A REAL-TIME SOFTWARE PERFORMANCE ANALYSIS CHIP

Mehul S. Dave
B.E., University of Bombay, India, 1987

A thesis submitted to the faculty
of the Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Electrical Engineering

September, 1990

The thesis "A Real Time Software Performance Analysis Chip" by Mehul S. Dave has been examined and approved by the following Examination Committee:

Charles L. Saxe,
Thesis Advisor,
Tektronix Inc.

Dr. John Murray
Associate Professor,
Department of Electrical and Computer Engineering,
Oregon State University, Corvallis, OR

Dr. V. S. Rao Gudimetla
Assistant Professor,
Department of Applied Physics and Electrical Engineering,
Oregon Graduate Institute of Science and Technology

Dr. Raj Solanki
Associate Professor
Department of Applied Physics and Electrical Engineering,
Oregon Graduate Institute of Science and Technology

ACKNOWLEDGMENTS

I would like to thank several people without whose help this project would not have been possible. My thanks to Dr. V. S. Rao Gudimetla who got the project for me and who was extremely supportive all through the project. I am also thankful to Dr. Paul Davis for supporting this project and to Dr. John Murray and Dr. Raj Solanki for agreeing to examine this thesis.

Special thanks are due to my thesis advisor, Chuck Saxe. He provided valuable guidance throughout the project and taught me many things. Without his guidance and support, this project would not have been possible. My thanks to all the members of the Advanced Development Group (ADG) at Tektronix. In particular, thanks to Craig Overhage for suggesting this project and his help with critical path analysis using the fastta timing analyzer. Thanks to Dave McKinney for designing new standard cells almost on demand and for giving extremely helpful suggestions about the design of the chip. Finally, I am thankful to Tektronix Inc. for funding this project.

Table of Contents

Abstract vii

1. Introduction 1

2. Background 3

   2.1 Need for testing and performance analysis of software 3

   2.2 Types of testing 5

   2.3 Real time needs and traditional methods of testing and performance analysis 6

   2.4 Progress toward comprehensive real-time testing techniques and tools 9

3. Macroarchitecture of the chip 14

   3.1 Goals of this project 14

   3.2 Discussion of possible architectures 16

   3.3 Description of macroarchitecture of the chip 20

4. Microarchitecture of the chip 25

   4.1 Introduction 25

   4.2 Global view of the microarchitecture 26

      4.2.1 The Processing Section 26

      4.2.2 The Input Section 30

      4.2.3 The Output Section 33

   4.3 The Range Recognizer Circuit 35

4.4 The Priority Resolver Circuit 41

4.5 The RAM circuit 44

4.6 Results and discussion 49

5. Summary and conclusions 59

References 61

Appendix A 63

Appendix B 67

Appendix C 74

Appendix D 104

List of Figures

Figure 1 27

Figure 2 36

Figure 3 42

Figure 4 45

ABSTRACT

A REAL TIME SOFTWARE PERFORMANCE ANALYSIS CHIP

Mehul S. Dave, M.S.

Oregon Graduate Institute of Science and Technology, 1990

Supervising Professor: Charles L. Saxe

An essential stage in software development is debugging the code and checking it for correctness. Another related stage is performance analysis of the developed software which involves making sure that the program in question runs at the optimal speed required for a particular application.

In many real time applications, fast and efficient code is needed. However, the techniques and tools for the debugging and performance analysis of real time software are still, by and large, primitive and have not kept pace with the advances in software technology. Traditional hardware tools such as logic and state analyzers are useful only at the machine language or assembly language level. Software based performance analysis tools such as program profilers interfere with the real time behavior of the program by inserting additional code and thus do not provide accurate information. A tool to perform non-intrusive performance analysis at the source level for programs written in high level languages is needed. In this thesis, the design and implementation of a CMOS chip to aid high level non-intrusive performance analysis is reported.

1. INTRODUCTION

The testing and validation of programs has received considerable amount of attention in the past decade. With the increasing availability and use of high level languages, the programmer has moved increasingly away from the low level machine details and more complex abstractions can now be implemented with ease. With the increasing complexity of programming projects, the need for testing and debugging the code becomes even more crucial. To handle the increasing complexity, the tools for debugging and testing also must become more sophisticated.

Very often, it is not only necessary to test the program for correctness but also performance. There are programs that may run correctly i.e they may perform the necessary function but execute too slowly. This has sometimes been called a "performance bug".[1] It would be desirable to do some performance analysis of such a program to improve its running time. In real time systems, it is often mandatory to make such improvements. Consider, for example, a mechanical arm in an assembly line that moves at fixed intervals of time. The software driving the arm must be capable of finishing its task within this interval. A tool to facilitate performance analysis of such software would be extremely helpful to programmers. However, there are very few tools available to facilitate the real time programmer in testing and debugging. As Robert Glass puts it,[2] the world of debugging and testing of real time software is a "lost world".

The techniques and tools available to the programmer of real time systems have not kept pace with the advances in software. A typical real time program is still

tested at the machine language or the assembly level. Traditional tools such as logic and state analyzers help the programmer in debugging at the machine language or assembly language level but are not useful to do performance analysis at the high level in which programs are typically written. Software tools such as profilers, where available, are not useful for real time systems since they are intrusive by nature and thus do not give accurate measure of the timing of some sections of the code.

In this thesis, I will discuss the design and implementation of a CMOS chip which is intended to be a hardware tool for non-intrusive real time performance analysis and which can be used at high level with proper software. Chapter 2 presents the history and background of software testing and performance analysis. Some other tools developed for non-intrusive performance analysis and related work are also discussed.

In chapter 3, the macroarchitecture of the chip is discussed. Various modes and features of the chip and the reasons for choosing a certain architecture instead of other possible architectures is discussed.

Chapter 4 discussed the microarchitecture and the logic level details of the chip. The results of simulation using the Tektronix ADG standard cell library are presented. The final layout of the chip and the simulation after the layout are also presented.

Discussion of results, conclusions and suggestions for further research are presented in chapter 5.

2. BACKGROUND

2.1 Need for testing and performance analysis of software

The need to test software is obvious. The program should be able to implement the function that it is designed to implement. Since any program of a reasonable size can, and almost always does have bugs, one needs to test the programs for their correctness. The amount, scope and methods of testing of programs may vary depending upon several factors such as the nature of the application, the resources available to the software developer, ease with which testing can be done etc. However, at the very least, functional testing of a program must be done, i.e. it must be made certain that the program implements the required function correctly and that there are no bugs in the program. Detection and elimination of redundant code, unused variables, bad programming constructs, incorrect logic, rare conditions which can lead to bugs and optimizing code for performance etc. are among the goals of testing to develop more reliable software. Bug free code is always desirable and a little time invested in testing can uncover sources of potential later disasters.

The level of testing done depends on several factors such as the nature of the programs, the tools available, the cost, the nature of the application etc. In certain cases, extensive testing must be done to make certain that no bugs are left in the code because the costs of errors, both tangible and intangible can be enormous. Consider, for example, programs for missile control or control of equipment in a nuclear power plant. In other, less critical applications, such as for example, payroll processing, the standards for testing and optimization may not be as strict as the former case because

such standards are not needed and neither are they economically viable. The amount of testing done is also largely dependent on the ease of testing. If sophisticated tools are available to enable fast, easy, efficient, comprehensive and cost effective testing, the programmer will devote more time to testing since bug free code is always desirable. On the other hand, in absence of such tools, only as much of testing as is absolutely necessary, will be done. It is clear, however, that irrespective of the level of testing done, it is almost mandatory to perform some testing on programs.

The increasing complexity of software emphasizes the need for testing even more. As software has evolved from the machine language to assembly language to finally the high level languages of today, the programmer is increasingly shielded from the low level machine details. It is now possible to implement increasingly complex abstractions much more easily by using sophisticated compiler technology. Software written using such languages can thus perform functions far more complex and far wider in scope than before. This also means, however, that the bugs in such software are more difficult to uncover and that the likelihood of having bugs is increased because several layers of software lie between the programmer and the hardware. Therefore not only are more sophisticated tools for debugging needed but also more extensive testing of code is needed. The tool for testing or performance analysis has to interpret the low machine level events and pass through several layers of abstractions to output data at the source level. Thus, tools with greater functionality and more sophistication are needed.

While it is true that unlike debugging, it is not necessary to do performance analysis in all cases, in many applications it is desirable and even mandatory to undertake performance analysis. Real time applications are a very good example. Real time software is constrained by latencies between events. The software must complete its task in the time between inter-event latencies. It is often the case that while the software implements the required function, it runs too slowly and may be

unacceptable for a real time application. In such a case, performance analysis of such software is necessary to ascertain which areas of the code need improvement. Even in case of batch processing, if the programs run for a long period of time, it would be desirable to analyze the program to discover the performance bottlenecks and correct them so that the program executes faster. Thus performance analysis of code is also a crucial part of testing for several applications.

2.2 Types of testing

Considerable attention has been directed in the last decade to the subject of testing and reliability of software. A lot of literature has been published on the subject (see, for instance[3-5]). Tools such as static analyzers, program provers, profilers and source level debuggers are more widely available. The testing techniques may be broadly classified as *static testing* and *dynamic testing*. In static testing, the source code is run through a program which analyzes the code for certain errors that can be detected solely by looking at the text of the program. Such a tool can identify such errors as incorrect data constructs, uninitialized variables, unreachable code, non-portable code etc. An example of such a tool is *Lint*6 for the C programming language. In dynamic testing, the program is executed in a controlled manner so as to test that the required functions execute correctly. This generally involves giving the program a set of predetermined inputs and observing the output.

As discussed previously, several times it may be the case that the program in question executes correctly but it executes too slow for the task at hand. Performance analysis of the program needs to be done for such a case to determine in which part of the code does the program spend most of its time. The efforts may then be directed at this part of the code to make it more efficient by rewriting it. Program profilers are the tools designed to facilitate performance analysis of a program. Typically, a program profiler inserts counters at critical nodes in the code and the counters are incre-

mented dynamically when that part of the code is reached. Then, based on the available count and a statistical sampling of the program counter during the execution of the program, the profiler gives the data about which part of the code is most frequently used. The *gprof*[7] profiler on UNIX† operating system is an example of such a tool.

The tools for testing and performance analysis may be hardware or software tools. Programs such as static analyzers and program profilers are software tools. On the other hand, one can use hardware tools such as logic analyzers, state analyzers and in-circuit simulators also as tools for testing and performance analysis. Such hardware tools most often monitor the signals at the pins of the microprocessor or the bus and when a certain combination of signals is detected, they start taking the data till a certain point of time. They also may be time triggered i.e. at regular intervals of time, they interrupt the microprocessor for a brief period and acquire data and store it for later analysis.

2.3 Real time needs and traditional methods of testing and performance analysis

Most of the traditional tools of testing and performance analysis are ill suited or inadequate for testing real time software. As Robert Glass observes,[2] the real-time debug and test is still a "lost world" compare to the "civilization" developed in other areas of software (referring to the advances of software from assembly level to high level languages and development of fast compilers and link loaders). Not much work has been done in the area since Glass's 1980 paper. There are several reasons why debugging and performance analysis of real time software remains a difficult job even with so much work done in the general area of testing and reliability of software. All these reasons stem from the nature of real time applications; its

† UNIX is a trademark of AT&T Bell Laboratories.

requirements and the limitations imposed by its environment. A discussion of these follows.

Real time software is software running on a computer that interacts with functioning external devices. It is called real-time because the software actions control activities that are occurring in an ongoing process.[2] The task of developing reliable real time software is particularly more difficult for several reasons. Most of the times, such a software is part of a larger system (sometimes also called embedded systems) and must interact with several different devices often functioning asynchronously. In contrast with the traditional computer systems, such systems do not have the sophisticated supporting software such as complex operating systems, debuggers, program profilers and other utility programs. In fact, in most of the systems, the hardware is cheap, light and small due to economic reasons. Consider, for example, a computer driving a missile system. It is a throw away piece of hardware and it is therefore necessary to make it as cheap as possible. This results in complexity being transferred from hardware to software. It is also desirable to transfer complexity from hardware to software since multiple copies of software do not cost much but every additional copy of hardware costs. Also, many times due to the nature of environment in which a real time system operates such as excessive heat or moisture, it is not possible to use complex hardware which requires more stable environmental conditions. So the function must be taken care of by software again. To add to this already bad picture, real time software is constrained by latencies between events. The software must be able to complete its task within the time between interevent latencies. So the real time software developer has to develop more powerful software in a more primitive environment.

Given these conditions, it follows that a tool to facilitate debugging and performance analysis which is geared towards real-time environments would be extremely valuable to the real-time developer. Most of the traditional tools for testing and

debugging are, however, not geared for the real-time environment. Static performance analyzers are not too useful because most of the software developed for real-time applications is developed in a host-target environment. The program is developed on the host, a more powerful and more sophisticated computer and then cross-compiled for the real-time environment. The static analyzer can be useful in the host environment and it will eliminate certain bugs but it is not enough.

Program profilers are not useful since they are intrusive by nature. They add some additional code to ascertain the real-time behavior of the program and thus provide an inaccurate measure of the timing. They often intrude to the point where the distortion is significant and thus the tool loses its effectiveness. Also, in several applications such as embedded systems for instance, it may not be possible to insert additional code into the real-time program. In such systems, the program is hard coded and thus introduction of any additional piece of code is impossible. Also, such a tool will often miss some crucial event which occurs rarely and is a source of a bug. The program profilers also obtain their data on timing by sampling the location of the program counter. This technique is inherently statistical in nature and thus several runs of the program may be necessary before the accumulated data reaches as acceptable level of confidence.

Hardware tools are most commonly used for real-time debugging and testing. Tools such as logic and state analyzers, in circuit simulators etc. are very useful for debugging of real-time programs. These tools observe the signals at the pins of the microprocessor or the bus and acquire the data when a particular sequence of events is observed. Some of these tools use an interrupt technique based on event triggering. The tool interrupts the microprocessor when it detects a predefined event and acquires data and stores it for later analysis. Such a technique, by its nature is intrusive and is not suited for performance analysis. Other tools start data acquisition by storing some signals on the bus when they detect a particular event and stop the

acquisition after a period of time. This technique is not intrusive but since the amount of data to be stored is so much, typically the tool can only save the activity for a few milliseconds. When its buffer is exhausted, it cannot store any data any longer. Such a tool is not useful for observing the behavior of the program for extended periods of time.

The most serious limitation of traditional hardware tools, however, is that they cannot display the results of the program at the source level. Typically, they give data either in terms of machine language level or assembly level. This is a very serious limitation. The programmer, thus, has to be knowledgeable about assembly language and other low level details of the system he is working on. Thus more training effort has to be expended. And more importantly, since the low level machine details differ from one system to another, the training effort has to be expended for every new system. Moreover, debugging the program at assembly level is more difficult and time consuming. This combined with the fact that in real time systems, programmers typically have to deal with additional complexity transferred from hardware makes it clear that a tool which affords source level analysis of real-time software is required.

2.4 Progress toward comprehensive real-time testing techniques and tools

There have been a few attempts to develop tools and techniques particularly suited for real-time applications. There have even been attempts at a systematic study of the field and formalization.[8] Some of the tools reported were developed exclusively for some particular system [9,10] and while these make important observations, they are not helpful for systems other than those for which they are designed. There were also some early attempts which suggested hardware support for debugging and testing but these did not strictly deal with the subject of real-time testing but rather with other related subjects such as multiprogramming environments.[11]

Many of the other tools developed focussed on the problem of debugging and testing and proposed having large buffers to store all the activity of the program from a certain point and then try to reconstruct the program to trace back to the point where the error occurred.[10, 12] Such tools are suited more for the task of debugging and not for performance analysis and they also propose storage of large amount of data, quite a bit of which may be useless. Besides, the problem of being able to store only limited amount of data due to limited size of buffer remains and these tools, therefore cannot be used for programs which execute for long time. The tools also require a reconstruction of the program after its execution and it may not be easy to accomplish this in all cases and the effort required to develop the software to reconstruct the program from the stored data is quite large.

There are other tools proposed[13] which are of more interest since they propose real time monitoring of the program and storing only the essential data. Plattner[14] proposes a tool to facilitate real time monitoring of programs and develops means to specify actions to be taken by the monitoring program based on predicates formed on the text of the source program. He describes, in detail, an elaborate hardware scheme to accomplish this objective and reports the development of a prototype. A similar tool is reported by Bemmerl.[15]

The main advantage of these tools is that they afford real-time analysis at the *source level*. This is a significant improvement over other methods proposed. It is also an attempt to bring the tools for debugging and testing up to date with the advances in software technology. Such a tool would be of great help to a real-time software developer particularly if working in host-target environment where the software is written in a high level language on a host and cross compiled for the target. However, there are some disadvantages. Most significantly, the disadvantage is the limited capability of the tool. For instance, in the prototype implemented by Plattner, only 10 monitoring actions can be specified for a single run of the program.

This limitation overshadows most of the advantages of the tool. The tool offers a variety of functions. Any sort of monitoring function can be specified to aid debugging. One can keep track of changes in values of a particular variable, trace execution of a set of statements, set counters based on conditions etc. But only 10 of such actions can be executed for one run of the program. So, most of the capabilities of the tool which make it potentially useful for real-time applications are rendered useless since only a few can be used at a time.

There has been progress made even in logic and state analyzers.[16] The state analyzers have become more sophisticated. Tools such as microprocessor analyzers can be used to interpret the activities on the pins of a specific microprocessor and relate these signals to high level events. However, these are microprocessor specific and not useful in general for several systems. Hewlett-Packard's HP 64340A software analyzer is another hardware tool to aid software analysis.[17] The tool helps the analysis of software at high-level. The disadvantage of this tool, like Plattner's, is that the tool has limited capability. It can only monitor four ranges of code for a given run of the program.

Another tool recently reported overcomes several of the disadvantages of the tools mentioned above. *Softanalyst* by Northwest Instruments Inc., offers the capability to do non-intrusive source level performance analysis in real-time.[18-21] The tool offers several capabilities. The symbolic trace option lets the user keep a trace of the order of program execution and trace individual statements. Performance analysis mode lets the user time up to 80 ranges of code and the code coverage option keeps a track of which statements of the program were executed in a single run. The significant feature of the tool is that it allows the user to observe the performance of the program at source level. By looking at the symbolic table generated by the compiler, the tool relates the software events such as entry into a procedure and exit from it or activity of a particular variable etc. to their hardware equivalents. The tool gath-

ers the data, processes it and displays the results back in terms of source level constructs. It has many more ranges than the other tools and unlike logic analyzers, it can gather execution data for the programs having execution time of the order of seconds rather than the typical milliseconds for logic analyzers.

Softanalyst represents a significant advance in non-intrusive, real-time, source-level performance analysis. However, in spite of its wider capabilities and resources, it is often not extremely useful. The timing of code is the important capability offered by it. But owing to the architecture of the tool, it can only store data for programs whose execution time is of the order of seconds. After that, its buffer overflows and it can no longer gather data. It also requires a relatively complex front-end probe for each specific microprocessor which can provide it the necessary signals to detect the occurrence of hardware events. The Softanalyst is event-triggered. It detects the occurrence of up to 256 events on the bus and then starts its timing when the occurrence of the event is detected and stops when another event is detected. This presents some problems for timing of code. Firstly, this means that one cannot satisfactorily time procedures with multiple entries and exits. Secondly, it becomes difficult for it to account for occurrence of interrupts and it is also difficult to use for multi user systems.

It should be observed that in all the previous efforts to develop a tool for debugging and performance analysis, the main bottleneck was the limitation of resources of the tool. Most tools offer very advanced capabilities but only a few of them can be used at a particular time. The goal of the designers of the system was to offer a tool which can handle several different tasks. This increased the complexity of the job and resources were limited because each different task consumes away a part of the limited resources available. In my opinion, it would be better to concentrate the resources on a particular task rather than to divide them among several tasks. One can design a small but efficient tool which would handle a particular task such as per-

formance analysis and leave tasks such as debugging to other tools. A system can then be constructed out of several different components, each dedicated to a particular task. This would achieve a more efficient utilization of resources and better capabilities for each particular task. The next chapter discusses the goals of this project with this perspective and the macroarchitecture of the chip.

3. MACROARCHITECTURE OF THE CHIP

3.1 Goals of this project

The goal of this project was to develop a very cheap and effective way to do performance analysis. We wanted to develop a tool which can time the execution of code for extended periods of time and which uses relatively simple circuitry interfacing with the system bus to detect events. Since this tool was to be used for non-intrusive analysis in real time, it could not be implemented software. A part of it would have to be hardware which, in turn, is driven by software. We decided to try to implement a chip which can allow a user to monitor several ranges of the program and gather performance analysis data and store it in on-chip memory which can later be read off by appropriate software. This thesis discusses only the chip. The software to run the chip has not yet been developed.

As discussed in the previous chapter, the problem with most tools was that the number of ranges which could be simultaneously monitored for gathering performance analysis data, was too small. One of the goals of the project was to try to overcome this limitation. The available chip area would constrain this number. But if one could implement a relatively simple design so that each chip would be very cheap, one could use several chips for a system and thus provide a large number of ranges capable of collecting data for extended periods of time which can be used for performance analysis purposes. These chips used in conjunction with a state or a logic analyzer would make a very effective tool for non-intrusive debugging and performance analysis.

There was the choice of implementing several features to enable debugging and performance analysis. However, putting more features would involve implementing complex logic on the chip. This involves a large amount of time and effort and increased costs. More importantly, however, there is a trade off involved between more features and increased capability for performance analysis. This is so because real time performance analysis involves gathering and processing of data at least as fast as the program being analyzed executes so that no data is missed. Since there are always limitations on the amount of data that can be stored and processed in a certain interval of time, one can either have large number of features but a few of which can be used at a time due to the limitations of chip area or one can implement a relatively small number of features, all of which can be used and which can be used much more effectively. In designing chips, chip area is always a precious resource and it is always beneficial to maximize the usage of all the logic. Therefore, the latter approach seems preferable. Another advantage offered by the latter approach to design is that one does not have to implement complex logic since the number of functions that the hardware is required to perform, is limited. Therefore, the latter approach to design was chosen i.e. implementing a relatively small number of features but try to maximize their capabilities and usage. All features to help debugging of code were, therefore, excluded and focus was on performance analysis.

Many of the debugging features can be more conveniently handled through other means. Good static analyzers can uncover several errors from the source text of the program. Such tools are typically available on large operating systems. Since real time software development is often done in host-target environments, static analyzers available on the host can be used to eliminate some errors from the programs. Source level debuggers are also increasingly available on large systems. So the programmer already has powerful tool to facilitate debugging on the host. The debugging features useful for real time environment such as monitoring memory locations for changes in

values, gathering data after occurrence of a particular event etc. can be provided by a state analyzer.

Therefore, the merit of this approach to testing is that the task of testing is thus distributed to different tools. Each tool is dedicated to a particular task and thus its resources can be entirely devoted to that task. One can build a powerful testing system by combining such tools. Some part of the system helps debugging and other part does performance analysis. In this manner, maximum utilization of all resources of the system can be achieved and it is also more economical and efficient to implement the system in this manner.

The chip designed, therefore, only offers the capability to do performance analysis. But since its range of features is now delimited, one can devote the freed resources to provide additional capabilities. For instance, this chip was designed to run at a clock frequency of 100 MHz; fast enough for the current generation, high speed RISC microprocessors. It maintains a 48 bit counter for measuring the time spent for each range and a 32 bit counter for measuring the number of entries into the range. At the clock speed of 100 MHz (cycle time of 10 ns), this means that data can be collected for execution time of approximately 78 hours for each range. Thus, data can be collected for long, uninterrupted runs of programs. This is a very desirable performance analysis feature which is absent from the other tools discussed previously. This benefit is, of course, obtained at the cost of eliminating other features but considering the fact that a low cost tool such as this, in conjunction with a traditional tool such as a state analyzer, can provide a powerful software testing tool, it is a good tradeoff.

3.2 Discussion of possible architectures

As discussed previously, the feature required of the chip was the ability to time sections of a code for extended periods of time. There are at least two significantly

different architectures which can accomplish the same objective. In the following, the pros and cons of both and why one was chosen over the other, are discussed.

The first approach to architecture is the event triggered approach taken by *SoftAnalyst*.[18] In this approach, some particular function of the tool is triggered when the tool detects the occurrence of a predefined event such as, for instance, entering or exiting a procedure. The tool starts collecting data when the event is detected and stops collecting data when another event is detected. For instance, in *SoftAnalyst*, the user can specify up to 256 events. The tool maintains an on-chip timer and a status qualifier. When it detects the occurrence of any one of the defined events, it saves some tags from the status qualifier, the time of detection and the event number in a FIFO (first-in-first-out memory). The data in the FIFO is then compressed by a data compression chip which, in turn, feeds another microprocessor which analyzes the data.

There are some advantages to this approach. Firstly, since event detection is just a question of ANDing the data bits with a set of latches, this function is easy to implement and also takes very little of chip space.† Secondly, the interpretation of the events is not restricted by the hardware and is up to software. In *SoftAnalyst*, for instance, the same event recognizers are used to perform several functions such as tracing values of variables, recognizing entry to or exit from a procedure etc.

There are, however, disadvantages with this approach. The most serious problem with it is the storing and processing of data. The data is typically stored into a buffer (the FIFO in *SoftAnalyst*) which is emptied at regular intervals. The data must be processed fast enough so that the buffer does not overflow. This bottleneck is very difficult to overcome particularly with today's high speed microprocessors. The data would be, typically, written into memory which is generally a RAM . It is

---

† It should be noted that *SoftAnalyst* is not implemented on a single chip but by using several other chips. This, however, is irrelevant from the viewpoint of architecture. The same architecture could, in theory, be implemented on a chip.

difficult to design RAMS which could be read and written at fast rates. The speed requirement of this project was 100 MHz. This would mean a 10 ns read/write accessible on-chip RAM, clearly a very difficult task. The problem, however, does not end here. Even supposing that a 10 ns RAM could be designed, one is still left with the requirement that post processing must be done at the rate of 10 ns. Taking the example of maintaining a count of time spent in a procedure, if the count was to be 32 bits long, this would mean doing a 32 bit addition in 10 ns in the worst case. There are ways to design around this problem but the logic becomes quite complicated.

Another small disadvantage with this approach is that it is very difficult to time procedures with multiple entries and exits. It is also difficult to account for interrupts. The timer needs to be stopped when executing interrupts. This requires front-end circuitry to detect interrupts. If the target system is a time sharing system, it is not possible to have a count of the timing unless some signal to stop the timer can be provided. In systems where such a signal is not available, this architecture is not useful.

Finally, there is the disadvantage of limited size of the buffer. Since the buffer is limited in size, it is not possible to time programs which run for extended periods of time. The buffer of *SoftAnalyst* for instance, fills up if the time of execution is of the order of seconds. This architecture is, thus not well-suited for the requirements of the project.

