NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

THE DESIGN OF A
PREDICTIVE READ CACHE

by

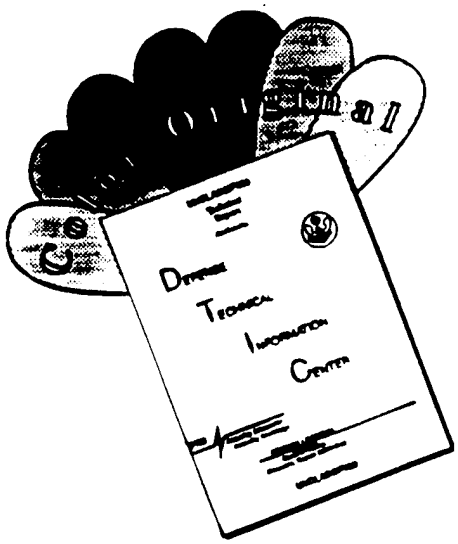
Joseph R. Robert, Jr.

March, 1996

Thesis Advisor: Douglas J. Fouts

Approved for public release; distribution is unlimited.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1996.	Master's Thesis

4. TITLE AND SUBTITLE
THE DESIGN OF A PREDICTIVE READ CACHE

5. FUNDING NUMBERS

6. AUTHOR(S) ROBERT, Joseph Roy, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The objective of this research has been the creation of a hardware design for a Predictive Read Cache (PRC). The PRC is a developmental cache intended to replace second-level caches common in modern microprocessor systems. The PRC has the potential of being faster and cheaper than current second-level caches and is distinctive in its ability to predict data addresses to be referenced by a central processing unit.

Previous research has analyzed the behavior that the PRC must exhibit. During the described research, the behavior was modeled in the Verilog hardware description language. Verilog-XL was used for simulation, which uses the Verilog behavioral model as input. The behavioral model suggests that the internal structure of the PRC could be divided into six modules, each performing part of the function of the whole PRC. Each of these blocks was studied for hardware equivalents, easing the development of the total structural model.

Using Verilog structural models as input, Epoch was used to automatically perform a very large-scale integrated (VLSI) circuit layout and to generate timing information. The Epoch output files are used for further simulation with Verilog-XL to identify critical parts of the design. The result of this research is a complete hardware design for the PRC.

14. SUBJECT TERMS VLSI (very large scale integrated) design; memory address prediction; VERILOG; EPOCH; cache.			15. NUMBER OF PAGES 200
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

THE DESIGN OF A PREDICTIVE READ CACHE

Joseph R. Robert, Jr.
Lieutenant, United States Navy
B.S., State University of New York at Buffalo, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1996

Author: Joseph R Robert Jr
Joseph R. Robert, Jr.

Approved by: Douglas J. Fouts
Douglas J. Fouts, Thesis Advisor

Frederick W. Terman
Frederick W. Terman, Second Reader

Herschel H. Loomis, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The objective of this research has been the creation of a hardware design for a Predictive Read Cache (PRC). The PRC is a developmental cache intended to replace second-level caches common in modern microprocessor systems. The PRC has the potential of being faster and cheaper than current second-level caches and is distinctive in its ability to predict data addresses to be referenced by a central processing unit.

Previous research has analyzed the behavior that the PRC must exhibit. During the described research, the behavior was modeled in the Verilog hardware description language. Verilog-XL was used for simulation, which uses the Verilog behavioral model as input. The behavioral model suggests that the internal structure of the PRC could be divided into six modules, each performing part of the function of the whole PRC. Each of these blocks was studied for hardware equivalents, easing the development of the total structural model.

Using Verilog structural models as input, Epoch was used to automatically perform a very large-scale integrated (VLSI) circuit layout and to generate timing information. The Epoch output files are used for further simulation with Verilog-XL to identify critical parts of the design. The result of this research is a complete hardware design for the PRC.

TABLE OF CONTENTS

I. INTRODUCTION

A. HISTORY

Billingsley and Fouts demonstrated the viability of using an address predicting buffer to reduce memory latency in computer systems. "The implementation of a MPB [Memory Prediction Buffer] is less expensive than a next-level cache and delivers a comparable performance enhancement." (Billingsley, 1992)

With this in mind, Nowicki designed a Read Prediction Buffer (RPB) as part of his thesis work in 1992 (Nowicki, 1992). This RPB was capable of prefetching data based on the previous pattern of memory accesses. Continuing the work of Nowicki, Aguilar tested that design and suggested several enhancements to improve it (Aguilar, 1995). A tentative design of this new Predictive Read Cache (PRC) was a part of his thesis work.

Aguilar proposed a design consisting of six modules which together would comprise the PRC. He designed four of those six modules, testing each independently, but not together.

B. PRINCIPLE OF OPERATION

The Predictive Read Cache stores data only, not instructions. The design is based on a couple of observations about data fetches from main memory. First, within a specific block of data, the accesses often occur in sequential patterns such as every element in order, or every other

element in reverse order. The second observation is that a program often uses several blocks of data concurrently.

The PRC takes advantage of the access patterns to predict future memory access addresses. The prediction is based on a linear displacement of the addresses. The PRC calculates the difference between two given addresses, then adds the difference to the most recent address to arrive at the predicted address. For example, if the Central Processing Unit (CPU) accesses the data at address 20h (hexadecimal 20) and then at address 40h, the PRC predicts that the CPU soon will need the data at 60h. Once the PRC has predicted an address, it fetches the data from that address. Once the data is stored in the PRC, the PRC can deliver that data to the CPU much more quickly than the main memory could deliver the data.

The PRC handles multiple data blocks through its "lines." Each line is capable of tracking the pattern of accesses within a unique block of data. Thus, the PRC can track only as many access patterns as it has lines.

When the cache is full and a new access pattern begins, a line has to be replaced. Lines that have not been used recently become aged. Aged lines are the first to be replaced when the cache is full.

Data incoherency is avoided through the process of flushing lines. When a line is flushed, that line is marked as containing invalid data and is made available for tracking new access patterns. If the CPU writes data to an address from which the PRC has prefetched data, the PRC flushes the line with that data.

C. RESEARCH GOALS

The objective of this research is to create a complete hardware design of the PRC. Completing the design has priority over the performance, though the performance must be better than the performance of main memory for this design to be of any value.

The performance is measured in terms of the rate at which the Central Processing Unit (CPU) can access the data in the PRC. In the microprocessor system for which this PRC design is created, data accesses occur in groups. The groups are called "bursts." Each access within a burst is called a "beat." With a 60-ns memory and a 66-MHZ system clock, the four-beat burst operation takes 8-3-3-3 cycles, that is, eight cycles for the first beat and three more cycles for each of the three remaining beats. The design of the PRC must perform at least this well and preferably much faster.

D. THESIS STRUCTURE

The Testbench is presented first, which is the Verilog model of the environment in which the PRC is expected to operate. This description includes a summary of the bus protocol and results of tests that show the correct performance of the Testbench.

The description of the behavioral model design phase is presented next. This chapter presents a simple psuedocode model of the PRC which is used to develop an appropriate data structure and block diagram for the PRC. The individual blocks are each modeled with Verilog and then connected

together in the Testbench to verify that the entire PRC works as desired.

Once the behavioral model design phase is complete, each block is converted into a hardware (structural) model. This phase of the design is detailed in Chapter IV.

This thesis also contains a description of the Computer Aided Design (CAD) tools used for this research. The descriptions include tips for making their use easier and descriptions of any problems encountered.

II. TESTBENCH

This chapter describes the Testbench, the environment in which the Predictive Read Cache (PRC) was designed to operate. In particular, it summarizes the bus arbitration protocol and explains the important aspects of each part of the Testbench. The chapter concludes with the test results of the Testbench itself.

A. OVERVIEW OF TESTBENCH

The Testbench models and simulates the environment in which the PRC design was tested. As indicated in Figure 1, it comprises four blocks, one of which is the PRC itself. The Testbench was developed with Verilog behavioral models. The CPU module simulates various functions of a PowerPC-603. The Memory module simulates the behavior of a 60-ns dynamic random access memory (DRAM). The Arbiter controls access to both the address and data busses. Each of these modules is described in more detail in the following sections, after a description of the PowerPC-603 bus protocol.

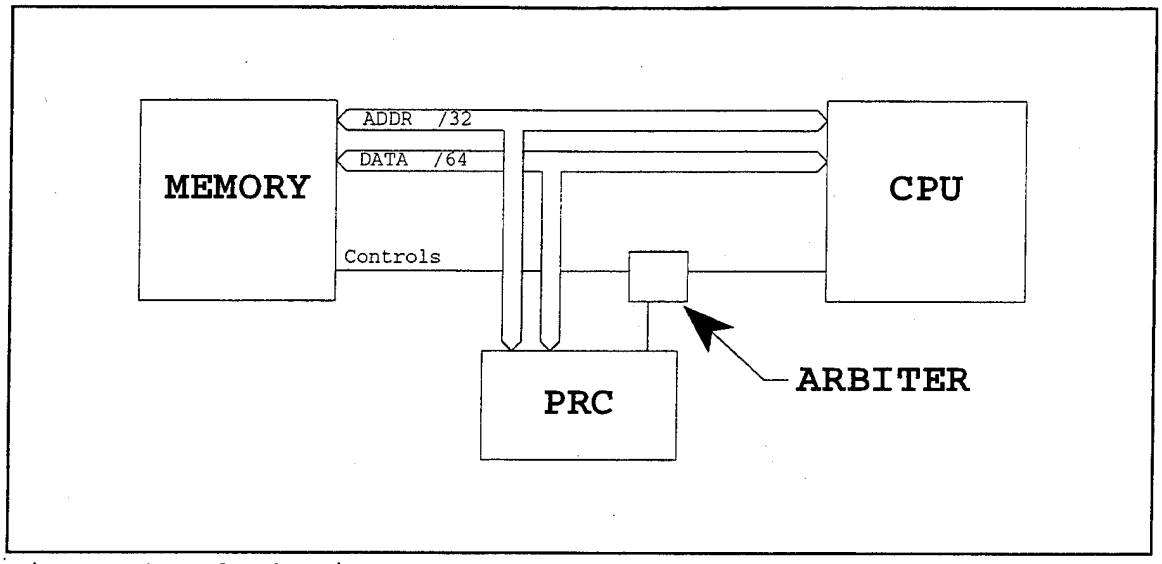


Figure 1. Block Diagram of Testbench.

There were four major decisions made regarding the design of the Testbench. The first decision was to use a PowerPC-603 microprocessor system as the environment in which this PRC will operate. The work of Aguilar was started using the '603 (Aguilar, 1995). It is still a current member of the PowerPC family; the protocol should not be out of date for quite some time.

The second design decision was to limit the '603 to in-order transactions. The '603 is capable of performing certain sequences of data transfers out of order. That is, the order of the data bus cycles can be different from the order of the address bus cycles. Prohibiting these transactions made the CPU model simpler and simplified the design of the PRC. This did not undermine the demonstration of the PRC as a viable memory management tool.

The third design decision was to use a 66-MHZ system bus and CPU clock rate. Sixty-six-MHZ is a reasonably fast system

bus speed. Designing for a slower bus speed could severely reduce the applicability of this design to modern systems.

The fourth decision was to use the 64-bit data bus vice the optional 32-bit configuration. When configured with the 64-bit data bus, the PowerPC-603 can access memory in one of two modes: single-beat or four-beat burst. A single beat is one memory access of one to eight bytes. A four-beat burst is a sequence of four sequential memory accesses, eight bytes per beat totaling 32 bytes. When configured with the 32-bit data bus, the '603 can access memory in one of three modes: single-beat (one to four bytes), two-beat burst (eight bytes), or eight-beat burst (32 bytes). Data transfers are less complicated with the 64-bit data bus since there are fewer transfer options and a smaller number of beats. Also, the time from one cache miss to the next is independent of the data bus size. Since a burst transfer on the 32-bit bus takes more cycles, there is much less time between cache misses for the PRC to do its job, perhaps too little time. Further, the 32-bit mode is specific to the '603; therefore, the PRC would have to be redesigned to be used with the other 64-bit bus members of the PowerPC family. A disadvantage of the 64-bit option is the increased number of pins required for the PRC from about 108 to about 140.

B. SUMMARY OF '603 PROTOCOL

The PowerPC-603 has separate data and address busses, each with independent cycles, referred to as tenures by the Motorola engineers. Tenure has three phases: Arbitration, Transfer and Termination.

The system has a bus arbitration unit which controls the passing of bus mastership between the requesting units. In this implementation, the CPU and the PRC are the only candidates for bus mastership. Module Arbiter is the arbitration unit.

When a unit wants the bus, it asserts *BR_* (bus request). If the unit can have the bus next, the arbiter asserts *BG_* (bus grant) back to that unit. Then the unit waits, if necessary, for the previous master to finish its tenure, after which the unit takes mastership by asserting *ABB_* (address bus busy). When the current master is done with the address bus, it negates *ABB_*.

This system has no external cache or multiple processors; thus, there are no address-only transactions. If a unit wants the address bus, it will also want the data bus. After granting the address bus by asserting *BG_*, the arbiter then grants the data bus by asserting *DBG_*.

Both *BG_* and *DBG_* remain asserted until the requesting unit takes mastership or withdraws its request by negating *BR_*. If there are no pending bus requests, the arbiter "parks" the CPU by granting it the busses. If the CPU is parked, it does not have to take the time to request the bus, thereby reducing the time for the memory access. If the CPU is parked and the PRC requests the bus, the arbiter unparks the CPU and grants the bus to the PRC.

C. TESTBENCH

The Testbench is the highest level in the design hierarchy. It connects the CPU, PRC, memory, and arbitration unit. This module establishes the system clock rate and controls the simulation time.

D. CPU

The CPU module simulates PowerPC-603 memory accesses. The Sequencer is a sub-module of the CPU which makes the Testbench able to simulate every transaction relevant to the memory and PRC. These transactions can occur in any order. Many of the possible '603 transactions are not applicable to this particular system configuration. For example, none of the "address only" transactions are relevant, since they are for systems with multiple processors or second-level caches. Bus arbitration is accurately modeled, including the pipelined address tenures.

E. MEMORY

This module emulates the main memory of the system. For simulation efficiency, the memory has only enough physical address space for four-beat burst reads: 128 bytes. The address bus width allows a virtual address space of four Gbytes. Accesses to addresses past the first 128 bytes map to addresses within the first 128 bytes.

The time required for memory accesses are determined by the use of the parameters *Delay1* and *Delay2*. The heading in

the file *memory.v* describes how to adjust these parameters to achieve a realistic memory access rate.

There were two significant decisions made about the main memory design. First, the memory emulates a 60-ns DRAM memory. With a 60-ns memory and a 66-MHZ system clock, the four-beat burst operation takes 8-3-3-3 cycles, that is, eight cycles for the first beat and three more cycles for each of the three remaining beats.

The second design decision was to add a cancel feature to the main memory chip. The memory module has an input called *CANX* which cancels the current read operation. It is through this signal that the PRC stops the memory module from delivering data to the CPU when the PRC already has the data.

Another option would be to put the PRC between the CPU and Memory, not allowing a read request to get to the memory chip until after the PRC had checked its contents. This scheme would increase the time of all memory accesses.

F. ARBITER

The Arbiter emulates the external bus arbitration unit, implemented as a Finite State Machine (FSM) corresponding to the state diagram in Figure 2.

The memory unit in this Testbench is capable of handling up to two memory accesses in the pipeline at a time, which is the maximum that the CPU will ever cause. Adding the PRC to the system creates the possibility of three accesses in the pipe. For example, the PRC could initiate a third address tenure before the first of two CPU transactions is complete. This potential problem is handled by the Arbiter which keeps track of the pipelining depth. It will not grant the address

bus to any unit if that address tenure would put a third transaction in the pipeline. Rather, the Arbiter will stall until the data tenure from the first transaction is complete, after which the Arbiter will grant the address bus to the requesting unit.

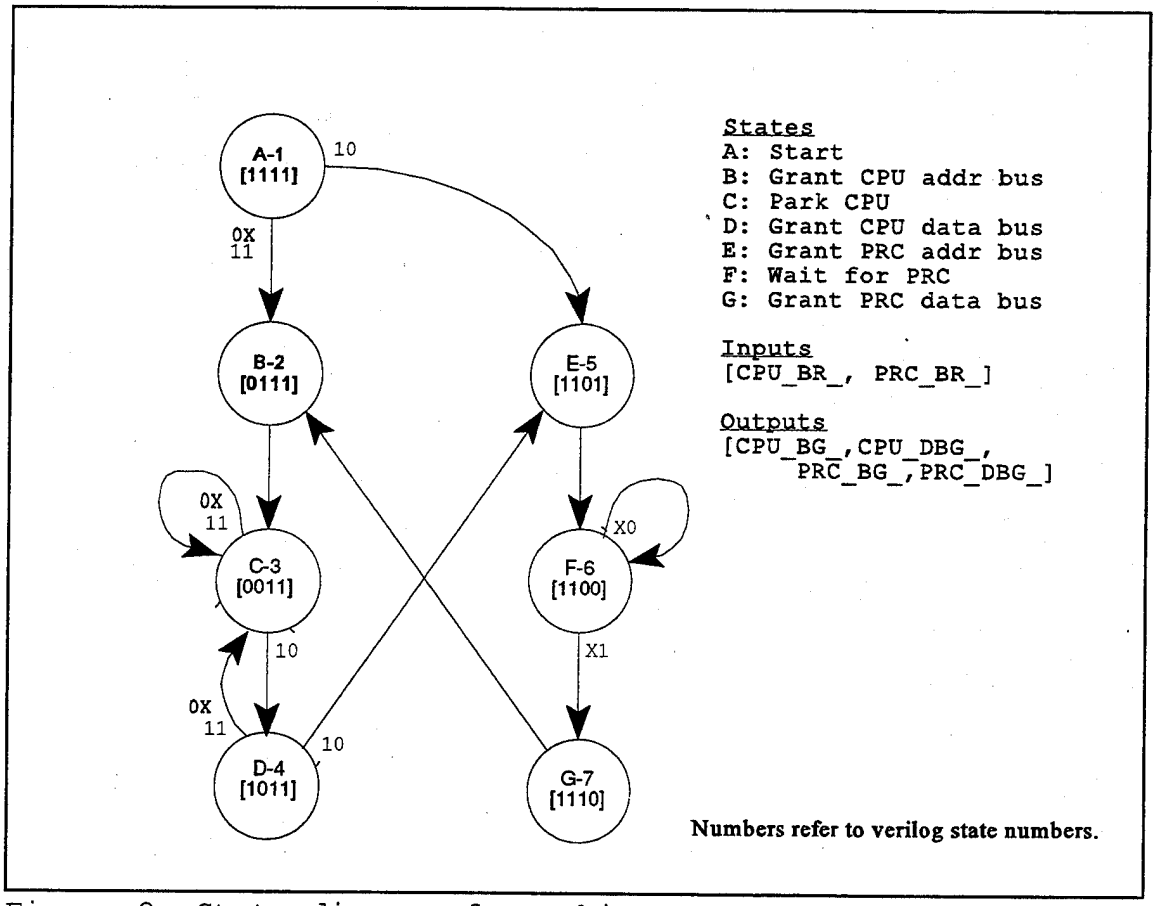


Figure 2. State diagram for Arbiter FSM.

G. TEST RESULTS

Testing the Testbench itself was important to establish that the models matched the behavior described in the *Power PC User's Manual*. The Testbench passed all tests of reads,

writes and burst operations, in various sequences of transactions and using an assortment of memory access delays.

Figure 3 shows the fastest possible burst operations, as if the memory access time were not the limiting factor. Note again that the address tenure of the second transaction can start before the data tenure of the first transaction is complete.

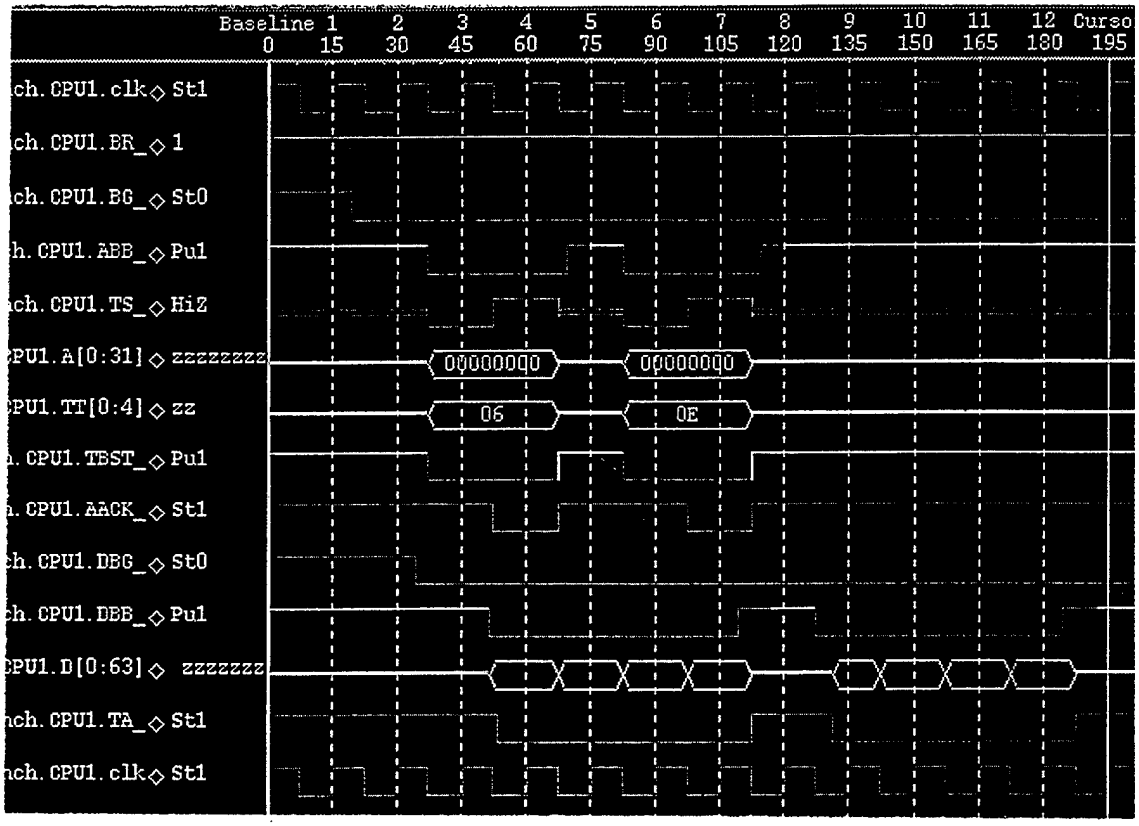


Figure 3. Burst write, then burst read. Delay=0. [cWaves output]

Figure 4 shows a burst write transaction with an access delay of three cycles and a delay of one cycle in between each beat. A realistic 60-ns DRAM will have a delay of 8-3-3-3

rather than the 3-1-1-1 shown here. The PRC however should be able to supply data this quickly.

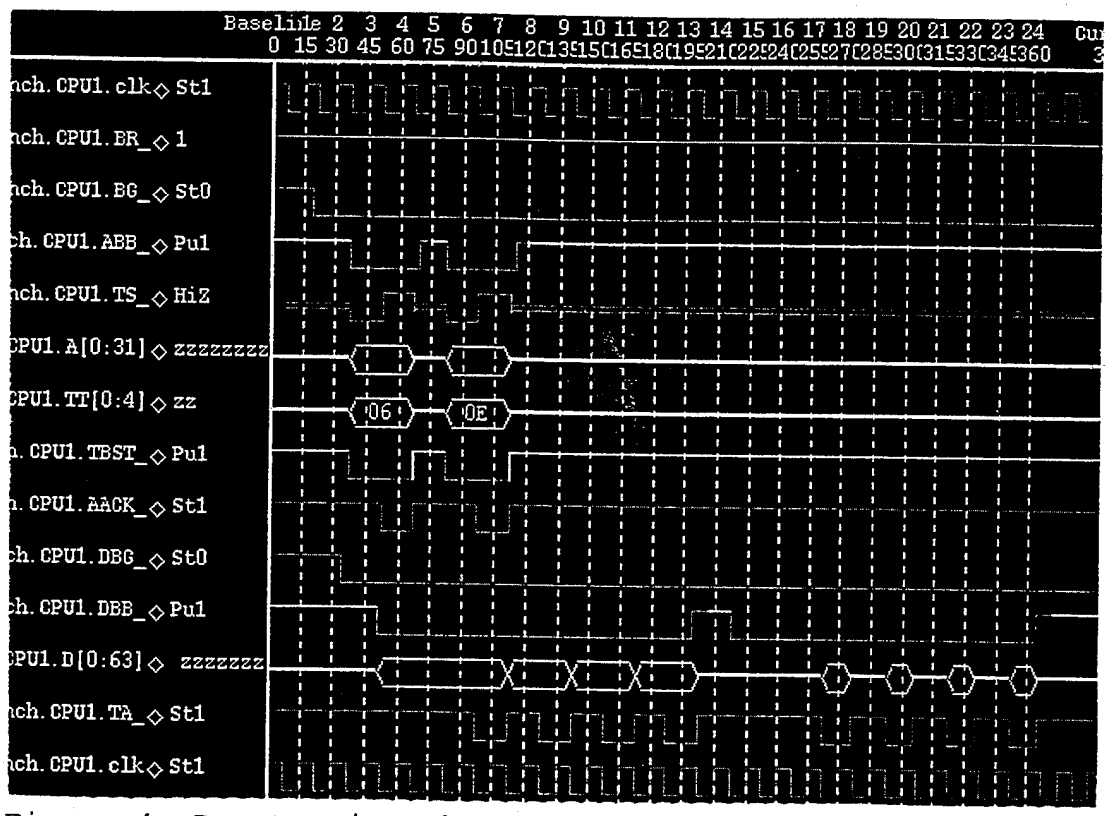


Figure 4. Burst write, burst read. Delay=3-1-1-1. [cWaves output]

III. PRC BEHAVIORAL MODEL DESIGN PHASE

This chapter presents the development of the behavioral models for the PRC. A simple pseudocode model is presented first. This model was used to develop an appropriate data structure and block diagram for the PRC. The individual blocks in this block diagram were implemented with Verilog behavioral modules and tested together to verify the behavioral model of the PRC. The next step was to convert each module into a hardware model compatible with Epoch, detailed in the next chapter.

A. PSEUDOCODE MODEL

The behavior of the PRC is explained in detail in the paper by Fouts & Billingsley (1994, p.113) and summarized in the Introduction chapter of this thesis. Another way of summarizing this behavior is through a pseudocode model as shown in Figure 5, which is just detailed enough to identify the most significant capabilities the PRC must have. The purpose of taking this approach was to clarify the function of the PRC and to aid in identifying specific behaviors of this cache which the hardware needs to exhibit.

B. DATA STRUCTURE

A possible data structure for the PRC is shown in Figure 6. Each of the 128 lines within the PRC must contain two

addresses, some status information and data. The two addresses are required to maintain the memory access pattern.

There are also two seven-bit pointers, each containing a value in the range of zero to 127. The ActiveLine pointer contains the number of the line that is currently being used by the PRC. The ReplaceLine pointer contains the number of the next line to be replaced when a new line is needed.

```
*** PRC BEHAVIOR MODEL IN PSEUDOCODE ***

// CAR     = current address register
// MRMA    = most recent memory address
// PredMA = predicted memory address

always at negative edge of HRESET_
  clear all status flags;
  put PRC in IDLE state;
  ActiveLine = 0; ReplaceLine = 0;

<IDLE>
  wait for next transaction

CASE (transaction)

  data burst-read:
    if CAR hits in PRC,  //PRC has requested data
      switch ActiveLine to line that was hit;
      send data to CPU;
      send cancel signal to memory;
      predict next address;
      if next address is not already in PRC,
        read next address;
        store in ActiveLine;
        update MRMA and PredMA;

    else if CAR misses, //PRC does not have requested data
      switch ActiveLine to the next ReplaceLine;
      if this is the first miss for this line,
        store this address in MRMA;
      if this is the second miss for this line,
        initiate search for next ReplaceLine;
        predict next address;
        if next address not already in PRC,
          read next address;
          store in ActiveLine;
          update MRMA and PredMA;

  burst-write, or write:
    if CAR hits,
      flush matching line;

  data read or instruction transaction:
    ignore;

endcase;

goto IDLE;
```

Figure 5. PRC Pseudocode Model

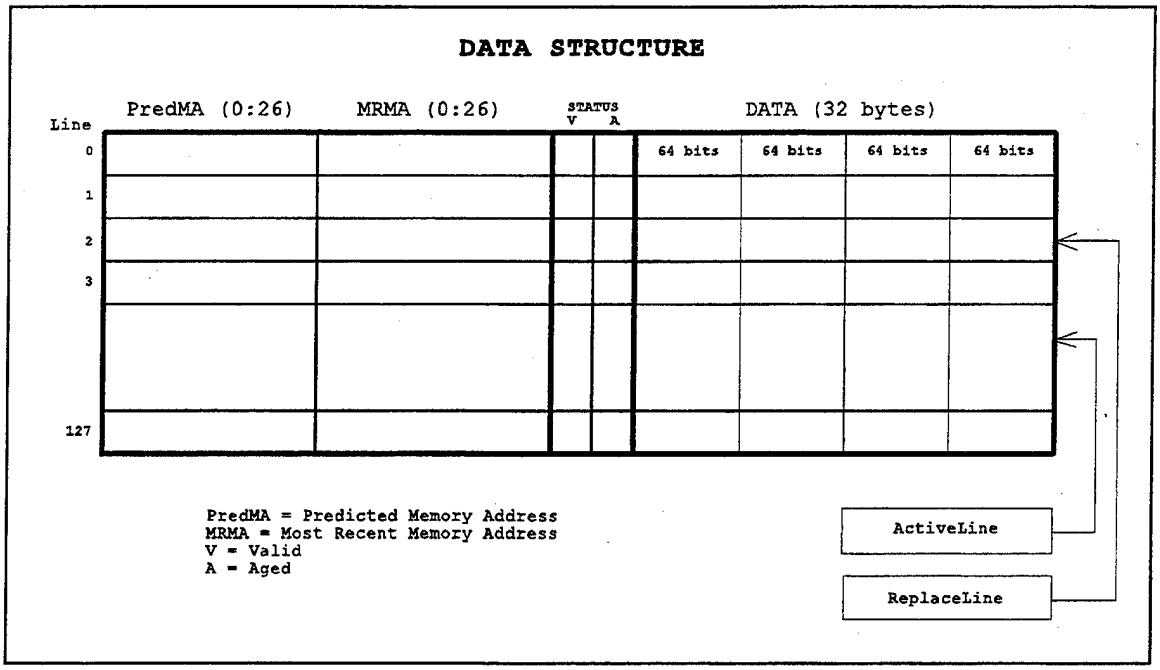


Figure 6. PRC Data Structure.

C. BLOCK DIAGRAM

The pseudocode model revealed several specific tasks the PRC must be able to accomplish. Identifying and clarifying these tasks resulted in the development of six blocks within the PRC. These blocks are shown in the block diagram of Figure 7 and are described briefly here.

The Snooper watches transactions between the CPU and memory, raising appropriate signals if the transaction is one in which the PRC is interested.

The Line Manager contains the Address List and Line Replacement Unit as sub-blocks. The Address List contains all the recently-accessed memory addresses and all the predicted addresses. The Line Replacement Unit determines which of the 128 lines will be replaced the next time a new line is needed. These two blocks are grouped together because they share

status information about the lines and work closely together for line management.

The Predictor module uses its two input addresses to predict its output address.

The Data List stores 128 lines of data, 32 bytes in each line, which is the amount of data in each burst read or burst write.

The Bus Interface handles the protocol of data transfers in to and out of the PRC.

Finally, the Controller coordinates the actions of all the other functional blocks to accomplish the mission of the PRC.

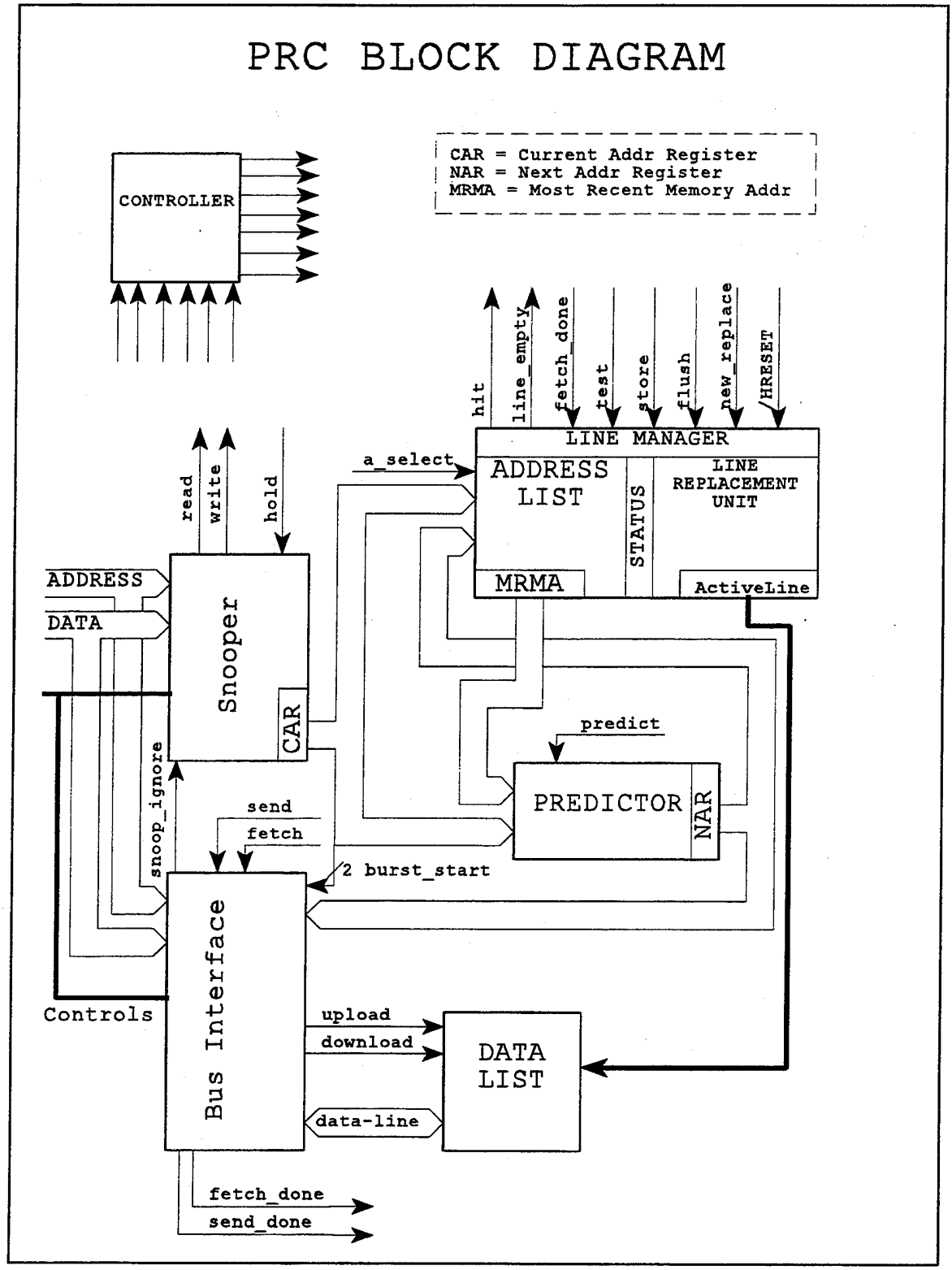


Figure 7. PRC Block Diagram.

D. CONTROLLER

This module is a Finite State Machine which coordinates the actions of all the other functional blocks of the PRC. All control signals are synchronous with the system clock. *HRESET_* causes the Controller to go to the IDLE state. The state diagram and state output tables are shown in Figures 8 and 9.

Controller State Output Table

		test		store		send	new	replace	
STATE	a select		predict		flush		hold		fetch
IDLE	X	0	0	0	0	0	0	0	0
TEST_CAR(R)	CAR	1	0	0	0	0	1	0	0
SEND_DATA	X	0	1	0	0	1	0	0	0
TEST_NAR	NAR	1	0	0	0	0	0	0	0
FETCH_DATA	NAR	0	0	0	0	0	0	0	1
IS_LINE_EMPTY	X	0	0	0	0	0	1	0	0
PREDICT_NA	X	0	1	0	0	0	1	1	0
STORE_CAR	CAR	0	0	1	0	0	1	0	0
TEST_CAR(W)	CAR	1	0	0	0	0	1	0	0
FLUSH_LINE	X	0	0	0	1	0	1	0	0

Figure 8. Controller State Output Table.

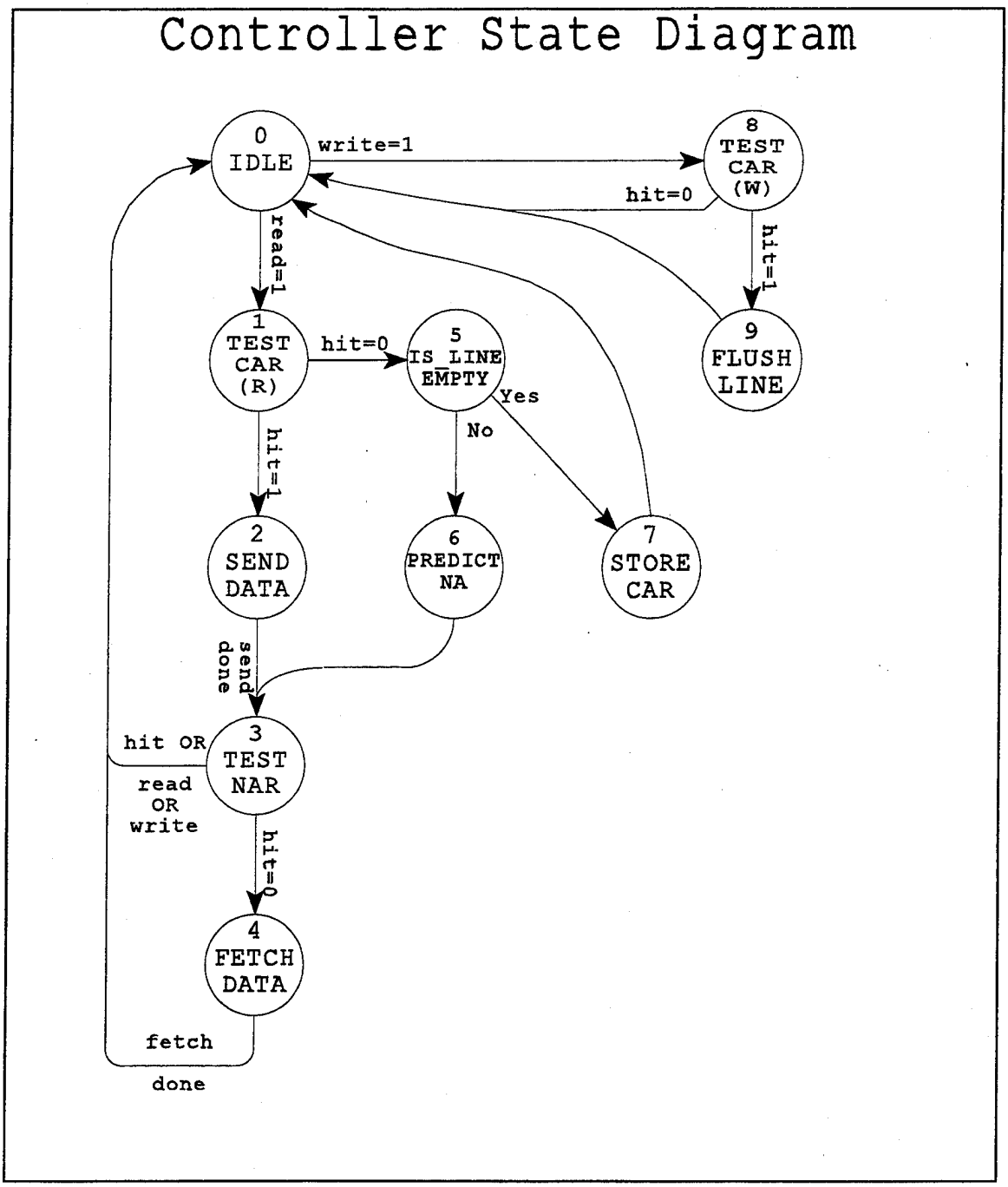


Figure 9. Controller State Diagram.

E. SNOOPER

This module watches the system bus activity and makes appropriate reports to the PRC Controller.

If the transaction is a data burst read or any kind of write and if the address parity is correct, then two actions occur. First, *read* or *write* is asserted as appropriate. Second, the address is placed in the Current Address Register (CAR). The *snoop_ignore* signal tells this unit to ignore the current transaction, because it was initiated by the Bus Interface Unit. The *snoop_ignore* signal must be asserted concurrently with the transfer attributes.

