DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/91D-11

Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator

THESIS

Paul J. Taylor, Jr.
Captain, USAF

AFIT/GCE/ENG/91D-11

AFIT/GCE/ENG/91D-11

Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Paul J. Taylor, Jr., B.S.E.
Captain, USAF

December, 1991

Acknowledgments

I would like to thank my advisor, Dr. (Major) William Hobart, for providing the guidance and inspiration that were indispensible during this effort. I would also like to thank my committee members, Dr. Thomas Hartrum and Captains Mark Mehalic and Keith Jones for their patience and invaluable feedback.

I am deeply indebted to Messrs. Richard Norris and Russell Milliron. Their tolerance of my never-ending requests for better computer support were always cheerfully met.

The support and encouragement provided by fellow students was critical, I would like to thank all of my classmates. In particular, special thanks must be extended to Kenn Scribner and Rod Taylor. Without the hours of listening and explaining these two so generously provided, I could not have made it.

I would like to thank, and commend my son, Michael, for surviving the last eighteen months. The teen years are particularly trying, and he's gone it pretty much alone – I'm back.

Most of all I wish to thank my wife, Jo Anne. Her ordeal was by far greater than mine, for she had little control over my seemingly endless struggles. It was through her love and understanding that "we" survived. Her presence and support, not only at AFIT, but in all I do is paramount. Thank you Jo, I love you.

Paul J. Taylor, Jr.

Table of Contents

List of Figures

List of Tables

AFIT/GCE/ENG/91D-11

Abstract

An analysis of a general Discrete Event Simulation (DES), executing on the distributed architecture of an eight node Intel *i*PSC/2 hypercube, was performed. The most time consuming portions of the general DES algorithm were determined to be the functions associated with message passing of required simulation data between processing nodes of the hypercube architecture. A behavioral description, using the IEEE standard VHSIC Hardware Description and Design Language (VHDL), for a general DES hardware accelerator is presented. The behavioral description specifies the operational requirements for a DES coprocessor to augment the hypercube's execution of DES simulations. The DES coprocessor design implements the functions necessary to perform distributed discrete event simulations using a conservative time synchronization protocol.

Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator

I. Introduction

1.1 Background

Computer simulations are used in a broad range of diverse applications such as engineering, medicine, social sciences, and the military. Traditionally, simulations were designed for and executed on sequential processors. However, dramatic increases in the size and complexity of simulations over the past 20 years have resulted in simulation models "whose computational requirements cannot be reasonably satisfied with even the fastest sequential processors" (28:8).

The design of electronic hardware is one area where the increased complexity of simulation models is very evident. The rapid growth in component to chip densities requires simulation of ever larger circuits. Since 1960 the circuit to chip ratio has nearly doubled every year, resulting in densities greater than 500,000 transistors per chip (12:449). Although this growth rate has slowed to a doubling about every two years, the required logic simulation has become a major limitation in the overall design process.

The Air Force has a large investment in electronic hardware, and the development costs continue to increase as the hardware becomes more complex. The Air Force's increased reliance on electronic hardware is contributed significantly to the Department of Defense's Very High Speed Integrated Circuit (VHSIC) program. A primary objective of the VHSIC program is to develop and promote the use of high-density integrated circuits in military systems.

VHSIC technology is heavily dependent on simulation for the design and verification of these complex electronic components. Logic verification and fault analysis are essential in the design of VHSIC chips and must be performed extensively before chip fabrication. This complex testing, done through simulation, often consumes months of computer time and has become a bottleneck in the logic design process (12:449).

The VHSIC Hardware Description Language (VHDL) program was started in 1983 to standardize the tools needed to design, test, and document large-scale digital electronics more efficiently. Initial implementations of VHDL were developed by Intermetrics, Inc. under a DoD contract in 1983. Evolution and improvements in the language led to the IEEE Standard VHDL Language Reference Manual in 1987. VHDL has become important enough in recent years that the Department of Defense Advanced Research Projects Agency (DARPA) has sponsored the QUEST project. One objective of the QUEST project is simulation acceleration, specifically a thousand-fold speedup in VHDL simulations of VHSIC designs is desired.

1.2 Problem

The limitations of traditional sequential processors have increased research in the area of applying parallel computer architectures and multiprocessor technology to meet the computational requirements of large simulations. Theoretically, if a sequential simulation is logically partitioned into separate processes, placed on separate processors and run in parallel, the amount of speedup attainable should be equal to the number of processors used.

The theoretical speedup possible through parallel, or as it is more commonly known, distributed simulation has yet to be realized. Several obstacles inherent to distributed processing must be minimized to approach the theoretical speedup. Among these obstacles are: the communications overhead associated with the necessary exchange of information between logical processes; the load imbalance related to the static allocation of logical processes to processors; and the synchronization delay necessary to ensure event-driven simulations do not process events out of order.

This thesis investigates possible enhancements to the discrete-event distributed simulation process that can be realized through a hardware implementation. The purpose of this research is to specify the detailed requirements of such an implementation.

1.3 Summary of Current Knowledge

Simulation models are classified by Pritsker as either discrete, continuous, or combined. The basis for this classification is how the dependent variables of the simulation model change with respect to time. In discrete simulation the dependent variables change at specified points in simulated time known as event times and generally do not change values between event times. Discrete simulation is further classified by the relationship between events, activities, and processes. Continuous simulation results when the dependent variables of the simulation model change continuously over simulated time. Combined simulation occurs when dependent variables change discretely, continuously, or a combination of both (26:63-64).

A time-based classification for simulation is also proposed by Neelamkavil. In this classification, time can be advanced in two ways. The first is a synchronous, interval-oriented simulation, where time is advanced from time t to $t + \Delta t$ in uniform fixed increments of Δt. The second method, event-oriented simulation, is asynchronous and time may advance in variable intervals. Using this approach, time is "incremented from time t to the next event time t', whatever the value of t'" (24:136).

The emphasis of current research is on discrete event simulation. This approach is well-suited to digital logic simulation where only a small portion of the circuit, typically 10-15 percent, is active at a given time (9:67). Hence, the inefficiency of simulating every element in a circuit, when only a fraction are switching, is avoided.

Efforts to improve the performance of logic simulation fall into two categories. The first is a top down approach of divide and conquer. That is, divide the circuit into smaller, more manageable modules for which the simulation costs are not so severe. This approach is often plagued by difficulties in providing effective tests for the interfaces between modules (12:449).

The second approach is to optimize the performance of the simulation itself through various speed-up techniques. One avenue considered in this approach is to identify those portions of the simulation software that occur frequently and are time consuming to execute. According to Wong the operations to consider for recoding are event-list manipula-

tion, function evaluation, and net-list searching, as they account for 85% of the execution time of a logic simulation algorithm (31:47). Recoding of this software attempts to improve efficiency through the use of hand optimized assembly language. Unfortunately, algorithm optimization seldom yields more than a three fold increase in speed (4:130).

Another prevalent approach to accelerating simulation is the use of special purpose hardware and digital computers tailored to logic simulation. Special purpose computers can have performance orders of magnitude faster than the current software simulators (12:449).

Special purpose hardware architectures attempt to exploit the concurrency within the simulation algorithm. This concurrency results when different events are scheduled for the same time, which occurs frequently in logic simulation (1:84). Catlin offers two approaches to parallelizing this inherent concurrency in simulations, data partitioning and functional partitioning. Data partitioning employs several processors performing identical functions on different portions of the input data. However, the complex interprocessor communications and elaborate hardware requirements make this approach unattractive. Functional partitioning takes advantage of the structure of the simulation algorithm. The algorithm is broken into portions of approximately equal complexity and disjoint data structures. Each portion of the algorithm is then assigned to a separate processor for execution (4:130).

Pure parallelism is not always possible. Often processes must access the same data as in the case of an event list maintained in one memory. Concurrency is still possible with multiple processing elements. Each processing element performs an individual task while data flows between them in a pipeline fashion (1:84).

1.4 Constraints

Benefits from speed-up improvements in discrete event simulation can extend not only to digital logic simulation but also to a variety of applications. The need to speed up digital logic simulation is obvious; however, this research applies to the broader area of discrete event simulation in general. The potential for speedup of digital logic simulation may be

limited by focusing on the greater objective of enhancing all discrete event simulations. This larger domain of discrete event simulation applications may constrain the design of accelerator hardware. Design options that would enhance aspects unique to digital logic simulation would necessarily be dismissed in favor of more general applications.

The majority of discrete event simulations are currently executed on the Intel iPSC/2 hypercube. Continued use of this distributed architecture, based on the Intel 80386 CPU, is anticipated as it represents near state-of-the-art technology and is readily available for research use.

1.5 Scope

Implementation of a specific hardware simulation accelerator was not the goal of this effort. The research focused instead on a detailed requirements analysis for the design of specific hardware enhancements to accelerate discrete event computer simulations using the Intel iPSC/2.

The detailed requirements specification was documented using VHDL. Validation and evaluation of the design and degree of speedup realized was conducted through VHDL simulations.

The target architecture is a distributed parallel computer; however, design testing was performed on a single processor model, representing a single processing node of the iPSC/2 hypercube. The effects of interprocessor communication, processor synchronization, and load balancing were not measured in this configuration, rather the accelerator performance, relative to CPU execution time, was evaluated.

1.6 Standards

The evaluation of simulation speed is sometimes ambiguous. Simulation performance is rated using different measurements throughout industry and current literature. Common measurements include gate evaluations per second, instructions per second, and events per second. Each measurement provides different information about a simulation's performance.

This effort focused on the simulation execution time for a particular class of simulation modeling — discrete event. The actual run times of a specific discrete event simulation provided the datum for this effort.

The proposed hardware accelerator design was evaluated with respect to to this standard. Abramovici contends that an order of magnitude speed up is a minimum design goal (1:33); however, DARPA's thrust is a speedup of three orders of magnitude over traditional simulations through the use of parallel processing, supplemented with a dedicated hardware accelerator.

The objective of this effort was to determine how a general discrete event simulation might be improved through a hardware accelerator, and to design such a hardware enhancement. DARPA's speedup goal is not entirely dependent upon this effort. Related research in techniques to optimize simulation process distribution and minimize the effects of load imbalance, communications overhead, and synchronization delay in a parallel implementation will add to the performance gains realized through hardware acceleration.

1.7 Approach/Methodology

The analysis of a general discrete event simulation model provided a definition for the problem space and was used as the foundation for the remainder of this effort. This analysis clearly defined the portions of the simulation model that exhibit the greatest potential for speedup through hardware enhancements. Specifically, those areas of the simulation model that required the greatest portion of overall execution time and relative frequency of execution were emphasized in the design of a hardware accelerator.

The potential for simulation speedup via the application of special purpose hardware was evaluated along with the trade offs associated with the a hardware implementation. The hardware accelerator requirements were specified and implemented using VHDL.

A testbed was devised to evaluate the VHDL accelerator design. A VHDL behavioral model of the Intel 80386 CPU was not available, hence a complete CPU/accelerator system evaluation could not be performed. Rather test vectors representing discrete event

simulation instructions and data along with CPU interface and control signals were used for accelerator design evaluation.

The VHDL design tests were iterative in nature and revealed both design strengths and shortcomings. The test evaluation and feedback process was instrumental in the design's evolution. Portions of the design remain as VHDL behavioral descriptions; however the detailed requirements for a discrete event hardware accelerator are completely specified when considering the design as a whole.

II. Simulation Acceleration Issues

2.1 Introduction

The use of computer simulation to predict the outcome of events or the performance of physical processes is not new. Computers have provided a means to simulate a broad range of problems in fields as diverse as engineering, economics, sociology, and weather. This proliferation of computer simulation has led to models of increased complexity and often time-consuming simulation programs.

The Department of Defense is keenly aware of the time-consuming nature of complex simulations. Time delays to conduct simulations may adversely impact a commander's ability to make an informed decision or delay the development of a new system.

Considering the diverse applications and increased reliance on computer simulations, the Department of Defense is investigating methods of speeding up the simulation process. The DoD's emphasis on Very-High Speed Integrated Circuit (VHSIC) technology is one area that requires significant improvement in simulation speed. This is readily apparent if one considers that simulating one second of real-time for an application specific integrated circuit may take days of dedicated processor time (14:42).

This chapter is an overview of different approaches available for accelerating computer simulation. Various simulation methods are described and options for accelerating the simulation process from a hardware perspective are presented.

2.2 Simulation Techniques

The two main categories of simulation are continuous (time-driven) and discrete-event simulation. The time-driven approach is characterized by regular advances, of a predetermined and fixed increment, of a simulation clock. The values of all simulation variables are evaluated and updated after each clock advance. If no variables are affected, the clock simply advances. Event-driven simulations use a clock that advances to the future time of the next scheduled event. In discrete-event simulation, scheduled events are repeatedly fetched from a queue and simulated and only those variables affected by the

event are updated. Each event simulation may spawn new events which are inserted into the event queue at the appropriate time (22:39).

A discrete-event simulation allows the simulator to skip intervals of time where no events are scheduled. The modeling of complex digital circuits is well suited for discrete-event simulation since signal values change at discrete times and only a limited number of circuit elements are active at any given time.

2.3 Distributed Processing

Discrete-Event Simulation (DES) programs often require computational capabilities that exceed the fastest available machines (13:81). Parallel computer architectures have the potential to overcome the speed limitations of single processor computers and thus, have received widespread attention.

2.3.1 Taxonomy for DES Architectures A notation similar to Flynn's for parallel architectures (e.g. MIMD, SIMD etc.) can be used to describe the main architectural features of DES machines. The basis for this taxonomy is the DES algorithm and its three essential elements:

- Time control
- Event list control
- Event (function) evaluation

The implementation of these components may vary between simulators but, in one form or another, they are all present in any DES (12:450). The time control component (clock control) determines the progression of simulated time. The event list control component schedules events in increasing time order and the event evaluation component processes the accessed events and determines if new events should be scheduled.

The taxonomy has four components: two specify time control characteristics, and one each for specifying event list control and event evaluation components (see Table 2.1). The time control mechanisms define the simulation's classification with respect to time –

unit increment corresponding to continuous time and event based increment corresponding to discrete event time. In a multiprocessor system, synchronization may be provided by a single "global clock" or each processor may maintain a "local clock."

Table 2.1. DES Taxonomy Components

1. TIME CONTROL MECHANISMS	
A. TIME ADVANCE	
1) Unit Increment	(UI)
2) Event-based Increment	(EI)
B. TIME SYNCHRONIZATION	
1) Global Clock	(GC)
2) Local Clock	(LC)
2. EVENT LIST ATTRIBUTES	
1) Single List	(SL)
2) Multiple List	(ML)
3. EVENT/FUNCTION EVALUATION	
1) Single Machine	(SM)
2) Multiple Machine	(MM)

Similarly, the event list can be distributed and portions maintained by each processor or totally by a single processor. A distributed, or multiple event list, eliminates the delay time to communicate the next scheduled event and is potentially faster than the single event list.

The last component, event/function evaluation, indicates whether a single or multiple processors are used. Using this taxonomy sixteen possible machine architectures can be specified by the tuple:

Time Advance/Time Synchronization/Event List Attributes/Event Evaluation

Eight of the sixteen possible architectures are implemented with a single machine (SM) and represent traditional sequential architectures. The remaining eight are multiple machine (MM) architectures which include the Intel iPSC/2 hypercube.

2.3.1.1 Parallel Architectures The multiple machine architectures represent parallel processing systems. Speedup is obtained by distributing the simulation workload

among several concurrent processors; however, this is not without some cost. Each parallel architecture has some limitation that must be considered and its effects evaluated during system design (12:452).

2.3.1.2 Multiple Machines with Global Clocks – X/GC/Y/MM These architectures take advantage of parallel function evaluation to speed up simulation. A single processor acts as a master and maintains the global clock. The use of a global clock minimizes time synchronization problems; however, an efficient communications network is required and the logical processes must be partitioned for effective load balancing. Parallel event list manipulation is also possible with multiple event lists. This eliminates the potential bottleneck of a centralized event list, but also requires distribution of event time information between master and slave processors to ensure global clock updating.

2.3.1.3 Multiple Machines with Local Clocks – X/LC/Y/MM Potential speedup in these architectures is obtained through parallel function evaluation, parallel event list manipulation, and distributed time management. MIMD machines, such as the Intel iPSC/2 hypercube, are represented in the taxonomy as EI/LC/ML/MM. Here the simulation is mapped as a set of autonomous communicating processes that exchange time synchronization and state information through asynchronous message passing (5:198). This distributed time management allows variable states to be evaluated as each input value changes.

2.3.2 Distributed Discrete Event Algorithms In a distributed processing environment, discrete-event simulations map one or more server/queue pairs onto the active processors in the network. Each processor operates with its own simulation clock and messages are timestamped to reflect the simulated time at the sending node. Individual processors may have separate processes executing on them and messages are routed between the processor pairs by directed channels (22:51).

Various distributed discrete event algorithms have been proposed, but two approaches, the Chandy-Misra algorithm and the Time Warp algorithm, are most notable (28:8). The distinguishing feature between these algorithms is how they manage simulation time.

2.3.2.1 Optimistic Paradigm – Time Warp Algorithm The Time Warp algorithm relies on general lookahead - rollback as its fundamental synchronization mechanism (20:404). Each local simulation clock advances independently unless conflicting information (i.e., a message from the past) occurs, at which point the local simulation clocks are "rolled back" to a consistent state, antimessages are sent to override the erroneous messages, and execution advances along a revised path (28:8).

The underlying principle of Time Warp is the concept of "virtual time." Virtual time is a temporal coordinate system used to measure progress and ensure synchronization. Each processor is updated with the global virtual time, which only progresses forward, in addition to its own simulation, or local virtual time. For a given real time, the global virtual time represents the minimum of all local virtual times and the virtual send times of all messages that have yet to be processed (20:417).

The primary overhead cost of Time Warp is associated with rollbacks and the communication of antimessages needed to implement a rollback (20:416). Additionally, previous state information must be maintained to allow message cancellation and rollback to the current global virtual time.

2.3.2.2 Conservative Paradigm – Chandy - Misra Algorithm The Chandy-Misra algorithm models a physical system as a distributed network of logical processes communicating via messages. The event list and global simulation clock, of traditional sequential simulations, are replaced with an event list and local clock at each logical process.

An effective implementation of the Chandy-Misra algorithm is dependent upon the following requirements (5:198-199):

- The behavior, at time t, of the physical process being modeled must not be affected by messages transmitted after t. This is referred to as the *realizability condition.*
- Messages between processes must increase monotonically in time (*monotonicity condition*).
- Messages between logical processes must correspond exactly to the sequence of messages between physical processes (*predictability*).

The primary difference between the Chandy-Misra algorithm and the Time Warp technique is the use of "null" messages. Null messages are encoded with a timestamp to tell the receiving node that no *real* message will be transmitted before the specified time. Hence the receiving node may process existing messages without the possibility of reversal at a later time (28:9).

2.4 Discrete-Event Logic Simulation

Digital logic circuits are simulated by modeling the circuit elements to determine signal values for a given sequence of input signals. The data necessary to simulate an element is referred to as the element record. The element record typically contains current input values, current output, one or more delay time values, the element type code, fan-out count and destination, and a set of exception flags (30:4). The major functions of a discrete-event logic simulator include element data management, element evaluation, event management, and exception handling.

Any change in the value of an input, output, or state variable of a given element is referred to as an event. Events occur at discrete points in simulated time. An element whose input or state variable has changed is evaluated to determine its new output and state. Transitions of state variables and generation of new outputs must be scheduled for some future time as delays are usually associated with the operation of elements (1:83).

Scheduled events are maintained on an event queue. A simulation time-flow mechanism manipulates the events and ensures that they occur in correct temporal order (1:83). When all events at the current simulation time are exhausted, the time is advanced to the next time for which events are scheduled.

Manipulation of the event queue ensures the proper time sequencing of evaluations. Additionally, only those elements scheduled for an event are evaluated at a given simulation time. This reduction in number of elements evaluated incurs the additional overhead of manipulating the event queue. Comfort estimates that between 32% and 40% of all non-input/output computer time may be spent in event queue processing (8:117).

2.5 *Speedup Alternatives*

Rarely content with current technology and capabilities, computer and software designers continue to investigate methods of speeding up computer operations. This section presents an overview of speedup efforts in the area of simulation.

2.5.1 Software Acceleration Alternatives for accelerating the execution of logic simulations have been proposed. The first approach often considered is recoding software for the most frequently occurring element routines and the event queue manager (4:130). This approach improves efficiency through the use of hand-optimized assembly language. Unfortunately, this approach seldom realizes more than a three-fold increase in speed (4:130). Additionally, this implementation limits the transportability and maintainability of the software (30:2).

2.5.2 Application Specific Hardware Another approach is to acquire a faster machine or to develop hardware exclusively for simulation. This option provides the greatest performance increase and can be as much as 100 to 500 times the speed of software simulations run on a sequential microprocessor (3:27-29). The disadvantage to this approach is that special hardware is usually difficult to modify in the field and often cannot be used for anything else (4:130). Direct implementation of the simulation software in hardware is also feasible but expensive and inflexible (3:21).

Several design options for special purpose hardware to speed up simulation are available. Smith suggests the use of one or more stages of microcoded hardware designed especially for high performance simulation (30:2). Using this approach, four processors could form a pipeline with stages for event queue management, evaluation routines, and signal change propagation.

2.5.3 Functional Partitioning Catlin and Paseman contend that the structure of the simulation algorithm can be exploited through functional partitioning. The simulation algorithm is broken into three pieces of approximately equal complexity. A separate processor is assigned to each portion of the algorithm and its associated data structures. The tasks of queue management, state maintenance, and element evaluation are performed

in disjoint processors and therefore operate simultaneously. Communication between the processors is through low bandwidth First In First Out (FIFO) channels and processing is done in a dataflow fashion. A host microprocessor serves as the nucleus of the system and provides a user interface during the simulation (4:130-132).

A network of inexpensive but powerful microprocessing elements is viewed by Comfort as the best method of attaining high instruction execution rates at a moderate cost (7:197). Similar to Catlin and Paseman, Comfort also proposes partitioning the simulation into functional processes. The function of event set processing comprises one partition and is assigned a variable number or processors each having 'next,''schedule,' and 'cancel' functions. The remaining partition consists of all other processing associated with the simulation and is assigned to the host processor. The host processor polls the event set processors for their event notice of smallest next processing time. The host then selects the notice with the smallest (global) time and acts upon it (8:118).

2.5.4 Content-Addressable Memories The use of random access memories for data storage and retrieval has inherent drawbacks because of its word-at-a-time, location-addressed implementation (6:51). Addressing by location is inefficient, particularly if data is dynamically unordered during processing.

Content-Addressable Memories (CAMs) are capable of accessing data based on content rather than memory location. This ability permits data searches for exact matches with a specified key or relative comparisons for an ordered data retrieval (25:725).

Considerable speedup in processing time is possible with content-addressable memories. This results from the simultaneous access of data in parallel and the elimination of the need to store data in sorted order (15:509,518).

2.6 Summary

Simulation is an integral part of decision making in various disciplines. The use of computers for simulation has increased dramatically over the past 20 years and simulation models have become more complex. The increased model complexity necessitates computer

enhancements to minimize the time required to run these simulations—particularly in digital logic simulation where simulations may take days to run.

The dominant approach to enhancing computer simulations is to distribute the workload among multiple processors working in parallel. Several options of parallelizing the simulation are available to the designer. Processor networks operating in a dataflow fashion are feasible as are pipelines of multiple stages. In both approaches the simulation algorithm is partitioned among the processors for independent processing.

Every designer must consider the cost of design implementation. An additional consideration for the design of a simulation accelerator is flexibility. Application specific hardware is often inflexible and one must consider the tradeoffs between speedup potential and the opportunity for reuse in other applications.

III. Methodology

3.1 Introduction

The design of a Discrete Event Simulation (DES) hardware accelerator requires a detailed analysis of a general DES algorithm. The objective of this analysis is to identify simulation functions and routines that are frequently invoked and/or account for a large portion of the overall simulation execution time. Once determined, simulation acceleration is possible through implementation of these functions in hardware(31:47).

The methodology used to analyze a general distributed DES is presented in this chapter. A description of the simulation testbed and the configuration of simulation logical processes is given.

The parallel architecture of the Intel iPSC/2 hypercube is described and the different simulation topologies employed are presented. The methods used for collecting simulation data and the metrics for evaluating the data are presented along with the results of this analysis.

3.2 Discrete Event Simulation Testbed

The parallel Discrete Event Simulation (DES) environment for this effort consisted of an eight node Intel iPSC/2 hypercube employing the SPECTRUM simulation protocol interface designed by the University of Virginia. The conservative, Chandy-Misra null message protocol was used for parallel synchronization.

3.2.1 SPECTRUM Interface SPECTRUM is a generic testbed designed for evaluating parallel simulation protocols (29:865). Through the use of user defined protocol filters, SPECTRUM provides a transparent interface between the application being modeled and the parallel processing architecture used to execute the simulation.

The application to be simulated contains one or more physical processes which are modeled through Logical Processes or LPs. Each simulation Logical Process (LP) is composed of three separate entities when executed under SPECTRUM. Referring to Figure 3.1, each LP contains an application component, a process manager, and a node manager.

Application components are portions of the original application which may be executed concurrently. The process manager provides routines to support typical simulation requirements such as managing simulation time and event queues. Low level system requirements, such as message passing between LPs and scheduling, when multiple LPs are mapped to a single processor, are provided by the node manager.

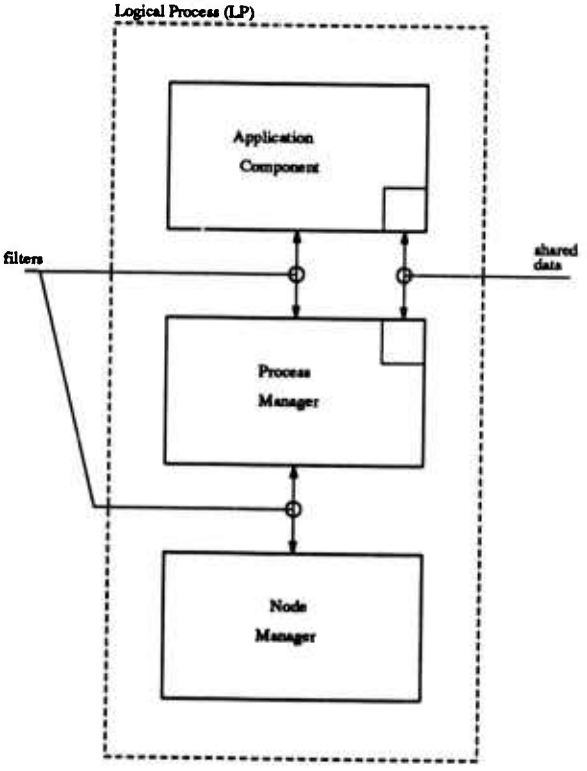

Figure 3.1. SPECTRUM Testbed Logical Process(29:868)

The simulation protocol is implemented with SPECTRUM via a user defined filter. The filter provides the synchronization functions necessary for effective simulation execution. The basic filter functions required for discrete event simulation are: initialize, get-next-event, post-event, advance-time, and post-message. All but the last function occur between the application layer and the process manager. Post-message is a message handling function which occurs between the process and node managers(29:868). Hence the use of filters provides the interface between separate modules within each LP while providing the user easy access for modifying, or replacing, the synchronization protocol.

3.2.2 Simulation Application The application used for analysis and modeling a general Discrete Event Simulation is a simple car wash. The physical process of washing cars is modeled by three logical processes. A source, which generates customers for the system. A wash, where the customer service is simulated, and an exit where the customers depart from the system.

Parallelism is achieved through multiple instances of source and wash LPs. Figure 3.2 shows the configuration of LPs for the car wash simulation. Although different configurations are possible (i.e., more exits or fewer washes), extensive revisions of the application source code would be necessary. Since the multiple instances of sources and washes are mutually exclusive (i.e., no data dependencies) in this configuration, concurrent execution of these LPs is possible.

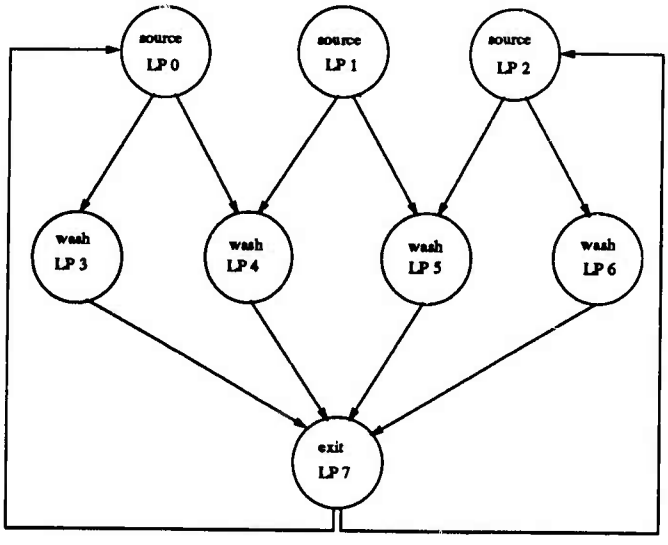

Figure 3.2. Car Wash Simulation, Logical Processes

The interconnecting arcs, which provide message passing channels between LPs, are established during initialization and remain fixed throughout the simulation. The simulation is deterministic in that customer arrival rates are constant, although customers are generated at different frequencies at each source. Likewise, the service rate for a given wash is fixed; however, this rate also varies between individual wash LPs. The routing of customers from source to exit follows the interconnecting arcs and is also deterministic,

with the path taken at forks being a function of the customer, or car number.

3.2.3 Parallel Processing Architecture An eight node Intel *i*PSC/2 hypercube provided the parallel processing architecture for executing a gener: DES. Using Franklin's taxonomy from Section 2.3.1.3, this architecture can be classified as EI/LC/ML/MM, since the DES is event driven, uses local clocks for simulation time, multiple lists for scheduling next events, and multiple processors in a hypercube configuration.

The basic architecture of each cube node is a self-contained computer with a CPU, local memory for programs and data, and an input/output (I/O) subsystem. The distinguishing feature of the *i*PSC/2 is the set of bidirectional I/O channels linking each node to its n immediate neighbors in the hypercube.

The number of immediate neighbors, n, also represents the dimension of the hypercube. With $n = 3$, a three dimensional graph representation of the *i*PSC/2 is shown in Figure 3.3. This figure depicts an eight node configuration of the hypercube and the nearest neighbor interconnections.

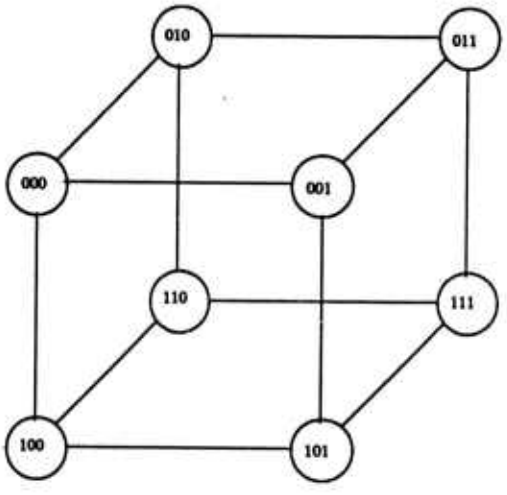

Figure 3.3. 8 Node Hypercube Configuration (17:1830)

The distributed memory architecture of the hypercube necessitates message passing between nodes when information must be shared. The bidirectional I/O channels, linking nearest neighbors, play a central role in the hypercube's performance. The *i*PSC/2 uses

Direct-Connect Modules (DCM) which provide the necessary routing logic in the hypercube interconnect topology for this purpose.

Earlier versions of the hypercube used a store-and-forward communication scheme requiring approximately 1 ms to pass messages between adjacent nodes (17:1832). The DCM of the iPSC/2 uses a 16 bit routing probe to encode node address information. This allows the sending node to establish an end-to-end link with the receiving node by routing through intermediate nodes along the path (10:448-452).

3.3 Simulation Configuration

Two simulation parameters and mappings of LPs to processing nodes were varied during the analysis. The effects of feedback on the simulation's execution were investigated by either routing customers from the exit back to the source for resubmission or allowing a straight exit.

Additionally, an artificial workload, referred to as a *spin loop*, was implemented by inserting varying size loops with floating point operations within each LP. This variable workload provided a more realistic and general simulation for analysis as compared to the strictly deterministic carwash. The effects on function execution frequency and overall function execution time were analyzed by varying the computational intensity between LPs.

Figure 3.2 shows the carwash LP configuration which was fixed for all simulation runs. Representing the physical process with eight LPs provided a direct one-to-one mapping of LPs to the eight processing nodes on the iPSC/2 hypercube. To investigate the effect of multiple processes executing on each node, the mapping shown in Figure 3.4 was used. Similar to the original mapping, the basic LP configuration of the simulation is unaltered; however, only four computing nodes of the hypercube, each having two processes, are employed. Although many mapping options of the eight LPs are possible, this mapping was chosen since it consolidates communication paths and minimizes off-node communication(21:4-2).

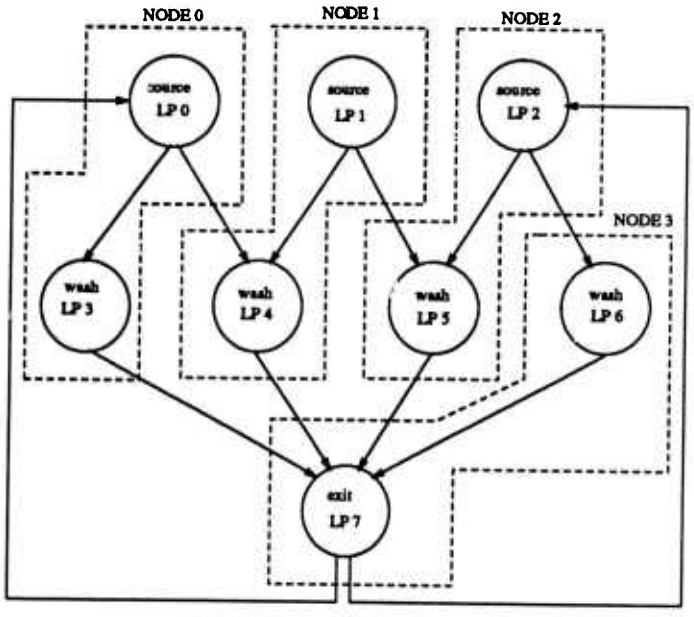

Figure 3.4. Carwash Configured with Two LPs per Node

3.4 Data Collection and Analysis

A direct means for parallel algorithm analysis was not available during this effort, therefore the DES algorithm had to be instrumented for data collection.

3.4.1 Algorithm Instrumentation Section 3.2.1 described the levels and modularity of the SPECTRUM testbed. Each level of SPECTRUM has a corresponding level of software in the DES algorithm. The carwash application level (afitwash.c) has direct visibility to the process manager (lp_man.c) for event list and time management functions. The process manager in turn has visibility to the synchronization protocol filter (myfilters.c) and the node level message passing functions (cube2.c).

The analysis of a general DES required instrumenting the functions of the process manager, the protocol filter, and the node level routines. Figures 3.5-3.7 show the function hierarchy of each level of the DES algorithm.

