Methodology for VHDL Performance Model Construction and Validation

by

Srilekha Vuppala

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master Of Science
in
Electrical Engineering

APPROVED:

Dr. F.G.Gray, Chairman

Dr. J.R.Armstrong

Dr. W.R.Cyre

May 1996

Blacksburg, Virginia

Keywords: VHDL, Performance Modeling, Model Construction, Model Validation.

Methodology for VHDL Performance Model Construction and Validation

by

Srilekha Vuppala

Dr. F.G.Gray, Chairman

Electrical Engineering

(ABSTRACT)

Hardware description languages(HDLs) are frequently used to construct performance models to represent systems early in the design process. This research study has resulted in the development of a methodology to construct VHDL performance models which will help to significantly reduce the time from an initial conception to a working design. To further reduce development time, reuse of existing structural primitives is emphasized.

Typical models of multi-processor architectures are very large and complex. Validation of theses models is difficult and time consuming. This thesis also develops a methodology for model validation.

A seventeen processor raceway architecture, that was developed as a part of the ongoing RASSP(Rapid Prototyping of Application Specific Signal Processors) project, is used as a template to illustrate the new methodologies of performance model construction and model validation. The design consists of seventeen processors interconnected by multiple crossbar switches. Two software algorithms were mapped onto the architecture: a Synthetic Aperture Radar(SAR) Range Processing Algorithm and a SAR Multiswath Processing Algorithm.

The methodologies developed in this thesis will considerably reduce the amount of time needed to construct and validate performance models of complex multi-processor architectures.

To my mom and dad

Acknowledgments

Numerous people are responsible for the success of this research work. I take this opportunity to thank them. My sincere appreciation goes to my academic advisor, Dr. F.G.Gray for his unerring guidance and support during the past year. I would like to thank my co-advisor, Dr. J.R.Armstrong, for his valuable ideas and suggestions. Appreciation is due to Dr. W.R.Cyre for serving on my graduate committee.

Special thanks to Dr. Geoff Frank and Mr. Bud Clark of Research Triangle Institute for their invaluable ideas and suggestions. I would like to extend my thanks to Todd Carpenter and John Shackleton of Honeywell, Inc. for answering all my questions on the model library and their support.

This work also owes much to the inspiration and encouragement I received from my friend, Nitin Bhat. My sincere appreciation to my brother-in-law for his support. Finally my sincere appreciation to my parents, sisters and brother for their love and support without which I could not have got this far in my career.

Table Of Contents

List of Illustrations

Chapter 1. INTRODUCTION

1.1 Motivation

The design and development of high performance computing systems is becoming increasingly complex[8]. Hardware Description Languages(HDLs) are frequently used to construct performance models to represent systems early in the design process. A primary ingredient of a detailed system performance model is a sound construction methodology.

Once the model has been constructed, validation and interpretation of the results can be a complex and time consuming task. Since there are no good tools to assist in validation, there is potential for much improvement in this area.

In this thesis, an efficient methodology is developed for the construction of VHDL performance models and their validation. These techniques will considerably reduce the overhead time required to construct and validate VHDL performance models.

1.2 Task Description

This research work is a part of the *Library Task* of the ongoing RASSP(Rapid Prototyping Of Application Specific Signal Processors) project at Virginia Tech. The aim is to develop a high level VHDL library of hardware models and software algorithms and to evaluate the performance of these models.

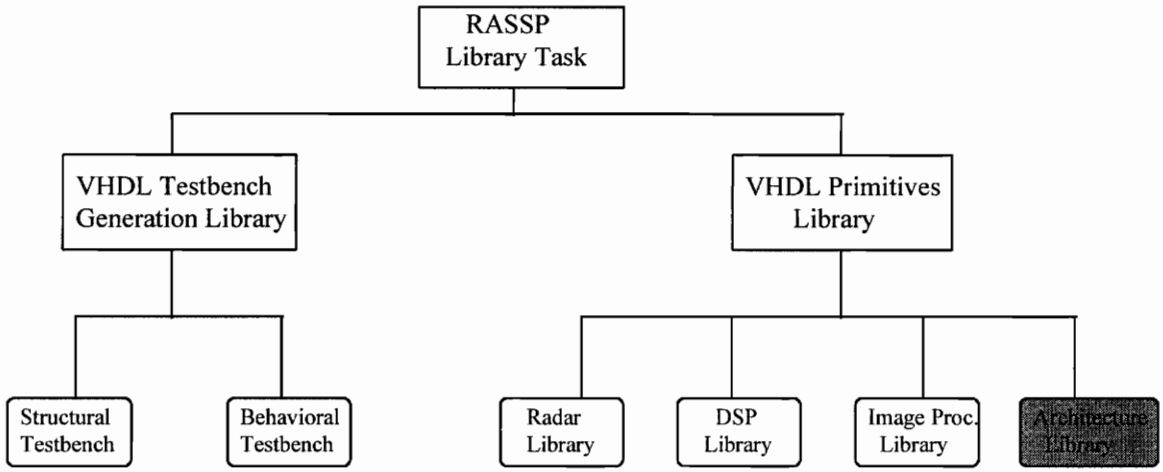


Figure 1.1 The VT RASSP Project

Figure 1.1 shows the organization of the Virginia Tech RASSP Library task project. The two main libraries developed are the *VHDL Testbench Generation Library* and the *VHDL Primitives Library.*

The *VHDL Primitives library* is a database to serve as a repository for objects developed in the RASSP project. The *Radar* library contains one Synthetic Aperture Radar(SAR) model that is currently undergoing testing. The library of *Digital Signal Processing* primitives currently contains fully tested models for fast fourier transforms, discrete fourier transforms, FIR and IIR filters developed by Ram Gummadi[12]. The *Image Processing* library contains an infra red sensing and tracking(IRST) model that

was initially developed by H.P.Commissariat[9] and significantly improved by Sailesh Kottapalli[11] and some automatic target recognition(ATR) models[3]. The *architecture* library was developed as part of the current thesis. Figure 1.2 shows the organization of the RASSP VHDL Architecture Library.

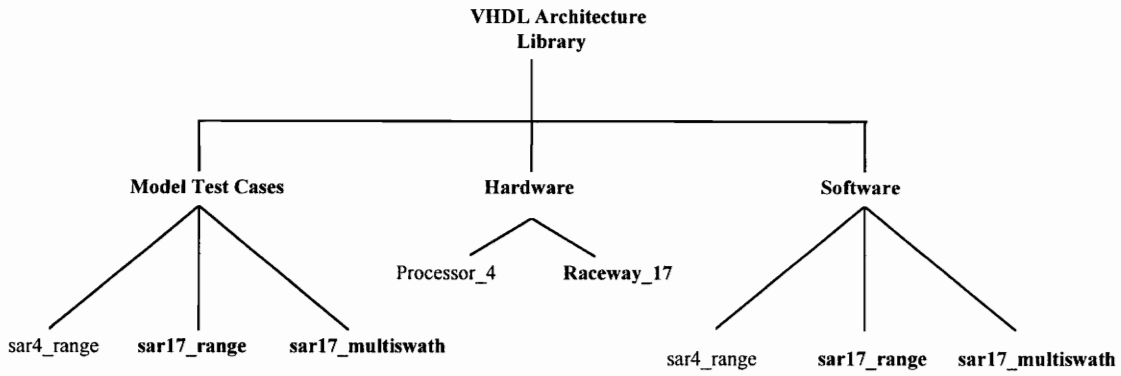


Figure 1.2 The RASSP VHDL Architecture Library

The RASSP VHDL Architecture Library consists of three main sub-directories. The *Hardware* library contains hardware structural models. *Raceway_17* is the seventeen processor raceway architecture hardware model developed in the present research study. *Processor_4* is a four processor hardware architecture developed by H.P. Commissariat[9]. The *Software* Library contains the application software algorithms that are mapped onto the hardware architecture models. The *sar17_range* and *sar17_multiswath* are respectively the Synthetic Aperture Radar(SAR) Range Processing and SAR Multiswath Processing algorithms which were developed as part of this thesis work. The *sar4_range*, a smaller SAR algorithm was developed by H.P.Commissariat[9] as part of an earlier research project. The *Model Test Cases* library contains the simulation results of the software algorithms running on the hardware models.

The *VHDL testbench generation library* contains VHDL testbench models. A

testbench is an executable VHDL model which instantiates a model under test(MUT) and provides capability to drive the MUT with a set of test vectors and compare its response with the expected response. The testbench is the topmost level in the hierarchy. The *behavioral testbench* library contains high level behavioral models of the testbench that were developed using CASE tools[13]. The *structural testbench* library contains structural models of the testbench. In this case, a conventional schematic capture tool selects test bench components from the *structural testbench* library[14]. The MUT is chosen from the *radar library* or the *image processing library* of the *VHDL Primitives library*. Ilogix Express VHDL and Cadence SPW tools were used to create the behavioral testbench models and the Synopsys Graphical Environment tool to generate structural testbench models[3],[11].

1.3 Research Contributions

A methodology for construction of VHDL performance models has been developed. This technique helps to considerably reduce the time required from an initial concept to a working VHDL design. The following features ensure reusability of the developed models.

- **Use Of Generics:** Important device and system parameters are made generics so that they can be easily modified whenever required.
- **Modular Design:** The complete design is composed of independent modules linked together. Hence, for other applications the individual blocks of the model can be reused to construct other performance models. For example, a portion of the SAR Multiswath Processing algorithm employs a similar type of processing as the SAR Range Processing algorithm. Hence that part of the application software could be reused without much modification. This also allows easy upgrade of the model due to model year changes.

- **Documentation and coding style:** Good documentation and coding style improves the readability of the VHDL code and facilitates reuse of the entire model as well as reuse of individual components.

A methodology for validation of VHDL performance models has also been developed. This methodology helps to verify that the simulation results of the model actually describe the real system. Two main techniques were adopted to validate the simulation results.

- **Decomposition:** The application software algorithm running on the hardware model was decomposed into small portions. These pieces of the algorithm were individually simulated and each of these results validated.
- **Scaling:** The algorithm was scaled down and verified for correctness of the results. For example, the SAR Range processing algorithm has one of the processors waiting on 512 pulses from the input, *radar*. This was scaled down to a value of just 4 pulses and simulated and the results were checked. Once the scaled results are verified to be sound, the actual whole algorithm should be run. We did not run the whole algorithm because the execution time of the algorithm was very large.

A seventeen processor raceway architecture is developed to illustrate the above methodologies of VHDL performance model construction and model validation. Figure 1.3 shows the schematic of the Raceway model.

The primitives for this model were obtained from the Performance Model Library(PML) developed by Honeywell Inc.[8],[5]. Some of these models were reused without modification while some were customized to meet the present research needs.

The seventeen processor raceway architecture is a structural inter-connection of different primitive components. The processors are organized into clusters. These clusters

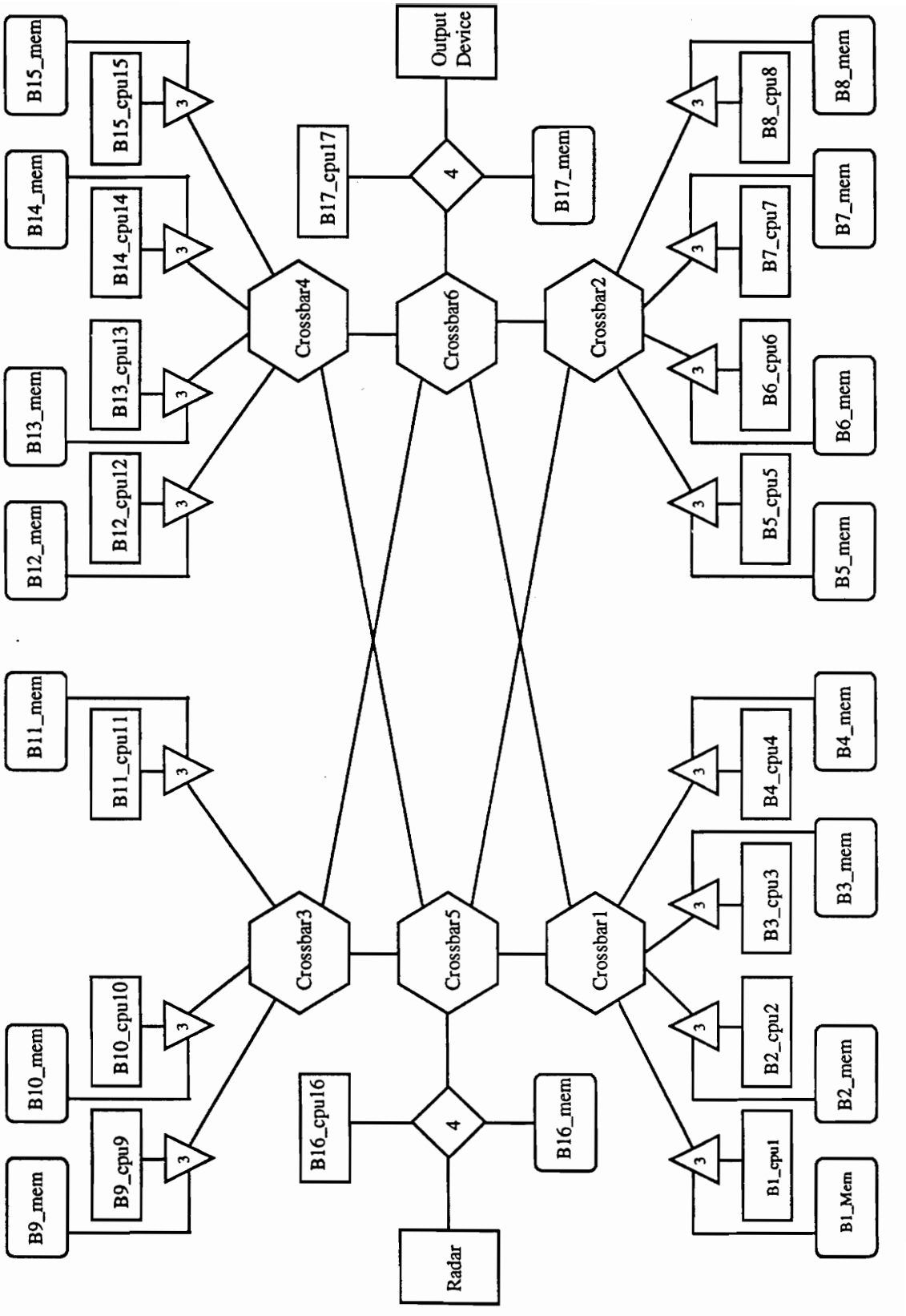


Figure 1.3 Schematic Of The Seventeen Processor Raceway Architecture

are interconnected by six crossbar switches which help reduce the traffic congestion on the global bus and also improve inter-device communication. The stimulus to the system is given by the input device which models a *radar*. The output device is the component where the results are stored.

There are seventeen clusters in total. Each cluster consists of a processor and its local memory connected to one of the ports of the crossbar component. The input cluster consists of the *radar*, the input processor, its local memory and one of the ports of the crossbar all connected to a common bus. The output cluster has the output device, output processor, its local memory and one of the ports of the crossbar all connected to a common bus.

Two software algorithms were mapped onto the above hardware model: the Synthetic Aperture Radar(SAR) Range Processing algorithm and the SAR Multiswath Processing Algorithm. These algorithms are explained in detail in chapter 4.

The *input processor* at the input cluster processes the data from the *radar*. The four processors connected to crossbar1 are the *range processors* which do the *range processing*. The B11_cpu11 processor connected to crossbar5 is the *corner turn processor* which does the *corner turn processing*. The B15_cpu15 processor connected to crossbar4 is the *image processor*. The *output processor* is connected to crossbar6 which stores the result in the *output device*. All the remaining nine processors are the *Azimuth processors*.

The above two algorithms were successfully simulated on the raceway architecture and the results of the simulation were validated. These results are detailed in Chapter 4.

1.4 Thesis Organization

Chapter 2, "Background", gives an overview of VHDL and its potential capabilities. Different types of VHDL models are briefly described. Also, the design hierarchy, performance models and a description of our performance model tradeoff system are detailed.

Chapter 3, "Description Of The Model Library", describes the different components in the Honeywell Performance Model Library(PML).

Chapter 4, "Methodology for Model Construction", defines the methodology developed to construct VHDL models. The seventeen processor architecture is explained. The results of the mapping of the two algorithms, the range processing and multiswath processing algorithms onto this architecture are detailed.

Chapter 5, "Methodology For Model Validation", describes a methodology to validate and debug VHDL models. Different examples are described.

Chapter 6, "Conclusions and Future Work", is the concluding chapter of the thesis. It highlights the contribution of this thesis towards advancement in VHDL performance model construction and validation techniques and makes suggestions for further improvement.

Chapter 2: BACKGROUND

2.1 On Performance Modeling

A design can be modeled at any level of hierarchy, from the system level to the gate level. Figure 2.1 shows a picture of the design hierarchy[1].

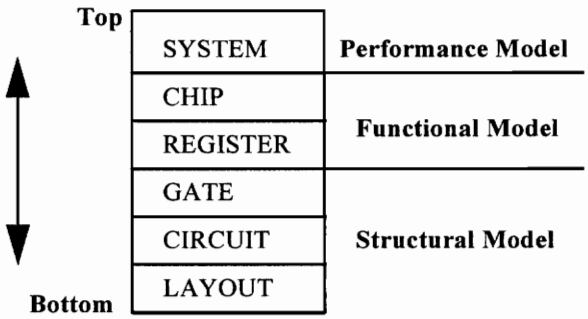


Figure 2.1 Design Hierarchy

The following is from [1]:

"At the lowest level, the layout/silicon level, the basic primitives are geometric shapes that represent diffusion areas, polysilicon gates etc. The interconnection of these patterns models the fabrication process. At the next level, the circuit level, the representation is that of an interconnection of electrical circuit elements such as resistors, capacitors and transistors. The next level, the gate level is the major level for digital

design. The basic primitives are the AND, OR , INVERT operators and flip-flops. The interconnection of these primitives forms combinational and sequential logic circuits while Boolean equations define the behavior at this level. These three levels are structural interconnections of the basic primitives called the structural primitives[1].

The next level is the register level. Examples of basic primitives are registers, counters and ALUs. These are called functional blocks. Hence this level is often referred to as the functional level. The next level is the chip level. The structural primitives at this stage are microprocessors, memories, I/O ports, etc. The key aspect is that a large block of logic is to be represented in which long and frequently convergent data paths must be modeled from inputs to the outputs".

The System level is the highest in the design hierarchy. **Performance models[8]** are used to model system level behavior. At this level, the system is in the block-diagram stage. Only the main function of the block, relative to the system as a whole, is specified. The intricate details of the chips or gates within each block are not known at this stage. The goal is to define a top level view of the whole system. In spite of the lack of details, evaluation of the system performance can be done at this stage. Performance models are used to accomplish the following tasks:

- Different performance statistics such as bandwidth of a bus, percentage utilization of different blocks in the system, the throughput of the blocks, etc. can be studied.
- Traffic congestion and communication bottlenecks of the system can be identified.
- Designer's can make a comparison of different architectures and can help decide the best architecture for a particular application.
- High level design trade-offs such as cost-effectiveness versus better performance can be made.

Structural primitives at this level include complex components such as hard disk, radar, switching components, computers, etc.

2.2 Role Of VHDL

VHDL is an acronym for VHSIC Hardware Description Language(VHSIC is an acronym for Very High Speed Integrated Circuit). It is a hardware description language that can be used to model a digital system at many levels of abstraction, ranging from system level to the gate level. The digital system may be described hierarchically and timing can be explicitly modeled in the same description.

Some of the features of VHDL that differentiate it from other hardware description languages are listed below[2].

- VHDL supports flexible design methodologies: top-down, bottom-up or both.
- The language supports hierarchy, i.e., a digital system can be modeled as a set of interconnected components, where each component may be further modeled as a set of interconnected subcomponents.
- Various digital modeling techniques, such as finite state machine descriptions, algorithmic descriptions and Boolean equations can be modeled using this language.
- It supports both asynchronous and synchronous timing models.
- The language has elements that make large scale modeling easier, for example, packages, procedures, functions.
- Test benches can be written in VHDL to test other VHDL models.
- Important device parameters can be easily modified by the use of *generics*.
- A model can not only describe the functionality of a design, but also contain information about the design itself in terms of user defined attributes, such as area and speed.
- Timing properties such as setup time, hold time and propagation delays can easily be described in this language.
- VHDL is an IEEE and ANSI standard that is widely used in industry and government[6].

2.3 Basic VHDL Models

The internal details of a design in VHDL can be specified by an architecture body using any of these styles[1]

- a structural model, in which a model is described as a set of interconnected components.
- dataflow model, in which a model is described as a set of concurrent assignment statements.
- behavioral model, in which a model is described by defining its input/output response

Figure 2.2 shows the different model types.

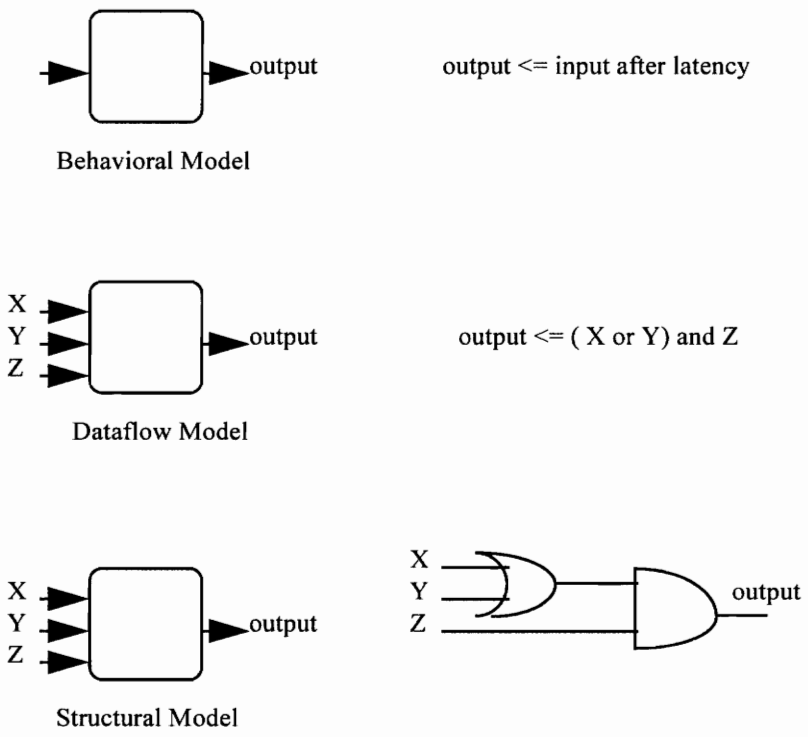


Figure 2.2 Representation Of Different Model Types

2.4 Overview of our System

The present work is a part of a joint effort by Virginia Tech and Research Triangle

Institute located in Research Triangle Park, NC. Figure 2.3 shows an overview of the system being developed[4]

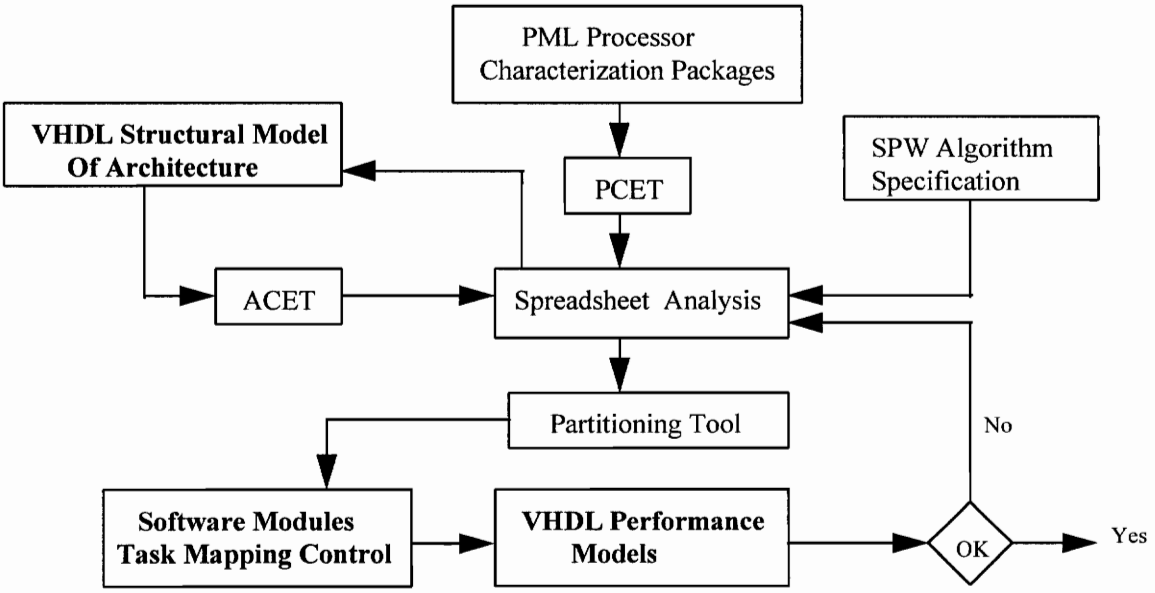


Figure 2.3 Architecture Tradeoff Environment

The main objective for the development of the above system is to make architectural trade-offs. The following are the steps adopted in the trade off study[4].

- Extract the processor characterization data from the PML processor library using the Processor Characterization Extraction Tool(PCET).
- Determine the required number of processors to run a given application from the Spreadsheet Analysis.
- Build a VHDL structural model of the architecture.
- Automatically create the Analysis Spreadsheet using the Architecture Characteristic Extraction Tool(ACET).
- Use the Spreadsheet to obtain the partitioning of the software application algorithm.

- Given the partitioned software application specifications, the software is converted to PML code(Performance Model Library) and mapped onto the specific processors in the hardware architecture model.
- Evaluate the performance of the model. If the results are not satisfactory, go back and change the partitioning of the software or the hardware model and repeat the remaining steps.

This thesis concentrates on creation of the hardware architecture model, mapping of the software tasks onto the different processors and evaluation of the performance models.

2.4.1 Role Of PML

The Performance Model Library(PML) is a library of primitive components that were used to develop the present performance models. The Model Library is fully described in chapter 3. These primitives were obtained from a previous research work[8]. The reuse of the existing models and the models developed is ensured in the following ways:

- **Use Of Generics:** Important device parameters like the throughput of the component, bus width, etc. are made generics so that their values can be easily changed.
- **Modular Design:** In this, the complete design is composed of independent but linked blocks. These blocks do not depend on global signals or variables. Hence, individual blocks can be salvaged for use in other applications.
- **Documentation and Coding Style:** Good documentation and coding styles can go a long way in the reuse of the model as well as individual blocks.
- **Ease Of Extendibility:** The models are designed in such a way that they could be customized to meet a variety of needs. For example, the basic token structure(which is a fundamental signal) has a few redundant fields. This helps to prevent modifying all the components of the library if an extra field has to be added to the token in the future.

All the above features make upgrades an easier task.

2.5 Performance Statistics[15]

The main statistics that were used to evaluate the performance of the PML models are latency, utilization and throughput.

Latency is the amount of time it takes for a token to go from point A to point B.

Utilization is the percentage of the total amount of time a device is busy during the whole simulation period.

Throughput is the amount of data processed per unit time.

Chapter 3. DESCRIPTION OF THE MODEL LIBRARY

3.1 Library Structure

The VHDL Performance Model Library has been developed by Honeywell, Inc. and Omniview[8]. The PML_02a version of the library has been used for the present research work. It is composed of the following sub-libraries shown in Figure 3.1.

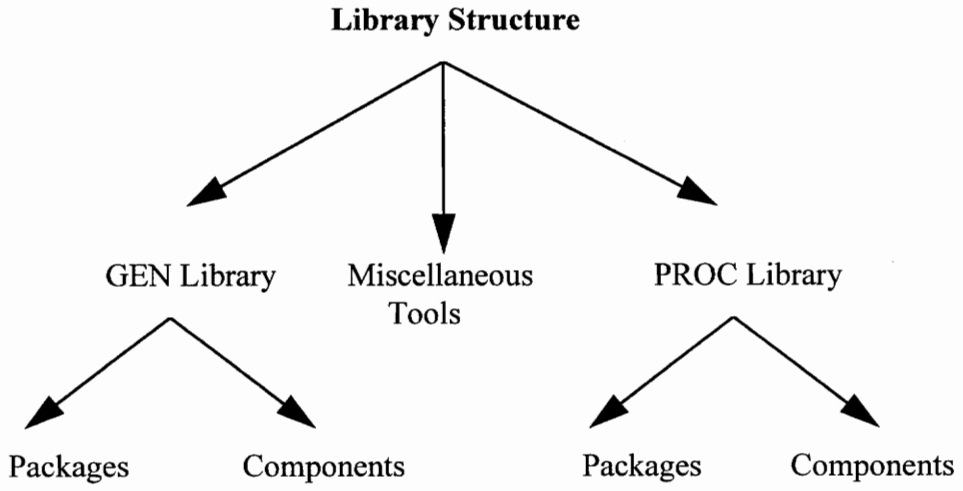


Figure 3.1 The PML VHDL Architecture Library Structure

3.2 The GEN Library

The GEN library has two sub-libraries: *Packages* library and *Components* library. The *Packages* library contain important packages which define the basic means of communication in a model, routines which track simulation statistics, various non-standard math functions, string manipulation procedures, etc. The *Components* library contains different devices present in an architecture like I/O devices, memory devices, etc.

3.2.1 GEN Library Packages

Important GEN Packages

- **Base_Types Package**
- **Statistics Package**
- **Utility Package**
- **Math Package**
- **Strings Package**
- **GDL Package**

Figure 3.2 Main GEN Packages

Two very important packages are explained below.

