國 立 交 通 大 學

電 機 學 院 IC 設計產 業 研 發 碩 士 班

碩 士 論 文

# 有效成本控制的三明治型乒乓記憶體之設計與分析

# Study on cost-efficient Sandwich Ping-Pong Memory

研 究 生：魏文俊

指導教授：董蘭榮 教授

中華民國 九十七 年 十二 月

# 有效成本控制的三明治型乒乓記憶體之設計與分析

# Study on cost-efficient Sandwich Ping-Pong Memory

研 究 生：魏 文 俊　　Student：Wen-Chun Wei

指導教授：董 蘭 榮　　Advisor：Lan-Rong Dung

國 立 交 通 大 學

電機學院 IC 設計產業研發碩士班

碩 士 論 文

A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Industrial Technology R & D Master Program on
IC Design

December 2008

Hsinchu, Taiwan, Republic of China

中華民國 九十七 年 十二 月

# 有效成本控制的三明治型乒乓記憶體之設計與分析

研究生：魏 文 俊　　　　指導教授：董 蘭 榮　博士

國立交通大學電機學院產業研發碩士班

## 摘要

本研究旨在探討記憶體(緩衝器)的成本與效能上之取捨，我們提出一個適用於快速資料傳輸上的三明治型乒乓記憶體(緩衝器)　，並驗證其適切性。相對於現行之乒乓記憶體的而言，此記憶體使用較少的面積，並且在操作頻率上有更多選擇及彈性的空間。使用者可調整三明治型乒乓記憶體的容量大小，進而決定操作頻率為何，藉以有效控制成本。此外，為了測試此記憶體，我們也根據現行常用的測試演算法，發展出專屬於三明治型乒乓記憶體之測試演算法，並經由國家晶片系統設計中心，成功地完成下線並製作晶片，更進一步利用此測試演算法，偵測出一般常見的缺陷模型，結果在缺陷包容度上亦達到 100%。在晶片控制單元的設計上，我們發現能使晶片控制單元所佔的面積能最小之方法。運用此方法，當減少 500 個記憶體單元面積的同時，亦能使額外的晶片控制單元小於 500 個閘數。

本研究尚依據研究結果，針對三明治型乒乓記憶體(緩衝器)實務方面之應用，以及未來之研究提出建議。

關鍵詞：乒乓記憶體、轉置緩衝器、記憶體測試演算法、閒置的記憶體

# Study on cost-efficient Sandwich Ping-Pong Memory

Student: Wen-Chun Wei    Advisor: Dr. Lan-Rong Dung

Industrial Technology R & D Master Program of

Electrical and Computer Engineering College

National Chiao-Tung University

## Abstract

This thesis is about memory (buffer) and we present a Sandwich Ping Pong Memory. The area in the Sandwich Ping Pong Memory is much less than in a Ping Pong Memory. Besides, it is more flexible on the operation frequency compared with a Ping Pong Memory. Data are written into the Sandwich Ping Pong Memory row by row and read from it column by column simultaneously. Based on March C- algorithm, we also developed the test algorithm for the Sandwich Ping Pong Memory and named it the modified March C- algorithm. It can detect the stuck-at fault, transition fault, address fault, and coupling fault. We also successfully taped out a 64-byte Ping Pong Memory in process 0.35 $\mu m$ 2p4m in National Chip Implementation Center (CIC). Finally, we do the verification and testing. As a result, the fault coverage is at 100% of each fault. The chip is 1310 x 1100 micro meters squared. In order to design the control unit, the area overhead is under five hundred gate counts at the range of Common Bar is under 512 unit memory cells.

Keywords：Ping Pong buffer, Transpose buffer, March C- algorithm, Idled memory

# Acknowledgement

# 致謝

回首在交通大學的研究生涯，使我培養了獨立思考及冷靜面對並解決問題的能力，讓我在各方面都有所成長且有更深切的體會，在此謹向我的指導教授董蘭榮老師致上最崇高的敬意，由衷的感謝老師的悉心指導，在研究過程及課業方面，甚至是在台上演說，均給予多方的指導與建議。

此外，也要感謝中央大學劉建男教授、交通大學黃俊達教授以及范倫達教授，在論文上的指導與匡正，使本論文更臻完善。

同時更要感謝實驗室一起相處、努力的同學們，登琦、嘉洋、宇佑、建勛、展嘉、志恆以及學弟建樺，謝謝你們熱心的協助與指導，使得我在研究過程中遇到的困境能夠迎刃而解。也因為有你們的陪伴，在我的課閒生活上也增添了許多歡樂，有個多采多姿的研究生活。

當然，還要感謝我家人，親愛的外公、外婆、父親、母親，感謝他們的養育之恩，在我求學生涯中給我最大的鼓勵與支持，使我得以在精神與生活上無後顧之憂，順利完成學業。適逢母親五十大壽，在此，祝她笑口常開、青春永駐。

最後，我要感謝毓涵，你的鼓勵一直是我研究的動力所在，謝謝你。

感謝你們，僅以本論文獻給摯愛的大家。

文俊　于新竹交大

2008年12月

# Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

## 1.1 Research Objective

The Ping Pong Memory (buffer) is widely used in variety of applications and is studied [1], [2], [3], and [4]. Because it spends less time on waiting the data compared with the single port memory. Its operation frequency is higher than the single port memory. Additionally, the area of the Ping Pong Memory is much less than the dual port memory. Therefore, when it comes to the memory, Ping Pong Memory must be the first one we think about. It can be operated at the high frequency and the cost of area is pretty less.

The address sequence of writing and reading is different. Figure 1.1 shows data are written into Ping/Pong Memory row by row and read from Pong/Ping Memory column by column simultaneously. We found that there were some memory cells in the idled condition when the Ping Pong Memory was on operation. As a result, we combined these idled memory cells to make a Common Bar and developed the Sandwich Ping Pong Memory.

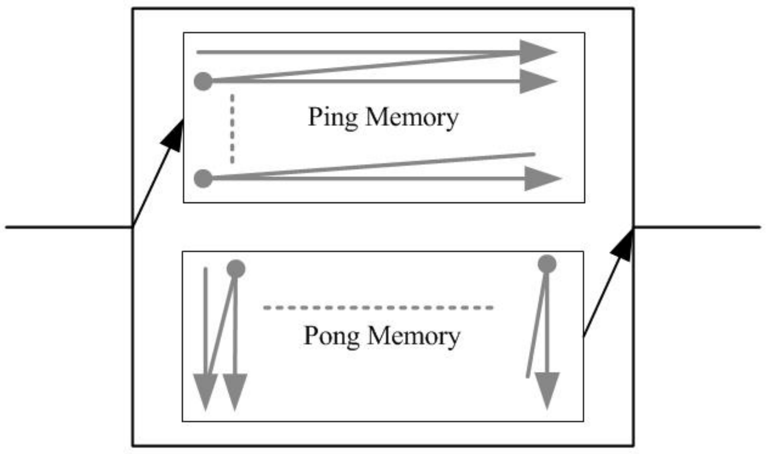


**Figure 1.1** Ping-Pong Memory is on write/read operation

## 1.2 The Proposed Memory

We proposed a Sandwich Ping Pong Memory. The architecture of the Sandwich Ping Pong Memory is illustrated in figure 1.2. Common Bar is between the Ping and Pong Memory. It replaces the idled memories when Ping Pong Memory is on operation. Therefore, there is less area in the Sandwich Ping Pong Memory than in Ping Pong Memory.

It is more flexible on operation frequency because the operation frequency is affected by the size of the Sandwich Ping Pong Memory, especially the size of the Common Bar. It can not be operated on fix frequency. Once we decided the size of the Common Bar, we decided the frequency of the Sandwich Ping Pong Memory.

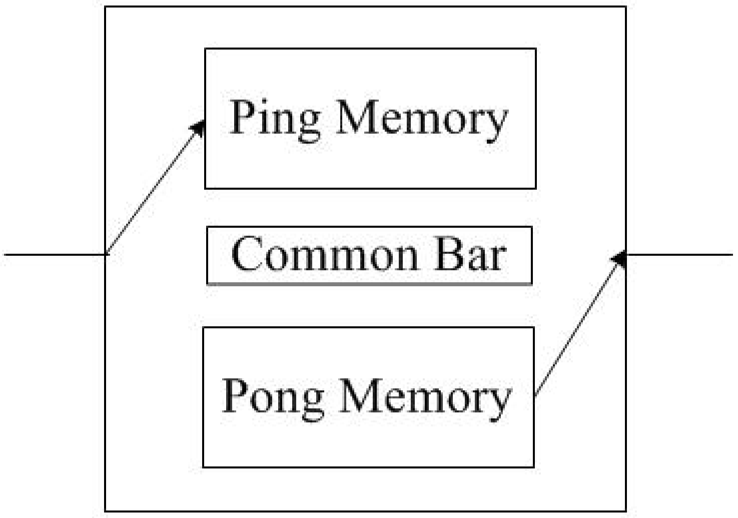


**Figure 1.2** Architecture of Sandwich Ping Pong Memory

## 1.3 Organization

The rest of this thesis is organized as followed. In Chapter 2, we find the applications of the Ping Pong Memory. In Chapter 3, we present our proposed the Sandwich Ping Pong Memory, its read and write operation, the timing analysis, and its control unit circuit. In order to test it, we develop its own test algorithm which is based on a previous test algorithm in Chapter 4. We taped out our design through the National Chip Implementation Center successfully. In Chapter 5, we introduce the chip implementation. Finally, we give a few conclusions in Chapter 6.

# Chapter 2 Background

In this chapter, we introduce a simple, and perhaps obvious, technique that eliminates the need for the two memory operations during each time slot. We call the technique "**ping-pong buffering**." A ping-pong buffer, shown in figure 2.1, uses two conventional single-ported memories in parallel, so that while "Ping" memory is written "Pong" memory can be read, and while "Ping" memory is read "Pong" memory can be written. The two memories are arranged so that from the outside, they appear to be a single buffer. Ping-pong buffering is widely used including image compression and network communication.

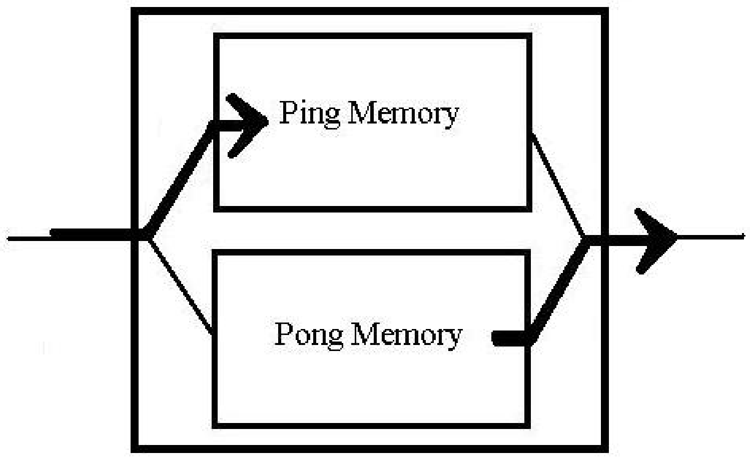


**Figure 2.1** A Ping-Pong Buffer

The easiest way to explain how a ping-pong buffer works is to take a real world example. It is a nice sunny day and you have decided to get the paddling pool out, only you can't find your garden hose. You'll have to fill the pool with buckets. So you fill one bucket (or buffer) from the tap, turn the tap off, walk over to the pool, pour the water in, walk back to the tap to repeat the exercise. This is analogous to single buffering. The tap has to be turned off while you "process" the bucket of water.

Now consider how you would do it if you had two buckets. You would fill the first bucket and then swap the second in under the running tap. You then have the length of time it takes for the second bucket to fill in order to empty the first into the paddling pool. When you return you can simply swap the buckets so that the first is now filling again, during which time you can empty the second into the pool. This can be repeated until the pool is full. It is clear to see that this technique will fill the pool far faster as there is much less time spent waiting, doing nothing, while buckets fill. This is analogous to double buffering. The tap can be on all the time and does not have to wait while the processing is done.

## 2.1 Ping Pong Buffers for Image Compression

Discrete Cosine Transform (DCT) is a mathematical tool that has a lot of electronics applications, from audio filters to video compression hardware. DCT transforms the information from the time or space domains to the frequency domain, such that other tools and transmission media can be run or used more efficiently to reach application goals: compact representation, fast transmission, memory savings, and so on.

In image compression, ping-pong buffering (or double buffering) is a widely used technique especially for the 2-dimensional discrete cosine transform. The discrete cosine transforms (DCT) are a family of similar transforms closely related to the discrete sine transform and the discrete Fourier transform. The DCT-II is the most commonly used form and plays an important role in coding signals and images [5], e.g. in the widely used standard JPEG compression. The discrete cosine transform was first introduced by Ahmed, Natarajan, and Rao [6], [7], and [8]. Later Wang and Hunt [9] introduced the complete set of variants.

### 2.1.1 One-dimensional Discrete Cosine Transform

Formally, the discrete cosine transform is a linear, invertible function $F$: $\mathbf{R}^N$ -> $\mathbf{R}^N$ (where $\mathbf{R}$

denotes the set of real numbers), or equivalently an invertible $N \times N$ square matrix. There are several variants of the DCT with slightly modified definitions. The $N$ real numbers $x_0$, $x_{N-1}$ are transformed into the $N$ real numbers $X_0$, $X_{N-1}$ according to one of the formulas:

**DCT-I**

$$X_k = \frac{1}{2}(x_0 + (-1)^k x_{N-1}) + \sum_{n=1}^{N-2} x_n \cos\left(\frac{\pi}{N-1} nk\right) \qquad k = 0, \dots, N-1. \qquad (2\text{-}1)$$

Some authors further multiply the $x_0$ and $x_{N-1}$ terms by $\sqrt{2}$, and correspondingly multiply the $X_0$ and $X_{N-1}$ terms by $1/\sqrt{2}$. This makes the DCT-I matrix orthogonal, if one further multiplies by an overall scale factor of $\sqrt{2/(N-1)}$, but breaks the direct correspondence with a real-even DFT.

The DCT-I is exactly equivalent (up to an overall scale factor of 2), to a DFT of $2N - 2$ real numbers with even symmetry. For example, a DCT-I of $N$=5 real numbers *abcde* is exactly equivalent to a DFT of eight real numbers *abcdedcb* (even symmetry), divided by two. (In contrast, DCT types II-IV involve a half-sample shift in the equivalent DFT.)

Note, however, that the DCT-I is not defined for $N$ less than 2. (All other DCT types are defined for any positive $N$.)

Thus, the DCT-I corresponds to the boundary conditions: $x_n$ is even around $n$=0 and even around $n$=$N$-1; similarly for $X_k$.

**DCT-II**

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad k = 0, \dots, N-1. \qquad (2\text{-}2)$$

The DCT-II is probably the most commonly used form, and is often simply referred to as "the

DCT". This transform is exactly equivalent (up to an overall scale factor of 2) to a DFT of $4N$ real inputs of even symmetry where the even-indexed elements are zero. That is, it is half of the DFT of the $4N$ inputs $y_n$, where $y_{2n} = 0$, $y_{2n+1} = x_n$ for $0 \le n < N$ , and $y_{4N-n} = y_n$ for $0 < n < 2N$. Some authors further multiply the $X_0$ term by $1/\sqrt{2}$ (see below for the corresponding change in DCT-III). This makes the DCT-II matrix orthogonal, if one further multiplies by an overall scale factor of $\sqrt{2/N}$ , but breaks the direct correspondence with a real-even DFT of half-shifted input.

The DCT-II implies the boundary conditions: $x_n$ is even around $n$=-1/2 and even around $n$=$N$-1/2; $X_k$ is even around $k$=0 and odd around $k$=$N$.

**DCT-III**

$$X_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left[\frac{\pi}{N} n\left(k+\frac{1}{2}\right)\right] \qquad k = 0,\ldots,N-1. \tag{2-3}$$

Because it is the inverse of DCT-II (up to a scale factor, see below), this form is sometimes simply referred to as "the inverse DCT" ("IDCT"). Some authors further multiply the $x_0$ term by $\sqrt{2}$ (see above for the corresponding change in DCT-II), so that the DCT-II and DCT-III are transposes of one another. This makes the DCT-III matrix orthogonal, if one further multiplies by an overall scale factor of $\sqrt{2/N}$ , but breaks the direct correspondence with a real-even DFT of half-shifted output.

The DCT-III implies the boundary conditions: $x_n$ is even around $n$=0 and odd around $n$=$N$; $X_k$ is even around $k$=-1/2 and even around $k$=$N$-1/2.

**DCT-IV**

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)\right] \qquad k = 0,\ldots,N-1. \tag{2-4}$$

The DCT-IV matrix becomes orthogonal if one further multiplies by an overall scale factor of $\sqrt{2/N}$ . A variant of the DCT-IV, where data from different transforms is *overlapped*, is called the modified discrete cosine transform (MDCT).

The DCT-IV implies the boundary conditions: $x_n$ is even around $n$=-1/2 and odd around $n$=$N$-1/2; similarly for $X_k$.

**DCT V-VIII**

DCT types I-IV are equivalent to real-even DFTs of even order (regardless of whether $N$ is even or odd), since the corresponding DFT is of length $2(N-1)$ (for DCT-I) or $4N$ (for DCT-II/III) or $8N$ (for DCT-VIII). In principle, there are actually four additional types of discrete cosine transform (Martucci, 1994), corresponding essentially to real-even DFTs of logically odd order, which have factors of $\mathrm{N} \pm 1/2$ in the denominators of the cosine arguments.

Equivalently, DCTs of types I-IV imply boundaries that are even/odd around either a data point for both boundaries or halfway between two data points for both boundaries. DCTs of types V-VIII imply boundaries that even/odd around a data point for one boundary and halfway between two data points for the other boundary.

However, these variants seem to be rarely used in practice. One reason, perhaps, is that FFT algorithms for odd-length DFTs are generally more complicated than FFT algorithms for even-length DFTs (e.g. the simplest radix-2 algorithms are only for even lengths), and this increased intricacy carries over to the DCTs as described below.

(The trivial real-even array, a length-one DFT (odd length) of a single number $a$, corresponds to a DCT-V of length $N$=1.)

### 2.1.2 Multidimensional Discrete Cosine Transform

Multidimensional variants of the various DCT types follow straightforwardly from the one-dimensional definitions: they are simply a separable product (equivalently, a composition) of DCTs along each dimension.

For example, a two-dimensional DCT-II of an image or a matrix is simply the one-dimensional DCT-II, from above, performed along the rows and then along the columns (or vice versa). That is, the 2d DCT-II is given by the formula (omitting normalization and other scale factors, as above):

$$X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1} x_{n_2} \cos\left[\frac{\pi}{N_1}\left(n_1 + \frac{1}{2}\right)k_1\right] \cos\left[\frac{\pi}{N_2}\left(n_2 + \frac{1}{2}\right)k_2\right] \tag{2-5}$$

$$k_1 \ or \ k_2 = 0,\ldots,N-1.$$

Technically, computing a two- (or multi-) dimensional DCT by sequences of one-dimensional DCTs along each dimension is known as a *row-column* algorithm (after the two-dimensional case). The algorithm used for the calculation of the 2D DCT is based on the equation (2-5). First, the 1D DCT of the rows are calculated and then the 1D DCT of the columns are calculated. The 1D DCT coefficients for the rows and columns can be calculated by separating equation (2-5) into the row part and the column part. As with multidimensional FFT algorithms, however, there exist other methods to compute the same thing while performing the computations in a different order (i.e. interleaving/combining the algorithms for the different dimensions).

In figure 2.2, for the case of *8x8* block region, a 1D 8-point DCT/IDCT followed by an internal double buffer memory (or ping-ping or transpose buffer), followed by another 1D 8-point DCT provided the 2D DCT architecture. The buffer memory is to store the data computed from the first 1-D DCT/IDCT part and re-sequence them to the second 1-D DCT/IDCT part with correct ordering. The double buffer memory performs a matrix transpose operation and needs to be fast enough to keep up with the data received from the first 1-D DCT/IDCT part and to supply the data going to the second 1-D DCT/IDCT part. The transpose memory has to hold an entire block of $N$ x $N$ data points because the second 1-D DCT/IDCT part can not start computations until the first 1-D DCT/IDCT part finished an entire block.

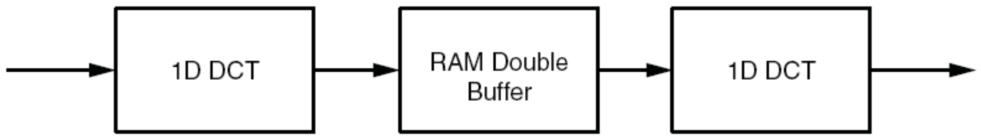


**Figure 2.2** The generic 2D-DCT architecture

Vector processing using parallel multipliers is a method used for implementation of DCT. The advantages in the vector processing method are regular structure, simple control and interconnect, and good balance between performance and complexity of implementation.

The inverse of a multi-dimensional DCT is just a separable product of the inverse(s) of the corresponding one-dimensional DCT(s) (see above), e.g. the one-dimensional inverses applied along one dimension at a time in a row-column algorithm.

In figure 2.3, the image to the right shows combination of horizontal and vertical frequencies for an 8 x 8 ($N_1$ = $N_2$ = 8) two-dimensional DCT. Each step from left to right and top to bottom is an increase in frequency by 1/2 cycle. For example, moving right one from the top-left square yields a half-cycle increase in the horizontal frequency (goes from white to black). Another move to the right yields two half-cycles (white to black to white). A move down yields two half-cycles horizontally and a half-cycle vertically. The source data (8x8) is transformed to a linear combination of these 64 frequency squares.

