Fault Simulation in ADLIB-SABLE

Sumit Ghosh and Willem Vancleemput

Technical Report No. 83-242

March 1983

This work was supported by the *ARMY RESEARCH OFFICE CONTRACT NO DAAG-29-80- K-0046*

FAULT SIMULATION USING ADLIB-SABLE

SUMIT GHOSH and WILLEM VANCLEEMPUT

TECHNICAL REPORT No. 83-242

MARCH 1983

Center for integrated Systems, Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University, Stanford, CA-94305

Abstract

This technical report presents work in the area of deductive fault simulation.[1] This technique, one of the three fault simulation techniques discussed in the literature, has been implemented in ADLIB-SABLE, a hierarchical multi-level simulator designed and used at Stanford University. Most of the fault models illustrated in this report consider only two fault types : single stuck-at-0 and single stuck-at-l. The technique can be easily extended to consider other single stuck-at faults such as stuck-at-Z(high impedance). Gate level fault models have been built for most commonly used gates. The ability to model the fault behaviour of functional blocks in ADLIB-SABLE is also demonstrated. The motivation is that for many functional blocks, a gate level description may not be available or that the designer wishes to sacrifice detailed analysis for a higher simulation speed. Functional fault models are built for many commonly used blocks, using a decomposition technique. The ratio of functional fault simulation speed to gate level fault simulation speed has been observed to be of the order of 5 for the typical functional block sizes considered. The ratio however, is not the upper limit and will be larger for larger-sized functional blocks. It was also proved that the functional fault models are invariant with respect to the internal implementation details. A design discipline for sequential circuits is worked out which allows deductive fault simulation. Extensions to the simple [0,1] deductive technique are studied and fault models built in the extended domain are observed to be useful in modelling gates of some technologies. A comparison between deductive and concurrent fault simulation methods is given. Performance of deductive fault simulation, implemented in ADLIB-SABLE, shows that for sequential as well as combinational circuits, the CPU time increases linearly with increasing number of components simulated, an advantage over fault simulators which simulate one fault at a time and display a quadratic behaviour.

Key Words and Phrases : Fault Simulation, Fault Models, Deductive

[1]***This work was supported by the ARMY RESEARCH OFFICE CONTRACT NO DAAG-29-80-K-0046***

Table of Contents

ACKNOWLEDGEMENTS

The first author is extremely grateful to Prof VanCleemput and Prof Dutton for their very helpful suggestions,guidance and encouragement. Sincere thanks are also expressed to Warren Cory of Computer Systems Laboratory and Roy Russo of IBM Yorktown.

INTRODUCTION

SABLE is a simulation system developed and currently in use at Stanford University[1,8]. It supports structured multi-level simulation of digital designs. The user expresses interconnectivity via SDL (Structured Design Language) and the component behaviour is described in a higher level language ADLIB (A Design Language for Indicating Behaviour) which is a superset of Pascal. The SABLE sirnulation system uses SDL to create as many instances of components as required and then performs simulation.

Deductive fault simulation has been implernented in ADLIB-SABLE at both the gate and the functional level. This technical report is divided into seven chapters, each discussing a particular aspect of fault simulation. The first chapter discusses the basics of SABLE and deductive fault simulation. The next chapter on gate level deductive fault sirnulation explains the basic data structures and the fault simulation programs. The third chapter discusses techniques to pre-process sequential circuits before they can be fault simulated. Functional fault modelling is discussed in the fourth chapter where several functional fault models are given. Chapter 5 talks of multi-level fault simulation and extensions to the sirnple [0,1] Deductive technique. This chapter also contains a comparative study between deductive and concurrent techniques. Chapter 6 discusses the performance results of deductive fault simulation implemented in ADLIB-SABLE and other fault and logic simulators.

Chapter 1

Basics of SABLE,Deductive Fault Simulation

1 .1. Basics of SABLE

The two important inputs to the SABLE system are the ADLIB and SDL descriptions. The ADLIB source contains the bchavioural description of the different component types such as counters,multipliers,2 input And gates, invertors etc. There is no way of telling from the ADLIB code how many instances of each component type there will be. The definition of the logical behaviour of a particular component is called the "comptype". A comptype written in ADLIB is a specification of the input to output relationship of the particular type of component the cornptype is representing. Essentially all information that passes between components must go through well-defined I/O interfaces called 'nets'. The SDL source contains the interconnectivity between the different components. In addition, the SDL description also tells how many of each of the components are present. Thus the SDL description is specific to a particular circuit. Fig 1-l shows a global flow chart of the ADLIB/SABLE system.

First the ADLIB compiler reads in the ADLIB source,checks the syntax and then generates a PASCAL source. It also generates a LIBRARY file which contains a library of the comptypes, the nets associated with each comptype and any formal parameters that may be associated with a comptype. The SDL compiler reads in the SDL source and the LIBRARY,checks the syntax and then generates a TOPOLOGICAL file. This file essentially contains all the information (the type and number of components,interconnection and any actual parameters to be passed to the comptypes) to be later used by the SABLE system to communicate between various components through nets, **The** PASCAL generated before is linked with an ASSIST package which is the core of the SABLE simulation package(it contains the event scheduler) and the simulation is ready for execution.

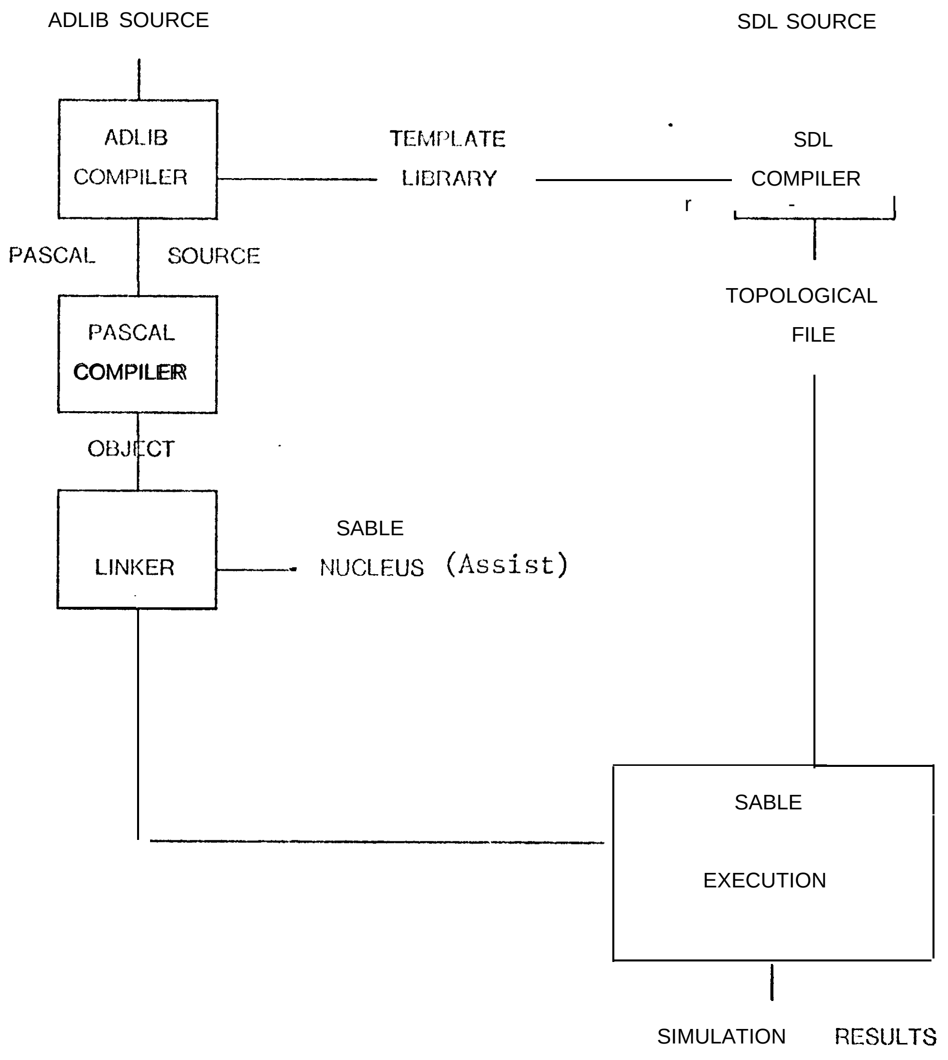


FIG 1-1: SABLE SIMULATION SYSTEM

1.1 .1. An Example

A small example is presented to illustrate how to use ADLIB/SABLE. A system is defined to consist of five basic types of units,an INPUT-PROCESS,a 2 bit AND gate,a 2 bit NOR gate,an INVERTOR and an OUTPUT-PROCESS. The INPUT-PROCESS and OUTPUT-PROCESS are artifacts of SABLE; it is a way of introducing external inputs to the simulation environment and obtaining outputs from the simulation to the external world. Fig l-2 shows the actual structure using components of the types mentioned before. Figs l-3 through 1-6 contain the ADLIB source code for each of the component types and Fig 1-7 describes the SDL description for the example.

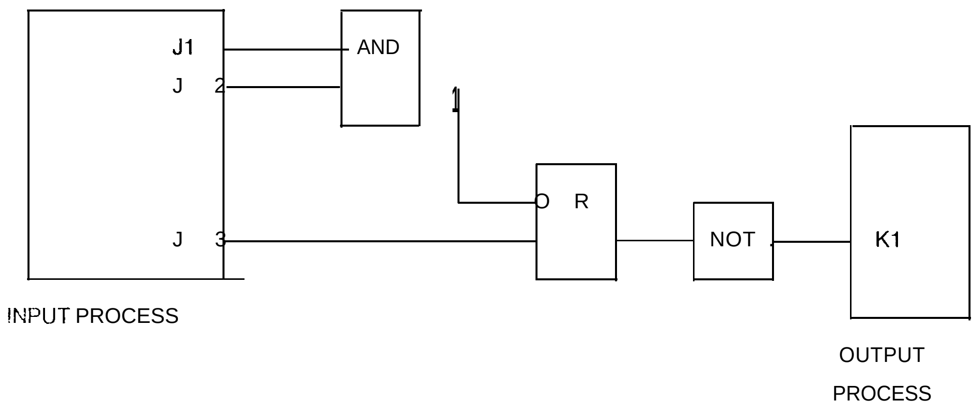


FIG 1-2: AN EXAMPLE

```
ADLIB CODE:
Comptypc  INPUT-PROCESS;
Outward j1,j2,j3:intnet;
Var  x:integer;
Begin
 While true do begin
    readln(x);assign x to jl;
      readln(x);assign x to j2;
    readln(x);assign x to j3;
   waitfor true delay 1.0;
   end;
End;

Comptype  OUTPUT-PROCESS;
Inward k1 :intnet;
Begin
  While true do begin
   Waitfor true check kl ;
     writeln(tty,kl);
   end;
End;
```

Figu re 1- 1: Comptypes INPUT-PROCESS,OUTPUT-PROCESS

```
Comptype  INVERTOR;
Inward in1 :intnet;
Outward   out:intnet;
Var tl :integer;
BEGIN
 While true do begin
   Waitfor true check in1 ;
   if in1 = 0 then tl: = 1 else tl : = 0;
   Assign  tl  to  out;
   end;
END;
```

Figure 1-2: Comptype Invertor

```
Comptype AND2;
Inward in1,in2:intnet;
Outward out:intnet;
Var t1:integer;
Begin
  While true do begin
    Waitfor true check in1,in2;
    if in1 = 1 and in2 = 1 then tl: = 1
               else t1 : = 0;
    assign t1 to out;
    end;
END;
```

Figure 1-3: Comptype AND2

```
Comptype OR2;
Inward in1,in2:intnet;
Outward out:intnet;
Vnr t1:integer;
Begin
  While true do begin
  Waitfor true check in1,in2;
  if in1 = 1 or in2 = 1 then tl: = 1
             else tl : = 0;
  assign tl to out;
  end;
END;
```

Figu re l-4: Comptype OR2

```
SDL CODE:
Name:Test;
Types:INPUT-PROCESS,INVERTOR,AND2,OR2,OUTPUT-PROCESS
INPUT-PROCESS:inp;
 OUTPUT-PROCESS:ou tp;
  INVERTOR:inv;
 AND2:and;
OR2:or;
end;
netsegment;
net1 = inp.j1,and.in1;
net2 = inp.j2,and.in2;
net3 = inp.j3,or.in2;
net4 -- and.out,or.in1;
net5 = or.out,inv.in1;
net6 = inv.out.outp.k1;
endnets;
```

Figure 1-5: SDL Code

The 'inp' is the process that reads values of j1,j2,j3 from an input file and passes them over to the 'and' and 'or'. It then goes to sleep for 1 .0 time units. None of the other components are activated yet since none of them have all their inputs assigned yet. After 'inp' has assigned values to j ! ,j2,j3, 'and' starts to work followed by 'or' , 'inv' and finally 'outp'. 'outp' directs whatever value it gets to the terrninal screen and then goes to sleep. 'inp' wakes up again and a fresh set of values are directed to its outward nets. The same story repeats and thus a continuous stream of either 0 or 1 is seen on the terminal screen. The reader is requested to bear this example in mind for it will be referred to while discussing deductive fault simulation.

1.2. Basics of Deductive Fault simulation

Logic circuits are prone to faults of various types. The types of faults encountered are stuck-at faults,bridging faults and delay faults. This report will restrict to only single stuck-at faults. First,because this problem can be handled reasonably well. Second,many complex faults can be inferred from the stuck-at fault pattern. Single stuck-at faults imply that for a component, only One faulty node will be considered at a time unless there is reason to do otherwise. 'This special reason will be discussed in chapter 5. The idea behind fauit simulation is as follows : given an input pattern (test pattern),one is interested in finding out which of the stuck-at faults,considered one at a time, will render the output(s) faulty. For,these are the stuck-at fauits that can be detected with the test pattern.

There are four types of fault simulation techniques that are generally used : One fault at a time,Parallel,Deductive and Concurrent[4]. The Deductive technique is of particular interest to us in the context of ADLIB-SABLE system. It will be later shown in chapter 5 that it is also possible to implement Concurrent fault simulation in ADLIB-SABLE.

Deductive fault simulation has been defined in a variety of ways by different authors[13]. The next section will provide an example to illustrate the basic concepts.

1.2.1. An Example

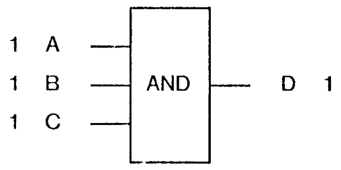


FIG I-8: 3 INPUT AND GATE

Consider a very simple circuit,a 3 input AND gate(Fig I-8). The inputs are a,b,c and the output is d. The possible stuck-at faults are a(s-a-0),a(s-a-1),b(s-a-0),b(s-a-1),c(s-a-0),c(s-a-l),d(s-a-0) and d(s-a-l). Also let the inputs be a = 1 ,b = 1 and c = 1. If there are no faults,then d = 1. However,if there is a fault in the circuit leading to a faulty output 0 then it could be either of ‘d’ stuck-at 0 or ‘a’ stuck-at 0 or ’b’ stuck-at 0 or ’c’ stuck-at 0. Mathematically written,F(d) = d(s-a-O)Va(s-a-O)Vb(s-a-O)Vc(s-a-O), where F(d) is called the fault list at node ‘d’ for single stuck-at faults.

If this gate had been a part of a large circuit with its inputs from other gates,then each of the nodes ’a’,’b’,’c’ will have fault lists associated with them. Thus F[a] will have all the faults from previous nodes that will lead to a faulty value for ’a’,namely 0. Similarly there will be fault lists F[b] and F[c] which will have a list of faults from previous nodes that will cause errors in the value of nodes ‘b’ and ‘c’. Then F[d] = d(s-a-O)VF[a]VF[b]VF[c] and this will be a longer list when expanded. The fault list so obtained is input dependent and is only valid for single stuck-at faults.

1.2.2. More Examples

The technique has just been shown to work for a 3 input AND gate. In this subsection, it is applied on OR and NOT gates. First the 'OR' gate as shown in Fig 1-9.

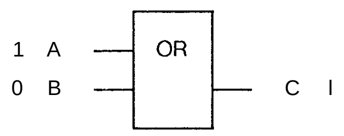


FIG 1-9: 2 INPUT OR GATE

Here,the inputs are a,b and the output is c. Also let the inputs be a = 1 ,b = 0. Now,the correct output for this input is 1. If the output is 0 then it is due to a fault 'c' stuck at 0. Also if 'a' is stuck at 0 then the output is in error. Thus the fault list is *F(c) = c(s-a-0)Va(s-a-0)* where F(c) is the fault list for single stuck-at faults. If however,this OR gate is a part of a large circuit and its inputs are from other gates then the fault list is obtained as follows. There will be fault lists associated with the input nodes 'a' and 'b', namely F[a] and F[b]. All faults in F[a] lead to a value 0 for 'a' and all faults in F[b] lead to a value 1 for 'b'. Now,all faults in F[a] which are not occuring in F[b] will cause faulty output. Thus the final fault list is F[c] = c(s-a-0)V{F[a]-F[b]}.

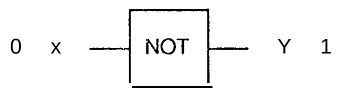


FIG 1 - 10: AN INVERTOR

Here,the input is x and the output is y. Let the input x be 0. In an analogous way, the fault list is *F(y) = y(s-a-0)Vx(s-a-1)* for single stuck-at faults. If this is a part of a large circuit,as before,then F[y] = F[x]Vy(s-a-0).

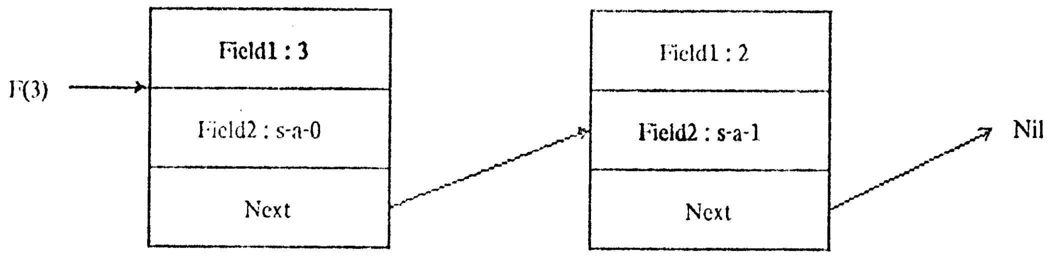
Chapter 2
GATE LEVEL FAULT SIMULATION

2.1. The Mechanism and Flow Control

Observe in section 1.2 of chapter 1, how the fault lists F(a),F(b),F(c) were merged with d(s-a-0) to create the Final fault list F(d). This selective merging of fault lists, for a given input pattern, will be called as the fault list marching from left (input end) to the right (output end). Fault simulation of a circuit,for a given input pattern is complete when the fault list for the last output node(s) is created and complete.

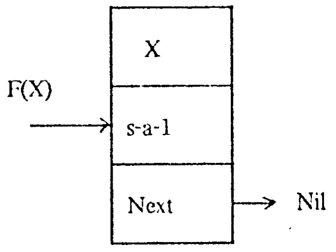
We term the entire art of fault list marching as flow control in the "deductive fault simulation" terminology. This will reccur in fault simulation for sequential circuits and functional fault simulation. The data structures and the procedures employed are described in Figs 2-l through 2-4.

