Chapter 3

Self test of the board

The components of the board that can be self-tested are only those to which stimulus can be applied and response can be checked, directly or indirectly, by the microprocessor. This excludes the possibility of testing the analog to digital converter since it cannot be directly written to. The items we can self-test are

1. Microcontroller
2. RAM
3. EEPROM
4. Latch
5. Decoder
6. Display

Out of these items, we test the microcontroller, RAM and EEPROM directly. The latch, decoder and the interconnections are checked indirectly during the test. The display checking approach is discussed later.

3.1 Start small approach

We use the start small approach in the self-test. This means that every part of the circuit and every instruction is considered working only after being tested. Now, when we start the testing process, nothing has been tested previously. The first thing we test is the display. If we get the message on the display, this means that the microcontroller is able to fetch and decode instructions. The address/data latch, address decoder and display are also working. At this point we can put display, decoder and latch in our passed items set. If the display test fails, we do not

know much. The fault can be in the microcontroller, EEPROM, display, or the interconnections. Next, we want to test the EEPROM which contains the system program. But the instructions and registers used for this test are still untested. So we need to test them first. If this test fails, it indicates either

1. a fault in the instruction under test,
2. a fault in the instructions used for testing this instruction, or
3. a fault in the any of the registers that were used during instruction execution.

But if the test passes, we put all the instructions and registers involved in our passed items set and we can then use them for testing other parts of the circuit. There is, however, a small possibility of the test passing even if the component is faulty, but this depends to a large extent on the quality of the test program. In this manner, we build our testing process.

The order of the items to be tested has to be carefully chosen. The main considerations are to pick that item first which is

1. least dependent on functioning of other parts of the circuit
2. other parts of the circuit depend on the functioning of this part

So, we test the components in the following order

1. Display
2. EEPROM
3. RAM
4. Microcontroller

Every pass or fail message is to be indicated by the display. So, we need to test the display first. The EEPROM contains the system program consisting of both the application program and the added self-test code. If the system program data is corrupted, the application program and/or the self-test code will not work correctly. So, we have to test the EEPROM next. After that, the RAM has to be tested since the self-test routines for the microcontroller use the RAM, and we have to make sure that it is working first. Then we test the microcontroller thoroughly, checking

all its instructions. Finally, assured of working of the program, RAM and microcontroller, we can test the peripheral, or I/O devices, provided we can both read/write from/to them. Normally we can either write or read to an I/O device, but not both. In that case, we have to devise some other method of testing the I/O devices. If we have two I/O devices such that one of them is the input and the other is the output and their data formats match, we can connect the two externally. Then we output the data through the output device which will then be input to the system through the input device. This is called a wrap-around test. Serial I/O chips like the UART, can be easily and efficiently tested using this method. The board which we are going to test in this project has an analog to digital converter (ADC) as an input device and the LCD driver as an output device. The ADC has a one line analog input while the LCD driver has an 8 line digital output. So, we cannot do wrap-around test in this case. To test these I/O devices, we then have to add some external circuitry. In this project, since we are investigating the merits/demerits of our self-test technique, we do not test the I/O devices since that would involve data transfer between the board and external added circuitry.

3.2 Approach used for fault detection

The self-test routines check the functionality of each component and microcontroller instruction. The approach used for different parts of the circuit is:

DISPLAY: A test message is sent out to the display

EEPROM: Checksum of the data contained in the EEPROM is calculated and verified against the expected checksum

RAM: March X algorithm applied to the RAM to detect faults based on a reduced functional fault model

MICROCONTROLLER: All the instructions and registers tested for their proper operation

LATCH: The 74HC573 latch is used to demultiplex addresses from the address/data line. This address is then applied to external components such as the EEPROM, the PPI and the I/O components. An observed error during testing of these components may be due to a fault in the latch. The suspect set for an EEPROM failure thus contains the latch.

DECODER: The decoder 74HC138 is used for address decoding of the external components. Using a reasoning similar to that given for the latch, we come to the same conclusion that the suspect set for an EEPROM failure also contains the decoder.

Now there can be three ways of fault manifestation during self-test:

1. First is that the self-test routine does not get the expected result due to the fault and thus generates the error message.
2. Second is that due to the fault, the routine and hence the system hangs up. To take care of this second case, we have included the COP (computer operating properly) option of the 68HC11. The COP system is enabled at the start of every routine. This starts the count down of a timer. If the routine works properly, it should be finished before the end of count down time. But if due to some error, the system hangs up or the program flow is altered, the routine may be still unfinished at the end of count down time. In that case, a system reset is executed by the COP system generating a timeout message for the LCD display. The process for using the COP system is as follows:

(i) Set the NOCOP bit in the CONFIG register to zero. This enables the COP reset mechanism.

(ii) The next step is to select the length of period for which the watch dog timer runs before time-out. This is done by writing to bits CR1 and CR0 in the OPTION register. The following options are available for a 4 MHz crystal :

Table 3.1 Time-out options for watch-dog timer

CR[1:0]	Time-out
0 0	32.768 ms
0 1	131.072 ms
1 0	524.28 ms
1 1	2.098 s

To select appropriate values for CR[1:0], we should know the execution time of our subroutines. This can be done easily by first simulating the subroutines on a VHDL simulator/debugger and noting the time from the waveform viewer. The time taken by a routine can also be calculated manually by adding the number of clock cycles each instruction takes, and then multiplying the result by the time period of the clock.

(iii) Just before the start of a subroutine, "arm" the COP reset mechanism. Arming the COP reset mechanism means enabling a down counter and loading it with a count value. The counter starts down counting and will trigger the reset if it is not disabled before reaching a zero value. This disabling of COP is also called clear COP reset operation. The test routine is executed between the arming and clearing operations so that if the test routine takes more time than the time out period of the counter, a COP reset occurs. Arming is done by writing 55H to the COPRST register, whereas clearing is done by writing AAH to the COPRST register.

3. A third type of fault manifestation is for the system to hang up because it cannot execute the program instructions at all and hence it cannot give the pass/fail or the timeout message. This type of fault is generally due to complete failure of some chip, open /short or stuck at fault in the interconnection circuitry. Such a fault is hence detected but no diagnostic information is produced.

3.3 Display Test

We start our testing with the Display test. As already mentioned, if we get a pass message for this test, this means that the microcontroller can fetch and decode instructions, and the latch and decoder are working. So, if the test passes, we put all these in our passed items set.

$$P = \{D, L, DC\}$$

where P = passed items set
D = display
L = Latch
DC = Decoder

If the display is working, this also means that the portion of EEPROM which contains display test instructions, as well as the test instructions themselves are working. But we cannot put the EEPROM in the passed items set because it is not fully tested. A thorough test is needed to fully verify the EEPROM contents.
If this test fails, we have no idea at this point about the location of the fault. So, our suspect set SS is

$$SS = \{D, L, DC, MICROCONTROLLER, EEPROM\}$$

3.4 Initmicro Test

Next, we are going to test the EEPROM and RAM. But before that, we need to test the instructions used in these tests. The Initmicro test is for testing these instructions.

Initially, we select a register and test its corresponding load and store instructions. For example, we select register A and test the instructions LDAA and STAA. But we also need two more instructions here, CMPA for comparison between the actual and expected values and a branch instruction BNE which has to branch depending on the result of the test. Also, to display the results to the LCD driver, we need to send out ASCII data. The display routines make use of JSR, JMP and RTS instructions, apart from the load and store instructions. Upon successful completion of the test, we put these instructions also in our passed items set. However, we should know the following points:

- If there is an error in JMP we will still get the error message although the microprocessor will not halt in case of error.
- An error in RTS will cause the program to lose its normal flow and the system can actually hang up.
- If JSR fails, then we will not be able to enter the microcontroller testing routine in the first place. Once we are in the controller testing routine, this means that JSR is working,

If there is an error in the load, store, compare instruction or the register itself, we may not get the error message if BNE is faulty and fails to branch. So, immediately after passing the load, store, compare instructions for register A, we test the BNE instruction. Now, there are two possibilities:

- Load, store and compare instructions and register A was working. If BNE is faulty, we will detect the fault here
- Any of the load, store, compare instructions or register is faulty. In that case, BNE may be faulty and escape detection. But with the load, store, compare instructions used so frequently in the program, we will detect the fault somewhere, although it may not be properly located.

The instructions and registers tested are shown in Table 3.2.