An alternate approach to the problem of timing the code is to count the number of memory addresses on the system bus which lie in a particular range. Programs are arranged sequentially in memory with instructions and data interspersed. The microprocessor puts out either instruction fetches or data fetches on the system bus during its execution. If one could detect and store the instruction fetch addresses, compare them with preprogrammed limits and detect if they are in or out of that range, one could maintain a count of the number of addresses detected in a range.

The count is approximately proportional to the time spent in the range and thus performance analysis data is obtained.

The disadvantage of this approach is that the count obtained is not accurate. But it is possible to overcome this limitation if additional signals from the system are available. One can continue counting between two successive instruction fetches assuming that the microprocessor is either executing the instruction or issuing data fetches. With such an arrangement, accurate timing information can be provided. Of course, this assumes the capability to stop the counting by an appropriate signals when needed. But when such a signal is not available, the tool can still provide a fairly accurate, though not exact, indication of where the program spends most of its time during execution. Another disadvantage of this approach is that range comparators are needed at the input to compare the input address with preprogrammed limits. In contrast to a simple ANDing of bits, significant amount of logic is now required to accomplish this function. This demands more chip area.

The most important advantage of this approach, however, is that it needs very little post processing of data and it does not need to store large amounts of data in memory. Only a count of the addresses being in or out of a range need be maintained which is not very difficult to accomplish in hardware. Since there is no buffer, the problem of having too much incoming data is completely eliminated. All that needs to be done is to maintain a counter for each range which is incremented when an address is in the range. This architecture is well suited for a chip since it is relatively simple to implement. It does not require fast on chip buffers and complicated logic to process the data in the buffer. Since this architecture appears much more appropriate for the requirements of the project, it was chosen for the chip in preference to the approach taken by *SoftAnalyst*.

The capability to do performance analysis at the source level is easily provided if the symbol table generated by the compiler is available. From the symbol table,

appropriate software can obtain the entry and exit points of all the procedures and program the upper and lower limits of the range recognizers for the areas of codes which the user wants to time. If the number of ranges available does not cover all the procedures of a program, one can set up the chip so that it divides the address space of the program evenly among the available range recognizers. After one run of the program, the area of code where maximum time is spent can be narrowed down and then in the second run of the program, the chip can be set up to time various parts of that section of the code.

3.3 Description of macroarchitecture of the chip

The chip consists of several range recognizers (sixteen in the version implemented for this project), each of which contains the following.

1. Two 32 bit latches which hold and lower and upper limit addresses for comparison (`ul[0:31]` for upper limit and `ll[0:31]` for lower limit). The recognizer increments its counter when the incoming address is greater than or equal to the lower limit and less than or equal to the upper limit. The data at the input pins is assumed to be 32 bits. The input data is used to write into the latches when the `prg_chip` signal is raised. The data is written into the lower limit latch if the `limit` signal is low and it is written into the upper limit latch if the `limit` signal is high. A 4 bit range recognizer number must also be input along with the data for the lower and upper limits.
2. A 48 bit counter which is incremented whenever the range recognizer detects an address that lies between the two limits and the `active` bit of the range recognizer is set.
3. An `active` bit which is set when the range recognizer detects a valid address within the programmed range. The bit is reset when the range

recognizer detects a valid input address which is outside the programmed range.

4. A 32 bit entry/exit counter which is used to count the number of times a range was entered. The counter is incremented when a valid input address is outside the programmed range and when the `active` bit was set in the previous cycle. That is, the count is incremented when the active bit changes from high to low. To provide for recursions, the entry/exit counter is also incremented when the active bit was set in the previous cycle and the input address is equal to the lower limit.

In addition, the chip has the following signals which are global.

5. The input clock. The maximum clock frequency is 100 MHz.
6. Input data (`a[0:31]`) to be used for comparison and counting by the range recognizers. The input data is latched in at the the falling edge of the clock and can change at the clock frequency.
7. A `valid` bit for the chip indicating if the incoming data is valid or not. This bit can be used to maintain a count of particular types of addresses on the bus of the microprocessor; for example, instructions only or data only.
8. An `en_timer` bit for the chip. When this bit is set and the chip is in the `time_a_range` mode (see the description of the `chip_mode` bit below) and the active bit of the range recognizer is set, the address/time counter is incremented.
9. A `chip_mode` bit for the chip. When this bit is set (high), the chip is in the `time_a_range` mode. In this mode, the counter of each range recognizer is incremented if the `active` bit of that range recognizer is set and the `en_timer` bit is set. If the `chip_mode` bit is reset (low), the chip is in the `count_addr` mode. In this mode, the counter of each range

recognizer is incremented if the incoming data is valid (i.e. if the `valid` bit of the chip is set) and the incoming address lies between the upper and lower limits of the range recognizer.

10. Four input bits (`d[0:3]`) to indicate the range recognizer number. These bits are used to select a particular range recognizer when the upper and lower limits of a range recognizer are being programmed or when the address/time or the entry/exit count of a particular range recognizer is to be read out.

11. A `strb` signal to synchronize the writing of data into the range recognizer latches. After the signals `d[0:3]`, `limit`, `prg_chip` and `a[0:31]` are applied, they take some time to propagate to the range recognizer latches. The delays for each of these paths is unequal so it is necessary to synchronize them. The `strb` signal is used for this function. It is is applied when the signals have propagated through to their destinations. The write enable signal of the latch is raised when the strb signal is applied.

12. The `nr` signal. This signal is the master reset for the chip. It resets all the counters to zero.

The entry/exit or the address/time count can be read out after the chip has finished collecting data for a single program run. The following signals are provided on the chip for this function.

13. Eight bits (`c[0:7]`) which are used to output the value of address or entry/exit count of a range recognizers in pieces of eight bits i.e. in bytes.

14. Three bits (`enr[0:2]`) to select which byte of the counter is output. Valid values for the bits are 0 through 5 for the address/time count and 0 through 3 for the entry/exit count. The least significant byte of the counter (i.e. bits 0 through 7) is output when the value is 0.

15. A `ren` (read enable) signal. The counters can be read only when this signal is set (high).

16. A `clk4` signal. This signal outputs an internal clock of the chip which runs four times as slow as the input clock. Due to the internal structure of the chip, the signals to read the data of from the counters must be applied when the falling edge of this clock is detected. This signal is provided for the interfacing circuitry to allow it to properly synchronize the read signals that it applies to the chip.

17. An `eccnt` signal. When it is set (high), the value of the entry/exit counter is output and when it is reset (low), the value of the address/time counter is output.

17. Four bits (`m[0:3]`) which indicate the number of the range recognizer for which the data is to be read.

In addition to the above signals, the chip provides the following two additional signals used for testing the on-chip RAM.

18. The `wen` (write enable) signal. When this signal is high, the data input at the `a[0:31]` bits and the `c[0:7]` bits are used to write into the upper 40 bits of the address/time counter. (Note that the `c[0:7]` pins of the chip are bidirectional and are used as input pins when the `wen` signal is high). If the `eccnt` signal is low, the data is written into the upper 40 bits of the address counter and if the `eccnt` is high, the data (`c[0:23]`) is written into the upper 24 bits of the entry/exit counter. The lower 8 bits of both counters cannot be written into.

19. The `strb2` signal, which is used to synchronize the writing of data into the on-chip RAM. This signal is applied when the `wen` signal and the data have propagated through their paths. This signal is needed because of the

unequal delays in these paths.

Appendix A contains a behavioral simulation of the chip written in the C language. The next chapter discussed the microarchitecture of the chip.

4. MICROARCHITECTURE OF THE CHIP

4.1 Introduction

The microarchitecture of the chip is discussed in this chapter. First, the global architecture is discussed. This is followed by a description of the of blocks that make up the chip. Results from simulations are shown at the end of the chapter along with several plots of the relevant signals.

The software tools used for simulation and analysis are the tools used by the Advanced Development Group (ADG) at Tektronix Inc. The chip was designed using the ADG 1.5 μm standard cell library. The `fastsim` digital simulator was used for simulations, the `fastplot` plotting program was used to plot out the simulation results and the `fastta` timing analyzer was used to obtain critical path timing data. All the simulation data included in this thesis is in the form of plots. A description of each standard cell is included in Appendix B and the netlist is included in Appendix C.

The initial analysis of the chip was done using a value of 0.15 pF of wire capacitance for all the cells. This figure is used by the ADG group for most of their simulations. To account for the degradation of speed with temperature, a degradation of 0.3% per °C was assumed (as per ADG data). The chip was designed to work at 100 MHz at the junction temperature of 70°C. The `timescale` factor in the simulation control file is used for this purpose. All the timings of the cells were multiplied by this factor before being used by the simulator.

4.2 Global view of the microarchitecture

The chip may be broadly divided into the input section, the processing section and the output section. The processing section takes up most of the part of the chip and is the heart of it so it will be described first. The input and output sections are described later.

4.2.1 The Processing Section

Figure 1 shows the architecture of the processing section of the chip. The chip consists of 16 comparators each having programmable 32 bit latches for upper and lower limits and a 32 bit input port for data. The output of the comparator is fed to two eight bit counters. One of the counters maintains the entry/exit count while the other maintains the time/address count. The output carry of each counter is fed to a priority resolver which feeds a large on chip RAM. The RAM maintains the upper 40 bits for the time/address count of each range recognizer. An identical arrangement is used for the entry/exit count except that the RAM for this is only 24 bits wide (i.e. a 32 bit count is maintained for the entry/exit count as opposed to the 48 bit count for the time/address count).

Since the chip runs at 100 MHz, it must be able to accept an address for counting every 10 ns. It is impossible to do a 32 bit compare in 10 ns with the standard cell library available. My solution was to use pipelined comparators. The details of this arrangements are discussed later. For the following discussion, it is assumed that the comparators provide an increment signal for each of the counters every cycle i.e. every 10 ns. The problem, thus, is to accomplish a 48 bit increment in 10 ns. This is discussed below.

Three different kinds of architectures are possible to solve the problem. Either separate counters and incrementers for each range recognizer could be provided or a

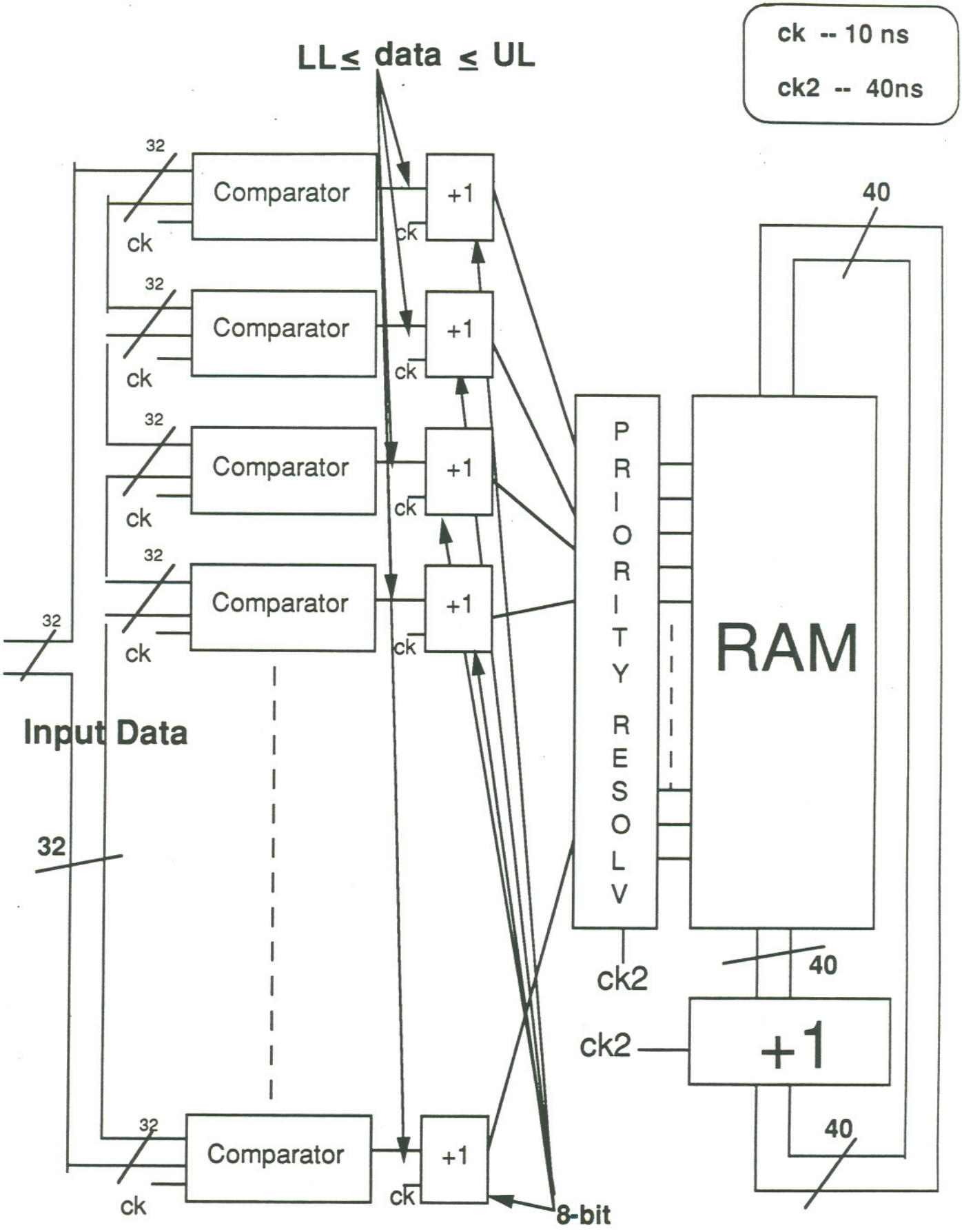

ck -- 10 ns
ck2 -- 40ns
LL≤ data ≤ UL
32
Comparator
ck
+1
40
PRIORITY RESOLV
RAM
Input Data
ck2
8-bit

Figure 1

fast single incrementer for the all or some of the range recognizers which is fast enough to do the increments for several range recognizers, can be provided or a combination of the two approaches can also be used. The merits and drawbacks of these arrangements are discussed in the following.

In the first approach, a separate counter is provide for each range recognizer. To accomplish a 48 bit increment at the rate of 10 ns, the incrementer has to be pipelined i.e. the output carry of each stage of computation is used as the input carry for the next stage and the counter can accept requests for incrementing from the comparators each cycle. Such an arrangement, however, has some disadvantages. The counter would be implemented using flip-flops for each bit. Flip-flops occupy large chip area. If a separate counter for each is provided, a significant area of the chip is taken up by the counters and thus the number of range recognizers that can be provided, is reduced. Since one of the goals of the design was to try to squeeze as many range recognizers on the chip as possible, this approach was rejected since it violated the goals of the design.

In the second approach, only one or more high speed incrementers are used for all the range recognizers and the count is stored in an on-chip RAM. The advantage of this approach is that an on-chip RAM occupies very little space as compared to a flip-flop. Another advantage is that, since in most cases, only a few of the range recognizers will be active and counting, only a few incrementers are needed for all the range recognizers (as opposed to one for each as in the previous approach). However, there are problems with this approach.

The access time for reading and writing for RAMs is typically large. For instance, the C533 standard cell of the 1.5 μm library (a 16 bit, single input output RAM) has an access time of 16.56 ns plus 2.34 ns for each pF of capacitance on its output. Clearly, this is unacceptable for the required speed of 10 ns. Even if a faster RAM could be designed to meet the design requirements, there is another problem

with this approach. It must be realized that in the worst case, a 48 bit increment is needed every 10 ns *for each range recognizer*. Keeping this in mind, consider the following sequence of events. Let the incrementer pipeline have 4 stages. Thus, it takes 40 ns to accomplish the 48 bit increment and write the incremented value back into memory. Let the initial count in the memory be 10. At time t = n, there is a request to increment the count and the count is read out and loaded into the incrementer. The incremented count will be available in the memory at n+40 ns. If, within this period, there is another request for an increment, the counter will read out the *old* count again i.e. 10 instead of the correct count which should be 11. So, the count will be incorrect. Since the range recognizers may provide a request for incrementing count every 10 ns, this approach is unacceptable.

In the third approach, a combination of the above two architectures is used to provide an optimal arrangement which captures the advantages of both. The problem with the RAM was that it received the increment requests too fast. If the requests can, somehow be slowed down, the RAM can be used. This can be done when it is realized that a counter is a divide by n circuit where n is the number of bits in the counter. The carry out at the n^{th} bit occurs 2^n times as slow as the carry input to the counter. So, if we have a 4 bit counter, the carry out at the 4th bit can occur at the maximum rate of 160 ns if the rate at which the input increment requests can occur is 10 ns. Now, the increment requests have slowed down sufficiently so that the second approach above can be implemented. In such an arrangement, each range recognizer will have its own n bit counter which stores the lower n bits of the 48 bit count. The carry out of this counter can be used to increment the upper (48-n) bits of the count which are stored in an on-chip RAM. The on-chip RAM can feed a single (48-n) bit incrementer which, after incrementing the data, writes it back to the RAM. This will be the only incrementer for all the range recognizers on the chip and thus significant chip area can be saved. The value of n must be chosen so that the correct timing and

optimization is achieved. This depends on the timings associated with the RAM, the number of range recognizers and whether overlapping ranges are allowed or not. The ADG standard cell library has a fast 4 bit incrementer (X137F) which, when used with a standard 4 bit flip-flop (X351W4), can read the data out of the flip-flop, increment it by one and store it back to the flip-flop within 10 ns. Assuming non-overlapping ranges and a 4 bit counter for each range recognizer and a total of 16 range recognizers on the chip, if there was a single incrementer for all the range recognizers for the upper 44 bits (with the count being stored in the RAM), the RAM circuit (the priority resolver, the RAM and the incrementer etc. see figure 1.) would have to finish a 44 bit increment in 10 ns in the worst case. As discussed above, the access time for the ADG standard cell RAM itself is higher than 10 ns so this solution is unworkable. As is shown later, a 40 ns clock period for the RAM circuit is optimal. (A 20 ns clock period could be used but it was found that a lot of additional logic was required to accomplish this; for instance a pipelined incrementer was required and more parallelism in the priority resolver was also necessary. The write back circuit to the RAM also became quite complicated). If every range recognizer had a 6 bit counter, the RAM circuit will be able accomplish the job of doing a 44 bit increment for 16 range recognizers in the worst case. However, the range would still have to non-overlapping. Since a 4 bit incrementer standard cell was readily available and since using an 8 bit counter for each range recognizer with a 40 ns clock circuit would give us the added feature of overlapping ranges, an 8 bit counter (one each for address/time count and entry/exit count) for each range recognizer was implemented. The upper 40 bits are stored in the RAM.

4.2.2 The Input Section

The input section consists of the following. (Please see Appendix C for the netlist).

1. All the input signals are input to the C920 standard cell (a pad) whose output is the input for the C501 TTL to CMOS level shifter. The `IPADT` cell (which is actually a macro) on the netlist is a combination of the pad and the level shifter. The output signals from the level shifter is then used as the input signal for the various cells in the chip.

2. The input data, `a[31:0]`, is fed to the pads whose outputs are fed to inverting TTL to CMOS level shifters (the `IPADI` macro which consists of the pad C920 and the buffer C507). The output of the level shifters (`pa[31:0]`) are fed to inverting buffers (cell S101X2). The output of the buffers (`b[31:0]`) are then fed to the range recognizers.

3. The clock buffer generates the clock signal which is routed to all the flip-flops in the chip. The input clock signal from the TTL to CMOS level shifter (`pck`) feeds four non-inverting buffer cells (Z101) connected in parallel. The output of these four cells (`ck1`) are wire-or'ed and then fed to 28 Z101 cells connected in parallel. The outputs of these cells are wire-or'ed to form the clock for the chip (`clk`) which is then routed to all the flip-flops in the chip.

4. Several flip-flops in the chip use an asynchronous reset signal. In order to synchronize the reset signal with the clock, the reset signal is fed to a 4 bit flip-flop such that the output of each bit is the input of the next bit. The final output is the reset signal (`nr.3`) properly synchronized with the clock which is then fed to a large non-inverting buffer (Z101X4) whose output (the `bnr.3` signal) serves as the reset signal for the entire chip. The four bit flip-flop is used to avoid the flip-flop from becoming metastable and thus providing an erroneous reset signal.

5. The `limit` signal is fed to the pad and the level shifter. The output of the level shifter (`plimit`) is fed to a non-inverting and an inverting buffer. The outputs of the two buffers (the `blimit` and the `bnlimit` signals) are fed to

each of the range recognizers. The strb signal, which is used to synchronize the writing of data into the latches of the range recognizers is also routed to the range recognizers after being passed through two Z101X4 buffers. The prg_chip signal, similarly, is fed to a pad whose output is fed to a level shifter. The output of the level shifter (pprg_chip) then feeds other parts of the chip.

6. The four input bits, d[3:0] which select a particular range recognizer are, like the other signals, fed to a pad whose output feed a level shifter. The output of the level shifters (pd[3:0]) is fed to a decoder tree. The decoder tree uses the X417 standard cell which is a two input four output decoder with an enable signal. The upper 2 bits, pd[3:2] feed one decoder cell. The pprg_chip signal is the enable signal for this decoder so that the decoder outputs are enabled only when prg_chip is high. The outputs of the decoder (decode[0:3]) are fed to the enable inputs of four other decoder cells. The two inputs of the four decoder cells are the lower two bits, pd[1:0]. Each of the four decoders generate four outputs which are then fed to the range recognizer cells (signals prg_lat[0:15] in the netlist).

7. Both the valid and the en_timer signals, are fed to pads whose outputs are fed to level shifters. The outputs of the level shifters (pen_timer and pvalid) are fed to the DLY4X delay cells. This is needed to equalize the delay between these signals and the b[31:0] signals i.e. the input data. The output of the delay cells (den_timer and dvalid) is then stored in flip-flops. The output of the flip-flops is fed to buffers which transmit the signals (bvalid and bnen_timer) to the range recognizers (Note that the en_timer signal is inverted before being distributed to the chip). The chip_mode signal is also trapped in a flip-flop and then fed into an inverting and a non-inverting buffer. The outputs of these buffers (bchip_mode and

bnchip_mode) are routed to the range recognizers.

4.2.3 The Output Section

The output sections consists of the following. (Please see Appendix C for the netlist).

1. The ren (read enable), wen, eccnt and strb2 signals are fed to input pads which feed TTL to CMOS level shifters. The eccnt signal, as previously explained, is used to read either the time/address or the entry/exit count. The output of the level shifter for this signal (peccnt) is fed to two large non-inverting buffers (Z101X4 cells) and their output is routed to other parts of the chip (the beccnt signal). The eccnt signal is also fed to an inverter (X101 cell) whose output (neccnt signal) is used to generate the read signal for the address/time count. The output of the level shifter for the ren signal (the pren signal) is ANDed with the beccnt signal to generate the ecenr signal which is the read signal for the entry/exit count RAM. Similarly, the neccnt and the pren signals are ANDed to generate the ccenr signal which is the read signal for the time/address count.

   The wen signal is the signal used to write into the RAMs. This signal is used only for testing the RAM. In the normal operation of the chip, all the locations of the RAM are initialized to zero. The wen signal is fed to the pads whose output is fed to the level shifter whose output is the pwen signal. The pwen signal is ANDed with the neccnt signal and the pprg_chip signal to generate the write signal for the address/time count RAM. Similarly, a signal for the entry/exit count RAM is also generated. The strb2 signal is used to synchronize the writing of data into the RAM. This is explained later.

2. The enr[0:2] bits decide which byte of the count is to be read. They are, as usual, fed to input pads and level shifters whose outputs (penr[0:2]) are fed

to three inverters and to AND gates to decode the three bits into six separate signals (`rd[0:5]`). The signal which is high indicates the number of the byte of the count to be read.

3. Just like the input section, the output section also has a decoding tree which is used to generate signals for which range recognizer count is to be read. The signals `m[0:3]` are decoded into `n[0:15]` which, after being fed to inverting buffers, are fed to a mux which drives the read port (or a mux which drives the write port) of the appropriate RAM. A separate decoding tree is provided in the output circuit because of the delay associated with the signals reaching the read port of the RAM and the structure of the RAM circuit. If the decoding tree of the input section were used, the delays involved are unacceptable so a separate decoding tree was needed. This will be elaborated upon later.

4. The output of the entry/exit count and the time/address count RAMs and counters are fed to a large MUX tree. The first stage of the MUX tree multiplexes the output of the two RAMS (`j[31:8]` and `k[31:8]`). The `beccnt` signal is the control signal for these muxes. The outputs of these muxes (`l[31:8]`) and the output of the counters of the range recognizers are then fed into 6 to 1 muxes. These muxes are used to send a particular byte of the count to the output. The `rd[5:0]` signals are used as the control signals for the muxes.

5. The final output of the 6 to 1 muxes (`l[7:0]`) is then fed to bidirectional pads. The bidirectional pads consist of one incoming and one outgoing tristate buffers connected to the pad input. The pad can, thus be used for input when the incoming buffer is enabled and it can be used as an output pad when the outgoing buffer is enabled. The outgoing buffer is enabled by the `nenr` signal i.e. when the count is to be read out. The `l[7:0]` signals are then output as `c[7:0]`. The incoming buffer is enabled by the `nenw` signal i.e. when data is to be writ-

ten into the RAM. The `wr[32:39]` bits are used to write to the upper 8 bits of the time/address count RAM. This is used only for testing the internal RAM.

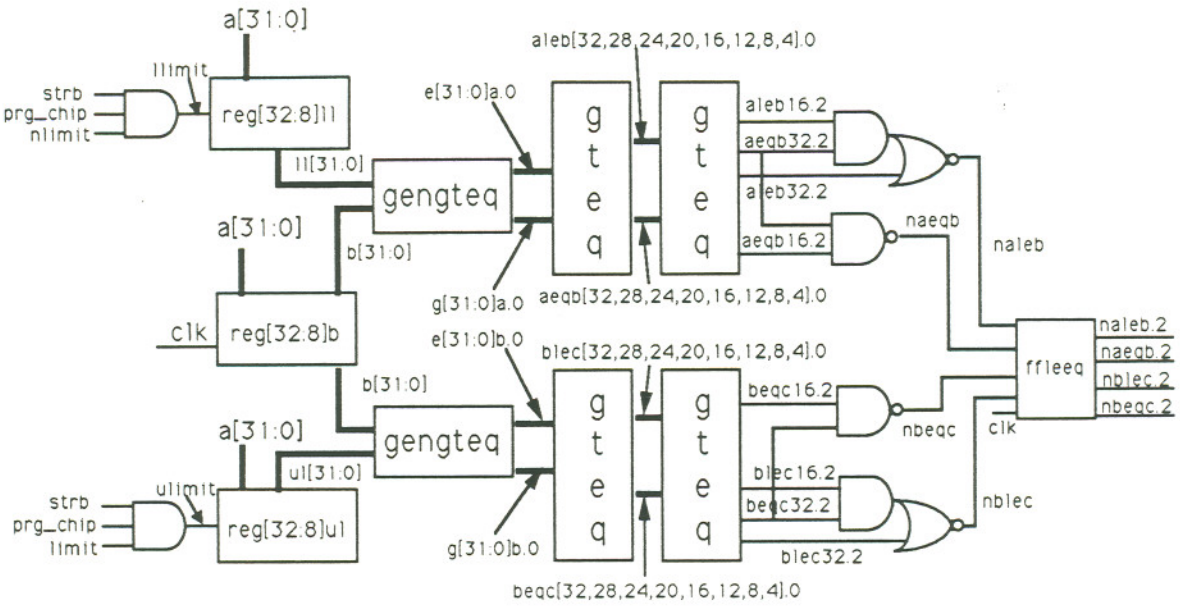
- Lastly, the internally generated 40ns clock, `clk4`, is driven out to the pins so that it can be used as a synchronization signal for applying the `enr[2:0]` signals. The signals must be applied within 20ns after the falling edge of `clk4` is detected or the output data will be incorrect.