The algorithm was instrumented to gather function execution data. The relative function execution frequency of each logical process was calculated at each level (i.e. filter,

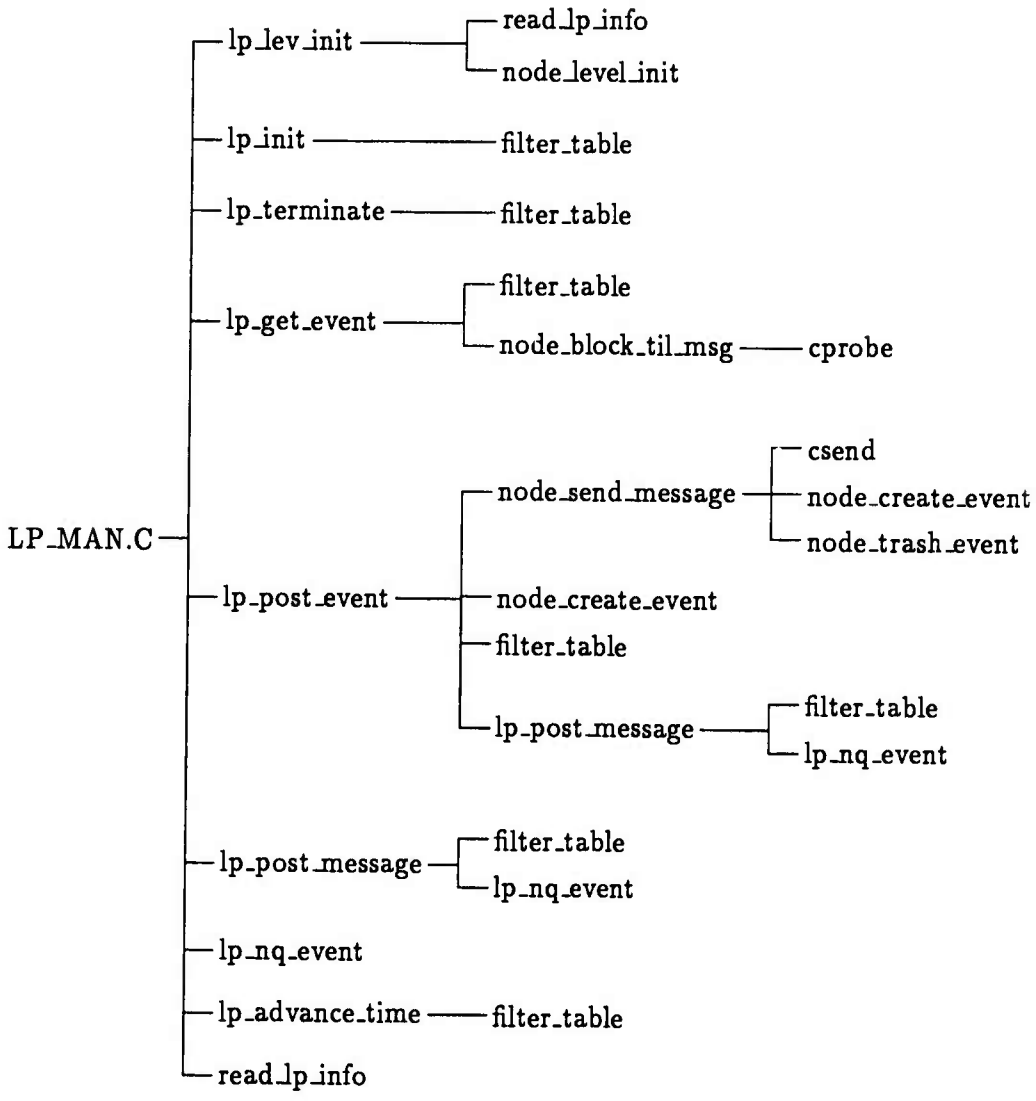

Figure 3.5. Function Hierarchy of Process Manager Level

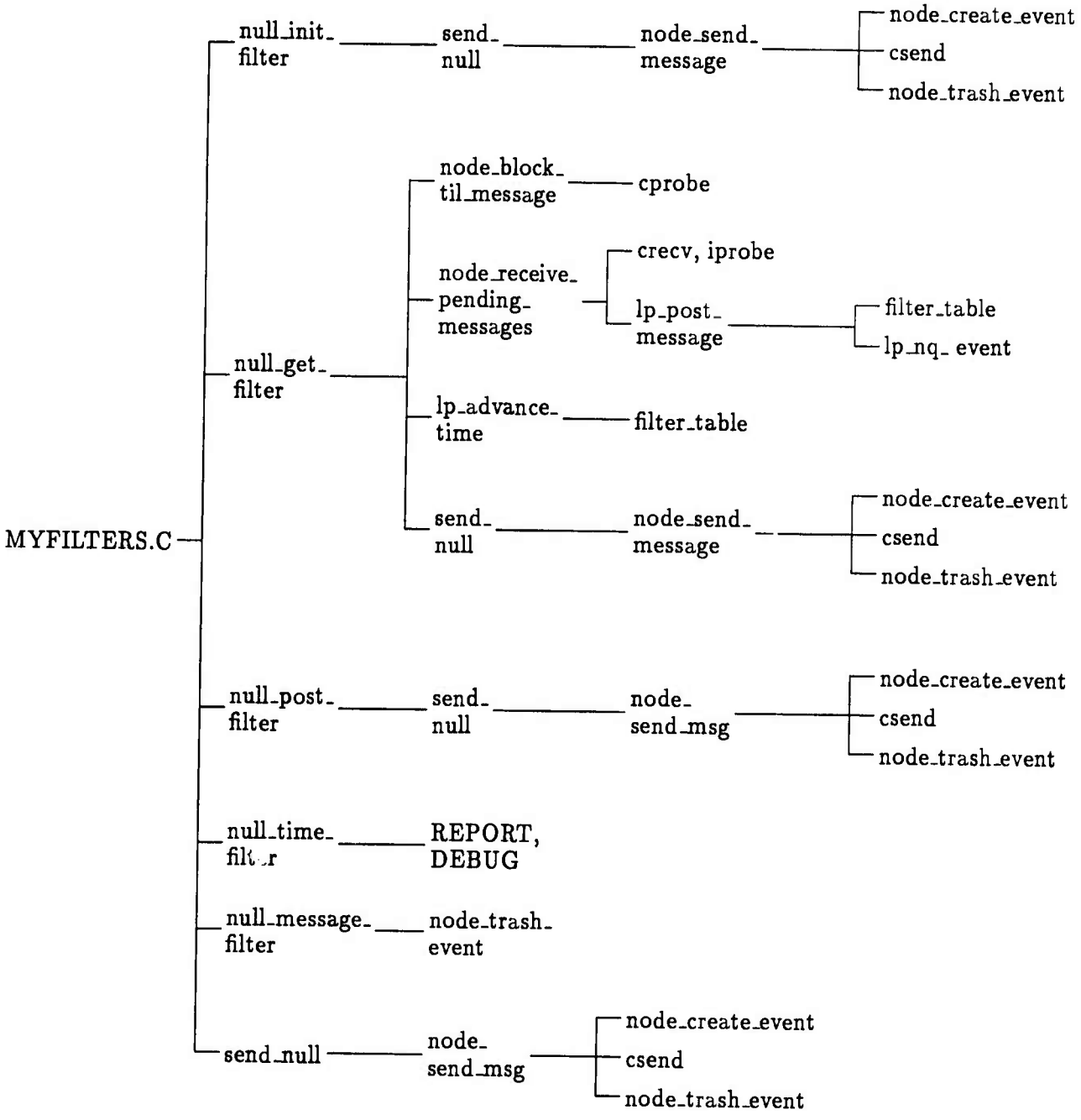

Figure 3.6. Function Hierarchy of Filter Level

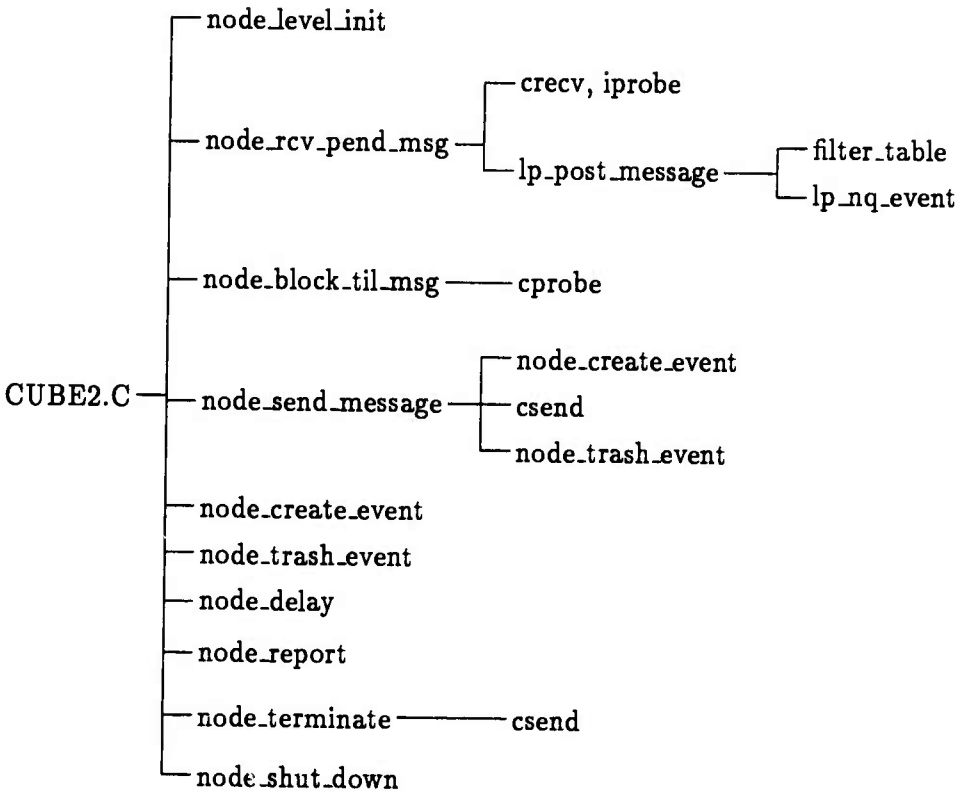

Figure 3.7. Function Hierarchy of Node Level

process manager, and node manager) of the DES algorithm. Therefore data was collected from each LP separately and then categorized based on algorithm level.

The instrumentation was implemented by encapsulating each function with variables to monitor function count and execution time. The function execution count was incremented during each successive function call, while the cumulative function execution time was updated with the difference of each function's start and end times. Variable updating was concurrent with simulation execution; however, the final data totals were reported only after the simulation had completed, thus avoiding the significant overhead of updating data files during simulation execution.

3.4.2 Data Analysis Metrics The function hierarchies of Figures 3.5–3.7 show the interrelation between software levels and functions in the SPECTRUM testbed described in Section 3.2.1. The complexity of determining the portion of overall simulation execution time for each level is compounded by the inter-level dependencies and function calls. To simplify the analysis, each level of the simulation software was considered separately with the knowledge that subfunction calls are an integral part of the algorithm which contribute to the calling function's total execution time. A relative comparison, with respect to the total number of function executions and total execution time, of each level's primary functions (i.e. top tier of the function hierarchy) was then made for each level.

By averaging the data of each level's primary functions across all simulation LPs, a general view of the individual function performance was obtained. This average execution data reveals the primary functions and portions of the algorithm that are the most time consuming, and thus have the greatest need for acceleration.

An analysis of relative simulation execution times for multiple LPs per computing node, as shown in Figure 3.4, was also made. The need for this analysis is based on the fact that the modeling of many physical processes generates more logical processes than the number of available processing nodes. Modeling with large complex logical processes, by combining smaller LPs, is possible but may not always result in a one-to-one mapping between LPs and processors.

The analysis of data for processing nodes with multiple LPs considered the additional overhead for task scheduling and switching. The measurement of switching time for individual LPs is extremely difficult, and in fact, was not possible on the iPSC/2 as the system clock resolution is in milliseconds, while the task switching time for the Intel 386 DX is about 17 μs (19:5-335). Task scheduling is an operating system function on the hypercube that is performed in a round-robin fashion. Each LP on a given processing node runs for approximately 50 ms, or until it blocks to send or receive a message (18:2-56).

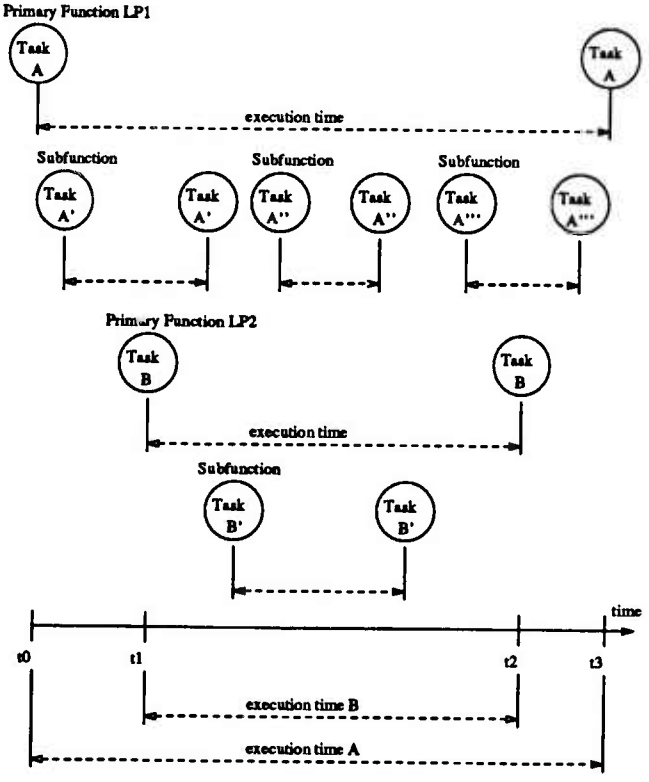

Figure 3.8. Task Swapping Runtimes

To get an accurate representation of relative simulation execution times, the data for multiple LPs per processing node had to be adjusted to account for the effects of task switching. Figure 3.8 is an example of two LPs whose execution times overlap as a result of task switching. The runtime for the second LP, (B), occurs while LP (A) is *swapped out* and the entire runtime of LP (B) is included within (A's) total execution time. Likewise, portions of (B's) execution time can be attributed to LP (A) regaining the CPU; therefore,

both LPs' overall runtimes must be adjusted accordingly. This adjustment was possible by tracking not only the execution start and finish times, but also the LP's processing node and process ID number. Once a clear distinction between LPs is made, the adjustment to function execution times was made by subtracting the execution time for the second LP's function from the first. After adjusting the LP execution runtimes, the average relative simulation execution times, for each level's primary functions, were calculated as described above for a single LP per processing node.

3.5 Logical Process Function Execution

The analysis of simulation test data for single and multiple LPs per processing node, both with and without feedback, and for various spin loops all yielded similar results. The communications overhead of message passing, necessary to implement the conservative synchronization protocol, accounted for the largest relative portion of simulation execution time.

The data collected during this analysis is tabularized in the following sections. The tables are not exhaustive in that, only those functions with the largest relative execution times for each level were of primary interest and thus are included. Many of the functions that were omitted required little or no measurable execution time (i.e., lp_terminate, node_terminate, node_trash_event), regardless of the simulation configuration. Although some primary functions have been omitted, the tables typically account for 80 percent, or more, of the overall simulation execution time at each level.

3.5.1 One LP per Processing Node Tables 3.1 through 3.3 show the primary functions with the largest average relative execution times for each level of the DES algorithm used for the carwash simulation. These tables reflect data from simulations executed with one LP per processing node both without an artificial workload and with equally sized spin loops executed on all LPs (i.e. sources, washes, and exit) in the simulation.

Table 3.1 indicates that the process manager level expends the greatest time, on average, posting events. In the conservative synchronization protocol this entails sending

event messages to one or more designated LPs, while null messages with a safe lookahead time for tnat channel are sent on all remaining output channels.

Table 3.1. Mean Relative Execution Time for Process Manager, 1 LP per node

Primary Function	No Spin		Spin (1/1/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
lp_post_event	0.371	0.334	0.398	0.379	0.371
read_lp_info	0.343	0.379	0.337	0.333	0.348
lp_post_msg	0.151	0.150	0.137	0.183	0.155

The time spent executing read_lp_info occurs only during simulation initialization. Although a significant portion of the overall average relative execution time, its one-time execution make it an unlikely candidate for a hardware accelerator.

Table 3.2 clearly shows that a significant portion of the overall average execution time for the filter level is dedicated to supporting communication requirements. The overhead of the send_null function translates to idle processor time. During this idle time, the processor waits for a two-way request to send and acknowledgement from the receiving node necessary for the csend subfunction of figure 3.6.

Table 3.2. Mean Relative Execution Time for Filter, 1 LP per node

Primary Function	No Spin		Spin (1/1/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
send_null	0.400	0.425	0.412	0.429	0.417
null_msg_flt	0.253	0.240	0.224	0.216	0.233
null_get_flt	0.197	0.194	0.204	0.199	0.199

The primary functions, with the largest average relative execution times, of the node manager level are shown in table 3.3. As in the other DES algorithm levels, a majority of the average execution time is directly related to communication overhead. Similar to the send_null function of the filter level, both node_btm (block_til_message) and node_rpm (receive_pending_messages) are implemented with subfunctions (see figure 3.7) that require the processor to wait (i.e. crecv - wait to receive, and cprobe - wait for message on channel(18:2-17,19)).

This waiting for communication is an inherent requirement of the conservative Chandy-Misra synchronization protocol discussed in section 2.3.2.2. This waiting again indicates CPU idle time which, given the opportunity, could be redirected to other simulation requirements.

Table 3.3. Mean Relative Execution Time for Node Manager, 1 LP per node

Primary Function	No Spin		Spin (1/1/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
node_btm	0.429	0.344	0.454	0.286	0.378
node_rpm	0.334	0.359	0.292	0.372	0.339
node_sm	0.086	0.108	0.101	0.140	0.109

3.5.2 Variable Spin with One LP per Processing Node The logical process mapping of figure 3.2 was used for LPs of varying artificial workloads to demonstrate a broader and more generic class of simulations than the deterministic carwash. The workloads, or spin loops, were implemented arbitrarily but in proportions considered to approximate the expected computational intensity of the particular logical process. Therefore the ratios shown in table 3.4 through table 3.6 represent the ratio of computational workloads of the sources, washes, and exit respectively.

The effects of an increased workload on average function execution time are apparent at the process manager level shown in table 3.4. The sources and exit continue to generate messages at approximately the same rate, however the washes, with a considerably larger workload, require more time to process a backlog of incoming messages. Hence, considerable time is spent posting received messages which includes a linked-list queueing subfunction (see figure 3.5) of time complexity $O(n)$.

The time dedicated to posting events is significant but relatively constant. Since the workloads of the sources and exit are unchanged they continue to post events at nearly the same rate as without an artificial workload (see table 3.1).

The relative execution times of the synchronization protocol are shown in the filter level of table 3.5. The addition of spin loops had little effect on relative execution times of this level's primary functions. The communication overhead of sending messages again

Table 3.4. Mean Relative Execution Time for Process Manager, 1 LP Variable Spin

Primary Function	Spin (1/5/1)		Spin (1/10/1)		Spin (1/20/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	FDBK	NOFDBK	
lp_post_event	0.340	0.328	0.332	0.315	0.335	0.316	0.328
lp_post_msg	0.184	0.201	0.198	0.210	0.210	0.221	0.204
read_lp_info	0.174	0.169	0.138	0.140	0.107	0.107	0.139

accounts for a significant portion of the average execution time. The slight decline in send_null execution time, relative to LPs with no spin loops (see table 3.2), is the result of additional computational activity at the wash LPs, hence fewer output messages.

Table 3.5. Mean Relative Execution Time for Filter, 1 LP Variable Spin

Primary Function	Spin (1/5/1)		Spin (1/10/1)		Spin (1/20/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	FDBK	NOFDBK	
null_msg_flt	0.303	0.302	0.362	0.379	0.456	0.467	0.378
send_null	0.337	0.324	0.256	0.234	0.201	0.192	0.257
null_get_flt	0.215	0.234	0.214	0.237	0.205	0.201	0.218

As expected the node manager level of table 3.6 received the greatest impact from the increase in computational workload. The floating point calculations used in the spin loops clearly accounted for the majority of average function execution time at this level.

Table 3.6. Mean Relative Execution Time for Node Manager, 1 LP Variable Spin

Primary Function	Spin (1/5/1)		Spin (1/10/1)		Spin (1/20/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	FDBK	NOFDBK	
node_spin	0.371	0.425	0.443	0.500	0.493	0.549	0.464
node_rpm	0.232	0.253	0.216	0.236	0.205	0.228	0.228
node_btm	0.249	0.135	0.193	0.097	0.154	0.058	0.148

Table 3.6 also shows that communication overhead accounts for significant execution time at the node level, in spite of the added computational workload. The bottleneck at

the wash LPs requires the node_rpm function to receive and post the the influx of messages, while computations in the spin loop are executed.

3.5.3 Multiple LPs per Processing Node Tables 3.7 through 3.9 show data from simulations executed with two LPs per processing node. Data both with and without an artificial workload, and also for the wash LPs performing ten times the computational intensity of the sources and exit is shown.

At the process manager level (see table 3.7) the lp_adv_time function accounted for the greatest average relative execution time. Although not a communications related function, each time advance requires a memory access to update the LP's local simulation time. The use of sequential random access memory and the frequency of time updates contributes to the average execution time for this function.

Posting events, or sending output messages, requires over 20 percent of the average execution time at the process manager level. This portion of execution time is dedicated to updating output LPs with new events and time information but does little, unless feedback is involved, to advance the local LP's simulation state.

Table 3.7. Mean Relative Execution Time for Process Manager, 2 LPs per Node

Primary Function	Spin (1/1/1)		Spin (1/10/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
lp_adv_time	0.711	0.662	0.709	0.694	0.694
lp_post_event	0.214	0.250	0.220	0.217	0.225

From table 3.8, approximately 60 percent of the protocol filter's average execution time is spent sending null messages when two LPs execute on each processing node. Null messages are necessary to ensure deadlock avoidance with the conservative protocol(22:57), and implementation on a distributed hypercube architecture requires the csend subfunction, a block and wait communication procedure (see figure 3.6).

The communication overhead associated with receiving and sending messages clearly accounts for the majority of average relative execution time at the node manager level. Table 3.9 shows that node_rpm and send_sm together, average approximately 80 percent

Table 3.8. Mean Relative Execution Time for Filter, 2 LPs per Node

Primary Function	Spin (1/1/1)		Spin (1/10/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
send_null	0.689	0.468	0.587	0.618	0.591
null_get_flt	0.131	0.308	0.187	0.179	0.201
null_msg_flt	0.152	0.141	0.172	0.150	0.154

of the node manager level's execution time, regardless of feedback or spin loop size. Both functions support communication and are implemented with subfunctions which block and wait (see figure 3.7). Here, as with other block and wait type functions, better utilization of the associated processor idle time might be possible.

Table 3.9. Mean Relative Execution Time for Node Manager, 2 LPs per Node

Primary Function	Spin (1/1/1)		Spin (1/10/1)		Overall Mean
	FDBK	NOFDBK	FDBK	NOFDBK	
node_rpm	0.513	0.542	0.510	0.413	0.495
node_sm	0.319	0.274	0.321	0.301	0.304
node_btm	0.168	0.184	0.169	0.184	0.176

3.5.4 Speedup Potential The data in tables 3.1 through 3.9 does not indicate the portion of overall simulation execution time expended in each level of the DES algorithm. However, the speedup potential for the individual levels was approximated by considering each level of the DES algorithm independent of the others.

Ideally, the addition of a hardware accelerator will reduce the average execution time of the most time consuming primary functions to near zero. Therefore, by representing each level's total execution time as 1, the speedup potential for each function, S_f, is approximated by:

$$S_f = \frac{1}{1 - \mu_f} \tag{3.1}$$

Where μ_f is the overall mean execution time for the primary function at that level.

Total speedup potential for each level, S_l, was approximated by considering the cummulative effects of reducing the average execution time of all time-consuming primary functions for each level to zero. Eliminating the average execution time for all primary functions at a given level results in the speedup potential shown in equation 3.2.

$$S_l = \frac{1}{1 - \sum_{i=1}^{n} \mu_{f_i}} \tag{3.2}$$

The speedup potential for each DES algorithm level, and the simulation configuration analyzed, are summarized in tables 3.10–3.12. Tables 3.10 and 3.11 (i.e., 1 LP per node) show little potential for significant speedup in any given level of the DES algorithm. Assuming equal execution times for each level of the DES algorithm, averaging the sums of Equation 3.2, indicates the potential total speedup is only 4.83 for one LP per node with no spin and 3.51 times for one LP per node with added spin.

Table 3.10. Speedup Potential by Algorithm Level (1 LP per node, no spin)

Algorithm Level	Primary Functions	Mean Time Ratio	Speedup Potential S_f	Speedup Potential S_l
LP_MAN	lp_post_event	0.371	1.590	2.110
	lp_post_msg	0.155	1.183	
FILTER	send_null	0.417	1.715	6.623
	null_msg_flt	0.233	1.304	
	null_get_flt	0.199	1.248	
NODE	node_btm	0.378	1.608	5.747
	node_rpm	0.339	1.513	
	node_sm	0.109	1.122	

Considerably better speedup potential is exhibited when two LPs share a single node as shown in table 3.12. Using the assumption of equal execution times per algorithm level and Equation 3.2, the total potential simulation speedup, for two LPs per processing node, is 23.62 times.

A more accurate approximatiton of potential simulation speedup was made by using the measured average execution times for each level of the DES algorithm. The calculation

Table 3.11. Speedup Potential by Algorithm Level (1 LP per node, variable spin)

Algorithm Level	Primary Function	Mean Time Ratio	Speedup Potential S_f	Speedup Potential S_l
LP_MAN	lp_post_event	0.328	1.488	2.137
	lp_post_msg	0.204	1.256	
FILTER	null_msg_flt	0.378	1.608	6.803
	send_null	0.257	1.346	
	null_get_flt	0.218	1.279	
NODE	node_rpm	0.228	1.295	1.603
	node_btm	0.148	1.174	

Table 3.12. Speedup Potential by Algorithm Level (2 LPs per node)

Algorithm Level	Primary Function	Mean Time Ratio	Speedup Potential S_f	Speedup Potential S_l
LP_MAN	lp_adv_time	0.694	3.268	12.346
	lp_post_event	0.225	1.290	
FILTER	send_null	0.591	2.445	18.519
	null_get_flt	0.201	1.252	
	null_msg_flt	0.154	1.182	
NODE	node_rpm	0.495	1.980	40.000
	node_sm	0.304	1.437	
	node_btm	0.176	1.214	

of overall speedup potential was made as a weighted average, using each level's overall mean portion of simulation execution time and its potential for speedup at each level. The overall mean portion, t_μ, of simulation execution time for each algorithm level is given for one LP per node in Tables 3.13 and 3.14, and two LPs per node in Table 3.15.

Table 3.13. Ratio of Simulation Execution Time by Algorithm Level (1 LP per Node, no spin)

Algorithm Level	No Spin		Overall Mean (t_μ)
	FDBK	NOFDBK	
LP_MAN	0.265	0.301	0.283
FILTER	0.299	0.311	0.305
NODE	0.436	0.387	0.412

Table 3.14. Ratio of Simulation Execution Time by Algorithm Level (1 LP per Node, variable spin)

Algorithm Level	Spin (1/5/1)		Spin (1/10/1)		Spin (1/20/1)		Overall Mean (t_μ)
	FDBK	NOFDBK	FDBK	NOFDBK	FDBK	NOFDBK	
LP_MAN	0.323	0.321	0.344	0.357	0.377	0.390	0.352
FILTER	0.274	0.277	0.252	0.246	0.215	0.215	0.247
NODE	0.403	0.402	0.404	0.396	0.409	0.395	0.402

Table 3.15. Ratio of Simulation Execution Time by Algorithm Level, 2 LPs per Node

Algorithm Level	No Spin		Spin (1/10/1)		Overall Mean (t_μ)
	FDBK	NOFDBK	FDBK	NOFDBK	
LP_MAN	0.419	0.287	0.292	0.205	0.301
FILTER	0.274	0.329	0.318	0.207	0.282
NODE	0.306	0.385	0.390	0.588	0.417

The total potential for simulation speedup for each configuration, S_{pot}, was then calculated using the equation:

$$S_{pot} = \frac{1}{\sum_{i=1}^{3} \frac{t_{\mu_i}}{S_{l_i}}} \tag{3.3}$$

The resulting potentials for simulation speedup are:

- 1 LP per node, no spin = 3.97
- 1 LP per node, variable spin = 2.13
- 2 LP per node, combined = 19.99

3.6 Summary

The Intel iPSC/2 hypercube used to analyze a general discrete event simulation showed adequate, yet not optimum performance. Idle processor time, resulting from communication overhead during processor message passing, is clearly an area where improvement is needed.

The potential for simulation speedup was calculated as a function of average execution time, for each level of the simulation algorithm, and the functions, within those levels, that required the largest portion of that execution time. The potential for speedup is clearly less than desired, ranging from two times for one logical process per node, to nearly four times when two logical processes are executed per node.

The greater potential for speedup, exhibited with more logical processes operating per node, is proportional to the increase in processor communication load and its associated idle time. Varying the processor computation load had little effect on which functions required the greatest portion of simulation execution time. The communications required of the message intensive conservative synchronization protocol remained significant regardless of processor workload.

Better utilization of idle processor time, associated with sending and receiving messages, is clearly a viable approach to accelerating the execution of discrete event simulations on the Intel iPSC/2 hypercube.

IV. DES Coprocessor Design

4.1 Introduction

When considering hardware acceleration options for the execution of discrete event simulations on a multiprocessor architecture, two approaches are possible. The parallel architecture may be viewed from a system level and alternatives for improving the efficiency of the system (i.e., memory bandwidth, interconnection networks, etc.) may be considered. An alternative approach is to consider the individual processors, which make up the parallel system, and consider hardware alternatives for improving processor efficiency and utilization, thereby improving the overall system performance.

The parallel architecture considered in this thesis was the Intel iPSC/2 hypercube. This second generation hypercube incorporates several advances over the earlier iPSC/1 which improve the architecture's performance – primarily the more powerful 80386, 32-bit microprocessor, and the enhanced circuit-switching internode communications provided by the direct connect modules (17:1831).

Currently, efforts are underway at Intel to improve the system performance via communication modules that implement a mesh interconnection network, to supplement to the existing hypercube configuration(27). Intel's research efforts will undoubtedly improve the overall performance of the parallel architecture for a wide variety of applications. This thesis, however, focuses on the specific application, Discrete Event Simulation (DES), whose potential for accelerated execution is improved by focusing on the hardware implementation of the primary DES algorithm functions.

With the goal of improved performance for DES, the design approach and requirements for such an application specific accelerator are described in this chapter. This design focuses on the individual processor level rather than the larger parallel system, yet remains within the constraints of the hypercube architecture. Although higher level system improvements may be possible, the degree of improvement for a specific application, such as discrete event simulation, would be overshadowed by more general and widely applicable enhancements.

4.2 Accelerator System Requirements

Bottlenecks to the efficient execution of Discrete Event Simulations on the *i*PSC/2 hypercube are clearly evident. Although the carwash simulation analyzed in Chapter 3 is but a single model, the use of artificially induced workloads and different logical process mappings provided an indication of general DES execution performance on the hypercube architecture.

The conservative synchronization protocol in use requires continual communication between logical processes, and therefore requires an efficient interconnection network. Regardless of the communications efficiency, some processor idle time is expected, as logical processes must wait for messages at various points throughout the simulation.

The interconnection network of the *i*PSC/2, incorporating the direct connect modules, is being upgraded by Intel. Additionally, because of corporate proprietary restrictions (27), the lack of available hardware documentation makes the independent upgrade this of system virtually impossible without considerable reverse engineering.

4.2.1 Processor Utilization While the independent improvement of the hypercube's communication network is unlikely, minimizing processor idle time during communications is possible. Increasing processor utilization is a paramount concern, since discrete event simulations spend as much as 50 percent of the execution time receiving or sending messages (see Tables 3.8 and 3.9).

Processor idle time, associated with communication overhead, may be reduced by incorporating a coprocessor to handle communication, thus freeing the processor for operations that directly support of the simulation. A discrete event simulation coprocessor, providing support for simulation message passing, would enhance nearly every major simulation task. Initializing the simulation requires the receipt and dissemination of logical process information. Posting incoming messages requires monitoring, receiving, and maintaining status on incoming channels. Posting outgoing events involves sending both event and null messages. The last major function, getting events, while not directly a communications function, involves events that were previously received and posted via a communication process. Relieving the processor of these synchronization protocol functions allows

redirection of the processing power to simulation execution and ultimately to accelerated simulation execution times.

4.2.2 Memory Management The distributed parallel architecture of the iPSC/2 was studied to investigate the requirements for a DES hardware accelerator. This platform incorporates technology and an architectural design that limit the enhancement options available for a DES accelerator. The use of an Intel 80386 microprocessor, with its integrated segmentation and paging memory management unit (MMU), reduces the likelihood of accelerating performance through improved hardware for memory management.

The on-chip MMU performs all virtual-to-physical address translations, segmentation, and paging violation checking. The MMU uses pipelining and parallel execution to generate physical addresses by storing the translation, segment, and page descriptors on chip. Additionally, the use of a translation look-aside buffer (TLB) significantly reduces paging translation, which is otherwise performed through a two-step table lookup. Although the TLB has a high hit ratio of 98 percent, the paging unit is supplemented with special purpose hardware capable of performing a page translation in nine clock cycles (11:18-21). Therefore, as with the hypercube's interconnection network, the potential to improve performance via improved memory management is small.

4.3 Design Approach

The design of a coprocessor specifically for distributed discrete event simulations using the conservative synchronization protocol has yet to be documented in the literature. Therefore, this initial design takes a rather abstract, chip-level, approach to defining the requirements necessary to implement such an application specific coprocessor.

Using a chip-level approach, the coprocessor requirements were defined in terms of input/output (I/O) response and the algorithm that the chip implements(2:5). The necessary operations of the Discrete Event Simulation (DES) coprocessor were defined using a behavioral description of the coprocessor's functional components. The IEEE standard, VHSIC Hardware Description Language (VHDL), was used as the design tool to implement this behavioral description.

4.3.1 Hardware Implementation of DES Algorithm Based on the significant communications overhead and subsequent idle processor time associated with the DES algorithm, virtually every portion of the simulation algorithm exhibits some potential for acceleration through a hardware implementation. From Figures 3.5 and 3.6 it is clear that the primary functions at both the logical process manager and the filter levels require communications interaction through subfunction calls.

The potential for speedup addressed in Section 3.5.4 is realizable if the processor has actual work pending. Pending jobs could receive processor time in lieu of the processor remaining idle while the currently scheduled job waits for communications. Hence the purpose of this design is to implement the DES algorithm in hardware and thus relieve the processor from the administrative overhead associated with the conservative synchronization protocol as it exists in the SPECTRUM testbed. The hardware coprocessor will provide processing for incoming messages, schedule and package outgoing messages, manage simulation time, monitor communication arc status and the next event list, and provide event inputs to the processor when requested. The coprocessor will support the simulation execution for all logical processes on the node, transparent to the processor's operation and, if a sufficient workload is available, increase overall processor utilization.

4.3.2 Process Model As a supplemental, application-specific system for the Intel 80386 microprocessor, the DES coprocessor was viewed as a finite state machine. State transitions are controlled by the processor when needed and follow the process model graph of Figure 4.1.

Primarily an I/O device, the coprocessor responds to control signals from the processor. Figure 4.1 shows the three necessary coprocessor states: start, cpu_io, and execute.

With power applied and no processor request pending, the coprocessor remains in the start, or idle state. Basic housekeeping functions, which consist primarily of maintaining the coprocessor's local memory, are performed while waiting for processor requests.

Transition to the cpu_io state is controlled by the processor. The coprocessor, which resides in the processor's I/O address space, monitors the CPU control signals and the system data bus for opcodes and operands when activated by the processor.

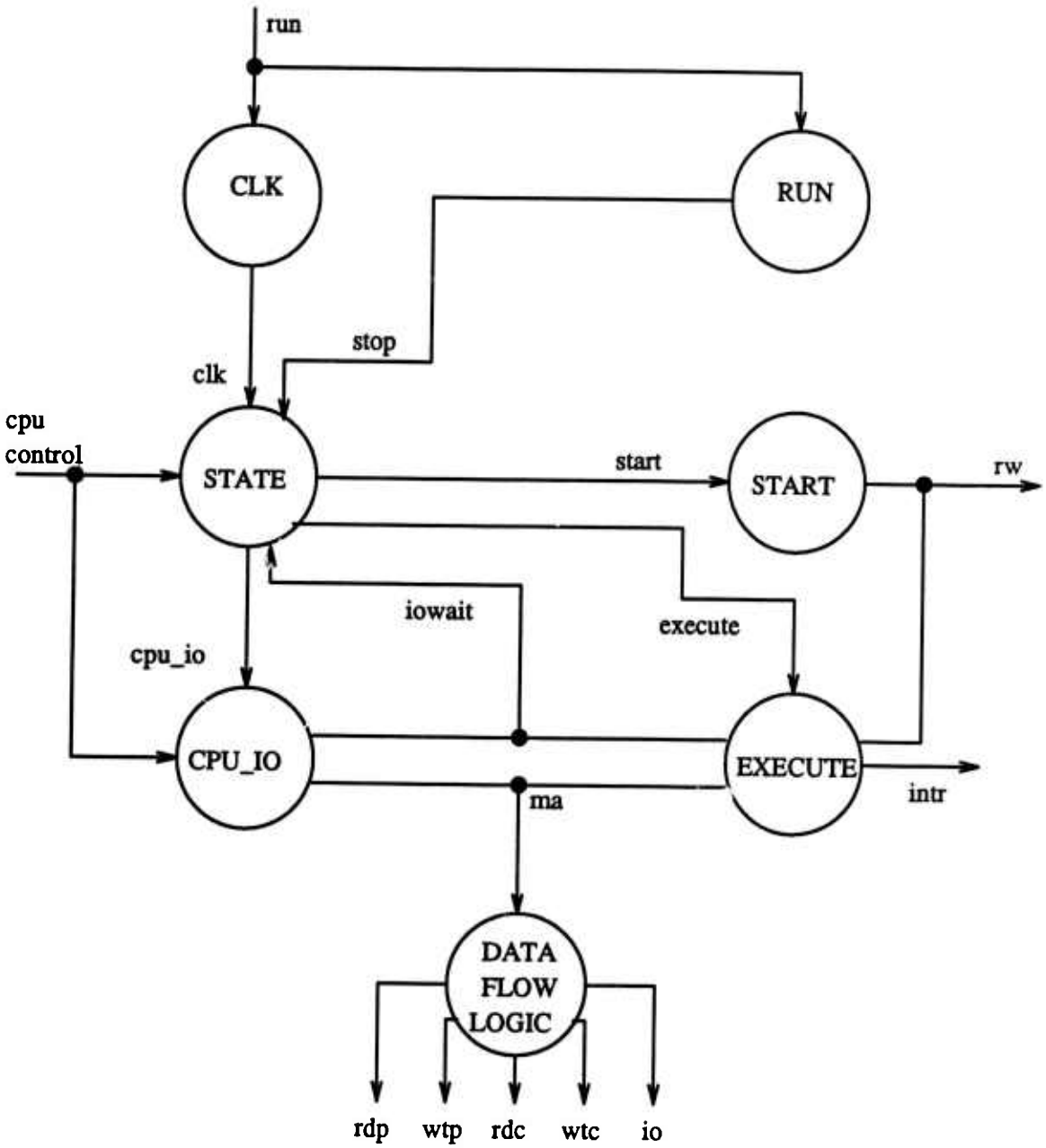

Figure 4.1. Process Model Graph

The execution state is entered at the completion of cpu_io. All necessary DES functions are executed while in this state. Data flow within the coprocessor is controlled primarily from this state; however, control of parallel input and writing to local Random Access Memory (RAM) from the cpu_io state are also supported.

