國立交通大學

資訊科學與工程研究所

碩士論文

以固定式暫存器分配方法為基礎之
低功耗低面積資料格式轉換器設計
Low-Power Area-Efficient Data Format Converter Design Using Static Register Allocation

研究生：王得安

指導教授：范倫達　博士

中華民國九十七年七月

以固定式暫存器分配方法為基礎之

低功耗低面積資料格式轉換器設計

Low-Power Area-Efficient Data Format Converter Design Using Static Register Allocation

研究生：王得安　　　　Student：Te-An Wang

指導教授：范倫達博士　　　　Advisor：Dr. Lan-Da Van

國立交通大學
資訊科學與工程研究所
碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年七月

以固定式暫存器分配方法為基礎之
低功耗低面積資料格式轉換器設計

學生：王得安　　　　　　　　指導教授：范倫達 博士

國立交通大學
資訊科學與工程研究所

摘　　要

在本篇論文之中，我們針對資料格式轉換器(DFC)的架構設計提出了一項低功率低面積的暫存器分配之演算法。我們所提出的固定式暫存器分配方法不只是具有最小的功率消耗與暫存器資料寫入次數，而且在 DFC 的設計中使用的面積也不會太大。從 16 位元的 3x3, 4x4, 16x16 的矩陣轉置器與 IIR filter 的 DFC 測試項目中，在 0.18um CMOS 製程上與 SSRA 的方法比較來看，功率消耗分別減少了 27.4%, 45.3%, 50.2 %以及 25.7%，而晶片面積分別減少了 44.6%, 51%, 53.9%以及 38%；另外從 16 位元的 1-D DWT, Zigzag scanner, 4x4 par-transposer 的 2-D DFC 測試項目中，在 0.18um CMOS 製程上與 SSRA 的方法比較來看，功率消耗分別減少了 5.3%, 13.6% 以及 16.1 %，而晶片面積分別減少了 28.9%, 33.6% 以及 26.4%，所提出的固定式暫存器分配方法有著最低的功率消耗與較低的晶片面積在這幾種方法之中。最後利用 SRA 方法設計 WiMAX 傳輸端之的 Interleaver，以驗證在本文所提出的想法。

Low-Power Area-Efficient Data Format Converter Design Using Static Register Allocation

Student：Te-An Wang Advisor：Dr. Lan-Da Van

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

ABSTRACT

In this thesis, we explore one low power and area efficient register allocation algorithm for data format converter (DFC) architecture designs. The proposed static register allocation (SRA) approach not only minimizes the power and number of register transitions, but also achieves a comparable area cost for DFC designs. From the implementation results of 16-bit 3x3, 4x4, 16x16 transposer, and IIR filter benchmarks using 1-D SRA, the power consumption can be alleviated by 27.4%, 45.3%, 50.2% and 25.7% respectively, compared with the SSRA design in 0.18 um CMOS process. The core area reduction by 44.6%, 51%, 53.9% and 38% can be achieved for the same cases. From the implementation results of 16-bit 1-D DWT, Zigzag scanner and 4x4 par-transposer benchmarks using 2-D SRA, the power consumption can be alleviated by 5.3%, 13.6% and 16.1%, respectively, compared with the SSRA design in 0.18 um CMOS process. The core area reduction by 28.9%, 33.6% and 26.4% can be achieved for the same cases. Thus, the proposed SRA-based design has lowest power consumption and cost effective among the several approaches. Finally, we implement

the interleaver using SRA for WiMAX system.

誌謝

對於這篇論文能順利完成，首先感謝指導教授 范倫達老師在這一年多以來的悉心教導與鼓勵，雖然只有一年多的相處時間，但老師的用心與付出，讓我感到日子過的真是充實又幸福，無奈時間總是不曾停下腳步，只能將一切深深地銘記在心中，永遠不會忘記。亦要感謝 鍾崇斌老師在我的碩一研究生活中所提出的各種指導與建議，讓我成長、茁壯。在此先對二位師長獻上無以倫比的感激。

其次是感謝 VIP Lab 的同學與學弟們，每當我的研究遇到挫折、難題或是不如意需要幫忙時，你們都可以適時地給我建議與幫助，讓我有持續做下去的信心與毅力。一起打球、拍照、吃飯的回憶，是我想忘也忘不了的，當然也要感謝 System Lab 的學長姐及同學們，當我正在處極度失意的時候，你們熱心、無私地提供協助，讓我可以繼續我的碩士學業，也感謝所有在我的研究生生活中，帶給我溫馨與歡笑的每一個人，你們簡單的一句話或是無心的一個小動作，也都豐富了我這兩年的生活與回憶。

最後要感謝家人和親友們的關心、支持與鼓勵，以及從小讓我養成自動自發的習慣與獨立自主的精神。我想沒有他們，我今天也不會有這個機會，完成此篇論文。在此將這篇論文獻給我最敬愛的父母親。

Contents

List of Tables

Chapter 5

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 1

Introduction

DATA format converters (DFCs) [2-9] has been widely used in digital signal processing (DSP), image, and video processing such as matrix transposers [2, 7-8], serial to parallel converter [2], digital filter [2, 9], one-dimensional (1-D) discrete wavelet transform (DWT) [4,12], two-dimensional (2-D) discrete wavelet transform (DWT) [23-26], JPEG image compression [21-22] and interleaver for WiMAX [27-29]. The DFC consisting of data registers and control unit serves to permute the data from one format to another. The registers of DFC read in data from the input bus and place data on the output bus. The registers of the conventional DFC communicate with each other via dedicated interconnections. Due to the largely growth of low power demands for portable multimedia-communication designs, the parallel, folded, pipelined architectures have been widely applied to these computation engines to save power [1, 2]. In the above low-power architectures, the DFC plays an important role of implementing these computations. However, few papers [7-8] focus on improving power consumption for DFC design. Thus, we are motivated to propose a lower-power DFC design. Generally, the power consumption of a CMOS VLSI circuits can be formulated as $P = \alpha C_L V_{dd}^2 f$, where α , C_L, V_{dd}, f denote the number of transitions, the effective load capacitance, the power supply voltage, and the clock frequency of the circuits, respectively [1,18]. At the algorithm and architecture level of DFC design, under the same operating frequency and supply voltage, we have little room to directly reduce capacitance for power saving. Thus, an effective way of reducing the power consumption of the DFC is by alleviating the number of register transitions. This is equivalent to reducing the number of variables that move from one register to another.

In brief review, many register allocation techniques [2]-[9],[19-20] have been proposed to design DFC's under the constraint of the minimum number of registers, and other register allocation schemes [10-11,15-17] are applied to the register file used in the instruction-based processor and the multi-dimensional signal processing systems. The forward-backward register allocation (FBRA) proposed in [2-3] is the pioneer systematic work to solve the register allocation problem for DFC designs. The FBRA scheme results in a serial interconnection of registers; thereby, the number of transitions is increased. A two dimensional (2D) extension of the FBRA scheme has been proposed in [4], where multiple data are input and output at the same time. A video data format converter based on FBRA is proposed in [19]. The design methodology presented in [5-6] for implementing 2-D DFC architecture results in a small area. A general framework for synthesis of data format converters is proposed in [20]. All of these schemes require larger number of register transitions such that the larger power consumption is incurred. The sequencer-based data path synthesis scheme [9] is the technique that tries to reduce the number of memory/register accesses by exploiting the pattern properties. Next, a semi-static register-allocation scheme has been proposed by SSRA to improve the number of transitions and power consumption for DFC architecture [7-8]. However, as mentioned in [8], since many tri-state buffers are applied to this semi-static register allocation-based DFC design, the buffers own the large portion of area and power consumption. On the other hand, using SSRA scheme, the last input variable is required to transit from the fist register to the last register. It is observed that the transition power has not been minimized using SSRA scheme. Thus, the SSRA scheme results in large control area and power consumption.

1.1 Motivation

The variables move every cycle in the FBRA approach. It costs large power consumption. In this thesis, we primarily concentrate on reducing the number of transitions and power consumption with a slightly penalty of the increased area cost for DFC designs. We propose a new register-allocation scheme, called static register allocation (SRA), where each variable is allocated to the fixed register at each iteration. We verify the correctness of this approach by implementing and experimenting with seven examples including 3x3 transposer, 4x4 transposer, 16x16 transposer, IIR filter, 1-D DWT, 4x4 par-transposer and Zigzag Scanner. Although we need slightly larger number of multiplexers than that of SSRA, the SRA scheme results in much less control area due to zero tri-state buffer. From the architecture analysis and post-layout simulation results, the proposed design using SRA has lowest register transition and power consumption with satisfactory area cost. Other register allocation schemes [10-11] are applied to register file and high level synthesis with the application of embedded processor and computer instead of DFC designs. Thus, this kind of register allocation schemes [10-11] always work with tool chain such as compiler.

1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 introduces the fundamentals of 1-D and 2-D register allocation schemes including the forward backward register allocation (FBRA), semi-static register allocation (SSRA), and two-dimensional (2-D) register allocation. The proposed 1-D SRA algorithm and architecture is presented in Chapter 3. The proposed 2-D SRA algorithm and architecture is presented in Chapter 4. Four 1-D and four 2-D DFC benchmarks using three different approaches are compared in terms of transition activity, power consumption, and area in Chapter 5. Meanwhile, we implement the interleaver for WiMAX system using the SRA approach and compare the power consumption with the conventional memory-based design. Finally, the conclusion and the future work are remarked in Chapter 6.

Chapter 2

Fundamentals of 1-D and 2-D Register Allocation

In this chapter, we will introduce the fundamentals of 1-D and 2-D register allocation schemes including forward backward register allocation (FBRA), semi-static register allocation (SSRA), and two-dimensional (2-D) register allocation for data format converter (DFC) designs.

For convenience of demonstrating the difference of the FBRA and SSRA schemes, we use matrix transposer as an example.

Assume that X_{3x3} and Y_{3x3} denote the input matrix and the transpose matrix of X_{3x3}, respectively, where the relationship is expressed in (2.1).

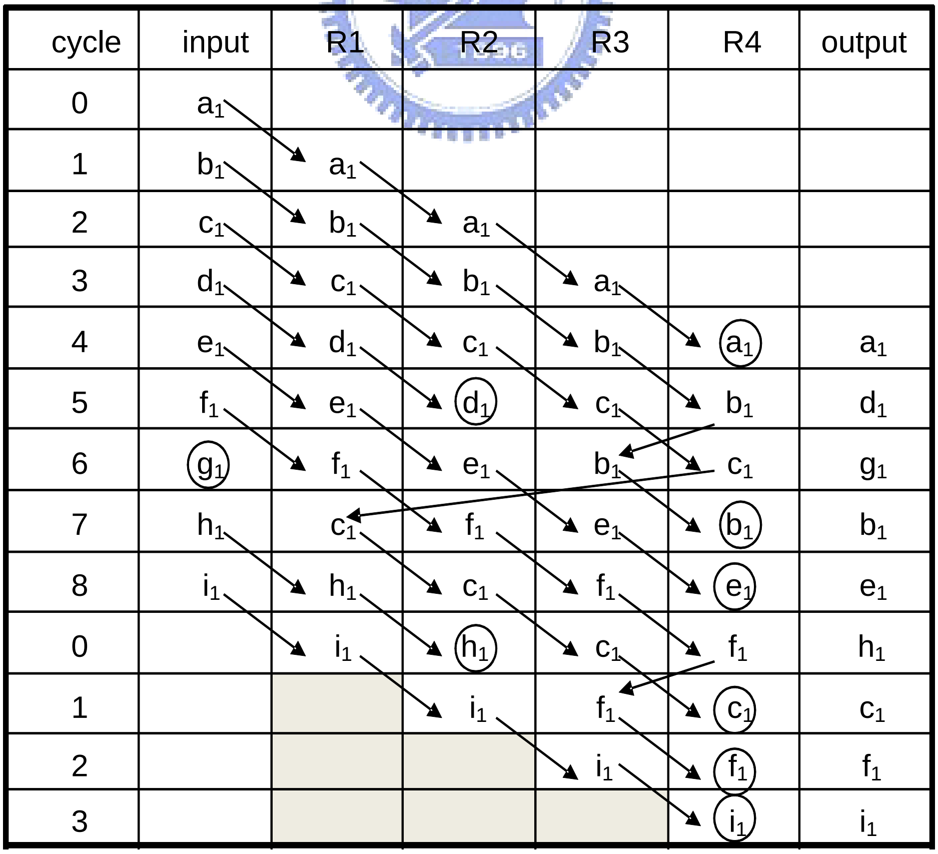
$$\mathbf{Y}_{3x3} = \mathbf{X}_{3x3}^{T}. \tag{2.1}$$

Let $\mathbf{X}_{3x3} = \begin{bmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{bmatrix}$, we obtain $\mathbf{Y}_{3x3} = \begin{bmatrix} a_1 & d_1 & g_1 \\ b_1 & e_1 & h_1 \\ c_1 & f_1 & i_1 \end{bmatrix}$, where the input and output data sequences are scanned in $\{a_1,b_1,c_1,d_1,e_1,f_1,g_1,h_1,i_1\}$ and $\{a_1,d_1,g_1,b_1,e_1,h_1,c_1,f_1,i_1\}$, respectively.

2.1 Forward-Backward Register Allocation (FBRA)

The FBRA scheme presented in [2-3] uses a single data path having a pipeline-like serial interconnection of registers. Thus, the control method of FBRA is much simple such that the control overhead is low. However, the variables storing in registers always move forward or backward every cycle such that larger power consumption is incurred. Figure 2.1 shows the forward-backward register allocation table for the 3x3 transposer, where arrow denotes the register transition. In this case, the number of transitions for each iteration is 36.

Figure 2.2 shows the architecture of 3x3 transposer using FBRA. The 3x3 transposer architecture of FBRA uses one three-to-one multiplexer and two two-to-one multiplexers to control the DFC data flow.



cycle	input	R1	R2	R3	R4	output
0	a_1					
1	b_1	a_1				
2	c_1	b_1	a_1			
3	d_1	c_1	b_1	a_1		
4	e_1	d_1	c_1	b_1	a_1	a_1
5	f_1	e_1	d_1	c_1	b_1	d_1
6	g_1	f_1	e_1	b_1	c_1	g_1
7	h_1	c_1	f_1	e_1	b_1	b_1
8	i_1	h_1	c_1	f_1	e_1	e_1
0		i_1	h_1	c_1	f_1	h_1
1			i_1	f_1	c_1	c_1
2				i_1	f_1	f_1
3					i_1	i_1

Figure 2.1 Allocation table for 3x3 transposer using FBRA.

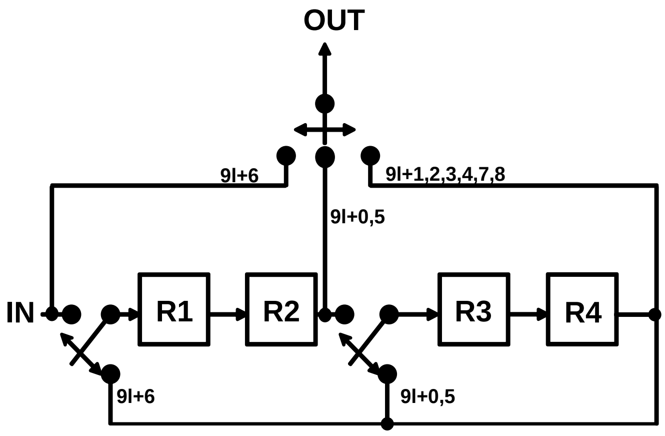
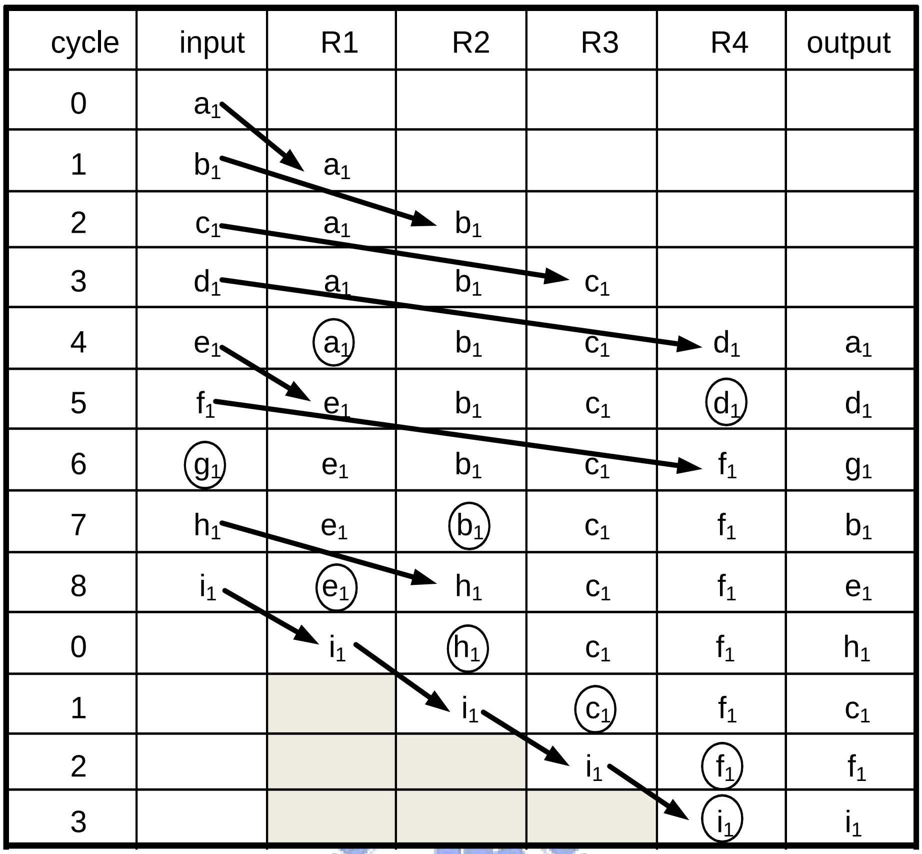


Figure 2.2 Architecture of the 3x3 transposer using FBRA.

2.2 Semi-Static Register Allocation (SSRA)

The semi-static register allocation (SSRA) scheme [7-8] has been proposed to improve the number of transitions and power consumption for DFC architecture. In the same case, Figure 2.3 shows the SSRA-based allocation table for the 3x3 transposer, where arrow denotes the register transition. From Figure 2.3, the number of transitions for each iteration is 11. Using the SSRA scheme, the last input variable is required to transit from the fist register to the last register. It is observed that the transition power has been reduced; thus, the SSRA scheme results in less power consumption.

Figure 2.4 shows the architecture of *N*x*N* transposer using SSRA. The global inter-register buses and I/O buses, driven by tri-state buffers, are used in the SSRA scheme to transfer data between any two registers and the I/O. Since many tri-state buffers are applied to the SSRA-based DFC design, the buffers own the large portion of area. Thus, the SSRA scheme results in large control area.



cycle	input	R1	R2	R3	R4	output
0	a_1					
1	b_1	a_1				
2	c_1	a_1	b_1			
3	d_1	a_1	b_1	c_1		
4	e_1	a_1	b_1	c_1	d_1	a_1
5	f_1	e_1	b_1	c_1	d_1	d_1
6	g_1	e_1	b_1	c_1	f_1	g_1
7	h_1	e_1	b_1	c_1	f_1	b_1
8	i_1	e_1	h_1	c_1	f_1	e_1
0		i_1	h_1	c_1	f_1	h_1
1			i_1	c_1	f_1	c_1
2				i_1	f_1	f_1
3					i_1	i_1

Figure 2.3 Allocation Table for 3x3 transposer using SSRA.

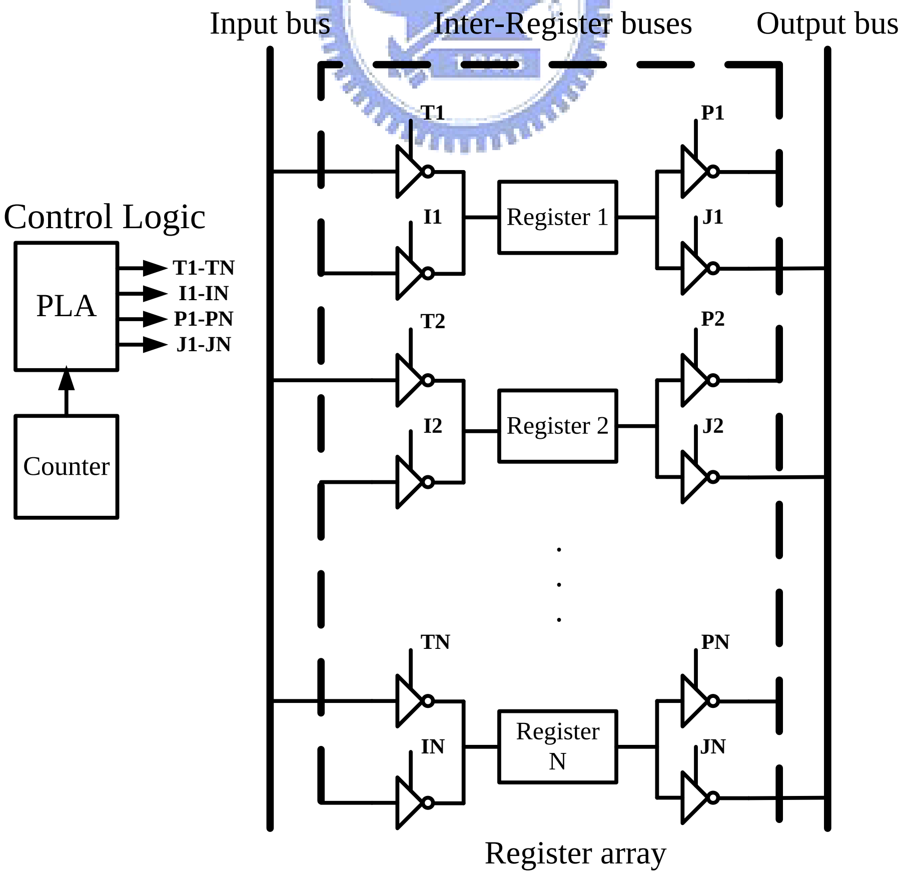


Figure 2.4 Architecture of the *N*x*N* transposer using SSRA.

2.3 Two-Dimensional Register Allocation

One of the 2-D register allocation schemes [4] is the extension of the FBRA scheme. It has been proposed to reduce the area of the 2-D DFC. Since 2-D register allocation scheme is used for multiple inputs and multiple outputs, it uses multiple data paths of vertical pipeline of registers and gated clocks to reduce control area overhead and power consumption.

According to [4], the interface format of a DFC is defined as $(m_1,d_1)\rightarrow(m_2,d_2)[N]$. This general format can be used to describe both word-level as well as bit-level converters. For bit-level converters, primarily used in signal processing systems, this can be read as: d_1 bits of m_1 words are input every input clock cycle, each having a word-length of *N*, while d_2 bits of m_2 words are output every clock cycle. For word-level converters used in two-dimensional image/video processing applications, the above format may be interpreted as: d_1 samples of m_1 rows of the image are input every input clock cycle, each row having *N* samples, and d_2 samples of m_2 rows are output every clock cycle.

Such an input specification can be used for almost all DFC applications, except for a few like the zigzag scanner.