Table 3.2 Registers and Instructions tested by Initmicro test

Registers (R_1)	A,B,D,X
Instructions (I_1)	LDAA,STAA,CMPA.,LDAB,STAB,CMPB,LDD,STD,CPD, LDX,STX,CPX,BNE,JMP,CLC,BCS,SEC,ADCB,INX,DEX,ORAA,TAP, TPA,RTS

If the test passes, the passed items set would be:

$$P = \{D, L, DC, I_1, R_1\}$$

where I_1 = Instructions used and tested in the initmicro test

R_1 = Registers used and tested in the initmicro test

If the test fails, the suspect set would be :

$$SS = \{I_1, R_1, EEPROM, RAM\}$$

3.5 EEPROM Test

The goal G12 of testing EEPROM is satisfied by verifying the checksum which is the test group TG121. For both off-line and on-line schemes, the only testing that can be done for read-

only-memories consists of verifying that the device contains the original contents. To this end, the approach normally used is to confirm a checksum. A checksum is a data value that is formed by doing a mathematical operation on all the data in the memory. For a faulty memory, the new checksum would most likely be different from the one calculated for the fault free memory. A common method to form the checksum is modulo-2 addition of all the data elements of the memory. The modulo-2 addition can be implemented in software by the XOR operation. The problem with this technique is that in case of an EPROM / EEPROM any even number of stuck-at faults in a column go undetected. Another method is to use the ADD instruction to add all the data elements of the memory. In that case, an error in any column would either cause the corresponding checksum bit or the carry to be in error. Thus if the error is in the MSB, the error information contained in the carry may be lost. We finally decided to use the "add with carry" operation on all elements of the data. This can be implemented with the ADCA instruction of the 68HC11 microcontroller. Using ADCA for checksum, all single stuck at faults would be detected and the carry problem on the MSB would be eliminated. However, two complementary faults occurring in the same column would remain undetected. But due to the physical nature of EEPROM/EPROMs, this is extremely unlikely; normally the faults occur in only one direction. Depending on the data present in the memory, multiple stuck at faults in the same direction can also go undetected.

We used the checksum checking approach to test the 8 K external EEPROM that contains the program code in our system. The checksum is first calculated using a high level language program that accepts the object file of the program as the input. We call the result obtained as the checksum word and place it in a memory location. Then during self-test, we add all the bytes including the checksum word using the ADCA and compare the result with the already computed checksum. When computing the checksum, the memory location in which it was subsequently placed was empty. So, after putting the checksum byte there, the new checksum value would be different from its previously calculated value. In our comparison operation, if we are still comparing with the old checksum, the test would always fail. To avoid this, we:

1. write the self-test routine such that it compares the checksum with a predetermined value say X, not the actual value of the checksum
2. put a value at the checksum location that will make the total result equal to the predetermined value X of step 1.

Here, we choose X to be equal to AA H. Let, our computed checksum be equal to Y and let the value that we want to put at the checksum location be equal to Z. In this case,

$$Z = X - Y = AA - Y$$

If the memory whose checksum we are going to confirm is other than the system memory, then if we select the value of X to be 00 or FF, then an all stuck-at-0 or all stuck-at-1 memory respectively will pass the test. So, a good choice is to choose a value that contains both 1's and 0's so that errors of this type can be detected.

The development of self-test code for the EEPROM proceeded as follows:

1. Write a C program that computes the checksum of the memory data based on the ADD WITH CARRY operation performed on the contents of all memory locations. The input to the program is the memory image file (.MIF) that contains the address and value of each location of the memory. The program first strips off the address header from the data and then adds all the data with carry. At the end of this step, we have the sum of the data based on add with carry operation (Y). Next, we subtract it from the value selected for the comparison (AA). So, we get the result Z which we would place at the checksum location. This program is given in Appendix B.
2. Fill all the unused locations of EEPROM with zero in the memory image file. Put the value of Z, as calculated in step (1) at the last memory location. The zeroing operation is necessary when running the code from the VHDL model because the VHDL debugger/simulator treats unused memory locations as having undefined 'U' logic state and this causes problems in computing the checksum during self-test. Zeroing of all unused memory locations is also required in case of the physical EEPROM because otherwise the garbage data in unused locations would make the result incorrect.
3. Write a self-test routine for the EEPROM in assembly language to recalculate the checksum during self-test. In this routine, all the locations are added using the ADCA instruction. With

the value of Z placed at the checksum location, the result we would get will be AAH. In the comparison instruction, we compare the checksum with AAH.

4. If an error occurs, display the error message and stop. If there is no error, give the pass message and continue. The EEPROM test pass message indicates that the EEPROM is fault free.

The passed items set and the suspect set at this point are:

$$P = \{D, L, DC, I_1, R_1, EEPROM\}$$

$$SS = \{EEPROM, RAM\}$$

However, if there is a fault message at this stage, the most probable cause of failure is EEPROM rather than RAM because very few RAM locations are used in this test.

3.6 RAM Test

One should accept from the beginning that it is impractical to give a 100 % test to prove that a RAM operates correctly in all combinations within the specification. Some fundamental calculations on the RAM illustrates the futility of attempting to gain 100 % confidence [FeeW77]:

- A RAM containing 4096 cells can contain any one of 2^{4096} different data patterns
- Without using the same address twice in an addressing sequence, 4096! (factorial) different sequences are possible. A comprehensive pattern would probably address each cell many times, further increasing the number of required address sequences.
- Timing variations, temperature, duty cycle, refresh rate (for DRAMS), etc. are additional variables.

However, with judicious application of product knowledge, and appropriate modeling of the memory, one comes up with a fault set that includes most common faults, yet is simple enough so that the testing algorithms can be applied in a reasonable amount of time.

3.6.1 RAM models

Modeling is the simplification and structuring of an entity and its environment. Modeling a system means introducing a level of abstraction while describing its behavior. The higher the level of abstraction, the further one is away from the physical level at which the system is actually implemented. This means that an event or phenomenon at a higher level cannot necessarily be related to a particular lower-level event or phenomenon. Also, many low level events and phenomenon can map to the same higher level event or phenomenon. Going from highest to lowest levels, the types of models are: behavioral, functional, logical, electrical and geometrical. [GoorA91]

Most tests for faults in memory chips are based on a simplified functional model. Hence we will mainly discuss the functional RAM model here.

A functional model for a RAM chip is shown in Figure 3.1 [GoorA91]. Block A, the address latch, contains the address. Higher order bits of the address are connected to the row decoder, B, which selects a row in the memory cell array, D. The low order address bits go to the column decoder, C, which selects the required columns. The number of columns selected depends on the number of data lines in the chip, which determines how many bits can be accessed during a read or write operation.

When the R/W line indicates a read operation, the contents of the selected cells in the memory cell array are amplified by the sense amplifiers, F, loaded into the data register, G, and presented on the data-out lines. During a write operation the data on the data-in lines are loaded into the data register and written into the memory cell array through the write driver, E. Often, the data-in and data-out lines are combined to form bi-directional data lines.

The chip enable line enables the data register and together with the R/W line, controls the write driver.

