FUNCTIONAL LEVEL FAULT SIMULATION OF LSI DEVICES

by

SHIRISH K. SATHE

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTERS OF SCIENCE
in
Electrical Engineering

APPROVED:

Dr. J.R. Armstrong

Dr. F.G. Gray

Dr. J.G. Tront

June, 1982
Blacksburg, Virginia

ACKNOWLEDGEMENTS

I wish to express deep gratitude to Dr. J. R. Armstrong, my principal advisor. He was a constant source of encouragement, knowledge and inspiration. I also acknowledge the guidance and help given by Dr. F. G. Gray and Dr. J. G. Tront for reviewing the thesis and being on my graduate committee. I am also indebted to Dr. C. E. Nunnally for agreeing to be on my final examination committee.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I

INTRODUCTION

1.1 DIGITAL SIMULATION

Simulation is a process which makes it possible to model, either mathematically or functionally, the behavior of a real system. With the increasing complexity of the integrated circuits, the use of simulation gained popularity for various stages of the development of digital systems. Some of the applications of the simulation are as follows [1].

1. Hardware Design Verification
    a) Verify logical correctness
    b) Timing analysis
2. Fault Analysis
    a) Stuck-at faults
    b) Timing analysis

Simulation can be done at various levels, ranging from a component level to simulation of large functional blocks. The gate level simulator which employs primitive models such as AND, OR, NAND gates or JK, D flip-flops, has been effectively used for Small Scale and Medium Scale Integrated circuits. With the advent of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI), each device consists of

very large number of gates and these complex hardware systems require effective design support tools more than ever [2].

As mentioned above, simulation may be performed not only at the gate level but also at the functional level and the instruction level [3]. In the gate level simulation of complex LSI devices such as a microprocessor, memory chip or other peripheral chips, the model for each IC consists of thousands of gates and as a result the test patterns tend to get very large. This results in large simulation time requirments for each simulation run, making such a procedure prohibitively expensive. Moreover, accurate gate level models are usually known only to the manufacturer who is normally unwilling to release this proprietary information. Finally, maintaining accurate gate level models (each model containing tens of thousands of gates) is an extremely difficult task. There are therefore, good reasons for moving to a higher level of representation [4] [5] [6]. The instruction level simulation is efficient and results in fast execution of software programs, but this level of simulation is performed without considering the internal architecture of the device and as a result, the internal data flow within different functional blocks or the timing relations between various signals are not taken into account.

1.2 FUNCTIONAL LEVEL SIMULATION

Simulation at the functional level is a compromise between the two levels described above. Though the functional level model does not contain as much information as the gate level model, it does retain the important characteristic features of the device. In functional level simulation of an LSI device, internal micro-operations and interface signal timings are simulated. Each device is represented by a program that changes the states of the output pins and samples the input pins, duplicating the behavior of the actual device. Functional level simulation is thus clearly superior to the instruction level simulation. In addition, some faults not covered by gate level simulator can be treated by this approach. For example, there is an increasing variety of MOS integrated circuit elements whose logical behavior and faults are not adequately treated by existing gate level simulators [7]. Also line stuck-at faults, which can be implemented by a gate level simulator easily, do not cover all possible faults in the circuit [8]. Using the functional level simulation, many of these faults can be incorporated. Thus in the development of LSI and VLSI chips, the functional level simulator can serve as an invaluable tool.

With the rising complexity of the LSI devices, more and more circuit designers are using functional level simulation

as a means to verify designs of new products, to study the effects of various types of faults on the chip behavior and to simulate and evaluate self check circuitry and failsafe circuitry [1]. The use of a simulator also enables the designer to test the timing constraints and the signal interface between the functional blocks of an LSI device.

1.3 FAULT SIMULATION OF THE LSI DEVICES

At present, the only known way of checking the effectiveness of the test vector or the self-test software is to conduct "fault injection experiments". These experiments can be conducted either on a real hardware system or through simulation. Using a hardware system is not feasible since obtaining LSI devices with known internal defects is actually much more difficult than obtaining good devices [9].

In case of the chip level simulation,1 once the model for a chip is developed, a faulty chip can be obtained by assessing the effect that the fault has, on the behavior of the model, and then incorporating this effect into the model. This is done by (i) deleting part of the model microcode, (ii) modifying the microcode, (iii) adding additional microcodes or (iv) some combination of (i), (ii) and (iii).

---

1 The term chip level simulation implies functional level simulation with the model boundaries being the actual chip interface.

Since the functional level model simulates the internal timings, micro-operations and internal data-flow, defects such as incorrect micro-operation, timing faults or stuck-at faults can be inserted. Thus, a 'faulty' model for a LSI device can be developed and a system which is made up of various such models can be simulated. By applying various input patterns, the response of these faulty models can be observed to study how different types of faults manifest themselves.

Two important applications of the functional level simulator are the test generation for LSI chips and software validation. When simulation is used to generate chip test routines, as a first step a sample test vector is generated. If the test vector is a short one, the test can be conducted by applying the input patterns at predetermined simulation times. When testing a complex LSI device or a system consisting of several LSI devices, the test vector tends to get long and this approach may become impractical. In that case, the test vector is converted into the assembly language program and a microprocessor module is used to conduct the test. The assembled test program is stored in the memory module (e.g. RAM or ROM). When the system is simulated, the microprocessor module executes this test program and as a result, the test patterns are applied to the chip under test

and it's response is sampled by the processor. Any time the response deviates from that of the good device, a flag is set indicating the fault detection and the execution is terminated. When simulating the faulty chips, the test program is halted on the detection of an error. The entire test program is not executed thus reducing the CPU time on the host computer. By making several runs of the test programs with different faults in the chip under consideration, the fault coverage can be determined. In the process, the test program can also be improved to obtain better fault coverage.

The use of functional level simulation for the test validation and the fault analysis for an LSI system usually involves many fault simulation runs. Each fault is injected by modifying the model code. In the past, it was done manually and was slow and time consuming. It is thus clearly desirable to develop an automated fault injection procedure for a functional level simulator.

The purpose of this thesis is to provide methods for accurate modeling of faults in LSI devices and an efficient system for conducting functional fault simulations.

1.4 OUTLINE OF THESIS BODY

Chapter 2 explains the operation of the chip level multi-module logic simulator in detail. The instruction set used to model the devices, as well as frequently used monitor commands implemented by the simulator are discussed.

Chapter 3 presents the fault insertion techniques. It also suggests a systematic approach to model designing, which will facilitate the fault insertion process.

Chapter 4 analyses the approach to the fault simulation of the digital systems using GSP. It describes various routines developed to simulate the faulty systems efficiently.

Chapter 5 compares functional level faults in a chip level module with the stuck-at- faults in a gate level module.

Chapter 6 presents recommendations for future improvements that would further automate the fault insertion procedure.

Chapter II

GSP SIMULATOR OVERVIEW

2.1 GSP SIMULATOR

A chip level simulator which can simulate any digital device, has been developed at Virginia Tech. The original version, which could simulate one device was developed by Marvin D. Ellis, Jr.[3]. It was later modified by Donald E. Devlin [10], to simulate a number of LSI devices either independently or as one single system, with signals propagating between the devices. This chip level multimodule logic simulator is known as GSP (General Simulator Program). Using GSP, micro-operations and detailed interface signal timings are simulated without including the detailed internal gate structure of the chip.

The GSP program is written in FORTRAN and currently the system runs in either a batch (MVS) or interactive (CMS) mode on IBM processors. The program has also been converted to run on other host computers, e.g. the VAX 11/780 and CYBER 173.

2.2 GSP SIMULATION PROCEDURE

The simulation of a system on GSP consists of three separate phases.

Phase I: Chip Description. The user models the device at the chip level and codes its description using the GSP assembly language. An example of a source code is given in Figure 1. This code description is then processed by the GSP assembler to produce an integer micro-code file (object file), which is used for subsequent simulation of that device. This integer microcode file is referred to as a 'GSP module' or simply a 'module' corresponding to the chip under consideration.

Phase II: Interconnect Description. A Command file containing the initial signal values and input vector specifications is created. The COMMAND file is a list of GSP monitor commands that have to be executed before simulation can start. The GSP simulator has a monitor(MONI) which controls the action of the simulator. Using the monitor commands, the user can preset pin states, connect modules, set break points, switch the display (output panel) ON or OFF, and change the level of details to be displayed for a particular module. Details of these monitor commands are given in [10]. The COMMAND file is usually set up as shown in Figure

```
        REG(1)   ENOLD
        PIN      EN(1), INP(2,9), OUTP(10,17)
        EVW      DEL(50)
                                ;
        BEQ   EN, ENOLD, QUIT   ; quit if same
        BNE   EN, UPDAT         ; check falling edge
        MOV(DEL) INP, OUTP      ; enable buffer
UPDAT:  MOV   EN, ENOLD         ; update
 QUIT:  MOV   #0, EXIT
```

Figure 1: GSP source code for 8 bit latch

2. The first line of the Command file is usually the headings for the display of the output panel. The 'Y' command is used to merge the microcode of various modules into the simulator. Using the 'T' command, maximum simulation time is defined. The 'C' command is used to connect different pins of various modules. As an example, referring to line 6 of Figure 2,

```
C  1,4  2,20
```

the 'C' command connects pin 4 of module 1 to pin 20 of module 2. Using the command 'A' as shown below,

```
A  3 5 50 0
```

an event is added in the time queue to set the pin 5 of module 3 to value 0 when the simulation time is 50 ns. The level of details to be displayed for a particular module is controlled by the 'L' command.

Phase III: Simulation. Once the microcode file for the individual chips and the command file for the entire system are ready, the system can be simulated. As shown in Figure 3, the micro-code descriptions of the individual modules are linked into the GSPLNK file. This GSPLNK file holds the microcode for the entire system. In addition, as the simulation progresses, it stores the states of system signals.

```
;ADDRH   ADDRL     DATA      MREAD      SELECT
Y
; 1-7, 11-18      21-28       30         33
T 7000000
;#1 =>M8080, #2 =>M8228, #3 =>ROM
C   1,4         2,20
C   1,5         2,21
C   1,6         2,22
N 1 9 17
N 1 150 151
A  1 12 0 1
A  3  5 0 0
L 3 0
X
B 5000
G
```

Figure 2: GSP command file

```
GSPSIM: GSPLNK FILE IS SYSTEM1
YOU WILL BE ASKED TO ENTER OBJ FILES.
A CARRIAGE RETURN BRINGS YOU OUT OF THE LOOP.

ENTER NAME OF OBJ FILE
m8080
EXECUTION BEGINS...
MODULE  1 CONTAINS M8080
SATISFIED?(YES/NO) DEFAULT YES
y
ENTER NAME OF OBJ FILE
rom
EXECUTION BEGINS...
THESE ARE THE CURRENTLY LOADED MODULES:
MODULE  1 CONTAINS M8080
MODULE  2 CONTAINS ROM
SATISFIED?(YES/NO) DEFAULT YES

ENTER NAME OF OBJ FILE

ARE YOU SATISFIED WITH THE LOADING? (YES/NO) DEFAULT YES
y
ENTER COMMAND INPUT FILE ( DEFAULT SYSTEM1
```

Figure 3: Procedure to link modules

Once the GSPLNK file is ready, the control is transferred to MONI. Entering the monitor command 'F' results in execution of commands from the predefined Command file and the system is initialized. Module 0 (zero) serves as the output panel. Using Monitor command X, the output panel can be switched ON or OFF. When the output panel is ON and the signal to any of it's pins (1 to 49) changes, all the pin values are displayed. For example, in case of a processor based system the address Bus, data Bus, various control signals, e.g. RD, NWR, Chip Select etc., are connected to module 0.

Using 'B t' (Break at time = t), the simulation can be temporarily halted and the GSP monitor entered. The contents of the internal registers and pin values for module 'M' can be examined with the help of the 'R M' command. The 'G' command resumes the simulation. When simulating in an interactive mode (CMS), the system output is always displayed on the user's terminal. The user can also route this output to the printer or to the disk. To reduce costs, most of the longer simulation runs have been made in the batch mode as against the interactive (CMS) mode.

2.3 THE MAIN SIMULATION LOOP OF GSPSIM EXEC

Most of the simulation work using GSP was done on the IBM computer. The host computer has a IBM system/370 operating system, which supports EXEC programs. CMS EXEC is a command programing language designed to allow an user to issue a sequence of commands by typing only a single command line. An EXEC called GSPSIM controls the various phases of the GSP simulator. By executing different programs, 'GSPSIM EXEC' allows the user to merge the object files or modules into GSPLNK file, specify the Command file or direct the simulation output to a terminal, printer or disk, and simulate the system.

Using the 'GSPSIM EXEC' procedure, a multimodule system can be simulated. When simulation is used for software validation or fault analysis, it is necessary to carry out a large number of simulation runs. By replacing the faulty module M' by M" in the system, fault simulation runs can be made. With the original GSPSIM EXEC procedure, the GSPLNK file had to be recreated for each run which was slow and inefficient process. To overcome this difficulty, the GSPSIM EXEC procedure was modified to facilitate the multiple fault runs in sucession. The modified GSPSIM EXEC allows the user to simulate the system with different Command files or to replace a module in the GSPLNK file.

Figure 4 gives the flow chart of the main loop of GSPSIM EXEC. The main loop of GSPSIM EXEC consists of the following steps.

i) The name of the GSPLNK file is read. If the file does not exist, a new file is created.

ii) User specified modules are merged into the GSPLNK file.

iii) The name of the system Commnad file is read and the output of the simulation session is routed to appropriate destination i.e. disk, terminal or printer.

iv) The GSP monitor is entered and the system is initialized by executing the monitor commands from the Command file and the system is simulated. At the end of the simulation, the result of the simulation run is stored on the log file if specified by the user.

v) The user is asked to enter a number between 1 and 6. The action taken is as follows.

   1. Terminates the EXEC procedure and returns the control to the host computer.
   2. Deletes existing GSPLNK file and returns to step (v).
   3. The step (i) is executed.

4. The control is transfered to the step (ii) to add modules in the GSPLNK file.
5. Goes to step iii to read new Command file.
6. A module from the GSPLNK file is replaced and the control is returned to the step (v).

Thus using the options 5 & 6 in step (v), a number of simulation runs can be carried out without interruption.

