CONCURRENT DETECTION OF TRANSIENT FAULTS IN MICROPROCESSORS

by

Mohammad Ziaullah Khan

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

APPROVED:

Dr. Joseph G. Tront (Chairman)

Dr. James R. Armstrong

Dr. Dong S. Ha

Dr. Ezra A. Brown

Dr. Charles E. Nunnally

May 1989

Blacksburg, Virginia

CONCURRENT DETECTION OF TRANSIENT FAULTS IN MICROPROCESSORS

by

Mohammad Ziaullah Khan

Dr. Joseph G. Tront (Chairman)

Electrical Engineering

(ABSTRACT)

A large number of errors in digital systems are due to the presence of transient faults. This is especially true of microprocessor-based systems working in a radiation environment that experience transient faults due to single event upsets. These upsets cause a temporary change in the state of the system without any permanent damage. Because of their random and non-recurring nature, transient faults are difficult to detect and isolate, hence they become a source of major concern, especially in critical real-time application areas. Concurrent detection of these errors is necessary for real-time operation.

Most existing fault tolerance schemes either use redundancy to mask effects of transient faults or monitor the system for abnormal operations and then perform recovery operation. Although very effective, redundancy schemes incur substantial overhead that makes them unsuitable for small systems. Most monitoring schemes, on the other hand, only detect control flow errors.

A new approach called Concurrent Processor Monitoring for on-line detection of transient faults is proposed that attempts to achieve high error coverage with small error detection latency. The concept of the execution profile of an instruction is defined and is used for detecting control flow and execution errors. To implement this

scheme, a watchdog processor is designed for monitoring operation of the main processor. The effectiveness of this technique is demonstrated through computer simulations.

Dedication

To my parents, and for their love and affection.

Acknowledgements

I am deeply indebted to Dr. Joe Tront, my advisor, for his help, guidance, and patience throughout this endeavor. I sincerely thank Dr. Jim Armstrong for introducing me to digital system modeling and simulation. My thanks also to Drs. Dong Ha, Charles Nunnally and Ezra Brown for serving on my committee. I also wish to thank Bob Lineberry for the endless hours he spent to get the new computer running for this work. This work was initially supported in part by Naval Research Laboratories under contract number FE238311.

I wish to express my deep appreciation for my uncle and aunt, Dr. and Mrs. for their support that has helped me survive this academic marathon called the doctorate program.

I am grateful to my brothers and sisters who through their letters and phone calls have provided the emotional support. And last, but not least, I wish to express my gratitude to my parents for their love and affection. Their faith in my abilities has always been a great source of inspiration for me.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

The cost and flexibility of microprocessors has made them very attractive for use in control applications. These include real-time control systems where very high reliability and availability is required because these systems cannot stop to recover from an error. In addition to high reliability, these systems should also have self repair capability. This is especially true for systems that are used for the guidance and control of satellites. These spacecrafts, once launched, remain unattended for their entire life span and, therefore, must continue to perform satisfactorily in the presence of faults [1-3]. Several systems have been developed which satisfy these requirements by using redundancy schemes [2,4-7]. Since small satellite controllers have very limited power and weight budgets, the use of redundancy techniques is not a desirable option.

The constraints involved in achieving fault tolerance in digital systems are explored in this chapter. The types of faults that may occur in a digital system are enumerated with emphasis on transient faults. A discussion of some general techniques for fault

tolerant design and their limitations in detecting transient faults is presented. An outline of the organization of this dissertation concludes this chapter.

1.1 Faults in Digital Systems

Faults in a computer system are classified by extent, value, and duration. Extent indicates whether errors produced by faults are localized or distributed; value denotes if faults generate fixed or varying erroneous data values; duration refers to whether a fault is permanent or temporary [8]. **Permanent faults** can be due to a variety of reasons including incorrect logic design, manufacturing defects, harsh environmental abuse, worn out parts, total dose of radiation deposit, and software bugs. Permanent faults appear as stuck-at faults, floating faults, or parametric faults and remain in the system until removed by an explicit action such as repair or replacement [9,10]. Most of these faults can be eliminated by taking the necessary precautions in the design, manufacturing and testing phases. Environmental abuse can be minimized by providing proper housing and shielding of components. **Temporary faults** are classified as either intermittent or transient. An **intermittent fault** manifests itself at regular intervals and is due to partially defective components, loose connections, devices with small tolerances, or poor design. A marginally working component functions properly some of the time but not at other times and is a typical example of an intermittent fault [11]. A **transient fault**, on the other hand, is a random, non-recurring fault which is present for a short interval of time. It may be caused by cosmic radiations, protons, alpha particles, other high energy nuclei, or glitches in the power supply [11-20]. [11.-20.]. Since a transient fault disappears after the interference is removed and

leaves no permanent damage to the device, there is no faulty hardware to repair or replace. That is why transient faults are also referred to as **soft faults.**

1.2 Single Event Upset Phenomenon

Anomalous behavior in digital circuits of communication satellites has been attributed to transient faults. This behavior is characterized by changes in the normal operation of the system without an explicit command such as modification of data stored in the memory, improper activation of certain control signals, or improper execution of a sequence of instructions in a program. Research has indicated that space-borne digital systems are susceptible to transient faults due to the radiation environment in which they operate [13-20]. A transient fault caused by radiation can produce a change in the state of a flip flop. This phenomenon is called a **single event upset (SEU).** For instance, a change in the state of a flip flop occurs when a charged particle of sufficient energy is incident on the base emitter junction of a transistor. The charge on junction capacitance of the sensitive transistor is increased until the turn-on voltage is reached [13]. The amount of charge needed to turn-on the device is called the critical charge and is a function of the capacitance of the sensitive node, the dimensions of the device, and the technology. This phenomenon has been studied experimentally by bombarding integrated circuits with several types of high energy particles including protons, alpha particles, carbon (C-12), and iron (Fe-56) nuclei [21-26]. Typical sources of charged particles include cosmic radiations, other charged particles in the environment, and the naturally occurring alpha particles in packaging materials of semiconductor devices. The capacitance of a device node decreases with decreasing dimensions thus lowering the upset threshold of the de-

vice [3,13]. Considering the ever decreasing feature sizes in VLSI circuits, transient faults will remain a problem in digital systems that operate in harsh environments.

SEUs were first discovered in flip flops and, therefore, most of the literature on the subject relates to the study of storage elements. But SEUs also affect combinational logic. Typically, a digital system consists of several stages of combinational logic followed by a storage latch. An SEU in combinational logic produces a voltage transient which propagates through the circuit. For this pulse to produce an error, the following criteria must be satisfied [27-29]:

1. the pulse must be *strong* enough to propagate through the circuit,
2. a critical *pipe* should exist from the affected node to a latch,
3. the pulse must have sufficiently high amplitude and width to be able to write the latch, and
4. the arrival of the pulse at the latch must coincide with its write-enable strobe.

This implies that there is a small probability that an SEU, occurring in combinational logic, will be registered in a latch. Even if it does get latched, it appears as if the latch itself experienced an upset. Thus, most of the errors due to SEUs in combinational logic can be represented by corresponding errors in latches and, therefore, it is sufficient to consider SEUs in latches only for a complete analysis.

1.3 SEU Induced Errors

Errors produced by SEU may be broadly classified into three categories: data errors, control flow errors, and execution errors.

1.3.1 Data Errors

A data error is produced when an upset occurs in a register or a memory location. The choice of a data error detection strategy is influenced by the location of the error: memory, bus or the processor. Error tolerant memory architectures either employ single error correction codes with periodic memory scrubbing or use redundant memory units. Data errors on the system bus or in the processor can be detected by coding redundant data with non-redundant data to form a modified value e.g., Hamming codes, residue number system, etc. Error codes for detecting data errors have been well-studied [11,30,31]. Data errors will not be discussed further.

1.3.2 Control Flow Errors

A change in the state of the system that results in loss of control is called a control flow error. These can be thought of as errors in the addressing logic of the microprocessor that alter the normal sequence of instructions. These addressing errors can result in the following situations:

1. execution of an incorrect sequence of instructions, or
2. a branch to/from an incorrect address.

An incorrect jump results in the loss of control and may put the system in an improper state making it difficult to initiate a successful recovery.

1.3.3 Execution Errors

A perturbation in the internal registers of a microprocessor during execution of an instruction can produce an execution error. This can result in the following scenarios:

1. no activity takes place,
2. a different instruction is executed instead of the one intended,
3. in addition to the current instruction, another instruction is also executed,
4. an instruction gets aborted during execution i.e., partial instruction execution,
5. an illegal opcode pattern is created, or
6. an improper control signal is generated.

The difficulty in detection of instruction execution errors arises from the fact that the internal signals and register operations of a microprocessor are not externally observable.

1.4 Fault Tolerance Concepts

Faults in a digital system can produce errors that can be detrimental to its performance. There are two strategies to combat these: fault prevention and fault tolerance [8,11,12,30,31]. **Fault prevention** schemes attempt to prevent, by construction, the occurrence of faults. A priori knowledge of all possible failure modes is required in designing a completely fault free system. Given the complexity of modern circuits, this is almost an impossible goal to achieve. Therefore, effort is made to reduce the

probability of a fault to an acceptably low level. **Fault tolerance** schemes, on the other hand, assume that a certain number of faults will occur despite using reliable components. Consequently an attempt is made to design the system so that it automatically counter acts the effects of faults.

Techniques for incorporating fault tolerance in a computer system can be broadly classified into three categories: self diagnosis, redundancy, and system monitoring. An overview of these schemes and their suitability for detection of transient faults follows.

1.4.1 Self Diagnosis

In this approach, the processor executes a diagnostic program which tests each of its functional blocks. This requires a minimal amount of external hardware, such as interrupt logic, that periodically triggers the diagnostics [3,32]. Since the processor is not available for normal processing when diagnostics are executed, it is not a good system for real-time applications. Also, since faults are detected only when diagnostics are run, this scheme cannot detect transient faults that occur during normal operation.

1.4.2 Redundancy

Redundancy schemes detect or mask faults by using additional hardware or software. Fault masking by majority voting is the principle behind many redundancy schemes. Redundancy can be achieved in hardware, software, or time. **Hardware redundancy** calls for voting on the outputs of several identical tightly coupled hardware units

working concurrently [1,11,30]. A number of fault-tolerant general purpose computers that use hardware redundancy are commercially available. Triple Modular Redundancy (TMR) is a popular method where three identical units are used. Although hardware redundancy has a very small fault latency, there is a high hardware overhead cost associated with it. **Software redundancy** is attained by using more than one algorithm to accomplish a given task; possibly on separate processors. The outputs of different algorithms are voted in software to obtain the final result [9,10,33,34]. **Time redundancy** requires executing the same algorithm on the same or different processors at different times. The outputs from different runs can then be voted in software later to obtain a final result.

The voting mechanism in redundancy schemes is the single point of failure and must be made very reliable. Redundancy techniques are very effective in detecting faults, especially transient faults, but have a high overhead penalty which makes them unsuitable for small satellite control systems that have limited power and weight budget.

1.4.3 System Monitoring

The concept of system monitoring by observing certain attributes of a system has received considerable attention recently. This strategy calls for adding a moderate amount of hardware to the processor in order to monitor its data, address and control lines to detect faults. The monitoring hardware can either be external or internal to the processor. The external monitor is also called a **watchdog processor**. An internal monitor has a higher fault observability than an external monitor because it has ac-

cess to all internal signals of the processor but cannot be used for existing off-the-shelf microprocessors.

System monitoring methods can be classified into on-line and off-line schemes. An **on-line monitoring scheme** detects errors concurrently in real-time when the system is operational. The **off-line monitoring scheme**, on the other hand, collects performance data during normal processing and analyzes it later to verify correct operation. An on-line monitoring scheme has a higher error detection latency than a hardware redundancy scheme but has a much lower hardware overhead. It operates in real-time and is, therefore, effective against transient faults. An off-line scheme, on the other hand, does not detect faults in real-time and, therefore, is not useful for detecting transient faults [35,36].

1.5 Purpose and Significance of this Research

Transient faults will continue to be a problem in digital systems operating in radiation environments and the situation may become more aggravated in future designs because of smaller component dimensions. It is important that faults are detected as quickly as possible after their occurrence and an appropriate recovery action is taken to limit the propagation of errors. A fault detection strategy that tries to minimize the detection time and has a high error coverage is essential.

Most existing techniques attempt to provide fault tolerance by either using redundancy for fault masking or by monitoring the processor for proper operation. The use of redundancy, although very effective, may not be suitable for applications that have constraints on their power and weight budget. Most processor monitoring schemes

have limited scope and can only detect certain types of errors. This limits their error coverage. For instance, most monitoring techniques using signature analyzers detect only control flow errors. Execution errors are either not covered at all or are detected with a very long latency. This is an undesirable situation because a small latency is essential for efficient recovery. Therefore, there is a need to develop methods for implementing fault tolerance at reasonable costs.

The major thrust of this research is to develop a technique that is capable of detecting, in timely fashion, errors produced by SEUs. To this end, the proposed scheme is aimed at detecting both control flow and execution errors. The implementation of this scheme shows that most of the hardware used for detecting execution errors can also be used to detect control flow errors with very few modifications.

1.6 Organization of the Dissertation

A brief discussion of the problems associated with SEU phenomenon has been presented in this chapter. The background of the research reported in this dissertation is presented in Chapter 2. This includes a review of the existing techniques for detecting transient faults in digital systems and their salient features. Next, the limitations of these schemes for SEU environment are discussed. In chapter 3, a functional fault model of SEU induced errors is presented. The concept of the execution profile of an instruction is defined and is shown to be effective in detecting transient faults. The details of the proposed error detection scheme are presented in Chapter 4. The strategy for testing and evaluating the performance of this scheme is discussed in chapters 5 and 6, respectively. The results of the testing are analyzed in Chapter 7.

In Chapter 8, conclusions are drawn from this research and thoughts on further work are presented.

Chapter 2

Previous Research

Interest in the design of fault tolerant computers was initially caused by the error susceptibility of the components: vacuum tubes and relays. The advent of integrated circuit technology greatly improved component reliability. It became difficult to justify the cost of automatic fault recovery mechanisms. The emphasis shifted to the testing of logic so that faulty hardware may be replaced after a quick fault detection [1]. This concept is very cost effective but requires external help when faulty hardware is identified. Also the system has to be shut down in order to perform diagnostic tests. This is not a viable solution for spacecraft systems where external repair facilities are generally not available and continuous operation is desired. Therefore, an on-line testing mechanism is required that works concurrently with the system. The ability to detect faults with small latency is highly desirable. Because of the very nature of transient faults, all that is needed is to detect them and initiate a recovery. There is no faulty hardware to repair or replace as transient faults only disturb the state of the system and do not leave any permanent damage.

2.1 Fault Tolerance Techniques

The problem of SEUs in digital circuits of spacecraft hardware has been of interest for a long time. Both theoretical and practical work has been done to better understand the SEU phenomenon and establish criteria for fault tolerant design. This has led to several fault tolerance approaches that vary in their design and concept. These include improvement in the manufacturing process, changes in geometric layout of the IC components, fine tuning of the system based on statistical models, and error detection schemes that range from pure hardware to pure software approaches. A brief discussion of these concepts follows.

2.1.1 Technology Approach

There have been several experimental studies to understand the nature of SEUs and their effects on digital devices. In experimental studies the actual environment in space has been simulated by bombarding integrated circuits with high energy particles [19-29]. It has been observed that certain nodes in a circuit are more susceptible to SEUs than others [37]. This has led to the creation of radiation hardened (rad hard) technologies. Rad hard components exhibit several orders of magnitude improvement in immunity to SEUs. An example of how circuits can be rad hardened can be seen in memory devices where a transistor memory cell may be made more resistant to SEUs by increasing the coupling resistance of sensitive nodes. Although these measures have significantly decreased the susceptibility of components to SEUs, the fact remains that it is not possible to completely eliminate SEUs.

Many SEUs are caused by high energy particles passing through a circuit and depositing energy in the form of movement of charge. The energy deposit needed for an upset to occur is proportional to the capacitance of the node. This capacitance decreases with decreasing dimensions thus lowering the upset threshold of the device [24]. By increasing the capacitance of the device it is possible to reduce the error rate. This means keeping gate areas large enough to store the critical charge and source/drain areas small to minimize the path length of incident particle in silicon [38]. Although experimental evidence supports this idea, it is but a temporary solution as the device dimensions will undoubtedly become smaller with advancing technologies. Additionally, increasing capacitor size both increases power consumption and decreases speed, both of which are undesirable effects.

2.1.2 Statistical Approach

A model for predicting the tolerance of a system to transient errors is described in [39]. It is based on data collected from four existing systems which experienced transients due to radioactivity in the packaging material of the integrated circuits. A statistical model has also been developed for designing microprocessor-based industrial controllers which is to operate in an electrically noisy environment [40,41]. An assessment of the probability of recovery following a random jump due to an SEU is discussed in [42] wherein practical measures have been suggested that can increase the probability of recovery necessitating only minor changes in hardware and software.

2.1.3 Hardware Approach

Redundancy techniques have been successfully employed in many reliable systems. Redundancy in hardware, software, or time, attempts to eliminate errors by masking faults [1,10,11,30,31,33]. **On-line** or **concurrent detection** of faults in microprocessors has been proposed in the literature using external detectors, watchdog timers, or partial hardware redundancy [35,36]. One recovery scheme for transient faults has been implemented by having the microprocessor generate timing pulses at regular intervals. In the case of a transient error, the train of pulses is disturbed prompting the external hardware to trigger a reset signal [43]. A **self testing** methodology for periodic testing of a microprocessor is described in [32]. It is important to note that periodic testing may not be very effective against SEUs which occur randomly for short durations and do no permanent damage.

A **containment set theory** has been proposed in [44-46]. It defines the operation of a microprocessor system by a containment set which consists of a finite number of mutually exclusive states covering all possible transfer functions (both valid and invalid). A transition matrix is then defined over this set which lists the probabilities of transition from one state to another following an upset. The fault tolerance of the system can be calculated using this transition matrix. For practical implementations of these concepts, an experimental study was conducted to assess the performance of different error detection mechanisms [47]. A hardware detector based on this concept for detecting transient faults is described in [48,49].

In another hardware solution, the exact execution of an instruction is verified by comparing its actual signature with a predetermined signature value [50]. The sig-

natures are generated by using the control signals of the processor as inputs to a parallel linear feedback shift register. This scheme has been implemented for an 8085 microprocessor. The instruction set of the 8085 is mapped in 21 valid signatures. This scheme has been implemented for an 8085 microprocessor, whose entire instruction set is mapped into 21 valid signatures. Fault masking in this approach is significantly high because many instructions have identical signatures.

2.1.4 Software Approach

A number of purely software approaches to SEU detection have also been proposed. One such scheme calls for **self testing software** which can detect transient errors by checking the program flow. The software is partitioned into blocks. A unique tag is created upon entry to a block and cleared upon exit. Any improper jump into another block induced by an SEU is detected by checking the tag [50,51]. An automated scheme for incorporating self testing capability in application software has been implemented in [53]. A translator program scans an Ada application software and inserts the necessary code for implementing the self testing mechanisms in software. Another scheme incorporates fault tolerance by embedding **executable assertions** in the software [54,55]. These assertions test the outcome of an algorithm to see if the results are reasonable. An error is detected if an assertion is violated. A similar approach calls for using an alternate algorithm if an error is detected [56]. All of these schemes detect control flow errors only.

Watchdog timers have also been successfully employed to detect certain types of errors [57]. A watchdog timer is essentially a count down timer which is preset by the executing software before a block of code is entered. This preset value is slightly

longer than the actual execution time for a particular block. If the watchdog timer counts down to zero before the end of the block is reached, the processor has likely inadvertently branched incorrectly. This error condition is signalled by the watchdog timer.

2.1.5 Hybrid Approach

The fault tolerance schemes described in previous sections imply that some types of errors are easily detectable using hardware solutions while others are more suited to software solutions. To incorporate the best of both worlds, hybrid schemes have been proposed that have both hardware and software components. These schemes are primarily designed to detect control flow errors by monitoring the sequence of instructions being executed on the main processor. The structure of the application program is represented by a directed graph; the nodes of the graph represent the blocks or branch free intervals in the program and the arcs indicate the valid transitions i.e., jump instructions. The proper execution of the program is checked by tracing on the graph, the transitions made by the program during normal operation and verifying that no improper path was traversed.

In many of these hybrid approaches, signature analyzers are employed to trace the program flow during normal operation. A **signature analyzer** is a data compression mechanism in which a selected number of signals are fed into a linear feedback shift register at every clock cycle. The feedback circuit is defined by a primitive polynomial that ensures that the number of patterns generated in this feedback process is large. This minimizes any fault masking. The contents of the shift register is called the **signature** of the shift register. A signature produced during a fault-free

run is known as a golden signature and is saved for later comparison. An on-line signature is generated and is compared at specified intervals with predetermined values that are embedded in the program at compile time [11,30,58-60].

The implementation of signature analyzers requires the use of external hardware that may consist of an additional dedicated processor, also called a **maintenance processor** or a **watchdog processor** [35,36]. The processor being monitored is called the **main processor**. The effectiveness of a watchdog processor is a function of the architecture of the main processor and the cooperation of its operating system. A set of observable features of the main processor is established by monitoring a number of its I/O pins. This is used to determine the current state of the main processor. The watchdog processor also knows the expected response of the main processor in the absence of faults. Any deviation in the normal sequence of operations due to a fault can then be detected. The design of a dedicated watchdog processor is very application specific, making it very costly to implement.

A number of schemes for checking control flow of programs have been proposed that require low complexity watchdog processors. Some of these schemes include **structural integrity checking (SIC), path signature analysis (PSA), signatured instruction streams (SIS), asynchronous signatured instruction streams (ASIS), embedded signature monitoring (ESM), continuous signature monitoring (CSM), roving monitoring processor (RMP), control state checking (CSC),** and **synchronized monitoring by a watchdog processor (Cereberus-16)** [57,61-72].

In the SIC scheme, the high-level control flow structures in an application program are identified and unique signatures are assigned to every structure. A companion program, containing this structural information about the application program, is cre-

ated for execution on a watchdog processor. The watchdog processor verifies the program flow during run time by comparing the signatures produced by the main processor with those embedded in its own program. In the PSA scheme, a deterministic signature is derived for each node of the program graph; the signature represents some characteristic of the node. The signature is inserted at the beginning of each node. This signature is distinguished from other program instructions by using two tag bits. During normal operation, the watchdog processor captures these signatures as they appear on the bus while the main processor executes a no operation (NOP) instruction. The watchdog processor computes the signature of the node concurrently. At the end of the node, the watchdog processor compares the computed signature to the actual signature to detect any errors. The amount of memory used for storing the signatures can be significant for programs with small branch free intervals. A novel approach has been proposed in the SIS scheme for reducing this memory overhead. The signature of a node is hashed with the destination address of the jump instruction at the end of the node. This memory overhead can be further reduced as demonstrated by the ESM and CSM techniques.

Most of the schemes mentioned above require that the signatures be embedded in the instruction stream thereby increasing the execution time of the program. This time penalty can be eliminated by storing the signatures in the local memory of the watchdog processor. Three such schemes have been proposed. The first is a synchronous scheme using a watchdog processor called Cereberus-16. The information about the program graph and signatures is included in the program executed by the watchdog processor. The instruction set of Cereberus-16 consists of two types of instructions; instructions for representing control flow of the program graph and instructions for initialization and communication with the main processor. For every

node executed by the main processor, a corresponding instruction, that includes the signature of the node, is executed by the watchdog processor. The actual and observed signatures are compared after every node is executed. The other two schemes, the ASIS and the RMP schemes, are asynchronous and suited for multiprocessor environment. Every processor has a signature generator which generates a stream of signatures and stores it in a signature queue. A centralized monitor samples the signature queues of every processor to check if the signatures generated on-line match with those already stored in the local memory of the monitor.

The concept of control flow checking can also be used to verify certain operations in the control section of a processor. The CSC scheme checks the sequence of control signals corresponding to each opcode by assigning a unique signature to each opcode. These signatures are stored in a ROM. When an instruction is executed by the processor, certain control signals of the processor are used to generate a signature concurrently. The same opcode is used to fetch its actual signature from the ROM for comparison.

2.1.6 Circuit Design Approach

The architecture of a processor is an important factor in the design of detection mechanisms. **Design for testability** considerations in hardware are concerned with making internal registers and control signals accessible to an external tester. This makes the design of the external tester very simple [73]. However, it also involves additional hardware which could make the system more vulnerable to SEUs. Other options include modifying the microprogrammed control unit of the main processors to embed signature keys for signature analysis at the microinstruction level, incor-

porating error codes in the control unit, and duplicating critical sections of the control unit [74-81]. It has also been suggested that the set up times of flip flops and latches be made sufficiently longer than the expected maximum duration of a transient. This will mask an SEU on a line before it is clocked into a register [38]. This is not a good solution as it puts a limit on maximum clock rate.

A concurrent testing scheme using a signature analyzer is proposed in [62,69]. This approach verifies the program flow after execution of each instruction. It is implemented using either a special field in each microinstruction or defining a special CPU instruction for storing intermediate signatures. These ideas, although very attractive, are only feasible if we have the luxury of designing new processors instead of using off-the-shelf processors.

2.2 Limitations of Previous Techniques

The primary focus of this research is to devise a means for detecting transient errors produced by SEUs. A number of the techniques proposed in the literature deal with microprogrammed control units that incorporate fault tolerance by modifying the internal architecture of the processor. These techniques call for redesigning the control units by incorporating error codes, duplication of critical sections of the hardware, extending microinstructions to include keys for signature analysis, or simply making the entire microprogram control unit redundant. These ideas are not applicable to existing off-the-shelf processors whose internal architectures are not modifiable.

Most techniques for detecting control flow errors have a high fault coverage and significant error detection latency. Also, there are some faults that are not covered and can potentially lead to system failure. Execution errors are the result of the processor being perturbed after data or a program instruction has been read from main memory. More specifically, these are errors due to a fault in the instruction register or the execution logic of the processor. For instance, the opcode of an instruction can inadvertently be changed to that of another instruction. This is difficult to detect since the executing instruction is not externally observed is the techniques described earlier.

There are a number of error scenarios that are not covered by any of the techniques mentioned previously and can potentially lead to system failure. Some of these scenarios include:

1. If the destination address of a jump instruction is altered because of an error, it may result in the execution stream being resumed at an incorrect address. This can be detected using signature analysis or signature monitoring schemes except when:
   a. the erroneous jump resumes execution at the beginning of an incorrect block.
   b. the erroneous jump resumes execution at the second or subsequent byte of a multi-byte instruction. This can potentially set up an endless loop in the software that can lead to system failure [52].
   c. the erroneous destination address is in an unused or non-existent area of the memory.
2. A potential for an endless loop exists when the increment logic of the program counter gets disabled; a special case of dependent multiple errors [64].

3. The opcode of an instruction may be changed to a pattern that is not defined in the instruction set of the processor i.e., an illegal opcode.
4. The opcode of an instruction may be changed to that of another instruction.
5. Instruction execution errors are not detected unless they also produce control flow errors.

The above mentioned situations clearly indicate the need for a detection strategy that can provide a high coverage for these types of faults. Since there are limitations on power and weight allocations in a small system, it is important to minimize the amount of additional hardware and software for implementing the error detection scheme.

Chapter 3

Concurrent Processor Monitoring Scheme

Transient faults caused by SEUs are random and non-recurring in nature. As there is no permanent damage to the hardware, their detection and isolation is difficult. The most important consideration in the development of a fault tolerance scheme is to ensure that there is no loss of control in the system. Secondary issues include the detection of execution of illegal opcodes, execution errors, initiation of erroneous execution at the wrong memory addresses and data errors that may cause the output of the system to be incorrect. Theoretical background of the proposed scheme is developed in this chapter starting with the fault models and basic concepts.

3.1 Fault Observation

The classical approach to fault detection in digital systems is to apply a test that is developed by representing the effect of a fault by means of a model that represents the change the fault produces in circuit signals. The inputs of the circuit are then configured in such a way that the effect of the fault is propagated to an observable

point. The process of manipulating the inputs to propagate a signal to an observable point is called path sensitization [11,30,82-84].

The above mentioned testing approach is limited by the fact that the system under test must suspend its normal operation so that test inputs can be applied. This precludes any concurrent testing that is necessary for detecting transient faults. Another consideration is the complexity of modern high density VLSI circuits; a very small number of their internal signals are externally observable. This limited observability makes it difficult to propagate most faults to an observation point.

The presence of transient faults can be observed at a higher level called the **upset level** [45,46]. At this level the system is viewed as responding to arrival and departure of a transient fault in two stages; transient response and steady state response. The transient response of the system, present during or soon after the occurrence of an SEU, may have a temporary effect on the system such as a data error, loss of synchronization, skipped execution step, etc. It leaves no lasting effect and the system function remains the same. On the other hand, the steady-state response of the system results in a functional transformation. That is, after departure of the fault, the system is no longer performing the same transfer function. This is analogous to a jump into a different loop in the software. For implementation of an error detection scheme based on this concept, the system must be described by a set consisting of a finite number of mutually exclusive states covering all possible transfer functions. This is not an easy goal to achieve especially for systems of moderate to large complexity.

An off-line scheme for detecting permanent faults is presented in [85]. The algorithm calls for observing execution time and memory read/write sequences for each in-

struction while explicit test are performed for every type of faults postulated in [86]. This method has a high fault coverage but is not suitable for transient faults because it does not detect faults in real-time.

3.2 Concurrent Processor Monitoring

The classical methods of fault detection have limitations in the context of transient faults in modern VLSI circuits. An alternative approach is to observe the performance of the system at a higher level. At this level a sophisticated control system monitors the external signals of the processor to determine if the transient fault has caused an upset. By noticing any deviations in its operation from normal execution a fault can be detected. Most monitoring techniques described in Chapter 2 are based upon this concept.

An approach that calls for concurrent observation of an instruction under execution is proposed in this dissertation. The proposed approach, called the **concurrent processor monitoring (CPM) scheme**, performs continuous on-line monitoring of the processor and is, therefore, useful in detecting transient faults. For implementation of the CPM scheme, a **watchdog processor (WDP)** is required to act as a coprocessor to the main processor and monitor the system bus signals and the status signals of the main processor. Since high reliability is of prime importance in real-time systems, the WDP must concurrently monitor and detect faults in the system during normal operation. This has two attractive features [35,36]:

1. Since the WDP is devoted to continuous monitoring of the system, most abnormal conditions can be handled promptly.

2. The main processor can concentrate on information processing, its main task, without the added functionality and architectural complexity required to control and maintain itself in the face of transient upsets.

These features are especially useful for the current off-the-shelf microprocessors which do not have built-in self monitoring mechanisms.

In order to ensure correct operation of the main processor by means of on-line monitoring carried out by the WDP, it is necessary to define monitoring specifications. This includes defining a set of observable features of the main processor that can be compared to a previously determined set of reference features that is a subset of the operating states of the main processor [35,36,46]. The following issues must be addressed while defining these specifications:

1. What features of the main processor are observable?
2. What are the symptoms of incorrect performance that can be identified by observing these features?
3. What subset of these observable features should be selected for monitoring?
4. How will the WDP interpret these feature to determine the performance of main processor?
5. How will the WDP respond if it detects a fault?

These and other questions are addressed in the following sections when the proposed scheme is explained.

3.3 Assumptions

The CPM scheme monitors certain attributes of the processor to detect faults. To define monitorable attributes of the processor, we first present the necessary assumptions. The effect of a transient fault is similar to that of a permanent fault except that a transient fault occurs at a certain point in time and does not persist. The following assumptions have been made:

1. There is only one transient fault in the system at any time.
2. A transient fault persists for one instruction cycle only.
3. A transient fault changes the contents of a latch/register but does no permanent damage to the hardware.
4. The programs under execution do not modify themselves i.e., no self modifying code.

The assumptions listed above do not place any severe restrictions on the types of errors that are considered in this dissertation. Experimental and analytical data reported in the literature show that the frequency of SEUs in a typical satellite control system operating in low orbit near earth is about one error per day [16-18,28]. This error rate may be as high as one error per hour in the worst case. This implies that the assumptions of one fault in the system at any time is reasonable.

3.4 Transient Fault Model

As indicated earlier in this chapter, the complexity of modern VLSI circuits makes it very difficult to perform testing at gate level in a comprehensive and economic way. An alternative approach is to use abstract, formal functional description of the circuit to express its operation at a higher level. A functional fault model can then be defined with respect to a certain set of functional faults in the circuit. These functional faults are actually circuit-level faults manifested at the functional level.

Models for permanent faults have been developed for a general graph theoretic model of a microprocessor at the functional level [86]. The microprocessor is represented by a set of functions such as register decoding function, data paths, data manipulation functions, and instruction sequencing operations. A functional fault is then developed for each of these functions. These functional fault models are capable of describing faulty behavior of the microprocessor at a higher level of abstraction that is independent of implementation details. The functional models for permanent faults have been defined in [86]. In this section, these models are extended to include transient faults as well. As we are primarily interested in control flow and instruction execution errors only, the following discussion is limited to these faults.

Let I be the set of all instructions of a microprocessor, i.e., $I = \{I_1, I_2, \ldots, I_n\}$. Then the types of faults that can occur during the execution of an instruction are:

1. I_j/ϕ **Fault:** During the execution of an instruction I_j, no activity takes place in the processor.

2. I_j/I_k **Fault:** During the execution of an instruction I_j, the microoperations of a different instruction I_k are executed i.e., a different instruction I_k is executed.
3. $I_j/I_j + I_k$ **Fault:** During the execution of an instruction I_j, two instructions I_j and I_k are executed to completion.
4. I_j/I_j' **Fault:** The execution of an instruction I_j is aborted before completion.
5. $I_j,I_k/I_j,I_m$ **Fault:** Following the execution of instruction I_j, the next instruction fetch is out of sequence. Thus instead of I_k, a different instruction I_m is executed.
6. I_j/I_o **Fault:** During the fetching of an instruction I_j, an illegal opcode pattern is encountered.
7. I_j/μ **Fault:** During the execution of an instruction I_j, an incorrect sequence of microoperations is executed resulting in generation of improper control signals.

