Fault tolerance in digital controllers using software techniques

Halse, Robert G.

FAULT TOLERANCE IN DIGITAL CONTROLLERS USING SOFTWARE TECHNIQUES

by

Robert G. Halse

FAULT TOLERANCE IN DIGITAL CONTROLLERS USING SOFTWARE TECHNIQUES

Robert G. Halse

ABSTRACT

Microprocessor based systems for controlling gas supplies require very high levels of reliability for safety reasons. Non-redundant systems are considered to be inadequate, and an alternative approach is necessary. In digital systems, transient faults are as much as fifty times more common than permanent faults. Therefore mechanisms which allow for recovery from transients will provide large improvements in reliability. However, to enable effective design of recovery mechanisms it is necessary to understand failure modes.

The results from practical interference tests, designed to simulate transient faults, are presented. They show that corruption to the correct flow of program execution is a common failure, and that subsequent instruction fetches can be performed from any of the memory locations. Under these conditions any value of operation code can be interpreted as an instruction, including those undeclared by the manufacturers. Four commonly used microprocessors are investigated to establish the functions of the undeclared codes, and other undeclared operations are revealed.

Analyses on the sequence of events following a random jump into the four main memory types of data, program, unused and input areas, are presented. Recovery from this type of execution can be achieved by the addition of restart codes into the areas, so that execution can transfer to a recovery routine. The effect of this mechanism on the recovery process is investigated.

Finally, some methods of testing systems, to check the levels of reliability improvement obtained by these techniques, are considered.

ACKNOWLEDGEMENTS

I would like to express my gratitude to the people and organisations that have contributed to the work presented in this thesis. In paricular, to the Science and Engineering Research Council, and the British Gas Corporation's Engineering Research Station at Killingworth, for providing financial support. To my supervisor Dr. Clive Preece for his encouragement, guidance and general advice throughout the research. To Dr. Ken Jenkins of the British Gas Corporation for providing much useful information and equipment. To Dr. Mansour Sahardi for his interest and discussion on the project. To Mandy for translation and typing work. To the electrical technicians (Jack, Trevor, Michael, Colin, Steve, Ian and Ian) for their co-operation and assistance while I have been at the university. Finally, I would like to thank the Fleetham family for allowing me to practice my building skills on their house during my spare time.

FAULT TOLERANCE IN DIGITAL CONTROLLERS USING SOFTWARE TECHNIQUES

by

Robert G. Halse

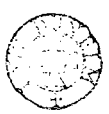
Thesis Submitted for the Degree of

Doctor of Philosophy

in the Faculty of Science

University of Durham

November 1984



List of Contents

CHAPTER 7

Flow of Execution Between Different Memory Areas

CHAPTER 8

Selection of Error Detection Mechanisms

CHAPTER 9

Development of a Facility to Test Redundant Systems

List of Figures

List of Tables

List of Symbols and Abbreviations

A/C	Alternating Current
ACM	Association of Computing Machinery
AFIPS	American Federation of Information Processing Societies
AIAA	American Institute of Aeronautics and Astronautics
CCD	Charge-Coupled Device
CE	Chip Enable
CMOS	Complementary Metal Oxide Semiconductor
D/C	Direct Current
DAG	Demand Activated Governing
DX	Op-Code Fetch from the Second Byte of a Double Byte Instruction
e	The Exponential Function
EA	External Access
EMI	Electromagnetic Interference
EMP	Electromagnetic Pulse
ENSIMAG	Ecole Nationale Superieure D'Informatique et de Mathematiques Appliquees Grenoble
EPROM	Erasable Programmable Read Only Memory
FMEA	Fault Mode Effect Analysis
FTCS	Fault Tolerant Computing Symposium
HLT	Halt Instruction
I	Number of Instruction Cycles or Transfers
i.c.	Integrated Circuit
I/O	Input/Output
IEE	Institution of Electrical Engineers
IEEE	Institute of Electrical and Electronic Engineers
in. w.g.	Inches Water Gauge
IRQ	Interrupt Request

J	Joules
JMP	Jump Instruction to a Non-Specific Location
K	Number of Instructions Executed
K	Kilo-Bytes
kV	Kilo-Volts
L	Length of Instructions in Bytes
ln	Natural Logarithm
LSI	Large Scale Integration
LSTTL	Low Power Schottky Transistor Transistor Logic
mA	Milli-Amperes
mbar	Milli-bars
mJ	Milli-Joules
ms	Milli-seconds
MHz	Mega-Hertz
mm	Millimetres
mV	Milli-Volts
MTBF	Mean Time Between Failures
N	Number of Instructions Executed
N_B	Total Number of Bytes in the Program Area
N_{BE}	Expected Number of Data Bytes Read
N_{CJ}	Number of Conditional Jump Instructions in the Instruction Set
N_{DA}	Actual Number of Data Bytes in the Memory Map
N_{DI}	Number of Double Byte Instructions in the Program Area
N_I	Total Number of Instructions in the Program Area
N_J	Number of Bytes Interpreted as Jumping Instructions
N_K	Number of Execution Sequences of K Instructions
N_{LJ}	Number of Bytes Interpreted as Jumping Instructions of Length L

N_{LNJ}	Number of Bytes Interpreted as Non-Jumping Instructions of Length L
N_{NJ}	Number of Bytes Interpreted as Non-Jumping Instructions
N_{PB}	Number of Program Bytes which Appear in the Memory Map
N_{RST}	Effective Number of Restart Instructions
N_S	Total Number of Execution Sequences
N_T	Total Number of Possible Op-Codes
N_{TB}	Total Number of Bytes in the Memory Map
N_{TI}	Number of Triple Byte Instructions in the Program Area
NASA	National Aeronautics and Space Administration
NATO	North Atlantic Treaty Organisation
NB_{AV}	Average or Expected Number of Bytes Read Before a Jump
NEC	Nippon Electric Company
NI_{AV}	Average or Expected Number of Instruction Executed Before a Jump
NI_E	Expected Total Number of Instructions Executed
NI_L	Upper Limit on the Number of Instructions Executed
NMI	Non-Maskable Interrupt
NMOS	N-Channel Metal Oxide Semiconductor
NMR	N-Modular Redundancy
NOP	No Operation
NR_{AV}	Average Number of Instructions Executed Before Resuming Valid Instruction Fetches
ns	Nano-Second
op-code	Operation Code
P_{CJ}	Probability that a Conditional Instruction will cause a Jump
P_D	Probability of Entering a Data Area
$P_{D\underline{X}}$	Probability of Entering the Operand Field of a Double Byte Instruction
P_E	A Proportion of the Total Errors

P_I	Probability of Entering an Area of Input Data
P_{II}	Probability of Entering the Input Area Twice
P_{IIL}	Probability of a Loop after Entering the Input Area Twice
P_J	Probability of Interpreting a Jump Instruction
P_{LD}	Probability of Forming a Loop in the Data Area
P_{LI}	Probability of Forming a Loop in the Input Area
P_{LU}	Probability of Forming a Loop in the Unused Area
P_{NC}	Probability that Particular Data is Not Corrupted
P_{NI}	Probability of Executing a Given Number of Instructions or More
P_{NJ}	Probability of Interpreting a Non-Jumping Instruction
P_P	Probability of Entering a Program Area
P_R	Probability of Resuming Valid Instruction Fetches
P_{RST}	Probability of Interpreting a Restart Instruction
$P_{T\underline{X}X}$	Probability of Entering the Second Byte of a Triple Byte Instruction
$P_{TX\underline{X}}$	Probability of Entering the Third Byte of a Triple Byte Instruction
P_U	Probability of Entering an Unused Memory Area
P_{UU}	Probability of Entering the Unused Area Twice
P_{UUL}	Probability of a Loop after Entering the Unused Area Twice
P_{UXi}	Probability of a Transfer from an Unused Area to Memory Area Xi
P_X	Probability of Entering an Operand Field in the Program Area
P_{Xf}	Probability of Reaching a Particular Final State
P_{XiU}	Probability of Transfering from Memory Area Xi to an Unused Area
P_{XjXi}	Probability of Transfering from Memory Area Xj to Memory Area Xi
P_{XRST}	Probability of Interpreting a Restart in an Operand Field
pF	Pico-Farads
PROM	Programmable Read Only Memory

RAM	Random Access Memory
RC	Resistor and Capacitor
RET	Return Instruction
RFI	Radio Frequency Interference
ROM	Read Only Memory
RST	Restart Instruction
SDK	System Design Kit
SEC/DED	Single Error Correction/Double Error Detection
SERC	Science and Engineering Research Council
SPC	Jump Instruction to a Specific Location
SSI	Small Scale Integration
TMR	Triple Modular Redundancy
TXX	Op-Code Fetch from the Second Byte of a Triple Byte Instruction
TXX	Op-Code Fetch from the Third Byte of a Triple Byte Instruction
uF	Micro-Farad
UPS	Uninterruptible Power Supply
us	Micro-Second
US	United States
v	Volts
Xf	Represents a Particular Final State
Xi	Represents a Particular Memory Area
Xj	Represents Each of the Four Different Memory Areas

No material contained in this thesis has previously been submitted for a degree in this or any other university.

CHAPTER 1

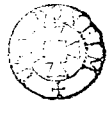
Introduction and Review of System Reliability

1.1 The Need for a Reliable Controller

With the conversion from town gas to natural gas, the supply to the consumer has changed from a large number of isolated networks to one fully integrated distribution system. This system is connected to the supplies of natural gas in the North and Irish seas, and transports it around the country in large diameter pipes (up to 1050mm), at high pressures (up to 70 bar). The pressure is reduced in stages into smaller diameter pipes until it is at a safe level to supply to the consumer.

An analogy can be drawn with the National Grid for electricity supply, where voltage corresponds to pressure, and current corresponds to flow rate. However, unlike the electricity system, gas can and must be stored within the network. This is necessary because the supply is obtained at a constant rate from the gas fields, whereas the demand by the consumer varies both throughout the day and throughout the year. Also any peak demand within a particular area must be supplied locally due to the time delay in transporting gas through the system. Therefore the network presents a complex arrangement requiring sophisticated control.

At the high pressure end of the system large volumes of gas are being handled, and small increases in efficiency result in significant financial savings. Also failure at this level is likely to affect a large number of consumers, and this justifies high expenditure on control and safety equipment. As the pressures reduce, the quantities and hence the value of gas being handled becomes less, and high expenditure on control equipment is not justified. A low cost controller is therefore required and this is the aim of the current research.



At the low pressure end of the system, accurate pressure control is required for several reasons. Much of this part of the network is constructed from short sections of cast iron pipes laid many years ago, and it is estimated that there are almost 60 million joints. Small leaks can occur but rarely create a safety problem. However, taken collectively they represent a substantial loss of revenue. Therefore methods which reduce this leakage can have both financial and safety benefits. An extensive programme to replace old sections of the network has been in progress for several years and has cost hundreds of millions of pounds. Meanwhile a reduction of pressure within the system provides significant benefits by reducing leakage, and also repair and maintenance costs. It can also postpone eventual reinforcement of the network necessary to cater for increased demand, thus saving revenue equivalent to borrowed capital interest.

Obviously the pressure cannot be reduced below a certain level or the gas would not reach the consumer. This would lead to the possibility of air entering the pipework producing a potentially hazardous condition. Low pressures can also affect the efficiency of some appliances. For these reasons a statutory minimum pressure has been set. This is 5 in. w.g. (inches water gauge) which is equivalent to the height of a column of water that the pressure can support, and is approximately 12.3 mbar. Clearly the aim is to supply the consumer with the minimum acceptable pressure throughout the daily load cycle.

1.1.1 Present Mechanical Control

Traditionally, an entirely mechanical approach has been adopted for the control of the low pressure distribution system. The use of diaphragm operated gas regulators, as shown in figure 1.1, is widespread, with

approximately 17,000 installed throughout the country. They aim to reduce the pressure to a steady value independent of the flow rate. This is achieved by feeding back the down-stream pressure into a chamber under the diaphragm. The force on the diaphragm is balanced by a spring or a series of weights. Any imbalance causes the valve to open or close and has the effect of increasing or reducing the down-stream pressure. By adjusting the loading, different output pressures can be maintained. However, this arrangement does not give perfect pressure control. The pressure tends to fall as the flow rate increases, and this is known as the 'droop' characteristic of the regulator.

So far only the pressure at the outlet of the regulator has been considered. However, the consumer may be over a mile away from the outlet and therefore, by simple fluid mechanics theory, a pressure drop will exist along the pipework and will be proportional to the square of the flow rate. Consequently, with the simple regulator described above, it is necessary to set the output pressure at a higher level to guarantee that the consumer will be supplied with at least the minimum statutory pressure, at times of peak demand. Clearly this will result in a pressure well above the statutory minimum at other times. This is refered to as the 'over pressure' of the system, and has a maximum value of the sum of the regulator droop, the pipework losses and a safety margin, as the flow rate reduces. The safety margin is included to allow back-up equipment to intervene if an abnormally low pressure is detected. Obviously the aim is to reduce the 'over pressure' to a minimum.

The above example considers only one regulator and one consumer, in reality the situation is in fact far more complicated. Low pressure networks can be fed by more than one regulator and supply several thousands

of consumers. Due to varying demands from the system, the low pressure point may not always be at the same physical location. This makes effective control even more difficult.

In the past, methods have been devised to provide automatic changes in the set point of a regulator to try and follow the pattern of demand. However, these do not operate directly on the district pressure, but on other parameters which are associated with demand, such as the time of day or the ambient temperature. Both of these parameters are not strongly linked with demand but control based on them has provided some savings. A third approach has used the flow through the regulator to adjust the set point, and has proved most successful. This is commonly know as demand activated governing (DAG).

Spearman (98) has reported a number of DAG schemes which have all shown significant savings in repair and maintenance costs. They have provided DAG by mechanical means but have several disadvantages. At least three additional valves and a substantial amount of extra pipework is required. A complex setting up and commissioning procedure is necessary to ensure optimum performance, and this has to be repeated periodically to allow for changes in the network or demand. Therefore there is scope for further improvements.

1.1.2 Future Micro-Electronic Control

To overcome the problems with mechanically implemented DAG mentioned above, and to allow for other developments, it has been proposed that micro-electronic techniques could be applied to the control of gas pressure. A simple arrangement for such a system is shown in figure 1.2. It contains a microprocessor which reads the remote low pressure point from a transducer, and activates the valve to maintain a steady supply. To

ensure overall system safety and availability, all three parts must be both reliable and must be fail-safe.

The valve could be operated by a simple solenoid providing only open and closed positions. The pressure would then be controlled by pulse width modulation on the supply to the solenoid. Although this is a simple solution, it tends to be very unreliable due to the large number of operations needed to maintain a steady pressure. Another disadvantage is that failures in either of the normal operating positions produce dangerous conditions.

A better solution would be to use a motorised valve. This will be more reliable as actuations are only required when the pressure changes, resulting in less mechanical wear. However, the response under fault conditions will be poor.

The arrangement which has been used in initial trials with digital control utilises an indirect approach. The main pressure reduction regulator is retained in the traditional configuration, but with the set point controlled by the microprocessor. This provides a much better solution as failure of the microprocessor system causes control to revert to mechanical pressure regulation.

To adjust the set point a method of increasing and decreasing the spring loading within the regulator is required. Two prototype arrangements have been built. The first uses a stepper motor to adjust the length of the spring and hence the loading. The second uses two solenoid valves to feed up-stream or down-stream pressure under a second diaphragm which acts on the spring to adjust the loading. The solenoid system is preferred as it can be arranged to 'fail safe' on power failure, by setting the regulator to its maximum set point. Burrow (20) states that generally

the 'fail safe' approach has been neglected. It is much cheaper to implement than 'fail operational' designs, and is clearly acceptable in this application as failure will only result in a reversion to a high pressure setting within the network. The 'fail safe' approach still ensures that no area drops below the statutory minimum pressure. The stepper motor, however, will stay at its current position during a power failure. As mentioned above, this reverts to mechanical control, but, if demand increases, the low pressure point will fall below the statutory minimum.

Initial trials have been carried out with both arrangements. As mentioned previously, the ideal solution is to monitor the remote low pressure point and relay the information back to the controller, and this requires some sort of telemetry link. The use of hard-wired links is expensive, and therefore other methods of transmitting the data are being investigated. However, a system operating in the United States, described by Reese (84), uses telemetry and has shown that the cost of the equipment can be recovered within the first year, due to the reduction in lost gas alone.

These initial trials have shown that micro-electronic control of the gas network is both feasible and economic. Another area to which it could be applied is the control of storage facilities. As indicated previously, it is necessary to store gas within the network and a number of arrangements have been developed such as gas holders, liquefaction plants and underground caverns. Recent interest has been directed towards the use of the medium pressure part of the network as a means of storage, and can be achieved by increasing the pressure and thus compressing the gas. This is known as 'line-pack' and is possible in this particular part of the

network because the pipework is relatively new and does not suffer from leakage.

Due to the very stringent safety requirements, it was felt that further work should be carried out to investigate methods of increasing the reliability of these control systems. British Gas has had long term experience with mechanical regulators, and, as a result, has in-depth knowledge and expertise on their operation. This has led to the development of very reliable equipment. With regard to the control of the low pressure network, micro-electronics has only recently been used by the Corporation. Therefore, this work is aimed at investigating methods of increasing the reliability of the micro-electronic parts of the systems.

1.2 Source of Failures

All equipment can fail, and usually does so in a variety of different ways. In a complex electronic system the cause of failure can be due to design errors, component failures or to environmental effects. In micro-processor based systems, design errors can occur in both the hardware and software, and can be introduced at the specification, implementation or construction phases of a project. Shooman (92) gives an example of a data acquisition system where, over a nine month period, nearly half the failures were due to software errors. At the specification stage errors can be made due to an insufficient knowledge of the system to be controlled, or by an incomplete description of the required response under all operating conditions. The importance of these errors is emphasised by Soi and Gopal (97) who suggest that nearly 60% occur at this stage in the software. At the implementation stage, the choice of the wrong type of components in hardware, or the wrong algorithm in software, can lead to failure. Finally, errors can be made during the construction of hardware

or the coding of software.

Components can fail due to a number of different failure mechanisms, and generally they follow the familiar 'bath-tub' curve. It shows a high failure rate at the beginning of their life due to manufacturing defects. This is followed by a period of constant failure rate, due to random effects, which is normally considered to be the useful life of the component. After this period the failure rate increases again due to wear out. A description of the types of failures observed in electronic components is given by Doyle (31), and a study of microprocessor devices is presented by Hnatek (47) who describes a number of physical failure mechanisms and how they can be detected.

The correct operation of electrical and electronic systems can be disturbed by environmental conditions. In analogue devices it can result in noisy signals, but in digital equipment severe disruption of the processing sequence can occur. Sources of disruption include radio frequency interference (RFI), electromagnetic interference (EMI), radiation effects, static discharges and power supply variations.

Whallen et al (113) have shown that RFI can disrupt digital circuits by changing their state. Sources of EMI in high voltage substations are listed by Pellegrini et al (79), and most are due to various forms of switching. The effects of lightning are also considered. May and Woods (64) highlight the problem of alpha particle interaction originating from packaging material. This has become a problem with the development of higher density chips, and affects most devices. General radiation effects on semiconductors have been investigated by Sexton et al (90), and they have shown that device parameters drift with dosage.

Faults can be either permanent or temporary. Permanent faults occur

as a result of catastrophic failure of a component or subsystem, and also from inherent design errors. Some sources of temporary faults are described by Ng and Avizienis (70) and include component drifts around the limits of their specifications, and environmental factors. However, the errors produced by permanent faults may appear temporary. For example, a single node stuck at zero can only produce an error when it should be set at one, and this is illustrated by Gunther and Carter (39). Also a part of the circuit which is infrequently used may not cause any errors until it is exercised. Goldberg (37) indicates that some design faults, such as timing problems, can appear to be induced environmentally, and may be difficult to distinguish. For these reasons faults can remain undetected for a considerable length of time.

McConnel et al (61) draw a distinction between intermittent errors and transient errors. Intermittents occur as a result of an underlying permanent fault and will periodically reappear, whereas a particular transient will occur only once. Ball and Hardie (5) indicate from practical experience that 90% of field failures are intermittent and are particularly difficult to isolate.

Most reliability work in the past has considered only stuck at faults. More recently bridging faults have been considered where electrical contact is made between adjacent tracks, and these are described by Kodandapani and Pradham (52). Toschi and Watanbe (103) state that soft fails in memories can also be due to data patterns, timing and read/write sequencing. All these produce intermittent errors and are particularly difficult to identify.

1.3 Methods of Increasing Reliability

There are two complimentary approaches available to increase

reliability and these are described by Avizienis (4). The first attempts to eliminate all sources of failure and is known as the fault intolerance approach. The second recognises that failures will occur, and attempts to mask their effects by the use of redundancy, this is known as fault tolerance. To achieve very high reliability a combination of both these approaches is necessary, and can be applied to each of the three sources of failure described above.

1.3.1 Reducing Failures due to Design Errors

Errors in hardware design have been reduced to a very low level by the implementation of rigorous procedures at all stages. Complex computer programs are used to analyse and simulate the hardware to check for a number of faults. Hazard and race conditions in logic circuits can be detected, interconnections can be checked for the correct routing, and loading on each node can be analysed to ensure, for example, that maximum fan-out is not exceeded. Once the hardware is constructed, thorough testing is carried out to verify correct operation.

Fault free design is more easily achieved due to recently developed integrated circuits which have themselves been designed for simple interconnection. This reduces the amount of work necessary by the system designer, but increases the effort required by the chip designer. Design errors within large scale integrated circuits (LSI) are more likely to occur due to the increased complexity of these devices. This problem is highlighted by Sequin (89).

An advantage with microprocessor based hardware is that the basic circuit can be used for many applications. This reduces the possibility of introducing errors into new projects. Software, however, has been treated in a different manner in the past, and remains a serious source of failure.

This is due mainly to the unlimited way in which software can be arranged and that, in almost all cases, new code is written for each application.

Recently much more emphasis has been placed on software reliability. This is due to the increased proportional cost of the software within systems, which results from increased complexity and reduced hardware costs. Greenspan and McGowan (38) state that 70% of US Air Force computing expenditure was for software in 1972, and this is expected to rise to 90% by 1985. Both fault intolerant and fault tolerant approaches have been investigated to alleviate this problem. The advantages of structured programming are widely recognised, making programs easier to read and understand, and thus simplifying the process of identifying errors. It tends to force the programmer to divide the problem into a series of modules. Nelson (66) reports on analyses which have shown that the error rate increases with the routine size. This is because smaller modules are far more easy to understand and test, and therefore methods which enforce the use of smaller modules will increase reliability.

As well as the language itself, the environment under which programs are developed is also important in enabling efficient testing and isolation of errors. For these reasons the United States Department of Defense has sponsored an extensive project to design a new language (Ada), and its associated development environment. Programming in Ada is more difficult than other languages due to tight restrictions on syntax and variable types. But it facilitates the early detection of errors at both compile and run time, reducing the overall development time. It also makes the code easier to understand and modify. This is particularly important as Dunn and Ullman (32) have shown; in badly written packages more errors can be introduced than are removed at the debugging stage, making the whole

system less reliable.

The fault tolerant approach recognises that bugs will remain in the software, and two methods of counteracting their effects have been proposed. Randell (83) suggests the use of recovery blocks. In this method an acceptance test is executed after each program module, and, if the results fail the test, an alternate algorithm is used. This process can be repeated until an acceptable set of results is obtained, or until all the alternate algorithms have been tried. In the latter case a different form of recovery must then be used. A practical example of recovery blocks in action is given by Anderson and Kerr (1).

The other approach is called N-version programming as described by Chudleigh (23). In this case all versions of a particular program module are executed and a majority vote is taken on all the results.

Both methods have their own advantages and disadvantages. For example, the recovery block procedure operates much faster in the absence of errors, but, to enable accurate error detection, complex acceptance tests are sometimes necessary. These can themselves be a source of error, as can the voting software in N-version programming. However, in the latter case the critical software is much smaller and will be less susceptible to errors. A comparison between the two techniques is given by Wei (109). He concludes that N-version programming is better than the use of recovery blocks because of problems with acceptance tests.

1.3.2 Reducing Failures due to Component Malfunctions

Unlike design errors which can, theoretically, be eliminated from a system, component failures are always possible. In the past both fault tolerant and fault intolerant approaches have been favoured at different times. In the early days of digital computers thousands of valves were

used in each machine, and reliability was poor due to the high failure rates of the components. Redundancy was necessary to improve performance. With the advent of the transistor and the subsequent development of the integrated circuit, less emphasis has been placed on redundancy due to the vast increase in the reliability of the components. Examples of fault tolerance in early computers is given by Carter and Bouricius (21).

In more recent years some computer applications have required even higher levels of reliability. These include cases where human life is involved or large financial losses are incurred on failure, such as manned and unmanned space flight, hospital life support equipment and aircraft control. Attempts have been made to increase still further the reliability of components used in these applications. Significant improvements can be achieved by screening out weak devices, and Pappu et al (75) describe methods of detecting them. Burn-in is a popular technique whereby equipment is operated at elevated temperatures before actual use. Even with these improvements it has again been necessary to use the fault tolerant approach.

A popular arrangement has been the use of triple modular redundancy (TMR) which was first proposed by Von Neumann in 1956 (105). TMR consists of three identical modules, each performing the same function, which are connected to a majority voting circuit. If one module fails, the voting circuit masks any errors by outputting the values from the other two. Clearly this requires at least three times as much hardware as a simplex system.

In many cases the extra cost could not be justified, and, in these cases, dual systems have been used. They can be configured in a number of ways. In a cold standby arrangement, a second module is maintained in an

inactive state and requires initialisation before use. Lonn et al (58) describe a hot standby system where the second module continually monitors the process, ready to take immediate control. In many control applications the switch over to the standby system is performed manually after the activation of an alarm. In a duplex arrangement both modules perform identical operations and comparisons are made between their outputs. This provides simple error detection but does not readily indicate which module is in error.

As the cost of hardware has fallen and the requirements of reliability have increased, more complex arrangements have been developed. N-modular redundancy (NMR), where N represents the number of modules, has been proposed in cases where the reliability of TMR is considered insufficient. Examples using four channels have been constructed for the F-8 fighter aircraft described by Bumby (19), and for NASA's space shuttle described by Gelderloos and Wilson (36). In both cases the requirement is for safe operation in the presence of two failures.

An important property of these systems is that reliability is drastically reduced after each failure. For example, TMR is at least twice as unreliable as a simplex system after a single failure, and therefore it is important to repair failed modules quickly. In closed systems, such as unmanned spacecraft, manual repair is not possible. NMR can be used to survive several failures by increasing the number of modules. Alternatively a number of standby spares can be provided so that the system can reconfigure itself in order to substitute a failed component or subsystem for a good one. This effectively provides automatic repair.

Wensley (110) proposes the use of a number of loosely connected units, with the fault tolerance implemented by software. In this way critical

tasks can be executed on several units with voting carried out in the program. This arrangement allows dynamic reconfiguration to eliminate faulty units after they have been identified. This sort of arrangement is commonly used in telephone switching equipment, and has also been proposed for aircraft applications by Hamill and Phillips (40).

Hybrid systems utilising a combination of the above architectures to exploit their individual advantages are becoming more popular. For example, Hopkins (48) describes a processing concept for space vehicles which uses duplex, TMR and standby sparing.

1.3.3 Reducing Failures due to Environmental Effects

Significant improvements in reliability can be obtained by reducing the effects of environmental phenomena. The fault intolerant approach does this by providing a stable local environment for the equipment. Basu (9) and Williamson (115) give comprehensive details of possible steps for reducing the effects of noise, and indicate that the design of the system enclosure is of great importance. In the United States the level of EMI emitted from digital equipment is restricted. To meet these requirements good shielding is necessary, which not only reduces emissions, but also reduces the susceptibility of the equipment from external EMI.

Boothman (14) describes methods of designing cabinets for optimum shielding, and suggests the use of metals, metalised coatings on plastics or conductive plastics. Ideally a continuous unbroken metal enclosure forming a Faraday cage is preferred in order to eliminate most electrical interference. However, all systems need to communicate with the outside world and most require an external power supply. Therefore apertures in the enclosure are inevitable and Boothman shows the importance of both their size and location relative to the internal components. Rostek (86)

suggests a 'Rule of Thumb' of restricting maximum openings to 25mm for each nanosecond rise time of the digital circuits.

He also emphasises the importance of conducted interference on power supply and signal lines and suggests the use of comprehensive filtering. Routing of power and signal cables and the quality of their sheilding is also important, and is discussed by Dick (30). With the development of fibre optics, data transmission can be made far more secure. Dyer (33) recommends their use, especially in military equipment, for immunity of both EMI and the more damaging EMP generated by nuclear explosions.

As well as EMI superimposed on the power supply, brown-outs and black-outs can occur, where the voltage is reduced or lost completely over a period of time. In these cases filtering alone is not sufficient. These problems occur frequently and have led to the development of uninterruptible power supplies (UPS). A number of arrangements have been developed for large installations, with the standby power source provided by batteries or diesel generators. These are described by Sulway (99), and in these cases an A/C supply is maintained.

In smaller systems batteries alone can directly provide the necessary D/C levels. This has led to the development of higher capacity minature batteries, such as the zinc/air type described by Pytches (82). Rechargeable batteries can be trickle charged when the external source is available, ensuring that they are in good condition when required. This sort of arrangement has been used in the NATO III communication satellites described by McKinney and Briggs (62). Solar cells provide the external source to charge several sets of batteries, and are required for peak demand and to ensure continuous operation during solar eclipses.

Other environmental factors such as mechanical shock and vibration

must also be taken into account, and can usually be suppressed by suitable damping. Thermal effects are also important. It is widely recognised that high component temperature leads to an increased failure rate. It can also cause a general drift in component properties, therefore, methods which restrict temperature such as cooling fins or convective fans will produce benefits. However, the use of fans drawing air in from outside the cabinet can have detrimental effects, since openings are required to allow for the passage of air, and these introduce the possibility of increasing the susceptibility to EMI as described above. It also allows moisture, solid particles and corrosive substances to enter the enclosure. Filtering can be used to reduce the possibility of contamination, but in certain cases a totally sealed unit is preferred.

Shielding is effective in many cases but does not prevent all external interaction. Ziegler and Lanford (117) have shown that even half a metre of concrete has little effect on reducing interference to charge-coupled devices (CCD) from certain types of cosmic rays. However, they do suggest that the orientation of the devices can be used to reduce the problem. Shielding is also ineffective against internally generated interference and alpha particle interaction originating from package material.

In these cases the components themselves can be designed to be less susceptible to certain disturbances. Brodsky (16) suggests methods of improving RAM's against alpha particle attack, and Kim et al (51) describe methods of hardening devices against general forms of radiation. Certain device technologies are inherently less susceptible to radiation than others. Barton et al (8) show that bipolar devices are superior to complementary metal oxide semiconductors (CMOS), which in turn are better than N-channel devices (NMOS).

As described above, fault intolerance can be used to reduce the influence of environmental phenomena. In general, great improvements can be obtained for a small cost if careful consideration is taken at the design stage. Additional improvements can be made but usually involve ever increasing costs. In such cases the fault tolerant approach is worth while.

Environmental disturbances can either cause permanent or temporary damage to systems. With the precautions taken above, damage will be reduced and transient effects will predominate. The redundancy techniques mentioned in the previous section will be effective provided that simultaneous faults in different channels do not occur. Much work has been aimed at developing techniques to detect and correct errors in memory systems. Most techniques rely on error detection and correction codes, such as those proposed by Hamming (41). Extra bits of information are added to each of the data words and these can indicate which particular bit is in error if a fault occurs. Levine and Meyers (57) indicate the number of check bits required for single error correction and double error detection (SEC/DED). However, if more than two bits fail they may not be detected or an erroneous correction may be made. In these cases Walker et al (108) describe a memory system which is capable of masking off failed bits to survive multiple faults.

Time redundancy is a useful means of counteracting transient faults. This method uses the re-execution of a program segment at a later time, in the anticipation that transient disturbances will have subsided. A number of different stategies can be adopted. For example, a particular segment could be executed repeatedly until two or three consecutive results are the same. Alternatively the segment could be executed a fixed number of times

and a majority vote taken. This is similar to N-version programming with all versions identical.

Rollback techniques are another effective defence against transient faults. In this case the program periodically saves information about its current state. This is known as a checkpoint. When an error is detected, execution can then be restarted at one of these points. O'Brien (72) studies several checkpointing stategies, and he recognises that in control applications the speed of recovery is usually critical, requiring the frequent insertion of rollback points. He shows that a large overhead is necessary with regard to both execution time and memory space. To limit overheads, a checkpoint should be saved when the critical data is at a minimum, and this will normally occur at the end of a calculation. It is recommended that a checkpoint should be saved at least once during each control loop.

A disadvantage of this technique is that added complexity in the software is both costly and prone to error. Barigazzi and Strigini (6) suggest that the setting of recovery points should be transparent to the programmer to overcome these problems. This has been implemented on the Cm* computer by Siewiorek (93). Checkpointing and rollback are similar to the use of recovery blocks, where instead of using alternate algorithms the same segment is repeated until an acceptible set of results is obtained. Lee et al (56) have proposed a method of reducing the programming requirement in the use of recovery blocks by using a recovery cache. This automatically saves critical data as the program executes and could be used in simple rollback recovery.

1.4 Reliability Improvements Obtained

To determine the improvements obtained by adding one of the features

described above, it is necessary to determine the failure rate of the system with and without the modification. Modern microprocessor based systems have high reliability with a mean time between failures (MTBF) of several thousand hours. Therefore practical testing under normal operating conditions is both time consuming and costly. For individual component failure rates a number of data bases exist, such as MIL-HDBK-217D (121) compiled by the US Military, and HRD3 (122) compiled by British Telecom. HRD3 is based mainly on field data, whereas MIL-217D is based both on field data and accelerated life testing. A comparison between various failure rate data bases is given by Siewiorek et al (94).

Accelerated life tests have become very popular. They aim to speed up the failure process by subjecting the device to a more severe environment than normal, such as increased humidity, vibration or temperature. However, great care must be taken with the results. Siewiorek et al (94) show how the Arrhenius equation can be used to translate accelerated test data to ambient conditions and indicate that a factor of 62 difference in predicted failure rate can be obtained by the choice of activation energy. Another problem with accelerated tests is that if the conditions are varied too much, then failures due to other mechanisms can occur which will not be present under normal conditions, and this is illustrated by Hart et al (42).

However, these tests do provide useful results if care is taken, but, unfortunately, are normally carried out only at the component level. Full system testing can be achieved but requires bulky equipment and is time consuming. For these reasons a great deal of research has been aimed at modelling systems and predicting overall failure rates from the components. To assist in the calculations several computer programs have been written,

such as ARIES, described by Ng and Avizienis (71), and PREDICTION, described by Bell et al (10).

Improvements obtained by techniques to counteract software design errors are difficult to quantify. They are dependent on the knowledge of the failure rate before and after implementation, and this information is not readily available. Musa (65) states that assembly language programs have an average of between 3-8 errors per 1000 lines before testing. He proposes that the number remaining in a system is proportional to the time between error detection during testing, and suggests that this can be used to predict the failure rate of the final version. Hecht (44) proposes a model for the reliability of software systems using recovery blocks, and evaluates their effectiveness by trying 'what if' numbers in the model. He concludes that for a given level of reliability, the goal can be reached more cheaply by using the fault tolerant approach.

An alternative approach to determine improvements is to simulate the hardware on another computer. A variety of faults can then be injected into the simulator and the response of the system observed. This method was adopted for the Saturn V launch vehicle digital computer, and is described by Ball and Hardie (5).

1.5 Importance of Error Detection

From the types of investigations mentioned above, a large number of predictions have been made for the improvements obtained by each of the redundancy techniques. In many cases a large variation in the results exist, and this is due mainly to the assuptions made about failure mechanisms and recovery response. For most arrangements, error detection and fault location is of prime importance, for both recovery and maintenance. Triplex systems provide simple identification of single

failed units, whereas with duplex systems fault location is more difficult. Significant benefits can be obtained with the addition of fault detection mechanisms especially in systems relying on software implemented recovery.

A number of techniques have been developed and generally they fall into two main catagories of continuous monitoring and periodic checking. Continuous monitoring can be provided by self checking circuits or arithmetic codes. Self checking circuits are designed to fail in a secure manner, and are described by Williamson (114). One approach is to duplicate all signals using complementary logic, and this is described by Sedmak and Liebergot (88). In this way all single point faults and most multiple faults are easily detected. Arithmetic codes are discussed by Avizienis (3), they are an extension of error correcting codes in memories, but their properties are maintained during arithmetic and some logical operations. They can therefore be used to detect errors in memory, on the bus and in the processor, but require special processing units.

Periodic checks can be initiated by software to exercise all elements of the system, in order to test for correct operation. Barraclough et al (7) state that it is impossible to test for all faults, and therefore partial testing of each functional block is recommended. This approach is adopted in an aircraft application, using duplex redundancy, described by Johnson and Shaw (50), and is used in conjunction with other techniques such as rollback and reconfiguration.

Processor testability is discussed by Robach et al (85), who suggest that a systematic approach should be adopted where blocks are tested by elements which have already been verified. Clearly, some blocks must be assumed fault free initially, and the aim is to reduce this hard core to a minimum. Smith (96) investigates four different methods of testing

processors and concludes that the systematic approach is the best. Example programs for functional testing of the 8080 are given by Peckett (78) and Nichols (69). The 6805 has an in-built test program which is described by Boney (13). Unfortunately it requires a specific external configuration and therefore cannot be used as a built in test feature.

Random access memory (RAM) tests have been studied extensively. It is recognised that exhaustive testing for all possible pattern sensitive faults is not realistic. This has led to the development of a number of selective tests which are designed to reveal certain expected faults. Thatte and Abraham (102) describe a number of failure mechanisms and the tests necessary to detect them. Read only memory (ROM) can be tested by the evaluation of a checksum, and this method is explained by Jack et al (49). Input and output lines can be cross connected for testing, or in a closed loop control situation, the response to a small disturbance by the controller can be monitored to reveal faults in all the interfacing circuits. This latter procedure is suggested by Kurzhals and Deloach (55) for an aircraft application.

These checking routines can be executed in a background mode similar to that proposed by Preece and Stewart (80). In all cases the aim is to detect errors quickly so that they cannot propogate and prevent recovery. Using these methods it is possible to detect some faults before they have disrupted program execution, and is due to the error latency of digital circuits. This is the time taken for a fault to generate an error on the output of the device. Shedletsky and McClusky (91) show that even in a simple four state sequential circuit the error latency can be several tens of cycles, and will be far more in complex circuits.

These types of self checking procedures are particularly useful in

duplex arrangements and those using stand-by spares, to locate failed units during operation. Another important use is in applications having short mission times. For these, the importance of a fault free system prior to use is illustrated by Tasar (100) in connection with aircraft control. He suggests that 90% of faults can be detected in this way, with only a basic knowledge of the hardware.

Error detection is an important aspect of fault tolerance, but without error correction Kopetz (53) has shown that availability is reduced.

1.6 Possible Dangers of Adding Redundancy

Careful consideration must be taken when adding redundancy to a system, as increased complexity can lead to design errors. Even correct designs can be less reliable than non-redundant systems. For example, if a single voting arrangement is adopted in a TMR system, then the voters must be more reliable than a single channel to achieve an overall improvement, and this is shown by Wakerly (107). In equipment containing standby spares, Losq (59) has shown that a system with a large number of spares is less reliable than the corresponding simplex arrangement, due to the complexity of the switch. Elkland and Siewiorek (34) show that memory error detection and correction systems can also be less reliable, due to failure of the additional memory and correction circuits.

Another important factor is the concept of coverage which was first introduced by Bouricius et al (15). It is the probability that a system will recover from a fault without any loss of essential information. Clearly the aim is for a high level of coverage, and Arnold (2) has shown that even a small percentage of uncovered faults has a severe effect on the reliability of redundant systems. These faults are called common mode failures, and can be the major source of system unreliability. Westermeier

(112) shows that adding redundancy with low coverage actually reduces overall reliability.

Most of the techniques mentioned so far are designed to counteract particular classes of faults. If these fault types are not common in the final system then the methods will be ineffective and may even reduce overall reliability. For example, most fault tolerant memory systems are designed to detect and correct single bit failures, where multiple bit failures may be more common due to simultaneous disturbances in several chips. Exhaustive memory tests to detect faults are not possible due to restrictions of time. For this reason tests have been developed for certain types of fault such as interactions between adjacent cells. However, Heftman (45) describes modern devices with extra rows of cells which can be substituted for faulty ones. This severely reduces the effectiveness of the tests.

Wulf (116) states that increasing the reliability of individual components has little effect on the mission time, but increasing the coverage of the most probable fault produces significant improvements. It is therefore of great importance to know what type of failures will occur in the real system, so that only methods suitable to counteract those particular faults are adopted.

Previous sections have indicated that a particular error detection and correction mechanism is not effective against all faults. It is therefore necessary to use a number of techniques. Pearson et al (77) describe a hierarchical approach with different levels of fault recovery. At a low level modular redundancy and memory protection are transparent to the program, and are independent of the application. At higher levels the mechanisms become more application dependent, with the use of software

techniques. The highest level must cover all other undetected faults, and is usually provided by a watchdog timer. This device is periodically updated by the control program, and is normally configured to generate a master reset if it fails to receive correct signals. The benefits obtained by watchdogs are recognised, but the following chapters show that careful consideration for their design is necessary.

1.7 Requirements for Different Applications

Different applications have varying operating requirements, and in each case a particular technique is sometimes necessary. A number of applications and their specifications are given in table 1.1. For systems such as aircraft control very short program loops are necessary to maintain stability. Therefore detection and correction of errors must occur very rapidly, and requires the use of TMR or NMR. This usually prevents any interruption of program execution.

In telephone switching systems, short interruptions are permissible, but repair must be quick and effective. Emphasis is placed more on the detection and isolation of faulty modules, and this is achieved by a large number of processing units each adopting a duplex arrangement. In this way faults within a particular unit are easily identified, while recovery is performed by reallocation of processing tasks.

In many industrial control situations, such as coal fired power stations described by Bland et al (11), it is only necessary to detect an error and to switch safely to mechanical or manual control. In these cases, processing power can be lost for several seconds or minutes without severe damage to the plant.

For the British Gas application of micro-electronic implementation of DAG, the latter case is acceptable for most networks. This is because only

the set point of the regulator requires adjustment with local mechanical control of the pressure. If the set point is changed too often then instability between the two control mechanisms can occur. Spearman (98) suggests a time interval between adjustments in the range of 5 to 120 seconds. Therefore a loss in processing time of a similar duration will not be detrimental, provided that the regulator is not driven to its lowest setting during failure. However, in networks containing large industrial loads, rapid changes in demand can occur and this requires a faster response with a correspondingly shorter control loop.

An architecture for a small digital controller suitable for this application has been proposed by Pearson (76). It consists of a triplicated processor arrangement with voting, connected to a single block of RAM which contains single bit error correction and double bit error detection. The control program is stored in two different EPROM's so that if one fails the other one can be used. This architecture does have a high coverage for a number of fault conditions, but is susceptible to several possible common mode failures, which can remain undetected by the hardware. In these cases detection and correction methods within the software are required.

An alternative approach, which could be used, is described by Obac-Roda and Davies (73). They suggest using three independent microprocessor systems connected in a ring structure. Each system operates in loose sychronism with the other two, and voting on results is achieved in software. A similar arrangement could be used but with all the channels working in complete isolation. The outputs could then be brought together at the actuators, and even these could be isolated by using seperate ones for each channel. With such isolation, Dellacorna et al (29) have

indicated that it would not be necessary to use the same processor in each unit, and therefore each one could be designed and programmed by a different development group to eliminate the possibility of nearly all common mode failures.

By reducing the interaction between modules, a great deal of physical and electrical isolation can be achieved, especially with the use of fibre optics. Emfinger and Flannigan (35) describe how physical isolation is used to improve the survivability of a fighter aircraft from attack. The use of these methods in the British Gas application could reduce the risks from rare events such a direct lightning strikes and vehicle impacts, which have occurred in the past.

1.8 Contents of the Thesis

The aim of the work described in this thesis is to investigate methods of increasing system reliability with particular attention given to software techniques. It has been indicated in the foregoing discussion that both transient and intermittent failures are common, and therefore improvements in this area are most likely to give significant benefits. To prevent failure, both error detection and error correction must be effective, and detection mechanisms receive particular attention.

It has been shown that the actual failure mechanisms are important in the development of redundancy techniques. Most researchers have adopted a policy of considering only single point failures. This is a legacy from early reliability studies on systems containing descrete components and small scale integration (SSI). With the development of large scale integration (LSI), it is an increasingly more complex process to analyse systems at the transistor and gate levels. Also, faults are less likely to be limited to single nodes due to their physical size and very close

proximity to each other. Despite this there is very little information available about failure mechanisms observed at the subsystem level.

Chapter 2 contains a description of a number of practical test which were carried out on a small microprocessor based system. These were primarily concerned with electrical interference on the power supply rails. Errors observed at the chip level are presented. The tests revealed that a number of mechanisms exist which cause the corruption of the program counter, resulting in the possible resumption of execution at any location in the memory map. This demonstrates the importance of the undeclared operation codes in microprocessors which may be read under these conditions. Chapter 3 investigates the undeclared codes of several processors and reveals other undeclared properties.

Chapters 4, 5 and 6 look at the response of different processors to erroneous execution in specific parts of the memory maps. Analysis is performed by a series of mathematical models derived from Markov diagrams. In some cases they have been verified by computer simulations. Chapter 7 studies the flow of erroneous execution between different memory areas, and represents the response to a random jump within the memory map. A comparison between processors and different memory arrangements are made, and the effects of adding error detection is presented.

Chapter 8 shows how the reliability of specific systems can be improved by the addition of the techniques developed in the previous sections, and also suggests some hardware detection mechanisms. Chapter 9 describes a testing facility which has been constructed to physically check error detection and correction mechanisms. It allows the injection of a large variety of faults, and permits rapid testing.

Finally, the conclusions drawn from the research and the suggestions

for future development are presented in chapter 10.

CHAPTER 2

Practical Tests to Determine Transient Failure Mechanisms

2.1 Introduction

It has been shown in the previous chapter that methods of increasing the reliability of a system are generally designed to counteract a particular fault type, and are only effective if these faults are common. Ball and Hardie (5) have indicated that over 90% of field failures are due to intermittent or transient faults. Therefore techniques which enable recovery from the errors resulting from these faults will have a significant effect on reliability. The cause of these events have been discussed, but details of their effects, especially at the time of failure, are not fully understood. This is due to the random nature of their occurrence, which means that analysis of failure is usually possible only after the event when little data is available. The only indication that a transient has occurred may be that the system has crashed or an erroneous output has been made.

To enable the development of effective detection and recovery techniques, it is necessary to have a more detailed understanding of the mechanisms of failure. There are three methods available for investigation, and these are theoretical evaluation, computer simulation and practical tests. Theoretical evaluation relies on the assumption of certain fault conditions, such as single nodes stuck at 0 or 1, and the evaluation of their effects on the rest of the system. This is known as fault mode effect analysis (FMEA), and provides information about possible failure mechanisms. However, without a knowledge of the occurrence rate of the assumed faults, it is not possible to determine the most common failures.

With computer simulation, a model of the system at the transistor or gate level is produced, and this was the approach adopted for the Saturn V guidance computer described by Ball and Hardie (5). Faults can then be simulated and the effects observed, but this suffers from the same disadvantages as FMEA. Practical tests are the only way of determining which faults will occur in real systems. Once these have been established, FMEA and simulation can then be used more effectively.

Little information is available on practical testing of systems under transient disturbances. Those which are reported have focused their attention on methods of eliminating disruption by shielding or filtering. For example, Teets (101) states that short interruptions, of a few milli-seconds, can cause corruption to the contents of memory and also non-programmed jumps. He suggests that these problems can be overcome by the use of uninterruptible power supplies. Although vast improvements can be made in this way, it is not 100% effective in all cases, especially for unanticipated phenomena. For these cases it is necessary to adopt the fault tolerant approach.

This chapter describes work carried out to identify possible failure mechanisms, in small digital controllers, by the use of practical tests.

2.2 Test System

The test system which has been constructed for the purpose of identifying fault modes and their frequencies, is described in detail in the following sections. The hardware consists of a single processing board powered by a purpose built power supply unit.

2.2.1 Processor Board

The processor board is based on the design of a small single board computer given in the 8085 User's Manual (119). However, a few

modifications have been made for this application. A block diagram of the system is shown in figure 2.1, and the layout of the components is given in figure 2.2. Two main modifications have been added. Extra circuitry has been included to fully decode the on board memory, and an RS232 interface provides serial communications with a terminal.

The main components of the system are:-

8085	8 bit microprocessor
8155	256 byte RAM + 22 parallel I/O lines + timer
8755	2 K byte EPROM + 16 parallel I/O lines
6.144 MHz	Crystal

The power supplies to the three main integrated circuits, to the decoding circuits and to the RS232 interface are not permanently connected together, but are joined by removeable links. This allows the connection of an alternative supply to different parts of the board, so that the effects of interference on individual components can be observed. It should then be possible to identify levels of interference that effect different components before trying to analyse the whole system.

Resistors are connected to the data lines so that they can be pulled high or low.

2.2.2 Decoding Circuitry

The decoding circuitry consists of three LSTTL integrated circuits, and a logic diagram is given in figure 2.3. The inputs are taken from address bits 8-15 on the system bus and the outputs are connected to the chip select pins on the memory devices. The 8755 EPROM chip is mapped to the address range 0000 to 07FF (hexadecimal), and the 8155 RAM chip is mapped to the range FF00 to FFFF.

By fully decoding the memory and applying a suitable combination of

pull up and pull down resistors to the bus, a fixed data byte is forced onto the data lines when an attempt is made to access a non-populated memory address. By setting the data byte equal to a restart instruction, a software interrupt is generated when an instruction is fetched from a non-existant memory location. This can be used to detect some transient errors.

The chip selects are connected via wire-wrap or soldered links so that the behaviour of the system with full or partial decoding can be observed.

2.2.3 Power Supply Unit

For a computer to function correctly it is essential for the integrated circuits to be supplied with a good steady voltage. If the power supply can filter out mains borne transients then fewer errors will occur. The power supply therefore plays an important role in the overall system reliability. A circuit diagram of the test supply unit is shown in figure 2.4. The unit contains two transformers (one laminar and one toroidal), and three smoothing capacitors of different values. Two switches allow the selction of any combination of transformer and smoothing capacitor. This allows testing of the processor board to determine levels of interference that cause errors for different arrangements of the power supply.

2.2.4 Software

Two software packages have been written to run on the test equipment. One is designed to test the whole system and to display messages if an error is detected. The other is for identifying data errors in the RAM chip.

2.2.4.1 SYSTEST

SYSTEST is a software package designed to test the whole system. The

main part of the program writes a data byte into memory and then reads it back again. It then compares the value with a reference byte stored in memory and with another stored in the C register. If either values disagree an error code is sent to the terminal. This process continues by using the same byte in successive memory locations until the whole memory block FF10 to FFFF has been tested. The data byte is incremented and the process repeats until all values have been tried. If no errors are detected, a character is sent to the terminal to indicate that the system is functioning correctly, and the program restarts at the beginning.

Recovery software is included at the low order addresses of the memory, so that if a hardware interrupt, a software interrupt or a total reset is erroneously executed, then an error code is sent to the terminal and testing is restarted. This will occur if the program jumps into any of the unpopulated memory, provided that full decoding is used and the data lines are pulled high to force the execution of a Restart 7 instruction.

A number of different codes are included to indicate different errors so that the type of failure can be easily recognised. The test system has no monitor program, so the software package includes a subroutine to generate the software controlled serial output. To output a character the ascii code is passed to the routine in the C register, which then generates the serial data together with start and stop bits.

2.2.4.2 RAMTEST

RAMTEST is a software package designed to test for data errors in the RAM chip. The program requests a byte of data to be used in the test. It then writes that value into all the RAM locations FF00 to FFFF. When complete, a prompt is sent to the terminal and the program waits for an input before continuing. The data is then read back and displayed at the

terminal before starting again with a new byte of data. By including the wait between writing and reading, interference can be applied to the memory device during writing, during reading, between writing and reading, or during any combination of these.

This software is designed to test for corruption of the memory, and therefore the program cannot use the RAM for its own operation. The software includes a number of subroutines which deal with the serial communications. Subroutine calls are not used in the normal way, as the tests would corrupt the system stack. Instead, the return address, at which execution must resume, is loaded into the HL register pair. The routine is then entered by a normal jump instruction. At completion the PCHL instruction is used to load the program counter with the address stored in the HL register pair, and execution continues at that address. In this way all information for the correct operation of the program is stored in the internal registers of the processor rather than in memory.

2.3 Practical Tests Performed

Faults in digital circuits occur very infrequently, for example, a system similar to that described above has been operating continuously for over 4 months. Occasional interruptions to the power supply have caused full resets, but apart from these, no other errors have been detected. In order to observe the effects of faults, as they happen, it is necessary to induce failure.

In the British Gas application, the digital controllers will be situated in remote areas and will generally receive their electrical power from street lighting circuits. These are not particularly clean supplies, due to noise picked up from a number of sources. Bull (18) suggests that interference on the supply from devices such as thyristors, motors and gas

discharge lamps can cause disruption, or even permanent damage, to digital circuits. Therefore conducted interference on the power supply is expected to be a possible source of failure, and the tests have been aimed at this area.

Initial tests involved variations in the 5 volt supply rail. The levels at which errors occurred were recorded during manual reductions of a variable output supply. Other disturbances were created on the A/C mains input to the experimental power supply unit, using a Schaffner interference simulator. This consists of a main frame into which a number of plug-in units can be fitted. Three such units were available, and these cause short interruptions to the supply, or superimpose high or low energy spikes onto the mains.

The equipment generates interruptions of between 1.5ms and 500ms to simulate the change over of generators or breaks in the line. The low energy pulses of 2mJ have a rise time of 5ns or 10ns and an amplitude from 50 to 2,500 volts, to simulate interference from electromechanical switches and relays in close proximity. The high energy pulses of 2J have a rise time of approximately 0.3us and an amplitude of up to 5,000 volts to simulate the effects of thyristors, atmospheric discharges, high voltage current breakers and electrical machinery.

All tests with the high energy pulses showed no observable disruption to normal program execution. Using a digital storage scope, the effects of the spikes on the 5 volt rail were examined. With symmetric interference applied between live and neutral, no fluctuations were seen on the rail. However, with asymmetric interference between the two supply lines and ground, a 0.2 MHz oscillation of 0.6v amplitude, damped out after four cycles, was observed. This produced a minimum of 4.4 volts on the supply,

which is shown later to be insufficient to cause corruption. No variation in the response occurred with different values of smoothing capacitors.

To observe the effects of the other forms of interference, a Dolch logic analyser with a personality pod for the 8085 was used. This not only provides an indication of the states of each of the pins on the processor, during each clock cycle, but also provides a disassembly of the instructions executed. Unfortunately, with fast spikes the interference is sufficiently harsh to affect the operation of both the system under test and the logic analyser, and did not provide much useful data. Information about program execution during voltage reductions and short interruptions was readily obtained. However, during some testing, incorrect disassemblies were generated. This appeared to be due to the generation of additional clock pulses within the pod, causing the analyser to take extra erroneous samples. But by reverting to the display of binary states, it was possible to evaluate the actual processor response.

2.4 Test Results

As mentioned above, the test system was designed so that separate power supplies could be connected to each major device. Therefore interference tests were carried out on the individual chips, before being repeated on the whole board. This approach was adopted to try and identify the most likely sources of failure in a complete system. The results of these tests are given in the following sections.

Investigations on the effects of adding pull-up or pull-down resistors to the data lines, revealed only minor variations in susceptibility to interference. In all subsequent tests pull-up resistors were connected at all times.

2.4.1 Interference to the RAM

A random access memory (RAM) device has three main functions. These are to accept data from another device, to store the information, and to pass it back when required. Errors can occur during each of these states, and are termed write, data and read errors respectively. Voltage level tests were carried out, using the RAMTEST software, to determine the sensitivity of each of these operations.

The voltage levels at which the first errors occurred for different devices, are given in table 2.1. It shows that the read and write operations are the most susceptible to this sort of disturbance, while the data remains valid internally until at least another 1.4 volt drop in the supply. There is also a significant variation between devices. R3 and R4 were manufactured by Intel and are corrupted more easily than R5 and R6 which were manufactured by NEC. Slight variations in the level of first corruptions were observed for different data values, but these were all less than 80 mV.

An interesting observation was the variation in the location and value of the first error for different data bytes. These are summarised in table 2.2. All initial write errors gave a value of FF when read back, this was the value to which all locations were initialised before disruption. However, this was observed at various locations with the Intel devices, whereas the NEC devices always showed the first failure at location FF00. Similar observations were made with read errors, except one Intel device showed single bit errors at various locations, while the other consistently failed to FF at address FF00. For data errors the first events observed were single bit changes, and these occurred at various locations. However, a further reduction of only 50 mV resulted in multiple bit changes.

