東 海 大 學 應 用 數 學 研 究 所

碩 士 論 文

應用在NAND快閃記憶體的LFSR測試

Testing of the NAND Flash Memory with Linear Feedback Shift Registers.

研 究 生：蘇 明 財

指 導 教 授：黃 皇 男 博 士

中 華 民 國 九 十 六 年 七 月

中文摘要

本篇論文的主要目的在探討消費性電子產品NAND 快閃記憶體的簡化功能性錯誤的測試，採用線性同餘法和線性回饋移位暫存法產生頁位址的亂數來檢測NAND 快閃記憶體的簡化功能性錯誤。本研究所發展軟體配合現有硬體檢測設施可節省NAND 快閃記憶體的簡化功能性錯誤的測試成本。測試的結果顯示線性回饋移位暫存器測試法比其他的方法有更好的功效。

Abstract

The main purpose of this thesis is to research the tests of the reduced functional faults of the consuming electronic product: NAND flash memory. The linear congruential method and linear feedback shift registers (LFSR) are used to product random numbers that locate the page address for testing the reduced functional faults of the NAND flash memory. The purpose of this study is to develop a software testing environment to drive a low cost hardware device in identifying the page with reduced functional faults for NAND flash memory. The test result shows that LFSR has better performance than the others.

Contents

1 Introduction

The motivation and organization of this thesis is described as follows.

1.1 Motivation

The current personal computers usually involve of three subsystems: the CPU, input/output devices, and the memory subsystem. The memory subsystem usually contains two types of memories (refer to [1]):

1. Cache and main memory, which hold programs and data during processing. They have a short access time and a high cost per bit. Main memory are made up of memory chips, each of which can hold up to several million bits of data.

2. Secondary memory, such as disk or tape, which has the advantage of a low cost per bit at the expense of a long access time. It is used for permanent and archival storage of programs and data.

Consider the fact that memory chips form a momentous part of the total number of chips in a digital system, memory may be a prime contributor to the overall failure rate. Therefore effective tests which supply for fault detection and localization are important.

Historically the number of bits per (DRAM) chip has quadrupled about every 3.1 (or π) years. The exponential increase in the number of bits per chip has caused the price to decrease exponentially as well. Because of economic reasons the test cost per chip (which is directly related to the test time) can't be allowed to increase significantly. At the same time the number of bits per chip is increasing exponentially and the sensitivity to faults increases and the faults become more complex. We take NAND flash memory (refer to [2] for its data sheet) for the example, tests for detecting faults which are dependent on the neighboring cells are more complex and require a longer execution time than those which don't.

Researches on flash memory tests have been conducted for many years [1, 3, 4] and the results show that the occurrence of functional faults depends on the manufacturing process and many other factors. In order to understand the relationship between the test patterns and the types of flash memory functional faults, one needs to setup an testing environment to carry the identification work. Although many test systems for flash memory has been designed, e.g. Agilent 4082F [5], these systems are too expansive, functionality is quite complicated and test methods are classified. Some advanced testing equipment, such as Advantest's T5593 or T5588 [6] each costs 4.5 million USD.

In this study, we try to build up a software testing environment with the simple purpose to perform random testing for functional faults. This software system will drive the hardware devices (shown in Figures 4.2 and 4.3), which is designed and provided by Dr. Chi-Chung Ai, to perform the NAND flash memory testing.

1.2 Organization

Since we are concerned with testing NAND flash memory faults via pseduo-random pattern generators in this thesis. A software testing environment using the C++ program under Windows XP is developed to carry the memory functional tests. The organization of this thesis is as following: Basic properties and functional faults of NAND flash memories are presented in chapter 2. In chapter 3, we review the methods to generate random numbers and pseduo-random pattern. In chapter 4, the testing plant is described and the associated test algorithm and test results are also presented for illustrative purpose. Conclusion is given in chapter 5.

2 Memory Models and Faults

In this chapter, the model of memory and associated memory functional faults are presented.

2.1 NAND Flash Memory

In this chapter we will discuss the feature of the NAND flash memory and associated memory faults (refer to [2] for its data sheet). The feature summaries of the NAND flash memory are follows:

1. High density
   (a) Up to 1Gbit memory array
   (b) Up to 32Mbit spare area
   (c) Cost effective solutions for mass storage applications
2. NAND Interface
   (a) x8 or x16 bus width
   (b) Multiplexed address/data
3. Page size
   (a) x8 device: (512 + 16 spare) Bytes
   (b) x16 device: (256 + 8 spare) Words

4. Block size
   (a) x8 device: (16K + 512 spare) Bytes
   (b) x16 device: (8K + 256 spare) Words
5. Page read and program
   (a) Random access: 12 μs ($3V$)/15 us ($1.8V$)
   (b) Sequential access: 50 ns
   (c) Page program time: 200 μs

Note: 1 second = 10^3 ms = 10^6 μ s = 10^9 ns = 10^{12} ps

The memory array is made up of NAND structures where 16 cells are connected in series. The memory array is organized in blocks where each block contains 32 pages. The array is split into two areas, the main area and the spare area. The main area of the array is used to store data whereas the spare area is typically used to store Error correction Codes, software flags or Bad Block identification. In x8 devices the pages are split into a main area with two half pages of 256 Bytes each and a spare area of 16 Bytes. In the x16 devices the pages are split into a 256 Word main area and an 8 Word spare area. Refer to Figure 2.1., Memory Array Organization.

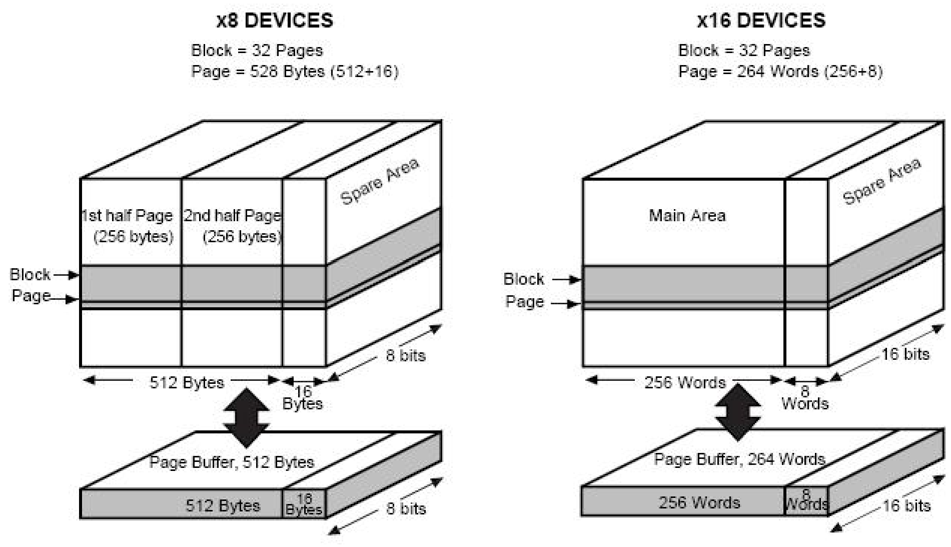


Figure 2.1: Memory Array Organization [2]