I_j/ϕ, I_j/I_k and $I_j/I_j + I_k$ faults were postulated in [86]. These fault models represent errors in the instruction logic of the microprocessor. To cover other error scenarios, four additional fault types are defined. An I_j/I'_j fault covers the case when execution of an instruction is prematurely aborted due to some error in the execution logic of the microprocessor. A transient fault in addressing logic of the microprocessor can result in the fetching of an instruction which is out of the normal sequence of instructions. This is represented by an $I_j,I_k/I_j,I_m$ fault. This is a control flow error that may occur in an SEU environment. An opcode pattern which is not in the instruction set of the microprocessor is called an illegal opcode. It may be created when the instruction register or the instruction queue of the microprocessor, if there is one, is perturbed. This is called an I_j/I_o fault. It is actually a special case of I_j/I_k fault. In addition to these faults, an SEU may mask a control signal, trigger an improper control signal, change the microinstruction sequence, or divert program flow by changing the test condition for a branch. These situations are represented as I_j/μ faults.

3.5 Instruction Execution Profile

During the execution of an instruction, the microprocessor generates a number of signals to interact with external devices. For example, while executing a memory read instruction, the microprocessor emits control signals for reading the memory. These and other signals during the execution of an instruction and are used to define the execution profile of the instruction. In the following discussion, the main processor is assumed to be Intel's 8086 microprocessor. For every instruction I_j in the instruction set of the microprocessor, a set of observable parameters are defined below.

Execution Time: The number of clock cycles required by the microprocessor to execute an instruction I_j is the execution time of the instruction and is represented by $T(I_j)$. Since the exact time $T(I_j)$ is difficult to determine for those microprocessors that prefetch instructions, by definition, $T(I_j)$ will not include the time for fetching the instruction from the memory. $T(I_j)$ is defined for each instruction in the data manuals of a microprocessor and can also be found experimentally.

Byte Count: The number of bytes in an instruction, I_j, including any immediate data or displacement offset is the byte count of the instruction and is represented as $B(I_j)$. All instructions of a microprocessor have a fixed number of bytes in their instruction word which includes any immediate data, address, or operand displacement offset.

Data Transfer Count: The number of memory read or write operations during execution of the instruction I_j is the number of data transfers in the instruction. The data transfer count of an instruction I_j is represented as $X(I_j)$. In-

structions that operate on memory are of three types; memory read, memory write, or memory read-modify-write operations. Thus, there are either one or two memory accesses during the execution of an instruction. An exception to this rule is the situation when a two byte word is at an odd address. The 8086 microprocessor will run two bus cycles to fetch this word. For the purpose of this discussion, this is considered as a single bus access.

Data Transfer Size: The size of the data transferred during the execution of an instruction I_j is the data transfer size of the instruction. The data transfer size is represented as $S(I_j)$. This data transfer is between I/O devices or memories and the microprocessor. Depending upon the addressing capabilities of the microprocessor, zero, one, two, or more bytes may be transferred in one instruction cycle.

Instruction Address: The address of the location where the first byte of an instruction I_j is stored in the program memory is called the address of the instruction. It is represented as $A(I_j)$.

3.6 Detectable Errors

In this section, the types of faults that are detectable by monitoring the execution profile of an instruction are established. A set of functions is defined that map the set of instructions, I , into the set of integers, Z, inducing a unique partition on I.

Definition 1.1: A function F_1 maps I into the set Z , $(F_1: I \rightarrow Z)$ i.e., $F_1(I_j) = T(I_j)$, $\forall I_j \in I$. $T(I_j)$ is the execution time of instruction I_j.

Definition 1.2: The function F_1 induces a partition, π_1 , on the set I creating blocks which are defined as:

$I^{t_i} = \{I_j \mid I_j \in I \text{ and } T(I_j) = t_i\}$, where the t_i are integers.

Definition 2.1: A function F_2 maps I into the set Z , $(F_2: I \rightarrow Z)$ i.e., $F_2(I_j) = B(I_j)$, $\forall I_j \in I$. $B(I_j)$ is the number of bytes in instruction I_j.

Definition 2.2: The function F_2 induces a partition, π_2 , on the set I creating blocks which are defined as:

$I^{b_i} = \{I_j \mid I_j \in I \text{ and } B(I_j) = b_i\}$, where b_i are integers.

Definition 3.1: A function F_3 maps I into the set Z , $(F_3: I \rightarrow Z)$ i.e., $F_3(I_j) = X(I_j)$, $\forall I_j \in I$. $X(I_j)$ is the number of data transfers during the execution of instruction I_j.

Definition 3.2: The function F_3 induces a partition, π_3 , on the set I creating blocks which are defined as:

$I^{x_i} = \{I_j \mid I_j \in I \text{ and } X(I_j) = x_i\}$, where the x_i are integers.

Definition 4.1: A function F_4 maps I into the set Z , $(F_4: I \rightarrow Z)$ i.e., $F_4(I_j) = S(I_j)$, $\forall I_j \in I$. $S(I_j)$ is the size of data transfers during the execution of instruction I_j.

Definition 4.2: The function F_4 induces a partition, π_4 , on the set I creating blocks which are defined as:

$I^{s_i} = \{I_j \mid I_j \in I \text{ and } S(I_j) = s_i\}$, where the s_i are integers.

Functions F_1, F_2, F_3, and F_4 define relations on the set I and partition it into blocks. By observing the execution of an instruction, the block that contains that particular instruction can be determined. In order to show that the execution profile of an instruction can be used to detect execution errors, a number of results are stated and proved.

Theorem 1: If $I_j \in I^{t_x}$, $I_k \in I^{t_y}$ and $t_x \neq t_y$, then for an I_j/I_k fault, $T(I_j/I_k) \neq t_x$.

Proof: If instruction I_k is executed instead of I_j, then $T(I_j/I_k) = T(I_k) = t_y$. As $t_y \neq t_x$, $T(I_j/I_k) \neq t_x$.

Corollary 1: If $I_j \in I^{t_x}$, $I_k \in I^{t_y}$ and $t_x \neq t_y$, then all I_j/I_k faults can be detected by observing the execution of I_j alone and comparing the observed execution time with the actual value of $T(I_j)$.

Proof: For an I_j/I_k fault, $T(I_j/I_k) = T(I_k) = t_y$. But $T(I_j) = t_x$ and $t_x \neq t_y$. Therefore, $T(I_j) \neq T(I_j/I_k)$. Hence, observed and expected values of instruction execution time are different and the fault can be detected.

Theorem 2: If $I_j \in I^{t_x}$, $I_k \in I^{t_y}$ and $t_x \neq t_y$, then for an $I_j/I_j + I_k$ fault, $T(I_j/I_j + I_k) = \max(t_x, t_y)$.

Proof: In the case of the $I_j/I_j + I_k$ fault, both instructions are executed. Therefore, $T(I_j/I_j + I_k) = \max(T(I_j), T(I_k)) = \max(t_x, t_y)$.

Corollary 2: If $I_j \in I^{t_x}$, $I_k \in I^{t_y}$ and $t_x < t_y$, then all $I_j/I_j + I_k$ fault can be detected by observing the execution of I_j alone and comparing the observed execution time with the actual value of $T(I_j)$.

Proof: For an $I_j/I_j + I_k$ fault, we have $T(I_j/I_j + I_k) = \max(T(I_j), T(I_k)) = \max(t_x, t_y) = t_y$. Since $t_x < t_y$, therefore, $T(I_j) \neq T(I_j/I_j + I_k)$. Hence, observed and expected values of execution time are different and the fault can be detected.

Theorem 3: If $I_j \in I^{t_x}$ then for an I_j/I_j' fault, $T(I_j/I_j') < t_x$.

Proof: The I_j/I_j' fault means that the execution of instruction I_j gets aborted before it is completed. An instruction requires a finite number of clock cycles for complete execution. If the instruction is terminated before completion, it would take less than the required clock cycles i.e., $T(I_j/I_j') < t_x$.

Corollary 3: If $I_j \in I^{t_x}$ and $T(I_j') < t_x$, then all I_j/I_j' faults can be detected by observing execution of I_j alone and comparing the observed execution time with the actual value of $T(I_j)$.

Proof: For an I_j/I_j' fault, $T(I_j/I_j') = T(I_j') < t_x$. Thus, the expected and observed value of execution times are different and the fault can be detected.

Theorem 4: If $I_j \in I^{t_x}$ and $t_x > 1$, then for an I_j/ϕ fault, $(I_j) > T(\phi)$.

Proof: In the case of an I_j/ϕ fault, no instruction is executed. Since there is no permanent damage, the microprocessor would fetch the next instruction for execution. Thus $T(I_j/\phi) = T(\phi) = 1$. As all instructions require more than 1 clock cycle for execution (e.g., in the 8086 microprocessor), $T(I_j) > 1$ and $T(I_j) > T(\phi)$.

Corollary 4: If $I_j \in I^{t_x}$ and $t_x \neq T(\phi)$, then all I_j/ϕ faults can be detected by observing execution of instruction I_j and determining $T(I_j)$.

Proof: For an I_j/ϕ fault, $T(I_j/\phi) = T(\phi) = 1$. But for any instruction in the 8086 microprocessor, $T(I_j) > 1$. Thus the expected and observed execution time values are different and the fault is detected.

Theorems 1 through 4 can similarly be proved for each of the other parameters i.e., $B(I_j)$, $X(I_j)$, and $S(I_j)$, defined earlier.

In the case of an I_j/I_o fault, the behavior of the microprocessor is difficult to predict. For example, it has been found that some illegal opcode patterns of the 8085 microprocessor actually perform certain operations not included in the documentation. It has been suggested that these opcodes can be used to enhance programming [87]. Since these operation modes are not supported by the manufacturer, it is possible that a device from a different production run may operate differently. Therefore, undefined opcodes should not be used and instead treated as illegal instructions. Some modern processors have built-in illegal opcode detectors e.g., Motorola's 68000 and

68020, as well as Intel's 80286 and 80386 each have illegal opcode detectors. All I_j/I_o faults can be detected by these detectors.

Errors in addressing logic can result in an I_j, I_k/I_j, I_m fault. In order to detect this fault, it is necessary to have information about the dynamic behavior of the executing program. The following theorem provides the basis for monitoring an instruction sequence.

Theorem 5: The address of an instruction I_k, that succeeds instruction I_j , barring any jump or interrupts, is $A(I_k) = A(I_j) + B(I_j)$.

Proof: The execution of instructions is sequential in a program unless a jump or interrupt instruction is encountered. The address of instruction I_j is $A(I_j)$. Since the length of the instruction I_j is $B(I_j)$ bytes, the next sequential instruction I_k must be located at $A(I_k) = A(I_j) + B(I_j)$.

Corollary 5: For any sequence of instructions I_j followed by I_k , a control flow error can be detected by comparing the address of instruction I_k, i.e., comparing $A(I_k)$ with $A(I_j) + B(I_j)$.

Proof: Since I_j and I_k are sequential, $A(I_k) = A(I_j) + B(I_j)$. Thus, any control flow error that produces a next address which is different from $A(I_k)$ is detected.

The response of a microprocessor to an I_j/μ fault may depend upon its particular architecture. For instance, consider the 8086 microprocessor. It has a built-in queue for prefeteching instructions. An incorrect queue update signal can result in over or underflow of the queue. This can be detected by a watchdog processor that tracks the operation of the microprocessor's queue by maintaining a copy of the queue and updating it whenever there is a queue operation.

The 8086 microprocessor differentiates between data and program memory locations and addresses them differently using separate segment registers. Thus, an addressing error which results in access to program memory instead of data memory or vice versa can be detected if the watchdog processor knows the type of access required. Another type of memory addressing problem may result when the address of a non-existent memory location is generated. This is called **undefined memory** that does not exist physically. An address checking mechanism can be used to cover this type of fault. A stack is indispensable for subroutine calls. A change in the value of stack pointer can result in loss of the return address. This type of error can be detected by monitoring the value of the stack pointer in the monitoring hardware.

3.7 Detection Mechanisms

In order to utilize the concepts developed in the preceding sections, a number of detection mechanisms are needed. These detection mechanisms can be classified into two categories; generic mechanisms and processor-specific mechanisms. A **generic mechanism** is conceptually applicable to any microprocessor. Generic mechanisms can be further classified as instruction-related or processor-related. An **instruction-related generic mechanism** detects an error by monitoring the execution of an instruction. Detection mechanisms based on the execution profile of an instruction are classified as instruction-related generic mechanisms. These include mechanisms that will be defined later as: execution time check (ETC), byte count check (BCC), data transfer check (DTC), data size check (DSC), and next address check (NAC). On the other hand, a **processor-related generic mechanism** monitors a specific characteristic of the processor. This category is comprised of illegal

opcode check (IOC), stack address check (SAC), and undefined memory access check (UMA). These mechanisms are defined later.

A **processor-specific mechanism** makes use of a unique feature of the microprocessor that may not be available in other processors. For an 8086 microprocessor, these mechanisms, to be defined later, include improper queue signals (IQS) and improper memory access (IMA). A distinction is made here between a processor-related generic mechanism and a processor-specific mechanism in that the former monitors a characteristic of the processor that is common to most processors e.g., stack operations, illegal opcodes etc., and the latter observes processor characteristics that are not shared by every processor e.g., the instruction prefetch queue in the 8086 microprocessor. A list of all detection mechanisms for an 8086 microprocessor is shown in Table 1. These mechanisms are formally defined in Chapter 4. The 8086 microprocessor was chosen for this work so that processor specific mechanisms can be implemented.

Table 1. Error Detection Mechanisms

INSTRUCTION-RELATED GENERIC MECHANISMS
• ETC - Execution Time Check • BCC - Byte Count Check • DTC - Data Transfer Check • DSC - Data Size Check • NAC - Next Address Check
PROCESSOR-RELATED GENERIC MECHANISMS
• IOC - Illegal Opcode Check • SAC - Stack Address Check • UMA - Undefined Memory Access
PROCESSOR-SPECIFIC MECHANISMS
• IQS - Improper Queue Signals • IMA - Improper Memory Access

Chapter 4

Design Implementation

Implementation of the proposed concurrent processor monitoring (CPM) scheme requires a watchdog processor (WDP) that is capable of determining the state of the microprocessor by observing its status and control lines and traffic on the system bus. An interface between the microprocessor, hereinafter referred to as the main processor, and its watchdog processor is shown in Figure 1. In the following sections, design requirements for the watchdog processor are outlined and details of all detection mechanisms are discussed.

4.1 Design Requirements

The WDP is to implement a scheme that monitors a set of observable features of the main processor. Although the concept of observable features is quite general, its actual implementation is highly processor specific. Design of the WDP is strongly influenced by the architecture of the main processor because there is large variation in the characteristics between different microprocessors. This implies that it is not

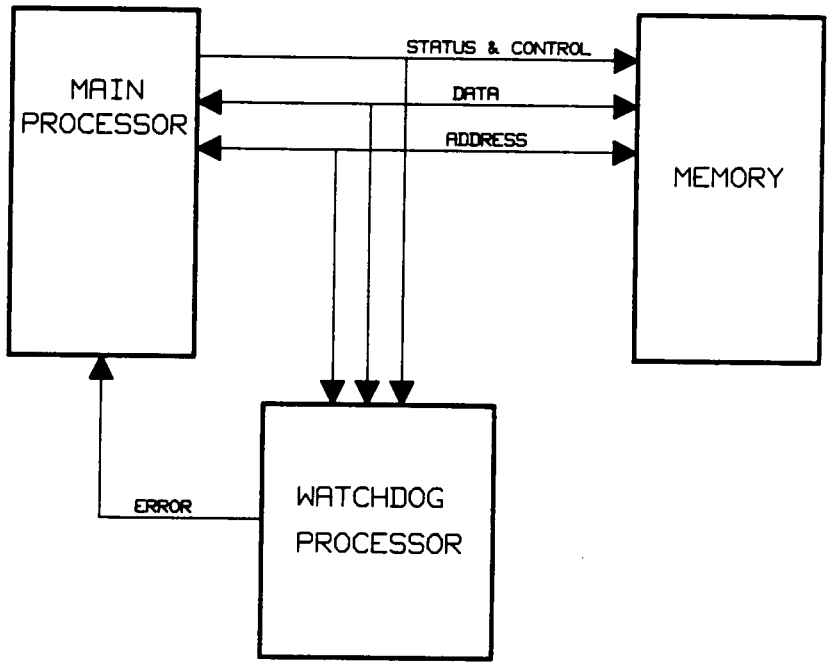

Figure 1. Watchdog Processor Interface

feasible to design a single general purpose WDP to go along with every microprocessor. The design of the WDP should satisfy the following requirements [35]:

1. It should not interfere with the normal operation of the main processor.
2. It should be able to determine the state of the main processor by monitoring its external status and control signals and bus traffic. Faults should be detected on the basis of this information.
3. It should have low-complexity logic so that hardware overhead can be kept to a minimum.
4. It should have high fault coverage.
5. It should have low fault detection latency so that error propagation can be minimized.

4.2 8086 Basics

The CPM scheme presented here is for an 8086-based system. Therefore, some comments about the architecture of the 8086 microprocessor are in order. The 8086 CPU consists of two functional units; the execution unit (EU) and the bus interface unit (BIU). The two units operating independently are able, most of the time, to overlap instruction fetch with instruction execution and thus optimize the use of the system bus. The EU performs all arithmetic/logic operations and depends upon the BIU for all bus accesses. An instruction queue is maintained in the CPU. Usually this queue contains at least one byte from the instruction stream and thus the EU does not have to wait for an instruction to be fetched. If the EU executes a jump instruction or needs data from a memory location, it requests that the BIU make a bus access. The BIU

either drops its current operation to perform the requested fetch or delays the request by one clock cycle.

The 8086 microprocessor has two modes of operation; minimum mode and maximum mode. When operating in the maximum mode, the 8086 provides information about instruction queue operations through lines qs0 and qs1. The queue has four modes of operation: no operation (qs0,qs1 = 0,0), first byte of an instruction taken from the queue (qs0,qs1 = 0,1), queue reinitialized (qs0,qs1 = 1,0), and subsequent byte of an instruction taken from the queue (qs0,qs1 = 1,1). The CPU has eight types of bus cycles that are encoded on lines s0*, s1* and s2*. These types include: interrupt acknowledge, read I/O, write I/O, halt, instruction fetch, read memory, write memory, and no operation. The external processor uses these status signals to determine the type of operation being performed by the CPU [93].

4.3 Execution Profile Verification

Information about the expected execution parameters of an instruction (i.e., $T(I_j)$, $B(I_j)$, $S(I_j)$, and $S(I_j)$) can be obtained either from the data manuals of the microprocessor or observed experimentally. This information is programmed into a PLA in the WDP. This is called the **instruction profile table (IPT)**. At the time of execution, the WDP monitors bus traffic and fetches instruction opcodes along with the main processor. It partially decodes the instructions in parallel with the main processor and reads their corresponding parameters from the IPT. The WDP also observes the instruction parameters concurrently as execution progresses. After completion of the execution of an instruction, the parameter values read from the IPT are compared

with the ones observed concurrently to detect any error. The WDP resets its monitoring logic in conjunction with the beginning of the next instruction. The address of every instruction executing on the main processor can be noted directly from the address bus. This address information is used by the address verification scheme to be discussed later. The algorithm for detection of execution errors is described below.

4.3.1 The Execution Profile Verification Algorithm

1. Calculate (or experimentally determine) for each instruction, the parameters $T(I_j)$, $B(I_j)$, $X(I_j)$, and $S(I_j)$, and save them in the IPT.
2. Initialize the next address value in the WDP.
3. For the next instruction fetched by the main processor, the WDP must:
   a. verify the address of the newly fetched instruction by comparing it with the next address value calculated in the previous instruction cycle. This step is omitted if the current instruction is the first instruction of the program.
   b. partially decode the instruction to determine its type and read its execution profile from the IPT.
   c. observe the actual execution profile on-line.
4. Upon completion of the instruction, compare the observed execution profile with the profile read from the IPT.
5. If the two execution profiles do not match, signal an error. Else go to step 3.

4.3.2 Discussion

This algorithm, by design, checks the execution profile of an instruction following the execution of the instruction. Thus, an error that produces an incorrect execution profile would be detected after the faulted instruction is completed. This observation implies that the error detection latency in these cases would be less than two instruction cycles.

4.4 Address Verification

Faults in the addressing logic of the microprocessor produce control flow errors that can be detected by the next address verification scheme. This scheme detects any breaks in the sequence of addresses of sequential instructions. The step 3(a) of the execution profile verification algorithm described in section 4.3.1 calls for address verification of every instruction fetched for execution. The details of this process are described by the following algorithm.

4.4.1 The Next Address Verification Algorithm

1. Initialize *valid_next_address* = NO.
2. Read the address of the next instruction. Set *present_address* = the address of the new instruction.
3. If *valid_next_address* = YES, compare the *present_address* with *next_address*. If *present_address* = *next_address*, the address of the new instruction is validated. Else indicate an error.

4. If the new instruction is not a jump instruction, set *valid_next_address* = YES and *offset* = *instr_length* and go to step 6. The instruction length is in the IPT.
5. If the new instruction is a jump instruction, and the offset for the jump destination is encoded in the instruction, set *valid_next_address* = YES and *offset* = *jump_offset* and go to step 6. Else set *valid_next_address* = NO and go to step 6.
6. Set *next_address* = *present_address* + *offset.*
7. Go to step 2.

4.4.2 Discussion

The executing instruction is partially decoded by the WDP to determine if it is a jump instruction and its length i.e., $B(I_j)$, is read from the IPT. Upon fetching the next instruction, its address is checked against the previously calculated next address to ensure that correct instruction i.e., one from the correct address, has been fetched. As the address of the next instruction is always calculated during the execution of the current instruction, there will not be a *next_address* during execution of the first instruction of a program. An initialization flag i.e., *valid_next_address*, is used to tell the WDP not to test the *next_address* of the first instruction of a program.

Most programs have blocks of 8 to 10 sequential instructions followed by a jump instruction [64]. Jump instructions are of three types; conditional jumps, unconditional jumps, and interrupts. Since unconditional jumps are always taken, the address of the next instruction is included as part of the opcode and can be calculated by the WDP. An exception to this rule is the case when an unconditional jump to a destination is taken whose address depends upon some data value in a register or mem-

ory location, e.g., a jump through a register. Since the contents of a data register is not observable by the WDP, it may not be possible for the WDP to calculate the address of the next instruction. Therefore, this algorithm does not verify the next address following a jump through memory or register instruction. An error in the destination address of such an instruction is not detected. However, the WDP does check to ensure that a jump is taken.

In the case of a conditional jump instruction, there are two possible destinations; a jump address or the next sequential address. The choice between these two paths generally depends upon some condition flag. As the WDP has no access to the condition flags, it may not be able to make a determination about the correct next address. However, the WDP has a knowledge of both the addresses and can detect an error if an instruction is fetched from an address which is different from these two addresses.

From the program execution point of view, an interrupt has the effect of producing a jump operation. This is similar to an unconditional jump except that the program control returns to this location after the completion of the interrupt service routine and additional clock cycles are used for storing return address. Hence, the WDP treats an interrupt as an unconditional jump.

4.5 Additional Detection Mechanisms

The detection mechanisms discussed so far belong to the class of instruction-related generic mechanisms. In order to increase the fault coverage of the WDP, additional mechanisms are needed. These include processor-related generic mechanisms and

processor specific mechanisms. The specific mechanisms in the context of the 8086 microprocessor are considered next.

Processor-related generic mechanisms include: Illegal Opcode Check (IOC), Stack Address Check (SAC), and Undefined Memory Access Check (UMA). The WDP has a complete list of all valid opcodes and their parameters are contained in the IPT. Therefore, any opcode pattern that is not in the IPT is considered an illegal instruction. Thus, an illegal opcode detector can be implemented at virtually no extra cost. The illegal opcode detector is useful only if the fault occurs in the program memory or the system bus. An I_j/I_o fault inside the processor cannot be detected because the instruction register of the processor is not visible to the WDP. However, this fault may produce other errors that could be detected either by the IOC mechanism or one of the other mechanisms.

A copy of the stack pointer is maintained by the WDP. It is verified and updated whenever there is a stack operation. Thus, if an error changes the contents of the stack pointer register, the error can be detected the next time the stack is used. Certain addressing errors can result in access to an undefined memory location. A special circuit designed to detect this type of memory access has also been included in the WDP. All available memory locations and memory mapped devices are fully decoded so that they have unique addresses.

Processor specific mechanisms for the 8086 microprocessor include: Improper Queue Signals Check (IQS) and Improper Memory Access Check (IMA). An improper queue update signal may result in an attempt to read from the queue when it is empty or to write into the queue when it is full. A queue overflow or underflow due to an incorrect queue update signal must be detected. The queue pointer points to the

lowest address, zero in this case, if the queue is empty and to the highest address, six in this case, if the queue is full. An improper queue update signal can be detected by tracking the value of the queue pointer and verifying that it is within the above mentioned address bounds.

An IMA results when the main processor generates a read memory cycle for fetching data but the addressed memory location is in the program memory area or vice versa. The WDP can determine the type of memory access cycle being executed by the 8086 by observing the status signals of the 8086. Hence, an addressing error that produces access to incorrect type of memory location can be detected by the WDP.

4.6 Watchdog Processor Architecture for the 8086

The architectures of modern microprocessors make it difficult to observe their internal registers and internal control signals. Ideally, a fault detection mechanism should be able to determine the state of the main processor at all times. This is accomplished by monitoring some observable features of the main processor. The only observable features of a microprocessor are its I/O pins. The WDP is designed to monitor the traffic on the address, data, and control buses to determine the state of the main processor.

To monitor the execution parameters of an instruction, the WDP must know the particular instruction under execution on the main processor. This information is not easily available on those microprocessors that have instruction prefetch queues because the memory read and write signals for instruction fetches are not synchronized with the execution of the instructions. This is the case for the 8086 microprocessor.

However, the 8086 microprocessor provides status signals (s2,s1,s0) to indicate the type of bus cycle being run. It also provides queue status signals (qs1,qs0) that may be interpreted to determine the instruction that is currently being executed on the processor. Status signals for code/data segments are also provided to indicate the particular memory segment being accessed. This information can be used by the WDP to determine the particular instruction being executed by the main processor, the memory segment being addressed, and the type of bus cycle being run at any time. This suggests that the WDP must have an elaborate bus tracking mechanism and an instruction queue similar to the main processor. In addition, the WDP should also have an address queue to store the addresses of all instructions in the queue. This information is needed for address verification of the next instruction. A block diagram of the WDP architecture is shown in Figure 2. Architectural details of the individual mechanisms are described in the following sections.

4.6.1 Execution Time Check (ETC)

The number of clock cycles required for complete execution of an instruction is the execution time of the instruction. It is measured as the number of clock cycles between two consecutive instruction fetch signals. As mentioned earlier, the BIU of the 8086 microprocessor may sometimes delay a bus access request from the EU. This delay must be accounted for when the total instruction execution time is computed. The ETC design incorporates the necessary provision for determining these lost cycles. Lost bus cycle delay is subtracted from the observed execution time before comparing the actual execution time with the expected execution times as contained in the IPT. A block diagram of the ETC mechanism is shown in Figure 3. The completion of an instruction is implied when the main processor fetches the first byte of

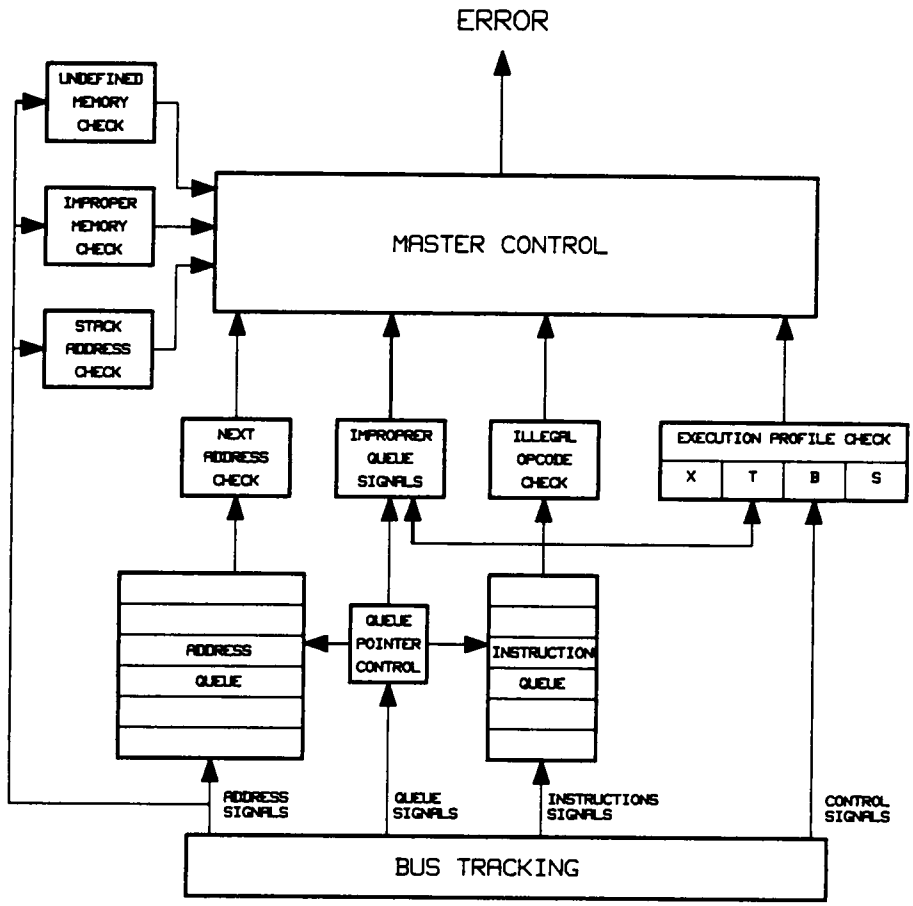

Figure 2. Architecture of the Watchdog Processor

the next instruction. The comparator circuit is activated after the completion of an instruction.

4.6.2 Byte Count Check (BCC)

The queue status signals for the 8086 microprocessor indicate when a byte of an instruction, either the first or a subsequent byte, is taken from the queue. A count of all bytes taken from the queue between two consecutive fetches of a "first" byte may be used to determine the byte count of a particular instruction. This is implemented as a counter that increments on a subsequent byte read signal and preloads a '1' on the first byte read signal. The block diagram of the BCC mechanism is shown in Figure 4. The comparator circuit is activated upon the completion of an instruction.

4.6.3 Data Transfer Check (DTC)

A sequence of read or write signals is generated by the 8086 microprocessor whenever it transfers data to or from a peripheral. These signals are different from those generated during an instruction fetch cycle where only program memory is read. Thus, for an instruction that requires a memory or I/O access, the 8086 microprocessor generates the appropriate bus cycle. By counting the number of times bus access cycles are generated, the number of data transfers can be determined. The comparator is enabled following the completion of an instruction. This signal is also used to reset the counting mechanism. A block diagram of the DTC mechanism is shown in Figure 5.