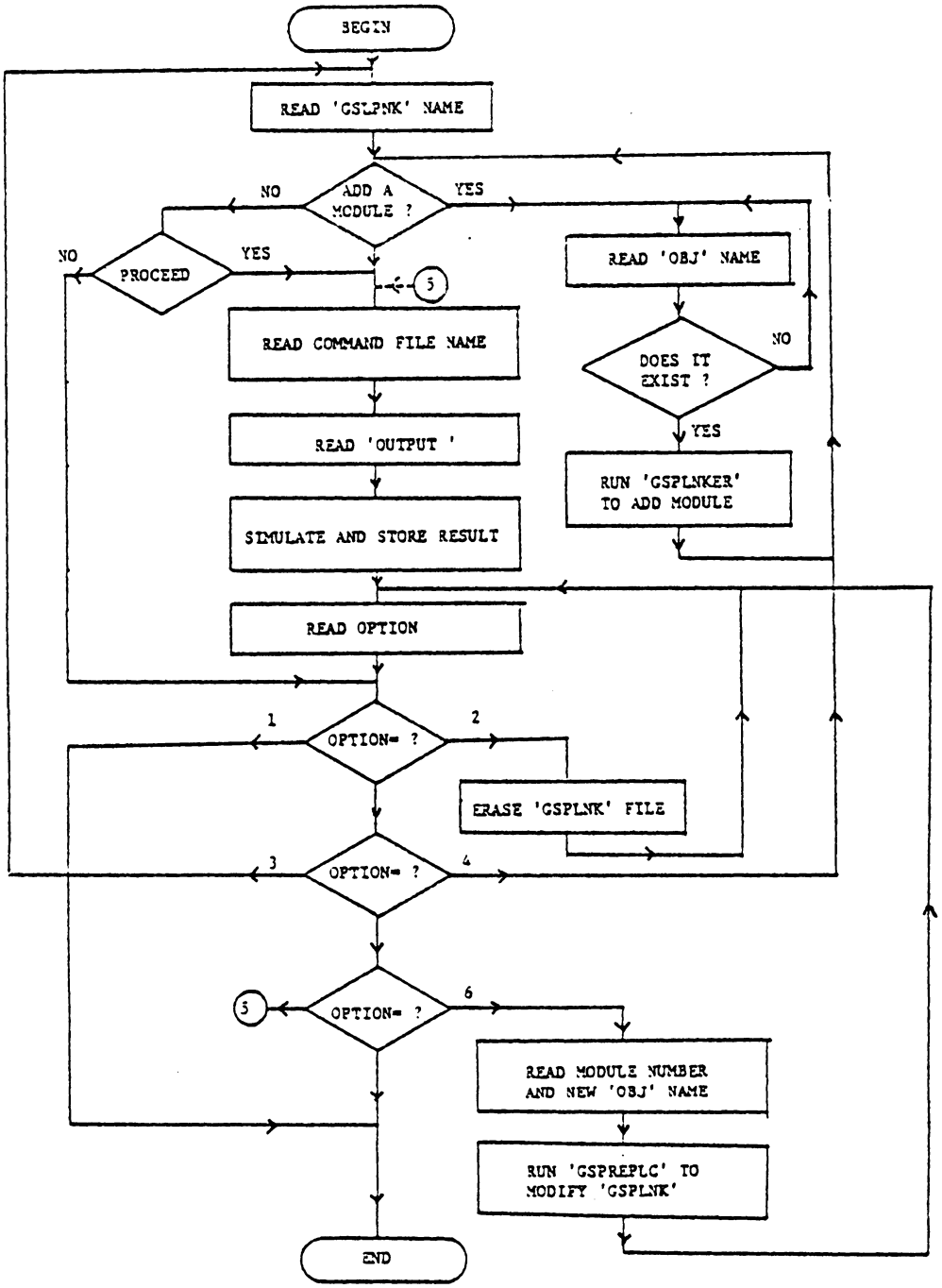

Figure 4: Flow-chart for GSPSIM EXEC

2.4 GSP: THE INSTRUCTION SET AND MODELING TECHNIQUES

The LSI device modeling process consists of the following steps. (i) Studying the manufacturer's specifications (block diagrams, word description and timing diagrams) to gain a thorough understanding of the device's operation. (ii) Generation of a detailed model flowchart. (iii) Coding the model and (iv) Assembling the model code to produce a microcode file.

Details of the GSP and its associated modeling language are given in [10]. The modeling instructions have the standard assembly language form; a subset of the language used is explained here.

2.4.1 GSP Instruction Set

Control Instructions

```
BRU LABEL           ;branch unconditionally to LABEL
BEQ R1, R2, LABEL   ;branch to LABEL if R1=R2.
BEQ R1, LABEL       ;branch to LABEL if R1=0.
JSR LABEL           ;jump to subroutine at LABEL
RTS                 ;return from subroutine.
EXIT                ;exit the module procedure.
```

Logical Instructions

```
OP R1, R2, R3       ;combine R1 AND R2, result in R3
Where OP = AND, OR, or XOR
```

```
OP R1, R2           ;same as above but result in R2.
COM R1              ;logical complement.
```

Arithmetic Instructions

```
OP R1, R2, R3       ;same as logical conventions
OP R1, R2           ;
Where OP = ADD or SUB.

OP R1               ;two's complement or increment
Where OP = NEG or INC
```

Move Instructions

```
MOV R1, R2          ;R2 assumes the value of R1.
MOV (DEL) R1, R2    ;R2 assumes value of R1 after DEL
                    ;time units
MOV #N, R1          ;immediate data (N) to R1.
```

Some Special Instructions

```
CAN n1              ;cancel all events for the
                    ;pin n1 from the time queue.
IDX R1(L), M, N     ;move M bits starting from bit L
                    ;of R1 to indexed register N.
```

2.4.2 Modeling Technique

The first step in the simulation process is to make a model for the device. The model source code consists of a declaration section, code procedure and data (see Figure 1, for example). In the declaration section, all pins and registers as well as timing delays are specified. The pins may be declared in groups of one to eight and the register size may also vary from one to eight bits. The pins and the registers may correspond to those in the actual device or they may be 'pseudo' ones used for modeling.

As an example, consider a model for an eight bit buffer with a enable signal. On a one to zero transition of the enable signal, the data on the input pins propagates to output pins in 50 ns (nano-seconds). Figures 5 and 1 show the flow chart and the GSP code for this module respectively. Note that the state of the input pins is never tested by the program. On the other hand, for a change in enable signal, the module procedure is called and the values are updated.

Using this basic technique, more complex digital chips can be modeled. As an example, consider the INTEL 8212 (Figure 6), an 8 bit I/O buffer [11]. The GSP model code of the device is given in appendix A. In the declaration section, pins DI, DO, STB, MD, DS2, NDS1, NINT and NCLR represent the pins of the real chip while the rest of the pins

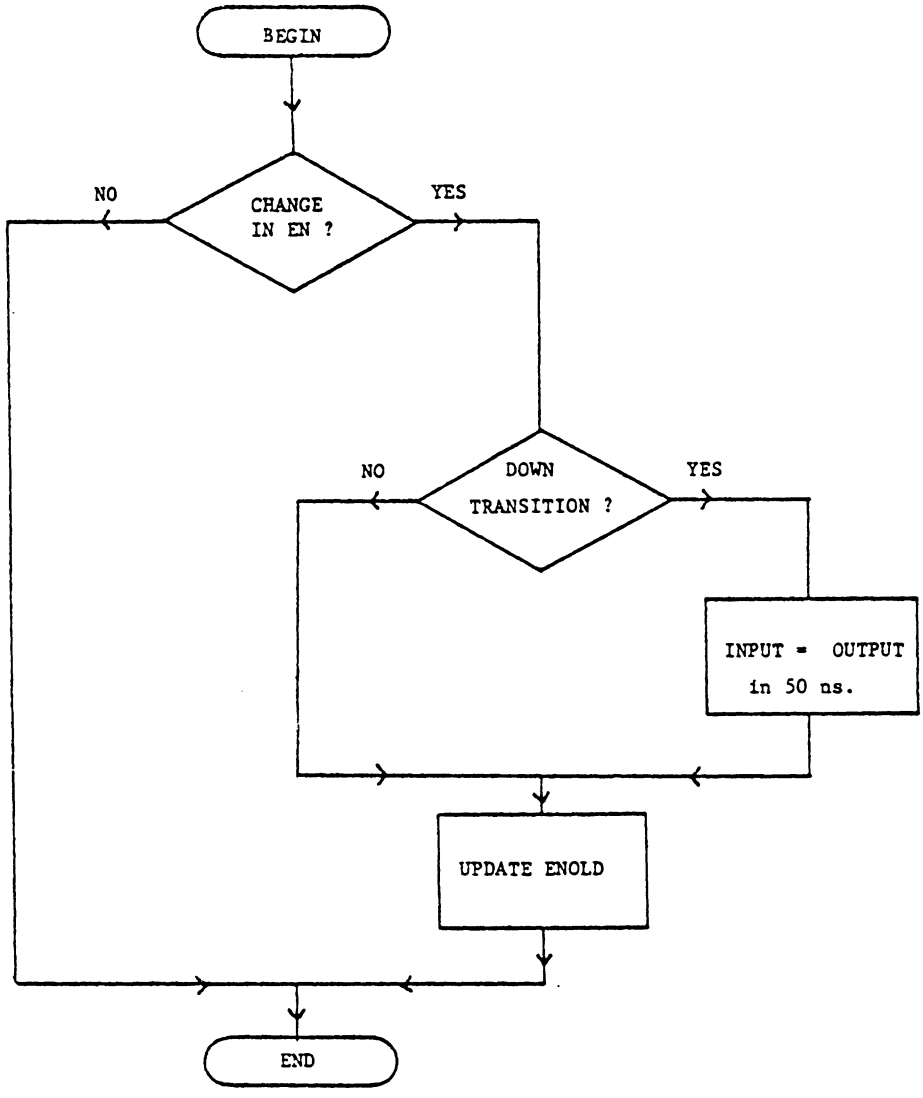

Figure 5: Flow-chart for 8 bit buffer

are pseudo pins used to control the program flow in the module. Since we wish to model the INTEL 8212 is an asynchronous device, the module procedure is 'called' or the module code is executed for a change in any of the input pins. In general, the module may be called by a transition on any of the input pins or by a 'self-call'. A self-call occurs when, while executing a module procedure, a schedule is made for the same module to be called at some later time (for the purpose of further processing) [12]. In a digital device various signals propagate simultaneously; this is simulated using self-calls.

As a first step in modeling, the specifications for the chip are studied and the internal logic equations of the device are found to be:

1. SEL = DS1'.DS2
2. EN = MD + SEL
3. DCLK = MD'.STB + MD.SEL

Referring to the manufacturer's specifications [11], the internal timing structure (signal propagation) for the model is developed using the following events:

1. The enable signal (EN) propagates to output buffers in 35ns.
2. Propagation time of the CLR signal is 30ns.

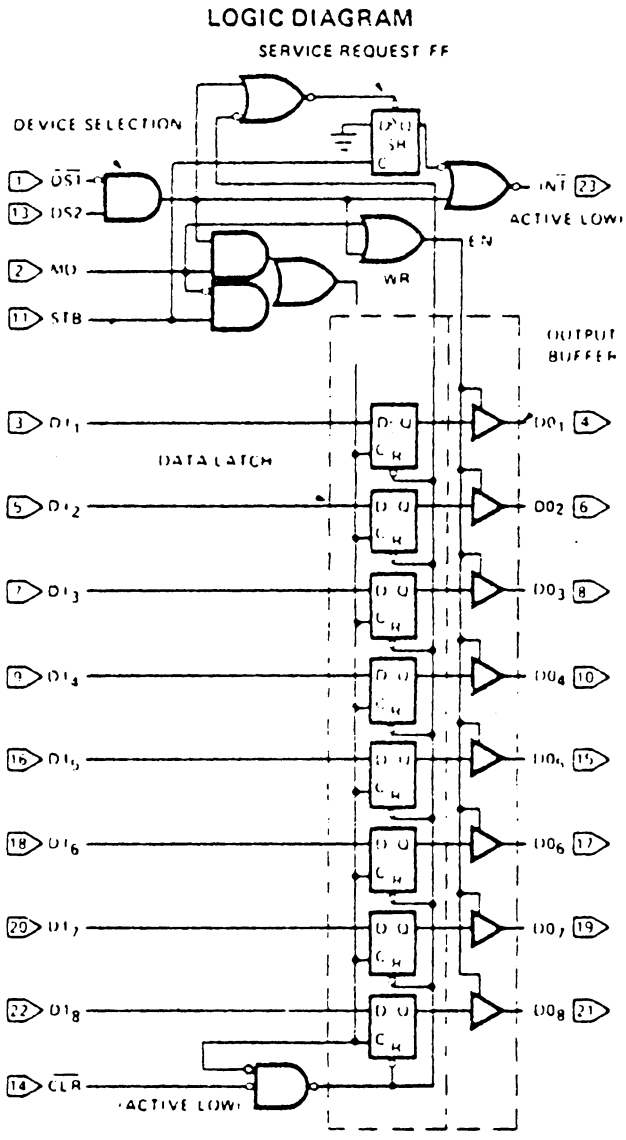

Figure 6: Circuit Diagram of INTEL 8212 Chip

3. Clock to D flip-flops (data latch) is generated 30 ns after the change in input signals.
4. With DCLK and EN high, data on input pins propagates to output pins in 40ns.
5. The INT flip-flop is clocked in 30 ns., for a high to low transition on STR.
6. The INT f/f is reset 30 ns after a change in SEL or CLR signals.
7. The data output buffer has a propagation delay of 10 ns.

On the basis of the above equations and specifications, the model is implemented using the GSP assembly language. This assembly language code can be roughly divided into three sections: i) By comparing the previous values with the present values, the pin on which a transition occured is located. ii) The changed input is propagated through various functional blocks using self calls and thus internal timing is simulated. iii) The old values of the input signals are updated so that the next transition on the input signals can be located.

Referring back to Figure 1 (the source code for an 8 bit latch), these three sections are distinct, since the device operation is simple.

Following this approach, more complex chips such as the INTEL 8080 microprocessor and it's support chips were modeled using the GSP assembly language. Under a Rome Air Development Center (R.A.D.C.) contract, a microprocessor system consisting of INTEL 8080, RAM, ROM and 8251 (serial I/O), 8255 (parellel I/O) chips was simulated [9].

Chapter III

FAULT INSERTION TECHNIQUES

With the increase in complexity of the LSI device (each device consisting of 10,000 or more gates), it is not feasible to carry out the fault analysis of the device for all possible stuck-at faults. Moreover, in order to simulate the device at the gate level, a detailed description of the chip is required which is not readily available. These problems can be alleviated by simulating the device at the functional level and injecting functional faults instead of gate level faults. The number of faults to be injected at the functional level are much smaller than the number of faults to be injected at the gate level and the simulation can be carried out without any need for the detailed gate-level information [4].

3.1 FAULT INSERTION TECHNIQUES

When simulating a digital device at the functional level using GSP, the faults injected in the model can be roughly divided into five different categories:

1. Faulty micro-operations
2. Timing faults
3. Internal stuck-at faults

4. Interconnect faults
5. Transient faults

3.1.1 Faulty micro-operations

When simulating the LSI chip at the functional level, it is divided into functional blocks, where each block is realized by a number of micro-operations. Simulating these micro-operations in the proper sequence results in duplicating the actual chip behavior. The micro-operations performed by the LSI devices can be divided into following categories:

1. Data transfer
2. Decoding
3. Data manipulation
4. Sequencing the state changes
5. Control signal generation

In general, these micro-operations are common to a number of instuctions or operations of the device and a faulty micro-operation may manifest itself in a number of different ways. So it is important to study the effect of faulty micro-operations on the chip behavior.