The early tests before 1980 were not based on a fault model (such as the stuck-at, coupling or pattern sensitive fault models); consequently, their quality, in terms of fault converge, can't be proved. Their main reason for existence was that they were satisfactory to the extent that they provided for adequate fault coverage for the test time they required. Some of these tests require test times in the order of $O(n)$ or even $O(n^2)$, where n is the number of bits in the chip. Newer generations of megabit chips would require test times which are not economically feasible. Table 2.1 lists the required test time in seconds, assuming a memory cycle time of 100 ns, as a function of the algorithm complexity and the memory size.

Table 2.1: Test time as a function of memory size [1]

n	Algorithm complexity	
	$O(n)$	$O(n^2)$
1k	0.0001	0.105
4k	0.0004	1.7
16k	0.0016	27
64k	0.0066	410
256k	0.0026	1.8 h
1M	0.105	30.6 h
4M	0.42	500 h

2.2 Reduced functional faults

A system failure occurs or is present when the service of the system differs from the specified service, or the service that should have been offered. In other words: the systems fails to do what it has to do. A failure is caused by an error. There is an error in the system (the system is in an erroneous state) when its state differs from the state in which it should be, in order to deliver the specified service. An error is caused by a fault. A fault is present in the system when there is a physical difference between the 'good' or 'correct' system and the current system.

Most tests for faults in memory chip are based on a reduced functional model. Figure 2.2 shows a general model of a RAM chip and Table 2.2 lists some functional faults that can occur in RAM chip (the list is not complete).

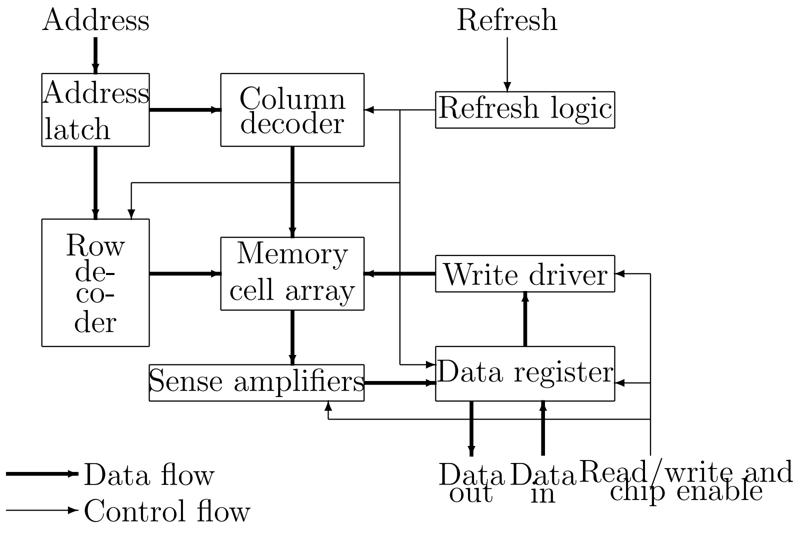


Figure 2.2: Functional model of a RAM chip [1]

Table 2.2: List of functional faults [1]

	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 line
k	Short between address lines
l	Open decoder
m	Wrong access
n	Multiple access
o	Cell can be set to 0 but no to 1 (or vice-versa)
p	Pattern sensitive interaction between cells

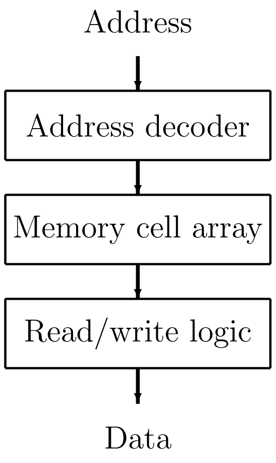


Figure 2.3: Reduced functional model [1]

During chip testing one isn't interested in locating a fault because a chip can't be repaired. One is only interested in detecting a fault. For this reason the model of Figure 2.2 can be simplified without loss of information. For functional testing a model is used that contains only three blocks: the address decoder, the memory cell array and the read/write logic (Thatte, 1977) and (Nair, 1978); see Figure 2.3. Using the reduced functional model of Figure 2.3, the list of faults given in Table 2.2 can be mapped onto the faults of Table 2.3. These faults can be classified as follows:

1. Faults in which a single cell is involved.

   These are the stuck-at and transition faults.

2. Faults in which two cells are involved.

   These are the coupling faults.

3. Fault involving *k* cells.

   The *k* cells are allowed to be located anywhere in memory.

Table 2.3: Relationship between functional and reduced functional faults [1]

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 line
AF	k	Short between address lines
AF	l	Open decoder
AF	m	Wrong access
AF	n	Multiple access
TF	o	Cell can be set to 0 but no to 1 (or vice-versa)
NPSF[1]	p	Pattern sensitive interaction between cells

SAF: The stuck-at fault (SAF) can be defined as follows: The logic value of a stuck-at (SA) cell or line is always 0 (a SA0 fault) or 1 (a SA1 fault); it is always in state 0 or in state 1 and can't be changed to the opposite state.

[1]NPSF will be introduced into the reference [1] and we will not discuss.

TF: The transition fault (TF) is a special case of SAF and defined as follow: A cell which fails to undergo a $0 \rightarrow 1$ transition when it is written is said to contain an up transition fault; similarly, a down transition fault is the impossibility of making a $1 \rightarrow 0$ transition.

CF: In the coupling fault (CF), two cells are involved per fault. These two faults can have the following relationships:

CFin: An $0 \rightarrow 1$ (or $1 \rightarrow 0$) transition in one cell inverts the contents of a second cell.

CFid: An $0 \rightarrow 1$ (or $1 \rightarrow 0$) transition in one cell forces the contents of a second cell to a certain value, 0 or 1.

AF: Address decoder faults (AFs) concern faults in the address decoder.