The next few sections discuss the range recognizer circuit, the priority resolver, and the RAM circuit in detail.

4.3 The Range Recognizer Circuit

Figure 2 shows the components of the range recognizer in two parts. The netlist for the range recognizer is in Appendix C starting with the "model compare" statement. The range recognizer takes as its input the signals provided by the input section and outputs the carry bit for the two counters and the lower 8 bits for the counters.

The range recognizer uses several flip-flops with asynchronous resets. The master reset signal is first fed to a local buffer of the range recognizer and the output of the buffer is fed to the reset inputs of the various flipflops. As shown in the figure, the input data, `a[31:0]` is fed to two latches and a flipflop. The lower limit latch holds the 32 bits of the lower limit address, `ll[31:0]` and the upper limit latch holds the 32 bits for the upper limit address, `ul[31:0]`. The write enable signal for the lower limit latch is generated by ANDing the `prg_lat`, the `nlimit` and the `strb` signals. The `prg_lat` signal is connected to the output of the decoder tree of the input section and the other two signals are connected to their counterparts from the input section. Thus, the latches would be written into when the `prg_chip` signal is high, the range recognizer is chosen by the decoding tree, the


a[31:0]
llimit
strb
prg_chip
nlimit
reg[32:8]ll
ll[31:0]
gengteq
e[31:0]a.0
g[31:0]a.0
b[31:0]
a[31:0]
clk
reg[32:8]b
aleb[32,28,24,20,16,12,8,4].0
aeqb[32,28,24,20,16,12,8,4].0
aleb16.2
aeqb32.2
aleb32.2
aeqb16.2
naeqb
naleb
e[31:0]b.0
g[31:0]b.0
blec[32,28,24,20,16,12,8,4].0
beqc[32,28,24,20,16,12,8,4].0
beqc16.2
nbeqc
blec16.2
beqc32.2
blec32.2
nblec
ffleeq
naleb.2
naeqb.2
nblec.2
nbeqc.2
clk
b[31:0]
a[31:0]
ulimit
strb
prg_chip
limit
reg[32:8]ul
ul[31:0]
gengteq
g
t
e
q

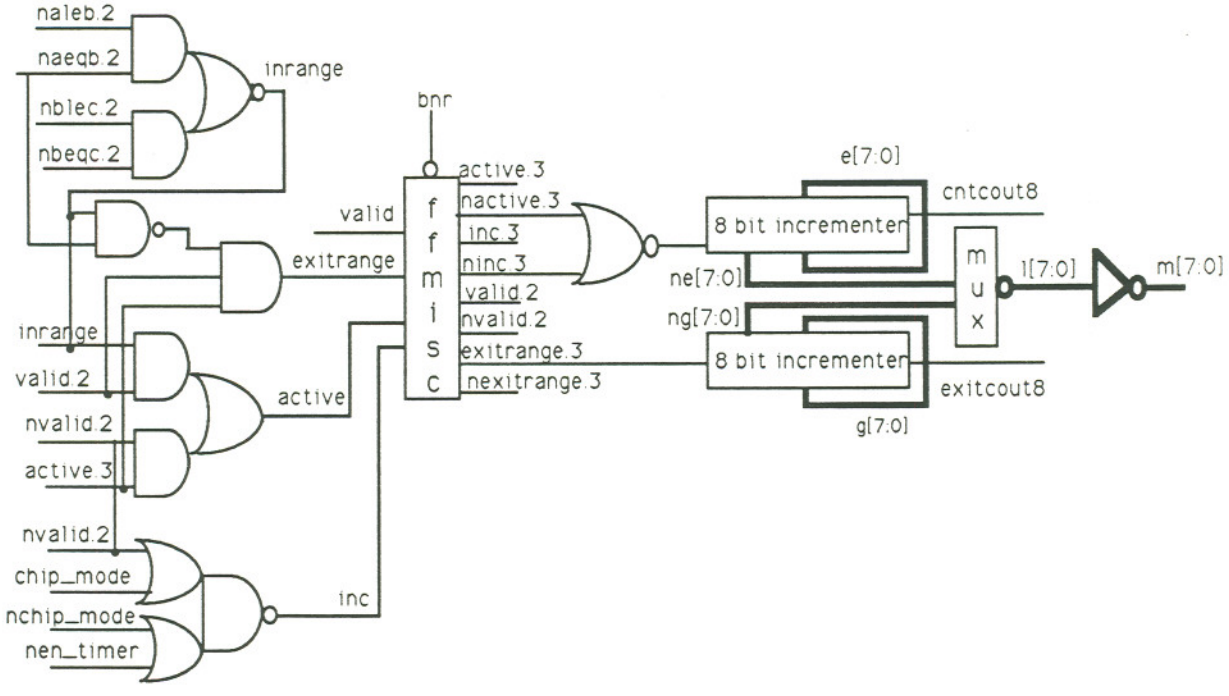

naleb.2
naeqb.2
nblec.2
nbeqc.2
inrange
bnr
valid
exitrange
inrange
valid.2
nvalid.2
active.3
nvalid.2
chip_mode
nchip_mode
nen_timer
active
inc
active.3
nactive.3
inc.3
ninc.3
valid.2
nvalid.2
exitrange.3
nexitrange.3
fmisc
e[7:0]
8 bit incrementer
cntcout8
ne[7:0]
ng[7:0]
8 bit incrementer
exitcout8
g[7:0]
mux
l[7:0]
m[7:0]

Figure 2

proper latch is chosen by the `limit` signal and the `strb` signal is raised. The delay for the `prg_lat`, `limit` and the `a(31:0]` are unequal so the `strb` signal is necessary to provide the right synchronization. The incoming data is also stored in the input four bit flipflops `reg[31:4]b`.

The range recognizer logic is designed to do a 32 bit compare. The input address, `a[31:0]` is compared with the lower and upper limit addresses and a signal for whether the input address is less than or equal to upper limit and greater than or equal to lower limit, is generated. Since a comparison must be done every 10 ns, the comparator is pipelined. The logic equation for comparison can be derived as follows. Given two numbers A and B, each n bits long, we generate two signals, e_i and g_i for each bit of both the numbers such that,

$$e_i = a_i\, b_i + \overline{a_i}\, \overline{b_i}$$
$$g_i = \overline{a_i}\, b_i$$

The e_i signal is true when the two bits, a_i and b_i are equal and the g_i signal is true when b_i is greater than a_i. To compare the numbers A and B, we start comparing from the highest bit. If the highest bit of B is greater than A, then we have the result immediately or if they are equal, we must compare the next lower bit. So, if the numbers are four bits long, the equation for $A < B$ is,

$$A < B = g_3 + e_3 g_2 + e_3 e_2 g_1 + e_3 e_2 e_1 g_0 \qquad (1)$$

This can be rewritten as

$$A < B = g_3 + e_3 g_2 + e_3 e_2 (g_1 + e_1 g_0)$$

Now, if we define

$$aleb_3 = g_3 + e_3 g_2 \qquad aeqb_3 = e_3 e_2$$

and

$$aleb_1 = g_1 + e_1 g_0$$

then

$$A<B = aleb_3 + aeqb_3 \cdot aleb_1$$

Note that this is the same equation as the equation for $A<B$ if A and B were 2 bits long, with e_1, g_1 and g_0 replaced by $aeqb_3$, $aleb_3$ and $aleb_1$ respectively. Thus, a 4 bit compare can be accomplished in two stages: in the first stage generate the *e* and *g* signals and use them to generate the *aleb* and *aeqb* signals which are the 2 bit equivalents of the 1 bit *e* and *g* signals. Then use the *aleb* and *aeqb* signals to generate the final signals for A<B in the second stage. This method can be extended to 32 bits and this is the method followed in the logic of the range recognizer.

As shown in figure 2, the input address and the lower and upper limits are fed to the `gengteq` blocks. These blocks generate the *e* and *g* signals for each bit. The *g* signal is generated by feeding the corresponding bits to an XNOR gate (X421) and the *e* signal is generated by feeding the complements of the corresponding bits to a NOR gate (X102). The *e* and *g* signals from the `gengteq` blocks are then fed to the `gteq` block. This block consists of several `gtgen` cells which do a four bit compare with the corresponding *e* and *g* inputs (see equation (1) above). The output of these cells are the *aleb* signals. In addition, the *e* signals are fed to a four input AND gate (X404) whose output is the *aeqb* signal. The various *aleb* and *aeqb* signals are trapped in flip-flops. This is, thus, the first stage of the pipeline for comparison of the input address and the addresses in the latches. The output of the first stage is then fed to another `gteq` block which generates four signals, `aleb16.2`, `aeqb16.2`, `aleb32.2`, `aeqb32.2` as its output (similar signals are generated for the comparator for the input address and upper limit latch). The equation for $A<B$ when A and B are 32 bits long, thus, is,

$$A<B = aleb\,32.2 + (aeqb\,32.2) \cdot (aleb\,16.2)$$

The *naleb* signals is the $\overline{A<B}$ signal and the *naeqb* is the $\overline{A=B}$ signal. Similarly, the *blec* and *beqc* signals are generated from the comparator comparing the upper limit address and the input address. These signals are trapped in a flip-flop. This

completes the second and final stage of the pipeline for the comparison.

The bottom part of figure 2 shows the logic which uses the signals provided by the comparators to generate the entry/exit and the time/address count. The `inrange` signal is true if the input address lies between the upper and the lower limit i.e. if $ll[31:0] \le a[31:0] \le ul[31:0]$. The logic equation for this is,

$$inrange = (aleb + aeqb) \cdot (blec + beqc)$$

Using DeMorgan's law, this can be rewritten as

$$inrange = \overline{\overline{aleb} \cdot \overline{aeqb} + \overline{blec} \cdot \overline{beqc}}$$

(In the circuit, the complement outputs of the `ffleq` flipflops are used to generate the `inrange` signal. Figure 2 shows these signals.) Along with the `inrange` signal, the `active` signal for each range is also generated. Recall that the `active` signal indicates if the current range is active and counting or not. The `active` signal must be updated every cycle. The `active` signal should be true if the input address is valid and in the range or if the current input address is invalid and current range was active in the last cycle. The logic equation for the *active* signal at time *t* thus becomes,

$$active_t = active_{t-1} \cdot \overline{valid} + valid \cdot inrange$$

The $active_{t-1}$ signal is the `active.3` signal on the diagram, the $active_t$ is the `active` signal the *valid* signal and its complement are `valid.2` and `nvalid.2` respectively. The signal for whether the current input address signifies an exit from the range, `exitrange`, is true when the current input address is valid and not in the range and the range recognizer was active in the last cycle. In addition, to account for recursive procedures, the `exitrange` signal is asserted when the input address is valid and is equal to the lower limit and the range recognizer was active in the previous cycle. Thus the logic equation for the `exitrange` signal is,

$$exitrange = valid \cdot active_{n-1} \cdot (\overline{inrange} + aeqb)$$

Using DeMorgan's law,

$$exitrange = valid \cdot active_{n-1} \cdot \overline{(inrange \cdot \overline{aeqb})}$$

The $\overline{aeqb}$ is the `naeqb.2` signal in the figure. The `exitrange` signal is used to increment the entry/exit counter.

The increment signal for the address/time counter is the `inc` signal in the figure. As mentioned previously, the count is incremented when either of the following conditions are met: the chip is in count-address mode and the input address is valid and in the range or the chip is in time-a-range mode, the `en_timer` signal is high and the range is active and counting. As shown in the figure, a part of the increment signal is generated in parallel with the active signal. The logic equation for the `inc` signal is,

$$inc = valid \cdot \overline{chip\_mode} + en\_timer \cdot chip\_mode$$

Using DeMorgan's law,

$$inc = \overline{(\overline{valid} + chip\_mode)(\overline{en\_timer} + \overline{chip\_mode})}$$

This is shown in the figure. The `inc` signal is trapped in the `ffmisc` flip-flop. Various other signals are also trapped in the flip-flop as shown in the figure. It should also be noted that this flip-flop has a reset signal, `bnr`. At initialization time, this signal is asserted and all the bits of th flip-flop are set to zero when the signal is asserted. The complements of the `inc.3` and `active.3` signals from the flip-flop are fed to a NOR gate whose output is fed to an eight bit incrementer. Similarly, the `exitrange.3` signal is the increment signal which is fed to an identical eight bit incrementer.

The eight bit incrementer consists of two four bit flip-flops and two four bit incrementers. A single four bit increment is done in each cycle i.e. each 10 ns. The output of the flipflop feeds the incrementer whose output is fed back to the flip-flop inputs. The carry-out from the lower 4 bits is trapped in a flip-flop and then fed to

the next stage. The output carry of each incrementer is fed to the priority resolver logic (discussed later). The output bits of the flip-flops are fed to 2 to 1 inverting muxes whose control signal is the `eccnt` signal. These muxes controls which count is sent to the pins when the count is to be read out. The output of the muxes feed a tristate buffer which is enabled when the read signal for this particular range recognizer is asserted. The tristate buffer feeds an eight bit bus which is common to all the 16 range recognizers on the chip. The appropriate buffer drives the bus when it is enabled. The tristate buffers reduce the number of wires and thus minimize wiring capacitances.

Thus, in summary, the range recognizer blocks take a 32 bit address as input and output two eight bit counts and two carryout signals, `cntcout8` and `exitcout8`. The next section discusses the priority resolver circuit.

4.4 The Priority Resolver Circuit

As explained previously, the specifications of the chip allow overlapping ranges to be programmed. Thus more than one range recognizer may output a carry at the same time. The output carry is a signal for the RAM circuit to increment the upper bits of the count for the range recognizer. Since there is only one incrementer in the RAM circuit, only one of the increment requests can be handled at a time by the RAM circuit. Thus some mechanism to ensure that the requests are queued up and sent one after the other to the RAM circuit, is required. The priority resolver circuit performs this function. The RAM circuit takes 40 ns to complete one increment. The priority resolver takes as its input, the carry out signals from the range recognizer and stores them till the RAM circuit is free to do an increment. When the RAM circuit proceeds with the increment for a particular range recognizer, the priority resolver clears the corresponding carry out bit from its own flip-flops. This bit can be set again by another carry out signal from the respective range recognizer. Since the

carryout from each range recognizer can only occur at the rate of once in 256 cycles (2560 ns), and since there are only 16 range recognizers on the chip, with the RAM circuit cycle time of 40 ns, the priority resolver will always take care of all the increment signals correctly and never miss any of them even in the worst case of all the 16 range recognizers giving an increment signal at the same time.

The priority resolver implements a simple algorithm for deciding which range recognizer's request is honored first. The request from a range recognizer whose number is lower is always honored first i.e. the request from range recognizer 0 has top priority followed by range recognizer 1 etc. and range recognizer number 15 has the last priority. There is no specific reason to implement this particular order but it is the simplest to implement (instead of some scheme like first in first out) so it was chosen. The objective is to prevent an overlap of increment requests so the particular priority resolving algorithm is not of importance so far as it accomplishes the objective. Figure 3 shows a part of the priority resolver circuit.

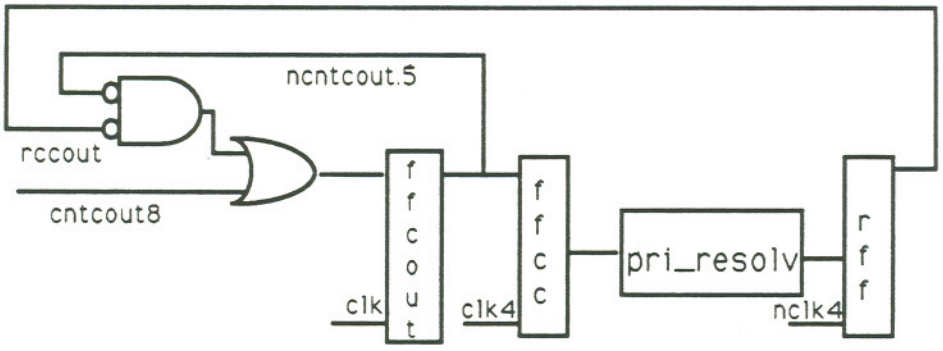


Figure 3

The carryout signal from the incrementers is fed to an OR gate and the output of the OR gate is trapped in a flipflop. The other input of the OR gate is fed by a NOR gate (which is used in place of an AND gate since inverted inputs are available) whose inputs are the output carry signal from the flipflop (`ncntcout.5`) and the output signal of the priority resolver (trapped in another flipflop). The reason why

this logic is needed is the following. The output carry from the incrementer of the range recognizer gets updated every 10 ns. However, the priority resolver will not look at this signal every cycle. So the output carry needs to be saved till the priority resolver has had a chance to look at it and finally apply it to the RAM after which it can be cleared. The `pri_resolv` circuit will assert at its output the carry out signal for a particular range recognizer only when that range recognizer has the highest priority at that instant. When this is done, the priority resolver has now passed on the increment request to the RAM circuit and thus it must clear the carry out bit. Now, consider the logic equation,

$$cntcout.5_t = cntcout8 + cntcout.5_{t-1} \cdot \overline{rccout}$$

The `rccout` signal is the signal to be applied to the read ports of the RAM. Note that the `rccout` is normally low and is only asserted when the `pri_resolv` block forwards the carry request of the range recognizer to the RAM. So, the $\overline{rccout}$ signal is normally high and thus the output of the AND gate is the same as $cntcout.5_{t-1}$. This is ORed with the current carry out from the range recognizer i.e. `cncout8`. So, if either of these signals is high, it gets trapped in the flip-flop and the output of the flip-flop remains high till the `rccout` signal is asserted i.e. when the $\overline{rccout}$ signal goes low. Now, the output of the flip-flop is cleared and thus the carry out is also cleared till the next carryout signal (i.e. `cntcout8` signal) occurs again.

Note that the `cntcout.5` signal is the output of a flip-flop whose clock is `clk` which runs at 10 ns. This signal is fed to another flip-flop which traps the signal with the `clk4` signal which is the internally generated 40 ns clock. This flip-flop thus provides the proper interface between the 10 ns and 40 ns circuits. The output of the priority resolver is trapped with the complement of the 40 ns clock which implies that the propagation delay through the `pri_resolv` block is less than 20 ns. The logic equation for the `pri_resolv` logic is,

$$rcout_n = cout_n \cdot \overline{cout_{n-1}} \cdot \overline{cout_{n-2}} \ldots\ldots\ldots\ldots\ldots \overline{cout_0}$$

where $rcout_n$ is the output of the `pri_resolv` block and $cout_n$ etc. are the inputs to it.

4.5 The RAM circuit

The RAM circuit is responsible for maintaining the upper bits of the count. The priority resolver and RAM circuits for the entry/exit and address/time count are identical except that the RAM circuit for the former has only 24 bits of upper count as opposed to the 48 bits for the latter. So only the address/time count RAM circuit will be discussed here. The discussion applied mutatis mutandis to the entry/exit count RAM circuit. All the flip-flops in the RAM circuit are clocked with the internally generated 40 ns clock. This clock is generated by feeding the 10 ns clock to a divide by 4 circuit. The output of the circuit is a 40 ns clock and its complement. This output is fed to buffers whose outputs (`clk4` and `nclk4`) are routed to the various flip-flops in the RAM circuit.

Figure 4 shows a block diagram of the RAM circuit. The increment request from the priority resolver output flip-flp is fed to a mux whose output feeds a buffer which, in turn, drives the read ports of the RAM. (The output of the decoder tree in the output section feeds the other input of the mux. This input is used to read out the contents of the RAM). Only one of the 16 outputs from the priority resolver would be asserted at a time which implies that one location of the RAM will be read out. It is important to understand the clocking sequence of the flip-flops in the RAM circuit properly. All the flip-flops are falling edge flip-flops unless specified otherwise.

As explained previously, the input flip-flop for the priority resolver uses the 40 ns clock. The delay through the priority resolver is about 16 ns. The output flip-flop of the priority resolver is clocked with the complement of the 40 ns clock. This means that the output of the priority resolver gets trapped 20 ns after the falling edge

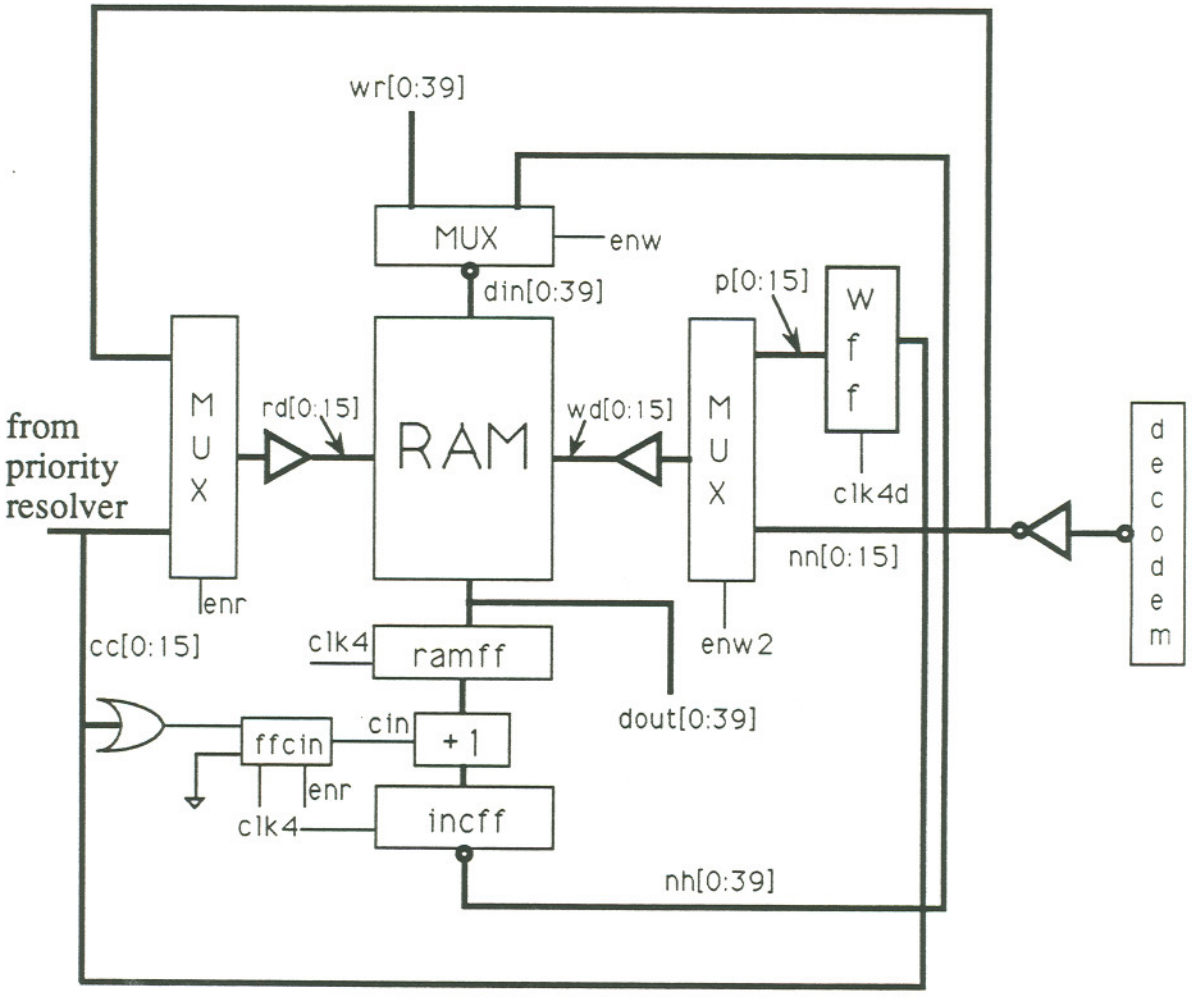

wr[0:39]
MUX
enw
din[0:39]
p[0:15]
W
f
f
clk4d
from
priority
resolver
M
U
X
rd[0:15]
RAM
wd[0:15]
M
U
X
nn[0:15]
d
e
c
o
d
e
m
enr
cc[0:15]
clk4
ramff
enw2
dout[0:39]
cin
+1
ffcin
enr
clk4
incff
nh[0:39]

Figure 4

of the 40 ns clock, `clk4`. This output will be then driven to the read port of the RAM if the `enr` signal is low i.e. if the chip is counting currently. The RAM is actually made up of several C533F RAM cells which are single bit I/O, 16 location fast RAM cells. Each cell has 16 read and write ports and one input and one output. This means that the mux at the read port of the RAM must drive 40 such cells. The output of the RAM is trapped in a flip-flop which is clocked with `clk4`. The worst case read time for the RAM cell is 10.20 ns plus 2.34 ns for each pF of load on its output. This means that the output of the RAM will be valid at about 12 ns after the read address to the RAM read port is applied (assuming a load of about 0.5 pF). Given the 20ns delay through the priority resolver and a delay of about 3 ns for the flip-flop outputs to become stable, the mux must drive the data from the flipflops to the RAM read ports in 5 ns or less. Given that the mux has to drive 40 individual ports, this is not possible to accomplish (with the ADG cell library) without using buffers. Therefore, buffers are used to drive the data to the RAM read ports.

The output of the RAM is trapped in a flip-flop (`ramff`) which is clocked with `clk4`. At the same time, the carry-in for the 40 bit incrementer is trapped into a mux flip-flop. The carry-in is obtained by feeding the outputs of the priority resolver to a 16 input OR gate (constructed using NAND and NOR gates). The output of the OR gate is trapped into a mux flip-flop. This flip-flop traps the output of the OR gate if the `enr` signal is low. The other input of the flip-flop is grounded so the flip-flop traps a zero if the `enr` signal is high. Thus the count will not be incremented when the data is being read out of the RAM to feed it to the output pins. The output of `ramff` feeds a 40 bit incrementer. The output of the incrementer is trapped in the `incff` flip-flops. These flip-flops are also clocked with `clk4` so the increment must be finished in 40 ns. The output of these flip-flops is the updated count which is then fed to a mux which feeds the data input port of the RAM.

