on the idea of flexibility and re-programmability, the choice of Xilinx as a platform yields the CLBs and RAM cells necessitated by a FIFO-based architecture. Division of responsibilities between the heap manager controller and address list manager and the unification of the heap manager into a cooperative platform provides the impetus for reaching the speed requirements of the ALM while facilitating the development of the interfaces between the controller and the PC and CAMAC crate. The combination of re-usable resources found in the Xilinx family and the subdivision of the heap manager results in a system which performs under the specifications of MVD.

From the beam test prototype, the ability to create a data acquisition system utilizing FPGA technology and custom ASIC designs is demonstrated; and, by taking advantage of the re-programmable functionality of the Xilinx FPGAs, changes in the original floorplans of the controller and address list manager are effected without having to re-fabricate the architectures in a silicon process, which is both time consuming and expensive. Since the goal of PHENIX and MVD is to measure physics events which originate from a collision of particle beams, the use of a custom chip set allows the design

to be tailored to the objectives of the experiment while precluding the ill effects of power overhead from off-the-shelf components. Utilization of the FPGAs in route to a working heap manager and the successful implementation of the heap manager in the beam test prototype favors the ultimate goal of an optimized electronics unit for the PHENIX MVD.

References

References

[1] Krane, Kenneth. Modern Physics. New York: John Wiley and Sons, 1983.

[2] Gettys, Edward, Frederick Keller, and Malcolm Skove. Classical and Modern Physics. New York: McGraw-Hill, 1989.

[3] Kehoe, W. L., et al. PHENIX Conceptual Design Report. New York: Brookhaven National Laboratory, 1993.

[4] Maguire, Charlie. "Goals of the PHENIX Experiment." [http://phnxmu.lanl.gov/nodes.html]. May 1996.

[5] Britton, C. L., L. C. Clonts, A. L. Wintenberg, K. F. Read, E. J. Kennedy, R. S. Smith, B. K. Swann, and J. A. Musser. "An Analog Random-Access Memory In the AVLSI-RA Process for an Interpolating Pad Chamber." *IEEE Transactions on Nuclear Science*. Vol. 42, No. 6, 1995.

[6] Milgrome, O. B., S. A. Kleinfelder, and M. E. Levi. "A 12-Bit Analog to Digital Converter for VLSI Applications in Nuclear Science." *IEEE Transactions on Nuclear Science*. Vol. 39, No. 4, 1992.

[7] "CMOS SyncFIFO." Integrated Device Technology: August, 1993.

[8] Wintenberg, A. L., et al. "Monolithic Circuits for the WA98 Lead Glass Calorimeter." *Conference Record of the 1994 Nuclear Science Symposium and Medical Imaging Conference*. November, 1994.

[9] Anghinolfi, F., et al. "Characteristics of a 'Harp' Signal Processor with Analog Memory Operated with Segmented Silicon Detectors." *IEEE Transactions on Nuclear Science*. Vol. 41, No. 4. August 1994. pp. 1130-1134.

[10] Ericson, M. N., M. S. Musrock, C. L. Britton, J. W. Walker, A. L. Wintenberg, G. R. Young, and M. D. Allen. "A Flexible Analog Memory Address List Manager for PHENIX." *IEEE Transactions on Nuclear Science*. Vol. 43, No. 3. June 1996. pp. 1629-1633.

[11] "High-Performance RAM-Based FIFO." Xilinx Application Note XAPP 044.000. The Programmable Logic Data Book. 1994.