Although errors occurred at various locations for different data bytes, the results were always consistent for a particular device. For example, the first data errors for RAM chip R5 are given in table 2.3. This shows that bit changes in the device are more likely in certain bit positions. The table shows that, for device R5, bit 2 at address FFBF will always be the first to change if it is set to zero. Similar results were obtained for the other chips, but the errors occurred at different locations. This information could be used to test for general corruptions of data. The most susceptible bit could be checked periodically, and if correct would indicate that other corruptions were unlikely. However, this would create major problems in construction and maintenance, as each chip would have to be tested and the software modified accordingly.

For short interruption testing, a variable resistor was connected in parallel to the device under test, and adjusted to maintain a constant load of 500 mA on the power supply unit. This arrangement was adopted to allow comparisons to be made between different parts of the circuit. Table 2.4 shows the length of the interruption, in cycles, which caused the first errors for each part. As expected, a larger smoothing capacitor needed a longer interruption before errors occurred.

During this testing, RAM chip R4 suffered a permanent failure, and this is discussed further in section 2.6.2. Table 2.4 shows the results for device R3, and in each case the 5 volt rail dropped to a minimum of about 3.8 volts, before the first errors occurred. This is over 1 volt higher than expected from the previous results. However, in this case the software package SYSTEST was being used, indicating that the susceptibility to errors is dependent on the program being executed. This was confirmed by repeating the voltage reduction test while running SYSTEST, and showed

initial errors at around 3.8 volts.

Detailed investigations into the effects of applying low energy fast spikes to individual devices were not carried out. This was because no useful information could be obtained, from the logic analyser, due to corruptions caused by the interference. Limited results for full board testing under this type of interference are given in section 2.4.4.

Tests on early 4K RAMs have been carried out by Hnatek et al (46). The device studied required three different voltage levels of +5v, -5v and 12v. Supply reductions to the 5v rail showed initial data errors at around 1.2v, which is similar to those observed for the 8155. He also discovered devices which lost bits of data after 3 seconds if they were not accessed. Investigations showed that leakage currents, as a result of faulty manufacture, caused the bits to change state. This failure mechanism is particularly serious as in-circuit tests are designed to operate in the shortest possible time and would not detect them. A preventive solution is to refresh the memory as often as possible. The importance of this type of refreshing in counteracting the effects of soft errors due to alpha particle hits has been shown by Smith (95). However, in this case refreshing does not need to be carried out as often. An attempt to reproduce delayed errors on modern devices was unsuccessful. Two 8155s and sixteen 2114s were left unaccessed for ten days while filled with the value AA. This was repeated with complementary data, but in both cases no errors occurred.

2.4.2 Interference to the EPROM

Testing the erasable programmable read only memory (EPROM) was a much simpler process, as voltage variations can only cause read errors. Initially the supply was gradually reduced until the program started

sending error codes to the terminal. Repeating the test with the logic analyser connected showed no alteration in the voltage level at which first errors occurred, indicating that it did not affect the results.

Single bit errors were observed at a level of 3.43 volts. These were all changes from 0 to 1 in bit location 5, and occurred at a number of addresses. This resulted in the misinterpretation of instructions or the incorrect reading of operands. Further reductions in the supply caused more bits to change from 0 to 1, until FF was read during each instruction fetch at a level of 3.25 volts. In this condition the restart 7 instruction is executed repeatedly, pushing a return address onto the stack each time. This results in the stack extending through the entire memory map, destroying all volatile data.

Only one device was tested in this way. However, in previous tests on 2716 EPROMs, in a different system, similar results were observed. A common failure for one device was the misreading of a jump address, resulting in execution passing to an unpopulated area of memory. Another was the misreading of the operand in a compare instruction. In both cases the same bit showed a transition from 0 to 1. A similar device programmed with identical data also showed these types of transitions but in different bit locations. A similar response is therefore expected with other 8755s.

The lengths of interruptions necessary to cause corruptions are given in table 2.4. The minimum supply level reached for each capacitor was about 3.4 volts, which agrees with the previous results. One failure mode encountered during interruptions was the repetitive execution of interrupt routines. This was only observed with the logic analyser connected, and was due to oscillations on the interrupt lines. The problem was cured by removing the analyser, or by tying the lines low. Other failures were

similar to those for gradual reductions of the supply. Bit 5 showed the initial failures with other bits corrupted during longer interruptions.

2.4.3 Interference to the Processor

Voltage reductions on the processor revealed initial errors at a level of 2.74 volts, these consisted of bit 1 incorrectly read as 1 instead of 0, at several addresses. At a level of 2.72 volts the program counter showed signs of incorrect operation. This resulted in execution skipping over single and multiple bytes in the program. For example, the third and fourth bytes following a jump instruction were read as the jump address. This sort of execution was observed in several parts of the program.

Continuous servicing of interrupts, in the same way as in the previous section, was also observed. Again this was eliminated by grounding the interrupt lines. Another failure mode encountered, was the cyclic reading of data through memory. In this case the processor would read successive locations to the end of the memory map, and then repeat from the beginning. This mode was always entered if the supply was reduced to below 2.45 volts and then raised slowly. The processor would not leave this state with the application of a TRAP, which is supposed to be a non-maskable interrupt. A full reset is necessary to exit from this mode. A similar sequence of operation is encountered when certain op-codes are executed on the 6800. The fact that no further useful processing is performed under these conditons is particularly important from a reliability point of view, and this is discussed further in section 3.4.3.

The length of interruptions required to cause errors in the processor are given in table 2.4. First errors occurred when the supply reached a minimum of about 2.8 volts. Incorrect read and write operations were observed under these conditions. Slightly longer interruptions, causing a

dip down to 2.5 volts, revealed program counter malfunctions, as described above. Further reductions caused the processor to execute a sequence of restart 7 instructions (FF), but as the supply recovered the cyclic read mode was entered. This occurred for all interruptions which resulted in the supply rail falling to a value between 2.5 and 0.3 volts. If the supply dropped below this range, the power-on reset circuit would generate a correct reset.

2.4.4 Interference to the Complete System

Raising the power supply slowly from 0 to 5 volts, for the whole board, caused the processor to enter the cyclic read mode. This indicates that care must be exercised in starting up a system, and is to be expected as the power-on reset circuit will not operate correctly unless the supply is restored quickly.

Reductions in the power supply revealed initial memory read errors at 3.73 volts. This is a similar level to that observed with a reduction to the RAM supply. At 3.66 volts, memory read errors occurred at the stack locations, resulting in the incorrect execution of return instructions. At 3.46 volts, the system could not send error codes to the terminal. This was due to the incorrect reading of the EPROM, and prevented normal execution.

Interruption testing revealed comparable failures. The lengths of interruptions required to cause initial failures are given in table 2.4. As expected, they are similar to those for the RAM, which is the most susceptible part of the system to this sort of disturbance.

Low energy fast spikes were applied to the whole system, but even with 2.5 kV pulses having a 5 ns rise time, no observed failures were produced, provided that correct earthing and shielding of the equipment was used.

Without such an arrangement, errors could be induced. As mentioned above, the logic analyser could not be used effectively, to observe the point of failure, as it suffered from the interference. However, it could be used after the event to identify the final outcome of the fault.

Without grounding the chassis of the interference simulator, corruption of the stack pointer so that it pointed to an address in the EPROM, was observed. On returning from a subroutine, an arbitrary address was retrieved, and execution continued from that point. Subsequent calls attempted to overwrite the current stack position without success, and the following returns passed execution back to the same location as before. Corruption of the stack pointer was also observed during interruption testing on another system. This shows the importance of checking the stack pointer, or the return address, before leaving a subroutine.

The cyclic read mode could also be entered as a result of this type of interference. On another occasion the wait state was entered, and by applying a TRAP and observing the location to which execution returned, it was possible to establish the last byte executed before the wait. The processor had in fact read the operand of a conditional jump, which was equivalent to the code for a HALT instruction. Again, the repetitive servicing of interrupts was also observed, when the interrupt lines were allowed to float.

Finally, a few investigations were carried out with the analyser connected. Although the output was corrupted, a few conditions could be interpreted. These revealed occasions where the processor misread instructions. For example, a triple byte instruction was interpreted as three single byte instructions.

2.5 Significance of the Results

The test programme described above, relied on the assumption that the errors produced, under the various forms of interference, were representative of those which do occur in real systems. As with accelerated life testing, described in chapter 1, the experiments may reveal mechanisms which do not occur under normal operating conditions. However, the types of interference used were chosen to be similar to that expected in the particular British Gas application being considered. The aim was to simulate naturally occurring events, rather than to induce failure by altering the environmental conditions.

Another factor which suggests that the failure mechanisms observed will occur under normal operating conditions, is that in many cases a particular mechanism was observed as a result of different disturbances. This happened not only with similar interference on different parts of the circuit, but also with different types of interference. The results for the low energy fast spikes and the short interruptions are particularly important. Gradual reductions are less significant because they appear the same as short interruptions at the instruction level. Although sharp dips seem to occur as a result of an interruption, the minimum voltage is maintained for over a millisecond. During this time approximately one thousand instructions will be executed, and therefore individual instructions will see the interference as a steady low voltage level.

2.6 Observations of Permanent Failures

Although this work is aimed mainly at transient events, the detection and recovery processes should not be developed without consideration for permanent failures. Over the past three years several permanent component failures have been observed, and these are described in the following

sections.

2.6.1 Processor Failures

Two processor chips have experienced permanent failure, but despite this they have not failed completely. Certain parts of the integrated circuits still function correctly. Both failures occurred while the processors were operating on an Intel 8085 system design kit (SDK) board. The first processor appeared to fail for no particular reason and may have been a random failure. When connected to a system it seems to successively read through every memory location from 0000 to FFFF and then repeats continuously, in the same way as the cyclic read mode encountered during interference testing. All the control signals are correct for a successful read and the logic analyser confirms that the correct data for each address goes onto the data bus in the normal way.

This failure mechanism is particularly important when designing a watchdog timer for a system. It would not seem unreasonable to retrigger the timer on a certain address in the control program. Then if the system crashed and execution no longer continued around that address, the watchdog would reset the system and control would be restored. This arrangement is proposed by Oppenheimer (74) to recover from transient disturbances to the power supply. However, if the cyclic read mode is entered, the trigger address still appears at regular intervals and no reset or alarm would be set off, if the timing of the watchdog is not critical. This state could continue unnoticed for a considerable time. A complete memory cycle lasts for approximately 65 ms, with a 6.144 MHz crystal, and therefore the watch-dog must be set to a shorter time interval if address triggering is used. Oppenheimer also suggests that the watchdog could be designed to generate a non-maskable interrupt. It has been established from the tests that such

an interrupt is not recognised during the cyclic read mode and therefore cannot enable recovery from this state.

The second processor was damaged when the power supply failed during interference testing. Interruptions which result in restoration of the supply during a peak in the mains cycle cause a sharp spike in the current drawn. For the power supply used, the spike had a peak amplitude of up to 16 Amps, compared with a normal demand of approximately 300 mA, and was often sufficient to blow the input fuse. The power supply failed during interruption testing and was probably due to these surge currents. At the same time an LSTTL dual D flip-flop (74LS74) failed. All functions of the chip were lost and it took a current of approximately 1 Amp when attached to a 5 volt supply.

The only damage that appeared to occur to the processor was that one of the multiplexed address and data lines (AD5) stuck at '1'. This meant that for each instruction fetched, that particular bit would be read as a '1'. Therefore only half of the instructions could be read successfully, but it seemed that for the instruction that the processor had read, the correct execution followed. The program counter incremented internally in the normal way, with only the single bit corrupted externally on the address bus.

Again this failure is important when considering watchdog designs. One method of resetting the timer is to connect it to a port, and to use the OUT command. If a single bit stuck at '1' fault occurred which did not affect the OUT instruction or the port address, then it is possible for execution to continue in such a way that the watchdog would not generate a reset or alarm. This processor also entered the cyclic read mode occasionally but with the failed bit stuck at '1'.

2.6.2 RAM Failure

During interruption testing a permanent failure in an 8155 RAM chip occurred. Subsequent reading of the device gave a value of 3F at all locations. The chip was removed from the circuit and later replaced, at which time all locations appeared to be stuck at 00. At this stage the device drew a current of over 0.5 amps, compared with a normal consumption of about 40 mA.

The execution of the processor was affected. It operated with bit 7 stuck at 0, and read the value 3F from unpopulated memory. In the resulting execution a HALT instruction was incorrectly interpreted, causing the processor to enter the WAIT state. During subsequent tests, the processor would not respond in any way with the failed device in the circuit.

2.6.3 Crystal Failure

During initial trials of the single board test system regular problems were encountered in initiating correct operation of the processor. This problem was particularly evident when the 5 volt supply was instantaneously applied to the board. By slowing down the rise time of the 5 volt rail the problem ceased. However, the timing constraints for power on reset were satisfied in the original state.

Further investigations revealed that the problem was caused by the crystal. Under certain conditions it would oscillate at 18 MHz, three times its rated frequency of 6.144 MHz. Once it had started at that frequency it was necessary to apply a capacitance to the crystal to force it into its correct operation. A hardware reset had no effect, so a watchdog timer connected to the reset line on the processor would not restore correct operation from this failed state.

This problem has been cured by permanently connecting a 20 pF capacitor from the crystal to ground. The 8085 User's Manual (119) suggests that this should be done for crystal frequencies below 4 MHz. Their design for a single board computer does not include the capacitor, therefore they must consider it unnecessary at 6 MHz. This suggests that the crystal may have been faulty. However, this fault did not appear when four other processors were tested. Three of these were manufactured by NEC, whereas the other one, and the original, were manufactured by Intel. This seems to indicate an isolated fault in the internal oscillator circuit of the suspect device. All other functions of the chip operated normally.

2.7 Summary

The aim of the tests described in this chapter, was to identify failure mechanisms which are likely to occur in digital controllers. The mechanisms observed fit into two main catagories of corruption of data and disruption in the sequence of program execution. They both occurred under different types of interference applied to various parts of the circuit, and suggests that they will occur in real systems.

Corruption of data results from interference to each of the main elements of a digital system. Disturbances to the RAM allows data to be destroyed within the device, or during read and write transfers. In the case of the EPROM and processor, incorrect interpretation of instructions can result in the wrong data being accessed or the wrong operations being performed.

Disruption of the sequence of program execution can also originate from all three devices. Corruption of the stack data in the RAM results in incorrect returns from subroutines. Misinterpretation of instructions, due to interference in the EPROM or processor, can result in the execution of

erroneous jump, halt or stack operations. Disruption can also result from the direct corruption of the stack pointer and the program counter within the processor. Finally, the cyclic read mode and repetitive servicing of interrupts, both prevent any further meaningful execution.

Both these groups of failure are of great importance in control systems. The effects of data corruptions have been studied by a number of researchers, and methods have been developed to detect and correct them. These consist of the use of recovery blocks, N-version programming, rollback, time redundancy and reasonableness checks, and are discussed in chapter 1.

However, the sequence of events following disruption in the flow of program execution, has received less attention, and is studied further in chapters 4, 5, 6 and 7. It is of particular significance as without a resumption of valid execution, the data correction methods mentioned above, cannot function.

The results of the tests have shown that any value of op-code can be executed by the processor, either as a result of misreading instructions, or by accessing erroneous addresses. It is therefore necessary to know the effects of every op-code. This is discussed further in the following chapter.

It has been indicated that some failure mechanisms can have serious implications for the effective operation of watchdog timers. Further design considerations for these devices are also presented in the following chapter.

CHAPTER 3

Undeclared Operations of Microprocessors

3.1 Introduction

From a reliability point of view it is extremely important to know all the possible operations of a microprocessor. Without a full knowledge of their operation it may not be possible to design effective methods to counteract the results of transient or permanent faults. The manufacturers provide information about their devices, but this is not comprehensive. An obvious area of omission is in the declaration of the effects of all possible operation codes. This is important because the execution of a program can depart from its normal route, either due to a programming error or to some external interference. This has been demonstrated in the practical tests, described in the previous chapter.

Other undeclared operations are difficult to reveal. For example, the memory cycling mode on the 8085 was found by testing, and could not otherwise have been forseen.

Three manufacturers (Intel, NEC and Motorola) were contacted to see if they would release any further information other than that which is readily available, but they were not prepared to do so. Therefore the information required was only available from independent sources, or had to be established by experimentation.

3.2 Undeclared Operation Codes

The full instruction map for most 8-bit microprocessors has a total of 256 possible instruction codes. These take the values 00 to FF in hexadecimal. For a particular device a certain number of these codes will be defined by the manufacturer to perform specific tasks, but usually this does not cover the entire instruction map. The remaining codes remain

undeclared but inherently must operate in some way. An initial reaction might be to assume that they perform in the same manner as the instruction called a 'no-operation' (NOP). This is a slightly misleading name because although no data is altered, the program counter is incremented by one. Therefore even a NOP causes a change in the overall state of the processor. Alternatively if these undeclared codes cause a halt in the execution of instructions, this also is a change in the overall state.

As the codes are undeclared by the manufacturers there is a possibility that they may not perform in a logical fashion, or may not be repeatable even under similar conditions. Also, there is no guarantee that a particular response on one processor will be observed on another. This is particularly important where a specific processor is manufactured by several different companies. In this case it is possible that the chips may be fabricated using different masks and it will be highly probable that the undeclared codes will function differently. For example, it has been suggested in (118) that Intel and National Semiconductor use the same masks for the 8080, whereas NEC and AMD have developed independent designs. This has been established from the operation of the auxiliary carry flag, which does not always function correctly on the first two manufacturers devices.

However, it is believed with the 8085 that Intel have specified, to other manufacturers, exactly what each code should do, and the codes which they say are undefined are in fact only undeclared to the final user of the device. Similar cases may also exist with other processors, but should be treated with extreme caution as new modifications may depart from previous arrangements. This has been demonstrated by Nemmour (67) who reports on differences between 6800 microprocessors manufactured before and after 1977 by the same companies. He suggests that some of the changes were to

correct design errors in the original masks.

The undeclared op-codes of various microprocessors are discussed in the following sections, along with some other undisclosed functions.

3.3 Operations of the 8085

The 8085 is a typical 8 bit microprocessor with a 16 bit address bus. It interprets all operation types from a single byte, and therefore 256 different op-codes exist. Intel only define 246 of these codes leaving 10 undeclared. The functions performed by the undeclared codes have been investigated by Dehnhardt and Sorensen (28). Not only do they perform in a logical way, but they also provide some very useful operations, such as 16 bit additions, subtractions and rotations. The same results can be achieved using sequences of other instructions, but this involves extra execution time and memory space.

Also revealed in (28) is that two of the bits in the condition code register, which are supposedly undefined, also perform in a logical fashion. They state that bit 1 indicates a two's complement overflow, whereas bit 5 indicates an unsigned overflow for data changes between 0000 and FFFF, when executing 16 bit increment and decrement instructions. These flags are used by some of the undeclared instructions.

This leads to the question of why the codes and flags are not declared by the manufacturers. The 8085 has close links with both the 8080 and the Z80, with most of the op-codes performing in the same way. Therefore the extra codes may have been left undeclared to maintain a high level of software compatability between the devices. When asked about the codes, the manufacturers stated that they could not be guaranteed to work under all conditions, suggesting that pattern sensitive faults, introduced at the design or manufacturing stages, may be present.

Dehnhardt and Sorensen (28) suggest that the op-codes and flags can be used to enhance programming, and it is known that they have been used in some applications. Clearly this is a dangerous situation if pattern sensitive faults do exist. Investigations on an Intel 8085 by Buchholz (17) revealed pattern sensitivity in the over-flow flag. During addition and subtraction, 25 particular operations resulted in the incorrect setting of the flag. Similar errors were observed with the compare instruction.

As indicated above, processors from different manufacturers, or different batches, may vary in their response, and for this reason modern devices were tested to compare with the published results. The undeclared op-codes were executed on an Intel SDK board. The monitor program, provided with the kit, allowed the setting of registers and flags prior to the test, and also the interrogation of their values afterwards. A Dolch logic analyser, with an 8085 personality pod, was connected to the processor to enable all external pins to be monitored. Most of the instructions can be checked without the analyser, especially with a prior knowledge of their operation. However, it does provide verification of data transfers, and is particularly useful in monitoring the flow of execution after conditional jump instructions. These operations are difficult to monitor with software alone.

Full testing of all the instructions for every possible combination of data values would take a considerable length of time. For this reason, tests were carried out, both with random data, and data selected to check specific responses. All ten undeclared codes were executed on an NEC 8085 and responded in the same way as that described by Dehnhardt and Sorensen. NEC have developed independent designs for the 8080 (118) and the 8035/8048 (see section 3.5), which suggests that Intel may have specified to other

manufacturers how all codes of the 8085 must perform. A nuclear hardened version of the 8085, described by Kim et al (51), was developed from information provided by Intel and has all the op-codes defined. This suggests that all 8085s should operate in the same way.

However, tests were carried out to attempt to reproduce the apparent malfunctions observed by Buchholz (17). Both NEC and Intel 8085s were subjected to the same operations which were reported to have incorrectly set the proposed two's complement overflow flag. At all times during testing the flag was set correctly. This indicates that the errors were observed on an isolated faulty component, or that a fault existed in the masks of a particular batch which has been corrected on other devices.

Another undeclared operation, which was discovered during interference testing, is the continuous cyclic reading of memory. This is described further in chapter 2, and its implications for reliability are discussed in section 3.8.1. Due to the complex structure of a microprocessor, other modes of operation, which have not been discovered, may exist.

3.4 Operations of the 6800

The 6800 is also an 8 bit microprocessor with a 16 bit address bus. Again there are 256 different possible operation codes, but only 197 are defined, leaving 59 undeclared. The functions performed by all the codes have been studied by Nemmour (67). However, practical tests were carried out to determine the operation of the undeclared codes, without a prior knowledge of the published results. The methods used are described in detail as they can be used in the study of other processors.

3.4.1 Determination of the Undeclared Instructions

Studying the positions of the undeclared codes, in relation to the defined instructions in the instruction map, provides a useful starting

point. A number of the undeclared codes are situated in adjacent locations, suggesting that they may have similar operations but use different addressing modes. By considering the defined codes, alongside the one under investigation, it is possible to suggest the likely addressing mode. These suggestions proved correct in the majority of cases and assisted greatly in the determination of many of the operations.

To check the expected operations provided by the codes, they were executed on a small 6800 based system. A short assembly program was written to assist in the investigations, and a full listing is given in appendix 1. It effectively uses the MIKBUG routines to read in values from the terminal and to set the registers accordingly, before the execution of the required op-code. The data read in is stored in successive locations in memory and then the stack pointer is set to the location above the block. A return from interrupt instruction is then executed to load the correct values into the corresponding registers. This ensures that all the registers and the condition codes can be set to any value. A series of software interrupt instructions are placed after the op-code to use the MIKBUG routine to print out the contents of the registers and condition codes.

This provides a clear indication of any changes that have occurred within the processor due to the specific op-code. However, it does not give any indication of external events such as reading and writing to memory, and is of little use in cases where a jump or branch is generated. In these cases a logic analyser was used to monitor the states of the address and data buses, and the read/write and valid memory address lines. This enabled all external data transfers to be monitored, and clearly indicated the flow of execution after jump instructions. Without

monitoring the external pins of the processor, it would not have been possible to establish all the operations performed.

3.4.2 Functions of the Undeclared Codes

The functions performed by the undeclared codes fit into two main groups, those which perform totally new operations, and those which perform identical or similar operations to the instructions already defined. Most of the codes are similar to the ones specifically defined by Motorola. They perform roughly the same operation but will manipulate the flags differently or not change the contents of a register. For example there is an add accumulators instruction indentical to the defined instruction except that the half carry flag is not affected.

Some of the codes are indentical to defined ones and appear to be due to the instruction map not being fully decoded in some places. Examples of this are the four addressing modes for the compare X register instructions. These are normally codes 8C, 9C, AC and BC, but also appear at CC, DC, EC and FC. This suggests that bit 6 is ignored when the instructions are decoded.

Some of the codes are substantially different and appear to perform useful tasks, however these functions can also be performed by two or more of the defined instructions. For example there is an add accumulator to the complement of memory instruction. It works for both accumulators, and for all four addressing modes. All the flags except for the half carry and the interrupt mask are affected by the result of the operation. Another useful instruction performs a logical AND on the two accumulators and puts the result in the A register, three of the flags are affected. A similar instruction affects the flags but does not change the contents of the accumulators.

Store immediate operations exist for the A, B and X registers and for the stack pointer. To be consistent with the load immediate instructions they should store the data in the memory locations immediately following the instruction, but this does not occur. Instead, the first byte is skipped and the data is written to the following locations. However, the program counter is adjusted accordingly so that the next instruction is read from the location immediately after the one into which the last data byte is written. This effectively makes the store A and B registers into triple byte instructions, and the store X register and stack pointer into instructions with four bytes. But in all cases only one byte is read.

3.4.3 Cycling Through Memory

Four of the undeclared op-codes cause the processor to cycle through memory indefinitely. This state is of particular importance when considering reliability. It means that if one of these op-codes is inadvertently executed, either due to an error in programming or to some external interference, then the processor will 'lock-up' and will not execute any further instructions until some external intervention is initiated.

Operation codes 9D and DD cause the processor to read through memory starting at the direct address following the code. Once in this state it will not respond to either a non-maskable interrupt (NMI) or an interrupt request (IRQ), even if the interrupt mask is cleared beforehand. The only way of leaving this state is to exert a full reset on the processor. The contents of the A, B and X registers are not altered from the state that they were in before the op-code was executed. This was determined by generating an interrupt immediately after the reset. Unfortunately the interrupt will not occur until after the first instruction has been

executed. In the system used the first instruction loads the stack pointer, and therefore its contents at the time of the reset could not be determined.

Those condition codes which were not affected by the first instruction or the reset, remained in the same state that they were in originally. This suggests that no change occurs in any of the internal registers while the processor is cycling through memory. Therefore the only data lost are the contents of the program counter and the state of the interrupt mask, which are both set by the reset sequence. The contents of the registers will not be of great use after the reset, as some unforseen sequence of instructions will have been executed before the undeclared op-code was reached. However they may give some sort of indication of how that particular state was entered.

The result of executing operation codes 3C and 3D is similar to that obtained by the codes 9D and DD, in that the processor ends up cycling through memory reading successive locations. After executing the code, it differs by saving the address of the next byte onto the stack, before reading the next location on the stack. It rereads the previous location and then starts cycling through memory from the top of the stack.

While in this state the processor will not respond to NMI or IRQ, as before. Again the only means of leaving this state is by a reset. Nemmour (67) suggests that this is due to the way in which interrupts function. They do not respond until the completion of an instruction, and therefore, because these operations never finish, no interrupts can be initiated.

However, the B and X registers are not changed from the state that they were in before the undeclared op-code was executed, but the A register is changed. Bits 1-7 are cleared while bit 0 remains unaffected. Again,

it was not possible to determine the value of the stack pointer after the reset. If it remains unaltered then the address at the top of the stack will point to the byte immediately after the illegal op-code that was executed. This is a very important point when attempting to diagnose the original fault, and could prove very useful.

The reason for these modes of operation is unclear, but it is believed that they may be for testing purposes. Hayes and McCluskey (43) propose a test sequence for the 8080 which starts by executing NOPs repeatedly. This is designed to reveal faults on the address bus. However, the cyclic read mode is not only suitable for revealing address bus faults, but can also indicate data bus and memory failures.

3.4.4 Comparison with Published Data

The investigations by Nemmour (67) were carried out in a similar manner, but in addition he studied the masks to enable cross checking with practical tests. Devices from different manufacturers (SESCOSEM and Motorola) were used, however these are constructed from identical masks. In all cases the instructions operated in the same manner as the independent investigations described above. This shows consistency between devices from the same manufacturer, but variations may be obtained if different masks have been developed. Again, it is unclear why these operations are not disclosed. Design or manufacturing difficulties could have caused problems, and these may have been corrected subsequently.

Nemmour reveals several changes that were made to the masks in 1977, some of these were to correct initial errors. For example, on original devices, the application of a non-maskable interrupt, during certain cycles of the execution of a software interrupt, caused the servicing of the maskable interrupt routine. This sort of fault is particularly difficult

to locate, and others of a similar nature may exist.

3.5 Operations of the 48-series Microprocessors

The 48-series microprocessors are also 8 bit devices but have a very different architecture from the 8085 and 6800. They consist of a central processing unit, 27 I/O lines, a single interrupt and an internal timer/ counter. In addition to this a quantity of internal read only and random access memory is provided, the size of which depends on the particular device, and is given in table 3.1. The processors are designed for small scale control applications where the final program would reside in one of the ROM based chips. The other devices are primarily for use in the development and debugging stages.

The address bus is 12 bits wide allowing a maximum possible address range of 4K bytes. The program counter is however only 11 bits long, and effectively splits the memory map into two separate blocks. Access to each area is controlled by software which can alter the most significant bit of the address bus. The internal RAM is not accessed by the main bus, and its contents can only be treated as data, no instruction fetches can be made from it. Therefore the normal arrangement is to locate the program within the 4K address range, and to use the internal RAM for data storage. However, fixed data values can be stored in the main memory map, but they are less easily accessed.

External memory devices can be attached to the processors to supplement the internal memory. Alternatively, devices can be mapped to the same locations as the internal ROM, and the processor forced to access them instead. This can be used in the development stage, or to provide an alternative program, such as for testing purposes.

The processors interpret the instruction type from 8 bits, and

therefore 256 possible op-codes exist. Only 230 are defined, leaving 26 undeclared. No published work on the undeclared operations of these devices has been found, and therefore investigations were carried out to determine the effects of executing the undeclared codes, and to discover other undisclosed functions. Full details of these studies are given in the following sections.

3.5.1 Undeclared Memory in the 8035

In all published literature, the major manufacturers state that the 8035, 8039 and 8040 have no internal ROM. However, it was suspected that this might not be the case, and attempts were made to read internal memory of 8035s as if they were 8048s. For nine devices from three different manufacturers (Intel, NEC and National Semiconductor) a logical program of up to 1K was revealed. The Intel 8035 contained a games program which read 9 bits of parallel data from port 1 and test input T1, and used bit 7 of port 2 and test input T0 for transmitting and receiving serial data. It is therefore clear that the 8035 is in fact an 8048 but sold under a different name. When approached on this matter, Intel did admit that they are the same device, and that 8048s which do not operate at the required speed, or have faults in the ROM, are sold as 8035s.

This fact raises two important points. Firstly, any details of the undeclared codes of the 8035 will relate directly to the 8048. Secondly, the existence of an internal program might have serious consequences with respect to reliability. The internal program is disabled by holding the external access (EA) pin high, but an internal chip failure could cause the pin to be disabled resulting in bus conflict or the correct execution of the internal program. This could result in a dangerous sequence of signals appearing at the ports and could mislead any external hardware monitoring

the state of the system.

The external access pin does not operate in the same way for all 8035s. If allowed to float, the Intel chip accesses the external memory, whereas the NEC chip accesses the internal memory. For the National Semiconductor device access to both memories appears to occur. Normally the pin would be tied high or low, but an internal wire bond failure, due to thermal stress or vibration, could cause it to float. For this type of failure a particular device will continue without error, depending on which memory contains the main control program.

3.5.2 Determining the Undeclared Instructions

In order to determine the operation of the undeclared codes, a small 8035 based system was constructed. A block diagram of the system is shown in figure 3.1. It consists of the processor, an 8-bit latch and a 2K EPROM. The latch is necessary in order to demultiplex the address and data bus. An EPROM emulator was used to enable quick and easy modifications to the program being run.

The software used to investigate each code is given in appendix 1. The program outputs the contents of the accumulator onto port 1, executes the undeclared op-code and then re-outputs the accumulator to port 1, before incrementing the accumulator and restarting. All unused memory is set to 04, this causes a jump to address 004 if an attempt is made to execute outside the program. This method is also used to recover execution after the undeclared code. A subroutine call during each cycle is included to monitor the state of the stack.

A logic analyser was used to monitor the state of the ports and bus. The clock output on the T0 pin was used as the clock input to the logic analyser, causing one sample to be taken during each T state. This is

equivalent to five samples during each program cycle. In this way it was possible to determine the number of bytes associated with each code and the number of cycles it took to execute. Any effects on the ports, bus or accumulator could also be seen.

As the processor is designed to be used as a single chip controller, many of the instructions result in only internal actions, and cannot be observed externally. In order to establish internal operations, further investigations were carried out using a Prompt 48 microcomputer design aid. This allows programs to be executed from RAM and enables access to all of the internal registers and flags. Using this system it was possible to reveal any internal effects of the undeclared codes.

3.5.3 The Effects of Executing the Undeclared Codes

A detailed list of the effects of executing each of the undeclared op-codes is given in appendix 2. Unlike the 8085 and 6800, in this case, processors from different manufacturers give different results. Devices from the three manufacturers of Intel, NEC and National Semiconductor, were studied. The results from the National Semiconductor 8035/8048 were identical to those from Intel, and therefore have not been included in the detailed descriptions in the appendix. It seems to be the case that National Semiconductor do not produce independent designs for their devices, and this is supported in (118).

The full instruction maps, including the undeclared codes, for both the Intel and NEC devices are given in figures 3.2 and 3.3. Descriptions of each of the operations of the undeclared codes are given below.

3.5.3.1 Intel 8035/8048

Figure 3.2 shows that, for the Intel chip, out of 26 undeclared codes, 17 perform a No-Operation, 4 cause a jump in execution and the remaining 5

affect the input/output lines. Three of the jump instructions are logical extensions to the standard instruction set. They are conditional on a particular flag being clear and, in the instruction map, they are adjacent to their corresponding jump, conditional on the flag being set. The fourth additional jump instruction is unconditional and branches to an address within the current page. This is not provided for directly in the standard instruction set, and enables program modules to be relocated on a different page without modification.

Four of the additional I/O instructions are identical to codes in the standard instruction set. They are copies of the four operations involving port 2, and each one is adjacent to its copy in the instruction map, suggesting that bit 0 is not used in decoding these instructions. The fifth code involving the I/O lines has the value 38. By considering the adjacent locations in the map, an OUTL BUS,A instruction would be expected, which outputs the contents of the accumulator to the Bus. This undeclared code does take two machine cycles to execute, which is necessary for an I/O function, but no read or write signal is generated to perform a correct bus operation. The value 00 does appear on the Bus during T4 of the second machine cycle, but this does not seem to perform a useful task. No other part of the processor appears to be affected.

3.5.3.2 NEC 8035/8048

Most of the undeclared codes for the NEC device are the same as those already described above. All the jump and I/O operations are the same, but six of the No-Operations have been replaced by useful instructions. Four of these fit logically into the instruction map and perform functions not previously provided. They fill the gaps for the indirect addressing modes of the decrement, and the decrement and jump if not zero instructions,

which are omitted from the standard instruction set. There does not seem to be any logical reason why these instructions should be omitted. Errors during initial development of the processor may have caused problems which have now been corrected by NEC.

Two of the instructions perform functions totally unrelated to those already defined. One has the effect of clearing the upper nibble of the accumulator (bits 4-7). The other loads the accumulator with the lower 8 address bits of the next sequential instruction to be executed. The first instruction is useful when manipulating nibbles, whereas the second could be useful when debugging a program. In the latter case this code could be placed in several locations in a program followed by an output to a port. Then by monitoring the port it would be possible to trace execution past these points.

3.5.4 Other Devices in the Series

All investigations were carried out on the 8035/8048. The only declared difference, with the other devices in the series, is the size of the internal memory. These chips are therefore likely to have similar properties. For example, the undeclared op-codes are expected to function in the same way as those in the 8035/8048, and the devices which are defined as having no internal ROM are expected to have internal program memory.

3.6 Operations of the 68000

The discussion up to now has been directed towards 8 bit micro-processors, but it is now worth mentioning the Motorola 68000, which has a 16 bit internal architecture, and a 24 bit address bus. The type of operation performed is determined from a full 16 bit data word, and therefore the total number of possible op-codes is much greater than for an

8 bit machine, and is in fact 65,536. Obviously with such a large number of possible codes, there will be a substantial quantity which are not defined. The 68000 has 56 basic functions, but with all the addressing modes and register references approximately 45,800 op-codes perform defined operations leaving over 19,700 unused. However, the processor has been designed to signal an exception if it detects the illegal execution of any of these codes. This effectively means that each one of them acts as if it were a software interrupt.

A study of the full instruction map reveals that the unused codes appear in isolated locations as well as large groups, some up to 4K. The manufacturers state that codes in the large groups may be used in later designs. For this reason they cause the execution of a different exception handling routine from the other codes, if an attempt is made to execute them. This allows the emulation of new instructions on the original devices. It was felt that if any of the unused codes were going to perform undeclared operations, then the isolated ones would be the most likely to do so. For this reason, a number of the codes were executed on a small 68000 based single board computer. In all the cases that were tried, a correct response from the exception handling logic was observed. No unusual operations were revealed.

As well as the detection of unused op-codes, internal logic is provided to detect other erroneous states, such as an attempt to perform an instruction fetch from an odd address. The processor is also specifically designed to have external logic to detect unsuccessful memory transfers. All these checking modes are important from a reliability point of view. They reduce the probability of executing a large number of erroneous instructions before detection.

Although this is an advantage in the detection of errors due to transient faults, the permanent failure rate will be higher than that for 8 bit processors due to the increased complexity of the chip. This may also increase the susceptibility to transients.

3.7 Operations of the 6809 and Z80

The 6809 and Z80 are both 8 bit microprocessors, but they differ from those described above. In some cases the instruction type is not established from 8 bits alone. This allows the possibility of undeclared op-codes at different levels. The 6809 is a modified version of the 6800, and has a similar instruction set with the majority of the instructions, at the first level, still having the same operation codes. This is particularly evident in the ranges 20-2F and 40-FF where nearly all the codes are the same.

An interesting point is that two of the previously undeclared codes are replaced with instructions which would logically be expected from looking at the memory map. Code 21 has been programmed to execute a branch never instruction, which is the logical opposite of code 20, the branch always instruction. Code 9D executes a jump to subroutine using direct addressing. This was a previously omitted form of addressing in calling subroutines, and fits in between the other addressing modes. Nemmour (68) has identified 498 undeclared op-codes, at different levels, in the 6809, four of these cause the cyclic reading of memory in the same way as those described for the 6800 and 8085.

No further studies were carried out on the 6809 and Z80. They have been mentioned here to indicate possible problem areas for other processor architectures.

3.8 Implications of the Undeclared Operations on Reliability

Undeclared operations of microprocessors can be divided into two broad catagories, those which occur as a result of executing an undeclared op-code, and those produced by other mechanisms. Most of the undeclared codes in microprocessors operate in a similar way to the declared instructions, and therefore their importance does not differ significantly from the erroneous execution of defined instructions. However, there are some codes which operate very differently, and are of great significance. These result in the processor cycling through memory and prevent the execution of further instructions until a reset. Therefore some sort of watchdog timer must be included in a system to recover from these states.

3.8.1 Significance for Watchdog Design

When considering the design of a watchdog timer the following two points should be noted. Firstly, the highest level of fault recovery must initiate at least a full reset, and secondly the address lines alone should not be used to trigger the timer. The second point is particularly important in the 6800, which uses the address lines for calculating branch and indexed addresses. The triggering must incorporate the write signal which is never present unless a valid write operation is being performed.

The existence of memory cycling can be considered as an advantage or a disadvantage depending on the application. If the accuracy of a system is more important than its timing, then this mode of operation would be an advantage as it has the effect of suspending execution, preventing any further output. On the other hand if timing is more important, the considerable amount of time which could elapse between the occurrence of the fault and the detection of the error by the watchdog, would be a disadvantage. Further time could be lost resetting the system and

reinitialising variables.

However, even in the first case, the major drawback is that recovery has to be initiated by some hardware. If the timer fails the whole system fails. A better solution would be to attempt to detect and correct errors under program control and only rely on external hardware when this approach fails. This cannot be achieved with these particular codes, therefore the only method of providing a back-up procedure in the case of a watchdog failure is to include further hardware to monitor its operation.

The existence of internal memory in the 8035, when being used for control purposes, is also important for watchdog design. A chip failure, such as a wire bond fracture or an internal short, can result in the execution of the internal program. It may then operate in such a way that the errors go undetected. This is possible, as the devices are used in I/O intensive situations and therefore the ports, to which a watchdog would be connected, will probably be highly active. If a simple triggering sequence is used with non-critical timing the internal program could generate signals which would satisfy the timer. However, a complex triggering sequence will reduce the likelihood of non-detection of this type of failure.

3.8.2 Powering down to Enable Recovery

It has been shown by the crystal failure that it may be necessary to provide a level of recovery which goes further than a reset and actually powers down the system before powering up in a controlled manner. This is because with the crystal oscillating at three times its natural frequency the application of the reset has no effect and the power has to be removed before correct operation will resume. This situation has been cured by the addition of a small capacitor, but does at least demonstrate that it is not

always sufficient just to apply a reset.

3.8.3 Use of Non-Maskable Interrupts

The memory cycling mode has shown that non-maskable interrupts should not be used to initiate recovery. However, if they are used for other purposes, great care must be exercised in their handling. A noisy signal on the input can cause multiple interrupts and result in a large quantity of data being stored on the stack, which may result in overflow and the overwriting of critical data areas. It is therefore advisable to reset the stack at the beginning of the routine, if the return address is not required, or to at least check that the stack pointer is within certain limits.

3.8.4 The Most Important Undeclared Operations

The failure modes which present the major threat to the integrity of a system are those which have not been discovered and cannot be forseen. Any amount of time can be spent designing against the effects of known or expected failure modes, but inevitably it is the unknown modes which cannot be designed against fully. It is hoped that high level detection mechanisms, such as watchdogs, will allow recovery from these types of failure.

3.9 Summary

This chapter has shown that microprocessors perform a number of operations which are not declared by the manufacturers. Some of these can have serious consequences in the design of error detection and correction techniques, and therefore a knowledge of these modes of operation is necessary in order to achieve high reliability.

A common failure mode observed during the interference testing, described in chapter 2, was a transfer of program execution to a non-

specific memory location. This can result in instruction fetches from data areas or operand fields, and any value of op-code can be read. The functions performed by executing each op-code have been determined for the 8085, 6800, 8048 and 68000, either from published data or from practical tests. With a knowledge of all the op-codes it is possible to predict the flow of execution in different memory areas, after an erroneous jump, and this is discussed in detail in chapters 4, 5, 6 and 7. From these studies it is possible to design more effective error detection and recovery processes.

It has been established that the undeclared operations do not always function in the same way in devices from different manufacturers, and changes can occur between different revisions of the masks. Therefore the results may not always be consistent between any two devices. It has been suggested by Nemmour (67), and by Dehnhardt and Sorensen (28), that the undeclared codes can be used to enhance programming, but this would seem to be a very dangerous practice.

CHAPTER 4

Erroneous Execution in Data Areas

4.1 Introduction

During the practical tests on the small single board system, described in chapter 2, it was shown that corruption to the normal flow of execution could be generated by applying different types of interference to particular parts of the circuit. The three main elements on the board, consisting of the processor, EPROM and RAM, could each cause such a failure. Although the particular failure mechanism is different in each case, they can be divided into the two main catagories of the misinterpretation of instructions, and the incorrect return from subroutines.

The misinterpretation of instructions occurs either due to the incorrect transmission of data from the EPROM, or to the corruption of the program counter within the processor resulting in the wrong bytes being read. Incorrect returns from subroutines occur by the corruption of either, the stack pointer within the processor, or the stack data stored in the RAM.

The tests therefore show that this class of failure is likely to occur in real systems under certain types of interference. It is particularly important because without knowledge of the behaviour of a system after such a failure, it is not possible to effectively design hardware or software methods to detect and correct system operation. For example, a common solution to this problem is to attach a hardware watchdog timer but, as will be shown later, without careful consideration to the design, certain failures will not be detected by the circuit.

4.1.1 Random Jump Within the Memory Map

When program execution departs from its predefined sequence it must continue at some other location. For the purpose of the following analysis it is assumed that the location is random within the full memory map. Therefore the failure mode being investigated is equivalent to the erroneous execution of a jump instruction to a random address. For a typical 8-bit microprocessor with a 16-bit address bus this gives a possible 65,536 different locations at which execution could resume after the fault.

However, certain parts of the memory map have different properties dependent on the type and sequence of values which are read when various locations are accessed. In the following sections three main categories are studied, these are program areas, data areas and unused areas. The effects of memory mapped input and output is also considered. This chapter studies the flow of execution after a random jump into a data area.

4.2 Analysis of Execution

If as a result of an error execution resumes in a data area, the processor will interpret the data as instructions and perform the corresponding operations. Obviously the type of data and its arrangement in a particular block will depend very much on the application and method of programming, and in the case of random access memory, will change during execution of the program. Therefore to analyse this type of execution for the general case it is necessary to assume that the sequence of bytes is totally random.

From this it follows that when a data byte is interpreted as an instruction one of two possible outcomes will be performed. Either, a jump will be generated causing control to pass to another part of the memory

map, or a non-jumping instruction will be interpreted passing control to the next logical byte. For the latter case the whole process repeats again. If P_J , the probability of interpreting a jump instruction, is zero then execution would continue to the end of the data block. However, assuming random data, P_J will be dependent on the particular instruction set of the processor, and is given by:-

$$P_J = \frac{N_J}{N_T} \qquad \text{Eqn. 4.1}$$

Where:- N_J is the number of bytes which cause a jump or branch.

N_T is the total number of possible op-codes (256 for a normal 8-bit processor).

Clearly, P_{NJ} the probability of interpreting a non-jumping instruction is given by:-

$$P_{NJ} = 1 - P_J \qquad \text{Eqn. 4.2}$$

It follows that, $P_J(K)$, the probability that K instructions will be executed before control passes to another part of the memory map, can be obtained from:-

$$P_J(K) = P_{NJ}^{(K-1)} \cdot P_J \qquad \text{Eqn. 4.3}$$

An important quantity, which will be used later in chapter 8, is the average or expected number of instructions which will be executed before the jump, NI_{AV} , and is given by:-

$$NI_{AV} = \sum_{K=1}^{\infty} K \cdot P_J(K) \qquad \text{Eqn. 4.4}$$

It is useful for determining both the time taken to initiate recovery, and the probability of corruption of specific data. The average number of bytes read, NB_{AV} , will be greater than NI_{AV} because each instruction interpreted can consist of one or more bytes, therefore NB_{AV} will be given

by:-

$$NB_{AV} = (NI_{AV} - 1) \cdot \sum_{L=1}^{\infty} \frac{L \cdot N_{LNJ}}{N_{NJ}} + \sum_{L=1}^{\infty} \frac{L \cdot N_{LJ}}{N_{J}} \qquad \text{Eqn. 4.5}$$

Where:- N_{LNJ} and N_{LJ} are the numbers of bytes interpreted as non-jumping and jumping instructions of length L.

N_{NJ} and N_{J} are the total number of bytes interpreted as non-jumping and jumping instructions.

To assist in the calculation of both NI_{AV} and NB_{AV} , a short FORTRAN program was written. It requests a number of details about the particular instruction set and then calculates the values using equations 4.4 and 4.5.

NB_{AV} is used later in chapter 7 when considering the flow of execution as it passes between different parts of the memory map. The following section looks at a few microprocessors and goes through the necessary steps to calculate the above quantities.

4.2.1 Response of Different Processors

To determine the expected response for a particular processor it is necessary to make a detailed study of the instruction set. For execution in the data area the important instructions are those which cause a jump or transfer of program execution. Appendix 3 lists a number of parameters for the 8085, 6800, 8048 and 68000 microprocessors, it includes the effects of the undeclared codes. The importance of chapter 3 in determining the undeclared op-codes is now clear, as without the knowledge of them inaccurate results would be obtained.

The instructions are divided into two groups, those which always cause a jump and those which are conditional on some internal state of the processor. To include the properties of the conditional jump instructions, equation 4.1 is modified to:-

$$P_J = \frac{N_J + \sum_{i=1}^{N_{CJ}} P_{CJ}(i)}{N_T} \qquad \text{Eqn. 4.6}$$

Where:- $P_{CJ}(i)$ is the probability that the ith op-code of N_{CJ} conditional instructions causes a jump.

For ease of calculation it is assumed that there is a 50% chance that a jump will occur. Although this is not strictly true in individual cases, overall the assumption is valid. This is because in most cases the instructions have a logical pair which tests the inverse state of a particular condition, and therefore any variations in the probabilities will be cancelled out. In this case equation 4.6 simplifies to:-

$$P_J = \frac{N_J + 0.5 \cdot N_{CJ}}{N_T} \qquad \text{Eqn. 4.7}$$

However in a few cases a different approach was adopted. For the 8048 decrement and jump if not zero instructions, it is assumed that they always cause a jump. Provided that the contents of the particular register concerned is random, then there is only a 1 in 256 chance that the jump will not occur. They are therefore grouped together with the other jump instructions.

Special treatment has been given to the 68000 instruction set. This is due to the fact that most of the instructions which can cause a transfer of control have several possible outcomes. For example, the branch on condition code instruction first makes a test and if not true, no branch occurs. This is assumed to have a probability of 0.5 for the same reasons as above. If a jump does occur it is assumed to be random, in which case there is a 50% chance that the address will be odd. The processor can only read instructions from even addresses and generates an exception if an

attempt is made to access an odd address. Therefore if a branch on condition code instruction is executed, the probability of no jump is 0.5, the probability of generating an exception is 0.25 and that of a successful jump is also 0.25. To simplify the calculations the op-codes for this instruction are split in the same proportions to give an effective number of op-codes for each outcome. A similar treatment has been adopted with the other instructions and the proportions in which they are divided are given in appendix 3.

4.2.2 Results from the Analysis

Using the data in appendix 3, together with the FORTRAN program mentioned in section 4.2, values of P_J , NI_{AV} and NB_{AV} have been evaluated for the 8085, 6800, 8048 and 68000 processors. The values of these quantities are given in table 4.1. The upper curve on each of the graphs in figures 4.1 (a)-(e) show the probability that a certain number of instructions, or less, will be executed before a jump. The other curves will be explained in the following section.

4.3 Transfer from the Data Area

The analysis so far has only considered the number of instructions or bytes read before a jump. It would also be useful to know where execution will continue so that methods can be developed to generate an ordered recovery to the correct program. Consideration of the jump instructions reveals four distinct types of halts, restarts, returns and unspecified jumps. These are shown in figure 4.2 and are described in detail below.

4.3.1 Halt Instructions

Halt instructions are those which prevent further execution of any instructions until an interrupt is applied to the processor. If no provision is made to exit from this state, then no recovery is possible.

4.3.2 Restart Instructions

Restart instructions cause the processor to jump to a specified location in the memory map. The particular address varies between processors and can either be generated internally or is read from another location. In the case of the 8085, restarts jump to the low end of memory and continue to execute from that point. If no consideration for erroneous restarts have been made then values read from those locations will be interpreted as instructions.

Turner (104), in an example of a program for a security system, states that it is all right to place the code over the restart vectors if they are not being used. This would be acceptable as long as the system functions without errors and is not susceptible to external interference, however this is difficult to guarantee. If restarts do occur then execution will resume at some location within the program, but will not necessarily pick up correct instructions immediately, as shown in chapter 5.

The program in the example is short enough to finish before the end of the restart table, in particular, it does not occupy the restart 7 location. This is of special importance because the op-code for the restart instruction is FF, and it is usually the case that unused locations of ROM or EPROM are also left at FF. Therefore if such a code is erroneously executed the processor will jump to the restart location, immediately read another restart instruction and continue to loop indefinitely. This condition is similar to the execution of a halt in that no other instructions will be executed, except that a restart saves the return address on the stack. If multiple restarts occur, the stack will grow through the entire memory map destroying all the data.

On the 6800 a restart is generated by the software interrupt

instruction. It differs from the 8085 in that the address at which execution resumes is read from the top end of memory. Therefore if those particular locations have been used for some other purpose an unspecified address will be read.

The restart instructions are of great importance in returning program control to a recovery routine. In the following analysis it will be assumed that the restart vectors have been set, and that full recovery is achieved if any of the restart instructions are executed.

4.3.3 Return Instructions

If a return instruction is read, then execution will resume at the address obtained from the top of the stack. This will result in control passing back to the program provided that two conditions are met. Firstly, the last information pushed onto the stack before the fault must have been a valid program address, and secondly, both the stack pointer and the stack data must not have been corrupted by the fault or subsequent processing.

In the following sections it will be assumed that a valid program address is not read from the stack, and therefore execution continues at some undefined location, which is considered to be random in nature. This is a reasonable approach if the programming technique has been adopted where data is stored on the stack immediately after entering a subroutine. In this case the return address from the subroutine only occupies the last position on the stack for a very short time.

4.3.4 Unspecified Jumps

The last of the instructions are those which jump to a location dependent either on the contents of the bytes following the instruction or the contents of a register. In this case it is assumed that a random jump occurs.

4.4 Modification to the Analysis

Having divided the jump instructions into the four groups mentioned above, it is now possible to split the probability function, of equation 4.3, into its constituent parts corresponding to each group. The new functions will be proportional to the original probability and will depend on the relative number of each instruction type. For instance the probability of a restart $P_{RST}(K)$ is given by:-

$$P_{RST}(K) = \frac{N_{RST}}{N_J} \cdot P_J(K) \qquad \text{Eqn. 4.8}$$

Where:- N_{RST} is the effective number of restart instructions.

Similar equations can be obtained for the other three groups.

Figures 4.1 (a)-(e) show graphs for the probability function for each of the processors under investigation. Two graphs (c) and (d) are given for the 8048, one for each of the manufacturers. This is due to the dissimilar instruction sets. However, no noticeable variation can be seen in the results despite the differences.

The graphs show that the proportions of the different types of jump vary enormously between processors. Assuming that recovery is only obtained from restarts, as mentioned in section 4.3.2, the 68000 has the best response by recovering on 95% of the occasions of a random jump into a data area. This is due to the large number of undeclared op-codes which effectively generate restarts by initiating exception handling. The 8085 is the next best at 32%, followed by the 6800 at 4%. The 8048 has no restart instructions and therefore cannot recover in this manner. These figures represent the worst case, as recovery can be initiated after jumps to other parts of the memory map, and these will be considered later in chapter 7.

4.5 Improvements in Recovery

In order to increase the chances of successfully completing recovery it is necessary to initiate the recovery process as quickly as possible, so that the corruption of data is kept to a minimum. The easiest method of initiating the process is via the restarts, therefore the aim is to increase the number of jumps caused by restarts and to reduce the number of instructions executed prior to the jump.

The obvious solution is to seed the area with restart instructions, additional to those found randomly within the data. The problem is to establish the optimum number and position of the extra codes. An initial reaction could be to split the data into separate blocks so that execution can not transfer from one to another. This requires a string of adjacent single byte restart instructions equal to the length of the longest instruction. It will be shown later that this solution does not represent the best use of resources in most cases.

4.6 Simulation of Execution in Data Areas

When considering the execution in non-random data, the derivation of accurate equations to represent the response of the processor becomes very complex. An alternative approach, which was adopted, is to simulate the process on a computer. The program developed generates a block of random data which can then be modified to include certain types of instructions, such as restarts. Then, starting at a particular location, it translates the data into a sequence of instruction types, and calculates both the number of instructions and the number of bytes encountered before a jump.

The data structures considered have consisted of a certain number of random bytes separated by a given number of a particular instruction type. Execution begins randomly between the start of the first block and the

start of the second block. In each complete run the response is evaluated for a number of sequences, each one starting with a new set of data.

For a paricular sequence, the probability, $P'_J(K)$, that K instructions are executed from N_S sequences, is given by:-

$$P'_J(K) = \frac{N_K}{N_S} \qquad \text{Eqn. 4.9}$$

Where:- N_K is the number of sequences where K instructions are executed. This will give a representative result provided that N_S is large.

Initial runs were carried out with totally random data to provide a means of determining a reasonable number of sequences for each run. The value chosen was 5000, which consistently gave results within 2% of the results obtained from the original analysis, proving that both methods are consistent.

4.7 Optimum Seeding of Data

The optimum seeding of data was established by completing a number of runs on the simulator with different data structures. A selection of the results are shown in table 4.2. The percentage overhead signifies the additional memory requirement, for a particular arrangement. However, for a given overhead there are a number of ways in which the data can be seeded.

4.7.1 Data Structures for the 8085

With the 8085 and a 20% overhead the following structures were considered; 20 bytes of random data followed by 4 adjacent single byte restarts, 15 followed by 3, 10 followed by 2 and finally, 5 followed by 1. Assuming that execution of a restart generates a successful recovery, table 4.2 shows that the original suggestion of totally separating the data blocks does not give the best chance of recovery. It also shows that no

advantage is achieved by separating the blocks by more than the length of the longest instruction.

