

國立交通大學

資訊科學與工程研究所

碩 士 論 文

快 速 非 同 步 平 方 數 產 生 器

A Fast Asynchronous Square Number Generator

研 究 生：李榮祥

指 導 教 授：陳昌居 教授

中 華 民 國 九 十 八 年 六 月

# 快速非同步平方數產生器

學生：李榮祥

指導教授：陳昌居教授

國立交通大學

資訊科學與工程研究所

## 摘要

平方數經常會在數學計算和影像處理中使用到。在傳統上，大都是使用乘法器來產生平方數。但是乘法器實作上複雜且處理速度較慢。而如何有效率的計算出平方數就變得十分的重要。

在 1998 年，謝明得 等教授在 *IEEE Transaction on Computers* 提出一個使用查表法的平方產生器 [4]，這個產生器使用簡單的加法來取代乘法，以達到加速平方數的計算。後來到 2008 年，又提出以遞迴方式，運用更小的 table 使作平方產生器 [5]。

依據前述的遞迴方式，需要作多次的加法運算。我們提出一個“快速演算法”來減少加法運算的次數，再搭配非同步加法器實作平方產生器。在 TSMC 0.13 製成的標準元件下實作後合成，當最佳的狀況，運用快速演算法可最高可加快 18.13%。

# **A Fast Asynchronous Square Number Generator**

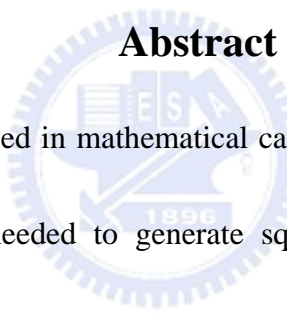
**Student: Zhong-Xiang Lee**

**Advisor: Prof. Chang-Jiu Chen**

**Department of Computer Science and Information Engineering**

**National Chiao Tung University**

## **Abstract**



Square number is often used in mathematical calculation and video compression. In the past, multipliers are widely needed to generate square numbers. However, multiplier is complex, hard to implement and slow. How to reduce overhead of generating square number is a very important issue.

In 1998, Chin-Long Wey and Ming-Der Shieh proposed the Square generator [4]. Instead of the multiplier, a look-up table is needed to generate the square number. This method can accelerate operating speed to generate square numbers. In 2008, Tsai proposed the recursive scheme to reduce the ROM size to implement the square generator [5].

On our work, we propose the “Fast Algorithm” and implemented it with asynchronous circuits. As a result, the square generator with Fast Algorithm implemented on TSMC

0.13  $\mu\text{m}$  is 18.13% faster in the best case than the original design.



# Acknowledgement

能夠順利完成這份論文，首先要感謝我的指導教授-陳昌居老師。在老師兩年的指導之下，讓我獲益良多，也啟發我對研究的興趣。接著要感謝實驗室的博士班學長-鄭緯民學長、張元騰學長以及蔡宏岳學長給予我的論文許多寶貴的意見以及實驗器材操作上許多的指點，讓我少走許多冤枉路，省下許多時間。再來要感謝實驗室夥伴們、關心我的朋友們，還有大學和碩士班時期的同學們給予的加油與鼓勵，有你們的支持，才讓我有動力完成論文。最後要感謝家人的支持，讓我在研讀碩士之路沒有後顧之憂。感謝大家！



# CONTENTS

摘要.....	i
Abstract.....	ii
Acknowledgement.....	iv
CONTENTS.....	v
List of Figures.....	vii
List of Tables.....	ix
Chapter 1 Introduction.....	1
Chapter 2 Related Works.....	3
2.1 Asynchronous Circuit Design.....	3
2.1.1 Advantages.....	3
2.1.2 Handshaking Protocol.....	5
2.1.3 C-element.....	9
2.1.4 Pipeline.....	11
2.2 Square Generator.....	12
2.2.1 Square Generator with Folding Approach.....	12
2.2.2 Square Generator with Recursive Approach.....	18
Chapter 3 Design.....	20
3.1 Fast Algorithm.....	20
3.2 Analysis of the Fast Algorithm.....	25
Chapter 4 Implementation.....	28
4.1 Architecture of Square Generator with Fast Algorithm and Asynchronous Adder.....	28
4.2 One's Complementer.....	29
4.3 DValue Generator and Fast Algorithm.....	31
4.3.1 Implementation of DValue Generator.....	31

4.3.2 Implementation of Fast Algorithm.....	33
4.3.3 Hybrid Adder with ZeroPass Scheme.....	38
4.4 Implementation of Hybrid Adder.....	40
<b>Chapter 5 Simulation.....</b>	<b>45</b>
5.1 Time Simulation.....	45
5.2 Area Simulation.....	47
<b>Chapter 6 Conclusion and Future Works.....</b>	<b>48</b>
<b>Reference.....</b>	<b>49</b>



# Lists of Figures

Figure 1 (a) Bundle data channel.....	6
Figure 1 (b) Four-phase bundled data protocol.....	6
Figure 2 (a) Dual-rail.....	8
Figure 2 (b) Dual-rail protocol.....	8
Figure 3 Transfer diagram.....	8
Figure 4 (a) C-element.....	10
Figure 4 (b) C-element with Reset.....	10
Figure 5 (a) Four-phase bundled-data pipeline.....	11
Figure 5 (b) Four-phase bundled-data pipeline with combinational logic.....	12
Figure 6 Architecture of Square Generator with Two Folding Approach.....	15
Figure 7 Architecture of Square Generator with Four Folding Approach.....	17
Figure 8 (a) Original Addition of DValue.....	21
Figure 8 (b) Addition of DValue with Fast Algorithm.....	21
Figure 9 Expression of DValue Flag.....	23
Figure 10 Pseudo Code of Fast Algorithm.....	24
Figure 11 Rate of the Reducing Operand Bits.....	26
Figure 12 Improvement Rate with Fast Algorithm.....	26
Figure 13 Architecture of Square Generator with Fast Algorithm and Asynchronous Adder.....	29
Figure 14 The Flow Path of One's Complementer.....	30
Figure 15 Gate level of One's Complementer.....	30
Figure 16 Architecture of DValue generator and FastAlgorithm.....	31
Figure 17(a) First Half of DValue Generator.....	32
Figure 17(b) Second Half of DValue Generator.....	33
Figure 18 (a) Probability of Putting Ball into Box in $n=16$ .....	35



Figure 18 (b) Probability of Putting Ball into Box in $n=20$ .....	36
Figure 18 (c) Probability of Putting Ball into Box in $n=24$ .....	36
Figure 18 (d) Probability of Putting Ball into Box in $n=28$ .....	37
Figure 18 (e) Rate of Reducing Operand Bits in 4-bits DValue.....	37
Figure 19 Circuit of Fast Algorithm.....	38
Figure 20 (a) Four Phase Control with Counter in 32-bit Hybrid Adder.....	39
Figure 20 (b) Data Path of 32-bit Hybrid Adder.....	40
Figure 21 32-bits Hybrid Adder for Square Generator.....	42
Figure 22 (a) 1-bit Hybrid Adder.....	42
Figure 23 (b) 1-bit Hybrid Adder on the LSB.....	43
Figure 24 (c) 1-bit Hybrid Adder on the 30th Bit.....	43
Figure 25 Alternative C-element.....	44
Figure 26 (a) Best-case for Fast Algorithm when input value = $(AA55)_h$ .....	46
Figure 27 (b) Worst-case for Fast Algorithm when input value = $(6655)_h$ .....	46

# Lists of Tables

Table 1 Encoding method.....	7
Table 2 Truth table of C-element.....	9
Table 3 Square Generator with Two-Folding Approach.....	14
Table 4 Square Generator with Four-Folding Approach.....	16
Table 5 Square Generator with Recursive Approach.....	19
Table 6 Numerical result of number of operand bits with Fast Algorithm.....	26
Table 7 (a) Probability of Putting Ball into Box if $n = 16$ .....	34
Table 7 (b) Probability of Putting Ball into Box if $n = 20$ .....	34
Table 7 (c) Probability of Putting Ball into Box if $n = 24$ .....	34
Table 7 (d) Probability of Putting Ball into Box if $n = 28$ .....	34
Table 8 Latency between with Fast Algorithm and without Fast Algorithm.....	47
Table 9 Area of between With Fast Algorithm and Without Fast Algorithm.....	47

# Chapter 1 Introduction

Generating square number is often used in digital signal processing, mathematical calculation and video compression. In the past, generating square number was by multiplier. However, multiplier has some disadvantages. First, it is complex and hard to implement. Then, speed of multiplier is slow. At last, it has very high overhead of area. How to reduce overhead and speedup of generating square number is a very important issue.

In the recently, this topic was focused. In 1998, Chin-Long Wey and Ming-Der Shieh purposed the Square generator [2]. Their method uses the look-up table to produce the square number. This method can accelerate operating speed to generate square number. According to Wey's algorithm, it used the recursive scheme to reduce the ROM size. In 2008, Tsai's paper had another approach to improve square generator [3].

Asynchronous circuit design [4, 5] is a circuit design methodology. Asynchronous circuit design and synchronous circuit design are totally different. Asynchronous circuit design uses the handshake protocol to communicate with sub-circuits without a global clock. With the protocol, the speed of circuit does not set the worst-case clock cycle time; on the other hand, it can achieve average-case speed. Asynchronous circuit design has other advantages for circuit design, such as modularity, no clock skew problems and low power consumption etc [7-9].

Our design unifies the benefits of the asynchronous circuit design and square generator. In asynchronous circuit design, the complete detection adder which is added into square generator performs the average-case performance. Then, the proposed Fast Algorithm

improves the square generator with recursive scheme. According to mathematical analysis, Fast Algorithm can speed-up about 11% comparing to the original square generator. Finally, the asynchronous square generator with Fast Algorithm is implemented on TSMC 0.13  $\mu m$  process and the area is 175963.23  $\mu m^2$ . The latency of our design is 64.87ns at most. Compare with asynchronous square generator without Fast Algorithm, the rate of improving latency is about 18.13%.



# Chapter 2 Related Works

This chapter will give introduction of asynchronous circuit design and square generator.

We show the advantages, handshaking protocols, pipeline and C-element of asynchronous circuit design. We also give descriptions of square generator. There are two ways to implement square generator: folding approach and recursive approach.

## 2.1 Asynchronous Circuit Design

Asynchronous circuit design is a circuit design methodology. Asynchronous circuit is practically different from synchronous circuit. There are many advantages in the asynchronous circuit design and special ways to design circuit. The following will list advantages and handshaking protocols.

### 2.1.1 Advantages

Comparing with the synchronous circuit design, the asynchronous circuit design has no global clock and use handshaking protocol between the sub-circuits to perform synchronization and communication. The following are advantages of the asynchronous circuit design :

- 2.1.1.1 Low power consumption: Asynchronous circuit do not supply extra power to generate clock tree. The sub-circuits in a synchronous circuit are clock-driven, whereas they are demand-driven in an asynchronous circuit. This means that the sub-circuits in an asynchronous circuit are only active when and where needed. That means that there is no clock tree. According

to [6], the power consumption of asynchronous circuit is up to 36% to 40%.

2.1.1.2 Average-case speed: The elasticity of the asynchronous pipeline has led to the outcome that an asynchronous pipeline can perform 'average' processing rather than worst case time for each of synchronous pipeline stages. When asynchronous pipeline has been completed to receive the data, receiver could send acknowledge signal to sender. Sender can do next job early.

However, the pipeline of synchronous circuits choose the longest time of pipeline stage to be clock cycle time. Comparing with asynchronous pipeline, asynchronous pipeline has individual time of stage. It is independent on each asynchronous stage. For this reason, the asynchronous circuit can accomplish average-case performance.

2.1.1.3 Modularity: Because of the simple handshake interfaces and local timing, the asynchronous circuit is easy to be divided into different modules. Designers only need to take care of synchronization between different modules. The speed of a module does not influence other modules.

2.1.1.4 No clock distribution and clock skew problems: Clock plays an important role on communication. But there is a serious problem about

clock skew. System becomes larger and larger, so it is more and more difficult to transmit clock signals. However, the asynchronous circuit has no clock distribution. It uses the handshaking protocol to avoid clock skew problem.

### **2.1.2 Handshaking Protocol**

Handshaking protocol is a way to communicate in the asynchronous circuit design. In general, there are two parts in the circuit design. One is data path, and the other is control path. In the Handshaking protocol, it defines these parts below:

- Data path: bundled-data, dual-rail, 1-of-n encoding etc.
- Control path: 4-phase, 2-phase etc.

The bundled-data and dual-rail data are two common ways to transfer data between sender and receiver. First, the bundled-data channel (Figure 1 (a)) which data signals use normal Boolean levels to encode information and separates request and acknowledge wires are bundled with the data signals (Figure 1(b)).

Four-phase bundled-data uses REQ and ACK signals as the synchronization signals between the sender and receiver. Four-phase protocol is known as the return-to-zero handshake protocol. Initially, REQ and ACK signals are all 0. After DATA is valid in the SENDER, REQ signal is asserted to 1 by the SENDER (1). If the RECEIVER has accepted the DATA from the SENDER, the RECEIVER would have asserted ACK signal to 1(2). After the SENDER receives the ACK signal from

the RECEIVER, the SENDER will dessert the REQ signal to 0(3) and then stop transfer DATA. Finally, RECEIVER will reset the ACK signal to 0 after RECEIVER receives REQ = 0(4). When SENDER and RECIEVER finish these four operations, it completes a handshaking. At this point, the SENDER can do next communication cycle can the started (5).

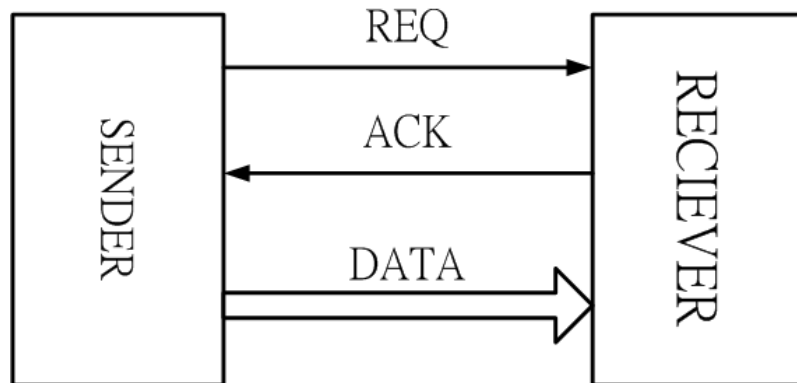


Figure 1(a) Bundled-data channel

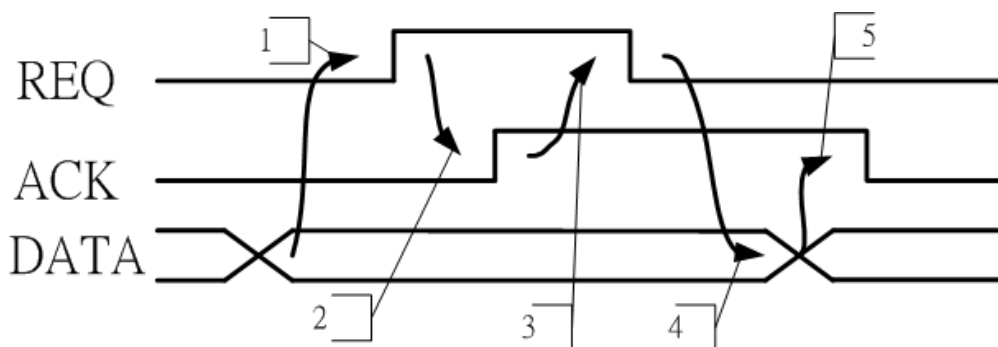


Figure 1(b) Four-phase bundled-data protocol

The other way of handshaking is dual-rail (Figure 2(a)). In stead of “bundled-data”



with the REQ and ACK signals, the dual-rail encoding encodes each 1-bit data with 2 bits. The encoding method shows in Table 1. It uses 00 to show that there is no data, 01 to encode the data of 0 and 10 to encode the data of 1. If the system uses dual-rail protocol to transfer n-bits data, there will have  $2*n$  data lines.

	d.t	d.f
Empty "E"	0	0
Valid "0"	0	1
Valid "1"	1	0
Not used	1	1

Table 1 Encoding method

Because the dual-rail circuit does not have REQ signal, the RECEIVER needs extra circuits to detect DATA signals are arrival. This is called completion detection.