3.1.1.1 Data Transfer Operations

There are different types of data transfer operations such as:

1. Address set up before READ or WRITE operations
2. Read data
3. Data write operation
4. Data transfer between two internal registers

The Mano machine is a hypothetical processor with a simple instruction set [15]. The GSP model for this processor is given in appendix B. For a Mano machine, the instruction fetch cycle is simulated as shown in Figure 7. The address is set up by first two lines of the code. To inject the fault such that the higher byte of the address always appears to be zero, the second line of code from Figure 7 is modified as follows:

```
MOV   #0, MARH   ; MARH = 0
```

Thus, higher byte of the address will always appear to be zero.

After one clock pulse from address setup, the data is read from memory into the memory buffer register, using the following code.

```
MOV  DINL, MBRL    ;data input to
MOV  DINH, MBRH    ;memory buffer register
```

Suppose the fault to be injected in this operation is such that, the lower two of the data input pins are stuck-at one. This is incorporated by modifying the first line of code by:

```
MOV   #1, RW     ;READ active
MOV   PCL, MARL ; setup the
MOV   PCH, MARH ; address
                ;
JSR   CLOCK      ; wait for 1 clock
MOV   DINL,MBRL ; read
MOV   DINH,MBRH ; data
INC   PCL
.
```

Figure 7: Fetch cycle in Mano machine

```
OR   #3, DINL, MBRL
```

so that, at the end of the data read, the lower two bits of data will be set to 1's.

Consider the data output or write operation. Appendix A has the source code for the GSP model of the INTEL 8212 (8 bit parallel I/O) chip. Suppose this device has a faulty output buffer e.g. the output buffers are never enabled. Referring to appendix A, the output buffers are simulated by the subroutine 'OUT' as shown in Figure 8. When simulating a good module, the data output pins are tristated (by outputting all 1's) if the enable signal EN is 0. Otherwise the output of the data latch is sent to the output pins in 10ns. The fault is injected in the module by modifying the output subroutine as shown in Figure 9, such that the output buffers are never enabled.

When dealing with a bi-directional bus, tri-stating the bus at particular times is critical. On GSP, a high impendence state is simulated by outputting all 1's on the output pins. Referring to Figure 8, when ENS=0, the data bus is tri-stated. This operation can be faulted by replacing the last two lines of this code by

```
HIZ:   RTS
```

```
OUT: BEQ ENS,HIZ              ; EN = 0 ?
     MOV(W10) DOD,DO          ; DO = DOD IN 10NS
     RTS
HIZ: MOV(W10) #255,DO         ; DO =ALL 1'S IN 10NS
     RTS
```

Figure 8: GSP code for the output buffer in INTEL 8212

```
OUT: BEQ ENS,HIZ              ; EN = 0 ?
     MOV(W10) #255,DO         ; DO = all 1's IN 10NS
     RTS
HIZ: MOV(W10) #255,DO         ; DO =all 1'S IN 10NS
     RTS
```

Figure 9: INTEL 8212 faulty output buffer

As an example of data transfer between two registers, consider a CPU with four internal registers A1, B1, C1, & D1. The data transfer from the register A1 to others can be modeled as follows.

```
        IDX  IR(0), 2, 1     ; two LSB of IR =>  1.
                             ;
        MOV  A1, TBL1@1      ;(A1) => appropriate register.
                             ;
TBL1:   BYT  A1, B1, C1, D1
```

This code can be modified so that the data from the register A1 is moved to the incorrect destination register. For the correct operation, the registers A1 through D1 are selected for the indexed register one (@1) containing 0 to 3 respectively. So by rearranging the sequence of the registers in TBL1 as shown below, the fault can be injected so that instead of register B1, the data will be written into the register C1. If the data transfer in the model is done by:

```
MOV  A1, B1
```

then the above fault is injected by modifying this line of code by

```
MOV  A1, C1
```

so that the faulty data transfer takes place.

Consider the fault such as multiple register select on WRITE e.g.

MOV A1,B1 also writes into register C1.

Referring to the Figure 10, this fault can be injected by adding two lines to the existing code so that the data movement from A1 to B1 also copies A1 to C1; move to C1 or D1 are unaffected.

```
TBL1:  BYT A1, C1, C1, D1
```

3.1.1.2 Decoding

In case of a processor, once an instruction is read from memory, it has to be decoded before the processor can take any action. For a simple processor, this is done in one step, while for a procesor with a large instruction set, it is done in several steps. Figure 11 shows the GSP code to decode the instruction for Mano machine. Each instruction is a 16 bit word. The bits 15 and 12-14 are transferred to registers I and OPR respectively. At the end of instruction fetch, the next cycle is decided depending on the contents of registers I and OPR. This is done by modifying the contents of F and R flip-flops.

The control is transferred to an Indirect cycle for the following conditions.

1. (OPR) = 7

```
        MOV   A1, TBL1@1      ; correct operation
                              ;
        MOV   A1, TBL2@1      ; faulty operation
                              ;
TBL1:   BYT   A1,B1,C1,D1
TBL2:   BYT   A1,C1,C1,D1
                              ;
```

Figure 10: Multiple Register Select on WRITE

```
     IDX  MBRH(4),3,1    ; bits 12-14
     MOV  @1, OPR        ; to OPR
     IDX  MBRH(7),1,2    ; bit 15
     MOV  @2, I          ; to I
     JSR  CLOCK          ;
                         ;
     BEQ  #7, OPR, 1010 ; Is it an indirect
     BEQ  I, 1010        ;      cycle ?
     OR   #1, FR         ; no.
     BRU  100            ; next cycle
1010:OR   #2, FR         ;indirect cycle
 100:IDX  FR(0),2,1      ;
     JSR  CLOCK
     BRU  TBL1@1
TBL1:BYT  1000,2000,3000,4000
```

Figure 11: Instruction decoding in Mano machine

2. (OPR) .ne. 7 and I=0

Referring to Figure 11, by deleting the line

```
BEQ  I, 1010
```

a fault can be injected such that the control is not tranferred when the second condition is valid. Instead, an execution cycle follows.

In case of memory chips, for each READ or WRITE operation, the address is decoded to select proper memory location. The GSP code for a decoder in 32 byte ROM is as follows.

```
        MOV  ADDR, ADB      ; move ADD to buffer
        IDX  ADB(0), 5,1    ; @1 has 5 bits of ADB
        MOV  MEM@1, TEMP8   ; DATA READ
MEM:    BYT  01, 00, 31, 01,...
```

Where MEM is a table containing the data. To inject a fault such that, the higher half of the memory is not accessed any time (reading from locations 16 through 31 results in reading from 0 through 15 respectively), the code is modified as given below.

```
        MOV  ADDR, ADB
        IDX  ADB(0), 4,1    ; 1 has 4 LSBs of ADB.
        MOV MEM@1, TEMP8
```

Thus the indexed register one (@1) gets lower four bits from the register ADB. This is equivalent of setting the most significant bit to zero, which results in accessing only lower half of the memory.

3.1.1.3 Data manipulation

Some common data manipulation operations are as follows:

1. Clear register
2. Complement the register contents
3. Increment or decrement the register contents
4. Bit set/reset operation
5. Logical AND, OR, XOR

Once the code which simulates these operations is located, the fault can be injected by making appropriate modifications in this code. One such example is given below. When a processor executes an instruction manipulating data, the contents of accumulator (AC) are modified and the result is stored back in AC. As an example, consider the following instruction.

Increment accumulator.

When executed, the contents of accumulator are incremented by one and the carry flag CY is set in case of an overflow. When simulating on the GSP, the simulator register C is au-

tomatically set on overflow. Using this fact, the above instruction is simulated as follows:

```
INC  AC     ; increment
MOV  C, CY  ; store carry
```

To inject a fault such that the carry is not affected by this operation, the second line of code is deleted.

Similarly, the faults can be injected in other data manipulation operations.

3.1.1.4 Sequencing the state changes

For an LSI device, each operation consists of a number of micro-operations performed in proper sequence. So the improper sequence of these micro-operations will result in the faulty operation. As an example, for the Mano machine, the instruction 'AND AC' consists of following micro-operations.

1. Setup the address and generate the READ signal.
2. Read the data.
3. Perform logical AND between the data read and the accumulator and store the result in the accumulator itself.

If the address setup or the generation of the READ signal is delayed by one clock pulse, incorrect data will be read resulting in faulty operation. The GSP code for this operation

is given in Figure 20. To inject the fault such that the address setup is delayed by one clock cycle, the second line of code in Figure 20 is placed after the first line. Thus the address will be setup one clock pulse from the generation of READ signal.

As another example, the GSP source code given in Figure 11 simulates the instruction decoding and the state transition for Mano machine. The TBL1 is used to determine the next machine cycle. By rearrenging the entries in the TBL1(Figure 11), the faulty sequencing of the states can be simulated.

3.1.1.5 Control signal generation

In case of Mano machine, the micro-operations such as address setup and READ signal generation are simulated by the first three instructions of Figure 7. By modifying the third line from

```
MOV        #1, RW     ;
```

to

```
MOV(W50)   #1, RW     ; RW=1 after 50 ns.
```

the READ signal can be generated 50 ns later than present simulation time. By modifying this line to

```
MOV(W50)   #0, RW     ; RW=0
```

an incorrect READ signal can be generated.

3.1.2 Timing Faults

A micro computer system consists of several LSI chips such as CPU, RAM, ROM and I/O interface chips. The execution of the microprocessor program consists of a sequence of READ and WRITE operations; each transfers a byte of data between the CPU and a particular memory or I/O device. These READ and WRITE are the only communications between the processor and the outside world. This orderly sequence of events requires precise timing, and synchronization of various signals is critical. Thus, it is important to study the effect of the faulty signal timings.

Consider a faulty processor such that generation of the READ pulse is delayed by 200 ns. The timing of the correct READ pulse can be simulated by the subroutine MEMRD as shown in Figure 12. The delays of 100 and 600 ns are associated with the labels DFALL and DRISE respectively. The READ pulse is assumed to be active low. Execution of this subroutine results in generation of the READ pulse of 500 ns duration, starting 100 ns from the present simulation time. The timing fault is injected by changing the timing delays associated with the labels DFALL and DRISE as shown in Figure 13. Increasing these numbers by 200 each results in generating the READ pulse 200 ns later.

```
PIN  READ(1)
EVW  DFALL(100), DRISE(600)
     ;
MOV(DFALL) #0, READ       ; read = 0 after 100ns.
MOV(DRISE) #1, READ       ; read = 1 after 600ns.
                          ;
```

Figure 12: Generation of READ signal

```
PIN  READ(1)
EVW  DFALL(300), DRISE(800)
                    ;
MOV(DFALL) #0, READ ; read = 0 after 300ns.
MOV(DRISE) #1, READ ; read = 1 after 800ns.
                    ;
```

Figure 13: Delayed READ signal

In general, by changing these timing delays (defined in the declaration section of the module), the timing faults can be injected. Though this procedure allows the user to inject the timing faults, it is still necessary to make these modifications in the module code and then assemble the code before each fault simulation. To overcome this limitation, a command 'J M' was added to the existing set of the GSP monitor commands. By executing this command, the timings of all the signals in the module M are jittered (within +/- 20% randomly). Using this feature, critical timings for the system can be tested by a single simulation run. Since the device code is not modified, the GSPLNK file consisting of good modules can be used.

3.1.3 Internal stuck-at faults

An LSI device has storage elements or internal registers and the bit stuck-at faults in these registers are of interest for the test engineer. Also, such faults are likely to occur in case of memory chips.

Referring to the example of the processor (from the section 3.1.1) with four internal registers A1, B1, C1 and D1, the bit stuck-at fault in one of the registers can be injected in the module as shown in the following example. Suppose the fault to be injected is the bit 0 of register C1

stuck-at one. As the first step, the GSP model is studied to note all the occurences where the data is written into register C1. Then, the fault is injected by adding one line of code following all these occurences, where the extra code added is given by :

```
OR #1, C1        ; set LSB of C1.
```

This will result in setting the bit zero (0) of register C1 after every data transfer into register C1 and the least significant bit of C1 will always appear to be stuck at one.

Assuming C1 to be an 8 bit register, the stuck-at zero fault in bit 0 can be injected by replacing the extra line of code by a new code:

```
AND #254, C1        ; reset LSB of C1
                    ; 254 d => 1111 1110 b
```

By varying the immediate data in these instructions, the faulty bit can be changed. Using this technique, bit stuck-at faults can be injected in any of the internal registers of the module.

3.1.4 Interconnect faults

For a system consisting of several reliable LSI devices, interconnect faults (e.g. dry solder joint, defects in the printed circuit boards) are very likely [13]. When simulat-

ing the digital system in GSP, the interconnect faults such as wired AND, wired OR, line stuck-at one or zero can be injected in the system by making appropriate modifications in the Command file.

Referring to Figure 1, the connection between various pins of different modules is done by monitor command C. As an example, the following command

```
C  1,1  2,5  3,8
```

will connect pin 1 of module 1 to pin 5 of module 2 and pin 8 of module 3. If this line in the Command file is replaced by

```
C  1,1       3,8
A  C  2  5  0
```

it will result in no connection to pin number 5 of the module 2; instead the second command 'A C' will initialize the pin 5 to zero. This will result in pin 5 of module 2 stuck at zero for the entire simulation session. By changing this second command by

```
A  C  2  5  1
```

the pin 5 can be initialized to one and will appear to be stuck at one. Thus using this method, the stuck-at faults in the input pins of any GSP module can be injected.

