VLSI MEMORY SHARING PROCESSOR ARRAY

A Doctorate Dissertation
Submitted to the Graduate School of
Tokyo Institute of Technology
for the degree of Doctor of Philosophy
in Electrical and Electronic Engineering

by

Dongju LI

ADVISOR : Professor Hiroaki KUNIEDA

Dept. of Electrical and Electronic Engineering
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, Japan

March 5, 1998

Acknowledgments

I would like sincerely to thank my supervisor Prof. Hiroaki Kunieda who have been offering many helpful suggestions and thoughtful advises.

I am fortunate to have received many useful comments and suggestions from many professors, specialists, friends in the world. I would like to thanks those scholars for their help. I am grateful to Associate Professor Kazuhito Ito of Saitama University, Associate Professor Tomohiko Ootsuka of Tokyo Technical School, Research Associate Tsuyoshi Isshiki, other professors and colleagues who have given me many kind of helps during my 3 year's study life in Japan.

I would like to thank Chinese Government and Japanese Government for giving me the chance to further my doctor course study in Tokyo Institute of Technology. I would also like to thank the Japanese Government for the scholarship and financial support through the graduate course.

My research work has been engaged as a project in CAD21 Research Body of Tokyo Institute of Technology. I would like to thank all the members of CAD21 Research Body of Tokyo Institute of Technology for their suggestions, cooperations and financial support.

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 VLSI Digital Signal Processing

Traditional Signal Processing takes on infinite filtering techniques, such as analog LC, RC active filter and so on. Typically, for an N input filter, the computation amount is O(N). In digital Signal Processing, time-oriented method, such as FIR IIR algorithms, translation techniques of DFT, lattice adaptive filtering algorithms are used. The computation amount is increased to O(N^2). In the earlier days, around 1960's, of digital signal processing, real-time signal processing was impossible. Since 1965, Cooley-Turkey's FFT appeared, which decreased computation amount of DFT to O(NlogN). From 70's, real-time signal processing hardware such as high-speed microprocessor, high-speed parallel multiplier etc. were developed. At the same time, some other techniques, such as, discrete analog device, such as CCD, surface audio wave (SAW) were also well developed. Since then, real-time processing became possible. At the beginning of 80's, one-chip DSP appeared, which combined microprocessor and high-speed parallel multiplier together. These chips use modified Harvard structure, where, their algorithm and control are realized by software. At that time, one chip DSP could finish 10^6 -10^7 computations per second.

Modern signal processing algorithms require very high processing speed in order to resolve more and more complex and heavy computation problems. The common point of these algorithms lay on using a large amount of linear computations, matrix-like computations, such as, matrix-matrix multiplication, reverse matrix vector decomposition, resolution group of linear equation, and so on [1], [2], [3], [4], [5]. Computation speed is required to be $O(N^3)$. i.e 10^9 for real-time processing. For example, in real-time vision processing system, such as target object searching requires the real-time rate of 512 × 512 pixel image frame per millisecond. Then, 2.5 × 10^8 pixels are processed in every second. If each pixel requires 10 operations, a speed requirement of 2500 MOPS become necessary. It is quickly apparent that special-purpose parallel processing architecture are indispensable. Therefore, high-speed, heavy amount of data stream, and real-time processing are main features of modern signal processing.

Because of flexibility, programmability, reconfigurability, and so on, digital signal processing becomes dominant method rather than other methods such as optical method. Digital signal processing appears everywhere in today's daily life. We can hear "DSP" all the time - from CD players, electronic music synthesizers, to the sound cards in PCs. The trend of digital signal processing requires the organic combination among signal processing theory, algorithm, structure and implementation. Among these, the most practical approach is the combination of parallel processing technique and VLSI technique.

Due to VLSI restriction of two dimensional plane, one of the important things for the architecture of VLSI digital signal processing is to show a high degree of concurrency for wide range of different algorithms to meet real time constraints. Another important point is that their silicon areas should be kept as small as possible to minimize manufacturing cost.

Image processing algorithms usually consist of a sequence of different algorithms. These algorithms can be classified according to their complexity of data access and structure of control flow. The model proposed in [6] distinguishes low- medium- and high-level algorithms. Low-level algorithms process one or two dimension image data with regular and data independent data flow, regular or simple data such as convolution-like algorithms or transforms. Because low-level algorithms usually operate on overlapping or non-overlapping independent segments, a high grade of parallelization can be achieved. Medium-level algorithms are more complex than low-level algorithms. Their input is image data, and output is structured symbolic data. Their instruction or data flow is mainly irregular and data dependent, such as histograming, Hough transform and variable-length-coding. High-level algorithm process symbolic data structures for control and analysis purposes. Control and data flows are irregular and not only dependent on data, but also on knowledge-based. The size of structures is usually small and irregular. Thus, many of these algorithm are inherently sequential. According to the analysis of several state-of-the-art image processing applications and the employed algorithms such as hybrid coding, street boundary recognition, pattern recognition, computer graphics, and synthetic aperture radar (SAR) image synthesis, the share of medium- and high-level algorithms within a whole application is lower than 20%. How to deal with low-level algorithm of 80% share effectively as much as possible on limited chip area requires us to find a architecture with high parallelism. Current and future applications show a high share of low and medium level image processing algorithms. To achieve a high overall performance within limited computation time, a possibly high grade of concurrent processing has to be achieved. For the implementation of a intensive parallel processing units within limited silicon area, we must introduce effective parallel VLSI architecture [7], [8].

1.1.2 VLSI Technology

VLSI Development

Recently, microelectronics technology has been developed dramatically. The areas of chips are made available to increase functional density towards wafer scale integration. From the beginning of 1990's, the density of integrated chip has increased in a speed of 2 times in every 2 years or less. In 1989, the process was promoted up from 1 micro-meter to 0.8 micro-meter. While in the beginning of 1992, Hitachi company of Japan developed a high speed microprocessor with operation speed of more than 200MOPS. It includes more than 1 million transistors in a 8 × 8 mm^2 chip area with 0.3 μm BI-CMOS process. Nowadays, 0.35 μm foundry line is very popular in the world. 0.25μm process is also practical. In 2000, if 0.15 μm and 0.1 μm is realized, it will be possible to integrate more than 50 million, 100 million transistors, and reach the upper limit of 1 billion transistors. Then, current one board of signal processing system will be integrated in one chip. System design will be changed from PCB design to VLSI/ULSI chip design.

The main problem in VLSI process technology is precision constraint. It is decided by the precision of graphics generation and etching facilities. Generally speaking, the precision of photoetching is 0.5 μm. Electron beam etching is 0.3 μm. Ion beam can realize etching with much higher precision. Now, single GaAs transistor with the high speed of 10 ps (10^{-12} second) has been realized with 0.1 μm minimum geography feature. However, integration of more transistors with the same precision will suffer from some following difficulties.

(1)Firstly, with the increasing of VLSI density, switching speed of transistor is expected to increase in inversely proportional to the size reduction of transistors. However, with the decreasing of transistor size, stray capacitors, which constraints the speed of chip, become dominant.

(2)Secondly, the number of I/O Pins is restricted to limited count. There have been extensive researches to overcome I/O pin constraints of LSIs. Since in common technology I/O pads are placed in a line on the surroundings, the number of I/O pins cannot be increased at the same rate as the number of transistors placed in square areas. I/O constraints become one of the bottle necks for huge LSI systems. Highly parallel processings can be considered in a large chip area, but they will be restricted due to their requirement of concurrent data I/Os.

(3)Thirdly, with increasing of integration density, electromagnetic induction will result in a chaos of chip.

(4)Fourthly, end-to-end propagation delay problem [9].

(5)Fifthly, Light is transfer with a speed of 4 ns per meter. If data transfer interval is 10 ns, then the reflected wave can be transfered to 2.5 meter during the period. If the metal wire is less than 20-30 cm, then, the reflected wave will disappear within the 10 ns. If the clock is increased to 500 MHz, then, the transfer time is reduced to 2 ns, within the 2 ns, the reflected wave can go to 40-60 cm. The first or second reflected wave will appear during the next active data transfer time. If the reflected wave comes at the same time as the next data come, then, data transfer error will occur. Therefore, clock speed is difficult to be raised up to 1 GHz (1ns period). Generally speaking, bus structure is used only under 400 MHz because of reflected wave problem.

Trend in VLSI technology

1. Deep Submicron Process

   VLSI process technology continuously reduces the design rules from 1.2μ, 1.0μ, 0.8μ, 0.6μ and 0.35μ. This trend is promising until 0.25μ, 0.18μ to 0.1μ process. This development also accelerates the increase of the feasible number of transistors in a chip. Today, chips including 5 million transistors have been distributed in the market. "System on chip" is proposed in various entities so as to cope with complex LSI design associated and development of IC technology. Current "system on chip" is considered as a simple extension of systems realized in PCB. Typical examples are a microprocessor with DRAM memory or multi-processor systems using plural microprocessor cores. Actually, the current systems to some extent will be realized in one chip in the next 10 years. There are much room to be considered a new architecture suitable for large area for system implementation.

2. Vertical Integration Technology

   VLSI will reach the limit of scaling feature in the near future. Advances beyond that might depend on the vertical integration technology. The three dimension integration is expected to provide several advantages, such as, (1) parallel processing, (2) high-speed operation, (3) high packaging density and (4) multi-functional operation [10]. With the increase integration density in two dimension technology, the interconnection delay both within the chip and between deferent chips become dominant. This problem is able to be resolved by using 3D technology because the signal exchange between upper and lower active circuit layers is available through via holes. Moreover, in a 3D IC, each layer or a set of several active layers can have its own function. Different types of technologies, bipolar, VMOS, CMOS maybe assigned to each active layer. It is possible to fabricate a video sensor on the top layer, then an A/D convert, and ALU, a memory, and a CPU in the lower layers to realize an intelligent image processor in a multi-layered 3-D structure.

3. Wafer Scale Integration

   Another tendency is WSI(wafer scale integration) technology. Researches on WSI started by the beginning of 1980's. It is used to integrate whole system with wafer size. In this way, not only integration density but also circuit function can be increased rapidly [11], [12]. It is prospected that WSI containing 10^8 transistors will be realized around year 2000.

VLSI Role in Information Era

With (1) the emergence of new VLSI technology, such as multi-layered three dimension IC, wafer scale IC, hybrid process IC, (2) with modern design facilities, such as workstation, (3) with CAD tools, such as, Synopsys, Cadence, Mentor, Compass, Avent, etc. (4) with codesign methodology by hardware and software advances in computer technology, VLSI technology is just leading the information era we are involved in.

1. Supercomputer

   The pace of research and development in the supercomputer technology continues to accelerate. In particular, array processors, such as the systolic and wavefront array processors have covered an extremely broad range of applications in signal and image processing. Therefore, research and development on these array processors will play a key role in the trend of future supercomputer technology. Furthermore, hybrid VLSI/optical array processors represents a new dimension for future light speed supercomputing technology. Therefore, in some important area, VLSI/optical hybrid array processor have the potential to become the next generation technology, surpassing electronic computers.

2. ASIC systems

   The need for ASIC systems is growing sharply. The speed of CMOS LSI becomes increased until CMOS LSI works at 1 GHz and more. At the same time, demands for high speed processings become larger. Multimedia communications concerning not only video signal transmission but 3D graphic signal transmission are under developments. Those problems can be solved only by dedicated systems or ASIC LSIs. ASIC systems will cover applications of such a high speed requirement.

1.1.3 Architecture for Parallel Processing

With regards to processor architecture, there are many classification. According to different concurrency stage of a program, processors are classified as shown in the Fig.1.1. Pipeline processor can realize micro-operation level parallel processing. Vector processor, superscalar, VLIW processor, systolic array processor can realize instruction level parallel processing. While, shared memory processor and array processors can realize task level processing.

Usually, parallel VLSI processing is realized by using pipeline, array, and multiprocessor architectures. Almost all of current computers have pipeline processing

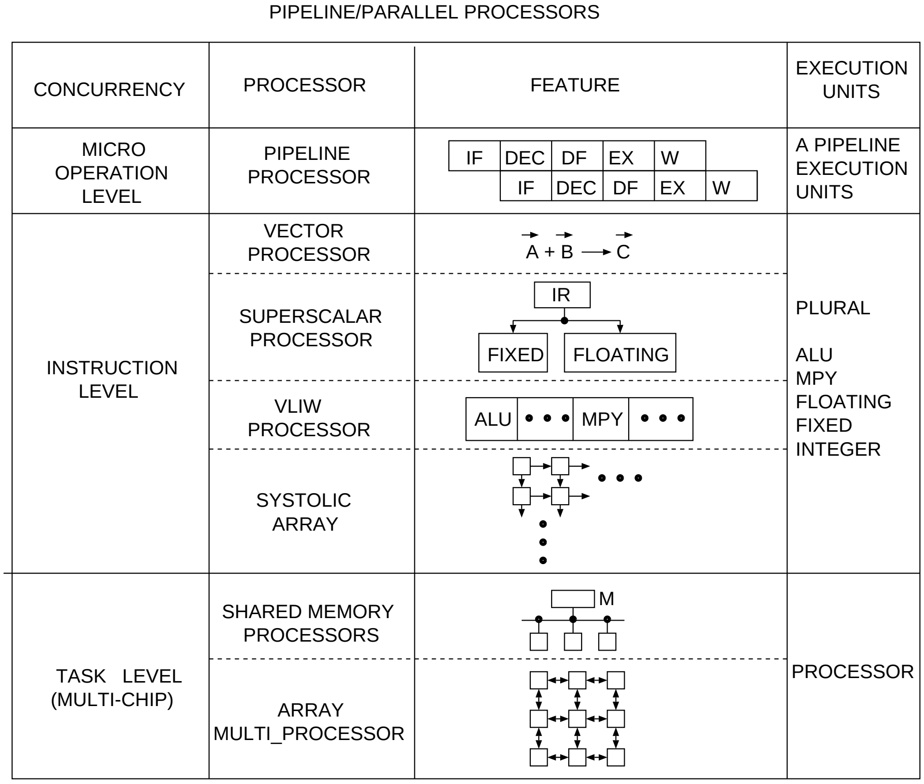


Fig. 1.1. Classification for Processor Architecture

mechanism [13], [14]. Pipeline principle can be used not only in operation level but also in higher processing level, such as instruction level. The purpose of pipeline processing is to perform overlapped computations to exploit temporal parallelism. Depending on the instruction or data types, pipeline processors can also be classified as scalar pipelines and vector pipelines. A scalar pipeline processes a sequence of scalar operands under the control of a DO loop. Vector pipelines are specially designed to handle vector instructions over vector operands. Computers having vector instructions are often called vector processors.

A synchronous array of parallel processors is called an array processor, which consists of multiple processing elements(PEs) under the supervision of one control unit (CU). An array processor can handle single instruction and multiple data (SIMD) stream.

As we know, Intel 386, Intel 486 are pipeline processors. They run in multiple pipelines fashion by introducing MPU structure to realize parallel processing in operation level. Intel 586 (Pentium) is an instruction level parallel superscalar processor. It realizes instruction parallel processing by introducing two instruction pipelines and by separating caches into instruction cache and data cache. However, Pentium is still limited to time-concurrency parallel processing architecture. It can not deal with space-concurrency job because its architecture in operation level remains is still hardware sharing for the two instruction pipelines. Therefore, its speed-up is achieved mainly by increasing instruction pipelines stages with superscalar instruction structure, and some techniques such as high-speed cache, multiple pipelines on operation level such its floating point multiply unit, write back, branch prediction, etc. Based on the same motivation, RISC, VLIW, etc have also been achieving the high speed of processings.

However, in DSP application which requires massively parallel data processing, pipeline architecture and instruction improvement are not effective because of physical restriction. On the other hand, array processing is the most active research topics in current DSP application.

Almost all of DSP adopt array processing technique. TMS320C80 is a shared-memory multiprocessor with a combined architecture of ADSP and RISC. Inside, the processors act as MIMD processors. It has a processing power to perform over two billion operations per second. It is still the same as other supercomputers in a sense that they deal with instructions by fetching, decoding, then executing inside the same pipeline. Therefore, they increase their speed by increasing the number of pipelines and complicated instruction structure and management, but not processing mechanism/structure.

1.2 VLSI Microarchitecture

1.2.1 Real time Parallel Processing

In modern signal/image processing field, many applications require us to realize real time parallel processing at very high speed. All of these use all kinds of parallel techniques, such as, pipeline processing in multi-level, time-concurrency and space-concurrency, and so on. Parallel processing can be introduced in various levels such as logic level, micro operation level, instruction level and processor level. High effective system may be one which mobilizes all levels of parallel processing. However, cost effective design is the most important under area constraints. From viewpoint of architecture, general-purpose supercomputers are often not very suitable for these real-time signal/image processing. As mentioned earlier, array processors contributes to these computation-bound applications because they provide multiple PEs with successive executions and data broadcasting synchronously.

All of VLSI speed constraints indicate that it is difficult to increase processing speed by increasing clock speed. In order to realize real time digital signal/image processing, the most important approach is developing array processor architecture with data parallel processing mechanism. With the development of VLSI, it is possible to implement array processor in one chip.

VLSI requires same or similar function of each processor element, regular allocation of PE, local communication between PEs, and so on. The promising technique may be a combination between digital signal processing algorithm and VLSI design. Around the 1970's and 1980's, many array processor architectures were presented. The most famous one are systolic array proposed by H.T.Kung of Garnegie-Mellon Uni. and wavefront array. Their modularity, regularity of connection networks, and local interconnection, and so on lead them a readiness to VLSI realization. Moreover, they have higher processing efficiency by employing high parallel pipeline technique. Its appearance brings high speed real time processing in a new era.

1.2.2 VLSI Requirements

Highly parallel computing structures have structural properties that are suitable for VLSI implementation. A high performance parallel processor often balance following techniques:

time parallel and space parallel,
static/dynamic scheduling and/or data dependency,
local communication and/or shared memory,
ASIC and/or general purpose,
hierarchy or flat structure,
software or hardware,
multiprocessor,
multiple data streams and memory bandwidth,
globle communication or data broadcast,
pin constraint.

Key attributes of VLSI computing structures are described below [15].

Area Restriction VLSI chip area is directly corresponding to hardware costs. Compact or area small design contributes not only design cost, but the better performance. Since communications between different components in a VLSI chip become more and more time expensive, smaller design leads to high speed circuits.

Simplicity and regularity In VLSI design, if a structure can truly be decomposed into a few types of building blocks which are used repetitively with simple interfaces, great savings on design cost can be achieved. Array processor is just the most appropriate architecture. It is characterized with many identical processing elements. In order to cope with the complexity of various applications, simply and regular VLSI architecture and its design methodology are essential.

Design Complexity Design itself become very complicated. Though we try to use CAD systems for design of current VLSIs, the current design systems may not follow the development of LSI system size. The design style must be coped with such a design of huge systems. The floor planning must be roughly decided and performance evaluation must be expected even in the system design level. Otherwise, small area and high performance can not be achieved by the logic and physical designs.

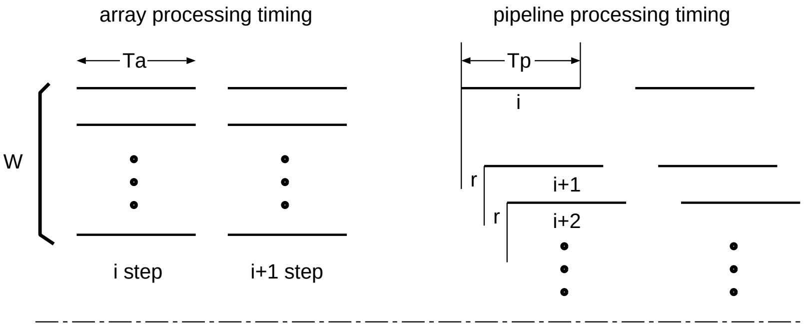
Computation Speed Due to the physical constraints of component speed, any major improvement in computation speed must come from the concurrent use of multi processing elements. With VLSI implementation consideration, (the routing cost dominate the power, time, and area required) an architecture with a large number of PE working simultaneously but in the meantime with only simple, regular communication and control mechanism is required.

Local Communications and Data Broadcastings Local communication in a chip can lead to high speed operations of VLSI architectures, which are restricted by the wire delays. Local communications between modules in a chip will also contribute small routing areas and thus simpler physical design. However, they may sometimes degrade the performance efficiency to balance the processing speed of PEs with data transmission speed. They also spend the temporal data storages to distribute data to many modules or PEs. Memory sharing with plural PEs has so advantages as to reduce the required memory capacity and to transfer data at highest rate. Of course, its scheme has disadvantages in the aspects of bus neck problems. The ideal system might be one who use properly both local communications and data broadcasting.

I/O Bottle-neck VLSI processing structure are suitable for implementing computed-bound algorithms rather than I/O-bound computations. In another words, the I/O-bound problems are not suitable for VLSI because VLSI packaging constraint. While in the design procedure of VLSI architecture, how to overcome I/O bottle-neck problem should be seriously considered [16].

1.2.3 Array Processing

Comparison between Pipeline Processor and Array Processor Currently, V-LSI parallel processing architectures can be classified into two kinds of processing principle, pipeline processing and array processing. Pipeline processing performs overlapped computations to realize temporal parallelism, while array processing uses multiple synchronized arithmetic logic units to achieve spatial parallelism. As shown in Fig.1.2, it shows a comparison between two principles with regards to the parallel processing efficiency. In order to perform N operations, array processing needs $T_A \times \frac{(N)}{W}$ processing time, where the T_A is instruction cycle for one operation, and W is the number of PEs. In pipeline processing case, operation time is $T_p + \tau \times (N-1)$, where T_p is instruction cycle for one operation, and the τ is the processing time of each pipeline. If $\frac{T_A}{W} < \tau$,



The number of PEs: W
The number of operation: N
Operation cycle:Ta
then
Total time = Ta (N/W)

Operation cycle/pipeline: Tp
Operation cycle/stage of a pipeline: r
then
Total time = Tp+r(N-1)

	Operation time/step	
	N=1	N=∞
Array processing	Ta	Ta/W
Pipeline processing	Tp	r

Fig. 1.2. Speed Limit Comparison between Array Processor and Pipeline Processor

and $T_A < T_p$, then, array processing is better than pipeline processing. It will be noticed that array processing efficiency is $N \times W \times (\frac{N}{W})$, while pipeline processing efficiency is $N/(\frac{T_p}{\tau+N-1})$. Because the number of PEs can be increased without physical limit theoretically, while in the other hand, decreasing of τ will be physically restricted to some extent. It is obvious that array processing is very effective particularly for very heavy data processing applications, such as, DSP, image processing, and computer vision. According to H.T.Kung in 1978, the relation between computation algorithm and processor architecture can be classified into two types. One is called computation-bound (constrained by computation), and the other one is called I/O-bound (constrained by I/O ports). It is undoubtable that array processor is the most suitable architecture for resolving computation-bound problem.

Array Processors/Systolic Array Nowadays, in many applications with computationally intensive operations, there are many kind of array processor architecture, such as Micro-Grain Array Processor (MGAP) [17], [18], [19], [20], asynchronous array processor for data-dependent system [21], mesh-connected multiprocessor [22], [23], [24], array processor with parallel I/O [25], Multi-rate array [26], coarse-grained architecture [27], [28], etc [29]. However, the most typical architecture of synchronous array processor is systolic array, it is widely used in various aspects including matrix-multiplication-like algorithm, convolution-like algorithm, various transformation such as DFT, FFT, DCT, DWT, various digital fiterings such as FIR, IIR, median filter, etc, [30], [31], [32], [33], [34], [35], [36], [37], [38], [39]. The systolic architectural concept was developed by Kung and associates at Carnegie-Mellon University, and many versions of systolic processors are being designed by universities and industrial organizations . Systolic array consists of a set of interconnected simple processing elements (PEs). Information in a systolic array flows between PEs in a pipelined fashion [40]. Communication with the outside world occurs only at the boundary PEs. The principle of systolic array is described as shown in Fig.1.3, The function of the memory is analogus to that of the heart. It pulses data through the array of PEs. By this approach, once a data is brought out from the memory, it can be used efficiently at each PE when it passes. But the high processing speed is achieved by its advantages with parallelism, modularity, regularity, local and regular communication are also its typical advantages. These are very important features for VLSI architecture design.

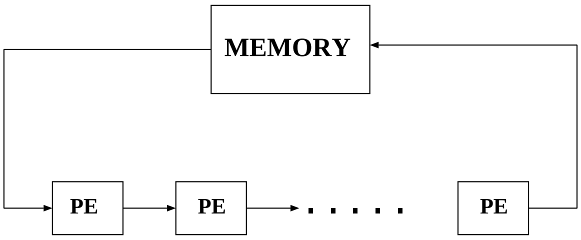
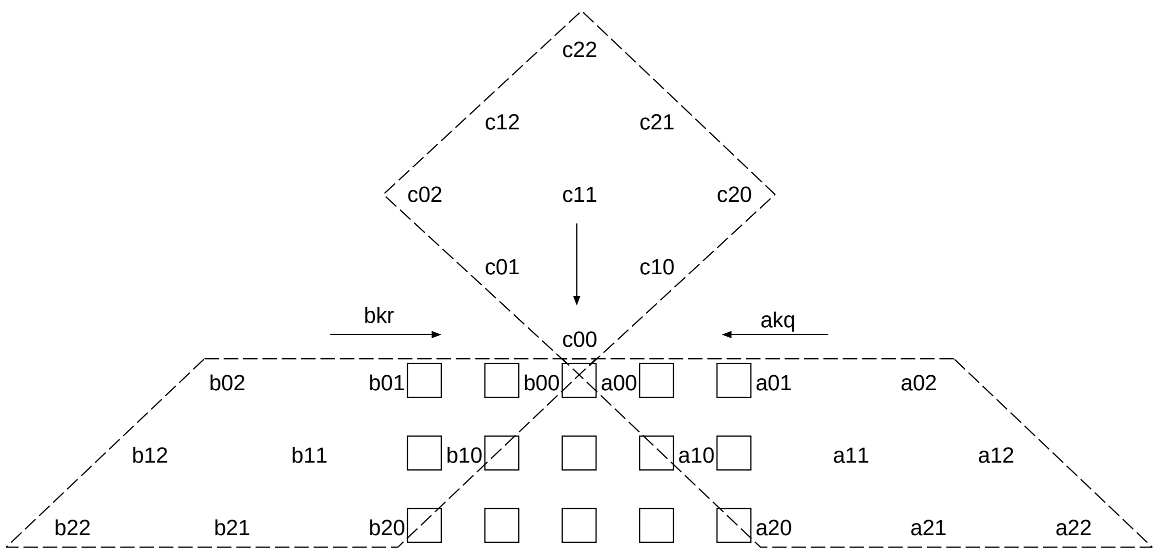


Fig. 1.3. Systolic Array Principle

Problem in Systolic Array As an typical array processor architecture, systolic arrays and their design have been an active research area [41]. It achieves a high parallel processing with multiple data streams, small data control circuits and local communication between adjacent processors. Because of these advantages, it has been broadly used in image processing, computer vision, nuclear physics, structure analysis, speech, radar, astronomical, medical signal processing applications, etc. Array processor is often used as coprocessor, enhancing the computing capability of the host machine. Typically, a complete systolic system consists of a host computer, an array of processors and an interface. The host computer runs a main program and requests the systolic array for specialized tasks on specific data. The array of processors performs intensive computation on these data and delivers results.

In the application-oriented fields, researches on systolic array are very important. However, in a practical VLSI design, systolic array sometimes suffers from a low parallel efficiency and large amount of overlapping data storages, I/O bottleneck and large processor element amount. Some of them are due to excessive local communication requirements, Special care must also be taken for data input. Some systolic array look working well with special data input format. For practical implementation, such data input schemes may require additional hardware for data broadcasting. If the number of I/O ports is limited, the systolic array does not work efficiently because of I/O bottleneck. Though I/O and data conversion are usually treated as separate problems from the design of systolic array, we should consider the total system including both array processing and I/O problems [42], [43], [44], [45].



EXAMPLE OF SYSTOLIC ARRAY
BIT LEVEL ARRAYS FOR INNER PRODUCT COMPUTATION

Fig. 1.4. Classification for Processor Architecture

There have been a lot of researches in developing design methods to map uniform dependence algorithms onto systolic processor arrays [46], [47], [48], [49], [50]. The problem of mapping an algorithm to a processor array is characterized by a linear mapping matrix which consists of a schedule vector and an allocation matrix. Due to the difficulty on searching for iterative optimal scheduling vector[51], many researchers are rather trying to improve mapping methods, such as the General Parameter Method (GPM) [52], Integral Matrix Theory [53], etc. In order to increase the performances of systolic array architecture, such as, total time of computation, number of processors, electronical complexity, and so on, some authors presented some optimization methods. However, these methods often leads to problems of non-linear optimization quite hard to process [54], [55].

The development of simple and automatic design for matrix mapping is quite important for the development of programmable or reconfigurable arrays, even though such a method is not truly optimal.

1.3 MSPA Solution

In this thesis, we use a linear transformation of index set to deal with the mapping problem under the assumption that the number of processor elements is less than the number of uniform algorithm size. This assumption is useless to realize systolic arrays because mapping result determines the number of processor elements. Any specified number of processor element (PE) is accepted in our MSPA design. This reduction in the PE number greatly contributes to the advantages of MSPA architecture.

1. *Hardware reduction* The hardware area for executions is reduced independently to the algorithm size. Of course, the memory capacity in MSPA depends on the algorithm size.
2. *Design simplification* Straight forward scheduling and resource allocation from the data flow graph is obtained from derived theorems based on a reduced PE number. Systolic array design usually requires a complicated iterative procedure to get feasible scheduling and resource allocation mapping with complete local communication.
3. *Memory storage minimization*
4. *Small number of I/O ports*
5. *High data transfer rate*
6. *High operation throughput*

Systolic arrays stores data in each PE. In addition to that local data storages, multiple data flowing into the array require additional memory storage or some form of data conversion outside of the array. Usually, those schemes are discussed in separated topics. On the contrary, our design method includes those data conversion under restricted I/O number and memory management. Minimization of required memory storage is achieved to some extent in our design.

Our design methodology is roughly described as a procedure that: first, fix the number of processor elements based on the hardware limitation, next, find optimal operation scheduling vector and resource allocation vector, then, set memory management unit parameters, and ending up with configurating address generation units corresponding to I/O port limit [56], [57].

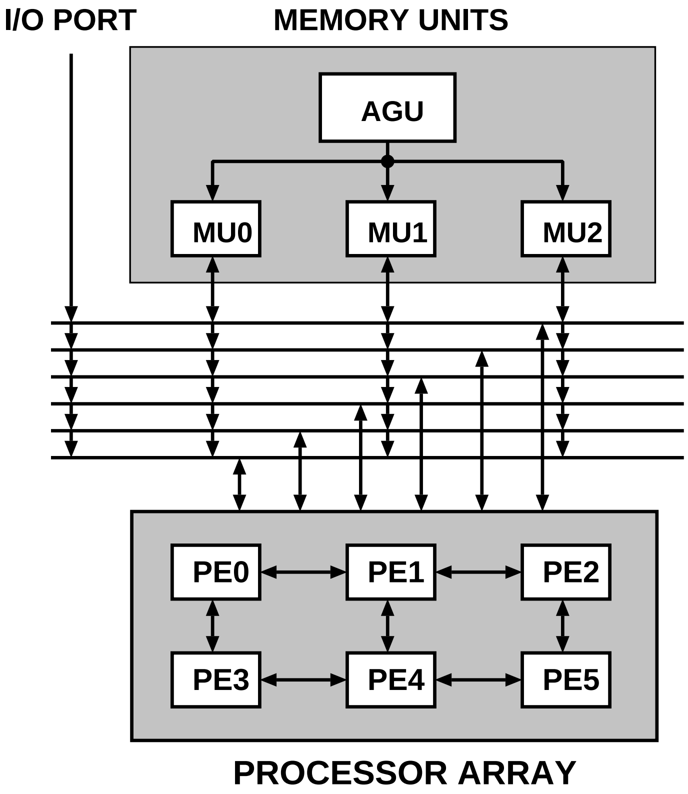


Fig. 1.5. Memory Sharing Processor Array (MSPA) Architecture

The array named Memory Sharing Processor Array (MSPA), as shown in Fig.1.5, consists of a systolic-like processor array with several memory units and their address generation hardware units to minimize the data storage [58]. Because the MSPA architecture has been taken account of the VLSI design criteria with the sound balance between I/O port and chip-in computation, it achieves a higher cost-efficiency than systolic array by restricting the number of processor elements. If the number of concurrent data input ports is limited in a practical situation, the systolic array does not work efficiently.

The initial goal of developing MSPA architecture is to cope with heavy data processing problems with less hardware amount, which result in an obvious quantitive distinction from systolic array on several points. MSPA uses much less processor elements than the algorithm size, while systolic array uses more processor elements than the algorithm size. This assumption also leads to optimal operation schedule and the resource allocation method in the closed form. Another difference is that MSPA possesses the data access schemes of the common bus architecture, while the systolic array insists on the local communication between PEs. However, MSPA limits

its usage only when those non-local communications do not slow down the speed in VLSI realization. By introducing the shared memory and its address generation logic, we realize the I/O data format conversion from data-serial input scheme to parallel distributing scheme effectively based on our operation scheduling and resource allocation theories.

Though we also use the linear transformation method to deal with operation scheduling problem, our points are either to find a optimal schedule for a given area-oriented (or the number of processor elements (PEs)) specification, or to get the smallest number of PEs for a given slower time/speed specification. On the contrary, the design of systolic array is absolutely an algorithm mapping separated from the internal data stream conversion and control. It results in that the number of PEs is always more than or equals to algorithm size. For the reasons of different goals and different architectures, the design methodologies are quite different between systolic array and MSPA.

This thesis consists of 7 chapters. The backgrounds of VLSI technology and processor architecture are well introduced in the first chapter. After that we present our solution to processor architecture, which is called memory sharing processing array (MSPA). In the second chapter, some important concepts about MSPA are introduced. The detail design methodology is explained in the third chapter. In order to deal with the applications with serial input scheme, we proposed data format conversion theory in the fourth chapter. In the fifth chapter, we give some examples to illustrate how to apply the proposed MSPA design methodology and DFC theory to VLSI implementation. In the sixth chapter, we propose a developed MSPA architecture for window operation in image processing, which is called Window-MSPA. Its design methodology is also described in detail in this section. Some examples are given to show its application in DSP. At last, a conclusion is given in the seventh chapter with some prospection in the future work. The structure of the thesis is shown in Fig.1.6.

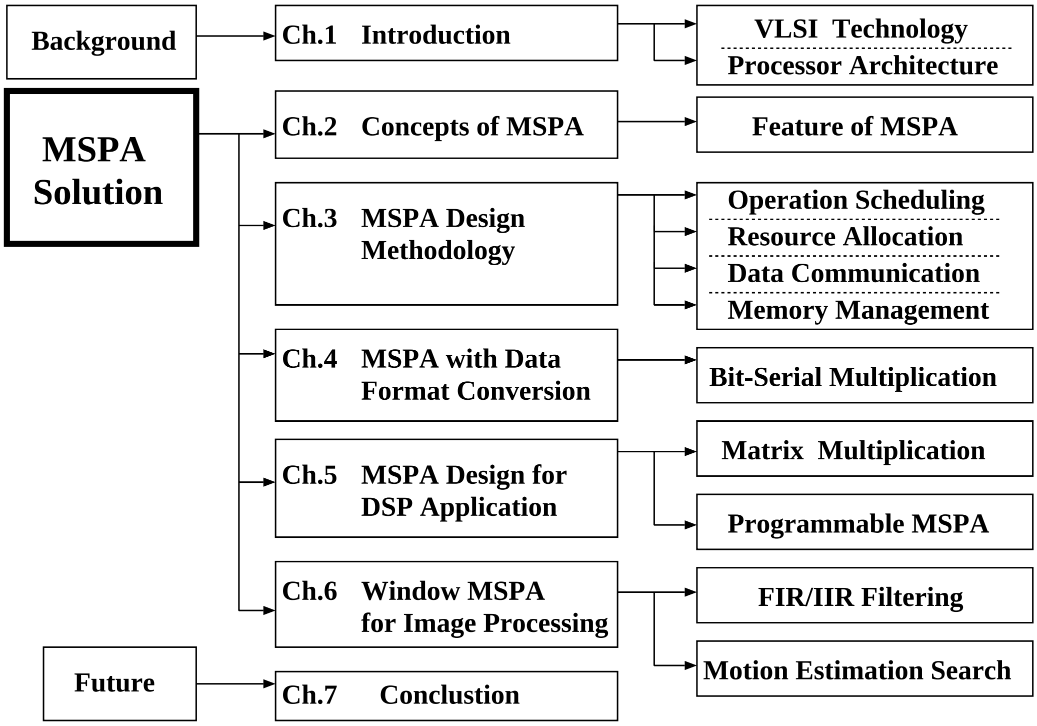


Fig. 1.6. Content Table

Chapter 2

Basic Concepts of MSPA Theory

MSPA architecture and its design methodology looks similar to the conventional systolic arrays [59], but tries to overcome the overlapping data storages, idle processing time and the I/O bottleneck problems, which mostly degrade the performance of the systolic array. It has the practical advantages over the systolic array in the view of area-efficiency, high throughput and practical input schemes.