Figure 2(b) shows the process of data transfer using dual-rail protocol. Initially, DATA is EMPTY, and ACK is 0. When DATA is Valid and RECEIVER detects that DATA is ready, RECEIVER captures DATA and pulls up ACK. Then SENDER stops sending DATA and DATA changes to EMPTY. Finally RECEIVER pulls down ACK and the transfer is finish.

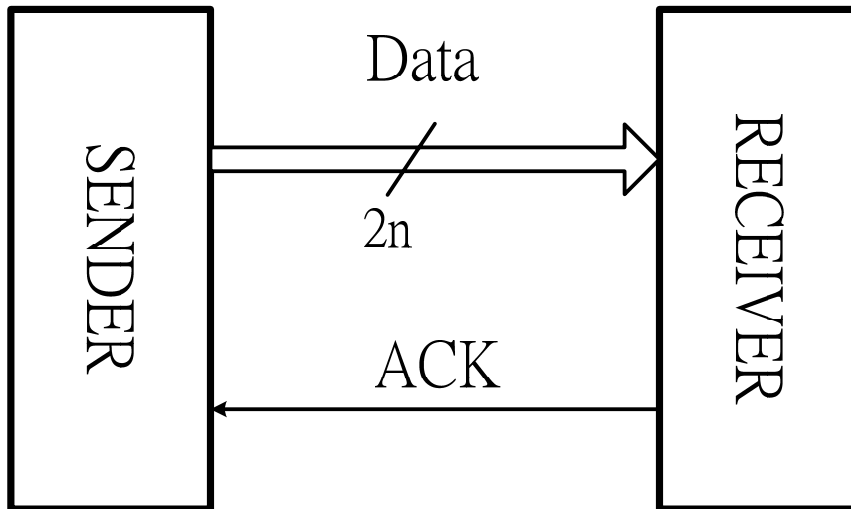


Figure 2(a) Dual-rail

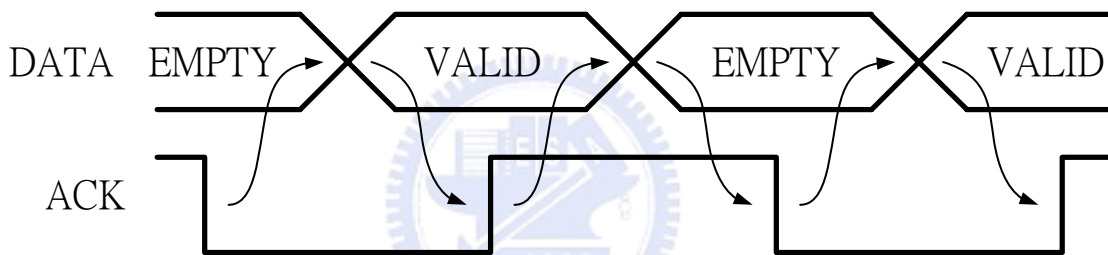


Figure 2(b) Dual-rail protocol

Valid DATA will be separated. Dual-rail protocol uses EMPTY to separate each DATA

(Figure 3).

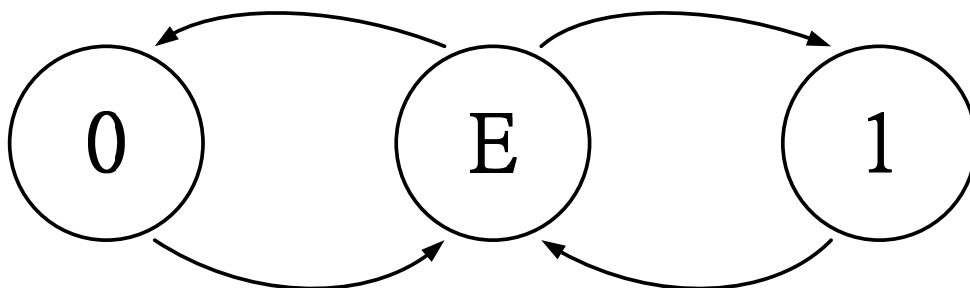


Figure 3 Transfer diagram

### 2.1.3 C-element

The C-element plays an important role on asynchronous circuit design. It usually deals with REQ signal and ACK signal. Table 2 shows behaviors of the C-Element. When both inputs are 0, the output Z is set to 0. And when both inputs are 1, the output Z is set to 1. When the inputs are different, the output Z is equal to 'No Change' (i.e. keeping previous output). Because C-element is state-holding gate, the output is only changed by the different inputs. Figure 4(a) shows the symbol and gate-level design of C-element.

A	B	Z
0	0	0
0	1	No Change
1	0	
1	1	1

Table 2 Truth table of C-element

C-element is usually used on circuit of complete detector. This circuit often appears on the asynchronous circuit. Because we use the RTL design method, Figure 4(b) shows that two inputs C-element with reset scheme.

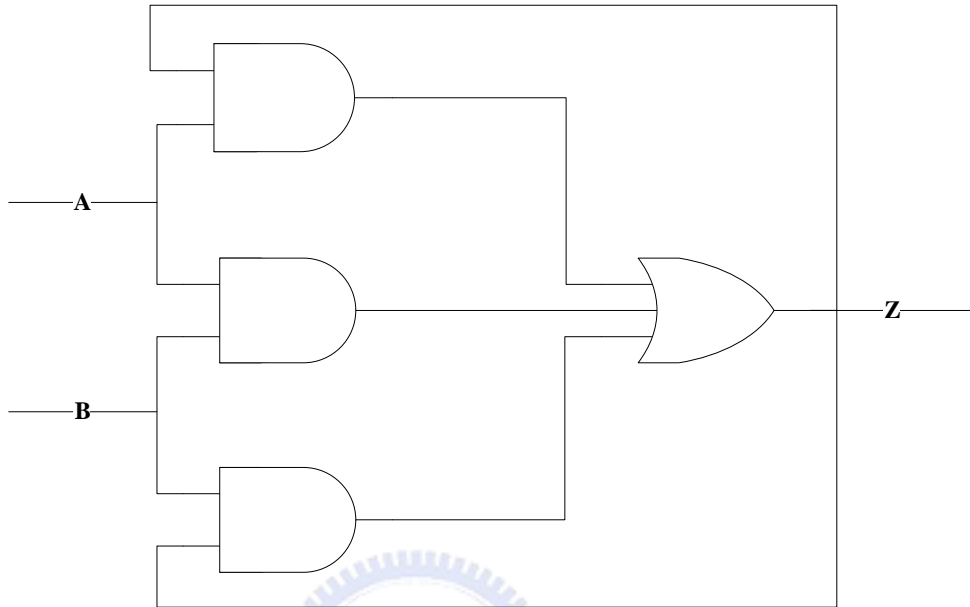
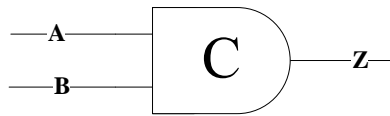


Figure 4(a) C-element

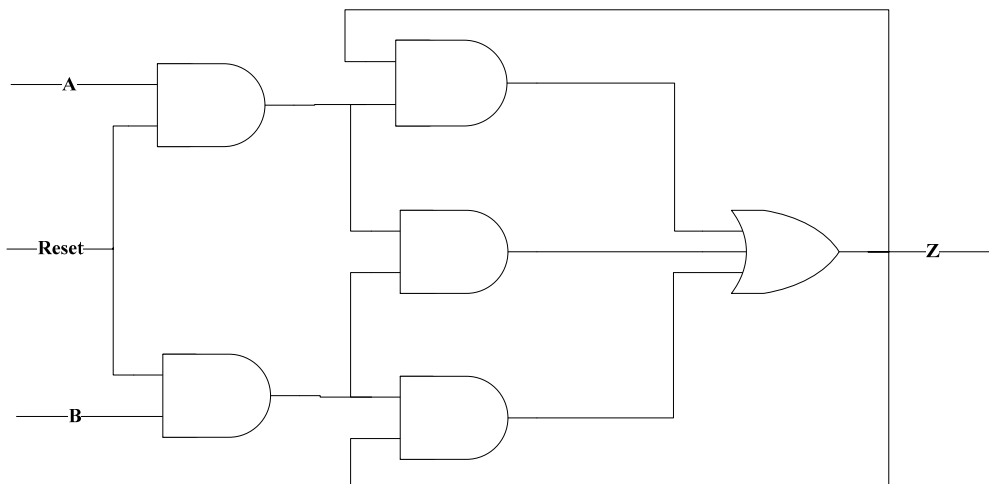
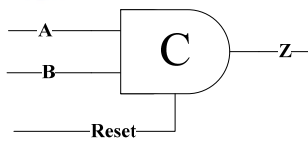


Figure 4(b) C-element with Reset

## 2.1.4 Pipeline

A four-phase bundled-data pipeline is particularly simple and similar to synchronous pipeline with local clock. Figure 5(a) shows a FIFO which is without data processing and Figure 5(b) shows how to add combinational logic into four-phase bundled-data pipeline. This pipeline has several characteristics which are listed below:

- 1). It is simple and easy to implement.
- 2). If the right stage do not receive the acknowledge signal, this pipeline will fill and stall.
- 3). To maintain correct behavior, the matching delay should be inserted in the request signal paths.
- 4). When pipeline is full, only half the latches store data.
- 5). It is similar to a master-slave flip-flop.

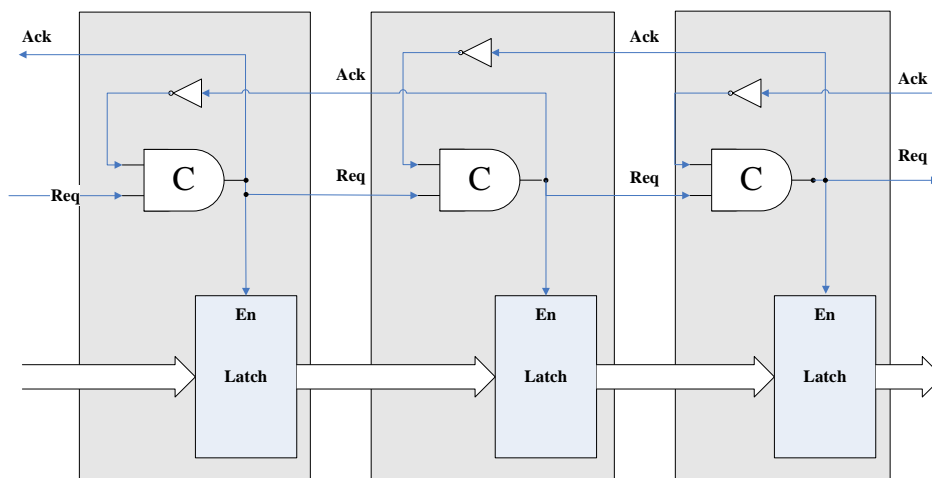


Figure 5(a) Four-phase bundled-data pipeline

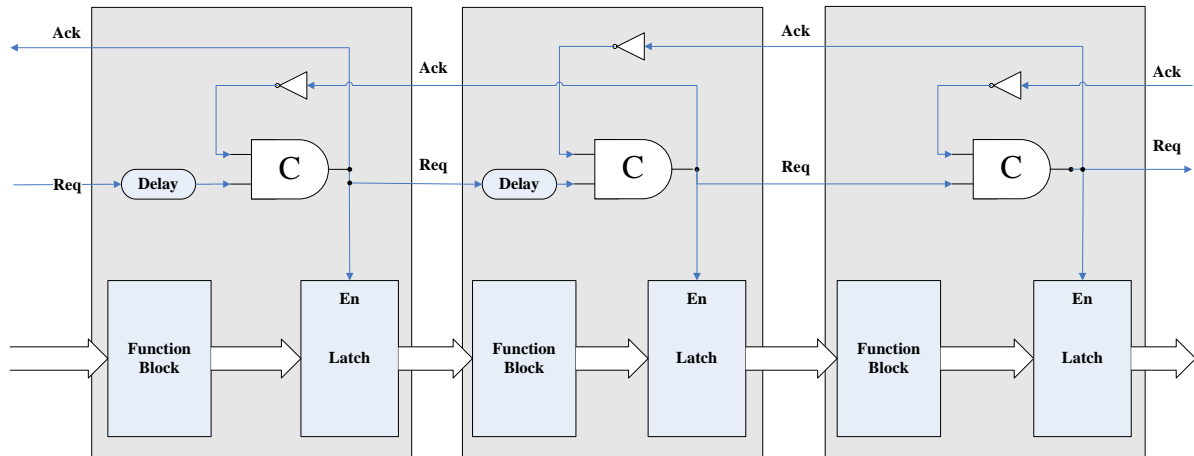


Figure 5(b) Four-phase bundled-data pipeline with combinational logic

## 2.2 Square Generator

High-speed generating square numbers are often required for digital signal processing, video compression processing, and many other mathematical calculations. In order to generate square number, the multiplier should be used. But multiplier has high area cost and is complex. However, square number is often needed on many domains. In the recent years, the square generator is developed by different ways [1-4]. Next two sections will introduce the square generator. First, generator is implemented by folding approach. Second, it use recursive scheme to implement.

### 2.2.1 Square Generator with Folding Approach

In 1998, Chin-Long Wey and Ming-Der Shieh proposed “Design of a High-Speed Square Generator” [2] in the IEEE Transactions on Computers. The simplest way for calculating square number used the look-up table in the ROM.