The best solution for the 8085 is to spread the seeded data, such as the Restart 7 instruction (op-code FF), evenly throughout the data area. Not only does this provide the greatest chance of recovery, but it also gives the lowest average for the number of instructions executed before a jump. One disadvantage of this arrangement is that execution is not restrained within a block. It can skip over the restart instructions and therefore there is no limit to the number of instructions which could be read.

However, the probability of execution continuing for a long time is small, and in this case a higher level of fault detection, such as a hardware watchdog timer, should provide the necessary coverage.

4.7.2 Data Structures for the 6800

The 6800 gives a totally different set of results. The optimum solution is to spread the restarts (sofware interrupt instruction code 3F) within the data area, but rather than placing them individually, they should be positioned in groups of two. The reason for this is the high number of double and triple byte instructions in the instruction set, which increases the probability of skipping over individual bytes.

4.7.3 Data Structures for the 8048

A different approach is necessary for the 8048, because the instruction set does not contain any restart type instructions. To initiate recovery it is necessary to jump to a given location which contains a recovery routine. This can be achieved using straight forward jump instructions, but requires a greater overhead, as more than one byte is needed for a given jump. The problem is to ensure that the instruction

is executed correctly, so that the address is not interpreted as an instruction.

One possible solution is to make the address equal to the op-code of the instruction. For example, the op-code 04 causes a jump to page 0 of the address map, with the low order address being read from the second byte. Therefore if execution enters a string of 04's at any point, control will always transfer to address 004. Similar effects can be obtained with the other jump instructions. An alternative method is to place one or more no-operation (NOP) instructions before the jump.

However, in both cases it is important to consider the last byte in the string. If just two bytes, such as 04, are used to separate the data blocks then the second byte can be interpreted as an instruction. This happens if either, a double byte instruction is read immediately before it, or if a direct jump to that byte occurs. This would result in a jump to an unspecified location dependent on the first byte of the next data block.

By replacing the last byte with a NOP (00), execution in this case will continue in the next data block and gives the opportunity of recovery if it reaches the end of the block. Test results have shown that this does in fact improve the probability of recovery.

The seeded data used for the results shown for the 8048 in table 4.2, where 04, 04, 00 for the triple byte strings, and 04, 00 for the double byte strings. In the first case control can pass to address 004 or 000, and in the latter case only to 000. For a single recovery address the first sequence could be changed to 00, 04, 00. Different recovery addresses can be obtained using different jump instruction codes.

Table 4.2 shows that the optimum response is obtained with the double byte strings. This is due to the large proportion of single byte non-

jumping instructions in the instruction set. Separate runs for 8048's from different manufacurers were not carried out due to the close agreement obtained from previous analyses. Instead, the data used contained the average number of particular instruction types.

4.7.4 Data Structures for the 68000

For the 68000 the level of recovery from execution in the data area is 95% without any modification to the system, apart from the addition of a recovery routine. It is unlikely that any appreciable improvement will be obtained by altering the structure of the data area. Therefore no further analysis was carried out on this processor.

4.8 The Effect of Data Block Size on Recovery

Having obtained the optimum recovery string length for each of the processors, a number of further simulations were carried out. These were designed to determine the effects on recovery, of altering the data block size. Obviously, a reduction in block size results in a greater requirement for memory, to store the extra recovery strings, and therefore has a greater overhead.

The results from these runs are given in figure 4.3. The graph shows that a large improvement in recovery is obtained with only a small increase in the data area. Further increases continue to make an improvement, but with a reduced effect.

For all three processors the greatest benefits are obtained with an increase in data area of around 20%. However, in most systems it is rare that the whole data area is used, in which case the data should be seeded with sufficient restarts to fill all the unused locations. This provides an immediate improvement without the need for any alterations to the hardware. If further improvements are required, additional memory is

necessary.

Figure 4.4 shows how the average number of instructions executed, reduces as the amount of seeded data increases. The effects on the 8048 are less than that for the other two processors because the original average is lower and the seeded data generates proportionally fewer recoveries.

4.9 Summary

This chapter has shown how erroneous execution in data areas can be detected and can then lead to recovery. All that is required is to force the processor to jump to a specific location where a recovery routine is initiated.

The 68000 microprocessor is particularly good in this respect, due to the large number of illegal and unassigned instructions which invoke exception handling. For the 8085, 6800 and 8048 it is necessary to seed the data area with certain values to improve the probability of recovery. The particular values required for each processor have been discussed, together with their optimum grouping and positioning.

The results from this analysis are used in chapter 7 where the flow of erroneous execution between different memory areas is considered.

CHAPTER 5

Erroneous Execution in Program Areas

5.1 Introduction

This chapter looks at the sequence of events following a random jump into a program area, and derives equations for the probabilities of different outcomes. Unlike the data area, the program area contains a logical sequence of instructions and therefore a different approach is necessary. Again the sequence of bytes will be dependent on the application and the method of programming. In order to analyse the general case, bytes in the program area are divided into different instruction types, and then the probabilities of different sequences of these types are studied.

The first analysis adopts a more detailed approach than the second by allowing a greater number of byte types. It therefore gives better results but has only been developed to cater for processors having single, double or triple byte instructions. However, it could be extended to include four byte instructions, such as those found on the Z80. The second analysis is less accurate but can be applied to any processor regardless of instruction length.

5.2 Detailed Analysis

When execution jumps randomly into a program area the first byte read can either be a valid op-code from the program, or it can be an operand from a multi-byte instruction. In both cases the processor will interpret the byte as an instruction and perform the corresponding operation.

Figure 5.1 shows the type of byte which can be read. Clearly, the probability of reaching each of the particular states is dependent on the type of instructions in the program. $P_R(0)$, the probability of resuming

valid instructions at the first cycle after the erroneous jump, is given by:-

$$P_R(0) = \frac{N_I}{N_B} \qquad \text{Eqn. 5.1}$$

Where:- N_I is the total number of instructions in the program.

N_B is the total number of bytes in the program area.

$P_{DX}(0)$, $P_{TXX}(0)$ and $P_{TXX}(0)$, the probabilities of entering the operand fields of double and triple byte instructions immediately after the erroneous jump, are given by:-

$$P_{DX}(0) = \frac{N_{DI}}{N_B} \qquad \text{Eqn. 5.2}$$

$$P_{TXX}(0) = P_{TXX}(0) = \frac{N_{TI}}{N_B} \qquad \text{Eqn. 5.3}$$

Where:- N_{DI} is the number of double byte instructions.

N_{TI} is the number of triple byte instructions.

It is now necessary to consider the flow of execution after each of the above states has been reached. For the case where a valid instruction has been read, the processor will continue to fetch and execute valid instructions, as it will have resynchronised instruction fetches with the program. However, this situation may not continue indefinitely if certain instructions are encountered. For example a return from subroutine instruction will cause an undefined jump if the stack pointer has been corrupted, or if the last information pushed onto the stack was data rather than a return address.

Where an operand byte is read, it could be interpreted in such a way that control is passed to another part of the memory map. If the operand byte is interpreted as a non-jumping instruction, then another byte would

be read, which again could either be a valid instruction or another operand byte. As with the analysis of execution in data areas, it is useful to know where execution continues if a jump occurs. Therefore the same approach has been adopted where the jump instructions are divided into four separate groups of halts, restarts, random jumps and returns.

The possible sequence of events after the initial jump is shown in figure 5.2. Provided that the probability of entering the operand field is less than one, execution will eventually perform a jump to another part of the memory map or resynchronise instruction fetches with the program. In order to calculate the likelihood of each of these two outcomes it is necessary to determine all the possible ways of transferring from one state to another.

This is achieved by considering all the possible sequences of bytes which allow transfer between the states. The probability that a particular sequence will occur is obtained by multiplying together the probabilities that certain types of bytes will appear in specified locations in the sequence. The overall probability of a particular transfer, from one state to another, is then obtained by adding together the probabilities that each sequence for that transfer will occur.

A list of all the possible sequences for each of the transfers, together with the derivation of the probability equations, is given in appendix 4. It shows that each value can be determined provided that the probability of certain bytes appearing in given locations is known. These can be evaluated by assuming equal use of each instruction for a particular processor and random data in the operand fields, or by analysing the occurrence of certain types of bytes in programs under investigation.

It is then possible to find expressions for the probabilities that the

processor will have resumed execution of the program or will have transferred to another part of the memory map at, I, instruction cycles after the initial erroneous jump. These expressions are also given in appendix 4. The final outcome of resuming or transferring is therefore given by the probability equations when I is equal to infinity. In most cases the probability that operand bytes are still being read after about five cycles is small, and therefore it is only necessary to consider the first ten cycles.

Again it is important to calculate the average number of instructions executed, but in this case it is necessary to calculate a value for both of the possible outcomes. NR_{AV}, the average number of instructions executed before resuming, is given by:-

$$NR_{AV} = \sum_{I=1}^{\infty} I . \frac{(P_R(I) - P_R(I-1))}{P_R(\infty)} \qquad \text{Eqn. 5.4}$$

Similar expressions can be obtained for the average number of instructions executed before the other outcomes.

Clearly the type of instructions in a particular instruction set, and the way in which the instructions are used, will affect the overall results. A comparison of different instruction sets and programs is given in the following section.

5.2.1 Comparison Between Instruction Sets

The response of execution in program areas will obviously be dependent on the arrangement and frequency of use of different instruction types. The analysis in the previous section requires a total of 24 different parameters to enable a solution. These can be obtained directly from the instruction set by assuming that each op-code is used the same number of times, and that the data in the operand field is random.

Table 5.1 contains results obtained, using the previous assumptions, for the 8085, 6800 and 8048 microprocessors. For all three processors it shows that if execution enters a program area, then there is over a 90% probability that instruction fetches will resynchronise with the program. It also indicates that the number of instructions executed before reaching one of the final states is small. The average value in all cases is less than two, implying that very few erroneous instructions will be executed and consequently little corruption of data will occur.

Only one figure has been given for the average number of instructions executed before a transfer to another part of the memory map, because each individual transfer gave almost identical results. Similarly, for the 8048, one set of results is shown, as only slight variations were observed between processors from different manufacturers.

Figures 5.3 (a), (b) and (c) show graphs of the relationship between the probability of reaching a particular outcome and the number of instructions executed, for each of the processors. They indicate the short transition period between the initial erroneous jump into the program area and the transfer to the next state. In all cases the probability of still reading an operand byte after five instruction cycles is less than 0.5%.

5.2.2 Comparison Between Actual Programs

The results from the previous section give an indication of the inherent properties of a particular instruction set. However, there are many instructions, such as the logical operators which are rarely used, and others such as the jump instructions, which are frequently used. Therefore the previous results are unlikely to be representative of actual programs using a particular instruction set.

In order to evaluate the effects of different instruction code usage,

a number of actual programs were analysed. Results from these analyses are given in table 5.2. Programs A and C are monitor programs for small scale 8085 and 6800 based systems respectively. They were chosen to give a comparison between software designed to perform similar operations but using a different instruction set. The table shows that the probability of resuming valid instructions, and the average number of instructions executed before reaching the final outcomes, are almost identical. There is a slight variation in the probabilities where control transfers to another part of the memory map, but these values are small anyway.

Programs B, D, E and F are taken from industrial control and data transmission systems. Again, close agreement is obtained for the probability of resuming valid instructions and for the average number of instructions executed. These values are also similar to those given by the monitor programs.

Therefore it seems that for erroneous execution in program areas, the probability of resychronising instruction fetches with the program is approximately 95%, regardless of the processor. This suggests that, despite differences in the instruction sets, particular instruction types tend to be used in the same proportions.

5.3 Simplified Analysis

The previous analysis is suitable for processors having single, double and triple byte instructions, and could be extended to include four byte instructions. However, to enable comparisons to be made with the 68000, which has instructions up to five words long, a more simplified approach is necessary. This is achieved by considering fewer execution states and less complex transfers. Figure 5.4 shows the different states for this analysis and the transfers between them. It shows that attempting an instruction

fetch from any of the operand fields is represented by the same state.

The probability of resuming valid instructions immediately after the erroneous jump, $P_R(0)$, remains unchanged, and clearly the probability of entering the operand field, $P_X(0)$, is given by:-

$$P_X(0) = 1 - P_R(0) \qquad \text{Eqn. 5.5}$$

Again, if a valid instruction is read, the processor will continue in step with the program. However, an instruction fetch from the operand field will either pass control to another part of the memory map or the next logical byte will be read. The probability of interpreting a jump instruction is dependent on the proportion of bytes in the operand field which will cause a jump. If the data within the field is considered to be random, then the probabilities of executing different jump instruction types can be obtained from the proportion of each particular type within the instruction set. Alternatively they can be evaluated from the analysis of particular programs under investigation.

If the next logical byte is read, this analysis assumes that the probability of reading a valid instruction is dependent on the ratio of the number of instructions in the program to the total number of bytes in the program area, which is equal to $P_R(0)$. This is effectively equivalent to random fetches within the program area until either a valid instruction is read or a jump is generated.

It follows that the probabilities that the processor has resumed or jumped at the end of I, instruction cycles after the erroneous jump, are given by:-

$$P_R(I) = P_R(I-1) + P_R(0) \cdot (1 - P_J) \cdot P_X(I-1) \qquad \text{Eqn. 5.6}$$

$$P_{RST}(I) = P_{RST}(I-1) + P_{XRST} \cdot P_X(I-1) \qquad \text{Eqn. 5.7}$$

Where:- P_J is the probability of reading any jump instruction type.

$P_{RST}(l)$ is the probability of reaching the restart state within l instruction cycles of the erroneous jump.

P_{XRST} is the probability of reading a restart type instruction in the operand field.

Similar expressions can be obtained for the other jump instruction types. The probabilities of the final outcomes of resuming or transferring, is given by the above equations when l is equal to infinity. In practice only the first ten cycles are important.

A comparison between the results from this analysis and the previous more detailed analysis is given in the following section. It shows that despite the different approach the results are in fairly close agreement.

5.4 Comparison Between the Detailed and Simplified Analyses

The simplified analysis described above was carried out on each of the programs studied in section 5.2.2, and the results from these programs are shown in table 5.3. By making a comparison with the previous values in table 5.2 it can be seen that both approaches give similar results. Therefore the simplified analysis is an acceptable approximation to erroneous execution in program areas.

The main reason for developing this approach was to enable a comparison to be made between the 8-bit processors and the 68000, which has a 16-bit architecture. To obtain a set of results for the 68000, a monitor program for a small single board system was investigated. Values for $P_X(0)$ and $P_R(0)$ were obtained by counting instructions within the software. The other parameters were estimated by assuming that the operand fields contained random data. This was necessary due to the large instruction map of 65,536 codes, which makes the determination of the effect of particular values extremely difficult.

The results obtained for the 68000 are given in table 5.3. It shows that the probability of resuming valid instruction fetches is around 20% lower than for the 8-bit processors. This difference is made up by the increase in the number of restarts, in the form of exception handling. As will be shown in section 5.6, this results in a better chance of detecting errors quickly, and improves the prospects of recovery.

The relationship between the probability of reaching a particular outcome, and the number of instructions executed, is given in figure 5.5. The graph shows that the 68000 reaches the final outcome, of execution in the program area, in approximately the same time as the other processors. This is further supported by the average number of instructions executed, which also shows close agreement.

5.5 Verification of Results

In order to check the accuracy of the results, tests were carried out on the monitor program for the 8085. From a set of random numbers, 200 addresses were selected which fell within the program area. Then starting at each of these addresses, the bytes were translated into instructions and the flow of execution, which the processor would follow, was determined. Only two possible outcomes were considered, that of resuming valid instructions and that of a transfer to another part of the memory map. The probability of resuming came to 94.1%, and that of a jump to 5.9%. Comparison between these values and those in table 5.2, obtained from the detailed analysis, show direct agreement proving that the process gives accurate results.

5.6 Improvements in Recovery

To improve the chances of recovery the processor must be able to detect that an error has occurred. This can be achieved by software in one

of two ways. Firstly, at a low level by increasing the probability that a restart will be generated, or secondly at a higher level, by encouraging execution to resynchronise with correct instructions and to detect the error from within the program. The first solution will give the quickest recovery, but as will be shown in the following sections, it is not easy to attain.

5.6.1 Low Level Detection

This can be achieved, in the same way as in the data area, by increasing the probability that a restart instruction will be interpreted. It is therefore necessary to force the restart op-codes into the operand fields within the program. The most commonly used operands are those which contain program or data addresses. These can be forced to contain particular values by the suitable positioning of memory blocks.

For example, many 8085 based systems contain RAM starting at address 2000 hexadecimal, and as a result a significant proportion of the third bytes in triple byte instructions contain the value 20. By moving the data area to the address range FF00 to FFFF these values are replaced by FF, the restart 7 instruction. A similar arrangement is possible for the 6800 by moving the data area to the address range 3F00 to 3FFF, so that more bytes of the value 3F (op-code for a software interrupt) appear in the operand fields.

This type of procedure could also be employed with the 68000. For this processor, address ranges A000 to AFFF and F000 to FFFF can be used. These are all values of unassigned op-codes which initiate exception handling if an attempt is made to execute them. This provides a much larger data area of up to 8192 bytes if both blocks are used. This method cannot be used for the 8048 because it does not have any restart type

instructions in the instruction set.

Table 5.4 shows the effect of increasing the number of restart type instructions in the operand fields of the 8085 and 6800. It contains results given by the detailed analysis for three programs, A^*, B^* and C^*, where the modifications have been made. These correspond to the original programs A, B and C in table 5.2. By comparing the values it can be seen that the number of restarts are increased quite substantially, but still do not form the major outcome from execution in this area.

A further means of increasing the number of restarts would be to move the program area as well. However in most small systems containing only one program area, this is not possible because the memory block must be positioned to coincide with the reset, restart and interrupt vectors. Also, in the case of the 6800, because it only has one restart type instruction, both the data area and the program area could not be moved to utilise this effect.

For both processors this is only suitable for data blocks up to 256 bytes long. Any larger areas would increase the number of op-codes, adjacent to the restarts, within the operand field. For both the 8085 and the 6800 this would reduce the chances of recovery by introducing more undesirable jump instruction types. Therefore, unless the data blocks can be split up into 256 byte lengths, this does not provide a means of increasing error detection which would improve the chances of recovery.

5.6.2 High Level Detection

Another way of detecting that an error has occurred is to encourage execution to resume valid instruction fetches from the program. It is then possible to test certain conditions from within the software. This would seem to be the better solution in the case of the 8085, 6800 and 8048

because there is already such a high probability that execution will resychronise with the program.

This can be further increased by the same methods described in the previous section. However, the positions to which the blocks of memory should be moved, are to those which increase the number of non-jumping instruction types within the operand fields. Ideally, single non-jumping instructions should appear in the second byte of double byte instructions and also in the third byte of triple byte instructions. Double non-jumping types should appear in the second location of triple instructions.

It would be possible to write programs such that the above conditions were met at all times, but this would impose tight restrictions on the software, by eliminating the use of certain addresses and data.

5.7 Summary

This chapter has shown that after an erroneous jump into a program area for the 8085, 6800 and 8048, execution has the probability of about 95% that it will resychronise instruction fetches with the program. Slight variations in this figure can be obtained by suitable hardware design and programming, but the most efficient method of detecting errors is from within the software. A number of these software mechanisms are described in chapter 8.

For the 68000 processor the probability of resychronisation is much lower at 72%, and the probability of a restart or exception is around 26%. Therefore it is necessary to have a recovery routine at the restart addresses and to have fault detection within the software.

The results from these analyses, together with those from the previous chapter, are used in chapter 7 where the flow of execution between different memory areas is considered.

CHAPTER 6

Erroneous Execution in Unused and Input/Output Areas

6.1 Introduction

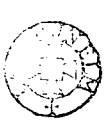
This chapter looks at the response of different processors to an erroneous jump into unused areas of the memory map, and those parts which are used for input and output devices. It then goes on to consider ways in which recovery can be initiated from these types of execution.

6.2 Execution in Unused Areas

There are two distinct types of unused locations. Those parts of the memory map which are not populated by memory devices, and those which do reference particular devices but the locations within them are not used. In the latter case it has already been demonstrated, in chapter 4, that with data areas any spare locations should be used to seed the information with restart instructions. However, with program areas, no improvement in recovery is obtained by dispersing the unused locations within the software. Therefore they appear as a single block at the end of the program area, or as smaller groups separating program modules.

In most control systems the software is written into read only memory in the form of PROM or EPROM, and consequently any unused locations are left unprogrammed, usually taking the value FF in hexadecimal. In the case of the 8085 and 68000, instruction fetches from these locations generate restarts. For the 8085 the restart 7 instruction is interpreted, and for the 68000 an unassigned instruction is encountered which initiates exception handling. Therefore recovery can be performed by a suitable error handling routine.

For the 6800 and 8048, the op-code FF is interpreted as a non-jumping instruction and therefore successive locations will be accessed until



another memory area is reached. This can be prevented by using the locations to pass control to a recovery routine, and can be achieved by adding restart or jump instructions. For the 6800 all the spare locations should be set to 3F, the code for a software interrupt. Whereas, for the 8048 the value 04 can be used to generate a jump to location 004, in the same way as that described in section 4.7.3. However the success of this method for the 8048 depends on the amount of the memory map which is used, and the state of the memory bank select flip-flop after the error. This is discussed further in section 6.2.3.

6.2.1 Unpopulated Memory Areas

Erroneous execution in unpopulated areas of the memory map is dependent on both the processor being used and the hardware attached to it. In particular it is determined by the state of the data bus when no memory devices are driving it high or low. When this state is known it is possible to establish the instructions interpreted and how execution will proceed.

This normally forces the processor to jump immediately to another location or to repeatedly read a fixed value and to continue executing the same instruction until a used block of memory is encountered. However, in the case of some processors which have a multiplexed address and data bus, the address at which an instruction fetch is attempted remains on the bus during the read cycle if there are no other external influences. This results in a series of consecutive numbers being interpreted as instructions, with appropriate adjustments made where multibyte functions are encountered. For this case it is possible to trace through the sequence of instructions which will be executed, for a particular instruction set, starting at each possible address.

It is very simple to influence the response of a processor when reading unused memory locations. In this state the data lines will tend to float, and by attaching resistors between them and the power supply rails, any value can be forced onto the bus. This can be used to generate a jump to a specific location and then to execute a recovery routine.

The following sections discuss the response of each of the processors being studied, and proposes methods of improving the chances of recovery in each case.

6.2.2 Unpopulated Areas of the 8085

The 8085 has a multiplexed address and data bus, and as a result, values read from unpopulated memory areas are dependent on the capacitance and loading of the bus. Under normal conditions of the bus being connected directly to buffers, the capacitance and loading is such that the low order byte of the address always remains valid during the subsequent read if it is not driven to any specific value.

As mentioned in the previous section, this results in consecutive values being interpreted as instructions. These values can fall anywhere in the range 00 to FF, and therefore there are 256 different positions within the sequence where execution can commence. The outome from entering at each of these locations has been determined by tracing through the sequence of instructions which the processor would interpret. For sequences where conditional jump instructions are encountered, it was assumed that the probability of a jump would be 50%. Such sequences were divided proportionally into the different outcomes which could be generated. Then the effective number of locations within the 256 byte page which cause each of the possible outcomes was calculated. These results are given in appendix 5, together with the probability of each of the

transfers.

As in the previous chapters, divisions into halts, restarts, random jumps and returns can be made, but in this case a number of specific jumps are also possible. The probabilities of each of these groups, and the average number of instructions executed before them, are given in table 6.1.

However, the specific jumps occur to locations in the range C000 to FFFF, and in most small scale control applications it is unlikely that many of these locations will be populated with memory. In this case execution will continue as before until another transfer is reached, and the effects of this arrangement are shown in table 6.2.

The results indicate that on half of the occasions, of a random jump into unpopulated areas of the 8085, a halt will be executed. If no mechanism is built into the system to recover from this situation then total failure will occur. The other outcome which has a high probability is that a return instruction will be executed. As mentioned in section 4.3.3, the address to which control passes in this case depends on the contents of the stack, and can either be a valid program address or a random location.

The results also show that it is highly probable that a large number of erroneous instructions will be executed before leaving the area, with the average for all transfers at around 40.

This type of execution can be totally eliminated by suitable loading of the bus, as indicated in section 6.2.1. By applying pull-up resistors between the data lines and the power supply rail the value FF will always be read when unpopulated memory areas are accessed. This will result in the interpretation of the restart 7 instruction, and recovery can then be

performed by a suitable routine.

6.2.3 Unpopulated Areas of the 8048

The 8048 uses a multiplexed address and data bus when accessing external memory, and consequently a similar response to the 8085 is observed when unpopulated areas of the memory map are accessed. Again the low-order byte of the address remains on the bus during the read cycle under normal buffering arrangements.

The effect of a jump into each of the 256 different possible locations in the observed sequence has been studied, and the results are given in appendix 5. They show that on 88% of the occasions, of a random jump into unpopulated areas, a transfer to specific locations occur. On 9% of the occasions, a return instruction will be executed passing control to the address stored at the top of the stack. The remaining 3% will cause a relative jump, dependent on the contents of the accumulator, to a location within the 256 byte page being accessed. This passes control back into the unpopulated area where execution will continue until another transfer is reached. When considering all transfers, the average number of instructions executed is approximately 14.

For the 8048 the response of the processor after a transfer from unpopulated memory is very dependent on the particular hardware arrangements. This is due to its architecture which is very different from normal 8-bit processors, and is described in more detail in section 3.5.

Instruction fetches are limited to a 4K address space and are referenced by a 12-bit bus. However, only 11 bits of the program counter operate in the normal way. The 12th bit is set by the state of the memory bank select flip-flop when a call to a subroutine or an absolute jump occurs, or is loaded from the stack when a return is executed. The state

of the flip-flop is only affected by the two instructions; select memory bank 0 and select memory bank 1.

This effectively splits the address range into two separate 2K blocks. Therefore the execution following a transfer from an unpopulated area is dependent on the state of the memory bank select flip-flop and the amount of memory which is used. Figure 6.1 shows four common memory arrangements for the 8048 which leave part of the memory map unused. In each of these arrangements a transfer out of the unpopulated area can result in execution reentering the same area. Due to the layout of the instruction map, this can occur a number of times, and in some cases results in the possibility of executing several hundred instructions before reaching the final state. The final states reached after a random jump into the unpopulated area for each of the four arrangements, and for both conditions of the memory bank select flip-flop, are given in table 6.3.

It shows that, under certain conditions, there is a high probability that a return instruction will be reached, in which case the address at the top of the stack, after the error, determines the location at which execution will continue. If the address is within the unpopulated area, then the process will repeat, and execution of another return instruction will probably occur. In this way the processor tends to search through the stack looking for a valid program address. However, during the erroneous execution, a number of stack locations are corrupted, and if a valid address is not found within the first few positions on the stack then an infinite loop will be formed. If a valid program address is found then execution will return to the program, but the memory bank select flip-flop may be left in the wrong state. In this case, if it is not reset before a call or an absolute jump is executed, then control will pass to the wrong

memory block.

Execution in the unpopulated areas can be controlled in some cases by suitable loading of the bus. The effect of jumping into a string of 04's has been discussed in previous sections, and this method can be employed here by forcing the value onto the bus with suitable resistors connected to the power supply rails. Unfortunately, this is only effective all of the time for memory arrangements C and D. For the other two conditions execution will loop continuously within the upper 2K block if the memory bank select flip-flop is set to one.

An alternative solution is available for memory arrangement B, by loading bits 1 to 7 on the bus with the values 1110010. This forces the values E4 and E5 alternately into the unpopulated area. The corresponding instructions interpreted by the processor are jump to address in page 7 and select memory bank zero. In this way control will always transfer to location 7E5 regardless of the position of the erroneous jump into the unpopulated area.

For memory arrangement A, there is no simple method of ensuring that control passes back to the program area. If the memory block is external to the processor, then partial decoding can be used to create another image of the program in the upper memory bank, and the solution for memory arrangements C and D will then work. Otherwise, it is necessary to ensure that a program address is always left on the stack and that the memory bank select flip-flop is reset before each call or absolute jump instruction in the program. This will ensure that on most occasions control will pass back to the program, provided that the stack and stack pointer are not corrupted by the fault. For the occasions when an infinite loop is formed it is necessary to rely on a higher level of recovery provided by external

hardware.

6.2.4 Unpopulated Areas of the 6800 and 68000

Both the 6800 and 68000 microprocessors have separate address and data buses. When accessing unpopulated areas the data bus floats high and therefore the value FF is read. For the 68000 this is interpreted as an unassigned instruction and causes the immediate initiation of exception handling, and provides a good method of recovery without any alteration to the hardware.

In the case of the 6800, the value FF is interpreted as a triple byte instruction to store the X register using extended addressing. This means that instruction fetches will occur at every third successive byte until another memory block is encountered. One method of recovery from this situation is to trap execution when it reaches the next block. However this could result in a substantial delay if a large unpopulated area exists, as the average number of instructions interpreted will be proportional to the size of the block. Also the contents of the location FFFF will be destroyed.

A better solution is to load the data bus so that the value 3F appears in all the unpopulated areas. This will immediately generate a software interrupt and enable rapid recovery without any further corruption of data.

6.3 Execution in Memory Mapped I/O

When input and output devices are mapped into the normal memory area, it is possible that an erroneous jump may occur into these locations. In the case of output lines the response will be the same as that for unused locations, as they will have no active effect on the bus. Therefore the same approach can be adopted as for the unpopulated areas, in the form of bus loading to force certain values to be read.

For input lines the external data will be interpreted as an instruction and the corresponding function will be performed. If a number of different ports appear in consecutive locations they will appear to have the same effect as a data area. However, it is common practice to partially decode port addresses so that the same data will appear in a number of consecutive locations, sometimes as much as 4K. As in the case of unused locations, this will result in an immediate jump or the repetitive execution of the same instruction. In the latter case, the average number of instructions executed will depend on the length of the instruction interpreted and the size of the input area. The particular response will change according to the state of the input lines.

It has been assumed that the state of the lines is random in nature, and from this the probabilities for the different outcomes have been calculated for particular instruction sets. The results for the 8085, 6800 and 68000 are given in table 6.4.

For the 8085, execution will on most occasions exit from the block of input data, but may take a substantial amount of time to do so if large blocks exist. Alternatively, a number of specific jumps are possible. In the previous analyses for execution in program and data areas, these codes produced random jumps because the operand fields were not dependent on the particular code. In this case the operand bytes are the same as the code, and therefore jumps to specific addresses are generated. Due to the layout of the instruction map, these cause control to transfer to particular locations in the range C000 to FFFF.

For the 6800, again the majority of cases will result in execution leaving the area. However some of these are due to relative branches backwards out of the beginning of the block. If the preceding area is

unused and the data lines have been left floating, then execution will pass back to the input area and form a continuous loop until the data on the port changes.

Execution for the 68000 will tend to continue through to the end of the area or will generate a restart.

This type of execution can be eliminated in all three processors by fully decoding the ports so that the input data only appears at single locations. In the case of the 8085 there is no need to use memory mapped input, unless more than 2048 lines are required, and therefore should be avoided if possible.

6.3.1 Execution of Input Data by the 8048

If no external memory is connected to an 8048 the bus can be used as a port, and could be used for input data. In this case the data will appear at all memory locations which were previously left unpopulated in memory arrangement A shown in figure 6.1. An erroneous jump into this area with the memory bank select flip-flop set at one, will provide no means of escape as execution will be restricted within the upper 2K memory block. With the flip-flop set at zero, the probability of forming an infinite loop will depend on which half of the memory map the erroneous jump occurs. For the upper half the value is 96.7%, whereas for the lower half it is only 12.0%.

The formation of infinite loops should be prevented if at all possible so that it is not necessary to rely on external hardware to initiate recovery. This can be achieved by avoiding the use of the bus as a port. However, if it is required it should be used for output and the same precautions taken as those discussed for unpopulated areas.

6.4 Summary

This chapter has shown that an erroneous jump into both unused and input/output areas can result in a complex sequence of execution, which can last several hundred instruction cycles or even form infinite loops. In the latter case, recovery can only be initiated by the intervention of some additional hardware.

Methods of controlling execution within these areas have been discussed, and a simple solution for most processors, of loading the data bus, has been described.

The results obtained are used in the following chapter where the flow of execution between different memory areas is considered.

CHAPTER 7

Flow of Execution Between Different Memory Areas

7.1 Introduction

The previous analyses have produced methods of determining the flow of execution within certain types of memory areas. This chapter considers the transfer of execution between these areas, to evaluate the overall response of a processor after an erroneous jump to any location within the memory map. Figure 7.1 shows the various states and transitions which will be studied. Areas of memory mapped output are included with the unused areas, as they have the same effect. Four final states are present in the model, these indicate that the processor is expected to halt operation, to enter an infinite loop, to resume executing valid instructions or to recover from the error.

7.2 Method of Analysis

This analysis uses a similar approach to that used for the erroneous execution in program areas and the equations derived are of the same form. For example, the probability of execution being in a particular memory area after a given number of transfers, between the different areas, is given by:-

$$P_{Xi}(l) = \sum_{j=1}^{4} P_{Xj}(l-1) \cdot P_{XjXi} \qquad \text{Eqn. 7.1}$$

Where:- Xi represents a particular memory area.

Xj represents each of the four different areas.

P_{XjXi} is the probability of the transfer from Xj to Xi.

l is the number of transfers after the initial error.

The probability equations for reaching each of the final states are given by:-

$$P_{Xf}(l) = P_{Xf}(l-1) + \sum_{j=1}^{4} P_{Xj}(l-1) \cdot P_{XjXf} \qquad \text{Eqn.7.2}$$

Where:- Xf represents a particular final state.

Solutions to these equations can be found for all positive integer values of l, provided that the initial conditions and the probabilities for each of the transfers is known. The methods of evaluating these quantities are given in the following three sections.

7.3 Initial Error

By assuming that the initial error causes a jump to a random location within the memory map, it follows that the probability of entering each of the different areas is proportional to the relative size of the block. In this case the size of the block includes all areas where that particular memory type appears. If certain memory devices are not fully decoded then multiple copies of the data will appear in the map and therefore it will be more likely that execution will enter that area.

The probability of entering the program area immediately after the erroneous jump, $P_P(1)$, is given by:-

$$P_P(1) = \frac{N_{PB}}{N_{TB}} \qquad \text{Eqn. 7.3}$$

Where:- N_{PB} is the total number of program bytes which appear in the memory map.

N_{TB} is the total number of bytes in the memory map.

$P_D(1)$, $P_U(1)$ and $P_I(1)$, the probabilities of entering the data areas, the unused areas and the input areas, are found in the same way.

7.4 Transfer from Different Memory Areas

Erroneous execution in different memory areas has been considered in the previous three chapters. They have shown that transfer out of each of

the areas is generated in up to five particular ways. These are; halts, restarts, unspecified jumps, returns and specific jumps. Also, in the case of program areas, execution can resychronise with the program and resume valid instruction fetches.

These transfers can be easily converted into those given in figure 7.1. Clearly, the execution of halts and resuming valid instruction fetches correspond directly and need no alteration. The restarts can have a number of different effects, dependent on the processor and the contents of particular locations.

In systems such as the 8085, where a restart causes execution to commence at a given location, it is normally the case that read only memory will be mapped to these locations. If no consideration for erroneous restarts has been included it is not uncommon for part of the program to reside in this area. Under these conditions a restart will cause a transfer into a program area and execution will continue in the manner described in chapter 5.

For other systems such as the 6800, the address at which execution continues after a restart, is read from a particular location. Again, read only memory will normally be mapped to this area, but in this case, regardless of whether program or data appears at these locations, execution will transfer to some arbitrary location within the memory map. If the particular location is considered to be random, then a transfer similar to the initial jump will occur. This is acceptable when analysing a single restart for the general case. However, any number of a particular restart in a specific system will always give the same result. This is considered further in section 7.5.

For an erroneous restart in both types of system, recovery from the

error can be achieved by the addition of a suitable recovery routine which is always executed when the restart occurs.

The unspecified jumps cause execution to transfer to arbitrary locations within the memory map, similar to the condition described above. These transfers are considered to be random in nature and therefore will have the same effect as the initial jump.

The returns cause transfers dependent on the contents of the top of the stack. In the following analysis it is assumed to be random and the corresponding transfers are the same as the initial jump. This is a reasonable assumption if the stack is used to store data as well as return addresses, or if it is corrupted by the fault.

Finally, the specific jumps always cause a fixed transfer to a particular memory area.

7.5 Execution of an Infinite Loop

The type of execution which has not been determined in the previous sections is the formation of an infinite loop. In this case the processor continually executes a fixed sequence of instructions, and no recovery is possible without some external intervention. The formation of loops in three different areas have been considered. In all cases the analysis estimates the probability of executing the same bytes twice, and if this happens it is assumed that a loop has formed. In real systems this situation will not necessarily result in a loop, because data may change in such a way that returns and conditional jump instructions will act in a different way the second time that they are executed. Therefore the analysis will tend to make an over estimate of the true value.

7.5.1 Loops in Data Areas

The first area in which a loop has been considered is the data area.

For this case, at the end of each transfer after the initial error, a calculation is made to determine the expected number of data bytes which will have been read. This is obtained from the following equation:-

$$N_{BE}(l) = \sum_{k=1}^{l} P_D(k) \cdot NB_{AV} \qquad \text{Eqn. 7.4}$$

Where:- $N_{BE}(l)$ is the expected number of data bytes read after l transfers.

NB_{AV} is the average number of bytes read during erroneous execution in data areas.

Assuming that transfers into the data area are random, the probability of entering a loop in the data area, $P_{LD}(l)$, is given by:-

$$P_{LD}(l) = P_D(l) \cdot \frac{N_{BE}(l)}{N_{DA}} \qquad \text{Eqn. 7.5}$$

N_{DA} is the actual number of data bytes in the memory map, it does not include the extra bytes which appear if partial decoding is used. This is important because without full decoding identical strings appear in more than one location, and therefore it is more likely that a loop will form with a particular string.

7.5.2 Loops in Unused Areas

Execution in unused areas follows a number of fixed sequences for a given processor and hardware arrangement. If a particular sequence is executed twice it is assumed that a loop has formed. Again this will tend to give an over estimate, for the same reasons as before. For this analysis it is necessary to evaluate the probability that execution has been in the unused area. For each state the following expression is used:-

$$P_{UXi}(l) = \sum_{j=1}^{4} P_{UXj}(l-1) \cdot P_{XjXi} \qquad \text{Eqn. 7.6}$$

Where:- $P_{UXi}^{(I)}$ is the probability of execution in a given area after execution in an unused area.

Xj corresponds to each of the memory areas.

XjXi represents the transfer from area Xj to Xi.

From this it follows that the probability of entering the unused area twice, $P_{UU}^{(I)}$, is given by:-

$$P_{UU} = \sum_{i=1}^{4} P_{UXi}^{(I)} \cdot P_{XiU} \qquad \text{Eqn. 7.7}$$

Where:- P_{XiU} is the probability of a transfer from memory area Xi to the unused area.

However, not all double entries into the unused areas will cause a loop, because in some cases a number of different sequences appear in the area. Therefore $P_{LU}^{(I)}$, the probability of forming a loop in the unused area, is given by:-

$$P_{LU}^{(I)} = P_{UUL} \cdot P_{UU}^{(I)} \qquad \text{Eqn. 7.8}$$

P_{UUL} is the probability of forming a loop after entering the unused area twice. In the following examples it is given a value equal to the proportion of sequences which cause specific transfers. This will also give an over estimate for the probability of forming a loop, as the second transfer may not be the same as the first. However the figures from the overall analysis in the following sections, using the previous assumptions, indicate that the probability of forming a loop is small. Therefore the inaccuracies in the model cannot have much of an effect on the final results.

7.5.3 Loops in Input Areas

The formation of loops in the input areas is treated in the same way as those for the unused areas, and similar expressions to equations 7.6 and

7.7 are obtained. Therefore the probability of forming a loop in the input area, $P_{LI}(I)$, is given by:-

$$P_{LI}(I) = P_{IIL} \cdot P_{II}(I) \qquad \text{Eqn. 7.9}$$

Where:- P_{II} is the probability of entering the input area twice.

P_{IIL} is the probability of forming a loop after entering the input area twice.

P_{IIL} is evaluated by considering individual arrangements of the memory map. For a single block it will take the value of 1. For multiple blocks which are separated by other memory types, a value equal to the reciprocal of the number of different blocks will give accurate results for the first reentry to the area, but will be less accurate on subsequent entries. For adjacent blocks execution can pass between them and the formation of a loop is more likely. If the probability that execution passes through the area is high, which is true for most processors, it will tend to reach the end of the last block regardless of the starting point. In this case the probability P_{IIL} tends to the value of 1.

7.6 The Expected Number of Instructions Executed

In the previous chapters the average number of instructions interpreted during erroneous execution in each of the memory areas, has been established. Now by combining these values with the probabilities of passing through the different areas, it is possible to estimate, the expected number of instructions executed, NI_E , between the original error and reaching the final outcome. In the following examples it has been obtained from:-

$$NI_E = \sum_{I=1}^{\infty} \left(\sum_{i=1}^{4} P_{Xi}(I) \cdot NI_{AVi} \right) \qquad \text{Eqn. 7.10}$$

Where:- $P_{Xi}(I)$ is the probability that execution is in the 'i'th memory

area at I states after the error.

NI_{AVi} is the average number of instructions executed for the corresponding memory area.

A more accurate result could be obtained by considering the average number of instructions executed before each of the possible transfers. These could then be combined with the probabilities of the corresponding transfers, and would give individual values for each of the outcomes. However, in most cases the averages do not vary significantly, and therefore the overall value will be a reasonable approximation. The cases where a large variation does exist are for input and unused areas where no fault tolerance has been considered.

Once the expected number of instructions has been established for a particular system, the average length of time of erroneous execution can be determined from the clock frequency. This is a very important quantity when considering watchdog designs, and is discussed further in section 7.8. It is also useful in determining the probable damage, to the data within the system, that will be caused by the execution of erroneous instructions, this is studied in section 7.9.

7.7 The Effects of Memory Map Usage on Erroneous Execution

The previous sections have built up a model for the flow of execution following an erroneous jump to a random location in the memory map. From this model a series of investigations have been carried out to study the effects of varying the amounts of different memory types. The improvements achieved by adding the fault tolerant features, described in the previous chapters, have also been studied. Clearly the results vary between processors, and they are discussed individually in the following sections.

7.7.1 Memory Maps of the 8085

The values used to obtain results for this section are taken from the analyses in the previous chapters. For each of the memory areas both fault tolerant and non-fault tolerant structures are considered. For the data area the fault tolerant case consists of single restarts separating 5 byte blocks, which is the optimum seeding with a 20% overhead. For the program area the results from the standard and modified versions of program B are used. Both the unloaded and loaded conditions of the bus are studied for the unused areas. To simplify the results, no areas of input data are considered in this section.

From this information the effects of varying the amounts of each memory area, and the addition of the fault tolerant features, have been established. When varying the size of one memory type it is inevitable that at least one other must alter in size. To overcome this problem, the size of each particular memory type was varied between 2 and 62 K bytes, while the other two filled the remainder of the space in equal proportions. The effects of adding the fault tolerant features can then be seen by comparing the results between the unmodified and the modified arrangements. These are shown in graphical form in figures 7.2, 7.3, 7.4 and 7.5.

In each case the results for the non-fault tolerant memory area include a recovery routine, so that the execution of any restart generates an ordered recovery from the error. Without the routine, the restarts in the 8085 cause execution to transfer to the low order addresses. In most cases without any fault tolerance, the program will reside in this area. Previous results have shown that around 95% of these transfers will cause a resumption of program execution. Therefore the removal of the recovery routine forces nearly all of the outcomes, which previously generated

recovery, to resume program execution. A series of tests were carried out to check this arrangement, and they showed almost identical results for the formation of loops and the execution of halts.

7.7.1.1 Fault Tolerant Program Area

Figure 7.2 shows the effects of adding fault tolerance to the program area, by forcing restart instructions into the operand fields. It shows that the probability of recovery increases with the program size, but even with a large program most of the errors will result in a resumption of program execution. Therefore it indicates that the most significant improvements can be obtained by detecting the error after execution has reentered the program.

However, this does not mean that no consideration should be given to the positioning of memory types. In most systems memory map decoding is arbitrary, and a number of different arrangements can be obtained with the same hardware, and only minor modifications to the interconnections between decoders and memory devices. Therefore, if this concept is considered at the design phase, no added cost in hardware or software design will be incurred. Also the hardware reliability will not be reduced, as there are no additional components.

The added advantage of detecting the error by the erroneous execution of a restart, is the speed of recovery, which will be initiated within a few instruction cycles. If detection is carried out within the program, a long delay is possible before reaching the checking routines, and even then they may fail to detect the error. It would then remain uncorrected until detected at a higher level, and would result in a further delay. This is particularly important in critical high speed applications where errors must be detected and corrected quickly.

7.7.1.2 Fault Tolerant Data Area

The effects of seeding the data area are shown in figure 7.3. As expected the improvements obtained increase with the size of the data area, but in all cases it is only moderate. This has to be offset against the increase in the amount of hardware necessary. In the example 20% extra memory is required which will produce a corresponding decrease in the hardware reliability of the system.

This gives a clear demonstration that adding fault tolerance for a certain class of fault can reduce the reliability in connection with another fault type, and therefore can result in an overall degredation of the full system performance. It has been suggested by Castillo et al (22) that transient failures are up to 50 times more frequent than permanent failures. This figure was obtained for medium sized computers which would normally be subjected to stable electrical and environmental conditions. For industrial control conditions it is expected that the transient error rate is much higher, and therefore the seeding of the data area may produce an overall improvement. However, other methods of recovery can be employed which are more likely to give a greater improvement. These are described in chapter 8, and require little extra hardware.

A disadvantage with these methods is the delay between the fault and the detection of the subsequent errors, as mentioned in the previous section. This is further illustrated in figure 7.5 (a), where the effect on the average number of instructions executed before reaching the final outcome, is shown for data areas with and without fault tolerance. The seeding of the data area results in fewer instructions being executed, and will give a more rapid recovery. It is therefore useful in time critical systems, particularly with large data areas.

In systems which have extra capacity within the data area, an improvement will always be made by using the spare locations to seed the area with restarts, as no additional hardware is required.

7.7.1.3 Fault Tolerant Unused Areas

The effects of loading the bus, so that restart instructions are interpreted when execution enters an unused area, are shown in figure 7.4. Once again the improvements achieved increase with the size of the area, but in this case they are quite substantial even for a small area. The only extra hardware that is required are 8 pull up resistors. These components are highly reliable when compared with integrated circuits, and will have a negligible effect on the overall hardware reliability. Also failure to open circuit by itself will not cause total system failure, it will only result in the response to an error reverting to the non-fault tolerant condition.

An additional advantage of this arrangement is the reduction in the number of erroneous instructions executed following a fault. This is shown in figure 7.5 (b). Not only does it reduce the time taken to initiate recovery, but it also reduces the probability of destroying data within the system. The advantages of adding fault tolerance to the unused areas are very significant, and therefore should be incorporated in all 8085 systems.

7.7.2 Memory Maps of the 6800

The effects of adding fault tolerant features to the 6800 are shown in figures 7.6, 7.7 and 7.8. As with the 8085, the non-fault tolerant memory areas are shown with a recovery routine. The restart on the 6800 reads the address, at which execution resumes, from the high order memory area. If the vector has not been set an arbitrary jump will occur, which is assumed to be random. The memory map is considered to be arranged with the data

area at the low order addresses, and the program area at the high order addresses. This is the normal arrangement so that non-volatile memory is resident at the restart and interrupt vectors, and so that direct addressing can be used for frequently accessed data in the zero page. With this situation a jump into the non-fault tolerant unused area results in a transfer back into the program area. Therefore most restarts, without the vector set, will cause a resumption of program execution.

Figure 7.6 shows the effects of adding fault tolerance to the program area. A similar result is obtained to the 8085 and the same conclusions can be drawn.

For the data area, fault tolerance is added in the form of two restart bytes separating 10 byte blocks of data, representing the optimum arrangement for a 20% overhead. A very different set of results are obtained, and these are shown in figure 7.7. This is due to the ratio between the number of halt and restart instructions in the 6800 instruction map. In this case the seeding of the area with restarts does have a significant effect, especially for systems with large data areas. Therefore it is more likely to produce an overall improvement in system performance despite the additional hardware required. In any case, spare bytes should be used in pairs to separate blocks of data.

For the unused area, the results are shown in figure 7.8. Again a very different response is obtained from that given by the 8085. Because of the memory layout, erroneous execution in the non-fault tolerant unused area, leads to a resumption of program execution. Therefore it might be suggested that fault detection could be carried out within the program. But as execution continues sequentially through the unused area, a very long delay could be generated. For example, the average number of

instructions executed in a 6K block will be 1024, because triple byte non-jumping instructions are interpreted. Therefore fault tolerance in the form of bus loading should be included in all 6800 systems to enable rapid recovery from erroneous execution in unused areas. As with the 8085, this has a negligible effect on the hardware reliability.

7.7.3 Memory Maps of the 68000

For the 68000 significant improvements cannot be obtained by forcing restart instructions into the operand fields, as very few erroneous instructions are interpreted during execution in program areas. As with the other processors, on most occasions it is necessary to detect erroneous execution in the program area from within the software. Very little effect is possible on the execution in data areas as very few instructions will be executed. Also, approximately 95% of the transfers out of this area will be restarts in the form of exception handling.

For the unused area it was shown, in section 6.2.4, that instruction fetches will generate restarts by reading the code for an unassigned operation. This occurs without any modifications. However in later versions of the device, the code may be assigned a function. Therefore in view of future developments, a better solution would be to force a valid restart instruction onto the bus.

The necessity of setting the restart vectors and providing a recovery routine are obvious from the discussions for the other processors. For the 68000 it is even more important because of the generation of restarts at each unused location. In the same way as the 6800 an arbitrary jump will occur if the vector is not set. If the address to which execution transfers is also unused another restart will be generated. This will repeat in an infinite loop with no means of escape, except from external

intervention from hardware. Due to the large addressing range of 16 M bytes, it is likely that only a small proportion will be used, especially for industrial control, and therefore the setting of the vectors is even more critical.

7.7.4 Memory Maps of the 8048

Memory map variations for the 8048 are fairly limited. With 12 address lines, instruction fetches are restricted to only 4 K of memory. Random access memory is mapped to separate locations and cannot be executed as instructions. However fixed data can appear in the 4 K map and may therefore be read as instructions under fault conditions.

Due to these tight constraints, very little can be done to the program area to improve error detection. Again it is necessary to carry out the checking process from within the program. Seeding of the data area was investigated in section 4.7.3, and showed that improvements were very slight due to the absence of a restart instruction in the processor. However, as the data is always known before execution, it is possible to check for any sequences which would result in undesirable execution, such as an infinite loop. If such sequences are found, the data could be rearranged to eliminate them.

The type of execution expected for different unused blocks was discussed in section 6.2.3. Bus loading was shown to be particularly important, as without it there is a high probability of forming an infinite loop for certain arrangements.

Because there is less scope for the detection and correction of errors by the processor, it is necessary to rely more heavily on an external hardware monitor, such as a watchdog timer. However, this can result in long delays before correct execution is restored, due to the time-out

period of such devices.

7.8 Number of Erroneous Instructions Executed

In the previous sections the analyses have led to a figure for the expected number of erroneous instructions executed between the initial error and reaching the final state. This gives an indication of the probable length of erroneous execution, but does not produce limits for the most likely events.

These can be achieved by assuming that the distribution of the probability, $P_{NI}(N)$, that N instructions or more will be executed, follows an exponential curve. $P_{NI}(N)$ is then given by:-

$$P_{NI}(N) = e^{-A \cdot N} \qquad \text{Eqn. 7.11}$$

Where:- A is a constant.

It can be shown that the expected value for this function is equal to the reciprocal of A. From this information it is possible to determine NI_L, the limit of the number of instructions executed for a given proportion, P_E, of the errors. These quantities are related by:-

$$NI_L = - NI_E \cdot \ln P_E \qquad \text{Eqn. 7.12}$$

Where:- NI_E is the expected number of instructions executed, determined from the previous sections.

For example from equation 7.12, NI_L takes the value 22.2 when NI_E is equal to 9.65 and P_E is equal to 0.1. This means that where the expected number of instructions executed is 9.65, 90% of the errors will result in less than 23 instructions being executed before reaching the final states. This gives close agreement with figure 4.1 (a) for execution in the data area of the 8085, which has an average number of instructions executed of 9.65. It therefore suggests that this is likely to be a reasonable approximation for the overall execution.

These limits are useful in estimating the proportion of errors which will be detected by a watchdog for various time out periods. Another use for these values is in estimating the damage to data within the system. This is discussed further in the following section.

7.9 Probability of Data Corruption

Having established a method of estimating the number of erroneous instructions executed, it is possible to determine the probable effects that this will have on the data within the system. Every instruction has the effect of changing at least one quantity, as they all alter the contents of the program counter. For the 8085, the effective number of instructions which change other quantities is shown in table 7.1, and the probability that a single instruction will not cause a corruption is also given. This assumes that the instructions interpreted are totally random in nature. For N instructions the probability, $P_{NC}(N)$, that no corruption occurs, is given by:-

$$P_{NC}(N) = P_{NC}(1)^{N} \qquad \text{Eqn. 7.13}$$

Figure 7.9 shows how the probability, that no corruption will occur to the accumulator and the B register, decreases as more instructions are executed.

From values obtained using the previous section, this leads to the estimation of the lower bounds on the probability that no corruption to a particular data element will occur. Using the previous example, of less than 23 erroneous instructions being executed, the probability of no corruption occurring to the B register in an 8085 is 33.0%. Whereas the probability of no corruption to the Accumulator is only 0.01%.

7.10 Summary

This chapter has used the information derived from the previous three

chapters, to determine the flow of erroneous execution following a jump to a random location in the memory map. The effects of varying the amounts of different memory types have been studied for a variety of processors, and the relative merits of the different methods of introducing fault tolerance to each of the areas have been established.

It has been shown for all processors that bus loading, to cause restarts in unused locations, is a very effective way of initiating rapid recovery. For example, with the 8085 the proportion of errors resulting in recovery can be increased from around 20% to over 90%. This level of improvement is obtained when only a small proportion of the memory map is used, which is the case in most small scale industrial controllers.

The positioning of memory areas, to introduce particular values into the operand fields of programs, provides improvements of less than 10% and also increases the speed of recovery. Although the benefits are small the method should be considered when designing systems, as these improvements are obtained without involving any additional costs.

Seeding data areas with restart instructions requires a substantial increase in hardware if spare capacity is not available. Not only does this increase costs but it also reduces the overall hardware reliability. Only the 6800 showed the capability of a significant improvement in recovery, and therefore it is the only processor for which it is worth considering the use of this method. However, in order to provide an overall improvement the increase in reliability due to the recovery from transient faults must be greater than the reduction in reliability due to permanent hardware failures. Therefore significant improvements can only be obtained, by this method, in systems which suffer from a high proportion of transient failures.

Finally, methods of determining the limits of the number of erroneous instructions executed, have been presented. These are used in the following chapter, where examples of adding fault tolerance to a specific system will be studied.

CHAPTER 8

Selection of Error Detection Mechanisms

8.1 Introduction

In the previous chapters it has been shown that corruption to the flow of program execution can occur in a number of ways. For this reason, methods have been proposed for the detection of erroneous execution so as to enable the early initiation of recovery processes. So far the individual methods have been considered in isolation when applied to general systems. This chapter looks at a specific system and investigates the effects of adding each of the mechanisms, to establish which ones should be adopted. In addition, some hardware mechanisms to detect erroneous execution are also discussed.

8.2 Specific System Considered

The specific system considered is a general purpose single board computer based on the 8085 microprocessor. It has been used for a number of applications within the British Gas Corporation. The system contains 4K EPROM, 2K RAM, four 8 bit input ports and four 8 bit output ports. The memory locations at which these devices can be accessed are shown in figure 8.1. The EPROM, RAM and each input port are selected as 4K blocks, therefore the RAM is mapped into two adjacent 2K blocks and each individual port can be accessed from 4096 different locations. All the output ports appear within a 4K block and are individually selected by the states of four address lines. Therefore if all four lines are active, within the 4K block, all the output ports will be selected together. This means that individual output ports can be selected from 2048 different addresses which appear in blocks of 256 locations. Pull-up resistors are connected to all the data lines so that the value FF is read from all unused locations.

8.3 The Effects of Adding Error Detection Mechanisms

A number of analyses, on the effects of adding error detection mechanisms, have been carried out based on the layout of the system described above. Results from these investigations are given in table 8.1. Some fault tolerance was considered in the design of the system and has already been incorporated. Therefore, to show the advantages of including those features, additional studies have been performed on the corresponding design without the features.