As seen from the figure, a mux also feeds the write ports of the RAM. This mux

is fed either from a flip-flop or from the output of the decoder tree in the output section. The decoder tree output is used when the `enw2` signal is asserted. This mode is used for testing the RAM. The output of the mux is again driven using buffers to the read ports of the RAM. As shown, the output of the priority resolver is fed to the `wff` flip-flop. However, the clock for this flip-flop is `clk4d`. This is a signal generated by feeding the `clk4` signal to some delay cells. The delay is approximately 8 ns. The result is that the output of the priority resolver is stored into this flip-flop about 8 ns after it is stored in the flip-flop at the read port of the RAM.

In order to understand the operation of the RAM circuit, consider a sequence of events as follows. At time **t**, the carry out signals from the range recognizers are trapped at the input of the priority resolver with the `clk4` clock. At **t+20** ns, the priority resolver output is trapped in the flip-flop at the read port (the priority resolver output flip-flop). The mux then applies this value to the RAM read ports. At **t+48** ns, the address applied to the read port of the RAM (which is the same as the output of the priority resolver) is stored in the `wff` flip-flop. The next clock edge for the read flip-flop will occur at 40 ns while the next clock edge for the write flip-flop will occur at **t+88** ns. The RAM outputs the appropriate data after **t+40** ns which is trapped in the `ramff` flip-flop which feeds the incrementer. The output of the incrementer is trapped at `t+80` ns and is fed back to the input of the RAM. Since the data at the write ports will change only at **t+88ns**, the output of the incrementer is written into the correct location (i.e. where it came from before being incremented). The same sequence of events repeats every cycle.

It is now easy to see why the inputs to the read and write port of the RAM are critical. Note that one of the write ports of the RAM is addressed all the time. This means that whatever appears at the data input port will be written into that location. So, the inputs to the ports must be timed very precisely or else the data in the RAM will be wrong. This is also the reason why a separate decoder tree for the output sec-

tion was necessary. If the input decoder tree is used, its inputs get excessively loaded resulting in an increase in the delay through it. This delay is unacceptable for the RAM circuit so a separate decoder tree is used. This is not much of a problem, though, since the decoder tree takes very little space on the chip. When the data from the RAM is to be read out, the `enr` signal is asserted and the output of the decoder tree is fed to the read port of the RAM. The RAM output is trapped in the `ramff` flip-flop whose output is then sent out to a mux whose output feeds the pins. The read address at the input pins of the chip must be maintained till the data is read out. It is clear now that the read address must be applied in such a way so that the output of the decoder tree is applied to the read ports, about 12 ns before the falling edge of the `clk4` clock occurs. The graphs of simulation results attached at the end of this chapter show this. The `clk4` signal is output at the pins so that it can be used to synchronize the application of the read address to the pins.

To test the RAM, an additional pin is provided on the chip as discussed previously. When this pin is asserted along with the `prg_chip` pin, the `enw` signal is asserted for the appropriate RAM circuit (entry/exit or address/time) based on the `beccnt` signal. This will feed the `wr[0:39]` signals to the RAM data input. As discussed previously, the 32 bits of input data pins (the lower 24 bits for the entry/exit count RAM) and the 8 bits of bidirectional output/input data pins (not used for the entry/exit count RAM) are used to feed the RAM. The output of the decoder tree is the write address. It will be fed to the RAM when the `enw2` signal is asserted. This signal is asserted when, in addition to `prg_chip` and `enw`, the `strb2` pin is asserted. The `strb2` pin is used to synchronize the writing of data into the RAM since the data takes longer to set up than the address.

The next section discusses the layout of the chip and shows the results of a typical run of the chip. Several plots showing the key signals are shown.

4.6 Results and discussion

As mentioned previously, all the simulations of the chip before layout were done using the 1.5 μm standard cell library. The chip layout was done using automatic layout tools and the capacitances were recalculated from the layout. The critical path was 14.6 ns, 4.6 ns slower than the required. The critical path was found in one of the range recognizers (this was expected). It must be noted, however, that only a few of the 16 range recognizers had critical paths while others met the timings. It must also be noted that almost no effort was put into planning the layout of the chip. Since the chip consists of 16 range recognizers and since even with the automatic layout, many of the range recognizers meet the required timing, it is reasonable to assume that if a custom layout of one range recognizer is done, it would not only meet the required timings but also save significant amount of chip area.

The current layout of the chip uses the 1.5 μm standard cell library. The ADG group has already developed a standard cell library for a channel length of 1 μm. If the current layout is scaled to 1 μm channel length, according to ADG, the gain in performance would eliminate the critical paths that are present in the 1.5 μm version of the layout and the chip would meet the required specs (100 MHz clock speed at 70^{o} C). The chip has not yet been actually manufactured and therefore no real data for actual silicon is available. However, as discussed before, there are reasonable grounds to assume that there will be no critical paths in a manufatured chip with some effort being invested in layout of the chip. In the following, the results of simulation of a typical run of the chip are presented. The input vectors were generated using a vector generating program used by the ADG group called `vgp`. The actual list of vectors is too long to include here but the `vgp` program input used to generate the vectors is included in appendix D. All the simulations shown in this thesis and the timings shown in this section are scaled up by 32% from the original timings to account for the speed up that can be gained by scaling the chip from 1.5 μm to 1 μm technology

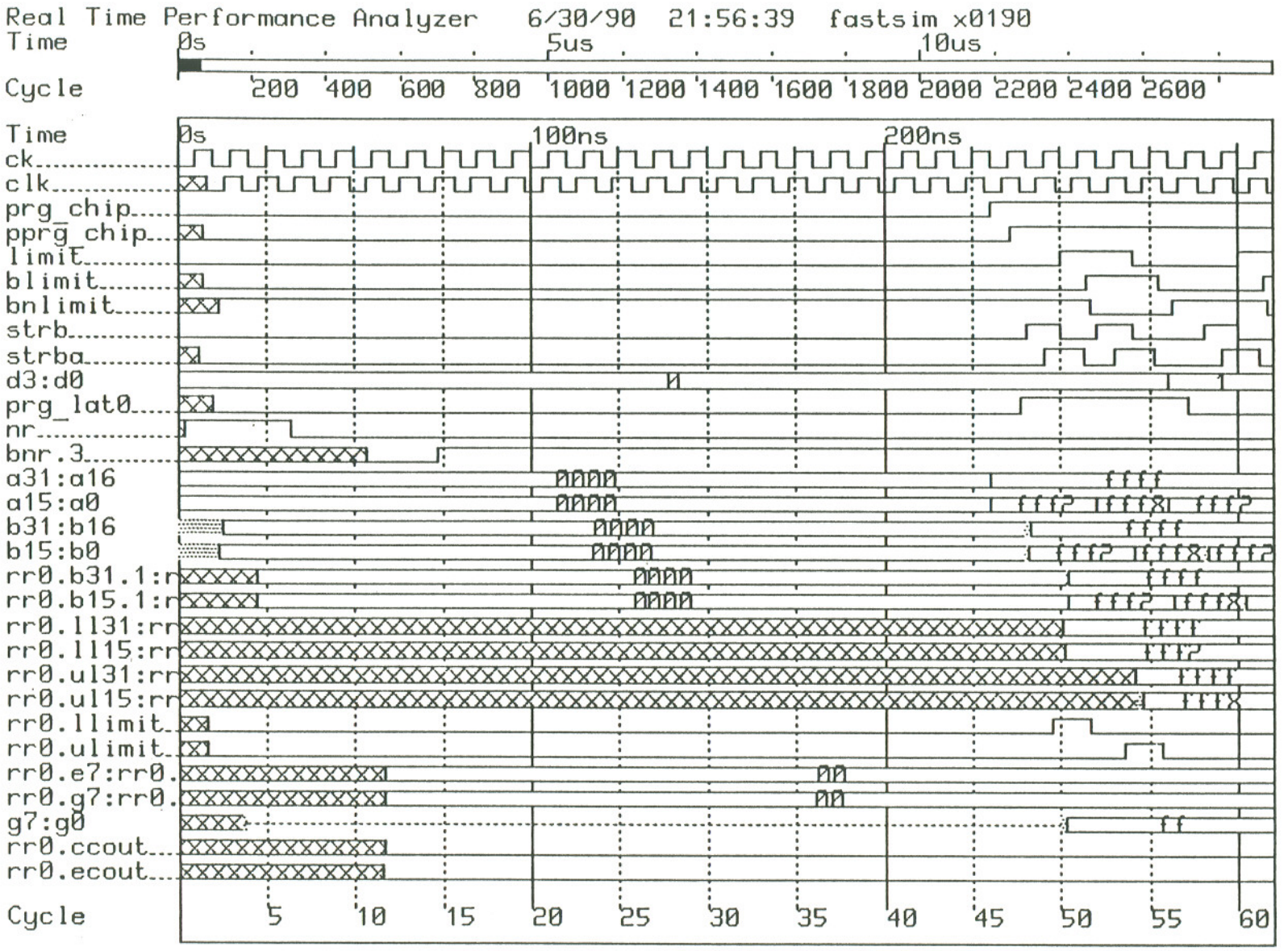
and better layout (i.e. all the time factors were multiplied by 0.68 after accounting for the temperature degradation of 0.3% per ^{o}C). The results of the simulations are shown in the form of plots which are located at the end of this chapter.

The plots show values of some signals for a typical run of the chip. The chip is first initialized by applying the chip reset signal, `nr` (see plot 1). The part of the plots which shows a series of x's indicates that the value of the signal is undefined at that time. This is expected at the time of initialization. The reset signal `nr` is fed to buffers whose output, `bnr.3` (asserted low) resets the counters for the range recognizers (signals `rr0.e[7:0]` and `rr0.g[7:0]`. The carryout signals also get initialized. These, in turn, feed the flipflops which feed the RAM circuit (see plot 4). Thus the RAM flip-flops also get initialized to zero. The internal 40 ns clock generated from the 10 ns clock is also seen in plot 4.

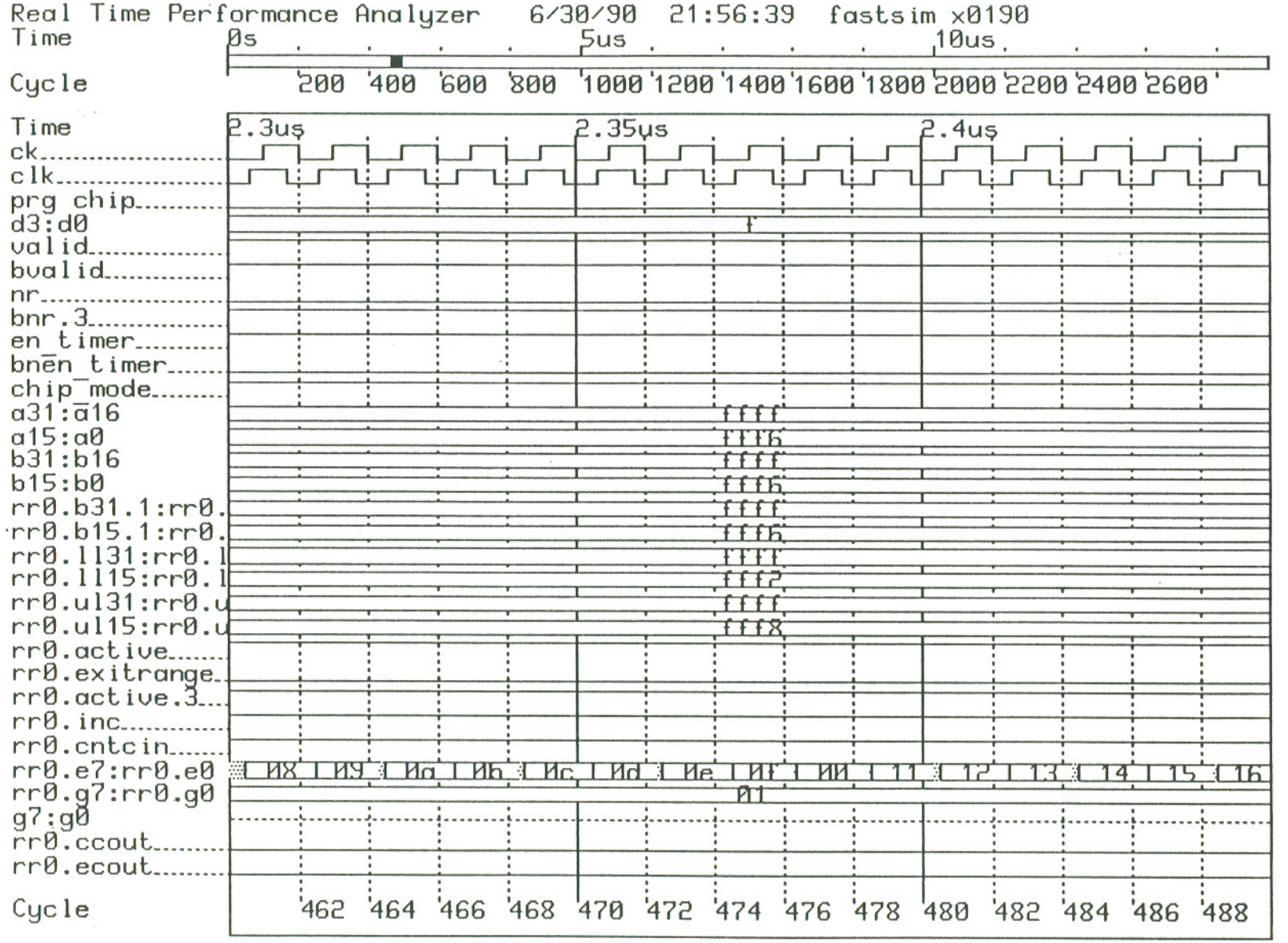
Referring back to plot 1, at about 220 ns, the `prg_chip` signal is raised and data is applied at the input. This data is used to program the upper and lower limits of the range recognizers. The lower limit of all range recognizers is programmed to the value FFFFFFF2 (hex) and the upper limit is programmed to FFFFFFF8 (hex). The data is latched into the appropriate latches (signals `rr0.ll[31:0]` and `rr0.ul[31:0]` on the plot) using the `limit` signal and is strobed in using the `strb` signal. The `d[3:0]` signals select the appropriate range recognizer. The plots attached show the data for range recognizer number 0. The write enable signals for the latches, `rr0.llimit` and `rr0.ulimit` can also be seen on plot 1. The `g[7:0]` signal is the inverted output of the mux in each range recognizer which sends lowest eight bits of the time/address count (`rr0.g[7:0]`) or the entry/exit count (`rr0.e[7:0]`) to another mux which sends it to the output pins.

Now, referring to plot 2, we see the range recognizer 0 counting. The input data is held at the value FFFFFFF6 and the `valid` signal is held high. The count increments every cycle, as seen in the plot. Plot 3 shows both the entry/exit and the

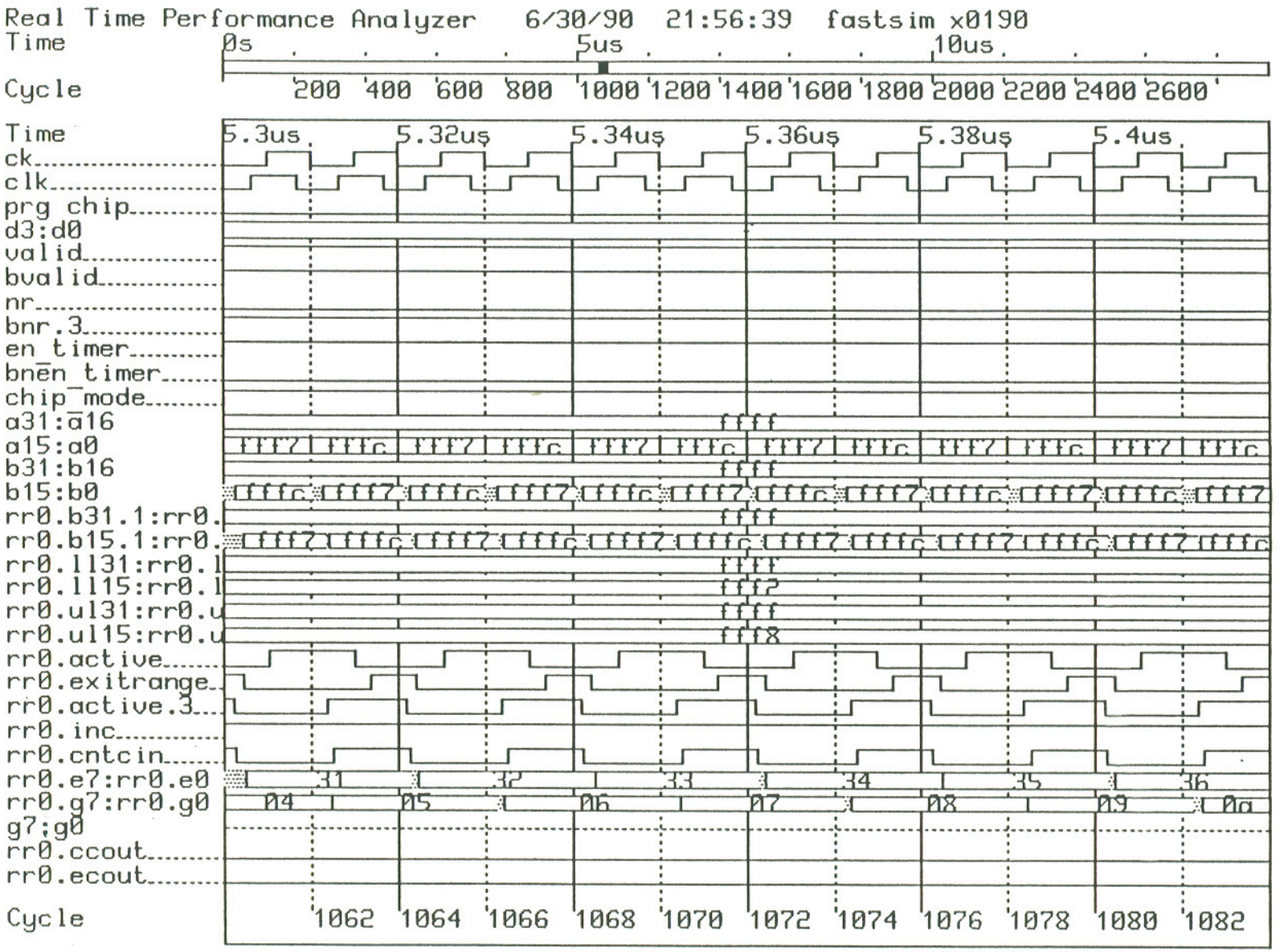
address/data counters counting. The input data alternates between the values FFFFFFF7 and FFFFFFFC. The first address is within the range and the second is outside the range so both counters will be counting. As mentioned previously, plot 4 shows the flip-flops in the RAM circuit being initialized. Plot 5 shows data being written into the RAM. This is the test mode of the RAM. In order to see the 40 bit increment, we write a value of FFFFFFFFFF into the 40 bit address/time count RAM and a value of FFFFFF into the 24 bit entry/exit count RAM (not shown). The plot also shows the address being applied to the `m[3:0]` pins and the output of the decoder tree being applied to the RAM write port, (signal `ramccnt.w[15:0]`). Note also that the output of the decoder tree is applied to the RAM only after the `strb2` signal is raised and held. Plot 6 shows the RAM circuit counting. Recall that all the range recognizers were programmed with same upper and lower limits to see that even overlapping ranges work correctly. This is seen from the plot. Notice that the `cc[15:0]` signals start with the value of FFFF i.e. all range recognizers requesting an increment at the same time. The output of the priority resolver, `rcc[15:0]` applies each one at a time to the RAM read ports, `ramccnt.rd[15:0]`. The output of the RAM is trapped in the output flip-flop (`ramccnt.dout[39:0]`) and stored back to the RAM after being incremented (`ramccnt.din[15:0]`). The address at the read port is trapped into flip-flops at the write port after being delayed (`ramccnt.w[15:0]`). The delay is achieved by using the delayed 40 ns clock, `ramccnt.clk4d` to clock these flip-flops. Finally, plot 7 shows the results being read out from the RAM and the range recognizers. The input range recognizer number `m[3:0]` (which is actually the RAM address) is applied to the read port of the RAM and the output data is trapped in flip-flops (`ramccnt.dout[39:0]`). The output of the flip-flops feed a mux which drives a byte of data to the output pins, `c[7:0]`, depending on the value of the `enr[2:0]` signals. (The plot shows the value decoded to six signals, `rd[5:0]`).


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
0s
100ns
200ns
ck
clk
prg_chip
pprg_chip
limit
blimit
bnlimit
strb
strba
d3:d0
prg_lat0
nr
bnr.3
a31:a16
a15:a0
b31:b16
b15:b0
rr0.b31.1:r
rr0.b15.1:r
rr0.l131:rr
rr0.l115:rr
rr0.ul31:rr
rr0.ul15:rr
rr0.llimit
rr0.ulimit
rr0.e7:rr0.
rr0.g7:rr0.
g7:g0
rr0.ccout
rr0.ecout
Cycle
5 10 15 20 25 30 35 40 45 50 55 60

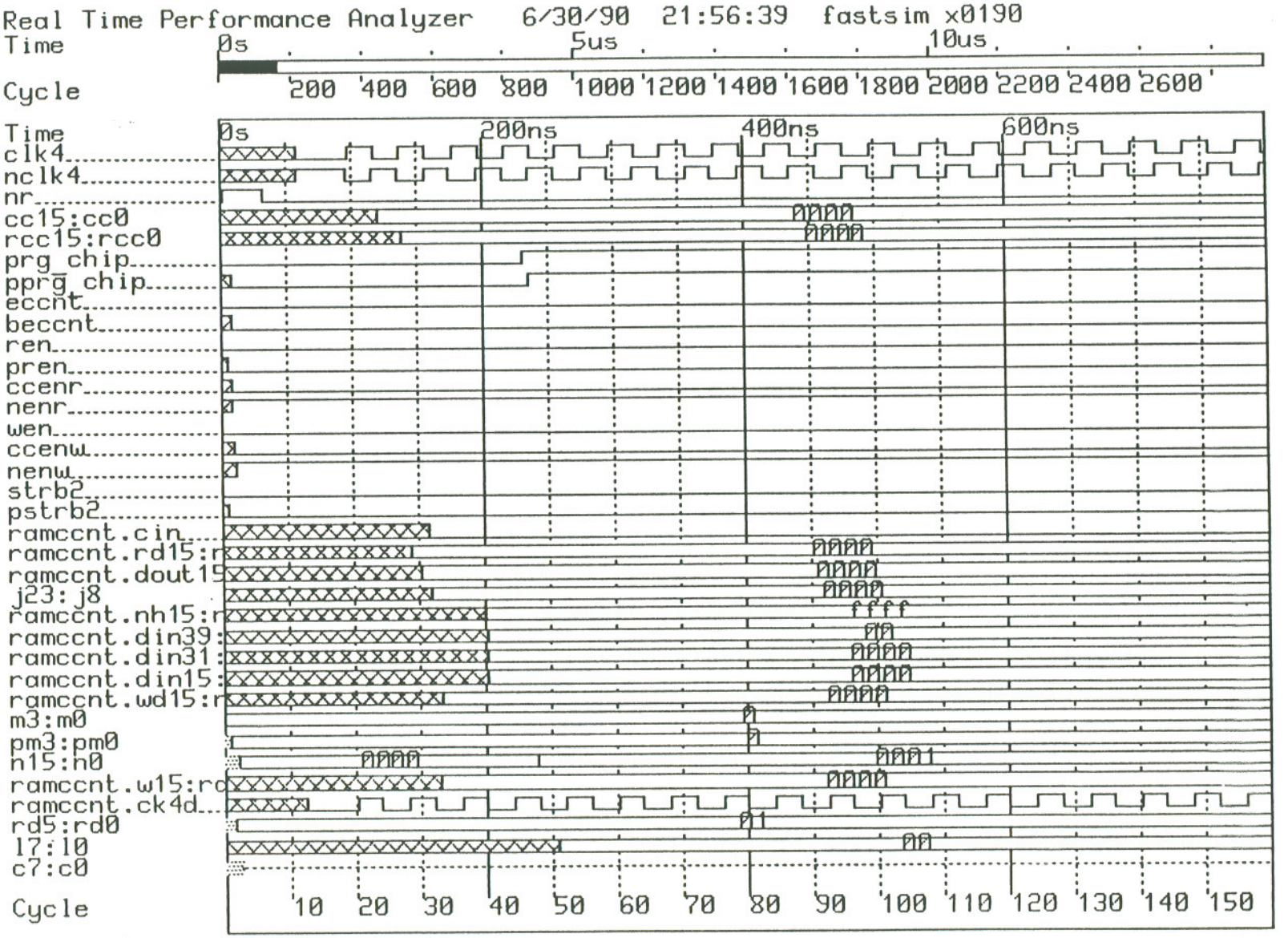
Plot 1


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
2.3us
2.35us
2.4us
ck
clk
prg chip
d3:d0
valid
bvalid
nr
bnr.3
en_timer
bnen_timer
chip_mode
a31:a16
a15:a0
b31:b16
b15:b0
rr0.b31.1:rr0.
rr0.b15.1:rr0.
rr0.ll31:rr0.l
rr0.ll15:rr0.l
rr0.ul31:rr0.u
rr0.ul15:rr0.u
rr0.active
rr0.exitrange
rr0.active.3
rr0.inc
rr0.cntcin
rr0.e7:rr0.e0
rr0.g7:rr0.g0
g7:g0
rr0.ccout
rr0.ecout
08 09 0a 0b 0c 0d 0e 0f 00 11 12 13 14 15 16
01
Cycle
462 464 466 468 470 472 474 476 478 480 482 484 486 488