This section describes development of square generator with huge loop-up

ROM table. The following are property of square generator. Let  $A = (a_{n-1}a_{n-2} \dots a_1a_0)$

be an n-bit binary number and  $|A|$  can express as :

$$|A| = \sum_{i=0}^{n-1} a_i 2^i \dots \dots \dots (1)$$

Property 1.  $A = (a_{n-1}a_{n-2} \dots a_1a_0)$ ,  $B = (b_{n-1}b_{n-2} \dots b_1b_0)$  : n-bit binary number, where

$b_i = 1 - a_i$ ,  $0 \leq i \leq n - 1$ . The difference formula can be expressed as

$$|D| = |A|^2 - |B|^2 = -b_{n-1}2^n(2^n - 1) + |(a_{n-2} \dots a_1a_0 0b_{n-2} \dots b_1b_0 1)| \dots \dots \dots (2)$$

Proof : The difference value  $|D|$

$$\begin{aligned} |D| &= \left( \sum_{i=0}^{n-1} a_i 2^i \right)^2 - \left( \sum_{i=0}^{n-1} b_i 2^i \right)^2 = \left[ \sum_{i=0}^{n-1} (a_i - b_i) 2^i \right] \left[ \sum_{i=0}^{n-1} (a_i + b_i) 2^i \right] \\ &= \left[ \sum_{i=0}^{n-1} (1 - 2b_i) 2^i \right] \left[ \sum_{i=0}^{n-1} 2^i \right] \\ & \quad (a_i - b_i = 1 - 2b_i, a_i + b_i = 1) \\ &= \left( 2^n - 1 - 2 \sum_{i=0}^{n-1} b_i 2^i \right) (2^n - 1) \\ &= \left( -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \right) 2^{n+1} + 2 \left( b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \right) + 1 \\ &\Rightarrow |D| = -b_{n-1} 2^n (2^n - 1) + |(a_{n-2} \dots a_1 a_0 0 b_{n-2} \dots b_1 b_0 1)| \end{aligned}$$

For example, there are two five-bit binary numbers  $A = (a_4a_3a_2a_1a_0) = (10101)$ ,  
 $B = (b_4b_3b_2b_1b_0) = (01010)$ , where  $b_i = 1 - a_i$ . Their square number is  $|A|^2 = 441$  and  
 $|B|^2 = 100$ . The difference value is  $|D| = 341 = (01010 \ 10101) = (a_3a_2a_1a_0 0$   
 $b_3b_2b_1b_0 1)$ .

There are two conditions that A and B are unsigned number and  $b_i$  is one's complement of  $a_i$ . If  $a_{n-1} = 1$ , the  $b_{n-1} = 0$ . By difference formula, the difference value is:

$$|D| = |A|^2 - |B|^2 = |(a_{n-2} \dots a_1 a_0 \overline{0 a_{n-2} \dots a_1 a_0} 1)| \dots \dots \dots (3)$$

According to property 1, the following property concludes.

Property 2 Two-Folding Approach: Given an n-bit binary number  $N = (a_{n-1} a_{n-2} \dots a_1 a_0)$ . Dependent on Property 1, we obtain  $|N|^2 = |N^*|^2 + |D|$  (Two-folding approach), where  $N^*$  and  $D$  are defined as:

$a_{n-1}$	$N^*$	$D$
0	$(a_{n-2} \dots a_1 a_0)$	$(000 \dots 0 \ 000 \dots \dots 0)$
1	$(\overline{a_{n-2} \dots a_1 a_0})$	$(a_{n-2} \dots a_1 a_0 \ 0 \ \overline{a_{n-2} \dots a_1 a_0} \ 0)$

Table 3 Square Generator with Two-Folding Approach

In the folding approach, square generator uses the look-up table simplifies the process as a simple addition or subtraction and table look-up. For instance,  $N$  is six-bit binary number. If  $N = (011 \ 011) = 27$  (i.e.  $a_n = 0$ ), the  $N^* = (11 \ 011)$  and  $D = (000000 \ 000000)$ . It represents reducing MSB to loop-up table and then generates  $|N^*|^2$  to be equal to  $N^2$ . Another condition is  $a_n = 1$ . If  $N = (110 \ 010) = 50$ , the  $N^* = (01 \ 101)$  and  $D = (100100 \ 011011)$ . Then square generator looks up the  $|N^*|^2$  from the ROM. Finally,  $|N^*|^2 + |D|$  is the square result of input data. If input data is n-bit binary number, the ROM size will be  $2^{n-1} \times 2(n-1)$  bits. However, the two-folding approach has huge overhead on the ROM size.



Figure 6 shows the architecture of square generator which consists of three major parts:

- (1). a one's complementer(OC)
- (2). a ROM(store the look-up table)
- (3). a D-value generator (DG)

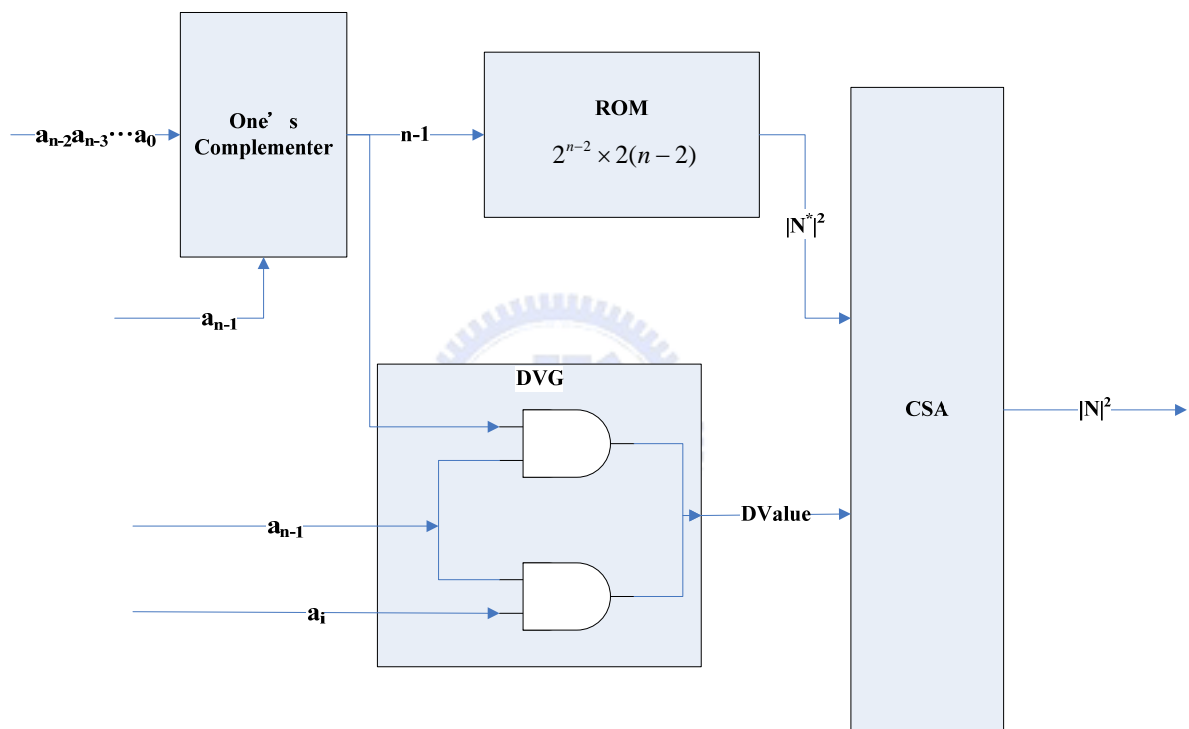


Figure 6 Architecture of Square Generator with Two Folding Approach

The above approach can reduce more than half size of look-up ROM table. Nevertheless, the look-up ROM is still very huge and square generator needs  $2*(n-1)$  bits adder. Next Property will introduce the four-folding approach which is better than the two-folding approach in the ROM size.

Property 3 Four-Folding Approach: There is a n-bit binary number  $N = (a_{n-1}a_{n-2}\dots a_1a_0)$ . Based on the Property 2 gets  $|N|^2 = |N^*|^2 + |D|$  equation, where  $N^*$  and  $D$  are defined as:

$a_{n-1}$	$a_{n-2}$	$N^*$	$D$
0	0	$(a_{n-3}\dots a_1a_0)$	$(000\dots 0\ 000\dots\dots 0)$
1	0	$(a_{n-3}\dots a_1a_0)$	$(01a_{n-3}\dots\dots a_1a_0\ 00\dots\dots 0)$
0	1	$(\overline{a_{n-3}}\dots\overline{a_1a_0})$	$(00a_{n-3}\dots a_1a_0\ \overline{0a_{n-3}\dots a_1a_0}1)$
1	1	$(\overline{a_{n-3}}\dots\overline{a_1a_0})$	$(1a_{n-3}\dots a_1a_00\ \overline{0a_{n-3}\dots a_1a_0}1)$

Table 4 Square Generator with Four-Folding Approach

This approach extends the two-folding approach. It depends on the most significant two bits to generate  $N^*$  and difference value. Table 4 describes how to use the four-folding approach. The following is an example to show the four-folding approach. Figure 7 illustrates the architecture of four-folding approach. There are a few differences between the two approaches. One's complementer decreases 1 bit. A  $2^{n-2} \times 2(n-2)$  ROM is used in the square generator. Four folding approach divides difference value generator into DValue\_High and DValue\_Low two part which can depend on their characteristics to design.

For example, a five-bit binary number  $N$  is equal to 10111(23). We know the  $(a_4, a_3) = (1, 0)$ . Then based on the Table 3, we can obtain the  $N^* = (111)$ ,  $D = (01111\ 00000)$ , and  $N^2 = |N^*|^2 + D = (11\ 0001) + (01111\ 00000) = (10\ 0001\ 0001) = 529$ . The look-up table size becomes  $2^{n-2} \times 2(n-2)$  bits. Thus the needed ROM size is about 50% less than that of two-folding approach. Addition length of square generator can reduce two bits slightly. The four-folding approach not only improves the area cost, but also the operation speed of square generator (because of less lengths of addition).

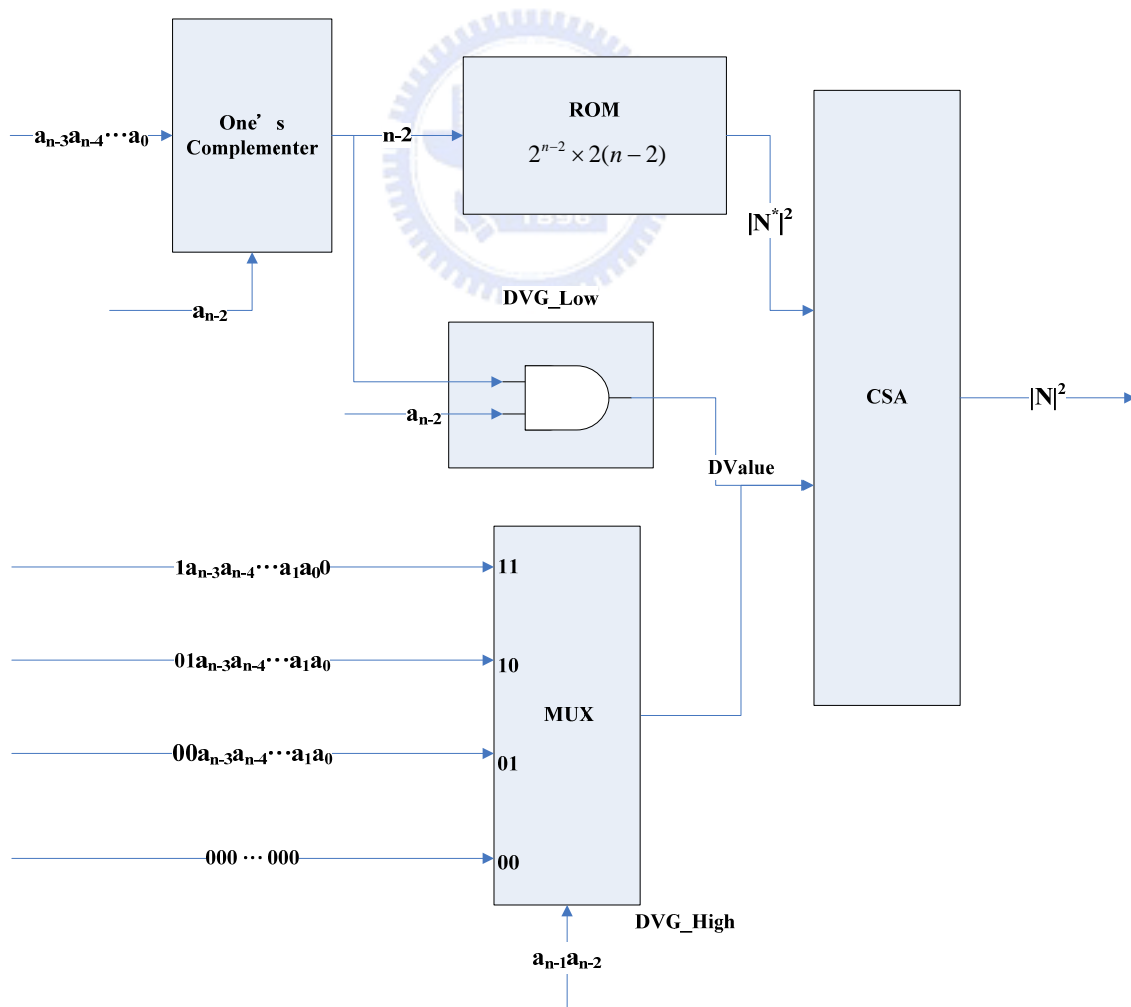


Figure 7 Architecture of Square Generator with Four Folding Approach

### 2.2.2 Square Generator with Recursive Approach

Due to overhead on the ROM size, Wei-Chang Tai in 2008 proposed a new approach [2] whose square generator is implemented with small look-up table. Tai was rewritten the equation by recursive scheme and four-folding approach. Equation (2) shows the calculation of new approach. In the equation (2), under the N and D value represents the width of values. The Table 5 shows the recursive scheme for general case. The r number represents times of recursive.

$$\begin{aligned}
 |N|_{2n}^2 &= |N^*|_{2(n-2)}^2 + |D|_{2n} \\
 &= (|N^{**}|_{2(n-4)}^2 + |D|_{2(n-2)}) + |D|_{2n} \quad \dots (2) \\
 &= \dots \\
 &= |D|_{2n} + |D|_{2(n-2)} + \dots + |D|_8 + |D|_4
 \end{aligned}$$

Based on four-folding approach, the ROM size is reduced by generating more difference values. It may sacrifice the performance reduce the cost of ROM size more than 75%. If we do recursive scheme  $n/2$  times, square generator has no look-up table ROM and there are  $n/2$  difference values. It has to do addition  $\log_2(n/2)$  times.

$a_{n-2r+1}$	$a_{n-2r}$	$N^{*(r)}$	$D_i$
0	0	$(a_{n-2r+1} \cdots a_1 a_0)$	$(000 \dots 0 \ 000 \dots 0)$
1	0	$(a_{n-2r+1} \cdots a_1 a_0)$	$(01a_{n-3} \dots a_1 a_0 \ 00 \dots 0)$
0	1	$(\overline{a_{n-2r+1}} \cdots \overline{a_1 a_0})$	$(00a_{n-3} \dots a_1 a_0 \ 0\overline{a_{n-3}} \dots \overline{a_1 a_0} 1)$
1	1	$(\overline{a_{n-2r+1}} \cdots \overline{a_1 a_0})$	$(1a_{n-3} \dots a_1 a_0 0 \ 0\overline{a_{n-3}} \dots \overline{a_1 a_0} 1)$

Table 5 Square Generator with Recursive Approach



# Chapter 3 Designs

Due to square generator with recursive scheme, there are too many operands. If we use synchronous circuits design, system has to set the worst-case delay to be clock cycle time, the part of adder tree is the critical path in the square generator. For this reason, we design a new circuit to implement square generator with recursive scheme. We use the asynchronous circuit design which does not have a global clock to achieve the average-case performance, and use zeroflags to reduce the additions.