Reads that are not burst reads or data related are ignored by the PRC. The *CAR* is updated only on transactions relevant to the PRC.

Due to the two-stage pipelining capability of the PowerPC with respect to memory accesses, a second address tenure can occur shortly after the first, well before the first data tenure is complete. To compensate for this, the *read* and *write* outputs of the Snooper remain exerted until acknowledged by the Controller with *hold*. The rising edge of *hold* indicates that the *read* or *write* signal was received by the Controller. The Snooper then can negate these signals but must leave CAR alone until *hold* is negated. After *hold* is negated, *CAR* can be updated to the new address.

F. LINE MANAGER

This module contains the address list, status flags for each line (Valid, Aged), a general status flag (*line_empty*), the line replacement unit, and a couple of pointers (ActiveLine, ReplaceLine). On *HRESET_*, Valid=0 (all lines), Aged=0 (all lines), *line_empty*=1, ActiveLine=0.

The MRMA output is always the MRMA of the ActiveLine. The *line_empty* flag indicates that the currently active line has no addresses in it yet; therefore, the addresses cannot be used by the PRC to make a prediction.

The input *a_select* determines which address input is used for a particular operation. The two address inputs are the CAR and the NAR.

When the Line Manager receives a *test* signal, it compares the input address with the contents of the PredMA List. If there is a match with the CAR, it asserts the *hit* signal and changes the ActiveLine pointer to the line number of the hit.

If there is a miss with the CAR, then the ActiveLine switches to the same line to which ReplaceLine points.

If, during a test, there is a match with the NAR, two actions occur. First, *hit* is asserted. Second, the value in ActiveLine becomes irrelevant since it will not be used. If there is a miss with the NAR, the ActiveLine must remain unchanged from the test.

The *fetch_done* signal from the Bus Interface causes the NAR to be stored in PredMA[ActiveLine], the CAR to be stored in MRMA[ActiveLine], the Valid flag to be set, and the Aged flag to be reset.

The *flush* signal causes the current ActiveLine to become invalid by setting Valid[ActiveLine] = 0.

The *store* signal causes the input address to be stored into the MRMA of the ActiveLine. This is only used for the first address in a new line. The *store* signal also causes the *line_empty* flag to be reset.

Line replacement: ReplaceLine always points to the line to be replaced at the next PRC miss. *HRESET_* causes this to be zero.

As soon as the PRC starts predicting the first address for a line it asserts *new_replace*. The replacement unit then finds a new line to mark as the next ReplaceLine according the following procedure.

```
Done=false;
repeat
   ReplaceLine = ReplaceLine + 1; (mod 128 addition)
   if not(Valid[ReplaceLine])
      Done=true;
   elseif (all_line_are_valid AND Aged[ReplaceLine]) then
      Done = true;
   else
      Aged[ReplaceLine] = 1;
until Done;
line_empty=1;
```

In words, the Line Replacement Unit searches sequentially for the next line with invalid data and marks that line as the next line to be replaced. If all lines contain valid data, then it scans for the next line that is "aged," indicated by

a set Aged flag. As it scans for an aged line, it sets the Aged bits in the "unaged" lines it passes. Therefore, as it wraps around in the search for an aged line, it will eventually come upon one, even if none were aged when the search began.

All of this occurs while the PRC is fetching data. Therefore, the PRC has several clock periods in which to complete the search.

G. PREDICTOR

The Predictor module has two address inputs, the Most Recent Memory Address (MRMA) and the Current Address (stored in the Current Address Register, CAR). It has a single output, the Next Address which is stored in the Next Address Register, NAR.

This module calculates the Next Address based on the Most Recent Memory Address and the Current Address. The rising edge of *predict* initiates the prediction calculation. The original equation is

$$NAR = CAR + (CAR - MRMA)$$

which is implemented as

$$NAR = 2*CAR - MRMA.$$

The output NAR remains latched and valid until next *predict* leading edge.

H. DATA LIST

The inputs to the Data List are *upload, download* and ActiveLine. The 256-bit bus data_line is an input and output.

An *upload* signal causes the Data List to store the data on data_line into the address specified by ActiveLine. A *download* signal causes the Data List to assert onto data_line the data in the address specified by ActiveLine.

I. BUS INTERFACE UNIT

This module handles the protocol of data transfers in to and out of the PRC, coordinating these activities through the use of a Finite State Machine.

When this module receives a *fetch* signal, it latches the address in the NAR and requests the bus for a burst read. It stores the incoming data until all four bursts have been received. Then, it uploads the data into the Data List and asserts *fetch_complete*.

When this module receives a *send* signal, it sends a cancel signal (CANX) to the memory module, downloads data from the Data_List and then sends the data to the CPU. When the transfer is finished, it asserts *send_done*.

J. PREDICTION TESTS

There are two large-scale tests included in this thesis. The first is the Prediction Test. The second is the Line Replacement Test. Together, these tests are sufficient to demonstrate that the behavioral model functions as desired.

Once the behavioral model of the PRC passed these tests, it was ready for conversion to a hardware model.

The tests are both conducted by connecting the behavioral model of the PRC to the Testbench described in the previous chapter and running a simulation with a sequence of events. The sequence of events for the Prediction Test is included in the *sequencer4.v* file. The sequence of events for the Line Replacement Test is located in the *sequencer5.v* file. The following procedure lists the steps necessary to conduct a test:

1. Change directories (cd) to the *...verilog/behavior/* directory.

2. Modify the file *verilog_arguments* so that it contains *sequencer4.v* or *sequencer5.v* as desired and all the parts to the PRC and to the Testbench.

3. Modify the file *testbench.v* to set the simulation duration as described in the heading of the desired sequencer. Modify the *trace* flags in every file listed in *verilog_arguments* as described in the sequencer file.

4. At the Unix command prompt, enter the command *verilog -f verilog_arguments*.

The Verilog-XL outputs of both tests are included in the appendices. Together, these tests show that this behavioral model performs all the desired functions.

The Prediction Test, using Sequencer4, causes a series of CPU transactions that tests the ability of the PRC to make the prediction calculation and to fetch the data. The transactions are as follows:

Burst_read at 00h: The PRC stores this address.

burst_read at 20h: The PRC should predict a next address of 40h and then fetch the data from that address.

burst_read at 180h: The PRC should store this address in a new line.

burst_read at 1A0h: The PRC should predict a next address of 1C0h and then fetch the data from that address.

burst_read at 40h: This data is already in the PRC, so the PRC should send it to the CPU and then fetch data from 60h.

burst_write, 1C0h: This data is in the PRC, so this line should be flushed.

burst_read at 60h: The PRC should deliver this data to the CPU and then fetch the data at 80h.

burst_read at 100h: The PRC should start a new line and store this address.

This test successfully demonstrates a majority of the capabilities of the PRC, showing when the Line Manager selects new lines, when and how the Predictor functions, and when the CPU starts a read or write and the data involved. The test shows when the Bus Interface Unit fetched data from memory. The Data List reported the flow of data in and out of itself.

The only significant behavior not exercised by this test is the function of the Line Replacement Unit when the PRC is full. That is handled with Sequencer5 in the Line Replacement Test.

The Line Replacement Test was accomplished by a series of CPU transactions that quickly fill the PRC. The test shows

that the Line Replacement Unit correctly selected invalid lines to be replaced first. When all the lines in the PRC contained valid data, the Line Replacement Unit executed the algorithm described in the section on the Line Replacement Unit.

K. CONCLUSION

At this point in the development of the PRC, the behavioral model was functioning properly. Therefore, it could be converted piece by piece into a hardware model. This was accomplished using the subset of Verilog understood by Epoch, as described in the next chapter.

IV: PRC STRUCTURAL MODEL DESIGN PHASE

This chapter presents the development of the hardware model of the PRC. In this phase of the design process, each of the behavioral blocks developed in the previous phase was implemented with hardware. Converting the blocks in order of increasing complexity proved to work out well, making it easier to concentrate first on learning how to use Epoch.

Like the behavioral models, the hardware (structural) models are Verilog files. Epoch uses these Verilog files to create VLSI layouts. From those layouts, Epoch calculates timing information and generates new VerilogOut files with this timing information. As each block is converted into hardware, the new VerilogOut model can replace the original behavioral model in the Testbench for testing with Verilog-XL. The following hardware blocks result from using this procedure.

Each section of this chapter also includes a figure displaying some important geometric information about the module, including surface area and transistor count. This information can be obtained from Epoch with the shell command *geostat -trancount <module name>*.

A. PRC

The top level module is only a connection of each of the modules described in the following sections. The geostat information is shown in Figure 10. Of particular significance are the transistor count and the total chip area.

```
Bounding Box:
      9080.748 x 11278.224 microns, 102414707.226 square microns.
      357.510 x 444.025 mils, 158743.109 square mils.

      Number of Pins = 316.
      Number of unique cells = 6.
      Number of Datapaths = 1
      Number of Sub-Glues = 5
      Total Number of Instances = 6

      Total number of nets = 498.
      Total metal1 layer route length = 2120297.98 microns.
      Total metal2 layer route length = 699802.75 microns.
      Total metal3 layer route length = 0.00 microns.
      Total route length = 2820100.74 microns.
      Total number of vias = 2460.
      Total number of segments = 16989.
Reading transistor view ...
      Total number of 454310 transistors.
      0.349 Square mils per Transistor.
      2.862 Transistors per square mil.
Power Dissipation = 4742486.500 micro-watts.
```

Figure 10. PRC Geostat Information. [Epoch output]

B. CONTROLLER

This module is a Finite State Machine which coordinates the actions of all the other functional blocks of the PRC. All control signals are synchronous with the system clock. *HRESET_* causes the Controller to go to the IDLE state. The revised state output table (Figure 11) and the revised state diagram (Figure 12) give more details.

Of significance are the wait states added to the state diagram of the behavioral model. These changes are boldface in the Revised Controller State Output Table. The changes were required by the Line Manager in which there is a significant propagation delay for the addresses. This delay is described in more detail in the Line Manager section of

this chapter and is a prime candidate for future work to improve this design of the PRC. The geostat information is shown in Figure 13.

Controller State Output Table

STATE	a select	test	predict	store	flush	send	new	replace	hold	fetch
IDLE	**CAR**	0	0	0	0	0	0	0	0	0
WAIT_A	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_B	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_C	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_D	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_E	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_F	**CAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
TEST_CAR(R)	CAR	1	0	0	0	0	1	0	0
SEND_DATA	**NAR**	0	1	0	0	1	0	0	0
TEST_NAR	NAR	1	0	0	0	0	0	0	0
FETCH_DATA	NAR	0	0	0	0	0	0	0	1
IS_LINE_EMPTY	X	0	0	0	0	0	1	0	0
PREDICT_NA	**NAR**	0	1	0	0	0	1	1	0
WAIT_G	**NAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_H	**NAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
WAIT_I	**NAR**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
STORE_CAR	CAR	0	0	1	0	0	1	0	0
TEST_CAR(W)	CAR	1	0	0	0	0	1	0	0
FLUSH_LINE	X	0	0	0	1	0	1	0	0

Figure 11. Revised Controller State Output Table. Changes highlighted.

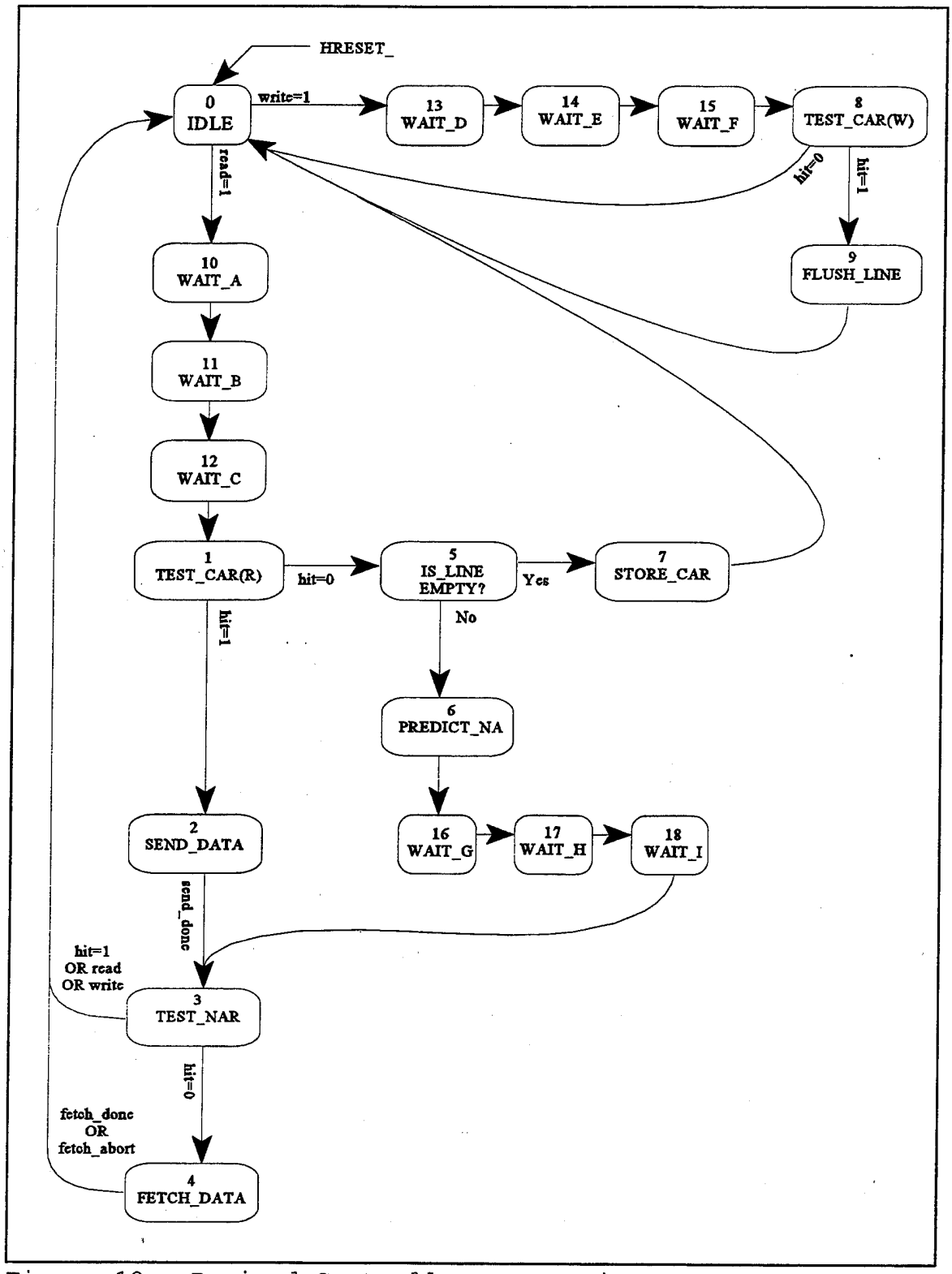


Figure 12. Revised Controller State Diagram.

```
Bounding Box:
     267.516 x 215.964 microns, 57773.825 square microns.
     10.532 x 8.503 mils, 89.550 square mils.

     Number of Pins = 26.
     Number of unique cells = 18.
     Number of Standard cells = 60
     Total Number of Instances = 60

     Total number of nets = 71.
     Total metal1 layer route length = 7073.14 microns.
     Total metal2 layer route length = 7073.46 microns.
     Total metal3 layer route length = 0.00 microns.
     Total route length = 14146.60 microns.
     Total number of vias = 226.
     Total number of segments = 1074.
Reading transistor view ...
     Total number of 460 transistors.
     0.195 Square mils per Transistor.
     5.137 Transistors per square mil.
Power Dissipation = 3665.888 micro-watts.
```

Figure 13. Controller Geostat Information. [Epoch output]

C. SNOOPER

This module watches the system bus activity and makes appropriate reports to the PRC Controller.

If the transaction is a data-burst read or any kind of write and if the address parity is correct, then the *read* or *write* signal is asserted as appropriate. Also, the address is placed in the CAR. The *snoop_ignore* signal tells this unit to ignore the current transaction, because it was initiated by the Bus Interface Unit. The *snoop_ignore* signal must be asserted concurrently with the transfer attributes. Reads

that are not burst or data related are ignored by the PRC. The CAR is updated only on transactions relevant to the PRC.

Due to the two-stage pipelining capability of the PowerPC with respect to memory accesses, a second address tenure can occur shortly after the first, well before the first data tenure is complete. To compensate for this, the *read* and *write* outputs of the Snooper remain asserted until acknowledged by the Controller with *hold*. The rising edge of *hold* indicates that the *read* or *write* signal was received by the Controller. The Snooper then can negate these signals, but must leave CAR alone until *hold* is negated. After *hold* is negated, CAR can be updated to the new address.

In Stage 0, the transfer attributes are latched in registers. Combinational logic determines if these transfer attributes represent a valid read or a valid write and if the address parity is correct. If the transaction is valid and one in which the PRC is interested, then Stage 0 raises a *transaction_waiting* signal.

A Finite State Machine in Stage One sits in the IDLE state until it receives the *transaction_waiting* signal. Then it latches the signals needed from Stage 0, resets the *transaction_waiting* signal and then waits for the *hold* signal to go low. A high *hold* signal indicates that the PRC is not done with the previous transaction. Once *hold* goes low, the *read* and *write* flags are set according to the type of the current transaction. Also, the input address is stored in the Current Address Register. The FSM then waits for the rising edge of *hold* before returning to the IDLE state where it can check if there is another transaction waiting. The geostat information is shown in Figure 14.

```
Bounding Box:
     607.500 x 409.536 microns, 248793.127 square microns.
     23.917 x 16.123 mils, 385.630 square mils.

     Number of Pins = 88.
     Number of unique cells = 19.
     Number of Standard cells = 169
     Total Number of Instances = 169

     Total number of nets = 219.
     Total metal1 layer route length = 28547.10 microns.
     Total metal2 layer route length = 14615.39 microns.
     Total metal3 layer route length = 0.00 microns.
     Total route length = 43162.49 microns.
     Total number of vias = 464.
     Total number of segments = 2268.
Reading transistor view ...
     Total number of 3608 transistors.
     0.107 Square mils per Transistor.
     9.356 Transistors per square mil.
Power Dissipation = 26722.156 micro-watts.
```

Figure 14. Snooper Geostat Information. [Epoch output]

D. LINE MANAGER

This structural model uses a high speed RAM (*hsram*) for the MRMA List. The CAR is stored into this RAM on a *store* or *fetch_done* signal.

The predicted_ma_list is a register file for storing predicted memory addresses. This list is composed of 128 address registers, 128 equality comparators and 128 Valid status flags. The NAR is stored in this list at the *fetch_done* pulse. If there is a match with the input address (*in_addr*), a priority encoder (*ENC_C*) determines which line matches.

The Line Replacement Unit determines the next line to be replaced whenever the PRC needs to start a new line. It first selects invalid lines. If all the lines are valid, then it selects lines that have been "aged." A priority encoder (*ENC_1*) chooses the line with the lowest index among all the lines that can be replaced. If all lines are valid, the output enable (*oe*) signal of the encoder is used to cause aging. A line X can be replaced if the following holds true for that line:

not (X=ActiveLine) AND {not Valid[X] OR (all_lines_valid AND Aged[X])}

Aging is accomplished by the use of a seven-bit counter (ager_counter), initially set to zero. When the *cause_aging* signal from the encoder is high, the counter advances. A decoder (*DEC_B*) output causes the appropriate Aged flag to be set.

Changing values of the CAR or NAR have a propagation delay of 25 ns (1.8 cycles) through the input address multiplexer (*in_addr mux*). This required the addition of wait states in the Controller before each of the tests. The Revised Controller State Output Table and the Revised Controller State Diagram found in the Controller section of this chapter show the required changes. The geostat information is shown in Figure 15.

```
Bounding Box:
      6704.064 x 8897.364 microns, 59648499.103 square microns.
      263.940 x 350.290 mils, 92455.359 square mils.

      Number of Pins = 505.
      Number of unique cells = 22.
      Number of Standard cells = 123
      Number of Blocks = 1
      Number of Sub-Glues = 2
      Total Number of Instances = 126

      Total number of nets = 357.
      Total metal1 layer route length = 1017746.50 microns.
      Total metal2 layer route length = 463265.70 microns.
      Total metal3 layer route length = 0.00 microns.
      Total route length = 1481012.19 microns.
      Total number of vias = 2157.
      Total number of segments = 10524.
Reading transistor view ...
      Total number of 207467 transistors.
      0.446 Square mils per Transistor.
      2.244 Transistors per square mil.
Power Dissipation = 1777694.500 micro-watts.
```

Figure 15. Line Manager Geostat Information. [Epoch output]

E. PREDICTOR

The purpose of this module is to calculate the Next Address (stored in NAR) based on the Most Recent Memory Access (MRMA) and the Current Address (in the CAR). The prediction calculation is

$$NAR = 2*CAR - MRMA$$

In this structural implementation of the Predictor, the *predict* signal is the latch for the CAR and MRMA registers. The subtraction is accomplished as a two's compliment addition with a high speed adder.

The CAR is multiplied by two, an arithmetic shift left of one bit. The most significant bit of the CAR is not retained, as it will not have an effect on the 27-bit output of the adder. This will adversely affect address prediction only around the midpoint of the four gigabytes of memory. The applicable Golden Rule of computer design "is to make the common case fast: In making a design tradeoff, favor the frequent case over the infrequent case." (Hennessy, 1990)

A number is negated in two's compliment by inverting all the bits and adding '1'. The MRMA is negated by inverting all its bits. Adding the required '1' is implemented as a Carry-In to the adder.

The Epoch TACTIC tool reported the propagation delay from *predict* to NAR to be 4.90 ns. The geostat information is shown in Figure 16.

```
Bounding Box:
     261.900 x 895.824 microns, 234616.293 square microns.
     10.311 x 35.269 mils, 363.656 square mils.

     Number of Pins = 113.
     Number of unique cells = 10.
     Number of Blocks = 107
     Total Number of Instances = 107

     Total number of nets = 230.
     Total metal1 layer route length = 12158.68 microns.
     Total metal2 layer route length = 15209.06 microns.
     Total metal3 layer route length = 0.00 microns.
     Total route length = 27367.74 microns.
     Total number of vias = 392.
     Total number of segments = 1793.
Reading transistor view ...
     Total number of 3027 transistors.
     0.120 Square mils per Transistor.
     8.324 Transistors per square mil.
Power Dissipation = 27722.887 micro-watts.
```

Figure 16. Predictor Geostat Information. [Epoch output]

F. DATA LIST

This module stores the data retrieved from memory in anticipation of a request by the CPU. The basic memory cell is the Epoch part *hsramoe* (high speed ram with output enable). Since each *hsram* has a maximum word size of 128 bits, there are two *hsram* parts in parallel to get the required 256-bit width.

An *upload* signal causes the Data List to store the data on *data_line* into the address specified by *ActiveLine*. The input *upload* has to be inverted to match the active-low *WR* input of the Epoch *hsram* component. A *download* signal causes

the Data List to assert onto *data_line* the data in the address specified by *ActiveLine*. This signal also has to be inverted for the same reason.

Both the invertors can probably be removed if the Bus Interface Unit makes the *upload* and *download* signals active low. That could only improve the response time of the data memory.

Epoch calculated the following timing delays:

download -> hsramoe.DOUT 2.3 ns
ActiveLine -> hsramoe.DOUT 7.3 ns

A design alternative is to use the regular speed version, ramoe, which gives the following timing delays:

download -> ramoe.DOUT 4 ns
ActiveLine -> ramoe.DOUT 16 ns

Using this slower RAM is possible, but would require a significant modification to the PRC behavior to handle the longer delay and would add a cycle delay to CPU reads when there is a hit in the PRC.

Putting the VerilogOut file of this module into the original PRC behavioral model for mixed-mode simulation caused a timing error that had to be corrected in the Bus Interface Unit behavioral model. After an *upload* to the Data List, *data_line* must remain valid long enough to meet the data hold time requirement of the Epoch part *hsramoe*. The geostat information is shown in Figure 17.

```
Bounding Box:
      3834.792 x 3222.936 microns, 12359289.299 square microns.
      150.976 x 126.887 mils, 19156.938 square mils.

      Number of Pins = 282.
      Number of unique cells = 3.
      Number of Standard cells = 2
      Number of Blocks = 2
      Total Number of Instances = 4

      Total number of nets = 269.
      Total metal1 layer route length = 198805.54 microns.
      Total metal2 layer route length = 52952.76 microns.
      Total metal3 layer route length = 0.00 microns.
      Total route length = 251758.30 microns.
      Total number of vias = 728.
      Total number of segments = 2422.
Reading transistor view ...
      Total number of 214712 transistors.
      0.089 Square mils per Transistor.
      11.208 Transistors per square mil.
Power Dissipation = 2181481.250 micro-watts.
```

Figure 17. Data List Geostat Information. [Epoch output]

G. BUS INTERFACE

This module connects the PRC with the system bus. It handles the protocol of data transfer in and out of the PRC.

When this module receives a *fetch* signal, it latches the address in the *NAR* and requests the bus for a burst read. It stores the incoming data until all four bursts have been received. Then it uploads the data into the Data List and asserts *fetch_done*. If there is a parity error during the fetch, the Bus Interface informs the Controller by asserting *fetch_abort*. Also, the transaction is canceled.

When this module receives a *send* signal, it sends a cancel signal (*CANX*) to the memory module, downloads data from the Data List and then sends the data to the CPU. When the transfer is finished, it asserts *send_done*.

The coordination of these activities is accomplished through the use of two Finite State Machines. One acts as an address bus master. The other controls the flow of data. The geostat information is shown in Figure 18.

```
Bounding Box: -6264, -6408, 2246040, 1972980.
      2252.304 x 1979.388 microns, 4458183.285 square microns.
      88.673 x 77.929 mils, 6910.198 square mils.

      Number of Pins = 448.
      Number of unique cells = 56.
      Number of Standard cells = 1393
      Number of Sub-Glues = 1
      Total Number of Instances = 1394

      Total number of nets = 1843.
      Total metal1 layer route length = 676479.94 microns.
      Total metal2 layer route length = 469079.94 microns.
      Total metal3 layer route length = 0.00 microns.
      Total route length = 1145559.87 microns.
      Total number of vias = 9679.
      Total number of segments = 44298.
Reading transistor view ...
      Total number of 24403 transistors.
      0.283 Square mils per Transistor.
      3.531 Transistors per square mil.
Power Dissipation = 237269.750 micro-watts.
```

Figure 18. Bus Interface Geostat Information. [Epoch output]

H. TESTING

The most significant large-scale test of the structural model is the Prediction Test, which is similar to the Prediction Test of the behavioral model. The test runs the

same series of CPU transactions to exercise all functional blocks of the PRC. The sequence of events for the Prediction Test is included in the *sequencer4.v* file.

The following steps are required to conduct a test:

1. Change directories (cd) to the *...verilog/hardware/* directory on the Computer Center (CC) system.
2. At the Unix command prompt, enter the command *verilog -f verilog_arguments*.

The Verilog-XL output of the test is included in the appendices. This test shows that the structural model of the PRC performs the desired functions. The output of the structural model test is different from the output of the behavioral model test mainly because the new structural model does not contain the same display commands. These commands interfere with the Epoch compilation of the modules. Other display commands were added to the Testbench, which is still a behavioral model. The displays are sufficient to show that PRC performs as desired.

While compiling the source files, Verilog-XL reports four warnings about implicit wires having no fanin. These wires are labeled NC0 and NC1, deriving their initials from "not connected." They are unused outputs on a couple of Epoch parts. Therefore, these warnings can be ignored.

The section with comments about SDF Annotation is the result of incorporating the Epoch timing analysis into the Verilog model. Once that annotation is complete, the actual simulation begins.

The error messages at the beginning of the simulation can be ignored. These error messages are generated by Epoch parts

and indicate improper signal values or timing. All these errors occur before the system hard reset and are expected. Having those errors after the system hard reset would have indicated a real problem.

Once the system has reset, the CPU starts its series of transactions, beginning with reads from addresses 00h and 20h. The comment "PRC requested the bus" indicates that the PRC is prefetching data. It appears that the prefetch occurs before the start of the second CPU transaction, but in reality it occurs just after the second CPU address tenure, which is not shown in the report. Also not shown because of the limitation of display commands with the PRC is the data prefetched by the PRC. That the data is correct can be seen later in the report, when the PRC sends the data to the CPU.

During the CPU to Memory transactions, there is 60 ns between each of the four beats of data. When the CPU reads from address 40h, the speed advantage of the PRC is demonstrated. Note that there is now only 15 ns between each beat. That is the period of the system clock and is therefore the maximum possible rate the CPU can receive data.

The write to address 1C0h occurred after the PRC had prefetched that data. The PRC should have flushed the prefetched data, because it was no longer valid. Later, when the CPU performs a read from the same address, it can be seen from the read data and from the timing (60 ns per beat) that the CPU is getting the data from main memory. In accordance with its design, the PRC did not try to give the stale data to the CPU.

V. CAD TOOLS

The three primary design tools used in the development of this PRC were Verilog-XL, cWaves and Epoch. This chapter describes some of the particularly useful features of these tools and gives some tips for using these tools together.

A. VERILOG-XL

Verilog-XL allows the modeling of circuits in a programming language. Circuits can be modeled by behavior or structure. For the complex design of the PRC, it was convenient to start by dividing the design into six blocks and then using Verilog to model the behavior of each block. This allowed clarification of the required behaviors, deferring the search for hardware solutions until after the desired behaviors were well defined.

Currently, Verilog-XL is available only on the Computer Center (CC) network. The following steps make it easier to use from an Electrical and Computer Engineering (ECE) workstation:

1. Add the following line to the *.cshrc* file in the ECE account: *alias rcc 'xhost in50204.cc.nps.navy.mil; rlogin -l <username> in50204.cc.nps.navy.mil'*.
2. Re-source the session by typing "sc <return>".
3. Type "rcc <return>" to log into the CC account.
4. Add the following line to the *.cshrc* file in the CC account: *alias remote3 'setenv DISPLAY*

sun3.ece.nps.navy.mil:0.0' The *.cshrc* file can contain similar lines for other workstations.

5. Re-source as in Step 2.

Now the ECE workstation becomes the display for the CC workstation. Typing "filemgr &" will call up the CC file manager.

Typing "verilog <return>" should give a list of options for use with Verilog-XL and will verify access to the program. One particularly useful option is to put all the arguments in a file, such as *verilog_arguments* and put the following line in the CC *.cshrc* file:

```
alias veri 'verilog -f verilog_arguments'
```

Typing "veri" is much easier than listing the names of all the files that need to be included in the simulation.

The Cadence online documentation can be accessed with the command "openbook &". The Main Menu is the starting point. The Alphabetical List on the bottom is the easiest way to find the desired information. In this list there is a Verilog-XL section which contains hyperlinks to the Verilog-XL Reference Manual and Tutorial.

B. CWAVES

This tool is indispensable for the analysis of complicated circuits. There is nothing like seeing a timing diagram to track down design errors.

The database for the cWaves Viewer is created while running the Verilog simulation. The highest level Verilog

module should have the following two lines in an "initial" block:

```
$shm_open;
$shm_probe(<name>,"AS");
```

where <name> is the instance name of the module to be observed. More information about these $shm commands can be found in the cWaves Reference Manual, which is a little difficult to find. It is in the Cadence Online Library accessed with "openbook & <return>". Once the Main Menu appears, select the Alphabetical List on the bottom. The cWaves Reference Manual is filed under Composer (Schematic Entry), Design Framework II. Section 4 of this manual is particularly useful.

C. EPOCH

A circuit designer would find it very convenient if Epoch would take as input the raw behavioral models, but it does not. Each behavioral block must be converted into a structural model. Then, Epoch can automatically generate a Very Large Scale Integrated (VLSI) circuit layout using a rule set from a specific manufacturer. From the layout, Epoch performs a timing analysis of the circuit and generates a new Verilog file, which includes the timing information. This new file then can replace the behavioral model for resimulation with Verilog-XL. This allows the designer to verify each block as it is designed. CWaves can be used to track down timing errors.

Epoch is available on the ECE system. To access Epoch, add "/tools3/epoch/bin" to the "set path" command in the *.cshrc* file. Also, add "setenv CASCADE /tools3/epoch".

The *Epoch User's Tutorial* and the *Epoch Verilog Interface Reference* are both very useful. The former is located at */tools3/epoch/data/examples/tutorial*. The latter can be accessed through pull-down menus in Epoch:

Help => On-Line Manual...

Sometimes calling up this manual causes a FrameViewer error, but the manual does come up after a slight delay.

The VerilogOut option proved very useful in the development of the PRC. With this option, Epoch creates a new Verilog file after laying out a design. The new model can be inserted in place of the old behavioral model for simulation with Verilog-XL. The Verilog Interface reference describes how this is done. In addition to the procedures described there, it will be necessary to take a few extra steps.

1. If the files must be moved from the *vout* directory to another directory for simulation with Verilog-XL, correct the $sdf_annotate path in the .v file.

2. In all the behavioral files, add a `timescale directive like the one in the .v file generated by Epoch. This must appear before the "module" statement.

3. It may be necessary to copy *primelib.v* from */tools3/epoch/data/verilog* into the CC directory.

The PowerPC uses bit zero as the most significant bit of buses, so it was convenient to follow that convention in this PRC design. For example, the PowerPC address bus is

designated A[0:31]. Unfortunately, this causes a problem with the VerilogOut program, which reorders some of the indices and connects busses in reverse order. This problem seems to be unique to the VerilogOut file generation. The physical layout itself gets connected correctly regardless of the index numbering convention. Resolving this problem required renumbering the indices of all modules used for Epoch input so that the most significant bit had the highest index, such as A[31:0].

VI: CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

In conclusion, the objective of this research has been met. This thesis presents a complete hardware design for the PRC. The simulation results show that the PRC can deliver data to the CPU at the rate of 8-1-1-1, that is eight cycles for the first beat and one cycle for each of the remaining three beats. This performance is better than the performance of main memory (8-3-3-3). With a little more work on the design, the PRC should be able to deliver data at a rate of 4-1-1-1.

Aguilar proposed a design consisting of six modules which together would comprise the PRC. He took a bottom-up approach, designing four of those six modules, testing each independently, but not together. (Aguilar, 1995) As a result, the designs of these modules require modifications to enable them to function correctly together. Rather than redesigning the four modules, the approach taken during this research was top-down. That is, a single working behavioral model was divided into six behavioral models that functioned together, and then each of the six behavioral models was converted into a hardware model. The result is still a six-module design, but the six modules of this design have different functions than the six modules of the design by Aguilar. The top-down approach worked exceedingly well to clarify the design and to minimize inter-module signal problems.

This research required a total of three academic quarters. The work during the first quarter primarily involved studying the problem, analyzing the design requirements, and learning about the PowerPC system. Two more quarters were required for the creation of the design, one quarter each for the behavioral design phase and the structural design phase.

Epoch and Verilog-XL proved reliable and highly useful during the development of this hardware design. Verilog-XL performed the simulations necessary to verify the design. Epoch performed the VLSI circuit layout and timing analysis that were required by Verilog-XL in order to produce simulation results that could be considered accurate.

Simulations with Verilog-XL are conveniently short while testing small modules. However, simulations of the entire PRC design typically ran for half an hour on a SUN SPARC-10 work station. Similarly, on small designs Epoch runs fast enough that a user could wait at the work station. To compile complex modules Epoch requires much more time. For example, Epoch takes over an hour to compile the Bus Interface of the PRC and more than three hours to compile the entire PRC.

Both Verilog-XL and Epoch have functions and options which are not readily apparent. That problem is compounded by inadequate indexes in the user's manuals for each of these tools. On the other hand, the tutorials are very helpful for revealing some of those functions and options.

Some of the options in Epoch require significant studying before use. The pull-down menus in Epoch could be better organized. Both of these characteristics work to make Epoch less user-friendly than it should be.

B. RECOMMENDATIONS

As with any complex design, there is much more that a designer could do to improve this PRC. This section describes some areas of potential future research related to this hardware design.

The first recommendation is to consider including the Arbiter on the PRC chip. This PRC design was developed for a PowerPC-603 microprocessor system, in which both the PRC and the CPU are candidates for bus mastership. This requires that there be a bus arbitration unit to prevent both devices from trying to use the bus simultaneously. The bus arbitration unit is a simple device whose function can be fulfilled with a single finite state machine (FSM). It would be very easy to add this FSM to the PRC chip, eliminating the requirement to fabricate a separate integrated circuit chip.

The second recommendation is in regards to improving the Line Manager design. The Line Manager is the block that requires the wait states in the Controller State Diagram. The impact of these wait states is a delay of three cycles in determining if there is a hit within the PRC. Finding a way of eliminating these wait states could improve the speed at which the PRC delivers the first beat of data to the CPU and the speed at which the PRC prefetches data from main memory. Specifically, the performance would improve from 8-1-1-1 to 5-1-1-1. There is a strong chance that Epoch would prove useful in this endeavor. Epoch has timing analysis routines and can perform layouts in such a way as to minimize propagation delays for critical signals. Epoch also has automatic buffer sizing algorithms which could be used to ensure the output signals of each part are buffered sufficiently to drive their

loads. These capabilities of Epoch do require considerable CPU time. For example, running an automatic compilation on the current design of the Bus Interface Unit takes over an hour of actual CPU time on a Sun SPARC 10 workstation if the buffer sizing option is selected.

The next recommendation is to study the rest of the design for critical paths. With Epoch as an analysis tool, it should be uncomplicated to analyze the entire PRC for critical timing paths. Some timing limitations may be improved through the buffer-sizing and timing-critical layout capabilities of Epoch. Other timing limitations may require modifying the design. The current PRC design includes only parts that were available in the Epoch library. It may be possible to design parts that outperform the Epoch parts.

The final recommendation regards fabrication. If the PRC design detailed in this thesis is to be fabricated, it must undergo two steps. First, the power rails should be studied using Epoch to determine if there is a requirement for additional power and ground rails. Second, the design must be put inside a pad ring. Epoch may be able to create the pad ring automatically with minimal intervention by the designer.

APPENDIX A. LAYOUTS

This appendix contains the VLSI (Very-Large-Scale-Integrated) circuit layouts for the PRC. These layouts were all generated by Epoch.