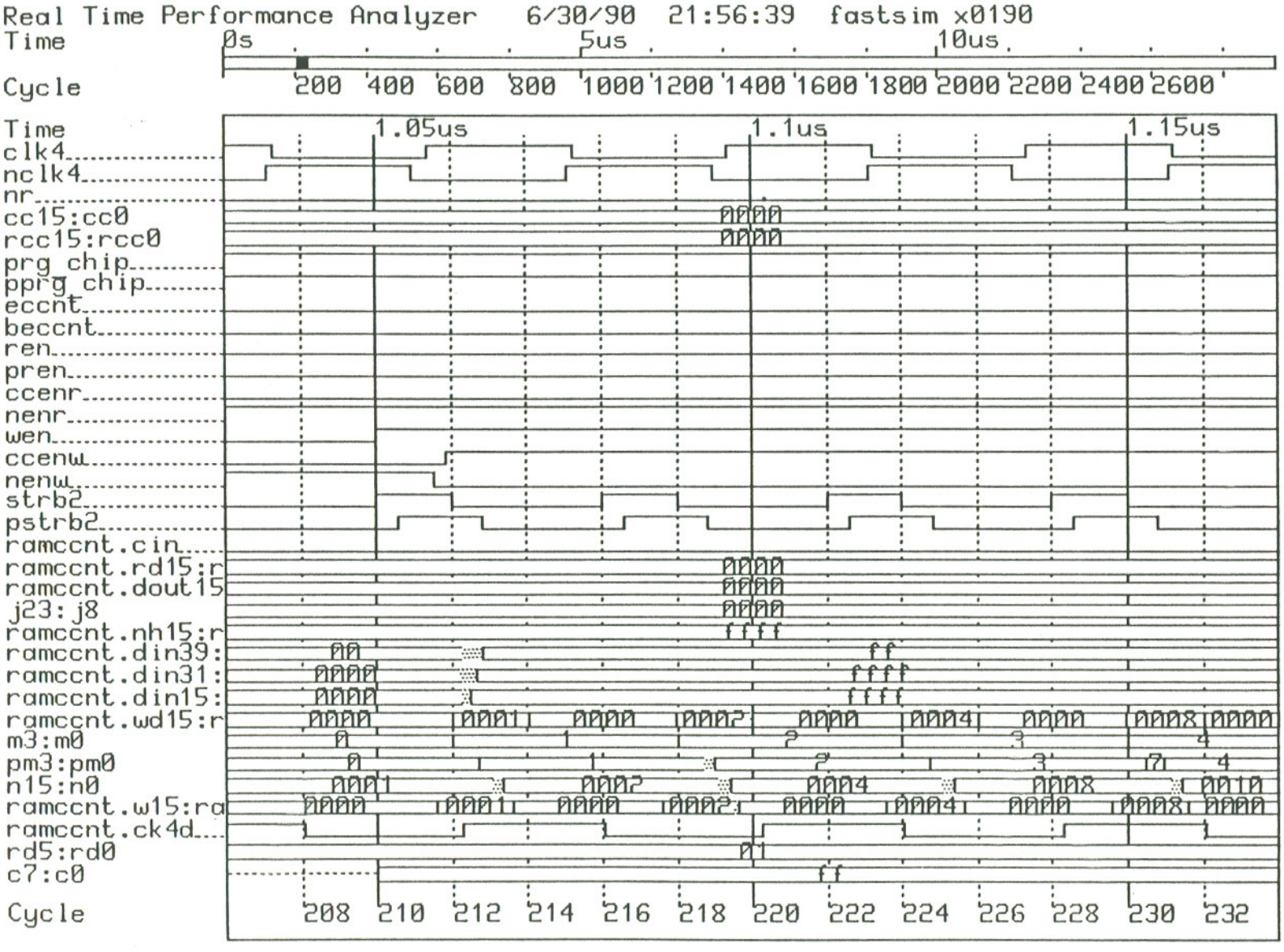
Plot 2


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
5.3us
5.32us
5.34us
5.36us
5.38us
5.4us
ck
clk
prg chip
d3:d0
valid
bvalid
nr
bnr.3
en_timer
bnen_timer
chip_mode
a31:a16
a15:a0
b31:b16
b15:b0
rr0.b31.1:rr0.
rr0.b15.1:rr0.
rr0.ll31:rr0.l
rr0.ll15:rr0.l
rr0.ul31:rr0.u
rr0.ul15:rr0.u
rr0.active
rr0.exitrange
rr0.active.3
rr0.inc
rr0.cntcin
rr0.e7:rr0.e0
rr0.g7:rr0.g0
g7;g0
rr0.ccout
rr0.ecout
Cycle
1062 1064 1066 1068 1070 1072 1074 1076 1078 1080 1082

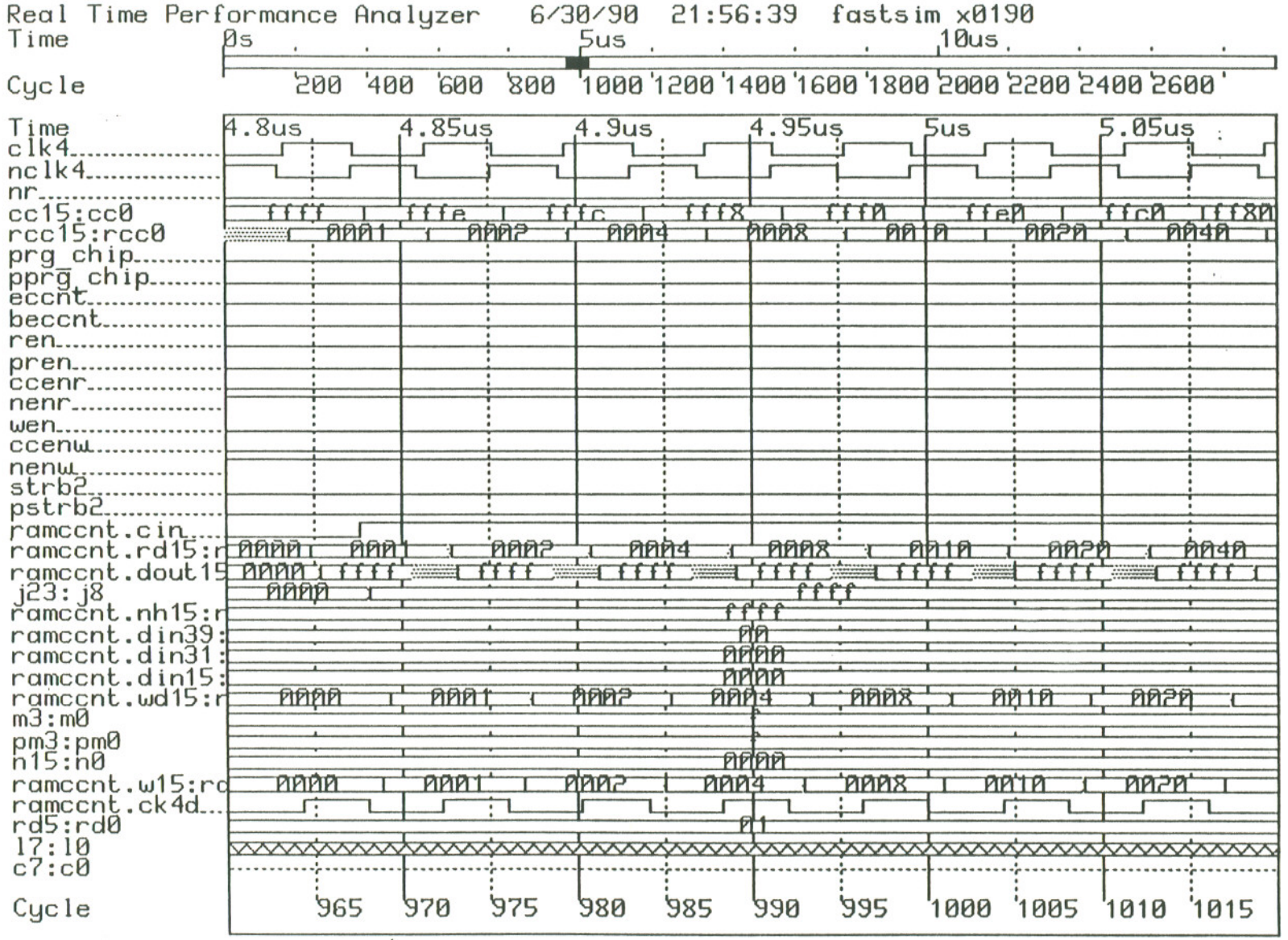
Plot 3


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
0s
200ns
400ns
600ns
clk4
nclk4
nr
cc15:cc0
rcc15:rcc0
prg_chip
pprg_chip
eccnt
beccnt
ren
pren
ccenr
nenr
wen
ccenw
nenw
strb2
pstrb2
ramccnt.cin
ramccnt.rd15:r
ramccnt.dout15
j23:j8
ramccnt.nh15:r
ramccnt.din39:
ramccnt.din31:
ramccnt.din15:
ramccnt.wd15:r
m3:m0
pm3:pm0
n15:h0
ramccnt.w15:rc
ramccnt.ck4d...
rd5:rd0
l7:l0
c7:c0
Cycle
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150

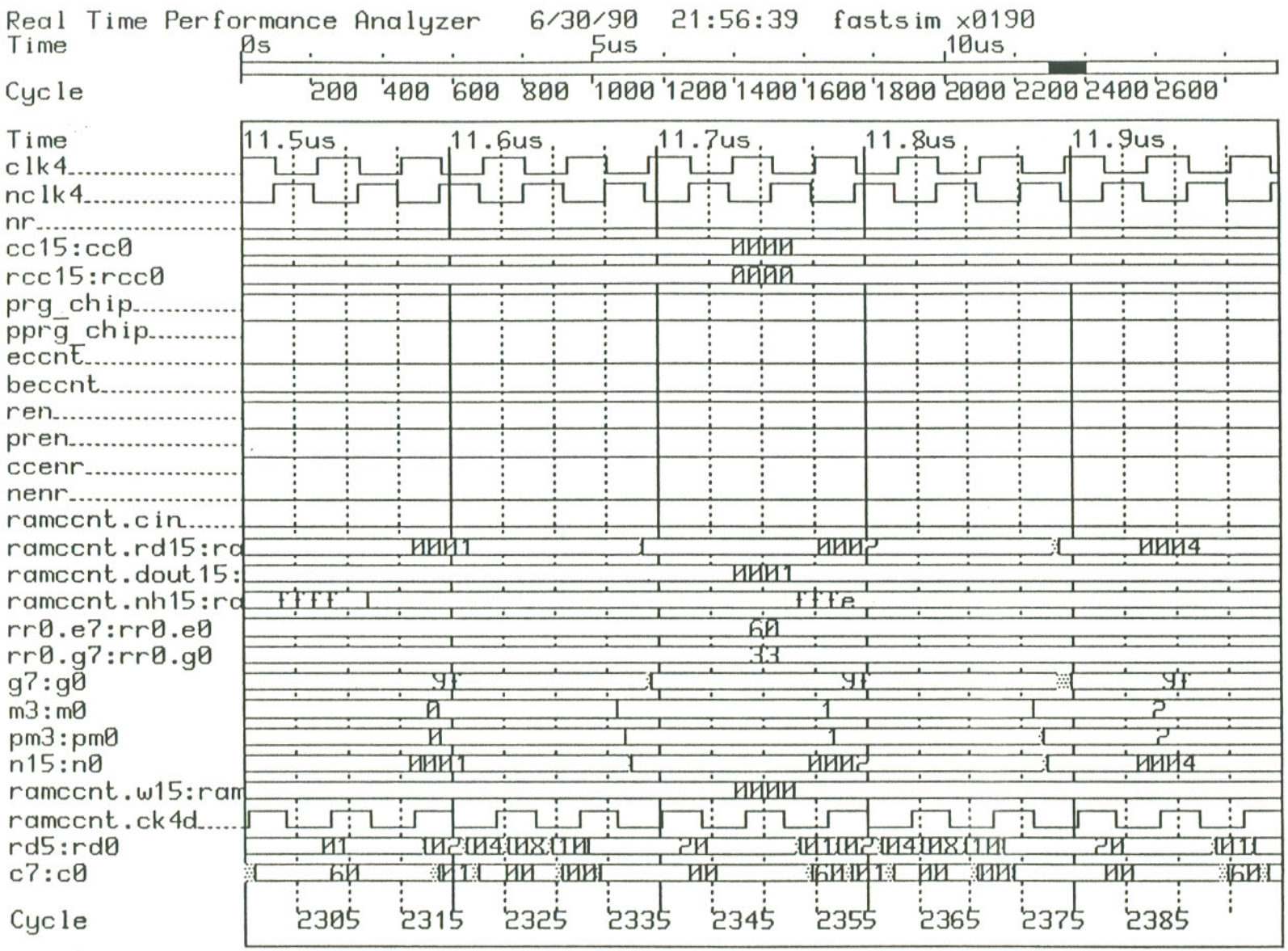
Plot 4


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
1.05us
1.1us
1.15us
clk4
nclk4
nr
cc15:cc0
rcc15:rcc0
prg_chip
pprg_chip
eccnt
beccnt
ren
pren
ccenr
nenr
wen
ccenw
nenw
strb2
pstrb2
ramccnt.cin
ramccnt.rd15:r
ramccnt.dout15
j23:j8
ramccnt.nh15:r
ramccnt.din39:
ramccnt.din31:
ramccnt.din15:
ramccnt.wd15:r
m3:m0
pm3:pm0
n15:n0
ramccnt.w15:ra
ramccnt.ck4d
rd5:rd0
c7:c0
Cycle
208 210 212 214 216 218 220 222 224 226 228 230 232

Plot 5


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
4.8us
4.85us
4.9us
4.95us
5us
5.05us
clk4
nclk4
nr
cc15:cc0
rcc15:rcc0
prg_chip
pprg_chip
eccnt
beccnt
ren
pren
ccenr
nenr
wen
ccenw
nenw
strb2
pstrb2
ramccnt.cin
ramccnt.rd15:r
ramccnt.dout15
j23:j8
ramccnt.nh15:r
ramccnt.din39:
ramccnt.din31:
ramccnt.din15:
ramccnt.wd15:r
m3:m0
pm3:pm0
n15:n0
ramccnt.w15:rc
ramccnt.ck4d...
rd5:rd0
l7:l0
c7:c0
Cycle
965 970 975 980 985 990 995 1000 1005 1010 1015

Plot 6


Real Time Performance Analyzer 6/30/90 21:56:39 fastsim x0190
Time
0s
5us
10us
Cycle
200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400 2600
Time
11.5us
11.6us
11.7us
11.8us
11.9us
clk4
nclk4
nr
cc15:cc0
rcc15:rcc0
prg_chip
pprg_chip
eccnt
beccnt
ren
pren
ccenr
nenr
ramccnt.cin
ramccnt.rd15:ra
ramccnt.dout15:
ramccnt.nh15:ra
rr0.e7:rr0.e0
rr0.g7:rr0.g0
g7:g0
m3:m0
pm3:pm0
n15:n0
ramccnt.w15:ram
ramccnt.ck4d
rd5:rd0
c7:c0
Cycle
2305 2315 2325 2335 2345 2355 2365 2375 2385

Plot 7

5. SUMMARY AND CONCLUSIONS

The need for a cheap and effective tool to accomplish non-intrusive source level performance analysis of real time software was identified in Chapter 2. Some efforts to develop such a tool and the pros and cons of these tools were discussed.

The macroarchitecture of a real time performance analysis chip was discussed in Chapter 3. The reason for choosing a particular architecture and its advantages were discussed. It was shown that by reducing the demand on the capabilities of the tool, high performance at a low price could be obtained by designing a simple but efficient chip which uses a simple technique to aid non-intrusive performance analysis.

In Chapter 4, the microarchitecture of the chip was presented. The logic equations and the implementation of the required functions using the ADG 1.5 μm standard cell library was presented. Results of the layout of the chip and a post layout simulation were presented.

In summary, a high performance chip to aid real time non-intrusive performance analysis was sucessfully simulated and laid out. The chip implements a simple algorithm to accomplish this purpose and provides the capability to do performance analysis at speeds as high at 100 MHz. A set of such chips combined with a state analyzer can make an extremely effective system for debugging and performance analysis. The current version of the chip has only 16 ranges. This is admittedly a small number but this was largely necessitated by the speed requirement. Further, as already discussed, almost no effort was put into laying out the chip. The regular and hierarchical structure of the chip assures that a custom layout would save significant amount of space. The addition of more ranges merely involves adding more range

recognizer blocks and expanding the priority resolve and the RAM circuit which is fairly easy to accomplish. It is possible to further reduce the chip area by reducing the number of pipeline stages in the comparators in the range recognizers. This can be accomplished by designing standard cells to accomplish this function. Also using 1 micron technology would buy more chip area. No software has yet been written to program and run the chip and and the question of integrating the chip with state analyzer has not yet been studied. It is my opinion that further work should be directed at these areas.

References

1. W. Morven Gentleman and Henry Hoeksma, ''Hardware assisted high level debugging (preliminary draft),'' *ACM national conference proceedings*, pp. 140-144, 1983.

2. Robert L. Glass, ''Real-time: The "lost world" of software debugging and testing,'' *Communications of the ACM*, vol. 23, Number 5, pp. 264-271, May 1980.

3. Victor R. Basili and Richard W. Selby, ''Comparing the effectiveness of software testing strategies,'' *IEEE transactions on software engineering*, vol. SE-13, No. 12, pp. 1278-1296, December 1987.

4. Richard Hamlet, ''Special section on software testing,'' *Communications of the ACM*, vol. 31, Number 6, pp. 662-667, June 1988.

5. David Gelperin and Bill Hetzel, ''The growth of software testing,'' *Communications of the ACM*, vol. 31, Number 6, pp. 687-695, June 1988.

6. S. C. Johnson, ''Lint, a C Program Checker,'' *Comp. Sci. Tech. Rep. No. 65*, December 1977.

7. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, ''gprof: A call graph execution profiler,'' *SIGPLAN Notices 17*, pp. 120-126, June 1982.

8. Bernhard Plattner and Jurg Nievergelt, ''Monitoring program execution: a survey,'' *IEEE Computer*, vol. 14, pp. 76-93, Nov. 1981.

9. C. A. Witschorik, ''The real-time debugging monitor for the Bell system 1A processor,'' *Software-Practice and Experience*, vol. 13, pp. 727-743, 1983.

10. Richard N. Taylor, ''Debugging real-time software in a host-target environment,'' *Technology and science of informatics*, vol. 3, pp. 229-236, 1984.

11. Niklaus Wirth, ''Toward a discipline of real-time programming,'' *Communications of the ACM*, vol. 20, pp. 577-583, August 1977.

12. Kwang-Ya Fang and Carl K. Chang, ''Non interference software debugging and testing for real-time systems,'' *IEEE computer and software application conference proc.*, pp. 326-32, 1985.

13. Charles R. Hill, ''A real-time microprocessor debugging technique,'' *ACM national conference proceedings*, pp. 145-148, March 20-3, 1983.

14. Bernhard Plattner, ''Real-time execution monitoring,'' *IEEE Transactions on software engineering*, vol. SE-10, No. 6, pp. 756-764, Nov. 1984.

15. Thomas Bemmerl, ''Realtime high level debugging in host/target environments,'' *Microprocessing and microprogramming*, vol. 18, pp. 387-400, Dec. 1986.

16. Charles H. Small, ''Login analyzers evolve in response to high-level languages,'' *EDN*, pp. 61-72, February 6, 1986.

17. Charles H. Small, ''Software analyzer traces high-level program execution in real time,'' *EDN*, pp. 83-84, April 18, 1985.

18. Bruce Ableidinger, Nirmal Agarwal, and Chuck Nobles, ''Real-time analyzer furnishes high-level look at software operation,'' *Electronic Design*, September 19, 1985.

19. Bruce Ableidinger, ''Hardware tool streamlines software testing,'' *Computer Decisions*, October 1, 1985.

20. Ted Gary, ''Optimizing and verifying software in embedded systems,'' *Digital Design*, pp. 58-62, August 1986.

21. Gus Noelcke, ''Debug system targets multiprocessor design,'' *Computer Design*, pp. 105-114, November 1, 1986.

APPENDIX A

```
/* Behavioral simultion of real time performance analyzer
 * Second Version: Written Sept 15, 1989 by Mehul Dave
 *
 * Reads input from a file line wise.
 * Format of line is
 * f1  f2  f3  f4  f5 f6 f7 f8 f9 f10 f11
 * where
 *    f1  : Chip reset (set all counters to zero)
 *      f2  : program chip (0/1)
 *    f3  : Upper/lower(1/0) limit address - for programming
 *    f4  : mode of analysis for chip
 *                0 -> Count number of addresses in a range
 *                1 -> Time a range
 *    f5  : Enable/disable timer.  For time a range mode.  If disabled, don't
 *          count.
 *    f6  : valid/invalid(1/0) input data.  for PA measurements
 *    f7  : input data (32 bit in hex)
 *    f8  : Range Recognizer (RR) number
 *    f9  : read time/address count or entry/exit count
 *    f10 : read enable
 *    f11 : the byte number to be read (of the count)
 *
 * Outputs result of the performance analysis on stdout.
 */

#include <stdio.h>
#define MAXLINE 1000 /* Maximum character in one input line */
#define UPPER 1    /* data word indicates upper limit for compare */
#define LOWER 0    /* data word indicates lower limit for compare */
#define MAX  16    /* Number of range recognizers */
#define COUNT_ADDR 0 /* chip mode to count addreses in a range */
#define TIME_A_RANGE 1 /* chip mode to time a range */

int reset,prg_chip,limit,chip_mode,en_timer,valid,rrnumber,eccnt,
     ren,enr;
int active[MAX];        /* Bit to indicate if RR is active or not */
long data,data2;
long upper[MAX], lower[MAX];  /* Upper and lower limits of RRs */
long ccount[MAX], ecount[MAX]; /* Time/Address or entry/exit count for RRs */
int linenumber = 0;

main ()
{
```

```
    int  i,j;
    char line[MAXLINE];

    i = j = 0;
    /* Get a line from input file till end of file encountered */

    while (fgets(line,MAXLINE,stdin) != NULL) {
      ++linenumber;

        i = (sscanf(line,"%d %d %d %d %d %d %x %d %d %d %d",
            &reset, &prg_chip, &limit, &chip_mode, &en_timer, &valid,
            &data, &rrnumber, &eccnt, &ren, &enr));
/*
      for (j = 0; j < MAX; ++j)
            printf ("line %d: active[%2d] = %d0,linenumber,j,active[j]);
*/
        if (i != 11) {
          printf("Invalid data format on line %d0,linenumber);
          printf("Each line must have exactly 11 fields0);
          printf("0borting Program execution0);
          exit (1);
        }

        if (reset) {

        /* Set the time/address counts and entry/exit counts for all
           range recognizers to zero */

          for ( i=0; i < MAX; ++i) {
              ccount[i] = 0;
              ecount[i] = 0;
          }
        }
        else if (prg_chip)
            program_chip();
        else if (((chip_mode == COUNT_ADDR) && valid) ||
                ((chip_mode == TIME_A_RANGE) && en_timer))
          accept_data();
        else if (ren) {  /* Read the required byte of required count from RAM */
          if (eccnt)
            data2 = (ecount[rrnumber] & (0xFF << (8*enr)));
          else
            data2 = (ccount[rrnumber] & (0xFF << (8*enr)));

          printf("Eccnt = %1d,  Byte %1d for RR %2d = %2x0,eccnt,
                  enr,rrnumber,data2);
        }
```

```
        else {
            printf("Invalid data conditions on line %d0, linenumber);
          printf("0borting Program execution0);
          exit(1);
        }
    }

    printf("-----------------------------------------0);
    printf("|  RRNumber  |  Time/Event  |  Entry/exit  |0);
    printf("|            |     Count    |     Count    |0);
    printf("|-----------------------------------------|0);
    for (i = 0; i<MAX; ++i)
      if ((ccount[i] != 0) || (ecount[i] != 0)) {
    printf("|    %3d     |   %6d     |   %6d     |0,i,ccount[i],ecount[i]);
    }
    printf("-----------------------------------------0);
}

program_chip()
{

    if ((rrnumber >= MAX) || (rrnumber < 0)) {
        printf("Invalid rrid %d on line %d0,rrnumber,linenumber);
        exit(1);
        }
    else if (!((chip_mode == TIME_A_RANGE)||(chip_mode == COUNT_ADDR))) {
      printf("Invalid chip mode %d on line %d0,chip_mode,linenumber);
            exit(1);
    }
    else if (limit == UPPER)
        upper[rrnumber] = data;
    else if (limit == LOWER)
      lower[rrnumber] = data;
    else {
        printf("Invalid limitid %d on line %d0, limit, linenumber);
        exit(1);
     }
}

accept_data()
{
    int i = 0;

    for (i=0; i<MAX; ++i) {
      if (valid) {
          if ((data >= lower[i]) && (data <= upper[i]))  {
            ++ccount[i];                /* increment address/time count */
```

```
            /* if data equals lower limit when the range is active,
               increment entry/exit count.  This is for recursive
               procedures */

            if ((active[i]) && (data == lower[i]))
               ++ecount[i];
            if (!active[i])
                active[i] = 1; /* Activate status bit */
          }
          else

          /* if addr. equals upper limit or out of range, then
            end of that range if that range was active */

            if (((data < lower[i]) || (data > upper[i])) &&
                  (active[i])) {
                active[i] = 0; /* deactivate status bit */
                ++ecount[i];
              }
       }

      /* If timing a range, then increment count if inside that range i.e.
         if the range is active */

       else if ((!valid) && en_timer && (chip_mode == TIME_A_RANGE)) {

          if (active[i])
            ++ccount[i];
       }
     }
}
```

APPENDIX B

LIST OF ADG 1.5u CMOS STANDARD CELLS

LEVEL SHIFTERS

C501 (Q IN)	TTL to CMOS level shifter (non-invert)
C507 (Q IN)	TTL to CMOS level shifter (inverting)
C510 (VSS)	GROUND cell
C511 (VDD)	VDD cell

PAD CELLS

C592 (PG NG DOUT)	pad output driver (inverting)
C592HV (PG NG DOUT)	pad output driver (inverting, 1/2 power)
C592I (PG NG DOUT)	pad output driver (non-invert)
C592IHV (PG NG DOUT)	pad output driver (non-invert,1/2 power)
C593 (PG NG DOUT CSB)	tri-state pad output driver (invert)
C593HV (PG NG DOUT CSB)	tri-state pad output driver (invert, 1/2 power)
C593I (PG NG DOUT CS)	tri-state pad output driver (non-invert)
C593IHV (PG NG DOUT CS)	tri-state pad output driver (non-invert, 1/2 power)
C920 (PAD)	input pad with protection
C920PU (PAD)	input pad with protection and pull-up
C920PD (PAD)	input pad with protection and pull-down
C920E (PAD)	input pad with protection (ECL vers.)
C930 (PAD PGATE NGATE)	output pad
C930D (PAD PGATE NGATE)	output pad (double-strength)
C930HV (PAD PGATE NGATE)	output pad (half-strength)
C930PU (PAD PGATE NGATE)	output pad with pull-up
C930PD (PAD PGATE NGATE)	output pad with pull-down

BUFFERS

X101 (Q I1)	inverter
S101 (Q I1)	inverting buffer - double size

```
S101X2 (Q  I1)          inverting buffer - 2x drive
S101X3 (Q  I1)          inverting buffer - 3x drive
S101X4 (Q  I1)          inverting buffer - 4x drive
Z101   (Q  I1)          non-inverting buffer - double size
Z101X2 (Q  I1)          non-inverting buffer - 2x drive
Z101X4 (Q  I1)          non-inverting buffer - 4x drive

DLY2X (Q I1)            delay cell (2 inverters)
DLY4X (Q I1)            delay cell (4 inverters)
DLY8X (Q I1)            delay cell (8 inverters)

C519 (NQ DIS IN)        tristate buffer (inverting)
C520 (Q  DIS IN)        fast tristate buffer (non-inverting)

BUFSUS (NODE)           tristate bus line sustainer
```