Every net in a circuit has an unique integer number. A net may be interconneting several nodes of several components and will have one logical value at any time. The fault list is a linked list of records. Each net will have its own fault list denoted by a global array variable F. A net with number 3 will have a fault list F(3) where F(3) is actually the header of the linked list. Each entry in the linked list describes a net (by its identifying integer number) and the nature of the stuck-at fault. Thus F(3) may have two entries as shown below.

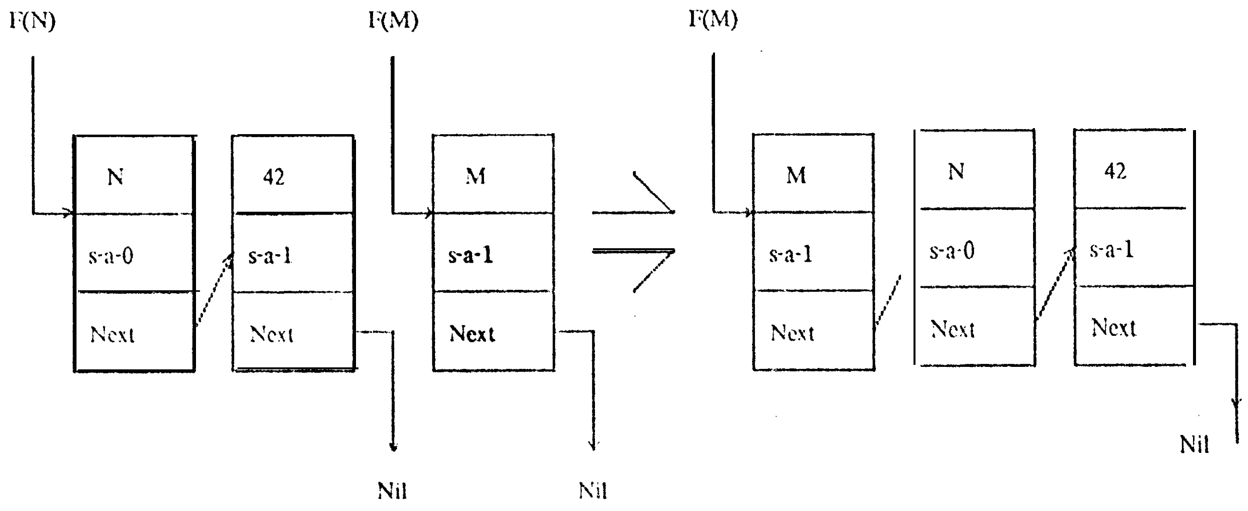


This means net 3 has a correct value of 1 and will be faulty if either net 3 is stuck-at logical 0 or net 2 is st uck at logical 1. Field1 of the record can be any valid integer while field2 can be either of s-a-l or s-a-0.

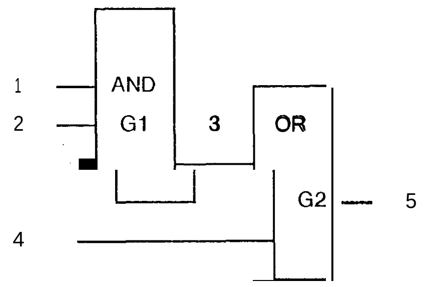
Now that F(n) and F(m) are known the three operations performed by the three procedures can be explained. They are Create-list, Merge-lists and Print-out. Print-out types out the contents of the fault list of a net and takes in the net number as an argument. If a net 'X' has a non faulty value 0 and is encountered for the first time in simulation, then a fault list F(X) is created by Create-fist with the following entries :



The procedure Merge-lists takes in two arguments m,n and it merges the contents of the fault list pointed at by the header F(n) to the fault list F(m). The result of merging is pointed at by the header F(m) and is shown below.



In the implementation on the Dee-20 system at Stanford,the net numbers are assigned as parameters from the SDL description. For the circuit shown below, the SDL description would be :



AND2 : G1(I1 = "1",I2 = "2",O1 = "3");
AND2 : G2(I1 = "3",I2 = "4",O1 = "5");

These actual parameters are passed by SABLE to the comptype AND2 which has i1, i2 and 01 as formal parameters. The Adlib code would be like :

```
Comptype AND2(i1 ,i2,o1 :integer);
Begin

End;
```

The codes for the data structures and the procedures are given in the figures below.

```
Const
 u = 0;one = 1 ;zero = -1;
Type
 Intnet-vat = (u,one,zero);
 s-a = (s-a- 1 ,s-a-0);
 q = record
    field 1 :O. .99;
     field2:s-a;
     next:↑q;
    end;
 ptr = ↑q;
Nettype
 Intnet = Intnet-val;
Var
F:array[1..100]of ptr;
p:ptr;z:integer;
```

Figw re 2- I: Data Structures

```
Procedure  Create-list(m:integer;zl  :s-a);
Begin
 If F[m]<>nil then writeln('Fault list already created')
 Else if F[m] = nil then begin
 new(p);
  p↑.field1: = m;p↑.field2: = zl ;
  pt.next: = F[m];F[m]: = p;
 end;
End;
```

Figure 2-2: Procedure : Create-list

```
Procedure   Print-out(y:integer);
(*Printing out is enabled at user request*)
Var x:ptr;
Begin
 IF trace = true then Begin
   if F[y]<>nil then begin
     x: = F[y];
     writeln(tty,'y = ',y);
     while x<>nil do begin
     writeln(tty,xt.fieldl  ,x↑.field2);
     x: = xt.next;
    end;
writeln;
end; end;
End;
```

Figure 2-3: Procedure : Print-out

```
Procedure   Merge-lists(m,n:integer);
Var w:ptr;
Begin
 w: = F[m];
 while wt.next<>nil do begin
     w: = w↑.next;
     end;
 w↑.next: = F[n];
End;
```

Figure 2-4: Procedure : Merge-lists

At this stage consider the example in Fig 1-2 in chapter 1 and march **the fault lists** from left to right. The figure is drawn again (Fig 2-6) this time without the OUTPUT-PROCESS but all the nets given unique integer numbers. The inputs are 1,1 ,1 for jl ,j2 and j3 assigned by the input process which does it only once. The fault lists in the sequence they are created are shown in the figure Fig 2-5 below.

FAULT LISTS :
F(2) = 2(s-a-0)
F(3) = 3(s-a-0)
F(5) = 5(s-a-0)
F(4) = 4(s-a-0)VF(2)VF(3) = 4(s-a-0)V2(s-a-0)V3(s-a-0)
F(6) = 6(s-a-0)
F(7) = 7(s-a- 1)VF(6) = 7(s-a- 1)V6(s-a-0)

FIG 2-5

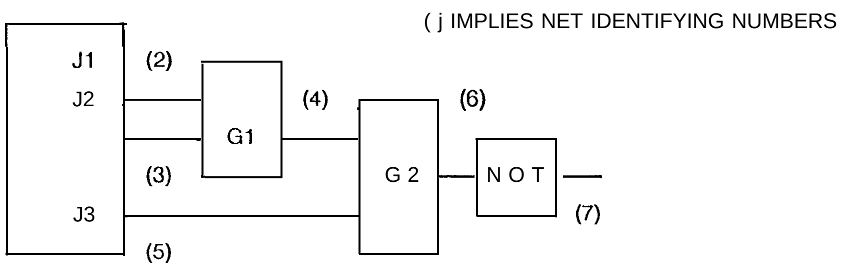


FIG 2-6 : EXAMPLE

2.2. A Fault Model for a 2 input And Gate

For all the fault models that will be described in the remaining portion of this report the following value system will be used, unless otherwise stated. 'zero' indicates logical 0, 'one' indicates logical 1 and 'u' denotes uninitialized value. At the start of simulation in ADLIB-SABLE, ail nets are initialized to 'u'.

```
Comptype AND2(i1,i2,o 1 :integer;delay1:real);
Inward in1,in2:intnet;
Outward   out:intnet;
Var  temp:intnet;
Begin
  Waitfor (in1 <> u) and (in2 <> u) check inl, in2;
  If ((in1 = zero) or (in2 = zero)) then temp: = zero
                        else temp: = one;
   if in1 = zero then create-list(i1 ,s-a-l) else
                        create-list(i1  ,s-a-0);
    if in2 = zero then create-list(i2,s-a-1) else
                       create-list(i2,s-a-0);
      if temp = zero then create-list(o1 ,s-a-l) else
                         create-list(o1 ,s-a-0);
         if (in1 = zero) and (in2 = zero) then
                         assign zero to out delay delay1
      else if ((in1 = one) and (in2 = zero)) then begin
                    merge-lists(o1 ,i1);
                     assign zero to out delay delay1 ;
                    end
    else if (in1 = zero) and (in2 = one) then begin
                    merge-lists(o1  ,i2);
                     assign zero to out delay delay1 ;
                    end
  else begin
      merge-lists(o1 ,i1); merge-l.ists(ol ,i2);
       assign one to out delay delay1 ;
      end;
End;
```

Fig 2-7 : Comptype AND2

The code (Fig 2-7) has the following effect on the AND gate. Until both input nets in1,in2 are assigned a value either 0 or 1 the comptype is not awakened (initiating execution). This ensures that a gate is fault simulated only when it is its turn, that is, all the inputs are assigned. Once both nets in1,in2 are assigned then, depending on the input values, fault lists F(i1),F(i2) are created using the procedure Create-list. The value of the output is evaluated using the input output relationship and the fault list F(o1) is created. Now, based on the input pattern and remembering that only single stuck-at faults are detected, Merge-lists is utilized to generate the final fault lists. If in1 = 1 and in2 = 1 then the model performs Merge-lists(i1,o1) and Merge-lists(i2,ol). The fault list F(o1) is now complete and the value of the output is assigned to the output net so that subsequent gates can be fault simulated. In the case, where the gate is a part of a larger circuit and thus fault lists F(i1),F(i2) already created, the code (Fig2-7) writes out the output fault list at node 01.

2.3. Additional Fault Models

The fault models for an INVERTOR(Fig 2-9) and a 2 input OR gate(Fig 2-10),in ADLIB-SABLE,are given below in Figs 2-8 and 2-l 1 respectively. They use the same philosophy as the comptype AND2 described in the earlier section.

```
Comptype INVERTOR(i1 ,ol :integer;delay1 :real);
Inward in1 :intnet;
Outward  out:intnet;
Var  temp:intnet;
Begin
  Waitfor in1 <> u check inl;
   if in1 = one then temp: = zero
         else temp: = one;
    if in1 = zero then create-list(i1 ,s-a-l) else
                 create-list(i1  s-a-0);
      if temp = zero then create-list(o1 ,s-a-1) else
                   create-list(o1  s-a-0);
 merge-lists(o1  ,i1);
   if in1 = zero then assign one to out delay delay1
      else assign zero to out delay delay1 ;
End;
```

Fig 2-8 : Comptype INVERTOR

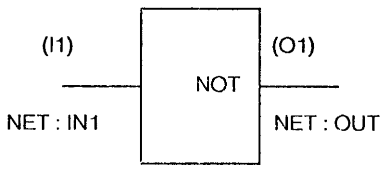


FIG 2-9: INVERTOR

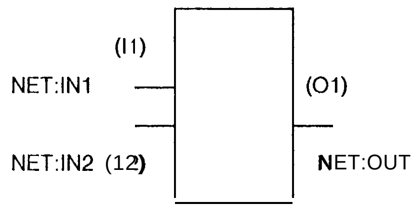


FIG 2-10: OR2

```
Comptype OR2(i1 ,i2,o1 :integer;delayl :real);
Inward in1,in2:intnet;
Outward   out:intnet;
Var  temp:intnet;
Begin
   Waitfor (in1 <> u) and (in2 <> u) check inl, in2;
   If (in1 = one) or (in2 = one) then temp: = one else temp: = zero;
    if in1 = zero then create-list(i1 ,s-a-l) else
                      create-list(i1 s-a-0);
      if in2 = zero then create-list(i2,s-a-1) else
                      create-list(i2,s-a-0);
      if temp = zero then create-list(i3,s-a-1) else
                      create-list(i3,s-a-0);
    if (in1 = zero) and (in2 = zero) then
           begin
            merge-lists(o1,i1);
             merge-lists(o1,i2);
            assign zero to out delay delay1 ;
           end
    else if (in1 = one) and (in2 = zero) then
           begin
             merge-lists(o1,i1);
             assign one to out delay delay1 ;
           end
    else if (in1 = zero) and (in2 = one) then
           begin
             merge- lists(o1,i2);
             assign one to out delay delay1 ;
           end
    else assign one to out delay delay1 ;
End;
```

Fig 2-l 1 : Comptype OR2

2.4. An Example

Fig 2-12 shows how to build a fault simulation to fault simulate the circuit in example Fig l-2 in chapter 1.

```
-----------
EXAMPLE.ADL
-----------
Comptype AND2(i1,i2,o1:integer;delay1:real);
.....
Begin
....
End;

Comptype OR2(i1 ,i2,o1 :integer;delay1:real);
.....
Begin
.....
End;

Comptype INVERTOR(i1,o1:integer;delay1:real);
....
Begin
......
End;

Comptype  Input-Process; ! This assigns inputs to the circuit !
Begin
 Assign one to jl; } Logical 1 values assigned to
  Assign one to j2; } all three inputs.
 Assign one to j3; }
End;
```

```
----------
EXAMPLE.SDL
----------
Name:Test;
Types
 AND2:and(i1 = "1 ",i2 = "2",o1 = "4");
 OR2:or(i1 = "4",i2 = "3",o1 = "5");
 INVERTOR:inv(il = "5",o1 = "6");
 INPUT-PROCESS:inp;
End;
Netseg  ment
  net1 = inp.jl ,and.in1;
  net2 = inp.j2,and.in2;
  net3 = inp.j3,or.in2;
  net4 = and.out,or.in1;
  net5 = or.out,inv.inl ;
Endnets;
```

Fig 2-12 : ADLIB,SDL Codes for the Example.

For the fault simulation example just described, the trace of the creation and merging of the fault lists in the sequence they occur, is given below.

1. Step1 : F(2) = 2(s-a-0); F(3) = 3(s-a-0);
2. Step2 : F(4) = 4(s-a-0);
3. Step3 : F(4) = F(4) V F(2) = 4(s-a-0) V 2(s-a-0);
4. Step4 : F(4) = F(4)VF(3) = 4(s-a-0) V 2(s-a-0) V 3(s-a-0);
5. Step5 : F(5) = 5(s-a-0);
6. Step6 : F(6) = 6(s-a-0);
7. Step7 : F(7) = 7(s-a-1);
8. Step8 : F(7) = F(7) V F(6) = 7(s-a-1) V 6(s-a-0);

The final complete fault list is F(7) = 7(s-a-1) V 6(s-a-0).

Chapter 3
Processing Sequential Circuits

3.1. Realizing the problem

The previous chapter discussed deductive fault simulation on combinational logic circuits. Unfortunately when applied to circuits with feedback loops the technique of chapter 2 fails. In the following example it will be shown how the technique fails and the rest of the chapter discusses ways to reasonably fault simulate sequential circuits.

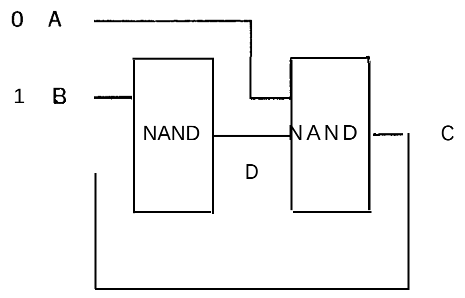


FIG 3-1: CIRCUIT WITH FEEDBACK

In the above figure, NAND gates G1,G2 are configured in a way that output node ‘d’ of G1 is an input node of G2 and the output node ‘c’ of G2 is an input node of G1. Input nodes ‘a’ and ‘b’ are assigned values externally. The irnplementation as explained in chapter 2 requires that at least one of the two gates G1,G2 have both its inputs assigned first. Until that happens fault simulation cannot begin. Here, neither G1 nor G2 qualify and thus fault simulation fails.

The reason fault simulation failed in the previous example is because of the closed loop 'cd'. Elements in the closed path will have values either stable or changing when power is switched on. Looking at the circuit one can tell that net 'd' can have value 1 or 0 (consequently net 'c' can be 0 or 1) but one cannot be more precise. Since fault simulation is dependent on the input values, one has to be precise about the value of the nets 'c' and 'd'.

3.2. A Solution

The solution proposed can be described as follows. The feedback loop is identified first. Then it is broken at any point by the user and a delay element is inserted which carries with it an initial value. An example would like this :

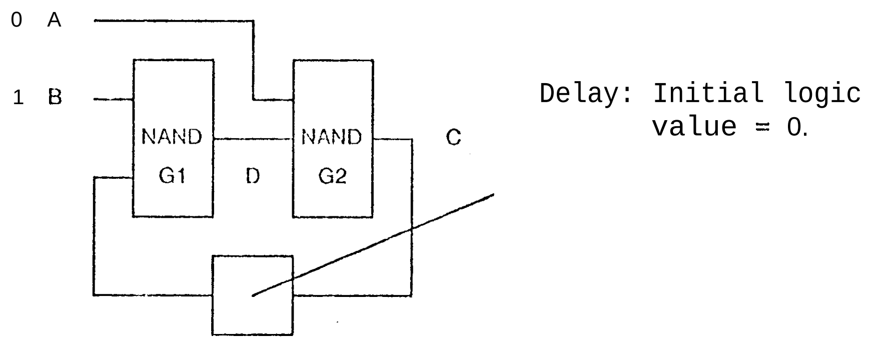


FIG 3-1B : CIRCUIT WITH FEEDBACK BROKEN

Let the initail logic value of the delay be 0. This initial value could be either 0 or 1 and there is no way of telling unless there is some mechanism which ensures a particular value always at power up or there is a reset button. With the initial value, gate G1 has both its inputs assigned and is simulated first. Hence F(b) = b(s-a-0); F(c) = c(s-a-1); F(d) = d(s-a-O)VF(c) which can be expanded as F(d) = d(s-a-0)Vc(s-a-1). Next gate G2 gets simulated and fault lists are : F(a) = a(s-a-1); F(k) = k(s-a-O)VF(a) = k(s-a-0)Va(s-a-1).

If the implementation discussed in the previous chapter was followed, then sirnulation would have stopped and concluded that F(k) was complete. However in this case fault simulation is not yet complete. Once gate G2 is evaluated, the new value of k would be 1. This is different from the previous initial value 0 and suggests that the results have not yet stabilized. Until the values in the different nets of the loop stabilize, fault simulation should be continued. The next cycle of fault simulation on the above example is continued. F(c) = c(s-a-0); F(b) = b(s-a-0); F(d) = d(s-a-1)VF(c)VF(b) which is rewritten as F(d) = d(s-a-1)Vc(s-a-O)Vb(s-a-0); F(k) = k(s-a-0). The value of 'c' is 1 and hence the next value of k would be 1. This was also the last time value of k and suggests stability has set in. A third simulation pass must be carried out to verify stability. Observations show that sequential circuits with a single loop stabilize in two iterations if they will stabilize at all. Some circuits like the 3 invertors arranged in a loop will never stabilize since it is a deliberate oscillator. Such cases are detected and terminated . That is, all the components in the loop are no longer executed. Observations also indicate that the fault lists do not depend on the location of the delay.