In order to design the proposed method with the asynchronous circuit, the handshake protocol selected is the four-phase bundled-data. Furthermore, an asynchronous adder with complete detection is used to speed up the calculation. Besides, through observing the Table 4, we develop the “Fast Algorithm” to reduce the number of bits of addition. This chapter describes the model and the analysis of Fast Algorithm.

## 3.1 Fast Algorithm

According to Table 5, we observe a condition that second half part of the difference value (DValue) is zero when  $(a_{n-2r+1}, a_{n-2r}) = (1, 0)$ . If there is a  $2n$ -bits DValue ( $|D|_a$ ) with all zero in the second half bits, and we can use Fast Algorithm to find a  $n$ -bits non-zero DValue ( $|D|_b$ ). The new DValue ( $|D|_c$ ) is generated by combining  $|D|_a$  with  $|D|_b$ . For instance,  $N$  is 12-bit binary number. After using recursive scheme, square generator produces six difference values which are  $|D|_{24}$ ,  $|D|_{20}$ ,  $|D|_{16}$ ,  $|D|_{12}$ ,  $|D|_8$  and  $|D|_4$ .

Figure 8(a) expresses this condition. It should be noted that the width should be matched.

i.e.  $|D|_{12}$  can only put into  $|D|_{24}$ ,  $|D|_8$  can be put into both  $|D|_{24}$  and  $|D|_{20}$ , and  $|D|_4$  which has the minimum width can be put into  $|D|_{24}$ ,  $|D|_{20}$  and  $|D|_{16}$ . We suppose second half of  $|D|_{24}$ ,  $|D|_{20}$  and  $|D|_{16}$  are zero and all of  $|D|_{12}$ ,  $|D|_8$  and  $|D|_4$  are non-zero number. In the Figure 8(a) condition, (iv) can put into (i), (v) can input to (ii), and (vi) can put into (iii). Finally, Figure 8(b) illustrates the result after finished previous operations. This approach is called the “Fast Algorithm”.

$$\begin{aligned}
 |D|_{24} &= 01d_{21}d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12} \boxed{000000000000} & \text{(i)} \\
 |D|_{20} &= 01d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10} \boxed{0000000000} & \text{(ii)} \\
 |D|_{16} &= 01d_{13}d_{12}d_{11}d_{10}d_9d_8 \boxed{00000000} & \text{(iii)} \\
 |D|_{12} &= \boxed{d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0} & \text{(iv)} \\
 |D|_8 &= \boxed{d_7d_6d_5d_4d_3d_2d_1d_0} & \text{(v)} \\
 +) |D|_4 &= \boxed{d_3d_2d_1d_0} & \text{(vi)}
 \end{aligned}$$

Figure 8(a) Original Addition of DValue

$$\begin{aligned}
 |D|_{24} &= 01d_{21}d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12} ( |D|_{12} ) & \text{(i)} \\
 |D|_{20} &= 01d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10} ( |D|_8 ) & \text{(ii)} \\
 |D|_{16} &= 01d_{13}d_{12}d_{11}d_{10}d_9d_8 ( |D|_4 ) & \text{(iii)} \\
 |D|_{12} &= 000000000000 & \text{(iv)} \\
 |D|_8 &= 00000000 & \text{(v)} \\
 +) |D|_4 &= 0000 & \text{(vi)}
 \end{aligned}$$

### Figure 8(b) Addition of DValue with Fast Algorithm

Because previous description is hard to understand, we re-define the relationship between the first half difference values and the second half difference values. We define a  $n/2$  bits flag as “DValue”. DValue includes the ‘Box’ and ‘Ball’ flags which describes as Figure 9. Box is a flag to record if all second half bits of first half DValues are zeros. If the value of Box is 1, the second half bits of the DValue are zeros. Ball is a flag to represent if all second half difference values are non-zero values. If the value of Ball is 1, the DValue are non-zero values. For example,  $\text{Box} = \{x_{n/4-1}x_{n/4-2} \cdots x_1x_0\}$  and  $\text{Ball} = \{b_{n/4-1}b_{n/4-2} \cdots b_1b_0\}$ . If  $x_i = 1$ , it expresses  $(a_{n/2+2i+1}, a_{n/2+2i}) = (0, 1)$ , and  $b_j = 1$  when  $(a_{2j+1}, a_{2j}) = (0, 1)$ . We assume that the input data is a uniform statistical distribution. The probability of  $x_i = 1$  is 0.25(i.e. (I)), which of  $x_i = 0$  is equal to 0.75(i.e. (II)). For the same reason, the probability of  $b_j = 1$ (exist the ball) is 0.75(i.e. (III)). On the other hand, the  $b_j = 0$ (no ball) is 0.25(i.e. (IV)). Based on the above description of probability, next section will compile statistics about improvement rate with Fast Algorithm.



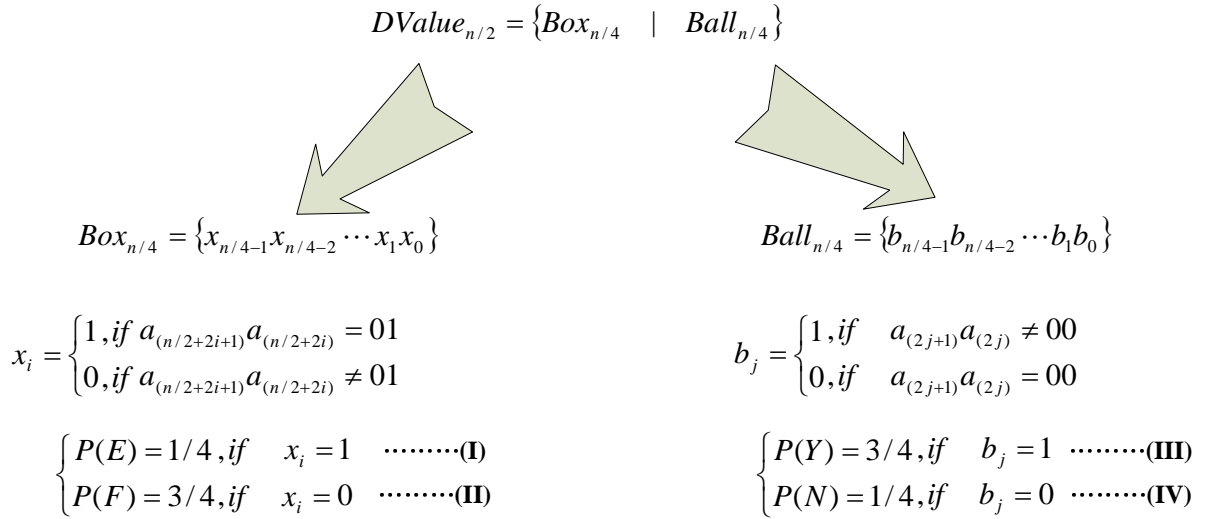


Figure 9 Expression of DValue Flag

Figure 10 is the pseudo code of Fast Algorithm which expresses how to search empty box and available ball. First of all, if an available ball is scanned, it will enter the ScanBoxes function to find a suitable box. The way to find the suitable box in the ScanBoxes is setting the bound to search. The ScanPoint variable records box point of last searching position.

```

1 NumBall = N/4;           // # of Balls
2 NumBox = N/4;           // # of Boxes
3 Ball[NumBall];         // Ball[i] = 1 => available, 0 => No ball
4 Box[NumBox];           // Box[i] = 1 => Empty, 0 => Full
5 ScanPoint = NumBox;     // Record box point of scanning
6
7 for i from NumBall to 1 by i-- do           // Scan all available Balls
8 begin
9     if ScanPoint != 0                       // do not scan all of the Boxes
10    begin
11        if Ball[i] == 1                     // there is a ball
12            ScanBoxes(Box, i, ScanPoint);
13        else
14            ;                               // no-op, scan next ball
15    end
16    else
17        ;                               // no-op
18 end
19
20 ScanBoxes(Box, NOBall, ScanPoint)
21 begin
22     Temp_Bound = 2*NOBall - NumBox; // Set up bound for specific size boxes
23     if ( Temp_Bound > 0 )
24         Bound = max(Temp_Bound, 1);
25     else
26         Bound = 1;
27
28     for j from ScanPoint to Bound by j-- do
29     begin
30         if (Box[j] == 1)                 // There is a fitting box
31         begin
32             ScanPoint--;
33             DValue_Low[NumBall + 4*j] = DValue[4*NOBall];
34             exit scan box;
35         end
36         else                             // Full box
37             ScanPoint--;
38     end
39 end

```

Figure 10 Pseudo Code of Fast Algorithm

In terms to advantage of asynchronous adder, Fast Algorithm can speed-up the square generator. In addition, Figure 9 and Table 5 is expressed as probability of  $|D|_i = 0$  is 0.25. As soon as operands of addition are zero, asynchronous adder send complete signal to accelerate system performance. So, the asynchronous adder is a good choice to implement the square generator.

## 3.2 Analysis of the Fast Algorithm

This section describes the analysis of the Fast Algorithm. We assume a uniform statistical distribution of the input data whose width is 8 bits to 72 bits. The numerical results on number of reducing bits is obtained from (I), (II), (III), (IV) (p.p 21) and Fast Algorithm by computer program execution, are reported in Table 5. The respect value of number of Operand Bits without Fast Algorithm on the addition is equal to  $3/4 \times (n \times (n/2 + 1))$ , where  $3/4$  is probability of non-zero values and  $(n \times (n/2 + 1))$  expresses the sum of number of all DValues bits.

Figure 11 illustrates that the rate of the reducing operand bits depends on the width of input data progressively. The asynchronous adder design with Fast Algorithm is 8% ~ 11% faster than that without Fast Algorithm in square generator (Figure 12). Asynchronous adder design is faster than synchronous if there is a DValue = 0 and each width of DValue is not equal.

n	Number of Operand Bits without Fast Algorithm	Number Of Reducing Bits	Number of Operand Bits with Fast Algorithm	Rate of Reducing Operand Bit
8	30	2.390625	27.609375	7.96875%
12	63	5.633789063	57.36621094	8.9425223%
16	108	10.00982666	97.99017334	9.268358%
20	165	16.04515457	148.9548454	9.7243361%
24	234	23.28934193	210.7106581	9.9527102%
28	315	32.20196016	282.7980398	10.2228445%
32	408	42.35878338	365.6412166	10.3820547%
36	513	54.18483815	458.8151618	10.5623466%
40	630	67.27547568	562.7245243	10.6786469%
44	759	82.03470844	676.9652916	10.808262%
48	900	98.07197134	801.9280287	10.8968857%
52	1053	115.7771509	937.2228491	10.9949811%
56	1218	134.7700275	1083.229972	11.0648627%
60	1395	155.4303336	1239.569666	11.1419594%
64	1584	177.385669	1406.614331	11.1985902%
68	1785	201.0081215	1583.991879	11.2609592%
72	1998	225.931363	1772.068637	11.307876%

Table 6 Numerical result of number of operand bits with Fast Algorithm

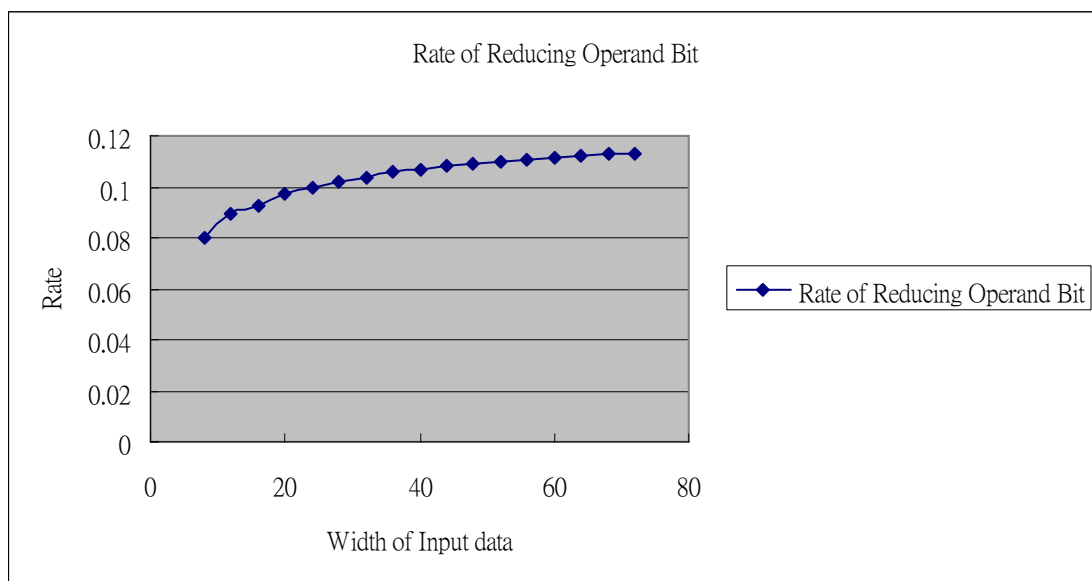


Figure 11 Rate of the Reducing Operand Bits

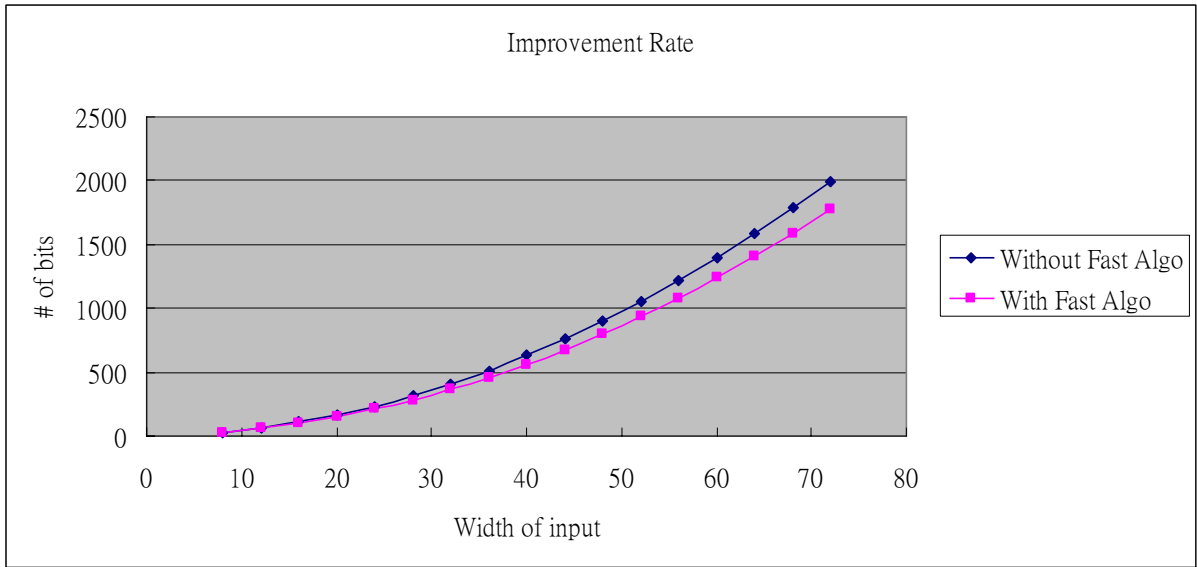


Figure 12 Improvement Rate with Fast Algorithm



# Chapter 4 Implementation

This chapter will describe the implementation of square generator with Fast Algorithm in asynchronous circuit design. Our model uses pipeline to achieve higher performance and the following sections shows the details of the model.