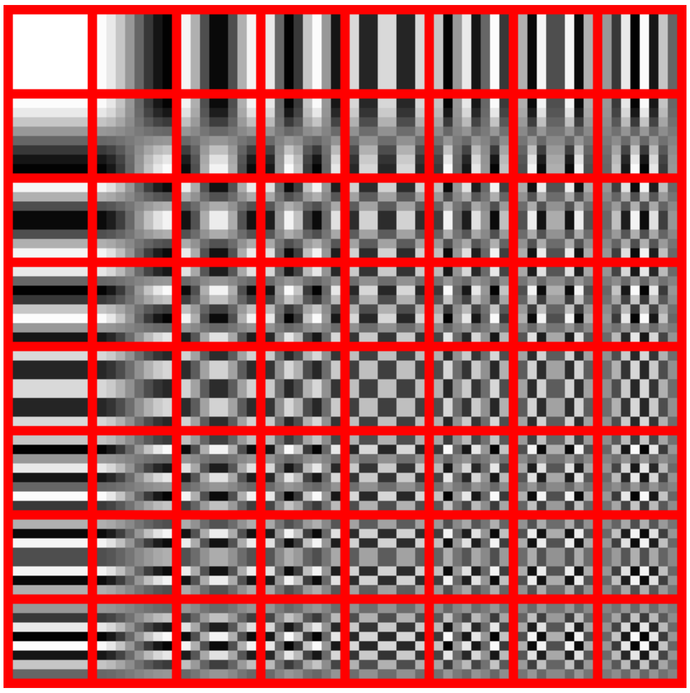


**Figure 2.3** 2D-DCT frequencies.

## 2.2 Ping Pong Buffers for Transmission

Memory Bandwidth is frequently a limiting factor in the design of high-speed switches and routers. A buffering scheme called ping-pong buffering increases memory bandwidth by a factor of two. Ping-pong buffering halves the number of memory operations per unit time allowing faster buffers to be built from a given type of memory.

Figure 2(a) shows a memory buffer with arrival (*An*) and departure (*Dn*) processes of cells. In each cell time, which we call a time-slot, zero (*An* = 0) or one (*An* = 1) new cell may arrive, and zero (*Dn* = 0) or one (*Dn* = 1) cell may depart from the buffer. This means that two independent memory operations are required per cell time: one write, and one read. If dual-ported memory is used, it would be possible for both operations to take place simultaneously. However, commercial considerations generally dictate that conventional single-ported memory be used. As a result, the total memory bandwidth must be at lease twice the line rate.

Figure 2(b) shows a ping-pong buffer of total capacity M (cells), with the arrival and the departure processes denoted as *An* and *Dn*, respectively. The main benefit of a ping-pong buffer is that using conventional memory devices, it allows the design of buffers operating twice as fast. But ping-pong buffer's benefit comes with a penalty. If the amount of memory is not increased, the overflow rate is from a ping-pong buffer is larger than for a conventional buffer. In the worst case, half of the memory is wasted. Using simulations, fortunately, the problem is eliminated by the addition of just 5% more memory [9].

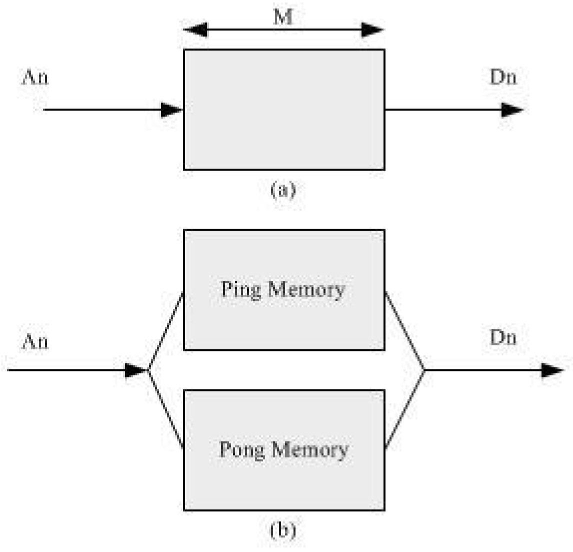


**Figure 2.4** (a) A buffer of capacity M. (b) A Ping-Pong Memory

In network communication, a transmission buffering method that involves two buffers: one buffer receives transmissions while the second deletes earlier transmissions. The two alternate functions, which helps to keep transmissions close to continuous. A ping-pong buffer contains two separate buffers; while one buffer is receiving new transmission information the other buffer is deleting the previous transmission.

We can also find ping-pong buffers in a front end system. The system view is illustrated in figure 2.5. For example, a front end interface consists of a number of ping-pong buffers, one for each LAN attachment; an internal bus with an arbitrator and a bus interface; a microprocessor; and main memory. An incoming frame is placed in one of the ping-pong buffers, if a buffer is available. Once the buffer is full or the last bit of the frame is received, then a signal is raised to inform the bus arbitrator that a buffer is ready for being emptied. Various scheduling policies such as First Come First Served, Served in Fixed Order, or non-preemptive priority scheme, are possible to serve multiple ping-pong buffers associated with various attachments. If a buffer is not available, then an incoming frame is assumed to be lost. An algorithm is developed and used to investigate the performance characteristics of the ping-pong buffering scheme [10].Once a permission to transfer is received by a ping-pong buffer, the frame is transferred from the ping-pong buffer to the main memory via the internal bus. It is further assumed that the main memory is large enough that it does not cause any loss of segments. Finally, the transfers from the memory to the front end processor are not explicitly considered here.

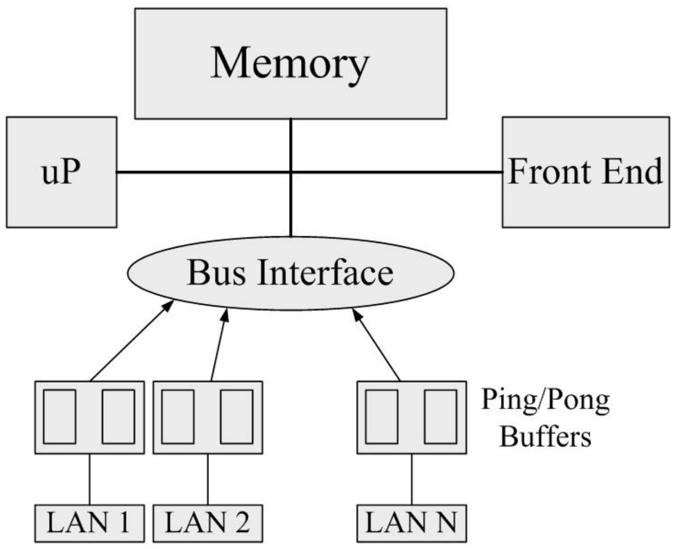


**Figure 2.5** System View[4]

# Chapter 3 Sandwich Ping-Pong Memory

In this chapter, we will propose our design. First of all, we will show the architecture and the operation of a double buffer (Ping-Pong or transpose buffer). We will find the two idled area in a Ping Pong Memory and develop a Common Bar to replace them. As a result, we develop a Sandwich Ping Pong Memory. Second, we introduce the operations of the Sandwich Ping-Pong Memory and derive the formula of *Initial Time* and *Idle Time*. We find some conditions from doing the timing analysis for this design finally.

## 3.1 The use of Sandwich Ping-Pong Memory

### 3.1.1 The Operation of Ping Pong Memory

We have presented a double buffer roughly and realized that its application is in the 2-D DCT architecture in chapter 2. We are going to introduce the architecture and the operation of a double buffer, also known as, Ping-Pong buffer which is shown in figure 3.1.

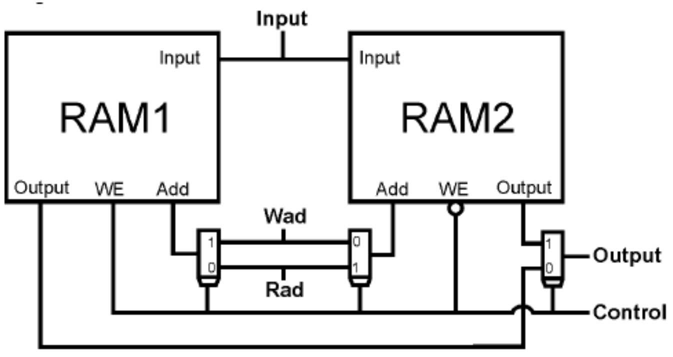


**Figure 3.1** The architecture of a Ping-Pong buffer

There are RAMs in figure 3.1 with some signals such as input data, output data, and address signals for writing and reading. The Control signal controls the write and read operation of the Ping Pong Memory. When RAM1/RAM2 is on read operation, RAM2/RAM1 is on read operation. There are address signals for write and read operations. The address sequence of writing and reading is different, showed in figure 3.2. We write data into RAM1/RAM2 row by row and read data from RAM2/RAM1 column by column simultaneously. There are some things we really concern about. Is there any memory cell idled during the write or read operation? If so, what could we do?

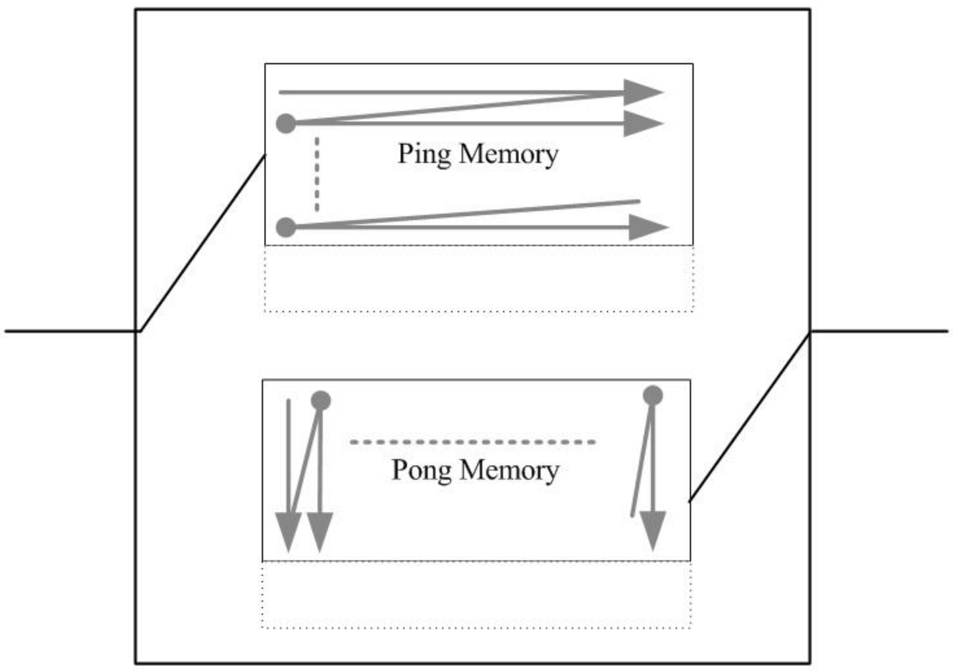


**Figure 3.2** Ping Pong Memory is on write/read operation

## 3.1.2 Common Bar

There are some idled memories in a Ping Pong Memory. The idled memories are showed in figure 3.3. There are one block of dotted line area in the Ping Memory and one in the Pong Memory, respectively. The dotted line area means the idled memories in a Ping Pong Memory.

We combine the two dotted line area into one and named it "Common Bar." As a result, there is a block of memory, Common Bar, between the Ping and Pong Memory. That is the Sandwich Ping Pong Memory.

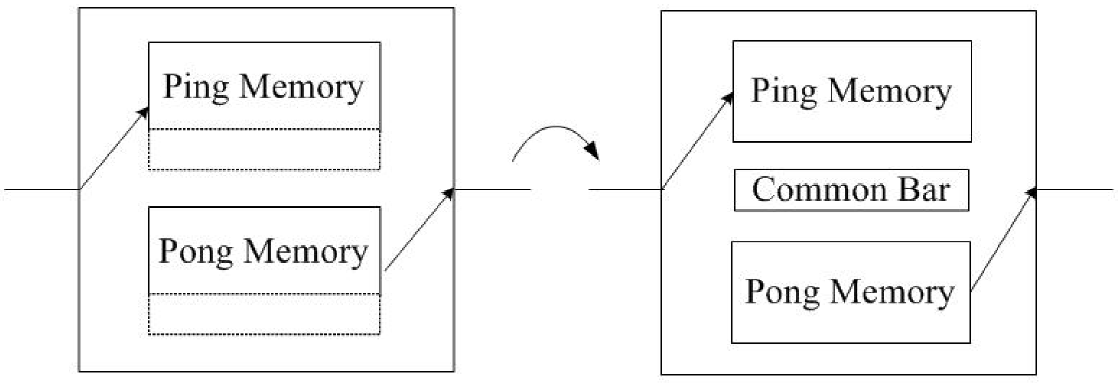


**Figure 3.3** Common Bar

## 3.1.3 Sandwich Ping Pong Memory

We develop a Sandwich Ping-Pong Memory based on the double buffer. Figure 3.4 shows the architecture of the Sandwich Ping-Pong Memory which is built up by adding one single-port memory between the ping memory and pong memory.

The architecture of the Common Bar is exactly the same as the ping or pong memory, because they are all the same type of construction. In theory, the double buffer is used in the architecture of 2-D DCT, so is the Sandwich Ping-Pong Memory. By the result of simulation and verification on FPGA, we can prove that Sandwich Ping-Pong Memory work as transpose buffer which is used to connect the two 1-D DCT architectures once the first 1-D DCT outputs are row-wise and the second 1-D DCT inputs must be column-wise.

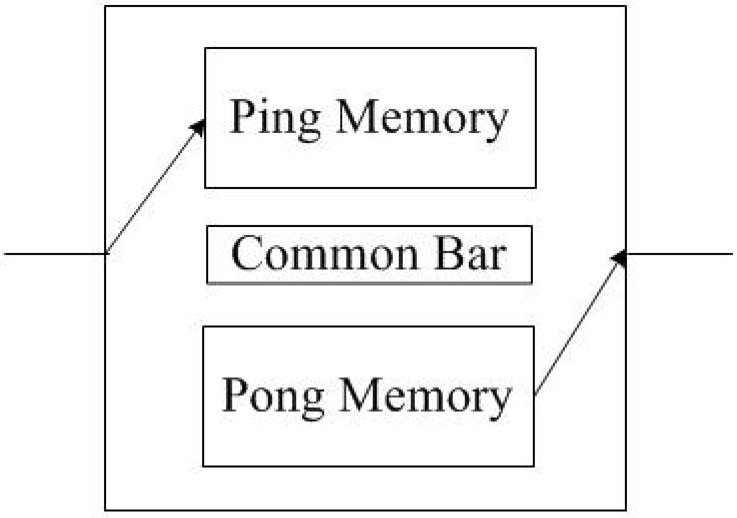


**Figure 3.4** The architecture of Sandwich Ping Pong Memory

## 3.2 Read / Write Operation

Figure 3.5 shows when the first 1-D DCT architecture writes the results row by row in one memory (ping or pong memory), the second 1-D DCT architecture reads the input values column by column from the other memory (pong or ping memory). The read and write signal addresses are generated by a control block and this control block defines, by control signal, which memory is used to Read/Write at each memory access step.

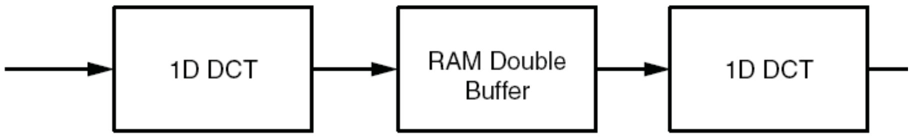


**Figure 3.5** The architecture of 2-D DCT

### 3.2.1 Row-column block memory

For a given 2-D spatial data sequence $\{X_{ij}; i, j = 0, 1, \ldots, N-1\}$, the 2-D DCT data sequence $\{Y_{pq}; i, j = 0, 1, \ldots, N-1\}$ is defined by:

$$Y_{pq} = E_p E_q \frac{2}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} X_{ij} \cos\left[\frac{\pi}{N}\left(i+\frac{1}{2}\right)p\right] \cos\left[\frac{\pi}{N}\left(j+\frac{1}{2}\right)q\right] \qquad (3\text{-}1)$$

where

$$E_x = \begin{cases} \frac{1}{\sqrt{2}}, & x = 0 \\ 1, & x \neq 0 \end{cases}$$

The forward and inverse transforms are merely mappings from the spatial domain to the transform domain and *vice versa*. The DCT is a separable transform and as such, the row-column decomposition can be used to evaluate (3-1).

Denoting:

$$\cos\left[\frac{\pi}{N}\left(h+\frac{1}{2}\right)l\right]$$

By $c_{lh}$ and neglecting the scale factor $E_p E_q \frac{2}{N}$, the column transform can be expressed as:

$$Y_{pq} = \sum_{j=0}^{N-1} Z_{pj} c_{qj}, \qquad p, q = 0, 1, 2, \ldots, N-1 \tag{3-2}$$

And the row transform can be expressed as:

$$Z_{pj} = \sum_{i=0}^{N-1} X_{ij} c_{pi}, \qquad p, j = 0, 1, 2, \ldots, N-1 \tag{3-3}$$

In order to compute an *N x N*-point DCT (where *N* is even), *N* row transforms and *N* column transforms need to be performed. However, by exploiting the symmetries of the cosine function, the number of multiplications can be reduced from $N^2$ to $N^2/2$. In this case each row transform given by (3-3) can be written as matrix-vector multipliers via,

$$Z_{pj} = \sum_{i=0}^{N/2-1} \left[ X_{ij} + (-1)^p X_{(N-1-i)j} \right] c_{pi} \tag{3-4}$$

Using a matrix notation, for *N*=8, (4) can be written as

$$\begin{bmatrix} Z_{00} \\ Z_{20} \\ Z_{40} \\ Z_{60} \end{bmatrix} = \begin{pmatrix} c_{00} & a_{01} & c_{02} & a_{03} \\ c_{20} & a_{21} & c_{22} & a_{23} \\ c_{40} & c_{41} & c_{42} & a_{43} \\ c_{60} & c_{61} & c_{62} & a_{63} \end{pmatrix} \begin{bmatrix} X_{00} + X_{70} \\ X_{10} + X_{60} \\ X_{20} + X_{50} \\ X_{30} + X_{40} \end{bmatrix} \tag{3-5}$$

$$\begin{bmatrix} Z_{10} \\ Z_{30} \\ Z_{50} \\ Z_{70} \end{bmatrix} = \begin{pmatrix} c_{10} & a_{11} & c_{12} & a_{13} \\ c_{30} & a_{31} & c_{31} & a_{33} \\ c_{50} & c_{51} & c_{51} & a_{53} \\ c_{70} & c_{71} & c_{71} & a_{73} \end{pmatrix} \begin{bmatrix} X_{00} - X_{70} \\ X_{10} - X_{60} \\ X_{20} - X_{50} \\ X_{30} - X_{40} \end{bmatrix} \tag{3-6}$$

Equations (3-5) and (3-6) describe the computation of the even and odd coefficients, for the row transform for N=8, respectively. The computation for the second 1-D DCT i.e. the column transform described by (3-2) can also be computed using matrix-vector multipliers similar to that described by (3-4). Hence both the row and column transform can be performed using the same architecture.

According to the 2-D DCT algorithm, there should be a row-column block ping pong memories to access the data. For example, the data of the computation of the even and odd coefficients should be stored in some memory. In the next section, we will present the scan line in Sandwich Ping Pong Memory.

### 3.2.2 Scan line of the Sandwich Ping-Pong memory

According to the 2-D DCT algorithm, the scan line of the write and read operations in Sandwich Ping Pong Memory are row by row and column by column, shown in figure 3.6 and 3.7, respectively.

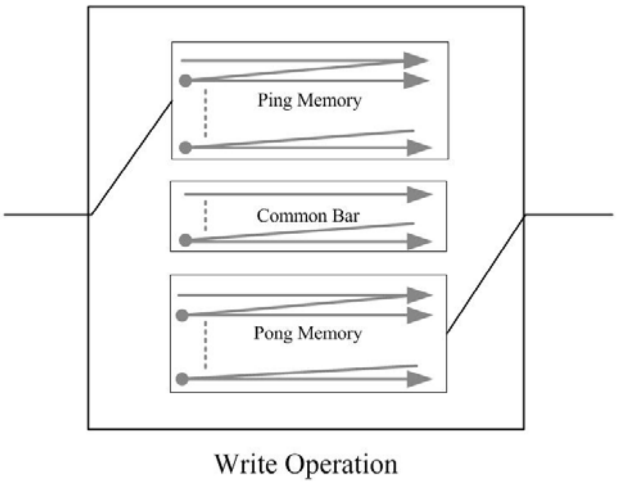


**Figure 3.6** Row by row on write operation

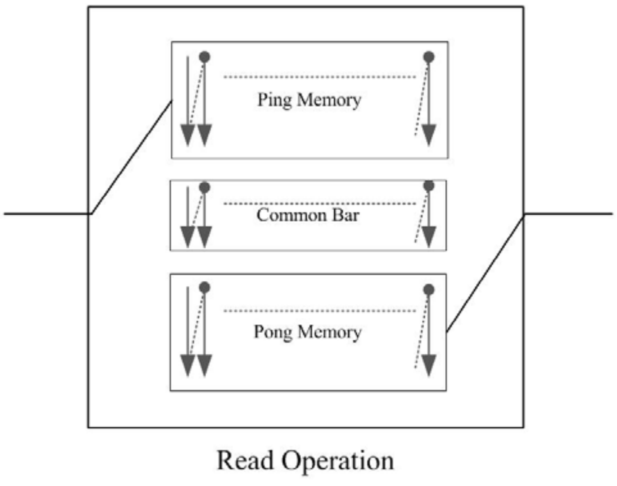


**Figure 3.7** Column by column on read operation

The scan line of writing is as the following step. When write operation starts, data is write in the Pong/Ping memory and Common Bar in sequence. First, data is written in the Ping/Pong memory row by row till the Ping/Pong memory is full. After the Ping/Pong memory is fully occupied, data is written in the single port memory, as know as *Common Bar*. Finally, data is written in the Pong/Ping memory row by row definitely.