In this chapter, a Matrix Vector multiplication example will be used to explain the general design principle. It will be found that our design methodology based on MSPA architecture shows a programmability and reconfigurability because of the systematic design procedure.

2.1 MSPA Architecture

Fig.2.1 shows a schematic architecture of MSPA, which is used to execute matrix-vector multiplication such as [A(i,j)][x(j)]. It consists of a processor array with direct links and buses associated with memory units (MUs) and hardware address generation unit (AGU). It accepts a limited number of regular input data through practical Input Ports. Input data are loaded from input port to MUs in serial data scheme. The MUs can provide a multiple data stream into the processor array under the control of AGU. These data flow can be seen as a data format conversion function. Preload pipeline and execution pipeline are performed in parallel. There are two kinds of inputs, *compile-time input* which needs a preload and a *run-time input* which needs no preload. The input $x(i,j)$ in the Fig.2.1 is a *run-time* input. In the case of *run-time input*, we introduce registers as temporary memory to deal with data storage problem. A special design methodology to deal with data format conversion problem has been worked out for *run-time input.* By that theory, the data storage space can be minimized to six registers for input $x(i,j)$.

Another *compile-time input* $a(i,j)$ has to be preloaded to the memory in serial scheme from input port. The memory units and AGU play the role of changing the input data format scheme to parallel scheme in order to meet the parallel processing

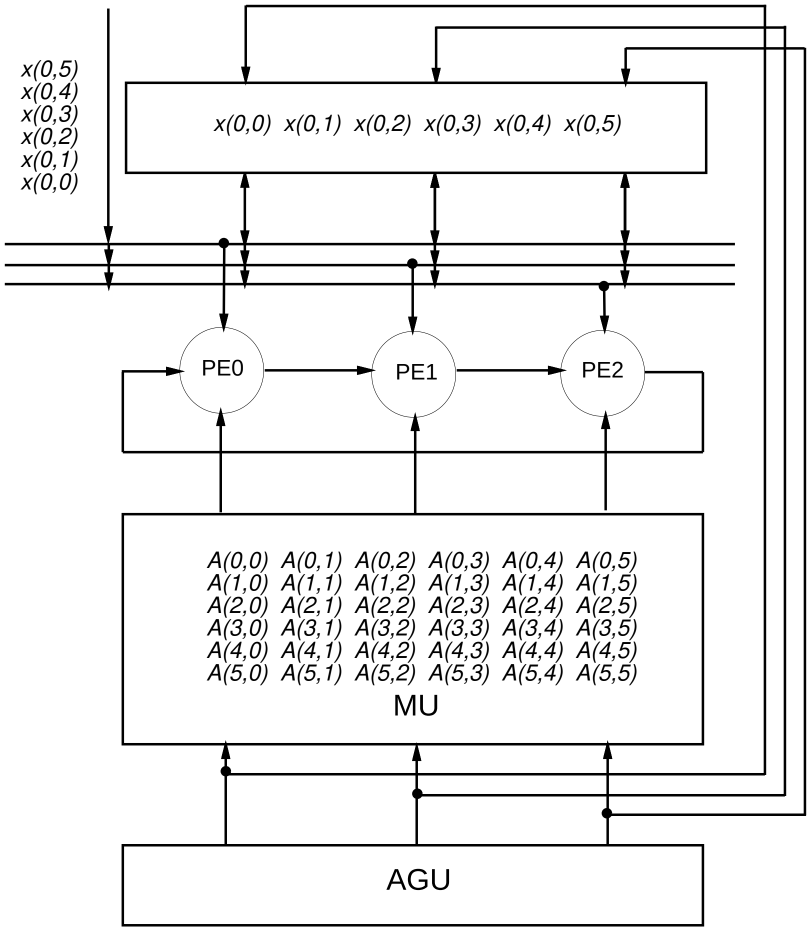


Fig. 2.1. MSPA Matrix-Vector Multiplier

requirement of PEs. At the same time, both of the temporary memory and the MU provide three data streams to PEs, respectively. The data are shared by all the PEs. Data transfers between processors are accomplished by mainly local direct communications. Those processor cells operate synchronously at discrete time steps. For simplicity, each processor cell is assumed to complete its assigned operation within 1 step.

2.2 Design Methodology

There has been a lot of research on developing design methods to map uniform dependence algorithms to processor arrays. Most of these methods are based on Data Dependency Method [46], [47]. By this method, the problem of mapping an algorithm to a processor array is characterized by a linear mapping matrix $S = \begin{bmatrix} T \\ O \end{bmatrix}$

, where T is the scheduling vector and O is the resource allocation matrix. Due to the difficulty on searching for optimal elements of T, many researchers are taking effort on improving the iterative mapping method, such as General Parameter Method [52], Algorithm Partition Method [60], [61]. Because almost all of these methods are based on systolic array architecture, which needs more processor elements than the algorithm dimension, it's difficult to realize it in practical applications due to heavy hardware burden, including the I/O port bottleneck problem. In this thesis, we deal with the mapping problem by a linear transformation of the index set with an assumption that the number of processor elements is less than the number of uniform algorithm dimension. Our design methodology is outlined as the following procedure: Firstly, choose the number of processor elements based on the hardware limitation. Secondly, find the optimal operation scheduling vector and the resource allocation vector. They are explicitly given in the theorems. Thirdly, set parameters about memory management units. At last, configurate an address generation unit according to I/O port limitation. Fig.2.2 shows the design procedure.

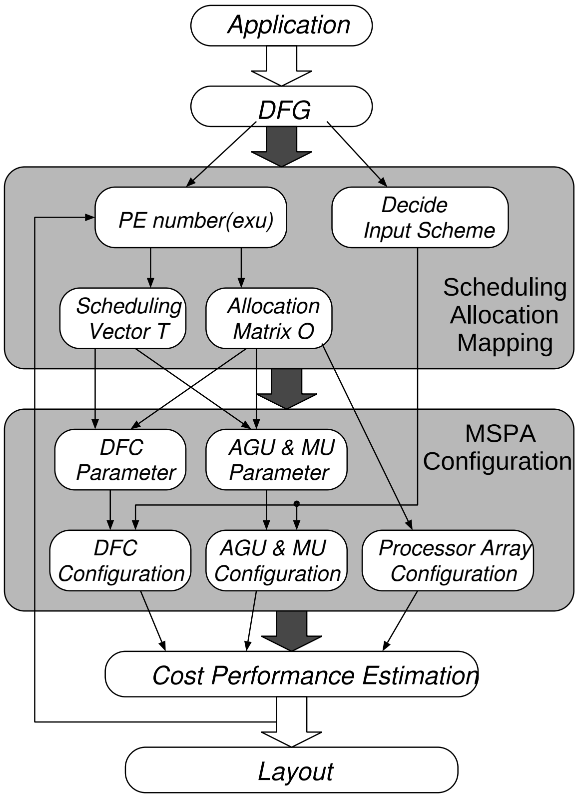


Fig. 2.2. MSPA Architecture Design Procedure

The mapping vectors T and O satisfying the local communication condition depend on the dependence vectors of the given algorithms so that it is very difficult to get the optimal T and O theoretically in systolic array. In MSPA, we can easily derive the optimal T and O, because MSPA relaxes the local communication conditions. Therefore, we take a different approach from systolic one to this problem. Firstly, we select the optimal T and O under the restriction of the number of PEs and the resource conflict free condition. Since we do not take account of the local communication condition in this stage, the optimal T and O are derived in the simple closed form. Then, we select the data transfer schemes of each data dependence vector under the derived optimal T and O.

2.2.1 Algorithm Description

We consider algorithms described by the uniform recurrence relations as

```
for $i_1 = 0$ to $U-1$ step 1
for $i_2 = 0$ to $U-1$ step 1
   ...
for $i_l = 0$ to $U-1$ step 1
   ...
      $v_1(p) = f_1(v_1(p-d_{11}), \cdots, v_m(p-d_{l1}), \cdots, v_m(p-d_{m1}))$
      $v_2(p) = f_2(v_1(p-d_{12}), \cdots, v_m(p-d_{l2}), \cdots, v_m(p-d_{m2}))$
      ...
      $v_l(p) = f_l(v_1(p-d_{1l}), \cdots, v_m(p-d_{kl}), \cdots, v_m(p-d_{ml}))$
      ...
```

(2.1)

where U represents the number of iterations in each loop. $v_l(p)$ denotes the value at the index point p of the l-th variable. $p = (i_1, \cdots, i_n)^T$ is an index point vector of the variables in the index space. d_{kl} is a column vector in the same n dimensional (nD) index space to show the data dependency of the variable $v_k(p)$ against the variable $v_l(p)$. The function $f_k()$ represents the operation to produce the variable $v_k(p)$.

2.2.2 Execution Ordering

We consider the time coordinate in the same manner as the systolic array by the linear transformation with a row vector T for each operation set f_k. For an algorithm description of

$$v_k(p) = f_k(v_1(p-d_{1k}), \cdots, v_m(p-d_{mk})), \tag{2.2}$$

the operation $f_k(p)$ invokes at

$$Tp = t_1 i_1 + \cdots + t_n i_n \tag{2.3}$$

and produces the variable $v_k(p)$.

The mapping with T can be selected so that it satisfies the valid execution ordering for any data dependence. For a data dependence d_{kl},

$$Td_{kl} > 0. \tag{2.4}$$

must hold [62], [63] . This condition is called valid execution ordering condition. It indicates that the variable $v_l(p)$ is generated by using the variable $v_k(p - d_{kl})$ whose generation time is earlier than that of $v_l(p)$. This execution ordering cannot be applied to the input variables, because it does not require any operation from execution units. However, we can decide their input times either from memory or from I/O port, which can be expressed in the same manner as Tp.

Among the mappings with the valid execution ordering, we select the optimal mapping which requires the minimum number of concurrent data storages in order to generate processor arrays with minimum hardware area in VLSI.

2.2.3 Resource Allocation

Suppose that the position of each processing cell in the array is described by one dimensional coordinates. This is because the resource allocation on linear array is easily extended to 2 dimensional array case. We introduce the linear mapping with the modular operations against the given algorithm as

$$M_O = \begin{pmatrix} m \end{pmatrix} \tag{2.5}$$

$$Op \bmod M_O = \begin{pmatrix} (o_{11}i_1 + \cdots + o_{1n}i_n) \bmod m \end{pmatrix}. \tag{2.6}$$

This method increases the resource utilization rate while it preserves the regularity of the algorithm. In fact, the data transfer schemes derived by this mapping remain in regular fashion. Of course, the conventional linear mapping for the systolic array is included as this special cases. The mapping with O, M_O can be selected in such a way that any two operations of the same execution times are not mapped onto the same operation cell. This can be described as follows.

For any two index points p and p' which invoke at the same time ($Tp = Tp'$), the mapping with O and M_O must satisfy the inequivalence of

$$Op \not\equiv Op' \bmod M_O \tag{2.7}$$

This necessary condition is called resource conflict free allocation.

2.2.4 Data Communication Schemes

In order to minimize the MSPA hardware, we should realize data transfers between adjacent processing cells as much as possible. A data transfer corresponds to a data dependence. A data dependence d_{kl} indicates that the variable $v_l(p)$ is generated by using the variable $v_k(p - d_{kl})$. Therefore, the data transfer time Δt_{kl} for the data dependence d_{kl} is obtained from the time difference between the generating time of the variable $v_k(p)$ and the consuming time for the variable $v_l(p)$ as

$$\Delta t_{kl} = Tp - T(p - d_{kl}) = Td_{kl}. \tag{2.8}$$

The data transfer distance $\Delta dist_{kl}$ for the dependence d_{kl} is obtained by Manhattan distance between the processing cell positions for operations on those variables as

$$\Delta dist_{kl} = dist(Op \bmod M_O, \;\; O(p - d_{kl}) \bmod M_O) \tag{2.9}$$

where the function *dist* decides the distance between two PE allocations. The relation between time and distance for data transfer specifies the data transfer schemes associated with MSPA architectures.

For any two index points p and p' $(p = p' - d_{kl})$, the necessary condition for local communication should be described as

$$\Delta t_{kl} \geq \Delta dist_{kl}. \tag{2.10}$$

This condition indicates that data transfers corresponding to data dependencies can be realized by successive data transfers between adjacent PEs.

According to the derived optimal T and O, we select the data communication schemes corresponding to each data dependence vector. PE distance of the data transfer is given by $(Od_{kl} \bmod M)$ for the data dependence vector d_{kl}, while its time duration is given by Td_{kl}. We can classify the dependence vectors into two cases. For each case, we select the appropriate data communication schemes.

1. Local/Bus scheme: $(Od_{kl} \bmod M) \leq Td_{kl}$
   The dependence vectors in this range can satisfy the local communication condition. The local communications by the successive data transfers between adjacent PEs are available for any dependence vectors. The Bus communications are available only for the dependence vector whose communication times are long enough to access bus.

2. Direct/Bus scheme: $(Od_{kl} \bmod M) > Td_{kl}$
   The data transfers by either direct links or the bus via or not via memory units are available for the dependence vectors in this range.

Fig.2.3 shows the above two ranges of dependence vectors in an example. Almost all dependence vectors can be realized by the local communications in this example. The employment of the direct links between non-adjacent PEs makes all the dependence vectors to be realized without slowing down the execution speed.

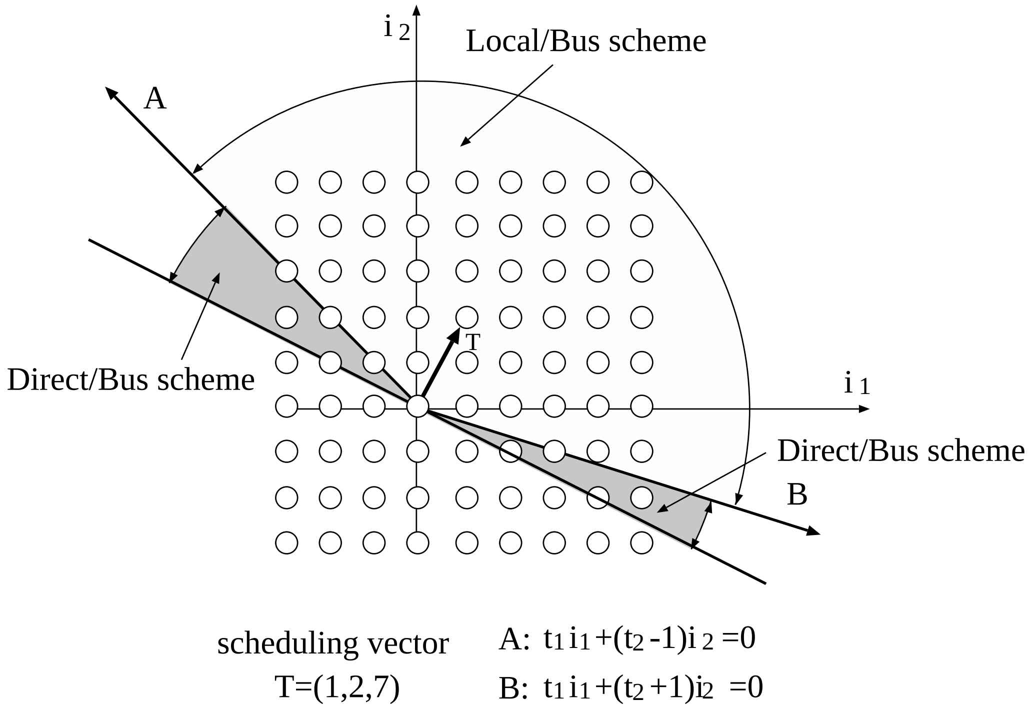


Fig. 2.3. Communication scheme

2.2.5 Memory Management and Address Generation

We prepare different memory units for each input variable. Each memory unit stores the input variable which are provided to each PE. Each memory unit is organized with the vector addresses corresponding to the index points of each variable. For example, a variable $a(i_1, i_2)$ is stored with (i_1, i_2) address.

In order to provide the plural data to processor array, each data can be sent to any PEs at any time. The data addresses and their destination PE number for *exu* PEs are input to the memory unit by the address generation units.

Address Generation Unit consists of the execution index points generator and the converter. The execution index points generator generates the invoking index points at each execution steps. The converter converts the index points into the physical addresses for each memory units. In this scheme, we can employ an unique execution index points generator for the different memory units as shown in Fig.2.4.

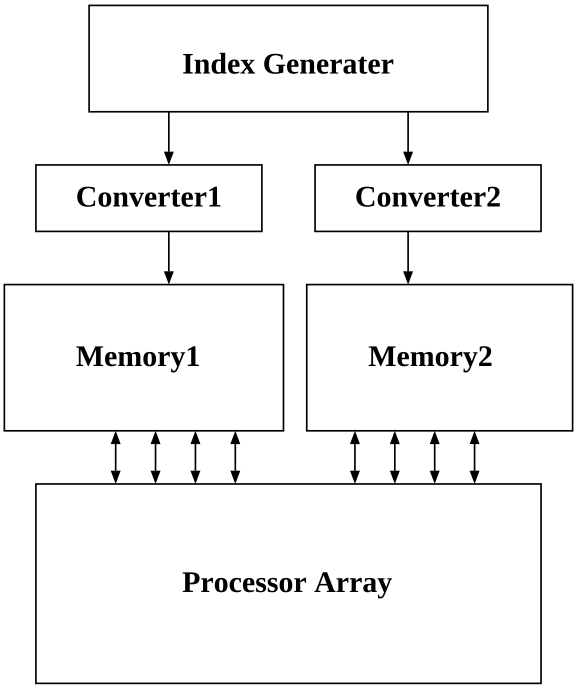


Fig. 2.4. Memory and address generation

Chapter 3

MSPA Design Methodology

3.1 Operation Scheduling Approach

3.1.1 Data Dependency

In the previous chapter, the algorithm of uniform recurrence relation is described by Eq.2.1 The following is an example of the algorithm description for the matrix-vector multiplication $[a_{i_1,i_2}][x_{i2}]$. We use this example to explain the design procedure of our MSPA architecture.

The matrix coefficient is described as $a(i_1, i_2)$ while the vector $(x(0), \cdots, x(U-1))^T$ is expressed in the same dimensional expression as $x(0, i_2)$. The i-th element of the resulting vector $(y_0, \cdots, y_{U-1})^T$ is derived as the value of $y(i_1, U-1)$ in the expression.

```
for $i_1 = 0$ to $U - 1$ step 1
for $i_2 = 0$ to $U - 1$ step 1
    $y(i_1, i_2) = y(i_1, i_2 - 1) + a(i_1, i_2) * x(0, i_2)$
```

where $y(i_1, 0) = 0$ $(0 \leq i_1 \leq U - 1)$.

Fig.3.1 shows the data flow graph expressing the data dependence relations in the index space (i_1, i_2) for the example. The partial sums propagate in vertical direction through the product-sum shaded nodes. The dependence vectors of this algorithm are written as

$$\begin{aligned} d_{ay} &= (0,0)^T \\ d_{xy} &= (i_1,0)^T \\ d_{yy} &= (0,1)^T. \end{aligned} \tag{3.1}$$

3.1.2 Constraints On Operation Scheduling

According to the previous chapter, constraints on operation schedulig is described as tha condition for valid execution ordering.

Valid execution ordering: *The mapping with T can be selected to satisfy the valid*

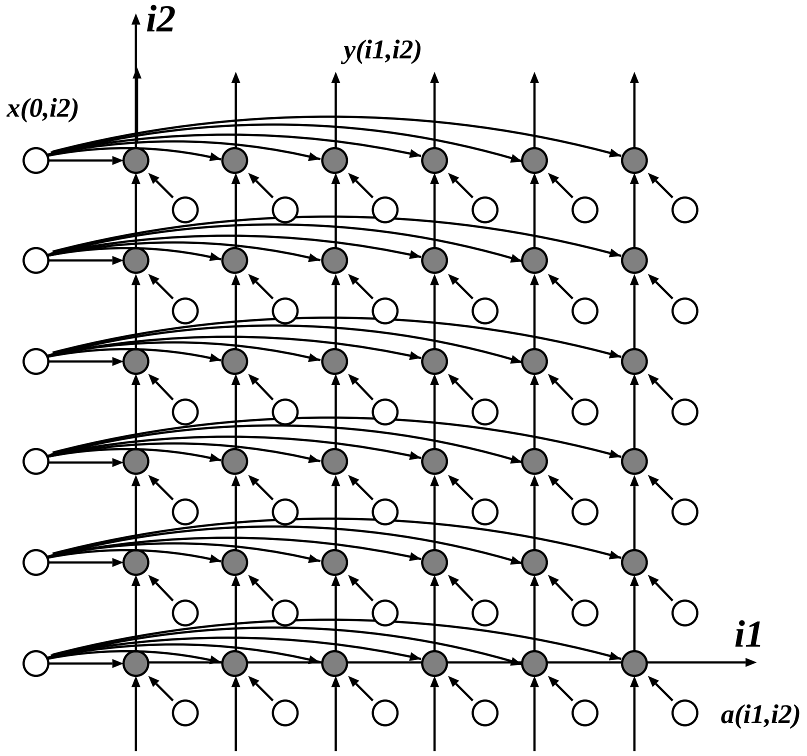


Fig. 3.1. Data Flow Graph of a Matrix-Vector Multiplication

execution ordering for any data dependence. For a data dependence d_{kl}*,*

$$Td_{kl} > 0 \tag{3.2}$$

must be held.

We select the optimal T among the valid scheduling vectors, which makes the number of the concurrent operations less than or equal to the given PE number.

Since the number of PEs is specified, we should select a scheduling vector, which invokes less number of concurrent execution nodes than the PE number exu. Concurrent execution nodes at t are indicated by the index points p lying on the "cut" line in the index space, which satisfies $Tp = t$. Under the processor number restriction, the number of index points satisfying $Tp = t$ must not exceed exu at any execution time t.

In the example here, we obtain optimal schedule vectors achieving the smallest total execution time under the constraint of 3 execution units as $T = (1, 2)$. Fig.3.2 shows the execution ordering of Fig.3.1. Each cut line corresponds to each execution time propagate in the north-east direction. This schedule achieves total execution within 15 steps.

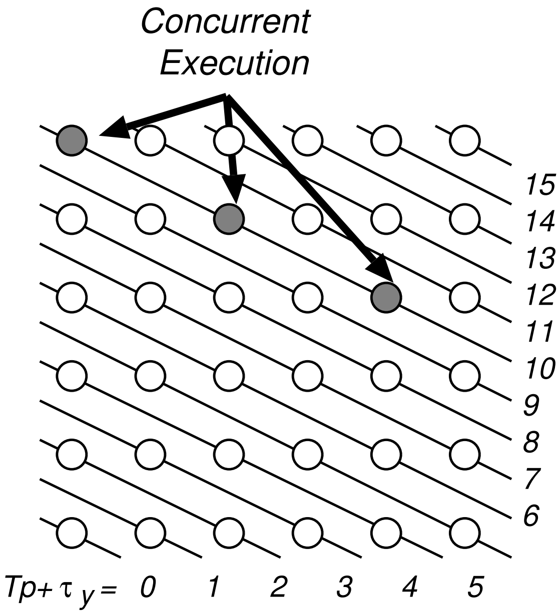


Fig. 3.2. Execution Ordering of Matrix-vector Multiplication

3.1.3 Scheduling Theory for 2D Algorithm

Firstly, we only consider the scheduling vector satisfying the PE number restriction without taking into account the valid execution ordering for the dependence vectors. We extend this consideration to 2 dimensional array case in this section.

Theorem 1 (2D Scheduling Vector under PE number Restriction) *The mapping vector on exu processing elements which achieves the minimum total execution time under the PE number restriction for the 2D uniform recurrence algorithm with the two U iteration loops of $exu < U$ is given by*

$$T = (1, \lfloor \frac{U}{exu} \rfloor) \tag{3.3}$$

where $\lfloor x \rfloor$ represents the smallest integer more than or equal to x.

Proof 1 *The cut line passing through the index point $(0, U-1)$ includes the largest number of nodes among cut lines as shown in Fig.3.3. The index points on the cut line are $(0, U-1), (t_2, U-1-t_1), \cdots, ((exu-1)\cdot t_2, U-1-(exu-1)\cdot t_1), \cdots$. Since their number should not exceed the PE number exu, the index point $(exu \cdot t_2, U-1-exu \cdot t_1)$ must be out of the index range. Under the assumption of $t_2 > t > 1$, the i_1 coordinate $exu \cdot t_2$ must be larger than the boundary of the range $U-1$ as*

$$exu \cdot t_2 > U - 1. \tag{3.4}$$

Since the inequality

$$\lfloor\frac{U}{exu}\rfloor - 1 < \frac{U}{exu} \leq \lfloor\frac{U}{exu}\rfloor \tag{3.5}$$

holds as the property of the floor function, we can obtain

$$\lfloor\frac{U}{exu}\rfloor - 1 - \frac{1}{exu} < \frac{U-1}{exu} \leq \lfloor\frac{U}{exu}\rfloor - \frac{1}{exu}. \tag{3.6}$$

This fact indicates that the smallest integer larger than $\frac{U-1}{exu}$ *is* $\lfloor\frac{U}{exu}\rfloor$*. We derive the condition of* t_1 *and* t_2 *for the scheduling vector satisfying the PE number restriction as*

$$t_1 < t_2, \quad \lfloor\frac{U}{exu}\rfloor \leq t_2. \tag{3.7}$$

On the other hand, the total execution time is given by $t_{total} = t_1(U-1) + t_2(U-1)$*. The minimum total execution time requires the smallest* t_1 *and* t_2*, which satisfy the ineq.(3.7). They are*

$$T = (1, \lfloor\frac{U}{exu}\rfloor). \tag{3.8}$$

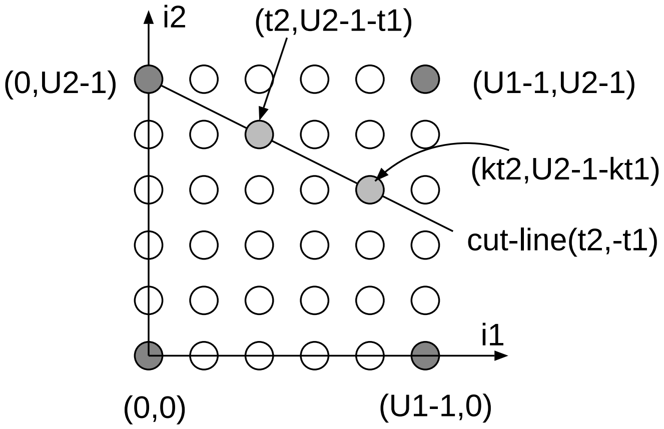


Fig. 3.3. Cut Line including the Largest Number of Nodes

3.1.4 Scheduling Theory for 3D Algorithm

3.1.5 Expressing 3D operation order on 2D plane

The scheduling problem is mainly a problem of finding out an optimal operation vector T with minimum total calculation time as well as avoiding computation conflicts and communication data link conflicts[64]. While maintaining regularity of the design procedure and the processor array, we try to extend the 2D solution to the 3D case. For this purpose, a 3D index space (i_1, i_2, i_3) is considered as a collection of 2D planes as $(i_1, i_2, 0), (i_1, i_2, 1), (i_1, i_2, 2), \cdots$. We describe the 3D case similarly as in a 2 Dimensional space of (j_1, j_2). This is a dimension reduction method, which simplifies n dimensional problem to 2D plane problem.

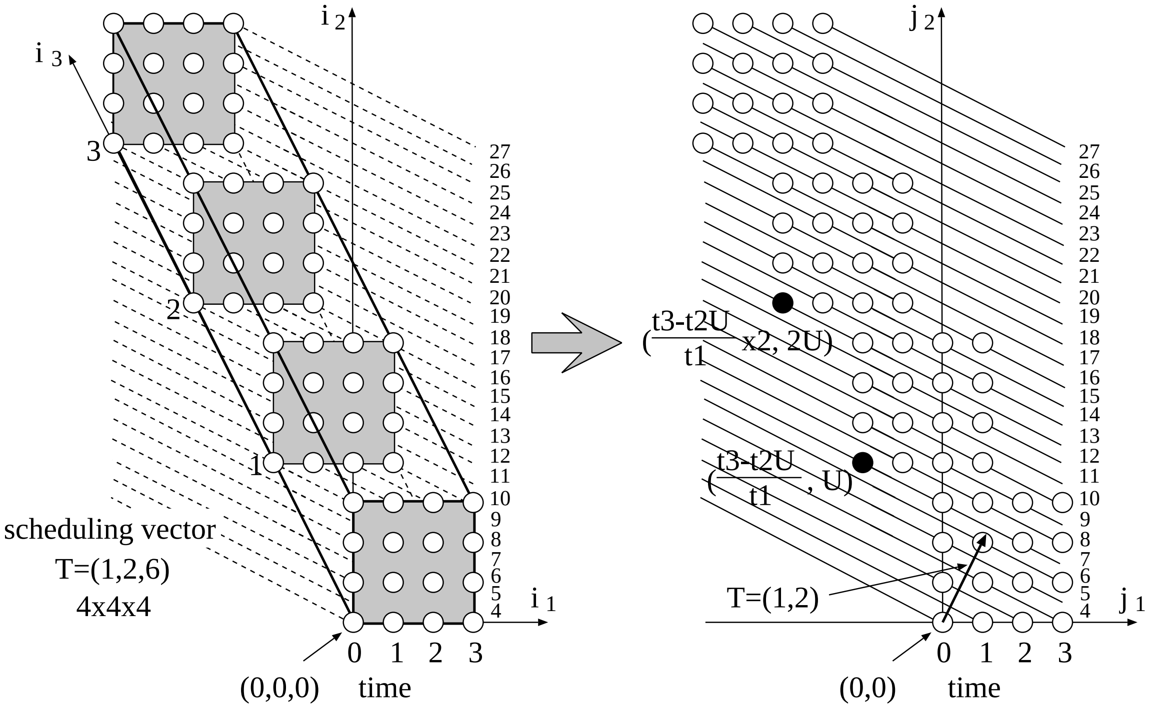


Fig. 3.4. Expressing 3 Dimensional Function on 2D Plane

As shown in Fig.3.4, 3D index space is represented by U 2D sub-planes. We express those 2D sub-planes without overlapping with each other in converted 2D plane. The first sub-plane is placed at the same position in the converted 2D plane. The second sub-plane is placed over the first one by adding U to the vertical coordinate. In order to extend the results of the linear mapping for 2D algorithms to 3D case, the horizontal position of the second sub-plane is decided to make the cut lines cross the concurrent operations lying in different sub-planes.

In the 3D case, the execution scheduling vector is denoted by $T = (t_1, t_2, t_3)$, while the execution index coordinate is denoted as $p = (i_1, i_2, i_3)$. In order to convert this 3D case into 2D problem, we introduce 2 dimensional index space $q = (j_1, j_2)$. The index point p is converted to q so as to coincide the two mappings as $Tp = Sq$. This condition leads to

$$j_1 = \frac{t_3 - t_2 U}{t_1} i_3 + i_1 \tag{3.9}$$

$$j_2 = Ui_3 + i_2 \tag{3.10}$$

relation between the original 3D coordinates and its converted 2D coordinates.

In this converted 2D index space,

$$\begin{array}{c} (i_1, i_2, 0) \text{ plane presides the range of } 0 \leq j_2 \leq U - 1 \\ (i_1, i_2, 1) \text{ plane presides the range of } U \leq j_2 \leq 2U - 1 \\ \cdots \end{array}$$

Therefore, the index points in original 3D plane do not overlap with each other.

Furthermore, we select the execution scheduling vector in the converted 2D plane as $S = (t_1, t_2)$. The simple calculation leads to the relation

$$Sq = t_1 j_1 + t_2 j_2 = t_1 i_1 + t_2 i_2 + t_3 i_3 = Tp. \tag{3.11}$$

The cut line defined by S in the converted 2D plane includes all concurrent execution nodes in the original 3D plane.

Fig.3.4 shows the 2D plane converted from $(4 \times 4 \times 4)$ 3D index space of the case $T = (1, 2, 6)$. $(i_1, i_2, 0)$ space occupies $0 \leq j_1 \leq 3$, $0 \leq j_2 \leq 3$ region in the converted (j_1, j_2) space. $(i_1, i_2, 1)$ space occupies $-2 \leq j_1 \leq 1$ and $4 \leq j_2 \leq 7$ region in the converted (j_1, j_2) space. The cut line of $Sq = 6$ as shown by the line in Fig.3.4 includes $(2, 2)$, $(0, 3)$, and $(-2, 4)$ index points in the 2D converted space. This cut line includes the index points of two planes in the original 3D space. This indicates that $(2, 2, 0)$, $(0, 3, 0)$ and $(0, 0, 1)$ index points in the original 3D space are scheduled to invoke at the same time by the vector $T = (1, 2, 6)$.

3.1.6 Available Scheduling Vector Constraint To Data Dependence

All the angles between any two of the dependence vectors must be less than 180°. Otherwise, there is no feasible linear scheduling vector T. The feasible range for the valid scheduling vectors is derived by the given dependency vectors as shown in Fig.5.

According to this constraint, the angle between the scheduling vector T and any data dependence vector must less than 90^0. Therefore, we generally get the feasible range which is described as $r_1 < \frac{t_2}{t_1} < r_2$ as shown in Fig.3.5. The scheduling vectors should locate inside the shaded range. The r_1 and r_2 correspond to the two boundary data dependencies d_1 and d_2.

3.1.7 MSPA 3D Scheduling

In the 3D case, there may be many solutions. One of the purpose of this section is to derive one analytical and practical solution. Therefore, we search a solution among the scheduling vectors whose cut lines do not pass through more than 2 planes of the original 3D space. This condition is described by the inequality $t_2 exu < 2U$.