The clock and run processes are included for completeness. Coprocessor timing is referenced to the main processor's system clock, while the run process is analogous to chip enable or connection of primary drive voltage. Similar to the Intel 80386 microprocessor, the DES coprocessor is designed to use an internally generated clock at half the system clock frequency for instruction executions (19:5-347).

4.3.3 DES Coprocessor Interface The DES coprocessor relies on the Intel 80386 microprocessor for controlling state transitions and data transfers. Similar to the Intel 80387 DX math coprocessor, this application-specific coprocessor interfaces to the system bus architecture for all data transfers and is connected directly to the CPU control signals (19:5-441,447).

Control of the coprocessor relies on two-way communication between the CPU and the coprocessor. Operating in the processor's addressable I/O space, the coprocessor is controlled via two address lines (A15 and A2) and the M_IO line from the processor. The choice of a specific address location for the coprocessor was arbitrary but falls within the general requirements specified for the Intel 80386.

The Intel 80386 can address 16K, 32-bit ports in its I/O address space, 0H--FFFFH. Intel has reserved addresses 00F8H–00FFH for use with its math coprocessor (80387), which is activated by asserting address line A31 high and toggling address line A2 to distinguish data from opcodes(19:5-308,309). Intel has advised that while I/O addresses are available, the iPSC/2's NX operating system reserves additional I/O addresses that are unique to each system's configuration, and can only be identified through reference to the system documentation which was unavailable at this time(23). Addressing the coprocessor at the chosen I/O port is therefore similar to the Intel 80387, in that asserting address line A15 and M_IO, activates the coprocessor, while data and opcodes are distinguished through address line A2.

Additional control lines are required to provide coprocessor state and to send requests to the CPU. The Intel 80386 monitors a ready input, READY#, to terminate or wait on bus transactions with a bus slave(19:5-349). The coprocessor supports this requirement with a ready signal, READY0, which it asserts low to end bus transfer cycles, as needed by the Intel 80386.

An interrupt request line is also required for the coprocessor interface. Since direct access to the network switching circuitry of the Direct Connect Module (DCM) is not possible, the coprocessor will rely on the Intel 80386 to pass outgoing messages to the DCM. When needed the coprocessor will assert an interrupt request for message output and, if not masked by the CPU, will forward the outgoing message to the DCM.

Similar to the Intel math coprocessor, the DES coprocessor is designed with additional control lines to indicate an active state or an error condition. When active the BUSY signal tells the CPU that the coprocessor is executing an instruction, while the ERROR signal reflects a coprocessor exception.

4.3.4 DES Coprocessor Functional Components A block diagram of the DES coprocessor system is shown in Figure 4.2. The design consists of five functional blocks needed to implement the discrete event simulation using a conservative synchronization protocol and parallel I/O ports to interface with the CPU's system data bus.

As a chip-level design using VHDL behavioral descriptions, detailed hardware specifications are not given in this design. The design objective is to specify the hardware characteristics and behavior necessary to implement the simulation algorithm as it exists in the SPECTRUM testbed, while ensuring compatibility with the Intel 80386 32-bit architecture used on the iPSC/2 hypercube.

4.3.4.1 Parallel Input/Output A parallel interface with the system data bus of the Intel 80386 requires the use of 32-bit wide buffered latches. Using this approach, the coprocessor can take full advantage of doubleword aligned bus transfers and be disconnected via a high impedance state when no bus transaction is active.

The parallel I/O port design closely follows that presented by Armstrong in his VHDL

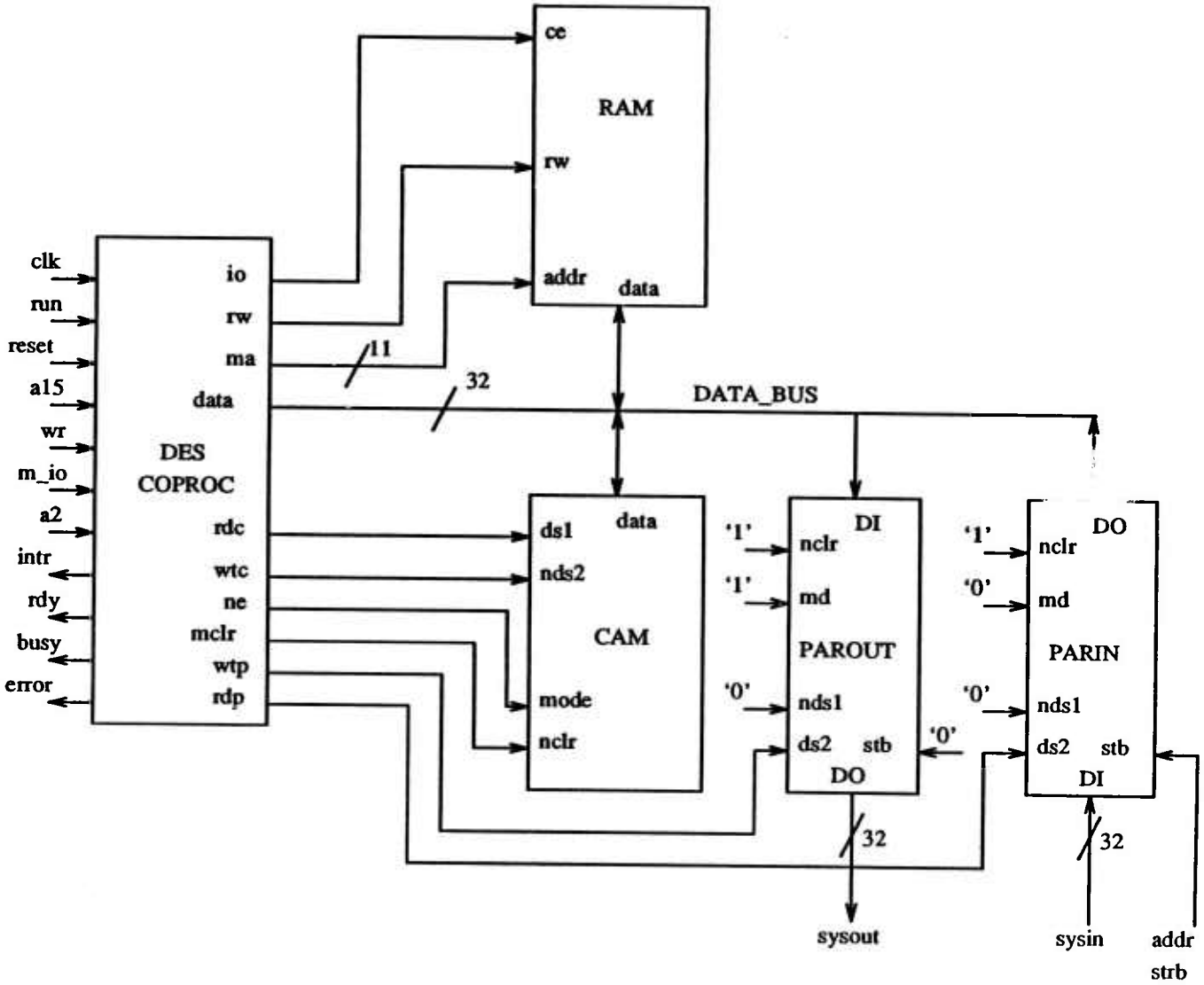

Figure 4.2. DES Coprocessor System

description of the Mark II processor(2:120-123). Referencing Figure 4.2, the parallel input and output blocks are designed with identical behavioral descriptions and the direction of data flow is determined by the I/O port mapping and mode selection for each device.

The respective I/O port is activated by the coprocessor when needed via a device select line, DS2. Assertion of this line is in response to interface signals between the coprocessor and the CPU. Parallel input requires the additional interface of a strobe line, addr_strb#, from the CPU to determine when data is valid on the system bus and thus may be latched.

Interfacing with slave units is not anticipated or required at this time, therefore no provision for an interrupt capability is included in the design of the I/O ports.

4.3.4.2 Random Access Memory The Random Access Memory (RAM) of Figure 4.2 supports several critical functions in the DES coprocessor design. Its primary purpose is to hold simulation information relative to each logical process simulated on the processing node. Additionally, it provides storage space for coprocessor required assembly code and swap space, if needed, to support overflow from the Content Addressable Memory (CAM).

The RAM memory incorporates several fundamental design decisions. The most obvious is the use of doubleword (32-bit) alignment for all memory transactions. This allows a direct interface to the system data bus of the CPU and permits transfer of data with fewer bus cycles. Additionally, the data format currently used by the iPSC/2 for integers, which are the predominant data type used in the simulation analyzed, is a four-byte doubleword. The choice of doubleword alignment also eliminates the need for two address lines, as the distinction between the four individual bytes that makeup the doubleword is no longer necessary.

Determining the size of RAM memory required is based on several assumptions relative to the simulations supported by the DES coprocessor. A portion of RAM is required by each logical process being simulated on the processor and the iPSC/2 operating system allows a maximum of 20 processes per computing node(18:1-37). Therefore the RAM design incorporates a separate partition to store each LP's simulation data. Additionally,

an address pointer table with 20 entries is provided to reference the base address for each LP's partition.

The RAM design is organized as shown in Figure 4.3. Each logical process simulated on a given CPU maintains data identifying the other logical processes on which it is dependent (i.e., must communicate with). Assuming a maximum of ten input and output logical processes are to communicate with each LP on the node, 20 doubleword addresses are available for each LP on the node. The limit of ten inputs/outputs was chosen based on the fan-in and fan-out heuristic for similar electronic devices.

Representation of the unique identities for all input/output LPs within the 20 addresses in coprocessor RAM is contingent on the ability to represent each LP_id with only 32 bits. Each LP is identified by its processing node number and its logical process number on that node, both of which are 32-bit integers on the iPSC/2. The field size needed to represent this information however is significantly less when using an eight node hypercube, limited to 20 LPs per node. Therefore the RAM design will store LP_ids as a single doubleword, with the node number occupying the upper two bytes and the process number contained in the lower two bytes.

In addition to the above memory requirements, each LP must maintain data that reflects the simulation state and supports the conservative synchronization protocol. The local simulation time is unique for each LP and occupies a doubleword within its coprocessor RAM partition. Additionally, a single doubleword location is provided for each LP to maintain the inherent delay time for the LP's simulation process. Storage of the LP's delay time is only necessary for deterministic simulations, where the LP's execution time is constant and can be stored during initialization.

The number of input and output communication channels is also required to monitor incoming and outgoing messages. Similar to the LP_ids, the memory representation for the number of input and output arcs is stored as a single doubleword. The number of inputs occupying the upper two bytes and the number of outputs in the lower two bytes.

The last entry in each LP's RAM partition is a status register containing input message information. As described in Section 2.3.2.2, a message must be received on every

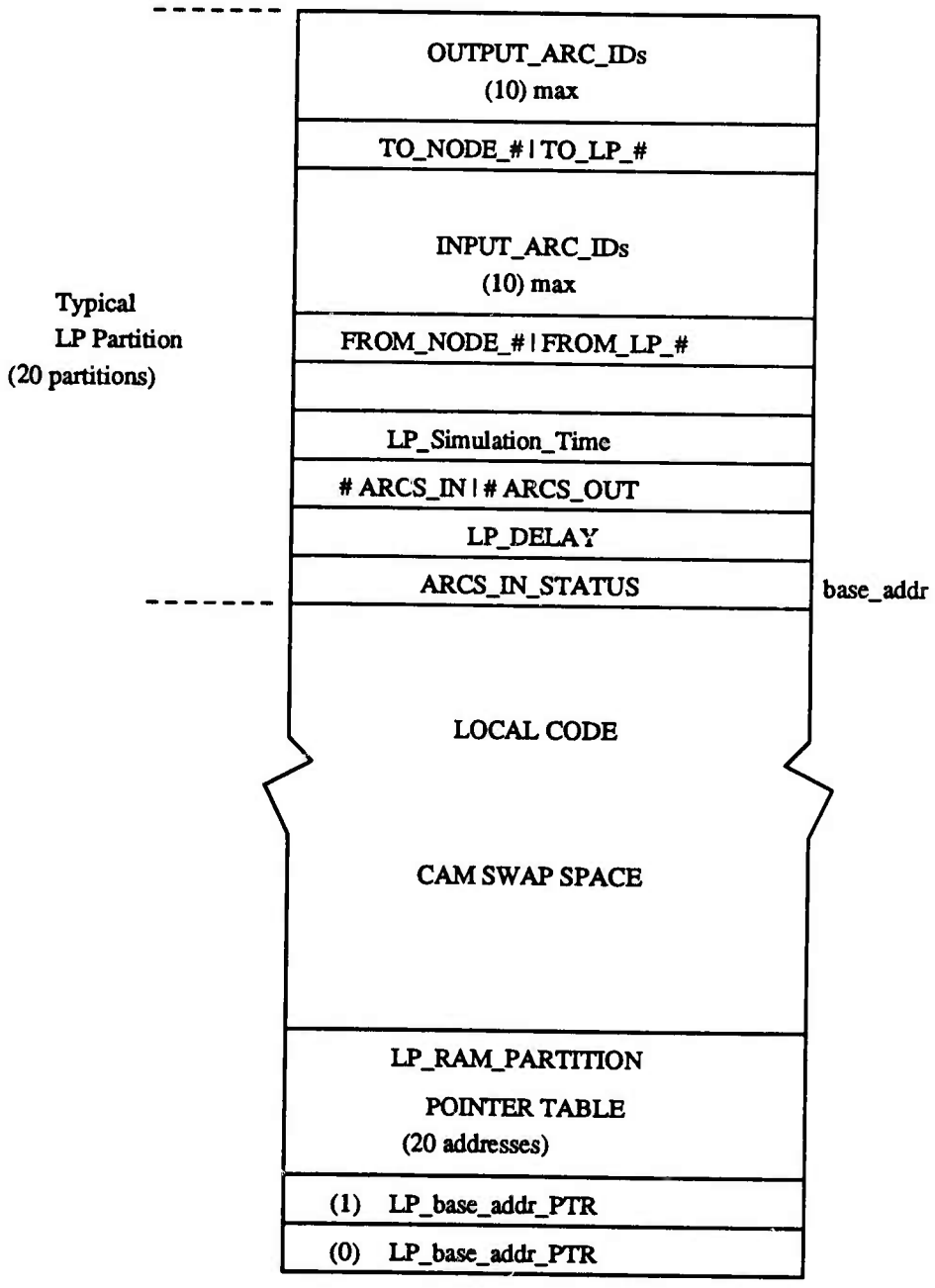

Figure 4.3. Coprocessor RAM Memory Organization

input arc to ensure the next event may be safely executed. The ARCS_IN_STATUS register reflects received messages with a '1' in the bit position that corresponds to the input LP and a '0' otherwise.

To meet the above memory requirements, the DES coprocessor design requires 4Kbytes of dynamic RAM. This total is obtained by considering that a minimum of 2Kbytes, $(20\frac{LPs}{node})(25\frac{Dwords}{LP})(4\frac{bytes}{Dword})$, are need to store simulation data and a working buffer of 2Kbytes is anticipated for CAM overflow and local code storage.

4.3.4.3 Content Addressable Memory A Content Addressable Memory (CAM) is included in the DES coprocessor design to store and maintain the next event lists for each logical process executing on the node. Although the effects of next event list management were not significant in the car wash simulation analyzed, the number of events stored requires a time complexity of $O(n)$ when a sequential algorithm is used for an in-order retrieval. The CAM provides the ability to perform searches of memory in parallel and thus reduce the time complexity to O(1).

As with the RAM memory, the size of CAM required must be justified in the design. Unlike the RAM however, the CAM requires data storage in compliance with a strict format in order to operate effectively.

The CAM size determination and searching process are defined in terms of how each simulation event is stored in the CAM. Figure 4.4 shows an event list entry in the CAM. Although some encoding of event information will be done by the DES coprocessor, a total of 78 bits is necessary to completely define each event on the next event list.

The valid bit is used by the CAM to determine the status of each entry (i.e., if it's been used). The remaining fields uniquely identify each event. The "TO_LP" field identifies which LP on the node is to receive that event. This field requires five bits to uniquely identify which of the 20 possible LPs receives the event. The "FROM_NODE" and "FROM_LP" fields, which uniquely identify the sending LP, are also encoded by the DES coprocessor. The "FROM_NODE" is limited to eight possible nodes on the *i*PSC/2, hence it requires only 3 bits. Similarly, the "FROM_LP" is one of 20 possible LPs on that node and is therefore represented with five bits.

The "TIME_TAG" and "MEMR_PTR" are generated by the iPSC/2 and remain in their original 32-bit format. The "MEMR_PTR" must remain unchanged as the CPU requires this memory reference to locate the actual event and its associated data structures, which are maintained in CPU local memory.

VALID	TO_LP	FROM_NODE	FROM_LP	TIME_TAG	MEMR_PTR
1 bit	5 bits	3 bits	5 bits	32 bits	32 bits

Figure 4.4. CAM Event Field Entries

The size of CAM required for next event storage will vary as a function of the physical process being simulated and the number of events generated during the simulation. CAM size for a general purpose DES coprocessor is based on the following assumptions and median values:

- median value: 10 LPs/node
- median value: 5 input arcs/LP
- assume: 10 events/input pending

Given these median values and assuming an average of 10 events are pending at each node, the number of events stored in the CAM is calculated as:

$$(10\frac{LP}{node})(5\frac{inputs}{LP})(10\frac{events}{input}) = 500\frac{events}{node}$$

Based on the storage requirement of 500 events/node, a CAM of 4Kbytes minimum is needed, since each event requires 78 bits.

3.4.4 DES Coprocessor The DES coprocessor's operation follows the basic approach envisioned by von Neumann and described by Hayes(16:179-183). A small single-address instruction set and a minimal number of registers are used in a fetch, decode, execute, and store sequence, that is initiated and controlled by the CPU.

The operation of the DES coprocessor is depicted in Figure 4.5. The coprocessor remains idle until activated with an instruction from the CPU. The instruction operands

are sent to the coprocessor along with the opcode during an initial fetch cycle, which occurs in the cpu_io state. Instruction decoding is performed at the start of the execution state. Instructions requiring additional operands must repeat a fetching operation, however this is done from local coprocessor RAM where unique logical process information (i.e. simulation time, number of inputs/outputs, etc.) received from the CFU during initialization is stored.

The execution of each instruction requires of both combinational and sequential logic functions within the DES coprocessor. In addition to instruction and accumulator registers, the DES coprocessor design incorporates a 32-bit flag register and ten general purpose registers to support these operations. The primary role of the flag register is to monitor the DES coprocessor's memory status. The flag fields reflect whether the CAM is full or not and the number of events temporarily stored in RAM.

Several functions of an Arithmetic Logic Unit (ALU) are also performed by the DES coprocessor. A counter function for incrementing and decrementing register values is included to maintain the status of input message channels and to monitor the sending of messages to output channels. Additionally, the ability to mask specific bit fields is provided for combining of multiple data (i.e., LP node and LP number) into single 32-bit fields to limit the storage requirements and reduce the number of bus cycles needed to transfer information.

4.4 Design Implementation

The DES coprocessor system design is implemented using a VHDL behavioral description for each of the functional blocks shown in Figure 4.2. Each block in the design is described using one or more VHDL processes. Functions and procedures that are performed multiple times with a given process are incorporated within the package body for the given functional block. Those functions that are global in nature (i.e., required by more than one functional block) are incorporated in the system package. A complete listing of the VHDL source code for the DES coprocessor system is included in the appendices.

4.4.1 System Packages The system packages define types, constants, and functions that are required by all functional blocks of the DES coprocessor design. Multi-valued logic

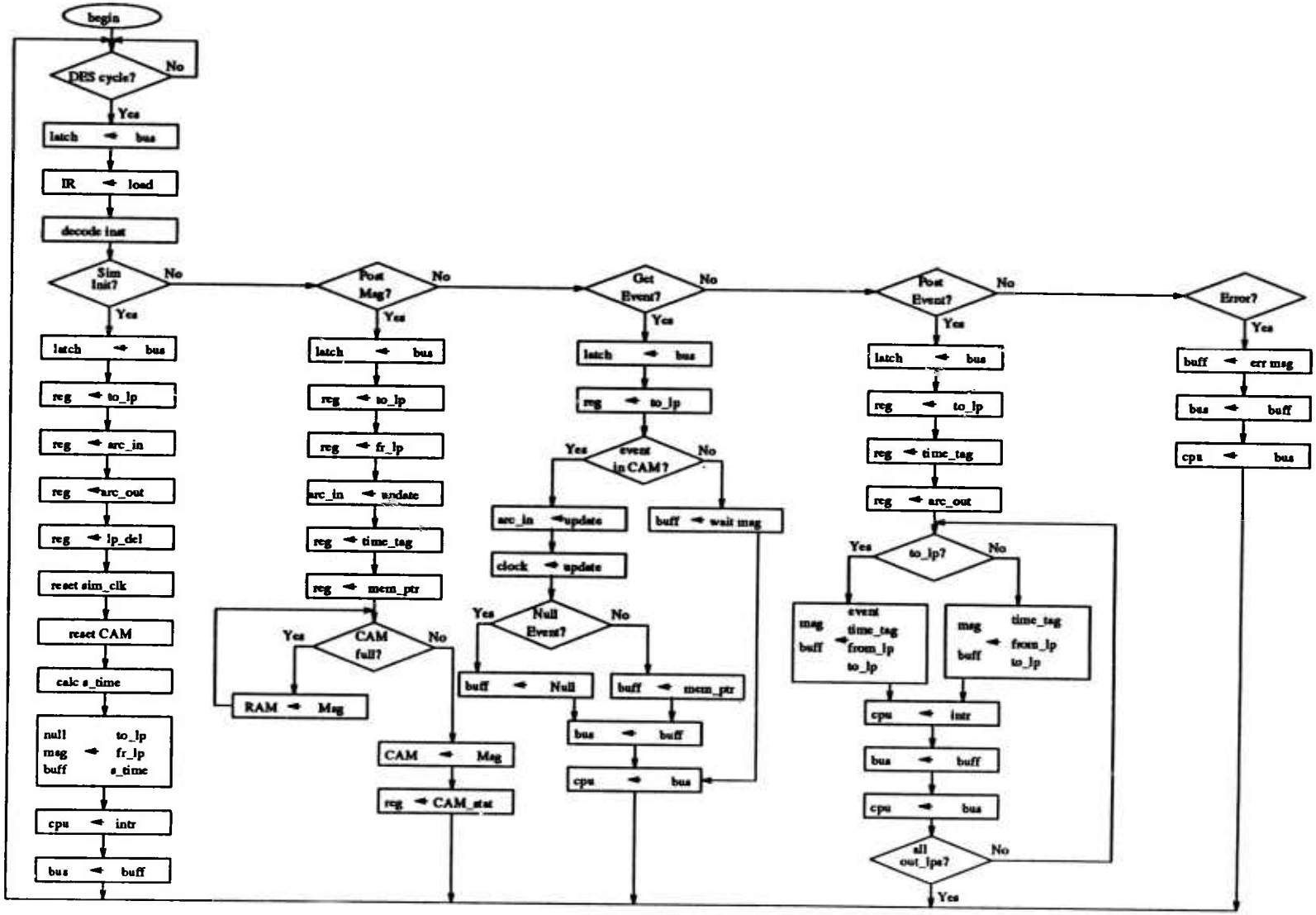

Figure 4.5. Operation of the DES Coprocessor

(i.e., MVL7), which is a standard type included with the Zycad VHDL system, was chosen to represent signals within the DES design, as this more closely reflects actual signal values(32:10:17,18).

Resolution of the DES coprocessor data bus is necessary as there are five possible drivers for this bus (see Figure 4.2). The bus resolution function is a variation of the *wiredX* function provided with Zycad VHDL(32:10:84-90). The DES coprocessor design differs from the original Zycad by using function and argument names that reflect the DES coprocessor's operation.

The need for type conversions is inherent with the VHDL design language. The "DWORD" subtype, which is a resolved signal, is included to avoid unnecessary type conversions for signals connecting the functional blocks. Several functions contained in the system package (see Appendix A) are provided to support the type conversions needed by the various processes.

4.4.2 DES Coprocessor Behavior The behavioral description for the DES coprocessor is the largest functional body of the overall system design. It incorporates the remaining package bodies and relies on the functions they provide to specify the total DES coprocessor design.

The coprocessor's behavior (see Appendix B.2) is implemented with five separate processes: run, state, start, cpu_io, and execute. The *run* process is included as a primary system driver and is analogous to a chip select or drive voltage being applied to the coprocessor chip. The inclusion of the run process directly supports the testbench used to verify the coprocessor design.

The state process provides the mechanism for state transitions as shown in Figure 4.1. State transitions are possible during both positive and negative clock transitions. The progression between states is somewhat sequential in that the coprocessor must perform an I/O state with the CPU prior to entering the execute state. The sequential requirements of the state transitions are implemented by evaluating the CPU control signal values at each clock transition.

4.4.2.1 Start Process The *start* process represents the DES coprocessor's state when not executing a CPU instruction. Required maintenance of the coprocessor's CAM takes place while in this state. The maintenance activity is based on the simplifying assumption that events will arrive in time sequence after the CAM overflows, thereby allowing an ordered storage in the DES coprocessor's RAM.

The DES flag register is checked for CAM overflow status. If the CAM has experienced an overflow, the presence and number of overflowed events (i.e., events temporarily in RAM) is detected in this register.

The overflowed events temporarily in RAM are stored in a first-in-first-out (FIFO) queue. As simulation events are executed, space becomes available in the CAM and the flag register is updated. During the start process, if an event is in the RAM, and CAM space is available, the events are taken from the head of the FIFO queue and passed to the CAM for storage.

4.4.2.2 CPU_IO Process The *cpu_io* process handles all incoming bus transfers from the CPU in an asynchronous cycle regulated by the system clock and the coprocessor's assertion of the ready line low. Additionally, this process monitors the status of CPU control signals to determine if the bus transfer is an instruction (i.e., A2 = '0') or operand (i.e., A2 = '1').

During the cpu_io process, CPU bus transfers are received from the parallel input port through a buffer register. Opcodes are routed to the instruction register while operands are stored sequentially in internal, general purpose, 32-bit registers or the coprocessor RAM, depending on the opcode to execute.

Additional operands, that are unique to the executing LP, are required for the initialize simulation opcode. Using the LP's process number (i.e., 0 - 19 from register 1) to index the RAM partition pointer table, the base address for storing additional operands in memory is retrieved.

The saving of additional operands, when required, is done during the CPU bus cycle. The CPU waits for the ready low assertion while the cpu_io process stores operands at a sequential offset from the LP's RAM partition pointer.

Anticipated hardware timing delays were specified for required register transfers, memory access and read/write times, and the generation of output control signals to the CPU. As an asynchronous process, the CPU is required to wait for the coprocessor to initiate termination of the bus transfer cycle. Hence, the timing delays specified (see Appendix A) are somewhat arbitrary, while attempting to satisfy the Intel 80386 requirement of two clock cycles per bus transfer (19:5-358).

At completion of this process the DES coprocessor has all necessary operands, either in general purpose registers or both RAM and registers, along with the opcode to execute in the instruction register.

4.4.2.3 Execute Process The *execute* process is entered immediately after every cpu_io cycle. The instruction register is decoded for one of four possible simulation instructions: initialize simulation, post message, get next event, or post an event. Once decoded, the appropriate procedure is called for execution.

The execution of simulation functions follows the DES coprocessor operation outlined in Figure 4.5. VHDL procedure calls, within the execute process, are used to implement each of the required simulation functions.

4.4.2.4 Initialize Simulation Procedure A new simulation is initialized with the *init_sim* procedure. This procedure uses essential LP data (i.e., to_lp, lp_delay, number of inputs and outputs) stored in general purpose registers one through four, and additional operands (i.e., input and output LP node and process numbers) to initialize the simulation.

The local simulation clock is reset and the minimum safe time is calculated. The CAM is cleared of previous entries as is the corresponding flag register.

Null messages identifying the sending LP and containing the "TO_LP" address and minimum safe time are routed to the CPU for output via an interrupt request procedure. Null messages are sent to every output-arc LP identified in the local RAM partition. After all null messages have been sent, the *init_sim* procedure saves LP essential data at the appropriate RAM location and a state transition occurs at the next system clock transition.

4.4.2.5 Post Message Procedure The *post_msg* procedure supports the CPU's requirement to receive both event and null messages during the discrete event simulation. The incoming message is received in three or four bus cycles from the CPU, depending on the type of message. Since null messages have no *real* event associated with them, no pointer to the event storage address in CPU memory is associated with this message type.

Simulation data, essential to the receiving LP (i.e., number of in/out arcs, arcs-in status, simulation time, and lp delay), are loaded from the RAM partition into general purpose registers. The arcs-in status register is then updated to reflect the receipt of an input message for the *from_lp* arc.

CAM full status is checked, via the flag register, and the received message is routed either to the CAM or RAM swap space for saving. Prior to actual storage of the message, some compression of the message fields, as described in Section 4.3.4.3, is performed to conserve memory storage space and minimize the number of memory write cycles.

4.4.2.6 Get Event Procedure The *get_event* procedure performs the reverse operation of the post message procedure. Again the essential data for the executing LP are loaded from RAM and the arcs-in status register is checked to determine if an event is ready for the CPU or if a wait for event message must be sent.

If an event is available for execution, the coprocessor retrieves it by asserting a CAM read along with the to_lp identifier for next event searching. The next event from the CAM is then routed to the CPU for execution, while the coprocessor updates the LP's arcs-in register and local simulation clock. Additionally, the CAM returns an extra bit with the next scheduled event which indicates if additional events from the same input arc remain in the CAM for later execution.

4.4.2.7 Post Event Procedure The *post_event* procedure is executed after the CPU has processed an event action and the subsequent output message is ready for transmission. This procedure builds the event message for the designated to_lp(s) and returns the message to the CPU for sending via the DCM module. In addition, this procedure constructs a null message containing the LP's identifier and the new safe look ahead time

which is routed via the DCM module to all remaining output arcs.

4.4.2.8 Signal Multiplexing The VHDL behavioral design for the DES coprocessor requires the use of several processes, often requiring access to the same data (i.e., general purpose registers) or status signals (i.e., the coprocessor ready0 line). The need for additional resolution functions was avoided by using a signal multiplexing technique described by Armstrong (2:88,89). This approach permits multiple processes to drive signal 'X' while the actual value assigned is that which is most recently applied (i.e., not signalX'quiet).

4.4.3 Parallel I/O Behavior The behavior of the parallel I/O ports is borrowed from Armstrong (2:120-123), and only slightly modified for this design. The behavior of a buffered latch is implemented in this design. Control signals from the coprocessor and the CPU are used to activate combinational logic and 'D' flip-flops, providing a bus latching capability as well as high impedance disconnect from the bus when not enabled.

Output from the parallel I/O ports uses a guarded block construct, which is dependent on the device select and the clear latch inputs. Unlike Armstrong's design, the parallel I/O ports of the DES coprocessor system lack an interrupt capability as it is not necessary in the coprocessor design. Constants defining inherent hardware timing delays are included with the generic map (see Appendix A) and are based on bus transfer timing requirements of the Intel 80386.

4.4.4 RAM Memory Behavior Enabled)

The behavior of RAM memory is defined by a memory model process. Two procedures, (do_read and do_write), are called to perform the basic memory functions. The memory functions are activated with control lines for I/O and read/write from the DES coprocessor while the memory location is valid on the local address bus.

The RAM organization is shown in Figure 4.3, and this data structure is implemented as an array with 1K entries, each of which is a 32-bit array representing a memory location. The RAM design incorporates a text_io read operation to load the LP base address pointer

table during component initialization. Therefore, when the VHDL design is executed, the RAM is preloaded for the necessary memory operations.

An assertion statement is included as a safeguard with the RAM design to indicate if an address changes during memory read/write operations. Similar to other functional blocks that drive the DES coprocessor data bus, the RAM defaults to a high impedance state when no memory transactions are active.

Constant hardware time delays for memory access, reads, and writes are defined in the DES system package of Appendix A and are established to provide a timing basis for functional block interaction. The timing delays were again chosen arbitrarily, as actual values depend on the both technology employed for the hardware implementation and whether or not the RAM can be included within the DES coprocessor chip package.

4.4.5 CAM Memory Behavior The Content Addressable Memory behavior is also implemented with a VHDL process construct. No requirement for an address bus was anticipated nor included in the design, as operations involving specific addresses are not initiated by the DES coprocessor.

The CAM event storage fields shown in Figure 4.4 provide a reference for the read operation. The CAM algorithm to search for the next scheduled event is implemented with a sequential loop operation which differs from the hardware implementation which will be performed for all memory locations in parallel.

A read hit is guaranteed as the coprocessor checks the LP's arcs-in status register prior to requesting the next event. The search process is performed on all valid memory locations that match the requesting LP's id. The next scheduled event is determined through a less than comparison of the time field of all valid events that match the requesting LP's id.

Once located the next event is parsed into three portions (i.e., to/from identifier, time tag, and memory pointer) for bus transfer to the DES coprocessor. The most significant bit of the first transfer is set high ('1') if additional events from the same sending LP remain in the CAM for future scheduling. This information allows the DES coprocessor to update the executing LP's arcs-in status register.

After sending the next event to the DES coprocessor, the CAM process clears the next event's valid bit, to allow storage of a received event at that location. Similarly, the DES coprocessor can update the flag register, if necessary, to indicate available storage space in the CAM.

CAM write operations are performed in a straight forward fashion. The DES coprocessor writes events to the CAM only when space is available, based on flag register status. The received event is inserted in the first available storage location as determined by the CAM's sequential (i.e., parallel in hardware) search of the event fields' valid bits.

4.5 Summary

The design of a DES hardware accelerator was based on the requirements of individual processors rather than the overall parallel system of the Intel iPSC/2 hypercube. Bottlenecks to the efficient execution of conservative discrete event simulations were found to result in significant CPU idle time. Therefore, the purpose of a DES coprocessor is to free the CPU from the overhead of executing discrete event simulation functions and allow pending jobs immediate access to CPU execution.

Operation of the DES coprocessor is controlled by the CPU when requested. It functions as a finite state machine performing I/O with the CPU and executing the basic functions of simulation initialization, posting events, getting the next scheduled event, and posting output messages when generated.

The conservative synchronization protocol is implemented by the DES coprocessor, hence the overhead of simulation time management, next event list maintenance, and input/output message traffic are transparent to the CPU.

The requirements for the DES coprocessor operation are specified in VHDL. This behavioral description of the DES coprocessor is implemented with process constructs and procedures necessary to support the conservative synchronization protocol for general discrete event simulations.

The inclusion of timing delays for hardware modeling is somewhat arbitrary. The timing constraints of the Intel 80386 provided the basis for selecting these timing values.

Actual timing values will, of course, depend on the technology employed for a hardware implementation and the package size available for the coprocessor functional blocks.

V. DES Coprocessor Design Test

5.1 Introduction

A high-level system description and the required functional behavior of a Discrete Event Simulation coprocessor were implemented using the VHSIC Hardware Description and Design Language (VHDL). A complete source code listing of this behavioral description is included in the appendices.

Testing modes for the DES coprocessor design were limited since a VHDL design for the Intel iPSC/2 and the Intel 80386 were not available. Hence, a complete system test of the hypercube architecture incorporating the DES coprocessor was not feasible, nor was a comprehensive integration test with the Intel 80386 microprocessor possible.

A VHDL test bench was designed to simulate the interface of the Intel 80386 microprocessor and the DES coprocessor on a single hypercube processing node. Testing centered on verifying the DES coprocessor's implementation of the conservative synchronization protocol, discrete event, simulation algorithm. The Intel 80386 microprocessor was modeled as a system driver addressing the DES coprocessor as an I/O port. A detailed behavior of the Intel 80386 was not necessary as only interface control signals and bus transactions were required to verify the DES coprocessor's operation.

A secondary purpose of testing the DES coprocessor design was to analyze the CPU/DES coprocessor interface. The DES coprocessor was designed with an asynchronous interface to the CPU, hence a timing analysis of device signals during data transfers was performed.

The following sections describe the test methodology and DES coprocessor test configurations used to analyze the coprocessor design. Simulation test data that verify the DES coprocessor's implementation of the discrete event simulation algorithm are presented. A summary of the test data along with logic analyzer traces are included for analysis of the CPU/DES coprocessor interface and coprocessor performance evaluation.

5.2 Design Test Methodology

The DES coprocessor design is composed of functional blocks as shown in Figure 4.2. The behavior of each block was designed to support the implementation of the DES coprocessor operations shown in Figure 4.5. The design was tested to verify the implementation of a general discrete event simulation algorithm using the Chandy-Misra conservative synchronization protocol.

Unit testing of the functional components was performed prior to overall system testing. The DES coprocessor component required support from all other functional components in the design, hence system testing encompassed the unit testing for the DES coprocessor.

Implementation of the basic discrete event simulation functions of initialize, post-message, get-next-event, advance-time, and post-event was verified during the system testing. Additionally, DES coprocessor performance, in terms of execution time, and the CPU/DES coprocessor interface were analyzed during system testing.