For convenience of demonstrating the difference of the 2-D register allocation schemes, we use 1-D DWT as an example.

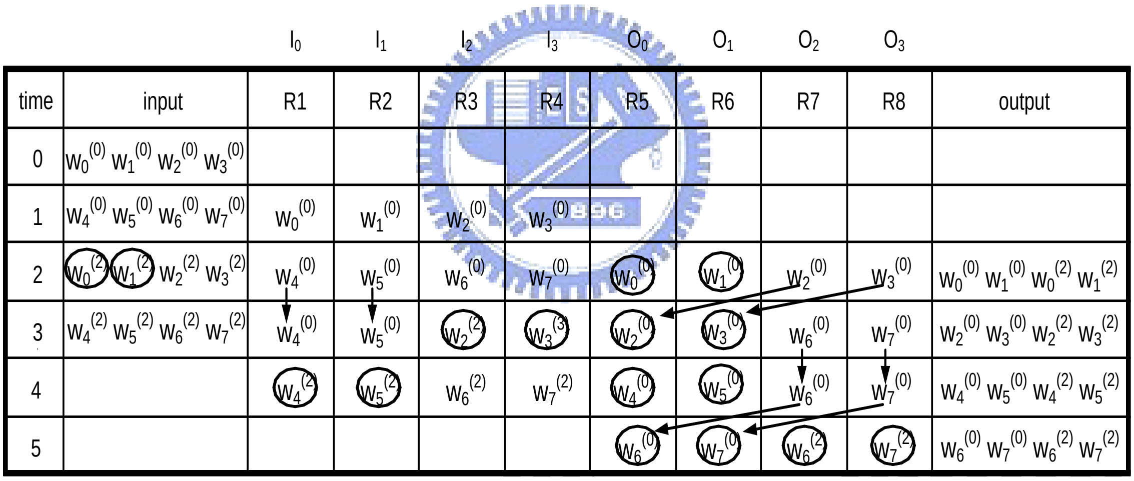
Assume that X and Y denote the input matrix and the output matrix, respectively.

Let $\mathbf{X}=\begin{bmatrix} w_0^{(0)} & w_1^{(0)} & w_2^{(0)} & w_3^{(0)} \\ w_4^{(0)} & w_5^{(0)} & w_6^{(0)} & w_7^{(0)} \\ w_0^{(2)} & w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \\ w_4^{(2)} & w_5^{(2)} & w_6^{(2)} & w_7^{(2)} \end{bmatrix}$, we obtain $\mathbf{Y}=\begin{bmatrix} w_0^{(0)} & w_1^{(0)} & w_0^{(2)} & w_1^{(2)} \\ w_2^{(0)} & w_3^{(0)} & w_2^{(2)} & w_3^{(2)} \\ w_4^{(0)} & w_5^{(0)} & w_4^{(2)} & w_5^{(2)} \\ w_6^{(0)} & w_7^{(0)} & w_6^{(2)} & w_7^{(2)} \end{bmatrix}$, where the input and output data sequences are scanned in $\{w_0^{(0)}, w_1^{(0)}, w_2^{(0)}, w_3^{(0)}\}$, $\{w_4^{(0)}, w_5^{(0)},$

$w_6^{(0)}$, $w_7^{(0)}$}, {$w_0^{(2)}$, $w_1^{(2)}$, $w_2^{(2)}$, $w_3^{(2)}$}, {$w_4^{(2)}$, $w_5^{(2)}$, $w_6^{(2)}$, $w_7^{(2)}$} and {$w_0^{(0)}$, $w_1^{(0)}$, $w_0^{(2)}$, $w_1^{(2)}$}, {$w_2^{(0)}$, $w_3^{(0)}$, $w_2^{(2)}$, $w_3^{(2)}$}, {$w_4^{(0)}$, $w_5^{(0)}$, $w_4^{(2)}$, $w_3^{(2)}$}, {$w_4^{(2)}$, $w_5^{(2)}$, $w_6^{(2)}$, $w_7^{(2)}$}, respectively.

Figure 2.5 shows allocation table for the 1-D discrete wavelet transform (DWT) using the 2-D register allocation. The specification of the 1-D DWT DFC can be given as (1,4)→(2,2)[8].

Figure 2.6 shows architecture for the 1-D DWT using the 2-D register allocation. The 2-D register allocation use multiple local interconnects and multiple global interconnects as well as reduce the number of interconnections by maximizing the reuse of the interconnections.



		I_0	I_1	I_2	I_3	O_0	O_1	O_2	O_3	
time	input	R1	R2	R3	R4	R5	R6	R7	R8	output
0	$W_0^{(0)}$ $W_1^{(0)}$ $W_2^{(0)}$ $W_3^{(0)}$									
1	$W_4^{(0)}$ $W_5^{(0)}$ $W_6^{(0)}$ $W_7^{(0)}$	$W_0^{(0)}$	$W_1^{(0)}$	$W_2^{(0)}$	$W_3^{(0)}$					
2	$W_0^{(2)}$ $W_1^{(2)}$ $W_2^{(2)}$ $W_3^{(2)}$	$W_4^{(0)}$	$W_5^{(0)}$	$W_6^{(0)}$	$W_7^{(0)}$	$W_0^{(0)}$	$W_1^{(0)}$	$W_2^{(0)}$	$W_3^{(0)}$	$W_0^{(0)}$ $W_1^{(0)}$ $W_0^{(2)}$ $W_1^{(2)}$
3	$W_4^{(2)}$ $W_5^{(2)}$ $W_6^{(2)}$ $W_7^{(2)}$	$W_4^{(0)}$	$W_5^{(0)}$	$W_2^{(2}$	$W_3^{(3}$	$W_2^{(0}$	$W_3^{(0}$	$W_6^{(0)}$	$W_7^{(0)}$	$W_2^{(0)}$ $W_3^{(0)}$ $W_2^{(2)}$ $W_3^{(2)}$
4		$W_4^{(2}$	$W_5^{(2}$	$W_6^{(2)}$	$W_7^{(2)}$	$W_4^{(0}$	$W_5^{(0}$	$W_6^{(0)}$	$W_7^{(0)}$	$W_4^{(0)}$ $W_5^{(0)}$ $W_4^{(2)}$ $W_5^{(2)}$
5						$W_6^{(0}$	$W_7^{(0}$	$W_6^{(2}$	$W_7^{(2}$	$W_6^{(0)}$ $W_7^{(0)}$ $W_6^{(2)}$ $W_7^{(2)}$

Figure 2.5 Allocation Table for 1-D DWT using 2-D register allocation.

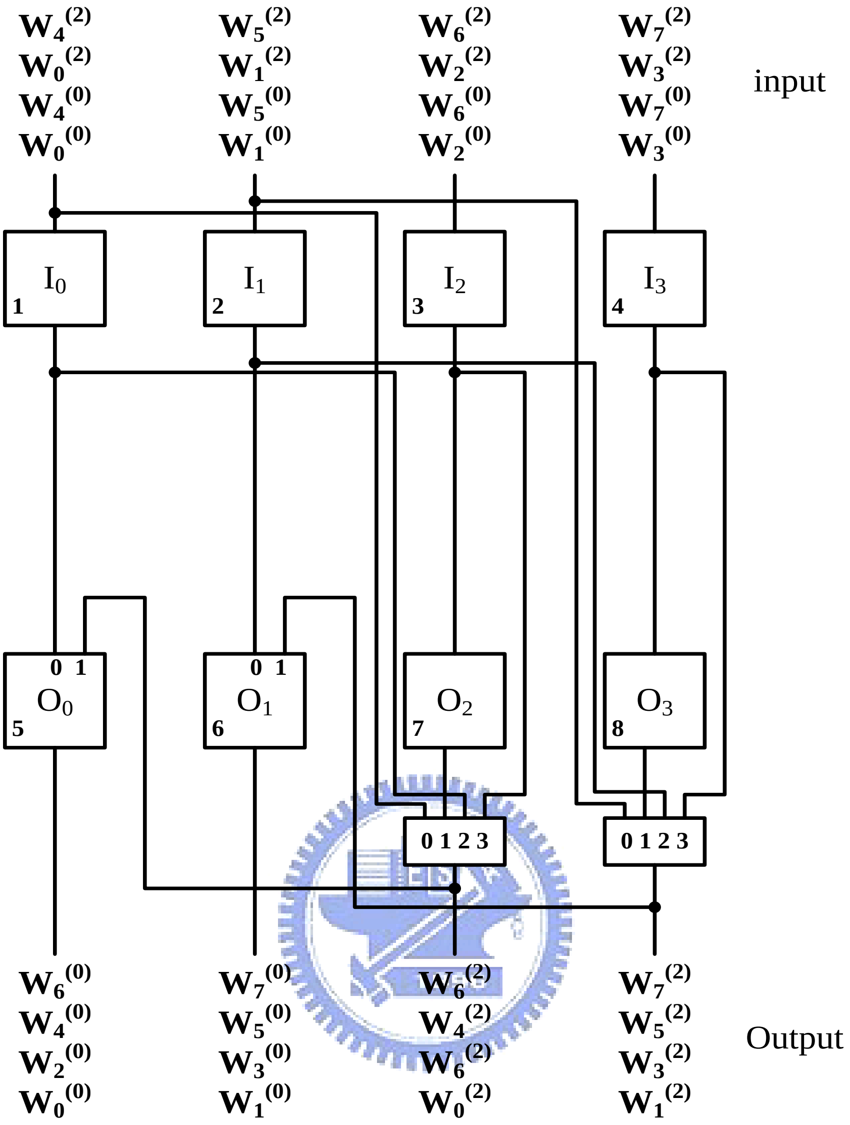

$W_4^{(2)}$ $W_0^{(2)}$ $W_4^{(0)}$ $W_0^{(0)}$
$W_5^{(2)}$ $W_1^{(2)}$ $W_5^{(0)}$ $W_1^{(0)}$
$W_6^{(2)}$ $W_2^{(2)}$ $W_6^{(0)}$ $W_2^{(0)}$
$W_7^{(2)}$ $W_3^{(2)}$ $W_7^{(0)}$ $W_3^{(0)}$
input
I_0 1
I_1 2
I_2 3
I_3 4
0 1 O_0 5
0 1 O_1 6
O_2 7
O_3 8
0 1 2 3
0 1 2 3
$W_6^{(0)}$ $W_4^{(0)}$ $W_2^{(0)}$ $W_0^{(0)}$
$W_7^{(0)}$ $W_5^{(0)}$ $W_3^{(0)}$ $W_1^{(0)}$
$W_6^{(2)}$ $W_4^{(2)}$ $W_6^{(2)}$ $W_0^{(2)}$
$W_7^{(2)}$ $W_5^{(2)}$ $W_3^{(2)}$ $W_1^{(2)}$
Output

Figure 2.6 Architecture for 1-D DWT using 2-D register allocation.

Chapter 3

1-D Static Register Allocation Algorithm and Architecture

In this chapter, we explore the 1-D static register allocation (SRA) algorithm and the corresponding low-power area-efficient DFC architecture. In the same chapter, the properties for *N*x*N* matrix transposer and control unit of the 2-D DFC design are presented.

3.1 Algorithm and Architecture

Given the input time and output time of all variables.

The design steps of the 1-D SRA algorithm are described as follows.

Step 1: Determine the minimum number of registers using the lifetime analysis.

Step 2: Assign the input variable to the available register in ascending order until exceeding the minimum number of registers.

Step 3: Return to and search for the available register from the first register when exceeding the minimum number of registers.

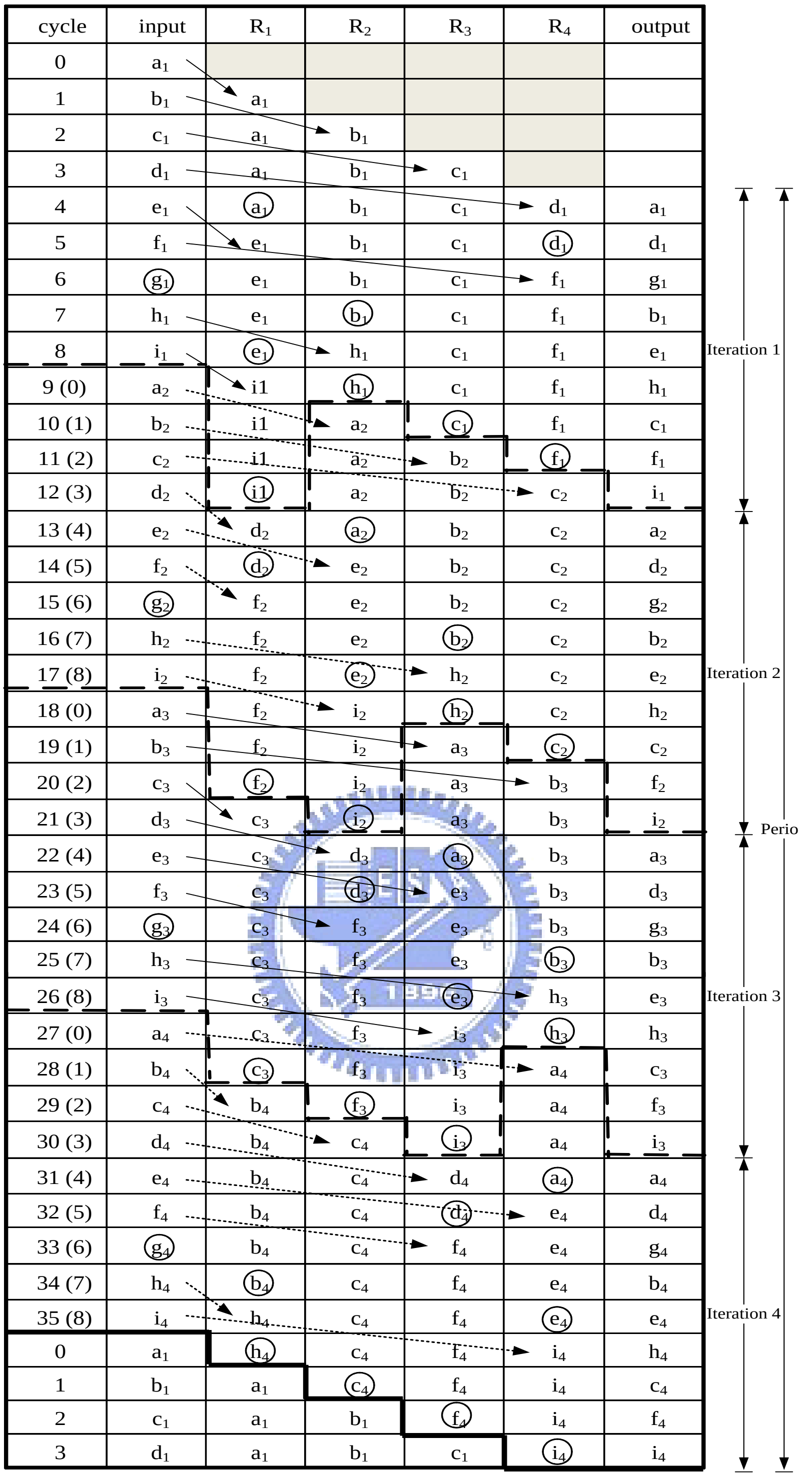
Step 4: Repeat steps 2 and 3 as required until one iteration allocation is complete.

Step 5: Go to the following iterations and repeat steps 2, 3 and 4 as required until one period allocation is complete.

In this thesis, the iteration is defined as the required cycles to finish a computation process. The period is defined as the required cycles to finish a complete computation process, where each complete computation process has the same register allocation assignment for all variables. If the sum of life time of the corresponding variables storing at the same register is larger than the number of cycles for one iteration, the DFC possesses the feature of the multiple iterations. Otherwise, the DFC will belong to the single iteration. For the single iteration, the period has the same cycle count as the iteration. Otherwise, the period is equal to the cycles of the multiple iterations. Without loss of the generality, we use four benchmarks including 3x3 transposer, 4x4 transposer, 16x16 transposer and IIR filter to verify and demonstrate the above design steps. In the first benchmark, assume X_{3x3} and Y_{3x3} denote the input matrix and the transpose matrix of X_{3x3}, respectively, where the relationship is expressed in (3.1).

$$\mathbf{Y}_{3x3} = \mathbf{X}_{3x3}^{T}. \qquad (3.1)$$

Let $\mathbf{X}_{3x3} = \begin{bmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{bmatrix}$, we obtain $\mathbf{Y}_{3x3} = \begin{bmatrix} a_1 & d_1 & g_1 \\ b_1 & e_1 & h_1 \\ c_1 & f_1 & i_1 \end{bmatrix}$, where the input and output data sequences are scanned in $\{a_1,b_1,c_1,d_1,e_1,f_1,g_1,h_1,i_1\}$ and $\{a_1,d_1,g_1,b_1,e_1,h_1,c_1,f_1,i_1\}$, respectively. The length of both sequences is nine. In step 1, using the lifetime analysis in [2], the minimum number of registers is four. In step 2, we use allocation table to assign the input variables to available registers in ascending order as shown in Figure 3.1, where arrow denotes the register transition. Thus, the input sequences $\{a_1, b_1, c_1, d_1\}$ are inputted to the corresponding registers $\{R_1, R_2, R_3, R_4\}$. For next sequence data e_1, we need to return to the first register and search for which register is available from R_1 to R_4 in step 3. In this case, R_1 is available for input sequence data e_1. About the next input sequence data f_1, we find that R_2 and R_3 are not empty as shown in Figure 3.1 and then skip R_2 and R_3. Thus, the input data f_1 is inputted to the R_4 register.



cycle	input	R_1	R_2	R_3	R_4	output
0	a_1					
1	b_1	a_1				
2	c_1	a_1	b_1			
3	d_1	a_1	b_1	c_1		
4	e_1	a_1	b_1	c_1	d_1	a_1
5	f_1	e_1	b_1	c_1	d_1	d_1
6	g_1	e_1	b_1	c_1	f_1	g_1
7	h_1	e_1	b_1	c_1	f_1	b_1
8	i_1	e_1	h_1	c_1	f_1	e_1
9 (0)	a_2	i1	h_1	c_1	f_1	h_1
10 (1)	b_2	i1	a_2	c_1	f_1	c_1
11 (2)	c_2	i1	a_2	b_2	f_1	f_1
12 (3)	d_2	i1	a_2	b_2	c_2	i_1
13 (4)	e_2	d_2	a_2	b_2	c_2	a_2
14 (5)	f_2	d_2	e_2	b_2	c_2	d_2
15 (6)	g_2	f_2	e_2	b_2	c_2	g_2
16 (7)	h_2	f_2	e_2	b_2	c_2	b_2
17 (8)	i_2	f_2	e_2	h_2	c_2	e_2
18 (0)	a_3	f_2	i_2	h_2	c_2	h_2
19 (1)	b_3	f_2	i_2	a_3	c_2	c_2
20 (2)	c_3	f_2	i_2	a_3	b_3	f_2
21 (3)	d_3	c_3	i_2	a_3	b_3	i_2
22 (4)	e_3	c_3	d_3	a_3	b_3	a_3
23 (5)	f_3	c_3	d_3	e_3	b_3	d_3
24 (6)	g_3	c_3	f_3	e_3	b_3	g_3
25 (7)	h_3	c_3	f_3	e_3	b_3	b_3
26 (8)	i_3	c_3	f_3	e_3	h_3	e_3
27 (0)	a_4	c_3	f_3	i_3	h_3	h_3
28 (1)	b_4	c_3	f_3	i_3	a_4	c_3
29 (2)	c_4	b_4	f_3	i_3	a_4	f_3
30 (3)	d_4	b_4	c_4	i_3	a_4	i_3
31 (4)	e_4	b_4	c_4	d_4	a_4	a_4
32 (5)	f_4	b_4	c_4	d_4	e_4	d_4
33 (6)	g_4	b_4	c_4	f_4	e_4	g_4
34 (7)	h_4	b_4	c_4	f_4	e_4	b_4
35 (8)	i_4	h_4	c_4	f_4	e_4	e_4
0	a_1	h_4	c_4	f_4	i_4	h_4
1	b_1	a_1	c_4	f_4	i_4	c_4
2	c_1	a_1	b_1	f_4	i_4	f_4
3	d_1	a_1	b_1	c_1	i_4	i_4

Figure 3.1 Allocation Table for 3x3 matrix transposer using 1-D SRA.

In step 4, we repeat steps 2 and 3 recursively until one iteration allocation is done as shown in Figure 3.1, where the dash-bold line denotes the boundary of one iteration. Equivalently, one computation process, 3x3 transposing, is calculated under one iteration. Finally, in step 5, we repeat steps 2, 3, 4 as required until one period allocation is finished as shown in Figure 3.1, where the solid-bold line denotes the boundary of one period. In this case, the 3x3 transposer has nine cycles and 36 cycles for one iteration and period, respectively. From the allocation table as shown in Figure 3.1, all input variables are static in one register until they become desired outputs. Hence, this method is referred to 1-D static register allocation (SRA). According to the proposed 5-step 1-D SRA algorithm, the number of transitions can be further minimized compared with that of [2-3] and [7-8]. In this case, the numbers of transitions for each iteration handled by FBRA, SSRA, and 1-D SRA schemes are 36, 11, and 8, respectively. The corresponding new 3x3 transposer architecture is depicted in Figure 3.2. In similar behavior, the proposed 1-D SRA is capable of treating the register allocation for higher-order transposer. In the case of 4x4 transposer, the numbers of transitions for each iteration handled by FBRA, SSRA, and 1-D SRA schemes are 144, 24, and 15, respectively. For larger size case, 16x16 transposer has 57600, 570, and 255 transitions for each iteration performed by FBRA, SSRA, and 1-D SRA schemes, respectively. As a consequence, the 1-D SRA scheme can lead to lower transitions for *N*x*N* transposer.