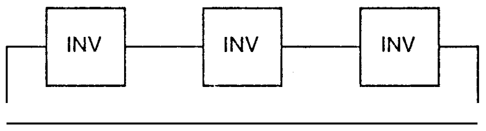


FIG 3-2 : AN OSCILLATOR

The method of preprocessing sequential circuits to enable fault simulation requires that 1) All feedback loops be identified. 2) Delays are inserted at proper places by the user. 3) Many (three for a single loop) simulation cycles are to be executed on all the components chained in the feedback loop.

These three requirements reveal some difficulties and will be discussed. In large sequential circuits, there may be a large number of loops and the number of components chained in a loop may be large. Requirement 3 indicates that the CPU time taken to fault simulate such cases may be significant.

The reason one needed to perform more than on& simulation pass is because one was not aware of the correct value of the nets in the feedback loop. If it could be insisted upon that sequential circuit designs have identifiable RS-latches so that there is some mechanism which initializes feedback loops, then the problem would be greatly simplified. Sequential circuits could then be simulated like combinational ones. Such an insistence may not be unrealistic or too demanding in many technologies and in system level designs. However, one can imagine certain scenarios where this is not obeyed. For those cases the straigtforward method discussed in the begining of this section can be applied provided that the circuits are reasonably small. In TTL technology initialization of feedback loops in sequential circuits is done by either hitting a clear button or by designing them such that they change only on **a** clock edge. Thus during the low state of the clock, the feedback loops get initialized and then when the clock goes low, the circuit is activated. Based on the assumption that there are identifiable RS-latches, a fault simulation technique is discussed next.

3.2.1. Circuits with RS latches breaking loops

Assume that all the feedback loops have been broken by RS-latches. During the initialization phase, the feedback loops and thus the Q,QP nodes of each RS-latch get some value assigned. On the arrival of the active edge fault simulation is performed as well as evaluation of the circuit.

A RS-latch in a sequential circuit will be treated as a single component. That is, given the R,S inputs and the present state of the Flipflop(Q,QP values) new values for Q, QP and the fault lists F(Q),F(QP) can be generated. How one obtains the fault lists will be discussed in section 3.3. Using the RS-latch fault model and the fault model for 2 input AND gates shown in chapter 2, a small sequential circuit will be fault sirnulated next.

In the example of figure 3-3, observe that the feedback loops have been broken by a single latch. There are two gates G1 ,G2, a latch and to external inputs 'x' and 'y'. As pei the model,

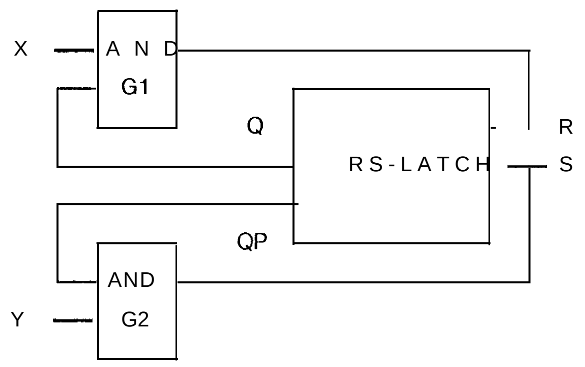


FIG 3-3: A SMALL SEQUENTIAL CIRCUIT

initial values are assigned to Q and QP nets. In this particular case, the initial values are given by the user. In other cases, the initialization input pattern itself can do the job. Once, initial values are assigned to nets Q and QP, the gates G1,G2 can be fault simulated simultaneously. Following them, the RS-latch will be fault simulated to yield the complete fault lists. Let the inputs to 'x' and 'y' be 1 and 0 while Q = 1 and QP = 0. The fault lists are : F(R) = R(s-a-0)Vx(s-a-O)VQold(s-a-0); F(S) = S(s-a-1). The values of R and S are 1 and 0. With these values and the Q,QP values, the final state of the RS-latch as well as the fault lists for nets Q,QP are evaluated. The fault lists so obtained may contain stuck-at faults for Qold,QPold nets. This means that the initialization is faulty. And a faulty initialization is the result of some faults that occured in the circuit when the previous input pattern was applied.

The above method of fault simulating sequential circuits is quite sirniiar to the technique for simulating combinational circuits with the exception of identifying latches and assigning initial values. It is also quite practical timewisc as shall be seen in chapter 7. Since the discipline of breaking feedback loops by RS-latches is enforced on designs, the location of the latches is known at design time. The extra burden of identifying feedback loops at a later time is avoided.

3.2.2. The general case

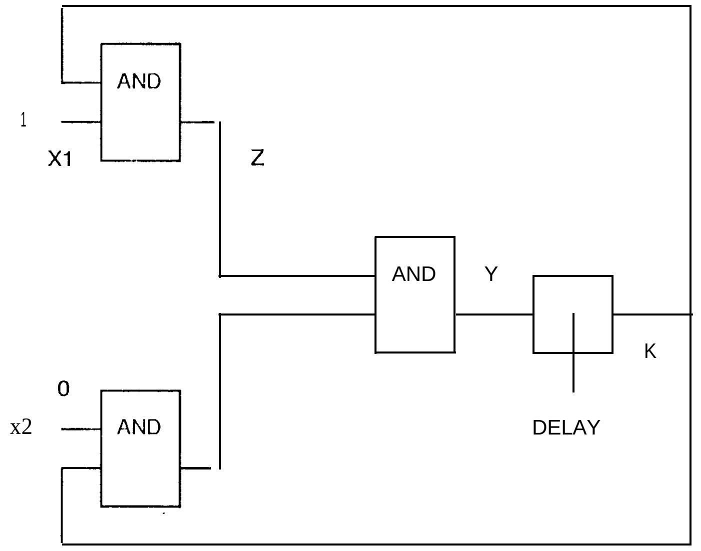
The method just described in subsection 3.2.1 is quite useful yet there is the need to fault simulate the RS-latch itself at the gate level and other small circuits that do not have feedback loops broken by latches. To fault simulate such circuits the technique discussed in the begining of section 3.2 will be used. Some issues are :

1. Develop an algorithm to enumerate the minimum set of edges which when removed renders the remaining circuit acyclic.
2. What is the difference if the delay is placed in one arm of a feedback loop as opposed to any other.
3. If there are two or more loops that share a common edge, should it make a difference if one delay was used at the common edge as against two delays at two other places in the two loops.

The minimal set of edges whose removal makes the remaining circuit graph acyclic is termed as minimum-feedback arc set in the literature. Deo[17] describes two algorithms that can extract the minimal set. However both of the algorithms reported are computationally complex.

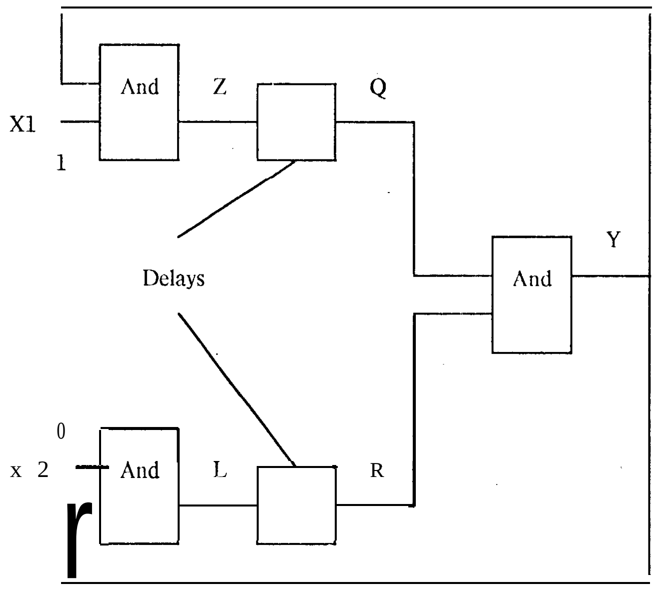
To answer the last two questions, consider an example where there are two loops sharing a common edge. First time simulation will be done using one delay on the common edge. Second time, simulation will be done using two delays inserted in two different arms of the two loops. Each case is shown in the two figures 3-5a and 3-5b below along with the fault lists.

The fault lists in both the cases are the same, but that does not imply that the answers to questions 2 and 3 are universally no. For the same circuit, if 'xl' and 'x2' are both 1, different fault lists are obtained by placing delay units arbitrarily and assigning arbitrary initial values. The reason different fault lists are obtained, for the above case lies in the fact that the loop can be stable in either a 0 or a 1 state. The different fault lists correspond to the different initial values of the feedback loop. This is no error. In another simulator, SALOGS[18], sequential circuits with feedback loops are fault simulated in the following way. None of the feedback



LET INIT VALUE OF K BE 0
Z = O,F(Z) = Z(S-A-1)
L = O,F(L) = L(S-A-1)
Y = O,F(Y) = Y(S-A-1)
NEXT STEP,K = 0
STABILISED
Y = K
F(K) = F(Y) = Y(S-A- 1)

FIG 3-5A: DELAY INSERTED AT ONE ARM



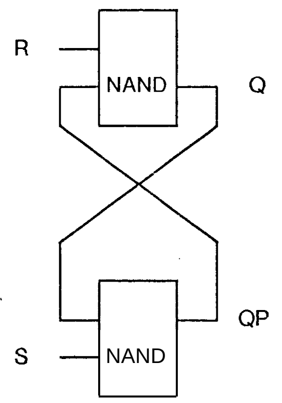
LET INITIAL LOGICAL VALUES
OF Q AND R BE BOTH 0
Y = O,F(Y)= Y(S-A-1)
Z=O,F(Z)=Z(S-A-I)VY(S-A-i)
L=O,F(L)= L(S-A-1)
NEXT STEP
Q = 0.R = 0. STABILISED
Q=Z,R=L
F(Q) = F(Z) = Z(S-A-l)VY(S-A-l)
F(R) = F(L) = L(S-A-1)

Fig 3-5B : TWO DELAYS INSERTED

loops are broken nor are any delays inserted. Any input of a gate that lies in the feedback loop is set to an unknown state. If a gate, on evaluation, has an output of either 0 or 1, then that gate is fault simulated. Otherwise, it is not fault simulated. Thus, for the example in fig 3-5b, for xl = x2 = 1 input set, all of the outputs will be undefined. None of the three gates will be fault simulated. A different approach is adopted in this report. Instead of not fault simulating the gates in this case, the user specifies the initial value of the delay units and the fault lists are generated for that initialization. The user may specify different initial values and thus get a different set of fault lists. The user therefore gets some information rather than nothing at all.

3.3. RS-latch Fault Model

The entire structure of the fault simulation method as described in section 3.2.1 lies in a single assumption : that all feedback loops are identifiable and that there is a fault rnodel for a RS-latch. This section will derive this model. Consider a NOR implementation of the latch and derive the model. A NAND implementation can be handled similarly. Figure 3-6 contains the RS-latch and the truth table. However, the fault model has more extensions built into it than a simple RS-latch. There are two variables Prev-Q and Prev-QP which store the initial values of Q and OP. Four cases will be considered corresponding to the four combinations of R and S inputs and derive the fault list in each case. Note that Q = 1 and QP = 1 cannot be stable values.



R	S	Q	QP
0	1	1	0
1	0	0	1
1	1	0	0
0	0	PREV Q	PREV QP

FIG 3-6 : RS LATCH AND THE TRUTH TABLE

Case 1: R = 0, S = 1. Using the method given in 3.2.2, the fault lists are:

F(Q) = Q(s-a-0)VQP(s-a-1)VR(s-a-l).

F(QP) = QP(s-a-l).

However if S is s-a-0 and Prev-Q = 0 then the new Q value will be 0 and is faulty, Similarly if S is s-a-0 and Prev-QP = 1 then the new value of QP will be 1 and this is faulty too. The final fault lists must read

F(Q) = {F(Q) if Prev-Q = 1
{F(Q)VS(s-a-0) if Prev-Q = 0.

F(QP) = {F(QP) if Prev-QP = 0
{F(QP)VS(s-a-0) if Prev-QP = 1.

Case 2: R = 1, S = 0. Using the method in 3.2.2, the fault lists are:

F(QP) = QP(s-a-0)VQ(s-a-1)VS(s-a-1).

F(Q) = Q(s-a-1).

However if R is s-a-0 and Prev-Q = 1 then the new Q value will be 1 and is faulty. Similarly if R is s-a-0 and Prev-QP = 0 then the new value of QP will be 1 and this is faulty too. The final fault lists must read

F(Q) = {F(Q) if Prev-Q = 0
{F(Q)VR(s-a-0) if Prev-Q = 1.

F(QP) = {F(QP) if Prev-QP = 1
{F(QP)VR(s-a-0) if Prev-QP = 0.

Case 3: R = 1, S = 1. Using the method in 3.2.2, the fault lists are:

F(Q) = Q(s-a-1)VS(s-a-0)

F(QP) = QP(s-a-1)VR(s-a-0)

Case 4: R = 0, S = 0. For this condition the new values of Q and QP are the initialized values Prev-Q and Prev-QP respectively. Only two cases need be considered and the other one is ignored.

Case(i): Prev-Q = 0, Prev-QP = 1; Hence Q = 0, QP = 1.
Using the method in 3.2.2 with the delay element initialized to 0, fault lists are :
F(Q) = Q(s-a-l)VQP(s-a-O)VS(s-a-l)
F(QP) = QP(s-a-0)VQ(s-a-1)VS(s-a-1)

Case(ii): Prev-Q = 1, Prev-QP = 0; Hence Q = 1, QP = 0.
Using the method in 3.2.2 with the delay element initialized to 1, the fault lists are :
F(Q) = Q(s-a-O)VQP(s-a-l)VR(s-a-l)
F(QP) = QP(s-a-1)VQ(s-a-0)VR(s-a-1)

If Prev-Q = 0 and Prev-QP = 0 (which may result from previous R = S = 1) and the new R and S inputs are both 0 then the final stabilized values of Q and QP will be outcome of a race between the two gates. The fault lists will therefore depend on the finai outcome and such faults are termed by Armstrong[13] as star faults. This case is ignored as the fault simulator cannot handle races.

Chapter 4
Functional Level Simulation

4.1. The principle

4.1.1. Introduction

The last two chapters talked about deductive fault simulation of combinational and sequential circuits at the gate level. The time to perform fault simulation of a circuit is a function of the number of components simulated. Hence it would mean considerable savings in time if simulation could be done at a much higher level, whereby a single component represents rnany gates. This is also supported by the fact that, in normal computer circuits, a large number of functional blocks, like adders,counters, multipliers are present. All one is concerned about in a functional block is the input-output relationship, represented either by a boolean equation (or a se&) or by a truth table. In some cases, where the manufacturer supplies only the functional specification and not the gate level description (one such example would be a microprocessor chip), a functional level fault simulation is all that can be performed.

Apart from saving considerable tirne, functional fault models of 'sequential blocks like counters,registers may be much simpler to deal with than at the gate level. A price, associated with this advantage, is the loss of accuracy and detailed information regarding internal faults. The fault lists obtained in functional fault simulation contain no information on the nature of 'faults of the gates that make up the functional block. Fig 4-1 shows the functionai model of a l-bit full adder by a black box with 3 inputs a, b, Cin and 2 outputs s, Cout and the boolean equations described below. The notation '*' means boolean multiplication ' + ' means boolean addition and '-' implies complement. The next subsection shows how to build a fault model for a 1 -bit adder.

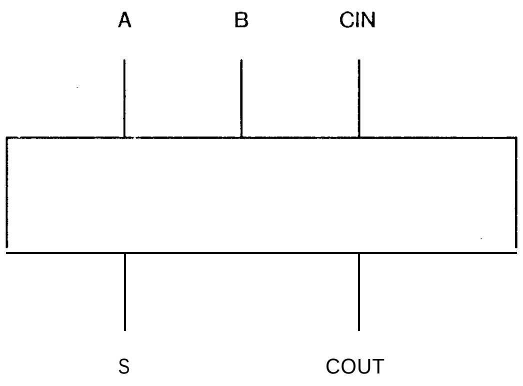


FIG 4-1 : A 1-BIT ADDER

S = Cin * (-A * -B + A * B) + -Cin * (-A * B + A * -B);
Cout = Cin * (A + B) + A * B;

Fig 4-l : A 1 -bit Adder

4.1.2. Building a fault model

In this subsection the functional fault model for a l-bit full adder will be built. Tables 4.1 and 4.2 contain the truth tables for Cout(carry out) and S(sum) outputs. For a given input set of values for Cin,a,b one can read the Cout and the S bit from the tables 1 and 2. Let Cin = 1 ,a = 0,b = 0. At this point recall the data structures and the procedures that were discussed in chapter 2. Fault lists F(Cin) = Cin(s-a-0), F(a) = a(s-a-l) and F(b) = b(s-a-l) will be created. For this input pattern the correct outputs are S = 1, Cout = 0,as read from the tables 1 and 2. The fault lists for these nodes are F(S) = S(s-a-O),F(Cout) = Cout(s-a-l). There are three single stuck-at faults possible at the inputs. They are considered one at a time. If Cin is stuck-at 0 then the faulty input pattern is Cin = 0,a = 0,b = 0 for which Cout is 0 (from table 1) and S is 0 (from table 2). Since Cout stuck-at 0 makes the S output differ from the correct value, it must be included in the fault list for S. Hence F(S) = F(S)VCin(s-a-0). For 'a' stuck-at 1, Cout = 1 and S= 0 and similarly the fault lists are deduced to be F(S) = F(S)Va(s-a-l); F(Cout) = F(Cout)Va(s-a-l). Finally for 'b' stuck-at I the fault lists are as follows. F(S) = S(s-a-O)VCin(s-a-O)Va(s-a-l)Vb(s-a-l); F(Cout) = Cout(s-a-1)Va(s-a-1)Vb(s-a-1). The fault lists are now complete and the fault model demonstrated here is coded in Fig 4-2. In the code, the functions Table1 and Table2 accept 3 arguments namely values of Cin,a,b and return the values of outputs Cout and

CIN	A	B	COUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

TABLE 4.1 : TRU H TABLE FOR THE COUT OUTPUT

CIN	A	B	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

TABLE 4.2 TRUTH TABLE FOR THE S OUTPUT

```
Comptype 1 -bit-adder(i1 ,i2,i3,o1,o2:integer;delay1:real);
(* il is node number for 'a',i2 for 'b',i3 for 'Cin',ol
  for 's' and 02 for 'Cout' *)
Inward  a,b,Cin:intnet;
Outward  s,Cout:intnet;
Var tl ,t2,t3,t4,t5:intnet-val;
Function Table1 (g,h,m:intnet-val):intnet-val;
begin . . . . . . . end;
Function Table2(g,h,m:intnet-val):intnet-val;
begin . . . . . . . end;
Begin
Waitfor (a<>u)and(b<>u)and(Cin<>u) check a,b,Cin;
  tl : = a;t2: = b;t3: = Cin;
  t4: = table1 (tl ,t2,t3);
  t5: = table2(t1,t2,t3);
  if tl = zero then create-list(i1 ,s-a-l)
        else create-list(i1,s-a-0);
  ................
  if t5 = zero then create-list(o2,s-a-1)
      else create-list(o2,s-a-0);
  if (tl = zero) and (t4<>table1(one,t2,t3))
     then merge-lists(o1 ,i1)
  else if (tl = one) and (t4<>table1(zero,t2,t3))
     then merge-lists(o1 ,i1);
  if (tl = zero) and (t5<>table2(one,t2,t3))
     then merge-lists(o2,i1)
  else if (tl = one) and (t5<>table2(zero,t2,t3))
     then merge-lists(o2,o1);
  ..........................
  if (t3 = zero) and (t5<>table2(t1,t2,one))
     then merge-lists(o2,i3)
  else if (t3 = one)and(t4<>table1(tl ,t2,zero))
     then merge-lists(o2,i3);
Assign t4 to s delay delay1 ; Assign t5 to Cout delay delayl;
End;
```

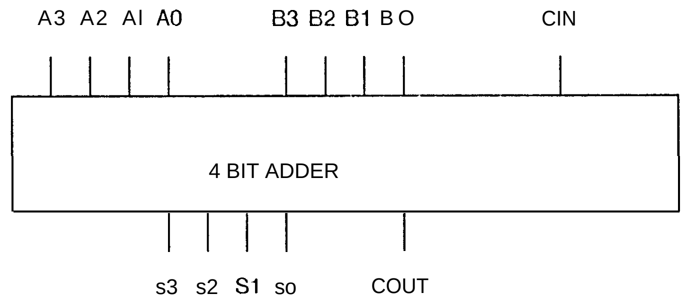
Figure: 4-2 : Comptype 1 -bit Adder

4.2. Several Functional Module Examples

In the last section, a method was shown for building a functional fault model for a l-bit adder. As the reader can see, for a functional block with more number of inputs and outputs, the tables quickly become very large. One can point out that reading the table is equivalent to logically evaluating the outputs using the boolean equations. And with an increasing number of inputs and outputs as is the case with complex functional blocks, the boolean equations become quite complex. The remaining portion of this chapter will discuss elegant methods to modularize complex functional blocks and render fault simulation easy.