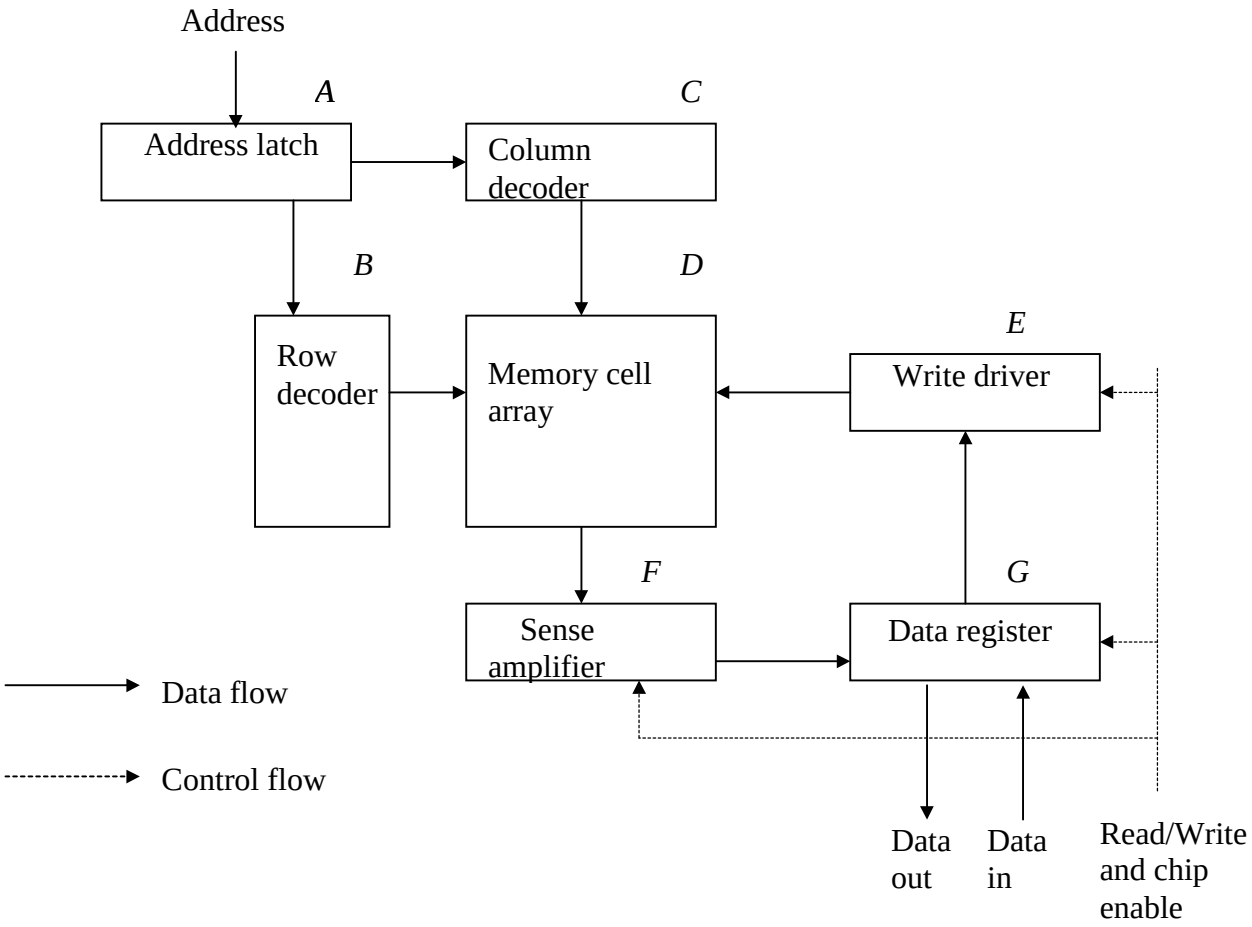

Figure 3.1 Functional model for a RAM chip.

In the chip of Figure 3.1, many different faults can occur as shown in Table 3.3. [GoorA91]:

Table 3.3 Functional RAM faults.

	Functional fault
A	Cell stuck
B	Driver stuck
C	Read/write line stuck
D	Chip-select line stuck
E	Data line stuck
F	Open in data line
G	Short between data lines
H	Crosstalk between data lines
I	Address line stuck
J	Open in address lines
K	Shorts between address lines
L	Open decoder
M	Wrong access
N	Multiple access
O	Cell can be set to 0 but not to 1 (or vice versa)
P	Pattern sensitive interaction between cells

Although a chip manufacturer may be interested in locating faults within the RAM chip, the test engineer is interested only in detecting the faults to the chip level so that it can possibly be replaced. With fault detection and not location within the chip as the goal, the model of Figure 3.1 can be simplified to the model shown in Figure 3.2. We reduce the functional model so that it consists of just three blocks: the address decoder, the memory cell array and the read/write logic.

The address latch A, the row decoder B and the column decoder C are combined to form the address decoder, because they all concern addressing the right cell or word. The write driver, E, the sense amplifier F and the data register G are combined in the read/write logic because they all involve the transport of data from and to the memory cell array. Using this model, we get a new fault list which can be mapped onto the faults of Table 3.3

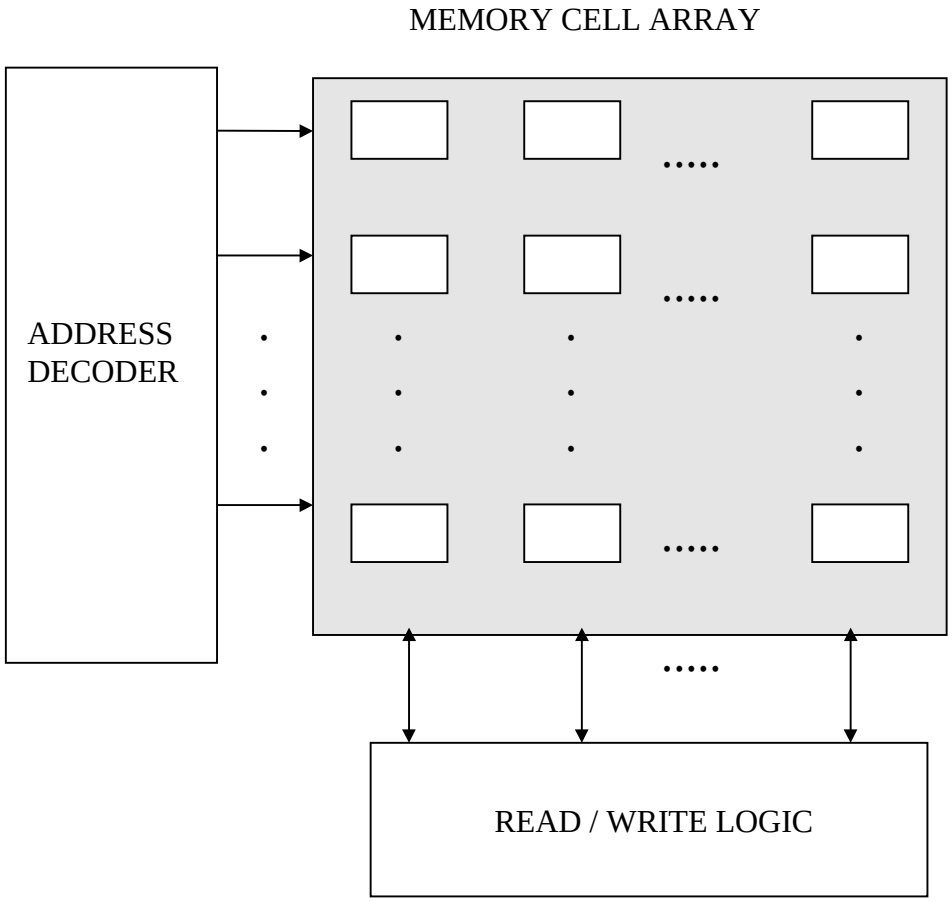

Figure 3.2 Simplified funcional model for a RAM memory

First we define the reduced functional faults[GoorA91]:

- Faults in which a single cell is involved. These are stuck-at and transition faults.
- Faults in which two cells are involved. These are the coupling faults.
- Faults involving ‘k’ cells.

    - The k cells are allowed anywhere in the memory. These are the k-coupling, the bridging and the state coupling faults
    - The k cells are clustered together in a physical neighborhood. These are the neighborhood pattern sensitive faults.
- Faults concerned with the address decoder

Stuck-at fault. The stuck-at fault (SAF) causes the logic value of a stuck-at cell or line to be always 0 (a SA0 fault) or 1 (a SA1 fault). A test that has to detect and locate all SAFs must satisfy the following requirement:
For each cell, a 0 and a 1 must be written and read.

Transition fault. A cell or line which fails to undergo a 0→1 transition when it is written is said to contain an up transition fault; similarly, a down transition fault is the inability to make a 1→0 transition. A test that has to detect and locate all transition faults (TF) should satisfy the following requirement:
Each cell must undergo an ↑ transition (state of the cell goes from 0 to 1), and an ↓ transition (state of the cell goes from 1 to 0), and be read after each transition before undergoing any further transitions.

Coupling fault. If a write operation which generates an ↑ or ↓ transition in one cell changes the contents of a second cell, it is a coupling fault (CF). If it involves only two cells, it is a 2-coupling fault which is a special case of the more general *k-coupling fault.* A k-coupling fault uses the same two cells as the 2-coupling fault, and in addition, only allows the fault to occur when another (k-2) cells are in a certain state. There are various types of k-coupling faults, which are briefly described as follows:

The *Inversion coupling fault (CFin)* occurs when an ↑ or ↓ transition in one cell inverts the contents of a second cell. A test that has to detect all CFins should satisfy the following requirement:

For all cells which are coupled cells, each cell should be read after a series of possible CFins may have occurred (by writing into the coupling cells), with the condition that the number of transitions in the coupled cell is odd (i.e., the CFins do not mask each other).

An *Idempotent coupling fault (CFid)* is defined as follows: An ↑ or ↓ transition in one cell forces the contents of a second cell to a certain value, 0 or 1. A test that has to detect all CFids should satisfy the following requirement:
For all cells which are coupled cells, each cell should be read after a series of possible CFids may have occurred (by writing into the coupling cells), in such a way that the sensitized CFids do not mask each other.

The *Bridging coupling fault (CFb)*, consists of an electrical connection (called *Bridge*) between two (or more) cells or lines. It is a bi-directional fault in that the state of a line or cell and not the transition causes a fault. Depending on the logic value of the bridge, we can have an *AND Bridging fault* or an *OR Bridging fault.*

There is yet another type of coupling fault: the *State Coupling fault (CFs).* A coupled cell or line, i, is forced to a certain value, x, only if the coupling cell or line is in a given state, y.

Neighborhood pattern sensitive fault. *A Pattern Sensitive Fault (PSF)* is defined as follows: The contents of a cell, or the ability to change the contents, is influenced by the contents of all other cells in the memory. The PSF can be considered the most general case of the k-coupling fault, namely the case whereby k=n (n represents all cells in the memory). The *neighborhood* is the total number of cells involved in a particular fault model. The *base cell* is the cell under test. The neighborhood with the base cell excluded is called the *deleted neighborhood*. The PSF model allows the deleted neighborhood to take on any position in the memory array. When the deleted neighborhood is allowed to take on only a single position (such that it is surrounding the base cell), one speaks about a *Neighborhood pattern sensitive fault (NPSF)*. The NPSF model is therefore a subset of the PSF model. Figure 3.3 shows a 5 element neighborhood. The 9 element neighborhood is also common.