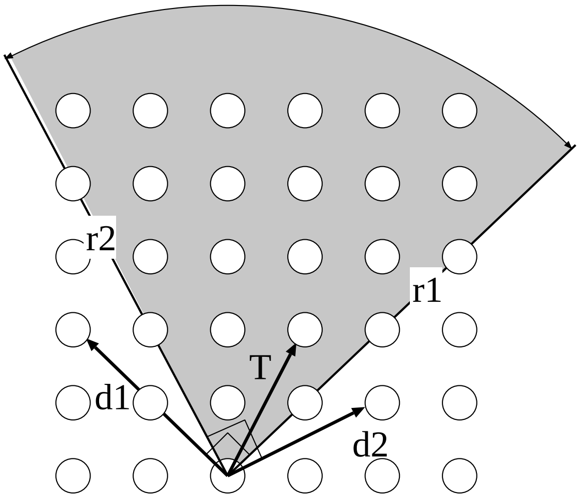


Fig. 3.5. Feasible range of the scheduling vectors

Theorem 2 (3D Scheduling Vector under PE number Restriction) *The operation scheduling vector for the uniform recurrence algorithm with three U iteration loops under the condition* $t_2 exu < 2U$ *is given by*

$$t_1 = 1$$
$$t_2 = \lfloor \frac{U}{exu} \rfloor$$
$$t_3 = t_1 U + t_2 U - t_1 t_2 exu.$$

Proof 2 *We consider this problem with the converted 2D plane as shown in Fig.3.6. The cut line, which passes through the index point* $(0, U-1)$ *in the converted 2D plane, lie over the tiled nodes. Therefore, the distance between the leftmost and rightmost nodes must be smaller than* $t_2 \cdot exu - 1$ *as*

$$(U-1) - (\frac{t_3 - t_2 U}{t_1}) \leq t_2 exu - 1 \tag{3.12}$$

$$t_3 \geq t_1 U + t_2 U - t_1 t_2 exu. \tag{3.13}$$

For the minimum total execution time, the smallest t_3 *should be selected. This fact together with the feasible range of* t_3 *leads to the solution of*

$$t_3 = t_1 U + t_2 U - t_1 t_2 exu. \tag{3.14}$$

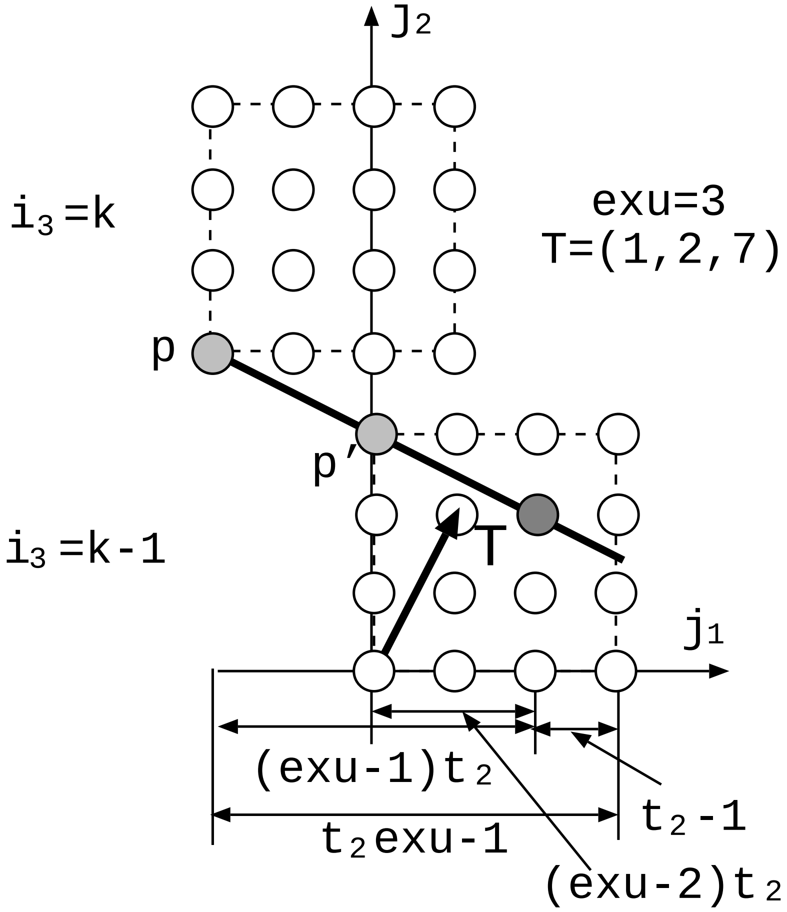


Fig. 3.6. The proof of Theorem 4

3.2 Resource Allocation Approach

As described in last chapter, we introduce a linear mapping with modular operations against the given algorithm.

$$Op \bmod M_O = \begin{pmatrix} (o_{11}i_1 + \cdots + o_{1n}i_n) \bmod m_x \\ (o_{21}i_1 + \cdots + o_{2n}i_n) \bmod m_y \end{pmatrix} \tag{3.15}$$

$$M_O = \begin{pmatrix} m_x \\ m_y \end{pmatrix} \tag{3.16}$$

Resource conflict free allocation: *Let A_{S_k} be the set of the index points for the operation f_k. For any two index points p and p' ($p = p' - d_{kl}$) of A_{S_k} which invoke at the same time ($Tp = Tp'$) , the mapping with O_k and M_{O_k} must satisfy the* *inequivalence of*

$$O_k p \not\equiv O_k p' \pmod{M_{O_k}}. \tag{3.17}$$

This is a necessary condition for conflict free allocation.

Local communication condition: *In order to minimize the MSPA, we should realize the data transfers between adjacent processing cells as much as possible. Viewing from this point, another necessary condition for local communication should be described as*

$$\Delta t_{kl} \geq \Delta dist_{kl} \tag{3.18}$$

where $\Delta dist_{kl} = dist(Op' \bmod M_O, O(p' - d_{kl}) \bmod M_O)$.

The communication time must be larger or equal to Manhattan distance between PEs, because it takes $\Delta dist_{kl}$ unit times for succesive local communications. This condition indicates that data transfers corresponding to the data dependencies can be realized by the successive data transfers between the adjacent PEs. The data transfers not satisfying the local communication condition are also realized by either direct links between PEs or data transfers via memory units.

In the case of the above matrix-vector calculation example, we obtain

$$O = (0, 1), \mathrm{mod} 3, \tag{3.19}$$

with a scheduling vector of $T = (1, 2)$. The resource allocation result is shown in Fig.3.7. The horizontal lines indicate that the nodes on a line are allocated to a PE. The bold line nodes are allocated to the same PE0 because of the modular operation of the mapping function. The data communication for data dependency vector d_{yy} is

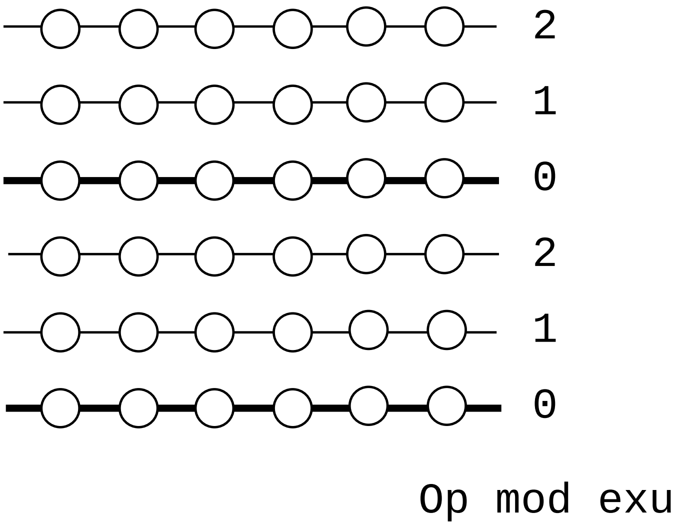


Fig. 3.7. MSPA Resource Allocation on a Linear Array

described by

$$\begin{aligned} Td_{yy} &= (1,2)(0,1)^T = 2 \qquad (3.20) \\ Od_{yy}\ mod\ 3 &= (0,1)(0,1)^T = 1. \qquad (3.21) \end{aligned}$$

This result indicates that data communication for d_{yy} satisfies the local communication condition. Therefore, the PE should send the data $y(i_1, i_2)$ to the adjacent PE $Od_{yy}\ mod\ 3 = 1$ far from the current PE (next right PE) within $Td_{yy} = 2$ unit times.

3.2.1 2D Resource Allocation Theory

We consider the mapping onto $exu_1 \times exu_2 (= exu)$ two dimensional processor array. It is difficult to derive the resource allocation vector in case that exu_1 and exu_2 have a common divisor. In order to develop the theory including this case, we introduce the resource allocation matrix including a fractional number. The allocation result may also be expressed by a fractional number. We assume that the actual PE coordinate is decided after the truncation of fraction number. For example, if PE allocation result is given by (2, 1.5), it is assumed to indicate a PE position of $(2, 1)$.

Theorem 3 (2D Conflict Free Allocation) *For 2 dimensional algorithm with an execution ordering vector $T = (t_1, t_2) = (1, t_2)$, the resource allocation matrix*

$$O = \begin{bmatrix} 0, & t_1 \\ t_1, & t_2 - t_1 + s\frac{t_1}{exu_1} \end{bmatrix} \quad M = \begin{bmatrix} exu_1 \\ exu_2 \end{bmatrix} \qquad (3.22)$$

where

$$s = \begin{cases} 0 & :exu_1 \text{ and } exu_2 \text{ with no common divisor} \\ 1 & :exu_1 \text{ and } exu_2 \text{ with a common divisor,} \end{cases}$$

can realize a conflict free allocation on $exu_1 \times exu_2 = exu$ processor array.

Proof 3 *A conflict free condition requires $O\Delta p_c \neq 0 \mod M$ for all the vectors Δp_c on a cut of $T\Delta p_c = 0$. The vectors on a cut are described by*

$$\Delta p_c = k \begin{bmatrix} -t_2 \\ t_1 \end{bmatrix} \qquad (3.23)$$

with an integer k of $1 \leq k \leq exu_1 exu_2 - 1$. The PE distance $dist(O\Delta p_c \mod M)$ for this vector Δp is given as

$$dist(kt_1^2 \begin{bmatrix} 1 & \mod exu_1 \\ -(1 - \frac{s}{exu_1}) & \mod exu_2 \end{bmatrix}). \qquad (3.24)$$

The first component of the resource allocation vector becomes zero as $(kt_1^2) \bmod exu_1 = 0$ *when* k *becomes a multiple of* exu_1*. Let consider the value of the second component in the case of* $k = k' \cdot exu_1$*, where* k' *satisfies the inequivalence of* $0 < k' < exu_2$*. The second component for this special* k *become*

$$[-kt_1^2(1 - \frac{s}{exu_1}) = k'(exu_1 - s)t_1^2] \bmod exu_2. \tag{3.25}$$

If exu_1 *and* exu_2 *do not have any common divisors, the term with* $s = 0$ *in eq.(3.25) does not become zero.*

On the contrary, if exu_1 *and* exu_2 *have a common divisor, we can select* $s = 1$ *because* $exu_1 - 1$ *and* exu_2 *are proved not to have any common divisors. In either cases, there guarantees the conflict free condition of* $Op_c \bmod M \neq 0$ *for any two index points on the cut.*

Theorem 4 (2D Dependence Vector Range for Local Communication): *The resource allocation matrix described by eq.(3.22) satisfies the local communication condition for all the precedence vectors in the first quadrant of its resource allocation coordinates.*

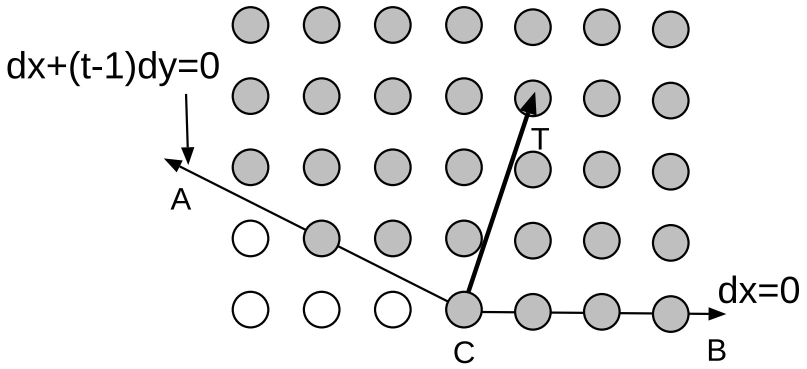


Fig. 3.8. Range of Local Communication Dependence Vectors

Proof 4 *Let the precedence vector be denoted by* $d_{kl} = (d_x, d_y)^t$*, which corresponds to the data transfer between PEs. The data transfer time* Δt *of the precedence vector is expressed by*

$$\Delta t = T d_{kl} = t_1 d_x + t_2 d_y. \tag{3.26}$$

The distance between PEs for the data transfer is described by

$$\begin{aligned} & dist(Od_{kl} \bmod M) \\ = \; & dist \begin{bmatrix} t_1 d_y & mod\ exu_1 \\ t_1 d_x + (t_2 - t_1) d_y + s t_1 \frac{d_y}{exu_1} & mod\ exu_2 \end{bmatrix} \end{aligned}$$

$$= \quad dist \begin{bmatrix} t_1 d_y & mod\ exu_1 \\ \Delta t - t_1 d_y + s t_1 \frac{d_y}{exu_1} & mod\ exu_2 \end{bmatrix}.$$

Since the precedence vector in the first quadrant of the resource allocation coordinates satisfies $t_1 d_y \geq 0$ *and* $\Delta t - t_1 d_y \geq 0$*, the distance is further described by* $dist(Od_{kl} \bmod M) = (t_1 dy \bmod exu1) + (\Delta t - t_1 d_y + s t_1 \frac{d_y}{exu_1} \bmod exu_2)$.

We consider the range where $kexu_1 \leq d_y < (k+1)exu_1$ *or* $k \leq \frac{d_y}{exu_1} < k+1$ *for any positive integer* k*. Then, in the case of the actual allocation,* $\frac{d_y}{exu_1}$ *is evaluated by an integer as* k*. Therefore, the inequality as*

$$\begin{aligned} dist(Od_{kl}) &\leq t_1(d_y - kexu_1) + (\Delta t - t_1 d_y + s t_1 k) \\ &\leq \Delta t - t_1 k(exu_1 - s) \leq \Delta t \end{aligned}$$

holds.

Furthermore, within the small range of $exu_1 > t_1 d_y \geq 0$ and $exu_2 > t - t_1 d_y \geq 0$, the transfer time equals to its PE distance so that the data transfer is achieved locally without the additional data storage for the timing adjustment. Fig.3.8 shows the range of the dependence vectors in the first quadrant of $Od_{kl} \geq 0$, in which data transfers of dependence vectors originated from node C to the shaded nodes are realized by local communications. Since the derived feasible range for the dependence vector covers a wide area so that the scheduling and the resource allocation vectors given by theorems can be used in practical applications.

Corollary 1 (2D Conflict Free Allocation on Linear Array): *For a 2-dimensional algorithm with an execution ordering vector* $T = (1, t_2)$*, the resource allocation matrix*

$$O = (0, 1), \ \ M = exu \tag{3.27}$$

can realize a conflict free allocation on $exu \times 1$ *linear processor array.*

Here, we will show some resource allocation results on two dimensional array for 2 dimensional matrix-matrix multiplication examples.

1. $exu = exu_1 \times exu_2 = 3 \times 2$ case with $U \times U = 16 \times 16$ algorithm size
   Since exu_1 and exu_2 do not have any common divisor, we can select the parameter $s = 0$ of the allocation matrix. The scheduling and allocation vectors are given by

$$\begin{aligned} T &= (1, 3) \\ O &= \begin{bmatrix} 0, & 1 \\ 1, & 2 \end{bmatrix} \end{aligned}$$

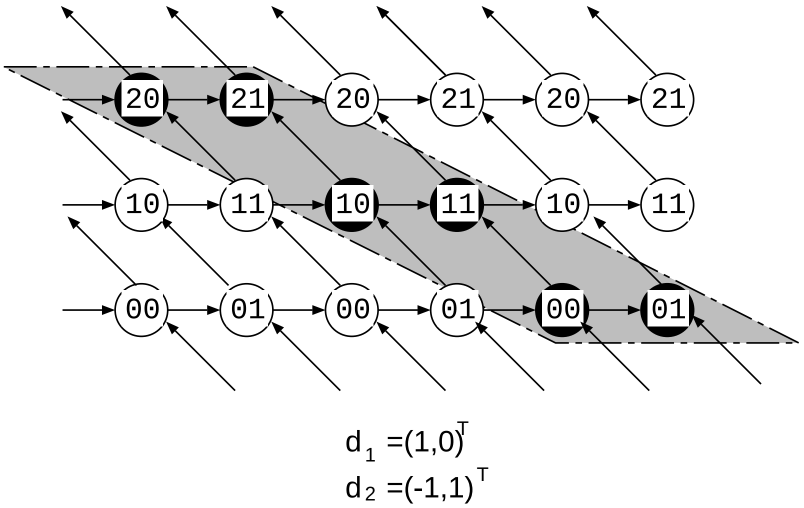


Fig. 3.9. Resource Allocation on 3 × 2 Processor Array

The resource allocation result is shown in Fig.3.9. In Fig.3.9, the numbers in the nodes express the allocated PE coordinates of the nodes. The shaded area including 6 nodes corresponds to the PE placement with local communication links.

With a similiar method to resolve resource allocation problem, the interconnection network of the array can also be described by a matrix. The MSPA is configurated as shown in Fig.3.12.

With regards to the interconnection matrix, it is explained more detail as follows. There are two data dependencies in the application, which are expressed as:

$$d_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

and

$$d_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Based on the two data dependencies, the interconnnection matrix I becomes:

$$I_1 = Od_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

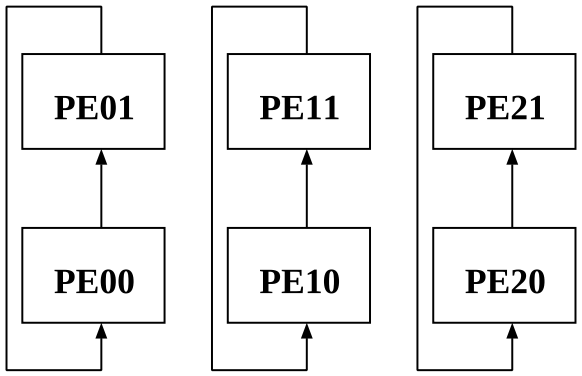


Fig. 3.10. PEs Interconnection for Data Dependency d_1 in 3×2 Processor Array

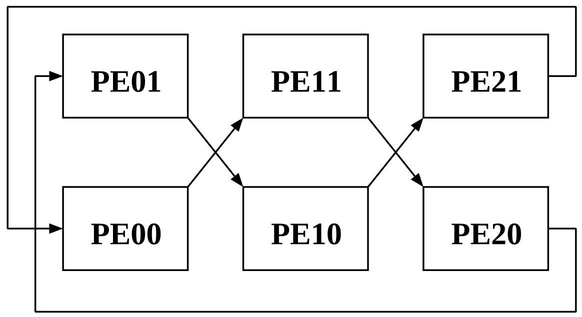


Fig. 3.11. PEs Interconnection for Data Dependency d_2 in 3×2 Processor Array

$$I_2 = Od_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

The interconnection of the PEs corresponding to the two data dependencies can be achieved by referring to (I_1 mod exu_1) and (I_2 mod exu_2), which are shown in Fig.3.10 and Fig.3.11, respectively. Combining both Fig.3.10 and Fig.3.11, the configuration of MSPA is obtained as shown in Fig.3.12.

2. $exu = exu_1 \times exu_2 = 4 \times 2$ case with $U \times U = 16 \times 16$ algorithm size
   This is the case that exu_1 and exu_2 have a common divisor. With the parameter $s = 1$, we get the following mapping vectors.

$$T = (1, 2) \quad O = \begin{bmatrix} 0, & 1 \\ 1, & 1.25 \end{bmatrix}$$

The resource allocation result is shown in Fig.3.13. The configuration of the 2 dimensional PEs is shown in Fig.3.14.

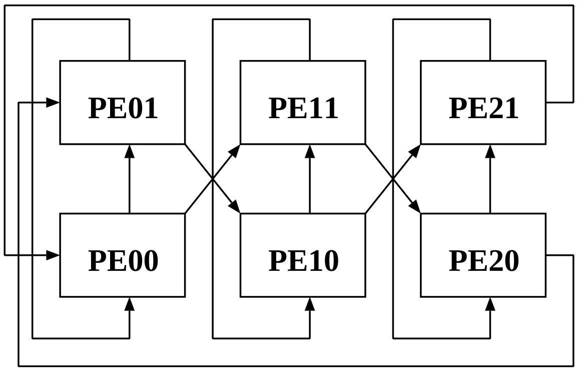


Fig. 3.12. Configuration of 3 × 2 Processor Array

3. $exu = exu_1 \times exu_2 = 8 \times 1$ linear array case with $U \times U = 16 \times 16$ algorithm size
   The scheduling and allocation vectors are given as
   $$T = (1, 2) \quad O = \begin{bmatrix} 0, 1 \end{bmatrix}.$$
   Fig. 3.15 shows the PEs allocation result in the case of $exu_1 = 8$ and $exu_2 = 1$. Fig.3.16 shows the configuration of the PEs in 8x1 linear array processor.

3.2.2 3D Resource Allocation Theory

We can resolve 3D resource allocation by converting the 3D coordinate $p = (i_1, i_2, i_3)$ into 2D expression $q = (j_1, j_2)$. It is easy to show $O\Delta p \neq 0$ for all the difference vectors Δp between index vectors on a cut in 3D plane by converting a conflict free resource allocation of

$$O^{(2)}\Delta q_c = \begin{bmatrix} 0 + t_1\Delta j_2 \\ t_1\Delta j_1 + (t_2 - t_1 + s\frac{t_1}{exu_1})\Delta j_2 \end{bmatrix} \tag{3.28}$$

in the converted 2D plane into the relation in the original 3D coordinate. By substituting the relation between Δp and Δq into eq.(3.28), we obtain the following theorems.

Theorem 5 (3D Conflict Free Allocation) *For 3 dimensional algorithm whose execution ordering vector is represented by* $T = (t_1, t_2, t_3)$ *with* $t_3 = t_1U + t_2U - t_1t_2exu$*, the following resource allocation matrix can realize the conflict free allocation on* $exu_1 \times exu_2 = exu$ *2 dimensional processor array.*

$$O = \begin{bmatrix} 0, & t_1 & t_1U \\ t_1, & t_2 - t_1 & t_3 - t_1U + s\frac{t_1}{exu_1}U \end{bmatrix}$$

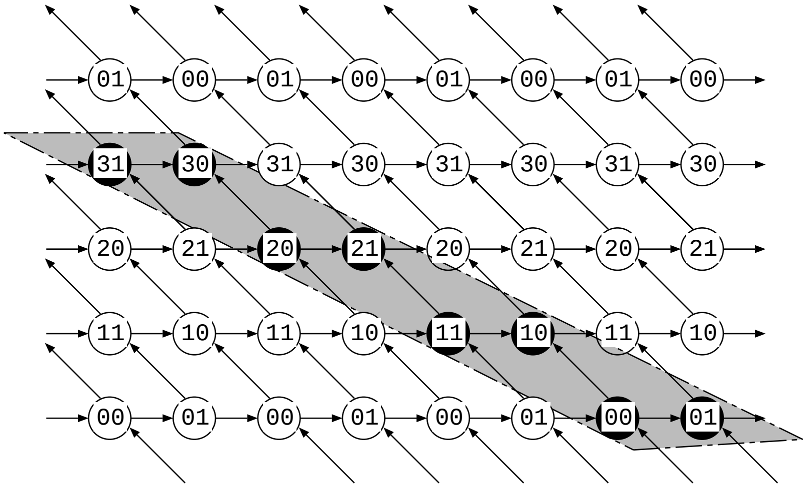


Fig. 3.13. Resource Allocation on 4 × 2 Processor Array

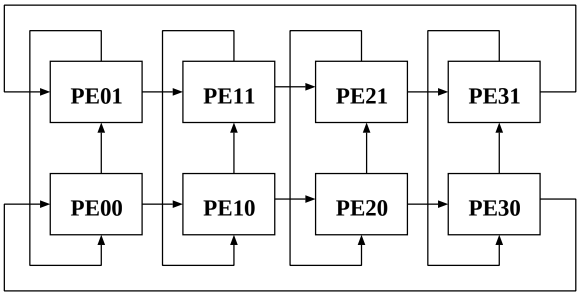


Fig. 3.14. Configuration of 4 × 2 Processor Array

$$M = \begin{bmatrix} exu_1 \\ exu_2 \end{bmatrix} \tag{3.29}$$

The parameter s should be selected as

$$s = \begin{cases} 0 : exu_1 \text{ and } exu_2 \text{ with no common divisor} \\ 1 : exu_1 \text{ and } exu_2 \text{ with a common divisor.} \end{cases}$$

Corollary 2 (3D Conflict Free Allocation on Linear Array) *For 3 dimensional algorithm whose execution ordering vector is represented by $T = (t_1, t_2, t_3)$ with $t_3 = t_1U + t_2U - t_1t_2exu$, the following resource allocation matrix can realize the conflict free allocation on $exu \times 1$ linear processor matrix.*

$$O = \begin{bmatrix} 0, & t_1 & t_1U \end{bmatrix} \quad M = exu \tag{3.30}$$

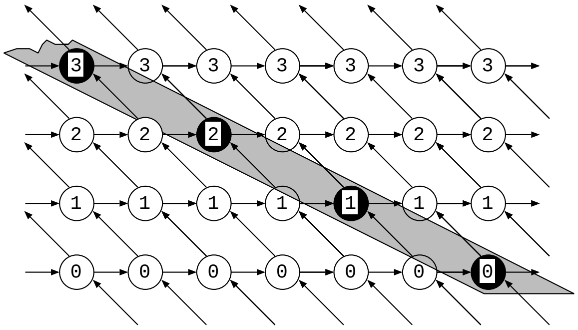


Fig. 3.15. Resource Allocation on Linear Processor Array

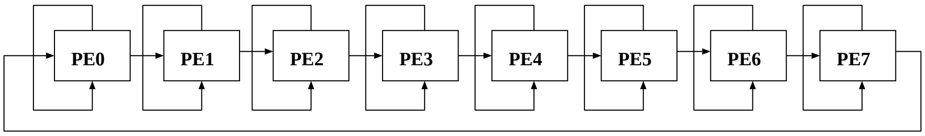


Fig. 3.16. Configuration of 8 × 1 Processor Array

By dealing with a 3-D problem in 2-D space, we can also prove the following theorem.

Theorem 6 (3D Dependence Vector Range for Local Communication) *The resource allocation matrix described by eq.(3.29) satisfies the local communication condition for all the precedence vectors in the first quadrant of its resource allocation coordinates.*

3.3 MSPA Memory Management

After we design the scheduling vector T and the resource allocation matrix O, the data communication schemes corresponding to the data dependence vectors are derived. In our matrix-vector example, the multiple data streams to PE array are shown in Fig.3.17. Three data streams of input $x(i,j)$ are provided by the temporary memory which plays a function of data format conversion. The load pipeline for run-time input $x(i,j)$ is performed in parallel with execution pipeline. The temporary memory acts both data storage and the function of a data format converter (DFC). The DFC converts input data from serial scheme into parallel scheme according to operation scheduling. The order of data streams from temporary memory is subject to the requirement of PEs. Other three data streams of input $A(i,j)$ are provided by memory. Since the number of matrix elements is the same as the number of function nodes, there must be input before the execution starts. That is to say the input

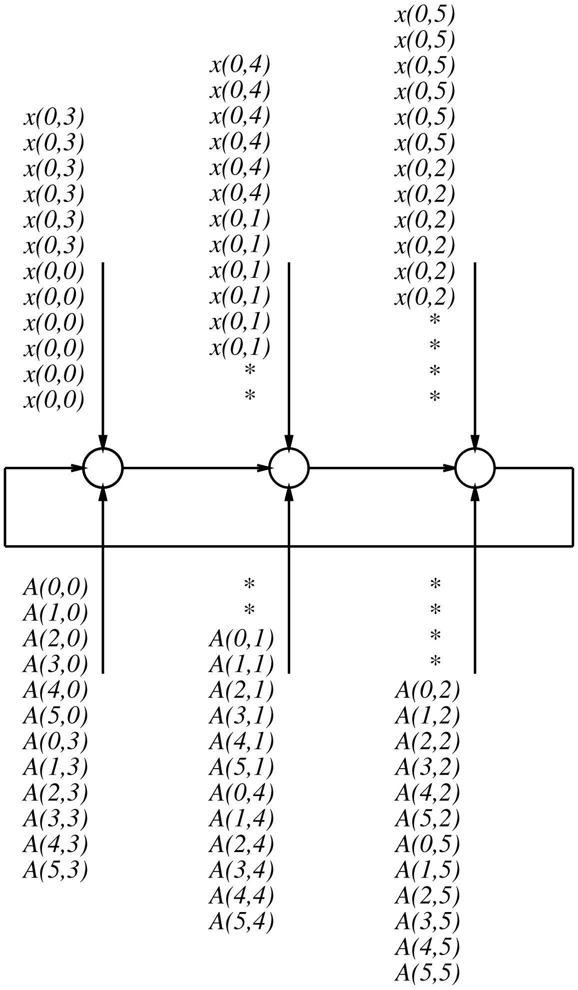


Fig. 3.17. Data Stream to PE Array

scheme $A(i, j)$ is *the compile time input.* The order in each data stream is determined by the memory AGU.

Fig.3.18 shows the memory unit which stores matrix elements of input $A(i, j)$ in advance. The memory size is decided by the matrix size. The memory holds the data in matrix form differently from the systolic array case. Systolic array requires data storage which bring data streams to the parallel processing PEs. Therefore, it needs much larger memory space due to the frequent reuse of the same input data in different PEs. In the proposed case, the memory only needs to store the input matrix. Then, the orders of each data stream from memory to PEs are controlled by the memory AGU. Driven by the AGU, the multiple read-port memories produce plural data streams to PE array. Each read-port of the memory needs one AGU. In the figure, there are three PEs in the array. Then one three read-port memories or three

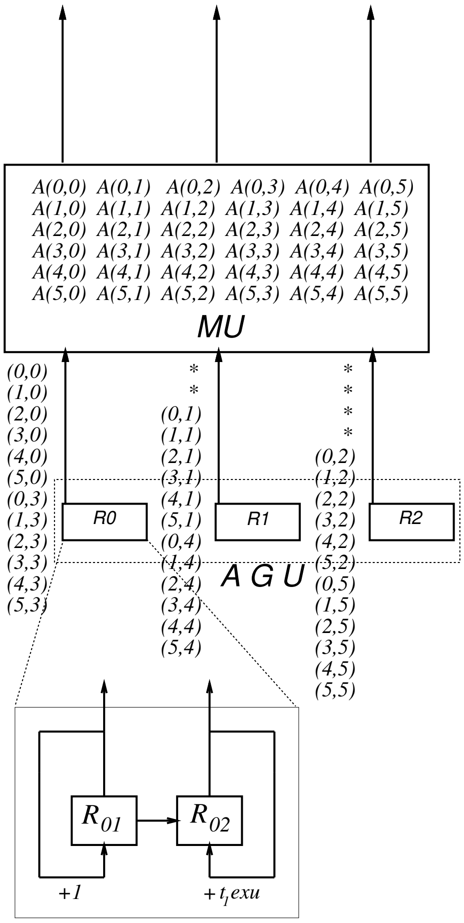


Fig. 3.18. Memory for Matrix Elements

one read-port memory and three AGUs are required. The three AGUs include three address registers R_0, R_1 and R_2 respectively. Each AGU address register consists of two sub-registers R_i1 and R_i2 $(i = 0, 1, 2)$ with simple calculation mechanism as shown in Fig.3.18. The value of each index point register R_ij $(j = 1, 2)$ expresses one of the address elements of the data to be read from memory. There are two address elements, row and column addresses for the matrix memory management.

Fig.3.19 shows the temporary memory unit for run time input $x(i, j)$. The vector elements are input from I/O port in the serial data format and stored in the temporary memory units. At the same time the temporary memory works in DFC mechanism in order to convert the data format to three parallel data streams. Each vector element is repeatedly loaded to PE array from memory with only six registers. The number of registers are decided by the lifetime of each input data. This scheme reduces the data storage requirement together with conversion from data-serial to multiple data streams by data circulation principle among the registers based on DFC theory. The

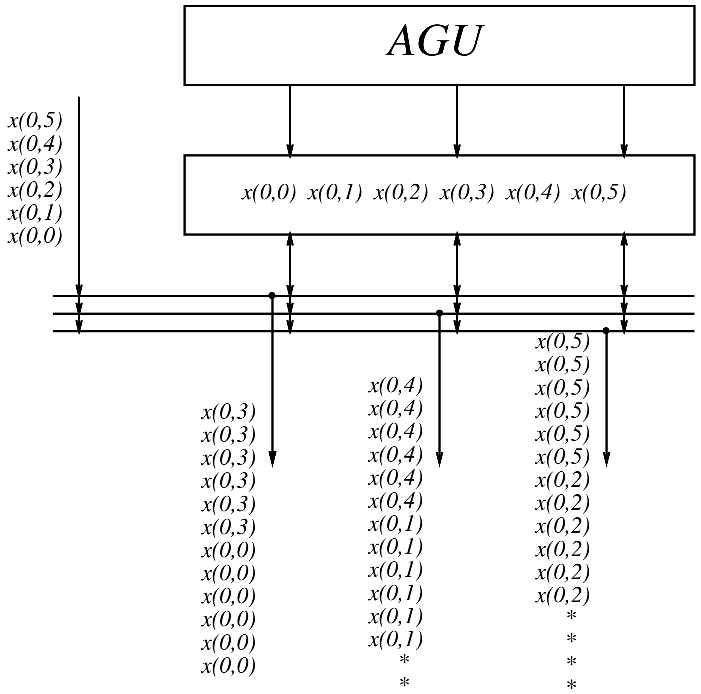


Fig. 3.19. Memory for Vector Elements

derived MSPA Matrix-Vector Multiplier is shown in Fig.2.1.

3.3.1 MSPA 3D Memory Management

In the 3D case, memory management is almost the same as in the 2D case, since the relation $t_3 = t_1U + t_2U - t_1t_2exu$ brings a similar situation in the converted 2D plane. We prepare exu index point register $R_u = (R_{u1}, R_{u2}, R_{u3})$ $u = 0, 1, \cdots exu-1$, each sub-register of which have a length of t_2exu , U and U respectively. Index point register R_u generates only one index point of the corresponding u-th, $u + exu$-th, $\cdots$-th rows in the converted 2D index point space at a time.

The operation of the index point register is as follows. R_{u1} increments $+1$ at every time steps. When R_{u1} reaches $t_1exu - 1$, it produces carry, which increments R_{u2} by $+t_1exu$. While R_{u1} is between U and $t_2exu - 1$, the register R_u indicates an invalid index points.

If R_{u1} reaches $U - 1$ and R_{u2} is greater than or equal to $U - t_1exu$, R_{u1} and R_{u2} produce a carry. Then, this carry increments R_{u3} by $+1$. At the same time, R_{u2} is decremented by $U - t_1exu$ and R_{u1} is reset. The generation scheme for an example of $exu = 8, U = 9, t_1 = 1, t_2 = 2, t_3 = 11$ is shown in Fig.3.20. The directed track is indicated in Fig.3.20 for the case $u = 0$. The time sequence of the index points is

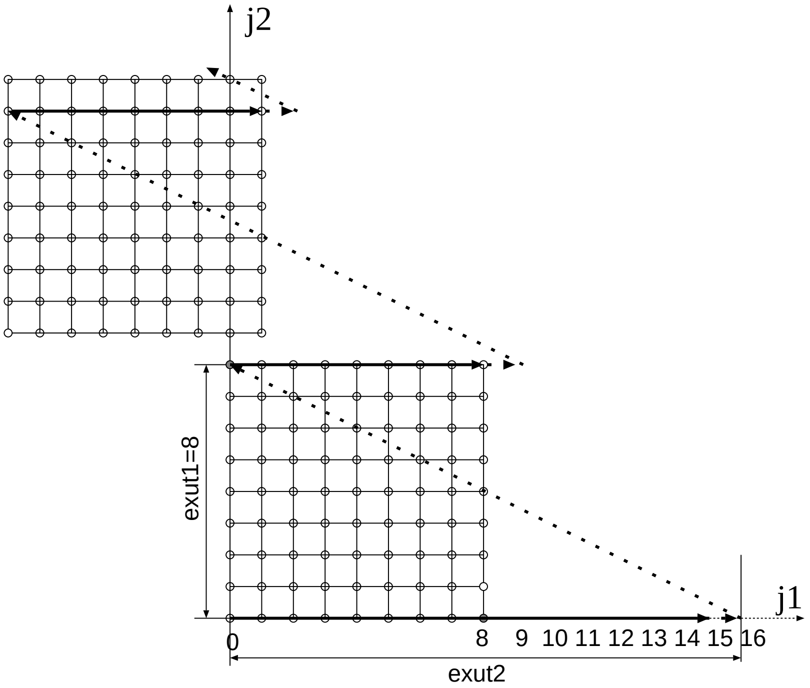


Fig. 3.20. Execution Index Point Generation

described as

$$\begin{array}{ccccc} (0,0,0) & (1,0,0) & \cdots & (8,0,0) & \cdots \quad (15,0,0) \\ (0,8,0) & (1,8,0) & \cdots & (8,8,0) & \\ (0,7,1) & (1,7,1) & \cdots & (8,7,0) & \\ (0,6,2) & \cdots & \cdots & & \end{array}$$

In this case, the increment of R_{02} occurs at $(15,0,0)$. The increment of R_{03} with decrement of R_{02} by $1 = U - t_1 exu$ occurs at $(8,8,0)$ and $(8,7,0)$ because $R_{01} = 8 = U - 1$ and $R_{02} \geq 1 = U - t_1 exu$ hold.