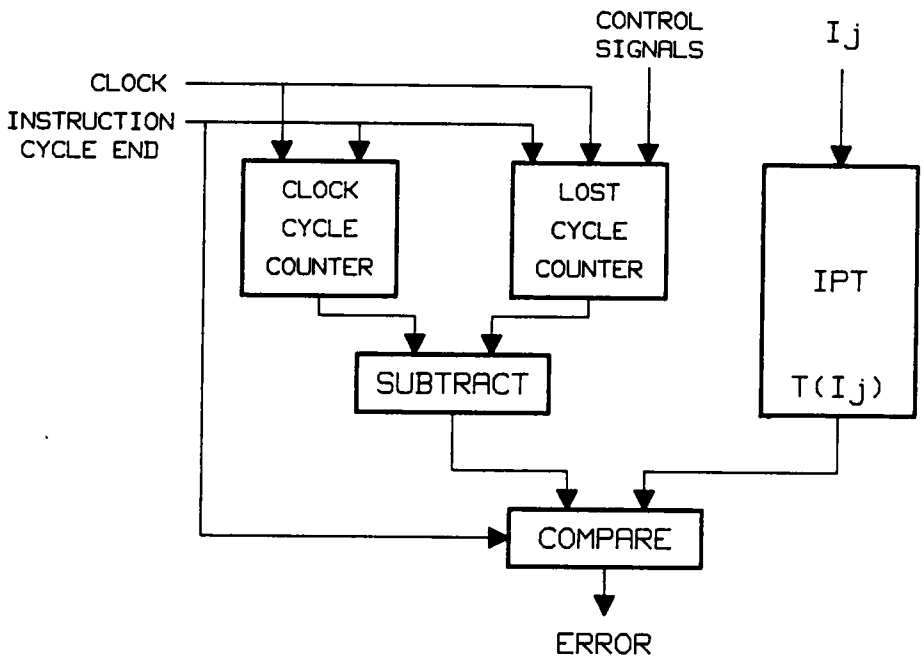

Figure 3. Execution Time Check

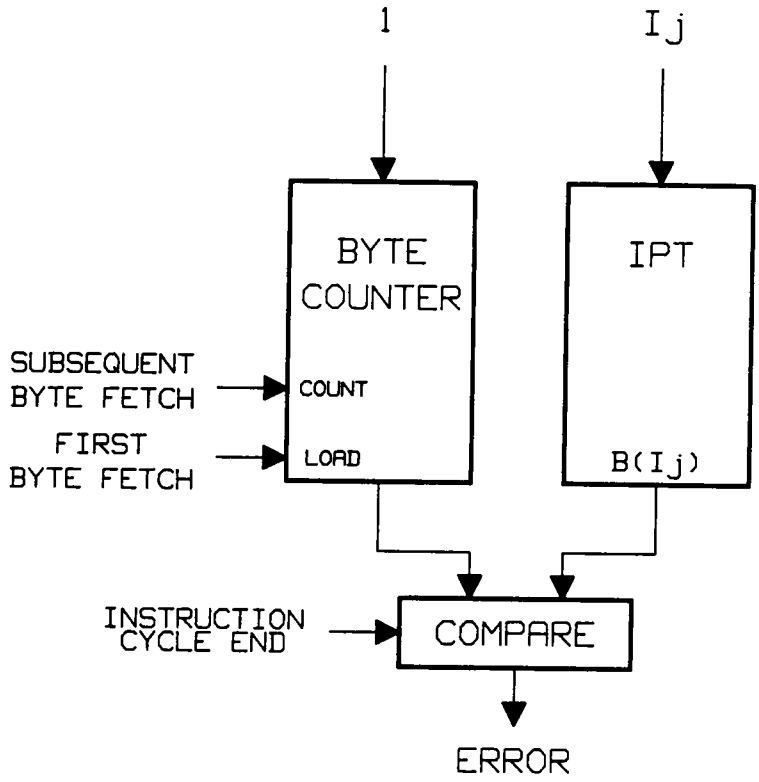

Figure 4. Byte Count Check

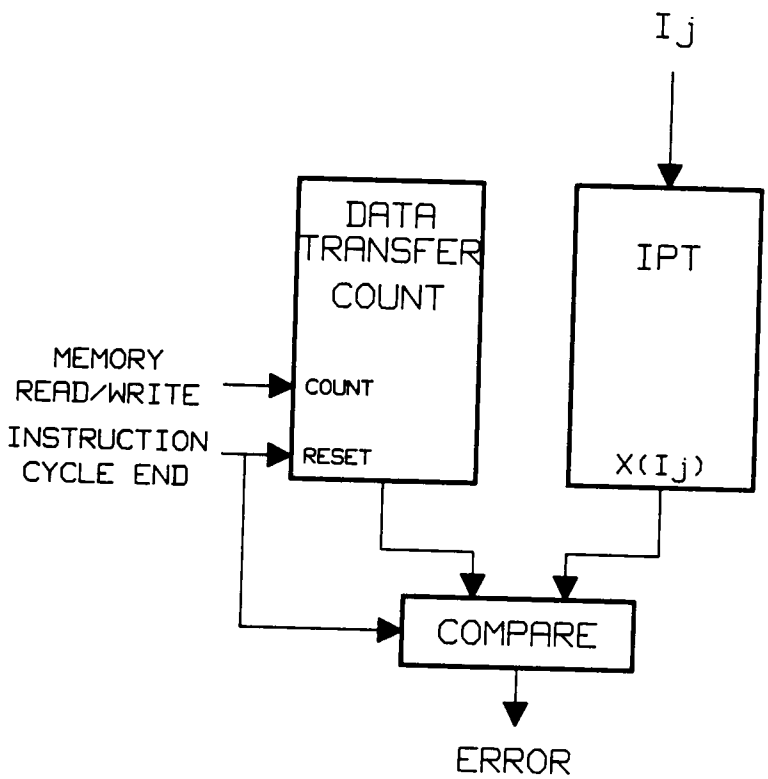

Figure 5. Data Transfer Check

4.6.4 Data Size Check (DSC)

An instruction that requires data transfers performs either a one or two byte transfer in the 8086. Exceptions to this rule are the string instructions. During each memory access, the address lines of the 8086 namely, *bhe** and *a0**, are decoded to determine the number of bytes transferred. The 8086 can carry out one of the four possible memory transfers during the execution of an instruction; a two byte transfer (bhe*,a0* = 0,0), transfer upper byte to/from an odd address (bhe*,a0* = 0,1), transfer lower byte to/from an even address (bhe*,a0* = 1,0), or no transfers (bhe*,a0* = 1,1). This is implemented as a simple multiplexor in the WDP. As the 8086 generates bus access cycles during the execution of an instruction that requires memory or I/O accesses, the monitoring circuits are designed to monitor bhe* and a0* signals to determine the size of the data transfer. The completion of the instruction activates the comparator and resets the data size count. A block diagram of the DSC mechanism is shown in Figure 6.

4.6.5 Next Address Check (NAC)

In addition to the instruction queue, the WDP is also provided with an address queue. Thus, the address of an executing instruction is always known. For non-jump instructions, the known byte count of the instruction, i.e., $B(I_j)$, is added to its address to obtain the address of the next instruction. When the next instruction is fetched, its actual address is compared with the next address calculated earlier. If there is a discrepancy, the addressing error is detected immediately. In the case of a jump instruction, a different approach is needed. Since an unconditional jump instruction is

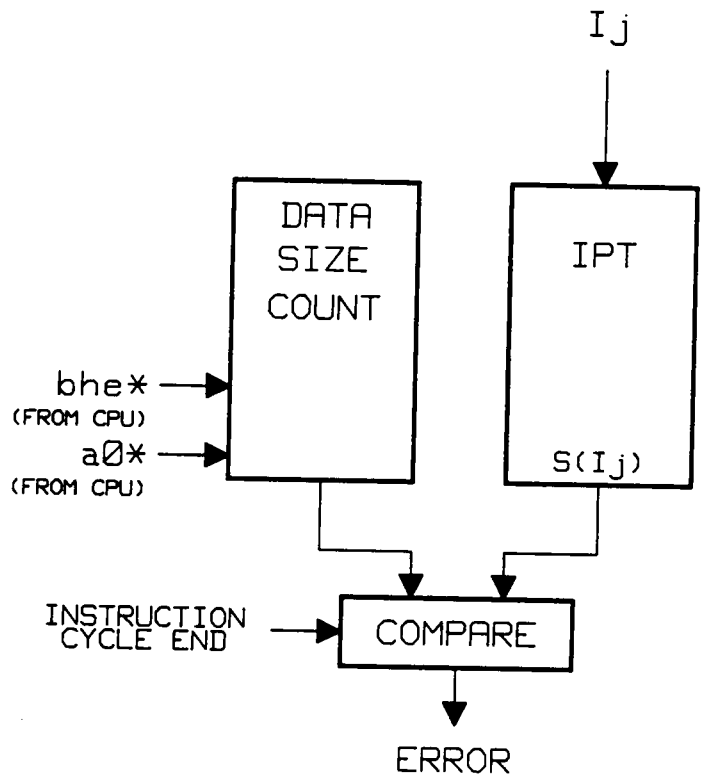

Figure 6. Data Size Check

always taken, the next address is checked to ensure that a jump was taken. Many instructions include the destination address in the instruction word following the opcode. If the destination address is included in the jump instruction, the next address is checked to ensure that program execution resumed at the proper address. If the destination address is not available because it depends on some data value in a register or a memory location, the only available option is to verify that a jump was taken. The destination address cannot be verified in this case.

A conditional jump instruction is taken depending upon the status of a condition flag. For this type of instruction, if the jump address is included in the instruction, both the next sequential address and the jump address are calculated by the WDP. The next instruction executed must come from one of these two addresses. An error is detected if a jump to an address that is different from the two calculated addresses is made. However, this mechanism will not detect an error if a incorrect branch is made because the condition flag was in error. The algorithm for this scheme is described in Section 4.4. A block diagram of the NAC mechanism is shown in Figure 7.

4.6.6 Illegal Opcode Check (IOC)

A list of all valid opcode patterns is available in the IPT. An opcode that cannot be found in this list by the WDP is declared an illegal instruction. The IPT can be implemented as a PLA so that all illegal opcodes can be detected in one cock cycle. Additionally, the coding scheme of the instruction set of the 8086 makes it possible to implement the IPT as a very compact PLA. This mechanism can only detect errors that occur in program memory or on the system bus. An illegal opcode produced due to an upset in the instruction register of the main processor cannot be detected by

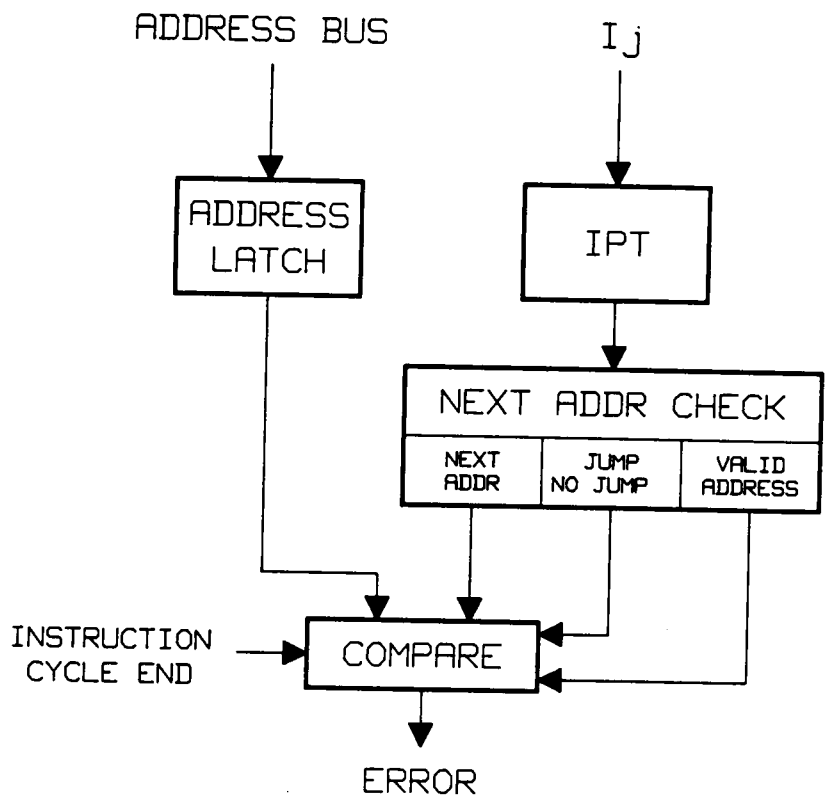

Figure 7. Next Address Check

this mechanism because the register is not visible to the WDP. However, the illegal opcode in turn produces other errors that can be be detected by the WDP. An implementation of the IOC mechanism is shown in Figure 8.

4.6.7 Stack Address Check (SAC)

This mechanism is similar in concept to the NAC mechanism except that there is no ambiguity in the next address values as in the case of conditional jumps. The status signals s3* and s4* of the 8086 microprocessor indicate the segment register being used for a particular memory access. These status signals also indicate if a stack location is being addressed. The stack operations increment or decrement the stack pointer in multiples of two. The exact multiple can be found by partially decoding the instruction. In this implementation of the WDP, the increment or decrement value is stored in the IPT. The WDP maintains a copy of the stack pointer. Whenever the main processor performs a stack operation, the WDP captures the address of the stack location, as indicated by the stack pointer of the main processor, from the address bus and compares it with its own copy of the stack pointer. An error is indicated if the two values do not match. The WDP then appropriately increments or decrements the value of its copy of the stack pointer. The block diagram of the SAC mechanism is shown in Figure 9.

4.6.8 Undefined Memory Access (UMA)

An address where no physical memory exists is called an undefined memory location. All memory and I/O device select signals are fully decoded so that the de-

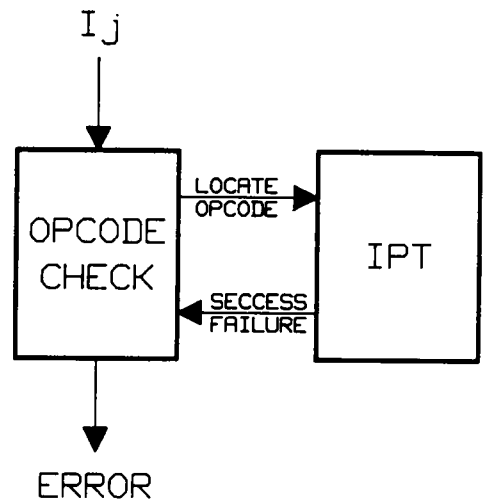

Figure 8. Illegal Opcode Check

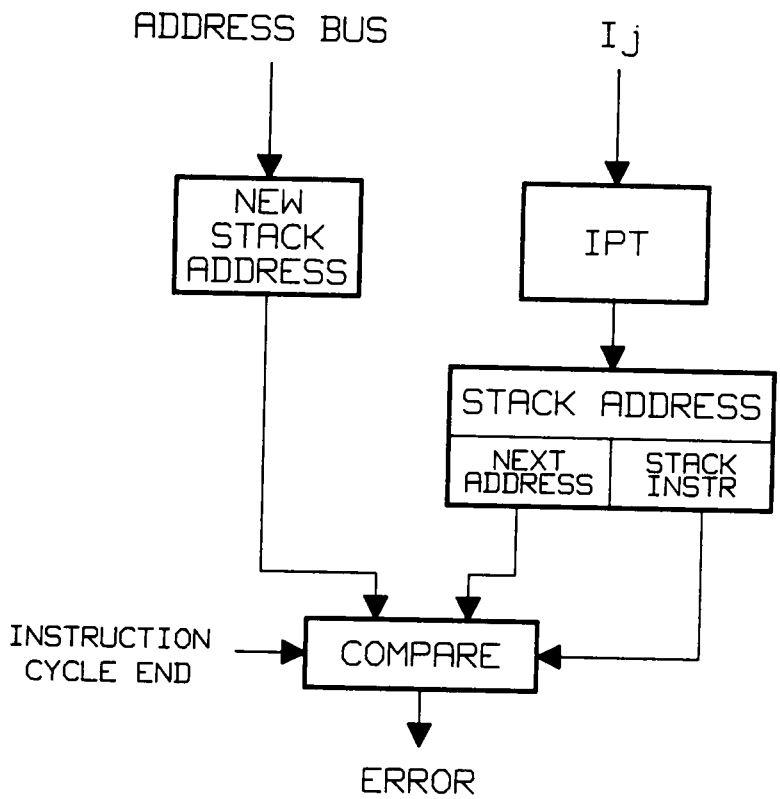

Figure 9. Stack Address Check

vices have unique addresses. In order to distinguish between a valid memory address and an undefined memory location, the data bus is connected to pull-up or pull-down resistors. This has an effect of keeping a prespecified data value on the bus if it is not being driven by any device. This is called the biasing of the bus as it effectively puts a default value on the bus. In order to detect an access to an undefined memory location, the default value on the bus is selected to be the opcode pattern of the reset instruction. The effectiveness of all these schemes described in this chapter is demonstrated through simulation. In the simulation experiments, an access to an undefined memory location returns a special pattern, indicating an error. The block diagram of the UMA mechanism is shown in Figure 10.

4.6.9 Improper Queue Update Signals (IQS)

An attempt to write a new value in the instruction queue of the 8086 when it is full or to read from it when it is empty is exposed by observing the value of the queue pointer in the WDP. As mentioned in Section 4.5 earlier, the value of the instruction queue pointer must always be within a specified range for proper operation. Whenever an access to the instruction queue is made, the queue pointer value is checked first to verify that it is within these bounds. It must be noted that if the queue pointer is perturbed so that it points to an incorrect but valid location within the queue, the resulting error will go undetected by the IQS mechanism. However, the next instruction executed by the main processor will be different from the one pointed to by the instruction queue pointer in the WDP. This is essentially a control flow error due to a very small jump in the sequence of instructions. An error may be detected after the instruction is executed if the two instructions have different execution profiles. The block diagram of the IQS mechanism is shown in Figure 11.

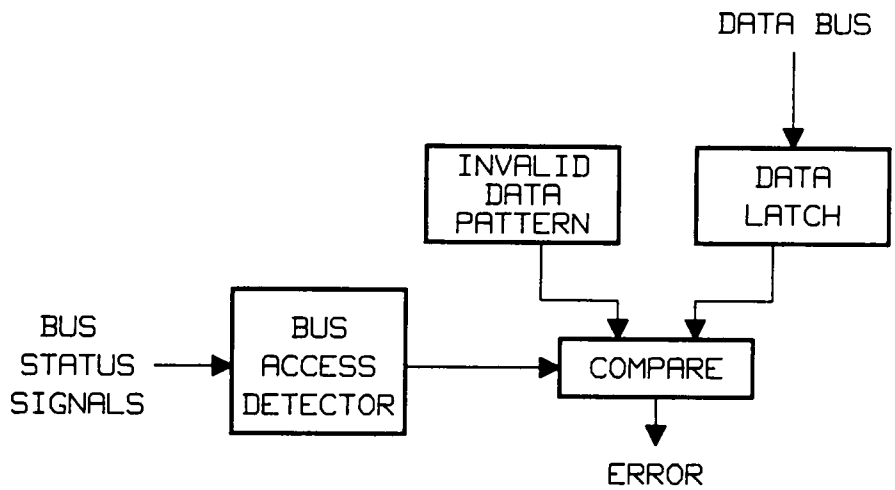

Figure 10. Undefined Memory Access Check

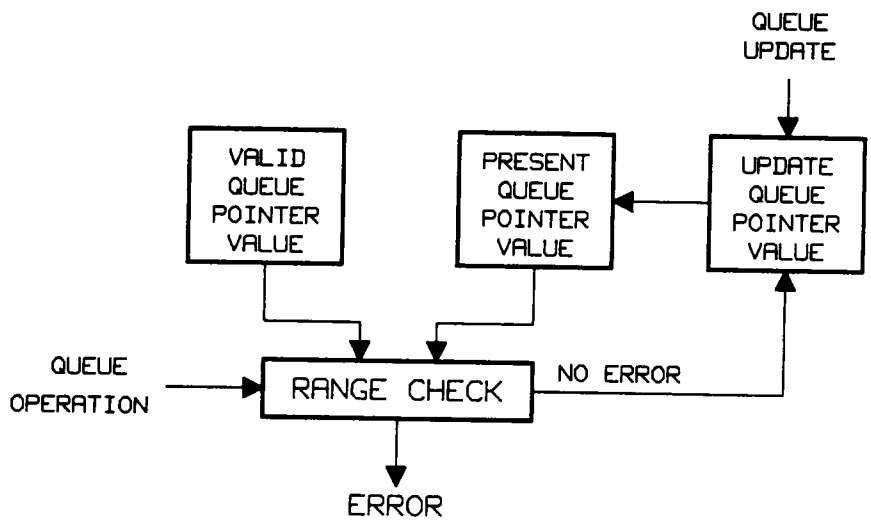

Figure 11. Improper Queue Update Signals Check

4.6.10 Improper Memory Access (IMA)

The 8086 microprocessor addresses data and instruction memories differently. It uses the DS segment register for calculating the physical address of data and CS segment register for determining an instruction byte location. An I_i/μ fault can cause the microprocessor to use an incorrect segment register or make it access instruction memory when an access to data memory is required. This is detected by observing the segment register used for memory access during execution of an instruction. By noting the type of instruction executing on the main processor the WDP determines the type of bus access cycles that may be required. Therefore, the WDP can detect an error if an incorrect segment register is used for accessing a data or memory location. The block diagram of the IMA mechanism is shown in Figure 12.

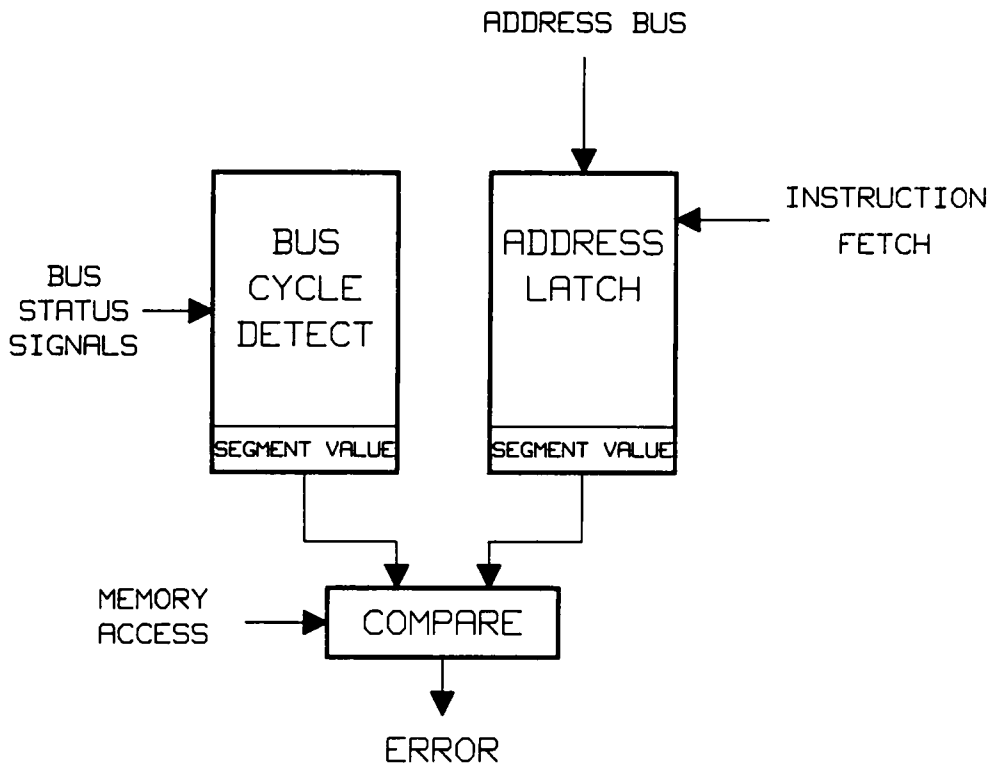

Figure 12. Improper Memory Access Check

Chapter 5

Concept Verification

In order to demonstrate the effectiveness of the proposed concurrent processor monitoring (CPM) scheme, a WDP is implemented for the 8086 microprocessor. A small prototype system is designed to assess the performance of the WDP. The details of the testing and verification process are described in this chapter.

5.1 The Need for Simulation

Transient faults can be artificially induced in a digital system by either exposing it to high energy radiations or introducing transients in its power supply [15-17,20-26,40,41]. Neither are controlled events and introduce random upsets whose time and place of occurrence are non-deterministic. Computer simulations are used to overcome this lack of specificity in time and location of transient faults. The effect of an SEU on a digital the system can be modeled in a simulation environment by injecting transient faults at a particular time and location in a precise manner. Another advantage of computer simulation is that any fault can be recreated

any number of times to study its effects in detail. The performance of the main processor and its WDP after the injection of a fault, especially its internal signals, can be observed by using the trace option of the simulator.

In light of the above mentioned arguments, computer simulations are used to study the the effect of an SEU on the prototype system and the performance of the WDP. During each simulation run, one transient fault is injected while the main processor executes a test program. A large number of simulation runs are required to collect data that is statistically significant.

5.2 Level of Simulation

The next step is to decide upon the level of simulation. Gate level models have been extensively used in simulating SSI and MSI circuits. This approach requires detailed gate level information about the circuit that is usually not available for more complex LSI and VLSI circuits. Even if this information were available, the complexity of many VLSI devices such as microprocessors etc, makes modeling at the gate level infeasible due to high cost of writing the models and the computational resources required to run the simulation. Many computer hardware description languages such as VHDL, HILO, ISP′ or GSP-2, can model circuits at a higher level of abstraction called the functional level. At this level, the device is modeled by its register transfer characteristics wherein a detailed knowledge of its gate level implementation is not required [88]. Although functional modeling and simulation has the disadvantage that the link between the functional model and the actual implementation of a circuit is not very strong, it does have the benefit of reducing the programming effort and

simulation time to an acceptable level [88-90]. The HILO simulation system was used for the purpose of this dissertation. The HILO hardware description language can describe digital circuits at at a functional as well as a gate level [91].

5.3 Simulation Environment

A model for the SEU environment is required to simulate the occurrence of errors. In an SEU environment, transient faults produced in the digital system that are randomly distributed in time and place. Since observing the performance of the system in the face of SEUs is the primary interest, it is not necessary to have an exact model of the SEU environment. Instead, a simple pseudo-random number generator can be used to choose the time and place of the transient fault. The selection of the time of injection of an SEU can be done by triggering the fault activation logic at a time specified by the pseudo-random number generator.

An SEU primarily affects the storage elements such as flip flops in a digital system. In a functional description of a system, the storage elements corresponding to data and control registers are explicitly described while those in control section of the processor are implied. The pseudo-random number generator is used to select the location of a transient fault. The choice of fault targets can be limited to a finite number of storage elements and control structures in the microprocessor model. These potential fault sites include the program counter, processor register file, the stack pointer, microprogram sequencing registers, instruction registers, interrupt registers, control registers, and other holding registers. In a total dose radiation study of an 8086 microprocessor it was observed that 70% of the failures were in the

program counter and the interrupt logic while another 20% were in the instruction execution logic [92]. Although these are total dose radiation results, they do indicate the relative sensitivity of these registers to faults. Therefore, the above mentioned possible fault targets can be used for fault injection experiments.

5.4 Experimental Configuration

The model of the prototype test system used to demonstrate the proposed CPM scheme is described here. As the emphasis is on those transient upsets occurring internal to the microprocessor, for the purpose of these experiments, it is assumed that the peripheral devices are either immune to SEUs or have error detection and correction mechanisms built into them.

5.4.1 Prototype Test System

The test prototype has an 8086 microprocessor operating in its maximum mode. In this mode, the microprocessor emits several control signals that can be decoded externally to determine the type of bus cycle being run at any time. All part numbers given in this discussion are from Intel Corporation, Inc. An 8288 bus controller uses these signals to generate appropriate handshake signals for addressing the devices on the bus. During a bus cycle, the address of external devices is saved in a bank of 8282 latches. Data is exchanged between the 8086 and other devices through a set of 8286 transceivers. These latches and transceivers effectively demultiplex the bus into separate address and data buses. Three pairs of memory banks are provided for storing instructions and permanent data. Two pairs of memory banks are

ROMs that are used for storage of program instructions and data. The third pair consists of RAMs and is used during program execution as scratch pad memory. The address of all devices on the bus is completely decoded so that there are no multiple images of a device in the address space. The circuit diagram of the prototype system is shown in Figure 13.

A custom designed WDP, that implements the error detection scheme described in previous chapters, is interfaced with the prototype system. It is completely transparent to the rest of the system. A functional model of the WDP was written in the HILO modeling language. The WDP is connected to the address, data, and control busses. As it is used as a monitoring device, it does not have an address on the bus and cannot be accessed by any device. Thus, it is completely transparent to the rest of the system. The WDP is synchronized with the 8086 and operates with the same clock signal. The WDP captures the addresses and instructions appearing on system bus to maintain an instruction queue similar to the one in the 8086. In addition, the WDP also monitors the status and control lines of the 8086 to determine the type of bus cycle being run by the processor and the status of the instruction queue. The error signal generated by the WDP is connected to the maskable interrupt input of the 8086 processor. The detailed listing of this model is provided in Appendix A. The net list of the prototype system depicting its interconnection configuration is shown in Appendix B.

5.4.2 Testbed Setup

The testbed consists of the HILO simulation environment running on an HP-350 computer. All components of the test hardware are described by a software model at ei-

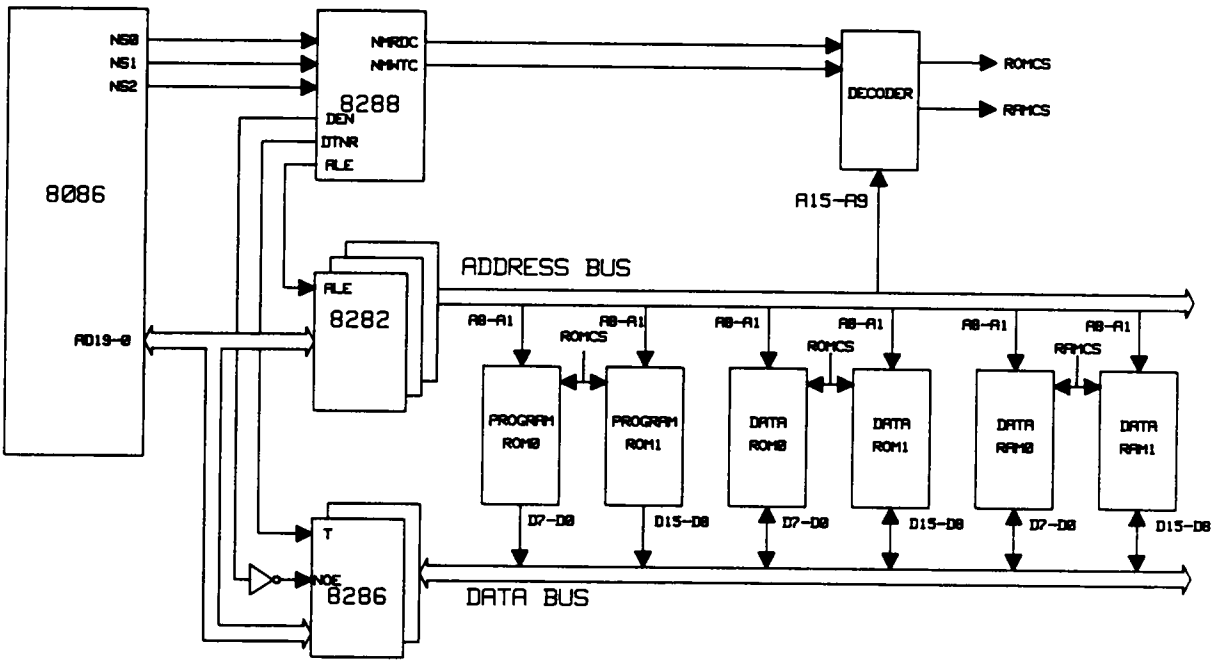

Figure 13. The Prototype Test System

ther the functional or the gate level. The model of the 8086 microprocessor was modified to facilitate the injection of SEUs into the microprocessor. The simulation environment for this experiment is shown in Figure 14. A program written in C uses a pseudo-random number generator to create the waveform files for successive simulation runs. The simulation is controlled by a simulation control file that links the stimuli provided by the waveform file to the prototype system and specifies the particular signals to be monitored during simulation. The system model file contains information about the components used in the system and the circuit topology. The individual components are in separate files that are linked together at simulation time. External stimuli for the system are provided through a waveform file which also includes information about injected faults. The simulation output is captured in a log file. A translator program written in C is used to extract pertinent data from the log file and put it in a format suitable for analysis. Quattro, a spread sheet program is used for statistical analysis of data.

5.4.3 Test Program

An important consideration is the nature of the test software to be executed on the processor during simulation. The choice of test program is influenced by the following factors:

1. Ideally, a test program should check all possible instructions of a microprocessor. However, there are practical considerations that limit this exercise. For instance, the 8086 microprocessor has about 14,000 opcodes in its instruction set. It would require a very long time to test every instruction, let alone doing it

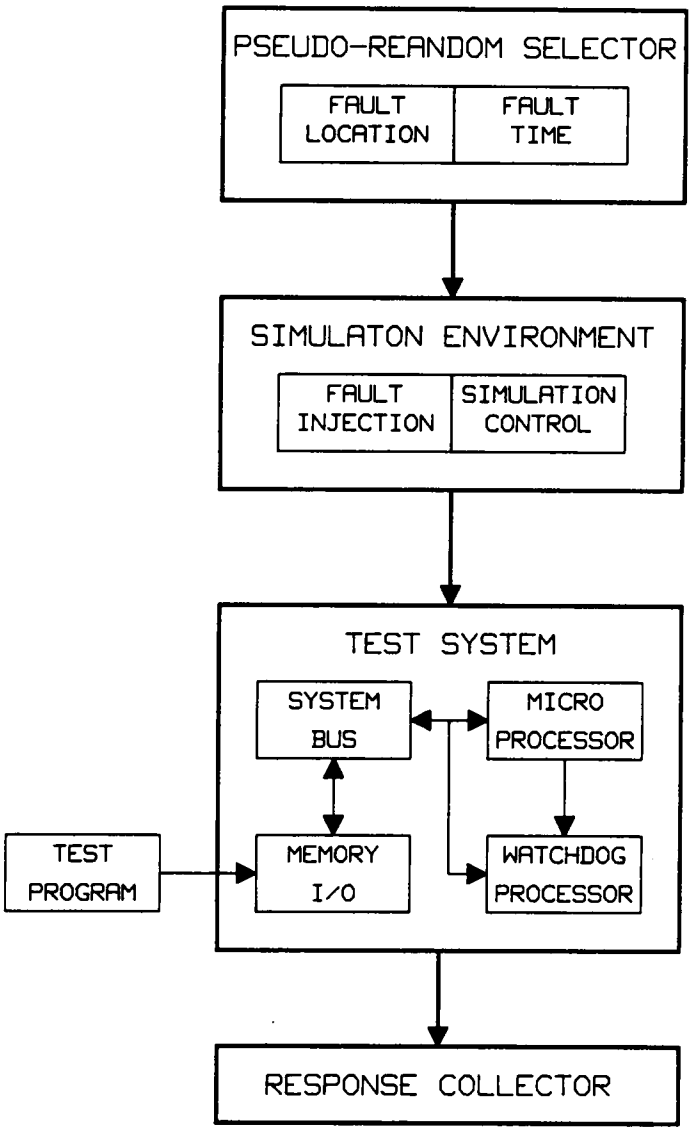

Figure 14. The Simulation Environment

on a simulator which is several order of magnitude slower than the actual processor.

2. Different types of instructions exercise different parts of a microprocessor and some sections may be more sensitive than others. Therefore, the test program must be so designed that it is similar to the programs that will run on the actual system.

In the light of the above arguments, a small test program was developed that exercises a number of address and data manipulation operations that were chosen so as to include at least one instruction from each instruction type described in the user manual of the 8086 microprocessor [93]. The main program calls a subroutine that computes the sum of an array stored in memory. The first item in the memory is the length of the array. The sum is passes back to the main program which in turn counts the the number of ones in the word representing the sum. This program exercises memory read/write operations, stack operations, arithmetic and logic operations, and subroutine calls. A listing of this test program is given in Appendix C. The output of a typical run for fault-free and fault simulation cases are privided in Appendices D and E respectively.

5.4.4 Injected Fault Types

The type of fault to be injected in a simulation run is coded in the waveform file. A C program generates the waveform file by using a pseudo-random number generator to select the type, the time, and the place of the injected fault. Injected faults are chosen from a list of faults types shown in Table 2. A discussion about the fault types shown in this table is given in section 3.4. The I_i/I_o faults are not included in this list

Table 2. Fault Types

Fault Type	Fault Name
0	l_j/μ Fault
1	l_j/ϕ Fault
2	l_j/l_k Fault
3	$l_j/l_j + l_k$ Fault
4	l_j/l_j' Fault
5	$l_j,l_k/l_j,l_m$ Fault
6	Stack Fault

because they are a special case of the I_j/I_k faults and are, therefore, covered. Faults are introduced so that they correspond to a change in the state of a bit in a register or control signal for one instruction cycle only. This change disappears when that register is rewritten. Only one fault is injected during each simulation run. To obtain statistically significant results, a large number of simulations are required.