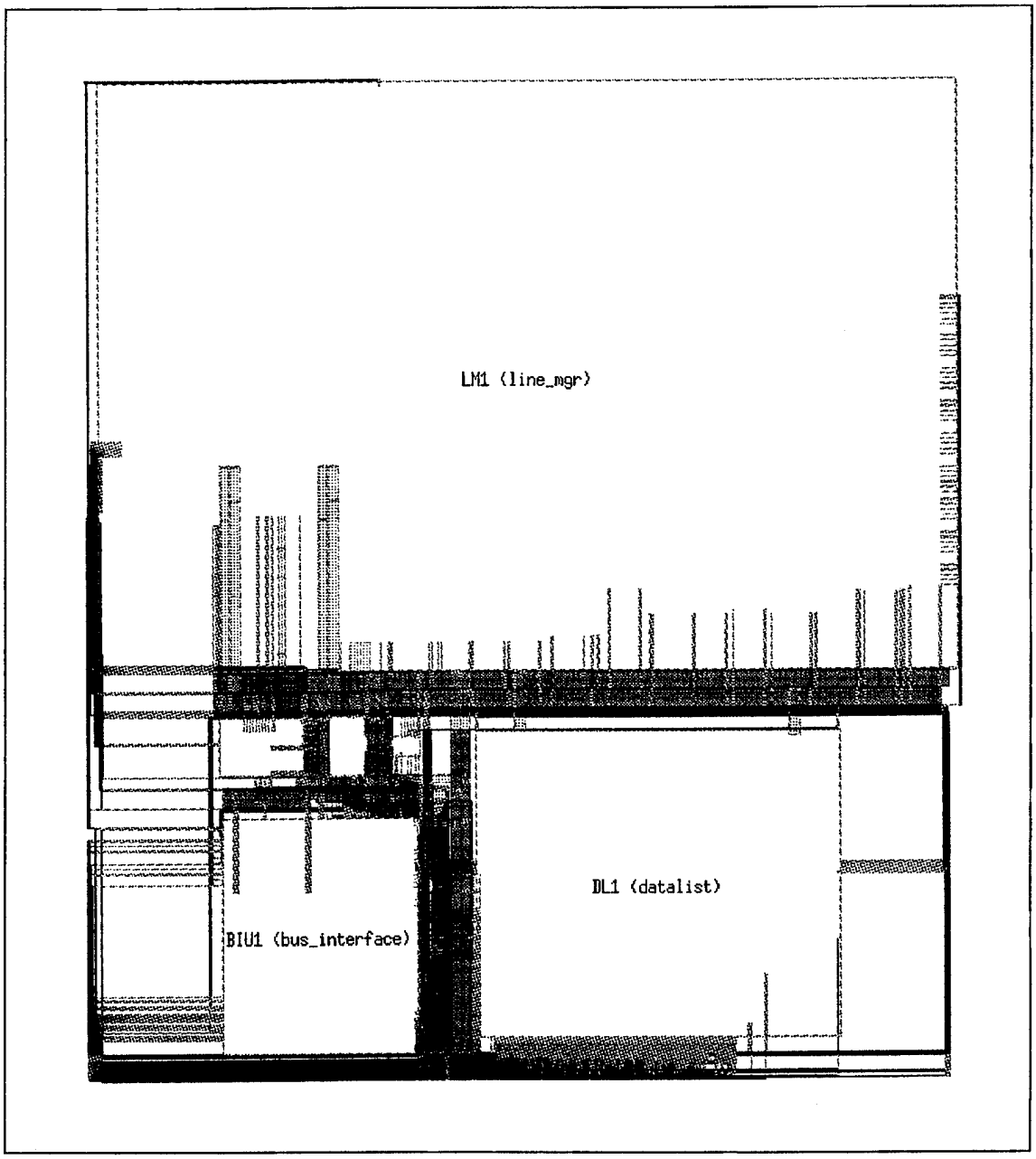


Figure A1. The PRC expanded to the first level. The four blocks in the lower left corner, in order of decreasing size, are the Bus Interface, Predictor, Snooper, and Controller. [Epoch output]

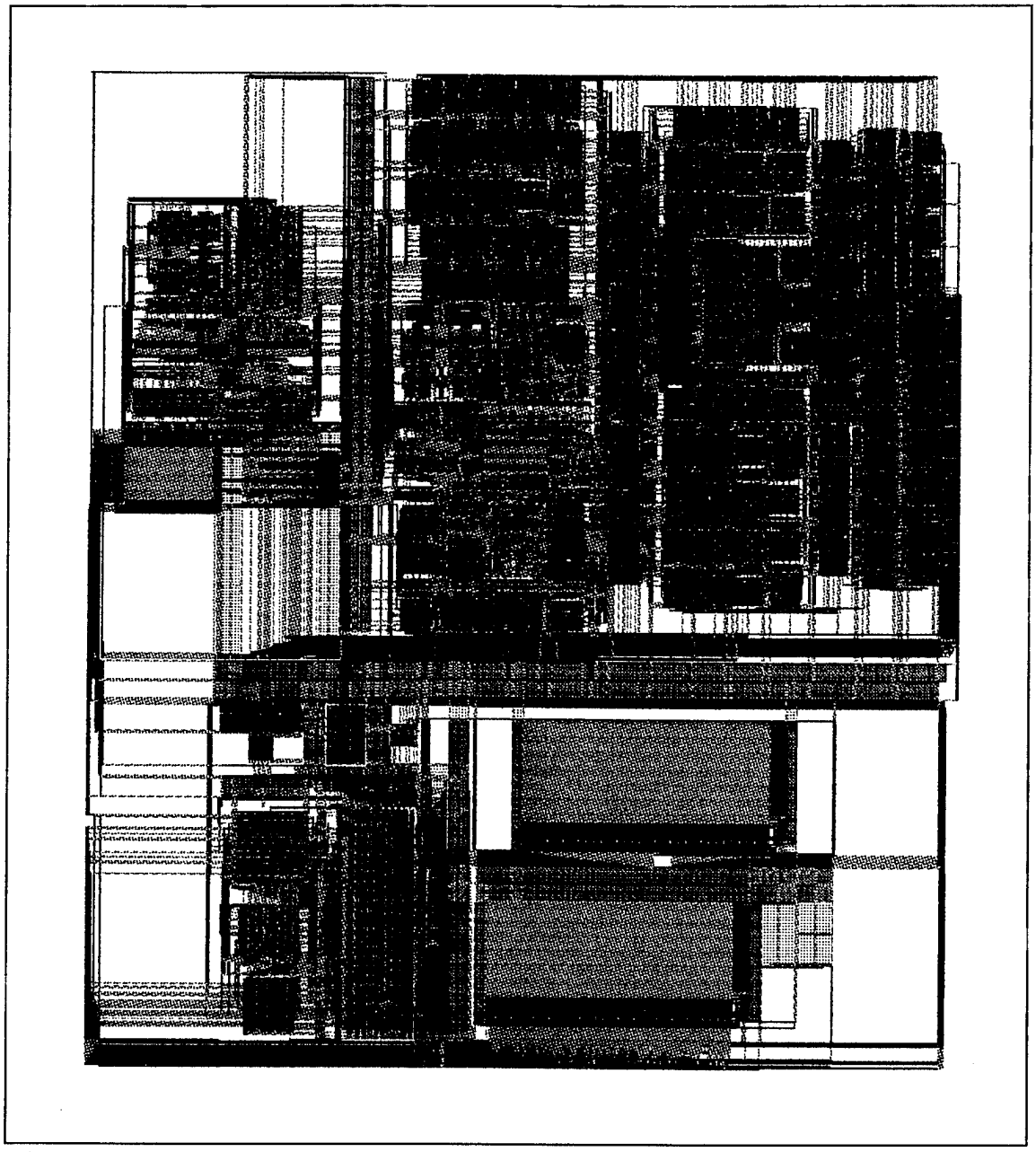


Figure A2. The PRC fully expanded. [Epoch output]

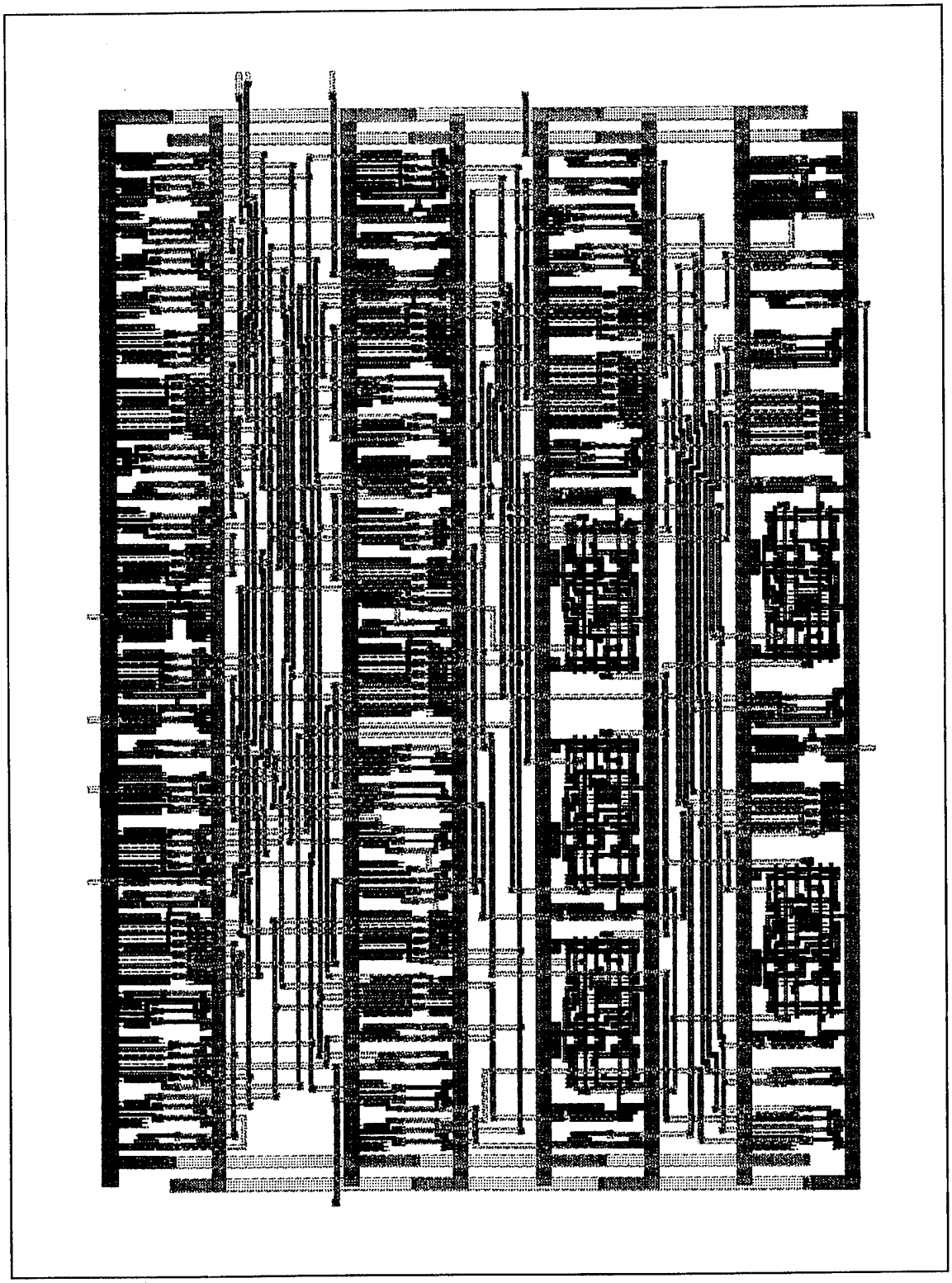


Figure A3. The Controller. [Epoch output]

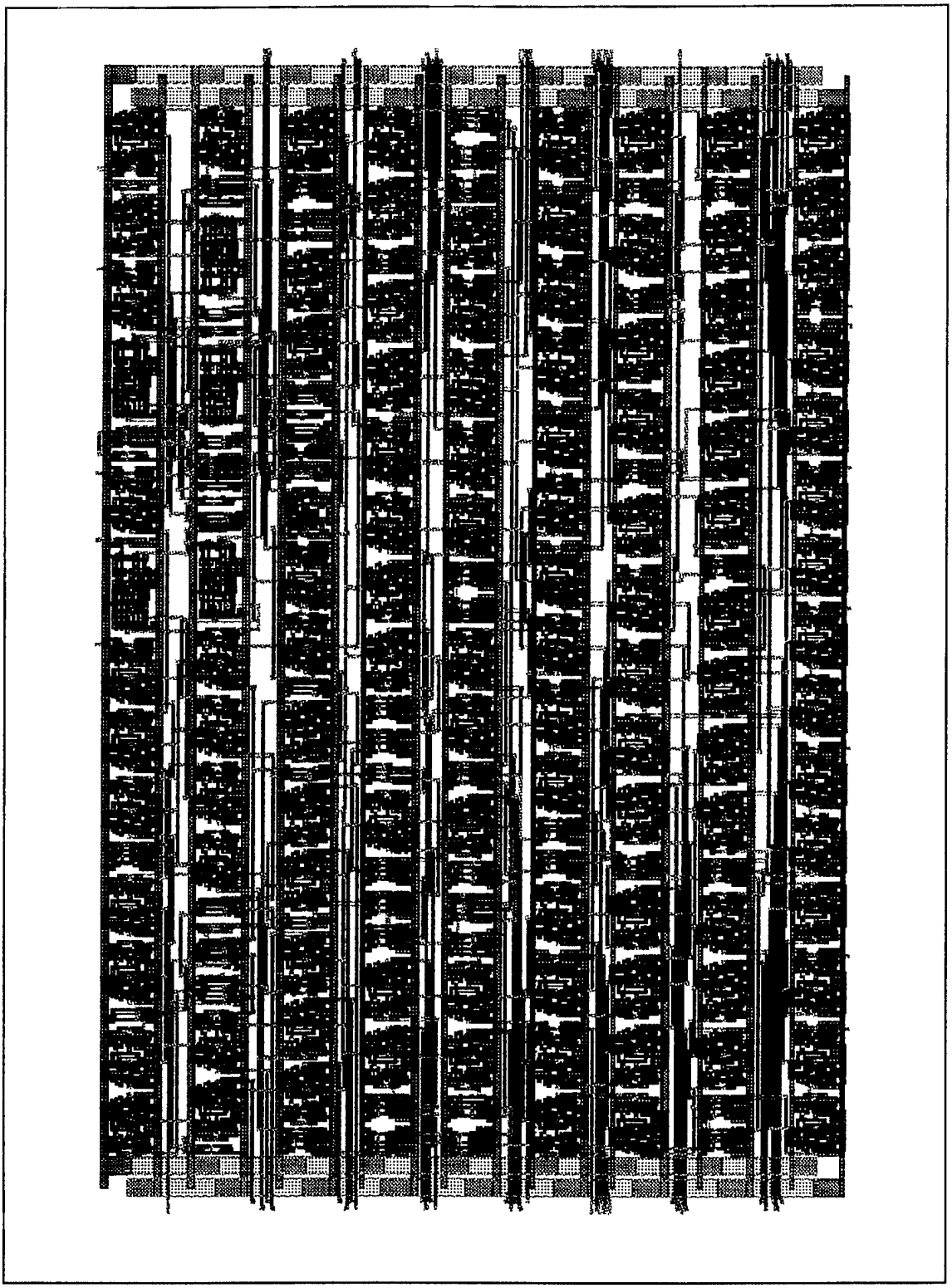


Figure A4. The Snooper. [Epoch output]

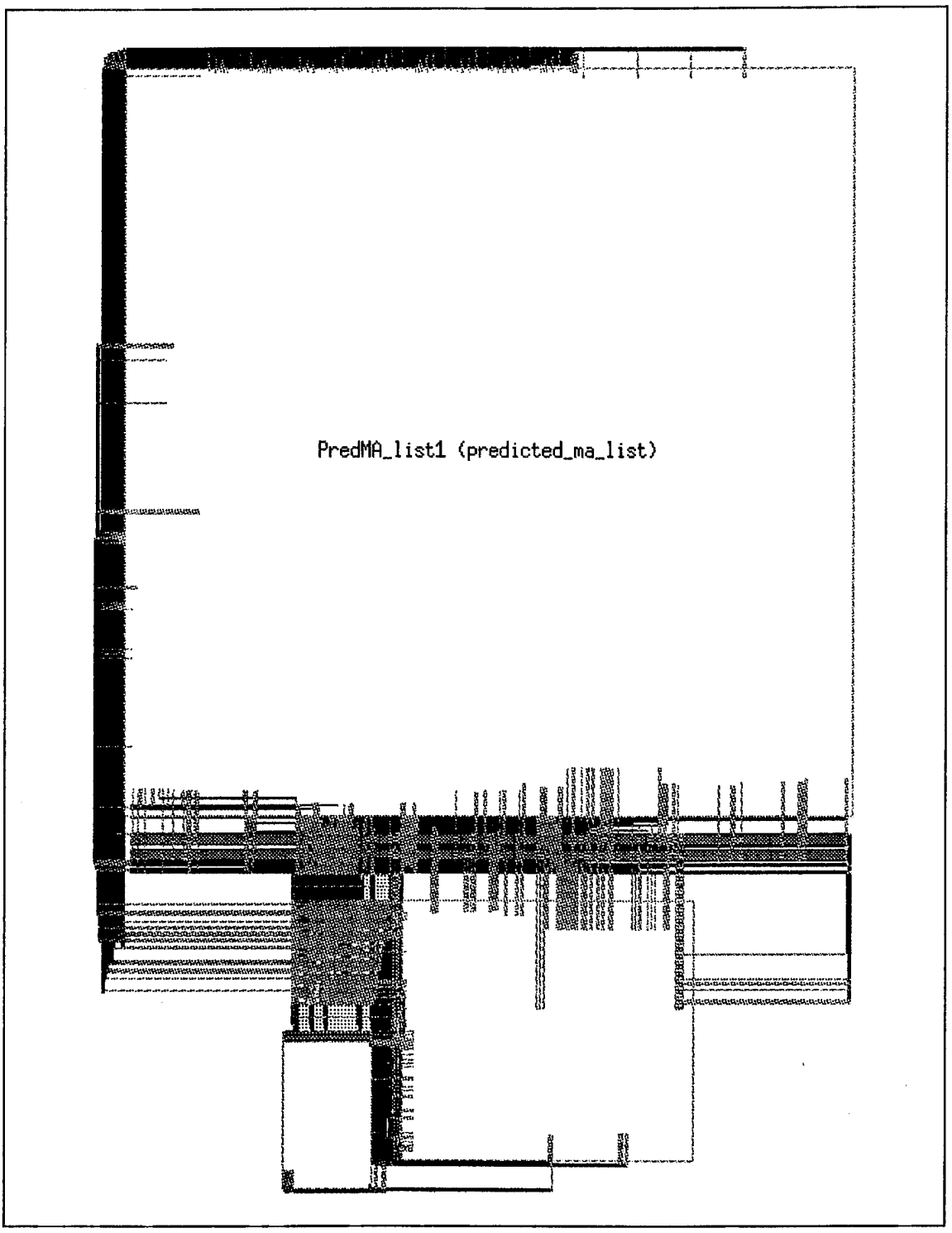


Figure A5. The Line Manager expanded one level. The bottom portion is shown in more detail in the next figure. [Epoch output]

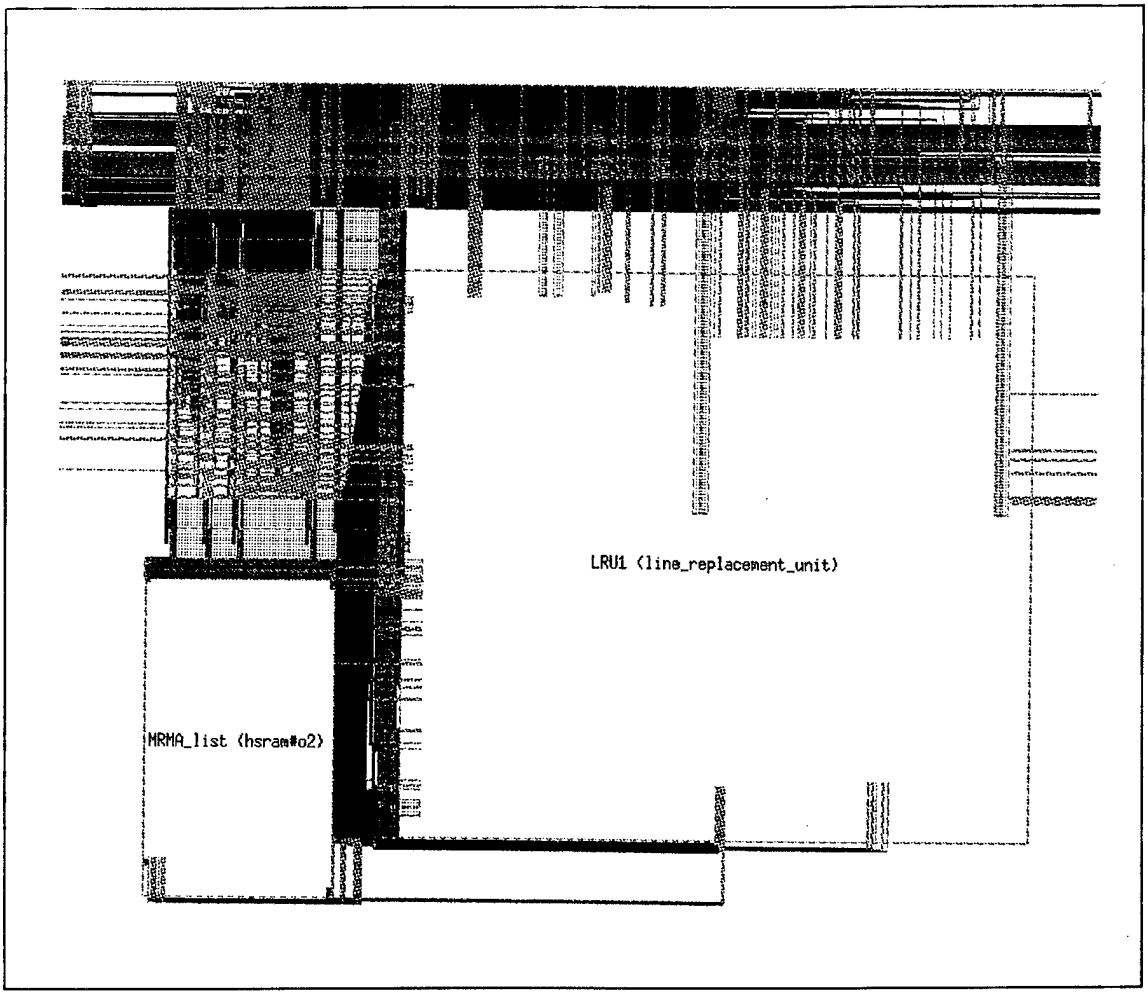


Figure A6. The Line Manager. Detail of the bottom portion in the previous figure. [Epoch output]

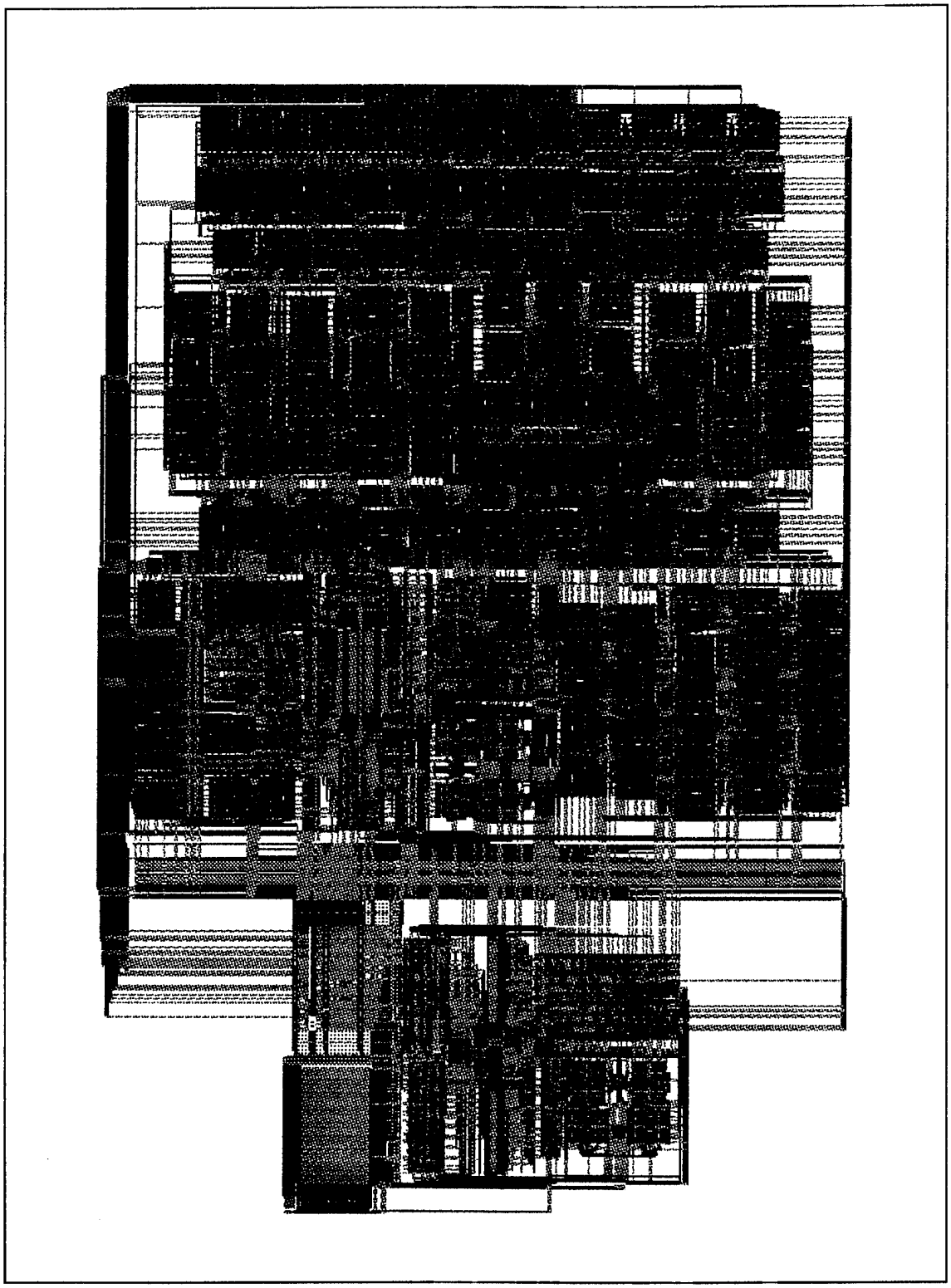


Figure A7. The Line Manager fully expanded. [Epoch output]

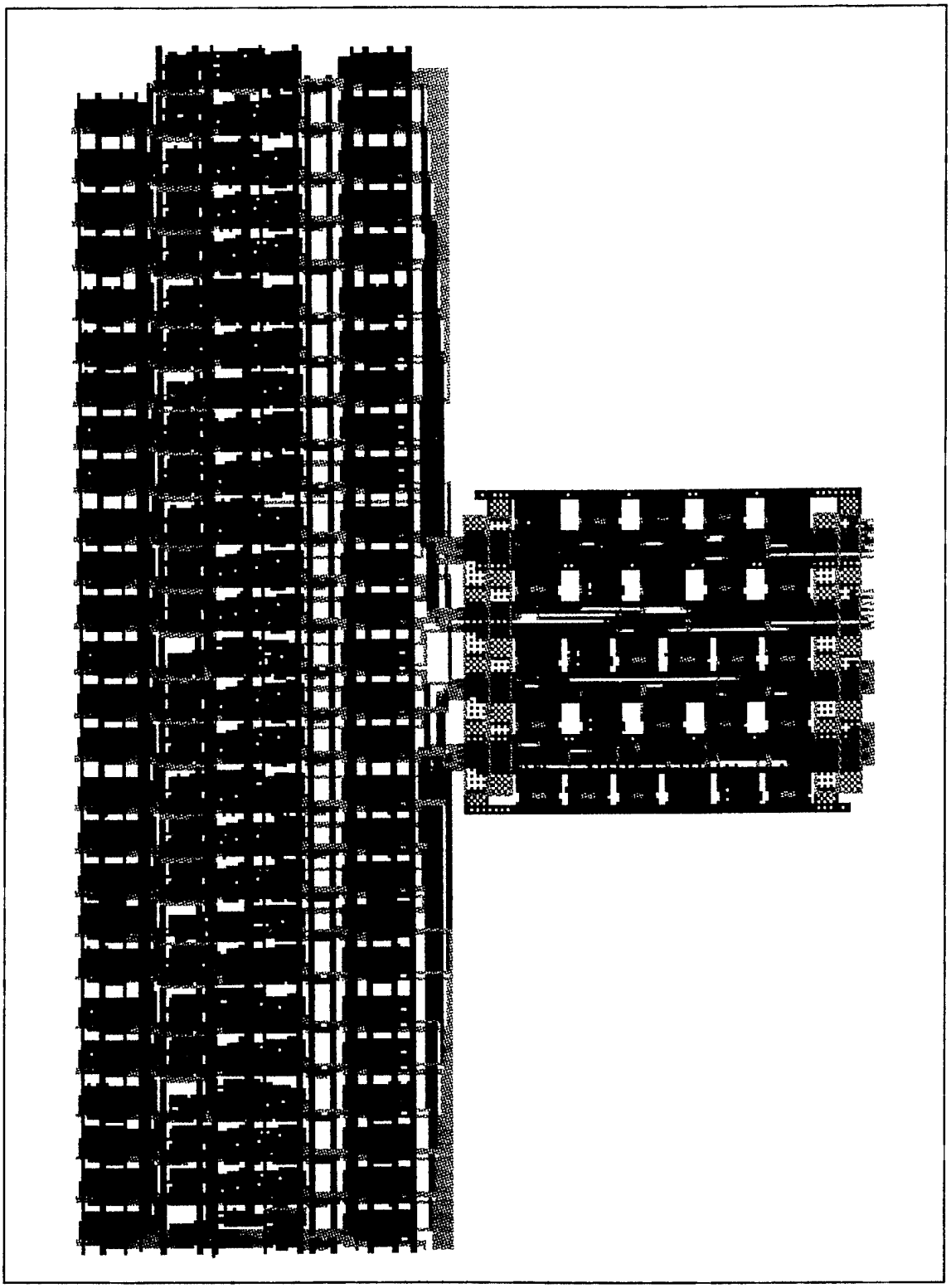


Figure A8. The Predictor fully expanded. [Epoch output]

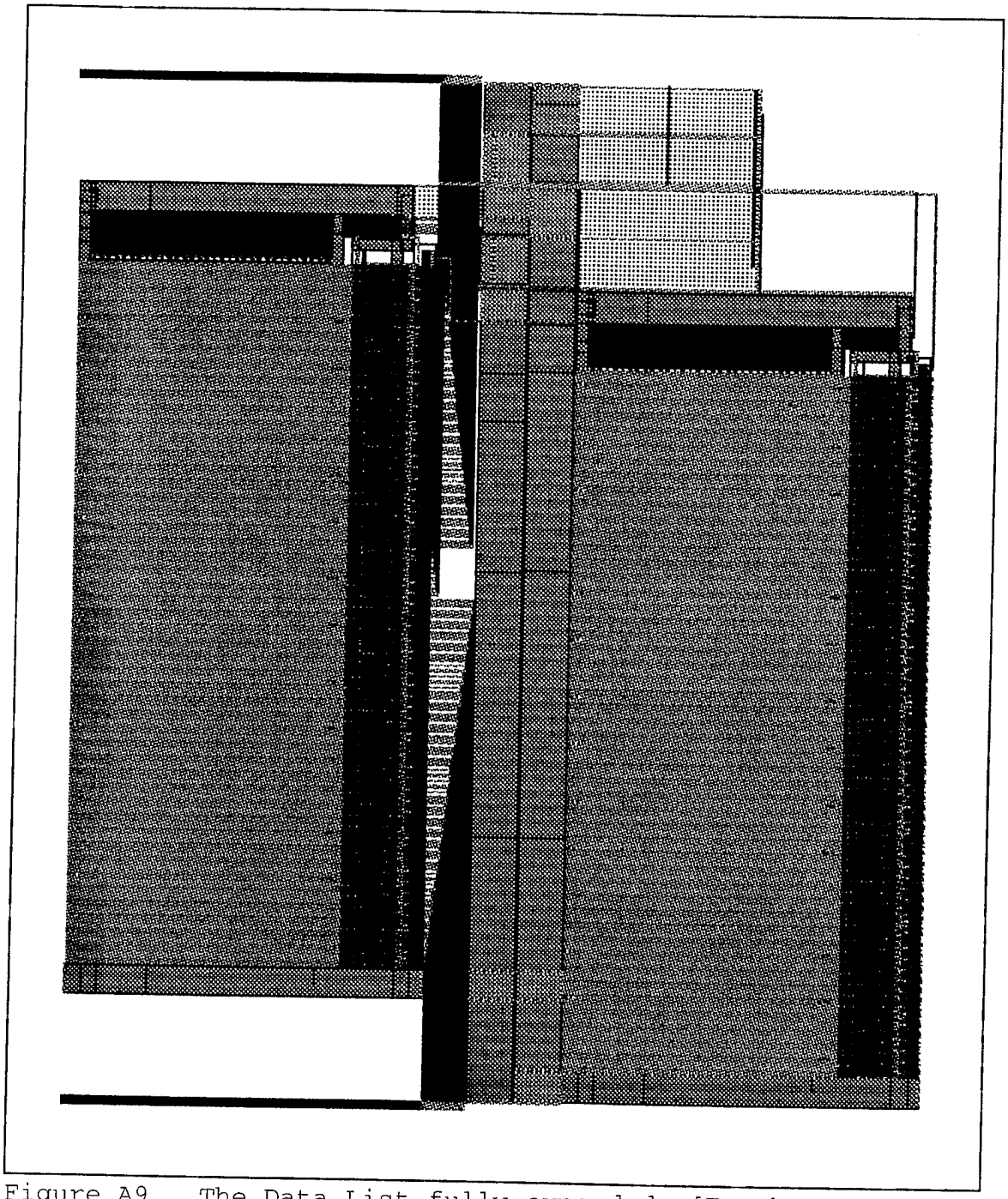


Figure A9. The Data List fully expanded. [Epoch output]

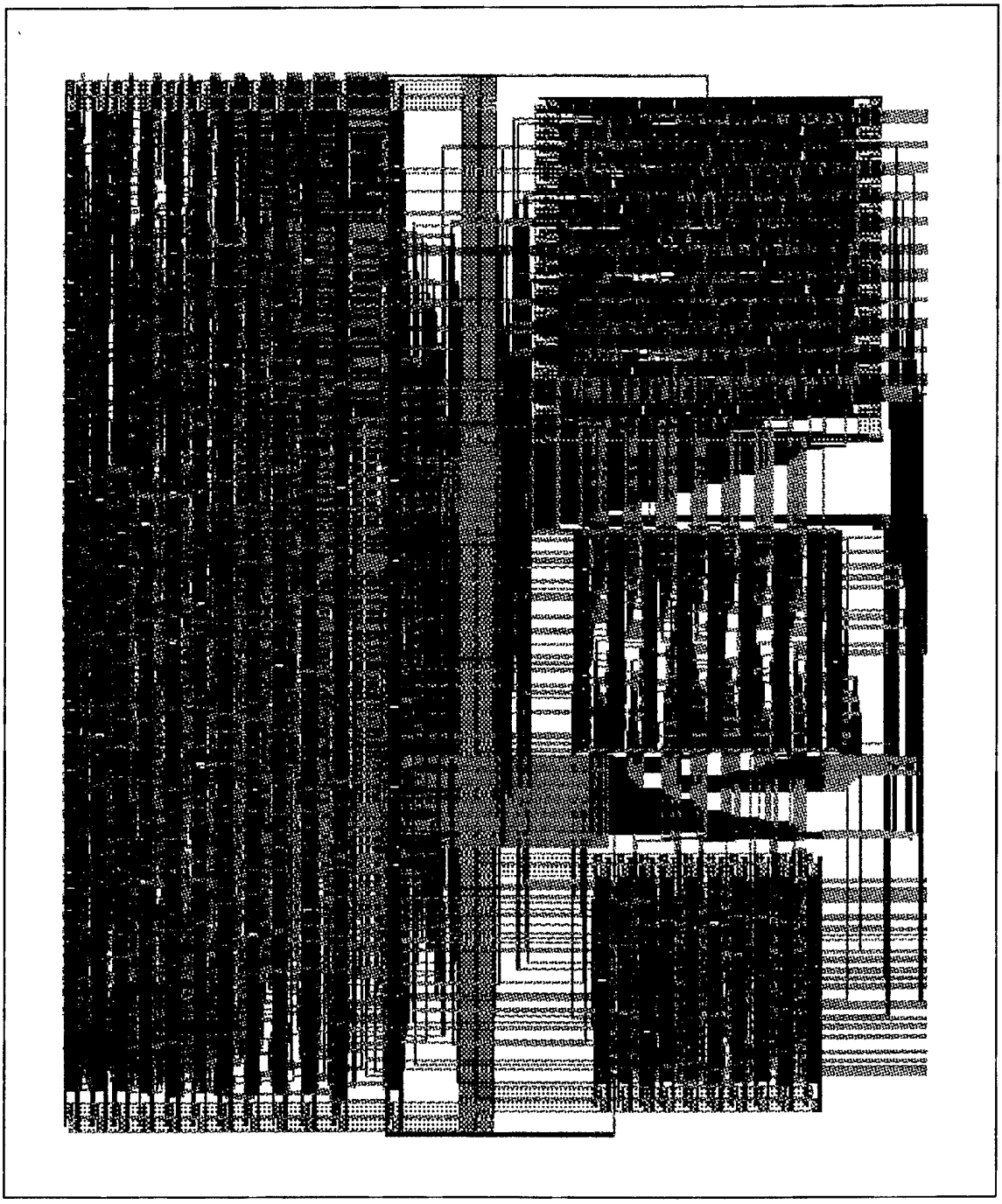


Figure A10. The Bus Interface fully expanded. [Epoch output]

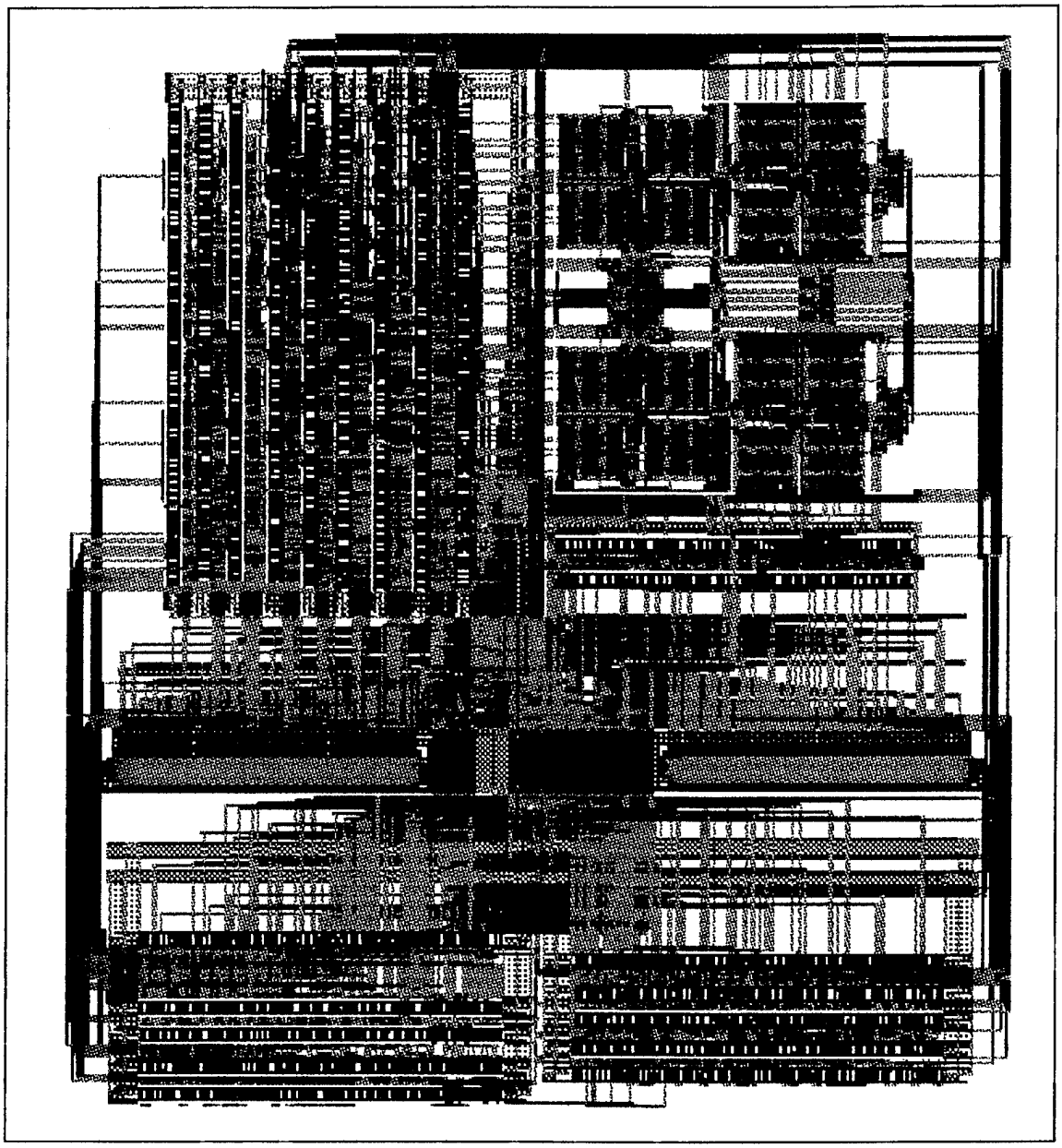


Figure A11. The Line Replacement Unit fully expanded. [Epoch output]