8.3.1 The Non-Fault Tolerant System

The results for the entirely non-fault tolerant arrangement are labelled 'A' in table 8.1. They show that a large number of jumps into random locations within the memory map terminate with the entering of the wait state, by the execution of a halt instruction. This is due to the property of the unused locations which tend to lead execution towards the halt instructions. Another observation made is the large number of erroneous instructions executed before reaching one of the final states, and this is due to the large portion of the memory which is mapped to very loosely decoded input ports. In most cases execution passes straight through these areas repeatedly executing the same instruction several hundred times.

From this set of basic results, the aim is to select error detection mechanisms to improve the response of the system under fault conditions. It has been shown previously that some methods can produce an overall degredation, despite an improvement with regard to an individual fault type. This is usually as a result of increased complexity which is inevitable when adding extra features. Therefore, it is clear that any additions must be both simple and effective against the considered fault.

For the error detection mechanisms studied in the previous chapters it has been shown that their effectiveness is related to the size of the particular memory block. Therefore the greatest improvements are obtained by implementing the features associated with the largest blocks. For the system considered, these consist of unused areas and input ports.

8.3.2 Removal of Input Areas from the Memory Map

The input ports take up one quarter of the memory map, and therefore will have a significant effect on the respose to a random jump. Arrangement 'B' considers the effect of removing the input ports from the memory map. Table 8.1 shows that little change occurs in the probability of reaching each of the final outcomes. However, a vast decrease, of nearly 95%, in the average number of erroneous instructions executed, is produced.

A similar response is observed in section 8.3.4 when the ports are removed while other features are present. A reduction in erroneous execution is important to limit the amount of damage which might be done during that time. It also enables rapid recovery, which is required in control situations where time is critical. It has been indicated so far that the aim is to initiate a recovery process. However, the recovery process may not be successful if too much damage is done to the data within the system. A discussion of the effects of delays in initiating a recovery routine, on the success of recovery, is presented by Preece et al (81).

Therefore the improvements obtained by removing the ports are highly desirable, and can be easily implemented in this case. The 8085 allows for separately mapped I/O by the use of the IO/$\overline{M}$ line from the processor. This can be connected directly to the enable pins on the input buffers, and does not require any other logic. Therefore, no detrimental effects to the

response to other fault types is expected, and clearly this modification should be included.

8.3.3 Addition of a Recovery Routine

In the previous arrangements discussed, no specific recovery is possible, as no provision has been made for it to occur. On approximately one quarter of the occasions, execution does resume with the interpretation of valid instructions. In a control application where all data is read in at the beginning of each cycle, the resumption of program execution will give full recovery at the start of the next cycle. However, it is normally the case that information is passed from previous calculations, and therefore a resumption of program execution will not provide acceptable recovery. This mechanism is also unsuitable in cases where a single wrong output can be harmful to the system.

In these cases it is necessary to include recovery software to generate an ordered return to correct execution. This is written most effectively as a restart routine, to enable easy access and to automatically initiate recovery when a restart instruction is erroneously executed. The effect of adding a recovery routine, which is entered by any restart instruction, is given by arrangement 'C' in table 8.1. It shows that over 15% of the final outcomes transfer from a resumption of program execution to a complete recovery. However, the full benefits are not realised until efforts are made to force erroneous execution to interpret more restarts.

The addition of a recovery routine does add to the complexity of the system. If spare capacity is not available extra memory will be required which will result in a reduction in overall hardware reliability. However, provided that the routine is small, failures resulting from its

implementation will be negligible in relation to the benefits obtained, and therefore it should be included.

8.3.4 Forcing Restart Instructions into the Unused Areas

The majority of the memory map is unused, and it was shown in chapter 6 that these locations could easily be made to appear as restart 7 instructions by the addition of pull-up resistors to the data lines. This modification is given in 'D', and represents the system as it was designed. It demonstrates the vast improvements which can be obtained by this method, lifting the proportion of errors leading to recovery to well over 90%. However, the number of instructions executed before recovery can still be very high, and this is due to the input areas in the memory map.

Arrangement 'E' shows the effect of removing the input ports from the map while retaining the other features. As before, very little change occurs in the proportion of the final outcomes, because, as indicated previously, most execution passes straight through and reaches the unused areas following these blocks. However, the number of erroneous instructions executed is reduced to single figures. In 90% of the cases less than 4 will be executed.

As indicated above, implementation of this feature is straight forward and has a negligible effect on hardware reliability, and should therefore be included in the system.

8.3.5 Modifying the Program and Data Areas

It was shown, in chapters 4 and 5, that modification to the data and program areas does not produce large improvements in the error detection process. Both these areas are relatively small, in the system being studied, and therefore little improvement is to be expected. The effects of adding methods of encouraging recovery during erroneous execution in the

program and data areas are shown in arrangements F, G, H and I. For the program area, this consists of organising the software so that more restart instructions appear in the operand fields. For the data areas, restart instructions are interspersed within the memory to limit erroneous execution. Both methods do provide further improvements, but only of the order of 1%.

The implementation of these techniques is complex with the placing of tight restrictions on the software, or with the addition of extra hardware. These both require significant development resources, and can themselves lead to design errors. The costs involved in implementation are not justified for the level of improvements that can be obtained, and therefore these techniques should not be included.

8.3.6 Detection Within the Software

The previous sections have shown that the preferred arrangement, for the specific system considered, is labelled 'E' in table 8.1. In this case recovery from erroneous execution is expected on 93% of the occasions. However, 6% of the time execution will resume with the valid interpretation of instructions. Some of these can be detected by a watchdog timer, and this is discussed below. For the other cases it is necessary to detect the errors from within the software.

If these errors are not detected, software fault tolerance against other failures, such as memory errors, may operate incorrectly. For example, the errors could cause a jump into a reasonableness test without the preceeding code being executed. If the test failed the processor would retry that particular block of code and reapply the test. It could then interpret the error as a transient and continue execution assuming that full recovery had been achieved, when in fact a higher level of recovery

was required.

The flow of execution can be monitored in a number of ways. Chudleigh (23) suggests the use of a 'relay-runner' in which a 'baton' or password is carried along with execution. This can be implemented with a single register which is incremented periodically during execution. Then at various points in the control loop the contents of the register is checked against the expected value. A discrepancy indicates that execution has not followed the correct path. This technique does not require a substantial amount of extra code. All that is required are single byte increment instructions dispersed throughout the program and a few comparisons to check the register contents.

Alternatively, the flow of execution can be monitored by checking the return addresses before leaving subroutines, or by periodically checking the current stack level. However, the use of the stack has been shown to be a possible source of errors, and can be eliminated completely while still retaining subroutines. The return address can be loaded into the HL register pair and then the PCHL instruction causes the required transfer of control. An advantage of this arrangement is that the address can be stored in multiple locations and comparisons made between the values before a transfer of control occurs.

By using the techniques proposed above, together with those from the previous sections, erroneous execution will result in the initiation of the recovery process on around 99% of the occasions.

8.4 Watchdog Timers

Watchdog timers can be used to detect a proportion of the errors resulting from erroneous execution. Some of the factors which must be considered when designing them have been indicated in previous chapters.

Their importance can now be seen from the results obtained for the system described above. Control systems are usually configured so that the timer is updated periodically; typically once during each control loop. However, tight constraints are not normally used. For example, Debelle et al (26) describe a control system for a power station boiler where the watchdog is updated once every second. If a fault occurs immediately after an update then a full second of erroneous execution could follow, and this corresponds to the execution of approximately one million instructions. Clearly a great deal of damage could occur in that time. More seriously, if a simple updating mechanism is used, such as an access to a single address, then this could occur erroneously allowing further incorrect execution.

However, the previous results have shown that, for a non-fault tolerant system, erroneous execution will only last for a few thousand instructions before a final state is reached. A watchdog will detect most cases where a loop is formed or where a wait state is entered, as the trigger is unlikely to occur at the correct interval. A watchdog is less likely to detect an error when execution resumes the interpretation of valid instructions as the trigger sequence will reappear.

With the addition of the error detection mechanisms, the watchdog is less effective as halts and loops are virtually eliminated. If the time-out period is longer than twice the time interval between updates then no errors, which resume program execution, will be detected. This is because the worst case is where a fault causes execution to jump from a point immediately before an update, to a point immediately after. By reducing the time-out period to the same length as the update time, half the errors will be detected. For the other half execution effectively jumps forward

and generates an update before the normal time. This situation can be detected by setting a minimum time. Taken further, the watchdog could be arranged to detect the update at a specific clock cycle, and could then detect any sequence of erroneous execution.

This places tight restrictions on both the hardware and the software, and would probably lead to more failures due to other failure mechanisms. Therefore the use of watchdogs for the detection of erroneous execution is ineffective if other mechanisms have been incorporated. However, it is recognised that they must be built into systems requiring high reliability to provide a level of recovery to cater for unanticipated faults.

8.5 Other Hardware Implemented Detection Mechanisms

A number of other hardware mechanisms to detect erroneous states can be used, and a selection of these, applicable to the 8085, are described below.

8.5.1 Wait State Recognition

The wait state can be detected, from the status lines, by the simple circuit shown in figure 8.2. A rising edge appears on the output as the wait state is entered. This can be connected directly to the TRAP pin on the processor, so that the interrupt routine is initiated immediately after the halt instruction has been executed. Recovery from this state would also occur with a watchdog timer, but a long delay could result.

8.5.2 Illegal Instruction Fetches

The status lines also indicate when an operation code fetch is being performed. Therefore, the circuit shown in figure 8.3 can be used to detect illegal instruction fetches outside the program area. The chip enable ($\overline{CE}$) signals, from all devices containing instructions, are ANDed together at gate 1, which produces a high output when none of the devices

are selected. This signal, together with the status lines, produces a positive going pulse on the output of gate 2 if an instruction fetch is attempted from an invalid area. This could also be connected directly to the TRAP pin on the processor.

Illegal instruction fetches from the operand fields within the program areas could be detected by the addition of an extra bit associated with each location. The bit corresponding to a valid instruction could be programmed to 0 while operands or data would be labelled with a 1. Detection of an illegal instruction fetch within the program area could then be achieved by replacing the output from gate 1 in figure 8.3 with the extra data line. By adding a pull-up resistor to the line, all illegal instruction fetches from any memory location would be detected.

This arrangement requires a substantial amount of extra hardware and would not be worthwhile in the system being studied. The hardware could be reduced by the development of 9 bit wide read only memories, as this would limit the extra logic to only a few gates.

8.5.3 Detection of a Write Outside RAM Areas

A simple development from the circuit shown in figure 8.3, allows the detection of a write into a program area, and a suitable circuit is shown in figure 8.4. It is strongly recommended that programs should be stored in read only memory for control applications, and in these cases the above circuit will be applicable. However, if it is necessary for the program to be altered during normal operation, the circuit must be modified to disable the output during loading of the program. At other times, while enabled, it will provide some protection against corruption of the code.

This concept can be extended to the detection of any writes to locations outside random access memory areas. A suitable circuit is shown

in figure 8.5, where the chip enables ($\overline{CE}$) are from all the RAM devices.

8.5.4 Detection of Undeclared or Unused Instructions

Another illegal state which can be detected is the execution of an undeclared operation code. This has been investigated by Marchal and Courtois (63) in connection with permanent stuck-at failures on the data lines. They suggest that after failure the average detection time is 11 instruction cycles for both the 6800 an 68000. This is a useful mechanism in the 68000 because it is already built into the device. However, with other processors a substantial amount of extra hardware is required, and therefore is not worthwhile. The effectiveness of this mechanism, in detecting erroneous execution after a transient fault, will be very low for the specific system studied with the fault tolerant features added. This is because very few erroneous instructions are executed.

The detection process is dependent on the number of undeclared instructions in a processor. Clearly, it will be more effective for the 6800, which has 59 undeclared codes, than for the 8085 which has only 10. However, this concept could be extended further to detect all unused operation codes within a particular program, but would require changes in the hardware when different instructions are used. Investigations into instruction usage by Lunde (60) revealed that only 75% of the codes were used, and that half of these accounted for 99% of the execution time. Therefore programming with a reduced instruction set would not be severely restrictive, and could be imposed for all programs. But even with this arrangement detection of erroneous execution will still be limited.

8.5.5 Voltage Level Detection

The hardware mechanisms described above have all been designed to detect errors after they have been produced. The voltage level detection

mechanism attempts to prevent errors occuring by suspending execution while the output from the power supply is insufficient to drive the system. This mechanism can be implemented with a single 8 pin integrated circuit. The Texas 7705 monitors the power supply rail and holds the reset line, to the processor, low while the voltage is less than 4.75 volts. When the supply rises above this value the reset is held low for an additional time interval, which is set by an RC network. This allows the internal state of the processor to stabilise before correct execution can commence.

Interruption testing, similar to that described in chapter 2, was repeated with the voltage level detection circuit added. For short interruptions, which did not cause the supply to drop below 4.66 volts, no errors were detected. For all other interruptions a full reset occurred when the supply was restored.

A disadvantage of this arrangement is that the delay in restoring execution can be relatively long. For example, a delay of 10 ms is recommended for the 8085 (119), and 50 ms is recommended for the 8035/8048 (120). Tests on the processor, described in chapter 2, showed that it could recover from an interruption which caused the supply to drop to as low as 2.5 volts. However, disruption to program execution will occur while the supply is between 2.5 and 3.8 volts, but once it has been restored, the recovery mechanisms described above can initiate the recovery process within micro-seconds. In cases where rapid recovery is required, a voltage level detection circuit should be set to activate at around 2.5 volts, and the other mechanisms can be used to recover from smaller dips in the supply. Alternatively, a second level detection circuit could be set at a higher level to initiate an interrupt routine as soon as the supply reaches the level above which no errors will occur.

8.6 Choice of Mechanisms for General Systems

The previous sections have shown that hardware mechanisms to detect erroneous execution, are not effective for the system described in section 8.2, with fault tolerance added. This is because it has a good response to erroneous execution, which is due to the large proportion of the memory map which is unused. In systems where more of the map is populated the response will be different. However, the unused areas and input areas should be looked at first before considering other parts of the map.

For systems containing a large data area, the hardware mechanism to detect instruction fetches outside the program area will be effective. For large program areas the mechanism to detect instruction fetches from the operand and data fields should be considered, but they can only produce small improvements. This is because most erroneous jumps into program areas result in an immediate resumption of the interpretation of valid instructions. Therefore detection within the software will give greater improvements than the hardware method.

This type of procedure to select detection mechanisms can generally be followed for other systems. However, the ease of implementation of some mechanisms will depend on the particular processor. For example, the detection of an illegal instruction is built into the 68000, and in order to generate recovery a suitable routine is all that is required. Conversely, the indication of an operation code fetch in the 6800 is not readily available, and therefore instruction fetches from illegal locations are difficult to detect. For single chip processors, such as the 8035/8048, there is less scope for the implementation of detection mechanisms as few signals are available externally. For these reasons mechanisms must be chosen with consideration for both the memory map usage

and the ease of implementation.

8.7 Summary

This chapter has investigated the implementation, on a specific system, of the detection mechanisms for erroneous execution, which were studied earlier. It has shown that a very high level of detection can be achieved by minor hardware changes, and with the addition of some extra software.

Other detection mechanisms have been studied, and their effectiveness for different systems has been indicated. It has been established that the choice of mechanisms, to achieve the greatest improvements in reliability, depends on both the memory map usage and the processor within the system.

CHAPTER 9

Development of a Facility to Test Redundant Systems

9.1 Introduction

This chapter presents the development of a facility to test the response of digital control systems which are subjected to a variety of transient disturbances. Testing is necessary to check the correct functioning of the error detection and recovery mechanisms. With redundant software parts of the code will not be executed under normal operating conditions. The test facility aims to simulate faults to enable all paths in the program to be executed.

Several other methods of testing were considered. For example, field trials provide accurate results but, due to the infrequent rate of failures in digital systems, they require a considerable length of time before any improvements can be established. Another important factor is that failure of the system in the field could have serious consequences, although this can be avoided by testing the system in a monitoring mode without any direct control.

To reduce the period of testing, methods can be used to increase the failure rate by subjecting the system to a hostile environment. This approach was adopted for the tests, described in chapter 2, to investigate failure mechanisms. During those tests it was established that different hardware did not always react in exactly the same way. Therefore, to obtain a representative set of results for all hardware it is necessary to test a large number of components.

A solution, which speeds up the whole procedure, is to use simulation. This approach was adopted for the Saturn V guidance computer, and is described by Ball and Hardie (5). In this case all internal functions of

the computer were simulated at the gate level, and the effects of single node stuck at 0 or 1 faults were investigated. A less detailed approach was adopted by Courtois (24) for a 6800 system. Instead of considering the gate level of the processor, a functional simulation was developed. Clearly, this reduces the amount of work required in the development of the model.

An alternative solution is to simulate faults on an actual processor. This eliminates the need for a detailed knowledge of the internal workings of the device, and prevents the introduction of errors into the simulation at this stage. A small 8085 based system was developed using this approach, and it is described in detail in the following sections. It was designed around an Intel 8085 system design kit (SDK) board, which has an additional memory card containing up to 6K RAM and 8K EPROM. Hardware modifications to the printed circuit boards were kept to a minimum to allow the system to be used for other purposes.

9.2 Fault Injection

To enable full testing it is necessary to simulate faults so that the recovery process can be observed. A number of methods of fault injection were considered. A simple solution would be to corrupt the data, address and control buses by deliberately holding individual lines high or low. A more sophisticated version could involve some logic circuitry to monitor the lines and inject faults when a certain pattern appears, or at defined time intervals.

This sort of approach has been adopted by Decouty et al (27). Their system intercepts signals before reaching individual chips in a similar way to the memory masking circuit described below. However, they are careful not to generate any short cicuits which clearly can occur in real systems.

Therefore, only a limited number of different faults are allowed, and these consist of stuck-at-0 and stuck-at-1 conditions.

An alternative arrangement is to use a second microprocessor which shares part of the memory with the main processor. It would then be able to monitor the execution of the test routines and, when predefined conditions occur, inject faults into the system. These could involve corruption of the system buses, or data stored in the shared memory. A wide range of faults could be simulated in this way, with the exception of corruption of the internal registers of the microprocessor. For these to be changed to specific values it is necessary for the processor to execute valid load instructions, and therefore cannot be achieved externally.

The dual processor approach has been used by Kuczynski and Price (54), but was limited to investigating the specific fault condition of single bit corruptions in the program code. To achieve this, the second processor copies the corrupted program into a shared memory block which emulates the EPROM of the system under test. The test system is then started and the following execution observed. Although this has given some useful results for that particular fault condition, it cannot be used to simulate other faults.

The solution which was finally adopted is much more flexible and only uses a single microprocessor. External logic circuitry generates an interrupt during execution of the test program. The interrupt routine can be written to simulate a large number of faults, and corruption of the test program, stored data and internal registers can be implemented. The timing of the interrupt is set by the control software, so both the type of fault and the position in the program, that it occurs, can be easily altered.

9.3 Generation of Interrupts

To provide thorough testing, it is desirable to inject faults in as many places as possible. Interrupts are only recognised at the completion of execution of an instruction. Therefore to inject the greatest number of faults, by this method, it is necessary to cause an interrupt during the execution of each instruction.

In order to generate interrupts at successive locations in a program, the expansion 8155 (Memory-I/O-Timer) i.c. on the SDK board is used. The timer section is designed to give an output after a certain number of pulses have been applied to its input. The number of pulses needed before triggering is programmable, and can be set by the system software. In order to be able to cause an interrupt during successive locations in the program it is necessary to generate one pulse for each instruction. This is achieved by detecting an operation code fetch which can be determined by the condition of the status lines S0, S1 and IO/$\overline{M}$. For an op-code fetch they are 1,1,0 respectively. Combining these together with logic is insufficient for the input to the timer, as this conditon remains steady throughout certain single byte instructions. For example, a string of no operations (NOPs) will produce a single pulse. By including the status of the read ($\overline{RD}$) line, which is low for only a short period of the op-code fetch, it is possible to generate a single pulse for each individual instruction.

The logic requires that the output is high when S0 and S1 are high together with IO/$\overline{M}$ and $\overline{RD}$ being low. In boolean algebra:-

$$F = A \cdot B \cdot \overline{C} \cdot \overline{D} \qquad \text{Eqn. 9.1}$$

$$= \overline{\overline{A \cdot B} + \overline{\overline{C} \cdot \overline{D}}} \qquad \text{Eqn. 9.2}$$

$$= \overline{\overline{A \cdot B} + C + D} \qquad \text{Eqn. 9.3}$$

Where:- F is the output.

A, B, C and D are the inputs.

The circuit shown in figure 9.1 satisfies the logic given by equation 9.3 by using OR, NOR and NAND gates. However, to reduce the number of devices necessary, only NOR and NAND gates were used. Figure 9.2 shows the final layout that is wired onto the SDK board. S0, S1, IO/$\overline{M}$ and $\overline{RD}$ signals are all taken from the expansion bus, and the TIMER IN signal is connected to the input of the 8155. Output from the Timer (TIMER OUT) is connected to the interrupt 7.5 (RST 7.5) pin on the 8085. This pin is also used for the Vector Interrupt (VECT INTR) key on the SDK keypad which incorporates an RC network to prevent multiple interrupts. Therefore, to ensure a quick sharp response to the timer out signal, the RC network has to be disconnected.

9.4 Memory Boundary on Test Programs

The test facility described so far is capable of providing useful results for faults involving the data of a test routine. However, if faults are injected into the program itself, causing corruption of the program counter, then control could be passed to the SDK monitor. To prevent this from occurring, additional hardware was designed to restrict the test routine to a section of memory away from the monitor. However, during execution of the control program, and during the interrupt routines, it is necessary to allow the processor to have access to all locations.

Due to the layout of the system it was not possible to restrict the test routine to half of the memory map, as this would prevent the control program from using the expansion memory board. It was therefore decided to allocate the top quarter (16K) of the map for use by the test routine, and this requires that the top two address lines (A14, A15) are held high

during execution of the routine. To satisfy the buffers and address decoders, it is necessary to control the two address lines before they reach the SDK board.

The solution adopted was to construct a small circuit raised above the SDK board. A 40 pin wire-wrap socket, plugged into the normal processor location, provides the electrical connections, and the mechanical support, for the extra circuit board. All of the lines make direct contact between the 8085 and the SDK board, except the pins associated with A14 and A15 which are diverted through the extra logic to enable some memory accesses to be restricted.

Careful consideration was needed between the timing of the control software and the masking of the address lines to ensure the correct transition between the control and test programs. This is achieved by writing to certain ports, which the extra logic circuitry detects and latches. However, the masking is not altered until the processor has read the following jump instruction.

Four transitions to and from the test routine occur for each run. Three of these, (from the control program to test routine, fault routine to test routine, and test routine back to the control program) are each catered for by the above solution. The fourth transition, caused by the fault injecting interrupt, is treated in a slightly different manner. The logic detects the interrupt acknowledge on the status lines, and waits until after the return address has been pushed onto the stack, before releasing the address lines.

Figure 9.3 shows the circuit diagram for the address masking logic. In addition to the details shown, 1Kilo-ohm pull up resistors have been connected to all the data and control signals taken from the micro-

processor, and 0.1 uF capacitors have been connected across the power supplies to most of the devices.

The circuit operates in the following manner. Writing any value to one of the ports FC, FD, FE and FF, will cause a short low level pulse at the output of i.c. 3, and presetting of flip-flop 4a will occur, forcing the $\overline{Q}$ output low. The S0 status line remains high until the end of the op-code fetch for the jump instruction, and then remains low until both address bytes have been read. During this time the output from the OR gate (5a) will have changed from a high level to a low level. The rising level of the S0 line will produce a similar rise on the output of 5a, and triggering of both flip-flops 4a and 6b will occur, clearing 4a. Flip-flop 6b has its inverted output fed back into its input, so that the output is toggled each time the device is triggered. The Q output is connected to two OR gates, 5c and 5d, these form the link between the processor and the SDK board for the two address lines A14 and A15. When the Q output from 6b is high, A14 and A15 on the SDK board remain fixed high, whereas with a low output they follow the normal outputs from the processor.

For the transition caused by the interrupt, the bottom half of the circuit is activated. After the interrupt has occurred, the status lines indicate that it has been acknowleged. A short low level pulse is generated at the output of 2a which presets the flip-flop 4b. The S1 status line goes low during the writing of the return address onto the stack. At the end of this operation a rising edge occurs at the output of 5b, triggering the flip-flop 6a and setting its $\overline{Q}$ output low. This clears 6b allowing normal addressing, and also clears both flip-flops 4b and 6a.

Provided that the logic is triggered in the correct sequence of, an output to port, an interrupt, and two more outputs to port, then the

desired masking will occur. To ensure the correct initialisation of the logic, the reset line is connected to flip-flop 4a and, through the NAND gates 7a and 7b, to flip-flop 6b. Therefore when a reset is activated on the SDK board, 4a and 6b are cleared which in turn clear both 4b and 6a.

9.5 Software Design

When designing the control software three main criteria were considered, speed of operation, ease of reprogramming and flexibility. The time taken to complete each individual test is of great importance, as injecting faults during the execution of each instruction can lead to a very large number of runs. Therefore the control software needs to be short and efficient. However, to enable quick changeover to injecting a different fault, or testing another routine, it was desirable to make reprogramming as simple as possible. These two criteria have conflicting requirements, so a compromise solution was adopted. In addition to this the overall flexibility of the system had to be considered. The aim was to avoid the necessity of rewriting most of the basic control software when new test routines or fault types are developed.

Figure 9.4 shows the final structure of the program, and fully commented listings appear in Appendix 6. Basically, the test routine is executed a number of times, injecting a fault in successive points of the program, until each location has been tested. It is reloaded into the test area before each run so that corruption of the code does not affect later tests. However, this is only representative of systems which execute programs stored in RAM. For high reliability applications the software must not be held in volatile storage to ensure that the program cannot be corrupted during erroneous execution or other disturbances. To simulate both arrangements of volatile and non-volatile program memory, an EPROM

emulator can be mapped into the test area and the write line can be connected, or not, accordingly.

In more detail, the program performs the following operations. It starts by storing the initial value of the timer trigger into memory for future use. An opening message is displayed on the terminal requesting the end address of the test routine, and the routine is then copied into the test area. The 8155 Timer i.c. is set so that it will generate the interrupt at the correct moment, and the initialisation subroutine is called to set initial values in the system. The timer is started, and the masking hardware, described above, is enabled to restrict execution to the upper 16K memory block. Control is passed to the test routine and continues until the interrupt is generated, releasing the masking circuitry. The interrupt routine sets the upper two address bits on the stack pointer, before retrieving the return address, to ensure that it is read from wtihin the upper memory area. The address is then saved as part of a jump instruction at the end of the interrupt routine. The software has been arranged so that the last two bytes are mapped into RAM, to enable the return address to be written into them, whereas the rest of the program is in EPROM.

All the internal registers are then saved, so that the fault injection routine does not affect the internal status of the processor unless this is intended. The 'fault' is then injected by calling a subroutine which changes the required data. The stack pointer and internal registers are reloaded with their original or modified values, the address mask is set, and execution returns to the test program at the point at which the interrupt occurred.

At the end of the test routine the address mask is reset and a jump is

made back into the control program. A check is made to ensure that the interrupt has occurred, and indicates whether there are still more locations to be tested. If all locations have been tried, then a closing message is sent to the terminal and execution returns to the SDK monitor. Otherwise, a subroutine is called to check the results. For a correct solution an 'S' is sent to the terminal to indicate success, alternatively, in the case of a failure, the value of the timer trigger is printed. The program continues by jumping to the start, where the trigger is incremented and the whole process is repeated.

9.6 Initial Results

Initial testing was carried out by simulating data corruptions only. The effects of these are reasonably straight forward to predict, and therefore the results from the test facility were easily verified. For example, a non-fault tolerant 8 bit addition routine was investigated. It read two numbers, from separate locations, into the internal registers, added them together and stored the answer back in the memory. As expected, corruptions to the input data in memory only caused errors if they occurred before reading the information into the registers. Conversely, corruption of the output location in memory only caused errors after the result had been stored.

This trivial case shows that the susceptibility of systems to transient memory faults can be reduced by holding critical data within the processor for as long as possible. But clearly, this will increase the susceptibility to register faults. This demonstrates the necessity to know which fault types are most common. The practical tests described in chapter 2 indicated that the memory was less resistant to interference than the processor, and therefore the registers provide a safer storage area.

Obviously, all the data cannot be stored in the registers, and consequently, an alternative approach is necessary. Hardware methods such as protective coding has been discussed, but these can fail due to multiple bit faults or transients affecting the correction mechanisms. To overcome these problems, or in the case where no memory protection is available, individual data can be stored in several locations. Clearly, this requires a large amount of extra memory space, and can only be justified for critical data.

A simple 8 bit addition routine incorporating triple storage was investigated. Even such a basic operation can be organised in several different ways. For example, the data could be compared as it is read in, and a single set chosen for manipulation by a majority vote or selection of a mid-value. The result would then be stored in memory, either in a single location or in three separate locations. Alternatively, calculations could be carried out on all three sets, and a selection made before storage. Taken one stage further, separation could be maintained throughout, and comparisons made after a number of other operations.

When considering corruptions of single locations, multiple storage of data gives large improvements in reliability. However, this must not be considered in isolation. It is possible for a large number of locations to become corrupted. This can occur as a result of an extensive memory disturbance, or by erroneous execution overwriting data. In the latter case an erroneous loop containing a call, without a return, will overwrite all volatile memory with the same 16 bit word. It is therefore suggested that if multiple copies are used, then they should not all be stored in an identical way. For example, one or more copies could be complemented. This will increase the complexity of the checking routines, but will be

more effective against extensive errors.

So far only data corruptions have been considered. Disruption to the flow of execution is possible, and this can also be tested on this system. However, a few problems are envisaged with this type of error, and suggestions for modifications are given in the following section.

9.7 Possible Developments

For data corruptions alone, execution will follow a logical sequence of instructions, provided that the software does not contain any errors. However, with the disruption in the sequence of execution, an arbitrary combination of instructions will be interpreted and as a result the test facility can fail in two ways. Firstly, the erroneous execution of an output to one of the ports FC, FD, FE and FF will cause the premature activation of the masking circuit, and an unpredictable response will follow. Secondly, the formation of a continuous loop within the test routine will prevent the return to the control program, and thus suspend any further runs.

The former case occurs infrequently as the probability of picking such an instruction at random is approximately 1 in 16,000. But if a large number of runs are attempted the failure rate may be unacceptable. It can be improved by tightening the conditions required to activate the masking circuit and could be achieved by testing for a particular value at the port. The formation of loops is more likely, however the resulting problems can be reduced by adding another hardware timer. This would be set at the beginning of each run, and if it 'timed out' before execution re-entered the control software a failure would be indicated and the next run initiated. Alternatively, the same timer as that used for fault injection could be reset before leaving the fault routine, so as to allow a

maximum time for further execution.

Finally, to obtain meaningful results, it is necessary to perform a very large number of runs. In order to simplify the analysis, it is suggested that the output, from the test facility, is captured by an intelligent device which can perform data reduction operations. This would enable the rapid evaluation of both the number and type of the failed runs.

9.8 Summary

This chapter has presented some ideas on how testing can be performed on fault tolerant software, and a particular facility has been described in detail. In this type of software, execution will pass through different segments depending on the number and type of errors in the system. Under normal operating conditions errors will be rare, and testing of all segments is not possible without fault injection. The test facility therefore provides an aid to the full functional testing of fault tolerant routines.

CHAPTER 10

Conclusions

10.1 Introduction

It is generally accepted that transient and intermittent faults are far more common, in digital circuits, than permanent faults. It has been suggested that they are as much as 50 times more likely. Therefore, in order to obtain high reliability, the greatest improvements will be achieved by designing in mechanisms to counteract the effects of transients. However, recovery cannot be initiated until errors have been detected and therefore the detection mechanisms play a very important role in the recovery process.

Investigations have been carried out into detection mechanisms with particular emphasis on software techniques. However, they cannot be evaluated until the modes of failure are understood. For this reason practical tests were performed to study actual failure modes. These attempted to reproduce the type of transient disturbances which are expected in industrial control applications.

10.2 Practical Tests to Determine Failure Mechanisms

The results of the tests showed that two broad types of failures can occur; corruption to the data within the system, and disruption to the correct flow of program execution. Both of these groups of failures occurred under different types of interference to each of the main elements of the system. The fact that similar failures occur under different operating conditions indicates that they will appear in real systems. This is true even if the types of interference, used during testing, were not representative of those which do occur in industrial controllers.

Data errors can be detected and corrected either by external hardware,

or internally by the software. Hardware mechanisms have been investigated thoroughly in the past and the majority of current systems are designed to detect and correct single bit errors. The tests did show that single bit errors do occur, but are restricted to a narrow band of interference level. In the majority of cases multiple bit errors occurred and therefore single bit correction mechanisms would not be effective.

Errors in the flow of program execution are more serious, as these result in the interpretation of an unspecified sequence of instructions. While in this state the processor cannot perform any useful tasks, and the data error correction mechanisms cannot work. Therefore it is of paramount importance to be able to detect this type of failure so that it is possible to re-establish useful execution.

In order to be able to develop suitable detection mechanisms, it is necessary to determine the sequence of events following corruption of execution. The tests indicated that a fault can cause an erroneous jump to any location in the memory map and that, subsequently, the values read would be interpreted as instructions. This revealed the importance of knowing the exact function of every possible operation code in a micro-processor.

10.3 Undeclared Operations in Microprocessors

Investigations were carried out to discover the effects of executing the codes which are undeclared by the manufacturers. In most cases useful operations were revealed, which leads to the question of why these instructions are not declared. The manufacturers were not willing to reveal information on this subject, but it is believed that some of the codes are left undeclared to retain compatibility between different devices, whereas others are not disclosed because original design errors

mean that they do not function correctly under all operating conditions.

Some of the codes are particularly undesirable from a reliability point of view. These are the ones which cause the processor to cycle continually through memory reading successive locations indefinitely. The only means of recovery from this state is a full reset which has to be generated by some external hardware. This has revealed that not only is it necessary to have external hardware to enable recovery from some errors, but also the way in which it is designed is important. For example, watchdog timers which generate interrupts, or are updated by the access to a single address, will not be effective.

Other undeclared operations of microprocessors have also been discovered, such as the cycling through memory in the 8085 as a result of power supply disturbances. These operations are particularly important because they cannot be forseen readily, unlike the functions of the undeclared codes which, clearly, must exist. Without a full knowledge of all possible operations in microprocessors it is more difficult to design effective error detection and correction mechanisms. This demonstrates the need for a much more co-operative attitude from the manufacturers in revealing full information about their devices.

10.4 Execution Following an Erroneous Jump

Having determined the functions of all the operation codes of the 8085, 6800, 8035/8048 and 68000, analyses were performed to establish the sequence of events following an erroneous jump to a random location. The execution which follows depends on the particular type of memory into which the jump occurs. Four different memory types were considered: data areas, program areas, unused areas and input areas.

Data areas were assumed to contain random values, and therefore each

operation code was equally likely to be read. It was found that execution would interpret a number of instructions before encountering a jump. The average ranged from between 2 and 10, depending on the processor.

Program areas contain a logical sequence of instructions, but an erroneous jump will not necessarily pass control directly to a valid instruction, as an operand field can be read. However, the analysis revealed that there is a high probability that a valid instruction will be read immediately, in which case the processor will continue to read valid instructions in step with the program. If an operand field is entered initially, the probability of reading a valid instruction at the next fetch is very high, and it has been shown that resynchronisation with the program tends to occur very rapidly, usually in less than three or four instruction cycles.

For unused areas the response depends on the state of the data bus when no active devices are connected to it, and this is determined by the processor and associated hardware. If the bus floats high the value FF will be read. Depending on the instruction set, this may be interpreted as a jump instruction in which case control will pass elsewhere, otherwise the next location will be accessed and the process will repeat until another memory block is encountered. For processors with a multiplexed address and data bus the address can remain valid during the subsequent read cycle. This results in the execution of a predefined sequence of instructions dependent on the instruction set and the location of the first read. For the 8085 this type of execution terminates with a halt for about one half of the initial starting points.

The data from input ports can be read as instructions if the ports are memory mapped. For a number of ports which are fully decoded into adjacent

locations they will appear to have the same properties as data areas, and can be treated in the same way. However, it is common practice to use only partial decoding, and therefore the same value will appear in adjacent locations, sometimes in as many as 4 K. With rapidly changing data on the ports a sequence of different instructions will be read, but in the analysis presented it was assumed that data remains stable for several milliseconds. In this case a jump instruction will be interpreted immediately, or the same instruction will be executed repeatedly until the end of the block is reached.

10.5 Recovery from Erroneous Execution

Having established the possible sequences of execution following an erroneous jump for a non-fault tolerant system, methods were considered which would allow recovery from erroneous execution. Clearly, the aim is to force the processor to execute a recovery routine and this can be achieved by encouraging the execution of a restart instruction.

For the data area the code for a restart can be placed at regular intervals so that they may be read as instructions if execution enters the area. However, if multibyte instructions appear before them, they are less likely to be executed. This can be overcome by grouping the restart codes together. Investigations were carried out to determine both the optimum spacing and optimum grouping to give the greatest benefits. It was established that around a 20% content of restart codes provides the best solution, but that the optimum grouping depends on the particular instruction set. In cases where there are a large number of multibyte instructions the restart codes should be grouped together in two's or three's. Although there is an increase in the probability of recovery from erroneous execution in this area, by using this method it is not

considered worthwhile. This is due to the large amount of extra hardware that is required, which will itself be prone to failure. A hardware mechanism to detect operation code fetches from data areas has been presented. It provides immediate detection using only a few simple logic gates and will therefore be much more effective.

The contents of the program areas can be influenced by the positioning of various memory blocks. For example, the addresses in a heavily accessed data area will appear in many locations in the program. Therefore by certain positioning of blocks particular op-codes can be made to appear more often. This concept was investigated, but revealed that only marginal improvements in recovery could be obtained. As before, in most cases execution resynchronises with the program. Therefore mechanisms incorporated in the software are more effective, to detect that execution has not followed the correct path.

The unused areas can be modified very simply and effectively by the addition of resistors between the data and power supply lines. This forces a single value into all locations and can be selected to be equivalent to a restart instruction, so that recovery is initiated immediately. This should be incorporated in all systems.

For the input areas the ports should be removed from the memory map, if possible, otherwise a high level of decoding should be used. This is particularly important if rapid recovery is required as large blocks of input data can lead to very long sequences of erroneous execution.

10.6 Choice of Recovery Mechanisms

The choice of a particular combination of mechanisms depends on the size of each type of memory. Generally, modification to the unused areas, and detection within the program, should be included. The addition of a

combination of these techniques ensures that erroneous execution will be detected quickly in most cases, but there will be occasions when they will fail. It is therefore necessary to provide a higher level of detection in the form of a hardware watchdog timer. It has been shown that the design of such a timer is important. For example, simple updating methods should be avoided as these may be erroneously generated under fault conditions. Interrupts must not be used to initiate recovery, as they may not function. At least a full reset must be used, and in some cases it may be necessary to power-down the system before recovery is possible.

10.7 Summary

This thesis has concentrated on error detection mechanisms, however the recovery process is equally important and requires careful consideration. It may vary from a simple reset to a thorough check-out of the entire system followed by an attempt to reconstruct all critical data that was lost.

The techniques studied provide the greatest improvements to non-redundant systems. They can also be used in redundant systems to enable the recovery of a failed unit or to recover from common mode failures. For the British Gas application of digital control, a simplex system incorporating these techniques and, perhaps, containing some additional fail-safe mechanisms, may be considered to give high enough reliability. If higher standards are required it will be necessary to adopt a redundant arrangement in the hardware. This can be achieved in a number of ways, from a tightly coupled system with voting at each clock cycle, to a very loosely coupled system maintaining separate channels from the transducers to the actuators.

The latter arrangement is preferred because it essentially consists of

several simplex channels, which will all receive the full benefits from the techniques described above. Taken individually, each channel will be easy to design and maintain, and will therefore be more readily accepted into an industry which has been concerned traditionally with mechanical controllers. The arrangement is highly immune to common mode failures, and is also very adaptable for other applications requiring different levels of reliability. It is simply a case of adding or removing modules as required.

Finally, for any application requiring high reliability, full testing of the system is essential before it undertakes active control. Some methods of testing have been presented, but these should be followed by comprehensive field trials to establish whether specified levels of reliability have been reached.

References

1 Anderson, T. and Kerr, R., 'Recovery Blocks in Action', University of Newcastle upon Tyne, Technical Report Series, No. 93, July 1976, pp. 1-11.

2 Arnold, T.F., 'The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System', IEEE Trans. Computers, Vol C-22, No. 3, March 1973, pp. 251-254.

3 Avizienis, A., 'Arithmetic Algorithms for Error-Coded Operands', IEEE Trans. Computers, Vol C-22, No. 6, June 1973, pp. 567-572.

4 Avizienis, A., 'Fault-Tolerant Systems', IEEE Trans. Computers, Vol C-25, No. 12, December 1976, pp. 1304-1312.

5 Ball, M. and Hardie, F., 'Effects and Detection of Intermittent Failures in Digital Systems', AFIPS Proc. Spring Joint Computer Conference, 1969, pp. 329-335.

6 Barigazzi, G., and Strigini, L., 'Application-Transparent Setting of Recovery Points', 13th Annual Int. FTCS, June 1983, pp. 48-55.

7 Barraclough, W., Chiang, A.C.L. and Sohi, W., 'Techniques for Testing the Microcomputer Family, Proc. IEEE, Vol 64, No. 6, June 1976, pp. 943-950.

8 Barton, S.K. et al, 'Communications Engineering Research Satellite', SERC Report, Rutherford Appleton Labs, RAL-84-016, March 1984.

9 Basu, R.N., 'Measurement of Small Signals in a Noisy Environment', IEE Conference on Electrical Interference in Instrumentation, 1970, pp. 109-114.

10 Bell, E.M., Kwiatkowski, C. and Ross, C.E.J., 'Computer Aids for Reliability Prediction and Spares Provisioning', Electrical Communication, Vol 54, No. 2, 1979, pp. 136-142.

11 Bland, G.M.S., Bradbury, K.J. and Smith T.D., 'Distributed Computer Control of a Large Coal Fired Generating Unit:- A Design Study', IEE Conference on Distributed Computer Control, November 1977, pp. 1-12.

12 Bologna, S. et al., 'A Computerized Protection System for a Fast Research Reactor', IEEE Trans. Nuclear Science, Vol NS-27, No. 1, February 1980, pp. 803-807.

13 Boney, J., 'Let Your Next Microcomputer Check Itself and Cut Down Your Testing Overhead', Electronic Design, September 1979, pp. 100-105.

14 Boothman, G., 'Designing Business Machine Cabinets for Optimal EMI Shielding', Wescon '82 Conference Record, Anaheim, USA, September 1982, pp. 11-1/1-5.

15 Bouricius, W.G., Carter, W.C., Jessep, D.C., Schneider, P.R. and Wadia, A.B., 'Reliability Modelling for Fault Tolerant Computers', IEEE Trans. Computers, Vol C-20, No. 11, November 1971, pp. 1306-1311.

16 Brodsky, M., 'Hardening RAMs Against Soft Errors', Electronics, April 1980, pp. 117-122.

17 Buchholz, S., 'Besitzt der Mikroprozessor Intel 8085 ein Overflow-Flag?', fernmelde-praxis, Vol 58, June 1981, pp. 428-436.

18 Bull, J.H., 'Interference to Instumentation due to Transients in the Supply System', IEE Conf. Electrical Interference on Instrumentation, 1970, pp. 94-100.

19 Bumby, E.A., 'Redundancy Management for Fly-by-Wire Systems', AIAA Guidance and Control Conference, Paper 72-884, 1972, pp. 1-5.

20 Burrow, L.D., 'The Fail Soft Design of Complex Systems', IEE Conf. on Distributed Computer Control, November 1977, pp. 151-156.

21 Carter, W.C. and Bouricius, W.G., 'A Survey of Fault Tolerant Computer Architecture and its Evaluation', Computer, Vol 1, January 1971, pp. 9-16.

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

23 Chudleigh, M., 'Software Must be Tolerant Too', Computer Systems, December 1982, pp. 43-45.

24 Courtois, B., 'Some Results About the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunctions', 9th Annual International FTCS, June 1979, pp. 71-74.

25 De, B.B. and Krarau, H.B., 'Fault-Tolerance in a Multiprocessor, Digital Switching System', IEEE Trans. Reliability, Vol R-30, No. 3, August 1981, pp. 246-252.

26 Debelle, J. et al, 'First Belgian Application of a Digital Computer for the Control of a 280 MW Boiler of the Thermal Power Station at Genk-Langerlo', Digital Computer Applications to Process Control, 1977, pp. 769-788.

27 Decouty, B., Michel, G. and Wagner, C., 'An Evaluation Tool of Fault Detection Mechanisms Efficiency', 10th Annual International FTCS, October 1980, pp. 225-227.

28 Dehnhardt, W. and Sorensen, V.M., 'Unspecified 8085 Op-Codes Enhance Programming', Electronics, 18th January 1979, pp. 144-145.

29 Dellacorna, L., Morganti, M. and Novielli, G., 'A Micro-processor Based Control Unit for High Availability Applications', 10th Annual Int. FTCS, 1980, pp. 357-362.

30 Dick, I.J., 'Low Frequency Electrical Interference in Process Control Computing', IEE Conference on Electrical Interference in Instrumentation, 1970, pp. 74-80.

31 Doyle, E.A. Jr., 'How Parts Fail', IEEE Spectrum, Vol 18, No. 10, October 1981, pp. 36-43.

32 Dunn, R.H. and Ullman, R.S., 'A Workable Software Quality/Reliability Plan', Proc. Annual Reliability and Maintainability Symposium, 1978, pp. 210-217.

33 Dyer, G., 'Protecting Military Systems and Equipment from EMP', Communications International, April 1982.

34 Elkland, S.A. and Siewiorek, D.P., 'Reliability and Performance of Error-Correcting Memory and Register Arrays', IEEE Trans. Computers, Vol C-29, No. 10, October 1980, pp. 920-927.

35 Emfinger, J., and Flannigan, J., 'Fly by Wire Technology', AIAA Guidance and Control Conference, Paper 72-882, 1972, pp. 1-6.

36 Gelderloos, H.C. and Wilson D.V., 'Redundancy Management of Shuttle Flight Control Sensors', Proc. IEEE Conf. on Decision and Control, 1976, pp. 462-475.

37 Goldberg, J., 'New Problems in Fault-Tolerant Computing', 5th Annual International FTCS, 1975, pp. 29-34.

38 Greenspan, S.J. and McGowan, C.L., 'Structuring Software Development for Reliability', Microelectronics and Reliability, Vol 17, 1978, pp. 75-84.

39 Gunther, N.L. and Carter, W.C., 'Remarks on the Probability of Detecting Faults', 10th Annual Int. FTCS, 1980, pp. 213-215.

40 Hamill, T.G. and Phillips, R., 'A Fault Tolerant Reconfigurable Multiprocessor System', IEE Conference on Distributed Computer Control, November 1977, pp. 139-144.

41 Hamming, R.W., 'Error Detecting and Error Correcting Codes', Bell Systems Technical Journal, Vol 29, 1960, pp. 147-160.

42 Hart, A., Teng, T. and McKenna, A., 'Reliability Influences from Electrical Overstress on LSI Devices', 18th Annual Proc. of Reliability Physics, April 1980, pp. 190-196.

43 Hayes, J.P. and McCluskey, E.J., 'Testability Considerations in Microprocessor-Based Design', IEEE Computer, Vol 13, No. 3, March 1980, pp. 17-26.

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

45 Heftman, E., 'Growing Concern over Memory Soft Errors Prompts Intense Alpha-Particle Research', Electronic Design, April 1979, p. 27.

46 Hnatek, E.R, Graves, W. and Schmitt, R.G., 'How Static is the Static 4K RAM?', IEEE Semiconductor Test Symposium, 1976, pp. 3-8.

47 Hnatek, E.R., 'Microprocessor Device Reliability', Microprocessors, Vol 1, No. 5, June 1977, pp. 299-303.

48 Hopkins, A.L. Jr., 'A Fault Tolerant Information Processing Concept for Space Vehicles', IEEE Trans. Computers, Vol C-20, November 1971, pp. 1394-1403.

49 Jack, L.A., Kinney, L.L. and Berg, R.O., 'Comparison of Alternative Self Check Techniques in Semiconductor Memories', 7th Annual International FTCS, 1977, pp. 170-174.

50 Johnson, J.N. and Shaw, J.L., 'System Malfunction Detection & Correction Studies Software for a Fault Tolerant Computer - Dual Processor with Monitor', Boeing Company, Document Number D180-19249-2, July 1976, pp. 1-21.

51 Kim, W.S. et al, 'Radiation-Hard Design Principles Utilised in CMOS 8085 Microprocessor Family', IEEE Trans. Nuclear Science, Vol NS-30, No. 6, December 1983, pp. 4229-4234.

52 Kodandapani, K.L. and Pradhan, D.K., 'Undetectability of Bridging Faults and Validity of Stuck-At Fault Test Sets', IEEE Trans. Computers, Vol C-29, No. 1, January 1980, pp. 55-59.

53 Kopetz, H., 'Software Reliability', Macmillan Press, 1979.

54 Kuczynski, M. and Price, B.L., 'EPROM Evaluation: A Technique for the Software Evaluation of Microprocessor Based Burner Controllers', British Gas Internal Report, July 1982. (British Gas reports are not normally available to other organisations).

55 Kurzhals, P.R. and Deloach, R., 'Integrity in Flight Control Systems', Proc. Joint Automatic Control Conference, Vol 1, 1977, pp. 489-497.

56 Lee, P.E., Ghani, N. and Heron, K., 'A Recovery Cache for the PDP-11', IEEE Trans. Computers, Vol C-29, No. 6, June 1980, pp. 546-549.

57 Levine, L. and Meyers, W., 'Semiconductor Memory Reliability with Error Detecting and Correcting Codes', IEEE Computer, Vol 9, No. 10, October 1976, pp. 43-50.

58 Lonn, W.M., Moore, G.H. and Speckman, B.M., 'Operating Experience with Dual DDC Computer System Pittsburg Power Plant Unit No.7', Proc. 16th Int. I.S.A. Power Instrumentation Symposium, Vol 16 A73, 1973, pp. 75-85.

59 Losq, J., 'Influence of Fault-Detection and Switching Mechanisms on the Reliability of Stand-by Systems', 5th Annual International FTCS, 1975, pp. 81-86.

60 Lunde, A., 'Emperical Evaluation of Some Features of Instruction Set Processor Architectures', Communications of the ACM, Vol 20, No. 3, March 1977, pp. 143-153.

61 McConnel, S.R., Siewiorek, D.P. and Tsao, M.M., 'The Measurment and Analysis of Transient Errors in Digital Computer Systems', IEEE Ch1396-1/79/0000-0067$00.75, 1979, pp. 67-70.

62 McKinney, H.N. and Briggs, D.C., 'Electrical Power Subsystem for the NATO III Communications Satallite', 11th Int. Energy Conversion Conference, 1976, paper 769242, pp. 1408-1413.

63 Marchal, P. and Courtois, B., 'On Detecting the Hardware Failures Disrupting Programs in Microprocessors', 12th Annual International FTCS, 1982, pp. 249-256.

64 May, T.C. and Woods, M.H., 'A New Physical Mechanism for Soft Errors in Dynamic Memories', Proc. Int. Reliability Physics Symposium, April 1978, pp. 33-40.

65 Musa, J.D., 'Measuring and Managing Software Reliability', IEEE Proceeding of the 2nd Annual Conference on Computers and Communications, March 1983, pp. 105-109.

66 Nelson, E.C., 'Software Reliability', 5th Annual International FTCS, 1975, pp. 24-28.

67 Nemmour, M., 'Etude du Fonctionnement Interne des Microprocesseurs 6800', ENSIMAG, Final Report (NM4), Grenoble, May 1979.

68 Nemmour, M., 'Etude du Fonctionnement du Microprocesseur MC 6809', ENSIMAG, Contract EDFI25IA2739, Grenoble.

69 Nichols, N., '8080/8080A Microcomputer', Intel Reliability Report, RR-10, March 1976, pp. 1-10.

70 Ng, Y.W. and Avizienis, A., 'A Model for Transient and Permanent Fault Recovery in Closed Fault Tolerant Systems', 6th Annual International FTCS, 1976, pp. 182-188.

71 Ng, Y.W. and Avizienis, A., 'ARIES - An Automated Reliability Estimation System for Redundant Digital Structures', Proc. 1977 Annual Reliability and Maintainability Symposium, USA, January 1977, pp. 108-113.

72 O'Brien, F.J., 'Rollback Point Insertion Strategies', 6th Annual International FTCS, 1976, pp. 138-142.

73 Obac-Roda, V. and Davies, O.J., 'Aspects of Fault Tolerant Ring Structures', IEE Colloquim, London, Digest No. 1982/67, October 1982, pp. 3/1-9.

74 Oppenheimer, C.P., 'Reliable Designs Begin with the Basics', Computer Design, August 1983, pp. 93-99.

75 Pappu, R.V., Harris, E. and Yates, M., 'Screening Methods and Experience with MOS Memory', Microelectronics and Reliability, Vol 17, No. 1, 1978, pp. 193-200.

76 Pearson, J.C., 'Reliability of Small Digital Controllers', PhD Thesis, University of Durham, 1983.

77 Pearson, J.C., Halse, R.G. and Preece, C., 'Reliable Digital Controller Architecture for Gas Distribution Regulators', Reliability Engineering, Vol 8, 1984, pp. 179-189.

78 Peckett, D., 'Fault-Finding with Aid of Self-Test Programs', Practical Computing, December 1979, pp. 102-107.

79 Pellegrini, G., Raimo, A. and Reynaud. C., 'EMC Problems in H.V. Sub-Stations', IEEE International Symposium on Electromagnetic Compatability, 1976, pp. 106-109.

80 Preece, C. and Stewart T.R., 'Multilevel Fault Recovery in Real-Time Digital Controllers', IEE Colloquim, London, May 1979.

81 Preece, C, Pearson, J.C. and Halse, R.G., 'The Introduction of Fault Tolerance into Digitally Controlled Gas Regulators', Int. Gas Research Conference, London, June 1983.

82 Pytches, D., 'Zinc-Air Cells: Power Source of the Future', Electronics and Power, Vol 29, No. 7/8, July 1983, pp. 577-580.

83 Randell, B., 'System Structure for Sofware Fault Tolerance', IEEE Trans. Software Engineering, Vol SE-1, No. 2, June 1975, pp. 220-232.

84 Reese, S.E., 'Low Point Control System Reduces Gas Losses', Pipeline and Gas Journal, July 1975, pp. 42-46.

85 Robach, C., Saucier, G, and Lebrun, J., 'Processor Testability and Design Consequences', IEEE Trans. Computers, Vol C-25, No. 6, June 1976, pp. 645-652.

86 Rostek, P.M., 'Techniques of Shielding and Filtering Digital Computers for EMI Susceptibility', IEEE Electromagnetic Compatibility Symposium Record, San Antonio, USA, October 1975, Session 4B, pp. e1-7.

87 Russell, P.J., 'Non-Commercial Non-Stop Processing', Computer Systems, December 1982, pp. 47-49.

88 Sedmak, R.M. and Liebergot, H.L., 'Fault-Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration', 8th Annual International FTCS, June 1978, pp. 137-143.

89 Sequin, C.H., 'Instruction in MOS LSI Systems Design', IEEE Computer, March 1980, pp. 67-73.

90 Sexton, F.W. et al, 'Radiation Testing of the CMOS 8085 Microprocessor Family', IEEE Trans. Nuclear Science, Vol NS-30, No. 6, December 1983, pp. 4235-4239.

91 Shedletsky, J.J. and McClusky, E.J., 'The Error Latency of a Fault in a Sequential Digital Circuit', IEEE Trans. Computers, Vol C-25, No. 6, June 1976, pp. 655-659.

92 Shooman, M.L., 'The Spectre of Software Reliability and its Exorcism', Proceedings of the Joint Automatic Control Conference, 1977, pp. 225-231.

93 Siewiorek, D.P., 'Transparency in Distributed, Fault Tolerant Computing Systems', 14th International Computer Society Conference, pp. 276-278.

94 Siewiorek, D.P., Kini, V., Joobbani, R. and Bellis, H., 'A Case Study of C.mmp, Cm* and C.vmp: Part II - Predicting and Calibrating Reliability of Multiprocessor Systems', Proc. of IEEE, Vol 66, No. 10, October 1978, pp. 1200-1220.

95 Smith, A.L., 'Hard and Soft Failures in Dynamic RAM Fault Tolerant Memories', IEEE Trans. Reliability, Vol R-30, No. 1, April 1981, pp. 58-60.

96 Smith, D.H., 'Microprocessor Testing - Method or Madness', IEEE Semiconductor Test Symposium, 1976, pp. 27-29.

97 Soi, I.M. and Gopal, K., 'Some Aspects of Reliable Software Packages', Microelectronic Reliability, Vol 19, 1979, pp. 379-386.