The DES coprocessor system testing was conducted using the carwash simulation configurations outlined in Section 3.2.2. A VHDL testbench (see Figure 5.1) was designed to test the DES coprocessor operation on a single hypercube processing node. The DES coprocessor was activated and exercised by a testbench driver representing an Intel 80386 microprocessor. The DES coprocessor was addressed within the testbench as an I/O port connected to the microprocessor. The detailed behavior of the microprocessor was not necessary as the coprocessor's implementation of a general discrete event simulation algorithm could be verified by tracking the coprocessor state, values of internal variables, and bus transfers with the CPU.

Testbench timing was provided by a system clock, similar to the system clock required for actual hardware operation. The testbench clock was operated at a frequency half that of the actual Intel 80386 system clock on the iPSC/2. The testbench clock therefore represented the internal clock frequencies of the Intel 80386(19:5-347) and the DES coprocessor.

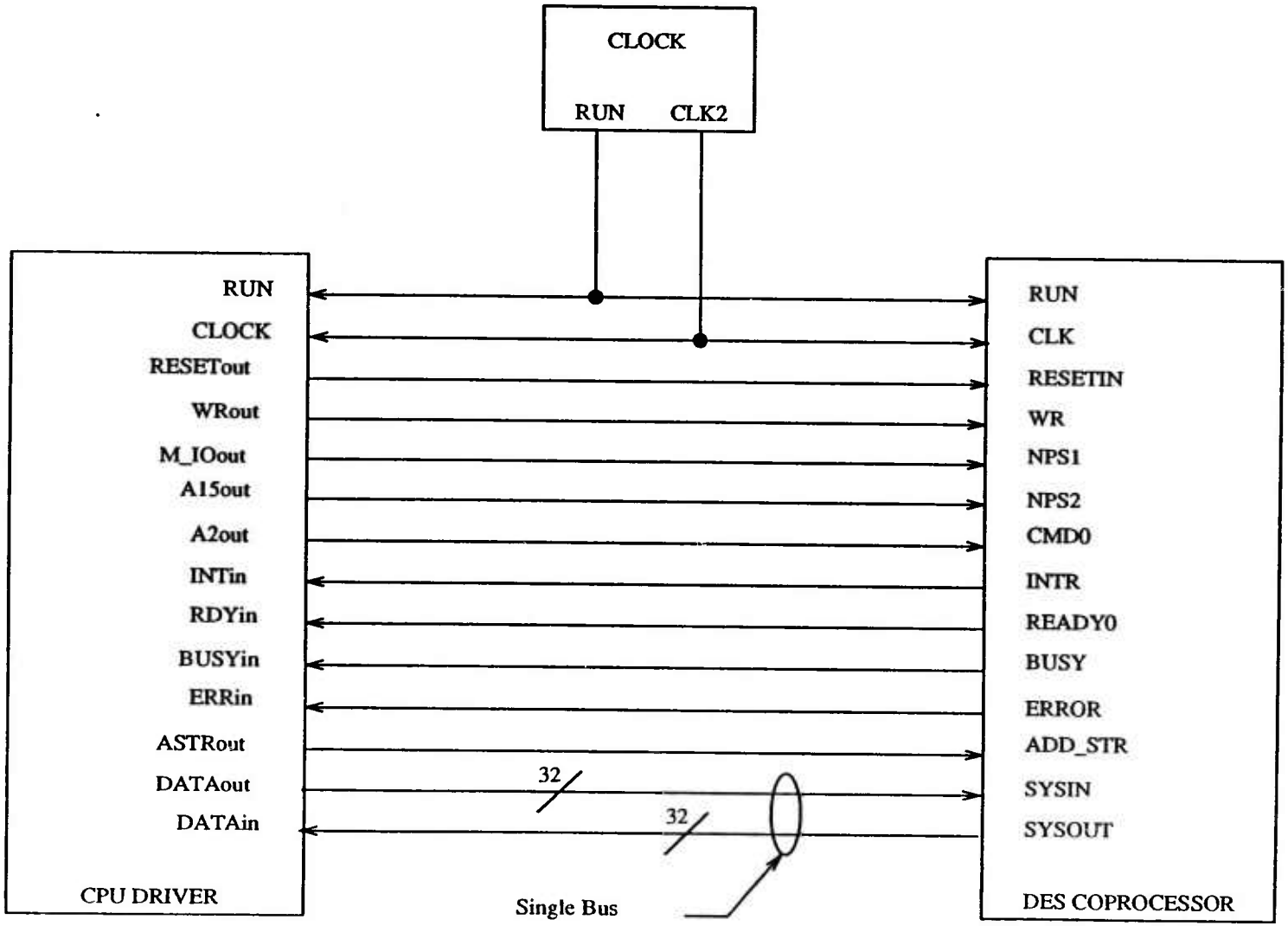

Figure 5.1. DES Coprocessor Testbench

Once activated, all testbench signals and variables were available for monitoring with the Zycad VHDL system. By selectively monitoring these variables and signals the DES system's state, internal processes, and interface activity were analyzed. Recorded simulations of the DES coprocessor testbench execution provided the necessary data to verify the implementation of the DES algorithm and to analyze the coprocessor's performance.

5.2.1 CPU Interface The CPU/DES coprocessor interface was tested to verify the coprocessor's response to CPU requests. This testing involved tracking the status of both the CPU and DES coprocessor control lines and data bus activity between the two components.

Operation of the DES coprocessor and subsequent state transitions were regulated by the CPU control lines M_IO, WR, ASTR and address lines A15 and A2. The coprocessor state was determined by a combination of these CPU control lines and, in turn, feedback was provided to the CPU via coprocessor status lines INTR, READY0, BUSY, and ERROR.

The DES coprocessor was addressed when the CPU accessed the I/O port addressed by A15. The type of bus transaction was determined by the control line WR and address line A2. Writing to the coprocessor occurred when WR = '1', while WR = '0' represented a coprocessor read. Address line A2 was used by the coprocessor to discern between opcode instructions (i.e., A2 = '0') and operand data (i.e., A2 = '1') during CPU write cycles.

Bus transfers between the CPU and DES coprocessor were performed with double-word aligned data (i.e., 4 bytes or 32 bits). On coprocessor writes the CPU asserted the address strobe line (ASTR) when bus data was valid(19:5-349). Conversely, the DES coprocessor asserted its ready line (READY0) to acknowledge the end of both read and write bus cycles with the CPU.

The asynchronous timing of the CPU/DES coprocessor interface was regulated by the system clock in conjunction with the address strobe and coprocessor ready line. New bus cycles were initiated on either positive or negative transitions of the system clock and ended on a clock transition when the ready line was asserted low.

5.2.2 General DES Algorithm Functions The primary functions of the DES algorithm were implemented as procedures within the behavioral description of the DES coprocessor (see Appendix B.2). The DES function to execute is determined by the opcode instruction issued by the CPU and function execution results from a procedure call within the DES coprocessor's execute process.

The execution paths of the DES operation flow chart shown in Figure 4.5 correspond to unique CPU opcode instructions. Execution of the DES algorithm functions, via VHDL behavioral procedure calls, corresponds to these flow chart paths.

5.2.3 Simulation Initialization Prior to execution each processing node is initialized with required simulation information. This process is performed by the Sim_Init procedure, within the execute process, of the DES coprocessor (see Appendix B.2).

Testing of this basic DES function required verification that the DES coprocessor stored the processing LP's configuration in Random Access Memory (RAM) and relayed the earliest time for subsequent messages to all output LPs via "null" messages.

Simulation configurations with both one and two LPs per processing node, identical to the carwash simulation, were tested. Since the two LP configuration requires additional support from the DES coprocessor, and the one LP test is incorporated as a subset of this configuration; the remainder of this chapter will focus on the testing of the two LPs per processing node configuration.

The simulation's LP configuration was stored in predetermined partitions within the DES coprocessor's RAM (see Figure 4.3). Access to the designated partition was determined via a RAM partition pointer table, which was stored prior to coprocessor activation, and is referenced by the LP's logical number (i.e., 0, 1, 2, ...). The stored data consisted of essential simulation data and the identities of the interacting input and output LPs (i.e., input and output arcs) of the simulation.

The first four addresses of the LP's RAM partition store essential simulation data for each LP on the node. This data includes the input LP status (i.e., ARCS_IN_STATUS register), the LP's inherent delay time, the total number of input and output arcs, and the current simulation time of the LP.

After each LP's RAM partition is configured with simulation data, "null" messages are sent to each connected output LP. The sending of "null" messages requires interaction with the CPU for access to the interconnection circuitry of the iPSC/2. Hence, the DES coprocessor must issue an interrupt request (i.e., INTR asserted high) to the CPU in order to send these messages. Each "null" message is comprised of four doubleword fields (i.e., TO_LP, FROM_LP, SAFE_TIME, and a NULL) which are routed to each output arc by the CPU.

Verification of the simulation initialization function was achieved by monitoring the RAM partition of each LP supported by the DES coprocessor. End-to-end transmission of "null" messages was not possible; however, by monitoring the CPU/DES coprocessor interface, the required "null" message transmissions to all output arcs, via CPU interrupt requests and subsequent acknowledgements, were verified.

5.2.4 Post Message The scheduling of simulation events for an LP in a distributed processing architecture is done through a message passing scheme. Received messages are not executed immediately, rather they are posted in an event list and scheduled for execution in a time-increasing order.

Adding incoming messages to the event list is performed by the Post_msg procedure. The next-event list is maintained in a Content Addressable Memory (CAM) which provides an event retrieval time complexity of O(1).

The message to post consists of four doubleword fields containing the TO_LP, the FROM_LP, a TIME_TAG, and a memory pointer to the message in the CPU's primary memory. Receipt of the message data was verified by monitoring the bus transfer cycles with the CPU and ensuring the general purpose registers within the DES coprocessor were subsequently loaded with this data.

Actual posting of an event to the CAM is contingent upon available storage space within the CAM. The CAM free space status bit of the FLAGS register must be verified prior to storage of the event in the CAM. Verification of event posting was accomplished by monitoring the CAM control lines and the DES system bus. The contents of the CAM

were also examined to verify event posting and updating of the event valid bit for each added event.

After posting a new event, the CAM provides the DES coprocessor with an update of its free space status. The CAM status bit in the FLAGS register (i.e., bit 0) is then updated accordingly. In addition to verifying the updated FLAGS register, the receiving LP's input arcs status was checked to ensure it reflected an input message from the sending LP.

Posting of new events, after the CAM capacity was reached, uses the DES RAM swap space (see Figure 4.3). The DES coprocessor was able to discern this condition by monitoring the CAM free space status bit in the FLAGS register.

5.2.5 Get Next Event The DES coprocessor's ability to retrieve the next scheduled event for CPU processing was evaluated by initiating a request for a specific LP's next event from the CPU. In order to execute the Get_Event instruction, the DES coprocessor must first read the specified LP's RAM partition and check the arcs in status, to determine if all input arcs have posted an incoming message.

Depending on the input arcs status, the DES coprocessor either responds with a wait for event message to the CPU or the next event for the specified LP is retrieved from the CAM and sent to the CPU. Next event retrieval was verified by monitoring the CAM control lines and examining the CAM's content to ensure the earliest valid event for the requesting LP was retrieved. Additionally, the valid bit for the next event in the CAM was checked to ensure its status was changed after providing the next event.

The simulation time advance function is also executed when the next event is sent to the CPU for processing. Advancing the LP's simulation time was verified by monitoring the LP's previous and updated simulation times. The time advance involves the update of the current simulation time by advancing the simulation clock to the time of the next event's scheduled execution.

If the next event to execute was null (i.e., an event memory pointer of 0) the time advance function still occurs and must be verified; however, with this condition the output of a "null" message, containing the new safe time, to all output arcs must also occur.

Additionally, execution of another Get_Event operation must take place to satisfy the CPU's outstanding request for the next event.

5.2.6 Post Event The generation of event messages was verified by sending simulation results for posting from the CPU to the DES coprocessor. The CPU result is composed of the executing LP's identity, the identity of the LP to receive the result, and a memory pointer addressing the event to post in the CPU's primary memory.

Evaluation of the Post_Event operation required verification that an event message, with an updated time tag, was sent to the intended receiver, while "null" messages were sent to all remaining output arcs of the executing LP. Verifying the execution of this procedure required a check of the LP RAM partition read, for the identities of all output arcs connected to the executing LP. Additionally, the DES coprocessor's ability to discern between which LP received the event message vice those that received "null" messages was also verified.

The proper time tag for event messages was verified as the sum of the current simulation time and the executing LP's inherent delay. Similarly, the "null" message safe time was verified to be the same value.

5.3 DES Coprocessor Design Testing

The VHDL behavioral design of the DES coprocessor system was tested as outlined in the previous section. Each component module of the design was tested separately, culminating in an overall DES coprocessor system test. Two configurations were used for system testing. Initially, a single LP per processing node was tested and then the DES coprocessor system was tested with two LPs sharing the same CPU.

The two LP configuration test encompassed the individual module tests and incorporated the single LP configuration as an integral part of the test. Because of its broad scope, the two LP per processing node system test was used to evaluate the VHDL design and to verify the implementation of the fundamental DES simulation functions.

The two LPs per processing node configuration is reflected in the logical process

mapping used for the carwash simulation of Figure 3.4. The source and wash LPs of node 0 and their interconnections, with feedback from the wash exit, were used to verify the DES coprocessor design.

Design verification was accomplished by analyzing test data provided by the Zycad VHDL system. Script outputs, from the testbench configuration (see Figure 5.1), provided signal values and event times, while showing process variables and state values throughout the system test. Additionally, the Zycad VHDL system's General Purpose Post processor (GPP) was used to generate signal traces for selected system ports. The signal traces provided a graphical representation of signal activity and system variable values.

5.3.1 CPU Interface The control lines and asynchronous bus cycles interfacing the CPU and DES coprocessor were analyzed during execution of the DES coprocessor

functions. One or more CPU write cycles occurred for each DES coprocessor function. These write cycles demonstrated the required interactions between the CPU and DES coprocessor.

The initiation of an opcode transfer bus cycle is shown below. The opcode for the Sim_Init instruction is encoded in the three most significant bits (i.e., "000") of the doubleword output on the data bus. The CPU control lines, WR and M_IO, indicated the transmission of an opcode to the DES coprocessor (i.e., A2 = '0' and A15 = '1'), while the DES port values reflected the corresponding receive mode for this transfer.

```
5 NS                          -- CPU data_bus
     M94:      EVENT /DES_SYS_TEST_BENCH/CPU/DATAOUT (value =
               "00000000000000000000000000000000")
                              -- CPU control signals
     M88:      EVENT /DES_SYS_TEST_BENCH/CPU/A2OUT (value = '0')
     M87:      EVENT /DES_SYS_TEST_BENCH/CPU/A15OUT (value = '1')
     M85:      EVENT /DES_SYS_TEST_BENCH/CPU/WROUT (value = '1')
     M86:      EVENT /DES_SYS_TEST_BENCH/CPU/M_IOOUT (value = '0')
                              -- DES Coprocessor port values
     M17:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CMD0 (value = '0')
     M16:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS2 (value = '0')
     M14:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WR (value = '1')
     M15:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS1 (value = '0')
```

After one-half clock cycle (i.e., 62.5 ns for 8 Mhz), the CPU's address strobe line, ASTR, was asserted, indicating valid data on the bus. The coprocessor entered the I/O state, as shown by monitor M52, and asserted a parallel read at 139 ns. The data bus was latched, via the parallel I/O port (i.e., RDP = '1'), and buffered in the DES coprocessor at 191 ns. Loading of the instruction register with the opcode was verified at 196 ns. The bus cycle was then terminated by the DES coprocessor asserting the ready status line, READY0, low at 253 ns.

```
62 NS                          -- CPU address strobe
     M93:      EVENT /DES_SYS_TEST_BENCH/CPU/ASTROUT (value = '1')
     M13:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CLK (value = '1')
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
                          -- DES Coprocessor STATE
     M52:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CPU_IO (value = TRUE)
     M60:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IOWAIT (value = '1')
124 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
139 NS                    -- DES Coprocessor read
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '1')
159 NS                    -- DES data_bus
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "00000000000000000000000000000000")
186 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
191 NS                    -- DES Coprocessor input buffer
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000000000")
196 NS                    -- DES Coprocessor instruction register
     M79:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IR (value =
               "00000000000000000000000000000000")
201 NS
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '0')
221 NS
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
248 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
     M60:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IOWAIT (value = '0')
     M52:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CPU_IO (value = FALSE)
253 NS                    -- DES ready line ends bus cycle
     M20:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/READY0 (value = '0')
     M90:      EVENT /DES_SYS_TEST_BENCH/CPU/RDYIN (value = '0')
```

A graphical representation of the DES coprocessor opcode read bus cycle is shown in

Figure 5.2. This representation of the first bus transfer cycle of the simulation is somewhat misleading in that, the DES coprocessor, because of testbench limitations, requires a system clock transition at simulation startup, before entering the I/O state. Hence, the first half clock cycle for the DES coprocessor is idle (i.e., until READY0 = '1') at test startup.

This timing diagram shows a typical CPU to DES coprocessor write cycle. The CPU control lines access the DES coprocessor, while the address strobe, ASTR, indicated valid data on the system bus. After latching the data, the DES coprocessor signaled the end of the bus transfer cycle by asserting the ready line, READY0, low.

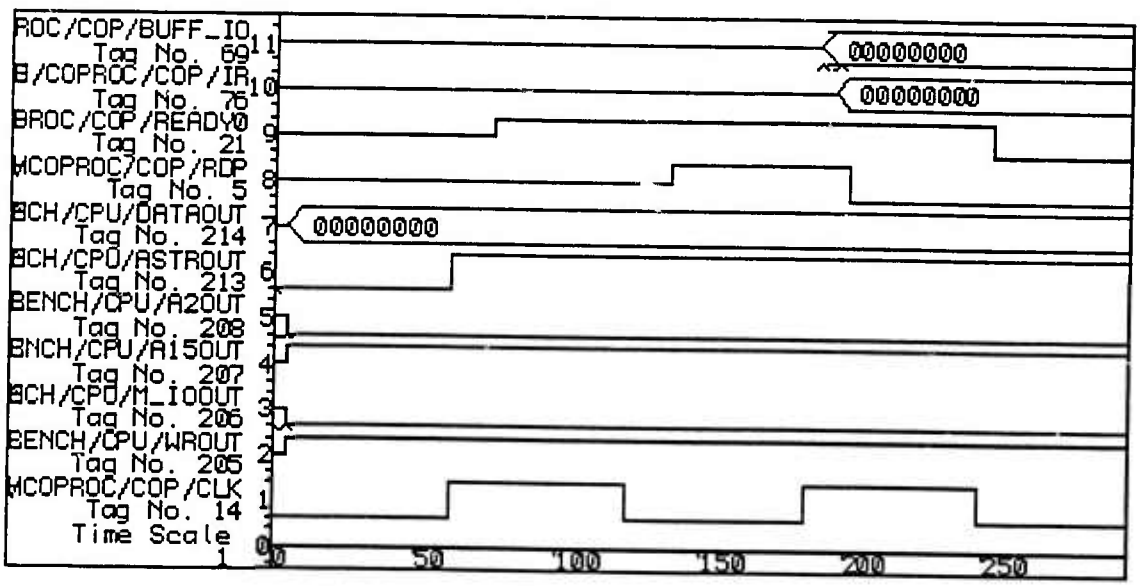

Figure 5.2. Opcode Read Bus Cycle of DES Coprocessor

5.3.2 Simulation Initialization Function Each simulation logical process on a processing node was configured with essential data defining the overall simulation configuration. After sending the opcode instruction, the CPU transferred the required initialization operands to the DES coprocessor. An excerpt of trace data from the system test, showing the first LP's (i.e., node 0 source LP, from Figure 3.4) receipt of the fourth Sim_Init operand (i.e., number of output LPs), is listed below.

The coprocessor's general purpose registers were loaded with the operands for the initialization function (see monitor M68). Register 1 was loaded with the TO_LP (i.e., node 0, LP 0), while the deterministic LP delay, the number of input arcs, and the number

of output arcs (i.e., 4, 2, and 3 respectively) were loaded into registers 2 through 4. The remaining general purpose registers have yet to be used this point in the simulation, hence their values were unknown (i.e., 'X').

```
1131 NS                        -- DES Coprocessor read line
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '1')
1151 NS                        -- DES Coprocessor input
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "00000000000000000000000000000011")
1178 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
1183 NS                        -- DES Coprocessor input buffer
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000000011")
1188 NS                -- Sim_Init operands loaded in Registers 1 to 4
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
             -- TO_LP (i.e., node 0, LP 0)          -- LP delay
        ("00000000000000000000000000000000","00000000000000000000000000000100",
           -- number of input arcs                  -- number of output arcs
          "00000000000000000000000000000010","00000000000000000000000000000011",
          "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
          "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
          "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
```

Figure 5.3 shows a GPP trace of the general purpose registers loaded with the primary Sim_Init operands. This figure shows the register loading sequence of buffering the system data bus and loading the operands sequentially into the general purpose registers. This LP specific data was used to initialize the simulation, therefore it was maintained in local registers while executing the Sim_Init procedure and stored in the LP's RAM partition after function execution.

The addresses of the input and output LPs, for the Sim_Init operation were also provided by the CPU; however, since a maximum of 20 input/output arcs are possible, these additional operands were stored in the LP's RAM partition and read when needed. The DES coprocessor's local data bus, the DES coprocessor/RAM interface, and the LP's RAM partition were monitored to ensure the proper storage of the input/output arc operands.

Test data showing the receipt and storage of the LP's first input arc in the RAM partition follows. Monitor, M72, shows an input to LP 0 on node 0 from itself (i.e., doubleword representing LP # and node #). This reflects the operation of a source LP

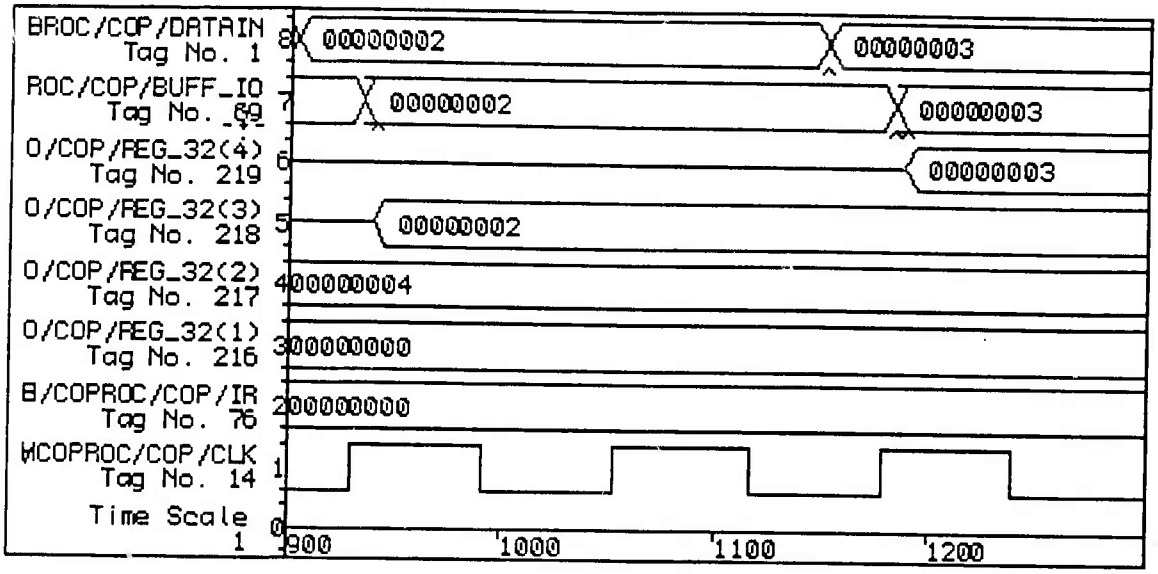

Figure 5.3. DES Coprocessor Registers with Sim_Init Operands

generating a new car event in the carwash simulation. The operand was temporarily held in general purpose register 8, while the RAM partition base address was read from index '0' (i.e., LP 0) of the RAM partition pointer table. After reading the RAM partition base address (i.e., 544_{10}), it is temporarily held in general purpose register 9 and the RAM write cycle followed.

```
1431 NS                        -- LP's 1st input arc
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000000000")
1436 NS
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000000000000100",
         "00000000000000000000000000000010","00000000000000000000000000000011",
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
                                              -- operand in (Register 8)
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","00000000000000000000000000000000",
         "XXXX[illegible]XXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
1441 NS
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '0')
1451 NS                        -- DES RAM read signals
     M11:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '1')
     M12:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RW (value = '0')
1461 NS
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
1466 NS                        -- RAM read address
      M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "00000000000")
```

```
                        -- LP RAM partition base_address
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "00000000000000000000001000100000")
1488 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
1501 NS                        -- base_address to input buffer
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000001000100000")
1506 NS
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000000000000100",
         "00000000000000000000000000000010","00000000000000000000000000000011",
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
         "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","00000000000000000000000000000000",
             -- LP's base_address
         "00000000000000000000001000100000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
```

The RAM write cycle, storing the first input arc for LP 0, is reflected in the following excerpt of test data. The received operand is put onto the DES system data bus at 1541 ns while the RAM partition address and read control line are asserted at 1551 ns and 1566 ns respectively. As designed, the first input operand was stored at an offset of four (i.e., 548_{10}) from the RAM partition base address, leaving space for the operands remaining in general purpose registers one through four. The RAM partition address was then incremented by one and saved in general purpose register six for additional write cycles.

```
1516 NS
     M11:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '0')
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
1521 NS                        -- buffer data for RAM
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000000000")
     M12:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RW (value = 'Z')
1536 NS                        -- DES Coprocessor I/O operation
     M11:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '1')
1541 NS                        -- DES Coprocessor data_bus
      M1:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
               "00000000000000000000000000000000")
1550 NS
     M13:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CLK (value = '1')
1551 NS                        -- RAM write address
      M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "01000100100")
1566 NS                        -- RAM write signal
     M12:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RW (value = '1')
1581 NS
```

```
M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
   ("00000000000000000000000000000000","00000000000000000000000000000100",
    "00000000000000000000000000000010","00000000000000000000000000000011",
                                                -- next RAM partition addr
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","00000000000000000000001000100101",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX","00000000000000000000000000000000",
        -- LP's base_addr
    "00000000000000000000001000100000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
1596 NS
 M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "ZZZZZZZZZZZ")
M11:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '0')
```

Figure 5.4 is a GPP logic diagram of the DES coprocessor write operation described above. The assertion of the read control line occurs after both data and the RAM memory address are stable, allowing the RAM to read and store the input arc identity for later retrieval. Storage was performed in consecutive RAM partition locations by incrementing the memory address between successive write operations.

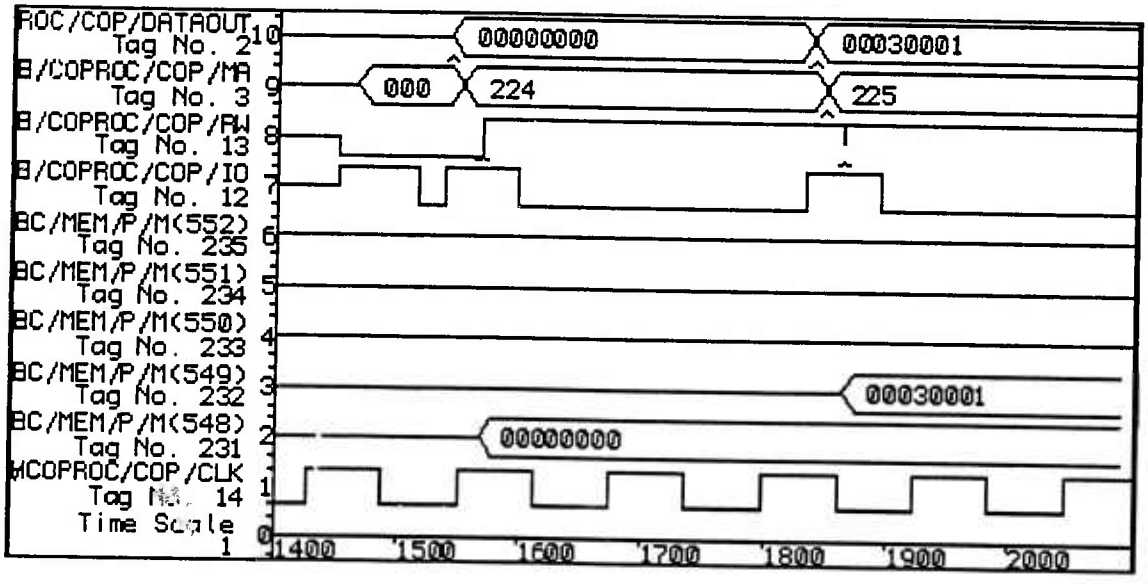

Figure 5.4. DES Coprocessor Write to RAM Partition

The sending of "null" messages to all output arcs was the final action of the Sim_Init function. This operation was verified by monitoring the DES coprocessor/CPU interface to ensure complete messages, with accurate safe times, were routed through the CPU to each output arc.

A sample of one portion (i.e., sending of the TO_LP field) of a typical "null" message send is reflected in the following test data excerpt. The DES coprocessor asserted an

interrupt request to send the first of four "null" message fields (i.e., TO_LP, FROM_LP, Safe_Time, Null) at 3154 ns. The CPU, in turn, serviced the interrupt with a coprocessor read (i.e., LP 1 on node 1 at 3217 ns) which was terminated by the coprocessor asserting the READY0 line low at 3247 ns. After receiving all required inputs, the CPU is able to build an output message and route it accordingly, via the interconnection circuitry of the Intel *i*PSC/2 hypercube.

```
3154 NS                    -- CPU interrupt request
      M3:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/INTR (value = '1')
     M89:     EVENT /DES_SYS_TEST_BENCH/CPU/INTIN (value = '1')
3162 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
3172 NS                    -- CPU acknowledge via control signals
     M85:     EVENT /DES_SYS_TEST_BENCH/CPU/WROUT (value = '0')
     M87:     EVENT /DES_SYS_TEST_BENCH/CPU/A15OUT (value = '1')
     M86:     EVENT /DES_SYS_TEST_BENCH/CPU/M_IOOUT (value = '0')
     M14:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WR (value = '0')
     M16:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS2 (value = '1')
     M15:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS1 (value = '0')
3187 NS                    -- DES Coprocessor write
      M5:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTP (value = '1')
      M3:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/INTR (value = '0')
      M1:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
              "00000000000000010000000000000001") -- TO_LP
     M89:     EVENT /DES_SYS_TEST_BENCH/CPU/INTIN (value = '0')
3217 NS
     M95:     EVENT /DES_SYS_TEST_BENCH/CPU/DATAIN (value =
              "00000000000000010000000000000001")
3224 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
3247 NS
      M5:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTP (value = '0')
                         -- DES Coprocessor ends bus cycle
     M20:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/READY0 (value = '0')
     M90:     EVENT /DES_SYS_TEST_BENCH/CPU/RDYIN (value = '0')
```

5.3.3 Post Message Function The posting of an incoming message involved all components of the DES coprocessor design. The parallel I/O ports received the necessary operands from the CPU, while related LP data, stored during initialization, was held in the coprocessor's RAM, and the next event list was maintained in the coprocessor's CAM.

For testing, several messages, both "null" and real, were posted to each LP's next event list in the CAM. The CPU memory pointers used for testing were arbitrarily cho-

sen to monitor message field changes. Therefore, the memory pointers have no direct correspondence to actual memory addresses. Loading of the opcode instruction for this procedure, and all DES coprocessor simulation functions, is identical to that of the Sim_Init instruction previously described.

Operands for the Post_Msg function included the four fields of the message to post (i.e., TO_LP, FROM_LP, Time_Tag, and a memory pointer to the event). Therefore, four CPU bus write cycles were required to load the message fields. The last bus transfer cycle, containing the CPU memory pointer to the event, is shown below.

```
9920 NS                        -- CPU address strobe
     M93:      EVENT /DES_SYS_TEST_BENCH/CPU/ASTROUT (value = '1')
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
9935 NS                        -- DES Coprocessor read
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '1')
9955 NS                        -- DES Coprocessor input
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "11101011101011101011101011101011")
9982 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
9987 NS                        -- DES Coprocessor input buffer
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "11101011101011101011101011101011")
9992 NS
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
               -- TO_LP                          -- FROM_LP Node 3 & LP 1 (exit)
        ("00000000000000000000000000000000","00000000000000110000000000000001",
               -- event Time_Tag                 -- event memory pointer
         "00000000000000000000000000001111","11101011101011101011101011101011",
               -- current data in remaining registers
         "00000000000000000000000000000000","00000000000000000000001000111110",
         "00000000000000110000000000000001","00000000000000110000000000000001",
         "00000000000000000000001000111000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
9993 NS
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
9997 NS
      M4:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDP (value = '0')
10017 NS
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
10044 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
     M60:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IOWAIT (value = '0')
     M52:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/CPU_IO (value = FALSE)
```

```
10049 NS                    -- DES Coprocessor ends bus cycle
     M20:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/READYO (value = '0')
     M90:     EVENT /DES_SYS_TEST_BENCH/CPU/RDYIN (value = '0')
     M94:     EVENT /DES_SYS_TEST_BENCH/CPU/DATAOUT (value =
              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
```

The DES coprocessor has read and buffered the fourth field of the message to post at 9987 ns. After transfer to a general purpose register, the complete message to post was contained in general purpose registers one through four as reflected in monitor M68.

Prior to storing the event, the DES coprocessor compared the FROM_LP identity, in register two, with the input arc identities maintained in the RAM partition, to update the LP's arcs-in status. This was accomplished through a sequential read of the input arcs from the RAM partition, until a match was found.

Reading of input LPs from RAM is shown below. The test data indicated that the second input arc, stored at RAM address 549_{10}, was the sending LP for the received message. Monitor M68 at 10666 ns, reflects the TO_LP's arcs-in status in general purpose register six. The status was updated, by setting the second least significant bit, as a result of the input LP match. Following the arcs-in status update, the new arcs-in status was saved in the LP's RAM partition for later use with the Get_Event function.

```
10601 NS                    -- DES Coprocessor I/O operation
     M11:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '1')
10602 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
10616 NS                    -- RAM read address (549)
      M2:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "01000100101")
                            -- RAM data equals input arc ID
   VMON5:      READ /DES_SYS_TEST_BENCH/COPROC/MEM/P/M(549) (value =
              "00000000000000110000000000000001")
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "00000000000000110000000000000001")
10651 NS
     M72:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
              "00000000000000110000000000000001")
10664 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
10666 NS                    -- RAM matches input_arc ID
     M64:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MATCH (value = TRUE)
      M2:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "ZZZZZZZZZZZZ")
```

```
     M11:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '0')
     M68:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000110000000000000001",
         "00000000000000000000000000001111","11101011101011101011101011101011",
                                              -- 2nd ARC_IN updated
         "00000000000000000000001000100000","00000000000000000000000000000010",
         "00000000000000000000000000000100","00000000000000100000000000000011",
         "00000000000000000000000000000000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
10671 NS                      -- DES Coprocessor ends RAM read
     M12:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RW (value = 'Z')
10686 NS
     M11:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '1')
10691 NS            -- update ARCS_IN_STATUS in LP RAM partition
      M1:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
              "00000000000000000000000000000010")
```

Posting of the message to the next event list involved three bus transfers to the CAM. During the first cycle the TO and FROM_LP fields of the event were sent to the CAM (see CAM fields in Figure 4.4). The event Time_Tag and pointer to CPU memory followed in subsequent doubleword bus transfer cycles.

Test data verifying the third CAM write bus cycle is shown below. The event memory pointer from register four was written to the CAM and appended to the to the previous data to complete the next event.

```
11001 NS                      -- Event memr_ptr assigned by CPU
     M72:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
              "11101011101011101011101011101011")
11016 NS                      -- DES Coprocessor data_bus
      M1:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
              "11101011101011101011101011101011")
11031 NS                      -- CAM write signal
      M8:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTC (value = '1')
                              -- NOT get_next_event
      M9:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NE (value = '0')
  VMON15:     READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(1) (value =
               "10000001100001                                    -- TO/FROM_LP
                00000000000000000000000000001111                  -- Time_Tag
                11101011101011101011101011101011")                -- Memr_Ptr
```

The most significant bit of the posted event in the CAM was set to indicate a valid event. The posted event represents a feedback message from the carwash exit (i.e., node 3

LP 1 from Figure 3.4), scheduled for the simulation time of 15 units. The event memory pointer indicates the event data structure is stored in CPU main memory at address EBAEBAEBh.

After posting the new event to the next event list, the CAM returned a free-space status to the DES coprocessor. The DES coprocessor maintains the CAM free-space status in the FLAGS register, bit (0), which is read to determine if CAM overflow has occurred.

5.3.4 Get Next Event Function The Get_Event function was verified by monitoring the DES coprocessor's use of the requested LP's arcs_in_status register and ensuring the proper action (i.e., wait message or retrieval of next event) was taken. To verify the proper next event was provided, the contents of the CAM were checked both before and after event retrieval.

The time advance of the LP's simulation clock was also verified during this portion of system testing by ensuring the LP's simulation clock was updated with the next event time. Additionally, the sending of "null" messages to all output arcs, which is required when the next scheduled event contains a "null" memory pointer, was verified.

The DES coprocessor's execution of the Get_Event instruction requires additional operands (i.e., arcs_in_status and simulation time) which were read from the LP's RAM partition. Below is test data that verifies the RAM partition was accessed to retrieve the necessary LP data for Get_Event execution.