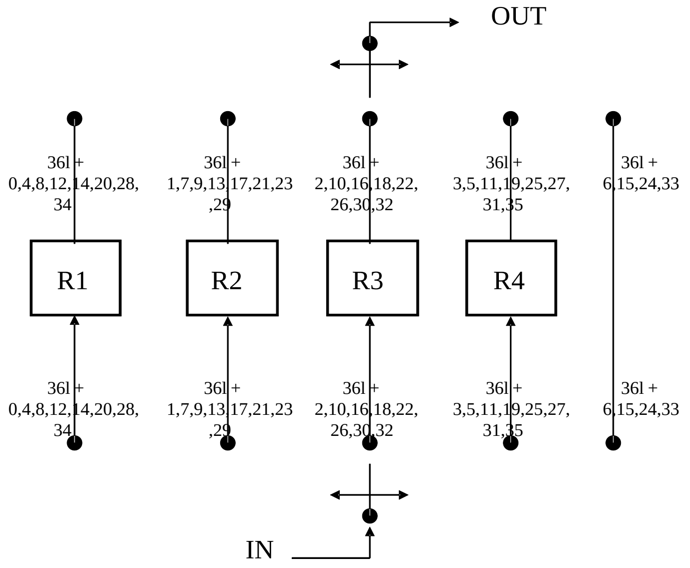
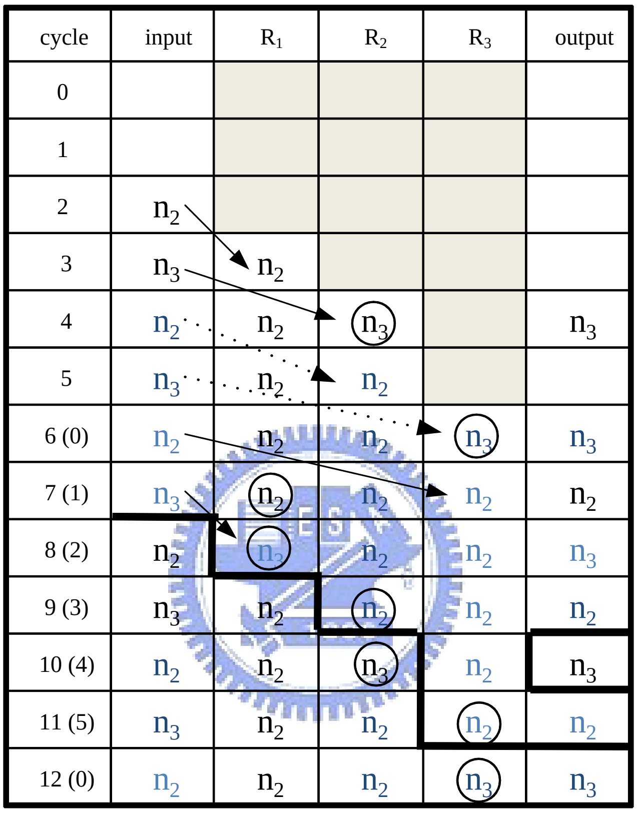

OUT
36l + 0,4,8,12,14,20,28, 34
36l + 1,7,9,13,17,21,23 ,29
36l + 2,10,16,18,22, 26,30,32
36l + 3,5,11,19,25,27, 31,35
36l + 6,15,24,33
R1
R2
R3
R4
36l + 0,4,8,12,14,20,28, 34
36l + 1,7,9,13,17,21,23 ,29
36l + 2,10,16,18,22, 26,30,32
36l + 3,5,11,19,25,27, 31,35
36l + 6,15,24,33
IN

Figure 3.2 Block diagram of 3x3 transposer using 1-D SRA.

In the fourth benchmark, IIR filter that computes $y(n) = ay(n\text{-}3) + by(n\text{-}5) + x(n)$ is folded, we apply the SRA approach to improve the transition activities. The resulting allocation table and the improved IIR filter architecture are depicted in Figure 3.3 and 3.4, respectively, where the same notations are adopted as that in Fig. 6.19 of [2].



cycle	input	R_1	R_2	R_3	output
0					
1					
2	n_2				
3	n_3	n_2			
4	n_2	n_2	n_3		n_3
5	n_3	n_2	n_2		
6 (0)	n_2	n_2	n_2	n_3	n_3
7 (1)	n_3	n_2	n_2	n_2	n_2
8 (2)	n_2	n_2	n_2	n_2	n_3
9 (3)	n_3	n_2	n_2	n_2	n_2
10 (4)	n_2	n_2	n_3	n_2	n_3
11 (5)	n_3	n_2	n_2	n_2	n_2
12 (0)	n_2	n_2	n_2	n_3	n_3

Figure 3.3 Allocation table for IIR filter using 1-D SRA.

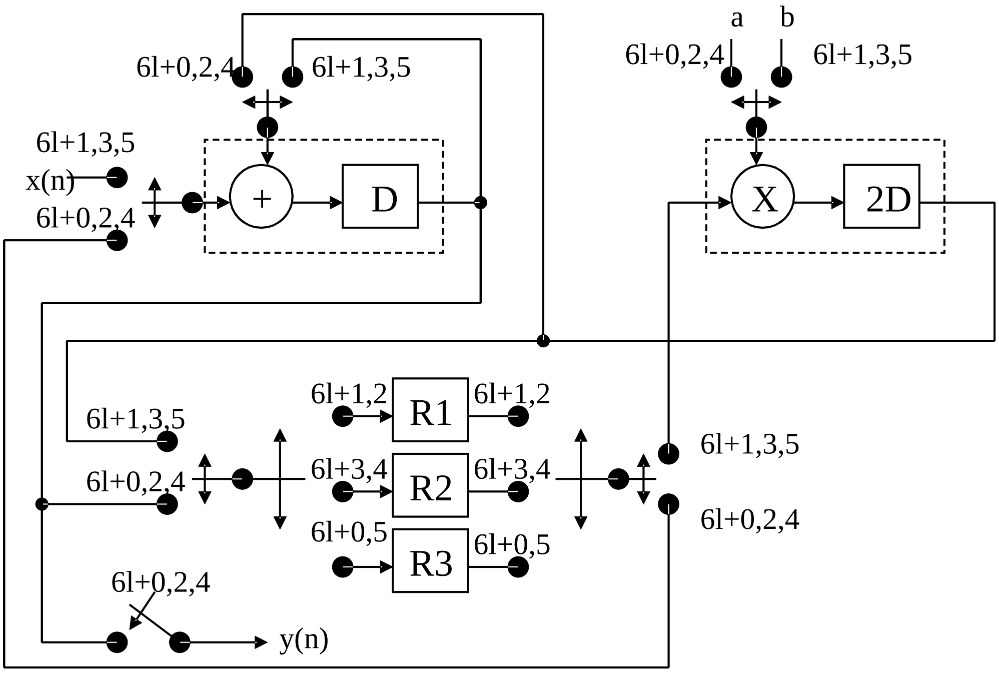


Figure 3.4 Block diagram of IIR filter using 1-D SRA.

3.2 Properties of *N* x *N* Transposer Using 1-D SRA

From the SRA-based allocation table, we observe that the periodic allocation occurs for 3x3, 4x4, and higher-order transposer. For example, if register R_j is occupied with a variable in the first cycle, then R_j owns the identical variable in (l + C_P)-th cycle, where C_P denotes the number of cycles for one period. We can calculate C_P by the following properties 2 and 3.

Without loss of the generality, the relationship of the *N*x*N* input matrix and transposed matrix can be expressed in (3.2).

$$\mathbf{Y}_{N\times N} = \mathbf{X}_{N\times N}^{T} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{bmatrix}^{T} = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{N,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{N,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,N} & a_{2,N} & \cdots & a_{N,N} \end{bmatrix} \qquad (3.2)$$

We can derive the following properties for *N*x*N* transposer.

Property 1: The number of registers N_R equals (N-1)^2 for NxN transposer.

Proof:

Input and output data sequences in (3.2) can be scanned in the following.

$$X_s = \{a_{1,1}, a_{1,2}, ..., a_{1,N}, a_{2,1}, a_{2,2}, ..., a_{2,N}, ..., a_{N,1}, ..., a_{N,N}\} \tag{3.3a}$$

$$Y_s = \{a_{1,1}, a_{2,1}, ..., a_{N,1}, a_{1,2}, a_{2,2}, ..., a_{N,2}, ..., a_{1,N}, ..., a_{N,N}\} \tag{3.3b}$$

From (3.3a) and (3.3b), since NxN transposer is a causal system, we need to shift right the transposed sequence in (3.3b). The variable in (3.3a) compared with that in (3.3b) which has the longest distance will be the aligned point. In this general case, $a_{N,1}$ is the aligned point. Thus, the shift-right distance implemented by registers is equal to the number of registers for NxN transposer. Hence, the required number of registers can be presented in (3.4).

$$N_R = N(N-1)+1-N = (N-1)^2 . \quad \blacklozenge \tag{3.4}$$

Property 2: The number of cycles for one iteration, C_I, equals N^2 for NxN transposer.

Proof:

According to the iteration definition and allocation table as addressed in this chapter, we can see that each iteration consumes the cycles equaling the total number of variables in the transposer. Thus, for NxN transposer, the number of cycles for each iteration can be expressed in (3.5).

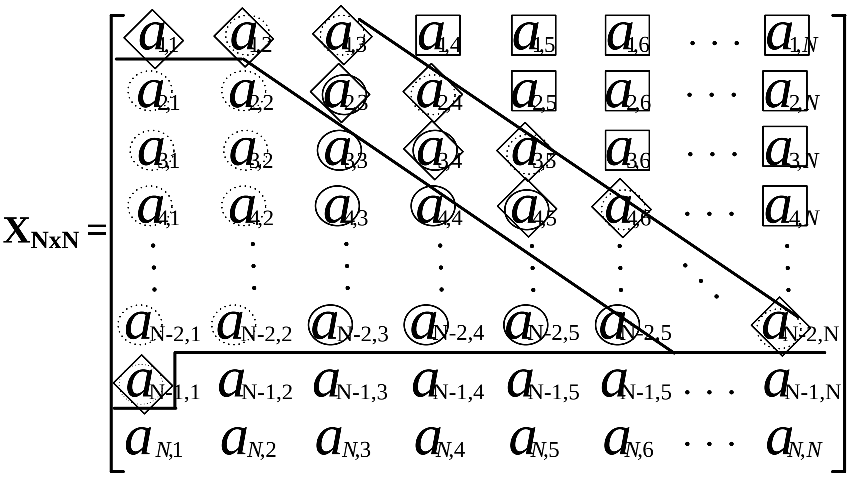
$$C_I = N^2 . \quad \blacklozenge \tag{3.5}$$

Under the multiple-iteration case (i.e., $N \geq 3$), we can obtain Property 3.

Property 3: The number of iterations for one period, I_P, equals 2(N-1) for NxN transposer.

Proof:

For convenience of derivation, the input matrix X is repeated in (3.6).


X_NxN =
a1,1 a1,2 a1,3 a1,4 a1,5 a1,6 . . . a1,N
a2,1 a2,2 a2,3 a2,4 a2,5 a2,6 . . . a2,N
a3,1 a3,2 a3,3 a3,4 a3,5 a3,6 . . . a3,N
a4,1 a4,2 a4,3 a4,4 a4,5 a4,6 . . . a4,N
aN-2,1 aN-2,2 aN-2,3 aN-2,4 aN-2,5 aN-2,5 aN-2,N
aN-1,1 aN-1,2 aN-1,3 aN-1,4 aN-1,5 aN-1,5 . . . aN-1,N
aN,1 aN,2 aN,3 aN,4 aN,5 aN,6 . . . aN,N

(3.6)

At the first iteration, from input sequence in (3.3a) and aligned output sequence of (3.3b), the output variable $a_{1,1}$ occurs at the same time instance as the input variable $a_{N-1,2}$. Similarly, the output variable $a_{N-1,2}$ occurs at the same time instance as the input variable $a_{N,N}$. These three variables are certainly allocated at the same register, R_1. That means the input variable $a_{1,1}$ at the next iteration cannot be allocated at R_1 register and moved to R_2 register. At the second and third iteration, the new first input variable $a_{1,1}$ has to be fed into the next available register R_2 and R_3, respectively, via SRA approach. For N=3, at the forth iteration, the new first input variable $a_{1,1}$ has to be fed into the available register R_4 (i.e., a_4 is fed into R_4 as shown in Figure 3.1). However, for $N \geq 4$, at the fourth iteration, the first input variable will encounter the unavailable register R_4 that is occupied by other variable. That means R_4 register has longer life time while the input variable $a_{1,1}$ is arriving.

For instance, we show the allocation table of 4x4 transposer in Figure 3.3 to illustrate the above situation. In this case, $a_{1,1}$ is needed to input to available register R_7 and each input variable will be only appeared in dedicated registers for $N \geq 4$. As a consequence, the input variables can be separated into Q groups (i.e., G_1, G_2, …, G_Q) to finish the allocation. In the case of 4x4 transposer, there exist two groups in Figure 3.5. The group G_1 is composed of R_1, R_2, R_3, R_7, R_8, R_9 and the group G_2 consists of R_4, R_5,

R_6.

After transposing the NxN matrix, at the first iteration, some registers store multiple variables and some registers store single variable. The sum of the lifetime of all variables stored at the register under the first iteration boundary can be longer than C_I, less than (C_I-1), or equal to C_I and (C_I-1). Thus, the utilized registers from R_1 to $R_{(N\text{-}1)^2}$ can be classified into five types. The lifetime of one register storing the single variable is longer than C_I and then is referred to as type-1 register. The lifetime of one register storing multiple variables is longer than C_I, and then is named as type-2 register. The lifetime of one register storing the single variable is equal to (C_I-1), and then is referred to as type-3 register. The lifetime of one register storing multiple variables is equal to (C_I-1), and then is named as type-4 register. The lifetime of one register storing the single variable is less than (C_I-1), and then is referred to as type-5 register. Thus, for $N \geq 3$, the difference value of the type-3, and 4 register can be defined as the sum of the lifetime of all variables storing in this register in (3.7).

$$D_{R_i} = C_{LT,R_i} - C_I + 1. \tag{3.7}$$

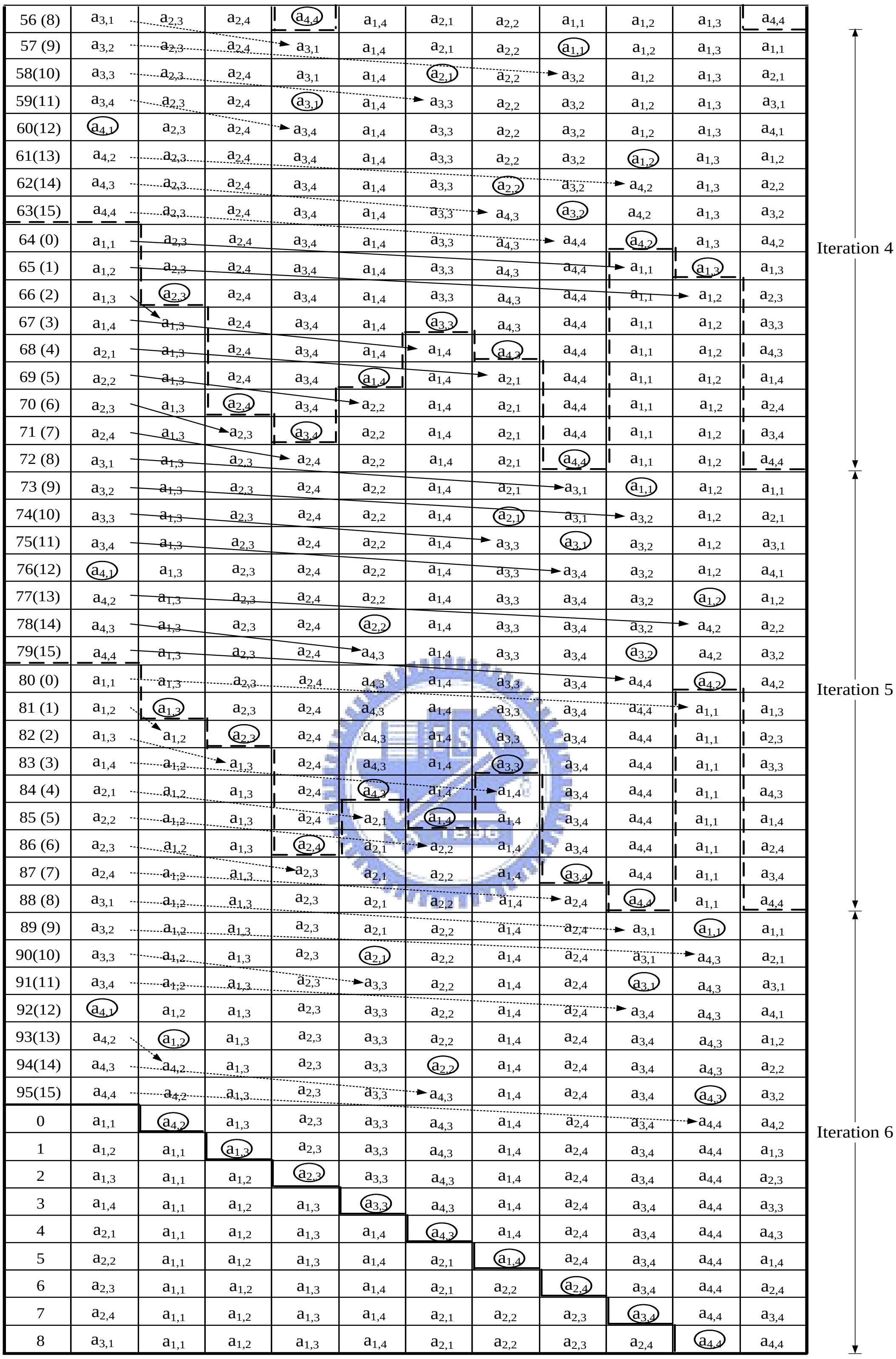
where C_{LT,R_i} denotes the life time of the register R_i. For $a_{1,1}$, it belongs to the type-2 register; from Property 1, the difference cycle $Da_{1,1}$ is $(N\text{-}1)^2$ due to the shift distance. For other input variables marked in dash-line circle, they belong to the type-3 and 4 registers. The difference values of the corresponding registers are equal to zero in (3.7), where the specified input variables $a_{i,j}$ are allocated at $R_{((i\text{-}1)\text{x}N+j)}$. According to the above calculation, the input variable belongs to which register can be determined.

cycle	input	R1	R2	R3	R4	R5	R6	R7	R8	R9	output
0	$a_{1,1}$										
1	$a_{1,2}$	$a_{1,1}$									
2	$a_{1,3}$	$a_{1,1}$	$a_{1,2}$								
3	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$							
4	$a_{2,1}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$						
5	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$					
6	$a_{2,3}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$				
7	$a_{2,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$			
8	$a_{3,1}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$		
9	$a_{3,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{1,1}$
10	$a_{3,3}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{2,1}$
11	$a_{3,4}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{3,1}$
12	$a_{4,1}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{4,1}$
13	$a_{4,2}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,2}$
14	$a_{4,3}$	$a_{3,2}$	$a_{4,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{2,2}$
15	$a_{4,4}$	$a_{3,2}$	$a_{4,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{3,2}$
16 (0)	$a_{1,1}$	$a_{4,4}$	$a_{4,2}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{4,2}$
17 (1)	$a_{1,2}$	$a_{4,4}$	$a_{1,1}$	$a_{1,3}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,3}$
18 (2)	$a_{1,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{2,3}$
19 (3)	$a_{1,4}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{3,3}$
20 (4)	$a_{2,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,4}$	$a_{1,4}$	$a_{4,3}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{4,3}$
21 (5)	$a_{2,2}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,4}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$
22 (6)	$a_{2,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{2,4}$
23 (7)	$a_{2,4}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{3,4}$	$a_{3,4}$
24 (8)	$a_{3,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$
25 (9)	$a_{3,2}$	$a_{3,1}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{1,1}$
26(10)	$a_{3,3}$	$a_{3,1}$	$a_{3,2}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,1}$
27(11)	$a_{3,4}$	$a_{3,1}$	$a_{3,2}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$
28(12)	$a_{4,1}$	$a_{3,4}$	$a_{3,2}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,1}$
29(13)	$a_{4,2}$	$a_{3,4}$	$a_{3,2}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{1,2}$
30(14)	$a_{4,3}$	$a_{3,4}$	$a_{3,2}$	$a_{4,2}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$
31(15)	$a_{4,4}$	$a_{3,4}$	$a_{3,2}$	$a_{4,2}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,2}$
32 (0)	$a_{1,1}$	$a_{3,4}$	$a_{4,4}$	$a_{4,2}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,2}$
33 (1)	$a_{1,2}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{1,3}$
34 (2)	$a_{1,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{1,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,3}$
35 (3)	$a_{1,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{3,3}$
36 (4)	$a_{2,1}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,3}$	$a_{1,4}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{4,3}$
37 (5)	$a_{2,2}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,1}$	$a_{1,4}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{1,4}$
38 (6)	$a_{2,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{2,4}$
39 (7)	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,4}$
40 (8)	$a_{3,1}$	$a_{2,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{4,4}$
41 (9)	$a_{3,2}$	$a_{2,4}$	$a_{3,1}$	$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{1,1}$
42(10)	$a_{3,3}$	$a_{2,4}$	$a_{3,1}$	$a_{3,2}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,1}$
43(11)	$a_{3,4}$	$a_{2,4}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,1}$
44(12)	$a_{4,1}$	$a_{2,4}$	$a_{3,4}$	$a_{3,2}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{4,1}$
45(13)	$a_{4,2}$	$a_{2,4}$	$a_{3,4}$	$a_{3,2}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{1,2}$
46(14)	$a_{4,3}$	$a_{2,4}$	$a_{3,4}$	$a_{3,2}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,2}$
47(15)	$a_{4,4}$	$a_{2,4}$	$a_{3,4}$	$a_{3,2}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,2}$
48 (0)	$a_{1,1}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{4,2}$
49 (1)	$a_{1,2}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{1,1}$	$a_{1,3}$	$a_{2,3}$	$a_{1,3}$
50 (2)	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,3}$	$a_{2,3}$
51 (3)	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{3,3}$
52 (4)	$a_{2,1}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,4}$	$a_{4,3}$	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{4,3}$
53 (5)	$a_{2,2}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,1}$	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
54 (6)	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$
55 (7)	$a_{2,4}$	$a_{2,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{3,4}$
56 (8)	$a_{3,1}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{4,4}$

Iteration 1

Iteration 2

Iteration 3



56 (8)	$a_{3,1}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{4,4}$
57 (9)	$a_{3,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,1}$
58(10)	$a_{3,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{2,1}$
59(11)	$a_{3,4}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{3,1}$
60(12)	$a_{4,1}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{4,1}$
61(13)	$a_{4,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,2}$
62(14)	$a_{4,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{2,2}$	$a_{3,2}$	$a_{4,2}$	$a_{1,3}$	$a_{2,2}$
63(15)	$a_{4,4}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{3,2}$	$a_{4,2}$	$a_{1,3}$	$a_{3,2}$
64 (0)	$a_{1,1}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{4,4}$	$a_{4,2}$	$a_{1,3}$	$a_{4,2}$
65 (1)	$a_{1,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,3}$	$a_{1,3}$
66 (2)	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,3}$
67 (3)	$a_{1,4}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{3,3}$	$a_{4,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{3,3}$
68 (4)	$a_{2,1}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{1,4}$	$a_{4,3}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{4,3}$
69 (5)	$a_{2,2}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$	$a_{1,4}$	$a_{2,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,4}$
70 (6)	$a_{2,3}$	$a_{1,3}$	$a_{2,4}$	$a_{3,4}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,4}$
71 (7)	$a_{2,4}$	$a_{1,3}$	$a_{2,3}$	$a_{3,4}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{3,4}$
72 (8)	$a_{3,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{4,4}$
73 (9)	$a_{3,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{3,1}$	$a_{1,1}$	$a_{1,2}$	$a_{1,1}$
74(10)	$a_{3,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{3,1}$	$a_{3,2}$	$a_{1,2}$	$a_{2,1}$
75(11)	$a_{3,4}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{3,1}$	$a_{3,2}$	$a_{1,2}$	$a_{3,1}$
76(12)	$a_{4,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{3,2}$	$a_{1,2}$	$a_{4,1}$
77(13)	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{3,2}$	$a_{1,2}$	$a_{1,2}$
78(14)	$a_{4,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{2,2}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{3,2}$	$a_{4,2}$	$a_{2,2}$
79(15)	$a_{4,4}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{3,2}$	$a_{4,2}$	$a_{3,2}$
80 (0)	$a_{1,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{4,2}$	$a_{4,2}$
81 (1)	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{1,3}$
82 (2)	$a_{1,3}$	$a_{1,2}$	$a_{2,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,3}$
83 (3)	$a_{1,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{3,3}$
84 (4)	$a_{2,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{4,3}$	$a_{1,4}$	$a_{1,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,3}$
85 (5)	$a_{2,2}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{2,1}$	$a_{1,4}$	$a_{1,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{1,4}$
86 (6)	$a_{2,3}$	$a_{1,2}$	$a_{1,3}$	$a_{2,4}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{2,4}$
87 (7)	$a_{2,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,1}$	$a_{3,4}$
88 (8)	$a_{3,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{4,4}$	$a_{1,1}$	$a_{4,4}$
89 (9)	$a_{3,2}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,1}$	$a_{1,1}$	$a_{1,1}$
90(10)	$a_{3,3}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{2,1}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,1}$	$a_{4,3}$	$a_{2,1}$
91(11)	$a_{3,4}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,1}$	$a_{4,3}$	$a_{3,1}$
92(12)	$a_{4,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,3}$	$a_{4,1}$
93(13)	$a_{4,2}$	$a_{1,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,3}$	$a_{1,2}$
94(14)	$a_{4,3}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{2,2}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,3}$	$a_{2,2}$
95(15)	$a_{4,4}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,3}$	$a_{3,2}$
0	$a_{1,1}$	$a_{4,2}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{4,2}$
1	$a_{1,2}$	$a_{1,1}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,3}$
2	$a_{1,3}$	$a_{1,1}$	$a_{1,2}$	$a_{2,3}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{2,3}$
3	$a_{1,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{3,3}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{3,3}$
4	$a_{2,1}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{4,3}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{4,3}$
5	$a_{2,2}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{1,4}$
6	$a_{2,3}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,4}$	$a_{3,4}$	$a_{4,4}$	$a_{2,4}$
7	$a_{2,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{3,4}$	$a_{4,4}$	$a_{3,4}$
8	$a_{3,1}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{4,4}$

Figure 3.5 Allocation table for 4x4 transposer using 1-D SRA.