In our designs, the input data size is 16 bits and output size is 32 bits. We use four-phase bundled-data protocol and three pipeline stages to implement the square generator which contains a 32 bits hybrid adder with recursive scheme.

## 4.1 Architecture of Square Generator with Fast Algorithm and Asynchronous Adder

In our design, we separate the square generator into three parts: one's complementer, difference value generator (DVG), and asynchronous adder. In the first stage, we have to generate the eight one's complement numbers of  $|N|_{32}^2: |D|_{32} + |D|_{28} + \dots + |D|_8 + |D|_4$ , and then we store the these values in each time of recursive scheme in order to produce all of the difference values in the next stage. In the second stage, DValues are generated by DValue generator (DVG) and passed to Fast Algorithm to check if they can be combined with others. If the DValue can be combined, it will be ignored in the next stage, and one operation can be omitted. In the final stage, an asynchronous adder adds up the eight DValues to produce a square number (Figure 13).

In the previous two stages, there are two matched-delays in the four phase control path. Length of the delays depends on the logic in their stages. To make sure the delay

length correct is very important. If delay length is too long, whole system will be slowed down; on the contrary, the delay length will be short. In the final stage, our design is used the local four-phase counter and complete detection to control last stage.

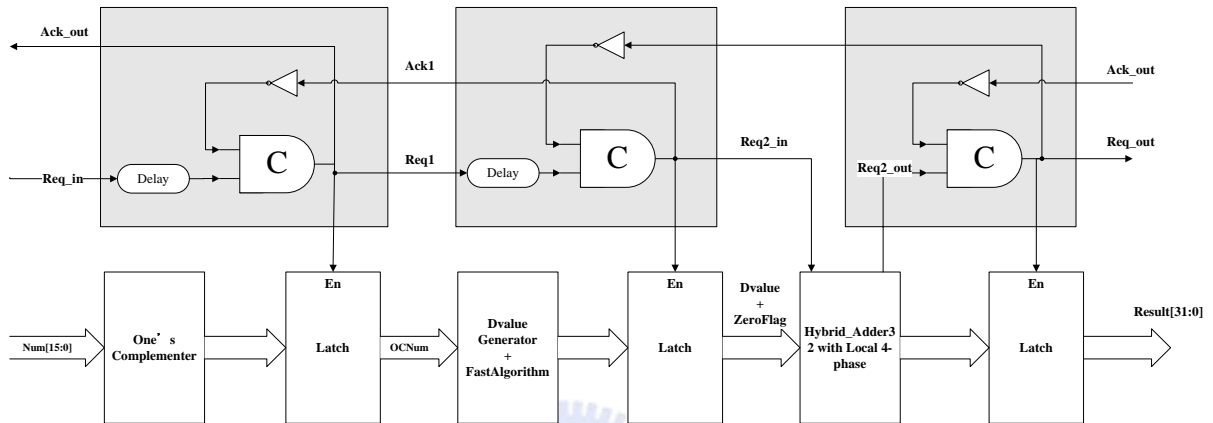


Figure 13 Architecture of Square Generator with Fast Algorithm and Asynchronous Adder

## 4.2 One's Complementer

Based on square generator with recursive scheme ((2) in p.p 17), one's complementer generates seven  $|N^*|$  values in the first stage. Each of the recursion produces one  $|N^*|$  that effects the DValue in the next stage (Figure 14). Figure 15 illustrates all of the one's complementers are composed of XOR gates. Because there are seven times to do recursive scheme, there are seven one's complementers whose size are 2, 4, ... , and 14 bits. As the result, the total gate counts are 56 XOR gates.

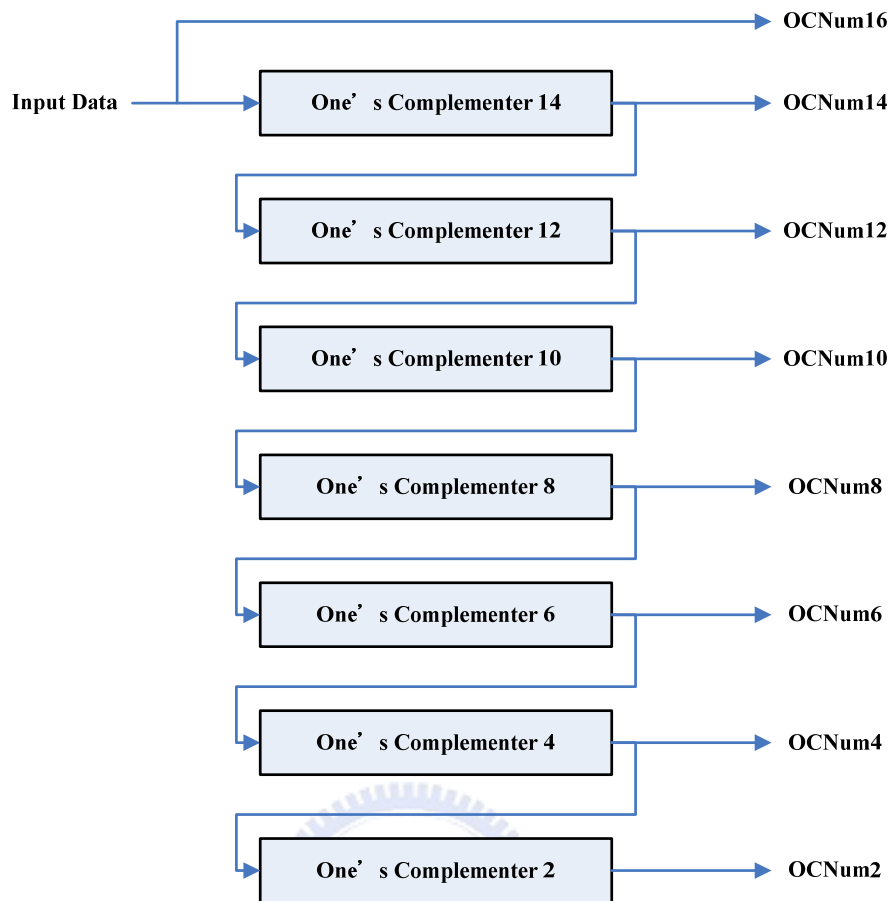


Figure 14 The Flow Path of One's Complementer

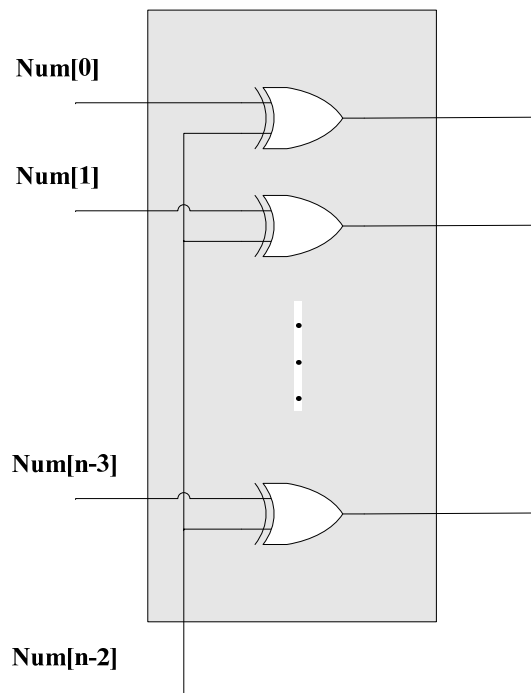


Figure 15 Gate level of One's Complementer



### 4.3 DValue Generator and Fast Algorithm

As One's complement numbers are generated in the first stage, we produce the DValues and use Fast Algorithm to combine the DValues in the second stage. The stage is composed (Figure 16) of five parts: Box Flag, Ball Flag, Zero Flag, DVG and Fast Algorithm. According to Fast Algorithm, Box Flag and Ball Flag are used to identify if the Ball can be put into Box. Zero Flag is equal to one when the DValue is zero.

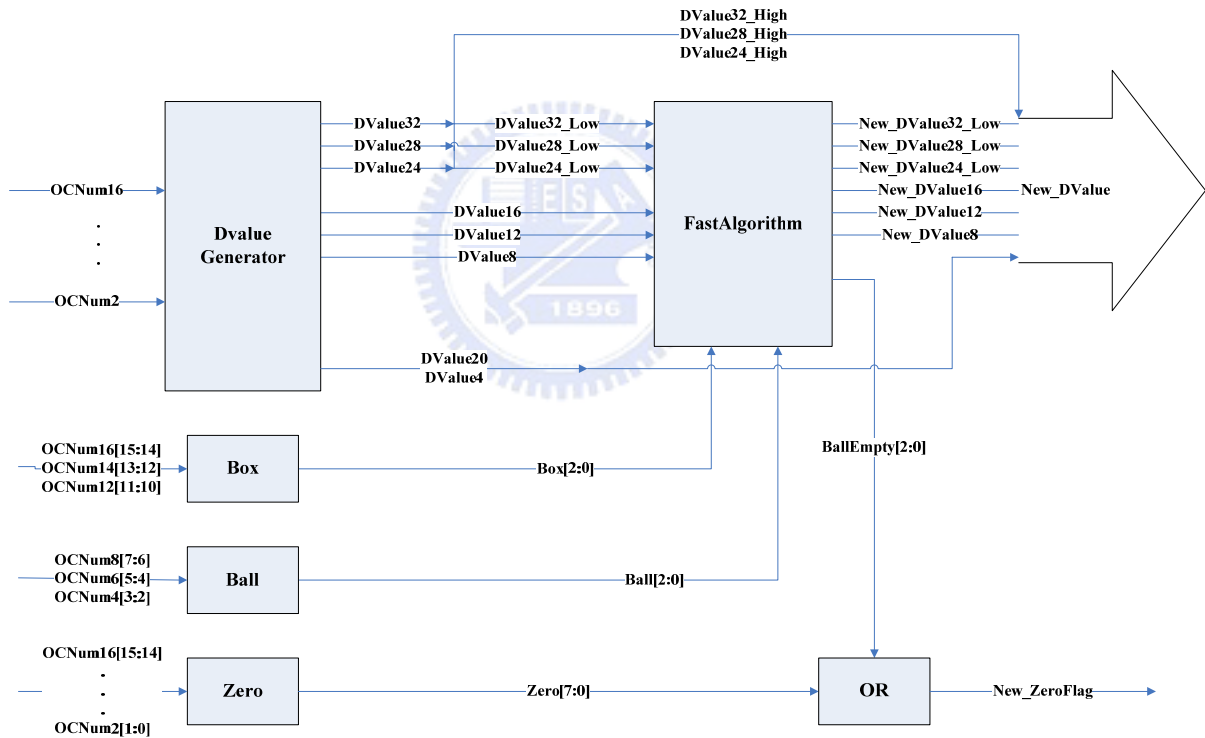


Figure 16 Architecture of DValue generator and FastAlgorithm

#### 4.3.1 Implementation of DValue Generator

In [6] and Table 5, DVG is partitioned into DVG\_H and DVG\_L. Figure 15(a) and 15(b) shows the hardware implementation of DVG\_H and DVG\_L respectively.

In order to save hardware cost, only the DVG\_H is implemented with 4-to-1 multiplexers.. Because the NOR gate is one of the basic gates, the area and delay of NOR gate much less than multiplexer. The DVG\_L is implemented by using (n-2) NOR gates and one inverter.

### DVG\_H

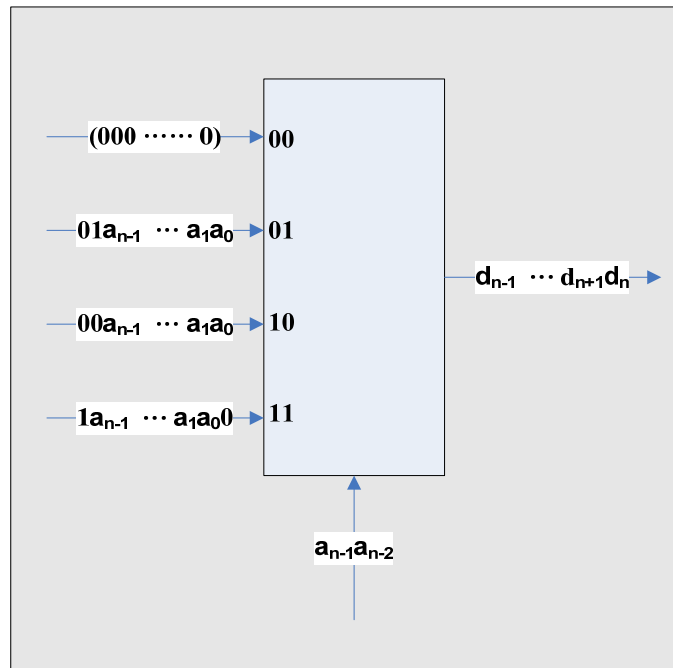


Figure 17(a) First Half of DValue Generator

### DVG\_L

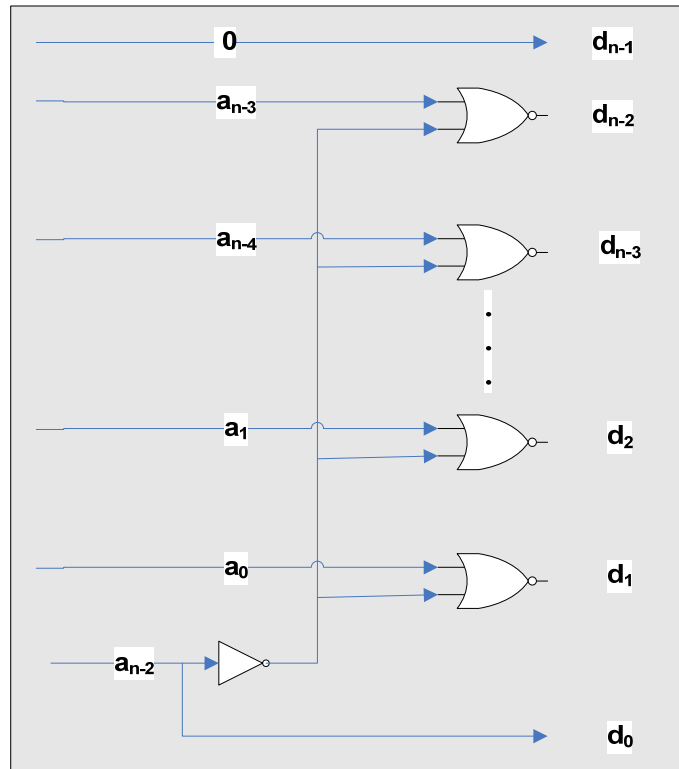


Figure 17(b) Second Half of DValue Generator

### 4.3.2 Implementation of Fast Algorithm

In the previous chapter, Fast Algorithm has been introduced and it could provide performance speed up about 9% to 11%. In practice, we use the multiplexer to implement the Fast Algorithm, so the hardware implementation of Fast Algorithm may be the critical path in the whole square generator. In order to accelerate the Fast Algorithm, we use the mathematical analysis to find a better approach to design.

n=16

Width of DValue	Probability of Putting Ball into Box	Rate of Reducing Operand Bits
4	0.108535767	0.043371687
8	0.290222168	0.231949805
12	0.354492188	0.424973018
16	0.1875	0.29970549

Table 7(a) Probability of Putting Ball into Box if n = 16

n=20

Width of DValue	Probability of Putting Ball into Box	Rate of Reducing Operand Bits
4	0.074933052	0.018680544
8	0.182693481	0.091089671
12	0.405166626	0.303019799
16	0.354492188	0.353494569
20	0.1875	0.233715418

Table 7(b) Probability of Putting Ball into Box if n = 20

n=24

Width of Dvalue	Probability of Putting Ball into Box	Rate of Reducing Operand Bits
4	0.055988073	0.009616085
8	0.137329817	0.047173447
12	0.324520111	0.167211308
16	0.405166626	0.278353336
20	0.354492188	0.304424392
24	0.1875	0.193221432

Table 7(c) Probability of Putting Ball into Box if n = 24

n=28

Width of DValue	Probability of Putting Ball into Box	Rate of Reducing Operand Bits
4	0.04014175	0.004986249
8	0.094671026	0.02351932
12	0.210719883	0.078524369
16	0.430890083	0.214093841
20	0.405166626	0.251640971
24	0.354492188	0.264201696
28	0.1875	0.163033554

Table 7(d) Probability of Putting Ball into Box if n = 28

We still assume a uniform statistical distribution of the input data whose width varies from 16 bits to 28 bits. The numerical results on probability of putting ball into box is obtained from (I), (II), (III), (IV) and Fast Algorithm by computer program execution, are reported in Table 7(a), 7(b), 7(c) and 7(d). Figure 18(a), 18(b), 18(c) and 18(d) are observed that the probability of putting 4-bits DValue into Box is about 4% to 10%. The rate of reducing operand bits on 4-bits DValue in Figure 18(e) is decreased exponentially by width of input data.