The LP requesting the next event was provided by the CPU along with the opcode instruction and was maintained in general purpose register one. Using the LP number as an index, the DES coprocessor accessed the proper RAM partition, via the partition pointer table. The contents of four RAM addresses, starting with the partition base address, were read and used to update registers three through six with the essential LP data (i.e., arcs-in status, LP_Delay, number of input/output arcs, and Simulation_Time).

An excerpt of the system test results below, show the coprocessor reading the LP's simulation time from the RAM partition (i.e., memory address 547_{10}) and updating the general purpose registers. The contents of register three indicated that all input arcs (i.e.,

two for LP 0 on node 0) have posted messages; therefore, the next event for LP 0 may be retrieved from CAM memory.

```
                    -- LP 0 executing Get_Event
16978 NS                      -- RAM address of LP simulation time
      M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "01000100011")
   VMON3:       READ /DES_SYS_TEST_BENCH/COPROC/MEM/P/M(547) (value =
               "00000000000000000000000000000000")
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "00000000000000000000000000000000")      -- LP simulation time
16988 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
17013 NS
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000000000")
17018 NS
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
            -- LP 0                                -- RAM partition base_addr (544)
        ("00000000000000000000000000000000","00000000000000000000001000100000",
            -- ARCS_IN_STATUS register            -- LP delay (4)
         "00000000000000000000000000000011","00000000000000000000000000000100",
            -- # input | # output arcs            -- LP simulation time
         "00000000000000010000000000000011","00000000000000000000000000000000",
         "00000000000000000000000000000100","00000000000000010000000000000011",
         "00000000000000000000000000000000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
17028 NS
      M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "ZZZZZZZZZZZ")
     M11:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/IO (value = '0')
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
```

The DES coprocessor then retrieved the next event from the CAM, via a CAM read cycle; however, the CAM was first given the identity of the requesting LP to perform a search for the earliest scheduled event for that LP.

The system test data that verifies a next event read from CAM is shown below. Initially, the CAM was given the identity of the LP requesting the next event. The 'NE' control line alerted the CAM that a new event request was initiated, vice appending a new event to the next event list.

The contents of the CAM were checked, via monitors 15 through 17, which indicated three valid events, two of which were for LP 0. At time 17113 ns, the CAM selected the next event for LP 0 by toggling the valid bit of the third event in the CAM. The first CAM

event (i.e., VMON15) is also scheduled for LP 0; however, the scheduled time of 15 units occurs later than the selected next event which is scheduled for 10 units.

```
                    -- LP 0 executing Get_Event
17093 NS
      M1:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
              "00000000000000000000000000000000")
17108 NS                    -- LP 0 requesting next_event
      M8:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTC (value = '1')
      M9:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NE (value = '1')
   VMON15:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(1)
              (value = "10000001100001                    -- TO/FROM_LP
              00000000000000000000000000001111            -- Time_Tag
              11101011101011101011101011101011")          -- Memr_Ptr
   VMON16:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(2)
              (value = "10000100000000                    -- TO/FROM_LP
              00000000000000000000000000010100            -- Time_Tag
              11001100110011001100110011001100")          -- Memr_Ptr
   VMON17:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(3)
              (value = "10000000000000                    -- TO/FROM_LP
              00000000000000000000000000001010            -- Time_Tag
              11110011110011110011110011110000")          -- Memr_Ptr
17112 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
17113 NS
   VMON15:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(1)
               (value = "10000001100001
               00000000000000000000000000001111
               11101011101011101011101011101011")
   VMON16:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(2)
              (value = "10000100000000
              00000000000000000000000000010100
              11001100110011001100110011001100")
   VMON17:    READ/DES_SYS_TEST_BENCH/COPROC/CAM/CAM_PROC/EVENT(3)
              (value = "00000000000000                -- toggle "valid" bit
              00000000000000000000000000001010
              11110011110011110011110011110000")
```

The CAM event, which consists of 77 bits, was then parsed into three segments and sent to the DES coprocessor for submission to the CPU. The most significant bit of the first segment (i.e., TO/FROM_LP identities) is used by the CAM to reflect that additional events, received from the same LP, still remain in the CAM. Since no additional events from the same LP, remain in the CAM, this bit is not set which alerts the DES coprocessor to update the arcs-in status register of the requesting LP.

```
17168 NS                    -- CAM read signal
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '1')
      M9:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NE (value = 'Z')
      M8:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTC (value = '0')
                    -- event TO/FROM_LP (source generates car)
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "00000000000000000000000000000000")
17174 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
17223 NS
     M68:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000001000100000",
         "00000000000000000000000000000011","00000000000000000000000000000100",
         "00000000000001000000000000000011","00000000000000000000000000000000",
               -- event to/from_lp
         "00000000000000000000000000000000","00000000000000010000000000000011",
         "00000000000000000000000000000000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
17233 NS
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '0')
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
17236 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
17248 NS
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '1')
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "00000000000000000000000000001010")
17298 NS
     M72:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
              "00000000000000000000000000001010")  -- event Time_Tag
     M33:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
17303 NS
     M68:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000001000100000",
         "00000000000000000000000000000011","00000000000000000000000000000100",
         "00000000000001000000000000000011","00000000000000000000000000000000",
               -- event to/from_lp              -- event time_tag
         "00000000000000000000000000000000","00000000000000000000000000001010",
         "00000000000000000000000000000000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
17313 NS
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '0')
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
17328 NS
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '1')
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "11110011110011110011110011110000")
17360 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
17378 NS                    -- Event memory pointer assigned by CPU
     M72:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
```

```
            "11110011110011110011110011110000")
17383 NS
     M68:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000001000100000",
         "00000000000000000000000000000011","00000000000000000000000000000100",
         "00000000000000010000000000000011","00000000000000000000000000000000",
                -- event to/from_lp                -- event time_tag
         "00000000000000000000000000000000","00000000000000000000000000001010",
                -- event memr_ptr
         "11110011110011110011110011110000","XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"))
```

Updating of the arcs-in status register was accomplished by a sequential read of the input arc identities from RAM and searching for the LP that provided the next event. At 17393 ns, general register 10 has been loaded with the next event's input LP identity. This was extracted from the first event segment sent by the CAM, and was used as a reference for the LP identities read from RAM.

```
17393 NS
      M7:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/RDC (value = '0')
       M:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
     M68:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","00000000000000000000001000100000",
         "00000000000000000000000000000011","00000000000000000000000000000100",
         "00000000000000010000000000000011","00000000000000000000000000000000",
                -- event to/from_lp                -- event time_tag
         "00000000000000000000000000000000","00000000000000000000000000001010",
                -- event memr_ptr                  -- next event input LP
         "11110011110011110011110011110000","00000000000000000000000000000000"))
17422 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
```

5.3.5 Post Event Function The Post_Event function was verified by checking that the DES coprocessor received the event to post from the CPU and combined the simulation time with the inherent LP delay to schedule the event's occurrence. To verify posting of the event, the CPU/DES coprocessor interface was monitored to ensure that all output arcs received a message. The designated TO_LP was verified to receive the event message while all other LPs received a "null" message (i.e., memory pointer to event was 0).

Similar to all other DES functions, the opcode instruction and primary operands were received from the CPU. The primary operands for the Post_Event instruction were

the FROM_LP (i.e., the LP to post the event), a pointer to the event data structure in CPU memory, and the TO_LP (i.e., designated LP to receive the event). After receipt, these operands were stored in the first three general purpose registers by the DES coprocessor.

The additional operands for the Post_Event function (i.e., LP_DELAY, number of input/output arcs, and the LP's simulation time) were read from RAM, which was referenced by the partition pointer corresponding to the LP posting the event, and loaded in registers five through seven.

Test data verifying the DES coprocessor's implementation of the required register loading for the Post_Event function are shown below. Monitor, M68, at 23890 ns, reflects the DES coprocessor's registers loaded with the CPU provided operands and the additional operands fetched from LP 0's RAM partition. At time 23895 ns, the sum of registers five and seven (i.e., LP_DELAY and simulation time) was stored in register eight. This sum is the new simulation time which was updated to account for the occurrence of the next event and will be used to time tag the output messages.

```
23855 NS                        -- RAM address with LP simulation time
      M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "01000100011")
   VMON3:      READ /DES_SYS_TEST_BENCH/COPROC/MEM/P/M(547) (value =
               "00000000000000000000000000001010")
       M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
               "00000000000000000000000000001010")     -- LP simulation time
23870 NS
     M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
23890 NS
     M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
               "00000000000000000000000000001010")
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
               -- LP to post event                -- memr_ptr to event
        ("00000000000000000000000000000000","01110111011101110111011101110111",
         "00000000000000010000000000000001","00000000000000000000001000100000",
               -- LP delay                        -- # of input | # output arcs
         "00000000000000000000000000000100","00000000000001000000000000000011",
               -- LP simulation time
         "00000000000000000000000000001010","00000000000000000000000000001010",
         "11110011110011110011110011110000","00000000000000000000000000000000"))
23895 NS
     M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
        ("00000000000000000000000000000000","01110111011101110111011101110111",
         "00000000000000010000000000000001","00000000000000000000001000100000",
         "00000000000000000000000000000100","00000000000001000000000000000011",
```

```
                                                    -- post event time_tag
    "00000000000000000000000000001010","00000000000000000000000000001110",
    "11110011110011110011110011110000","00000000000000000C00000000000000"))
```

Sending messages was performed by the DES coprocessor reading each of the LP's output arc identities from RAM and building the output message (i.e., "null" or event) by comparing with the designated receiving LP's identity held in register three. The test data below shows the output arc, read from RAM, that matches the designated receiving LP identity in register three. The LP identity read from RAM is loaded into register nine. If it does not match the identity of the LP to receive the event, a null message is sent to this output arc.

```
24105 NS                        -- RAM address of output_arc
     M2:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/MA (value = "01000101000")
                             -- output LP (node 1, LP 1)
   VMON8:     READ /DES_SYS_TEST_BENCH/COPROC/MEM/P/M(552) (value =
              "00000000000000010000000000000001")
      M:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAIN (value =
              "00000000000000010000000000000001")
24118 NS
    M83:      EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
24140 NS
    M72:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
              "00000000000000010000000000000001")
24145 NS
    M68:      EVENT /DES_SYS_TEST_BENCH/COPROC/COP/REG_32 (value =
       ("00000000000000000000000000000000","01110111011101110111011101110111",
                -- TO_LP for message
        "00000000000000010000000000000001","00000000000000000000001000100000",
        "00000000000000000000000000000100","00000000000001000000000000000011",
        "00000000000000000000000000001010","00000000000000000000000000001110",
        "00000000000000010000000000000001","00000000000000000000000000000000"))
```

Both event and "null" messages are composed of four fields. The CPU/DES coprocessor interface activity is similar for both "null" and real messages; the only difference being the message's memory pointer, which is set to zero for "null" messages. System test data, verifying the required interface to send an event message is shown below. The test data verifies a CPU read of the first field (i.e., TO_LP) of the output message. The CPU read cycle was initiated by an interrupt request (i.e., INTR = '1') from the DES coprocessor and also terminated by the DES coprocessor by asserting the ready line low at

24313 ns. The remaining message fields in the DES coprocessor registers were subsequently transferred to the CPU, where the message is packaged and routed over the hypercube's interconnection circuitry.

```
24215 NS                    -- CPU interrupt request
      M3:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/INTR (value = '1')
     M89:     EVENT /DES_SYS_TEST_BENCH/CPU/INTIN (value = '1')
24238 NS                    -- CPU acknowledges with control lines
     M85:     EVENT /DES_SYS_TEST_BENCH/CPU/WROUT (value = '0')
     M87:     EVENT /DES_SYS_TEST_BENCH/CPU/A15OUT (value = '1')
     M86:     EVENT /DES_SYS_TEST_BENCH/CPU/M_IOOUT (value = '0')
     M14:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WR (value = '0')
     M16:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS2 (value = '1')
     M15:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/NPS1 (value = '0')
24242 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '1')
24253 NS                    -- DES begins write
      M5:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTP (value = '1')
      M3:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/INTR (value = '0')
                            -- TO_LP for message
      M1:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/DATAOUT (value =
              "00000000000000001000000000000001")
     M89:     EVENT /DES_SYS_TEST_BENCH/CPU/INTIN (value = '0')
24283 NS
     M95:     EVENT /DES_SYS_TEST_BENCH/CPU/DATAIN (value =
              "00000000000000001000000000000001")
24304 NS
     M83:     EVENT /DES_SYS_TEST_BENCH/CPU/CLOCK (value = '0')
24313 NS
      M5:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/WTP (value = '0')
                            -- DES Coprocessor ends bus cycle
     M20:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/READY0 (value = '0')
     M90:     EVENT /DES_SYS_TEST_BENCH/CPU/RDYIN (value = '0')
     M72:     EVENT /DES_SYS_TEST_BENCH/COPROC/COP/BUFF_IO (value =
              "00000000000000000000000000000000")
```

The DES coprocessor relies on the CPU for access to the interconnection circuitry of the iPSC/2 hypercube, hence an interrupt request (i.e., INTR = '1') was used to route the message fields to the CPU for message output. The CPU/DES coprocessor interface signals, shown in Figure 5.5, provided the control needed to transfer message fields between the DES coprocessor and the CPU. The individual bus transfer cycles were asynchronously regulated via the interrupt request and ready lines from the DES coprocessor.

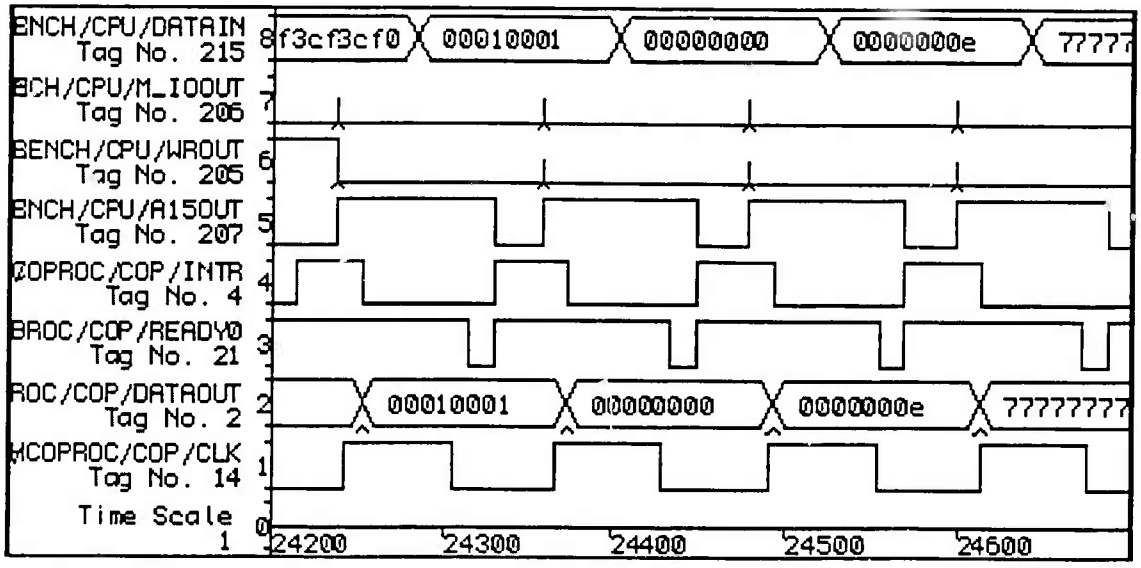

Figure 5.5. CPU/DES Interface Signals for Post_Event Output

5.4 DES Coprocessor System Performance

Implementation of the general DES algorithm was done through the VHDL behavioral description of the DES coprocessor system. In addition to verifying the algorithm's operation, an overall timing analysis was made, using the Zycad VHDL system and the associated general purpose post processor.

Timing data was collected during testbench simulation of the DES coprocessor system. Using this data, the average execution times of the primary DES functions and the CPU/DES coprocessor bus transfer cycle times were obtained.

Table 5.1 is a summary of DES coprocessor system execution times. Timing data used to compile this table was gathered from testbench simulations of the DES coprocessor system supporting two logical processes per computing node.

The significant variance in execution times of several DES functions is evident, but not unexpected. The average execution times for the DES functions will vary, depending on the LP configuration of the simulation.

The large variance in execution times for the Sim_Init and Post_Event functions is attributed to the difference in number of output arcs associated with the two executing LPs. The conservative synchronization protocol requires a message transmission, to each out-

Table 5.1. Function Execution Times for DES Coprocessor System

DES Function	Time (ns)			
	min	max	μ	σ
Sim_Init	1118	2468	1793	675.0
Post_Msg	962	1060	996	38.0
Get_Event	1355	1355	1355	0.0
Post_Event	1113	2313	1713	600.0
CPU_IO	201	325	212	29.8
Intr_CPU	93	105	103	3.4
Send_CPU	65	65	65	0.0

put arc of the LP, every time these functions are executed. Since the two logical processes analyzed (i.e., node 0 of Figure 3.4) have one and three output arcs respectively, a considerable difference in execution times was expected for functions requiring comprehensive communications to output arcs.

The variance in execution time for the Post_Msg function is also a function of the simulation's LP configuration. Unlike the Sim_Init and Post_Event functions, the variance in Post_Msg execution times is related to the number of input arcs passing messages to the LP. The sequential reading of RAM, to update the arcs-in status during execution of the Post_Msg function, has a variable execution time. Given n input arcs to an LP, the Post_Msg function will require $O(n/2)$ time, on average, to match the input LP, using a sequential read of RAM. The LPs simulated have one and two inputs respectively hence, the variance in Post_Msg execution time is attributed to the difference in average RAM read times required to update the LP's input arcs status.

The variance in CPU bus cycle transfers, reflected in the CPU_IO function, are a function of RAM access by the DES coprocessor and a limitation of the testbench design. The DES coprocessor requires one-half clock cycle, at test startup, prior to a transition to the CPU_IO state. The significance of this idle time diminishes as the number of CPU writes increases.

The RAM access time affects average CPU_IO execution time when operands for the Sim_Init function are written to the DES coprocessor. The limited number of general

purpose registers available to the DES coprocessor, requires RAM storage of a portion of the Sim_Init operands. The inclusion of RAM write cycles, particularly the initial write, which involves a read of the partition pointer table, has a significant impact on average CPU_IO time. The effect of RAM write cycles is also diminished as the simulation execution time is increased, since the only function requiring RAM access is Sim_Init, which is executed only once per simulation for each LP.

Average bus transfer cycle times, between the CPU and DES coprocessor, comply with the two clock cycle standard of the Intel 80386 (19:5-353). However, strict compliance to this standard is not required with an asynchronous interface, as the CPU inserts wait states, when needed, to complete longer bus cycles.

VI. Results and Recommendations

6.1 Introduction

A general distributed Discrete Event Simulation (DES) was analyzed to determine the system requirements for a hardware accelerator. A two-phased approach was used in this effort. Initially, a generic DES, represented by a carwash model was analyzed to determine algorithm bottlenecks that exhibited a potential for acceleration through a hardware implementation.

A behavioral description of a hardware coprocessor, implementing the general DES functions, was then specified using the IEEE standard VHSIC Hardware Description Language (VHDL). The coprocessor behavioral description was then simulated using a test-bench representation of the carwash model.

The results of this effort are summarized in this chapter. Additionally, specific topics and related issues that merit further consideration are presented.

6.2 Summary of Findings

Test data collected from simulations of the carwash model, executed on the eight node Intel iPSC/2 hypercube, led to results that are not surprising. The most time consuming portions of this general DES algorithm are those requiring communications support in the distributed hypercube architecture.

The conservative time synchronization protocol, used in distributed computing architectures, requires a significant amount of message passing, with its associated communications overhead, to ensure simulation progress and deadlock avoidance. The basic functions of a distributed Discrete Event Simulation (DES) algorithm were implemented with the SPECTRUM simulation testbed. These basic functions (i.e., initialization, post-message, get-event, advance-time, and post-event) were found to require varying degrees of communications support.

The communication requirements were of two types: sending and receiving messages. In both cases, the required communications resulted in significant time spent waiting to

complete the communications operation. During this waiting time, the processor remained idle and minimal progress, toward simulation completion was accomplished.

Acceleration of general DES simulations is possible by eliminating this processor idle time. One approach to freeing the processor from idle wait time is to implement the basic discrete event simulation functions in a separate coprocessor. Ideally, this will allow the processor to focus on computational activity, with little idle time, while the coprocessor executes the basic simulation functions.

Assuming such a coprocessor is possible and a sufficient workload is available to keep the processor active, the potential for speedup was calculated. A factor of four speedup is possible when a single logical process is executed on each processing node, while the speed-up potential for two logical processes per node approaches twenty. Both estimates assume ideal coprocessor support in that the processor is relieved of all communications idle time associated with the discrete event simulation execution.

The behavioral description of a DES coprocessor was presented. A state machine representation, using a conventional von Neumann architecture, was used to define the system requirements for the design of this discrete event simulation coprocessor.

The coprocessor architecture employs a set of general purpose working registers, a local random access memory for operand and code storage, and a content addressable memory for next event list management. The DES coprocessor system was designed with a 32-bit architecture to provide direct compatibility with proven microprocessor technology.

Simulations of the DES coprocessor behavioral description verified the implementation of the basic DES algorithm functions and confirmed the compatibility with the Intel 80386 32-bit microprocessor. Low-level interfacing with the communications circuitry of the Intel iPSC/2 hypercube was not included in this behavioral description as proprietary restrictions limited access to the operation of this hardware.

The importance of simulating physical systems continues to increase in many fields. The size and complexity of the simulation models employed is also increasing, making the need for improved techniques for executing these simulations paramount. The use of a discrete event simulation coprocessor, has potential for overcoming the problem of

communications overhead and accelerating the execution of discrete event simulations, in a distributed hypercube architecture.

6.3 Recommendations

Several issues concerning the design and implementation of a discrete event simulation accelerator were discovered during the course of this effort. Consideration should be given to the further investigation of these issues to determine their merit and potential for enhancing the current coprocessor design.

6.3.1 CAM Storage As designed the coprocessor relies on the superior memory management capability of the Intel 80386 processor for actual event storage. Hence, the coprocessor is forced to maintain a 32-bit pointer to the address in physical memory where the event and associated data structures are maintained.

The maintenance of this 32-bit pointer in CAM memory is unnecessary as it provides no information the CAM can utilize in searching for the next scheduled event. The potential to double the CAM storage capacity exists, if this physical memory pointer can be maintained outside the CAM and still be associated with the next event search fields.

6.3.2 CAM Overflow The issue of potential CAM overflow and a mechanism to handle such an event was not thoroughly addressed by this thesis. Eventually, the limited CAM capacity will be reached and exceeded as simulations get larger and more events are generated.

The use of multiple CAMs in tandem or a hierarchical structure of multiple CAMs could be employed to delay this eventual overflow. However, the time required to swap in previously overflowed events from main memory, and the best swap in paradigm (i.e., number of events/block size) requires further consideration.

6.3.3 Input Message Status Knowledge of which input arcs have satisfied the message received requirement is paramount to implementing the Chandy-Misra conservative synchronization protocol. The design requirements specify a sequential search algorithm, for the receiving logical process, to update an input arcs status register.

For a small number of input arcs (e.g., 10 was assumed for the design) this technique is sufficient. However, as the number of arcs is increased, the time spent updating this necessary status register may be unacceptable. The possibility of using a second CAM, or a portion of the next event CAM, to update the input arc status has potential for further acceleration as the number of input channels is increased.

6.3.4 Interface to CPU Communications Hardware Incorporation of the direct connect module in the *i*PSC/2 represents a major improvement in the hypercube's message passing efficiency. However, the DES coprocessor design must rely on the CPU to access the DCM module to send and receive messages. In both cases this involves either interrupting the current process or delaying the next process' activation.

Access to DCM hardware documentation was not available, yet the potential for even greater simulation acceleration is present by allowing the DES coprocessor to access the DCM directly. A direct coprocessor to DCM interface would eliminate CPU delays in routing the event and null messages of the simulation and deserves further consideration.

›pendix A. *DES System Packages*

The appendices that follow provide the source code listings of the VHDL files that comprise the DES coprocessor system design and testbench. This first appendix contains the system packages which define the types, subtypes, constants, and functions required by the remaining files to implement the DES coprocessor design.

A.1 Bus Resolution Package

This following appendix presents the bus resolution function used in the DES coprocessor design. The source code listing is taken from the *Zycad VHDL Reference Manual* (32:10-17,18), and tailored for this application. The Bus_sys package provides the unique bus type (i.e., sys_bus) used to define the DES coprocessor and the necessary resolution function, in terms of the Zycad defined multi-valued logic, to avoid bus conflicts. Additionally, type conversion functions, allowing MVL7_Vector-to-Sys_bus and Sys_bus-to-MVL7_Vector is provided.

```
------------------------------------------------------------
-- FILE:  bus_sys.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE: Package for declaring types, subtypes, and
--          functions necessary to provide bus resolution
--          for the DES coprocessor system.
-- REFERENCE: Zycad Reference Manual pp. 10-17,18; 10-78..84
------------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.ATTRIBUTES.all;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;

package Bus_sys is

-------------------BUS RESOLUTION TYPES & FUNCTIONS-------
--  type SYS_BUS is array (INTEGER range <>) of MVL7;
--  function BUS_FUNC(INPUT : SYS_BUS) return MVL7;
--  subtype BUS_BIT is BUS_FUNC MVL7;
--  type BUS_TYPE is array (INTEGER range <>) of BUS_BIT;

-----------------BUS RESOLUTION with ZYCAD---------------
        -- truth table for "BUS_FUNC" function
        constant tbl_BUS_FUNC: MVL7_TABLE :=
        --  ------------------------------------------
        --  |  X    0    1    Z    W    L    H |      |
        --  ------------------------------------------
            (('X', 'X', 'X', 'X', 'X', 'X', 'X'),  -- | X |
             ('X', '0', 'X', '0', '0', '0', '0'),  -- | 0 |
             ('X', 'X', '1', '1', '1', '1', '1'),  -- | 1 |
             ('X', '0', '1', 'Z', 'W', 'L', 'H'),  -- | Z |
             ('X', '0', '1', 'W', 'W', 'W', 'W'),  -- | W |
             ('X', '0', '1', 'L', 'W', 'L', 'W'),  -- | L |
             ('X', '0', '1', 'H', 'W', 'W', 'H')); -- | H |
------------------------------------------------------------
  function BUS_FUNC(INPUT: MVL7_VECTOR) return MVL7;
    attribute REFLEXIVE of BUS_FUNC: function is TRUE;
    attribute RESULT_INITIAL_VALUE of BUS_FUNC: function is MVL7'POS('Z');
    attribute TABLE_NAME of BUS_FUNC: function is "BUS_SYS.tbl_BUS_FUNC";

  subtype BUS_BIT is BUS_FUNC MVL7;
  type SYS_BUS is array (NATURAL range <>) of BUS_BIT;

------------------TYPE CONVERSION FUNCTIONS--------------
  function CHANGE(INPUT : MVL7_VECTOR) return SYS_BUS;          -- ZYCAD drives overloaded
  function CHANGE(INPUT : SYS_BUS) return MVL7_VECTOR;          -- ZYCAD drives overloaded
    attribute CLOSELY_RELATED_TCF of CHANGE: function is TRUE;  -- ZYCAD drive attribute

end Bus_sys;
```

```
package body Bus_sys is

        ---------------
        -- BUS_FUNC
        ---------------
-------------------------------------------------------------
-- Purpose: Resolution function for MVL7 signals.  Used
--          with the DATA_BUS for multiple driver processes.
--          Zycad version (WiredX) from VHDL reference manual
--          (10-84)
----------------------- -------- ------ ---------------------

function BUS_FUNC(INPUT : MVL7_VECTOR) return MVL7 is

        variable RESOLVED : MVL7 := 'Z';

begin
        for i in INPUT'range loop
           RESOLVED := tbl_BUS_FUNC(RESOLVED, INPUT(i));
        end loop;
        return RESOLVED;
end BUS_FUNC;

-----------------------------------------------------------------------
--
-- conversion functions for driving various types
--
-----------------------------------------------------------------------

  function CHANGE (INPUT:SYS_BUS) return MVL7_VECTOR is
  begin
        return MVL7_VECTOR(INPUT);
  end CHANGE;

  function CHANGE (INPUT: MVL7_VECTOR) return SYS_BUS is
  begin
        return SYS_BUS(INPUT);
  end CHANGE;

end Bus_sys;
```

A.2 DES Coprocessor System Package

This appendix contains the system package containing the necessary type and constant declarations for the DES coprocessor design. The generic time delays used in the DES VHDL behavior are defined in this package. Functions that provide type coversion operations and bit vector manipulation required by the DES coprocessor behavior are included in this package.

```
-------------------------------------------------------
-- FILE:  system.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE: Package for encapsulating types and constants
--          for the DES Coprocessor Design.
-------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.ATTRIBUTES.all;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;
use WORK.BUS_SYS.all;

package System is

-------------------CONSTANT TIME DELAYS--------------------------------
  constant ODEL    : TIME := 15 ns;            -- DES output delay
  constant RDEL    : TIME := 60 ns;            -- RAM/DES read delay
  constant WDEL    : TIME := 60 ns;            -- DES write delay
  constant ALUDEL  : TIME := 60 ns;            -- ALU function delay (PER/2)
  constant MADEL   : TIME := 30 ns;            -- DES memr access delay
  constant PER     : TIME := 125 ns;           -- clock period (16 MHz)
  constant DISDEL  : TIME := 20 ns;            -- RAM disable delay
  constant GDEL    : TIME := 5 ns;             -- Parallel I/O gate delay
  constant FFDEL   : TIME := 5 ns;             -- Parallel I/O F-F delay
  constant BUFDEL  : TIME := 10 ns;            -- Parallel I/O buffer delay

-------------------MISC CONSTANTS--------------------------------------
  constant IRsize  : POSITIVE  := 32;          -- Bits in Instr Reg
  constant MAsize  : POSITIVE  := 11;          -- Bits in RAM address
  constant Ndata   : POSITIVE := 32;           -- RAM Data bus size
  constant Naddr   : POSITIVE := 11;           -- RAM Addr bus size
  constant N       : POSITIVE := 32;           -- Parallel I/O port size

-------------------SUBTYPES FOR BIT_VECTORS----------------------------
  subtype DWORD is SYS_BUS(31 downto 0);       -- resolved data_bus
  subtype ADDR  is MVL7_VECTOR(31 downto 0);
  subtype M_ADD is MVL7_VECTOR(10 downto 0);   -- 4K RAM (4 byte/Addr)

-------------------GENERAL PURPOSE REGISTERS---------------------------
  type REG32 is array (NATURAL range <>) of DWORD; -- GP (32 bit) registers

-------------------MISC FUNCTIONS--------------------------------------
  function DWORD_to_MADD(DBL: DWORD) return M_ADD;
  function JOIN_DWORDS(HI, LO: DWORD) return DWORD;
  function MAP_FIELDS(TO_LP, FROM_LP: DWORD) return DWORD;
  function HI_LO_ADD(DBL_WORD :DWORD) return DWORD;


end System;
```

```
package body System is

        ------------------
        -- DWORD_to_MADD
        ------------------
-------------------------------------------------
-- Purpose: Converts (32) bit DWORD to (MAsize) bit
--          RAM memory address (32 bit -> 11 bit)
-------------------------------------------------
function DWORD_to_MADD(DBL: DWORD) return M_ADD is

        variable ADDRESS : M_ADD;

begin
  for I in MAsize-1 downto 0 loop
    if DBL(I) = '0' then
      ADDRESS(I) := '0';
    elsif DBL(I) = '1' then
      ADDRESS(I) := '1';
    else ADDRESS(I) := '1';
    end if;

    assert (DBL(I) = '0' or DBL(I) = '1')
    report "Invalid Memory Address";
  end loop;
  return ADDRESS;

end DWORD_to_MADD;

        ------------------
        -- JOIN_DWORDS
        ------------------
-------------------------------------------------
-- Purpose: joins "hi" WORD (1st DWORD) with "lo"
--          WORD (2nd DWORD) for return DWORD
-------------------------------------------------
function JOIN_DWORDS(HI, LO: DWORD) return DWORD is

        variable JOINED : DWORD;

begin
  for I in 31 downto HI'LENGTH/2 loop
    if HI(I - (HI'LENGTH/2)) = '0' then
        JOINED(I) := '0';
    elsif HI(I - (HI'LENGTH/2)) = '1' then
        JOINED(I) := '1';
    else
        JOINED(I) := '1';
    end if;
```

```
    assert (HI(I - (HI'LENGTH/2)) = '0' or HI(I - (HI'LENGTH/2)) = '1')
    report "Invalid DWORD splicing";
  end loop;

  for I in (LO'LENGTH/2 - 1) downto 0 loop
    if LO(I) = '0' then
        JOINED(I) := '0';
    elsif LO(I) = '1' then
        JOINED(I) := '1';
    else
        JOINED(I) := '1';
    end if;

    assert (LO(I) = '0' or LO(I) = '1')
    report "Invalid DWORD splicing";
  end loop;

return JOINED;
end JOIN_DWORDS;

        ------------------
        -- MAP_FIELDS
        ------------------
-------------------------------------------------------------
-- Purpose:  maps valid fields of (2) input DWORDs to
--           construct return DWORD
-------------------------------------------------------------
function MAP_FIELDS(TO_LP, FROM_LP: DWORD) return DWORD is

        variable COMPACT : DWORD;

begin
  for I in 31 downto 13 loop
    COMPACT(I) := '0';
  end loop;

  for I in 12 downto 8 loop
    if TO_LP(I - 8) = '0' then
        COMPACT(I) := '0';
    elsif TO_LP(I - 8) = '1' then
        COMPACT(I) := '1';
    else
        COMPACT(I) := '1';
    end if;

    assert (TO_LP(I - 8) = '0' or TO_LP(I - 8) = '1')
    report "Invalid Event Address Mapping";
  end loop;

  for I in 7 downto 5 loop
    if FROM_LP(I + 11) = '0' then
```

```
        COMPACT(I) := '0';
      elsif FROM_LP(I + 11) = '1' then
        COMPACT(I) := '1';
      else
        COMPACT(I) := '1';
      end if;

      assert (FROM_LP(I + 11) = '0' or FROM_LP(I + 11) = '1')
      report "Invalid Event Address Mapping";
    end loop;

    for I in 4 downto 0 loop
      if FROM_LP(I) = '0' then
        COMPACT(I) := '0';
      elsif FROM_LP(I) = '1' then
        COMPACT(I) := '1';
      else
        COMPACT(I) := '1';
      end if;

      assert (FROM_LP(I) = '0' or FROM_LP(I) = '1')
      report "Invalid Event Address Mapping";
    end loop;

  return COMPACT;
  end MAP_FIELDS;

        ---------------
        -- HI_LO_ADD
        ---------------
  ------------------------------------------------------
  -- Purpose: Add high and low WORDS of DWORD to return
  --          a DWORD "SUM"
  ------------------------------------------------------
  function HI_LO_ADD(DBL_WORD : DWORD) return DWORD is

        variable HI_WORD, LO_WORD, SUM : DWORD;

  begin

    HI_WORD(15 downto 0) := DBL_WORD(31 downto 16);
    LO_WORD(15 downto 0)  := DBL_WORD(15 downto 0);
    HI_WORD(31 downto 16) :=  "0000000000000000";
    LO_WORD(31 downto 16) := "0000000000000000";
    SUM := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(HI_WORD))) +
                                  BVtoI(MVL7VtoBV(CHANGE(LO_WORD))))));

    return SUM;
  end HI_LO_ADD;

  end System;
```

Appendix B. *DES Coprocessor VHDL Design*

This appendix contains the chip level architecture for the DES coprocessor system as shown in Figure 4.2. The entity declaration and components that make up the DES coprocessor system is given. VHDL behavioral descriptions, as described in Chapters 3 and 4, for each component are included in the following appendices.