The difference of the type-1 and type-5 registers can be calculated from the corresponding variables owing to the single variable. The difference of type-1 register designated by square box as shown in (3.6) can be generally represented in (3.8).

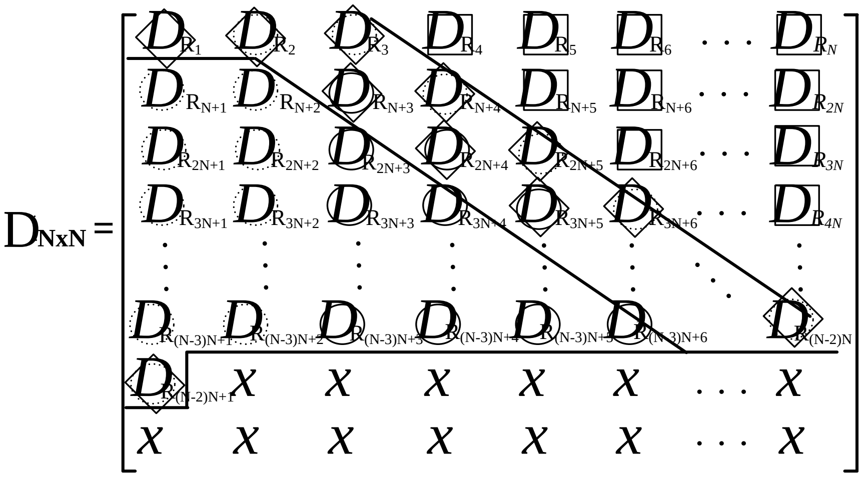
$$\begin{aligned} &D_{a_{i,i+k+3}} = C_{LT,a_{i,i+k+3}} - C_I = (k+1)(N-1) \\ &for\, 1 \le i \le (N-k-3),\, k = 0,1,2,\ldots,(N-4), \text{ and } N \ge 4 \end{aligned} \tag{3.8}$$

where $C_{LT,a_{i,j}}$ denotes the life time of the input variable a$_{i,j}$. Each detailed equation can be obtained from (3.8) as follows. Da$_{1,4}$ = Da$_{2,5}$ = ... = Da$_{N-3,N}$ = (N-1), Da$_{1,5}$ = Da$_{2,6}$ = ... = Da$_{N-4,N}$ = 2(N-1), Da$_{1,6}$ = Da$_{2,7}$ = ... = Da$_{N-5,N}$ = 3(N-1), ..., Da$_{1,N}$ = (N-3)(N-1), where a$_{i,j}$ will be allocated at R$_{((i-1)\text{x}N+j)}$. In similar behavior, at the first iteration, the difference of each type-5 register designated by circle box as shown in (3.6) can be generally represented in (3.9).

$$\begin{aligned} &D_{a_{i+k+1,i+2}} = C_{LT,a_{i+k+1,i+2}} - C_I = -(k+1)(N-1) \\ &for\ 1 \le i \le (N-k-3),\, k = 0,1,2,\ldots,(N-4),\, N \ge 4 \end{aligned} \tag{3.9}$$

Each detailed equation can be obtained from (3.9) as follows. Da$_{2,3}$ = Da$_{3,4}$ =...= Da$_{N-2,N-1}$ = -(N-1), Da$_{3,3}$ = Da$_{4,4}$ = ... = Da$_{N-2,N-2}$ = -2(N-1), ..., Da$_{N-2,3}$ = -(N-3)(N-1), where a$_{i,j}$ will be allocated at R$_{((i-1)\text{x}N+j)}$. Note that, for 3x3 transposer, equations (3.8) and (3.9) do not exist.

According to (3.7), (3.8) and (3.9), we can calculate matrix **D** in (3.10) to represent the relationship of difference of registers.


D$_{N\text{x}N}$ =
D_{R_1} D_{R_2} D_{R_3} D_{R_4} D_{R_5} D_{R_6} . . . D_{R_N}
$D_{R_{N+1}}$ $D_{R_{N+2}}$ $D_{R_{N+3}}$ $D_{R_{N+4}}$ $D_{R_{N+5}}$ $D_{R_{N+6}}$ . . . $D_{R_{2N}}$
$D_{R_{2N+1}}$ $D_{R_{2N+2}}$ $D_{R_{2N+3}}$ $D_{R_{2N+4}}$ $D_{R_{2N+5}}$ $D_{R_{2N+6}}$ . . . $D_{R_{3N}}$
$D_{R_{3N+1}}$ $D_{R_{3N+2}}$ $D_{R_{3N+3}}$ $D_{R_{3N+4}}$ $D_{R_{3N+5}}$ $D_{R_{3N+6}}$ . . . $D_{R_{4N}}$
$D_{R_{(N-3)N+1}}$ $D_{R_{(N-3)N+2}}$ $D_{R_{(N-3)N+3}}$ $D_{R_{(N-3)N+4}}$ $D_{R_{(N-3)N+5}}$ $D_{R_{(N-3)N+6}}$ $D_{R_{(N-2)N}}$
$D_{R_{(N-2)N+1}}$ X X X X X . . . X
X X X X X X X . . . X

(3.10)

In (3.6), G_1 group as designated by diamond box consists of the input variables $a_{1,1}$, $a_{1,2}$, $a_{1,3}$, $a_{2,3}$, $a_{2,4}$,..., $a_{N\text{-}2,N\text{-}1}$, $a_{N\text{-}2,N}$, $a_{N\text{-}1,1}$. Using (3.6), (3.7), (3.8), (3.9) and (3.10), the cycles of D_{G1} can be derived as

$$D_{G1} = D_{R1} + D_{R2} + D_{R3} + \ldots + D_{R(N^2-2N-1)} + Da_{N-2,N-1} + D_{R(N-1)^2}$$

$$= D_{R1} + D_{R2} + D_{R3} + \ldots + D_{R(N^2-2N-1)} + D_{R(N(N-2))} + D_{R(N-1)^2}$$

$$= (N-1)^2 - (N-3)(N-1) = 2(N-1)\,, \qquad N \geq 3 \tag{3.11}$$

where D_{G1} denotes the sum of difference cycles of registers in G_1 group. However, it is difficult to formulate the relationship between difference value of D_G of other group and order *N*. From (3.10), the value of D_G will be consumed by D_G iterations. That means one complete periodic allocation process can be finished after D_{G1} iterations. Thus, we obtain (3.11) as follows.

$$I_P = 2(N-1)\,, \quad N \geq 3\,. \qquad \blacklozenge \tag{3.12}$$

Example 1: For 3x3 transposer whose input matrix is shown in (3.13), at the first iteration, the life time cycle of each register is listed from (3.14a) to (3.14d) and the 3x3 matrix D is shown in (3.15).

$$\mathbf{X_{3x3}} = \begin{bmatrix} \boxed{a_{1,1}} & \boxed{a_{1,2}} & \boxed{a_{1,3}} \\ \boxed{a_{2,1}} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \tag{3.13}$$

and

$$D_{R1} = 13 - 9 = 4, \tag{3.14a}$$

$$D_{R2} = 9 - 9 = 0, \tag{3.14b}$$

$$D_{R3} = 9 - 9 = 0, \tag{3.14c}$$

$$D_{R4} = 9 - 9 = 0. \tag{3.14d}$$

$$\mathbf{D_{3x3}} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & x & x \\ x & x & x \end{bmatrix} \qquad (3.15)$$

In this case, the group G$_1$ consists of R$_1$, R$_2$, R$_3$, and R$_4$. We obtain the number of iterations by summing (3.14a) to (3.14d). I_p=D$_{G1}$=D$_{R1}$+D$_{R2}$+D$_{R3}$+D$_{R4}$=4. Thus, we require four iterations to complete one period allocation for 3x3 transposer. ◆

Example 2: For 4x4 transposer whose input matrix is shown in (3.16), at the first iteration, the life time cycle of each register is listed from (3.17a) to (3.17i) and the 4x4 matrix R is shown in (3.18).

$$\mathbf{X_{4x4}} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \boxed{a_{1,4}} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \qquad (3.16)$$

and

$$D_{R1} = 25 - 16 = 9, \qquad (3.17a)$$

$$D_{R2} = 16 - 16 = 0, \qquad (3.17b)$$

$$D_{R3} = 16 - 16 = 0, \qquad (3.17c)$$

$$D_{R4} = 19 - 16 = 3, \qquad (3.17d)$$

$$D_{R5} = 16 - 16 = 0, \qquad (3.17e)$$

$$D_{R6} = 16 - 16 = 0, \qquad (3.17f)$$

$$D_{R7} = 13 - 16 = -3, \qquad (3.17g)$$

$$D_{R8} = 16 - 16 = 0, \qquad (3.17h)$$

$$D_{R9} = 16 - 16 = 0, \qquad (3.17i)$$

$$\mathbf{D}_{4x4} = \begin{bmatrix} 9 & 0 & 0 & \boxed{3} \\ 0 & 0 & -3 & 0 \\ 0 & x & x & x \\ x & x & x & x \end{bmatrix} \qquad (3.18)$$

Note that there exist two groups. The group G_1 is composed of R_1, R_2, R_3, R_7, R_8, R_9 and the group G_2 consists of R_4, R_5, R_6. We obtain the number of iterations by summing (3.17a), (3.17b), (3.17c), (3.17g), (3.17h), and (3.17i). $I_p = D_{G1} = D_{R1} + D_{R2} + D_{R3} + D_{R7} + D_{R8} + D_{R9} = 9\text{-}3x1 = 6$. Thus, we require six iterations to complete one period allocation for 4x4 transposer. ◆

Hence, we are able to calculate the number of cycles for one period via multiplying (3.5) and (3.12) for multiple iterations as expressed in upper part of (3.19). For single iteration, C_P=C_I as expressed in lower part of (3.19).

$$C_P = \begin{cases} 2N^2(N-1) & for\ \ N \geq 3 \\ C_I = N^2 & for\ \ N = 2 \end{cases}. \qquad (3.19)$$

In the case of the 3x3 transposer, we can directly use (3.19) to calculate the number of cycles for one period (i.e., $C_P = 36$).

3.3 Control Unit of DFC Using 1-D SRA

The control unit of DFC designs affects the control area size and power consumption of the DFC design. Since the control unit of the 1-D SRA-based design has to handle multiple iterations (one period), the control overhead is larger than that of the conventional designs. The signals generated by control unit are responsible for controlling the multiplexor and register writing. Since the control signal can be partitioned to several groups, the number of groups depends on the size of the DFC design (i.e., the 4x4 transposer with two groups). In the multiple-iteration case, the group control signal bits of the second iteration can be obtained by rotating right 1-bit of the group control signal bits of the first iteration. The group control signal bits of the third iteration can be obtained by rotating right 1-bit of the group control signal bits of the second iteration and so forth until the last iteration. An example is shown in Figure 3.6. According to the above analysis, we can use barrel shifters to reduce overhead and complexity of the control unit. The architecture of control unit using SRA is shown in Figure 3.7.

The architecture of 4x4 transposer using SRA is shown in Figure 3.8. Compared with SSRA, the input ports of all registers connect to data input port of the top module to reduce control overhead and each writing signal of register is independently generated by iteration based control unit.

The control signal of Iteration 1 :

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
Cycle 0	1	0	0	0	0	0	0	0	0
Cycle 1	0	1	0	0	0	0	0	0	0
Cycle 2	0	0	1	0	0	0	0	0	0
Cycle 3	0	0	0	1	0	0	0	0	0
Cycle 4	0	0	0	0	1	0	0	0	0
Cycle 5	0	0	0	0	0	1	0	0	0
Cycle 6	0	0	0	0	0	0	1	0	0
Cycle 7	0	0	0	0	0	0	0	1	0
Cycle 8	0	0	0	0	0	0	0	0	1
Cycle 9	1	0	0	0	0	0	0	0	0
Cycle 10	0	0	0	0	1	0	0	0	0
Cycle 11	0	0	0	0	0	0	0	0	1
Cycle 12	0	0	0	0	0	0	0	0	0
Cycle 13	0	1	0	0	0	0	0	0	0
Cycle 14	0	0	0	0	0	1	0	0	0
Cycle 15	1	0	0	0	0	0	0	0	0

The control signal of Iteration 2 :

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
Cycle 0	0	1	0	0	0	0	0	0	0
Cycle 1	0	0	1	0	0	0	0	0	0
Cycle 2	0	0	0	0	0	0	1	0	0
Cycle 3	0	0	0	0	1	0	0	0	0
Cycle 4	0	0	0	0	0	1	0	0	0
Cycle 5	0	0	0	1	0	0	0	0	0
Cycle 6	0	0	0	0	0	0	0	1	0
Cycle 7	0	0	0	0	0	0	0	0	1
Cycle 8	1	0	0	0	0	0	0	0	0
Cycle 9	0	1	0	0	0	0	0	0	0
Cycle 10	0	0	0	0	0	1	0	0	0
Cycle 11	1	0	0	0	0	0	0	0	0
Cycle 12	0	0	0	0	0	0	0	0	0
Cycle 13	0	0	1	0	0	0	0	0	0
Cycle 14	0	0	0	1	0	0	0	0	0
Cycle 15	0	1	0	0	0	0	0	0	0

Figure 3.6 Control signals of the 4x4 transposer using 1-D SRA.

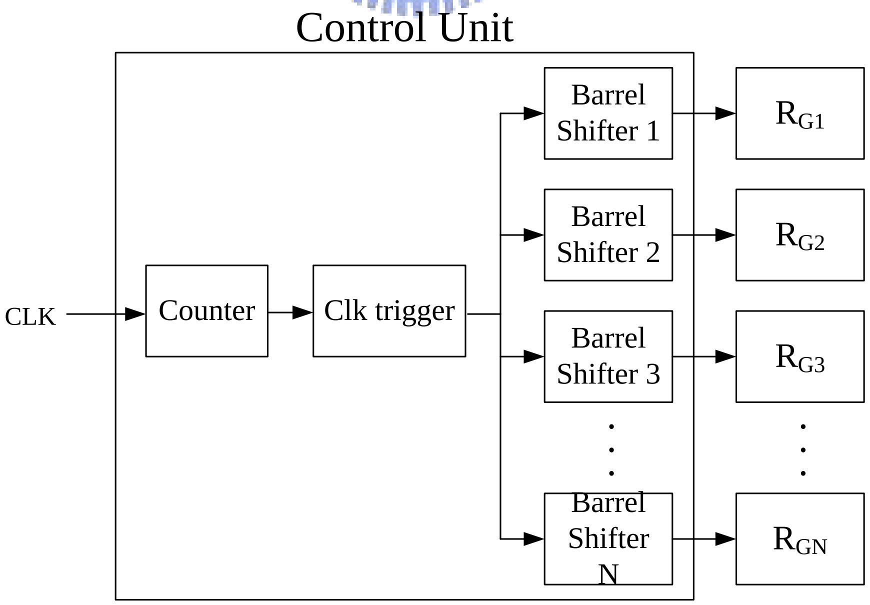


Figure 3.7 Architecture of control unit of DFC using 1-D SRA.

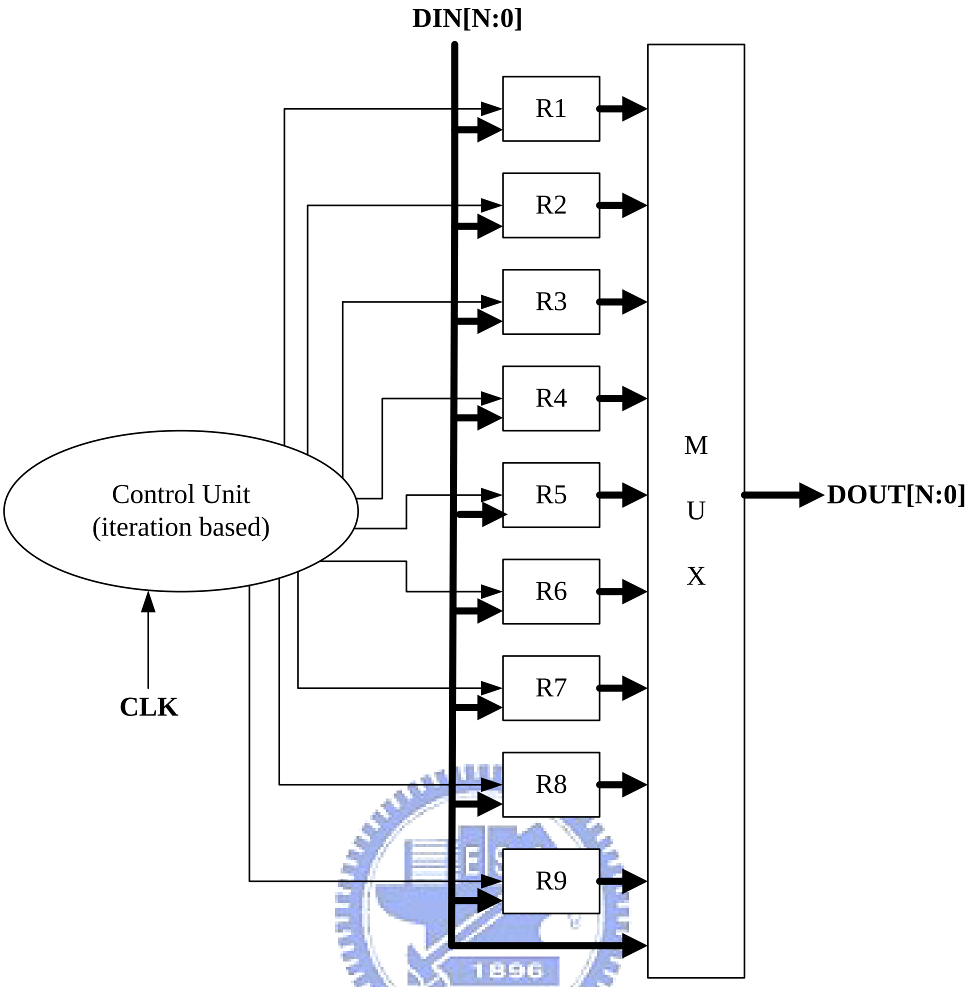


Figure 3.8 Architecture of 4x4 transposer using 1-D SRA.

Chapter 4

2-D Static Register Allocation Algorithm and Architecture

In this chapter, we propose the 2-D static register allocation (2-D SRA) algorithm and the corresponding low-power area-efficient 2-D DFC architecture. In the same chapter, the properties for *N*x*N* par-transposer and control unit of the 2-D DFC design are presented.

4.1 Algorithm and Architecture

According to [4], the throughput of the 2-D DFC is maintained constant, i.e., the input and output data rates are the same. From the given specifications, the number of data samples input in every input clock cycle $= m_1 \times d_1$, while the number of samples output in every output clock cycle $= m_2 \times d_2$. Thus, if g = gcd($m_1 \times d_1, m_2 \times d_2$), then the input cycle period = $(m_1 \times d_1)/g$ and the output cycle period = $(m_2 \times d_2)/g$. This ensures that the number of samples input and output from the DFC for every clock cycle remains constant at g samples per time unit.

The design steps of the 2-D SRA algorithm are described as follows.

Step 1: Determine the minimum number of registers using the lifetime analysis.

Step 2: Assign the input variables to the available registers in ascending order until exceeding the minimum number of registers.

Step 3: Return to and search for the available registers from the first register when exceeding the minimum number of registers.

◆Allocate the corresponding variable to the same register when the variable of the last iteration stores at the register.

◆Allocate the variable with the shorter lifetime to the register with the lower index and allocate other variables with the longer lifetime to the next register with larger index in ascending order.

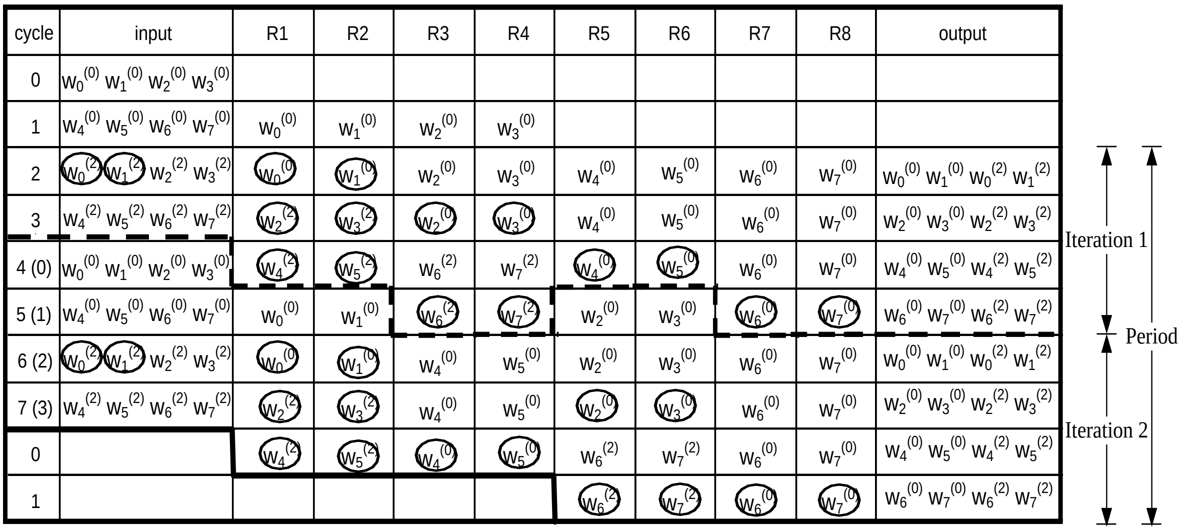
Step 4: Repeat steps 2 and 3 as required until one iteration allocation is complete.