GATES

```
X102   (Q I1 I2)                  2 input nor
X102X2 (Q I1 I2)                  2 input nor, 2x drive
X102X3 (Q I1 I2)                  2 input nor, 3x drive
X102X4 (Q I1 I2)                  2 input nor, 4x drive
X103   (Q I1 I2 I3)               3 input nor
X104   (Q I1 I2 I3 I4)            4 input nor
X105   (Q I1 I2 I3 I4 I5)         5 input nor
X106   (Q I1 I2 I3 I4 I5 I6)      6 input nor
X107   (Q I1 I2 I3 I4 I5 I6 I7)   7 input nor
X108   (Q I1 I2 I3 I4 I5 I6 I7 I8) 8 input nor

X162   (Q I1 I2)                  2 input nand
X162X2 (Q I1 I2)                  2 input nand, 2x drive
X162X4 (Q I1 I2)                  2 input nand, 4x drive
X162X6 (Q I1 I2)                  2 input nand, 6x drive
X163   (Q I1 I2 I3)               3 input nand
X164   (Q I1 I2 I3 I4)            4 input nand
X165   (Q I1 I2 I3 I4 I5)         5 input nand
X166   (Q I1 I2 I3 I4 I5 I6)      6 input nand
X167   (Q I1 I2 I3 I4 I5 I6 I7)   7 input nand
X168   (Q I1 I2 I3 I4 I5 I6 I7 I8) 8 input nand

X402   (Q I1 I2)                  2 input and
X402X2 (Q I1 I2)                  2 input and, 2x drive
X402X4 (Q I1 I2)                  2 input and, 4x drive
X403   (Q I1 I2 I3)               3 input and
```

```
X403X2 (Q I1 I2 I3)                  3 input and, 2x drive
X404   (Q I1 I2 I3 I4)               4 input and
X404X2 (Q I1 I2 I3 I4)               4 input and, 2x drive
X405   (Q I1 I2 I3 I4 I5)            5 input and
X406   (Q I1 I2 I3 I4 I5 I6)         6 input and
X407   (Q I1 I2 I3 I4 I5 I6 I7)      7 input and
X408   (Q I1 I2 I3 I4 I5 I6 I7 I8)   8 input and

X432   (Q I1 I2)                     2 input or
X433   (Q I1 I2 I3)                  3 input or
X434   (Q I1 I2 I3 I4)               4 input or
X435   (Q I1 I2 I3 I4 I5)            5 input or
X436   (Q I1 I2 I3 I4 I5 I6)         6 input or
X437   (Q I1 I2 I3 I4 I5 I6 I7)      7 input or
X438   (Q I1 I2 I3 I4 I5 I6 I7 I8)   8 input or
```

DATA LATCHES

```
TTLLAT (NQ EN D)              d-latch, TTL-level input    (low enable)
X318   (Q NQ EN D)            d-latch                     (low enable)
X318X2 (Q NQ EN D)            d-latch, 2x drive           (low enable)
X319   (Q NQ QT EN D DIS)     d-latch with tri-state      (low enable)
X320   (Q NQ EN D NR)         d-latch  with Reset         (low enable)
X321   (Q NQ QQ EN D NR)      d-latch  with Nand          (low enable)
X322   (Q NQ EN D)            d-latch                     (high enable)
X322W4 (Q0..Q3 NQ0..NQ3, D0..D3 EN) 4-bit d-latch        (high enable)
X322W8 (Q0..Q7 NQ0..NQ7, D0..D7 EN) 8-bit d-latch        (high enable)
X323   (Q NQ EN D NR)         d-latch  with Reset         (high enable)
```

FLIP FLOPS

```
X351   (Q NQ CK D)                     D-FF                      (neg edge)
X351W4 (Q0..Q3 NQ0..NQ3 CK D0..D3)     4-bit D-FF                (neg edge)
X351W5 (Q0..Q4 NQ0..NQ4 CK D0..D4)     4-bit D-FF                (neg edge)
X351DY (Q NQ CK D)                     dynamic D-FF              (neg edge)
X351DYW4 (Q0..Q3 NQ0..NQ3 CK D0..D3)   4-bit dynamic D-FF        (neg edge)
X351DYW8 (Q0..Q7 NQ0..NQ7 CK D0..D7)   8-bit dynamic D-FF        (neg edge)
X352   (Q NQ CK D NR)                  D-FF with Reset           (neg edge)
X352W4 (Q0..Q3 NQ0..NQ3 CK D0..D3 NR)  4-bit D-FF with Reset     (neg edge)
X353   (Q NQ CK D NS)                  D-FF with Set             (neg edge)
X354   (Q NQ CK D NR NS)               D-FF with Reset and Set   (neg edge)
```

```
X361    (Q NQ CK D)                      D-FF                          (pos edge)
X361W4  (Q0..Q3 NQ0..NQ3 D0..D3 CK)      4-bit D-FF                    (pos edge)
X361DY    (Q NQ CK D)                    dynamic D-FF                  (pos edge)
X361DYW3 (Q0..Q2 NQ0..NQ2 D0..D2 CK)     3-bit dynamic D-FF            (pos edge)
X361DYW4 (Q0..Q3 NQ0..NQ3 D0..D3 CK)     4-bit dynamic D-FF            (pos edge)
X361DYW5 (Q0..Q4 NQ0..NQ4 D0..D4 CK)     5-bit dynamic D-FF            (pos edge)
X362    (Q NQ CK D NR)                   D-FF with Reset               (pos edge)
X362X2  (Q NQ CK D NR)                   D-FF with Reset,2x drive      (pos edge)
X364    (Q NQ CK D NR NS)                D-FF with Reset and Set       (pos edge)


X371 (Q NQ CK J K)                       JK-FF                         (neg edge)
X372 (Q NQ CK J K NR)                    JK-FF with Reset              (neg edge)
X374 (Q NQ CK J K NR NS)                 JK-FF with Reset and Set      (neg edge)


X381    (Q NQ CK A B SELA )               2 input FF                   (neg edge)
X381W4 (Q0..Q3 NQ0..NQ3 CK A0..A3 B0..B3 SELA)
                                  4-bit 2 input FF          (neg edge)
X381W5 (Q0..Q4 NQ0..NQ4 CK A0..A4 B0..B4 SELA)
                                  5-bit 2 input FF          (neg edge)
X381DY (Q NQ CK A B SELA)              2 input dynamic FF         (neg edge)
x381DYW4 (Q0..Q3 NQ0..NQ3 CK A0..A3 B0..B3 SELA)
                                4-bit 2 input dynamic FF(neg edge)
X382    (Q NQ CK A B SELA NR)             2 input FF with Reset     (neg edge)
X383    (Q NQ CK A B SELA NS)             2 input FF with Set       (neg edge)


X641 (Q NQ CK D NWE)             D-FF with Hold                    (neg edge)
X641W4 (Q0..Q3 NQ0..NQ3 CK D0..D3 NWE)
                           4-bit D-FF with Hold             (neg edge)
X691 (Q NQ CK D NWE SD NSS)      D-FF with Scan and Hold           (neg edge)
X692 (Q NQ CK D NWE SD NSS NR)   D-FF with Scan, Hold and Reset    (neg edge)
X693 (Q NQ CK D NWE SD NSS NS)   D-FF with Scan, Hold and Set      (neg edge)


DECODERS & COMPLEX GATES
------------------------

X412 (Q0 Q1 Q2 Q3 X Y)           2 to 4 line DECODER
X413 (Q0 Q1 Q2 Q3 X Y DIS)       2 to 4 line DECODER w/Disable (X=LSB)
X414ND (Q I0 I1 I2 I3 X Y )      1 of 4 DATA SELECTOR (X=LSB)
X414 (Q I0 I1 I2 I3 X Y DIS)     1 of 4 DATA SELECTOR w/Disable (X=LSB)
X415 (Q A B SELA)                inverting 1 of 2 DATA SELECTOR
X416 (Q A B SELA)                1 of 2 DATA SELECTOR
X417 (Q0 Q1 Q2 Q3 X Y EN)        2 to 4 line DECODER with En (X=LSB)
```

```
X418W4 (Q0 Q1 Q2 Q3 A0 B0 A1 B1 A2 B2 A3 B3 SELB)
                          inverting 4 of 8 DATA SELECTOR
X419W4 (Q0 Q1 Q2 Q3 A0 B0 A1 B1 A2 B2 A3 B3 SELB DIS)
                          inverting 4 of 8 DATA SELECTOR w/Disable

X422 (Q A B C)                  3-Input Majority Gate

X452 (Q A B SELA)               inverting 1 of 2 DATA SELECTOR
X453 (Q A B C SA SB SC)         inverting 1 of 3 DATA SELECTOR
X454 (Q A B C D SA SB SC SD)    inverting 1 of 4 DATA SELECTOR
X455 (Q A B C D E SA SB SC SD SE)
                          inverting 1 of 5 DATA SELECTOR
X456 (Q A B C D E F SA SB SC SD SE SF)
                          inverting 1 of 6 DATA SELECTOR

X113 (Q BI1 BI2 A1)              NAND of 1 2-input OR && 1-Input
X114 (Q A1 B1 A2 B2)            NAND of 2 2-input OR gates
X116 (Q A1 B1 A2 B2 A3 B3)     NAND of 3 2-input OR gates
X124 (Q A1 B1 A2 B2)             NOR of 2 2-input AND gates
X126 (Q A1 B1 A2 B2 A3 B3)      NOR of 3 2-input AND gates
X133 (Q A1 B1 I)                 NOR of 1 2-input AND && 1-Input

Z114 (Q A1 B1 A2 B2)            AND of 2 2-input OR gates
Z116 (Q A1 B1 A2 B2 A3 B3)     AND of 3 2-input OR gates
Z124 (Q A1 B1 A2 B2)             OR of 2 2-input-AND gates
Z126 (Q A1 B1 A2 B2 A3 B3)      OR of 3 2-input-AND gates
GTGEN4
(Q G3 E3 G2 E2 G1 E1 G0)  OR of 4-3-2-1 input AND


ARITHMETIC FUNCTIONS
--------------------

X136  (S0  S1 CO A0  A0 CIN)       2-bit INCREMENTER      (S0 MSB)
X137  (S0..S3 CO A0..A3 CIN)       4-bit INCREMENTER      (S0 MSB)
X137F (S0..S3 CO A0..A3 CIN)       Fast 4-bit INCREMENTER (S0 MSB)
X138  (S0..S7 CO A0..A7 CIN)       8-bit INCREMENTER      (S0 MSB)
X139  (S0  S1 CO A0  A1 CIN)       2-bit DECREMENTER      (S0 MSB)

X420 (Q I1 I2)                     EXCLUSIVE OR
X421 (Q I1 I2)                     EXCLUSIVE NOR

X592  (S0 S1 C0 A0 B0 A1 B1 CIN)   2-bit ADDER
X592N (NS0 NS1 C0 A0 B0 A1 B1 CIN) 2-bit ADDER (inverted outputs)
X594  (S0 S1 S2 S3 C0 A0 B0 A1 B1 A2 B2 A3 B3 CIN)
                             4-bit ADDER
X594N (NS0 NS1 NS2 NS3 C0 A0 B0 A1 B1 A2 B2 A3 B3 CIN)
```

```
                          4-bit ADDER (inverted outputs)
MUL12X12 (P0..P23 NX0..NX11 Y0..Y11 )
                               12-bit signed multiplier
MUL16X16 (P0..P31 NX0..NX15 Y0..Y15 SGNDX SGNDY NSTDBY)
                                 16-bit signed/unsigned multiplier


RAMs
----

C525  ( DOUT RD0..RD3 DIN WR0..WR3 )    4 word x 1 bit
C525F ( DOUT RD0..RD3 DIN WR0..WR3 )    faster, higher power version

C527  ( DOUT RD0..RD7 DIN WR0..WR7 )     8 word x 1 bit
C527F ( DOUT RD0..RD7 DIN WR0..WR7 )    faster, higher power version

C533  ( DOUT RD0..RD15 DIN WR0..WR15 ) 16 word x 1 bit
C533F ( DOUT RD0..RD15 DIN WR0..WR15 ) faster, higher power version
```

LIST OF ADG 1.5u CMOS MACRO CELLS

These macros are distributed in the file cmos15mc. It is important to note that these macros are provided as application examples for the convenience of our customers. Final netlists submitted to ADG must be flattened to the standard cell level. Netlists so flattened will contain NO macros.

```
PAD Macros
----------

IPADI   (DIN, PAD)          inverting TTL input pad macro
IPADT   (DIN, PAD)          non-inverting TTL input pad macro
OPADI   (PAD, DOUT)         inverting TTL/CMOS output pad macro
OPADIHV              1/2 power,
      (PAD, DOUT)         inverting TTL/CMOS output pad macro
OPADT   (PAD, DOUT)         non-invert. TTL/CMOS output pad macro
OPADTHV              1/2 power,
      (PAD, DOUT)         non-invert. TTL/CMOS output pad macro
TSPADI (PAD, DOUT CSB)    inverting TTL/CMOS tristate output
                                pad macro with disable
TSPADIHV             1/2 power,
      (PAD, DOUT CSB)     inverting TTL/CMOS tristate output
                                pad macro with disable
```

```
TSPADT (PAD, DOUT CSB)    non-invert. TTL/CMOS tristate output
                                pad macro with disable
TSPADTHV              1/2 power,
      (PAD, DOUT CSB)     non-invert. TTL/CMOS tristate output
                                pad macro with disable
```