On the other hand, when read operation starts, data is read from the Pong/Ping memory and Common Bar. Data is read from the Pong/Ping memory column by column. However, reading scan line is a little different from writing scan line. On write operation, we do write data into the Common Bar until we finish writing them into Ping/Pong memory. However, on the read operation, we read the data from Pong/Ping memory and Common Bar by turns. Data is read from the Pong/Ping memory, Common Bar, and back to Pong/Ping memory. Finally, we read the last data from the last on address of Common Bar; we finished the complete read operation. We must know that there is data written in the memory, at the same moment, there is data read from the memory. Write and read operation are took place simultaneously.

The operation of the transpose memory can be explained if we visualize it as an 8 x 8 array. It is actually implemented as a 64-byte SRAM. The first eight bytes of the SRAM correspond to the first row of the array, the second eight bytes, to the second row, and so on. Let *mode 1* be a sequence of accesses to locations {0, 1, 2, 3, 4, 5, 6, 7, 8 ...} in that order. This corresponds to scanning rows starting at the top left corner. Let *mode 2* be accesses to locations {0, 8, 16, 24, 32, 40, 48, 56, 1, 9 ...} in that order. This corresponds to scanning columns starting at the top left corner.

The transposition occurs as follows. Data is read out according to mode 1 for the first 64 clock cycles. New data (that needs to be transposed) is also written according to mode 1. A write always follows a read; i.e., a read from a location is always followed by a write to that location. For the next 64 clock cycles, reads and writes occur according to mode 2. The data which is read out is the transpose of the data which was written in during the previous 64 clock cycles. As a result, the latency of the transpose operation is 64 clock cycles.

## 3.3 Timing Analysis

In the section, we will derive the timing analysis about *Initially Idle Time* and *Idle Time* and some conditions or constraints for the Sandwich Ping Pong Memory.

### 3.3.1 The Initially Idle Time

When could we start to read the data from the Sandwich Ping Pong Memory? We have already derived when to read, the *Initially Idle Time*. Let's see our derivation.

We make an example for the Sandwich Ping Pong Memory which is a *N X M* rectangle in size, in figure 3.8. There are three coefficients in the derivation: ***N, M*** and ***P***. The coefficient ***N*** represents the cell number of columns of the Sandwich Ping Pong Memory. The coefficient ***P*** represents the cell number of rows of the Common bar. Hence, the cell number of rows of the Ping or Pong Memory is *M*-***P***. The individual size of Ping (or Pong) memory and Common Bar are $(M-P)\times N$ and $P\times N$.

In addition, we assume that the access time to one the memory cell is one unit time. Therefore, the time to write data in Ping/Pong memory is $(M-P)\times N$, and the time to write data in Common Bar is $P\times N$.

The scan line of writing operation is row by row in the Ping/Pong memory and Common Bar in sequence. We should wait for a period of time named *Idle Time* and continue the next write operation. On the other hand, the scan line of reading operation is column by column by turns of Pong/Ping memory and Common Bar. In the same manner, we should wait a moment and continue the next read operation. First, we derive the *Initial Time*, and followed by *Idle Time.*

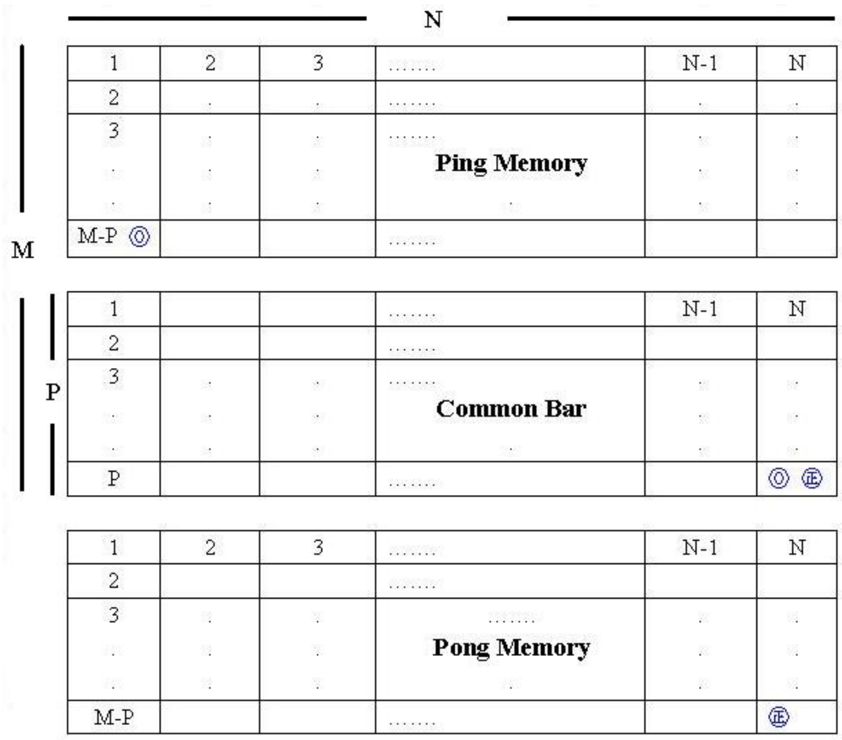


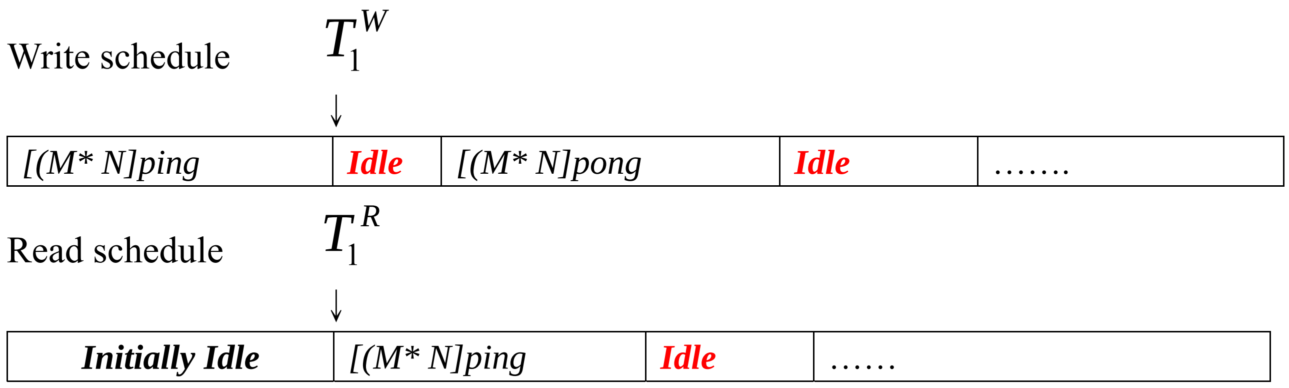
**Figure 3.8** Coefficients for Sandwich Ping Pong Memory

From the time schedule and the figure 3.8, we know that the data is read in the Ping memory column by column by turns of Ping/Pong memory and Common Bar. The first step to operate the Sandwich Ping Pong Memory is to write the data into the part of Ping memory and Common Bar. After writing, we start to read the data from them. The duration to read the first data is defined as "*Initially Idle Time*". After the *Initially Idle* we can read the data from the Ping memory and Common Bar, and the *Initially Idle* is presented as below. We make a time schedule to explain the derivation. Some constraints come to us.

After filling in the Ping memory and Common Bar with data, we start to read the data. There is the first constraint in Fig.3.9

**Condition 1:**

**Initial nonzero utilization constraint:** $T_1^W \geq T_1^R$



$$T_1^W \geq T_1^R \quad \Rightarrow \quad M * N \geq \mathit{Initially\ Idle}$$

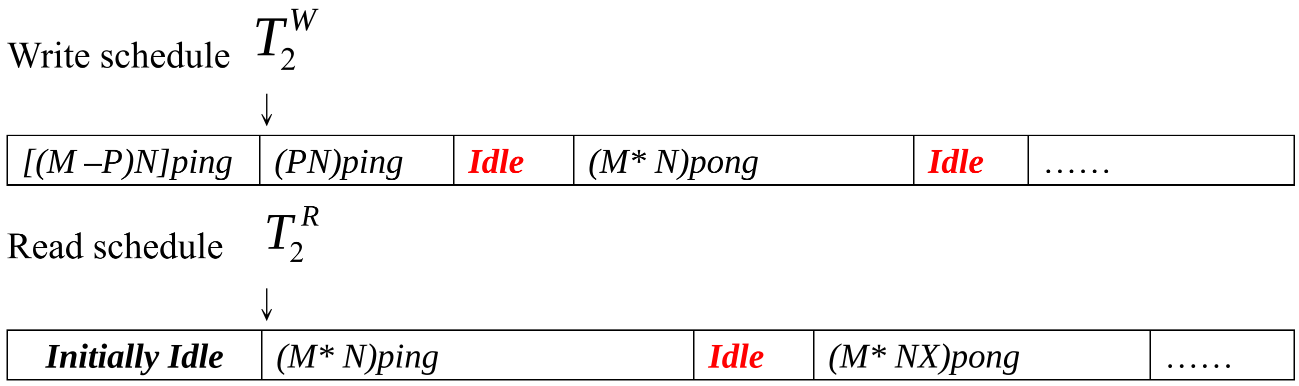
**Figure 3.9** The first constraint

In Fig. 3.9, the time to fill the data in the Ping memory and Common Bar ($T_1^W$) must be great than the *Initially Idle* time ($T_1^R$). Because we have to write the data in the Ping memory and Common Bar first, we read the data from them after the *Initially Idle* time.

After observing the write operation in detail, we found that data are written into the Ping memory first and into the Common Bar later. The second constraint comes up in Fig. 3.10.

**Condition 2:**

**Ping memory read contention constraint:** $T_2^W \leq T_2^R$



$$T_2^W \leq T_2^R \quad \Rightarrow \quad (M - P) * N \leq \mathit{Initially\ Idle}$$

**Figure 3.10** The second constraint

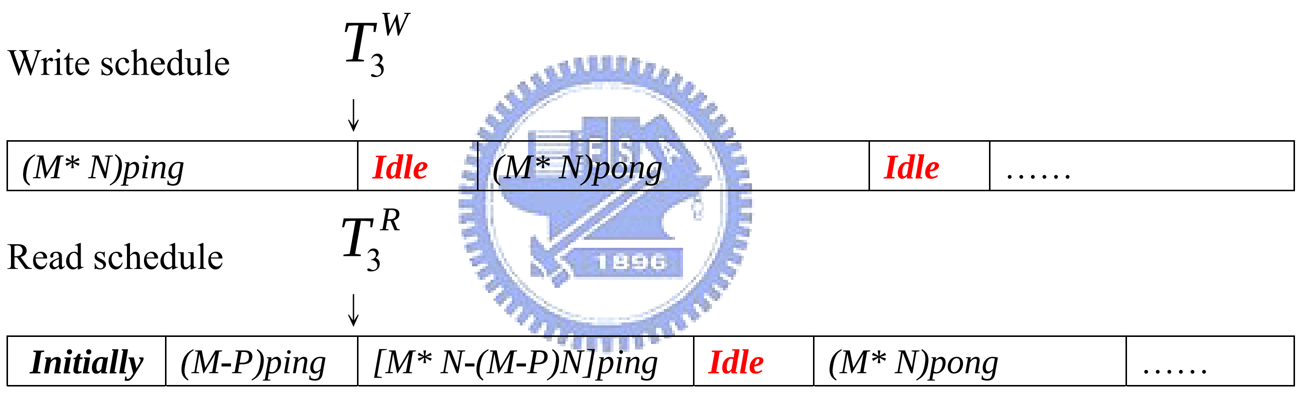
In Fig. 3.10, the time to fill the data in the Ping memory ($T_2^W$) must be shorter than the *Initially Idle* time ($T_2^R$). That is because we want to read the data earlier. We can read the data from the Ping memory and write the others into the Common Bar simultaneously.

In addition, the way we read the data is column by column. Before reading the data from the Common Bar, the data should already be written into the Ping memory and Common Bar. Therefore, we have the third constraint in Fig. 3.11

.

**Condition 3:**

**Common Bar - read memory contention constraint:** $T_3^W \le T_3^R$



$$T_3^W \le T_3^R \;\Rightarrow\; M * N \le Initially + (M - P)$$

**Figure 3.11** The third constraint

In Fig. 3.11, the time to write the data in the Ping memory and Common Bar ($T_3^W$) must be shorter than the time to read the data from the first column in Ping memory ($T_3^R$). Because after filling the data into the Common Bar, we could read the data it later.

In conclusion, according condition 1, 2 and 3, we derive the formula (3-7).

$$\begin{cases}(M-P)*N \le Initially\ Idle \le\ M*N \\ M*N \le Initially\ Idle + (M-P)\end{cases}$$

$$\Rightarrow \begin{cases}(M-P)*N \le Initially\ Idle \le\ M*N \\ M*N-(M-P) \le Initially\ Idle\end{cases}$$

$$\because P>1 \cap N>M$$

$$\therefore PN > M-P$$

$$\Rightarrow M*N-(M-P) \le Initially\ Idle \le M*N \tag{3-7}$$

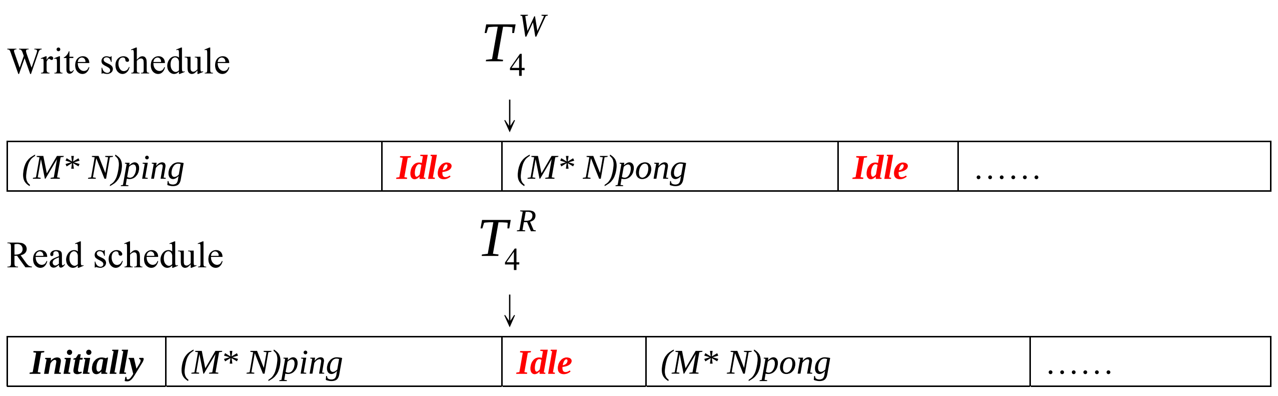
## 3.3.2 The Idle Time

After deriving the *Initially Idle Time*, we present the *Idle Time or Idle*. Again, from the time schedule and the figure 3.8, we know that the data is written in the Ping memory row by row in sequence of Ping memory and Common Bar. After the Ping memory is fully occupied by data, we wait for a period of the time, *Idle Time* because the Common Bar. The *Idle time* is presented with some conditions as below.

On the write schedule, we write the data into the Ping memory and Common Bar and wait for while, "*Idle Time*." Then we write the data into the Pong memory and Common Bar and so on. Simultaneously, on the read schedule, after the *Initially Idle Time*, we read the data from the Ping memory and Common Bar and wait for a while, "*Idle Time*." There should be a constraint to prevent the Sandwich Ping Pong memory from being on null operation. Therefore, we have the forth constraint in Fig. 3.12.

**Condition 4:**

**Run-time nonzero-utilization constraint:** $T_4^W \le T_4^R$



$$T_4^W \le T_4^R \quad \Rightarrow \quad M*N + Idle \le Initially + M*N$$

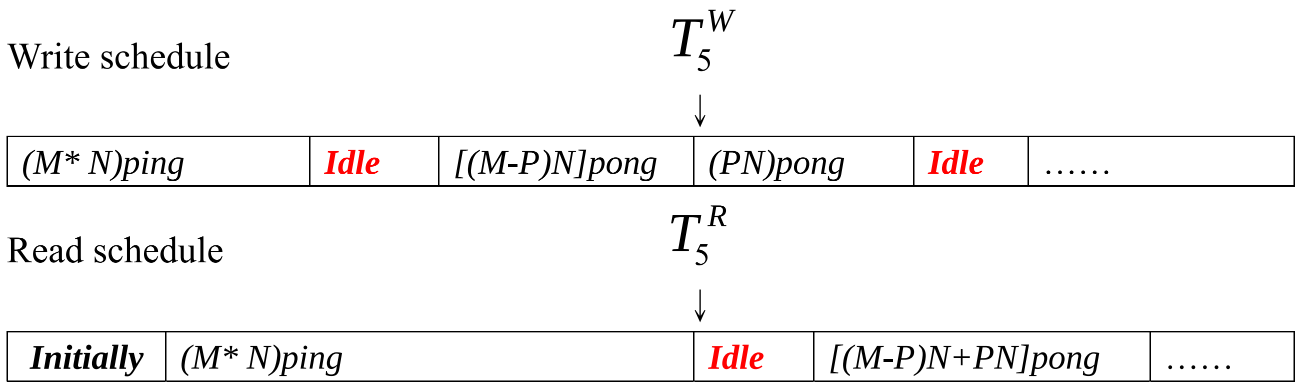
**Figure 3.12** The fourth constraint

In Fig. 3.12, during writing the data in the Ping memory, Common Bar and the *Idle* time ($T_4^W$) must be shorter than the time to read the data from the Ping memory, Common Bar ($T_4^R$). Because we have to prevent the *Idle* time on write operation and on the read operation from being happened in the meanwhile. If we don not have this constraint, the memory would be on null operation.

Before we write the data into the Common Bar, the data should already be read from the Ping memory and Common Bar. We have the fifth constraint in Fig. 3.13.

**Condition 5:**

**Common Bar: write memory contention constraint:** $T_5^W \geq T_5^R$



$$T_5^W \geq T_5^R \quad \Rightarrow \quad M*N + Idle + (M-P)*N \geq Initially + M*N$$

**Figure 3.13** The fifth constraint

In Fig. 3.13, during writing the data in the Ping memory, Common Bar, the *Idle* time and the Pong memory ($T_5^W$) must be greater than the time to read the data from the Ping memory, Common Bar ($T_5^R$). Because after reading the data from the Common Bar, we could write the data it later.

Therefore, according condition 4, 5, we derive the formula (3-8).

$$\begin{cases} M*N + Idle \leq Initially + M*N \\ M*N + Idle + (M-P)*N \geq Initially \ + M*N \end{cases}$$
$$\Rightarrow Initially - (M-P)*N \leq Idle \leq Initially \qquad (3\text{-}8)$$

We take the minimum of the *Initially Idle Time* in the formula (3-7), and we derive the formula (3-8).

$$\Rightarrow P*N - (M-P) \leq Idle \leq M*N - (M-P) \qquad (3\text{-}9)$$

In addition, we take the maximum of the *Initially Idle Time* in the formula (3-7), and we derive the formula (3-10).

$$\Rightarrow P * N \leq Idle \leq M * N \qquad (3\text{-}10)$$

No matter what the data is, they all access to the Sandwich Ping Pong Memory one by one. In fact, we usually use *8 x 8* block matrix in 2-D Discrete Cosine Transform. Hence, we should put some conditions and coefficients for formula (3-7) and (3-9).

Here is an example for M=4, N =4 and P=1, shown in figure 3.9.