Appendices

Appendix A

System Code

```
/*
Program: beam_tst.cpp
MVD Beam Test Control Program V2

By: Mark Douglas Allen
Oak Ridge National Laboratory
PO Box 2008, MS 6006
Oak Ridge, TN 37831
Phone: 423-574-6330
e-mail: allenmd@custom.ic.ornl.gov

This program controls the MVD Beam Test including
all serial operations and data packet readout. Two
Digital I/O serial "cheesy" cards are used to control
I/O.

Card #1 is used for serial operations and mode control
Card #2 is used for data interface (DCM interface)

ADDRESS MAP TABLE - CARD #1

Addr      Write        Read
-----     -----        ----
2A0       DO[0-15]     DI[0-15]
2A2       DO[16-31]    DI[16-31]

ADDRESS MAP TABLE - CARD #2

Addr      Write        Read
-----     -----        ----
3E8       DO[0-15]     DI[0-15]
3EA       DO[16-31]    DI[16-31]
=========================================================================

CARD #1 Pin/signal Designations (Serial/Mode Control):


HM PCB J10 (from comp to board) - Assigned to DO[0-15] (board1_port1)
------------------------
Pin#            Signal
----            ----------
1       DO0     HM_DIN
2       DO1     HM_SCLK
3       DO2     HM_SLOADB
4       DO3     XIN1
5       DO4     PC_DIN
6       DO5     PC_CCLK
7       DO6     PC_PROG
8       DO7     CPU_RST
9       DO8     SR1_DIN
10      DO9     SR1_SCLK
11      D10     SR1_RST
12      D11     XIN2
13      DO12    SR2_DIN
14      DO13    SR2_SCLK
15      DO14    SR2_RST
16      DO15    XIN3

HM PCB J12 (from board to comp) - Assigned to DI[16-31] (board1_port2)
------------------------
Pin#            Signal
----            ----------
1       DI0     DAC1_DOUT
```

```
2       DI1     DAC2_DOUT
3       DI2     HM_DOUT
4       DI3     XOUT1
5       DI4     XOUT2
6       DI5     XOUT3
7       DI6     XOUT4
8       DI7     XOUT5
9       DI8     PC_INIT
10      DI9     PC_INIT2
11      DI10    DONE
12      DI11    DONE2

HM PCB J11 (from comp to board) - Assigned to DO[16-31] (board1_port2)
------------------------
Pin#            Signal
----            ----------
1       DO16    DAC1_LDACB
2       DO17    DAC1_DIN
3       DO18    DAC1_SCLK
4       DO19    DAC1_RSTB
5       DO20    DAC2_LDACB
6       DO21    DAC2_DIN
7       DO22    DAC2_SCLK
8       DO23    DAC2_RSTB
9       DO24    MB0
10      DO25    MB1
11      DO26    MB2
12      DO27    MB3             (CPU system reset)
13      DO28    MB4
14      DO29    XO4
15      DO30    XO5
16      DO31    XO6
================================================================

CARD #2 Pin/Signal Assignments (Data/DCM Interface):


HM PCB J5 (from board to comp) - Assigned to DI[0-15] (board2_port1)
------------------------
Pin#              Signal
----            ----------
1-16  DI[0-15]  PACKET0-PACKET15

HM PCB J14 (from comp to board) - Assigned to DO[0-15] (board2_port1)
------------------------
Pin#            Signal
----            ----------
1       DO0     PCFIFO_R
2       DO1     PCFIFO_RCLK
3       DO2     PCFIFO_RS
4       DO3     PCFIFO_LOAD
5       DO4     DAC1_CSB
6       DO5     DAC2_CSB
7       DO6     PC_DIN2
8       DO7     PC_CCLK2

HM PCB J6 (from board to comp) - Assigned to DI[16-31] (board2_port2)
------------------------
Pin#            Signal
----            ----------
1       DI16    PCFIFO_FF
2       DI17    PCFIFO_PAF
3       DI18    PCFIFO_HF
```

```
4       DI19    PCFIFO_PAE
5       DI20    PCFIFO_EF

*/

#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "bcd.h"

int i=0;                        /* counting variable */
long int j=0;                   /* counting variable */

/* cheesy card setup */

int board1_port1=0x2a0;         /* address of board 1 DX[0-15]   */
int board1_port2=0x2a2;         /* address of board 1 DX[16-31] */
int board2_port1=0x3e8;         /* address of board 2 DX[0-15]   */
int board2_port2=0x3ea;         /* address of board 2 DX[16-31] */

int control11=0;                /* control for board 1 port 1 */
int control12=0;                /* control for board 1 port 2 */
int control2=0;                 /* control for board 2 */

/* DCM Interface */
FILE *fpd;                      /* file pointer for packet data file */
char fdat[15]="mvd.dat";        /* packet data file */
char fdat_def[15]="mvd.dat";    /* default packet data file */

/* PC FIFO flag variables */
int ff=0;                       /* pc fifo full flag */
int paf=0;                      /* pc fifo almost full flag */
int hf=0;                       /* pc fifo half full flag */
int pae=0;                      /* pc fifo almost empty flag */
int ef=0;                       /* pc fifo empty flag */
unsigned int packet=0;          /* data packet */
unsigned int pack_dat[128];     /* packet data */

/* masks for PC FIFO flags */
int ff_mask=0x1;                /* mask for pc fifo full flag */
int paf_mask=0x2;               /* mask for pc fifo almost full flag */
int hf_mask=0x4;                /* mask for pc fifo half full flag */
int pae_mask=0x8;               /* mask for pc fifo almost empty flag */
int ef_mask=0x10;               /* mask for pc fifo empty flag */
unsigned int packet_mask=0xFFF; /* mask for data packet */

/* PC FIFO offsets */
int pcfifo_r=0x1;               /* pc fifo read */
int pcfifo_rclk=0x2;            /* pc fifo clock */
int pcfifo_rs=0x4;              /* pc fifo reset */
int pcfifo_load=0x8;            /* pc fifo load */

/* end DCM Interface */

/* Xilinx Configuration Interface */
FILE *fpctrl;                   /* pointer for ASCII file for controller */
FILE *fpalm;                    /* pointer for ASCII file for alm */
char fctrl[15]="mvdctrl.rbt";   /* ASCII configuration file for controller */
char fctrl_def[15]="mvdctrl.rbt"; /* default file for controller */
char falm[15]="alm.rbt";        /* ASCII configuration file for ALM */
char falm_def[15]="alm.rbt";    /* default file for ALM */
int file_error=0;               /* flag file error */
int line_count=0;               /* line counting variable for files */
```

```
int rbt=0;                          /* character from .rbt file */

/* Xilinx configuration variables */
int din=0;                          /* configuration data bit from file */
int done1=0;                        /* configuration complete for MVDCTRL */
int done2=0;                        /* configuration complete for MVDALM  */
int pc_init1=0;                     /* configuration error for MVDCTRL  */
int pc_init2=0;                     /* configuration error for MVDALM  */

/* offsets for configuration data */
int pc_din=0x10;                    /* offset for configuration data for
controller */
int pc_cclk=0x20;                   /* offset for configuration clock for
controller */
int pc_prog=0x40;                   /* offset for bit to clear Xilinx
configuration memory (/PROG) */
int pc_din2=0x40;                   /* offset for configuration data for ALM */
int pc_cclk2=0x80;                  /* offset for configuration clock for ALM */

/* masks for configuration flags */
int pc_init1_mask=0x100;            /* mask for configuration error for MVDCTRL */
int pc_init2_mask=0x200;            /* mask for configuration error for MVDALM  */
int done1_mask=0x400;               /* mask for configuration DONE for MVDCTRL */
int done2_mask=0x800;               /* mask for configuration DONE for MVDALM  */

/* end Xilinx Configuration Interface */

/* Mode Control Interface */
/* mode bits */
int mb[5];                          /* LVL-1 trigger is mb[4] */

/* mode bit offsets */
int mb_offset=0x100;
int cpu_reset=0x80;

/* end Mode Control Interface */

/* Serial Interface */
/* HM serial offsets */
int hm_sclk=0x2;                    /* HM serial clock offset */
int hm_sloadb=0x4;                  /* HM load offset */

int ir_freq=1000;                   /* default IR frequency */
int lvl1_delay=1;                   /* spacing between triggers in Raw Mode */
int corr_mode=0;                    /* correlator enable flag */

/* serial shift data */
unsigned int sr1_din[8];            /* serial shift data 1 */
unsigned int sr2_din[8];            /* serial shift data 2 */
int cal_en=0;                       /* calibration enable */

/* HM Serial input */
unsigned int hm_din[18];            /* HM serial string */

/* serial shift offsets */
int sr1_din_offset=0x100;           /* serial shift data 1 offset */
int sr2_din_offset=0x1000;          /* serial shift data 2 offset */
int sr1_sclk=0x200;                 /* serial shift clock 1 offset */
int sr2_sclk=0x2000;                /* serial shift clock 2 offset */
int sr1_rst=0x400;                  /* serial shift reset 1 offset */
int sr2_rst=0x4000;                 /* serial shift reset 2 offset */

/* DAC shift strings */
```

```
/* Board 1 DACs */
unsigned int dac1A_sh[12];      /* DAC A - Vref  */
unsigned int dac1B_sh[12];      /* DAC B - Vth   */
unsigned int dac1C_sh[12];      /* DAC C - Vgate */

/* Board 2 DACs */
unsigned int dac2A_sh[12];      /* DAC A - Vref  */
unsigned int dac2B_sh[12];      /* DAC B - Vth   */
unsigned int dac2C_sh[12];      /* DAC C - Vgate */

/* DAC serial data offsets */
int dac1_ldacb=0x1;             /* offset for load DAC Board 1 input */
int dac2_ldacb=0x10;            /* offset for load DAC Board 2 input */
int dac1_din=0x2;               /* offset for serial data DAC Board 1 */
int dac2_din=0x20;              /* offset for serial data DAC Board 2 */
int dac1_sclk=0x4;              /* offset for DAC Board 1 clock */
int dac2_sclk=0x40;             /* offset for DAC Board 2 clock */
int dac1_rstb=0x8;              /* offset for DAC Board 1 reset */
int dac2_rstb=0x80;             /* offset for DAC Board 2 reset */
int dac1_csb=0x10;              /* DAC Board 1 chip select */
int dac2_csb=0x20;              /* DAC Board 2 chip select */


/* DAC voltages */
float dac1_volt[12];            /* DAC Board 1 output voltage */
float dac2_volt[12];            /* DAC Board 2 output voltage */
int dac1_bin[12];               /* 8-bit binary equivalent of DAC 1 voltage */
int dac2_bin[12];               /* 8-bit binary equivalent of DAC 2 voltage */
unsigned int bit_mask=0;        /* bit-wise mask for shift values */

/* default DAC voltages */
int vref_init=180;              /* Vref DAC startup voltage */
int vth_init=0;                 /* Vth DAC startup voltage */
int vgate_init=0;               /* Vgate DAC startup voltage */

/* define variables for menu choices */
char schoice1[1];       /* main menu choice character */
int choice1=0;          /* integer equivalent of main menu choice */

char schoice2[1];       /* MCI choice character */
int choice2=0;          /* integer equivalent of MCI choice */

char schoice3[1];       /* DCM choice character */
int choice3=0;          /* integer equivalent of DCM choice */

char schoice4[1];       /* Serial choice character */
int choice4=0;          /* integer equivalent of Serial choice */

char schoice5[1];       /* Xilinx choice character */
int choice5=0;          /* integer equivalent of Xilinx choice */

char schoicef[1];       /* IR choice character */
int choicef=0;          /* integer equivalent of IR choice */

char schoicel[1];       /* LVL-1 Spacing choice character */
int choicel=0;          /* integer equivalent of LVL-1 spacing choice */

char schoicem[1];       /* MUX choice character */
int choicem=0;          /* integer equivalent of MUX choice */

char schoicen[1];       /* Channel Enable choice character */
int choicen=0;          /* integer equivalent of Channel Enable choice */
```

```
char schoiced[1];       /* DAC choice character */
int choiced=0;          /* integer equivalent of DAC choice */

char schoicev[1];       /* DAC voltage choice character */
int choicev=0;          /* integer equivalent of DAC choice */

char schoiceb[1];       /* DAC block change choice character */
int choiceb=0;          /* integer equivalent of DAC block choice */

char schoicec[1];       /* correlator choice character */
int choicec=0;          /* integer equivalent of correlator choice */


void main(){

  /* function declarations */
  void reset_fifo(void);
  void sys_reset(void);
  void reset_dacs(void);
  void init_io_boards(void);
  void init_fifo(void);
  void hm_load(void);
  void hm_shift(void);
  void load_dacs(void);
  void init_dac_strings(void);
  void clear_mode_bits(void);
  void sys_init(void);
  void read_fifo_flags(void);
  void pcfifo_ren(void);
  void pcfifo_renb(void);
  void ir_mode(void);
  void resynch_pipeline_mode(void);
  void cont_run_mode(void);
  void cont_run_lvl1_mode(void);
  void cont_run_xlvl1s_mode(void);
  void halt_all_mode(void);
  void halt_acq_mode(void);
  void read_packet(void);
  void fifo_read_clock(void);
  void screen_print_packet(void);
  void file_print_packet(void);
  void hit_key(void);
  void sr_shift(void);
  void cont_packet_print(void);
  void open_new_ctrl(void);
  void open_new_dat(void);
  void open_new_alm(void);
  void read_xilinx_flags(void);
  void sr_reset(void);
  void clear_xilinx(void);
  void clock_xilinx1(void);
  void clock_xilinx2(void);
  void program_controller(void);
  void program_alm(void);
  void check_fpga_config(void);
  void dac_noop(void);
  void update_board1_dacA(void);
  void update_board1_dacB(void);
  void update_board1_dacC(void);
  void update_board2_dacA(void);
  void update_board2_dacB(void);
  void update_board2_dacC(void);
  void update_all_dacs(void);
```

```
void dcm_interface(void);
void mode_control_interface(void);
void enter_dac_voltage(void);
void check_dac_voltage(void);
void serial_mode_interface(void);
void SI_choice1(void);
void SI_choice2(void);
void SI_choice3(void);
void SI_choice4(void);
void SI_choice5(void);
void SI_choice6(void);
void SI_choice7(void);
void SI_choice8(void);
void SI_choice9(void);
void SI_choice10(void);
void xilinx_configuration_interface(void);

/* initialize DAC arrays */
for(i=0;i<12;i++){
  dac1_volt[i]=0.0;
  dac2_volt[i]=0.0;
  dac1_bin[i]=0;
  dac2_bin[i]=0;
  dac1A_sh[i]=0;
  dac1B_sh[i]=0;
  dac1C_sh[i]=0;
  dac2A_sh[i]=0;
  dac2B_sh[i]=0;
  dac2C_sh[i]=0;
}
cpu_reset=8*mb_offset;              /* re-define cpu_reset as mode bit 3 */

/* initialize HM serial string */
for(i=0;i<18;i++){
  hm_din[i]=0;
}

/* initialize serial strings */
for(i=0;i<8;i++){
  sr1_din[i]=1;
  sr2_din[i]=1;
}

/* initialize data packet */
for(i=0;i<128;i++){
  pack_dat[i]=0;
}

/* open packet data file */
if((fpd=fopen(fdat,"a"))==NULL){
  printf("\nError opening packet data file %s\n",fdat);
}

/* initialize system */
sys_init();

/* main menu */
while(choice1 != 7){

  clrscr();                          /* clear screen */
  printf("\t\tMVD Beam Test Main Menu\n");
  printf("\t\t-----------------------\n\n");
  printf("1) Mode Control Interface\n");
```

```
    printf("2) DCM Interface\n");
    printf("3) Serial Interface\n");
    printf("4) Xilinx Configuration Interface\n");
    printf("5) Reset Cheezy Cards\n");
    printf("6) Return to Initialization\n");
    printf("7) QUIT\n");
    gets(schoice1);
    sscanf(schoice1,"%d",&choice1);

    /* make sure menu choice is valid */
    while(choice1>7 || choice1<1){
      gets(schoice1);
      sscanf(schoice1,"%d",&choice1);
    }

    /* Mode Control Interface */
    if(choice1==1){
      mode_control_interface();
    } /* end if(choice==1) */

    /* DCM Interface */
    if(choice1==2){
      dcm_interface();
    } /* end if(choice==2) */

    /* Serial Mode Interface */
    if(choice1==3){
      serial_mode_interface();
    } /* end if(choice1==3) */

    /* Xilinx Configuration Interface */
    if(choice1==4){
      xilinx_configuration_interface();
    } /* end if(choice1==4) */

    if(choice1==5){
      outpw(board1_port1,0);
      outpw(board1_port2,0);
      outpw(board2_port1,0);
    }

    if(choice1==6){
      /* re-init system */
      sys_init();
    }

    if(choice1 != 7) choice1=0;
  } /* end Main Menu - while(choice1 !=7) */

  fclose(fpd);
  clrscr();
  printf("See you next time in the wonderful world of physics
experiments!!!");
  printf("\nBrought to you by Doogie allenmd");
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
} /* end main() */


/*************************************************************************/
/*                      Function Definitions                             */
/*-----------------------------------------------------------------------*/
```

```
void reset_fifo(void){

  /* reset external fifo */
  control2=control2&(~pcfifo_rs);
  outpw(board2_port1,control2);
  control2=control2|pcfifo_rs;
  outpw(board2_port1,control2);

} // void reset_fifo(void)


void sys_reset(void){

  /* reset system */
  control11=control11|cpu_reset;
  outpw(board1_port1,control11);

  for(i=0;i<100;i++);                    /* hold system reset */

  control11=control11&(~cpu_reset);
  outpw(board1_port1,control11);

} // void sys_reset(void)


void reset_dacs(void){

  /* reset DACs */
  control12=control12&(~dac1_rstb);
  control12=control12&(~dac2_rstb);
  outpw(board1_port2,control12);
  control12=control12|dac1_rstb;
  control12=control12|dac2_rstb;
  outpw(board1_port2,control12);

} //void reset_dacs(void)


void init_io_boards(void){

  /* initialize I/O boards for standard operation mode */
  control11=sr2_rst+sr1_rst+pc_prog+hm_sloadb;
  outpw(board1_port1,control11);

  /* start up in RUN mode */
  control12=dac1_ldacb+dac2_ldacb+dac1_rstb+dac2_rstb+2*mb_offset+mb_offset;
  outpw(board1_port2,control12);

  control2=pcfifo_rs+pcfifo_load+pcfifo_r+dac1_csb+dac2_csb;