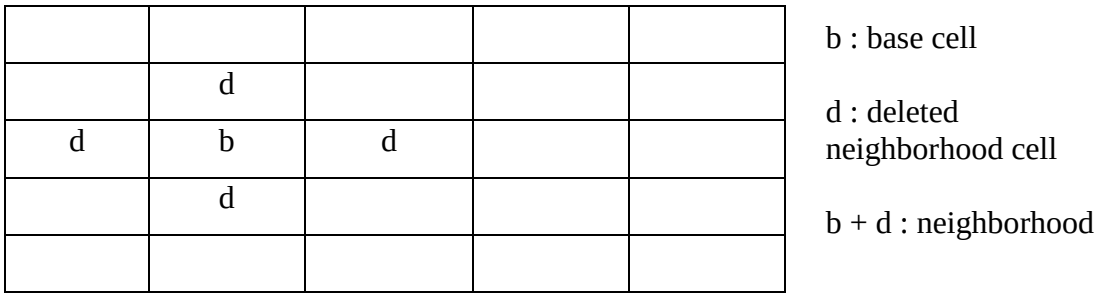

Figure 3.3 Illustration of a 5 element neighborhood

Three types of NPSFs can be distinguished :

Active NPSF (ANPSF), or a *dynamic NPSF*: the base cell changes its contents due to a change in the deleted neighborhood pattern. A test that has to detect and locate ANPSFs should satisfy the requirement:
Each base cell must be read in state 0 and in state 1, for all possible changes in the deleted neighborhood pattern.

Passive NPSF (PNPSF): the contents of the base cell cannot be changed (it cannot make a transition) due to a certain neighborhood pattern. A test that had to detect and locate PNPSFs should satisfy the requirement:
Each base cell must be written and read in state 0 and in state 1, for all permutations of the deleted neighborhood pattern.

Static NPSF (SNPSF): the content of a base cell is forced to a certain state due to a certain deleted neighborhood pattern. A test that has to detect and locate SNPSFs should satisfy the requirement:
Each base cell must be read in state 0 and in state 1, for all permutations of the deleted neighborhood pattern.

Address decoder faults. Address decoder faults (AF) can take the following four forms:
Fault 1:With a certain address, no cell will be accessed.
Fault 2: A certain cell is never accessed

Fault 3: With a certain address, multiple cells are accessed simultaneously

Fault 4: A certain cell can be accessed with multiple addresses.

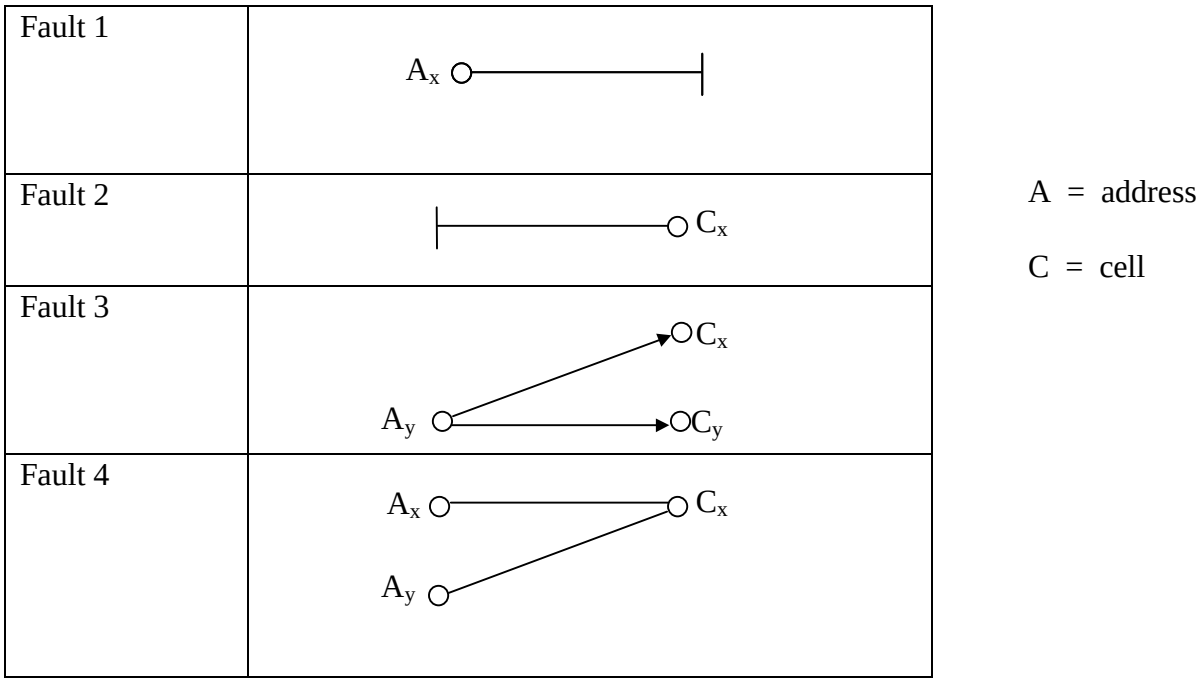

Figure 3.4 Address decoder faults

Another important concept is that of *linked* and *unlinked faults*. When multiple faults occur in a chip, if a fault influences the behavior of other faults, then it is a linked fault; otherwise it is an unlinked fault. Linked faults may be of the same type or of different types.

3.6.2 Relationship between functional and reduced functional faults

The next step is to map the reduced functional faults to the set of faults for the functional model given in Table 3.2. This mapping is given in Table 3.4

Table 3.4 Mapping reduced functional faults to functional faults.

Reduced functional fault		**Functional fault**
SAF	a	Cell stuck
SAF	b	Driver stuck
SAF	c	Read/write line stuck
SAF	d	Chip-select line stuck
SAF	e	Data line stuck
SAF	f	Open in data line
CF	g	Short between data lines
CF	h	Crosstalk between data lines
AF	i	Address line stuck
AF	j	Open in address lines
AF	k	Shorts between address lines
AF	l	Open decoder
AF	m	Wrong access
AF	n	Multiple access
TF	o	Cell can be set to 0 but not to 1 (or vice versa)
NPSF	p	Pattern sensitive interaction between cells

3.6.3 Testing strategy for RAM

The main points we kept in mind while developing the test strategy for RAM were

- Fault coverage should be as high as possible
- Execution time should be as short as possible

Based on that we reviewed all the existing algorithms for RAM testing. These algorithms can be divided into the following categories

- Traditional tests, which have been used extensively in the past, and are not based on a functional fault model [GoorA91]. Examples are Zero-One, Checkerboard, GALPAT, Sliding Diagonal & Butterfly tests.
- Tests for stuck-at, transition and coupling faults. This group of tests are based on the reduced functional fault model. They test for faults in the read/write logic, the memory cell array and the address decoder. These algorithms are said to belong to the family of *March tests*.
- Tests for Neighborhood pattern sensitive faults. This group of tests is also based on the reduced functional fault model. They test for NPSF in the memory array, and possibly also in the read/write logic. The address decoder is tested separately.
- Other memory tests, like Moving inversions (MOVI) which is a test for detecting coupling faults sensitized by (a) read operations, (b) transition as well as non-transition write operations.

Most of the well known traditional tests either take too much time, or their fault coverage is not adequate. For example, the Checkerboard test takes comparatively less time since it has an execution time of the order of n, where n is the number of cells in the memory, but it can only detect the Stuck-at-faults. On the other hand, the GALPAT test detects SAF,TF,CF and AF but its execution time is of the order $O(n^2)$.

In order to apply the neighborhood pattern sensitive faults, we should know the neighbors of the cell, i.e. the *physical layout* of the memory cell array has to be known. Normally within most memory chips, the address lines, the memory rows and the columns are scrambled; so, cells with logically adjacent addresses will probably not lay adjacent to each other. Unfortunately, the physical layout is most often only known to the chip manufacturer who may not be interested in publishing the *scrambling table*, which describes the relationship between the logical addresses and the physical addresses. When the scrambling table is not known, a test of NPSF is less useful [GoorA91]. Hence the conclusion is that an NPSF test may be applied by the manufacturer while developing or troubleshooting his chips, but not by the test engineer who is given the board containing that chip.