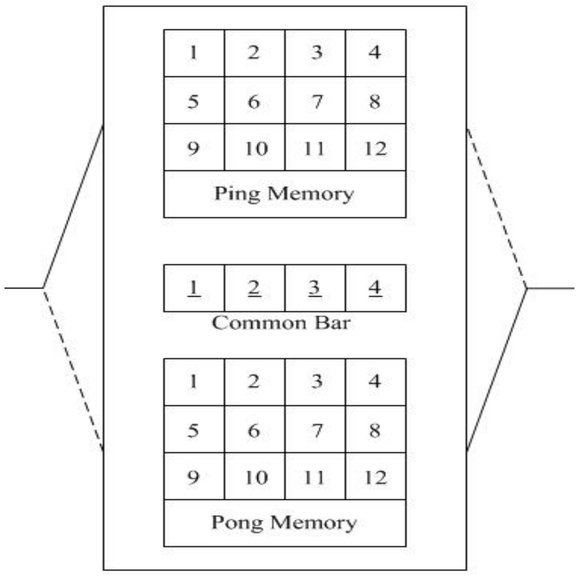
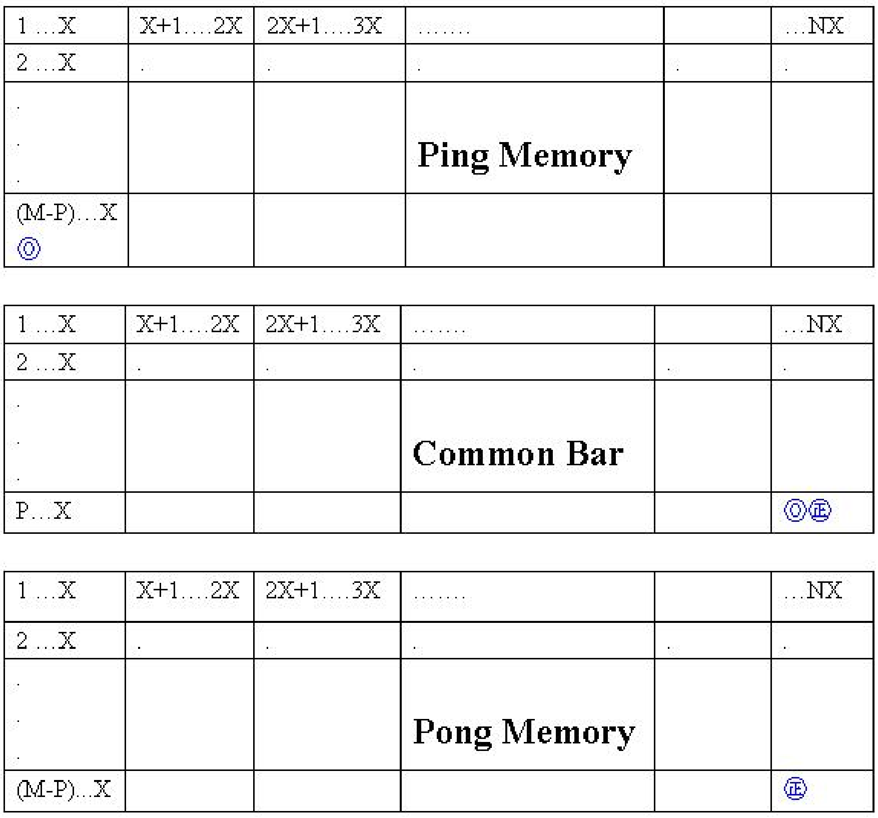


**Figure 3.14** Example for *Initial* and *Idle Time*

There are twelve memory cells in Ping and Pong Memory, respectively. There are four cells in Common Bar. We write data into the Ping Memory, then Common Bar and Pong Memory row by row. We read from the Ping Memory, then Common Bar and Pong Memory column by column. According formula (3-7) and (3-9), the *Initially Idle Time* and *Idle Time* are 13 and 1 unit time.

## 3.3.3 Line Buffer

Many algorithms and VLSI architectures for the fast computation of one-dimensional (1-D) and two-dimensional (2-D) DCT have been proposed [11]. For and effective VLSI implementation of an orthogonal transform, the corresponding algorithm should be numerically stable, and its computational structure should be regular (recursive and repetitive structure). The experiences with VLSI implementations show that the regularity of the algorithm is prime concern.[7] Almost all VLSI chips are implemented for fixed 8x8 or 16x16 square block sizes.[8]



| | | | | | |
|---|---|---|---|---|---|
| 1 ...X | X+1....2X | 2X+1....3X | ....... | | ...NX |
| 2 ...X | . | . | . | . | . |
| .<br>.<br>. | | | **Ping Memory** | | |
| (M-P)...X<br>⓪ | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| 1 ...X | X+1....2X | 2X+1....3X | ....... | | ...NX |
| 2 ...X | . | . | . | . | . |
| .<br>.<br>. | | | **Common Bar** | | |
| P...X | | | | | ⓪㊣ |

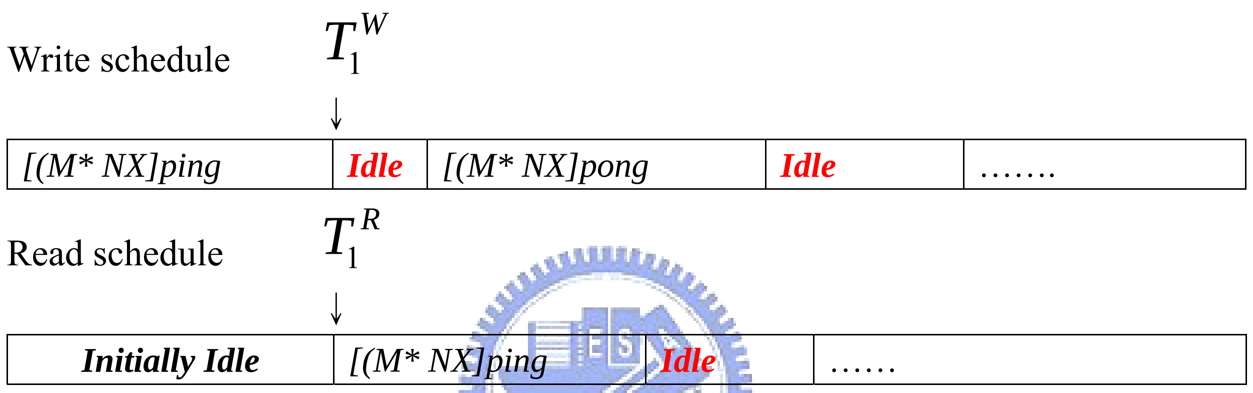
| | | | | | |
|---|---|---|---|---|---|
| 1 ...X | X+1....2X | 2X+1....3X | ....... | | ...NX |
| 2 ...X | . | . | . | . | . |
| .<br>.<br>. | | | **Pong Memory** | | |
| (M-P)...X | | | | | ㊣ |

**Figure 3.15** Coefficients for line buffer

Therefore, we have to put one more coefficient to represent the square block sizes. We choose *X* as our coefficient, and further, we modify the two formula (3-7) and (3-9). An example, in figure 3.10, is made for the Sandwich Ping Pong Memory and is called ling buffer. We explain the derivations of the *Initially Idle Time* formula in general form (3-11) with a general time schedule as below. After filling in Ping memory and Common Bar with data, we start to read the data. From Fig., we have the first constraint.

**Condition 1 for general form:**

**Initial nonzero utilization constraint:** $T_1^W \geq T_1^R$



$$T_1^W \geq T_1^R \quad \Rightarrow \quad M * NX \geq \textit{Initially Idle}$$

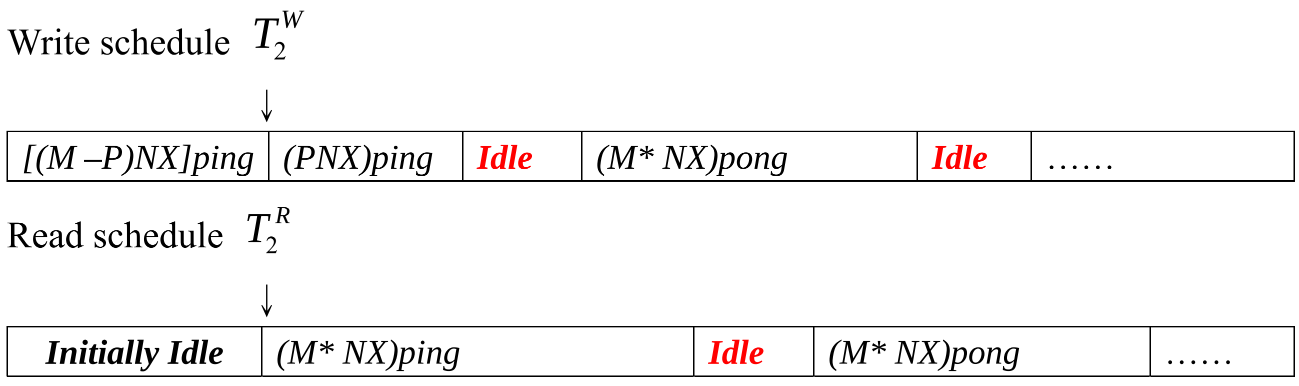
**Figure 3.16** The first constraint for general form

In Fig. 3.16, the time to fill the data in the Ping memory and Common Bar ($T_1^W$) must be great than the *Initially Idle* time ($T_1^R$). Because we have to write the data in the Ping memory and Common Bar first, we read the data from them after the *Initially Idle* time.

After observing the write operation in detail, we found that data are written into the Ping memory first and into the Common Bar later. The second constraint comes up in Fig. 3.17.

**Condition 2 for general form:**

**Ping memory read contention constraint:** $T_2^W \le T_2^R$



$$T_2^W \le T_2^R \quad \Rightarrow \quad (M-P)*NX \le Initially\ Idle$$

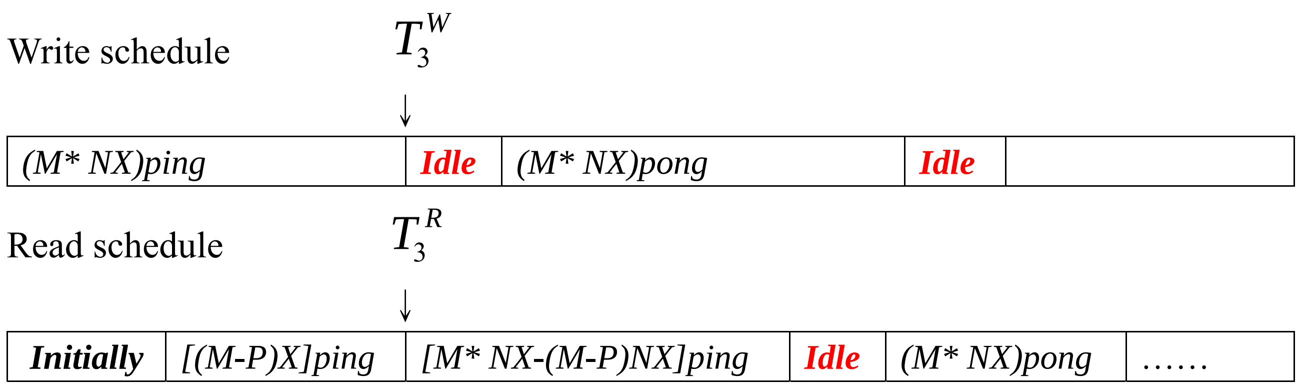
**Figure 3.17** The second constraint for general form

In Fig. 3.17, the time to fill the data in the Ping memory ($T_2^W$) must be shorter than the Initially Idle time ($T_2^R$). That is because we want to read the data earlier. We can read the data from the Ping memory and write the others into the Common Bar simultaneously.

Before reading the data from the Common Bar, the data should already be written into the Ping memory and Common Bar. Therefore, we have the third constraint in Fig. 3.18.

**Condition 3 for general form:**

**Common Bar - read memory contention constraint:** $T_3^W \le T_3^R$



$$T_3^W \le T_3^R \Rightarrow \quad M * NX \le Initially + (M - P)X$$

**Figure 3.18** The third constraint for general form

In Fig. 3.18, the time to write the data in the Ping memory and Common Bar ($T_3^W$) must be shorter than the time to read the data from the first column in Ping memory ($T_3^R$). Because after filling the data into the Common Bar, we could read the data it later.

In conclusion, according condition 1, 2 and 3 in general form; we derive the formula (3-11)

$$\begin{cases} (M-P) * NX \le Initially\ Idle \le \ M * NX \\ Initially\ Idle + (M-P) * X \ge M * NX \end{cases}$$

$$\Rightarrow \left(M * N - (M-P)\right) * X \le Initially\ Idle \le M * NX \qquad (3\text{-}11)$$

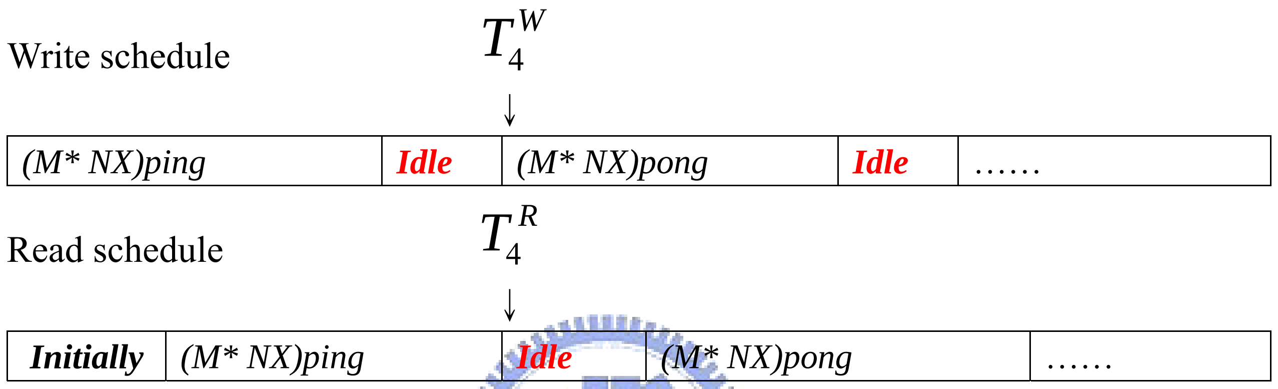
From the figure 3.10, we know that the data is read column by column in the Ping memory in *M* by *NX* block size. After the *Initially Idle Time*, we can read the data from the Ping memory and Common Bar, and the *Initially Idle Time* is presented in formula (3-11).

Similarly, we have to modify the formula (3-9) and (3-10) and derive the *Idle Time* is below. There should be a constraint to prevent the Sandwich Ping Pong memory from being on null operation. Therefore, we have the forth constraint in Fig. 3.19

.

**Condition 4 for general form:**

**Run-time nonzero-utilization constraint:** $T_4^W \le T_4^R$



$$T_4^W \le T_4^R \quad \Rightarrow \quad M * NX + Idle \le Initially + M * NX$$

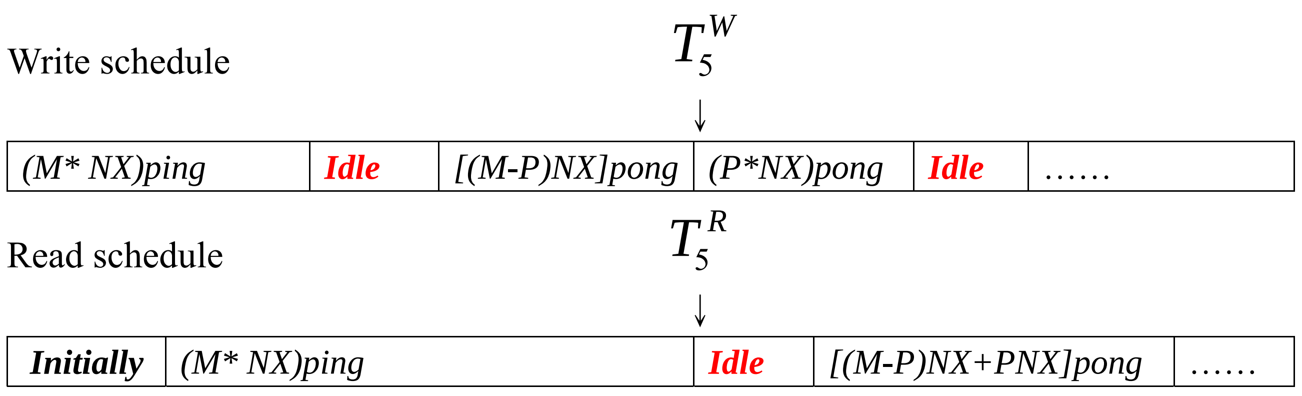
**Figure 3.19** The fourth constraint for general form

In Fig. 3.19, during writing the data in the Ping memory, Common Bar and the *Idle* time ($T_4^W$) must be shorter than the time to read the data from the Ping memory, Common Bar ($T_4^R$). Because we have to prevent the *Idle* time on write operation and on the read operation from being happened in the meanwhile. If we don not have this constraint, the memory would be on null operation.

Before we write the data into the Common Bar, the data should already be read from the Ping memory and Common Bar. We have the fifth constraint in Fig. 3.20.

**Condition 5 for general form:**

**Common Bar: write memory contention constraint:** $T_5^W \geq T_5^R$



$$T_5^W \geq T_5^R \quad \Rightarrow \quad M*NX + Idle + (M-P)*NX \geq Initially + M*NX$$

**Figure 3.20** The fifth constraint for general form

In Fig. 3.20, during writing the data in the Ping memory, Common Bar, the *Idle* time and the Pong memory ($T_5^W$) must be greater than the time to read the data from the Ping memory, Common Bar ($T_5^R$). Because after reading the data from the Common Bar, we could write the data it later.

Therefore, according condition 4, 5, we derive the formula (3-12) and (3-13).

$$\begin{cases} M*NX + Idle \leq Initially + M*NX \\ Initially + M*NX \leq M*NX + Idle + (M-P)*NX \end{cases}$$

$$\Rightarrow Initially - (M-P)*NX \leq Idle \leq Initially \qquad (3\text{-}12)$$

We take the minimum and maximum of the *Initially Idle Time* in the formula (3-12), and we derive the formula (3-13) and (3-14).

$$\Rightarrow \left[P*N-(M-P)\right]X \leq Idle \leq \left[M*N-(M-P)\right]X \qquad (3\text{-}13)$$

$$\Rightarrow P*NX \leq Idle \leq M*NX \qquad (3\text{-}14)$$

Generally, for block sizes larger than 16x16, the complexity of derivation of a signal flow graph for a given algorithm increase. Furthermore, no direct 2-D DCT algorithm has been shown up to now with its computational structure for a rectangular M x NX block size.

Moreover, there should be some conditions for the coefficients *M*, *N*, and *X*. For instance, *M* and *N* must be equal to or larger than *X*. Besides, *M* and *N* are usually less than 16 [12].

## 3.4 Control Unit Circuit

We make up the Sandwich Ping Pong Memory with two parts, a Ping Pong Memory and a Common Bar. We build a control unit circuit to turn the original signals and addresses into those for Sandwich Ping Pong Memory (SPPM). Saving area is our destination and we do save the area of Ping Pong Memory actually. We could save the area of Ping Pong Memory; however, we gained the area of the control unit circuit. Therefore, the less area we used for control unit circuit, the more area we saved in Sandwich Ping Pong Memory.

| Idle | WE | Addr_W>12 | Addr_W>12 | | WE_Ping | WE_Pong | WE_CMB |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 0 | 1 | X |
| 0 | 0 | 0 | 1 | | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | | X | X | X |
| 0 | 1 | 0 | 0 | | 1 | 0 | X |
| 0 | 1 | 0 | 1 | | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | | X | X | X |
| 1 | 0 | 0 | 0 | | X | 0 | 1 |
| 1 | 0 | 0 | 1 | | X | 0 | 1 |
| 1 | 0 | 1 | 0 | | X | 0 | 1 |
| 1 | 0 | 1 | 1 | | X | X | X |
| 1 | 1 | 0 | 0 | | 0 | X | 0 |
| 1 | 1 | 0 | 1 | | 0 | X | 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | | 0 | X | 0 |
| 1 | 1 | 1 | 1 | | X | X | X |

**Table 3.1** Signals and Addresses for 32bit SPPM with Common Bar 1X4

Table 3.1 shows the signals for 32-bit SPPM with Common Bar 1 X 4. We define the address for writing or reading is 0 to 15. When they are bigger than 12, the common bar is on operation. The inputs of it are WE, Addr_W and Addr_R represented the original signals of Write Enable, Address for Writing and Address for Reading, respectively. Its outputs are the WE_Ping, WE_Pong and WE_CMB represented the signals of Write Enable for Ping Memory, Pong Memory and Common Bar. It seems that we are using a Ping Pong Memory. In fact, we are using a Sandwich Ping Pong Memory. Figure 3.21 shows the signals and addresses transformed by the control unit circuit.