3.2.1.1 Base_Types Package[5]

This package contains the basic type declarations that are used throughout the model library. The time base and the basic token are defined. Also, the bus resolution function and some token manipulation functions are included. The TOKEN is the means of communication among different components in the model. It is a record structure containing different fields shown in Figure 3.3.

```
TYPE utoken IS
  RECORD
   -- user fields
   parm1_real    : name_type;          -- used to accommodate for the return path
                                       -- of the token(ack.) since route gets
                                       -- stripped off when it reaches the destn.
   parm2_real    : REAL;               -- not used at present
   parm1_int     : INTEGER;            -- not used at present
   parm2_int     : INTEGER;            -- not used at present

  -- control flow
   destination   : name_type;          -- the name of the destination device
   source        : name_type;          -- the name of the source device
   t_type        : token_type;         -- token_type: DATATOKEN, READTOKEN,
                                       -- WRITETOKEN, CONTROLTOKEN
   -- performance fields
   size          : data_size;          -- Size/ number of this token. For instance,
                                       -- if this were a `memory read' token, size
                                       -- would be construed to mean the number
                                       -- of words to read from the memory.
   value         : INTEGER;            -- The token might have a particular value.
                                       -- For ex., a memory write token to a local
                                       -- addre might actually place a value there.

   -- token tracking or statistics fields
   id            : uGIDType;           -- Unique token identification number
   start_time    : TIME;               -- time when token was created

   -- communication fields
   priority      : INTEGER;                  -- The message might have a priority assoc.
                                       -- with it in order to resolve contention.
   state         : State_Type;         -- Indicates state of signal (busy,idle, etc
                                       -- Used by brf.
   protocol      : Protocol_Type;      -- Bus Protocol to use for this token.
                                             -- Used by brf.

   -- user communication tracking and control fields
   collisions    : INTEGER;            --There might be a collision counter to track
                                       -- how many times the token contended with
                                       -- other tokens for the same resource.
   retries       : INTEGER;                  -- This tracks the number of times the token
                                       -- tried to obtain a particular resource
   route         : name_type;          -- In case the token must traverse multiple
                                       -- resources to get to its destination, this
                                       -- contains route string.

  END RECORD;
```

Figure 3.3 The Basic Token Structure[5]

The bus resolution function is used to resolve the value of a signal that has more than one driver. There is an enumerated type called bus status. It has four values:(idle, request, ack, busy). The basic token structure has a field for bus status called *state*. The bus resolution function uses this field, along with fields, *protocol* and *priority* to arbitrate the bus[9]. The bus resolution function is shown in Figure 3.4.

	Idle	Request	Ack	Busy
Idle	First Idle	Request	Ack	Busy
Request	Request	Highest Priority	Ack	Busy
Ack	Ack	Ack	Warning!	Busy
Busy	Busy	Busy	Busy	Highest Priority

Fig 3.4 The Bus Resolution Table

3.2.1.2 GDL Package[8]

The Generic Distribution Language was developed by Honeywell Inc. It is used to describe generic values. This allows time and data dependent behavior to be specified in a straightforward and robust fashion. For instance, one might wish to characterize the behavior of an input sensor. This particular sensor might produce a reading at 30 Hz. Each time data is produced, the size of the data can vary with a uniform distribution between 10 and 20 Kbytes, in 4 Kbytes increments. This would be captured on the VHDL model by adding the following generic values:

```
GENERIC SIZE_INFO       : STRING   := ``UNIFORM RANGE 10 20";
GENERIC THROUGHPUT_INFO : STRING   := ``30";
GENERIC UNIT            : data_size := 4 kbyte;
```

3.2.2 GEN Library Components

3.2.2.1 InDevice[8, 5]

This device generates the tokens which stimulate the whole model under study. These tend to be(from a performance point of view) purely data sources and characterized by the rate at which they produce data. Data can be generated with a variety of desired distributions, like UNIFORM, POISSON, GUASSIAN, etc. For e.g. a sensor. The entity declaration for this component is shown in Figure 3.5.

```
ENTITY Indevice is
GENERIC( DUTYCYCLE_INFO: STRING := "CONSTANT 25";
        INST: STRING := "_In";
        LATENCY_INFO: STRING := "CONSTANT 1000";
        PRIORITY_INFO: STRING := "CONSTANT 0";
        QUEUE_DEPTH : POSITIVE := 2;
        REFERENCE: STRING := "Spec #";
        SIZE_INFO: STRING := "CONSTANT 1";
        SOURCE_INFO: STRING := "ITEM NULL";
        THROUGHPUT_INFO: STRING := "CONSTANT 100";
        TOKEN_PROTOCOL: protocol_type := handshake;
        TRACE: BOOLEAN := FALSE;
        TTYPE: TOKEN_TYPE:= DATATOKEN;
        UNIT: DATA_SIZE := 64 kbyte;
        VALUE_INFO: STRING := "CONSTANT 25";
        PULSES: NATURAL := 1 ;
        SOME_DELAY_INFO: STRING := "CONSTANT 25");
PORT(Dout: INOUT Token);
END Indevice;
```

Figure 3.5 Entity Declaration for Input Device

The Generics are explained below:

- Port *Dout* is the output port for the device
- *DUTYCYCLE_INFO* is a GDL string representing the percentage of a cycle that the device is active generating a particular output.
- *INST* is a string. This is the name of the component instance.
- *LATENCY_INFO* is a GDL string representing the length of time it takes the leading wavefront of a token to propagate through this instance.
- *PRIORITY_INFO* is a GDL string representing the change in priority this instance will inflict upon tokens. This priority value is added to the priority of the incident token to transfer across the communications medium. Higher priority wins over lower priority. The default is "CONSTANT 0" which will leave the incident token unaffected upon output.
- *QUEUE_DEPTH* is an integer which specifies the depth of the token output queue.
- *REFERENCE* is a nonfunctional string purely for documenting any references which might be appropriate or useful to the modeler.

- *SIZE_INFO* is a GDL string indicating how large the output token is. The *UNIT* field specifies the base units of the output token, and this field specifies how many of those units are generated as follows:

  output_token.size := size*unit
- *SOURCE_INFO* is a GDL string indicating the source of the token. This is used to permit the indevice to lie about the source of the token, for instance when sending a request to a memory which will then return the request to a different instance. If this is "ITEM NULL", the inst generic is used for the source field of the token.
- *THROUGHPUT_INFO* is a GDL string representing the rate at which tokens of size *UNIT* are processed through this instance.
- *TOKEN_PROTOCOL* is the protocol of the output token. The default value is handshake, which permits lossless communication for all point-to-point connections.
- *TRACE* is a BOOLEAN. This controls whether this node enters runtime performance information in the output trace file.
- *TTYPE* is of type TOKEN_TYPE. This specifies type of the token which will be generated.
- *UNIT* is of type DATA_SIZE. This represents the granularity of operation of this component. This attribute along with *THROUGHPUT* are used to determine the amount of time a component instance is busy for each incident token. For instance, if a given data processing node which works at the rate of 100 Hz on 64kB data packets receives an input of only 32kB, it will be busy for 5 ms.
- *VALUE_INFO* is a GDL string representing the "value" field in the output token.
- *PULSES* is a natural number. This is used to specify the number of pulses the indevice would generate before waiting for a particular time(*SOME_DELAY_INFO*).
- *SOME_DELAY_INFO* is a GDL string to specify the amount of time the device will wait after the generation of the number of pulses specified by *PULSES*.

3.2.2.2 Outdevice[8]

One way to decompose a model is into the stimulus or inputs, the unit under study or design, and the outputs or its response. This device consumes or sinks the tokens from the model. This outdevice component is intended to accept the responses of the system under study/design. For e.g. hard disk, display.

3.2.2.3 Memory[8]

The memory element is basically a programmable delay with the ability to return tokens to the sender. This iodevice component is intended to manipulate stimulus as part of the system under study/design, and provide some response to the environment. Suppose a processor requests data from the memory, this module waits for a period of time equal to the read access time for the specific amount of data requested and sends an acknowledgment. The entity declaration for this device is shown in Figure 3.6

```
ENTITY memory is
GENERIC ( INST: STRING := "_IO";
        REPLY_TO : STRING := "_Proc";
        DMA_DEVICE_INFO : STRING := "ITEM _Sensor";
        HIT_INFO: STRING := "CONSTANT 1";
        LATENCY_INFO: STRING := "CONSTANT 1000";
        PRIORITY_INFO: STRING := "CONSTANT 1";
        QUEUE_DEPTH : POSITIVE := 2;
        REFERENCE: STRING := "Spec # -1";
        SWITCH_TOKEN: BOOLEAN := FALSE;
        THROUGHPUT_INFO: STRING := "CONSTANT 100";
        TOKEN_PROTOCOL: protocol_type := handshake;
        TRACE: BOOLEAN := FALSE;
        TXFORM_INFO: STRING := "CONSTANT 1.0";
        UNIT: DATA_SIZE := 64 kbyte;
        CAPACITY:DATA_SIZE := 64 Kbyte );
PORT(IO: inout token);
END memory;
```

Figure 3.6 Entity Declaration for the Memory Component

The ports and generics are explained below:

- Port *IO* is both the input and the output of the device.

- *REPLY_TO* is a string which is the name of a device. An interrupt is sent to this device after a DMA. This is normally the processor on the local bus of the memory module.
- *DMA_DEVICE_INFO* is a string which is the name of a device that performs the DMA operation on the memory. After the DMA operation, the memory sends an interrupt to the device specified in the *REPLY_TO generic.*
- *HIT_INFO* is a GDL string representing the probability that a particular memory request was in the memory.
- *INST* is the name of the component instance.
- *LATENCY_INFO* is a GDL string representing the length of time it takes the leading wavefront of a token to propagate through this instance.
- *PRIORITY_INFO* is a GDL string representing the change in priority this instance will inflict upon tokens.
- *QUEUE_DEPTH* is the depth of the token output queue.
- *REFERENCE* is a nonfunctional string purely for documenting any references which might be appropriate or useful to the modeller.
- *SWITCH_TOKEN* is a BOOLEAN that indicates the iodevice is merely to switch the source and destination fields of the token and return it.
- *THROUGHPUT_INFO* is a GDL string representing the rate at which tokens of size *UNIT* are processed through this instance.
- *TOKEN_PROTOCOL* is the protocol of the output token.
- *TRACE* is a BOOLEAN that controls whether this node enters runtime performance information in the output trace file.
- *TXFORM_INFO* is a GDL string representing the factor by which incident data is changed (increased or decreased).
- *UNIT* represents the granularity of operation of this component.

More clearly, the memory access time is calculated as follows

access time = token size/(Unit * Throughput_Info)

where token size is the amount of data to be accessed.

3.2.2.4 Queue[8]

A component must be capable of scheduling outputs based on latency and priority. This output queue model is intended to interface to the outputs of all standard components. The queue will handle ordering and arbitration on external signals to ensure the token is successfully transferred.

3.2.2.5 Bus Interface Unit(BIU)[8, 9]

The bus interface unit is a programmable delay and a token filter. This module contains two parts. One part is called the local side and the second part is called the global side. Figure 3.7 shows the schematic of the BIU[9].

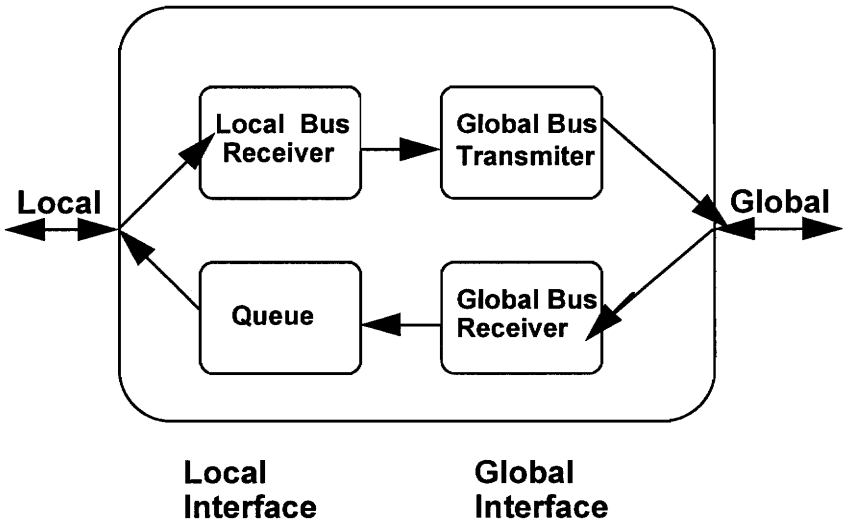


Figure 3.7 Schematic Of The Bus Interface Unit

The local interface has two logical components. The *local bus receiver* handles outgoing messages, the *queue* handles tokens which have been received from the global communications medium and controls arbitration and transfer across the standard token

bus. This latter element is simply the standard token "queue"(Q) component which handles token queuing and output arbitration.

The global interface also has two logical components. The *global bus transmitter* handles outgoing messages, the *global bus receiver* receives tokens from the global bus. Both incoming and outgoing messages have latency delays associated with them. The entity declaration for the BIU is shown in Figure 3.8.

```
ENTITY comm_intH IS
 GENERIC ( ACK_TIME_INFO:    STRING := "CONSTANT 100";
        BUS_TIMEOUT_INFO: STRING := "CONSTANT 100";
        GLOBALID:        STRING := "?*";
        SCREENID:        STRING := "XYZ";
        GLOBAL_PROTOCOL:  PROTOCOL_TYPE := PIBUS32;
        INST:            STRING := "_CIF";
        LOCALID:         STRING := "?*";
        QUEUE_DEPTH:     INTEGER := 2;
        RX_LATENCY_INFO:  STRING := "CONSTANT 100";
        RX_PRIORITY_INFO: STRING := "CONSTANT 0";
        THROUGHPUT_INFO:  STRING := "CONSTANT 100";
        TOKEN_PROTOCOL:   PROTOCOL_TYPE := HANDSHAKE;
        TRACE:           BOOLEAN := FALSE;
        TX_LATENCY_INFO:  STRING := "CONSTANT 100";
        TX_PRIORITY_INFO: STRING := "CONSTANT 0";
        UNIT:            DATA_SIZE := 64 KBYTE );
 PORT ( Local:  INOUT  token;
      Global: INOUT  token );
END comm_intH;
```

Figure 3.8 Entity Declaration for the Bus Interface Unit

The ports and generics are described below:

- Port *Local* is the local (non shared) port of the communication interface element.
- Port *Global* is the global (shared) port of the communications interface. It is used as both a transmitter and receiver for tokens on the shared media (signal) connecting the other communications interfaces. This global port should only be connected to other global ports of communications elements utilizing the same protocol.

- *ACK_TIME_INFO* is a GDL string representing the length of time it takes the intended receiver of a message to acknowledge a request for a connection.
- *BUS_TIMEOUT_INFO* is a GDL string representing the length of time a communications element will wait for an acknowledge of a request for a connection.
- *GLOBALID* is the address of the global port of this communication element. The destination fields of incident tokens will be wildcard matched against this string value to determine whether they should be accepted.
- *GLOBAL_PROTOCOL* specifies the communications protocol for the global or shared side of this communication element.
- *INST* is the name of the component instance.
- *LOCALID* is the address of the local port of this communication element. The destination fields of incident tokens will be wildcard matched against this string value to determine whether they should be accepted.
- *QUEUE_DEPTH* is the depth of the token output queue.
- *RX_LATENCY_INFO* is a GDL string representing the length of time it will take a message incident upon a receiving communication element to propagate through the receiver.
- RX_PRIORITY_INFO is a GDL string specifying the priority which will be placed on the token which are received from the communications medium and placed on the local signal.
- *THROUGHPUT_INFO* is a GDL string representing the rate at which tokens of size *UNIT* are processed through this instance.
- *TOKEN_PROTOCOL* is the protocol of the output token
- *TRACE* controls whether this node enters runtime performance information in the output trace file.
- *TX_LATENCY_INFO* is a GDL string representing the length of time it will take a message incident upon a transmitting communication element to propagate through the transmitter to appear on the global communications media.

- *TX_PRIORITY_INFO* is a GDL string representing the priority of this particular communication instance. This priority value is added to the priority of the incident token to transfer across the communications medium. Higher priority wins over lower priority.
- *UNIT* represents the granularity of operation of this component.

The important parameters for this device are the tx_latency(extracted from *TX_LATENCY_INFO*), rx_latency(extracted from *RX_LATENCY_INFO*) and tx_time.

The tx_latency is the amount of time after which a **token** received on the *local port* of a BIU is put on the *global bus*. The tx_time is the amount of time the global bus is going to remain busy *after* the **token** is put on the bus. The tx_time is calculated as follows:

tx_time = token size(in bytes)/ (*UNIT* * *THROUGHPUT_INFO*)

where *UNIT* is interpreted as word size(bytes/word) and *THROUGHPUT_INFO* as words/sec, while the **token** size is the size of the token data in bytes.

Figure 3.9 shows the relationship between tx_latency and tx_time.

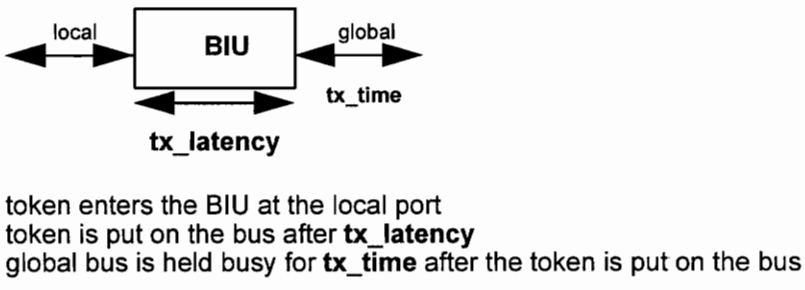
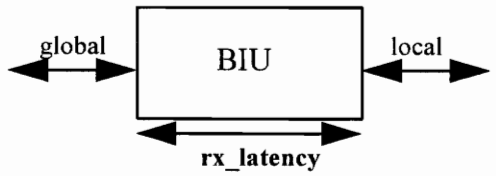


Figure 3.9 Interpretation Of tx_latency and tx_time

Figure 3.10 shows the situation when a token enters the BIU from the *global port*. The token is put on the local queue present on the local side of the BIU after the

global bus becomes idle i.e. after tx_time. The token is put on the local bus after rx_latency.



- token enters the BIU at the global port
- token is put on the local queue after the global bus becomes idle(ie. after **tx_time**)
- token is put on the local bus after **rx_latency**

Figure 3.10 Interpretation Of rx_latency and tx_time

It is important to note that the total time for transmission of a particular token from a source to any destination however distant apart in the architecture is equal to the sum of the tx_time and the latencies as shown in Figure 3.11.

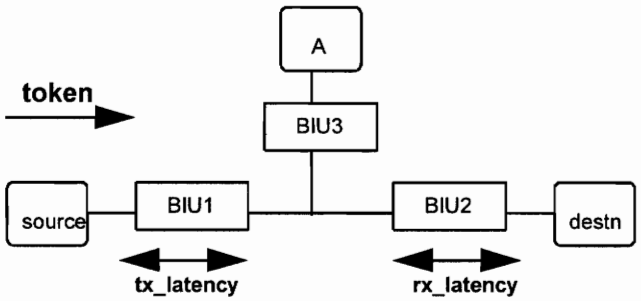


Figure 3.11 Interpretation Of tx_time

In the above figure, the time for the token to reach the "destn" from "source" is

total time = tx_time + tx_latency of BIU1 + rx_latency of BIU2

BIUs are used when a bus is driven by multiple drivers. Figure 3.12 shows a typical interconnection of BIUs.

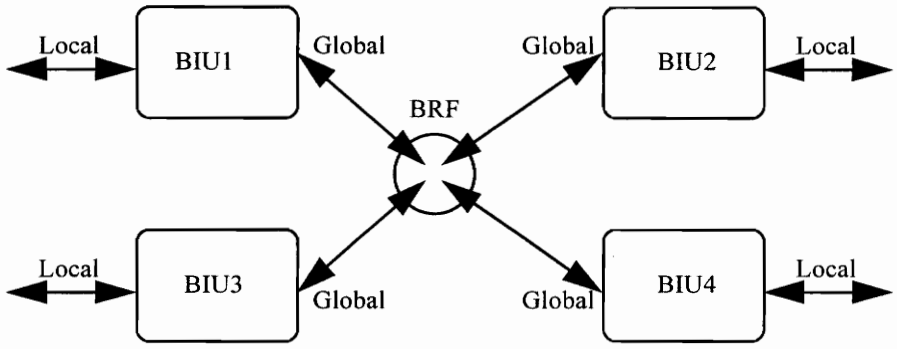


Figure 3.12 Interconnection Of Bus Interface Units

The component devices like memory, indevice, processor, etc are connected to the local port while the global ports face each other and are connected to a common bus. The bus resolution function resolves contention on the common global bus.

3.2.2.6 Crossbar Interconnection Device[9]

When a number of components are connected through a common bus, the bus becomes the communication bottleneck and results in underutilization of the resources. Hence a crossbar was developed to reduce the traffic on a common bus. This device was developed at Virginia tech in an earlier thesis[9].

The crossbar is used to provide routing of messages between various devices. This device has six ports to which external devices are connected. Figure 3.13 shows that the crossbar is a structural model consisting of six fully inter-connected individual routers called crossblocks.

Figure 3.14 shows that each crossblock has one external port and five internal ports connected to the remaining five ports of the crossbar. Each crossblock has two logic modules. The *Receiver*(Rx_Proc) receives a token from the external port and looks up in

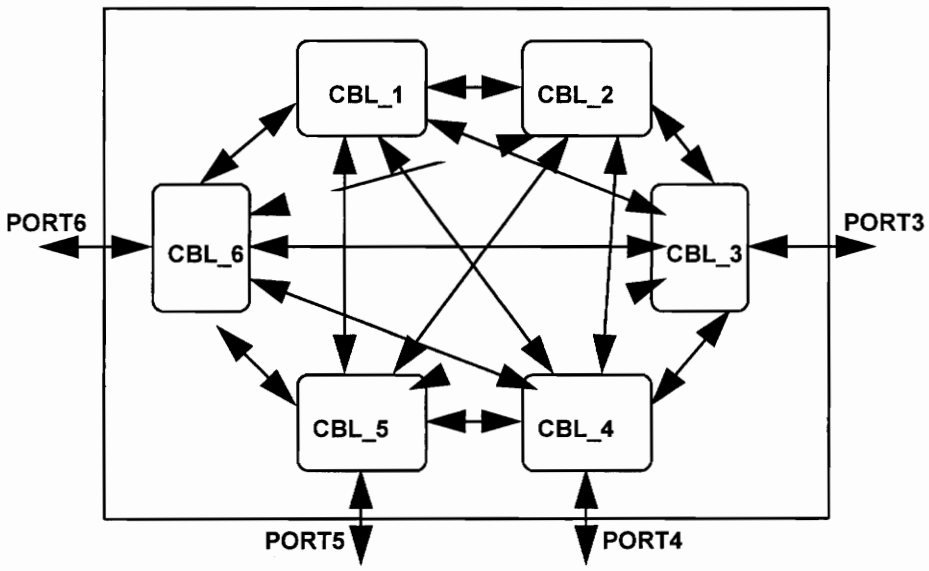


Figure 3.13 Crossbar Interconnection Device

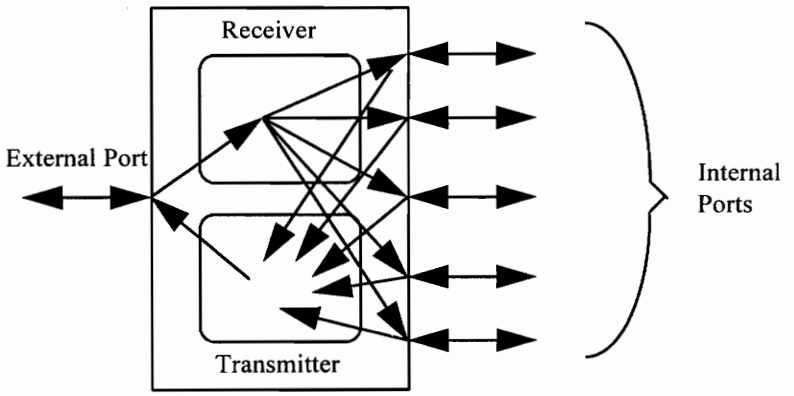


Figure 3.14 Internal Schematic Of The Crossblock Component

a table the port number of the destination specified in the route field of the token. It then sends the token to that particular port. The *Transmitter*(Tx_Proc) receives routed tokens

from the internal signals and puts them onto the external port. The detailed algorithm for each of the logic modules is shown in Figure 3.15[9].

Rx_Proc(The Receiver Module)

- Wait on External Port(Don't false trigger if it is the token from Tx_Proc)
- Set state to Busy
- Get token's route
- Strip out device's name from the route string
- Calculate Latency
- Track token latency
- Look up for device name with names in the look up table
- Delay for Processing time
- If a match is not found in the look up table
  complain seriously
  else
  check if the state of the internal port signal is Idle
  If it is Idle
  send the token out after latency
  wait for tx_time
  else
  wait on the internal signal until it is Idle
- Track utilization
- Reset all internal signals to Idle

Tx_Proc(The transmitter Module)

- Wait on any of the 5 internal ports
- Set signal state to Busy(across all internal signals)
- Track token latency
- Put the token out after latency
- wait for tx_time
- Track Utilization
- Reset all internal signals to Idle

Figure 3.15 Algorithm for the Crossblock component of the Crossbar

When a source is communicating with a remote destination, the path to be taken by the token is specified in the *route* field of the token. Suppose the source is connected to CBAR1 and the destination to CBAR2 and suppose CBAR1 and CBAR2 are connected. Then the *route* field of the token is given by CBAR1.CBAR2.destination. As the token passes through each crossbar, that particular device name is stripped off and the token continues to its destination. In the above case, as the token enters the CBAR1,

CBAR1 is stripped off and the new route is now CBAR2.destination. This continues until the token reaches it destination. This is one of the main functions of the crossbar. Figure 3.16 shows the entity declaration of the crossbar module

```
ENTITY Crossbar IS
 GENERIC (
        INST: STRING := "_CBar";
        NUM_PORTS : INTEGER := 6;
        ROUTING_TABLE : TABLE_TYP; --:= (others => (others => NUL) );
        PROC_TIME_INFO: STRING := "CONSTANT 100";
        TIMEOUT_INFO: STRING := "CONSTANT 100";
        PRIORITY_INFO: STRING := "CONSTANT 0";
        QUEUE_DEPTH : POSITIVE := 2;
        PROTOCOL: PROTOCOL_TYPE := handshake;
        TX_LATENCY_INFO: STRING := "CONSTANT 100";
        THROUGHPUT_INFO: STRING := "CONSTANT 100";
        TRACE: BOOLEAN := FALSE;
        RX_LATENCY_INFO: STRING := "CONSTANT 100";
        UNIT: DATA_SIZE := 64 Kbyte
       );
  PORT ( ExtPorts : INOUT token_vector (1 to NUM_PORTS) );
END Crossbar;
```

Figure 3.16 Entity Declaration for the Crossbar module

Port declarations and generics peculiar to the Crossbar are explained below:

- Port *ExtPorts* are the external ports of the crossbar to which external devices are connected.
- *NUM_PORTS* is an integer and is used to configure the number of ports on the crossbar.
- *ROUTING_TABLE* is the look up table which has the names of the devices connected to the external ports of the crossbar. It is an array of strings which specify the names of the devices.

3.2.2.7 global_intcbar

The global side of the BIU connected to a crossbar is modified and is in a file called global_intcbar-a.vhdl. A new generic called *screenid* was introduced in the BIU.

The *screenid* is somewhat of an inverse of the *globalid.* While the BIU picks up any token which matches its *globalid,* it rejects/ignores the tokens which matches its *screenid.* This is done to make sure that the crossbar does not pick up any token from a port where the source and destination is connected to the same port. The following example makes the concept clear.

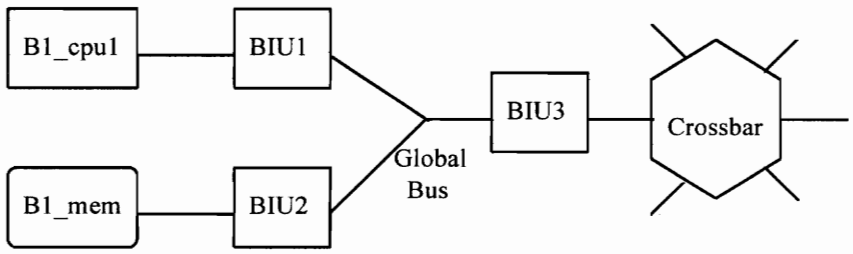


Figure 3.17 Example of the use of global_intcbar

In Figure 3.17, suppose a token is being sent from B1_cpu1 to B1_mem. The generic, *globalid* for BIU1 and BIU2 are set as "B1_cpu1", and "B1_mem" respectively meaning, these BIUs are going to pick up any token which has a destination matching their *globalid* and filter all the other tokens. The *globalid* for BIU3 should be given as "?*" indicating that this BIU is going to pick up all tokens irrespective of the destination. This is done because the crossbar has many components connected to it and it is not possible to give just a single instance name for the *globalid.*

In the above case, the token being sent from B1_cpu1 to B1_mem would also be picked up not only by BIU2 but also by BIU3 which is then routed to the crossbar. In order to avoid this, *screenid* for this BIU3 connected to the crossbar was introduced. The *screenid* in this case is set to "B1_?*" so that this BIU does not pick up the token being sent from B1_CPU1 which has a destination field as "B1_mem" and hence is filtered before reaching the crossbar.