3.3.2 Design Consideration for Adress Generation Unit

For 3-D U dimensional algorithms, we present an algorithm to generate execution index point addresses denoted by p=(i_1, i_2, i_3). The algorithm is designed so as to realize AGU without using any complex arithmetics. Let $R_k = (i_1, i_2, i_3)$ be the k-th register storing the current index points. Then we can set exu registers $R_0, R_1, \cdots, R_{exu-1}$ to store execution index addresses required by exu PEs at every unit times. In order to simplify the hardware address generation procedure of the execution indices, we fix the destination address of every R_k by making the R_k number $(k = 0, 1, \cdots, exu - 1)$ correspond to the PE_k number $(k = 0, 1, \cdots, exu-1)$. This means that the generated address streams stored in any $R_l(i_1, i_2, i_3)(l \in (0, 1, \cdots, eux-1)$ must be used to read out the data from memory to $PE_l(l \in (0, 1, \cdots, eux-1)$. In order to reduce hardware area, we assume $t_1 \leq t_2 \leq t_3$ and take the following strategies to complete the index point generation procedure.

(1) Use a increment unit to increase i_1 in every execution steps.

(2) Use a ALU unit to generate a new i_2 in order to start a new i_2 coordinate.

(3) Use a increment unit to generate a new i_3 in order to record a new plane.

(4) By use of mathematical relations among the algorithm, T, O, and PEs number, we can generate i_2, i_3 according to the value of i_1, and new i_2, i_3 will be generated when only their values are renewed.

According to the analysis, we can use the proposed algorithm described by Table I to design AGU for execution index address generation required by every PEs automatically.

```
loop1 for $i_1 = 0, 1, \cdots, t_2 exu$ step 1
    $i_1 \leftarrow i_1 + 1$ (P1)
        if1 $i_1 = t_2 exu \cap i_2 < \Delta(\Delta = U - t_2 exu)$ (P2)
            (1) $i_1 \leftarrow 0$
            (2) $i_2 \leftarrow i_2 + t_1 exu$
            (3) loop1
        else if2 $i_1 = U - 1 \cap \Delta \leq i_2 \leq U - 1$
            if3 $i_3 = U - 1 \cap i_2 = U - 1$
                stop
            else (P3)
                (1)$i_1 \leftarrow 0$
                (2)$i_2 \leftarrow i_2 - \Delta$
                    if4 $\Delta \leq i_2 \leq U - 1 \cap i_3 < U - 1$ (P4)
                        (1) $i_3 \leftarrow i_3 + 1$
                        (2) loop1
                    else loop1
        else loop1
```

Table I Algorithm of index address generation for execution

We explain the principle of the generation algorithm. As shown in Fig.3.20, the next execution index point $p'(i'_1, i'_2, i'_3)$ can be generated by the index point $p(i_1, i_2, i_3)$ invoked at the current execution time step. If the $i_1 < U - 1$, we can finish the index point generation procedure by only increasing the value of i_1 to $i'_1 = i_1 + t_1$, while i_2, i_3

hold unchangeable. Otherwise, when i_1 get to its boundary value U-1, we generate a new i_2 (some time also i_3 when p and p' locate on different planes).

The difference $\Delta p(\Delta i_1, \Delta i_2, \Delta i_3)$ between the index point $p(i_1, i_2, i_3)$ and $p'(i'_1, i'_2, i'_3)$ can be expressed by the difference of the indices in the converted 2-D plane as

$$\begin{array}{l} \Delta j_1 = \frac{t_3 - t_2 U}{t_1} \Delta i_3 + \Delta i_1 \\ \Delta j_2 = U \Delta i_3 + \Delta i_2. \end{array}$$

The generation of new i_2

Since the $R_l(i_1, i_2, i_3)$ covers the execution indices on every exu rows, the generation of new i_2 occurs when the following condition holds in the converted 2D plane as

$$\begin{array}{l} \Delta j_1 = t_2 exu \\ \Delta j_2 = t_1 exu \end{array}$$

where $\Delta j_1 = j'_1 - j_1$ *and* $\Delta j_2 = j'_2 - j_2$. There are two cases of $\Delta i_3 = 0$ or 1 for this i_2 generation.

When $\Delta i_3 = 0$ holds, the p and p' are on the same plane. Therefore, we obtain the relation such as

$$\begin{array}{l} \Delta i_1 = \Delta j_1 = t_2 exu \\ \Delta i_2 = \Delta j_2 = t_1 exu \end{array}$$

for the condition of i_2 generation. It shows that i_2 must be generated at the moment when $i_1 = t_2 exu$. This means that the i_1 must be increased to the value of $t_2 exu$, and during the time from moment U to $t_2 exu - 1$, the processor works in an idle state. This is the principle of P2 shown in Table I. At moment when $i_1 = t_2 exu$, the i_2 is changed with an increment of $t_1 exu$.

When $\Delta i_3 = 1$ holds, the p and p' are located on two different adjacent planes. In this case, we obtain the relation as

$$\begin{array}{l} \Delta i_1 = \Delta j_1 - \frac{t_3 - t_2 U}{t_1} = U \\ \Delta i_2 = \Delta j_2 - U = -(U - t_1 exu) = -\Delta \end{array}$$

where $\Delta = U - t_1 exu$.

These equations mean that i'_2 must be generated at the moment of $i_1 = U$, and the increment is $-\Delta$. This part corresponds to the hardware P3 in Table I.

The generation of i_3

A new generation of i_3 occurs when i'_2 exceeds the boundary of one 2D sub-plane. It is expressed by the condition of $i_2 + t_1 exu \geq U$ or. $i_2 \geq \Delta$. Therefore, i'_3 is generated only under the condition that $i_2 \geq \Delta$ (P4).

Chapter 4

MSPA with Data Format Converter

In this chapter, we describe more practical ASIC design methodology derived from the MSPA design for a recurrence uniform algorithm [65]. The new design methodology is developed with emphasis on a run-time serial input, registers for data format conversion, and a linear processor array [66]. In the general case which includes both run-time and compile-time inputs, several memory units must be used for data conversion in addition to registers. This chapter only deals with the run-time serial input case.

The execution ordering and resource allocation described here are similar to the MSPA which are described in a previous chapter. Data conversion is employed, which converts from regular inputs such as bit by bit for bit-serial inputs, and line by line (or row by row) for data-serial inputs to multiple data streams into the array. Parameters to describe input scheme are introduced to develop systematically data format converters (DFC).

By demonstrating the design of a two's complement multiplier with the proposed design methodology, we make it clear that the proposed methodology is available for both bit-serial and data-serial input schemes, and achieves higher parallel efficiency than conventional methodologies do.

4.1 Design Procedure

This chapter focuses on the DFC design. Some results such as vector/matrix mapping theories of execution ordering and resource allocation derived from previous chapters are used in this chapter. Fig.4.1 shows the proposed design procedure. At first, an initial number of processor elements is chosen. Next, corresponding operation scheduling vector T and resource allocation matrix O corresponding to the number of PEs are searched. Then, based on these derived parameters and input schemes, DFCs are synthesized. The cost-performance is defined as the product of the total

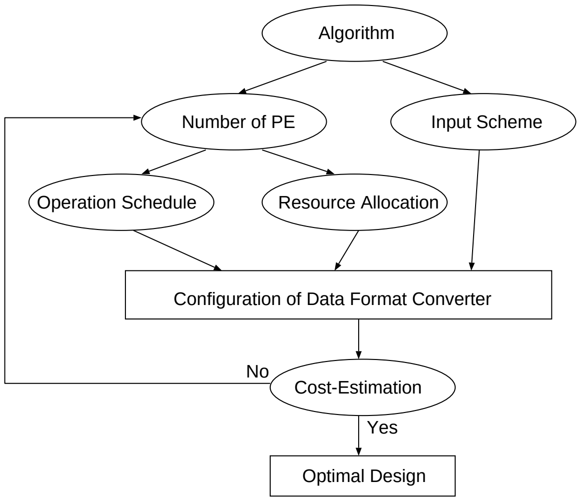


Fig. 4.1. The Design Procedure Flow Chart

execution time and the total equivalent number of flip-flops in the architecture. By estimating the cost-performance of the design for the current number of PEs, it is possible to obtain an optimal architecture by this design procedure. Since the design procedure is automatic for ASIC design, the design methodology can be extended to a reconfigurable and/or variable algorithm size design. The most obvious contribution of this procedure comparing with the original MSPA design lies in the introduction of an automatic generation of DFCs.

4.2 Data Format Converter Synthesis

Both the execution ordering and resource allocation determine the data communication schemes which include the input data loading and the distribution of input data and variable data. Here a systematic serial input data loading and distribution methodology to PE array are considered by introducing the concept of data format converters.

A systematic synthesis methodology for DSP data format converter architectures with a minimum number of registers have been proposed [67]. However, the register organization designed by the method requires a complicate control mechanism, so that the chip area of the converter becomes so large as to degrade the advantage of the minimum number of registers. We have developed a new method based on register units to generate data format converters. By limiting the number of registers in each register unit, we desire to achieve a minimum number of registers. At the same time, the control mechanism of each register unit is simplified so as to achieve a lower hardware cost.

The data input time denoted by t_{in} is defined as the time when data is input from

I/O port to a register unit of DFCs in a run-time serial input scheme. The data load time t_{load} is defined as the time when data is delivered to a PE from a DFC unit (DFCU).

Necessary condition for available register DFC

Let an input set be denoted by $A(i_1, i_2, i_3)$ $(i_k \in (0, 1, \cdots, U-1)(k \in (1, 2, 3))$. *Then if the inequivalence*

$$t_{in_k} \leq t_{load_k}(k \in A(i_1, i_2, i_3)) \tag{4.1}$$

is satisfied, the register DFC is useful for the application.
t_{in_k} represents the input time from I/O to DFC of a k-index datum and t_{load_k} represents the time when the k-index data is delivered to a PE from the DFC.

Any input which satisfies eq.(4.1) is called run-time input. Otherwise, it is called compile-time input. Although we consider only run-time inputs, which can be easily handled using register-based data format conversion, it is worth noting that compile-time inputs are well processed using shared memory units.

The DFC consists of exu DFC units, and the positions of DFCUs are described in the space coordinate as $0, 1, 2, \cdots, exu - 1$, which also correspond to the positions of the corresponding processor cells in a linear array.

4.2.1 DFC Data Path Configuration

The proposed DFC consists of two stages of registers, one is called *load register stage*, and the other is called *circulate register stage*. The first one is used to load data from I/O port to DFCU, and the second one is used to reuse loaded data. A load-circulate select control unit generates control signals of the DFCs. When the load-circulate select control signal is high, DFC goes into its input loading state. Otherwise, it remains in a circulating state. By generating no-overlapping load-circulate select controls for every DFCU, DFCUs can accept input data successively without I/O port conflict with each other. Using this type of architecture, the register number can be reduced to the least extent.

Data Load

For an input variable at input time t_{in}, the number of DFCU, denoted as NR_1, in the first stage equals the time difference between the load and input times of the input variable as

$$NR_1 = \triangle t_{load} = t_{load} - t_{in}. \tag{4.2}$$

Since the input data are used for the operation located at index point $p(i_1, i_2, i_3)$, the load time t_{load} of the input from I/O port is given by:

$$t_{load} = Tp(i_1, i_2, i_3). \tag{4.3}$$

The load position P_{load} of the input variable at the index coordinate p_{in} is expressed as

$$P_{load} \;=\; Op_{in} \bmod exu. \tag{4.4}$$

The above result illustrates that the input variable p_{in} at the input time t_{in} is loaded to Op_{in}-th PE by $(t_{load} - t_{in})$ cycle delays after data-serial input starts. This scheme can be realized by the a cascade connection of the first stage registers in every Op_{in} register units, which consists of $(t_{load} - t_{in})$ registers. Fig.4.2 shows the architecture of a load stage registers with 4 kinds of different design parameters. Each shaded box represents one register with one cycle delay, and i_2 indicates one of its coordinates in index point space.

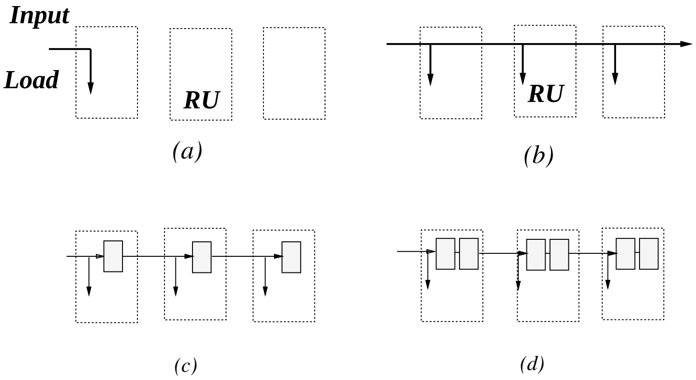


Fig. 4.2. The First Stage Register (a)$\triangle t_{load} = 0, P_{load} = 0$, (b)$\triangle t_{load} = 0, P_{load} = i_2$, (c)$\triangle t_{load} = i_2, P_{load} = i_2$, (d)$\triangle t_{load} = 2i_2, P_{load} = i_2$

Data Circulation

After the first register stage allocation, the method to distribute input variables through register units must be decided. They are repeatedly reused by the processor array. This scheme is achieved by employing the second stage registers in each register unit.

Consider the case when the same input variable at the index coordinate p_{in} is reused by an execution at the differential coordinate $(p_{in} + \Delta p_{circ})$. Because the PE positions to consume the variable with index coordinates of p_{in} and $(p_{in} + \Delta p_{circ})$ are specified by $(Op_{in} \bmod exu)$ and $(O(p_{in} + \Delta p_{circ}) \bmod exu)$, respectively, the loaded data of the input are transferred to the register unit with the differential position rightward of

$$\triangle P_{circ} = O\Delta p_{circ} \tag{4.5}$$

from current register unit. The transfer time required by the procedure should be described as

$$\triangle t_{circ} = T \Delta p_{circ}. \tag{4.6}$$

This scheme can be realized by a cascade connection of the second stage of $\triangle t_{circ}$ registers with Δp_{circ} position rightward. Fig.4.3 shows configurations of circulation stage registers with two types of synthesis parameters.

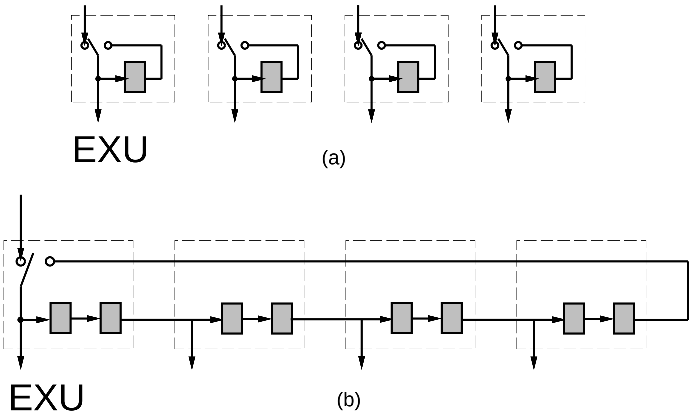


Fig. 4.3. The Second Stage Register (a)$\triangle t_{circ} = 1, \triangle P_{circ} = 0$, (b)$\triangle t_{circ} = 2, \triangle P_{circ} = 1$

4.3 Bit-Serial Multiplier

A practical design application of a new bit-serial data-path synthesis methodology is presented here as an design example with DFC theory [69]. The algorithm description for a bit-serial multiplier $[x_j][y_i]$ for an 8-bits serial multiplier is given by

```
for $i = 0$ to $7$ step $1$
for $j = 0$ to $7$ step $1$
    $o(i,j) = f_1(x(0,j), y(i,0), o(i+1, j-1), c(i-1, j))$
    $c(i,j) = f_2(x(0,j), y(i,0), o(i+1, j-1), c(i-1, j))$,
```

where $o(i+1,-1) = 0,\ c(-1,j) = 0$. The function f_1 corresponds to the partial sum of one bit full adder, while the function f_2 generates the carry of one bit full adder.

Design Procedure

For simplicity, the case of an 8-bit × 8-bit multiplier with 4 full-adders is explained. The optimal schedule and resource allocation vectors are derived as

$$T = (1, 2) \quad O = (0, 1) \quad exu = 4. \tag{4.7}$$

This schedule achieves a total execution within 22 steps.

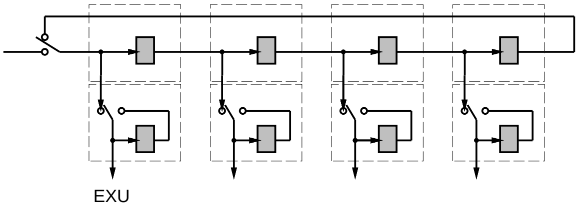


Fig. 4.4. The x_j's Data Format Converter

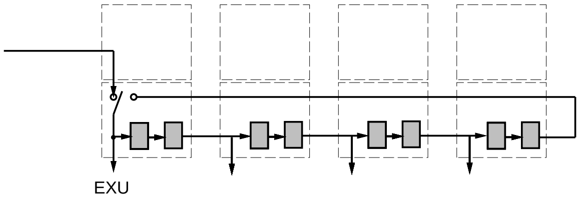


Fig. 4.5. The y_i's Data Format Converter

The parameters for the load stage register synthesis of the DFC are given by

$$\begin{aligned}
\triangle t_{load}(x_j) &= (t_1 0 + t_2 j) - j = j \\
\triangle t_{load}(y_i) &= (1i + t_2 0) - i = 0. \\
P_{load}(x_j) &= o_1 0 + o_2 j = 0 \times 0 + 1j = j \ (\mathrm{mod} exu) \\
P_{load}(y_i) &= o_1 i + o_2 0 = 0i + 1 \times 0 = 0.
\end{aligned}$$

The parameters for the circulation stage register synthesis of the DFC are:

$$\begin{aligned}
\triangle t_{circ}(x_j) &= (t_1 1 + t_2 j) - (t_1 0 + t_2 j) = t_1 = 1 \\
\triangle t_{circ}(y_i) &= (t_1 i + t_2 1) - (t_1 i + t_2 0) = t_2 \\
\triangle P_{circ}(x_j) &= (o_1 1 + o_2 j) - (o_1 0 + o_2 j) = o_1 = 0 \\
\triangle P_{circ}(y_i) &= (o_1 i + o_2 1) - (o_1 i + o_2 0) = o_2 = 1.
\end{aligned}$$

These parameters generate the DFC automatically. The DFC for the two inputs are shown in Fig. 4.4 and Fig.4.5.

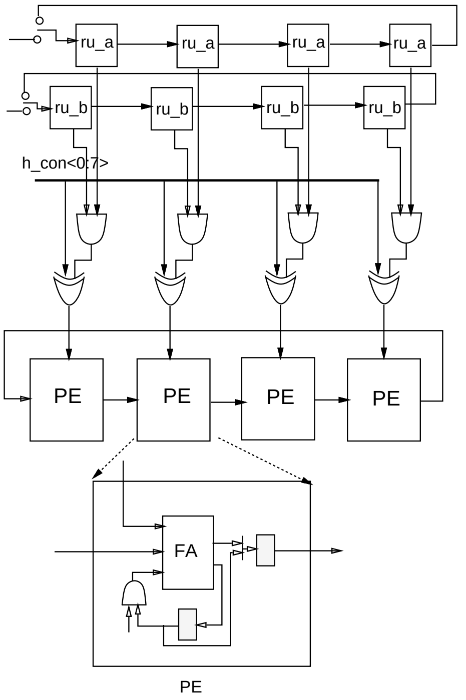


Fig. 4.6. Final design of 8 bit two's complement multiplier

Final Circuit

A practical multiplier needs to perform signed multiplications. The multiplier is developed here without sign consideration because the exceptional irregularity of the algorithm is avoided in order to enjoy the highly regular structure of a multiplication algorithm. However, it turns out that this structure can be kept intact by using the Hatamian algorithm for two's complement multiplication [59]. The sign modification with an algorithm leads to the original structure with the conditionally inverted terms and additions of two 1's. It is realized by slightly modifying the control circuit. Fig.4.6 shows the final design of an 8-bits two's complement multiplier.

Comparisons between the 16×16-bits multipliers of Lyon [59], Smith [70], Bi [71] and the proposed one are also summarized in Tab.I in terms of efficiency, performance and structural regularity. Here n is used to express the bit number. The real time means that the computation and data fetch procedures continuously performed in the form of two parallel processing pipelines at the same speed. When the computation cycle is 50ns, the data fetch speed become one fetch in every 50ns. Parallel efficiency means the processor cell utilization efficiency of one multiply execution, and it can be expressed as:

$$\frac{Necessary\ total\ computation\ time\ by\ one\ PE}{(Practical\ computation\ time\ by\ exu\ PEs) \times (exu)}.$$

Table I Performance Comparisons

	Lyon	Smith	Bi	Proposed
chip area per stage	64	36	160	62
No. of stages	n	n	n	$\frac{n}{2}$
No. of cycles per product	n	2n	n	2n
Latency	2n+1	2	2n+4	2n
Parallel efficiency	0.348	0.5	0.348	0.696
Computation time	46	32	46	46
Real time	yes	no	yes	yes
Structure regularity	systolic	semi systolic	systolic	semi systolic

Table II Performance Comparisons

PE (Exu)	Time (Unit time)	Area (Flip-flop's)	Area-Time (Cost-performance)
2	136	87	11832
4	76	93	7068
8	46	105	4830
16	46	190	8740

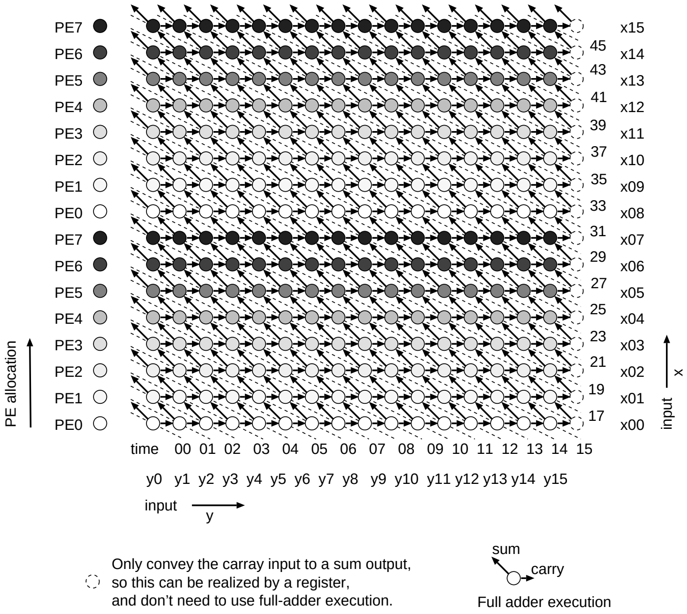


Fig. 4.7. Execution Ordering and Resource Allocation of the Multiplier

Chip area is estimated in terms of the number of dual input AND gates under the assumption that each gate occupies a unit area. It is estimated that a delay element and a full adder occupy a chip area equivalent to 6 and 8 unit areas, respectively. The result shows that the proposed design is effective from viewpoint of processor cell utilization efficiency.

Tab.II shows the MSPA design results for 16 × 16 bits multipliers with different number of PEs. According to the bit number of the multiplier, the effective *exu* candidates denoted as *PE* should be 2, 4, 8, or 16. The Area stands for the count of flip-flops included in the MSPA, where we assume that the area of one PE is equivalent

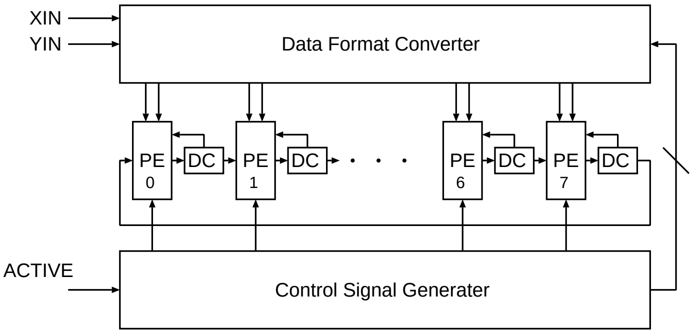


Fig. 4.8. Final design of 16 bit two's complement multiplier

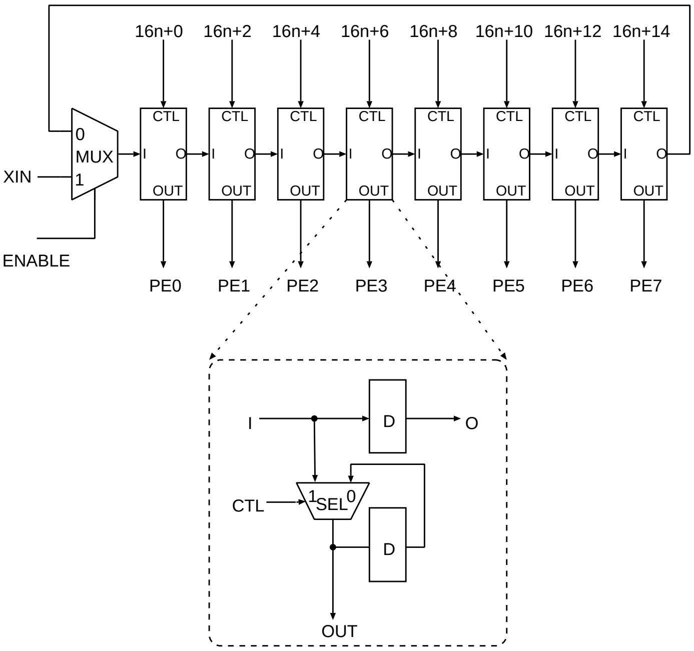


Fig. 4.9. The x_j's Data Format Converter

to the gate size of 4 flip-flops. By estimating the cost-performance with the product of Area and Time for different *exu* values, we get the optimal design parameter $PE = 8$ with $T = (1, 2)$, $O = (0, 1)$. Fig.4.7 shows the execution ordering and the resource allocation. This schedule achieves the total execution within 46 steps.

According to the Tab.II, the cost-performance of the 16-bit multiplier designed with MSPA architecture is nearly 1.8 times as large as that of the systolic array as described in the last line. Fig.4.8 shows the final design of 16-bits two's complement multiplier. Since the life-time analysis of the input variable data $x(0, j)$ and $y(i, 0)$, we obtain the largest life-time overlapping number of 16. So we can reduce the number of registers to the least 16 in the data converter for $x(0, j)$ and $y(i, 0)$ as shown in

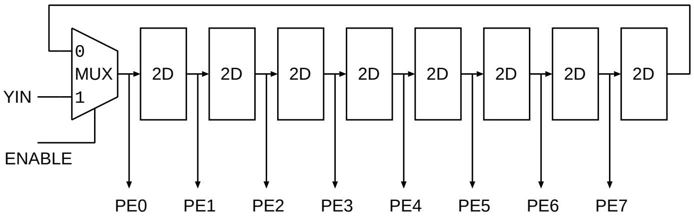


Fig. 4.10. The y_i's Data Format Converter

Fig.4.9 and Fig.4.10 respectively.

In this chapter, we present the necessity of designing data format converters (DFCs) which are derived systematically with a minimum number of registers and simplified controllers. As a synthesis example, we present a new bit-serial multiplier with higher performance than conventional bit-serial multipliers, such as systolic array. The typical feature of our multiplier is to select the number of the processor cells efficiently. We can select any number of full adders by the same systematic design procedure without increasing control scheme complexity. If we adopt the same number of full adders as the number of data bits, we can derive the conventional systolic multiplier. Our bit-serial multiplier selects full adders of half the number of bits based on the cost-performance analysis. As a result, the multiplier with MSPA architecture is shown to be attractive in terms of real-time operation, flexible function and structural regularity, small chip area and high parallel processing efficiency.

Chapter 5

MSPA Design for DSP Applications

In this chapter, we present practical MSPA designs for DSP applications. At first, LSI design for matrix-matrix multiplier is presented. In order to overcome I/O bottleneck and data storage problems and to achieve high cost effective [72], [73], we configurate the MSPA LSI with 8 PEs in a linear array. It accepts input data sequentially and converts them to parallel data by both registers based DFC theory and data memory based on memory management theory. Each PE performs multiply-add operations and sends its results to adjacent PEs. Our simple and explicit design for MSPA architecture allows us to realize MSPA, which can perform matrix multiplication of variable matrix size.

In the second section of this chapter, we present programmable MSPA LSI design. It is designed to apply various different DSP applications, which usually consists of regular algorithms. Those algorithms can be processed in MSPA of fixed PE number by changing interprocessor communication schemes. In other words, MSPA design procedure itself can be embedded onto chip, which can adjust controller for different kinds of DSP algorithms.

At last, we show some MSPA LSI design results for other applications.

5.1 Matrix Multiplication LSI

The algorithm of matrix multiplication can be described as

```
for $i = 0$ to $U - 1$ step 1
    for $j = 0$ to $U - 1$ step 1
        for $k = 0$ to $U - 1$ step 1
        $c(i, j, .k) = f_c(c(i, j, k - 1), a(i, 0, k), b(0, j, k))$
        next
    next
next
```

where c(i, j, -1)=0 .
The data dependence is shown as Fig.5.1 For this kind of data dependence of the algorithm, the successive data transfer scheme is available based on our resource allocation theory.

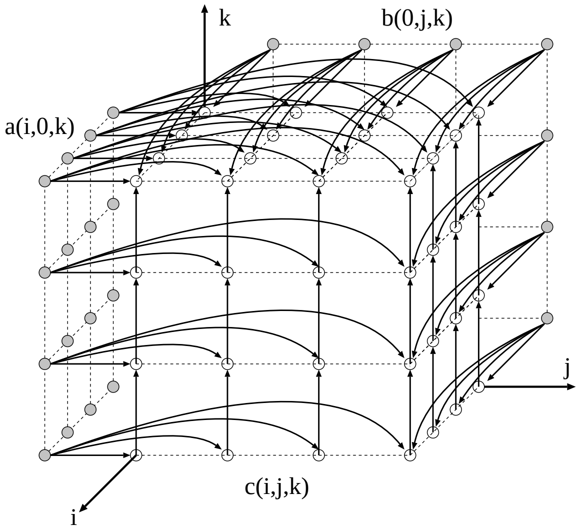


Fig. 5.1. Data dependence for matrix multiplication

5.1.1 Basic Design Parameters

The number of matrix dimension U is changeable from 9 to 16, and the processor element (PE) number exu is fixed to 8. For this application, we obtain operation scheduling design vector and resource allocation design matrix as:

$$\begin{aligned} &T = (t_1, t_2, t_3) \\ &O = (0, 1, U) \bmod 8 \\ &M_O = 8, \end{aligned}$$

where

$$\begin{aligned} &t_1 = 1 \\ &t_2 = 2 \\ &t_3 = 3U - 16. \end{aligned}$$

The design procedure for this ASIC MSPA LSI is referenced with the Fig.2.2.

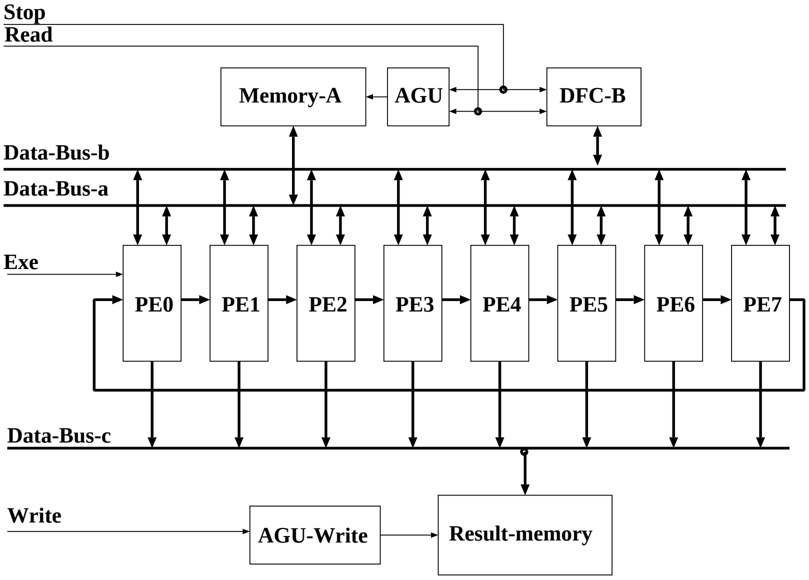


Fig. 5.2. MSPA Architecture For Matrix Multiplication

5.1.2 Architecture

Fig.5.2 describes the schematic architecture of the MSPA used for matrix multiplication. It consists of a processor array with direct links and buses associated with input data format conversion registers, data memory and I/O ports. There are 2 input ports (Data-Bus-a and Data-Bus-b) and one output port (Data-Bus-c). The data of some bandwidth come in a serial scheme and are loaded directly to PEs or data format conversion registers or data memory. The registers hold and reuse input data and act as a data format converter (DFC) for the run-time input scheme. The data memory is used to preload compile-time input data in serial and deliver them in parallel.

5.1.3 Data Format Converter

The DFC configuration relies on the input scheduling of input variables, and the input scheduling is constrained by the data dependence of application to some extent. Assume that the two input matrices are denoted as $A\{a(i_2, i_1)\}, B\{b(i_1, i_3)\}$ and the output matrix is denoted by $C\{c(i_2, i_3)\}$, where $i_1, i_2, i_3 = 0, 1, 2, \cdots, U-1$ respectively, and the U denotes the dimension of matrix multiplication. Then, it is reasonable that input $b(i_1, i_3)$ is transfered to PEs in a column order with the column length of U^*. According to this input scheme, the input time for input $b(i_1, i_3)$ can be expressed as $t_{in_b} = i_1 + U^* i_3$, where the U^* must satisfy $U^* \geq U$. Because the load time is determined by the execution operation schedule, i.e. $t_{load_b} = Tp$. Therefore, we get $t_{in_b} < t_{load_b}$. According to the *Necessary condition for available register DFC* in last chapter, it is obvious that we can deal with data format conversion problem for input $b(i_1, i_3)$ by DFC approach. Based on the input scheme of input $b(i_1, i_3)$ and the data dependence of matrix multiplication, it is easy to prove that input $a(i_2, i_1)$ does not satisfy the *Necessary condition for available register DFC.* Therefore, we use data memory to resolve data format conversion problem for input $a(i_2, i_1)$.

Followings describe the data format converter design for input $b(i_1, i_3)$.

Data Load

Since the data $b(i_1, i_3)$ is used for the operation of (i_1, i_2, i_3), and is loaded only when $i_2 = 0$, we get:

$$\triangle t_{load-in_b} = (t_3 - U^*)i_3. \tag{5.1}$$

The load position P_{load} for the variable $b(i_1, i_3)$ is expressed as

$$\begin{aligned} P_{load_b} &= o_1 i_1 + o_3 i_3 \\ &= U \times i_3 (\mathrm{mod} exu). \end{aligned} \tag{5.2}$$

The above result illustrates that the $(i_1 + U^* i_3)$-th data of the input variable $b(i_1, i_3)$ is loaded to $(U \times i_3 \bmod exu)$-th DFCU after $(t_3 - U^*)i_3$ cycles of data-serial input cycle start. This scheme can be realized by the cascade connection of the first stage registers in every $(U \times i_3 \bmod exu)$ register units, which consists of $(t_3 - U^*)i_3$ registers.