4.2.1. Adder Fault Model

For a 4-bit adder (Fig 4-3) functional diagram, the outputs SO through S3,C3 are functions of A0 through A3,B0 through B3 and Cin. The table for 9 inputs would be prohibitively large. Equivalently, the set of boolean equations with some of the equations having 9 variables would be quite complex.



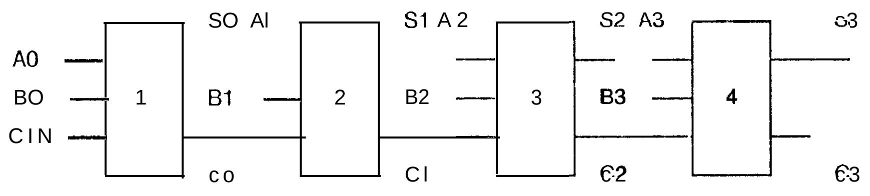


FIG 4-3: DECOMPOSED FOUR BIT ADDER

A technique is adopted whereby the 4-bit adder is functionally decomposed into four l-bit adders which are simuiated one-by-one. The least significant bit adder has the inputs AO,BO,Cin assigned and is simulated first. The values of the outputs S0,C0 and the fault fists F(SO),F(CO) are evaluated in terms of A0,B0 and Cin. CO is the intermediate carry bit and has been created to enable fault propagation down to the next bit adder. The next bit adder is fault simulated next and the fault lists F(SI), F(C1) are generated in terms of AO, Al ,B0,B1 ,Cin. Proceeding in this way fault lists F(S3),F(C3) are generated. If there is a l-bit adder Comptype in the ADLIB source code, then to simulate a 4-bit adder, interconnect four of these using SDL. The SDL code is given below:

```
SDL CODE:
Types:1 -bit-adder;
N1 :1 -bit-adder(cin,aO,bO,sO,cO);
N2:1 -bit-adder(cO,al ,bl ,s1 ,c1);
N3:1 -bit-adder(c1 ,a2,b2,s2,c2);
N4:1-bit-adder(c2,a3,b3,s3,c3);
net1 = N1.out2,N2.in1;
net2 = N2.out2,N3.in1;
net3 = N3.out2,N4.in1;
End;
```

There is one disadvantage with the above approach. A user trying to fault simulate a 4-bit adder might get unnecessarily confused with the fault lists associated with the internal nodes C0,C1 ,C2. A higher-level functional model is created and called comptype "4-bit-adder". This model will describe the function of all four 1 -bit adders in a single comptype description. What was earlier called C0,C1 ,C2 are now internal variables to the fault mode! and the user only gets the fault lists F(S0),F(S1),F(S2),F(S3) and F(C3). In this comptype there is a procedure 'Adderl' which does the work of simulating a l-bit adder that is called four times. The three internal carry variables are temp-c0,temp-c1 and temp-c2. The code is listed on the following page.

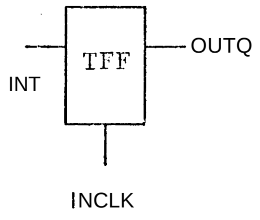
```
Comptype 4-bit-Adder(i1,i2,i3,i4,i5,
  i6,i7,i8,i9,o1,o2,o3,o4,o5:integer;delay:real);
Inward a0,b0,a1,bl,a2,b2,a3,b3,Cin:intnet;
Outward s0,s 1,s2,s3,Cout:intnet;
Var
Temp-cO,Temp-cl ,Temp-c2:integer;
taO,tbO,tal ,tbl ,ta2,tb2,ta3,tb3,tCin:intnet;
ts0,ts1,ts2,ts3,tCout,tc0,tc1,tc2:intnet;

Procedure  Adder1  (ia,ib,ic,od,oe:integer;
               xa,xb,xc,xd,xe:intnet);
(* Code similar to the code as in section 4.1.2 *)
Begin
 xd: = table1(xa,xb,xc);xe: = table2(xa,xb,xc);
 if xa = zero then create-list(ia,s-a-1) else
               create-list(ia,s-a-0);
   if xb = zero then create-list(ib,s-a-l) else
                create-list(ib,s-a-0);
     if xc = zero then create-list(ic,s-a-1) else
                create-list(ic,s-a-0);
     if xd = zero then create-list(od,s-a-1) else
               create-fist(od,s-a-0);
   if xe = zero then create-list(oe,s-a-1) else
               create-list(oe,s-a-0);
 if (xa = zero) and xd<>table1(one,xb,xc) then
                merge-lists(od,ia)
 else if .......
 ....
End; (* Procedure *)

Begin
Waitfor (a0<>u) and (..) and ....(Cin<>u)
      check a0 ,...... Cin;
ta0: = a0;tb0: = b0;ta1: = al ;tbl : = bl ;ta2: = a2;Temp-c0: = 9666;
tb2: = b2;ta3: = a3;tb3: = b3;tCin: = Cin;Temp-cl : = 9667;
Temp-c2: = 9668;
 Adder1 (a0,b0,Cin,s0,Temp-c0,ta0,tb0,tCin,tc0,ts0);
 (* First 1 -bit Adder *)
  Adder1 (al ,b1 ,Temp-cO,sl ,Temp-cl ,ta1 ,tbl ,tcO,tcl ,tsl);
    (* Second 1 -bit Adder *)
    Adder1(a2,b2,Temp-c 1,s2,Temp-c2,ta2,tb2,tc1,tc2,ts2);
  (* Third 1 -bit Adder *)
  Adder1 (a3,b3,Temp-c2,s3,Cout,ta3,tb3,tc2,tCout,ts3);
 (* Fourth 1 -bit Adder *)
Assign ts0 to s0 delay delay1 ;
 Assign tsl to sl delay delay1 ;
  Assign ts2 to s2 delay delay1 ;
 Assign ts3 to s3 delay delay1 ;
Assign tCout to Cout delay delay1 ;
End;
```

4.2.2. T-Flipflop Fault Model

A T type Flipflop is considered and its fault model presented. For simplicity, assume that the Flipflop has a clock input (Inclk),a T input (InT) an output(OutQx) and a state OldQx which needs to be initialised by the user. This state is also the same as the last output OutQx. All inputs and outputs can be either 0 or 1. The clock value is encoded. A 1 implies a correct clock edge switched from a logical 0 **to** 1 while a 0 irnplies otherwise. The table describing the input output relationship and the boolean equation are both given below. Before the TFF can be fault simulated, the state variable has to be initialised by the user. The fault model can be extended to incorporate reset and clear inputs by increasing the code as well and the number of variables in the boolean inptit-output relationship.



INCLK	INT	OLDQX	OUTQ
0	0	0	0
0	0	1	1
0	1	0	0
0	0	1	1
1	0	0	0
1	1	1	1
I	1	0	1
1	1	1	0

TABLE 4-3: TRUTH TABLE FOR A T FF

The TFF functional fault model, described by the code in Fig 4-4, is built analogous to the adder model descibed in an earlier section. A function Eval-Qx accepts Inclk,InT,OldQx as arguments and returns the value of the output OutQx. The variable OldQx in the TFF model is also assigned an **initial** value.

```
Cornptype TFF(Iclk,IT,OQx,OIQx:integer;delay1:real);
 Inward Inclk,InT:intnet;
  Outward  OutQx:intnet;
 Var t1,t2,t3,t4,t5:intnet;
Function  Eval-Qx(g,h,j:intnet):intnet;
  begin .......end;
Begin
     OldQx: = one;
     Waitfor true (Inclk<>u) and (InT<>u) check Inclk,InT;
     t1: = Inclk;t2: = InT;t3: = OldQx;TQx: = Eval-Qx(t1,t2,t3);
     if Inclk = zero then Create-list(Iclk,s-a-1)
                else  Create-list(Iclk,s-a-0);
     .............
     if TQx = zero then Create-list(OQx,s-a-l)
                 else Create- tist(OQx,s-a-0);
     if (OldQx = zero) and (TQx<>Eval-Qx(one,t2,t3)) then
                    merge-lists(OQx,OIQx)
     else if (OldQx = one) and (TQx<>Eval-Qx(zero,t2,t3)) then
                    merge- lists(OQx,OIQx);
     ..............
     Assign TQx to OutQx delay delay1 ;
     Print-out(OQx);
End;
```

Figure 4-4 : Comptype T-Flipflop

If the fault list F(OQx) contains a stuck-at fault at node OIQx, all it means is that some stuck-at faults in the previous input pattern or in the circuit with the previous input pattern has left the flipflop in a wrong state for the present input pattern.

4.2.3. 4-bit Register Fault Model

To illustrate a simple functional fault model of a 4-bit register, consider a 4-bit register with a clock input (the active edge is the low to high edge), a serial input (SIN), a serial output (SOUT) and four outputs QO through Q3. Figure 4-5 below shows a picture of it. The fault model can be easily extended to incorporate shift-rightshift-leftload-parallel inputs by adding more variables to the boolean equation(s) and adding more code to the comptype.

Four state variables S0,S1,S2,S3 contain the initial state of the shift register and require user initialization before fault simulation can begin. The clock Inclk is encoded such that a 1 means a correct transition while a 0 means otherwise. When Inclk = 1 and SIN = 1 then the

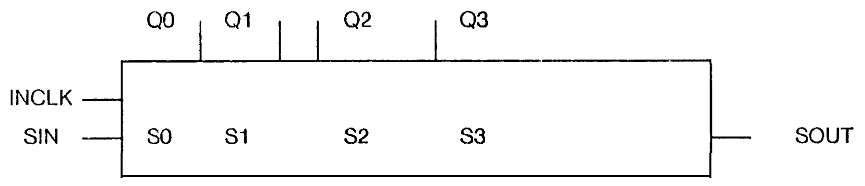


FIG 4-5: A 4 BIT REGISTER

new value of QO and SO become 1. The new value of Q1 and S1 become same as the old value of SO while that of Q2 and S2 become equal to the old value of S1 and so on. To build a fault model for the 4-bit register, it is decomposed into four conceptual blocks (each a l-bit register) first. In figure 4-6 below, yl ,y2,y3 are the temporary internal links from one l-bit register to another. What is intended to be done is something similar to what was done in the case of the adder in section 4.2.1. A procedure within the comptype which can build the fault lists for a single l-bit register is called four times. For a 8-bit or 16-bit register, the procedure will have to be called 8 or 16 times.

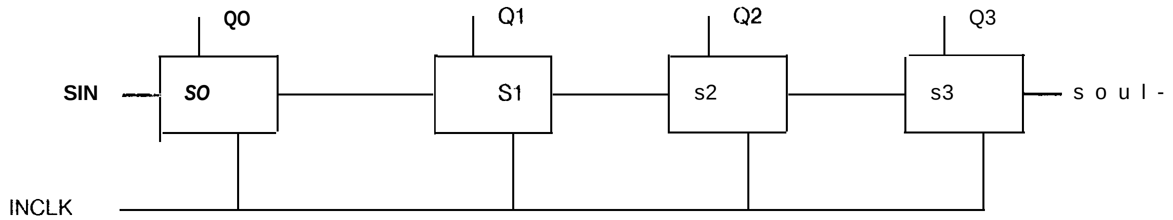
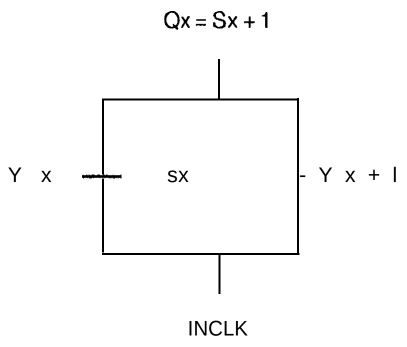


FIG 4-6 : DECOMPOSED 4 BIT REGISTER

The truth table for a single 1 -bit register and a single 1 -bit register are shown in table 4-4.

The final fault lists F(Q0) through F(Q3) may contain stuck-at faults at nodes SO through S3. What all it means is that some stuck-at faults in the previous input pattern or in the circuit with the previous input pattern has left the register in an erroneous state for the present input pattern. The ADLIB code is given in Fig 4-7.



INCLK	Yx	Sx	Qx
0	0	0	0
0	0	1	
0		0	0
0			
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

TABLE 4-4 : TRUTH TABEL FOR THE 1 BIT REGISTER

```
Comptype 4-bit-register(Iclk,SIN,S0,...S3,OQ0,..OQ3:integer);
 Inward SIN,Inclk:intnet;
 Outward Q0,..Q3:intnet;
Var TQ0,..TQ3,TSIN,TCLK:intnet-val;
Procedure 1 -bit-register(.....);
  begin
   ..This fault simulates a ‘1 -bit register..
  end;
Begin
  TSO: =1; . . . ..TS3: = 1; (* Assign initial values *)
   Waitfor (Inclk<>u) and (TSIN<>u) check TSIN,Inclk;
    if Inclk = 0 then Create-list(lclk,s-a-l)
                  else  Create-list(lclk,s-a-0);
     1-bit-register(.....  )(* fault simulates first 1 -bit register *)
       1 -bit-register(.....)( * fault simulates second 1 -bit register *)
     1-bit-register(.....)(* fault simulates third 1 -bit register *)
    1 -bit-register(.....)(* fault simulates final 1 -bit register *)
   Assign TQO to QO delay 0.0; . . . ..Assign TQ3 to Q3 delay 0.0;
End;
```

Fig 4-7 : Comptype 4-bit Register

4.2.4. Asynchronous Counter Fault Model

In this subsection, a simple implementation of an asynchronous counter from T-flipflops is considered and the fault model deduced. Four Tff's are interconnected in a manner shown in the figure 4-8 below. The clock input of the leftmost one is connected to an external clock and assume that each Tff is active on the positive edge of the clock.

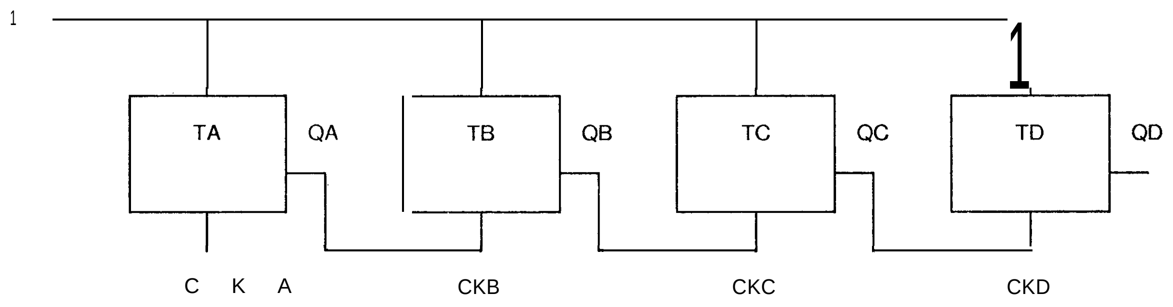


FIG 4-8 : DECOMPOSED ASYN COUNTER

Each Tff has an input clock, a T-input that is tied permanently to logical 1, an output Q and two state variables. One state variable OldQa stores the old value of Qa (or the state) and another state variable OldCKa remembers the last logical value of the clock input. For each of these l-bit counters, there is a model (comptype l-bit counter). The input output truth table and the boolean equation for a single l-bit counter is given below. In this model,the states OldQa,.....OldQd,OldCka,....OldCkd will all have to be initialized before simulation can begin.

On a value assigned from an external clock to Cka, the A l-bit counter is fault simulated first. The fault list F(Qa) involves the nodes Ta,Cka,OldQa and OldCka. Once F(Qa) is obtained, the next B l-bit counter is fault simulated and so on. This modular fault model of the asynchronous counter is easily extensible to more than 4-bit asynchronous counter.

TA	)LDQA	OLDCKA	CKA	QA
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

TA	OLDQA	OLDCKA	CKA	QA
	0	0	0	0
	0	0	1	1
	0	1	0	0
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	1
	1	1	1	1

TABLE 4-5: TRUTH TABLE FOR ASYN COUNTER

4.25 Synchronous Counter Fault Model

Consider a 4-bit synchronous counter with four outputs Qa,Qb,Qc,Qd,one clock input Inclk (encoded in the same manner as the Tff in subsection 3.2.2), a ripple carry output(to connect synchronous counters) and four state variables OldQa through OldQd. A set of 5 boolean equations relating the inputs to the outputs is used to construct the fault model in a manner similar to the ones described earlier.

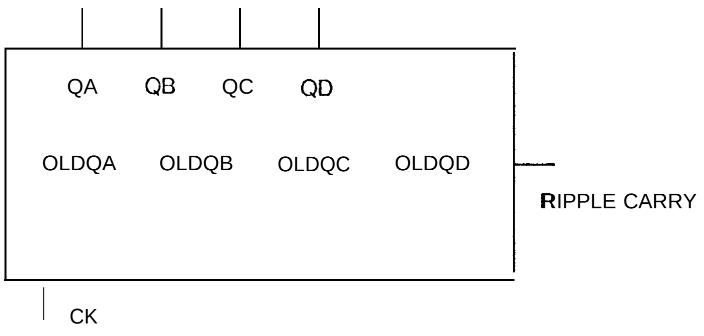


FIG4-9 : SYNCHRONOUS COUNTER

4.2.6. Memory Unit Fault Model

The problem of functionally fault simulating a memory is severe since there is usually no boolean relationship between the address(the input) and the contents (output). This is explained with the help of an example. I-he example considered is a 8-bit address memory with 8-bits of output. If the memory unit is a ROM, then its contents are supplied to the simulation program by the user. If the memory unit is a RAM then, its contents can change in the course of a simulation.