A march test consists of a finite sequence of march elements (Suk, 1981). A march element is a finite sequence of operation applied to every cell in memory before proceeding to the next cell. Here we use (w0) to denote the action of erase the cell and then write 0 into the cell. And (w1) denotes the action of writing 1 into the cell. Similarly, (r0) and (r1) denote the read operation from the cell with expected value 0 and 1, separately. Also $\Uparrow$ or $\Downarrow$ denotes the cell to be operated with address increasing or decreasing accordingly. And $\Updownarrow$ corresponds to the operation by increasing the address of the cell first down to the end, and then it decreases the address of the cell again till the beginning of the memory.

The march test, March A, that detects the above SAF, TF, and CF can be expressed as:

$$\{\underbrace{\Updownarrow (\text{w0})}_{M0}; \underbrace{\Uparrow (\text{r0, w1})}_{M1}; \underbrace{\Downarrow (\text{r1, w0, r0})}_{M2}\}$$

The scheme of March A test is expressed as follow:

```
$M0$ : {March element $\Updownarrow$ (w0)}
  for all cells do
      $A[cell] = 0$;
$M1$ : {March element $\Uparrow$ (r0, w1, r1, w0, r0, w1)}
  for cell = 0 to n-1
  begin
      read A[cell]; (Expected value=0)
      write 1 to A[cell];
  end;
$M2$ : {March element $\Downarrow$ (r1, w0, w1, w0)}
  for cell = n-1 to 0
  begin
```

```
        read A[$cell$]; (Expected value=1)
        write 0 to A[$cell$];
        read A[$cell$]; (Expected value=0)
    end;
```

The march test, March B, detects the above faults and can be expressed as:

$$\{\underbrace{\Updownarrow(\text{w0})}_{M0};\underbrace{\Uparrow(\text{r0, w1, r1, w0, r0, w1})}_{M1};\underbrace{\Uparrow(\text{r1, w0, w1})}_{M2};$$

$$\underbrace{\Downarrow(\text{r1, w0, w1, w0})}_{M3};\underbrace{\Uparrow(\text{r0, w1, w0})}_{M4}\}$$

The scheme of March B test is expressed as follow:

```
$M0$ : {March element $\Updownarrow$ (w0)}
  for all $cells$ do
      $A[cell] = 0$;
$M1$ : {March element $\Uparrow$ (r0, w1, r1, w0, r0, w1)}
  for $cell$ = 0 to n-1
  begin
      read A[$cell$]; (Expected value=0)
      write 1 to A[$cell$];
      read A[$cell$]; (Expected value=1)
```

```
        write 0 to A[$cell$];
        read A[$cell$]; (Expected value=0)
        write 1 to A[$cell$];
    end;
$M2$ : {March element $\Uparrow$ (r1, w0, w1)}
    for $cell = 0$ to n-1
    begin
        read A[$cell$]; (Expected value=1)
        write 0 to A[$cell$];
        write 1 to A[$cell$];
    end;
$M3$ : {March element $\Downarrow$ (r1, w0, w1, w0)}
    for $cell =$ n-1 to 0
    begin
        read A[$cell$]; (Expected value=1)
        write 0 to A[$cell$];
        write 1 to A[$cell$];
        write 0 to A[$cell$];
    end;
$M4$ : {March element $\Downarrow$ (r0, w1, w0)}
```

```
for $cell$ = n-1 to 0
begin
    read A[$cell$]; (Expected value=0)
    write 1 to A[$cell$];
    write 0 to A[$cell$];
end;
```

We defined the simple basic march test, March Basic Test, that is expressed as:

$$\{\Updownarrow (\text{w0}); \Uparrow (\text{w1, r1, w0, r0})\}$$

3 Random Number Generators

In this chapter, we review methods to generate the random number and psuedo-random pattern.

3.1 The Properties of Random Number Generators

It is difficult to find “good” random number generators(RNGs). The experts use the following criteria to measure a random number generator. See Pierre L’Ecuyer’s Random Number Generation [7] for a more thorough discussion of quality criteria.

1. **Good Theoretical Basis.**

   The RNG should be based on an algorithm with provably good statistical properties. But statistical tests are very limited in scope due to computational limitations. We can’t search all numbers produced by an RNG with a period of 2^{131} for statistical anomalies. It’s similar to cryptography. Just because no one has cracked our cipher, doesn’t mean we have a good cipher. Statistical tests can prove that an RNG is bad; they can’t prove that an RNG is good.

2. **Long Period.**

   The RNG should have a ”long” period. Every RNG based on a finite state eventually repeats the numbers it generates. The period should be long enough to ensure that the RNG does not cycle in practice.

3. **"Pass" Empirical Statistical Tests.**

   Most statistical tests compute a p-value which should be uniform over [0,1). Poor RNGs will fail many tests with p-values at or very close to 0 or 1.

4. **Efficient.**

   Many researchers place a strong emphasis on efficiency. Obviously, an RNG which is very cpu-intensive will be impractical. The emphasis we place on this criterion depends a great deal on we application for the RNG. If we require generating many numbers very quickly, then we'll place a high value on performance. Likewise if you need to implement the RNG in hardware and have very limited resources. But, for many applications, efficiency will be low on the list of priorities.

5. **Repeatable.**

   For simulation, we would like an RNG that is repeatable. In other words, given the same starting state, it should generate the same numbers. For cryptographers this is less important; they need RNGs that are unpredictable.

6. **Portable.**

   In order for an RNG to be generally useful, it should be portable. That means that it should be relatively easy to implement on a wide range of hardware, operating systems, and programming environments. RNGs which use only 32-bit integers are more portable than RNGs that use 64-bit integers.

3.2 Linear Congruential Method

System-supplied rand()s are almost always linear congruential generators, which generate a sequence of integer $I_1, \ I_2, \ \cdots$, each between 0 and $m-1$ by the recurrence relation (refer [8])

$$I_{j+1} = aI_j + c \pmod m \tag{3.1}$$

Here m is called the modulus, and a and c are positive integers called the multiplier and the increment respectively. The recurrence (3.1) will eventually repeat itself, with a period that is obviously no greater than m. If $m, \ a$, and c are properly chosen, then the period will be of maximal length, i.e., of length m. In this case, all possible integers between 0 and $m-1$ occur at some point, so any initial "seed" choice of I_0 is as good as any other: the sequence just takes off from that point.

3.2.1 Rand 0 Test

There is good evidence, both theoretical and empirical, that the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod{m} \tag{3.2}$$

can be as good as any of the more general linear congruential generators that have $c \neq 0$ (3.1) – if the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "Minimal Standard" (refer [8]) generator based on the choices

$$a = 7^5 = 16807 \quad m = 2^{31} - 1 = 2147483647 \tag{3.3}$$

The generator isn't "perfect", but only that it is a good minimal standard against which other generators should be judged.