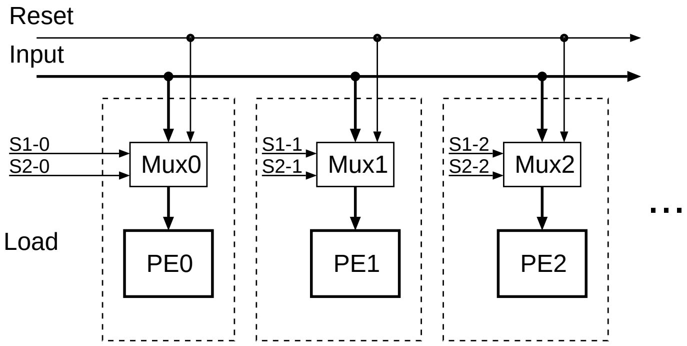


Fig. 5.3. The First Stage Register $\triangle t_{load-in} = 0, P_{load} = i_3$

Fig.5.3(a) shows the loading schemes of $b(i_1, i_3)$ in U=9 case with $U^* = t_3$. The result shows that we can reduce the number of registers for input loading with $U^* = t_3$ for input $b(i_1, i_3)$. It's clear that, when $U^* = t_3$, the data load problem of $b(i_1, i_3)$ can be dealt directly without employing registers.

Data Circulation

After finishing the allocation of first stage register for data load, we design the second stage register for data circulation. This stage resolves how to distribute the input variable $b(i_1, i_3)$ through the register units so that they can be reused by the processor cell array.

Input $b(i_1, i_3)$, will be reused when $i_2 > 0$. Therefore, the loaded data $b(i_1, i_3)$ is transferred to the register unit of

$$\triangle P_{circ_b} = O\Delta i_2 = o_2 \times 1 = 1. \tag{5.3}$$

For position rightward from the current register unit position, the transfer time should be

$$\triangle t_{circ_b} = T\Delta i_2 = t_2 \times 1 = t_2. \tag{5.4}$$

The lifetime of every loaded data for input $b(i_1, i_3)$ will be $t_2 \times (U - 1)$ because the reuse takes place when every i_2 changes. The data usage is finished after (U-1) times rightward circulation among the second stage registers.

Fig.5.4 shows the circulation schemes for $b(i_1, i_3)$. Based on the design methodology for DFC, we input data in a stream length of $U^* = t_3$ for every column. That is to say, the next column data has to wait until the load of current column lasts for enough U^* load periods. Between the period from U to U^*, we can input nothing to the I/O port. The block denoted by DFC-b as shown in Fig.5.5 represents the architecture of DFC for the input variables $b(i_1, i_3)$.

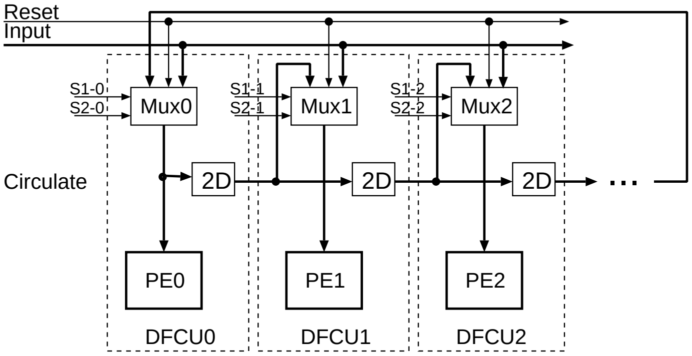


Fig. 5.4. The Second Stage Register
$\triangle t_{circ} = 2, \triangle P_{circ} = 1$

The DFC consists of *exu* DFCUs, and the positions of the DFCUs are described in the space coordinate as $0, 1, 2, \cdots, exu - 1$, which correspond to the positions of the corresponding processor cell.

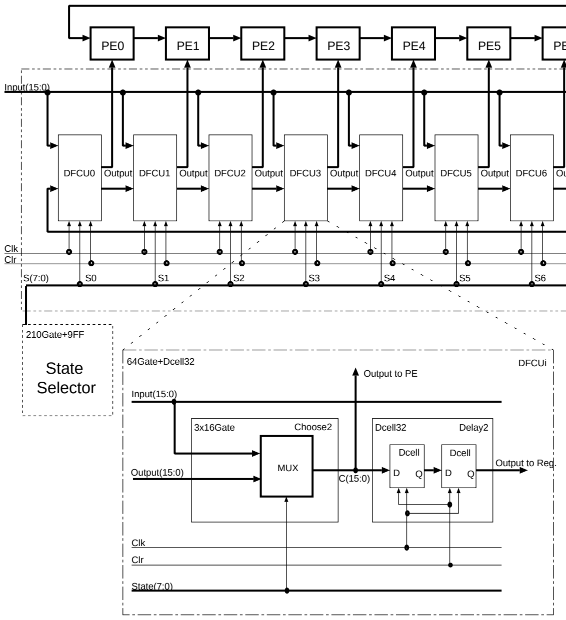


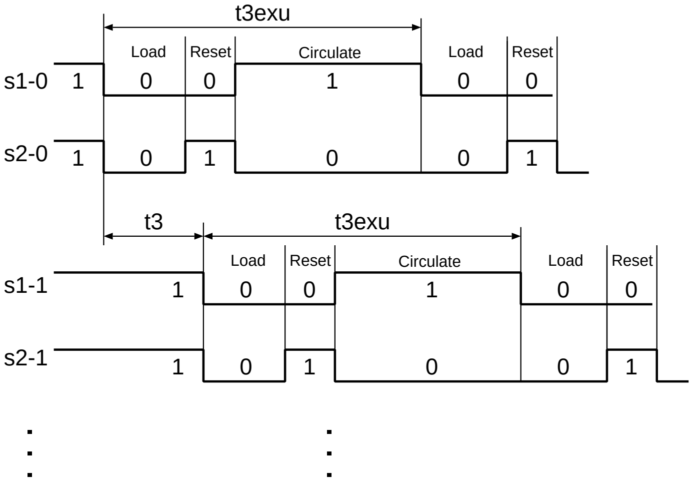
Fig. 5.5. Architecture of DFC

DFC Control Logic

Here, we explain the principle to design the select control logic for input $b(i_1, i_3)$. Control signals s_{1-j} and s_{2-j}, $(j \in \{0, \cdots, (exu-1)\})$ are loaded to the control terminals of multiplexers Muxs as shown in Fig.5.3 and Fig.5.4. Based on the analysis of data load, the number of load stage registers can be reduced with an optimal input schedule $U^* = t_3$, which will make the number of the first stage registers become 0. In this case, data can be loaded to PEs directly without using any latch registers. Because the PE address of load process will change according to column of the input matrix, an input load control signal for each DFCU must be designed without overlapping with each other. In order to end the data circulation of an old column and start the circulation procedure of a new input column, a reset signal is introduced to clear registers. In addition to the load and reset signals, the register also goes into a circulation state. These signals can be generated by using simple state machines. Fig.5.6 shows the control codes and their timing charts of the Muxs control signals. The period of each control signal for every DFCU should be $t_3 exu$, where the exu represents the number of PE.

Reconfigurability of DFC Theory

According to the result of DFC design, the hardware structure of a DFC will rely on the parameter t_2. If t_2 is fixed, the DFC architecture is also fixed. This is because



s1-i s2-i (i=0,...,7)	State
00	Load
01	Reset
10	Circulation

Fig. 5.6. DFC State Control Codes and Timing Charts

the same scheduling vector T with the algorithm independent resource allocation matrix O and the PE number exu determine the DFC architecture. However, this fixed DFC architecture can also be used to deal with variable dimension algorithms and their reconfigurable control logic if the change of dimension does not result in a change of t_2. It is obvious the control signal will change according to the matrix dimension due to its relation with t_3. Taking U as a parameter, the load-circulate select control signal can be designed automatically to vary with the value of U . By use of this reconfigurable control logic, rigid DFC can be developed for variable dimension algorithm.

Furthermore, when t_2 is taken as a parameter, the DFC architecture itself becomes reconfigurable too. In this case, the dimension of the algorithm is allowed to change freely. Therefore, the DFC synthesized by the new methodology can be developed to a programmable/configurable architecture. We have extended a ASIC synthesis software for the programmable/configurable DFC for uniform recurrence algorithms.

5.1.4 Memory Design for Input $a(i_2, i_1)$

The following introduces some considerations about memory management of MSPA in the design example of matrix multiplier. As mentioned in previous section, we deal with the data format conversion problem for input $a(i_2, i_1)$ by memory.

I/O bandwidth constraint and memory size

The large I/O bandwidth becomes a crucial problem for a number of real-time applications in image or signal processing, which will either lead to very expensive chip realizations or imposes an upper bound on the algorithm size. One method to resolve the I/O constraint problem is to introduce on-chip memory to change the data transfer rate of I/O and processing. Because memory size should be a function of I/O rate, mapping of the scheduling and resource allocation of execution onto the processor elements, algorithm size, input and output wordlengths, it's possible to minimize the on-chip memory size by polishing the whole design procedure.

For matrix-product like application, a set of data, used repeatedly during the process, reminds in a chip instead of being input every time when it is required. According to this characteristics, the most simple way to decide the memory size roughly is to select U^2, where the U stands for the matrix size. Then, we use an address generation unit to generate the addresses required by each PEs to adjust data streams.

Fig.5.7 shows an example to use memories to change data transfer rates. Assume that the input rate of input variable In_a is nw_{in} per clock cycle (PCC), the w_{in} denotes the wordlength of the input data, while the input rate of input variable In_b is w_{in} PCC. After the data conversion by memories, the data must be delivered to the PEs in a deliver rate of $exu \times w_{in}$ PCC to achieve a processing rate of $exu \times w_{in}$ PCC.

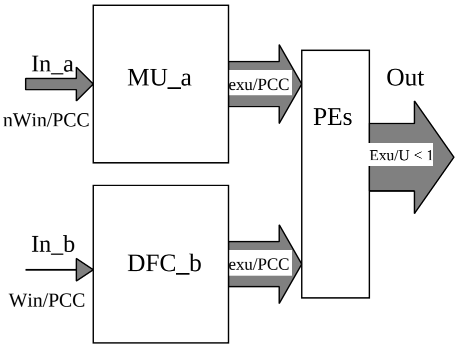


Fig. 5.7. Data transfer rate

Memory size is possible to be minimized corresponding to a given input rate. Suppose the input data denoted as $a(i_2, i_1)$ is input in a rate of one data PCC, i.e. input rate is 16 bits PCC, then the deliver rates of each new data from memory to PEs are:

$$a(i_2, i_1) : \quad 8\times \ new\ data\ = 128bits\ PCC$$

The lifetime of each new data of input $a(i_2, i_1)$ is given by:

$$lifetime = (U-1) \times t_3 + 1.$$

It is seen that even for the case of least algorithm size, U=9, the lifetime already exceeds the size of U^2. This is the reason why we choose the memory size based on the algorithm size, $U^2 \times 16bits$. The memory size can be reduced more according to following considerations.

(1) Calculate the time difference between total execution time and the time of preloading the whole input matrix so that the execution starts before finishing the preload procedure

(2) Increase the input rate up to several times as high as the processing rate.

As an example, we consider only the case when U=9. According to the schedule vector T, the total execution time becomes 113 clock cycle (CC). The input data $a(i_2, i_1)$ should be preloaded in advance and the total preload time is usually given by $U^2 = 81$. Because the PEs require the first data deliver on the last column of $a(i_2, i_1)$ at the time

$$T_{last} = t_2[U-1] = 16 \ ,$$

and the column length of data $a(i_2, i_1)$ is U, we get the minimized preload time

$$preload - time = (U^2 - U - T_{last})CC = 56CC.$$

If we increase the input rate in a sense of increasing the input speed to n times, the preload time becomes less than:

$$preload - time = (U^2 - U - n \times T_{last})CC$$

For example, when n=3, the preload tome will be less than 24CC.

Address generation

In our memory management method, Address Generation Unit consists of an execution index points generator and a converter. The converter is used to convert the index points into the read source addresses (RSA) for each memory unit and calculate the read destination addresses (RDA). Each memory requires one converter, while each converter consists of many register units. The number of the register units should be the same as the number of the processor elements because the processor arrays need their own addresses for parallel processing. All of these converters and register units share a common execution index points generator.

General AGU design is described in the third chapter. We let $R_k = (i_1, i_2, i_3)(k = 0, 1, \cdots, exu-1)$ be the k-th AGU register unit storing the current RSA and RDA. There are exu register units $R_0, R_1, \cdots, R_{exu-1}$ respectively to store RSAs and RDAs

Table I Data stream for input $a(i_2, i_1)(U = 9)$

time	$PE0$	$PE1$	$PE2$	$PE3$	$PE4$	$PE5$	$PE6$	$PE7$
0	**a00**							
1	**a01**							
2	**a02**	**a10**						
3	**a03**	**a11**						
4	**a04**	**a12**	**a20**					
5	**a05**	**a13**	**a21**					
6	**a06**	**a14**	**a22**	**a30**				
7	**a07**	**a15**	**a23**	**a31**				
8	**a08**	**a16**	**a24**	**a32**	**a40**			
9	.	**a17**	**a25**	**a33**	**a41**			
10	.	**a18**	**a26**	**a34**	**a42**	**a50**		
11	.	a00	**a27**	**a35**	**a43**	**a51**		
12	.	a01	**a28**	**a36**	**a44**	**a52**	**a60**	
13	.	a02	a10	**a37**	**a45**	**a53**	**a61**	
14	.	a03	a11	**a38**	**a46**	**a54**	**a62**	**a70**
15	.	a04	a12	a20	**a47**	**a55**	**a63**	**a71**
16	**a80**	a05	a13	a21	**a48**	**a56**	**a64**	**a72**
17	**a81**	a06	a14	a22	a30	**a57**	**a65**	**a73**
18	**a82**	a07	a15	a23	a31	**a58**	**a66**	**a74**
19	**a83**	a08	a16	a24	a32	a40	**a67**	**a75**
20	**a84**	.	a17	a25	a33	a41	**a68**	**a76**
21	**a85**	.	a18	a26	a34	a42	a50	**a77**
22	**a86**	.	a00	a27	a35	a43	a51	**a78**
23	**a87**	.	a01	a28	a36	a44	a52	a60
24	**a88**	.	a02	a10	a37	a45	a53	a61
25	a70	.	a03	a11	a38	a46	a54	a62
26	a71	.	a04	a12	a20	a47	a55	a63
27	a72	a80	a05	a13	a21	a48	a56	a64
28	a73	a81	a06	a14	a22	a30	a57	a65
29	a74	a82	a07	a15	a23	a31	a58	a66
30	a75	a83	a08	a16	a24	a32	a40	a67
31	a76	a84	.	a17	a25	a33	a41	a68
32	b77	a85	.	a18	a26	a34	a42	a50
33	a78	a86	.	a00	a27	a35	a43	a51
34	a60	a87	.	a01	a28	a36	a44	a52
35	a61	a88	.	a02	a10	a37	a45	a53
36	a62	a70	.	a03	a11	a38	a46	a54
37	a63	a71	.	a04	a12	a20	a47	a55
38	a64	a72	a80	a05	a13	a21	a48	a56
39	a65	a73	a81	a06	a14	a22	a30	a57
40	a66	a74	a81	a07	a15	a23	a31	a58
41	a67	a75	a82	a08	a16	a24	a32	a40
42	a68	a76	a83	.	a17	a25	a33	a41
43	a50	a77	a84	.	a18	a26	a34	a42
44	a51	a78	a85	.	a00	a27	a35	a43
45	a52	a63	a86	.	a01	a28	a36	a44

Table II Data stream for input $b(i_1, i_3)$

time	$PE0$	$PE1$	$PE2$	$PE3$	$PE4$	$PE5$	$PE6$	$PE7$
0	b00							
1	b10							
2	b20	b00						
3	b30	b10						
4	b40	b20	b00					
5	b50	b30	b10					
6	b60	b40	b20	b00				
7	b70	b50	b30	b10				
8	b80	b60	b40	b20	b00			
9	.	b70	b50	b30	b10			
10	.	b80	b60	b40	b20	b00		
11	.	b01	b70	b50	b30	b10		
12	.	b11	b80	b60	b40	b20	b00	
13	.	b21	b01	b70	b50	b30	b10	
14	.	b31	b11	b80	b60	b40	b20	b00
15	.	b41	b21	b01	b70	b50	b30	b10
16	b00	b51	b31	b11	b80	b60	b40	b20
17	b10	b61	b41	b21	b01	b70	b50	b30
18	b20	b71	b51	b31	b11	b80	b60	b40
19	b30	b81	b61	b41	b21	b01	b70	b50
20	b40	.	b71	b51	b31	b11	b80	b60
21	b50	.	b81	b61	b41	b21	b01	b70
22	b60	.	b02	b71	b51	b31	b11	b80
23	b70	.	b12	b81	b61	b41	b21	b01
24	b80	.	b22	b02	b71	b51	b31	b11
25	b01	.	b32	b12	b81	b61	b41	b21
26	b11	.	b42	b22	b02	b71	b51	b31
27	b21	b01	b52	b32	b12	b81	b61	b41
28	b31	b11	b62	b42	b22	b02	b71	b51
29	b41	b21	b72	b52	b32	b12	b81	b61
30	b51	b31	b82	b62	b42	b22	b02	b71
31	b61	b41	.	b72	b52	b32	b12	b81
32	b71	b51	.	b82	b62	b42	b22	b02
33	b81	b61	.	b03	b72	b52	b32	b12
34	b02	b71	.	b13	b82	b62	b42	b22
35	b12	b81	.	b23	b03	b72	b52	b32
36	b22	b02	.	b33	b13	b82	b62	b42
37	b32	b12	.	b43	b23	b03	b72	b52
38	b42	b22	b02	b53	b33	b13	b82	b62
39	b52	b32	b12	b63	b43	b23	b03	b72
40	b62	b42	b22	b73	b53	b33	b13	b82
41	b72	b52	b32	b83	b63	b43	b23	b03
42	b82	b62	b42	.	b73	b53	b33	b13
43	b03	b72	b52	.	b83	b63	b43	b23
44	b13	b82	b62	.	b04	b73	b53	b33
45	b23	b03	b72	.	b14	b83	b63	b43

Table III Comparison with GPM and DM

U	Items	T_{comp} GPM	T_{total} GPM	PE GPM	LK's DM	SF's DM	**MSPA**
8	Pre. time	43	22	23			1
	Comp.time	50	50	71			134
	Sum time	93	72	94			135
	PE	22	29	8			4
	Time-area	**2046**	**2088**	**752**			**540**
	Comp. effi.	**47%**	**35%**	**90%**			**96%**
	Total effi.	**25%**	**25%**	**68%**			**95%**
15	Pre. time				43	379	98
	Comp.time				239	225	449
	Sum time				282	604	547
	PE				43	43	8
	Time-area				**12126**	**25972**	**4376**
	Comp.effi.				**33%**	**35%**	**94%**
	Total effi.				**28%**	**13%**	**77%**
16	Pre. time	69	69	87			129
	Comp.time	121	121	271			526
	Sum time	190	190	358			655
	PE	76	76	16			8
	Time-area	**14440**	**14440**	**5728**			**5240**
	Comp. effi.	**45%**	**45%**	**94%**			**97%**
	Total effi.	**28%**	**28%**	**72%**			**78%**

required by exu PEs at each time. In order to simplify the memory RSA and RDA generation procedure, we fix the destination address RDA of each R_k by making the number of $R_k(k = 0, 1, \cdots, exu - 1)$ corresponding to the number of $PE_k(k = 0, 1, \cdots, exu - 1)$.

The data streams generated by the AGU for two matrix inputs (U=9) $a(i_1, i_3)$ and $b(i_3, i_2)$ are shown in Tab.I and Tab.II [74].

5.1.5 Design Comparison

Comparisons with other methods, such as, General Parameter Method (GPM) [75] [76], Dependence Method (DM) either by Lee and Kedem (LK) or by Shang and Fortes (SF), are summarized in Tab.III in terms of preload time ($Pre.\ time$), total computation time ($Comp.\ time$), sum of the preload and computation time ($Sum\ time$), the number of PEs (PE), time-area efficiency ($Time - area$), and parallel-efficiency respectively. The preload time is defined as the input time duration before the computation starts. Here, we assume the data transfer speed from input port to memory is 4 times of computation speed. The time-area performance is evaluated by the product of the $Sum\ time$ and the number of PEs. The computation parallel-efficiency ($Comp.\ effi$) and total parallel-efficiency ($Total\ effi$) are derived

Table IV Comparison with GPM

U	GPM Systolic Array				Proposed MSPA			
	T_{exe}	exu	$T_{exu} \cdot exu$	$Para.Effi.$	T_{exe}	exu	$T_{exu} \cdot exu$	$Para.Effi.$
4	19	10	190	34	34	2	68	94
5	29	13	377	33	49	3	147	85
10	109	28	3052	33	208	5	1040	96
17	305	49	14925	33	577	9	5193	95
53	1717	261	448137	33	8425	18	151650	98
100	5644	496	2799424	36	20098	50	1004900	100
201	15001	1401	21016401	39	80801	101	8160901	100

according to *Comp. time* and *Sum time*, respectively. The U indicates the matrix dimension. T_{comp} *GPM* and T_{total} *GPM* indicate the computing-time-optimal and completion-time-optimal GPMs, respectively, and the *PE GPM* indicates the PE-optimal GPM. In Tab.III, the *Comp. time* in our method shows larger values than others, while the PE shows very practical values. It is obvious that the proposed methodology achieves higher parallel efficiency. Moreover, the DFC with very limited register number is derived. A full custom IC design was performed for this matrix multiplier, in which a 16-bit fixed point multiplier and adder are included in each PE. The DFC including its control logic has been designed with same large area as that of one PE.

Tab.IV shows synthesis result mainly in terms of execution time denoted by T_{exe}, the number of PEs (exu), the product of T_{exu} and exu, and parallel efficiency denoted by $Para.Effi.$. The T_{exe} term in the proposed method shows larger values than the GPM, while the exu term shows very practical values. If the cost-performance of the design is evaluated by the value of $T_{exe} \cdot exu$, the proposed method achieves higher cost performance. It is obvious that the design results by proposed method achieve higher parallel efficiency.

5.2 Programmable MSPA

The regular structure and systematic design procedure of MSPA design methodology make it possible to realize programmable and reconfigurable MSPA. A controller adjusts operation scheduling, allocates resources and manipulates data transfer between I/O interface and processing elements (PEs) for specified application algorithms. It also controls memory address generation. Thus, we can design programmable MSPA for various applications without losing fast processing advantages of original ASIC-MSPA.

Fig.5.8 describes the programmable MSPA architecture for wide range of applications. It consists of a processor array with direct links and buses associated with

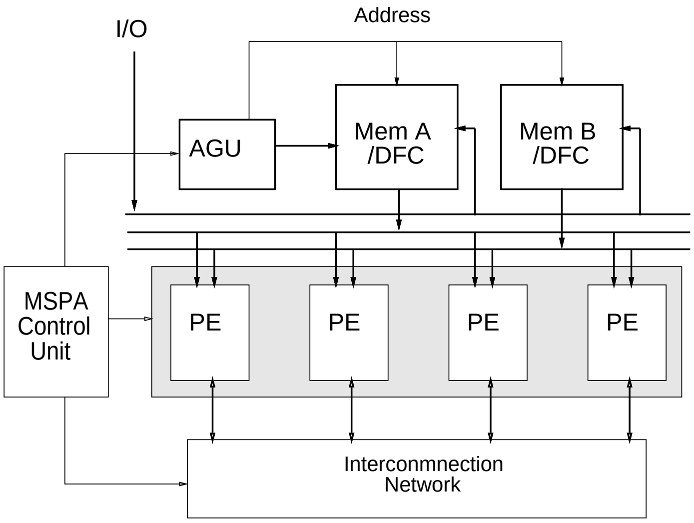


Fig. 5.8. Programmable MSPA Architecture

memory units (MU) or data format converter (DFC) and I/O ports. There is only one I/O port for the input of each variable. The data of a certain bandwidth comes in a word-serial scheme or a bit-serial scheme and is loaded directly to PEs or memory/DFC. The memory units store and load input data to processor array according to the control signal generated by the address generation unit (AGU).

The type of the processor array is controlled by interconnection network, which also covers the loosely coupled linear or 2D processor array, so that the data transfers between processors are accomplished by local direct communications. Direct bus communications or bus communications via memory/DFC units are also available in this MSPA architecture.

5.2.1 Design Specification

The number of processing elements (PEs) denoted by exu is fixed to 4 in the MSPA. The function of the PEs is multiply-accumulation. The size of algorithms is denoted as U. In order to simplify the design, we assume that the U and exu keep the relation of integer multiples [77], and U is constrained to be 4, 8, 12, 16. The outline of MSPA control unit is shown in Fig.5.9.

The programmable MSPA is originally designed not only MSPA processor, but general-purpose DSP. There are two data memories, one program memory in the chip. The MSPA works as a special work mode of the general-purpose DSP. In MSPA-mode, data will be input from two data memories in the input-write cycle of 4 times as high as execution cycle. Memory read cycle will be synchronized by execution cycle. The data memories are one-write, one-read with free read and write function.

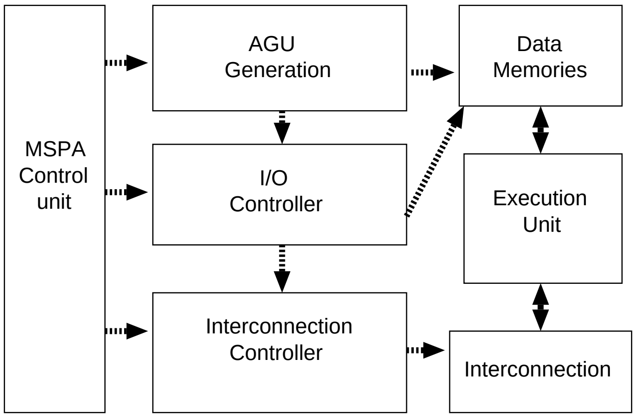


Fig. 5.9. Design Outline of Programmable MSPA

5.2.2 Data-path Design

The data-path in MSPA-mode is shown in Fig.5.10. There are at least four data registers for keeping input data from each data memory. The pipeline schedule is shown in Fig.5.11. The start time of the 3 pipelines is decided by the MSPA start instruction and enable instruction. When instruction "MSPA start" is decoded, the first input-write pipeline starts data load procedure. After one clock, the read pipeline starts driven by the write start signal. After the instruction "MSPA parameter-set", the MSPA finish reconfiguration procedure and fix the hardware mode. The execution stage begins soon after the instruction "MSPA enable" comes. The active time of output write pipeline will be determined by the *Out-write decode* logic which is reconfigurated by the *MSPA parameter-set* instruction.

Interconnection Design

Local communication between PEs can be employed corresponding to different algorithm size or operation schedule. Then, the reconfigurable MSPA architecture is realized in this simple programmable way. We design this part in schematic level by CADENCE CAD software in order to minimize the number of gates in this part. Fig.5.12 shows an example for the block design of reconfigurable interconnect network.

Execution Pipelining

MSPA utilizes instruction-level pipeline processing for the regular structured algorithms. Their pipeline fashion is in such a way that each execution unit repeatedly performs a part of processing for the target algorithm one after another.

The further speed-up can be achieved by employing the pipeline execution units in order to exploit execution-level pipeline processing. However, the pipeline executions

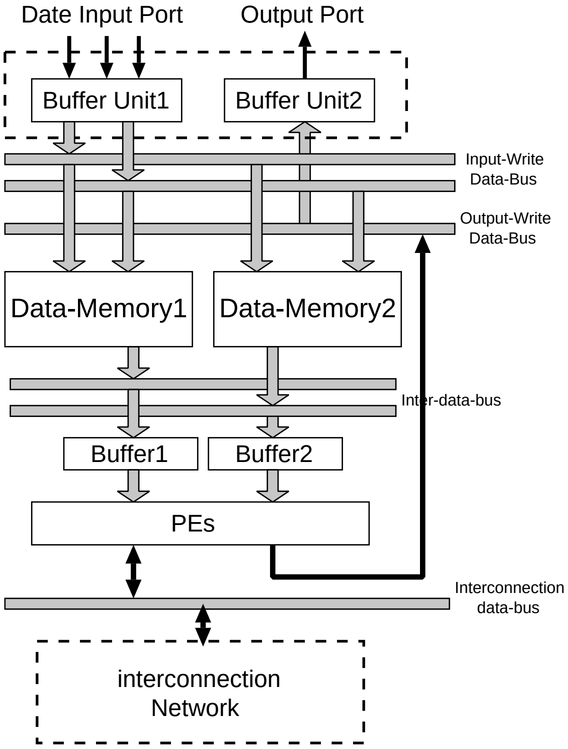


Fig. 5.10. Data-path in MSPA-model

requires plural independent operations, among which there are no precedence relation or data transfers. Fortunately, MSPA design matches with this extension. MSPA scheduling and resource allocation based on the specified PE number exploit the parallelism of the same number of operations as PE number. It can simply be modified by the design on PEs with pipeline execution units.

Let exu be the number of PEs and $stage$ be the number of pipeline stages of all execution units in PEs. We perform the scheduling of MSPA operations with $exu \times stage$ execution units. If the algorithm size U is still less than $exu \times stage$, MSPA design schedule the $exu \times stage$ operations to be invoked concurrently. While, the resource allocation is performed in the same way with exu. Thus, the $stage$ operations are assigned to each pipeline PEs.

Fig.5.13 shows $exu = 4$ and $stage = 4$ scheduling in MSPA design. Fig.5.14 shows its operation scheduling in each PE. As shown in Fig.5.13, each PE run $stage$ times as fast as no pipeline execution units, but requests $stage$ times as many data supply as no pipeline one. Thus, MSPA scheduling exploits independent operations so easily that it can be extended to achieve not only instruction pipeline, but execution pipeline operations.

In practical design, the clock rate of $5ns$ is achieved both for memory access and

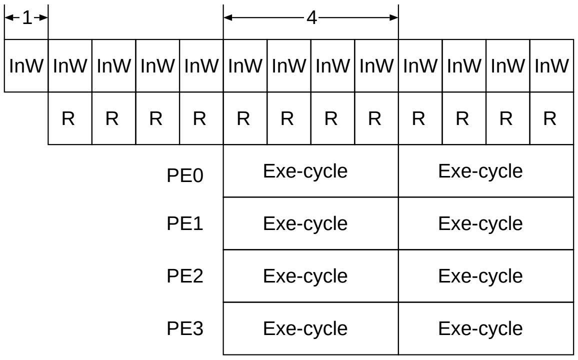


Fig. 5.11. Pipeline Schedule in MSPA-model

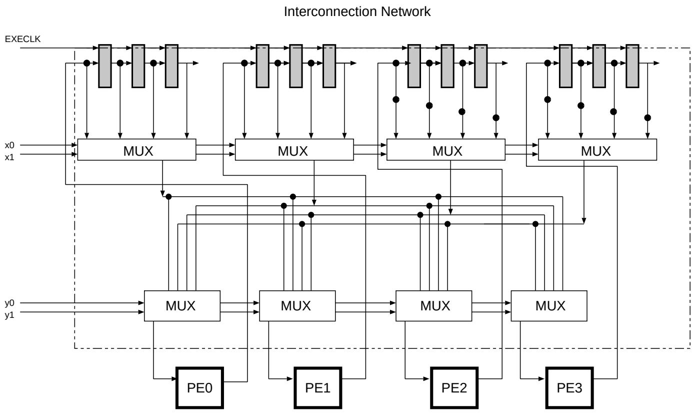


Fig. 5.12. Interconnection Network Architecture

4 stage pipeline operation of a multiply-add. Therefore, our current design expects that a chip including such 4 pipeline stage PEs can achieve 0.8G operations per a second. The simple design procedure of MSPA under the fixed number of PE is excellently embedded into micro-architecture. Fig.5.15 shows the controller layout of the proposed programmable MSPA based on the design described here. Its speed is expected to be 0.2 G multiply-add operation per second. Furthermore, the study for further extension to general purpose array processor including programmable MSPA function is being performed.

5.2.3 Controller Design

The controller in MSPA-mode is a programmable and configurable logic unit. Controller consists of AGU(Address Generation Unit) controller, run-time input con-

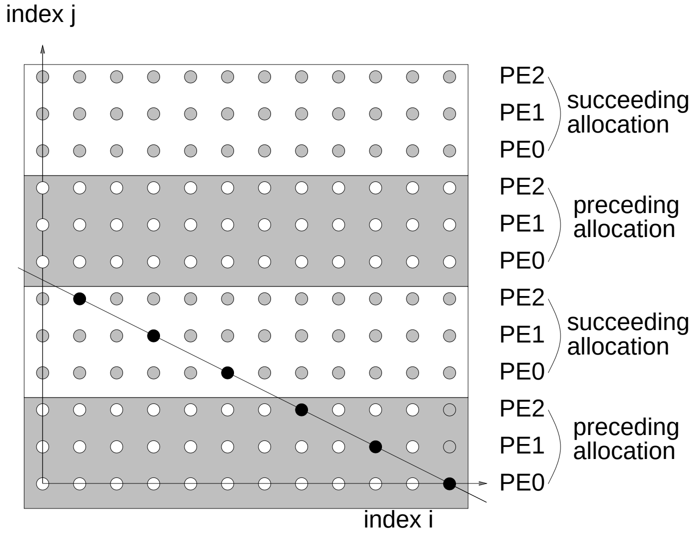


Fig. 5.13. MSPA with pipeline scheduling

troller, interconnection controller and out-write controller.

The *AGU controller* controls the data fetch to each PE. It generates exu invoking indecies and produces the actual addresses of Memory units. At the same time, it produces the destination PEs for each read data from Memory units. Then, the data is controlled to be transfered to appropriate PEs. The *Run-time input controller* starts input-write (input from I/O port and write to Memory unit) pipeline based on *MSPA start* instruction and generates input-write timing, read-timing, and execution timing signals. The *Interconnection controller* is used to reconfigurate the local communication path between PEs. The *Out-write controller* is used to calculate the start and interval time of output write pipeline of each PE.

In order to generate the concurrent invoked indecies, we prepare the exu index point registers $R_u = (R_{u1}, R_{u2}, R_{u3})$ for u-th PE ($u = 0, 1, \cdots exu - 1$), each subregister of which have a length of U. Each index point register R_u generates only one index point of the u-th, $u + exu$-th, $\cdots$ rows in the converted 2D index point space at a time, which are corresponding to the PE allocation. Therefore, each register generates active indecies for each PE.

The increment of time t usually makes an increment of i_1.

$$t = t_1U + t_2i_2 + t_3i_3 = t_10 + t_2(i_2 + exu) + t_3i_3 \tag{5.5}$$

holds, a carry produced by R_{u1} must increment i_2 by exu.

In case R_{u2} and R_{u1} reaches at U, R_{u2} and R_{u1} are reset and since

$$t = t_1U + t_2U + t_3i_3 = t_10 + t_20 + t_3(i_3 + 1) \tag{5.6}$$

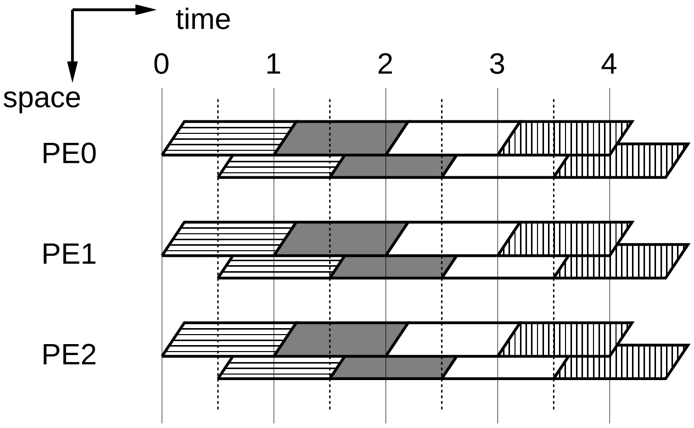


Fig. 5.14. Operation scheduling in each PE

holds, a carry produced by R_{u2} and R_{u1} must increment i_3 by 1.