So, the best option is the group of March tests. Table 3.5 shows a comparison of fault coverage and execution time of different March tests [GoorA91]:

Table 3.5 Comparison of different March algorithms

Algorithm	**Fault coverage**					**Test time**	
	AF	SAF	TF	CF	Others	Order	1M memory
MATS	DS	D	-	-	-	O(n)	0.42s
MATS+	D	D	-	-	-	O(n)	0.52s
Marching 1/0	D	D	D	-	-	O(n)	1.5s
MATS++	D	D	D	-	-	O(n)	0.63s
March X	D	D	D	D	Unlinked Cfins	O(n)	0.63s
March C-	D	D	D	D	Unlinked Cfids	O(n)	1.0s
March A	D	D	D	D	Unlinked CFs	O(n)	1.6s
March Y	D	D	D	D	Linked TFs	O(n)	0.85s
March B	D	D	D	D	Linked CFs	O(n)	1.8s

We note that the March X, C-, A, Y and B tests detect AF, SAF,TF, CF and one other type of fault which is different for each test. Since we do not have any data on the probability of occurrence of these faults, we selected the one which had the least execution time. So, we decided to use March X test for our self-test program.

3.6.4 March X Algorithm

The March X test detects all AFs, SAFs, TFs not linked with CFins and unlinked inversion CFs. The proof that the March X test detects all these faults can be found in the reference [GoorA91].

The March X algorithm is given below:

```
for i in 0 to n-1 do
     A[i] := 0;
end
for i in 0 to n-1 do
     read A[i];
     compare (A[i],0);
     Branch FAULT if not equal;
     A[i] := 1;
end
for i in n-1 to 0 do
     read A[i];
     comp (A[i],1);
     Branch FAULT if not equal;
     A[i] := 0;
end
for i in 0 to n-1 do
     read A[i];
     comp (A[i],0);
     Branch FAULT if not equal;
end
```

The March X algorithm consists of the following steps:

1. All the memory is filled with zeros in the ascending order.
2. In the ascending order, these zeros are read and ones are written in place of them
3. In descending order, these ones are read and zeros are written in place of them
4. In ascending order, the zeros are read.

3.6.5 March X algorithm applied to RAM testing

Based on the reduced functional fault model of the RAM, we define four test groups for RAM, which are stuck-at faults (TG121), transition faults (TG122), Coupling faults (TG123), address decoder faults (TG124). All these test groups and hence the RAM test goal G12 is satisfied by applying the March X algorithm.

In our system, the March X algorithm is coded in assembly language and then is applied at start-up to the internal RAM of the 68HC11 microcontroller. This internal RAM is 256 bits wide for the 68HC11A8 used in the VHDL model, and 512 bits long for the 68HC11E9 used in the hardware implementation of the system. A major hurdle to the straightforward application of the March X algorithm to the RAM is the location of the stack. The stack must be located somewhere in the RAM. The March X test fills the whole memory with 1's or 0's during its application and then reads it. Since the contents of the stack are also overwritten, we will lose program control. So, to avoid this, we partition the RAM for testing purposes as:

For VHDL model,

Partition 1: 00 - EF, Stack Pointer: FF

Partition 2: F0 – FF, Stack Pointer: 0F

For hardware implementation,

Partition 1: 00 - EF, Stack Pointer: FF

Partition 2: F0 – 1FF, Stack Pointer: 0F

When we are testing for partition 1, we locate the stack in partition 2. And when testing for partition 2, we locate the stack in partition 1. This scheme gives a fault coverage slightly less than the original method, since we cannot check for coupling faults between the cells located in different partitions. The passed items set and the suspect sets after the execution of the RAM test are:

$$P = \{D, L, DC, I_1, R_1, EEPROM, RAM\}$$

$$SS = \{RAM\}$$

3.7 Microcontroller Test

Microprocessor, microcontroller and other instruction oriented devices are the most complicated of all LSI devices and, therefore, need special attention. A typical microprocessor / microcontroller contains the following elements:

Program counter: Contains the address of the memory which contains the next instruction to be fetched.

Accumulator: Register to store operands and to store the results of arithmetic/logic operations. Also used for data transfer between registers and memory.

Register array: Used for general purpose temporary storage or as an operand of an arithmetic/logic instruction.

Index register(s): Contains part of the address from where the data is to be accessed.

Stack pointer: Contains the address of the memory location (stack) which is used to temporarily store processor's internal register contents using software instructions like PUSH, POP or due to interrupts and/or subroutine calls.

Flag register: Composed of bits whose states depend individually on the results of certain arithmetic / logic instructions or configuration of the processor.

ALU: Performs arithmetic and logic operations.

A microcontroller has in addition some RAM, EPROM or EEPROM, and also possibly I/O ports. For testing purposes, the internal memory is treated just like the external memory. So, the basic microprocessor and microcontroller test is almost the same.

We again use the start small approach to microcontroller testing:

1. Start by testing an instruction. Now the test routine for that instruction contains other instructions. So if the test passes, then all the instructions of the routine, as well as instruction under test are working. Add this set of instructions as well as registers involved to the passed item set.

2. Using resources of the passed items set, test the other instructions and registers, one at a time. If the test is successful, add the new item to the passed items set, otherwise notify error and stop.
3. Repeat step 2 until all instructions are covered.

At the start of this test, the passed items set consists of the instructions and registers used and tested in the display and initmicro1 test. We test other instructions and registers based on these initial ones and then keep on adding them to the passed items set until none is left untested. The task of microprocessor test G14 is thus split up into testing individual instructions and registers.

3.7.1 Application to the 68HC11 microcontroller

Now, we apply our testing strategy to the 68HC11 microcontroller which is present in the board we are going to test. We have already tested the instructions I_1 and registers R_1 in the initmicro test. Now with these instructions and registers available, we can test other instructions and registers. We start by testing the remaining registers i.e. Y and SP, and the corresponding load, store and compare instructions. If all of the above tests are successful, then we have the passed instructions and registers as shown in Table 3.6.

Table 3.6 New passed items set.

Registers	A,B,D,X,Y,SP
Instructions	LDAA,STAA,CMPA.,LDAB,STAB,CMPB,LDD,STD,CPD, LDX,STX,CPX,LDY,STY,CPY,LDS,STS,BNE,JMP,CLC,BCS,SEC,ADCB, INX,DEX,ORAA,TAP, TPA,RTS

We have already covered the compare instructions CMPA, CMPB, CPD, CPX, CPY. There is one more compare instruction CBA, which compares the accumulators B and A. So, we test this instruction next to complete the compare instructions testing. The compare instruction involves subtraction of data contained in accumulators A and B. For this and all other instructions

involving two operands, our approach is to exercise all possible combinations on a bit by bit basis. For each bit position, the two operands should contain

- 1 and 1
- 0 and 0
- 1 and 0
- 0 and 1

For example for CBA instruction we first put FF in both accumulators and execute CBA. This means subtracting 1 from 1 on each bit position. Then, we put 00 in both and again execute CBA. This means subtracting 0 from 0 on each position. After that we put 55 in A and AA in B. This means subtracting 0 from 1 at all even bit positions and 1 from 0 at all odd bit positions. Finally, we put AA in A and 55 in B. This means subtracting 1 from 0 at all odd bit positions and 0 from 1 at all even bit positions. Thus all combinations for each bit are tried as shown in Figure 3.5.

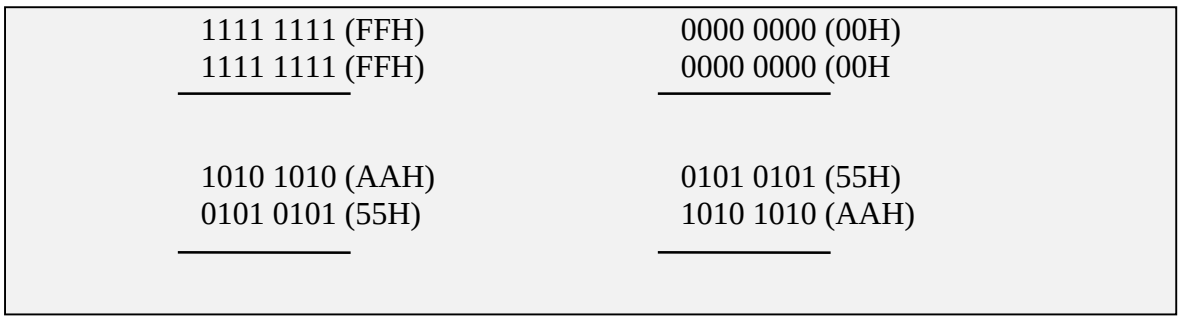

Figure 3.5 Bit combinations used to test arithmetic and logic instructions

So in four combination of operands, we test the instruction. A complete testing would involve trying all possible combinations of the two operands. Hence if 'n' is the number of bits per operand, a complete testing would require $2^n \times 2^n = 2^{2n}$ combinations of operands. For 8-bit operands, we require 65536 combinations. Clearly if we go for complete testing and apply 2^{2n} combinations for each instruction, that would take a very long time. With our approach, we can test the instructions in a reasonable amount of time and yet get adequate though less than perfect fault coverage.

Next, we check for the increment instructions. These are INCA, INCB, INX, INY and INC. Now, increment instructions are one operand instructions. To test them, we select the operands so that we get a 0 to 1 and a 1 to 0 transition for each bit.