Chapter 6

Performance Evaluation

After the WDP implementation of the CPM scheme is designed, tested, and verified to be operating correctly it must be shown to be effective in detecting transient faults. This is done by defining performance criteria for measuring the detection scheme's effectiveness. In this chapter, the metrics of performance evaluation are defined to assess the effectiveness of the proposed scheme. The actual performance of the CPM scheme is then determined.

6.1 Assessment Parameters

The primary requirement of a reliable system is that it should be able to tolerate faults so that its output is always correct. Since this is not always possible, a small error detection latency time is desired so as to limit the damage and initiate efficient recovery procedures with minimal performance penalty. In general, the reliability and availability measures may be considered an adequate reflection of the performance of a system. These depend upon error detection latency and error coverage

characteristics of a system. The longer a fault remains undetected, the more likely it is that it may cause more damage to the system. Once a fault is detected, an appropriate recovery action must be taken to undo the damage, if any, and resume normal operation. Hence, a small fault latency, a high fault coverage, and an efficient recovery would be the most desirable characteristics of a fault tolerant system. In addition to determining the probability that a fault is detected, the performance metrics should also measure the probability of a false alarm. A set of parameters satisfying these objectives would include the following [94,95]:

1. Probability that a fault is detected (fault coverage).
2. Probability that a fault is incorrectly indicated (false alarm).
3. Time to error detection (error detection latency).
4. Cost of reliability (overhead).

These parameters specify the most desirable characteristics of a reliable system. The improvement in reliability of a system due to incorporation of fault tolerance mechanisms must be balanced against the cost associated with these schemes. This cost can be broadly divided into hardware costs, software costs, and the performance penalty in terms of time and speed. The hardware cost mainly involves space, power, and weight of the additional circuitry. The software cost, on the other hand, is somewhat more difficult to assess. It is affected at three levels; operating system software, application software, and diagnostic software [94]. A first order estimate of software cost may be made in terms of the extra memory space needed for diagnostic programs. The performance penalty consists of two components; performance loss during normal operation and performance loss due to exception processing. The performance loss during normal operation is due to the processing required to peri-

odically save the state of the system e.g., creating a check point or reading signatures embedded in a program. If an error is detected, the system must perform a recovery operation such as restarting from the previous check point. This processing of the exception also results in performance loss.

An important point that must be addressed is the way error detection latency time is measured. The execution profile verification scheme detects faults following the completion of an instruction. Thus, a fault that occurs during the execution of an instruction can be detected only after the instruction is completed i.e., error detection latency is a function of the execution time of the instruction. This does not accurately represent effectiveness of the detection scheme if the latency is measured in clock cycles. Therefore, error detection latency is also measured in terms of instruction execution cycles.

6.2 Analytical Results

Results describing the error coverage and error detection latency of the CPM scheme can be calculated analytically for the execution profile of the instructions. Since the execution profile parameters i.e., $T(I_j)$, $B(I_j)$, $X(I_j)$, and $S(I_j)$, only monitor the instruction execution and not necessarily cover other types of errors such as control flow errors, stack errors, etc., these results give only a lower bound on the fault coverage. The following analysis considers four mechanisms only, i.e., ETC, BCC, DTC and DSC.

6.2.1 Error Coverage

The execution profile verification algorithm described earlier can detect all instruction execution errors if the product of the partitions of the set *I*, i.e., $\pi_1\pi_2\pi_3\pi_4$, has blocks that contain at most one element. For many microprocessors this may not be the case. For example, consider Intel's 8086 microprocessor which has about 14,000 different types of instructions in its instruction set. The number of blocks in the product $\pi_1\pi_2\pi_3\pi_4$ can be computed for the 8086 as follows. For a first order estimate, instructions that have data dependent execution times may be neglected. For example a *move string* instruction with a repeat prefix for the 8086 microprocessor has an execution time which is a function of the data and cannot be predetermined. In this analysis, such instructions are not included. Instructions not considered include CMPS, DIV, IDIV, IMUL, MUL, MOVS, MOVSB, MOVSW, etc. The WDP design described earlier does not cover these instructions. However, it can be incorporated in the scheme by changing the architecture of DSC and DTC mechanisms. The remaining instructions can be divided into 46 different classes according to their execution times. Hence partition π_1 has 46 blocks. An 8086 instruction can be 1 to 6 bytes in length. Therefore, partition π_2 has 6 blocks. During execution, an instruction may perform 0, 1, or 2 memory transfers. These transfers correspond to no memory operation, a read or write memory operation, or a read-modify-write memory operation for the 8086 microprocessor. Thus, partition π_3 has 3 blocks. The size of data transferred can either be one or two bytes. Hence, partition π_4 has 2 blocks. Therefore, the product $\pi_1\pi_2\pi_3\pi_4$ has 1656 blocks. Hence, on the average, there may be 14000/1656 = 8.45 instructions in a block. Instructions that are in the same block are indistinguishable from each other.

To determine error coverage, consider now the types of errors that may occur in a radiation environment. Since SEUs are the primary source of transient faults, only errors that produce a single bit change in the contents of a register/latch are considered. It can be assumed that an instruction word can change to another instruction if, and only if, the two words are at a Hamming distance of 1 from each other. Thus a 16-bit word may only change to one of 16 other words that are at a Hamming distance of 1 from it. This is shown with the help of two examples.

Example 1:

Consider an opcode from the instruction set of Intel's 8086 microprocessor.

ADD (BX) + (SI), IMM8

This is represented in machine code as 8000 (Hex). It can change to 16 other words which are listed in Table 3. Also included in the table are corresponding parameters (X,T,B,S) of each instruction. Data in the table shows that some instructions generated by the fault can be distinguished from the original instruction. These are indicated by an asterisk. Thus 13/16 = 81.25% of the errors can be detected using only these parameters.

Example 2:

As another example, consider a single byte instruction that can change to eight other possible opcodes:

PUSH ES

Its opcode in machine language is 06 (Hex). Its variants are shown in Table 4. Examining the differences in the execution profile parameters indicates that error coverage is 87.5%. That is, this technique will detect 87.5% of the variants of this instruction as an error. A similar calculation for a number of other instructions gave an overall average fault coverage of about 85% for the 8086 instruction set.

Table 3. Faults in a 16-bit Instruction

Opcode (hex)	Opcode (assembly)	T	B	X	S	
8000	ADD (BX) + (SI), IMM8	24	3	2	8	
8001	ADD (BX) + (DI), IMM8	25	3	5	8	*
8002	ADD (BP) + (SI), IMM8	25	3	2	8	*
8004	ADD (SI), IMM8	22	3	2	8	*
8008	OR (BX) + (SI), IMM8	24	3	2	8	
8010	ADC (BX) + (SI), IMM8	24	3	2	8	
8020	AND (BX) + (SI), IMM8	24	3	2	8	
8040	AND (BX) + (SI) + D8, IMM8	28	4	2	8	*
8080	ADD (BX) + (SI) + D16, IMM8	28	5	2	8	*
8100	ADD (BX) + (SI), IMM16	24	4	2	16	*
8200	ADD (BX) + (SI), IMM8	24	3	2	8	*
8400	TEST (BX) + (SI), AL (invalid)	24	4	2	8	*
8800	MOV (BX) + (SI), AL	10	2	1	8	*
0000	ADD AL, (BX) + (SI)	16	2	1	8	*
90--	NOP	3	1	0	0	*
A0--	MOV AL, MEM8	10	3	1	8	*
C0--	Not Used	-	-	-	-	*

Table 4. Fault in an 8-bit Instruction

Opcode (hex)	Opcode (assembly)	T	B	X	S	
06	PUSH ES	10	1	1	16	
07	POP ES	8	1	1	16	*
04	MOV AL, IMM8	4	2	0	0	*
02XX	ADD R8, R8/M8	3	2-4	0-1	0-8	*
0E	PUSH CS (illegal)	10	1	1	16	*
16	PUSH SS	10	1	1	16	
26	ES: (prefix)	2	1	0	0	*
46	INC SI	2	1	0	0	*
86	XCHG R8, R8	4	2	0	0	*

6.2.2 Error Detection Latency

As explained earlier, the observed and expected parameters of an instruction are compared after the completion of an instruction. Any fault that occurs during the execution of an instruction can be detected only after the instruction is executed to completion. Thus, this scheme, by design, detects errors as soon as the faulted instruction is completely executed. In most cases, the error detection latency would be less than two instruction cycles. Since some faults may produce control flow errors, the next address detection scheme is used to detect any break in the normal sequence of instructions. This can detect most errors as soon as the next instruction is fetched for execution. The exception to this includes the case when during the execution of a conditional jump instruction the control flag, that determines if the jump should be taken, is perturbed. In this situation, no error is detected as long as the program resumed at one of the two valid addresses. This can produce an inaccurate result but there will not be a loss of control error.

Another situation where a control flow error may not be detected is when the jump address has an offset which is stored in a data register. Here again, the WDP cannot determine the destination address. The only option available is to verify that a jump did indeed take place without verifying the destination address. Therefore, the average latency of next address detector depends upon the nature of the program and may be slightly higher than 2 instruction cycles. To translate this observation to a practical real-time environment, it would be necessary to determine the average length of execution of all of the instructions used in the application software. A more accurate measure would include a weighting factor to account for repeated use of certain instructions.

6.3 Experimental Results

To obtain statistically significant data, 2006 fault simulation experiments were performed. During each simulation, one transient fault is injected into the system. The time and the type of fault during a simulation run is pseudo-randomly chosen as described in section 5.4.2. The results obtained from these experiments are presented in the following sections.

6.3.1 Distribution of Injected Faults

There are four possible outcomes of a fault injection experiment: the injected fault does not cause an error i.e., the no error (NE) case; the injected fault causes an error but it is not detected i.e., the undetected error (UE) case; the injected fault creates an error which is detected i.e., the detected error (DE) case; and the injected fault did not cause an error but a fault was incorrectly indicated i.e., the false alarm case. The false alarm case was never observed and will not be discussed further. To distinguish between the NE-case and the UE-case, a fault-free simulation run was done first. This is known as the golden simulation run. The responses of the fault experiments were compared with those of the golden simulation run to make a distinction between NE- and UE-cases.

All simulation runs were done on the same testbed under identical conditions. Faults were injected during an experiment in a precise and reproducible manner. The preliminary ananlysis of the outcome of experiments indicated that not every fault produced an error. This is evident from Figure 15 that shows the outcome of simu-

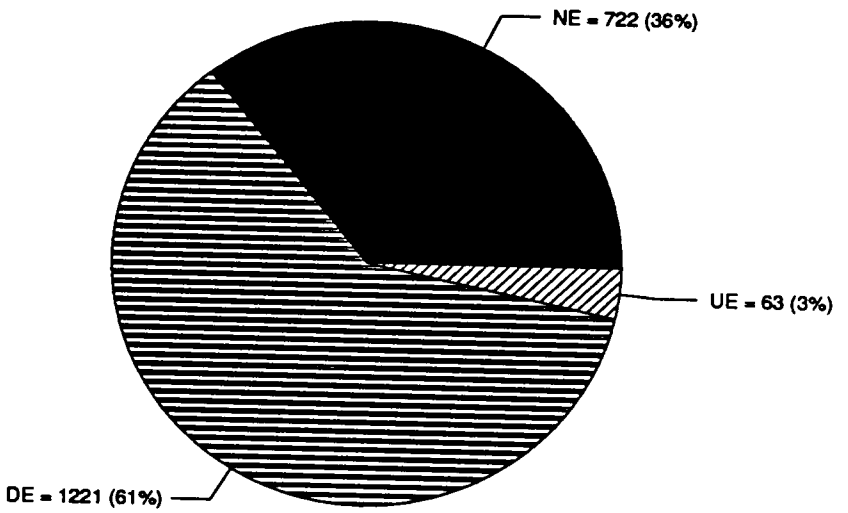

Figure 15. Simulation Results

lation experiments by classifying them into three categories defined earlier: NE-, UE-, and DE-cases. The figures indicate that a significant number of the modeled transient faults injected in these experiments do not produce an error (36%); about 61% of the faults produce errors that are detected by the proposed mechanisms; nearly 3% of the faults are not detected.

As indicated earlier, the type of injected faults is pseudo-randomly chosen. Thus, for a large sample, all types of faults are equirobable. The distribution of the types of injected faults used for these experiments is shown in Figure 16. The injected fault types shown in the figure were described in section 5.4.4. It was observed that some types of faults produced more errors than others. This is evident from Figure 17 which shows the distribution of the errors produced due to injected faults. This shows that fault types 1, 3, 4, and 5 produce most of the errors.

Initial simulation data indicated that a significant number of injected faults do not produce an error. This is especially true of an I_j/I_k fault in the instruction register. The reason for this is that the CPU reads the instruction register only during the first clock cycle of an instruction to decode it. If the I_j/I_k fault occurs after this, it will not produce an error since the CPU no longer uses the contents of the instruction register. Also, if the fault occurs just before the instruction fetch cycle, it will not affect the operation since it will be overwritten by the new instruction fetched by the CPU. Thus, for a perturbation of the instruction register to cause an error, it must occur during the first clock cycle of an instruction after the new instruction has been fetched from the instruction queue and before the CPU has decoded it for execution. This implies that in order to study the response of the WDP to I_j/I_k faults, a very large number of experiments would be required. To limit this sample size, an attempt was

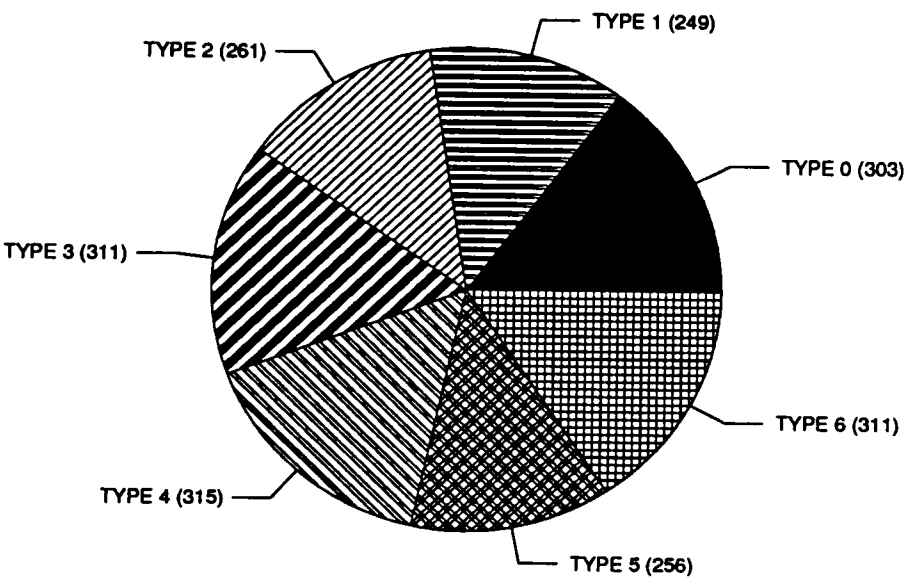

Figure 16. Distribution of Injected Faults

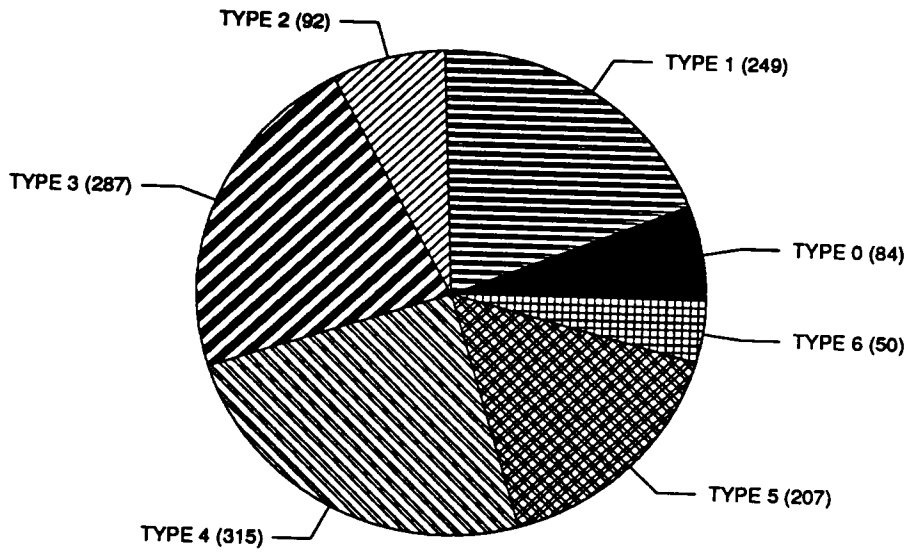

Figure 17. Distribution of Errors

made to ensure that an I_j/I_k fault is clocked into the instruction register before the CPU attempts to decode it. This is achieved by injecting the faults in the instruction queue of the processor instead of the instruction register. This will ensure that an I_j/I_k fault always produces an error. Also. this is why the number of errors produced due to I_j/I_k faults, including both detected and undetected, seems disproportionately large.

6.3.2 Error Coverage

A summary of the simulation experiments was shown in Figure 15. Initially, a test sample of 100 simulations were performed. Calculations for a 99 percent confidence interval with allowable error of less than 2% indicated that a sample size of about 2000 is required. Hence, a total of 2006 fault experiments were conducted. Of these, 1284 or 64% produced errors; 1221 or 95.1% of the errors were detected by one or more mechanisms; only 63 or 4.9% of the faults that produced errors were not detected. Thus, the CPM scheme exhibits a high error coverage.

The distribution of detected errors for every fault type is shown in Table 5. The WDP shows perfect error coverage for I_j/ϕ, I_j, I_k/I_j, I_m , and stack faults. A high coverage (in the 99th percentile) is also achieved for I_j/μ and I_j/I'_j faults. An overall fault coverage of 96.8% is achieved for all faults. The reasons for relatively poor performance of the detection scheme in case of I_j/I_k and $I_j/I_j + I_k$ faults are investigated in Chapter 8.

The overall performance of individual mechanisms is shown in Figure 18. The number at the top of the bars indicates the percentage fault coverage provided by each particular mechanism. As pointed out earlier, an injected fault can produce errors that are detected by more than one detection mechanism. This is evident from Figure

Table 5. Total Error Coverage for every Fault Type

Fault Type	Fault Name	Faults Simulated	Errors Produced	Errors Detected	Error Coverage
0	l_j/μ Fault	303	84	82	99.3%
1	l_j/ϕ Fault	249	249	249	100.0%
2	l_j/l_k Fault	261	92	59	87.3%
3	$l_j/l_j + l_k$ Fault	311	287	260	91.3%
4	l_j/l'_j Fault	315	315	314	99.6%
5	l_j, l_k/l_j, l_m Fault	256	207	207	100.0%
6	Stack Fault	311	50	50	100.0%
Total	All Faults	2006	1284	1221	96.8%

18 where the sum of error coverage percentage of all mechanisms is more than 100%. An examination of these numbers also indicates that a large number of errors are detected by just five mechanisms, namely ETC, BCC, IOC, NAC, and UMA. The reasons why these five mechanisms perform so well are explored in Chapter 8.

The performance of an individual mechanism against every type of fault is shown in Table 6. Data in Table 6 also implies that a fault may be detected by more than one mechanism. Since individual mechanisms have different error detection latencies, the recovery procedure can be initiated by the mechanism that is first to detect the error. Also, there is a good chance that a fault that escapes detection from one mechanism is detected by another mechanism.

6.3.3 Error Detection Latency

Average error detection latency for all detection mechanisms in terms of clock cycles and instruction cycles is shown in Figure 19 and Figure 20, respectively. The numbers at the top of each bar in Figure 20indicate the standard deviation for error detection latency values of the individual mechanisms. As pointed out earlier, a better sense of the effectiveness of the proposed error detection mechanisms can be obtained by measuring the error latency in instruction cycles instead of clock cycles. Therefore, in following discussion, the error detection latency is measured in instruction cycles only. The average error detection latency for all mechanisms is 1.47 instruction cycles. A low error detection latency value is useful because it limits the propagation of an error.

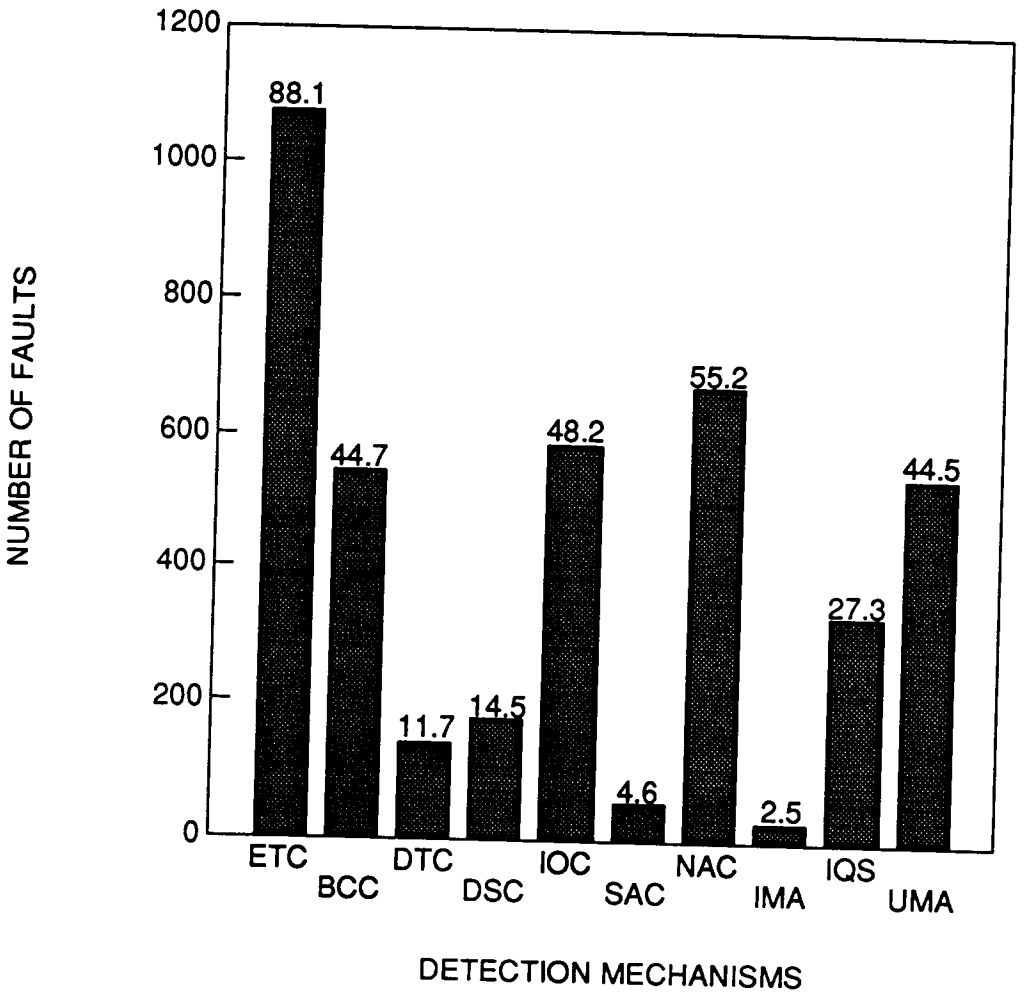

Figure 18. Performance of Individual Mechanisms

Table 6. Error Coverage of Individual Mechanisms for every Fault Type

Fault	Total	Fault Mechanisms									
Type	Faults	ETC	BCC	DTC	DSC	IOC	SAC	NAC	IMA	IQS	UMA
0	82	51	26	29	20	17	1	31	25	12	18
1	249	249	122	2	2	114	0	113	0	170	67
2	59	58	36	18	19	17	0	19	0	2	2
3	260	183	134	51	86	78	4	78	5	117	79
4	314	310	116	30	33	170	1	199	0	29	187
5	207	175	112	13	17	142	0	184	0	3	140
6	50	50	0	0	0	50	50	50	0	0	50

The latency of each individual mechanism in terms of instruction cycles for every type of fault, is shown in Table 7. This data shows that mechanisms ETC, IMA, IQS, and UMA detect most of the errors in less than two instruction cycles. Almost all errors are detected within three instruction cycles with the exception of stack faults. The reason why stack faults are detected with much longer latency are explored in Chapter 8.

6.4 Hardware Overhead of the CPM Scheme

In order to estimate the hardware overhead for incorporating the CPM scheme in a system, the WDP must be implemented in silicon. However, for the purpose of this analysis, a first order measure of the complexity of the WDP can be obtained by estimating the area of the WDP chip. To estimate the area of the WDP chip, it is assumed that most of the circuitry is implemented as PLAs. The structure of a PLA is very regular and its area can be estimated easily.

The functional description of the WDP is given in Appendix A. A sizable portion of this model is represented by 'case' statements in the model. The size of the PLA implementing this statement is determined as follows. First consider the structure of a typical case statement construct. A case statement monitors a number of signals included in its sensitivity list. A value in this sensitivity list is decoded to select one of many blocks in the body of the case statement. Every block has one or more statements called the action list. When implemented as a PLA, the number of signals in the sensitivity list is the number of inputs to the PLA, the number of blocks in the case statement are the number of product terms in the PLA, and the number of

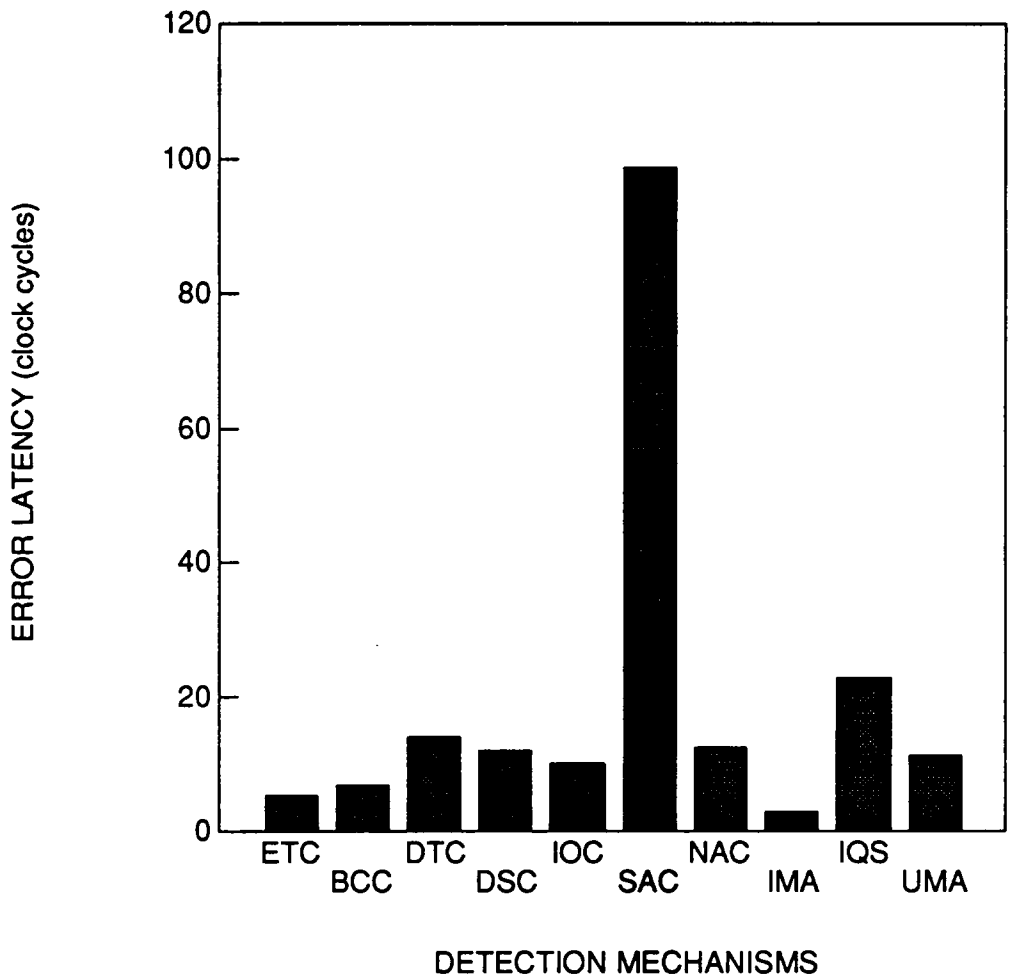

Figure 19. Error Latency of Individual Mechanisms (in clock cycles)

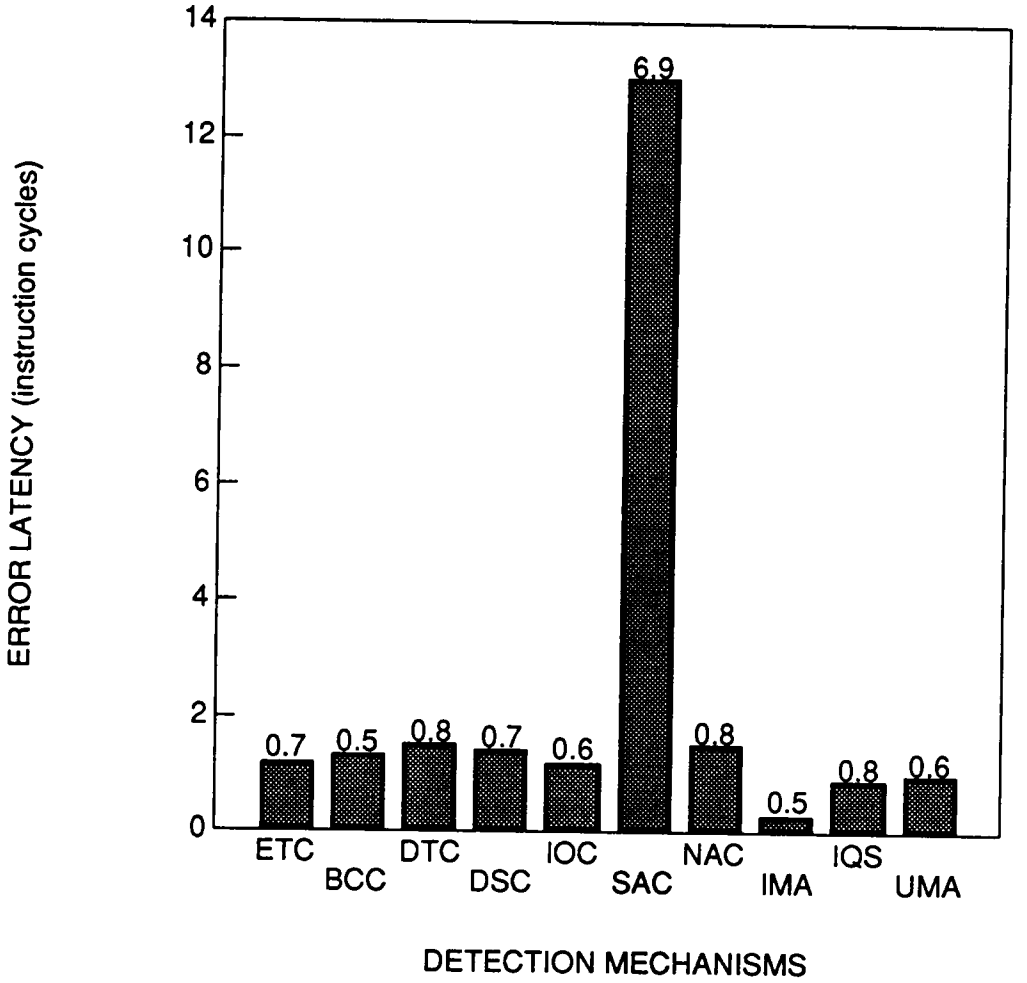

Figure 20. Error Latency of Individual Mechanisms (in instruction cycles)

Table 7. Error Latency of Individual Mechanisms (instruction cycles)

Fault	Total	Fault Mechanisms									
Type	Faults	ETC	BCC	DTC	DSC	IOC	SAC	NAC	IMA	IQS	UMA
0	82	1.57	3.00	1.07	1.30	2.24	4.00	2.52	0.36	1.58	0.83
1	249	1.09	2.06	3.00	3.00	2.16		2.62		2.02	1.96
2	59	2.52	1.97	2.56	2.58	2.06		2.32		1.50	1.00
3	260	1.28	1.13	1.73	1.36	1.40	3.00	1.97	0.00	0.77	0.85
4	314	1.57	3.00	2.47	2.55	2.14	0.00	2.68		1.93	1.14
5	207	1.94	2.79	2.92	2.82	1.89		1.89			1.36
6	50	15.7				14.7	13.7	15.7			13.7

unique signals generated in all action lists are the number of outputs of the PLA. The area of an nMOS NOR-NOR PLA having i inputs, o outputs, and p, product terms is given by the following equation [96]:

$$\text{Area of PLA} = (16i + 8o + 50) \times (8p + 77)\lambda^2$$

The area of the instruction decoding logic of the WDP can be estimated by considering the case statement that is used to model it. There are 8 signals in the sensitivity list of the case statement, 107 action blocks, and it generates 55 unique signals. Thus, $i = 8$, $p = 107$, and $o = 55$. The estimated area of the PLA implementing this statement is calculated to be $576594\lambda^2$. A similar calculation is performed for all other case statements in the model. The net area estimate for all PLAs implementing every case statement in the WDP model is $983604\lambda^2$.

Besides the case statement, the WDP model also contains other constructs such as if-then-else, storage elements, and when statements. For a rough estimate, the if-then-else statement can be thought of as a case statement with $i = 1$ and $p = 2$. The number of outputs is different for every statement. For these calculations, a mean value of $o = 4$ is assumed. Thus the area associated with the implementation of an if-then-else statement is $9114\lambda^2$. As there are about 25 if-then-else statements in the WDP model, the total area estimate for these statements is about $227850\lambda^2$. The WDP model employs a significant number of storage elements; 411 1-bit registers in all. Assuming a 1-bit register to be $500\lambda^2$ in area, the total silicon area used or all storage elements is about $205500\lambda^2$.