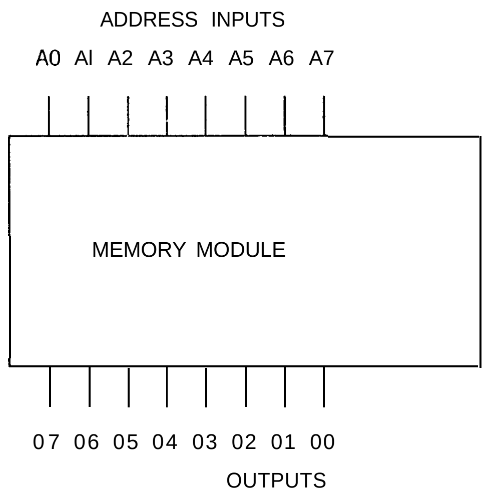


FIG 4-10 : MEMORY UNIT

Let the address input be 11111100. Let the address lines be called A7 through AO. To do deductive simulation the content of locations 11111100,11111101,11111110,11111000,11110100,11101100,11011100,10111100 and 01111100 have to be known and only these have to be stored in the program. Each of these,except the first, correspond to a faulty address based on a single bit stuck-at fault at each of the 8 bit positions of the address. Let the outputs be 00 through O7.Only those faulty inputs which lead to a content different from that of the location 11111100 will contribute to the fault list. For example if only content of location 10111100 is different from that of the correct one in the first position then the fault list will be *F(O7) = O7(s-a-0/1)VA6(s-a-0)*. For a memory of address length 16 or 32, the fault simulation requires the knowledge of the contents of 16 or 32 specific locations, when only one address vector is used in fault simulation. When the memory is the part of a large circuit and each of the address pins A0 through A7 have fault lists F[AO] through F[A7] associated with them,then a check must be made whether there are any faults common between F[A0],F[A1],..F[A7]. If there are, that means two or more pins(AO through A7) may be simultaneously faulty due to a single stuck-at fault at an earlier node (this may happen when there is a reconvergence in the circuit preceeding the memory unit). The contents of the memory locations, addressed by the specified faulty bits will be necessary. In the course of fault simulation, the memory mcdule may be fault simulated more than once with different address vectors. For this reason, it might be necessary to store the contents of all the memory locations in a table.

In [16],Akers describes how to use binary decision trees to handle complex functional boolean relationships. Our strategy,namely looking at the contents of memory locations,each address being off from the non-faulty one by a single stuck-at fault,is very economic and is similar to Akers' idea.

4.2.7. ALU Fault Model

The ALU used as an example is the TI-SN74381 (Fig 4-l 1), a 4-bit wide circuit. It has three select lines for choosing the function,4-bit input 'A',4-bit input 'B' and a single bit Cin. The function select table is given in Table 4-6. To do deductive fault simulation it is necessary to know the truth table for 12 inputs and 5 outputs. To avoid this difficulty the 4-bit ALU is functionally decomposed into four 1 -bit ALU's. The lowest significant 1 -bit is simulated first and the relevant fault list is generated before the next significant bit is simulated. The functions add(A + B), subtract(A-B), subtract(B-A) propagate carry's and borrow's from the lowest significant bit to the most significant bit. This naturally supports propagating the fault lists from the lowest significant 1 -bit ALU to the most significant 1 -bit MU. For the truth table for a single 1-bit ALU, please refer to the T.I. catalog. It can be used by all of the four 1-bit ALU's. The outputs are F0 and Cnext(useful only for add and subtract). Cnext from one significant bit becomes Cin for the next significant bit. The fault model would be quite analogous to the several models deduced in the earlier sections.

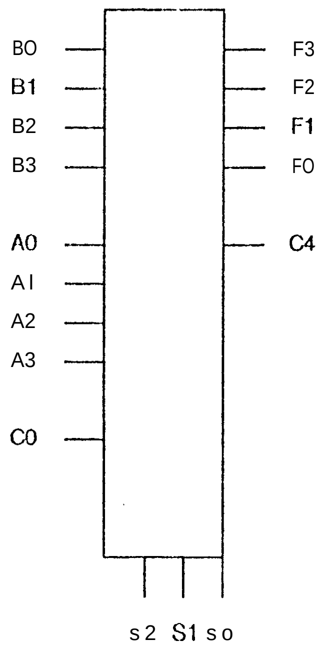


FIG 4-11: ALU

s2	S1	S0	FUNCTION
0	0	0	CLEAR
0	0	1	B-A
0	1	0	A-B
0	1	1	A+B
1	0	0	A EXOR B
1	0	1	A OR B
1	1	0	A AND B
1	1	1	PRESET

TABLE4-6 : FUNCTION SELECT TABLE

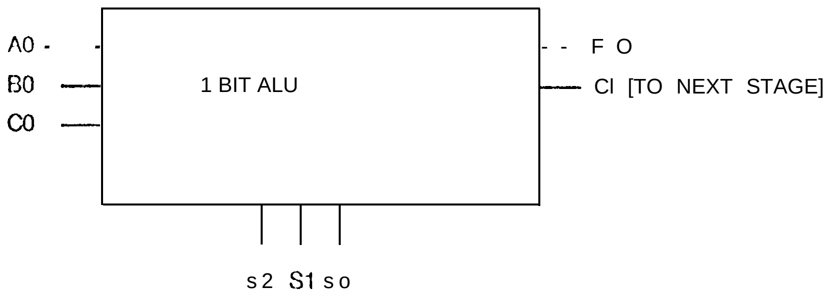


FIG 4-1lt3: ONE BIT ALU

4.2.8. Multiplier Fault Model

Simulation of a 16 by 16 unsigned binary multiplier chip can be achieved using a technique of functional decomposition into manageable modules. Conventionally one requires a table with 32 inputs(16 from multiplier and 16 from multiplicand) and 32 outputs(the result) or an equivalent set of complex boolean equations. Instead a method is adopted which is explained with a simple 3 bit X 3 bit multiplication example.

```
             a2    al    a0
        X     b2    bl    b0
         ---------------------------
           a2.b0  a1.b0  a O . b O   --(1)
   +     b1.a2  b l . a l  b1.a0        --(2)
----------------------------------------
   +     bl .a2 (a2.b0 + al .bl) (al .bO + bl .aO) a0.b0 --(3)
b2.a2   b2.a1  b2.a0                    --(4)
--------------------------------------------
b2.a2 (bl .a2 + al .b2) (b2.a0 + a2.b0 + al .b1) (al .b0 + bl .aO) a0.b0--(5)
```

In the example above is shown a simple 3bit X 3bit multiplication with all the intermediate partial products. Along with the value of partial product 'aO.bO' , is also obtained the fault list involving the stuck-at faults at nodes a0 and b0. This is done for all the partial products on lines 1 and 2. Partial sums like 'al .bO + a0.b1' are obtained in line 3 and once again, the fault list involving stuck-at faults for al ,b0,a0,b1 can be obtained. In this manner eventually the fault lists for SO through S5 involving stuck-at faults at nodes a0 through b2 are deduced. This method, shown for a 3bit X 3bit multiplication, can be extended to a 16X1 6 multiplier.

4.2.9. Fault Models for other Functional Blocks

Multiplexers,combinational PLA's ,even other functional sequential blocks that can be represented by a set of boolean equations (or a partial truth table) can be fault simulated. In the worst case where the complex functional block cannot be reasonably modularised, one can still work with those complex boolean equations.

4.3. An Invariance Principle

A very interesting observation has been made in the functional simulation technique. All along when building fault models for adders,counters,registers either the implementation details have been ignored or a particular implementation was assumed. It is quite legitimate to imagine that there could be more than one implementation or that there could be a number of different boolean representations for the same functional block. Deductive functional fault simulation will always give the same result no matter what implementation or what boolean equations are assumed while building the fault model. Intuitively, the fault lists obtained, involve stuck-at faults at only the input and output nodes and hence should be indifferent to the internal implementation. A formal proof of the invariance principle, as applied to combinational and sequential circuits, is given in the two following subsections 4.3.1 and 4.3.2.

4.3.1. Corn binational Functional Blocks

So far, combinational functional models have been built without having worried about the implementation details. For example, while building the functional fault model for the 4-bit Adder in section 4.2.1, a particular implernentation strategy was assumed and the model deduced. However, the 4-bit Adder can be implemented in a variety of ways. It appears intuitively obvious that the output fault lists, given input fault lists , will be indifferent to the internal implementation details since the combinational functional model concerns only with the stuck-at faults at the input and output nodes. In this section, a formal proof is presented to that effect.

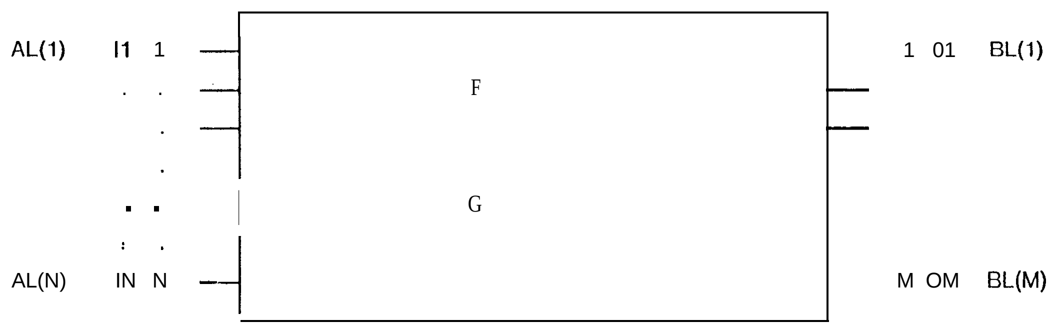


FIG 4-12A : ONE REPRESENTATION OF THE FUNCTIONAL BLOCK

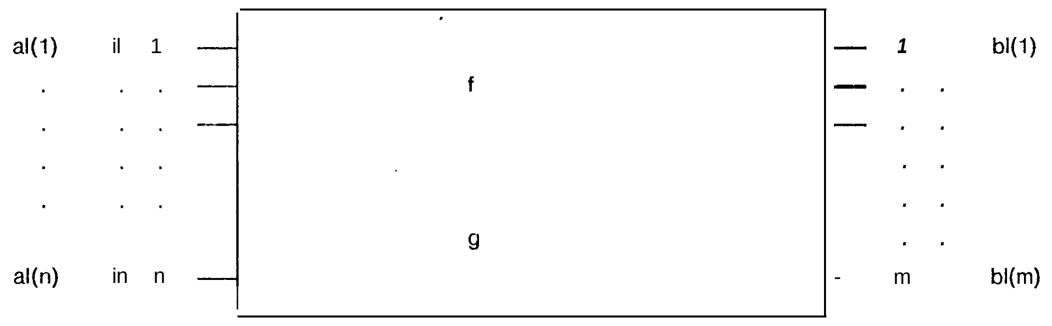


FIG 4-12B : ANOTHER FUNCTIONAL REPRESENTATION

In Figures 4-12a and 4-12b, F, f are the boolean mappings while G, g are the fault list mappings for the two representations of the same combinational functional block. Input boolean vectors I,i, output boolean vectors O,o, input fault list vectors AL,al and output fault list vectors BL,bl are shown expanded below. I = (I1,I2,..IN), i=(i1,i2,.in), O = (O1,O2 ,.... OM), 0 = (o1,o2,.... om), AL = (AL(1),...... AL(N)), al = (al(l) ,..... al(n)), BL = (BL(1),....BL(M)), bl = (bl(1),.....bl(m)).

Each of the two figures, shown above, represent the same functional block. The figure

4-12A representation is characterised by the input output mapping F while the figure 4-12B representation is characterized by the mapping f. Incidentally, the proof is restricted to only {0,1} logic states and single stuck-at faults. The assumption is that mappings F, f are equivalent. Also the same input boolean vectors and input fault lists are applied to both representations.

Stated mathematically,

$$I = i$$

$$0 = F(I) = f(i) = o$$

$$AL = al$$

$$BL = bl \qquad \text{...eqn(1)}$$

The fault list for any node is a set (collection) of stuck-at faults at some nodes, each of which can force a complement of the correct value at that particular node. The only operation on a fault list is addition i.e. a new stuck-at fault is added to the set of faults in the fault list. This addition is commutative, hence the order in which the stuck-at faults are added to a fault list is immaterial. Two fault lists corresponding to the same node are identical if their contents are identical. The idea of fault simulation is to apply the input fault lists to a functional block and obtain output fault lists. Mathematically expressed, there is a transformation between the input and output fault lists.

In fig 1 : BL = G(AL) where AL,BL are input and output fault list vectors.

In fig 2 : bl = g(al) where al,bl are input and output fault list vectors.

The aim is to prove that the transformations G and g are equivalent. Already F and f are equivalent. If there is a way of mapping G to g and F to f then there is a chance of proving the point. A three step strategy is adopted. The first step will try to map input fault lists into {0,1} input boolean vectors. In the second step, these input vectors are operated by mappings F and f as appropriate and mapped into output boolean vectors. The third and final step will try to map the output boolean vectors into output fault lists. In each of the three steps, if a one to one correspondence between the two cases can be found and also the output fault lists are shown identical the job is done. The theorem is formally stated.

Theorem : If two representations of the same combinational functional block (both representations having the same number of inputs and outputs) are such that their input output boolean mappings are equivalent, then their input output fault list mappings will also be equivalent given the same input vectors and same fault list vectors.

PROOF :

Step 1: The fault list needs to be understood first. For example, consider the fault list AL(1) for node 1 having a correct boolean value Il. This list is a set of stuck-at faults corresponding to some nodes, each of which can force node Il to have a boolean value of -I1. When considering the fault list AL(1) all other input nodes are supposed to have correct boolean values. The presence of fault list AL(1) is therefore equivalent to an input boolean vector AIX1 which differs from the correct input vector I in the first element position. That is AIX1 = (-Il ,I2 ,... IN). In a similar manner, fault lists AL(2) ,.... AL(N) will be mapped to input vectors AIX2,AIX3,....AIXN all of which differ from the correct input vector I. For figure 4-12B, the input fault lists al(l),..... al(n) will be mapped to input vectors aix1,.....aixn, all of which differ from the correct input vector i.

I = (I1,I2,....... IN) i = (i1,i2,...... in)

AIX1 = (-I1,I2,....... IN) aixl = (-i1,i2 ,...... in)

AIX2 = (Il, -12 ,....... IN) aix2 = (i1, -i2 ,...... in)

....

AIXn = (I1,I2 ,....... -IN) aixn = (il ,i2 ,...... -in) . . . . eqn(2)

Using eqn(1), AIX1 = aixl , AIX2 = aix2 ,....... AIXN = aixn.

The mapping in step 'I can be expressed as :

AL(1) = >AIX1, AL(2) = >AIX2, ...AL(N)=>AIXN for figure 4-l 2A
and
al(l) = >aix1, al(2) = >aix2, ... al(n) = >aixn for figure 4-12B . . . . eqn(3)

Step2: All the input vectors, in both the figures, are now operated on by the respective boolean equations. The output vectors corresponding to the input vectors AIX1 through AIXN are evaluated using the mapping F while those of aixl through aixn are evaluated using the mapping f. The corresponding output vectors are AOX1 through AOXN and aoxl through aoxn respectively. Thus :

For fig 4-12A:
BOX1 = F(AIX1)

BOX2 = F(AIX2)
....

BOXN = F(AIXN) ..........*............*.....eqn(4a)
For fig 4-12B:

box1 = f(aix1)

box2 = f(aix2)
....

boxn = f(aixn) ....eqn(4b)

Using eqns (l), (2), (4a), (4b) :

BOX1 = box1, BOX2 = box2, BOX3 = box3,......., BOXN = boxn . . . . eqn(5)

The mapping in step 2 can be expressed as :

AIX1 = >BOX1, AIX2 = >BOX2,...... AIXN = >BOXN
and
aixl = >box1, aix2 = >box2,......, aixn = >boxn ....eqn(6)

Step 3: This step attempts to map output boolean vectors into output fault lists. In steps 1 and 2 the corresponding results for figures 4-12A and 4-12B bear a one to one correspondence. If it can be proved that the output fault lists in the two figures are identical the equivalence of the mappings G and g are will be proved.

The output vectors BOX1 through BOXN are either all identical to the correct output vector 0 or at least one of them are different. If output vectors BOX1 through BOXN are all identical to the vector 0 then the output vectors box1 through boxn will all be identical to the vector o. Under these circumstances, none of the faults in the input fault lists will be detected at the output and the transformations G and g will both be null. Consider a very general case where all output vectors BOX1 through BOXN differ from 0 in all element positions. 'This incidentally implies vectors BOX1 through BOXN are all identical to one another. Also the output vectors box1 through boxn will differ from correct output vector o in all element positions and will all be identical. The equations are given below.

BOX1 = BOX2 = ... = BOXN =(-O1,-O2,-O3,.....,-ON). . . . . . .eqn(7a)

box1 = box2 = ... = boxn = (-01,-o2,-o3,.....,-on) . . . . .eqn(7b)

At this point recall that an output fault list say AL(l) for node 1 and having correct value 01, is defined as a collection of stuck-at faults which when true will force a value -01 on the node 1. From eqn(7a), node 1 will be forced a value -01 by output vectors AOX1 through AOXN. By definition therefore, fault list for node 1, AL(1) must contain all the faults that cause output vectors BOX1 through BOXN. From eqns (3) and (6) one gets : AL(1) = >BOXl , AL(2) = >BOX2 ,......,AL(N) = >BOXN. This means that input fault list AL(1) causes output vector BOX1 and so on. The output fault list BL(l) is given by :

BL(l) = {AL(1),AL(2) ,.... AL(N)}.......................eqn(8a)

Similarly the output fauit list bl(1) for fig 2 is given by:

bl(1) = {al(1),al(2) ,..... al(n)}......................eqn(8b)

Using eqns (l), (8a) and (8b) one gets BL(l) = bl(1).

In a similar manner it can be shown BL(2) = bl(2),BL(3) = bl(3).......BL(N) = bl(n).

Hence, BL = (BL(1),...BL(N)) = bl = (bl(1),bl(2),...bl(n)) = G(AL) = g(al).

This proves that transformations G and g are equivalent which in turn implies that the mapping of input fault lists to output fauit lists is independent of the implementation details of the functional block. This obviously assumes that the logic inside the functional block is fault free.

4.3.2. Sequential Functional Blocks

In the last subsection, it was proved that the rnapping from input fault lists to output fault lists was invariant to the implementation details of the combinational functional block. This section will attempt to prove the same principle, this time with respect to sequential circuits. A sequential circuit is characterized as a block with primary inputs I1,..IN, external outputs 01 ,..ON and internal state variables **SO1** ,..SOK. These state variables retain some information from the past, affect the present outputs and are re-evaluated. The re-evaluated values of the state variables SN1 ,..SNK are fed back to be used in evaluating the outputs next time around.