3.2.2.8 biu_four component

The biu_four was developed to connect three different devices and one of the ports of the crossbar to a common bus. This component is a structural inter-connection of four Bus Interface Units(BIUs) and has four ports. The architecture for this component is shown in Figure 3.18.

```
ARCHITECTURE Structural of BIU_FOUR is

COMPONENT comm_int
 GENERIC ( ACK_TIME_INFO:    STRING := "CONSTANT 100";
       BUS_TIMEOUT_INFO: STRING := "CONSTANT 100";
       GLOBALID:       STRING := "?*";
       GLOBAL_PROTOCOL:  PROTOCOL_TYPE := PIBUS32;
       INST:           STRING := "_CIF";
       LOCALID:        STRING := "?*";
       QUEUE_DEPTH:     INTEGER := 2;
       RX_LATENCY_INFO:  STRING := "CONSTANT 100";
       RX_PRIORITY_INFO: STRING := "CONSTANT 0";
       THROUGHPUT_INFO:  STRING := "CONSTANT 100";
       TOKEN_PROTOCOL:   PROTOCOL_TYPE := HANDSHAKE;
       TRACE:          BOOLEAN := FALSE;
       TX_LATENCY_INFO:  STRING := "CONSTANT 100";
       TX_PRIORITY_INFO: STRING := "CONSTANT 0";
       UNIT:           DATA_SIZE := 64 KBYTE );
 PORT ( Local: INOUT token;
     Global: INOUT token );
 END COMPONENT;

 COMPONENT comm_inth
 GENERIC ( ACK_TIME_INFO:    STRING := "CONSTANT 100";
       BUS_TIMEOUT_INFO: STRING := "CONSTANT 100";
       GLOBALID:       STRING := "?*";
       SCREENID:       STRING := "XYZ";
       GLOBAL_PROTOCOL:  PROTOCOL_TYPE := PIBUS32;
       INST:           STRING := "_CIF";
       LOCALID:        STRING := "?*";
       QUEUE_DEPTH:     INTEGER := 2;
       RX_LATENCY_INFO:  STRING := "CONSTANT 100";
       RX_PRIORITY_INFO: STRING := "CONSTANT 0";
       THROUGHPUT_INFO:  STRING := "CONSTANT 100";
       TOKEN_PROTOCOL:   PROTOCOL_TYPE := HANDSHAKE;
       TRACE:          BOOLEAN := FALSE;
       TX_LATENCY_INFO:  STRING := "CONSTANT 100";
       TX_PRIORITY_INFO: STRING := "CONSTANT 0";
       UNIT:           DATA_SIZE := 64 KBYTE );
 PORT ( Local: INOUT token;
```

```
      Global: INOUT  token );
  END COMPONENT;

  FOR ALL: comm_intH USE ENTITY gen.comm_intH(system);
  FOR ALL: comm_int USE ENTITY gen.comm_int(system);

  SIGNAL interconnect :token;
BEGIN

  BIU_TOP: comm_int
  GENERIC MAP(  ACK_TIME_INFO    => ACK_TIME_INFO,
           BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
           GLOBALID         => TOP_GLOBALID,
           GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
           INST             => inst&"_TOP_BIU",
           LOCALID          => TOP_LOCALID,
           QUEUE_DEPTH      => QUEUE_DEPTH,
           RX_LATENCY_INFO  => RX_LATENCY_INFO,
           RX_PRIORITY_INFO => RX_PRIORITY_INFO,
           THROUGHPUT_INFO  => THROUGHPUT_INFO,
           TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
           TRACE            => TRACE,
           TX_LATENCY_INFO  => TX_LATENCY_INFO,
           TX_PRIORITY_INFO => TX_PRIORITY_INFO,
           UNIT             => UNIT )
   PORT MAP( Local => TOP_Local,
          Global=> interconnect);

  BIU_DOWN: comm_int
  GENERIC MAP(  ACK_TIME_INFO    => ACK_TIME_INFO,
           BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
           GLOBALID         => BOTTOM_GLOBALID,
           GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
           INST             => inst&"_DOWN_BIU",
           LOCALID          => BOTTOM_LOCALID,
           QUEUE_DEPTH      => QUEUE_DEPTH,
           RX_LATENCY_INFO  => RX_LATENCY_INFO,
           RX_PRIORITY_INFO => RX_PRIORITY_INFO,
           THROUGHPUT_INFO  => THROUGHPUT_INFO,
           TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
           TRACE            => TRACE,
           TX_LATENCY_INFO  => TX_LATENCY_INFO,
           TX_PRIORITY_INFO => TX_PRIORITY_INFO,
           UNIT             => UNIT )
   PORT MAP( Local => BOTTOM_Local,
          Global=> interconnect);

  BIU_LEFT: comm_int
  GENERIC MAP(  ACK_TIME_INFO    => ACK_TIME_INFO,
           BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
           GLOBALID         => LEFT_GLOBALID,
```

```
        GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
        INST             => inst&"_LEFT_BIU",
        LOCALID          => LEFT_LOCALID,
        QUEUE_DEPTH      => QUEUE_DEPTH,
        RX_LATENCY_INFO  => RX_LATENCY_INFO,
        RX_PRIORITY_INFO => RX_PRIORITY_INFO,
        THROUGHPUT_INFO  => THROUGHPUT_INFO,
        TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
        TRACE            => TRACE,
        TX_LATENCY_INFO  => TX_LATENCY_INFO,
        TX_PRIORITY_INFO => TX_PRIORITY_INFO,
        UNIT             => UNIT )
  PORT MAP( Local => LEFT_Local,
       Global=> interconnect);

 BIU_RIGHT: comm_inth
GENERIC MAP(   ACK_TIME_INFO    => ACK_TIME_INFO,
        BUS_TIMEOUT_INFO => BUS_TIMEOUT_INFO,
        GLOBALID         => RIGHT_GLOBALID,
        SCREENID         => SCREENID,
        GLOBAL_PROTOCOL  => GLOBAL_PROTOCOL,
        INST             => inst&"_RIGHT_BIU",
        LOCALID          => RIGHT_LOCALID,
        QUEUE_DEPTH      => QUEUE_DEPTH,
        RX_LATENCY_INFO  => RX_LATENCY_INFO,
        RX_PRIORITY_INFO => RX_PRIORITY_INFO,
        THROUGHPUT_INFO  => THROUGHPUT_INFO,
        TOKEN_PROTOCOL   => TOKEN_PROTOCOL,
        TRACE            => TRACE,
        TX_LATENCY_INFO  => TX_LATENCY_INFO,
        TX_PRIORITY_INFO => TX_PRIORITY_INFO,
        UNIT             => UNIT )
PORT MAP( Local => RIGHT_Local,
          Global=> interconnect);

END Structural;
```

Figure 3.18 Architecture for the biu_four component

As seen from Figure 3.18, the instantiation *BIU_RIGHT* is the BIU connected to one of the ports of the crossbar module and hence is different from the other three instantiations(as explained in section 3.2.2.7). The entity declaration for this component is shown in Figure 3.19.

```
ENTITY BIU_FOUR  IS
 GENERIC ( ACK_TIME_INFO:              STRING := "CONSTANT 100";
```

```
        BUS_TIMEOUT_INFO:     STRING := "CONSTANT 100";
        TOP_GLOBALID:         STRING := "?*";
        RIGHT_GLOBALID:       STRING := "?*";
        LEFT_GLOBALID:        STRING := "?*";
        BOTTOM_GLOBALID:      STRING := "?*";
        SCREENID:         STRING := "XYZ";
        GLOBAL_PROTOCOL:      PROTOCOL_TYPE := PIBUS32;
        INST:             STRING := "_CIF";
        TOP_LOCALID:          STRING := "?*";
        LEFT_LOCALID:       STRING := "?*";
        RIGHT_LOCALID:        STRING := "?*";
        BOTTOM_LOCALID :    STRING := "?*";
        QUEUE_DEPTH:          INTEGER := 2;
        RX_LATENCY_INFO:      STRING := "CONSTANT 100";
        RX_PRIORITY_INFO:     STRING := "CONSTANT 0";
        THROUGHPUT_INFO:      STRING := "CONSTANT 100";
        TOKEN_PROTOCOL:       PROTOCOL_TYPE := HANDSHAKE;
        TRACE:            BOOLEAN := FALSE;
        TX_LATENCY_INFO:      STRING := "CONSTANT 100";
        TX_PRIORITY_INFO:     STRING := "CONSTANT 0";
        UNIT:             DATA_SIZE := 64 KBYTE );
 PORT ( Left_Local    : INOUT  token;
      Top_Local     : INOUT  token;
      Right_Local : INOUT token;
      Bottom_Local   : INOUT  token );
END BIU_FOUR;
```

Figure 3.19 Entity Declaration for the biu_four component

The port declarations and the generics peculiar to this component are described below.

- Ports *Left_Local*, *Top_Local* and *Bottom_Local* are the left, top and bottom ports of the biu_four component connected to any of the external devices except a crossbar.
- Port *Right_Local* is the port connected to one of the ports of a crossbar.

- *Top_Globalid*, *Left_Globalid*, *Bottom_Globalid* are strings used to specify the names of the devices connected to the corresponding ports.
- *Right_Globalid* is a string which is usually set to "?*" since the right port is connected to the crossbar.

- *Top_Localid, Left_Localid, Bottom_Localid, Right_Localid* are strings. This *localid* screens traffic from the local bus onto the global bus. So only tokens which have a destination field matching the *localid* will be picked up by the BIU and put on the global port. Hence these generics are usually set to "?*", allowing all tokens to be put on the global bus.

It may be further noted that the individual BIUs in the biu_four component inherit their generics from the biu_four component generics.

3.2.2.9 biu_star component

The biu_star component is similar to the biu_four component but is a structural interconnection of *three* individual BIUs. All the generics are similar to that of the biu_four component.

3.3 THE PROC LIBRARY[8]

The PROC library describes the processor component. The processor is the most complex device of the whole model library. It has numerous components and packages associated with it.

The processor model is divided into four major parts:

- the software models
- the scheduler or thread manager
- the processor hardware model
- Dedicated hardware under processor control

The relationship between the components of the processor model is shown in Figure 3.20[8].

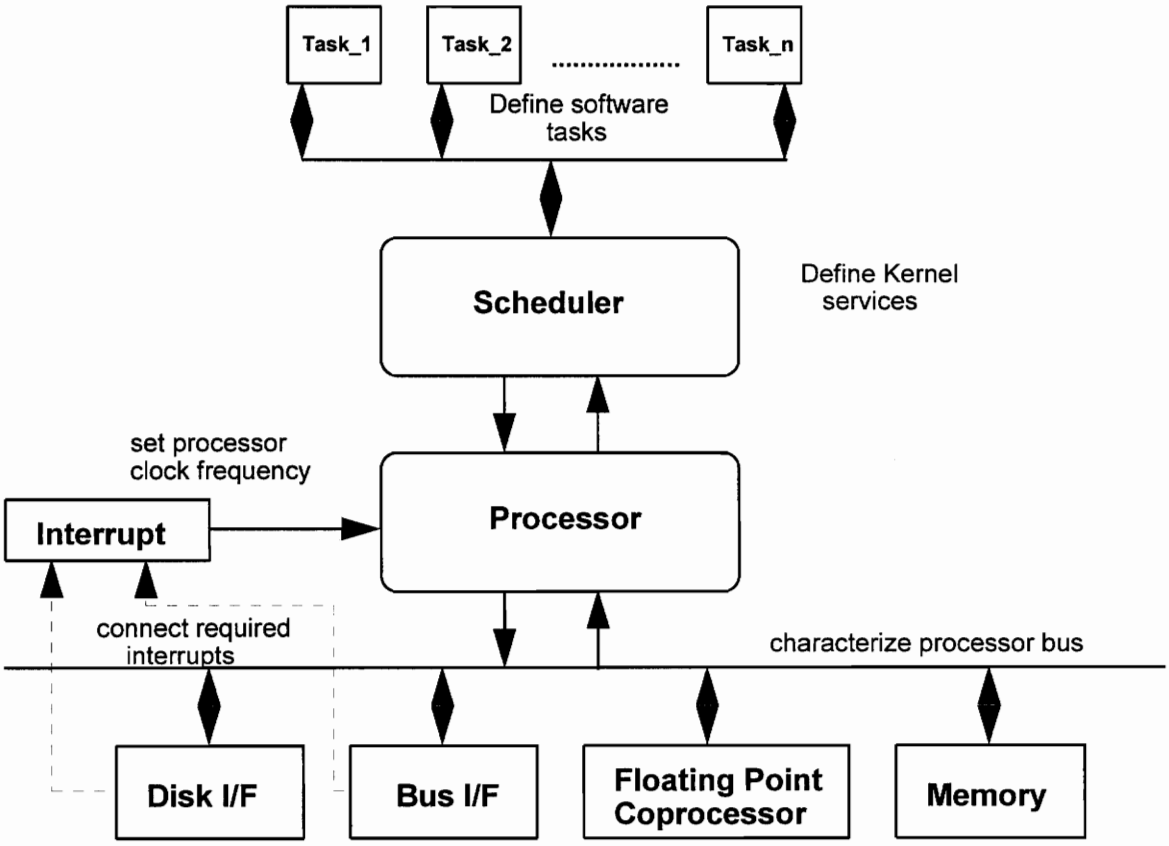


Figure 3.20 Processor model supports multiple characterization levels

The software model is the actual application running on the processor. This is called the application software. The application task software models communicate and request services through the task bus. A process or task is represented by a VHDL procedure that requests services from the scheduler and hardware by calling procedures defined in the packages SchServices of the PROC library. This procedure provides the means for the user to simulate the software activities on the processor.

The SchThreadManager entity consists of a scheduler and a collection of control threads. The scheduler uses the thread to run the software tasks. It passes software request for services that require intervention from the hardware to the CPU for execution. Other requests such as post event, wait for an event, delay, etc. are performed by the scheduler.

The scheduler always chooses for execution the process with the highest priority among those processes that are ready to run.

The processor hardware model receives input control via requests to perform hardware operations from the scheduler and from interrupts from its peripherals. It supports three types of requests. First, the fixed command set controls the internal processor state, allowing the upper level components to request services that change state in the interrupt, cache, etc. The second set, soft instructions, correspond to actual data processing that will be performed by the processor. This will contain commands such as floating-point-multiply, divide, integer add and so on. The final set supports interaction with separate hardware components under the processors control. This includes BIUs, memory, etc.

The processor model can automatically provide reports that detail processor task activity timelines, missed deadline reports, processor utilization, task processor utilization profiles and overall latency.

3.4 Miscellaneous Tools

The Miscellaneous Tools include shell scripts to compile the libraries, shell scripts to run simulations, and postprocessing tools.

The simulation result is stored in a STD_OUTPUT trace file. A sample of the trace file is shown below.

```
LATENCY B16_CPU16_sharc TID := 2 Time := 2430.5 US Accumulated Latency := 5 US
B16_CPU16_sharc ==> B16_Mem
TOKEN B16_CPU16_sharc ==> B16_Mem CurrentTime := 2430.5 US TID := 2 size := 8192
MEMORY B16_Mem CurrentTime := 2637.5 US MemSize (kB) := 1
TOKENCOUNT B16_Mem CurrentTime := 2637.5 US NumberOfTokens := 2
LATENCY B16_Mem TID := 2 Time := 2637.5 US Accumulated Latency := 1000 US B16_Mem ==>
B16_CPU16_sharc
UTIL B16_Mem CurrentTime := 2650.5 US TimeBusy := 14 US Utilization := 0% TimesFired := 2
LastActivity := 13 US
```

```
UTIL B16_CPU16_sharc CurrentTime := 3841.5 US TimeBusy := 1411.5 US Utilization := 36%
TimesFired := 4 LastActivity := 1411 US
3.8415 MS (CPU16, 4) (TASK_1, 4, 101) JSN(none) HW_EXECUTE:Stopping operation.  start time =>
2.4305 MS
            Pole Read
3.8415 MS: UTILIZATION (CPU16, TASK_1, 4) HW_EXECUTE (start => 2.4305 MS)(elapsed =>
1.411 MS)
3.8415 MS (CPU16, 4) (TASK_1, 4, 101) JSN(none) HW_EXECUTE:Resuming thread.
            Pole Read
3.8415 MS (CPU16, 4) (TASK_1, 4, 101) JSN(none) EXECUTE:Scheduling processor operation.
            unpack: 16256 IADDs
3.8415 MS (CPU16, 4) (TASK_1, 4, 101) JSN(none) EXECUTE:Starting processor operation.
            unpack: 16256 IADDs
UTIL B16_CPU16_sharc CurrentTime := 4247.9 US TimeBusy := 1817.9 US Utilization := 42%
```

Figure 3.21 Sample of the STD_OUTPUT file

As can be observed from the above sample, every component in the model under study has its utilization and latency of the token through the component reported into the log file. The STD_OUTPUT file is postprocessed to obtain the activity and latency plots.

The activity plot plots the activity of each component in the architecture model versus time. The shell script for the activity plot extracts the utilization values for different components. The output is a color postscript file. Different colors represent varying degrees of utilization.

The latency plot is similar to the above plot where the latency profiles of the components versus time is plotted. Examples of activity plot and latency plot are shown in Figure 3.22 and Figure 3.23 respectively.

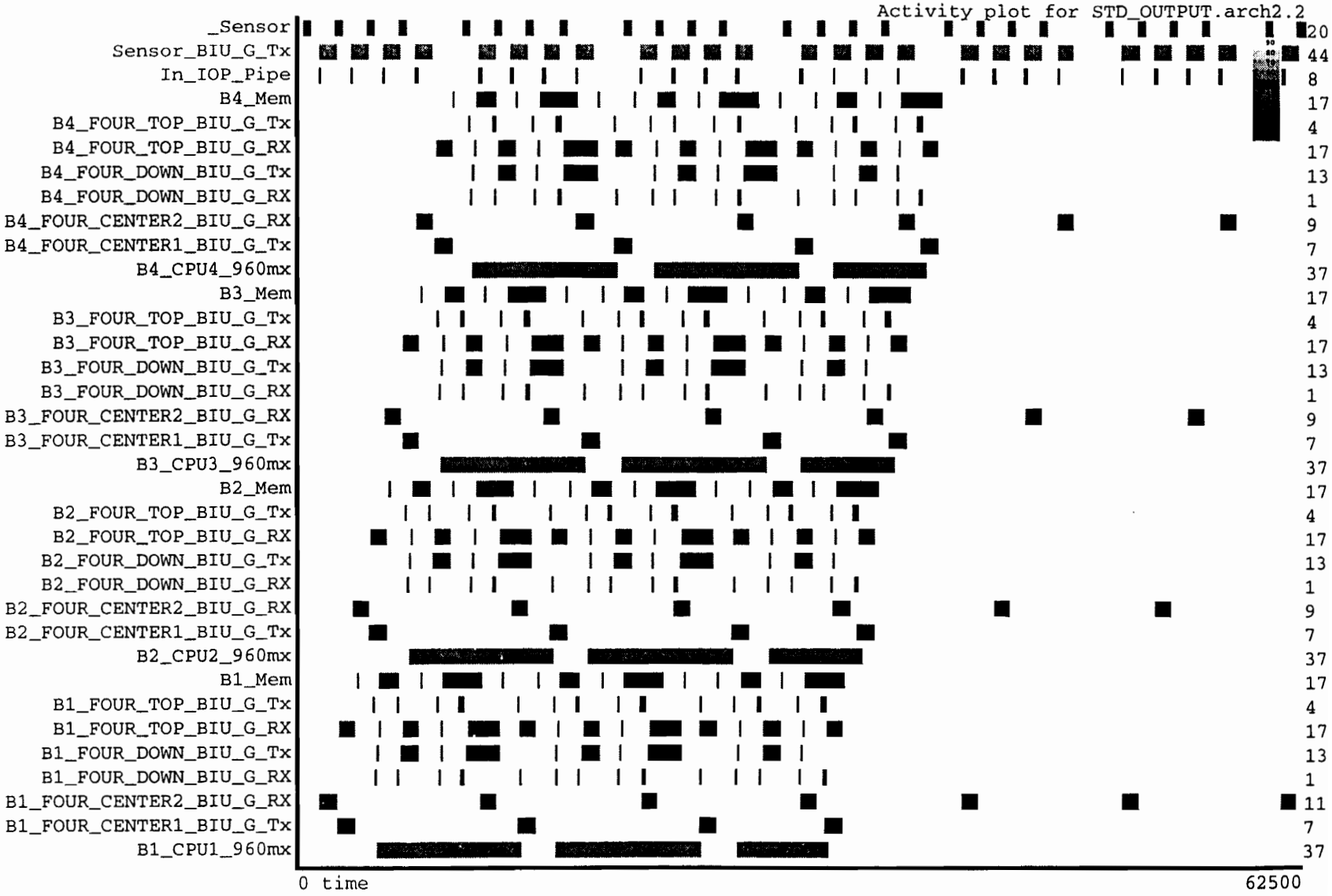


Figure 3.22 Example Of An Activity Plot

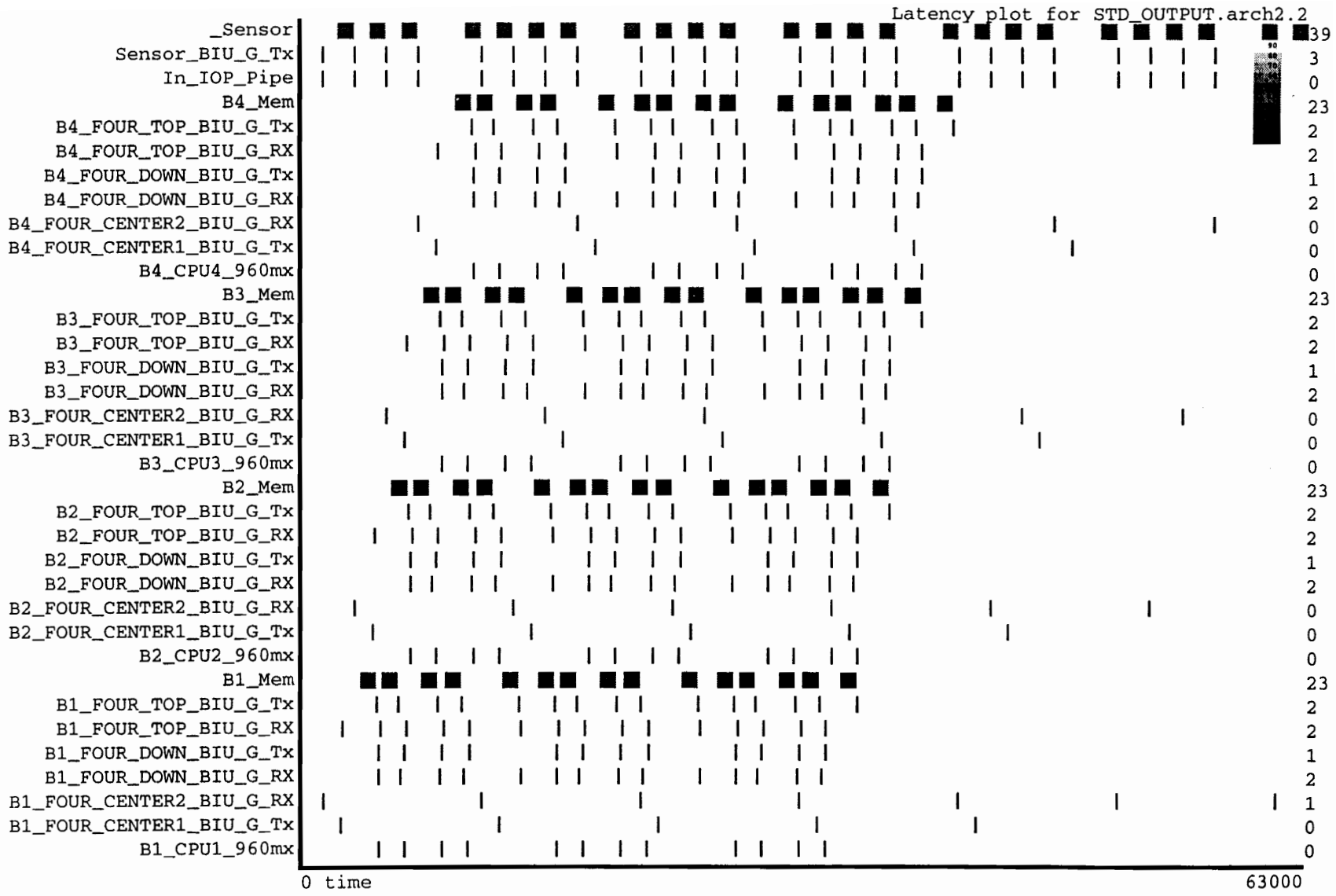


Figure 3.23 Example Of A Latency Plot

3.5 Modifications to the Original PML library

1. Bus Interface Unit(BIU): The global side of the BIU connected to a crossbar is modified and is in a file called global_intcbar-a.vhdl. A new generic called *screenid* was introduced in the BIU. The global_intcbar is described in detail in section 3.2.2.7

In the original BIU, when a token is picked up on the global port, it is put on the local queue on the local side of the BIU *after* global bus became idle i.e. after the time for the total transmission of the token which depends on the size of the token and the generics *Unit* and *throughput* of the BIU. This is modified in the global_intcbar. The token from the global bus is put on the local queue as soon as it sees the token and does not wait until the global bus becomes idle.

2. A new component BIU_FOUR was developed which is basically a structural inter-connection of four individual BIUs. This was developed to connect four different devices to a common bus.

3. All the components connected to a single port of a *crossbar* has been termed as a *bank.* All the components in a single *bank* should have the generic, *inst(*which is basically the name of the device) starting with the same prefix. For example, as shown in figure 3.14, all the components connected to the single port of the crossbar start with "B1_".

4. Modification to the Base_Types Package in the GEN library: In the basic token structure, the field of the token, *route* was an integer and *parm1_real* was a real number. The type definition for both these fields were modified to a "name_type"(which is a string of length 80). The *route* was modified to accommodate for the path of the token through multiple crossbars and the *parm1_real* was to provide for the return route for an acknowledgment.

5. A task can signal the occurrence of an event to another task. A task can be suspended pending the occurrence of one or more events. There were basically sixteen events in the schdefs-p.vhdl file in the PROC library. This way, we could send a maximum of sixteen control tokens between different processors in the architecture. This was increased to twenty-two to accommodate for the large and complex Multiswath Algorithm.

6. The memory device was modified to perform a DMA and also to send an acknowledge token to the sender by making the *parm1_real* field of the receiving token as the *route* field of the acknowledgment token. Also, a new generic, *capacity* was added to specify the capacity of the memory module.

Chapter 4. METHODOLOGY FOR MODEL CONSTRUCTION

4.1 Methodology

This section explains how the components of the model library discussed in chapter 3 were utilized. The following describes a methodology developed for efficient model construction. Figure 4.1 shows a flowchart for model construction.

- A cursory study of all the components in the GEN and PROC library has to be made to get an overall view of the VHDL architecture library. It is important to know the main function of each of the devices.

- The next step would be to analyze the task/problem and translate the task into hardware and software specifications.

- The above step gives an insight into the hardware components needed to run a particular software algorithm. A complete study of all the components needed to construct the hardware model should be done. One has to have a good understanding of all the generics that go with the hardware components.

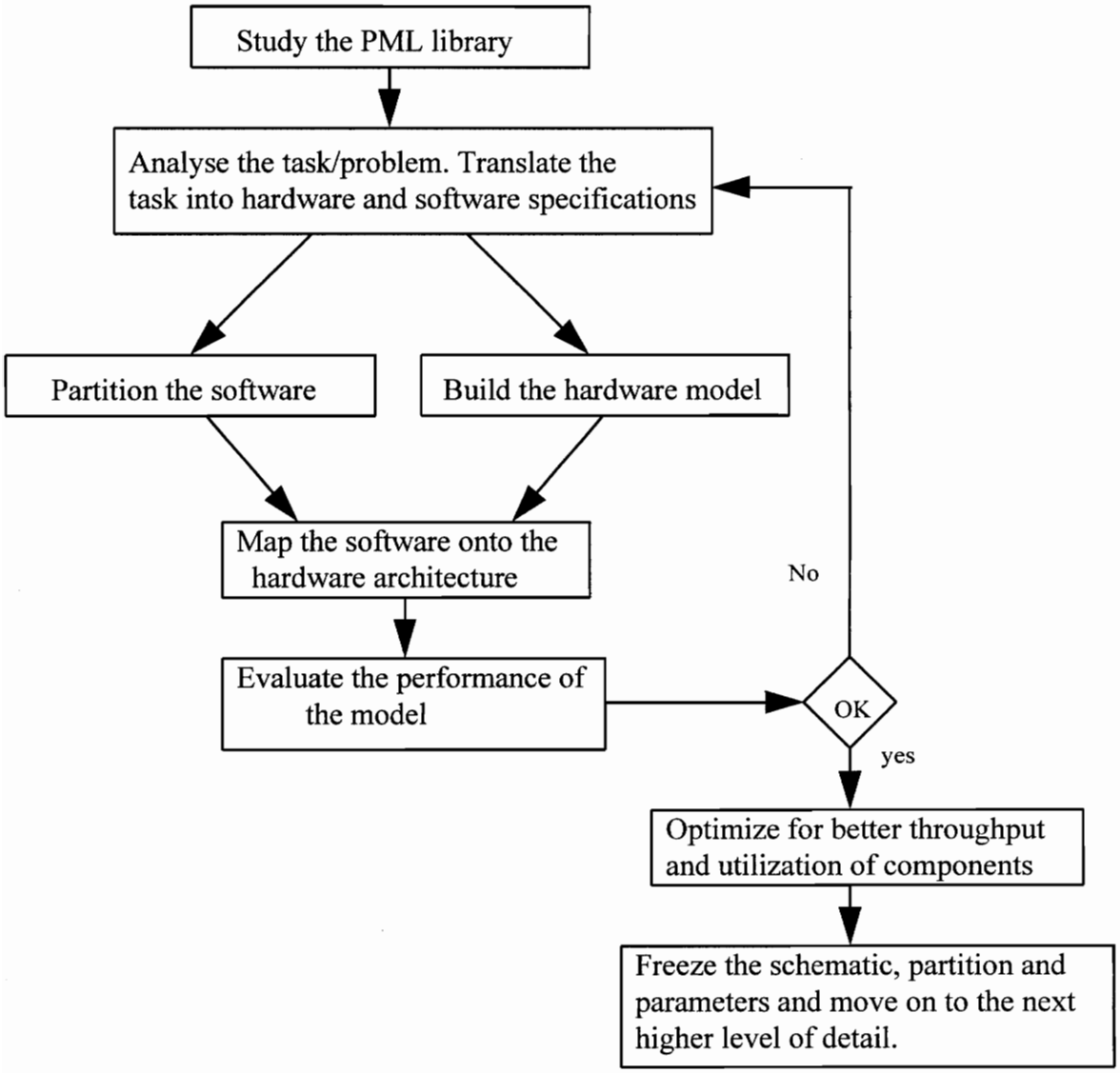


Figure 4.1 The Model Construction Flowchart

- Given the software specifications, the software has to be partitioned to be mapped onto the processors. Study of the software has to be made to check if there is any inherent parallelism within the software algorithm.

- The next step is to make a reasonable choice of the architecture schematic and the device parameters or generics. Then the required hardware components are selected from the library. New components should be developed if need arises. A structural inter-connection of the hardware components is made to obtain the required architecture.

- Given the partitioned software algorithm, different tasks(partitions) of the algorithm have to be mapped onto different processors. This is called the application software which is written as a VHDL code in the file application_sw-a.vhdl in the PROC library.

- Run simulations and analyze the results. Set different generics(device parameters) and observe the behavior of the system.

- A detailed analysis of the results has to be made to check for validity.

- Make changes to the architecture and/or device parameters if the results are not satisfactory and repeat steps from translation of the task into hardware and software specifications.

- Once the results are analyzed to be sound, optimization should be done for better throughput and utilization of the system components.

- Freeze the hardware architecture, the device parameters and the software partitioning and move onto the next level of design process involving a greater degree of detail.

The above methodology was applied to build the seventeen processor architecture.

4.2 The Seventeen Processor Raceway Architecture

The RACE™ architecture was introduced in May 1993 by Mercury Computer Systems[10]. It has been deployed in systems scaling from four to more than 700 processors. As technology advances(eg. in processor design, chip packaging and software tools) it becomes evident that an architecture must be able to rapidly evolve to maintain parity with the available technology. The RACE architecture for multiprocessors provides a high-bandwidth, low latency system for solving real-time applications[10].

In the present research study, a seventeen processor mercury raceway architecture has been developed. Two algorithms, a Range-Processing Algorithm and a Multiswath Processing Algorithm are mapped onto this raceway architecture. The schematic of the architecture model is shown in Figure 4.2.

As can be seen from Figure 4.2, the model contains an input device which models the radar. This is the stimulus to the system under study. This architecture has a provision for a DMA(direct memory access) where the radar writes directly to the memory without the intervention of the CPU. The generic **some_delay_info** for the input device is set to specify the time between two pulses. The lower the value, the faster the generation of pulses. But, enough time must elapse between pulses to insure that the first pulse has time to be processed and the bus to become idle before the arrival of the next pulse, else the pulse(token) would be lost waiting for the bus to become idle.