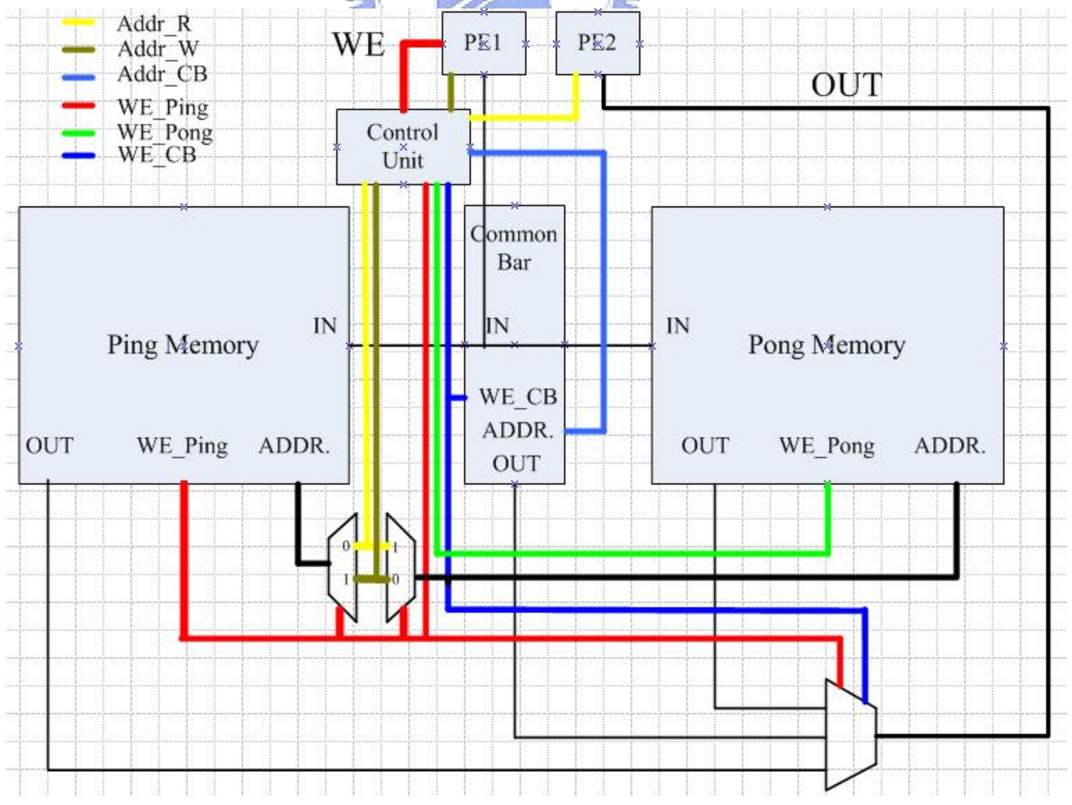
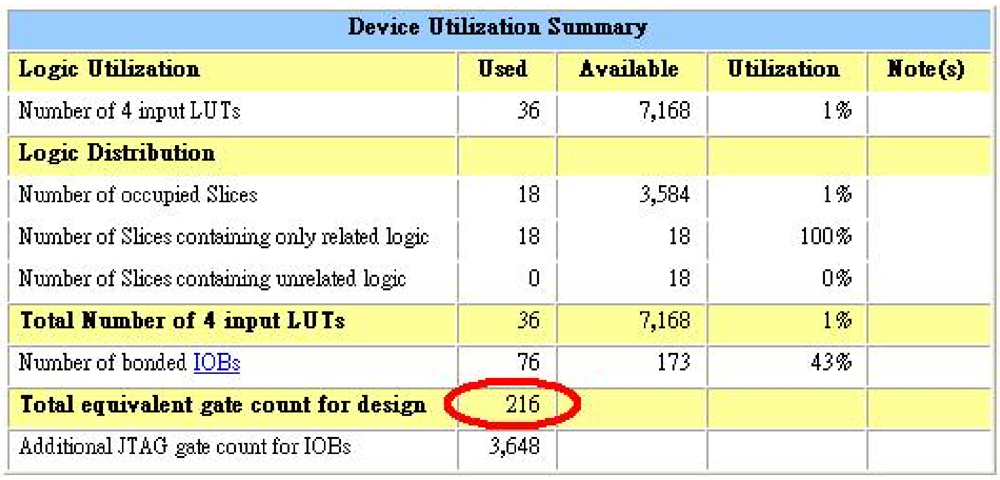


**Figure 3.21** Signals and addresses for Sandwich Ping Pong Memory

We synthesize a control circuit with HDL. When the size of Common Bar is *1 x 8K*, the gate count of the control circuit is 216 gate counts, shown in figure 3.22.



| Device Utilization Summary | | | | |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** | **Note(s)** |
| Number of 4 input LUTs | 36 | 7,168 | 1% | |
| **Logic Distribution** | | | | |
| Number of occupied Slices | 18 | 3,584 | 1% | |
| Number of Slices containing only related logic | 18 | 18 | 100% | |
| Number of Slices containing unrelated logic | 0 | 18 | 0% | |
| **Total Number of 4 input LUTs** | 36 | 7,168 | 1% | |
| Number of bonded IOBs | 76 | 173 | 43% | |
| **Total equivalent gate count for design** | 216 | | | |
| Additional JTAG gate count for IOBs | 3,648 | | | |

**Figure 3.22** The gate count of control unit for Common Bar

| M | N | P | X | Total Size (bit) | Initially Idle (Unit time) | Idle Time (Unit time) | Throughput Reduction (%) | Common Bar (gate count) | Control Unit (gate count) | Area Saved (gate count) |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 512 | 1 | 16 | 128K | 130832 | 7936 | 5.709 | 8192 | 216 | 7976 |
| 16 | 512 | 2 | 16 | 128K | 130848 | 15872 | 10.801 | 16384 | 230 | 16154 |
| 16 | 512 | 3 | 16 | 128K | 130864 | 23808 | 15.371 | 24576 | 251 | 24325 |
| 16 | 512 | 4 | 16 | 128K | 130880 | 31744 | 19.496 | 32768 | 260 | 32508 |
| 16 | 512 | 5 | 16 | 128K | 130896 | 39680 | 23.238 | 40960 | 264 | 40696 |
| 16 | 512 | 6 | 16 | 128K | 130912 | 47616 | 26.647 | 49152 | 266 | 48886 |
| 16 | 512 | 7 | 16 | 128K | 130928 | 55552 | 29.766 | 57344 | 268 | 57076 |
| 16 | 512 | 8 | 16 | 128K | 130944 | 63488 | 32.631 | 65536 | 283 | 65253 |
| 16 | 512 | 9 | 16 | 128K | 130960 | 71424 | 35.271 | 73728 | 287 | 73441 |
| 16 | 512 | 10 | 16 | 128K | 130976 | 79360 | 37.712 | 81920 | 289 | 81631 |
| 16 | 512 | 11 | 16 | 128K | 130992 | 87296 | 39.976 | 90112 | 283 | 89829 |
| 16 | 512 | 12 | 16 | 128K | 131008 | 95232 | 42.081 | 98304 | 280 | 98024 |
| 16 | 512 | 13 | 16 | 128K | 131024 | 103168 | 44.043 | 106496 | 264 | 106232 |
| 16 | 512 | 14 | 16 | 128K | 131040 | 111104 | 45.877 | 114688 | 238 | 114450 |
| 16 | 512 | 15 | 16 | 128K | 131056 | 119040 | 47.594 | 122880 | 223 | 122657 |
| 16 | 512 | 16 | 16 | 128K | 131072 | 126976 | 49.206 | 131072 | 210 | 130862 |

**Table 3.2** Data sheet for a Sandwich Ping Pong Memory

Table 3.2 is a data sheet for the Sandwich Ping Pong Memory in size 128K. It shows the control unit size of the different Common Bar in the Sandwich Ping Pong Memory. Every control unit is under three hundred gate counts. Figure 3.23 shows the Common Bar and Area Saved. Figure 3.24 shows the relationship between area saved and throughput reduction of the Sandwich Ping Pong Memory in the size 128k

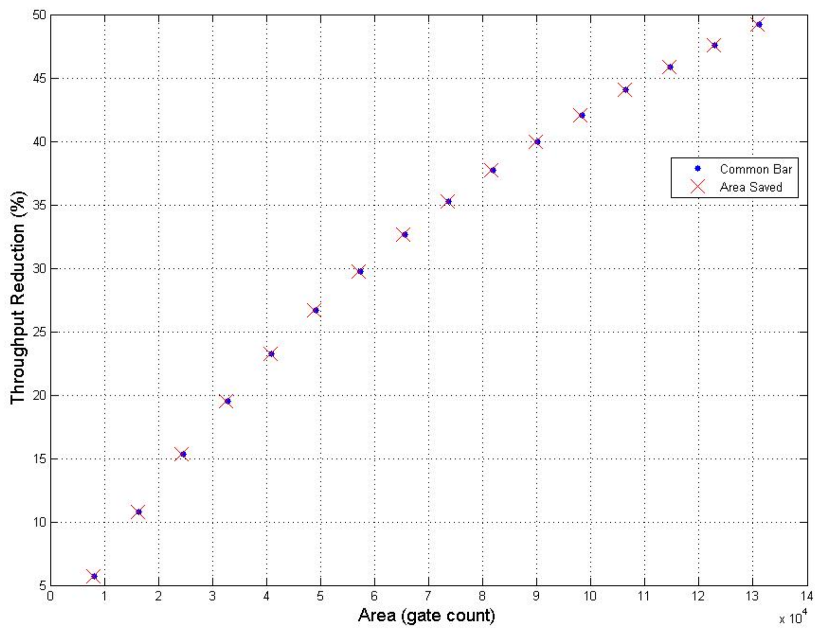


**Figure 3.23** Common Bar and Area Saved

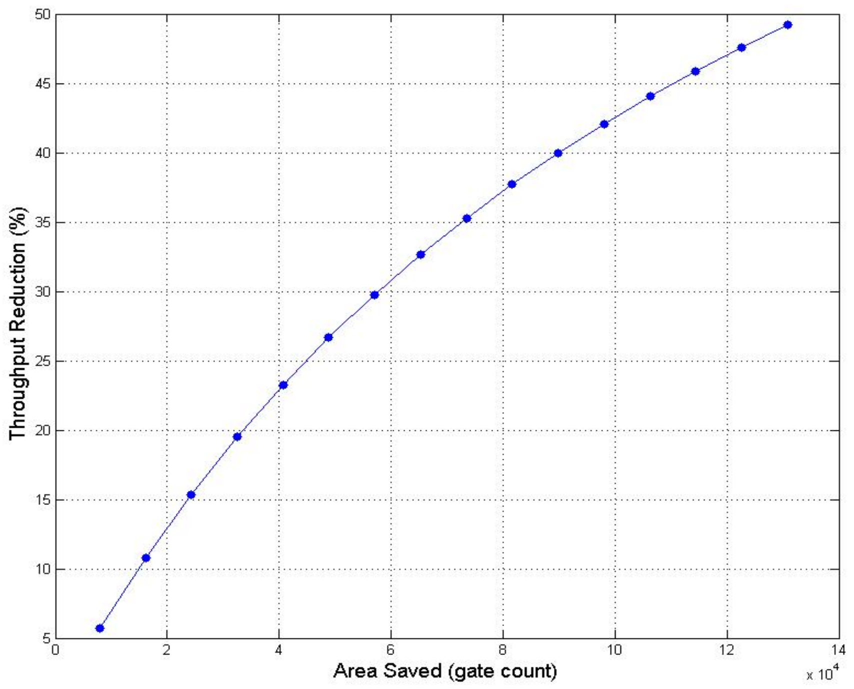


**Figure 3.24** Area Saved vs. Throughput Reduction

# Chapter 4 Test Algorithm

Semiconductor memories have invented for decades and have been designed, produced, tested by customers all over the world. The test algorithms have been studied for decade years. [13]~[23] It has been said that "memory testing is simple." In fact, it is logistically simple about memory testing. The complex part of memory testing is the numerous ways that a memory would fail. Patterns are the essence of memory testing. However, there is no single pattern is sufficient to test a memory for all defect types. There are many algorithms had been proposed such as, Zero-One, Checker, March, GALPAT, Butterfly, etc…. Table 4.1 lists the required test time as a function of the algorithm complexity and the memory size.

| Size | Algorithm complexity | | | |
|---|---|---|---|---|
| $n$ | $n$ | $n\log n$ | $n^{3/2}$ | $n^2$ |
| 1K | 0.0001s | 0.001s | 0.0033s | 0.105s |
| 16K | 0.0016s | 0.0224s | 0.21s | 27s |
| 256K | 0.0256s | 0.46s | 13.4s | 1.9h |
| 1M | 0.102s | 2.04s | 14.3m | 1.27d |
| 16M | 1.64s | 39.36s | 15.25h | 326d |
| 256M | 26.24s | 12.25m | 5.1d | 229y |

s: second; m: minute; h: hour; d: day; y: year

**Table 4.1** Test time as a function of memory size

This chapter introduces popular memory fault models and many March algorithms. For Sandwich Ping-Pong Memory, we show a modified March C- algorithm for testing. The testing time is shorten and with high fault coverage.

## 4.1 Fault Models

This section gives a formal definition for the most popular fault models.[24] First, we introduce the notation used to represent the fault models are listed here:

0: Denotes that a cell is in logical state 0.

1: Denotes that a cell is in logical state 1.

? : Denotes that a cell is in logical state, which means "don't care."

$\uparrow$ : A raising cell transition or denotes a write 1 operation to a cell containing a 0.

$\downarrow$ : A falling cell transition or denotes a write 0 operation to a cell containing a 1.

$\updownarrow$ : Either a rising or falling cell transition.

$\forall$ : denotes any operation; $\forall \in \{0,1,\uparrow,\downarrow,\updownarrow\}$

$< I / F >$: denotes a fault in a single cell. $I$ describes the condition for sensitizing the fault, $F$ describes the value of the faulty cell. $I \in \{0,1,\uparrow,\downarrow,\updownarrow\}$, and $F \in \{0,1\}$.

The most popular fault models are listed as follows:

*Stuck-at fault (SAF )*: The logic value of a stuck-at *(SA)* cell or line is always 0 ( a *stuck-at-0* fault, *SA0* ) or 1 ( a *stuck-at-1* fault, *SA1* ).

*Transition fault ( TF )*: The cell or line which fails to transit from 0 to 1 (a $<\uparrow/0>$ *TF* ) or from 1 to 0 ( a $<\downarrow/1>$ *TF* ).

*Inversion coupling fault ( CFin )*: An transition ($\uparrow$ or $\downarrow$) in one cell inverts the content of another cell.

*Idempotent coupling fault ( CFid )*: An transition ($\uparrow$ or $\downarrow$) in one cell forces the content of another cell to a certain value , 0 or 1.

*State coupling fault ( CFst )*: A coupled cell is forced to a certain value only if the coupling cell is in a given state.

*Stuck-open fault ( SOF )*: The cell fails to be accessed or a broken word/bit line.

*Address decoder fault ( AF* ): It is a functional fault in the address decoder that results in one of the following four cases shown in figure 4.1.

Fault A: With a certain address, no cell will be accessed.

Fault B: A certain cell is never accessed.

Fault C: With a certain address, multiple cells are accessed.

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

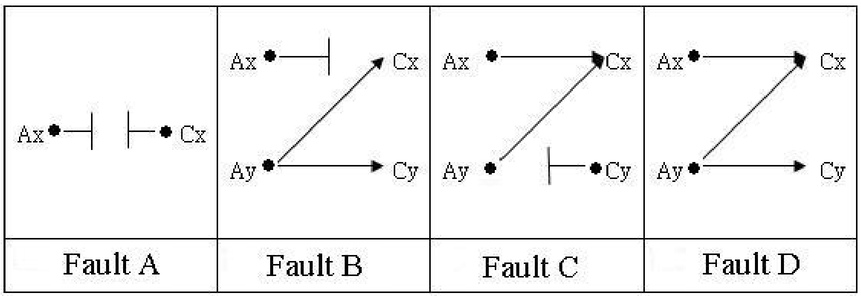


**Figure 4.1** Different types of address decoder faults

## 4.2 March Algorithms

The simplest tests which detect SAFs, TFs and CFs are called ‘marches’. A March test is composed of a finite sequence of March elements. A March element is a finite sequence of write/read operations applied to every cell in memory before proceeding to the next cell. The address sequence can be either an increasing ( $\Uparrow$ ) address order (e.g. from address 0 to address *N*-1), or a decreasing ( $\Downarrow$ ) address order which is the opposite of the $\Uparrow$ address order.

A write/read operation can be $(wa)$, $(w\bar{a})$, $(ra)$, and $(r\bar{a})$ where $a$ is the background pattern and $\bar{a}$ is the inverted background pattern; $a \in \{0,1\}$ ; $(wa)$ means “write the cell/word $a$”; $(w\bar{a})$ means “write the cell/word $\bar{a}$”; $(ra)$ means “read a expected cell/word

$a$"; $(r\overline{a})$ means "read a expected cell/word $\overline{a}$." A March algorithm example is shown in figure 4.2, and its flow is depicted in figure 4.3. Once the fault simulation is complete (all faults have been emulated), the fault coverage can be determined for the set of test vectors. The fault coverage, *FC*, is a quantitative measure of the effectiveness of the set of test vectors in detecting faults, and in it most basic form is given by:

$$FC = \frac{D}{T} \qquad (4\text{-}1)$$

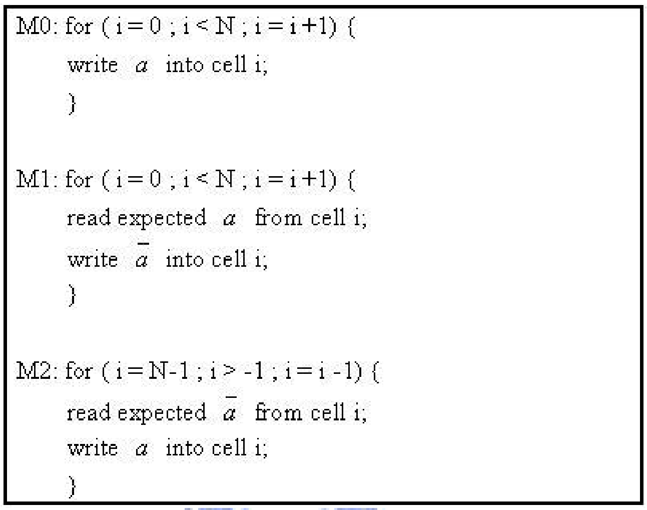
Where *D* is the number of detected faults and *T* is the total number of faults in the fault list. For design verification, the fault coverage can not only give the designer a rough quantitative measure of how well the design has been exercised, but also the undetected fault list can provide valuable information on those sub-circuits that have not been exercised as thoroughly as other sub-circuits.

Although example will be used in which the $\Uparrow$ address order goes from address 0, 1, 2. . . *n*-2 to *n*-1, this is not strictly necessary. It is necessary that the address-orders $\Uparrow$ and $\Downarrow$ are each other's invert. For instance, when the address-orders $\Uparrow$ is chosen for some reason to be: 1, 0, 7, 5, 6 ,4 ,2 ,3 ; the address order $\Downarrow$ has to be:3, 2, 4, 6, 5, 7, 0, 1.This means that the march test $\{\Uparrow(r1, w0); \Downarrow(r0, w1)\}$ has the same fault coverage as the test $\{\Downarrow(r1, w0); \Uparrow(r0, w1)\}$.

In Table 4.2, we show some popular March algorithms. And we also show the fault coverage in Table 4.3.

$$\underbrace{\Uparrow(wa);\quad \Uparrow(\overbrace{ra}^{\text{read operation}}, w\bar{a});\quad \overbrace{\Downarrow(r\bar{a}, wa)}^{\text{March element: M2}}}_{\text{March test}}$$

**Figure 4.2** A March algorithm example



```
M0: for ( i = 0 ; i < N ; i = i +1) {
      write  $a$  into cell i;
      }

M1: for ( i = 0 ; i < N ; i = i +1) {
      read expected  $a$  from cell i;
      write  $\bar{a}$  into cell i;
      }

M2: for ( i = N-1 ; i > -1 ; i = i -1) {
      read expected  $\bar{a}$  from cell i;
      write  $a$  into cell i;
      }
```

**Figure 4.3** The procedure for the March algorithm example

| Name<br>Algorithm | Element | Faults Detected |
|---|---|---|
| MATS++ | $\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0, r0)$ | SAF/AF |
| March X | $\Updownarrow(w0); \Uparrow(r0, w1); \Downarrow(r1, w0) \Updownarrow(r0)$ | AF/SAF/TF/CFin |
| March Y | $\Updownarrow(w0); \Uparrow(r0, w1, r1); \Downarrow(r1, w0, r0) \Updownarrow(r0)$ | AF/SAF/TF/CFin |
| March C- | $\{\Updownarrow(w0); \Uparrow(r0, w1); \Uparrow(r1, w0); \Downarrow(r0, w1); \Downarrow(r1, w0); \Updownarrow(r0)\}$ | SAF/AF/TF/CF |

**Table 4.2** Some March algorithms

| Fault | MATS++ | March X | March Y | March C- |
|---|---|---|---|---|
| SAF | 100% | 100% | 100% | 100% |
| TF | 100% | 100% | 100% | 100% |
| SOF | 100% | 0.2% | 100% | 0.2% |
| AF | 100% | 100% | 100% | 100% |
| CFin | 75.0% | 100% | 100% | 100% |
| CFid | 37.5% | 50.0% | 50.0% | 100% |
| CFst | 50.0% | 62.5% | 62.5% | 100% |

**Table 4.3** Fault coverage of some popular March algorithms

## 4.3 A Test Algorithm for Sandwich Ping-Pong Memory

### 4.3.1 March C- Algorithm

March C- algorithm in Figure 4.4 satisfies the conditions of detecting simple (unlinked) faults such as SAFs, TFs, CFs, AFs, and SOFs [5]. This section shows a modified March C- algorithm in Figure 4.5 which is derived from March C- and proofs fault detection capabilities.

$$\begin{array}{cccccc} \{\Updownarrow (w0); & \Uparrow (r0, w1); & \Uparrow (r1, w0); & \Downarrow (r0, w1); & \Downarrow (r1, w0); & \Updownarrow (r0)\} \\ \text{M0} & \text{M1} & \text{M2} & \text{M3} & \text{M4} & \text{M5} \end{array}$$

**Figure 4.4** March C- algorithm

| Condition | March element |
|---|---|
| 1 | $\Uparrow(rx,\ldots,w\overline{x})$ |
| 2 | $\Downarrow(r\overline{x},\ldots,wx)$ |

**Table 4.4** Conditions for detecting address decoder faults

March C- satisfies the Conditions 1 and 2 for Address Faults in Table 4.4 [2]. When $x$ =0 by means of March elements M2 and M5, when $x$ =1 by means of March elements M3 and M4. March C- will detect SAFs and TFs because all cells are read in states 0, 1, 0 … Thus, both $\uparrow$ and $\downarrow$ transitions, and read operations after them, have taken place. March C- will also detect idempotent and inversion coupling faults, with the restriction that these coupling faults are unlinked.