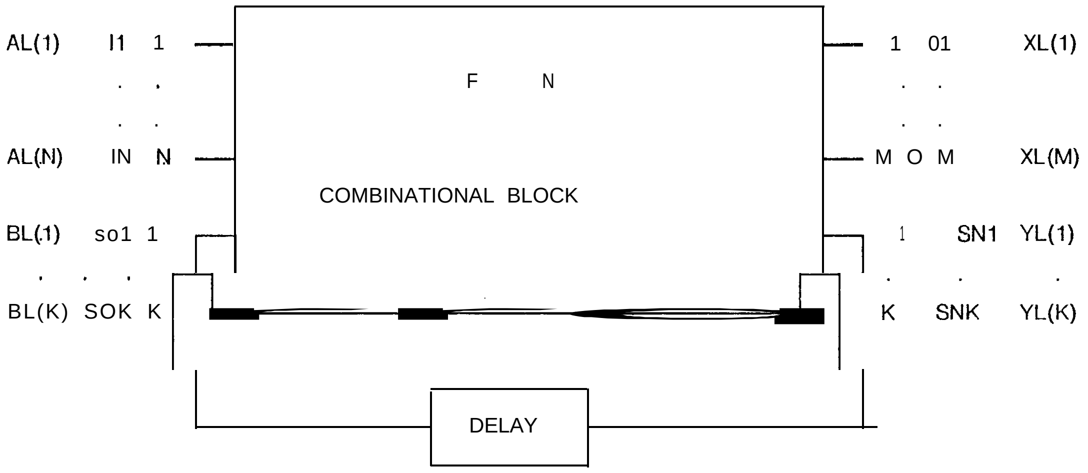


FIG 4-13: SEQUENTIAL CIRCUIT MODEL

Figure (4-13) shows the sequential circuit. Associated with the functional block are two mappings F and N which are explained in the equations below.

o = F(I,SO) where SO = (SO1,SO2,. . . .SOK).

SN = F(I,SO) where SN = (SN1,SN2 ,... SNK).

Also, let the input state variables SO1 ,...SOK be associated with hypothetical nodes 1,...K and similarly the output state variables with the hypothetical nodes 1,....K. Henceforth these nodes will be referred to as state nodes. The sequential circuit can thus be seen as a combinational circuit at each time instant. At time instant t, the input vectors are I, SO. The output vectors, evaluated using F,N mappings, are 0 and SN. The output state vector SN becomes the input state vector SO in the next time around and in the next time instant it once again behaves as a combinational circuit.

If AL(1),....AL(N) represents the input fault lists for the input nodes 1,...N and BL(1),....BL(K) represents the input fault lists for the input state nodes 1 ,...K, then the output fault lists XL(1),...XL(M) and YL(l),....YL(K) can be obtained using the transformations G and M as described in the equations below. Once again, the output state node fault lists YL(l) through YL(K) become the input state node fault lists XL(1) through XL(K) in the next time instant.

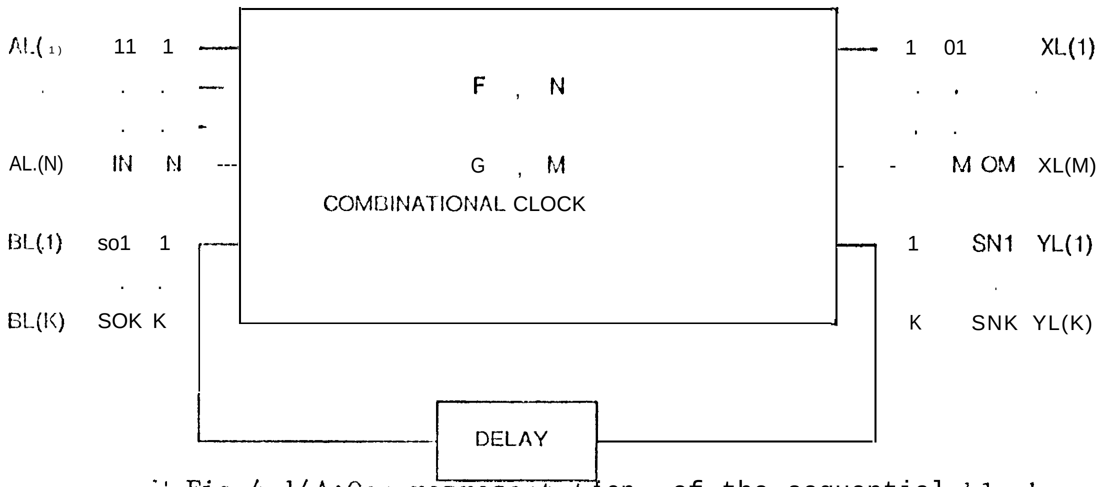
$$XL = G(AL,BL)$$

$$YL = M(AL,BL)$$

With this background, the theorem is formally stated.

Theorem : If a sequential circuit has two functional representations (both having same number of input nodes, output nodes and state variables) which have equivalent mappings on input boolean vectors given the same primary input boolean vectors and input state vectors, then their mappings on input fault list vectors will be equivalent provided that the fault lists at the primary input nodes and input state nodes of one representation is same as those of the corresponding nodes in the other representation.Figures 4-14a and 4-14b are two representations of the same sequential circuit.* In figure 4-14A, the input boolean vector is I, input state vector is SO, output boolean vector is 0 and output state vector SN. The primary inputs fault list vector is AL while that for the state nodes is BL. The output fault list vector for the external outputs is XL while that for the state nodes is YL. The mapping functions F,N,G,M are explained in the equation below.

[2] Both these figures depict snapshots of the sequential circuit at a certain time instant 't'.



'' Fig 4-14A:One representation of the sequential block

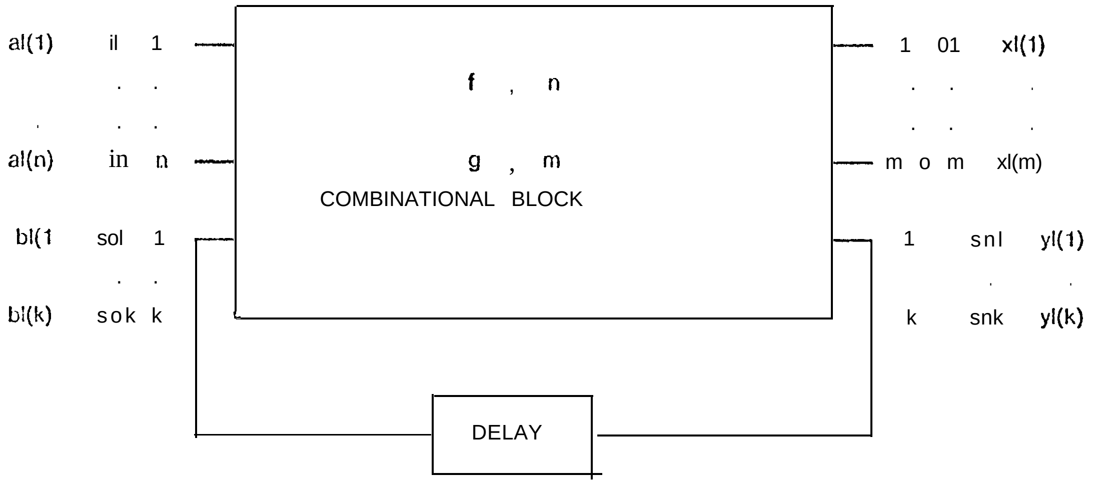


FIG 4-14B : ANOTHER REPRESENTATION OF THE SEQUENTIAL BLOCK

o = F(I,SO)

SN = N(I,SO)

XL = G(AL,BL)

YL = M(AL,BL)

In figure 4-14B, the input boolean vector is i, input state vector is so, output boolean vector is o and output state vector sn. The primary inputs fault list vector is al whiie that for the state nodes is bl. The output fault list vector for the external outputs is xl while that for the state nodes is yl. The mapping functions f,n,g,m are explained in the equation below.

o = f(i,so)

sn = n(i,so)

xl = g(al,bl)

yl = m(al,bl)

It must be emphasised that apart from having the same number of inputs and outputs, the number of state variables in the two figures is the same. This last condition is necessary since without it, equations (2),(5) and (6) cannot be used in the proof below.

PROOF :

Given the following:

I = i ....eqn(1)
SN = SO ....eqn(2)
AL = al ....eqn(3)
BL = bl ....*.....eqn(4)

Also given that F = f and N = n. ....eqns(5)&(6)
Hence:

0 = F(I,SO) = f(i,so) = o

and

SN = N(I,SO) = n(i,so) = so

In figures 4-14A,4-14B the mappings F and f are equivalent. Both these mappings map input boolean vectors (vectors I and SO in figure 4-14A; vectors i and so in figure 4-14B) to output boolean vectors (vector 0 in figure 4-14A; vector o in figure 4-14B). Using the theorem proved in the last subsection, it can be therefore said that

XL = G(AL,BL) = g(al,bl) = xl

Similarly, mappings N and n in the two figures are given equivalent. The mapping N relates the output boolean vector SN to the input boolean vectors SO and I while n relates the output boolean vector sn to the input boolean vectors so and i. Using the theorem proved in the last subsection, it is therefore deduced that

YL = M(AL,BL) = m(al,bl) = yl

Thus the output fault lists {XL(I),...XL(M),YL(1),...YL(K)} and {xl(1),....xl(m),yl(1),....yl(k)} for figures 4-14A and 4-14B are identical when the two representations of the functional block are fault simulated at time instant t. The same result will hold true for the next time instant and for all other time instants after that. At every time instant however, the output fault list vectors YL and yl from the previous time instant become equal to the input fault list vectors BL and bl for the present time instant.

4.4. Summary

Functional fault model building has been demonstrated with a few of the standard functional blocks. Other functional blocks which can be represented by an input output boolean equation(s) can be analogously fault simulated. Chapter 7 will quote some numbers from actually fault simulating several functional models in ADLIB-SABLE. A limitation that the above technique has is as follows : One can imagine very complex functional blocks like microprocessors, where the entire input output relationship is not defined. That is, some input patterns are not impiemented. If a stuck-at fault in such a model, forces the input pattern to an undefined input combination, the model fails. An extension to Deductive fault simulation will be discussed in chapter 5 which will address this problem.

Chapter 5
Mixed-Level And Multivalued Fault Simulation

5.1. The Principle of Mixed-Level F.S.

Chapters 2 and 3 discussed fault simulation at the gate level while chapter 4 discussed functional level fault models. The former is more accurate but takes more time white the latter is faster but less informative. One can imagine a scenario where one might need to run fault simulation on a big circuit at a functional level (due to time constraints) but simultaneously require accurate fault lists (fault simulation at gate level) for a particular component say an adder. ADLIB-SABLE can support this dual purpose. All the user needs to do is to have gate level fault models for the adder on top of the functional fauit models, interconnect the gates so that the functional model as well as the gate models get the same input pattern(s). The simulation can now be run and the result will be as desired. The next section illustrates this with an example.

5.1 .1. An Example of Mixed-Level F.S.

Figure 5-l shows the different components and their interconnections. The nets which carry the input values to the functional adder also carry the inputs to the appropriate gates of the box representing the gate level implementation of the adder. The outputs S,Cout of the box go nowhere. After simulation, one gets the fault lists F(S),F(Cout) in terms of the nodes a, b, Cin as well as the internal nodes of the gates that actually build the adder in addition to the fault lists at the functional level. A brief sketch of the ADLIB and SDL codes are given below :

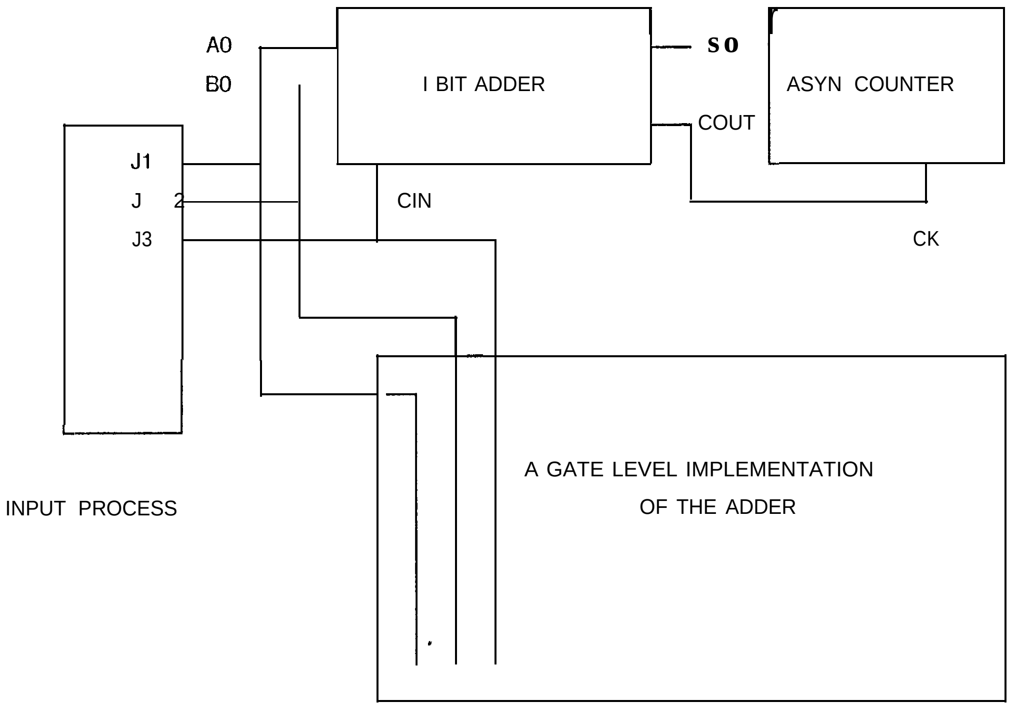


FIG 5-1: MIXED-LEVEL FAULT SIMULATON

```
ADLIB CODE:
{ data structures and procedures}
Comptype Adder(. . .);
End;
Comptype Asyn-Counter(...);
End;
Comptype AND2(...);
End;
.....
Comptype INPUT-PROCESS;
End;


SDL CODE :
Types: Adder,Asyn-Counter,AND2,
. . . ..INPUT-PROCESS.
Adder: Adder1(....... );
AND2: G1(..... ),G2( ........);
..........
INPUT-2ROCESS:lnp;
end;
Netsegment;
net1 = lnp.j1,Adder.a0,G6.in1,G9.in1,G11 .inl;
net2 = lnp.j2,Adder.b0,G1.in2,G5.in1,G9.in2;
......
End;

Begin (* Main *)
End. (* Main *)
```

5.2. Multivalued Fault Simulation

5.2.1. Issues in Model Accuracy

So far, only 0,1 simulation and stuck-at 0 and stuck-at 1 faults have been considered in the deductive technique. It is possible,in real circuits,that some nodes may not quite be logical 0 or 1 but some value in-between. This,commonly denoted by 'u',may arise due to bridging faults or non initialization and may be quite important in some technologies. For TTL and ECL 0,1 simulation may be quite adequate but for MOS and CMOS it may be required to provide one or more states without which the simulation may be incomplete or inaccurate. This chapter will be restricted to three-valued fault simulation. That is, 0,1 ,U will be considered as.

possible logic states and thus have three stuck-at faults s-x-0,s-x-1 ,s-x-U for a node 'x'. Extending to four or more valued simulation is also possible.

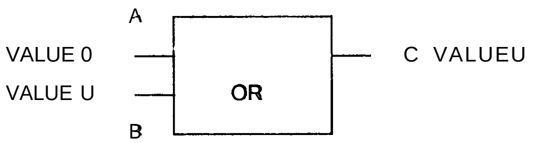


FIG 6-I : AN OR GATE

Fig 5-2: AN OR GATE

Imagine a scenario where the inputs to an OR gate (Figure above) are 0 and U. The output is U. If 0,1 simulation method was used and only s-a-l ,s-s-0 fault types considered then no fault list will be generated for the output node of the OR gate. Another approach, called "least pessimistic approach" by Breuer[4] is adopted, in which one can have a s-a-U fault along with the s-a-0 and s-a-l faults. Instead of having a single fault list F(X) for node X, one now has three fault lists FO(X),Fl(X) and FU(X) for every node X. FO(X) will contain all the stuck-at faults which will force node X to have faulty value 0. F1(X) list contains all the stuck-at faults that will force node X to a faulty value 1 and finally FU(X) will contain all those that will force node X to a faulty value U. The example OR gate in the figure above is considered deductive fault simulation is performed. Node A is 0. Hence Fl (A) = A(s-a-l); FO(A) = nil; FU(A) = A(s-a-U). Node B is U. Hence FU(B) = nil; FO(B) = B(s-a-0); F1(B) = B(s-a-l). The output node C will be U. Hence FO(C) = C(s-a-O)VFO(B); F1 (C) = C(s-a-l)VFl (A)VF1 (B); FU(C) = nil. Contrary to the single fault list F(C) which would have been got in the earlier method, one now has three fault lists F1(C),F0(C) and FU(C). These three lists contain more information than not having any fault list for C (which would have been the case in 0,1 simulation) and can be quite useful in some technologies.

5.2.2. [0,1] Concurrent and [0,1] Deductive

In concurrent simulation[15], given a circuit and an input set,one proceeds as follows. For every stuck-at fault occuring at a given node, create a copy of the entire circuit with the gate in question replaced by a faulty one. All such copies are evaluated simultaneously. While evaluating, if in any of the copied circuits a node value becomes identical to the corresponding node in the original circuit, then the fault giving rise to the copy in question will go undetected. This copy is therefore dropped. Creating copies is termed as splitting while dropping copies from consideration is termed merging in concurrent fault simulation. For terminology, please refer to [1 5].

This section illustrates concurrent fault sirnulation with an example and then shows how it can be implemented in ADLIB-SABLE. Next deductive fault simulation will be carried out on the same example with the same input pattern and the two simulation techniques will be compared. For simplicity of explaining two-valued simulation is used although 3 or more valued simulation could be used analogously.

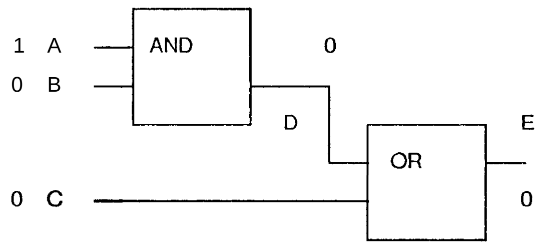


Fig 5-3: AN EXAMPLE

In the above figure, the primary inputs a,b,c get values 1,0 and 0 respectively. The correct value of nodes d and e will be both 0.

deductive Technique :

Step 1: Fault simulate gate G1 since both inputs are assigned.

F(a) = a(s-a-0)

F(b) = b(s-a-l)

F(d) = d(s-a-1) V b(s-a-l)

Step 2 : Now fault simulate gate G2.

F(c) = c(s-a-1)

F(e) = e(s-a-1) V F(d) V F(c)

= e(s-a-l) V d(s-a-l) V b(s-a-l) V c(s-a-l)

The fault lists are now complete.

5.2.2.1. Pure Concurrent Fault Simulation

For every stuck-at fault a separate copy is created. The possible stuck-at faults cases are a(s-a-O),b(s-a-l),c(s-a-l),d(s-a-l) and e(s-a-l). Five copies of the circuit are to be created. Since concurrent does not impose any restriction regarding the order in which the copies are created, the following is done.