The procedure to inject wired AND and wired OR faults require a block of code added to one of the GSP models or an

extra module can be added to the system. The GSP code for such module is given in Figure 14. By making connections to pins 1 through 4 and taking the outputs from pins 5 and 6, the wired AND or wired OR faults in any two lines can be injected.

```
REG(1)  TEMP1, TEMP2

PIN   A1(1), A2(2), O1(3), O2(4)
PIN   AOUT(5), OOUT(6)

EVW   Z(0)
  .
  .
AND     A1, A2, TEMP1
OR      O1,O2,TEMP2
MOV(Z)  TEMP1, AOUT
MOV(Z)  TEMP2, OOUT
EXIT
```

Figure 14: Interconnect faults (shorts)

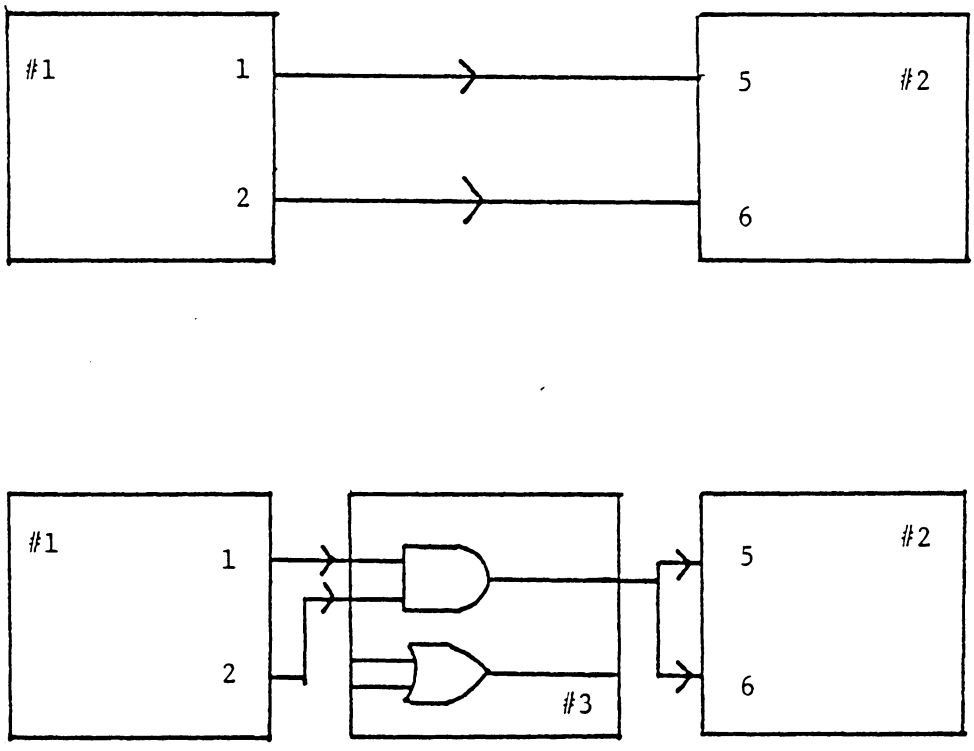

Figure 15: Example of wired AND fault

3.1.5 The Transient Faults

It has been observed that permanent faults cause only a fraction of all detected errors, typically less than 10 percent [14]. In light of this, when simulation is used for the fault analysis of an LSI device, injecting transient and intermittent faults is very important.

When simulating on the GSP, three types of transient faults can be injected.

Functional faults: The fault code is imbedded in the module (as explained in the next section). By adding the events to the time queue using Moni command 'A', inputs to certain pseudo pins are changed thus resulting in a transient fault which changes the module function.

Inter-connect faults: The Moni command 'C' is used to connect two or more pins where as command 'D' disconnects them. Using these two commands at different simualtion times, intermittent faults can be injected.

Signal faults: Using Moni command 'A', an event is added to the time queue so that a bit in the internal registers of any module can be set or reset. Thus, at any specified simulation time, a signal fault can be injected.

Using the above two techniques, various transient faults can be injected in the system to observe their effect on the system performance.

3.2 IMBEDDED FAULTS

In the previous section, various techniques were discussed to inject different types of faults in LSI modules. These methods allow the user to model the faults with significant accuracy. However, when injecting a large number of faults in a particular module, it is necessary to edit the source code, make the changes, assemble the model to form the object code and finally to reform the GSPLNK file. Bypassing these steps will result in considerable amount of saving in the user's as well as CPU time. This can be accomplished by modeling the LSI device such that the faults are imbedded in the model. Extra pseudo pins are added to the module so that when the module procedure is executed, the program flow is controlled depending on the signals on these pseudo pins. Figure 1 shows the source code for an 8 bit buffer. It consists of 8 data lines in parallel. In order to simulate the buffer with all possible (simple) stuck-at faults, it would be necessary to create 16 faulty modules, each with a unique stuck-at fault. This can be avoided if the model code is modified as shown in Figure 16.

A stuck-at fault is injected whenever the pin 'FLT' is set to one. The faulty line is selected by the (binary) value on the pins 'LINE'. The type of fault (stuck-at 1 or stuck-at 0) can be chosen with the value on the pseudo pin

```
        REG(8)     TEMP, TEMP8
        PIN        TYPE(1), FLT(2), LINE(3,5)
        EVW        DEL(50)
          .
        MOV        INP, TEMP8
        BEQ        FLT, NOFLT
        MOV        LINE, @2
  TBL:  BYT        1,2,4,8,16,32,64,128
        MOV        TBL@2, TEMP
        BEQ        TYPE, SA0
        OR         TEMP, TEMP8
        BRU        NOFLT
  SA0:  COM        TEMP
        AND        TEMP, TEMP8
NOFLT:  MOV(DEL)   TEMP8, OUTP
        .
        .
```

Figure 16: Imbedded stuck-at fault in 8 bit buffer

'TYPE'. Thus, by varying the inputs to these pseudo pins, the same module code can be used to simulate a good buffer or a faulty one with a stuck-at fault in one of the lines.

By extending this approach the stuck-at faults can be injected in the device internal registers where the input to the pins 'SEL' is used to select the faulty register. As an example, consider a device with four internal registers A1, B1, C1 and D1. The extra pins and the code added to the module is as shown in the Figure 17. In this example, with FLT=1, TYPE=1, LINE=0 and SEL=01, the bit zero of the register B1 will appear to be stuck-at one. This method is very useful for injecting stuck-at faults in memory modules.

```
        REG(8)   TEMP, TEMP8

        PIN      TYPE(11),FLT(12),LINE(13,15),SEL(16,17)

        BEQ      FLT, NOFLT     ;
        MOV      SEL, @3        ; @3 points to faulty reg.
        MOV      LINE, @2       ; @2 points to faulty bit
        MOV      TBL1@2, TEMP   ;
        BEQ      TYPE, SA0      ; TYPE=0 => S-A-0
        OR       TEMP, TBL2@3   ;
        BRU      NOFLT          ;
 SA0:   COM      TEMP           ;
        AND      TEMP, TBL2@3   ;

 TBL2:  BYT      1,2,4,8,16,32,64,128

 TBL3:  BYT      A1, B1, C1, D1

NOFLT:      .
            .
```

Figure 17: The bit stuck-at in the internal register

3.3 A SYSTEMATIC APPROACH TO MODEL DESIGN

In section 2.4.2, a modeling technique was discussed using the INTEL 8212 as an example. Since simulation of digital chips is used for fault injection and test validation, it is important to structure the model code so that various functional blocks in the chip are distinct. In general, the chip operation can be divided into several functional blocks, where each block is realized by simulating a number of micro-operations in sequence. These micro-operations are common to more than one macro-operation. If these micro-operations are scattered in the module code, in order to simulate faulty micro-operations correctly, it will be necessary to locate them in the entire module code. In which case, the fault injection procedure may become difficult and inaccurate. Thus, proper structuring of the module code is important.

If each micro-operations is modeled in the form of a subroutine, the problem will be reduced to choosing the correct subroutine and modifying the code in only one place. Thus, the procedure to inject functional faults will be simplified.

As an example, a hypothetical processor with a small instruction set (Mano machine) [15] is modeled. The register configuration for this machine is as shown in Figure 18.

The memory unit has capacity of 4096 words and each word contains 16 bits. Referring to Figure 19, processor operation consists of these four machine cycles.

1. INSTRUCTION FETCH
2. EXECUTION CYCLE
3. INDIRECT CYCLE
4. INTERRUPT CYCLE

At the end of one cycle, the next cycle is decided by contents of F and R flip-flops. Each instruction has two or three machine cycles where each machine cycle is made up of four clock pulses T0 through T3.

In the instruction fetch cycle of the Mano machine, an instruction is read from memory. In terms of register transfer relations, this process consists of following steps. In each clock cycle, following events take place.

1. During T0 clock pulse, the address which is in PC, is transferred to MAR.
2. The instruction is transferred from memory to MBR during T1 clock. Also the program counter is incremented.
3. During T2 clock, the operation part and the mode bit of the instruction are transferred from MBR into OPR and I respectively, while the address part of instruction remains in MBR.

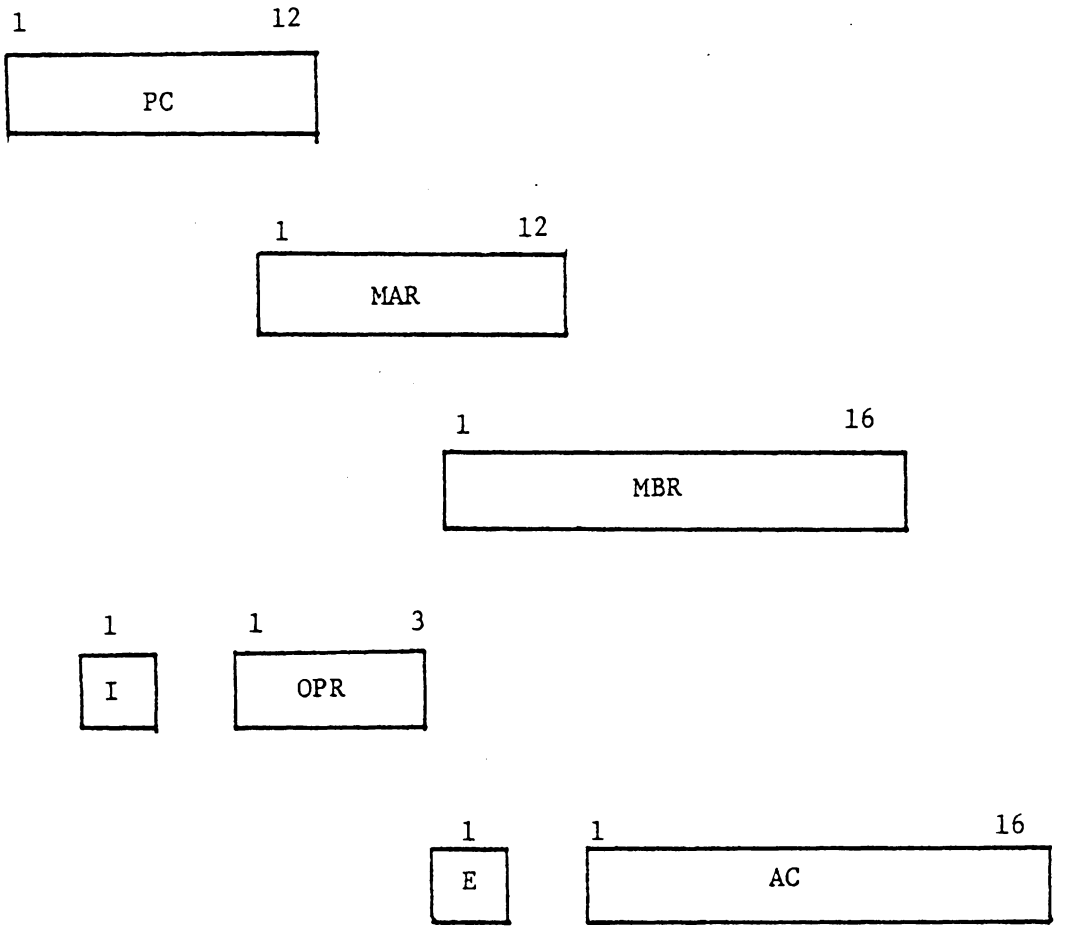

Figure 18: The register configuration in Mano machine

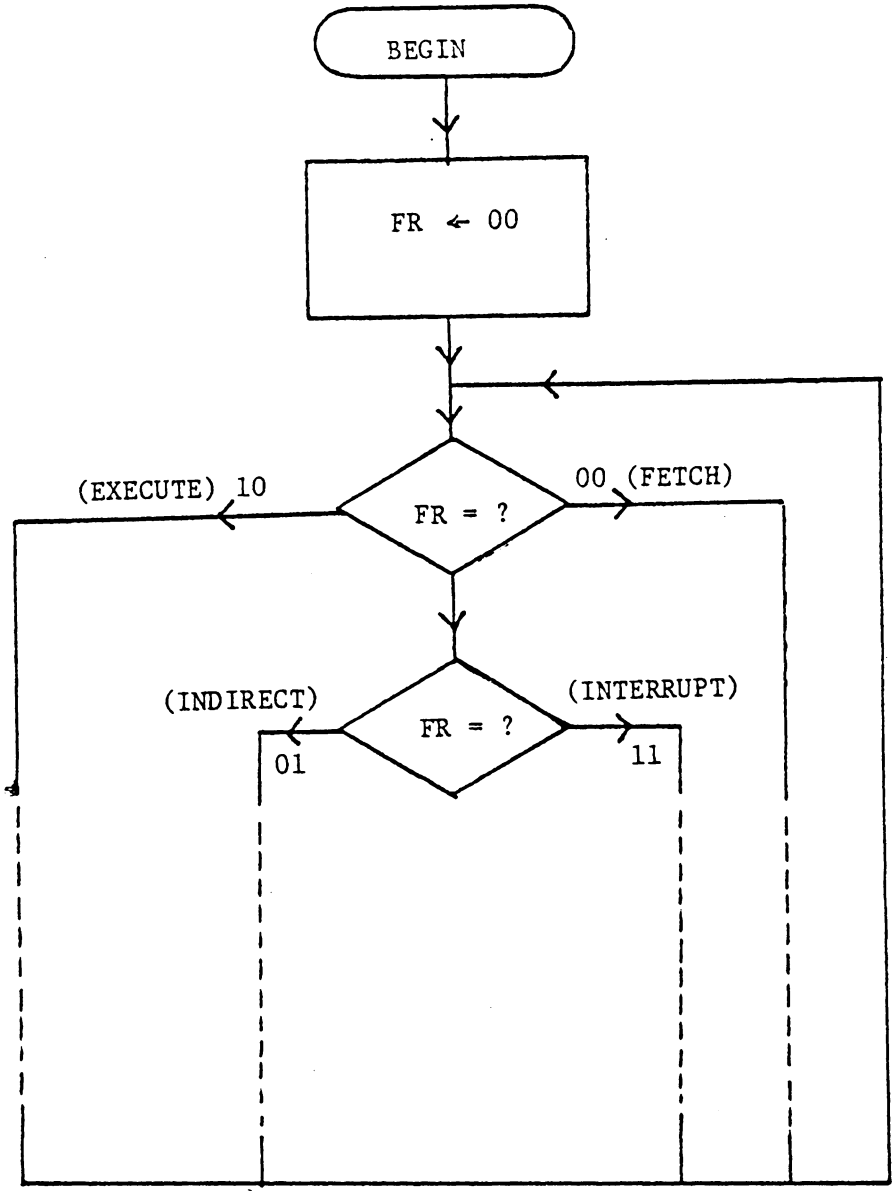

Figure 19: The Flow Chart of Mano Machine Operation

4. In T3, depending on contents of OPR and I, the next cycle is determined as follows.
   a) If (OPR) = 111, the F flip-flop is set, indicating the next cycle as execution cycle.
   b) Otherwise, the instruction is memory reference instruction. Then, depending on contents of the I register, following decesion is made.
      i) (I) = 0 indicates direct instruction and by setting F flip-flop to 1, the control goes to execution cycle.
      ii) For (I) = 1, control is transferred to indirect cycle by setting the R flip-flop.

