# FUNCTIONAL FAULT MODELING AND TEST VECTOR DEVELOPMENT FOR VLSI SYSTEMS

by

Anil K. Gupta

THESIS submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

APPROVED:

DR. JAMES R. ARMSTRONG, Chairman

DR. CHARLES E. NUNNALLY

DR. JOSEPH G. TRONT

March, 1985
Blacksburg, Virginia

FUNCTIONAL FAULT MODELING AND TEST VECTOR GENERATION FOR VLSI SYSTEMS

by

Anil K. Gupta

Committee Chairman : Dr. James R. Armstrong

Electrical Engineering

(ABSTRACT)

The attempts at classification of functional faults in VLSI chips have not been very successful in the past. The problem is blown out of proportions because methods used for testing have not evolved at the same pace as the technology. The fault-models proposed for LSI systems are no longer capable of testing VLSI devices efficiently. Thus the stuck-at and short/open fault models are outdated. Despite this fact, these old models are used in the industry with some modifications. Also, these gate-level fault models are very time-consuming and costly to run on the mainframe computers.

In this thesis, a new method is developed for fault modeling at the functional level. This new method called 'Model Perturbation' is shown to be very simple and viable for automation. Some general sets of rules are established for fault selection and insertion. Based on the functional

fault model introduced, a method of test vector development is formulated. Finally, the results obtained from functional fault simulation are related to gate level coverage.

The validity and simplicity of using these models for combinational and sequential VLSI circuits is discussed. As an example, the modeling of IBM's AMAC chip, the work on which was done under contract YD 190121, is described.

ACKNOWLEDGEMENT

My deepest gratitude goes to Dr. J. R. Armstrong, a guide and a friend. It has been an honor to study under him. His help and guidance will always be treasured.

Thanks are due to Drs. C. E. Nunnally and J. G. Tront, my committee members. They have invested much time and support directly and indirectly.

Many others have supported me personally and financially and I am grateful to the gentlemen at IBM, Manassas; to Mrs. McWhorter; to Ms. Pat Wojciechouski, Ms. Leslie Cobb and Ms. Lorri Newman, the secretaries in the EE department. I appreciate their patience and helpfulness. My sincere regards also to my teachers, who inspired me and friends, who stood with me through pleasure and pain.

I dedicate my work to my Parents without whose love and care I would be nothing.

# TABLE OF CONTENTS

FUNCTIONAL FAULT MODELING AND TEST VECTOR GENERATION FOR VLSI SYSTEMS . . . . . . . . . . . . . . . . . ii

ACKNOWLEDGEMENT . . . . . . . . . . . . . . . . . . . . . . iv

Chapter

page

I. INTRODUCTION . . . . . . . . . . . . . . . . . . . . . 1

- Need for Testing . . . . . . . . . . . . . . . . . . 1
- Method of Testing . . . . . . . . . . . . . . . . . 3
- The Fault-Model . . . . . . . . . . . . . . . . . . 6
- Objectives of this Thesis . . . . . . . . . . . . 7

II. LITERATURE SURVEY . . . . . . . . . . . . . . . . . . . 11

- The Thatte-Abraham Technique . . . . . . . . . . . 12
- The Robach-Saucier Technique . . . . . . . . . . . 13
- The Clary-Joobbani Technique . . . . . . . . . . . 15
- Other Techniques . . . . . . . . . . . . . . . . . 18

III. FUNCTIONAL LEVEL MODELING . . . . . . . . . . . . . . . 19

- GSP : The Simulation Language . . . . . . . . . . 19
- Methods of functional modeling . . . . . . . . . . 21
  - Look-up Table Model . . . . . . . . . . . . . . 22
  - Micro-operation Model . . . . . . . . . . . . . 24
- Functional Modeling of AMAC Chip . . . . . . . . . 28
  - The Adder . . . . . . . . . . . . . . . . . . . 29
  - The Multiplier . . . . . . . . . . . . . . . . 32
  - The Register with parity generator . . . . . . 34

IV. FAULT MODELING AT THE FUNCTIONAL LEVEL . . . . . . . 36

- The Model Perturbation Method . . . . . . . . . . 36
  - Truth-Table Modification (TTM) . . . . . . . . 39
  - Micro-operation Modification (MOM) . . . . . . 42

V. AUTOMATIC FAULT-INJECTION AND TEST-GENERATION . . . 49

- Techniques for Fault-injection and Test-generation . . . . . . . . . . . . . . . . . . 49
  - Automatic fault-injection . . . . . . . . . . . 50
  - Automatic Test Generation . . . . . . . . . . . 51

VI. CONCLUSIONS . . . . . . . . . . . . . . . . . . . . . 53

BIBLIOGRAPHY . . . . . . . . . . . . . . . . . . . . . . 55

Appendix

page

A. FLOW-CHARTS FOR THE AMAC MODELS . . . . . . . . . . . 58

B. GSP MODEL DESCRIPTIONS FOR THE AMAC UNIT . . . . . 62

C. GSP ON P.C. . . . . . . . . . . . . . . . . . . . . 96

Need for the Simulator on PC . . . . . . . . . . . 96
GSP Simulator on the PC . . . . . . . . . . . . . 97
Performance . . . . . . . . . . . . . . . . . . . 99
Conclusions . . . . . . . . . . . . . . . . . . . 101

D. PROBLEMS FACED IN TRANSFERRING GSP TO P.C. . . . 102

Problems Faced and Methods used in Transfering GSP from CMS to IBM PC. . . . . . . . . . 102

# LIST OF TABLES

Table page

1. TI 74181 ALU Functions used in AMAC chip . . . . . . . 33
2. GSP Simulator Package . . . . . . . . . . . . . . . . . 98
3. Performance Data . . . . . . . . . . . . . . . . . . . 100
4. Representation of Characters as Integers, Internally. . . . . . . . . . . . . . . . . . . 106

## LIST OF FIGURES

Figure

page


1. Levels of Representation of Digital Circuits . . . . 5

2. Block Diagram of the AMAC unit used for Simulation . 9

3. GSP Simulation Structure . . . . . . . . . . . . . . . 20

4. Actual Function vs. Look-up Table Model . . . . . . . 23

5. "Look-up Table" model . . . . . . . . . . . . . . . . 25

6. Graphical representation of Micro-operation model . 26

7. "Micro Operation" model . . . . . . . . . . . . . . . 27

8. Functional Units of the AMAC model . . . . . . . . . . 30

9. Functional Fault Modeling Methodology for the AMAC chip . . . . . . . . . . . . . . . . . . . . . . . . 31

10. Fault Mapping from Different Levels . . . . . . . . . 37

11. Truth-Table Modification Technique . . . . . . . . . . 41

12. AND function failing to an OR function . . . . . . . . 46

13. NOR function failing to a NAND function . . . . . . . 47

# Chapter I

# INTRODUCTION

The progress in fabrication technology of digital devices has brought us into an era of VLSI circuits. Circuit densities have increased manifold. Speeds have also increased tremendously because of faster devices and close proximity on the layout. The increased density results in increased number of faults per unit area, creating problems in testing.

## 1.1 NEED FOR TESTING

Testing is the problem of determining whether a product has been manufactured correctly [33,34]. In general, many problems are faced during the design, production, and operation of a digital IC. "Testing constitutes detection of these problems and location of their sources"[28]. As a result of testing, the yield of a process is enhanced. In the manufacture of digital IC's, Yield (Y) is the portion of chips which is believed to finally operate correctly [29]. Yield is related to two other parameters as shown in eq.(1) [3,34].

$$DL = 1 - Y^{(1-T)} \qquad (1)$$

$$0 < Y,T,DL < 1$$

where, Testability (T) is the extent to which the circuit can be tested for faults by the test (also called 'fault coverage').

And, Defect Level (DL) is the probability of the existence of defects in the tested product. In other words, it gives a measure of confidence to the predicted yield.

Evidently, for DL to be low, T has to be very high. As an example, consider a board containing 20 modules, each with a DL of 2%. The probability of obtaining a good board, assuming perfect assembly, is :

$$Y=(1-0.02)^{20} = 0.6676$$

Hence, 33.24% of the boards will have one or more faults. From the equation, in order to have a DL of 2% at this level of packaging, the required value for T is 95%. Thus, to be 98% confident that the yield is 66.76%, the required amount of testability is 95%.

This shows that a great degree of time and effort needs to be spent on testing, to obtain a product with a minimum number of defects. For the same reasons of reliable operation, the users of these digital circuits need to test them in the field.

## 1.2 METHOD OF TESTING

The problem of testing involves [33,34] :

1. Test generation, and
2. Fault simulation.

According to the IEEE Standard Dictionary [19], a Fault is the condition in a network that will yield incorrect results for a given input pattern. A defect on the other hand, may or may not cause a fault. A Test for a fault is that input sequence, in response to which, the output of the correctly operating network is different from that of the faulty network. Thus, "for a test to exist : (1) inputs to the network must be controllable, and (2) proper conditions must exist for the fault propagation of test value to the output" [14].

Test generation is done automatically through the use of special test generation algorithms run on computers. As complexity of circuits increases, the storage requirements increase tremendously and the algorithms to generate automatic tests become less efficient.

Fault simulation involves applying a set of test patterns to the fault-free machine and to each of the copies of the machine containing a single fault. It renders a "..measure of worth for test patterns applied to the product..." in terms of fault coverage [34]. As stated,

simulations are performed one fault at a time since the test generation for multiple faults leads to impractical computations because of the dramatic increase in the number of fault machines. Increasing size and complexity plague the fault simulation with respect to computer run time (RT) which depends on the gate count (n) as :

$$RT = K\, n^2 \qquad (2)$$

Digital logic simulation has been used traditionally for obtaining test patterns and fault signatures for existing systems and more recently, for design verification and system validation. An important aspect of simulator design is the efficiency, which is defined as the ratio of the host CPU time to the real logic time. Efficiency depends on the level of simulation [5,8] and generally decreases with the increase in the complexity of representation [fig.1].

Gate level Simulation techniques used to obtain test vectors for the less populated LSI chips have become very cumbersome and slow for VLSI circuits. Such simulations have been seen to run for days on dedicated mainframes [5].

Functional or chip level digital logic simulation is a more viable and useful approach in the context of these new developments. Functional level simulations are performed on those 'descriptions' of the digital logic which are a level

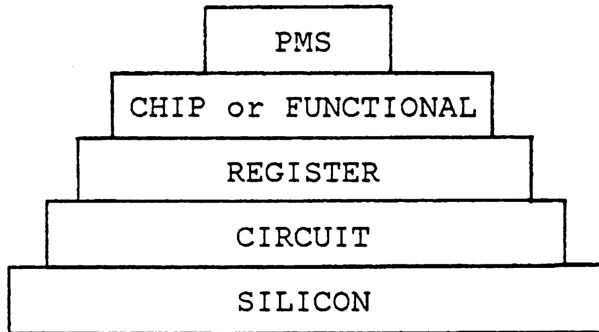



Figure 1: Levels of Representation of Digital Circuits

or two higher than the gate level 'descriptions'. This makes them much faster and less complicated than the gate level simulators. And yet, sufficient timing information and signal-flow detail can be incorporated into these descriptions.

## 1.3 THE FAULT-MODEL

In order to make the simulation of faults in IC's meaningful, an appropriate fault model is necessary. A Fault Model is the mapping from physical defects to the simulated faults. It describes how a circuit correctly functions, in terms of the absence of the fault, to permit the testing of other parts of the circuit and shows how a circuit could fail in terms of the fault. More importantly, it aims at imposing a limit on the number of test vectors required while at the same time keeping the coverage high.

At the gate level, a widely used fault model is the stuck-at fault model. Although many of the physical defects can be modeled as stuck-at faults the mapping from physical defects to stuck-at faults is not complete [14]. The problem has become worse due to the evolution of new fabrication technologies e.g. CMOS, HNMOS, and the shift from LSI to VLSI systems. However, because of the lack of a proven better method and because of the fact that the single

stuck-at fault model has allowed generation of effective chip tests and resulted in a low defect level, its popularity remains.

To date, a generalized and effective functional fault model like the gate level stuck-at model has eluded the researchers. The difficulty lies in the higher level of representaion of the functional level descriptions.

## 1.4 OBJECTIVES OF THIS THESIS

This thesis deals with the details of the functional fault modeling work done on some typical digital circuits to establish the generality and the effectiveness of the functional fault model [FM] proposed here [5,6,8]. Comparison of the gate and functional coverage levels allowed inference of rules for functional fault modeling, and in some cases the experience gained required further iterations of the process. The problem of automating test generation and fault simulation has not been tackled in this work. This implies that testing was done 'manually' for the unit-under-test (UUT).

For the purpose of modeling, a section [fig.2] of IBM's AMAC [Add Multiply and ACcumulate] chip was used. It consists of a 9-bit and two 17-bit LSSD Registers, three Parity checkers, a 16-bit (4-bit slice) Adder (TI 74181) and

an 8x8 modified Dadda-type hardware Multiplier [11,32]. The chip is designed based on the LSSD principle, wherein the sequential logic is in the form of combinational circuits interspersed with sets of Shift Register Latches (SRL) connected serially [15]. The gate count for the section of logic modeled was 3700 gates.

This chapter introduced the topic of digital IC testing, specifying the need for it, and explaining the inadequacy of the currently popular testing methods in the face of latest advancements in technology. The utility and potential of functional level fault modeling and simulation for the field of VLSI circuits have been emphasized.

Chapter II is a literature survey of the work done in functional fault simulation. Obviously, there is an incredible amount of literature on this topic of current interest. But only the work pertaining to the discussion in this thesis has been mentioned.

In chapter III the development of the various functional modules is described with the aid of flow-charts. The reasons for the choice of particular modeling methodologies for specific functions are given.

In chapter IV the two fault models utilized are discussed in detail.

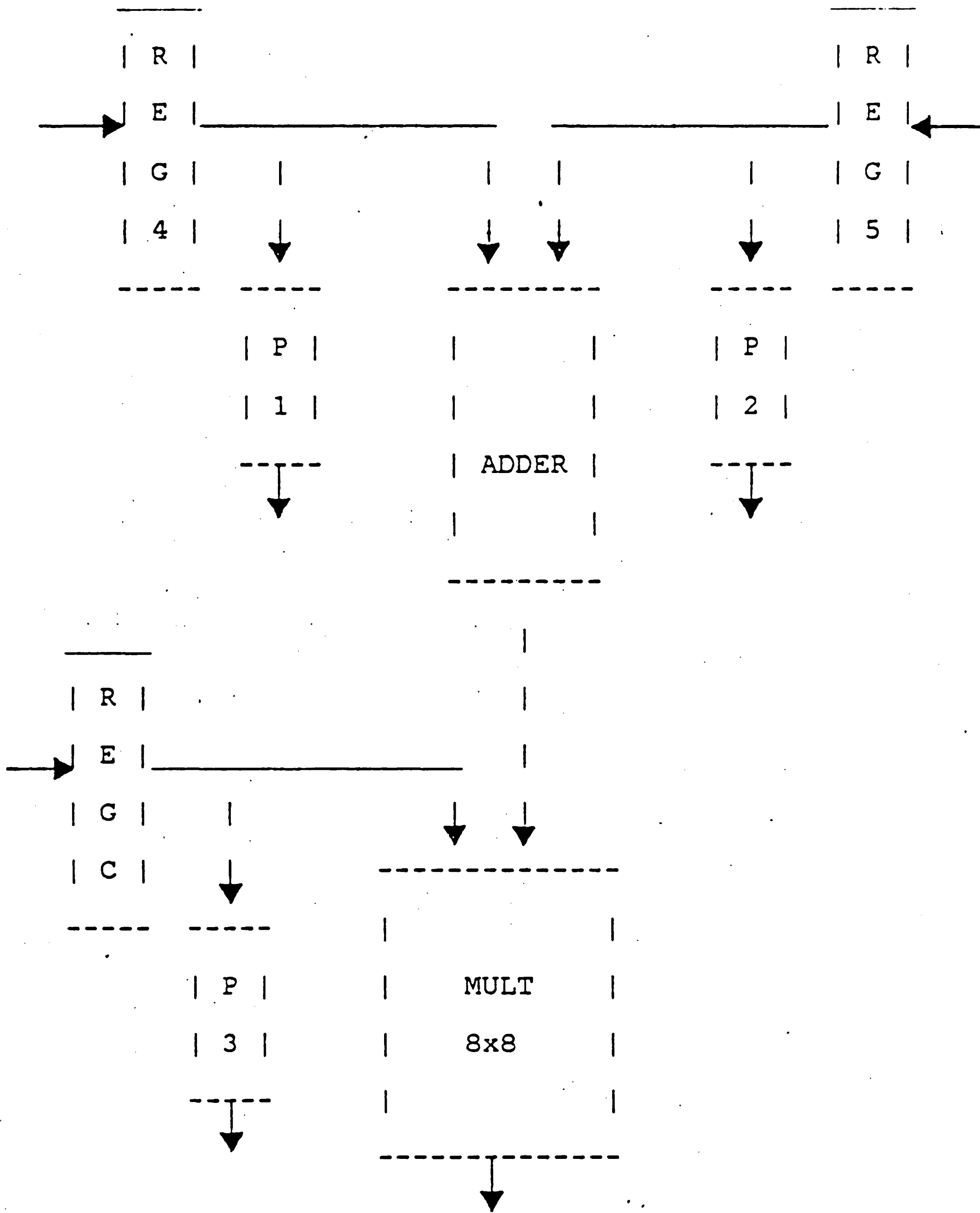



Figure 2: Block Diagram of the AMAC unit used for Simulation

Chapter V deals with the techniques of fault injection and proposes further work to be done in this area to automate the whole process of fault injection and test generation.

Chapter VI is the conclusion section. The results of the fault modeling work done on the AMAC chip are discussed. The relevance of the fault model proposed in this thesis is studied, based on the results achieved.

Appendix A consists of flow-charts and appendix B consists of the GSP model listings for the UUT. Appendices C and D describe the transfer of the modeling language, GSP, to the IBM PC and the Z-100 PC.

## Chapter II

## LITERATURE SURVEY

One of the most important problems concerning the semiconductor industry today is the problem of VLSI testing. Researchers are devoting their time and effort in the search of an economically viable automated method which would be independent of the technology used for fabrication. Consequently, many different techniques have been proposed for fault modeling and simulation at higher levels of representation than the gate level. In order to find quick solutions for the growing problem, heirarchical fault simulation techniques have also been proposed. These techniques utilize the testing methods already in use in the industrial environments, but with a few enhancements [22]. The enhancements are in terms of a high level representation which is coupled with a gate level representation and simulation is peformed by switching back and forth between the two. These are only temporal solutions to an everlasting problem.

Others have looked at this problem independently. Notable among these is the work in the area of functional testing of microprocessors, described below.

## 2.1 THE THATTE-ABRAHAM TECHNIQUE

S.M. Thatte and J.A. Abraham [30,31] proposed a test generation procedure in which they describe the microprocessor as a graph model at the register-transfer level. Each register in the microprocessor is represented by a node and the data flow among registers during execution is represented as labeled directed edges. In order to derive fault models at a higher level, independent of the details of implementation, they partition the various functions of a microprocessor into five classes :

i) the register decoding function,

ii) the instruction decoding and control function,

iii) the data storage function,

iv) the data transfer function, and

v) the data manipulation function.

General fault models are derived for each of these functions, and tests are derived with only one fault in a function at a time.

The fault model for the register decoding function allows accessing of one or more or none of the registers, erroneously. For the instruction decoding function, the faulty behavior is specified as a wrong instruction being executed, an extra instruction being executed, or no instruction being executed. While, in the fault model for

the data storage function, any cell of a storage module is allowed to be stuck at 0 or 1. The data transfer function includes faults like line stuck-at and coupled lines failing to carry different logic values. And finally, no specific fault model is used for the data manipulation function since there are ".. a variety of designs for the ALU and shift logic..." It is assumed that complete test sets can be derived for these units for a given fault model.

Thatte and Abraham formulated a set of algorithms to generate the necessary test patterns. Each of these algorithms, designed to detect a particular class of faults, steps the microprocessor through a precisely defined set of instructions and addresses. The algorithms were tested on an 8-bit microprocessor and yielded a coverage equivalent to 90% of the single stuck-at faults.

## 2.2 THE ROBACH-SAUCIER TECHNIQUE

C. Robach and G. Saucier [24,25] developed another general graph-theoretical functional level model in an effort to automate generation of test patterns for microprocessors. This approach considers both the functional and the structural features of the microprocessor; functional data being the instruction set and structural/hardware data being the functional block diagram showing the architecture.

An 'abstract execution graph' is associated with every instruction of the microprocessor, indicating the functions and the hardware exercised during the execution of that instruction. The 'sources' and 'sinks' of a graph represent the memory elements (registers and memory), and intermediate nodes are micro-operation nodes. For complex micro-operations, more than one level of this graph is used. There is an edge between a memory element and a micro-operation if the micro-operation processes the data stored in that memory element. Significant test properties for these graphs are looked for, in terms of complexity which is measured by the amount of hardware exercised by the instruction, and accessibility which is measured by the ease of access of the test information while executing the instruction.

Using the relations of functional and structural dominance for the graphs, the instructions of the microprocessor are divided into separate classes, with an instruction belonging to a particular class if and only if it is not dominated by another instruction of a different class. This ordering among the set of instructions is used in the start-small method of testing, where in, each additional test adds a small amount of hardware to the previously tested parts. Any fault detected belongs to the part added for that test.

In the start-big method of testing, the test starts with the verification of a large amount of hardware and if it succeeds, it proceeds with another part. Hence, a minimal set of instructions is chosen which exercises at least once every memory element and every micro-operation. This set consists of the subset of instructions not structurally dominated by any other instruction and an additional subset of instructions needed to achieve full coverage.

Robach and Saucier have also developed algorithms for testing. For every instruction, three sets of test vectors are defined, one each to ensure the graph identity, to check the memory elements, and to check the micro-operations. A study of four 8-bit microprocessors (Signetics 2650, Motorola M6800, Intel 8080, Zilog Z80) and one 16-bit microprocessor (Motorola M68000) showed the applicability of this work.

## 2.3 THE CLARY-JOOBBANI TECHNIQUE

J.B. Clary, R.K. Joobbani and F.M. Smith [10] of the Research Triangle Institute (RTI) developed a methodology for verifying the military computer family built-in-test [MCF-BIT] performance specifications. They used the Instruction Set Processor (ISP) language to describe the functional modules. The functional fault model developed

started with basic fault occurrences at the logic level and related these to their functional level manifestations.

The study was performed with an intent to see what relationships exist between physical faults and functional faults, since the mapping of gate-level faults into functional faults can potentially reduce the number of fault classes considerably. The work is based on Thatte and Abraham's technique which was extended to the PDP-11/70 and partitioned into the same functions. The fault classes were also the same except for two changes :

i) It was necessary to define a fault class for the data manipulation function (ALU) which was not done by Thatte and Abraham. Simulation was done at the gate level to observe different classes of faults for the ALU. This resulted in three major functional faults:

   a) a constant operation on the output i.e., the output is exclusively ORed with a constant m, or the output is added/subtracted with a constant n.

   b) a change of operation i.e., instead of adding two numbers, they are exclusively ORed, or they are subtracted.

   c) an output bit stuck-at-0 or stuck-at-1.

ii) The instruction decode and control function was modified to include six categories instead of three:

a) no instruction is executed

b) instead of executing the correct instruction, a non-existent instruction is executed

c) an instruction is incorrectly executed, not necessarily as another single instruction, but possibly as a combination of instructions (due to microcode not decoding)

d) instead of the intended instruction, some other instruction is executed

e) an extra instruction is executed (due to the failing of the microsequencer), and

f) a 2- or 3-word instruction is executed as two or more instructions.