In summary, although we reduce the minimum box and ball to sacrifice a few performance, our design obtains lower cost. Figure 19 shows that the circuit of the Fast Algorithm which is a multiplexer. This multiplexer is based on our algorithm to merge the DValues and output the BallEmpty flag to next stage.

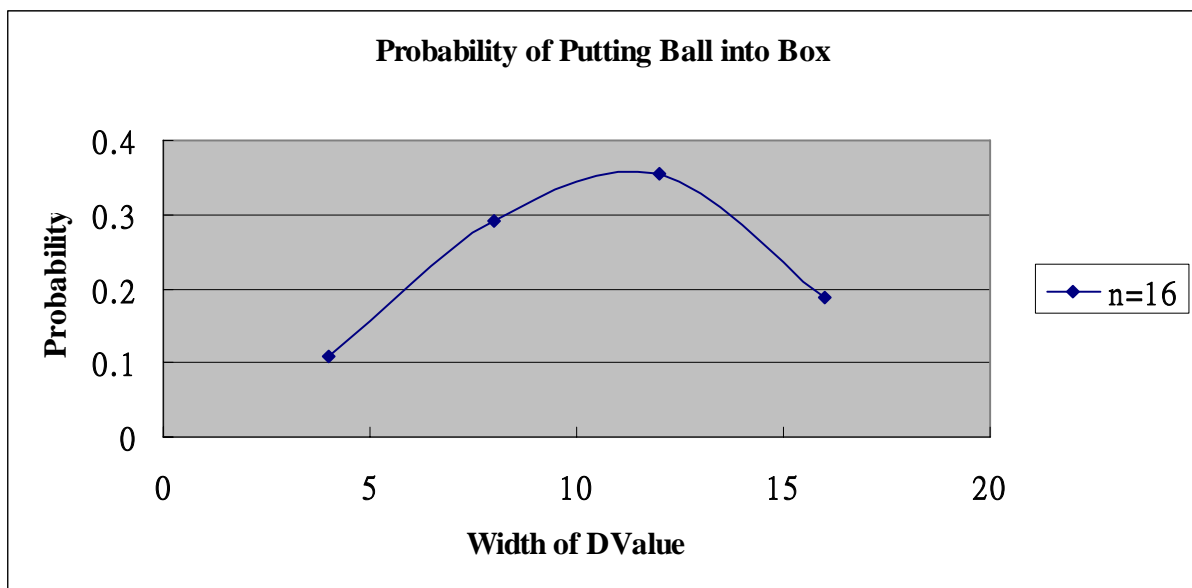


Figure 18(a) Probability of Putting Ball into Box in n=16

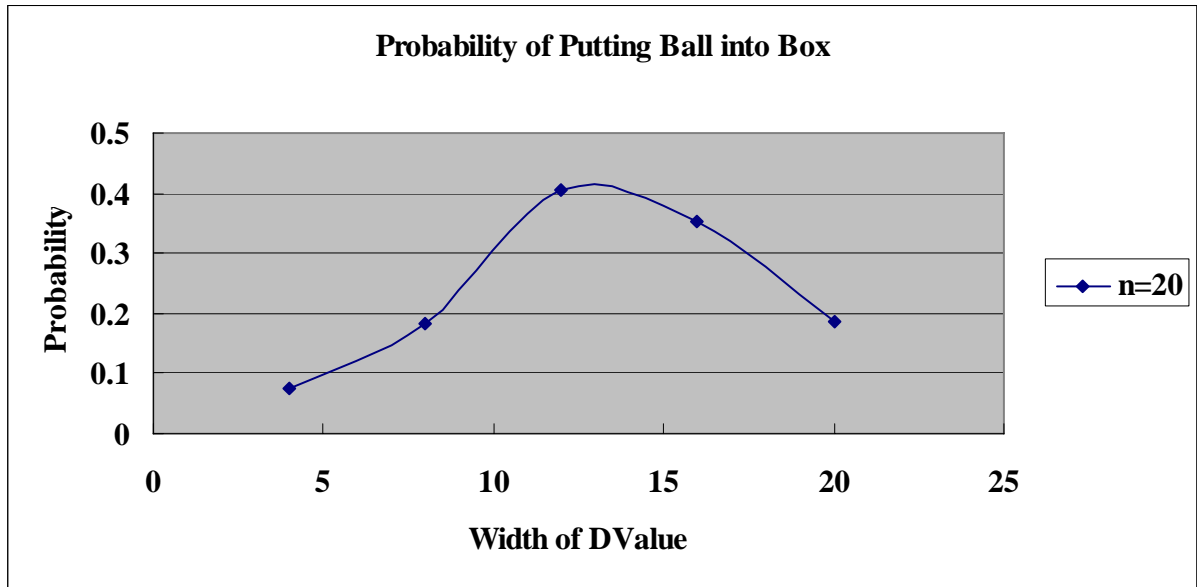


Figure 18(b) Probability of Putting Ball into Box in n=20

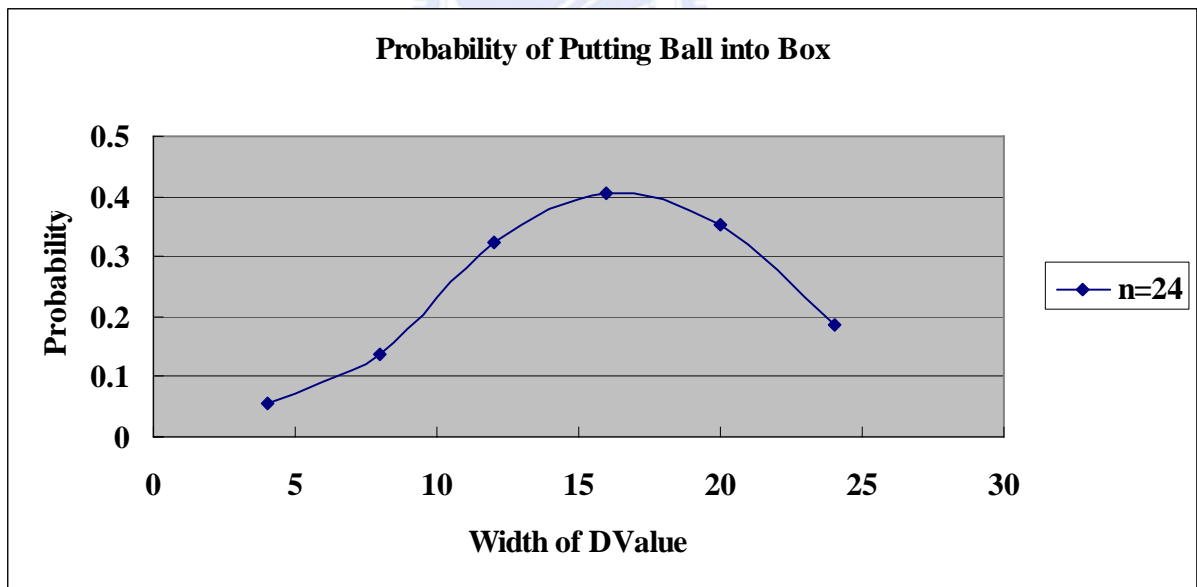


Figure 18(c) Probability of Putting Ball into Box in n=24

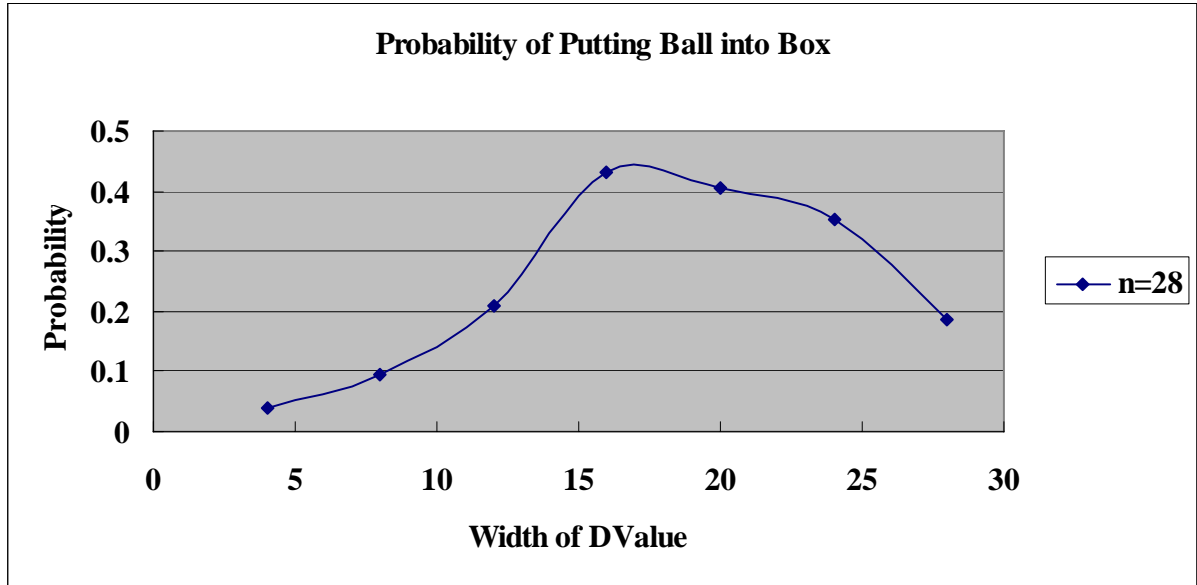


Figure 18(d) Probability of Putting Ball into Box in n=28

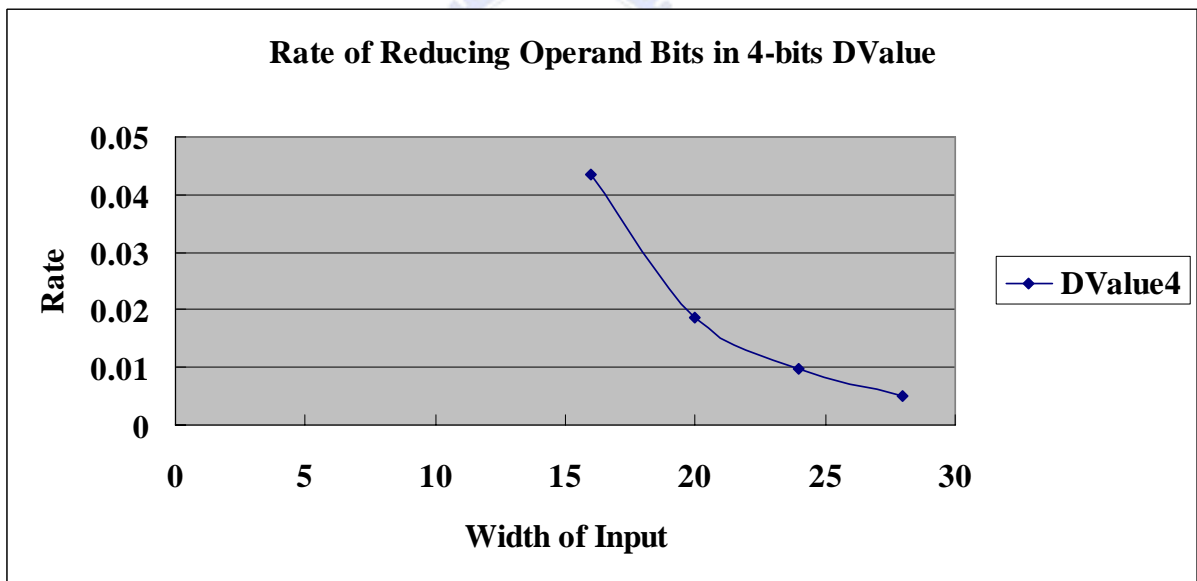


Figure 18(e) Rate of Reducing Operand Bits in 4-bits DValue

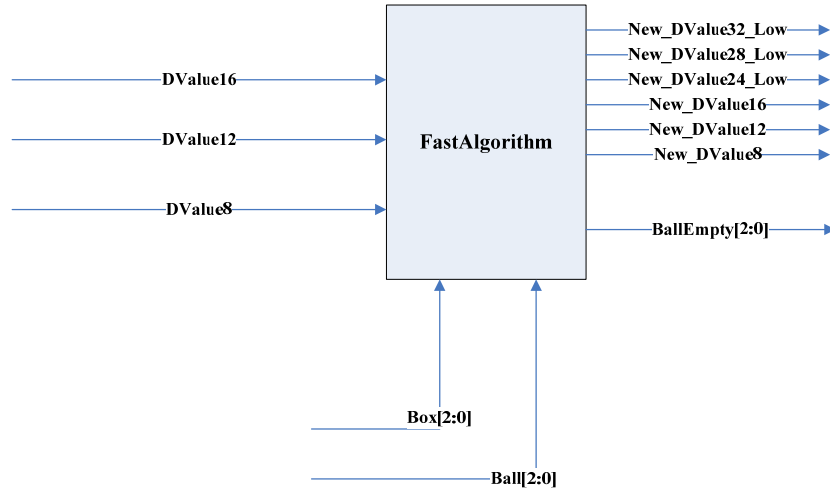


Figure 19 Circuit of Fast Algorithm

### 4.3.3 Hybrid Adder with ZeroPass Scheme

In the final stage, an asynchronous adder adds up the eight DValues to generate a square number. Asynchronous adder has to do seven additions at most so it must have a counter to control the input data. Figure 20(a) and Figure 20(b) are four phase control with counter and 32-bit hybrid adder respectively. Counter in the control path is implemented by local four phase controller. Controller has two part: counter and ZeroPass circuit. First of all, the counter increases until the controller is triggered by global request signal. The Adder will do seven times to generate the square number. When the counter is equal to 7, this controller is stopped and passes the global complete signal. Second, the ZeroPass scheme detects the zero flag if DValue is zero. If DValue is zero, the ZeroPass scheme will send early complete signal without waiting adder complete signal.