APPENDIX C

```
CIRCUIT WIREC

; The input pads

padlimit plimit,limit IPADT
padstrb  pstrb, strb  IPADT
padprgc  pprg_chip, prg_chip IPADT
padnr  pnr, nr IPADT
padvalid pvalid,valid IPADT
padtimer pen_timer, en_timer IPADT
padcm  pchip_mode, chip_mode IPADT

padcka       pck, ck IPADT

; The input pads for input data.  These pads have inverting buffers so
; data is inverted.

pada31       pa31, a31 IPADI
pada30       pa30, a30 IPADI
pada29       pa29, a29 IPADI
pada28       pa28, a28 IPADI
pada27       pa27, a27 IPADI
pada26       pa26, a26 IPADI
pada25       pa25, a25 IPADI
pada24       pa24, a24 IPADI

pada23       pa23, a23 IPADI
pada22       pa22, a22 IPADI
pada21       pa21, a21 IPADI
pada20       pa20, a20 IPADI
pada19       pa19, a19 IPADI
pada18       pa18, a18 IPADI
pada17       pa17, a17 IPADI
pada16       pa16, a16 IPADI

pada15       pa15, a15 IPADI
pada14       pa14, a14 IPADI
pada13       pa13, a13 IPADI
```

```
pada12       pa12, a12 IPADI
pada11       pa11, a11 IPADI
pada10       pa10, a10 IPADI
pada9  pa9, a9 IPADI
pada8  pa8, a8 IPADI

pada7  pa7, a7 IPADI
pada6  pa6, a6 IPADI
pada5  pa5, a5 IPADI
pada4  pa4, a4 IPADI
pada3  pa3, a3 IPADI
pada2  pa2, a2 IPADI
pada1  pa1, a1 IPADI
pada0  pa0, a0 IPADI

padd3  pd3, d3 IPADT
padd2  pd2, d2 IPADT
padd1  pd1, d1 IPADT
padd0  pd0, d0 IPADT

; Inverting Buffers to drive output of pa31-pa0 into the latches and ffs

bufa31      b31, pa31 S101X2
bufa30      b30, pa30 S101X2
bufa29      b29, pa29 S101X2
bufa28      b28, pa28 S101X2
bufa27      b27, pa27 S101X2
bufa26      b26, pa26 S101X2
bufa25      b25, pa25 S101X2
bufa24      b24, pa24 S101X2

bufa23      b23, pa23 S101X2
bufa22      b22, pa22 S101X2
bufa21      b21, pa21 S101X2
bufa20      b20, pa20 S101X2
bufa19      b19, pa19 S101X2
bufa18      b18, pa18 S101X2
bufa17      b17, pa17 S101X2
bufa16      b16, pa16 S101X2

bufa15      b15, pa15 S101X2
bufa14      b14, pa14 S101X2
bufa13      b13, pa13 S101X2
bufa12      b12, pa12 S101X2
bufa11      b11, pa11 S101X2
bufa10      b10, pa10 S101X2
bufa9 b9, pa9 S101X2
```

```
bufa8 b8, pa8 S101X2

bufa7 b7, pa7 S101X2
bufa6 b6, pa6 S101X2
bufa5 b5, pa5 S101X2
bufa4 b4, pa4 S101X2
bufa3 b3, pa3 S101X2
bufa2 b2, pa2 S101X2
bufa1 b1, pa1 S101X2
bufa0 b0, pa0 S101X2

; build the clock buffer

ckbufp      cka, pck Z101
ckbufa      ck1, cka Z101
ckbufb      ck1, cka Z101
ckbufc      ck1, cka Z101
ckbufd      ck1, cka Z101
ckbuf1      clk, ck1 Z101
ckbuf2      clk, ck1 Z101
ckbuf3      clk, ck1 Z101
ckbuf4      clk, ck1 Z101
ckbuf5      clk, ck1 Z101
ckbuf6      clk, ck1 Z101
ckbuf7      clk, ck1 Z101
ckbuf8      clk, ck1 Z101
ckbuf9      clk, ck1 Z101
ckbuf10     clk, ck1 Z101
ckbuf11     clk, ck1 Z101
ckbuf12     clk, ck1 Z101
ckbuf13     clk, ck1 Z101
ckbuf14     clk, ck1 Z101
ckbuf15     clk, ck1 Z101
ckbuf16     clk, ck1 Z101
ckbuf17     clk, ck1 Z101
ckbuf18     clk, ck1 Z101
ckbuf19     clk, ck1 Z101
ckbuf20     clk, ck1 Z101
ckbuf21     clk, ck1 Z101
ckbuf22     clk, ck1 Z101
ckbuf23     clk, ck1 Z101
ckbuf24     clk, ck1 Z101
ckbuf25     clk, ck1 Z101
ckbuf26     clk, ck1 Z101
ckbuf27     clk, ck1 Z101
ckbuf28     clk, ck1 Z101
```

```
wiclk1      ck1   WIREOR: ci=1/4 ; split capacitance on the node between 4 Z101s
wiclk2      clk   WIREOR: ci=1/28 ; split capacitance on the node between 28 Z101

; The flip-flop to capture and generate the reset signal

ffreset     nr.3 nr.2 nr.1 # # # # nr.0, clk nr.2 nr.1 nr.0 pnr X351W4

; Buffer to drive the reset signal into each compare block

rstbuf      bnr.3, nr.3 Z101X4

; Generate upper or lower limit programming signals

bufl  blimit, plimit Z101
invl  bnlimit, plimit X101

bustrba     strba, pstrb Z101X4
bustrbb     strbb, pstrb Z101X4

; The decoder tree for programming the latches of each range

decodeu     decode0 decode1 decode2 decode3, pd2 pd3 pprg_chip X417

decode1     prg_lat0 prg_lat1 prg_lat2 prg_lat3, pd0 pd1 decode0 X417
decode2 prg_lat4 prg_lat5 prg_lat6 prg_lat7, pd0 pd1 decode1 X417
decode3 prg_lat8 prg_lat9 prg_lat10 prg_lat11, pd0 pd1 decode2 X417
decode4 prg_lat12 prg_lat13 prg_lat14 prg_lat15, pd0 pd1 decode3 X417

; Delay the valid and en_timer signals to synchronize data with these
; signals

dlyvalid dvalid, pvalid dly4x

dlytimer1 den_timer, pen_timer dly4x


; Timer enable-disable,valid,chip_mode ffs

ffvalid valid.1 valid.0 en_timer.1 en_timer.0 nvalid.1 $
      nvalid.0 nen_timer.1 nen_timer.0, clk $
      valid.0 dvalid en_timer.0 den_timer bnr.3 X352W4

fftimer en_timer.2 nen_timer.2, clk en_timer.1 bnr.3 X352
ffcm    chip_mode.3 nchip_mode.3, clk pchip_mode bnr.3 X352

; Buffers to drive the chip_mode, en_timer and valid signals
```

```
bufvalid bvalid, valid.1 Z101X2
bufcm bchip_mode, chip_mode.3 Z101X2
bufncm      bnchip_mode, nchip_mode.3 Z101X2
bufnent     bnen_timer, nen_timer.2 Z101X2

; The range recognizers

rr0   rr0.ccout rr0.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat0 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc0 rec0 nn0 beccnt compare

rr1   rr1.ccout rr1.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat1 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc1 rec1 nn1 beccnt compare

rr2   rr2.ccout rr2.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat2 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc2 rec2 nn2 beccnt compare

rr3   rr3.ccout rr3.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat3 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc3 rec3 nn3 beccnt compare

rr4   rr4.ccout rr4.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat4 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc4 rec4 nn4 beccnt compare

rr5   rr5.ccout rr5.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat5 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc5 rec5 nn5 beccnt compare

rr6   rr6.ccout rr6.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat6 strba clk bnen_timer bvalid $
```

```
  bchip_mode bnchip_mode bnr.3 rcc6 rec6 nn6 beccnt compare

rr7   rr7.ccout rr7.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat7 strba clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc7 rec7 nn7 beccnt compare

rr8   rr8.ccout rr8.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat8 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc8 rec8 nn8 beccnt compare

rr9   rr9.ccout rr9.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat9 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc9 rec9 nn9 beccnt compare

rr10  rr10.ccout rr10.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat10 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc10 rec10 nn10 beccnt compare

rr11  rr11.ccout rr11.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat11 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc11 rec11 nn11 beccnt compare

rr12  rr12.ccout rr12.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat12 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc12 rec12 nn12 beccnt compare

rr13  rr13.ccout rr13.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat13 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc13 rec13 nn13 beccnt compare

rr14  rr14.ccout rr14.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
```

```
  blimit bnlimit prg_lat14 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc14 rec14 nn14 beccnt compare

rr15  rr15.ccout rr15.ecout g7 g6 g5 g4 g3 g2 g1 g0 $
      b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20 b19 b18 b17 $
      b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 $
      blimit bnlimit prg_lat15 strbb clk bnen_timer bvalid $
      bchip_mode bnchip_mode bnr.3 rcc15 rec15 nn15 beccnt compare

; Generate the 40ns clock from the master clock by divide by 4 circuit

ff1   clk2 nclk2, clk nclk2 bnr.3 X352
ff2   clk40 nclk40, clk2 nclk40 bnr.3 X352
bufnclk nclk4, nclk40 Z101
bufclk1     clk4, clk40 Z101X4
bufclk2 clk4, clk40 Z101X4

wiclk4      clk4 WIREOR : ci = 1/2

; Input flipflops which trap the carry from comparators

ffcc4 cc3 cc2 cc1 cc0 ncc3 ncc2 ncc1 ncc0, clk4 $
      rr3.ccout rr2.ccout rr1.ccout rr0.ccout X351W4
ffcc8 cc7 cc6 cc5 cc4 ncc7 ncc6 ncc5 ncc4, clk4 $
      rr7.ccout rr6.ccout rr5.ccout rr4.ccout X351W4
ffcc12      cc11 cc10 cc9 cc8 ncc11 ncc10 ncc9 ncc8, clk4 $
      rr11.ccout rr10.ccout rr9.ccout rr8.ccout X351W4
ffcc16      cc15 cc14 cc13 cc12 ncc15 ncc14 ncc13 ncc12, clk4 $
      rr15.ccout rr14.ccout rr13.ccout rr12.ccout X351W4

ffec4 ec3 ec2 ec1 ec0 nec3 nec2 nec1 nec0, clk4 $
      rr3.ecout rr2.ecout rr1.ecout rr0.ecout X351W4
ffec8 ec7 ec6 ec5 ec4 nec7 nec6 nec5 nec4, clk4 $
      rr7.ecout rr6.ecout rr5.ecout rr4.ecout X351W4
ffec12      ec11 ec10 ec9 ec8 nec11 nec10 nec9 nec8, clk4 $
      rr11.ecout rr10.ecout rr9.ecout rr8.ecout X351W4
ffec16      ec15 ec14 ec13 ec12 nec15 nec14 nec13 nec12, clk4 $
      rr15.ecout rr14.ecout rr13.ecout rr12.ecout X351W4

; Take data from pads and generate enr/enw signals

padcnt            peccnt, eccnt IPADT
padren            pren, ren IPADT
padwen      pwen, wen IPADT
padstb2     pstrb2, strb2 IPADT
invenr            nenr, pren S101
invenw            nenw, pwen X101
```

```
bufeccnt1   beccnt, peccnt Z101X4
bufeccnt2   beccnt, peccnt Z101X4
inveccnt    neccnt, beccnt X101
wibeccnt    beccnt WIREOR: ci=1/2

andccenr ccenr, neccnt pren X402X2
andccenw ccbenw, neccnt pprg_chip pwen X403
andccenw2 ccenw2, neccnt pprg_chip pstrb2 X403X2
bufccenw ccenw, ccbenw Z101X2

andecenr ecenr, beccnt pren X402X2
andecenw ecbenw, beccnt pprg_chip pwen X403
andecenw2 ecenw2, beccnt pprg_chip pstrb2 X403X2
bufecenw ecenw, ecbenw Z101X2

; pads to input the signals to decide which bits of RAM to read

padrw0      penr0, enr0 IPADT
padrw1      penr1, enr1 IPADT
padrw2      penr2, enr2 IPADT

; Generate inverts of originals

invenr0     nenr0, penr0 X101
invenr1 nenr1, penr1 X101
invenr2 nenr2, penr2 X101

; Decode the input signals to six seperate signals

decodenr rd0 rd1 rd2 rd3, penr0 penr1 nenr2 X417
andrd4   rd4, penr2 nenr1 nenr0 X403
andrd5       rd5, penr2 nenr1 penr0 X403

; pads to input read address

padm0 pm0, m0 IPADT
padm1 pm1, m1 IPADT
padm2 pm2, m2 IPADT
padm3 pm3, m3 IPADT

; input read signals to read RAM after incrementing is over
; A decoder decodes 4 bits to 16 bits which are then fed to the read
; MUX which feeds the RAM

decodem5  sel0 sel1 sel2 sel3, pm2 pm3 pprg_chip X417

; Rest of the decoding tree
```

```
decodem4  n12 n13 n14 n15, pm0 pm1 sel3   X417
decodem3  n8 n9 n10 n11, pm0 pm1 sel2     X417
decodem2  n4 n5 n6 n7, pm0 pm1 sel1       X417
decodem1  n0 n1 n2 n3, pm0 pm1 sel0       X417

; Invert the output of the decoders

invn0 nn0, n0 S101X2
invn1 nn1, n1 S101X2
invn2 nn2, n2 S101X2
invn3 nn3, n3 S101X2
invn4 nn4, n4 S101X2
invn5 nn5, n5 S101X2
invn6 nn6, n6 S101X2
invn7 nn7, n7 S101X2

invn8 nn8, n8 S101X2
invn9 nn9, n9 S101X2
invn10      nn10, n10 S101X2
invn11      nn11, n11 S101X2
invn12      nn12, n12 S101X2
invn13      nn13, n13 S101X2
invn14      nn14, n14 S101X2
invn15      nn15, n15 S101X2

; The RAM circuit for address/time count

ramccnt     j8 j9 j10 j11 j12 j13 j14 j15 j16 j17 j18 j19 j20 j21 j22 j23 $
      j24 j25 j26 j27 j28 j29 j30 j31 nj32 nj33 nj34 nj35 nj36 nj37 $
      nj38 nj39 nj40 nj41 nj42 nj43 nj44 nj45 nj46 nj47, $
      ccenr ccenw clk4 nclk4 ccenw2 $
      cc0 cc1 cc2 cc3 cc4 cc5 cc6 cc7 cc8 cc9 cc10 cc11 cc12 cc13 $
      cc14 cc15 ncc0 ncc1 ncc2 ncc3 ncc4 ncc5 ncc6 ncc7 ncc8 ncc9 $
      ncc10 ncc11 ncc12 ncc13 ncc14 ncc15 $
      rcc0 rcc1 rcc2 rcc3 rcc4 rcc5 rcc6 rcc7 rcc8 rcc9 $
      rcc10 rcc11 rcc12 rcc13 rcc14 rcc15 $
      nn0 nn1 nn2 nn3 nn4 nn5 nn6 nn7 nn8 nn9 nn10 nn11 nn12 nn13 $
      nn14 nn15 e0 e1 e2 e3 e4 e5 e6 e7 $
      pa0 pa1 pa2 pa3 pa4 pa5 pa6 pa7 pa8 pa9 pa10 pa11 pa12 pa13 $
      pa14 pa15 pa16 pa17 pa18 pa19 pa20 pa21 pa22 pa23 pa24 pa25 $
      pa26 pa27 pa28 pa29 pa30 pa31 wr32 wr33 wr34 wr35 wr36 wr37 $
      wr38 wr39 ram40

; The RAM circuit for entry/exit count

ramecnt     k8 k9 k10 k11 k12 k13 k14 k15 k16 k17 k18 k19 k20 k21 k22 k23 $
      k24 k25 k26 k27 k28 k29 k30 k31, $
```

```
  ecenr ecenw clk4 nclk4 ecenw2 $
      ec0 ec1 ec2 ec3 ec4 ec5 ec6 ec7 ec8 ec9 ec10 ec11 ec12 ec13 $
      ec14 ec15 nec0 nec1 nec2 nec3 nec4 nec5 nec6 nec7 nec8 nec9 $
      nec10 nec11 nec12 nec13 nec14 nec15 $
      rec0 rec1 rec2 rec3 rec4 rec5 rec6 rec7 rec8 rec9 $
      rec10 rec11 rec12 rec13 rec14 rec15 $
      nn0 nn1 nn2 nn3 nn4 nn5 nn6 nn7 nn8 nn9 nn10 nn11 nn12 nn13 $
      nn14 nn15 g0 g1 g2 g3 g4 g5 g6 g7 $
      pa0 pa1 pa2 pa3 pa4 pa5 pa6 pa7 pa8 pa9 pa10 pa11 pa12 pa13 $
      pa14 pa15 pa16 pa17 pa18 pa19 pa20 pa21 pa22 pa23 ram24

; Mux to select the exit count or the time count bits to be driven out

muxeccnt12  l11 l10 l9 l8, j11 k11 j10 k10 j9 k9 j8 k8 beccnt X418W4
muxeccnt16  l15 l14 l13 l12, j15 k15 j14 k14 j13 k13 j12 k12 $
            beccnt X418W4
muxeccnt20  l19 l18 l17 l16, j19 k19 j18 k18 j17 k17 j16 k16 $
            beccnt X418W4
muxeccnt24  l23 l22 l21 l20, j23 k23 j22 k22 j21 k21 j20 k20 $
            beccnt X418W4
muxeccnt28  l27 l26 l25 l24, j27 k27 j26 k26 j25 k25 j24 k24 $
            beccnt X418W4
muxeccnt32  l31 l30 l29 l28, j31 k31 j30 k30 j29 k29 j28 k28 $
            beccnt X418W4

; Muxes to select which bit of the count to be driven out

mux0  l0, g0 l8 l16 l24 nj32 nj40 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux1  l1, g1 l9 l17 l25 nj33 nj41 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux2  l2, g2 l10 l18 l26 nj34 nj42 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux3  l3, g3 l11 l19 l27 nj35 nj43 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux4  l4, g4 l12 l20 l28 nj36 nj44 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux5  l5, g5 l13 l21 l29 nj37 nj45 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux6  l6, g6 l14 l22 l30 nj38 nj46 rd0 rd1 rd2 rd3 rd4 rd5 X456
mux7  l7, g7 l15 l23 l31 nj39 nj47 rd0 rd1 rd2 rd3 rd4 rd5 X456

; Bidirectional pad cells to drive data from read ports out of the
; chip and drive write data form RAM into the chip

iopad0      wr32 c0, nenw l0 nenr iopad
iopad1      wr33 c1, nenw l1 nenr iopad
iopad2      wr34 c2, nenw l2 nenr iopad
iopad3      wr35 c3, nenw l3 nenr iopad
iopad4      wr36 c4, nenw l4 nenr iopad
iopad5      wr37 c5, nenw l5 nenr iopad
iopad6      wr38 c6, nenw l6 nenr iopad
iopad7      wr39 c7, nenw l7 nenr iopad
```

```
; Drive the 40ns clock out to provide synchronization with data

clkpad      pck4, clk4 OPADI

MODEL compare subckt: nodes = ( cntcout.5 exitcout.5 m7 m6 m5 m4 m3 m2 $
      m1 m0, $
      a31 a30 a29 a28 a27 a26 a25 a24 a23 $
      a22 a21 a20 a19 a18 a17 a16 a15 a14 a13 a12 a11 a10 a9 a8 a7 $
      a6 a5 a4 a3 a2 a1 a0 limit nlimit prg_lat strb ck nen_timer $
      valid chip_mode nchip_mode nr rccout recout nenr eccnt )

; Regenerate the reset signal

rstbuf bnr, nr Z101

; Generate upper or lower limit programming signals

andll llimit, prg_lat nlimit strb X403X2
andul ulimit, prg_lat limit  strb X403X2

; The upper and lower limit registers.  The latches are written into
; when the llimit or ulimit signal is high.

reg32ll     ll31 ll30 ll29 ll28 ll27 ll26 ll25 ll24 nll31 $
      nll30 nll29 nll28 nll27 nll26 nll25 nll24, a31 a30 $
      a29 a28 a27 a26 a25 a24 llimit X322W8
reg24ll     ll23 ll22 ll21 ll20 ll19 ll18 ll17 ll16 nll23 $
      nll22 nll21 nll20 nll19 nll18 nll17 nll16, a23 a22 $
      a21 a20 a19 a18 a17 a16 llimit X322W8
reg16ll     ll15 ll14 ll13 ll12 ll11 ll10 ll9 ll8 nll15 nll14 $
      nll13 nll12 nll11 nll10 nll9 nll8, a15 a14 a13 a12 a11 $
      a10 a9 a8 llimit X322W8
reg8ll      ll7 ll6 ll5 ll4 ll3 ll2 ll1 ll0 nll7 nll6 nll5 $
      nll4 nll3 nll2 nll1 nll0, a7 a6 a5 a4 a3 a2 a1 a0 $
      llimit X322W8

reg32ul     ul31 ul30 ul29 ul28 ul27 ul26 ul25 ul24 nul31 $
      nul30 nul29 nul28 nul27 nul26 nul25 nul24, a31 a30 $
      a29 a28 a27 a26 a25 a24 ulimit X322W8
reg24ul     ul23 ul22 ul21 ul20 ul19 ul18 ul17 ul16 nul23 $
      nul22 nul21 nul20 nul19 nul18 nul17 nul16, a23 a22 $
      a21 a20 a19 a18 a17 a16 ulimit X322W8
reg16ul     ul15 ul14 ul13 ul12 ul11 ul10 ul9 ul8 nul15 nul14 $
      nul13 nul12 nul11 nul10 nul9 nul8, a15 a14 a13 a12 a11 $
      a10 a9 a8 ulimit X322W8
reg8ul      ul7 ul6 ul5 ul4 ul3 ul2 ul1 ul0 nul7 nul6 nul5 $
      nul4 nul3 nul2 nul1 nul0, a7 a6 a5 a4 a3 a2 a1 a0 $
```

```
  ulimit X322W8

; Flip-flops to trap the input data bits

reg32b b31.1 b30.1 b29.1 b28.1 nb31.1 nb30.1 nb29.1 nb28.1 $
      ck a31 a30 a29 a28 X351W4
reg28b b27.1 b26.1 b25.1 b24.1 nb27.1 nb26.1 nb25.1 nb24.1 $
       ck a27 a26 a25 a24 X351W4
reg24b b23.1 b22.1 b21.1 b20.1 nb23.1 nb22.1 nb21.1 nb20.1 $
       ck a23 a22 a21 a20 X351W4
reg20b b19.1 b18.1 b17.1 b16.1 nb19.1 nb18.1 nb17.1 nb16.1 $
       ck a19 a18 a17 a16 X351W4
reg16b b15.1 b14.1 b13.1 b12.1 nb15.1 nb14.1 nb13.1 nb12.1 $
       ck a15 a14 a13 a12 X351W4
reg12b b11.1 b10.1 b9.1 b8.1 nb11.1 nb10.1 nb9.1 nb8.1 $
      ck a11 a10 a9 a8 X351W4
reg8b b7.1 b6.1 b5.1 b4.1 nb7.1 nb6.1 nb5.1 nb4.1 $
      ck a7 a6 a5 a4 X351W4
reg4b b3.1 b2.1 b1.1 b0.1 nb3.1 nb2.1 nb1.1 nb0.1 $
      ck a3 a2 a1 a0 X351W4

; Generate the e (a equal b) and g (a gt b) signals for comparison

cmp32a.0 e31a.0 g31a.0 e30a.0 g30a.0 e29a.0 g29a.0 e28a.0 g28a.0, $
       l131 l130 l129 l128 nl131 nl130 nl129 nl128 b31.1 b30.1 b29.1 $
       b28.1 nb31.1 nb30.1 nb29.1 nb28.1 gengteq
cmp32b.0 e31b.0 g31b.0 e30b.0 g30b.0 e29b.0 g29b.0 e28b.0 g28b.0, $
       b31.1 b30.1 b29.1 b28.1 nb31.1 nb30.1 nb29.1 nb28.1 ul31 $
       ul30 ul29 ul28 nul31 nul30 nul29 nul28 gengteq

cmp28a.0 e27a.0 g27a.0 e26a.0 g26a.0 e25a.0 g25a.0 e24a.0 g24a.0, $
       l127 l126 l125 l124 nl127 nl126 nl125 nl124 b27.1 b26.1 b25.1 $
       b24.1 nb27.1 nb26.1 nb25.1 nb24.1 gengteq
cmp28b.0 e27b.0 g27b.0 e26b.0 g26b.0 e25b.0 g25b.0 e24b.0 g24b.0, $
       b27.1 b26.1 b25.1 b24.1 nb27.1 nb26.1 nb25.1 nb24.1 ul27 $
       ul26 ul25 ul24 nul27 nul26 nul25 nul24 gengteq

cmp24a.0 e23a.0 g23a.0 e22a.0 g22a.0 e21a.0 g21a.0 e20a.0 g20a.0, $
       l123 l122 l121 l120 nl123 nl122 nl121 nl120 b23.1 b22.1 b21.1 $
       b20.1 nb23.1 nb22.1 nb21.1 nb20.1 gengteq
cmp24b.0 e23b.0 g23b.0 e22b.0 g22b.0 e21b.0 g21b.0 e20b.0 g20b.0, $
       b23.1 b22.1 b21.1 b20.1 nb23.1 nb22.1 nb21.1 nb20.1 ul23 $
       ul22 ul21 ul20 nul23 nul22 nul21 nul20 gengteq

cmp20a.0 e19a.0 g19a.0 e18a.0 g18a.0 e17a.0 g17a.0 e16a.0 g16a.0, $
       l119 l118 l117 l116 nl119 nl118 nl117 nl116 b19.1 b18.1 b17.1 $
       b16.1 nb19.1 nb18.1 nb17.1 nb16.1 gengteq
```

```
cmp20b.0 e19b.0 g19b.0 e18b.0 g18b.0 e17b.0 g17b.0 e16b.0 g16b.0, $
       b19.1 b18.1 b17.1 b16.1 nb19.1 nb18.1 nb17.1 nb16.1 ul19 $
       ul18 ul17 ul16 nul19 nul18 nul17 nul16 gengteq

cmp16a.0 e15a.0 g15a.0 e14a.0 g14a.0 e13a.0 g13a.0 e12a.0 g12a.0, $
       l115 l114 l113 l112 nl115 nl114 nl113 nl112 b15.1 b14.1 b13.1 $
       b12.1 nb15.1 nb14.1 nb13.1 nb12.1 gengteq
cmp16b.0 e15b.0 g15b.0 e14b.0 g14b.0 e13b.0 g13b.0 e12b.0 g12b.0, $
       b15.1 b14.1 b13.1 b12.1 nb15.1 nb14.1 nb13.1 nb12.1 ul15 $
       ul14 ul13 ul12 nul15 nul14 nul13 nul12 gengteq

cmp12a.0 e11a.0 g11a.0 e10a.0 g10a.0 e9a.0 g9a.0 e8a.0 g8a.0, l111 $
       l110 l19 l18 nl111 nl110 nl19 nl18 b11.1 b10.1 b9.1 $
       b8.1 nb11.1 nb10.1 nb9.1 nb8.1 gengteq
cmp12b.0 e11b.0 g11b.0 e10b.0 g10b.0 e9b.0 g9b.0 e8b.0 g8b.0, b11.1 $
       b10.1 b9.1 b8.1 nb11.1 nb10.1 nb9.1 nb8.1 ul11 ul10 ul9 $
       ul8 nul11 nul10 nul9 nul8 gengteq

cmp8a.0      e7a.0 g7a.0 e6a.0 g6a.0 e5a.0 g5a.0 e4a.0 g4a.0, l17 l16 $
       l15 l14 nl17 nl16 nl15 nl14 b7.1 b6.1 b5.1 b4.1 nb7.1 $
       nb6.1 nb5.1 nb4.1 gengteq
cmp8b.0      e7b.0 g7b.0 e6b.0 g6b.0 e5b.0 g5b.0 e4b.0 g4b.0, b7.1 b6.1 $
       b5.1 b4.1 nb7.1 nb6.1 nb5.1 nb4.1 ul7 ul6 ul5 ul4 nul7 $
       nul6 nul5 nul4 gengteq

cmp4a.0      e3a.0 g3a.0 e2a.0 g2a.0 e1a.0 g1a.0 e0a.0 g0a.0, l13 l12 $
       l11 l10 nl13 nl12 nl11 nl10 b3.1 b2.1 b1.1 b0.1 nb3.1 $
       nb2.1 nb1.1 nb0.1 gengteq
cmp4b.0      e3b.0 g3b.0 e2b.0 g2b.0 e1b.0 g1b.0 e0b.0 g0b.0, b3.1 b2.1 $
       b1.1 b0.1 nb3.1 nb2.1 nb1.1 nb0.1 ul3 ul2 ul1 ul0 nul3 $
       nul2 nul1 nul0 gengteq

; Feed the e and g signals to the 4 bit comparator (gtgen) cell
; and generate a less than b (aleb) signals.  Also generate an
; a equal b (aeqb) signal for each four bits by ANDing the e signals
; for the individual bits

gteq32a.0   aleb32.0 aeqb32.0, g31a.0 e31a.0 g30a.0 e30a.0 $
      g29a.0 e29a.0 g28a.0 e28a.0 gteq
gteq32b.0   blec32.0 beqc32.0, g31b.0 e31b.0 g30b.0 e30b.0 $
      g29b.0 e29b.0 g28b.0 e28b.0 gteq

gteq28a.0   aleb28.0 aeqb28.0, g27a.0 e27a.0 g26a.0 e26a.0 $
      g25a.0 e25a.0 g24a.0 e24a.0 gteq
gteq28b.0   blec28.0 beqc28.0, g27b.0 e27b.0 g26b.0 e26b.0 $
      g25b.0 e25b.0 g24b.0 e24b.0 gteq
```

```
gteq24a.0   aleb24.0 aeqb24.0, g23a.0, e23a.0 g22a.0 e22a.0 $
      g21a.0 e21a.0 g20a.0 e20a.0 gteq
gteq24b.0   blec24.0 beqc24.0, g23b.0 e23b.0 g22b.0 e22b.0 $
      g21b.0 e21b.0 g20b.0 e20b.0 gteq

gteq20a.0   aleb20.0 aeqb20.0, g19a.0 e19a.0 g18a.0 e18a.0 $
      g17a.0 e17a.0 g16a.0 e16a.0 gteq
gteq20b.0   blec20.0 beqc20.0, g19b.0 e19b.0 g18b.0 e18b.0 $
      g17b.0 e17b.0 g16b.0 e16b.0 gteq

gteq16a.0   aleb16.0 aeqb16.0, g15a.0 e15a.0 g14a.0 e14a.0 $
      g13a.0 e13a.0 g12a.0 e12a.0 gteq
gteq16b.0   blec16.0 beqc16.0, g15b.0 e15b.0 g14b.0 e14b.0 $
      g13b.0 e13b.0 g12b.0 e12b.0 gteq

gteq12a.0   aleb12.0 aeqb12.0, g11a.0 e11a.0 g10a.0 e10a.0 $
      g9a.0 e9a.0 g8a.0 e8a.0 gteq
gteq12b.0   blec12.0 beqc12.0, g11b.0 e11b.0 g10b.0 e10b.0 $
      g9b.0 e9b.0 g8b.0 e8b.0 gteq

gteq8a.0    aleb8.0 aeqb8.0, g7a.0 e7a.0 g6a.0 e6a.0 $
      g5a.0 e5a.0 g4a.0 e4a.0 gteq
gteq8b.0    blec8.0 beqc8.0, g7b.0 e7b.0 g6b.0 e6b.0 $
      g5b.0 e5b.0 g4b.0 e4b.0 gteq

gteq4a.0    aleb4.0 aeqb4.0, g3a.0 e3a.0 g2a.0 e2a.0 $
      g1a.0 e1a.0 g0a.0 e0a.0 gteq
gteq4b.0    blec4.0 beqc4.0, g3b.0 e3b.0 g2b.0 e2b.0 $
      g1b.0 e1b.0 g0b.0 e0b.0 gteq

; Trap the first stage aleb and aeqb signals in flip-flops

ff32lea     aleb32.1 aleb28.1 aleb24.1 aleb20.1 # # # #, ck aleb32.0 $
      aleb28.0 aleb24.0 aleb20.0 X351W4
ff32leb     blec32.1 blec28.1 blec24.1 blec20.1 # # # #, ck blec32.0 $
      blec28.0 blec24.0 blec20.0 X351W4

ff16lea     aleb16.1 aleb12.1 aleb8.1 aleb4.1 # # # #, ck aleb16.0 $
      aleb12.0 aleb8.0 aleb4.0 X351W4
ff16leb     blec16.1 blec12.1 blec8.1 blec4.1 # # # #, ck blec16.0 $
      blec12.0 blec8.0 blec4.0 X351W4

ff32eqa     aeqb32.1 aeqb28.1 aeqb24.1 aeqb20.1 # # # #, ck aeqb32.0 $
      aeqb28.0 aeqb24.0 aeqb20.0 X351W4
ff32eqb     beqc32.1 beqc28.1 beqc24.1 beqc20.1 # # # #, ck beqc32.0 $
      beqc28.0 beqc24.0 beqc20.0 X351W4
```

```
ff16eqa     aeqb16.1 aeqb12.1 aeqb8.1 aeqb4.1 # # # #, ck aeqb16.0 $
      aeqb12.0 aeqb8.0 aeqb4.0 X351W4
ff16eqb     beqc16.1 beqc12.1 beqc8.1 beqc4.1 # # # #, ck beqc16.0 $
      beqc12.0 beqc8.0 beqc4.0 X351W4

; Feed output of flipflops to second stage gtgen cells.
; Do a 6 bit gtgen in this stage

; The gtgen'ing of first 4 bits for the previous 16 bits of result


gteq32a     aleb32.2 aeqb32.2, aleb32.1 aeqb32.1 aleb28.1 aeqb28.1 $
      aleb24.1 aeqb24.1 aleb20.1 aeqb20.1 gteq
gteq32b     blec32.2 beqc32.2, blec32.1 beqc32.1 blec28.1 beqc28.1 $
      blec24.1 beqc24.1 blec20.1 beqc20.1 gteq

gteq16a     aleb16.2 aeqb16.2, aleb16.1 aeqb16.1 aleb12.1 aeqb12.1 $
      aleb8.1 aeqb8.1 aleb4.1 aeqb4.1 gteq
gteq16b     blec16.2 beqc16.2, blec16.1 beqc16.1 blec12.1 beqc12.1 $
      blec8.1 beqc8.1 blec4.1 beqc4.1 gteq

; Do another gtgen with the previous outputs.  Thus a 6 bit comparison
; is done in this stage

gt32a naleb, aleb16.2 aeqb32.2 aleb32.2 x133
gt32b nblec, blec16.2 beqc32.2 blec32.2 x133

eq32a naeqb, aeqb32.2 aeqb16.2 x162
eq32b nbeqc, beqc32.2 beqc16.2 x162

; Trap second level gtgen output in a ff

ffleeq      naleb.2 nblec.2 naeqb.2 nbeqc.2 # # # #, ck naleb nblec naeqb $
      nbeqc X351W4

; Generate the in-range signal

aoi1  inrange, naleb.2 naeqb.2 nblec.2 nbeqc.2 X124
ao2   active, inrange valid.2 nvalid.2 active.3 Z124
oai3  inc, nen_timer nchip_mode chip_mode nvalid.2 X114
nand1 exitrange1, naeqb.2 inrange X162
and0  exitrange, exitrange1 active.3 valid.2 X403

; Trap active and exitrange signals

ffmisc      exitrange.3 valid.2 active.3 inc.3 nexitrange.3 $
      nvalid.2 nactive.3 ninc.3, ck exitrange valid $
```

```
  active inc bnr X352W4

ffcout      cntcout.4 exitcout.4 cntcout.5 exitcout.5 ncntcout.4 $
      nexitcout.4 ncntcout.5 nexitcout.5, ck cntcout4 exitcout4 $
      cntcout81 exitcout81 bnr X352W4

; Generate increment signal for the time/address count

norcnt      cntcin, nactive.3 ninc.3 X102

; The incrementer and the associated flipflop

ffcnt4      e3 e2 e1 e0 ne3 ne2 ne1 ne0, ck d3 d2 d1 d0 bnr X352W4
incr4 d3 d2 d1 d0 cntcout4, e3 e2 e1 e0 cntcin X137F

ffcnt8      e7 e6 e5 e4 ne7 ne6 ne5 ne4, ck d7 d6 d5 d4 bnr X352W4
incr8 d7 d6 d5 d4 cntcout8, e7 e6 e5 e4 cntcout.4 X137F

norcnt2     1, ncntcout.5 rccout X102
or1   cntcout81, 1 cntcout8 X432

; Incrementer for range entry exit

ffrr4 g3 g2 g1 g0 ng3 ng2 ng1 ng0, ck f3 f2 f1 f0 bnr X352W4
incrr4      f3 f2 f1 f0 exitcout4, g3 g2 g1 g0 exitrange.3 X137F

ffrr8 g7 g6 g5 g4 ng7 ng6 ng5 ng4, ck f7 f6 f5 f4 bnr X352W4
incrr8      f7 f6 f5 f4 exitcout8, g7 g6 g5 g4 exitcout.4 X137F

norext2 2, nexitcout.5 recout X102
or2   exitcout81, 2 exitcout8 X432

; Mux the entry-exit and range count bits

muxeccnt4 l3 l2 l1 l0, ne3 ng3 ne2 ng2 ne1 ng1 ne0 ng0 eccnt X418W4
muxeccnt8 l7 l6 l5 l4, ne7 ng7 ne6 ng6 ne5 ng5 ne4 ng4 eccnt X418W4

; Drive the increment and range entry exit count to tristate buffers

bufl7 m7, nenr l7 C519
bufl6 m6, nenr l6 C519
bufl5 m5, nenr l5 C519
bufl4 m4, nenr l4 C519
bufl3 m3, nenr l3 C519
bufl2 m2, nenr l2 C519
bufl1 m1, nenr l1 C519
bufl0 m0, nenr l0 C519
```

```
ENDM compare

MODEL gengteq subckt: nodes = (e3 g3 e2 g2 e1 g1 e0 g0, a3 a2 a1 a0 $
                  na3 na2 na1 na0 b3 b2 b1 b0 nb3 nb2 nb1 nb0 )

XNOR3 e3, na3 nb3       X421

XNOR2 e2, na2 nb2       X421

XNOR1 e1, na1 nb1       X421

XNOR0 e0, na0 nb0       X421

NO3   g3, a3 nb3        X102

NO2   g2, a2 nb2        X102

NO1   g1, a1 nb1        X102

NO0   g0, a0 nb0        X102

ENDM gengteq

model gteq subckt: nodes = (aleb aeqb, g3 e3 g2 e2 g1 e1 g0 e0)

gtgen aleb, g3 e3 g2 e2 g1 e1 g0    gtgen4

Aeqout  aeqb, e3 e2 e1 e0     X404

ENDM gteq

model dly6x subckt: nodes = (out, in)

dly2x 1, in DLY2X
dly4x out,1 DLY4X

ENDM dly6x

model ram40 subckt: nodes = ( f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 $
      f12 f13 f14 f15 f16 f17 f18 f19 f20 f21 f22 f23 nf24 nf25 nf26 $
      nf27 nf28 nf29 nf30 nf31 nf32 nf33 nf34 nf35 nf36 nf37 nf38 $
      nf39, enr enw clk4 nclk4 enw2 $
      b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 $
      nb0 nb1 nb2 nb3 nb4 nb5 nb6 nb7 nb8 nb9 nb10 nb11 nb12 nb13 $
      nb14 nb15 $
      q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 q10 q11 q12 q13 $
      q14 q15 $
```

```
  nn0 nn1 nn2 nn3 nn4 nn5 nn6 nn7 nn8 nn9 nn10 nn11 nn12 nn13 $
      nn14 nn15 e0 e1 e2 e3 e4 e5 e6 e7 $
      wr0 wr1 wr2 wr3 wr4 wr5 wr6 wr7 wr8 wr9 wr10 wr11 wr12 wr13 $
      wr14 wr15 wr16 wr17 wr18 wr19 wr20 wr21 wr22 wr23 wr24 wr25 $
      wr26 wr27 wr28 wr29 wr30 wr31 wr32 wr33 wr34 wr35 wr36 wr37 $
      wr38 wr39 )

; the priority resolver

resolv      nq1 nq2 nq3 nq4 nq5 nq6 nq7 nq8 nq9 $
      nq10 nq11 nq12 nq13 nq14 nq15 b0 b1 b2 b3 b4 b5 b6 b7 b8 $
      b9 b10 b11 b12 b13 b14 b15 nb0 nb1 nb2 nb3 nb4 nb5 nb6 nb7 nb8 $
      nb9 nb10 nb11 nb12 nb13 nb14 nb15 pri_resolv

; output of priority resolver trapped in flipflops with rising edge
; of the clock to take care of the 20ns delay thru the resolv circuit

rff4  nd3 nd2 nd1 nd0 q3 q2 q1 q0, nclk4 nq3 nq2 nq1 nb0 X351W4
rff8  nd7 nd6 nd5 nd4 q7 q6 q5 q4, nclk4 nq7 nq6 nq5 nq4 X351W4
rff12 nd11 nd10 nd9 nd8 q11 q10 q9 q8, nclk4 nq11 nq10 nq9 nq8 X351W4
rff16 nd15 nd14 nd13 nd12 q15 q14 q13 q12, nclk4 nq15 nq14 nq13 nq12 X351W4

; Mux to drive output of priority resolver or decoder to RAM read ports

muxr4 d3 d2 d1 d0, nd3 nn3 nd2 nn2 nd1 nn1 nd0 nn0 enr X418W4
muxr8 d7 d6 d5 d4, nd7 nn7 nd6 nn6 nd5 nn5 nd4 nn4 enr X418W4
muxr12      d11 d10 d9 d8, nd11 nn11 nd10 nn10 nd9 nn9 nd8 nn8 enr X418W4
muxr16      d15 d14 d13 d12, nd15 nn15 nd14 nn14 nd13 nn13 nd12 nn12 enr $
      X418W4

; Drive the mux output to the RAM using buffers

bufrd0      rd0, d0 Z101X4
bufrd1      rd1, d1 Z101X4
bufrd2      rd2, d2 Z101X4
bufrd3      rd3, d3 Z101X4

bufrd4      rd4, d4 Z101X4
bufrd5      rd5, d5 Z101X4
bufrd6      rd6, d6 Z101X4
bufrd7      rd7, d7 Z101X4

bufrd8      rd8, d8 Z101X4
bufrd9      rd9, d9 Z101X4
bufrd10     rd10, d10 Z101X4
bufrd11     rd11, d11 Z101X4
```

```
bufrd12     rd12, d12 Z101X4
bufrd13     rd13, d13 Z101X4
bufrd14     rd14, d14 Z101X4
bufrd15     rd15, d15 Z101X4

; The RAM

bit0  dout0 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din0 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit1  dout1 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din1 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit2  dout2 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din2 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit3  dout3 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din3 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit4  dout4 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din4 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit5  dout5 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din5 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit6  dout6 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din6 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit7  dout7 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din7 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit8  dout8 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din8 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit9  dout9 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din9 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit10  dout10 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din10 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit11  dout11 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din11 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit12  dout12 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din12 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit13  dout13 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
```

```
  rd13 rd14 rd15 din13 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit14  dout14 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din14 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit15  dout15 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din15 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit16  dout16 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din16 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit17  dout17 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din17 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit18  dout18 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din18 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit19  dout19 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din19 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit20  dout20 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din20 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit21  dout21 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din21 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit22  dout22 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din22 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit23  dout23 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din23 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit24  dout24 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din24 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit25  dout25 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din25 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit26  dout26 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din26 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit27  dout27 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din27 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit28  dout28 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din28 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
```

```
bit29  dout29 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din29 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit30  dout30 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din30 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit31  dout31 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din31 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit32  dout32 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din32 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit33  dout33 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din33 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit34  dout34 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din34 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit35  dout35 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din35 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit36  dout36 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din36 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit37  dout37 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din37 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit38  dout38 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din38 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit39  dout39 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
       rd13 rd14 rd15 din39 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
       wd10 wd11 wd12 wd13 wd14 wd15 C533F

; Delayed clock4 from clk4

delay1 ck4d, clk4 DLY2X

; 16 input OR gate for trapping the signal for carry-in to be fed to the
; incrementer.  Set the signal to low when the enr signal is high i.e. the
; RAM is being read.

nand1 1, nd0 nd1 nd2 nd3 nd4 nd5 nd6 nd7 X168
nand2 2, nd8 nd9 nd10 nd11 nd12 nd13 nd14 nd15 X168
or1  3, 1 2 X432
ffcin cin ncin, clk4 LO 3 enr X381
```

```
; The ff to load RAM data

ramff5      f4 f3 f2 f1 f0 # # # # #, clk4 dout4 dout3 dout2 dout1 dout0 $
      X351W5
ramff10     f9 f8 f7 f6 f5 # # # # #, clk4 dout9 dout8 dout7 dout6 dout5 $
      X351W5
ramff15     f14 f13 f12 f11 f10 # # # # #, clk4 dout14 dout13 dout12 $
      dout11 dout10 X351W5
ramff20     f19 f18 f17 f16 f15 # # # # #, clk4 dout19 dout18 dout17 $
      dout16 dout15 X351W5
ramff25     f24 f23 f22 f21 f20 nf24 # # # #, clk4 dout24 dout23 dout22 $
      dout21 dout20 X351W5
ramff30     f29 f28 f27 f26 f25 nf29 nf28 nf27 nf26 nf25, clk4 dout29 $
      dout28 dout27 dout26 dout25 X351W5
ramff35     f34 f33 f32 f31 f30 nf34 nf33 nf32 nf31 nf30, clk4 dout34 $
      dout33 dout32 dout31 dout30 X351W5
ramff40     f39 f38 f37 f36 f35 nf39 nf38 nf37 nf36 nf35, clk4 dout39 $
      dout38 dout37 dout36 dout35 X351W5

; The incrementer

inc4  g3 g2 g1 g0 co4, f3 f2 f1 f0 cin X137F
inc8  g7 g6 g5 g4 co8, f7 f6 f5 f4 co4 X137F
inc12 g11 g10 g9 g8 co12, f11 f10 f9 f8 co8 X137F
inc16 g15 g14 g13 g12 co16, f15 f14 f13 f12 co12 X137F
inc20 g19 g18 g17 g16 co20, f19 f18 f17 f16 co16 X137F
inc24 g23 g22 g21 g20 co24, f23 f22 f21 f20 co20 X137F
inc28 g27 g26 g25 g24 co28, f27 f26 f25 f24 co24 X137F
inc32 g31 g30 g29 g28 co32, f31 f30 f29 f28 co28 X137F
inc36 g35 g34 g33 g32 co36, f35 f34 f33 f32 co32 X137F
inc40 g39 g38 g37 g36 #, f39 f38 f37 f36 co36 X137F

; Trap output of incrementer

incff5      # # # # # nh4 nh3 nh2 nh1 nh0, clk4 g4 g3 g2 g1 g0 X351W5
incff10     # # # # # nh9 nh8 nh7 nh6 nh5, clk4 g9 g8 g7 g6 g5 X351W5
incff15     # # # # # nh14 nh13 nh12 nh11 nh10, clk4 g14 g13 g12 g11 g10 $
      X351W5
incff20     # # # # # nh19 nh18 nh17 nh16 nh15, clk4 g19 g18 g17 g16 g15 $
      X351W5
incff25     # # # # # nh24 nh23 nh22 nh21 nh20, clk4 g24 g23 g22 g21 g20 $
      X351W5
incff30     # # # # # nh29 nh28 nh27 nh26 nh25, clk4 g29 g28 g27 g26 g25 $
      X351W5
incff35     # # # # # nh34 nh33 nh32 nh31 nh30, clk4 g34 g33 g32 g31 g30 $
      X351W5
incff40     # # # # # nh39 nh38 nh37 nh36 nh35, clk4 g39 g38 g37 g36 g35 $
```