The output device, as shown in Figure 4.2, is a device to sink the tokens from the model. The results of an algorithm are written into the output device. It may be further

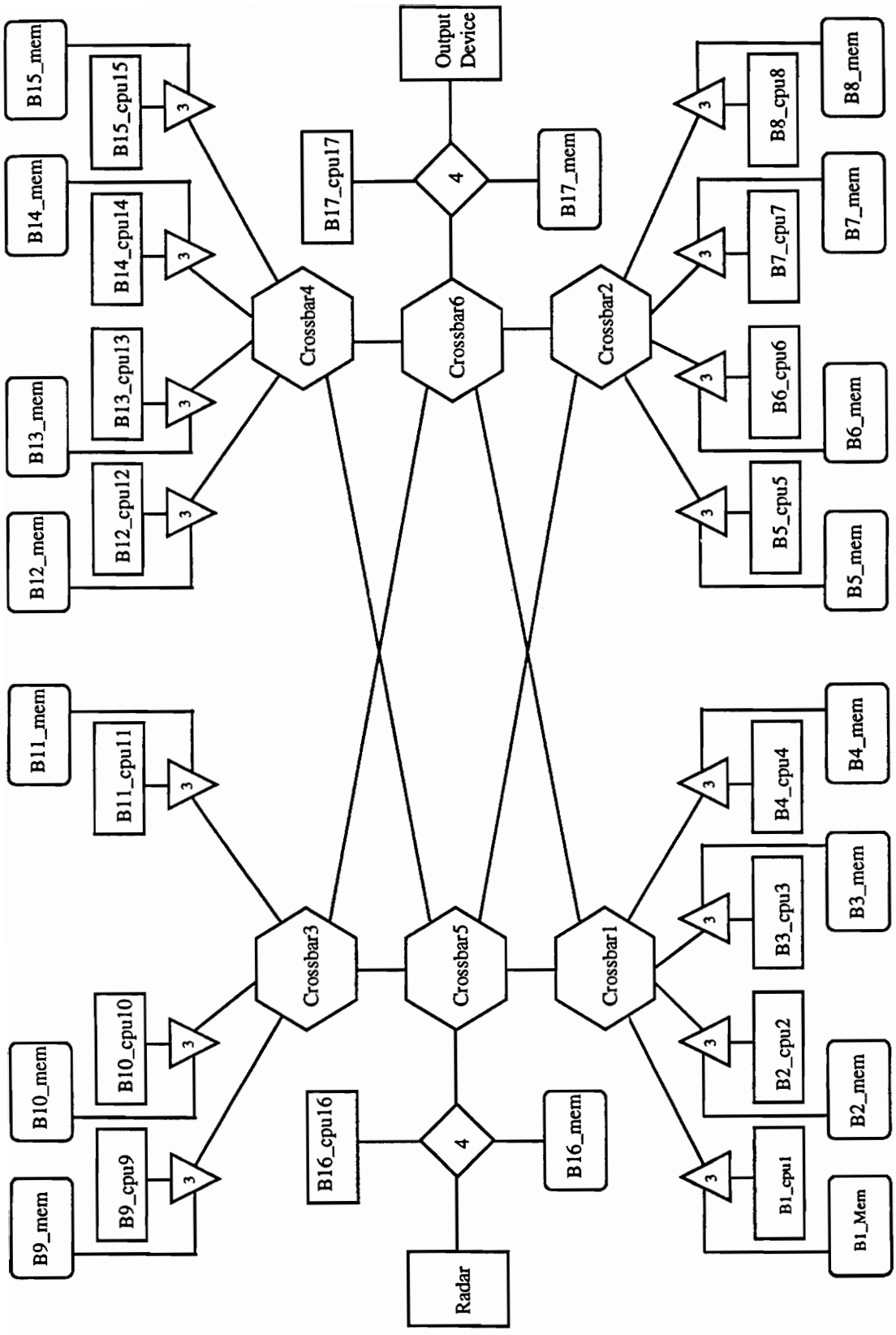


Figure 4.2 Schematic Of The Seventeen Processor Raceway Architecture

noted that this device has no provision for sending an acknowledgment to the sender.

There are seventeen processors(CPUs) in total in the architecture. In Figure 4.2, these processors are represented as B*i*_cpu*i*(*i* = 1 to 17). Each processor has its own local memory module connected to it. The memory is shown as B*i*_mem*i*(*i* = 1 to 17).

There are six crossbar modules which are used to connect the seventeen processors together. They are shown by hexagonal blocks in Figure 4.2. These crossbars help prevent congestion on the global bus and facilitate faster inter-device communication.

There are fifteen biu_star clusters(refer to Figure 4.3) in the raceway architecture. The internal details of the biu_star are explained in Chapter 3. The biu_star component is used to connect a processor, its local memory and one of the ports of the crossbar to a common bus. These biu_star components are represented by triangle blocks in Figure 4.2(with "3" written inside the block).

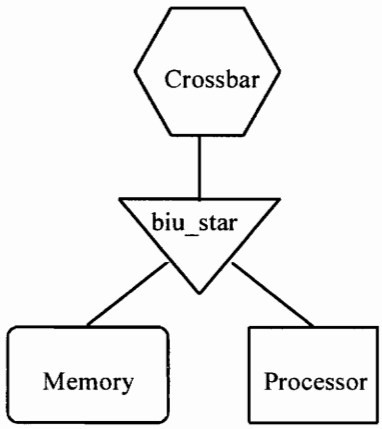


Figure 4.3 A biu_star cluster

There are two biu_four components in the seventeen processor architecture. In Figure 4.2, these components are represented as diamond blocks(with "4" written inside

the block). As explained in Chapter 3, the biu_four has four ports and is used in the present architecture at the input and output of the model. The first biu_four is employed to connect the radar, B16_cpu16, B16_mem and Crossbar5 to a common bus. The second biu_four is used at the output to connect the output device, B17_cpu17, B17_mem and Crossbar6 to a common bus.

4.3 The SAR Range-Processing Algorithm

The Range-Processing Algorithm is a portion of the Synthetic Aperture Radar(SAR) algorithm. This software is mapped onto the Seventeen Processor Architecture described in section 4.2.

4.3.1 The Task Mapping

The simulation begins with the input device, *radar* sending a pulse(termed pole). The *radar* writes the data directly into the memory(by Direct Memory Access) of the *input processor*(B16_cpu16). This memory, B16_mem sends an interrupt to the B16_cpu16 when it begins its processing. The B16_cpu16 unpacks the data received from each radar pulse and sends the processed data to the four *range processors*(B1_cpu1, B2_cpu2, B3_cpu3, B4_cpu4) on rotation. These *range processors* process the data in parallel. Once the range processing is done, the processed data is sent to the *corner turn processor*(B11_cpu11). The B11_cpu11 waits on 512 processed radar pulses and carries out the bin processing. This is then sent to the *image processor*(B15_cpu15) for assembling the image which in turn sends the processed data to the *output processor*(B17_cpu17) which stores the image in the *output device*.

Figure 4.11 shows the complete sequence of processing steps for each of the individual processors. This data has been produced by a mapping tool created at the Research Triangle Institute located in Research Triangle Park, NC. These steps must be converted to PML code(Performance Model Library code). This conversion is currently

done manually. Automation of the conversion process is currently being studied. The PML code is called the *application software* and must be written in the application_sw-a.vhdl file in the PROC library. The actual mapping of these tasks onto the processors is done in a processor configuration file called processor-c.vhdl in the PROC library.

To illustrate the process of converting the specifications into PML code, a portion of the hardware model of Figure 4.2 is extracted as Figure 4.4. Cbar5 and cbar1 are crossbars, B16_mem and B2_mem are memory modules, B16_cpu16 is the input processor and B2_cpu2 is one of the range processors.

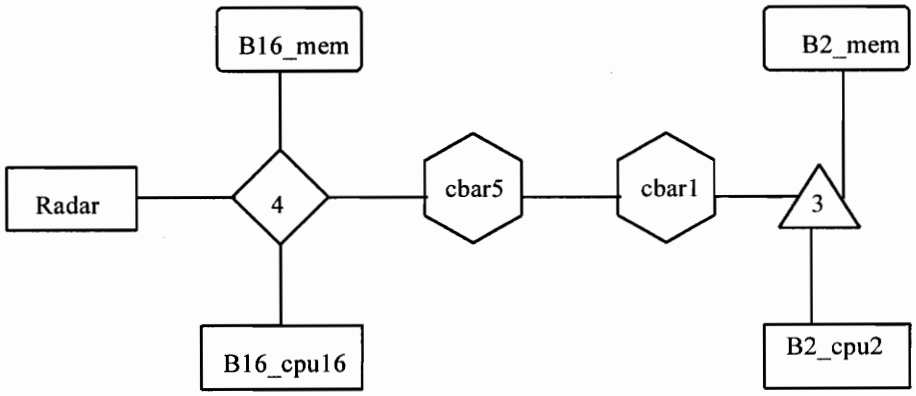


Figure 4.4 Part Of The Hardware Model Of Figure 4.2

The PML code is written in VHDL. First consider steps 2 and 3 for Task 1 on the input processor as shown in Figure 4.11. These steps are repeated below in Figure 4.5.

Task 1 for input processor(B16_cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
2	pole_read	16 bytes	16256 bytes	Read	cpu16	B16_mem
3	pole_read_ack	16256 bytes	XX	Data	B16_mem	cpu16

Figure 4.5 Sample of the software application specification for a Read Operation

The Figure 4.5 specification is interpreted as follows. In step 2, cpu16 must send a 16 byte read request to the B16_mem module. The amount of data to be read by the processor is 16256 bytes. In step 3, the B16_mem must send an acknowledgment to the processor.

To convert step 2 into PML code, a temporary token is built as shown in Figure 4.6. The *destination* field for the token is "B16_mem", the *t_type* field(token type) is READTOKEN, the *size* field is 16 bytes(size of token) and the *value* field is 16256 bytes(number of bytes to be read). The *route* field is just "B16_mem" since this token has a local destination and does not move through the crossbars. Procedure "Hwexecute" in the Schservices package of the PROC library is called to send the token to the destination. This is shown below:

HWEXECUTE(me, send, receive, msg, hw_op, boolean, num_instr, comment).

where *me* is the task id number, *send* is the outgoing signal to the scheduler, and *receive* is the incoming signal from the scheduler. These three parameters should not be changed by the user. *Msg* is the hardware token sent down to the processor core, *hw_op* is the hardware operation to execute on the processor, *boolean* is set to TRUE if the processor is waiting for an acknowledgment from the destination before proceeding to its next execution, *num_instr* is the number of hardware instructions to be simulated on the processor and is presently set to an arbitrary value of 1 throughout the application software and *comment* is a textual comment to be added to the log record.

Since the above specification requires the processor to wait for an ack from B16_mem, the parameter, *boolean* in the procedure "Hwexecute" is set to *true.* The hardware operation to be executed is a read operation and hence the *hw_op* parameter is set to HW_READ.

```
temp_token.destination := (OTHERS => NUL);
temp_token.destination(1 TO 7) := "B16_Mem";
```

```
temp_token.route := (OTHERS => NUL);
temp_token.route(1 TO 7)      := "B16_Mem";
temp_token.size       := 16 byte;     -- read request
temp_token.value      := 16256;   --
temp_token.t_TYPE     := READTOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_READ, TRUE, 1, "Pole Read");
```

Figure 4.6 Token For a Read Operation

Figure 4.7 shows the specification for a write operation in which processor cpu16 has to write 8140 bytes of data into B2_mem module. As shown in Figure 4.8, the PML code creates a temporary token in which the destination is "B2_mem", the *t_tpye* is set to WRITETOKEN, the *size* field is set to the actual amount of data to be written(8140 bytes), the *value* is a don't care and is set to 1000 throughout. The *route* field is "CBar5_?*.CBar1_?*.B2_Mem" since the token has to pass through crossbars Cbar5 and Cbar1 to reach its destination. The *parm1_real* field is the route for an acknowledge token from the B2_mem module to the cpu16 processor and hence is set to "CBar1_?*.CBar5_?*.B16_CPU16_?*". The "_?*" after the name of the crossbars in the *route* and *parm1_real* fields is needed because the name of the process being executed by the crossbar(either Transmission, Tx or Receiving, Rx) is concatenated with the crossbar name by the simulator at run time. The "_?*" after B16_CPU16 is needed because the kind of processor being used(e.g. pentium, 386, etc) is concatenated with B16_CPU16 at run time.

Since the software specification requires the processor to wait for an ack from B2_mem, the parameter, *boolean* in the procedure "Hwexecute" is set to *true.* The hardware operation to be executed is a write operation and hence the *hw_op* parameter is set to HW_WRITE.

Task 1 for input processor(B16_cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
7	odd_write_req	8140 bytes	XX	Write	cpu16	B2_mem
8	odd_write_ack	16 bytes	XX	Ack	B2_mem	cpu16

Figure 4.7 Sample of the software application specification for a Write Operation

```
temp_token.destination(1 TO 6) := "B2_Mem";
temp_token.route(1 TO 24)      := "CBar5_?*.CBar1_?*.B2_Mem";
temp_token.parm1_real(1 to 30) := "CBar1_?*.CBar5_?*.B16_CPU16_?*";
temp_token.size      := 8140 byte;   -- read request
temp_token.value     := 1000;   -- don't care
temp_token.t_TYPE    := WRITETOKEN;
msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, TRUE, 1, "Odd Write");
```

Figure 4.8 Token For a Write Operation

Figure 4.9 shows the specification for an interrupt operation. In this case, cpu16 has to send an interrupt token to processor cpu2. As shown in Figure 4.10, the PML code creates a temporary token in which the *t_tpye* is set to CONTROLTOKEN, the *size* field is set to 16 bytes(the size of the token), the *value* field is a very important field and should be set with care. The *value* field defines the name of the interrupt or event being sent. This is explained in detail in section 5.2.2. The *value* field is the position of the event name in an array of events pre-defined in the Scheduler Definitions Package of the PROC library. The position of Event_2 in the array is 20. Hence, the *value* is assigned to be 20. In case the event name happens to be Event_3, then the *value* field is assigned 21 as the position of Event_3 in the array is 21 and so on. The *route* field is "CBar5_?*.CBar1_?*.B2_CPU2_?*" since the token has to pass through crossbars Cbar5 and Cbar1 to reach its destination. The processor in the PML library has no provision to send an acknowledgment on receipt of a control token. Hence the parameter *boolean* for the procedure "Hwexecute" is set to *false*. Since *parm1_real* field is the route for an acknowledge token, and since a processor does not send an acknowledgment, this field is a don't care.

Task 1 for input processor(B16_cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1 1	SEND_EVENT	16 bytes	***Event_2***	Control	cpu16	cpu2

Figure 4.9 Sample of the software specification for an Interrupt Operation

```
   temp_token.destination(1 TO 10) := "B2_CPU2_?*";
   temp_token.route(1 TO 28)      :=  "Cbar5_?*.CBar1_?*.B2_CPU2_?*";
   temp_token.size        := 16 byte;  --  don't care
   temp_token.value       := 20 ;      -- EVENT_2
   temp_token.t_type      := CONTROLTOKEN; -- interrupt_token
   msg.proc_msg.token_msg := temp_token;
HwExecute(me, send, receive, msg, HW_WRITE, FALSE, 1, " Sending EVENT_2..");
```

Figure 4.10 Token For an Interrupt Operation

The above method was applied to convert the set of specifications shown in Figure 4.11 into PML code.

Task 1 for RADAR

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	pole	16256 bytes	XX	Write	RADAR	B16_mem
2	pole_ack/intr.	16 bytes	***Event_1***	Control	B16_mem	cpu16

Task 1 for *Input Processor*(B16_cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_1*					
2	pole_read	16 bytes	16256 bytes	Read	cpu16	B16_mem
3	pole_read_ack	16256 bytes	XX	Data	B16_mem	cpu16
4	Unpack	(16256 clocks)--- processing on cpu16				
5	even_write_req	8140 bytes	XX	Write	cpu16	B16_mem
6	even_write_ack	16 bytes	XX	Ack	B16_mem	cpu16
7	odd_write_req	8140 bytes	XX	Write	cpu16	B16_mem
8	odd_write_ack	16 bytes	XX	Ack	B16_mem	cpu16

the following steps are processed by cpu16 from each pulse and sent to the *range processors* successively, cpui(where i=1,2,3,4).

9	even_write	8140 bytes	XX	Write	cpu16	Bi_mem
10	even_write_ack	16 bytes	XX	Ack	Bi_mem	cpu16
11	SEND_EVENT	16 bytes	***Event_2***	Control	cpu16	cpui
12	odd_WRITE	8140 bytes	XX	Write	cpu16	Bi_mem

13	odd_WRITE_ack	16 bytes	XX	Ack	Bi_mem	cpu16
14	SEND_EVENT	16 bytes	***Event_3***	Control	cpu16	cpui

Task 1 for the *Range Processors* , cpui(where I= 1, 2, 3, 4)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_2*					
2	even_read	16 bytes	8140 bytes	Read	cpui	Bi_mem
3	even_read_ack	8140 bytes	XX	Data	Bi_mem	cpui
4	Even FIR	(6087 clocks) -- processing on cpui				

Task 2 for the *Range Processors* , cpui(where I= 1, 2, 3, 4)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_3*					
2	odd_read	16 bytes	8140 bytes	Read	cpui	Bi_mem
3	odd_read_ack	8140 bytes	XX	Data	Bi_mem	cpui
4	Odd FIR	(6087 clocks) -- processing on cpui				
5	Form Complex (4060 clocks)					
6	weight_read_req	16 bytes	16240 bytes	Read	cpui	Bi_mem
7	weight_read_ack	16240 bytes	XX	Data	Bi_mem	cpui
8	Complex VPM	(8120 clocks)				
9	Zero Pad	(18 clocks)				
10	Complex FFT	(67584 clocks)				
11	cpulse_write_req	16384 bytes	XX	Write	cpui	Bi_mem
12	cpulse_write_ack	16 bytes	XX	Ack	Bi_mem	cpui
13	cpulse_read_req	16 bytes	16384 bytes	Read	cpui	Bi_mem
14	cpulse_read_ack	16384 bytes	XX	Data	Bi_mem	cpui
15	cpulse_WRITE	16384 bytes	XX	Write	cpui	B11_mem
16	cpulse_WRITE_ack	16 bytes	XX	Ack	B11_mem	cpui
17	SEND_EVENT	16 bytes	***Event_4***	Control	cpui	cpu11

Task 1 for the *Corner Turn Processor* , B11_cpu11

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for 512 consecutive Event_4s*					
FOR J IN 1 TO 2048 -- Process range bins in the frame						
2	bin_read	16 bytes	8192 bytes	Read	cpu11	B11_mem
3	bin_read_ack	8192 bytes	XX	Data	B11_mem	cpu11
4	bin_WRITE	8192 bytes	XX	Write	cpu11	B15_mem
5	bin_WRITE_ack	16 bytes	XX	Ack	B15_mem	cpu11
6	SEND_EVENT	16 bytes	***Event_5***	Control	cpu11	cpu15
END LOOP; -- range bins in the frame						

Task 1 for the *Image Processor* , B15_cpu15

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for 2048 consecutive Event_5s*					
2	bin_read	16 bytes	8192 bytes	Read	cpu15	B15_mem
3	bin_read_ack	8192 bytes	XX	Data	B15_mem	cpu15
4	Assemble Image (2097152 clocks)					
5	image_WRITE	8388608 bytes	XX	Write	cpu15	B17_mem
6	image_WRITE_ack	16 bytes	XX	Ack	B17_mem	cpu15
7	SEND_EVENT	16 bytes	***Event_6***	Control	cpu15	cpu17

Task 1 for the *Output Processor* , B17_cpu17

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_6*					
2	image_read	16 bytes	8388608 bytes	Read	cpu17	B17_mem
3	image_read_ack	8388608 bytes	XX	Ack	B17_mem	cpu17
4	image_STORE	8388608 bytes	XX	Write	cpu17	OutDevice

Figure 4.11 Sequence Of Steps for the Different Processors in The Model

4.3.2 Results

One cycle of simulation is defined as the time between the arrival time of the first pulse and the time at which the result is written into the output device. For the Range-Processing Algorithm, one simulation cycle consists of 512 radar pulses being sent to the four *range processors* on rotation. The processed data is sent then to the *corner turn processor* for bin processing. A time difference of 700 micro seconds was set between the arrival of two radar pulses so the first pulse is finished processing before the arrival of the next one. When this time was decreased, the Bus Interface Units at the input of the architecture timed out while waiting for the bus to become idle and the radar tokens were lost. If this time is increased, we are obviously under-utilizing the system. Hence the optimum time is constrained by the communication bottle-necks and the processing speed.

In all the tests described below, all the processors are running at 40 MHz(25 ns clock period). The word size is set to 4 byte long. The memory access time is set to .025

micro seconds per word and the transfer time for all the transfer devices(crossbar, biu_four, biu_star) is set to .007 micro seconds per word.

4.3.2.1 Test 1

In this test, four radar pulses were sent and only a portion of the algorithm was run. This is to see clearly the **parallelism** in the four *range processors*. The delay between the arrival of two pulses, i.e. the generic *some_delay_info* for the radar is set to 700 micro seconds.

The output trace file is postprocessed to obtain the activity and the latency plots. The activity plot shows the percentage of the time each device in the architecture is being utilized during the whole simulation time. The latency_plot shows the latency distributions of the different devices. Figure 4.12 shows the activity_plot and Figure 4.13 shows the latency_plot for this particular simulation.

From Figure 4.12, it is clear that the four range processors are running in parallel. The utilization of the four *range processors* is 46 % while that of the *input processor* is 41%. Further, the bus connecting crossbar5, crossbar3 and the bus connecting crossbar5, crossbar1 is being utilized 4% of the total time.

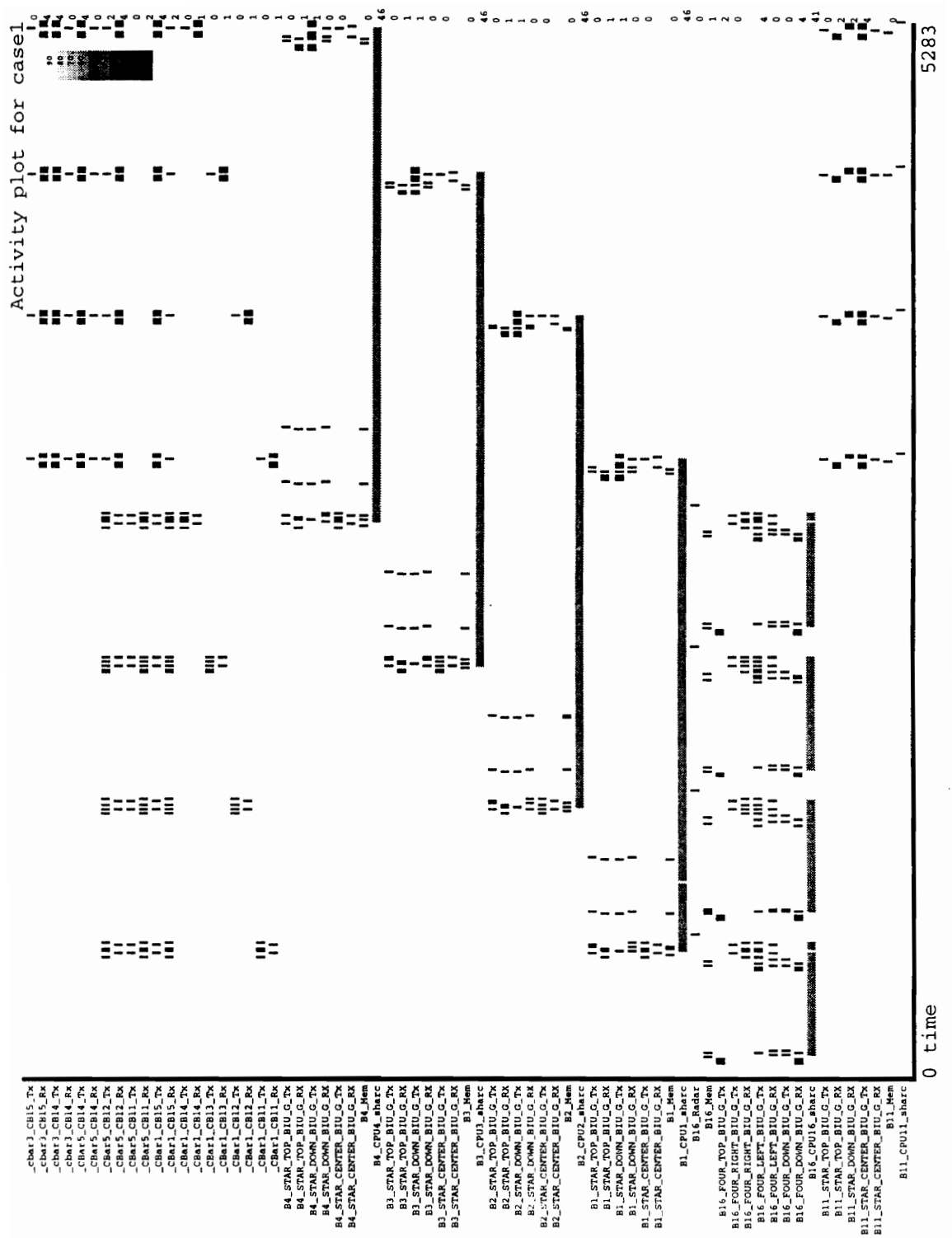


Figure 4.12 Activity Plot For Test 1

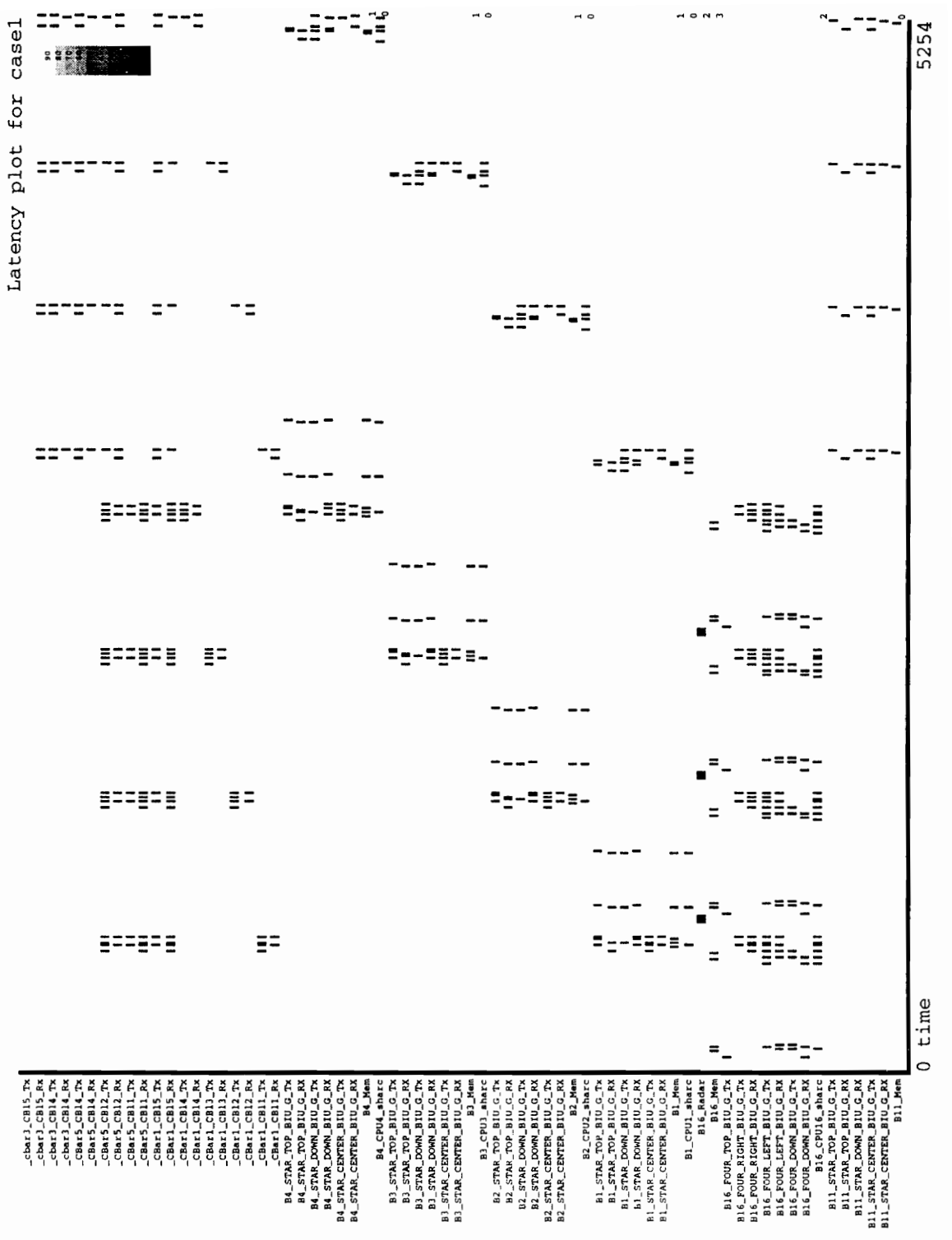


Figure 4.13 Latency Plot For Test 1

4.3.2.2 Test 2

The actual algorithm required sending a total of 512 radar pulses. To simulate such an amount is estimated to take around four days. Since this is not practical, the loops within the algorithm have been reduced by a factor of 64.

Hence, in the second simulation(refer to the specifications in Figure 4.11), 8 pulses(instead of the original 512) were sent by the radar. The *corner turn processor* processes the data 32 times in a loop(instead of the original 2048). The *image processor* waits on this data and assembles the image taking 32768 clock cycles(instead of the original 2097152 clocks) and writes 131072 bytes of data to the *output processor*(instead of the original 8388608 bytes). All the above values are reduced by a factor of 64. The delay between the arrival of two pulses of the radar is set to 700 micro seconds.

The activity and latency plots for this simulation are shown in Figure 4.14 and Figure 4.15 respectively. As can be seen from Figure 4.14, the total time for processing the whole algorithm is 11910 micro seconds i.e. 11.9 ms. The utilization of the *range processors* is 41% and that of the *input processor* is 36%. As can be seen from Figure 4.14, 8 pulses are sent by the radar and the input processors sends the processed data to the four range processors on rotation. The parallelism among the range processors can be clearly seen.

The utilization for the *corner turn processor*(B11_cpu11) is 16%, the *image processor*(B15_cpu15) is 9% and that of the *output processor*(B17_cpu17) is 1%. In Figure 4.14, a large region of activity can be seen after the range processing. This is when the *corner turn processor* begins the bin processing, the image processor generating the image and the output processor recording the image data.