In the combinational part, there are two latches to store DValue to add. The above latch in the Figure 20(b) store the temporary sum and the other pass DValue to asynchronous adder. The following section will describe the hybrid adder in detail.

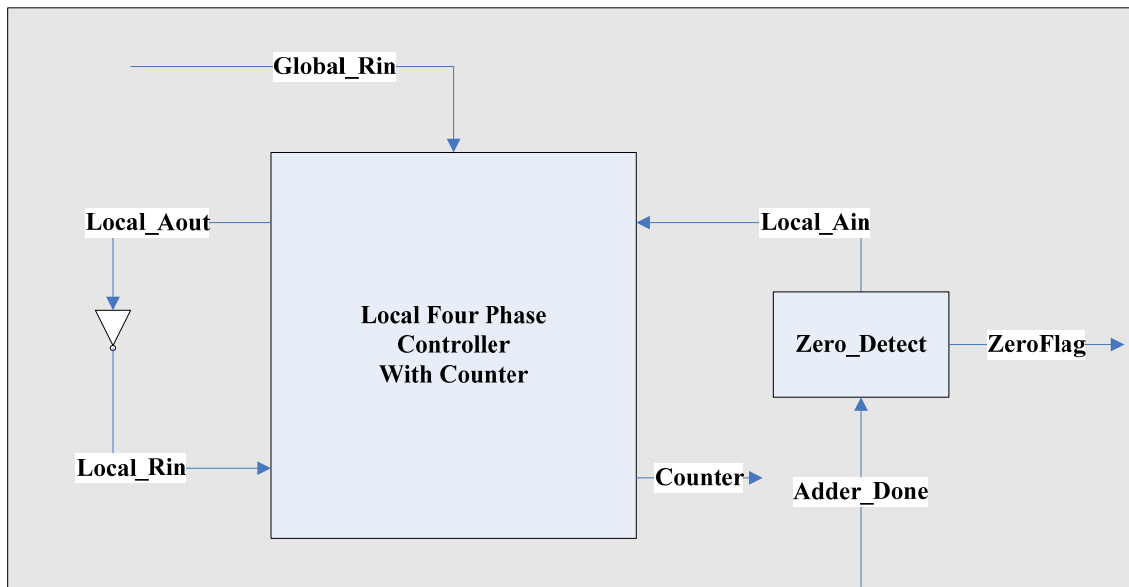


Figure 20(a) Four Phase Control with Counter in 32-bit Hybrid Adder

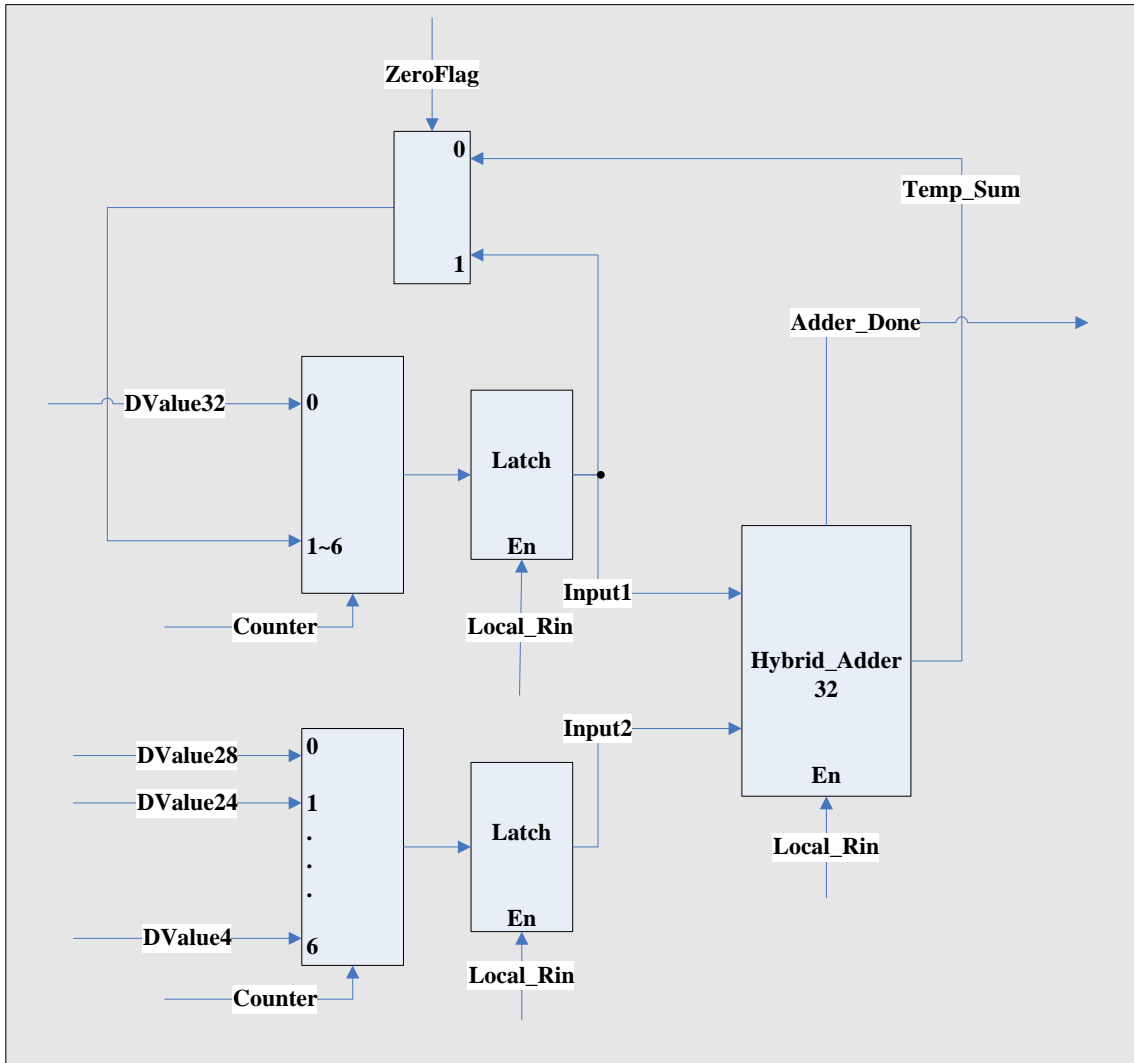


Figure 20(b) Data Path of 32-bit Hybrid Adder

#### 4.4 Implementation of Hybrid Adder

Asynchronous adders in [3, 5-8] can send complete signal if carry chain is finished.

However, square generator has two properties. First, the size of square number from input data is fixed so there is no carry out in the MSB. i.e. size of input data  $N$  is  $n$ -bit, and then size of square number is  $2n$ -bit. Next, the carry in always is equal to zero.

Based on above two characteristics, Figure 21 shows the architecture of specific 32-bits

hybrid adder [7] for square generator. The complete signal can be used to decrease one bit signal due to no carry out.

The Figure 22(a), 22(b) and 22(c) shows architecture of one-bit asynchronous adders, which is modified from the self-time adder in AMULET1 processor [12]. The Figure 22(a) whose operands and the result is bundled data and the carry is dual-rail. Because carry-in signal is equal to zero, the Figure 22(b) specific design reduces the gate count than Figure 22(a). Similarly, Figure 22(c) is slightly modified to detect the 30th bit carry out signal.

The fan-in of complete detector in hybrid adder is so many inputs that implementation of C-element has heavy area cost. In order to solve above problem, Figure 23 illustrates the completion detector which make fan-in put into the AND gate and NAND gate.

