Chapter 4

Fault injection and performance evaluation

If all the test routines are written correctly, there is no hardware design fault in the system and the system is modeled properly, then we should get the pass messages for all the components of the system.

- Any component is considered passed only when we get the pass message for it.
- If we get a fail message during the test for a specific component, then most likely the component is faulty, although there are usually several possible sources of error as given by the suspect sets for each test
- If we get a timeout message, then some part of the circuit is faulty but we do not know which part it is, or whether it is a chip or interconnection fault. But we know that it can only be in a component not tested until that point.
- If the system hangs-up indefinitely without giving any timeout message, then the most probable cause is an open/short in some interconnection, or a catastrophic failure in some chip such as a completely inoperable microprocessor chip.

4.1 Fault injection

To determine whether our test routines actually detect faults, we inject faults in the system. This is much easier to do in the VHDL model than in the hardware implementation. All the fault injections described in this report were done on the VHDL model. For systems modeled in VHDL, we change the architecture of a component so that its function is the same as that of a faulty component. In this project, most of the models we are using come from Synopsys Smart Model Library. They do not supply the VHDL source code for their models. So, we cannot make changes to the architecture of Smart Model Library components. We have to use other ways to test these components. To simulate stuck at faults on pins, and open/shorts on lines, we make the

corresponding changes on the SGE schematic of the circuit, reanalyze it and then simulate it. For the smart model components, we adopt the following approach for fault injection:

4.1.1 EEPROM

The EEPROM is being used for storing the system program. The effect of a fault is to change the contents of the EEPROM. Our test program calculates the checksum of the program from its contents, and then compares it with the expected checksum. So, we randomly changed the contents of one or more of the EEPROM locations. Our test routine detected the fault and gave the error message.

Pin Faults

The results of fault injection on pins are shown below:

Table 4.1 Fault injection on EEPROM pins

Fault	Result	Comments
OE s-a-0	HU	OE always zero, this corrupts the Address/Data bus
OE s-a-1	HU	System program not read since OE is disabled
WE s-a-0	HU	When CPU enters a read cycle, WE is also low. This causes Error Message from simulator also : "Value on OE pin went high during the write cycle, device behavior is unspecified"
WE s-a-1	ND	If in the original system, this error occurs after program loading in EEPROM, then it is undetectable, since nothing is written to EEPROM after that. In the simulator, mif file is loaded in EEPROM by some file I/O procedure, which does not involve WE pin, hence it is undetected in the simulator also
CE s-a-0	N/A	No fault, CE is already connected to GND
CE s-a-1	HU	Chip behavior same as OE s-a-1

OE: Output Enable WE: Write Enable CE: Chip Enable

HU: Hangs up ND: not detected

4.1.2 RAM

For the internal RAM of the 68HC11, we cannot inject a fault since it is internal to the microcontroller and we do not have its VHDL architecture since it comes from the Smart model library. Since it is fairly straightforward to make the VHDL model for a RAM, we made our own functional RAM model and mapped it into the external address space of the microcontroller. We applied the same March algorithm to test this external functional RAM and got the external RAM pass message. Next, we modify the architecture of this RAM so that it behaved like a faulty RAM. We make the faulty RAM architectures of all the faults shown in Table 4.2, one fault at a time. The results from fault injection on this external RAM are equally applicable to the internal RAM that we are actually using in the system. We achieved the address decoding for the RAM by using the same 74138 decoder, and an OR gate.

Table 4.2 shows the results for different types of faults:

Table 4.2 Injected faults for RAM

Fault	Result
Cell Stuck-at 0	D
Cell Stuck-at 1	D
$0 \rightarrow 1$ cell transition	D
$1 \rightarrow 0$ cell transition	D
Idempotent cell coupling	D
Inversion cell coupling	D
Address decoder faults (1+2)	ND
Address decoder faults (3+4)	D
Address decoder faults (1+3)	ND
Address decoder faults (2+4)	D

D : detected, ND : Not detected For explanation of RAM faults, see 3.6.1

All the stuck-at and transition faults are detected by the test. The idempotent and inversion coupling faults are also detected. The four types of address decoder faults discussed in 3.6.2 normally occur in combination. Hence the combinational address decoder faults 1+2, 3+4, 1+3

and 2+4 are injected. Theoretically, all the address decoder faults are detected by the March X algorithm, but in simulation, two of the AFs go undetected. There are two reasons for this:

1. The March X algorithm assures detection of all single address decoder faults but here we are testing with multiple AFs.
2. Another probable reason is the way in which the components are modeled in the Smart model library. We explain this with the help of address decoder combination fault (1+2). AF 1 is a fault in which no cell is accessed with a certain address, whereas in AF 2, a cell is not accessible with any address. This fault is the one in which no cell is accessed when we apply a certain address. The combination fault 1+2 is thus the one in which we apply a valid address and the corresponding cell is not accessed. It is AF 1 because no cell is accessed by the given address; it is AF 2 because the corresponding cell is thus not accessible with any address. Suppose when we apply address 32, the location 32 is not accessed. Now, the March X algorithm first makes all locations equal to zero. When it will reach 32, it will try to make it zero but since the cell is not accessible, it will fail to do so. Next, all the zeros are read and ones are written in their place. When location 32 is reached, the microcontroller will try to read the value from there and compare it with 0. Since the location is not accessible, the high impedance value of the bus would be read and will be treated as '0' or '1'. So, if it is treated as the value opposite to that of the expected value, we should get the fail message. But during simulation, this fault always goes undetected.

Next, we inject the pin faults. The results are shown in Table 4.3.