The other machine cycles can be similarly explained in terms of register transfer relations. For detail description, refer to [15].

Using the GSP assembly language, the Mano machine is modeled as shown in Appendix B. The model is structured so that each machine cycle is divided into micro-operations. Each micro-operation is represented by a subroutine. Thus, each machine cycle is made of subroutine calls where the subroutines simulate internal data transfer or the data manipulation in the internal registers.

As an example, consider the instruction 'AND to AC'. This instruction performs the AND logic operation on pairs of

bits in AC and the memory bit specfied by the effective address. The result of the operation remains in AC.

Figure 20 shows the GSP code to simulate this instruction. The micro-operations such as read pulse generation, address set up, reading data or the clock pulse generation are common to more than one instruction. So they are represented in the form of the subroutines (appendix B).

On the other hand, the actual AND operation is unique and is executed in only one place. So it is simuated locally.

After validating the module code such that it duplicates the behavior of the actual chip, faults can be inserted using the various techniques explained above. In the well structured model the faults can be imbedded in the different subroutines which can be controlled by inputs to the pseudo pins of the module. As an example, consider subroutine five (SUB5) in the Mano machine module as shown in Figure 21. Here the contents of memory buffer register (MBR) are added to the accumulator.

(MBR) + (ACC) => (ACC) and CARRY

In the Mano machine the accumulator is 16 bits wide while the GSP simulator at present can handle upto 8 bit registers; so the above operation is split into two eight bit additions. The fault can be imbedded in the routine such that the subroutine simulates faulty behavior when the pin 'FADD'

```
3100: JSR  SUB9          ; READ active
      JSR  SUB1          ; setup ADDRESS
      JSR  CLOCK         ; clock pulse
      JSR  SUB2          ; read data
      JSR  CLOCK         ;
      AND  MBRH, ACH     ;logical AND
      AND  MBRL, ACL     ;
      BRU  5000          ; check interrupt
```

Figure 20: GSP code for instruction AND to AC in Mano machine

of the module is set to one. The modified code is given in Figure 22 to inject a fault so that the carry does not get set.

```
SUB5:  ADD   MBRL, ACL
       BEQ   C, SKIP1
       ADD   #1, ACH
SKIP1: ADD   MBRH, ACH
       MOV   C, CARRY
SKIP2: RTS
```

Figure 21: Subroutine to ADD in GSP module of Mano machine

```
SUB5:  ADD   MBRL, ACL
       BEQ   C, SKIP1
       ADD   #1, ACH
SKIP1: ADD   MBRH, ACH
       BNE   FADD, SKIP2
       MOV   C, CARRY
SKIP2: RTS
```

Figure 22: ADD routine in Mano machine with imbedded fault

Chapter IV

THE FAULT SIMULATION SYSTEM

In Chapter 3, different methods to inject faults in the GSP module of a digital chip were discussed. Once this is done, the system with a faulty module is simulated to study the effect of these faults on the model response.

When simulating a simple digital chip like a counter or a buffer, the system can be completely tested by applying all possible combinations of inputs and studying the outputs. Figure 23 shows the block diagram of the eight bit latch. Using GSP monitor command 'A', the pulse to the strobe pin (#1) can be generated as follows.

```
;     mod pin  time  value
   A   1   1    0     0        ; EN = 0 at T=0
   A   1   1   1000   1        ; EN = 1 at T=1000
   A   1   1   1400   0        ; EN = 0 at T=1400
;
```

This sequence of commands results in a pulse with rising edge at T = 1000 ns. and duration of 400 ns. which will enable the latch. Using this procedure, any input vector can be applied to the system at a predetermined simulation time.

When simulating a small system consisting of few simple modules (e.g. Flip-flops, counters, buffers), the system in-

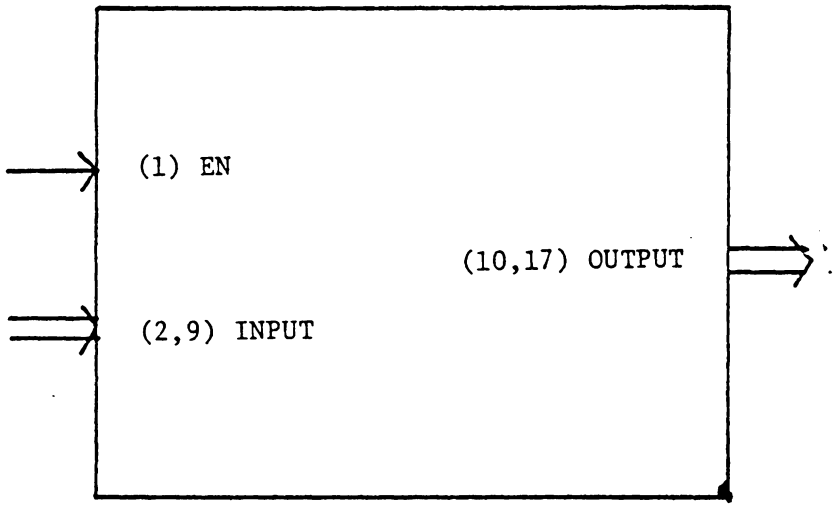

Figure 23: The gated buffer(latch)

put can be applied using the Command file. In chapter 5, this procedure is used to simulate a system consisting of INTEL 8212 (8 bit I/O buffer). This method is simple and very effective when simulating small and simple systems with few input pins.

When simulating a bigger system with several modules for LSI chips, this procedure is impractical as the input vector set becomes very large. In the case of a system consisting of microprocessor, memory and peripherals, it is not feasible to test the system by applying input vectors with the help of the Command file. Also, INTERRUPT, WAIT and various other control signals (e.g. used for hand shaking etc.) increase the complexity of the problem.

This problem can be simplified by developing an assembly language test program and loading it into a memory module. When this system consisting of a processor, memory and other modules is simulated, the prosessor module executes the program and as a result, the set of inputs is applied to a particular module and the response is read by the processor. Using this procedure, the module can be validated or the test program can be developed.

When the simulation is used for validating the test program to test an LSI device, the first step is to develop a model using GSP assembly language. Then, a set of likely

faults for the device is chosen and using the techniques discussed in chapter 3, faulty modules or the Command files are formed. The system without a faulty module is then simulated to find the simulation time required to complete the test successfully. Using this as a maximum time, the system is simulated repeatedly for each fault. Then at the end of these simulation runs, the fault coverage of the test routine can be determined. At this point the test can be improved so that the fault coverage is increased. Using this method and simulating a large number of faults, a reliable test routine with high fault coverage can be developed.

In case of an LSI chip, the list of possible faults can be very large. For each fault run with a faulty module, the system (GSPLNK file) has to be recreated. Thus for a large number of fault simulation runs, the simulation time required can be significant. We next discuss mechanisms which were added to the GSP simulator to alleviate this problem.

4.1 SUBROUTINE RESULT

Functional level simulation can be effectively used to validate the diagnostic software for a processor based system. The test program is designed such that it checks the various functional aspects of the module in a particular sequence and at any point if the response is incorrect, a flag

is set indicating that a fault was detected and the program halts. Hence generally when simulating a faulty system the test routine is not completely executed.

When simulating such a system, all the user needs to know is whether the test program was completed or not and if it was, whether it detected the fault. In general, when simulating a system, the state of the device can be determined by knowing the pin values and the contents of the internal registers. By looking at the contents of Program Counter (PC), Instruction Register (IR) and various internal registers, the user gets all the information necessary concerning the simulation run. If a more detailed fault signature is required, the pin values and the register contents of other modules in the system can also be checked.

The GSP monitor command 'R' dumps the pins and the register contents for any module in GSPLNK. The monitor command 'S' ends the simulation session and brings the control to the host computer. The simulator program has been modified so that on requesting the 'S' command, subroutine RESULT is entered before returning control to the host computer. As shown in Figure 24, the user is asked if register and pin dump is needed for any module. The monitor reads in the module names and dumps the information in a log file. If the module name provided by the user is incorrect, the user is

told so. The monitor also writes the present time, the date and the names of GSPLNK file, Command file and the modules in the system.

Using this option one may carry out a number of simulation runs without trying to interprete the results. At the end using the information stored in the LOG file, each simulation run can be analysed. Thus it is not necessary to verify the simulation results at the end of every simulation run.

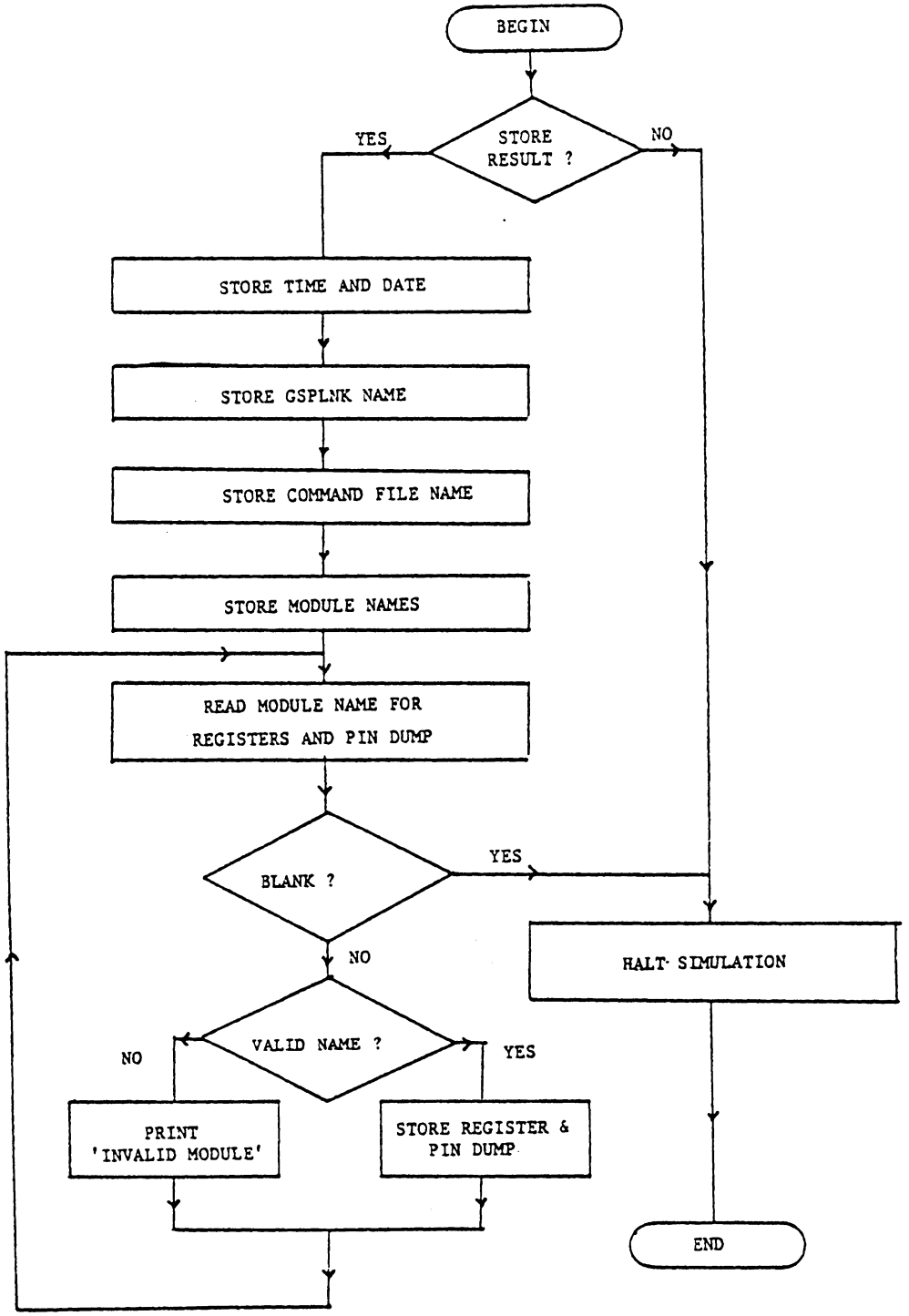

Figure 24: Flow chart of GSP routine RESULT

4.2 ROUTINE TO REPLACE A MODULE IN GSPLNK FILE

At present the GSP simulator can simulate a system consisting of upto 16 modules. The GSPLNK file which represents the system is formed by attaching modules in the proper sequence. To add a module into the GSPLNK file the program GSPLNKER is executed. The program consists of four steps:

1. Read in the existing GSPLNK file.
2. Read the object file (module) to be added.
3. Update the pointers and form new GSPLNK file.
4. Restore the new GSPLNK file.

Hence to run the system consisting of eight modules, GSPLNK file is formed by running the GSPLNKER program eight times. To simulate a system with a faulty module in place of a good module it is necessary to recreat the GSPLNK file. This old procedure of forming a new GSPLNK file for each fault run was very inefficient and time consuming.

To simplify this, the program GSPLNKER was modified such that when adding a module to the GSPLNK file, an array is formed which stores the pointers and thus keeps a record of the position of all the modules in the GSPLNK file. Program GSPREPLC FORTRAN was developed which uses this information and replaces any module in the GSPLNK file by another module (object file) specified by the user. The GSPREPLC program consists of the following steps:

1. Read GSPLNK file in primary array.
2. Read module number to be replaced and check if it is valid.
3. Form GSPLNK file with M-1 modules in secondary array.
4. Read in new object file (Mth module) and add to secondary array.
5. Point to (M+1) st module in primary array.
6. Copy M+1 to N modules from primary array on top of existing secondary array.
7. Restore modified GSPLNK file (secondary array).