From this study, several useful relationships were derived between the gate level (stuck-at) faults and their functional manifestations. An important result is that a single gate level fault does not manifest itself at the functional level very accurately and explicitly because of the visibility of the faults and their quick manifestations.

## 2.4 OTHER TECHNIQUES

The techniques mentioned above are based on application of test patterns internally by forcing the UUT to execute a testing procedure, which is possible only for systems like microprocessors that can execute programs. Some test generation techniques for general (V)LSI circuits have also been looked into.

Yinghua Min and Stephen H. Su [21,17] have extended the work of Thatte and Abraham for testing general LSI circuits. The fault classes remain the same and path-sensitization technique is used to generate tests for the five classes of faults.

M.S. Abadir and H.K. Reghbati [1,2] use path-sensitization and D-algorithm [26] on functional descriptions of different modules in the circuit. The model for the functions is a set of binary decision diagrams, introduced as a functional description tool by S.B. Akers [4]. Abadir and Reghbati argue that their technique can be easily automated because these diagrams are amenable to extensive logical analysis on the computer.

# Chapter III

# FUNCTIONAL LEVEL MODELING

## 3.1 GSP : THE SIMULATION LANGUAGE

The functional fault models were prepared using GSP (General Simulation Program) [5,6,8,9,15]. GSP is a general purpose, two-valued (1,0) simulation language which was developed at Virginia Tech specifically to perform the simulation of VLSI devices at the chip level [9]. Its most useful application is the modeling and simulation of complicated VLSI circuits and microprocessors. The language has been used extensively for modeling functional-level faults in simple and complex VLSI devices [27]. It also has the capability to model such interface timing specifications as setup time, hold time and minimum pulsewidth [5,8,9,23].

Modeling in GSP is done in an assembly language with special instructions for hardware description. The instruction set is illustrated in appendix B in the model descriptions. The GSP manual [15] contains a detailed explanation on the usage of each instruction.

The GSP simulation system structure is shown in fig.3 Each module description file is assembled to obtain the microcode file. The microcode files are merged together with the states into the LINK file. The DATA file has the

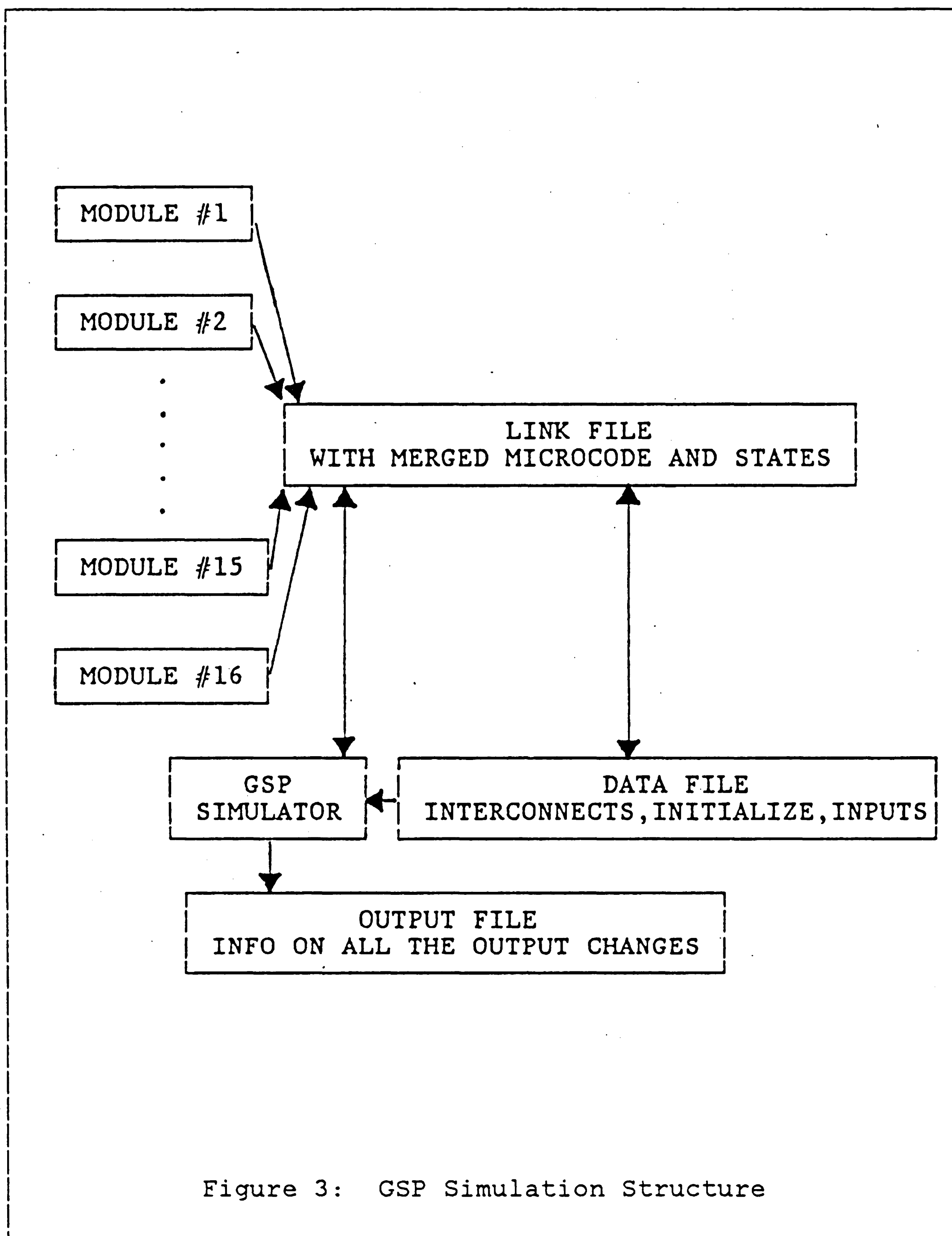



Figure 3: GSP Simulation Structure

information on module interconnections, initializations and inputs. The simulator reads the data file at the beginning of simulation and executes the microcode during simulation, generating the outputs.

## 3.2 METHODS OF FUNCTIONAL MODELING

The modeling process involves :

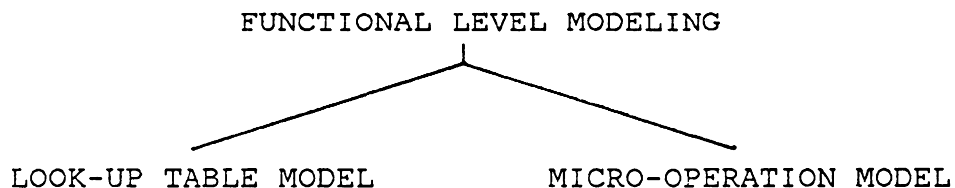
i) detailed examination of manufacturer's specifications,

ii) generation of the model flow-chart,

iii) coding of the model, and

iv) model checkout to verify the correctness of the model.

The functional level models can be prepared in different ways. Two general methods for modeling digital devices at the functional level are shown below :



### 3.2.1 Look-up Table Model

In this method, the functional unit is represented in the form of a truth-table (combinational logic) or a state table (sequential logic). The discussion here pertains to combinational logic. In order to access a particular value in the truth-table, the inputs to the functional unit are decoded to point to the location containing that value in the 'Look-up Table'. Fig.4 shows the hypothetical similarity between the actual function and its look-up table model. As evident, this is a very simple approach to modeling. Several such truth-tables for the different functions are put together to form the model for the whole device. Also, the functional units that are repeatedly used in the device can be made into subroutines and 'called' whenever needed, during the data flow of the device.

In GSP, the decoding constructs are used to perform this operation. The example in fig.5 describes the 'look-up table' model for an And-Or-Invert function of three inputs, $F(x1,x2,x3) = (x1x2 + x2x3 + x3x1)'$. As can be seen from the figure, the number of bits of the input register that are to be decoded, are moved into one of the index registers (index register 1, in the example). The index register is used as the pointer to the locations of a table (table AOI, in the example), and the value contained in the location

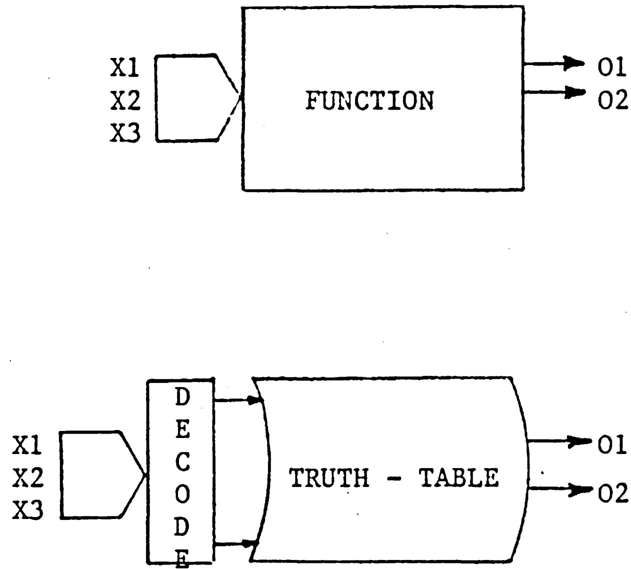



Figure 4: Actual Function vs. Look-up Table Model

pointed by the contents of the index register is then moved out to the destination (pin OUT) after a delay of 40 ns. (DEL1).

### 3.2.2 Micro-operation Model

In this approach, the functional unit is defined as a sequence of model micro-operations, using the constructs of the modeling language. The functional model can be viewed as a nodal graph with two kinds of edges interconnecting the nodes. Each node is a set of model micro-operations and control and data get passed from one node to another along the edges [5]. The dotted lines in fig.6 indicate control transfer while the solid lines indicate the passage of variables from one node to another.

The functional unit is not described in terms of the inputs and the truth table as in the previous case; instead, modeling language constructs are used to manipulate the data and obtain the resultant output. The example in fig.7 describes the 'micro-operation' model for an And-Or-Invert function of three inputs, similar to the one in fig.5 In GSP, modeling constructs like AND, OR, and NOT are used in a sequence of micro-operations which yield the final output.

AND-OR-INVERT (AOI)

```
; registers for the model
;
REG(3)    OLDX
;
; pins for the module  X1,X2,X3 : 1,2,3 ; AOI : 4
;
PIN       X1X3(1,3),OUT(4)
;
; delays for the module
;
EVW       DEL1(40)
;
; module description
;
     BNE   X1X3,OLDX,PROC
     EXR
;
PROC:MOV   X1X3,OLDX        ; STORE FOR NEXT CHECK
     IDX   OLDX(0),3,1      ; STARTING WITH 0TH BIT,
                            ; MOVE 3 BITS INTO INDEX REG.1
     MOV(DEL1) AOI@1,OUT    ; MOV THE CONTENTS OF LOCATION
                            ; POINTED BY INDEX REG.1 TO
                            ; THE OUTPUT, AOI, AFTER DEL1.
     EXR
;
;LOCATIONS 0  1  2  3  4  5  6  7
;
AOI  : BYT #1,#1,#1,#0,#1,#0,#0,#0
;
END
```

Figure 5: "Look-up Table" model

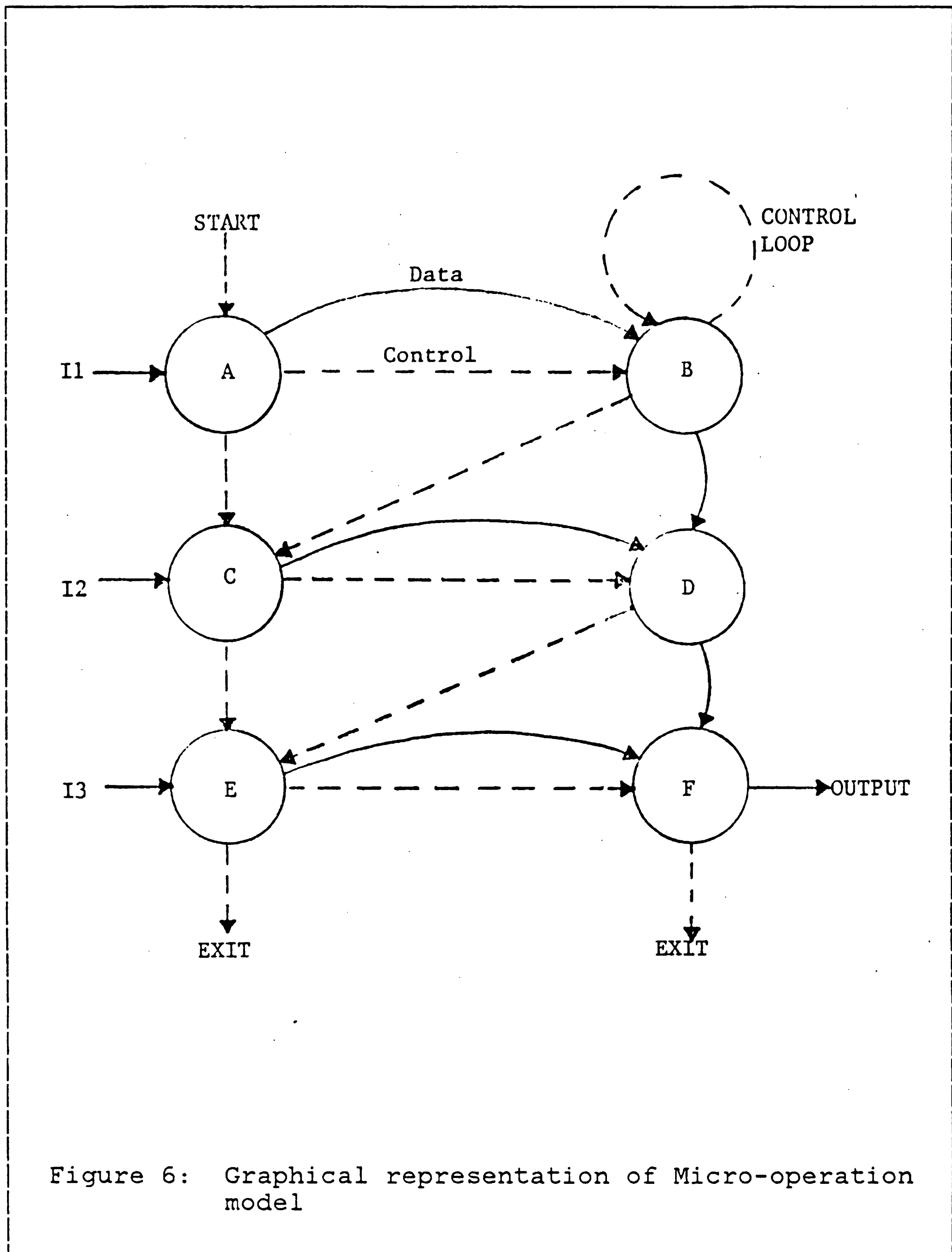



Figure 6: Graphical representation of Micro-operation model

AND-OR-INVERT (AOI)

```
; registers
REG(1)     OLDX1,OLDX2,OLDX3
REG(1)     AND12,AND23,AND31
REG(1)     ORBUF
; pins
PIN        X1(1),X2(2),X3(3),OUT(4)
; delays
EVW        DEL1(40)
;
; description
;
      BNE  X1,OLDX1,PROC  ; BRANCH IF VALUE CHANGED.
      BNE  X2,OLDX2,PROC
      BNE  X3,OLDX3,PROC
      EXR
;
PROC: MOV  X1,OLDX1       ; FOR COMPARISON ON NEXT
      MOV  X2,OLDX2       ; SIGNAL CHANGE OF X1, X2, X3.
      MOV  X3,OLDX3       ;
;
      AND  X1,X2,AND12    ; (AND12) = (X1) . (X2)
      AND  X2,X3,AND23
      AND  X3,X1,AND31
      OR   AND12,AND23,ORBUF
      OR   AND31,ORBUF,ORBUF ; DESTINATION =ORBUF ITSELF
      MOV(DEL1) ORBUF, OUT
      EXR
END
```

Figure 7: "Micro Operation" model

## 3.3 FUNCTIONAL MODELING OF AMAC CHIP

As mentioned in chapter 1, a section of IBM's AMAC chip was used for the purpose of this study. Accurate functional models were prepared for the UUT, based on timing specifications, layouts, and other logic level information [18]. Fig.8 is the block diagram showing the various functional units of the AMAC chip. Three separate models were prepared, one each for the adder, the multiplier, and the register with parity. The three registers and their odd parity generators work in the same way and hence, only one model was developed for the registers. But, for the purpose of simulation, three copies of the same model were used with the required modifications.

The flow-chart of fig.9 shows the approach taken during the research involving the simulation of faults and the generation of functional level test vectors. As shown in the figure, two different approaches were tried for obtaining a list of functional faults :

i) Functional level defects obtained solely from the model description, i.e. model perturbation, and

ii) Circuit level defects mapped into functional level defects.

The procedure resulted in two fault lists; List 1 consisted of faults from the models and List 2 consisted of circuit level defects mapped onto functional level defects. The two lists were merged and faults were injected into the functional descriptions of the circuits under test. Functional level Simulations were then performed with several different inputs to obtain the fault detection vectors. These functional level test vectors were subsequently run on the gate level models furnished by IBM, to determine the equivalent gate level coverage. Comparison of the gate and functional coverage levels allowed inference of rules for functional fault modeling, and in some cases the experience gained required further iterations of the process.

### 3.3.1 The Adder

The Adder is a 16-bit ALU consisting of four 4-bit slices. Each slice is a TI 74181 ALU. The carry-look-ahead circuit provided with the four ALU's obtains a substantial speedup. The adder can potentially perform 48 operations, but in the AMAC unit it is used only in one particular mode to realize 32 functions. Table &Add gives the ALU functions used.

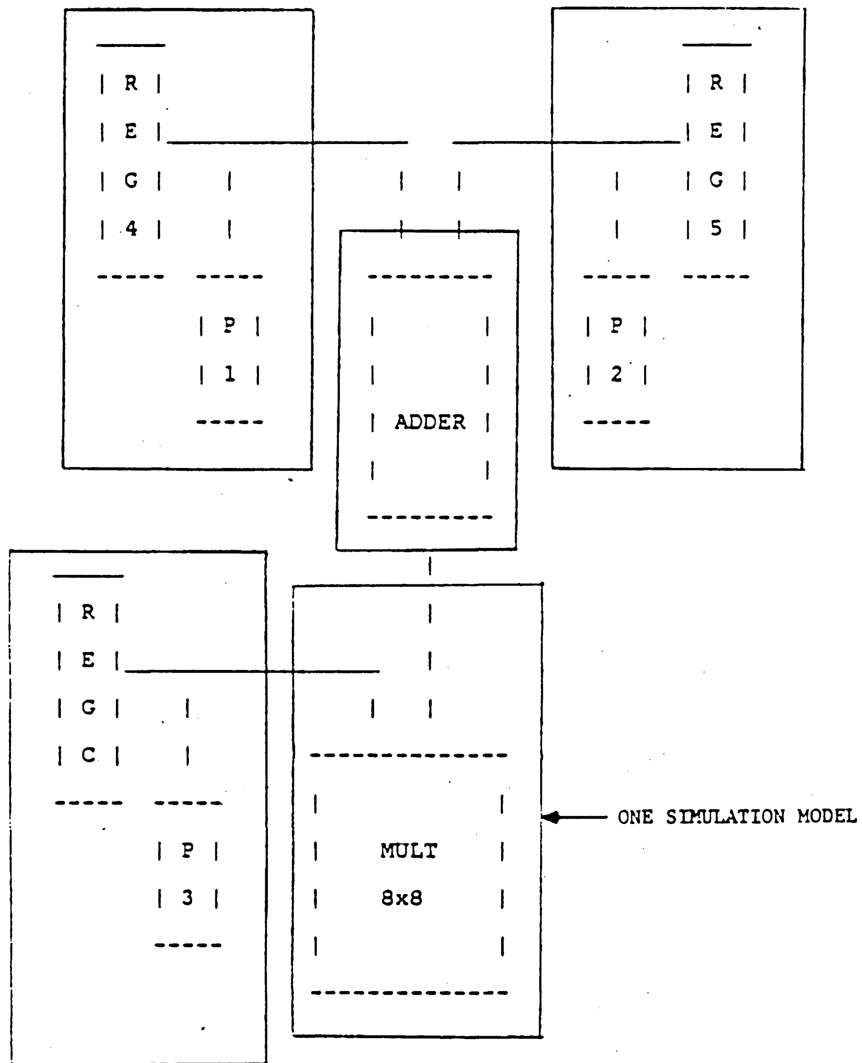



Figure 8: Functional Units of the AMAC model

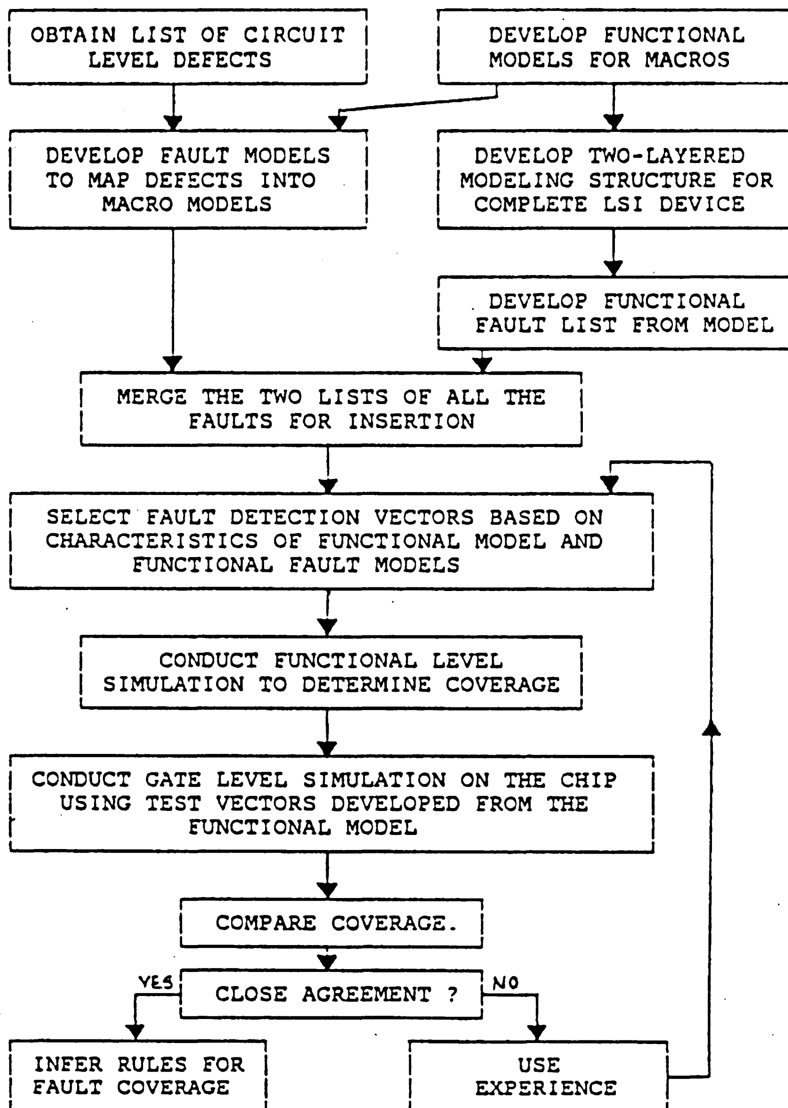



Figure 9: Functional Fault Modeling Methodology for the AMAC chip

The model for the adder was prepared as a single functional unit. Each of the 32 functions performed by the adder is defined as a set of micro-operations of the model. The control inputs are decoded to jump to the section of the code containing the corresponding set of micro-operations. These micro-operations manipulate 16-bit data at a time instead of 4-bit slices of data. The carry-look-ahead function was modeled implicitly with the data manipulation. Thus, the model was not based on the physical structure of the circuit. This was done to obtain a higher level description which also speeds up the simulation to a great extent.