B.1 DES Coprocessor Structure

This appendix contains the architectural body of the DES coprocessor. The entity declaration defines the DES coprocessor in terms of sytem input and output ports. The components that make the DES coprocessor system are declared and defined with generic parameters and port descriptions. The signal mapping that connects the system components is also given.

```
-------------------------------------------------------------------------
-- FILE:  des_structure.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE:  Structural architectural body of the DES
--           coprocessor.  Entity declaration defining the inputs and
--           outputs for the DES Coprocessor is given.  Components and
--           signals for the DES coprocessor system are declared and
--           then instantiated.
-------------------------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use ZYCAD.COMPONENTS.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;

entity DES_sys is

  port(
        RUN      : in  MVL7;                  -- status enable/run (Vcc)
        CLK      : in  BIT;                   -- System CLK2  (1/2 CPU sys)
        RESETIN  : in  MVL7;                  -- RESET from CPU
        WR       : in  MVL7;                  -- W/R* from CPU
        NPS1     : in  MVL7;                  -- M/IO* from CPU
        NPS2     : in  MVL7;                  -- A15 from CPU
        CMD0     : in  MVL7;                  -- A2 from CPU ('1'- data, '0'- opcode)
        INTR     : out MVL7;                  -- interrupt request to CPU
        READY0   : inout MVL7;                -- wait state cntrl sig (xfer acknowledge)
        BUSY     : out MVL7;                  -- coprocessor status signal
        ERROR    : out MVL7;                  -- coprocessor error signal
        ADD_STR  : in  MVL7;                  -- ADS* address valid strobe
        SYSIN    : in  DWORD;                 -- CPU data bus
        SYSOUT   : out DWORD);                -- CPU data bus
end DES_sys;

architecture CHIP_LEVEL of DES_sys is

-------------------------------------------------------------------------
--
-- DES Synchronization Coprocessor component
--
-------------------------------------------------------------------------
 component DES

       generic(RDEL, WDEL, ODEL, MADEL, PER: TIME);

       port(   DATAin              : in DWORD;        -- data_bus port
               DATAout             : out DWORD;       -- data_bus port
               MA                  : out M_ADD;       -- ram address port
               INTR                : out MVL7;        -- int request to CPU
```

```
                RDP, WTP              : out MVL7;         -- parallel i/o control
                NINTin                : in MVL7;          -- parallel i/o request
                RDC, WTC, NE, RST     : out MVL7;         -- CAM i/o control
                IO, RW                : out MVL7;         -- RAM i/o control
                CLK                   : in BIT;           -- 1/2 CPU system clock
                WR, NPS1, NPS2, CMD0: in MVL7;            -- CPU control input
                RESETIN, RUN          : in MVL7;          -- CPU control input
                READY0                : inout MVL7;       -- DES status/control
                BUSY, ERROR           : out MVL7);        -- DES status/control output

 end component;

-------------------------------------------------------------------------
--
-- Memory component
--
-------------------------------------------------------------------------
component RAM_MEM

       generic (Ndata: Positive := 32;                    -- # of data lines
                Naddr: Positive := 11;                    -- # of address lines
                RDEL, DISDEL: TIME);                      -- read and disable delay

       port     (DATAI: in DWORD;                         -- data in lines
                DATAO: out DWORD;                         -- data out lines
                ADDR: in MVL7_VECTOR(Naddr-1 downto 0); -- address lines
                CE: in MVL7;                              -- chip enable (high)
                RW: in MVL7);                             -- read (low) and
                                                          -- write (high)
end component;

-------------------------------------------------------------------------
--
-- Parallel I/O Latch component
--
-------------------------------------------------------------------------
 component PAR

       generic(GDEL, FFDEL, BUFDEL: TIME);                -- delay times

       port(    DI : in DWORD;                           -- data input
                DO : out DWORD;                           -- data output
                NDS1, DS2, MD, NCLR : in MVL7;            -- I/O control
                STB : in MVL7;                            -- addr latch enable
                NINT : out MVL7);                         -- interrupt request

 end component;

-------------------------------------------------------------------------
--
-- Content Addressable Memory (CAM) component
```

```
--
-------------------------------------------------------------------------
 component C_MEM

       generic(RDEL, WDEL, DISDEL: TIME);              -- delay times

       port(   DATAinto : in DWORD;                    -- data in
               DATAoutof : out DWORD;                  -- data ouput
               CLK : in BIT;                           -- DES clock
               DS1, NDS2, MODE, N_CLR : in MVL7);      -- CAM control

 end component;

-------------------------------------------------------------------------
-- Signal declarations connecting chip_level
-- components in DES coprocessor.
-------------------------------------------------------------------------
  signal DATA_BUS : DWORD;                             -- DES data bus
  signal DATAin, DATAout : DWORD;
  signal MA : M_ADD;
  signal DATAI, DATAO : DWORD;
  signal NINTin, IO, RW, RDP, WTP : MVL7;
  signal RDC, WTC, NE, RST : MVL7;
  signal ADDR : MVL7_VECTOR(10 downto 0);
  signal DATAinto, DATAoutof : DWORD;
  signal CE : MVL7;
  signal DS1, NDS2, MODE, N_CLR : MVL7;
  signal NDS1, DS2, MD, NCLR, STB, NINT : MVL7;
  signal ZERO : MVL7 := '0';
  signal ONE : MVL7 := '1';
-------------------------------------------------------------------------
-- Component instantiations.
-------------------------------------------------------------------------
begin

  COP: DES

       generic map(RDEL, WDEL, ODEL, MADEL, PER)

       port map(
               DATAin => DATAin,
               DATAout => DATAout,
               MA => MA,
               INTR => INTR,
               IO => IO,
               RW => RW,
               RDP => RDP,
               WTP => WTP,
               NINTin => NINTin,
               RDC => RDC,
               WTC => WTC,
```

```
                NE => NE,
                RST => RST,
                RESETIN => RESETIN,
                RUN => RUN,
                CLK => CLK,
                WR => WR,
                NPS1 => NPS1,
                NPS2 => NPS2,
                CMD0 => CMD0,
                READY0 => READY0,
                BUSY => BUSY,
                ERROR => ERROR);

MEM: RAM_MEM

      generic map(Ndata, Naddr, RDEL, DISDEL)

      port map(
              DATAI => DATAI,
              DATAO => DATAO,
              ADDR => ADDR,
              CE => CE,
              RW => RW);

PARIN: PAR

  generic map(GDEL, FFDEL, BUFDEL)          -- Logic delays

  port map(   DI => SYSIN,                  -- CPU input
              DO => DATA_BUS,               -- out to DES
              NDS1 => ZERO,                 -- low device select
              DS2 => RDP,                   -- variable dev. sel.
              MD => ZERO,                   -- Read Mode
              NCLR => ONE,                  -- no clear
              STB => ADD_STR,               -- CPU strobe (high)
              NINT => NINT);                -- low interrupt

PAROUT: PAR

  generic map(GDEL, FFDEL, BUFDEL)          -- Logic delays

  port map(   DI => DATA_BUS,               -- DES output
              DO => SYSOUT,                 -- out to CPU
              NDS1 => ZERO,                 -- low device select
              DS2 => WTP,                   -- variable dev. sel.
              MD => ONE,                    -- Write Mode
              NCLR => ONE,                  -- no clear
              STB => ZERO,                  -- no strobe
              NINT => open);                -- no interrupt

CAM: C_MEM
```

```
        generic map(RDEL, WDEL, DISDEL)

        port map(
                DATAinto => DATAinto,
                DATAoutof => DATAoutof,
                CLK => CLK,
                DS1 => DS1,
                NDS2 => NDS2,
                MODE => MODE,
                N_CLR => N_CLR);
-----------------------------------
-- Signal mapping between components
-----------------------------------
        DATAin   <= DATA_BUS;                   -- cop input
        DATA_BUS <= DATAout;                    -- cop output
        DATAI    <= DATA_BUS;                   -- ram input
        DATA_BUS <= DATAO;                      -- ram output
        DATAinto <= DATA_BUS;                   -- cam input
        DATA_BUS <= DATAoutof;                  -- cam output
        ADDR     <= MA;
        NINTin   <= NINT;
        CE       <= IO;
        DS1      <= RDC;
        NDS2     <= WTC;
        MODE     <= NE;
        N_CLR    <= RST;

end CHIP_LEVEL;
```

B.2 DES Coprocessor Behavior

The DES coprocessor architectural behavior is given in this appendix. Each of the states of the DES coprocessor, shown in Figure 4.1, is implemented by a dedicated VHDL process within this behavior. The primary DES simulation operations (i.e., initialize, post-message, get-next-event, and post-event) are included as procedures that are called in the execute process.