  outpw(board2_port1,control2);

} // void init_io_boards(void)


void init_fifo(void){

  /* initialize fifo by toggling read clock */
  control2=control2|pcfifo_rs;
  control2=control2|pcfifo_r;
  control2=control2|pcfifo_load;
  control2=control2&(~pcfifo_rclk);
  outpw(board2_port1,control2);
```

```
  control2=control2|pcfifo_rclk;        /* raise read clock */
  outpw(board2_port1,control2);

  control2=control2&(~pcfifo_rclk);     /* lower read clock */
  outpw(board2_port1,control2);

} // void init_fifo(void)


void hm_load(void){

  /* load HM mirror register */
  control11=control11&(~hm_sloadb);     /* lower hm_sloadb */
  outpw(board1_port1,control11);

  control11=control11|hm_sloadb;        /* raise hm_sloadb */
  outpw(board1_port1,control11);

} // void hm_load(void)


void hm_shift(void){

  /* shift in HM string */
  for(i=0;i<18;i++){
    control11=control11|hm_din[i];      /* set up data */
    outpw(board1_port1,control11);

    control11=control11|hm_sclk;        /* clock in data */
    outpw(board1_port1,control11);

    control11=control11&(~hm_sclk);     /* remove clock */
    outpw(board1_port1,control11);

    control11=control11&(~hm_din[i]);   /* re-init control11 */
  }

  hm_load();

} // void hm_shift(void)


void load_dacs(void){

  /* load DACs with serial strings */

  /* enable DACs */
  control2=control2&(~dac2_csb);
  control2=control2&(~dac1_csb);
  outpw(board2_port1,control2);

  /* clock in serial strings for Vgate DACs */
  for(i=0;i<12;i++){

    /* set up data */
    control12=control12|(dac2_din*dac2C_sh[i]);
    control12=control12|(dac1_din*dac1C_sh[i]);
    outpw(board1_port2,control12);

    /* clock in data */
    control12=control12|dac2_sclk;
    control12=control12|dac1_sclk;
```

```
  outpw(board1_port2,control12);
  control12=control12&(~dac2_sclk);
  control12=control12&(~dac1_sclk);
  outpw(board1_port2,control12);

  /* clear data from control12 */
  control12=control12&(~(dac2_din*dac2C_sh[i]));
  control12=control12&(~(dac1_din*dac1C_sh[i]));
}

/* clock in serial strings for Vth DACs */
for(i=0;i<12;i++){

  /* set up data */
  control12=control12|(dac2_din*dac2B_sh[i]);
  control12=control12|(dac1_din*dac1B_sh[i]);
  outpw(board1_port2,control12);

  /* clock in data */
  control12=control12|dac2_sclk;
  control12=control12|dac1_sclk;
  outpw(board1_port2,control12);
  control12=control12&(~dac2_sclk);
  control12=control12&(~dac1_sclk);
  outpw(board1_port2,control12);

  /* clear data from control12 */
  control12=control12&(~(dac2_din*dac2B_sh[i]));
  control12=control12&(~(dac1_din*dac1B_sh[i]));
}

/* clock in serial strings for Vref DACs */
for(i=0;i<12;i++){

  /* set up data */
  control12=control12|(dac2_din*dac2A_sh[i]);
  control12=control12|(dac1_din*dac1A_sh[i]);
  outpw(board1_port2,control12);

  /* clock in data */
  control12=control12|dac2_sclk;
  control12=control12|dac1_sclk;
  outpw(board1_port2,control12);
  control12=control12&(~dac2_sclk);
  control12=control12&(~dac1_sclk);
  outpw(board1_port2,control12);

  /* clear data from control12 */
  control12=control12&(~(dac2_din*dac2A_sh[i]));
  control12=control12&(~(dac1_din*dac1A_sh[i]));
}

/* disable DACs */
control2=control2|dac2_csb;
control2=control2|dac1_csb;
outpw(board2_port1,control2);

/* toggle LDACB (load DAC) */
control12=control12&(~dac1_ldacb);
control12=control12&(~dac2_ldacb);
outpw(board1_port2,control12);

control12=control12|dac1_ldacb;
```

```
  control12=control12|dac2_ldacb;
  outpw(board1_port2,control12);

} // void load_dacs(void)


void clear_mode_bits(void){

  /* clear mode bits from control12 register */
  control12=control12&(~(16*mb_offset));
  control12=control12&(~(8*mb_offset));
  control12=control12&(~(4*mb_offset));
  control12=control12&(~(2*mb_offset));
  control12=control12&(~(mb_offset));

} // void clear_mode_bits(void)


void read_fifo_flags(void){

  /* read flags from external fifo */
  ff=(inpw(board2_port2) & ff_mask)/ff_mask;
  paf=(inpw(board2_port2) & paf_mask)/paf_mask;
  hf=(inpw(board2_port2) & hf_mask)/hf_mask;
  pae=(inpw(board2_port2) & pae_mask)/pae_mask;
  ef=(inpw(board2_port2) & ef_mask)/ef_mask;

} // void read_fifo_flags(void)


void pcfifo_ren(void){

  /* enable read for external fifo */
  control2=control2&(~pcfifo_r);
  outpw(board2_port1,control2);

} // void pcfifo_ren(void)


void pcfifo_renb(void){

  /* disable read for external fifo */
  control2=control2|pcfifo_r;
  outpw(board2_port1,control2);

} // void pcfifo_renb(void)


void ir_mode(void){

  /* Integrator Reset mode 00001 */
  clear_mode_bits();
  control12=control12|mb_offset;        /* mode bit 0 */
  outpw(board1_port2,control12);

} // void ir_mode(void)


void resynch_pipeline_mode(void){

  /* resynch pipeline mode 00010 */
  clear_mode_bits();
  control12=control12|(2*mb_offset);    /* mode bit 1 */
```

```
  outpw(board1_port2,control12);

} // void resynch_pipeline_mode(void)


void cont_run_mode(void){

  /* continuous run mode (no LVL-1s) 00011 */
  clear_mode_bits();
  control12=control12|(2*mb_offset);     /* mode bit 1 */
  control12=control12|mb_offset;         /* mode bit 0 */
  outpw(board1_port2,control12);

} // void cont_run_mode(void)


void cont_run_lvl1_mode(void){

  /* run mode with single LVL-1 10011 */
  clear_mode_bits();
  control12=control12|(2*mb_offset);     /* mode bit 1 */
  control12=control12|mb_offset;         /* mode bit 0 */
  outpw(board1_port2,control12);

  for(i=0;i<30;i++);                     /* delay for LVL-1 */

  control12=control12|(16*mb_offset);    /* mode bit 4 */
  outpw(board1_port2,control12);

  for(i=0;i<30;i++);                     /* hold LVL-1 */

  control12=control12&(~(16*mb_offset));
  outpw(board1_port2,control12);

} // void cont_run_lvl1_mode(void)


void cont_run_xlvl1s_mode(void){

  /* continuous run with multiple LVL1s */

  clrscr();
  printf("Triggering continuously");
  printf("\n\nHit any key to stop");
  while(!kbhit()){
    for(i=0;i<2000;i++);                 /* delay for resort */
    cont_run_lvl1_mode();
  }
  getch();                               /* eat any character in buffer */
  printf("\n");

} // void cont_run_xlvl1s_mode(void)


void halt_all_mode(void){

  /* halt all mode 00100 */

  clear_mode_bits();
  control12=control12|(4*mb_offset);     /* mode bit 2 */
  outpw(board1_port2,control12);

} // void halt_all_mode(void)
```

```
void halt_acq_mode(void){

  /* halt acquisition mode 00110 */

  control12=control12|(4*mb_offset);     /* mode bit 2 */
  control12=control12|(2*mb_offset);     /* mode bit 1 */
  outpw(board1_port2,control12);

} // void halt_acq_mode(void)


void fifo_read_clock(void){

  /* pulse fifo read clock */

  control2=control2|pcfifo_rclk;
  outpw(board2_port1,control2);
  control2=control2&(~pcfifo_rclk);
  outpw(board2_port1,control2);

} // void fifo_read_clock(void)


void read_packet(void){

  /* get data packet from PC FIFO */

  /* provide clock to fifo for startup case */
  init_fifo();

  /* enable read */
  pcfifo_ren();

  /* read data for one packet */
  for(i=0;i<71;i++){
    read_fifo_flags();

    if(ef==1){
      fifo_read_clock();
      packet=inpw(board2_port1)&packet_mask;
      pack_dat[i]=packet;
    }
    else{
      pack_dat[i]=9999;          /* no packet data present */
    } /* end if(ef==1){ */
  }

  /* disable fifo read clock */
  pcfifo_renb();

} // void read_packet(void)


void hit_key(void){

  /* hit a key to continue */

  printf("\n\nHit any key to continue");
  while(!kbhit());
  getch();                              /* eat any character in buffer */
  clrscr();
```

```
  printf("\n");

} // void hit_key(void)


void screen_print_packet(void){

  /* print packet to screen */

  for(i=0;i<71;i++){
    if(i==0 || i==1){
      printf("Header = ");
    }
    if(i==2){
      printf("Event Number = ");
    }
    if(i==3){
      printf("Module # = ");
    }
    if(i==4){
      printf("AMU Cell # = ");
    }
    if(i==5){
      printf("\nBoard 1\n");
      printf("-------\n");
    }
    if(i==37){
      printf("\nBoard 2\n");
      printf("-------\n");
    }
    if(i==69){
      printf("\n");
    }
    if(i>68){
      printf("Trailer = ");
    }
    if(i>4 && i<69){
      printf("Channel %d = ",(i-5));
    }
    if(pack_dat[i] != 9999){
      printf("\t%d\n",pack_dat[i]);
    }
    else{
      printf("\tPacket for #%d not found\n",i);
    }
    if(i==4 || i==20 || i==36 || i==52){
      hit_key();
    }

  } /* end for(i=0;i<71;i++) */
  hit_key();

} // void screen_print_packet(void)


void file_print_packet(void){

  /* print packet to file */

  for(i=0;i<71;i++){
    if(i==0 || i==1){
      fprintf(fpd,"Header = ");
    }
```

```
    if(i==2){
      fprintf(fpd,"Event Number = ");
    }
    if(i==3){
      fprintf(fpd,"Module # = ");
    }
    if(i==4){
      fprintf(fpd,"AMU Cell # = ");
    }
    if(i==5){
      fprintf(fpd,"\nBoard 1\n");
      fprintf(fpd,"-------\n");
    }
    if(i==37){
      fprintf(fpd,"\nBoard 2\n");
      fprintf(fpd,"-------\n");
    }
    if(i==69){
      fprintf(fpd,"\n");
    }
    if(i>68){
      fprintf(fpd,"Trailer = ");
    }
    if(i>4 && i<69){
      fprintf(fpd,"Channel %d = ",(i-5));
    }
    if(pack_dat[i] != 9999){
      fprintf(fpd,"\t%d\n",pack_dat[i]);
    }
    else{
      fprintf(fpd,"\tPacket for #%d not found\n",i);
    }

  } /* end for(i=0;i<71;i++) */
  fprintf(fpd,"\n");

} // void file_print_packet(void)


void sr_shift(void){

  /* shift in calibration enables */

  /* enable shift registers */
  control11=control11&(~sr1_rst);
  control11=control11&(~sr2_rst);
  outpw(board1_port1,control11);

  for(i=0;i<8;i++){

    /* shift enables */
    control11=control11|(sr1_din[i]*sr1_din_offset);
    control11=control11|(sr2_din[i]*sr2_din_offset);
    outpw(board1_port1,control11);

    /* clock in data */
    control11=control11|sr1_sclk;
    control11=control11|sr2_sclk;
    outpw(board1_port1,control11);

    control11=control11&(~sr1_sclk);
    control11=control11&(~sr2_sclk);
    outpw(board1_port1,control11);
```

```
    /* clear data */
    control11=control11&(~(sr1_din[i]*sr1_din_offset));
    control11=control11&(~(sr2_din[i]*sr2_din_offset));
  }
  outpw(board1_port1,control11);

} // void sr_shift(void)


void cont_packet_print(void){

  /* read out continuously all data in FIFO and format as a packet */

  /* provide clock to fifo for startup case */
  init_fifo();

  read_fifo_flags();

  /* check to see if data is present */
  if(ef==1){
    pcfifo_ren();
    fifo_read_clock();
    packet=inpw(board2_port1)&packet_mask;
    pack_dat[i]=packet;
    i=i+1;
  } /* end if(ef==1){ */

  pcfifo_renb();

  /* end of packet */
  if(i==71){
    clrscr();
    printf("Continuous Read Mode\n");
    printf("--------------------\n");
    printf("Hit Any Key to Stop\n\n");
    for(i=0;i<18;i++){
      printf("#%2d=%4d\t",i,pack_dat[i]);
      printf("#%2d=%4d\t",(i+18),pack_dat[i+18]);
      printf("#%2d=%4d\t",(i+36),pack_dat[i+36]);
      printf("#%2d=%4d\n",(i+54),pack_dat[i+54]);
    }
    for(j=0;j<1000000;j++);
    i=0;
  }

} // void cont_packet_print(void)


void open_new_ctrl(void){

  /* open new controller configuration (.rbt) file */

  fclose(fpctrl);
  clrscr();
  printf("The current controller file is: %s\n\n",fctrl);
  printf("Please enter name for new controller file:\n");
  scanf("%s",fctrl);
  if((fpctrl=fopen(fctrl,"r"))==NULL){
    clrscr();
    printf("\nError: Cannot open file %s (default file %s will be used ");
    printf("instead).\n",fctrl,fctrl_def);
    hit_key();
```

```
    for(i=0;i<15;i++){
      fctrl[i]=fctrl_def[i];
    }
    if((fpctrl=fopen(fctrl,"r"))==NULL){
      clrscr();
      printf("Error: could not open %s\n",fctrl);
      hit_key();
   }
 }

} // void open_new_ctrl(void)


void open_new_dat(void){

  /* open new data log file */

  if((fpd=fopen(fdat,"a"))==NULL){
    clrscr();
    printf("\nError: Cannot open packet data file ");
    printf("%s (default file %s ",fdat,fdat_def);
    printf("will be used instead).\n");
    hit_key();
    for(i=0;i<15;i++){
      fdat[i]=fdat_def[i];
    }
    if((fpd=fopen(fdat,"a"))==NULL){
      printf("\nError: Cannot open packet data file %s\n",fdat);
      hit_key();
    }
  }

} // void open_new_dat(void)


void open_new_alm(void){

  /* open new ALM configuration (.rbt) file */

  if((fpalm=fopen(falm,"r"))==NULL){
    clrscr();
    printf("\nError: Cannot open packet data file ");
    printf("%s (default file %s ",falm,falm_def);
    printf("will be used instead).\n");
    hit_key();
    for(i=0;i<15;i++){
      falm[i]=falm_def[i];
    }
    if((fpalm=fopen(falm,"r"))==NULL){
      clrscr();
      printf("Error: could not open %s\n",falm);
      hit_key();
    }
  }

} // void open_new_alm(void)


void read_xilinx_flags(void){

  /* read Xilinx programming flags */
  pc_init1=(inpw(board1_port2) & pc_init1_mask)/pc_init1_mask;
  pc_init2=(inpw(board1_port2) & pc_init2_mask)/pc_init2_mask;
```

```
  done1=(inpw(board1_port2) & done1_mask)/done1_mask;
  done2=(inpw(board1_port2) & done2_mask)/done2_mask;

} // void read_xilinx_flags(void)


void sr_reset(void){

  /* reset calibration shift register */
  control11=control11|sr1_rst;
  control11=control11|sr2_rst;
  outpw(board1_port1,control11);

} // void sr_reset(void)


void clear_xilinx(void){

  /* clear Xilinx configuration memory */
  for(i=0;i<250;i++){
    outpw(board1_port1,control11);
  }
  control11=control11&(~pc_prog);

  for(i=0;i<22250;i++){
    outpw(board1_port1,control11);
  }
  control11=control11|pc_prog;

  for(i=0;i<2500;i++){
    outpw(board1_port1,control11);
  }

} // void clear_xilinx(void)


void clock_xilinx1(void){

  /* clock in data for Xilinx Part 1 (HM Controller) */

  /* set up configuration bit */
  control11=control11|(pc_din*din);
  outpw(board1_port1,control11);

  /* clock configuration data */
  control11=control11|pc_cclk;
  outpw(board1_port1,control11);
  control11=control11&(~pc_cclk);
  outpw(board1_port1,control11);

  /* reset control vector */
  control11=control11&(~(pc_din*din));

  outpw(board1_port1,control11);

} // void clock_xilinx1(void)


void clock_xilinx2(void){

  /* clock in data for Xilinx Part 2 (ALM) */

  /* set up configuration bit */
```

```
  control2=control2|(pc_din2*din);
  outpw(board2_port1,control2);

  /* clock configuration data */
  control2=control2|pc_cclk2;
  outpw(board2_port1,control2);
  control2=control2&(~pc_cclk2);
  outpw(board2_port1,control2);

  /* reset control vector */
  control2=control2&(~(pc_din2*din));

  outpw(board2_port1,control2);

} // void clock_xilinx2(void)


void program_controller(void){

  /* program heap manager controller FPGA (XC4010) */

  line_count=0;                  /* reset line count */
  while((rbt=fgetc(fpctrl)) != EOF){

    /* increment line count when CR is encountered */
    if(rbt==10){
      line_count++;
    }

    if(line_count>5){          /* ignore .rbt header */

      /* set din to 0 or 1 */
      if(rbt=='0'){
        din=0;
      }
      if(rbt=='1'){
        din=1;
      }

      /* ignore all characters but 0 or 1 */
      if(rbt=='0' || rbt=='1'){
        clock_xilinx1();
        /* read next configuration bit from file */
      }
    }  /* end if(line_count>5) */

  } /* end while(rbt=fgetc(fpctrl)) != EOF){ */

  /* final clock (this is necessary to enable the configuration) */
  control11=control11|pc_din;
  control11=control11|pc_cclk;
  outpw(board1_port1,control11);

  /* clear clock and data */
  control11=control11&(~pc_din);
  control11=control11&(~pc_cclk);

} // void program_controller(void)


void program_alm(void){

  /* program address list manager FPGA (XC4005) */
```

```
  line_count=0;                  /* reset line count */
  while((rbt=fgetc(fpalm)) != EOF){

    /* increment line count when CR is encountered */
    if(rbt==10){
      line_count++;
    }

    if(line_count>5){            /* ignore .rbt header */

      /* set din to 0 or 1 */
      if(rbt=='0'){
        din=0;
      }
      if(rbt=='1'){
        din=1;
      }

      /* ignore all characters but 0 or 1 */
      if(rbt=='0' || rbt=='1'){
        clock_xilinx2();
        /* read next configuration bit from file */
      }
    } /* end if(line_count>5) */

  } /* end while(rbt=fgetc(fpalm)) != EOF){ */

  /* final clock (this is necessary to enable the configuration) */
  control2=control2|pc_din2;
  control2=control2|pc_cclk2;
  outpw(board2_port1,control2);

  /* clear clock and data */
  control2=control2&(~pc_din2);
  control2=control2&(~pc_cclk2);

} // void program_alm(void)


void check_fpga_config(void){

  /* check FPGA configuration status */

  /* delay to allow DONE bits to be set */
  for(i=0;i<1000;i++);

  /* check done flags */
  done1=(inpw(board1_port2) & done1_mask)/done1_mask;
  done2=(inpw(board1_port2) & done2_mask)/done2_mask;

  /* if DONE bits are not high, warn user */
  if(done1 == 0){
    clrscr();
    printf("\nWARNING: Controller DONE bit is NOT HIGH\n");
    hit_key();
  }

  if(done2 == 0){
    clrscr();
    printf("\nWARNING: ALM DONE bit is NOT HIGH\n");
    hit_key();
  }
```

```
} // void check_fpga_config(void)


void dac_noop(void){

  /* set up all DACs for NOOP (no operation) */

  /* A1=X; A0=1; C1=0; C0=0 */
  dac1A_sh[0]=dac1B_sh[0]=dac1C_sh[0]=dac2A_sh[0]=dac2B_sh[0]=dac2C_sh[0]=0;
  dac1A_sh[1]=dac1B_sh[1]=dac1C_sh[1]=dac2A_sh[1]=dac2B_sh[1]=dac2C_sh[1]=1;