The operation of the index point register is as follows.

1. R_{u1} increments +1 at every time steps and produces a carry when R_{u1} exceeds U.
2. R_{u2} increments $+exu$ by the carry of R_{u1}. and produces a carry when R_{u2} exceeds U.
3. R_{ui} for $i = 3, \cdots, n$ increments +1 by the carry of $R_{u(i-1)}$.

Run time input controller controller is also realized in similar fashion to AGU controller.

5.3 MSPA Matrix Vector Multiplier

We presented a MSPA LSI design for a 16 × 16 bit-serial matrix-vector multiplier. By the same method, data serial matrix vector multipliers are designed as shown in Tab.V. In order to demonstrate the flexibility of our MSPA architecture, we show the realization results for a matrix vector multiplication of different algorithm size U with fix number of PEs, $exu = 3 \times 2$. The scheduling vector T and the resource allocation O are derived as

$$T = (1, \lfloor \frac{U}{exu} \rfloor)$$
$$O = \begin{bmatrix} 0, & 1 \\ 1, & \lfloor \frac{U}{exu} \rfloor - 1 \end{bmatrix}.$$

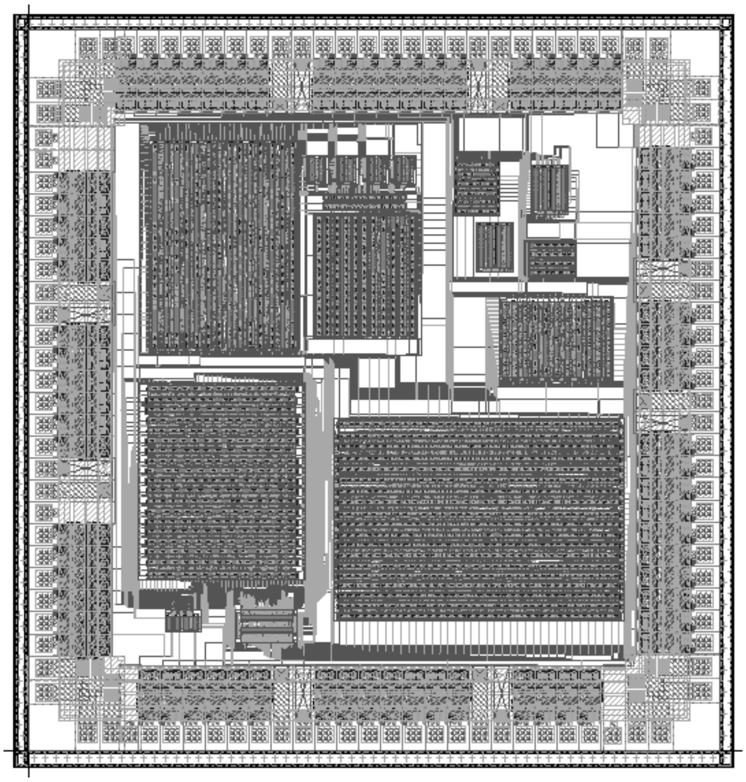


Fig. 5.15. Layout of the programmable MSPA Controller

The total execution time is obtained by

$$
\begin{aligned}
t_{total} &= t_1(U-1) + t_2(U-1) && (5.7)\\
&= (1 + \lfloor \frac{U}{exu} \rfloor)(U-1). && (5.8)
\end{aligned}
$$

Parallel Efficiency is derived by

$$Eff. = \frac{U \times U}{t_{total} exu} = \frac{U \times U}{(1 + \lfloor \frac{U}{exu} \rfloor)(U-1)exu}. \qquad (5.9)$$

From numerical design results as shown in Tab.V, it is seen that the larger the ratio $\frac{U}{exu}$ becomes, the closer to 1, 100% the parallel efficiency reaches. This high parallel efficiency can be achieved without employing additional hardware resources. It requires only an increase of the data memory capacity, which corresponds to algorithm size U. This aspect is one of the advantages of MSPA architectures. Furthermore, for this kind of data dependence of the algorithm, we can confirm that successive data transfer scheme is available based on our resource allocation theory.

Table V Design Result

U	Schedule Vector	Allocation Vector (Array)	Exec. Time	Parallel Efficiency
8	(1,2)	$\begin{bmatrix} 0, & 1 \\ 1, & 1 \end{bmatrix}$	22	48 %
18	(1,3)	$\begin{bmatrix} 0, & 1 \\ 1, & 2 \end{bmatrix}$	69	78 %
36	(1,6)	$\begin{bmatrix} 0, & 1 \\ 1, & 5 \end{bmatrix}$	246	88 %
72	(1,12)	$\begin{bmatrix} 0, & 1 \\ 1, & 11 \end{bmatrix}$	924	94 %
144	(1,24)	$\begin{bmatrix} 0, & 1 \\ 1, & 23 \end{bmatrix}$	3576	97 %

5.4 Other Application in DSP

Table VI Design Results for Various applications

Application	Condition			T_{exe}	Area (Gates)		
	U	PE Func.	exu		$PE \times exu$	Mem.	DFC/Ctl
Matrix-vector	18	16 bits mult-add	6	69	2000×6	20736	2200
Matrix-matrix	16	16 bits mult-add	8	526	2000×8	16384	2000
Sorter	16	8 bits compare	2	47	100×2	0	830
Bit Serial Multiplier	16	1 bit adder	8	46	48×8	0	306
Bit serial square rooter	14	1 bit signed digit adder	6	52	32×6	0	702

Tab.VI shows the design results for various applications. The first three applications perform the word-level operations[77], while the last two applications perform the bit-level operations[65]. Only the matrix-matrix multiplier belongs to a 3D algorithm, while the others belong to 2D algorithms. The functions of each processing element are also described. U, exu and T_{exe} indicate the algorithm size, the number of PEs, and the total execution time in terms of the unit time, respectively. For performance evaluation, the number of gates of PEs, memory units and DFC or control logic circuits are calculated as area description.

Chapter 6

Window MSPA for Image Processings

In this chapter, we present a novel architecture named as Window-MSPA architecture which targets to window operations in image processing, as an extension of MSPA architecture. Window-MSPA deals with window operations in image processing by minimizing data I/O ports and the number of processing elements. The input scheme of image data is restricted to row by row input which simplifies the I/O architecture. Under this practical I/O restriction, the fastest processings are achieved. The architecture can be used for not only conventional image filterings, but practical window operations such as motion vector search in MPEG2. Moreover, the derived architecture is flexible enough to satisfy user's requirement for either area or speed.

6.1 Introduction

As hardware cost continues to drop and high performance requirement becomes critical in the DSP area such as telecommunication, computer vision, image processing, and signal processing, more special-purpose high throughput system are of great interest. The most time-consuming parts of the algorithmic descriptions are iterative loops for operations over all pixel data, which mostly behaves as window operation in VLSI hardware realization [78].

Window operations in image processings, which perform repetitive operations over the partial area in image data space, mostly consist of the identical operations over the pixels. Real-time operation of window-operations with high data throughput rates and high computational rates can only be carried out by parallel and pipelines processing [79], [80]. Currently, there are two ways to resolve this kind of problems [81]. One is to use general purpose multiprocessor system. Another one is to use systolic array processor. The problem in the first solution is that low throughput rate suffered from the limited communication rate of a global bus system. Since window operation usually shares data resources in a very heavy way, it is very common that multiple processors accesses share data resources simultaneously. It results in communication bottlenecks. While in the second solution, the local communication style is used to resolve low throughput problem but some disadvantages in systolic array are difficult to be avoided.

The conventional systolic architecture has been applied to the window operations [82], [83], [84]. However, there are three major common drawbacks to be resolved. The first is the large latency problem due to time taken by data input of reference or coefficient to the processor array. In some systolic designs, the data input such as reference or coefficient data for window operations must be performed in advance before the window processing starts. The second is the low parallel efficiency. It is due to idle time of each PE waiting for arrival of the required data through local communications. The third is the high hardware cost. Processor array requires multiple data streams for parallel processing. Those data streams mostly consist of regular data streams including the idle data in order to synchronize input data with parallel processings in the array. Especially, for window operations, each pixel data are used multiple times in the array as many as the number of related window operations. Each data need to be fed repeatedly when it is required by the array. This scheme requires additional hardware such as controller or Data Format Converter (DFC) to convert simple data streams to specified multiple data streams. On the other hand, if a simpler input data scheme is employed, local memory or shift registers are required to meet the data transfer requirement to the array.

All of these degrade the parallel efficiency and cost/performance such as area-time product. Therefore, minimization for both the number of processing elements (PEs) and the area for temporal data storage is necessary in order to achieve a small area-time product. However, conventional design which directly maps algorithms onto systolic array is hard to take area-oriented into consideration.

In this chapter, we present the excellent architecture dedicated to window operations in image processing, as an extension of MSPA architecture [77], [66]. MSPA concepts of loosely coupled array with common bus are inherited by Window-MSPA. Window-MSPA uses external image memory as common memory and receives data through internal common buses. If we look at the processor array chip together with external image data memory, the whole system behaves similarly to original MSPA architecture. Window-MSPA consists of processing element (PE) array, each of which is herein called as Window-PE. Window-PEs in a row shares the internal input and output data bus. By those buses, image data which are input from small number of I/O ports are broadcasted over Window-PEs. Input data are consumed soon in Window-MSPA when they are input to Window-MSPA. Therefore, there need no internal data storages. Those scheme contributes to achievement of high parallel efficiency and low area-time product.

Our main concern for the design of Window-MSPA are regular input data scheme, small number of data I/O ports and fast parallel processing of window operations. The derived architecture is flexible enough to satisfy the user's requirement for both area and speed. The developed theory explicitly derives the number of Window-PEs under the optimal I/O data input. Our design can also trade between area and time according to user's requirement. It will be used for various window operations in practical image processings such as linear/nonlinear filterings. Application to motion estimation search in MPEG2 coding is also performed in different projects, which results in great area reduction for IC implementation. based on the developed theory. The theoretical comparison between systolic array proves the promising advantages over the conventional approaches.

6.2 Window Operations

There are various kinds of repetitive window operations or repetitive finite area operations over the input data array in computer vision and image processings. They are listed in Tab.I [85], [86].

Table I Window Operation Applications

Spatial domain linear noise cleaning
Correlation function between two images
Image enhancement filtering
Adaptive histogram modification
Spatial Derivatives in edge detection
ieeetexfile/IEEEwoperation Pattern recognization
Nonlinear filtering (Median Filtering)
Enumeration sorting
Motion vector search in MPEG2 encoder

As shown in Tab.I, the first 5 kinds of applications can be seen as linear filtering process performed on a discrete image data array, whose convolution operators are performed over finite area. In addition to these applications, median filtering, motion estimation in MPEG2 encoder, etc are also taken place within the finite window area such as 3×3 pixels. Those identical window operation are repeated over the whole input image.

For example, in motion vector search in MPEG2 encoder, window operations appear in such a way that a reference block data of the current image frame is compared with blocks of the same size within the certain search areas of the previous image frames. The window operations correspond to the comparison operation such as summation of absolute difference of pixel values between the reference block and the search blocks.

The concept of the window operations in image processings is shown in Fig.6.1. The certain $K \times K$ window operation is performed over the same size area of the input

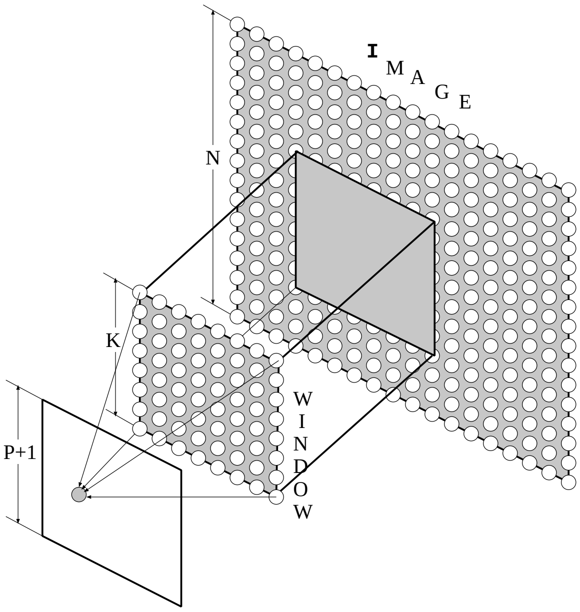


Fig. 6.1. Window operations

image data. In most cases, one output data is produced by each window operation. Those identical window operations are repeated over the whole input image data $N \times N$. The different window area is obtained by shifting one column rightwards or leftwards and one row upperwards or lowerwards from a window area. Since window is able to move along $N - K = P$ columns in horizontal direction and also to move along $N - K = P$ rows in vertical direction, the number M_{wop} of window operations in one frame is written by

$$M_{wop} = (P + 1) \times (P + 1), \tag{6.1}$$

where

$$P = N - K. \tag{6.2}$$

We define here *row window processing*, which is a set of $P + 1$ window operations over the same K row data of image data. There are $P + 1$ row window processings among all window processings. An example of row window processings for $N = 7$, $K = 3$ and $P+1 = 5$ is shown in Fig.6.2. The first row window processing is indicated, which consists of five 3×3 window operations. A row window processing is performed by processors in the same row in the processor array, which will be explained in detail later.

We consider here the usual cases that there are no dependency relation between different window operations. If we denote the result values of a window operation as

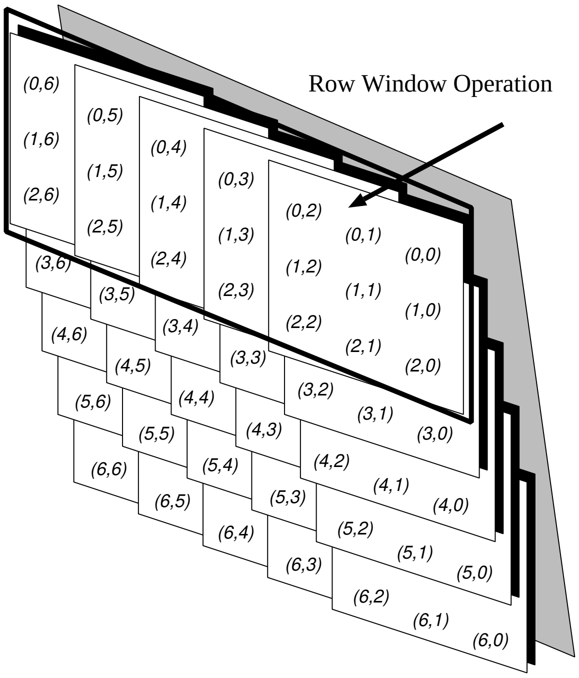


Fig. 6.2. Row window operations

y_{rs}, where $(r, s = 0, 1, 2, \cdots, P)$, input image data as $x_{r+i,s+j}$ and window operator as $w_{i,j}$, 2D window operation can be described as

$$y_{rs} = f(W, X_{rs}), \tag{6.3}$$

where $(i, j = 0, 1, 2, \cdots, K-1)$, $W = [w_{ij}], and X_{rs} = [x_{i+r,j+s}]$. W represents $K \times K$ window coefficient matrix for window operation and X_{rs} represents $K \times K$ image data matrix around (r, s) pixel.

As a special case such as 2D convolution, window operator is weighted summation over the window data as weight coefficient. That is described as

$$y_{rs} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} w_{i,j} x_{i+r,j+s}. \tag{6.4}$$

Similarly, for motion vector search of MPEG 2, we denote absolute sum as y_{rs}, current image frame data as $x_{r+i,s+j}$ and window reference block data as $w_{i,j}$. The window operation in this case can be described as

$$y_{rs} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} |x_{i+r,j+s} - w_{i,j}|, \tag{6.5}$$

where $(r, s = 0, 1, 2, \cdots, P)$. Different from 2D convolution case, the result values of $(P + 1) \times (P + 1)$ window operations are compared to find the best matched block with the reference block.

The consecutive window operations over the adjacent window areas in input image data reduce the required number of new data. In other words, each image data is used for $K \times K$ window operations. This fact minimizes the number of I/O ports to the processing array. It is noted that eq.(6.4)(6.5) are calculated by consecutive multiply-add or subtract-absolute operations in each Window-PE. The results of partial sum are stored and consumed in the next step.

The problem here is to derive the optimal processor array architecture for these repetitive window operations in image processing. The optimality depends on the user's specification so that the architecture must have a flexibility on architecture area and processing time.

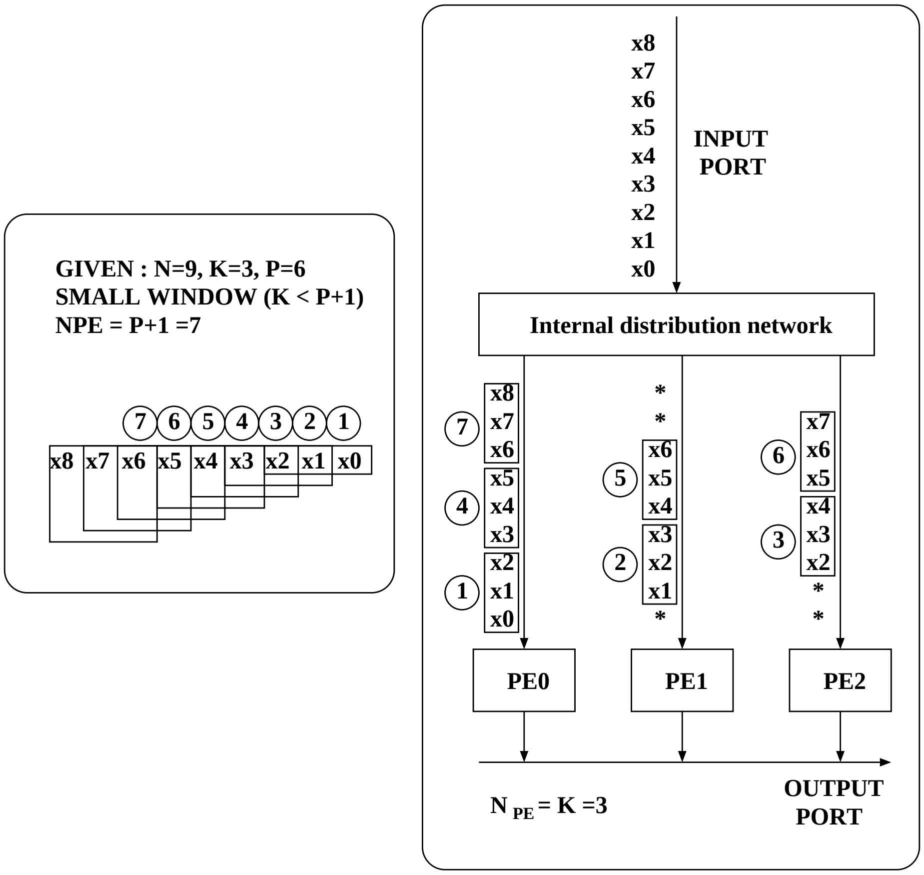


Fig. 6.3. FIR Window-MSPA Architecture

6.3 Simple Example of Window-MSPA

Window-MSPA theory can be easily used in many applications. We give a simple example here to demonstrate Window-MSPA operation principle. We take an example of one dimensional IIR filter, which can be expressed with

weight coefficient $W = w_1, w_2, \cdots, w_k, R = r_1, r_2, \cdots, r_h,$
initial value $Y_{initial} = y_{-h+1}, \cdots, y_{-1}, y_0,$
and input $X = x_1, x_2, \cdots, x_n.$

The problem is how to get

$$y_i = \sum_{j=1}^{k} w_j x_{i+j-1} + \sum_{j=1}^{h} r_j y_{i-j}. \tag{6.6}$$

FIR filter is considered as the special case of IIR filter with $r_j = 0$. With the Window-MSPA design theory, the Window-MSPA architecture for FIR application is shown in Fig.6.3.

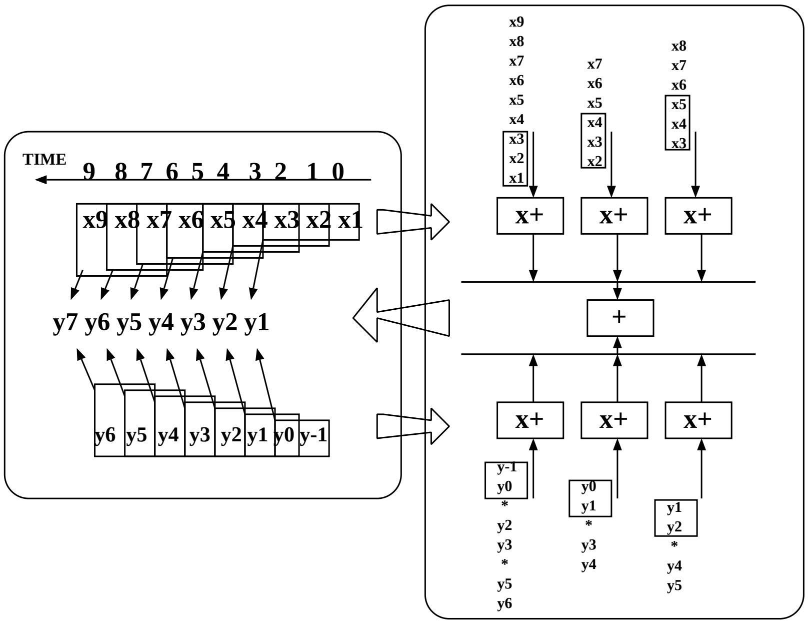


Fig. 6.4. IIR Window-MSPA Architecture (N=9, K=3, R=2)

In this example, there are 9 data to be dealt with. One window operation includes 3 data. The first window operation includes data x_0, x_1, x_2, the second window operation includes data x_1, x_2, x_3, and so on. We use three PEs to deal with each window

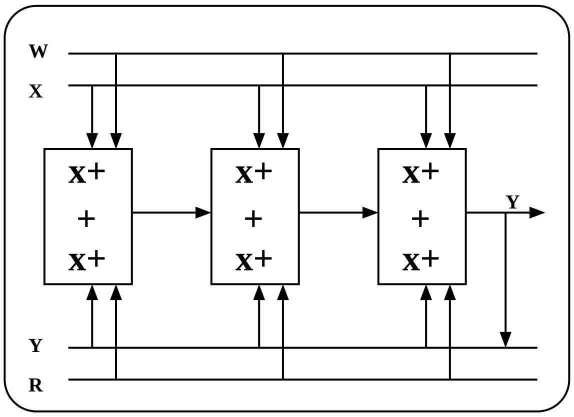


Fig. 6.5. Modified IIR Window-MSPA Architecture (N=9, K=3, R=2)

operation in parallel. The first window operation will be performed in PE0, while the second one will be performed in PE1, the third one will be performed in PE2. In order to share same data by multiple PEs with minimum data storage space, we resolve data transfer problem by introducing internal distribution network. By this format, the processing start time for each window operation will be delayed to certain unit time.

By similar method, the architecture of Window-MSPA for IIR is shown in Fig.6.4 Fig.6.5 shows the modified IIR architecture of Window MSPA. Comparing with systolic array shown in Fig.6.6, it is noted that proposed Window-MSPA architecture can achieve high data transfer rate, higher operation throughput, and high cost performance than systolic realization.

6.4 Window-MSPA Architecture

Window-MSPA configuration is shown in Fig.6.7. Window-MSPA architecture is dedicated to the repetitive window operations over the whole image. It consists of $N_{WPE}(C)$ rows of Window-PEs. Each row includes $N_{WPE}(R)$ Window-PEs with a row local data distribution network.

The input network switches input data through I/O ports to local data distribution networks in such a way that each local distribution network receives multiple rows of the image input in a parallel scheme. We configure both the input broadcast and local data distribution networks with multiplexer to realize data format conversion from input ports to each Window-PEs. The local data distribution network manipulates data from input broadcast network and control the data streams to each Window-PE in parallel. Data through internal buses are shared by multiple window operations. Data transfer rate of a bus is made the same as the processing rate of Window-PEs

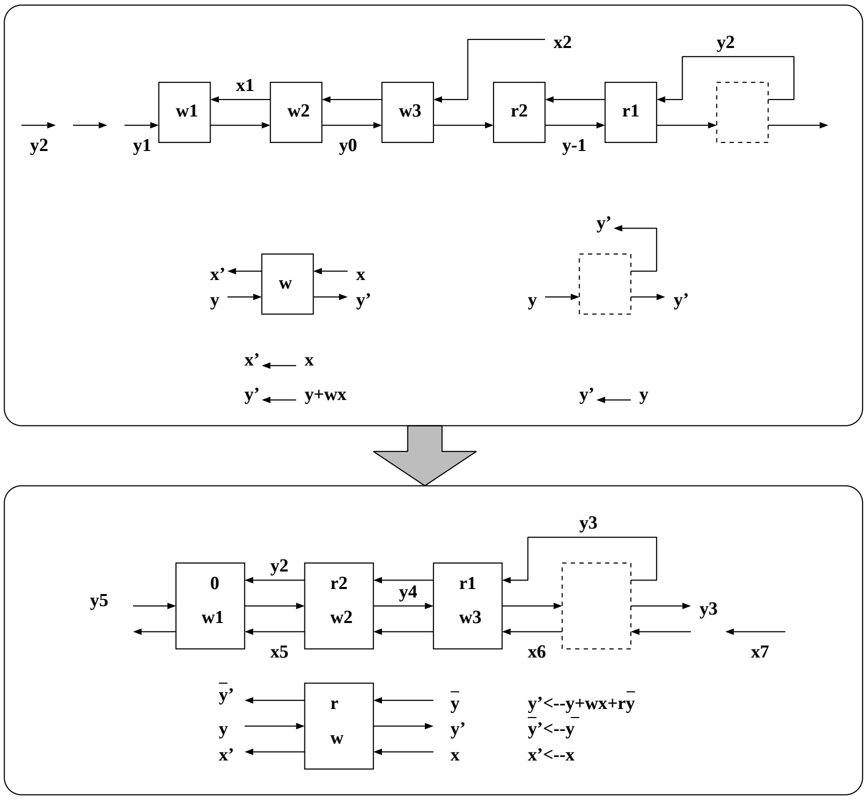


Fig. 6.6. Systolic IIR Window-MSPA Architecture

in order not to require additional data storages.

We denote the number of PE in one Window-PE as N_{pe}. Therefore, the total number N_{PE} of PE in Window-MSPA architecture is given by

$$N_{PE} = N_{WPE}(R) \times N_{WPE}(C) \times N_{pe}. \tag{6.7}$$

To simplify the explanation, we explain only the case $N_{pe} = 1$ when each Window-PE consists of a single processor element. We describe $N_{pe} \neq 1$ case as a modified design in section 7. In this case, Each Window-PE may be realized by a single PE, systolic array or MSPA. Each Window-PE accepts one pixel data and performs one step of a window operation in one unit time.

6.4.1 Input Scheme

We assume that the input data are transfered to Window-PE array row by row. This assumption makes control of the practical data stream quite easy. In following sections, it will be seen that input data schemes will be determined by some parameters

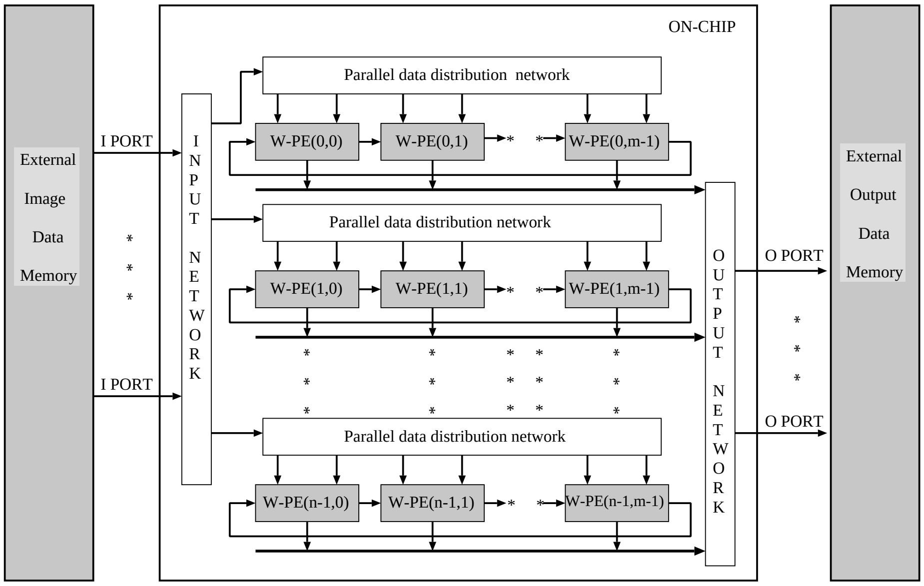


Fig. 6.7. Window-MSPA Architecture ($m = N_{PE}(R)$, $n = N_{PE}(C)$)

of window operation application. This kind of scheme will give a best solution to input data bottleneck problem especially in large window operation case.

The typical example of input scheme is shown in Fig.6.8. The left block shows an external image data memory, which transfers the data through I/O buses to I/O ports of Window-MSPA.

The data input time through each input port for one row of $N \times N$ image data is N. The delay between two adjacent rows of image input data are set as K unit time. It will be shown that this scheme enables each Window-PE to consume K consecutive row data input.

Because the repetitive usage of an I/O bus for row data transfers, we can reduce the number of I/O buses or I/O ports. In order to transfer N row data in every K unit times duration, at least $\lfloor \frac{N}{K} \rfloor$ I/O ports are necessary. The first $\lfloor \frac{N}{K} \rfloor$ rows of image data use different I/O buses. When $(\lfloor \frac{N}{K} \rfloor + 1)$-th row of image data is transfered, the first I/O bus already finish the input of the first row of image data. It can be reused for the transfer of $(\lfloor \frac{N}{K} \rfloor + 1)$-th row of image data to I/O port. Therefore, the number N_I of input ports is described as

$$N_I = \lfloor \tfrac{N}{K} \rfloor .$$

The number of internal buses for each row in Window-MSPA is at most N_I since input ports can convey all the input data to Window-MSPA no matter how the scheduling of window operations are. However, in some cases, the number of internal buses is reduced from N_I, which will be described later.

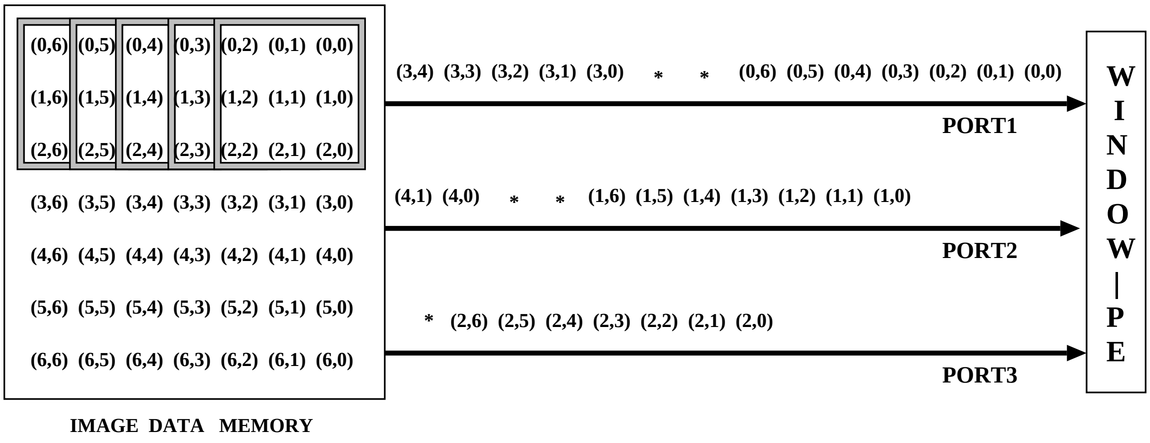


Fig. 6.8. Data Input Scheme ($N = 7$, $K = 3$, $P = 4$)

In case of Fig.6.8 for $N = 7$, $K = 3$ and $P = 4$, the left block represents 7×7 image data memory. The number of I/O ports are derived by

$$N_I = \lfloor \frac{N}{K} \rfloor = \lfloor \frac{7}{3} \rfloor = 3. \tag{6.8}$$

Actually, Window-PEs accepts 7 rows of image data through 3 I/O ports.

6.4.2 Definition of Large and Small Window

In case of $N_{pe} = 1$, the problem becomes how to reduce the number of Window-PEs. We will resolve the problem by two steps separately. The first step is how to reduce $N_{WPE}(R)$, which is an optimization problem for the number of Window-PEs located in the same row of Window-MSPA architecture. The second step is how to reduce $N_{WPE}(C)$, which is an optimization problem for the row number of Window-PEs in the Window-MSPA.

In case the window size $K \times K$ is large compared relatively with the $N \times N$ image data size, one window operation completes after all window operations of the row window operation started. In one row window operation, each Window-PE works in $K \times K$ unit times. There are $P + 1$ window processings within one row window operation, each of which starts at every unit times. Therefore, the large window processing is described by

$$K^2 \geq P + 1. \tag{6.9}$$

It describes that the first window operation terminates at K^2 after the last window operation of the first row window operation started at P+1. The difference between small ($K < 5$) and large ($K \leq 5$) window operations are shown in Fig.6.9 for $N = 22$ case. In small window case of $K = 3$, $P = 19$, the first window operation in the

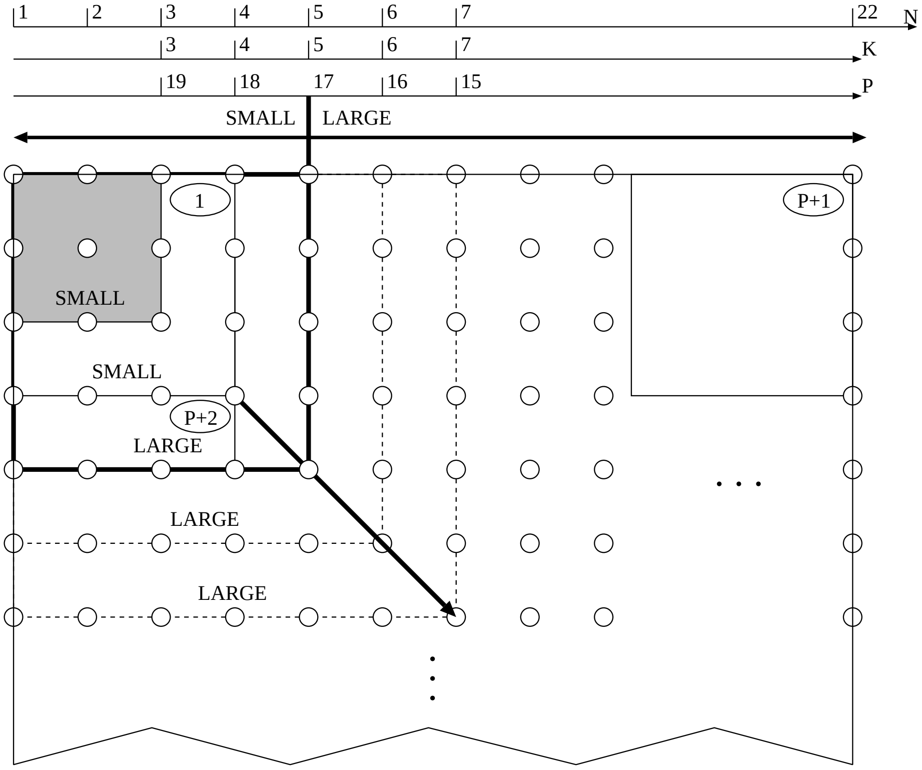


Fig. 6.9. Definition for Large and Small Window Operation

second row starts at $P + 2 = 21$ unit time after the finishing of the P+1-th (20th) window operation in the first row. Therefore, the first row window operation can be performed by sharing common Window-PEs. On the other hand, in case of large window case of $K = 7$, $P = 15$, the P+2-th (17th) window operation in the second row starts at $P+2 = 17$ unit time before the P+1-th (16th) window operation finished its window processing. It becomes difficult to share common Window-PEs between two adjacent rows of window operations.