Thus using this routine, GSPLNK file can be modified in a single step and it is not necessary to recreate the GSPLNK file. Figure 25 shows the procedure to replace a module. As an example, module 2 from the GSPLNK file 'SYSTEM1' is replaced by the 'ROM' module.

```
GSPSIM: GSPLNK FILE IS SYSTEM1
YOU WILL BE ASKED TO ENTER OBJ FILES.
A CARRIAGE RETURN BRINGS YOU OUT OF THE LOOP.

ENTER NAME OF OBJ FILE

ARE YOU SATISFIED WITH THE LOADING? (YES/NO) DEFAULT YES
n
DO YOU WISH TO:
     1) STOP
     2) ERASE GSPLNK FILE
     3) GET NEW GSPLNK FILE
     4) ADD MORE OBJ FILES
     5) REPLACE COMMAND FILE
     6) REPLACE A MODULE IN GSPLNK FILE
ENTER NUMBER [1..6] DEFAULT 1
6
ENTER THE MOD NUMBER AND NAME OF NEW OBJ FILE.
2 rom
EXECUTION BEGINS...
THESE ARE THE CURRENTLY LOADED MODULES:
MODULE  1 CONTAINS M8080
MODULE  2 CONTAINS ROM
MODULE  3 CONTAINS M8228SH
SATISFIED?(YES/NO) DEFAULT YES
Y

ALL DONE.
DO YOU WISH TO:
     1) STOP
     2) ERASE GSPLNK FILE
     3) GET NEW GSPLNK FILE
     4) ADD MORE OBJ FILES
     5) REPLACE COMMAND FILE
     6) REPLACE A MODULE IN GSPLNK FILE
ENTER NUMBER [1..6] DEFAULT 1
```

Figure 25: Replacing a module in GSPLNK SYS3

4.3 ROUTINE ICFAULT FORTRAN

One of the major advantages of using the gate level simulation is the simplicity in injecting stuck-at faults. In the previous chapter a way to modify the Command file such that stuck at faults at input pins of a module are injected was discussed. A digital system in general consists of several of these LSI chips connected together and interconnect faults are very likely faults. To simulate a system with all possible stuck-at faults, it is necessary to make two fault runs (s-a-0 and s-a-1) for each input pin. Therefore, it is necessary to create twice as many copies of the Command file as the number of input pins of the LSI device under test. Each version of the faulty Command file thus formed has a unique stuck-at fault and it is different from all others. The procedure of editing these files manually to add fault connection is time consuming and prone to human errors. The problem can be eliminated by automating the procedure to create faulty Command files.

Routine ICFAULT reads in the Command file and the module number(M) for which faults are to be inserted. It scans through the file and locates the connection to any pin of module 'M'. For every connection to module M, it creates two copies of the command file such that in one, the pin is stuck at 0 while in the other the pin is stuck-at 1. Figure

26 shows an example of this. If module 'M' has n input pins, the pogram ICFAULT creats 2n versions of the command files covering all possible stuck-at faults.(assuming only one fault exists).

```
C  1,2  2,5
C  1,3  2,8

GOOD CONNECTION


C  1,3  2,8
A C  2 5 0   ; 2,5 stuck-at zero

  FAULT I


C  1,3  2,8
A C  2 5 1   ; 2,5 stuck-at one

  FAULT II
```

Figure 26: Faulty interconnections

4.4 SIMULATION OF IMBEDDED FAULTS

In section 3.2, the method to imbed faulty code in the module was discussed. As an example, using similar technique, stuck-at faults can be imbedded in the memory module. Few extra pins and some extra code is added to the existing module so that stuck-at fault can be inserted in any of the memory location. By changing the inputs to these pins, the type of fault (s-a-0 or s-a-1) as well as the faulty bit can be modified. This method offers three advantages.

1. The same module code can be used to simulate a number of faults and thus it is not necessary to assemble the module source code for each fault injected.
2. The same GSPLNK file is used for a number of fault runs. The fault type can be varied by changing the Command file.
3. By adding events with the 'A' command (with the help of Command file), the inputs to the pseudo pins can be changed. Thus transient faults can be simulated.

As an example, Figure 27 shows pseudo pins added to the RAM module. The fault is injected only if the pin FAULT is set to 1. Depending on the value of the pin 'TYPE', s-a-0 or s-a-1 fault is injected and the input vector 'FBIT' determines the faulty bit. The faulty location can be selected by input to pin 'LOC'. Thus, by changing the input vector

to pins FBIT, LOC, TYPE and FAULT, the fault type and the faulty location can be varied. Figure 28 shows the code added to the (32 byte) RAM module.

Figure 29 shows a sample Command file to inject a transient fault in the RAM module. When the system is simuated with this Command file, bit 2 of location 7 will be stuck-at 0 from simulation time 100 to 1000 ns. From 1000 ns onwards, bit 5 of location 16 will appear to be stuck-at 1.

Using similar sequence of commands, the stuck-at faults in the internal registers can also be simulated.

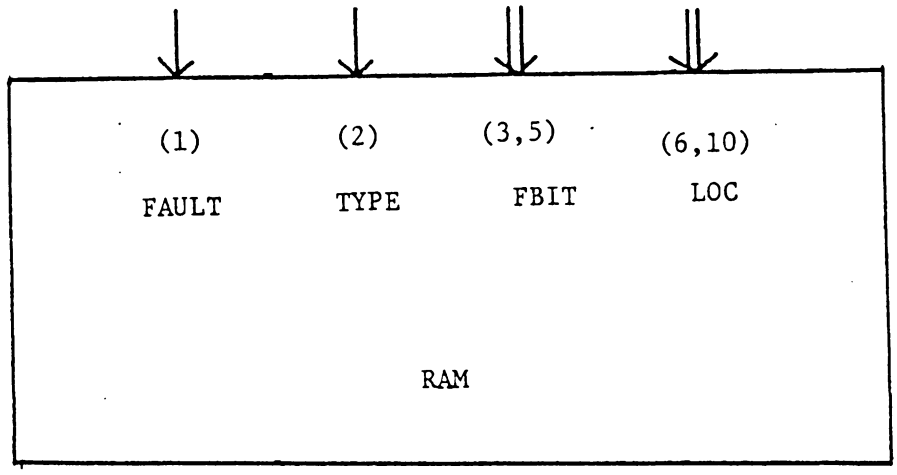

Figure 27: Pseudo pins added to the RAM module

```
        REG(8) TEM, TEMP8
        PIN    FAULT(1), TYPE(2), FBIT(3,5), LOC(6,10)

        BEQ    FAULT, NOFLT
        MOV    FBIT, @5            ;
        MOV    LOC, @6             ;
        MOV    TBL2@5, TEMP8       ;
        MOV    MEM@6, TEM          ;
        BRU    NOFLT               ;
  SA0:  COM    TEMP8               ;
        AND    TEMP8, TEM          ;
NOFLT:   .
         .
 TBL2:  BYT  1,2,4,8,16,32,64,128
```

Figure 28: Source code added to the RAM module to imbed stuck-at fault

```
A 1 1 100 1    ; pin FLT = 1
A 1 2 100 0    ; s-a-0
A 1 3 100 2    ; bit 2
A 1 6 100 7    ; location 7
               ;
A 1 2 1000 1   ; s-a-1
A 1 3 1000 5   ; bit 5
A 1 6 1000 16  ; location 16
```

Figure 29: Command file to simulate transient fault in RAM module

4.5 AUTOMATIC SEQUENCING OF SIMULATION RUNS

With the rising complexity in digital systems, it is necessary to make many simulation runs for detailed analysis and testing. Previously while using GSP, it was necessary to stop after every simulation run in order to note the results and then modify the system for the next fault simulation run. When simulating a system with several modules, this procedure is slow and impractical. It is necessary to simplify the procedure such that the simulation runs can be automated.

This is done by modifying the GSPSIM EXEC program such that several simulation runs can be made at a time without loss of information. Using the various techniques explained in Chapter 3, faulty Command files (to insert interconnect faults or imbedded faults) and faulty modules (faulty micro-operation, timing faults etc.) are formed in the beginning. Then using the GSPAUTO EXEC procedure, all the fault-runs for any particular module can be made as follows.

The GSPLNK file is formed by adding one module at a time to the system. Then the number of simulation runs and the required parameters are specified. In phase I the system is simulated with a good GSPLNK file and faulty command files; in phase II, the system is simulated with a good Command file and GSPLNK file with a faulty module(M'). Between two

runs of phase II, using the procedure explained in section 4.2, one faulty module in the GSPLNK file is automatically replaced by another one to form the system with a different fault.

When GSPLNK file is formed, the user specifies the Command file names in phase I. In phase II one has to specify the Command file, the module number to be replaced and the file names of the faulty modules. Figure 29 shows the procedure for this. At the beginning of each simulation run, the fault run number is displayed on the terminal. At the end of each run the Result file is updated and the information regarding this run is spooled to the reader so that the user may use it for detailed study.

At the end of all the simulation runs, the control is returned to the host computer; user interaction is thus reduced.

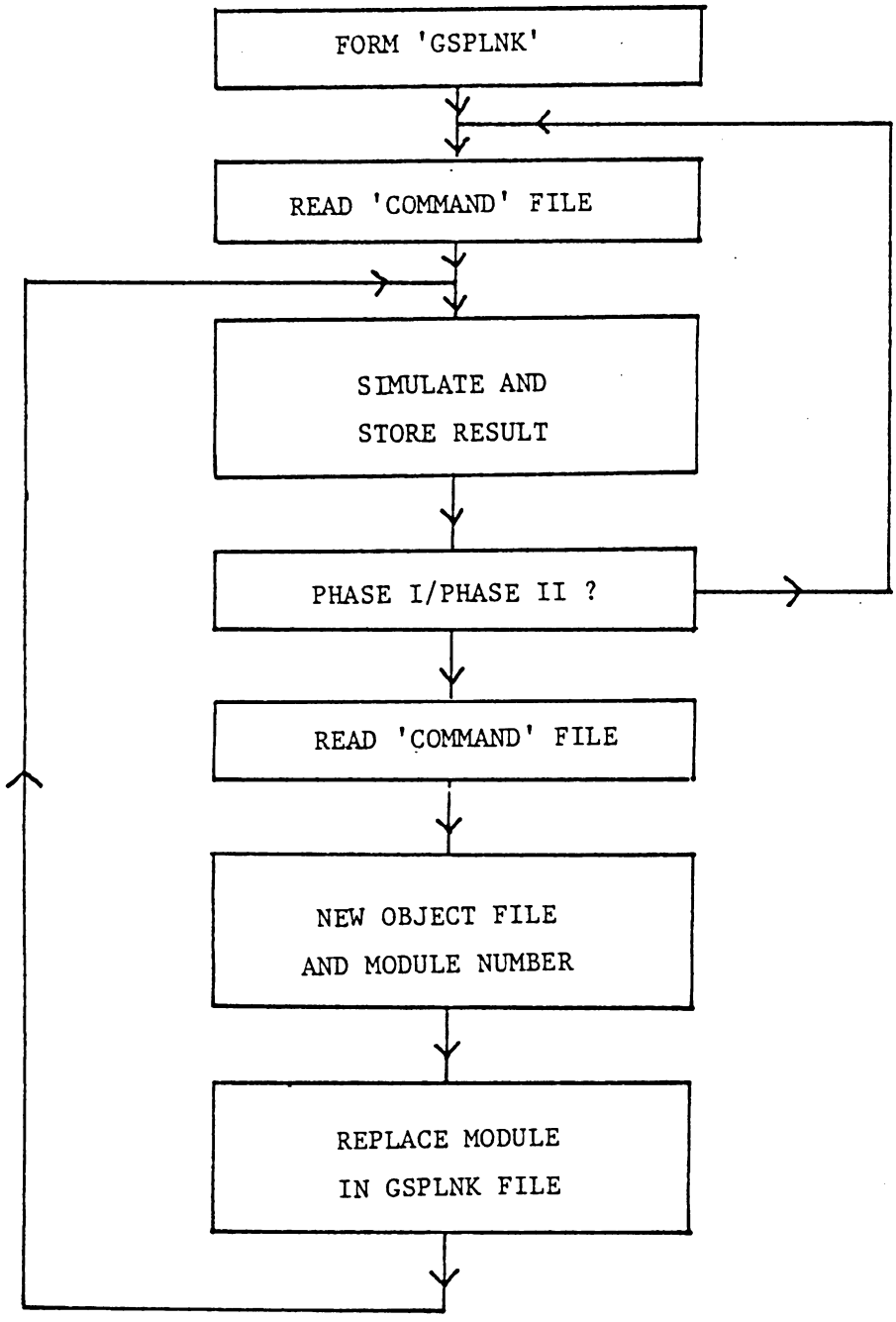

Figure 30: Automatic sequencing of fault simulation runs

Chapter V

RELATION BETWEEN THE FUNTIONAL AND GATE LEVEL SIMULATION

In the chapters III and IV, different methods to inject the functional level faults and simulating the faulty system were discussed. To test a digital circuit completely, it is necessary to apply all the possible combinations of input patterns. For an LSI chip with 20 input pins, the number of possible input patterns is 2^{20} which is very large and it is not feasible to test a chip for all these input patterns. Even after eliminating the illegal input combinations, the number of input patterns remaining is very large. With a device containing sequential logic, the complexity of the problem is furthur increased as the test consists of input patterns in a particular sequence. Apart from this, the large number of primitive elements (gates, flip-flops) and unavailablity of the gate level circuit diagram for the LSI and VLSI devices has forced the designers and the test engineers to go to a higher level of simulation as gate level simulation is clearly inadequate.

Fault analysis of a device using the gate level simulator is carried out by injecting two types of faults i.e. i) line stuck-at faults and ii) shorts between two lines (wired

AND/OR). In case of the SSI or MSI chips, this procedure was found to be effective for the fault analysis. For the more complex chips (LSI & VLSI), the simulation is done at the higher level where the actual circuit is not simulated but the behavior of the device is modeled. The lines in which stuck-at faults could be injected in a gate level model are not present in the functional level model and it is necessary to find new methods to select a set of faults for the LSI devices. Also it has been found that certain types of faults in these complex devices cannot be represented by the conventional stuck-at fault approach, which makes the problem of choosing the set of faults more complex.

An attempt has been made to solve this problem by using a different approach. Instead of selecting the functional level faults which cover most of the stuck-at faults, a set of faults were selected from the functional description of the device given by the manufacturer. Care was taken so that the set covers all the micro-operations of the chip. Using such faulty models, a test was developed which could detect all of the above faults. Then the chip was modeled at the gate level. All possible stuck-at faults were simulated and by applying the above test vector, the fault coverage of the test was determined. This approach gave an idea of the merit of the test generated, using the functional level simula-

tion. As an example, an INTEL 8212 (an 8 bit I/O buffer chip), was selected for analysis.

5.1 THE FUNCTIONAL LEVEL FAULTS

Referring to the manufacturer's specifications the functional description of various input signals was studied. Then the functional faults selected were as given below.

1. Improper generation of clock to D flip-flops i.e. the output of the data latch does not follow data on the input pins properly.
2. Faulty EN signal or the malfunctioning output buffers.
3. Clear (CLR) does not set the service request flip-flop.
4. High to low transition on the strobe signal fails to clock the 'SR' flip-flop.
5. Clear (CLR) does not clear all D flip-flops in the data latch.
6. The device select pins (DS1 and NDS2) do not function properly.
7. Faulty interrupt pin.

These are seven general categories. For the detailed fault analysis of the chip, a number of functional faults from each category could be selected, depending on the complexity

of the functions or the fault data available. In this case, one fault from each category was selected.