Additional procedures, that implement frequently needed operations (i.e., interrupt CPU and send null message) are also included in this behavior. The signal multiplexing, used to resolve signals driven by multiple processes, is also included at the end of the behavior.

```
------------------------------------------------------------
-- FILE:  des_beh.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE: Architectural BEHAVIOR of the DES Coprocessor
--          Implements four (4) basic operations (Initialize,
--          Post_Message, Get_Next_Event, and Post_Event for
--          a conservative synchronization protocol, discrete
--          event simulation.
------------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;
-----------------------------------------
--  ENTITY declaration for DES coprocessor
-----------------------------------------
entity DES is

        generic(RDEL, WDEL, ODEL, MADEL, PER: TIME);

        port(DATAin               : in DWORD;   -- DATA_BUS PORT
             DATAout              : out DWORD:=
            "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- DATA_BUS PORT
             MA                   : out M_ADD;  -- RAM ADDRESS PORT
             INTR                 : out MVL7;   -- Int request to CPU
             RDP, WTP             : out MVL7;   -- PAR I/O CONTROL
             NINTin               : in MVL7;    -- PAR I/O STATUS REQUEST
             RDC, WTC, NE, RST    : out MVL7;   -- CAM I/O CONTROL & Mstr CLR
             IO, RW               : out MVL7;   -- RAM CONTROL
             CLK                  : in BIT;     -- 1/2 CPU SYSTEM CLOCK
             WR, NPS1, NPS2, CMDO : in MVL7;    -- CPU CONTROL INPUT
             RESETIN, RUN         : in MVL7;    -- CPU CONTROL INPUT (RUN=STEN)
             READYO               : inout MVL7; -- CPU CONTROL OUTPUT
             BUSY, ERROR          : out MVL7);  -- CPU CONTROL OUTPUT

end DES;
-----------------------------------------
--  ARCHITECTURAL BEHAVIOR of DES coprocessor
-----------------------------------------
architecture BEHAVIOR of DES is

        signal STOP: BIT;
        signal BUSYB, BUSYC, BUSYS, BUSYE: MVL7;
        signal RDY, RDYC, RDYE: MVL7;
        signal BRW, RWS, RWC, RWE: MVL7;
        signal BIO, IOS, IOC, IOE: MVL7;
        signal BMA, MAS, MAC, MAE: M_ADD;
        signal RDCB, RDCS, RDCE: MVL7;
```

```
        signal WTCB, WTCS, WTCE: MVL7;
        signal NEB, NES, NEE: MVL7;
        signal CPU_IO, CPU_IOS, CPU_IOC, START, EXECUTE: BOOLEAN;
        signal DONE, DONEC, DONEE: BOOLEAN;
        signal IOWAIT, IOWAITS, IOWAITC, IOWAITE: BIT;
        signal MATCH : BOOLEAN := false;     -- update ARCS_IN_STAT
        signal LOADED: BOOLEAN := false;     -- GP Regs have next event
        signal HAD_EVENT: BOOLEAN := false; -- have event for CPU
        signal SEND_NULL: BOOLEAN := false; -- Get_Event was "null"
-------------------------------------------------------------------------
-- General purpose REGISTERS are declared as signals.
-- Receive inputs from START, CPU_IO_PROC, and EXECUTE_PROC which are mux'd
-- and the most recent process assertion is new value of Reg_32(X).
-- ------------------------------------------------------------------------
        signal Reg_32, Reg_32S, Reg_32C, Reg_32E : REG32(1 to 10);
        signal BUFF_IO, BUFF_IOS, BUFF_IOC, BUFF_IOE: DWORD :=
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        signal FLAGS, FLAGSS, FLAGSE: DWORD;
        signal IR, ACC: DWORD;
        signal OPCODE: MVL7_VECTOR(31 downto 0);        -- temp for OPCODE

begin
--*************************************************************************
-- Run (STEN) Process   (Chip select circuitry or Vcc)
--*************************************************************************
  RUN_PROC: process(RUN)

  begin
  if RUN = '1' then
    STOP <= '0';
  else
    STOP <= '1';
  end if;

  end process RUN_PROC;
--*************************************************************************
-- State Process        (CURRENT STATE and state transitions)
--*************************************************************************
  STATE: process(RUN, STOP, IOWAIT, CLK, START, CPU_IO, EXECUTE)

  begin
  if (not RUN'STABLE) and (RUN = '1') then
    START <= true;
  elsif ((RUN'STABLE) and (STOP = '0') and (IOWAIT = '0')
                                    and (not CLK'STABLE)) then
    if (START and (NPS1 = '0') and (NPS2 = '1') and (WR = '1')) then
      CPU_IOS <= true;
    else
      CPU_IOS <= false;
    end if;
    if (START and (WR /= '1')) then
```

```
      EXECUTE <= true;
    else
      EXECUTE <= false;
    end if;
  end if;

  end process STATE;
--*************************************************************************
-- Start Process        (Housekeeping and Operating in Idle state)
--*************************************************************************
--
--  CAM overflow swapping and memory management (garbage collection).
--  Checks for CAM overflow events stored in RAM.  Uses a circular queue to
--  take events (one per cycle) from head and move to CAM.
        -- Check FLAGS register:
                -- bit(0): '1' = CAM full; '0' = CAM not full
                -- bits(28->23): Qty of events in RAM
                -- bits(22->12): start addr (head) of events in RAM
                -- (3)*DWORD per event in RAM
-----------------------------------------------------------------------
  START_PROC: process
-----------------------------------------------------------------------
-- variable/constants to manage circular queue of CAM overflow events
-----------------------------------------------------------------------
        constant First_Q_Addr : DWORD := "00000000000000000000001000011111";
        constant Last_Q_Addr  : DWORD := "00000000000000000000000110111111";
        variable Next_Addr    : DWORD := "00000000000000000000000000000000";

  begin
  wait on START until START;
                                                -- unassert "busy" signal
  BUSYS <= '0',
           '1' after ODEL;
----------------------
-- reset/initialization
----------------------
  if RESETIN = '1' then
     IOWAITS <= '1';
     FLAGSS(11 downto 0) <= "010000111110";     -- tail of RAM queue
                                                -- and CAM status bit
     FLAGSS(22 downto 12) <= "01000011111";     -- head of RAM queue
     FLAGSS(28 downto 23) <= "000000";          -- # events in RAM
     IOWAITS <= '0';
  end if;
-------------------------------------------------
-- if CAM not full and events in RAM -> move to CAM
-------------------------------------------------
  if (FLAGS(0) = '0' and
      BVtoI(MVL7VtoBV(CHANGE(FLAGS(28 downto 23)))) > 0) then
    IOWAITS <= '1';
    Next_Addr(10 downto 0) := FLAGS(22 downto 12);      -- head of queue
```

```
-------------------------------------------------------------------------
-- loop to read RAM and load GP registers with event to move to CAM
        -- Reg_32(2) <= TO/FROM_LP ids
        -- Reg_32(3) <= event Time_Tag
        -- Reg_32(4) <= Memr_Ptr to event
-------------------------------------------------------------------------
    LOAD_REG_LOOP:
    for I in 2 to 4 loop                                    -- 2 to 4: indx regs
        MAS <= DWORD_to_MADD(Next_Addr) after MADEL;        -- RAM address
        RWS <= '0' after ODEL;                              -- RAM read cycle
        IOS <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOS <= DATAin after FFDEL;                     -- event data
        wait for FFDEL;
        RWS <= 'Z' after ODEL + GDEL;
        IOS <= '0' after ODEL;
        MAS <= "ZZZZZZZZZZZZ" after ODEL;
        Reg_32S(I) <= BUFF_IOE after FFDEL;                 -- load event data
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
        wait for ODEL;
    end loop LOAD_REG_LOOP;
-------------------------------------------------------------
-- CAM write loop (event_id, event time_tag, memr_ptr to msg)
-------------------------------------------------------------
    CAM_SAVE_EVENT:
    for I in 2 to 4 loop                                    -- use GP reg indx
      BUFF_IOS <= Reg_32(I) after FFDEL;                    -- next event field
      wait for FFDEL;
      DATAout <= BUFF_IOE after ODEL,                       -- put data on bus
                 "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
      wait for ODEL;
      NES <= '0' after ODEL;
      WTCS <= '1' after ODEL;
      wait for WDEL;
      WTCS <= '0' after ODEL;
      NES <= 'Z' after ODEL;
      wait for ODEL;
    end loop CAM_SAVE_EVENT;
-------------------------------------------------------------------
-- Update CAM_FULL bit (FLAGS(0)) with automatic reading of CAM status
-------------------------------------------------------------------
    RDCS <= '1' after ODEL;                                 -- CAM read
    wait for RDEL;
    BUFF_IOS <= DATAin after FFDEL;
    RDCS <= '0' after ODEL;
    wait for ODEL;
    if BUFF_IOS(0) = '1' then                               -- CAM is full
      FLAGSS(0) <= '1';
    end if;
    IOWAITS <= '0';
  end if;
```

```
  end process START_PROC;
--*************************************************************************
-- CPU_IO Process       (Get instruction and data from CPU)
--
-- Pseudo code
--
-- If INPUT (WR = '1')
--   read parallel (RDP = '1') and load into BUFF_IO register
--   End bus_cycle (READY0 = '0')
--     If OPCODE (A2 = '0')
--       load instruction register (IR <= BUFF_IO)
--     If OPERAND (A2 = '1')
--       load operand(s) into gen purpose register (REG_32(X) <= BUFF_IO)
-- Transition to execute state at end of last bus xfer cycle
--*************************************************************************
  CPU_IO_PROC: process

        variable num : INTEGER := 1;            -- register counter
        variable Next_Addr : DWORD;             -- another GP register

  begin
  wait on CPU_IO until CPU_IO;                  -- CPU writes only!!

-------------------------------------------------------
-- reinitialize num counter if previous opcode was executed
-------------------------------------------------------
  if DONE then
    num := 1;
    DONEC <= false;
    BUSYC <= '1' after ODEL;                    -- prompt for Test_Bench
  end if;

  IOWAITC <= '1';                               -- no state changes
  RDYC <= '1' after ODEL;                       -- new bus cycle
  wait until not CLK'stable;                    -- synchronization
  RDP <= '1' after ODEL;
  wait until not CLK'stable;                    -- read in data bus
  RDP <= '0' after ODEL;                        -- DES reads (after STRB)
  BUFF_IOC <= DATAin after FFDEL;               -- DATA_BUS into BUFF_IO
  wait for FFDEL;
-------------------------------
-- opcode was read from data_bus
-------------------------------
  if (CMD0 = '0') then
    IR <= BUFF_IOC after FFDEL;                 -- load inst reg w/opcode
    wait for FFDEL;
    OPCODE <= CHANGE(IR);                       -- ease of reading
    BUFF_IOC <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 1 ns;
                                                -- clears buff (resolve)
-------------------------------
```

```
-- operand was read from data_bus
--------------------------------
  elsif (CMD0 = '1') then                          -- data_bus was operand
-------------------------------------------------
-- (4) essential operands to store in GP registers
-------------------------------------------------
    if (num <= 4) then
      Reg_32C(num) <= BUFF_IOC after FFDEL;        -- load essential registers
      wait for FFDEL;                              -- reg xfer delay
      BUFF_IOC <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 1 ns;
                                                   -- resolve buff
      num := num + 1;                              -- next GP register
-------------------------------------------------
-- store additional operands in LP's RAM partition
-------------------------------------------------
    elsif (num > 4) then
      Reg_32C(8) <= BUFF_IOC after FFDEL;          -- temp for next operand
      wait for FFDEL;
-----------------------------------------------------------------
-- get pointer to LP's RAM partition (this is first add'l operand)
-----------------------------------------------------------------
      if (num = 5) then
        MAC <= DWORD_to_MADD(Reg_32(1)) after MADEL;
                                                   -- LP RAM partition tbl addr
        RWC <= '0' after ODEL;                     -- RAM read cycle
        IOC <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOC <= DATAin after FFDEL;            -- LP RAM base addr ptr
        wait for FFDEL;
        Reg_32C(9) <= BUFF_IOC after FFDEL;        -- for future use (Init_Sim)
        RWC <= 'Z' after ODEL + GDEL;
        IOC <= '0' after ODEL;                     -- must chg before RW
        wait for ODEL;
-------------------------------------
-- LP's data base_addr in RAM + offset
-------------------------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(BUFF_IOC))) + num - 1)));
        BUFF_IOC <= Reg_32(8) after FFDEL;
        wait for FFDEL;
        DATAout <= BUFF_IOC after FFDEL + ODEL, -- put operand on data_bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
        MAC <= DWORD_to_MADD(Next_Addr) after MADEL;  -- RAM address
        RWC <= '1' after 3*ODEL;                   -- RAM write cycle
        IOC <= '1' after ODEL;
        wait for WDEL;
        RWC <= 'Z' after ODEL + GDEL;
        IOC <= '0' after ODEL;                     -- must chg before RW
        MAC <= "ZZZZZZZZZZZ" after ODEL;
        num := num + 1;                            -- next GP register
------------------
-- next RAM address
```

```
------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
        Reg_32C(6) <= Next_Addr;                -- needed in future (Init_Sim)
-------------------------------------------------------------
-- already have LP's RAM address: store add'l operands in RAM
-------------------------------------------------------------
      elsif (num > 5) then
        wait for FFDEL;                         -- xfer of buff_ioc to reg_32(8)
        BUFF_IOC <= Reg_32(8) after FFDEL;
        wait for FFDEL;
        DATAout <= BUFF_IOC after FFDEL + ODEL, -- put operand on data_bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;

        MAC <= DWORD_to_MADD(Next_Addr) after MADEL;  -- RAM address
        RWC <= '1' after 3*ODEL;                -- RAM write cycle
        IOC <= '1' after ODEL;
        wait for WDEL;
        RWC <= 'Z' after ODEL + GDEL;
        IOC <= '0' after ODEL;                  -- must chg before RW
        MAC <= "ZZZZZZZZZZZ" after ODEL;
        num := num + 1;                         -- next GP register
------------------
-- next RAM address
------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_addr))) + 1)));
        Reg_32C(6) <= Next_Addr;                -- needed in future (Init_Sim)
      end if;
    end if;
  end if;
-----------------------------------------------------------------
-- toggle state for multiple CPU_IO state operations in succession
-----------------------------------------------------------------
  wait until not CLK'stable;                    -- synchronization
  CPU_IOC <= false;                             -- can return to cpu_io
  IOWAITC <= '0';                               -- allow state changes
  RDYC <= '0' after GDEL,                       -- delay for synch
          '1' after PER/2 + GDEL;
  wait for ODEL;

  end process CPU_IO_PROC;
--*************************************************************************
-- Execute Process      (DES operation execution)
--*************************************************************************
  EXECUTE_PROC: process

        variable ARCS_IN_STAT : DWORD;          -- status for LPs in
        variable Next_Addr    : DWORD;          -- next RAM addr for r/w
                                                -- same as CPU_IO_PROC

--*************************************************************************
--  PROCEDURES AND FUNCTIONS required for EXECUTE process
```

```
--*************************************************************************
-- INTR_CPU_SEND procedure (output to CPU with Interrrupt Request)
--
-- Intr_CPU_Send pseudo/algorithm
--
-- Assert Interrupt request to CPU
--      busy wait for CPU to acknowledge request (i.e. control signals)
-- Register (DWORD) to send to CPU is put on data_bus
-- Ensure ready line "RDY" is high for CPU bus xfer cycle
-- Assert parallel I/O (write)
-- Pull ready line "RDY" low to end CPU bus xfer cycle
--*************************************************************************
   procedure Intr_CPU_Send (OUTPUT : in DWORD) is

        variable MSG_OUT : DWORD;

   begin

        MSG_OUT := OUTPUT;                          -- output to CPU
        BUFF_IOE <= MSG_OUT;
        INTR <= '1' after ODEL;
        wait until ((NPS1 = '0') and (NPS2 = '1') and (WR = '0'));
        INTR <= '0' after ODEL;
-----------------
-- output MSG_OUT
-----------------
        DATAout <= BUFF_IOE after ODEL,
           "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + RDEL;
        WTP <= '1' after ODEL;                      -- parallel i/o write
        wait for RDEL;
        WTP <= '0' after ODEL;
        RDYE <= '0' after ODEL,                     -- end bus cycle
                '1' after PER/2 + ODEL;

   end Intr_CPU_Send;
--*************************************************************************
-- SEND_CPU procedure (output to CPU without Interrupt Request)
--
-- Send_CPU pseudo/algorithm
--
-- Register (DWORD) to send to CPU is put on data_bus
-- Ensure ready line "RDY" is high for CPU bus xfer cycle
-- Assert parallel I/O (write)
-- Pull ready line "RDY" low to end CPU bus xfer cycle
--*************************************************************************
   procedure Send_CPU (OUTPUT : in DWORD) is

        variable MSG_OUT : DWORD;

   begin
```

```
        RDYE <= '1' after ODEL;                          -- bus cycle strb
        MSG_OUT := OUTPUT;                               -- output to CPU
        BUFF_IOE <= MSG_OUT;
        wait for FFDEL;
 ----------------
 -- output MSG_OUT
 ----------------
        DATAout <= BUFF_IOE after ODEL,
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + RDEL;
        WTP <= '1' after ODEL;                           -- parallel write
        wait for RDEL;
        WTP <= '0' after ODEL;
        RDYE <= '0' after ODEL,                          -- end bus cycle
                '1' after PER/2 + ODEL;

   end Send_CPU;
 --*************************************************************************
 -- SEND_NULL_MSG procedure (send "null_msg" to all output arcs)
 --
 -- SEND_NULL_MSG pseudo/algorithm
 --
 --*************************************************************************
  procedure Send_null_msg(RAM_Addr : DWORD; Outputs : INTEGER) is

        constant NULL_MSG : DWORD := "00000000000000000000000000000000";
        variable Next_Addr: DWORD;
        variable Num_out  : INTEGER;

  begin

    Next_Addr := RAM_Addr;                               -- LP's ARCS_OUT
    Num_out := Outputs;                                  -- # of arcs_out

    Send_Null_Loop:
    while (Num_out > 0) loop
-------------------------------------------------------------------
-- get next LP_OUT from RAM (TO_LP for "null_msg") & store (Reg_32(7))
-------------------------------------------------------------------
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;     -- addr for next LP_OUT
        RWE <= '0' after ODEL;                           -- RAM read cycle
        IOE <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOE <= DATAin after FFDEL;                  -- next LP_OUT
        wait for FFDEL;
        Reg_32E(7) <= BUFF_IOE after FFDEL;              -- next TO_LP
        IOE <= '0' after ODEL;
        RWE <= 'Z' after ODEL + GDEL;                    -- must be after IO
        MAE <= "ZZZZZZZZZZZ" after ODEL;
        wait for ODEL;
-----------------------------------------------------------
-- decrement Next_Addr in LP_RAM_Partition (reading top down)
```

```
-------------------------------------------------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE
                        (Next_Addr))) - 1)));
        wait for ALUDEL;
-----------------------------------
-- output TO_LP for "null_msg" to CPU
-----------------------------------
        RDYE <= '1' after ODEL;
        wait for ODEL;
        Intr_CPU_Send(Reg_32(7));                    -- out_node|out_LP
        wait until READY0 = '0';
-------------------------------------
-- output FROM_LP for "null_msg" to CPU
-------------------------------------
        RDYE <= '1' after ODEL;
        wait for ODEL;
        Intr_CPU_Send(Reg_32(1));                    -- FROM_LP (this LP)
        wait until READY0 = '0';
-------------------------------------------------
-- output SAFE_LOOKAHEAD_TIME for "null_msg" to CPU
-------------------------------------------------
        RDYE <= '1' after ODEL;
        wait for ODEL;
        Intr_CPU_Send(Reg_32(8));                    -- load Safe_Time
        wait until READY0 = '0';                     -- delay to next out
-----------------------------------
-- output NULL_MSG (null value) to CPU
-----------------------------------
        RDYE <= '1' after ODEL;
        wait for ODEL;
        Intr_CPU_Send(NULL_MSG);
        wait until READY0 = '0';

    Num_out := Num_out - 1;                          -- next output LP
    end loop Send_Null_Loop;

  end Send_null_msg;
--*****************************************************************************
-- INIT_SIM procedure (initialize simulation)
--
-- Init_Sim pseudo code
--
-- GP registers loaded during CPU_IO_PROC
        -- REG_32-1 <= TO_LP                         -- initialize "this" LP
        -- REG_32-2 <= TO_LP(DELAY)                  -- this LPs delay
        -- REG_32-3 <= # ARCS_IN
        -- REG_32-4 <= # ARCS_OUT
        -- RAM_XXXX <= IN_1_NODE# | IN_1_LP#         -- (2) data in (1) reg
        --                                           -- add'l arcs_in
        -- RAM_XXXX <= OUT_1_NODE# | OUT_1_LP#       -- (2) data in (1) reg
        --                                           -- add'l arcs_out
```

```
--
-- assert clear/reset for CAM storage                    -- assert m_clr
-- reset and store sim_clock for lp_id                   -- DWORD
        -- REG_32-5 reserved for Local_Sim_Time          -- local sim clock
-- calculate min_safe_time (sim_clock + LP_delay)        -- dynamic
-- send null_msg to all ARCS_OUT LPs                     -- xmit bus cycles
        -- REG_32-6 <= Next_Addr (next free @ top of LP RAM partition)
-- store LP significant data (regs) in LP_RAM_Partion (Reg_32(1--4))
--*************************************************************************
    procedure Init_Sim is                                 -- init LPs for sim

        variable NUM_OUT : INTEGER;

    begin

    NUM_OUT :=  BVtoI(MVL7VtoBV(CHANGE(Reg_32(4))));      -- # ARCS_OUT (loop indx)

    RST <= '0' after ODEL,                                -- mstr_clr for CAM
           '1' after PER/2 + ODEL;

    Next_Addr := Reg_32(6);                               -- saved in CPU_IO_PROC
---------------------------
-- initialize the FLAGS reg
---------------------------
    FLAGSE(11 downto 0) <= "010000111110";                -- tail of RAM queue
                                                          -- and CAM "full" stat
    FLAGSE(22 downto 12) <= "01000011111";                -- head of RAM queue
    FLAGSE(28 downto 23) <= "000000";                     -- # events in RAM
-----------------------------------------
-- reset LP's simulation clock (i.e. start)
-----------------------------------------
    Reg_32E(5) <= "00000000000000000000000000000000" after FFDEL;
    wait for FFDEL;
---------------------------------------
-- add LP delay to sim_clk for safe_time
---------------------------------------
    ACC <= CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32E(5)))) +
                                  BVtoI(MVL7VtoBV(CHANGE(Reg_32(2)))))))
                                  after ALUDEL;
    wait for ALUDEL;                                      -- PER/2
    Reg_32E(8) <= ACC after FFDEL;                        -- safe_time reg
----------------------
-- top entry in LP RAM
----------------------
    Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE
                               (Next_Addr))) - 1)));
    wait for FFDEL;
-------------------------------------
-- LOOP for "null_msg" to all ARCS_OUT
-------------------------------------
    IOWAITE <= '1';                                       -- no state changes
```

```
    BUSYE <= '1' after ODEL;                                -- prompt for Test_Bench
    Send_null_msg(Next_Addr, NUM_OUT);
    RDYE <= '1' after ODEL;                                 -- ready for next proc
-------------------------------------------------------------------
-- store essential LP data (4 registers) for future operations
-- use LP_RAM_Partition starting address (base_addr) -> done earlier
-- align data in GP registers for RAM write LOOP
-- NOTE: Reg_32(2) already contains LP_Delay
-- Reg_32(3) : combine # ARCS_IN & # ARCS_OUT in single DWORD
-------------------------------------------------------------------
    Next_Addr := Reg_32(9);                        -- LP_RAM_Partition (base_addr)
    Reg_32E(1) <= "00000000000000000000000000000000" after FFDEL;
                                                   -- init ARCS_IN_STATUS
    Reg_32E(3) <= JOIN_DWORDS(Reg_32(3), Reg_32(4)) after ALUDEL;
    wait for ALUDEL;
    Reg_32E(4) <= Reg_32(5) after FFDEL;                    -- LP_Simulation_Time
-----------------------------------------
-- Loop to write registers to LP_RAM_Partition
-----------------------------------------
    IOWAITE <= '1';                                         -- no state changes
    SAVE_LP_DATA:
    for I in 1 to 4 loop
        BUFF_IOE <= Reg_32(I) after FFDEL;                  -- next reg to store
        wait for FFDEL;
        DATAout <= BUFF_IOE after FFDEL + ODEL,             -- put data on bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;        -- RAM address
        RWE <= '1' after 3*ODEL;                            -- RAM write cycle
        IOE <= '1' after ODEL;
        wait for WDEL;
        IOE <= '0' after ODEL;
        RWE <= 'Z' after ODEL + GDEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        wait for ODEL;                                      -- ensure valid M_addr
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE
                        (Next_Addr))) + 1)));
    end loop SAVE_LP_DATA;
    IOWAITE <= '0';                                         -- allow state chgs
    BUSYE <= '0' after ODEL;                                -- prompt for Test_Bench

    end Init_Sim;
--*************************************************************************
-- POST_MSG procedure (process received event/null messages)
--
-- Post_Msg pseudo code
--
-- CPU_IO_PROC has loaded the following registers
        -- REG_32-1 <= TO_LP                    -- rcvd @ this LP
        -- REG_32-2 <= IN_NODE # | IN_LP #      -- FROM_LP
        -- REG_32-3 <= EVENT TIME_TAG
        -- REG_32-4 <= MEMR_PTR to msg          -- null_msg : ptr = 0
```

```
-- load GP registers with TO_LP essential data from RAM -- read (4*DWORD)
-- update ARCS_IN status (toggle FROM_LP bit)
-- check CAM_FULL bit in FLAGS register
--    if not FULL -> store message & update FLAGS reg
--      (TO_LP, FROM_LP, TIME_TAG, MEMR_PTR)
--      update CAM_FULL bit in FLAGS reg by receiving CAM status msg
--    else -> store in RAM temporarily -> START/IDLE will handle
--*************************************************************************
    procedure Post_Msg is                                   -- rcv EVENT/NULL msg

        constant Last_Q_Addr : DWORD := "00000000000000000000000110111111";
        variable STAT_BIT  : INTEGER := 0;                  -- array index bit
                                                            -- for ARCS_IN_STAT
    begin
-------------------------------------------------------------------------
-- load LP_GP_working registers with essential data from RAM Partition
-- must get pointer to LP_RAM_Partition base_addr first
-------------------------------------------------------------------------
    IOWAITE <= '1';                                         -- no state changes
    BUSYE <= '1' after ODEL;                                -- prompt for Test_Bench
    MAE <= DWORD_to_MADD(Reg_32(1)) after MADEL;            -- part tbl ptr to
                                                            -- base addr
    RWE <= '0' after ODEL;                                  -- RAM read cycle
    IOE <= '1' after ODEL;
    wait for RDEL;
    BUFF_IOE <= DATAin after FFDEL;                         -- LP RAM base_addr
    wait for FFDEL;
    RWE <= 'Z' after ODEL + GDEL;                           -- after IOE
    IOE <= '0' after ODEL;
    MAE <= "ZZZZZZZZZZZZ" after ODEL;
    Reg_32E(5) <= BUFF_IOE after FFDEL;                     -- temp for base_addr
    Next_Addr := BUFF_IOE;                                  -- increment in loop
    wait for ODEL;
-------------------------------------------------------------------------
-- loop to read RAM and load LP_GP registers
        -- Reg_32(6) <= ARCS_IN_STAT    Reg_32(8) <= # ARCS_IN | # ARCS_OUT
        -- Reg_32(7) <= LP_DELAY        Reg_32(9) <= LP_Simulation_Time
-------------------------------------------------------------------------
    LOAD_REG_LOOP:
    for I in 6 to 9 loop                                    -- 6 to 9: indx regs
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;        -- RAM address
        RWE <= '0' after ODEL;                              -- RAM read cycle
        IOE <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOE <= DATAin after FFDEL;                     -- LP data
        wait for FFDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        Reg_32E(I) <= BUFF_IOE after FFDEL;                 -- load LP data
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
```

```
        wait for ODEL;
    end loop LOAD_REG_LOOP;
-------------------------------------------------------------------
-- update ARCS_IN_STATUS (i.e. set FROM_LP bits (9->0) high)
--
-- Read ARCS_IN addresses (loop 0 to (# ARCS_IN - 1) times)
--   base_addr + offset (4 ess. data) in RAM Partition to LP_IN
-- Compare FROM_LP address to LP_IN addresses (same loop iteration)
--   find match then EXIT
-- Set ARCS_IN_STAT bit (bit # to set = loop iteration #)
--
-- start of ARCS_IN addresses in LP_RAM_Partition
-------------------------------------------------------------------
    Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(5)))) + 4)));

    ARCS_Loop:
    loop
      MAE <= DWORD_to_MADD(Next_Addr) after MADEL;      -- RAM address
      RWE <= '0' after ODEL;                            -- RAM read cycle
      IOE <= '1' after ODEL;
      wait for RDEL;
      BUFF_IOE <= DATAin after FFDEL;                   -- IN_LP id
      wait for FFDEL;
      RWE <= 'Z' after ODEL + GDEL;
      IOE <= '0' after ODEL;
      MAE <= "ZZZZZZZZZZZZ" after ODEL;
      wait for ODEL;
      if (Reg_32(2) = BUFF_IOE) then
         MATCH <= true;                                 -- exit: IN_LP=FROM_LP
         exit;
      end if;
      Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
      STAT_BIT := STAT_BIT + 1;                         -- ARCS_IN_STAT(X)
    end loop ARCS_Loop;

    wait on MATCH until MATCH;                          -- find FROM_LP in RAM
    Reg_32E(6)(STAT_BIT) <= '1';                        -- Set STATUS_BIT
    MATCH <= false;                                     -- reset MATCH signal
    wait for FFDEL;
-------------------------------------------------------------------
-- Update ARCS_IN_STATUS stored in LP's RAM partition base_addr
-------------------------------------------------------------------
    DATAout <= Reg_32(6) after FFDEL + ODEL,
            "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
    MAE <= DWORD_to_MADD(Reg_32(5)) after MADEL;        -- base_addr
    RWE <= '1' after 3*ODEL;                            -- RAM write cycle
    IOE <= '1' after ODEL;
    wait for WDEL;
    RWE <= 'Z' after ODEL + GDEL;
    IOE <= '0' after ODEL;
    MAE <= "ZZZZZZZZZZZZ" after ODEL;
```

```
    wait for ODEL;
-------------------------------------------------------------------------
-- post message -> store event/null in CAM
--
-- CAM event has (5) fields |TO_LP|FROM_NODE_#|FROM_LP_#|TIME_TAG|MEM_PTR|
--      filter first (3) fields for compact representation
--      - TO_LP (32 bits)       -> 5 bits  (20 LPs/node max)
--      - FROM_NODE_# (16 bits) -> 3 bits  (8 nodes max in cube)
--      - FROM_LP_# (16 bits)   -> 5 bits  (20 LPs/node max)
-- (3) CAM write cycles: (13 'valid' bits), (32 bits), (32 bits)
--
-- !!!! IF CAM IS NOT FULL -> STORE EVENT !!!!
-- else store temporarily in RAM and let START/IDLE handle
-------------------------------------------------------------------------
    Reg_32E(2) <= MAP_FIELDS(Reg_32(1), Reg_32(2));     -- compact event_id
    wait for ALUDEL;
-------------------------------------------------------------
-- CAM write loop (event_id, event time_tag, memr_ptr to msg)
-------------------------------------------  ----------------
    if FLAGS(0) = '0' then                                -- CAM is NOT full
      CAM_SAVE_EVENT:
      for I in 2 to 4 loop                                -- use GP reg indx
        BUFF_IOE <= Reg_32(I) after FFDEL;                -- next event field
        wait for FFDEL;
        DATAout <= BUFF_IOE after ODEL,                   -- put data on bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
        wait for ODEL;
        NEE <= '0' after ODEL;
        WTCE <= '1' after ODEL;
        wait for WDEL;
        WTCE <= '0' after ODEL;
        NEE <= 'Z' after ODEL;
        wait for ODEL;
      end loop CAM_SAVE_EVENT;
-----------------------------------------------------------------------
-- Update CAM_FULL bit (FLAGS(0)) with automatic reading of CAM status
-----------------------------------------------------------------------
      RDCE <= '1' after ODEL;                             -- CAM read
      wait for RDEL;
      BUFF_IOE <= DATAin after FFDEL;
      RDCE <= '0' after ODEL;
      wait for ODEL;
      if BUFF_IOE(0) = '1' then                           -- CAM is full
        FLAGSE(0) <= '1';
      end if;                                             -- else NOT full
    elsif FLAGS(0) = '1' then                             -- CAM is full
-------------------------------------------------------------
-- Store Post_Msg in RAM (temporarily): START/IDLE will get out
-- Tail of RAM queue is in FLAGS(11 downto 1)
-------------------------------------------------------------
      Next_Addr(10 downto 0) := FLAGS(11 downto 1);       -- Tail of RAM queue
```

```
      RAM_SAVE_EVENT:                                          -- store in RAM
      for I in 2 to 4 loop                                     -- use GP reg indx
        BUFF_IOE <= Reg_32(I) after FFDEL;                     -- next event field
        wait for FFDEL;
        DATAout <= BUFF_IOE after FFDEL + ODEL,                -- put data on bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;           -- RAM address
        RWE <= '1' after 3*ODEL;                               -- RAM write cycle
        IOE <= '1' after ODEL;
        wait for WDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) - 1)));
        wait for ODEL;
      end loop RAM_SAVE_EVENT;
-------------------------------------------
-- Update FLAGS to show events stored in RAM
-------------------------------------------
      FLAGSE(23) <= '1';
---------------------------------------
-- Update RAM queue tail address in FLAGS
---------------------------------------
      if Next_Addr = Last_Q_Addr then                          -- circular Q
        FLAGSE(11 downto 1) <= "01000011111";                  -- back to top of Q
      else
        FLAGSE(11 downto 1) <= Next_Addr(10 downto 0);         -- incremented by 1
      end if;
    end if;
    IOWAITE <= '0';                                            -- allow state change
    BUSYE <= '0' after ODEL;                                   -- prompt for Test_Bench

    end Post_Msg;
--*****************************************************************************
-- GET_EVENT procedure (get NEXT_EVENT for specified LP)
--
-- Get_Event pseudo code
--
-- read lp_id (TO_LP stored in Reg_32(1) from cpu_io_proc)
-- if ARCS_IN for lp_id satisfied (msg on all ARCS_IN)
--   assert rdc (read cam - next lp_id event)
--      - update ARCS_IN register
--      - update lp_id sim_clk (sim_clk + time_tag)
--      - read event memr_ptr
--      - update CAM (reset available bit)
--      - send NEXT_EVENT to CPU                              -- xmit bus cycles
--      if NEXT_EVENT = NULL
--         send null with updated sim_time to all output arcs
--         do Get_Event again
-- else
--   send wait signal to CPU (no next_event)                  -- xmit bus cycles
```

```
--*************************************************************************
    procedure Get_Event is                          -- get NEXT_EVENT from CAM

        constant WAIT_MSG    : SYS_BUS :=
                               "11110000111100001111000011110000";
        variable NUMARCS_IN  : INTEGER;
        variable NUMARCS_OUT : INTEGER;
        variable HAVE_EVENT  : BOOLEAN := false; -- consider FLAGS bit (31-29)
        variable HAD_NULL    : BOOLEAN := false; -- consider FLAGS bit (31-29)
        variable RAM_ADDR    : DWORD;           -- start of arcs_out for null
        variable STAT_BIT    : INTEGER := 0;    -- ARCS_IN_STAT(#)

    begin
-------------------------------------------------------------------------
-- THIS IS WHERE WE ADVANCE THE LOCAL SIMULATION CLOCK!!!
-- CONTIGENT UPON HAVING A NEXT EVENT/NULL MESSAGE!!!
--
-- load LP_GP_working registers with essential data from RAM Partition
-- must get pointer to LP_RAM_Partition first
-------------------------------------------------------------------------
    IOWAITE <= '1';                                     -- no state changes
    BUSYE <= '1' after ODEL;
    MAE <= DWORD_to_MADD(Reg_32(1)) after MADEL;        -- part tbl ptr to
                                                        -- base addr
    RWE <= '0' after ODEL;                              -- RAM read cycle
    IOE <= '1' after ODEL;
    wait for RDEL;
    BUFF_IOE <= DATAin after FFDEL;                     -- LP RAM base_addr
    wait for FFDEL;
    RWE <= 'Z' after ODEL + GDEL;
    IOE <= '0' after ODEL;
    MAE <= "ZZZZZZZZZZZZ" after ODEL;
    Reg_32E(2) <= BUFF_IOE;                             -- temp for base_addr
    Next_Addr := BUFF_IOE;                              -- increment in loop
-------------------------------------------------------------------------
-- loop to read RAM and load LP_GP registers
        -- Reg_32(3) <= ARCS_IN_STAT    Reg_32(5) <= # ARCS_IN | # ARCS_OUT
        -- Reg_32(4) <= LP_DELAY        Reg_32(6) <= LP_Simulation_Time
-------------------------------------------------------------------------
    LOAD_REG_LOOP:
    for I in 3 to 6 loop                                -- 3 to 6: indx regs
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;    -- RAM address
        RWE <= '0' after ODEL;                          -- RAM read cycle
        IOE <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOE <= DATAin after FFDEL;                 -- LP data
        wait for FFDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        Reg_32E(I) <= BUFF_IOE after FFDEL;             -- load LP data
```

```
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
 --------------------------------
 -- determine number of input ARCS
 --------------------------------
        if I = 5 then
          wait for 2*FFDEL;                                  -- load Reg_32(5)
          NUMARCS_IN := BVtoI(MVL7VtoBV(CHANGE(Reg_32(I)(31 downto 16))));
        end if;
        if I = 6 then
          LOADED <= true;                                    -- GP Regs loaded
        end if;
    end loop LOAD_REG_LOOP;
 ------------------------------------------------------------------------
 -- check for ARCS_IN satisfied (logical AND of ARCS_IN_STAT input bits)
 ------------------------------------------------------------------------
    wait on LOADED until LOADED;                             -- GP Regs loaded
    LOADED <= false;                                         -- reset flag
    for I in 0 to (NUMARCS_IN - 1) loop
      if Reg_32(3)(I) = '0' then                             -- NO next_event
        HAVE_EVENT := false;
        exit;
      elsif Reg_32(3)(I) = '1' then                          -- still checking
        HAVE_EVENT := true;
      end if;

      assert (Reg_32(3)(I) = '0' or Reg_32(3)(I) = '1')
      report "Invalid ARCS_IN_STAT";
    end loop;
    wait for ALUDEL;                                         -- do bit check above
 ------------------------------------------------------------------------
 --  either tell CPU to "wait -> no next_event" or get next_event from CAM
 --  tell CAM "who" needs Next_Event
 ------------------------------------------------------------------------
    if HAVE_EVENT then                                       -- CAM for next_event
      BUFF_IOE <= Reg_32(... after FFDEL;                    -- tell CAM for "who"
      wait for FFDEL;
      DATAout <= BUFF_IOE after ODEL,
             "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
      wait for ODEL;
      NEE <= '1' after ODEL;                                 -- init Next_Event
      WTCE <= '1' after ODEL;                                -- CAM write cycle
      wait for WDEL;
      WTCE <= '0' after ODEL;
      NEE <= 'Z' after ODEL;
 -------------------------
 -- read Next_Event from CAM
 -------------------------
      READ_CAM_LOOP:                                         -- read 3*DWORD
      for I in 7 to 9 loop                                   -- Reg_32(Indx)
        RDCE <= '1' after ODEL;                              -- CAM read cycle
        wait for RDEL;
```

```
        BUFF_IOE <= DATAin after FFDEL;
        wait for FFDEL;
        RDCE <= '0' after ODEL;
        Reg_32E(I) <= BUFF_IOE after FFDEL;
        wait for ODEL;
      end loop READ_CAM_LOOP;
      LOADED <= true;                                    -- have next event
      HAD_EVENT <= true;
------------------
-- tell CPU to wait
------------------
    else
      Send_CPU(WAIT_MSG);                                -- output WAIT_MSG
    end if;

-- REMAINING CODE EXECUTED ONLY WHEN CAM HAS NEXT_EVENT (including "nulls")
--*************************************************************************
-- update ARCS_IN_STAT -> conditional on ADD'L_LP_EVENT_PENDING bit
-- bit 31 of first DWORD returned from CAM: IF = '1'-> add'l events pending
-- and no update of ARCS_IN_STAT is required
--
-- IF UPDATE: must determine which INPUT_LP provided next_event
-- read LP_IDs of INPUT_ARCS from LP_RAM_Partition and compare with
-- the FROM_LP field of next_event from CAM
--*************************************************************************
  wait on HAD_EVENT until HAD_EVENT;
  if (not LOADED'stable and LOADED) then

     LOADED <= false;                                    -- reset flag
     HAD_EVENT <= false;                                 -- reset flag
     FLAGSE(0) <= '0';                                   -- got event -> not FULL
-------------------------------------------------------------------------
-- unmask TO & FROM_LP id from CAM next_event (Reg_32(7)(12 downto 0))
-------------------------------------------------------------------------
     for I in 0 to 4 loop
       Reg_32E(10)(I) <= Reg_32(7)(I);                   -- FROM_LP #
     end loop;

     for I in 5 to 15 loop
       Reg_32E(10)(I) <= '0';                            -- rest of FROM_LP #
     end loop;

     for I in 5 to 7 loop
       Reg_32E(10)(I+11) <= Reg_32(7)(I);                -- FROM_LP node #
     end loop;

     for I in 19 to 31 loop
       Reg_32E(10)(I) <= '0';                            -- rest of FROM_LP node #
     end loop;

   wait for ALUDEL;                                      -- above bit twiddle
```

```
        for I in 0 to 4 loop
          Reg_32E(7)(I) <= Reg_32(7)(I+8);              -- TO_LP
        end loop;

        for I in 5 to 30 loop                           -- 31 is stat bit
          Reg_32E(7)(I) <= '0';                         -- rest of TO_LP
        end loop;

        Reg_32E(7)(31) <= Reg_32(7)(31);

      wait for ALUDEL;                                  -- above bit twiddle

      if (Reg_32(7)(31) = '0') then                     -- DO UPDATE
 ----------------------------------------
 -- start of ARCS_IN ids in LP_RAM_Partition
 ----------------------------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(2)))) + 4)));

        ARCS_Loop:
        loop
          MAE <= DWORD_to_MADD(Next_Addr) after MADEL;  -- RAM address
          RWE <= '0' after ODEL;                        -- RAM read cycle
          IOE <= '1' after ODEL;
          wait for RDEL;
          BUFF_IOE <= DATAin after FFDEL;               -- IN_LP id
          wait for FFDEL;
          RWE <= 'Z' after ODEL + GDEL;
          IOE <= '0' after ODEL;
          MAE <= "ZZZZZZZZZZZZ" after ODEL;
          if (Reg_32(10) = BUFF_IOE) then               -- IN_LP=FROM_LP?
            MATCH <= true;
            exit;
          end if;
          Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
          STAT_BIT := STAT_BIT + 1;                     -- ARCS_IN_STAT(#)
          wait for ODEL;
        end loop ARCS_Loop;
        wait on MATCH until MATCH;
        Reg_32E(3)(STAT_BIT) <= '0';                    -- Set STATUS_BIT
        MATCH <= false;                                 -- reset flag

      end if;             -- if (Reg_32(7)(31) = '0')   -- DO UPDATE
 ------------------------------------------------------------------
 -- update LP_Sim_Time -> jump local clock to next_event scheduled time
 ------------------------------------------------------------------
      Reg_32E(6) <= Reg_32(8) after FFDEL;              -- update LP_SIM_Time
      wait for FFDEL;
 ------------------------------------------------------------
 -- check next_event for "null": IF NULL -> do Get_Event again
 -- Also: build "null_msg" for later transmission to arcs_out
```

```
--      safe_look_ahead = Reg_32(8) = sim_time + lp_delay
--      number of output arcs = Reg_32(5)(15 downto 0)
--      ARCS_OUT in RAM = base_addr + (sum arcs in/out) + 3)
-------------------------------------------------------------
    if (BVtoI(MVL7VtoBV(CHANGE(Reg_32(9)))) = 0) then   -- Next_Event = NULL
      HAD_NULL := true;                                 -- send null_msg later
      ACC <= CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(6)))) +
                                    BVtoI(MVL7VtoBV(CHANGE(Reg_32(4)))))))
                                    after ALUDEL;
      wait for ALUDEL;
      Reg_32E(8) <= ACC after FFDEL;                    -- null safe_time
      NUMARCS_OUT := BVtoI(MVL7VtoBV(CHANGE(Reg_32(5)(15 downto 0))));
      wait for ALUDEL;
      ACC <= CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(2)))) +
                                    BVtoI(MVL7VtoBV(CHANGE(Reg_32(5)))) +
                                    3))) after ALUDEL;
      wait for ALUDEL;
      RAM_ADDR := ACC;                                  -- ARCS_OUT in LP RAM
      if (Reg_32(7)(31) = '1') then                     -- multiple events
        Get_Event;                                      -- try again
      else
        Send_CPU(WAIT_MSG);
        wait until READY0 = '0';
-- ******************** MUST SEND NULL MESSAGES OUT TOO!!! ********************
--      using RAM part base_addr
--              get number of ARCS_OUT from RAM
--              loop through ARCS_OUT and send "null_msg"
--              end loop SEND_NULLS;
--
--    During subsequent "Get_Event" in above "if" we retrieved a "real"
--    next_event --> we currently update the sim_time and pass the "real"
--    event to the CPU.  Must deal with the fact that a "null" was processed
--    first --> i.e., must send "nulls" to all output arcs to update
--    LP safetimes for the rest of the simulation.
--*****************************************************************************
       end if;
    elsif (BVtoI(MVL7VtoBV(CHANGE(Reg_32(9)))) > 0) then  -- send CPU Next_Event
-----------------------
-- TO_LP for Next_Event
-----------------------
      Reg_32E(7)(31) <= '0' after FFDEL;                -- clear multiple bit
      BUFF_IOE <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after FFDEL;
      wait for ODEL;
      Send_CPU(Reg_32(7));                              -- send TO_LP
      wait until READY0 = '0';
-------------------------
-- FROM_LP for Next_Event
-------------------------
      BUFF_IOE <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after FFDEL;
      wait for ODEL;
      Send_CPU(Reg_32(10));                             -- load FROM_LP
```

```
------------------------
-- Memr_PTR for Next_Event
------------------------
      wait until READY0 = '0';
      wait for ODEL;
      BUFF_IOE <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after FFDEL;
      Send_CPU(Reg_32(9));                                  -- load MEM_PTR
      wait until READY0 = '0';

    end if;     -- if Next_Event = NULL
-------------------------------------------------------------------------
--  UPDATE LP's RAM Partition with new ARCS_IN_STATUS and new SIM_TIME
--  currently in Reg_32(3) and Reg_32(6) respectively & base_addr = Reg_32(2)
-------------------------------------------------------------------------
      Next_Addr := Reg_32(2);                               -- RAM Part base_addr
      wait for FFDEL;
      RAM_UPDATE:                                           -- update RAM partition
      for I in 1 to 2 loop                                  -- use GP reg indx
        BUFF_IOE <= Reg_32(I*3) after FFDEL;                -- next event field
        wait for FFDEL;
        DATAout <= BUFF_IOE after FFDEL + ODEL,             -- put data on bus
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 2*ODEL + WDEL;
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;        -- RAM address
        RWE <= '1' after 3*ODEL;                            -- RAM write cycle
        IOE <= '1' after ODEL;
        wait for WDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZ" after ODEL;
        wait for ODEL;
-------------------------------------------------------------------
-- offset address by '3' for SIM_TIME location in LP's RAM partition
-------------------------------------------------------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 3)));
        if HAD_NULL then
          HAD_NULL := false;
          SEND_NULL <= true after WDEL + FFDEL;             -- finish RAM_UPDATE
        end if;
      end loop RAM_UPDATE;
-----------------------------------------------------------
-- if Get_Event was "null" we now send null_msg to all arcs_out
-----------------------------------------------------------
      if (not SEND_NULL'stable and SEND_NULL) then
        Send_null_msg(RAM_ADDR, NUMARCS_OUT);
      end if;

    end if;     -- if HAVE_EVENT
    IOWAITE <= '0';
    BUSYE <= '0' after ODEL;

  end Get_Event;
```

```
--*************************************************************************
-- POST_EVENT procedure (send EVENT &/or NULL messages)
--
-- Post_Event pseudo code
--
-- Registers loaded during CPU_IO_PROC
        -- REG_32-1 <= TO_LP (will be sender)          -- LP with event
        -- REG_32-2 <= MEMR_PTR to event               -- with CPU result
        -- REG_32-3 <= OUT_NODE # | OUT_LP #           -- TO_LP for msg
-- update event_time (SIM_TIME + delay)                -- add DWORD
-- read ARCS_OUT reg (send output to)
-- if ARCS_OUT = TO_LP  (cycle all out arcs)
--   send EVENT_MSG
--      - (from_lp, to_lp, time_tag, event)           -- 4*DWORD
-- else
--   send NULL_MSG
--      - (from_lp, to_lp, safe_time)                 -- 3*DWORD
--*************************************************************************
    procedure Post_Event is                          -- send EVENT/NULL msg

        variable NUMARCS_OUT : INTEGER;

    begin
-------------------------------------------------------------------------
-- load LP_GP_working registers with essential data from RAM Partition
-- must get pointer to LP_RAM_Partition first
-------------------------------------------------------------------------
    IOWAITE <= '1';                                  -- no state changes
    MAE <= DWORD_to_MADD(Reg_32(1)) after MADEL;     -- part tbl ptr to
                                                     -- base addr
    RWE <= '0' after ODEL;                           -- RAM read cycle
    IOE <= '1' after ODEL;
    wait for RDEL;
    BUFF_IOE <= DATAin after FFDEL;                  -- LP RAM base_addr
    wait for FFDEL;
    RWE <= 'Z' after ODEL + RDEL;
    IOE <= '0' after ODEL + RDEL;
    MAE <= "ZZZZZZZZZZZ" after ODEL;
    Reg_32E(4) <= BUFF_IOE;                          -- temp for base_addr
-------------------------------------------------------------------------
-- don't need ARCS_IN_STATUS -> Next_Addr := base_addr + (1) offset
-------------------------------------------------------------------------
    Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(BUFF_IOE))) + 1)));
    wait for ALUDEL;                                 -- time to increment
-------------------------------------------------------------------------
-- loop to read RAM and load LP_GP registers
        -- Reg_32(5) <= LP_DELAY        Reg_32(7) <= LP_Simulation_Time
        -- Reg_32(6) <= # ARCS_IN | # ARCS_OUT
-------------------------------------------------------------------------
    LOAD_REG_LOOP:
    for I in 5 to 7 loop                             -- 5 to 7: indx regs
```

```
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;     -- RAM address
        RWE <= '0' after ODEL;                           -- RAM read cycle
        IOE <= '1' after ODEL;
        wait for RDEL;
        BUFF_IOE <= DATAin after FFDEL;                  -- IP data
        wait for FFDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        Reg_32E(I) <= BUFF_IOE;                          -- load LP data
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) + 1)));
        wait for FFDEL;
    end loop LOAD_REG_LOOP;
-------------------------------------------------------------
-- Calculate event/null_msg time_tag (LP_Sim_Time + LP_Delay)
-------------------------------------------------------------
    Reg_32E(8) <= CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(7)))) +  -- Time_Tag
                                BVtoI(MVL7VtoBV(CHANGE(Reg_32(5)))))));
    wait for ALUDEL;

-------------------------------------------------------------------------
-- Output event/null_msg loop with # ARCS_OUT iterations
        -- # ARCS_OUT is LO_WORD of Reg_32(6)->(15 downto 0)
-- Calculate base_addr + offset for OUT_LP_ARC entries in RAM
        -- already have base_addr in Reg_32(4)
        -- offset = tot # IN/OUT ARCS + 3 (4 - essential LP data 0:3)
        -- Next_Addr := Reg_(4) + sum(HI & LO WORDS in Reg_32(6)) + 3
-- Read LP OUT_ARC from RAM and send EVENT/NULL_msg as required
-------------------------------------------------------------------------

    Reg_32E(9) <= HI_LO_ADD(Reg_32(6));                  -- sum # IN/OUT ARCS
    NUMARCS_OUT := BVtoI(MVL7VtoBV(CHANGE(Reg_32(6)(15 downto 0))));
    wait for ALUDEL;

    ACC <= CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Reg_32(4)))) + -- LP_Part base_addr
                           BVtoI(MVL7VtoBV(CHANGE(Reg_32(9)))) +
                           3)));                                                -- (4) reserved RA
    wait for ALUDEL;
    Next_Addr := ACC;
-------------------------------------------------------------
-- LOOP through OUT_ARCS in RAM, read & send EVENT or NULL_msg
-------------------------------------------------------------
    IOWAITE <= '1';
    Send_Msg_Loop:
    while NUMARCS_OUT > 0 loop
-------------------------------------------------------------
-- get next LP_OUT from RAM (TO_LP for "msg") & store (Reg_32(9))
-------------------------------------------------------------
        MAE <= DWORD_to_MADD(Next_Addr) after MADEL;     -- addr for next LP_OUT
        RWE <= '0' after ODEL;                           -- RAM read cycle
        IOE <= '1' after ODEL;
```

```
        wait for RDEL;
        BUFF_IOE <= DATAin after FFDEL;                -- next LP_OUT
        wait for FFDEL;
        RWE <= 'Z' after ODEL + GDEL;
        IOE <= '0' after ODEL;
        MAE <= "ZZZZZZZZZZZZ" after ODEL;
        Reg_32E(9) <= BUFF_IOE after FFDEL;            -- temp for TO_LP
-----------------------------------------
-- decrement Next_Addr in LP_RAM_Partition
-----------------------------------------
        Next_Addr := CHANGE(BVtoMVL7V(ItoBV(BVtoI(MVL7VtoBV(CHANGE(Next_Addr))) - 1)));
        wait for ALUDEL;
----------------------
-- TO_LP output for msg
----------------------
        Intr_CPU_Send(Reg_32(9));                      -- TO_LP
        wait until READY0 = '0';                       -- end bus cycle
        RDYE <= '1' after ODEL;
------------------------
-- FROM_LP output for msg
------------------------
        Intr_CPU_Send(Reg_32(1));                      -- FROM_LP (this LP)
        wait until READY0 = '0';                       -- end bus cycle
        RDYE <= '1' after ODEL;
-------------------------------------------------------
-- TIME_TAG output for msg (safe_lookahead_time for null_msg)
-------------------------------------------------------
        Intr_CPU_Send(Reg_32(8));                      -- Time_Tag
        wait until READY0 = '0';                       -- end bus cycle
        RDYE <= '1' after ODEL;
-------------------------------------------------------
-- condition (Is TO_LP the designated LP_OUT for event?)
-- If YES -> send event else "null" -> loop to next OUT_ARC
-------------------------------------------------------
        if (Reg_32(3) = Reg_32(9)) then                -- send EVENT_MSG
          Intr_CPU_Send(Reg_32(2));                    -- PTR to EVENT
          wait until READY0 = '0';                     -- end bus cycle
          RDYE <= '1' after ODEL;
        else                                           -- send "null"
          Intr_CPU_Send("00000000000000000000000000000000");
          wait until READY0 = '0';
          RDYE <= '1' after ODEL;
        end if;
        NUMARCS_OUT := NUMARCS_OUT - 1;

    end loop Send_Msg_Loop;
    BUSYE <= '0' after ODEL;
    IOWAITE <= '0';                                    -- allow state changes

   end Post_Event;
--*************************************************************************
```

```
-- Decode procedure (Instr from CPU: M_IO='0'; A15='1'; A2='0')
-- Decode and call identified procedure for execution
--*************************************************************************
  procedure DECODE (signal Opcode : MVL7_VECTOR(2 downto 0)) is

  begin

    case Opcode is
        when "000" => Init_Sim;                  -- INITIALIZE SIMULATION
        when "001" => Post_Msg;                  -- POST_MSG (RCV from CPU)
        when "010" => Get_Event;                 -- GET_EVENT
        when "011" => Post_Event;                -- POST_EVENT (SEND to_LPs)
        when others => assert FALSE report "Invalid DES opcode"
                       severity FAILURE;
    end case;
  end DECODE;

  begin

    wait on EXECUTE until EXECUTE;
      DECODE(OPCODE(31 downto 29));              -- decode & exec DES funcs
      DONEE <= true;                             -- opcode executed
  end process EXECUTE_PROC;

--*************************************************************************
-- Process Output Multiplexing
--*************************************************************************
IOWAIT <=       IOWAITC when not IOWAITC'quiet else     -- WAIT for state change
                IOWAITE when not IOWAITE'quiet else
                IOWAITS when not IOWAITS'quiet else
                IOWAIT;

FLAGS <=        FLAGSE when not FLAGSE'quiet else       -- FLAGS register
                FLAGSS when not FLAGSS'quiet else
                FLAGS;

BUSYB <=        BUSYS when not BUSYS'quiet else         -- BUSY signal
                BUSYC when not BUSYC'quiet else
                BUSYE when not BUSYE'quiet else
                BUSYB;

BUSY <= BUSYB;

RDY <=  RDYC when not RDYC'quiet else                   -- end CPU bus cycle
        RDYE when not RDYE'quiet else
        RDY;

READYO <= RDY;

BMA <=  MAS when not MAS'quiet else                     -- RAM address
        MAC when not MAC'quiet else
```

```
        MAE when not MAE'quiet else
        BMA;

MA  <=  BMA;

BRW <=  RWS when not RWS'quiet else                    -- RAM Read/Write
        RWC when not RWC'quiet else
        RWE when not RWE'quiet else
        BRW;

RW  <=  BRW;

BIO <=  IOS when not IOS'quiet else                    -- RAM I/O
        IOC when not IOC'quiet else
        IOE when not IOE'quiet else
        BIO;

IO <=   BIO;

RDCB <= RDCS when not RDCS'quiet else                  -- CAM read
        RDCE when not RDCE'quiet else
        RDCB;

RDC <=  RDCB;

WTCB <= WTCS when not WTCS'quiet else                  -- CAM write
        WTCE when not WTCE'quiet else
        WTCB;

WTC <=  WTCB;

NEB <=  NES when not NES'quiet else                    -- CAM new event
        NEE when not NEE'quiet else
        NEB;

NE <=   NEB;

BUFF_IO <=      BUFF_IOC when not BUFF_IOC'quiet else  -- IO buffer register
                BUFF_IOE when not BUFF_IOE'quiet else
                BUFF_IOS when not BUFF_IOS'quiet else
                BUFF_IO;

CPU_IO <=       CPU_IOC when not CPU_IOC'quiet else    -- CPU IO state
                CPU_IOS when not CPU_IOS'quiet else
                CPU_IO;

DONE <= DONEC when not DONEC'quiet else                -- DONE flag
        DONEE when not DONEE'quiet else
        DONE;

Reg_32(1) <= Reg_32S(1) when not Reg_32S(1)'quiet else -- GP registers
```

```
                Reg_32C(1) when not Reg_32C(1)'quiet else
                Reg_32E(1) when not Reg_32E(1)'quiet else
                Reg_32(1);

 Reg_32(2) <= Reg_32S(2) when not Reg_32S(2)'quiet else
                Reg_32C(2) when not Reg_32C(2)'quiet else
                Reg_32E(2) when not Reg_32E(2)'quiet else
                Reg_32(2);

 Reg_32(3) <= Reg_32S(3) when not Reg_32S(3)'quiet else
                Reg_32C(3) when not Reg_32C(3)'quiet else
                Reg_32E(3) when not Reg_32E(3)'quiet else
                Reg_32(3);

 Reg_32(4) <= Reg_32S(4) when not Reg_32S(4)'quiet else
                Reg_32C(4) when not Reg_32C(4)'quiet else
                Reg_32E(4) when not Reg_32E(4)'quiet else
                Reg_32(4);

 Reg_32(5) <= Reg_32S(5) when not Reg_32S(5)'quiet else
                Reg_32C(5) when not Reg_32C(5)'quiet else
                Reg_32E(5) when not Reg_32E(5)'quiet else
                Reg_32(5);

 Reg_32(6) <= Reg_32S(6) when not Reg_32S(6)'quiet else
                Reg_32C(6) when not Reg_32C(6)'quiet else
                Reg_32E(6) when not Reg_32E(6)'quiet else
                Reg_32(6);

 Reg_32(7) <= Reg_32S(7) when not Reg_32S(7)'quiet else
                Reg_32C(7) when not Reg_32C(7)'quiet else
                Reg_32E(7) when not Reg_32E(7)'quiet else
                Reg_32(7);

 Reg_32(8) <= Reg_32S(8) when not Reg_32S(8)'quiet else
                Reg_32C(8) when not Reg_32C(8)'quiet else
                Reg_32E(8) when not Reg_32E(8)'quiet else
                Reg_32(8);

 Reg_32(9) <= Reg_32S(9) when not Reg_32S(9)'quiet else
                Reg_32C(9) when not Reg_32C(9)'quiet else
                Reg_32E(9) when not Reg_32E(9)'quiet else
                Reg_32(9);

Reg_32(10) <= Reg_32S(10) when not Reg_32S(10)'quiet else
                Reg_32C(10) when not Reg_32C(10)'quiet else
                Reg_32E(10) when not Reg_32E(10)'quiet else
                Reg_32(10);

end BEHAVIOR;
```

B.3 Parallel I/O Behavior

This appendix provides the source listing of the VHDL architectural behavior of the parallel I/O ports used in the DES coprocessor system. The behavior is taken from Armstrong's chip-level model of the Mark II processor (2:120-123). The parallel I/O component instantiation of the DES coprocessor does not use the interrupt line provided with this behavior.

```
-------------------------------------------------------
-- FILE:  par_port_beh.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE: Architectural BEHAVIOR of the PARALLEL PORT
-- REFERENCE: Chip-Level Modeling with VHDL
--           (James Armstrong pp. 120-123)
-------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;
use WORK.SYSTEM.all;

entity PAR is
  generic(GDEL, FFDEL, BUFDEL: TIME);
  port( DI: in DWORD;
        DO: out DWORD:= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        NDS1, DS2, MD, NCLR: in MVL7;
        STB: in MVL7;
        NINT: out MVL7);
end PAR;
-------------------------------------------------------

architecture BEHAVIOR of PAR is

  signal S0, S1, S2, S3: MVL7;
  signal SRQ: MVL7;
  signal Q, Q1, Q2: DWORD:= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

begin

  S: block (S1 = '1' and NCLR = '1')
  begin
        Q1 <= guarded DI after FFDEL;
        Q2 <= "00000000000000000000000000000000" after FFDEL when (NCLR='0')
        else Q2;
        DO <= Q after BUFDEL when (S3 = '1')
        else "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after BUFDEL;
  end block;

  S0 <= not NDS1 and DS2 after GDEL;
  S1 <= (S0 and MD) or (STB and not MD) after (2*GDEL);
  S2 <= S0 or not NCLR after GDEL;
  S3 <= S0 or MD after GDEL;

  SERVRQ: process (S2, STB)
  begin
    if (S2 = '0') then
      SRQ <= '1' after FFDEL;
    elsif (S2 = '1') and (not STB'STABLE) and (STB = '0') then
```

```
      SRQ <= '0' after FFDEL;
    else
      SRQ <= SRQ;
    end if;
  end process SERVRQ;


  NINT <= not SRQ nor S0 after GDEL;
  Q <= Q1 when not Q1'QUIET else
       Q2 when not Q2'QUIET else
       Q;

end BEHAVIOR;
```

B.4 RAM Memory Behavior

The RAM memory behavior is shown in this appendix. The basic operation follows that of an example RAM memory included with the Zycad VHDL system (32:10-51, 10-53). The behavior includes procedures for both read and write operations and is initialized with the RAM partition pointer table via a file read operation provided by the standard textio package.

```
-------------------------------------------------------
-- FILE:  ram_mem_beh.vhd
-- AUTHOR: JT, PH, GWH
-- PURPOSE: Architectural BEHAVIOR of the RAM_MEM
-- DATE:  27 Aug 91
-- SOURCE: ZYCAD User's Manual pp. 10-51 -- 10-53
-- HISTORY:  None
-------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;
use STD.TEXTIO.all;

entity RAM_MEM is
        generic(Ndata: Positive;                          -- # of data lines
                Naddr: Positive;                          -- # of addr lines
                RDEL, DISDEL: TIME);                      -- delay times
        port(DATAI: in DWORD;                             -- data in lines
             DATAO: out DWORD:=
                "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";      -- data out lines
             ADDR: in MVL7_VECTOR(Naddr-1 downto 0);      -- address lines
             CE: in MVL7;                                 -- chip enable (high)
             RW: in MVL7);                                -- read (low) and
                                                          -- write (high)
end RAM_MEM;
-------------------------------------------------------
architecture BEHAVIOR of RAM_MEM is
begin
---------------------------------------
-- assertion for changes in address lines
---------------------------------------
  Assertion: process
    begin
        assert not(RW='1' and CE = '1' and ADDR'EVENT)
           report "Address lines changed while RAM is Write Enabled"
           severity WARNING;
        wait on CE, RW, ADDR;
    end process Assertion;
-------------------
-- the memory model
-------------------
  P: process
     subtype ETYPE is DWORD;
     type MEMTYPE is array (Natural range <>) of ETYPE;
     variable m: MEMTYPE(0 to 2**ADDR'length-1);
     variable TEMP : BIT_VECTOR(31 downto 0);          -- for init only
     file RAMDEF : TEXT IS IN "ram_text";              -- memory define file
```

```
    variable STARTUP : BOOLEAN := true;                -- memory need init?
    variable L : LINE;                                 -- current input line from file
    variable J : INTEGER;                              -- RAM index

    procedure do_read is
    begin
       for i in ADDR'RANGE loop
         if ADDR(i) = 'X' or ADDR(i) = 'Z' then
            DATAO <= (others => 'Z');
            return;
         end if;
       end loop;
       DATAO <= m(BVtoI(MVL7VtoBV(ADDR)));
    end do_read;

    procedure do_write(data: ETYPE) is
    begin
       for i in ADDR'RANGE loop
         if ADDR(i) = 'X' or ADDR(i) = 'Z' then
               assert false
               report "Attempted write to bad RAM address"
               severity WARNING;
            DATAO <= (others => 'Z');
            return;
         end if;
       end loop;
       m(BVtoI(MVL7VtoBV(ADDR))) := data;
       DATAO <= data;
    end do_write;

begin
   -- The variable m and the port DATAO are initialized correctly
   -- at elaboration time.  This makes it best to wait at the top
   -- of the process.

   if STARTUP then                                     -- Initialize the RAM only once
      for J in 0 to 19 loop
        readline(RAMDEF, L);
        read(L, TEMP);
        m(J) := CHANGE(BVtoMVL7V(TEMP));               -- Loading RAM
      end loop;
      STARTUP := false;
   end if;

   wait on CE, RW, DATAI, ADDR;

   if (CE = '1') then
      if DATAI'EVENT and RW = '1' then
         do_write(DATAI);
      elsif ADDR'EVENT and RW /= 'X' and RW /= 'Z' then
         if RW = '1' then
```

```
            do_write(DATAI);
          else
            do_read;
          end if;
        elsif RW'EVENT then
          if RW = '1' then
            do_write(DATAI);
          elsif RW = '0' then
            do_read;
          else
            do_write((DATAI'RANGE => 'X'));
          end if;
        end if;
      else
        DATAO <= (others => 'Z');
      end if;
    end process P;

end BEHAVIOR;
```

B.5 CAM Memory Behavior

The source code listing for the behvior of the CAM memory, described in Section 4.3.4.3, is included in this appendix. A single CAM process is regulated by control signals from the DES coprocessor. The CAM's operation is sensitive to both read and write control lines, which are constantly monitored for changes.