Adding all the area estimates computed above, the total silicon surface required is about $1416954\lambda^2$. Assuming a 50% overhead for control circuitry, routing and I/O pads, the total WDP chip area is about $2125431\lambda^2$. If implemented in 2 *micron* nMOS

technology, the actual area of the WDP chip would be about 8501724 *micron*^{2}. The area of the 8086 microprocessor is about 33387000 *micron*^{2} (225 mil X 230 mil). Thus, the WDP chip is about 25% of the 8086 microprocessor in terms of silicon area. It must be pointed out that the above calculations are not only a rough estimate of the area of the WDP chip and an accurate figure can be made by actually implementing the chip in silicon.

6.5 Reliability

The probability that a given system will perform its required function under specified conditions for a specified period of time is the reliability of the system. The reliability of a system can be improved by using highly reliable components that form the building blocks of the system. In fault tolerant systems, reliability of a system is also improved by incorporating error detection and correction circuitry. This enhanced reliability can be assesses by comparing the reliability of the system operating with and without the fault tolerance mechanisms.

The improvement in reliability of the prototype system provided by the CPM scheme can be determined analytically. A simple error model has been assumed for this analysis. The following assumptions have been made:

1. The system consists of an 8086 microprocessor, the necessary interface logic, data and program memory, and the WDP. However, faults are considered for the 8086 only.
2. All system components are assumed to have constant and independent failure rates.

3. The failure rate of a hardware unit is a proportional to its gate count.

4. All detected errors are assumed to be recovered from.

Let $R(t)$ be the reliability of the 8086 microprocessor, i.e., $R(t)$ is the probability that an error will not occur before time t. For constant failure rate, the reliability of the processor without the WDP is:

$$R(t) = e^{-\lambda t}$$

Let E represent the error coverage of the WDP and F be the relative size of the WDP with respect to the microprocessor. Then reliability of the WDP is $R^F(t)$. In addition to the WDP, there is overhead associated with the recovery hardware also. This overhead has been neglected to keep this analysis simple. Thus reliability of the entire system is:

$$System\ Reliability = R(t)R^F(t) + (1 - R(t))R^F(t)E$$

As mentioned earlier, the error coverage of the CPM scheme is 98.6%. Thus $E =$ 0.968. The relative size of the WDP is determined to be 25% of the 8086 microprocessor. Hence $F = 0.25$. For an error rate of 1 SEU/day, the reliability of the system with and without the WDP is shown in Figure 21. The graph indicates that the CPM scheme provides a significant improvement in overall system reliability.

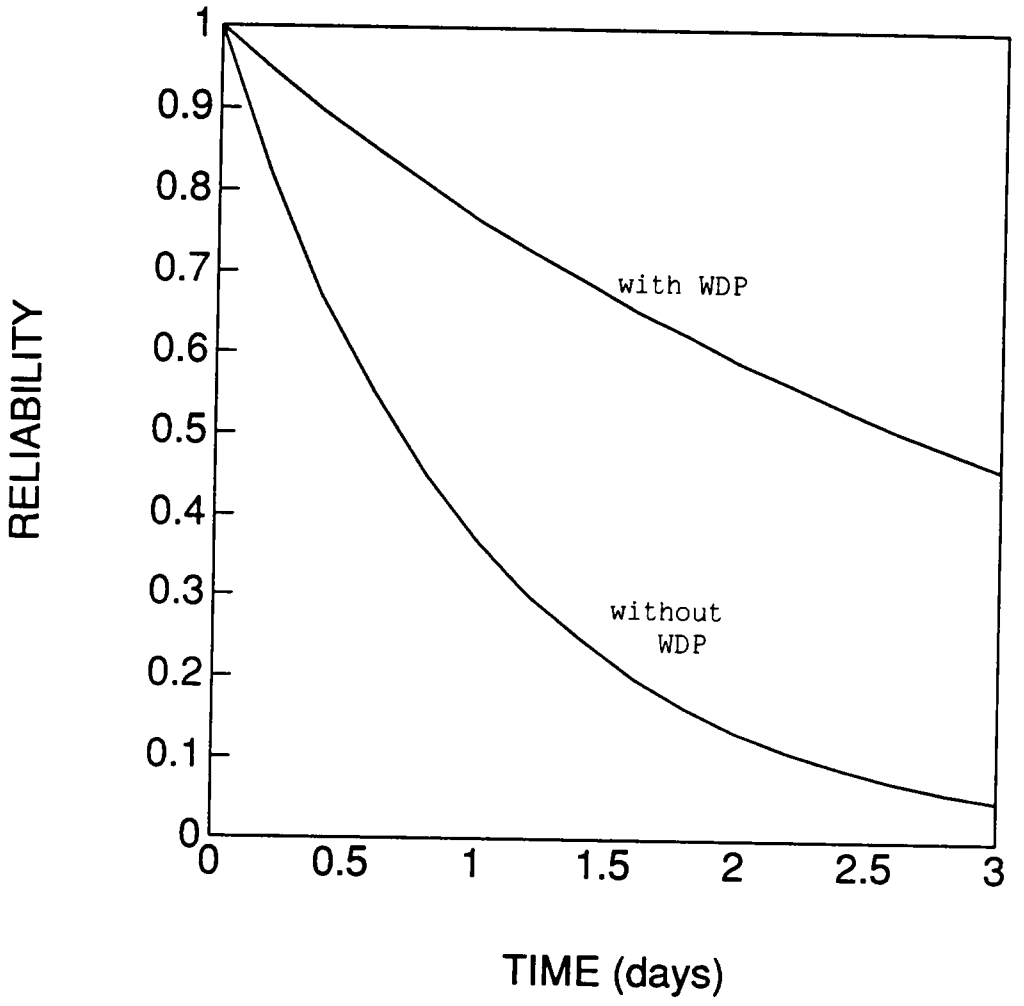

Figure 21. Improvement in System Reliability

Chapter 7

Analysis of Results

The experimental results presented in Chapter 6 indicate that a significant number of the 2006 injected faults do not produce an error (36%) in the 8086. This is explained by the fact that a transient fault, occurring in a section of the microprocessor that is not currently being used, does not affect the execution of the instruction. Since a transient fault has no permanent effect, the faulted register performs properly when used later. About 61% of the injected faults produced an error and were detected. A small number of faults, which resulted in errors, were not detected (3.1%). Thus, an overall error coverage of 96.8% was achieved. A detailed analysis of every type of fault against every mechanism for error coverage and detection latency is presented in the following sections.

7.1 Error Coverage

The performance of individual transient fault detection mechanisms is shown in Figure 18 in Chapter 6. The number on top of each bar represents the percentage of

errors detected by the corresponding mechanism. The figures indicate that a large number of the errors are detected by only five mechanisms. These are: ETC (88.1%), BCC (44.7%), IOC (48.2%), NAC (55.2%), and UMA (44.5%). Of these mechanisms, ETC, BCC, and IOC are primarily directed at detecting execution errors while NAC and UMA at control flow errors.

The data implicitly indicates that an error may be detected by more than one detection mechanism. This is evident from the fact that the sum of the error coverage percentages for individual mechanisms is more than 100%. As mentioned earlier, this is because a transient fault can produce more than one type of error as its effect propagates through the microprocessor's circuits. Thus, the same fault may produce many errors that can be detected by more than one mechanism. As such, different mechanisms may have different error detection latencies for the same fault. This effect is evident from Figure 22 and Figure 23 which show multiple detection of an error by more than one mechanism. The latencies are measured in clock cycles and instruction cycles in Figure 22 and Figure 23, respectively. The highest point on the bar for each mechanism indicates the total error coverage and different portions show the time of detection relative to others. The first area shows the number of times that a particular mechanism was among the first to indicate the error, the second area indicates the number of times it was among the second and so forth. During the experiment, the responses were recorded for the first 50 clock cycles following the detection of an error by any mechanism. The data was tabulated for the first three detections only.

An analysis of undetected errors was performed to better understand why they were not detected. For every type of fault, Table 8 shows the total number of faults simu-

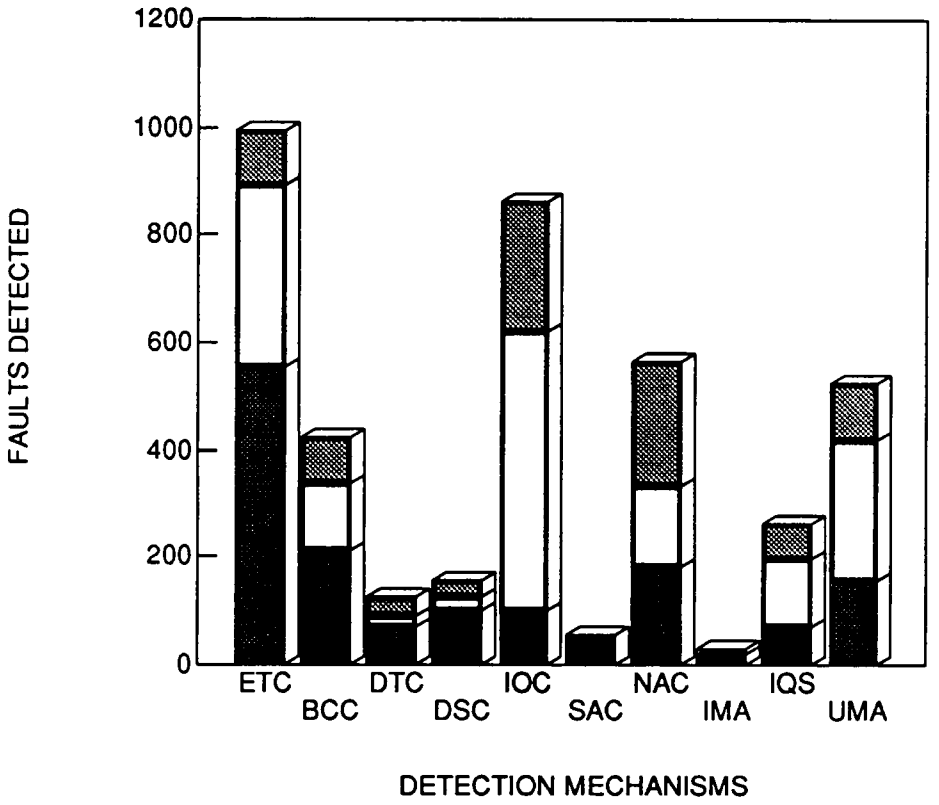

Figure 22. Multiple Detections of a Fault (in clock cycles)

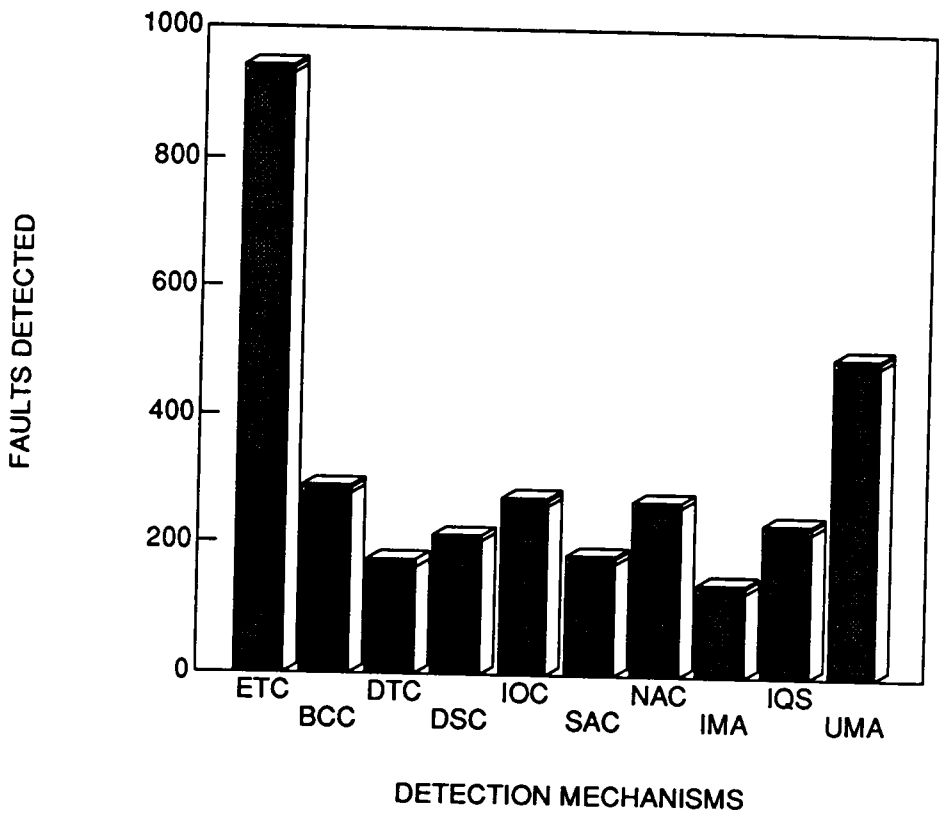

Figure 23. Multiple Detections of a Fault (in instruction cycles)

lated, number of faults that resulted in errors, and the number and percentage of undetected faults. This data indicates that a significant number of type 2 and 3 faults are not detected. These are I_j/I_k and $I_j/I_j + I_k$ faults. A detailed study of the simulation results indicated that in many cases the fault produces a transition of the microprocessor from one valid state to another valid state. These two states are indistinguishable from each other. It corresponds to some operation internal to the microprocessor that does not produce any detectable change on the external lines. Conceptually, this corresponds to the case when the two instructions, I_j and I_k, are in the same block of the product $\pi_1\pi_2\pi_3\pi_4$. For example, the instructions for moving data between memory and internal registers of the 8086 are encoded so that the internal registers have addresses that differ by one bit only. Therefore, a change in an opcode can result in the selection of an incorrect data register being used for an operation. For many of these opcodes, the corresponding execution profiles are identical. Thus, a fault that produces a decoding error in selection of an internal data register is not detected by the WDP.

A somewhat similar situation occurs for the $I_j/I_j + I_k$ fault. If the instructions I_j and I_k have identical execution profiles, they cannot be distinguished from each other. Since the WDP can neither determine the contents of internal data register nor can it monitor the output of the register selection logic, it cannot detect a register decoding fault.

Another case of a undetected error is when a conditional jump instruction is executed. If the condition flag, that indicates if jump should be taken, is altered due to a fault, an incorrect jump may be taken. The WDP cannot detect this since the only information available to it is the jump address and the next sequential address. The

Table 8. Undetected Errors

Fault Type	Fault Name	Faults Simulated	Errors Produced	Undetected Errors	Undetected Errors %
0	l_j/μ Fault	303	84	2	0.66%
1	l_j/ϕ Fault	249	249	0	0.00%
2	l_j/l_k Fault	261	92	33	2.64%
3	$l_j/l_j + l_k$ Fault	311	287	27	8.68%
4	l_j/l'_j Fault	315	315	1	0.32%
5	l_j, l_k/l_j, l_m Fault	256	207	0	0.00%
6	Stack Fault	311	50	0	0.00%

WDP verifies that execution resumes at one of the two addresses. However, it cannot check if the correct branch is taken.

A special case of an I_j/I_k fault is when an opcode changes to F4 (hex). This is the opcode for the HALT instruction for the 8086 microprocessor. Upon executing this instruction, the microprocessor enters the halt state and no further action takes place. In this situation, the WDP continues to wait for an instruction termination signal so that it can detect if there were any errors. The result is an endless wait. A watchdog timer may be used for detection of this condition.

7.2 Error Detection Latency

The average error detection latency of every mechanism was shown in Figure 19 and Figure 20. For most mechanisms, the average error latency is small which is consistent with the requirements of the WDP. A fault can produce errors that may be detected by more than one mechanism. The individual mechanisms have different error detection latencies. A better idea of overall error detection efficiency of the WDP can be obtained by plotting the number of fault detections versus instruction cycles. This is shown in Figure 24. The number at the top of each bar represents the percentage of faults detected within a certain cycle. This graph points out that a majority of error detections occur in the first few instruction cycles following the incidence of a fault. Specifically, about 79.2% of all errors are detected in less than 2 instruction cycles. After the first few cycles, the error detection quickly declines to zero. However, some faults are detected after a long time as indicated in the graph.

A close examination of the simulation results reveal that these are mostly stack faults.

The value of the stack pointer is verified only when there is a stack operation performed by the main processor. In the test program, stack is used when the subroutine is entered and exited. If there is a stack fault after the subroutine has been entered successfully, it will not be detected until the subroutine is exited because that is when stack is used again. Thus, the structure of the software has a direct effect on the error detection latency of a stack fault.

The error detection latency of each mechanism for every type of fault was shown in Chapter 6. The instruction parameter monitoring mechanisms (ETC, BCC, DTC, and DSC) detect errors following the completion of the current instruction. This implies that the average error detection latency should be less than 2 instruction cycles. However, the experimental evidence shows that the average latency values for ETC and BCC are slightly higher than 2 instruction cycles. To explain this, consider an I_j/I_k fault. As mentioned earlier in section 6.3.1, this fault is injected by introducing errors in the instruction queue of the microprocessor and not its instruction register. But the execution of the faulted instruction does not start until the preceding instruction is completed. Thus, the latency time of the faulted instruction partially includes the execution time of the preceding instruction. Hence, simulation results give a higher time than that the actual latency value. A similar argument can be made for the IOC mechanism that detects I_j/I_o faults because all I_j/I_o faults are subsets of I_j/I_k faults.

All I_j/μ faults that result in an illegal memory access are detected with very short latency. This is because the IMA mechanism indicates an error as soon as it detects

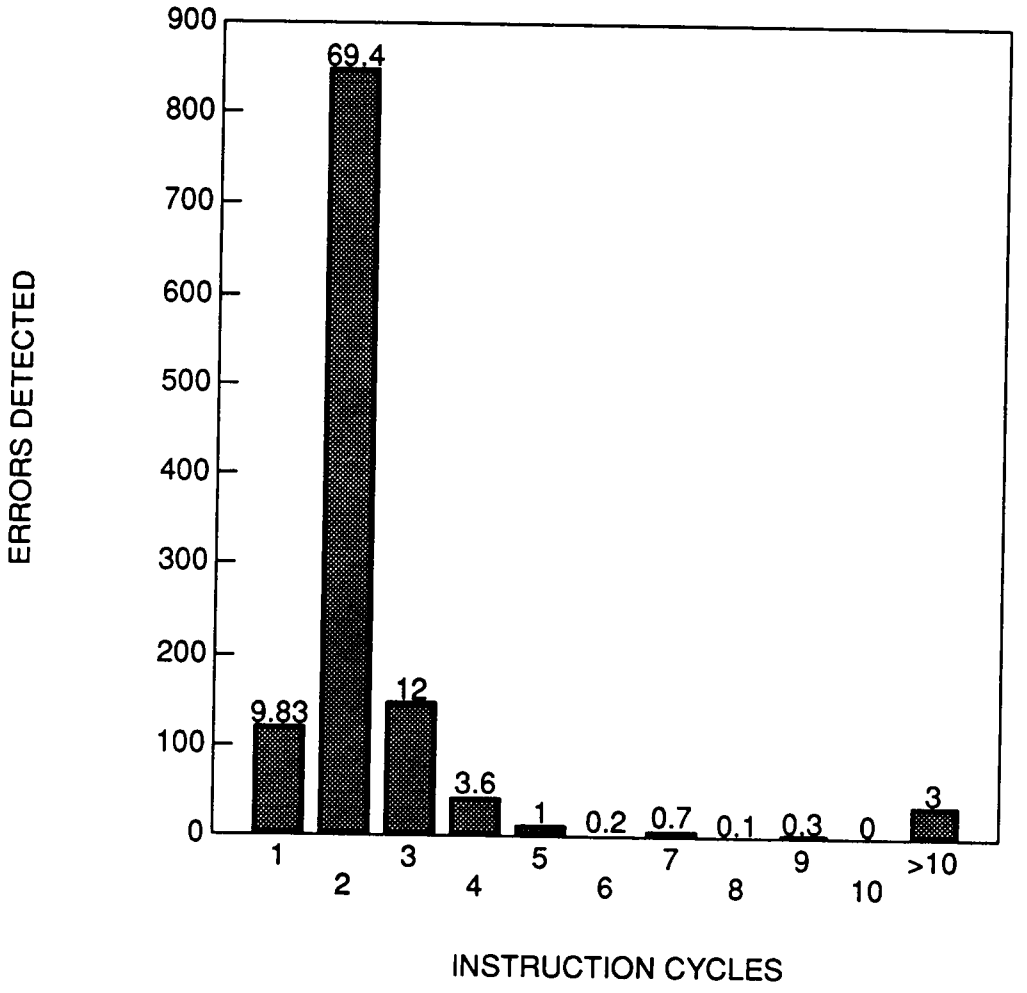

Figure 24. Performance Efficiency of WDP

it and does not wait for the end of an instruction. On the other hand, an I_j/μ fault that results in an access to an undefined memory location may have a significantly longer latency. This is because an access to such a location fetches an invalid data value. If this is requested by the microprocessor for immediate use, the UMA mechanism will check it for errors. But if this access was for normal prefetching of instructions, it will be saved in the instruction queue and will not be checked until it is fetched for execution. Thus, in some cases, the error may have a long latency time. This explains why the UMA mechanism has an average latency of 2.4 instruction cycles.

Chapter 8

Conclusions

The frequency of occurrence of transient faults is much higher than that of permanent faults in digital systems. This is especially true of space-borne systems that experience single event upsets due to the radiation environment in which they operate. These upsets occur inside the processor and must be detected concurrently to allow for real-time operation.

A new approach, called continuous processor monitoring scheme, aimed at concurrent detection of transient faults has been developed. Its effectiveness has been demonstrated through computer simulations. A transient fault model is proposed that defines the types of faults that may occur in an SEU environment. These include both control flow as well as execution errors. For every opcode in the instruction set of a processor, a set of observable parameters are defined that constitute its execution profile. A watchdog processor concurrently monitors the execution profile of every instruction and compares it with a previously determined execution profile to detect any errors.

8.1 Comparison of CPM With Other Schemes

Most monitoring schemes proposed in the literature detect control flow errors during the execution of a program by embedding signatures in the application program. This affects the performance of the system because of following reasons:

1. The application program must be recompiled before it is executed on a system that has the error detection hardware. The signatures are embedded in the program during this compilation.
2. Additional memory is needed for storing the embedded signatures.
3. The performance of the main processor is degraded when signatures are fetched from the memory.

Signatures of branch free intervals are computed during compilation and inserted at appropriate places in the program. This means that software modifications are needed before an application program can be executed on a system with the error detection hardware. The proposed CPM scheme does not have this limitation because it does not require any changes in the software. In fact, it is completely transparent to the rest of the system. Also, there is no memory overhead for the CPM scheme because it does not require any signatures to be inserted in the program.

In a signature monitoring system, the main processor executes a no operation (NOP) instruction when a signature is fetched from the memory. This results in the loss of performance as these clock cycles are wasted. This may not be a desirable situation, especially in real-time systems. The CPM scheme does not suffer from this performance loss as it does not need any embedded signatures.

A salient feature of signature monitoring schemes is that they verify the signatures at the completion of a branch free interval. As discussed in Chapter 2, certain types of errors can result in setting up of potentially endless loops that bypass the signature. These errors will not be detected by a signature monitoring scheme as the section of code containing the signature is never executed. The CPM scheme, on the other hand, verifies the instruction addresses at the beginning of the execution of every instruction. Thus, there is a better chance that a CPM scheme will detect these types of errors.

Most monitoring schemes using signature analyzers are aimed at detecting control flow errors. But in an SEU environment, a processor can also experience execution errors that must be detected. The CPM scheme is capable of detecting both control flow and execution errors. In the CPM scheme, mechanisms aimed at verification of execution profile of an instruction check for execution errors.

The proposed CPM scheme has been shown to be effective; both analytically and experimentally. An overall transient fault coverage of 96.8% with an average error detection latency of about 1.47 instruction cycles was achieved. This small error detection latency is very important for initiating an efficient recovery procedure. In most cases, a simple instruction retry may be an effective recovery procedure.

The CPM scheme is superior if high coverage, low hardware overhead, low performance overhead, and low error latency dominate the system requirements. Its performance is better that most schemes that use signature analyzers because those schemes detect only control flow errors and have significant error latency. On the other hand, a hardware redundancy scheme has better performance but suffers from a high hardware overhead penalty.

The WDP for the CPM scheme is transparent to the rest of the system. This means that it can be incorporated into existing systems without much changes in the hardware. As mentioned earlier, the CPM scheme has no software component. This suggests that existing application software can be run on a system using CPM scheme with out any changes.

In the experiments reported here, the WDP circuitry was not subjected to transient faults. For a complete analysis, the situation when a transient fault occurs in the WDP must also be addressed. Such a fault may make WDP indicate an error when there is none, i.e., false alarm. As the WDP is much smaller in area than the main processor, it is less susceptible to an SEU. When it does experience an upset, and consequently indicates a false alarm, a recovery procedure will be invoked. Thus the only penalty is the loss of performance of the system. However, this loss is expected to be minimal because of the frequency of single event upsets; one per hour in the worst case. Although a fully redundant or self checking WDP can be designed to eliminate the possibility of false alarms, it is not very cost efficient as the penalty of a false alarm is an occasional invocation of recovery procedures. Given the frequency of SEUs, it may be a tolerable penalty in most systems.

8.2 Future Directions

A significant amount of hardware in the WDP is required to monitor the bus traffic and extract the appropriate information for error detection. This overhead can be reduced if the CPM monitor is implemented on the same chip as the microprocessor. In addition, the on chip detector will make internal signals of the microprocessor visible

to the WDP. This added observability means that those faults that go undetected because certain internal control signals are not observable can be detected. As mentioned in Chapter 6, these faults include errors in register decoding logic, control flow errors due to change in the state of a condition flag, etc. For instance, consider an I_j/I_k fault that changes the opcode of an instruction to that of another instruction. As mentioned previously, the WDP cannot distinguish between the two instructions if they have identical execution profiles. This type of fault results in the selection of an incorrect source or destination register for an operation. However, with access to the internal signals, e.g., in this case the register selection logic, the WDP can note the address generated by the register selection logic. The WDP can also determine the correct address by partially decoding the instruction. By comparing the two register addresses, the WDP can check if correct register was selected.

Another aspect to be considered is the time penalty for executing a recovery scheme. To illustrate this, consider an ideal system that has a 100% fault coverage and a recovery procedure that is guaranteed to recover from every type of error condition. The time penalty for the recovery is a function of how often the recovery procedure is invoked and the execution time of the recovery procedure. Even with an ideal system, the processing of vital computations may get bogged down if the fault frequency is very high. The use of rad hard components in building blocks of the system can significantly reduce the frequency of occurrence of transient faults. Although these rad hard components have a high tolerance to SEUs, they are not totally immune to the types of faults discussed in this dissertation. Therefore, use of rad hard components cannot completely eliminate the need for error detection schemes.

Another factor to consider is the implementation of the recovery procedure. For minimizing the performance loss due to an error, it is necessary that the execution time of the recovery process be small. The scope of a recovery procedure is a function of the extent of damage due to an error. A fault tolerant system may have a number of recovery procedure that may range from a simple instruction retry to a complete restoration of the state of the processor. The selection of an appropriate recovery procedure depends upon the information about the extent of damage due to an error. Therefore, it important that the error detection mechanism not only notify that an error has occurred but also provide some information about the extent of damage. This information can then be used to invoke the most efficient recovery procedure. Further work in development of error detection schemes should also try to incorporate a damage assessment scheme.

Bibliography

1. D. A. Rennels, ***Fault-Tolerant Computing - Concepts and Examples,*** IEEE Transactions on Computers, Vol. C-33, No. 12, December 1984, pp. 1116-1129.

2. P. C. Carney, ***Selecting On-Board Satellite Computer Systems,*** Computer, Vol. 16, No. 4, April 1983, pp. 35-41.

3. D. A. Ballard, ***Designing Fail-safe Microprocessor Systems,*** Electronics, January 4, 1979, pp. 139-143.

4. J. D. Blair and R. D. McCorkle, ***Dual Digital Flight Control Redundancy Management System Development Program,*** Guidance and Control Conference, AIAA, 1979, pp. 40-46.

5. W. E. Arens and D. A. Rennels, ***A Fault-Tolerant Computer for Autonomous Spacecraft,*** International Symposium on Fault-Tolerant Computing, 1983, pp. 467-470.

6. R. D. Rasmussen and E. C. Litty, ***A Voyager Attitude Control Prospective on Fault Tolerant Systems,*** Guidance and Control Conference, AIAA, 1981, pp. 241-248.

7. J. G. Yee and Y. H. Su, ***A Scheme for Tolerating Faulty Data in Real-Time Systems,*** 2nd International Computer Software and Applications Conference, November 1978, pp. 663-667.

8. T. Anderson and P. A. Lee, ***Fault Tolerance: Principles and Practice,*** Prentice Hall International, Englewood Cliffs, NJ, 1981.

9. D. B. Sarrazin and M. Malek, ***Fault-Tolerant Semiconductor Memories,*** Computer, Vol. 17, No. 8, August 1984, pp. 49-56.

10. T. Anderson (Ed), ***Resilient Computing Systems,*** Vol. I, John Wiley & Sons, NY, 1985.

11. P. K. Lala, ***Fault Tolerant and Fault Testable Hardware Design,*** Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1985.

12. E. J. McCluskey, ***Hardware Fault Tolerance,*** Proceedings of the IEEE Computer Society International Conference (COMPCON), Feb 25-28, 1983, pp. 260-263.

13. D. Binder, E. G. Smith and A. B. Holman, ***Satellite Anomalies from Galactic Cosmic Rays,*** IEEE Transaction s on Nuclear Science, Vol. NS-22, N0. 6, December 1975, pp. 2675-2680.

14. M. Geilhufe, ***Soft Errors in Semiconductor Memories,*** IEEE Computer Society International Conference (COMCON 79), 1979, pp. 210-216.

15. J. C. Pickel and J. T. Blandford, ***Cosmic Ray Induced Errors in MOS Memory Cells,*** IEEE Transactions on Nuclear Science, Vol. NS-25, No. 6, December 1978, pp. 1166-1170.

16. J. C. Pickel and J. T. Blandford, ***Cosmic Ray Induced Errors in MOS Devices,*** IEEE Transactions on Nuclear Science, Vol. NS-27, No. 2, April 1980, pp. 1006-1015.

17. J. C. Pickel and J. T. Blandford, ***CMOS RAM Cosmic Ray Induced Error Rate Analysis,*** IEEE Transactions on Nuclear Science, Vol. NS-28, No. 6, December 1981, pp. 3962-3967.

18. R. D. Rasmussen, ***Computing in the Presence of Soft Bit Errors,*** American Control Conference, 1984, pp. 1125-1130.

19. J. F. Ziegler and W. A. Lanford, ***Effects of Cosmic Rays on Computer Memories,*** Science, Vol. 206, 16 November 1979, pp. 776-788.

20. T. C. May and M. H. Wood, ***Alpha-Particle-Induced Soft Errors in Dynamic Memories,*** IEEE Transactions on Electron Devices, Vol. ED-26, No. 1, January 1979, pp. 2-9.

21. J. P. Woods, D. K. Nichols and W. E. Price, ***Investigation for Single-Event Upsets in MSI Devices,*** IEEE Transactions on Nuclear Science, Vol. NS-28, No. 6, December 1981, pp. 4022-4025.

22. D. K. Nichols, W. E. Price, C. J. Malone and L. S. Smith, ***A Summary of JPL Single Event Upset Test Data from May 1982 through January 1984,*** IEEE Transactions on Nuclear Science, Vol. NS-31, No. 6, December 1984, pp. 1186-1192.

23. D. K. Nichols, W. E. Price amd C. J. Malone, ***Single Event Upset (SEU) of Semiconductor Devices - A Summary of JPL Data,*** IEEE Transactions on Nuclear Science, Vol. NS-30, No. 6, December 1983, pp. 4520-4525.

24. W. A. Kolasinski, J. B. Blake, J. K. Anthony, W. E. Price and E. C. Smith, ***Simulation of Cosmic Ray Induced Soft Errors and Latchup in Integrated Circuit Computer Memories,*** IEEE Transactions on Nuclear Science, Vol. NS-26, No. 6, December 1979, pp. 5087-5091.

25. R. Koga and W. A. Kolasinski, ***Heavy Ion-Induced Single Event Upsets of Microcircuits: A Summary of Aerospace Corporation Test Data,*** IEEE Transactions on Nuclear Science, Vol. NS-31, No. 6, December 1984, pp. 1190-1195.

26. K. E. Martin, M. K. Gauthier, J. R. Cross, A. R. V. Dantas and W. E. Price, ***Total-Dose Radiation Effects Data for Semiconductor Devices: 1985 Supplement,*** JPL

Publication 85-43, Volume II, Parts A and B, Jet Propulsion Lab, California Institute of Technology, Pasadena, CA, May 15, 1986.

27. S. E. Diehl, J. E. Vinson, B. D. Shafer and T. M. Mnich, ***Considerations for Single Event Immune VLSI Logic,*** IEEE Transactions on Nuclear Science, Vol. NS-30, No. 6, December 1983, pp.4501-4507.

28. S. E. Diehl-Nagel, J. E. Vinson and E. L. Peterson, ***Single Event Upset Rate Predictions for Complex Logic Systems,*** IEEE Transactions on Nuclear Science, Vol. NS-31, No. 6, December 1984, pp.1132-1138.

29. A. L. Friedman, B. Lawton, K. R. Hotelling, J. C. Pickel, V. H. Strahan and K. Loree, ***Single Event Upset in Combinational and Sequential Current Mode Logic,*** IEEE Transactions on Nuclear Science, Vol. NS-32, No. 6, December 1985, pp.4216-4218.

30. D. P. Sieworek and R. S. Swarz, ***The Theory and Practice of Reliable System Design,*** Digital Press, Digital Equipment Corp., 1982.