As the next step, a set of test vectors was developed to test the functional operation of the 8212 chip. By simulating the INTEL 8212 and applying the test input, the response of the good module was noted. Using the techniques explained in chapter III, the above set of faults were injected in the 8212 module. The faulty modules were simulated and by repeating the process of noting the response of the faulty module and improving the test vector, a final test vector was developed which could detect all of the above faults. In other words, by simulating any of the faulty modules, the response to the test vector deviated from that of the good module at least once, thus indicating the faulty behavior. The test vector generated is given in Table 1. Next, this test vector was used on the gate level model of the INTEL 8212.

TABLE 1

Test vector for an INTEL 8212

Simulation	Control Inputs					Data Input
time(ns)	NDS1	DS2	MD	STB	CLR	(hex)
0	0	0	0	1	1	55
100	0	1	0	0	1	55
200	0	1	0	0	1	F0
300	1	1	0	0	1	F0
400	0	1	0	0	1	AA
500	0	1	1	0	1	AA
600	0	1	1	0	1	28
700	1	1	1	0	1	5A
800	1	1	1	1	1	0F
900	1	1	0	0	0	0F
1000	0	1	0	0	1	0F

5.2 THE GATE LEVEL SIMULATION

LOGSIM is the gate level simulator developed at Virginia Tech. When simulating on the LOGSIM, all the elements are assumed to have unit delay; so a timing for the actual device cannot be simulated. The D flip-flop in the LOGSIM is negative edge triggered. In INTEL 8212, the D flip-flops for the data latch are level triggered; so, 8 level triggered D flip-flops were designed using the combinational logic (NAND gates). Figure 6 shows the gate level circuit diagram for Intel 8212. The faults injected in this model can be roughly divided into three categories.

1. Input or output pins stuck-at faults.
2. Stuck-at faults in the input or output of the gates.
3. Shorts between the two lines.

On LOGSIM, first a good model was simulated by applying the test vector (developed as above using the fuctional level model) the the model response was observed. Then all the line stuck-at faults and shorts between the lines were simulated.

5.3 RESULTS

It was observed that from a total of 90 different faults injected, the test vector detected all but 7 gate level faults. The undetected faults were as listed below.

1. Output buffer #2 always enabled (s-a-1).
2. Output buffer #4 always enabled (s-a-1).
3. Output buffer #6 always enabled (s-a-1).
4. Output buffer #8 always enabled (s-a-1).
5. Input to the NOR gate R1 (line 7 in Figure 31) stuck-at zero.
6. Clear input to flip-flop D6 stuck-at one.
7. Clear input to flip-flop D8 stuck-at one.

Referring to Table 1, it was observed that by changing the data input at simulation time 0 from 55h to 0FFh, the test vector could have detected the first four of the above faults.

Similarly, by proper choice of other data inputs in the test sequence, it may be possible to detect the last two of the undetected gate level faults. Clearly, to achieve this, information about the internal structure of the device is required. The INTEL 8212 being a simple device, the gate level circuit diagram was available. Using the functional level fault analysis, a test vector was found which could detect 83 out of 90 gate level faults injected. Using the

information about the internal structure of the device, the test vector could be further improved to detect 89 out of 90 faults.

In general, for an LSI device, the gate level circuit is not availble and so the gate level fault simulation is not feasible. In that case, once a test vector is developed using the functional level fault analysis, it cannot be further improved to get better fault coverage. So a pessimistic figure for the fault coverage (in this case 83 detected out of 90 gate level faults) should be used. For INTEL 8212, the test vector generated from the functional level analysis could detect 92% of the gate level faults.

Thus it clearly follows that the functional level fault analysis can be effectively used to develop a test vector for an LSI device. Moreover, for a complex LSI device, it is not feasible to conduct the detailed gate level fault analysis. As an example, in case of an INTEL 8080 chip, the gate count is of the order of 10,000 and simulation of the chip for all stuck-at faults is an impossible task. So in case of LSI devices, functional level fault analysis is superior to traditional gate level simulation.

Chapter VI

RESULTS AND CONCLUSION

6.1 SIMULATION EFFICIENCY

When simulating a microprocessor based system consisting of several LSI devices, simulation efficiency can be measured in terms of the processor clock cycles simulated per CPU seconds of host computer. Experiments were conducted, to determine the simulation efficiency. For a processor based system, an assembly language program was loaded into the ROM module. Three sample simulation runs were made. SYSTEM1 consisted of an INTEL 8080 processor and ROM module. SYSTEM2 was made up of 8080, ROM and 8228 (8 bit bi-directional buffer). SYSTEM3 consisted of 8 different modules; these were 8080, ROM, RAM, 8228, 8251 (serial I/O), 8255 (parallel I/O), system data bus and chip select logic module. In each case, the same assembly language program was executed while simulating these systems. For the INTEL 8080 microprocessor, the clock cycle is of 500 ns. Table 2 summarizes the results of these simulation runs. The column 3 has the figures for the number of microprocessor clocks simulated per CPU second of the host computer. It was clearly observed that the simulation efficiency dropped drastically with the increase in the number of modules in the system.

TABLE 2

Simulation Efficiency For Different Systems

SYSTEM NAME	NUMBER OF MODULES	p CLKS / CPU SEC
SYSTEM1	2	99
SYSTEM2	3	67
SYSTEM3	8	14

6.2 FAULT SIMULATION OF INTEL 8080 SYSTEM

Under an R.A.D.C. contract [15], SYSTEM3 was simulated to develop the diagnostic self test software. To validate this software, total 118 faults were injected in various modules of the system. With high reliability of LSI devices, the most likely faults will be interconnect faults, e.g. dry solder joints and printed circuit board defects. Therefore, a high priority was given to interface defects and 43% of the faults injected were interface faults.

Table 3 gives the summary of the results of these fault injection experiments. The simulation results were divided into three categories.

1. Detected (DET): The test routine clearly detected these faults.
2. Loss of Program Control (PCL): The fault simulation resulted in a loss of program control and the test program was not completed. These faults were detected by a watch dog timer.
3. Not Detected (NOT DET): The test program was completed successfully and the test routine could not detect these faults.

The results show that the system was particularly sensitive to faults in the data path involving BUS, ROM and 8228. In case of fault simulation in these modules, the fault cover-

TABLE 3

Fault injection experiment Results

SYSTEM COMPONENTS	DET	PCL	NOT DET	TOTAL
CPU	33	2	1	36
8228	4	-	2	6
BUS	-	3	-	3
ROM	1	3	1	5
RAM	2	2	3	7
8255	31	-	7	38
8251	21	-	2	23
TOTALS	92	10	16	118
PERCENT	78	8	14	100

age was poor as these faults affect the instruction and the data read by the processor; resulting in alteration of the program flow. On the other hand, the coverage of internal CPU faults was excellent. For the detail information about these fault simulation runs, refer to [9].

6.3 RECOMMENDATIONS

At present, when simulating an LSI device on the GSP simulator, the modeling is done in the GSP assembly language. This source code is then assembled using the GSP assembler to form an object code for the module. The GSP simulator then simulates the device using this object code. If the modeling is done in the LSI 11 assembly language, when simulating on the VAX computer, one level of interpretation will be eliminated, thus boosting the simulation efficiency.

When simulation is employed for the fault analysis, a list of generic faults should be selected before modeling the device. Once the faults to be inserted are known, the device can be modeled in such a way that insertion of these faults is simplified. Using this approach, a hypothetical processor Mano machine was modeled. Appendix B has the GSP source code for the model.

At present, the module pins are addressed using the labels. If these pins can also be accessed by actual pin num-

bers, a general purpose fault subroutines could be created. By partitioning the module pins, a group of pins could be reserved for fault injection. Then a library of fault routines for the GSP simulator can be created.

Using GSP, stuck-at faults at the input pins can be injected by modifying the Commnad file. But in case of faults at output pins, the source code of the model has to be modified which is a slow process. By modifying the output processor (module 0), this procedure could be simplified.

6.4 CONCLUSIONS

The functional level simulator can be effectively used for fault analysis of LSI devices. When modeling in GSP assembly language, the size of the model code is reduced considerably as compared to a gate level model. Also, the higher level of representation simplifies the procedure of injecting complex functional faults. But the problem of selecting faults still remains unsolved. As the effect of circuit level faults on the chip behavior is not known, the functional level faults selected may not represent the set of probable faults; in which case, the fault simulation results could be unreliable.

Appendix A

SOURCE CODE FOR INTEL 8212 CHIP

```
      REG(8)  D,TEMP8,DOD
      REG(1)  INTFF,SEL,DCLK,DCLKO,EN,ENO
      REG(1)  CLCHG,STBO,NSSR,TEMP,TEMPA
                              ;
      PIN     DI(1,8),DO(9,16),NDS1(17),DS2(18)
      PIN     MD(19),STB(20),NCLR(21),NINT(22)
      PIN     SCEX(150),DRSC(151),IFSC(152)
      PIN     ENSC(153),DRSCF(154),IFSCF(155)
      PIN     IFSCF(155),ENS(156)
                              ;
      EVW     W40(40),W35(35),W30(30),W10(10)
                              ;
      ;=======================================
                              ;
      MOV NDS1,TEMP           ; SEL=NDS1'.DS2
      COM TEMP
      AND DS2,TEMP,SEL
      MOV MD,TEMP             ; DCLK=MD'.STB+MD.SEL
      COM TEMP
      AND STB,TEMP,DCLK
      AND SEL,MD,TEMP
      OR  TEMP,DCLK
      OR  MD,SEL,EN           ; EN=MD+SEL
      BEQ DRSC,NFLG           ; CHECK DRSCF
      MOV #0,DRSC
      BEQ #1,DRSCF,DATR       ; TRANFER DATA INTO OR ZERO D?
      MOV #0,D                ; D=0
      BRU REGO
DATR:MOV DI,D                 ; D=DI
      MOV #0,DRSCF
REGO:MOV D,DOD                ; SET UP OUTPUT DATA
      JSR OUT
NFLG:BEQ IFSC,NFLG2           ; CHECK IFSCF
      MOV #0,IFSC
      BEQ #1,IFSCF,SFF        ; SET OR RESET?
      MOV #0,INTFF            ; RESET FLIPFLOP
      BRU NFLG2
SFF:  MOV #1,INTFF            ; SET FF
      MOV #0,IFSCF
NFLG2:BEQ ENSC,NEX1           ; CHECK ENSC
      MOV #0,ENSC
      JSR OUT
NEX1:XOR EN,ENO,TEMP          ; CHECK ENABLE
```

```
      BEQ TEMP,NEX3
      MOV(W35) EN,ENS
      MOV(W35) #1,ENSC          ; SCHEDULE SELF CALL
NEX3:MOV DCLK,TEMP              ; FORM DCLKO.DCLK'
      COM TEMP
      AND DCLKO,TEMP,CLCHG
      BEQ CLCHG,NEX4            ; EQUAL 0?
      MOV(W30) #1,DRSCF         ; SET UP SELFCALL
      MOV(W30) #1,DRSC
NEX4:AND DCLK,EN,TEMP           ;FORM EN.DCLK
      BEQ TEMP,NEX5
      MOV DI, D                 ; D = DI
      MOV(W40) D, DO            ; DO = D IN 40NS
NEX5:BEQ #1,CLCHG,NEX6          ; RESET D ?
      BEQ #1,NCLR,NEX6
      MOV(W30) #0,DRSCF
      MOV(W30) #1,DRSC
NEX6:MOV STB,TEMP               ; FORM STB'.STBO
      COM TEMP
      AND STBO,TEMP
      BEQ TEMP,NEX7
      MOV(W30) #0,IFSCF         ; RESET INTFF
      MOV(W30) #1,IFSC
      BRU NEX8
NEX7:MOV NCLR,TEMP
      COM TEMP
      OR SEL,TEMP,NSSR
      COM NSSR                  ;
      BNE NSSR, NEX8
      MOV(W30) #1,IFSCF         ; SET INTFF
      MOV(W30) #1,IFSC
NEX8:MOV INTFF,TEMP             ; COMPUTE NINT
      COM TEMP
      OR TEMP,SEL,TEMP
      COM TEMP
      XOR TEMP,NINT,TEMPA       ; CHANGE IN NINT ?
      BEQ TEMPA,UPDATE
      MOV(W10) TEMP,NINT        ; INT'=0 IN 10 NS
UPDATE:MOV EN,ENO
      MOV STB,STBO
      MOV DCLK,DCLKO
      MOV #0,SCEX               ; EXIT
OUT:  BEQ ENS,HIZ               ; EN = 0 ?
      MOV(W10) DOD,DO           ; DO = DOD IN 10NS
      RTS
HIZ:  MOV(W10) #255,DO          ; DO =ALL 1'S IN 10NS
      RTS
      END
```