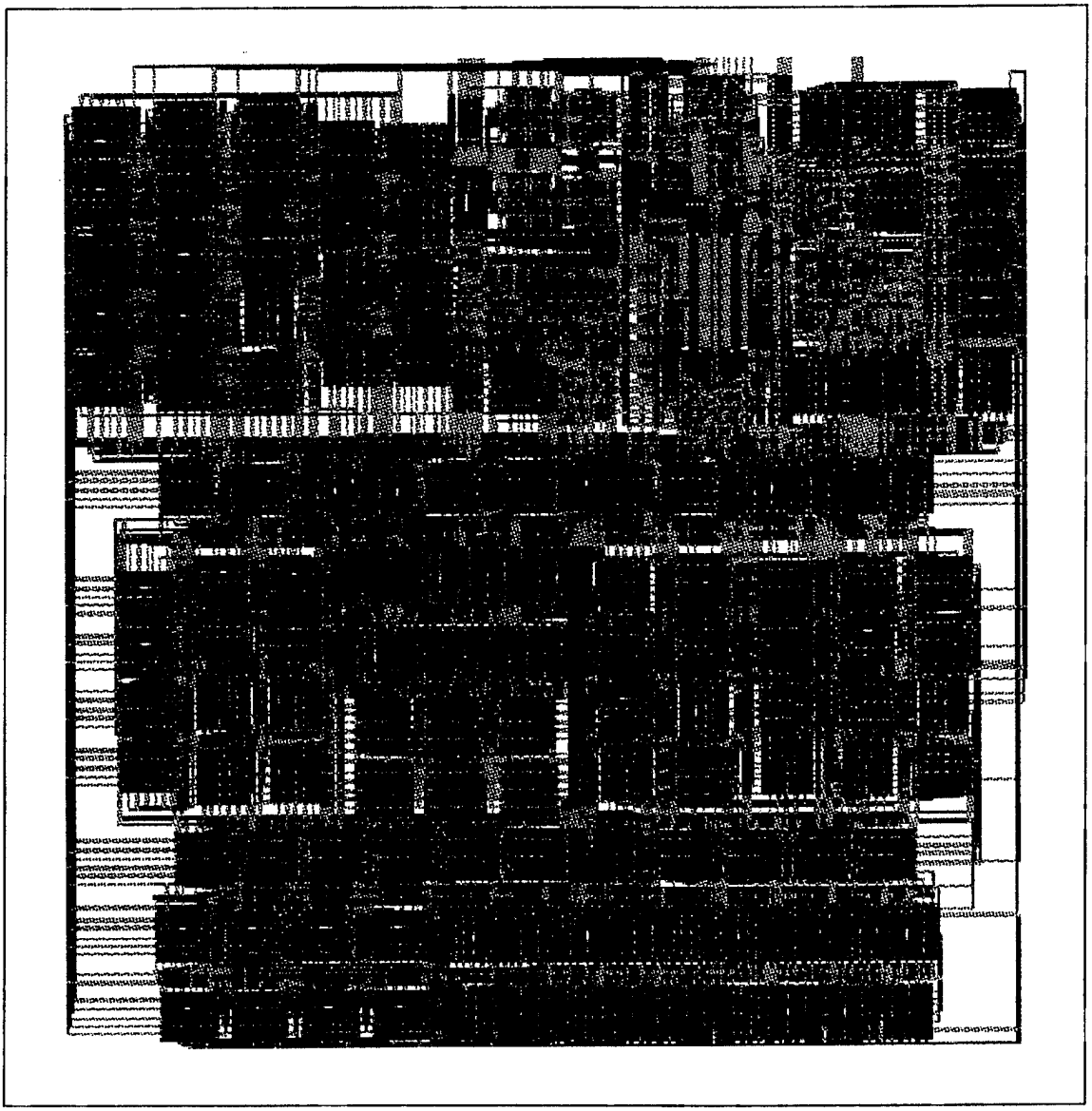


Figure A12. The Predicted Memory Address List fully expanded. [Epoch output]

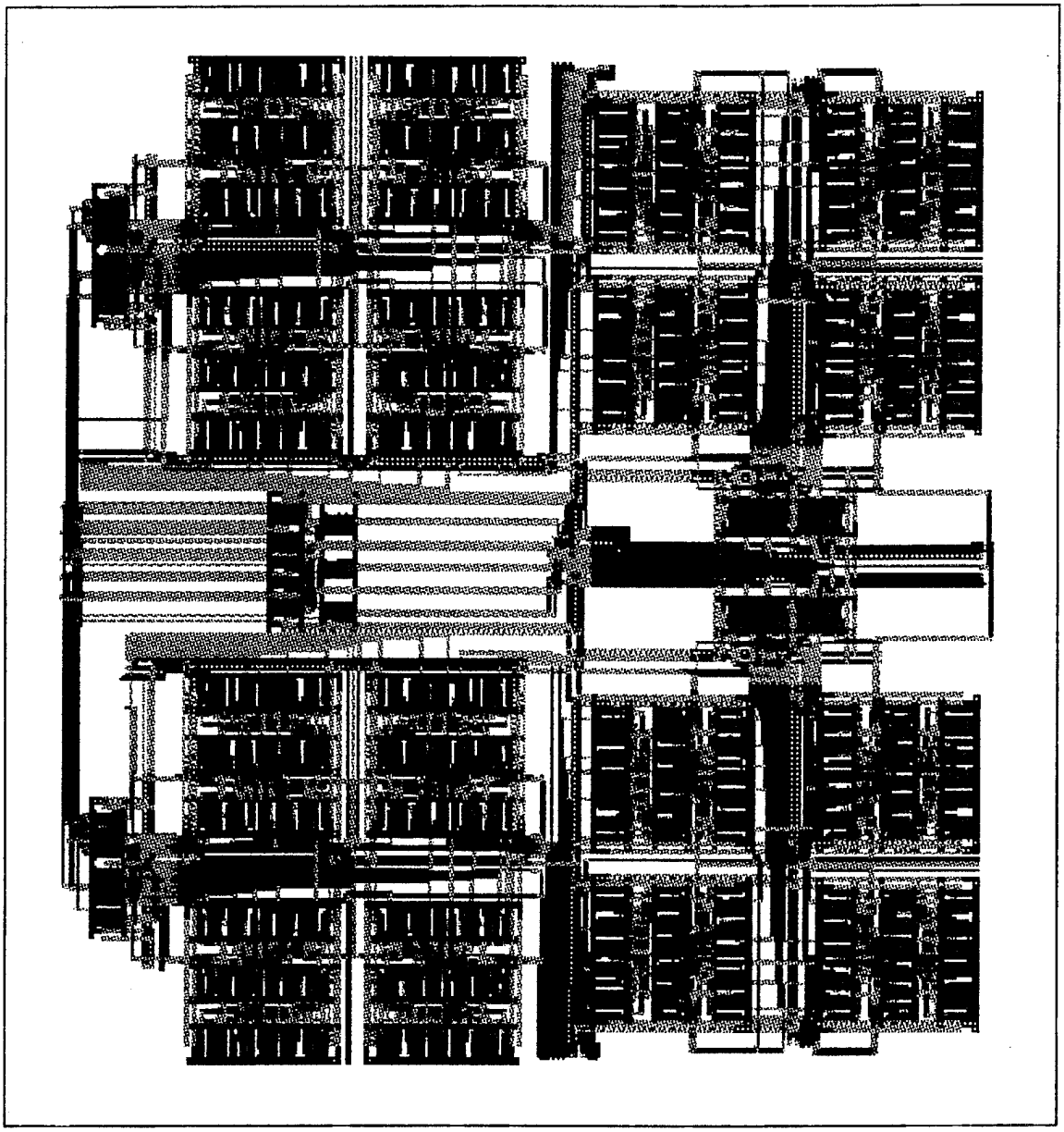


Figure A13. The 128-to-7 Priority Encoder fully expanded. [Epoch output]

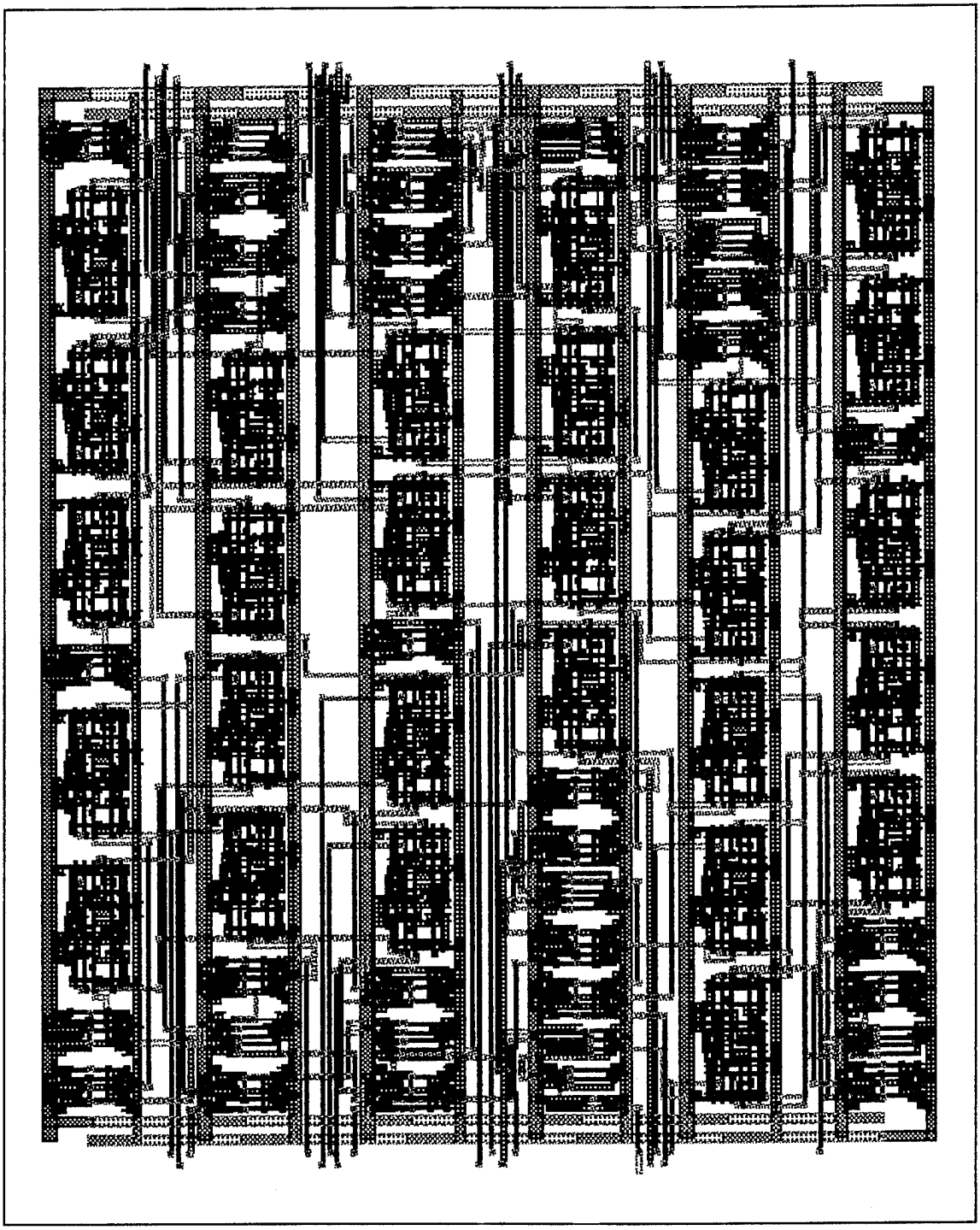


Figure A14. The Predicted Address Register fully expanded. [Epoch output]

APPENDIX B. TESTBENCH VERILOG FILES

This appendix contains the Verilog files for the Testbench. They are all behavioral models, used together to test the PRC design. The file are located on the Computer Center system at *joshua_u2/jrrobert/thesis/verilog/behavior*.

A. TESTBENCH

```
/*********************************************************************
* TESTBENCH
* Filename: testbench.v
* Author: Joseph R. Robert, Jr.
* Date:  24AUG95
* Revised: 10JAN96
*
* Purpose: This module is the highest level in the design hierarchy.  It
* emulates a complete computer system, composed of
*    1. cpu: a PowerPC-603 microprocessor.
*    2. ram: random access memory.
*    3. arbiter: the bus arbitration unit.
*    4. prc: the predictive read cache under design.
*
*    System configuration and features:
*      Single CPU
*      64-bit data bus
*      No out-of-order split-bus transactions.
*      Synchronous interface: all I/O sampled on rising edge of bus clock.
*      66 MHZ system clock, 66 MHZ CPU clock.
*
*    Simulation should be done with a time unit = 1 ns.
*
*********************************************************************/
module testbench;

// Signal Declarations - conforms to PowerPC-603 notation

  // Address Arbitration
  wire CPU_BR_,          //Bus Request
     CPU_BG_;            //Bus Grant
  tri1 ABB_;             //Address Bus Busy
```

```
  tri1 TS_;                  //Transfer Start (memory only, not I/O)

  // Address bus
  wire [0:31] A;  //Address (note Motorola's reverse notation)
  wire [0:3] AP;  //Address Parity
  wire APE_;      //Address Parity Error

  // Transfer attributes
  wire [0:4] TT;             //Transfer Type
  wire [0:2] TSIZ;           //Transfer Size
  wire [0:1] TC;             //Transfer Code
  tri1 TBST_;     //Transfer burst
  wire GBL_,
     CI_,
     WT_,
     CSE;

  // Address Termination
  tri1 AACK_;     //Address Acknowledge
  reg  ARTRY_;               //Address Retry

  // Data Arbitration
  wire CPU_DBG_;             //Data Bus Grant
  reg  DBWO_;     //Data Bus Write Only
  tri1 DBB_;      //Data Bus Busy

  // Data Transfer
  wire [0:63] D;             //Data
  wire [0:7] DP;             //Data Parity
  wire DPE_,                 //Data Parity Error
     DBDIS_;     //Data Bus Disable

  // Data Termination
  tri1 TA_;       //Transfer Acknowledge
  reg  DRTRY_;               //Data Retry
  reg  TEA_;      //Transfer Error Acknowledge

  // System control
  reg  HRESET_; //Hard Reset
  wire PRC_BR_;      //PRC Bus Request
  wire CANX;

//Declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0;

//Initialize values.
initial
```

```
  begin
    DBWO_ = hi;          //Limits CPU to in-order transactions.
    TEA_  = hi;  //Only asserted for nonrecoverable bus error events.
    ARTRY_ = hi;         //Retries used only with multiprocessor or multi-
    DRTRY_ = hi;         //  level memory systems.
    HRESET_= hi;
  end


//define system clock, 66 MHz, T = 15 ns.
  reg clk;
  initial clk = 1;
  always
    begin
      #7 clk = 0;
      #8 clk = 1;
    end


//Connect parts
  cpu    CPU1(CPU_BR_,CPU_BG_,ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,TBST_,GBL_,
           CI_,WT_,CSE,AACK_,ARTRY_,CPU_DBG_,DBWO_,DBB_,D,DP,
           DPE_,DBDIS_,TA_,DRTRY_,TEA_,clk);
  memory  MEM1(ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,TBST_,GBL_,CI_,WT_,CSE,AACK_,
           DBWO_,DBB_,D,DP,DPE_,DBDIS_,TA_,TEA_,CANX,clk);
  arbiter ARB1(CPU_BR_,CPU_BG_,CPU_DBG_,PRC_BR_,PRC_BG_,PRC_DBG_,
           ABB_,DBB_,clk);
  prc    PRC1(CPU_BR_,PRC_BR_,PRC_BG_,ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,
           TBST_,AACK_,PRC_DBG_,DBB_,D,DP,DPE_,TA_,HRESET_,CANX,clk);


//run simulation
initial
  begin
    //$shm_open;
    #5 HRESET_ = low;    //Reset entire system.
    #5 HRESET_ = hi;
    //#4000;
    //$shm_probe(PRC1,"AS");
    #152000 $finish;     //Adjust this time according to the instructions
                         //in the sequencers.
  end

endmodule
```

B. CPU

```
/*********************************************************************************
* PowerPC-603 CPU
* Filename: cpu.v
* Author: Joseph R. Robert, Jr.
* Date:  24AUG95
* Revised: 10JAN96
*
* Purpose: This module emulates the PowerPC-603 microprocessor. Note that
* most signals are active low. This makes it slightly more difficult to work
* one's way through all the double negatives in this code's conditional
* statements, but makes it much easier to correlate against the timing diagrams
* in the PowerPC-603 User Manual. This model uses the same notations for
* signals that connect to other modules.
*  This module uses the sequencer module to determine the operations the CPU
* will perform. This model of the PowerPC-603 is capable of performing reads,
* writes, burst reads, and burst writes. It handles bus arbitration just like
* the '603 including the pipelined address tenures. Please refer to the
* PowerPC-603 User Manual for a detailed description of the nature and timing
* of each signal.
*
*********************************************************************************/
module cpu (BR_,BG_,ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,TBST_,GBL_,CI_,WT_,CSE,AACK_,
        ARTRY_,DBG_,DBWO_,DBB_,D,DP,DPE_,DBDIS_,TA_,DRTRY_,TEA_,clk);

// Signals are defined in system.v.
input  BG_,AACK_,DBG_,DBWO_,DBDIS_,TA_,ARTRY_,DRTRY_,TEA_,clk;
output BR_,APE_,CI_,WT_,CSE,DPE_;
inout  [0:31] A;
inout  [0:63] D;
inout  [0:7] DP;
inout  [0:4] TT;
inout  [0:3] AP;
inout  [0:2] TSIZ;
inout  [0:1] TC;
inout  ABB_,TS_,TBST_,GBL_,DBB_;

reg  BR_,APE_,CI_,WT_,CSE,DPE_;
tri  [0:31] A;
tri  [0:63] D;
tri  [0:7] DP;
tri  [0:4] TT;
tri  [0:3] AP;
tri  [0:2] TSIZ;
tri  [0:1] TC;
tri  ABB_,TS_,TBST_,GBL_,DBB_;
```

```
//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
            FALSE = 1'b0,
            hi    = 1'b1,
            low   = 1'b0,
            trace = FALSE;

  //Address related
  wire [0:31] seq_addr;
  reg [0:31] addr_reg, address[0:1];
  reg [0:31] a_reg;
    assign A = a_reg;
  reg [0:3] ap_reg, addr_parity_in, addr_parity_calc;
    assign AP = ap_reg;

  //Data related
  reg [0:63] data [0:1];
  wire [0:63] seq_data;
  reg [0:63] d_reg, load_data, data_reg;
    assign D = d_reg;
  reg [0:255] line_reg, line [0:1];
  wire [0:255] seq_line;
  reg [0:7] dp_reg, d_parity_in, d_parity_calc;
    assign DP = dp_reg;

  //Other external control signals
  reg Transfer_start [0:1];
  reg abb_reg_, dbb_reg_, ts_reg_, tbst_reg_;
    assign ABB_  = abb_reg_;
    assign TS_   = ts_reg_;
    assign DBB_  = dbb_reg_;
    assign TBST_ = tbst_reg_;

  reg  [0:4] Transfer_type [0:1];
  wire [0:4] seq_Transfer_type;
  reg  [0:4] tt_reg;
    assign TT = tt_reg;
    parameter //for Transfer_type
          none       = 5'bz,
          write      = 5'b00010,      //02
          write_atomic = 5'b10010,    //12
          read       = 5'b01010,      //0A
          read_atomic  = 5'b11010,    //1A
          burst_write  = 5'b00110,    //06
          burst_read   = 5'b01110,    //0E
          burst_read_atomic = 5'b11110;  //1E

  reg  [0:2] Transfer_size [0:1];
  wire [0:2] seq_Transfer_size;
  reg  [0:2] tsiz_reg;
```

```
    assign TSIZ = tsiz_reg;

  reg  [0:1] Transfer_code [0:1];
  wire [0:1] seq_Transfer_code;
  reg  [0:1] tc_reg;
    assign TC = tc_reg;
    parameter //for Transfer_code
      data_transfer    = 2'b00,
      touch_load       = 2'b01,
      instruction_fetch = 2'b10,
      reserved         = 2'b11;


  //Other internal control signals
  reg need_bus_;
  wire need_bus_trigger_;
  reg AB_Master,DB_Master, Addr_termination;
  wire qual_BG_,qual_DBG_;
  reg [0:7] index;
  wire parked;
  wire pp;
  reg dpp;
  event transfer_acknowledged;


//initialize signals
initial
  begin
    a_reg     <= 32'bz;
    ap_reg    <= 4'bz;
    addr_parity_in <= 4'bz;
    addr_parity_calc <= 4'bz;
    addr_reg  <= 32'bz;
    address[0] <= 32'bz;
    address[1] <= 32'bz;

    data[0] <= 64'bz;
    data[1] <= 64'bz;
    d_reg   <= 64'bz;
    line[0] <= 256'bz;
    line[1] <= 256'bz;
    line_reg <= 256'bz;
    d_parity_in  <= 8'bz;
    d_parity_calc <= 8'bz;
    dp_reg  <= 8'bz;

    APE_ <= 'bz;
    BR_ <= hi;
    CI_ <= hi;
    CSE <= low;
```

```
    DPE_ <= 'bz;
    WT_  <= hi;
    abb_reg_  <= 'bz;
    dbb_reg_  <= 'bz;
    ts_reg_   <= 'bz;
    tbst_reg_ <= 'bz;
    Transfer_type[0] <= none;
    Transfer_type[1] <= none;
    tt_reg <= none;
    Transfer_size[0] <= 0;
    Transfer_size[1] <= 0;
    tsiz_reg <= 'bz;
    Transfer_code[0] <= reserved;
    Transfer_code[1] <= reserved;
    tc_reg <= 2'bz;
    Transfer_start[0] <= FALSE;
    Transfer_start[1] <= FALSE;

    AB_Master <= FALSE;
    DB_Master <= FALSE;
    Addr_termination <= FALSE;
    need_bus_ <= hi;
    dpp <= 0;
  end

//

sequencer SEQ1(seq_Transfer_size,clk,pp,seq_addr,seq_data,seq_line,
        seq_Transfer_type,seq_Transfer_code,need_bus_trigger_,ABB_ );

always @(negedge need_bus_trigger_)
  begin
    address[pp] <= seq_addr;
    data[pp]    <= seq_data;
    line[pp]    <= seq_line;
    Transfer_type[pp] <= seq_Transfer_type;
    Transfer_size[pp] <= seq_Transfer_size;
    Transfer_code[pp] <= seq_Transfer_code;
  end


//
//ADDRESS BUS TENURE

  // *** 1. Address bus arbitration

always @(negedge need_bus_trigger_)
  need_bus_ = low;

  //Parked means that the CPU can take the bus as soon as it needs it.
```

```
assign parked = (!BG_ & ABB_ & ARTRY_);

//If CPU needs bus, it needs to assert BR_ only if not parked.
always @(posedge clk)
  if (BR_ == hi)
    BR_ = #7 ~(need_bus_==low & parked==FALSE);

assign qual_BG_ = ~(need_bus_==low & parked==TRUE);

//Assume mastership
always @(posedge clk)
  if (qual_BG_ == low)
    begin
      abb_reg_ = #7 low;
      AB_Master = TRUE;
      BR_ <= #1 hi;
      need_bus_ <= #2 hi;
    end

// *** 2. Address Transfer

always @(posedge clk)
  if (qual_BG_ == low)
    begin
      addr_reg = address[pp];
      addr_parity_calc[0] <= ~^addr_reg[0:7];
      addr_parity_calc[1] <= ~^addr_reg[8:15];
      addr_parity_calc[2] <= ~^addr_reg[16:23];
      addr_parity_calc[3] <= ~^addr_reg[24:31];
      ts_reg_ = #7 low;
      Transfer_start[pp] <= TRUE;
      a_reg   <= address[pp];
      ap_reg  <= addr_parity_calc;
      tt_reg  <= Transfer_type[pp];
      tsiz_reg <= Transfer_size[pp];
      tc_reg  <= Transfer_code[pp];
      if (Transfer_type[pp] == burst_read
        || Transfer_type[pp] == burst_write)
        tbst_reg_ <= low;
      //insert other address transfer characteristics here.
    end

always @(posedge clk)
  if (AB_Master & TS_==low)
    begin
      ts_reg_ = #7 hi;
      wait (AACK_==low);
      Addr_termination = TRUE;
    end
```

```
  always @(posedge clk)
    if (Addr_termination)
      begin
        #7 ts_reg_ <= 'bz;
        a_reg     <= 'bz;
        ap_reg    <= 'bz;
        tt_reg    <= 'bz;
        tc_reg    <= 'bz;
        tsiz_reg  <= 'bz;
        tbst_reg_  <= 'bz;
        //insert other addr transfer characteristics here.
        abb_reg_ <= #2 hi;
        abb_reg_ <= #8 'bz;
        AB_Master = FALSE;
        Addr_termination = FALSE;
      end
//
//DATA BUS TENURE

  assign qual_DBG_ = ~(!DBG_ & DBB_ & DRTRY_);

  always @(posedge clk)
    begin
      if (TA_ == low)
        -> transfer_acknowledged;
    end

  always
    begin
      #2 dpp = ~dpp;
      case(Transfer_type[dpp])
        none: begin end

//Note: TS is an implied data bus request.  CPU can assume mastership if it
//has a qualified data bus grant.

        read: begin
              //wait for qualified data bus grant and transfer start.
              wait(qual_DBG_==low & Transfer_start[dpp]);
              @(posedge clk) //assume data bus mastership
              dbb_reg_ <= #7 low;
              @(transfer_acknowledged) //latch data and terminate read
              data[dpp] <= D;
              data_reg  <= D;
              d_parity_in <= DP;
              Transfer_type[dpp] <= none;
              Transfer_code[dpp] <= reserved;
              Transfer_start[dpp] = FALSE;
              d_parity_calc[0] <= ~^data_reg[0:7];
              d_parity_calc[1] <= ~^data_reg[8:15];
```

```
        d_parity_calc[2] <= ~^data_reg[16:23];
        d_parity_calc[3] <= ~^data_reg[24:31];
        d_parity_calc[4] <= ~^data_reg[32:39];
        d_parity_calc[5] <= ~^data_reg[40:47];
        d_parity_calc[6] <= ~^data_reg[48:55];
        d_parity_calc[7] <= ~^data_reg[56:61];
        if (trace) begin
         $display("CPU read %h from address %h.",
               data[dpp],address[dpp]);
         $display("   Completed at time %d",$time);
         end
        dbb_reg_ = #4 hi;
        dbb_reg_ = #8 'bz;
        if (d_parity_in != d_parity_calc)
          begin
            $display("CPU: data parity error.");
            $display("   Calculated parity: %b",
                  d_parity_calc);
            $display("   Recevied parity:   %b",
                  d_parity_in);
          end
      end

write: begin
        data_reg = data[dpp];
        d_parity_calc[0] <= ~^data_reg[0:7];
        d_parity_calc[1] <= ~^data_reg[8:15];
        d_parity_calc[2] <= ~^data_reg[16:23];
        d_parity_calc[3] <= ~^data_reg[24:31];
        d_parity_calc[4] <= ~^data_reg[32:39];
        d_parity_calc[5] <= ~^data_reg[40:47];
        d_parity_calc[6] <= ~^data_reg[48:55];
        d_parity_calc[7] <= ~^data_reg[56:61];
        //wait for qualified data bus grant and transfer start.
        wait(qual_DBG_==low & Transfer_start[dpp]);
        @(posedge clk) //assume data bus mastership
        dbb_reg_ = #7 low;
        d_reg <= data[dpp];
        dp_reg <= d_parity_calc;
        @(transfer_acknowledged) //terminate write
        d_reg <= #7 64'bz;
        dp_reg <= #7 8'bz;
        Transfer_type[dpp] <= none;
        Transfer_start[dpp] = FALSE;
        if (trace) begin
         $display("CPU wrote %h to address %h.",
               data[dpp],address[dpp]);
         $display("   Completed at time %d",$time);
         end
        dbb_reg_ = #4 hi;
```

```
        dbb_reg_ = #8 'bz;
      end

 burst_read: begin
      //wait for qualified data bus grant and transfer start.
      wait(qual_DBG_==low & Transfer_start[dpp]);
      @(posedge clk) //assume data bus mastership
      dbb_reg_ <= #7 low;

      if (trace)
       $display("CPU started read from address %h at time %d.",
            address[dpp],$time);
      repeat (4) begin
       @(transfer_acknowledged) //latch beat
       data[dpp] <= D;
       data_reg <= D;
       d_parity_in = DP;
       #1 if (trace)
        $display("    CPU read: %h at %d",data[dpp],$time);
       d_parity_calc[0] <= ~^data_reg[0:7];
       d_parity_calc[1] <= ~^data_reg[8:15];
       d_parity_calc[2] <= ~^data_reg[16:23];
       d_parity_calc[3] <= ~^data_reg[24:31];
       d_parity_calc[4] <= ~^data_reg[32:39];
       d_parity_calc[5] <= ~^data_reg[40:47];
       d_parity_calc[6] <= ~^data_reg[48:55];
       d_parity_calc[7] = ~^data_reg[56:61];
       #2 if (d_parity_in != d_parity_calc)
         begin
           $display("CPU: data parity error.");
           $display("  Calculated parity: %b",
                d_parity_calc);
           $display("  Recevied parity:  %b",
                d_parity_in);
         end
      end

      Transfer_type[dpp] <= none;
      Transfer_code[dpp] <= reserved;
      Transfer_start[dpp] <= FALSE;
      dbb_reg_ = #4 hi;
      dbb_reg_ = #8 'bz;
    end

burst_write: begin
      //wait for qualified data bus grant and transfer start.
      wait(qual_DBG_==low & Transfer_start[dpp]);
      if (trace)
       $display("CPU started write to address %h at time %d.",
            address[dpp],$time);
```

```
@(posedge clk) //assume data bus mastership
dbb_reg_ = #6 low;
line_reg = line[dpp];
data_reg = line_reg[0:63];
d_parity_calc[0] <= ~^data_reg[0:7];
d_parity_calc[1] <= ~^data_reg[8:15];
d_parity_calc[2] <= ~^data_reg[16:23];
d_parity_calc[3] <= ~^data_reg[24:31];
d_parity_calc[4] <= ~^data_reg[32:39];
d_parity_calc[5] <= ~^data_reg[40:47];
d_parity_calc[6] <= ~^data_reg[48:55];
d_parity_calc[7] = ~^data_reg[56:61];
dp_reg <= d_parity_calc;
d_reg = line_reg[0:63];
#1 if (trace)
  $display("    CPU write beat 1: %h at %d",d_reg,$time);
@(transfer_acknowledged);     //first beat done

data_reg = line_reg[64:127];
d_parity_calc[0] <= ~^data_reg[0:7];
d_parity_calc[1] <= ~^data_reg[8:15];
d_parity_calc[2] <= ~^data_reg[16:23];
d_parity_calc[3] <= ~^data_reg[24:31];
d_parity_calc[4] <= ~^data_reg[32:39];
d_parity_calc[5] <= ~^data_reg[40:47];
d_parity_calc[6] <= ~^data_reg[48:55];
d_parity_calc[7] = ~^data_reg[56:61];
dp_reg <= d_parity_calc;
#7 d_reg = line_reg[64:127];
#1 if (trace)
  $display("    CPU write beat 2: %h at %d",d_reg,$time);
@(transfer_acknowledged);     //second beat done

data_reg = line_reg[128:191];
d_parity_calc[0] <= ~^data_reg[0:7];
d_parity_calc[1] <= ~^data_reg[8:15];
d_parity_calc[2] <= ~^data_reg[16:23];
d_parity_calc[3] <= ~^data_reg[24:31];
d_parity_calc[4] <= ~^data_reg[32:39];
d_parity_calc[5] <= ~^data_reg[40:47];
d_parity_calc[6] <= ~^data_reg[48:55];
d_parity_calc[7] = ~^data_reg[56:61];
dp_reg <= d_parity_calc;
#7 d_reg = line_reg[128:191];
#1 if (trace)
  $display("    CPU write beat 3: %h at %d",d_reg,$time);
@(transfer_acknowledged);     //third beat done

data_reg = line_reg[191:255];
d_parity_calc[0] <= ~^data_reg[0:7];
```

```
            d_parity_calc[1] <= ~^data_reg[8:15];
            d_parity_calc[2] <= ~^data_reg[16:23];
            d_parity_calc[3] <= ~^data_reg[24:31];
            d_parity_calc[4] <= ~^data_reg[32:39];
            d_parity_calc[5] <= ~^data_reg[40:47];
            d_parity_calc[6] <= ~^data_reg[48:55];
            d_parity_calc[7] = ~^data_reg[56:61];
            dp_reg <= d_parity_calc;
            #7 d_reg = line_reg[192:255];
            #1 if (trace)
              $display("    CPU write beat 4: %h at %d",d_reg,$time);
            @(transfer_acknowledged);     //fourth beat done
            d_reg <= #7 64'bz;
            dp_reg <= #7 8'bz;
            line_reg <= #7 256'bz;
            Transfer_type[dpp] <= #7 none;
            Transfer_code[dpp] <= #7 reserved;
            Transfer_start[dpp] <= #7 FALSE;
            dbb_reg_ = #4 hi;
            dbb_reg_ = #8 'bz;
          end

      default: $display("CPU module has bad TT[%b] = %b",dpp,
                Transfer_type[dpp]," at time %d.",$time);
    endcase
  end

endmodule
```

C. ARBITER

```
/*****************************************************************************
* BUS ARBITRATION UNIT
* Filename: arbiter.v
* Author: Joseph R. Robert, Jr.
* Date:   24AUG95
* Revised: 10JAN96
*
* Purpose:  This module emulates the system's external bus arbitration unit.
* It is implemented as a Finite State Machine.
* There are only two possible bus masters in this system: the CPU and the PRC.
* Also, the address bus and data bus are each arbitrated for independently,
* though the data bus arbitration occurs after the corresponding address bus
* arbitration.
*   If a unit wants the address bus, it asserts BR_.  If the bus is available,
* the aribter asserts BG_ back to that unit, which can then take mastership by
* asserting ABB_.  When it is done with the address bus, it negates ABB_.
```

```
*    It is assumed that if a unit wanted the address bus, it will also want the
* data bus. "Address only" transactions will not occur in this system, since
* there is no external cache or multiprocessors. Therefore, after asserting
* BG_ to the requesting unit, the arbiter asserts DBG_ on the next cycle.
*    BG_ and DBG_ are both asserted until the requesting unit takes mastership,
* unless the requesting unit withdraws its request by negating BR_.
*    If there are no pending bus requests, the arbiter "parks" the CPU by
* granting it the busses. This reduces memory access time for the CPU. If the
* CPU is parked, and then the PRC requests the bus, the CPU is unparked, and
* the arbiter can then grant the bus to the PRC.
*    The PowerPC can conduct a second address tenure long before the first data
* tenure is complete. This pipelining has a maximum depth of two transactions,
* meaning that a third address tenure will not start before the first data
* tenure is complete. The Memory Unit in this Testbench is capable of handling
* that situation. However, adding the PRC to the system creates the
* possibility that the PRC will initiate a third address tenure before the
* first of two CPU transactions is complete. This situation is handle by this
* Arbiter which keeps track of the pipelining depth. It will not grant the
* address bus to any unit if that address tenure would put a third transaction
* in the pipeline. Rather, the arbiter will stall until the data tenure from
* the first transaction is complete, and then will grant the address bus to the
* requesting unit.
*
*********************************************************************************/
module arbiter (CPU_BR_,CPU_BG_,CPU_DBG_,PRC_BR_,PRC_BG_,PRC_DBG_,
          ABB_,DBB_,clk);

output CPU_BG_, CPU_DBG_, PRC_BG_, PRC_DBG_;
input CPU_BR_, PRC_BR_, ABB_, DBB_, clk;
reg  CPU_BG_,CPU_DBG_, PRC_BG_, PRC_DBG_;
wire CPU_BR_, PRC_BR_, clk;

//Declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0;
  reg [1:0] requests;          //concatenated input signals
  reg [1:0] depth;
  tri stall;

//Finite State Machine variables and parameters
  reg [2:0] state, next_state;
  parameter start  = 1,
        grant_cpu_a = 2,
        park_cpu    = 3,
        grant_cpu_d = 4,
        grant_prc_a = 5,
        wait_for_prc = 6,
        grant_prc_d = 7;
```

```
//Initialize outputs
initial
  begin
    CPU_BG_ <= hi;
    CPU_DBG_ <= hi;
    PRC_BG_ <= hi;
    PRC_DBG_ <= hi;
    state     <= start;
    next_state <= start;
    requests <= 'b11;
    depth <= 0;
  end

//Track depth of pipeline
always @(posedge ABB_)
 begin
   depth = depth + 1;
 end

always @(posedge DBB_)
 begin
   depth = depth - 1;
 end

assign stall = (depth > 1);

//
//Arbitration

always
  begin

    wait (!stall);

    #5 state = next_state;

    #1 case (state)
      start: //1
        begin
          CPU_BG_ <= hi;
          CPU_DBG_ <= hi;
          PRC_BG_ <= hi;
          PRC_DBG_ <= hi;
          @(posedge clk) requests = {CPU_BR_,PRC_BR_};
          case (requests)
            2'b00: next_state = grant_cpu_a;
            2'b01: next_state = grant_cpu_a;
            2'b10: next_state = grant_prc_a;
            2'b11: next_state = grant_cpu_a;
          endcase
```

```
    end

  grant_cpu_a: //2
    begin
      CPU_BG_ <= low;
      CPU_DBG_ <= hi;
      PRC_BG_ <= hi;
      PRC_DBG_ <= hi;
      @(posedge clk);
      next_state = park_cpu;
    end

  park_cpu: //3
    begin
      CPU_BG_ <= low;
      CPU_DBG_ <= low;
      PRC_BG_ <= hi;
      PRC_DBG_ <= hi;
      @(posedge clk) requests = {CPU_BR_,PRC_BR_};
      case (requests)
        2'b00: next_state = park_cpu;
        2'b01: next_state = park_cpu;
        2'b10: next_state = grant_cpu_d;
        2'b11: next_state = park_cpu;
      endcase
    end

  grant_cpu_d: //4
    begin
      CPU_BG_ <= hi;
      CPU_DBG_ <= low;
      PRC_BG_ <= hi;
      PRC_DBG_ <= hi;
      @(posedge clk) requests = {CPU_BR_,PRC_BR_};
      case (requests)
        2'b00: next_state = park_cpu;
        2'b01: next_state = park_cpu;
        2'b10: next_state = grant_prc_a;
        2'b11: next_state = park_cpu;
      endcase
    end

  grant_prc_a: //5
    begin
      CPU_BG_ <= hi;
      CPU_DBG_ <= hi;
      PRC_BG_ <= low;
      PRC_DBG_ <= hi;
      @(posedge clk);
      next_state = wait_for_prc;
```

```
        end

      wait_for_prc: //6
        begin
          CPU_BG_ <= hi;
          CPU_DBG_ <= hi;
          PRC_BG_ <= low;
          PRC_DBG_ <= low;
          @(posedge clk) requests = {CPU_BR_,PRC_BR_};
          case (requests)
            2'b00: next_state = wait_for_prc;
            2'b01: next_state = grant_cpu_d;
            2'b10: next_state = wait_for_prc;
            2'b11: next_state = grant_prc_d;
          endcase
        end

      grant_prc_d: //7
        begin
          CPU_BG_ <= hi;
          CPU_DBG_ <= hi;
          PRC_BG_ <= hi;
          PRC_DBG_ <= low;
          wait (DBB_ == hi);
          @(posedge clk) requests = {CPU_BR_,PRC_BR_};
          case (requests)
            2'b00: next_state = grant_cpu_a;
            2'b01: next_state = grant_cpu_a;
            2'b10: next_state = grant_prc_a;
            2'b11: next_state = grant_cpu_a;
          endcase
        end

      default: $display("state error in module arbiter");

    endcase

  end

endmodule
```