A fault is linked when that fault may influence the behavior of other faults. A fault is unlinked when that fault does not influence the behavior of other faults. Here is an example, as shown below.

**Example**

Suppose that there are two coupling faults in a memory, as shown in Figure 4.5. The first fault is that cell $i$ is $<\uparrow;1>$ coupled to cell $j$; the second fault is that cell $k$ is coupled to cell $l$. The March test $\{\Updownarrow(w0);\Uparrow(r0,w1);\Updownarrow(w0,w1);\Updownarrow(r1)\}$ will detect both faults if $i \neq k$ (Figure 4.5(a)). The $<\uparrow;1>$ CF will be detected by the 'r0' operation of March element, when operating on cell $i$. The $<\uparrow;0>$ CF will be detected by the 'r1' operation of the last March element, when it operates on cell $k$.

However, this test will not detect the combination of the faults which occurs when $i \equiv k$ (Figure 4.5(b)). The 'link' between the faults (in this case the effect that the coupled cells are the same) can cause the test not to find any fault; this effect is called **masking**. The 'r0' operation of the march element $\Uparrow(r0, w1)$ will not detect the linked CF because when operation on cell $i$ the cell will contain a 0 value due to the $<\uparrow;0>$ CF. The 'r1' operation of the last march element will not detect the linked CF because, when operating on cell $i$, it will contain a 1 value due to the $<\uparrow;1>$ CF sensitized by the march element $\Updownarrow(w0, w1)$ when it operates on cell $j$.

The following test has been designed to detect the faults of Figures 4.5(a) and 4.5(b): $\{\Updownarrow(w0);\Uparrow(r0,w1);\Updownarrow(w0,w1);\Updownarrow(r1,w0,w1)\}$. The $<\uparrow;1>$ CF of Figure 4.5(a) will be detected by the 'r0' operation of March element $\Uparrow(r0, w1)$ when it operates on cell $i$; the CF of Figure 4.5(a) will be detected by the 'r1' operation of the last March element when it operates on cell $k$. The linked fault of Figure 4.5(b) will be detected by the 'r1' operation of the last March element when it operates on cell $i$.

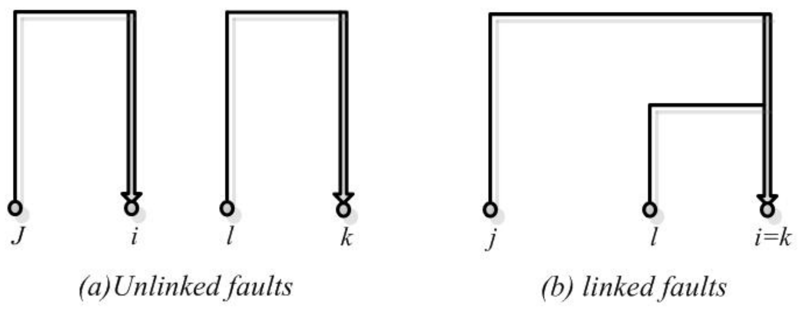


**Figure 4.5** Masking of coupling faults

The proof that March C- is complete is given below:

- AFs are detected because the conditions of in Table 4.3.
- SAF1 faults are detected by the read operations of M1, M2, M4, and M6.
- SAF0 faults are detected bye the read operations of M3, M5, and M7.

- Unlinked $<\uparrow/0>$ TFs are detected by M1 followed by M2 or by M3 followed by M4.
- Unlinked $<\downarrow/1>$ TFs are detected by M2 followed by M3 or by M4 followed by M5.
- Unlinked CFins $<\uparrow;\uparrow>$ are detected by M3 followed by M4; CFins $<\downarrow;\uparrow>$ are detected by M4 followed by M5.
- Unlinked CFids $<\uparrow;0>$ are detected by M3 followed by M4; CFids $<\uparrow;1>$ are detected by M1; CFids $<\downarrow;0>$ are detected by M2; CFids $<\downarrow;1>$ are detected by M4 followed by M5.

### 4.3.2 The Modified March C- Algorithm

There are two memory arrays which are called Ping memory (or Block A) and Pong memory (Block B) in a Ping Pong memory, introduced in Section 3.1.1. We have derived an algorithm for testing Ping memory and Pong memory simultaneously. We named this algorithm the Modified March C- shown in figure 4.5. In addition, the modified March C- is depicted carefully in figure 4.6, the upper sequence is testing the Block A memory array, meanwhile, the lower one is testing the Block B memory array.

There are two more March elements (M1 and M6) in this algorithm than in March C- algorithm. However, the fault coverage of this algorithm is the same as the one of March C- algorithm. Because these two March elements (M1 and M5) are *nop (no operation)* operations needed to be inserted into the March algorithm. As a result, the total operations increase.

There are eight operations within the algorithm, the test length of the modified March C- is eight, i.e. total 8*N* read and write operations are need to apply the algorithm (N is the memory size).

The modified March C- will detect *CFins* and *CFids* as shown below.

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

M0 M1 M2 M3 M4 M5 M6 M7

**Figure 4.6** Modified March C- algorithm

$$A:\{(w_0^0,....,w_0^{N-1});(r_0^0,...,r_0^{N-1});(r_0^0,\ w_1^0,...,r_0^{N-1},w_1^{N-1});(r_1^0,\quad w_0^0,r_1^1,w_0^1...,r_1^{N-1},w_0^{N-1});(r_0^{N-1},$$
$$B:\{(r_?^0,....,r_?^{N-1});(w_0^0,...,w_0^{N-1},w_0^0);(r_0^0,w_1^0,...,r_0^{N-1},\quad w_1^{N-1});(r_1^0,w_0^0,r_1^1,.\ .\ ..,r_1^{N-1},w_0^{N-1});$$
$$w_1^{N-1},\dots,r_0^0,w_1^0);(r_1^{N-1},\ w_0^{N-1},...,r_1^0,w_0^0);(r_0^0,r_0^1,....,r_0^{N-1});(w_0^0,...,w_0^{N-1})\}$$
$$(r_0^{N-1},w_1^{N-1},...,r_0^0,w_1^0);(r_1^{N-1},w_0^{N-1},...,r_1^0,w_0^0,w_0^0,...,w_0^0);\ \ (r_0^0,....,r_0^{N-1})\}$$

**Figure 4.7** Modified March C- algorithm for block A and block B memory arrays

### 4.3.3 Testing of Sandwich Ping-Pong Memory

We will present how to test the Sandwich Ping-Pong Memory in this section. It is the modified March C- algorithm that we use for testing because it shows the high fault coverage. It can test the most popular the faults such as: SAFs, TFs, AFs, CFins, CFids, and CFsts and its fault coverage was already presented in table 4.2.

When testing the Sandwich Ping-Pong Memory, the basic and simple way is to separate it into two parts: one is ping-pong memory and another is single port memory, in figure 4.8. There is no doubt that the modified March C- algorithm can test not only the Ping Pong memory but also the Common Bar.

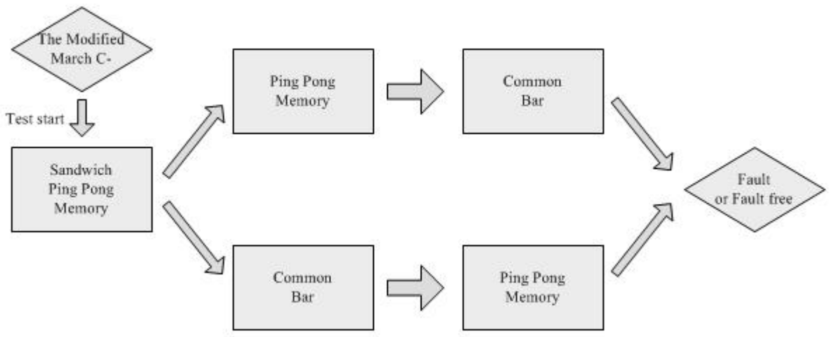


**Figure 4.8** Test flow

Modified March C- satisfies the conditions 1 and 2 for AFs of Table 4.3: when $x = 0$ by means of March elements M2 and M5, when $x = 1$ by means of March elements M3 and M4. Modified March C- will detect SAFs and unlinked TFs because all cells are read in states 0, 1, 0, . .. . Thus, both $\uparrow$ and $\downarrow$ transitions, and read operations after them, have taken place. Modified March C- will also detect idempotent and inversion coupling faults, with the restriction that these coupling fault are unlinked. The detection capabilities for idempotent and inversion coupling faults are proved below.

## Idempotent coupling faults

The proof that the modified March C- detects CFids is split into two cases: 1. faults with the addresses of the coupling cells lower than the coupled cell, and 2. faults with the address of the coupling cells higher than the coupled cell. The coupled cell will be denoted by $C_i$, and (one of) the coupling cells with $C_j$. As a reminder of the notation: $C_i$ is coupled to $C_j$ means that an $\uparrow$ transition in $C_j$ causes a o value in $C_i$.

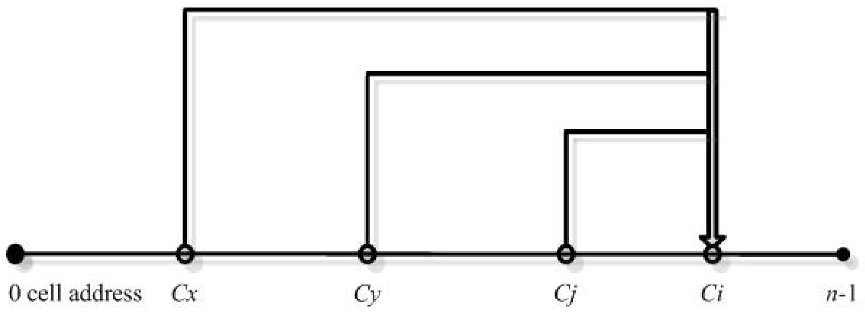


**Figure 4.9** Addresses of coupling and coupled cells

**Case 1:**

Let $C_i$ be coupled to any number of cells with address lower than *i*, and let $C_j$ be the highest of those cells (*j*<*i*), see figure 4.8. Four cases, corresponding to the four different types of CFids, can be distinguished: a. $<\uparrow;0>$, b. $<\uparrow;1>$, c. $<\downarrow;0>$, and d. $<\downarrow;1>$. These cases are proved below.

(a) If $C_i$ is $<\uparrow;0>$ coupled to $C_j$, then the fault will be detected in march element M4 followed by M5. See in figure 4.10 (a).

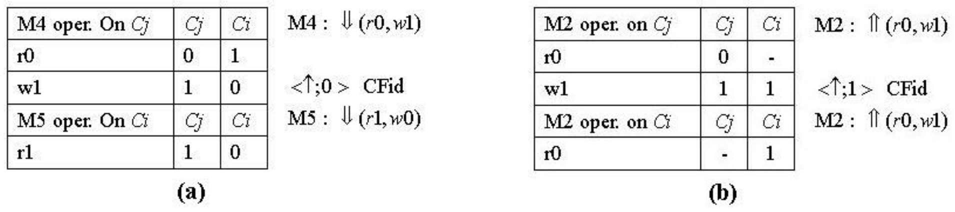
- In M4 a 1 is written in $C_j$ and due to the $<\uparrow;0>$ coupling fault $C_i$ will contain a 0.
- In M5 a read operation is performed on $C_i$ and a 0 instead of a 1 is read.
- Linked CFids will not be detected. For example, $C_i$ must not be $<\uparrow;1>$ coupled to cells with a lower address than j, because M4 operates on them after $C_j$. In that case a 1 would be read in M5 which is the expected value. Thus, the fault would not be detected.

(b) If $C_i$ is $<\uparrow;1>$ coupled to $C_j$ then the March element M2 will detect the fault. See in figure 4.10 (b).

- First a 0 will be read from $C_j$, then a 1 will be written. Due to the $<\uparrow;1>$ coupling fault $C_i$ will be forced to contain a 1.
- A 1 instead of a 0 is read when M2 operates on $C_i$.

(c) If $C_i$ is $<\downarrow;0>$ coupled to $C_j$ then he march element M3 will detect the fault. The proof is similar to above.

(d) If $C_i$ is $<\downarrow;1>$ coupled to $C_j$ then the March elements M5 followed by M6 will detect the fault. $C_i$ must not be $<\downarrow;1>$ coupled to cells with addresses lower than *j*. The proof is similar to above.



| M4 oper. On $Cj$ | $Cj$ | $Ci$ |
|---|---|---|
| r0 | 0 | 1 |
| w1 | 1 | 0 |
| M5 oper. On $Ci$ | $Cj$ | $Ci$ |
| r1 | 1 | 0 |

M4 : $\Downarrow(r0,w1)$

$<\uparrow;0>$ CFid

M5 : $\Downarrow(r1,w0)$

(a)

| M2 oper. on $Cj$ | $Cj$ | $Ci$ |
|---|---|---|
| r0 | 0 | - |
| w1 | 1 | 1 |
| M2 oper. on $Ci$ | $Cj$ | $Ci$ |
| r0 | - | 1 |

M2 : $\Uparrow(r0,w1)$

$<\uparrow;1>$ CFid

M2 : $\Uparrow(r0,w1)$

(b)

**Figure 4.10** Detecting CFids

**Case 2:**

Let $C_i$ be coupled to any number of cells with addresses higher than *i* and let $C_j$ be the lowest addressed cell of them (*j*>*i*). The proof is similar to Case1, whereby M2 should be replaced by M4, M3 by M5, M4 by M2, M5 by M3 and M6 by M4.

Above it has been shown that modified March C- will detect CFids. The CFids must be unlinked, because not all combinations of faults are allowed, as proved above.

## Inversion coupling faults

**Case 1:**

Let $C_i$ be coupled to any number of cells with addresses lower than *i* and let $C_j$ be the highest of those cells ($j<i$). Then, there are two cases, corresponding to eh two different types of CFins: $<\uparrow;\updownarrow>$ and $<\downarrow;\updownarrow>$. These cases are proven below.

(a) $C_i$ is $<\uparrow;\updownarrow>$ coupled to $C_j$; then M1 will detect the fault, as well as M4 followed by M4.

- M4 operates on $C_j$, making an $\uparrow$ transition and inverting the contents of $C_i$; when M5 is operated on $C_i$ a 0 instead of a 1 will be read.

| M4 oper. on *Cj* | *Cj* | *Ci* |
|---|---|---|
| r0 | 0 | 1 |
| w1 | 1 | 0 |
| M5 oper. on *Ci* | *Cj* | *Ci* |
| r1 | - | 0 |

M4 : $\Downarrow(r0, w1)$

$<\uparrow;\updownarrow>$ CFin

M5 : $\Downarrow(r1, w0)$

**Figure 4.11** Detecting CFins

(b) $C_i$ is $<\downarrow;\updownarrow>$ coupled to $C_j$; then M3 as well as M4 followed by M6 will detect the fault. The proof is similar to above.

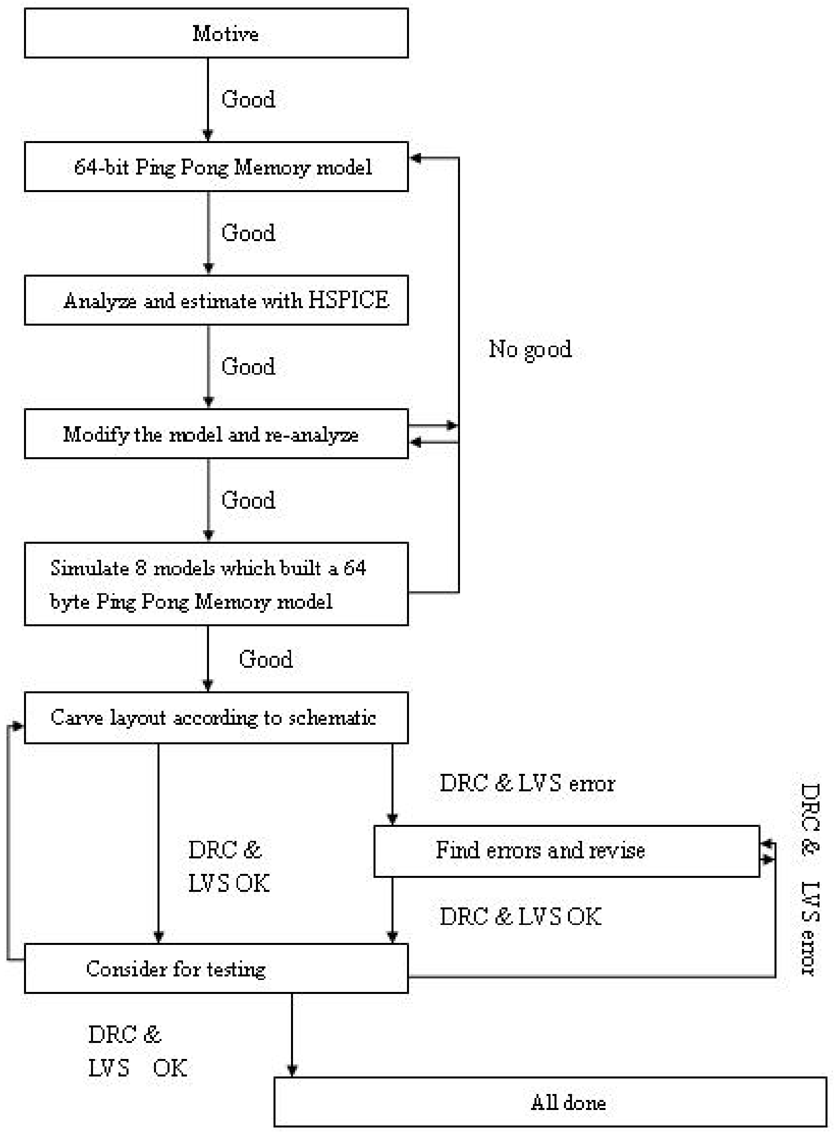
**Case 2:**

The proof for $j>i$ is similar to above.

As a result, we can use the modified March C- algorithm to test the Sandwich Ping Pong Memory. The fault coverage is at 100% for the stuck-at fault, transition fault, address fault, and coupling fault.

# Chapter 5 Chip Implementation

We implement our Ping Pong memory in Full-Custom Design by specifying the layout of each individual transistor and the interconnections. Figure 5.1 shows the design flow for 64-byte Ping Pong Memory.



Motive
Good
64-bit Ping Pong Memory model
Good
Analyze and estimate with HSPICE
Good
No good
Modify the model and re-analyze
Good
Simulate 8 models which built a 64 byte Ping Pong Memory model
Good
Carve layout according to schematic
DRC & LVS error
DRC & LVS OK
Find errors and revise
DRC & LVS error
DRC & LVS OK
Consider for testing
DRC & LVS OK
All done

**Figure 5.1** Design flow for 64-byte Pig Pong Memory

In order to be more flexible, we also complete the part of the Sandwich Ping Pong Memory, Common Bar, and the test circuit by means of HDL (Hardware Description Language) which could synthesize the circuit we want. We show the results on Sparan-3 Starter board which provides a powerful, self-contained development platform. It features a 200K gate Sparten-3, on-board I/O devices, and two large memory chips.

## 5.1 Chip Implementation of the Sandwich Ping Pong Memory

### 5.1.1 The architecture of a Ping-Pong buffer

We have presented a double buffer roughly and realized that its application is in the 2-D DCT architecture in chapter 2. In this section, we are going to introduce the architecture of a double buffer, also known as, Ping-Pong buffer which is shown in figure 5.2.

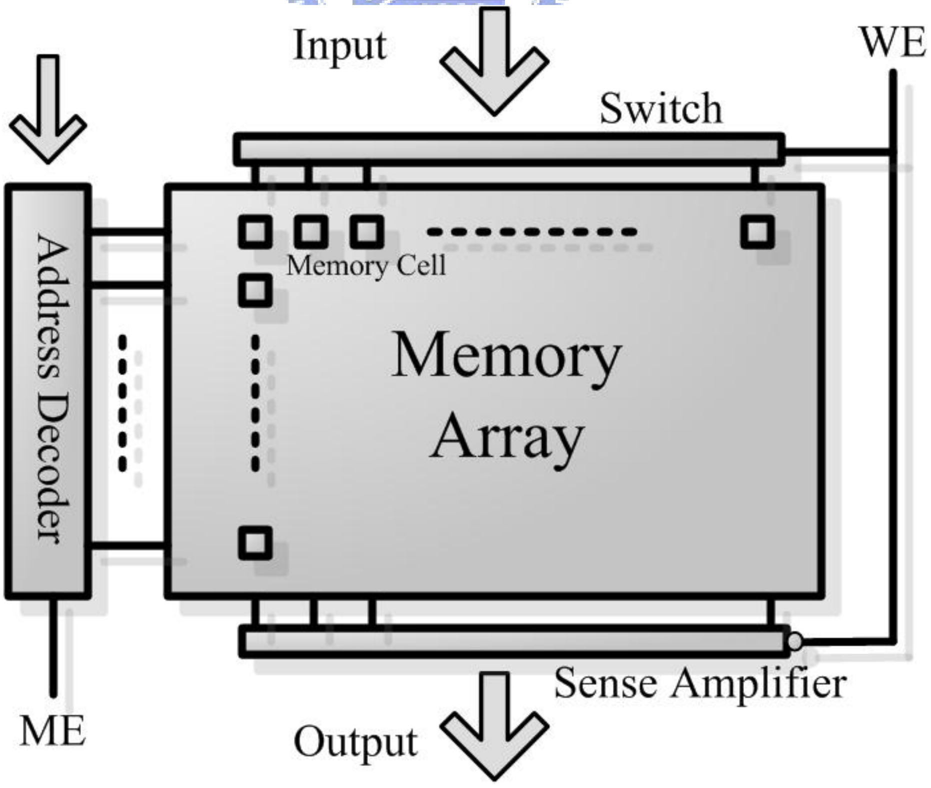


**Figure 5.2** Components in the Ping-Pong Memory

## 5.1.2 The detail design of a Ping-Pong buffer

A Ping-Pong buffer consists of the following kinds of component:

**Memory cell**

From the begging of introducing a Ping-Pong buffer, we talk about the basic component – the memory cell [27]. A memory cell can be implemented as SRAM, DRAM, and Flash. We take a six-device SRAM memory cell, also called CMOS SRAM cell shown in figure 5.3, as our memory cell. The load devices are PMOS enhancement mode transistors Q2 and Q4. Comparing to the depletion mode NMOS as the load devices, this further reduces the power requirements of the cell; except for some small leakage current, no power will be dissipated during the time the cell retains the stored logic value. The transistors Q5 and Q6 work as switches and are named access transistors. When the gate of Q5 and Q6 are activated, they are turned-on and the memory cell work.