6.5 Large Window Processing

6.5.1 Window-PEs in a Row

As explained above, Window-PEs in a row can not be reused for row window operation in large window case. Window-PEs in different rows must be prepared for different row window operation. Therefore, Window-MSPA requires the number $N_{WPE}(R)$ of Window-PEs in each row of Window-MSPA as

$$N_{WPE}(R) = P + 1. \tag{6.10}$$

Window-PEs in a row share the input data through input internal buses. Each one requires image data of K rows for a window operation. However, $\lfloor \frac{N}{K} \rfloor$ input ports are sufficient to convey all image data to Window-MSPA. It is easy to prove that $K > \lfloor \frac{N}{K} \rfloor$ holds for large window processings. Therefore, the number N_{II} of internal common buses for each row in Window-MSPA is given by

$$N_{II} = \lfloor \frac{N}{K} \rfloor. \tag{6.11}$$

6.5.2 Window-PEs in a Column

We decide the number $N_{WPE}(C)$ of the rows in Window-MSPA. The processing time of each Window-PE is K^2. The start time of each row of Window-PEs have a time delay of K between two adjacent rows of Window-PEs.

The first Window-PE ($WPE(0,0)$) in the first row starts a window operation at 0. Then, at K unit time, the first Window-PE ($WPE(1,0)$) in the second row starts its window operation. In such a way, the different row window operations start in every K unit time. In order to derive the row number $N_{WPE}(C)$ of window-PEs, we must consider whether the first row of Window-PEs can be reused for another rows of Window-PEs. We describe two kinds of time conditions for the first Window-PE ($WPE(0,0)$) in the first row to be reusable.

Condition for window operation termination
The first Window-PE ($WPE(0,0)$) in the first row terminates a window operation at $K \times K$ unit time. The time T_{WPE} for the first Window-PE $WPE(0,0)$ to become idle is written by

$$T_{WPE} = K \times K. \tag{6.12}$$

Condition for internal bus vacancy
Plural Window-PEs in the same row share input internal buses from I/O ports. Therefore, even if Window-PE finishes its operation, it might not be reused. It must wait until its input internal buses become vacant. The time is given by when all the first row of image data are transfered to the rows of Window-PEs. Since N Data in a row are sequentially input one by one through a bus, it takes N unit time to complete a row data input. However, each row data are determined to be input in every K unit times. Thus, the available reuse time T_{bus} for input internal bus must be given by the product of the number of I/O buses $\lfloor \frac{N}{K} \rfloor$ and row data input duration K, which become larger than N. It is written by

$$T_{bus} = \lfloor \frac{N}{K} \rfloor \cdot K. \tag{6.13}$$

$P + 1$ row Window-MSPA

Reuse of $WPE(0,0)$ is available after both terminations of a window operation and data input to the first I/O bus. Therefore, the available reuse time T_{reuse} of

$WPE(0,0)$ is given by

$$\begin{aligned} T_{reuse} &= \max\{T_{WPE}, T_{bus}\} && (6.14) \\ &= \max\{K, \lfloor\frac{N}{K}\rfloor\}\cdot K. && (6.15) \end{aligned}$$

The number of row window operations are $P+1$. Each one starts in every K times, one by one. Thus, the starting time for the last row window operation become $P\times K$. If $WPE(0,0)$ is not reusable at this moment, MSPA must prepare Window-PEs in a new row of Window-MSPA. This condition is described by

$$P\cdot K < \max\{K, \lfloor\frac{N}{K}\rfloor\}\cdot K. \qquad (6.16)$$

The above equation is simplified as

$$P < \max\{K, \lfloor\frac{N}{K}\rfloor\}. \qquad (6.17)$$

In that case, all row window operations are performed by Window-PEs in different rows. The number $N_{WPE}(C)$ of rows of Window-MSPA are written by

$$N_{WPE}(C) = P+1. \qquad (6.18)$$

Smaller row Window-MSPA

When Window-PE in the first row become reusable before all row window operations start, we can reduce the number $N_{WPE}(C)$ of rows in Window-MSPA from (P+1). The condition of this case is described by

$$P+1 > \max\{K, \lfloor\frac{N}{K}\rfloor\}. \qquad (6.19)$$

In this case, the number $N_{WPE}(C)$ of rows of Window-PEs is reduced to

$$N_{WPE}(C) = \max\{K, \lfloor\frac{N}{K}\rfloor\}. \qquad (6.20)$$

The $N_{WPE}(C)$ rows of Window-PEs are reused for new different row window operations repeatedly until all row window operations complete.

The number of Window-PEs in large window processing is summarized as bellows. In large window processing of $K^2 \geq P+1$, the column number of Window-PEs is described by

$$N_{WPE}(R) = P+1. \qquad (6.21)$$

While, the number $N_{WPE}(C)$ of rows of Window-PEs is described by two different equations(6.18)(6.20). However, in both cases, the smaller value between $P+1$

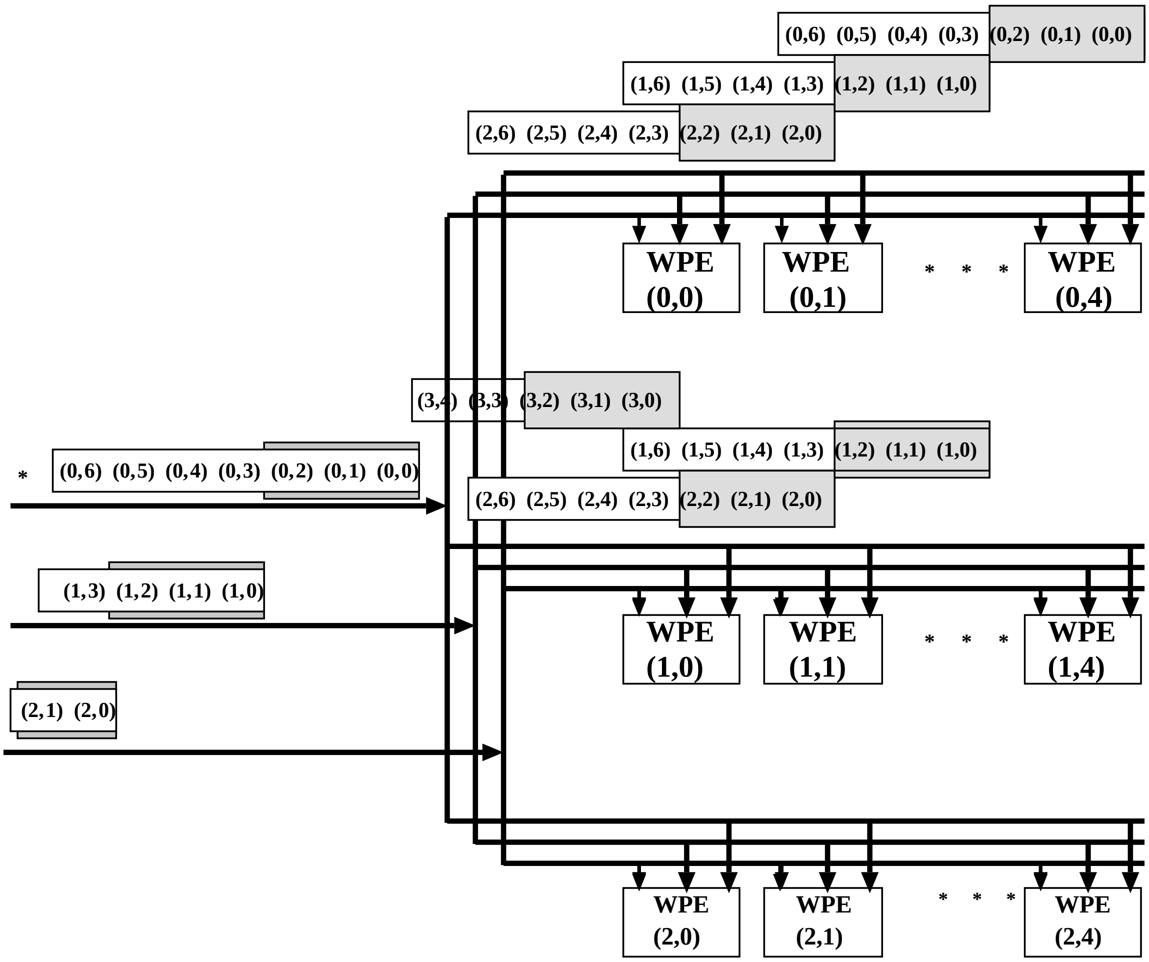


Fig. 6.10. Large Window Processing ($N = 7$,$K = 3$,$P = 4$)

and $\max\{K, \lfloor \frac{N}{K} \rfloor\}$ become $N_{WPE}(C)$. Therefore, the number $N_{WPE}(C)$ of rows of Window-PEs is described by

$$N_{WPE}(C) = \min\{P + 1, \max\{K, \lfloor \frac{N}{K} \rfloor\}\}. \tag{6.22}$$

The example of $N = 7$, $K = 3$, and $P = 4$ as shown in Fig.6.10. It has only 3 input ports, which transfer 3 rows of image data in parallel. Window-MSPA consists of 3×5 Window-PE array. Each row contains 5 Window-PEs with 3 internal buses, which are required to perform 3×3 window operations as indicated by the flows of input data to each Window-PE. 5 Window-PEs in each row completes 5 row window operations within 13 unit times. While 5 Window-PEs in the next row perform different 5 row window operations with 3 unit time delay.

6.5.3 Computation Time

The total computation time for one frame is equivalent to the time taken for the data input from image memory to I/O ports of Window-MSPA. Window-MSPA consumes their data in parallel and produces the results of window operations. When data input are completed, Window-MSPA also terminates its window operations. The latency is defined as the computation time T_T for one frame. It is given for any window size by

$$T_T = K \cdot (N - 1) + N. \tag{6.23}$$

The first term of eq.(6.34) corresponds to the time before the last row of image data is input, because each row is input to Window-MSPA in every window size (K) unit time. There are no irregular scheme for input image data. While, the second term corresponds to the input time for the last row of image data.

For repetitive multi-frame window operations, the window processing iterative period for each frame become more important than the latency. The period T_P is described by time duration of the repetitive data input of image frames. Removing the overlapping time between two adjacent frames, the iterative computation time for every frame is reduced to

$$T_P = K \cdot N. \tag{6.24}$$

6.5.4 Window Coefficient Input

For one frame window processings , the coefficient or reference data are repetitively used. They are input once through one additional input port to the first Window-PE in the first row of Window-MSPA as shown in Fig.6.11. They are also input to Window-PEs in different row in similar manner. They require a multiple of K delay in order to synchronize the input of image data. In the figure, the d indicates the shift register in each Window-PE. The C indicates the local control unit or shift registers used between adjacent two rows of Window-PEs.

Due to one unit time delay between two adjacent image data in the same row data, the coefficient will be input and shifted to the adjacent Window-PE one by one by the Window-PEs in the same row. Window-PEs will reuse the shifted coefficient data transferred from adjacent Window-PE. Coefficient data are shifted not only within the same row, but also among different rows of window operations because of the K delay image inputs scheme between two adjacent row data. The shift registers in each row of Window-PEs do not increase the area of Window-PEs so much. But the shift registers between two adjacent rows introduce a small area increase. In other words, the coefficient data input requires one more input port usually with a little area increase.

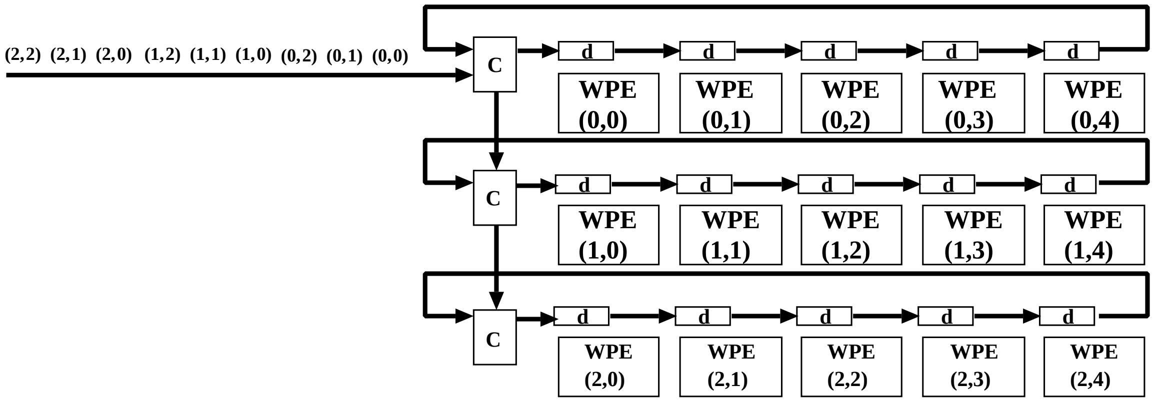


Fig. 6.11. Coefficient Input Scheme ($N = 7$,$K = 3$,$P = 4$)

6.6 Small Window Processing

In large window processing, window operations in a row window operations are performed by different Window-PEs in the same row of Window-MSPA. However, in the case of small window processing specified by

$$K^2 < P + 1, \tag{6.25}$$

each row window operation time becomes short and Window-PEs even in the same row can be reused for different window operations. Therefore, the number of Window-PEs in a row is less than $P + 1$, which is different from the large window case.

6.6.1 Window-PEs in a Row

We can reduce the number of Window-PEs $N_{WPE}(R)$, which perform window operations in a row window operation. When the first data comes in, the first Window-PE ($WPE(0,0)$) starts the first window operation. When the second data come in, the first Window-PE ($WPE(0,0)$) continues processing, while the second Window-PE ($WPE(0,1)$) starts the second window operation. In such a way, different Window-PEs in the same row start their window operations. The first Window-PE finishes the window operation at K^2 unit time for the window size of $K \times K$. Therefore, the first Window-PE can be reused for the next $K^2 + 1$-th window operation in the same row window operation. There are no need to employ new Window-PEs in the row. As a result, the required number $N_{WPE}(R)$ of Window-PE in a row is given by

$$N_{WPE}(R) = K^2. \tag{6.26}$$

Window-PEs in a row share K input row data through input internal buses. They are transferred through $\lfloor \frac{N}{K} \rfloor$ input ports. Since $K < \lfloor \frac{N}{K} \rfloor$ holds for small window processings, the number N_{II} of internal common buses for each row in Window-MSPA is given by

$$N_{II} = K. \tag{6.27}$$

6.6.2 Window-PEs in a Column

The optimization problem for rows of Window-PEs is similar to the case of large window processing. From the condition of small window operation $K^2 < (P+1)$, it is obvious that minimization of rows of Window-PEs will become more important. The difference from the case of large window operation is that each Window-PE will be reused even in a row window operation.

We consider the similar condition of reuse of the first row of Window-PEs for different row window operations. With regards to availability of Window-PEs in the first row, there exists at least an available Window-PE, because the reuse of Window-PEs in the same row in every K times are guaranteed by the small window operation condition ($K^2 < P+1$). It is noted that the reuse of Window-PE in a row for different row window operations need not start from the first Window-PE in the row.

Fewer Internal buses

We only consider the condition of data input termination, which is described by the available reuse time T_{bus} for input internal bus of the first row of Window-PEs as in eq.6.13. Since each row window operation starts in every K unit time, the number $N_{WPE}(C)$ of row window operations can start within available reuse time T_{bus}, which is given by

$$N_{WPE}(C) = \lfloor \frac{N}{K} \rfloor. \tag{6.28}$$

This number equals to the number $N_{WPE}(C)$ of rows for Window-MSPA.

By the way, it is easily proved under the small window operation condition of $K^2 < P+1$ that

$$P+1 > \lfloor \frac{N}{K} \rfloor \tag{6.29}$$

holds. This inequality indicates that rows of Window-PEs can be reused repeatedly for $P+1$ row window operations. Therefore, the number $N_{WPE}(C)$ of rows in Window-MSPA is described by eq.6.28.

An example of small window processing case is shown in Fig.6.12, where $N = 7$, $K = 2$ and $P = 5$. Window-PEs in the first row input the data sent through I/O port 1 and 2. The first Window-PE $WPE(0,0)$ performs 2×2 window operation while the second Window-PE $WPE(0,1)$ performs 2×2 window operation by use of the same row data. Since those Window-PEs terminate a row window operation within 4 unit times, they are reused for different window operations over the same rows of image data.

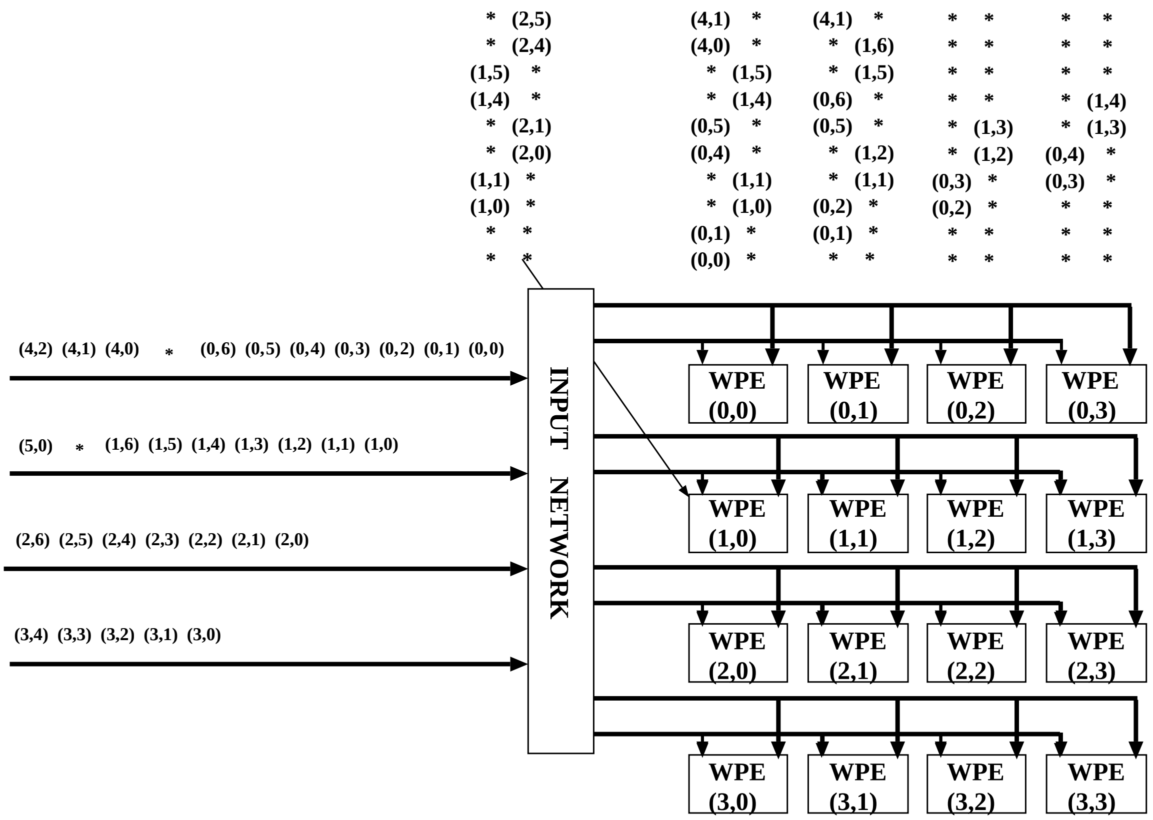


Fig. 6.12. Fewer Internal Buses in Small Window Processing ($N = 7$,$K = 2$,$P = 5$)

Fewer Window-PEs

We get further speed-up at the expense of one additional internal buses as shown in Fig.6.13. The number of rows in large window case is determined by the time for internal bus vacancy. We can relax this condition by increasing the number of input internal buses by one. New condition to restrict the number of rows become the time of the completion of the first row window operation. The first row window operations are taken place in Window-PEs in the first row. Each window operation in the first row window operation starts to invokes one unit time by one unit time. The time to start the last window operation in the first row window operation is $N - K$. After this time, Window-PEs in the first row are available to perform a new row window operation, though $\lfloor \frac{N}{K} \rfloor$ internal buses are still occupied. The time to start row window operations is restricted to a multiple of K due to K durations of row data input of image data. Therefore, the earliest reusable time for Window-PE in the first row is given by
]

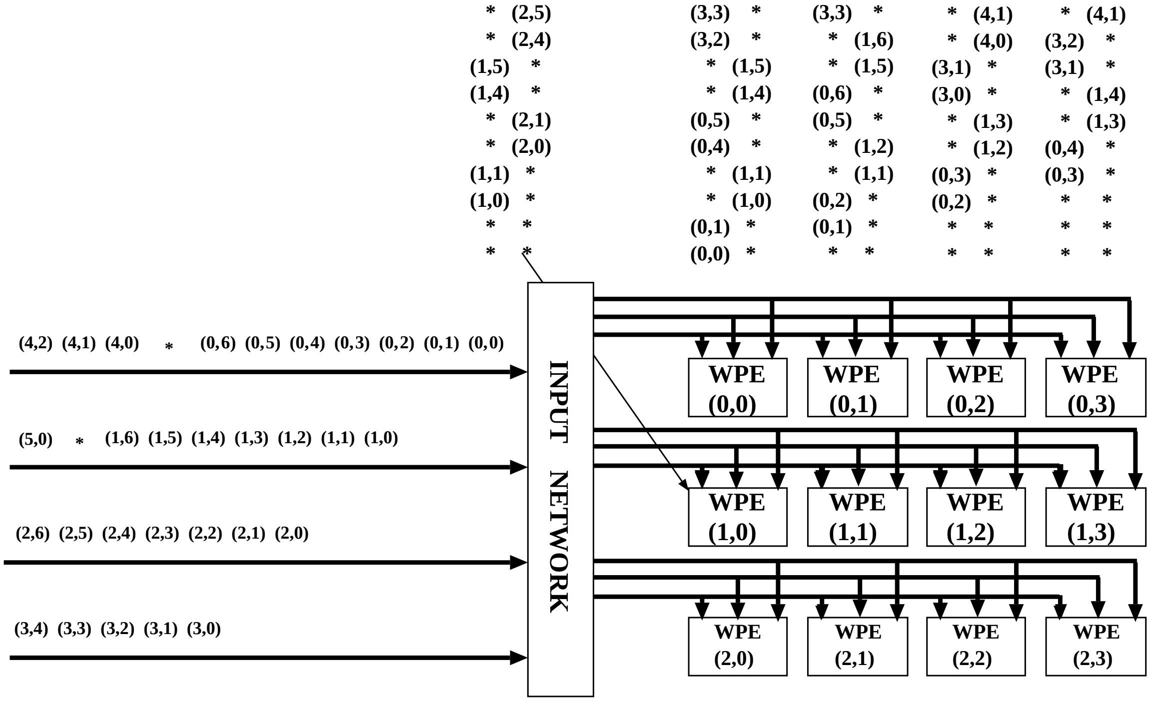


Fig. 6.13. Fewer Window-PEs in Small Window Processing ($N = 7, K = 2, P = 5$)

$$
\begin{aligned}
T_{reuse} &= \lfloor\frac{N-K}{K}\rfloor \cdot K && (6.30)\\
&= (\lfloor\frac{N}{K}\rfloor - 1) \cdot K. && (6.31)
\end{aligned}
$$

This fact indicates that T_{reuse}/K-th row window operation can be performed by the first row of Window-MSPA. Therefore, the number of rows of Window-MSPA is derived as

$$N_{WPE}(C) = \frac{T_{reuse}}{K} = \lfloor\frac{N}{K}\rfloor - 1, \tag{6.32}$$

which represents Window-MSPA with one row smaller than the case for fewer input internal buses.

The number N_{II} of input internal buses increases in this case up to

$$N_{II} = K + 1. \tag{6.33}$$

6.6.3 Computation Time

The total computation time T_T and T_P both for one frame and multi-frame window operations are decided by input scheme, which is independent on window size. Thus, they are expressed by the same equation as the large window processing case as

$$T_T = K \cdot (N-1) + N. \tag{6.34}$$

$$T_P = K \cdot N. \tag{6.35}$$

6.7 Modified Design

It will be seen that for very small window size case, our results employ larger number of Window-PEs than systolic array. On the other hand, for very large window size case, our results use quite few number of input ports and Window-PEs. Although our architecture achieves high area-time efficiency for both very small and very large case, it is still necessary to modify the proposed architecture to realize either *area small* or *fast speed* design. Modified design enables the trade off between area and speed up.

We will explain the area small and fast speed design, respectively, as one of the modifications for Window-MSPAs. The first modification shows how to reduce area by sacrificing processing speed, i.e. how to decrease the numbers of input ports and PEs. The second one is used to show how to speedup processing by increasing the numbers of input ports and PEs.

6.7.1 Area Small Design

According to area-oriented requirement, it is necessary to reduce the number of Window-PEs. We present area small design method in order to meet area-oriented design requirement. In order to reduce the number of Window-PEs. We firstly reduce the row number of Window-PEs, then, we reduce the column number of Window-PEs in a row further more. Because the method to reduce the column number is the same as to reduce the row number, we discuss how to reduce the row number only.

In the very small window case, $P+1$ row window operations are performed in several steps, each of which performs several row window operations. In order to meet the requirement of area-oriented design, the design procedure can be changed as follows. Different from the previous design, we decide the number of PEs first. The number of I/O ports or I/O scheme is determined by PE number.

1. Firstly, select the less number of rows of Window MSPA than normal small window case. The modified row number of Window-PE is denoted as $N_{WPE}(C)$. If the row number of Window-PE before modifying is given in eq.6.22, then, the $N_{WPE}(C)$ satisfies

$$1 \leq N_{WPE}(C) < \min\{P+1, \max\{K, \lfloor \frac{N}{K} \rfloor\}\}. \tag{6.36}$$

2. Secondly, according to the reduction of Window-PEs, modify the number of input port also denoted as N_I. Because $N_{WPE}(C)$ rows of Window-PEs need at most $N_{WPE}(C)+K-1$ rows of input data, the N_I should take the minimum value between original number of I/O ports $\lfloor \frac{N}{K} \rfloor$ and $N_{WPE}(C)+K-1$. Therefore, it is obtained by

$$N_I = min(\lfloor \frac{N}{K} \rfloor, N_{WPE}(C)+K-1). \tag{6.37}$$

3. Thirdly, decide how many sequential strips the whole frame window operations are divided into. The number of strips is denoted as N_S. $P+1$ row window operations for one frame data are divided into strips consisting of the same number of row window operation. They are performed one strip by one strip, sequentially. Accordingly, the image data are divided into several strips with data overlapping scheme. The input scheme is also modified as shown in Fig.6.14.

   In Fig.6.14, the window operations for the whole frame are divided into several strips (3 strips in Fig.6.14), which are performed sequentially. Each of them consists of $N \times (N_{WPE(C)}+K-1) = 21$ window operations.

   The whole processing time for one frame has a close relation with the number of strips. Because one row of Window-PE deals with one row of window operations, $N_{WPE}(C)$ row of Window-PE can deal with P+1 rows of window operations only

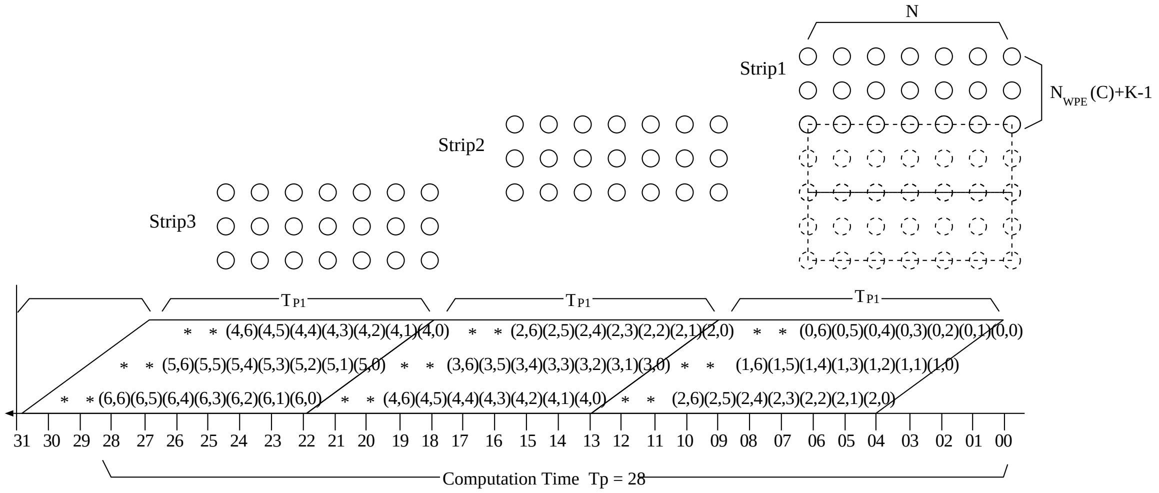


Fig. 6.14. Computation Time for Area-oriented Design ($N = 7$,$K = 2$,$P = 5$)

by separating the whole frame window operations to N_S times. Therefore, the N_S is given by

$$N_S = \lfloor \frac{P+1}{N_{WPE}(C)} \rfloor. \tag{6.38}$$

4. Fourthly, as shown in Fig.6.14, the window operation time for one strip of window operation denoted as T_{P1} is decided by

$$T_{P1} = N + (N_{WPE}(C) + K - 2) \times K. \tag{6.39}$$

In multiple strip window operations, the start time of the second strip is determined by three parameters as follows. One is the earliest start time for the first window operation in the second strip window operation. The data of the first row in the second strip window operation is input from the $(N_{WPE}(C) + 1)$-th row of the frame data. Because each row of data has a input delay of K, the earliest work time of the second strip will be

$$T_{earlies} = (N_{WPE}(C)) \times K. \tag{6.40}$$

The second parameter is the reuse time of input ports and internal data buses, it is given in previous section as

$$T_{bus} = \lfloor \frac{N}{K} \rfloor \times K. \tag{6.41}$$

The last parameter is the Window-PE reuse time. It is given by

$$T_{WPERe} = \begin{cases} K^2 & (\text{P+1} \le K^2) \\ \lfloor \frac{N}{K} \rfloor \times K & (P+1 > K^2) \\ (\lfloor \frac{N}{K} \rfloor - 1) \times K & (P+1 > K^2, N_{II} = K+1). \end{cases} \tag{6.42}$$

Thus, the start time of the first Window-PE in the second strip will be

$$T_{active} = max\{T_{earlies}, T_{bus}, T_{WPERe}\}. \tag{6.43}$$

Though the modify design method is able to be applied to both large and small window operations, it is specially more effective to small window operation. In small window operation case, the above formula will become

$$T_{active} = max\{N_{WPE}(C) \times K, \lfloor \frac{N}{K} \rfloor \times K\}. \tag{6.44}$$

Reference with the Fig.6.14, the total computation time for one frame window operation in multiple strip work mode will be

$$T_T = \begin{cases} N_S \times T_{active} + ((P+1) \bmod N_{WPE}(C) + K - 2) \times K \\ (\text{when (P+1) mod} N_{WPE}(C) \neq 0 \text{)} \\ N_S \times T_{active} + (N_{WPE}(C) + K - 2) \times K \\ (\text{ when (P+1) mod} N_{WPE}(C) = 0). \end{cases} \tag{6.45}$$

In multiple frame mode, the computation time per frame become

$$T_P = N_S \times T_{active}. \tag{6.46}$$

The example is shown in Fig.6.15. For a given $N = 7$, $K = 2$, $P = 5$, and $m = 2$, we obtained the parameter as $N_{WPE}(C) = 2$, $N_S = 3$, $N_I = 3$, $T_{earlist} = 4$, $T_{WPEre} = 8$, $T_{P1} = 8$, $T_T = 28$, $T_P + 24$.

For each strip window operation, $N_{WPE}(C)+K-1$ rows of frame data are required. The $K-1$ rows of data are used in both current strip and the next strip. This scheme requires $N_{REG} = N \times (K-1)$ registers for data storage inside Window-MSPA.

6.7.2 Speed Fast Design

In the very large window case, the processing time T_P can be shorten by employing N_{pe} PEs in a Window-PE. An example is shown in Fig.6.16. In case of $N_{pe} = 2$, each Window-PE in a row accepts two data at one unit time and performs 2 row window operations at the same time. Input scheme to input ports must also be modified in such a way that 2 adjacent row data of image data are input together into array. We

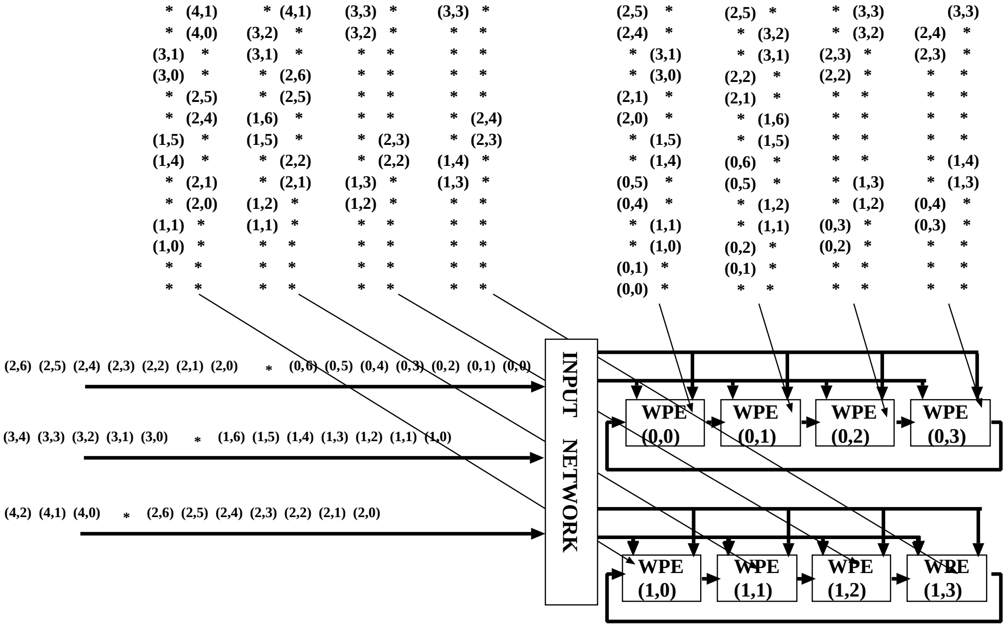


Fig. 6.15. Small Area Design in Small Window Processing ($N = 7, K = 2, P = 5$)

achieve almost 2 times as fast processing as original one at the expense of 2 times as many Window-PEs as original one.

Generally, the processing time T_P is shorten up to $\lfloor \frac{K(N-1)}{N_{pe}} \rfloor + N$. However, the number of input ports N_I' increases up to $N_{pe} \times N_I$ and the number of Window-PEs N_{WPE}' up to $N_{pe} \times N_{PE}$.

6.8 Performance Evaluation

Performance evaluation parameters are defined as follows.