98 Spearman, C.A., 'Improved District Pressure Control', The Institution of Gas Engineers, 8th March, 1977.

99 Sulway, B., 'AC Emergency and Uninterruptible Power Supplies', Communications International, Vol 3, December 1976, pp.62-65.

100 Tasar, V., 'Analysis of Fault Detection Coverage of a Self-Test Software Program', 8th Annual Int. FTCS, June 1978, pp. 65-71.

101 Teets, R.M., 'Protecting Minicomputers from Power Line Perturbations', Computer Design, Vol 15, June 1976, pp. 99-104.

102 Thatte, S.M. and Abraham, J.A., 'Testing of Semiconductor Random Access Memories', 7th Annual Int. FTCS, 1977, pp. 81-87.

103 Toschi, E.A. and Watanabe, T., 'An All-Semiconductor Memory with Fault Detection, Correction, and Logging', HP Journal, pp. 8-13.

104 Turner, R.C., 'Real-Time Programming with Microcomputers', Lexington Books, Toronto, 1980.

105 Von Neumann, J., 'Probabilistics, Logistics and the Synthesis of Reliable Organisms from Unreliable Components', Automata Studies from Annals of Mathematical Studies No. 34, Princeton University Press, 1956, pp. 43-99.

106 Wachter, W.J., 'System Malfunction Detection and Correction', 5th Annual International FTCS, 1975, pp. 196-201.

107 Wakerly, J.F., 'Microcomputer Reliability Improvement Using Triple-Modular Redundancy', Proc. IEEE, Vol 64, No. 6, June 1976, pp. 889-895.

108 Walker, W.K.S., Sundberg, C.W. and Black, C.J., 'A Reliable Spaceborne Memory with a Single Error and Erasure Correction Scheme', IEEE Trans. Computers, Vol C-28, No. 7, July 1979, pp. 493-500.

109 Wei, A.Y., 'Real Time Programming with Fault Tolerance', PhD Thesis, University of Illinois, USA, 1981.

110 Wensley, J.H., 'SIFT - Software Implemented Fault Tolerance', AFIPS Conference Proceedings, Vol 41, Part 1, 1972, pp. 243-253.

111 Wensley, J.H. and Levitt, K.N., 'A Comparative Study of Architectures for Fault-Tolerance', 4th Annual International FTCS, June 1974, pp. 4-(16-21).

112 Westermeier, T.F., 'Redundancy Management of Digital Fly-by-Wire Systems', Proc. Joint Automatic Control Conference, Vol 1, 1977, pp. 272-277.

113 Whallen, J.J., Tront, J., Larson, C.E. and Roe, J.M., 'Compter-Aided Analysis of RFI Effects in Integrated Circuits', IEEE Electro-magnetic Compatibility Symposium, June 1978, pp. 64-70.

114 Williamson, I., 'Design of Self-Checking and Fault-Tolerant Micro-programmed Controllers', IEEE Conf. Computer Systems and Technology, 1977, pp. 193-204.

115 Williamson, T., 'Designing Microcontroller Systems for Electrically Noisy Environments', Intel Corporation, Application Note AP-125, February 1982.

116 Wulf, W.A., 'Reliable Hardware/Software Architecture', IEEE Trans. Software Engineering, Vol SE-1, No. 2, June 1975, pp. 233-240.

117 Ziegler, J.F. and Lanford, J.F., 'Effect of Cosmic Rays on Computer Memories', Science, Vol 206, November 1979, pp. 776-788.

118 -----, 'The 8080As Are Not All Alike; You Should Know the Differences', Electronic Design, 18th January 1977, pp. 41-42.

119 -----, 'MCS-80/85 Family User's Manual', Intel Corporation, 1979.

120 -----, '48-Series Microcomputers Handbook', National Semicinductor Corporation, 1980.

121 -----, 'MIL-HDBK-217D Reliability Prediction of Electronic Equipment', U.S. Department of Defense, January 1982.

122 -----, 'HRD3 Handbook of Reliability Data', British Telecom, Materials and Components Centre, Birmingham, January 1984.

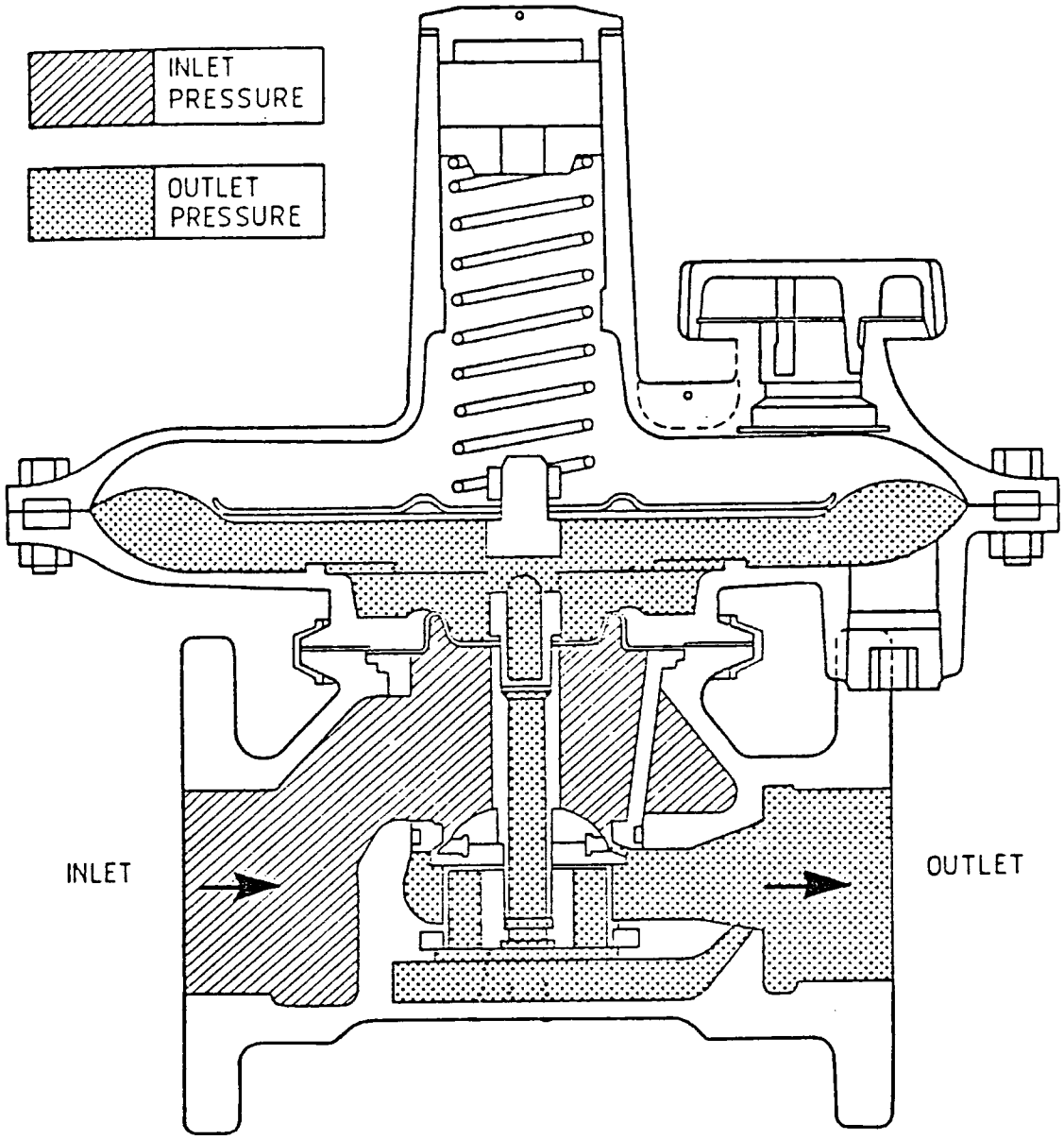


Figure 1.1 Typical Diaphragm Operated Regulator

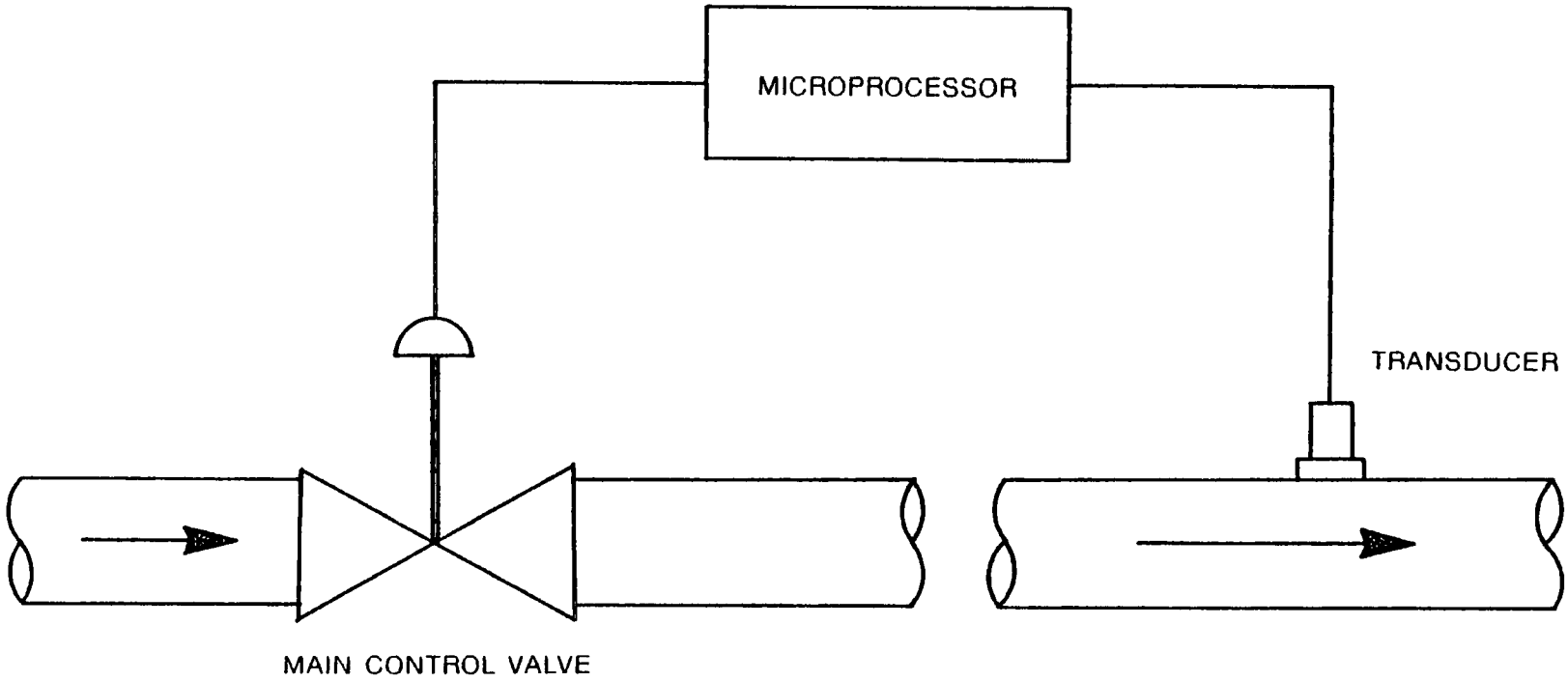


Figure 1.2 Simple Microprocessor Control Arrangement

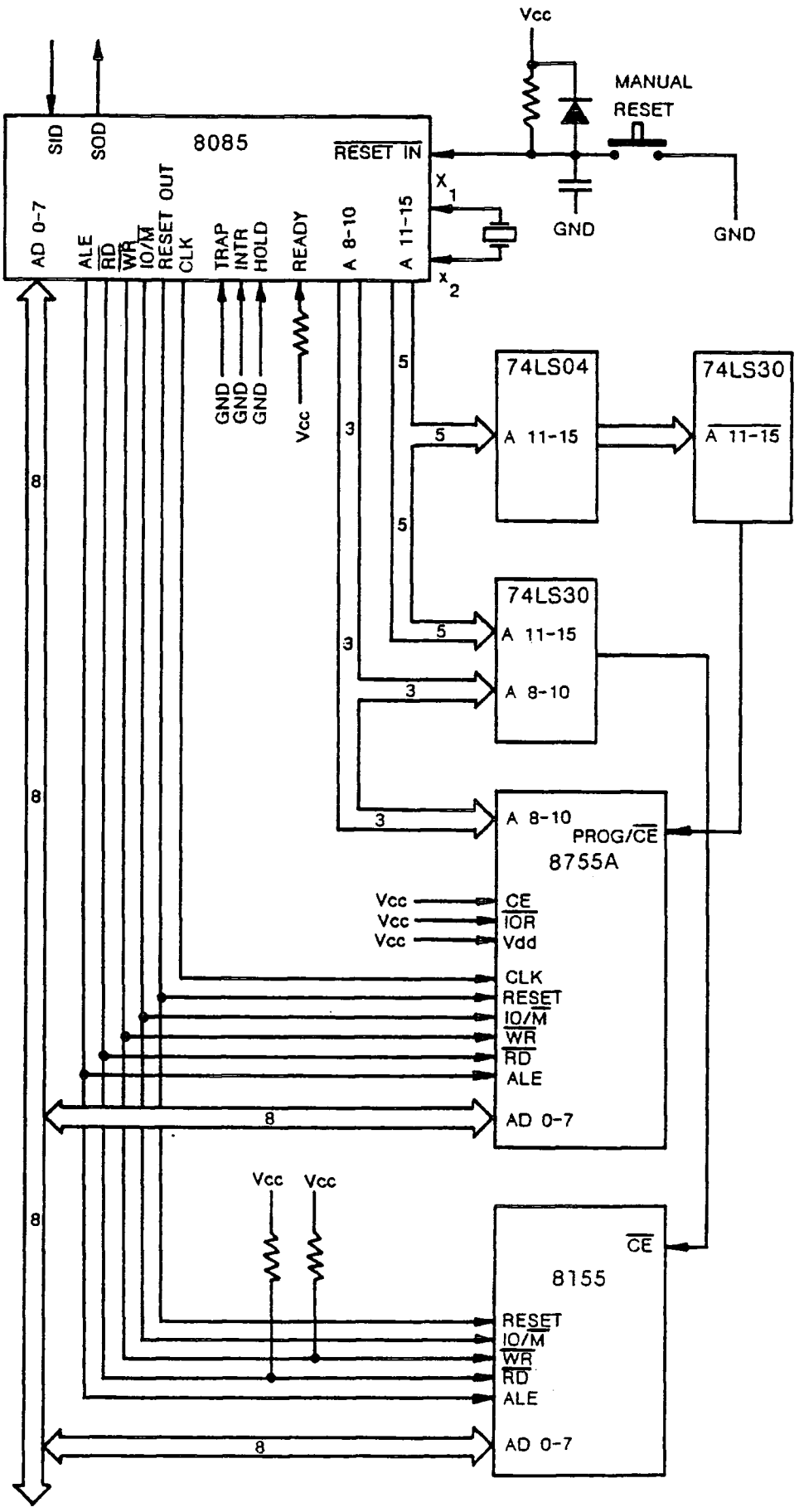


Figure 2.1 Block Diagram of the 8085 Test System

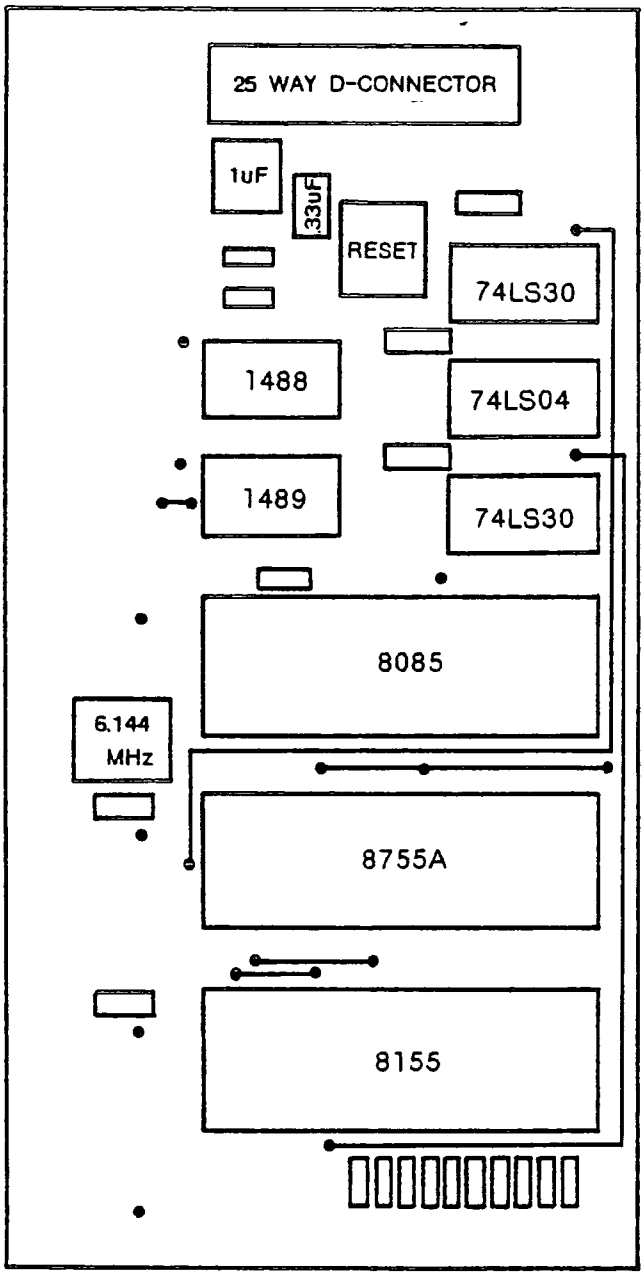


Figure 2.2 Layout of the Components

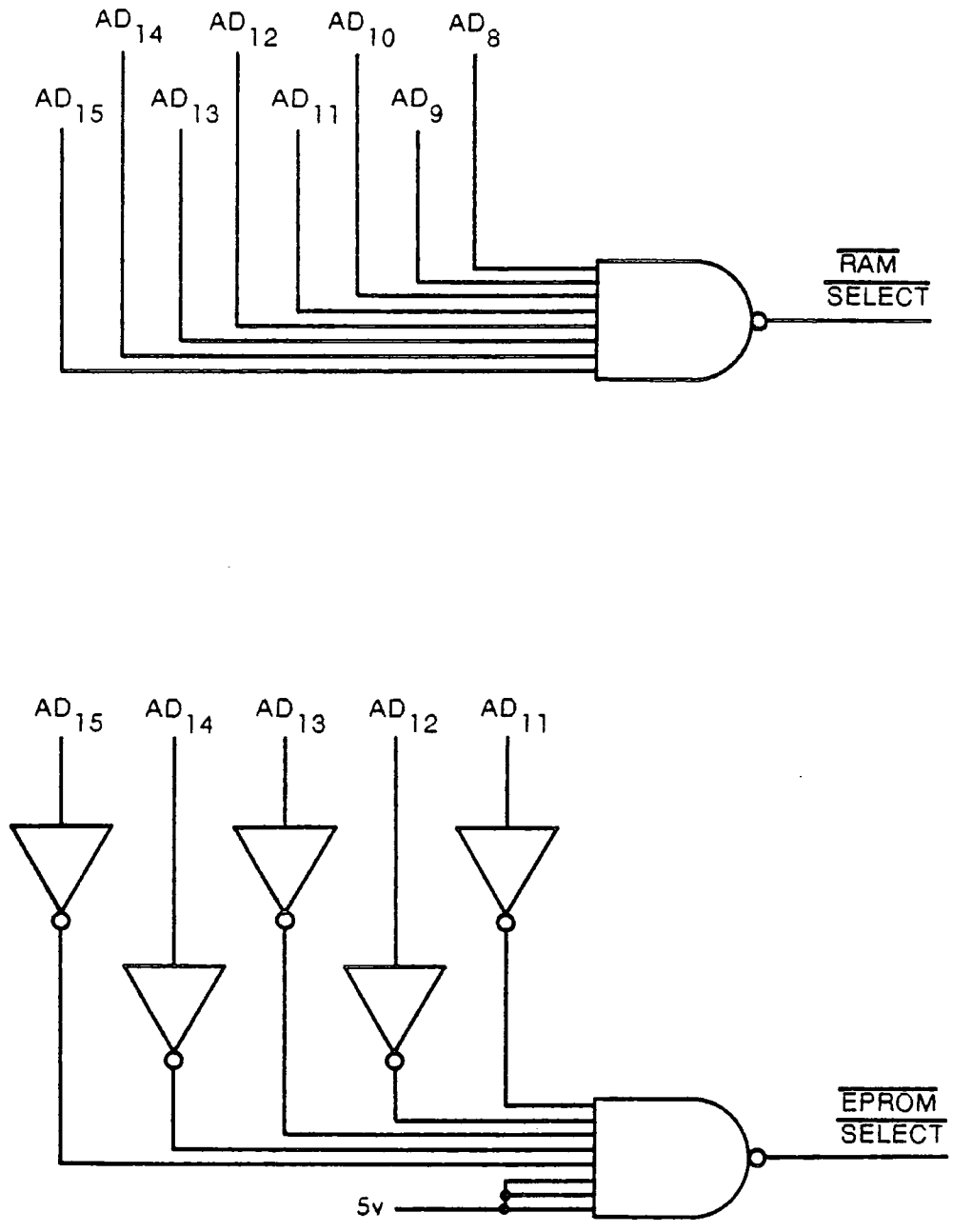


Figure 2.3 Logic Diagram of the Memory Decoding Circuitry

T1 240/6-0-6 Toroidal transformer
T2 240/0-10 Laminar transformer
C1 2,200 uF Capacitor
C2 4,700 uF Capacitor
C3 10,000 uF Capacitor
C4 220 nF Capacitor
C5 470 nF Capacitor
R1 4.7 K Ohm Resistor
R2 470 Ohm Resistor
D1 Green L.E.D

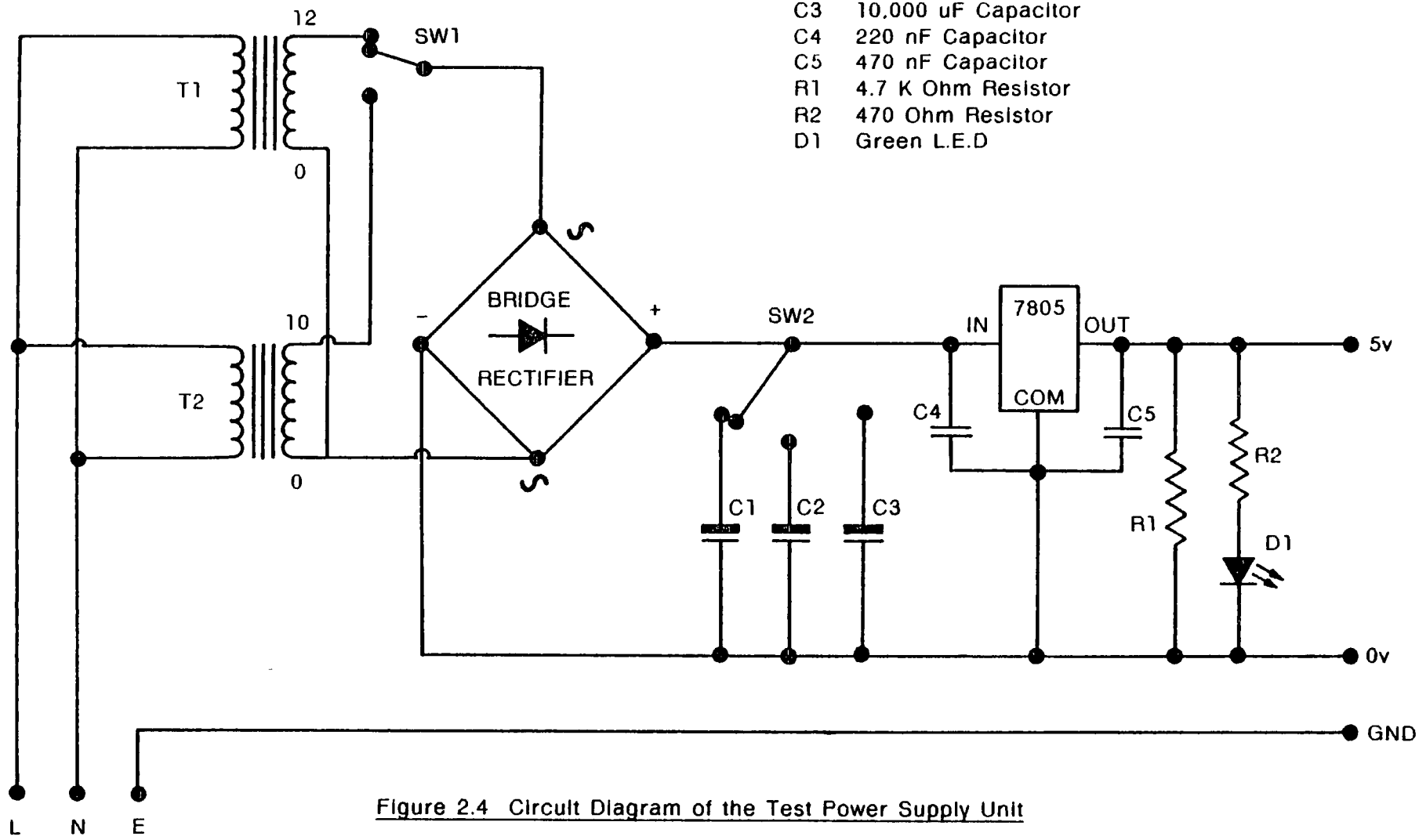


Figure 2.4 Circuit Diagram of the Test Power Supply Unit

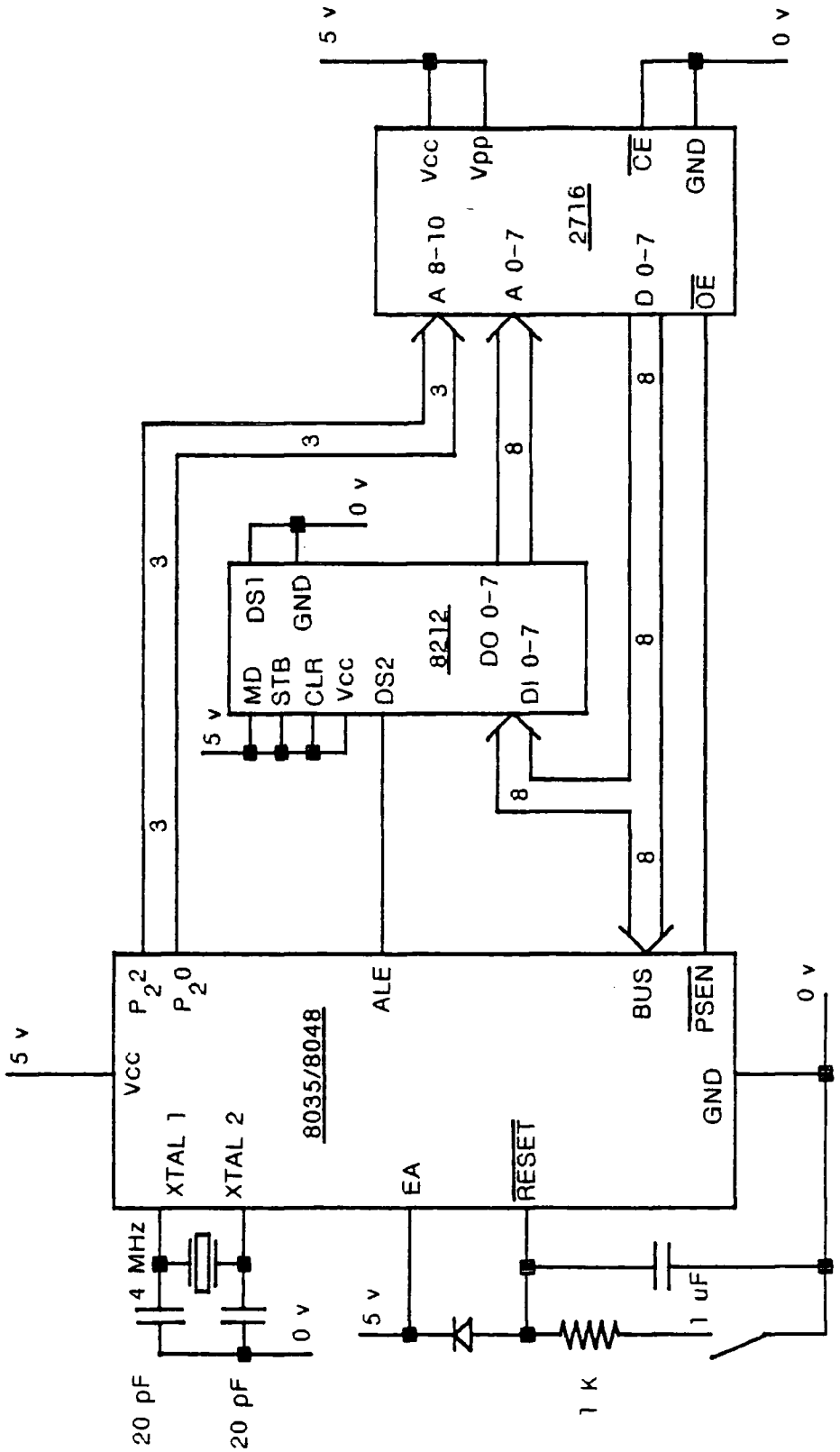


Figure 3.1 Block Diagram of the 8035/8048 Test System

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP 1B,1C	*NOP* 1B,1C	OUTL BUS,A 1B,2C	ADD A,#d 2B,2C	JMP 0XX 2B,2C	EN I 1B,1C	*JNTF* add 2B,2C	DEC A 1B,1C	INS A,BUS 1B,2C	IN A,P1 1B,2C	IN A,P2 1B,2C	*IN* A,P2 1B,2C	MOVD A,P4 1B,2C	MOVD A,P5 1B,2C	MOVD A,P6 1B,2C	MOVD A,P7 1B,2C
1	INC @R0 1B,1C	INC @R1 1B,1C	JB0 add 2B,2C	ADDC A,#d 2B,2C	CALL 0XX 2B,2C	DIS I 1B,1C	JTF add 2B,2C	INC A 1B,1C	INC R0 1B,1C	INC R1 1B,1C	INC R2 1B,1C	INC R3 1B,1C	INC R4 1B,1C	INC R5 1B,1C	INC R6 1B,1C	INC R7 1B,1C
2	XCH A,@R0 1B,1C	XCH A,@R1 1B,1C	*NOP* 1B,1C	MOV A,#d 2B,2C	JMP 1XX 2B,2C	EN TCNTI 1B,1C	JNT0 add 2B,2C	CLR A 1B,1C	XCH A,R0 1B,1C	XCH A,R1 1B,1C	XCH A,R2 1B,1C	XCH A,R3 1B,1C	XCH A,R4 1B,1C	XCH A,R5 1B,1C	XCH A,R6 1B,1C	XCH A,R7 1B,1C
3	XCHD A,@R0 1B,1C	XCHD A,@R1 1B,1C	JB1 add 2B,2C	*NOP* 1B,1C	CALL 1XX 2B,2C	DIS TCNTI 1B,1C	JT0 add 2B,2C	CPL A 1B,1C	*BUS* IDLE 1B,2C	OUTL P1,A 1B,2C	OUTL P2,A 1B,2C	*OUTL* P2,A 1B,2C	MOVD P4,A 1B,2C	MOVD P5,A 1B,2C	MOVD P6,A 1B,2C	MOVD P7,A 1B,2C
4	ORL A,@R0 1B,1C	ORL A,@R1 1B,1C	MOV A,T 1B,1C	ORL A,#d 2B,2C	JMP 2XX 2B,2C	STRT CNT 1B,1C	JNT1 add 2B,2C	SWAP A 1B,1C	ORL A,R0 1B,1C	ORL A,R1 1B,1C	ORL A,R2 1B,1C	ORL A,R3 1B,1C	ORL A,R4 1B,1C	ORL A,R5 1B,1C	ORL A,R6 1B,1C	ORL A,R7 1B,1C
5	ANL A,@R0 1B,1C	ANL A,@R1 1B,1C	JB2 add 2B,2C	ANL A,#d 2B,2C	CALL 2XX 2B,2C	STRT T 1B,1C	JT1 add 2B,2C	DA A 1B,1C	ANL A,R0 1B,1C	ANL A,R1 1B,1C	ANL A,R2 1B,1C	ANL A,R3 1B,1C	ANL A,R4 1B,1C	ANL A,R5 1B,1C	ANL A,R6 1B,1C	ANL A,R7 1B,1C
6	ADD A,@R0 1B,1C	ADD A,@R1 1B,1C	MOV T,A 1B,1C	*NOP* 1B,1C	JMP 3XX 2B,2C	STOP TCNT 1B,1C	*JNF1* add 2B,2C	RRC A 1B,1C	ADD A,R0 1B,1C	ADD A,R1 1B,1C	ADD A,R2 1B,1C	ADD A,R3 1B,1C	ADD A,R4 1B,1C	ADD A,R5 1B,1C	ADD A,R6 1B,1C	ADD A,R7 1B,1C
7	ADDC A,@R0 1B,1C	ADDC A,@R1 1B,1C	JB3 add 2B,2C	*NOP* 1B,1C	CALL 3XX 2B,2C	ENTO CLK 1B,1C	JF1 add 2B,2C	RR A 1B,1C	ADDC A,R0 1B,1C	ADDC A,R1 1B,1C	ADDC A,R2 1B,1C	ADDC A,R3 1B,1C	ADDC A,R4 1B,1C	ADDC A,R5 1B,1C	ADDC A,R6 1B,1C	ADDC A,R7 1B,1C
8	MOVX A,@R0 1B,2C	MOVX A,@R1 1B,2C	*NOP* 1B,1C	RET 1B,2C	JMP 4XX 2B,2C	CLR F0 1B,1C	JN1 add 2B,2C	*NOP* 1B,1C	ORL BUS,#d 2B,2C	ORL P1,#d 2B,2C	ORL P2,#d 2B,2C	*ORL* P2,#d 2B,2C	ORLD P4,A 1B,2C	ORLD P5,A 1B,2C	ORLD P6,A 1B,2C	ORLD P7,A 1B,2C
9	MOVX @R0,A 1B,2C	MOVX @R1,A 1B,2C	JB4 add 2B,2C	RETR 1B,2C	CALL 4XX 2B,2C	CPL F0 1B,1C	JNZ add 2B,2C	CLR C 1B,1C	ANL BUS,#d 2B,2C	ANL P1,#d 2B,2C	ANL P2,#d 2B,2C	*ANL* P2,#d 2B,2C	ANLD P4,A 1B,2C	ANLD P5,A 1B,2C	ANLD P6,A 1B,2C	ANLD P7,A 1B,2C
A	MOV @R0,A 1B,1C	MOV @R1,A 1B,1C	*NOP* 1B,1C	MOVP A,@A 1B,2C	JMP 5XX 2B,2C	CLR F1 1B,1C	*JNF0* add 2B,2C	CPL C 1B,1C	MOV R0,A 1B,1C	MOV R1,A 1B,1C	MOV R2,A 1B,1C	MOV R3,A 1B,1C	MOV R4,A 1B,1C	MOV R5,A 1B,1C	MOV R6,A 1B,1C	MOV R7,A 1B,1C
B	MOV @R0,#d 2B,2C	MOV @R1,#d 2B,2C	JB5 add 2B,2C	JMPP @A 1B,2C	CALL 5XX 2B,2C	CPL F1 1B,1C	JF0 add 2B,2C	*NOP* 1B,1C	MOV R0,#d 2B,2C	MOV R1,#d 2B,2C	MOV R2,#d 2B,2C	MOV R3,#d 2B,2C	MOV R4,#d 2B,2C	MOV R5,#d 2B,2C	MOV R6,#d 2B,2C	MOV R7,#d 2B,2C
C	*NOP* 1B,1C	*NOP* 1B,1C	*NOP* 1B,1C	*NOP* 1B,1C	JMP 6XX 2B,2C	SEL RB0 1B,1C	JZ add 2B,2C	MOV A,PSW 1B,1C	DEC R0 1B,1C	DEC R1 1B,1C	DEC R2 1B,1C	DEC R3 1B,1C	DEC R4 1B,1C	DEC R5 1B,1C	DEC R6 1B,1C	DEC R7 1B,1C
D	XRL A,@R0 1B,1C	XRL A,@R1 1B,1C	JB6 add 2B,2C	XRL A,#d 2B,2C	CALL 6XX 2B,2C	SEL RB1 1B,1C	*JMPP* add 2B,2C	MOV PSW,A 1B,1C	XRL A,R0 1B,1C	XRL A,R1 1B,1C	XRL A,R2 1B,1C	XRL A,R3 1B,1C	XRL A,R4 1B,1C	XRL A,R5 1B,1C	XRL A,R6 1B,1C	XRL A,R7 1B,1C
E	*NOP* 1B,1C	*NOP* 1B,1C	*NOP* 1B,1C	MOVP3 A,@A 1B,2C	JMP 7XX 2B,2C	SEL MB0 1B,1C	JNC add 2B,2C	RL A 1B,1C	DJNZ R0,add 2B,2C	DJNZ R1,add 2B,2C	DJNZ R2,add 2B,2C	DJNZ R3,add 2B,2C	DJNZ R4,add 2B,2C	DJNZ R5,add 2B,2C	DJNZ R6,add 2B,2C	DJNZ R7,add 2B,2C
F	MOV A,@R0 1B,1C	MOV A,@R1 1B,1C	JB7 add 2B,2C	*NOP* 1B,1C	CALL 7XX 2B,2C	SEL MB1 1B,1C	JC add 2B,2C	RLC A 1B,1C	MOV A,R0 1B,1C	MOV A,R1 1B,1C	MOV A,R2 1B,1C	MOV A,R3 1B,1C	MOV A,R4 1B,1C	MOV A,R5 1B,1C	MOV A,R6 1B,1C	MOV A,R7 1B,1C

@ - INDIRECT ADDRESSING
# - IMMEDIATE ADDRESSING
B - BYTES
C - CYCLES

add - ADDRESS
d - DATA
* * - UNDECLARED INSTRUCTION

Figure 3.2 Full Instruction Set for the 8048 Manufactured by Intel

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP 1B,1C	*NOP* 1B,1C	OUTL BUS,A 1B,2C	ADD A,#d 2B,2C	JMP 0XX 2B,2C	EN I 1B,1C	*JNTF* add 2B,2C	DEC A 1B,1C	INS A,BUS 1B,2C	IN A,P1 1B,2C	IN A,P2 1B,2C	*IN* A,P2 1B,2C	MOVD A,P4 1B,2C	MOVD A,P5 1B,2C	MOVD A,P6 1B,2C	MOVD A,P7 1B,2C
1	INC @R0 1B,1C	INC @R1 1B,1C	JB0 add 2B,2C	ADDC A,#d 2B,2C	CALL 0XX 2B,2C	DIS I 1B,1C	JTF add 2B,2C	INC A 1B,1C	INC R0 1B,1C	INC R1 1B,1C	INC R2 1B,1C	INC R3 1B,1C	INC R4 1B,1C	INC R5 1B,1C	INC R6 1B,1C	INC R7 1B,1C
2	XCH A,@R0 1B,1C	XCH A,@R1 1B,1C	*MOV* A,PC+1 1B,1C	MOV A,#d 2B,2C	JMP 1XX 2B,2C	EN TCNTI 1B,1C	JNT0 add 2B,2C	CLR A 1B,1C	XCH A,R0 1B,1C	XCH A,R1 1B,1C	XCH A,R2 1B,1C	XCH A,R3 1B,1C	XCH A,R4 1B,1C	XCH A,R5 1B,1C	XCH A,R6 1B,1C	XCH A,R7 1B,1C
3	XCHD A,@R0 1B,1C	XCHD A,@R1 1B,1C	JB1 add 2B,2C	*NOP* 1B,1C	CALL 1XX 2B,2C	DIS TCNTI 1B,1C	JT0 add 2B,2C	CPL A 1B,1C	*BUS* IDLE 1B,2C	OUTL P1,A 1B,2C	OUTL P2,A 1B,2C	*OUTL* P2,A 1B,2C	MOVD P4,A 1B,2C	MOVD P5,A 1B,2C	MOVD P6,A 1B,2C	MOVD P7,A 1B,2C
4	ORL A,@R0 1B,1C	ORL A,@R1 1B,1C	MOV A,T 1B,1C	ORL A,#d 2B,2C	JMP 2XX 2B,2C	STRT CNT 1B,1C	JNT1 add 2B,2C	SWAP A 1B,1C	ORL A,R0 1B,1C	ORL A,R1 1B,1C	ORL A,R2 1B,1C	ORL A,R3 1B,1C	ORL A,R4 1B,1C	ORL A,R5 1B,1C	ORL A,R6 1B,1C	ORL A,R7 1B,1C
5	ANL A,@R0 1B,1C	ANL A,@R1 1B,1C	JB2 add 2B,2C	ANL A,#d 2B,2C	CALL 2XX 2B,2C	STRT T 1B,1C	JT1 add 2B,2C	DA A 1B,1C	ANL A,R0 1B,1C	ANL A,R1 1B,1C	ANL A,R2 1B,1C	ANL A,R3 1B,1C	ANL A,R4 1B,1C	ANL A,R5 1B,1C	ANL A,R6 1B,1C	ANL A,R7 1B,1C
6	ADD A,@R0 1B,1C	ADD A,@R1 1B,1C	MOV T,A 1B,1C	*NOP* 1B,1C	JMP 3XX 2B,2C	STOP TCNT 1B,1C	*JNF1* add 2B,2C	RRC A 1B,1C	ADD A,R0 1B,1C	ADD A,R1 1B,1C	ADD A,R2 1B,1C	ADD A,R3 1B,1C	ADD A,R4 1B,1C	ADD A,R5 1B,1C	ADD A,R6 1B,1C	ADD A,R7 1B,1C
7	ADDC A,@R0 1B,1C	ADDC A,@R1 1B,1C	JB3 add 2B,2C	*NOP* 1B,1C	CALL 3XX 2B,2C	ENTO CLK 1B,1C	JF1 add 2B,2C	RR A 1B,1C	ADDC A,R0 1B,1C	ADDC A,R1 1B,1C	ADDC A,R2 1B,1C	ADDC A,R3 1B,1C	ADDC A,R4 1B,1C	ADDC A,R5 1B,1C	ADDC A,R6 1B,1C	ADDC A,R7 1B,1C
8	MOVX A,@R0 1B,2C	MOVX A,@R1 1B,2C	*NOP* 1B,1C	RET 1B,2C	JMP 4XX 2B,2C	CLR F0 1B,1C	JN1 add 2B,2C	*CLR* A4-A7 1B,1C	ORL BUS,#d 2B,2C	ORL P1,#d 2B,2C	ORL P2,#d 2B,2C	*ORL* P2,#d 2B,2C	ORLD P4,A 1B,2C	ORLD P5,A 1B,2C	ORLD P6,A 1B,2C	ORLD P7,A 1B,2C
9	MOVX @R0,A 1B,2C	MOVX @R1,A 1B,2C	JB4 add 2B,2C	RETR 1B,2C	CALL 4XX 2B,2C	CPL F0 1B,1C	JN2 add 2B,2C	CLR C 1B,1C	ANL BUS,#d 2B,2C	ANL P1,#d 2B,2C	ANL P2,#d 2B,2C	*ANL* P2,#d 2B,2C	ANLD P4,A 1B,2C	ANLD P5,A 1B,2C	ANLD P6,A 1B,2C	ANLD P7,A 1B,2C
A	MOV @R0,A 1B,1C	MOV @R1,A 1B,1C	*NOP* 1B,1C	MOVP A,@A 1B,2C	JMP 5XX 2B,2C	CLR F1 1B,1C	*JNF0* add 2B,2C	CPL C 1B,1C	MOV R0,A 1B,1C	MOV R1,A 1B,1C	MOV R2,A 1B,1C	MOV R3,A 1B,1C	MOV R4,A 1B,1C	MOV R5,A 1B,1C	MOV R6,A 1B,1C	MOV R7,A 1B,1C
B	MOV @R0,#d 2B,2C	MOV @R1,#d 2B,2C	JB5 add 2B,2C	JMPP @A 1B,2C	CALL 5XX 2B,2C	CPL F1 1B,1C	JF0 add 2B,2C	*NOP* 1B,1C	MOV R0,#d 2B,2C	MOV R1,#d 2B,2C	MOV R2,#d 2B,2C	MOV R3,#d 2B,2C	MOV R4,#d 2B,2C	MOV R5,#d 2B,2C	MOV R6,#d 2B,2C	MOV R7,#d 2B,2C
C	*DEC* @R0 1B,1C	*DEC* @R1 1B,1C	*NOP* 1B,1C	*NOP* 1B,1C	JMP 6XX 2B,2C	SEL RB0 1B,1C	J2 add 2B,2C	MOV A,PSW 1B,1C	DEC R0 1B,1C	DEC R1 1B,1C	DEC R2 1B,1C	DEC R3 1B,1C	DEC R4 1B,1C	DEC R5 1B,1C	DEC R6 1B,1C	DEC R7 1B,1C
D	XRL A,@R0 1B,1C	XRL A,@R1 1B,1C	JB6 add 2B,2C	XRL A,#d 2B,2C	CALL 6XX 2B,2C	SEL RB1 1B,1C	*JMPP* add 2B,2C	MOV PSW,A 1B,1C	XRL A,R0 1B,1C	XRL A,R1 1B,1C	XRL A,R2 1B,1C	XRL A,R3 1B,1C	XRL A,R4 1B,1C	XRL A,R5 1B,1C	XRL A,R6 1B,1C	XRL A,R7 1B,1C
E	*DJNZ* @R0,add 2B,2C	*DJNZ* @R1,add 2B,2C	*NOP* 1B,1C	MOVP3 A,@A 1B,2C	JMP 7XX 2B,2C	SEL MB0 1B,1C	JNC add 2B,2C	RL A 1B,1C	DJNZ R0,add 2B,2C	DJNZ R1,add 2B,2C	DJNZ R2,add 2B,2C	DJNZ R3,add 2B,2C	DJNZ R4,add 2B,2C	DJNZ R5,add 2B,2C	DJNZ R6,add 2B,2C	DJNZ R7,add 2B,2C
F	MOV A,@R0 1B,1C	MOV A,@R1 1B,1C	JB7 add 2B,2C	*NOP* 1B,1C	CALL 7XX 2B,2C	SEL MB1 1B,1C	JC add 2B,2C	RLC A 1B,1C	MOV A,R0 1B,1C	MOV A,R1 1B,1C	MOV A,R2 1B,1C	MOV A,R3 1B,1C	MOV A,R4 1B,1C	MOV A,R5 1B,1C	MOV A,R6 1B,1C	MOV A,R7 1B,1C