31. D. P. Sieworek, ***Architecture of Fault-Tolerant Computers,*** Computer, Vol. 7, No. 8, August 1984, pp. 9-18.

32. D. W. Haislett, ***A Methodology for Self Testing Microprocessors,*** Masters Thesis, Dept. of Electrical Engineering, Virginia Tech, Blacksburg, VA, 1982.

33. L. Chen and A. Avizienis, ***N-Version Programming: A Fault-Tolerant Approach to Reliability of Software Operation,*** International Symposium on Fault-Tolerant Computing, 1978, pp. 3-9.

34. J. C. Knight, N. G. Levenson and L. D. St. Jean, ***A Large Scale Experiment in N-version Programming,*** International Symposium on Fault-Tolerant Computing, 1985, pp. 135-139.

35. A. Avizienis, ***Fault Tolerance by Means of External Monitoring of Computer Systems,*** National Computer Conference, American Federation of Information Processing Societies (AFIPS), Vol. 50, 1981, pp. 27-40.

36. T. S. Liu, ***The Role of a Maintenance Processor for a General-Purpose Computer System,*** IEEE Transactions on Computers, Vol. C-33, No. 6, June 1984, pp. 507-517.

37. ***Rad-Hard/Hi-Rel Data Book,*** Harris Corporation, Melbourne, FL, 1987.

38. Y. Savaria, V. K. Agarwal, N. Rumin and J. F. Hayes, ***A design for Machines with Built-in Tolerance to Soft Errors,*** IEEE International Test Conference, 1984, pp. 649-659.

39. X. Castillo, S. R. McConnel and D. P. Siewiorek, ***Derivation and Calibration of a Transient Error Reliability Model,*** IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982, pp. 658-671.

40. J. Sosnowski, ***Transient Fault Tolerance in Data Acquisition System,*** Microprocessing and Microprogramming, Vol. 16, 1985, pp. 255-260.

41. J. Sosnowski, ***Transient Fault Effects in Microprocessor Controllers,*** Reliability Technology - Theory & Applications (REL-CON, EUROPE 86), 1986, pp. 239-248.

42. R. G. Halse and C. Preece, ***Erroneous Execution and Recovery in Microprocessor Systems,*** Software & Microsystems, Vol. 4, No. 3, June 1985, pp. 63-70.

43. M. O. Ahmad and D. V. Poornaiah, ***Design of a Logic Circuit for Microprocessor Recovery From a Power Failure and a Transient Fault,*** IEEE Transactions on Circuits and Systems, Vol. CAS-34, No. 4, April 1987, pp. 433-436.

44. R. E. Glaser and G. M. Masson, ***Transient Upsets in Microprocessor Controllers,*** International Symposium on Fault-Tolerant Computing, 1981, pp. 165-167.

45. R. E. Glaser and G. M. Masson, ***The Containment Set Approach to Crash-Proof Microprocessor Controller Design,*** International Symposium on Fault-Tolerant Computing, 1982, pp. 215-222.

46. R. E. Glaser and G. M. Masson, ***The Containment Set Approach to Upsets in Digital Systems,*** IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982, pp. 689-692.

47. M. E. Schmid, R. L. Trapp, A. E. Davidoff and G. M. Masson, ***Upset Exposure by Means of Abstraction Verification,*** International Symposium on Fault-Tolerant Computing, 1982, pp. 237-244.

48. K. W. Li, ***Detection of Transient Faults in Microprocessors by Means of External Hardware,*** Masters Thesis, Dept. of Electrical Engineering, Virginia Tech, Blacksburg, VA, 1984.

49. K. W. Li, J. R. Armstrong and J. G. Tront, ***An HDL Simulation of the Effects of Single Event Upsets on Microprocessor Program Flow,*** IEEE Transactions on Nuclear Science, Vol. NS-31, No. 6, December 1984, pp.1139-1144.

50. S. F. Daniels, ***A Concurrent Test Technique for Standard Microprocessors,*** 26th IEEE Computer Society International Conference (COMCON), San Francisco, CA, February 28-March 3, 1983, pp.389-394.

51. J. V. Oak, J. G. Tront and J. R. Armstrong, ***A Block Checking Approach to Self Testing Software,*** International Journal of Mini and Microcomputers, Vol. 7, No. 3, 1985, pp. 83-88.

52. J. V. Oak, ***Block Checking Approach to Self-Testing Software,*** Masters Thesis, Dept. of Electrical Engineering, Virginia Tech, Blacksburg, VA, 1984.

53. E. K. Heironimus, ***Automated Incorporation of Upset Detection Mechanisms in Distributed Ada Systems,*** Masters Thesis, Dept. of Electrical Engineering, Virginia Tech, Blacksburg, VA, 1988.

54. H. Hecht, ***Fault-Tolerant Software for Real-Time Applications,*** ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 391-407.

55. B. Randell, ***System Structure for Software Fault Tolerance,*** IEEE Transaction on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 220-232.

56. N. G. Leveson and T. J. Shimeall, ***Safety Assertions for Process-Control Systems,*** International Symposium on Fault-Tolerant Computing, 1983, pp. 236-240.

57. M. Namjoo and E. J. McCluskey, ***Watchdog Processors and Capability Checking,*** International Symposium on Fault-Tolerant Computing, 1982, pp. 245-248.

58. D. K. Bhavasar and R. W. Heckelman, ***Self Testing by Polynomial Division,*** IEEE International Test Conference, 1981, pp. 208-216.

59. K. S. Bhaskar, ***Signature Analysis: Yet Another Perspective,*** IEEE International Test Conference, 1982, pp. 132-134.

60. J. P. Robinson and N. R. Saxena, ***A Unified View of Test Compression Methods,*** IEEE Transactions on Computers, Vol. C-36, No. 1, January 1987, pp. 94-99.

61. M. Namjoo, ***Techniques for Concurrent Testing of VLSI Processor Operation,*** IEEE International Test Conference, 1982, pp. 461-468.

62. T. Sridhar and S. M. Thatte, ***Concurrent Checking of Program Flow in VLSI Processors,*** IEEE International Test Conference, 1982, pp. 191-199.

63. M. A. Schuette and J. P. Shen, ***Processor Control Flow Monitoring Using Signatured Instruction Streams,*** IEEE Transactions on Computers, Vol. C-36, No.3, March 1987, pp. 264-276.

64. K. D. Wilken and J. P. Shen, ***Embedded Signature Monitoring: Analysis and Technique,*** IEEE International Test Conference, 1987, pp. 324-333.

65. A. Mahmood and E. J. McCluskey, ***Concurrent Error Detection Using Watchdog Processors - A Survey,*** IEEE Transactions on Computers, Vol. C-37, No. 2, February 1988, pp. 160-174.

66. A. Mahmood, D. J. Lu and E. J. McCluskey, ***Concurrent Fault Detection Using a Watchdog Processor and Assertions,*** IEEE International Test Conference, 1983, pp. 622-628.

67. A. Mahmood and E. J. McCluskey, ***Watchdog Processors: Error Coverage and Overhead,*** International Symposium on Fault-Tolerant Computing, 1985, pp. 214-219.

68. A. Mahmood, A. Ersoz and E. J. McCluskey, ***Concurrent System Level Error Detection Using a Watchdog Processor,*** IEEE International Test Conference, 1985, pp. 145- 152.

69. M. Namjoo, ***CERBERUS-16: An Architecture for a General Purpose Watchdog Processor,*** CRC Technical Report No. 82-19, Center for Reliable Computing, Stanford University, December 1982.

70. J. P. Shen and M. A. Schuette, ***On-line Self-Monitoring Using Signatured Instruction Streams,*** IEEE International Test Conference, 1983, pp. 275-282.

71. J. B. Eifert and J. P. Shen, ***Processor Monitoring using Asynchronous Signatured Instruction Streams,*** IEEE International Test Conference, 1983, pp. 394-399.

72. K. D. Wilken and J. P. Shen, ***Continuous Signature Monitoring: Efficient Concurrent Detection of Processor Control Errors,*** International Test Conference, 1988, pp. 914-925.

73. J. P. Hayes and E. J. McCluskey, ***Testability Considerations in Microprocessor-Based Design,*** Computer, March 1980, pp. 17-26.

74. C. H. Tung and J. P. Robinson, ***On Concurrently Testable Microprogrammed Control Units,*** IEEE International Test Conference, 1986, pp. 895-900.

75. M. M. Yen, W. K. Fuchs and J. A. Abraham, ***Designing for Concurrent Error Detection in VLSI: Application to a Microprogram Control Unit,*** IEEE Journal of Solid State Circuits, Vol. SC-22, No.4, August 1987, pp. 595-605.

76. V. S. Iyangar and L. L. Kinney, ***Concurrent Fault Detection in Microprogrammed Control Units,*** IEEE Transactions on Computers, Vol. C-34, No. 9, September 1985, pp. 810-821.

77. A. M. Paschalis, C. Halatsis and G. Philokyprou, ***Concurrently Totally Self-Checking Microprogrammed Control Units with Duplication of Microprogram Sequencer,*** Microprocessing and Microprogramming, Vol. 20, 1987, pp. 271-281.

78. J. Duran and T. Mangir, ***Application of Signature Analysis to the Concurrent Test of Microprogrammed Control Units,*** Microprocessing and Microprogramming, Vol. 20, 1987, pp. 309-322.

79. A. Anatola, R. Negrini, M. Sami and N. Scarabottolo, ***Transient Fault Management in Microprogrammed Units: A Software Recovery Approach,*** EUROMICRO, 1985, pp. 453-461.

80. A. Anatola, R. Negrini, M. Sami and N. Scarabottolo, ***Hardening VLSI Microprogrammed Units Against Transient Failures,*** Proc. VLSI and Computers, First International Conference on Computer Technology, Systems and Applications (COMPEURO), Hamberg, May 1987, pp. 8-10.

81. A. Anatola, I. Erenyi and N. Scarabottolo, ***Transient Fault Management in Systems Based on the AMD 2900 Microprocessors,*** Microprocessing and Microprogramming, Vol. 20, 1987, pp. 205-217.

82. M. Marshal, ***Logic in Testingland - A Tale of Measurement Techniques,*** EDN Magazine, January 1976.

83. M. Marshal, ***Through the Memory Cells - Further Explorations of Integrated Circuits in Testingland,*** EDN Magazine, February 1976.

84. M. Marshal, ***Microprocessors in Testingland - the Jury Receives its Instructions,*** EDN Magazine, March 1976.

85. K. K. Saluja, L. Shen and S. Y. H. Su, ***A Simplified Algorithm for Testing Microprocessors,*** IEEE International Test Conference, 1983, pp. 668-675.

86. S. M. Thatte and J. A. Abraham, ***Test Generation for Microprocessors,*** IEEE Transactions on Computers, Vol. C-29, No. 6, June 1980, pp. 429-441.

87. W. Denhardt and V. M. Sorensen, ***Unspecified 8085 Opcodes Enhance Programming,*** Electronics, June 18, 1979, pp. 145-147.

88. J. Y. O. Fong, ***Microprocessor Modeling for Logic Simulation,*** IEEE International Test Conference, 1981, pp. 458-460.

89. P. S. Bottoroff, ***Functional Testing Folklore and Fact,*** IEEE International Test Conference, 1981, pp. 463-464.

90. J. A. Abraham, ***Functional Level Test Generation for Complex Digital Systems,*** IEEE International Test Conference, 1981, pp. 461-462.

91. HILO-3 Users Manual, Doc. #2522-0100, GenRad Inc., June 1985.

92. K. Mark and P. Measel, ***Total Dose Test Results for the 8086,*** IEEE Transactions on Nuclear Science, Vol. NS-29, No. 6, December 1982, pp. 70-80.

93. ***iAPX 86,88 User's Manual,*** Intel Corporation, Santa Clara, CA, July 1981.

94. J. B. Clany and R. A. Sacane, ***Self-Testing Computers,*** Computer, October, 1979, pp.49-59.

95. S. Osaki and T. Nishio, ***Reliability Evaluation of Some Fault-Tolerant Computer Architectures,*** Lecture Notes in Computer Science, Springr-Verlag, Berlin Heidelberg, West Germany, 1980.

96. D. S. Ha, ***On the Testable Design and Built-In Self-Test of PLAs,*** Ph. D. Dissertation, Dept. of Electrical and Computer Engineering, University of Iowa, Iowa, May 1987.

Appendix A
Simulation Model of the WDP

```
ic tfd ( a[19], a[18], a[17], a[16], ad[15], ad[14], ad[13], ad[12],
        ad[11], ad[10], ad[9], ad[8], ad[7], ad[6], ad[5], ad[4], ad[3],
        ad[2], ad[1], ad[0], clk, resetin, qs0, qs1, ns0, ns1, ns2,
        nbhes7, ready, msg, seuin )

**************************************************************************
**                                                                      **
**      name          : Watchdog Processor for Detection of Transient   **
**                    : Faults in (8086) Microprocessors                **
**      nick name     : TFD - transient fault detector                  **
**      designer      : Mohammad Ziaullah Khan                          **
**      programmer    : Mohammad Ziaullah Khan                          **
**      date          : 1989                                            **
**      reference     : Ph.D. Dissertation (MZK)                        **
**                    : Department of Electrical Engineering            **
**                    : Virginia Polytechnic Institute & State University **
**                      Blacksburg, Virginia, 24061                     **
**                                                                      **
**************************************************************************

clock0(1,1k,999m) power(powerup);

input   qs0 qs1 clk resetin ready ns0 ns1 ns2 msg seuin;
unid    powerup;
tri     a[19:16] ad[15:0] nbhes7;

**input/output registers

register(1,1)
        instaddr[19:0]                  ** instruction address register
        data[15:0];                     ** data input/output register

**instruction queue and queue pointer

register(1,1)
        iq1[7:0] iq2[7:0] iq3[7:0]      ** instruction queue
        iq4[7:0] iq5[7:0] iq6[7:0]
        qp[2:0]                         ** instruction queue pointer
        aq1[19:0] aq2[19:0] aq3[19:0]   ** address queue
        aq4[19:0] aq5[19:0] aq6[19:0]

**miscellaneous registers

        ir[7:0]                         ** instruction register
        ire[7:0]                        ** instruction register extension
        irex[7:0]                       ** temporary instruction register ext
        tar[19:0]                       ** address register
```

```
        obytecnt[3:0]                   ** observed byte count
        datasize[1:0]                   ** observed data size
        datanum[1:0]                    ** observed data count
        ocyclecnt[7:0]                  ** observed execution cycle count

        ebytecnt[3:0]                   ** expected byte count
        xfersize[1:0]                   ** expected data size
        xfernum[1:0]                    ** expected data count
        ecyclecnt[7:0]                  ** expected execution cycle count
        eatime[7:0]                     ** ea address calculation overhead

        stackadd[15:0]                  ** stack address register
        tempstackadd[15:0]              ** temporary stack address

        nextadd[19:0]                   ** next instruction address
        jumpadd[19:0]                   ** jump/branch address

        initflag                        ** cycle start flag
        stackflag                       ** stack operation flag
        cycleflag                       ** cycle count flag
        countflag                       **
        jumpflag[1:0]                   ** branch check flag
        readonlyflag                    ** memory read only flag
        qnopflag                        ** queue idle flag

**      temporary flags for internal operations

        s1[7:0]
        s2[7:0]
        s3[7:0]
        tempcount[7:0]
        instcnt[7:0];

*************************************************************************
**                                                                     **
**                          reset sequence                             **
**                                                                     **
*************************************************************************

when powerup(0 to 1) do
        if resetin = 1 then
                event resetseq
        endif;

when resetin(0:x to 1) then clk(0 to 1) do
        event resetseq;

when resetseq do
        qp[2:0] = bin 001
        instaddr[19:0] = 0
        obytecnt = 0
        ebytecnt = 0
        initflag = 1
        jumpflag = bin 10
        readonlyflag = 0
        stackflag = 0
        s1 = 0
        s2 = 0
        s3 = 0
        instcnt = 0
        cycleflag = 0
        ecyclecnt = 0
        eatime = 0
        stackadd = 0
        datasize = 0
        datanum = 0
```

```
        xfersize = 0
        xfernum = 0;

**      The TFD reads the stack address upon first use of stack

next 3*clk(0 to 1) do
        if resetin = 1 then
                event waitendrst
        else
                event illegalrst
        endif;

when waitendrst then resetin(1 to 0) then clk(0 to 1) do
        if powerup = 1 then
                event illegalrst
        else
                event waitstart
        endif;

when illegalrst do
        display "illegal reset action simulation aborted"
        finish;

when waitstart then 8*clk(0 to 1) reset resetseq do
        event prepstart;

*************************************************************************
**                                                                     **
**                      queue control                                  **
**                                                                     **
*************************************************************************

when prepstart or monqueue then qs0(? to ?) or qs1(? to ?)
                then clk(0 to 1) reset resetseq do
        event qpcase;

when qpcase wait 10 reset resetseq do
        case (qs1,qs0),
                00- event (monqueue qnop)
                        if msg = 1 then
                                display ("No quueue operation at = ",time)
                        endif,

                01- ir = iq1 tar = aq1
                        event (idecode monqueue cyclecnt)
                        iq1 = iq2
                        iq2 = iq3
                        iq3 = iq4
                        iq4 = iq5
                        iq5 = iq6
                        qp = qp-1
                        aq1 = aq2
                        aq2 = aq3
                        aq3 = aq4
                        aq4 = aq5
                        aq5 = aq6
                        if msg = 1 then
                                display ("First byte read from Q at = ",time)
                        endif,

                10- qp[2:0] = bin 001
                        event (monqueue reinit)
                        if msg = 1 then
                                display ("Q reinitialized at = ",time)
                        endif,

                11-     irex =  iq1 event (subyte chknext)
```

```
                        datasize = 0
                        qp = qp-1
                        iq1 = iq2
                        iq2 = iq3
                        iq3 = iq4
                        iq4 = iq5
                        iq5 = iq6
                        aq1 = aq2
                        aq2 = aq3
                        aq3 = aq4
                        aq4 = aq5
                        aq5 = aq6
                        if msg = 1 then
                                display ("Subsequent byte at = ",time)
                        endif,
        endcase;

when qpcase wait 5 reset resetseq do
        case (qs1,qs0),
                00-     ,
                01-     if initflag = 0 then
                                event cycleend
                        endif,
                10-     ,
                11-     ,
        endcase;

when chknext then clk(0 to 1) reset resetseq do
        event qpcase;

when idecode then subyte wait 1 reset resetseq do
        ire = irex;

when idecode wait 20  reset resetseq do
        s1 = 0
        s2 = 0
        s3 = 0
        countflag = 0
        qnopflag = 0
        tempcount = 0;

*************************************************************************
**                                                                     **
**                monitor processor actions                            **
**                                                                     **
*************************************************************************

when (ns0 or ns1 or ns2)+10 reset resetseq do
        case (ns2,ns1,ns0),
                000-    event intack
                        if msg = 1 then
                                display ("BUS:interrupt acknowledged at = ",time)
                        endif,
                001-    event (readio checkseg)
                        if msg = 1 then
                                display ("BUS:reading i/o at = ",time)
                        endif,
                010-    event (writeio checkseg)
                        if msg = 1 then
                                display ("BUS:writing i/o at = ", time)
                        endif,
                011-    event prochalt
                        if msg = 1 then
                                display ("BUS:processor halted at = ",time)
                        endif,
                100-    event (ifetch checkseg1)
                        if msg = 1 then
```

```
                        display ("BUS:Instruction fetch at = ",time)
                endif,
        101-    event (dsize readmem checkseg)
                if msg = 1 then
                        display ("BUS:reading memory at  = ",time)
                endif,
        110-    event (dsize writemem checkseg)
                if msg = 1 then
                        display ("BUS:writing memory at  = ",time)
                endif,
        111-    if msg = 1 then
                        display ("BUS:Passive at = ", time)
                endif
                event nobusop,
    endcase;

*************************************************************************
**                                                                     **
**                  monitor instruction fetches                        **
**                                                                     **
*************************************************************************

when ifetch then clk(0 to 1) reset resetseq do
    instaddr = (a[19:16], ad[15:0]);

next clk(0 to 1)  wait 5 reset resetseq do
    if ns2 = 1 & ns1 = 0 & ns0 = 0 then
        event contcyc
    endif;

when contcyc then 2*clk(1 to 0) reset resetseq wait 5 do
    data = ad
    if instaddr[0] = 1 then
        event onebyte
    else
        event twobyte
    endif;

next clk(0 to 1) wait 20 reset resetseq do
    event pushq;

when twobyte then pushq reset resetseq do
    case qp,
        000-    display ("TFD queue underflow at = ",time)
                event err9,
        001-    iq1 = data[7:0]
                iq2 = data[15:8] qp = 3
                aq1 = instaddr
                aq2 = instaddr + 1,
        010-    iq2 = data[7:0]
                iq3 = data[15:8]
                qp = 4
                aq2 = instaddr
                aq3 = instaddr + 1,
        011-    iq3 = data[7:0]
                iq4 = data[15:8]
                qp = 5
                aq3 = instaddr
                aq4 = instaddr + 1,
        100-    iq4 = data[7:0]
                iq5 = data[15:8]
                qp = 6
                aq4 = instaddr
                aq5 = instaddr + 1,
        101-    iq5 = data[7:0]
                iq6 = data[15:8]
                qp = 7
```

```
                        aq5 = instaddr
                        aq6 = instaddr + 1,
                11?-    display ("TFD queue overflow at = ", time)
                        event err9,
                default- display ("TFD queue corruption at = ", time)
                        event err9,
        endcase
        if data[7:0] = x | data[15:8] = x then
                event err10
        endif;

when onebyte then pushq reset resetseq do
        case qp,
                000-    display ("TFD queue underflow at = ", time)
                        event err9,
                001-    iq1 = data[15:8]
                        qp = 2
                        aq1 = instaddr,
                010-    iq2 = data[15:8]
                        qp = 3
                        aq2 = instaddr,
                011-    iq3 = data[15:8]
                        qp = 4
                        aq3 = instaddr,
                100-    iq4 = data[15:8]
                        qp = 5
                        aq4 = instaddr,
                101-    iq5 = data[15:8]
                        qp = 6
                        aq5 = instaddr,
                11?-    display ("TFD queue overflow at = ", time)
                        event err9,
                default- display ("TFD queue corruption at = ", time)
                        event err9,
        endcase
        if data[15:8] = x then
                event err10
        endif;

*************************************************************************
**                                                                     **
**                      primary instruction decode                     **
**                                                                     **
*************************************************************************

when idecode wait 20 reset resetseq do
        case ir,
                000000??-       event (evbx evb2 regmem tmod),
                00000100-       event (evb2 t4),
                00000101-       event (evb3 t4),
                00000110-       event (evb1 stackpush ta),
                00000111-       event (evb1 stackpop t8),

                000010??-       event (evbx evb2 regmem tmod),
                00001100-       event (evb2 t4),
                00001101-       event (evb3 t4),
                00001110-       event (evb1 stackpush ta),
                00001111-       event (err5 ),

                000100??-       event (evbx evb2 regmem tmod),
                00010100-       event (evb2 t4),
                00010101-       event (evb3 t4),
                00010110-       event (evb1 stackpush ta),
                00010111-       event (evb1 stackpop t8),

                000110??-       event (evbx evb2 regmem tmod),
                00011100-       event (evb2 t4),
```

```
00011101-       event (evb3 t4),
00011110-       event (evb1 stackpush ta),
00011111-       event (evb1 stackpop t8),

001000??-       event (evbx evb2 regmem tmod),
00100100-       event (evb2 t4),
00100101-       event (evb3 t4),
00100110-       event (evb1 t2),
00100111-       event (evb1 t4),

001010??-       event (evbx evb2 regmem tmod),
00101100-       event (evb2 t4),
00101101-       event (evb3 t4),
00101110-       event (evb1 t2),
00101111-       event (evb1 t4),

001100??-       event (evbx evb2 regmem tmod),
00110100-       event (evb2 t4),
00110101-       event (evb3 t4),
00110110-       event (evb1 t2),
00110111-       event (evb1 t4),

001110??-       event (evbx evb2 regmem tmod)
                  readonlyflag = 1,
00111100-       event (evb2 t4),
00111101-       event (evb3 t4),
00111110-       event (evb1 t2),
00111111-       event (evb1 t4),

0100????-       event (evb1 t2),
01010????-      event (evb1 stackpush tb),
01011????-      event (evb1 stackpop t8),
0110????-       event (err5),
0111????-       event (evb2 evbjumpshort t10or4),

10000000-       event (evbx evb3 regmem chkregmem ev80),
10000001-       event (evbx evb4 regmem chkregmem ev80),
1000001?-       event (evbx evbchk1 evb3 regmem chkregmem ev80),
100001??-       event (evbx evb2 regmem)
                  readonlyflag = 1,
100010??-       event (evbx evb2 regmem tmov)
                  readonlyflag = 1,

10001100-       event (evbx evbchk2 evb2 regmem tmov)
                  readonlyflag = 1,
10001101-       event (evbx evb2 memr16 leamov),
10001110-       event (evbx evbchk2 evb2 regmem tmov)
                  readonlyflag = 1,
10001111-       event (evbx evbxnop evb2 regmem stackpop)
                  readonlyflag = 1,
10010????-      event (evb1 t3),

10011000-       event (evb1 t2),
10011001-       event (evb1 t5),
10011010-       event (evb5 evbjump stackpush2 t1b),
10011011-       event (evb1),
10011100-       event (evb1 stackpush ta),

10011101-       event (evb1 stackpop t8),
1001111?-       event (evb1 t4),
101000??-       event (evb3 memr ta),
101001??-       event (evb1 evbnoimp),
10101000-       event (evb2 t4),

10101001-       event (evb3 t4),
1010101?-       event (evb1 memr evbnoimp),
101011??-       event (evb1 memr evbnoimp),
```

```
                10110???-       event (evb2 t4),
                10111???-       event (evb3 t4),

                1100000?-       event (err5),
                11000010-       event (evb3 evretshort stackpop tc),
                11000011-       event (evb1 evretshort stackpop t8),
                1100010?-       event (evbx evb2 memr16),
                11000110-       event (evbx evbxnop evb3 memr),

                11000111-       event (evbx evbxnop evb4 memr),
                1100100?-       event (err5),
                11001010-       event (evb3 evretlong stackpop2 t11),
                11001011-       event (evb1 evretlong stackpop2 t12),
                11001100-       event (evb1 evbjump stackpush3 t34),

                11001101-       event (evb2 evbjump stackpush3 t35),
                11001110-       event (evb1 evbjump stackpush3 t35or4),
                11001111-       event (evb1 eviret stackpop3 t18),
                1101000?-       event (evbx evbxnop1 evb2 regmem tlogicop),
                1101001?-       event (evbx evbxnop1 evb2 regmem),

                1101010?-       event (evbchknext evb2),
                11010110-       event (err5),
                11010111-       event (evb1 tb),
                11011???-       event (err5 ),  **ESC instruction
                111000??-       event (evb2 evbjumpshort),

                111001??-       event (evb2 ta),
                11101000-       event (evb3 call stackpush t13),
                11101001-       event (evb3 evbjump tf),
                11101010-       event (evb5 evbjump tf),
                11101011-       event (evb2 evbjumpshort tf),

                111011??-       event (evb1 t8),
                11110000-       event (evb1 t2),
                11110001-       event (err5),
                1111001?-       event (evb1 evbnoimp),
                1111010?-       event (evb1 t2),

                11110110-       event (evbx evbxnop2 evb3 regmem chkregmem1),
                11110111-       event (evbx evbxnop3 evb4 regmem chkregmem1),
                111110??-       event (evb1 t2),
                1111110?-       event (evb1 t2),
                11111110-       event (evbx evbxnop4 evb2 regmem1),

                11111111-       event (evbx evbxnop5 evb2 regmem2),
                default-        event err5,
        endcase;

*************************************************************************
**                                                                     **
**                    expected parameter values                        **
**                                                                     **
*************************************************************************

                ***************************************************
                **                                               **
                **             expected execution time           **
                **                                               **
                ***************************************************

when idecode then t2 then clk reset resetseq do
        ecyclecnt = hex 2;

when idecode then t3 then clk reset resetseq do
        ecyclecnt = hex 3;
```

```
when idecode then t4 then clk reset resetseq do
        ecyclecnt = hex 4;

when idecode then t5 then clk reset resetseq do
        ecyclecnt = hex 5;

when idecode then t8 then clk reset resetseq do
        ecyclecnt = hex 8;

when idecode then ta then clk reset resetseq do
        ecyclecnt = hex a;

when idecode then tb then clk reset resetseq do
        ecyclecnt = hex b;

when idecode then tc then clk reset resetseq do
        ecyclecnt = hex c;

when idecode then tf then clk reset resetseq do
        ecyclecnt = hex f;

when idecode then t11 then clk reset resetseq do
        ecyclecnt = hex 11;

when idecode then t12 then clk reset resetseq do
        ecyclecnt = hex 12;

when idecode then t13 then clk reset resetseq do
        ecyclecnt = hex 13;

when idecode then t18 then clk reset resetseq do
        ecyclecnt = hex 18;

when idecode then t1b then clk reset resetseq do
        ecyclecnt = hex 1b;

when idecode then t34 then clk reset resetseq do
        ecyclecnt = hex 34;

when idecode then t35 then clk reset resetseq do
        ecyclecnt = hex 35;

when idecode then t10or4 then clk reset resetseq do
        ecyclecnt = hex 10;

next cycleend wait 20 reset jumptaken do
        ecyclecnt = hex 4;

when idecode then t35or4 then stackop reset cycleend or resetseq do
        ecyclecnt = hex 35;

when idecode then t35or4 then cycleend reset stackop or resetseq do
        ecyclecnt = hex 4;

when tmod then subyte wait 10 reset resetseq or cycleend do
        case ire[7:6],
        00-     event mod00,
        01-     event mod01,
        10-     event mod10,
        11-     ecyclecnt = hex 4,
        endcase;

when tmod then (mod00 or mod01 or mod10)+1 reset resetseq or cycleend do
        if ir[1] = 0 then
                ecyclecnt = hex 10
        else
                ecyclecnt = hex 9
```

```
        endif;

when mod00 wait 1 reset resetseq do
        case ire[2:0],
        000-    eatime = hex 7,
        001-    eatime = hex 8,
        010-    eatime = hex 8,
        011-    eatime = hex 7,
        100-    eatime = hex 5,
        101-    eatime = hex 5,
        110-    eatime = hex 6,
        111-    eatime = hex 5,
        endcase;

when (mod01 or mod10)+1 reset resetseq do
        case ire[2:0],
        000-    eatime = hex b,
        001-    eatime = hex c,
        010-    eatime = hex c,
        011-    eatime = hex b,
        100-    eatime = hex 9,
        101-    eatime = hex 9,
        110-    eatime = hex 9,
        111-    eatime = hex 9,
        endcase;

when av80 then subyte wait 10 reset resetseq do
        case ire[7:6],
        00-     ecyclecnt = hex 11
                event mod00,
        01-     ecyclecnt = hex 11
                event mod01,
        10-     ecyclecnt = hex 11
                event mod10,
        11-     ecyclecnt = hex 4,
        endcase;

when tlogicop then subyte wait 10 reset resetseq or cycleend do
        case ire[7:6],
        00-     ecyclecnt = hex f
                event mod00,
        01-     ecyclecnt = hex f
                event mod01,
        10-     ecyclecnt = hex f
                event mod10,
        11-     ecyclecnt = hex 2,
        endcase;

when tmov then subyte wait 10 reset resetseq or cycleend do
        case ire[7:6],
        00-     ecyclecnt = hex 9
                event mod00,
        01-     ecyclecnt = hex 8
                event mod01,
        10-     ecyclecnt = hex 9
                event mod10,
        11-     ecyclecnt = hex 2,
        endcase;

when leamov then subyte wait 10 reset resetseq or cycleend do
        ecyclecnt = hex 2
        case ire[7:6],
        00-     event mod00,
        01-     event mod01,
        10-     event mod10,
        11-     ,
        endcase;
```

```
when cycleend do
        ecyclecnt = ecyclecnt + eatime
        eatime = 0;

                ************************************************
                **                                            **
                **      expected number & size of data transfers  **
                **                                            **
                ************************************************

when call wait 20 reset resetseq or cycleend do
        xfernum = bin 01
        xfersize = bin 10;

when evretshort wait 20 reset resetseq or cycleend do
        xfernum = bin 01;

when evretlong wait 20 reset resetseq or cycleend do
        xfernum = bin 10;

when evbjumpshort then cycleend reset resetseq do
        jumpadd[19:8] = tar[19:8]
        jumpadd[7:0] =  tar[7:0] + ire + (4 bin 0, obytecnt);

when idecode then evbjumpshort reset resetseq do
        jumpflag = bin 01;

when idecode then evbjump reset resetseq do
        jumpflag = bin 10;

when idecode then evretshort or evretlong or eviret or call reset resetseq do
        jumpflag = bin 11;

when evbnoimp reset resetseq do
        display "instruction and/or function not implemented yet";

when memr reset resetseq or cycleend do
        case ir[0],
                0-      xfersize = bin 01
                        xfernum = bin 01,
                1-      xfersize = bin 10
                        xfernum = bin 01,
        endcase;

when memr16 reset resetseq or cycleend do
        xfersize = bin 10
        xfernum = bin 10;

when regmem then subyte wait 20 reset resetseq or cycleend do
        if ire[7:6] = bin 11 then       ** register to register operation
                xfersize = bin 00
                xfernum = bin 00
        else                            ** memory operation
                case ir[1:0],
                        00-     xfersize = bin 01
                                if readonlyflag = 1 then
                                        xfernum = bin 01
                                else    xfernum = bin 10
                                endif,
                        01-     xfersize = bin 10
                                if readonlyflag = 1 then
                                        xfernum = bin 01
                                else    xfernum = bin 10
                                endif,
                        10-     xfersize = bin 01
                                xfernum = bin 01,
```

```
                        11-     xfersize = bin 10
                                xfernum = bin 01,
                endcase
                readonlyflag = 0
        endif;

when regmem1 then subyte wait 20 reset resetseq or cycleend do
        if ire[7:6] = bin 11 then
                xfersize = bin 00
                xfernum = bin 00
        else    xfersize = bin 01
                xfernum = bin 10
        endif;

when regmem2 then subyte wait 20 reset resetseq or cycleend do
        if ire[7:6] = bin 11 then
                xfersize = bin 00
                xfernum = bin 00
        else    case ire[5:3],
                        00?-    xfersize = bin 10
                                xfernum = bin 10,
                        default- xfersize = bin 10
                                xfernum  = bin 01,
                endcase
        endif;

when chkregmem then subyte wait 10 reset resetseq or cycleend do
        if ire[5:3] = bin 111 then
                readonlyflag = 1
        endif;

when chkregmem1 then subyte wait 10 reset resetseq or cycleend do
        case ire[5:3],
                000-    readonlyflag = 1,
                001-    event err5,
                1??-    readonlyflag = 1
                        event evbnoimp,
                default- ,
        endcase;

                ************************************************
                **                                            **
                **      expected number of instruction bytes  **
                **                                            **
                ************************************************

when evb1 wait 20 reset resetseq do
        ebytecnt = bin 0001;

when evb2 wait 20 reset resetseq do
        ebytecnt = bin 0010;

when evb3 wait 20 reset resetseq do
        ebytecnt = bin 0011;

when evb4 wait 20 reset resetseq do
        ebytecnt = bin 0100;

when evb5 wait 20 reset resetseq do
        ebytecnt = bin 0101;

when evbx then subyte wait 20 reset resetseq or cycleend do
        case ire[7:0],
                00???110-       ebytecnt = ebytecnt + bin 0010,
                01??????-       ebytecnt = ebytecnt + bin 0001,
                10??????-       ebytecnt = ebytecnt + bin 0001,
        endcase;
```

```
when evbchk1 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                001-    event err5,
                100-    event err5,
                110-    event err5,
                default- ,
        endcase;

when evbchk2 then subyte wait 20 reset resetseq or cycleend do
        case ire[5],
                1-      event err5,
                default- ,
        endcase;

when evbxnop then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                000-    ,
                default- event err5,
        endcase;

when evbxnop1 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                110-    event err5,
                default- ,
        endcase;

when evbchknext then subyte wait 20 reset resetseq or cycleend do
        case ire[7:0],
                00001010-       ,
                default-        event err5,
        endcase;

when evbxnop2 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                000-    event evb3,
                001-    event err5,
                default- event evb2,
        endcase;

when evbxnop3 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                000-    event evb4,
                001-    event err5,
                default- event evb2,
        endcase;

when evbxnop4 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                00?-    ,
                default- event err5,
        endcase;

when evbxnop5 then subyte wait 20 reset resetseq or cycleend do
        case ire[5:3],
                010-    event (stackpush call),
                011-    event (stackpush2 call),
                100-    event evbjumpshort,
                101-    event evbjump,
                110-    event stackpush,
                111-    event err5,
                default- ,
        endcase;

**************************************************************************
**                                                                      **
**                      observed parameter values                       **
```

```
**                                                                      **
**************************************************************************

                ***************************************************
                **                                                 **
                **      observe instruction byte count             **
                **                                                 **
                ***************************************************

when idecode wait 50 reset resetseq do
        initflag = 0
        obytecnt = bin 0001;

when subyte do
        obytecnt = obytecnt + 1;

                ***************************************************
                **                                                 **
                **              observe execution time             **
                **                                                 **
                ***************************************************

**      Cycle counting mechanism

when cyclecnt then clk(1 to 0) reset resetseq do
        ocyclecnt = 8 hex 1
        event cycle1;

when cycle1 then clk(0 to 1) reset cycleend or resetseq do
        event cycle2;

when cycle2 then clk(1 to 0) reset cycleend or resetseq do
        ocyclecnt = ocyclecnt + 1
        event cycle1;

**      Detect if a cycle is aborted to grant EU request

when ifetch then 5*clk reset resetseq do
        event halfcycle;

when ifetch then halfcycle reset nobusop or resetseq do
        if msg = 1 then
                display "Bus request granted with out loss of a cycle"
        endif;

when ifetch then nobusop reset halfcycle or resetseq do
        event cycleabort;

when ifetch then clk(0 to 1) then cycleabort then clk(1 to 0)
                reset subyte or idecode or readmem or writemem or resetseq do
        if msg=1 then
                display "bus cycle aborted for EU request -- 1 clock cycle lost"
        endif
        s1 = 1;

**      Detect if EU has to wait for bus request

when idecode then reinit reset cycleend or resetseq do
                countflag = 1
                tempcount = 0
                event reinitcnt1;

when reinit2 then reinit reset cycleend or resetseq do
                countflag = 1
                tempcount = 0
                event reinitcnt1;
```

```
when reinitcnt1 then clk(1 to 0) reset cycleend or ifetch or resetseq do
        tempcount = tempcount + 1
        event reinitcnt2;

when reinitcnt2 then clk(0 to 1) reset cycleend or ifetch or resetseq do
        event reinitcnt1;

when idecode or idecode2 then reinit then ifetch reset resetseq or cycleend do
        case jumpflag,
                00-     s2 = s2 + tempcount + 1,
                01-     if obytecnt = obytecnt then
                                s2 = s2 + tempcount
                        else
                                s2 = s2 + tempcount + 1
                        endif,
                10-     if obytecnt = obytecnt then
                                s2 = s2 + tempcount
                        else
                                s2 = s2 + tempcount + 1
                        endif,
                11-     if obytecnt = obytecnt then
                                case ir,
                                11101000- ,
                                11000011- s2 = s2 + tempcount,
                                endcase
                        else
                                s2 = s2 + tempcount + 1
                        endif,
        endcase
        countflag = 0;

next clk reset resetseq do event idecode2;

next clk reset resetseq do event reinit2;

when  clk(0 to 1) then qnop wait 1 reset idecode or resetseq do
        if qp = bin 001 then
                if obytecnt = obytecnt then
                        if msg = 1 then
                                display "EU waiting  but not on empty Q"
                        endif
                else
                        if msg = 1 then
                                display "EU waiting on empty Q"
                        endif
                        if countflag = 0 then
                                s3 = s3 + 1
                        endif
                endif
        endif;

when idecode then evbjumpshort or evbjump
                reset resetseq or cycleend or subyte do
        qnopflag = 1;

next qnop wait 10 reset resetseq or cycleend do
        qnopflag = 0;

**      Execution cycle termination

when cycleend wait 20 reset resetseq do
        ocyclecnt = ocyclecnt - s1 - s2 - s3
        countflag = 0;

                ************************************************
                **                                            **
                **              observe data transfers        **
```

```
                **                                                    **
                ******************************************************

**      Determine segment access

when checkseg then 3*(clk(0 to 1)) reset resetseq do
        case a[17:16],
                00-     if msg = 1 then
                                display "ES: Extra Segment accessed"
                        endif,
                01-     if msg = 1 then
                                display "SS: Stack Segment accessed"
                        endif
                        stackflag = 1,
                10-     display "program memory accessed in execution cycle"
                        event err8,
                11-     if msg = 1 then
                                display "DS: Data Segment accessed"
                        endif,
        endcase;

when checkseg1 then 3*(clk(0 to 1)) reset resetseq do
        if instaddr[17:16] = bin 10 & msg = 1 then
                display "CS"
        endif;

**      Determine if a byte or word is transferred

when dsize then clk(0 to 1) reset resetseq do
        case(nbhes7,ad[0]),
                00-     datasize = datasize + 2,
                01-     datasize = datasize + 1,
                10-     datasize = datasize + 1,
                11-     datasize = 0,
        endcase;

**      Determine read-write sequences

when readmem then cycleend reset resetseq or idecode do
        datanum = bin 01
        event telldatasize;

when writemem then cycleend reset resetseq or idecode do
        datanum = bin 01
        event telldatasize;

when readmem then writemem then cycleend reset resetseq or idecode do
        datanum = bin 10
        event telldatasize;

when telldatasize reset resetseq do
        if msg = 1 then
                case datasize,
                        00-     display ("No data transfer at  = ",time ),
                        01-     display ("One byte transferred at  = ",time ),
                        10-     display ("Two bytes transferred at  = ",time ),
                        11-     display ("Three bytes transferred at  = ",time),
                endcase
        endif;

when intack or readio or writeio reset resetseq do
        display "tests not implemented yet";

when prochalt reset resetseq do
        display ("PROCESSOR HALTED at =", time);
```

```
*****************************************************************************
**                                                                         **
**                        calculate next address                           **
**                                                                         **
*****************************************************************************

when cycleend reset resetseq do
        nextadd = tar + (16 bin 0, obytecnt);

*****************************************************************************
**                                                                         **
**                       calculate stack address                           **
**                                                                         **
*****************************************************************************

when prepstart then stackflag(0 to 1) reset resetseq do
        stackadd = tempstackadd;

when stackpush then writemem then clk(0 to 1) reset resetseq do
        stackadd = stackadd - 2
        tempstackadd = ad[15:0]
        event stackop;

when stackpop then readmem then clk(0 to 1) reset resetseq do
        stackadd = stackadd + 2
        tempstackadd = ad[15:0] + 2
        event stackop;

when stackpush2 then writemem then clk(0 to 1) reset resetseq do
        stackadd = stackadd - 4
        tempstackadd = ad[15:0]
        event stackop;

when stackpop2 then readmem then clk(0 to 1) reset resetseq do
        stackadd = stackadd + 4
        tempstackadd = ad[15:0] + 4
        event stackop;

when stackpush3 then writemem then clk(0 to 1) reset resetseq do
        stackadd = stackadd - 6
        tempstackadd = ad[15:0]
        event stackop;

when stackpop3 then readmem then clk(0 to 1) reset resetseq do
        stackadd = stackadd + 6
        tempstackadd = ad[15:0] + 6
        event stackop;


*****************************************************************************
**                                                                         **
**                    instruction parameter verification                   **
**                                                                         **
*****************************************************************************

              ***************************************************
              **                                                **
              **      instruction execution time verification   **
              **                                                **
              ***************************************************

when cycleend wait 50 reset resetseq do
        if ocyclecnt = ecyclecnt then
                if msg = 1 then
                        display "Execution Time verified"
                endif
        else
```

```
                event err1
        endif;

                ***************************************************
                **                                               **
                **              byte count verification          **
                **                                               **
                ***************************************************

when cycleend reset resetseq do
        if ebytecnt = obytecnt then
                if msg = 1 then
                        display "Byte Count verified"
                endif
        else
                event err2
        endif;

                ***************************************************
                **                                               **
                **              data transfer verification       **
                **                                               **
                ***************************************************

when telldatasize wait 1 reset resetseq do
        if datanum = xfernum then
                if msg = 1 then
                        display "Data Count verified"
                endif
        else
                event err3
        endif
        if datasize = xfersize then
                if msg = 1 then
                        display "Data Size verified"
                endif
        else
                event err4
        endif
        event cleatimerparms;

when cleatimerparms wait 1 reset resetseq do
        datanum = bin 00
        datasize = bin 00;

*************************************************************************
**                                                                     **
**                      next address verification                      **
**                                                                     **
*************************************************************************

when idecode wait 10 reset resetseq do
        case jumpflag,
                00-     if nextadd[19:0] = tar[19:0] then
                                if msg = 1 then
                                        display ("Next Address verified")
                                endif
                        else
                                event err7
                        endif,
                01-     if nextadd[19:0] = tar[19:0] then
                                if msg = 1 then
                                        display ("Next Address verified -- non jump opcode")
                                endif
                        else if jumpadd[19:0] = tar[19:0] then
                                if msg = 1 then
                                        display ("Next Address verified -- jump taken")
```

```
                        endif
                        event jumptaken
                    else
                        event err7
                    endif
                endif,
        10-     if msg = 1 then
                        display ("Jump taken")
                endif,
        11-     ,
    endcase
    jumpflag = bin 00;

*************************************************************************
**                                                                     **
**                    stack address verification                       **
**                                                                     **
*************************************************************************

when stackop then clk reset resetseq do
    if stackflag = 1 then
        if stackadd = tempstackadd then
            if msg = 1 then
                display "Stack Address verified"
            endif
        else
            event err6
        endif
    endif;
*************************************************************************
**                                                                     **
**          Instruction Cycle Count Mechanism For Error Latency        **
**                                                                     **
*************************************************************************

when seuin do
    instcnt = hex 0
    event instcnt2;

when instcnt2 then cycleend do
    instcnt = instcnt + 1
    event instcnt1;

when instcnt1 then cycleend do
    instcnt = instcnt + 1
    event instcnt2;

*************************************************************************
**                                                                     **
**                    Error Messages                                   **
**                                                                     **
*************************************************************************

when err1 wait 1 reset resetseq do
    display ("ERROR: Instruction execution time incorrect")
    display ("ERROR #1: time= ",time,,"Instruction cycles= ",, instcnt);

when err2 wait 1 reset resetseq do
    display ("ERROR: Byte count incorrect")
    display ("ERROR #2: time= ",time,,"Instruction cycles= ",, instcnt);

when err3  wait 1 reset resetseq do
    display ("ERROR: Number of data transfers incorrect")
    display ("ERROR #3: time= ",time,,"Instruction cycles= ",, instcnt);

when err4 wait 1 reset resetseq do
    display ("ERROR: Size of data transfer incorrect")
```

```
        display ("ERROR #4: time= ",time,,"Instruction cycles= ",, instcnt);

when err5 wait 1 reset resetseq do
        display ("ERROR: Illegal Opcode detected")
        display ("ERROR #5: time= ",time,,"Instruction cycles= ",, instcnt);

when err6 wait 1 reset resetseq do
        display ("ERROR: Incorrect stack address")
        display ("ERROR #6: time= ",time,,"Instruction cycles= ",, instcnt);

when err7 wait 1 reset resetseq do
        display ("ERROR: Incorrect next address")
        display ("ERROR #7: time= ",time,,"Instruction cycles= ",, instcnt);

when err8 wait 1 reset resetseq do
        display ("ERROR: Program memory accessed for data")
        display ("ERROR #8: time= ",time,,"Instruction cycles= ",, instcnt);

when err9 wait 1 reset resetseq do
        display ("ERROR: Wrong Q update signals from CPU")
        display ("ERROR #9: time= ",time,,"Instruction cycles= ",, instcnt);

when err10 wait 1 reset resetseq do
        display ("ERROR: Access to undefined memory location")
        display ("ERROR #10: time= ",time,,"Instruction cycles= ",, instcnt);

when err1 or err2 or err3 or err4 or err5 or err6 or err7 or err8 or err9
                or err10 then 50*clk do
        finish.

****)
```