D. MEMORY

```
/*************************************************************************
* RANDOM ACCESS MEMORY
* Filename: memory.v
* Author: Joseph R. Robert, Jr.
```

```
* Date:  24AUG95
* Revised: 10JAN96
*
* Purpose: This module emulates the system's main memory. For simulation
* efficiency, the memory has only enough physical address space for four burst
* reads. Thus, 128 bytes. The address bus width allows a virtual address space
* of 4 G-bytes. Accesses to addresses past the first 128 bytes map to within
* the first 128 bytes.
*     The time required for memory accesses are determined by Delay1 and
* Delay2. Delay1 is the delay, in cycle, required for the initial access.
* Delay2 is the delay required for each successive beat of four-beat
* operations. Set them both to 0 for fastest memory response. Set them to 8
* and 3 respectively for realistic memory response of a 60 ns DRAM. Do not set
* Delay2 > Delay1. That will not represent a realistic memory response, and
* will probably cause this module to act weird.
*     There is a two-stage pipeline involved with memory accesses, such that a
* memory tenure can be started while the previous data tenure is still active.
* To accomplish this, some signals have [0:1] in their declaration, and are
* indexed using pp and dpp, which are the address pipeline position pointer,
* and the data pipeline position pointer, respectively.
*     To keep this model simple, a single-beat read will always return a
*  single byte of data, regardless of TSIZ, in byte lane 0, which is different
* from the way the PowerPC really operates. See Table 10-4 on pg. 10-15 of
* the PowerPC-603 Users Manual for actual alignment. This simplification is
* irrelevant to the performance of the PRC which deals only with burst
* operations.
*     It is important to note that this memory module had to have one feature
* that is not typical of memory modules. It has a CANX input with cancels the
* current read operation. It is through this signal that the PRC stops the
* memory module from delivering data to the CPU when the PRC already has the
* data.
*
*********************************************************************************/
module memory (ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,TBST_,GBL_,CI_,WT_,CSE,AACK_,
      DBWO_,DBB_,D,DP,DPE_,DBDIS_,TA_,TEA_,CANX,clk);

// Signals are defined in system.v.
output AACK_,DBDIS_,TA_,APE_;
input [0:1] TC;
input  DBWO_,CI_,WT_,CSE,TEA_,DPE_,CANX,clk;
input  [0:31] A;
inout  [0:63] D;
inout  [0:7] DP;
input  [0:4] TT;
inout  [0:3] AP;
inout  [0:2] TSIZ;
inout  ABB_,TS_,TBST_,GBL_,DBB_;

wire  [0:31] A;
wire CI_,WT_,CSE,TEA_,DPE_,ARTRY_;
```

```
reg AACK_,APE_,DBDIS_,DRTRY_;
tri [0:63] D;
tri [0:7] DP;
tri [0:3] AP;
tri [0:2] TSIZ;
tri ABB_,TS_,TBST_,GBL_,DBB_,TA_;
reg [0:63] d_reg, data;
  assign D = d_reg;
//
//Declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0,
        Size  = 128, //Size of memory in bytes.
        Length = 7,  //Length of physical address in bits.
        Delay1 = 8,  //Delay for address translation.
        Delay2 = 3;  //Delay between successive beats.

  parameter //for Transfer_type
    none           = 5'bzzzzz,
    write          = 5'b00010,
    write_atomic   = 5'b10010,
    read           = 5'b01010,
    read_atomic    = 5'b11010,
    burst_write    = 5'b00110,
    burst_read     = 5'b01110,
    burst_read_atomic = 5'b11110;

  reg [0:31] virtual_addr, index;
  reg [0:3] addr_parity_calc,addr_parity_in;
  reg [0:Length-1]  pa_reg, physical_addr [0:1];
  reg [0:7]
     mem [0:Size-1],
     mem_reg;      //Memory data register
  reg [0:4] Transfer_type [0:1];
  reg [0:2] Transfer_size [0:1];
  reg burst [0:1];
  reg [0:1] i, burst_start;
  reg pp,dpp; //current pipeline and data pipeline positions
  reg abort;
  reg ta_reg_;
    assign TA_ = ta_reg_;

//Initialize memory
initial
  begin
    abort <= FALSE;
    AACK_  <= hi;
    addr_parity_calc <= 3'bz;
```

```
    addr_parity_in  <= 3'bz;
    DBDIS_ <= hi;
    ta_reg_  <= 'bz;
    d_reg  <= 64'bz;
    Transfer_type[0] <= none;
    Transfer_type[1] <= none;
    Transfer_size[0] <= 'bz;
    Transfer_size[1] <= 'bz;
    burst[0] <= 'bz;
    burst[1] <= 'bz;
    pp  <= 1'b1;
    dpp <= 1'b1;
    for (index = 0; index<Size; index=index+1)
      mem[index] = index;
  end

//
//ADDRESS TENURE
  always @(posedge clk)
    begin
      if (ABB_ == low)
        begin
          //latch address and attributes
          pp = ~pp;
          Transfer_type[pp] <= TT;
          Transfer_size[pp] <= TSIZ;
          burst[pp] <= TBST_;
          //insert other attributes here.
          addr_parity_in <= AP;
          virtual_addr = A;
          addr_parity_calc[0] <= ~^virtual_addr[0:7];
          addr_parity_calc[1] <= ~^virtual_addr[8:15];
          addr_parity_calc[2] <= ~^virtual_addr[16:23];
          addr_parity_calc[3] <= ~^virtual_addr[24:31];
          physical_addr[pp] = virtual_addr[32-Length:31];
          if (addr_parity_in != addr_parity_calc)
            begin
              $display("Memory: address parity error.");
              $display("   Calculated parity: %b",addr_parity_calc);
              $display("   Recevied parity:   %b",addr_parity_in);
            end
          AACK_ = #7 low;
          wait (AACK_==hi);
        end
    end

  always @(posedge clk)
    begin
      if (AACK_ == low)
        AACK_ = #7 hi;
```

```
    end

//DATA TENURE
  always @(posedge clk)
    begin
      if (CANX == hi)
        abort = TRUE;
    end


  always
    begin
      #1 dpp = ~dpp;
      #1 case (Transfer_type[dpp])
        none: begin end

        read:
          begin
            repeat(Delay1)@(posedge clk);
            #7 ta_reg_ <= low;
            d_reg[0:7] <= mem[physical_addr[dpp]];
            Transfer_size[dpp] <= 'bz;
            @(posedge clk)
              Transfer_type[dpp] <= none;
              #7 ta_reg_ = 'bz;
              d_reg[0:7] <= 'bz;
          end

        write:
          begin
            repeat(Delay1)@(posedge clk);
            #7 ta_reg_ <= low;
            @(posedge clk)
              //latch data
              data = D;
              mem[physical_addr[dpp]]   <= data[0:7];
              #7 ta_reg_ = 'bz;
              Transfer_size[dpp] <= 'bz;
              Transfer_type[dpp] <= none;
          end

        burst_read:
          begin
            //find critical double-word
            #2 pa_reg = physical_addr[dpp];
            burst_start = pa_reg[Length-5:Length-4];
            //align to cache line
            pa_reg[Length-5:Length-1] = 5'b00000;
            physical_addr[dpp] = pa_reg;
            if (!abort) if (Delay1-Delay2-1 >= 0)
```

```
        repeat(Delay1-Delay2-1)@(posedge clk);

      for (index=0; index<4; index=index+1)
        begin
          if (!abort) repeat(Delay2)@(posedge clk);
          if (Delay1-Delay2!=0 || index!=0) @(posedge clk);
          if (!abort) begin
            #7 ta_reg_ <= low;
            i = burst_start+index; //i is mod 4
            d_reg[ 0: 7]<=mem[physical_addr[dpp]+8*i];
            d_reg[ 8:15]<=mem[physical_addr[dpp]+8*i+1];
            d_reg[16:23]<=mem[physical_addr[dpp]+8*i+2];
            d_reg[24:31]<=mem[physical_addr[dpp]+8*i+3];
            d_reg[32:39]<=mem[physical_addr[dpp]+8*i+4];
            d_reg[40:47]<=mem[physical_addr[dpp]+8*i+5];
            d_reg[48:55]<=mem[physical_addr[dpp]+8*i+6];
            d_reg[56:63]<=mem[physical_addr[dpp]+8*i+7];
            if (Delay2!=0)
              begin
                ta_reg_ <= #13 'bz;
                d_reg <= #13 64'bz;
              end
            end
          else
            index <= 5;
        end

      @(posedge clk)
      ta_reg_ <= #7 'bz;
      d_reg <= #7 64'bz;
      Transfer_size[dpp] <= 'bz;
      Transfer_type[dpp] <= none;
      abort <= FALSE;
    end

  burst_write:
    begin
      //burst-writes are always performed in order
      if (Delay1-Delay2 >= 0)
        repeat(Delay1-Delay2)@(posedge clk);
      for (index=0; index<4; index=index+1)
        begin
          repeat(Delay2)@(posedge clk);
          #7 ta_reg_ <= low;
          i = index;
          @(posedge clk) //latch data
          data = D;
          mem[physical_addr[dpp]+8*i]   <= data[ 0: 7];
          mem[physical_addr[dpp]+8*i+1] <= data[ 8:15];
          mem[physical_addr[dpp]+8*i+2] <= data[16:23];
```

```
                mem[physical_addr[dpp]+8*i+3] <= data[24:31];
                mem[physical_addr[dpp]+8*i+4] <= data[32:39];
                mem[physical_addr[dpp]+8*i+5] <= data[40:47];
                mem[physical_addr[dpp]+8*i+6] <= data[48:55];
                mem[physical_addr[dpp]+8*i+7] <= data[56:63];
                if (Delay2!=0)
                    ta_reg_ <= #7 'bz;
              end
            ta_reg_ <= #7 'bz;
            data <= #7 64'bz;
            Transfer_size[dpp] <= 'bz;
            Transfer_type[dpp] <= none;
            @(posedge clk);
          end


        default: $display("Memory module received bad TT[%d] = %b",dpp,
                    Transfer_type[dpp]," at time %d", $time);
      endcase
    end

endmodule
```

APPENDIX C. PRC BEHAVIOR FILES

The files in this appendix are the result of the behavioral design phase. They include the verilog behavioral models of the PRC and the testing results. The files are located on the Computer Center system at *joshua_u2/jrrobert/thesis/verilog/behavior*.

A. PRC

```
/*************************************************************************
* Predictive Read Cache
* Filename: prc.v
* Author: Joseph R. Robert, Jr.
* Date:   02OCT95
* Revised: 10JAN96
*
* Purpose:  This module emulates the predictive read cache.
*
*************************************************************************/
module prc(CPU_BR_,BR_,BG_,ABB_,TS_,A,AP,APE_,TT,TSIZ,TC,TBST_,AACK_,
      DBG_,DBB_,D,DP,DPE_,TA_,HRESET_,CANX,clk);

// Signals are defined in system.v.  Notations follow conventions used in
//  PowerPC Users Manual.
input  CPU_BR_,BG_,AACK_,DBG_,TA_,HRESET_,clk;
output [0:1] TC;
output BR_,APE_,DPE_,CANX;
inout  [0:31] A;
inout  [0:63] D;
inout  [0:7] DP;
inout  [0:4] TT;
inout  [0:3] AP;
inout  [0:2] TSIZ;
inout  ABB_,TS_,TBST_,DBB_;

wire  [0:1] TC;
wire  BR_,APE_,DPE_,CANX;
wire  [0:31] A;
wire  [0:63] D;
wire  [0:7] DP;
```

```
 wire  [0:4] TT;
 wire  [0:3] AP;
 wire  [0:2] TSIZ;
 wire  ABB_,TS_,TBST_,DBB_;

 //declare variables, constants, parameters
   parameter TRUE  = 1'b1,
         FALSE = 1'b0,
         hi    = 1'b1,
         low   = 1'b0;

   //Other internal control signals
   wire CAR_latch, predict,snoop_ignore;
   wire [0:255] DATALINE;
   wire [0:26] CAR;  // current address register
   wire [0:26] NAR;  // next address register
   wire [0:26] MRMA; // most recent memory access
   wire [0:6] ActiveLine;
   wire [0:1] BURSTSTART;

//Connect parts
   bus_interface BIU1(NAR,BURSTSTART,BG_,CPU_BR_,AACK_,DBG_,
    send,fetch,clk,BR_,upload,download,fetch_done,
    send_done,CANX,snoop_ignore,
    DATALINE,D,A,DP,DPE_,TT,TSIZ,ABB_,TS_,TBST_,DBB_,TA_,HRESET_);
   snooper SNP1(A,AP,TT,TC,TS_,snoop_ignore,hold,clk,CAR,BURSTSTART,read,write);
   controller CON1(HRESET_,read,write,hit,send_done,fetch_done,
    line_empty,a_select,test,predict,store,
    flush,send,hold,new_replace,fetch,clk);
   predictor PRE1(MRMA,CAR,predict,NAR);
   line_mgr LM1(CAR,NAR,HRESET_,a_select,test,fetch_done,flush,store,
    new_replace,MRMA,ActiveLine,line_empty,hit);
   datalist DL1(DATALINE,ActiveLine,upload,download);

endmodule
```

B. CONTROLLER

```
/*************************************************************************
* CONTROLLER
* Filename: controller.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
* Revised: 05JAN96
*
* Purpose: This module is a Finite State Machine which coordinates the actions
* of all the other functional blocks of the PRC.  All control signals are
```

```
* synchronous with the system clock.  HRESET_ causes the Controller to go to
*  the IDLE state.  See state diagram and state output tables.
*
*************************************************************************************/
module controller (HRESET_,read,write,hit,send_done,fetch_done,
  line_empty,a_select,test,predict,store,
  flush,send,hold,new_replace,fetch,clk);

input HRESET_,read,write,hit,send_done,fetch_done,line_empty,clk;
output a_select,test,predict,store,flush,send,hold,new_replace,fetch;

reg a_select,test,predict,store,flush,send,hold,new_replace,fetch;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0,
        trace = FALSE;

  //Finite State Machine variable and parameters
    reg [0:3] state, next_state;
    reg [0:2] inputs3;
    reg [0:1] inputs2;
    reg input1;
    parameter idle   = 0,
          test_car_r = 1,
          send_data = 2,
          test_nar = 3,
          fetch_data = 4,
          is_line_empty = 5,
          predict_na = 6,
          store_car = 7,
          test_car_w = 8,
          flush_line = 9;

//initialize signals
initial
  begin
    state <= idle;          //The state variables must be initialized to
    next_state <= idle;     //avoid the default error message.
  end

//FINITE STATE MACHINE
always @(negedge HRESET_)
 begin
   state <= idle;
   next_state <= idle;
   wait(HRESET_ == hi);
 end
```

```
always
 begin
  #2 state = next_state;
  if (trace)
   $display("Controller entered state %d.",state);

  #1 case (state)
   idle: //0
    begin
     //a_select   <= low;
     test       <= low;
     predict    <= low;
     store      <= low;
     flush      <= low;
     send       <= low;
     hold       <= low;
     new_replace <= low;
     fetch      <= low;
     @(posedge clk) inputs2 = {read,write};
     if (HRESET_ == low)
       next_state = idle;
     else
      case (inputs2)
       2'b00: next_state = idle;
       2'b01: next_state = test_car_w;
       2'b10: next_state = test_car_r;
       2'b11: next_state = test_car_w;  //This should not happen.
      endcase
    end

   test_car_r: //1
    begin
     a_select   <= low;  //CAR
     test       <= hi;
     predict    <= low;
     store      <= low;
     flush      <= low;
     send       <= low;
     hold       <= hi;
     new_replace <= low;
     fetch      <= low;
     @(posedge clk) input1 = hit;
     case (input1)
      1'b0: next_state = is_line_empty;
      1'b1: next_state = send_data;
     endcase
    end

   send_data: //2
    begin
```

```
    //a_select   <= low;
    test       <= low;
    predict    <= hi;
    store      <= low;
    flush      <= low;
    send       <= hi;
    hold       <= hi;
    new_replace <= low;
    fetch      <= low;
    @(posedge clk) input1 = send_done;
    case (input1)
      1'b0: next_state = send_data;
      1'b1: next_state = test_nar;
    endcase
  end

 test_nar: //3
  begin
    a_select   <= hi; //NAR
    test       <= hi;
    predict    <= low;
    store      <= low;
    flush      <= low;
    send       <= low;
    hold       <= hi;
    new_replace <= low;
    fetch      <= low;
    @(posedge clk) inputs3 = {hit,read,write};
    case (inputs3)
      3'b000: next_state = fetch_data;
      3'b001: next_state = idle;
      3'b010: next_state = idle;
      3'b011: next_state = idle; //This should not happen.
      3'b100: next_state = idle;
      3'b101: next_state = idle;
      3'b110: next_state = idle;
      3'b111: next_state = idle; //This should not happen.
    endcase
  end

 fetch_data: //4
  begin
    a_select   <= hi; //NAR
    test       <= low;
    predict    <= low;
    store      <= low;
    flush      <= low;
    send       <= low;
    hold       <= hi;
    new_replace <= low;
```

```
    fetch      <= hi;
    @(posedge clk) input1 = fetch_done;
    case (input1)
     1'b0: next_state = fetch_data;
     1'b1: next_state = idle;
    endcase
   end

 is_line_empty: //5
   begin
    //a_select   <= low;
    test       <= low;
    predict    <= low;
    store      <= low;
    flush      <= low;
    send       <= low;
    hold       <= hi;
    new_replace <= low;
    fetch      <= low;
    @(posedge clk) input1 = line_empty;
    case (input1)
     1'b0: next_state = predict_na;
     1'b1: next_state = store_car;
    endcase
   end

 predict_na: //6
   begin
    //a_select   <= low;
    test       <= low;
    predict    <= hi;
    store      <= low;
    flush      <= low;
    send       <= low;
    hold       <= hi;
    new_replace <= hi;
    fetch      <= low;
    @(posedge clk) next_state = test_nar;
   end

 store_car: //7
   begin
    a_select   <= low; //CAR
    test       <= low;
    predict    <= low;
    store      <= hi;
    flush      <= low;
    send       <= low;
    hold       <= hi;
    new_replace <= low;
```

```
        fetch      <= low;
        @(posedge clk) next_state = idle;
       end

      test_car_w: //8
       begin
        a_select   <= low;  //CAR
        test       <= hi;
        predict    <= low;
        store      <= low;
        flush      <= low;
        send       <= low;
        hold       <= hi;
        new_replace <= low;
        fetch      <= low;
        @(posedge clk) input1 = hit;
        case (input1)
         1'b0: next_state = idle;
         1'b1: next_state = flush_line;
        endcase
       end

      flush_line: //9
       begin
        //a_select   <= low;
        test       <= low;
        predict    <= low;
        store      <= low;
        flush      <= hi;
        send       <= low;
        hold       <= hi;
        new_replace <= low;
        fetch      <= low;
        @(posedge clk) next_state = idle;
       end

      default:
       begin
        $display("state error in module controller.");
        $display("   state = %b.",state);
       end

    endcase
  end

endmodule
```

C. SNOOPER

```
/*****************************************************************************
* SNOOPER
* Filename: snooper.v
* Author: Joseph R. Robert, Jr.
* Date:  21DEC95
* Revised: 05JAN96
*
* Purpose: This module watches the system bus activity, and makes appropriate
* reports to the PRC Controller.
*  If the transaction is a data burst read or any kind of write, and if the
* address parity is correct, then the read or write signal is asserted as
* appropriate, and the address is placed in the CAR. The snoop_ignore signal
* tells this unit to ignore the current transaction, because it was initiated
* by the Bus Interface Unit. The snoop_ignore signal must be asserted
* concurrently with the transfer attributes.
*  Reads that are not burst or data related are ignored by the PRC. The CAR
* is updated only on transactions relevant to the PRC.
*  Due to the two-stage pipelining capability of the PowerPC, with respect to
* memory accesses, a second address tenure can occur shortly after the first,
* well before the first data tenure is complete. To compensate for this, the
* read and write outputs of the Snooper will remain exerted until acknowledged
* by the Controller with hold. The rising edge of hold indicates that the read
* or write signal was received by the Controller. The Snooper can then negate
* these signals, but must leave CAR alone until hold is negated. After hold is
* negated, CAR can be updated to the new address.
*
*****************************************************************************/
module snooper (A,AP,TT,TC,TS_,snoop_ignore,hold,clk,CAR,BURSTSTART,
        read_flag,write_flag);

input [0:31] A;
input [0:3] AP;
input [0:4] TT;
input [0:1] TC;
input TS_,snoop_ignore,hold,clk;
output [0:26] CAR;
output [0:1] BURSTSTART;
output read_flag,write_flag;

reg [0:26] CAR;
reg [0:1] BURSTSTART;
reg read_flag,write_flag;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
```

```
        hi   = 1'b1,
        low  = 1'b0;

    //Address related
    reg [0:31] address;
    reg [0:3] addr_parity,addr_parity_calc;

    //Other external control signals
    reg [0:4] Transfer_type;
    parameter //for Transfer_type
            none        = 5'bz,
            write       = 5'b00010,      //02
            write_atomic = 5'b10010,  //12
            read        = 5'b01010,      //0A
            read_atomic  = 5'b11010,  //1A
            burst_write  = 5'b00110,  //06
            burst_read   = 5'b01110,  //0E
            burst_read_atomic = 5'b11110;  //1E
    reg [0:1] Transfer_code;
    parameter //for Transfer_code
       data_transfer    = 2'b00,
       touch_load       = 2'b01,
       instruction_fetch = 2'b10,
       reserved         = 2'b11;
    reg ignore;

    //Other internal control signals
    reg valid_read_0, valid_read_1;  //The numbers indicate the pipeline stage.
    reg valid_write_0, valid_write_1;
    tri parity_valid;
    reg Transaction_waiting;

//initialize variables
initial
  begin
    CAR <= 27'bz;
    BURSTSTART <= 2'bz;
    read_flag <= low;
    write_flag <= low;
    address <= 32'bz;
    addr_parity <= 4'bz;
    addr_parity_calc <= 4'bz;
    Transfer_type <= none;
    Transfer_code <= none;
    ignore <= low;
    Transaction_waiting <= low;
  end

//BEHAVIOR
```

```
//Calculate address parity.
always @(address)
  begin
    addr_parity_calc[0] <= ~^address[0:7];
    addr_parity_calc[1] <= ~^address[8:15];
    addr_parity_calc[2] <= ~^address[16:23];
    addr_parity_calc[3]  = ~^address[24:31];
  end

  assign parity_valid = (addr_parity_calc == addr_parity);

//If there is a transaction,
//  and that transaction is a data burst read or any kind of write
//  and the transaction is not initiated by the PRC itself,
//  and if the address parity is correct
//then report the type of transaction to the Controller.

always @(posedge clk)
 begin
  if (TS_==low)
    begin  //latch address and attributes in stage 0.
     address <= A;
     Transfer_type <= TT;
     Transfer_code <= TC;
     ignore <= snoop_ignore;
     addr_parity = AP;
     #2 valid_read_0  = Transfer_code == data_transfer &
                  (Transfer_type == burst_read |
                   Transfer_type == burst_read_atomic);
     valid_write_0 = Transfer_type == write |
                 Transfer_type == write_atomic |
                 Transfer_type == burst_write;
     #4 if (!ignore & parity_valid & (valid_read_0 | valid_write_0))
       Transaction_waiting = hi;
    end
 end

always @(posedge hold)
 begin
  read_flag <= low;
  write_flag <= low;
 end


always
  begin
   wait(Transaction_waiting);
   valid_read_1  = valid_read_0;
   valid_write_1 = valid_write_0;
   Transaction_waiting = #2 low;
```

```
    wait(!hold);
    if (valid_read_1)
     begin
      read_flag <= hi;
      CAR = address[0:26];
      BURSTSTART = address[27:28];
     end
    else if (valid_write_1)
     begin
      write_flag <= hi;
      CAR = address[0:26];
      BURSTSTART = address[27:28];
     end
  end

endmodule
```

D. LINE MANAGER

```
/*********************************************************************************
* LINE MANAGER
* Filename: line_mgr.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
* Revised: 05JAN95
*
* Purpose: This module contains the address list, status flags for each line
* (Valid, Aged), a general status flag (line_empty), the line replacement unit,
* and a couple of pointers (ActiveLine, ReplaceLine).
*   The MRMA output is always the MRMA of the ActiveLine. The line_empty
* flag indicates that the currently active line has no addresses in it yet, and
* therefore, cannot be used by the PRC to make a prediction.
*   The input a_select determines which address input is used for a particular
* operation. The two address inputs are the CAR and the NAR.
*   When the Line Manager receives a test signal, it compares the input address
* with the contents of the PredMA List. If there is a match with the CAR, it
* asserts the hit signal, and changes the ActiveLine pointer to the line number
* of the match.
*   If there is a miss with the CAR, then the ActiveLine switches to the same
* line pointed to by ReplaceLine.
*   If, during a test, there is a match with the NAR, hit is asserted, and the
* value in ActiveLine is irrelevant since it will not be used. If there is a
* miss with the NAR, the ActiveLine must remain unchanged from the test.
*   The fetch_done signal from the Bus Interface Unit causes the NAR to be
* stored in PredMA[ActiveLine], the CAR to be stored in MRMA[ActiveLine], the
* Valid flag to be set, and the Aged flag to be reset.
*   The flush signal causes the current ActiveLine to become invalid by setting
```

```
* Valid[ActiveLine] = 0.
*  The store signal causes the input address to be stored into the MRMA of the
* ActiveLine. This is only used for the first address in a new line.  Store
* also causes the line_empty flag to be reset.
*
*********************************************************************************/
module line_mgr (CAR,NAR,HRESET_,a_select,test,fetch_done,flush,store,
   new_replace,MRMA_out,ActiveLine,line_empty,hit);

input [0:26] CAR,NAR;
input HRESET_,a_select,test,fetch_done,flush,store,new_replace;
output [0:26] MRMA_out;
output [0:6] ActiveLine;
output line_empty,hit;

reg [0:26] MRMA_out;
reg [0:6] ActiveLine;
reg line_empty,hit;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0;

  //Address related
  reg [0:26] in_addr;

  //Data structure
  reg [0:26] PredMA [0:127],
         MRMA [0:127],
         PredMA_reg,MRMA_reg;
  reg Valid [0:127],
     Aged [0:127];

  //Other internal control signals
  reg [0:7] i1,i2,i3;
  reg [0:6] ReplaceLine;
  reg match,temp,all_lines_are_valid,done;

//initialize variables
initial
 begin
  for (i1=0; i1<=127; i1=i1+1)
   begin
    PredMA[i1] <= 27'b0;
    MRMA[i1]   <= 27'b0;
   end
 end
```

```
//BEHAVIOR

always @(negedge HRESET_)
 begin
  for (i1=0; i1<=127; i1=i1+1)
   begin
    Valid[i1] <= low;
    Aged[i1]  <= low;
   end
  ActiveLine  <= 0;
  ReplaceLine <= 0;
  line_empty  <= hi;
  wait(HRESET_ == hi);
 end

always @(a_select or CAR or NAR) //address multiplexer
 begin
  if (a_select==0)
   in_addr = CAR;
  else
   in_addr = NAR;
 end

always @(ActiveLine)
 begin
  MRMA_out = MRMA[ActiveLine];
  $display("Line_mgr selected new ActiveLine = %d at $d",ActiveLine,$time);
 end

always @(posedge test)
 begin
  hit   = low;
  match = low;
  #2 i2 = 0;
  while (!match & i2<128)
   if (PredMA[i2] == in_addr & Valid[i2])
    match = hi;
   else
    i2 = i2 + 1;
  #2 if (match & a_select==0) //a match with the CAR
   begin
    hit <= hi;
    ActiveLine <= i2;
   end
  else if (match & a_select==1) // a match with the NAR
    hit <= hi;
  else if (!match & a_select==0) //a miss with the CAR
    ActiveLine <= ReplaceLine;
  else if (!match & a_select==1) //a miss with the NAR
    begin end// Do nothing.
```

```
  end

 always @(posedge fetch_done)
  begin
    MRMA[ActiveLine] <= CAR;
    MRMA_out        <= CAR;
    PredMA[ActiveLine] <= NAR;
    Valid[ActiveLine]  <= hi;
    Aged[ActiveLine] = low;
  end

 always @(posedge flush)
  begin
    Valid[ActiveLine] = low;
    $display("Line manager flushed line %d at time %d.",ActiveLine,$time);
  end

 always @(posedge store)
  begin
    MRMA[ActiveLine] = in_addr;
    MRMA_out = MRMA[ActiveLine];
    line_empty = 0;
  end

 /*****************************************************************************
 * LINE REPLACEMENT UNIT
 *
 *   ReplaceLine always points to the line to be replaced at the next PRC miss.
 *   As soon as the PRC starts predicting the first address for a line it
 * asserts new_replace, and the Line Replacement Unit can then find a new line
 * to mark as the next ReplaceLine. It searches sequentially for the next line
 * with invalid data and marks that line as the next to be replaced. If all
 * lines contain valid data, then it scans for the next line that is "aged",
 * indicated by a set Aged flag. As it scans for an aged line, it sets the Aged
 * bits in the lines it passes. Therefore, as it wraps around in search of an
 * aged line, it will eventually come upon one, even if none were aged when the
 * search began.
 *   All of this occurs while the PRC is fetching data, so it has several clock
 * periods in which to complete the search.
 *****************************************************************************/

 always
  begin
    temp = TRUE;
    for (i3=0; i3<=127; i3=i3+1)
      if (!Valid[i3])
        temp = FALSE;
    #1 all_lines_are_valid = temp;
  end
```

```
always @(posedge new_replace) // find the next ReplaceLine
 begin
  done = FALSE;
  #2 while (!done)
   begin
    ReplaceLine = ReplaceLine + 1;  //mod 128 addition
    if (!Valid[ReplaceLine])
     done = TRUE;
    else if (all_lines_are_valid & Aged[ReplaceLine])
     done = TRUE;
    else
     Aged[ReplaceLine] = 1;
   end
  line_empty = hi;
 end

endmodule
```

E. PREDICTOR

```
/*********************************************************************************
* PREDICTOR
* Filename: predictor.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
* Revised: 05JAN96
*
* Purpose: This module calculates the Next Address (stored in NAR) based on the
* Most Recent Memory Access (MRMA) and the Current Address (in the CAR).  The
* prediction calculation is
*
*    NAR = 2*CAR - MRMA
*
* The calculation is initiated upon each rising edge of the predict signal.
* The output NAR remains latched and valid until the next predict leading edge.
*
*********************************************************************************/
module predictor (MRMA,CAR,predict,NAR);

input [0:26] MRMA,CAR;
input predict;
output [0:26] NAR;

reg [0:26] NAR;

parameter TRUE = 1'b1,
      FALSE = 1'b0,
```

```
        trace = FALSE;

 // behavior
 always @(posedge predict)
  begin
    NAR = 2*CAR - MRMA;
    if (trace)
     begin
       $display("Predictor: NAR     = 2*CAR     - MRMA");
       $display("           %h = 2*%h - %h",{NAR,5'b0},{CAR,5'b0},{MRMA,5'b0});
     end
  end

endmodule
```

F. DATA LIST

```
/*****************************************************************************
* DATA LIST
* Filename: datalist.v
* Author: Joseph R. Robert, Jr.
* Date:   15DEC95
* Revised: 05JAN96
*
* Purpose: This module emulates the PRC's Data List.
*
*   An upload signal causes the Data List to store the data on data_line into
* the address specified by ActiveLine.
*   A download signal causes the Data List to assert onto data_line the data in
* the address specified by ActiveLine.
*
*****************************************************************************/
module datalist (data_line,ActiveLine,upload,download);

input [0:6] ActiveLine;
input upload,download;
inout [0:255] data_line;

tri [0:255] data_line;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
            FALSE = 1'b0,
            hi    = 1'b1,
            low   = 1'b0,
            trace = TRUE;
```

```
  //Data structure
  reg [0:255] line [0:127],
        line_reg,
        data_line_reg;
   assign data_line = data_line_reg;

//initialize signals
initial
  begin
    data_line_reg <= 256'bz;
  end

//BEHAVIOR
always @(posedge upload)
 begin
  line_reg = data_line;
  line[ActiveLine] = line_reg;
  if (trace) begin
   $display("DATALIST uploaded this data into line %h at time %d.",
        ActiveLine,$time);
   $display("    %h",line_reg);
   end
 end

always @(posedge download)
 begin
  line_reg = line[ActiveLine];
  data_line_reg = line_reg;
  if (trace) begin
   $display("DATALIST downloaded this data from line %h at time %d.",
        ActiveLine,$time);
   $display("    %h",line_reg);
   end
 end

always @(negedge download)
 begin
  data_line_reg = 256'bz;
 end

endmodule
```

G. BUS INTERFACE UNIT

```
/*********************************************************************************
* BUS INTERFACE UNIT
* Filename: bus_interface.v
* Author: Joseph R. Robert, Jr.
* Date:  09OCT95
* Revised: 05JAN96
*
* Purpose: This module connects the PRC with the system bus.  It handles
* the protocol of data transfer in and out of the PRC.
*  When this module received a fetch signal, it latches the address in the
* NAR, and requests the bus for a burst read.  It stores the incoming data
* until all four bursts have been received.  Then it uploads the data into the
* Data List and assserts fetch_complete.
*  When this module receives a send signal, it sends a cancel signal (CANX) to
* the memory module, downloads data from the Data List, and then sends the data
* to the CPU.  When the transfer is finished, it asserts send_done.
*  The coordination of these activities is accomplished through the use of a
* Finite State Machine.
*
*********************************************************************************/
module bus_interface (NAR_IN,BURSTSTART,BG_,CPU_BR_,AACK_,DBG_,
  send,fetch,clk,BR_,upload,download,fetch_done,
  send_done,CANX,snoop_ignore,
  DATALINE,D,A,DP,DPE_,TT,TSIZ,ABB_,TS_,TBST_,DBB_,TA_,HRESET_);

// Signals are defined in system.v.
input [0:26] NAR_IN;
input [0:1] BURSTSTART;
input BG_,CPU_BR_,AACK_,DBG_,send,fetch,clk,HRESET_;
output BR_,upload,download,fetch_done;
output send_done,DPE_,CANX,snoop_ignore;
inout [0:255] DATALINE;
inout [0:63] D;
inout [0:31] A;
inout [0:7] DP;
inout [0:4] TT;
inout [0:2] TSIZ;
inout ABB_,TS_,TBST_,DBB_,TA_;

reg BR_,upload,download,fetch_done,send_done,CANX,snoop_ignore;
tri [0:255] DATALINE;
tri [0:63] D;
tri [0:31] A;
tri [0:7] DP;
tri [0:4] TT;
tri [0:3] AP;
```

```
tri [0:2] TSIZ;
tri ABB_,TS_,TBST_,DBB_,TA_,DPE_;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
            FALSE = 1'b0,
            hi    = 1'b1,
            low   = 1'b0,
            trace = TRUE;

  //Address related
  reg [0:31] NAR;
  reg [0:31] a_reg;
    assign A = a_reg;
  reg [0:3] ap_reg, addr_parity_calc;
    assign AP = ap_reg;
  reg [0:1] burst_start;

  //Data related
  reg [0:255] data_line_reg, data_line;
    assign DATALINE = data_line_reg;
  reg [0:63] d_reg,data_reg;
    assign D = d_reg;
  reg [0:7] dp_reg, data_parity_calc, data_parity_in;
    assign DP = dp_reg;

  //Other external control signals
  reg [0:4] tt_reg,Transfer_type;
    assign TT = tt_reg;
    parameter //for Transfer_type
          none        = 5'bz,
          burst_write = 5'b00110,   //06
          burst_read  = 5'b01110,   //0E
          burst_read_atomic = 5'b11110;  //1E
  reg [0:2] tsiz_reg;
    assign TSIZ = tsiz_reg;
  reg abb_reg_,dbb_reg_,ts_reg_,tbst_reg_,ta_reg_;
    assign ABB_ = abb_reg_;
    assign DBB_ = dbb_reg_;
    assign TS_  = ts_reg_;
    assign TBST_ = tbst_reg_;
    assign TA_  = ta_reg_;

  //Other internal control signals
  reg [0:2] i; //counter
  reg [0:1] j; //counter
  wire qual_BG_,qual_DBG_;
  reg AB_Master,Transfer_in_progress,Transfer_start,Addr_termination,
     Data_Parity_Error;
    assign DPE_ = ~Data_Parity_Error;
```

```
    event transfer_acknowledged,start_send;

    //Finite State Machine variable and parameters
      reg [0:3] state, next_state;
      reg [0:1] inputs2;
      reg input1;
      parameter idle   = 0,
                fetch1 = 1,
                fetch2 = 2,
                fetch3 = 3,
                send1  = 5,
                send2  = 6;

//initialize signals
initial
  begin
    BR_ <= hi;
    upload <= low;
    download <= low;
    fetch_done <= low;
    CANX <= low;
    NAR <= 32'bz;
    a_reg <= 32'bz;
    ap_reg <= 4'bz;
    addr_parity_calc <= 4'bz;
    burst_start <= 2'bz;
    data_line_reg <= 256'bz;
    data_line <= 256'bz;
    d_reg <= 64'bz;
    data_reg <= 64'bz;
    dp_reg <= 8'bz;
    data_parity_calc <= 8'bz;
    data_parity_in <= 8'bz;
    tt_reg <= 5'bz;
    tsiz_reg <= 3'bz;
    abb_reg_ <= 'bz;
    dbb_reg_ <= 'bz;
    ts_reg_ <= 'bz;
    tbst_reg_ <= 'bz;
    ta_reg_ <= 'bz;
    i <= 3'bz;
    j <= 2'bz;
    AB_Master <= low;
    Transfer_in_progress <= low;
    Transfer_start <= low;
    Addr_termination <= low;
    Data_Parity_Error <= low;
    send_done <= low;
    snoop_ignore <= low;
    state <= 0;
```

```
    next_state <= 0;
    inputs2 <= 2'bz;
    input1 <= 'bz;
  end

//ADDRESS BUS ARBITRATION

  assign qual_BG_ = ~(!BG_ & ABB_);

  //Assume mastership
  always @(posedge clk)
    if (qual_BG_ == low)
      begin
        abb_reg_ = #2 low;
        AB_Master = TRUE;
        BR_  <= #1 hi;
      end

  //Calculate address parity.
  always @(NAR)
    begin
      addr_parity_calc[0] <= ~^NAR[0:7];
      addr_parity_calc[1] <= ~^NAR[8:15];
      addr_parity_calc[2] <= ~^NAR[16:23];
      addr_parity_calc[3]  = ~^NAR[24:31];
    end

  //Transfer address
  always @(posedge clk)
    if (qual_BG_ == low)
      begin
        ts_reg_ = #7 low;
        Transfer_start <= TRUE;
        a_reg <= NAR;
        ap_reg <= addr_parity_calc;
        tt_reg <= burst_read;
        tsiz_reg <= 3'b010;
        tbst_reg_ <= low;
        snoop_ignore <= hi;
        if (trace)
          $display("BIU started read from address %h at time %d.",
                NAR,$time);
      end

  always @(posedge clk)
    if (AB_Master & TS_==low)
      begin
        ts_reg_ = #7 hi;
        wait (AACK_==low);
        Addr_termination = TRUE;
```

```
      end

  //Address termination
  always @(posedge clk)
    if (Addr_termination)
      begin
        #7 ts_reg_ <= 'bz;
        a_reg     <= 'bz;
        ap_reg    <= 'bz;
        tt_reg    <= 'bz;
        tsiz_reg  <= 'bz;
        tbst_reg_ <= 'bz;
        snoop_ignore <= low;
        //insert other addr transfer characteristics here.
        abb_reg_ <= #2 hi;
        abb_reg_ <= #8 'bz;
        AB_Master = FALSE;
        Addr_termination = FALSE;
      end

//DATA BUS ARBITRATION FOR FETCHES

  assign qual_DBG_ = ~(!DBG_ & DBB_);

  always @(posedge clk)
    begin
      if (TA_ == low)
        -> transfer_acknowledged;
    end

  //calculate data parity.  Odd parity, including parity bit.
  always @(data_reg)
    begin
      data_parity_calc[0] <= ~^data_reg[0:7];
      data_parity_calc[1] <= ~^data_reg[8:15];
      data_parity_calc[2] <= ~^data_reg[16:23];
      data_parity_calc[3] <= ~^data_reg[24:31];
      data_parity_calc[4] <= ~^data_reg[32:39];
      data_parity_calc[5] <= ~^data_reg[40:47];
      data_parity_calc[6] <= ~^data_reg[48:55];
      data_parity_calc[7]  = ~^data_reg[56:61];
    end

 always
  begin
    //wait for qualified data bus grant and transfer start.
    wait(qual_DBG_==low & Transfer_start);
    @(posedge clk) //assume data bus mastership
    dbb_reg_ <= #7 low;
```

```
    i = 0;
    while (i<4)
     begin
      @(transfer_acknowledged) //latch beat
       data_reg <= D;
       data_parity_in = DP;
       #2 if (trace) $display("    BIU: %h at %d",data_reg,$time);
       #2 if (data_parity_in != data_parity_calc)
        begin
         $display("BIU: data parity error.");
         $display("  Calculated parity: %b",
               data_parity_calc);
         $display("  Recevied parity:  %b",
               data_parity_in);
         Data_Parity_Error = TRUE;
         i = 4;
        end
       else
        begin
          if (i==0) data_line[  0: 63] = data_reg;
          if (i==1) data_line[ 64:127] = data_reg;
          if (i==2) data_line[128:191] = data_reg;
          if (i==3) data_line[192:255] = data_reg;
          i = i+1;
        end
      end

    Transfer_in_progress <= FALSE;
    Transfer_start <= FALSE;
    dbb_reg_ = #4 hi;
    dbb_reg_ = #8 'bz;
  end


//DATA BUS PROTOCOL FOR SENDS (PRC acting as memory module)
always @(start_send)
 begin
  i = 0;
  while (i<4) begin
   @(posedge clk);
   #1 ta_reg_ = 'bz;
   j = burst_start+i; //j is mod 4
   if (j==0) data_reg = data_line[  0: 63];
   if (j==1) data_reg = data_line[ 64:127];
   if (j==2) data_reg = data_line[128:191];
   if (j==3) data_reg = data_line[192:255];
   d_reg = data_reg;
   #4 dp_reg <= data_parity_calc;
   ta_reg_ <= low;
   i=i+1;
```

```
    end
    send_done <= hi;
    @(posedge clk)
    ta_reg_ <= #7 'bz;
    d_reg <= #7 64'bz;
    dp_reg <= #7 8'bz;
   end


//FINITE STATE MACHINE
always @(negedge HRESET_)
  begin
   if (HRESET_ == low)
     begin
       state <= idle;
       next_state <= idle;
       wait(HRESET_ == hi);
     end
  end

always
  begin
   #2 state = next_state;

   #1 case (state)
     idle: //0
       begin
         upload <= low;
         fetch_done <= low;
         send_done <= low;
         CANX <= low;
         data_line_reg <= 256'bz;
         @(posedge clk) inputs2 = {send,fetch};
         case (inputs2)
           2'b00: next_state = idle;
           2'b01: next_state = fetch1;
           2'b10: next_state = send1;
           2'b11: next_state = idle;  //This should not happen.
         endcase
       end

     fetch1: //1
       begin
         //1. Latch next address.
         NAR[0:26] <= NAR_IN;
         NAR[27:31] <= 5'b0;
         //2. Request Bus
         BR_ <= low;
         Transfer_in_progress <= TRUE;
         @(posedge clk)
```

```
    next_state = fetch2;
  end

fetch2: //2
  begin
    //1. Wait for all data to be received.
    @(posedge clk) input1 = Transfer_in_progress;
    case (input1)
      1'b0: next_state = fetch3;
      1'b1: next_state = fetch2;
    endcase
  end

fetch3: //3
  begin
    //1. Upload the data line.
    data_line_reg <= data_line;
    upload <= hi;
    //2. Assert fetch_done.
    fetch_done <= hi;
    @(posedge clk)
      next_state = idle;
  end

send1: //5
  begin
    //1. Cancel the memory access.
    CANX <= hi;
    //2. Latch burst_start.
    burst_start <= BURSTSTART;
    //3. Download data from the data list.
    download <= hi;
    #5 data_line <= DATALINE;
    @(posedge clk)
      next_state = send2;
  end

send2: //6
  begin
    //1. Send data to CPU
    -> start_send;
    CANX <= low;
    download <= low;
    @(posedge clk) input1 = {send_done};
    case (input1)
      1'b0: next_state = send2;
      1'b1: next_state = idle;
    endcase
  end
```

```
    default:
     begin
      $display("state error in module bus_interface.");
      $display("   state = %b.",state);
     end

   endcase
  end


endmodule
```