@ - INDIRECT ADDRESSING
# - IMMEDIATE ADDRESSING
B - BYTES
C - CYCLES

add - ADDRESS
d - DATA
* * - UNDECLARED INSTRUCTION

Figure 3.3 Full Instruction Set for the 8048 Manufactured by NEC

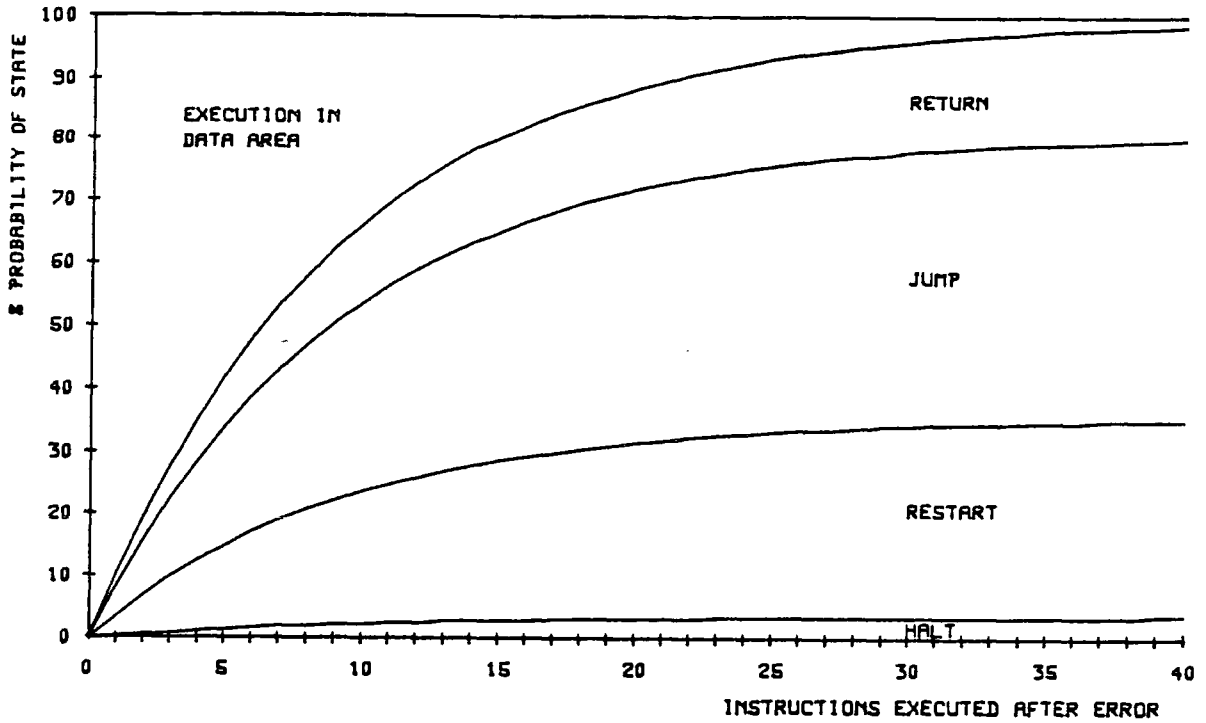


Figure 4.1 (a) Erroneous Execution in Data Areas of the 8085

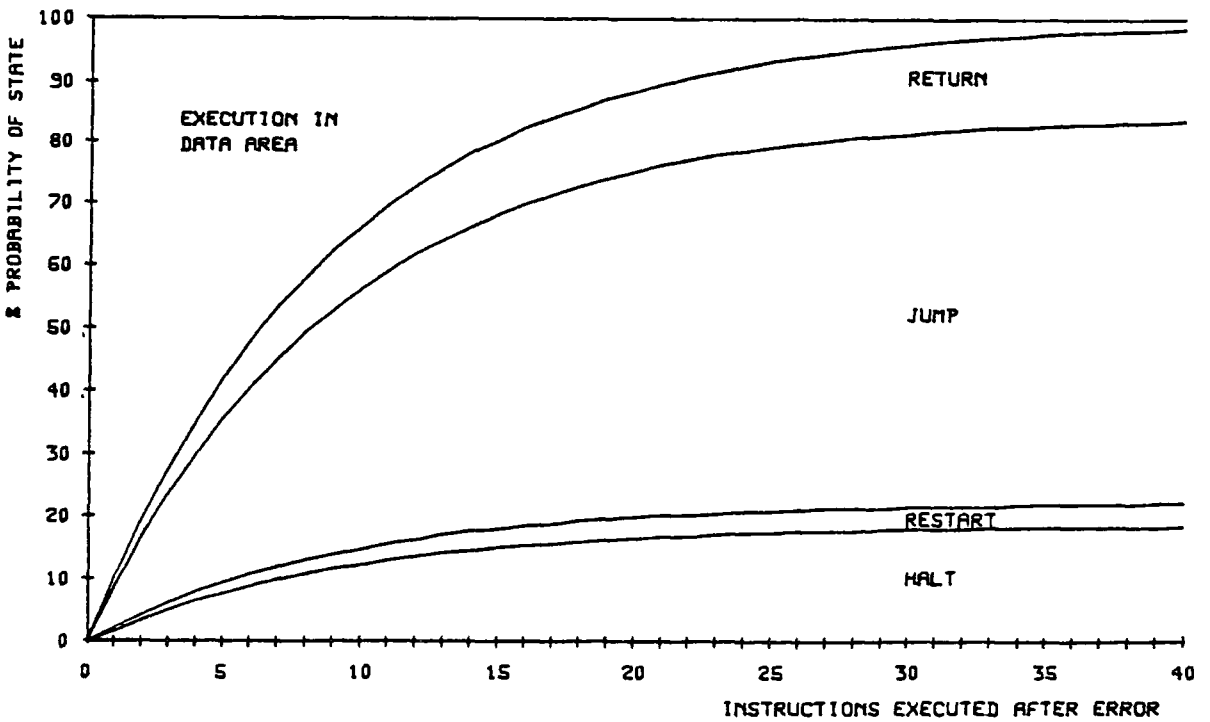


Figure 4.1 (b) Erroneous Execution in the Data Areas of the 6800

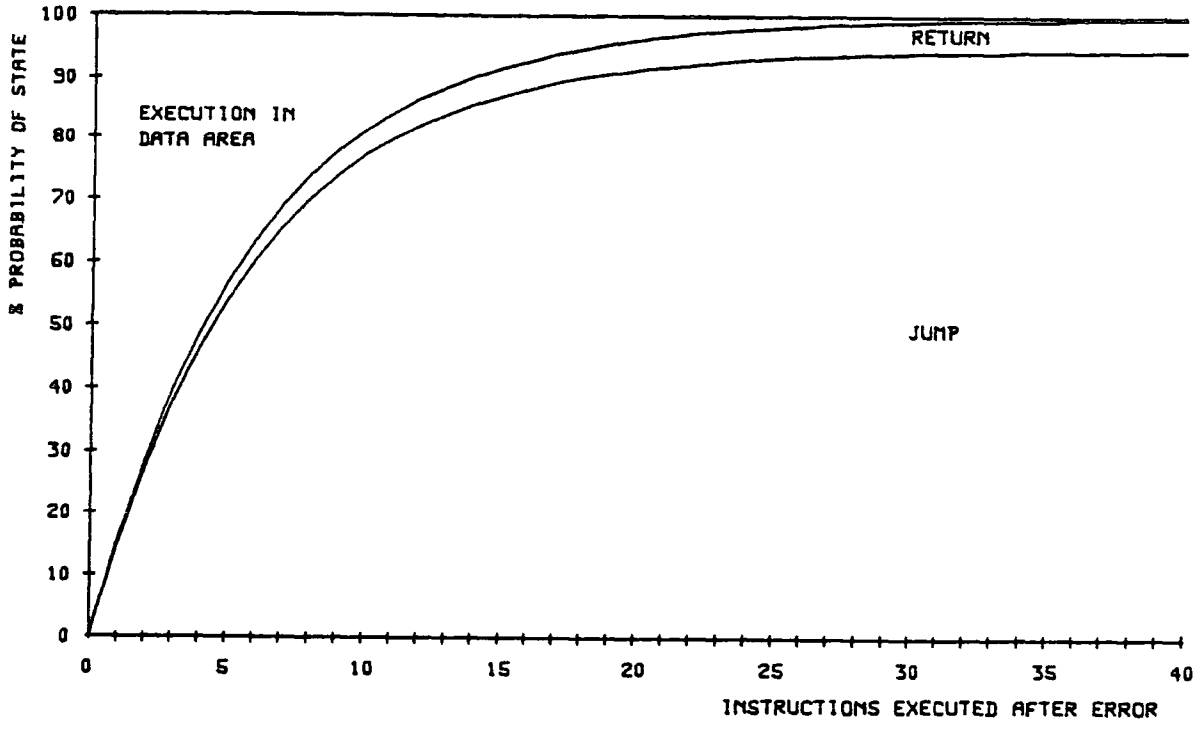


Figure 4.1 (c) Erroneous Execution in the Data Areas of the NEC 8048

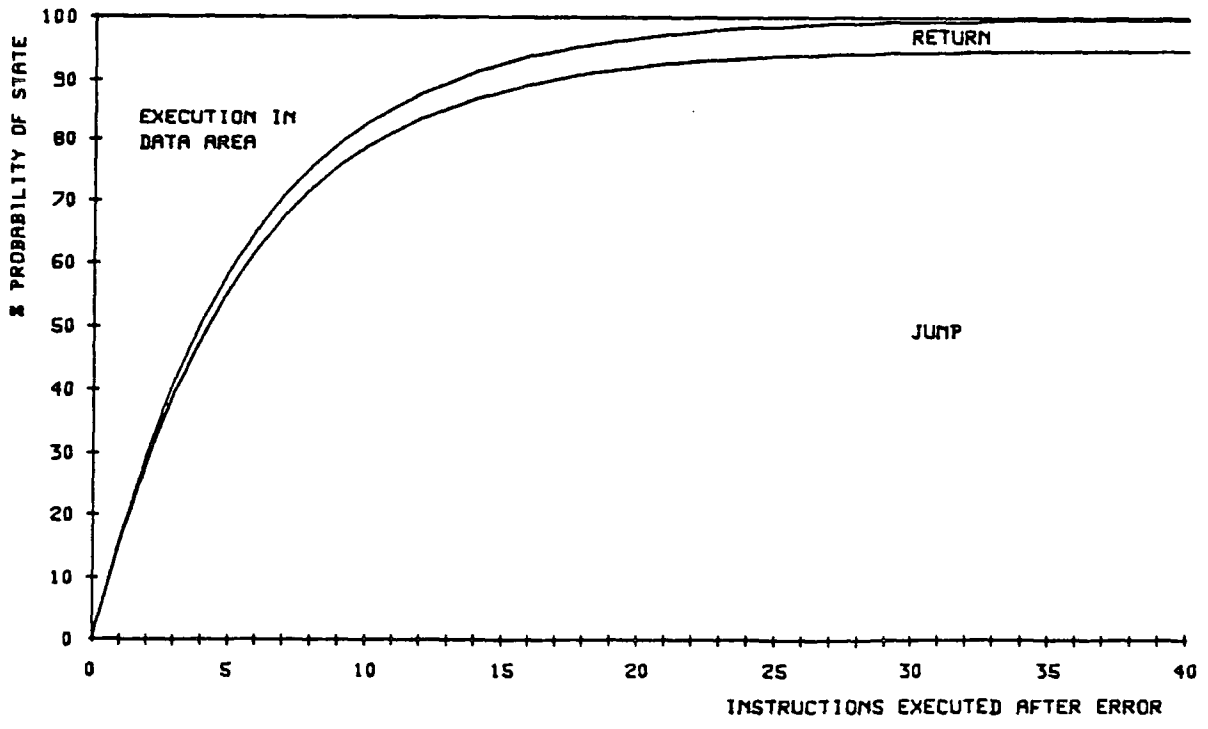


Figure 4.1 (d) Erroneous Execution in the Data Areas of the Intel 8048

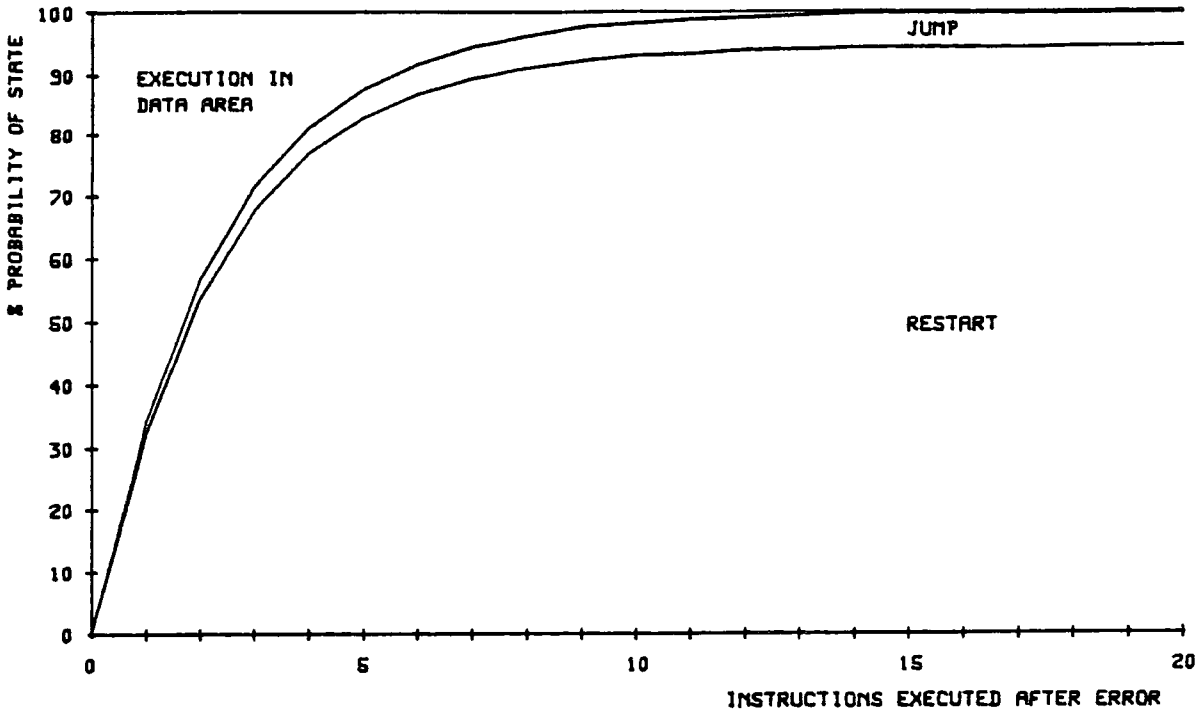


Figure 4.1 (e) Erroneous Execution in the Data Areas of the 68000

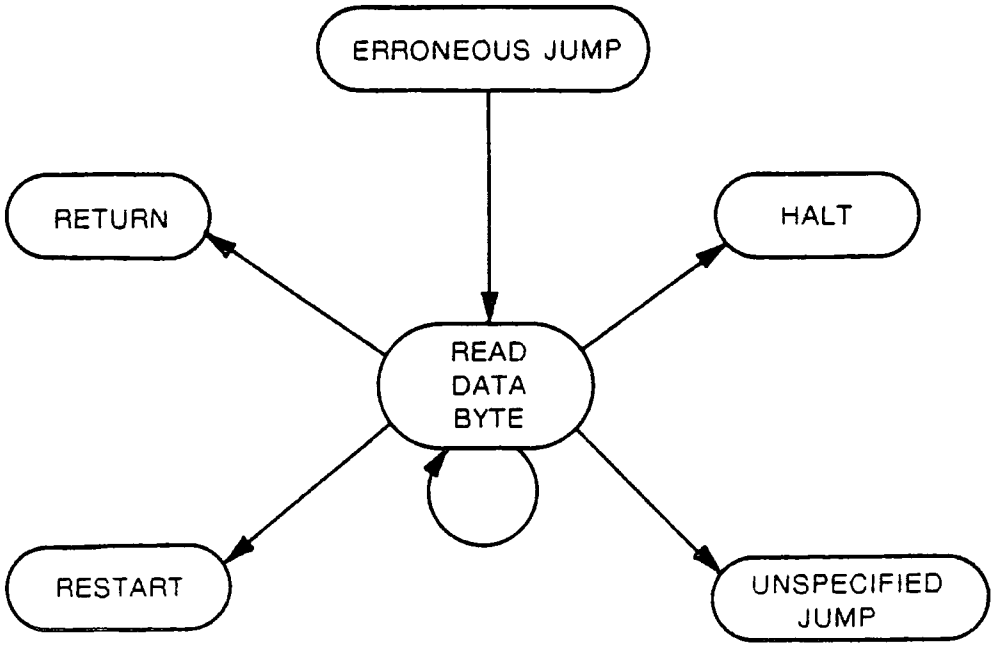


Figure 4.2 Flow of Execution in Random Data

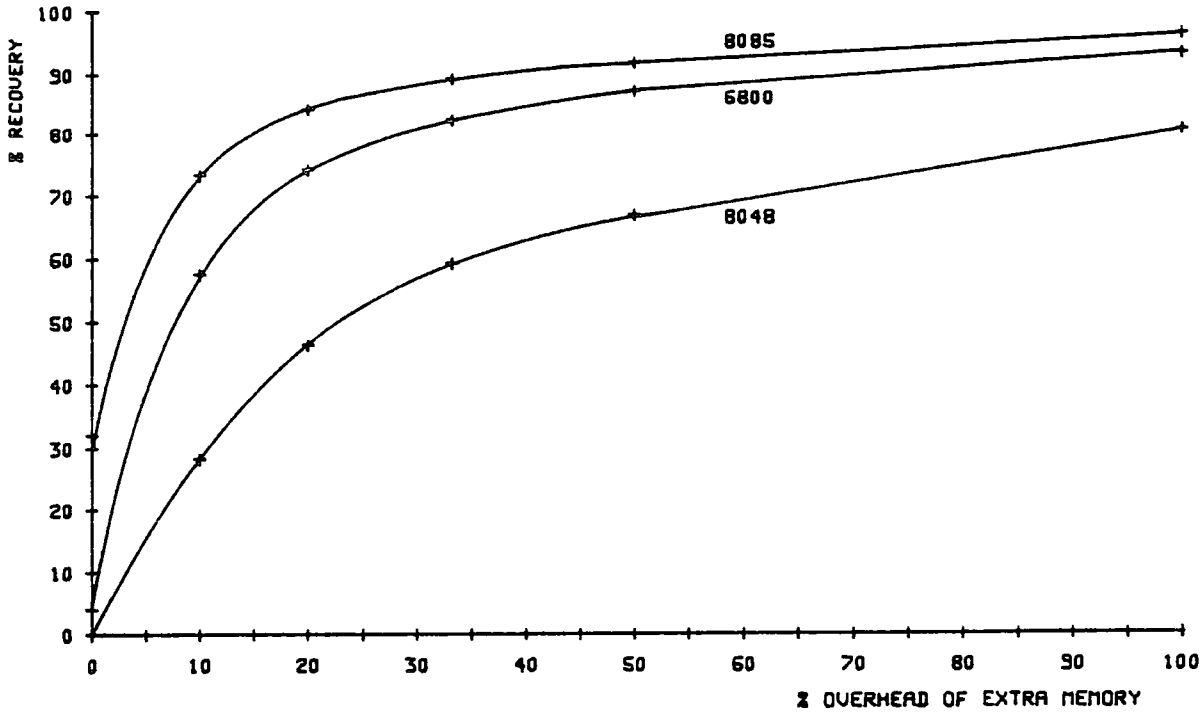


Figure 4.3 Recovery Improvements Obtained by Seeding the Data Areas

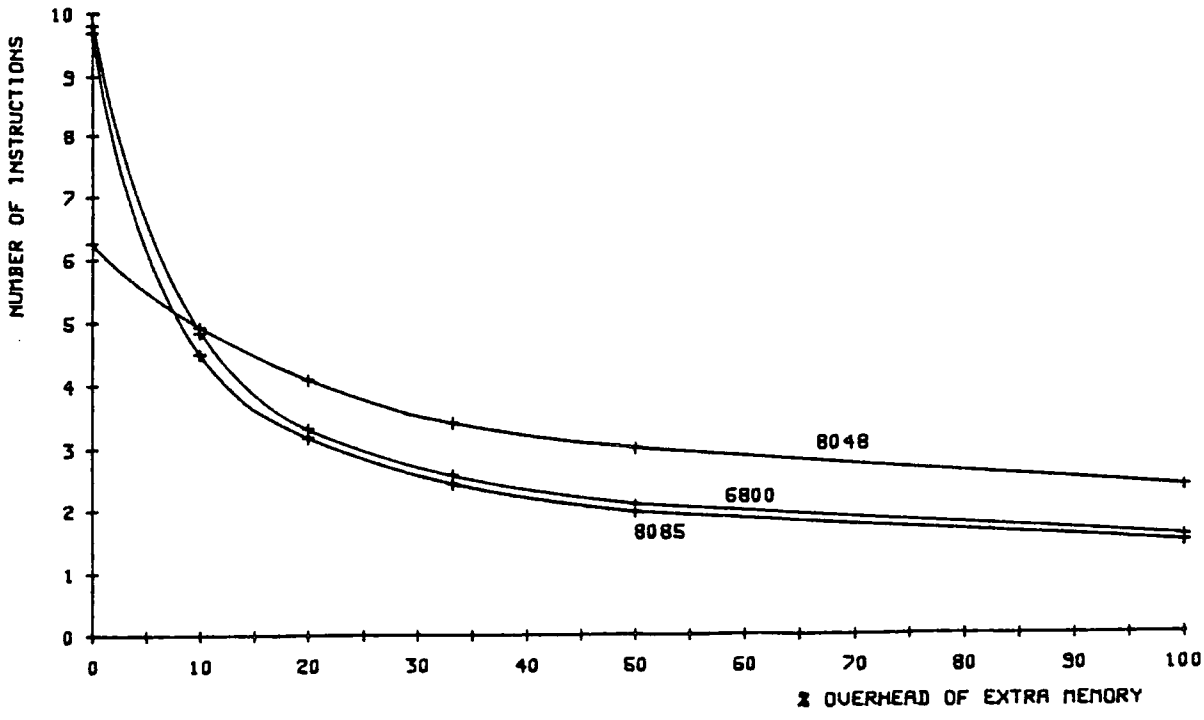


Figure 4.4 Average Number of Instructions Executed with Seeded Data Areas

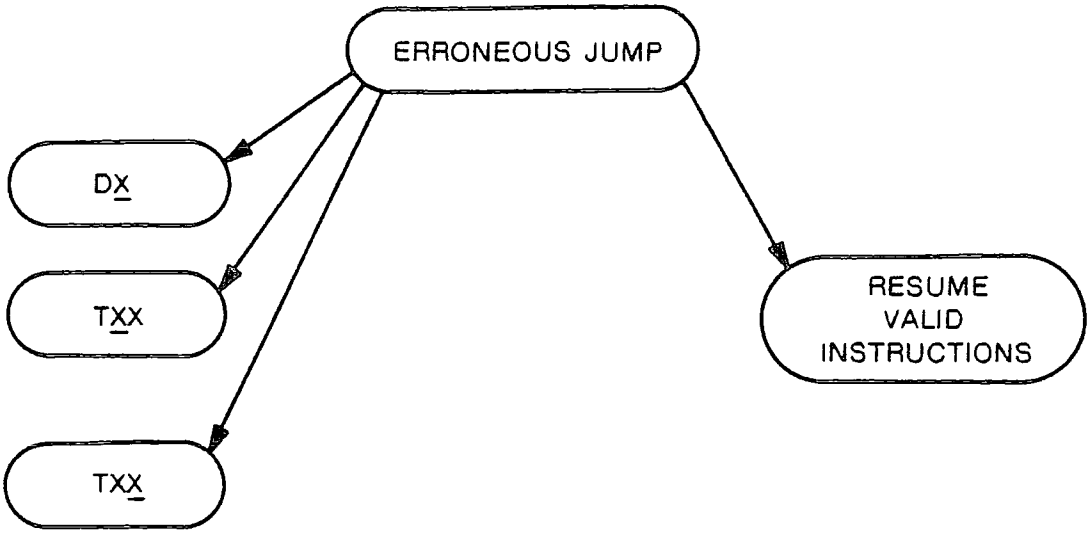


Figure 5.1 Erroneous Jump into a Program Area

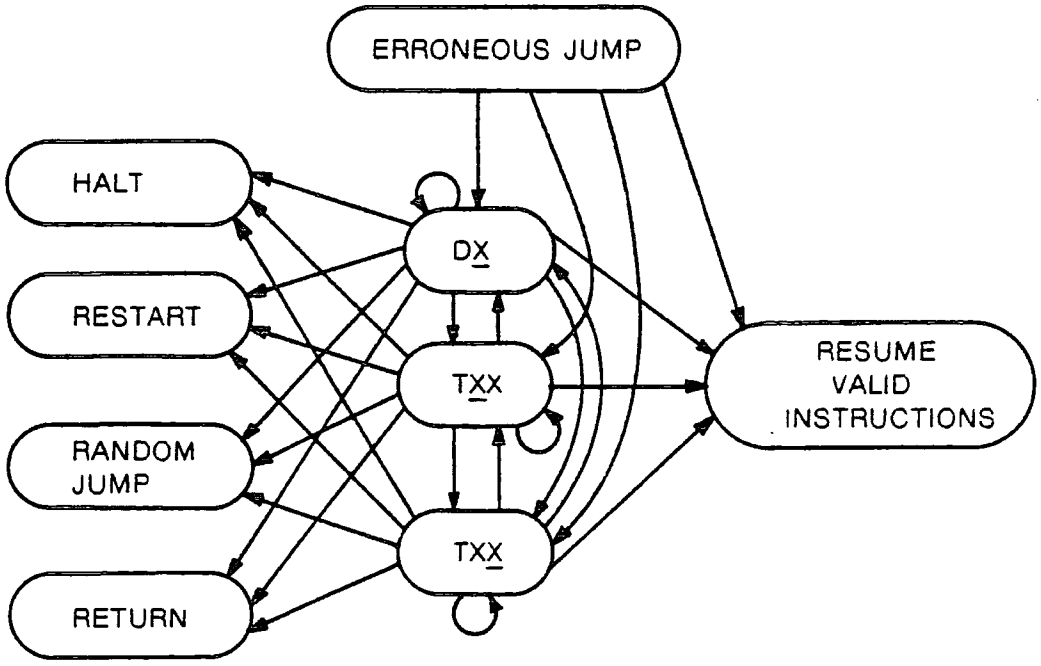


Figure 5.2 Flow of Erroneous Execution in Program Areas

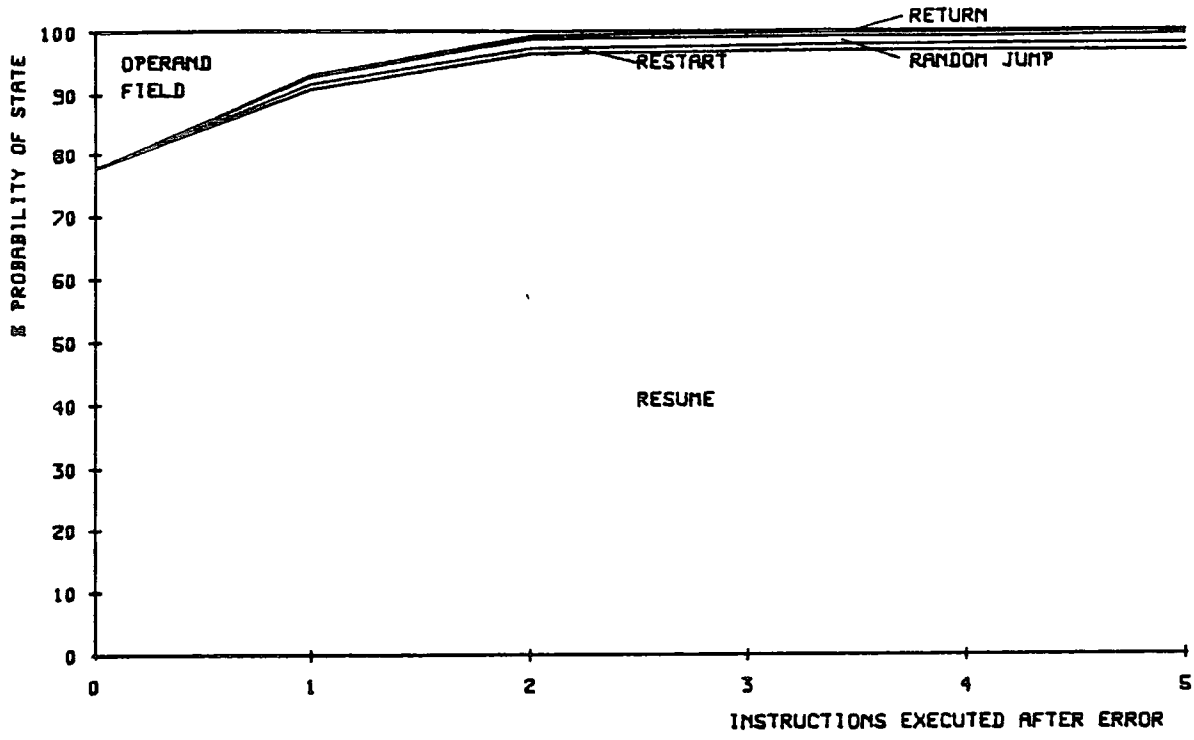


Figure 5.3 (a) Erroneous Execution in Program Areas of the 8085

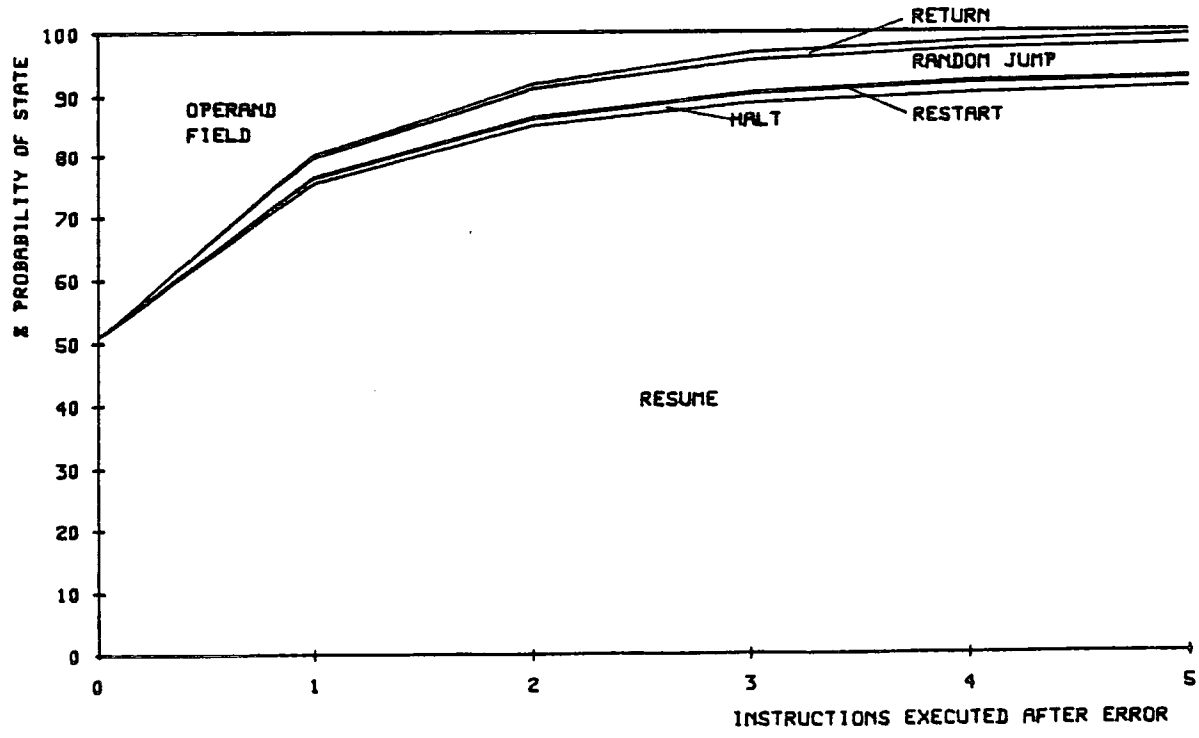


Figure 5.3 (b) Erroneous Execution in the Program Areas of the 6800

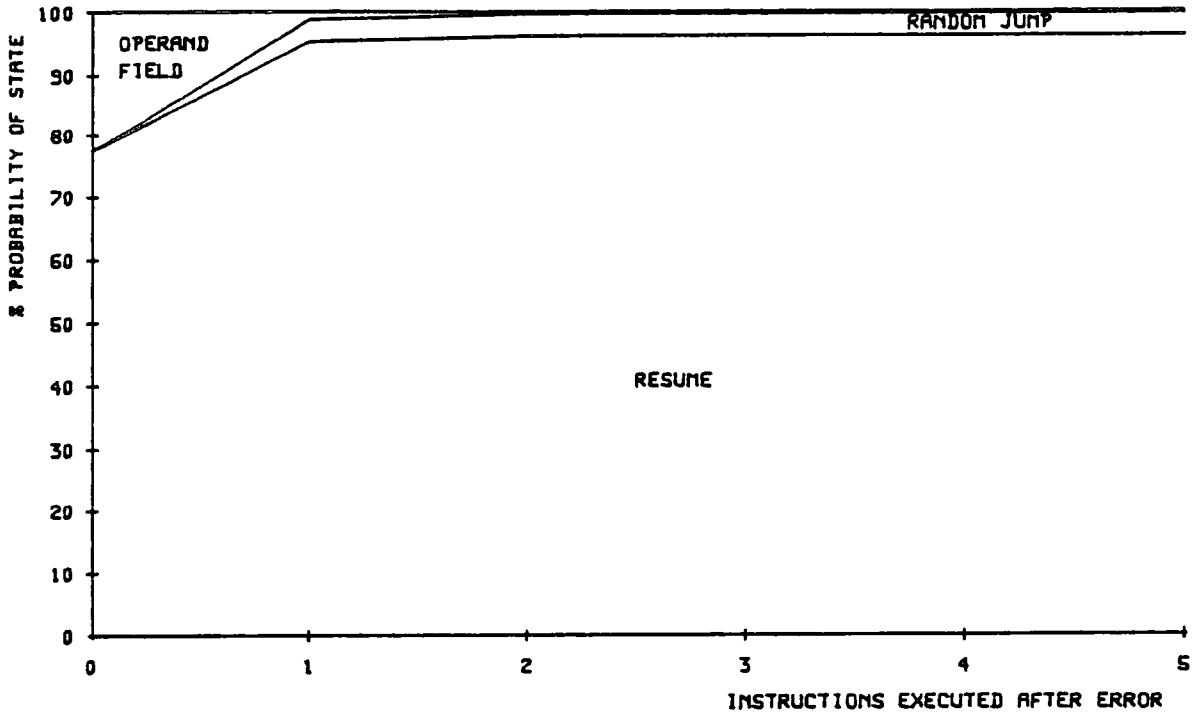


Figure 5.3 (c) Erroneous Execution in the Program Areas of the 8048

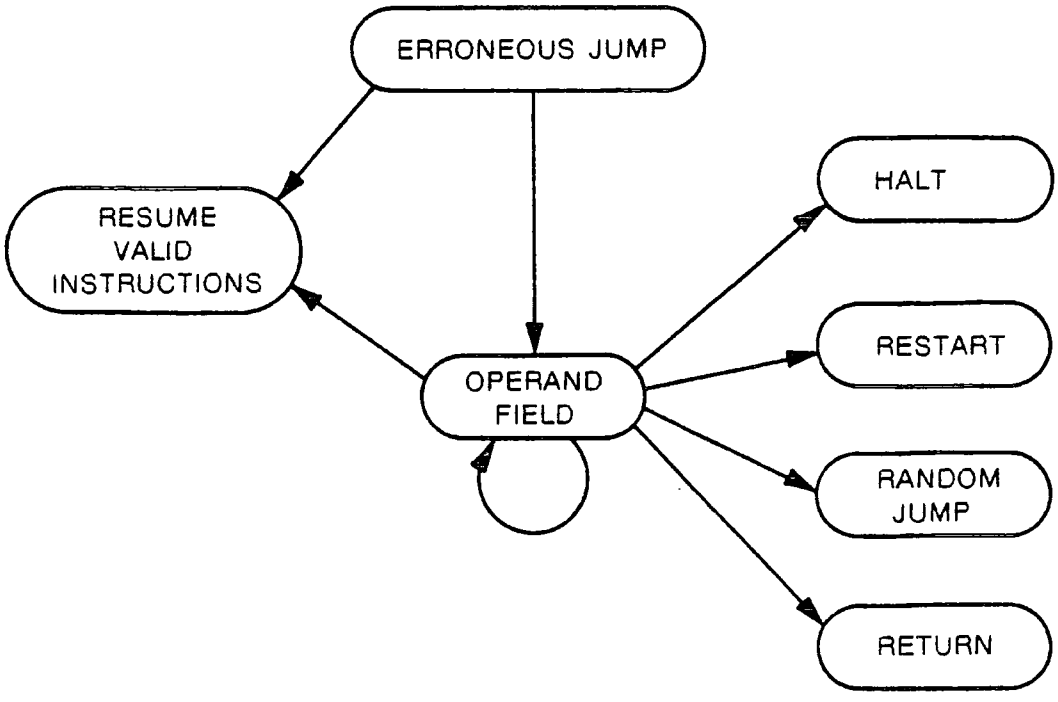


Figure 5.4 Simplified Flow of Execution in Program Areas

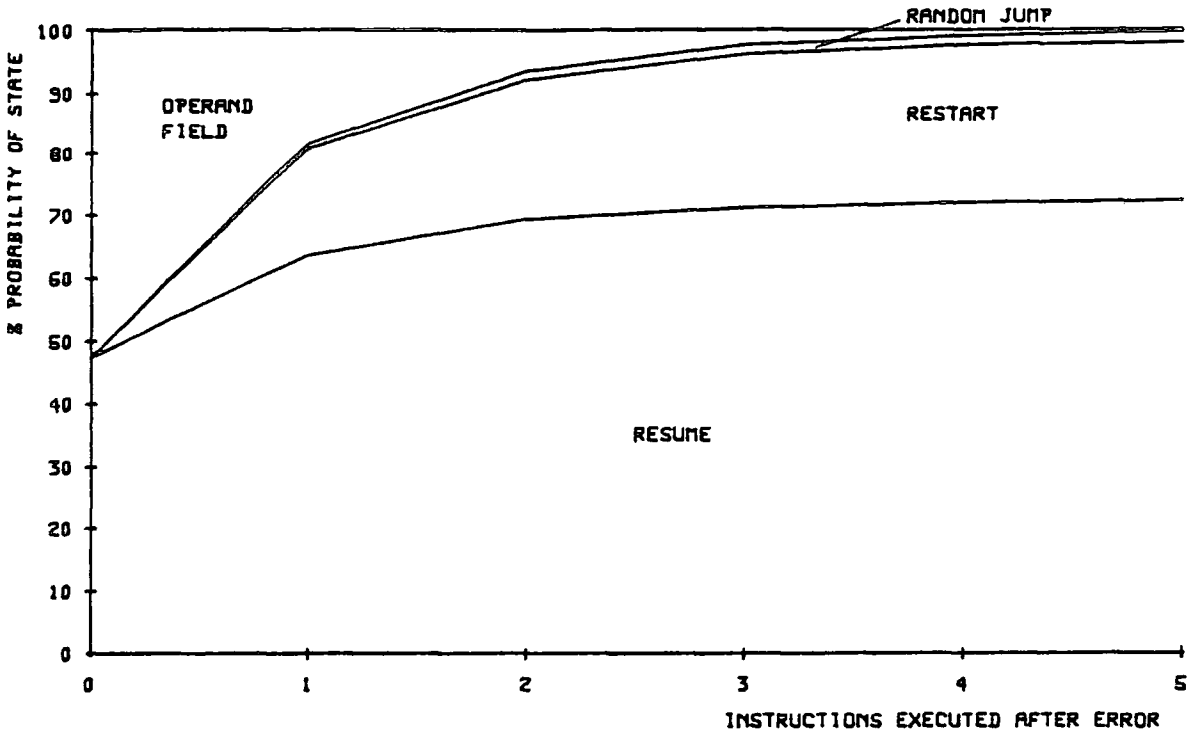


Figure 5.5 Erroneous Execution in Program Areas of the 68000

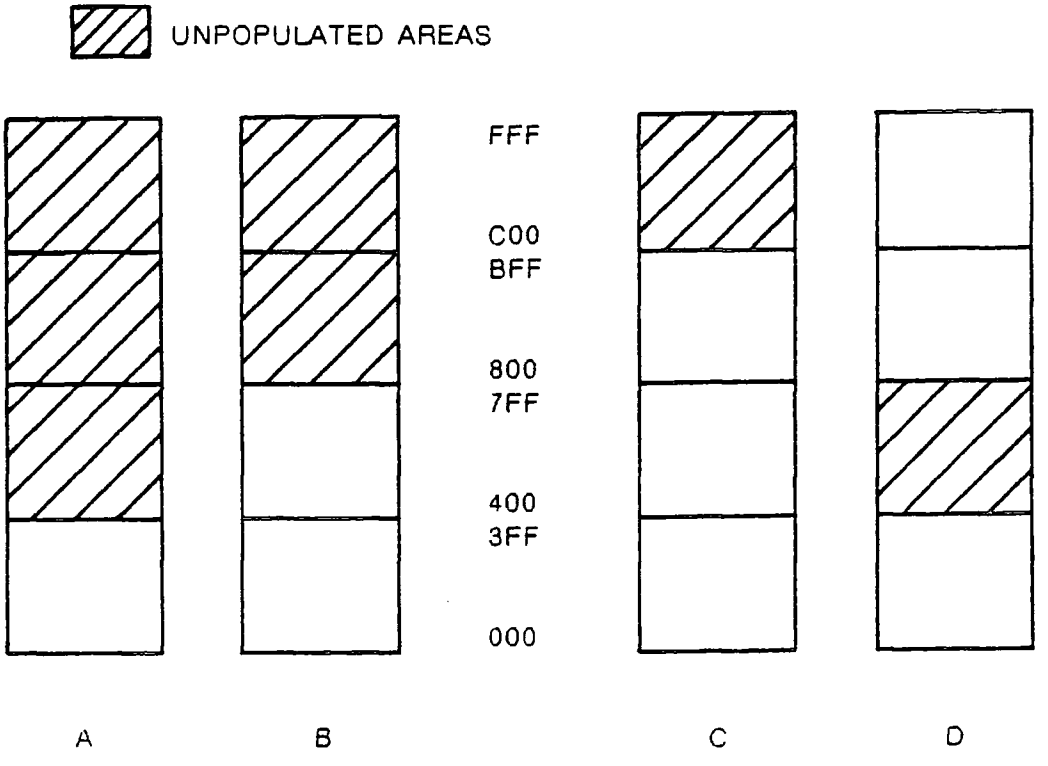


Figure 6.1 Common Memory Arrangements for the 8048

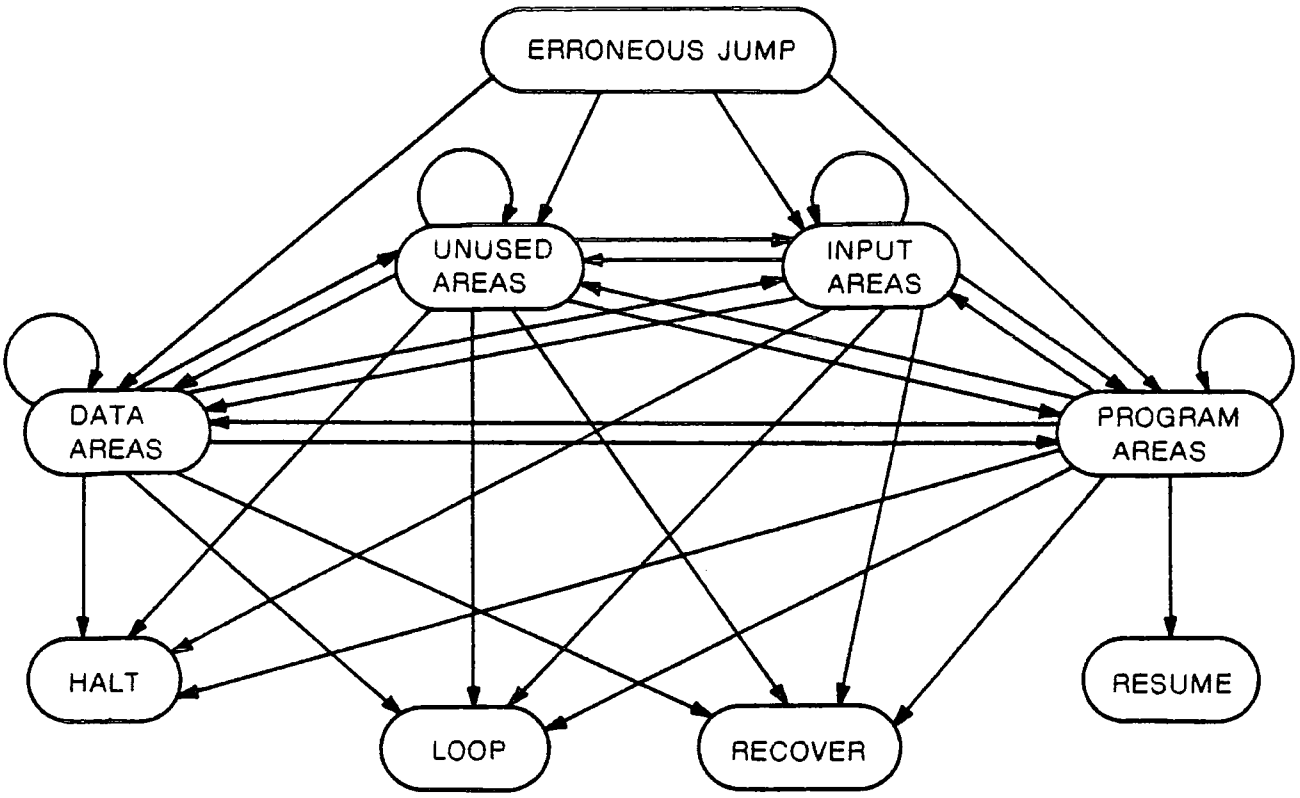
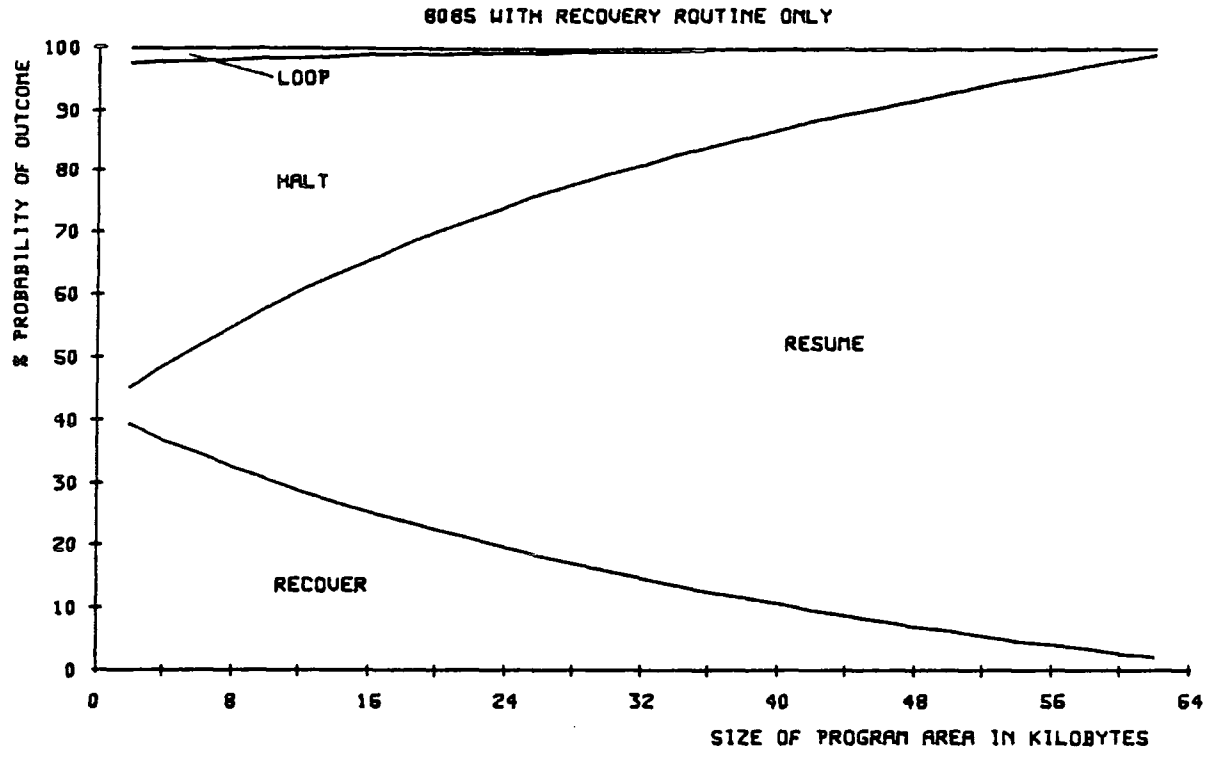


Figure 7.1 Flow of Execution Between Different Memory Areas



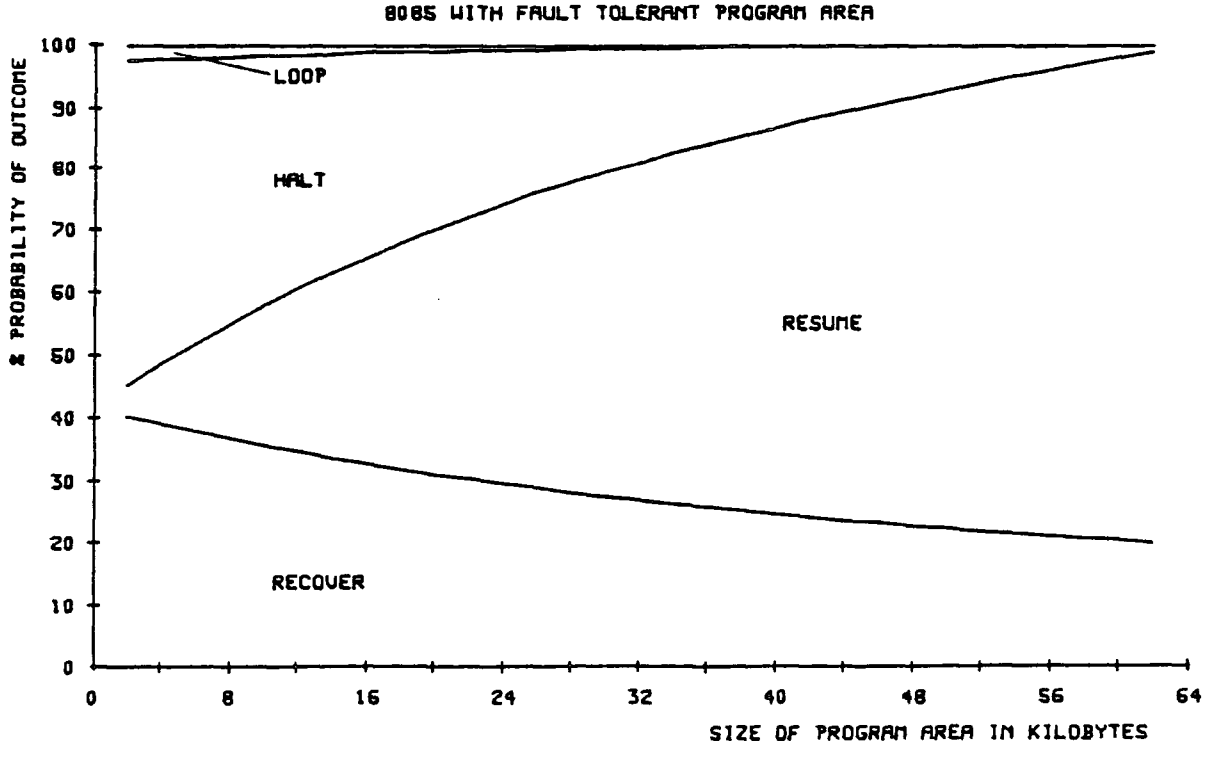
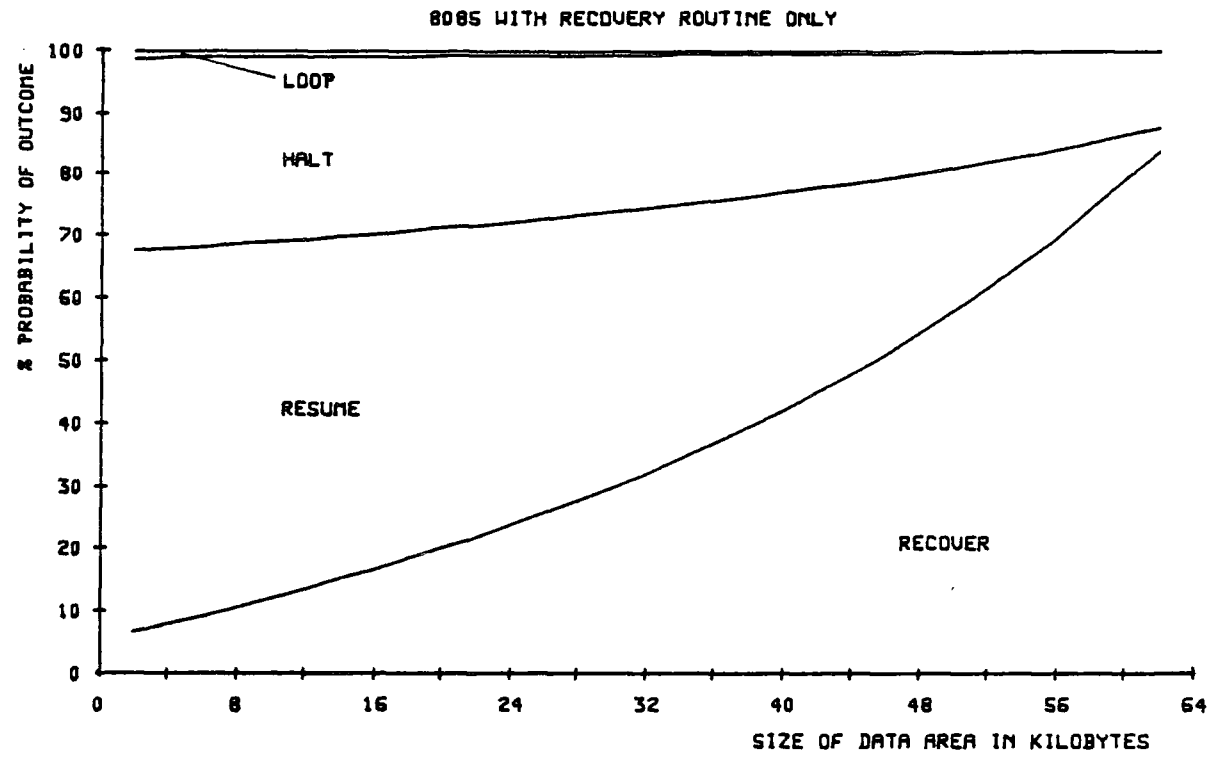


Figure 7.2 The Effects of Adding Fault Tolerance to the Program Areas of the 8085



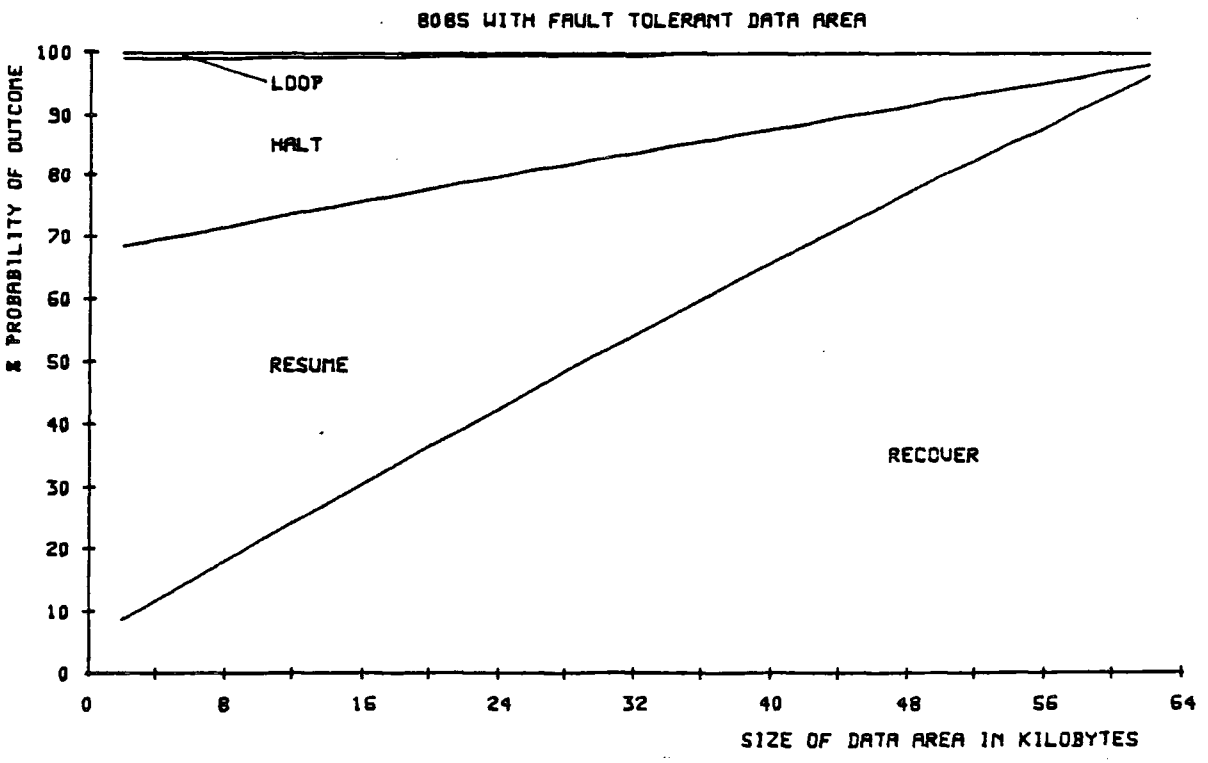


Figure 7.3 The Effects of Adding Fault Tolerance to the Data Areas of the 8085

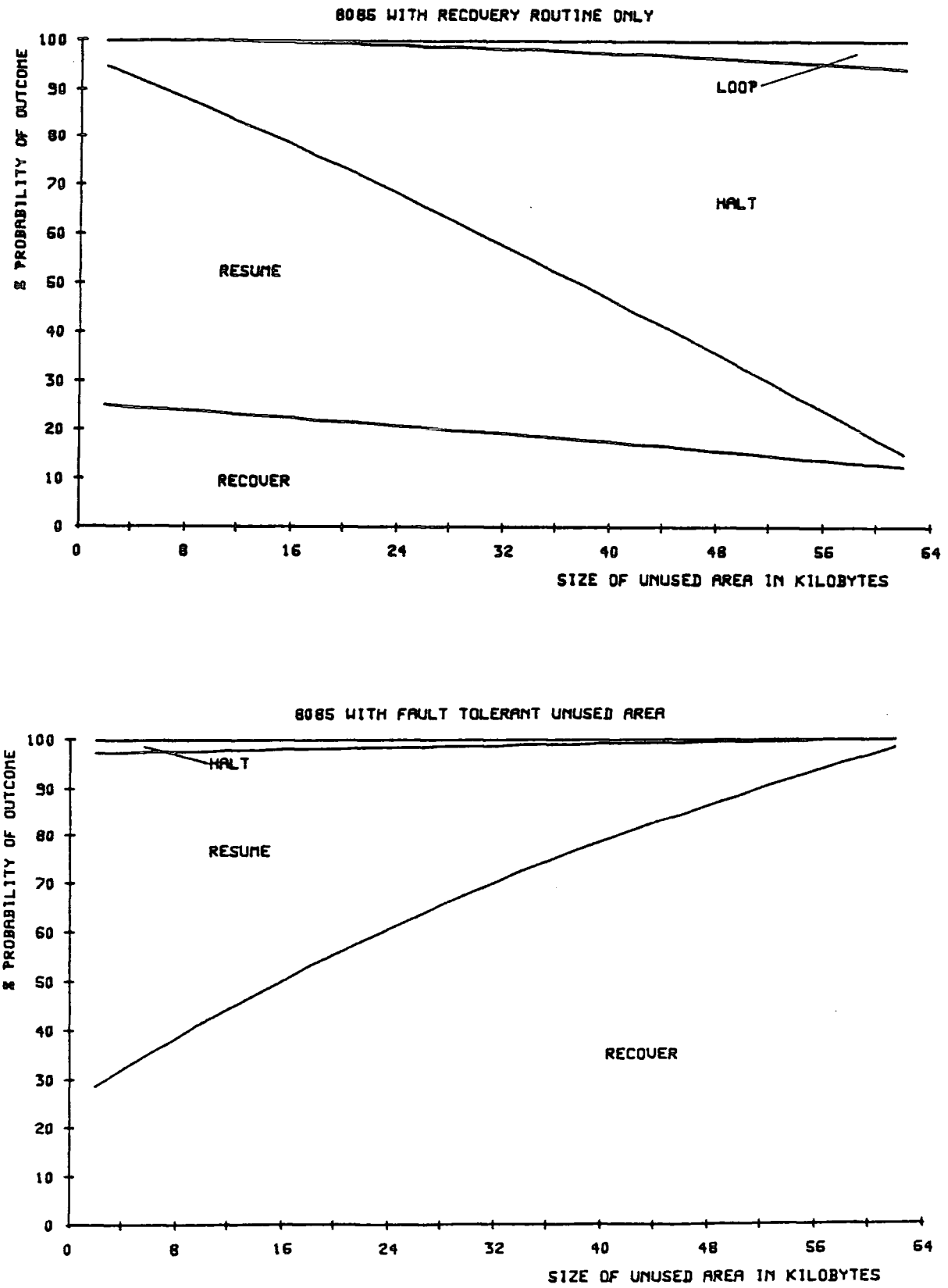
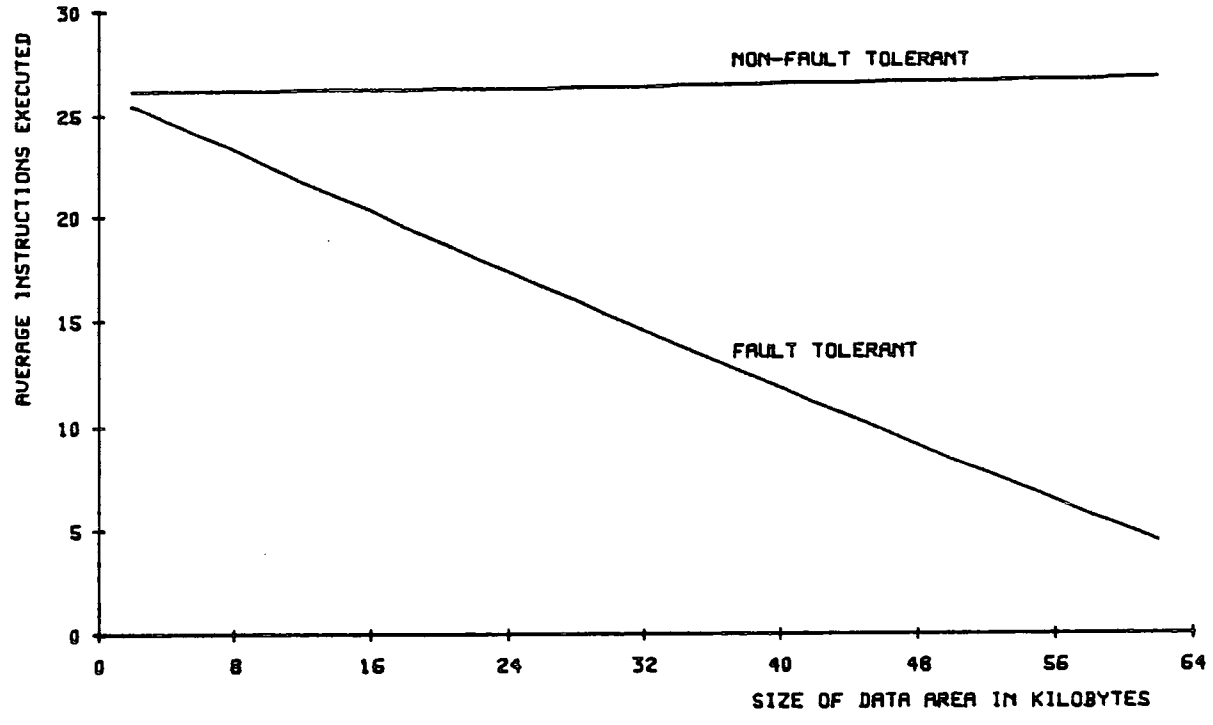


Figure 7.4 The Effects of Adding Fault Tolerance to the Unused Memory Areas of the 8085