Table 4.3 RAM pin faults

Fault	Result
OE s-a-0	HU
OE s-a-1	ND
WE s-a-0	HU
WE s-a-1	ND
CE s-a-0	Already at permanent ‘0’, no effect
CE s-a-1	ND

D : detected, ND : Not detected,

HU : Hangs up

4.1.3 Microcontroller

For the microcontroller, we cannot inject faults in the internal registers and instructions because we are using the smart model. However, we can inject pin faults. The results of injecting different pin faults is shown in Table 4.4.

Table 4.4 Microcontroller pin faults

Fault	Result	Comments
Stuck at fault on Address/Data lines AD(7:0)	HU / TIMEOUT	If the fault is activated before entering any component test routine, it will be hang up, if we have entered the routine and then the fault is activated, we will get a timeout message
Stuck at fault on Address lines A(15:8)	HU / TIMEOUT	-------- do --------
IRQ s-a-0	ISR/No effect	If the I flag bit remains set in

		the program, then no effect of fault, but ISR executes as soon as I bit is cleared
XIRQ s-a-0	ISR/No effect	If the X flag bit remains set in the program, then no effect of fault, but ISR executes as soon as X bit is cleared
RESET s-a-0	HU	The system keeps on resetting and going to location FFFE
R/W s-a-1	HU	Failure to write anything in the external address space
R/W s-a-0	HU	No program byte read from EEPROM
MODA s-a-0	HU	System operates in single mode and does not recognize any external address
MODB s-a-0	HU	System operates in special test mode
E s-a-0	HU	
E s-a-1	HU	
AS s-a-0	HU	
AS s-a-1	HU	

4.1.4 74HC138 decoder

Input to the 74HC138 decoder are the address lines A14, A13 and the R/W line. The chip is enabled by the signals E and A15. All these signals are coming from the microcontroller lines. The faults on these lines are already covered during microcontroller testing. Similarly, the output lines of the decoder are going to the read, write and enable signals of external components like EEPROM and RAM. The faults on these control signals of external chips are covered during their testing.

4.1.5 74HC573 latch

The data and LE inputs for the latch are coming from the microcontroller which are tested there. The other input is OE which is permanently grounded. It was made stuck-at-1 and the result was system hang-up because data cannot be passed to the output of latch in case of OE not enabled. The output of the latch is going to the data inputs to the external RAM, EEPROM and PPI. If there is a stuck at fault at the latch output, this will result in either system hang-up or timeout.

4.2 Step by step results

Table 4.5 shows the step by step results of the start small approach applied to the 68HC11 based temperature monitoring system. From the table it is clear that the passed items set continues to increase while the suspect set continues to decrease as the test progresses.

Table 4.5 Step by step results

Routine	Purpose	PASS	FAIL
DISPLAY	Test the display	Microcontroller can fetch and decode instructions, latch and decoder are working P = {D,L,C}	One or more of any chip or the display is totally faulty, no message would be displayed SS = {D,L,DC,MICROCONTROLLER}
INITMICRO1	Test the instructions that are going to be used in EEPROM and RAM test, and also for COP reset code	All these instructions and registers are working P = {D,L,DC,I_1,R_1}	One or more of the instructions or the registers of microcontroller is faulty, RAM or ROM fail SS = {I_1,R_1,EEPROM,RAM}
ROMTEST	Test the external EEPROM	EEPROM is working P = [D,L,DC,I_1,R_1,EEPROM}	EEPROM is faulty SS = {EEPROM,RAM}
RAMSTART1	Test the internal RAM locations 00 – EF	RAM locations 00-EF are working P = {D,L,DC,I_1,R_1,EEPROM, RAM}	RAM is faulty SS = {RAM}
RAMSTART2	Test the internal RAM locations F0 – 1FF	RAM locations F0-1FF are working P={D,L,DC,I_1,R_1,EEPROM, RAM}	RAM is faulty SS = {RAM}
MPTEST	Test all the remaining instructions and registers of the microcontroller	The instruction set of microcontroller works completely P={D,L,DC,I_1,R_1,EEPROM, RAM, MICROCONTROLLER}	Microcontroller is faulty SS = {MICROCONTROLLER}

4.3 Performance evaluation

For the system running on an 8 MHz crystal, that is an internal frequency of 2 MHz, our test code takes 1.4943 seconds to execute. The individual times taken by the routines are shown in Table 4.6.

Table 4.6 Time taken by test routines

Purpose of routine	**Time taken (ms)**
Initialize display	1228.4
Check display	18.79
Initial Microcontroller Test	19.09
ROM Test	117.04
RAM Test 1	27.94
RAM Test2	29.17
Complete Microcontroller Test	53.75

To make reading from display convenient, a delay of about 3.5 seconds was introduced between successive display messages. This made the total execution time of the code to be equal to 23.144 seconds.

We injected internal faults in the RAM and changed the data of the ROM so that it represented an internal fault. We found our technique to be good in detecting and locating those internal faults. For interconnection or pin faults, mostly the system hangs up because it is not able to start the program. But this hang-up does indicate fault in the system although it cannot be located. Some interconnection or pin faults go undetected.

For legacy systems, in which the testing features are to be incorporated later, first we have to see how much empty memory is present in them. We then have to write our program keeping in mind the space available. The space limitation generally affects the fault resolution. For example, in our case, we initially wrote a program that detected the microcontroller faults to the instruction level, but it was taking more than 8 KB of ROM memory that we had available. So, we then reduced the code so that it could fit into the memory available. But then, we had to sacrifice the instruction level fault resolution. Instead, our program now can detect a microcontroller fault but cannot resolve it to the instruction level. This is favorably acceptable in most of the applications.