### 3.3.2 The Multiplier

The AMAC chip has a fully combinational, very high speed Multiplier. It uses a modified Dadda-type scheme [11] to generate the 16-bit product of two 8-bit numbers. The numbers are represented in signed 2's-complement notation. The multiplier uses a set of input and output latches to store the data temporarily. Summands are obtained from the latched inputs and are used in groups of two or three to generate partial sums and carry's. Three stages of full adders are used to obtain the final sum and carry. A carry-look-ahead circuit then hastens the calculation of the final product.

TABLE 1

TI 74181 ALU Functions used in AMAC chip

| Selection | | | | M = L (Arithmetic Operations) | |
|---|---|---|---|---|---|
| S3 | S2 | S1 | S0 | Cnbar = H | Cnbar = L |
| L | L | L | L | F = A | F = A plus 1 |
| L | L | L | H | F = A + B | F = (A + B) plus 1 |
| L | L | H | L | F = A + B' | F = (A + B') plus 1 |
| L | L | H | H | F = minus 1 (2's compl) | F = zero |
| L | H | L | L | F = A plus AB' | F = A plus AB' plus 1 |
| L | H | L | H | F = (A+B) plus AB' | F = (A+B) plus AB' plus 1 |
| L | H | H | L | F = A minus B minus 1 | F = A minus B |
| L | H | H | H | F = AB' minus 1 | F = AB' |
| H | L | L | L | F = A plus AB | F = A plus AB plus 1 |
| H | L | L | H | F = A plus B | F = A plus B plus 1 |
| H | L | H | L | F = (A+B') plus AB | F = (A+B') plus AB plus 1 |
| H | L | H | H | F = AB minus 1 | F = AB |
| H | H | L | L | F = A plus A* | F = A plus A plus 1 |
| H | H | L | H | F = (A+B) plus A | F = (A+B) plus A plus 1 |
| H | H | H | L | F = (A+ B') plus A | F = (A+B') plus A plus 1 |
| H | H | H | H | F = A minus 1 | F = A |

The flow of control and data in the model for the multiplier was modified very little from the one in the actual layout. In this sense the model is based on the structure, but most of the individual functions have been defined non-structurally, using the two methods outlined above.

### 3.3.3 The Register with parity generator

The registers in the circuit are dual latches with serial-shift capability for scan. These are popularly known as LSSD registers [12]. Scan capability is added to sequential circuits in order to simplify testing. In the test-mode of operation, the registers are loaded serially with the desired patterns of binary values. These patterns are then passed through one stage of combinational logic, the results of which are latched to the subsequent stage of registers. The output is serially shifted out and checked for faults in the preceding stage of logic.

An odd-parity generator provided with each register in the UUT is used to check the parity of the data latched into the register.

Each register in the chip was modeled with its scan operations and its parity-check circuit [16]. The model was prepared using a combination of both the techniques specified above.

Appendix A has the flow-charts of all the models. The listings of the GSP description files for all the three models are given in appendix B.

# Chapter IV

# FAULT MODELING AT THE FUNCTIONAL LEVEL

## 4.1 THE MODEL PERTURBATION METHOD

There are two basic approaches to functional fault-modeling as shown in fig.10 In one, the physical structure of the device is given importance and circuit level defects are mapped onto functional level faults. While in the other, the correct functional model description/procedure is faulted to obtain an incorrect version of the procedure. The fault itelf MAY or MAY NOT be directly related to any specific circuit-level defect. Once faults are selected, various input vectors are tried to obtain test vectors that can detect those faults.

Previous research in this area shows that the first approach yields good results for gate-level simulation because of the closeness of the model description to the physical layout of the device. But the same does not hold for the case of functional-level simulation wherein, the physical structure of the device is not explicitly defined in the model.

As such, for functional-level fault simulation, the second approach is favored. It will be called the Model-Perturbation [MP] approach. However, for the AMAC

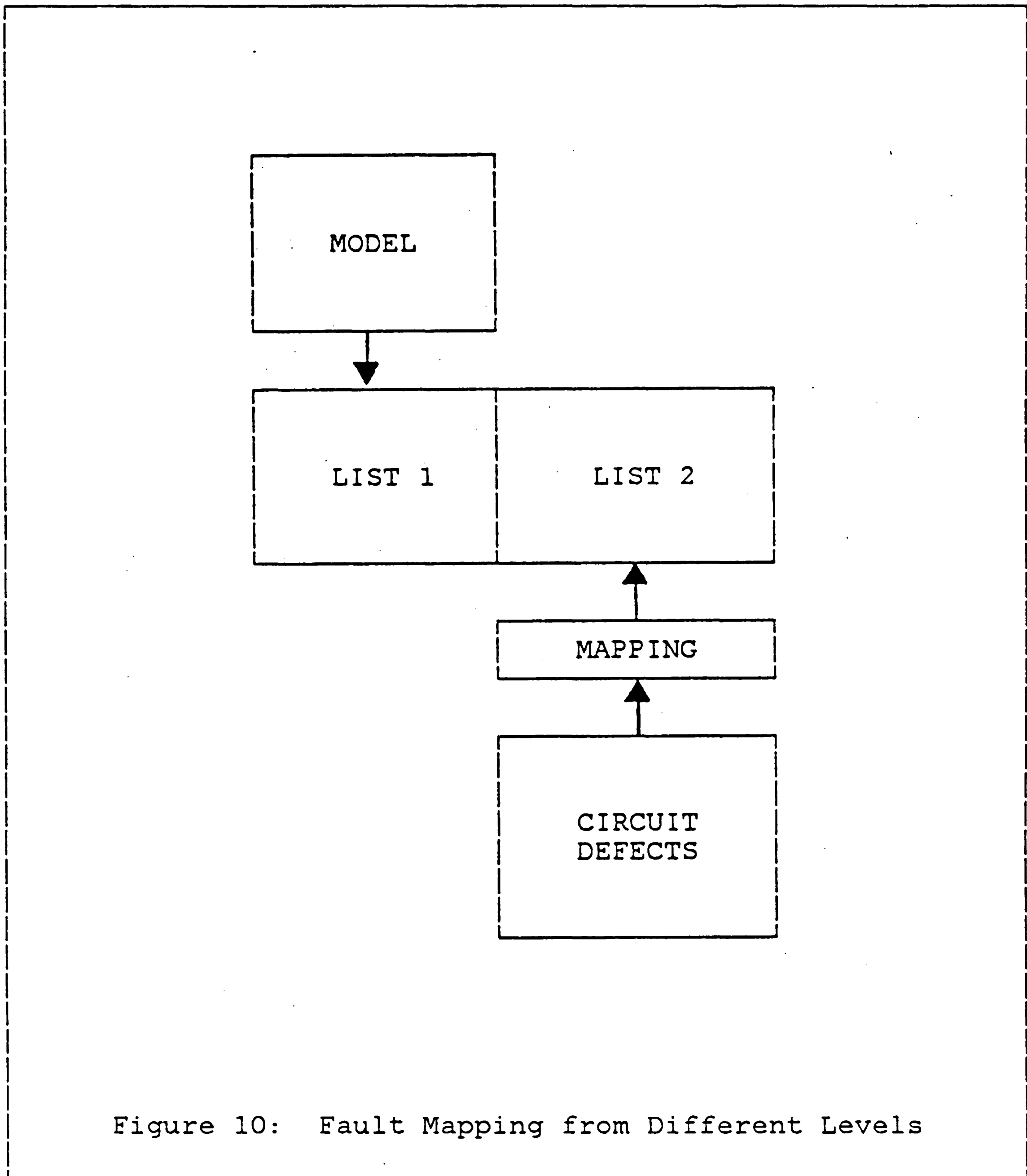


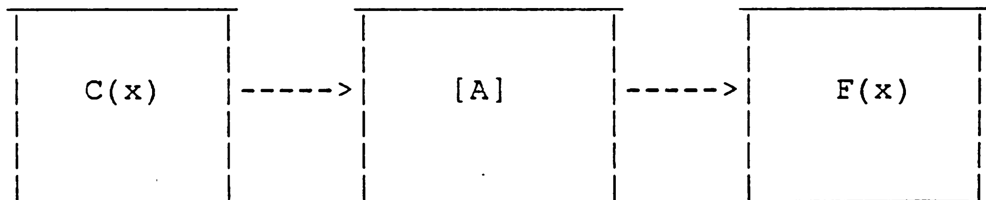

Figure 10: Fault Mapping from Different Levels

functional unit, simulations were performed using both the methods, with greater emphasis on Model-Perturbation.

Model-Perturbation can be defined in the following way. If the correct model-procedure is C(x) and the set of faulty model procedures is F(x), where 'x' is a set of inputs, then the transformation from the correct model procedure to the faulty one, can be represented as :

$$F(x) = [A]\ C(x)$$

where the transfer function, [A], is a set of faults injected into C(x).

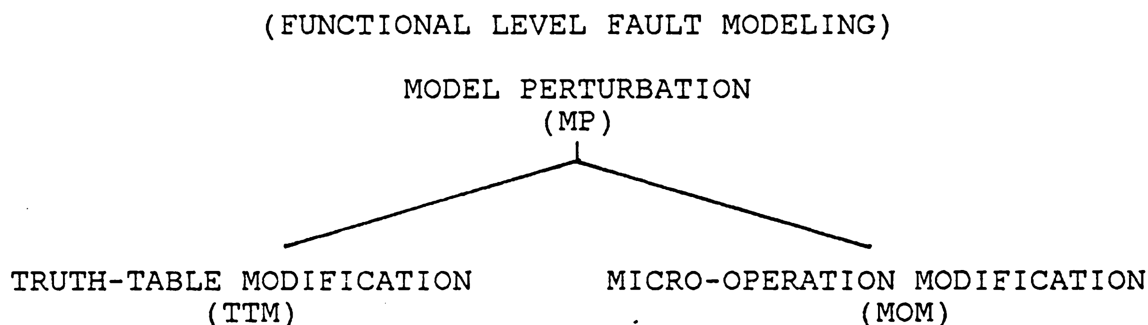



MODEL-PERTURBATION

This thesis deals with the finding of the operator [A] in order to maximize coverage of faults at the functional level. The extent of coverage is defined in terms of the gate-level coverage for the present, but the aim is to finally obtain an independent definition based on the results of the work in this area.

The models can be 'perturbed' in certain ways depending on the method employed for functional modeling.

Corresponding to the two methods of functional modeling described in chapter 3, there are two methods of model perturbation : "Truth-Table Modification" and "Micro-Operation Modification". In the modeling and simulation performed on the AMAC functional unit, both these methods were adopted.




### 4.1.1 Truth-Table Modification (TTM)

Detectable faults in a block of functional logic result in truth-tables with modified outputs. Hence, the truth-table ( 'look-up table' in GSP ) can be modified in many ways to simulate different faults. This is illustrated in fig.11 for the previously cited AOI example. The AOI table now contains certain incorrect values and for

corresponding combinations of the input, incorrect values get moved to the output.

It should be noted that the tables are not exhaustively modified because it would result in too many possible combinations, with each combination being tried for each run. For example, a table with 8 ( three inputs, 2**3 ) entries can be modified in (2**8 - 1) ways! It would take very long to run all these simulations even at the functional level. In the fault analysis done on the UUT, the fault-list for table modification was obtained from a study of the VLSI layouts for circuit level faults. The circuit-level faults were then mapped onto functional faults. Reference [20] has a detailed description on the choice of these circuit-level defects.

The extent of coverage using this method is dependent on the complexity of the model because for a simple function, there are not many different ways in which the look-up table can be faulted/modified. The functions employed in the modeling were simple. As a result, the number of test vectors obtained is not high and the few test vectors obtained are capable of detecting most of the injected faults.

One important aspect of this process is the method of mapping circuit level defects to the functional level

FAULT-INJECTED AND-OR-INVERT

```
; registers for the model
;
REG(3)    OLDX
;
; pins for the module  X1,X2,X3 : 1,2,3 ; AOI : 4
;
PIN       X1X3(1,3),OUT(4)
;
; delays for the module
;
EVW       DEL1(40)
;
; module description
;
     BNE   X1X3,OLDX,PROC
     EXR                      ; EXIT AND RESTART
;
PROC:MOV   X1X3,OLDX          ; STORE FOR NEXT CHECK
     IDX   OLDX(0),3,1        ; STARTING WITH OTH BIT, MOVE
                              ; 3 BITS INTO INDEX REG. 1
     MOV(DEL1) AOI@1,OUT      ; MOV THE CONTENTS OF LOCATION
                              ; POINTED BY INDEX REG.1 TO
                              ; THE OUTPUT, AOI, AFTER DEL1.
     EXR                      ; EXIT AND RESTART
;
; table is modified from the previous one, for
;    locations 2,5,7
;
;LOCATIONS  0  1  2  3  4  5  6  7
;
AOI :  BYT #1,#1,#0,#0,#1,#1,#0,#1
;
END
```

Figure 11: Truth-Table Modification Technique

faults. For the AMAC model (UUT), each circuit level defect chosen was studied to see how it affects the behavior of the corresponding function. This was done 'manually' and can be a serious limitation in terms of automating the fault simulation process. Inspite of its limitations, this method is attractive as it is very simple and easy to inject faults. The whole process of table modification and simulation of the model description can be automated. Chapter 5 discusses the fault-injection aspects of this method.

#### 4.1.2 Micro-operation Modification (MOM)

Every model description consists of model micro-operations<1> which define the behavior of the simulated device. The micro-operation model is especially well-suited for the method of fault modeling discussed here. In this method, the correct model of the device is taken to be an entity in itself and any relationship between the model and the physical circuit which it represents, is transparent to the fault modeling process. Thus, for all purposes of fault simulation, the model is the actual circuit. This assumption results in the requirement that the functional model available should be checked thoroughly for

<1> model micro-operations are, in general, not the same as the device micro-operations.

correct operation before fault simulation is performed on it. However, this is not any extra requirement on the modeling process; it is a well defined part of it.

As mentioned before, the micro-operation model is prepared from the control and data-manipulation constructs of the modeling language. It should be noted that the control type micro-operations of the model need not necessarily correspond to the control signals of the simulated device. The control type micro-operations of the model include conditional and unconditional branches, loops, decoding functions like CASE or computed GOTO statements, and jumps to subroutines or to some other section of the model-program. All other micro-operations constitute the data-manipulation micro-operations of the model.

Functional level fault modeling is done in terms of modifying these model micro-operations to make them faulty, one at a time. And simulations are performed to obtain inputs that detect each of these functional faults. The single-fault assumption is used only to simplify test generation. The test vectors obtained based on this assumption hold even for the case of multiple faults, unless a fault at one point gets 'corrected' by a fault at another.

The faulty control type micro-operations will be referred as control faults (FCON) and all other types of

faulty micro-operations shall be called <u>micro-operation faults</u> (<u>FMOP</u>). Micro-Operation Modification involves simulation with both types of faults.

To obtain the test vectors for the AMAC functional unit, the GSP modeling constructs like AND, OR, XOR, ADD, and SUB were modified. In addition, control constructs such as BEQ and BNE were also changed. Modifications like replacing each occurrence of AND by OR, OR by AND, ADD and SUB by XOR, ADD and SUB by OR, and changing conditional-branches to unconditional-branches and vice-versa, resulted in the generation of a substantial number of test vectors. Thus most of the model micro-operations were replaced by their logical duals to perform functional fault simulation. This technique of failing the model to the 'incorrect' mode, which is the <u>LOGICAL DUAL<1></u> of the 'correct' mode of operation, yielded very encouraging results in terms of fault coverage.

Perhaps, the biggest argument in support of this method is its simplicity and regularity, which make it amenable for automation. Automatic fault-injection and test-generation are the requirements for any fault-model to be used extensively, apart from the necessary condition that it yield high coverage to keep the defect level low.

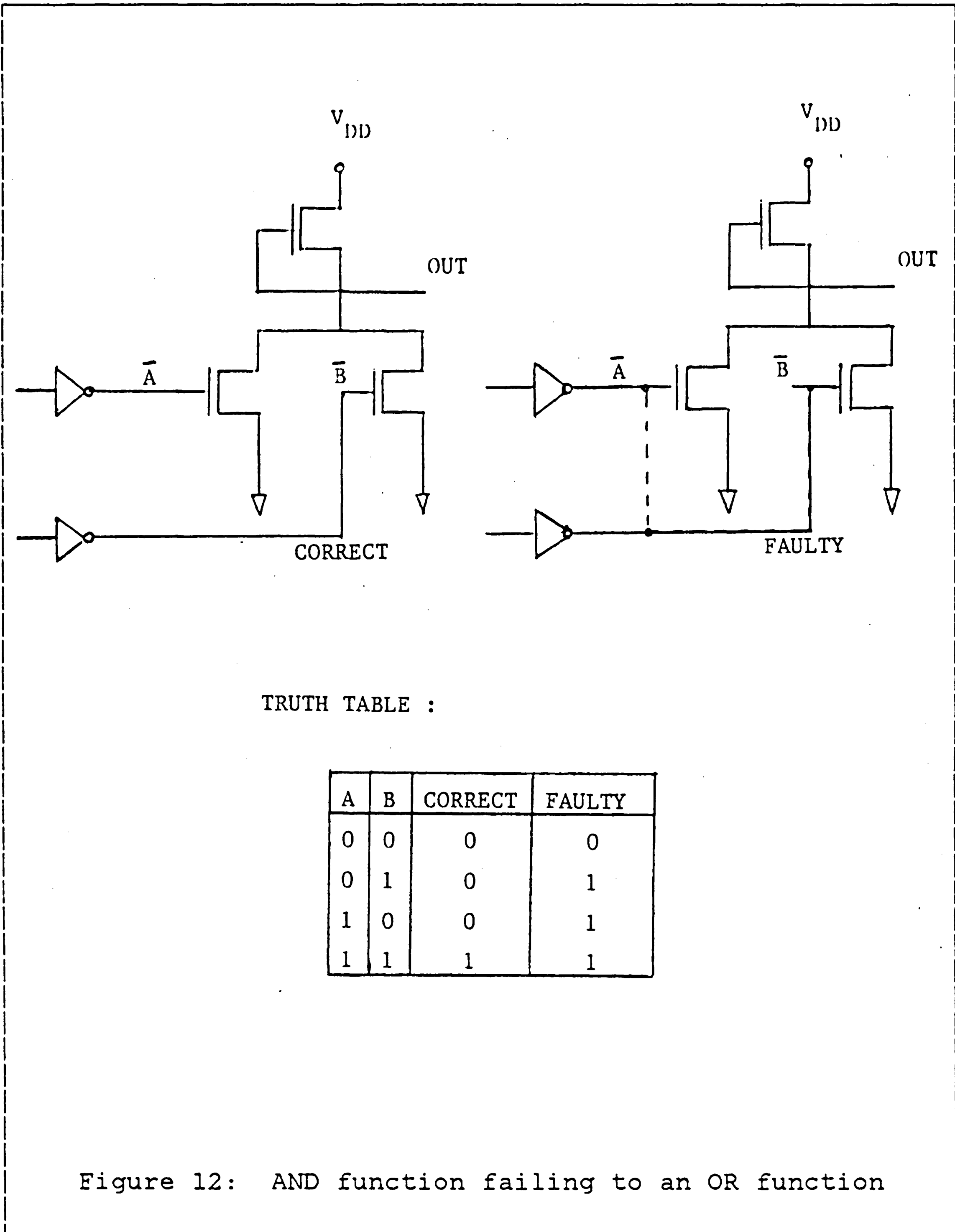
---

<1> Some Logical duals are : XOR vs. Equivalence, OR vs. AND.

A study of the circuit level faults offers a second argument in support of this technique. The actual logic implementing the model micro-operations can fail in many ways, resulting in the incorrect execution of the operation. The individual elements of these logic blocks are the transistors, which are connected in different ways to perform the corresponding function. Some of the faults occurring at the circuit level (e.g. in interconnections) can be directly related to the incorrect operation taking place at the higher level [6].

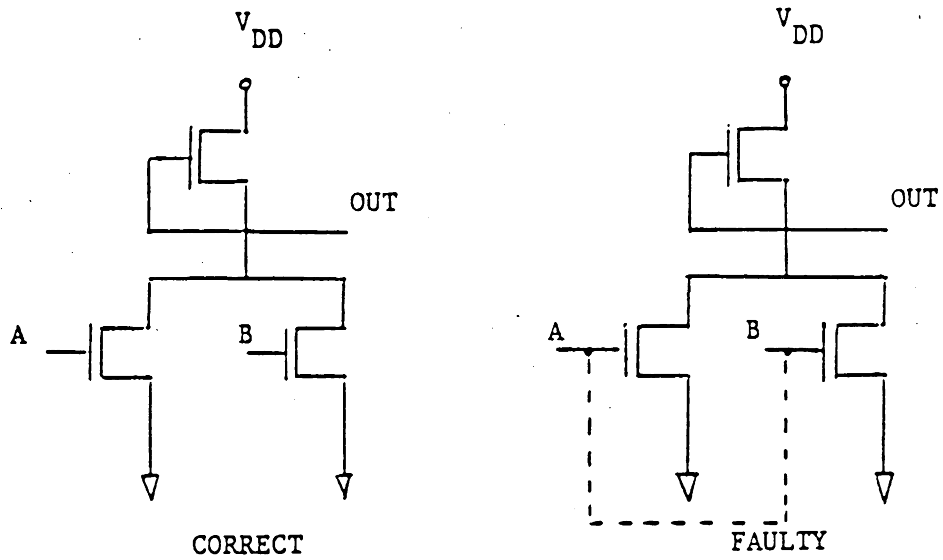
For example, a two-input AND function "fails to" (behaves as) an OR 'gate' if the complemented inputs are shorted [fig.12]. This is a common type of failure mode for this circuit since the inputs are layout neighbours. Fig.12 gives the truth table for both correct and faulty behavior [20]. In yet another case, for the same type of failure, a NOR function "fails to" a NAND function when the gate-lines of its input transistors get shorted. Fig.13 depicts this behavior [20]. The same physical failure results in two different functional faults : AND failing to an OR, and NOR failing to a NAND.