Step 5: Go to the following iterations and repeat steps 2, 3 and 4 as required until one period allocation is complete.

In this chapter, the iteration is defined as the required cycles to finish a computation process. The period is defined as the required cycles to finish a complete computation process, where each complete computation process has the same register allocation assignment for all variables. If the sum of life time of the corresponding variables storing at the same register is larger than the number of cycles for one iteration, the DFC possesses the feature of the multiple iterations. Otherwise, the DFC will belong to the single iteration. For single iteration, the period has the same cycle count as the iteration. Otherwise, the period is equal to the cycles of the multiple iterations. Without loss of the generality, we use four benchmarks including 1-D discrete wavelet transform (DWT), zigzag scanner, 4x4 par-transposer and 16x16 par-transposer to verify and demonstrate the above design steps. The DFC of 1-D DWT is being increasing used as a tool for multiscale analysis for image compress application [12]. The DFC of 1-D DWT is used to reorganize the data from the filter at the lower resolution level to be fed into the low-pass and high-pass filters of the next resolution level. The specification of the DFC of 1-D DWT can be given as (1,4) →(2,2)[8]. Note that the period of computation is four cycles, and four samples are processed each clock cycle.

In the first benchmark, assume X and Y denote the input matrix and the output matrix, respectively.

Let $\mathbf{X} = \begin{bmatrix} w_0^{(0)} & w_1^{(0)} & w_2^{(0)} & w_3^{(0)} \\ w_4^{(0)} & w_5^{(0)} & w_6^{(0)} & w_7^{(0)} \\ w_0^{(2)} & w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \\ w_4^{(2)} & w_5^{(2)} & w_6^{(2)} & w_7^{(2)} \end{bmatrix}$, we obtain $\mathbf{Y} = \begin{bmatrix} w_0^{(0)} & w_1^{(0)} & w_0^{(2)} & w_1^{(2)} \\ w_2^{(0)} & w_3^{(0)} & w_2^{(2)} & w_3^{(2)} \\ w_4^{(0)} & w_5^{(0)} & w_4^{(2)} & w_5^{(2)} \\ w_6^{(0)} & w_7^{(0)} & w_6^{(2)} & w_7^{(2)} \end{bmatrix}$, where the input and output data sequences are scanned in $\{w_0^{(0)}, w_1^{(0)}, w_2^{(0)}, w_3^{(0)}\}$, $\{w_4^{(0)}, w_5^{(0)}, w_6^{(0)}, w_7^{(0)}\}$, $\{w_0^{(2)}, w_1^{(2)}, w_2^{(2)}, w_3^{(2)}\}$, $\{w_4^{(2)}, w_5^{(2)}, w_6^{(2)}, w_7^{(2)}\}$ and $\{w_0^{(0)}, w_1^{(0)}, w_0^{(2)}, w_1^{(2)}\}$, $\{w_2^{(0)}, w_3^{(0)}, w_2^{(2)}, w_3^{(2)}\}$, $\{w_4^{(0)}, w_5^{(0)}, w_4^{(2)}, w_3^{(2)}\}$, $\{w_4^{(2)}, w_5^{(2)}, w_6^{(2)}, w_7^{(2)}\}$, respectively. The length of both sequences is four. In step 1, using the lifetime analysis in [2], the minimum number of registers is eight. In step 2, we use allocation table to assign the input variables to available registers in ascending order as shown in Figure 4.1. Thus, the input sequences $\{w_0^{(0)}, w_1^{(0)}, w_2^{(0)}, w_3^{(0)}\}$ and $\{w_4^{(0)}, w_5^{(0)}, w_6^{(0)}, w_7^{(0)}\}$ are inputted to the corresponding registers $\{R_1, R_2, R_3, R_4\}$ and $\{R_5, R_6, R_7, R_8\}$. For next sequence data $w_2^{(2)}$ and $w_3^{(2)}$, we need to return to the first register and search for which register is available from R_1 to R_8 in step 3. In this case, R_1 and R_2 are available for input sequence data $w_2^{(2)}$ and $w_3^{(2)}$. About the next input sequences data $\{w_4^{(2)}, w_5^{(2)}, w_6^{(2)}, w_7^{(2)}\}$, we find that R_1, R_2, R_3 and R_4 are not empty as shown in Figure 4.1. From step 3, since the lifetime of $w_4^{(2)}$ and $w_5^{(2)}$ is shorter than the lifetime of $w_6^{(2)}$ and $w_7^{(2)}$, the input data $w_4^{(2)}$ and $w_5^{(2)}$ are inputted to the R_1 and R_2 register and the input data $w_6^{(2)}$ and $w_7^{(2)}$ are inputted to the R_3 and R_4 register.


Iteration 1
Period
Iteration 2

cycle	input	R1	R2	R3	R4	R5	R6	R7	R8	output
0	$w_0^{(0)}$ $w_1^{(0)}$ $w_2^{(0)}$ $w_3^{(0)}$									
1	$w_4^{(0)}$ $w_5^{(0)}$ $w_6^{(0)}$ $w_7^{(0)}$	$w_0^{(0)}$	$w_1^{(0)}$	$w_2^{(0)}$	$w_3^{(0)}$					
2	$w_0^{(2)}$ $w_1^{(2)}$ $w_2^{(2)}$ $w_3^{(2)}$	$w_0^{(0)}$	$w_1^{(0)}$	$w_2^{(0)}$	$w_3^{(0)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_0^{(0)}$ $w_1^{(0)}$ $w_0^{(2)}$ $w_1^{(2)}$
3	$w_4^{(2)}$ $w_5^{(2)}$ $w_6^{(2)}$ $w_7^{(2)}$	$w_2^{(2)}$	$w_3^{(2)}$	$w_2^{(0)}$	$w_3^{(0)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_2^{(0)}$ $w_3^{(0)}$ $w_2^{(2)}$ $w_3^{(2)}$
4 (0)	$w_0^{(0)}$ $w_1^{(0)}$ $w_2^{(0)}$ $w_3^{(0)}$	$w_4^{(2)}$	$w_5^{(2)}$	$w_6^{(2)}$	$w_7^{(2)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_4^{(0)}$ $w_5^{(0)}$ $w_4^{(2)}$ $w_5^{(2)}$
5 (1)	$w_4^{(0)}$ $w_5^{(0)}$ $w_6^{(0)}$ $w_7^{(0)}$	$w_0^{(0)}$	$w_1^{(0)}$	$w_6^{(2)}$	$w_7^{(2)}$	$w_2^{(0)}$	$w_3^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_6^{(0)}$ $w_7^{(0)}$ $w_6^{(2)}$ $w_7^{(2)}$
6 (2)	$w_0^{(2)}$ $w_1^{(2)}$ $w_2^{(2)}$ $w_3^{(2)}$	$w_0^{(0)}$	$w_1^{(0)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_2^{(0)}$	$w_3^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_0^{(0)}$ $w_1^{(0)}$ $w_0^{(2)}$ $w_1^{(2)}$
7 (3)	$w_4^{(2)}$ $w_5^{(2)}$ $w_6^{(2)}$ $w_7^{(2)}$	$w_2^{(2)}$	$w_3^{(2)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_2^{(0)}$	$w_3^{(0)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_2^{(0)}$ $w_3^{(0)}$ $w_2^{(2)}$ $w_3^{(2)}$
0		$w_4^{(2)}$	$w_5^{(2)}$	$w_4^{(0)}$	$w_5^{(0)}$	$w_6^{(2)}$	$w_7^{(2)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_4^{(0)}$ $w_5^{(0)}$ $w_4^{(2)}$ $w_5^{(2)}$
1						$w_6^{(2)}$	$w_7^{(2)}$	$w_6^{(0)}$	$w_7^{(0)}$	$w_6^{(0)}$ $w_7^{(0)}$ $w_6^{(2)}$ $w_7^{(2)}$

Figure 4.1 Allocation Table for 1-D DWT using 2-D SRA.

In step 4, we repeat steps 2 and 3 recursively until one iteration allocation is done as shown in Figure 4.1, where the dash-bold line denotes the boundary of one iteration. Equivalently, one computation process is calculated under one iteration. Finally, in step 5, we repeat steps 2, 3, 4 as required until one period allocation is finished as shown in Figure 4.1, where the solid-bold line denotes the boundary of one period. In this case, the 1-D DWT has four cycles and eight cycles for one iteration and period, respectively. From the allocation table as shown in Figure 4.1, all input variables are static in one register until they become desired outputs. Hence, this method is referred to the 2-D static register allocation (2-D SRA) approach. According to the proposed 5-step 2-D SRA algorithm, the number of transitions can be further minimized compared with that of [4] and [7-8]. In this case, the numbers of transitions for each iteration handled by 2-D register allocation, SSRA, 2-D SRA schemes are 28, 16, and 14, respectively. The corresponding new 1-D DWT architecture is depicted in Figure 4.2. Similarly, the proposed 2-D SRA is capable of treating the register allocation for different 2-D DFC applications. The 2-D SRA scheme can account for lower transitions for 2-D DFC designs.

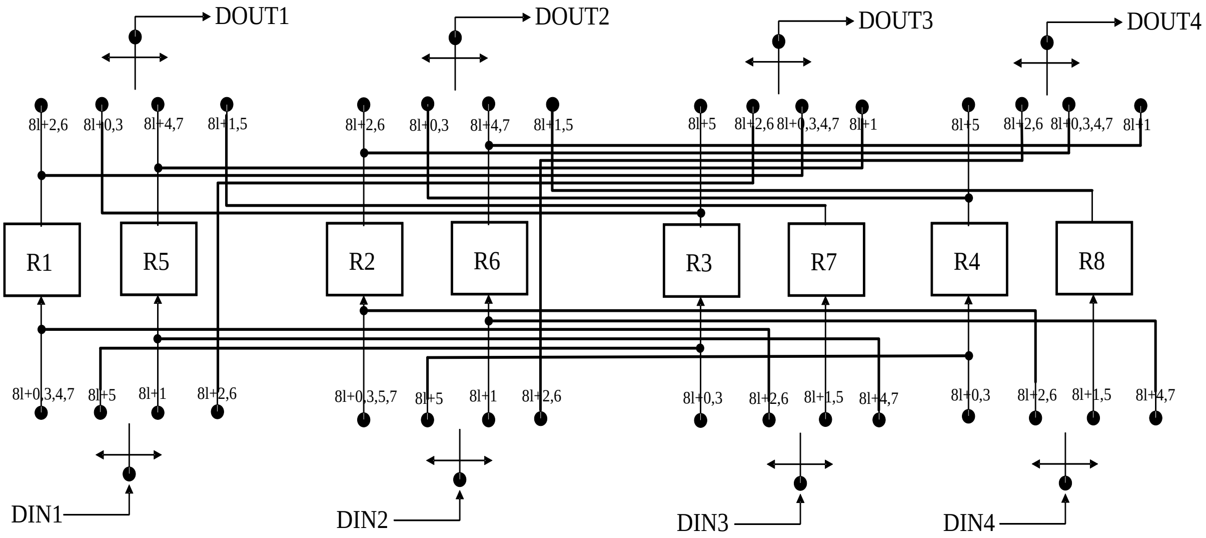
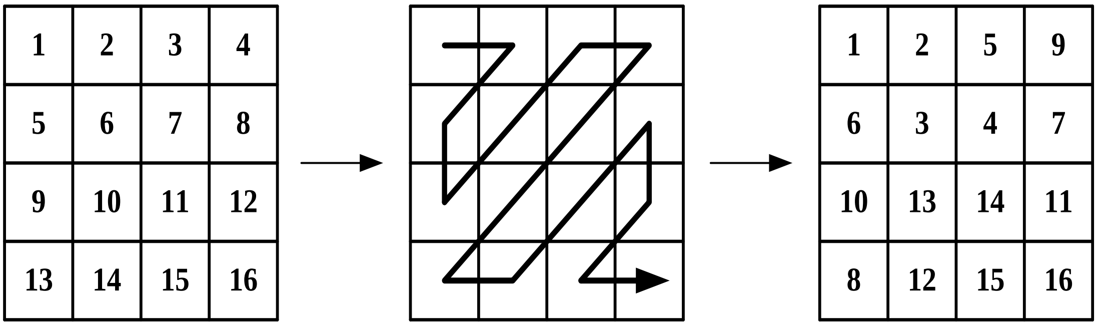


Figure 4.2 Block diagram of 1-D DWT using 2-D SRA.

In the second benchmark, since the discrete cosine transform (DCT) is an integral part of a JPEG compression system, the zigzag scanner is used in ordering the DCT coefficient for efficient are placed before high-frequency coefficients for efficient entropy. In the zigzag scanner benchmark, the zigzag scanner rearranges the coefficient into a 2-D array sorted from the DC value to the highest-order spatial frequency coefficient as shown in Figure 4.3. This is accomplished using zigzag sorting [13], a process which traverses the 4x4 block in a back-and-forth direction of increasing spatial frequency.

We apply the 2-D SRA approach to improve the transition activities. The resulting allocation table and the improved zigzag scanner architecture are depicted in Figure 4.4 and 4.5, respectively, where the same notations are adopted as that in Figure 6 of [4].



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

4x4 quantized DCT matrix
output indices

Zigzag scan pattern

1	2	5	9
6	3	4	7
10	13	14	11
8	12	15	16

Resulting 4x4 format

Figure 4.3 Illustration of zigzag scan for efficient coefficient encoding.

cycle	input	R1	R2	R3	R4	R5	R6	R7	R8	output
0	d_1 d_2 d_3 d_4									
1	d_5 d_6 d_7 d_8	d_1	d_2	d_3	d_4					
2	d_9 d_{10} d_{11} d_{12}	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_1 d_2 d_5 d_9
3	d_{13} d_{14} d_{15} d_{16}	d_{10}	d_{11}	d_3	d_4	d_{12}	d_6	d_7	d_8	d_6 d_3 d_4 d_7
0		d_{10}	d_{11}	d_{13}	d_{14}	d_{12}	d_{15}	d_{16}	d_8	d_{10} d_{13} d_{14} d_{11}
1						d_{12}	d_{15}	d_{16}	d_8	d_8 d_{12} d_{15} d_{16}

Iteration Period

Figure 4.4 Allocation table for zigzag scanner using 2-D SRA.

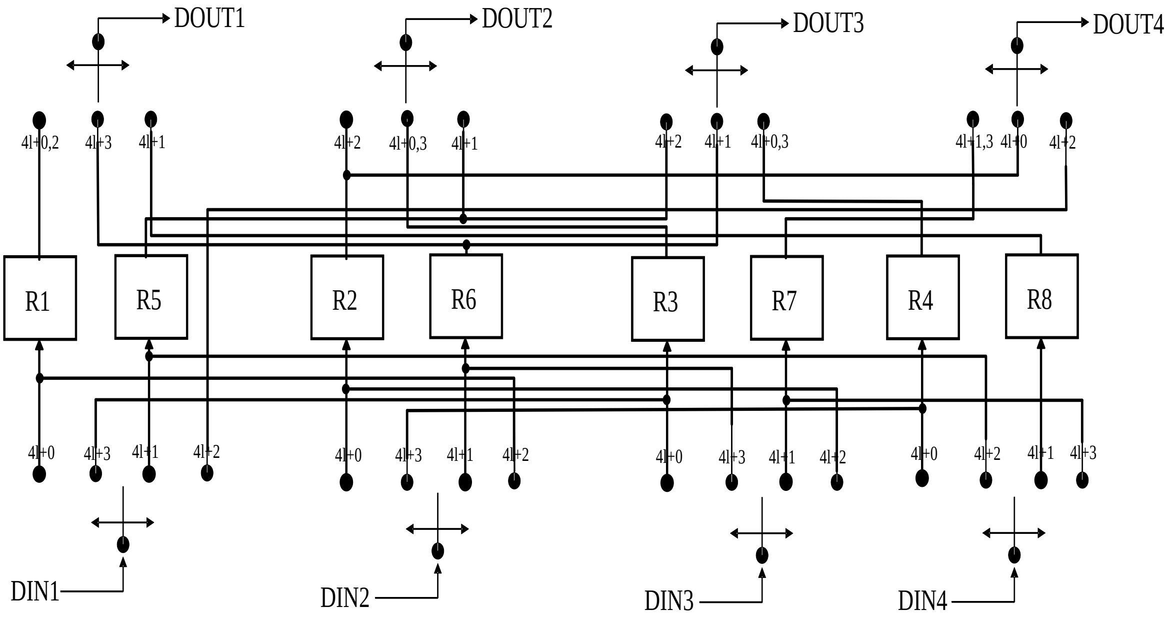
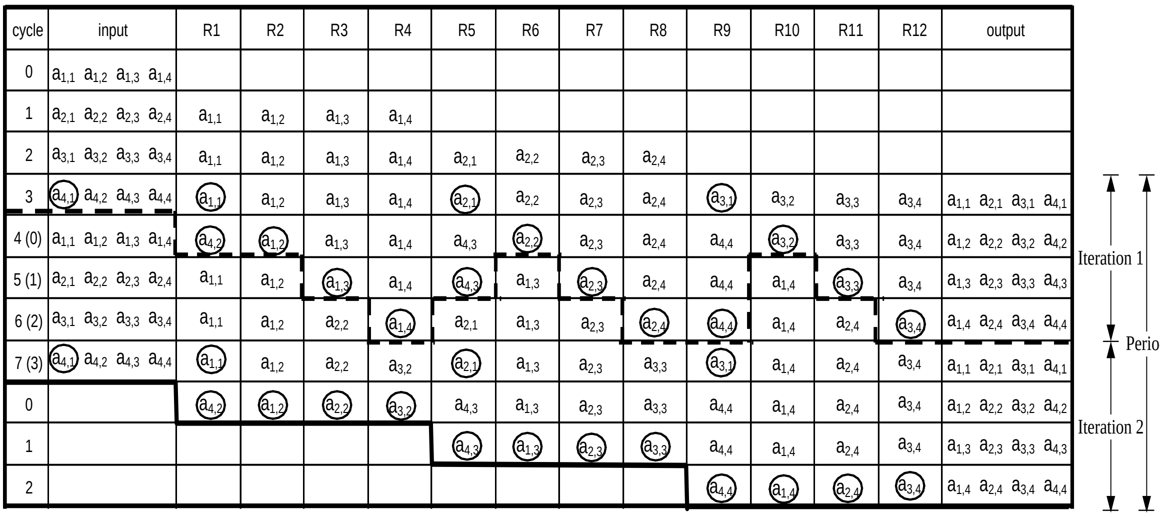


Figure 4.5 Block diagram of zigzag scanner using 2-D SRA.

In the third benchmark, assume X_{4x4} and Y_{4x4} denote the input matrix and the transpose matrix of Y_{4x4}, respectively, where the relationship is expressed in (4.1)

$$\mathbf{Y}_{4x4} = \mathbf{X}_{4x4}^{T} . \tag{4.1}$$

Let $\mathbf{X} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$, we obtain $\mathbf{Y} = \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} \\ a_{1,2} & a_{2,2} & a_{3,2} & a_{4,2} \\ a_{1,3} & a_{2,3} & a_{3,3} & a_{4,3} \\ a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} \end{bmatrix}$, where the input and output data sequences are scanned in $\{a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}\}$, $\{a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4}\}$, $\{a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}\}$, $\{a_{4,1}, a_{4,2}, a_{4,3}, a_{4,4}\}$ and $\{a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}\}$, $\{a_{1,2}, a_{2,2}, a_{3,2}, a_{4,2}\}$, $\{a_{1,3}, a_{2,3}, a_{3,3}, a_{3,4}\}$, $\{a_{1,4}, a_{2,4}, a_{3,4}, a_{4,4}\}$, respectively. The length of both sequences is four. Note that the period of computation is four cycles, and four samples are processed each clock cycle. In step 1, using the lifetime analysis in [2], the minimum number of registers is 12. In step 2, we use allocation table to assign the input variables to available registers in ascending order as shown in Figure 4.6. Thus, the input sequences $\{a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}\}$, $\{a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4}\}$ and $\{a_{3,1}, a_{3,2}, a_{3,3}, a_{3,4}\}$ are inputted to the corresponding registers $\{R_1, R_2, R_3, R_4\}$, $\{R_5, R_6, R_7, R_8\}$ and $\{R_9, R_{10}, R_{11}, R_{12}\}$. For next sequence data $\{a_{4,2}, a_{4,3}, a_{4,4}\}$, we need to return to the first register and search for which register is available from R_1 to R_8 in step 3. In this case, R_1, R_5 and R_9 are available for input sequence data $\{a_{4,2}, a_{4,3}, a_{4,4}\}$. From step 3, since the lifetime of $a_{4,2}$ is shorter than the lifetime of $a_{4,3}$ and $a_{4,4}$, the input data $a_{4,2}$ is inputted to the R_1 register. since the lifetime of $a_{4,3}$ is shorter than the lifetime of $a_{4,4}$, the input data $a_{4,3}$ is inputted to the R_5 register and the input data $a_{4,4}$ is inputted to the R_9 register.


Iteration 1
Period
Iteration 2

cycle	input	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	output
0	$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$													
1	$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$									
2	$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$					
3	$a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$
4 (0)	$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$	$a_{4,2}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{4,3}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$
5 (1)	$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{4,3}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{1,4}$	$a_{3,3}$	$a_{3,4}$	$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$
6 (2)	$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{1,4}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{2,4}$	$a_{4,4}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$ $a_{2,4}$ $a_{3,4}$ $a_{4,4}$
7 (3)	$a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$	$a_{1,1}$	$a_{1,2}$	$a_{2,2}$	$a_{3,2}$	$a_{2,1}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{3,1}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{1,1}$ $a_{2,1}$ $a_{3,1}$ $a_{4,1}$
0		$a_{4,2}$	$a_{1,2}$	$a_{2,2}$	$a_{3,2}$	$a_{4,3}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{4,4}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{1,2}$ $a_{2,2}$ $a_{3,2}$ $a_{4,2}$
1						$a_{4,3}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$	$a_{4,4}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{1,3}$ $a_{2,3}$ $a_{3,3}$ $a_{4,3}$
2										$a_{4,4}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$	$a_{1,4}$ $a_{2,4}$ $a_{3,4}$ $a_{4,4}$

Figure 4.6 Allocation table for 4x4 par-transposer using 2-D SRA.

In step 4, we repeat steps 2 and 3 recursively until one iteration allocation is done as shown in Figure 4.6, where the dash-bold line denotes the boundary of one iteration. Equivalently, one computation process is calculated under one iteration. Finally, in step 5, we repeat steps 2, 3, 4 as required until one period allocation is finished as shown in Figure 4.6, where the solid-bold line denotes the boundary of one period. In this case, the 4x4 par-transposer has four cycles and eight cycles for one iteration and period, respectively.

According to the proposed 5-step 2-D SRA algorithm, the number of transitions can be further minimized compared with that of [4] and [7-8]. In this case, the numbers of transitions for each iteration handled by 2-D register allocation, SSRA, 2-D SRA schemes are 41, 18, and 15, respectively. The corresponding new 4x4 par-transposer architecture is depicted in Figure 4.7.