  dac1A_sh[2]=dac1B_sh[2]=dac1C_sh[2]=dac2A_sh[2]=dac2B_sh[2]=dac2C_sh[2]=0;
  dac1A_sh[3]=dac1B_sh[3]=dac1C_sh[3]=dac2A_sh[3]=dac2B_sh[3]=dac2C_sh[3]=0;

} // void dac_noop(void)


void update_board1_dacA(void){

  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac1A_sh[1]=0;

} // void update_board1_dacA(void)


void update_board1_dacB(void){

  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac1B_sh[1]=0;

} // void update_board1_dacB(void)


void update_board1_dacC(void){

  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac1C_sh[1]=0;

} // void update_board1_dacC(void)


void update_board2_dacA(void){

  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac2A_sh[1]=0;

} // void update_board2_dacA(void)


void update_board2_dacB(void){
```

```
  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac2B_sh[1]=0;

} // void update_board2_dacB(void)


void update_board2_dacC(void){

  /* assume noop */
  dac_noop();

  /* A1=X; A0=0; C1=0; C0=0 */
  dac2C_sh[1]=0;

} // void update_board2_dacC(void)


void update_all_dacs(void){

  /* A1=X; A0=0; C1=0; C2=0 */
  dac1A_sh[0]=dac1B_sh[0]=dac1C_sh[0]=dac2A_sh[0]=dac2B_sh[0]=dac2C_sh[0]=0;
  dac1A_sh[1]=dac1B_sh[1]=dac1C_sh[1]=dac2A_sh[1]=dac2B_sh[1]=dac2C_sh[1]=0;
  dac1A_sh[2]=dac1B_sh[2]=dac1C_sh[2]=dac2A_sh[2]=dac2B_sh[2]=dac2C_sh[2]=0;
  dac1A_sh[3]=dac1B_sh[3]=dac1C_sh[3]=dac2A_sh[3]=dac2B_sh[3]=dac2C_sh[3]=0;

} // void update_all_dacs(void)


void init_dac_strings(void){

  /* set DACs to initialization values */

  for(i=0;i<4;i++){
    dac1_bin[i]=dac2_bin[i]=vref_init;
    dac1_volt[i]=dac2_volt[i]=5.0*((float)vref_init/256.0);
  }
  for(i=4;i<8;i++){
    dac1_bin[i]=dac2_bin[i]=vth_init;
    dac1_volt[i]=dac2_volt[i]=5.0*((float)vth_init/256.0);
  }
  for(i=8;i<12;i++){
    dac1_bin[i]=dac2_bin[i]=vgate_init;
    dac1_volt[i]=dac2_volt[i]=5.0*((float)vgate_init/256.0);
  }

  update_all_dacs();

  bit_mask=1;
  for(i=0;i<8;i++){
    dac1A_sh[11-i]=dac2A_sh[11-i]=(vref_init & bit_mask)/bit_mask;
    dac1B_sh[11-i]=dac2B_sh[11-i]=(vth_init & bit_mask)/bit_mask;
    dac1C_sh[11-i]=dac2C_sh[11-i]=(vgate_init & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }

} // void init_dac_strings(void)


void sys_init(void){
```

```
  /* initialize system */

  /* initialize I/O boards */
  init_io_boards();

  /* reset fifo */
  reset_fifo();

  /* initialize fifo */
  init_fifo();

  /* reset system */
  sys_reset();

  /* reset IR */
  hm_load();

  /* set up DACs */
  /* reset DACs */
  reset_dacs();

  init_dac_strings();

  load_dacs();

} // void sys_init(void)


void mode_control_interface(void){

  /* Mode Control Interface Menu */

  while(choice2 != 9){

    clrscr();           /* clear screen */

    printf("\t\tMode Control Interface\n");
    printf("\t\t----------------------\n\n");
    printf("1) Integrator Reset\n");
    printf("2) Resynch Pipeline\n");
    printf("3) Continuous Run (no LVL-1 trigger)\n");
    printf("4) Continuous Run (with LVL-1 trigger)\n");
    printf("5) Continuous Run (with multiple LVL-1 triggers)\n");
    printf("6) Halt All\n");
    printf("7) Halt Acquisition\n");
    printf("8) Reset System (CPU Reset)\n");
    printf("9) RETURN to Main Menu\n");
    gets(schoice2);
    sscanf(schoice2,"%d",&choice2);

    /* make sure menu choice is valid */
    while(choice2>9 || choice2<1){
      gets(schoice2);
      sscanf(schoice2,"%d",&choice2);
    }

    if(choice2==1){
      /* Integrator Reset */
      ir_mode();
    } /* end if(choice2==1) */

    else if(choice2==2){
      /* Resynch Pipeline */
```

```
      resynch_pipeline_mode();

      /* re-initialize HM serial */
      hm_load();
    } /* end if(choice2==2) */

    else if(choice2==3){
      /* Continuous Run (no LVL-1 trigger) */
      cont_run_mode();
    } /* end if(choice2==3) */

    else if(choice2==4){
     /* Continuous Run (with LVL-1 trigger) */
      cont_run_lvl1_mode();
    } /* end if(choice2==4) */

    else if(choice2==5){
      /* Continuous Run (with multiple LVL-1 triggers) */
      cont_run_xlvl1s_mode();
    } /* end if(choice2==5) */

    else if(choice2==6){
      /* Halt All */
      halt_all_mode();
    } /* end if(choice2==6) */

    else if(choice2==7){
      /* Halt Acquisition */
      halt_acq_mode();
    } /* end if(choice2==7) */

    else if(choice2==8){
      /* reset system */
      sys_reset();

      hm_load();
    } /* end if(choice2==8) */

    if(choice2 != 9) choice2=0; /* reset menu choice */

  } /* end Mode Control Interface - while(choice2 != 9) */
  choice1=0;            /* reset main menu */
  choice2=0;            /* reset MCI menu */

} // void mode_control_interface(void)


void dcm_interface(void){

  /* DCM Interface Menu */

  while(choice3 != 8){

    clrscr();           /* clear screen */

    printf("\t\tDCM Interface\n");
    printf("\t\t-------------\n\n");
    printf("1) Get Packet\n");
    printf("2) Continuous Read Mode\n");
    printf("3) Clear Log File\n");
    printf("4) Display FIFO Status\n");
    printf("5) Reset Fifo\n");
    printf("6) Change Log File (Currently: %s ",fdat);
```

```
printf("(if file exists, it will be appended))\n");
printf("7) Change Log File to Default (%s) ",fdat_def);
printf("(if file exists, it will be appended)\n");
printf("8) RETURN to Main Menu\n");
gets(schoice3);
sscanf(schoice3,"%d",&choice3);

/* make sure menu choice is valid */
while(choice3>8 || choice3<1){
  gets(schoice3);
  sscanf(schoice3,"%d",&choice3);
}

if(choice3==1){
  /* clear screen*/
  clrscr();

  /* get single packet */
  read_packet();

  /* print packet to screen */
  screen_print_packet();

  /* print packet to file */
  file_print_packet();

} /* end if(choice3==1) */

if(choice3==2){

  i=0;

  while(!kbhit()){
    cont_packet_print();
  } /* end while(!kbhit()) */

getch();      /* eat any character in buffer */
printf("\n");
} /* end if(choice3==2) */

if(choice3==3){
  fclose(fpd);
  if((fpd=fopen(fdat,"w"))==NULL){
    printf("\nError: Cannot re-open packet data file %s\n",fdat);
    hit_key();
  }
} /* end if(choice3==3) */

if(choice3==4){
  /* provide clock to fifo for startup case */
  init_fifo();

  /* read flags */
  read_fifo_flags();

  clrscr();
  printf("\nff  = %d",ff);
  printf("\npaf = %d",paf);
  printf("\nhf  = %d",hf);
  printf("\npae = %d",pae);
  printf("\nef  = %d",ef);

  hit_key();
```

```
    } /* end if(choice3==4) */

    if(choice3==5){
      reset_fifo();
    }

    if(choice3==6){
      fclose(fpd);
      clrscr();
      printf("The current log file is: %s\n\n",fdat);
      printf("Please enter name for new log file:\n");
      scanf("%s",fdat);
      open_new_dat();
    }

    if(choice3==7){
      fclose(fpd);
      clrscr();
      for(i=0;i<15;i++){
        fdat[i]=fdat_def[i];
      }
      open_new_dat();
    }

    if(choice3!=8) choice3=0;
  } /* end while(choice3 != 8) */
  choice1=0;              /* reset main menu */
  choice3=0;              /* reset DCM menu */

} // void dcm_interface(void)


void enter_dac_voltage(void){

  /* prompt user for decimal (0-255) voltage for DAC(s) */

  printf("Enter Decimal Equivalent of New Binary Voltage:\n");
  printf("-----------------------------------------------\n");
  gets(schoicev);
  sscanf(schoicev,"%d",&choicev);

} // void enter_dac_voltage(void)


void check_dac_voltage(void){

  /* make sure choice is valid */
  while(choicev>255 || choicev<0){
    clrscr();
    printf("Error: Voltage Out of Range (Full Scale=0-255)\n\n");
    enter_dac_voltage();
  }

} // void check_dac_voltage(void)


void SI_choice1(void){

  /* Serial Interface Menu choice 1 */

  while(choicef != 5){
    clrscr();
```

```
    printf("Current IR frequency is %d beamclocks\n\n",ir_freq);
    printf("Choose New Integrator Reset Frequency:\n");
    printf("1) Every 1000 Beamclocks\n");
    printf("2) Every 2000 Beamclocks\n");
    printf("3) Every 4000 Beamclocks\n");
    printf("4) Every 8000 Beamclocks\n");
    printf("5) Return to Serial Mode Interface Menu\n");
    gets(schoicef);
    sscanf(schoicef,"%d",&choicef);

    /* make sure menu choice is valid */
    while(choicef>5 || choicef<1){
      gets(schoicef);
      sscanf(schoicef,"%d",&choicef);
    }

    if(choicef==1){
      /* set IR frequency to 1000 beam clocks */
      ir_freq=1000;
      hm_din[0]=0;
      hm_din[1]=0;
    }
    else if(choicef==2){
      /* set IR frequency to 2000 beam clocks */
      ir_freq=2000;
      hm_din[0]=0;
      hm_din[1]=1;
    }
    else if(choicef==3){
      /* set IR frequency to 4000 beam clocks */
      ir_freq=4000;
      hm_din[0]=1;
      hm_din[1]=0;
    }
    else if(choicef==4){
      /* set IR frequency to 8000 beam clocks */
      ir_freq=8000;
      hm_din[0]=1;
      hm_din[1]=1;
    }

    hm_shift();

    if(choicef!=5) choicef=0;
  } /* end while(choicef != 5) */
  choicef=0;

} // void SI_choice1(void)


void SI_choice2(void){

  /* Serial Interface Menu choice 2 */

  while(choicem != 13){
    clrscr();
    printf("Choose Bit to Toggle:\n");
    printf("---------------------\n\n");
    printf("1) M1A2  = %d\n",hm_din[2]);
    printf("2) M1A1  = %d\n",hm_din[3]);
    printf("3) M1A0  = %d\n",hm_din[4]);
    printf("4) M2A3  = %d\n",hm_din[5]);
    printf("5) M2A2  = %d\n",hm_din[6]);
```

```
    printf("6) M2A1   = %d\n",hm_din[7]);
    printf("7) M2A0   = %d\n",hm_din[8]);
    printf("8) M3A3   = %d\n",hm_din[9]);
    printf("9) M3A2   = %d\n",hm_din[10]);
    printf("10) M3A1   = %d\n",hm_din[11]);
    printf("11) M3A0   = %d\n",hm_din[12]);
    printf("12) MUX Enable   = %d\n\n",hm_din[13]);
    printf("13) Return to Serial Interface Menu and Initiate Changes\n");
    gets(schoicem);
    sscanf(schoicem,"%d",&choicem);

    /* make sure menu choice is valid */
    while(choicem>13 || choicem<1){
      gets(schoicem);
      sscanf(schoicem,"%d",&choicem);
    }

    if(choicem>0 && choicem <13){
      hm_din[choicem+1]=abs(hm_din[choicem+1]-1);
    }

    if(choicem==13){
      /* shift in HM string */
      hm_shift();
    }/* end if(choicem==13) */

    if(choicem != 13) choicem=0;
  } /* end while(choicem != 13) */
  choicem=0;

} // void SI_choice2(void)


void SI_choice3(void){

  /* Serial Interface Menu choice 3 */

  clrscr();

  if(corr_mode==0){
    printf("Currently in Raw Mode\n\n");
  }
  if(corr_mode==1){
    printf("Currently in Correlator Mode\n\n");
  }

  printf("Select from the following choices:\n");
  printf("1) Raw Mode\n");
  printf("2) Correlator Mode\n");
  printf("3) Return to Serial Interface Menu\n");
  gets(schoicec);
  sscanf(schoicec,"%d",&choicec);

  /* make sure menu choice is valid */
  while(choicec>3 || choicec<1){
    gets(schoicec);
    sscanf(schoicec,"%d",&choicec);
  }
  if(choicec==1){
    corr_mode=0;
    hm_din[14]=0;
  }
  if(choicec==2){
```

```
    corr_mode=1;
    hm_din[14]=1;
  }
  if(choicec!=3){
    hm_shift();
  } /* end if(choicec!=3) */
  choicec=0;

} // void SI_choice3(void)


void SI_choice4(void){

  /* Serial Interface Menu choice 4 */

  while(choicel != 5){
    clrscr();
    printf("Current LVL-1 spacing in Raw Mode is ");
    printf("%d Beam Clock(s)\n\n",lvl1_delay);
    printf("Choose New LVL-1 Spacing:\n");
    printf("1) 1 Beam Clock\n");
    printf("2) 2 Beam Clocks\n");
    printf("3) 4 Beam Clocks\n");
    printf("4) 8 Beam Clocks\n");
    printf("5) Return to Serial Mode Interface Menu\n");
    gets(schoicel);
    sscanf(schoicel,"%d",&choicel);

    /* make sure menu choice is valid */
    while(choicel>5 || choicel<1){
      gets(schoicel);
      sscanf(schoicel,"%d",&choicel);
    }

    if(choicel==1){
      lvl1_delay=1;
      hm_din[16]=0;
      hm_din[17]=0;
    }
    else if(choicel==2){
      lvl1_delay=2;
      hm_din[16]=1;
      hm_din[17]=0;
    }
    else if(choicel==3){
      lvl1_delay=4;
      hm_din[16]=0;
      hm_din[17]=1;
    }
    else if(choicel==4){
      lvl1_delay=8;
      hm_din[16]=1;
      hm_din[17]=1;
    }

    hm_shift();

    if(choicel!=5) choicel=0;
  } /* end while(choicel != 5) */
  choicel=0;

} // void SI_choice4(void)
```

```
void SI_choice5(void){

  /* Serial Interface Menu choice 5 */

  while(choicen != 20){
    clrscr();
    printf("Choose Enable to Toggle (Enable=1 Disable=0):\n");
    printf("-------------------------------------------\n\n");
    printf("Note: Identical channels are the same chip-to-chip\n");
    printf("Board 1 (Channels 0-31) Board 2 (Channels 32-63)\n");
    printf("----------------------- ------------------------\n");
    for(i=0;i<8;i++){
      printf("Channel %d Enable = %d\tChannel %d Enable = %d\n",
        i,sr1_din[i],i+8,sr2_din[i]);
    }
    printf("16) Enable All Channels\n");
    printf("17) Disable All Channels\n");
    printf("18) Toggle Calibration Enable (Currently: Enable = %d)\n",cal_en);
    printf("19) Toggle cal_dis (Currently: cal_dis = %d)\n",hm_din[15]);
    printf("20) Return to Serial Interface Menu\n");
    gets(schoicen);
    sscanf(schoicen,"%d",&choicen);

    /* make sure menu choice is valid */
    while(choicen>20 || choicen<0){
      gets(schoicen);
      sscanf(schoicen,"%d",&choicen);
    }

    if(choicen<16){
      if(choicen<8){
        sr1_din[choicen]=abs(sr1_din[choicen]-1);
      }
      else{
        sr2_din[choicen-8]=abs(sr2_din[choicen-8]-1);
      }
    }

    if(choicen==16){
      for(i=0;i<8;i++){
        sr1_din[i]=1;
        sr2_din[i]=1;
      }
    }

    if(choicen==17){
      for(i=0;i<8;i++){
        sr1_din[i]=0;
        sr2_din[i]=0;
      }
    }

    if(choicen==18){    /* shift enables */
      if(cal_en==0){
        cal_en=1;
        sr_shift();
      }
      else{
        cal_en=0;
        sr_reset();
      }
    } /* end if(choice==18) */
```

```
    if(choicen==19){
      hm_din[15]=abs(hm_din[15]-1);
      hm_shift();
    } /* end if(choice==19) */

    if(choicen != 20) choicen=0;
  } /* end while(choicen != 20) */
  choicen=0;

} // void SI_choice5(void)


void SI_choice6(void){

  /* Serial Interface Menu choice 6 */

  choiced=0;
  while(choiced != 25){
    clrscr();
    printf("Make Selection from the Following\n");
    printf("(DAC A = Vref; DAC B = Vth; DAC C = Vgate)\n");
    printf("-------------------------------------------------\n");
    printf("1) Change Board 1 DAC A Channel 1 = %f V (binary = %d)\n",
      dac1_volt[0],dac1_bin[0]);
    printf("2) Change Board 1 DAC A Channel 2 = %f V (binary = %d)\n",
      dac1_volt[1],dac1_bin[1]);
    printf("3) Change Board 1 DAC A Channel 3 = %f V (binary = %d)\n",
      dac1_volt[2],dac1_bin[2]);
    printf("4) Change Board 1 DAC A Channel 4 = %f V (binary = %d)\n",
      dac1_volt[3],dac1_bin[3]);
    printf("5) Change Board 1 DAC B Channel 1 = %f V (binary = %d)\n",
      dac1_volt[4],dac1_bin[4]);
    printf("6) Change Board 1 DAC B Channel 2 = %f V (binary = %d)\n",
      dac1_volt[5],dac1_bin[5]);
    printf("7) Change Board 1 DAC B Channel 3 = %f V (binary = %d)\n",
      dac1_volt[6],dac1_bin[6]);
    printf("8) Change Board 1 DAC B Channel 4 = %f V (binary = %d)\n",
      dac1_volt[7],dac1_bin[7]);
    printf("9) Change Board 1 DAC C Channel 1 = %f V (binary = %d)\n",
      dac1_volt[8],dac1_bin[8]);
    printf("10) Change Board 1 DAC C Channel 2 = %f V (binary = %d)\n",
      dac1_volt[9],dac1_bin[9]);
    printf("11) Change Board 1 DAC C Channel 3 = %f V (binary = %d)\n",
      dac1_volt[10],dac1_bin[10]);
    printf("12) Change Board 1 DAC C Channel 4 = %f V (binary = %d)\n",
      dac1_volt[11],dac1_bin[11]);
    printf("13) Change Board 2 DAC A Channel 1 = %f V (binary = %d)\n",
      dac2_volt[0],dac2_bin[0]);
    printf("14) Change Board 2 DAC A Channel 2 = %f V (binary = %d)\n",
      dac2_volt[1],dac2_bin[1]);
    printf("15) Change Board 2 DAC A Channel 3 = %f V (binary = %d)\n",
      dac2_volt[2],dac2_bin[2]);
    printf("16) Change Board 2 DAC A Channel 4 = %f V (binary = %d)\n",
      dac2_volt[3],dac2_bin[3]);
    printf("17) Change Board 2 DAC B Channel 1 = %f V (binary = %d)\n",
      dac2_volt[4],dac2_bin[4]);
    printf("18) Change Board 2 DAC B Channel 2 = %f V (binary = %d)\n",
      dac2_volt[5],dac2_bin[5]);
    printf("19) Change Board 2 DAC B Channel 3 = %f V (binary = %d)\n",
      dac2_volt[6],dac2_bin[6]);
    printf("20) Change Board 2 DAC B Channel 4 = %f V (binary = %d)\n",
      dac2_volt[7],dac2_bin[7]);
```

```
printf("21) Change Board 2 DAC C Channel 1 = %f V (binary = %d)\n",
  dac2_volt[8],dac2_bin[8]);
printf("22) Change Board 2 DAC C Channel 2 = %f V (binary = %d)\n",
  dac2_volt[9],dac2_bin[9]);
printf("23) Change Board 2 DAC C Channel 3 = %f V (binary = %d)\n",
  dac2_volt[10],dac2_bin[10]);
printf("24) Change Board 2 DAC C Channel 4 = %f V (binary = %d)\n",
  dac2_volt[11],dac2_bin[11]);
printf("25) Return to Serial Interface Menu\n");
gets(schoiced);
sscanf(schoiced,"%d",&choiced);

/* make sure menu choice is valid */
while(choiced>25 || choiced<1){
  gets(schoiced);
  sscanf(schoiced,"%d",&choiced);
}

clrscr();

if(choiced<25){
  /* assume NOP */
  dac_noop();
}

if(choiced<13){
  clrscr();

  enter_dac_voltage();
  check_dac_voltage();

  /* store binary and analog voltage in array */
  dac1_bin[choiced-1]=choicev;
  dac1_volt[choiced-1]=5.0*((float)choicev/256.0);

  /* set shift register */
  if(choiced<5){
    /* set channel */
    if(choiced==1){
      dac1A_sh[1]=0;
      dac1A_sh[0]=0;
    }
    if(choiced==2){
      dac1A_sh[1]=1;
      dac1A_sh[0]=0;
    }
    if(choiced==3){
      dac1A_sh[1]=0;
      dac1A_sh[0]=1;
    }
    if(choiced==4){