H. PREDICTION TEST

```
/*****************************************************************************
* Transaction Sequencer - Prediction Test
* Filename: sequencer4.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
* Revised: 05JAN96
*
* Purpose: This is one in a set of modules which perform a sequence of CPU
* transactions. This sequencer causes a series of CPU operations that provide
* a comprehensive test of the PRC. It demonstrates a majority of the PRC's
* capabilities, showing when the Line Manager selects new lines, when and how
* the Predictor functions, when the CPU starts a read or write and the data
* involved. It shows when the Bus Interface Unit fetches data from memory.
* The DataList reports the flow of data in and out of it. The only significant
* behavior not exercised by this test is the function of the Line Replacement
* Unit when the PRC is full. That is handled with Sequencer #5.
*
* Sequence #4:
*  burst_read  00h
*  burst_read  20h  - PRC should predict 40h and fetch data.
*  burst_read 180h  - PRC should start a new line.
*  burst_read 1A0h  - PRC should predict 1C0h.
*  burst_read  40h  - already in PRC, should predict 60h.
*  burst_write 1C0h - should flush line.
*  burst_read  60h  - already in PRC, predicts 80.
*  burst_read 100h  - PRC should start a new line.
*
* When using this sequencer, set all trace flags to TRUE (except the
* Controller), and run the simulation for 6000 steps.
*
* General Timing instructions for all Sequencers:
* Use an initial block for each transaction. You must ensure that the
* following rules are adhered to:
```

```
*  1. Before the first transaction, use
*        repeat(2)@(posedge clk)
*  2. Before the first line of the second transaction, use
*        wait(ABB_==low);
*        wait(ABB_==hi);
*  3. There can be only two transactions pipelined at a time.  You must ensure
*     manually that the first operation is complete before the third begins.
*     When scheduling the current transaction, look at the transaction before
*     last.  Wait for that TA_ to finish.  Also, wait for the ABB_ from the
*     previous transaction to go high.
*  4. A burst read takes 330 simulation time units = 22 clock cycles.
*
*************************************************************************/
module sequencer(Transfer_size,clk,pp,address,data,line,Transfer_type,
          Transfer_code,need_bus_trigger_,ABB_ );

input clk,ABB_;
output pp,need_bus_trigger_;
output [0:31] address;
output [0:63] data;
output [0:255] line;
output [0:4] Transfer_type;
output [0:2] Transfer_size;
output [0:1] Transfer_code;
reg pp,need_bus_trigger_;
reg [0:31] address;
reg [0:63] data;
reg [0:255] line;
reg [0:4] Transfer_type;
reg [0:2] Transfer_size;
reg [0:1] Transfer_code;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
        FALSE = 1'b0,
        hi    = 1'b1,
        low   = 1'b0;

    parameter //for Transfer_type
          none       = 5'bz,
          write      = 5'b00010,      //02
          write_atomic = 5'b10010,  //12
          read       = 5'b01010,      //0A
          read_atomic  = 5'b11010,  //1A
          burst_write  = 5'b00110,  //06
          burst_read   = 5'b01110,  //0E
          burst_read_atomic = 5'b11110;  //1E
    parameter //for Transfer_code
      data_transfer    = 2'b00,
      touch_load       = 2'b01,
```

```
        instruction_fetch = 2'b10,
        reserved          = 2'b11;

  //initialize signals
  initial
    begin
      pp <= 0;
      address <= 32'bz;
      line <= 256'bz;
    end

  //Perform sequence of transactions
  initial
    begin
      repeat(2)@(posedge clk);
      //BURST READ
      pp <= ~pp;
      address <= 32'h00000000;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
    begin
      wait(ABB_==low);
      wait(ABB_==hi);
      //BURST READ
      pp <= ~pp;
      address <= 32'h00000020;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
    begin
      repeat(75)@(posedge clk);
      //BURST READ
      pp <= ~pp;
      address <= 32'h00000180;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
```

```
    begin
      repeat(100)@(posedge clk);
      //BURST READ
      pp <= ~pp;
      address <= 32'h000001A0;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
    begin
      repeat(150)@(posedge clk);
      //BURST READ
      pp <= ~pp;
      address <= 32'h00000040;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
    begin
      repeat(200)@(posedge clk);
      //BURST WRITE
      pp <= ~pp;
      address <= 32'h000001C0;
      Transfer_type <= burst_write;
      Transfer_code <= data_transfer;
      line <= {64'h7777777777777777, 64'h8888888888888888,
            64'h1111111111111111, 64'h3333333333333333};
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
    begin
      repeat(225)@(posedge clk);
      //BURST READ
      pp <= ~pp;
      address <= 32'h00000060;
      Transfer_type <= burst_read;
      Transfer_code <= data_transfer;
      need_bus_trigger_ <= #4 low;
      need_bus_trigger_ <= #6 hi;
    end

  initial
```

```
  begin
    repeat(250)@(posedge clk);
    //BURST READ
    pp <= ~pp;
    address <= 32'h00000100;
    Transfer_type <= burst_read;
    Transfer_code <= data_transfer;
    need_bus_trigger_ <= #4 low;
    need_bus_trigger_ <= #6 hi;
  end

endmodule
```

I. PREDICTION TEST RESULTS

```
Host command: verilog
Command arguments:
  -f verilog_arguments
     bus_interface.v
     prc.v
     snooper.v
     controller.v
     datalist.v
     line_mgr.v
     predictor.v
     testbench.v
     arbiter.v
     cpu.v
     memory.v
     sequencer5.v

VERILOG-XL 2.1.2 log file created Feb  2, 1996  13:14:29
VERILOG-XL 2.1.2   Feb  2, 1996  13:14:29

Copyright (c) 1994 Cadence Design Systems, Inc.  All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1994 UNIX Systems Laboratories, Inc.  Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
```

Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted Rights at 48 CFR 52.227-19, as applicable.

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at 1-800-CADENC2 or send email to crc_customers@cadence.com

For more information on Cadence's Verilog-XL product line send email to talkverilog@cadence.com

Compiling source file "bus_interface.v"
Compiling source file "prc.v"
Compiling source file "snooper.v"
Compiling source file "controller.v"
Compiling source file "datalist.v"
Compiling source file "line_mgr.v"
Compiling source file "predictor.v"
Compiling source file "testbench.v"
Compiling source file "arbiter.v"
Compiling source file "cpu.v"
Compiling source file "memory.v"
Compiling source file "sequencer5.v"
Highest level modules:
testbench

Line_mgr selected new ActiveLine = 0 at $d 5
Line_mgr selected new ActiveLine = 1 at $d 1162
Line_mgr selected new ActiveLine = 2 at $d 2287
Line_mgr selected new ActiveLine = 3 at $d 3412
Line_mgr selected new ActiveLine = 4 at $d 4537
Line_mgr selected new ActiveLine = 5 at $d 5662
Line_mgr selected new ActiveLine = 6 at $d 6787
Line_mgr selected new ActiveLine = 7 at $d 7912
Line_mgr selected new ActiveLine = 8 at $d 9037
Line_mgr selected new ActiveLine = 9 at $d 10162
Line_mgr selected new ActiveLine = 10 at $d 11287
Line_mgr selected new ActiveLine = 11 at $d 12412
Line_mgr selected new ActiveLine = 12 at $d 13537
Line_mgr selected new ActiveLine = 13 at $d 14662
Line_mgr selected new ActiveLine = 14 at $d 15787
Line_mgr selected new ActiveLine = 15 at $d 16912
Line_mgr selected new ActiveLine = 16 at $d 18037
Line_mgr selected new ActiveLine = 17 at $d 19162
Line_mgr selected new ActiveLine = 18 at $d 20287
Line_mgr selected new ActiveLine = 19 at $d 21412
Line_mgr selected new ActiveLine = 20 at $d 22537

Line_mgr selected new ActiveLine = 21 at $d 23662
Line_mgr selected new ActiveLine = 22 at $d 24787
Line_mgr selected new ActiveLine = 23 at $d 25912
Line_mgr selected new ActiveLine = 24 at $d 27037
Line_mgr selected new ActiveLine = 25 at $d 28162
Line_mgr selected new ActiveLine = 26 at $d 29287
Line_mgr selected new ActiveLine = 27 at $d 30412
Line_mgr selected new ActiveLine = 28 at $d 31537
Line_mgr selected new ActiveLine = 29 at $d 32662
Line_mgr selected new ActiveLine = 30 at $d 33787
Line_mgr selected new ActiveLine = 31 at $d 34912
Line_mgr selected new ActiveLine = 32 at $d 36037
Line_mgr selected new ActiveLine = 33 at $d 37162
Line_mgr selected new ActiveLine = 34 at $d 38287
Line_mgr selected new ActiveLine = 35 at $d 39412
Line_mgr selected new ActiveLine = 36 at $d 40537
Line_mgr selected new ActiveLine = 37 at $d 41662
Line_mgr selected new ActiveLine = 38 at $d 42787
Line_mgr selected new ActiveLine = 39 at $d 43912
Line_mgr selected new ActiveLine = 40 at $d 45037
Line_mgr selected new ActiveLine = 41 at $d 46162
Line_mgr selected new ActiveLine = 42 at $d 47287
Line_mgr selected new ActiveLine = 43 at $d 48412
Line_mgr selected new ActiveLine = 44 at $d 49537
Line_mgr selected new ActiveLine = 45 at $d 50662
Line_mgr selected new ActiveLine = 46 at $d 51787
Line_mgr selected new ActiveLine = 47 at $d 52912
Line_mgr selected new ActiveLine = 48 at $d 54037
Line_mgr selected new ActiveLine = 49 at $d 55162
Line_mgr selected new ActiveLine = 50 at $d 56287
Line_mgr selected new ActiveLine = 51 at $d 57412
Line_mgr selected new ActiveLine = 52 at $d 58537
Line_mgr selected new ActiveLine = 53 at $d 59662
Line_mgr selected new ActiveLine = 54 at $d 60787
Line_mgr selected new ActiveLine = 55 at $d 61912
Line_mgr selected new ActiveLine = 56 at $d 63037
Line_mgr selected new ActiveLine = 57 at $d 64162
Line_mgr selected new ActiveLine = 58 at $d 65287
Line_mgr selected new ActiveLine = 59 at $d 66412
Line_mgr selected new ActiveLine = 60 at $d 67537
Line_mgr selected new ActiveLine = 61 at $d 68662
Line_mgr selected new ActiveLine = 62 at $d 69787
Line_mgr selected new ActiveLine = 63 at $d 70912
Line_mgr selected new ActiveLine = 64 at $d 72037
Line_mgr selected new ActiveLine = 65 at $d 73162
Line_mgr selected new ActiveLine = 66 at $d 74287
Line_mgr selected new ActiveLine = 67 at $d 75412
Line_mgr selected new ActiveLine = 68 at $d 76537
Line_mgr selected new ActiveLine = 69 at $d 77662
Line_mgr selected new ActiveLine = 70 at $d 78787

Line_mgr selected new ActiveLine = 71 at $d 79912
Line_mgr selected new ActiveLine = 72 at $d 81037
Line_mgr selected new ActiveLine = 73 at $d 82162
Line_mgr selected new ActiveLine = 74 at $d 83287
Line_mgr selected new ActiveLine = 75 at $d 84412
Line_mgr selected new ActiveLine = 76 at $d 85537
Line_mgr selected new ActiveLine = 77 at $d 86662
Line_mgr selected new ActiveLine = 78 at $d 87787
Line_mgr selected new ActiveLine = 79 at $d 88912
Line_mgr selected new ActiveLine = 80 at $d 90037
Line_mgr selected new ActiveLine = 81 at $d 91162
Line_mgr selected new ActiveLine = 82 at $d 92287
Line_mgr selected new ActiveLine = 83 at $d 93412
Line_mgr selected new ActiveLine = 84 at $d 94537
Line_mgr selected new ActiveLine = 85 at $d 95662
Line_mgr selected new ActiveLine = 86 at $d 96787
Line_mgr selected new ActiveLine = 87 at $d 97912
Line_mgr selected new ActiveLine = 88 at $d 99037
Line_mgr selected new ActiveLine = 89 at $d 100162
Line_mgr selected new ActiveLine = 90 at $d 101287
Line_mgr selected new ActiveLine = 91 at $d 102412
Line_mgr selected new ActiveLine = 92 at $d 103537
Line_mgr selected new ActiveLine = 93 at $d 104662
Line_mgr selected new ActiveLine = 94 at $d 105787
Line_mgr selected new ActiveLine = 95 at $d 106912
Line_mgr selected new ActiveLine = 96 at $d 108037
Line_mgr selected new ActiveLine = 97 at $d 109162
Line_mgr selected new ActiveLine = 98 at $d 110287
Line_mgr selected new ActiveLine = 99 at $d 111412
Line_mgr selected new ActiveLine = 100 at $d 112537
Line_mgr selected new ActiveLine = 101 at $d 113662
Line_mgr selected new ActiveLine = 102 at $d 114787
Line_mgr selected new ActiveLine = 103 at $d 115912
Line_mgr selected new ActiveLine = 104 at $d 117037
Line_mgr selected new ActiveLine = 105 at $d 118162
Line_mgr selected new ActiveLine = 106 at $d 119287
Line_mgr selected new ActiveLine = 107 at $d 120412
Line_mgr selected new ActiveLine = 108 at $d 121537
Line_mgr selected new ActiveLine = 109 at $d 122662
Line_mgr selected new ActiveLine = 110 at $d 123787
Line_mgr selected new ActiveLine = 111 at $d 124912
Line_mgr selected new ActiveLine = 112 at $d 126037
Line_mgr selected new ActiveLine = 113 at $d 127162
Line_mgr selected new ActiveLine = 114 at $d 128287
Line_mgr selected new ActiveLine = 115 at $d 129412
Line_mgr selected new ActiveLine = 116 at $d 130537
Line_mgr selected new ActiveLine = 117 at $d 131662
Line_mgr selected new ActiveLine = 118 at $d 132787
Line_mgr selected new ActiveLine = 119 at $d 133912
Line_mgr selected new ActiveLine = 120 at $d 135037

```
Line_mgr selected new ActiveLine = 121 at $d          136162
Line_mgr selected new ActiveLine = 122 at $d          137287
Line_mgr selected new ActiveLine = 123 at $d          138412
Line_mgr selected new ActiveLine = 124 at $d          139537
Line_mgr selected new ActiveLine = 125 at $d          140662
Line_mgr selected new ActiveLine = 126 at $d          141787
Line_mgr selected new ActiveLine = 127 at $d          142912
Line_mgr selected new ActiveLine =  0 at $d          145162
Line_mgr selected new ActiveLine =  1 at $d          146287
Line_mgr selected new ActiveLine =  2 at $d          147412
Line_mgr selected new ActiveLine =  3 at $d          148537
L122 "testbench.v": $finish at simulation time 152010
31769681 simulation events + 8392 accelerated events
CPU time: 1.0 secs to compile + 0.9 secs to link + 116.2 secs in simulation
End of VERILOG-XL 2.1.2  Feb  2, 1996  13:16:34
```

J. LINE REPLACEMENT TEST

```
/*******************************************************************************
* Transaction Sequencer - Line Replacement Test
* Filename: sequencer5.v
* Author: Joseph R. Robert, Jr.
* Date:   05JAN96
* Revised: 05JAN96
*
* Purpose:  This is one in a set of modules which perform a sequence of CPU
* transactions.  This Sequencer causes a series of CPU operations which will
* quickly fill the PRC.  This will test the Line Replacement Unit's behavior
* when it needs to start replacing previously used lines.
*
* Sequence #5:
*
* for i = 0 to 132,
*   burst_read  i00h  - PRC should switch to new line i.
*   burst_read  i20h  - PRC should predict i40h, and store data in line i.
* next i
*
* When using this sequencer, set all trace flags to FALSE, except for the Line
* Manager, and run the simulation for 152000 steps.
*
* General Timing instructions for all Sequencers:
* Use an initial block for each transaction.  You must ensure that the
* following rules are adhered to:
*   1. Before the first transaction, use
*         repeat(2)@(posedge clk)
*   2. Before the first line of the second transaction, use
*         wait(ABB_==low);
```

```
*        wait(ABB_==hi);
*  3. There can be only two transactions pipelined at a time.  You must ensure
*     manually that the first operation is complete before the third begins.
*     When scheduling the current transaction, look at the transaction before
*     last.  Wait for that TA_ to finish.  Also, wait for the ABB_ from the
*     previous transaction to go high.
*  4. A burst read takes 330 simulation time units = 22 clock cycles.
*
*******************************************************************************/
module sequencer(Transfer_size,clk,pp,address,data,line,Transfer_type,
            Transfer_code,need_bus_trigger_,ABB_ );

input clk,ABB_;
output pp,need_bus_trigger_;
output [0:31] address;
output [0:63] data;
output [0:255] line;
output [0:4] Transfer_type;
output [0:2] Transfer_size;
output [0:1] Transfer_code;
reg pp,need_bus_trigger_;
reg [0:31] address;
reg [0:63] data;
reg [0:255] line;
reg [0:4] Transfer_type;
reg [0:2] Transfer_size;
reg [0:1] Transfer_code;

//declare variables, constants, parameters
  parameter TRUE  = 1'b1,
          FALSE = 1'b0,
          hi    = 1'b1,
          low   = 1'b0;

    parameter //for Transfer_type
          none       = 5'bz,
          write      = 5'b00010,      //02
          write_atomic = 5'b10010,  //12
          read       = 5'b01010,      //0A
          read_atomic  = 5'b11010,  //1A
          burst_write  = 5'b00110,  //06
          burst_read   = 5'b01110,  //0E
          burst_read_atomic = 5'b11110;  //1E
    parameter //for Transfer_code
      data_transfer    = 2'b00,
      touch_load       = 2'b01,
      instruction_fetch = 2'b10,
      reserved         = 2'b11;

  //Other internal control signals
```

```
  reg [0:7] i; // counter

//initialize signals
initial
  begin
    pp <= 0;
    address <= 32'bz;
    line <= 256'bz;
  end

//Perform sequence of transactions
initial
  begin
    repeat(2)@(posedge clk);
    //BURST READ
    pp <= ~pp;
    address <= 32'h00000000;
    Transfer_type <= burst_read;
    Transfer_code <= data_transfer;
    need_bus_trigger_ <= #4 low;
    need_bus_trigger_ <= #6 hi;
  end

initial
  begin
    wait(ABB_==low);
    wait(ABB_==hi);
    //BURST READ
    pp <= ~pp;
    address <= 32'h00000020;
    Transfer_type <= burst_read;
    Transfer_code <= data_transfer;
    need_bus_trigger_ <= #4 low;
    need_bus_trigger_ <= #6 hi;
  end

initial
  begin
    repeat(25)@(posedge clk);
    for (i=1; i<=132; i=i+1)
      begin

        repeat(50)@(posedge clk);
        //BURST READ
        pp <= ~pp;
        address <= {12'b0, i, 12'b0};
        Transfer_type <= burst_read;
        Transfer_code <= data_transfer;
        need_bus_trigger_ <= #4 low;
        need_bus_trigger_ <= #6 hi;
```

```
        repeat(25)@(posedge clk);
        //BURST READ
        pp <= ~pp;
        address <= {12'b0, i, 12'h020};
        Transfer_type <= burst_read;
        Transfer_code <= data_transfer;
        need_bus_trigger_ <= #4 low;
        need_bus_trigger_ <= #6 hi;

      end

  end

endmodule
```

K. LINE REPLACEMENT TEST RESULTS

```
Host command: verilog
Command arguments:
  -f verilog_arguments
     bus_interface.v
     prc.v
     snooper.v
     controller.v
     datalist.v
     line_mgr.v
     predictor.v
     testbench.v
     arbiter.v
     cpu.v
     memory.v
     sequencer4.v

VERILOG-XL 2.1.2 log file created Feb  2, 1996  13:22:22
VERILOG-XL 2.1.2  Feb  2, 1996  13:22:22

Copyright (c) 1994 Cadence Design Systems, Inc.  All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1994 UNIX Systems Laboratories, Inc.  Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND
```

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at
1-800-CADENC2 or send email to crc_customers@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkverilog@cadence.com

```
Compiling source file "bus_interface.v"
Compiling source file "prc.v"
Compiling source file "snooper.v"
Compiling source file "controller.v"
Compiling source file "datalist.v"
Compiling source file "line_mgr.v"
Compiling source file "predictor.v"
Compiling source file "testbench.v"
Compiling source file "arbiter.v"
Compiling source file "cpu.v"
Compiling source file "memory.v"
Compiling source file "sequencer4.v"
Highest level modules:
testbench

Line_mgr selected new ActiveLine =  0 at $d            5
CPU started read from address 00000000 at time           45.
   CPU read: 0001020304050607 at          181
   CPU read: 08090a0b0c0d0e0f at          241
   CPU read: 1011121314151617 at          301
   CPU read: 18191a1b1c1d1e1f at          361
CPU started read from address 00000020 at time           390.
BIU started read from address 00000040 at time           412.
   CPU read: 2021222324252627 at          496
   CPU read: 28292a2b2c2d2e2f at          556
   CPU read: 3031323334353637 at          616
   CPU read: 38393a3b3c3d3e3f at          676
   BIU: 4041424344454647 at          812
   BIU: 48494a4b4c4d4e4f at          872
   BIU: 5051525354555657 at          932
   BIU: 58595a5b5c5d5e5f at          992
DATALIST uploaded this data into line 00 at time          1008.
   404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f
CPU started read from address 00000180 at time          1140.
```

```
Line_mgr selected new ActiveLine =  1 at $d          1162
   CPU read: 0001020304050607 at            1276
   CPU read: 08090a0b0c0d0e0f at            1336
   CPU read: 1011121314151617 at            1396
   CPU read: 18191a1b1c1d1e1f at            1456
CPU started read from address 000001a0 at time          1515.
   CPU read: 2021222324252627 at            1651
BIU started read from address 000001c0 at time          1657.
   CPU read: 28292a2b2c2d2e2f at            1711
   CPU read: 3031323334353637 at            1771
   CPU read: 38393a3b3c3d3e3f at            1831
   BIU: 4041424344454647 at            1967
   BIU: 48494a4b4c4d4e4f at            2027
   BIU: 5051525354555657 at            2087
   BIU: 58595a5b5c5d5e5f at            2147
DATALIST uploaded this data into line 01 at time          2163.
   404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f
CPU started read from address 00000040 at time          2265.
Line_mgr selected new ActiveLine =  0 at $d          2287
DATALIST downloaded this data from line 00 at time          2313.
   404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f
   CPU read: 4041424344454647 at            2356
   CPU read: 48494a4b4c4d4e4f at            2371
   CPU read: 5051525354555657 at            2386
   CPU read: 58595a5b5c5d5e5f at            2401
BIU started read from address 00000060 at time          2482.
   BIU: 6061626364656667 at            2627
   BIU: 68696a6b6c6d6e6f at            2687
   BIU: 7071727374757677 at            2747
   BIU: 78797a7b7c7d7e7f at            2807
DATALIST uploaded this data into line 00 at time          2823.
   606162636465666768696a6b6c6d6e6f707172737475767778797a7b7c7d7e7f
CPU started write to address 000001c0 at time          3007.
   CPU write beat 1: 7777777777777777 at          3022
Line_mgr selected new ActiveLine =  1 at $d          3037
Line manager flushed line  1 at time          3048.
   CPU write beat 2: 8888888888888888 at          3158
   CPU write beat 3: 1111111111111111 at          3218
   CPU write beat 4: 3333333333333333 at          3278
CPU started read from address 00000060 at time          3390.
Line_mgr selected new ActiveLine =  0 at $d          3412
DATALIST downloaded this data from line 00 at time          3438.
   606162636465666768696a6b6c6d6e6f707172737475767778797a7b7c7d7e7f
   CPU read: 6061626364656667 at            3481
   CPU read: 68696a6b6c6d6e6f at            3496
   CPU read: 7071727374757677 at            3511
   CPU read: 78797a7b7c7d7e7f at            3526
BIU started read from address 00000080 at time          3607.
   BIU: 0001020304050607 at            3752
   BIU: 08090a0b0c0d0e0f at            3812
```

```
   BIU: 1011121314151617 at          3872
   BIU: 18191a1b1c1d1e1f at          3932
CPU started read from address 00000100 at time          3945.
DATALIST uploaded this data into line 00 at time          3948.
   000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Line_mgr selected new ActiveLine =  2 at $d          3982
   CPU read: 0001020304050607 at          4066
   CPU read: 08090a0b0c0d0e0f at          4126
   CPU read: 1011121314151617 at          4186
   CPU read: 18191a1b1c1d1e1f at          4246
L123 "testbench.v": $finish at simulation time 6010
1661039 simulation events + 265 accelerated events
CPU time: 0.8 secs to compile + 0.8 secs to link + 5.0 secs in simulation
End of VERILOG-XL 2.1.2  Feb  2, 1996  13:22:29
```

APPENDIX D. PRC STRUCTURE FILES

This appendix contains the Verilog files for the final hardware design. They include the Verilog structural models of the PRC and the testing results. The files are located on the ECE system at *home5/robert/thesis/epoch/verilog*.

A. PRC

```
/*****************************************************************************
* Predictive Read Cache
* Filename: prc.v
* Author: Joseph R. Robert, Jr.
* Date:  02OCT95
* Revised: 14MAR96
*
 Purpose: This module emulates the predictive read cache, connecting all the parts.
*
*****************************************************************************/
module prc(HRESET_,clk,BG_,DBG_,BR_,CANX,D,A,DP,TT,AP,TSIZ,TC,ABB_,AACK_,TS_,
      TBST_,DBB_,TA_,DPE_);

// epoch set_attribute FIXEDBLOCK = 1

input HRESET_,clk,BG_,DBG_;
output BR_,CANX;
inout [63:0] D;
inout [31:0] A;
inout [7:0] DP;
inout [4:0] TT;
inout [3:0] AP;
inout [2:0] TSIZ;
inout [1:0] TC;
inout ABB_,AACK_,TS_,TBST_,DBB_,TA_,DPE_;

wire [255:0] DATALINE;
wire [26:0] CAR,NAR,MRMA;
wire [6:0] ActiveLine;
wire [1:0] BURSTSTART;
wire fetch_done,fetch_abort,send_done,read,write,hit,line_empty,
   snoop_ignore,upload,download,BR_,CANX;
```

```
tri [63:0] D;
tri [31:0] A;
tri [7:0] DP;
tri [4:0] TT;
tri [3:0] AP;
tri [2:0] TSIZ;
tri [1:0] TC;
tri ABB_,AACK_,TS_,TBST_,DBB_,TA_,DPE_;


//Connect parts which have been converted to hardware.

  // epoch pre_compiled predictor
  predictor PRE1(MRMA,CAR[25:0],predict,NAR,HRESET_);

  // epoch pre_compiled line_mgr
  line_mgr LM1(CAR,NAR,HRESET_,a_select,test,fetch_done,flush,store,
    new_replace,MRMA,ActiveLine,line_empty,hit,clk);

  // epoch pre_compiled datalist
  datalist DL1(DATALINE,ActiveLine,upload,download);

  // epoch pre_compiled snooper
  snooper SN1(A,AP,TT,TC,TS_,snoop_ignore,hold,clk,CAR,BURSTSTART,
    read,write,HRESET_);

  // epoch pre_compiled bus_interface
  bus_interface BIU1(NAR,BURSTSTART,BG_,AACK_,DBG_,send,fetch,
    clk,BR_,upload,download,fetch_done,fetch_abort,
    send_done,CANX,snoop_ignore,DATALINE,D,A,AP,DP,DPE_,
    TT,TSIZ,TC,ABB_,TS_,TBST_,DBB_,TA_,HRESET_);

  // epoch pre_compiled controller
  controller CON1(HRESET_,read,write,hit,send_done,fetch_done,fetch_abort,
    line_empty,a_select,test,predict,store,
    flush,send,hold,new_replace,fetch,clk);


endmodule
```

B. CONTROLLER

```
/*****************************************************************************
* CONTROLLER
* Filename: controller.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
```

```
* Revised: 20MAR96
*
        Purpose: This module is a Finite State Machine which coordinates the actions of all the other functional
blocks of the PRC. All control signals are synchronous with the system clock. HRESET_ causes the Controller
to go to the IDLE state. The state diagram and state output tables give more details.
        Of significance are the wait states added to the state diagram of the behavioral model. These changes are
highlighted in the State Output Table. The changes were required by the Line Manager, in which there is a
significant propagation delay for the addresses. This is described in more detail in the Line Manager section of this
chapter. This is a prime candidate for future work to improve the PRC's design.
*
*********************************************************************************/
module controller (HRESET_,read,write,hit,send_done,fetch_done,fetch_abort,
  line_empty,a_select,test,predict,store,
  flush,send,hold,new_replace,fetch,clk);

// epoch set_attribute FIXEDBLOCK = 1

input HRESET_,read,write,hit,send_done,fetch_done,fetch_abort,line_empty,clk;
output a_select,test,predict,store,flush,send,hold,new_replace,fetch;

reg a_select,test,predict,store,flush,send,hold,new_replace,fetch;


//Finite State Machine

parameter // epoch enum stat
  idle        = 5'd0,
  test_car_r  = 5'd1,
  send_data   = 5'd2,
  test_nar    = 5'd3,
  fetch_data  = 5'd4,
  is_line_empty = 5'd5,
  predict_na  = 5'd6,
  store_car   = 5'd7,
  test_car_w  = 5'd8,
  flush_line  = 5'd9,
  wait_a      = 5'd10,
  wait_b      = 5'd11,
  wait_c      = 5'd12,
  wait_d      = 5'd13,
  wait_e      = 5'd14,
  wait_f      = 5'd15,
  wait_g      = 5'd16,
  wait_h      = 5'd17,
  wait_i      = 5'd18,
  dc_state    = 5'bx;

reg [4:0] /* epoch enum stat */ state, next_state;
reg  a_select,fetch,flush,hold,new_replace,predict,send,store,test;
```

```
always @(posedge clk or negedge HRESET_)
 begin
  if (!HRESET_)
   state = idle;
  else
   state = next_state;
 end

always @(state or read or write or hit or send_done or line_empty or
      fetch_done or fetch_abort)
 begin

  //default values
      a_select   = 1'b0; //CAR
      fetch      = 1'b0;
      flush      = 1'b0;
      hold       = 1'b0;
      new_replace = 1'b0;
      predict    = 1'b0;
      send       = 1'b0;
      store      = 1'b0;
      test       = 1'b0;

  case (state)

   idle: //0
    begin
     if     (read == 1'b0 & write == 1'b0) next_state = idle;
     else if (read == 1'b0 & write == 1'b1) next_state = wait_d;
     else if (read == 1'b1) next_state = wait_a;
     else next_state = dc_state;
    end

   wait_a: //10
    begin
     hold      = 1'b1;
     next_state = wait_b;
    end

   wait_b: //11
    begin
     hold      = 1'b1;
     next_state = wait_c;
    end

   wait_c: //12
    begin
     hold      = 1'b1;
     next_state = test_car_r;
    end
```

```
 wait_d: //13
  begin
   hold      = 1'b1;
   next_state = wait_e;
  end

 wait_e: //14
  begin
   hold      = 1'b1;
   next_state = wait_f;
  end

 wait_f: //15
  begin
   hold      = 1'b1;
   next_state = test_car_w;
  end

 test_car_r: //1
  begin
   test      = 1'b1;
   hold      = 1'b1;
   if (hit)
    next_state = send_data;
   else next_state = is_line_empty;
  end

 send_data: //2
  begin
   a_select   = 1'b1; //NAR
   predict    = 1'b1;
   send       = 1'b1;
   hold       = 1'b1;
   if (send_done)
    next_state = test_nar;
   else next_state = send_data;
  end

 test_nar: //3
  begin
   a_select   = 1'b1; //NAR
   test       = 1'b1;
   hold       = 1'b1;
   if ({hit,read,write} == 3'b000) next_state = fetch_data;
   else  next_state = idle;
  end

 fetch_data: //4
  begin
   a_select   = 1'b1; //NAR
```

```
    hold      = 1'b1;
    fetch     = 1'b1;
    if ({fetch_done,fetch_abort} == 2'b00) next_state = fetch_data;
    else next_state = idle;
   end

 is_line_empty: //5
   begin
    hold      = 1'b1;
    if (line_empty)
     next_state = store_car;
    else next_state = predict_na;
   end

 predict_na: //6
   begin
    a_select   = 1'b1;
    predict    = 1'b1;
    hold       = 1'b1;
    new_replace = 1'b1;
    next_state = wait_g;
   end

 wait_g: //16
   begin
    a_select   = 1'b1; //NAR
    hold       = 1'b1;
    next_state = wait_h;
   end

 wait_h: //17
   begin
    a_select   = 1'b1; //NAR
    hold       = 1'b1;
    next_state = wait_i;
   end

 wait_i: //18
   begin
    a_select   = 1'b1; //NAR
    hold       = 1'b1;
    next_state = test_nar;
   end

 store_car: //7
   begin
    store      = 1'b1;
    hold       = 1'b1;
    next_state = idle;
   end
```

```
    test_car_w: //8
     begin
      test      = 1'b1;
      hold      = 1'b1;
      if (hit)
       next_state = flush_line;
      else next_state = idle;
     end

    flush_line: //9
     begin
      flush     = 1'b1;
      hold      = 1'b1;
      next_state = idle;
     end

    default:
     begin
      next_state = dc_state;
     end

  endcase
 end

endmodule
```

C. SNOOPER

```
/*********************************************************************************
* SNOOPER
* Filename: snooper.v
* Author: Joseph R. Robert, Jr.
* Date:   21DEC95
* Revised: 06MAR96
*
```

Purpose: This module watches the system bus activity, and makes appropriate reports to the PRC Controller.

If the transaction is a data burst read or any kind of write, and if the address parity is correct, then the read or write signal is asserted as appropriate, and the address is placed in the CAR. The snoop_ignore signal tells this unit to ignore the current transaction, because it was initiated by the Bus Interface Unit. The snoop_ignore signal must be asserted concurrently with the transfer attributes. Reads that are not burst or data related are ignored by the PRC. The CAR is updated only on transactions relevant to the PRC.

Due to the two-stage pipelining capability of the PowerPC, with respect to memory accesses, a second address tenure can occur shortly after the first, well before the first data tenure is complete. To compensate for this, the read and write outputs of the Snooper will remain exerted until acknowledged by the Controller with hold. The rising edge of hold indicates that the read or write signal was received by the Controller. The Snooper can then

negate these signals, but must leave CAR alone until hold is negated. After hold is negated, CAR can be updated to the new address.

In Stage 0, the transfer attributes are latched in registers. Combinational logic determines if these tranfer attributes represent a valid read or a valid write, and if the parity address parity is correct. If the transaction is valid, and one that the PRC is interested in, then Stage 0 raises a transaction_waiting signal.