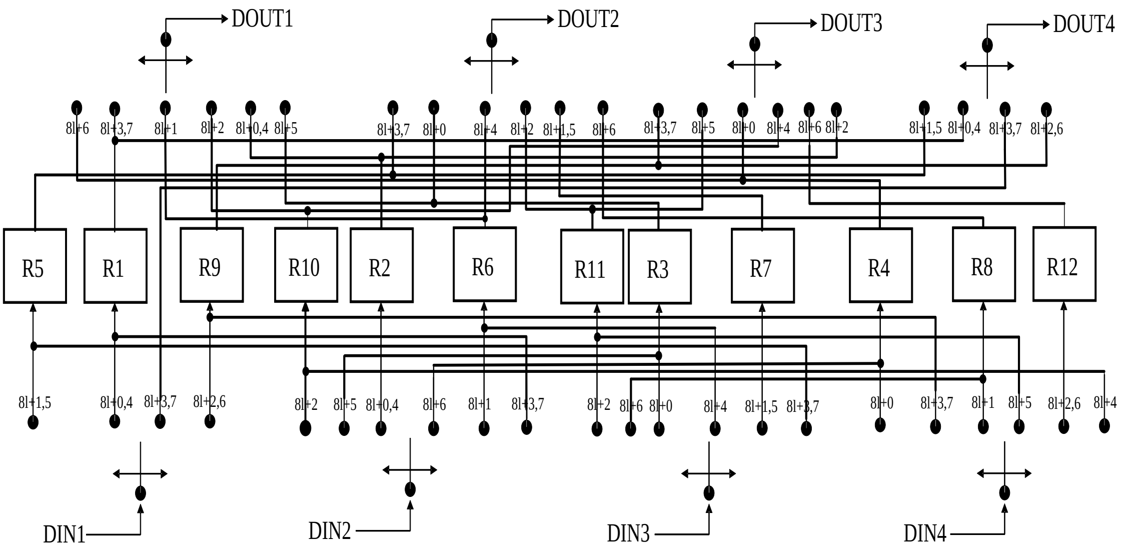


Figure 4.7 Block diagram of 4x4 par-transposer using 2-D SRA.

4.2 Properties of *N* x *N* Par-Transposer Using 2-D SRA

From the 2-D SRA-based allocation table, we observe that the periodic allocation occurs for 4x4, 16x16 and higher-order par-transposer. For example, if register R_j is occupied with a variable in the first cycle, then R_j owns the identical variable in (l + C_P)-th cycle, where C_P denotes the number of cycles for one period. We can calculate C_P by the following properties 2 and 3.

Without loss of the generality, the relationship of the *N*x*N* input matrix and transposed matrix can be expressed in (4.1).

$$\mathbf{Y}_{N\times N} = \mathbf{X}_{N\times N}^{T} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{bmatrix}^{T} = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{N,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{N,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,N} & a_{2,N} & \cdots & a_{N,N} \end{bmatrix} \quad (4.1)$$

We can derive the following properties for *N*x*N* par-transposer.

Property 1: The number of registers N_R equals *N* x (*N*-1) for the *N*x*N* par-transposer.

Proof:

Input and output data sequences in (4.1) can be scanned in the following.

$$X_s = \{a_{1,1}, a_{1,2}, \ldots, a_{1,N}, a_{2,1}, a_{2,2}, \ldots, a_{2,N}, \ldots, a_{N,1}, \ldots, a_{N,N}\} \tag{4.2a}$$

$$Y_s = \{a_{1,1}, a_{2,1}, \ldots, a_{N,1}, a_{1,2}, a_{2,2}, \ldots, a_{N,2}, \ldots, a_{1,N}, \ldots, a_{N,N}\} \tag{4.2b}$$

From (4.2a) and (4.2b), since *N*x*N* par-transposer is multiple inputs and multiple outputs, the number of input variables is *N* for one cycle and the number of output variables is *N* for one cycle. Since *N*x*N* par-transposer is a causal system, we need to shift right the transposed sequence in (4.2b). The variable in (4.2a) compared with that in (4.2b) which has the longest distance will be the aligned point. In this general case, the input sequences $\{a_{1,1}, a_{2,1}, \ldots, a_{N,1}\}$ is the aligned sequence. Thus, the shift-right distance implemented by registers is equal to the number of registers for the *N*x*N* par-transposer. Hence, the required number of registers can be presented in (4.3).

$$N_R = N(N-1)\,. \qquad \blacklozenge \tag{4.3}$$

Property 2: The number of cycles for one iteration, C_I, equals *N* for the *N*x*N* par-transposer.

Proof:

According to the iteration definition and allocation table as addressed in this chapter, we can see that each iteration consumes the cycles equaling the total number of rows in the transposer. Since the *N*x*N* par-transposer has multiple input and multiple output, the number of input variables is *N* every cycle. Thus, for the *N*x*N* par-transposer, the number of cycles for each iteration can be expressed in (4.4).

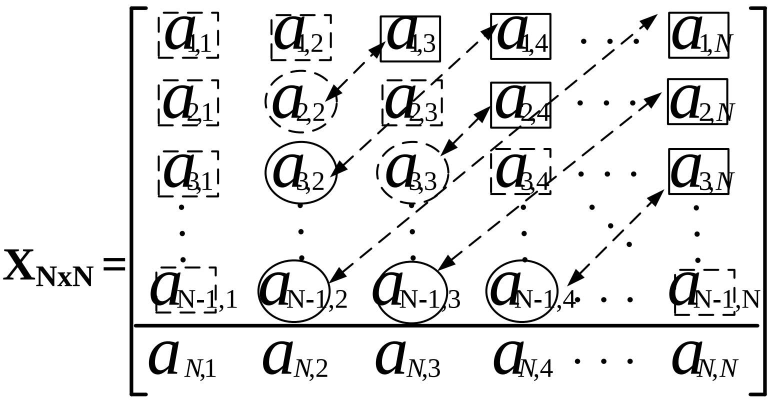
$$C_I = N\,. \qquad \blacklozenge \tag{4.4}$$

Under the multiple-iteration case (i.e., $N \geq 3$), we can obtain Property 3.

Property 3: The number of iterations for one period, $I_{P,}$ equals 2 for the *N*x*N* par-transposer.

Proof:

For convenience of derivation, the input matrix X is repeated in (4.5).


$X_{NxN} =$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,N}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,N}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,N}$
$a_{N-1,1}$ $a_{N-1,2}$ $a_{N-1,3}$ $a_{N-1,4}$ $a_{N-1,N}$
$a_{N,1}$ $a_{N,2}$ $a_{N,3}$ $a_{N,4}$ $a_{N,N}$

(4.5)

At the first iteration, from the SRA approach, the lifetime of input variables $a_{1,3}$, $a_{1,4}$ and $a_{2,4}$ are longer than C_I. That means the input variables $a_{1,3}$, $a_{1,4}$ and $a_{2,4}$ at the next iteration cannot be allocated at R_3, R_4 and R_8 registers and moved to R_6, R_{10} and R_{11} registers, respectively. At the second iteration, the new input variables $a_{1,3}$, $a_{1,4}$ and $a_{2,4}$ have to be fed into the next available registers R_6, R_{10} and R_{11}, respectively, via the SRA approach.

For instance, we show the allocation table of the 4x4 par-transposer in Figure 4.3 to illustrate the situation. The multiple input variables will be only appeared in dedicated registers for $N \geq 3$. As a consequence, the input variables can be separated into Q groups (i.e., G_1, G_2, …, G_Q) to finish the allocation. In the case of the 4x4 par-transposer, there exist nine groups in Figure 4.6. The group G_1 are composed of R_1, the group G_2 are composed of R_2, the group G_3 are composed of R_5, the group G_4 are composed of R_7, the group G_5 are composed of R_9, the group G_6 are composed of R_{12}, the group G_7 are composed of R_3, R_6, the group G_8 are composed of R_4, R_{10}, the group G_9 consists of R_8, R_{11}.

After transposing the NxN matrix, at the first iteration, some registers store multiple variables and other registers store single variable. The sum of the lifetime of all variables stored at the register under the first iteration can be longer than C_I, less than (C_I -1), or equal to C_I and (C_I -1). Thus, the utilized registers from R_1 to $R_{N(N-1)}$ can be classified into four types. The lifetime of one register storing the single variable is longer than C_I and then is referred to as type-1 register. The lifetime of one register storing the single variable is equal to (C_I -1), and then is named as type-3 register. The lifetime of one register storing the single variable is less than (C_I -1), and then is named as type-5 register. The lifetime of one register storing the single variable is equal to C_I, and then is named as type-6 register. The lifetime of one register storing multiple variables is equal to C_I, and then is named as type-7 register. Thus, for $N \geq 3$, the difference value of the type-3, 6 and 7 registers can be defined as the sum of the lifetime of all variables storing at this register in (4.6).

$$D_{R_i} = C_{LT,R_i} - C_I + 1\,. \tag{4.6}$$

Where C_{LT,R_i} denotes the lifetime of the register R_i. For input variables marked in dash-line circle, they belong to the type-3 registers. The difference values of the corresponding registers are equal to zero in (4.6), where the specified input variables $a_{i,j}$ are allocated at $R_{((i\text{-}1)\mathrm{x}N+j)}$. For other input variables marked in the dash-line square box, they belong to the type-6 and 7 registers. The difference values of the corresponding registers are equal to one in (4.6), where the specified input variables $a_{i,j}$ are allocated at $R_{((i\text{-}1)\mathrm{x}N+j)}$. According to the above calculation, the input variable belongs to which register can be determined.

The difference of the type-1 and type-5 registers can be calculated from the corresponding variables owing to the single variable. The difference of type-1 register designated by the square box as shown in (4.5) can be generally represented in (4.7).

$$\begin{aligned} &D_{a_{i,i+k}} = C_{LT,a_{i,i+k}} - C_I = k \\ &for\, 1 \leq i \leq (N-2),\, k = 2,\ldots,(N-1), \text{and } N \geq 3 \end{aligned} \tag{4.7}$$

where $C_{LT,a_{i,j}}$ denotes the life time of the input variable $a_{i,j}$. Each detailed equation can be obtained from (4.7) as follows. $Da_{1,3} = Da_{2,4} = \ldots = Da_{N\text{-}2,N} = 2$, $Da_{1,4} = Da_{2,5} = \ldots = Da_{N\text{-}4,N\text{-}1} = 3$, $Da_{1,5} = Da_{2,6} = \ldots = Da_{N\text{-}5,N\text{-}1} = 4$, ..., $Da_{1,N} = (N\text{-}1)$, where $a_{i,j}$ will be allocated at $R_{((i\text{-}1)\mathrm{x}N+j)}$. In similar behavior, at the first iteration, the difference of each type-5 register designated by circle box as shown in (4.5) can be generally represented in (4.8).

$$D_{a_{i,i+k}} = C_{LT,a_{i,i+k}} - C_I = -k \qquad (4.8)$$
$$for\ 2 \le i \le (N-1), k = 1,2,...,(N-3), N \ge 3$$

Each detailed equation can be obtained from (4.8) as follows. $Da_{3,2} = Da_{4,3} = ... = Da_{N\text{-}1,N\text{-}2} = \text{-}1$, $Da_{4,2} = Da_{5,3} = ... = Da_{N\text{-}1,N\text{-}3} = \text{-}2$, ..., $Da_{N\text{-}1,2} = \text{-}(N\text{-}3)$, where $a_{i,j}$ will be allocated at $R_{((i\text{-}1)\mathrm{x}N+j)}$. According to (4.6), (4.7) and (4.8), we can calculate the matrix **D** in (4.9) to represent the relationship of difference of registers.

$$\mathbf{D}_{\mathbf{NxN}} = \left[\begin{array}{ccccc} D_{R_1} & D_{R_2} & D_{R_3} & D_{R_4} \cdots & D_{R_N} \\ D_{R_{N+1}} & D_{R_{N+2}} & D_{R_{N+3}} & D_{R_{N+4}} \cdots & D_{R_{2N}} \\ D_{R_{2N+1}} & D_{R_{2N+2}} & D_{R_{2N+3}} & D_{R_{2N+4}} \cdots & D_{R_{3N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D_{R_{N(N-2)+1}} & D_{R_{N(N-2)+2}} & D_{R_{N(N-2)+3}} & D_{R_{N(N-2)+4}} \cdots & D_{R_{N(N-1)}} \\ \hline X & X & X & X \cdots & X \end{array}\right] \qquad (4.9)$$

In (4.5), the input variables, $a_{1,3}$ and $a_{2,2}$ are one group, $a_{1,4}$ and $a_{3,2}$ are one group, …, $a_{N\text{-}1,2}$ and $a_{1,N}$ are one group, $a_{3,3}$ and $a_{2,4}$ are one group, $a_{N\text{-}1,N\text{-}1}$ and $a_{N\text{-}2,N}$ are one group. The cycles of D_{G1}, D_{G1}, …, D_{GQ} can be derived as

$$D_{G1} = Da_{1,3} + Da_{2,2} = 2 + 0 = 2$$

$$D_{G2} = Da_{1,4} + Da_{3,2} = 3 - 1 = 2$$

$$D_{G3} = Da_{1,5} + Da_{4,2} = 4 - 2 = 2$$

…

$$D_{GQ} = Da_{N-1,N-1} + Da_{N-2,N} = 0 + 2 = 2$$

$$D_G = D_{G1} = D_{G2} = D_{G3} = \ldots = D_{GQ} = 2, \quad N \ge 3 \qquad (4.10)$$

where D_{GQ} denotes the sum of difference cycles of registers in G_Q group. Otherwise, in (4.9), the register as designated by dash-line square box consists of the register D_{R1}, D_{R2}, $D_{R(N+1)}$, $D_{R(N+3)}$, …, $D_{R(N(N-2)+1)}$, $D_{R(N(N-1))}$. Single register is one group. From (4.10), the value of D_G will be consumed by D_G iterations. That means one complete periodic allocation process can be finished after D_G iterations. Thus, we obtain (4.11) as follows.

$$I_P = 2\,, \quad N \geq 3\,. \qquad \blacklozenge \qquad (4.11)$$

Example 1: For the 3x3 par-transposer whose input matrix is shown in (4.12), at the first iteration, the life time cycle of each register is listed from (4.13a) to (4.13f) and the 3x3 matrix **D** is shown in (4.14).

$$\mathbf{X_{3x3}} = \left[\begin{array}{ccc} \boxed{a_{1,1}} & \boxed{a_{1,2}} & \boxed{a_{1,3}} \\ \boxed{a_{2,1}} & \textcircled{$a_{2,2}$} & \boxed{a_{2,3}} \\ \hline a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \qquad (4.12)$$

and

$$D_{R1} = 4 - 3 = 1, \qquad (4.13a)$$

$$D_{R2} = 4 - 3 = 1, \qquad (4.13b)$$

$$D_{R3} = 5 - 3 = 2, \qquad (4.13c)$$

$$D_{R4} = 4 - 3 = 1, \qquad (4.13d)$$

$$D_{R5} = 3 - 3 = 0, \qquad (4.13e)$$

$$D_{R6} = 4 - 3 = 1, \qquad (4.13f)$$

$$\mathbf{D_{3x3}} = \left[\begin{array}{ccc} \boxed{1} & \boxed{1} & \boxed{2} \\ \boxed{1} & \textcircled{0} & \boxed{1} \\ \hline x & x & x \end{array}\right] \qquad (4.14)$$

In this case, there exist five groups. The group G_1 consists of R_3 and R_5, the group G_2 consists of single register R_1, the group G_3 consists of single register R_2, the group G_4

consists single register R$_4$ and the group G$_5$ consists single register R$_5$. We obtain the number of iterations by summing (4.13c) and (4.13e). I_p=D$_{G1}$=D$_{R3}$+D$_{R5}$=2. Thus, we require two iterations to complete one period allocation for the 3x3 par-transposer. ◆

Example 2: For the 4x4 par-transposer whose input matrix is shown in (4.15), at the first iteration, the life time cycle of each register is listed from (4.16a) to (4.16l) and the 4x4 matrix D is shown in (4.17).

$$\mathbf{X}_{4x4}=\left[\begin{array}{cccc} \boxed{a_{1,1}} & \boxed{a_{1,2}} & \boxed{a_{1,3}} & \boxed{a_{1,4}} \\ \boxed{a_{2,1}} & \enclose{circle}{a_{2,2}} & \boxed{a_{2,3}} & \boxed{a_{2,4}} \\ \boxed{a_{3,1}} & \enclose{circle}{a_{3,2}} & \enclose{circle}{a_{3,3}} & \boxed{a_{3,4}} \\ \hline a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{array}\right] \tag{4.15}$$

and

$$D_{R1} = 5 - 4 = 1, \tag{4.16a}$$

$$D_{R2} = 5 - 4 = 1, \tag{4.16b}$$

$$D_{R3} = 6 - 4 = 2, \tag{4.16c}$$

$$D_{R4} = 7 - 4 = 3, \tag{4.16d}$$

$$D_{R5} = 5 - 4 = 1, \tag{4.16e}$$

$$D_{R6} = 4 - 4 = 0, \tag{4.16f}$$

$$D_{R7} = 5 - 4 = 1, \tag{4.16g}$$

$$D_{R8} = 6 - 4 = 2, \tag{4.16h}$$

$$D_{R9} = 5 - 4 = 1, \tag{4.16i}$$

$$D_{R10} = 3 - 4 = -1, \tag{4.16j}$$

$$D_{R11} = 4 - 4 = 0, \tag{4.16k}$$

$$D_{R12} = 5 - 4 = 1, \tag{4.16l}$$

$$\mathbf{D}_{4x4}=\left[\begin{array}{cccc} \boxed{1} & \boxed{1} & \boxed{2} & \boxed{3} \\ \boxed{1} & \enclose{circle}{0} & \boxed{1} & \boxed{2} \\ \boxed{1} & \enclose{circle}{-1} & \enclose{circle}{0} & \boxed{1} \\ \hline x & x & x & x \end{array}\right] \tag{4.17}$$

Note that there exist nine groups. The group G_1 is composed of R_3, R_6, the group G_2 consists of R_4, R_{10} and the group G_3 consists of R_8, R_{11}. Otherwise, from G_4 to G_9, the group consists of single register. We obtain the number of iterations by summing (4.16c) and (4.16f), (4.16d) and (4.16j), (4.16h) and (4.16k), respectively. I_p = $D_{G1} = D_{R3}+D_{R6} = D_{G2} = D_{R4}+D_{R10} = D_{G3} = D_{R8}+D_{R11} = 2$. Thus, we require two iterations to complete one period allocation for the 4x4 par-transposer. ◆

Hence, we are able to calculate the number of cycles for one period via multiplying (4.4) and (4.11) for multiple iterations as expressed in upper part of (4.18). For single iteration, C_P=C_I as expressed in lower part of (4.18).

$$C_P=\begin{cases} 2N & for\ \ N \geq 3 \\ C_I = N & for\ \ N = 2 \end{cases}. \tag{4.18}$$

In the case of the 3x3 par-transposer, we can directly use (4.18) to calculate the number of cycles for one period (i.e., C_P = 6).

4.3 Control Unit of DFC Design Using 2-D SRA

Due to multiple input and multiple output, thus, we use several multiplexers to select data from input port and use several multiplexers to select data output. The architecture of control unit using 2-D SRA is shown in Figure 4.8.

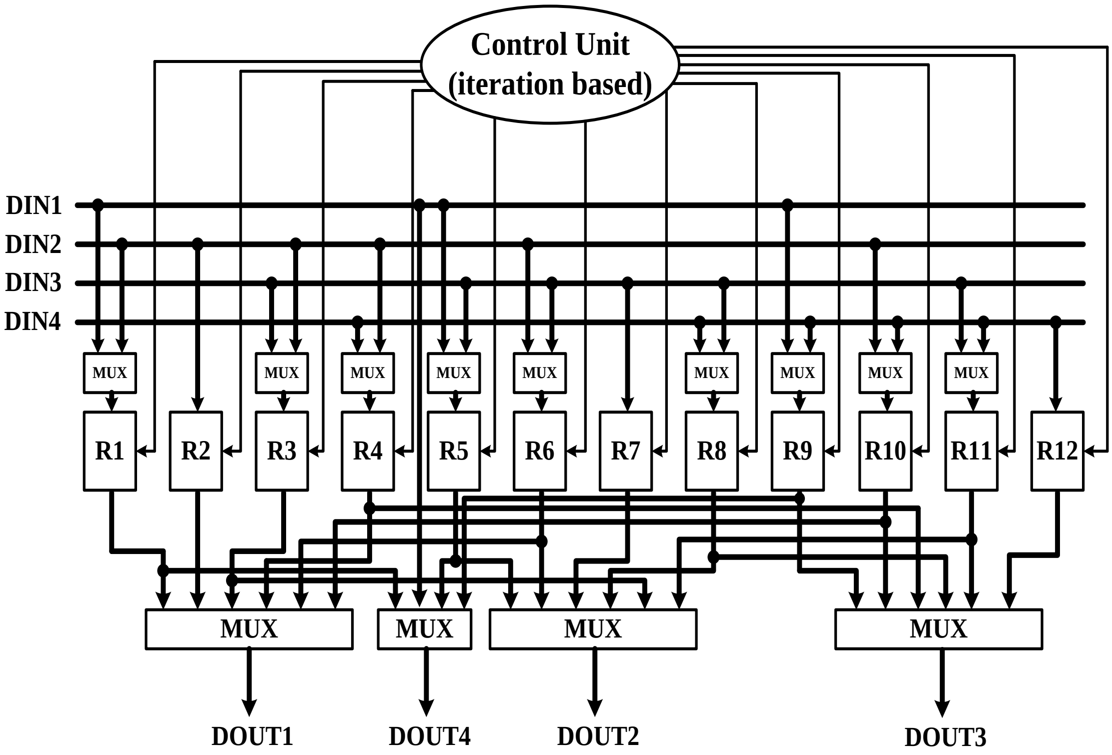


Figure 4.8 Architecture of 4x4 transposer using 2-D SRA.

Chapter 5

Comparison and Simulation Results

In this chapter, first, we will compare the number of register transitions using FBRA, SSRA, 1-D SRA for the DFC designs and compare the number of register transitions using 2-D register allocation, SSRA, 2-D SRA for the 2-D DFC designs.

Second, we compare the results of post layout simulation obtained by the existing methods [2-3] and [7-8] with the proposed 1-D SRA scheme and compare the results of post layout simulation obtained by the existing methods [4] and [7-8] with the proposed 2-D SRA scheme.

Finally, we will implement the interleaver for WiMAX system using the 2-D SRA approach and compare the power consumption with the conventional memory-based design.

5.1 Comparison Results of the Number of Transitions

In the case of the 3x3 transposer, the number of transitions for each iteration handled by FBRA, SSRA, 1-D SRA schemes are 36, 11 and 8, respectively. In the case of 4x4 transposer, the number of transitions for each iteration handled by FBRA, SSRA, 1-D SRA schemes are 144, 24 and 15, respectively. For large size case, 16x16 transposer has 57600, 570, and 255 transitions for each iteration performed by FBRA, SSRA, and 1-D SRA schemes, respectively.