Schrage's algorithm is based on an approximate factorization of m,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a], \; r = m \pmod{m} \tag{3.4}$$

with square brackets denoting integer part. If r is small, specifically $r < q$, and $0 < z < m-1$, it can be shown that both a(z mod q)and $r[z/q]$ lie in the range $0, \cdots, m-1$, and that

$$az \pmod{m} = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is } \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases} \tag{3.5}$$

The application of Schrage's algorithm to the constants (3.3) using the values $q = 127773$ and $r = 2836$ is called **rand 0** test (refer [8]). The period of rand 0 test is $2^{31} - 2 \approx 2.1 \times 10^9$.

3.2.2 Rand 1 Test

A characteristic of generators of the rand 0 test is that the value 0 must never be allowed as the initial seed (since it perpetuates itself) and it never occurs for any nonzero initial seed. The follow routine, rand1, uses the “Minimal Standard” for its random value, but it shuffles the output to remove low-order serial correlation. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. the shuffling algorithm is due to Bays and Durham as described in Knuth, and is illustrated in Figure 3.1 (refer [8]).

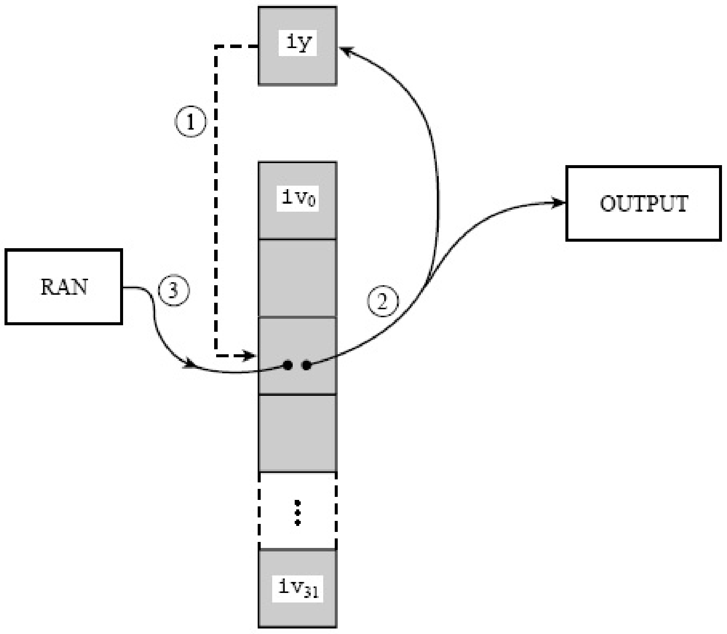


Figure 3.1: Shuffling procedure used in rand 1 test to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in *iy* is used to choose a random element in the array *iv*. That element becomes the output random number, and also is the next *iy*. Its spot in *iv* is refilled from the Minimal Standard routine. [8]

3.2.3 Rand 2 Test

The routine of rand 1 test passes those statistical tests that rand 0 test is known to fail. In fact, we don't know of any statistical test that rand1 fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are need, L'Ecuyer has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of either of them. A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \cdots, m-1$. Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, rand 2 test.

L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For current personal computers, period exhaustion is a practical impossibility.

3.3 Pseudo-random Pattern Generator

3.3.1 Linear Feedback Shift Register

The Pseudo-random pattern generators (PRPGs) are usually implemented using a linear feedback shift register (LFSR, refer [1]). A LFSR consists of bits connected as a shift register, and XOR-gates which allow outputs of certain bits to be fed back to the input bit of the LFSR.

The number of bits of the LFSR and the way the feedback paths are implemented, together with the initial seed (the initial contents of the bits), determine the length of the test sequence before it repeats. This sequence length (the number of different patterns that are generated) can be at most 2^n (for a LFSR with n bits).

The number of feedback paths, together with the way they are connected, can be expressed mathematically in a polynomial, called the characteristic polynomial or generator polynomial, $G(x)$. For the LFSR of Figure 3.2, which shows the implementation of a LFSR with three bits labeled x^0, x^1 and x^2, and three feedback paths (using two XOR gates), the characteristic polynomial is $1 + x + x^2 + x^3$. This is determined as follows.

The square blocks in Figure 3.2 represent the bits. The input of a bit is on the left and the output is on the right. The input of the second bit (x^1) is connectd to the output of the first bit (x^0). The output of x^1 is connected to the input of x^2. The input of x^0 is a signal that is the result of an XOR operation on the output signals of x^0, x^1 and x^2. The value shifted into x^0, during a next clock period at time $t+1$, is:

$$\left[x^0\right]_{t+1} = \left[x^0 + x^1 + x^2\right]_t \tag{3.6}$$

This means that the contents of x^0, after clock-pulse $t+1$, equals the modulo-2 sum of the contents of the bits x^0, x^1 and x^2 after clock-pulse t. Equation (3.6) can be rewritten as:

$$\left[x^0\right]_{t+1} = \left[x^0\right]_t + \left[x^1\right]_t + \left[x^2\right]_t \tag{3.7}$$

From inspecting Figure 3.3, which consists of the LFSR of Figure 3.2 extended with an infinite number of bits, it can be seen that

$$\left[x^0\right]_t = \left[x^1\right]_{t+1}, \quad \left[x^1\right]_t = \left[x^2\right]_{t+1}, \quad \text{and} \quad \left[x^2\right]_t = \left[x^3\right]_{t+1}$$

Equation (3.7) can therefore be written as:

$$\left[x^0\right]_{t+1} = \left[x^1\right]_{t+1} + \left[x^2\right]_{t+1} + \left[x^3\right]_{t+1}$$

or

$$x^0 = x^1 + x^2 + x^3 \tag{3.8}$$

The characteristic polynomial of this LFSR is now defined as:

$$x^0 + x^1 + x^2 + x^3$$

and using the notation $x^0 = 1$ allows the characteristic polynomial to be written as:

$$1 + x + x^2 + x^3 \quad \text{or} \quad x^3 + x^2 + x + 1$$

If the initial contents (also called the seed) of the bits (the state of the LFSR of Figure 3.2) is “000”, the next state of the LFSR (the contents of the bits of the LFSR after the clock-pulse) will also be “000”. So the sequence length (the number of different patterns which can be produced) will be 1 (only “000” is being produced). But when the seed is “100”, the length will be 4, see Table 3.1(a).