The CAM maintains an array of events, each 78 bits in width, by writing DES coprocessor inputs in the first available cell and provides the next event when read by the DES coprocessor. Reads require both an identical match (i.e., designated TO_LP) and a "less than" comparison (i.e., earliest time) of valid entries to retrieve the next event for CPU execution. Additionally, the CAM provides a free-space status to the DES coprocessor after each write and a multiple message indication, if the next event was provided by an LP that has additional messages waiting for execution in the CAM.

```
--*****************************************************************************
-- FILE: CAM_BEH.vhd
-- AUTHOR:  Paul J. Taylor
-- PURPOSE: Architectural BEHAVIOR of the DES CAM
--*****************************************************************************
-- Overview of CAM operation:
--
-- Must store all events for all LPs (20 max per node)
-- Each LP is limited to 10 Input_Arcs max
-- Using "median" values: (10 LP/node)(5 input/LP) = 50 inputs/node
--
-- Assuming an average of 10 events/LP pending => CAM capacity >= 500 events
--
-- Constraints:
--      Event fields/CAM row (or row) = 80 bits = 8 bytes
--      | valid bit  | TO_LP | FROM_NODE | FROM_LP | TIME_TAG | MEMR_PTR |
--      |   1 bit    | 5 bits|  3 bits   | 5 bits  | 32 bits  |  32 bits |
--      |     77     |76   72|71       69|68     64|63      32|31       0
--
--      1024 bytes => 128 events        2048 bytes => 256 events
--      4096 bytes => 512 events        8192 bytes => 1024 events
--              512 addresses => 10 address bits (to store 512 events)
--
-- Considerations:
--
-- VALID BIT: set "high = '1'" as event is read/stored by CAM
--            Toggle "low = '0'" when event is written/sent by CAM
--            Logical "AND" of all valid bits will determine when CAM is full
--
-- MULTIPLE BIT: set "high = '1'" if, during search for Next_Event, multiple
--               matches of both TO_LP and FROM_LP fields occurs.
--               Bit 31 of first DWORD sent to DES coprocessor during Get_Event
--               Doesn't require extra field in CAM row; rather, it's a fill
--               bit for the first DWORD sent to DES coprocessor.
--
-- Determine CAM full status after storing new event
-- DES coprocesor will read CAM status after event store (to update DES flags)
-- Might consider adding "control" line to distinguish "status" from "event" read
--*****************************************************************************
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use ZYCAD.BV_ARITHMETIC.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;
-------------
-- CAM ENTITY
-------------
entity C_MEM is
        generic(RDEL, WDEL, DISDEL: TIME);
```

```
        port(DATAinto : in DWORD;
             DATAoutof : out DWORD:= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
             CLK : in BIT;
             DS1, NDS2, MODE, N_CLR : in MVL7);
end C_MEM;
--*****************************************************************************
-- CAM Pseudo/Algorithm
--
-- Write to CAM:  (DES Post_Msg)
--   Read DATA_BUS for (3) successive DWORDS
--   Traverse array of CAM rows till first "invalid" (i.e. bit 77 = '0')
--      set bit 77 = '1' (i.e. valid)
--      save this location (address) for successive writes/stores
--   Mask least significant 13 bits of 1st DWORD in, and store in 76 downto 64
--   Store 2nd DWORD in 63 downto 32
--   Store 3rd DWORD in 31 downto 0
--   Check CAM_FULL status
--      logical "AND" of all CAM row "valid" bits ('1' => FULL)
--        for loop (size of CAM) exit on any "not FULL"
--   Send CAM_STATUS to DES coprocessor : write immediately after read
--
-- Read from CAM: (DES Get_Event)
--   Read DATA_BUS and buffer "who" (TO_LP) requires Next_Event
--   Traverse array of CAM rows
--      search for match on "TO_LP" field ("valid" bit 77 = '1')
--   Traverse array of matches for earliest TIME_TAG (Next_Event): NOTE ADDRESS
--      NOTE: additional pass searching for FROM_LP matches with Next_Event
--      if multiple matches -> set MULTIPLE bit "high = '1'"
--      Save Next_Event Address (send this row) -> Toggle "valid" bit to '0'
--   Send Next_Event (3*DWORD) to DES coprocessor
--      construct 1st DWORD: bit31 = MULTIPLE bit; bits 12 downto 0 = TO/FROM_LP
--      2nd DWORD: TIME_TAG;    3rd DWORD: MEMR_PTR (Next_Event)
--
--*****************************************************************************
-------------------
-- CAM ARCHITECTURE
-------------------
architecture BEHAVIOR of C_MEM is

begin

CAM_PROC : process

        subtype ADDR is MVL7_VECTOR(9 downto 0);                -- 512 event addr
        subtype ROW is MVL7_VECTOR(77 downto 0);                -- event store
        type MEMTYPE is array (Natural range <>) of ROW;

        variable EVENT: MEMTYPE(0 to 2**ADDR'length-1):=
        MEMTYPE'(0 to 2**ADDR'length-1 =>
"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ");
        variable Next_Event    : ROW;                           -- send to DES
```

```
        variable FULL         : BOOLEAN := false;
        variable SEND_STAT    : BOOLEAN := false;
        variable EVENT_ID_REG : DWORD;                          -- hold TO_LP
        variable HAVE_ADDR    : BOOLEAN := false;               -- new event location
        variable ROW_LOC      : INTEGER := 0;                   -- event addr index
        variable EARLIEST_TIME: INTEGER:= 2147483647;           -- 7FFFFFFF (max time)
        variable EARLIEST_ADDR: INTEGER;
        variable EVENT_SEG    : INTEGER := 1;                   -- event segment index
        variable HAVE_EVENT   : BOOLEAN := false;               -- have next event
        variable MULTIPLE     : BOOLEAN := false;               -- have multiple?

begin

  wait on DS1, NDS2, MODE, N_CLR;
-------------------------------------------
-- Clear CAM by reseting bit 77 in all event
-------------------------------------------
    if N_CLR = '0' then                                         -- clear CAM
     for I in 0 to 2**ADDR'length-1 loop
       EVENT(I)(77) := '0';                                     -- reset "valid" bits
     end loop;
  end if;
-----------------------
-- Write EVENT into CAM
-----------------------
  if (NDS2 = '1' and MODE = '0') then                           -- WTC = '1' and NE = '0'

     if not HAVE_ADDR then
       FREE_SPACE_LOOP:                                         -- traverse all CAM rows
       for I in 1 to 2**ADDR'length-1 loop
         ROW_LOC := ROW_LOC + I;
         if EVENT(ROW_LOC)(77) = '0' then                       -- "FREE" event space
           EVENT(ROW_LOC)(77) := '1';                           -- use this address
           HAVE_ADDR := true;
           exit;
         end if;
       end loop FREE_SPACE_LOOP;
     end if;                                                    -- HAVE_ADDR for event
-------------------------------------------------------------------------------
-- Wait for event address then Read data_bus for 3*DWORD event and store in CAM
-------------------------------------------------------------------------------
     if HAVE_ADDR then
       case EVENT_SEG is
-----------------------------------------------------------
-- Store first event field (event_id) and toggle "HAVE_ADDR"
-----------------------------------------------------------
         when 1 => EVENT(ROW_LOC)(76 downto 64) := CHANGE(DATAinto(12 downto 0));
            EVENT_SEG := EVENT_SEG + 1;
-------------------------------------
-- Store second event field (time_tag)
-------------------------------------
```

```
          when 2 => EVENT(ROW_LOC)(63 downto 32) := CHANGE(DATAinto);
             EVENT_SEG := EVENT_SEG + 1;
 ------------------------------------
 -- Store third  event field (memr_ptr)
 ------------------------------------
          when 3 => EVENT(ROW_LOC)(31 downto 0) := CHANGE(DATAinto);
             EVENT_SEG := 1;
             SEND_STAT := true;                                  -- report status to coprocessor
             HAVE_ADDR := false;
 ------------------------------------------
 -- EVENT is stored, must check CAM_FULL status
 ------------------------------------------
            CHECK_CAM_FULL_LOOP:                                 -- done in parallel
            for I in ROW_LOC to 2**ADDR'length-1 loop            -- full to ROW_LOC
              if EVENT(I)(77) = '0' then
                FULL := false;                                   -- row not in use
                exit;
              elsif EVENT(I)(77) = '1' then                      -- row in use
                FULL := true;                                    -- keep checking
              end if;
            end loop CHECK_CAM_FULL_LOOP;

          when others => assert FALSE report "Invalid CAM Write"
                         severity FAILURE;
        end case;
      end if;                      -- if HAVE_ADDR
    end if;                 -- if (NDS2 = '1' and MODE = '0')
 ------------------------------------------
 -- Report CAM_FULL status out to DES coprocessor
 ------------------------------------------
   if SEND_STAT then
      wait until DS1 = '1';                                      -- RDC = '1'
      if FULL then                                               -- send FULL stat msg
        DATAoutof <= "00000000000000000000000000000001" after FFDEL,
                     "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after RDEL;
      else                                                       -- send NOT FULL stat msg
        DATAoutof <= "00000000000000000000000000000000" after FFDEL,
                     "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after RDEL;
      end if;
      SEND_STAT := false;
    end if;
 ----------------------
 -- READ EVENT from CAM
 ----------------------
 --  wait on MODE until MODE = '1';                              -- read event coming
   if (NDS2 = '1' and MODE = '1') then                           -- WTC= '1' and NE= '1'
      EVENT_ID_REG := DATAinto;                                  -- TO_LP for Next_Event
 ------------------------------------------------------------------------------
 -- traverse array of CAM rows (actually done in parallel!)
 -- Match Conditions:
 --      must be valid event - bit (77) = '1'
```

```
--      must be for TO_LP - bits (76 downto 72) match TO_LP's (count matches)
--      must be earliest time - bits (63 downto 32) are smallest
-------------------------------------------------------------------------
    NEXT_EVENT_LOOP:
    for I in 0 to 2**ADDR'length-1 loop
      if (EVENT(I)(77) = '1' and (EVENT(I)(76 downto 72) =
                                  CHANGE(EVENT_ID_REG(4 downto 0)))) then
        if EARLIEST_TIME > BVtoI(MVL7VtoBV(EVENT(I)(63 downto 32))) then
          EARLIEST_TIME := BVtoI(MVL7VtoBV(EVENT(I)(63 downto 32)));
          EARLIEST_ADDR := I;
        end if;
        Next_Event := EVENT(EARLIEST_ADDR);
      end if;
      if I = 2**ADDR'length-1 then
        HAVE_EVENT := true;
        EVENT(EARLIEST_ADDR)(77) := '0';                -- "used"
        Next_Event(77) := '0';                          -- assume "NO" multiples
      end if;
    end loop NEXT_EVENT_LOOP;
    wait for FFDEL;                                     -- toggle bit(77) above
-------------------------------------------------------------------------
-- Search matches from above looking for multiple occurences of same FROM_LP
-- for updating ARCS_IN_STATUS
-------------------------------------------------------------------------
    if (MODE = '1' and HAVE_EVENT) then
      MULTIPLE_EVENT_LOOP:
      for I in 0 to 2**ADDR'length-1 loop
        if (EVENT(I)(77) = '1' and (EVENT(I)(76 downto 64) =
                                    Next_Event(76 downto 64))) then
                                                        -- Multiple FROM_LP events
          DATAoutof(31 downto 13) <= "1000000000000000000";
          exit;                                         -- toggle multiple event
        else
          DATAoutof(31 downto 13) <= "0000000000000000000";
        end if;                                         -- no multiple events
      end loop MULTIPLE_EVENT_LOOP;
      wait for FFDEL;                                   -- toggle MSB bit above
    end if;             -- if (MODE = '1' and HAVE_EVENT)
  end if;               -- if (WTC= '1' and NE= '1')
-------------------------------------------------------
-- Send Next_Event to DES (to include MULTIPLE status)
-------------------------------------------------------
  if (DS1 = '1' and HAVE_EVENT) then                    -- RDC = '1'
------------------------------
-- MUST LOOP FOR 3*DWORD OUTPUT
------------------------------
     case EVENT_SEG is
---------------------------------------
-- send 1st part of Next_Event (event_id)
---------------------------------------
       when 1 => DATAoutof(12 downto 0) <= CHANGE(Next_Event(76 downto 64));
```

```
        wait for FFDEL;
-----------------------------------------
-- send 2nd part of Next_Event (time_tag)
-----------------------------------------
       when 2 => DATAoutof <= CHANGE(Next_Event(63 downto 32));
         wait for FFDEL;
-------------------------------------------------
-- send 3rd part of Next_Event (memory pointer)
-------------------------------------------------
       when 3 => DATAoutof <= CHANGE(Next_Event(31 downto 0));
                 wait for FFDEL;
                 EVENT_SEG := 0;                          -- reset for next
                 HAVE_EVENT := false;                     --   "    "    "

       when others => assert FALSE report "Invalid CAM Read"
                      severity FAILURE;
     end case;
     EVENT_SEG := EVENT_SEG + 1;
     DATAoutof <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after RDEL;
  end if;

end process CAM_PROC;

end BEHAVIOR;
```

Appendix C. *DES Coprocessor System Test*

This appendix contains the DES coprocessor system configuration, the system testbench used to verify the DES coprocessor operation, and the CPU driver for testbench stimulation.

C.1 DES System Configuration

The DES coprocessor system configuration is given in this appendix. The configuration encapsulates the entity behviors required by the testbench. The source code listings for the entity behaviors is included in Appendix B.

```
library ZYCAD;
use ZYCAD.TYPES.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;

configuration Des_system of Des_sys_test_bench is
  for test
    for CLOCK_CKT: Sys_clk
      use entity WORK.CLOCK_CKT(BEHAVIOR);
    end for;
    for CPU: CPU_driver
      use entity WORK.CPU_driver(BEHAVIOR);
    end for;
    for COPROC: DES_sys
      use entity WORK.DES_sys(CHIP_LEVEL);
    end for;
  end for;
end Des_system;
```

C.2 DES Sytem Test Bench

The testbench entity for the DES coprocssor system is contained in this appendix. The three components making up the testbench (i.e., DES system, CPU driver, and Clock) are declared in the architectural body. The signal mapping between components is included to provide the testbench interconnections shown in Figure 5.1. A stopping process is also included to prevent simulation runaway. Simulation runtimes can be varied by adjusting the stop_sim time accordingly.

```
------------------------------------------------------------
-- FILE: Des_sys_test_bench.vhd
-- AUTHOR: Paul J. Taylor
-- PURPOSE: The test_bench for the DES coprocessor system
------------------------------------------------------------
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;
-------------------------
-- THE ENTITY DECLARATION:
-------------------------
entity des_sys_test_bench is
end    des_sys_test_bench;
-------------------------
-- THE ARCHITECTURAL BODY:
-------------------------
architecture test of des_sys_test_bench is
-----------------------------
component DES_sys
   port(RUN      :  in MVL7;
        CLK      :  in BIT;
        RESETIN  :  in MVL7;
        WR       :  in MVL7;
        NPS1     :  in MVL7;
        NPS2     :  in MVL7;
        CMD0     :  in MVL7;
        INTR     :  out MVL7;
        READY0   :  inout MVL7;
        BUSY     :  out MVL7;
        ERROR    :  out MVL7;
        ADD_STR  :  in MVL7;
        SYSIN    :  in DWORD;
        SYSOUT   :  out DWORD);
end component;
-----------------------------
component CPU_driver
   port(RUN       : in MVL7;
        CLOCK     : in BIT;
        RESETout  : out MVL7;
        WRout     : out MVL7;
        M_IOout   : out MVL7;
        A15out    : out MVL7;
        A2out     : out MVL7;
        INTin     : in MVL7;
        RDYin     : in MVL7;
        BUSYin    : in MVL7;
        ERRin     : in MVL7;
        ASTRout   : out MVL7;
```

```
        DATAout  : out DWORD;
        DATAin   : in DWORD);
end component;
---------------------------------------
component Sys_clk
   generic(PER: TIME := 125 ns);
   port(CLK2: inout BIT; RUN: in MVL7);
end component;
---------------------------------------
signal  STOP_SIM          : BOOLEAN := false;
signal  RUN               :  MVL7;
signal  CLK, CLOCK, CLK2  :  BIT;
signal  RESETIN, RESETout :  MVL7;
signal  WR, WRout         :  MVL7;
signal  NPS1, M_IOout     :  MVL7;
signal  NPS2, A15out      :  MVL7;
signal  CMD0, A2out       :  MVL7;
signal  INTR, INTin       :  MVL7;
signal  READY0, RDYin     :  MVL7;
signal  BUSY, BUSYin      :  MVL7;
signal  ERROR, ERRin      :  MVL7;
signal  ADD_STR, ASTRout  :  MVL7;
signal  SYSIN, DATAout    :  DWORD;
signal  SYSOUT, DATAin    :  DWORD;
-----
begin
-----
CLOCK_CKT: Sys_clk port map (CLK2, RUN);

CPU: CPU_driver port map (RUN, CLOCK, RESETout, WRout, M_IOout, A15out, A2out,
                 INTin, RDYin, BUSYin, ERRin, ASTRout, DATAout, DATAin);

COPROC: DES_sys port map (RUN, CLK, RESETIN, WR, NPS1, NPS2, CMD0, INTR,
                          READY0, BUSY, ERROR, ADD_STR, SYSIN, SYSOUT);
-----------------------------------------------
-- CPU_driver to DES_Sys signal correspondence
-----------------------------------------------
CLK     <= CLK2;
CLOCK   <= CLK2;
RESETIN <= RESETout;
WR      <= WRout;
NPS1    <= M_IOout;
NPS2    <= A15out;
CMD0    <= A2out;
INTin   <= INTR;
RDYin   <= READY0;
BUSYin  <= BUSY;
ERRin   <= ERROR;
ADD_STR <= ASTRout;
SYSIN   <= DATAout;
DATAin  <= SYSOUT;
```

```
--
RUN_TEST: process
 begin
   RUN <= '1';
   STOP_SIM <= true after 30_000 ns;
   wait for 100_000 ns;
end process RUN_TEST;
-------------------------------------------------------------------
-- STOP_CONTROL process
--
-- Purpose: Terminates the simulation
-------------------------------------------------------------------
STOP_CONTROL: process
 begin
   wait until STOP_SIM = true;
   assert false report "Simulation Done" severity failure;
end process STOP_CONTROL;

end Test;
```

C.3 CPU Driver Behavior

The testbench stimulus is provided by the CPU driver contained in this appendix. The following source code activates a DES coprocessor test with one logical process per computing node. Multiple LP configurations are implemented with an extension to this driver.

The CPU control signals and system bus are activated by the test process from the CPU driver entity. Procedures are included for frequently used operations such as loading opcode instructions and operands for the DES coprocessor system. An operate procedure is also included to simulate CPU's operation when not directly driving the DES coprocessor.

The CPU driver process is sensitive to the DES coprocessor's READY0 status line and bus cycles are initiated on both negative and positive system clock transitions.

```
--************************************************************
-- FILE: CPU_driver.vhd
-- AUTHOR: Paul J. Taylor
-- PURPOSE: CPU driver for DES coprocessor test_bench
--************************************************************
library ZYCAD;
library DESIGN;
use ZYCAD.TYPES.all;
use WORK.all;
use WORK.SYSTEM.all;
use WORK.BUS_SYS.all;
-------------------------
-- THE ENTITY DECLARATION:
-------------------------
entity CPU_driver is
   port(RUN      : in MVL7;
        CLOCK    : in BIT;
        RESETout : out MVL7;
        WRout    : out MVL7;
        M_IOout  : out MVL7;
        A15out   : out MVL7;
        A2out    : out MVL7;
        INTin    : in MVL7;
        RDYin    : in MVL7;
        BUSYin   : in MVL7;
        ERRin    : in MVL7;
        ASTRout  : out MVL7;
        DATAout  : out DWORD;
        DATAin   : in DWORD);
end CPU_driver;
-------------------------
-- THE ARCHITECTURAL BODY:
-------------------------
architecture BEHAVIOR of CPU_driver is
--
begin
--
--************************************************************
-- ONE_LP process
--
-- PURPOSE:  Exercise the DES coprocessor.  Run the fundamental
--           procedures (Initialize, Post_Message, Get_Event,
--           and Post_Event) on Discrete Event Simulation with
--           one (1) LP on CPU node (i.e. carwash config #1).
--************************************************************
ONE_LP: process

        variable SET_UP         : BOOLEAN := false;
        variable LOADED         : BOOLEAN := false;
        variable DO_INIT_SIM    : BOOLEAN := false;
        variable LOAD_POST_MSG  : BOOLEAN := false;
```

```
        variable DO_POST_MSG    : BOOLEAN := false;
        variable LOAD_GET_EVENT : BOOLEAN := false;
        variable DO_GET_EVENT   : BOOLEAN := false;
        variable LOAD_POST_EVENT: BOOLEAN := false;
        variable DO_POST_EVENT  : BOOLEAN := false;
--************************************************************
-- LOAD_INSTR procedure
--
-- PURPOSE:  Send OPCODE to DES Coprocessor
--************************************************************
 procedure LOAD_INSTR (INPUT : DWORD) is

        variable OPCODE : DWORD;

 begin

        OPCODE := INPUT;
        M_IOout <= 'Z', '0' after 5 ns;
        WRout   <= 'Z', '1' after 5 ns;
        A15out  <= 'Z', '1' after 5 ns;
        A2out   <= 'Z', '0' after 5 ns;
        DATAout <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ",
                   OPCODE after 5 ns;
        ASTRout <= '0', '1' after PER/2;

 end LOAD_INSTR;
--************************************************************
-- LOAD_DATA procedure
--
-- PURPOSE:  Send OPERAND to DES Coprocessor
--************************************************************
 procedure LOAD_DATA (INPUT : DWORD) is

        variable OPERAND : DWORD;

 begin

        OPERAND := INPUT;
        M_IOout <= 'Z', '0' after 5 ns;
        WRout   <= 'Z', '1' after 5 ns;
        A15out  <= 'Z', '1' after 5 ns;
        A2out   <= 'Z', '1' after 5 ns;
        DATAout <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ",
                   OPERAND after 5 ns;
        ASTRout <= '0', '1' after PER/2;
        wait for PER/2;

 end LOAD_DATA;
--************************************************************
-- INT_SERV procedure
--
```

```
-- PURPOSE:  Service interrupt requests from Coprocessor.
--           Acknowledges by changing status of control lines
--           to address the coprocessor (A15) in IO space (M_IO)
--           and setting mode to read (WRout).
--           After bus cycle (RDYin = '0'), disconnect COP by
--           resetting control lines.
--*************************************************************
 procedure INT_SERV is

 begin
---------------------
-- COP interrupts CPU
---------------------
     assert (RDYin = '1') report "not RDYin during INT Request"
     severity WARNING;

     if (RDYin = '1') then                        --  and (RDYin = '1')) then
       M_IOout <= '0' after ODEL;                 -- I/O address space
       A15out  <= '1' after ODEL;                 -- cop address
       WRout   <= '0' after ODEL;                 -- cpu read cycle
       wait for ODEL;
       wait until (RDYin = '0');                  -- ends cop write cycle
     end if;

 end INT_SERV;
--*************************************************************
-- OPERATE procedure
--
-- PURPOSE:  Perform CPU operations.  Wait until interrupt
--           received from DES Coprocessor, then service it.
--*************************************************************
 procedure OPERATE is

 begin

   WORK_Loop:                                              -- CPU "on" with RUN
   while ((RUN = '1') and (BUSYin = '1')) loop
     DATAout <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";        -- Release bus
     M_IOout <= 'Z' after ODEL;                            -- Release Coprocessor
     WRout <= 'Z' after ODEL;                              --      "       "
     A15out <= '0' after ODEL;                             --      "       "
     A2out <= '0' after ODEL;
     wait for ODEL;
     if (INTin = '1') then                                 -- service int rqst
       INT_SERV;
     end if;
   end loop WORK_Loop;

 end OPERATE;
--*************************************************************************
-- BEGIN ONE_LP PROCESS
```

```
--*****************************************************************************
 begin

--*****************************************************************************
-- Load OPCODE and OPERANDS for INIT_SIM procedure
--*****************************************************************************
 if not LOADED then                                          -- send instr & data

   if not SET_UP then                                        -- init coprocessor
     RESETout <= '1', '0' after PER/2;
     SET_UP := false;
   end if;

   LOAD_INSTR("00000000000000000000000000000000");           -- Init_Sim opcode
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000000");            -- TO_LP 0|0
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000100");            -- LP_DELAY 4 units
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000010");            -- # ARCS_IN
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000011");            -- # ARCS_OUT
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000000");            -- IN_NODE 0| IN_LP 0
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000011100000000000000000");            -- IN_NODE 7| IN_LP 0
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000000000000000000000000");            -- OUT_NODE 0| OUT_LP 0
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000001100000000000000000");            -- OUT_NODE 3| OUT_LP 0
   wait until ((RDYin = '0') and (not CLOCK'stable));        -- DES ends bus cycle

   LOAD_DATA("00000000000010000000000000000000");            -- OUT_NODE 4| OUT_LP 0
   wait until (RDYin = '0');                                 -- DES ends bus cycle
   LOADED := true;
   DO_INIT_SIM := true;
--*****************************************************************************
-- Test Init_Sim function
--*****************************************************************************
   if DO_INIT_SIM then

        OPERATE;
        if (BUSYin = '0') then
```

```
            DO_INIT_SIM := false;
            LOAD_POST_MSG := true;
            wait until (not CLOCK'stable);              -- next test synch
         end if;
   end if;
--*****************************************************************************
-- Load opcode and operands to run Post_Msg
--*****************************************************************************
   if (LOAD_POST_MSG and (BUSYin = '0')) then           -- load POST_MSG test

     wait until (not CLOCK'stable);                     -- start on clock
     LOAD_INSTR("00100000000000000000000000000000");    -- Post_Msg opcode
     wait until ((RDYin = '0') and (not CLOCK'stable)); -- DES ends bus cycle

     LOAD_DATA("00000000000000000000000000000000");     -- TO_LP (NODE 0| LP 0)
     wait until (RDYin = '0' and (not CLOCK'stable));   -- DES ends bus cycle

     LOAD_DATA("00000000000011100000000000000000");     -- FROM_LP (NODE 7| LP 0)
     wait until (RDYin = '0' and (not CLOCK'stable));   -- DES ends bus cycle

     LOAD_DATA("00000000000000000000000000001111");     -- TIME_TAG (15 units)
     wait until (RDYin = '0' and (not CLOCK'stable));   -- DES ends bus cycle

     LOAD_DATA('01010101010101010101010101010101");     -- MEMR_PTR (CPU mem_addr)
     wait until (RDYin = '0');                          -- DES ends bus cycle
     DO_POST_MSG := true;

   end if;
--*****************************************************************************
-- Test Post_Msg
--*****************************************************************************
   if DO_POST_MSG then

        DO_POST_MSG := false;
        OPERATE;
        if (BUSYin = '0') then
           LOAD_GET_EVENT := true;
           wait until (not CLOCK'stable);               -- synch for next test
        end if;
   end if;
-- *****************************************************************************
-- Load opcode and operands to run GET_EVENT
-- *****************************************************************************
-- All input arcs must have message in CAM
-- ARCS_IN_STAT must be verified as good
        -- Retrieve and send CPU a "real" event
        -- Request event but return "wait" message
-- *****************************************************************************
   if (LOAD_GET_EVENT and (BUSYin = '0')) then          -- load GET_EVENT test
-----------------------------------------------------------------
-- Satisfy message on all input arcs requirement for lp: 0 node: 0
```

```
-----------------------------------------------------------------
     LOAD_GET_EVENT := false;
     wait until not CLOCK'stable;
     LOAD_INSTR("00100000000000000000000000000000");     -- Post_Msg opcode
     wait until ((RDYin = '0') and (not CLOCK'stable)); -- DES ends bus cycle

     LOAD_DATA("00000000000000000000000000000000");      -- TO_LP (NODE 0| LP 0)
     wait until (RDYin = '0' and (not CLOCK'stable));    -- DES ends bus cycle

     LOAD_DATA("00000000000000000000000000000000");      -- FROM_LP (NODE 0| LP 0)
     wait until (RDYin = '0' and (not CLOCK'stable));    -- DES ends bus cycle

     LOAD_DATA("00000000000000000000000000001010");      -- TIME_TAG (10 units)
     wait until (RDYin = '0' and (not CLOCK'stable));    -- DES ends bus cycle

     LOAD_DATA("01001111010011110100111101001111");      -- MEMR_PTR (CPU mem_addr)
     wait until (RDYin = '0');                           -- DES ends bus cycle
     DO_POST_MSG := true;

     if DO_POST_MSG then
        DO_POST_MSG := false;
        OPERATE;
        if (BUSYin = '0') then
           LOAD_GET_EVENT := true;
           wait until (not CLOCK'stable);
        end if;
     end if;
--*****************************************************************************
-- Test GET_EVENT
--*****************************************************************************
     if (LOAD_GET_EVENT and (BUSYin = '0')) then
       LOAD_GET_EVENT := false;
       wait until not CLOCK'stable;

       LOAD_INSTR("01000000000000000000000000000000");  -- Get_Event opcode
       wait until ((RDYin = '0') and (not CLOCK'stable));

       LOAD_DATA("00000000000000000000000000000000");   -- TO_LP (NODE 0| LP 0)
       wait until (RDYin = '0');
-----------------------------------------------------------------
-- CPU releases bus and receives NEXT_EVENT from DES Coprocessor
-----------------------------------------------------------------
       DATAout <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
       EVENT_Loop:
       while ((RUN = '1') and (BUSYin = '1')) loop
         M_IOout <= '0' after ODEL;                    -- Read event
         WRout <= '0' after ODEL;
         A15out <= '1' after ODEL;
         wait for ODEL;
       end loop EVENT_Loop;
       LOAD_POST_EVENT := true;
```

```
      end if;
    end if;              -- if LOAD_GET_EVENT
--*****************************************************************************
-- Load opcode and operands to run Post_Event
--*****************************************************************************
    if (LOAD_POST_EVENT and BUSYin = '0') then          -- load Post_Event
      LOAD_POST_EVENT := false;
      wait until not CLOCK'stable;

      LOAD_INSTR("01100000000000000000000000000000");    -- Post_Event opcode
      wait until ((RDYin = '0') and (not CLOCK'stable));

      LOAD_DATA("00000000000000000000000000000000");     -- node 0 | LP 0
      wait until ((RDYin = '0') and (not CLOCK'stable)); -- "FROM_LP"

      LOAD_DATA("01110111011101110111011101110111");     -- Memory Pointer
      wait until ((RDYin = '0') and (not CLOCK'stable)); -- to EVENT

      LOAD_DATA("00000000000010000000000000000000");     -- node 4 | LP 0
      wait until (RDYin = '0');                          -- "TO_LP"

      DO_POST_EVENT := true;
--*****************************************************************************
-- Test Post_Event
--*****************************************************************************
    if DO_POST_EVENT then

        OPERATE;
        if (BUSYin = '0') then
           DO_POST_EVENT := false;
           wait until (not CLOCK'stable);              -- next test synch
        end if;
    end if;
--*****************************************************************************
-- DO_NOTHING (included for extra time in simulation)
--*****************************************************************************
     BUSY_LOOP2: loop
       wait until (not CLOCK'stable);                  -- next test synch
       exit BUSY_LOOP2 when (RUN = '0');
     end loop BUSY_LOOP2;

   end if;              -- if LOAD_POST_EVENT
 end if;        -- CHECK FOR STATE TRANSITIONS!

end process ONE_LP;

end BEHAVIOR;
```

Bibliography

1. Abramovici, Miron and others. "A Logic Simulation Machine," *IEEE Transactions on CAD of Integrated Circuits and Systems*, *2*:82–93 (April 1983).
2. Armstrong, James R. *Chip-Level Modeling with VHDL*. Prentice Hall, 1989.
3. Blank, Tom. "A Survey of Hardware Accelerators Used in Computer-Aided Design," *IEEE Design and Test*, *1*:21–39 (August 1984).
4. Catlin, Gary and Bill Paseman. "Hardware Acceleration of Logic Simulation Using a Data Flow Architecture." In *International Conference on Computer-Aided Design*, pages 130–132, Washington D.C.: IEEE, 1985.
5. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*:198–206 (April 1981).
6. Chisvin, Lawrence and R. James Duckworth. "Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM," *Computer*, pages 51–64 (July 1989).
7. Comfort, John C. "The Design of a Multi-Microprocessor Based Simulation Computer - II." In *16th Annual Simulation Symposium*, pages 45–53, New York: IEEE, 1983.
8. Comfort, John C. "The Simulation of a Master-Slave Event Set Processor," *Simulation*, *42*:117–124 (March 1984).
9. d'Abreu, Manuel A. "Gate-Level Simulation," *IEEE Design and Test*, *2*:63–71 (December 1985).
10. DeCegama, Angel L. *The Technology of Parallel Processing, Volume 1*. Prentice Hall, 1989.
11. El-Ayat, Khaled A. and Rakesh K. Agarwal. "The Intel 80386 - Architecture and Implementation," *IEEE MICRO*, pages 2–22 (December 1985).
12. Franklin, M. A. and others. "Parallel Machines and Algorithms for Discrete-Event Simulation." In *International Conference on Parallel Processing*, pages 449–458. Columbus, Oh.: IEEE, 1984.
13. Fujimoto, Richard M. and others. "The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp," *Distributed Simulation*, *19*:81–86 (February 1988).
14. Goering, Richard. "Simulation Accelerators Address Throughput Issues," *Computer Design*, pages 42–50 (March 1988).
15. Hanlon, A. G. "Content-Addressable and Associative Memory Systems A SURVEY," *IEEE Transactions on Electronic Computers*, *EC-15*:509–521 (August 1966).
16. Hayes, John P. *Computer Architecture and Organization*. McGraw-Hill Book Company, 1988.
17. Hayes, John P. and Trevor Mudge. "Hypercube Supercomputers." In *Proceedings of the IEEE*, pages 1829–1841, IEEE, 1989.

18. Intel Corporation. *iPSC/2 and iPSC/860 Programmer's Reference Manual*, 1990.

19. Intel Corporation. *Microprocessors, Volume II*, 1991.

20. Jefferson, David. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7:404–425 (July 1985).

21. Lee, Ann Kathryn. *An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation.* MS thesis, Air Force Institue of Technology, 1990. Technical Report AFIT/GCS/ENG/90D-08.

22. Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computering Surveys*, *18*:39–65 (March 1986).

23. Nasar, Steve, "telephone interview," August 1991. Intel technical representative, Beaverton, Or.

24. Neelamkavil, Francis. *Computer Simulation and Modelling.* John Wiley and Sons, 1987.

25. Parhami, Behrooz. "Associative Memories and Processors: An Overview and Selected Bibliography." In *Proceedings of the IEEE, vol. 61*, pages 722–730, IEEE, 1973.

26. Pritsker, A. Alan B. and Claude D. Pegden. *Introduction to Simulation and SLAM.* John Wiley and Sons, 1984.

27. Redman, David, "telephone interview," August 1991. Intel technical representative, Indianapolis, In.

28. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." In *Distributed Simulation*, pages 8–13, La Jolla CA: SCS, 1988.

29. Reynolds, Jr. Paul F. and Phillip M. Dickens. "SPECTRUM: A Parallel Simulation Testbed." In *4th Hypercube Conference*, pages 865–870, IEEE, 1990.

30. Smith II, Richard J. "Fundamentals of Parallel Logic Simulation." In *Proceedings of the IEEE 23rd Design Automation Conference*, pages 2–12, New York: IEEE, 1986.

31. Wong, Ken and Mark A. Franklin. "Performance Analysis and Design of a Logic Simulation Machine." In *14th International Symposium on Computer Architecture*, pages 46–55, Pittsburgh, Pa.: IEEE, 1987.

32. ZYCAD Corporation. *Zycad System VHDL Reference Manual*, 1990.

Vita

Captain Paul J. Taylor, Jr. was born on 19 August 1957 at Mitchell Field, Long Island New York. He graduated from Northern Burlington High School in 1975. He enlisted in the Air Force in 1977 and served five years as an explosive ordnance disposal technician before being accepted for the Airman's Educational Commissioning Program. He received his undergraduate engineering degree from the University of Central Florida and received his Air Force commission in 1985. Prior to entering AFIT he was chief of the Range Communications Branch, 554th Range Group, Nellis AFB Nevada.

Permanent address: 203 Arrahbella
Browns Mills, New Jersey
08015

AFIT/GCE/ENG/91D-11

Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator

THESIS

Paul J. Taylor, Jr.
Captain, USAF

AFIT/GCE/ENG/91D-11

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
REQUIREMENTS ANALYSIS FOR A HARDWARE DISCRETE EVENT SIMULATION ENGINE ACCELERATOR

5. FUNDING NUMBERS

6. AUTHOR(S)
Paul J. Taylor, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/91D-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA (LTC John Toole)
3701 N. Fairfax Dr.
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

An analysis of a general Discrete Event Simulation (DES), executing on the distributed architecture of an eight node Intel iPSC/2 hypercube, was performed. The most time consuming portions of the general DES algorithm were determined to be the functions associated with message passing of required simulation data between processing nodes of the hypercube architecture. A behavioral description, using the IEEE standard VHSIC Hardware Description and Design Language (VHDL), for a general DES hardware accelerator is presented. The behavioral description specifies the operational requirements for a DES coprocessor to augment the hypercube's execution of DES simulations. The DES coprocessor design implements the functions necessary to perform distributed discrete event simulations using a conservative time synchronization protocol.

14. SUBJECT TERMS
Simulation, Parallel Processing, Discrete Event Simulation, VHDL, Coprocessor, Simulation Accelerator

15. NUMBER OF PAGES
182

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO 1990-0-273-271