Similar behavior in other types of circuits leads us to conclude that we can directly relate micro-operation faults to certain circuit-level defects. As such, the functional




| A | B | CORRECT | FAULTY |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Figure 12: AND function failing to an OR function




TRUTH TABLE :

| A | B | CORRECT | FAULTY |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

Figure 13: NOR function failing to a NAND function

level micro-operation model can be 'perturbed' by replacing the 'correct' modeling construct by the 'incorrect' one, which is its logical dual.

# Chapter V

# AUTOMATIC FAULT-INJECTION AND TEST-GENERATION

## 5.1 TECHNIQUES FOR FAULT-INJECTION AND TEST-GENERATION

Every phase of the testing process is equally important. The availability of a very good fault model which cannot be used to generate tests automatically on the computer is a travesty in itself. In other words, a good fault model is one for which techniques exist to inject faults and generate tests automatically.

To test the validity of the methods of testing proposed in this thesis, functional level faults were injected into the correct model procedure of the UUT, one at a time. This was done 'manually', by making multiple copies of the correct procedure and modifying each copy to 'inject' a fault in it. Thus, as many 'faulty procedures' were obtained as the number of faults.

Simulation was then performed on each faulty procedure. Various inputs were tried on each procedure until an input detected the fault injected i.e., the output of the faulty procedure differed from the output of the correct one. Each such 'test vector' was added to the set of test vectors already obtained. When trying out a test vector for a new fault, the set of vectors already obtained was tried first.

If none of the vectors in the test-set detected the fault, a new vector was tried. All these simulations were performed manually, too. Needless to say, the process is both laborious and time-consuming.

Having stressed the need for automation, it is but imperative to look for algorithms for automatic fault injection and test generation.

### 5.1.1 Automatic fault-injection

The success and popularity of the stuck-at fault model at the gate level can be attributed to the ease with which nets in a circuit can be stuck to 1's and 0's. The functional fault model proposed in this thesis is an equally simple model, if not more. A simple program can be written to alter the micro-operations of the model procedure in the desired fashion, one after the other, until all the possible alterations have been used for test generation. Since the modified micro-operations are the logical duals of the correct ones, the algorithm becomes very simple and fast. Although the algorithm remains general, the actual speed and complexity of the implementations depend on the language used for functional modeling.

To implement the automatic procedure for the modeling language (GSP) used in this study, it is suggested that the

link file obtained from the linking of the assembled microcode be 'injected' with the faults. By altering the microcode directly, reassembly of the procedure is avoided and the process of fault-injection is hastened. The creation of multiple copies of faulty procedures can also be avoided by this technique, if the executable code is altered at simulation time, then recorrected after a test vector has been obtained, and a new fault injected for the next simulation run.

#### 5.1.2 Automatic Test Generation

Path-sensitization and D-algorithm, or some modifications of these [13], are the popular test generation techniques at the gate level. At the functional level, the manual method described above can be automated to generate test patterns. The algorithm to do it would be :

1. For the R(th) fault, apply a test vector [P] from the obtained vector list. If no vectors in the list, goto 3.
2. If the applied test vector from the list detected the fault, goto 5.
   Otherwise, P=P+1 (the next vector) and goto 1.
3. Vectors from the previous list did not detect the fault. Apply a new vector [N].

4. If new vector detected the fault, add it to the test vector list.

   If it did not, N=N+1 (another vector) and goto 3.

5. Make R=R+1 (the next fault),

   P=1 (the first vector in the list) and goto 1.

   If all faults tested, stop.

It should be noted that this algorithm does not give the minimal test-set because the new vectors added to the list are not tried again for faults already detected by previously used vectors. The algorithm will be far more complex and slow if the test-set were made minimal. Another problem that remains unsolved is the method of choosing the 'new' vector, as stated in the algorithm above.

Thus, the whole process of fault injection and test generation can be automated. Further study needs to be done in this area for better algorithms and also towards the method of generation of the test vectors.

# Chapter VI

# CONCLUSIONS

The methods for functional modeling and fault simulation proposed and described in this thesis were applied for modeling and fault simulation of the AMAC functional unit (under IBM contract YD 190121). As part of this work, functional level models were developed and model checkout was done to insure the correct functioning of the models. Faults were then selected for the macros, and test vectors were developed through computer simulation.

The results of the simulation are very encouraging. In all, 857 functional faults were injected into the models and 110 test vectors detected all these faults. Thus, the functional level coverage was 100% of the functional fault list. The same 110 test vectors were run on the gate level model available from IBM and the equivalent gate level coverage was 88.60% of the 8919 gate level faults. The high gate level coverage of 88.60%, obtained from the application of the functional level test vectors, lends credibility to the method of functional fault modeling developed in this thesis.

In conclusion, 'Model Perturbation', as formulated here, is a simple, feasible and effective technique to

inject functional level faults in VLSI systems. It has also been shown that the simplicity of the TTM and MOM procedures for fault injection makes them viable for automation. The new concept of injecting faults in terms of the logical duals of the modeling constructs (MOM) simplifies the whole procedure of fault simulation.

Further work in this direction should yield improved coverage. Simulation of more complex sequential circuits in the future should help in coming up with a well defined classification of functional level faults and an independent definition of functional fault coverage.

Apart from this, algorithms need to be developed to automate the process of functional fault injection and test pattern generation.

# BIBLIOGRAPHY

1. Abadir M.S. and Reghbati H.K. : 'Test Generation for LSI: A Case Study', 21st IEEE Design Automation Conference, June 1984.

2. Abadir M.S. and Reghbati H.K. : 'LSI Testing Techniques', IEEE MICRO, February 1983.

3. Agrawal V.D., Seth S.C., Agrawal P. : 'LSI Product Quality and Fault Coverage', 18th IEEE Design Automation Conference, June 1981.

4. Akers S.B. : 'Test Generation Techniques', COMPUTER, Vol. 13, No.3, March 1980.

5. Armstrong J.R. : 'Chip Level Modeling of LSI Devices', IEEE Transactions on Computer-Aided Design, Vol. CAD-3, No.4, October 1984.

6. Armstrong J.R., Gupta A.K., Stewart J.B. : 'Interim Report for IBM Contract : Functional Fault Modeling for VLSI Devices', Dept. of Electrical Engineering, VPI & SU, Blacksburg, Virginia, May 1984.

7. Armstrong J.R. : 'Interim Report for IBM Contract : Functional Fault Modeling for VLSI Devices', Dept. of Electrical Engineering, VPI & SU, Blacksburg, Virginia, December 1983.

8. Armstrong J.R. : 'Chip Level Modeling and Simulation', SIMULATION October 1983.

9. Armstrong J.R. and Devlin D.E. : 'GSP: A Logic Simulator for LSI', Proc. 18th IEEE Design Automation Conference, 1981.

10. Clary C., Joobbani R.K., Smith F.M. : 'Development of a Methodology for verifying Military Computer Family Built-In-Test Performance Specifications', Research and Development Technical Report, CORADCOM -80-0780-F, New Jersey, May 1980.

11. Dadda L. : 'Some Schemes for Parallel Multipliers', MAGGIO, Vol. XXXIV, N.5, 1965.

12. Eichelberger E.B. and Williams T. : 'A Logic Design Structure for LSI Testing', Proc. 14th IEEE Design Automation Conference, New Orleans, June 1977.

13. Goel P. : 'An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits', IEEE Transactions on Computers, Vol. C-30, No.3, March 1981.

14. Graf M.C. : 'Testing - A Major Concern for VLSI', SOLID STATE TECHNOLOGY, January 1984.

15. 'GSP Users Manual', Dept. of Electrical Engineering, VPI & SU, Blacksburg, Virginia, December 1982.

16. Hong S.J. and Ostapko D.L. : 'A Simple Procedure to Generate Optimum Test Patterns for Parity Logic Networks', IEEE Transactions on Computers, Vol. C-30, No.5, May 1981.

17. Hsieh Y.I. and Su S.Y.H. : 'Testing Functional Faults in Digital Systems described by Register Transfer Language', Digest of papers, 1981 IEEE Test Conference, 1981.

18. IBM : Information and Specifications provided by IBM on AMAC chip , Fall 1983.

19. IEEE : 'IEEE Standard Dictionary of Electrical and Electronic Terms' IEEE Press, New York, 1978.

20. Stewart J.B. : Work toward Masters Thesis, Dept. of Electrical Engineering, VPI & SU, 1984-1985.

21. Min Y. and Su S.Y.H. : 'Testing Functional Faults in VLSI', Proc. 19th IEEE Design Automation Conference, 1982.

22. Muehldorf E.I. and Savkar A.D. : 'LSI Logic Testing - An Overview', IEEE Transactions on Computers, Vol. C-30, No.1, January 1981.

23. Puthenpurayil V. : 'Functional Level Modeling of Digital Devices', Masters Thesis, Dept. of Electrical Engineering, VPI & SU, September 1982.

24. Robach C. and Saucier G. : 'Microprocessor Functional Testing', 1980 IEEE Test Conference, 1980.

25. Robach C., Saucier G., Aleonard C. : 'Microprocessor Systems Testing : a review and future prospects', EUROMICRO journal, January 1979.

26. Roth J.P., Bouricius W.G., Schneider P.R. : 'Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Lo IEEE Transactions on Electronic Computers, Vol. EC-16, No.5, October 1967.

27. Sathe S.K. : 'Functional Level Fault Simulation in LSI Devices', Masters Thesis, Dept. of Electrical Engineering, VPI & SU, June 1982.

28. Siewiorek D.P. and Lai L.K. : 'Testing of Digital Systems', Proc. of the IEEE, Vol. 69, No.10, October 1981.

29. Stapper C.H., McLaren A.N., Dreckmann M. : 'Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and Partially Good Product', IBM J. Res. Develop., Vol. 24, No.3, May 1980.

30. Thatte S.M. and Abraham J.A. : 'Test Generation for Microprocessors', IEEE Transactions on Computers, Vol. C-29, No.6, June 1980.

31. Thatte S.M. and Abraham J.A. : 'Test Generation for General Microprocessor Architectures', Fault Tolerant Computing Symposium, Madison (USA), June 1979.

32. Wallace C.S. : 'A Suggestion for a Fast Multiplier', IEEE Transactions on Electronic Computers, Vol. EC-13, February 1964.

33. Williams W.T. and Parker K.P. : 'Design for Testability - A Survey', Proc. of the IEEE, Vol.71, No.1, January 1983.

34. Williams T.W. and Parker K.P. : 'Testing Logic Networks and Design for Testability', COMPUTER, October 1979.

Appendix A

FLOW-CHARTS FOR THE AMAC MODELS

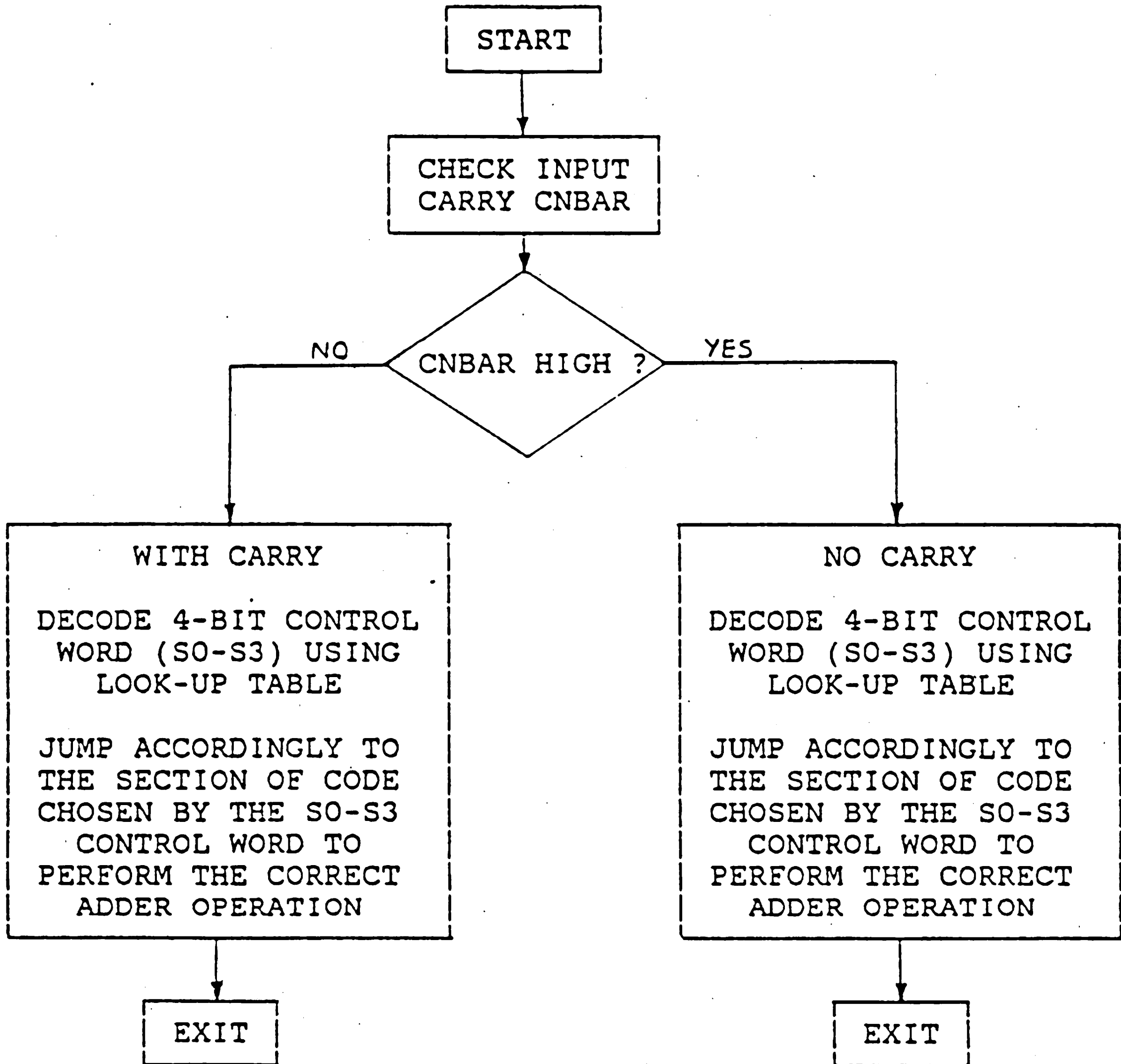



Fig.i Flow-chart for the ADDER

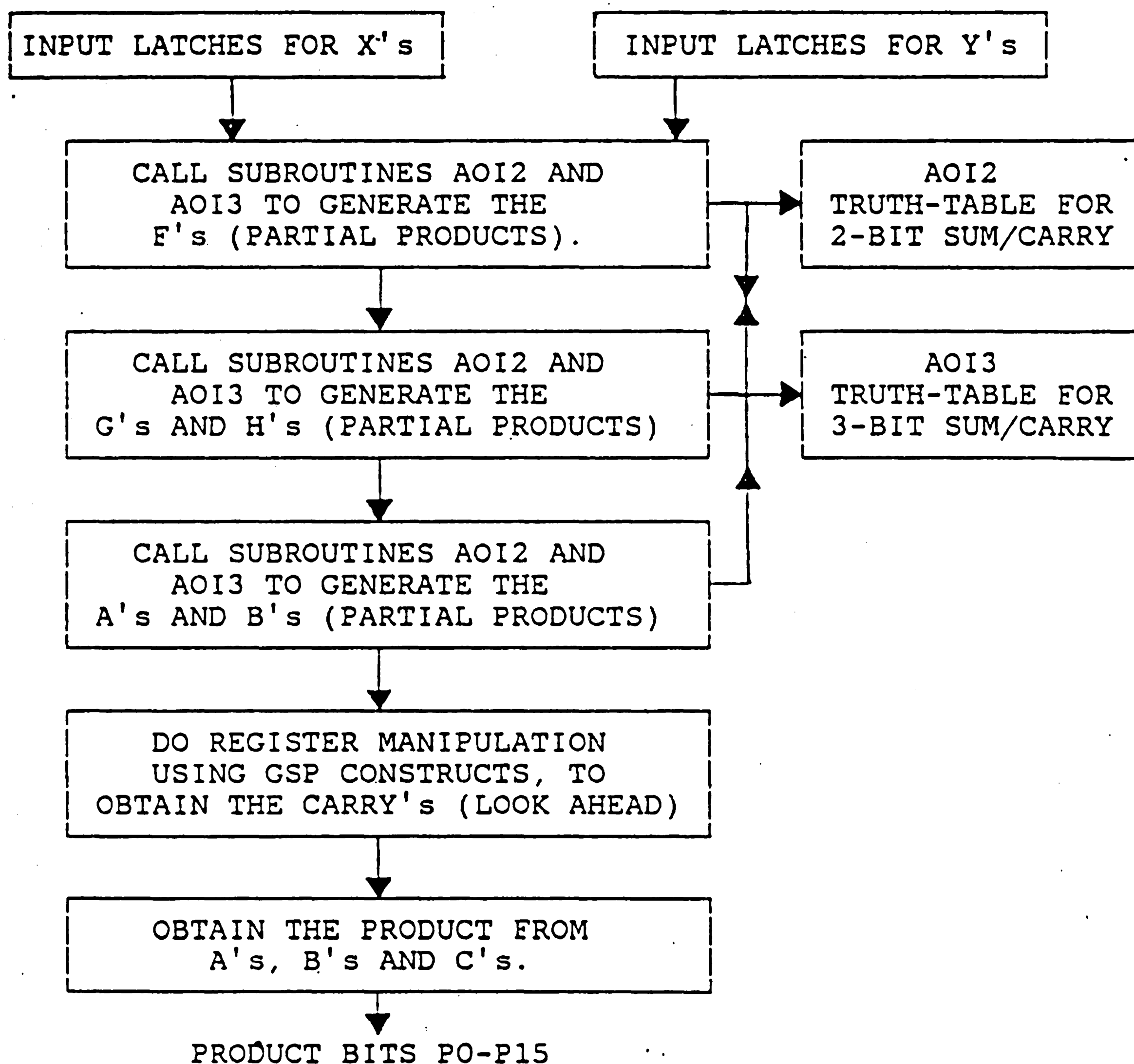



Fig.ii Flow-chart for the 8X8 MULTIPLIER.

START

INPUT CHANGE ?

NO

D0-D2 CHANGE ?

YES

STORE NEW VALUE

USE TRUTH-TABLE TO GENERATE 3-BIT EVEN PARITY

D3-D5 CHANGE ?

YES

STORE NEW VALUE

USE TRUTH-TABLE TO GENERATE 3-BIT ODD PARITY

D6-D8 CHANGE ?

YES

STORE NEW VALUE

USE TRUTH-TABLE TO GENERATE 3-BIT ODD PARITY

USE INDIVIDUAL BLOCK PARITY'S TO OUTPUT 9-BIT PARITY

EXIT

Fig.iii Flow Chart for the 9-bit Parity Generator

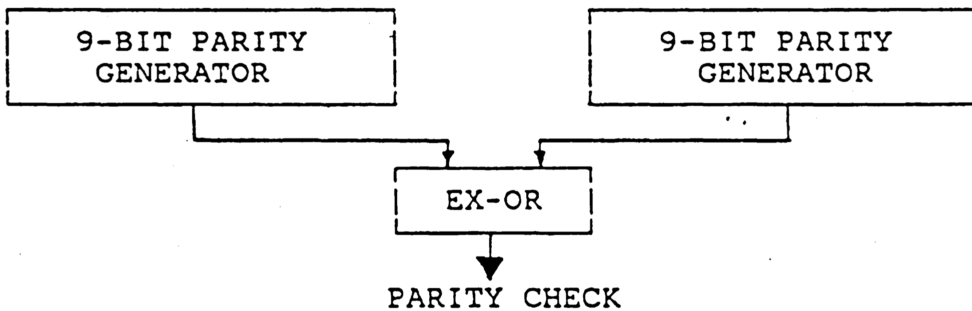



Fig.iv Odd-Parity Generator for REG4 and REG5.

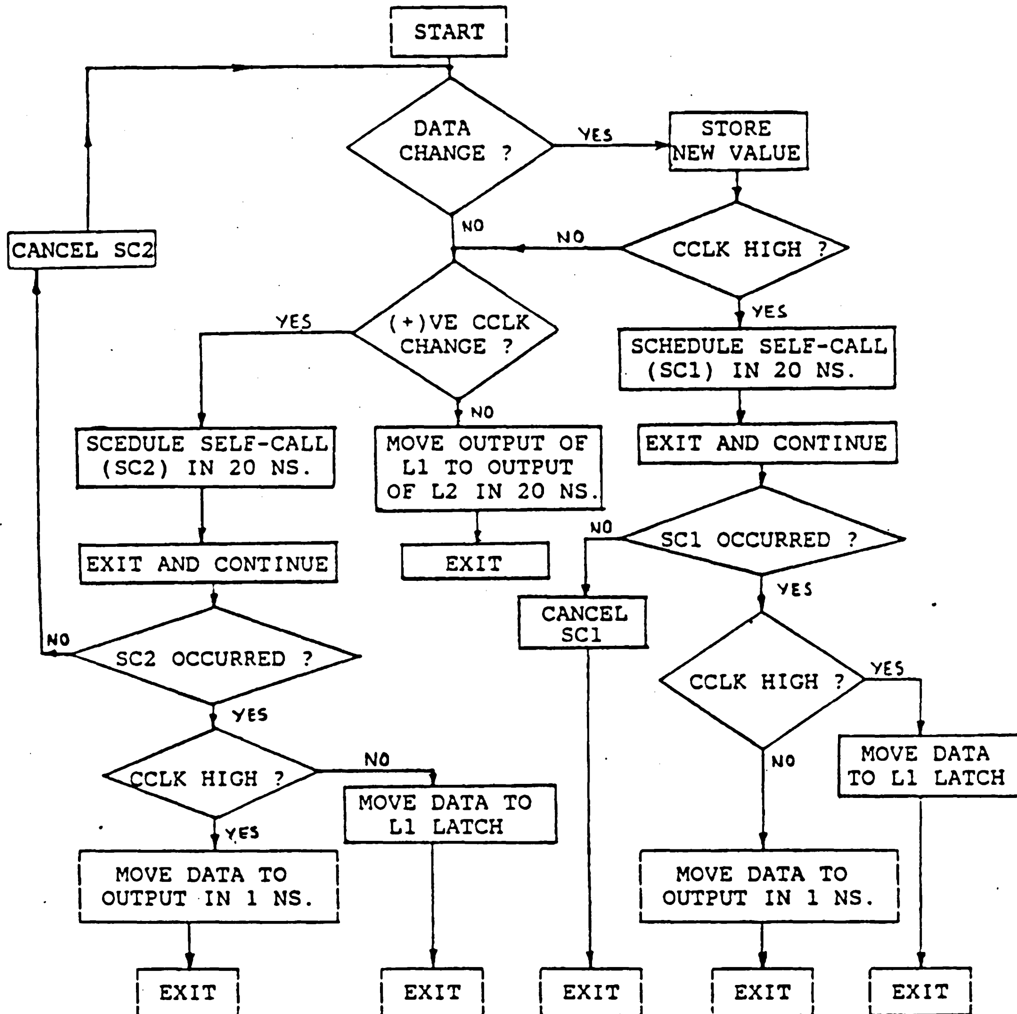



Fig. v. Flow-chart for the Dual-Latch LSSD Register without Scan.