Appendix B
Prototype System Configuration

```
cct       system   ( clk, resetin, ready, faulttype[2:0], faultbit[3:0],
                     seuin, msg);

in8086    mpu      ( gnd, ad[14], ad[13], ad[12], ad[11], ad[10], ad[9],
                     ad[8], ad[7], ad[6], ad[5], ad[4], ad[3], ad[2], ad[1],
                     ad[0], gnd, gnd, clk, gnd, resetin, ready, gnd, qs1,
                     qs0, ns0, ns1, ns2,  ,  ,  ,  , gnd, nbhe, ad[19],
                     ad[18], ad[17], ad[16], ad[15], vcc,
                     faulttype[2:0], faultbit[3:0], chkreg, seuin);

in8288    bc       ( vcc, clk, ns1, dtnr, ale, gnd, nmrdc,   ,
                     nmwtc, gnd,  ,  ,  ,  , vcc, den,  , ns2, ns0, vcc );

in8282    latch1   ( ad[0], ad[1], ad[2], ad[3], ad[4], ad[5], ad[6], ad[7],
                     gnd, gnd, ale, a7, a6, a5, a4, a3, a2, a1, a0, vcc)

          latch2   ( ad[8], ad[9], ad[10], ad[11], ad[12], ad[13], ad[14],
                     ad[15], gnd, gnd, ale, a15, a14, a13, a12, a11,
                     a10, a9, a8, vcc )

          latch3   ( ad[16], ad[17], ad[18], ad[19], nbhe,  ,  ,  , gnd, gnd,
                     ale,  ,  ,  , hibyte, a19, a18, a17, a16, vcc );

in8286    trans1   ( ad[0], ad[1], ad[2], ad[3], ad[4], ad[5], ad[6], ad[7],
                     nden, gnd, dtnr, d7, d6, d5, d4, d3, d2, d1, d0, vcc)

          trans2   ( ad[8], ad[9], ad[10], ad[11], ad[12], ad[13], ad[14],
                     ad[15], nden, gnd, dtnr, d15, d14, d13, d12, d11, d10,
                     d9, d8, vcc );

sn7404    inv      ( den, nden,  ,  ,  ,  , gnd,  ,  ,  ,  ,  , vcc );

decoder   decode   ( a19, a18, a17, a16, a15, a14, a13, a12, a11, a10, a9,
                     a8, a7, a6, hibyte, a0, nmrdc, nmwtc, coderom0, coderom1,
                     datarom0, datarom1, dataram0, dataram1, resetin );

rom0      crom0    ( a8, a7, a6, a5, a4, a3, a2, a1, d7, d6, d5, d4, d3, d2,
                     d1, d0, coderom0, nmrdc );

rom1      crom1    ( a8, a7, a6, a5, a4, a3, a2, a1, d15, d14, d13, d12, d11,
                     d10, d9, d8, coderom1, nmrdc );

ram0      dram0    ( a8, a7, a6, a5, a4, a3, a2, a1, d7, d6, d5, d4, d3, d2,
                     d1, d0, dataram0, nmwtc );

ram1      dram1    ( a8, a7, a6, a5, a4, a3, a2, a1, d15, d14, d13, d12, d11,
                     d10, d9, d8, dataram1, nmwtc );
```

```
rom0d   drom0   ( a8, a7, a6, a5, a4, a3, a2, a1, d7, d6, d5, d4, d3, d2,
                  d1, d0, datarom0, nmrdc );

rom1d   drom1   ( a8, a7, a6, a5, a4, a3, a2, a1, d15, d14, d13, d12, d11,
                  d10, d9, d8, datarom1, nmrdc );

tfd     wdp     ( ad[19], ad[18], ad[17], ad[16], ad[15], ad[14], ad[13],
                  ad[12], ad[11], ad[10], ad[9], ad[8], ad[7], ad[6], ad[5],
                  ad[4], ad[3], ad[2], ad[1], ad[0], clk, resetin, qs0, qs1,
                  ns0, ns1, ns2, nbhe, ready, msg, seuin );

tri     ad[19:0];

supply0 gnd;
supply1 vcc;

input   clk, resetin, ready, faulttype[2:0], faultbit[3:0], seuin, msg;.
```

Appendix C
Test Program Listing

```
8086/8087/8088 MACRO ASSEMBLER    TEST


RTCS/UDI V4.0 - 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE TEST
OBJECT MODULE PLACED IN ZIA.OBJ
ASSEMBLER INVOKED BY:  B:ASM86 ZIA.ASM


LOC  OBJ                  LINE     SOURCE

                             1     ;-----------------------------------------
                             2     ;   Program Name : 8086 test program
                             3     ;   Author       : Mohammad Ziaullah Khan
                             4     ;   Date         : October 1988
                             5     ;   Purpose      : To exercise 8086 for
                             6     ;                : transient faults detection
                             7     ;                : experiments.
                             8     ;-----------------------------------------
                             9     name    test
                            10     cgroup  group   code
                            11     dgroup  group   data
                            12     sgroup  group   stack
                            13     assume  cs:cgroup, ds:dgroup, ss:sgroup
                            14     ;
----                        15     stack   segment  stack  'stack'
0000 (10                    16             dw       10     dup(0)
     0000
     )
0014                        17     stack_top label  word
----                        18     stack   ends
                            19     ;
                            20     ;
----                        21     data    segment  public 'data'
0100                        22             org    0100h
0100 0400                   23     arrlen  dw     04d
0102 0700                   24     arrval  dw     07d, 04d, 01d, 09d
0104 0400
0106 0100
0108 0900
010A ????                   25     result  dw     ?
010C ????                   26     onecnt  dw     ?
----                        27     data    ends
                            28     ;
                            29     ; This program finds the sum of a data array
                            30     ; and then calculates the number of ones in
                            31     ; the sum. Upon completion the sum is in AX
                            32     ; and number of ones is in BX.
```

```
                                33      ;
----                            34      code    segment public  'code'
  0030                          35      array   equ   0030h     ; address of data
0000                            36              org   0h
0000 BD3000                     37      start:  mov   bp, array ; initialize array ptr
0003 E81700                     38              call  sum       ; find array sum
0006 BA1000                     39              mov   dx, 016d  ; set up for ones
0009 33DB                       40              xor   bx, bx    ; counting
000B D1D8                       41      next:   rcr   ax,1      ; check lowest bit
000D 7301                       42              jnc   zero      ; is it 0?
000F 43                         43              inc   bx        ; update ones count
0010 4A                         44      zero:   dec   dx        ; update bit count
0011 75F8                       45              jnz   next      ; check next bit
0013 D1D8                       46              rcr   ax, 1     ; restore data
0015 A30A01             R       47              mov   result, ax ; save sum
0018 891E0C01           R       48              mov   onecnt, bx ; save #ones
001C F4                         49              hlt             ; quit
                                50      ;
                                51      ; This subroutine calculates the sum of the
                                52      ; array. The first word is assumed to be the
                                53      ; length of array followed by the data values.
                                54      ;
001D                            55      sum     proc  near
001D 33C0                       56              xor   ax, ax    ; initialize count
001F 8B4E00                     57              mov   cx, [bp]  ; get first data value
0022 45                         58      again:  inc   bp        ; update array pointer
0023 45                         59              inc   bp        ;
0024 134600                     60              adc   ax, [bp]  ; add the data
0027 49                         61              dec   cx        ; decrement data count
0028 75F8                       62              jnz   again     ; done?
002A C3                         63              ret             ; return
                                64      sum     endp
----                            65              code  ends
                                66      end     start

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Appendix D
Fault-free Simulation Run

```
        THE INPUT WAVE FORM

   1 : waveform wave
   2 : stimulus clk, seuin = 0;
   3 : stimulus ready, resetin, chkreg = 1;
   4 : stimulus msg = 1;
   5 : stimulus faulttype[2:0], faultbit[3:0] = 0;
   6 : 0        clk = by 500 to 1m change0(250,+250);
   7 : 1400     resetin = 0;
   8 : 499999   chkreg = 0;
   9 : 500k     finish.

*
        THE SIMULATION COMMANDS
*
sim system wave
dch ( ,,,"====>" ,,"MPU_IR = ", (mpu_ir) hex ,, "WDP_IR = ",(wdp_ir) hex ).
*
        THE SIMULATION OUTPUT
*
1 OF IN8086
1 OF IN8288
3 OF IN8282
2 OF IN8286
1 OF SN7404
1 OF DECODER
1 OF ROM0
1 OF ROM1
1 OF RAM0
1 OF RAM1
1 OF ROM0D
1 OF ROM1D
1 OF TFD
24 OF PARTLATCH
Delayscale = NS
Time to load 34.59 cpu secs.
**WARNING LATCH3_LATCH[5]_D not connected to an output
**WARNING LATCH3_LATCH[6]_D not connected to an output
**WARNING LATCH3_LATCH[7]_D not connected to an output
**WARNING INV_A[2] not connected to an output
**WARNING INV_A[3] not connected to an output
**WARNING INV_A[4] not connected to an output
**WARNING INV_A[5] not connected to an output
**WARNING INV_A[6] not connected to an output
SYSTEM LOADED OK
Initialisation complete   cpu secs = 1.93   (total = 1.93)
                  MM           WW
                  PP           DD
```

```
                UU          PP
                __          __
                II          II
                RR          RR
                [[          [[
                73          73
                ]]          ]]


   ====> MPU_IR = XX  WDP_IR = XX

WDP:BUS:Passive at =        80
WDP:BUS:Instruction fetch at =      6085
WDP:Q reinitialized at =      6260
WDP:No quueue operation at =      6760
WDP:EU waiting but not on empty Q
WDP:BUS:Passive at =      7088
WDP:bus cycle aborted for EU request -- 1 clock cycle lost
WDP:BUS:Instruction fetch at =      7821
   ====> MPU_IR = BD  WDP_IR = XX

WDP:First byte read from Q at =      8260
   ====> MPU_IR = BD  WDP_IR = BD

WDP:Jump taken
WDP:No quueue operation at =      8760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      9088
WDP:Subsequent byte at =      9260
WDP:No quueue operation at =      9760
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =      9821
WDP:Subsequent byte at =     10260
   ====> MPU_IR = E8  WDP_IR = BD

WDP:Byte Count verified
WDP:First byte read from Q at =     10760
   ====> MPU_IR = E8  WDP_IR = E8
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     11088
WDP:Q reinitialized at =     11260
WDP:No quueue operation at =     11760
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =     11821
WDP:Subsequent byte at =     12260
WDP:BUS:Passive at =     12589
WDP:Subsequent byte at =     12760
WDP:No quueue operation at =     13260
WDP:EU waiting but not on empty Q
WDP:BUS:writing memory at =     13320
WDP:BUS:Passive at =     14588
WDP:SS: Stack Segment accessed
WDP:BUS:Instruction fetch at =     15321
WDP:Q reinitialized at =     15760
WDP:No quueue operation at =     16260
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     16588
WDP:BUS:Instruction fetch at =     17321
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     18588
WDP:BUS:Instruction fetch at =     19321
```

```
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      20588
   ====> MPU_IR = 33  WDP_IR = E8

WDP:Byte Count verified
WDP:Two bytes transferred at =      21255
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =      21260
   ====> MPU_IR = 33  WDP_IR = 33
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =      21321
   ====> MPU_IR = 31  WDP_IR = 33
   ====> MPU_IR = 33  WDP_IR = 33

WDP:Subsequent byte at =      21760
WDP:No quueue operation at =      22260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      22588
   ====> MPU_IR = 8B  WDP_IR = 33

WDP:Byte Count verified
WDP:First byte read from Q at =      23260
   ====> MPU_IR = 8B  WDP_IR = 8B
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =      23321
WDP:Subsequent byte at =      23760
WDP:No quueue operation at =      24260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      24588
WDP:Subsequent byte at =      26760
WDP:BUS:Instruction fetch at =      26821
WDP:No quueue operation at =      27260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      28088
WDP:BUS:reading memory at =      28821
WDP:BUS:Passive at =      30088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 45  WDP_IR = 8B

WDP:Byte Count verified
WDP:Two bytes transferred at =      31755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =      31760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =      32260
WDP:Byte Count verified
WDP:First byte read from Q at =      32760
WDP:Next Address verified
WDP:Execution Time verified
WDP:BUS:Instruction fetch at =      32821
WDP:No quueue operation at =      33260
   ====> MPU_IR = 13  WDP_IR = 45

WDP:Byte Count verified
WDP:First byte read from Q at =      33760
   ====> MPU_IR = 13  WDP_IR = 13
WDP:Next Address verified
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 11  WDP_IR = 13

WDP:BUS:Passive at =     34088
WDP:Subsequent byte at =     34260
WDP:No quueue operation at =     34760
WDP:BUS:Instruction fetch at =     34821
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     36088
WDP:Subsequent byte at =     37260
WDP:No quueue operation at =     37760
WDP:BUS:reading memory at =     38821
WDP:BUS:Passive at =     40088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     42755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     42760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     42821
WDP:No quueue operation at =     43260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     43760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     44088
WDP:Subsequent byte at =     44260
WDP:No quueue operation at =     44760
WDP:Q reinitialized at =     48260
WDP:BUS:Instruction fetch at =     48321
WDP:No quueue operation at =     48760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     49588
WDP:BUS:Instruction fetch at =     50321
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     51588
WDP:Byte Count verified
WDP:First byte read from Q at =     51760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified -- jump taken
WDP:Execution Time verified
WDP:No quueue operation at =     52260
WDP:BUS:Instruction fetch at =     52321
WDP:Byte Count verified
WDP:First byte read from Q at =     52760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     53260
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 13  WDP_IR = 45
```

```
WDP:BUS:Passive at =     53588
WDP:Byte Count verified
WDP:First byte read from Q at =     53760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     54260
WDP:BUS:Instruction fetch at =     54321
WDP:No quueue operation at =     54760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     55588
WDP:BUS:Instruction fetch at =     56321
WDP:Subsequent byte at =     57260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     57588
WDP:No quueue operation at =     57760
WDP:BUS:reading memory at =     58821
WDP:BUS:Passive at =     60088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     62755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     62760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     62821
WDP:No quueue operation at =     63260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     63760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     64088
WDP:Subsequent byte at =     64260
WDP:No quueue operation at =     64760
WDP:Q reinitialized at =     68260
WDP:BUS:Instruction fetch at =     68321
WDP:No quueue operation at =     68760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     69588
WDP:BUS:Instruction fetch at =     70321
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     71588
WDP:Byte Count verified
WDP:First byte read from Q at =     71760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified -- jump taken
WDP:Execution Time verified
```

```
WDP:No quueue operation at =     72260
WDP:BUS:Instruction fetch at =     72321
WDP:Byte Count verified
WDP:First byte read from Q at =     72760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     73260
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 13  WDP_IR = 45

WDP:BUS:Passive at =     73588
WDP:Byte Count verified
WDP:First byte read from Q at =     73760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     74260
WDP:BUS:Instruction fetch at =     74321
WDP:No quueue operation at =     74760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     75588
WDP:BUS:Instruction fetch at =     76321
WDP:Subsequent byte at =     77260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     77588
WDP:No quueue operation at =     77760
WDP:BUS:reading memory at =     78821
WDP:BUS:Passive at =     80088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     82755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     82760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     82821
WDP:No quueue operation at =     83260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     83760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     84088
WDP:Subsequent byte at =     84260
WDP:No quueue operation at =     84760
WDP:Q reinitialized at =     88260
WDP:BUS:Instruction fetch at =     88321
WDP:No quueue operation at =     88760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     89588
WDP:BUS:Instruction fetch at =     90321
WDP:Bus request granted with out loss of a cycle
```

```
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     91588
WDP:Byte Count verified
WDP:First byte read from Q at =     91760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified -- jump taken
WDP:Execution Time verified
WDP:No quueue operation at =     92260
WDP:BUS:Instruction fetch at =     92321
WDP:Byte Count verified
WDP:First byte read from Q at =     92760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     93260
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 13  WDP_IR = 45

WDP:BUS:Passive at =     93588
WDP:Byte Count verified
WDP:First byte read from Q at =     93760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     94260
WDP:BUS:Instruction fetch at =     94321
WDP:No quueue operation at =     94760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     95588
WDP:BUS:Instruction fetch at =     96321
WDP:Subsequent byte at =     97260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     97588
WDP:No quueue operation at =     97760
WDP:BUS:reading memory at =     98821
WDP:BUS:Passive at =    100088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =    102755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    102760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    102821
WDP:No quueue operation at =    103260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =    103760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    104088
WDP:Subsequent byte at =    104260
```

```
WDP:No quueue operation at =    104760
WDP:BUS:Instruction fetch at =    104821
   ====> MPU_IR = C3  WDP_IR = 75

WDP:BUS:Passive at =    105589
WDP:Byte Count verified
WDP:First byte read from Q at =    105760
   ====> MPU_IR = C3  WDP_IR = C3

WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified
WDP:No quueue operation at =    106260
WDP:BUS:reading memory at =    106320
WDP:Stack Address verified
WDP:BUS:Passive at =    107588
WDP:SS: Stack Segment accessed
WDP:BUS:Instruction fetch at =    108321
WDP:Q reinitialized at =    109260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    109588
WDP:No quueue operation at =    109760
WDP:EU waiting but not on empty Q
WDP:BUS:Instruction fetch at =    110321
   ====> MPU_IR = BA  WDP_IR = C3

WDP:Byte Count verified
WDP:Two bytes transferred at =    110755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    110760
   ====> MPU_IR = BA  WDP_IR = BA

WDP:Execution Time verified
WDP:No quueue operation at =    111260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    111588
WDP:Subsequent byte at =    111760
WDP:No quueue operation at =    112260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    112321
WDP:Subsequent byte at =    112760
   ====> MPU_IR = 33  WDP_IR = BA

WDP:Byte Count verified
WDP:First byte read from Q at =    113260
   ====> MPU_IR = 33  WDP_IR = 33
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    113588
WDP:Q reinitialized at =    113760
WDP:No quueue operation at =    114260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    114321
   ====> MPU_IR = 31  WDP_IR = 33
   ====> MPU_IR = 33  WDP_IR = 33

WDP:Subsequent byte at =    114760
WDP:No quueue operation at =    115260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    115588
   ====> MPU_IR = D1  WDP_IR = 33

WDP:Byte Count verified
WDP:First byte read from Q at =    116260
```

```
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    116321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    116760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    117260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    117588
WDP:Q reinitialized at =    117760
WDP:No quueue operation at =    118260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    118321
WDP:Subsequent byte at =    118760
WDP:No quueue operation at =    119260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    119588
   ====> MPU_IR = 43  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    120260
   ====> MPU_IR = 43  WDP_IR = 43
WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    120321
WDP:No quueue operation at =    120760
   ====> MPU_IR = 4A  WDP_IR = 43

WDP:Byte Count verified
WDP:First byte read from Q at =    121260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    121588
WDP:No quueue operation at =    121760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    122260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    122321
WDP:Subsequent byte at =    122760
WDP:No quueue operation at =    123260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    123588
WDP:Q reinitialized at =    127260
WDP:BUS:Instruction fetch at =    127321
WDP:No quueue operation at =    127760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    128588
WDP:BUS:Instruction fetch at =    129321
```

```
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    130260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    130588
WDP:Q reinitialized at =    130760
WDP:No quueue operation at =    131260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    131321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    131760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    132260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    132588
WDP:Q reinitialized at =    132760
WDP:No quueue operation at =    133260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    133321
WDP:Subsequent byte at =    133760
WDP:No quueue operation at =    134260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    134588
WDP:Q reinitialized at =    138260
WDP:BUS:Instruction fetch at =    138321
WDP:No quueue operation at =    138760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    139588
WDP:BUS:Instruction fetch at =    140321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    141260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    141588
WDP:No quueue operation at =    141760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    142260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    142321
WDP:Subsequent byte at =    142760
WDP:No quueue operation at =    143260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    143588
WDP:Q reinitialized at =    147260
```