A Finite State Machine in Stage One sits in the IDLE state until it receives that signal. Then it latches the signals needed from Stage 0, resets the transaction_waiting signal, and then waits for the hold signal to go low. A high hold signal indicates that the PRC is not done with the previous transaction. Once hold goes low, the read and write flags are set according to the type of the current transaction. Also, the input address is stored in the Current Address Register. The FSM then waits for the rising edge of hold before returning to the IDLE state where it can check if there is another transaction waiting.

```
**************************************************************************/
module snooper (A,AP,TT,TC,TS_,snoop_ignore,hold,clk,CAR,BURSTSTART,
        read_flag,write_flag,HRESET_);

// epoch set_attribute FIXEDBLOCK = 1

input [31:0] A;
input [3:0] AP;
input [4:0] TT;
input [1:0] TC;
input TS_,snoop_ignore,hold,clk,HRESET_;
output [26:0] CAR;
output [1:0] BURSTSTART;
output read_flag,write_flag;

wire [31:0] address0;
wire [28:0] address1;
wire [26:0] CAR;
wire [4:0] TransferType;
wire [3:0] addr_parity;
wire [1:0] BURSTSTART,TransferCode;
wire car_latch,flag_reset_,hold_,ignore,latch0,latch1,parity_error,
   read_flag,read_set_,TS,transaction_waiting,tw_set,tw_reset_,
   valid_op,valid_read0,valid_read1,valid_write0,valid_write1,
   w1,w2,w3,w5,w6,w7,write_flag_,write_set_,prelatch0;


//STAGE 0

//Stage 0 latches
stdinv TS_INV (TS_,TS);
stddff TS_Latch (.CLK(clk),.D(TS),.Q(prelatch0));
stdbuf Latch0Buffer (.IN0(prelatch0),.Y(latch0));
dff #(32,0,"AUTO","1") AddressLatch0 (.CLK(latch0),.D(A),.Q(address0));
dff #( 4,0,"AUTO","1") AddrParityLatch (.CLK(latch0),.D(AP),.Q(addr_parity));
dff #( 5,0,"AUTO","1") TransferTypeLatch (.CLK(latch0),.D(TT),.Q(TransferType));
dff #( 2,0,"AUTO","1") TransferCodeLatch (.CLK(latch0),.D(TC),.Q(TransferCode));
stddff IgnoreLatch (.CLK(latch0),.D(snoop_ignore),.Q(ignore));
```

```
//Odd parity checker
parityo_chk32
    OddParityChecker (.D(address0),.PIN(addr_parity),.ERROR(parity_error));

//Read checker
stdnor2 NOR_C (TransferCode[1],TransferCode[0],w1);
stdinv  INV_F (TransferType[0],TransferType0_);
stdand4 AND_D (TransferType[3],TransferType[2],TransferType[1],TransferType0_,
        w2);
stdand2 AND_E (w1,w2,valid_read0);

//Write checker
stdinv  INV_J  (TransferType[3],TransferType3_);
stdnand2 NAND_H (TransferType[4],TransferType[2],w3);
stdand4  AND_G  (TransferType3_,TransferType[1],TransferType0_,w3,valid_write0);

//Transaction checker
stdnor2 NOR_L (valid_write0,valid_read0,w5);
stdnor3 NOR_M (parity_error,w5,ignore,valid_transaction);

//Transaction Waiting Latch
stdand2 TW_SetAND  (latch0,valid_transaction,tw_set);
stdand2  TW_ResetAND (tw_reset1_,HRESET_,tw_reset_);
stdlatch_c TW_Latch
      (.D(tw_set),.CLR(tw_reset_),.EN(latch0),.Q(transaction_waiting));

//STAGE 1

//Stage 1 latches
dff #(29,0,"AUTO","1")
  AddressLatch1 (.CLK(latch1),.D(address0[31:3]),.Q(address1));
stddff ValidReadLatch1  (.CLK(latch1),.D(valid_read0),.Q(valid_read1));
stddff ValidWriteLatch1 (.CLK(latch1),.D(valid_write0),.Q(valid_write1));

//read and write flags
stdinv HOLD_INV (hold,hold_);
stdand2 FLAG_RESET_AND (.IN0(hold_),.IN1(HRESET_),.Y(flag_reset_));
stddff_c ReadFlagLatch
      (.CLK(flag_clk),.CLR(flag_reset_),.D(valid_read1 ),.Q(read_flag ));
stddff_c WriteFlagLatch
      (.CLK(flag_clk),.CLR(flag_reset_),.D(valid_write1),.Q(write_flag));

//Current Address Register
dff #(29,0,"AUTO","1")
  CA_Register (.CLK(car_latch),.D(address1),.Q({CAR,BURSTSTART}));


//FINITE STATE MACHINE

parameter // epoch enum stat
```

```
  IDLE        = 3'd0,
  LATCH       = 3'd1,
  OUTPUTS     = 3'd2,
  WAIT_FOR_HOLD = 3'd3,
  WAIT_FOR_NOT_HOLD = 3'd4,
  dc_state    = 3'bxx;

reg [2:0] /* epoch enum stat */ state, next_state;
reg latch1,tw_reset1_,flag_clk,car_latch;

always @(posedge clk or negedge HRESET_)
 begin
  if (!HRESET_)
   state = IDLE;
  else
   state = next_state;
 end

always @(state or transaction_waiting or hold)
 begin

  //default values
  latch1    = 1'b0;
  tw_reset1_ = 1'b1;
  flag_clk  = 1'b0;
  car_latch  = 1'b0;

  case (state)

   IDLE: begin
        if (transaction_waiting)
         next_state = LATCH;
        else next_state = IDLE;
       end

   LATCH: begin
         latch1    = 1'b1;
         tw_reset1_ = 1'b0;
         if (hold)
          next_state = WAIT_FOR_NOT_HOLD;
         else next_state = OUTPUTS;
        end

   WAIT_FOR_NOT_HOLD: begin
          if (hold)
           next_state = WAIT_FOR_NOT_HOLD;
          else next_state = OUTPUTS;
        end

   OUTPUTS: begin
```

```
            flag_clk  = 1'b1;
            car_latch = 1'b1;
            next_state = WAIT_FOR_HOLD;
          end

      WAIT_FOR_HOLD: begin
            if (hold)
              next_state = IDLE;
            else next_state = WAIT_FOR_HOLD;
          end

      default: begin
            next_state = dc_state;
           end

    endcase

  end

endmodule
```

1. Thirty-Two-Input, Odd-Parity Checker

```
/*****************************************************************************
* ODD PARITY CHECKER
* Filename: parityo_chk32.v
* Author: Joseph R. Robert, Jr.
* Date:   12FEB96
* Revised: 12FEB96
*
 Purpose: This module checks the parity of the input data, comparing it to the input parity.  Parity is odd including
the parity bit.

*****************************************************************************/
module parityo_chk32 (D,PIN,ERROR);

input [31:0] D;
input [3:0]  PIN;
output ERROR;

wire ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR;

parityco #(8,0,"AUTO","1")
      parity_group_0 (.D(D[ 7: 0]),.PIN(PIN[0]),.ERROR(ERROR_0));
parityco #(8,0,"AUTO","1")
      parity_group_1 (.D(D[15: 8]),.PIN(PIN[1]),.ERROR(ERROR_1));
parityco #(8,0,"AUTO","1")
```

```
    parity_group_2 (.D(D[23:16]),.PIN(PIN[2]),.ERROR(ERROR_2));
parityco #(8,0,"AUTO","1")
    parity_group_3 (.D(D[31:24]),.PIN(PIN[3]),.ERROR(ERROR_3));

stdor4 OR_A (ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR);


endmodule
```

D. LINE MANAGER

```
/*****************************************************************************
 LINE MANAGER
 Filename: line_mgr.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
 Revised: 20MAR9696

 Purpose: The function of this module is completely described in the behavioral model.
        This structural model uses a high speed RAM (hsram) for the MRMA List. The CAR is stored into this
RAM on a store or fetch_done signal.
        The predicted_ma_list is a register file for storing predicted memory addresses. This list is composed
of 128 address registers, 128 equality comparators, and 128 Valid status flags. The NAR is stored in this list at
the fetch_done pulse. If there is a match with the input address (in_addr), a priority encoder (ENC_C) determines
which line matches.
        The line replacement unit determines the next line to be replaced whenever the PRC needs to start a new
line. It first selects invalid lines. If all the lines are valid, then it selects lines that have been "aged". A priority
encoder (ENC_1) choses the line with the lowest index among all the lines that can be replaced. If all lines are
valid, the encoder's output enable (oe) signal is used to cause aging.
        Aging is accomplished by the use of a 7-bit counter (ager_counter), initially set to zero. When the
cause_aging signal from the encoder is high, the counter advances. A decoder (DEC_B) output causes the
appropriate Aged flag to be set.
        Changing values of the CAR or NAR have a propogation delay of 25 ns (1.8 cycles) through the input
address multiplexer (in_addr mux). This required the addition of wait states in the Controller before each of the
tests.
The Revised Controller State Diagram and Revised Controller State Output Table show the required changes.

*****************************************************************************/
module line_mgr (CAR,NAR,HRESET_,a_select,test,fetch_done,flush,store,
   new_replace,MRMA_out,ActiveLine,line_empty,hit,clk);
  // epoch set_attribute FIXEDBLOCK = 1

  input [26:0] CAR,NAR;
  input HRESET_,a_select,test,fetch_done,flush,store,new_replace,clk;
  output [26:0] MRMA_out;
  output [6:0] ActiveLine;
  output line_empty,hit;
```

```
wire [127:0] Valid;
wire [26:0] in_addr,in_addr_buf,MRMA_out;
wire [6:0] ActiveLine,ReplaceLine,match_line,w1;
wire MRMA_write,match,all_lines_valid,a_select_,w2,hit,
   hreset,store_,line_empty,le_set_;

//Address multiplexer
mux2 #(27,0,"AUTO","1")
   addrmux (.IN0(CAR),.IN1(NAR),.S0(a_select),.Y(in_addr));
buff #(27,0,"AUTO","20") InAddrBuffer (.IN0(in_addr),.Y(in_addr_buf));

//MRMA_list
stdnor2 MRMA_NOR (.IN0(store),.IN1(fetch_done),.Y(MRMA_write_));
hsram #(27,128,7,32,1,"2")
   MRMA_list (.A(ActiveLine),.DIN(CAR),.WR(MRMA_write_),.DOUT(MRMA_out));

//PredMA_list
// epoch pre_compiled predicted_ma_list
predicted_ma_list
   PredMA_list1 (NAR,in_addr_buf,ActiveLine,fetch_done,flush,HRESET_,
         Valid,match_line,match);
and128 all_valid_ands (Valid,all_lines_valid);

//Line Replacement Unit
// epoch pre_compiled line_replacement_unit
line_replacement_unit LRU1(Valid,ActiveLine,all_lines_valid,
            new_replace,fetch_done,HRESET_,clk,ReplaceLine);

//ActiveLine pointer
stdbufinv a_select_inv(.IN0(a_select),.Y(a_select_));
stdand2 AL_AND (.IN0(test),.IN1(a_select_),.Y(w2));
mux2 #(7,0,"AUTO","1")
   al_mux (.IN0(ReplaceLine),.IN1(match_line),.S0(match),.Y(w1));
dff_c #(7,0,"AUTO","1")
   ActiveLineReg (.CLK(w2),.CLR(HRESET_),.D(w1),.Q(ActiveLine));

//Hit status flag
stdlatch hit_latch(.D(match),.EN(test),.Q(hit));

//line_empty status flag
stdbufinv HRESET_inv(.IN0(HRESET_),.Y(hreset));
stdbufinv store_inv(.IN0(store),.Y(store_));
stdnor2 LE_NOR (.IN0(hreset),.IN1(new_replace),.Y(le_set_));
srlatch line_empty_latch(.S_(le_set_),.R_(store_),.Q(line_empty));

endmodule
```

1. Address Register With Equal Comparator

```
/*****************************************************************************
 ADDRESS REGISTER WITH EQUALITY COMPARITOR for PredMA storage
 Filename: addre.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
 Revised: 13FEB96

 Purpose: This structural model is a building block for the Predicted Memory Address List (PredMA_List). It
 consists of a single 27-bit register and an equality comparator. The output of the register is compared with the
 input address (in_addr).

*****************************************************************************/
module addre (NAR,in_addr,store_enable,eq,HRESET_);

  // epoch set_attribute FIXEDBLOCK = 1

  input [26:0] NAR,in_addr;
  input store_enable,HRESET_;
  output eq;

  wire [26:0] w1;
  wire eq;

  dff_c #(27,0,"AUTO","1") PredMA_reg (.CLK(store_enable),.CLR(HRESET_),
                        .D(NAR),.Q(w1));
  equal #(27,0,"AUTO","1") equal1 (.A(w1),.B(in_addr),.Y(eq));

endmodule
```

2. AND Gate With 128 Inputs and One Output

```
/*****************************************************************************
 128-INPUT AND GATE
 Filename: and128.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
 Revised: 20MAR96

 Purpose: This structural model is a 128-input AND gate.

*****************************************************************************/
module and128 (in,out);

  input [127:0] in;
```

```
  output out;
  wire out,out_unbuffered;

  wire [31:0] A;
  wire [7:0] B;
  wire [1:0] C;

  and4 #(32,0,"AUTO","1") AND_A (.IN0(in[127:96]),.IN1(in[95:64]),
                .IN2(in[63:32]),.IN3(in[31:0]),.Y(A));
  and4 #( 8,0,"AUTO","1") AND_B (.IN0( A[31:24]),.IN1( A[23:16]),
                .IN2( A[15:8]),.IN3( A[7:0]),.Y(B));
  and4 #( 2,0,"AUTO","1") AND_C (.IN0( B[7:6]),.IN1( B[5:4]),
                .IN2( B[3:2]),.IN3( B[1:0]),.Y(C));
  stdand2 AND_D (.IN0(C[0]),.IN1(C[1]),.Y(out_unbuffered));
  stdbuf #("15") OutputBuffer (.IN0(out_unbuffered),.Y(out));

endmodule
```

3. Codefile for Seven-to-128 Decoder (dec7to128e.codefile)

```
// PLA TABLE for 7 to 128 decoder with enable
// in0 in1 in2 in3 in4 in5 in6 EN
00000001        // line 0
00000011        // line
00000101        // line
00000111        // line
00001001        // line
00001011        // line
00001101        // line
00001111        // line
00010001        // line
00010011        // line
00010101        // line
00010111        // line
00011001        // line
00011011        // line
00011101        // line
00011111        // line
00100001        // line
00100011        // line
00100101        // line
00100111        // line
00101001        // line
00101011        // line
00101101        // line
00101111        // line
```

00110001 // line
00110011 // line
00110101 // line
00110111 // line
00111001 // line
00111011 // line
00111101 // line
00111111 // line
01000001 // line 32
01000011 // line
01000101 // line
01000111 // line
01001001 // line
01001011 // line
01001101 // line
01001111 // line
01010001 // line
01010011 // line
01010101 // line
01010111 // line
01011001 // line
01011011 // line
01011101 // line
01011111 // line
01100001 // line
01100011 // line
01100101 // line
01100111 // line
01101001 // line
01101011 // line
01101101 // line
01101111 // line
01110001 // line
01110011 // line
01110101 // line
01110111 // line
01111001 // line
01111011 // line
01111101 // line
01111111 // line
10000001 // line 64
10000011 // line
10000101 // line
10000111 // line
10001001 // line
10001011 // line
10001101 // line
10001111 // line
10010001 // line
10010011 // line

```
10010101    // line
10010111    // line
10011001    // line
10011011    // line
10011101    // line
10011111    // line
10100001    // line
10100011    // line
10100101    // line
10100111    // line
10101001    // line
10101011    // line
10101101    // line
10101111    // line
10110001    // line
10110011    // line
10110101    // line
10110111    // line
10111001    // line
10111011    // line
10111101    // line
10111111    // line
11000001    // line
11000011    // line
11000101    // line
11000111    // line
11001001    // line
11001011    // line
11001101    // line
11001111    // line
11010001    // line
11010011    // line
11010101    // line
11010111    // line
11011001    // line
11011011    // line
11011101    // line
11011111    // line
11100001    // line
11100011    // line
11100101    // line
11100111    // line
11101001    // line
11101011    // line
11101101    // line
11101111    // line
11110001    // line
11110011    // line
11110101    // line
11110111    // line
```

```
11111001        // line
11111011        // line
11111101        // line
11111111        // line 128

// END TABLE
```

4. One-Hundred-and-Twenty-Eight-Input, Seven-Output Encoder, Priority to Low Bits

```
/*************************************************************************
128 TO 7 ENCODER, PRIORITY LOW
Filename: enc128to7lo.v
Author: Joseph R. Robert, Jr.
Date:  21DEC95
Revised: 13FEB96

Purpose: This structural model is a 128-bit input, 7-bit output priority encoder. The highest priority is given to
the bit with the lowest index. Inputs and outputs are active high. It is composed of four 32 to 5 priority encoders
and the logic gates necessary to connect them together.
*************************************************************************/

module enc128to7lo (I,A,ei,eo,gs);

// epoch set_attribute FIXEDBLOCK = 1

input [127:0] I;
input ei;
output [6:0] A;
output gs,eo;

wire [4:0] g0A,g1A,g2A,g3A;
wire g3eo,g2eo,g1eo,g3gs,g2gs,g1gs,g0gs,eo,gs;

enc32to5lo ENCg3 (I[127:96],g3A,g2eo,  eo,g3gs);
enc32to5lo ENCg2 (I[ 95:64],g2A,g1eo,g2eo,g2gs);
enc32to5lo ENCg1 (I[ 63:32],g1A,g0eo,g1eo,g1gs);
enc32to5lo ENCg0 (I[ 31: 0],g0A,  ei,g0eo,g0gs);

//Group Select
stdor4 OR_A (g3gs,g2gs,g1gs,g0gs,gs);

//A6
stdor2 OR_B (g3gs,g2gs,A[6]);

//A5
stdor2 OR_C (g3gs,g1gs,A[5]);
```

```
//A4 - A0
stdor4 OR_D (g0A[4],g1A[4],g2A[4],g3A[4],A[4]);
stdor4 OR_E (g0A[3],g1A[3],g2A[3],g3A[3],A[3]);
stdor4 OR_F (g0A[2],g1A[2],g2A[2],g3A[2],A[2]);
stdor4 OR_G (g0A[1],g1A[1],g2A[1],g3A[1],A[1]);
stdor4 OR_H (g0A[0],g1A[0],g2A[0],g3A[0],A[0]);

endmodule
```

5. Thirty-Two-Input, Five-Output Encoder, Priority to Low Bits

```
/*************************************************************************
 32 TO 5 ENCODER, PRIORITY LOW
 Filename: enc32to5lo.v
 Author: Joseph R. Robert, Jr.
 Date:   21DEC95
 Revised: 13FEB96

 Purpose: This structural model is a 32-bit input, 5-bit output priority encoder. The highest priority is given to the
bit with the lowest index. Inputs and outputs are active high.
        This module is a composed of four 8 to 3 priority encoders and the logic gates necessary to connect them
together. This module is a building block for the 128 to 7 priority encoder.
*************************************************************************/

module enc32to5lo (i,A,ei,eo,gs);

// epoch set_attribute FIXEDBLOCK = 1

input [31:0] i;
input ei;
output [4:0] A;
output gs,eo;

wire [2:0] g0A,g1A,g2A,g3A;
wire g3eo,g2eo,g1eo,g3gs,g2gs,g1gs,g0gs,eo,gs;

enc8to3lo ENCg3 (i[31:24],g3A,g2eo,  eo,g3gs);
enc8to3lo ENCg2 (i[23:16],g2A,g1eo,g2eo,g2gs);
enc8to3lo ENCg1 (i[15: 8],g1A,g0eo,g1eo,g1gs);
enc8to3lo ENCg0 (i[ 7: 0],g0A,  ei,g0eo,g0gs);

//Group Select
stdor4 OR_A (g3gs,g2gs,g1gs,g0gs,gs);

//A4
stdor2 OR_B (g3gs,g2gs,A[4]);
```

```
//A3
stdor2 OR_C (g3gs,g1gs,A[3]);

//A2 - A0
stdor4 OR_D (g0A[2],g1A[2],g2A[2],g3A[2],A[2]);
stdor4 OR_E (g0A[1],g1A[1],g2A[1],g3A[1],A[1]);
stdor4 OR_F (g0A[0],g1A[0],g2A[0],g3A[0],A[0]);

endmodule
```

6. Eight-Input, Three-Output Encoder, Priority to Low Bits

```
/*************************************************************************
 8 TO 3 ENCODER, PRIORITY LOW
 Filename: enc8to3lo.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
 Revised: 13FEB96

 Purpose: This structural model is an 8-bit input, 3-bit output priority encoder. The highest priority is given to the
bit with the lowest index. Inputs and outputs are active high.

Truth table

Inputs                  Outputs
EI I7 I6 I5 I4 I3 I2 I1 I0  A2 A1 A0 GS EO
0 x x x x x x x x  0 0 0 0 0
1 1 0 0 0 0 0 0 0  1 1 1 1 0
1 x 1 0 0 0 0 0 0  1 1 0 1 0
1 x x 1 0 0 0 0 0  1 0 1 1 0
1 x x x 1 0 0 0 0  1 0 0 1 0
1 x x x x 1 0 0 0  0 1 1 1 0
1 x x x x x 1 0 0  0 1 0 1 0
1 x x x x x x 1 0  0 0 1 1 0
1 x x x x x x x 1  0 0 0 1 0
1 0 0 0 0 0 0 0 0  0 0 0 0 1
*************************************************************************/

module enc8to3lo (I,A,EI,EO,GS);

// epoch set_attribute FIXEDBLOCK = 1

input [7:0] I;
input EI;
output [2:0] A;
output GS,EO;
```

```
wire [7:0] I_;
wire [2:0] A;
wire EI_,GS,EO,EO_;
supply1 VDD;

//Standard cell implemenation is more efficient here. See User Man. 5-34.
inv #(8,0,"AUTO","1") INV1 (.IN0(I),.Y(I_));
stdinv INV_AA (.IN0(EI),.Y(EI_));

//Enable Output
  stdand4 AND_A (EI,I_[7],I_[6],I_[5],w1);
  stdand4 AND_B (I_[4],I_[3],I_[2],I_[1],w2);
  stdand3 AND_C (I_[0],w1,w2,EO);

//Group Select ("Got Something")
  stdnor2 NOR_D (EI_,EO,GS);

//Encode A2 = EI.(I7.I6_.I5_.I4_.I3_.I2_.I1_.I0_ + I6.I5_.I4_.I3_.I2_.I1_.I0_ +
//            I5.I4_.I3_.I2_.I1_.I0_ + I4.I3_.I2_.I1_.I0_)
//        = EI.(I7.I3_.I2_.I1_.I0_ + I6.I3_.I2_.I1_.I0_ +
//            I5.I3_.I2_.I1_.I0_ + I4.I3_.I2_.I1_.I0_)
//        = EI.I3_.I2_.I1_.I0_.(I7 + I6. + I5. + I4.)

  stdor4  OR_E  (I[7],I[6],I[5],I[4],w5);
  stdand4 AND_F (EI,I_[3],I_[2],I_[1],w6);
  stdand3 AND_FA (I_[0],w5,w6,A[2]);

//Encode A1 = EI.(I7.I6_.I5_.I4_.I3_.I2_.I1_.I0_ + I6.I5_.I4_.I3_.I2_.I1_.I0_ +
//            I3.I2_.I1_.I0_ + I2.I1_.I0_)
//        = EI.I1_.I0_.(I7.I5_.I4_. + I6.I5_.I4_ + I3. + I2)


  stdand3 AND_G (I[7],I_[5],I_[4],w10);
  stdand3 AND_H (I[6],I_[5],I_[4],w11);
  stdor4  OR_I  (w10,w11,I[3],I[2],w12);
  stdand4 AND_J (EI,I_[1],I_[0],w12,A[1]);

//Encode A0 = EI.(I7.I6_.I5_.I4_.I3_.I2_.I1_.I0_ + I5.I4_.I3_.I2_.I1_.I0_ +
//            I3.I2_.I1_.I0_ + I1.I0_)
//        = EI.I0_.(I7.I6_.I4_.I2_ + I5.I4_.I2_ + I3.I2_ + I1)

  stdand4 AND_K (I[7],I_[6],I_[4],I_[2],w15);
  stdand3 AND_L (I[5],I_[4],I_[2],w16);
  stdand2 AND_M (I[3],I_[2],w17);
  stdor4  OR_N  (w15,w16,w17,I[1],w18);
  stdand3 AND_P (EI,I_[0],w18,A[0]);

endmodule
```

7. Line Replacement Unit

```
/*****************************************************************************
LINE REPLACEMENT UNIT
Filename: line_replacement_unit.v
Author: Joseph R. Robert, Jr.
Date:  21DEC95
Revised: 13FEB96

Purpose: This structural model determines the next line to be replaced whenever the PRC needs to start a new
line. It first selects invalid lines. If all the lines are valid, then it selects lines that have been "aged". A priority
encoder (ENC_1) choses the line with the lowest index among all the lines that can be replaced. If all lines are
valid, the encoder's output enable (oe) signal is used to cause aging. A line X can be replaced if the following holds
true for that line:

        not (X=ActiveLine) AND {not Valid[X] OR (all_lines_valid AND Aged[X])}

        Aging is accomplished by the use of a 7-bit counter (ager_counter), initially set to zero. When the
cause_aging signal from the encoder is high, the counter advances. A decoder (DEC_B) output causes the
appropriate Aged flag to be set.

*****************************************************************************/
module line_replacement_unit(Valid,ActiveLine,all_lines_valid,
                new_replace,fetch_done,HRESET_,CLK,ReplaceLine);
  // epoch set_attribute FIXEDBLOCK = 1

  input [127:0] Valid;
  input [6:0] ActiveLine;
  input all_lines_valid,new_replace,fetch_done,HRESET_,CLK;
  output [6:0] ReplaceLine;

  supply1 Vdd;
  wire [127:0] w1,w2,w4,w5,w6,w7,set_,reset_,Aged,fetch_done128,
        all_lines_valid128,HRESET128_;
  wire [6:0] ager_line,ReplaceLine,HRESET7_;
  wire ager_en,cause_aging,latch_en,latch_en_buf,nc1,nc2;
  split128 fetch_done_split (fetch_done,fetch_done128);
  split128 alv_split (all_lines_valid,all_lines_valid128);
  split128 HRESET_split (HRESET_,HRESET128_);
  split7 HRESET_split7 (HRESET_,HRESET7_);

  decoder #(8,128,"verilog/dec7to128e.codefile","2")
    DEC_A (.SEL({Vdd,ActiveLine[0],ActiveLine[1],ActiveLine[2],
        ActiveLine[3],ActiveLine[4],ActiveLine[5],ActiveLine[6]}),
        .Y(w1));
  decoder_inv #(8,128,"verilog/dec7to128e.codefile","2")
    DEC_B (.SEL({Vdd,ager_line[0],ager_line[1],ager_line[2],
        ager_line[3],ager_line[4],ager_line[5],ager_line[6]}),.YBAR(set_));
```

```
  nand2 #(128,0,"AUTO","1") NAND_A (.IN0(w1),.IN1(fetch_done128),.Y(w2));
  and2  #(128,0,"AUTO","1") AND_E (.IN0(w2),.IN1(HRESET128_),.Y(reset_));
  srlatch128 AgedReg (set_,reset_,Aged);
  scntr_c #(7,0,"AUTO","1")
    ager_counter (.CLK(ager_en),.CLR(HRESET_),.EN(Vdd),.COUT(nc2),
          .Q(ager_line));
  stdand2 AND_F (.IN0(CLK),.IN1(cause_aging),.Y(ager_en));
  nand2 #(128,0,"AUTO","1")
    NAND_B (.IN0(all_lines_valid128),.IN1(Aged),.Y(w4));
  and2 #(128,0,"AUTO","1") AND_C (.IN0(w4),.IN1(Valid),.Y(w5));
  nor2 #(128,0,"AUTO","1") NOR_D (.IN0(w1),.IN1(w5),.Y(w6));
  stdor2 OR_F (.IN0(new_replace),.IN1(cause_aging),.Y(latch_en));
  stdbuf #("19") LatchEnableBuffer (.IN0(latch_en),.Y(latch_en_buf));
  enc128to7lo ENC1 (.I(w7),.A(ReplaceLine),.ei(Vdd),
          .eo(cause_aging),.gs(nc1));
  latch_c #(128,0,"AUTO","1")
    ReplaceLineLatch (.EN(latch_en_buf),.CLR(HRESET_),.D(w6),.Q(w7));

endmodule
```

8. OR Gate With 128 Inputs, One Output

```
/*****************************************************************************
 128-INPUT OR GATE
 Filename: or128.v
 Author: Joseph R. Robert, Jr.
 Date:   21DEC95
 Revised: 23JAN96

*****************************************************************************/

module or128 (in,out);  //An OR tree, equivalent to a 128-input OR gate.
  input [127:0] in;
  output out;
  wire out;

  wire [31:0] A;
  wire [7:0] B;
  wire [1:0] C;

  or4 #(32,0,"AUTO","1") OR_A (.IN0(in[127:96]),.IN1(in[95:64]),
                    .IN2(in[63:32]),.IN3(in[31:0]),.Y(A));
  or4 #( 8,0,"AUTO","1") OR_B (.IN0(A[31:24]),.IN1(A[23:16]),
                    .IN2(A[15:8]),.IN3(A[7:0]),.Y(B));
  or4 #( 2,0,"AUTO","1") OR_C (.IN0(B[7:6]),.IN1(B[5:4]),
                    .IN2(B[3:2]),.IN3(B[1:0]),.Y(C));
  stdor2 OR_D (.IN0(C[1]),.IN1(C[0]),.Y(out));
```

```
endmodule
```

9. Predicted Memory Address List

```
/*************************************************************************
PREDICTED MEMORY ADDRESS LIST
Filename: predma_list.v
Author: Joseph R. Robert, Jr.
Date:  21DEC95
Revised: 13FEB96

Purpose: This structural model is a register file for storing predicted memory addresses. This list is composed of
128 address registers, 128 equality comparators, and 128 Valid status flags. The NAR is stored in this list at the
fetch_done pulse. If there is a match with the input address (in_addr), a priority encoder (ENC_C) determines
which line matches.

*************************************************************************/
module predicted_ma_list (NAR,in_addr,ActiveLine,fetch_done,flush,HRESET_,
            Valid,match_line,match);
 // epoch set_attribute FIXEDBLOCK = 1

 input [26:0] NAR, in_addr;
 input [6:0] ActiveLine;
 input fetch_done,flush,HRESET_;
 output [127:0] Valid;
 output [6:0] match_line;
 output match;

 wire [127:0] store_en,store_en_buf,flush_enable_,set_,reset_,
        Valid,equal,m,HRESET128_;
 wire nc1,nc2;
 supply1 Vdd;

 split128 hreset_splitter (.in(HRESET_),.out(HRESET128_));
 decoder #(8,128,"verilog/dec7to128e.codefile","2")
     DEC_A (.SEL({fetch_done,ActiveLine[0],ActiveLine[1],ActiveLine[2],
         ActiveLine[3],ActiveLine[4],ActiveLine[5],ActiveLine[6]}),
         .Y(store_en));
 buff #(128,0,"AUTO","8") StoreEnBuffer (.IN0(store_en),.Y(store_en_buf));
 decoder_inv #(8,128,"verilog/dec7to128e.codefile","2")
     DEC_B (.SEL({flush,ActiveLine[0],ActiveLine[1],ActiveLine[2],
         ActiveLine[3],ActiveLine[4],ActiveLine[5],ActiveLine[6]}),
         .YBAR(flush_enable_));
 inv #(128,0,"AUTO","1") INV_A (.IN0(store_en_buf),.Y(set_));
 and2 #(128,0,"AUTO","1")
   AND_B (.IN0(flush_enable_),.IN1(HRESET128_),.Y(reset_));
 srlatch128 Valid_latch (.S_(set_),.R_(reset_),.Q(Valid));
```

```
  and2 #(128,0,"AUTO","1")
    AND_C (.IN0(Valid),.IN1(equal),.Y(m));
  or128 MATCH_OR (.in(m),.out(match));
  enc128to7lo ENC_C (.I(m),.A(match_line),.ei(Vdd),.eo(nc1),.gs(nc2));

// epoch pre_compiled addre
addre PredMA0 (NAR,in_addr,store_en_buf[0],equal[0],HRESET_);
addre PredMA1 (NAR,in_addr,store_en_buf[1],equal[1],HRESET_);
addre PredMA2 (NAR,in_addr,store_en_buf[2],equal[2],HRESET_);
addre PredMA3 (NAR,in_addr,store_en_buf[3],equal[3],HRESET_);
addre PredMA4 (NAR,in_addr,store_en_buf[4],equal[4],HRESET_);
addre PredMA5 (NAR,in_addr,store_en_buf[5],equal[5],HRESET_);
addre PredMA6 (NAR,in_addr,store_en_buf[6],equal[6],HRESET_);
addre PredMA7 (NAR,in_addr,store_en_buf[7],equal[7],HRESET_);
addre PredMA8 (NAR,in_addr,store_en_buf[8],equal[8],HRESET_);
addre PredMA9 (NAR,in_addr,store_en_buf[9],equal[9],HRESET_);
addre PredMA10 (NAR,in_addr,store_en_buf[10],equal[10],HRESET_);
addre PredMA11 (NAR,in_addr,store_en_buf[11],equal[11],HRESET_);
addre PredMA12 (NAR,in_addr,store_en_buf[12],equal[12],HRESET_);
addre PredMA13 (NAR,in_addr,store_en_buf[13],equal[13],HRESET_);
addre PredMA14 (NAR,in_addr,store_en_buf[14],equal[14],HRESET_);
addre PredMA15 (NAR,in_addr,store_en_buf[15],equal[15],HRESET_);
addre PredMA16 (NAR,in_addr,store_en_buf[16],equal[16],HRESET_);
addre PredMA17 (NAR,in_addr,store_en_buf[17],equal[17],HRESET_);
addre PredMA18 (NAR,in_addr,store_en_buf[18],equal[18],HRESET_);
addre PredMA19 (NAR,in_addr,store_en_buf[19],equal[19],HRESET_);
addre PredMA20 (NAR,in_addr,store_en_buf[20],equal[20],HRESET_);
addre PredMA21 (NAR,in_addr,store_en_buf[21],equal[21],HRESET_);
addre PredMA22 (NAR,in_addr,store_en_buf[22],equal[22],HRESET_);
addre PredMA23 (NAR,in_addr,store_en_buf[23],equal[23],HRESET_);
addre PredMA24 (NAR,in_addr,store_en_buf[24],equal[24],HRESET_);
addre PredMA25 (NAR,in_addr,store_en_buf[25],equal[25],HRESET_);
addre PredMA26 (NAR,in_addr,store_en_buf[26],equal[26],HRESET_);
addre PredMA27 (NAR,in_addr,store_en_buf[27],equal[27],HRESET_);
addre PredMA28 (NAR,in_addr,store_en_buf[28],equal[28],HRESET_);
addre PredMA29 (NAR,in_addr,store_en_buf[29],equal[29],HRESET_);
addre PredMA30 (NAR,in_addr,store_en_buf[30],equal[30],HRESET_);
addre PredMA31 (NAR,in_addr,store_en_buf[31],equal[31],HRESET_);
addre PredMA32 (NAR,in_addr,store_en_buf[32],equal[32],HRESET_);
addre PredMA33 (NAR,in_addr,store_en_buf[33],equal[33],HRESET_);
addre PredMA34 (NAR,in_addr,store_en_buf[34],equal[34],HRESET_);
addre PredMA35 (NAR,in_addr,store_en_buf[35],equal[35],HRESET_);
addre PredMA36 (NAR,in_addr,store_en_buf[36],equal[36],HRESET_);
addre PredMA37 (NAR,in_addr,store_en_buf[37],equal[37],HRESET_);
addre PredMA38 (NAR,in_addr,store_en_buf[38],equal[38],HRESET_);
addre PredMA39 (NAR,in_addr,store_en_buf[39],equal[39],HRESET_);
addre PredMA40 (NAR,in_addr,store_en_buf[40],equal[40],HRESET_);
addre PredMA41 (NAR,in_addr,store_en_buf[41],equal[41],HRESET_);
addre PredMA42 (NAR,in_addr,store_en_buf[42],equal[42],HRESET_);
addre PredMA43 (NAR,in_addr,store_en_buf[43],equal[43],HRESET_);
```

```
addre PredMA44 (NAR,in_addr,store_en_buf[44],equal[44],HRESET_);
addre PredMA45 (NAR,in_addr,store_en_buf[45],equal[45],HRESET_);
addre PredMA46 (NAR,in_addr,store_en_buf[46],equal[46],HRESET_);
addre PredMA47 (NAR,in_addr,store_en_buf[47],equal[47],HRESET_);
addre PredMA48 (NAR,in_addr,store_en_buf[48],equal[48],HRESET_);
addre PredMA49 (NAR,in_addr,store_en_buf[49],equal[49],HRESET_);
addre PredMA50 (NAR,in_addr,store_en_buf[50],equal[50],HRESET_);
addre PredMA51 (NAR,in_addr,store_en_buf[51],equal[51],HRESET_);
addre PredMA52 (NAR,in_addr,store_en_buf[52],equal[52],HRESET_);
addre PredMA53 (NAR,in_addr,store_en_buf[53],equal[53],HRESET_);
addre PredMA54 (NAR,in_addr,store_en_buf[54],equal[54],HRESET_);
addre PredMA55 (NAR,in_addr,store_en_buf[55],equal[55],HRESET_);
addre PredMA56 (NAR,in_addr,store_en_buf[56],equal[56],HRESET_);
addre PredMA57 (NAR,in_addr,store_en_buf[57],equal[57],HRESET_);
addre PredMA58 (NAR,in_addr,store_en_buf[58],equal[58],HRESET_);
addre PredMA59 (NAR,in_addr,store_en_buf[59],equal[59],HRESET_);
addre PredMA60 (NAR,in_addr,store_en_buf[60],equal[60],HRESET_);
addre PredMA61 (NAR,in_addr,store_en_buf[61],equal[61],HRESET_);
addre PredMA62 (NAR,in_addr,store_en_buf[62],equal[62],HRESET_);
addre PredMA63 (NAR,in_addr,store_en_buf[63],equal[63],HRESET_);
addre PredMA64 (NAR,in_addr,store_en_buf[64],equal[64],HRESET_);
addre PredMA65 (NAR,in_addr,store_en_buf[65],equal[65],HRESET_);
addre PredMA66 (NAR,in_addr,store_en_buf[66],equal[66],HRESET_);
addre PredMA67 (NAR,in_addr,store_en_buf[67],equal[67],HRESET_);
addre PredMA68 (NAR,in_addr,store_en_buf[68],equal[68],HRESET_);
addre PredMA69 (NAR,in_addr,store_en_buf[69],equal[69],HRESET_);
addre PredMA70 (NAR,in_addr,store_en_buf[70],equal[70],HRESET_);
addre PredMA71 (NAR,in_addr,store_en_buf[71],equal[71],HRESET_);
addre PredMA72 (NAR,in_addr,store_en_buf[72],equal[72],HRESET_);
addre PredMA73 (NAR,in_addr,store_en_buf[73],equal[73],HRESET_);
addre PredMA74 (NAR,in_addr,store_en_buf[74],equal[74],HRESET_);
addre PredMA75 (NAR,in_addr,store_en_buf[75],equal[75],HRESET_);
addre PredMA76 (NAR,in_addr,store_en_buf[76],equal[76],HRESET_);
addre PredMA77 (NAR,in_addr,store_en_buf[77],equal[77],HRESET_);
addre PredMA78 (NAR,in_addr,store_en_buf[78],equal[78],HRESET_);
addre PredMA79 (NAR,in_addr,store_en_buf[79],equal[79],HRESET_);
addre PredMA80 (NAR,in_addr,store_en_buf[80],equal[80],HRESET_);
addre PredMA81 (NAR,in_addr,store_en_buf[81],equal[81],HRESET_);
addre PredMA82 (NAR,in_addr,store_en_buf[82],equal[82],HRESET_);
addre PredMA83 (NAR,in_addr,store_en_buf[83],equal[83],HRESET_);
addre PredMA84 (NAR,in_addr,store_en_buf[84],equal[84],HRESET_);
addre PredMA85 (NAR,in_addr,store_en_buf[85],equal[85],HRESET_);
addre PredMA86 (NAR,in_addr,store_en_buf[86],equal[86],HRESET_);
addre PredMA87 (NAR,in_addr,store_en_buf[87],equal[87],HRESET_);
addre PredMA88 (NAR,in_addr,store_en_buf[88],equal[88],HRESET_);
addre PredMA89 (NAR,in_addr,store_en_buf[89],equal[89],HRESET_);
addre PredMA90 (NAR,in_addr,store_en_buf[90],equal[90],HRESET_);
addre PredMA91 (NAR,in_addr,store_en_buf[91],equal[91],HRESET_);
addre PredMA92 (NAR,in_addr,store_en_buf[92],equal[92],HRESET_);
addre PredMA93 (NAR,in_addr,store_en_buf[93],equal[93],HRESET_);
```

```
addre PredMA94 (NAR,in_addr,store_en_buf[94],equal[94],HRESET_);
addre PredMA95 (NAR,in_addr,store_en_buf[95],equal[95],HRESET_);
addre PredMA96 (NAR,in_addr,store_en_buf[96],equal[96],HRESET_);
addre PredMA97 (NAR,in_addr,store_en_buf[97],equal[97],HRESET_);
addre PredMA98 (NAR,in_addr,store_en_buf[98],equal[98],HRESET_);
addre PredMA99 (NAR,in_addr,store_en_buf[99],equal[99],HRESET_);
addre PredMA100 (NAR,in_addr,store_en_buf[100],equal[100],HRESET_);
addre PredMA101 (NAR,in_addr,store_en_buf[101],equal[101],HRESET_);
addre PredMA102 (NAR,in_addr,store_en_buf[102],equal[102],HRESET_);
addre PredMA103 (NAR,in_addr,store_en_buf[103],equal[103],HRESET_);
addre PredMA104 (NAR,in_addr,store_en_buf[104],equal[104],HRESET_);
addre PredMA105 (NAR,in_addr,store_en_buf[105],equal[105],HRESET_);
addre PredMA106 (NAR,in_addr,store_en_buf[106],equal[106],HRESET_);
addre PredMA107 (NAR,in_addr,store_en_buf[107],equal[107],HRESET_);
addre PredMA108 (NAR,in_addr,store_en_buf[108],equal[108],HRESET_);
addre PredMA109 (NAR,in_addr,store_en_buf[109],equal[109],HRESET_);
addre PredMA110 (NAR,in_addr,store_en_buf[110],equal[110],HRESET_);
addre PredMA111 (NAR,in_addr,store_en_buf[111],equal[111],HRESET_);
addre PredMA112 (NAR,in_addr,store_en_buf[112],equal[112],HRESET_);
addre PredMA113 (NAR,in_addr,store_en_buf[113],equal[113],HRESET_);
addre PredMA114 (NAR,in_addr,store_en_buf[114],equal[114],HRESET_);
addre PredMA115 (NAR,in_addr,store_en_buf[115],equal[115],HRESET_);
addre PredMA116 (NAR,in_addr,store_en_buf[116],equal[116],HRESET_);
addre PredMA117 (NAR,in_addr,store_en_buf[117],equal[117],HRESET_);
addre PredMA118 (NAR,in_addr,store_en_buf[118],equal[118],HRESET_);
addre PredMA119 (NAR,in_addr,store_en_buf[119],equal[119],HRESET_);
addre PredMA120 (NAR,in_addr,store_en_buf[120],equal[120],HRESET_);
addre PredMA121 (NAR,in_addr,store_en_buf[121],equal[121],HRESET_);
addre PredMA122 (NAR,in_addr,store_en_buf[122],equal[122],HRESET_);
addre PredMA123 (NAR,in_addr,store_en_buf[123],equal[123],HRESET_);
addre PredMA124 (NAR,in_addr,store_en_buf[124],equal[124],HRESET_);
addre PredMA125 (NAR,in_addr,store_en_buf[125],equal[125],HRESET_);
addre PredMA126 (NAR,in_addr,store_en_buf[126],equal[126],HRESET_);
addre PredMA127 (NAR,in_addr,store_en_buf[127],equal[127],HRESET_);

endmodule
```

10. One-to-128 Wire Splitter

```
/*****************************************************************************
1 TO 128 WIRE SPLITTER
Filename: split128.v
Author: Joseph R. Robert, Jr.
Date:   21DEC95
Revised: 23JAN96

Purpose: Splits a wire into 128 wires.
```

```
*************************************************************************/
module split128 (in,out); //Splits a wire into 128 wires.
  input in;
  output [127:0] out;

  assign out = {in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,
          in,in,in,in,in,in,in,in,in,in,in,in,in,in,in,in};
endmodule
```

11. One-to-Seven Wire Splitter

```
/*************************************************************************
 1 TO 7 WIRE SPLITTER
 Filename: split7.v
 Author: Joseph R. Robert, Jr.
 Date:   21DEC95
 Revised: 23JAN96

 Purpose: Splits a wire into 7 wires.

*************************************************************************/

module split7 (in,out); //Splits a wire into 7 wires.
  input in;
  output [6:0] out;

  assign out = {in,in,in,in,in,in,in};

endmodule
```

12. Set, Reset Latch

```
/*************************************************************************
 STANDARD SET,RESET LATCH
 Filename: srlatch.v
 Author: Joseph R. Robert, Jr.
 Date:   21DEC95
 Revised: 23JAN96
```

```
Reset has priority.

*****************************************************************************/
module srlatch (S_, R_, Q);
  input S_,R_;
  output Q;
  wire w1,w2,Q,Q_;

  stdnand2 NAND_A (.IN0(w2),.IN1(Q_),.Y(Q));
  stdnand2 NAND_B (.IN0(R_),.IN1(Q),.Y(Q_));
  stdnand2 NAND_C (.IN0(w1),.IN1(R_),.Y(w2));
  stdinv INV_D (.IN0(S_),.Y(w1));

endmodule
```

13. Set, Reset Latch Array 128 Bits Wide

```
/*****************************************************************************
 ARRAY OF 128 SET,RESET LATCHES
 Filename: srlatch128.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
 Revised: 23JAN96

*****************************************************************************/

module srlatch128 (S_, R_, Q);  //set-reset latch
  input [127:0] S_,R_;
  output [127:0] Q;
  wire [127:0] w1,w2,Q,Q_;

  nand2 #(128,0,"AUTO","1") NAND_A (.IN0(w2),.IN1(Q_),.Y(Q));
  nand2 #(128,0,"AUTO","1") NAND_B (.IN0(R_),.IN1(Q),.Y(Q_));
  nand2 #(128,0,"AUTO","1") NAND_C (.IN0(w1),.IN1(R_),.Y(w2));
  inv  #(128,0,"AUTO","1") INV_C  (.IN0(S_),.Y(w1));

endmodule
```

E. PREDICTOR

```
/*****************************************************************************
 PREDICTOR
 Filename: predictor.v
 Author: Joseph R. Robert, Jr.
 Date:  21DEC95
```

Revised: 06FEB96

Purpose: This module calculates the Next Address (stored in NAR) based on the Most Recent Memory Access (MRMA) and the Current Address (in the CAR). The prediction calculation is

NAR = 2*CAR - MRMA

In this structural implementation of the Predictor, the predict signal latches in the CAR and MRMA inputs. The subtraction is accomplished as a 2's compliment addition with a high speed adder.