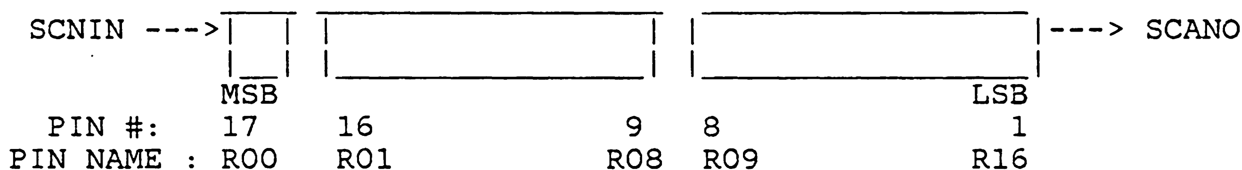
Appendix B

# GSP MODEL DESCRIPTIONS FOR THE AMAC UNIT

## 1. REGISTER WITH PARITY-CHECKER

[ SOURCE CODE FOR THE 17-BIT REGISTER WITH PARITY-CHECKER ]

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




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

;  REGISTERS FOR THE LSSD REGISTER
REG(8)  OLDAL,OLDAH,L1LO,L1HI,SRLL,SRLH
REG(1)  OD17,L117,L217,SRL17
REG(1)  ERRFL,OSCLK,OLSCN,OACLK
;
; REGISTERS FOR THE PARITY CHECKER
REG(3)  DL,DM,DH,DL2,DM2,DH2
REG(3)  ODL,ODM,ODH,ODL2,ODM2,ODH2
REG(3)  DOUL,DOUM,DOUH,DOUL2,DOUM2,DOUH2
REG(3)  DMID,TEMP,DMID2,TEMP2
REG(1)  OLP1,PARY1,PARY2
;
PIN     DATL(1,8),DATH(9,16),D17(17),SCLK(18),BCLK(19)
PIN     L2LO(20,27),L2HI(28,35)
PIN     ACLK(36),SCNOP(37),SCNIN(38),SCANO(39)
PIN     PARD8(40),ODPAR(41)
PIN     SC1(151),SC2(152)
;
EVW     W1(1),W20(20),W21(21),W25(25)
EVW     W33(33)
;
; DESCRIPTION FOR THE LSSD REGISTER
;
        MOV  #0,ERRFL              ; CLEAR ERROR-FLAG
        BEQ  #0,SCNOP,CHK1         ; for Latch-operation.
        BRU  CHK3                  ; for SCAN-operation and SRL.
;
;
CHK1:   BNE  DATL,OLDAL,YESIN      ; CHANGE IN LOW BYTE OF DATA ?
```

```
        BNE   DATH,OLDAH,YESIN        ; CHANGE IN HIGH BYTE OF DATA?
        BNE   D17,OD17,YESIN          ; CHANGE IN MSB OF DATA ?
CHK2:   BNE   SCLK,OSCLK,CLKIN        ; CLOCK CHANGE ?
        EXR
;
YESIN:  MOV   DATL,OLDAL    ; STORE LOW BYTE FOR SUBSEQUENT CHECK.
        MOV   DATH,OLDAH    ; STORE HI BYTE FOR SUBSEQUENT CHECKS.
        MOV   D17,OD17      ;
        BEQ   #0,SCLK,CHK2  ;  BRANCH IS SYS-CLK = LOW
        MOV(W20) #1,SC1     ;  SCHEDULE SELF-CALL AFTER 20 NS. TO
        EXC                 ;  INCORPORATE SETUP DELAY FOR DATA
;                                SINCE SYS-CLK IS HIGH.
        MOV   SCLK,OSCLK      ;  STORE SYS-CLK VALUE.
        BEQ   #0,SC1,NOSC1    ;  BRANCH IF PROC. CALL
;                                  NOT DUE TO SYS-CLK.
        BEQ   #0,SCLK,HOLD1   ;  CALL DUE TO SYS-CLK.
;                                  SIMULATE HOLD-TIME
        MOV   #0,SC1          ;  RESET SELF-CALL PIN.
        MOV   OLDAL,SRLL      ; LATCH LOW BYTE WITH NO HOLD-TIME.
        MOV   OLDAH,SRLH      ; LATCH HI BYTE WITH NO HOLD TIME.
        MOV   OD17,SRL17      ;  LATCH MSB.
        MOV   SRLL,L1LO       ;  LOW AND HIGH BYTES ...
        MOV   SRLH,L1HI       ;  .... APPEAR AT THE OUTPUT.
        MOV   SRL17,L117      ;
        BRU   CHK2
;
; MODEL HOLD-TIME HERE
;
HOLD1:  MOV   SCLK,OSCLK      ; SYSCLK HAS CHANGED.NOTE THE CHANGE.
        MOV   #0,SC1          ; RESET SELF-CALL PIN.
        MOV   OLDAL,SRLL      ;
        MOV   OLDAH,SRLH
        MOV   OD17,SRL17
        MOV(W1) SRLL,L1LO     ; LATCH THE LOW AND ...
        MOV(W1) SRLH,L1HI     ; ... BYTES.
        MOV(W1) SRL17,L117    ;
        MOV(W21) SRLL,L2LO    ; SIMULATE B-CLK W.R.T. SYS-CLK, AND
        MOV(W21) SRLH,L2HI    ; MOV DATA TO LATCH OUTPUT IN 21 NS.
        MOV(W21) SRL17,L217 ;
        EXR
;
; PROC. CALL WAS NOT DUE TO SELF-CALL
;
NOSC1:  CAN   SC1             ; CANCEL THE SCEDULED SALF-CALL.
        BRU   CHK1            ; START AGAIN
;
CLKIN:  MOV   SCLK,OSCLK      ; STORE NEW VALUE OF C-CLK
        BEQ   #0,SCLK,PROP    ; IF C-CLK = 0. IT CHANGED FROM 1 TO
                              ; 0...HENCE BRANCH TO PROPAGATE
        MOV(W20) #1,SC2       ; C-CLK = 1. IT CHANGED FROM 0 TO 1.
```

```
        EXC                     ; SCHED. CALL IN 20 NS. AND EXIT.
;
        MOV  SCLK,OSCLK         ; STORE C-CLK AT PROCEDURE CALL.
        BEQ  #0,SC2,NOSC2       ; IF SC2 = 0, CALL NOT DUE TO SC2.
        BEQ  #0,SCLK,HOLD2      ; SC2 =1. SCLK =0, IT CHANGED FROM
                                ; 1 TO 0, MODEL HOLD TIME.
        MOV  #0,SC2               ; RESET SELF CALL PIN-VALUE.
        MOV  OLDAL,SRLL           ; MOVE DATA TO SHIFT-REGISTERS.
        MOV  OLDAH,SRLH           ;
        MOV  OD17,SRL17           ;
        MOV  SRLL,L1LO            ; LATCH INPUT DATA.
        MOV  SRLH,L1HI            ;
        MOV  SRL17,L117           ; LATCH BIT-17
        EXR
;
HOLD2:  MOV  SCLK,OSCLK           ; SYSCLK CHANGED. NOTE THE CHANGE.
        MOV  #0,SC2               ; RESET SC2 PIN-VALUE
        MOV  OLDAL,SRLL           ; MOVE DATA INTO SHIFT REGISTERS
        MOV  OLDAH,SRLH           ;
        MOV  OD17,SRL17           ;
        MOV(W1) SRLL,L1LO         ; LATCH DATA AFTER HOLD TIME (1NS.)
        MOV(W1) SRLH,L1HI         ;
        MOV(W1) SRL17,L117        ;
        EXR
;
NOSC2:  CAN  SC2                  ; CANCEL SELF-CALL  'SC2'
        BRU  CHK1                 ;
;
PROP:   MOV(W20) SRLL,L2LO        ; MOVE DATA TO OUTPUT LATCH IN
        MOV(W20) SRLH,L2HI        ; PROPAGATION TIME (20 NS.)
        MOV(W20) SRL17,L217       ;
        EXR
;
;  Scan operation begins here
;
CHK3:    BNE  SCNIN,OLSCN,SCAN     ; CHECK FOR CHANGE IN SCAN DATA
CHK4:    BNE  ACLK,OACLK,SCNCL     ; OR A-CLK.
         EXR
;
SCAN:    MOV  SCNIN,OLSCN          ; STORE NEW SCANIN VALUE.
         BEQ  #0,ACLK,CHK4         ; BRANCH IF A-CLK NOT HIGH.
         MOV(W20) #1,SC1           ; A-CLK = 1. SCHED. SELF-CALL IN
         EXC                       ; 20 NS. EXIT UNTIL CALL.
;
         MOV  ACLK,OACLK         ; STORE A-CLK VALUE AT PROC. CALL
         BEQ #0,SC1,NOCAL        ; BRANCH IF CALL NOT DUE TO SC1.
         BEQ #0,ACLK,HOLT1       ; ACLK HAS CHANGED FROM 1 TO 0, TOO.
         MOV #0,SC1              ; RESET SELF-CALL PIN-VALUE
         BRU CHK4                ;
;
```

```
HOLT1:    MOV #0,SC1
;
SHIFT:    SHR SRLH               ; SHIFT RIGHT HIGH BYT(BITS 16-9)
          BEQ SRL17,BIT17        ; IS MSB (BIT 17) = 1 ?
          BIS #7,SRLH            ; IF YES, SET BIT-16.
BIT17:    BEQ OLSCN,RSET17       ; SCANIN = 0 ? ...
          MOV #1,SRL17           ; .. NO. SET BIT - 17.
RSET17:   MOV #0,SRL17           ; ..YES. RESET BIT - 17.
          BEQ C,NOTHI            ; IF BIT 9 WAS A 1,..
          SHR SRLL               ;.. SHIFT RIGHT LO BYT(BITS 8-1)
          BIS #7,SRLL            ; AND SET BIT 8.
          BRU NXT                ;
NOTHI:    SHR SRLL               ; BIT 9 WAS A 0, SHIFT RIGHT (MOVES
                                 ; 0 INTO BIT 8)
NXT:      MOV(W25) SRLL,L2LO     ; MOVE DATA TO OUTPUT AFTER 25NS.
          MOV(W25) SRLH,L2HI
          MOV(W25) SRL17,L217 ; SRL17 HAS THE NEW MSB.
          MOV(W25) C,SCANO       ; 'C' HAS OLD BIT-1 FROM LAST 'SHR'.
          EXR                    ; EXIT AND RESTART
;
NOCAL:    CAN SC1                ; PROC. CALL DUE TO WRONG
          BRU CHK3               ; PIN-VALUE CHANGE.
;
SCNCL:    MOV  ACLK,OACLK        ; STORE NEW VALUE OF A-CLK.
          BEQ  #0,ACLK,SHIFT     ; IF A=0, IT CHANGED FROM 1 TO 0.
                                 ; HENCE, JUMP TO SHIFT.
          MOV(W20) #1,SC2        ; IF A=1, IT CHANGED FROM 0 TO 1.
          EXC                    ; SCHED. SELF-CALL AFTER 20 NS. AND
;                                     EXIT.
          MOV  ACLK,OACLK        ; STORE VALUE OF A-CLK AT PRC. CALL.
          BEQ  #1,SC2,SELF2      ; IS PROC. CALL DUE TO SC2 ?
          CAN  SC2               ; NO.  CANCEL ALL EVENTS OF SC2 .
          BRU  CHK3
;
SELF2:    BEQ  #0,ACLK,HOLT2     ; PRC. CALL DUE TO SC2.IS A-CLK =0 ?
          MOV  #0,SC2            ; A-CLK = 1. RESET 'SC2'.
          EXR                    ; EXIT AND RESTART.
;
HOLT2:    MOV  #0,SC2            ; HOLD TIME. RESET SC2 PIN-VALUE.
          BRU SHIFT              ;
===========================================
  PREPARE REGISTERS FOR PARITY CHECKERS
===========================================
          IDX L2LO(0),3,1          ; BITS-1,2,3   IN DL
          MOV @1,DL                ;
          IDX L2LO(3),3,2          ; BITS-4,5,6   IN DM
          MOV @2,DM                ;
          IDX L2LO(6),2,3          ; BITS-7,8     IN DH
          MOV @3,DH                ;
          IDX L2HI(0),1,4          ; BIT-9        IN DH
```

```
        BEQ #0,@4,SKIP          ;
        BIS #2,DH               ;
;
SKIP:   IDX L2HI(1),2,1      ; BITS-10,11 IN DL2
        MOV @3,DL2
        SHL DL2              ; DL2(1,2)= BITS-10,11 RESPECTIV.
        BEQ PARD8,JUMP       ; PI-'PARD8' IS THE INPUT TO LSB
        BIS #0,DL2           ; DL2(0) = PARD8 = 1
JUMP:   IDX L2HI(3),3,2      ; BITS-12,13,14 IN DM2
        MOV @2,DM2           ;
        IDX L2HI(6),2,3      ; BITS-15,16 IN DH2
        MOV @3,DH2           ;
        BEQ L217,PARCK       ; IF BIT-17 = 0, GOTO PARITY CHECK.
        BIS #2,DH2           ; IF BIT-17 = 1, SET DH2(2).
;
;
;****************************
; CODE FOR PARITY CHECK BLOCK
;****************************
;
;CODE FOR FIRST 9-BIT ODD PARITY GENERATOR
;
PARCK:  MOV    PARY1,OLP1      ; STORE PARITY FOR LO BYT(R16,R09)
        BNE    DL,ODL,CHKDL    ; CHECK FOR CHANGE IN LO 9-BITS...
        BNE    DM,ODM,CHKDM    ; .. OF DATA
        BNE    DH,ODH,CHKDH    ;
NXTPR:  BNE    DL2,ODL2,CKDL2  ; CHECK FOR CHANGE IN HI 9-BITS...
        BNE    DM2,ODM2,CKDM2  ; .. OF DATA
        BNE    DH2,ODH2,CKDH2  ;
;
        BEQ  OLP1,PARY1,QUIT    ; BRANCH IF NO CHANGE IN PARY1
        XOR  PARY1,PARY2,PARY2  ; LO BYT PARITY CHANGED, FIND NEW
        MOV(W33) PARY2,ODPAR    ; ..ODD-PAR. MOV TO PIN IN 33 NS.
        EXR                     ; EXIT
;
CHKDL:  MOV    DL,ODL          ; STORE NEW VALUE OF LO 3-BITS.
        MOV    DL,TEMP         ; USE TEMP TO...
        JSR    FUNC2           ; ..DECODE AND OBTAIN 3-BITS' PARITY
        MOV    DMID,DOUL       ; STORE PARITY INTO 3-BIT BUFFER
        BEQ    DM,ODM,CHKDH    ; IF NO CHANGE IN MID 3-BITS,BRANCH.
CHKDM:  MOV    DM,ODM          ; STORE NEW VALUE OF MID 3-BITS.
        MOV    DM,TEMP           ; OBTAIN PARITY FOR MID 3-BITS.
        JSR    FUNC2             ;
        MOV    DMID,DOUM         ; MOVE PARITY INTO 3-BIT BUFFER
        ROR    DOUM              ; GET IT INTO CENTER-BIT OF..
        ROR    DOUM              ;  .... THE BUFFER
CHKDH:  BEQ    DH,ODH,CONT       ; CHANGE IN HI 3-BITS ?
        MOV    DH,ODH            ; STORE NEW VALUE
        MOV    DH,TEMP           ; OBTAIN PARITY FOR HI 3-BITS.
        JSR    FUNC1             ;
```

```
         MOV     DMID,DOUH          ; MOVE PARITY TO 3-BIT BUFFER
         ROR     DOUH               ;
CONT:    OR      DOUL,DOUM,TEMP     ; GET THE THREE 3-BIT PARITY'S
         OR      DOUH,TEMP          ; INTO TEMP.
         JSR     FUNC2              ; GET FINAL PARITY FOR LO 9-BITS
         IDX     DMID(0),1,3        ;
         MOV     @3,PARY1           ; LO 9-BIT PARITY.
         BRU     NXTPR              ; BRANCH. FIND HI 9-BIT PARITY
;
FUNC1: IDX    TEMP(0),3,1           ; DECODE TEMP TO FIND 3-BITS' ODD
       MOV    PAR1@1,DMID           ; PARITY FROM THE TRUTH TABLE.
       RTS
;
FUNC2: IDX    TEMP(0),3,2           ; DECODE TO FIND 3-BITS' EVEN
       MOV    PAR2@2,DMID           ; PARITY FROM THE TABLE.
       RTS
;
;CODE FOR SECOND 9-BIT ODD PARITY GENERATOR (PARGEN 2)
;
CKDL2:   MOV     DL2,ODL2           ; STORE NEW VALUE FOR LO 3-BITS
         MOV     DL2,TEMP2          ; FIND 3-BITS' EVEN PARITY
         JSR     FUN22              ;
         MOV     DMID2,DOUL2        ; MOV PARITY INTO 3-BIT BUFR (B0)
         BEQ     DM2,ODM2,CKDH2     ; CHANGE IN MIDDLE 3-BITS ?
CKDM2:   MOV     DM2,ODM2           ; STORE NEW VALUE
         MOV     DM2,TEMP2          ; OBTAIN 3-BITS' EVEN PARITY
         JSR     FUN22              ;
         MOV     DMID2,DOUM2        ; STORE PARITY IN BUFFER (B1)
         ROR     DOUM2              ;
         ROR     DOUM2              ; B1 = BIT #1 OF BUFFER
CKDH2:   BEQ     DH2,ODH2,CONT2     ; CHANGE IN HI 3-BITS ?
         MOV     DH2,ODH2           ; STORE NEW  VALUE
         MOV     DH2,TEMP2          ; GET ODD PARITY FOR THE 3-BITS
         JSR     FUN21              ;
         MOV     DMID2,DOUH2           ; STORE PARITY IN BUFFER (B2)
         ROR     DOUH2                 ;  B2 = BIT #2 OF BUFFER
CONT2:   OR      DOUL2,DOUM2,TEMP2     ; OBTAIN THE THREE PARITY BITS
         OR      DOUH2,TEMP2           ; INTO TEMP2
         JSR     FUN22                 ;
         IDX     DMID2(0),1,3          ; GET FINAL PARITY FOR HI ...
         MOV     @3,PARY2              ;  ....9-BITS.
         XOR     PARY1,PARY2,PARY2     ; OVERALL 18-BIT ODD PARITY .
         MOV(W33) PARY2,ODPAR          ; MOVE TO THE PIN IN 33 NS.
QUIT:    EXR
;
;
FUN21: IDX    TEMP2(0),3,1          ; DECODE TEMP2 TO OBTAIN 3-BITS'
       MOV    PAR1@1,DMID2          ; ODD PARITY.
       RTS
;
```

```
FUN22: IDX     TEMP2(0),3,2        ; DECODE TEMP2 TO OBTAIN 3-BITS'
       MOV     PAR2@2,DMID2        ; EVEN PARITY.
       RTS
;                                            TRUTH-TABLES :
PAR1: BYT #1, #0, #0, #1, #0, #1, #1, #0    ; ODD PARITY TABLE
PAR2: BYT #0, #1, #1, #0, #1, #0, #0, #1    ; EVEN PARITY TABLE
END
```

## 2. ADDER.SOR

[SOURCE CODE FOR THE ADDER UNIT]

```
; REGISTERS
REG(1)     CNREG,OPREG,CMDRG
REG(4)     OLDS
REG(8)     OLDAL,OLDAH,OLDBL,OLDBH,BUF1,BUF2,BUF3,BUF4
;
PIN       DATAL(1,8),DATAH(9,16),DATBL(17,24),DATBH(25,32)
PIN       CNBAR(33),OPER(34),CMND(35),ADDL(36,43),ADDH(44,51)
PIN       SOS3(151,154)
;
; DELAY W.R.T. TIME OF INPUT DATA ARRIVAL
;
EVW       W52(52)
;
          BNE      DATAL,OLDAL,PROC    ; PROCEED TO PROCESS IF ANY
          BNE      DATAH,OLDAH,PROC    ; OF THESE PINS CHANGE THEIR
          BNE      DATBL,OLDBL,PROC    ; VALUES.
          BNE      DATBH,OLDBH,PROC
          BNE      SOS3,OLDS,PROC
          EXR
;
PROC:  MOV   CNBAR,CNREG       ; STORE THE NEW VALUES IN REGISTERS
       MOV   DATAL,OLDAL       ; TO CHECK FOR CHANGE NEXT TIME.
       MOV   DATAH,OLDAH
       MOV   DATBL,OLDBL
       MOV   DATBH,OLDBH
       MOV   SOS3,OLDS
;
;=====================================================================
;  THE FOUR SEPARATE ALU UNITS, PLUS A CLA CIRCUIT HAVE BEEN
;   COMBINED INTO A SINGLE MODEL HERE. THE CARRIES, IF ANY,
;   ARE TAKEN CARE OF AUTOMATICALLY WITHOUT THE INCLUSION
;                 OF AN EXTRA MODEL FOR CLA.
;=====================================================================
       BEQ   #1,CNBAR,NOCAR
CARRY: IDX   OLDS(0),4,1                 ;  TABLE FOR OPERATIONS
       BRU   ALUCR@1                     ;  WITH INITIAL CARRY
;
ALUCR: BYT   200,201,202,203,204,205,206,207
       BYT   208,209,210,211,212,213,214,215
;-----------------------------------------
NOCAR: IDX   OLDS(0),4,1                 ;  TABLE FOR OPERATIONS
       BRU   ALUBR@1                     ;  WITHOUT INITIAL CARRY
;
ALUBR: BYT   100,101,102,103,104,105,106,107
       BYT   108,109,110,111,112,113,114,115
```

```
;
;
105:  MOV(W52) OLDAL,ADDL           ;   F = A
      MOV(W52) OLDAH,ADDH
      EXR
;
104:  OR   OLDAL,OLDBL,BUF1         ;  F = A+B
      OR   OLDAH,OLDBH,BUF2
      MOV(W52) BUF1,ADDL
      MOV(W52) BUF2,ADDH
      EXR
;
107:  MOV  OLDBL,BUF1               ;   F = A + B'S COMPLEMENT
      MOV  OLDBH,BUF2
      COM  BUF1
      COM  BUF2
      OR   OLDAL,BUF1,BUF3
      OR   OLDAH,BUF2,BUF4
      MOV(W52) BUF3,ADDL
      MOV(W52) BUF4,ADDH
      EXR
;
106:   MOV(W52)  #255,ADDL          ;  F = MINUS 1 (2'S COMPLEMENT)
       MOV(W52)  #255,ADDH
       EXR
;
101:   MOV  OLDBL,BUF1              ;  F = A PLUS (A.¢B)
       MOV  OLDBH,BUF2
       COM  BUF1
       COM  BUF2
       AND  OLDAL,BUF1,BUF1
       AND  OLDAH,BUF2,BUF2
       ADD  BUF1,OLDAL,BUF3
       BEQ  #0,C,NOPL1
       ADD  #1,BUF2,BUF2
NOPL1: ADD  BUF2,OLDAH,BUF4
       MOV(W52)  BUF3,ADDL
       MOV(W52)  BUF4,ADDH
       EXR
;
100:  OR   OLDAL,OLDBL,BUF1         ;  F = (A + B) PLUS (A.¢B)
      OR   OLDAH,OLDBH,BUF2
      MOV  OLDBL,BUF3
      MOV  OLDBH,BUF4
      COM  BUF3                     ;  ¢BL
      COM  BUF4                     ;  ¢BH
      AND  OLDAL,BUF3,BUF3          ;  (AL . ¢BL)
      AND  OLDAH,BUF4,BUF4          ;  (AH . ¢BH)
      ADD  BUF1,BUF3,BUF3
      BEQ  #0,C,MORE
```