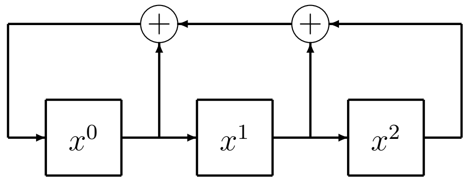


Figure 3.2: LFSR with characteristic polynomial $1 + x + x^2 + x^3$ [1]

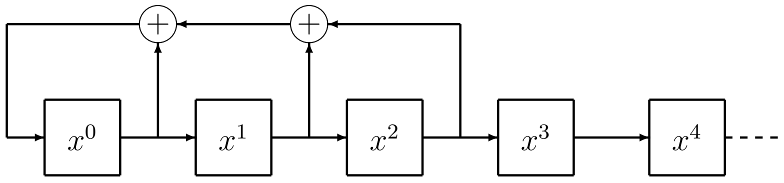


Figure 3.3: Endless LFSR with characteristic polynomial $1 + x + x^2 + x^3$ [1]

Table 3.1: Sequence of patterns generated by LFSRs [1]

(a)LFSR of Figure 3.2				(b)LFSR of Figure 3.3			
				1	0	0	Initial state
				1	1	0	
				1	1	1	
1	0	0	Initial state	0	1	1	
1	1	0		1	0	1	
0	1	1		0	1	0	
0	0	1		0	0	1	
1	0	0	Repetition	1	0	0	Repetition
⋮	⋮	⋮		⋮	⋮	⋮	

If the feedback paths of Figure 3.2 are connected according to Figure 3.3 the sequence length will be 7, see Table 3.1(b). This will be the case for any seed except all 0s. The characteristic polynomial of this LFSR can be determined to be $x^3 + x^2 + x + 1$. A length of $2^n - 1$ of an n-bit LFSR is called the maximum-length sequence.

The question is how to make a maximum-length LFSR? Wang [9, 10] show that for an n-bit LFSR to have the maximum-length $2^n - 1$, the characteristic polynomial should be a primitive polynomial. A primitive polynomial is a polynomial of degree n that divides $x^i + 1$, for $i = 2^n - 1$, but for no integer smaller than i. Table 3.2 lists primitive polynomials for degree 1 through 100. For each degree, a polynomial has been taken with the fewest number of terms. Only the exponents are shown in the table 3.2, for example the entry (18, 5, 2, 1, 0) represents the polynomial $x^{18} + x^5 + x^2 + x^1 + 1$

Table 3.2: List of primitive polynomials [1]

(1, 0)	(26, 6, 2, 1, 0)	(51, 6, 3, 1, 0)	(76, 5, 4, 2, 0)
(2, 1, 0)	(27, 5, 2, 1, 0)	(52, 3, 0)	(77, 6, 5, 2, 0)
(3, 1, 0)	(28, 3, 0)	(53, 6, 2, 1, 0)	(78, 7, 2, 1, 0)
(4, 1, 0)	(29, 2, 0)	(54, 6, 5, 4, 3, 2, 0)	(79, 4, 3, 2, 0)
(5, 2, 0)	(30, 6, 4, 1, 0)	(55, 6, 2, 1, 0)	(80, 7, 5, 3, 2, 1, 0
(6, 1, 0)	(31, 3, 0)	(56, 7, 4, 2, 0)	(81, 4, 0)
(7, 1, 0)	(32, 7, 5, 3, 2, 1, 0)	(57, 5, 3, 2, 0)	(82, 8, 7, 6, 4, 1, 0)
(8, 4, 3, 2, 0)	(33, 6, 4, 1, 0)	(58, 6, 5, 1, 0)	(83, 7, 4, 2, 0)
(9, 4, 0)	(34, 7, 6, 5, 2, 1, 0)	(59, 6, 5, 4, 3, 1, 0)	(84, 8, 7, 5, 3, 1, 0)
(10, 3, 0)	(35, 2, 0)	(60, 1, 0)	(85, 8, 2, 1, 0)
(11, 2, 0)	(36, 6, 5, 4, 2, 1, 0)	(61, 5, 2, 1, 0)	(86, 6, 5, 2, 0)
(12, 6, 4, 1, 0)	(37, 5, 4, 3, 2, 1, 0)	(62, 6, 5, 3, 0)	(87, 7, 5, 1, 0)
(13, 4, 3, 1, 0)	(38, 6, 5, 1, 0)	(63, 1, 0)	(88, 8, 5, 4, 3, 1, 0)
(14, 5, 3, 1, 0)	(39, 4, 0)	(64, 4, 3, 1, 0)	(89, 6, 5, 3, 0)
(15, 1, 0)	(40, 5, 4, 3, 0)	(65, 4, 3, 1, 0)	(90, 5, 3, 2, 0
(16, 5, 3, 2, 0)	(41, 3, 0)	(66, 8, 6, 5, 3, 2, 0)	(91, 7, 6, 5, 3, 2, 0)
(17, 3, 0)	(42, 5, 4, 3, 2, 1, 0)	(67, 5, 2, 1, 0)	(92, 6, 5, 2, 0)
(18, 5, 2, 1, 0)	(43, 6, 4, 3, 0)	(68, 7, 5, 1, 0)	(93, 2, 0)
(19, 5, 2, 1, 0)	(44, 6, 5, 2, 0)	(69, 6, 5, 2, 0)	(94, 6, 5, 1, 0)
(20, 3, 0)	(45, 4, 3, 1, 0)	(70, 5, 3, 1, 0)	(95, 6, 5, 4, 2, 1, 0)
(21, 2, 0)	(46, 8, 5, 3, 2, 1, 0)	(71, 5, 3, 1, 0)	(96, 7, 6, 4, 3, 2, 0)
(22, 1, 0)	(47, 5, 0)	(73, 6, 4, 3, 2, 1, 0)	(97, 6, 0)
(23, 5, 0)	(48, 7, 5, 4, 2, 1, 0)	(73, 4, 3, 2, 0)	(98, 7, 4, 3, 2, 1, 0)
(24, 4, 3, 1, 0)	(49, 6, 5, 4, 0)	(74, 7, 4, 3, 0)	(99, 7, 5, 4, 0)
(25, 3, 0)	(50, 4, 3, 2, 0)	(75, 6, 3, 1, 0)	(100, 8, 7, 2, 0)