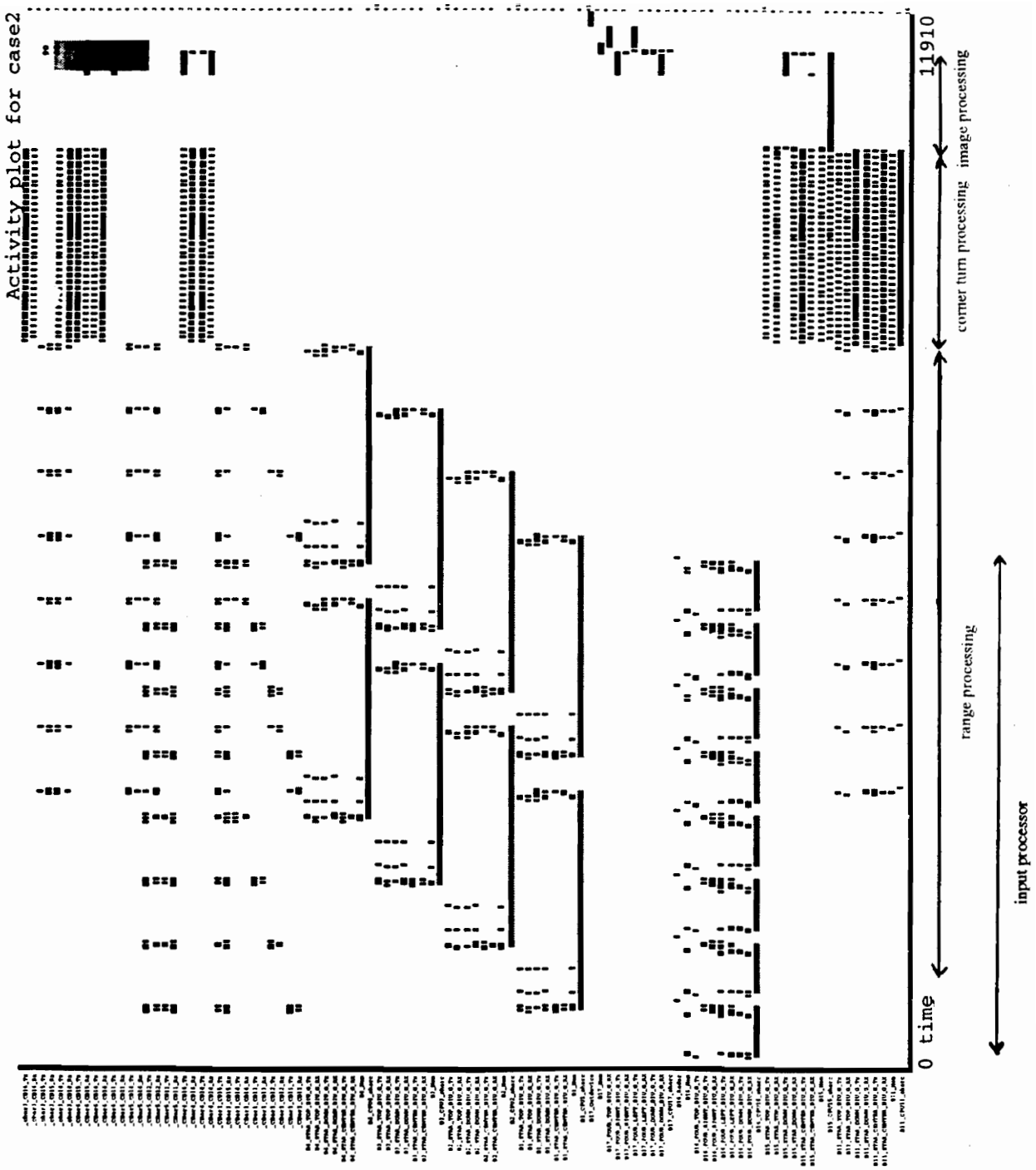


Figure 4.14 Activity Plot For Test 2

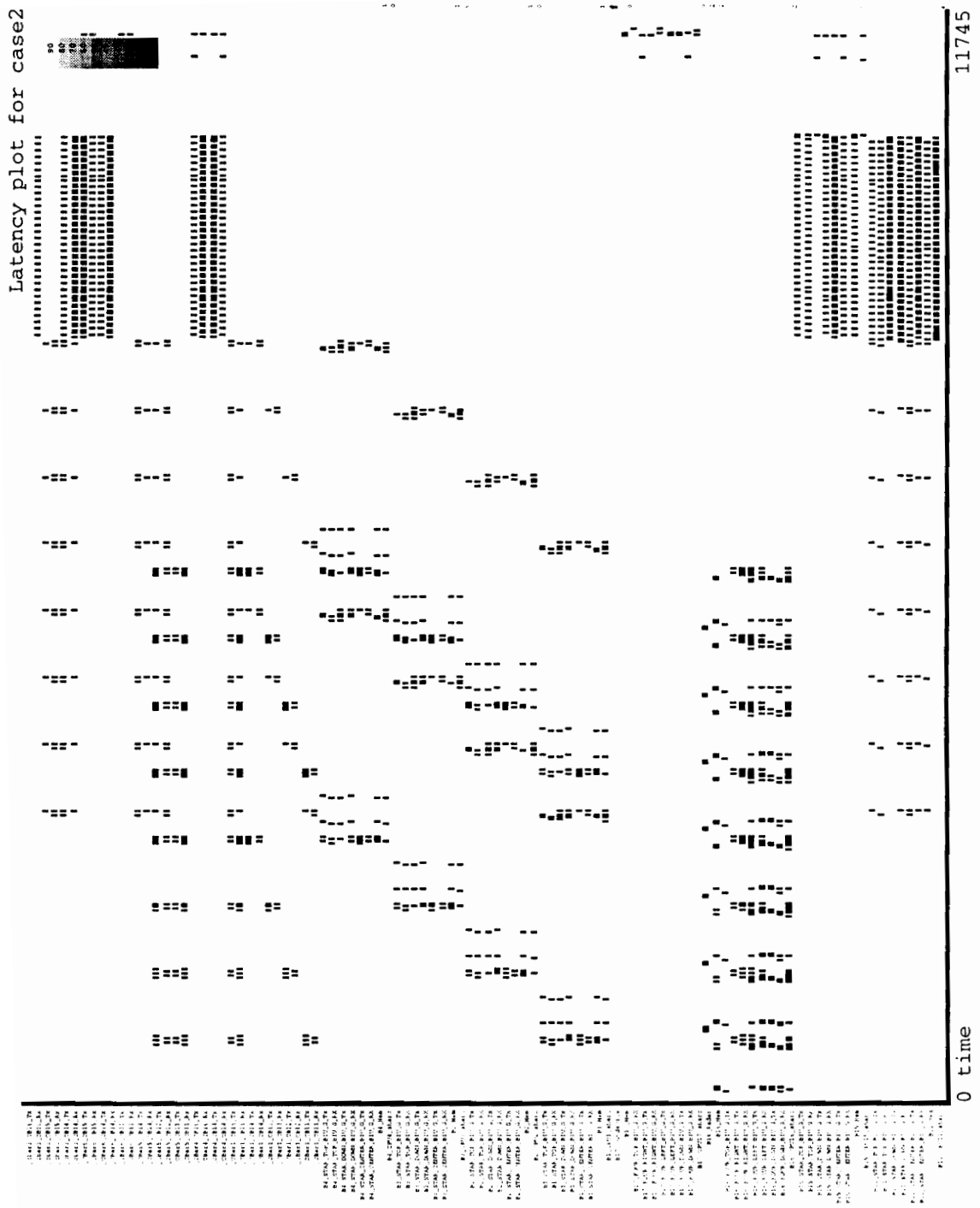


Figure 4.15 Latency Plot For Test 2

From the above result, the final result, while running the complete range processing algorithm with the correct number of 512 pulses and right number of loops within the algorithm can be approximately estimated. Since the whole algorithm was scaled down by a factor of 64, the final activity plot would be a replication of the one obtained in the above test. Hence the final processing would take approximately 64 * 11.9 ms which is 761.6 ms. The percentage utilization of the processors would more or less be the same as the result in Test 2- input processor- 36%, range processors- 41%, corner turn processor- 16%, image processor- 9%, output processor- 1%.

4.3.2.3 Simulation Statistics

The simulation statistics are tracked by the Vantage Simulator by setting the track statistic option of the simulator. The circuit statistics and the simulation statistics are tracked. The circuit statistics are tracked during the static analysis of the system(during set up). These statistics include the entity name, architecture name, configuration name, number of components portmapped in the structural model, number of signals, number of processes, drivers, etc.

The simulation statistics are the statistics tracked during run time. The important ones are explained below,

Current simulation time: The total time for which the system was simulated.

Elaboration CPU time : This is the total CPU time the simulator spent in setting up the system prior to simulation. This includes initialization of variables, processes, etc.

Simulation CPU time: The total CPU time the simulator spent in simulating the design after set up.

Transaction Count: The total number of signal transactions that took place during the simulation time.

These statistics are shown in Figure 4.16 for the Range processing Algorithm.

Circuit statistics

Entity Name : Arch24t_testbench
Architecture Name : testbench
Configuration Name : STP_24t
Number of Components : 1147
Number of Processes : 547
Number of Signals : 810020
Number of Drivers : 556560

Simulation statistics
Current simulation time : 1200 ms
Current simulation delta : 2416583
Elaboration CPU time (second) : 7.043333e+01
Simulation CPU time (second) : 7.022833e+02
Transaction Count : 21511348
Event Count : 1483959
Transacts / SimulationCpuSecond : 3.063058e+04
Events / SimulationCpuSecond : 2.113049e+03

Figure 4.16 Simulation statistics

As can be seen from the above statistics, the Range Processing algorithm uses 1147 components, 547 processes, 810020 signals and 556560 drivers. The CPU spent about 70 seconds preparing the system for simulation which includes initialization of variables and processes, etc. The CPU spent about 700 seconds in simulating the circuit after it was ready for simulation.

The Simulation Efficiency(E) is defined as follows[1]

$$E = \frac{\text{real logic time}}{\text{host CPU time}}$$

where real logic time is the actual time required to complete an activity sequence in a real logic circuit and host CPU time is the time required to simulate the same activity sequence using a logic simulator running on a host CPU[1]. In the present situation, the simulation efficiency is 1200 ms/ 700 sec which is about 0.2%(2000 to 1).

Another possible measure of efficiency is the number of events that a simulator can process per unit time[1]. The Vantage simulator could process about 2113 events/ SimulationCpuSecond.

4.4 The Multiswath Processing Algorithm

The Multiswath Processing Algorithm is a part of the Synthetic Aperture Radar(SAR) algorithm. The difference from the Range Processing Algorithm is not in the number of processors but in the more complex software the processors are executing. This algorithm is also mapped onto the seventeen processor architecture.

4.4.1 The Task Mapping

The simulation begins with the *radar* sending pulses and writing the data to the B16_mem by DMA. This memory then interrupts the *input processor,* B16_cpu16. The B16_cpu16 then begins its processing and transmits the data from each pulse to each of the *range processors* successively. The data is then processed by the *range processors* and sent to the *corner turn processor*, B11_cpu11. This processor waits for a whole frame of 512 pulses and does the *corner turn processing*. The *range processors* interrupt the *input processor* at the middle of the frame at the 256 th pulse of a frame of 512 pulses. The *input processor* processes the data and sends the *auxillary data* to the *corner turn processor* and the *image processor*, B15_cpu15.

When the B15_cpu15 receives the *auxillary data*, it is sent as a separate message to the *output processor,* B17_cpu17(in reality, it is concatenated on the front of the output image) and the B17_cpu17 writes this message to the *output device*.

When the B11_cpu11 receives both the *auxillary data* from the B16_cpu16 and the data from the *corner turn processing*, it begins the *range bin processing* by broadcasting the *subswath kernel* to all the nine *azimuth processors,* B5_cpu5, B6_cpu6, B7_cpu7, B8_cpu8, B9_cpu9, B10_cpu10, B12_cpu12, B13_cpu13 and B14_cpu14. Once the *subswath processing* is done by the *azimuth processors*, the result is sent to the

image processor, B15_cpu15 where the *image is assembled.* The B15_cpu15 sends out this *line of image* to the *output processor* which then stores it to the *output device.*
The specifications for this algorithm can be found in Appendix B.

4.4.2 Results

One cycle of simulation is defined as the time between the arrival of the first pulse and the time at which the result is written into the output device. For the Multiswath-Processing Algorithm, one simulation cycle consists of 512 radar pulses being sent to the four *range processors* on rotation. To simulate such an amount is estimated to take at least four days. Since this is not practical, the loops within the algorithm have been reduced.

In the final simulation, 8 radar pulses were sent(instead of the original 512). With reference to Appendix A, the loop of 2048 for Task 2 of the *corner turn processor* was reduced to 32. For the same processor, Task 4 has a loop of 16 which was reduced to 2 and the loop of 128 reduced to 18. Correspondingly, the loop in Task 2 of the *image processor* was set to 36(2 * 18 from above) and that of the *output processor* also to 36. A delay of 725 micro seconds was set between the arrival of two radar pulses so the first pulse is finished processing before the arrival of the next one.

The activity and latency plots for this simulation are shown in Figure 4.17 and Figure 4.18 respectively. From Figure 4.17, the first region of activity is the operation of the four *range processors* in parallel. The next region of activity is the corner turn processing and broadcasting the kernel to the *azimuth processors* by the *corner turn processor*. It can be further observed that all the nine *azimuth* processors are operating in parallel. The last region of activity is the assembling of the image by the *image processor* and writing it to the *output processor* which stores the image in the output device.

From Figure 4.17, it can be observed that the percentage utilization for the input processor is 17%, the range processors is 19%, azimuth processors is 27%, the corner turn processor is 15%, the image processor is 20% and the output processor is 2%. The total time for processing is 25292 micro seconds.

From the above result, the final result, while running the complete algorithm with the correct number of 512 pulses and right number of loops within the algorithm can be approximately estimated. The final processing would approximately take 1421 ms. This was calculated from the output trace file. For example, it took 8323.8 micro seconds for the range processing of 8 pulses, thus 512 pulses would take 532.87 ms. With the above data, the percentage utilization was calculated for each of the processors and it was observed that the final result would have utilization values similar to the above result. The final activity plot would be a replication of the one obtained in the above test.

As can be seen from the results, the utilization of the processors is very low. This shows that the architecture is over designed to run the above software application. As explained in the methodology in section 4.1, at this point, one should go back and change either the architecture or the partitioning of the software algorithm to obtain an optimum architecture for running these algorithms. It may further be noted that, the goal of the present research work is to demonstrate the methodology developed for construction of multi-processor architectures and not to optimize the architecture.

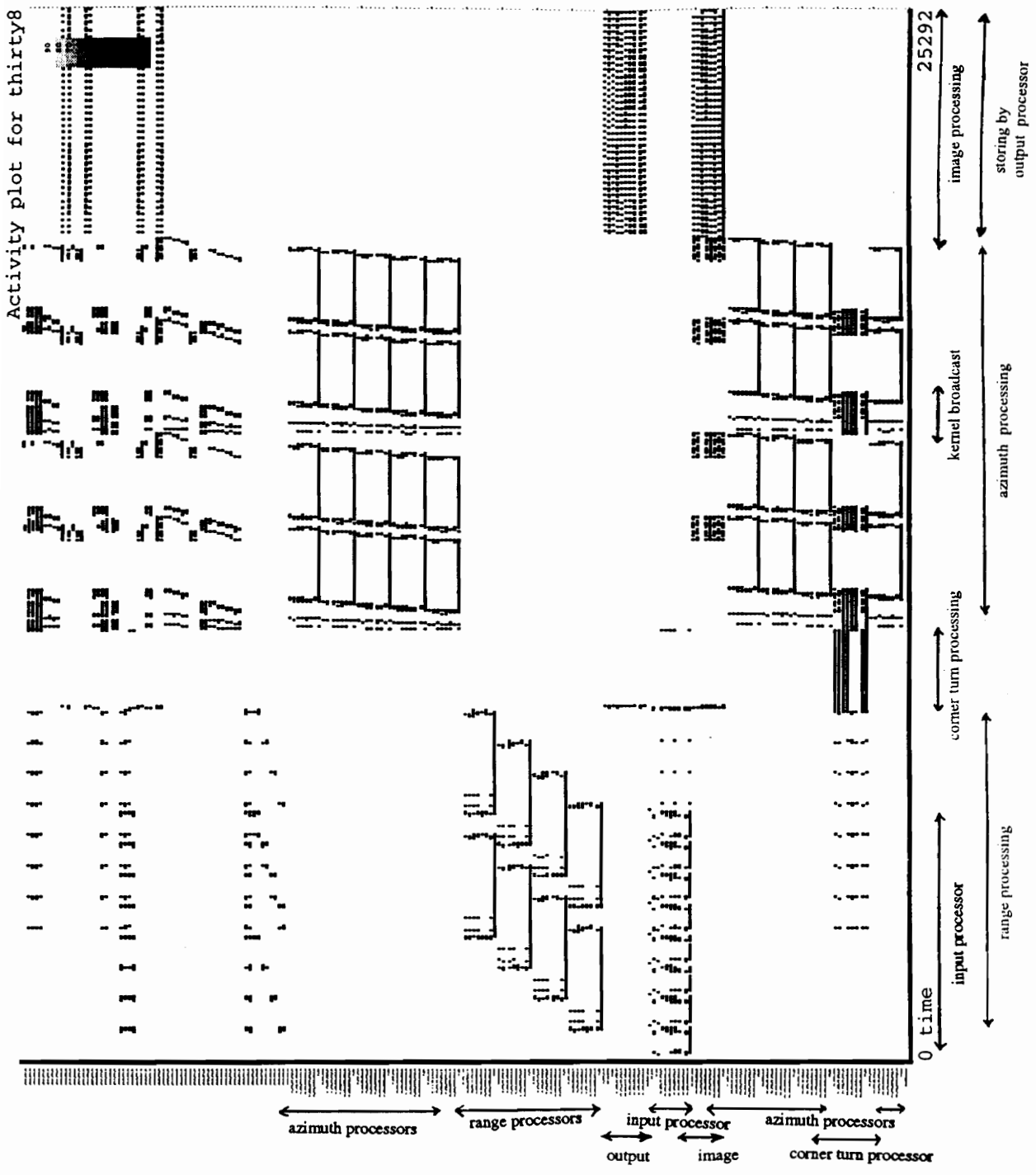


Figure 4.17 Activity Plot For The Multiswath Processing Algorithm

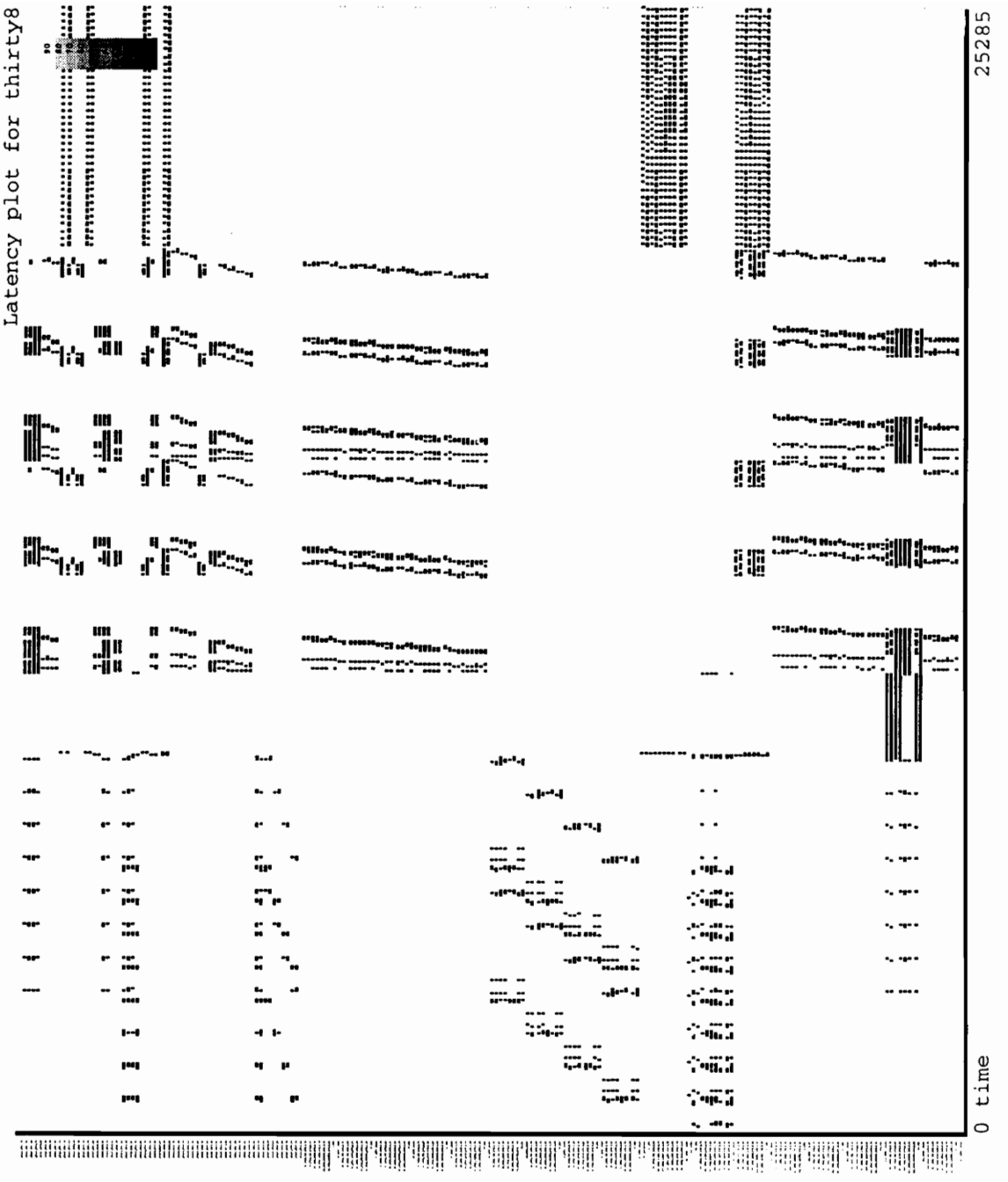


Figure 4.18 Latency Plot For The Multiswath processing Algorithm

Chapter 5. METHODOLOGY FOR MODEL VALIDATION

5.1 Model Validation

5.1.1 Methodology

It is very important to verify that the model actually describes the real system. To check the correctness of the results is a very difficult task especially when the software algorithm that is running on the hardware model is very big and complicated.

Two major techniques were adopted to validate the results of the range processing algorithm and the multiswath processing algorithm. The techniques are *decomposition* of the software algorithm and *scaling* of the algorithm. The model validation flowchart is shown in Figure 5.1. The following methodology was developed to verify the correctness of the simulation results.

1. If the software algorithm is very big and complicated, it is very difficult to verify the correctness of the results. Hence, the algorithm is *decomposed* into small portions.

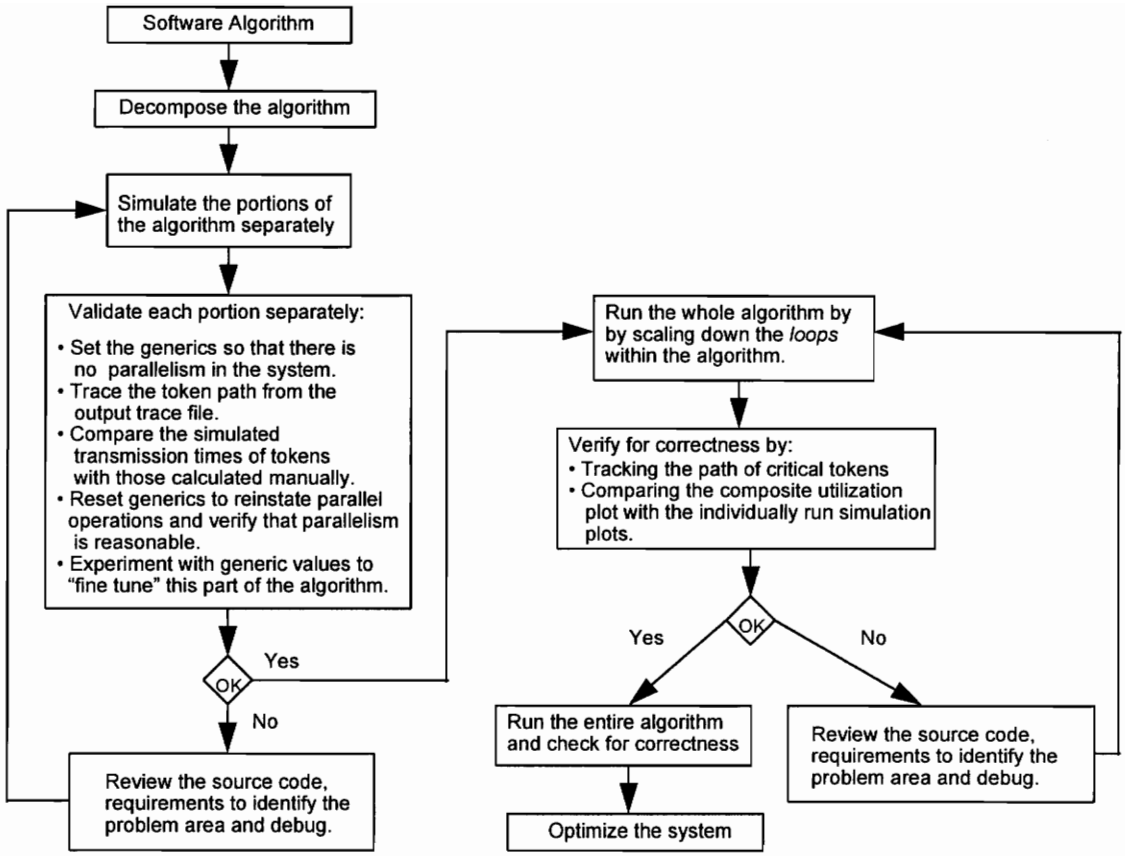


Figure 5.1 The Model Validation Flowchart

2A. Each of these portions are simulated separately and the results analyzed and checked for validity.

2B. The following methods were adopted to check for validity.

- The delay between the arrival of two pulses from the radar was set to a very large value in order to insure that there is no traffic congestion or loss of tokens due to bus conflicts. In this way the parallelism among the processors is removed and the tokens can be traced more easily. In other words, set the generics so that there is no parallelism in the system.

- Every token that is born is given a unique token ID during run time. The run time statistics are written into an output trace file called STD_OUTPUT. The path followed by every token can be traced in this file, from its birth till it reaches its destination with the help of this unique ID. One can also track the latency and busy time for all the transfer devices that a token passes through on its journey from its source to its destination. One can compare this against what is actually specified in the application software algorithm and make a valid conclusion whether the token has reached the correct destination in the expected amount of time.
- As described earlier in chapter 4, the total time for transmission of a token is the sum of the tx_time and latencies. So, the time for transmission can be manually calculated and checked against the results from the trace file.
- Reset the generics to reinstate parallel operations and verify that parallelism is reasonable.
- Experiment with different generic values to "fine tune" this part of the algorithm.

2C. If the results are not satisfactory, review the source code, the requirements and identify the problem area. Debug and repeat steps 2A and 2B.

3A. When each part of the algorithm was validated, the whole algorithm was run but the **loops** within the algorithm were *scaled* down.. For example, in the Multiswath Processing Algorithm, there were 128 range bins in a subswath (i.e. the processing was done 128 times within a loop), this was reduced to just 2 range bins for the initial test. This scaled version of the algorithm was run on the model and the results were validated.

3B. The following methods were adopted to check for validity.

- The critical tokens were traced because without these tokens reaching their destination, no further processing would have taken place.
- "ASSERT" statements were added wherever possible to display at run time the number of times a **loop** is being executed.
- the utilization plots were compared with the plots from the individual simulations.

- In case a few processors are doing the same kind of processing, then it is possible to use the tests as described in step 2 for one of the processors and check the utilization(activity plot) to see if all these processors show similar utilization. One can zoom on the activity plot for better comparison. This method was used to check a portion of the Range-Processing Algorithm in which all four *Range Processors* had similar processing steps.

3C. If the results are not satisfactory, review the source code, requirements and identify the problem area. Debug and repeat steps 3A and 3B.

4. Finally the whole algorithm should be run on the whole architecture and the results checked as described in step 3.

5. When the whole algorithm is verified to be running correctly, optimization for better throughput and better utilization of the components is done by varying the device parameters.

With the above approach, the SAR Range Processing Algorithm and the SAR Multiswath Processing Algorithm were validated.

5.1.2 Validation Of The SAR Range Processing Algorithm

Test 1: In this test, a portion of the range processing algorithm was run on the whole Raceway architecture. Only the processing for the *input processor* and the four *range processors* was run and tested.

All the processors in the architecture are running at 40 MHz. The memory access time for all the memory modules was set to .025 microseconds/word(4 bytes). The generics, **rx_latency_info** and **tx_latency_info** for all the transfer devices, *crossbar,*

biu_star and *biu_four* were set to 100 micro seconds. The transfer time for these devices was set for .007 micro seconds/word(4 bytes).

For the *radar*, the generic, **some_delay_info** was set to 25000 micro seconds. The *radar* would generate pulses with a delay of 25000 micro seconds between two pulses. This large value was given so that there is enough time for the first pulse to get processed before the arrival of the next pulse and hence there would be no traffic congestion or loss of *radar* tokens due to bus contention. This would also make the check easier. Just four *radar* pulses were sent by setting the generic, **pulses** to 4.

The activity plot and the latency plot for the above simulation results are shown in Figure 5.2 and Figure 5.3 respectively. As discussed in section 5.1, the tokens were tracked in the output trace file for one of the *range processors*. Since the processing steps for all the *range processors* are the same, one can logically conclude that the results are correct by observing similar utilization plots for the other *range processors*. Moreover, one can track only the critical tokens for the other processors. For example, after the processing of the four range processors, the results are written to B11_cpu11. Hence, for the remaining range processors, one can trace the path of the token to the B11_cpu11 and conclude that the results so far are correct.

From Figure 5.2, it can be observed that the *input processor* (B16_cpu16) is sitting idly after unpacking the first pulse from the *radar* and writing the result to the first *range processor*. The *input processor* receives the next pulse only after 25000 micro seconds. By that time, the first *range processor* is already finished its processing and is idle. When the next pulse is written to the second *range processor*, it is the only active *range processor*. Hence there is no overlap among the *range processors* which makes the token paths easier to trace.

This current model has no parallelism. By removing the parallelism, the correct number of tokens, the token paths, the transmission time for the token, the busy time and latencies of the components were more easily validated. After verifying the remaining part of the algorithm, parallelism will be restored to verify that bus bandwidths are not exceeded.