(a) Results for Different Sizes of Data Areas



(b) Results for Different Sizes of Unused Areas

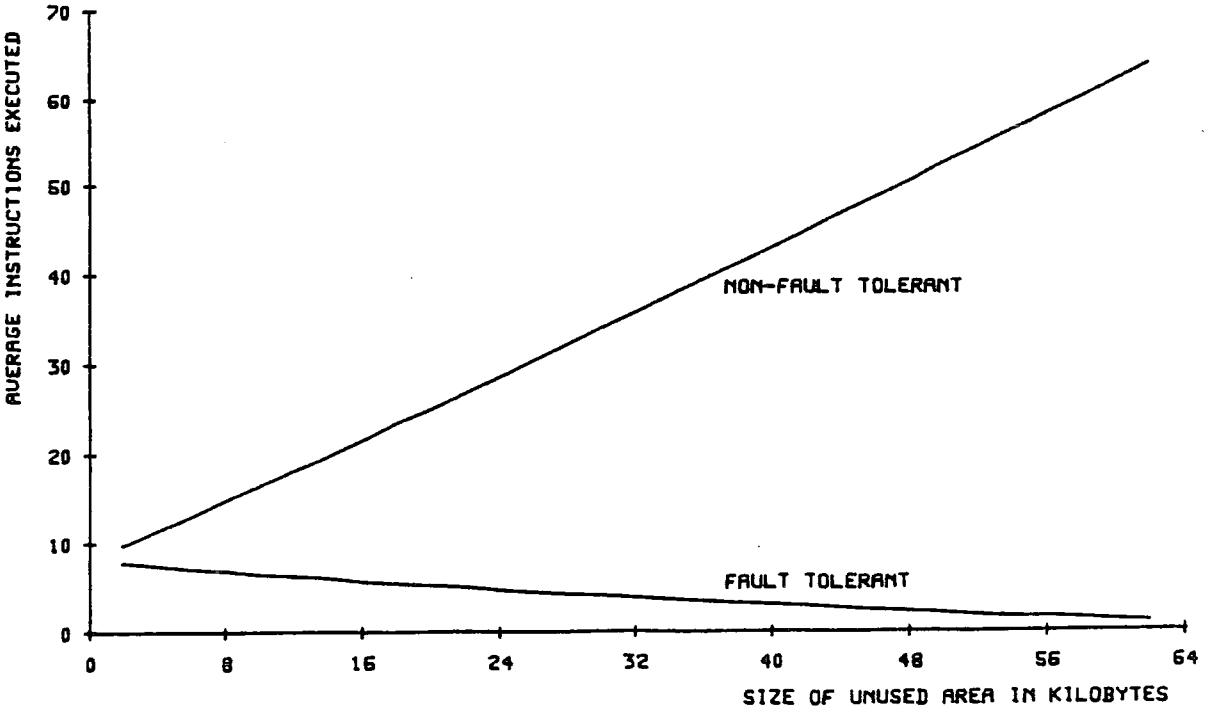
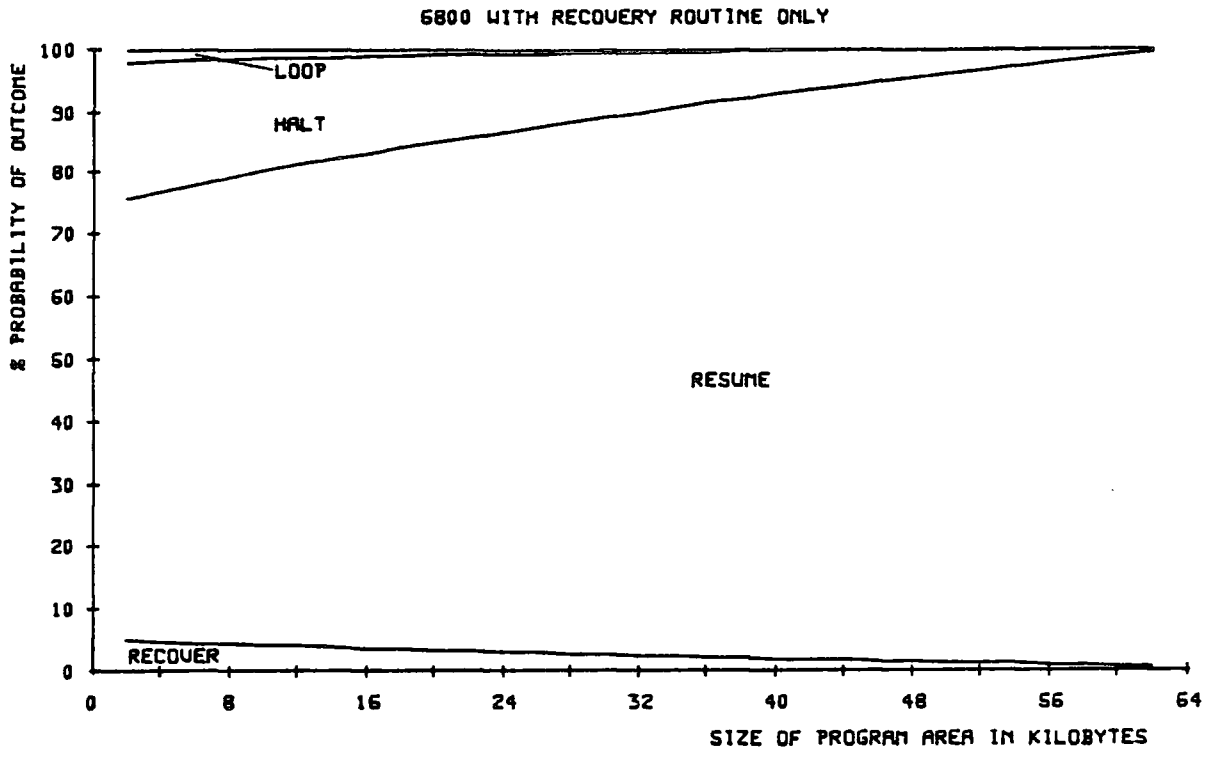


Figure 7.5 The Effects on the Average Number of Instructions Executed by Adding Fault Tolerance to the 8085



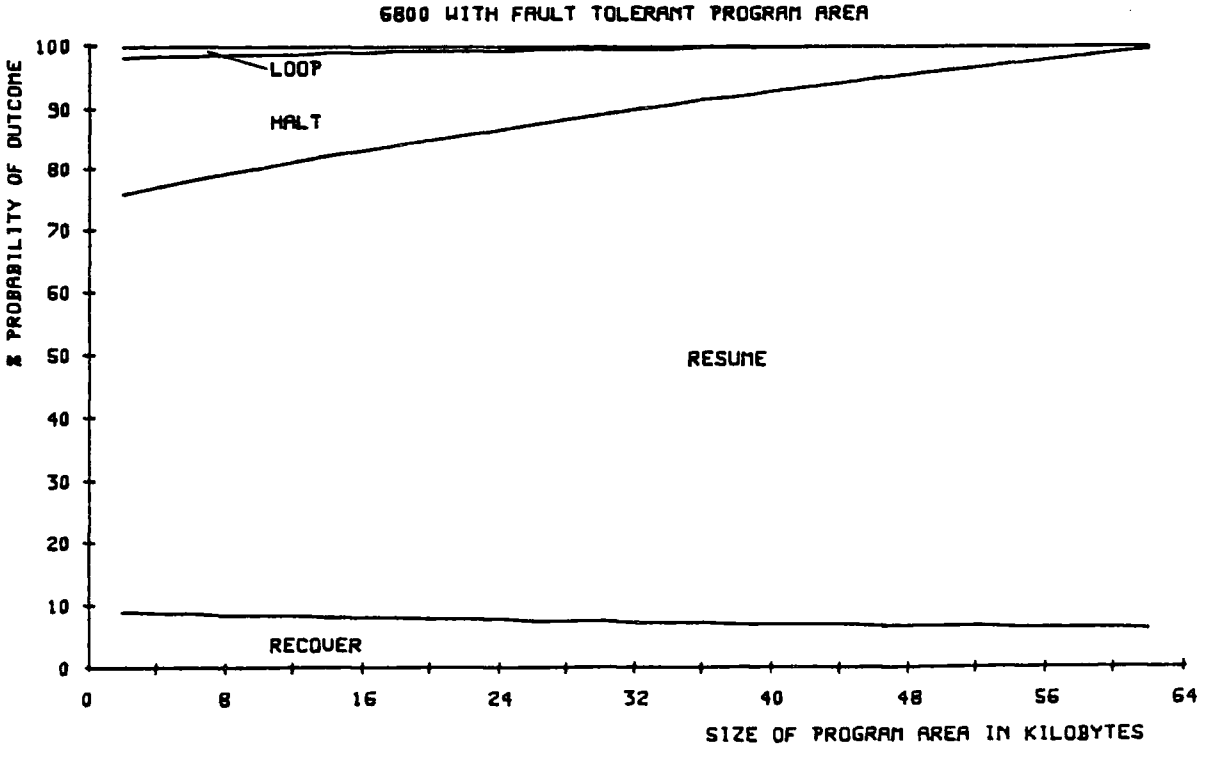


Figure 7.6 The Effects of Adding Fault Tolerance to the Program Areas of the 6800

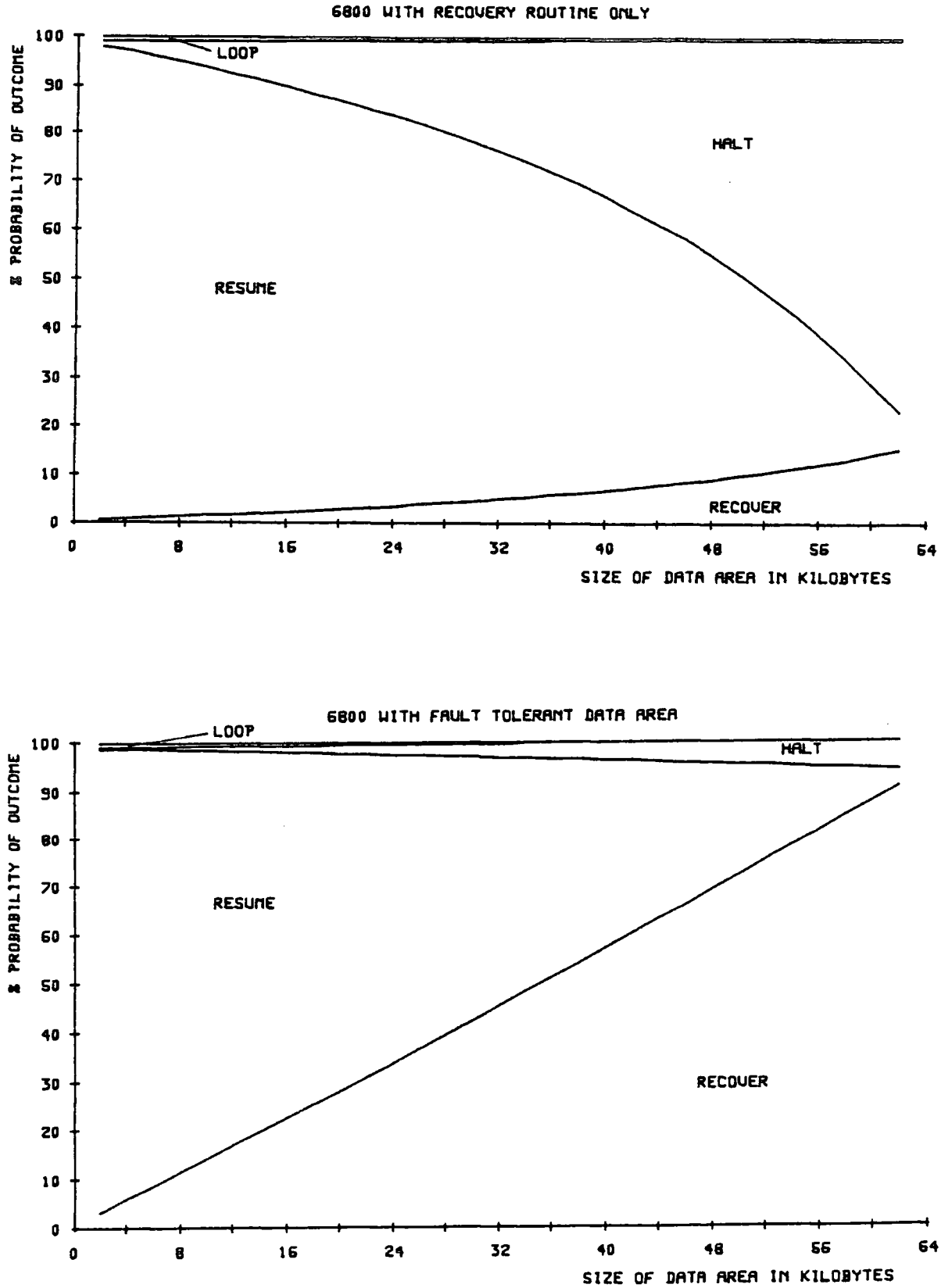


Figure 7.7 The Effects of Adding Fault Tolerance to the Data Areas of the 6800

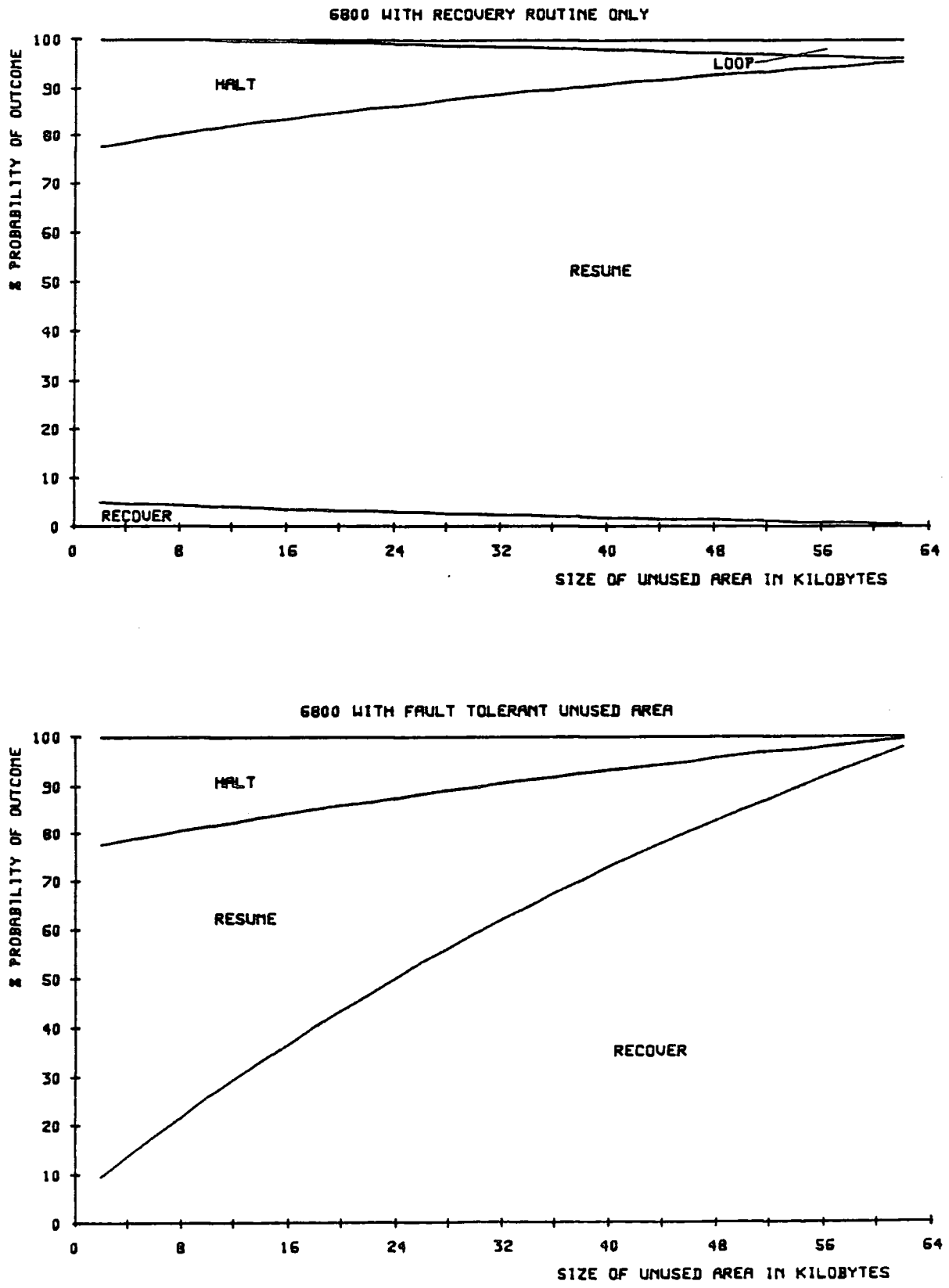


Figure 7.8 The Effects of Adding Fault Tolerance to the Unused Memory Areas of the 6800

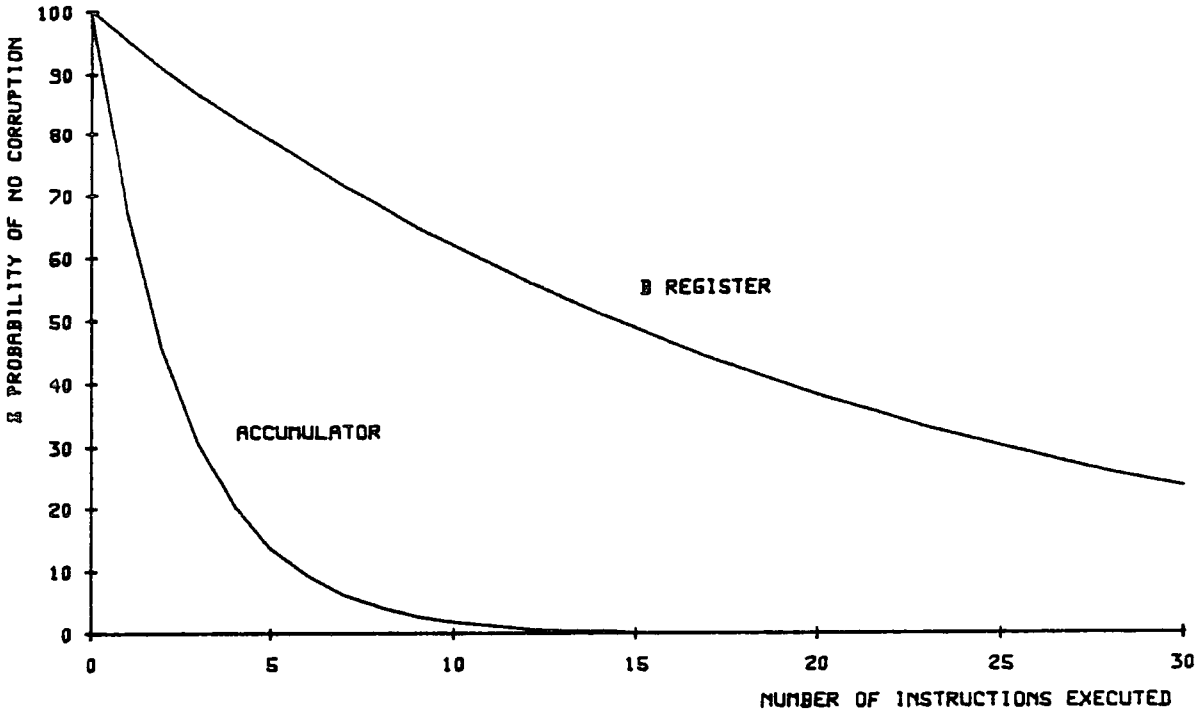


Figure 7.9 Probability of Data Corruptions in the 8085

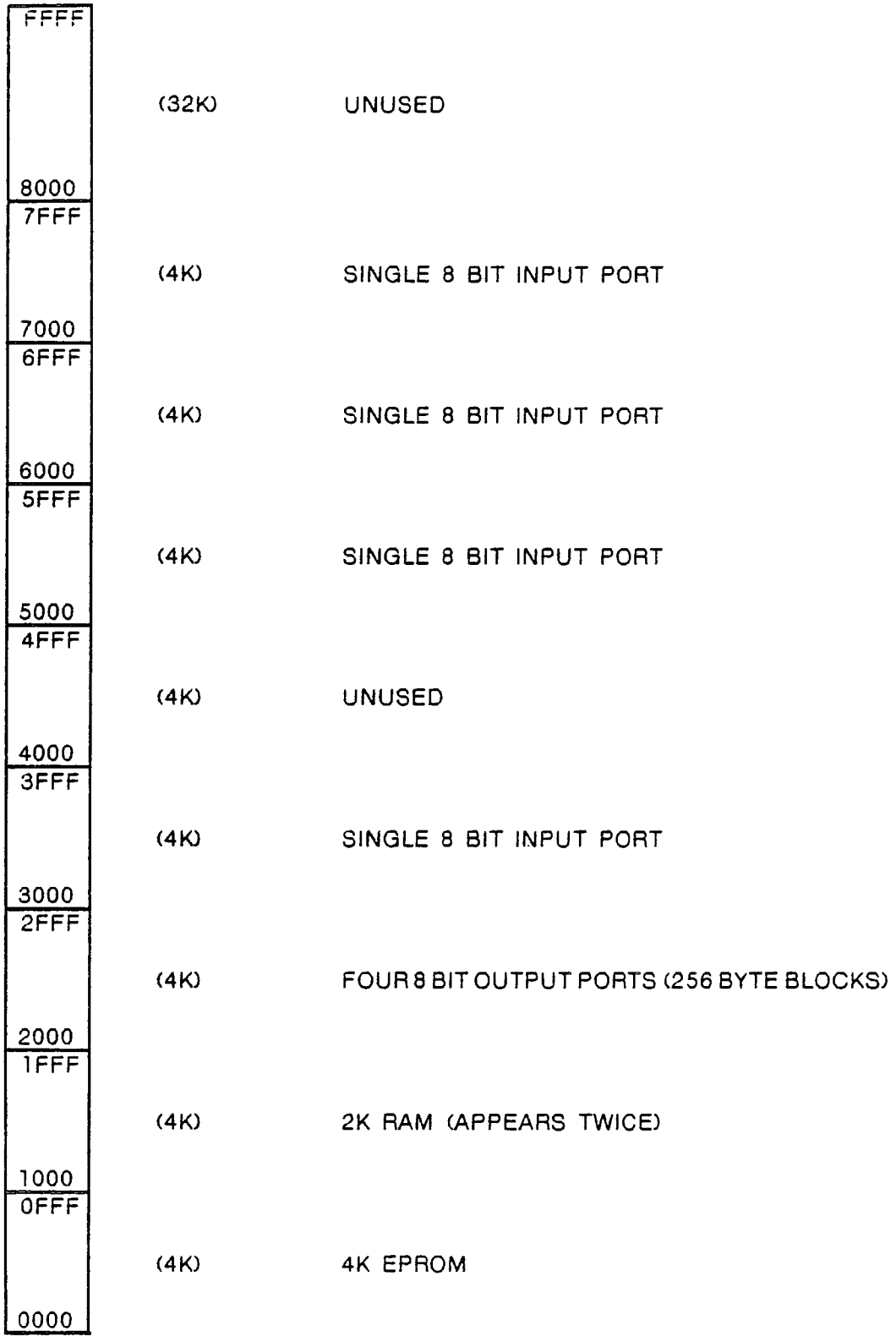


Figure 8.1 Memory Map of the Specific System Studied

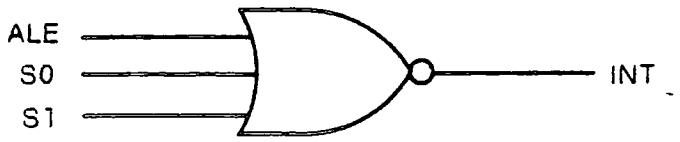


Figure 8.2 Wait State Recognition Circuit

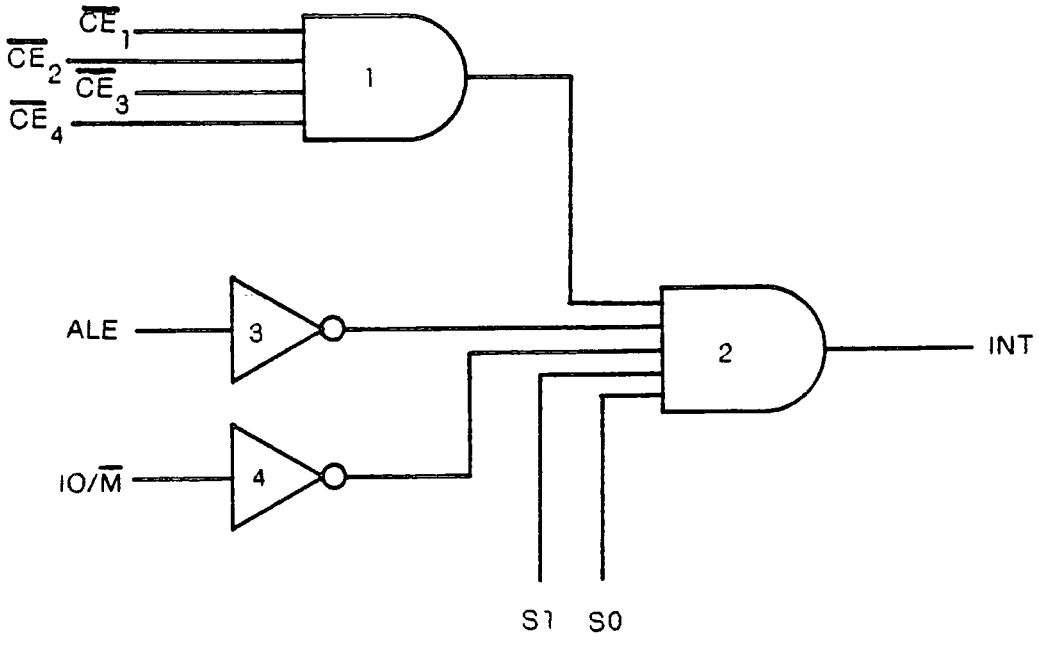


Figure 8.3 Circuit to Detect an Illegal Instruction Fetch

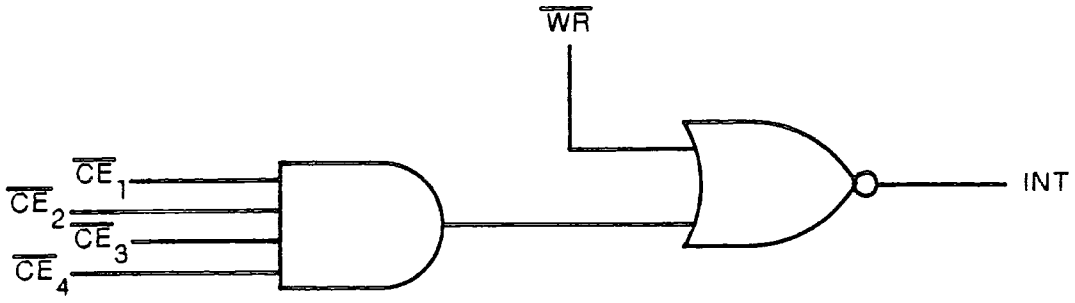


Figure 8.4 Circuit to Detect a Write into ROM

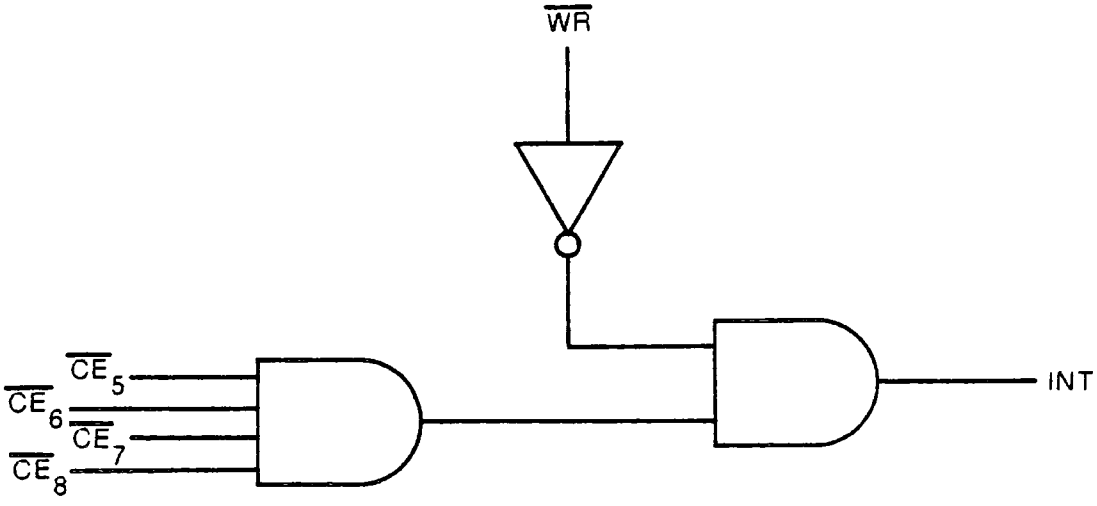


Figure 8.5 Circuit to Detect a Write Outside the RAM Areas

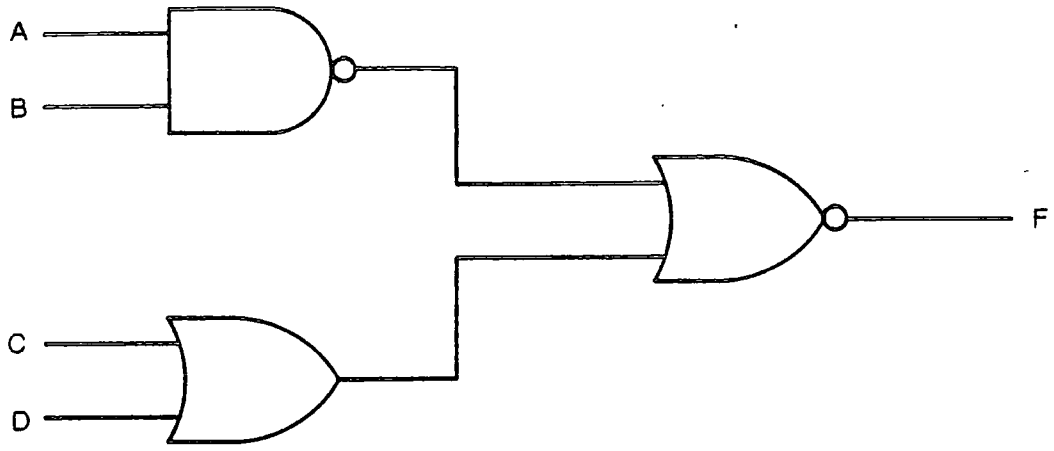


Figure 9.1 Logic Required to Detect Operation Code Fetches

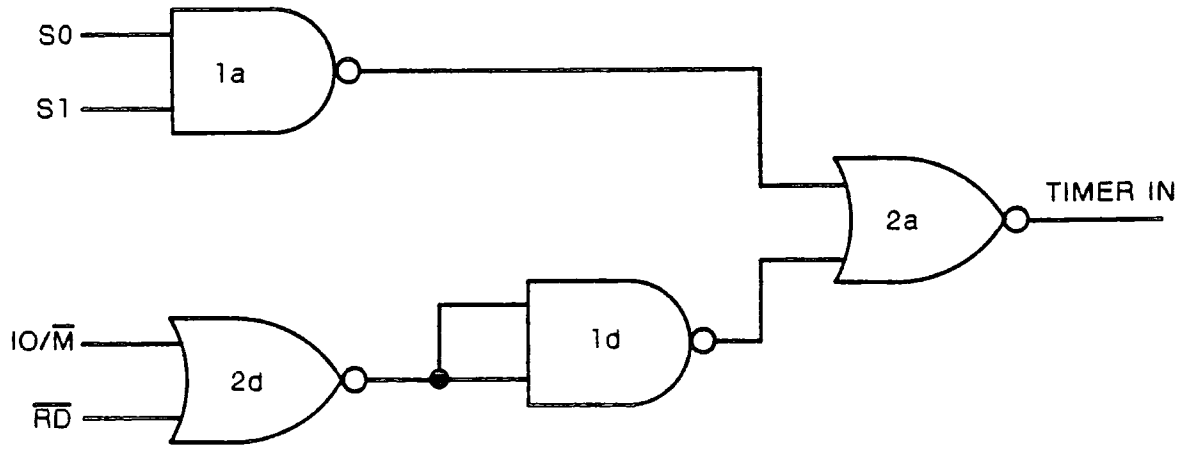


Figure 9.2 Implementation of Logic on Test System

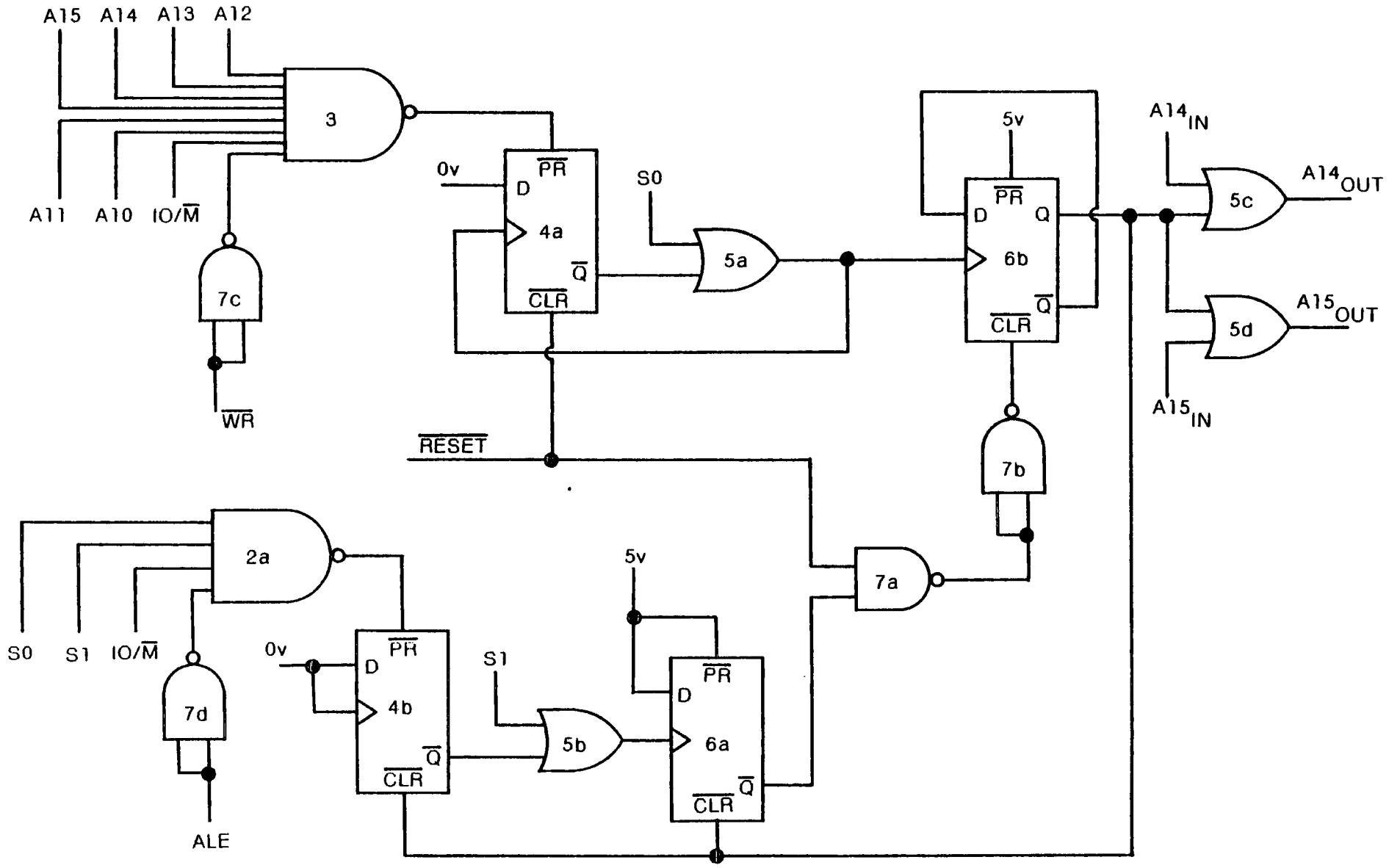


Figure 9.3 Circuit to Restrict Execution to 16K of Memory

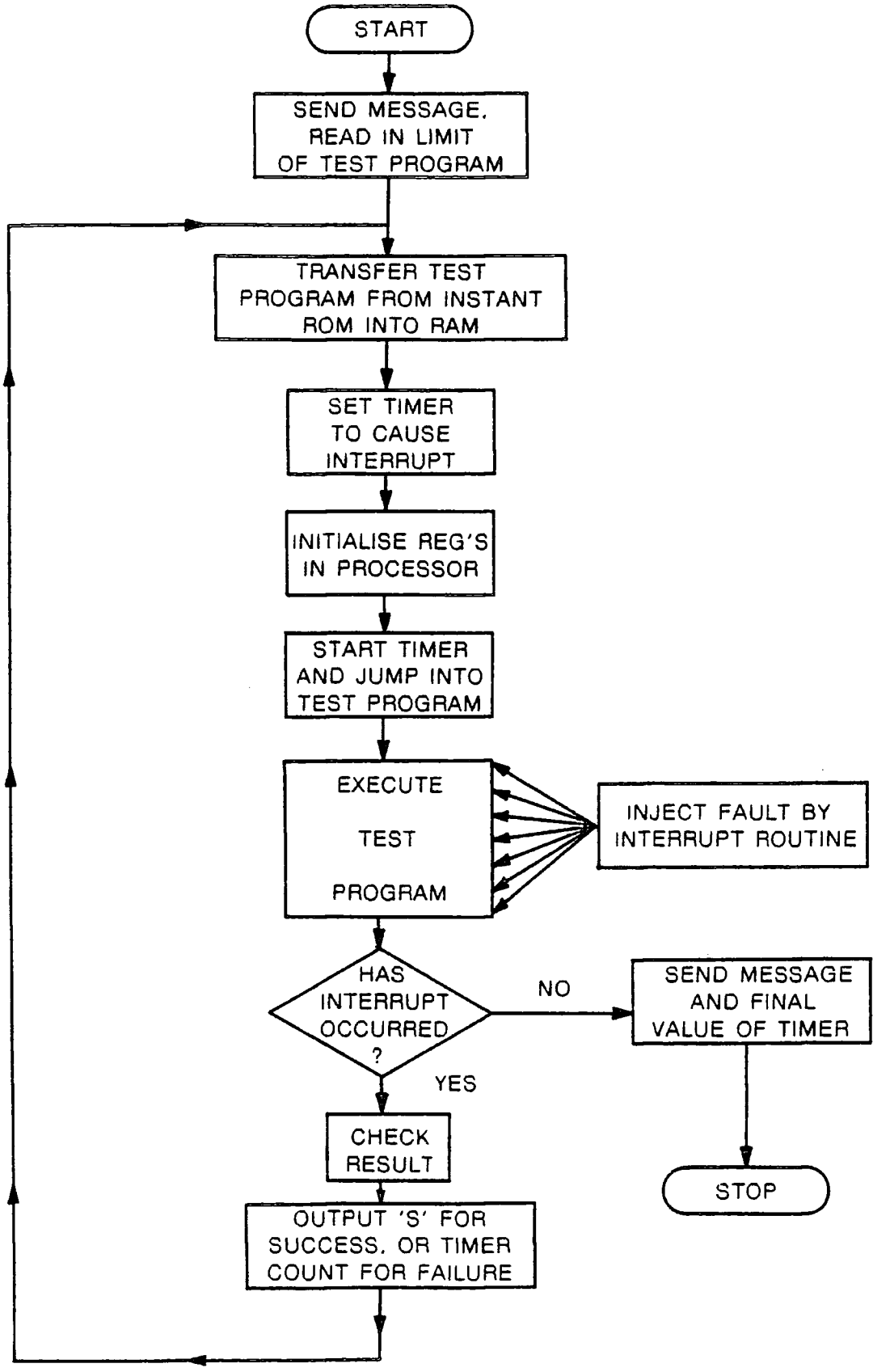


Figure 9.4 Software Flow Diagram for the Fault Injecting Test Facility

Type of Application	Requirements	Reference
Batch Processing	Recovery time of between 10 minutes and 2 hours	111
Communications	Recovery time of 1-15 minutes	111
Telephone Switching	Less than 2 hours down-time in 40 years Less than 2 calls lost in 10,000	25
Typical Industrial	Recovery within 250 milliseconds	87
Aerospace	Recovery within 10 milliseconds	111
Space	98% survivability over 5 years	106
Nuclear Reactor Safety System	10^{-6} - probability of failure on demand	12
Aircraft	10^{-8} - probability of failure during a 10 hour flight	110

Table 1.1 Reliability Requirements for Different Applications

DEVICE	ERROR TYPE		
	WRITE	DATA	READ
R3	3.33	1.13	3.31
R4	2.99	1.16	3.26
R5	2.61	1.10	2.61
R6	2.37	0.91	2.38

Table 2.1 Voltage Levels at which First Errors Occurred in 8155 RAM Chips

DEVICE	ERROR TYPE					
	WRITE		DATA		READ	
	LOCATION	VALUE	LOCATION	VALUE	LOCATION	VALUE
R3	VARIOUS	FF	VARIOUS	SINGLE BIT ERROR	FF00	FF
R4	VARIOUS	FF	VARIOUS	SINGLE BIT ERROR	VARIOUS	SINGLE BIT ERROR
R5	FF00	FF	VARIOUS	SINGLE BIT ERROR	FF00	FF
R6	FF00	FF	VARIOUS	SINGLE BIT ERROR	FF00	FF

Table 2.2 Location and Value of the First Errors Observed

DATA									ADDRESS
HEX	BINARY								HEX
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
00	0	0	0	0	0	0	0	0	FFBF
11	0	0	0	1	0	0	0	1	FFBF
22	0	0	1	0	0	0	1	0	FFBF
33	0	0	1	1	0	0	1	1	FFBF
88	1	0	0	0	1	0	0	0	FFBF
99	1	0	0	1	1	0	0	1	FFBF
AA	1	0	1	0	1	0	1	0	FFBF
BB	1	0	1	1	1	0	1	1	FFBF
44	0	1	0	0	0	1	0	0	FF3F
66	0	1	1	0	0	1	1	0	FF3F
CC	1	1	0	0	1	1	0	0	FF3F
EE	1	1	1	0	1	1	1	0	FF3F
55	0	1	0	1	0	1	0	1	FFFA
DD	1	1	0	1	1	1	0	1	FFFA
77	0	1	1	1	0	1	1	1	FFEC
FF	1	1	1	1	1	1	1	1	FF4B

0 1 - FIRST BITS CORRUPTED

Table 2.3 First Data Corruptions in RAM Chip R5

DEVICE	SIZE OF CAPACITOR IN TEST SUPPLY			MINIMUM VOLTAGE REACHED
	2.200 uF	4.700 uF	10.000 uF	
	Cycles	Cycles	Cycles	Volts
RAM	1.50	3.25	7.25	3.8
EPROM	1.75	3.25	7.75	3.4
PROCESSOR	2.00	4.25	9.25	2.8
COMPLETE SYSTEM	1.50	3.50	7.25	3.8

Table 2.4 Length of Interruptions to the Test Supply (in Cycles) Necessary to Cause Corruptions

DEVICE	RAM	ROM	EPROM
8035	64x8	NONE	----
8039	128x8	NONE	----
8040	256x8	NONE	----
8048	64x8	1Kx8	----
8049	128x8	2Kx8	----
8050	256x8	4Kx8	----
8748	64x8	----	1Kx8
8749	128x8	----	2Kx8

Table 3.1 Internal Memory of the 48-Series Microprocessors

PROCESSOR	PROBABILITY OF A JUMP (P_J)	AVERAGE NUMBER OF INSTRUCTIONS EXECUTED (NI_{AV})	AVERAGE NUMBER OF BYTES EXECUTED (NB_{AV})
8085	0.1035	9.65	12.5
6800	0.1035	9.65	18.5
8048 (INTEL)	0.1543	6.48	8.3
8048 (NEC)	0.1621	6.16	7.9
68000	0.3436	2.91	---

Table 4.1 Results of Execution in Random Data

PROCESSOR	BLOCK SIZE OF DATA	LENGTH OF RECOVERY STRING	% OVERHEAD	AVERAGE NUMBER OF INSTRUCTIONS EXECUTED	% PROBABILITY OF OUTCOME			
					HALT	RESTART (RECOVERY)	RANDOM JUMP	RETURN
8085	20	4	20	5.0	1.9	68.4	21.3	8.4
8085	15	3	20	4.5	1.5	73.8	17.8	6.9
8085	10	2	20	3.9	1.3	77.9	14.6	6.2
8085	5	1	20	3.2	0.9	84.1	10.6	4.4
6800	15	3	20	3.7	6.0	70.7	19.5	3.8
6800	10	2	20	3.3	4.7	74.1	17.1	4.1
6800	5	1	20	3.5	5.4	72.1	18.6	3.9
8048	15	3	20	4.1	0.0	42.7	54.4	2.9
8048	10	2	20	4.1	0.0	46.3	51.0	2.7

Table 4.2 Comparison Between Different Data Structures

PROCESSOR	% HALT	% RESTART	% RANDOM JUMP	% RETURN	% RESUME	AV. No. INSTRUCTIONS EXECUTED BEFORE ANY TRANSFER	AV No. INSTRUCTIONS EXECUTED BEFORE RESUMING PROGRAM
8085	0.1	1.0	1.4	0.6	96.9	1.2	0.3
6800	1.6	0.3	5.2	1.3	91.6	1.7	0.7
8048	0.0	0.0	3.5	0.2	96.3	1.0	0.2

Table 5.1 Comparison Between Processors for Erroneous Execution in Program Areas

PROCESSOR	PROGRAM	% HALT	% RESTART	% RANDOM JUMP	% RETURN	% RESUME	AV No. INSTRUCTIONS EXECUTED BEFORE ANY TRANSFER	AV No. INSTRUCTIONS EXECUTED BEFORE RESUMING PROGRAM
8085	A	0.0	2.8	2.2	0.9	94.1	1.2	0.7
8085	B	0.1	1.5	1.8	1.0	95.6	1.2	0.7
6800	C	0.0	0.6	2.6	2.4	94.4	1.5	0.7
8048	D	0.0	0.0	4.1	0.3	95.6	1.1	0.3
8048	E	0.0	0.0	3.7	0.2	96.1	1.1	0.3
8048	F	0.0	0.0	4.0	0.3	95.7	1.0	0.3

Table 5.2 Comparison Between Actual Programs

PROCESSOR	PROGRAM	% HALT	% RESTART	% RANDOM JUMP	% RETURN	% RESUME	AV. No. INSTRUCTIONS EXECUTED BEFORE ANY TRANSFER	AV No. INSTRUCTIONS EXECUTED BEFORE RESUMING PROGRAM
8085	A	0.0	4.1	3.2	1.3	91.4	1.8	0.8
8085	B	0.2	2.4	2.8	1.6	93.0	1.8	0.8
6800	C	0.0	0.7	3.2	2.8	93.3	1.8	0.8
8048	D	0.0	0.0	5.5	0.4	94.1	1.4	0.4
8048	E	0.0	0.0	4.7	0.3	95.0	1.4	0.4
8048	F	0.0	0.0	5.0	0.4	94.6	1.3	0.3
68000	G	0.0	26.1	1.5	0.0	72.4	1.5	0.5

Table 5.3 Results from the Simplified Analysis of Erroneous Execution in Program Areas

PROCESSOR	PROGRAM	% HALT	% RESTART	% RANDOM JUMP	% RETURN	% RESUME	AV. No. INSTRUCTIONS EXECUTED BEFORE ANY TRANSFER	AV No. INSTRUCTIONS EXECUTED BEFORE RESUMING PROGRAM
8085	A*	0.0	8.6	2.2	0.9	88.3	1.3	0.6
8085	B*	0.1	19.1	1.8	1.0	78.0	1.3	0.5
6800	C*	0.0	5.9	2.4	2.2	89.5	1.4	0.6

Table 5.4 Detailed Analysis of Modified Programs

TYPE OF TRANSFER	% PROBABILITY	AVERAGE NUMBER OF INSTRUCTIONS EXECUTED
HALT	47.7	54.6
RESTART	5.8	1.6
RANDOM JUMP	3.0	3.75
RETURN	35.0	33.6
SPECIFIC JUMP	8.5	2.2
ALL	100.0	38.2

Table 6.1 Probability of Different Outcomes after a Random Jump into an Unused Memory Area of an 8085

TYPE OF TRANSFER	% PROBABILITY	AVERAGE NUMBER OF INSTRUCTIONS EXECUTED
HALT	49.7	55.7
RESTART	7.2	2.3
RANDOM JUMP	5.0	5.0
RETURN	38.1	31.3
ALL	100.0	40.0

Table 6.2 Outcomes after a Random Jump into an Unused Memory Area of an 8085, Assuming Address Range C000 to FFFF is Unused

MEMORY ARRANGEMENT (see fig. 6.1)	STATE OF MEMORY BANK SELECT FLIP-FLOP AFTER ERROR	% PROBABILITY OF TRANSFER		
		JUMP OUT OF UNUSED AREA	RETURN	LOOP
A	0	49.8	49.2	1.0
A	1	0.0	99.0	1.0
B	0	90.5	9.4	0.1
B	1	29.2	69.8	1.0
C	0	90.9	9.0	0.1
C	1	89.8	9.2	1.0
D	0	89.8	9.2	1.0
D	1	90.9	9.0	0.1

Table 6.3 Transfer from Unpopulated Memory Areas of an 8048

PROCESSOR	% HALT	% RESTART	% RANDOM JUMP	% RETURN	% SPECIFIC JUMP	% EXIT FROM BLOCK
8085	0.4	3.3	0.4	2.0	4.3	89.6
6800	2.0	0.4	1.2	1.6	1.2	93.6
68000	0.0	32.5	1.9	0.0	0.0	65.6

Table 6.4 Transfer from Partially Decoded Memory Mapped Input Ports

LOCATION OF DATA	NUMBER OF INSTRUCTIONS WHICH CAUSE CORRUPTION	PROBABILITY THAT A SINGLE INSTRUCTION WILL NOT CORRUPT DATA
ACCUMULATOR	84	0.672
B REGISTER	12	0.953
C REGISTER	14	0.945
D REGISTER	16	0.938
E REGISTER	18	0.930
H REGISTER	22	0.914
L REGISTER	24	0.906
STACK POINTER	30.5	0.881
MEMORY	34.5	0.865
ALL FLAGS	105	0.590
SIGN FLAG	45.5	0.822
ZERO FLAG	45.5	0.822
AUXILIARY CARRY FLAG	45.5	0.822
PARITY FLAG	45.5	0.822
CARRY FLAG	44.5	0.826

Table 7.1 Data Corruptions in the 8085 Caused by Erroneous Execution

SYSTEM ARRANGEMENT	NON-MEMORY MAPPED PORTS	RECOVERY ROUTINE	PULL-UPS ON DATA LINES	MODIFIED PROGRAM AREA	FULL RAM DECODING	SEEDED DATA AREA	FINAL OUTCOME REACHED				EXPECTED NUMBER OF ERRONEOUS INSTRUCTIONS EXECUTED	90% CONFIDENCE LIMIT ON THE NUMBER OF INSTRUCTIONS EXECUTED
							% HALT	% LOOP	% RESUME	% RECOVER		
A							65.2	10.5	24.3	0.0	1045.3	2406.9
B	✓						69.2	7.8	23.0	0.0	56.6	130.3
C		✓					67.2	7.7	9.9	15.2	1055.6	2430.6
D		✓	✓				0.4	0.4	6.4	93.0	682.2	1570.8
E	✓	✓	✓				0.3	0.0	6.2	93.5	1.6	3.7
F	✓	✓	✓	✓			0.3	0.0	5.1	94.7	1.6	3.7
G	✓	✓	✓		✓		0.1	0.0	6.1	93.8	1.3	3.0
H	✓	✓	✓		✓	✓	0.0	0.0	6.0	94.0	1.1	2.5
I	✓	✓	✓	✓	✓	✓	0.0	0.0	4.9	95.1	1.1	2.5

Table 8.1 Erroneous Execution Under Different System Arrangements

Appendix 1. Software to Test the Effects of Executing Undeclared Operation Codes

This appendix contains full commented listings of the programs used to identify the effects of executing the undeclared operation codes of the 6800 and 8035/8048. Similar techniques, as those illustrated, can be employed on other microprocessors. However, the software alone is not usually sufficient to identify all functions, and it is necessary to use additional techniques such as the monitoring of all external signals with a logic analyser.

A1.1 Listing of the 6800 Test Program

```
                        NAM     M6800
                *****************************************************************
                *
                *               *****M6800.ASM*****
                *
                *****************************************************************
                *
                *   THIS PROGRAM IS DESIGNED TO TEST THE UNDECLARED
                *   OP-CODES IN THE MOTOROLA 6800 MICROPROCESSOR
                *
                *   IT ALSO TESTS IF THE INTERRUPTS ARE DISABLED BY THEM
                *
                *---------------------------------------------------------
                *
     E1D1       OUTEEE  EQU     $E1D1   ROUTINE TO OUTPUT A CHARACTER
     E055       BYTE    EQU     $E055   READS IN A BYTE OF DATA IN HEX
     E0E3       CONTRO  EQU     $E0E3   ENTRY POINT INTO MIKBUG
     E07E       PDATA1  EQU     $E07E   ROUTINE TO OUTPUT A STRING
                *
1FF0                    ORG     $1FF0   SET STACK LOCATIONS
1FF0 0001       STACK   RMB     1       POSITION OF TOP OF STACK
1FF1 0001       CTEMP   RMB     1       SPACE FOR CONDITION CODES
1FF2 0001       BTEMP   RMB     1       SPACE FOR ACCUMULATOR B
1FF3 0001       ATEMP   RMB     1       SPACE FOR ACCUMULATOR A
1FF4 0001       XTEMPH  RMB     1       HIGH BYTE OF X REGISTER
1FF5 0001       XTEMPL  RMB     1       LOW BYTE OF X REGISTER
1FF6 0002       PTEMP   RMB     2       SPACE FOR RETURN ADDRESS
                *
A048                    ORG     $A048   SET START ADDRESS FOR MIKBUG GO
A048 0100       GOADD   FDB     $0100   COMMAND
A000                    ORG     $A000
A000 0200               FDB     IRQVEC  SET VECTOR FOR IRQ
A006                    ORG     $A006
A006 0210               FDB     NMIVEC  SET VECTOR FOR NMI
```

6800 Test Program (cont.)

```
                        ORG     $0100     START OF TEST PROGRAM
                *
0100 CE 0137            LDX     ^RESU     LOAD ADDRESS TO GO TO AFTER RTI
0103 FF 1FF6            STX     PTEMP     STORE VALUE ON STACK
0106 BD 0142            JSR     CRLF      SET TERMINAL ON NEW LINE
0109 BD E055            JSR     BYTE      READ IN BYTE FOR CONDITION CODES
010C B7 1FF1            STAA    CTEMP     STORE ONTO STACK
010F BD 014D            JSR     SPACE
0112 BD E055            JSR     BYTE      READ IN BYTE FOR ACCUMULATOR B
0115 B7 1FF2            STAA    BTEMP     STORE ONTO STACK
0118 BD 014D            JSR     SPACE
011B BD E055            JSR     BYTE      READ IN BYTE FOR ACCUMULATOR A
011E B7 1FF3            STAA    ATEMP     STORE ONTO STACK
0121 BD 014D            JSR     SPACE
0124 BD E055            JSR     BYTE      READ IN HIGH BYTE OF X REGISTER
0127 B7 1FF4            STAA    XTEMPH    STORE ONTO STACK
012A BD E055            JSR     BYTE      READ IN LOW BYTE OF X REGISTER
012D B7 1FF5            STAA    XTEMPL    STORE ONTO STACK
0130 8E 1FF0            LDS     ^STACK    LOAD STACK TO POINT TO DATA BLOCK
0133 3B                 RTI               LOAD REGS AND JUMP TO TEST LOC
0134 01         START   NOP               NOPS IN LOOP TO WAIT FOR INTERRUPT
0135 01                 NOP
0136 01                 NOP
0137 01         RESU    NOP               TEST BYTE CAN BE INSERTED BY HAND
0138 01                 NOP               IN ONE OF THESE LOCATIONS
0139 01                 NOP
013A 01                 NOP
013B 20 F7              BRA     START     LOOP UNTIL INTERRUPT
013D 3F         TEST1   SWI
013E 3F                 SWI               STRING OF SOFTWARE INTERRUPTS TO
013F 3F                 SWI               CAPTUREEXECUTIONAFTERTESTCODE
0140 3F                 SWI               NECESSARY FOR SINGLE, DOUBLE
0141 3F                 SWI               OR TRIPLE BYTE INSTRUCTIONS
                *
                *
                *   SUBROUTINES
                *
                *
0142 86 0D      CRLF    LDAA    ^$0D      SUBROUTINE TO OUTPUT A CARRIAGE
0144 BD E1D1            JSR     OUTEEE    RETURN AND LINE FEED TO THE
0147 86 0A              LDAA    ^$0A      TERMINAL
0149 BD E1D1            JSR     OUTEEE
014C 39                 RTS
                *
014D 86 20      SPACE   LDAA    ^$20      SUBROUTINE TO OUTPUT A SPACE
014F BD E1D1            JSR     OUTEEE    TO THE TERMINAL
0152 86 20              LDAA    ^$20
0154 BD E1D1            JSR     OUTEEE
0157 39                 RTS
```

6800 Test Program (cont.)

```
                    *
                    *    INTERRUPT SERVICE ROUTINES
                    *
0200                        ORG    $0200
                    *
0200 CE 0217   IRQVEC       LDX    #IRQSTR   LOAD START ADDRESS OF STRING
0203 BD E07E                JSR    PDATA1    PRINT STRING TO INDICATE IRQ
0206 3B                     RTI
                    *
                            ORG    $0210
                    *
0210 CE 021F   NMIVEC       LDX    #NMISTR   LOAD START ADDRESS OF STRING
0213 BD E07E                JSR    PDATA1    PRINT STRING TO INDICATE NMI
0216 3B                     RTI
                    *
0217 20        IRQSTR       FCC    / IRQ /        STRING PRINTED BY IRQ
0218 20
0219 49
021A 52
021B 51
021C 20
021D 20
021E 04                     FCB    $04            DELIMITER
021F 20        NMISTR       FCC    / NMI /        STRING PRINTED BY NMI
0220 20
0221 4E
0222 4D
0223 49
0224 20
0225 20
0226 04                     FCB    $04            DELIMITER
                    *
                            END
```

A1.2 Listing of the 8035/8048 Test Program

```
;*************************************************************************
;
; PROGRAM TO TEST THE UNDECLARED OPCODES OF THE 8035/8048
;
;*************************************************************************
;
; ALL UNUSED LOCATIONS ARE SET TO 04.  THIS FORCES A JUMP TO
; ADDRESS 004 IF PROGRAM EXECUTION IS ATTEMPTED OUTSIDE THE
; NORMAL PROGRAM AREA
;
000  64    JMP 0300        JUMP TO INITIALISATION BLOCK
001  00
002  04                    UNUSED LOCATIONS SET TO 04
003  04
004  39    OUTL P1,A       OUTPUT CONTENTS OF ACCUMULATOR TO PORT
005  83    RET             RETURN TO MAIN LOOP
006  04
007  04                    UNUSED LOCATIONS SET TO 04, CAUSES JUMP
008  04                    TO ADDRESS 004 IF EXECUTED
;
;                          MAIN PROGRAM LOOP
;
100  75    ENT0 CLK        SET T0 AS A CLOCK OUTPUT FOR LOGIC ANALYSER
101  17    INC  A          INCREMENT TEST BYTE IN ACCUMULATOR
102  54    CALL 0200       CALL ROUTINE TO EXECUTE UNDECLARED CODE
103  00
104  24    JMP  0101       JUMP BACK TO BEGINNING OF LOOP
105  01
106  04
107  04                    UNUSED LOCATIONS
;
;                          SUBROUTINE TO EXECUTE UNDECLARED CODE
;
200  39    OUTL P1,A       OUTPUT CONTENTS OF ACCUMULATOR TO PORT
201  XX                    SPACE FOR UNDECLARED CODE
202  04                    SEQUENCE OF LOCATIONS SET TO 04, THIS ENSURES
203  04                    THAT EXECUTION WILL TRANSFER TO LOCATION 004
204  04                    REGARDLESS OF WHETHER THE UNDECLARED CODE IS
205  04                    A SINGLE, DOUBLE OR TRIPLE BYTE INSTRUCTION
;
;                          CODE FOR INITIALISATION OF PROCESSOR ON RESET
;
300  23    MOV A,0AAH      SETS ACCUMULATOR TO THE VALUE AA
301  AA
302  00    NOP             SPACE FOR SETTING OTHER REGISTERS OR FLAGS
303  00    NOP
304  24    JMP  0100       JUMP TO BEGINNING OF MAIN LOOP
305  00
306  04                    UNUSED LOCATIONS SET TO 04
307  04
;
           END
```

Appendix 2. The Effects of Executing the Undeclared Operation Codes of the 8035/8048

Appendix 2 contains a detailed description of the operations performed by all the instruction codes which are not declared for the 8035/8048. They appear in numerical order and are referenced by their hexadecimal value. In cases where the code performs a different function for different manufacturers, this is clearly marked and both operations are described.