3.3.2 The Fibonacci Implementation

The Fibonacci implementation is the easiest to implement LF-SRs in hardware (refer [8]), requiring only a single shift register n bits long and a few xor gates, the operation denoted in C by "$\wedge$". For the primitive polynomial $1 + x^1 + x^2 + x^5 + x^{18}$, the recurrence formula is

$$a_0 = a_{18} \wedge a_5 \wedge a_2 \wedge a_1 \tag{3.9}$$

The terms that are $\wedge$'d together can be though of as "taps" on the shift register, $\wedge$'d into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

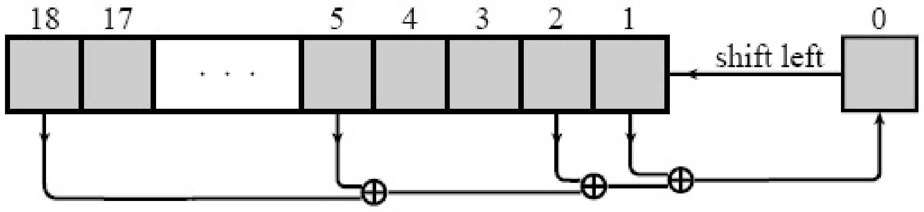


Figure 3.4: The Fibonacci Implementation. The contents of selected taps are combined by exclusive-or (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. [8]

3.3.3 The Galois Implementation

The Galois implementation is less suited to direct hardware implementation, but is beautifully suited to C (refer [8]). It modifies more than one bit among the saved n bits as each new bit is generated. It generates the maximal length sequence, but not in the same order as the Fibonacci implementation. The prescription for the primitive polynomial is:

$$\begin{aligned} a_0 &= a_1 \\ a_5 &= a_5 \wedge a_0 \\ a_2 &= a_2 \wedge a_0 \\ a_1 &= a_1 \wedge a_0 \end{aligned} \tag{3.10}$$

In general there will be an exclusive-or for each nonzero term in the primitive polynomial except 0 and n.

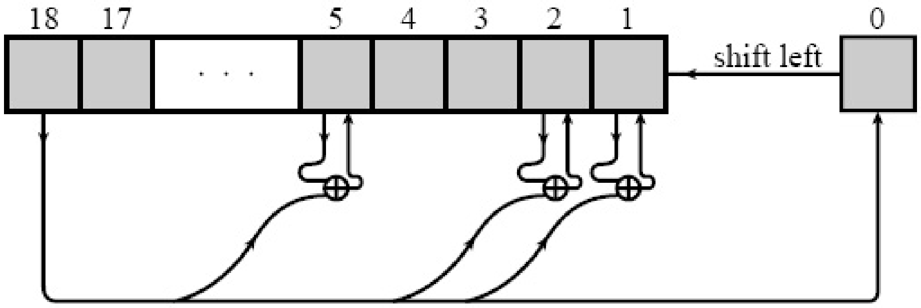


Figure 3.5: The Galois Implementation. Selected bits are modified by exclusive-or with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software. [8]

4 Memory Testing and Results

4.1 Memory Testing Plant

The NAND flash memory testing plant consists of the PC (Pentium 4 CPU 3.2 GHz and 3.21 Ghz with 1GB ram), the high driving capability 96-CH digital I/O card (PCI-7396), and the NAND flash memory testing card. Figure 4.1 shows the picture of the hardware of the testing plant. Figure 4.2 shows the PCI-7396 digital I/O card. The NAND flash memory testing card is shown in Figure 4.3.

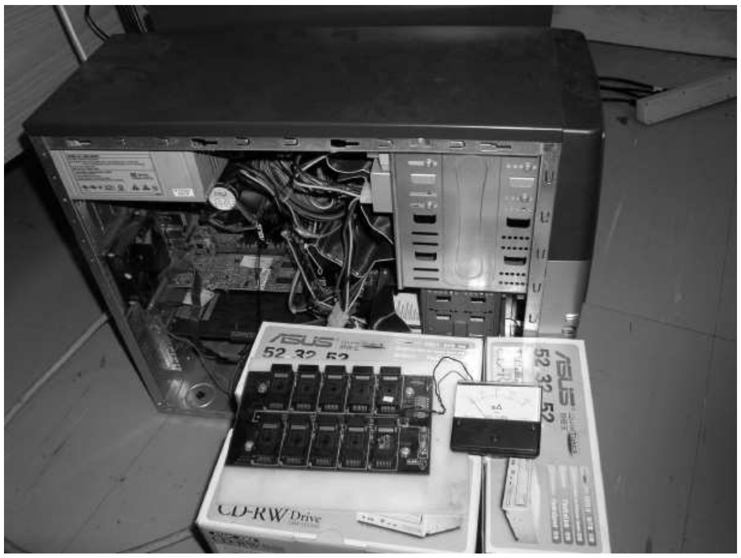


Figure 4.1: NAND Flash Testing Plant

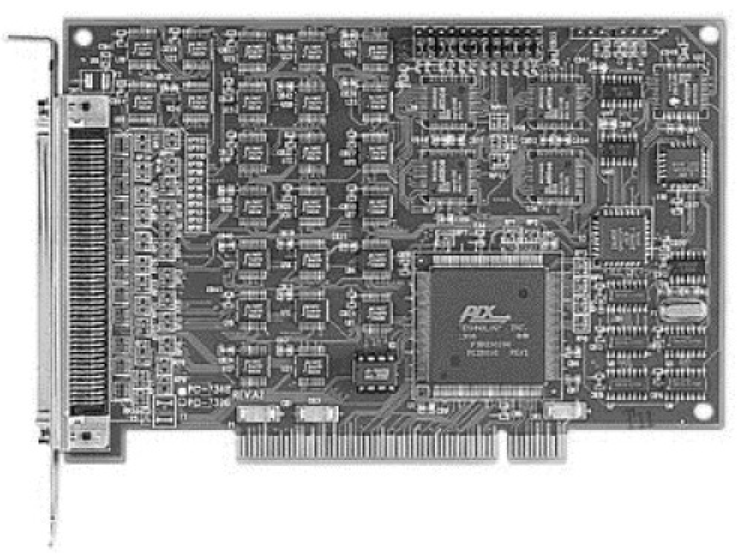


Figure 4.2: High Driving Capability 96-CH Digital I/O Cards (PCI-7396)