      dac1A_sh[1]=1;
      dac1A_sh[0]=1;
    }

    /* update single channel of DAC A on Board 1 */
    dac1A_sh[2]=dac1A_sh[3]=1;

    bit_mask=1;
    for(i=0;i<8;i++){
      dac1A_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }
```

```
} /* end if(choiced<5) */

if(choiced>4 && choiced<9){
  /* set channel */
  if(choiced==5){
    dac1B_sh[1]=0;
    dac1B_sh[0]=0;
  }
  if(choiced==6){
    dac1B_sh[1]=1;
    dac1B_sh[0]=0;
  }
  if(choiced==7){
    dac1B_sh[1]=0;
    dac1B_sh[0]=1;
  }
  if(choiced==8){
    dac1B_sh[1]=1;
    dac1B_sh[0]=1;
  }

  /* update single channel of DAC B on Board 1 */
  dac1B_sh[2]=dac1B_sh[3]=1;

  bit_mask=1;
  for(i=0;i<8;i++){
    dac1B_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }

} /* end if(choiced>4 && choiced<9) */

if(choiced>8 && choiced<13){
  /* set channel */
  if(choiced==9){
    dac1C_sh[1]=0;
    dac1C_sh[0]=0;
  }
  if(choiced==10){
    dac1C_sh[1]=1;
    dac1C_sh[0]=0;
  }
  if(choiced==11){
    dac1C_sh[1]=0;
    dac1C_sh[0]=1;
  }
  if(choiced==12){
    dac1C_sh[1]=1;
    dac1C_sh[0]=1;
  }

  /* update single channel of DAC C on Board 1 */
  dac1C_sh[2]=dac1C_sh[3]=1;

  bit_mask=1;
  for(i=0;i<8;i++){
    dac1C_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }
} /* end if(choiced>8 && choiced<13) */

/* set DAC voltage */
load_dacs();
```

```
} /* end if(choiced<13) */

if(choiced>12 && choiced<25){
  clrscr();

  enter_dac_voltage();
  check_dac_voltage();

  dac2_bin[choiced-13]=choicev;
  dac2_volt[choiced-13]=5.0*((float)choicev/256.0);

  /* set shift register */
  if(choiced<17){
    /* set channel */
    if(choiced==13){
      dac2A_sh[1]=0;
      dac2A_sh[0]=0;
    }
    if(choiced==14){
      dac2A_sh[1]=1;
      dac2A_sh[0]=0;
    }
    if(choiced==15){
      dac2A_sh[1]=0;
      dac2A_sh[0]=1;
    }
    if(choiced==16){
      dac2A_sh[1]=1;
      dac2A_sh[0]=1;
    }

    /* update single channel of DAC A on Board 2 */
    dac2A_sh[2]=dac2A_sh[3]=1;

    bit_mask=1;
    for(i=0;i<8;i++){
      dac2A_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }
  } /* end if(choiced<17) */

  if(choiced>16 && choiced<21){
    /* set channel */
    if(choiced==17){
      dac2B_sh[1]=0;
      dac2B_sh[0]=0;
    }
    if(choiced==18){
      dac2B_sh[1]=1;
      dac2B_sh[0]=0;
    }
    if(choiced==19){
      dac2B_sh[1]=0;
      dac2B_sh[0]=1;
    }
    if(choiced==20){
      dac2B_sh[1]=1;
      dac2B_sh[0]=1;
    }

    /* update single channel of DAC B on Board 2 */
    dac2B_sh[2]=dac2B_sh[3]=1;
```

```
        bit_mask=1;
        for(i=0;i<8;i++){
          dac2B_sh[11-i]=(choicev & bit_mask)/bit_mask;
          bit_mask=bit_mask << 1;
        }
      } /* end if(choiced>16 && choiced<21) */

      if(choiced>20 && choiced<25){
        /* set channel */
        if(choiced==21){
          dac2C_sh[1]=0;
          dac2C_sh[0]=0;
        }
        if(choiced==22){
          dac2C_sh[1]=1;
          dac2C_sh[0]=0;
        }
        if(choiced==23){
          dac2C_sh[1]=0;
          dac2C_sh[0]=1;
        }
        if(choiced==24){
          dac2C_sh[1]=1;
          dac2C_sh[0]=1;
        }

        /* update single channel of DAC C on Board 2 */
        dac2C_sh[2]=dac2C_sh[3]=1;

        bit_mask=1;
        for(i=0;i<8;i++){
          dac2C_sh[11-i]=(choicev & bit_mask)/bit_mask;
          bit_mask=bit_mask << 1;
        }
      } /* end if(choiced>20 && choiced<25) */

      load_dacs();

    } /* end if(choiced>12 && choiced<25) */

    if(choiced!=25){
      choiced=0;
      choicev=0;
    }

  } /* end while(choiced != 25) */
  choiced=0;

} // void SI_choice6(void)


void SI_choice7(void){

  /* Serial Interface Menu choice 7 */

  choiced=0;
  while(choiced!=7){
    clrscr();
    printf("1) Set All DACs to One Quarter Full Scale\n");
    printf("2) Set All DACs to One Half Full Scale\n");
    printf("3) Set All DACs to Three Quarters Full Scale\n");
    printf("4) Set All DACs to Full Scale\n");
```

```
printf("5) Set All DACs to Selected Value\n");
printf("6) Reset All DACs\n");
printf("7) Return to Serial Interface Menu\n");
gets(schoiced);
sscanf(schoiced,"%d",&choiced);

/* make sure menu choice is valid */
while(choiced>7 || choiced<1){
  gets(schoiced);
  sscanf(schoiced,"%d",&choiced);
}

if(choiced==1){
  choicev=63;
  for(i=0;i<12;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }
}

if(choiced==2){
  choicev=127;
  for(i=0;i<12;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }
}

if(choiced==3){
  choicev=191;
  for(i=0;i<12;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }
}

if(choiced==4){
  choicev=255;
  for(i=0;i<12;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }
}

if(choiced==5){
  clrscr();

  enter_dac_voltage();
  check_dac_voltage();

  for(i=0;i<12;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
```

```
      }
    }

    update_all_dacs();

    bit_mask=1;
    for(i=0;i<8;i++){
      dac1A_sh[11-i]=(choicev & bit_mask)/bit_mask;
      dac1B_sh[11-i]=(choicev & bit_mask)/bit_mask;
      dac1C_sh[11-i]=(choicev & bit_mask)/bit_mask;
      dac2A_sh[11-i]=(choicev & bit_mask)/bit_mask;
      dac2B_sh[11-i]=(choicev & bit_mask)/bit_mask;
      dac2C_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }

    load_dacs();

    if(choiced==6){
      for(i=0;i<12;i++){
        dac1A_sh[i]=dac1B_sh[i]=dac1C_sh[i]=0;
        dac2A_sh[i]=dac2B_sh[i]=dac2C_sh[i]=0;
        dac1_bin[i]=0;
        dac2_bin[i]=0;
        dac1_volt[i]=0.0;
        dac2_volt[i]=0.0;
      }
      reset_dacs();
    }

    if(choiced != 7) choiced=0;
  } /* end while(choiced!=7) */
  choiced=0;

} // void SI_choice7(void)


void SI_choice8(void){

  /* Serial Interface Menu choice 8 */

  while(choiceb!=10){

    clrscr();   /* clear screen */

    printf("\t\tChange Voltage for Block of DACs\n");
    printf("\t\t----------------------------------------\n\n");
    printf("1) Change Vref (DAC A) for Board 1\n");
    printf("2) Change Vth (DAC B) for Board 1\n");
    printf("3) Change Vgate (DAC C) for Board 1\n");
    printf("4) Change Vref (DAC A) for Board 2\n");
    printf("5) Change Vth (DAC B) for Board 2\n");
    printf("6) Change Vgate (DAC C) for Board 2\n");
    printf("7) Change Vref (DAC A) for Both Boards\n");
    printf("8) Change Vth (DAC B) for Both Boards\n");
    printf("9) Change Vgate (DAC C) for Both Boards\n");
    printf("10) Return to Serial Interface Menu\n");
    gets(schoiceb);
    sscanf(schoiceb,"%d",&choiceb);

    /* make sure menu choice is valid */
    while(choiceb>10 || choiceb<1){
      gets(schoice4);
```

```
  sscanf(schoice4,"%d",&choice4);
}

if(choiceb != 10){
  clrscr();

  enter_dac_voltage();
  check_dac_voltage();

  if(choiceb==1){
    for(i=0;i<4;i++){
      dac1_bin[i]=choicev;
      dac1_volt[i]=5.0*((float)choicev/256.0);
    }

    update_board1_dacA();

    bit_mask=1;
    for(i=0;i<8;i++){
      dac1A_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }
  } /* end if(choiceb==1) */

  if(choiceb==2){
    for(i=4;i<8;i++){
      dac1_bin[i]=choicev;
      dac1_volt[i]=5.0*((float)choicev/256.0);
    }

    update_board1_dacB();

    bit_mask=1;
    for(i=0;i<8;i++){
      dac1B_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }
  } /* end if(choiceb==2) */

  if(choiceb==3){
    for(i=8;i<12;i++){
      dac1_bin[i]=choicev;
      dac1_volt[i]=5.0*((float)choicev/256.0);
    }

    update_board1_dacC();

    bit_mask=1;
    for(i=0;i<8;i++){
      dac1C_sh[11-i]=(choicev & bit_mask)/bit_mask;
      bit_mask=bit_mask << 1;
    }
  } /* end if(choiceb==3) */

  if(choiceb==4){
    for(i=0;i<4;i++){
      dac2_bin[i]=choicev;
      dac2_volt[i]=5.0*((float)choicev/256.0);
    }

    update_board2_dacA();

    bit_mask=1;
```

```
  for(i=0;i<8;i++){
    dac2A_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }
} /* end if(choiceb==4) */

if(choiceb==5){
  for(i=4;i<8;i++){
    dac2_bin[i]=choicev;
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }

  update_board2_dacB();

  bit_mask=1;
  for(i=0;i<8;i++){
    dac2B_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }
} /* end if(choiceb==5) */

if(choiceb==6){
  for(i=8;i<12;i++){
    dac2_bin[i]=choicev;
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }

  update_board2_dacC();

  bit_mask=1;
  for(i=0;i<8;i++){
    dac2C_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }
} /* end if(choiceb==6) */

if(choiceb==7){
  for(i=0;i<4;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }

  update_board1_dacA();
  update_board2_dacA();

  bit_mask=1;
  for(i=0;i<8;i++){
    dac1A_sh[11-i]=(choicev & bit_mask)/bit_mask;
    dac2A_sh[11-i]=(choicev & bit_mask)/bit_mask;
    bit_mask=bit_mask << 1;
  }
} /* end if(choiceb==7) */

if(choiceb==8){
  for(i=4;i<8;i++){
    dac1_bin[i]=choicev;
    dac2_bin[i]=choicev;
    dac1_volt[i]=5.0*((float)choicev/256.0);
    dac2_volt[i]=5.0*((float)choicev/256.0);
  }
```

```
        update_board1_dacB();
        update_board2_dacB();

        bit_mask=1;
        for(i=0;i<8;i++){
          dac1B_sh[11-i]=(choicev & bit_mask)/bit_mask;
          dac2B_sh[11-i]=(choicev & bit_mask)/bit_mask;
          bit_mask=bit_mask << 1;
        }
      } /* end if(choiceb==8) */

      if(choiceb==9){
        for(i=8;i<12;i++){
          dac1_bin[i]=choicev;
          dac2_bin[i]=choicev;
          dac1_volt[i]=5.0*((float)choicev/256.0);
          dac2_volt[i]=5.0*((float)choicev/256.0);
        }

        update_board1_dacC();
        update_board2_dacC();

        bit_mask=1;
        for(i=0;i<8;i++){
          dac1C_sh[11-i]=(choicev & bit_mask)/bit_mask;
          dac2C_sh[11-i]=(choicev & bit_mask)/bit_mask;
          bit_mask=bit_mask << 1;
        }
      } /* end if(choiceb==9) */

      load_dacs();

      choicev=0;
    } /* end if(choiceb != 10) */
  } /*end while(choiceb!=10) */
  choiceb=0;

} // void SI_choice8(void)


void SI_choice9(void){

  /* Serial Interface Menu choice 9 */

  clrscr();

  printf("Board 1\n");
  printf("-------\n");
  printf("    DAC A (Vref)\tDAC B (Vth)\t\tDAC C (Vgate)\n");
  printf("    --------------\t--------------\t\t--------------\n");

  printf("A = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",
    dac1_volt[0],dac1_bin[0],dac1_volt[4],dac1_bin[4],dac1_volt[8],
    dac1_bin[8]);
  printf("B = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",
    dac1_volt[1],dac1_bin[1],dac1_volt[5],dac1_bin[5],dac1_volt[9],
    dac1_bin[8]);
  printf("C = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",
    dac1_volt[2],dac1_bin[2],dac1_volt[6],dac1_bin[6],dac1_volt[10],
    dac1_bin[10]);
  printf("D = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n\n\n",
    dac1_volt[3],dac1_bin[3],dac1_volt[7],dac1_bin[7],dac1_volt[11],
    dac1_bin[11]);
```

```
  printf("Board 2\n");
  printf("-------\n");
  printf("    DAC A (Vref)\tDAC B (Vth)\t\tDAC C (Vgate)\n");
  printf("    -------------\t-------------\t\t-------------\n");
  printf("A = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",dac2_volt[0],
    dac2_bin[0],dac2_volt[4],dac2_bin[4],dac2_volt[8],dac2_bin[8]);
  printf("B = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",dac2_volt[1],
    dac2_bin[1],dac2_volt[5],dac2_bin[5],dac2_volt[9],dac2_bin[8]);
  printf("C = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n",dac2_volt[2],
    dac2_bin[2],dac2_volt[6],dac2_bin[6],dac2_volt[10],dac2_bin[10]);
  printf("D = %1.4f V (%d)\t%1.4f V (%d)\t\t%1.4f V (%d)\n\n",dac2_volt[3],
    dac2_bin[3],dac2_volt[7],dac2_bin[7],dac2_volt[11],dac2_bin[11]);

  hit_key();
  clrscr();

} // void SI_choice9(void)


void SI_choice10(void){

  /* Serial Interface Menu choice 10 */

  clrscr();     /* clear screen */

  printf("Board 1 DAC A Register:");
  for(i=0;i<12;i++)printf("%u",dac1A_sh[i]);
  printf("\n");

  printf("Board 1 DAC B Register:");
  for(i=0;i<12;i++)printf("%u",dac1B_sh[i]);
  printf("\n");

  printf("Board 1 DAC C Register:");
  for(i=0;i<12;i++)printf("%u",dac1C_sh[i]);
  printf("\n");

  printf("Board 2 DAC A Register:");
  for(i=0;i<12;i++)printf("%u",dac2A_sh[i]);
  printf("\n");

  printf("Board 2 DAC B Register:");
  for(i=0;i<12;i++)printf("%u",dac2B_sh[i]);
  printf("\n");

  printf("Board 2 DAC C Register:");
  for(i=0;i<12;i++)printf("%u",dac2C_sh[i]);
  printf("\n\n");

  hit_key();

} // void SI_choice10(void)


void serial_mode_interface(void){

  /* Serial Mode Interface Menu */

  while(choice4 != 11){

    clrscr();   /* clear screen */
```

```
printf("\t\tSerial Interface\n");
printf("\t\t--------------------\n\n");
printf("1) Set Integrator Reset Frequency\n");
printf("2) Configure SPY MUXes\n");
printf("3) Raw\Correlator Mode Select\n");
printf("4) LVL-1 Spacing in Raw Mode\n");
printf("5) Set Calibration Enables\n");
printf("6) Set Single DAC Voltage\n");
printf("7) Set All DACs\n");
printf("8) Set Block of DACs\n");
printf("9) View DAC Voltages\n");
printf("10) View DAC Registers\n");
printf("11) RETURN to Main Menu\n");
gets(schoice4);
sscanf(schoice4,"%d",&choice4);

/* make sure menu choice is valid */
while(choice4>11 || choice4<1){
  gets(schoice4);
  sscanf(schoice4,"%d",&choice4);
}

if(choice4==1){
  SI_choice1();
  choice4=0;
} /* end if(choice4==1) */

if(choice4==2){
  SI_choice2();
  choice4=0;
} /* end if(choice4==2) */

if(choice4==3){
  SI_choice3();
  choice4=0;
} /* end if(choice4==3) */

if(choice4==4){
  SI_choice4();
  choice4=0;
} /* end if(choice4==4) */


if(choice4==5){
  SI_choice5();
  choice4=0;
} /* end if(choice4==5) */

if(choice4==6){
  SI_choice6();
  choice4=0;
} /* end if(choice4==6) */

if(choice4==7){
  SI_choice7();
  choice4=0;
} /* end if(choice4==7) */

if(choice4==8){
  SI_choice8();
  choice4=0;
} /* end if(choice4==8) */
```

```
    if(choice4==9){
      SI_choice9();
      choice4=0;
    } /* end if(choice4==9) */

    if(choice4==10){
      SI_choice10();
      choice4=0;
    } /* end if(choice4==10) */

    if(choice4 != 11){
      choice4=0;
      choicef=0;
      choicel=0;
      choicem=0;
      choicen=0;
      choiced=0;
      choicev=0;
      choiceb=0;
      choicec=0;
    }
  } /* end while(choice4 != 11) */
  choice1=0;              /* reset main menu */
  choice4=0;              /* reset Serial menu */
  choicef=0;
  choicem=0;
  choicen=0;
  choiced=0;
  choicev=0;
  choiceb=0;
  choicec=0;

} // void serial_mode_interface(void)


void xilinx_configuration_interface(void){

  /* Xilinx Configuration Menu */

  while(choice5 != 7){

    file_error=0;

    clrscr();   /* clear screen */

    printf("\t\tXilinx Configuration Interface\n");
    printf("\t\t-----------------------------\n\n");
    printf("1) Program Xilinx Parts (configuration files\n   ");
    printf("(%s and %s) must be present)\n",falm,fctrl);
    printf("2) Display Configuration Status Bits\n");
    printf("3) Change ALM Configuration File (Currently: %s) \n",falm);
    printf("4) Change Controller Configuration File (Currently: %s)\n",fctrl);
    printf("5) Change ALM Configuration File to Default (%s)\n",falm_def);
    printf("6) Change Controller Configuration File to Default (%s)\n",
      fctrl_def);
    printf("7) RETURN to Main Menu\n");
    gets(schoice5);
    sscanf(schoice5,"%d",&choice5);

    /* make sure menu choice is valid */
    while(choice5>7 || choice5<1){
      gets(schoice5);
      sscanf(schoice5,"%d",&choice5);
```

```
}

if(choice5==1){

  /* open file for controller */
  file_error=0;     /* reinitialize file_error flag */
  if((fpctrl=fopen(fctrl,"r"))==NULL){
    clrscr();
    printf("Error: could not open %s\n",fctrl);
    hit_key();
    file_error=1;
  }

  if((fpalm=fopen(falm,"r"))==NULL){
    clrscr();
    printf("Error: could not open %s\n",falm);
    hit_key();
    file_error=1;
  }


  if(file_error != 1){
    /* clear Xilinx configuration memory */
    clear_xilinx();

    /* /PROG must be asserted from now on to maintain configuration */

    /* program controller */
    program_controller();

    /* program Address List Manager */
    program_alm();

    /* check FPGA configuration status */
    check_fpga_config();

    /* re-initialize system */
    sys_init();

  } /* end if(file_error != 1) */
  /* End programming section */

  /* close configuration files */
  fclose(fpalm);
  fclose(fpctrl);
} /* end if(choice5==1) */

if(choice5==2){
  read_xilinx_flags();
  clrscr();
  printf("pc_init1 = %d\n",pc_init1);
  printf("pc_init2 = %d\n",pc_init2);
  printf("done1    = %d\n",done1);
  printf("done2    = %d\n",done2);
  hit_key();
} /* end if(choice5=2) */

if(choice5==3){
  fclose(fpalm);
  clrscr();
  printf("The current ALM file is: %s\n\n",falm);
  printf("Please enter name for new ALM file:\n");
  scanf("%s",falm);
```

```
      open_new_alm();
    }

    if(choice5==4){
      open_new_ctrl();
    }

    if(choice5==5){
      fclose(fpalm);
      for(i=0;i<15;i++){
        falm[i]=falm_def[i];
      }
      if((fpalm=fopen(falm,"r"))==NULL){
        clrscr();
        printf("Error: could not open %s\n",falm);
        hit_key();
      }
    }

    if(choice5==6){
      fclose(fpctrl);
      for(i=0;i<15;i++){
        fctrl[i]=fctrl_def[i];
      }
      if((fpctrl=fopen(fctrl,"r"))==NULL){
        clrscr();
        printf("Error: could not open %s\n",fctrl);
        hit_key();
      }
    }

  if(choice5 != 7) choice5=0;
  } /* end while(choice5 != 7) */
  choice1=0;              /* reset main menu */
  choice5=0;              /* reset Xilinx menu */

} // void xilinx_configuration_interface(void)
```