For the higher-order transposer performed by FBRA, SSRA, and 1-D SRA, we partition the number of transitions into several parts and use the equation of parameter *N* to replace them. Thus, for the *N*x*N* transposer, the FBRA, SSRA, and 1-D SRA approaches account for the formulation of the number of register transitions in (5.1), (5.2), and (5.3), respectively.

$$T_{FBRA} = (N-1)^2 \times N^2, \quad N \geq 2. \tag{5.1}$$

$$T_{SSRA} = \begin{cases} (N^2-1)+(N-1)^2-1, & N=3 \\ (N^2-1)+(N-1)^2-1+(N^2-5N+6)/2, & N \geq 4 \end{cases}. \tag{5.2}$$

$$T_{SRA} = N^2-1, \quad N \geq 2. \tag{5.3}$$

As a consequence, the 1-D SRA scheme can lead to lowest transition for the *N*x*N* transposer as listed in Table 5.1.

Table 5.1: Comparison results of the number of transitions among FBRA, SSRA, 1-D SRA approaches.

Benchmarks	FBRA [2-3]	SSRA [7]	1-D SRA
3x3 Transposer	36	11	8
4x4 Transposer	144	24	15
16x16 Transposer	57600	570	255
*N*x*N* Transposer	$(N-1)^2 \times N^2$	2.5N^2-4.5N+2	N^2-1
IIR Filter	6	4	2

On the other hand, In the case of 1-D DWT, the number of transitions for each iteration handled by 2-D register allocation, SSRA, 2-D SRA schemes are 28, 16 and 14, respectively. In the case of the 4x4 par-transposer, the number of transitions for each iteration handled by 2-D register allocation, SSRA, 2-D SRA schemes are 41, 18 and 15, respectively. For large size case, 16x16 par-transposer has 1240, 360, and 255 transitions for each iteration performed by 2-D register allocation, SSRA, and 2-D SRA schemes, respectively.

Comparison results of the number of transitions as listed in Table 5.2 among 2-D register allocation, SSRA, and the proposed 2-D SRA schemes.

Table 5.2: Comparison results of the number of transitions among 2-D register allocation, SSRA, and the proposed 2-D SRA approaches.

Benchmarks	2-D register allocation [4]	SSRA [7-8]	2-D SRA
1D-DWT	28	16	14
Zig-zag Scan	25	15	15
4x4 par- transposer	41	18	15
16x16 par-transposer	1240	360	255

5.2 Simulation and Implementation Results

In this chapter, the comprehensive comparison results as listed in Tables 5.3 and 5.4 among the FBRA, SSRA and the proposed 1-D SRA schemes are presented. For 3x3, 4x4, 16x16 transposer and IIR filter, in terms of register transitions, the proposed 1-D SRA can save up to 27.3%, 37.5%, 55.3%, and 50% compared with that of SSRA approach in [7-8].

Concerning the power consumption and core area measurement, the cell-based design flow with standard cell library is adopted and the four benchmarks have been implemented in 0.18 um CMOS process. Synopsys Design Compiler and Cadence SOC

Encounter are employed to synthesize the RTL design with the constraint of 10 ns and place and route, respectively. The post layout power and core area of the proposed and other schemes among four benchmarks are listed in Tables 5.3 and 5.4. In terms of power consumption of 16-bit 4x4 transposer, the power saving of the proposed 1-D SRA and conventional FBRA schemes can be achieved by 45.3% and -18.8% compared with that of the SSRA scheme, respectively. On the other hand, the proposed 1-D SRA and conventional FBRA schemes can save the area size by 51% and 54.9% compared with that of the SSRA scheme, respectively. Note that the SSRA scheme has the largest area cost and FBRA scheme is the most power hungry design among three DFC designs. For larger size case, 16x16 transposer, the proposed SRA scheme still outperforms other two schemes in terms of power saving. The layout of 16-bit 16x16 transposer is shown in Figure 5.1.

As a consequence, compared with representative register allocation designs as exposed in Tables 5.1, 5.3 and 5.4, the proposed 1-D SRA design possesses the lowest register transition and power consumption with slightly increment of hardware overhead.

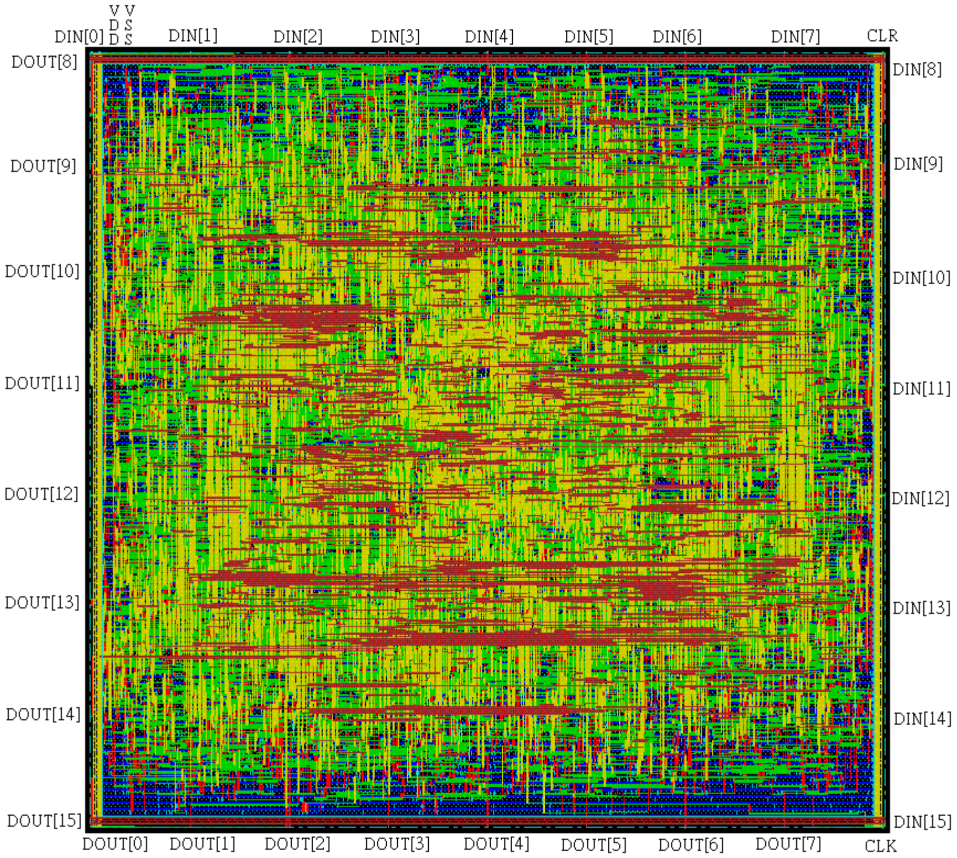


Figure 5.1 Layout of 16-bit 16x16 transposer using 1-D SRA.

Table 5.3: Comparison results of power consumption among four benchmarks (uW).

Benchmarks	# of bits	FBRA [2-3]	SSRA [7-8]	1-D SRA
3x3 Transposer	8-bit	668 (112.9%)	591.8 (100%)	434.9 (73.5%)
	16-bit	1261 (119.5%)	1055 (100%)	765.6 (72.6%)
4x4 Transposer	8-bit	1154 (119.7%)	964 (100%)	601 (62.3%)
	16-bit	2190 (118.8%)	1844 (100%)	1008 (54.7%)
16x16 Transposer	8-bit	18450 (182%)	10140 (100%)	5135 (50.6%)
	16-bit	36310 (186.2%)	19500 (100%)	9716 (49.8%)
IIR Filter	8-bit	685.3 (167.9%)	408.2 (100%)	340.9 (83.5%)
	16-bit	1258 (161.5%)	779.1 (100%)	579 (74.3%)

Table 5.4: Comparison results of chip area among four benchmarks (um^2).

Benchmarks	# of bits	FBRA [2-3]	SSRA [7-8]	1-D SRA
3x3 Transposer	8-bit	3483 (56.2%)	6203 (100%)	3764.9 (60.7%)
	16-bit	6238.1 (55.1%)	11325.4 (100%)	6269.5 (55.4%)
4x4 Transposer	8-bit	6735.1 (45.3%)	14851.5 (100%)	7858.1 (52.9%)
	16-bit	12278.3 (45.1%)	27218.4 (100%)	13337.2 (49%)
16x16 Transposer	8-bit	124587.8 (28.2%)	442502.8 (100%)	220666.3 (49.9%)
	16-bit	242662.9 (31%)	782386.9 (100%)	361052.5 (46.1%)
IIR Filter	8-bit	2320.8 (66.4%)	3494.7 (100%)	2469.4 (70.7%)
	16-bit	4480.9 (64.9%)	6907.2 (100%)	4285.3 (62%)

On the other hand, the comprehensive comparison results as listed in Tables 5.5 and 5.6 among the 2-D register allocation, SSRA and the proposed 2-D SRA schemes are presented. For 1-D DWT, 4x4 par-transposer and 16x16 par-transposer, in terms of register transitions, the proposed 2-D SRA can save up to 12.5%, 16.7% and 29.2% compared with that of SSRA approach in [7-8].

Concerning the power consumption and core area measurement, the cell-based design flow with standard cell library is adopted and the four benchmarks have been implemented in 0.18 um CMOS process. Synopsys Design Compiler and Cadence SOC Encounter are employed to synthesize the RTL design with the constraint of 10 ns and place and route, respectively. The post layout power and core area of the proposed and other schemes among four benchmarks are listed in Tables 5.5 and 5.6. In terms of power consumption of 16-bit 4x4 par-transposer, the power saving of the proposed 2-D SRA and conventional 2-D register allocation schemes can be achieved by 16.1% and

-14.9% compared with that of the SSRA scheme, respectively. On the other hand, the proposed 2-D SRA and conventional 2-D register allocation schemes can save the area size by 26.4% and 37.6% compared with that of the SSRA scheme, respectively. Note that SSRA scheme has the largest area cost and 2-D register scheme is the most power hungry design among three 2-D DFC designs. For larger size case, 16x16 par-transposer, the proposed 2-D SRA scheme still outperforms other two schemes in terms of power saving.

As a consequence, compared with representative register allocation designs as exposed in Tables 5.2, 5.5 and 5.6, the proposed 2-D SRA design possesses the lowest register transition and power consumption with slightly increment of hardware overhead.

Table 5.5: Comparison results of power consumption among three benchmarks (uW).

Benchmarks	# of bits	2-D register allocation [4]	SSRA [7-8]	2-D SRA
1D-DWT	8-bit	1732 (106.1%)	1632 (100%)	1579 (96.8%)
	16-bit	3211 (107.8%)	2980 (100%)	2822 (94.7%)
Zigzag Scanner	8-bit	1810 (113.3%)	1597 (100%)	1328 (83.2%)
	16-bit	3650 (108.9%)	3352 (100%)	2897 (86.4%)
4x4 par-transposer	8-bit	2449 (127.4%)	1923 (100%)	1617 (84%)
	16-bit	4730 (114.9%)	4116 (100%)	3454 (83.9%)

Table 5.6: Comparison results of chip area among three benchmarks (um^2).

Benchmarks	# of bits	2-D register allocation [4]	SSRA [7-8]	2-D SRA
1D-DWT	8-bit	6292.9 (52.7%)	11944 (100%)	8809.4 (73.8%)
	16-bit	11552.4 (52.4%)	22064.3 (100%)	15681.3 (71.1%)
Zigzag Scanner	8-bit	7744.6 (70.2%)	11032.1 (100%)	7967.8 (72.2%)
	16-bit	14041.4 (62.9%)	22318.6 (100%)	14808.5 (66.4%)
4x4 par-transposer	8-bit	10053.8 (62.3%)	16139.3 (100%)	11830.3 (73.3%)
	16-bit	19128.8 (62.4%)	30673.4 (100%)	22561.3 (73.6%)

5.3 Interleaver Implementation for WiMAX System

In this section, we will introduce the interleaving for communication system and implement the interleaver using the 2-D SRA approach for WiMAX system. Meanwhile, the power consumption compared with the conventional memory-based design will be presented.

5.3.1 Interleaving

According to [14], because most forward error-correction codes are not designed to deal with error bursts, interleaving is applied to randomize the occurrence of bit errors prior to encoding. At the transmitter, the coded bits are permuted in a certain way, which makes sure that adjacent bits are separated by several bits after interleaving. At the receiver, the reverse permutation is performed after decoding. A commonly used interleaving scheme is the block interleaver, where input bits are written in a matrix column by column and read out row by row. Figure 5.2 shows the bit numbers of a block interleaver operating on a block size of 48 bits. After writing the 48 bits in the matrix according to the order as depicted in Figure 5.2, the interleaved bits are read out row by row, so the output bit number are 0, 8, 16, 24, 32, 40, 1, 9, …,47.

0	8	16	24	32	40
1	9	17	25	33	41
2	10	18	26	34	42
3	11	19	27	35	43
4	12	20	28	36	44
5	13	21	29	37	45
6	14	22	30	38	46
7	15	23	31	39	47

Figure 5.2 Interleaving scheme.

5.3.2 Conventional Memory-Based Interleaver for WiMAX

The interleaving for WiMAX has two stages to reorder the sequence. The permutation of the first stage is defined in (5.1). The permutation of the second stage is defined in (5.2). The goal of interleaving for WiMAX is to separate adjacent bits to avoid bursty error.

$$i = (N_{CBPS}/16)(K \bmod 16) + floor(k/16), \quad k = 0,1,\ldots,\ N_{CBPS-1} \qquad (5.1)$$

Where k is the original index, i is the new index and N_{CBPS} is the coded bits per symbol.

$$j = s \times floor(i/s) + (i + N_{CBPS} - floor(16 \times i / N_{CBPS})) \bmod s, \quad \mathrm{i} = 0,1,\ldots,\ N_{CBPS-1} \qquad (5.2)$$

Where i is the original index, j is the new index, N_{CBPS} is the coded bits per symbol and

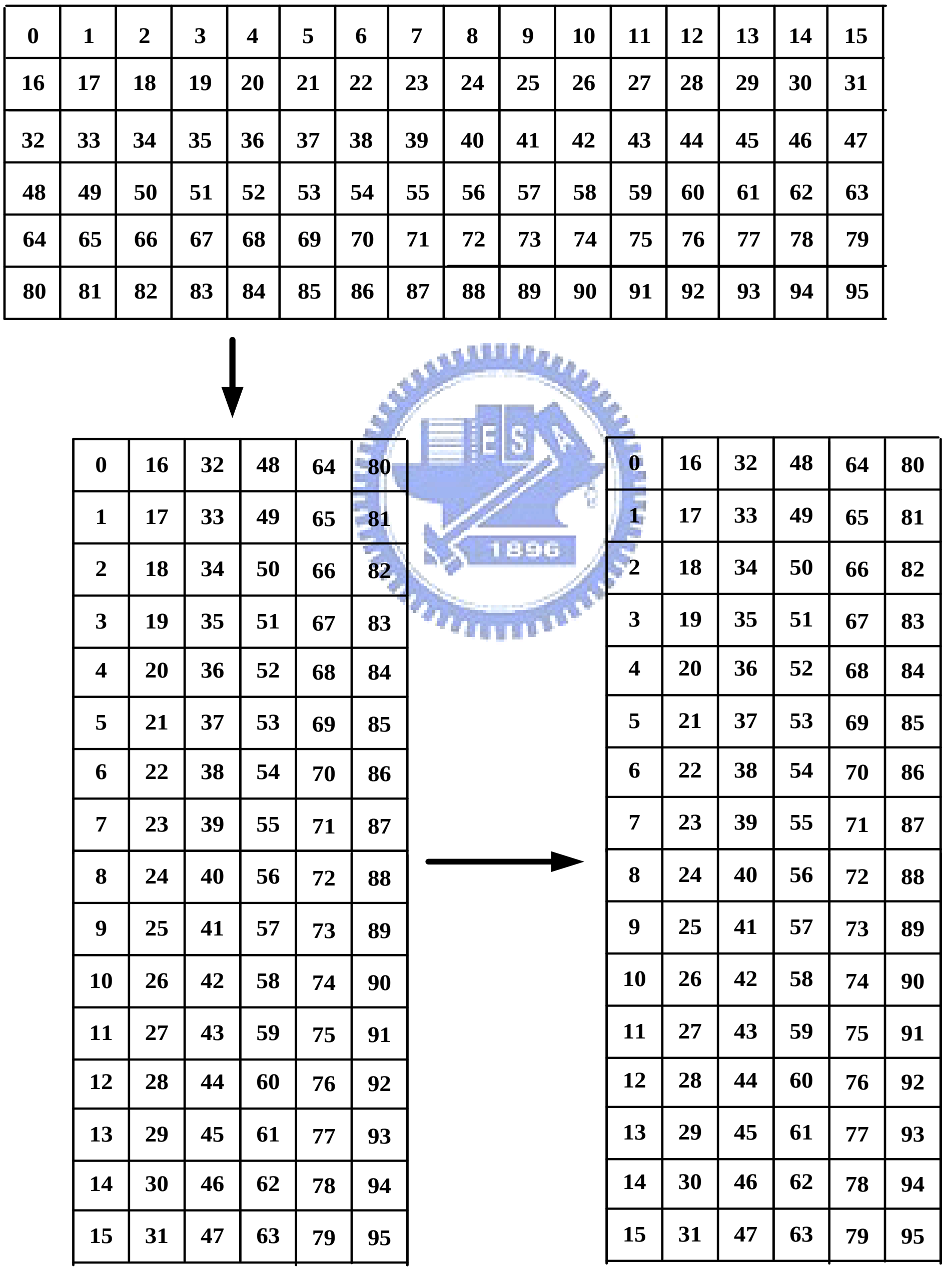
$$s = \max(N_{BPSC}/2,1) \qquad (5.3)$$

Where N_{BPSC} is the number of coded bits per subcarrier.

The rate-dependent parameters are shown in Table 5.7. Note that the interleaver for QPSK has two input bits and two output bits, which they are according to N_{BPSC} in Table 5.7. Since N_{CBPS} is 96 bits, the interleaver for QPSK totally require 96 bits to store the data sequence. The interleaver for 16-QAM has four input bits and four output bits, which they are according to N_{BPSC} in Table 5.7. Since N_{CBPS} is 192 bits, the interleaver for 16-QAM totally requires 192 bits to store the data sequence. The resulting sequence after interleaver for QPSK is shown in Figure 5.3. The resulting sequence after the first stage and the resulting sequence after the second stage are the same. The resulting sequence after interleaving for 16-QAM is shown in Figure 5.4.

Table 5.7: Rate-dependent parameters.

Modulation	Coded Bits per Subcarrier (N_{BPSC})	Coded Bits per OFDM Symbol (N_{CBPS})
QPSK	2	96
16-QAM	4	192



The origin sequence (QPSK N_{CBPS} = 96)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

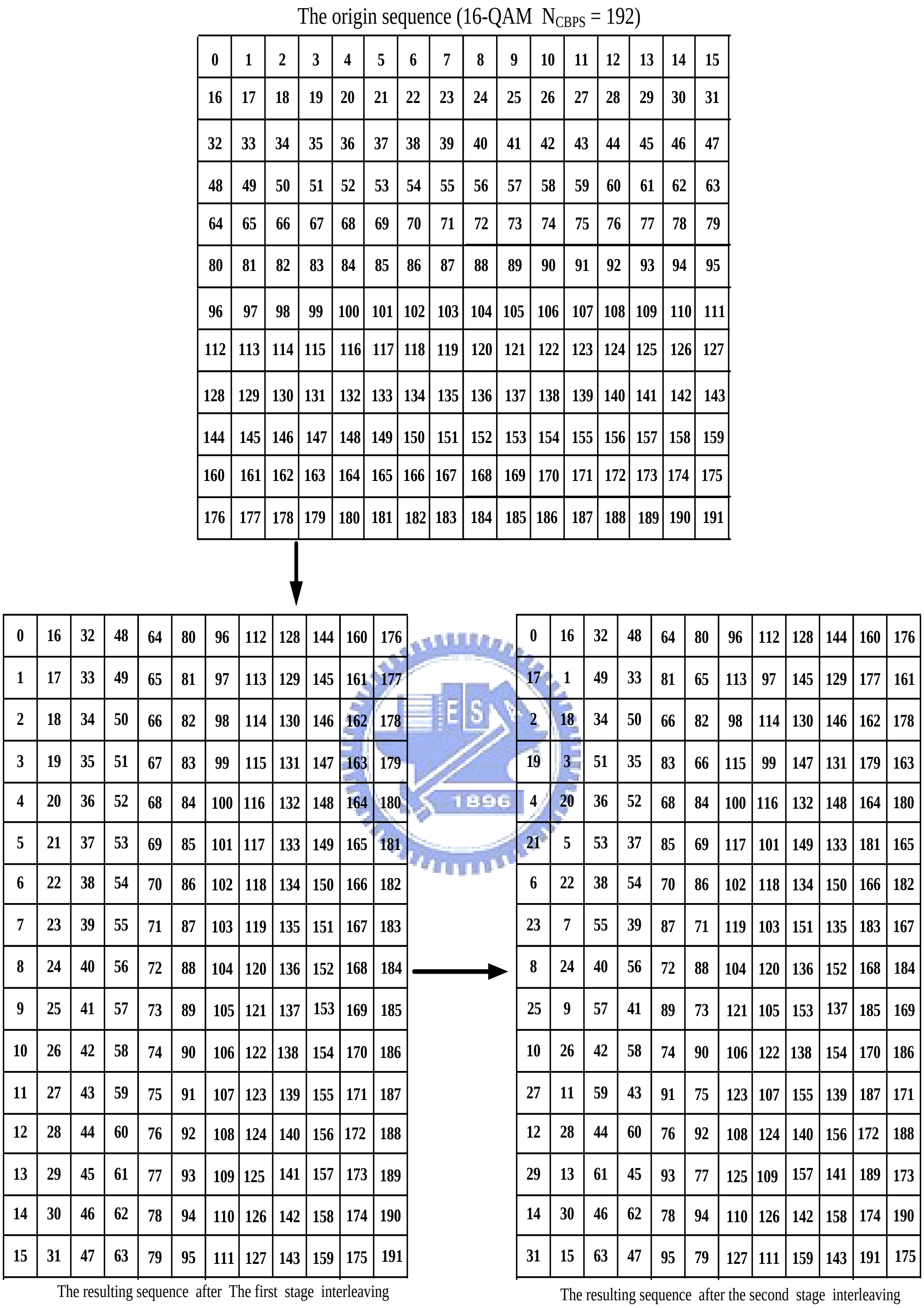
0	16	32	48	64	80
1	17	33	49	65	81
2	18	34	50	66	82
3	19	35	51	67	83
4	20	36	52	68	84
5	21	37	53	69	85
6	22	38	54	70	86
7	23	39	55	71	87
8	24	40	56	72	88
9	25	41	57	73	89
10	26	42	58	74	90
11	27	43	59	75	91
12	28	44	60	76	92
13	29	45	61	77	93
14	30	46	62	78	94
15	31	47	63	79	95

The result sequence after
first stage interleaving

0	16	32	48	64	80
1	17	33	49	65	81
2	18	34	50	66	82
3	19	35	51	67	83
4	20	36	52	68	84
5	21	37	53	69	85
6	22	38	54	70	86
7	23	39	55	71	87
8	24	40	56	72	88
9	25	41	57	73	89
10	26	42	58	74	90
11	27	43	59	75	91
12	28	44	60	76	92
13	29	45	61	77	93
14	30	46	62	78	94
15	31	47	63	79	95

The result sequence after
second stage interleaving

Figure 5.3 Interleaver operation for QPSK of WiMAX.