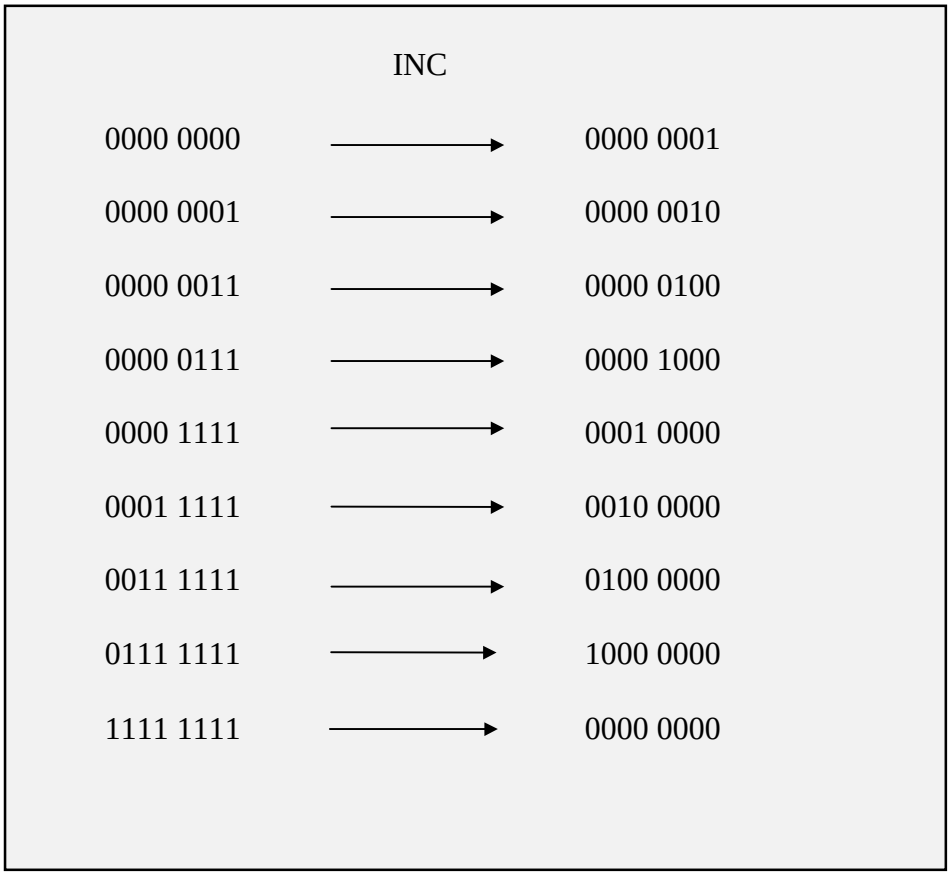

Figure 3.6 Patterns used for INC testing

With these operands, we can detect all stuck-at and transition faults. We then test the decrement instructions in a similar manner, as shown in Figure 3.7. At this stage, our passed items set contains all load, store, increment, decrement, JSR, JMP, BNE and RTS instructions along with the registers A,B,X,Y,SP. Since the accumulators A and B are used most extensively in any program, a thorough test for them seems justified. So, next we test the accumulators A and B. We apply all possible values to each accumulator and after every write, we read both of them, so that we are also assured of their uniqueness. The flow chart for this routine is given in Figure 3.8:

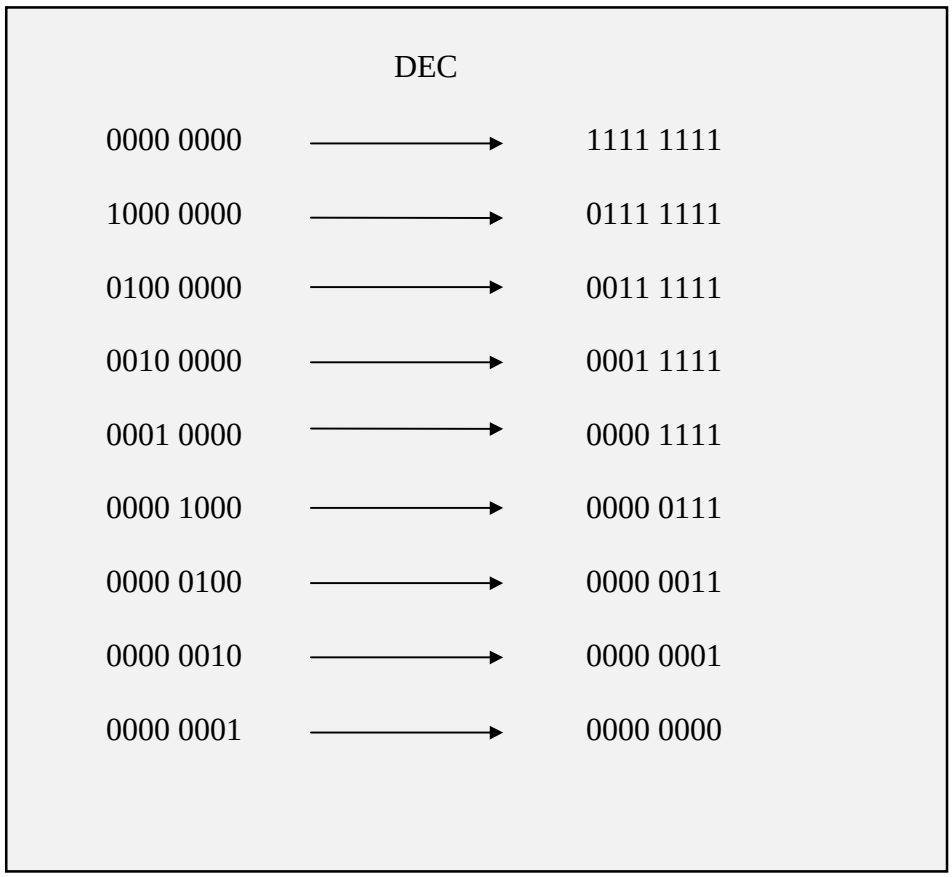

Figure 3.7 Patterns used for DEC testing

Next, we also want to make sure that the data paths between the two accumulators exist. So, we test the instructions TAB and TBA which transfer data between A and B. This testing is again exhaustive i.e. done for all possible values of data.

To complete the accumulator specific instructions, we test the CLRA and CLRB instructions next, which clear the accumulators A and B.

We consider the Branch instructions next. These instructions include BCC, BGT, BHI, BHS, BNE, BPL, BVC, BGE, BCS, BEQ, BLE, BLO, BLS, BMI, BVS and BRN. All these instructions check certain values of flags in order to make decisions about branching. So, we first apply known values to flags and then see whether the branching actions are as expected or not.

To put values in the Condition code register (CCR) which contains the flag bits, we can only use the TAP instruction. Note that the TAP instruction has not been tested as yet, so this instruction is tested itself as well. At this stage, we also test the TPA instruction which transfers the data back to the accumulator from CCR. However, if both the instructions TAP and TPA fail, we will not be able to detect the fault here. The reason is that we first execute TAP which transfers the data from the A to CCR. Then, we are loading this data back to A. It may be possible that both of the instructions failed and we are just reading the original data in A. If we try to put some intermediate value in A through some instruction between the TAP and TPA instructions, this instruction would itself change the flags. So this cannot be done. We will detect the multiple fault in TAP and TPA indirectly with the fault indication in the branch instructions that are to be tested immediately after TAP/TPA test.