The operations of a memory cell are the following steps. When some word line is chosen, the access transistors are on and connect the two bit lines $Q$ and $\overline{Q}$. At this moment, we can write/read some data into/from the memory cell. On the operation of writing, it is easy to understand that the data (the voltage on bit lines) can be stored and latched between two back-to-back CMOS. On the opposite, the operation of reading, we turn down the voltage of word line after the data (voltage on the gate of CMOS) is passed to the bit lines from the memory cell. Finally, the data (voltage on the bit lines) is forward to the next component.

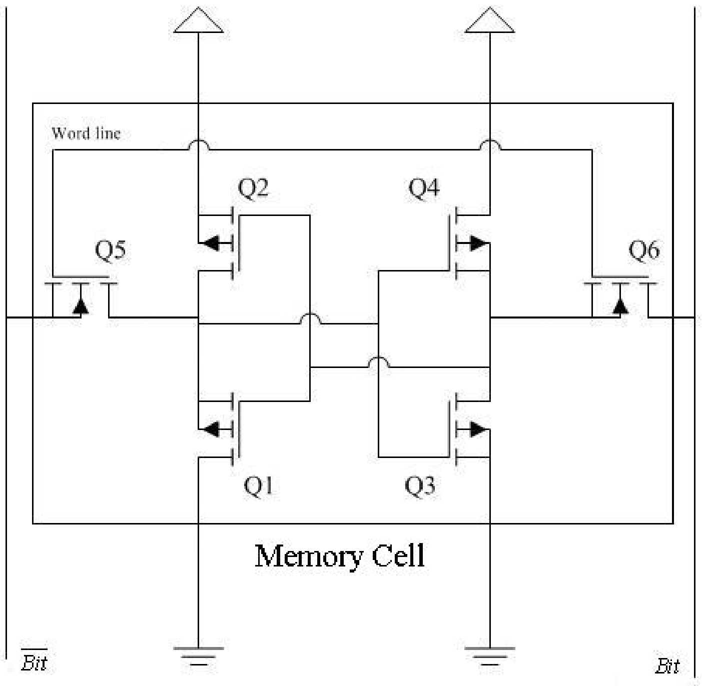


**Figure 5.3** Six-device SRAM memory cell

**Sense Amplifier, Equalization, and Pre-charge circuit**

When the data (voltage) is on the bit lines, they will bump into the next component - Sense Amplifiers. Memory cells are composed by CMOS, so are the sense amplifiers. In figure 5.4, we show the differential sense Amplifier, equalization, and pre-charge circuit.

The sense amplifier is the most critical component in a memory chip. Sense amplifiers are essential to the proper operation of Drams, and their use in Scrams results in speed and area improvements. Here, we describe a differential sense amplifier that employs positive feedback. Because the circuit is differential, it can be employed directly in Scrams where the SRAM cell utilizes both the $B$ and $\overline{B}$ lines. We assume that the memory cell whose output is to be amplified develops a difference output voltage between the $B$ and $\overline{B}$ lines. This signal, which can range between 30 mV and 500 mV depending on the memory type and cell design, will be applied to the input terminals of the sense amplifier. The sense amplifier in turn responds by providing a full logic-swing (0 to $V_{DD}$) signal at its output terminals. The particular amplifier circuit we shall discuss has a rather unusual property: *Its output and input terminals are the same!*

The sense amplifier is nothing but the familiar latch formed by cross-coupled two CMOS inverters: one inverter is implemented by transistors Q1 and Q2, and the other by transistors Q3 and Q4. The transistors Q5 and Q6 work as switches that connect the sense amplifier to ground and $V_{DD}$ only when data-sensing action is required. Otherwise, $\phi_s$ is low and the sense amplifier is turned off. This conserves power, an important consideration since usually there is one sense amplifier per column, resulting in thousands of sense amplifiers per chip. Note again that terminals x and y are both the input and the output terminals of the amplifier. As indicated, these I/O terminals are connected to the $B$ and $\overline{B}$ lines. The amplifier is required to detect a small signal appearing between $B$ and $\overline{B}$, and to amplify it to provide a full-swing signal at $B$ and $\overline{B}$. For instance, if during a read operation, the cell had a stored 1, then a small positive voltage will develop between $B$ and $\overline{B}$, with $V_B$ higher than $V_{\overline{B}}$. The amplifier will then cause $V_B$ to rise to $V_{DD}$ and $V_B$ to fall to 0 V. This 1 output is then directed to the chip I/O pin by the column decoder and at the same time is used to rewrite a 1 in the memory cell.

Figure 5.4 shows the pre-charge and equalization circuit. Operation for this circuit is straightforward: When $\phi_s$ goes high prior to a read operation, all three transistors conduct. While Q8 and Q9 pre-charge the $\overline{B}$ and $B$ lines to $V_{DD}$ /2, transistor Q7 helps speed up this process by equalizing the voltages on the two lines. This equalization is critical to the proper operation of the sense amplifier: Any voltage difference present between $B$ and $\overline{B}$ prior to commencement of the read operation can result in erroneous interpretation by the sense amplifier of its input signal. We show only one of the cells in this particular column namely, the cell whose word line is activated. The cell can be either an SRAM or a DRAM cell. All other cells in this column will not be connected to the $B$ and $\overline{B}$ lines.

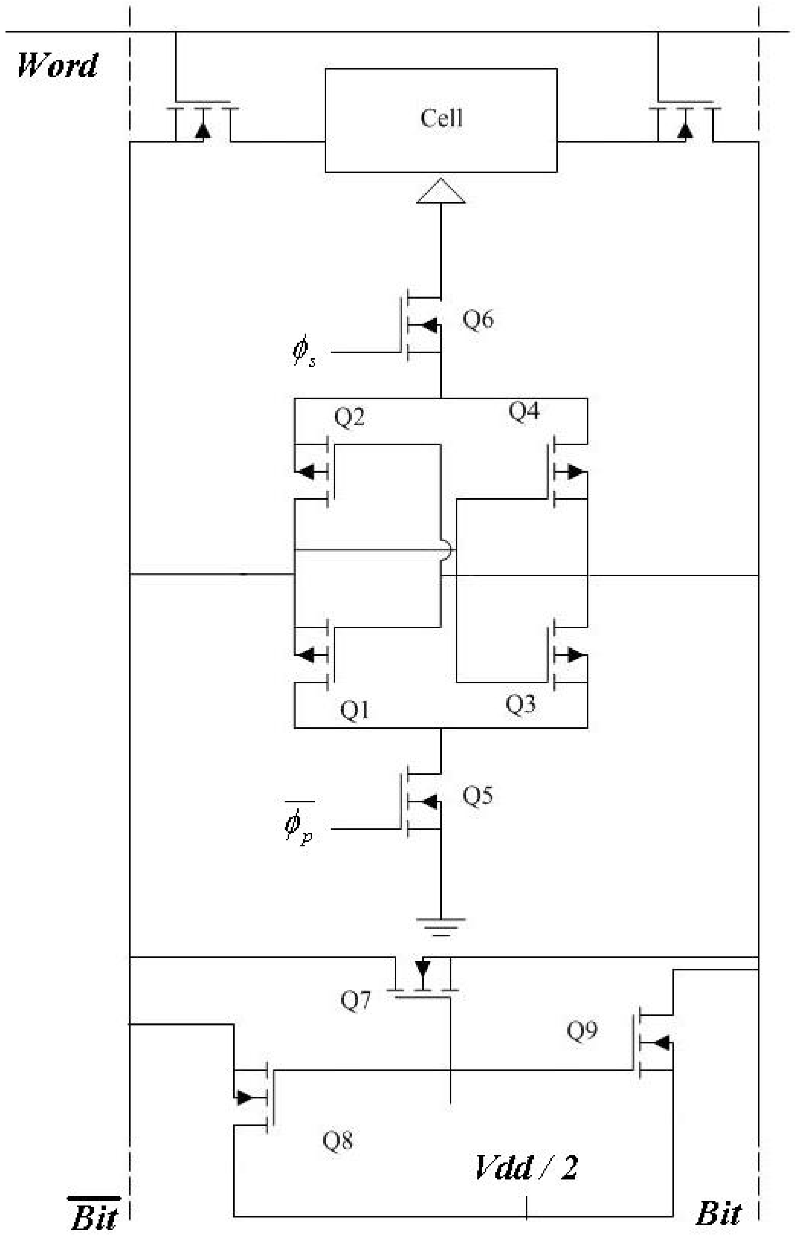


**Figure 5.4** The differential SA, Equalization, and Pre-charge circuit

### Address Decoder

Address decoders are composed by 4-intput NAND gates and the circuit is in figure 5.5. We design a 3-input NAND gate for a address decoder to control 8 different word lines of a memory array in the beginning. In order to control the specific memory array, we put one more NAND input for a address decoders as the memory enable (ME) signal. The reason that we design address decoder by NAND gates is to speed up the circuit with less layout area. So we can build up a 64 bytes Ping-Pong memory in 1310 X 1100 micro meter squared by using 0.35um process.

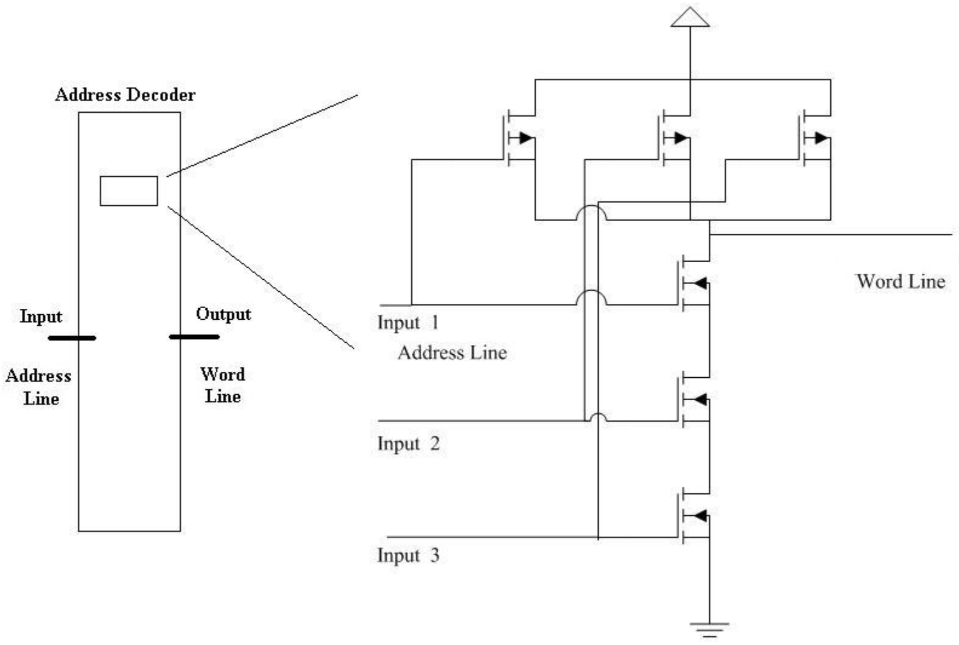


**Figure 5.5** Using NAND gates to build up address decoders

### 5.1.3 HSPICE Simulation of a Ping-Pong Memory

Here we show the design flow, shown in figure 5.6, and explain the HSPICE simulation result of Ping-pong memory. During the signal of read enable is off, the memory array is on write operation. The data is written into specific memory cells one by one. After 80ns, 8 bits data in one word line is finished the operation of writing. On other hand, during the signal of read enable is on, the memory array is on read operation. The data is read from specific memory cells gradually. Of course after 80ns, one byte in one word line is totally read. Therefore, the read and write operation is supposed be completed in the period of 160ns. However, we put a signal of “Memory Enable” (ME) to control the different memory array. The period of read cycle is not 160ns. It is the memory enable cycle that 160ns belongs to it. Therefore, the period of read cycle is 320ns. In figure 5.7, due to the access time of a specific memory cell is 10ns and we have 8 bits data on one word line, the period of read (write) enable and memory enable is 320ns and 160ns respectively.

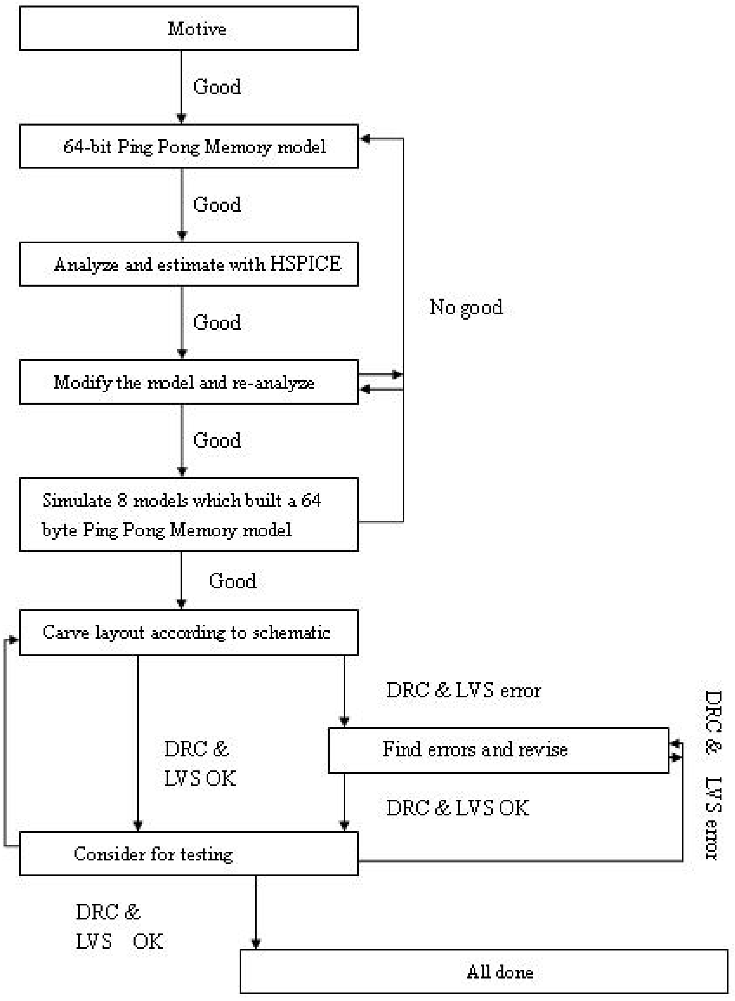

Motive
Good
64-bit Ping Pong Memory model
Good
Analyze and estimate with HSPICE
Good
No good
Modify the model and re-analyze
Good
Simulate 8 models which built a 64 byte Ping Pong Memory model
Good
Carve layout according to schematic
DRC & LVS error
DRC & LVS OK
Find errors and revise
DRC & LVS OK
DRC & LVS error
Consider for testing
DRC & LVS OK
All done

**Figure 5.6** Design flow

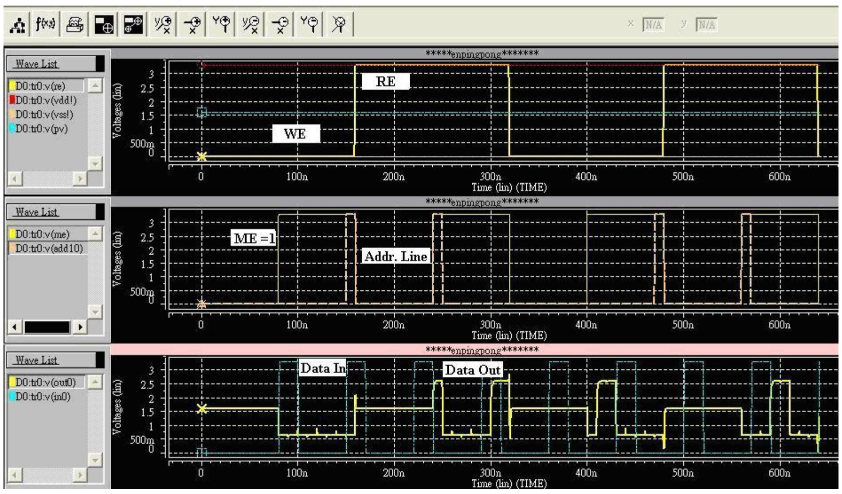


**Figure 5.7** HSPICE simulation of Ping-pong memory

## 5.2 Layout of the Ping Pong Memory

The Ping Pong Memory is implemented on TSMC 0.35um 2p4m 3.3v process by full-custom design [28] ~ [33]. In figure 5.8, we show the layout of an 8-bytes Ping-Pong buffer. To save more area, the layout is utilized by Common-Centriod Layout illustrated in figure 5.9. Hence, the total layout area with pads is 1380 x 1095 $\mu m^2 \mu m$. The difference between figure 5.9 and figure 5.10 is the global decoder. We turn the 64-bit memory into a 64-byte memory because we want to make a larger memory and make the most of the area in this chip. The global decoder controls the signal of the memory-enable (me). When memory-enable is 1, the specific memory array is on. We can turn on the eight memory array in turns by means of the global decoder.