The origin sequence (16-QAM N_{CBPS} = 192)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191

0	16	32	48	64	80	96	112	128	144	160	176
1	17	33	49	65	81	97	113	129	145	161	177
2	18	34	50	66	82	98	114	130	146	162	178
3	19	35	51	67	83	99	115	131	147	163	179
4	20	36	52	68	84	100	116	132	148	164	180
5	21	37	53	69	85	101	117	133	149	165	181
6	22	38	54	70	86	102	118	134	150	166	182
7	23	39	55	71	87	103	119	135	151	167	183
8	24	40	56	72	88	104	120	136	152	168	184
9	25	41	57	73	89	105	121	137	153	169	185
10	26	42	58	74	90	106	122	138	154	170	186
11	27	43	59	75	91	107	123	139	155	171	187
12	28	44	60	76	92	108	124	140	156	172	188
13	29	45	61	77	93	109	125	141	157	173	189
14	30	46	62	78	94	110	126	142	158	174	190
15	31	47	63	79	95	111	127	143	159	175	191

The resulting sequence after The first stage interleaving

0	16	32	48	64	80	96	112	128	144	160	176
17	1	49	33	81	65	113	97	145	129	177	161
2	18	34	50	66	82	98	114	130	146	162	178
19	3	51	35	83	66	115	99	147	131	179	163
4	20	36	52	68	84	100	116	132	148	164	180
21	5	53	37	85	69	117	101	149	133	181	165
6	22	38	54	70	86	102	118	134	150	166	182
23	7	55	39	87	71	119	103	151	135	183	167
8	24	40	56	72	88	104	120	136	152	168	184
25	9	57	41	89	73	121	105	153	137	185	169
10	26	42	58	74	90	106	122	138	154	170	186
27	11	59	43	91	75	123	107	155	139	187	171
12	28	44	60	76	92	108	124	140	156	172	188
29	13	61	45	93	77	125	109	157	141	189	173
14	30	46	62	78	94	110	126	142	158	174	190
31	15	63	47	95	79	127	111	159	143	191	175

The resulting sequence after the second stage interleaving

Figure 5.4 Interleaver operation for 16-QAM of WiMAX.

Figure 5.5 shows the transmitter block diagram for WiMAX. We implement the conventional memory-based interleaver including the serial-to-parallel (s2p) block and interleaving. The conventional memory based-interleaver designated by dash-line square box in Figure 5.5.

Figure 5.6 shows the conventional memory-based interleaver architecture for WiMAX system. The conventional memory-based interleaver uses one memory bank that is partitioned to six banks and then uses six 16-to-1 multiplexers to select data from six memory banks. Thus, the conventional memory-based interleaver needs to wait until the data is completely written into the memory bank and then the data can be outputted to switch block. First, the output sequence has finished the first permutation. Next, the output sequence of the switch block can be obtained after the second permutation. Finally, the sequence after interleaver is outputted to QAM mapping block.

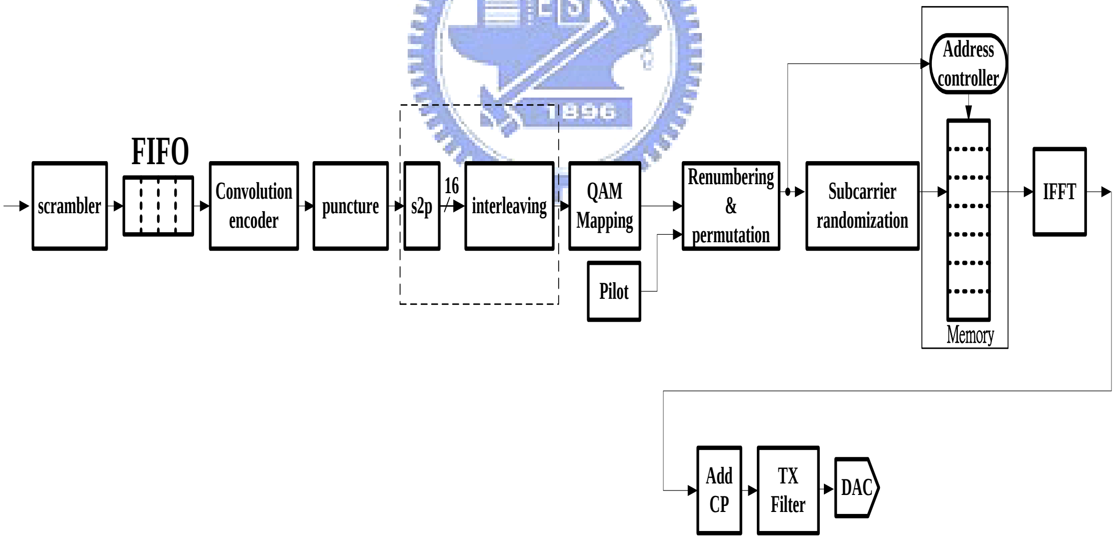


Figure 5.5 Block diagram for WiMAX.

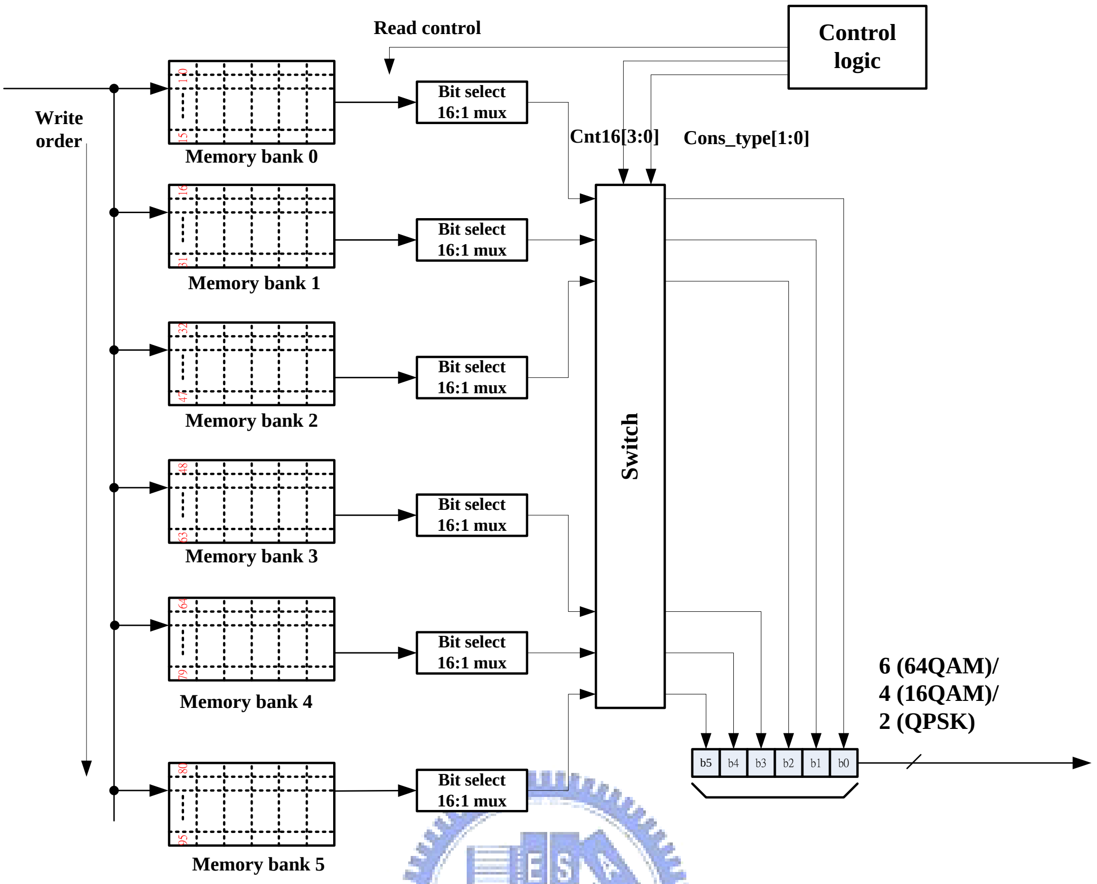


Figure 5.6 Architecture of the conventional memory-based interleaver for WiMAX.

5.3.3 Interleaver for WiMAX Using 2-D SRA

Due to the interleaving for WiMAX at the first stage, the permutation is actually *N*x*M* matrix transposer and the number of input data and output data is lower than that of the par-transposer. The number of multiple iteration cycles will be unexpected. Thus, the properties of the *N*x*N* par-transposer is not suitable for this. In this case, the interleaver using 2-D SRA only computes the single iteration to reduce iteration control overhead. Oppositely, the latency of interleaver using 2-D SRA will be increased.

The number of storage of interleaver using 2-D SRA is the minimum number of registers after using lifetime analysis [2-3]. The minimum number of registers for QPSK and 16-QAM is shown in Table 5.8. Figure 5.7 shows the architecture of interleaver using 2-D SRA. Due to the data width of register is only one bit, the control overhead is larger than that of the conventional memory based design. The minimum number of registers can be partitioned to several banks and control in the same writing signals. The interleaver using 2-D SRA uses two-level multiplexers to reduce the data transitions of multiplexers for output data of register banks.

Table 5.8: Minimum number of registers after lifetime analysis.

Modulation	Original register number	Register number after lifetime analysis
QPSK	96 (100%)	76 (79.2%)
16-QAM	192 (100%)	168 (87.5%)

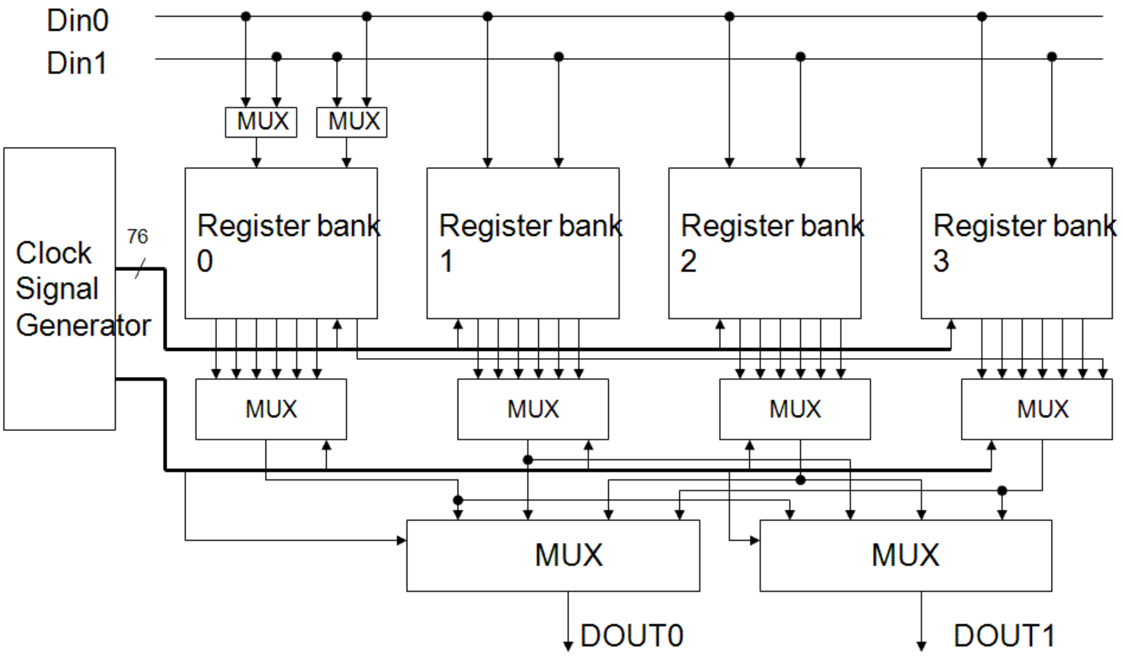


Figure 5.7 Architecture of the interleaver using 2-D SRA for WiMAX.

5.3.4 Implementation and Simulation Result

In this section, the comparison results as listed in Tables 5.8 between the conventional memory-based interleaver and the proposed 2-D SRA scheme are presented. For QPSK and 16-QAM, the proposed 2-D SRA scheme can save up to 20.8% and 12.5% compared with conventional memory-based approach in terms of number of registers.

Concerning the power consumption, the cell-based design flow with standard cell library is adopted and the two benchmarks have been implemented in 0.18 um CMOS process. Artisan Memory Compiler, Synopsys Design Compiler and Cadence SOC Encounter are employed to synthesize the RTL design with the constraint of 10 ns and place and route, respectively. The post layout power of the proposed and the conventional scheme between two benchmarks are listed in Tables 5.9. In terms of power consumption of QPSK and 16-QAM, the power saving of the proposed interleaver using 2-D SRA can be achieved by 52.2% and 65.1% compared with that of the conventional memory-based scheme, respectively. The layout of interleaver using 2-D SRA for 16-QAM WiMAX system is shown in Figure 5.8.

Table 5.9: Comparison results of power consumption among two benchmarks (uW).

Modulation	Method	Power consumption	Latency
QPSK	Conventional memory design	858 (100%)	48 (100%)
	Interleaver using 2-D SRA	410 (47.8%)	38 (79.2%)
16-QAM	Conventional memory design	1488 (100%)	48 (100%)
	Interleaver using 2-D SRA	519.2 (34.9%)	42 (87.5%)

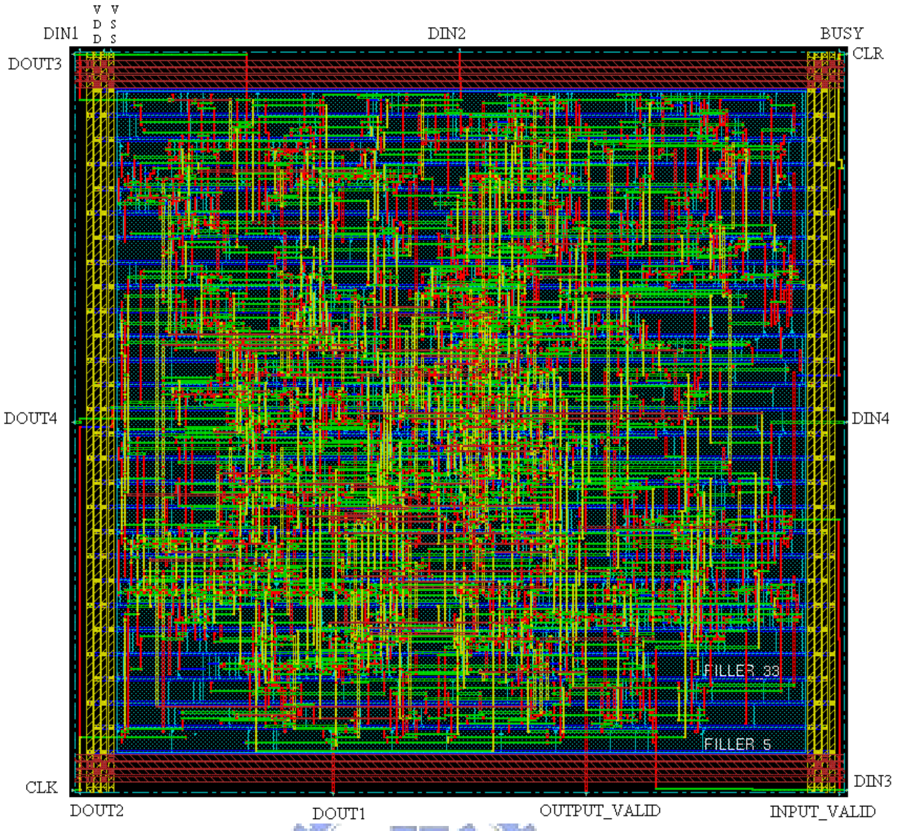


Figure 5.8 Layout of interleaver using 2-D SRA for 16-QAM WiMAX.

Chapter 6

Conclusion and Future Work

This thesis has addressed the DFC design using a new register allocation scheme which takes into account the power consumption and area cost. The proposed 1-D SRA algorithm results in the lowest number of transitions and power consumption with slightly increment of area cost. On average, the number of register transitions can be improved by 27.3%, 37.5%, 55.3%, and 50% compared with the SSRA approach among the 3x3, 4x4, 16x16 transposer, and IIR filter. On the other hand, for 16-bit 4x4 and 16x16 transposer, the power consumption can be alleviated by 45.3% and 50.2%, respectively, compared with the SSRA design.

On the other hand, this thesis has addressed the 2-D DFC design using a new register allocation scheme which takes into account the power consumption and area cost. The proposed 2-D SRA algorithm results in the lowest number of transitions and power consumption with slightly increment of area cost. On average, the number of register transitions can be improved by 12.5%, 16.7%, and 29.2% compared with the SSRA approach among the 1-D DWT, 4x4 and 16x16 par-transposer. On the other hand, for 16-bit 1-D DWT, zigzag scanner and 4x4 par-transposer, the power consumption can be alleviated by 5.3%, 13.6% and 16.1%, respectively, compared with SSRA design.

Finally, we implement the interleaver for WiMAX system and compare with the conventional memory-based interleaver between QPSK and 16-QAM, the power consumption can be improved by 52.2% and 62.1%.

The SRA scheme could be applied to other topics such as de-interleaver, reconfigurable interleaver for future research.

Bibliography

[1] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, “Low power CMOS digital design,” *IEEE J. Solid-State Circuits,* vol. 27, Apr. 1992.

[2] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, NY: Wiley, 1999, pp.157-170.

[3] K. K. Parhi, “Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation,” *IEEE Trans. Circuits Syst. II,* vol. 39, pp. 423–440, 1992.

[4] M. Majumdar and K. K. Parhi, “Design of a data format convereter using two-dimensional register allocation,” *IEEE Trans. Circuits Syst.II,* vol. 45, pp. 504–508, 1998.

[5] J. Bae, V. K. Prasanna, and H. Park, “Synthesis of a class of data format converters with specified delays,” in *Proc. IEEE Int. Conf. Applicat. Specific Array Processors*, 1994, pp. 283–294.

[6] J. Bae and V. K. Prasanna, “Synthesis of Area-Efficient and High-Throughput Rate Data Format Converters,” *IEEE Trans. VLSI systems,* vol. 6, no. 4, pp. 697-706, Dec. 1998.

[7] K. Srivatsan, C. Chakrabarti, and L. Lucke, “Low power data format converter design using semi-static allocation,” in *Proc. IEEE ICCD,* 1995, pp. 473–484.

[8] K. Srivatsan, C. Chakrabarti, L. Lucke, “A new register allocation scheme for

low power data format converters," *IEEE Trans. on Circuits and Sys.-II,* Vol. 46, pp. 1250-1253, Sep. 1999.

[9] M. Aloqeely and C. Y. R. Chen, "Sequencer based data path synthesis of regular iterative algorithms," in *Proc. 31st DAC 1994*, pp. 155–160.

[10] T. Monreal, V. Vinals, J. Gonzalez, A. Gonzalez, M. Valero, "Late allocation and early release of physical registers," *IEEE Trans. Computers,* vol. 53, pp. 1244-1259, Oct. 2004.

[11] H. R. Topcuoglu, R. Demiroz, M. Kandemir, "Solving the register allocation problem for embedded systems using a hybrid evolutionary algorithm," *IEEE Trans. Evolutionary Computation,* vol. 11, pp. 620-634, Oct. 2007.

[12] K. K. Parhi and T. Nishitani, "VLSI architectures for discrete wavelet transforms," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 116–119, June 1993.

[13] David J. Katz, Rick Gentile, *Embedded Media Processing*, MA: Newnes, 2005, pp. 347-349.

[14] Richard van Nee, Ramjee Prasad, *OFDM for Wireless Multimedia Communications*, MA: Artech House, 2000, pp. 59-60.

[15] L. Stok and J. A. G. Jess, "Foreground memory management in data path synthesis," *International Journal of Circuit Theory and Applications,* vol. 20, no. 3, pp. 235-255, 1992.

[16] F. Balasa, F. Catthoor, and H. De Man, "Dataflow-driven memory allocation for multi-dimensional signal processing systems," *in Proc. of IEEE International Conference on Computer-Aided Design,* Nov. 1994, pp. 31-34, (Santa Clara, CA).

[17] F. Franssen, F.Balasa, M. van Swaij, F. Catthoor, and H. De Man, "Modeling multi-dimensional data and control flow," *IEEE Trans. VLSI Systems,* vol. 1, pp. 319-327, Sept. 1993.

[18] A. P. Chandrakasan et al., “Optimizing power using transformations,” *IEEE Trans. Computer-Aided Design*, Jan. 1995, vol. 14, pp.12-31.

[19] K. K. Parhi, “Video data format converters using minimum number registers,” *IEEE Trans. Circuits Syst. Video Technol,*. vol. 2, pp. 255 – 267, June 1992.

[20] J. Bae and V. K. Prasanna, “A general framework for synthesis of data format converters,” *in Proc. IEEE Int. Conf. Parallel Processing*, pp. 197-200, 1994.

[21] G. K. Wallace, “The JPEG still picture compression standard,” *Commun. ACM*, vol. 34, pp. 30-44, Apr. 1991.

[22] _______, “A fast and area-efficient VLSI architecture for embedded image coding,” *in Proc. Int. Conf. Image Processing*, Oct. 1995, vol. 3, pp. 452–455.

[23] H. Park and V. K. Prasanna, “A class of optimal VLSI architectures for computing discrete Fourier transform,” in *Proc. Int. Conf. Parallel Processing*, 1992, pp. 61-68.

[24] C. Chakrabarti, M. Vishwanath, and R. M. Owens, “A survey of architectures for the discrete and continuous wavelet transforms,” *in Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, 1995, pp. 171-192.

[25] _______, “Synthesis of VLSI architectures for two-dimensional discrete wavelet transforms,” *in Proc. IEEE Int. Conf. Application Specific Array Processors*, July 1995, pp. 174-181.

[26] O. Rioul and M. Vetterli, “Wavelets and signal processing,” *IEEE Signal Processing Mag.*, pp. 14–38, Oct. 1991.

[27] *IEEE Std 802.16e and IEEE Std 802.16-2004/Cor 1-2005*, “Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems,” February 2006.

[28] Clint Smith, John Meyer, *3G Wireless with WiMAX and Wi-Fi:802.16 and*

802.11, NY: McGraw-Hill, 2004, pp. 177-195.

[29] Master thesis, Nctu, Joe Yee, “Design of Baseband Transmitter for IEEE 802.16e OFDMA”, 2008

Autobiography

王得安，彰化鹿港人，生於公元 1981 年 9 月。於公元 2003 年 6 月畢業於國立台北科技大學電子工程學系。2005 年 4 月退役後，隔年 8 月進入國立交通大學資訊科學與工程研究所就讀。其研究興趣為數位電路系統之軟硬體架構設計與嵌入式計算機處理器(ARM)之架構設計，碩士論文題目為以固定式暫存器分配方法為基礎之低功耗低面積資料格式轉換器設計。