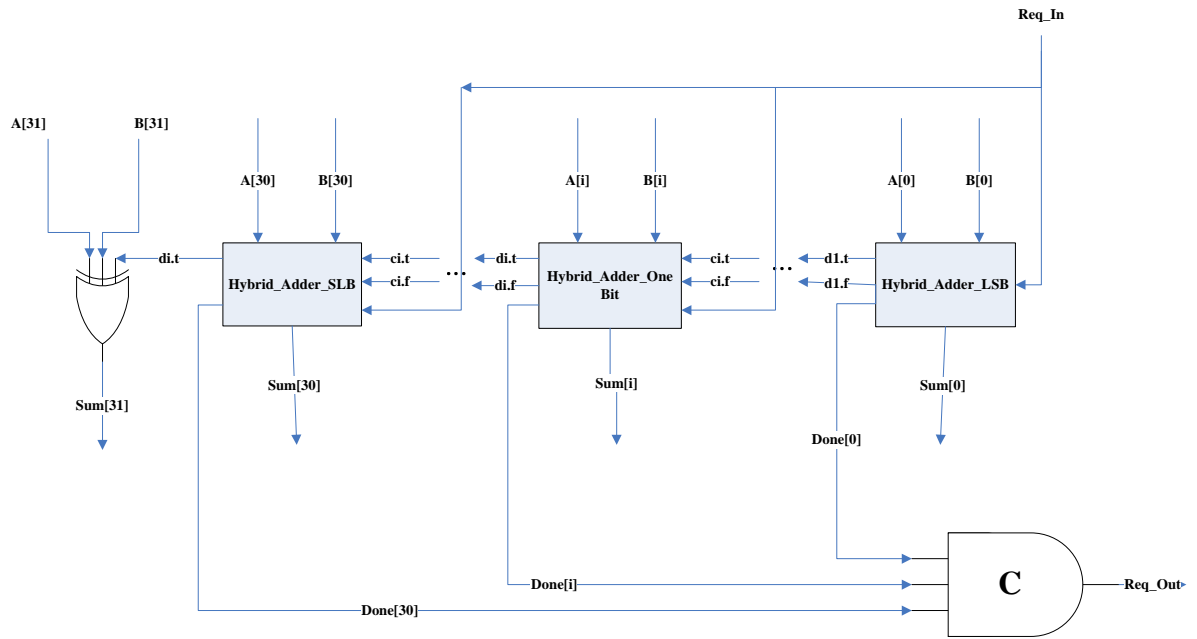


Figure 21 32-bits Hybrid Adder for Square Generator

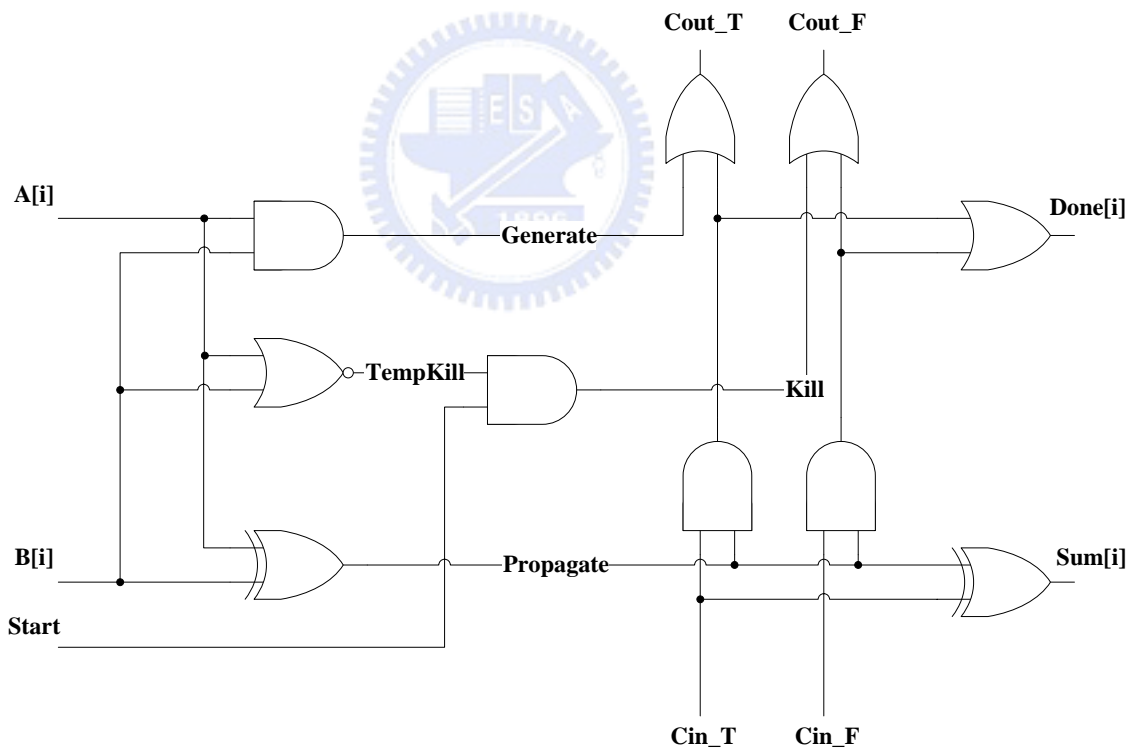


Figure 22(a) 1-bit Hybrid Adder

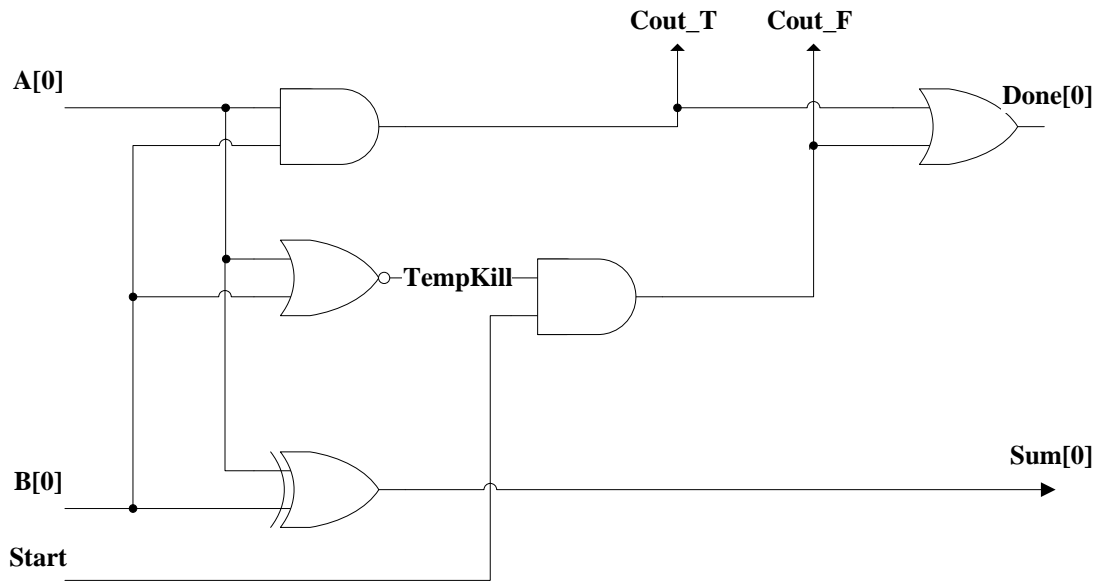


Figure 22(b) 1-bit Hybrid Adder on the LSB

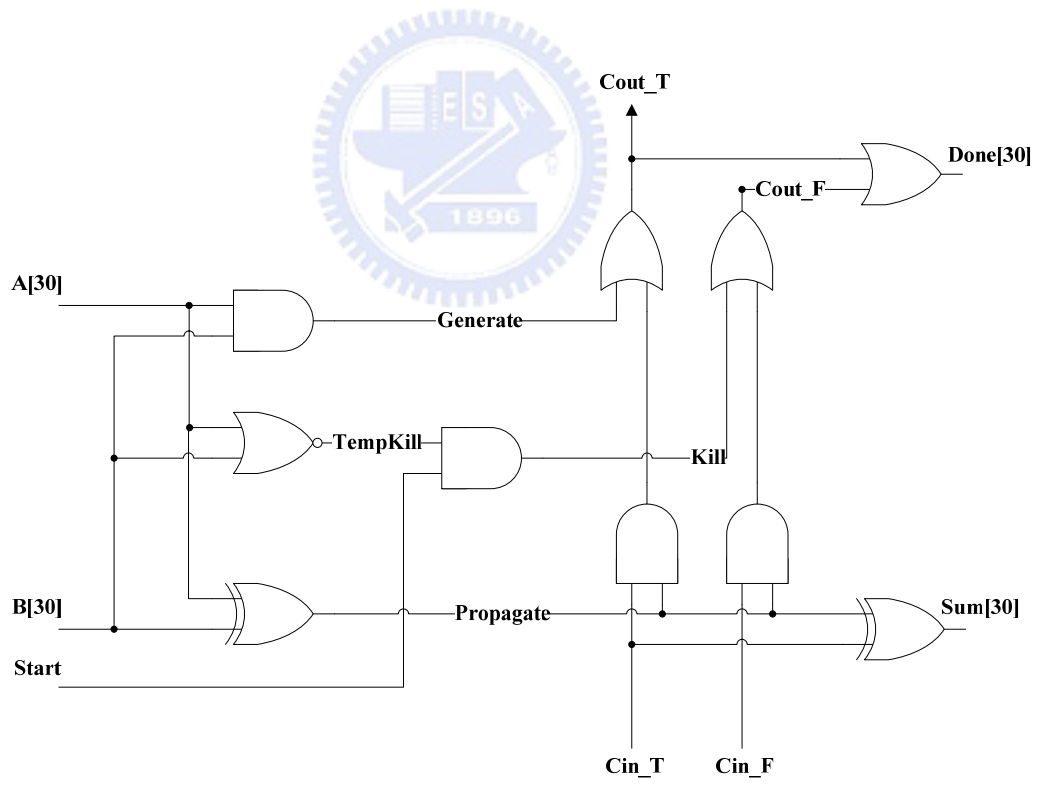


Figure 22(c) 1-bit Hybrid Adder on the 30th Bit

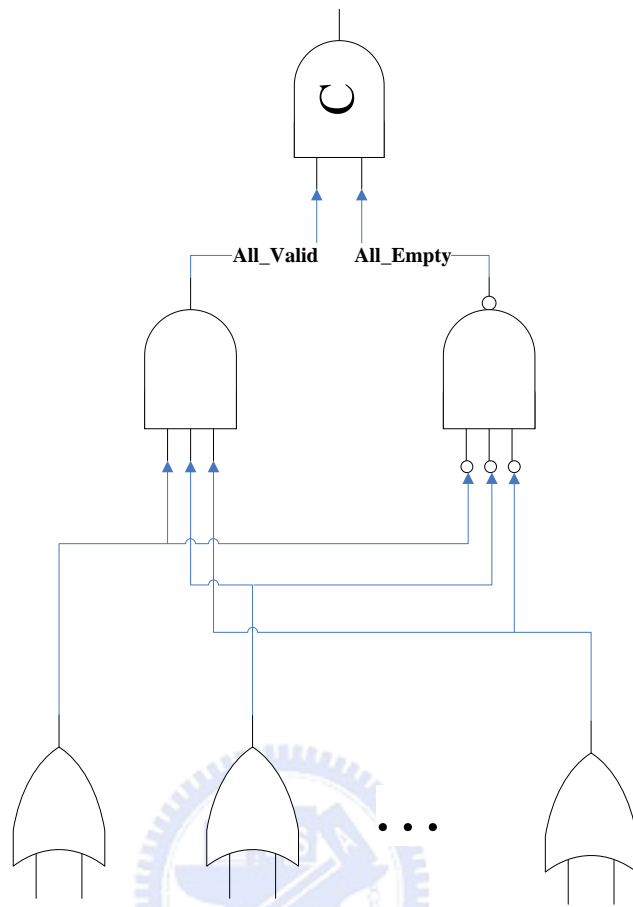


Figure 23 Alternative C-element

# Chapter 5 Simulation

In order to demonstrate the performance of the proposed asynchronous square generator with Fast Algorithm, the architecture presented in the previous chapter is implemented by using standard cell library of TSMC 0.13  $\mu m$  CMOS process.

We use the ModelSim 6.0 simulator to show the waveform and use Design Compiler to synthesize our design to gate-level model. The results of area and latency are described in the following sections.

## 5.1 Timing Simulation

Because the asynchronous square generator with Fast Algorithm has best-case and worst-case performance, the input values are  $(AA55)_h$  and  $(6655)_h$  respectively for the best-case and worst-case. The best-case represents that all of the balls can be put into the boxes; in contrast, the worst-case means that all of the balls can not be put into the boxes.

Figure 24(a) and Figure 24(b) show the waveforms and latencies for these two cases.

Table 8 shows the comparison of the asynchronous square generator with Fast Algorithm and without Fast Algorithm. In the best case, asynchronous square generator with Fast Algorithm is 18.13% faster than that without it; in other words, asynchronous adder with ZeroPass scheme provides speedup computation for the addition time. In the worst case, Fast Algorithm approach is 2.99% slower than that without it. Because Fast Algorithm does not combine with any DValues, it becomes the overhead of the system; that is, asynchronous square generator without Fast Algorithm can generate the DValues

to add directly.

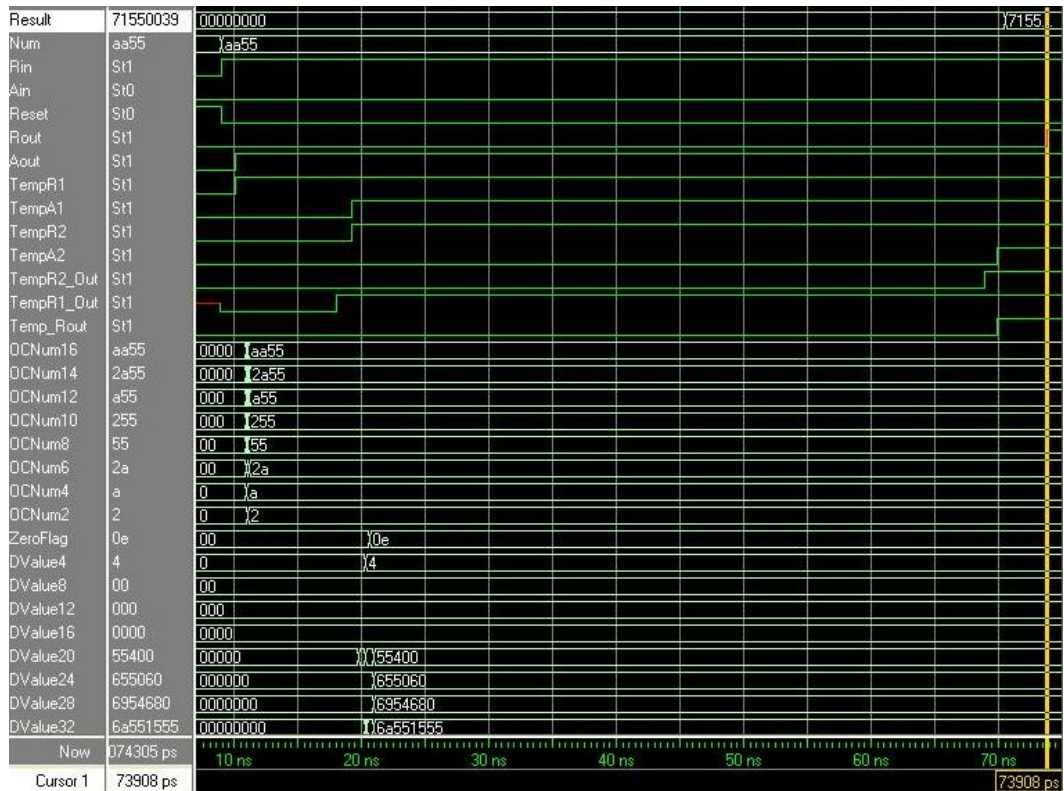


Figure 24(a) Best-case for Fast Algorithm when input value =  $(AA55)_h$

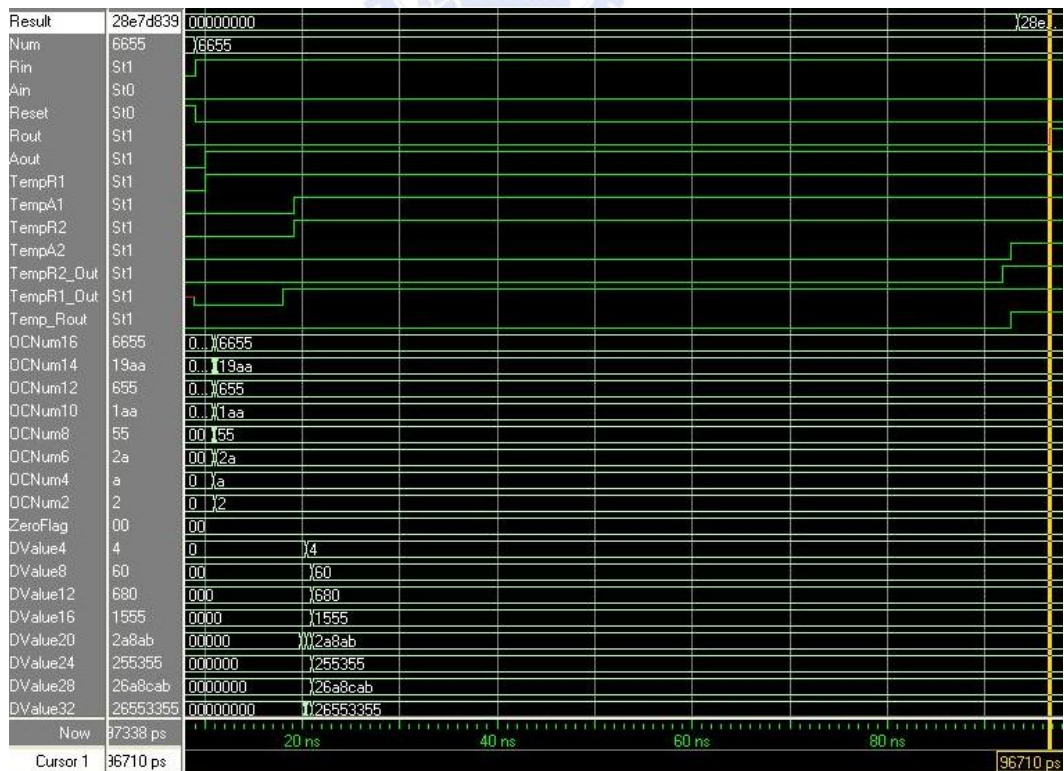


Figure 24(b) Worst-case for Fast Algorithm when input value =  $(6655)_h$



<b>Latency(ns)</b>	<b>Fast Algorithm</b>	<b>Without Fast Algorithm</b>	<b>Improvement Rate</b>
<b>Best Case</b>	64.869	79.232	18.13%
<b>Worst Case</b>	87.671	85.128	-2.99%

Table 8 Latency between with Fast Algorithm and without Fast Algorithm

## 5.2 Area Simulation

The area of asynchronous square generator with Fast Algorithm and without improving is shown in Table 8. The area of the asynchronous square generator with Fast Algorithm is  $175963.23 \mu m^2$  and the area of that without it is  $132179.82 \mu m^2$ . The area overhead with Fast algorithm is about 133.12%.

	<b>Area(<math>\mu m^2</math>)</b>
<b>FastAlgorithm</b>	175963.23
<b>Without FastAlgorithm</b>	132179.82

Table 9 Area of between With Fast Algorithm and Without Fast Algorithm

## Chapter 6 Conclusion and Future Works

In the design, we implement the square generator with Fast Algorithm in asynchronous circuit design. The differential values (DValues) can be combined by Fast algorithm; thus, square generator reduces the additions and improves the performance. The numerical result of analysis can improve about 8% ~ 11%.

Based on recursive scheme of square generator, our design is divided into three asynchronous pipeline stages. The one's complement numbers is produced in the first stage. Subsequently, the DValues are generated by DValue generator (DVG) and then they pass to Fast Algorithm to combine DValues. After finishing above operation, this stage send the zero flag and DValues are sent to the next stage. Finally, the hybrid adder with ZeroPass can accelerate the addition to produce the square number.

The results of simulation using standard cell library in TSMC 0.13  $\mu m$  CMOS process demonstrate that our design is 18.13% faster in the best-case than asynchronous square generator without Fast Algorithm.

In the future, improving Fast Algorithm plays an important role on the proposed asynchronous square generator. Furthermore, using faster asynchronous pipeline can also speed-up the synchronization with each stage.

# References

- [1] H. Ling, “*An Approach to Implementing Multiplication with Small Tables,*” IEEE Transactions on Computers, Vol. 39, No. 5, pp. 717-718, May 1990
- [2] Tien Chi Chen, “A Binary Multiplication Scheme Based on Squaring,” IEEE Transactions on Computers, Vol. 20, No. 6, pp. 678-680, June 1971
- [3] Michael J. Schulte, Louis Marquette, Shankar Krithivasan, E. George Walters III, and John Glossner, “*Combined Multiplication and Sum-of-Squares Units,*” Proceedings of the Application-Specific Systems, Architectures, and Processors, 2003
- [4] E. George Walters III, Jason Schlessman, and Michael J. Schulte, “*Combined Unsigned and Two's Complement Hybrid Squarers,*” Vol. 1, pp. 861-866, The Thirty-Fifth Asilomar Conference on Signals of Systems and Computers, 2001
- [5] Chin-Long Wey and Ming-Der Shieh, “*Design of a High-Speed Square Generator*”, Vol.47,No. 9, pp. 1021-1026, IEEE Transactions on Computers, Sep. 1998
- [6] Wei-Chang Tsai, Ming-Der Shieh, Wen-Chin Lin and Chin-Long Wey, “*Design of Square Generator with Small Look-up Table,*” pp. 172-175, IEEE Asia Pacific Conference on Circuits and Systems, 2008
- [7] Jens Sparso and Steve Furber, “*Principles of Asynchronous Circuit Design,*” London, 2001
- [8] S. Hauck, “*Asynchronous design methodologies: an overview,*” Proceedings of the IEEE, Vol. 83, Issue 1, pp. 69-93, Jan. 1995
- [9] David Duarte, Vijaykrishman Narayanan and Mary Jane Irwin, “*Impact of Technology*

*Scaling in the Clock System Power;*” pp. 59, IEEE International Computer Society Annual Symposium on VLSI, 2002

[10] Fu-Chiung Cheng, Stephen H. Unger, Michael Theobald and Wen-Chung Cho, “*Delay-Insensitive Carry-Lookahead Adders,*” Tenth International Conference on VLSI Design Proceedings, pp. 322-328, 1997

[11] Alessandro De Gloria and Mauro Olivieri, “*Statistical Carry Lookahead Adder,*” IEEE Transactions on Computers, VOL. 45, NO. 3, pp. 340-347, March. 1996

[12] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple, “*AMULET1: An Asynchronous ARM Microprocessor,*” Vol. 46, No. 4, pp. 385-398, IEEE Transactions on Computers, VOL. 46, NO. 4, April 1997