```
WDP:BUS:Instruction fetch at =    147321
WDP:No quueue operation at =    147760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    148588
WDP:BUS:Instruction fetch at =    149321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    150260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    150588
WDP:Q reinitialized at =    150760
WDP:No quueue operation at =    151260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    151321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    151760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    152260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    152588
WDP:Q reinitialized at =    152760
WDP:No quueue operation at =    153260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    153321
WDP:Subsequent byte at =    153760
WDP:No quueue operation at =    154260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    154588
   ====> MPU_IR = 43  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    155260
   ====> MPU_IR = 43  WDP_IR = 43
WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    155321
WDP:No quueue operation at =    155760
   ====> MPU_IR = 4A  WDP_IR = 43

WDP:Byte Count verified
WDP:First byte read from Q at =    156260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    156588
WDP:No quueue operation at =    156760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    157260
```

```
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    157321
WDP:Subsequent byte at =    157760
WDP:No quueue operation at =    158260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    158588
WDP:Q reinitialized at =    162260
WDP:BUS:Instruction fetch at =    162321
WDP:No quueue operation at =    162760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    163588
WDP:BUS:Instruction fetch at =    164321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    165260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    165588
WDP:Q reinitialized at =    165760
WDP:No quueue operation at =    166260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    166321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    166760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    167260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    167588
WDP:Q reinitialized at =    167760
WDP:No quueue operation at =    168260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    168321
WDP:Subsequent byte at =    168760
WDP:No quueue operation at =    169260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    169588
WDP:Q reinitialized at =    173260
WDP:BUS:Instruction fetch at =    173321
WDP:No quueue operation at =    173760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    174588
WDP:BUS:Instruction fetch at =    175321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    176260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    176588
WDP:No quueue operation at =    176760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    177260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    177321
WDP:Subsequent byte at =    177760
WDP:No quueue operation at =    178260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    178588
WDP:Q reinitialized at =    182260
WDP:BUS:Instruction fetch at =    182321
WDP:No quueue operation at =    182760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    183588
WDP:BUS:Instruction fetch at =    184321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    185260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    185588
WDP:Q reinitialized at =    185760
WDP:No quueue operation at =    186260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    186321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    186760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    187260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    187588
WDP:Q reinitialized at =    187760
WDP:No quueue operation at =    188260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    188321
WDP:Subsequent byte at =    188760
WDP:No quueue operation at =    189260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    189588
   ====> MPU_IR = 43  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    190260
   ====> MPU_IR = 43  WDP_IR = 43
WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified
```

```
WDP:BUS:Instruction fetch at =    190321
WDP:No quueue operation at =    190760
   ====> MPU_IR = 4A  WDP_IR = 43

WDP:Byte Count verified
WDP:First byte read from Q at =    191260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    191588
WDP:No quueue operation at =    191760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    192260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    192321
WDP:Subsequent byte at =    192760
WDP:No quueue operation at =    193260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    193588
WDP:Q reinitialized at =    197260
WDP:BUS:Instruction fetch at =    197321
WDP:No quueue operation at =    197760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    198588
WDP:BUS:Instruction fetch at =    199321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    200260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    200588
WDP:Q reinitialized at =    200760
WDP:No quueue operation at =    201260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    201321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    201760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    202260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    202588
WDP:Q reinitialized at =    202760
WDP:No quueue operation at =    203260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    203321
WDP:Subsequent byte at =    203760
WDP:No quueue operation at =    204260
WDP:Bus request granted with out loss of a cycle
```

```
WDP:BUS:Passive at =    204588
WDP:Q reinitialized at =    208260
WDP:BUS:Instruction fetch at =    208321
WDP:No quueue operation at =    208760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    209588
WDP:BUS:Instruction fetch at =    210321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    211260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    211588
WDP:No quueue operation at =    211760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    212260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    212321
WDP:Subsequent byte at =    212760
WDP:No quueue operation at =    213260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    213588
WDP:Q reinitialized at =    217260
WDP:BUS:Instruction fetch at =    217321
WDP:No quueue operation at =    217760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    218588
WDP:BUS:Instruction fetch at =    219321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    220260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    220588
WDP:Q reinitialized at =    220760
WDP:No quueue operation at =    221260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    221321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    221760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    222260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    222588
WDP:Q reinitialized at =    222760
```

```
WDP:No quueue operation at =    223260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    223321
WDP:Subsequent byte at =    223760
WDP:No quueue operation at =    224260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    224588
WDP:Q reinitialized at =    228260
WDP:BUS:Instruction fetch at =    228321
WDP:No quueue operation at =    228760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    229588
WDP:BUS:Instruction fetch at =    230321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    231260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    231588
WDP:No quueue operation at =    231760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    232260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    232321
WDP:Subsequent byte at =    232760
WDP:No quueue operation at =    233260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    233588
WDP:Q reinitialized at =    237260
WDP:BUS:Instruction fetch at =    237321
WDP:No quueue operation at =    237760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    238588
WDP:BUS:Instruction fetch at =    239321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    240260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    240588
WDP:Q reinitialized at =    240760
WDP:No quueue operation at =    241260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    241321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    241760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    242260
```

```
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    242588
WDP:Q reinitialized at =    242760
WDP:No quueue operation at =    243260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    243321
WDP:Subsequent byte at =    243760
WDP:No quueue operation at =    244260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    244588
WDP:Q reinitialized at =    248260
WDP:BUS:Instruction fetch at =    248321
WDP:No quueue operation at =    248760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    249588
WDP:BUS:Instruction fetch at =    250321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    251260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    251588
WDP:No quueue operation at =    251760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    252260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    252321
WDP:Subsequent byte at =    252760
WDP:No quueue operation at =    253260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    253588
WDP:Q reinitialized at =    257260
WDP:BUS:Instruction fetch at =    257321
WDP:No quueue operation at =    257760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    258588
WDP:BUS:Instruction fetch at =    259321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    260260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    260588
WDP:Q reinitialized at =    260760
WDP:No quueue operation at =    261260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    261321
   ====> MPU_IR = 51  WDP_IR = D1
```

```
WDP:Subsequent byte at =    261760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    262260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    262588
WDP:Q reinitialized at =    262760
WDP:No quueue operation at =    263260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    263321
WDP:Subsequent byte at =    263760
WDP:No quueue operation at =    264260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    264588
WDP:Q reinitialized at =    268260
WDP:BUS:Instruction fetch at =    268321
WDP:No quueue operation at =    268760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    269588
WDP:BUS:Instruction fetch at =    270321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    271260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    271588
WDP:No quueue operation at =    271760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    272260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    272321
WDP:Subsequent byte at =    272760
WDP:No quueue operation at =    273260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    273588
WDP:Q reinitialized at =    277260
WDP:BUS:Instruction fetch at =    277321
WDP:No quueue operation at =    277760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    278588
WDP:BUS:Instruction fetch at =    279321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    280260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    280588
WDP:Q reinitialized at =    280760
WDP:No quueue operation at =    281260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    281321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    281760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    282260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    282588
WDP:Q reinitialized at =    282760
WDP:No quueue operation at =    283260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    283321
WDP:Subsequent byte at =    283760
WDP:No quueue operation at =    284260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    284588
WDP:Q reinitialized at =    288260
WDP:BUS:Instruction fetch at =    288321
WDP:No quueue operation at =    288760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    289588
WDP:BUS:Instruction fetch at =    290321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    291260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    291588
WDP:No quueue operation at =    291760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    292260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    292321
WDP:Subsequent byte at =    292760
WDP:No quueue operation at =    293260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    293588
WDP:Q reinitialized at =    297260
WDP:BUS:Instruction fetch at =    297321
WDP:No quueue operation at =    297760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    298588
WDP:BUS:Instruction fetch at =    299321
   ====> MPU_IR = D1  WDP_IR = 75
```

```
WDP:Byte Count verified
WDP:First byte read from Q at =    300260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    300588
WDP:Q reinitialized at =    300760
WDP:No quueue operation at =    301260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    301321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    301760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    302260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    302588
WDP:Q reinitialized at =    302760
WDP:No quueue operation at =    303260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    303321
WDP:Subsequent byte at =    303760
WDP:No quueue operation at =    304260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    304588
WDP:Q reinitialized at =    308260
WDP:BUS:Instruction fetch at =    308321
WDP:No quueue operation at =    308760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    309588
WDP:BUS:Instruction fetch at =    310321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    311260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    311588
WDP:No quueue operation at =    311760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    312260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    312321
WDP:Subsequent byte at =    312760
WDP:No quueue operation at =    313260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    313588
WDP:Q reinitialized at =    317260
WDP:BUS:Instruction fetch at =    317321
WDP:No quueue operation at =    317760
```

```
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    318588
WDP:BUS:Instruction fetch at =    319321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    320260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    320588
WDP:Q reinitialized at =    320760
WDP:No quueue operation at =    321260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    321321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    321760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    322260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    322588
WDP:Q reinitialized at =    322760
WDP:No quueue operation at =    323260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    323321
WDP:Subsequent byte at =    323760
WDP:No quueue operation at =    324260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    324588
WDP:Q reinitialized at =    328260
WDP:BUS:Instruction fetch at =    328321
WDP:No quueue operation at =    328760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    329588
WDP:BUS:Instruction fetch at =    330321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    331260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    331588
WDP:No quueue operation at =    331760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    332260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    332321
WDP:Subsequent byte at =    332760
```

```
WDP:No quueue operation at =    333260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    333588
WDP:Q reinitialized at =    337260
WDP:BUS:Instruction fetch at =    337321
WDP:No quueue operation at =    337760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    338588
WDP:BUS:Instruction fetch at =    339321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    340260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    340588
WDP:Q reinitialized at =    340760
WDP:No quueue operation at =    341260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    341321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    341760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    342260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    342588
WDP:Q reinitialized at =    342760
WDP:No quueue operation at =    343260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    343321
WDP:Subsequent byte at =    343760
WDP:No quueue operation at =    344260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    344588
WDP:Q reinitialized at =    348260
WDP:BUS:Instruction fetch at =    348321
WDP:No quueue operation at =    348760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    349588
WDP:BUS:Instruction fetch at =    350321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    351260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    351588
WDP:No quueue operation at =    351760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    352260
```

```
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    352321
WDP:Subsequent byte at =    352760
WDP:No quueue operation at =    353260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    353588
WDP:Q reinitialized at =    357260
WDP:BUS:Instruction fetch at =    357321
WDP:No quueue operation at =    357760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    358588
WDP:BUS:Instruction fetch at =    359321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    360260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    360588
WDP:Q reinitialized at =    360760
WDP:No quueue operation at =    361260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    361321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    361760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    362260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    362588
WDP:Q reinitialized at =    362760
WDP:No quueue operation at =    363260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    363321
WDP:Subsequent byte at =    363760
WDP:No quueue operation at =    364260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    364588
WDP:Q reinitialized at =    368260
WDP:BUS:Instruction fetch at =    368321
WDP:No quueue operation at =    368760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    369588
WDP:BUS:Instruction fetch at =    370321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    371260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    371588
WDP:No quueue operation at =    371760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    372260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    372321
WDP:Subsequent byte at =    372760
WDP:No quueue operation at =    373260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    373588
WDP:Q reinitialized at =    377260
WDP:BUS:Instruction fetch at =    377321
WDP:No quueue operation at =    377760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    378588
WDP:BUS:Instruction fetch at =    379321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    380260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    380588
WDP:Q reinitialized at =    380760
WDP:No quueue operation at =    381260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    381321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    381760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    382260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    382588
WDP:Q reinitialized at =    382760
WDP:No quueue operation at =    383260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    383321
WDP:Subsequent byte at =    383760
WDP:No quueue operation at =    384260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    384588
WDP:Q reinitialized at =    388260
WDP:BUS:Instruction fetch at =    388321
WDP:No quueue operation at =    388760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    389588
WDP:BUS:Instruction fetch at =    390321
   ====> MPU_IR = 4A  WDP_IR = 73
```

```
WDP:Byte Count verified
WDP:First byte read from Q at =    391260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    391588
WDP:No quueue operation at =    391760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    392260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    392321
WDP:Subsequent byte at =    392760
WDP:No quueue operation at =    393260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    393588
WDP:Q reinitialized at =    397260
WDP:BUS:Instruction fetch at =    397321
WDP:No quueue operation at =    397760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    398588
WDP:BUS:Instruction fetch at =    399321
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    400260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    400588
WDP:Q reinitialized at =    400760
WDP:No quueue operation at =    401260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    401321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    401760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    402260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    402588
WDP:Q reinitialized at =    402760
WDP:No quueue operation at =    403260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    403321
WDP:Subsequent byte at =    403760
WDP:No quueue operation at =    404260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    404588
WDP:Q reinitialized at =    408260
WDP:BUS:Instruction fetch at =    408321
WDP:No quueue operation at =    408760
```

```
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    409588
WDP:BUS:Instruction fetch at =    410321
   ====> MPU_IR = 4A  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    411260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified -- jump taken
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    411588
WDP:No quueue operation at =    411760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    412260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    412321
WDP:Subsequent byte at =    412760
WDP:No quueue operation at =    413260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    413588
   ====> MPU_IR = D1  WDP_IR = 75

WDP:Byte Count verified
WDP:First byte read from Q at =    414260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    414321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    414760
   ====> MPU_IR = A3  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    415260
   ====> MPU_IR = A3  WDP_IR = A3
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    415588
WDP:Q reinitialized at =    415760
WDP:No quueue operation at =    416260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    416321
WDP:Subsequent byte at =    416760
WDP:Subsequent byte at =    417260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    417588
WDP:No quueue operation at =    417760
WDP:EU waiting but not on empty Q
WDP:BUS:Instruction fetch at =    418321
WDP:BUS:Passive at =    418829
WDP:bus cycle aborted for EU request -- 1 clock cycle lost
WDP:BUS:writing memory at =    419320
WDP:BUS:Passive at =    420588
WDP:DS: Data Segment accessed
WDP:BUS:Instruction fetch at =    421321
```

```
   ====> MPU_IR = 89  WDP_IR = A3

WDP:Byte Count verified
WDP:Two bytes transferred at =    421755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    421760
   ====> MPU_IR = 89  WDP_IR = 89

WDP:Next Address verified
WDP:Execution Time verified
WDP:Subsequent byte at =    422260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    422588
WDP:No quueue operation at =    422760
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    423321
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    424588
WDP:Subsequent byte at =    425260
WDP:BUS:Instruction fetch at =    425321
WDP:Subsequent byte at =    425760
WDP:No quueue operation at =    426260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    426588
WDP:BUS:writing memory at =    427321
WDP:BUS:Passive at =    428588
WDP:DS: Data Segment accessed
WDP:BUS:Instruction fetch at =    429321
   ====> MPU_IR = F4  WDP_IR = 89

WDP:Byte Count verified
WDP:Two bytes transferred at =    429755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    429760
   ====> MPU_IR = F4  WDP_IR = F4

WDP:Next Address verified
WDP:Execution Time verified
WDP:BUS:Passive at =    430079
WDP:No quueue operation at =    430260
WDP:BUS:writing memory at =    430820
WDP:BUS:Passive at =    432088
WDP:DS: Data Segment accessed
WDP:BUS:Instruction fetch at =    432821
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    434088
MPU:The contents of CPU registers at time =    500000
MPU: AX=  0000000000010101  BX=  0000000000000011
CX=  0000000000000000  DX=  0000000000000000  SI=  0000000000000000
MPU: CS=  0000000000000000  DS=  0000000100000000
SS=  0000000100000000  ES=  0000000000000000  DI=  0000000000000000
MPU: IP=  0000000000100010  BP=  0000000100001000
SP=  0000000000011110  IR=  11110100
finish at 500000
finish simulation   cpu secs = 140.83   (total = 142.76)
*
*
*
*quit

HILO END OF RUN 14-MAR-1989 16:08
```

Appendix E
Fault Simulation Run

```
        THE INPUT WAVEFORM

   1 : waveform wave
   2 : stimulus clk, seuin = 0;
   3 : stimulus ready, resetin, chkreg = 1;
   4 : stimulus msg = 1;
   5 : stimulus faulttype[2:0], faultbit[3:0] = 0;
   6 : 0        clk = by 500 to 1m change0(250,+250);
   7 : 1400     resetin=0;
   8 : 122346   faulttype = hex 2 faultbit = hex 6 ;
   9 : 122348   seuin=1;
  10 : 499999   chkreg = 0;
  11 : 500k     finish.

*
        THE SIMULATION COMMANDS
*
sim system wave
dch ( ,,,"====>" ,,"MPU_IR = ", (mpu_ir) hex ,, "WDP_IR = ",(wdp_ir) hex ).
*
        THE SIMULATION OUTPUT
*
1 OF IN8086
1 OF IN8288
3 OF IN8282
2 OF IN8286
1 OF SN7404
1 OF DECODER
1 OF ROM0
1 OF ROM1
1 OF RAM0
1 OF RAM1
1 OF ROM0D
1 OF ROM1D
1 OF TFD
24 OF PARTLATCH
Delayscale = NS
Time to load 34.74 cpu secs.
**WARNING LATCH3_LATCH[5]_D not connected to an output
**WARNING LATCH3_LATCH[6]_D not connected to an output
**WARNING LATCH3_LATCH[7]_D not connected to an output
**WARNING INV_A[2] not connected to an output
**WARNING INV_A[3] not connected to an output
**WARNING INV_A[4] not connected to an output
**WARNING INV_A[5] not connected to an output
**WARNING INV_A[6] not connected to an output
SYSTEM LOADED OK
Initialisation complete   cpu secs = 1.93   (total = 1.93)
```

```
                MM          WW
                PP          DD
                UU          PP

                __          __
                II          II
                RR          RR
                [[          [[
                73          73
                ]]          ]]


   ====> MPU_IR = XX  WDP_IR = XX

WDP:BUS:Passive at =        80
WDP:BUS:Instruction fetch at =      6085
WDP:Q reinitialized at =      6260
WDP:No quueue operation at =      6760
WDP:EU waiting but not on empty Q
WDP:BUS:Passive at =      7088
WDP:bus cycle aborted for EU request -- 1 clock cycle lost
WDP:BUS:Instruction fetch at =      7821
   ====> MPU_IR = BD  WDP_IR = XX

WDP:First byte read from Q at =      8260
   ====> MPU_IR = BD  WDP_IR = BD

WDP:Jump taken
WDP:No quueue operation at =      8760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =      9088
WDP:Subsequent byte at =      9260
WDP:No quueue operation at =      9760
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =      9821
WDP:Subsequent byte at =     10260
   ====> MPU_IR = E8  WDP_IR = BD

WDP:Byte Count verified
WDP:First byte read from Q at =     10760
   ====> MPU_IR = E8  WDP_IR = E8
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     11088
WDP:Q reinitialized at =     11260
WDP:No quueue operation at =     11760
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =     11821
WDP:Subsequent byte at =     12260
WDP:BUS:Passive at =     12589
WDP:Subsequent byte at =     12760
WDP:No quueue operation at =     13260
WDP:EU waiting but not on empty Q
WDP:BUS:writing memory at =     13320
WDP:BUS:Passive at =     14588
WDP:SS: Stack Segment accessed
WDP:BUS:Instruction fetch at =     15321
WDP:Q reinitialized at =     15760
WDP:No quueue operation at =     16260
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     16588
WDP:BUS:Instruction fetch at =     17321
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     18588
```

```
WDP:BUS:Instruction fetch at =     19321
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     20588
   ====> MPU_IR = 33  WDP_IR = E8

WDP:Byte Count verified
WDP:Two bytes transferred at =     21255
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     21260
   ====> MPU_IR = 33  WDP_IR = 33
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     21321
   ====> MPU_IR = 31  WDP_IR = 33
   ====> MPU_IR = 33  WDP_IR = 33

WDP:Subsequent byte at =     21760
WDP:No quueue operation at =     22260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     22588
   ====> MPU_IR = 8B  WDP_IR = 33

WDP:Byte Count verified
WDP:First byte read from Q at =     23260
   ====> MPU_IR = 8B  WDP_IR = 8B
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     23321
WDP:Subsequent byte at =     23760
WDP:No quueue operation at =     24260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     24588
WDP:Subsequent byte at =     26760
WDP:BUS:Instruction fetch at =     26821
WDP:No quueue operation at =     27260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     28088
WDP:BUS:reading memory at =     28821
WDP:BUS:Passive at =     30088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 45  WDP_IR = 8B

WDP:Byte Count verified
WDP:Two bytes transferred at =     31755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     31760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     32260
WDP:Byte Count verified
WDP:First byte read from Q at =     32760
WDP:Next Address verified
WDP:Execution Time verified
WDP:BUS:Instruction fetch at =     32821
WDP:No quueue operation at =     33260
   ====> MPU_IR = 13  WDP_IR = 45

WDP:Byte Count verified
WDP:First byte read from Q at =     33760
   ====> MPU_IR = 13  WDP_IR = 13
WDP:Next Address verified
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 11  WDP_IR = 13

WDP:BUS:Passive at =     34088
WDP:Subsequent byte at =     34260
WDP:No quueue operation at =     34760
WDP:BUS:Instruction fetch at =     34821
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     36088
WDP:Subsequent byte at =     37260
WDP:No quueue operation at =     37760
WDP:BUS:reading memory at =     38821
WDP:BUS:Passive at =     40088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     42755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     42760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     42821
WDP:No quueue operation at =     43260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     43760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     44088
WDP:Subsequent byte at =     44260
WDP:No quueue operation at =     44760
WDP:Q reinitialized at =     48260
WDP:BUS:Instruction fetch at =     48321
WDP:No quueue operation at =     48760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     49588
WDP:BUS:Instruction fetch at =     50321
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     51588
WDP:Byte Count verified
WDP:First byte read from Q at =     51760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified -- jump taken
WDP:Execution Time verified
WDP:No quueue operation at =     52260
WDP:BUS:Instruction fetch at =     52321
WDP:Byte Count verified
WDP:First byte read from Q at =     52760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     53260
WDP:Bus request granted with out loss of a cycle
```

```
   ====> MPU_IR = 13  WDP_IR = 45

WDP:BUS:Passive at =     53588
WDP:Byte Count verified
WDP:First byte read from Q at =     53760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     54260
WDP:BUS:Instruction fetch at =     54321
WDP:No quueue operation at =     54760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     55588
WDP:BUS:Instruction fetch at =     56321
WDP:Subsequent byte at =     57260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     57588
WDP:No quueue operation at =     57760
WDP:BUS:reading memory at =     58821
WDP:BUS:Passive at =     60088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     62755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     62760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     62821
WDP:No quueue operation at =     63260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     63760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     64088
WDP:Subsequent byte at =     64260
WDP:No quueue operation at =     64760
WDP:Q reinitialized at =     68260
WDP:BUS:Instruction fetch at =     68321
WDP:No quueue operation at =     68760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     69588
WDP:BUS:Instruction fetch at =     70321
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     71588
WDP:Byte Count verified
WDP:First byte read from Q at =     71760
   ====> MPU_IR = 45  WDP_IR = 45
```

```
WDP:Next Address verified -- jump taken
WDP:Execution Time verified
WDP:No quueue operation at =     72260
WDP:BUS:Instruction fetch at =     72321
WDP:Byte Count verified
WDP:First byte read from Q at =     72760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     73260
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 13  WDP_IR = 45

WDP:BUS:Passive at =     73588
WDP:Byte Count verified
WDP:First byte read from Q at =     73760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     74260
WDP:BUS:Instruction fetch at =     74321
WDP:No quueue operation at =     74760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     75588
WDP:BUS:Instruction fetch at =     76321
WDP:Subsequent byte at =     77260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     77588
WDP:No quueue operation at =     77760
WDP:BUS:reading memory at =     78821
WDP:BUS:Passive at =     80088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =     82755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =     82760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =     82821
WDP:No quueue operation at =     83260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =     83760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     84088
WDP:Subsequent byte at =     84260
WDP:No quueue operation at =     84760
WDP:Q reinitialized at =     88260
WDP:BUS:Instruction fetch at =     88321
WDP:No quueue operation at =     88760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     89588
```

```
WDP:BUS:Instruction fetch at =     90321
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 45  WDP_IR = 75

WDP:BUS:Passive at =     91588
WDP:Byte Count verified
WDP:First byte read from Q at =     91760
   ====> MPU_IR = 45  WDP_IR = 45

WDP:Next Address verified -- jump taken
WDP:Execution Time verified
WDP:No quueue operation at =     92260
WDP:BUS:Instruction fetch at =     92321
WDP:Byte Count verified
WDP:First byte read from Q at =     92760
WDP:Next Address verified
WDP:Execution Time verified
WDP:No quueue operation at =     93260
WDP:Bus request granted with out loss of a cycle
   ====> MPU_IR = 13  WDP_IR = 45

WDP:BUS:Passive at =     93588
WDP:Byte Count verified
WDP:First byte read from Q at =     93760
   ====> MPU_IR = 13  WDP_IR = 13

WDP:Next Address verified
WDP:Execution Time verified
   ====> MPU_IR = 11  WDP_IR = 13

WDP:Subsequent byte at =     94260
WDP:BUS:Instruction fetch at =     94321
WDP:No quueue operation at =     94760
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     95588
WDP:BUS:Instruction fetch at =     96321
WDP:Subsequent byte at =     97260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =     97588
WDP:No quueue operation at =     97760
WDP:BUS:reading memory at =     98821
WDP:BUS:Passive at =    100088
WDP:DS: Data Segment accessed
   ====> MPU_IR = 13  WDP_IR = 13

   ====> MPU_IR = 49  WDP_IR = 13

WDP:Byte Count verified
WDP:Two bytes transferred at =    102755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    102760
   ====> MPU_IR = 49  WDP_IR = 49
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    102821
WDP:No quueue operation at =    103260
   ====> MPU_IR = 75  WDP_IR = 49

WDP:Byte Count verified
WDP:First byte read from Q at =    103760
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified
```

```
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    104088
WDP:Subsequent byte at =    104260
WDP:No quueue operation at =    104760
WDP:BUS:Instruction fetch at =    104821
   ====> MPU_IR = C3  WDP_IR = 75

WDP:BUS:Passive at =    105589
WDP:Byte Count verified
WDP:First byte read from Q at =    105760
   ====> MPU_IR = C3  WDP_IR = C3

WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified
WDP:No quueue operation at =    106260
WDP:BUS:reading memory at =    106320
WDP:Stack Address verified
WDP:BUS:Passive at =    107588
WDP:SS: Stack Segment accessed
WDP:BUS:Instruction fetch at =    108321
WDP:Q reinitialized at =    109260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    109588
WDP:No quueue operation at =    109760
WDP:EU waiting but not on empty Q
WDP:BUS:Instruction fetch at =    110321
   ====> MPU_IR = BA  WDP_IR = C3

WDP:Byte Count verified
WDP:Two bytes transferred at =    110755
WDP:Data Count verified
WDP:Data Size verified
WDP:First byte read from Q at =    110760
   ====> MPU_IR = BA  WDP_IR = BA

WDP:Execution Time verified
WDP:No quueue operation at =    111260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    111588
WDP:Subsequent byte at =    111760
WDP:No quueue operation at =    112260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    112321
WDP:Subsequent byte at =    112760
   ====> MPU_IR = 33  WDP_IR = BA

WDP:Byte Count verified
WDP:First byte read from Q at =    113260
   ====> MPU_IR = 33  WDP_IR = 33
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    113588
WDP:Q reinitialized at =    113760
WDP:No quueue operation at =    114260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    114321
   ====> MPU_IR = 31  WDP_IR = 33
   ====> MPU_IR = 33  WDP_IR = 33

WDP:Subsequent byte at =    114760
WDP:No quueue operation at =    115260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    115588
   ====> MPU_IR = D1  WDP_IR = 33
```

```
WDP:Byte Count verified
WDP:First byte read from Q at =    116260
   ====> MPU_IR = D1  WDP_IR = D1
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    116321
   ====> MPU_IR = 51  WDP_IR = D1

WDP:Subsequent byte at =    116760
   ====> MPU_IR = 73  WDP_IR = D1

WDP:Byte Count verified
WDP:First byte read from Q at =    117260
   ====> MPU_IR = 73  WDP_IR = 73
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    117588
WDP:Q reinitialized at =    117760
WDP:No quueue operation at =    118260
WDP:EU waiting on empty Q
WDP:BUS:Instruction fetch at =    118321
WDP:Subsequent byte at =    118760
WDP:No quueue operation at =    119260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    119588
   ====> MPU_IR = 43  WDP_IR = 73

WDP:Byte Count verified
WDP:First byte read from Q at =    120260
   ====> MPU_IR = 43  WDP_IR = 43
WDP:Next Address verified -- non jump opcode
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    120321
WDP:No quueue operation at =    120760
   ====> MPU_IR = 4A  WDP_IR = 43

WDP:Byte Count verified
WDP:First byte read from Q at =    121260
   ====> MPU_IR = 4A  WDP_IR = 4A
WDP:Next Address verified
WDP:Execution Time verified

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    121588
WDP:No quueue operation at =    121760
   ====> MPU_IR = 75  WDP_IR = 4A

WDP:Byte Count verified
WDP:First byte read from Q at =    122260
   ====> MPU_IR = 75  WDP_IR = 75
WDP:Next Address verified
WDP:Execution Time verified

WDP:BUS:Instruction fetch at =    122321
WDP:Subsequent byte at =    122760
WDP:No quueue operation at =    123260
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    123588
WDP:Q reinitialized at =    127260
WDP:BUS:Instruction fetch at =    127321
WDP:No quueue operation at =    127760
WDP:EU waiting but not on empty Q
WDP:Bus request granted with out loss of a cycle
```

```
WDP:BUS:Passive at =    128588
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    129271 Instruction cycles=  00000000
WDP:BUS:Instruction fetch at =    129321
MPU:Instruction changed to another
MPU: FAULT_TYPE: 00000010
MPU: FAULT_BIT: 00000110
MPU: FAULT_TIME:    130000
   ====> MPU_IR = XX  WDP_IR = 75
MPU:illegal opcode detected
MPU:restart will discard op-code

WDP:Byte Count verified
WDP:First byte read from Q at =    130260
   ====> MPU_IR = XX  WDP_IR = XX
WDP:ERROR: Incorrect next address
WDP:ERROR #7: time=    130271 Instruction cycles=  00000001
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    130281 Instruction cycles=  00000001
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    130306 Instruction cycles=  00000001

WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    130588
WDP:Q reinitialized at =    130760
WDP:No quueue operation at =    131260
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    131271 Instruction cycles=  00000001
WDP:BUS:Instruction fetch at =    131321
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:ERROR: Byte count incorrect
WDP:ERROR #2: time=    131756 Instruction cycles=  00000010
WDP:First byte read from Q at =    131760
WDP:Next Address verified
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    131781 Instruction cycles=  00000010
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    131806 Instruction cycles=  00000010
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    132588
WDP:Q reinitialized at =    132760
WDP:No quueue operation at =    133260
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    133271 Instruction cycles=  00000010
WDP:BUS:Instruction fetch at =    133321
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:ERROR: Byte count incorrect
WDP:ERROR #2: time=    133756 Instruction cycles=  00000011
WDP:First byte read from Q at =    133760
WDP:ERROR: Incorrect next address
WDP:ERROR #7: time=    133771 Instruction cycles=  00000011
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    133781 Instruction cycles=  00000011
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    133806 Instruction cycles=  00000011
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    134588
WDP:Q reinitialized at =    134760
WDP:No quueue operation at =    135260
```

```
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    135271 Instruction cycles=  00000011
WDP:BUS:Instruction fetch at =    135321
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:ERROR: Byte count incorrect
WDP:ERROR #2: time=    135756 Instruction cycles=  00000100
WDP:First byte read from Q at =    135760
WDP:ERROR: Incorrect next address
WDP:ERROR #7: time=    135771 Instruction cycles=  00000100
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    135781 Instruction cycles=  00000100
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    135806 Instruction cycles=  00000100
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    136588
WDP:Q reinitialized at =    136760
WDP:No quueue operation at =    137260
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    137271 Instruction cycles=  00000100
WDP:BUS:Instruction fetch at =    137321
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:ERROR: Byte count incorrect
WDP:ERROR #2: time=    137756 Instruction cycles=  00000101
WDP:First byte read from Q at =    137760
WDP:ERROR: Incorrect next address
WDP:ERROR #7: time=    137771 Instruction cycles=  00000101
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    137781 Instruction cycles=  00000101
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    137806 Instruction cycles=  00000101
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    138588
WDP:Q reinitialized at =    138760
WDP:No quueue operation at =    139260
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    139271 Instruction cycles=  00000101
WDP:BUS:Instruction fetch at =    139321
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:ERROR: Byte count incorrect
WDP:ERROR #2: time=    139756 Instruction cycles=  00000110
WDP:First byte read from Q at =    139760
WDP:ERROR: Incorrect next address
WDP:ERROR #7: time=    139771 Instruction cycles=  00000110
WDP:ERROR: Illegal Opcode detected
WDP:ERROR #5: time=    139781 Instruction cycles=  00000110
WDP:ERROR: Instruction execution time incorrect
WDP:ERROR #1: time=    139806 Instruction cycles=  00000110
MPU:illegal opcode detected
MPU:restart will discard op-code
WDP:Bus request granted with out loss of a cycle
WDP:BUS:Passive at =    140588
WDP:Q reinitialized at =    140760
WDP:No quueue operation at =    141260
WDP:EU waiting on empty Q
WDP:ERROR: Access to undefined memory location
WDP:ERROR #10: time=    141271 Instruction cycles=  00000110
WDP:BUS:Instruction fetch at =    141321
```

```
MPU:illegal opcode detected
MPU:restart will discard op-code
finish at 141750
finish simulation   cpu secs = 36.37   (total = 38.30)
*quit

HILO END OF RUN 14-MAR-1989 17:04
```

The vita has been removed from the scanned document