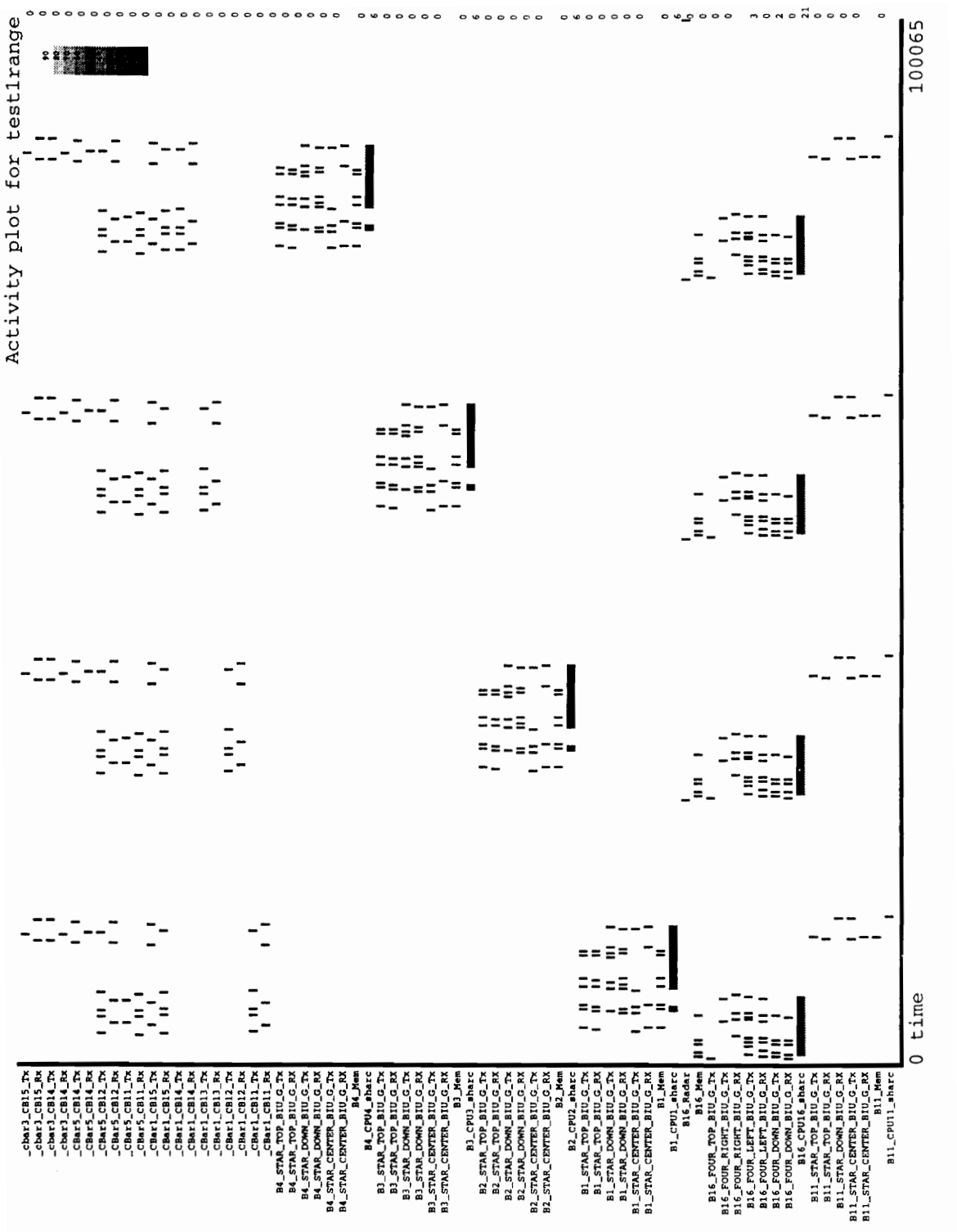


Figure 5.2 Activity Plot For Test 1

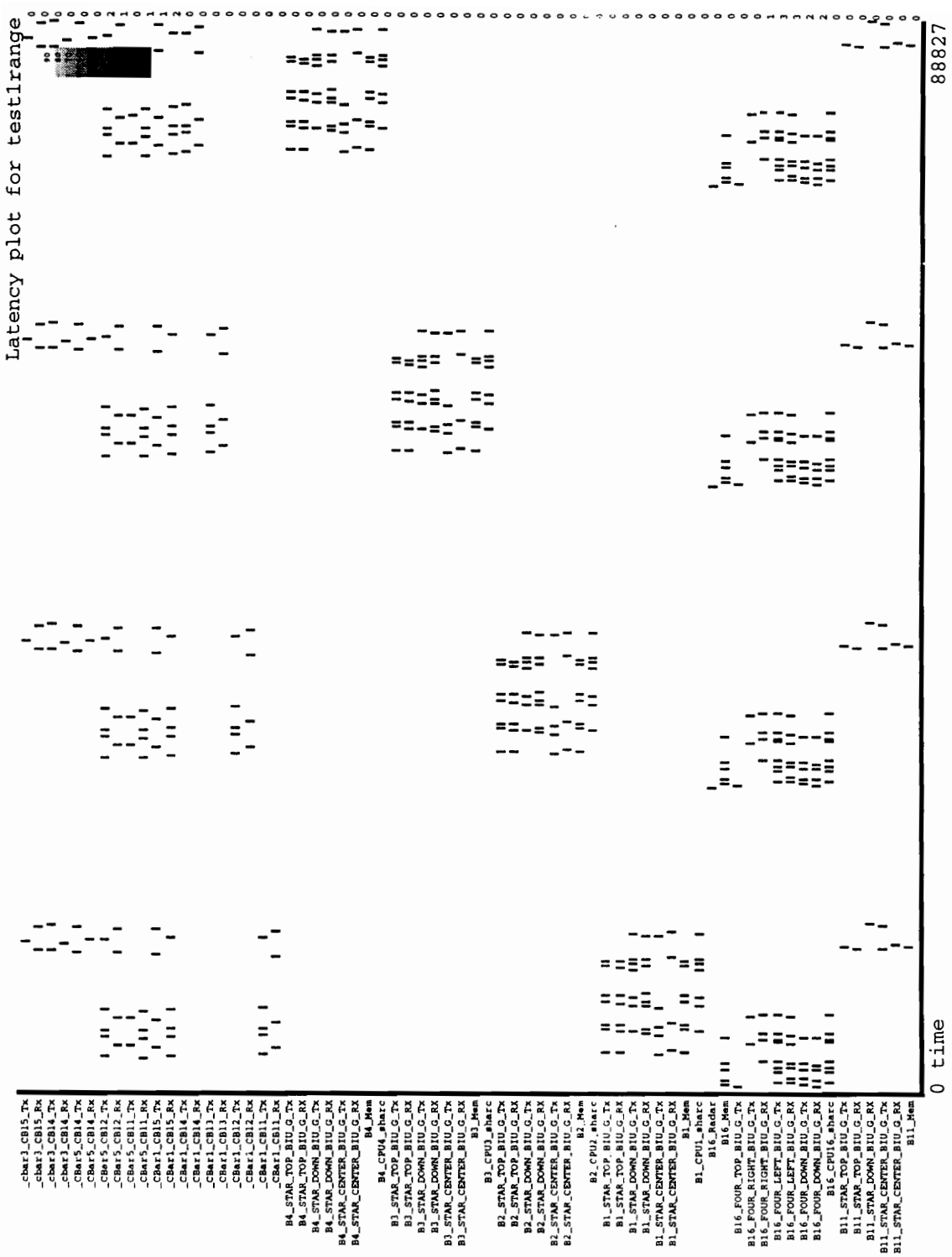


Figure 5.3 Latency Plot For Test 1

Test 2: In the remaining portion of the algorithm has the *corner turn processor* waits on the results from the *range processors* and then executes the *bin processing*. The results are then sent to the *image processor* for assembling of the image which is forwarded to the *output processor* for storing into the *output device*. This part was again simulated separately.

For this test, just a single pulse from the *radar* was sent. When the *input processor* received this pulse, it sent four interrupt tokens to the *corner turn processor*(equivalent to the results from the four *range processors*). In this way the corner turn processing, bin processing and storing part of the algorithm was separately simulated. This makes checking the results less complicated. All the other parameters are the same as in Test 1.

The activity plot and the latency plot for the above simulation are shown in Figure 5.4 and Figure 5.5 respectively. As mentioned in Section 5.1, the path of the tokens are traced from the output trace file. As there was a loop being executed by the *corner turn processor* in this portion of the algorithm(refer to specifications of this algorithm in Chapter 4), the tokens were tracked once through the loop and the number of times the loop was executed was checked by adding ASSERT statements to display messages at critical points in the loop.

As can be observed from Figure 5.4, the first small black region of activity is the *input processor*(B16_cpu16) sending four interrupts to the *corner turn processor(*B11_cpu11). The next region of activity is that of the *corner turn processor*(B11_cpu11) doing the bin processing at the end of which the result is written into the *image processor(*B15_cpu15). Thus the next region of activity seen in Figure 5.4 is that of the *image processor* (B15_cpu15). This image processor is writing the assembled image(8388608 bytes) into the output processor(B17_cpu17). It can be

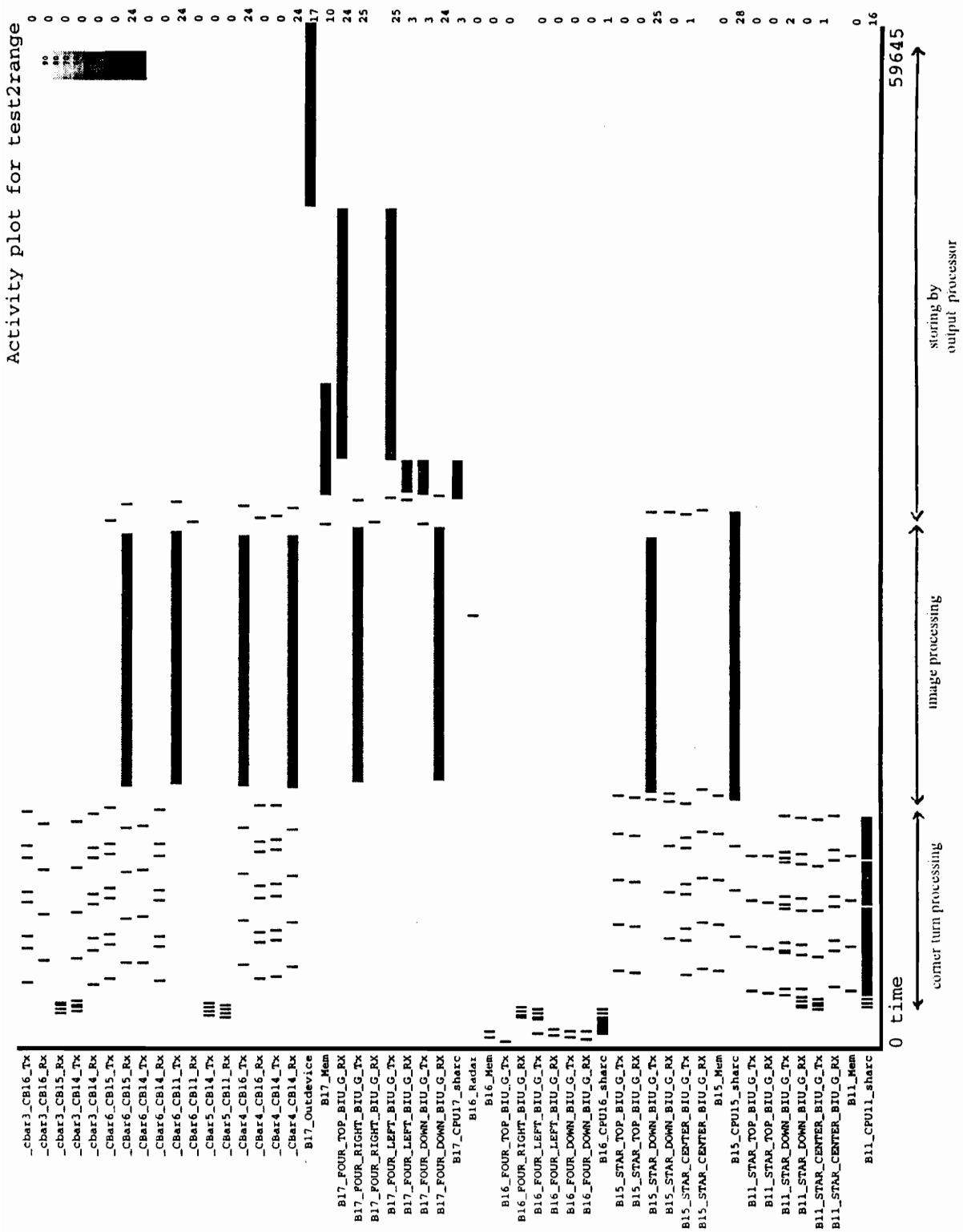


Figure 5.4 Activity Plot For Test 2

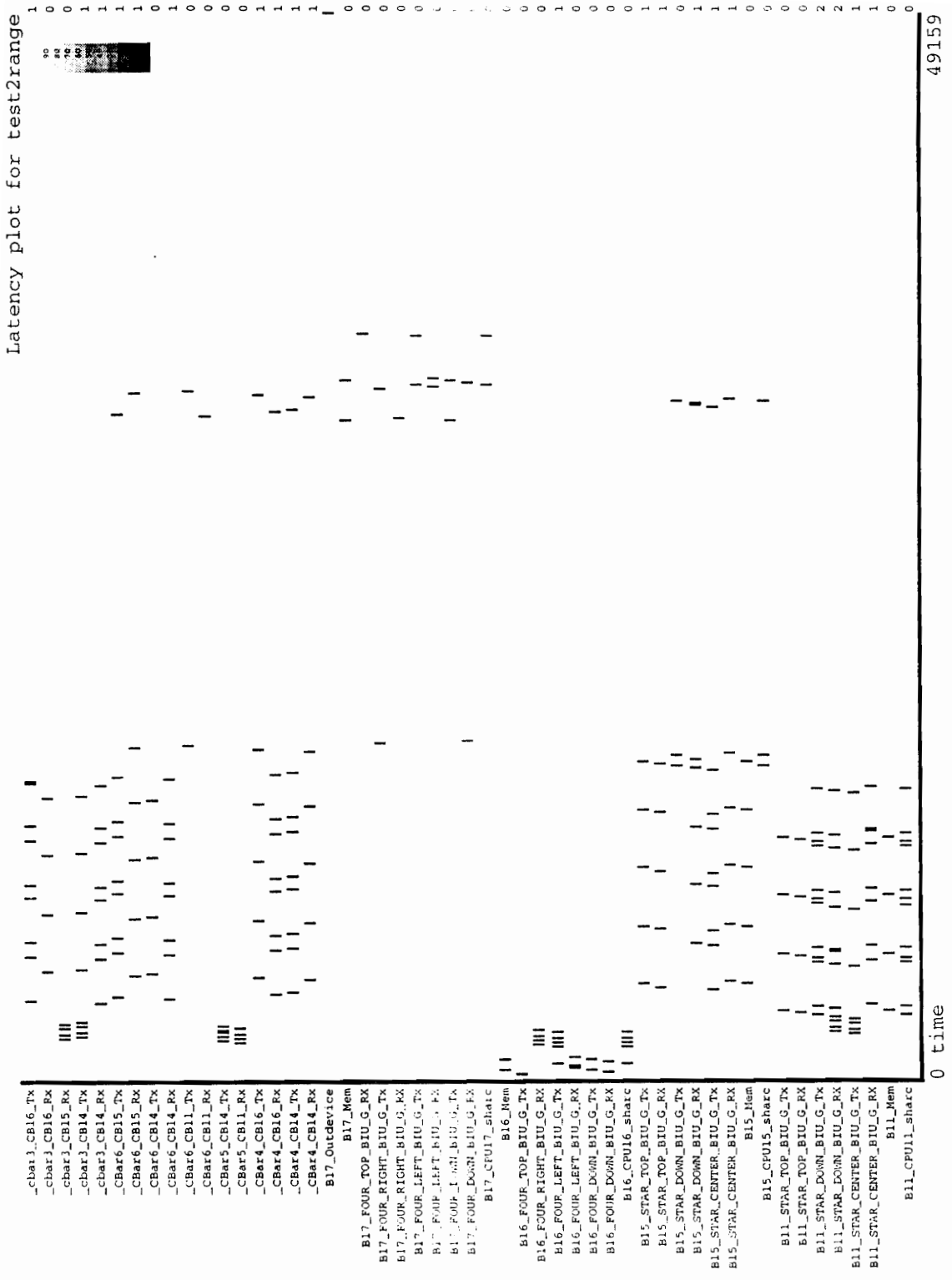


Figure 5.5 Latency Plot For Test 2

observed, as expected that all the bus paths being used by the image processor to write data to the output processor are busy. Finally, the result is written into the output device. This verifies that this portion of the simulation is valid.

Test 3: Once the different pieces of the algorithm are separately validated, the full algorithm is run on the whole hardware model. But the loops within the algorithm are scaled down to reduce simulation time. This decreases the number of tokens in the simulation and hence validation will be easier. The 512 pulses from the *radar* has been reduced to 4. Also the loop of 2048 for the bin processing by the *corner turn processor*(B11_cpu11) has been reduced to 4.

In Test 1, the delay between arrival of two pulses was deliberately set to be large in order to facilitate tracing tokens. At this time, different experiments were performed by gradually reducing the delay between two pulses until the system became saturated. It was observed that one could reduce the generic, **some_delay_info** of the radar to a minimum possible value of 11000 micro seconds. If the delay was further reduced, the tokens were lost due to bus contention. All the other generics are the same as in Test 1.

Thus Test 3 has **some_delay_info** set to 11000 micro sec and **pulses** equal to 4. The activity plot and the latency plot for the above simulation are shown in Figure 5.6 and Figure 5.7 respectively. These plots were compared with the ones on which only portions of the algorithm were run. Also the critical tokens are traced, because if these tokens do not reach the right place, further execution of the algorithm would not be possible. As can be observed from Figure 5.6, it is somewhat of a merge of the activity plots of the earlier two tests. Thus, the whole algorithm is validated to be correct.

This current model has no parallelism among the *range processors*. Yet, the

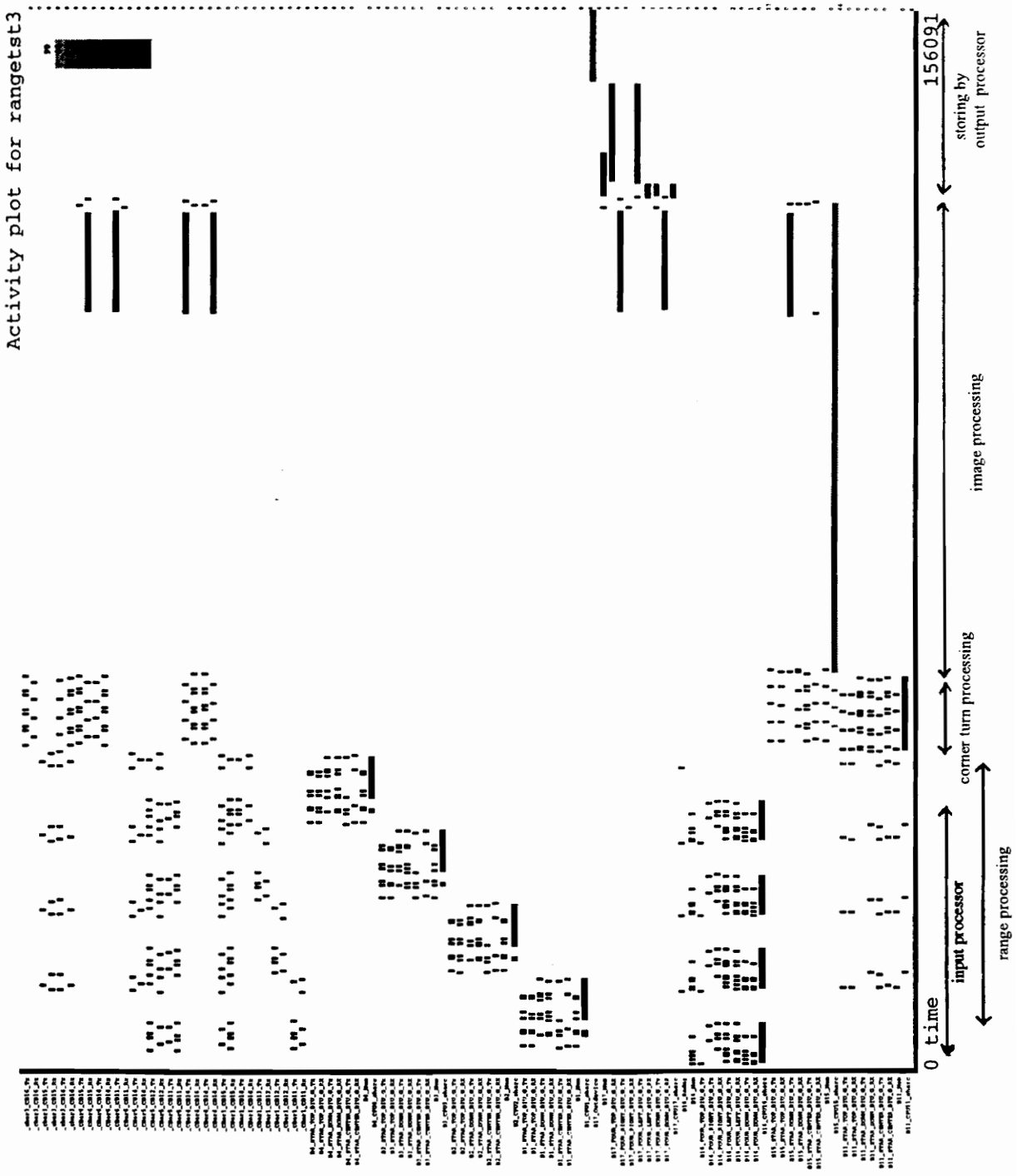


Figure 5.6 Activity Plot For Test 3

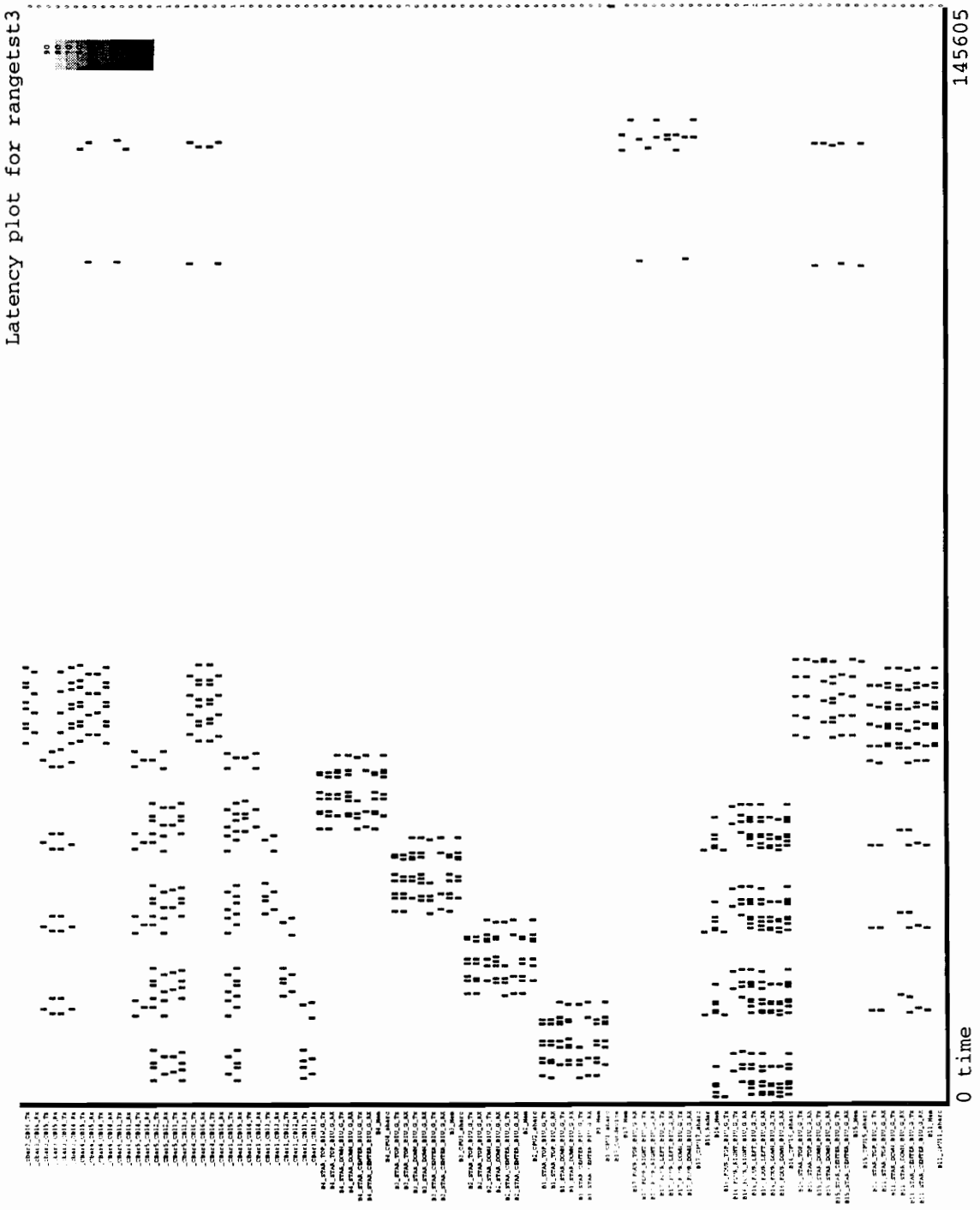


Figure 5.7 Latency Plot For Test 3

whole algorithm is validated to be running correctly. Further optimization by way of changing the device parameters has to be made for the *range processors* to run in parallel.

Test 4: This test was conducted to optimize the system. The generics, **rx_latency_info** and t**x_latency_info** were set to zero for all the transfer devices. For this test the memory access time is .025 micro sec per word and transfer time for the transfer devices is .007 micro sec per word. In this test, the delay between arrival of two pulses could be set as low as 700 micro seconds without there being any problems of congestion or loss of tokens.

In the earlier Test 3, the *input processor* was held busy for a very long time waiting for an ack token from the recipient before executing the next step. This ack token traveled through the transfer devices which had large latencies of 100 micro sec. Thus, the delay between pulses could not be reduced below 11000 micro seconds as the *input processor* itself took a long time to finish processing.

The activity plot and the latency plot for the above simulation are shown in Figure 5.8 and Figure 5.9 respectively. From Figure 5.8, it can be observed that when the latencies were set to zero, the time for processing for the *input processor* has dramatically reduced. This resulted in the four *range processors* operating in parallel. After the range and the bin processing, the long gray region of activity(refer to Figure 5.8) is that of the image processor(B15_cpu15) assembling the image which takes 2097152 clock cycles to write 8388608 bytes to the output processor(B17_cpu17). The next black region of activity is that of the output processor storing the data into the output device.

By comparing the results of Test 3 and Test 4, it can be observed how the generics affect the performance of the system. The application running during both tests is exactly the same. For the same software algorithm, while it takes 156091 micro sec for Test 3, it

only takes 99920 micro sec in Test 4 for the completion of the task. Test 4 is faster by almost 56171 micro seconds. The main reason being that the latencies for the transfer devices in Test 3 was set to 100 micro sec while it was zero for Test 4. Correspondingly, **some_delay_info** could be reduced to 700 us in Test 4 compared to 11000 us for Test 3.

Figure 5.8 Activity Plot For Test 4

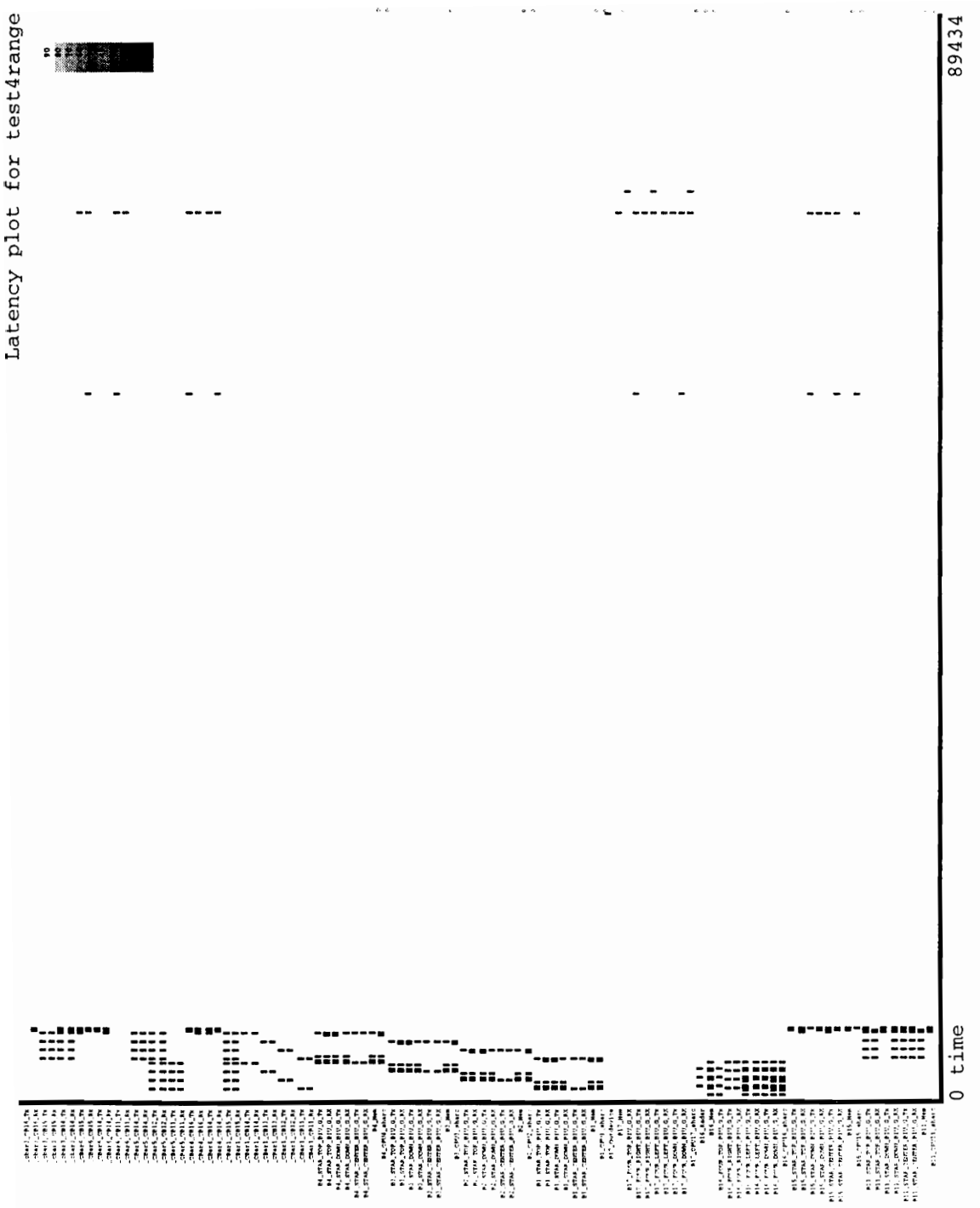


Figure 5.9 Latency Plot For Test 4

5.1.3 Validation Of The SAR Multiswath Processing Algorithm

The working of this algorithm has been explained in Section 4.3. The first part of this algorithm is similar to the SAR Range Processing Algorithm which has already been verified.