```
      ADD  #1,BUF4,BUF4
MORE: ADD  BUF2,BUF4,BUF4         ;  (AH + BH) PLUS (AH.¢BH)
      MOV(W52)  BUF3,ADDL
      MOV(W52)  BUF4,ADDH
      EXR
;
103:   SUB OLDAL,OLDBL,BUF1   ;  F = A MINUS B MINUS 1. (AL - BL)
       BEQ #0,C,MINUS         ; BRANCH IF NO BORROW OCCURS
       MOV #1,BUF2            ; BORROW OCCURRED...
       SUB OLDAH,BUF2,BUF2    ; .. SUBTRACT 1 FROM HIBYT
MINUS: MOV #1,BUF3
       SUB BUF1,BUF3,BUF1     ; MINUS 1
       BEQ #0,C,MIN2          ; BRANCH IF NO BORROW.
       SUB BUF2,BUF3,BUF2     ; BORROW. HENCE,BUF2 = UPDATED HIBYT
MIN2:  SUB BUF2,OLDBH,BUF2    ; BUF2 - OLDBH
       MOV(W52) BUF1,ADDL
       MOV(W52) BUF2,ADDH
       EXR
;
102:  MOV OLDBL,BUF1              ; F = (A.¢B) MINUS 1
      MOV OLDBH,BUF2
      COM BUF1                    ; ¢BL
      COM BUF2                    ; ¢BH
      AND OLDAL,BUF1,BUF3         ; (AL .¢BL)
      AND OLDAH,BUF2,BUF4         ; (AH .¢BH)
      MOV #1,BUF1
      SUB BUF3,BUF1,BUF3      ; SUBTRACT 1 FROM LOWER BYTE
      BEQ #0,C,QUIT           ; IF NO BORROW, QUIT
      SUB BUF4,BUF1,BUF4      ; BORROW. HENCE,SUBTRACT 1 FROM HIBYT
QUIT: MOV(W52) BUF3,ADDL
      MOV(W52) BUF4,ADDH
      EXR
;
113:  AND OLDAL,OLDBL,BUF1       ;  F = A PLUS (A.B)
      AND OLDAH,OLDBH,BUF2
      ADD OLDAL,BUF1,BUF3
      BEQ #0,C,NXT               ; BRANCH IF NO CARRY
      MOV #1,BUF4                ; 1 IS ADDED TO AH SUBSEQUENTLY.
      ADD BUF2,BUF4,BUF2
NXT:  ADD OLDAH,BUF2,BUF4        ; HIGHER BYTE
      MOV(W52) BUF3,ADDL
      MOV(W52) BUF4,ADDH
;
112:  ADD OLDAL,OLDBL,BUF1       ; F = A PLUS B
      BEQ #0,C,HIGH              ; BRANCH IF NO CARRY
      ADD #1,OLDAH,BUF2          ; ADD 1 TO ..
      ADD OLDBH,BUF2,BUF3        ; .. HIBYT AND OUTPUT.
      BRU OUT
HIGH: ADD OLDAH,OLDBH,BUF3
OUT:  MOV(W52) BUF1,ADDL
```

```
      MOV(W52) BUF3,ADDH
      EXR
;
115:    AND OLDAL,OLDBL,BUF1      ; F = (A+¢B) PLUS (A.B)
        AND OLDAH,OLDBH,BUF2
        MOV OLDBL,BUF3
        MOV OLDBH,BUF4
        COM BUF3                  ; ¢BL
        COM BUF4                  ; ¢BH
        OR  OLDAL,BUF3,BUF3       ; (AL + ¢BL)
        OR  OLDAH,BUF4,BUF4       ; (AH + ¢BH)
        ADD BUF1,BUF3,BUF3
        BEQ #0,C,HIBYT
        ADD #1,BUF2,BUF2
HIBYT:  ADD BUF2,BUF4,BUF4
        MOV(W52) BUF3,ADDL
        MOV(W52) BUF4,ADDH
        EXR
;
114:    AND OLDAL,OLDBL,BUF1         ; F = (A.B) MINUS 1
        AND OLDAH,OLDBH,BUF2
        MOV #1,BUF3
        SUB BUF1,BUF3,BUF1           ; (AL.BL) - 1
        BEQ #0,C,NOBOR               ; BRANCH IF NO BORROW
        SUB BUF2,BUF3,BUF2           ; BORROW OCCURRED
NOBOR:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
109:    MOV OLDAL,BUF1          ; F = A PLUS A
        MOV OLDAH,BUF2
        ADD OLDAL,BUF1,BUF1     ; (AL + AL)
        BEQ #0,C,UPBYT
        ADD #1,BUF2,BUF2        ; CARRY OCCURRED.BUF2 = (AH PLUS 1)
UPBYT:  ADD OLDAH,BUF2,BUF2     ; (AH + BUF2)
        MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
108:    OR  OLDAL,OLDBL,BUF1    ; F = (A+B) PLUS A
        OR  OLDAH,OLDBH,BUF2
        ADD OLDAL,BUF1,BUF1     ; (AL + BL) PLUS AL
        BEQ #0,C,NXBYT          ; BRANCH IF NO CARRY.
        ADD #1,BUF2,BUF2        ; CARRY.HENCE,BUF2=[(AH.BH) PLUS 1]
NXBYT:  ADD OLDAH,BUF2,BUF2     ; BUF2 = [(AH.BH) PLUS AH]
        MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
111:    MOV OLDBL,BUF1               ; F = (A+¢B) PLUS A
```

```
        MOV OLDBH,BUF2
        COM BUF1                    ; ¢BL
        COM BUF2                    ; ¢BH
        OR  OLDAL,BUF1,BUF3         ; (AL + ¢BL)
        OR  OLDAH,BUF2,BUF4         ; (AH + ¢BH)
        ADD OLDAL,BUF3,BUF3         ; BUF3 = AL PLUS (AL + ¢BL)
        BEQ #0,C,OTHER
        ADD #1,BUF4,BUF4            ; CARRY.BUF4=(AH + ¢BH) PLUS 1
OTHER:  ADD OLDAH,BUF4,BUF4         ; HIBYT
        MOV(W52) BUF3,ADDL
        MOV(W52) BUF4,ADDH
        EXR
;
110:    MOV OLDAH,BUF2              ; F = A MINUS 1
        MOV #1,BUF3
        SUB OLDAL,BUF3,BUF1         ; AL-1
        BEQ #0,C,BORNO
        SUB BUF2,BUF3,BUF2          ; BORROW.SUB 1 FROM HIBYT(AH).
BORNO:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
;=================================================================
;     OPERATIONS WITH INITIAL CARRY ARE PERFORMED HERE.
;  THIS INITIAL CARRY IS DETERMINED BY OPER, CMND(11) AND A(15).
;=================================================================
205:    MOV OLDAH,BUF2           ;   F = A PLUS 1
        ADD #1,OLDAL,BUF1
        BEQ #0,C,NOAD1
        ADD #1,BUF2,BUF2
NOAD1:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
204:    OR   OLDAL,OLDBL,BUF1       ;   F = (A+B) PLUS 1
        OR   OLDAH,OLDBH,BUF2
        ADD  #1,BUF1,BUF1
        BEQ  #0,C,NOCRY
        ADD  #1,BUF2,BUF2
NOCRY:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
207:  MOV  OLDBL,BUF1               ;   F = (A + ¢B) PLUS 1
      MOV  OLDBH,BUF2
      COM  BUF1
      COM  BUF2
      OR   OLDAL,BUF1,BUF3
      OR   OLDAH,BUF2,BUF4
      ADD  #1,BUF3,BUF3
```

```
        BEQ   #0,C,NOC
        ADD   #1,BUF4,BUF4
NOC:    MOV(W52) BUF3,ADDL
        MOV(W52) BUF4,ADDH
        EXR
;
206:     MOV(W52)  #0,ADDL          ;  F = ZERO
         MOV(W52)  #0,ADDH
         EXR
;
201:     MOV   OLDBL,BUF1           ;  F = A PLUS (A.¢B) PLUS 1 .
         MOV   OLDBH,BUF2
         COM   BUF1
         COM   BUF2
         AND   OLDAL,BUF1,BUF1
         AND   OLDAH,BUF2,BUF2
         ADD   BUF1,OLDAL,BUF3
         BEQ   #0,C,NOCRR
         ADD   #1,BUF2,BUF2
NOCRR:   ADD   BUF2,OLDAH,BUF4
         ADD   #1,BUF3,BUF3
         BEQ   #0,C,OUTPT
         ADD   #1,BUF4,BUF4         ; CARRY OCCURRED
OUTPT:   MOV(W52)  BUF3,ADDL
         MOV(W52)  BUF4,ADDH
         EXR
;
200:    OR    OLDAL,OLDBL,BUF1      ; F = (A + B) PLUS (A.¢B) PLUS 1.
        OR    OLDAH,OLDBH,BUF2
        MOV   OLDBL,BUF3
        MOV   OLDBH,BUF4
        COM   BUF3                  ; ¢BL
        COM   BUF4                  ; ¢BH
        AND   OLDAL,BUF3,BUF3       ; (AL . ¢BL)
        AND   OLDAH,BUF4,BUF4       ; (AH . ¢BH)
        ADD   BUF1,BUF3,BUF3
        BEQ   #0,C,MOR
        ADD   #1,BUF4,BUF4
MOR:    ADD   BUF2,BUF4,BUF4        ; (AH + BH) PLUS (AH.¢BH)
        ADD   #1,BUF3,BUF3
        BEQ   #0,C,THRO
        ADD   #1,BUF4,BUF4
THRO:   MOV(W52)  BUF3,ADDL
        MOV(W52)  BUF4,ADDH
        EXR
;
203:     SUB OLDAL,OLDBL,BUF1        ;  F = A MINUS B .  (AL - BL)
         BEQ #0,C,MINIS              ; BRANCH IF NO BORROW OCCURS
         MOV #1,BUF2                 ; BORROW OCCURRED...
         SUB OLDAH,BUF2,BUF2         ; .. SUBTRACT 1 FROM HIBYT
```

```
MINIS: SUB BUF2,OLDBH,BUF2         ; BUF2 - OLDBH
       MOV(W52) BUF1,ADDL
       MOV(W52) BUF2,ADDH
       EXR
;
202:   MOV OLDBL,BUF1              ; F = (A.¢B)
       MOV OLDBH,BUF2
       COM BUF1                    ; ¢BL
       COM BUF2                    ; ¢BH
       AND OLDAL,BUF1,BUF3         ; (AL . ¢BL)
       AND OLDAH,BUF2,BUF4         ; (AH . ¢BH)
       MOV(W52) BUF3,ADDL
       MOV(W52) BUF4,ADDH
       EXR
;
213:   AND OLDAL,OLDBL,BUF1        ;  F = A PLUS (A.B) PLUS 1
       AND OLDAH,OLDBH,BUF2
       ADD OLDAL,BUF1,BUF3
       BEQ #0,C,NXT2               ; BRANCH IF NO CARRY
       MOV #1,BUF4                 ; 1 IS ADDED TO AH SUBSEQUENTLY.
       ADD BUF2,BUF4,BUF2
NXT2:  ADD OLDAH,BUF2,BUF4         ; HIGHER BYTE
       ADD #1,BUF3,BUF3
       BEQ #0,C,CZRO
       ADD #1,BUF4,BUF4
CZRO:  MOV(W52) BUF3,ADDL
       MOV(W52) BUF4,ADDH
;
212:   ADD OLDAL,OLDBL,BUF1        ; F = A PLUS B PLUS 1
       BEQ #0,C,HIH2               ; BRANCH IF NO CARRY
       ADD #1,OLDAH,BUF2           ; ADD 1 TO
       ADD OLDBH,BUF2,BUF3         ;   HIBYT AND OUTPUT.
       BRU ADD1
HIH2:  ADD OLDAH,OLDBH,BUF3
ADD1:  ADD #1,BUF1,BUF1
       BEQ #0,C,OUT2
       ADD #1,BUF3,BUF3
OUT2:  MOV(W52) BUF1,ADDL
       MOV(W52) BUF3,ADDH
       EXR
;
215:     AND OLDAL,OLDBL,BUF1        ; F = (A+¢B) PLUS (A.B) PLUS 1.
         AND OLDAH,OLDBH,BUF2
         MOV OLDBL,BUF3
         MOV OLDBH,BUF4
         COM BUF3                    ; ¢BL
         COM BUF4                    ; ¢BH
         OR  OLDAL,BUF3,BUF3         ; (AL + ¢BL)
         OR  OLDAH,BUF4,BUF4         ; (AH + ¢BH)
         ADD BUF1,BUF3,BUF3
```

```
        BEQ #0,C,HIBY2
        ADD #1,BUF2,BUF2
HIBY2:  ADD BUF2,BUF4,BUF4
        ADD #1,BUF3,BUF3
        BEQ #0,C,EXZI                 ; NO CARRY, THEN BRANCH .
        ADD #1,BUF4,BUF4
EXZI:   MOV(W52) BUF3,ADDL
        MOV(W52) BUF4,ADDH
        EXR
;
214:    AND OLDAL,OLDBL,BUF1          ; F = (A.B)
        AND OLDAH,OLDBH,BUF2
        MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
209:    MOV OLDAL,BUF1                ; F = A PLUS A PLUS 1.
        MOV OLDAH,BUF2
        ADD OLDAL,BUF1,BUF1           ; (AL + AL)
        BEQ #0,C,UPBY2
        ADD #1,BUF2,BUF2              ; CARRY. BUF2 = (AH PLUS 1)
UPBY2:  ADD OLDAH,BUF2,BUF2           ; (AH + BUF2)
        ADD #1,BUF1,BUF1
        BEQ #0,C,CARNO
        ADD #1,BUF2,BUF2              ; CARRY OCCURRED
CARNO:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
208:    OR  OLDAL,OLDBL,BUF1          ; F = (A+B) PLUS A PLUS 1
        OR  OLDAH,OLDBH,BUF2
        ADD OLDAL,BUF1,BUF1           ; (AL + BL) PLUS AL
        BEQ #0,C,NXBY2
        ADD #1,BUF2,BUF2              ; CARRY OCCURRED...
                                      ;  ...BUF2= [(AH.BH) PLUS 1]
NXBY2:  ADD OLDAH,BUF2,BUF2           ; BUF2 = [(AH.BH) PLUS AH]
        ADD #1,BUF1,BUF1
        BEQ #0,C,RESUL
        ADD #1,BUF2,BUF2
RESUL:  MOV(W52) BUF1,ADDL
        MOV(W52) BUF2,ADDH
        EXR
;
211:    MOV OLDBL,BUF1                ; F = (A+¢B) PLUS A PLUS 1
        MOV OLDBH,BUF2
        COM BUF1                      ; ¢BL
        COM BUF2                      ; ¢BH
        OR  OLDAL,BUF1,BUF3           ; (AL + ¢BL)
        OR  OLDAH,BUF2,BUF4           ; (AH + ¢BH)
        ADD OLDAL,BUF3,BUF3           ; BUF3 = AL PLUS (AL + ¢BL)
```

```
        BEQ #0,C,OTHR
        ADD #1,BUF4,BUF4            ; CARRY.BUF4=(AH + ¢BH) PLUS 1
OTHR:   ADD OLDAH,BUF4,BUF4         ; HIBYT
        ADD #1,BUF3,BUF3
        BEQ #0,C,OPHIL
        ADD #1,BUF4,BUF4
OPHIL:  MOV(W52) BUF3,ADDL
        MOV(W52) BUF4,ADDH
        EXR
;
210:    MOV(W52) OLDAL,ADDL          ; F = A
        MOV(W52) OLDAH,ADDH
        EXR
END
```

3. 8X8MULT.SOR

[ SOURCE CODE FOR THE 8X8 MULTIPLIER UNIT ]

```
;REGISTERS FOR THE MODEL
REG(8)  OLX,OLY,REG0,REG1,REG2,REG3,REG4,REG5,REG6,REG7
REG(7)  REGA,REGB,BFOR,BFAND,ZREG
REG(5)  P4TO8
REG(4)  REGAL,REGBL,BFORL,BFANL,BUF4
REG(3)    DC03R,FLG2
REG(2)  SIGN,DC02R,TEMP2
REG(1)    FSUM3,FCAR3,FSUM2,FCAR2
REG(1)    BIT1,BIT2,BIT3,FLAG,C4,P10R
REG(1)  F32C,F2ZS,F3C,F3S,F5C,F5S,F8C,F8S,F9C,F9S
REG(1)  F10C,F10S,F12C,F12S
REG(1)  F13C,F13S,F16C,F16S,F17C,F17S,F18C,F18S
REG(1)  F19C,F19S,F20C,F20S,F21C,F21S,F22C,F22S
REG(1)  F24C,F24S,F25C,F25S,F26C,F26S,F29C,F29S,F31C,F31S
REG(1)    G1C,G1S,G2C,G2S,G3C,G3S,G4C,G4S,G5C,G5S,G6C,G6S,G7C,G7S
REG(1)    G8C,G8S,G9C,G9S,G10C,G10S,G11C,G11S,G12C,G12S
REG(1)  H1C,H1S,H2C,H2S,H3C,H3S,H4C,H4S,H5C
REG(1)  H5S,H6C,H6S,H7C,H7S,H8C,H0C
REG(1)  Z8,Z9,Z10,Z11,Z12,Z15,Z18,Z20,Z24
;
; PINS FOR THE MODEL
;
PIN     SIN(1,2),B0(3),SCIN(4)
; OUTPUT PINS
PIN     P15(5),P14(6),P13(7),P12(8),P11(9)
PIN     P10(10),P9(11),P48(12,16),P03(17,20)
; INPUT PINS FOR X, Y.
PIN     DATX(151,158),DATY(159,166)
;
; INPUT-TO-OUTPUT DELAY FOR THE MULTIPLIER
;
EVW     W121(121)
;
; MODULE CODE BEGINS HERE
;
;   INPUT LATCHES ARE MODELED HERE
;
       BNE DATX,OLX,MULT             ; CHECK FOR CHANGE IN X.
       BNE DATY,OLY,MULT             ; CHECK FOR CHANGE IN Y.
       EXR                           ; IF NO CHANGE, EXIT.
;
MULT:  MOV SIN,SIGN                  ; SIGN BITS FOR X, Y.
       MOV DATX,OLX                  ; OLX HAS NEW DATA FOR NEXT
       MOV DATY,OLY                  ; CHECK.
;
```

```
        COM OLX                  ; GIVES X0Y0,X1Y1,X2Y2,
        COM OLY                  ;    ..  X3Y3,X4Y4,
        OR  OLX,OLY,REG0         ;    ..  X5Y5,X6Y6,
        COM REG0                 ;    ..  X7Y7. GOT BY ¢[¢X + ¢Y].
        ROR OLX                  ; OLX = [X7 X0 X1 X2 X3 X4 X5 X6]
        OR  OLX,OLY,REG7         ; GIVES X7Y0,X0Y1,X1Y2,X2Y3,X3Y4,
        COM REG7                 ;    ..  X4Y5,X5Y6,X6Y7.
        ROR OLX                  ; OLX = [X6 X7 X0 X1 X2 X3 X4 X5]
        OR  OLX,OLY,REG6         ; GIVES X6Y0,X7Y1,X0Y2,X1Y3,X2Y4,
        COM REG6                 ;    ..  X3Y5,X4Y6,X5Y7.
        ROR OLX                  ; OLX = [X5 X6 X7 X0 X1 X2 X3 X4]
        OR  OLX,OLY,REG5         ; GIVES X5Y0,X6Y1,X7Y2,X0Y3,X1Y4,
        COM REG5                 ;    ..  X2Y5,X3Y6,X4Y7.
        ROR OLX                  ; OLX = [X4 X5 X6 X7 X0 X1 X2 X3]
        OR  OLX,OLY,REG4         ; GIVES X4Y0,X5Y1,X6Y2,X7Y3,X0Y4,
        COM REG4                 ;    ..  X1Y5,X2Y6,X3Y7.
        ROR OLX                  ; OLX = [X3 X4 X5 X6 X7 X0 X1 X2]
        OR  OLX,OLY,REG3         ; GIVES X3Y0,X4Y1,X5Y2,X6Y3,X7Y4,
        COM REG3                 ;    ..  X0Y5,X1Y6,X2Y7.
        ROR OLX                  ; OLX = [X2 X3 X4 X5 X6 X7 X0 X1]
        OR  OLX,OLY,REG2         ; GIVES X2Y0,X3Y1,X4Y2,X5Y3,X6Y4,
        COM REG2                 ;    ..  X7Y5,X0Y6,X1Y7.
        ROR OLX                  ; OLX = [X1 X2 X3 X4 X5 X6 X7 X0]
        OR  OLX,OLY,REG1         ; GIVES X1Y0,X2Y1,X3Y2,X4Y3,X5Y4,
        COM REG1                 ;    ..  X6Y5,X7Y6,X0Y7.
        ROR OLX                  ; OLX = [X0 X1 X2 X3 X4 X5 X6 X7]
; OUTPUT P15
        IDX REG0(0),1,1          ;
        MOV(W121) @1,P15         ;   P15 = X7Y7  AFTER 121 NS.
; OUTPUT P14
        IDX   REG7(0),1,1            ;
        IDX   REG1(1),1,2            ;
        MOV   @1,BIT1                ;
        MOV   @2,BIT2                ;  P14 = (X6Y7 .EXOR. X7Y6)
        XOR   BIT1,BIT2,BIT3         ;
        MOV(W121)   BIT3,P14         ;
        AND   BIT1,BIT2,BIT3
        MOV   BIT3,F32C              ; PARTIAL CARRY,...
                                     ; F32C= X6Y7 .AND. X7Y6
; F2Z
        IDX   REG0(7),1,1
        IDX   SIGN(0),2,2
;
; OBTAIN 1ST SET OF PARTIAL SUMS AND CARRY'S [F'S] :
;
        BEQ   #1,@1,ONE                  ;
        MOV   F2Z1@2,F2ZS                ;
        BRU   OTHFS                      ;
ONE:    MOV   F2Z2@2,F2ZS                ;
        BRU   OTHFS                      ;
```