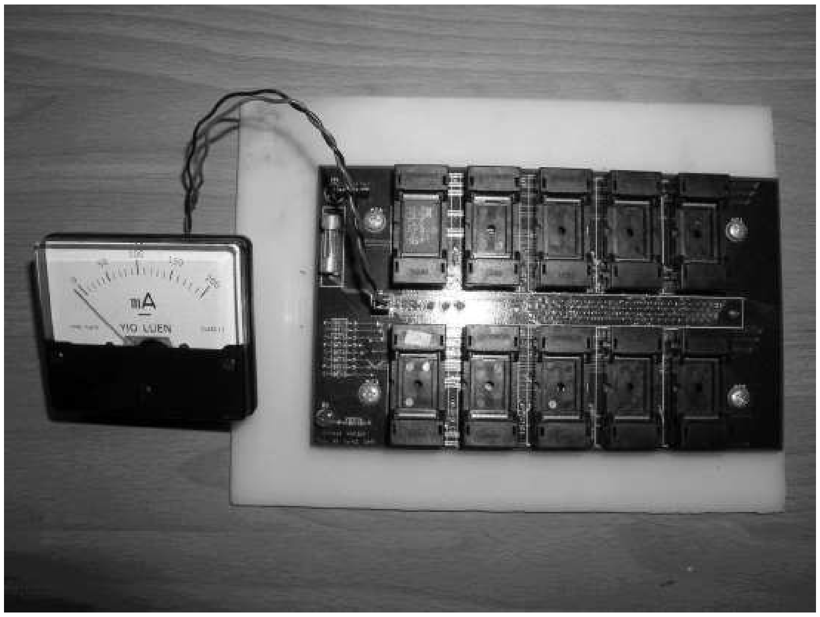
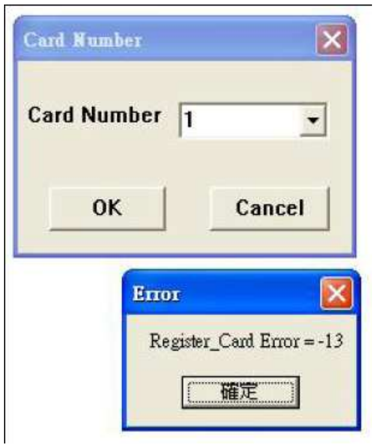


Figure 4.3: NAND Flash Testing Card

4.2 Memory Testing Algorithm

Step 1: Set up the testing plant, which is shown in Figure 4.1. We must supply 3.3V voltage from PC to the NAND flash memory, NAND flash memory testing card, and the voltmeter. When we set up the testing plant, we can see the shine of the red LED light in the NAND flash memory testing card.

Step 2: Execute the testing program, Util7396.exe. Then it will check the digital I/O card PCI-7396 in PC. If there isn't PCI-7396 card in PC, it will tells us the error, see Figure 4.4.



Register_Card Error = -13

Figure 4.4: Executing the testing program.

Step 3: Pass the "Check ID" button of the NAND memory testing program to check the validity. If it shows "7920", then we can know that the testing plant is usable, see Figure 4.5. If not, then we must find the trouble.

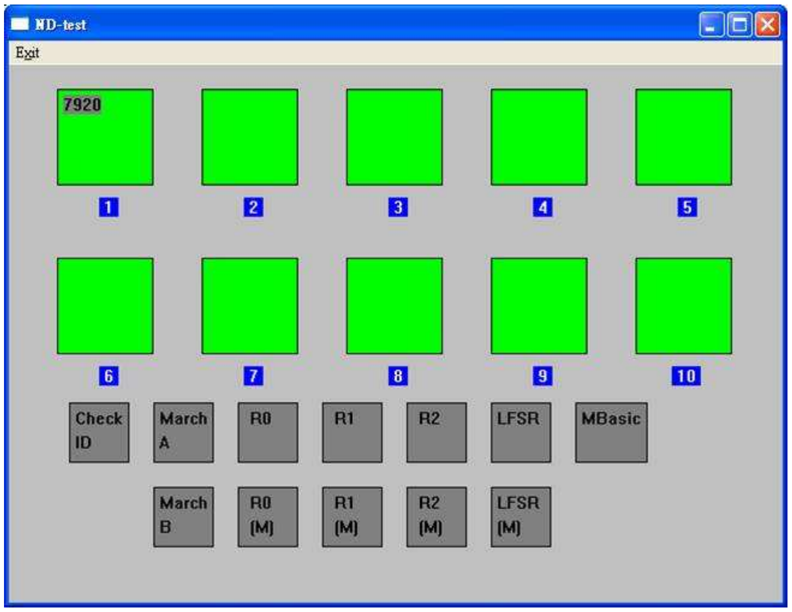


Figure 4.5: Check the id of the NAND Flash Memory.

Step 4: Pass the chosen button of the memory testing. When the testing finish, there is the word "Finish" showing in the block of Socket 1, see Figure 4.6

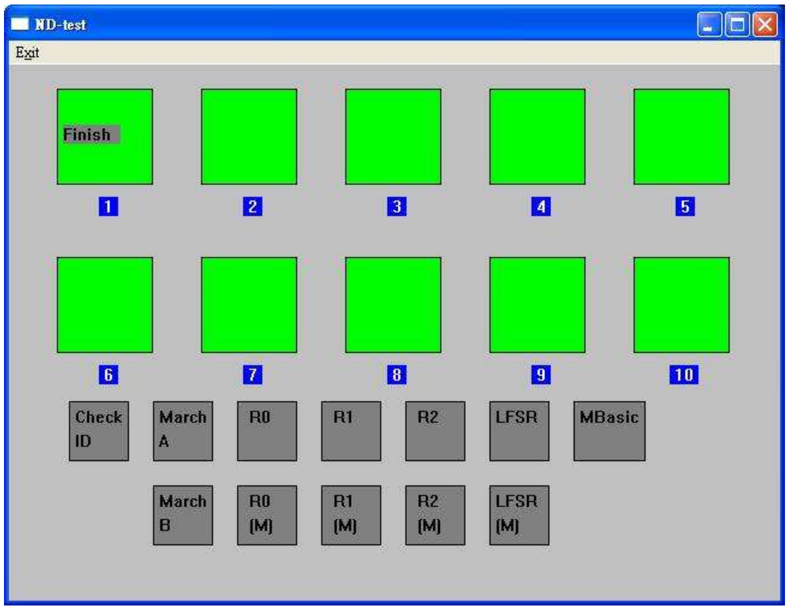
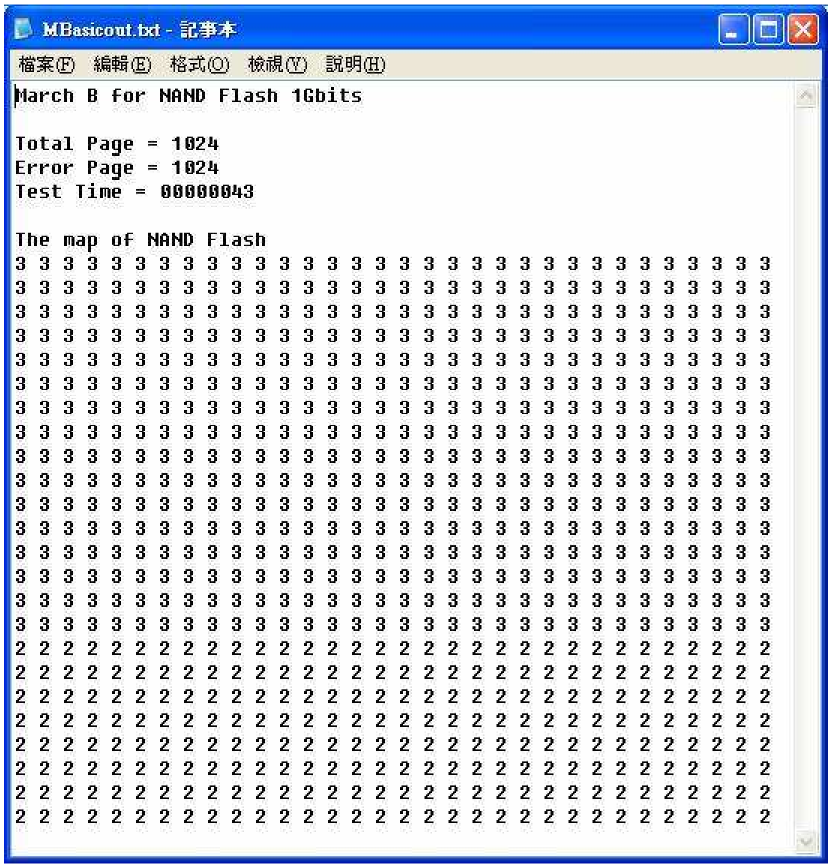