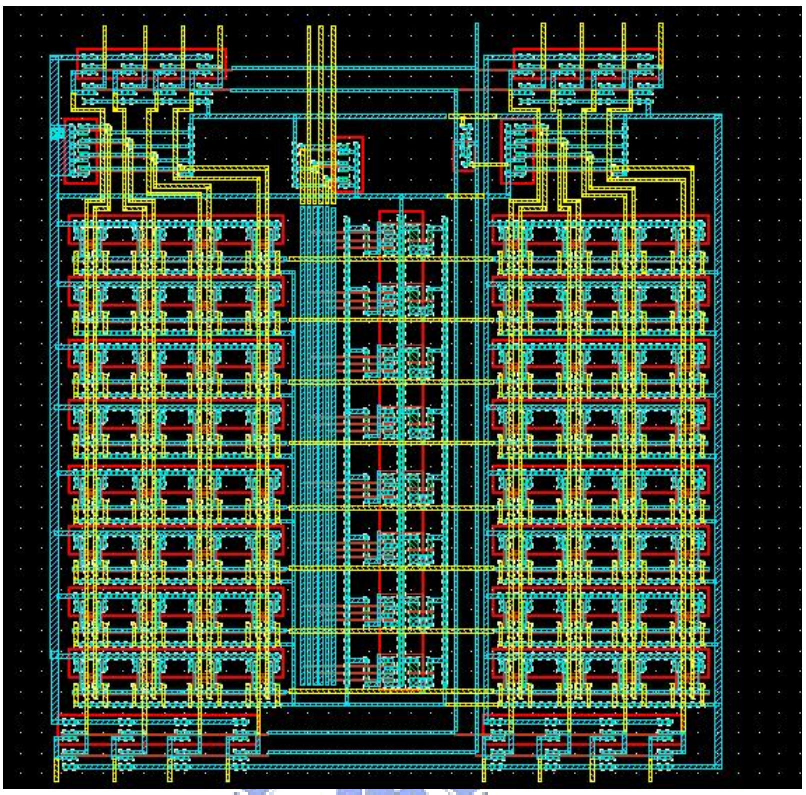


**Figure 5.8** Layout of an 8-byte Ping-Pong buffer.

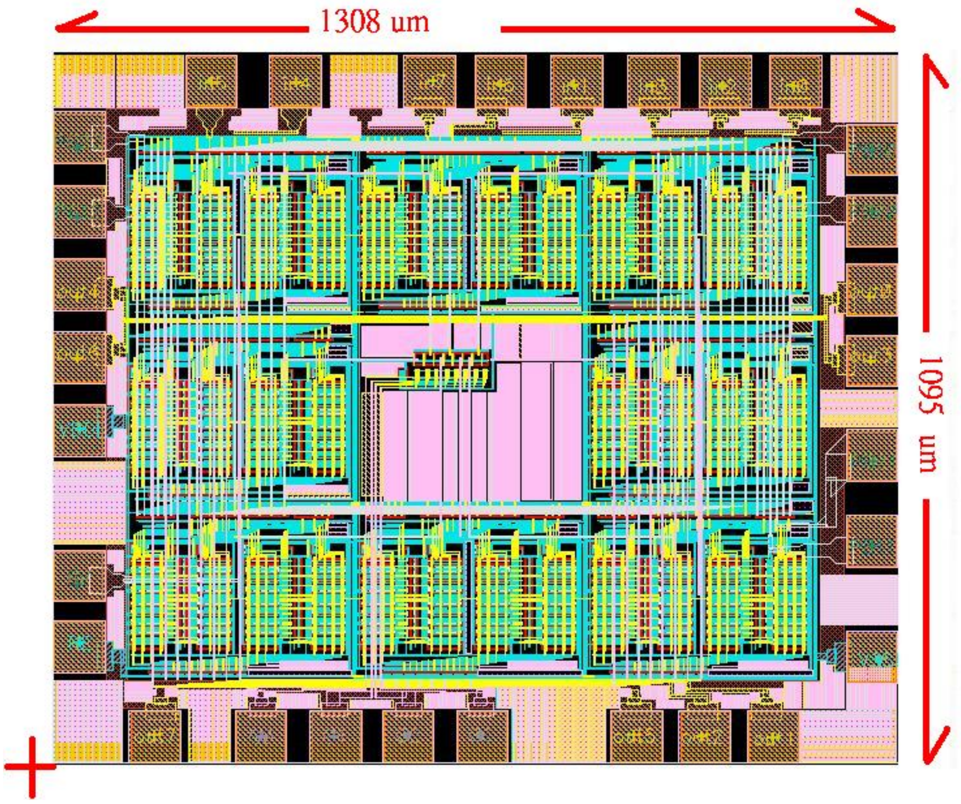


**Figure 5.9** Layout of 64-byte Ping Pong Memory with Pad

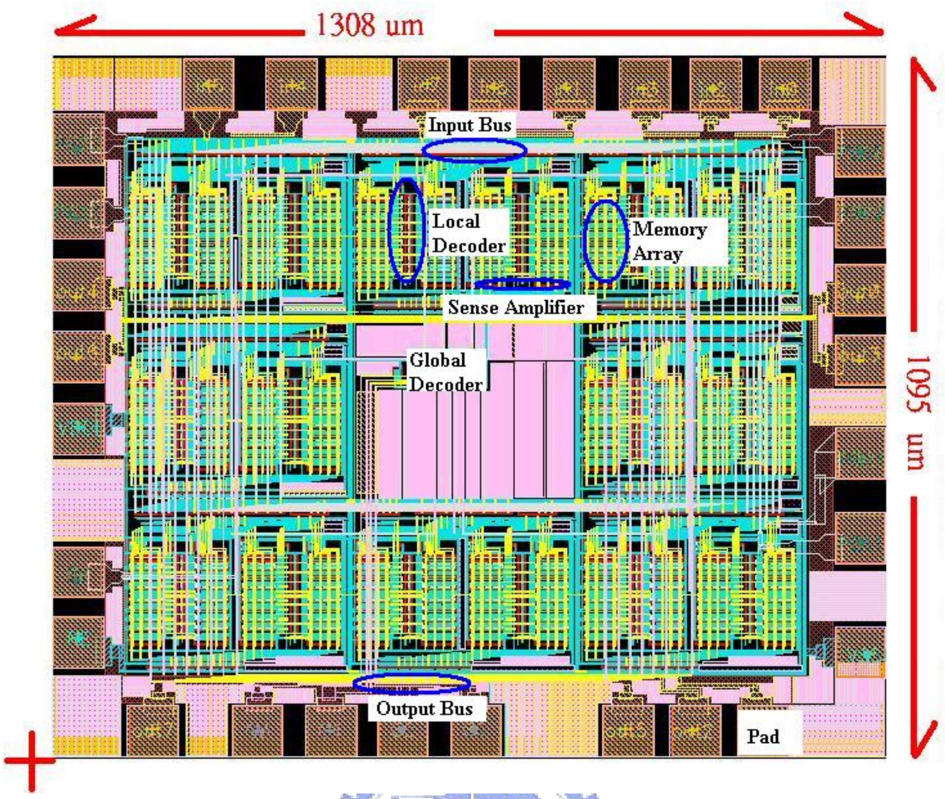


**Figure 5.10** Component introduction on layout

The pins definition is in table 5.1 and the specification is in table 5.2. Figure 5.11 shows the bonding of the chip. Figure 5.12 shows the photography of the die.

| Architecture | Ping-Pong SRAM Memory |
|---|---|
| Supply voltage(V) | 3.3V |
| MAX Frequency(Hz) | 100MHz |
| Power | 27mW |
| Technology | TSMC 0.35 2P4M |
| Core Size | 1070*840 $\mu m^2$ |
| Total Chip Area | 1310*1100 $\mu m^2$ |
| Total Transistors | 9830 |

**Table 5.1** Specification

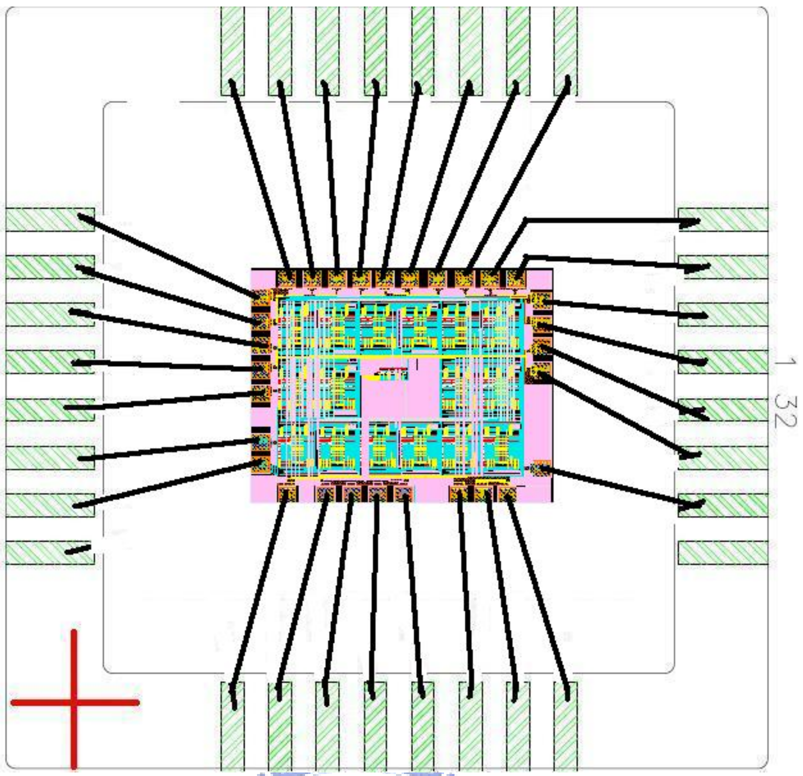


**Figure 5.11** Bonding figure

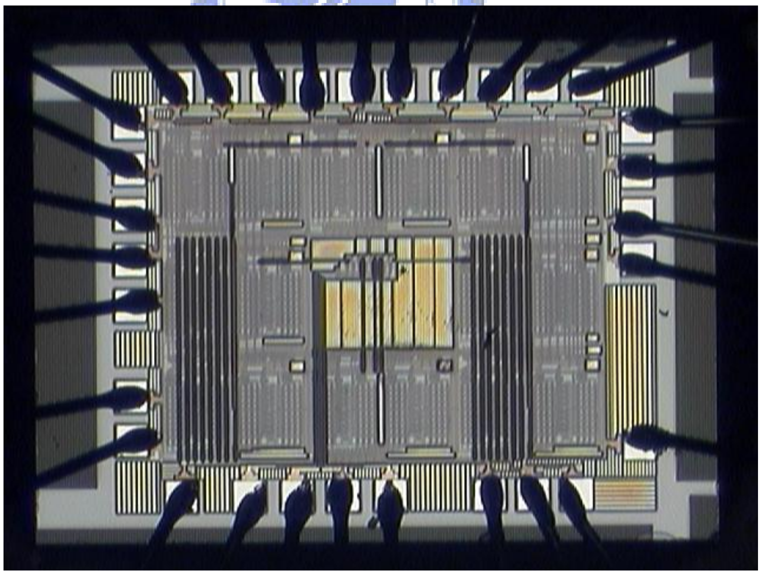


**Figure 5.12** Photo of the die

| Pin Number | Pin Name | Function Description |
|---|---|---|
| 1 | Out[3] | 4th output data |
| 2 | Out[0] | 1st output data |
| 3 | PE[20] | Signal 20 for local decoder |
| 4 | PE[10] | Signal 10 for local decoder |
| 5 | In[0] | 1st input data |
| 6 | In[2] | 3rd input data |
| 7 | In[3] | 4th input data |
| 8 | In[1] | 2nd input data |
| 9 | In[5] | 6th input data |
| 10 | In[7] | 8th input data |
| 11 | In[4] | 5th input data |
| 12 | In[6] | 7th input data |
| 13 | PE[22] | Signal 22 for local decoder |
| 14 | PE[12] | Signal 12 for local decoder |
| 15 | Out[4] | 5th output data |
| 16 | Out[6] | 7th output data |
| 17 | VDD! | Power line |
| 18 | N/A | N/A |
| 19 | PV | Pre-charge line |
| 20 | WE | Write Enable |
| 21 | Out[7] | 8th output data |
| 22 | G[0] | Signal 0 for global decoder |
| 23 | G[1] | Signal 1 for global decoder |
| 24 | G[2] | Signal 2 for global decoder |
| 25 | G[3] | Signal 3 for global decoder |
| 26 | Out[5] | Signal 5 for global decoder |
| 27 | Out[2] | Signal 2 for global decoder |
| 28 | Out[1] | Signal 1 for global decoder |
| 29 | N/A | N/A |
| 30 | VSS! | Ground line |
| 31 | PE[11] | Signal 11 for local decoder |
| 32 | PE[21] | Signal 21 for local decoder |

**Table 5.2** Pins definition

## 5.3 Test Circuit Design

In this section, we are going to implement the test circuit. After finishing the layout of the Ping Pong Memory, we consider about the test circuit to make sure the chip meet our specifications. The functional test based upon the reduced functional fault models that were discussed in chapter 4, such as the stuck-at, transition, and coupling faults.

The test circuit includes two essential functions as well as two additional functions that are necessary to facilitate execution of the testing feature. The two essential functions illustrated in figure 5.13 include the test pattern generator (TPG) and output response analyzer (ORA). While the TPG produces a sequence of patterns for testing the circuit under test (CUT), the ORA compacts the output responses of the CUT into some type of *Pass/Fail* indication.

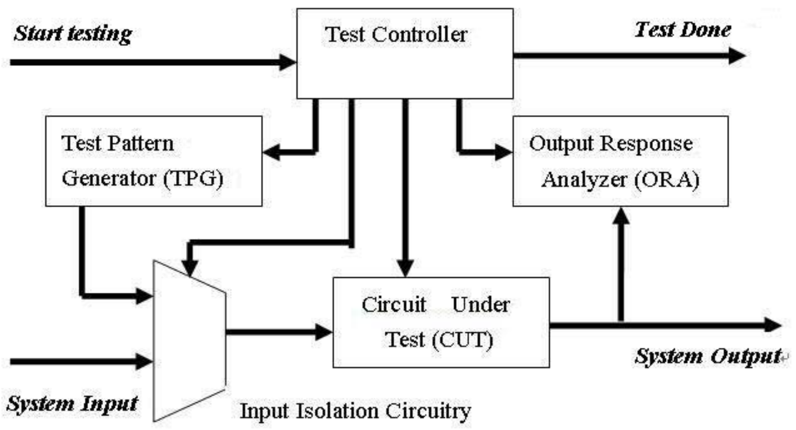


**Figure 5.13** Test architecture

The TPG circuit illustrated in figure in 5.14 consists of Counters, Algorithm and Counter Sets circuit. The Counter circuit outputs a sequence number call 'Qin' as a primary time unit. Qin sequence is passed to the Algorithm and the ORA circuit. The Algorithm circuit outputs control signal to the Counter Sets circuit and the signals of write enable (WE) and Common

Bar write enable (CBWE) to the Ping Pong Memory and Common Bar, respectively. When the Counter Sets circuit gets the signals from the Algorithm circuit, it outputs the Test Data and Address for the Ping Pong Memory and Common Bar, composed the Sandwich Ping Pong Memory.

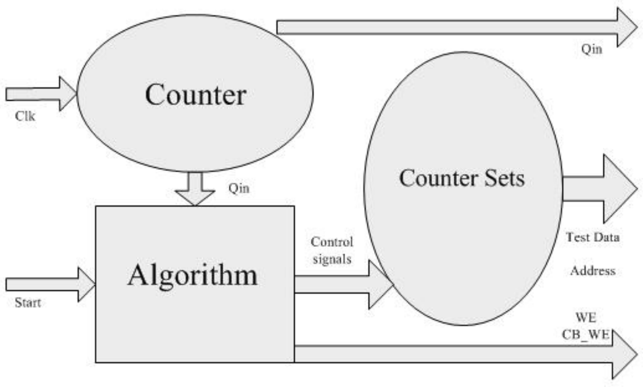


**Figure 5.14** TPG

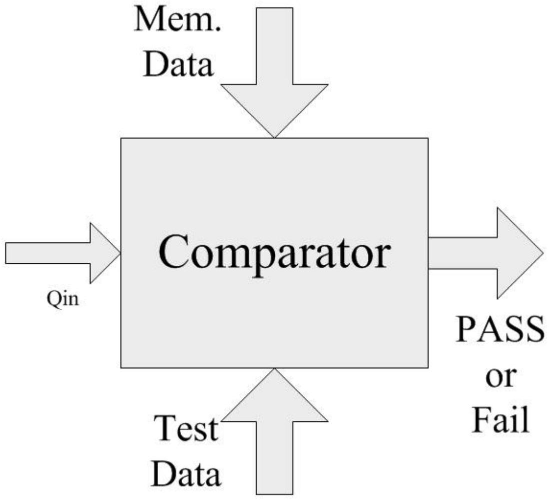


**Figure 5.15** ORA

After the Sandwich Ping Pong Memory access the Test Data, it outputs the Memory Data to the ORA circuit. The ORA circuit is composed with a set of logic gates such as “OR gate” and “Exclusive NOR gate”, shown in figure 5.15. It can compare the Memory Data with the Test

Data in appropriate time applied by Qin. If these two data are the same, it means fault-free in the Sandwich Ping Pong Memory. If these two data are different, it means fault and the chip would be fail. Using software like Xilinx-ISE, we write the HDL-verilog, and synthesize the total circuits.

# Chapter 6 Conclusions and Future Works

## 6.1 Conclusions

The proposed Sandwich Ping Pong Memory is a trade-off between memory area and operation frequency. It can save a significant amount of memory compared with the conventional Ping Pong Memory. In order to design the control unit, the area overhead is under five hundred gate counts at the range of Common Bar is under 512 unit memory cells. The operation frequency is affected by the *Idle Time*. Table 5.3 shows the *Idle Time* and control unit area.

Based on the March C- algorithm, we also developed the test algorithm for the Sandwich Ping Pong Memory and named it the modified March C- algorithm. The fault coverage is at 100% for the stuck-at fault, transition fault, address fault, and coupling fault. In addition, we successfully taped out a 64-byte Ping Pong Memory in process 0.35 $\mu m$ 2p4m in National Chip Implementation Center (CIC). The chip is 1310 x 1100 micro meters squared. Finally, we built the test platform by FPGA.

## 6.2 Future Works

Due to the area limitation of educational chip in National Chip Implementation Center, we only design a 64-byte Ping Pong Memory in process 0.35um. Maybe we should design a bigger memory like 128-byte or 1 Giga-byte and try more experiment. Make the Table 3.2 more complete and the relative between area and operation frequency would be clearer.

# Bibliography

[1] P. Adde, M. Jezequel, “Ping-pong Supervisor for Synchronous Links”, *In Proc. Euro ASIC* pp. 364-367, 1992.

[2] Zhongfeng Wang, Keshab Parhi, “Efficient interleaver memory architectures for serial turbo decoding”, *In Proceedings, ICASSP IEEE*, vol 2, pp 629-632, April 2003.

[3] Y. M. Joo, N. Mckeown, “Doubling Memory Bandwidth for Network Buffers”, *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 808-815, April 1998.

[4] R. O. Onvural, “On performance Characteristics of LAN Interfaces with Ping-Pong Buffers”, *In Proc. Local Computer Networks, 16th conference*, pp. 562-558, 1991.

[5] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1989.

[6] N. Ahmed, T. Natarajan, and K. R. Rao, “Discrete cosine transform”, *IEEE Trans. Comput.*, vol. C-23, pp. 90-93, 1974.

[7] K. R. Rao, and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages and Applications*. Boston, MA: Academic, 1990.

[8] Vladimir Britanak, K. R. Rao, ”Two-dimensional DCT/DST universal computational structure for $2^m \times 2^n$ block sizes**,”** *In Signal Processing, IEEE Transactions***,** vol. 48, Issue 11, pp.3250 - 3255 Nov. 2000.

[9] Z. Wang and B. R. Hunt, “The discrete W transform”, *Appl. Math. Comput.*, vol. 16, pp. 19-48, 1985.

[10] D. Legall, “A video compression standard for multimedia applications”, *Commun. ACM*, 34, (4), pp 46-58, 1991.

[11] S.C. Hsia , B. D. Liu, J. F. Yang, and B. L. Bai, “VLSI implementation of parallel coefficient-by-coefficient two-dimentional IDCT processor”, *IEE Trans. Circuits Syst. Video Techno.* 5, (5), pp. 396-406, 1995.

[12] S. Uramoto et al , “A 100 MHz 2-D discrete cosine transform core processor”, *IEEE J. Solid-State Circiuts.* vol. 27, pp. 492-499, Apr. 1992.

[13] R. Dekker, F. Beeker, L. Thijssen, “Fault Modeling and Test Algorithm Development for Static Random Access Memories”, *ITC*, 1988.

[14] C.-F. Wu, C.-T. Huang, K.-L. Cheng, and C.-W. Wu, “Simulation-based test algorithm generation for random access memories”, *in Proc. IEEE VLSI Test Symp. (VTS)*, Montreal, pp. 291–296, Apr. 2000.

[15] C.-F. Wu, C.-T. Huang, and C.-W. Wu, “RAMSES: a fast memory fault simulator”, *in Proc. Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT*), Albuquerque, pp. 165-173, Nov. 1999.

[16] A.J. van de Goor, G.N. Gaydadjiev, “March U: a test for unlinked memory faults”, *in Proc. IEE*

*Circuits, Devices and Systems*, pp 155 – 160, June 1997.

[17] J. F. Li, K.L. Cheng, C. T. Huang, and C. W. Wu, “March-based RAM diagnostic algorithms for stuck-at and coupling faults”, *Proc. IEEE ITC*, pp. 758-767, 2001.

[18] A.J. van de Goor, “Using March Test to test SRAMs”, *IEEE Design of Computers*, 1993.

[19] D. S. Suk and S. M. Reddy, “A March Test for Functional Faults in Semiconductor Random Access Memories”, *IEEE Transactions on Computers*, vol. C-30, No 12, Dec 1981.

[20] V.A Vardanian, Y. Zorian, “A March-Based Fault Location Algorithm For Static Random Access Memories”, *IEEE International Workshop MTDT* 2002.

[21] V.N Yarmolik, Y.V Klimets, A. J. van de Goor, S.N. Demidenko, “RAM Diagnostic Tests”, *ITC* 1996.

[22] R. Dedkker, F. Beeker, “A Realistic Fault Model and Test Algorithms For Static Random Access Memories”, *IEEE Transactions on Computers Aided Design*, vol. 9, No 6, June 1990.

[23] R. Nair, S.M. Thatte and J.C. Abraham, “Efficient Algorithm for Testing Semiconductor Random-Access Memories”, *IEEE Transactions on Computers*, Vol. C-27, No 6, June 1978.

[24] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, John Wiley & Sons, Chichester, England, 1991.

[25] Charles E. Stroud, *A designer’s guide to built-in self-test*, Kluwer Academic Publishers, Boston, England, 2002.

[26] Wikipedia , the free encyclopedia..

[27] Sedra and Smith, *Microelectronic Circuits fourth edition*, Oxford, 1998.

[28] Tegze P. Haraszti, *CMOS MEMORY CIRCUITS*, Kluwer Academic Publishers, 2000.

[29] Weste and Harris, *CMOS VLSI Design A Circuits and Systems Perspective third edition,* Addison Wesley, 2005.

[30] Wayne Wolf, *Modern VLSI design system-on-chip design third edition*, PEARSION Prentice Hall, 2002.

[31] Smith, *Application-Specific Integrated Circuits*, Addison Wesley, 2004.

[32] 唐經洲, 王立洋, *VLSI 設計概論與實習*, 高立, 2001.

[33] 林灶生, 劉紹漢, *Verilog FPGA 晶片設計*, 全華, 2004.

# Vita

# 作者簡介

## 個人資料：

| | |
|---|---|
| 姓　　名： | 魏文俊 (Wen-Chun Wei) |
| 生　　日： | 民國70年1月3日 |
| 出 生 地： | 臺灣 |
| 專　　長： | 數位電路分析與設計 |
| | 類比電路分析與設計 |

## 學　　歷：

| | |
|---|---|
| 2006.2 ~ 2008.12 | 交通大學IC設計產業研發碩士專班 |
| 1999.9 ~ 2003.7 | 義守大學電機工程學系 |
| 1996.9 ~ 1999.7 | 國立桃園高中 |

## 經　　歷：

| | |
|---|---|
| 2003.8 ~ 2005.3 | 服役 |