X351W5

```
; Mux to feed either output of flipflops or data from outside into the
; RAM.  Writing into RAM from outside used for testing

mux4  din3 din2 din1 din0, nh3 wr3 nh2 wr2 nh1 wr1 nh0 wr0 enw $
      X418W4
mux8  din7 din6 din5 din4, nh7 wr7 nh6 wr6 nh5 wr5 nh4 wr4 enw $
      X418W4
mux12 din11 din10 din9 din8, nh11 wr11 nh10 wr10 nh9 wr9 nh8 wr8 $
      enw X418W4
mux16 din15 din14 din13 din12, nh15 wr15 nh14 wr14 nh13 wr13 nh12 $
      wr12 enw X418W4
mux20 din19 din18 din17 din16, nh19 wr19 nh18 wr18 nh17 wr17 nh16 $
      wr16 enw X418W4
mux24 din23 din22 din21 din20, nh23 wr23 nh22 wr22 nh21 wr21 nh20 $
      wr20 enw X418W4
mux28 din27 din26 din25 din24, nh27 wr27 nh26 wr26 nh25 wr25 nh24 $
      wr24 enw X418W4
mux32 din31 din30 din29 din28, nh31 wr31 nh30 wr30 nh29 wr29 nh28 $
      wr28 enw X418W4
mux36 din35 din34 din33 din32, nh35 wr35 nh34 wr34 nh33 wr33 nh32 $
      wr32 enw X418W4
mux40 din39 din38 din37 din36, nh39 wr39 nh38 wr38 nh37 wr37 nh36 $
      wr36 enw X418W4

; Mux to drive output of priority resolver or decoder to RAM read ports

muxw4 w3 w2 w1 w0, p3 nn3 p2 nn2 p1 nn1 p0 nn0 enw2 X418W4
muxw8 w7 w6 w5 w4, p7 nn7 p6 nn6 p5 nn5 p4 nn4 enw2 X418W4
muxw12      w11 w10 w9 w8, p11 nn11 p10 nn10 p9 nn9 p8 nn8 enw2 X418W4
muxw16      w15 w14 w13 w12, p15 nn15 p14 nn14 p13 nn13 p12 nn12 enw2 X418W4

; Flip-flops to capture the data for the write ports of the RAM
; Use the delayed clock for these flip-flops so that the write address is
; clocked in some time after the read address is clocked in.  Thus, the
; output of the incrementer can be written into the previous RAM location
; while new data can be applied at the read port of the RAM

wff4  p3 p2 p1 p0 # # # #, ck4d nd3 nd2 nd1 nd0 X351W4
wff8  p7 p6 p5 p4 # # # #, ck4d nd7 nd6 nd5 nd4 X351W4
wff12 p11 p10 p9 p8 # # # #, ck4d nd11 nd10 nd9 nd8 X351W4
wff16 p15 p14 p13 p12 # # # #, ck4d nd15 nd14 nd13 nd12 X351W4

; Drive the mux output to the RAM write ports using buffers
```

```
bufwd0      wd0, w0 Z101X2
bufwd1      wd1, w1 Z101X2
bufwd2      wd2, w2 Z101X2
bufwd3      wd3, w3 Z101X2

bufwd4      wd4, w4 Z101X2
bufwd5      wd5, w5 Z101X2
bufwd6      wd6, w6 Z101X2
bufwd7      wd7, w7 Z101X2

bufwd8      wd8, w8 Z101X2
bufwd9      wd9, w9 Z101X2
bufwd10     wd10, w10 Z101X2
bufwd11     wd11, w11 Z101X2

bufwd12     wd12, w12 Z101X2
bufwd13     wd13, w13 Z101X2
bufwd14     wd14, w14 Z101X2
bufwd15     wd15, w15 Z101X2

endm ram40

model ram24 subckt: nodes = ( f0 f1 f2 f3 f4 f5 f6 f7 f8 $
      f9 f10 f11 f12 f13 f14 f15 f16 f17 f18 f19 f20 f21 f22 f23, $
      enr enw clk4 nclk4 enw2 $
      b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 $
      nb0 nb1 nb2 nb3 nb4 nb5 nb6 nb7 nb8 nb9 nb10 nb11 nb12 nb13 $
      nb14 nb15 $
      q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 q10 q11 q12 q13 $
      q14 q15 $
      nn0 nn1 nn2 nn3 nn4 nn5 nn6 nn7 nn8 nn9 nn10 nn11 nn12 nn13 $
      nn14 nn15 e0 e1 e2 e3 e4 e5 e6 e7 $
      wr0 wr1 wr2 wr3 wr4 wr5 wr6 wr7 wr8 wr9 wr10 wr11 wr12 wr13 $
      wr14 wr15 wr16 wr17 wr18 wr19 wr20 wr21 wr22 wr23 )

; the priority resolver

resolv      nq1 nq2 nq3 nq4 nq5 nq6 nq7 nq8 nq9 $
      nq10 nq11 nq12 nq13 nq14 nq15 b0 b1 b2 b3 b4 b5 b6 b7 b8 $
      b9 b10 b11 b12 b13 b14 b15 nb0 nb1 nb2 nb3 nb4 nb5 nb6 nb7 nb8 $
      nb9 nb10 nb11 nb12 nb13 nb14 nb15 pri_resolv

; output of priority resolver trapped in flipflops with rising edge
; of the clock to take care of the 20ns delay thru the priority resolv circuit

rff4  nd3 nd2 nd1 nd0 q3 q2 q1 q0, nclk4 nq3 nq2 nq1 nb0 X351W4
rff8  nd7 nd6 nd5 nd4 q7 q6 q5 q4, nclk4 nq7 nq6 nq5 nq4 X351W4
```

```
rff12 nd11 nd10 nd9 nd8 q11 q10 q9 q8, nclk4 nq11 nq10 nq9 nq8 X351W4
rff16 nd15 nd14 nd13 nd12 q15 q14 q13 q12, nclk4 nq15 nq14 nq13 nq12 X351W4

; Mux to drive output of priority resolver or decoder to RAM read ports

muxr4 d3 d2 d1 d0, nd3 nn3 nd2 nn2 nd1 nn1 nd0 nn0 enr X418W4
muxr8 d7 d6 d5 d4, nd7 nn7 nd6 nn6 nd5 nn5 nd4 nn4 enr X418W4
muxr12      d11 d10 d9 d8, nd11 nn11 nd10 nn10 nd9 nn9 nd8 nn8 enr X418W4
muxr16      d15 d14 d13 d12, nd15 nn15 nd14 nn14 nd13 nn13 nd12 nn12 enr $
      X418W4

; Drive the mux output to the RAM using buffers

bufrd0      rd0, d0 Z101X4
bufrd1      rd1, d1 Z101X4
bufrd2      rd2, d2 Z101X4
bufrd3      rd3, d3 Z101X4

bufrd4      rd4, d4 Z101X4
bufrd5      rd5, d5 Z101X4
bufrd6      rd6, d6 Z101X4
bufrd7      rd7, d7 Z101X4

bufrd8      rd8, d8 Z101X4
bufrd9      rd9, d9 Z101X4
bufrd10     rd10, d10 Z101X4
bufrd11     rd11, d11 Z101X4

bufrd12     rd12, d12 Z101X4
bufrd13     rd13, d13 Z101X4
bufrd14     rd14, d14 Z101X4
bufrd15     rd15, d15 Z101X4

; The RAM

bit0  dout0 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din0 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit1  dout1 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din1 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit2  dout2 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din2 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit3  dout3 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din3 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
```

```
bit4  dout4 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din4 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit5  dout5 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din5 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit6  dout6 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din6 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit7  dout7 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din7 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit8  dout8 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din8 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit9  dout9 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din9 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit10  dout10 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din10 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit11  dout11 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din11 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit12  dout12 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din12 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit13  dout13 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din13 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit14  dout14 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din14 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit15  dout15 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din15 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit16  dout16 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din16 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit17  dout17 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din17 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit18  dout18 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din18 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit19  dout19 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din19 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
```

```
  wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit20  dout20 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din20 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit21  dout21 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din21 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit22  dout22 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din22 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F
bit23  dout23 rd0 rd1 rd2 rd3 rd4 rd5 rd6 rd7 rd8 rd9 rd10 rd11 rd12 $
      rd13 rd14 rd15 din23 wd0 wd1 wd2 wd3 wd4 wd5 wd6 wd7 wd8 wd9 $
      wd10 wd11 wd12 wd13 wd14 wd15 C533F

; Delayed clock4 from clk4

delay1 ck4d, clk4 DLY2X

; 16 input OR gate for trapping the signal for carry-in

nand1 1, nd0 nd1 nd2 nd3 nd4 nd5 nd6 nd7 X168
nand2 2, nd8 nd9 nd10 nd11 nd12 nd13 nd14 nd15 X168
or1  3, 1 2 X432
ffcin cin ncin, clk4 LO 3 enr X381

; The ff to load RAM data

ramff5      f4 f3 f2 f1 f0 # # # # #, clk4 dout4 dout3 dout2 dout1 dout0 $
      X351W5
ramff10     f9 f8 f7 f6 f5 # # # # #, clk4 dout9 dout8 dout7 dout6 dout5 $
      X351W5
ramff15     f14 f13 f12 f11 f10 # # # # #, clk4 dout14 dout13 dout12 $
      dout11 dout10 X351W5
ramff20     f19 f18 f17 f16 f15 # # # # #, clk4 dout19 dout18 dout17 $
      dout16 dout15 X351W5
ramff24     f23 f22 f21 f20 # # # #, clk4 dout23 dout22 dout21 dout20 $
      X351W4

; The incrementer

inc4  g3 g2 g1 g0 co4, f3 f2 f1 f0 cin X137F
inc8  g7 g6 g5 g4 co8, f7 f6 f5 f4 co4 X137F
inc12 g11 g10 g9 g8 co12, f11 f10 f9 f8 co8 X137F
inc16 g15 g14 g13 g12 co16, f15 f14 f13 f12 co12 X137F
inc20 g19 g18 g17 g16 co20, f19 f18 f17 f16 co16 X137F
inc24 g23 g22 g21 g20 #, f23 f22 f21 f20 co20 X137F
```

```
; Trap output of incrementer

incff5      # # # # # nh4 nh3 nh2 nh1 nh0, clk4 g4 g3 g2 g1 g0 X351W5
incff10     # # # # # nh9 nh8 nh7 nh6 nh5, clk4 g9 g8 g7 g6 g5 X351W5
incff15     # # # # # nh14 nh13 nh12 nh11 nh10, clk4 g14 g13 g12 g11 g10 $
      X351W5
incff20     # # # # # nh19 nh18 nh17 nh16 nh15, clk4 g19 g18 g17 g16 g15 $
      X351W5
incff24     # # # # nh23 nh22 nh21 nh20, clk4 g23 g22 g21 g20 X351W4

; Mux to feed either output of flipflops or data from outside into the
; RAM.  Writing into RAM from outside used for testing

mux4  din3 din2 din1 din0, nh3 wr3 nh2 wr2 nh1 wr1 nh0 wr0 enw $
      X418W4
mux8  din7 din6 din5 din4, nh7 wr7 nh6 wr6 nh5 wr5 nh4 wr4 enw $
      X418W4
mux12 din11 din10 din9 din8, nh11 wr11 nh10 wr10 nh9 wr9 nh8 wr8 $
      enw X418W4
mux16 din15 din14 din13 din12, nh15 wr15 nh14 wr14 nh13 wr13 nh12 $
      wr12 enw X418W4
mux20 din19 din18 din17 din16, nh19 wr19 nh18 wr18 nh17 wr17 nh16 $
      wr16 enw X418W4
mux24 din23 din22 din21 din20, nh23 wr23 nh22 wr22 nh21 wr21 nh20 $
      wr20 enw X418W4

; Mux to drive output of priority resolver or decoder to RAM read ports

muxw4 w3 w2 w1 w0, p3 nn3 p2 nn2 p1 nn1 p0 nn0 enw2 X418W4
muxw8 w7 w6 w5 w4, p7 nn7 p6 nn6 p5 nn5 p4 nn4 enw2 X418W4
muxw12      w11 w10 w9 w8, p11 nn11 p10 nn10 p9 nn9 p8 nn8 enw2 X418W4
muxw16      w15 w14 w13 w12, p15 nn15 p14 nn14 p13 nn13 p12 nn12 enw2 X418W4

; Flip-flops to capture the data for the write ports of the RAM

wff4  p3 p2 p1 p0 # # # #, ck4d nd3 nd2 nd1 nd0 X351W4
wff8  p7 p6 p5 p4 # # # #, ck4d nd7 nd6 nd5 nd4 X351W4
wff12 p11 p10 p9 p8 # # # #, ck4d nd11 nd10 nd9 nd8 X351W4
wff16 p15 p14 p13 p12 # # # #, ck4d nd15 nd14 nd13 nd12 X351W4

; Drive the mux output to the RAM write ports using buffers

bufwd0      wd0, w0 Z101X2
bufwd1      wd1, w1 Z101X2
bufwd2      wd2, w2 Z101X2
bufwd3      wd3, w3 Z101X2
```

```
bufwd4      wd4, w4 Z101X2
bufwd5      wd5, w5 Z101X2
bufwd6      wd6, w6 Z101X2
bufwd7      wd7, w7 Z101X2

bufwd8      wd8, w8 Z101X2
bufwd9      wd9, w9 Z101X2
bufwd10     wd10, w10 Z101X2
bufwd11     wd11, w11 Z101X2

bufwd12     wd12, w12 Z101X2
bufwd13     wd13, w13 Z101X2
bufwd14     wd14, w14 Z101X2
bufwd15     wd15, w15 Z101X2

endm ram24

model pri_resolv subckt: nodes = (nq1 nq2 nq3 nq4 nq5 nq6 nq7 nq8 nq9 $
      nq10 nq11 nq12 nq13 nq14 nq15 a0 a1 a2 a3 a4 a5 a6 a7 a8 $
      a9 a10 a11 a12 a13 a14 a15 na0 na1 na2 na3 na4 na5 na6 na7 na8 $
      na9 na10 na11 na12 na13 na14 na15 )

nand1 nq1,  a1 na0                  X162
nand2 nq2,  a2 na1 na0        X163
nand3 nq3,  a3 na2 na1 na0          X164
and4  1,    na3 na2 na1 na0         X404

nand4 nq4,  a4 1              X162
nand5 nq5,  a5 na4 1                X163
nand6 nq6,  a6 na5 na4 1            X164
and7  2, na6 na5 na4 1        X404

nand7 nq7, a7 2               X162
nand8 nq8, a8 na7 2                 X163
nand9 nq9,  a9 na8 na7 2            X164

and108      9, na9 na8 na7 na6 na5 na4 na3 na2  X408
and103  7, na1 na0 9                X403X2

nand10      nq10, a10 7             X162
nand11      nq11, a11 na10 7        X163
nand12      nq12, a12 na11 na10 7         X164
and12 8, na12 na11 na10 7           X404

nand13      nq13, a13 8             X162
nand14      nq14, a14 na13 8        X163
nand15      nq15, a15 na14 na13 8         X164
```

```
endm pri_resolv

model iopad subckt : nodes = (in out, nenin dout nenout)

outpad       out, dout nenout TSPADT
inttlbuf tin, out C501
inzbuf       in, nenin tin C519

endm iopad

read ~fastsim/lib/cmos15lb
ENDC
```

APPENDIX D

```
#MODE = 2

#NO_HEADER

#DEFINE_INPUTS

      ck      = 1 S
      prg_chip = 2 S
      limit   = 3 S
      strb    = 4 S
      valid   = 5 S
      en_timer = 6 S
      nr      = 7 S
      chip_mode = 8 S
      a(31:0)       = 9:40 S
      d(3:0)        = 41:44 S
      m(3:0)        = 45:48 S
      enr(2:0) = 49:51 S
      c(7:0)        = 52:59 S
      eccnt   = 60 S
      ren     = 61 S
      wen     = 62 S
      strb2   = 63

#END

ck = 10 @ 1
strb2 = 0
ren = 0
wen = 0
eccnt = 0
enr(2:0) = 0
m(3:0) = 0
c(7:0) = z
prg_chip = 0
limit=0
strb=0
valid = 0
```

```
en_timer = 1
chip_mode=1
nr = 1
run 3
nr = 0
run 20

prg_chip = 1
loop i=0,FH {
    a(31:0) = FFFFFFF2H
    d(3:0)=i
    run 1
    strb=1
    run 1
    limit=1
    strb=0
    run 1
    a(31:0) = FFFFFFF8H
    strb=1
    run 1
    strb=0
    limit=0
    run 1
}

prg_chip=1
loop i=0,FH {
    m(3:0) = i
    eccnt=0
    run 2
    strb2=1
    wen=1
    a(31:0)=FFFFFFFFH
    c(7:0)=FFH
    run 1
    strb2=0
}
run 6

loop i=0,FH {
    m(3:0) = i
    eccnt=1
    run 2
    strb2=1
    wen=1
    a(31:0)=FFFFFFFFH
    c(7:0)=FFH
```

```
    run 1
    strb2=0
}
run 6

strb=0
strb2=0
wen=0
eccnt=0
c(7:0)=z
limit=0
prg_chip=0
run 1

ck = 10 @ 1
nr = 0
prg_chip=0
limit=0
valid=1
a(31:0) = FFFFFFF9H
run 1
a(31:0) = FFFFFFF4H
run 1
valid=0
a(31:0) = FFFFFFF1H
run 1
en_timer=0
valid=0
a(31:0) = FFFFFFF5H
run 1
valid=1
run 1
en_timer=1
run 1
a(31:0) = FFFFFFFAH
run 1
a(31:0) = FFFFFFF6H
run 300
a(31:0) = FFFFFFFBH
run 1
loop i=0,300D {
      a(31:0) = FFFFFFF7H
      run 1
      a(31:0) = FFFFFFFCH
      run 1
}
a(31:0) = FFFFFFF8H
```

```
run 1
a(31:0) = FFFFFFF2H
run 1
a(31:0) = FFFFFFF8H
run 1
a(31:0) = FFFFFFFDH
run 1
a(31:0) = FFFFFFF2H
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFFEH
run 1
a(31:0) = FFFFFFF4H
run 1
a(31:0) = FFFFFFFFH
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFF0H
run 1
a(31:0) = FFFFFFFAH
run 1
a(31:0) = FFFFFFFAH
run 1
a(31:0) = FFFFFFFAH
run 1
a(31:0) = FFFFFFF0H
run 8
prg_chip=1
eccnt=0
ren=1
run 1
loop j=0,FH {
    m(3:0)=j
    run 8
    loop i=0,5H {
      enr(2:0)=i
      run 2
    }
}
```

```
ren=0
run 5
```

VITA

Mehul S. Dave was born in Indore, Madhya Pradesh in India on June 5, 1965. He received his B.E. (Electrical) degree (first class with honors) from Victoria Jubilee Technical Institute, University of Bombay, Bombay, India in July 1987. In September 1987, he began his studies at Oregon Graduate Institute of Science and Technology. His areas of interest are VLSI design and computer architecture. He joined Intel corporation as a design engineer in April 1990 after finishing his graduate school.