Test 1: The most complicated portion of the algorithm is the Azimuth processing. The *corner turn processor* receives processed data from the *range processors* and *auxiliary data* from the *input processor*. It then broadcasts the *subswath kernel pointer* to all the *azimuth processors*. The *azimuth processors* process the data and send the *range bins* to the *image processor* for *assembling* of the image.

In this test, the *corner turn processor* receives data directly from the *input processor* and begins the subswath processing(instead of from the *radar* to the *range processors* and then to the *corner turn processor*). Then the *azimuth processors* write the result into the *image processor.* Only this portion of the algorithm was run separately and checked for validity. In this way one can easily walk through the trace file and track each and every token to check if it is reaching the right destination in the right amount of time. The debugging of the program also becomes easier should there be any errors.

In this test, all the processors in the architecture are running at 40 MHz. The memory access time for all the memory modules was set to .025 microseconds/word(4 bytes). The generics, **rx_latency_info** and **tx_latency_info** for all the transfer devices, *crossbar, biu_star* and *biu_four* were set to zero micro seconds. The transfer time for these devices was set for .007 micro seconds/word(4 bytes). The **some_delay_info** for the *radar* is set for 725 micro seconds.

The activity plot and the latency plot for the above simulation are shown in Figure 5.10 and Figure 5.11 respectively. As can be clearly observed from Figure 5.10, the first

region of activity is that of the input processor unpacking data from the radar and sending interrupt to the *corner turn processor*. Once the *corner turn processor*(B11_cpu11) receives the interrupt from the *input processor*, there is a patch of activity around this processor as it is executing the corner turn processing, broadcasting subswath kernel to the *azimuth processors* and sending range bins to the *azimuth processors*. The nine long regions of activity are those of the *azimuth processors*. It can be further observed that these nine azimuth processors are running in parallel.

As described earlier, all tokens were traced for one of the *azimuth processors* but only the critical tokens were traced for the others since all *azimuth processors* have similar processing steps. By observing that the utilization plots of all azimuth processors were similar to the one that was traced exhaustively, that all critical tokens reached their final destination, and that the remaining part of the algorithm started on schedule, we conclude that this part of the algorithm is validated.

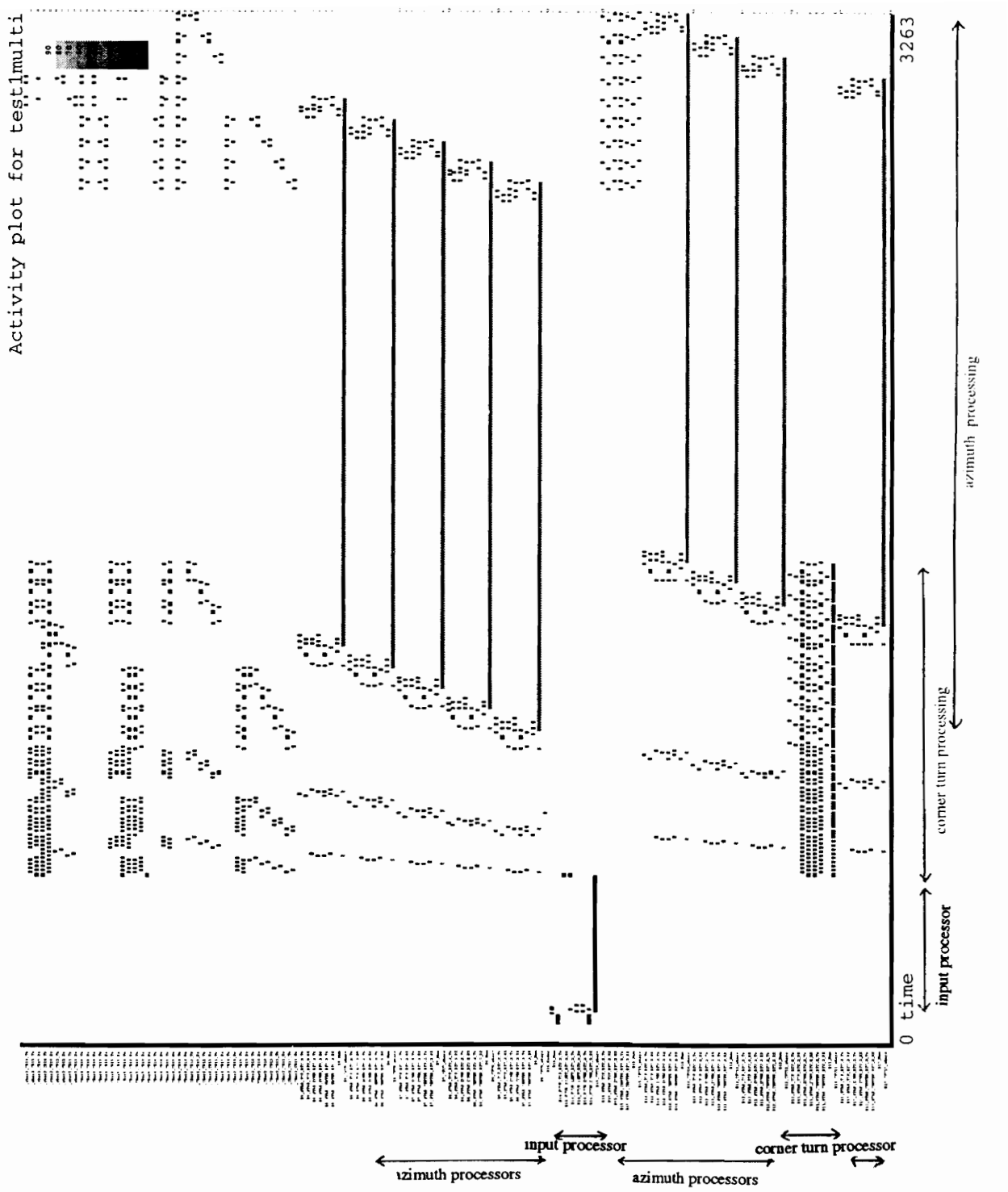


Figure 5.10 Activity Plot For Test 1

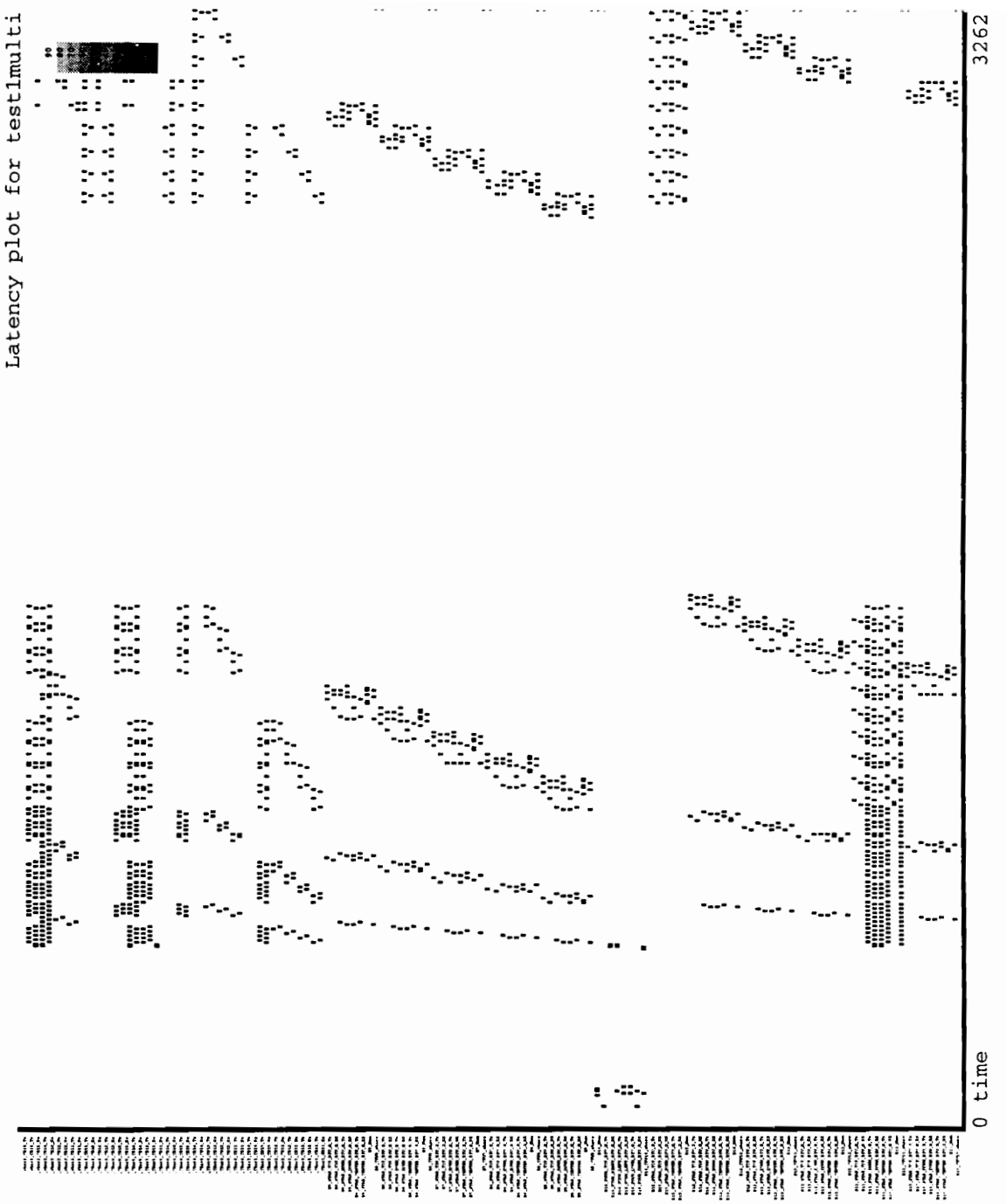


Figure 5.11 Latency Plot For Test 1

Test 2: Once the range and azimuth processing portions of the algorithm were validated separately, these two portions of the algorithm were run together(the processing of the *image processor* is not yet included). All the generics are the same as in Test 1.

The activity plot and the latency plot for the above simulation are shown in Figure 5.12 and Figure 5.13 respectively. As can be seen clearly from the activity plot in Figure 5.12, the first region of activity is the operation of the four *range processors* in parallel. The next region of activity is the corner turn processor executing the corner turn processing, broadcasting the subswath kernel and sending range bins to the azimuth processors. The next region is that of the nine *azimuth processors* running in parallel.

This composite result was checked against the individual results as part of the validation. Also, the critical tokens were traced to verify the correctness. Hence this portion of the algorithm has been validated.

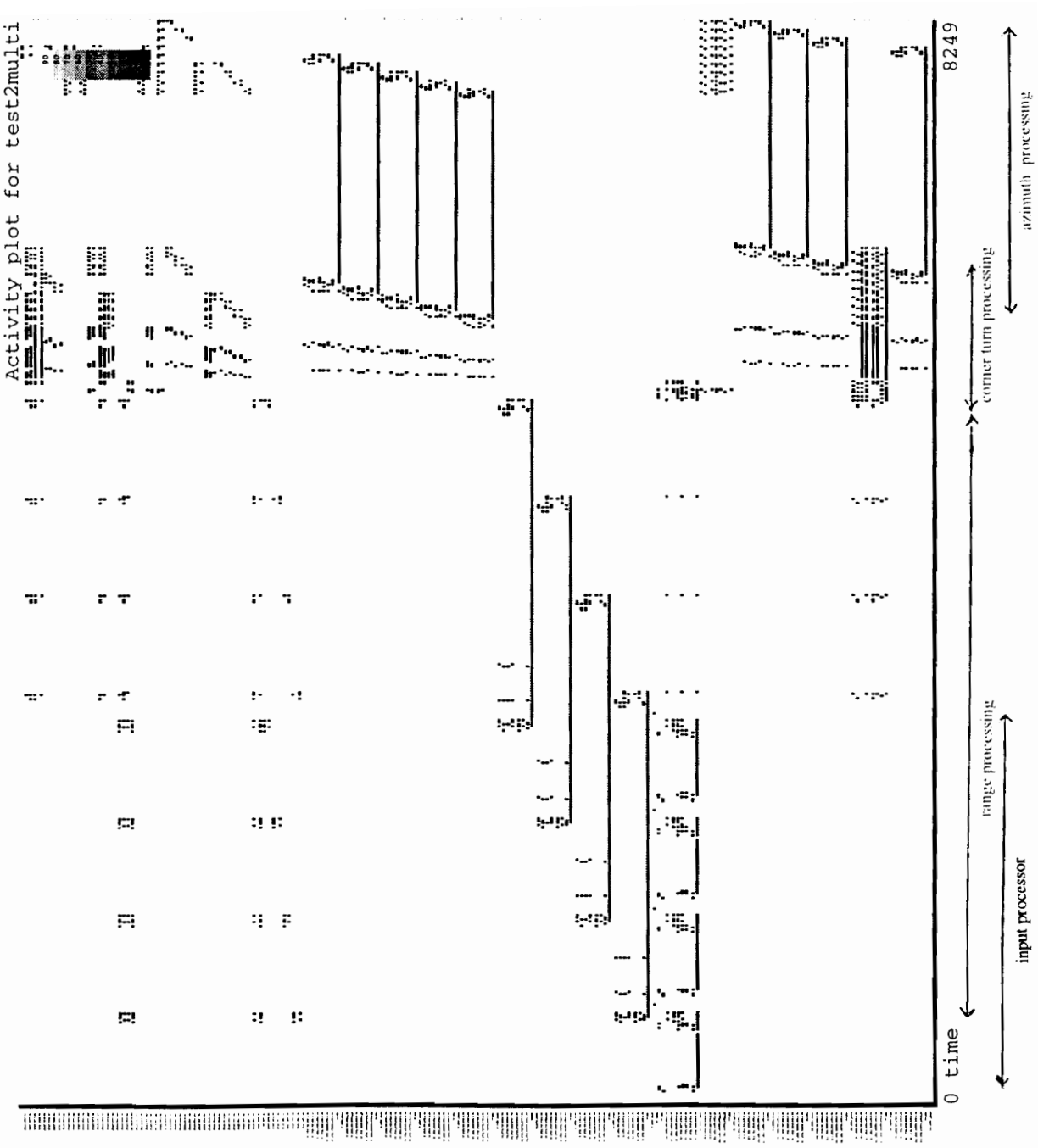


Figure 5.12 Activity Plot For Test 2

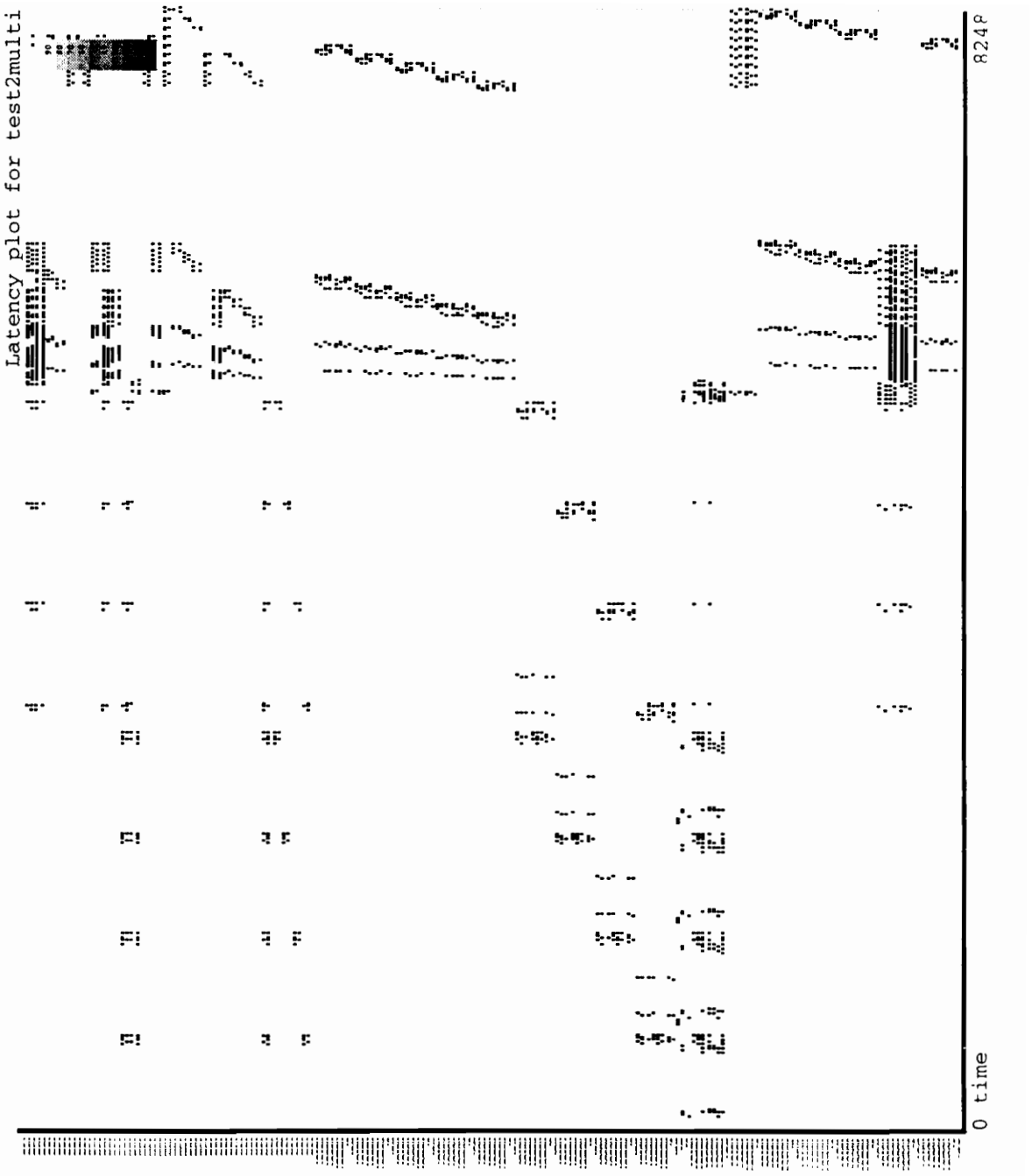


Figure 5.13 Latency Plot For Test 2

Test 3: In this test, the whole algorithm was run on the whole architecture model. However, the loops within the algorithm were reduced to verify if the logic still worked right when everything was put together since the number of loops within an algorithm complicates the verification. For example, the *corner turn processor* waits on 512 processed pulses from the range processors but this was reduced to 4. Also this processor executes a loop of 16 to process all subswaths this was cut down to just 1. While it also processes 128 range bins within a subswath, the loop was reduced to 9. In this way, one could walk through the whole algorithm once and verify the operation before running with the exact number of loops.

The activity plot and the latency plot for the above simulation are shown in Figure 5.14 and Figure 5.15 respectively. As can be seen clearly from the activity plot in Figure 5.14, the four *range processors* are running in parallel and also the nine *azimuth processors* are running in parallel. It can also be observed that the results from the *azimuth processors* are being sent to the *image processor* which sends the image to the *output processor* for storing. Refer to Chapter 4 for the final result.

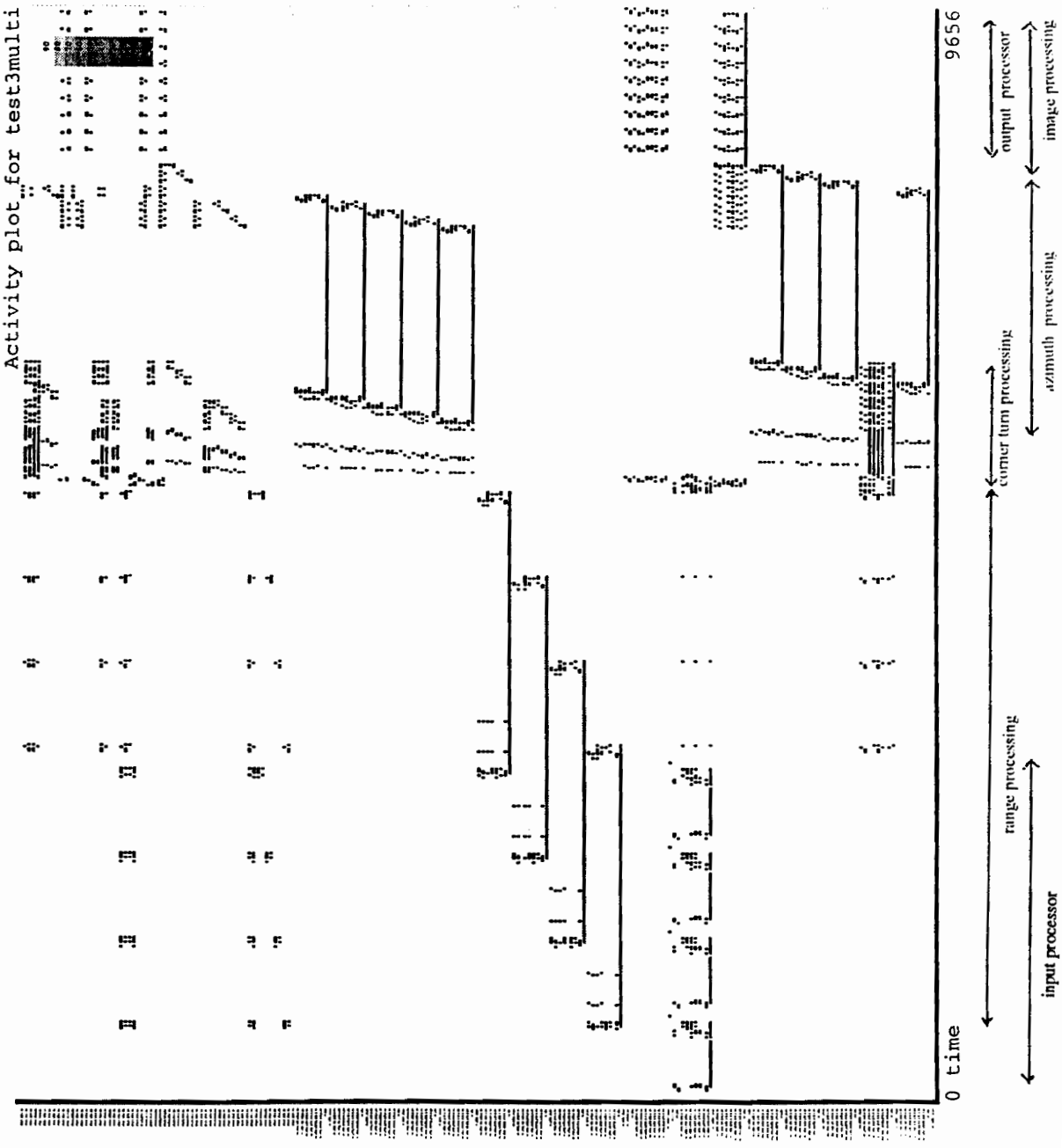


Figure 5.14 Activity Plot For Test 3

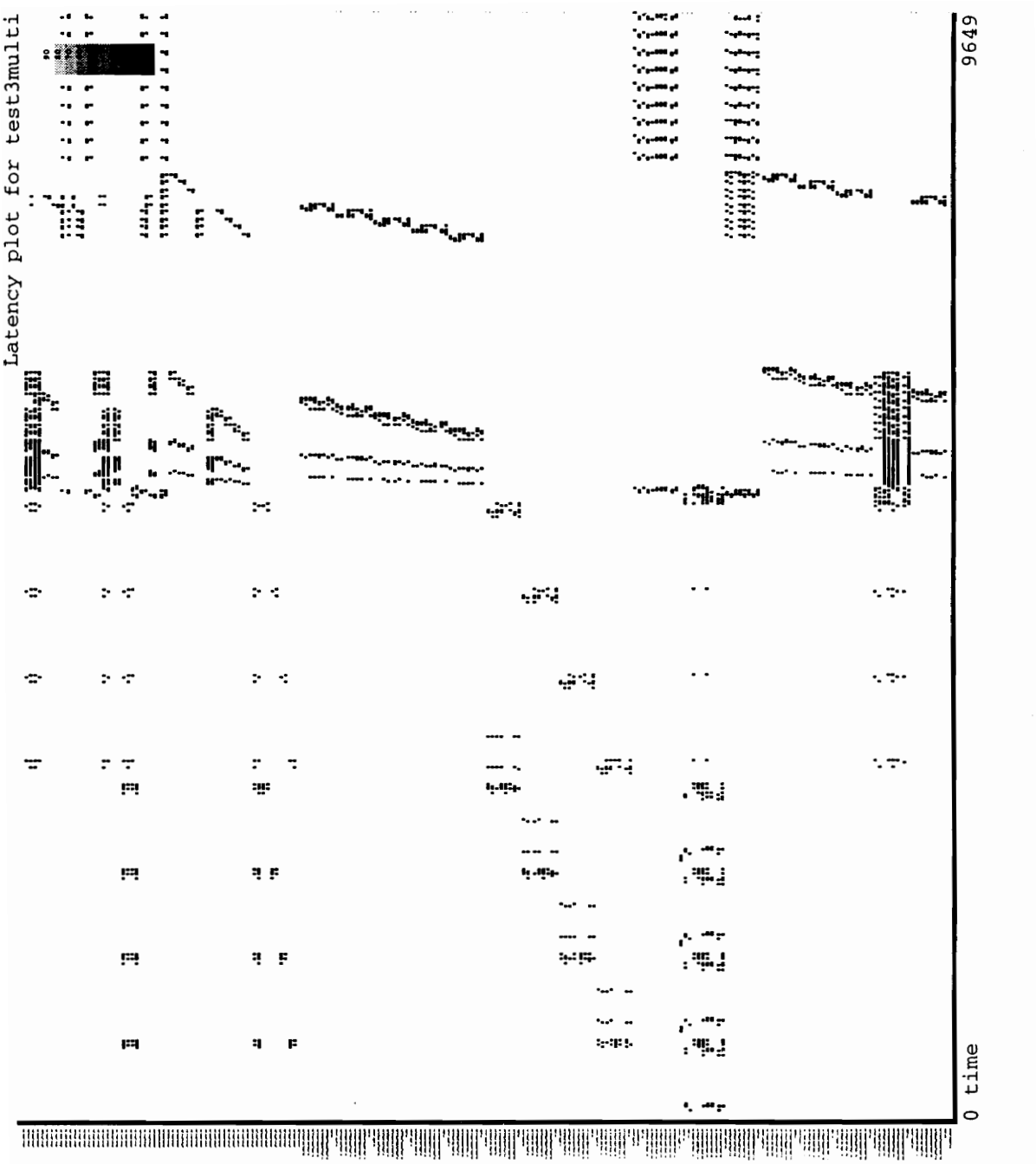


Figure 5.15 Latency Plot For Test 3

5.2 Methodology for Model Debugging

5.2.1 Methodology

Model debugging is a labor intensive and time consuming task. The following methodology helps to reduce the time required for debugging. Figure 5.18 shows the Model Debugging flowchart.

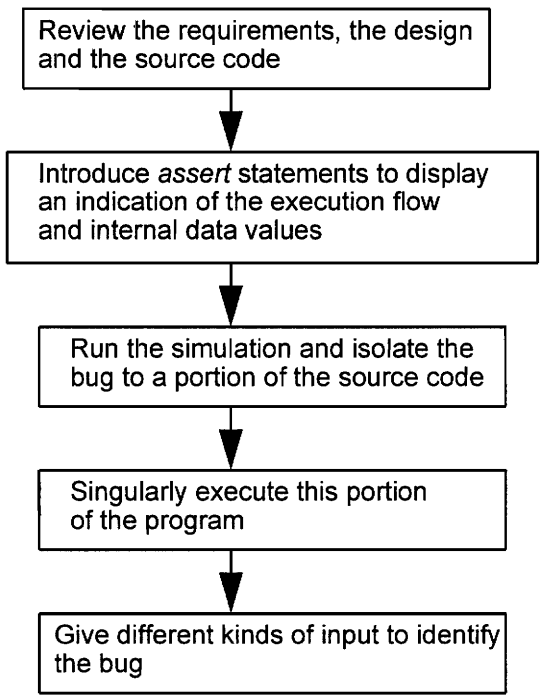


Figure 5.18 The Model Debugging Flowchart

- Preliminary investigations by way of review of requirements, review of the design and the source code is the first step towards debugging.

- By manipulating the source code, one can target important details which otherwise do not surface. This means adding statements that cause it to display an indication of the execution flow and internal data values. In VHDL this is done by introducing

"ASSERT" statements in the source code. This is a valuable tactic when chasing an elusive bug. In this manner, one is able to isolate the bug to a particular area of the source code.

- Once the particular portion of the source code containing the bug is isolated, it is a good idea to separately execute this portion of the program. The primary advantage of this is that the programmer has a great control over the scope of the information obtained.

- Simulate the problem portion of the program. Performing run-time observations is one area of debugging where one is most likely to find a root cause. Different inputs at run-time helps to unravel the program's flaws and identify the bug.

5.2.2 Illustration With Examples

1. A bug can remain dormant for years. The following example explains this fact.

When a token is sent from a source to the destination through multiple crossbars, say CB1 and CB2, the **route** field of the token(which is the actual path the token has to follow) is given as "CB1.CB2.destination". Once the token enters the crossbar CB1, "CB1" is stripped off from the **route** of the token and the token continues its journey with a new **route** of "CB2.destination" and this goes on until it reaches its destination. The code for stripping off had a bug which laid dormant, the reason being that multiple crossbars were never used in any of the earlier architectures.

To isolate this bug, "ASSERT" statements were introduced to display the path taken by a token throughout its journey through the bus interface units and the crossbars. In this way the bug could be isolated to the crossbar and a static review of the source code

of the crossbar solved the problem. It might take a while before one could isolate the bug but never make the mistake of assuming that the earlier code is always correct.

2. The component BIU_FOUR is a structural model of interconnection of four independent Bus Interface Units(BIU). The BIUs each have a local and a global side, the local side being connected to devices like memory, radar, etc. and all the global sides connected to a common bus. There are generics namely, **local_id** and **global_id** to be set for each of the four BIUs in the component BIU_FOUR.

The **local_id** screens traffic coming from the local port onto the global bus. So only tokens which have a destination field matching the local id will be accepted by the local side of the BIU and put on the global port. The **local_id** should normally be "?*" allowing all tokens to be put on the global bus.

The **global_id** screens traffic coming from the global bus onto the local bus so only the tokens which have a destination field matching the **global_id** will be accepted/picked-up by that BIU.