```
;
F2Z1:   BYT  #0,#1,#1,#0
F2Z2:   BYT  #1,#0,#0,#1
;
OTHFS:     IDX  REG6(4),1,1          ; 1. F9C AND F9S
           IDX  REG0(5),1,2          ;
           IDX  REG2(6),1,3          ; MOVE THE THREE PARTIAL PRODUCTS
           JSR  DCOD3                ; INTO A SINGLE 3-BIT REG (SUBR.
           JSR  AOI3                 ; DCOD3). OBTAIN SUM & CARRY FOR
           MOV  FCAR3,F9C            ; THEIR ADDITION.
           MOV  FSUM3,F9S
        IDX  REG7(4),1,1             ; 2. F12C AND F12S
        IDX  REG1(5),1,2
        IDX  REG3(6),1,3
        JSR  DCOD3
        JSR  AOI3
        MOV  FCAR3,F12C
        MOV  FSUM3,F12S
           IDX  REG0(4),1,1          ; 3. F16C AND F16S
           IDX  REG2(5),1,2
           IDX  REG4(6),1,3
           JSR  DCOD3
           JSR  AOI3
           MOV  FCAR3,F16C
           MOV  FSUM3,F16S
        IDX  REG3(1),1,1             ; 4. F18C AND F18S
        IDX  REG5(2),1,2
        IDX  REG7(3),1,3
        JSR  DCOD3
        JSR  AOI3
        MOV  FCAR3,F18C
        MOV  FSUM3,F18S
           IDX  REG1(4),1,1          ; 5. F19C AND F19S
           IDX  REG3(5),1,2
           IDX  REG5(6),1,3
           JSR  DCOD3
           JSR  AOI3
           MOV  FCAR3,F19C
           MOV  FSUM3,F19S
        IDX  REG2(0),1,1             ; 6. F21C AND F21S
        IDX  REG4(1),1,2
        IDX  REG6(2),1,3
        JSR  DCOD3
        JSR  AOI3
        MOV  FCAR3,F21C
        MOV  FSUM3,F21S
           IDX  REG0(3),1,1          ; 7. F22C AND F22S
           IDX  REG2(4),1,2
           IDX  REG4(5),1,3
           JSR  DCOD3
```

```
          JSR   AOI3
          MOV   FCAR3,F22C
          MOV   FSUM3,F22S
        IDX   REG3(0),1,1           ; 8. F24C AND F24S
        IDX   REG5(1),1,2
        IDX   REG7(2),1,3
        JSR   DCOD3
        JSR   AOI3
        MOV   FCAR3,F24C
        MOV   FSUM3,F24S
          IDX   REG1(3),1,1         ; 9. F25C AND F25S
          IDX   REG3(4),1,2
          IDX   REG5(5),1,3
          JSR   DCOD3
          JSR   AOI3
          MOV   FCAR3,F25C
          MOV   FSUM3,F25S
        IDX   REG4(0),1,1           ; 10. F26C AND F26S
        IDX   REG6(1),1,2
        IDX   REG0(2),1,3
        JSR   DCOD3
        JSR   AOI3
        MOV   FCAR3,F26C
        MOV   FSUM3,F26S
          IDX   REG5(0),1,1         ; 11. F29C AND F29S
          IDX   REG7(1),1,2
          IDX   REG1(2),1,3
          JSR   DCOD3
          JSR   AOI3
          MOV   FCAR3,F29C
          MOV   FSUM3,F29S
        IDX   REG6(0),1,1           ; 12. F31C AND F31S
        IDX   REG0(1),1,2
        IDX   REG2(2),1,3
        JSR   DCOD3
        JSR   AOI3
        MOV   FCAR3,F31C
        MOV   FSUM3,F31S
;
MORFS:    IDX   REG7(6),1,1         ; 1. F3C AND F3S
          IDX   REG1(7),1,2
          JSR   DCOD2
          JSR   AOI2
          MOV   FCAR2,F3C
          MOV   FSUM2,F3S
        IDX   REG6(5),1,1           ; 2. F5C AND F5S
        IDX   REG2(7),1,2
        JSR   DCOD2
        JSR   AOI2
        MOV   FCAR2,F5C
```

```
        MOV   FSUM2,F5S
          IDX  REG5(4),1,1        ; 3. F8C AND F8S
          IDX  REG3(7),1,2
          JSR  DCOD2
          JSR  AOI2
          MOV  FCAR2,F8C
          MOV  FSUM2,F8S
        IDX  REG4(3),1,1          ; 4. F10C AND F10S
        IDX  REG4(7),1,2
        JSR  DCOD2
        JSR  AOI2
        MOV  FCAR2,F10C
        MOV  FSUM2,F10S
          IDX  REG3(2),1,1        ; 5. F13C AND F13S
          IDX  REG5(7),1,2
          JSR  DCOD2
          JSR  AOI2
          MOV  FCAR2,F13C
          MOV  FSUM2,F13S
        IDX  REG2(1),1,1          ; 6. F17C AND F17S
        IDX  REG6(7),1,2
        JSR  DCOD2
        MOV  #1,FLAG
        JSR  AOI2
        MOV  FCAR2,F17C
        MOV  FSUM2,F17S
F20:    IDX  REG1(0),1,1       ;  F20C AND F20S
        IDX  REG7(7),1,2
        JSR  DCOD2             ;  OBTAIN THE TWO VALUES IN A 2-BIT
         IDX  SIGN(0),2,1      ;  REG.DECODE SIGN BITS AND THE
         IDX  DCO2R(0),2,2     ;  CONTENTS OF THE ABOVE REG. TO
         MOV  F20SM@2, F20S    ;  OBTAIN THE SUM ..
         BRU  SINXY@1
SINXY:   BYT  PLPL,PLMI,MIPL,MIMI ; ..AND THE CARRY, DEPENDING ON
PLPL:    MOV  CRY1@2,F20C         ; +X , +Y
         BRU  GBEGN
PLMI:    MOV  CRY2@2,F20C         ; +X , -Y
         BRU  GBEGN
MIPL:    MOV  CRY3@2,F20C         ; -X , +Y
         BRU  GBEGN
MIMI:    MOV  CRY4@2,F20C         ; -X , -Y .
         BRU  GBEGN
;
F20SM:   BYT  #0,#1,#1,#0       ; TRUTH-TABLE FOR F20S.
CRY1:    BYT  #0,#0,#0,#1       ; TRUTH-TABLES FOR F20C DEPENDING
CRY2:    BYT  #1,#1,#0,#1       ; ON THE SIGNS OF X AND Y...
CRY3:    BYT  #1,#0,#1,#1       ;    ...
CRY4:    BYT  #1,#0,#0,#0       ;    .
;
;
```

```
;
;=========================================
;  SUBROUTINES FOR AOI OPERATIONS
;=========================================
;
;************************
;  1) SUBROUTINE DCOD3
;************************
DCOD3:  MOV  #0,DCO3R     ; THIS SUBROUTINE GETS THREE DIFFERENT
        BEQ  @1,OTHER     ; VALUES IN INDEX REGS. 1,2 & 3,
        BIS  #0,DCO3R     ; AND PUTS THEM INTO A 3-BIT REGISTER,
OTHER:  BEQ  @2,LAST      ;'DCO3R', LSB TO MSB RESPECTIVELY.
        BIS  #1,DCO3R     ;
LAST:   BEQ  @3,RET       ;
        BIS  #2,DCO3R     ;
RET:    RTS
;
;************************
;  2) SUBROUTINE AOI3
;************************
AOI3:   IDX  DCO3R(0),3,4        ; OBTAINS THE INVERTED CARRY
        MOV  CARR3@4,FCAR3       ; AND THE SUM FOR THE THREE BITS
        MOV  SUM3@4,FSUM3        ; OF THE 'DCO3R' REGISTER.
        RTS
;
; TRUTH-TABLES
CARR3:  BYT  #1,#1,#1,#0,#1,#0,#0,#0   ; INVERTED CARRY.
SUM3:   BYT  #0,#1,#1,#0,#1,#0,#0,#1   ; SUM.
;************************
;  3) SUBROUTINE DCOD2
;************************
DCOD2:  MOV  #0,DCO2R          ; SIMILAR TO DCOD3. BUT PUTS TWO
        BEQ  #01,NXT1          ; SEPARATE BITS INTO A 2-BIT REG.,
        BIS  #0,DCO2R          ; 'DCO2R', FOR SUBSEQUENT DECODING.
NXT1:   BEQ  #02,RETUN         ;
        BIS  #1,DCO2R          ;
RETUN:  RTS
;
;**************************
;  4) SUBROUTINE AOI2
;**************************
AOI2:   IDX  SIGN(0),2,3       ; OBTAINS SUM AND CARRY FOR THE
        IDX  DCO2R(0),2,4      ; ADDITION OF 2 BITS.
        BEQ  #0,FLAG,NOF17     ; F17 HAS A DIFFERENT TRUTH-TABLE.
        MOV  #0,FLAG           ;  PROCESS TO OBTAIN F17S & F17C ..
        MOV  #3,TEMP2          ;  ... DIFFERENT OUTPUT ONLY FOR ..
        BEQ  TEMP2,SIGN,SPRAT     ;  ...  -X , -Y.
NOF17:  BRU  SXSY@3
SXSY:   BYT  XYPP,XYPN,XYNP,XYNN  ; +X+Y, +X-Y, -X+Y, -X-Y RESP.
;
```

```
XYPP:    MOV  CAR21@4,FCAR2          ; +X , +Y
         MOV  SUM21@4,FSUM2
         RTS
CAR21:   BYT  #0,#0,#0,#1
SUM21:   BYT  #0,#1,#1,#0
;
XYPN:    MOV  CAR22@4,FCAR2          ; +X , -Y
         MOV  SUM22@4,FSUM2
         RTS
CAR22:   BYT  #0,#1,#0,#0
SUM22:   BYT  #1,#0,#0,#1
;
XYNP:    MOV  CAR23@4,FCAR2          ; -X , +Y
         MOV  SUM23@4,FSUM2
         RTS
CAR23:   BYT  #0,#0,#1,#0
SUM23:   BYT  #1,#0,#0,#1
;
XYNN:    MOV  CAR24@4,FCAR2          ; -X , -Y
         MOV  SUM24@4,FSUM2
         RTS
CAR24:   BYT  #1,#0,#0,#0
SUM24:   BYT  #0,#1,#1,#0
;
SPRAT:   MOV  CRF17@4,FCAR2          ;
         MOV  SMF17@4,FSUM2
         RTS
CRF17:   BYT  #1,#1,#1,#0            ; F17C FOR -X , -Y
SMF17:   BYT  #1,#0,#0,#1            ; F17S FOR -X , -Y
;
;============================================================
;  G'S  ARE GENERATED FROM HERE. THESE ARE THE SECOND SET OF
;     PARTIAL SUMS AND CARRY'S.
;============================================================
GBEGN: IDX  REG0(6),1,1              ; 13. G1C AND G1S
       IDX  F5S(0),1,2
       IDX  F8C(0),1,3
       JSR  DCOD3
       JSR  AOI3                     ; USE THE SUBR. AOI3
       MOV  FCAR3,G1C
       MOV  FSUM3,G1S
         IDX  REG7(5),1,1            ; 14. G2C AND G2S
         IDX  REG1(6),1,2
         IDX  F8S(0),1,3
         JSR  DCOD3
         JSR  AOI3
         MOV  FCAR3,G2C
         MOV  FSUM3,G2S
       IDX  F9S(0),1,1               ; 15. G3C AND G3S
       IDX  F10S(0),1,2
```

```
IDX  F 3C(0),1,3
JSR  DCOD3
JSR  AOI3
MOV  FCAR3,G3C
MOV  FSUM3,G3S
  IDX  F12S(0),1,1          ; 16. G4C AND G4S
  IDX  F13S(0),1,2
  IDX  F17C(0),1,3
  JSR  DCOD3
  JSR  AOI3
  MOV  FCAR3,G4C
  MOV  FSUM3,G4S
IDX  F16S(0),1,1            ; 17. G6C AND G6S
IDX  F17S(0),1,2
IDX  REG4(2),1,3
JSR  DCOD3
JSR  AOI3
MOV  FCAR3,G6C
MOV  FSUM3,G6S
  IDX  F18S(0),1,1          ; 18. G8C AND G8S
  IDX  F19S(0),1,2
  IDX  F20S(0),1,3
  JSR  DCOD3
  JSR  AOI3
  MOV  FCAR3,G8C
  MOV  FSUM3,G8S
IDX  F21S(0),1,1            ; 19. G9C AND G9S
IDX  F22S(0),1,2
IDX  REG6(6),1,3
JSR  DCOD3
JSR  AOI3
MOV  FCAR3,G9C
MOV  FSUM3,G9S
  IDX  F19C(0),1,1        ; 1. G7C AND G7S
  IDX  REG6(3),1,2        ;  NEED TO WRITE SUBROUTINE GGEN
  IDX  F18C(0),1,3        ;  FOR G7, G11, G12
  JSR  DCOD3              ;  AS THE TRUTH TABLE IS DIFFERENT
  JSR  GGEN3              ;  FROM AOI3'S.
  MOV  FCAR3,G7C
  MOV  FSUM3,G7S
IDX  REG2(3),1,1              ; 2. G11C AND G11
IDX  F29C(0),1,2
IDX  REG4(4),1,3
JSR  DCOD3
JSR  GGEN3
MOV  FCAR3,G11C
MOV  FSUM3,G11S
  IDX  F29S(0),1,1            ; 3. G12C AND G12S
  IDX  F31C(0),1,2
  IDX  REG3(3),1,3
```

```
      JSR  DCOD3
      JSR  GGEN3
      MOV  FCAR3,G12C
      MOV  FSUM3,G12S
    IDX  F25S(0),1,1              ; 20. G10C AND G10S
    IDX  F24S(0),1,2
    COM  F26C
    IDX  F26C(0),1,3
    JSR  DCOD3
    JSR  AOI3                     ; USE AOI3 FOR THIS.
    MOV  FCAR3,G10C
    MOV  FSUM3,G10S
      IDX  REG5(3),1,1            ;  G5C AND G5S
      MOV  @1,BIT1
      XOR  BIT1,F16C,G5S
      COM  BIT1
      OR   BIT1,F16C,G5C
;=================================================================
;   H'S ARE GENERATED HERE. THIRD SET OF PARTIAL SUMS AND CARRY'S
;       USING F'S AND G'S.
;=================================================================
    IDX  F12C(0),1,1            ; 21. H2C AND H2S
    IDX  G4C(0),1,2
    IDX  G5C(0),1,3
    JSR  DCOD3
    JSR  AOI3                   ; USE AOI3 SUBROUTINE.
    MOV  FCAR3,H2C
    MOV  FSUM3,H2S
      IDX  F21C(0),1,1          ; 22. H5C AND H5S
      IDX  F22C(0),1,2
      IDX  G9C(0),1,3
      JSR  DCOD3
      JSR  AOI3
      MOV  FCAR3,H5C
      MOV  FSUM3,H5S
    IDX  F24C(0),1,1            ; 23. H6C AND H6S
    IDX  F25C(0),1,2
    IDX  G10C(0),1,3
    JSR  DCOD3
    JSR  AOI3
    MOV  FCAR3,H6C
    MOV  FSUM3,H6S
      IDX  G5S(0),1,1           ; 4. H3C AND H3S
      IDX  G7C(0),1,2           ;  NEED TO USE SUBROUTINE GGEN
      IDX  G6C(0),1,3           ;  FOR G7, G11, G12, H3, H4, H7
      JSR  DCOD3
      JSR  GGEN3
      MOV  FCAR3,H3C
      MOV  FSUM3,H3S
    IDX  F20C(0),1,1            ; 5. H4C AND H4S
```

```
       IDX  G8C(0),1,2
       IDX  G7S(0),1,3
       JSR  DCOD3
       JSR  GGEN3
       MOV  FCAR3,H4C
       MOV  FSUM3,H4S
         IDX  G11S(0),1,1          ; 6. H7C AND H7S
         IDX  F26S(0),1,2
         IDX  G12C(0),1,3
         JSR  DCOD3
         JSR  GGEN3
         MOV  FCAR3,H7C
         MOV  FSUM3,H7S
       IDX  G3C(0),1,1             ; 24. H1C AND H1S
       IDX  F9C(0),1,2
       COM  F10C
       IDX  F10C(0),1,3
       JSR  DCOD3
       JSR  AOI3                   ; USE SUBR. AOI3 FOR THESE.
       MOV  FCAR3,H1C
       MOV  FSUM3,H1S
;
; MOVE 0'S INTO REGISTERS FOR INITIAL VALUES
;
       MOV  #0,REGA
       MOV  #0,REGB
       MOV  #0,REGAL
       MOV  #0,REGBL
;
       IDX  F3S(0),1,1             ; 7. H0C AND A2BAR
       IDX  G1C(0),1,2
       IDX  F5C(0),1,3
       JSR  DCOD3
       JSR  GGEN3
       MOV  FCAR3,H0C
       BEQ  #0,FSUM3,JMP1
       BIS  #2,REGAL               ; ¢A2 IS 1. ( A2BAR = 1)
;
;  GENERATE P12, H8C AND P13
;
JMP1:  XOR  F31S,F32C,BIT1         ;   P13 IS OBTAINED
       MOV(W121)  BIT1,P13
;
       AND  F31S,F32C,BIT1         ;   P12 AND H8C
       COM  BIT1
       XOR  BIT1,G12S,BIT2
       MOV(W121)  BIT2,P12
       OR   BIT1,G12S,BIT3
       COM  BIT3
       MOV  BIT3,H8C
```

```
        BRU  ABBEG                      ; SUBROUTINES START NEXT.
                                        ; JUMP OVER THEM.
;=================================================================
;  SUBROUTINE GGEN3 FOR THE SPECIAL CASE OF GENERATING SOME G'S,
;     H'S ETC. IT'S A SLIGHTLY MODIFIED FORM OF AOI3.
;===========********************==================================
GGEN3:   IDX  DCO3R(0),3,5
         MOV  SUM@5,FSUM3               ; CORRECT OPERATION
         MOV  #2,FLG2
         XOR  @5,FLG2,DCO3R             ; XOR WITH 010 TO INVERT THE
         IDX  DCO3R(0),3,5              ; CENTER-BIT.
         MOV  CARR@5,FCAR3
         RTS
CARR:   BYT  #1,#1,#1,#0,#1,#0,#0,#0
SUM:    BYT  #0,#1,#1,#0,#1,#0,#0,#1
;=======================================================
;   COMPLEMENTS OF  A'S AND B'S ARE GENERATED HERE. THESE
;   ARE THE PENULTIMATE SUMS AND CARRY'S.
;          REGA IS USED TO STORE COMPLEMENTED A4 TO A10
;          REGAL IS USED TO STORE COMPLEMENTED A0 TO A3
;          REGB IS USED TO STORE COMPLEMENTED B4 TO B10
;          REGBL IS USED TO STORE COMPLEMENTED B0 TO B3
;=======================================================
ABBEG:  IDX  F3C(0),1,1                 ; 8. B0 AND A1
        IDX  H0C(0),1,2                 ;  NEED TO USE SUBROUTINE GGEN3
        IDX  F2ZS(0),1,3                ;  FOR (B0,A1),(B2,A3),(B3,A4)
        JSR  DCOD3                      ;      (B4,A5),(B5,A6),(B6,A7),
        JSR  GGEN3                      ;      (B7,A8),(B9,A10)
        BEQ  #0,FCAR3,A1
        BIS  #0,REGBL
A1:     BEQ  #0,FSUM3,B2
        BIS  #1,REGAL
;
B2:     IDX  H1C(0),1,1                 ; 9. B2 AND A3
        IDX  G2C(0),1,2
        IDX  G1S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A3
        BIS  #2,REGBL
A3:     BEQ  #0,FSUM3,B3
        BIS  #3,REGAL
;
B3:     IDX  G2S(0),1,1                 ; 10. B3 AND A4
        IDX  H1S(0),1,2
        IDX  H2C(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A4
        BIS  #3,REGBL
```

```
A4:     BEQ  #0,FSUM3,B4
        BIS  #0,REGA
;
B4:     IDX  H3C(0),1,1          ; 11. B4 AND A5
        IDX  H2S(0),1,2
        IDX  G3S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A5
        BIS  #0,REGB
A5:     BEQ  #0,FSUM3,B5
        BIS  #1,REGA
;
B5:     IDX  H3S(0),1,1          ; 12. B5 AND A6
        IDX  H4C(0),1,2
        IDX  G4S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A6
        BIS  #1,REGB
A6:     BEQ  #0,FSUM3,B6
        BIS  #2,REGA
;
B6:     IDX  H5C(0),1,1          ; 13. B6 AND A7
        IDX  H4S(0),1,2
        IDX  G6S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A7
        BIS  #2,REGB
A7:     BEQ  #0,FSUM3,B7
        BIS  #3,REGA
;
B7:     IDX  H6C(0),1,1          ; 14. B7 AND A8
        IDX  H5S(0),1,2
        IDX  G8S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A8
        BIS  #3,REGB
A8:     BEQ  #0,FSUM3,B9
        BIS  #4,REGA
;
B9:     IDX  H7C(0),1,1          ; 15. B9 AND A10
        IDX  G11C(0),1,2
        IDX  G10S(0),1,3
        JSR  DCOD3
        JSR  GGEN3
        BEQ  #0,FCAR3,A10
        BIS  #5,REGB
```

```
A10:    BEQ  #0,FSUM3,DOA0
        BIS  #6,REGA
;
DOA0:   IDX  SIGN(0),2,1        ;  A0, OBTAINED
        MOV  A0TBL@1,BIT1       ;  BY DECODING THE SIGN BITS.
        BEQ  #0,BIT1,B8
        BIS  #0,REGAL
        BRU  B8
A0TBL: BYT   #1,#0,#0,#0
;
B8:     COM  G9S                  ;   B8 AND A9
        OR   H6S,G9S,BIT1
        BEQ  #0,BIT1,A9
        BIS  #4,REGB
A9:     COM  H6S
        XOR  G9S,H6S,BIT2
        BEQ  #0,BIT2,B10
        BIS  #5,REGA
;
B10:    AND  H7S,H8C,BIT1         ;   B10 AND P11
        COM  BIT1
        BEQ  #0,BIT1,PCHK
        BIS  #6,REGB
PCHK:   XOR  H7S,H8C,BIT2
        MOV(W121)  BIT2,P11
;==================================================================
;       CARRY-LOOK-AHEAD (CLA) BLOCKS ARE CODED HERE.
;THESE USE THE A'S AND B'S AS INPUTS AND GIVE THE Z'S AS OUTPUTS.
;==================================================================
;
        OR   REGA,REGB,BFAND      ;  HIGHER 7 BITS. BFAND = ¢A + ¢B
        AND  REGA,REGB,BFOR       ;  HIGHER 7 BITS. BFOR = ¢A . ¢B
        OR   REGAL,REGBL,BFANL      ;  LOWER 4 BITS
        AND  REGAL,REGBL,BFORL
;
        COM  BFOR                   ;  ¢(¢A . ¢B) = A+B
        COM  BFAND                  ;  ¢(¢A + ¢B) = A.B
        COM  BFORL
        COM  BFANL
;
        COM  REGA                   ;  ¢A IS INVERTED TO GET A
        COM  REGB                   ;  REGB = ¢(¢B) = B
        COM  REGAL
        COM  REGBL
        BIR  #1,REGBL            ; BIT #1 (B1) DOESN'T EXIST AT ALL.
        XOR  REGAL,REGBL,REGBL   ; XOR(A,B) IN REGBL & REGB TO BE
        XOR  REGA,REGB,REGB      ; USED LATER WITH C'S.
;
        IDX  BFAND(6),1,1          ;  @1 = ( A10.B10 )
        IDX  REGB(5),1,2           ;  BIT2 = ( A9 .EXOR. B9 )
```