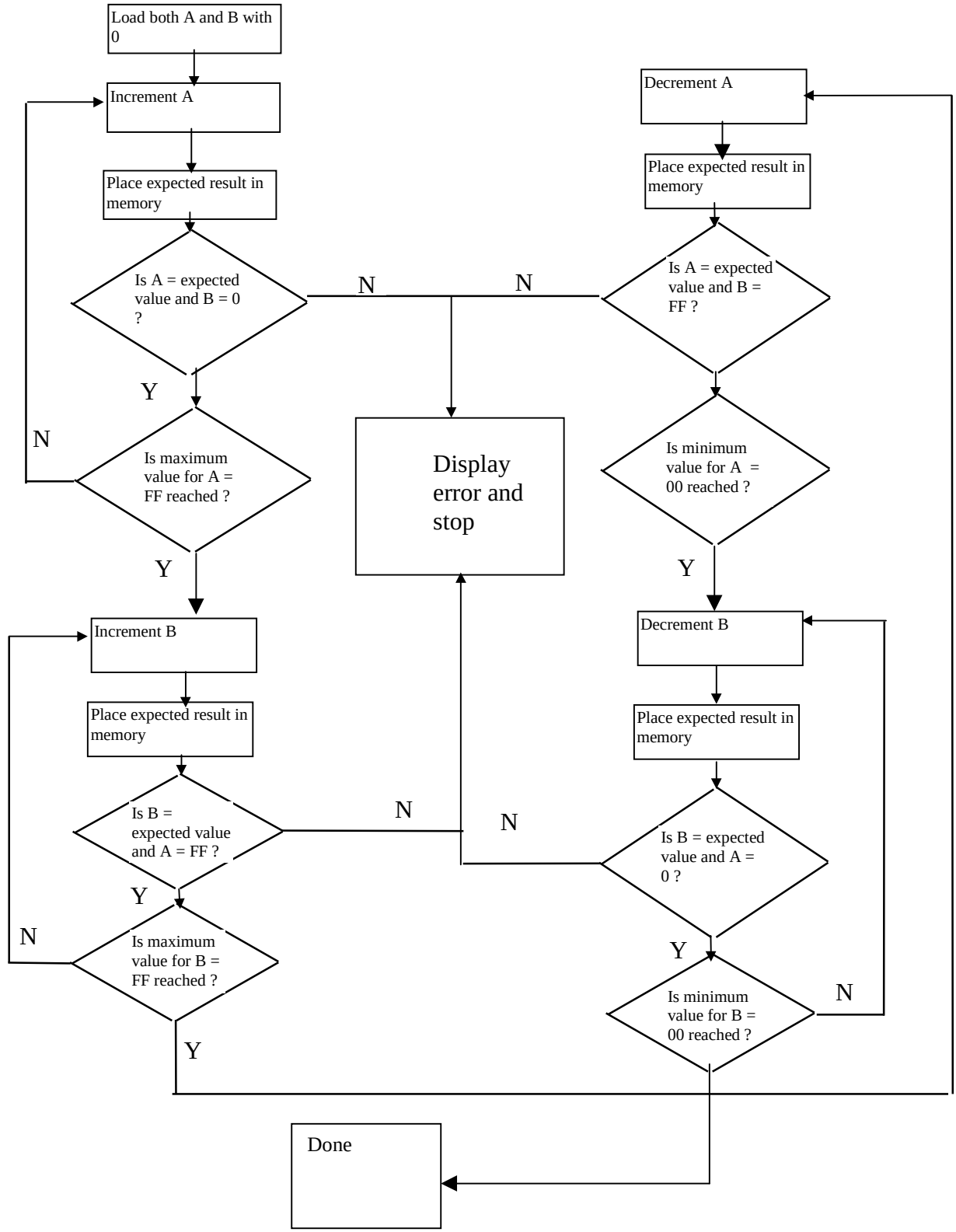

Figure 3.8 Flow chart for testing the accumulators

We then continue with the flag related instructions by testing instructions CLC and SEC. Other flag related instructions i.e., SEI, CLI, CLV, SEV can be tested more conveniently after we test AND and OR instructions.

At this point, our passed items set is as shown in Table 3.7.

Table 3.7 Current passed items set

Registers	A,B,D,X,Y,SP
Instructions	LOADs, STOREs, CoMPares, TAB,TBA,TAP,TPA INCrements, DECrements, Branch instructions, JMP, JSR, RTS,STOP, CLC,SEC,ADCB,ORAA

Making use of these resources, we can now test the ALU related instructions. We test all the add instructions ADDA, ADDB, ADDD, ABA, ABX, ABY. The strategy is again the same as for two operand instructions discussed earlier. In the same manner, we test the subtract instructions SUBA, SUBB, SUBD, SBA.

The add with carry and subtract with borrow instructions operate on the two operands and the carry. We exercised all 8 combination of bits for the least significant bits of two operands and carry. For other bits of the operands, the testing was the same as that for the ADD and SUB instructions. In this manner, tests for ADCA, SBCA and SBCB instructions were carried out.

Continuing with the ALU instructions, we test the MUL instruction next. MUL multiplies the 8-bit contents of accumulator A with those of accumulator B. The result is placed in register D. Again we use FF & FF, 00 & 00, 55 & AA, and finally AA & 55, because with these operand pairs, all the possible bitwise combinations of 0's and 1's are applied. This is again not a thorough test for a multiplier but simple and yet applies all possible combinations on a bit wise basis.

The division instruction IDIV divides the 16 bit contents of register D by 16 bit contents of register X. The quotient is put in D while remainder is put in X. These were tested with the same two operand pairs.

The DAA instruction corrects the result of a packed BCD addition. It applies a correction to convert the result which is a hex number to packed BCD format. A correction is applied whenever:

- the hex number contains any of the hex digits A to F, because the BCD digits are only 0 to 9
- the result of the addition is greater than or equal to 100 H. In this case, the carry is set to indicate that the result is greater than 100 H and the accumulator A contains the two least significant BCD digits.

For testing DAA instruction, the operands are so chosen that we exercise all 4 combinations for each pair of bit positions *and* generate a result which needs DAA correction. So, we choose the pairs of operands shown in Table 3.8.

Table 3.8 Operands for testing DAA instruction

Operands	1001 1001 (99 H) 1001 1001 (99 H)	0110 0110 (66 H) 0110 0110 (66 H)	1001 1001 (99 H) 0110 0110 (66 H)	0110 0110 (66 H) 1001 1001 (99 H)
ADD	0011 0010 (32 H)	1100 1100 (CC H)	1111 1111 (FF H)	1111 1111 (FF H)
DAA	1001 1000 (98 BCD)	0011 0010 (32 BCD)	0101 0101 (55 BCD)	0101 0101 (55 BCD)

Next, we test the following instructions, using the familiar patterns of FF, 00, 55 and AA.

ANDA, ANDB,
ORAB,
EORA, EORB,
COMA, COMB, COM,
NEGA, NEGB, NEG

Next, we turn to shift and rotate instructions. To test the arithmetic shift left (ASLA, ASLB, ASLD, ASL) instructions, we put a pattern of alternating 1's and 0's i.e. AA H, shift left 8 times

and check for expected results after each shift. To test the arithmetic shift right (ASRA, ASRB, and ASR) instructions, we apply the same procedure. The point to note this time is that the MSB of the operand would remain unchanged no matter how many shifts we apply. In comparison, when we test for the logical shift right instructions (LSRA, LSRB, LSRD, and LSR), the MSB would be shifted to the right *and* filled with zero from the left.

All the rotate instructions (RORA, RORB, ROR, ROLA, ROLB, ROL) were tested in the same manner.

Earlier, we did not test the flag related instructions CLI, SEI, CLV and SEV because AND and OR instructions were not yet available. Now, we test them because the AND and OR instructions have been tested.

The Bit test instructions (BITA, BITB) test whether a certain bit is 0 or 1. A mask byte is supplied to them. This mask byte is ANDed with the operand to see whether the bits under test of the operand are 0 or 1. To test the instruction, we put a known pattern in the accumulator, apply BITA to find the value of a bit and then check for the expected result.

The Branch Clear Instruction (BRCLR) branches if the bits of the operand it is checking are zero. BRCLR works by ANDing the operand with the mask byte so that at the bit positions where mask byte is 1, if the operand bits are zero, then the result is zero, otherwise it is non-zero. To test the instruction, we select 55 H as the operand and give the mask byte as AA H. In that case, the result would be zero and hence branch should occur if the instruction is working.

Branch Set Instruction (BRSET) works in a similar manner, except that it branches when any of the bits we are checking is 1. It is checked by selecting both the operand and mask byte as 55 H. In that case, a non-zero result would be obtained by ANDing the two and hence branch should occur if the instruction is working.

The Clear Bit(s) instruction (BCLR) clear all those bits of the operand which are 0 in the mask byte. It works by ANDing the operand with the mask byte. To test the instruction, we select FF as the operand and first select the mask byte as 55. This would clear all the odd numbered bits, so that the result would be AA. We compare the result with AA to see whether it is the same or not. Then, we reinitialize the operand to FF and select mask byte as AA. This would clear all even numbered bits, so that the result would be 55. We compare the result with 55 in this case.

The Set Bit(s) instruction (BSET) works in the similar manner as BCLR. The mask byte here is ORed with the operand so that all those bits of operand become 1 which are 1 in the mask byte. To test the instruction, we select 00 as the operand and first select the mask byte as 55. This would set all even numbered bits, so that the result would be 55. We compare the result with 55 to see whether it is the same or not. Then, we reinitialize the operand to 00 and select mask byte as AA. This would set all odd numbered bits, so that the result would be AA. We then compare the result with AA.

The TSTA, TSTB and TST instructions subtract 0 from the operand so that the result is the same as the operand *but* the flags change depending on its value. If the operand was 0, then Z flag will be set; if it was representing a negative number (1 in MSB), then N flag will be set. We test these instructions for both these cases.