Step 1: Consider ail the copies concerned with faults a(s-a-O),b(s-a-l), d(s-a-l). The reader will notice that the nodes a,b and d are all concerned with gate G 1. These copies are drawn below.

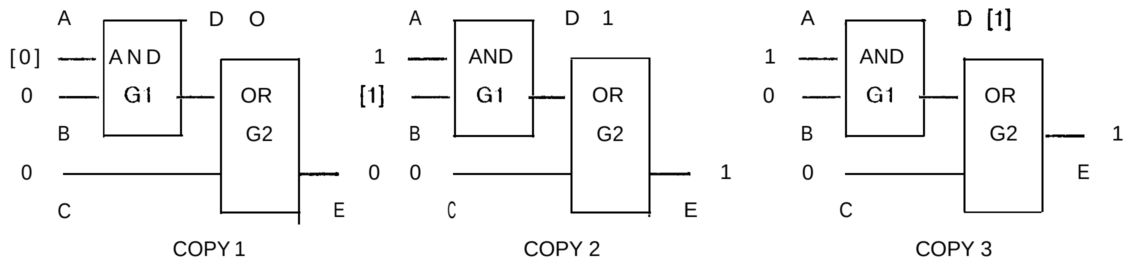


Fig 5-4: CONCURRENT FAULT SIMULATON

Each of the copies has one fault each and are evaluated. For copy 1 notice the output d is 0. This is the same as in the original circuit and hence the fault a(s-a-0) will not be detected at e. This copy is dropped. The copies 2 and 3 indicate that a faulty value at nodes d and e and hence faults d(s-a-l) and b(s-a-l) will be detected at node e.

Step 2 : Now consider the remaining two copies.

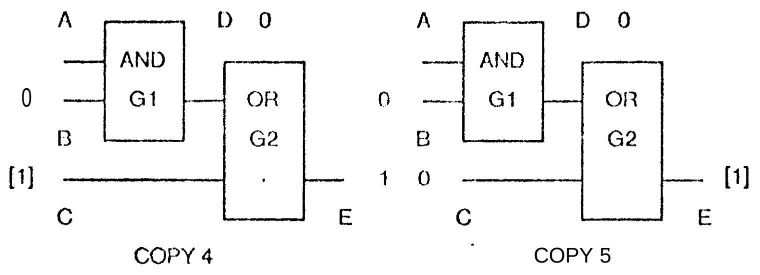


Fig 5-5: CONCURNENT FAULT SIMULATION

For copy 4 the output node e has a value 1 and hence the fault c(s-a-l) will be detected. Similarly the fault e(s-a-1) will be detected. The final fault list at node e will be F(e) = e(s-a-l)Vc(s-a-l)Vb(s-a-l)Vd(s-a-l).

5.2.2.2. Concurrent fault Simulation in ADLIB-SABLE

In ADLIB-SABLE, instead of concurrently evaluating ail the five copies, the following scheme is adopted. Somewhat like what was done in the deductive case, here, first consider the gate(s) which has all its inputs assigned. Then consider the stuck-at faults at the nodes of this gate and create one copy for every fault. Each copy has only one faulty node. Evaluate each copy and drop those which do not show an output different from that in the original circuit. For those copies that do show a faulty value at the output node, store the corresponding stuck-at faults in a list identified by the output node. To illustrate what happens next consider the the example circuit.

Step 1 : According to the scheme, Cl would bc considered first. Three of its nodes can be stuck-at faulty values namely a(s-a-0),b(s-a-1) and d(s-a-1). Create three copies and evaluate them along with the original.

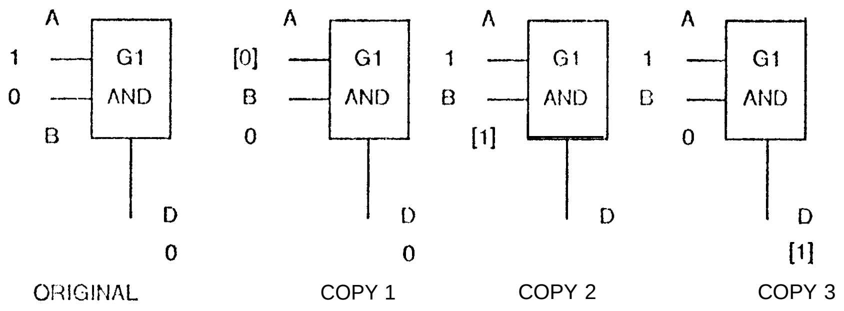


FIG 5-6: CONCURRENT FAULT SIMULATION IN ADLIB-SABLE

Copies 2 and 3 force a 1 on the node d. Hence b(s-a-l), d(s-a-l) are stored in a list F(d). What this list means is that if a faulty value of d = 1 can be detected at the next output then the faults b(s-a-1) and d(s-a-1) will be detected there too. Copy 1 is ignored since it does not force the output node faulty. The value of d is now assigned as G1 simulation is complete.

Step 2 : Now that node d is assigned, gate G2 will be considered. As before create three copies corresponding to d(s-a-1),c(s-a-1) and e(s-a-1). All three are evaluated along with the original. The figure below shows them.

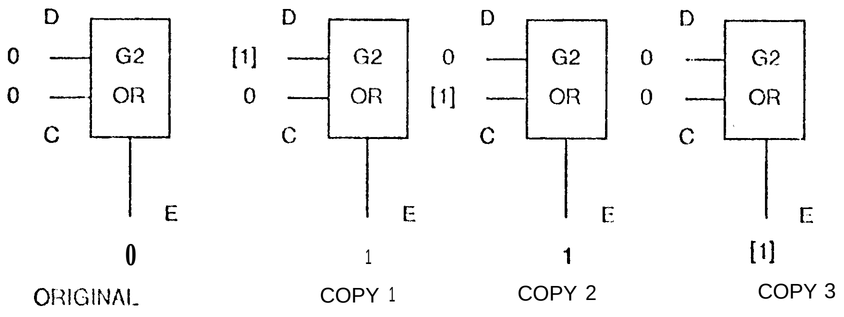


Fig 5-7: CONCURRENT FAULT SIMULATION IN ADLIB-SABLE

In each of the copies 1,2 and 3 the output node is forced to a faulty value. Hence the three faults d(s-a-l), c(s-a-l) and e(s-a-l) are stored in F(e). However a previously created fault list F(d) exists and recall that it was eariier said : if ever d(s-a-l) is detected at a later node then ail the faults in F(d) wilt be detected there too. The final answer is : stuck-at faults b(s-a-l), d(s-a-l), c(s-a-l) and e(s-a-l) are detectable.

5.2.2.3. Comparison

Observe that the step 1 of the deductive technique and step 1 in the last subsection are both quite similar. Steps 2 in both those cases are also similar. The conclusion is that implementations of the concurrent and the deductive technique in ADLIB-SABLE are both equivalent. This does not mean that those two techniques are same. The reason, their implementations in ADLIB-SABLE are equivalent is because both of them are forced to be done on a sequential machine. The deductive method is inherently sequential but the concurrent method can be much faster if processed on multiple processors. if in a multiprocessor machine, each copy is evaluated and compared against the original by a dedicated processor, the final outcome may be quite fast.

imagine there are N components and M nodes (where M and N would be of the same order). in a Deductive fault simulation implementation, let each fault model component take Td processing time. Hence the total tirne taken is N.Td since components are processed sequentially. Since there are M nodes there is a good chance of there being M stuck-at faults and hence M copies are to be evaluated in the Concurrent method. Also suppose there are K processors (where k is a reasonable number) and that each component takes Tc time to be logically evaluated and compared against the original. Tc wilt be much less than Td since in the Deductive case, a tot of complex analysis has to be done. Also assume that ail copies are evaluated right upto the end. This is indeed the worst case. in reality many of the copies wilt be dropped before they are completely evaluated and the freed processor can be alloted another copy from the bank of unprocessed copies.

One copy will take N.Tc time.

K processors process K copies in N.Tc time

Hence, M copies will take time of $\left\lceil \frac{M}{K} \right\rceil * N.T_c$

Timewise advantage of Concurrent over Deductive is $\frac{N.T_D}{\left\lceil \frac{M}{K} \right\rceil \cdot N.T_c} = \frac{T_D/T_c}{\left\lceil \frac{M}{K} \right\rceil}$

The actual advantage will also be less since inter-process communication might require some time in concurrent fault simulation. Also note that the deductive technique is best suited for a fast single processor machine whereas the concurrent technique can be best performed by a multiprocessor machine even having stower processing elements.

52.3. Limitations of Multivalued F.S.

in section 3.4, it was mentioned that a limitation of the functional fault model would be addressed later. it is done here. Many complex functional blocks like microprocessors do not have all the input combinations implemented. in other words, the complete input output boolean equation is not specified. The output behaviour is known for a certain set of input combinations. One can imagine a scenario where a certain stuck-at fault could alter the input pattern to an input combination not defined. With a 3-valued fault simulation the output(s) for an undefined input pattern can be treated as U. The fault model can now take care of unwarranted input combinations and it is the user's responsibility to interpret the results from simulating such functional blocks.

5.2.4. 3-Valued Deductive Fault Simulation

Subsection 5.2.1 discussed the 3-valued deductive model for a 2 input OR gate, the inputs of which were assumed primary. in that sense the model was incomplete. This section wilt redo the same example this time assuming nothing about the nature of the inputs.

The same OR gate with the same input values as in subsection 5.2.1 is redrawn in Fig 5-8.

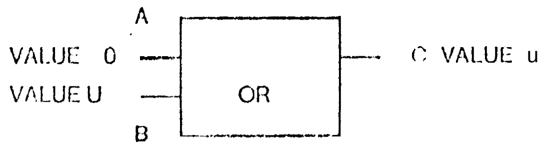


FIG 5-8: AN OR GATE

Fault lists associated with nodes A,B will be already existing. F0(A) and FU(5) are nil. FI (A), FU(A), F0(B), F1(B) may not be nil. Recall that the notations F0(X), F1(X), FU(X) mean there are stuck-at faults listed which can force the node X to 0, 1 and U respectively. The output C is U. Hence FO(C) -- C(s-a-0); F-i (C) = C(s-a-1). The fault lists for node C can be written.

If for some stuck-at faults node A is forced to 1, the output C will be 1. Hence F1(C) must contain F1 (A). Thus F1 (C) = C(s-a-1)VF1 (A). if for some faults , node B is forced to 1 then the output C will be 1 again. F1 (C) should therefore include F1 (B). Thus F1 (C) = C(s-a-1)VF1(A)VF1(U). If there are some faults common between F1 (A) and F1(B) then they have been included twice and must be eliminated. Finally F1(C) = C(s-a-1)VF1(A)V{F1(B) -F1(B) F1 (A)}. If node B is forced to 0 by some stuck-at faults, then output C will be 0. The fault list FO(C) must include F0(B). However, if any stuck-at fault in F0(B) is also common to F1(A) then for those faults, the output will be 1 instead. Hence these common faults have to be eliminated. it is also needed to eliminate fault lists common between F0(B) and FU(A) since for these faults the output C will be U. And U happens to be the correct output. FO(C) is finally rewritten as : FO(C) = C(s-a-0)V{F0(B)-F0(B) F1(A)-F0(B) FU(A)}

The fault lists are now complete. The reason one has to do such complex manipulation on the fault lists lies in reconvergent fanouts. Although only single stuck-at faults are considered, there cm be a stuck-d fault, earlier in the circuit, which may force more than one input of the same gate (or component) to go wrong. In the circuit below, if node a is stuck-at 0, both outputs A,5 go **faulty** and it is a case where two inputs of a NOR gate are faulty at once.

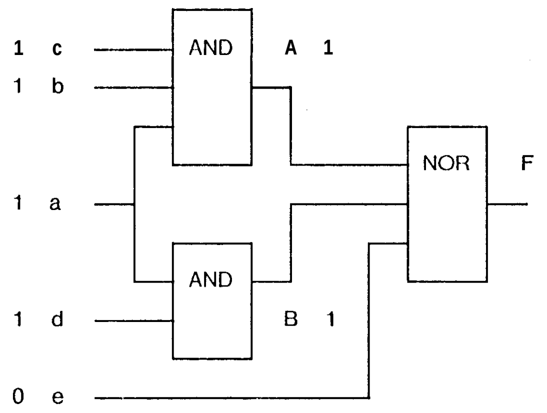


Fig 5-9: A CIRCUIT WITH RECONVERGENT FANOUT

52.5. Summary

Extensions to deductive fault simulation have been discussed which make it more applicable and suitable for a wider range of technologies. A 3-valued deductive fault simulation example was demonstrated which can be used as an example in extending the fault models from 2-valued to 3-valued fault models. Also concurrent fault simulation was evaluated and it was concluded that the structure of ADLIB-SABLE naturally supports both the deductive and the concurrent method. The next chapter quotes numbers from actually running simulations on various combinational and sequential fault models.

Chapter 6
Results and Analysis

Chapter 5 concluded that the structure of ADLIB-SABLE supports deductive fault simulation quite naturally. This chapter will quote several significant numbers from running fault simulation on some sequential and combinational blocks. Also quoted are numbers from running other simulators on the same example and the effectiveness of the technique is thereby brought out.

Some of the simulators used do not perform fault simulation. They are primarily meant for logic simulation either at gate level or at the circuit level. To perform fault simulation with these simulators, one would have to apply the input pattern, insert one stuck-at fault at a time, and then evaluate the entire circuit. If the final value differs from the correct value then that particular stuck-at fault will be detected. The method is repeated for each and every stuck-at fault that is of interest.

Also some of the sirnulators are run on different machines and are written in different languages. For a fair comparison, some benchmarks were run on different machines and the data was normalized. Some data was collected on running logic simulations in ADLIB-SABLE to compare the relative speeds of logic and fault simulation. The simulators used were the following:

ADLIB-SABLE : this is based on Pascal and runs on a Dee-20. Performs logic and fault simulation separately. It supports gate and functional models.

SALOGS : this is the Sandia Laboratories Logic Sirnulator which does only gate level simulation. It does logic simulation and fault simulation (one fault at a time), is implemented in Fortran **77** and runs on a Apollo work station.

DIANA : this does logic simulation only and is implemented in Fortran. It also runs on the Apollo and does gate level simulation.

Experimental comparison of several programs established the following relative run times :

Dee-20 = 8.3 times Apollo (Pascal)

Vax-750 = 2.5 times Apollo (Fortran 77)

6.1. Adder: Fault and Logic Simulation

Table 1A shows the execution times for fault simulation of 2-bit, 4-bit,6-bit and 8-bit Adder in two simulators. Table 1B shows the total time used by the simulators to process the circuit (read in network description,build up data structures) before simulation can be started. Table 2 shows the logic simulation times on the same examples this time on four simulators. For ADLIB-SABLE, numbers are quoted for simulation at the gate level and functional level. Graphs 1 and 2 are built from these tables and are shown below.

6.1 .1. Discussion

Logic simulation time for all simulation systems increase linearly with the number of components. In logic simulation, each component is simulated only once, hence the total time is proportional to the number of components simulated. The results are therefore as expected.

The fault simulation curve for SALOGS is non-linear. The order of non linearity is close to 2. The reason for this quadratic dependence on the number of components, lies in the method adopted by SALOGS to perform fault simulation. For each stuck-at fault at a node, **a** complete logic simulation run is done. A complete logic simulation time, as already commented on before, is proportional to the number of components. Since the number of stuck-at faults is proportional to the number of components, the total fault simulation time is proportional to the square of the number of components.

In the deductive technique, all on does is to manipulate fault lists. Each node has a fault list. Recall that while discussing the implementation of deductive fault simulation in chapter 2 it was mentioned that a gate (or component) is considered only once and all the fault lists corresponding to all its nodes are processed. The total simulation time therefore depends on the number of components times a factor which is the average number of nodes per

OF BIT ADDERS

	2	4	6	8
GATE LEVEL ADLIB-SABLE	io.3	16.8	22.4	29.1
FUNCTIONAL ADLIB-SABLE	3.7	4.7	5.7	7.3
SALOGS	11.3	22.2	38.8	63.3

TABLE 1 A: FAULT SIMULATION CPU TIME

OF BIT ADDERS

	2	4	6	8
GATE LEVEL ADLIB-SABLE	5.0	5.0	5.0	5.0
FUNCTIONAL ADLIB-SABLE	4.0	4.0	4.0	4.0
SALOGS	20.7	23.0	25.7	28.2

TABLE 1B: FAULT SIMULATION SET-UP TIME

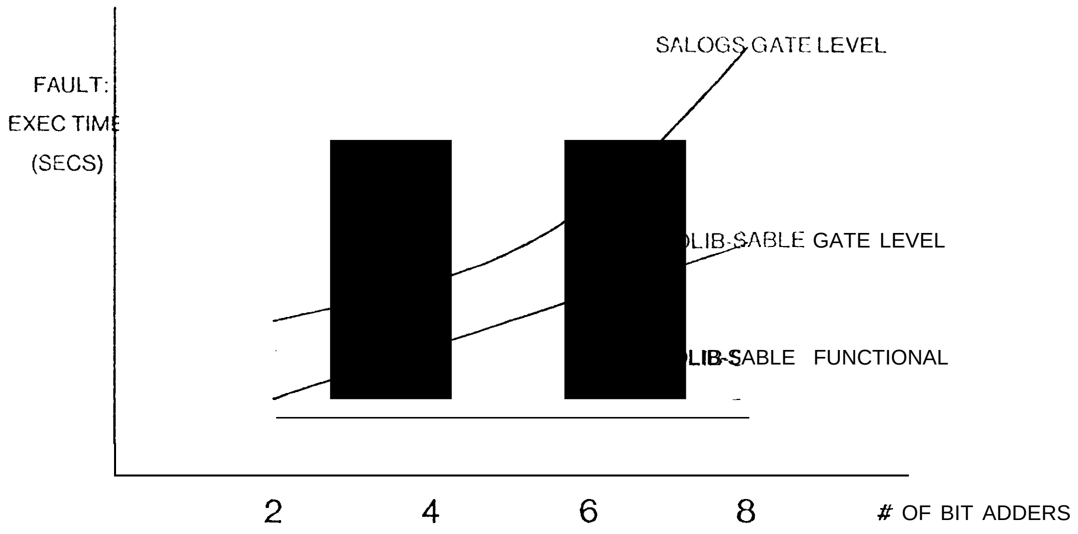
OF BIT ADDERS

	2	4	6	8
GATE LEVEL ADLIB-SABLE	5.8	9.1	11.6	14.9
FUNCTIOAL ADLIB-SABLE	2.4	3.7	4.1	4.5
DIANA	7.5	9.7	12	15
SAL.OGS	4.2	5.3	8.6	10.6

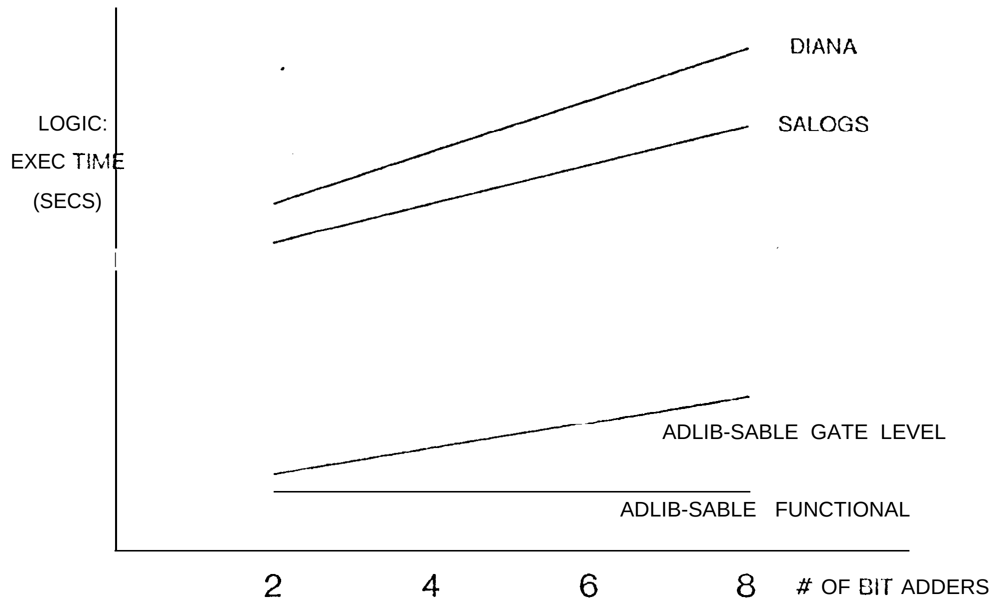
TABLE 2: TO-I-AL LOGIC SIMULATION TIME

component. Thus, simulation time in ADLIB-SABLE should linearly increase with increasing number of components and this is exactly what is seen.