In one instance, the local side of the three BIUs were connected to the RADAR, B16_CPU16 and B16_MEM. Hence the **global_id** for these three BIUs were given as the above names so that the global bus would pick up any tokens that had their destination fields the same as these names. The fourth port of BIU_FOUR was connected to crossbar CBAR5. Hence, the **global_id** for this fourth BIU was given as CBAR5. But no tokens were passing through this crossbar.

To find this bug, a series of experiments began by writing programs to uncover where the problem area was. When the RADAR was directly connected to the port of the crossbar, the tokens passed across the crossbar without any problems. When BIU_FOUR

was introduced back into the system, the token did not reach its destination. Hence, the error was isolated to the BIU_FOUR module. After proper understanding of the generics, it was determined that the **global_id** for the BIU connected to the CBAR5 had to be "?*" allowing all tokens to be picked up by the crossbar.

3. The processor sends a token to a destination by making a call to the HWEXECUTE function as follows:
HWEXECUTE(me, send, receive, msg, hw_op, boolean, num_instr, comment).

In the above function, the generic **boolean** is set to TRUE if the processor is waiting for an acknowledgment from the destination before proceeding to its next execution, else FALSE. It is also important to understand whether the destination device has a provision to send an acknowledgment when it receives a token. This was one problem faced when simulating the seventeen processor architecture. The component outdevice(Ex. Hard Disk, Display) did not have a provision in its source code to send an acknowledgment. So when **boolean** was set to TRUE(it becomes a habit because, the processors usually write to memory modules and memory has the capability to send an ack.) the processor kept waiting for a reply from the outdevice. One way to debug this kind of error is static debugging by reviewing the source code of all the components involved in this process. It is very important to know what each parameter in a function call stands for. Lack of proper documentation should not be an excuse for working on something which is not properly understood.

4. As mentioned in **1,** when a processor reads/writes to a remote destination, the "route" to the destination is specified. Since the "route" gets completely stripped off by the time the token reaches its destination, there is no way for the destination to know the route to send an acknowledgment to the source module. The generic for **boolean**(as explained in **3**) was set to TRUE, and hence the source was waiting indefinitely for an

acknowledgment. It looks very obvious where the problem is after the cause is known. But, to actually search for the cause required thorough review of the source code of all the components through which the token passed. Once it was clear that the destination module did not have a return route for an acknowledgment, one field of the token, **parm1_real** which had not been used so far, was used to communicate the return path of the token. Since the library is very complicated, to actually know every intricate detail of every possible component is very difficult. Moreover, since the simulation is done at the highest level of the hierarchy, very minute details about a component do not easily surface. Hence once the bug is isolated to a particular component, a thorough static review of that portion of the code has to be made.

5. While mapping the generics for software application in the Processor-c.vhdl file, it is important to know how the **task_data_record** is mapped. It is a record containing the following fields, (priority, period, deadline, kind, port_id).

This contains scheduling information about the simulated software tasks. Priority = 0 is allowed only for tasks that do not do anything that causes simulation time to advance (i.e., they can schedule other tasks to run or send signals but they may not perform an execute command). Period = 0 means that the task is not periodic, it is driven by the occurrence of an explicit event. Deadline time'HIGH means that the task has no deadline for its completion. Priority 0 tasks should have a deadline of 0 ns.

The problem discussed below was one of the very difficult errors encountered. The tasks on the seventeen processor model(for both the algorithms) were driven by events; in other words, a processor would start its processing when it received an event/interrupt from another processor or from another task on the same processor. Not realizing the consequences of giving an arbitrary value for the generic **period**, created a seemingly unfathomable bug. This arbitrary value for **period** was inherited from an

earlier algorithm but the results were erroneous for the current one. Since the logic was seemingly right(tested for one radar pulse and it worked but went haywire after a few pulses), there was obviously some mistake setting the generics. To figure out which generic exactly caused this problem required thorough review of the processor library.

6. When the seventeen processor model was being simulated initially, there was always this kind of error,

**Runtime Error: Value 0 is outside the range defined by 1 TO 6 during indexing an array * Occurred on line 489 of architecture SYSTEM of entity CROSSBLOCK while executing instance /TB/CBAR5/CBL1.

where TB is the testbench, Cbar5 is one of the crossbars in the model, and CBL1 is one of the ports of that crossbar.

The above message means that the token that entered CBAR5 has a **route** "CBAR5.destination" but that destination component is not connected to any of the ports of CBAR5. In other words, the token does not know where to go and hence resulted in an error. But, in reality, that particular component *was* connected to CBAR5. To make sure that the token was assigned the right **route** during simulation, "ASSERT" statements were introduced to display the token **route** every time it entered a Bus Interface Unit or a Crossbar. It was observed that the routes were correct, even then the problem persisted. The next debugging technique that was followed was to partition the software into small sub-programs and also reduce the size of the 17 processor model to just 6 PEs connected to a single crossbar. It was clear that there was nothing wrong in either the application software or the hardware model because all the smaller programs ran perfectly well. Then, allowing the same small programs to run, but increasing the size of the hardware to two crossbars with about 11 components connected to their ports, the above "Runtime Error" returned. This was the first clue that the problem might be related to some system

problem outside the program itself. When the swap space was increased to 2 GB(which is a tremendously large amount), the 17 processor model ran perfectly well.

It is not always true that the error messages given by your simulator tool are accurate. It is good to be aware of all the possible problems one might face when one is very sure that the logic is right. Hence it is very essential to make sure that the memory and the swap space on the system are enough to run one's model.

7. There are different types of tokens-**readtoken**, **writetoken** and **controltoken**. This type is specified in the **ttype** field of the token record. The **readtoken/writetoken** are used to read/write to a memory module or output device while **controltoken** is used to send an interrupt(called **event**) by one device to a destination processor. The software is written such that the processor waits on this **event** and then starts its processing. One of the errors encountered during the development of the model was, when a **controltoken** was sent from a source to the destination PE, the PE would wait on this **event** but continue to wait even after it received this **controltoken**.

There were no errors displayed during simulation, no warnings, nothing. From the simulation results, one could track down the path of the token from the source to the destination PE (as each new token was given a unique id). It was observed that the destination PE, in fact, *did* receive the token. But there was no clue why that PE did not start its processing. There was no clue whatsoever. Lack of proper documentation further aggravated the problem. So, we finally came to the logical conclusion that the PE was not able to interpret the token as a **controltoken**.

After further static analysis of the problem, it was realized that the **value** field of the token should be assigned a particular unique value which indicated a corresponding unique **event**. But an arbitrary value was given to the **value** field.

The processor model encodes the event in the **value** field(which is of type integer) as

value := event_name'POS(**event**);

and decodes it as

event := event_name'VAL(**value**);

where event_name is a one dimensional array of different event/interrupt names. For a, **readtoken/writetoken** the **value** field can be any arbitrary number as long as it is not in the range of the above array. The **controltoken** *has* to have a **value** field within the range of the above array. This type of problem could be solved only with better documentation.

8. There can be multiple tasks running on a single processor or these multiple tasks can be modeled as a single task. When a processor is waiting on multiple **events,** not separated enough temporally, the software is modeled as multitask software where each task just waits for the occurrence of just one **event**.

When the software was modeled as multiple tasks, it gave this error,

RunTime ERROR : "Multiple drivers on a ThreadBus"

As explained in **(v), task_data_record** contains the following fields:

(priority, period, deadline, kind, port_id)

It was required to map different values for the port_id for different tasks as

TASK_1=> (201, 0, time'HIGH, STATIC, 1)

TASK_2 => (210, 0, time'HIGH, STATIC, 2) and so on..

Originally, the port_ids were given the same values for all tasks, when the above change was made, the problem was solved. Again, lack of documentation was the source of the error.

9. The software to be run on one of the processors in the Multiswath processing algorithm, had multiple wait on **events** statements within a LOOP. So, there was no way of modeling this as multiple tasks. The software looked like this-

FOR I in 1 to 128 LOOP

```
        WAIT ON event_x;
         EX1 {  execute this..
                set of statements.....}
        WAIT ON event_y;
           EX2 {          execute this..
                set of statements.....}
         WAIT ON event_z;
END LOOP;
```

To make sure, this kind of software actually works, instead of incorporating this in the whole model, a separate experimental program of a single PE(say CPU1) executing the above task while another PE(CPU2) sending different **events**, **x, y, z** was written.
It was observed that CPU1 can wait on an event, say **event_y** for any amount of time after executing the set of steps, EX1. But if CPU2 sends the **event_y** while the CPU1 is still executing steps EX1, the **event_y** is not registered by CPU1 and it waits indefinitely for an **event_y**. Hence the timing becomes a very critical factor here. It has to be made sure that CPU1 is done with its processing before CPU2 sends out an **event**.

5.2.3 Comments

- If one must work with a program, one did not originally write, it is certainly important to make sure one has a clear understanding of what is normal.
- Keeping records may be helpful should one come across a similar problem.
- One should not allow unconscious presumptions to creep into one's analysis.
- Periodic review of one's progress during a debugging session to ensure that one maintains a clear understanding of the problem.
- If the software is very big and complicated, it is a good idea to break the software into pieces and test run each piece before incorporating as a single program.
- When generating a set of test trials, consider normal values, boundary values, and out-of-range values for the generics to uncover the weaknesses.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This thesis has presented a methodology to construct VHDL performance models which will help significantly reduce the time from an initial conception to a working design. To further reduce development time, existing structural primitives were reused. Also, an efficient methodology has been developed to validate performance models of complex multi-processor architectures. This method helps to make validation faster and less difficult. To illustrate these methodologies, a seventeen processor raceway architecture was constructed. Two software algorithms, SAR range processing algorithm and SAR multiswath processing algorithm were successfully run on the raceway architecture and the results were validated. Moreover, this research has resulted in the development of a high level VHDL library of hardware models and software algorithms. These models can be reused as primitives for the development of other new models with little or no modification.

Some of the areas in which future work is possible is as follows: The processor model used in the current version is very complex. It has many intricate details which

may not be used all the time. This reduces the simulation speed too. Hence, a simpler processor model is recommended.

Presently, the writing of the application software and the mapping of the software to on the processors is done manually which is quite a time consuming task. These processes could be automated.

Currently, the *route* of a token is specified statically. It could be made dynamic depending upon the availability of the bus.

The user-friendliness of the library could be improved by the use of Graphical User Interfaces(GUI).

Bibliography

[1] J.R.Armstrong , F.G.Gray, *Structured Logic Design with VHDL*, PTR Prentice Hall, Inc. 1993.

[2] J.Bhasker, *A VHDL Primer*, Kluwer Academic Publishers, 1995.

[3] F.G.Gray, J.R.Armstrong, "Reutilization Of VHDL Testbench and Library Primitives", *Proc. of AIAA Computing In Aerospace 10*, San Antonio, Tx, 1995, pp 691-700.

[4] *Semi Annual Project Review Meeting*, Blacksburg, Va, February 20, 1996.

[5] Comments from the VHDL code provided by Honeywell Inc. GOVERNMENT PURPOSE LICENSE RIGHTS LEGEND CONTRACT NUMBERS: F33615-94-C 1501(Omniview PMW), DAAL01-93-C-3380(Martin Marietta RASSP), F33615-94 C1495(VHDL Hybrid Models), F33615-92-C-3802(CDG), CONTRACTOR: Honeywell Inc. 27327.

[6] IEEE, *IEEE Standard VHDL Language Reference Manual*, New York, NY, IEEE Std. 1076-1987, March 31, 1988.

[7] J.M.Schoen et. al., *Performance and Fault Modeling with VHDL*, Prentice Hall, Inc., 1992.

[8] F.Rose, T.Steves, T.Carpenter, "VHDL Performance Modeling", *Proc. 1st Annual RASSP Conference*, Arlington, Va, August, 1994, pp 60-70.

[9] H .P.Commissariat, *Performance Modeling Of Single Processor and Multi-Processor Computer Architectures*, Master's Thesis, Virginia Tech., June 1995.

[10] B.S.Isenstein, B.C.Kuszmaul, "Overview of the RACE Hardware and Software Architecture", *Proc. 1st Annual RASSP Conference*, Arlington, Va, August, 1994, pp 179.

[11] S. Kottapalli, *A Test Plan Driven Test Bench Generation System,* Master's Thesis, Virginia Tech., May 1996.

[12] R.Gummadi, *Methodology For Structured VHDL Model Development*, Master's Thesis, Virginia Tech., May 1995.

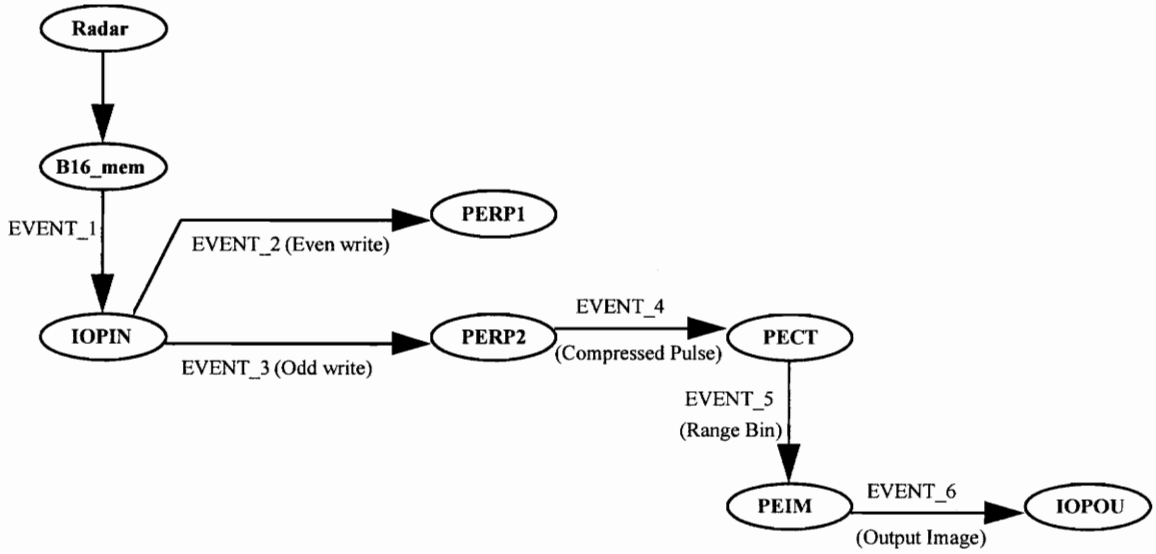
[13] S.Hrishikesh, *Behavioral Testbench Development For DSP Models*, Master's Thesis, February, 1995.

[14] P.Gowrishankaran, *Structural Testbench Development For DSP Models*, Master's Thesis, March, 1995.

[15] GPD VHDL Performance Modeling Methodology Document, Honeywell, Systems Research Center, SDRL 12, AF Contract No. F33615-90-C-3800, January 17, 1992.

APPENDIX A

Sequence Of Events for the Range Processing Algorithm.



- IOPIN - Input Processor(B16_cpu16)
- PERP1, PERP2 - Task 1 and Task 2 for Range Processors
- PECT - Corner Turn processor(B11_cpu11)
- PEIM - Image Processor
- IOPOU - Output Processor(B17_cpu17)
- Outdev - Output Device

APPENDIX B

Multiswath Processing Algorithm Specifications

The following are the sequence of processing steps for different processors:

Task for RADAR

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	pole	16256 bytes	XX	Write	RADAR	B16_mem
2	pole_ack/intr.	16 bytes	***Event_1***	Control	B16_mem	cpu16

Task 1 for *Input Processor*(cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_1*					
2	pole_read	16 bytes	16256 bytes	Read	cpu16	B16_mem
3	pole_read_ack	16256 bytes	XX	Data	B16_mem	cpu16
4	Unpack	(16256 clocks)--- processing on cpu16				
5	even_write_req	8140 bytes	XX	Write	cpu16	B16_mem
6	even_write_ack	16 bytes	XX	Ack	B16_mem	cpu16
7	odd_write_req	8140 bytes	XX	Write	cpu16	B16_mem
8	odd_write_ack	16 bytes	XX	Ack	B16_mem	cpu16

the following steps are processed by cpu16 from each pulse and sent to the *range processors* successively, cpui(where i=1,2,3,4).

9	even_write	8140 bytes	XX	Write	cpu16	Bi_mem
10	even_write_ack	16 bytes	XX	Ack	Bi_mem	cpu16
11	SEND_EVENT	16 bytes	***Event_2***	Control	cpu16	cpui
12	odd_WRITE	8140 bytes	XX	Write	cpu16	Bi_mem
13	odd_WRITE_ack	16 bytes	XX	Ack	Bi_mem	cpu16
14	SEND_EVENT	16 bytes	***Event_3***	Control	cpu16	cpui

Task 2 for *Input Processor*(cpu16)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for 512 consecutive Event_5s*					
2	aux1_write_req	308 bytes	XX	Write	cpu16	B16_mem
3	aux1_write_ack	16 bytes	XX	Ack	B16_mem	cpu16
4	aux2_write_req	308 bytes	XX	Write	cpu16	B16_mem
5	aux2_write_ack	16 bytes	XX	Ack	B16_mem	cpu16
6	aux1_read_req	16 bytes	308 bytes	Read	cpu16	B16_mem
7	aux1_read_ack	308 bytes	XX	Ack	B16_mem	cpu16
8	aux1_WRITE	308 bytes	XX	Write	cpu16	B15_mem
9	aux1_WRITE_ack	16 bytes	XX	Ack	B15_mem	cpu16
10	SEND_EVENT	16 bytes	***Event_7***	Control	cpu16	cpu15
11	aux2_read_req	16 bytes	308 bytes	Read	cpu16	B16_mem
12	aux2_read_ack	308 bytes	XX	Ack	B16_mem	cpu16
13	aux2_WRITE	308 bytes	XX	Write	cpu16	B11_mem
14	aux2_WRITE_ack	16 bytes	XX	Ack	B11_mem	cpu16
15	SEND_EVENT	16 bytes	***Event_8***	Control	cpu16	cpu11

Task 1 for the *Range Processors* , cpui(where I= 1, 2, 3, 4)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_2*					

2	even_read	16 bytes	8140 bytes	Read	cpui	Bi_mem
3	even_read_ack	8140 bytes	XX	Data	Bi_mem	cpui
4	Even FIR	(6087 clocks) -- processing on cpui				

Task 2 for the *Range Processors* , cpui(where I= 1, 2, 3, 4)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_3*					
2	odd_read	16 bytes	8140 bytes	Read	cpui	Bi_mem
3	odd_read_ack	8140 bytes	XX	Data	Bi_mem	cpui
4	Odd FIR	(6087 clocks) -- processing on cpui				
5	Form Complex (4060 clocks)					
6	weight_read_req	16 bytes	16240 bytes	Read	cpui	Bi_mem
7	weight_read_ack	16240 bytes	XX	Data	Bi_mem	cpui
8	Complex VPM	(8120 clocks)				
9	Zero Pad	(18 clocks)				
10	Complex FFT	(67584 clocks)				
11	cpulse_write_req	16384 bytes	XX	Write	cpui	Bi_mem
12	cpulse_write_ack	16 bytes	XX	Ack	Bi_mem	cpui
13	cpulse_read_req	16 bytes	16384 bytes	Read	cpui	Bi_mem
14	cpulse_read_ack	16384 bytes	XX	Data	Bi_mem	cpui
15	cpulse_WRITE	16384 bytes	XX	Write	cpui	B11_mem
16	cpulse_WRITE_ack	16 bytes	XX	Ack	B11_mem	cpui
17	SEND_EVENT	16 bytes	***Event_4***	Control	cpui	cpu11
18	SEND_EVENT	16 bytes	***Event_5***	Control	cpui	cpu16

Task 1 for the *Corner Turn Processor* , cpu11

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_8*					
2	extract srange (77 clocks)					
3	sel_ker_set	(512 clocks)				
4	SEND_EVENT	16 bytes	***Event_10***	Control	cpu11	cpu11

Task 2 for the *Corner Turn Processor* , cpu11

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for 512 consecutive Event_4s*					
FOR J IN 1 TO 2048						
2	bin_WRITE	8192 bytes	XX	Write	cpu11	B11_mem
3	bin_WRITE_ack	16 bytes	XX	Ack	B11_mem	cpu11
END LOOP;						
4	SEND_EVENT	16 bytes	***Event_6***	Control	cpu11	cpu11

Task 3 for the *Corner Turn Processor* , cpu11

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_16*					
	dest == source of Event_16					
2	bin_read	16 bytes	8192 bytes	Read	cpu11	B11_mem
3	bin_read_ack	8192 bytes	XX	Ack	B11_mem	cpu11
4	bin_WRITE	8192 bytes	XX	Write	cpu11	B11_mem
5	bin_WRITE_ack	16 bytes	XX	Ack	B11_mem	cpu11
6	bin_WRITE	8192 bytes	XX	Write	cpu11	dest_mem
7	bin_WRITE_ack	16 bytes	XX	Ack	dest_mem	cpu11

8	SEND_EVENT	16 bytes	***Event_17***	Control	cpu11	dest
9	SEND_EVENT	16 bytes	***Event_19***	Control	cpu11	cpu11

Task 4 for the *Corner Turn Processor* , cpu11

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_6 and Event_10*					

FOR S in 1 to 16 LOOP -- Process all subswaths
FOR I in 1 to 9 LOOP -- Make sure all Azimuth Processors are ready
Azimuth processors are cpui(i = 5, 6, 7, 8, 9, 10, 12, 13 14)

2	SEND_EVENT	16 bytes	***Event_11***	Control	cpu11	cpui
3	*Wait for Event_12*					

END LOOP -- Azimuth Processors are ready
FOR I IN 1 TO 9 -- Broadcast subswath kernel ptr to all Azimuth Processors

4	ptr_WRITE	4 bytes	XX	Write	cpu11	Bi_mem
5	ptr_WRITE_ack	16 bytes	XX	Ack	Bi_mem	cpu11
6	SEND_EVENT	16 bytes	***Event_13***	Control	cpu11	cpui

END LOOP; -- Kernel ptr broadcast
FOR I IN 1 TO 128 -- Process range bins in a subswath
azp_dest == source of Event_14

7	SEND_EVENT	16 bytes	***Event_15***	Control	cpu11	azp_dest
8	*Wait for Event_19*					

END LOOP; -- range bins in a subswath
END LOOP; -- Subswaths

Task 1 for the *Azimuth Processors*, cpui(where i = 5, 6, 7, 8, 9, 10, 12, 13, 14)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_11*					
2	SEND_EVENT	16 bytes	***Event_12***	Control	cpui	cpu11

Task 2 for the *Azimuth Processors*, cpui(where i = 5, 6, 7, 8, 9, 10, 12, 13, 14)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_13*					
2	SEND_EVENT	16 bytes	***Event_14***	Control	cpui	cpu11

Task 3 for the *Azimuth Processors*, cpui(where i = 5, 6, 7, 8, 9, 10, 12, 13, 14)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_15*					
2	SEND_EVENT	16 bytes	***Event_16***	Control	cpui	cpu11

Task 4 for the *Azimuth Processors*, cpui(where i = 5, 6, 7, 8, 9, 10, 12, 13, 14)

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_17*					
2	bin_read	16 bytes	8192 bytes	Read	cpui	Bi_mem
3	bin_read_ack	8192 bytes	XX	Ack	Bi_mem	cpui
4	ker_read	16 bytes	8192 bytes	Read	cpui	Bi_mem
5	ker_read_ack	8192 bytes	XX	Ack	Bi_mem	cpui
6	Complex FFT (30720 clocks)					
7	Complex VPM (4096 clocks)					
8	Complex FFT (30720 clocks)					
9	imbin_write_req	4096 bytes	XX	Write	cpui	Bi_mem
10	imbin_write_ack	16 bytes	XX	Ack	Bi_mem	cpui

11	imbin_read_req	16 bytes	4096 bytes	Read	cpui	Bi_mem
12	imbin_read_ack	4096 bytes	XX	Ack	Bi_mem	cpui
13	imbin_WRITE	4096 bytes	XX	Write	cpui	B15_mem
14	imbin_WRITE_ack	16 bytes	XX	Ack	B15_mem	cpui
15	SEND_EVENT	16 bytes	***Event_18***	Control	cpui	cpu15

Task 1 for the *Image_Processor* , cpu15

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_7*					
2	aux_read	16 bytes	154 bytes	Read	cpu15	B15_mem
3	aux_read_ack	154 bytes	XX	Ack	B15_mem	cpu15
4	aux_WRITE	154 bytes	XX	Write	cpu15	B17_mem
5	aux_WRITE_ack	16 bytes	XX	Ack	B17_mem	cpu15
6	SEND_EVENT	16 bytes	***Event_9***	Control	cpu15	cpu17

Task 2 for the *Image_Processor* , cpu15

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for 2048 consecutive Event_18s*					
FOR I IN 1 TO 2048 -- Send out lines of image to IOPOu						
2	image_read	16 bytes	4096 bytes	Read	cpu15	B15_mem
3	image_read_ack	4096 bytes	XX	Data	B15_mem	cpu15
4	Assemble Image (4096 clocks)					
5	image_WRITE	4096 bytes	XX	Write	cpu15	B17_mem
6	image_WRITE_ack	16 bytes	XX	Ack	B17_mem	cpu15
7	SEND_EVENT	16 bytes	***Event_20***	Control	cpu15	cpu17
END LOOP; -- Image lines						

Task 1 for the *Output_Processor* , cpu17

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
1	*Wait for Event_9*					
2	aux_read	16 bytes	154 bytes	Read	cpu17	B17_mem
3	aux_read_ack	154 bytes	XX	Ack	B17_mem	cpu17
4	aux_STORE	154 bytes	XX	Write	cpu17	OutDevice

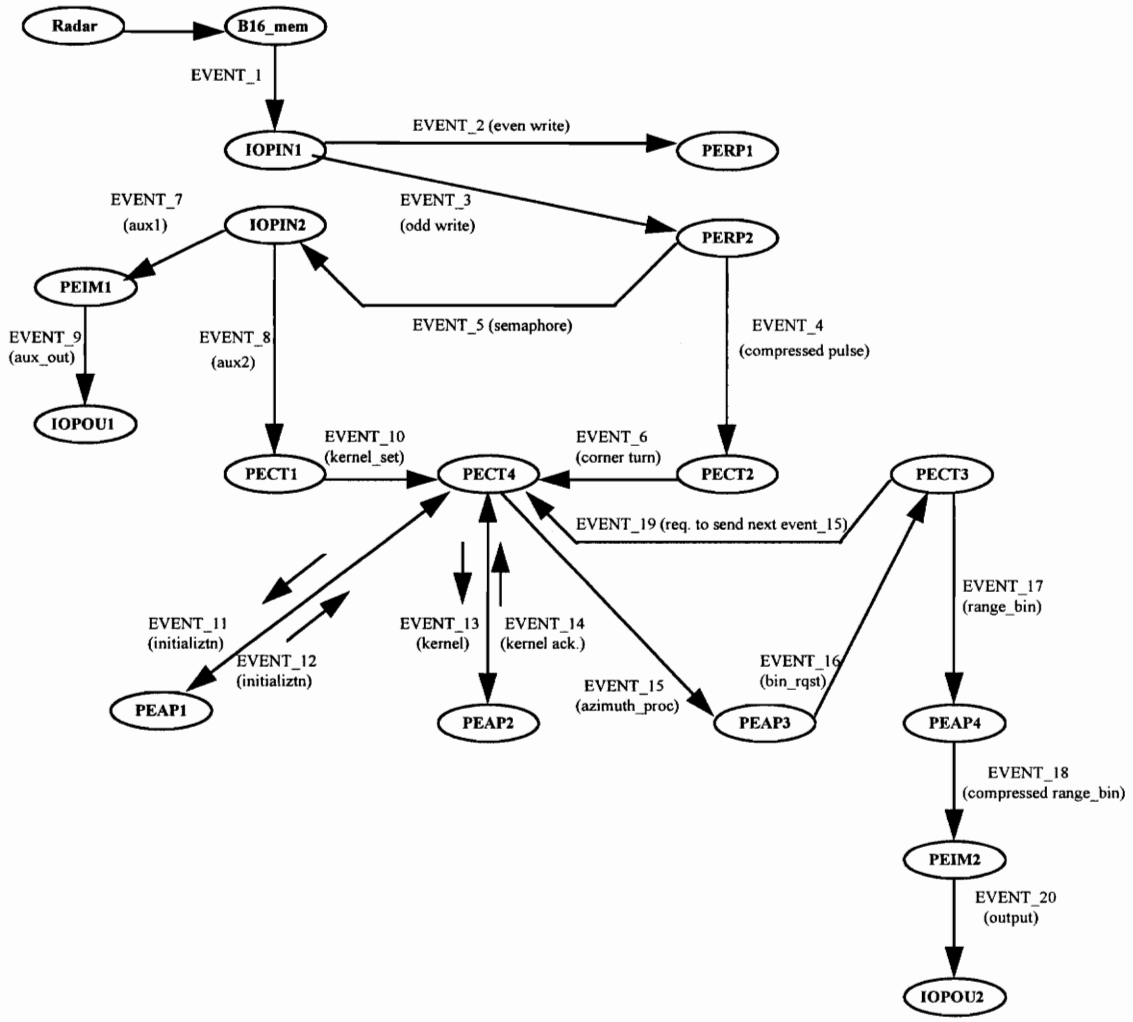
Task 2 for the *Output_Processor* , cpu17

No.	Step Name	msg_size	msg_value	Token Type	Source	Destination
FOR I IN 1 TO 2048 -- Send out lines of image to display						
1	*Wait for Event_20*-- An image line has arrived					
2	image_read	16 bytes	5210 bytes	Read	cpu17	B17_mem
3	image_read_ack	5210 bytes	XX	Ack	B17_mem	cpu17
4	image_STORE	5210 bytes	XX	Write	cpu17	OutDevice

END LOOP; -- Image lines

APPENDIX C

Sequence Of Events for the Multiswath Processing Algorithm(refer to Appendix B).



- IOPIN1, IOPIN2 - Task 1 and Task 2 for the input processor(B16_cpu16)
- PERP1, PERP2 - Task 1 and Task 2 for the range processor
- PEAP1, PEAP2, PEAP3, PEAP4 - Tasks 1- 4 for the azimuth processors
- PECT1, PECT2, PECT3, PECT4 - Tasks 1- 4 for the corner turn processor
- PEIM1, PEIM2 - Task 1 and Task 2 for the image processor(B15_cpu15)
- IOPOU1, IOPOU2 - Task 1 and Task 2 for the output processor(B17_cpu17)

VITA

Srilekha Vuppala was born on April 3, 1972 in Visakhapatnam, India. She got her Bachelor's degree in Electrical & Electronics in August 1993 from Jawaharlal Nehru Technological University, Hyderabad, India. She joined graduate school at the Virginia Polytechnic Institute and State University and received her Master's Degree in Computer Engineering in May, 1996. After graduating from Virginia Tech, Srilekha began employment with Hughes Networks System, Maryland.

Srilekha Vuppala