Appendix B

Publications

During the course of development for the MVD beam test system, a paper, entitled "A Flexible Analog Memory Address List Manager," was published in the *IEEE Transactions on Nuclear Science* (Vol. 4, No. 4, August 1995) and provides, in condensed form, an analysis of the early work on the address list manager for PHENIX. In the paper, a sixty-four address ALM, which is the predecessor for work which is presented in this thesis, is outlined. Also presented is a paper, published in *First Workshop on Electronics for LHC Experiments* in September 1995 and entitled "Multiplicity Vertex Detector Electronics for the PHENIX Detector," which specifies the front-end electronics and includes details and a quantitative analysis of the critical points in the designs of the ASICs.

A Flexible Analog Memory Address List Manager for PHENIX[1]

M. N. Ericson, M. S. Musrock, C. L. Britton, Jr., J. W. Walker, A. L. Wintenberg, and G. R. Young
Oak Ridge National Laboratory, Oak Ridge, TN 37831

M. D. Allen
The University of Tennessee, Knoxville, TN 37996

Abstract

A programmable analog memory address list manager has been developed for use with all analog memory-based detector subsystems of PHENIX. The unit provides simultaneous read/write control, cell write-over protection for both a Level-1 trigger decision delay and digitization latency, and re-ordering of AMU addresses following conversion, at a beam crossing rate of 105 ns. Addresses are handled such that up to 5 Level-1 (LVL-1) events can be maintained in the AMU without write-over. Data tagging is implemented for handling overlapping and shared beam-event data packets. Full usage in all PHENIX analog memory-based detector sub-systems is accomplished by the use of detector-specific programmable parameters -- the number of data samples per valid LVL-1 trigger and the sample spacing. Architectural candidates for the system are discussed with emphasis on implementation implications. Details of the design are presented including application specifics, timing information, and test results from a full implementation using field programmable gate arrays (FPGAs).

I. INTRODUCTION

Analog memory units (AMUs) play an important role in many physics experiments as data storage elements [1,2,3]. Use of these elements allows proper handling of delays associated with making trigger decisions and digitization latency [3,4,5]. Published methods used to control analog memory are bit-enabling schemes [1,2] and direct addressing [4]. Bit-enabling methods use individual enables to control the write/read operation of a particular AMU cell or column of cells. In direct addressing the AMU cells are memory mapped, and the addresses are manipulated for write/read operations. Bit-enabling methods can produce unacceptable delays due to the requirement of looking ahead for unprotected cells, which is particularly a problem if FPGAs are used for implementation. Direct addressing methods require the storage and manipulation of an address list (address list management) which can require significant amounts of memory. Additionally, the memory has to be partitioned into several separately controlled blocks. However, this method lends itself nicely to the AMU-based detector subsystems of PHENIX where multiple event handling (up to 5) with a programmable number of samples and sample spacing are required.

II. PHENIX FRONT-END MODULES

This paper presents an enhanced address list manager (ALM) designed to provide a solution for all AMU-based detector subsystems of PHENIX -- namely Multiplicity Vertex Detector (MVD), EM Calorimetry (Lead Glass and Lead Scintillator), Muon Identifier (MuID), and Muon Tracker (MuTR). Fig. 1 shows a generic block diagram of the PHENIX front-end module (FEM) for these subsystem types. Each FEM contains multiple analog signal processing

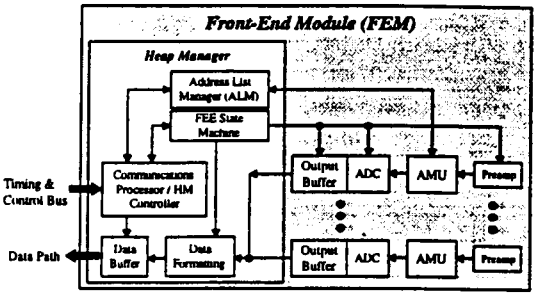


Fig. 1 Front-End Module Block Diagram

channels and sufficient control circuitry to operate as an independent system module. Timing signals and control commands are all the FEM requires to acquire data. The front-end electronics are composed of a preamplifier, AMU, and buffered ADC, that are mapped one to each detector element. All subsystems will use a voltage-write/voltage-read 64-cell deep AMU, and an 8 to 12-bit Wilkinson-type ADC per detector channel, both fabricated in 1.2μm CMOS [6]. The heap manager provides all the control for front-end sequencing, data collection and formatting, and communications functions associated with an FEM. Up to 256 detector channels will be controlled by a single heap manager. The number of associated detector channels is sub-system dependent and is related to detector pitch and physical partitioning.

[1]Research sponsored by the U.S. Department of Energy and performed at Oak Ridge National Laboratory, managed by Lockheed Martin Energy Systems, Inc. for the U.S. Department of Energy under Contract No. DE-AC05-84OR21400.

The address list manager has several programmable parameters that ensure full applicability to these 5 detector subsystems: number of samples per valid trigger, sample spacing specified as the number of beam clocks between adjacent samples, and LVL-1 trigger delay. The LVL-1 trigger delay value will be identical for all detector subsystems, but must be programmable since this value will not be exactly determined until the detector is operational. Table 1 gives the sampling requirements for the associated PHENIX detector subsystems.

Table 1
PHENIX Detector Subsystem Sampling Requirements

Detector Subsystem	*Samples per LVL-1 Accept*	*Sample Spacing in Beam Ticks*
Multiplicity Vertex Detector	2	1
EMCal		
Lead Glass	2	2-5
Lead Scintillator	2	2-5
Muon Identifier	2	2
Muon Tracker	3-4	Variable?