Before the speed of simulations are compared seriously, note that some amount of initialization or building up of data structures is hidden in the simulation execution time. It would be logically anticipated that this initialization tirne is proportional to the number of



GRAPH 1: FAULT SIMULATION CPU TIME



GRAPH 2: LOGIC SIMULATION CPU TIME

components and would remain a constant irrespective of whether fault simulation was run with 1 input pattern or 10 input patterns. Such a logical anticipation is exact in ADLIB-SABLE. The execution times for running the simulation with 10 input patterns is listed in table 3.

OF BIT ADDERS

	2	4	6	8
GATE LEVEL ADLIB-SABLE	46.8	89.0	120.3	163.4
FUNCTIONAL ADLIB-SABLE	11.8	20.3	28.5	36.6
SALOGS	29.9	61.8	122.8	202.1

TABLE 3: FAULT SIMULATION WITH 10 INPUT PATTERNS

6.1.2. Analysis

From the fault simulation times for **1** input pattern (table 1) and 10 input patterns (table 3), one can deduce the program initialization time and the execution time per input pattern. The method is illustrated with an example and this is analogously applied across all data in tables 1 and 3.

Consider the case of the 2-bit Adder: 46.8secs for 10 input pattern
and 10.3secs for a single input pattern. Let program initialization time be
i and actual simulation time be s/input pattern.
Now, $i + s = 10.3$
$i + 10.s = 46.8$
Solving the two equations --> i = 6.3secs ; s = 4.0secs/input pattern

Table 4A contains the program initialization times while 4B contains the execution times/input pattern. Using these graphs 3 and 4 are constructed. The tables and the graphs are given below.Analysis : Based on the data structures and the procedures in ADLIB-SABLE fault simulation program, the initialization time should vary linearly with increasing number of components. The slope however, should be small. This is exactly observed in graph 4. SALOGS exhibits **a** non linear behaviour in its initialization time. The curves I and II in graph 3 support the earlier hypothesis that the actual execution time in ADLIB-SABLE Deductive method must vary leariy with increasing number of components. Curve V proves that for the SALOGS implementation of parallel fault simualation, the simulation time increases as the square cf the number of components.

FAULT SIM	# OF BIT ADDERS 2	4	6	8
GATE LEVEL ADLIB-SABLE	6.3 6.3	8.8	11.5	14.2
FUNCTIONAL ADLIB-SABLE	2.8	3.0	3.2	4
SALOGS	9.3	17.8	29.5	47.9

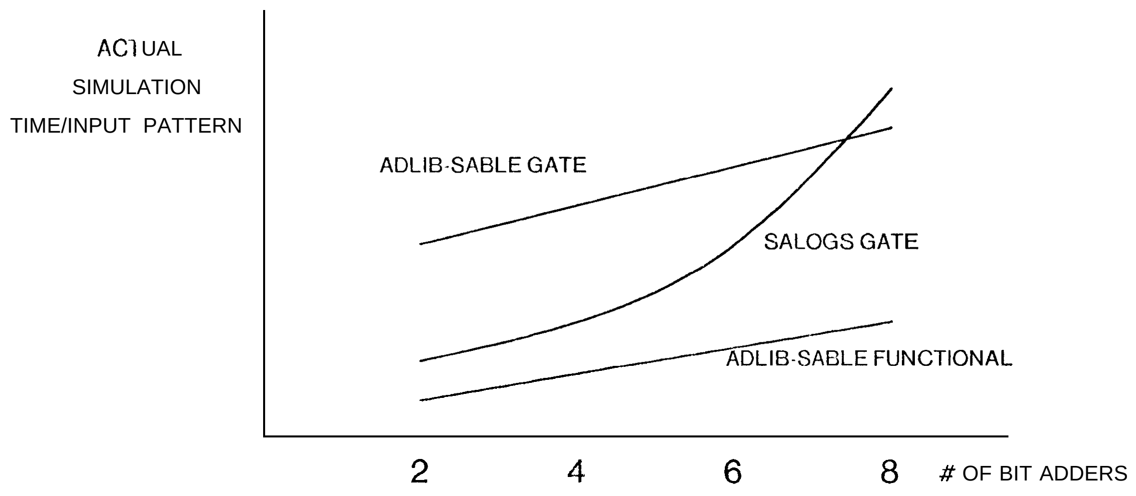
TABLE 4A: PROGRAM INITIALIZATION TIME

FAULT SIM	# OF BIT ADDERS 2	4	6	8
GATE LEVEL ADLIB-SABLE	4.0	8	10.9	14.9
FUNCTIONAL ADLIB-SABLE	0.9	1.7	2.5	3.2
SALOGS	2.0	4.4	9.3	15.4

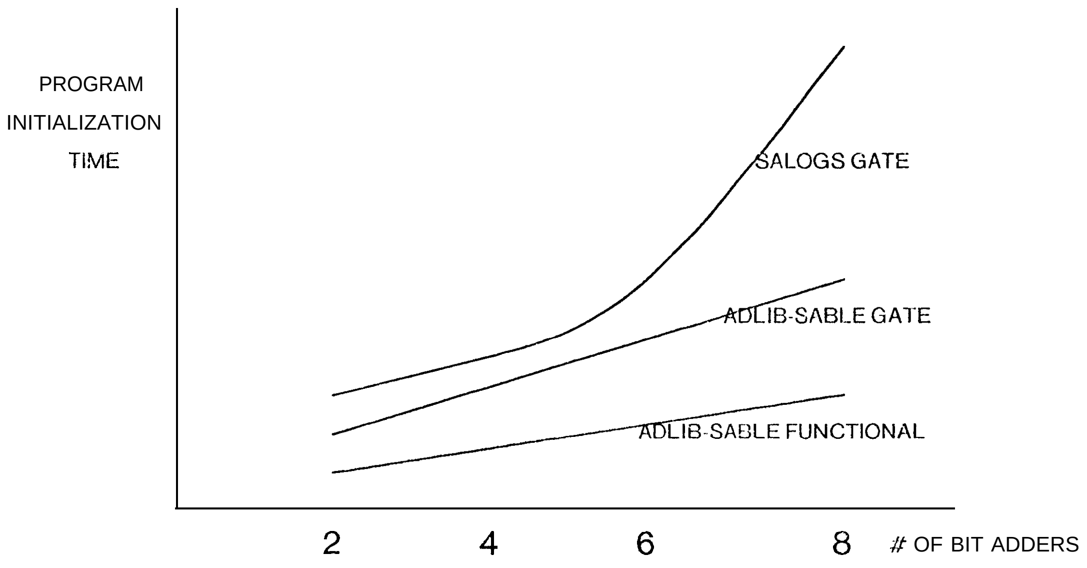
TABLE 4B: ACTUAL SIMULATION TIME

6.2. Asynchronous Counter

So far, the analysis of fault simulation has been confined to ADDERS which have a lot of parallelism in them. That is, a lot of the gates are fault simulated at once. Another example is considered, where there is little or no parallelism. In the case of an Asynchronous Counter, all the components are placed serially and each one is to be completely faullt simulated before



GRAPH 3: ACTUAL FAULT CPU TIME/INPUT PATTERN



GRAPH 4: PROGRAM INITIALIZATION TIME/INPUT PATTERN

the next one can be processed. Data has been taken only from running ADLIB-SABLE. Table 5 contains the data for 1 as well as 10 input patterns while the processed data (program initialization time and actual execution time) is stored in table 6. Graphs 6 and 7 are the plots of the contents of table 6.Analysis : Observe once again that the program initialization time as well as the actual execution time vary linearly with increasing number of components. Whether a component has parallelism or no parallelism in it, does not make any difference to

OF BIT COUNTERS

	4	8	12	16
1 INPUT PATTERN	0.6	1.4	2.3	3.3
10 INPUT PATTERNS	2.3	4.0	6.2	9.0

TABLE 5: FAULT SIMULATION EXEC TIME OF ASYN COUNTERS

OF BIT COUNTER

	4	8	12	16
ACTUAL EXEC TIME /INPUT PATTERN	0.2	0.3	0.4	0.5
PROGRAM INIT TIME /INPUT PATTERN	0.4	1.1	1.9	2.8

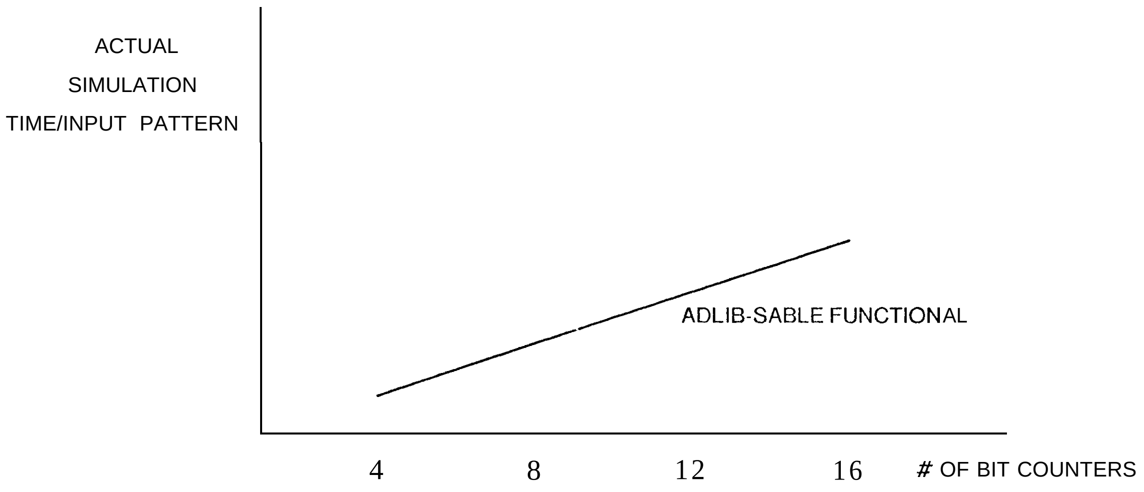
TABLE 6: PROGRAM INIT TIME, ACTUAL EXEC TIME

the iinearity. The reason is because ADLIB-SABLE runs on a single processor and any parallelism that might be present is killed by the sequential processing of the single processor.

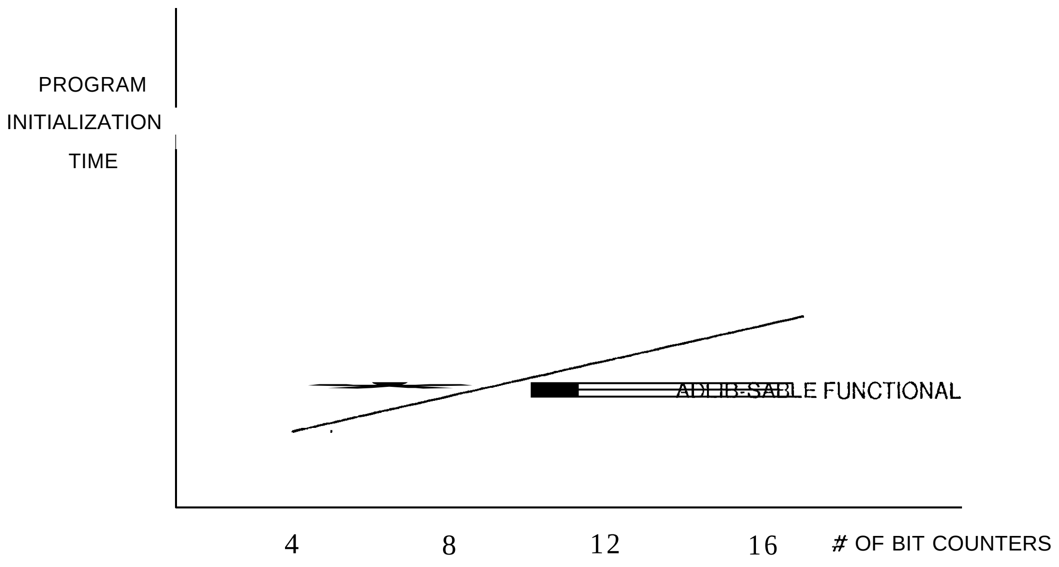
6.3. Comparing Functional and Gate Fault Models

This section quotes numbers from running gate and functional level fault simulation to see how effective the functional models are. The data is stored in table 7.

From table 7, observe that the timewise advantage of functional fault models varies between 3 and 5. The larger the number of gates that the functional model represents, the higher will be the timewise advantage. The 8-bit adder model, representing 84 gates, takes longer time to fault simulate than the 4-bit Synchronous counter model, representing 58



GRAPH 6: ACTCJAL FAULT CPU TIME/INPUT PATTERN



GRAPH 7: PROGRAM INITIALIZATION TIME /INPUT PATTERN

gates. The reason for this apparent inefficiency is that the synchronous counter is represented by a single functional block while the 8-bit Adder is made up by four 2-bit adder modules.

	GATE LEVEL	FUNCTIONAL LEVEL	RATIO
8 BIT ADDER(84 GATES)	19.9	4.4	4.3
4 BIT ASYN COUNTER(28 GATES)	7.9	2.6	3.0
4 BIT SYN COUNTER(58 GATES)	10.2	2.0	5.1

TABLE 7: COMPARISION OF GATE AND FUNCTIONAL SIMULATION TIMES

Chapter 7
Conclusions

The major conclusions can be summarized as follows :

1. A deductive fault simulator has been built in ADLIB-SABLE.

2. Gate level fault models have been built for most common gates. Although only single stuck-at 0 and single stuck-at 1 faults are considered, fault models can be easily extended to incorporate other single stuck-at faults.

3. Sequential circuits with feedback loops can be fault simulated. A design discipline has been worked out, namely breaking all feedback loops by at least an RS-latch, which when enforced, will enable quick and practical deductive fault simulation of large sequential circuits.

4. A decomposition discipline has also been developed, which renders functional fauit simulation of complex functional blocks. Functional fault models for many commonly used blocks are built. These fault models are invariant to their internal implementation and thus one can assume any convenient internal implementation of a functional block while building its fault model. This principle is also formally proved.

5. Deductive fault simulation is not confined to the [0,1] domain. It can be extended and it has been shown that fault models built in the extended domain can be very useful in modelling gates of certain technologies accurately.

6. A Comparative study between deductive and concurrent fault simulation methods reveal that the deductive technique is best suited for a fast single processor computer with large storage memory while the concurrent technique can be best exploited in a multi-processor machine with slower processing elements.

7. Experiments with deductive fault simulation of combinational and sequential circuits show that the fault simulation CPU time is linearly proportional to the number of components simulated. This is a great advantage over some simulators, which do one fault at a time and exhibit a quadratic increase in the simulation CPU time with the number of components fault simulated.

8. Functional fault simulation of typical-sized blocks were considered and they exhibit a simulation speed that is five times higher than if the simulation was

carried out at the gate level. This ratio will be larger for Functional blocks of larger size.

The limitations are as follows :

1. Fault models do not consider other types of faults such as bridging faults, intermittent faults, multiple faults etc.

2. Deductive fault simulation requires a large amount of storage. For this reason , large circuits will either have to be partitioned and each partition be fault simulated separately or a clever storage mechanism has to be invented. This work does not address that problem.

REFERENCES

1. Dwight Hill, "ADLIB Users Manual," Technical Report no 177. Computer Systems Laboratory,Stanford University,Stanford,CA-94305. Aug '79.

2. W.Cory,J.Duley and W.Vancleemput, "An Introduction to the DDL-P Language," Technical Report no 163. Computer Systems Laboratory,Stanford University. Mar '79.

3. W.Vancleemput, "An Hierarchical Language for the Structural Description of Digital Systems," Proceedings of the 14th Design Automation Conf. New Orleans '77. pp377-375.

4. M.A.Breuer and A.A.Friedrnan, "Diagnosis and Design of Digital Systems," Computer Science Press. Potomac,Maryland. 76.

5. T.I.Staff, "The TTL Handbook for Design Engineers", Texas Instruments Incorporated.76.

6. Jensen K. and Wirth N., " Pascal User Manual and Report," Springer Verlag,New York,74.

7. Organick E.,Forsythe A. and Plummer R., "Programming Language Structures," Academic Press, San Francisco. 78.

8. Dwight Hill, "Multi Level Simulator for Computer Aided Design," Ph.D. Thesis. C.I.S, Computer Systems Laboratory,Stanford University.

9. V.D.Agrawal,A.K.Bose, etal "A Mixed Mode Simulator," '80 Design Automation Conference. pp 618-625.

10. S.G.Chappel,P.R.Menon, etal "Functional Simulation in Lamp System," Journal of Design Automaiion and Fault Tolerant Computing, May 77. pp 203-215.

11 . D.Hill and W.Vancleemput, "SABLE : A Tool for Generating Structural Multilevel Simulation," Proceedings of the 16th Design Automation Conference,June 79. pp 272-279.

12. S.A.Szygenda and A.A.Lekkos, "Integrated Techniques for Functional and Gate Level Digital Logic Simulation," Proceedings of the 10th Design Automation Workshop, June 73. pp 159- 172.

13. D.B.Armstrong, "A Deductive Method of Simulating Faults in Logic Circuits," IEEE Transactions on Computers&lay 72. pp 464-471.

14. John P.Hayes, "Component Expansion Techniques in Computer Design," Digital Processes vol 4. '78. pp295-312.

15. E.G.Ulrich and T.Baker, “Concurrent Simulation of Nearly Identical Digital Networks,” Computer April ‘74. pp39-44.

16. Sheldon B.Akers, “On the Specification and Analysis of Large Digital Functions,” The Seventh Annual International Conference on Fault Tolerant Computing. June ‘77.

17. Narsingh Deo, “Graph Theory with Applications to Engineering and Computer Science,” Prentice-Hall Inc 74. pp 232.

18. Sandia Laboratories, “Sandia Logic Simulator-SALOGS,” New-Mexico.