Figure 4.6: The testing finish.

Step 5: When the testings finish, we can open the reports, see Figure 4.7.



```
March B for NAND Flash 1Gbits

Total Page = 1024
Error Page = 1024
Test Time = 00000043

The map of NAND Flash
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 4.7: The testing report.

4.3 Results and Discussion

The NAND flash memories manufactured by STMicroelectronics with the part No.NAND01GW3A0AN6 are used in our test. The size of the memory is 1 Gbit. The sample output of testing the memory chip device code No.79 and manufacturer code No.20 is shown in Figure 4.7. We use the sequential March B test as our baseline. Although the sequential March B test for the whole flash memory has been done but it takes over 133 hours, we only present the test result for 1024 pages. The number of error pages and execution time for performing test patterns (w0, w1, r0) and (w0, w1, r1, w0, r0) with random (Rand0, Rand1, Rand2) and LFSR(psuedo-random) addressing are listed in Table 4.1.

The result in Table 4.1 shows that LFSR has much better than Rand0 and Rand1 which is consistent with [1]. We note that there are some hardware faults which are detected during testing and at this time the NAND flash memory must be reset in order to access data for next testing.

Table 4.1: The number of error pages and execution time for March B, R0, R1, R2, and LFSR. Test Page: 0x01FE00 - 0x0201FF.

Test Methods		Error Pages	Testing Time
March A $\{\Updownarrow(w0); \Uparrow(r0,w1); \Downarrow(r1,w0)\}$		512	53
March B		512	43 (300)
March Basic (w0, w1, r1, w0, r0)		512	43
(w0, w1, r1)	Rand 0	312	18
	Rand 1	328	19
	Rand 2	320	21
	LFSR (Fibonacci)	512	31
(w0, w1, r1, w0, r0)	Rand 0	325	52
	Rand 1	342	49
	Rand 2	345	52
	LFSR (Fibonacci)	512	52

5 Conclusions and Future Study

In this thesis, a software testing environment using C++ for NAND flash memory fault detection has be implemented. Various random scheme and LFSR (pesudo-random pattern) have be included in the system. Sequential March B test is adopted as the based line. March A tests are performed with Rand0, Rand1, Rand2, and LFSR memory page addressing. The test result shows that LFSR test has better performance than the others.

Based on present thesis, the following directions are suitable for future study:

1. Based on March A and March B test, develop more March test pattern to identify each type of reduced functional faults.

2. Enhance GUI for the software testing environment.

3. For different manufacturing process of flash memory, develop a most suitable psuedo-random pattern to wholly address the memory page.

4. Based on the present software code and hardware device, develop a stand alone system which can be used to test the NAND flash memory without connecting to PC.

References

[1] Goor, A.J. van de (1991) *Testing Semiconductor Memories: Theory and Practice*, John Wiley & Sons Ltd. Batffins Lane. Chichester West Sussex PO19 1UD, England.

[2] http://www.st.com/stonline/products/literature/ds/10058/nand512w3a.pdf

[3] Bez, R., Camerlenghi, E., Modelli, A., and Visconti A. (2003) "Introduction to Flash Memory," In *Proc. of The IEEE*, Vol. 91, No. 4, pp. 489-502.

[4] Roth, D.R., Kinnison, J.D., Carkhuff, B.G., Lander, J.R., Bognaski, G.S., Chao, K., and Swift, G.M. (2000) "SEU and TID testing of the Samsung 128 Mbit and the Toshiba 256 Mbit flash memory," 2000 IEEE Radiation Effects Data Workshop, Reno, NV, USA, pp. 96-99, July 24-28.

[5] Agilent 4082F Flash Memory Cell Parametric Test System, Agilent Technologies, 2007

[6] http://www.advantest.co.jp/products/ate/t5593/en-index.shtml or
http://www.advantest.co.jp/products/ate/t5383/en-index.shtml

[7] L'Ecuyer, Pierre(2004) "Random Number Generation," *Handbook of Computational Statistics*, J.E. Gentle, W. Haerdle, and Y. Mori, eds., Springer-Verlag, pp. 35-70.

[8] William H. Press (1989) *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, New York.

[9] Wang, L.T. and McCluskey, E.J. (1986) Circuite for Pseudo-Exhaustive Pattern Generation. In *Proc. IEEE Int. Test Conference,* pp. 25-37.

[10] Wang, L.T. and McCluskey, E.J. (1986a) A Hybrid Design of Maximum Length Sequence Generators. In *Proc. IEEE Int. Test Conference,* pp. 38-47.

[11] Kernighan, Brian W. (1988) *The C Programing Language*, Prentice Hall, Inc., New Jersey.