Besides programmability, other special functions are implemented in the ALM, including maintaining address ordering and data tagging for multiple-event separation. Each AMU must be able to store up to 5 LVL-1 qualified events and protect these events while continuously writing to unprotected AMU cells until the qualified events are digitized. By maintaining proper address control, writing to the AMU is void of dead-time. After digitization, the address is placed back into the address list in appropriate order. Address reordering helps reduce the injected noise into the AMU during cell read/write operations. When considering multiple samples and possible multiple LVL-1 events, event data packets can become highly interspersed and will require separation. Data tagging allows more straightforward separation and reconstruction of the individual beam events from overlapping and/or shared data, particularly when this function has to be performed in the FEM. The AMU controller presented in this paper performs these functions at the PHENIX beam clock rate of 9.52 MHz.

III. ALM Circuit Topology

Fig. 2 shows a high-level block diagram for the address list manager. The method employed involves the shuffling of 6-bit AMU addresses. Though this requires memory space for address holding, it provides some distinct advantages when the special functions associated with multiple event buffering, AMU cell address reporting, and data tagging are considered. The basic ALM is a recirculating loop composed of three memory units. The LVL-1 Delay FIFO is used to implement the LVL-1 trigger delay and can be programmed to be as large as 48 beam-clock ticks. After the delay period has passed, the address is placed into the ADC Conversion FIFO to await digitization if qualified, or passed to the Available Address FIFO to be re-used. The AMU addresses in both the LVL-1 Delay FIFO and the ADC Conversion FIFO are protected from being overwritten. As digitization is completed, the associated address is released and placed in proper order in the Available Address FIFO for re-use. An additional FIFO (Sorting Memory) may be required depending on the time required to sort the address and the digitization time.

At initialization following a reset or power-up, a counter is used to fill the loop with addresses 0 through 63, which pass through a mux (MUX1) into the LVL-1 delay FIFO. When the counter outputs address 63, MUX1 is switched into 'feedback mode,' where the addresses from the Available Address FIFO are fed into the Level-1 Delay FIFO. The input address to the LVL-1 Delay FIFO is buffered and used as the current AMU write address. A new address is always available each beam clock to prevent the overwriting of valid data.

The LVL-1 Delay FIFO acts as a programmable depth shift register. The LVL-1 delay, specified in beam clock periods, is stored in the ALM at setup. At initialization, an address is written into the FIFO each beam clock cycle. The read from the FIFO is suppressed until the number of beam clocks specified by the programmed LVL-1 delay bits has passed. This function is performed using a beam clock counter and digital comparator. Once initialized, a write and a read operation are performed each beam clock until another reset is issued.

The routing of the AMU addresses from the LVL-1 Delay FIFO is determined by the LVL-1 accept bit. If a valid LVL-1 is generated, the address is written into the ADC Conversion FIFO. This operation removes the addresses from the available address list, thus preventing them from being overwritten. Other addresses associated with the same event that occur on following beam clocks are also written into this FIFO as each pops out of the LVL-1 Delay FIFO on subsequent beam clocks. Addresses that do not receive a valid LVL-1 are written into the Available Address FIFO. Addresses that have been used for digitization are also placed in this memory following a sorting procedure that slips the address back into the appropriate location. After initialization, the Available Address FIFO must output an address each beam clock to provide the AMU write address.

Proper sizing of FIFOs and control of the LVL-1 accept are required for correct operation of the system. LVL-1 accepts are limited to 5 total events with new LVL-1 accepts occurring no sooner than 40 μs. As a result, the ALM must be able to store up to 5 events in the ADC Conversion FIFO or as many as 20 addresses, depending on the subsystem. With a maximum 40-beam-clock LVL-1 delay and 5 events in the ADC Conversion FIFO, 4 addresses are left in the Available Address FIFO. Conversely, with a minimum 32-

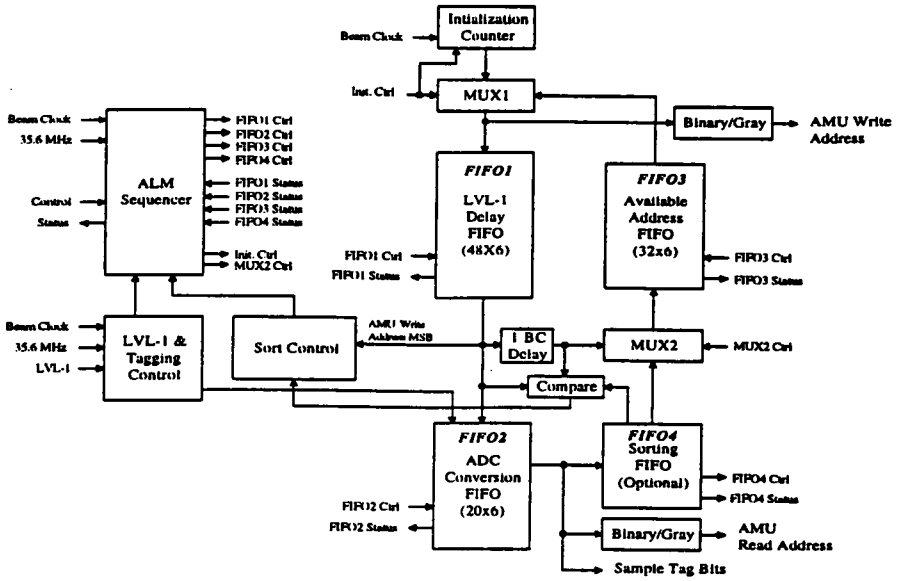


Fig. 2 Address List Manager Block Diagram

beam-clock delay and no events to be digitized, 32 addresses will be located in the Available Address FIFO. If a LVL-1 delay of > 40 beam clocks is needed, LVL-1 accepts will have to be further limited to ensure enough addresses are available to provide an AMU write address each beam clock.

Tests performed on CMOS AMU units indicate some noise benefit may be realized by using gray-coded AMU addresses. The ALM manages addresses in a binary format, and the conversion is made on both the AMU read and write addresses before being distributed to the AMU unit. Selection of the address output format (gray/binary) is a programmable function which will allow further characterization of the AMU and aid in ALM testing where binary addresses are more intuitive.

A. FPGA-Based FIFOs

As demonstrated in Fig. 2, FIFO memories play an important role in the address list manager. Due to the space limitations of all PHENIX detector subsystems, a highly integrated ALM was desired. Reconfiguration of the ALM functions make FPGA implementation more desirable than competing technologies including ASICs. SRAM-based FPGA families provide the ability to configure SRAM for use as FIFOs, SRAM blocks, and shift registers.

The base FIFO cell used in this design is shown in Fig. 3 [7]. A linear feedback shift register (LFSR) generates the address pointers for the SRAM, and a compare circuit keeps track of the relation between read and write addresses. To increase speed, the arbitration logic is configured to force a write to the SRAM block each cycle, which effectively removes one location from usable address space. The arbitration logic of the FIFO consists of two main signals, the PUSH (or write) and the POP (or read). These may be asserted simultaneously, although the PUSH has priority and will be executed first. The INPUT_RDY and OUTPUT_RDY signals may be examined to determine when data can be read from or written to the FIFO.

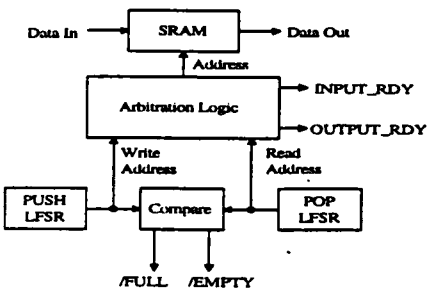


Fig. 3 Base FIFO Cell

B. Address Ordering

Re-ordering of the AMU addresses in the ALM is important to reduce the noise injected into the AMU cells from the transitioning of digital lines [8]. Referring to Fig. 2, a digital comparator is used to determine the proper placement of a binary address in the loop. Inclusion of a 1 beam-clock delay allows the comparator to look at two adjacent loop addresses and determine if the sort address fits between the two addresses. A simple function is performed by the

comparator: $addr_{LVL\text{-}1delay} > addr_{ADC} > addr_{1hodelay}$. This algorithm is valid unless the address to be sorted is at the boundary of the addresses -- that is, it is 0 or 63, or nearer to the boundary than any other address within the available address loop. This special case is handled by monitoring the MSB of the LVL-1 Delay FIFO and a status bit to determine if there is currently an AMU address left to be sorted. If so, a compare is done to see if $addr_{ADC}$ > both $addr_{LVL\text{-}1delay}$ and $addr_{1hodelay}$ or $addr_{ADC}$ < both $addr_{LVL\text{-}1delay}$ and $addr_{1hodelay}$. A true condition results in a read request to the ADC conversion FIFO and an additional write request to the available address FIFO, similar to the previous case. The control logic is identical for both cases once the special cases have been handled. The most limiting beam-clock cycle is one in which one read request and two write requests are made to the available Address FIFO. Under either sorting case, this sorting procedure takes only one complete lap of the addresses.

C. Data Grouping and Tagging

LVL-1 control and data tagging are generated using the circuit shown in Fig. 4. Five serial shift registers are loaded in a rotating fashion, one for each valid LVL-1 accept, with a sampling template that identifies the number of samples and

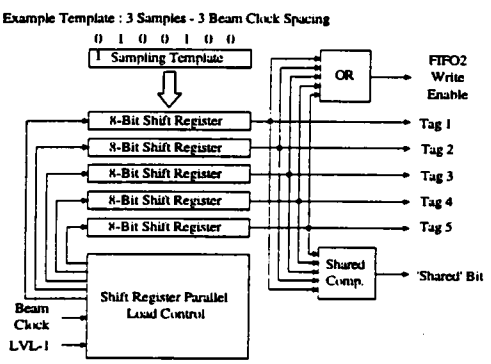


Fig. 4 LVL-1 Control and Sample Tagging

sample spacing. This provides the capability to handle as many as five overlapping or shared events. An example template given in Fig. 4 shows three samples per LVL-1 event with a spacing of three beam clocks (01001001). Every beam clock the shift registers are shifted to the right. When a '1' is output, the current AMU write address is associated with a valid event and is placed in the ADC Conversion FIFO. These output bits can also be placed in the ADC Conversion FIFO as event identifiers to help in the separation/reconstruction of data packets. This is particularly important since the data from multiple packets can be interlaced. The FIFO2 write enable is generated by performing an 'OR' on the tag bits. If multiple valid tags are generated on a single beam clock, a 'shared' bit is set and stored with the tag information. This function is significant because the contents of a single AMU cell cannot be digitized twice with sufficient accuracy. The use of this architecture utilizing sampling templates allows the data samples to be irregularly spaced.

IV. Implementation and Testing

To meet the requirements of PHENIX, all ALM and data manipulation functions must be reprogrammable in-circuit to accommodate future functional upgrades. Adhering to this requirement meant excluding ASICs from possible implementation technologies leaving programmable logic devices as the primary choice. The need to maximize integration further reduced the options to SRAM-based FPGAs that allow the incorporation of FIFO structures. Of the options available we chose the Xilinx XC4000 family.

Implementation of the ALM began with testing of 16-bit and 32-bit FIFOs. All FIFOs were incremented in sizes of 32 bits, the maximum allowed per XC4000 Xilinx configuration logic block (CLB) with individual read/write controls. The necessity of a larger 64-bit FIFO for the LVL-1 Delay FIFO requires additional logic for read/write control muxing and results in a slower speed of operation. Routed separately, individual 32-bit FIFOs easily operate at well over 40 MHz without using special routing methods.

Implementation was accomplished using schematic capture and a PC-based Viewlogic ProSeries system with XACT timing editor. The first system design using the default XACT timing specifications resulted in a maximum clock speed of 20 MHz. The design was slowed by the near-full capacity of the XC4005, which utilized 98% packed CLBs. Later designs included reassignment of the global timing and critical net timing specifications. Worst-case flip-flop to flip-flop delays (FFS_FFS) were on the order of 80 ns. Such a large delay results in little improvement in utilizing high speed parts since the 1-2 ns CLB delay improvement available becomes small in comparison to the routing delays. The current design was customized for the mid-speed XC4005-5 and reached clock speeds of 45 MHz. The identical LCA configuration was loaded into XC4005-4 and XC4005-6 parts resulting in clock speeds of approximately 40 MHz. Further synthesis effort in assigning critical nodes and time specifications should result in operating speeds on the order of 45 MHz for these devices also.

A timing diagram of the ALM is shown in Fig. 5. Besides general operation, this diagram demonstrates both normal operation and the special operations associated with address reordering. Here addresses 40, 42, and 63 have been removed from the list and are stored in the ADC Conversion FIFO. The lvl1_out and amu_wr_addr buses show absence of these addresses (40, 42, 63) that are awaiting proper re-

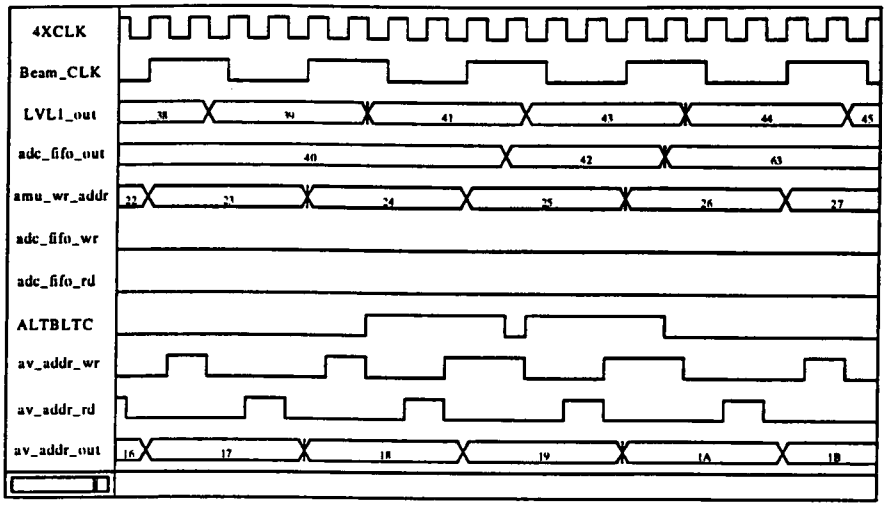


Figure 5. ALM Timing Diagram Demonstrating General Operation and Address Re-Ordering

sorting conditions. When a sorting condition is met, the adc_fifo_out address falls between the current lvl1_out and the delayed lvl1_out, and an additional write to the Available Address FIFO occurs. A read of the ADC Conversion FIFO also occurs if the FIFO is not empty. The Available Address FIFO is always read, guaranteeing a valid AMU write address each beam clock. Notice that two worst-case conditions occur on adjacent beam-clock cycles -- two writes and a read to the Available Address FIFO within one beam-clock cycle.