The CAR is multiplied times 2 by concatenating a zero at the least significant end. The most significant bit of the CAR is not retained, since it will not have an effect on the 27-bit output of the adder. This would adversely affect address prediction only around the mid-point of the 4 gigabytes of memory. The Golden Rule here is "Design for the common case."

A number is negated in 2's compliment by inverting all the bits and adding 1. The MRMA is negated by inverting all its bits. Adding the required 1 is implemented as a Carry-In to the adder.

Epoch's TACTIC reported the propagation delay from predict to NAR to be 4.90 ns.

**********/

```
module predictor (MRMA,CAR,predict,NAR,HRESET_);
//CAR is [30:5] of 32-bit address
//MRMA and NAR are [31:5] of 32-bit address

// epoch set_attribute FIXEDBLOCK = 1

input [26:0] MRMA;
input [25:0] CAR;
input predict,HRESET_;
output [26:0] NAR;

wire [26:0] NAR,A,B,C;
wire nc;

`define group "predictor"

supply0 gnd;
supply1 vdd;

assign A[0] = gnd;
dff_c #(26,1,`group,"1")
    CAR_latch (.D(CAR),.CLK(predict),.CLR(HRESET_),.Q(A[26:1]));

dff_c #(27,1,`group,"1")
    MRMA_latch (.D(MRMA),.CLK(predict),.CLR(HRESET_),.Q(C));

bufinv #(27,1,`group,"1","speed")
       bit_complement (.IN0(C),.Y(B));
```

```
addhs #(27,1,`group,"1")
    adder (.A(A),.B(B),.CIN(vdd),.COUT(nc),.SUM(NAR));

endmodule
```

F. DATA LIST

```
/*************************************************************************
DATA LIST
Filename: datalist.v
Author: Joseph R. Robert, Jr.
Date:  15DEC95
Revised: 07FEB96

Purpose: This module stores the data retreived from memory in anticipation of a request by the CPU.

        The basic memory cell is Epoch's hsramoe (high speed ram with output enable). Since each hsram has
a maximum word size of 128 bits, there are two hsram parts in parallel to get the required 256-bit width.
        An upload signal causes the Data List to store the data on data_line into the address specified by
ActiveLine. The input upload has to be inverted to match the active-low WR input of the Epoch hsram component.
        A download signal causes the Data List to assert onto data_line the data in the address specified by
ActiveLine. This signal also has to be inverted for the same reason.
        Both the inverters can probably be removed if the Bus Interface Unit makes the upload and download
signals active low. That could only improve the response time of this data memory.
        Epoch calculated the following timing delays:

        download -> hsramoe.DOUT  2.3 ns
        ActiveLine -> hsramoe.DOUT 7.3 ns

        A design alternative is to use the regular speed version, ramoe, with the following timing delays.

        download -> ramoe.DOUT  4 ns
        ActiveLine -> ramoe.DOUT 16 ns

Using this slower RAM is possible, but would require a significant modification to the PRC behavior to handle to
longer delay, and would add a cycle delay to CPU reads when there is a hit in the PRC.
        Putting this module's VerilogOut file into the original PRC behavioral model for mixed-mode simulation
caused a timing error that had to be corrected in the Bus Interface Unit. After an upload to the DataList, data_line
must remain valid for long enough to meet the data hold time requirement of Epoch's hsramoe.

*************************************************************************/
module datalist (data_line,ActiveLine,upload,download);
// epoch set_attribute FIXEDBLOCK = 1

input [6:0] ActiveLine;
input upload,download;
```

```
inout [255:0] data_line;

wire [255:0] data_line;
wire write_,enable_;

//STRUCTURE

stdbufinv upload_inv (.IN0(upload),.Y(write_));

stdbufinv download_inv (.IN0(download),.Y(enable_));

hsramoe #(128,128,7,32,1,"1")
  data_ram1 (.A(ActiveLine),.DIN(data_line[127:0]),.DOUT(data_line[127:0]),
        .WR(write_),.OE(enable_));
hsramoe #(128,128,7,32,1,"1")
  data_ram0 (.A(ActiveLine),.DIN(data_line[255:128]),.DOUT(data_line[255:128]),
        .WR(write_),.OE(enable_));

endmodule
```

G. BUS INTERFACE

```
/*****************************************************************************
* BUS INTERFACE UNIT
* Filename: bus_interface.v
* Author: Joseph R. Robert, Jr.
* Date:   09OCT95
* Revised: 20MAR96
*
 Purpose: This module connects the PRC with the system bus.  It handles the protocol of data transfer in and out
of the PRC.
        When this module receives a fetch signal, it latches the address in the NAR, and requests the bus for a
burst read.  It stores the incoming data until all four bursts have been received.  Then it uploads the data into the
Data List and assserts fetch_done.  If there is a parity error during the fetch, the Bus Interface informs the Controller
by asserting fetch_abort, and the transaction is cancelled.
        When this module receives a send signal, it sends a cancel signal (CANX) to the memory module,
downloads data from the Data List, and then sends the data to the CPU.  When the transfer is finished, it asserts
send_done.
        The coordination of these activities is accomplished through the use of two Finite State Machines.  One
acts as an address bus master, and the other controls the flow of data.
*
*****************************************************************************/
module bus_interface (NAR_IN,BURSTSTART,BG_,AACK_,DBG_,send,fetch,
  clk,BR_,upload,download,fetch_done,fetch_abort,
  send_done,CANX,snoop_ignore,DATALINE,D,A,AP,DP,DPE_,
  TT,TSIZ,TC,ABB_,TS_,TBST_,DBB_,TA_,HRESET_);
```

```
// epoch set_attribute FIXEDBLOCK = 1

// Signals are defined in system.v.
input [26:0] NAR_IN;
input [1:0] BURSTSTART;
input BG_,AACK_,DBG_,send,fetch,clk,HRESET_;
output BR_,upload,download,fetch_done,fetch_abort;
output send_done,DPE_,CANX,snoop_ignore;
inout [255:0] DATALINE;
inout [63:0] D;
inout [31:0] A;
inout [7:0] DP;
inout [4:0] TT;
inout [3:0] AP;
inout [2:0] TSIZ;
inout [1:0] TC;
inout ABB_,TS_,TBST_,DBB_,TA_;

tri [255:0] DATALINE;
tri [63:0] D;
tri [31:0] A;
tri [7:0] DP;
tri [4:0] TT;
tri [3:0] AP;
tri [2:0] TSIZ;
tri [1:0] TC;
tri ABB_,TS_,TBST_,DBB_,TA_,DPE_;

supply1 VDD;
supply0 GND;

//Address section wires
wire [26:0] a_reg,NAR;
wire [3:0]  ap_reg,addr_parity_gen;
wire qual_BG_;

//Data section wires
wire [255:0] data,mux_out;
wire [31:0] dparity,dparity_gen;
//wire [3:0] dreg_clk;
wire [1:0] burst_start;
wire bs_clk,dreg0_clk,dreg1_clk,dreg2_clk,dreg3_clk,data_parity_error,qual_DBG_,
   dreg0_clk_buf,dreg1_clk_buf,dreg2_clk_buf,dreg3_clk_buf,a_en_buf_,CANX,
   dataline_en_buf_,d_en0_buf_,d_en1_buf_,d_en2_buf_,d_en3_buf_,ta,
   latch0_delay,latch1_delay,latch2_delay,latch3_delay;

//ADDRESS BUS INTERFACE

assign qual_BG_ = ~(ABB_ & !BG_);
```

```
// Next Address Register
dff #(27,0,"AUTO","AUTO")
   NextAddressReg (.CLK(NARLatch),.D(NAR_IN),.Q(NAR));

//Generate address parity.
parityo_gen32
   AddrParityGen (.D({NAR,GND,GND,GND,GND,GND}),.PGEN(addr_parity_gen));

//Address Output Registers and buffers
dff #(27,0,"AUTO","AUTO")
   AddressReg (.CLK(a_latch),.D(NAR),.Q(a_reg));
dff #(4,0,"AUTO","AUTO")
   AddrParReg (.CLK(a_latch),.D(addr_parity_gen),.Q(ap_reg));
tribuf #(32,0,"AUTO","AUTO")
   a_buffer (.EN(a_en_buf_),.IN0({a_reg,GND,GND,GND,GND,GND}),.Y(A));
stdbuf #("9") AEN_BUF (.IN0(a_en_),.Y(a_en_buf_));
tribuf #(4,0,"AUTO","AUTO")
   ap_buffer (.EN(a_en_),.IN0(ap_reg),.Y(AP));
tribuf #(5,0,"AUTO","AUTO")
   tt_buffer (.EN(a_en_),.IN0({GND,VDD,VDD,VDD,GND}),.Y(TT));
tribuf #(3,0,"AUTO","AUTO")
   tsize_buffer (.EN(a_en_),.IN0({GND,VDD,GND}),.Y(TSIZ));
tribuf #(2,0,"AUTO","AUTO")
   tcode_buffer (.EN(a_en_),.IN0({GND,GND}),.Y(TC));

stdtribuf abb_buffer (.EN(abb_en_),.IN0(abb_reg_),.Y(ABB_));
stdtribuf tbst_buffer (.EN(tbst_en_),.IN0(GND),.Y(TBST_));
stdtribuf ts_buffer (.EN(ts_en_),.IN0(ts_reg_),.Y(TS_));


//ADDRESS FINITE STATE MACHINE

parameter // epoch enum astat
  A_IDLE       = 3'd0,
  WAIT_FOR_BG  = 3'd1,
  MASTER       = 3'd2,
  TRANSFER     = 3'd3,
  WAIT_FOR_AACK = 3'd4,
  TERMINATION  = 3'd5,
  WAIT_FOR_NOT_FETCH = 3'd6,
  dc_astate    = 3'bxx;

reg [2:0] /* epoch enum astat */ astate, next_astate;
reg a_latch,a_en_,abb_reg_,abb_en_,BR_,NARLatch,snoop_ignore,
   tbst_en_,ts_reg_,ts_en_;

always @(posedge clk or negedge HRESET_)
 begin
   if (!HRESET_)
     astate = A_IDLE;
```

```
  else
    astate = next_astate;
 end

always @(astate or fetch or qual_BG_ or AACK_)
 begin

  //default values
      a_latch     = 1'b0;
      a_en_       = 1'b1;
      abb_reg_    = 1'b1;
      abb_en_     = 1'b1;
      BR_         = 1'b1;
      NARLatch    = 1'b0;
      snoop_ignore = 1'b0;
      tbst_en_    = 1'b1;
      ts_reg_     = 1'b1;
      ts_en_      = 1'b1;

  case (astate)

   A_IDLE:
    begin
     if (fetch)
      next_astate = WAIT_FOR_BG;
     else next_astate = A_IDLE;
    end

   WAIT_FOR_BG:
    begin
     BR_         = 1'b0; // Request the bus.
     NARLatch    = 1'b1; // Latch the Next Address.
     if (qual_BG_ == 1'b0)
      next_astate = MASTER;
     else next_astate = WAIT_FOR_BG;
    end

   MASTER:
    begin
     a_latch     = 1'b1; // Latch transfer attributes.
     a_en_       = 1'b0; // Enable attribute outputs.
     abb_reg_    = 1'b0; // Take the address bus.
     abb_en_     = 1'b0;
     snoop_ignore = 1'b1; // Tell snooper to ignore this transaction.
     tbst_en_    = 1'b0; // Another transfer attribute.
     ts_reg_     = 1'b0; // Start the transfer.
     ts_en_      = 1'b0;
     next_astate = TRANSFER;
    end
```

```
    TRANSFER:
     begin
      a_en_       = 1'b0;
      abb_reg_    = 1'b0;
      abb_en_     = 1'b0;
      snoop_ignore = 1'b1;
      tbst_en_    = 1'b0;
      ts_reg_     = 1'b1;
      ts_en_      = 1'b0;
      if (AACK_ == 1'b1)
       next_astate = WAIT_FOR_AACK;
      else next_astate = TERMINATION;
     end

    WAIT_FOR_AACK:
     begin
      a_en_       = 1'b0;
      abb_reg_    = 1'b0;
      abb_en_     = 1'b0;
      snoop_ignore = 1'b1;
      tbst_en_    = 1'b0;
      if (AACK_ == 1'b1)
       next_astate = WAIT_FOR_AACK;
      else next_astate = TERMINATION;
     end

    TERMINATION:
     begin
      abb_reg_    = 1'b1; // Relinquish the address bus.
      abb_en_     = 1'b0;
      next_astate = WAIT_FOR_NOT_FETCH;
     end

    WAIT_FOR_NOT_FETCH:
     begin
      if (fetch == 1'b1)
       next_astate = WAIT_FOR_NOT_FETCH;
      else next_astate = A_IDLE;
     end

    default:
     begin
      next_astate = dc_astate;
     end

   endcase
  end


//DATA BUS INTERFACE
```

```
assign qual_DBG_ = ~(DBB_ &!DBG_);

// burst_start latch
stdand2 BS_AND (.IN0(send),.IN1(clk),.Y(bs_clk));
dff #(2,0,"AUTO","AUTO")
   BurstStartReg (.CLK(bs_clk),.D(BURSTSTART),.Q(burst_start));

// Odd Parity Generator/Checker
// epoch pre_compiled parityo_chkgen256
parityo_chkgen256 DataParityGen
  (.D(data),.PIN(dparity),.ERROR(data_parity_error),.PGEN(dparity_gen));
  assign DPE_ = ~data_parity_error;

//data registers
stdbufinv TA_INV (.IN0(TA_),.Y(ta));
//Delay buffer required for timing of latch signals.  Gates = 4 results in smallest layout area.
stddelaybuf #(1,4,"AUTO") LatchDelay0(.IN0(latch0),.Y(latch0_delay));
stddelaybuf #(1,4,"AUTO") LatchDelay1(.IN0(latch1),.Y(latch1_delay));
stddelaybuf #(1,4,"AUTO") LatchDelay2(.IN0(latch2),.Y(latch2_delay));
stddelaybuf #(1,4,"AUTO") LatchDelay3(.IN0(latch3),.Y(latch3_delay));
stdand3 #("CRITICAL")
  DR0_AND (.IN0(clk),.IN1(latch0_delay),.IN2(ta),.Y(dreg0_clk));
stdand3 #("CRITICAL")
  DR1_AND (.IN0(clk),.IN1(latch1_delay),.IN2(ta),.Y(dreg1_clk));
stdand3 #("CRITICAL")
  DR2_AND (.IN0(clk),.IN1(latch2_delay),.IN2(ta),.Y(dreg2_clk));
stdand3 #("CRITICAL")
  DR3_AND (.IN0(clk),.IN1(latch3_delay),.IN2(ta),.Y(dreg3_clk));
stdbuf #("CRITICAL") DR0_BUF (.IN0(dreg0_clk),.Y(dreg0_clk_buf));
stdbuf #("CRITICAL") DR1_BUF (.IN0(dreg1_clk),.Y(dreg1_clk_buf));
stdbuf #("CRITICAL") DR2_BUF (.IN0(dreg2_clk),.Y(dreg2_clk_buf));
stdbuf #("CRITICAL") DR3_BUF (.IN0(dreg3_clk),.Y(dreg3_clk_buf));
dff #(72,0,"AUTO","AUTO")
   DataReg0 (.CLK(dreg0_clk_buf),.D({mux_out[ 63:  0],DP}),
         .Q({data[ 63:  0],dparity[ 7: 0]}));
dff #(72,0,"AUTO","AUTO")
   DataReg1 (.CLK(dreg1_clk_buf),.D({mux_out[127: 64],DP}),
         .Q({data[127: 64],dparity[15: 8]}));
dff #(72,0,"AUTO","AUTO")
   DataReg2 (.CLK(dreg2_clk_buf),.D({mux_out[191:128],DP}),
         .Q({data[191:128],dparity[23:16]}));
dff #(72,0,"AUTO","AUTO")
   DataReg3 (.CLK(dreg3_clk_buf),.D({mux_out[255:192],DP}),
         .Q({data[255:192],dparity[31:24]}));

//multiplexer
mux2 #(128,0,"AUTO","AUTO")
  MUXA (.IN0({D,D}),.IN1(DATALINE[127:  0]),.S0(mux_sel),.Y(mux_out[127:  0]));
mux2 #(128,0,"AUTO","AUTO")
  MUXB (.IN0({D,D}),.IN1(DATALINE[255:128]),.S0(mux_sel),.Y(mux_out[255:128]));
```

```
//dataline output buffer
stdbuf DATALINE_EN_BUFFER (.IN0(dataline_en_),.Y(dataline_en_buf_));
tribuf #(128,0,"AUTO","AUTO")
  dataline_bufferA (.EN(dataline_en_buf_),.IN0(data[127: 0]),
            .Y(DATALINE[127: 0]));
tribuf #(128,0,"AUTO","AUTO")
  dataline_bufferB (.EN(dataline_en_buf_),.IN0(data[255:128]),
            .Y(DATALINE[255:128]));

//data output buffers
tribuf #(64,0,"AUTO","AUTO")
   data_buffer0 (.EN(d_en0_buf_),.IN0(data[ 63: 0]),.Y(D));
tribuf #(64,0,"AUTO","AUTO")
   data_buffer1 (.EN(d_en1_buf_),.IN0(data[127: 64]),.Y(D));
tribuf #(64,0,"AUTO","AUTO")
   data_buffer2 (.EN(d_en2_buf_),.IN0(data[191:128]),.Y(D));
tribuf #(64,0,"AUTO","AUTO")
   data_buffer3 (.EN(d_en3_buf_),.IN0(data[255:192]),.Y(D));

stdbuf DEN0_BUF (.IN0(d_en0_),.Y(d_en0_buf_));
stdbuf DEN1_BUF (.IN0(d_en1_),.Y(d_en1_buf_));
stdbuf DEN2_BUF (.IN0(d_en2_),.Y(d_en2_buf_));
stdbuf DEN3_BUF (.IN0(d_en3_),.Y(d_en3_buf_));

tribuf #(8,0,"AUTO","AUTO")
   dparity_buffer0 (.EN(d_en0_),.IN0(dparity_gen[ 7: 0]),.Y(DP));
tribuf #(8,0,"AUTO","AUTO")
   dparity_buffer1 (.EN(d_en1_),.IN0(dparity_gen[15: 8]),.Y(DP));
tribuf #(8,0,"AUTO","AUTO")
   dparity_buffer2 (.EN(d_en2_),.IN0(dparity_gen[23:16]),.Y(DP));
tribuf #(8,0,"AUTO","AUTO")
   dparity_buffer3 (.EN(d_en3_),.IN0(dparity_gen[31:24]),.Y(DP));

stdtribuf dbb_buffer (.EN(dbb_en_),.IN0(dbb_reg_),.Y(DBB_));
stdtribuf ta_buffer (.EN(ta_en_),.IN0(GND),.Y(TA_));
stdbuf #("26") CANX_BUF (.IN0(cancel),.Y(CANX));

//DATA FINITE STATE MACHINE

parameter // epoch enum dstat
   D_IDLE        = 5'd0,
   WAIT_FOR_DBG  = 5'd1,
   FIRST_BEAT    = 5'd2,
   SECOND_BEAT   = 5'd3,
   THIRD_BEAT    = 5'd4,
   FOURTH_BEAT   = 5'd5,
   FETCH_TERMINATE = 5'd6,
   UPLOAD1       = 5'd7,
   ABORT1        = 5'd8,
   D_WAIT_FOR_NOT_FETCH_A = 5'd9,
```

```
  D_WAIT_FOR_NOT_FETCH_B = 5'd10,
  START_SEND     = 5'd12,
  SEND00         = 5'd13,
  SEND01         = 5'd14,
  SEND02         = 5'd15,
  SEND03         = 5'd16,
  SEND10         = 5'd17,
  SEND11         = 5'd18,
  SEND12         = 5'd19,
  SEND13         = 5'd20,
  SEND20         = 5'd21,
  SEND21         = 5'd22,
  SEND22         = 5'd23,
  SEND23         = 5'd24,
  SEND30         = 5'd25,
  SEND31         = 5'd26,
  SEND32         = 5'd27,
  SEND33         = 5'd28,
  SEND_TERMINATE = 5'd29,
  dc_dstate      = 5'bxx;

reg [4:0] /* epoch enum dstat */ dstate, next_dstate;
reg cancel,dbb_reg_,dbb_en_,dataline_en_,d_en0_,d_en1_,d_en2_,d_en3_,
   download,fetch_done,
   fetch_abort,latch0,latch1,latch2,latch3,mux_sel,send_done,upload,ta_en_;

always @(posedge clk or negedge HRESET_)
 begin
  if (!HRESET_)
    dstate = D_IDLE;
  else
    dstate = next_dstate;
 end

always @(dstate or fetch or send or qual_DBG_ or TA_ or
      data_parity_error or burst_start)
 begin

  //default values
      cancel     = 1'b0;
      dbb_reg_   = 1'b1;
      dbb_en_    = 1'b1;
      dataline_en_ = 1'b1;
      d_en0_     = 1'b1;
      d_en1_     = 1'b1;
      d_en2_     = 1'b1;
      d_en3_     = 1'b1;
      download   = 1'b0;
      fetch_done = 1'b0;
      fetch_abort = 1'b0;
```

```
    latch0     = 1'b0;
    latch1     = 1'b0;
    latch2     = 1'b0;
    latch3     = 1'b0;
    mux_sel    = 1'b0;
    send_done  = 1'b0;
    ta_en_     = 1'b1;
    upload     = 1'b0;

case (dstate)

 D_IDLE:
  begin
   if (fetch)
    next_dstate = WAIT_FOR_DBG;
   else if (send) next_dstate = START_SEND;
   else next_dstate = D_IDLE;
  end

 WAIT_FOR_DBG:
  begin
   if (qual_DBG_==1'b0)
    next_dstate = FIRST_BEAT;
   else next_dstate = WAIT_FOR_DBG;
  end

 FIRST_BEAT:
  begin
   dbb_reg_ = 1'b0;
   dbb_en_  = 1'b0;
   latch0 = 1'b1;
   if (TA_==1'b1)
    next_dstate = FIRST_BEAT;
   else next_dstate = SECOND_BEAT;
  end

 SECOND_BEAT:
  begin
   dbb_reg_ = 1'b0;
   dbb_en_  = 1'b0;
   latch1 = 1'b1;
   if (TA_ == 1'b1)
    next_dstate = SECOND_BEAT;
   else next_dstate = THIRD_BEAT;
  end

 THIRD_BEAT:
  begin
   dbb_reg_ = 1'b0;
   dbb_en_  = 1'b0;
```

```
    latch2 = 1'b1;
    if (TA_ == 1'b1)
      next_dstate = THIRD_BEAT;
    else next_dstate = FOURTH_BEAT;
  end

FOURTH_BEAT:
  begin
    dbb_reg_ = 1'b0;
    dbb_en_  = 1'b0;
    latch3 = 1'b1;
    if (TA_ == 1'b1)
      next_dstate = FOURTH_BEAT;
    else next_dstate = FETCH_TERMINATE;
  end

FETCH_TERMINATE:
  begin
    dbb_reg_ = 1'b1;
    dbb_en_  = 1'b0;
    if (data_parity_error == 1'b1)
      next_dstate = ABORT1;
    else next_dstate = UPLOAD1;
  end

UPLOAD1:
  begin
    dataline_en_ = 1'b0;
    fetch_done = 1'b1;
    upload = 1'b1;
    next_dstate = D_WAIT_FOR_NOT_FETCH_A;
  end

ABORT1:
  begin
    fetch_abort = 1'b1;
    next_dstate = D_WAIT_FOR_NOT_FETCH_B;
  end

D_WAIT_FOR_NOT_FETCH_A:
  begin
    dataline_en_ = 1'b0; // To meet data hold requirements of hsram
                         // in Data List.
    fetch_done = 1'b1;
    if (fetch == 1'b1)
      next_dstate = D_WAIT_FOR_NOT_FETCH_A;
    else next_dstate = D_IDLE;
  end

D_WAIT_FOR_NOT_FETCH_B:
```

```
 begin
  fetch_abort = 1'b1;
  if (fetch == 1'b1)
   next_dstate = D_WAIT_FOR_NOT_FETCH_B;
  else next_dstate = D_IDLE;
 end

START_SEND:
 begin
  cancel = 1'b1;
  download = 1'b1;
  latch0 = 1'b1;
  latch1 = 1'b1;
  latch2 = 1'b1;
  latch3 = 1'b1;
  mux_sel = 1'b1;
  if (burst_start == 2'd0) next_dstate = SEND00;
  else if (burst_start == 2'd1) next_dstate = SEND11;
  else if (burst_start == 2'd2) next_dstate = SEND22;
  else if (burst_start == 2'd3) next_dstate = SEND33;
  else next_dstate = START_SEND;
 end

SEND00:
 begin
  ta_en_  = 1'b0;
  d_en0_ = 1'b0;
  next_dstate = SEND01;
 end

SEND01:
 begin
  ta_en_ = 1'b0;
  d_en1_ = 1'b0;
  next_dstate = SEND02;
 end

SEND02:
 begin
  ta_en_ = 1'b0;
  d_en2_ = 1'b0;
  next_dstate = SEND03;
 end

SEND03:
 begin
  ta_en_ = 1'b0;
  d_en3_ = 1'b0;
  send_done = 1'b1;
  next_dstate = SEND_TERMINATE;
```

```
 end

SEND11:
 begin
  ta_en_ = 1'b0;
  d_en1_ = 1'b0;
  next_dstate = SEND12;
 end

SEND12:
 begin
  ta_en_ = 1'b0;
  d_en2_ = 1'b0;
  next_dstate = SEND13;
 end

SEND13:
 begin
  ta_en_ = 1'b0;
  d_en3_ = 1'b0;
  next_dstate = SEND10;
 end

SEND10:
 begin
  ta_en_ = 1'b0;
  d_en0_ = 1'b0;
  send_done = 1'b1;
  next_dstate = SEND_TERMINATE;
 end

SEND22:
 begin
  ta_en_ = 1'b0;
  d_en2_ = 1'b0;
  next_dstate = SEND23;
 end

SEND23:
 begin
  ta_en_ = 1'b0;
  d_en3_ = 1'b0;
  next_dstate = SEND20;
 end

SEND20:
 begin
  ta_en_ = 1'b0;
  d_en0_ = 1'b0;
  next_dstate = SEND21;
```

```
    end

  SEND21:
    begin
      ta_en_ = 1'b0;
      d_en1_ = 1'b0;
      send_done = 1'b1;
      next_dstate = SEND_TERMINATE;
    end

  SEND33:
    begin
      ta_en_ = 1'b0;
      d_en3_ = 1'b0;
      next_dstate = SEND30;
    end

  SEND30:
    begin
      ta_en_ = 1'b0;
      d_en0_ = 1'b0;
      next_dstate = SEND31;
    end

  SEND31:
    begin
      ta_en_ = 1'b0;
      d_en1_ = 1'b0;
      next_dstate = SEND32;
    end

  SEND32:
    begin
      ta_en_ = 1'b0;
      d_en2_ = 1'b0;
      send_done = 1'b1;
      next_dstate = SEND_TERMINATE;
    end

  SEND_TERMINATE:
    begin
      next_dstate = D_IDLE;
    end

  default:
    begin
      next_dstate = dc_dstate;
    end

endcase
```

```
 end

endmodule
```

1. Odd Parity Checker/Generator With 256 Inputs

```
/*****************************************************************************
* ODD PARITY CHECKER AND GENERATOR
* Filename: parityo_chkgen256.v
* Author: Joseph R. Robert, Jr.
* Date:   29FEB96
* Revised: 29FEB96
*
 Purpose: This module checks the parity of the input data, comparing it to the input parity. Parity is odd including
the parity bit. This module also generates the parity for the input data in groups of eight input bits.

*****************************************************************************/

module parityo_chkgen256 (D,PIN,ERROR,PGEN);

  // epoch set_attribute FIXEDBLOCK = 1

input [255:0] D;
input [31:0]  PIN;
output [31:0] PGEN;
output ERROR;

wire ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR;

parityo_chk64 parity_group_0
      (.D(D[ 63:  0]),.PIN(PIN[ 7: 0]),.ERROR(ERROR_0),.PGEN(PGEN[ 7: 0]));
parityo_chk64 parity_group_1
      (.D(D[127: 64]),.PIN(PIN[15: 8]),.ERROR(ERROR_1),.PGEN(PGEN[15: 8]));
parityo_chk64 parity_group_2
      (.D(D[191:128]),.PIN(PIN[23:16]),.ERROR(ERROR_2),.PGEN(PGEN[23:16]));
parityo_chk64 parity_group_3
      (.D(D[255:192]),.PIN(PIN[31:24]),.ERROR(ERROR_3),.PGEN(PGEN[31:24]));

stdor4 OR_A (ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR);


endmodule

//*****************************************************************************

module parityo_chk64 (D,PIN,ERROR,PGEN);
```

```
input [63:0] D;
input [7:0]  PIN;
output [7:0] PGEN;
output ERROR;

wire ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR_4,ERROR_5,ERROR_6,ERROR_7,ERROR_A,
   ERROR_B,ERROR;

paritycgo #(8,0,"AUTO","1")
  parity_group_0 (.D(D[ 7: 0]),.PIN(PIN[0]),.ERROR(ERROR_0),.PGEN(PGEN[0]));
paritycgo #(8,0,"AUTO","1")
  parity_group_1 (.D(D[15: 8]),.PIN(PIN[1]),.ERROR(ERROR_1),.PGEN(PGEN[1]));
paritycgo #(8,0,"AUTO","1")
  parity_group_2 (.D(D[23:16]),.PIN(PIN[2]),.ERROR(ERROR_2),.PGEN(PGEN[2]));
paritycgo #(8,0,"AUTO","1")
  parity_group_3 (.D(D[31:24]),.PIN(PIN[3]),.ERROR(ERROR_3),.PGEN(PGEN[3]));
paritycgo #(8,0,"AUTO","1")
  parity_group_4 (.D(D[39:32]),.PIN(PIN[4]),.ERROR(ERROR_4),.PGEN(PGEN[4]));
paritycgo #(8,0,"AUTO","1")
  parity_group_5 (.D(D[47:40]),.PIN(PIN[5]),.ERROR(ERROR_5),.PGEN(PGEN[5]));
paritycgo #(8,0,"AUTO","1")
  parity_group_6 (.D(D[55:48]),.PIN(PIN[6]),.ERROR(ERROR_6),.PGEN(PGEN[6]));
paritycgo #(8,0,"AUTO","1")
  parity_group_7 (.D(D[63:56]),.PIN(PIN[7]),.ERROR(ERROR_7),.PGEN(PGEN[7]));

stdor4 OR_A (ERROR_0,ERROR_1,ERROR_2,ERROR_3,ERROR_A);
stdor4 OR_B (ERROR_4,ERROR_5,ERROR_6,ERROR_7,ERROR_B);
stdor2 OR_C (ERROR_A,ERROR_B,ERROR);

endmodule
```

2. Odd Parity Generator With 32 Inputs

```
/*****************************************************************************
* ODD PARITY GENERATOR
* Filename: parityo_gen32.v
* Author: Joseph R. Robert, Jr.
* Date:   12FEB96
* Revised: 29FEB96
*
 Purpose: This module generates odd parity bits for group of eight inputs.
*****************************************************************************/
module parityo_gen32 (D,PGEN);

input [31:0] D;
output [3:0] PGEN;
```

```
wire [3:0] PGEN;

parityo #(8,0,"AUTO","1") parity_group_0 (.D(D[ 7: 0]),.PGEN(PGEN[0]));
parityo #(8,0,"AUTO","1") parity_group_1 (.D(D[15: 8]),.PGEN(PGEN[1]));
parityo #(8,0,"AUTO","1") parity_group_2 (.D(D[23:16]),.PGEN(PGEN[2]));
parityo #(8,0,"AUTO","1") parity_group_3 (.D(D[31:24]),.PGEN(PGEN[3]));

endmodule
```

H. TEST RESULTS

```
Host command: verilog
Command arguments:
  -f verilog_arguments
    -v /tmp_mnt/h/joshua_u2/jrrobert/thesis/epoch/primlib.v
    prc.v
    prc_top.v
    sequencer4.v
    tarbiter.v
    tcpu.v
    testbench.v
    tmemory.v

VERILOG-XL 2.1.2 log file created Mar 19, 1996  11:53:03
VERILOG-XL 2.1.2  Mar 19, 1996  11:53:03

Copyright (c) 1994 Cadence Design Systems, Inc.  All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1994 UNIX Systems Laboratories, Inc. Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC. USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

        Cadence Design Systems, Inc.
        555 River Oaks Parkway
        San Jose, California 95134
```

```
For technical assistance please contact the Cadence Response Center at
1-800-CADENC2 or send email to crc_customers@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkverilog@cadence.com

Compiling source file "prc.v"
Compiling source file "prc_top.v"
Compiling source file "sequencer4.v"
Compiling source file "tarbiter.v"
Compiling source file "tcpu.v"
Compiling source file "testbench.v"
Compiling source file "tmemory.v"
Scanning library file "/tmp_mnt/h/joshua_u2/jrrobert/thesis/epoch/primlib.v"
Scanning library file "/tmp_mnt/h/joshua_u2/jrrobert/thesis/epoch/primlib.v"

Warning! Implicit wire has no fanin                [Verilog-IWFA]
      "prc.v", 23159: NC0

Warning! Implicit wire has no fanin                [Verilog-IWFA]
      "prc.v", 23159: NC1

Warning! Implicit wire has no fanin                [Verilog-IWFA]
      "prc.v", 23159: NC0

Warning! Implicit wire has no fanin                [Verilog-IWFA]
      "prc.v", 23159: NC1
Highest level modules:
testbench

  *** SDF Annotator version 1.6_beta.3
  ***   SDF file: /tmp_mnt/h/joshua_u2/jrrobert/thesis/verilog/hardware/prc.sdf
  ***   Back-annotation scope: testbench.PRC1.PRC1
  ***   No configuration file specified - using default options
  ***   SDF Annotator log file: sdf.log
  ***   No MTM selection parameter specified

  ***   No SCALE FACTORS parameter specified

  ***   No SCALE TYPE parameter specified

      Configuring for back-annotation...

      Reading SDF file and back-annotating timing data...


*** SDF back-annotation successfully completed
PRC granted the data bus.
(ERROR): WR and A are both unknown at time 6.700
(ERROR): WR and A are both unknown at time 6.700
```

```
(ERROR): WR and A are both unknown at time 6.700
(ERROR) WR transition to unknown and (din != MEM[a]) at time 7.000
     Instance: testbench.PRC1.PRC1.LM1.MRMA_list.hsram.inst1
(ERROR) WR transition to unknown and (din != MEM[a]) at time 7.000
     Instance: testbench.PRC1.PRC1.DL1.data_ram1.hsram.inst1
(ERROR) WR transition to unknown and (din != MEM[a]) at time 7.000
     Instance: testbench.PRC1.PRC1.DL1.data_ram0.hsram.inst1
System hard reset at time          35.
CPU started read from address 00000000 at time          45.
   CPU read: 0001020304050607 at          211
   CPU read: 08090a0b0c0d0e0f at          271
   CPU read: 1011121314151617 at          331
PRC requested the bus.
   CPU read: 18191a1b1c1d1e1f at          391
CPU started read from address 00000020 at time          420.
   CPU read: 2021222324252627 at          556
   CPU read: 28292a2b2c2d2e2f at          616
   CPU read: 3031323334353637 at          676
   CPU read: 38393a3b3c3d3e3f at          736
PRC granted the data bus.
CPU started read from address 00000180 at time          1215.
   CPU read: 0001020304050607 at          1381
   CPU read: 08090a0b0c0d0e0f at          1441
   CPU read: 1011121314151617 at          1501
   CPU read: 18191a1b1c1d1e1f at          1561
CPU started read from address 000001a0 at time          1665.
   CPU read: 2021222324252627 at          1831
PRC requested the bus.
   CPU read: 28292a2b2c2d2e2f at          1891
   CPU read: 3031323334353637 at          1951
   CPU read: 38393a3b3c3d3e3f at          2011
PRC granted the data bus.
CPU started read from address 00000040 at time          2490.
   CPU read: 4041424344454647 at          2641
   CPU read: 4041424344454647 at          2656
   CPU read: 5051525354555657 at          2671
   CPU read: 4041424344454647 at          2686
PRC requested the bus.
PRC granted the data bus.
CPU started write to address 000001c0 at time          3307.
   CPU write beat 1: 7777777777777777 at          3322
   CPU write beat 2: 8888888888888888 at          3488
   CPU write beat 3: 1111111111111111 at          3548
   CPU write beat 4: 3333333333333333 at          3608
CPU started read from address 00000060 at time          3765.
   CPU read: 6061626364656667 at          3916
   CPU read: 6061626364656667 at          3931
   CPU read: 7071727374757677 at          3946
   CPU read: 6061626364656667 at          3961
PRC requested the bus.
```

```
PRC granted the data bus.
CPU started read from address 000001c0 at time          4440.
   CPU read: 7777777777777777 at          4606
   CPU read: 8888888888888888 at          4666
   CPU read: 1111111111111111 at          4726
   CPU read: 3333333333333333 at          4786
L125 "testbench.v": $finish at simulation time 5035000
4 warnings
158647 simulation events + 266655 accelerated events + 926440 timing check events
CPU time: 6.1 secs to compile + 161.8 secs to link + 377.5 secs in simulation
End of VERILOG-XL 2.1.2   Mar 19, 1996  12:15:44
```

LIST OF REFERENCES

Aguilar F., M.E., "Testing of a Read Prediction Buffer Integrated Circuit and Design of a Predictive Read Cache," Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1995.

Billingsley, A.B. and D.J. Fouts, "Memory Latency Reduction Using an Address Prediction Buffer," *Twenty-Sixth Asilomar Conference on Signals, Systems, and Computers*, Vol.1, pp.78-82, 1992.

Fouts, D.J. and A.B. Billingsley, "Predictive Read Caches: An Alternative to On-Chip Second-Level Cache Memories," *Journal of Microelectronic Systems Integration*, Vol.2, No.2, 1994.

Hennessy, J.L. and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

Miller, R.W., "Simulation and Analysis of Predictive Read Cache Performance," Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1995.

Nowicki, G.J., "The Design and Implementation of a Read Prediction Buffer," Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1992.

PowerPC-603, *User's Manual*, IBM Microelectronics and Motorola, 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Dr. Douglas Fouts, Code EC/Fs 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Frederick W. Terman, Code EC/Tz 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

6. Raymond Bernstein, Code EC/Be 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

7. LT Joseph R. Robert 1
   49 Clifton St.
   Manchester, NY 14504