Appendix B

SOURCE CODE FOR MANO MACHINE

```
          REG(8)     MBRH,MBRL,ACH,ACL,PCH,PCL
          REG(8)     MARH,MARL,TEM8H,TEM8L
          REG(3)     OPR
          REG(2)     FR,TEMP2,TIMER
          REG(1)     E,I,S,IEN,TEMP1,CARRY
;
          PIN     ADDRL(1,8),ADDRH(9,16),DOUTL(17,24)
          PIN     DOUTL(17,24),DOUTH(25,32),DINL(33,40)
          PIN     DINH(41,48),RW(49),IEN(50),FGI(51)
          PIN     FGO(52),INP(53,60),OUT(61,68)
          PIN     EXIT(150),CLKSC(151)
;
          EVW        T(500),W0(0)
;
100:      IDX        FR(0),2,1       ;
          JSR        CLOCK           ;
          BRU        TBL1@1          ; WHICH CYCLE ?
TBL1:     BYT   1000,2000,3000,4000
;
;
;         FETCH CYCLE
;         ===========
1000:     JSR        SUB9            ; RW=1
          JSR        SUB8            ;
          JSR        CLOCK           ;
          JSR        SUB2            ;
          JSR        SUB4            ;
          JSR        CLOCK           ;
          IDX        MBRH(4),3,1     ;
          IDX        MBRH(7),1,2     ;
          MOV        @1,OPR
          MOV        @2, I           ;
          JSR        CLOCK           ;
;
;    C0.T3
          BEQ        #7, OPR, 1010   ; IF OPR=7,EXECUTE CYCLE
          BEQ        I, 1010         ; ELSE, IF (I) = 0, SAME.
          OR         #1,FR           ; SET R. INDIRECT CYCLE.
          BRU        100             ; NEXT CYCLE.
;
1010:     OR         #2, FR          ; SET F. EXECUTION CYCLE.
          BRU        100
;
```

```
;    ------------------------------------------------------
;                    INDIRECT  CYCLE
;                    ========  =====
;
2000:     JSR        SUB9              ; READ OPERATION
          JSR        SUB1
          JSR        CLOCK             ;
          JSR        SUB2              ; READ DATA
          JSR        CLOCK
          JSR        CLOCK
          MOV        #2, FR            ; NEXT IS EXECUTION CYCLE
          BRU        100               ; GO TO NEXT CYCLE.
;
;    ------------------------------------------------------
;                    INTERRUPT  CYCLE
;                    =========  =====
4000:     JSR        SUB7              ; RETURN ADDRESS
          MOV        #0, PCH           ; POINT TO LOCATION 0.
          MOV        #0, PCL           ; POINT TO LOCATION 0.
          JSR        CLOCK
          JSR        SUB8              ; 0 => MAR
          JSR        SUB10             ; 0 => RW
          JSR        SUB4
          JSR        CLOCK             ;
          JSR        SUB3              ; WRITE DATA
          MOV        #0, IEN           ; CLEAR INT ENABLE
          JSR        CLOCK             ;
          MOV        #0, FR            ; CLEAR 'F' & 'R'  F/F.
          BRU        100               ;
;
;    ------------------------------------------------------
;         SUBROUTINE       CLOCK
;         ==========       =====
CLOCK:    MOV(T)     #1, CLKSC         ; SET SELF CALL FLAG
          MOV        #1, EXIT          ; WAIT FOR CLOCK PULSE.
          MOV        #0, CLKSC         ;
          INC        TIMER             ; T0..T3
          BNE        TIMER,NOHLT       ; HALT ONLY IF T0
          BNE        S,NOHLT
          MON                          ; HALT.
NOHLT:    RTS
;
;         SUBROUTINE      SUB1
;         ==========      ====
;         MAR  =  MBR(AD)
SUB1:     MOV        MBRL,MARL         ; MOV LOWER BYTE
          AND   #15,MBRH, MARH         ; MBRL => MARL
          MOV(W0)    MARH, ADDRH       ; SET ADDRESS
          MOV(W0)    MARL, ADDRL       ; SET ADDRESS
```

```
          RTS
;
;         SUBROTINE       SUB2
;         ==========      ====
;         M  =>  MBR
SUB2:     MOV       DINL, MBRL      ; READ OPERATION
          MOV       DINH, MBRH      ; READ OPERATION
          RTS
;
;         SUBROUTINE      SUB3
;         ==========      ====
;         MBR  =>  M
SUB3:     MOV(WO)   MBRL, DOUTL     ; WRITE OPERATION
          MOV(WO)   MBRH, DOUTH     ; WRITE OPERATION
          RTS
;
;         SUBROUTINE      SUB4
;         ==========      ====
;         PC  =  PC  +  1
SUB4:     INC       PCL             ;
          BEQ       C, SKIP
          INC       PCH
SKIP:     RTS
;
;         SUBROUTINE      SUB5
;         ==========      ====
;         AC  =  AC  +  MBR
SUB5:     ADD       MBRL, ACL       ;
          BEQ       C, SKIP1        ;
          ADD       #1, ACH
SKIP1:    ADD       MBRH, ACH
          MOV       C,CARRY         ; STORE CARRY.
          RTS
;
;         SUBROUTINE      SUB6
;         ==========      ====
;         MBR  <=  AC
SUB6:     MOV       ACL, MBRL       ;
          MOV       ACH, MBRH
          RTS
;
;         SUBROUTINE  SUB7
;         ==========  ====
;         MBR  <=  PC
SUB7:     MOV       PCH,MBRH        ;
          MOV       PCL,MBRL
          RTS
;
;         SUBROUTINE  SUB8
```

```
;         ==========  ====
;         MAR  <=  PC
SUB8:     MOV        PCL,MARL        ;
          MOV        PCH,MARH        ;
          RTS
;
;         SUBROUTINE  SUB9
;         ==========  ====
;         SET   RW = 1
SUB9:     MOV(W0)    #1, RW
          RTS
;
;         SUBROUTINE  SUB10
;         ==========  =====
SUB10:    MOV(W0)    #0, RW          ;
          RTS
;
;    ------------------------------------------------
;
;         EXECUTION  CYCLE
;         =========  =====
3000:     BNE        #7, OPR, 3001   ; MEMORY REFERENCE.
          BEQ        I, 3800         ;I=0 => REG REFERENCE.
          IDX        OPR(0),3,2      ;
          BRU        TBL2@2          ; I/O INSTRUCTIONS
;
TBL2:     BYT  3840,3850,3860,3870,3880,3890,HALT,HALT
;
3001:     IDX        OPR(0),3,1      ;
          BRU        TBL3@1          ; BRANCH TO MEMORY REF
;
TBL3:     BYT  3100,3200,3300,3400,3500,3600,3700,3000
;
;
;         MEMORY REFERENCE INSTRUCTIONS
;         ====== ========= ============
;         AND  TO AC
3100:     JSR        SUB9            ; READ SIGNAL
          JSR        SUB1            ;
          JSR        CLOCK           ;
          JSR        SUB2            ;
          JSR        CLOCK           ;
          AND        MBRH, ACH       ;
          AND        MBRL, ACL       ;
          JSR        CLOCK
          BRU        5000            ; CHECK INTERRUPT
;
;         ADD TO AC
3200:     JSR        SUB9            ; READ SIGNAL
```

```
          JSR       SUB1            ;
          JSR       CLOCK
          JSR       SUB2            ;
          JSR       CLOCK
          JSR       SUB5            ;
          MOV       CARRY, E        ;
          BRU       5000            ;
;
;         LDA TO AC
3300:     JSR       SUB9            ; READ
          JSR       SUB1            ;
          JSR       CLOCK
          JSR       SUB2            ; READ DATA
          MOV       #0, ACH
          MOV       #0, ACL
          JSR       CLOCK
          JSR       SUB5            ; ADD
          BRU       5000            ;
;
;         STORE  AC
3400:     JSR       SUB1            ; WRITE SIGNAL
          JSR       SUB10           ;
          JSR       CLOCK
          JSR       SUB6            ; WRITE DATA
          JSR       CLOCK
          JSR       SUB3            ;
          BRU       5000            ;
;
3500:     JSR       SUB7            ; BUN
          JSR       CLOCK
          JSR       CLOCK
          BRU       5000
;
;         BRANCH AND SAVE RETURN ADDRESS
3600:     JSR       SUB1            ;
          JSR       SUB10
          MOV       PCH, TEM8H      ;
          MOV       PCL, TEM8L      ;
          MOV       MBRH,PCH        ;
          MOV       MBRL,PCL        ;
          MOV       TEM8H, MBRH     ;
          MOV       TEM8L, MBRL     ;
          JSR       CLOCK           ;
          JSR       SUB3            ;
          JSR       CLOCK
          JSR       SUB4
          BRU       5000            ;
;
;         INCREMENT AND SKIP IF ZERO.
```

```
3700:   JSR     SUB9            ; READ SIGNAL
        JSR     SUB1
        JSR     CLOCK
        JSR     SUB2
        JSR     CLOCK
        INC     MBRL            ;
        BEQ     Z, 3701         ;
        INC     MBRH
3701:   JSR     CLOCK
        JSR     SUB3
        BNE     MBRH, 3702
        BNE     MBRL, 3702
        JSR     SUB4
3702:   MOV     #0, FR
        BRU     100
;
;       REGISTER REFERENCE INSTRUCTIONS.
;       ======== ========= =============
3800:   IDX     MBRH(0), 4, 1   ; LOWER 4 OF MBRH.
        JSR     CLOCK           ;
        JSR     CLOCK           ;
        JSR     CLOCK           ;
                                ; Decode instruction
        BEQ     @1, 3801        ;
        BEQ     #8, @1, CLA     ;
        BEQ     #4, @1, CLE     ;
        BEQ     #2, @1, CMA     ;
        BEQ     #1, @1, CME     ;
3801:   BEQ   #128, MBRL, CIR   ;
        BEQ    #64, MBRL, CIL   ;
        BEQ    #32, MBRL, INCC  ;
        BEQ    #16, MBRL, SPA   ;
        BEQ    #8,  MBRL, SNA   ;
        BEQ    #4,  MBRL, SZA   ;
        BEQ    #2,  MBRL, SZE   ;
        BRU     HALT            ;
;
;
;       CLEAR ACC
CLA:    MOV     #0, ACL         ; CLEAR
        MOV     #0, ACH         ; ACCUMULATOR
        BRU     5000            ;
;       CLEAR  E
CLE:    MOV     #0, E           ;
        BRU     5000
;       COMPLEMENT    ACC
CMA:    COM     ACL             ;
        COM     ACH             ;
        BRU     5000            ;
```

```
;
;       COM  E
CME:    COM     E               ;
        BRU     5000
;
;       CIR     ACE
CIR:    MOV     E, @7           ;
        SHR     ACH             ;
        MOV     C, @7           ;
        SHR     ACL             ;
        MOV     C, E            ;
        BRU     5000
;
;       CIL     E-ACC
CIL:    ADD     ACL, ACL        ; SHIFT LEFT
        MOV     C, CARRY        ;
        BEQ     E, 3803         ;
        INC     ACL             ; E => LSB OF ACL
3803:   ADD     ACH, ACH        ;
        MOV     C, TEMP1        ; STORE CARRY
        BEQ     CARRY, 3805
        INC     ACH             ; MSB (ACL)=> LSB (ACH)
3805:   MOV     TEMP1, E        ; MSB OF ACH => E
        BRU     5000
;
INCC:   INC     ACL             ; INCREMENT ACC
        BEQ     C, 3808         ;
        INC     ACH             ;
        BEQ     C, 3808         ;
        MOV     #1, E           ;
3808:   BRU     5000
;
SPA:    IDX     ACH(7),1,2
        BEQ     @2, 3810        ; INC PC IF MSB (AC) ne 0
        JSR     SUB4            ;
3810:   BRU     5000
;
SNA:    IDX     ACH(7),1,2
        BNE     @2, 3811        ; INC PC IF MSB (AC)=0
        JSR     SUB4            ;
3811:   BRU     5000
;
SZA:    BNE     ACL, 3812
        BNE     ACH, 3812       ; INC PC IF (AC)=0
        JSR     SUB4            ;
3812:   BRU     5000
;
SZE:    BEQ     E, 3813         ; INC PC IF 'E' = 0
        JSR     SUB4            ;
```

```
3813:     BRU       5000
;
HALT:     MOV       #0, S           ; F/F 'S' = 0
          BRU       5000
;
;         ============================================
3840:     MOV(W0)   INP, ACL        ; INPUT DATA
          MOV(W0)   #0, FGI         ;
          BRU       5000
;
3850:     MOV(W0)   ACL, OUT        ; OUTPUT DATA
          MOV(W0)   #0, FGO         ; RESET FGO FLAG
          BRU       5000
;
3860:     MOV       FGI, TEMP1
          BEQ       TEMP1, 3862
          JSR       SUB4            ; INC PC
3862:     BRU       5000
;
;         SKO
3870:     MOV       FGO, TEMP1
          BNE       TEMP1, 3872
          JSR       SUB4            ; INC PC
3872:     BRU       5000
;
;         ION
3880:     MOV(W0)   #1, IEN         ;
          BRU       5000
3890:     MOV(W0)   #0, IEN         ; RESET IEN
          BRU       5000
;
;         ============================================
5000:     MOV       IEN, TEMP1
          BEQ       TEMP1, 5500
          MOV       FGI, TEMP1
          BNE       TEMP1, 5300
          MOV       FGO, TEMP1
          BEQ       TEMP1, 5500
5300:     OR        #1, FR          ; SET 'R'  F/F
          BRU       100             ; NEXT CYCLE
;
5500:     AND       #2, FR          ; SET 'F'  F/F
          BRU       100
;
;
;
          END
```

BIBLIOGRAPHY

[1] Breuer, Malvin A. and Friedman Arthur D., Diagnostic & Reliable Design of Digital Systems. Computer Science Press, INC. 1976.

[2] Levendel, Y. H. and Schwartz, W. C., Impact of LSI on Logic Simulation. COMPCON SPRING 1978, pp 102-105.

[3] Marvin, D. Ellis., A General Purpose Digital Network Simulator. Thesis, VPI & SU, 1979

[4] Joobbani, Rostan., Functional Level Fault Simulation. International Conference on Circuits and Computers, Oct 1980.

[5] Tokoro, Mario et. al., A Module Level Simulation Technique for Systems Composed of LSI's and MSI's.

[6] Abramovici, M., Breuer, M. A. and Kumar K., Concurrent Fault Simulation and Functional Level Modeling. 14th Design Automation Conference Proceedings, 1977, pp 128-135.

[7] Wadsack, R. L., Technology Dependent Logic Faults. IEEE 1978, COMCON, pp 124-129.

[8] Galiay, J., Crouzet, Y. and Vergniault, M., Physical versus Logical Fault Models in MOS LSI circuits, Impact on their Testability. IEEE 1979, Fault Tolerant Computing, pp 195-202.

[9] Gray, F. G. & Armstrong, J.R., Microprocessor Self-test: Final Technical Report. Blacksburg VA: Dept of Electrical Eng., VPI & SU, 1981.

[10] Devlin, Donald E. A Chip Level Multimodule Logic Simulator. Masters Thesis, VPI & SU, 1980.

[11] INTEL Corp., MCS-80 User's Manual. Santa Clara, CA: INTEL, 1977.

[12] Puthenpurayil, V. and Armstrong, J. R., Functional Level Modeling of LSI devices 1981 SSST, pp 290-293.

[13] Siewiorek, D. P., Bell C. G. and Newell, A., Computer Structures: Principles and Examples. McGraw Hill, 1982.

[14] McConnel, Stephen R., Siewiorek, Daniel P. and Tsao, Michael, M., The measurment and analysis of transient errors in digital computer systems. IEEE 1979, Fault Tolerant Computing, pp 67-70.

[15] Mano, Morris M., Computer System Architecture. Prentice-Hall, INC, N.J. 1976.

[16] Woddruff, Gary. Simulation Techniques For Microprocessors Master's Thesis, VPI & SU, August 1976.

[17] Lewin, Douglas, Computer Aided Design Of Digital Systems. Crane, Russak & Company, New York, 1977.

[18] Smith, Frank M., A Simulation Model For Functional Fault Injection written in ISP. International Conference on Circuits and Computers, 1980.

[19] Northcutt, Duane J., The Design and Implimentation of Fault Insertion Capabilities for ISPS. ACM 1980.

[20] Thompson, Edward W., Simulation - A tool in an Integrated Testing Environment. IEEE 1980 Test Conference, pp 7-12.

[21] Sherwood, Will. Simulation Hierarchy For Microprocessor Design. Design Automation And Microprocessors, 1977, pp 44-49.

The vita has been removed from the scanned document

FUNCTIONAL LEVEL FAULT SIMULATION OF LSI DEVICES

BY

SHIRISH K. SATHE

(ABSTRACT)

Procedures for the modeling and simulation of faults in LSI devices at functional level are developed. Generalized functional level fault classes are defined for digital LSI devices such as microprocessor and peripheral chips. General procedures to inject functional level faults in the LSI chip models are illustrated with the help of various examples. Next, techniques of automating the simulation of the faulty systems are discussed. Finally, simulation of faults at the functional level is compared with the gate level simulation in case of INTEL 8212 (8 bit I/O) chip.