V. CONCLUSIONS

A programmable analog memory address list manager has been developed for generic application to all analog memory-based PHENIX detector subsystems. The unit allows programming of the LVL-1 delay latency, number of data samples per valid LVL-1 trigger, and sample spacing. Sampling parameters are programmed as a template which allows irregularly spaced samples. Cell write-over protection, address re-ordering and sample tagging are also implemented. The design was synthesized in an FPGA and operated at 45 Mhz (11.25 Mhz beam clock). Power consumption was measured as 575 mW at PHENIX beam clock rates of 9.52 MHz.

VI. REFERENCES

[1] F. Anghinolfi, et al., "Characteristics of a 'Harp' Signal Processor with Analog Memory Operated with Segmented Silicon Detectors", *IEEE Trans. Nucl. Sci.*, Vol. 41, No. 4, August 1994, pp. 1130-1134.

[2] F. Anghinolfi, et al., "DYN1: a 66 MHz front end analog memory chip with first level trigger capture for use in future high luminosity particle physics experiments", *Nuclear Instruments & Methods in Physics Research Section A*, Vol. 344, 1994, pp. 173-179.

[3] A. L. Wintenberg, et al., "Monolithic Circuits for the WA98 Lead Glass Calorimeter", *Conference Record of the 1994 Nuclear Science Symposium and Medical Imaging Conference*, November, 1994.

[4] S. Kleinfelder, M. Levi, and O. Milgrome, "Toward a 62.5 Mhz Analog Virtual Pipeline Integrated Data Acquisition System, *Nucl. Phys.* B23A (1991) pp. 382-388.

[5] A. Gara, J. A. Parsons, and W. Sippach, "Readout Electronics for the GEM Calorimeter", *Conference Record of the Second International Conference on Electronics for Future Colliders*, May, 1992.

[6] O. B. Milgrome, S. A. Kleinfelder, and M. E. Levi, "A 12-Bit Analog to Digital Converter for VLSI Applications in Nuclear Science", *IEEE Trans. Nucl. Sci.*, Vol. 39, No. 4, 1992.

[7] Xilinx Application Note XAPP 044.000, "High-Performance RAM-Based FIFO", The Programmable Logic Data Book, 1994.

[8] C. L. Britton, et al., "An Analog Random Access Memory in the AVLSI-RA Process for an Interpolating Pad Chamber", to be published in *IEEE Trans. Nucl. Sci.*, 1995.

MULTIPLICITY-VERTEX DETECTOR ELECTRONICS FOR THE PHENIX DETECTOR

C. L. Britton, Jr., W. L. Bryan, M. S. Emery, M. N. Ericson, M. S. Musrock, M. L. Simpson, J. W. Walker, A. L. Wintenberg, F. Plasil, G. R. Young[1]
Oak Ridge National Laboratory

M. D. Allen, L. G. Clonts, E. J. Kennedy, R. S. Smith
The University of Tennessee

J. Boissevain, B. V. Jacak, J. Kapustinsky, J. Simon-Gillo, J. P. Sullivan, H. van Hecke, N. Xu
Los Alamos National Laboratory

ABSTRACT

This paper presents the electronics work performed to date for the Multiplicity-Vertex Detector (MVD) for the PHENIX collaboration at RHIC. The detector consists of approximately 34,000 channels of both silicon strips and silicon pads. The per-channel signal processing chain consists of a preamplifier-gain stage, a current-mode summed multiplicity discriminator, a 64-deep analog memory (simultaneous read-write), an analog correlator, and a 10-bit 5μs ADC. The system controller or Heap Manager, supplies all timing control, data buffering, and data formatting for a single 256-channel multi-chip module (MCM). Each chip set is partitioned into 32-channel sets. Prototype performance for the various blocks will be presented as well as the ionizing radiation damage performance of the 1.2μ nwell CMOS process used for fabrication.

1. INTRODUCTION

The Multiplicity-Vertex Detector (MVD) of the PHENIX detector at RHIC is one of the most challenging of the PHENIX detector subsystems. The requirements of low-power consumption, small physical area, a channel count of approximately 34,000, and flexible data handling have resulted in a challenging set of problems. The requirements of the system are to have signal-to-noise not less than 20:1 for a 1MIP signal which results in noise less than 2500 electrons rms. The added requirement of being able to discriminate on a 0.5MIP event for the per-channel multiplicity discriminator is much more difficult.

This paper presents measurements of the prototype component systems of the MVD detector electronics. The circuits include the preamplifier-discriminator, analog memory unit (AMU), analog-digital converter (ADC), and heap manager.

[1] Research sponsored by the U.S. Department of Energy and performed at Oak Ridge National Laboratory, managed by Martin Marietta Energy Systems, Inc. for the U.S. department of Energy under Contract No. DE-AC05-84OR21400.

2. ELECTRONICS ARCHITECTURE

A block diagram of the electronics is shown in Fig. 1. This includes the preamplifier, discriminator, current-sum output, analog memory-correlator, and ADC. The overall controller, or Heap Manger, is not shown but will also be discussed.

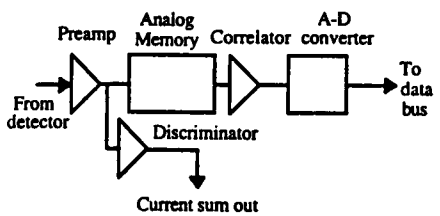


Figure 1. System block diagram.

The electronics will be mounted on multi-chip modules (MCM). Each MCM will be connected to 256 detectors and will contain 8 preamplifier-discriminator chips and 8 analog memory-ADC chips. Each of these chips will contain 32 channels of their respective functions. In addition, the MCM will contain the Heap Manager chips and associated control logic. The present prototype chips are 8 channels of respective functions.

A. Preamplifier

The preamplifier, versions of which are presently being used by not only PHENIX, but PHOBOS [1] (also at RHIC), and the Naval Research Laboratories (NRL) for Germanium spectroscopy[2], is presented in Fig. 2. It utilizes a PMOS cascode amplification stage for low 1/f noise. The input device operates in weak inversion with a drain current of 100μA. The drain current curve is shown in Fig. 3. The PHENIX and PHOBOS versions employ wideband gain stages after the preamplifier because the output will be processed by a correlated sampler. The NRL version employs a semi-Gaussian

shaper with 7μs peaking time. A summary of the performance of the three versions is presented in Table 1. The resultant power dissipation of the PHENIX preamplifier with the gain stages is approximately 1.2mW. The circuit is fabricated in 1.2μ n-well CMOS and has a pitch of 85μ.

Parameter	PHENIX (224ns double correlated)	PHOBOS (Wideband)	NRL (7μs peaking)
0pF noise	660e	750e	205e
Slope	69e/pF	30e/pF	8e/pF
Gain	44mV/fC	4.2mV/fC	11mV/fC
Drain current	100μA	200μA	100μA

Table 1. Performance of various preamplifier versions.

B. Discriminator

The multiplicity discriminator block diagram is presented in Fig. 2. Each time an input step greater than the threshold setting is detected, a fixed-value current source is switched to a summing node. The current-mode outputs of 256 discriminators are summed in this manner to produce a multiplicity output whose amplitude is proportional to the number of fired discriminators.

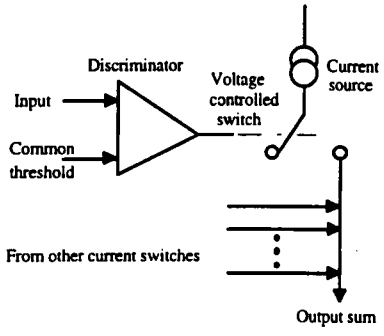


Figure 2. Block diagram of the discriminator

The total power consumption of the discriminator is approximately 500μW/channel.

C. Analog memory-correlator

The analog memory unit (AMU) is a 64-cell deep, voltage-write-voltage-read, deadtime-less topology with a power dissipation of approximately 1mW/channel. The memory is followed by an analog correlator that performs the double-correlated sampling. Both direct memory output and correlated output are available to the subsequent analog-digital converter (ADC). The memory reading and writing is addressed externally from the Heap Manager. The read and write logic decoders are completely independent to allow simultaneous read-write (deadtime-less) operation. A block diagram of the memory is presented in Fig. 3.

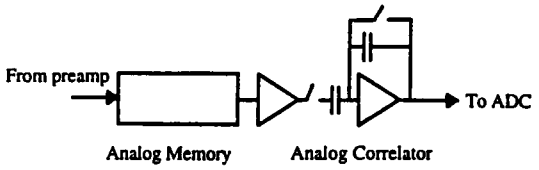


Figure 3. Memory block diagram

D. Analog-digital converter

The readout architecture of the subsystems within the PHENIX detector have been intentionally designed to resemble each other a great deal. This has been done for two reasons. The first is for maintainability. Using the same parts in many subsystems reduces the needed stocks of spares and simplifies troubleshooting. The second is cost. We can take advantages of the economies of scale by fabricating a large quantity of one part and use it in many subsystems within the detector. This philosophy is best exhibited in the AMU/ADC chip. We are designing a universal AMU/ADC chip that will be used in several subsystems. The architecture of the chip is 32 channels of 64-deep AMU and 32 channels of 12-bit ADC. The ADC will be switchable for 10-, 11-, or 12-bit operation. For the MVD, the ADC will be set for 10-bit operation for dynamic range requirements.

The analog-digital converter (ADC), presented in Fig. 4, is a 10-bit Wilkinson type[3] with a maximum dynamic range of 4.5 Volts. This prototype chip has eight channels.

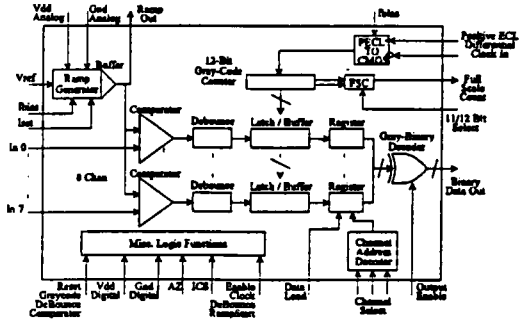


Figure 4. Block diagram of the ADC chip.

The ADC uses 3.75μW/MHz/channel in the digital circuitry and 6.5mW total in the ramp generator. This results in a power dissipation of approximately 0.6mW/channel for a 32-channel chip or 1.19mW/channel for the 8-channel prototype at 100MHz clock frequency (5μs conversion time for 10-bit conversions

E. Heap Manager

The Heap Manager controls all on-board functions including AMU address list management and control (for asynchronous read and write operations), correlator timing, Level-1 buffering associated with trigger latency, ADC precharge, acquisition and conversion control, data handling including data collection, formatting, and transmission, and command interpretation, execution, and timing. This function has been implemented using commercially available FPGAs allowing in-circuit programmability for future functional upgrades. A multi-function serial interface is used for FPGA programming, calibration setup, diagnostics, and monitoring.

A generic heap manager has been developed for use in all PHENIX detector subsystems using analog memory for front-end signal buffering during the Level-1 trigger decision. However, due to the reduced size and power requirements for the MVD, this functional block has been partitioned such that the minimum required electronics are physically located near the detectors. The remaining circuitry resides at the ends of the detector subassembly as interface cards to the data collection modules (DCMs) . Figure 5 is a block diagram of the MVD heap manager showing the basic functional blocks and system partitioning.

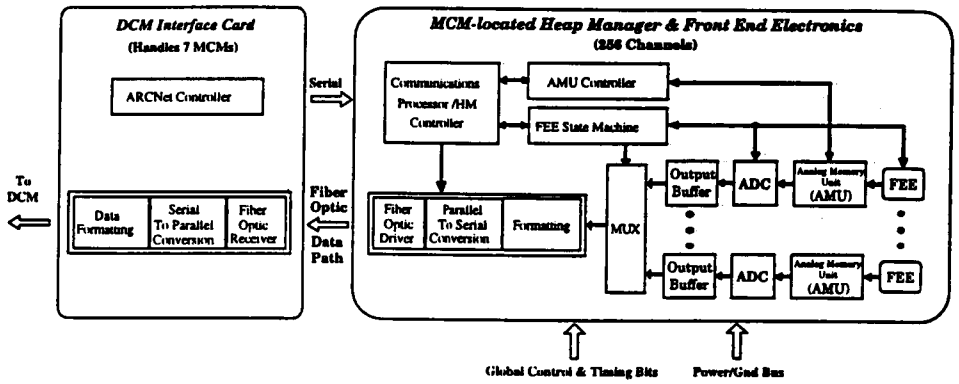


Figure 5. Block diagram of the Heap Manager.

III. Radiation Effects

Some significant (>20krad) radiation exposure of MVD is expected due to the proximity of the detector to the interaction region. Because of the desire to utilize a standard non-radiation hardened process, measurements of radiation effects upon the target 1.2μ CMOS process were made. Versions of the preamplifier, analog memory and ADC circuits have been irradiated to various doses of ionizing radiation. The process appears to be sufficient for doses up to greater than 75krad for most functions. The major problem is in the leakage current of the input protection for the preamplifier. Devices used for protection networks begin to exhibit increased leakage current due to radiation-induced decrease in the nmos threshold voltage. We are continuing to investigate improved input protection.

IV. Conclusions

The circuits for the PHENIX MVD has been presented. The readout consists of preamplifier, multiplicity discriminator, analog memory, ADC, and Heap Manager controller. Prototypes of the major functional blocks have been fabricated and have been tested. The radiation tolerance of the CMOS process is presently under test.

VI. Acknowledgments

The authors would like to thank Norma Hensley for her help in preparing this manuscript.

VII. References

[1] B. Wadsworth, MIT Laboratory for Nuclear Sciences, private communication.

[2] R. A. Kroeger, W. N. Johnson, R. L. Kinzer, J. D. Kurfess, S. Inderhees, M. D. Allen, G. T. Alley, C. L. Britton, L. C. Clonts, M. N. Ericson, and M. L. Simpson, "Charge Sensitive Preamplifier and Pulse Shaper Using CMOS Process for Germanium Spectroscopy", IEEE Trans. Nucl. Sci., Vol. 42, No. 4, Aug. 1995 (921-924).

[3] A. L. Wintenberg, T. C. Awes, C. L. Britton, Jr., M. S. Emery, M. N. Ericson, F. Plasil, M. L. Simpson, J. W. Walker, and G. R. Young, "Monolithic Circuits for the WA98 Lead Glass Calorimeter". Presented at the 1994 Nuclear Science Symposium, Norfolk, VA, Nov. 1994

Vita

Mark Douglas Allen was born on December 25, 1969, in Johnson City, Tennessee. In 1976, Mark moved, with his parents, to Maryville, Tennessee, where he currently resides. After receiving his diploma from Maryville High School in 1988, Mark attended The University of Tennessee, Knoxville, where he earned a Bachelor of Science Degree in Electrical Engineering in 1993 and immediately began pursuing a Master of Science degree in Electrical Engineering. Besides working as a teaching assistant, Mark, in 1994, began work with the Monolithic Systems Development Group at Oak Ridge National Laboratory. Mark is a member of Tau Beta Pi, Eta Kappa Nu, and IEEE; and his interests lie in the areas of VLSI, digital logic design, and digital communications. Outside of school, Mark likes to read (Faulkner and Dostoevsky, in particular); and he engages in creative writing whenever possible. He is currently in the care of his parents.