Symbols Used

The symbols used and the layout of the definitions is very similar to that used in the National Semiconductor 48-Series Microcomputers Handbook (120). Reference should be made to the handbook for descriptions of the standard instruction set.

Symbols	Description
A	The Accumulator
AC	The Auxiliary Carry Flag
addr	Program Memory Address
Bb	Bit Designator (b = 0-7)
BS	The Bank Switch
BUS	The Bus Port
C	Carry Flag
CLK	Clock Signal
CNT	Event Counter
D	Nibble Designator (4 bits)
data	Number or Expression (8 bits)
DBF	Memory Bank Flip-Flop
F0,F1	Flags 0,1
I	Interrupt
P	"In-Page" Operation Designator
Pp	Port Designator (p = 1,2 or 4-7)
PSW	Program Status Word
Rr	Register Designator (r = 0,1 or 0-7)
SP	Stack Pointer
T	Timer
TF	Timer Flag
T0,T1	Testable Inputs 0,1
X	External RAM
#	Prefix for Immediate data
@	Prefix for Indirect Address
(A)	Contents of Accumulator
((A))	Contents of Location Addressed by A
←	Replaced By

Operation Code:	01	01
Mnemonic:	NOP	
Operation:	No operation performed.	
Description:	No operation is performed; execution continues with the next sequential instruction	
Note:	Same operation as the defined instruction (code 00).	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Code:	06	06
Mnemonic:	JNTF addr	
Operation:	Jump to specified address if timer flag is clear.	
Symbolic Representation:	(PC 0-7) < addr if TF=0 (PC) < (PC) + 2 if TF=1	
Description:	If the internal timer/counter flag is set to a logic zero, the contents of the program counter are replaced by the address bits from byte 2. If the timer/counter flag is a logic one, the next sequential instruction is executed.	
Note:	This instruction is the logical inverse of the JTF instruction, except that the timer/counter flag is not affected.	
Special Conditions:	Cycles: 2 Bytes: 2	

Operation Code:	0B	0B
Mnemonic:	IN A,P2	
Operation:	Input data to accumulator from port 2	
Symbolic Representation:	(A) < (P2)	
Description:	Data present at port 2 is input into the accumulator.	
Note:	Same operation as the defined instruction (code 0A).	
Special Conditions:	Cycles: 2 Bytes: 1	

Operation Code: 22 **NEC 8048 ONLY** 22

Mnemonic: MOV A,PC+1

Operation: Move contents of the program counter into the accumulator and increment.

Symbolic Representation: (A) < (PC) + 1

Description: The contents of the program counter are moved to the accumulator and then the accumulator is incremented by one. After executing this instruction the accumulator contains the address of the next sequential instruction.

Note: This function is only performed by the processor manufactured by NEC.

Special Conditions: Cycles: 1
Bytes: 1

Operation Code: 22 **INTEL 8048 ONLY** 22

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction

Note: This code performs a no operation (code 00) on the 8048 manufactured by Intel. It performs a different function on the processor made by NEC.

Special Conditions: Cycles: 1
Bytes: 1

Operation Code: 33 33

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction.

Note: Same operation as the defined instruction (code 00).

Special Conditions: Cycles: 1
Bytes: 1

Operation Code:	38	38
Mnemonic:	BUS IDLE	
Operation:	No specific operation on the Bus	
Symbolic Representation:	(BUS) < 00	
Description:	The value 00 appears on the Bus during T4 of the second cycle of the instruction, but no read or write signal is generated. Therefore a valid Bus operation is not performed.	
Note:	This code does not appear to perform any useful function.	
Special Conditions:	Cycles: 2 Bytes: 1	

Operation Code:	3B	3B
Mnemonic:	OUTL P2,A	
Operation:	Output contents of accumulator to port 2.	
Symbolic Representation:	(P2) < (A)	
Description:	The contents of the accumulator are placed, and latched, at the output port 2.	
Note:	Same operation as the defined instruction (code 3A).	
Special Conditions:	Cycles: 2 Bytes: 1	

Operation Code:	63	63
Mnemonic:	NOP	
Operation:	No operation performed.	
Description:	No operation is performed; execution continues with the next sequential instruction.	
Note:	Same operation as the defined instruction (code 00).	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Code:	66	66
Mnemonic:	JNF1 addr	
Operation:	Jump to specified address if flag 1 is clear.	
Symbolic Representation:	(PC 0-7) < addr, if F1=0 (PC) < (PC) + 2, if F1=1	
Description:	If flag 1 is at a logic zero, the contents of the program counter are replaced by the address bits from byte 2. If flag 1 is a logic one, the next sequential instruction is executed.	
Note:	This instruction is the logical inverse of the JF1 instruction.	
Special Conditions:	Cycles: 2 Bytes: 2	

Operation Codes:	73,82	73,82
Mnemonic:	NOP	
Operation:	No operation performed.	
Description:	No operation is performed; execution continues with the next sequential instruction	
Note:	Same operation as the defined instruction (code 00).	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Code:	87 **NEC 8048 ONLY**	87
Mnemonic:	CLR A4-7	
Operation:	Clear accumulator high nibble.	
Symbolic Representation:	(A4-7) < 0	
Description:	Accumulator bits 4 through 7 are cleared to zero.	
Note:	This function is only performed by the processor manufactured by NEC.	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Code: 87 **INTEL 8048 ONLY** 87

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction

Note: This code performs a no operation (code 00) on the 8048 manufactured by Intel. It performs a different function on the processor made by NEC.

Special Conditions: Cycles: 1
Bytes: 1

Operation Code: 8B 8B

Mnemonic: ORL p2,#data

Operation: Logical-OR-immediate specified data with contents of port 2.

Symbolic Representation: (P2) ‹ (P2) OR data

Description: The data contained in byte 2 is logically ORed with the data on port 2, and the results are sent back to the port.

Note: Same operation as the defined instruction (code 8A).

Special Conditions: Cycles: 2
Bytes: 2

Operation Code: 9B 9B

Mnemonic: ANL P2,#data

Operation: Logical-AND-immediate specified data with port 2.

Symbolic Representation: (P2) ‹ (P2) AND data

Description: The data contained in byte 2 are logically ANDed immediately with the data on port 2, and the results are sent back to the port.

Note: Same operation as the defined instruction (code 9A).

Special Conditions: Cycles: 2
Bytes: 2

Operation Code:	A2	A2
Mnemonic:	NOP	
Operation:	No operation performed.	
Description:	No operation is performed; execution continues with the next sequential instruction	
Note:	Same operation as the defined instruction (code 00).	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Code:	A6	A6
Mnemonic:	JNF0 addr	
Operation:	Jump to specified address if flag 0 is clear.	
Symbolic Representation:	(PC 0-7) < addr, if F0=0 (PC) < (PC) + 2, if F0=1	
Description:	If flag 0 is at a logic zero, the contents of the program counter are replaced by the address bits from byte 2. If flag 0 is at a logic one, the next sequential instruction is executed.	
Note:	This instruction is the logical inverse of the JF0 instruction.	
Special Conditions:	Cycles: 2 Bytes: 2	

Operation Code:	B7	B7
Mnemonic:	NOP	
Operation:	No operation performed.	
Description:	No operation is performed; execution continues with the next sequential instruction	
Note:	Same operation as the defined instruction (code 00).	
Special Conditions:	Cycles: 1 Bytes: 1	

Operation Codes:	C0,C1	**NEC 8048 ONLY**	C0,C1

Mnemonic: DEC @ Rr

Operation: Decrement-indirect contents of RAM by one.

Symbolic Representation: ((Rr)) < ((Rr)) + 1; where r = 0 or 1

Description: The contents of the internal RAM location as addressed by bits 0 through 5 of register 'r', are decremented by one.

Note: This function is only performed by the processor manufactured by NEC.

Special Conditions: Cycles: 1
Bytes: 1

Operation Codes:	C0,C1	**INTEL 8048 ONLY**	C0,C1

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction.

Note: These codes perform a no operation (code 00) on the 8048 manufactured by Intel. They perform a different function on the processor made by NEC.

Special Conditions: Cycles: 1
Bytes: 1

Operation Codes:	C2,C3	C2,C3

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction

Note: Same operation as the defined instruction (code 00).

Special Conditions: Cycles: 1
Bytes: 1

Operation Code:	D6	D6
Mnemonic:	JMPP addr	
Operation:	Jump to specified address within address page.	
Symbolic Representation:	(PC 0–7) < addr	
Description:	The contents of the program counter are replaced by the address bits from byte 2.	
Note:	Performs an unconditional jump within the current address page. This operation is not provided in the standard instruction set.	
Special Conditions:	Cycles: 2 Bytes: 2	

Operation Codes:	E0,E1 **NEC 8048 ONLY**	E0,E1
Mnemonic:	DJNZ @ Rr addr	
Operation:	Decrement-indirect contents of RAM, test contents, jump if not zero.	
Symbolic Representation:	((Rr)) < ((Rr)) – 1; where r = 0 or 1 (PC 0–7) < addr, if ((Rr)) = 0 (PC) < (PC) + 2, if ((Rr)) = 0	
Description:	The contents of the internal RAM location, as addressed by bits 0 through 5 of register r, are decremented by one, and then tested to see if the contents equal zero. If the contents of the location equal zero, the next sequential instruction is executed. If the location is not zero, control passes to the instruction at the address designated in byte 2.	
Note:	This function is only performed by the processor manufactured by NEC, and provides an operation which is not available from the standard instruction set.	
Special Conditions:	Cycles: 2 Bytes: 2	

Operation Codes:	E0,E1	**INTEL 8048 ONLY**	E0,E1

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction

Note: These codes perform a no operation (code 00) on the 8048 manufactured by Intel. They perform a different function on the processor made by NEC.

Special Conditions:
Cycles: 1
Bytes: 1

Operation Codes:	E2,F3	E2,F3

Mnemonic: NOP

Operation: No operation performed.

Description: No operation is performed; execution continues with the next sequential instruction.

Note: Same operation as the defined instruction (code 00).

Special Conditions:
Cycles: 1
Bytes: 1

Appendix 3. Instruction Set Parameters

This appendix contains details of the instruction set parameters for the 8085, 6800, 8048 and 68000 microprocessors.

A3.1 Instruction Set Parameters for the 8085

The 8085 contains the following instruction types:-

Single Byte Instructions	Declared	Undeclared	Total
Non-Jumping	183	5	188
Conditional Jump	8	1	9
Jump	11	0	11
Total	202	6	208

Double Byte Instructions	Declared	Undeclared	Total
Non-Jumping	18	2	20
Conditional Jump	0	0	0
Jump	0	0	0
Total	18	2	20

Triple Byte Instructions	Declared	Undeclared	Total
Non-Jumping	8	0	8
Conditional Jump	16	2	18
Jump	2	0	2
Total	26	2	28

All Instructions	Declared	Undeclared	Total
Non-Jumping	209	7	216
Conditional Jump	24	3	27
Jump	13	0	13
Total	246	10	256

The effective number of jump instructions is 26.5.

Jump Instruction Types for the 8085

Unconditional Jumps

Code	Mnemonic	Length	Type
C3	JMP	3	JMP
CD	CALL	3	JMP
C9	RET	1	RET
C7	RST 0	1	RST
CF	RST 1	1	RST
D7	RST 2	1	RST
DF	RST 3	1	RST
E7	RST 4	1	RST
EF	RST 5	1	RST
F7	RST 6	1	RST
FF	RST 7	1	RST
E9	PCHL	1	JMP
76	HLT	1	HLT

Conditional Jumps

Code	Mnemonic	Length	Type
C2	JNZ	3	JMP
CA	JZ	3	JMP
D2	JNC	3	JMP
DA	JC	3	JMP
E2	JPO	3	JMP
EA	JPE	3	JMP
F2	JP	3	JMP
FA	JM	3	JMP
C4	CNZ	3	JMP
CC	CZ	3	JMP
D4	CNC	3	JMP
DC	CC	3	JMP
E4	CPO	3	JMP
EC	CPE	3	JMP
F4	CP	3	JMP
FC	CM	3	JMP
C0	RNZ	1	RET
C8	RZ	1	RET
D0	RNC	1	RET
D8	RC	1	RET
E0	RPO	1	RET
E8	RPE	1	RET
F0	RP	1	RET
F8	RM	1	RET
DD	***	3	JMP
FD	***	3	JMP
CB	***	1	RST

HLT -- Halt Instructions.
JMP -- Jump Instructions.
RST -- Restart Instructions.
RET -- Return Instructions.

*** -- Undefined Instructions.

A3.2 Instruction Set Parameters for the 6800

The 6800 contains the following instruction types:-

Single Byte Instructions	Declared	Undeclared	Total
Non-Jumping	47	25	72
Conditional Jump	0	0	0
Jump	4	4	8
Total	51	29	80

Double Byte Instructions	Declared	Undeclared	Total
Non-Jumping	85	12	97
Conditional Jump	14	1	15
Jump	4	4	8
Total	103	17	120

Triple Byte Instructions	Declared	Undeclared	Total
Non-Jumping	41	10	51
Conditional Jump	0	0	0
Jump	2	1	3
Total	43	11	54

Four Byte Instructions	Declared	Undeclared	Total
Non-Jumping	0	2	2
Conditional Jump	0	0	0
Jump	0	0	0
Total	0	0	2

All Instructions	Declared	Undeclared	Total
Non-Jumping	173	49	222
Conditional Jump	14	1	15
Jump	10	9	19
Total	197	59	256

The effective number of jump instructions is 26.5.

Jump Instruction Types for the 6800

Unconditional Jumps				Conditional Jumps			
Code	Mnemonic	Length	Type	Code	Mnemonic	Length	Type
20	BRA	2	JMP	22	BHI	2	JMP
6E	JMP(I)	2	JMP	23	BLS	2	JMP
7E	JMP(E)	3	JMP	24	BCC	2	JMP
8D	BSR	2	JMP	25	BCS	2	JMP
AD	JSR(I)	2	JMP	26	BNE	2	JMP
BD	JSR(E)	3	JMP	27	BEQ	2	JMP
39	RTS	1	RET	28	BVC	2	JMP
3B	RTI	1	RET	29	BVS	2	JMP
3E	WAI	1	HLT	2A	BPL	2	JMP
3F	SWI	1	RST	2B	BMI	2	JMP
38	***	1	RET	2C	BGE	2	JMP
3A	***	1	RET	2D	BLT	2	JMP
3C	***	1	HLT	2E	BGT	2	JMP
3D	***	1	HLT	2F	BLE	2	JMP
9D	***	2	HLT	21	***	2	JMP
CD	***	2	JMP				
DD	***	2	HLT				
ED	***	2	JMP				
FD	***	3	JMP				

HLT -- Halt Instructions.
JMP -- Jump Instructions.
RET -- Return Instructions.
RST -- Restart Instructions.

*** -- Undefined Instructions.

(I) -- Indexed Addressing.
(E) -- Extended Addressing.

A3.3 Instruction Set Parameters for the 8048

The 8048 instruction set is dependent on the manufacturer of the device. The main figures given are for processors made by Intel. Figures in brackets show the variations for processors made by NEC.

Single Byte Instructions	Declared	Undeclared	Total
Non-Jumping	161	20(18)	181(179)
Conditional Jump	0	0	0
Jump	3	0	3
Total	164	20(18)	184(182)

Double Byte Instructions	Declared	Undeclared	Total
Non-Jumping	22	2	24
Conditional Jump	20	3	23
Jump	24	1(3)	25(27)
Total	66	6(8)	72(74)

All Instructions	Declared	Undeclared	Total
Non-Jumping	183	22(20)	205(203)
Conditional Jump	20	3	23
Jump	27	1(3)	28(30)
Total	230	26	256

The effective number of jump instructions is 39.5 (41.5).

Jump Instruction Types for the 8048

Unconditional Jumps

Code	Mnemonic	Length	Type
04	JMP 0XX	2	JMP
24	JMP 1XX	2	JMP
44	JMP 2XX	2	JMP
64	JMP 3XX	2	JMP
84	JMP 4XX	2	JMP
A4	JMP 5XX	2	JMP
C4	JMP 6XX	2	JMP
E4	JMP 7XX	2	JMP
14	CALL 0XX	2	JMP
34	CALL 1XX	2	JMP
54	CALL 2XX	2	JMP
74	CALL 3XX	2	JMP
94	CALL 4XX	2	JMP
B4	CALL 5XX	2	JMP
D4	CALL 6XX	2	JMP
F4	CALL 7XX	2	JMP
B3	JMP @A	1	JMP
D6	***	2	JMP
E8	DJNZ R0	2	JMP
E9	DJNZ R1	2	JMP
EA	DJNZ R2	2	JMP
EB	DJNZ R3	2	JMP
EC	DJNZ R4	2	JMP
ED	DJNZ R5	2	JMP
EE	DJNZ R6	2	JMP
EF	DJNZ R7	2	JMP
83	RET	1	RET
93	RETR	1	RET

NEC 8048 ONLY

E0	DJNZ @R0	2	JMP
E1	DJNZ @R1	2	JMP

Conditional Jumps

Code	Mnemonic	Length	Type
12	JB0	2	JMP
32	JB1	2	JMP
52	JB2	2	JMP
72	JB3	2	JMP
92	JB4	2	JMP
B2	JB5	2	JMP
D2	JB6	2	JMP
F2	JB7	2	JMP
06	***	2	JMP
16	JTF	2	JMP
26	JNT0	2	JMP
36	JT0	2	JMP
46	JNT1	2	JMP
56	JT1	2	JMP
66	***	2	JMP
76	JF1	2	JMP
86	JNI	2	JMP
96	JNZ	2	JMP
A6	***	2	JMP
B6	JF0	2	JMP
C6	JZ	2	JMP
E6	JNC	2	JMP
F6	JC	2	JMP

HLT -- Halt Instructions.
JMP -- Jump Instructions.
RET -- Return Instructions.
RST -- Restart Instructions.

*** -- Undefined Instructions.

XX -- Low-order Byte of Jump Address.

A3.4 Instruction Set Parameters for the 68000

The 68000 contains the following instruction types:-

All Instructions	Declared	Undeclared	Total
Non-Jumping	41021	0	41021
Conditional Jump	4210	0	4210
Jump	20305	0	20305
Total	65536	0	65536

Jump Instruction Types for the 68000

For the 68000 it is not acceptable to assume that conditional jump instructions will cause transfer of execution on 50% of the occasions that they are executed. The list on the following page shows how the different instructions have been divided into the effective number of codes which fall into particular groups. Further details of the divisions are given in section 4.2.1.

Jump Instruction Types for the 68000 (cont.)

Instruction	No. Codes	Non-Jump	Exception(RST)	Jump (Type)
Bcc	3584	1792	896	896 (JMP)
BRA	256	0	128	128 (JMP)
BSR	256	0	128	128 (JMP)
CHK	424	106	318	0
DBcc	128	64	32	32 (JMP)
JMP	28	0	14	14 (JMP)
JSR	28	0	14	14 (JMP)
RTR	1	0	0.5	0.5 (RET)
RTS	1	0	0.5	0.5 (RET)
TRAP	16	0	16	0
TRAPV	1	0.5	0.5	0
Privilege Instructions				
STOP	1	0	0.5	0.5 (HLT)
RESET	1	0.5	0.5	0
RTE	1	0	0.5	0.5 (RET)
MOVE to SR	53	26.5	26.5	0
ANDI to SR	1	0.5	0.5	0
EORI to SR	1	0.5	0.5	0
ORI to SR	1	0.5	0.5	0
MOVE USP	16	8	8	0
Totals	4798	1999	1585	1214

In addition to those mentioned above, there are 19,717 illegal or unassigned op-codes which generate an exception if they are executed.

Overall Instruction Grouping

	Effective Number
HLT -- Halt Instuctions.	0.5
JMP -- Random Jump Instructions.	1212.0
RET -- Return Instructions.	1.5
RST -- Restart Instructions.	21302.0
Non-Jumping Instructions.	43020.0
Total	65536.0

Appendix 4. Equations for Transfers within a Program Area

This appendix contains the detailed derivation of the probability equations governing the transfers between states, during erroneous execution in program areas. The following derivations are valid for processors having single, double and triple byte instructions only.

In order to calculate the probabilities of reaching a particular state, it is necessary to determine the possible ways of transferring form one state to another. All the possible transfers from the three different operand fields are shown below. In all cases the first byte read is the fourth in the sequence.

Transfer from State DX

Transfer to Jump	. . D J . . . . .
Transfer to Resume	. . D S V . . . .
	. . D D S V . . .
	. . D T S S V . .
	. . D T D X V . .
Transfer to DX	. . D D D X . . .
	. . D T S D X . .
Transfer to TXX	. . D D T X X . .
	. . D T S T X X .
Transfer to TXX	. . D T T X X . .

Transfer from State TXX

Transfer to Jump	. T X J . . . . .
Transfer to Resume	. T X S V . . . .
	. T X D S V . . .
	. T X T S S V . .
	. T X T D X V . .
Transfer to DX	. T X D D X . . .
	. T X T S D X . .
Transfer to TXX	. T X D T X X . .
	. T X T S T X X .
Transfer to TXX	. T X T T X X . .

Transfer from State TXX

Transfer to Jump	. . T J X . . . .
Transfer to Resume	. . T D X V . . . . . T T X S V . .
Transfer to DX	. . T T X D X . .
Transfer to TXX	. . T T X T X X .
Transfer to TXX	. . T S X . . . .

The symbols used are as follows:-

_ Byte interpreted as an instruction.

V Any valid instruction bytes.
X Operand byte of any value.

S Single byte instruction op-code in the program.
D Double byte instruction op-code in the program.
T Triple byte instruction op-code in the program.

S Operand byte interpreted as a single byte non-jumping instruction.
D Operand byte interpreted as a double byte non-jumping instruction.
T Operand byte interpreted as a triple byte non-jumping instruction.

J Operand byte interpreted as a jump instruction type.

X Operand byte interpreted as an instruction.

The probability that a particular transfer occurs is evaluated by multiplying together the probabilities that each specific byte appears in that sequence. For example, the transfer from the operand field of a double byte instruction to resuming valid instruction fetches can be achieved in four different ways. The probability of each sequence is given by:-

$$P_{DXR1} = P_{DS} \qquad \text{Eqn. A4.1}$$

$$P_{DXR2} = P_{DD} \cdot P_S \qquad \text{Eqn. A4.2}$$

$$P_{DXR3} = P_{DT} \cdot P_S \cdot P_S \qquad \text{Eqn. A4.3}$$

$$P_{DXR4} = P_{DI} \cdot P_D \qquad \text{Eqn. A4.4}$$

Where:- R is the state of resuming valid instruction fetches.

DXR represents the transfer from DX to R.

For all the other quantities the same nomenclature has been used as above, so that the probability of interpreting an operand byte of a double byte instruction, as a single byte non-jumping instruction, is represented by P_{DS}.

As the transfer can occur in any one of these ways, the overall probability of the transfer, P_{DXR}, is given by:-

$$P_{DXR} = P_{DXR1} + P_{DXR2} + P_{DXR3} + P_{DXR4} \qquad \text{Eqn. A4.5}$$

Similar expressions can be obtained for all the other transfers.

The probability of a specific byte appearing at a given location is obtained from the ratio of that byte type to the total number of locations. For example P_{DS} is given by:-

$$P_{DS} = \frac{N_{DS}}{N_{DI}} \qquad \text{Eqn. A4.6}$$

Where:- N_{DS} is the number of single byte non-jumping instructions which appear in the operand field of double byte instructions.

N_{DI} is the number of double byte instructions.

P_S is given by:-

$$P_S = \frac{N_{SI}}{N_I} \qquad \text{Eqn. A4.7}$$

Where:- N_{SI} is the number of single byte instructions in the program area.

N_I is the total number of instructions.

Values of these probabilities can either be obtained by assuming equal use of each instruction and random data in the operand fields, or by analysing actual programs.

From the above expressions it is possible to derive equations for the

probability of being at a particular state, I instruction cycles after the erroneous jump. They are of the following form:-

$$P_{D\underline{X}}(I) = P_{D\underline{X}}(I-1) \cdot P_{D\underline{X}D\underline{X}} + P_{TX\underline{X}}(I-1) \cdot P_{TX\underline{X}D\underline{X}} + P_{T\underline{X}X}(I-1) \cdot P_{T\underline{X}XD\underline{X}}$$

Eqn. A4.8

Where:- $D\underline{X}D\underline{X}$ represents the transfer from $D\underline{X}$ to $D\underline{X}$.

$T\underline{X}XD\underline{X}$ represents the transfer from $T\underline{X}X$ to $D\underline{X}$.

$TX\underline{X}D\underline{X}$ represents the transfer from $TX\underline{X}$ to $D\underline{X}$.

Similar expressions can be obtained for $P_{TX\underline{X}}(I)$ and $P_{T\underline{X}X}(I)$.

For the probabilities of a jump to another part of the memory map or of resuming valid instruction fetches, the values are cumulative because it is assumed that once in these states, execution cannot transfer elsewhere. Therefore the following expressions apply:-

$$P_R(I) = P_R(I-1) + P_{D\underline{X}}(I-1) \cdot P_{D\underline{X}R} + P_{TX\underline{X}}(I-1) \cdot P_{TX\underline{X}R} + P_{T\underline{X}X}(I-1) \cdot P_{T\underline{X}XR}$$

Eqn. A4.9

The analysis of section 5.2 treats the jump instruction types as four separate groups. For clarity, the derivations so far have only considered a single jump type. However the expressions for the individual groups are the same as for the overall group, except that the probabilities of a particular jump type appearing in the operand field are reduced proportionally.

Section 5.2 also shows that the probabilities, when I equals zero, can be found. Therefore the probabilities for all other positive integer values of I can be evaluated from the above equations.

Appendix 5. Results of Execution in Unpopulated Memory Areas

This appendix gives detailed results of the execution following an erroneous jump into unpopulated memory areas of the 8048 and 8085. For both processsors instruction fetches read back the lower order byte of the address and therefore a 256 byte sequence appears in these areas. The results below show the effective number of starting points within the sequence which give a particular transfer, and from this the probability of each outcome has been calculated.

A5.1 Unpopulated Area Execution for the 8048

Final Instruction Executed	Effective Number of Start Addresses	% Probability of Transfer
JUMP 005/805	1.0	0.4
CALL 015/815	15.0	5.9
JUMP 125/925	1.0	0.4
CALL 135/935	46.0	18.0
JUMP 245/A45	1.0	0.4
CALL 255/A55	15.5	6.1
JUMP 365/B65	31.5	12.3
CALL 375/B75	16.0	6.3
JUMP 485/C85	1.0	0.4
CALL 495/C95	8.0	3.1
JUMP 5A5/DA5	16.0	6.3
CALL 5B5/DB5	7.5	2.9
JUMP 6C5/EC5	16.0	6.3
CALL 6D5/ED5	8.0	3.1
JUMP 7E5/FE5	24.0	9.4
CALL 7F5/FF5	15.0	5.9
CALL 815	0.5	0.2
CALL 935	0.5	0.2
CALL 7F5	1.0	0.4
JUMP @A	8.5	3.3
RET	15.0	5.9
RETR	8.0	3.1

Where two addresses have been given the transfer is dependent on the state of the memory bank select flip-flop, and the corresponding address will be used.

A5.2 Unpopulated Area Execution for the 8085

Address or Instruction Reached	Effective Number of Start Addresses	% Probability of Transfer	Transfer Type
HALT	122.2	47.7	HLT
RETURN	89.6	35.0	RET
Address in HL Register	3.5	1.4	JMP
DFDE	3.0	1.2	SPC
FFFE	2.2	0.9	SPC
RESTART 4	2.0	0.8	RST
RESTART 5	2.0	0.8	RST
RESTART 6	2.0	0.8	RST
RESTART 7	2.0	0.8	RST
F4F3	1.9	0.7	SPC
Address in DE Register	1.9	0.7	JMP
RESTART 1	1.8	0.7	RST
Address in BC Register	1.5	0.6	JMP
RESTART 0	1.5	0.6	RST
CFCE	1.5	0.6	SPC
RESTART 2	1.5	0.6	RST
RESTART 3	1.5	0.6	RST
C4C3	1.0	0.4	SPC
C5C4	1.0	0.4	SPC
D4D3	1.0	0.4	SPC
DCDB	1.0	0.4	SPC
E4E3	1.0	0.4	SPC
E6E5	1.0	0.4	SPC
EEED	1.0	0.4	SPC
F6F5	1.0	0.4	SPC
FCFB	1.0	0.4	SPC
FEFD	1.0	0.4	SPC
CECD	0.8	0.3	SPC
Address in PSW	0.8	0.3	JMP
C6C5	0.5	0.2	SPC
CCCB	0.5	0.2	SPC
RESTART 8	0.5	0.2	RST
DEDD	0.5	0.2	SPC
ECEB	0.5	0.2	SPC
D6D5	0.5	0.2	SPC

Transfer types:-

HLT	Halt.
RST	Restart.
JMP	Random Jump.
RET	Return.
SPC	Specific Jump.

Appendix 6. Software for the Fault Simulation Test Facility

This appendix contains full commented listings of the software written for the fault simulation test facility. It is split into a number of modules. CONTROL is the control program which organises the sequence of runs and calls the other modules. PREFAULT is the main core of the interrupt service routine. It retrieves the return address from the stack and saves all the registers before calling the fault injection routine. Both of these modules provide the basis for all testing and do not require alteration.

The remaining modules are test dependent and have to be rewritten for different faults or test programs. For these modules, the listings show a specific example. INIT initialises the state of the test system before each run. FAULT simulates the desired fault during the interrupt routine. CHECK gives an indication of the correctness of the result after execution. TEST is the program to be tested.

A6.1 CONTROL - Main Control Program

```
;*************************************************************************
;
;          CONTROL PROGRAM FOR TESTING FAULT TOLERANT ROUTINES
;       IT INJECTS A FAULT AT SUCCESSIVE LOCATIONS IN THE TEST PROGRAM
;
; THE TESTING FACILITY USES THE TIMER SECTION OF THE 8155 ON THE SDK BOARD
;          TO GENERATE INTERRUPTS AT DIFFERENT POINTS IN THE PROGRAM.
;
; THE INTERRUPT ROUTINE CAN THEN BE USED TO INJECT 'FAULTS' INTO THE SYSTEM
;                BY CORRUPTING REGISTERS OR MEMORY LOCATIONS.
;
;*************************************************************************
;
BLKEND  EQU     020B6H          ;STORE FOR END OF PROG ADDRESS
BLKST   EQU     07FFFH          ;START OF PROGRAM ADDRESS
CHECK   EQU     08700H          ;LOCATION OF CHECKING ROUTINE
CIN     EQU     00820H          ;READS IN SERIAL BYTE INTO A AND C REGS
COUT    EQU     00850H          ;OUTPUTS SERIAL CHARACTER IN C REG
ENDTC   EQU     02008H          ;STORES FINAL VALUE OF TIMER COUNT
GETAD   EQU     00626H          ;READS ADDRESS INTO BC REGS
INIT    EQU     08500H          ;LOCATION OF ROUTINE TO INITIALISE REGS
MASK    EQU     02007H          ;TEMPORARY STORE FOR INTERRUPT MASK
```

CONTROL (cont.)

```
MONIT   EQU     00408H          ;RETURN ADDRESS INTO SDK BOARD MONITOR
NUMFLG  EQU     02005H          ;FLAG FOR SINGLE FAULT INJECTION
PROG    EQU     0C000H          ;START OF TEST PROGRAM
RUNNUM  EQU     02003H          ;HOLDS THE VALUE OF THE CURRENT RUN NO.
STACK   EQU     020B0H          ;STACK POINTER
TCOUNT  EQU     020C0H          ;LOCATION FOR TIMER COUNT
TIMFLG  EQU     02006H          ;FLAG FOR SINGLE LOCATION FAULT INJECTION
UPDAD   EQU     00362H          ;DISPLAYS CONTENTS OF HL ON SDK BOARD
;
        ASEG
        ORG     09800H          ;LOCATE PROGRAM
;
        LXI     SP,STACK        ;SETS STACK FOR CONTROL PROG USE
        DI                      ;DISABLES INTERRUPTS WHILE SETTING TIMER
        LXI     H,06H           ;SET TIMER COUNT FOR FIRST RUN
        SHLD    TCOUNT          ;SAVES VALUE
        MVI     A,000H          ;SET FLAGS
        STA     TIMFLG
        STA     NUMFLG
        LXI     H,00000H        ;SET RUN NUMBER
        SHLD    RUNNUM
        LXI     H,MESS1         ;LOADS START ADDRESS OF MESSAGE
        CALL    STRING          ;AND OUTPUTS STRING
        CALL    GETAD           ;READS IN END OF PROGRAM BLOCK
        MOV     H,B             ;TRANSFER BLOCK END AND SAVE
        MOV     L,C
        SHLD    BLKEND
        LXI     H,MESS2         ;READY MESSAGE AND SEND
        CALL    STRING
        CALL    CIN             ;READ IN SINGLE BYTE
        ANI     07FH            ;REMOVE EXTRA BIT
        CPI     'S'             ;CHECK IF SINGLE LOCATION REQUIRED
        JNZ     SKIP1           ;IF ALL LOCATIONS SKIP RESETTING FLAGS
        LXI     H,MESS3
        MVI     A,0FFH          ;SET SINGLE LOCATION FLAG
        STA     TIMFLG
        CALL    STRING          ;REQUEST COUNT FOR SINGLE LOCATION
        CALL    GETAD           ;READ IN TIMER COUNT
        MOV     H,B             ;TRANSFER TO HL
        MOV     L,C
        SHLD    TCOUNT          ;STORE NEW VALUE OF TIMER COUNT
SKIP1:  LXI     H,MESS4         ;READY MESSAGE AND SEND
        CALL    STRING
        CALL    CIN             ;READ IN SINGLE BYTE
        ANI     07FH            ;REMOVE EXTRA BIT
        CPI     'S'             ;CHECK FOR SINGLE FAULT INJECTION
        JNZ     START           ;IF FULL RUN START EXECUTION
        MVI     A,0FFH          ;SET RUN NUMBER FLAG
        STA     NUMFLG
        LXI     H,MESS5         ;READY MESSAGE AND SEND
        CALL    STRING
        CALL    GETAD           ;READ IN RUN NUMBER
```

CONTROL (cont.)

```
        MOV     H,B             ;MOVE VALUE INTO HL
        MOV     L,C
        SHLD    RUNNUM          ;STORE VALUE IN MEMORY
        JMP     START           ;SKIP INCREMENT OF COUNTER
ISTART: LHLD    TCOUNT          ;RESET COUNTER FOR NEXT RUN
        INX     H
        SHLD    TCOUNT
START:  LXI     SP,STACK        ;SET STACK FOR CONTROL PROGRAM USE
        LHLD    TCOUNT          ;LOAD TIMER COUNT INTO HL REGS
        SHLD    ENDTC           ;SAVES VALUE FOR DISPLAY ON COMPLETION
        CALL    UPDAD           ;DISPLAY VALUE ON SDK BOARD
        LXI     B,BLKST         ;LOAD BLOCK START ADDRESS IN ROM
        LXI     D,PROG-1        ;LOAD BLOCK START ADDRESS IN RAM
        LHLD    BLKEND          ;LOAD BLOCK END ADDRESS IN RAM
MOVE:   INX     B               ;INCREMENT POINTERS
        INX     D
        LDAX    B               ;READ IN BYTE FROM INSTANT ROM
        STAX    D               ;STORE BYTE IN RAM
        MOV     A,L             ;CHECK FOR END OF BLOCK
        CMP     E
        JNZ     MOVE
        MOV     A,H
        CMP     D
        JNZ     MOVE
        LHLD    TCOUNT          ;LOAD COUNT FOR PROGRAMMING TIMER
        MVI     A,040H          ;STOP TIMER IF RUNNING
        OUT     028H
        MOV     A,L             ;LOADS LOW ORDER BYTE OF COUNT
        OUT     02CH
        MOV     A,H             ;LOADS HIGH ORDER BYTE OF COUNT
        OUT     02DH
        CALL    INIT            ;INITIALISES REGISTERS BEFORE TEST
        MVI     A,0C0H          ;START COUNT
        OUT     028H
        MVI     A,01BH          ;SET INTERRUPT TO ENABLE RST 7.5
        SIM
        POP     PSW             ;RESETS PSW BEFORE TEST
        LXI     SP,0C800H       ;SET STACK BEFORE JUMP
        EI
        OUT     0FFH            ;TRIGGERS HARDWARE TO MASK OFF A14,A15
        JMP     PROG            ;JUMPS INTO PROGRAM TO BE TESTED
;
; FOR THE FIRST RUN AN INTERRUPT WILL OCCUR DURING THE JUMP INSTRUCTION.
; IT IS THEREFORE POSSIBLE TO INJECT A 'FAULT' BEFORE EXECUTION OF THE
; FIRST INSTRUCTION IN THE TEST PROGRAM.
;
; AFTER A COMPLETE RUN OF THE TEST PROGRAM, FOLLOWING CODE WILL BE
;  EXECUTED TO DETERMINE SUCCESS OR FAILURE, THEN OUTPUT THE RESULT.
;
        ORG     09900H          ;FIX RETURN ADDRESS
;
RETURN: LXI     SP,STACK        ;RESET STACK AFTER TEST
```

CONTROL (cont.)

```
         RIM                    ;READ IN STATE OF INTERRUPTS
         STA      MASK          ;STORE IN TEMP LOCATION
         MVI      A,010H        ;RESET INTERRUPT OFF
         SIM
         CALL     CHECK         ;SUBROUTINE TO CHECK IF CORRECT RESULTS
         JNC      FAIL          ;CARRY NOT SET IF UNSUCCESSFUL
         MVI      C,'S'         ;OUTPUTS 'S' TO INDICATE SUCCESS
         CALL     COUT
         JMP      TEST1         ;RETURNS AFTER SENDING 'S'
FAIL:    CALL     OUTTC         ;OUTPUTS TIMER COUNT
TEST1:   LDA      TIMFLG        ;TEST FOR SINGLE LOCATION FAULT
         INR      A
         JZ       TEST2         ;SINGLE RUN JUMP TO SECOND TEST
         LDA      MASK          ;RELOAD STATE OF INTERRUPTS
         ANI      040H          ;TESTS TO SEE IF MORE RUNS REQUIRED
         JZ       ISTART        ;GOES BACK FOR NEXT RUN
         LXI      H,0006H       ;RESET TIMER COUNT FOR NEXT SET OF RUNS
         SHLD     TCOUNT        ;SAVE VALUE FOR LATER USE
TEST2:   LDA      NUMFLG        ;LOAD TEST FLAG INTO ACC
         INR      A
         JZ       PREND         ;JUMP IF SINGLE RUN OR END
         MVI      C,0DH         ;SEND CR,LF TO PLACE  RUN NUMBER IN
         CALL     COUT          ;LEFT HAND COLUMN OF LINE
         MVI      C,0AH
         CALL     COUT
         LHLD     RUNNUM        ;LOAD RUN NUMBER
         INX      H             ;INCREMENT READY FOR NEXT RUN
         SHLD     RUNNUM        ;SAVE FOR LATER USE
         CALL     OUTNUM        ;OUTPUTS VALUE TO SCREEN
         MVI      C,'*'         ;OUTPUT '*' TO MARK RUN NUMBER
         CALL     COUT
         JMP      START         ;JUMP BACK FOR NEXT RUN
;
PREND:   MVI      A,010H        ;RESET RST7.5 FLIP FLOP TO OFF
         SIM
         LXI      H,MESS6       ;OUTPUTS TERMINATING MESSAGE
         CALL     STRING
         LHLD     ENDTC         ;OUTPUT VALUE OF TIMER COUNT
         CALL     OUTNUM
         LXI      H,MESS7
         CALL     STRING
         LHLD     RUNNUM        ;OUTPUT VALUE OF RUN NUMBER
         CALL     OUTNUM
         MVI      C,01AH        ;SEND END OF FILE MARKER
         CALL     COUT
         JMP      MONIT         ;JUMP BACK TO SDK MONITOR
;
MESS1:   DB   0DH,0AH,'FAULT TOLERANT TESTING FACILITY',0DH,0AH,0AH
         DB   'TEST PROGRAM LOCATED FROM C000 TO $'
MESS2:   DB   0DH,0AH,0AH,'TYPE ''S'' FOR FAULT INJECTION AT A SINGLE'
         DB   ' LOCATION $'
MESS3:   DB   0DH,0AH,0AH,'ENTER TIMER COUNT FOR REQUIRED LOCATION $'
```

CONTROL (cont.)

```
MESS4:  DB      0DH,0AH,0AH,'TYPE ''S'' FOR SINGLE RUN ON EACH '
        DB      'LOCATION $'
MESS5:  DB      0DH,0AH,0AH,'ENTER RUN NUMBER FOR REQUIRED RUN $'
MESS6:  DB      0DH,0AH,07H,'EXECUTION TERMINATED AT TIMER COUNT $'
MESS7:  DB      ', AND RUN NUMBER $'
;
;*****************************************************************************
;
;           SUBROUTINE -- STRING          CALLS:-   COUT
;
; ROUTINE TO OUTPUT A STRING OF CHARACTERS DELIMITED BY A '$'
; START ADDRESS OF STRING MUST BE IN THE HL REG PAIR
;
;*****************************************************************************
;
STRING: MOV     C,M             ;GET BYTE FROM MEMORY
        MOV     A,C             ;CHECK FOR DELIMITER
        CPI     '$'
        RZ                      ;RETURN IF END OF MESSAGE
        PUSH    H               ;SAVE MEMORY ADDRESS
        CALL    COUT            ;OUTPUT CHARACTER
        POP     H
        INX     H               ;INCREMENT ADDRESS POINTER
        JMP     STRING          ;GO BACK FOR NEXT CHARACTER
;
;*****************************************************************************
;
; SUBROUTINE -- OUTTC/(OUTNUM)    CALLS -- AOUT, CONV, COUT
;
; CONVERTS 16 BIT ADDRESS STORED IN 'TCOUNT', (OR IN HL REGISTER PAIR),
; INTO 4 ASCII CODES AND OUTPUTS THEM TO THE SERIAL PORT
;
;*****************************************************************************
;
OUTTC:  LHLD    TCOUNT          ;LOAD IN NUMBER FOR OUTPUT
OUTNUM:  MOV    A,H
        CALL    AOUT            ;OUTPUTS HIGH BYTE
        MOV     A,L
        CALL    AOUT            ;OUTPUTS LOW BYTE
        RET
;
;*****************************************************************************
;
;       SUROUTINE -- AOUT               CALLS -- CONV, COUT
;
; CONVERTS SINGLE BYTE IN ACCUMULATOR INTO TWO ASCII CODES AND OUTPUTS
; THEM TO THE SERIAL LINE
;
;*****************************************************************************
;
AOUT:   MOV     B,A             ;SAVES BYTE IN B REG
        RAR                     ;SHIFTS UPPER BITS
```

CONTROL (cont.)

```
        RAR
        RAR
        RAR
        ANI     0FH             ;MASK OFF UPPER 4 BITS
        CALL    CONV            ;CONVERTS TO ASCII AND OUTPUTS
        MOV     A,B             ;RESTORE VALUE
        ANI     0FH             ;MASKS OFF BITS
        CALL    CONV            ;CONVERTS AND SENDS
        RET
;
;*************************************************************************
;
;       SUBROUTINE -- CONV          CALLS -- COUT
;
; CONVERTS HEX DIGIT IN ACCUMULATOR TO ASCII AND OUTPUTS TO SERIAL PORT
;
;*************************************************************************
;
CONV:   ADI     030H            ;CONVERT TO ASCII
        CPI     03AH            ;CHECK IF 0-9
        JM      SKIPC
        ADI     07H             ;READJUSTS FOR A-F
SKIPC:  MOV     C,A             ;MOVE CODE TO C REG FOR OUTPUT
        CALL    COUT            ;OUTPUTS ASCII CODE
        RET
;
        END
```

A6.2 PREFAULT - Main Core of Interrupt Service Routine

```
;*************************************************************************
;
;                   THIS IS THE FAULT INJECTING ROUTINE
;
; IT IS EXECUTED AFTER A RST 7.5 INTERRUPT.  IT SAVES THE CURRENT STACK
; POINTER AND ALL THE REGISTERS.  IT RETRIEVES THE RETURN ADDRESS FROM
; MEMORY AND STORES IT AS PART OF THE JUMP INSTRUCTION AT THE END OF
; THIS ROUTINE.  A SUBROUTINE IS THEN CALLED TO ACTUALLY INJECT THE FAULT
; BEFORE REINSTATING ALL THE REGISTERS AND THE ORIGINAL STACK POINTER.
;
;         CALLS :-    FAULT
;
;*************************************************************************
;
CYTEMP  EQU     020B2H          ;TEMP STORE FOR CARRY FLAG
DUMMY   EQU     00000H          ;DUMMY ADDRESS CHANGED LATER
HLTEMP  EQU     020B0H          ;TEMP STORE FOR HL REGISTERS
FAULT   EQU     08600H          ;LOCATON OF FAULT INJECTING ROUTINE
SPTEMP  EQU     020B4H          ;TEMP STORE FOR STACK POINTER
STACK   EQU     020B0H          ;TEMP STACK POINTER FOR THIS ROUTINE
;
```

PREFAULT (cont.)

```
        ASEG
        ORG     09FC2H          ;SET ADDRESS SO LAST TWO BYTES IN RAM
;
        SHLD    HLTEMP          ;SAVE HL REGISTERS
        MVI     H,000H          ;PREPARE TO SET TEMPORARY CARRY FLAG
        JNC     SKIP            ;TEST CURRENT CARRY FLAG
        MVI     H,0FFH          ;RESET TO INDICATE CARRY WAS SET
SKIP:   SHLD    CYTEMP          ;SAVE TEMPORARY CARRY FLAG
        LXI     H,00000H        ;CLEAR HL REGS
        DAD     SP              ;GET CURRENT STACK POINTER INTO HL REGS
        LXI     SP,STACK        ;LOAD TEMPORARY STACK POINTER
        PUSH    D               ;SAVE ALL REGISTERS
        PUSH    B
        PUSH    PSW
        MOV     E,L             ;STORE COPY OF ORIGINAL SP IN DE REGS
        MOV     D,H
        INX     H               ;ADJUST OLD SP FOR REMOVAL OF RETURN
        INX     H               ;       ADDRESS
        SHLD    SPTEMP          ;SAVE VALUE IN TEMP STORE
        XCHG                    ;RETURN OLD SP INTO HL REGS
        MOV     A,H             ;SET UPPER TWO BITS SO THAT STACK POINTS
        ORI     0C0H            ;                TO MASKED AREA
        MOV     H,A             ;RETURN HIGH BYTE TO H REG
        MOV     E,M             ;GET LOW BYTE OF RETURN ADDRESS
        INX     H
        MOV     D,M             ;GET HIGH BYTE
        XCHG                    ;TRANSFER TO HL REGS
        SHLD    RETURN+1        ;STORE AS PART OF JUMP INSTRUCTION
;
        CALL    FAULT           ;CALL ROUTINE TO INJECT FAULT
;
        POP     PSW             ;RESTORE ALL REGISTERS
        POP     B
        POP     D
        LHLD    CYTEMP          ;RETRIEVE TEMPORARY CARRY FLAG
        DAD     H               ;RESET CARRY FLAG
        LHLD    SPTEMP          ;GET OLD STACK POINTER
        SPHL                    ;RESET STACK FOR TEST PROGRAM USE
        LHLD    HLTEMP          ;RESTORE HL REGS
        OUT     0FFH            ;SET UP MASKING CIRCUITRY
RETURN: JMP     DUMMY           ;DUMMY CHANGED DURING EXECUTION
        END
```

A6.3 FAULT - Simulates Desired Fault

```
;*****************************************************************************
;
; THIS IS A FAULT INJECTING ROUTINE WHICH CORRUPTS DATA IN MEMORY LOCATIONS
; C100 AND C101.
;
;*****************************************************************************
;
NUMFLG  EQU     02005H          ;FLAG SET TO 0FFH ON LAST RUN
RUNNUM  EQU     02003H          ;STORAGE LOCATION FOR THE RUN NUMBER
VAL     EQU     0C100H          ;LOCATION OF INPUTS INTO TEST ROUTINE
;
FAULT:  LHLD    RUNNUM          ;LOADS IN RUN NUMBER
        LXI     D,VAL           ;LOAD ADDRESS OF INPUTS INTO DE REGS
        XCHG                    ;SWAP HL FOR DE
        MVI     B,00H           ;CLEAR B REG READY FOR LAST RUN FLAG
        MOV     A,D             ;TRANSFER HIGH BYTE OF RUNNUM INTO ACC
        ANI     01H             ;TEST LOW ORDER BIT
        JZ      SKIP1           ;NO CORRUPTION IF BIT NOT SET
        MOV     M,E             ;CORRUPT MEMORY BYTE
        MVI     B,0FH           ;SET HALF OF LAST RUN FLAG
SKIP1:  MOV     A,D             ;RELOAD HIGH BYTE OF RUN NUMBER
        ANI     02H             ;TEST SECOND BIT
        JZ      SKIP2           ;NO CORRUPTION IF BIT NOT SET
        INX     H               ;SET ADDRESS IN HL REG TO HIGH BYTE
        MOV     M,E             ;CORRUPT HIGH BYTE IN MEMORY
        MVI     A,0F0H          ;SET SECOND HALF OF LAST RUN FLAG
        ADD     B               ;ADD BOTH HALVES OF FLAG TOGETHER
        MOV     B,A             ;MOVE FLAG TO B REG
SKIP2:  MOV     A,E             ;LOAD LOW BYTE OF RUN NUMBER INTO ACC
        INR     A               ;TEST FOR LAST RUN
        RNZ                     ;RETURN IF NOT LAST RUN
        MOV     A,B             ;MOVE FLAG INTO ACC
        STA     NUMFLG          ;STORE FLAG IN MEMORY
        RET
        END
```

A6.4 INIT - Initialisation Routine

```
;*****************************************************************************
;
; THIS ROUTINE SETS UP INITIAL STATUS OF THE PROCESSOR
;
;*****************************************************************************
;
ERRFLG  EQU     0C7FFH          ;LOCATION OF ERROR FLAG
;
        LXI     H,00000H        ;SET WHAT WILL BECOME THE PSW
        XTHL                    ;PUSHES HL ONTO STACK, REMOVES RET ADD
        PUSH    H               ;REPLACES RETURN ADDRESS ONTO STACK
        LXI     B,00000H        ;SET INITIAL VALUE FOR BC REGS
        LXI     D,00000H        ;SET INITIAL VALUE FOR DE REGS
        LXI     H,00000H        ;SET INITIAL VALUE FOR HL REGS
        MVI     A,0FFH          ;SETS A REG TO FF
        STA     ERRFLG          ;SETS ERROR FLAG
        RET
        END
```

A6.5 CHECK - Checks Result after Execution

```
;*****************************************************************************
;
; CHECKING ROUTINE TO TEST FOR SUCCESS OR FAILURE OF THE TEST PROGRAM
;
;*****************************************************************************
;
VAL3    EQU     0C110H          ;LOCATION OF ANSWER FROM TEST PROGRAM
;
CHECK:  LDA     VAL3            ;LOAD IN ANSWER FROM TEST PROGRAM
        CPI     055H            ;CHECK HIGH BYTE
        JNZ     CLEAR           ;CLEAR CARRY AND RETURN
        STC                     ;SET CARRY TO INDICATE SUCCESS
        RET
CLEAR:  XRA     A               ;CLEAR CARRY
        RET
        END
```

A6.6 TEST - Program to be Tested

```
;*****************************************************************************
;
; THIS PROGRAM ADDS TWO 8-BIT NUMBERS TOGETHER AND STORES THE RESULT
;
;  INCLUDES ERROR DETECTION AND CORRECTION SOFTWARE
;
;*****************************************************************************
;
ERRFLG  EQU     0C7FFH          ;STORE FOR ERROR FLAG
RETURN  EQU     09900H          ;RETURN ADDRESS TO CONTROL PROGRAM
START   EQU     0C000H          ;START ADDRESS OF PROGRAM
;
```

TEST (cont.)

```
          ASEG
          ORG     START
;
          LXI     SP,STACK+10H    ;LOAD STACK POINTER
          LDA     VAL11           ;LOAD IN FIRST VARIABLE
          MOV     B,A
          LDA     VAL12           ;LOAD IN SECOND COPY
          SUB     B               ;SUBTRACT VALUES
          JZ      READ2           ;IF BOTH EQUAL, USE VALUE IN B REG
          XRA     A               ;CLEAR ACCUMULATOR
          STA     ERRFLG          ;ZERO FLAG TO INDICATE  ERROR
          LDA     VAL13           ;READ IN THIRD COPY
          MOV     C,A             ;TEMP STORE IN CREG
          SUB     B               ;TEST IF EQUAL
          JZ      READ2           ;USE VALUE IN B REG
                                  ;IF ONLY ONE ERROR, MUST BE IN VAL11
          MOV     B,C             ;TRANSFER THIRD COPY TO B REG AND USE
READ2:    LDA     VAL21           ;REPEAT FOR OTHER INPUT USING C REG
          MOV     C,A
          LDA     VAL22
          SUB     C
          JZ      CALC
          XRA     A               ;CLEAR ACCUMULATOR
          STA     ERRFLG          ;CLEAR FLAG TO INDICATE ERROR
          LDA     VAL23
          MOV     D,A
          SUB     C
          JZ      CALC
          MOV     C,D
CALC:     MOV     A,B             ;TRANSFER FIRST INPUT TO ACC
          ADD     C               ;ADD TO SECOND INPUT
          STA     VAL3            ;STORE THE RESULT
          DI                      ;PREVENT ANY FURTHER INTERRUPTS
          OUT     0FFH            ;CLEARS MASK FOR RETURN TO CONTROL PROG
          JMP     RETURN          ;JUMP BACK TO CONTROL PROGRAM
;
          ORG     START+100H
;
VAL11:    DB      012H
VAL13:    DB      012H
VAL12:    DB      012H
VAL21:    DB      043H
VAL22:    DB      043H
VAL23:    DB      043H
          ORG     START+110H
VAL3:     DB      000H
STACK:                            ;BOTTOM OF STACK
          END
```