```
        MOV   @2,BIT2
        COM   BIT2                  ; ( A9 .EXORN. B9 )
        XOR   @1,BIT2,BIT2          ;  BIT2 HAS P9BAR
        COM   BIT2                  ;  TO OBTAIN P9
        IDX   REGB(6),1,3           ;  @3 = ( A10 .EXOR. B10 )
        MOV   @3,P10R               ;  P10 IS THE OUTPUT PIN .
        MOV(W121) P10R,P10
        MOV(W121) BIT2,P9           ; OUTPUT P9,P10 IN 121 NS.
;
        IDX   BFOR(0),6,5
        BNE   #63,@5,BFOZ2          ; IF ALL ARE NOT 1'S BRANCH
        IDX   BFAND(6),1,6
        BEQ   @6,BFOZ2              ;  BRANCH IF @6=0
        BIS   #0,ZREG               ;  ZREG(0) = Z1 = 1
        BRU   PTZ2
BFOZ2:  BIR   #0,ZREG               ;  ZREG(0) = 0
PTZ2:   IDX   BFOR(0),5,5
        BNE   #31,@5,BFOZ3
        IDX   BFAND(5),1,6
        BEQ   @6,BFOZ3
        BIS   #1,ZREG               ;  ZREG(1) = Z2 = 1
        BRU   PTZ3
BFOZ3:  BIR   #1,ZREG               ;  ZREG(1) = 0
PTZ3:   IDX   BFOR(0),4,5
        BNE   #15,@5,BFOZ4          ; IF ALL ARE NOT 1'S BRANCH
        IDX   BFAND(4),1,6
        BEQ   @6,BFOZ4              ;  BRANCH IF @6=0
        BIS   #2,ZREG               ;  ZREG(2) = Z3 = 1
        BRU   PTZ4
BFOZ4:  BIR   #2,ZREG               ;  ZREG(2) = 0
PTZ4:   IDX   BFOR(0),3,5
        BNE   #7,@5,BFOZ5
        IDX   BFAND(3),1,6
        BEQ   @6,BFOZ5
        BIS   #3,ZREG               ;  ZREG(3) = Z4 = 1
        BRU   PTZ5
BFOZ5:  BIR   #3,ZREG               ;  ZREG(3) = 0
PTZ5:   IDX   BFOR(0),2,5
        BNE   #3,@5,BFOZ6           ; IF ALL ARE NOT 1'S BRANCH
        IDX   BFAND(2),1,6
        BEQ   @6,BFOZ6              ;  BRANCH IF @6=0
        BIS   #4,ZREG               ;  ZREG(4) = Z5 = 1
        BRU   PTZ6
BFOZ6:  BIR   #4,ZREG               ;  ZREG(4) = 0
PTZ6:   IDX   BFOR(0),1,5
        IDX   BFAND(1),1,6
        MOV   @6,BIT1
        AND   @5,BIT1,BIT1
        BEQ   #0,BIT1,BFOZ7
        BIS   #5,ZREG               ;  ZREG(5) = Z6 = 1
```

```
        BRU   PTZ7
BFOZ7:  BIR   #5,ZREG
PTZ7:   IDX   BFAND(0),1,6
        BEQ   #0,@6,BFOZ8
        BIS   #6,ZREG                 ;  ZREG(6) = Z7 = 1
        BRU   PTZ8
BFOZ8:  BIR   #6,ZREG
PTZ8:   IDX   BFORL(3),1,5
        MOV   @5,Z8
PTZ9:   IDX   BFANL(3),1,6
        MOV   @6,Z9
PTZ10:  IDX   BFORL(2),2,5
        MOV   TBL10@5,BIT1
        MOV   BIT1,Z10
        BRU   PTZ11
TBL10:  BYT   #0,#0,#0,#1
PTZ11:  IDX   REGAL(1),1,6            ; REGAL(1) = A1
        AND   @6,BIT1,Z11             ; Z11 = [ (A2+B2)(A3+B3)A1 ]
;
PTZ12:  IDX   BFORL(2),1,5            ;  CHECK FOR (A2+B2) = 0
        BEQ   #0,@5,12ZRO             ;  YES. Z12 = 0
        IDX   BFANL(2),1,6            ;  NO.
        BEQ   #0,@6,MOR12             ;  IF (A2.B2) = 0, BRANCH..
        MOV   #1,Z12                  ;  NO. Z12 = 1  [ (A2.B2) = 1 ]
        BRU   PTZ15
MOR12:  IDX   BFANL(3),1,6      ;  IS (A2+B2).¢(A2.B2).(A3.B3) = 1 ?
        BEQ   #0,@6,12ZRO       ;  NO ; Z12 = 0
        MOV   #1,Z12            ;  YES ; Z12 = 1
        BRU   PTZ15
12ZRO:  MOV   #0,Z12
;
PTZ15:  IDX   BFOR(1),2,5             ; Z15
        MOV   TBL15@5,Z15
        BRU   PTZ18
TBL15:  BYT   #0,#0,#0,#1
PTZ18:  IDX   BFOR(3),1,1             ; Z18
        IDX   BFAND(4),1,2
        MOV   @1,BIT1
        MOV   @2,BIT2
        AND   BIT1,BIT2,Z18
PTZ20:  IDX   BFOR(3),2,5             ; Z20
        MOV   TBL20@5,Z20
        BRU   PTZ24
TBL20:  BYT   #0,#0,#0,#1
PTZ24:  IDX   BFOR(5),1,1             ; Z24
        IDX   BFAND(6),1,2
        AND   @1,@2,Z24
;
;
;
```

```
;===========================================================
;     C'S ARE GENERATED HERE FROM THE Z'S OBTAINED ABOVE
;       THEY ARE STORED IN REGA AND REGAL.
;                    REGA                          REGAL
;        6  5   4   3   2   1   0          3   2   1   0
;       |--------------------------|      |-------------|
;       |__________________________|      |_____________|
;        0   0  C9  C8  C7  C6  C5         C4  C3  C2  C1
;===========================================================
        IDX   REGAL(1),1,2          ;  STORE BIT #1 (A1) IN @1
                                    ;  FOR C1 CALCULATION
;
        MOV   #0,REGAL              ;  RESET FOR FUTURE USE
        MOV   #0,REGA               ;  FOR COMPARISON WITH ZREG
        BNE   REGA,ZREG,C4EQ1
        MOV   #0,C4                 ;  ZREG = 0 . HENCE C4 = 0
        BIR   #3,REGAL              ;  AND   REGAL(3) = 0
        BRU   MORC
C4EQ1:  MOV   #1,C4                 ;  ZREG  .NE. 0  ; HENCE C4 = 1
        BIS   #3,REGAL              ;  AND   REGAL(3) = 1
;
MORC:   AND   C4,Z8,BIT1            ;  C3 IS DEALT WITH HERE
        OR    BIT1,Z9,BIT2
        BEQ   #0,BIT2,BFOC2
        BIS   #2,REGAL
        BRU   PTC2
BFOC2:  BIR   #2,REGAL
;
PTC2:   AND   C4,Z10,BIT1
        OR    BIT1,Z12,BIT2
        BEQ   #0,BIT2,BFOC1
        BIS   #1,REGAL              ; C2 = 1. REGAL(1) = 1
        BRU   PTC1
BFOC1:  BIR   #1,REGAL
;
PTC1:   AND   C4,Z11,BIT1           ;  A1 WAS MOVED INTO2 ABOVE.
        AND   @2,Z12,BIT2           ;  ( A1 + Z12 )
        OR    BIT1,BIT2,BIT3        ;  (C4.Z11) + (A1+Z12)
        BEQ   #0,BIT3,BFOC9
        BIS   #0,REGAL              ;  C1 = 1 . REGAL(0) = 1
        BRU   PTC9
BFOC9:  BIR   #0,REGAL
;
PTC9:   IDX   BFAND(5),1,1          ;  @1 = ( A9.B9 )
        OR    @1,Z24,BIT1
        BEQ   #0,BIT1,BFOC8
        BIS   #4,REGA               ;  C9 = 1.  REGA(4) = 1
        BRU   PTC8
BFOC8:  BIR   #4,REGA
;
```

```
PTC8:   IDX  BFOR(4),1,2            ;  @2 = ( A8+B8 )
        AND  @2,BIT1,BIT2
        IDX  BFAND(4),1,3           ;  @3 = ( A8.B8 )
        OR   @3,BIT2,BIT3           ;  C8 INTO BIT3
        BEQ  #0,BIT3,BFOC7
        BIS  #3,REGA                ;  C8 = 1. REGA(3) = 1
        BRU  PTC7
BFOC7:  BIR  #3,REGA
;
PTC7:   AND  Z20,BIT1,BIT2
        IDX  BFAND(3),1,4           ;  @4 = ( A7.B7 )
        OR   @4,Z18,BIT1
        OR   BIT2,BIT1,BIT3         ; C7 INTO BIT3
        MOV  BIT3,BIT2              ; TO BE USED FOR C6BAR AND C5BAR
        BEQ  #0,BIT3,BFOC6
        BIS  #2,REGA                ; C7 = 1.  REGA(2) = 1
        BRU  PTC6
BFOC6:  BIR  #2,REGA
;
PTC6:   IDX  BFOR(2),1,5            ;  @5 = ( A6+B6 )
        AND  @5,BIT2,BIT1           ;  BIT2 FROM ABOVE
        IDX  BFAND(2),1,6           ;  @6 = ( A6.B6 )
        OR   @6,BIT1,BIT3           ;  C6 INTO BIT3
        BEQ  #0,BIT3,BFOC5
        BIS  #1,REGA                ;  C6 = 1. REGA(1) = 1
        BRU  PTC5
BFOC5:  BIR  #1,REGA
;
PTC5:   AND  Z15,BIT2,BIT3     ; BIT2 FROM THE CALCULATION FOR C7BAR
        IDX  BFOR(1),1,7       ; @7 = ( A5+B5 )
        MOV  @7,BIT1
        AND  @6,BIT1,BIT1
        IDX  BFAND(1),1,8           ;  @8 = ( A5.B5 )
        OR   @8,BIT1,BIT2
        OR   BIT2,BIT3,BIT1         ;  C5 INTO BIT1
        BEQ  #0,BIT1,BFOPG
        BIS  #0,REGA                ;  C5 = 1. REGA(0) = 1
        BRU  PGEN
BFOPG:  BIR  #0,REGA
;
PGEN:   XOR  REGAL,REGBL,REGBL ;  XOR [ C, XOR(A,B) ] ==> REGBL
        XOR  REGA,REGB,REGB    ;                      AND REGB
;                             ;LOWER 5 BITS OF REGB GIVE P4 TO P8.
;==================================================================
; THIS PART OF THE PROGRAM MERELY OBTAINS THE MIRROR-IMAGE OF THE
;  CONTENTS OF REGB AND REGBL IN ORDER TO OUTPUT BITS P0 TO P8 IN
;    CORRECT ORDER OF SIGNIFICANCE.
;==================================================================
        MOV  #0,BUF4
        MOV  #0,REGAL
```

```
        MOV   #0,P4TO8
        AND   #1,REGB,REGA          ; CHECK REGB(0) .
        BEQ   REGA,SECND            ; BRANCH IF IT IS ZERO.
        BIS   #4,P4TO8              ; ELSE, SET P4TO8(4)
SECND:  AND   #2,REGB,REGA          ; SIMILAR TO ABOVE, FOR REGB(1).
        BEQ   REGA,THIRD
        BIS   #3,P4TO8
THIRD:  AND   #4,REGB,REGA          ; FOR REGB(2)
        BEQ   REGA,FORTH
        BIS   #2,P4TO8
FORTH:  AND   #8,REGB,REGA          ; FOR REGB(3)
        BEQ   REGA,FIFTH
        BIS   #1,P4TO8
FIFTH:  AND   #16,REGB,REGA         ; FOR REGB(4)
        BEQ   REGA,HIGH4
        BIS   #0,P4TO8
;
HIGH4:  AND   #1,REGBL,REGAL        ; FOR REGBL(0)
        BEQ   REGAL,TWO
        BIS   #3,BUF4
TWO:    AND   #2,REGBL,REGAL        ; FOR REGBL(1)
        BEQ   REGAL,PONE
        BIS   #2,BUF4
PONE:   AND   #4,REGBL,REGAL        ; FOR REGBL(2)
        BEQ   REGAL,ZERO
        BIS   #1,BUF4
ZERO:   AND   #8,REGBL,REGAL        ; FOR REGBL(3)
        BEQ   REGAL,OUT
        BIS   #0,BUF4
;
OUT:    MOV(W121) P4TO8,P48         ; OUTPUT P4 TO P8 AND
        MOV(W121) BUF4,PO3          ;  PO TO P3 IN 121 NS.
        EXR                         ; EXIT AND RESTART.
END
```

## Appendix C

## GSP ON P.C.

### C.1 NEED FOR THE SIMULATOR ON PC

The extremely high costs, non-transferability, and specific requirements on the main-frames restrict the use of IC-simulators to special environments. The need for a functional-level logic simulator which can run in all kinds of environments has always been felt. With the advent of PC's this has become possible.

The digital-logic simulators found in the manufacturing as well as academic environments run on main-frames and consume huge amounts of computer-money. The extra-high costs of main-frame usage can be offset at the expense of a marginal increase in execution time, which is inherent in the use of micro-computers (PC's). The additional feature of easy-transferability through the use of floppy disks is not to be overlooked.

## C.2 GSP SIMULATOR ON THE PC

GSP is now available on the PC as a functional level simulation package which can be used for fault-modeling also. The simulator is written in Microsoft Corporation's FORTRAN Subset and was tested on IBMPC and Zenith's Z-100 PC, under the Micro-Soft Disk-Operating System. The only additional requirement on the PC's is that there should be 300k bytes of memory.

The package consists of an assembler, a linker and a simulator. The assembler is in the form of nine different object modules (each of size less than 64k bytes) which are linked together in an executable file(55k bytes). The Linker is used to produce the correct sequence of code to be executed when one or more different modules are to be simulated [8]. The simulator is in five files which link together into an executable file of size 97k bytes [2].

To initialize some of the internal arrays, the Assembler and the Simulator read the respective data files- INIT.DAT and SIMINI.DAT.

The package is fairly interactive and is very easy to use.

TABLE 2

GSP Simulator Package

| Run File | Size | Purpose |
| --- | --- | --- |
| GSPASM.EXE | 55k bytes | Assembly |
| GSPLNKER.EXE | 41k bytes | Linking |
| GSPSIM.EXE | 97k bytes | Simulation |

## C.3 PERFORMANCE

Comparisons of real-time performance on the PC have been made with the VAX 11/780 system. Three different models of varying complexity were run on the two machines. The observations shown in table3 give a measure of performance.

The data available shows that simulations are only about 2.5 to 3 times slower on the personal computers when compared with the VAX. For this comparison, the times on the VAX 11/780 were obtained when no other users were on the system.

To obtain high speeds on the PC, electronic ram-drives were used. An electronic ram-drive is a special drive which can be obtained on the PC by executing an operating system program. The system can be configured such that a part of its internal memory functions as an extremely fast disk-drive, almost 30 times faster than a 5-1/4 inch disk-drive. This drive can be used like any other physical drive to store and read files, with the difference that the data on it is lost on system reset. A simulation program needs to do a lot of I/O with different files. The use of ram-drive obviously increases the speed of execution. On a system with sufficient memory ( in excess of 300k bytes, in this case ), it is easy to install a ram-drive. The ram-drive programs are readily available in the market for all kinds of PC's and operating systems.

TABLE 3

Performance Data

| Model | Machine | Assembly | Real-Time (secs.) Execution | Total |
|---|---|---|---|---|
| 1. Simple | VAX 11/780 | 16 | 60 | 76 |
| 2. Complex | VAX 11/780 | 11 | 78 | 89 |
| 3. More Complex | VAX 11/780 | 25 | 85 | 110 |
| 1. Simple | IBM PC | 58 | 100 | 158 |
| 2. Complex | IBM PC | 70 | 170 | 240 |
| 3. More Complex | IBM PC | 138 | 240 | 378 |

The use of disk-drives increased the times on the PC, as expected. With eight to ten users on the VAX and with ram-drives on the PC's, the times were only 1.5 to 1.8 as bad.

## C.4 CONCLUSIONS

Functional level modeling and simulation is the current trend in the industry in order to increase the efficiency of simulation and decrease the computer-processing costs. GSP is the functional level simulator which runs efficiently on the PC, with almost no costs of processing. The marginal increase in simulation times is inherent in the use of microprocessors when compared with main frames. But substantial savings in the cost of processing outweigh the disadvantage due to the small increase in execution times.

The use of PC also solves the software transferability problem. In academic environments especially, where PC's are gaining popularity, the simulation package can be easily made available to the students on floppy disks.

Some of the problems faced during the development and the steps taken to solve them are enumerated in Appendix D of this thesis.

## Appendix D

## PROBLEMS FACED IN TRANSFERRING GSP TO P.C.

### D.1 PROBLEMS FACED AND METHODS USED IN TRANSFERING GSP FROM CMS TO IBM PC.

1. Each object file can be no longer than 64k bytes for it to work on the PC. Hence, the assembler and simulator were broken down into small segments of 64k bytes or less, which were linked together for execution.

   The assembler resides in 9 separate files - G1,G2,G3,G4,G5,G6,G7,G8,I which are compiled separately and linked together to obtain the Run file [GSPASM.EXE]. The size of the run file is 55k bytes. The simulator is in 5 files, which yield a run file of 97k bytes.

2. There is a limit to the size of COMMON blocks for the MS FORTRAN on the PC. The maximum limit is 64k bytes, which results from the restriction on the maximum length of a segment for 8086 microprocessor.

   For this reason, some of the blocks were broken down into small ones.

   /TABLES/ into /TABLES/, /STOR1/ and /STOR2/

   /CHARTL/ into /CHART1/ and /CHART2/

The sizes of some of the arrays were reduced, too.

| | | |
|---|---|---|
| LIST(3000,7) | to | LIST(2000,7) |
| UCODE(15000) | to | UCODE(9000) |
| LABLST(4500) | to | LABLST(3000) |
| NAMLST(6000) | to | NAMLST(3000) |
| CODE1(10000) | to | CODE1(5000) |
| CODE2(15000) | to | CODE2(5000) |
| LABEL(10000) | to | LABEL(5000) |

3. The DATA statement cannot be used to initialize the variables and arrays declared in COMMON blocks, on the PC. For this reason, two data files had to be opened, one for the assembler [INIT.DAT] and the other for the simulator [SIMINI.DAT], to read and initialize some of the variables.
4. The internal representation of the alphabets(capitals & small), and the relative positioning of numbers, letters and symbols internal to the PC are used by the package. The assembler has to read the SOURCE file, much like a word-processor, to generate the executable code and to point out syntax errors, if any.

   In GSP, this relative positioning of numbers and letters in terms of the internal numerical

representation on CMS is used to perform certain actions. While, at the same time, the variables are also treated as CHARACTERS for input/output.

For example, for input/output, A & B are treated as characters (using A-format) and to decide if it was A or B, their internal representation as numbers i.e.,-1052753856 & -1035976640, is compared.

GSP was written for FORTG1 compiler wherein no "character" declaration is allowed and only the format decides if a variable is an integer or a character.

On the PC (MS FORTRAN 77 subset), character declaration is necessary. If this is done, comparison based on internal representation is not possible because characters cannot be compared as integers.

This problem can be solved by :

A) writing a look-up table representing each character by a number and checking what character is read (this will make the program longer and more time-consuming).

B) changing the existing program to handle character string comparison (this is too drastic a change).

C) looking at the relative positioning of the alphabets and numbers in the internal representation on the PC.

D) using the INTERNAL functions, CHAR and ICHAR in Fortran, to change integers to characters and vice-versa.

The problem was solved using solution (C) for the assembler and solution (D) for the simulator. The choice was based on the number of calls to the two functions and the delay involved in execution. The internal representation in the form of integers for the characters entered on the PC is shown below alongwith the representation on CMS.

As the relative positioning within alphabets and within numbers is the same on the PC, only minor changes were required.

5. Further, subroutines wherein only comparisons are made (no character input/output), the variables handling characters have been declared as integers (in assembler, on the PC). To do this, 'CHARACTER * 4' declaration is used in COMMON blocks on the PC because integer variables are allocated 4 bytes each, while characters are given only one by default.

TABLE 4

Representation of Characters as Integers, Internally.

| CHARACTER | REPRESENTATION ON CMS | PC |
|---|---|---|
| 0 | -264224704 | 538976304 |
| 1 | -247447488 | 538976305 |
| 2 | -230670272 | 538976306 |
| 3 | -230670272 | 538976307 |
| 4 | -213893056 | 538976308 |
| 5 | -197115840 | 538976309 |
| 6 | -180338624 | 538976310 |
| 7 | -163561408 | 538976311 |
| 8 | -130006976 | 538976312 |
| 9 | -113229760 | 538976313 |
| A | -1052753856 | 538976321 |
| B | -1035976640 | 538976322 |
| C | -1019199424 | 538976323 |
| D | -1002422208 | 538976324 |
| E | -985644992 | 538976325 |
| F | -968867776 | 538976326 |
| G | -952090560 | 538976327 |
| H | -935313344 | 538976328 |
| I | -918536128 | 538976329 |
| J | -784318400 | 538976330 |
| K | -767541184 | 538976331 |
| L | -750763968 | 538976332 |
| M | -733986752 | 538976333 |
| N | -717209536 | 538976334 |
| O | -700432320 | 538976335 |
| P | -683655104 | 538976336 |
| Q | -666877888 | 538976337 |
| R | -650100672 | 538976338 |
| S | -499105728 | 538976339 |
| T | -482328512 | 538976340 |
| U | -465551296 | 538976341 |
| V | -448774080 | 538976342 |
| W | -431996864 | 538976343 |
| X | -415219648 | 538976344 |
| Y | -398442432 | 538976345 |
| Z | -381665216 | 538976346 |

This way, some memory is lost but extra time is not lost in having a look up table. COMMON blocks /DELI/ and /SORC/ of the assembler are used for characters or integers depending on the use in the subroutine.

6. In subroutine PROC2 of the assembler, both comparison and input/output are done i.e., the variables are used as characters and integers at the same time. To handle this, the integer representation was changed to the character representation using CHAR(x) function. For example, to convert the integer representation of D i.e.,538976324 to its character representation, we do

```
INTEGER*4 TEMP
CHARACTER*1 CHARA
INTCHR = 538976325           ; code for 'D'.
TEMP =INTCHR - 538976256 ; SUBTRACT INTEGER VAL
                             ; OF FIRST ASCII CHTR.
CHARA = CHAR(TEMP)           ; CHARA CAN BE PRINTED
                             ; USING A-FORMAT
```

At some point in PROC2, the TEMP.TMP file is written with some data, then rewound and reread, using A-format. This data is written into the listing file. On the PC, we have written on and read from T7.TMP using I-format. To write the data into a

listing file, the above procedure was used with the extra set of variables (BCHAR) declared as characters. A similar procedure was used for the simulator.

7. Also, in MS FORTRAN, the iterations in READ statement (IMPLIED DO) should not be more than the data to be read.

   For example, to read

   538976329 538976334 538976336

   we cannot specify

   READ(5,7) (B(K3), K3=1,K2), where K2 > 3.

   Hence, a record of the number of terms written is kept in order to read the data again.[ variable array note(10) is used in the assembler ].

8. In order to improve the speed, metacommand NODEBUG is used. This improves the time by about 8%. (maximum of 15% - reports say)

In this way, the problems encountered while transfering GSP from CMS to the IBM PC were overcome. The package works perfectly well on the IBM and Z-100 PC's.

**The vita has been removed from the scanned document**