Next, we test the stack instructions. The PUSH instructions put the value of the operand on the stack whereas PULL instructions retrieve values from the stack. If we are testing a PUSH instruction, we will push the operand on the stack and then read it to confirm the operation. But after an operand is pushed onto the stack, we can only read it using the PULL instructions. This means that we have to test the PUSH and PULL instructions together. Also, if we pushed the operand and then pulled it back to the same location, there may be a chance that both the instructions are not working and we are actually reading the original value. To avoid that, after pushing, we load the operand location with a base value which is different from the value pushed, and then pull it. As far as the value of the operand is concerned, the use of two different

operands 00 and FF i.e. all 0's and all 1's seems adequate. In this manner, we check the PUSH and PULL instructions in pairs: PSHA & PULA, PSHB & PULB, PSHX & PULX, PSHY & PULY.

Now, with all PUSH and PULL instructions tested, we can test other stack instructions. We start with INS and DES, which increment and decrement the stack pointer, respectively. The important thing here is that the return address of the subroutines are present in the stack before we are going to test these instructions, and during testing, the execution of INS and DES will alter this value, and hence we will lose program control. So before testing these instructions, the current value of stack pointer should be first stored in memory and then reloaded after this test is complete. The INS and DES instructions are tested in exactly the same manner as the other increment and decrement instructions tested previously.

Next, we test the instructions involving data transfer between stack pointer and index registers X and Y. Again, since the stack pointer value is to be changed during the test, so we first store the current value in memory and retrieve later. We test these instructions in pairs i.e. TXS & TSX and TYS & TSY. We first transfer a pattern from the register to stack pointer (TXS,TYS), change the content of register to base value and then transfer the pattern back from the stack pointer (TSX,TSY). We use the patterns 00 and FF here.

The instructions XGDX and XGDY interchange the contents of registers D and X, and D and Y respectively. We use the following pattern pairs and after the execution, they should lie in the registers opposite to where they were actually present: FFFF and 0000, 0000 and FFFF, AAAA and 5555, 5555 and AAAA.

The program counter PC is not explicitly checked in this procedure. If it is faulty, then we will lose program control and hence the system will either time out or hang up. This will detect the PC fault indirectly.

The complete hierarchy for testing the instructions and registers of the microcontroller is given in Table 3.9.

Table 3.9 Complete hierarchy for microcontroller tests.

Registers	**Instructions**
A	LDAA, STAA, CMPA (Load, store, compare)
	BNE (Branch if not equal)
B	LDAB, STAB, CMPB (Load, store, compare)
D	LDD, STD, CPD (Load, store, compare)
X	LDX, STX, CPX (Load, store, compare)
Y	LDY, STY, CPY (Load, store, compare)
SP	LDS, STS (Load, store)
	INCA, INCB, INX, INY, INC (Increment)
	DECA, DECB, DEX, DEY, DEC (Decrement)
A, B (thorough)	
	TAB, TBA (Data transfer between accumulators)
	CLRA, CLRB (Clear accumulators)
	TAP, TPA (Data transfer between accumulator and CCR)
	BCC, BGT, BHI, BHS, BPL, BVC, BGE, BCS, BEQ, BLE, BLO, BLS, BMI, BVS, BRN (Branches)
	CLC, SEC (Carry flag clear and set)
	ADDA, ADDB, ADDD, ABA, ABX, ABY (Adds)
	SUBA,SUBB,SUBD,SBA (Subtracts)
	ADCA, ADCB (Add with carry)
	SBCA, SBCB (Subtract with borrow)
	MUL (multiply)
	IDIV (Division)
	DAA (Decimal adjust)

	ANDA, ANDB (Ands)
	ORAA, ORAB (Ors)
	EORA, EORB (Exclusive-Ors)
	COMA, COMB, COM (1's complement)
	NEGA, NEGB, NEG (2's complement)
	ASLA, ASLB, ASLD, ASL (Arithmetic shift left)
	ASRA, ASRB, ASR (Arithmetic shift right)
	LSRA, LSRB, LSRD, LSR (Logical shift right)
	RORA, RORB, ROR (Rotate right)
	ROLA, ROLB, ROL (Rotate left)
	CLI, SEI (Interrupt flag clear and set)
	CLV, SEV (Overflow flag clear and set)
	BITA, BITB (Bit test)
	BRCLR, BRSET (Branch on clear and set)
	BCLR, BSET (Bits clear and set)
	TSTA, TSTB, TST (Test for zero and minus)
	PSHA , PULA, PSHB , PULB, PSHX , PULX, PSHY , PULY (Push and pull instructions for registers)
	INS, DES (stack increment and decrement)
	TXS, TSX, TYS, TSY (transfer data between stack and index registers)
	XGDX, XGDY (Exchange contents of registers D & X and D & Y)

The passed items and suspect sets at the end of this test are:

$P = \{D, L, DC, I_1, R_1, EEPROM, RAM, MICROCONTROLLER\}$
$SS = \{MICROCONTROLLER\}$

3.8 Self-test incorporated in the system model

The self-test routines and the application program are then combined to form one program. The self-test routines are executed before the application program. When the system powers up, it first checks the display, then the external EEPROM, then it checks the internal RAM and then the microcontroller. If all the tests are passed, it then proceeds to execute the application program. The self-test pass or fail messages for all components are displayed in the same output window which is used for temperature result display. The output in case of a fault free system is given in Figure 3.9. This figure does not contain the display pass message because that routine was written later and applied to the hardware implementation of the system.

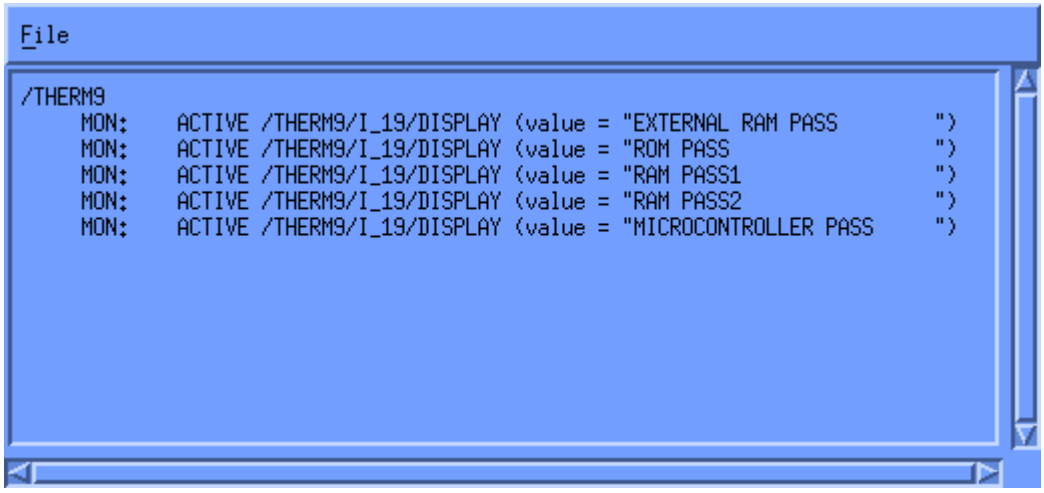

Figure 3.9 VHDL debugger/simulator output for the fault free system

The purpose and usage of external RAM is explained in Chapter 4. As explained in section 3.6.5, the internal RAM is divided into two portions for the purpose of testing. The RAM PASS1 and RAM PASS2 messages are for these two parts.

3.9 Debugging and Simulation

After the test routines for different portions of the circuit were written, the object code of the whole program (test program + application program) was first loaded in the Sim11 Motorola 68HC11 assembly language simulator/debugger.

The following points were noted while using the simulator:

- This simulator is basically used for validating the assembly code. It cannot detect any error involving interaction of microcontroller with external components like ROM, PPI etc., functioning of external components, or hardware design error.
- For the ROM test, the system has to add all the locations of EEPROM to get the checksum. But the program in EEPROM does not cover the entire memory space. The simulator treats the unused portion as uninitialized data and cannot do any operation involving uninitialized data. So, after loading the machine code of the program into the simulator, we have to put zeros on all the unused locations using the simulator commands before starting the simulation.
- The simulator cannot take any input from an external component. So, the simulation stops when a line is encountered which takes input from some external source. With reference to our system, we cannot supply it with the ADC output during the simulation. To avoid that, we modify the code for the purpose of simulation, so that the input is obtained directly from the program rather than the ADC output.

After the code is validated by the Sim11 simulator, the .s19 file is then used to create the Memory image files (.MIF) which are to be loaded in the VHDL models of components. Now, the whole system is modeled in VHDL, so that when we simulate it in the VHDL simulator, it can detect all possible errors that can occur in a physical system. This is in contrast to the assembly language Sim11 simulator which can just check the validity of the program code. However, the program code is very difficult to debug with the VHDL simulator since we have no access to the internal registers and operations of the microcontroller. We can just see what is appearing on the external pins of the microcontroller. This makes the task of debugging much more difficult and sometimes impossible. So, the approach used was first to verify the code using

the Sim11 simulator, load this code in the system memory and then simulate the whole system in the VHDL simulator.

If using this approach, we still get an error during VHDL simulation, then the main causes are an interconnection fault, a hardware design fault or improper configuration of the external components.