- Data Transfer Efficiency

  In order to evaluate the efficiency of an array processor architecture more in detail and practically, the data format conversion part should also be considered as a part of the whole system together with array processor itself. In high speed image processing, the I/O port bottleneck become an important problem when array processor achieves high parallel efficiency. This problem is usually resolved by introducing data format converter (DFC), such as temporary memory, shifter registers in systolic array. Thus, the efficiency of DFC also become very important to be concerned. However, the architecture of DFC depends on

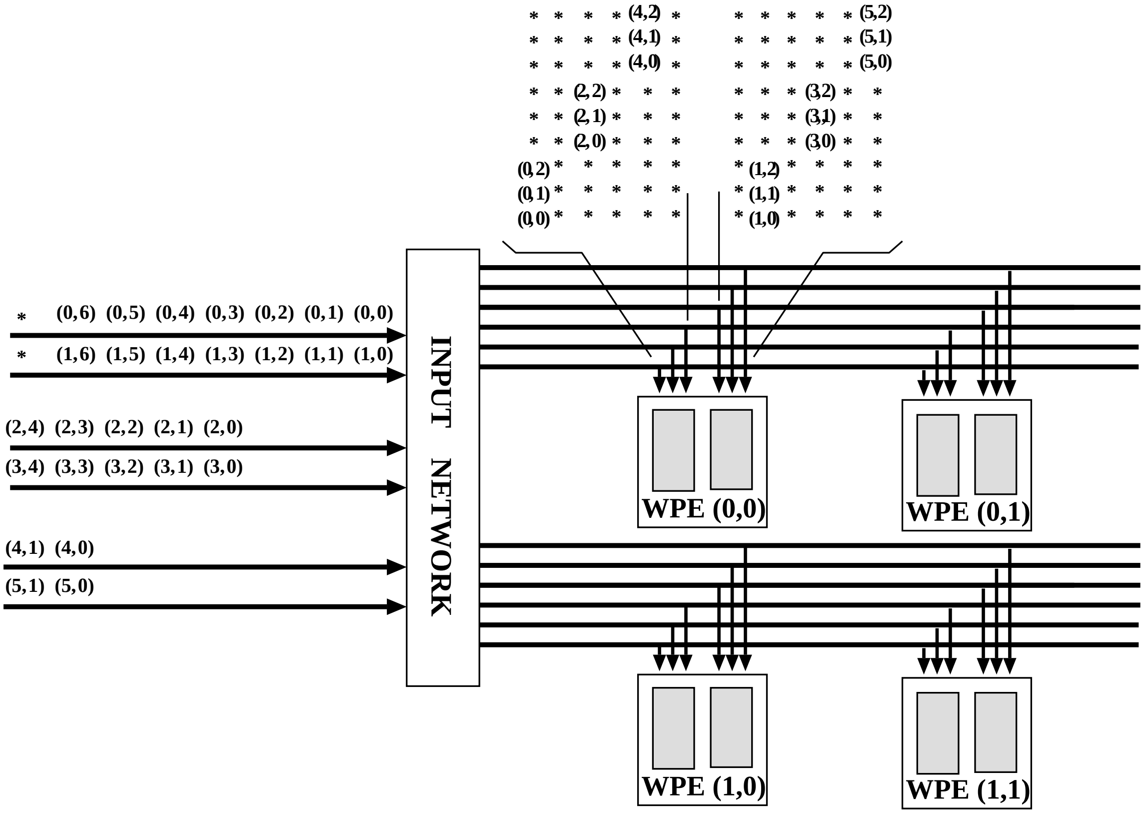


Fig. 6.16. Fast Speed Design in Large Window Processing ($N = 7, K = 3, P = 4$)

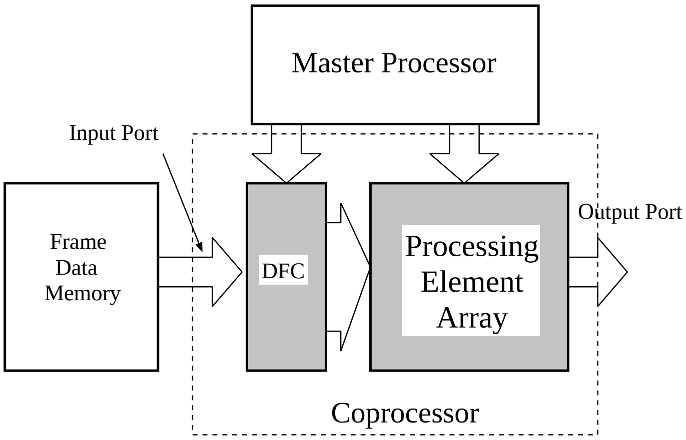


Fig. 6.17. Window operation system

scheduling and resource allocation of each array processor. It is difficult to evaluate the area cost of DFC directly. In an alternative way, we herein introduce data transfer efficiency denoted as E_{DT} to evaluate the DFC system.

The E_{DT} is defined as:

$$\begin{aligned} E_{DT} &= \frac{\text{Net number of transferred data} \times 100\%}{(\text{Transfer time}) \times (\text{Number of input ports})} \\ &= \frac{N^2}{T_P \times N_I} \times 100\%. \end{aligned} \tag{6.47}$$

The N^2 indicates the effective number of data in one frame. The T_P indicates the transfer time for one frame, which is the same as processing time for one frame. The N_I indicates the number of input ports for the data input of one frame.

The effective number of data includes the distinct number of data transferred through ports. If the data transfer includes idle data or repetitive data transfers of the same data, it worsens the data transfer efficiency. The more input ports are employed for the same number of data transfer in the same access rate, the lower the data transfer efficiency becomes. This is the case for systolic array, which requires many input ports with low data transfer efficiency for parallel processings.

The area cost of DFC can be evaluated indirectly by assuming that a low data transfer efficiency E_{DT} results in a large area waste of DFC relatively.

- Parallel Efficiency

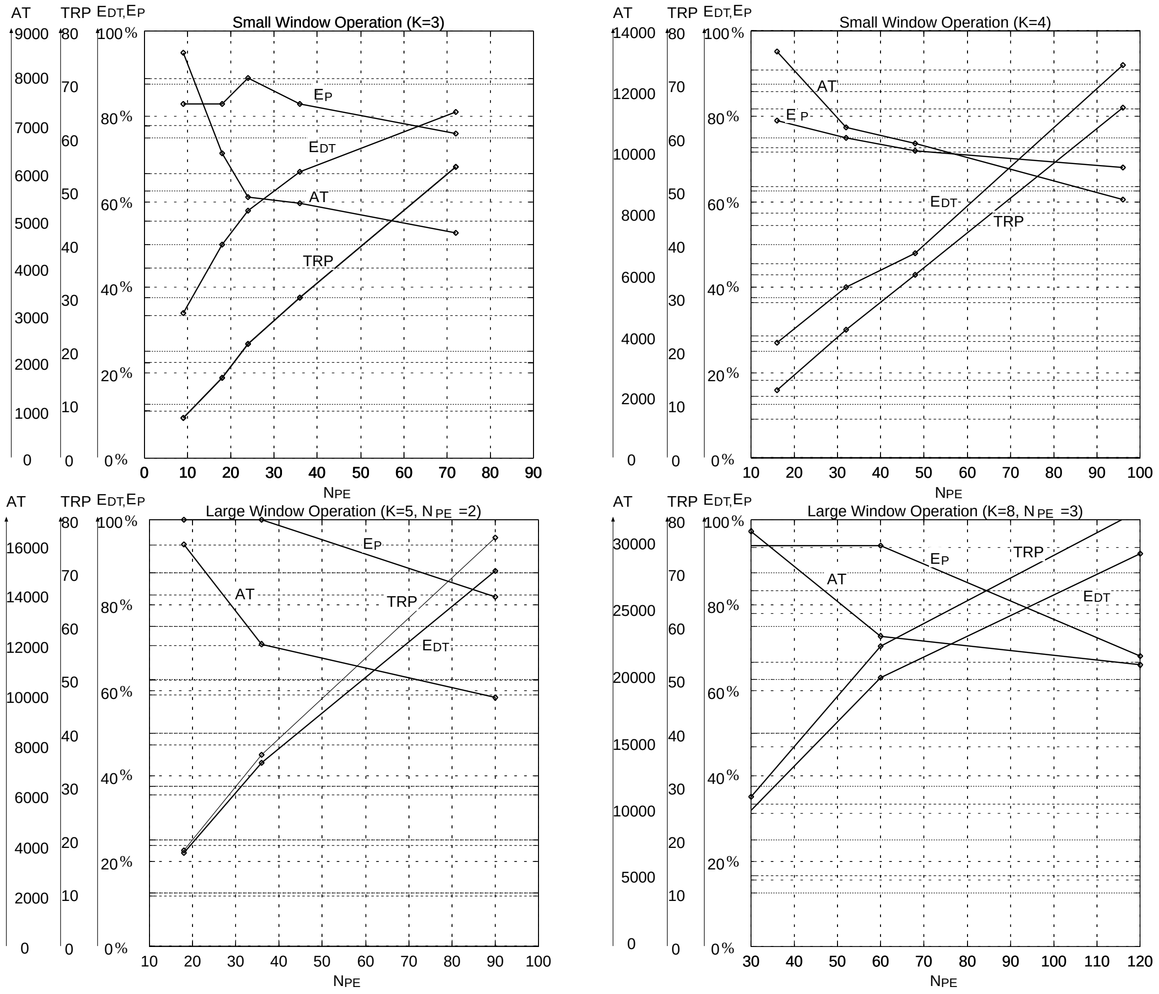


Fig. 6.18. Performance for Area Small Modified Designs

As a common case, we use parallel efficiency E_P as one of the measure parameters, which is defined by

$$
\begin{aligned}
E_P &= \frac{\text{Computation count} \times 100\%}{(\text{Computation time}) \cdot (\text{PE number})} \\
&= \frac{K^2 \times (P+1)^2}{T_P \cdot N_{PE}} \times 100\%. \qquad (6.48)
\end{aligned}
$$

Large value of E_P means more window operations can be executed at one unit time.

- Area-time Product

Since our data broadcast system is a switch-like control system without data storage, it has the advantage of both high speed and small area. Consequently, the area of the Window-MSPA can be simply evaluated by the number of PEs. On the other hand, systolic array always uses temporary memory to synchronize data with parallel processings in the array. It usually suffers from a large additional hardware cost. Therefore, it is unavoidable to take account the space of the temporary memory into the area estimation of systolic array. As a usual case, systolic array uses register array to keep current data and realizes data format conversion for regular processing. In order to compare area of our result with systolic array, we assume that one register area is equivalent to one fifth of one PE area.

As to area-time efficiency estimation, the product, denoted by AT_P, of array processor area and processing time is often used.

Because the area of array processor is very difficult to estimate, we regard array processor as a kind of coprocessor as shown in Fig.6.17. By this way, it is possible to compare various kinds of array processor in the same condition. As shown in Fig.6.17, we assume that the control units for the array processor are almost the same and they can be regarded as a part of master processor. Then, the area A of the array processor can be estimated by the sum of the numbers of PEs denoted as N_{PE} and one fifth of the DFC register number denoted as N_{REG}.

$$\mathrm{A} = N_{PE} + \tfrac{1}{5} \times N_{REG}$$

Here, we assume that the area of one register of DFC equals one fifth of one PE.

- Operation Throughput

  Operation throughput TRP is defined by

$$\begin{aligned} TRP &= \frac{\text{Computation count}}{\text{Computation time}} \\ &= \frac{K^2 \times (P+1)^2}{T_P}. \end{aligned} \tag{6.49}$$

  Throughput can be seen as the number of window operations per unit time performed by array processings.

Comparisons are performed with different kinds of systolic arrays [82] [83] [87] [88]. The results are shown in TableII. We select several window size of K under the fixed image data of $N = 22$.

In the first several columns, we show the design characteristics, such as the number N_I of input ports, the number N_{II} of internal data buses, the number N_{PE} of Window-PEs, the number N_{REG} of register in DFC, and the processing time T_p for one frame in repetitive multiple frame processing case. The performance is estimated by 4 items: operation throughput TRP, data transfer efficiency E_{DT}, the parallel efficiency E_P, area-time efficiency AT_P, respectively. For small window cases such as $K = 3$ and $K = 4$, both *fewer internal bus design* and *fewer Window-PE design* are performed. The results are shown in Table II by the name of MSPA1 and MSPA2, respectively. *Fewer Window-PE designs* require registers, which are different from the other MSPA design.

It is obvious that proposed array architecture obtains a very high area-time efficiency. In case that the window is not so small, the proposed architecture shows both very high data transfer and parallel efficiencies. In the opposite, if the window is not so large, our architecture also achieves very high processing throughput.

The design results for both modified cases are listed in Table.III, and Table.IV. In *Small Area Design*, the performance parameters corresponding to different value of N_{PE} for both small and large window operations are shown in Fig.6.18. It is seen that, in *Small Area Design*, the performance parameters E_{DT}, TRP, and AT except for E_P become worse along with the reduction of N_{PE} than those before modified. However, *Small Area Design* can dedicate to area-oriented design by improving original design while still keeping quite high performance. On other hand, the *Fast Speed Design* can realize time-oriented design.

From Fig.6.18, it is obvious that the operation throughput increases almost proportional to the increase of N_{PE}, which is different from saturation tendency in normal common bus multi-processor systems. Corresponding to this character, the parallel efficiency degrades slightly due to the increase of N_{PE}. All of these is own to the linear-like curve of E_{DT} as shown in Fig.6.18. It also shows that our design point leads to the good solution for high efficiency design. It is seen that, both of the modified designs can maintain a high area-time efficiency in comparison with other designs.

Table II Comparisons with Different Window Size($N \times N$=22 × 22)

N_I : the number of data input port

N_{II} : the number of internal data bus

N_{PE} : the number of processor elements

N_{REG} : the number of registers for DFC

T_P : computation time in multiple frame mode

TRP : operation throughput

E_{DT} : data transfer efficiency

E_P : parallel processing efficiency

AT_P : area-time efficiency

$K \times K$	Methods	N_I	N_{II}	N_{PE}	N_{REG}	T_P	TRP	E_{DT}	E_P	AT_P
3 × 3	Systolic1	2	0	9	10	809	4.45	30 %	49%	8,899
	Systolic2	3	0	9	66	443	8.13	36%	90%	9,835
	Systolic3	1	0	9	36	529	6.81	91 %	76%	8,570
	MSPA 1	**9**	**3**	**72**	**0**	**66**	**54.55**	**81 %**	**76%**	**4,752**
	MSPA 2	**9**	**4**	**63**	**44**	**66**	**54.55**	**81 %**	**87%**	**4,738**
4 × 4	Systolic1	3	0	16	21	738	7.83	22%	49%	14,908
	Systolic2	4	0	16	88	418	13.82	29%	86%	14,045
	Systolic3	1	0	16	51	551	10.48	88%	66%	18,183
	MSPA 1	**6**	**4**	**96**	**0**	**88**	**65.64**	**92 %**	**68%**	**8,448**
	MSPA 2	**6**	**5**	**80**	**66**	**88**	**65.64**	**92 %**	**82%**	**8,202**
5 × 5	Systolic1	4	0	25	36	673	12.04	18%	48%	21,671
	Systolic2	5	0	25	110	396	20.45	24%	82%	18,612
	Systolic3	1	0	25	64	573	14.14	84%	57%	21,659
	MSPA	**5**	**5**	**90**	**0**	**110**	**76.63**	**88 %**	**82%**	**9,900**
8 × 8	Systolic1	7	0	64	105	514	28.02	13%	44%	43,690
	Systolic2	8	0	64	176	330	43.64	18%	68%	32,736
	: Systolic3	1	0	64	91	638	22.57	76%	35%	52,444
	MSPA	**3**	**3**	**120**	**0**	**176**	**81.82**	**92 %**	**68%**	**21,120**
12 × 12	Systolic1	11	0	144	253	386	45.14	11%	31%	75,116
	Systolic2	12	0	144	264	242	72.00	17%	50%	47,626
	Systolic3	1	0	144	99	727	23.97	67%	17%	119,083
	MSPA	**2**	**2**	**121**	**0**	**264**	**66.00**	**92 %**	**55%**	**31,944**
16 × 16	Systolic1	15	0	256	465	354	35.44	9%	14%	123,546
	Systolic2	16	0	256	352	154	81.45	20%	32%	50,265
	Systolic3	1	0	256	75	815	15.39	59%	6%	229,015
	MSPA	**2**	**2**	**49**	**0**	**352**	**35.64**	**69 %**	**73%**	**17,248**

Table III Area small design with various value of N_{PE} $(N = 22)$

measure	Small Window Operation							Large Window Operation			
	K=3				K=4			K=5		K=8	
N_I	6	5	4	3	6	5	4	5	5	3	3
$N_{WPE}(C)$	4	3	2	1	3	2	1	2	1	4	2
N_S	5	7	10	20	7	10	19	9	18	4	8
N_{PE}	36	24	18	9	48	32	16	36	18	60	30
N_{REG}	44	44	44	44	66	66	66	88	88	154	154
T_P	120	168	240	480	168	240	456	225	450	256	512
TRP	30.00	21.43	15.00	7.50	34.38	24.07	12.67	36.00	18.00	56.25	28.13
E_{DT}	67%	58%	50%	34%	48%	40%	27%	43%	22%	63%	32%
E_P	83%	89%	83%	83%	72%	75%	79%	100%	100%	94%	94%
AT_P	5376	5510	6432	8544	10282	10848	13315	12060	16020	23245	31130

Table IV Speed Fast design with various value of N_{pe} $(N = 22, K = 16)$

measure	Value of N_{pe} for Speed Fast Design	
	$N_{pe} = 2$	$N_{pe} = 3$
N_I	4	6
N_{PE}	98	147
N_{REG}	330	330
T_P	190	134
TRP	66.02	93.61
E_{DT}	64%	60%
E_P	67%	64%
AT_P	26896	28542

Chapter 7

Conclusion

We have presented a novel architecture of a Memory Shared Processor Array for low-level algorithms in digital signal/image processing. A systematical design methodology for MSPA has also been established for practical VLSI implementation. In the design methodology, we proposed not only theoretical concepts on operation scheduling and PEs resource allocation but also a series of practical theorems. Moreover, effective design method for data format conversion is also presented, which is used to minimize the number of registers in VLSI implementation. Based on the basic MSPA concept, we proposed Window-MSPA architecture which is applied especially for window image processing,

MSPA and Window-MSPA inherit the advantages of both systolic arrays and common bus architectures. These architectures may be optimal candidates for limited I/O ports systems in some applications. By proposed array processor architecture and design methodology, some applications such as matrix-matrix multiplication, bit-serial vector-matrix multiplication, convolution-like algorithm, FIR, IIR, and etc, have been proposed. These design examples demonstrate the reduction of hardware and the achievement of high parallel efficiency in VLSI realization. Proposed design concept can be applied to either data-level or bit-level parallel algorithms.

Before we describe our future work, it is relatively important to view the architecture developing trend. Currently, it is commonly recognized that continuous advances in semiconductor processing allow the integration of increasing functionality into single-chip microprocessors. Today's high-performance microprocessors support several million transistors and include multiple functional units and large on-chip memories. Effective device densities and IC capacity improve at an exponential rate. We all now quite recognize the progress of microprocessors where operational performances increase by roughly 60 % per year while the number of gates increases by 25% per year. With 3 million gates available for today's modern microprocessors, we can expect to have over 12 million gates available by the end of the century [89]. Although supercomputers have become increasingly specialized in their pursuit of higher performance, very specializations in architecture that were originally responsible for higher performance are beginning to threaten supercomputing's very existence. Because the difference from application to application and insufficient commonality among applications, contemporary microprocessor architectures are poorly matched to most of the applications they run. It is obvious to employing specialized coprocessors for high-performance graphics, video processing, signal processing, etc..

If we take a look at ASIC field, we find that the heavy demands on processing power and data throughput for real time image/signal processing applications are usually mapped onto function in ASIC. This leads to compact realization with a minimum silicon area, enough for low cost mass production. The disadvantage of ASIC is the lack of flexibility. The adopted solutions may require a redesign even for minor changes or extensions of the algorithms involved. In DSP application, programmable DSP with software is widely used in order to explore the flexibility of ASIC DSP. However, the far more flexible implementation of these applications using programmable digital signal processors(DSPs) occupy a significantly large silicon area. An adaptive implementation is 6-7 times as efficient as the implementation in software on a programmable DSP [90] [91]. Therefore, how to combine the advantages and overcome the disadvantages of both general purpose microprocessor and ASIC will provide us a research topic. According to this motivation, we will extend our MSPA architecture to a system on silicon. We may call it as Multithread Memory Sharing Processor Array System (MMSPAS) as shown in Fig.7.1.

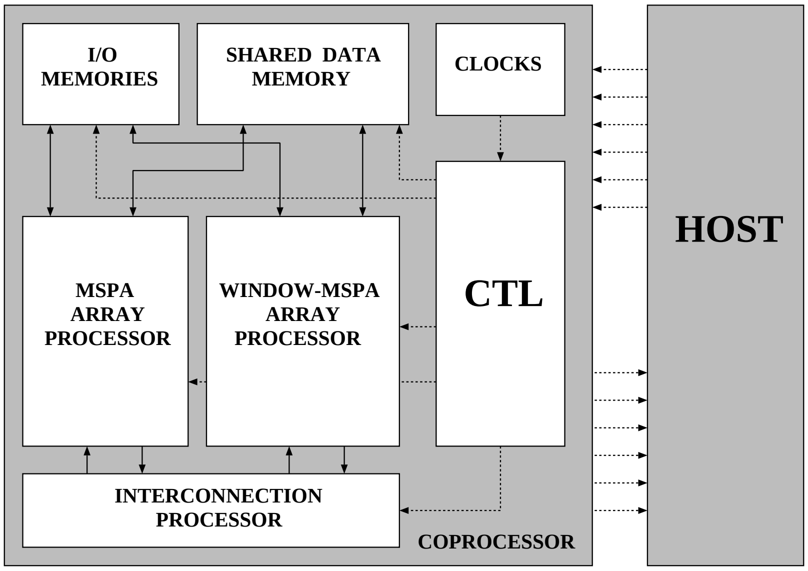


Fig. 7.1. Multithread MSPA System Architecture

It is noticed that unimicroprocessor is developing towards multiprocessor architecture while at the same time ASIC DSP is developing towards reconfigurable architecture. In order to meet the requirements from more and more complex applications, it is the era to compact the advanced techniques together as much as possible within the limit of process technology. The techniques such as vector processor, superscalar, SIMD, MIMD, have their own good features, but they are difficult to overcome their constraints. For example, vector pipelined processors is relatively easy to improve the vector processing rate but far more difficult to improve the scalar processing rate by the same amount, which makes vector multiprocessors more SIMD in character as the years pass [92].

As an solution to this problem, to narrow the gap between supercomputers and general purpose system is a credible direction. This means that processor architectures are required to change from uniprocessors to multiprocessors in order to provide higher performance. For example, Intel Pentium 32 bit 75/90/100/120/133/150/166/200 are Superscalar Architecture, which are uniprocessors. But Pentium-Pro takes on a dynamic execution microarchitecture. It is scalable up to 4 processors and 4 GB memory. Therefore, combination among current advantageous techniques will give a good solution to a high cost performance development in processor architecture field. As another example, PowerPC architecture exploited the technology that existed at both Motorola and IBM, which represents an exciting evolution within the computer industry [93]. The RISC Single Chip (RSC) microprocessor [94] became the base design for PowerPC 601. The Motorola 88110 microprocessor bus interface formed the basis of the development of the 601 bus interface [95]. Based on the PowerPC development, PowerPC 601-604 and 620, new architecture called Bull PowerScale architecture has been also developed to increase performance. The PowerScale architecture is a symmetrical multiprocessor (SMP) architecture which supports horizontal growth, by adding processors.

As a conclusion, the proposed MSPA architecture can be extended to a slave processor as shown in Fig. 7.1, where the array processors are a heart of a processor system. Suppose the host is a Sparc SIMD processor, then, some heavy data processing task can be performed by a large MSPA array processor. If the host is a MIMD processing, then, this processor system concept is valuable to process large amount of matrix multiplications, various kind of transformation in image processing and computer graphics applications. The MSPA array processor may be used to perform matrix multiplication while at the same time the Window-MSPA can provide some window processings. Since the MMSPAS gives a very large room to introduce new and current techniques, real-time high speed parallel processing can be derived. At the same time, the system is possible to be developed to reconfigurable processor system due to the flexibility feature of MSPA/Window-MSPA. By combining the high parallelism of proposed MSPA/Window-MSPA architectures with high performance CPU core, interconnection processor, current hybrid technology and MCM manufacture technique, complex system integration of DSP/ASIC for signal/image processing and multimedia communication system becomes capable to be implemented.

Bibliography

[1] C.K.Yuen, K.G. Beanchamp, G.P.S. Robinson ”Microprocessor systems and their application to signal processing” Academic Press, 1982.

[2] E.E. Swartzlander, Jr ”VLSI signal processing systems” Kluwer Academic Publishers, 1986.

[3] Shem-Tov Levi ”Real-time systems design” MaGraw-Hill, 1990.

[4] B.A. Bowen, W.R. Brown ”VLSI system design for digital signal processing” Englewood Cliffs: Prentice-Hall, 1982.

[5] S.Y.Kung, H.J. Whitehouse, T.Kailath ”VLSI and modern signal processing” Prentice-Hall, Inc. 1985.

[6] J. Schonfeld, P. Pirsch: ”Compact Hardware Realization for Hough Based Extraction of Line Segments in Image Sequences for Vehicle Guidance”. Proc. ICASSP93, p I-397 - I-401.

[7] J. Kneip, J.P. Wittenburg, M. Berekovic, K. Ronner, P. Pirsch:” An Algorithm Adapted Automous Controlling Concept for A Parallel Signal-Chip Digital Signal Processor” VLSI Signal Processing VIII Edited by Takao Nishitani, Keshob K. Parhi Published under the sponsorship of the IEEE Signal Processing Society, pp.41-50.

[8] S.M. Bhandarkar, and H.R. Arabnia, “Parallel Computer Vision on a Reconfigurable Multiprocessor Network”, *IEEE Trans on Parallel and Distributed Systems* , Vol.8, No.3, pp.292-309, March, 1997.

[9] D. Audet, Y. Savaria, and N. Arel, “Pipelining Communications in Large VLSI/ULSL Systems”, *IEEE Trans. on Very Large Scale Integration (VLSI) System*, Vol.2, No.1, March 1994.

[10] Ravi Shankar, Eduardo B. Fernandez, “VLSI and Computer Architecture” *VLSI Electronics Microstructure Science* Vol.120, pp.319-322.

[11] E.E. Swartzlander, V.K. Jain, and H. Hikawa, “A Radix-8 Wafer Scale FFT Processor”, *Architectures and Circuits*.

[12] F. Lombardi, Y.-N. Shen, and J. Muzio, "FFT Architecture for WSI with Concurrent Error Detection and Fault Location", *IEE Proceedings-E* , Vol.139, No.1, pp.13-20, Jan. 1992.

[13] C.M. Chang and S.L.Lu, "Design of a Static MIMD Data Flow Processor Using Micropipelines", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* , Vol.3, No.3, pp.370-403, Sep. 1995.

[14] R.F. Sproull, I.E. Sutherland, and C.E. Molnar, "Counterflow Pipeline Processor Architecture", Sun Microsystems Laboratories, Inc. SMLI TR-94-25, April, 1994.

[15] Kai Hwang, Faye A. Briggs "Computer Architecture and Parallel Processing" McGraw-Hill Book Company, 1985.

[16] A. Lafage and F. Jutand, "I/O Bandwidth Optimization of VLSI Architectures for Matrix Product-like Algorithm", IEEE, 0-7803-0946-4/93, pp.353-356, 1993.

[17] R.M. Owens, T.P. Kellier, and M.J. Irwin, "Building High Performance Signal Processors Cheaply and Quickly" .

[18] M. Borah, C. Nagendra, R. Owens, and M.J. Irwin, "The MGAP: A High Performance, User Programmable, Multifunctional Architecture for DSP", from internet, Copyright by the Association for Computing Machinery, Inc. 1993.

[19] R.M. Owens, M.J. Irwin, T.P. Kelliher, and M, Vishwanath, "Implementing a Family of High Performance, Micrograined Architecture",from internet, copyright by IEEE 1994.

[20] R.S. Bajwa, R.M. Owens, and M.J. Irwin, "A Massively Parallel, Micro-grained VILS Architecture",from internet, copyright of IEEE 1994.

[21] F.F. Engmond and E. F. Deprettere, "Regular Processor Array with Data-Dependent Dependencies".

[22] M.H. Sunwoo, S.Ong, B. Ahn, and K. Lee, "Design and Implementation of a Parallel Image Processor Chip for a SIMD Array Processor", *International Conference on Application-Specific Array Processors*, IEEE, 1995.

[23] H.M. Chang, M.H. Sunwoo, and T.H. Cho, "Implementation of a SliM Array Processor", *IEEE, Proceedings of IPPS'96* , pp.771-775, 1996.

[24] R.N. Mahapatra, A. Kumar, and B.N. Chatterji, "Performance Analysis of 2-D inverse Fast Cosine Transform Employing Multiprocessors", *IEEE Trans. on Signal Processing*, Vol.45, No.5, May, 1997.

[25] J. Bae and V.K. Prasanna, "Synthesis of VLSI Architecture for Two-Dimensional Discrete Wavelet Transforms", *International Conference on Application-Specific Array Processors*, IEEE, 1995.

[26] P. Lenders and S. Rajopadhye, "Synthesis of Multirate VLSI Arrays", *International Conference on Application-Specific Array Processors*, IEEE, 1995.

[27] A. Bugeja and W. Yang, "A Reconfigurable VLSI Coprocessing System for the Block Matching Algorithm", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* , vol.5, No.3 Sep. 1997.

[28] Arun K. Somani, "A New Parallel Supercomputer Architecture" , by internet, 1995.

[29] Y.S. Chen, S.D. Wang, and K.C. Su, "Synthesizing Efficient VLSI Array Processors from Iterative Algorithms by Excluding Pseudo-Dependences", *IEICE Trans. Fundamentals* , Vol/E78-A, No.10, pp.1369-1379, Oct. 1995.

[30] R. Baghaie and I. Hartimo, "An Efficient New Systolic Architecture for the Solution of Discrete Fourier Transform". pp. 453-461.

[31] K.J. Jones, "Parallel DFT Computation on Bit-serial Systolic Processor Array", *IEE Proceedings-E* , Vol.140, No.1, pp.10-18, Jan. 1993.

[32] V. Boriakoff, "FFT Computation With Systolic Arrays, A New Architecture", IEEE Trans. on CAS-II: Analog And Digital Signal Processing, Vol. 41, No. 4, pp.278-284, April 1994.

[33] Sung Bum Pan and Rae-Hong Park, "Unified Systolic Arrays for Computation of the DCT/DST/DHT", *IEEE Trans. on CAS for Video Technology* , Vol.7, No.2, April, 1997.

[34] L. Bierens and E. Deprettere, "Interleaving Fast Convolution of 2 Signals and Multirate Analysis with Applications to Radar Image".

[35] C. Chakrabarti and L. Lucke, " Highly Concurrent Architecture for Recursive Median Filters".

[36] V.K. Jain, L. Lin, S.A. Wadekar, and J. M. Wills, "DSP Coprocessor Cell for Systolic Arrays". pp.480-488.

[37] C. Chakrabarti, M. Vishwanath, and R. M. Owens, " Architecture for Wavelet Transforms".

[38] C. Nagendra, M.J. Irwin, and R.M. Owens, "Digit Pipeline Discrete Wavelet Transform", from internet, copyright of IEEE 1994.

[39] M.A. Aboelaze, D.L. Lee, B.W. Wah, “Two-Dimensional Digital Filtering Using Constant-I/O Systolic Arrays”, *Proc. IEEE Int'l Symp. on Circuits and Systems* , 1993.

[40] Will Moore, Andrew McCabe, and Roddy Urquhart, “Systolic Arrays”, Adam Hilger, Bristol and Boston, 1987.

[41] Weijia Shang and Jose A.B. Fortes, “Time optimal linear schedules for algorithms with uniform dependencies”, *International Conference on SYSTOLIC ARRAYS* , pp.393-402, May 1988.

[42] M. A. Sid-Ahmed, “A Systolic Realization for 2-d D Digital Filters”, *IEEE Trans. on Acoustics, Speech and Signal Processing*, Vol.37, No.4, pp.560-565, April 1989.

[43] V. K. Prasanna Kumar and Y. C. Tsai, “On Mapping Algorithm to Linear and Tolerant Systolic Arrays”, Proc. Int'l Conf. on Systolic Arrays, 1988.

[44] M. A. Aboelaze, B. W. Wah,”A Processor Array with Bounded I/O Ports for computing Transitive Closures” *Proc. Parallel and Distributed Computing*, Academic Press, 1995.

[45] D. Lavenier, F. Raimbault, P. Frison, “I/O and Computation Overlap on SIMD Systolic Arrays”, Publication internet 770, Nov. 1993.

[46] R.H.Kuhn, “Optimization and interconnection complexity for parallel processors, single stage networks and decision trees”, Ph.D dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Oct. 1980.

[47] D.I. Moldovan, “On the analysis and synthesis of VLSI algorithms”, *IEEE Transactions on Computers* , vol.C-31, pp.1121-1126, Nov. 1982. G.J Li and B. W. Wah, “The Design of Optimal Systolic Arrays”, *IEEE Trans. on Computers*, Vol. C-34, No.1, pp.66-77, Jan. 1985

[48] P. Lee and Z.M. Kedem, “Synthesizing Linear Array Algorithm from Nested For Loop Algorithms”, *IEEE Trans. on Computers*, Vol.37, No.12, pp.1578-1598, Dec. 1988.

[49] Weijia. Shang, and Jose,A.B. Fortes, “Time Optimal Linear Schedules for Algorithm with Uniform Dependencies”, *IEEE Trans. on Computers*, Vol.40, No.6, pp.723-742, June, 1991.

[50] L. Kazerouni, B. Rajan, and R.K. Shyamasundar, “Mapping Linear Recurrences onto Systolic Arrays”, *IEEE Proc. of IPPS'96* , pp.891-897, 1996

[51] Y. Wong and J.M. Delosme, “Optimization of Computation Time for Systolic Arrays”, *IEEE Trans. on Computers*, Vol.41, No.2, pp.159-177, Feb. 1992.

[52] K. Ganapathy and B.W. Wah, "Optimal synthesis of algorithm-specific lower-dimensional processor arrays ", *IEEE Trans. on Parallel and Distributed System, IEEE*, 1996.

[53] F. Lorenzelli and K. Yao, "A Unified Systolic Design Technique Based on Integral Matrix Theory".

[54] Y. Saouter, "Optimization of Number of Processors in VLSI Arrays", Publication internet 850, Sep. 1994.

[55] P. Corbett and R. Hartley, "Designing Systolic Array Using Digit-Serial Arithmatic" IEEE Trans. on CAS-II: Analog and Digital Signal Processing, Vol. 39, No. 1, Jan. 1992.

[56] P.E.R. Lippens, J.L. Van Meerbergen, A. Van Der Werf, W.F.J. Verhaegh, Anf B.T. McSweeney, "Memory synthesis for high speed DSP applications", *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference*, pp.11.7/1-4, May 1991.

[57] D.M. Grant and P.B. Denyer, "Address generation for array access based on modulus m counters", *EDAG. Proceedings of the European Conference on Design Aytomation*, pp.118-122, Feb. 1991.

[58] H. Kunieda and K. Hagiwara, "Effective processor array architecture with shared memory", *IEEE, APCCAS'94* , pp.113-118, 1994.

[59] M.F.X.B. Van Swaaij, F.V.M. Catthoor, and H.J. De Man. "Nonlinear transformations for high level regular array ASIC synthesis" in *Journal of VLSI Signal Processing* , 4(4):259-68, Nov. 1992.

[60] W. Shang and Jose A.B. Fortes, "Independent Partitioning of Algorithms with Uniform Dependencies", *IEEE Trans. on Computers* , Vol.41, pp. 190-206, No.2, Feb 1992.

[61] D. Moldovan and Jose A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", *IEEE Trans. on Computers* , Vol.C-35, No.1, pp.1-12, Jan. 1986.

[62] Kesami, Hagiwara,"Synthesis Method for ASIC Processor Address Generation Unit of Regular Algorithm", Master Dissertation, Feb. 1994.

[63] Tohoru Ishizaki, "Automatic Synthesis for Memory Sharing Array Processor", Master Dissertation, Feb. 1994.

[64] P.-Z. Lee and Z.M. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms" in *IEEE Transactions on Computers* , vol.C-37, pp.1578-1597, Dec.1988.

[65] H. Kunieda, Y.S. Liao, D.J. Li, K. Ito "Automatic design for Bit-serials MSPA architecture" in *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95* , pp.27-32, 1995

[66] Dongju Li and Hiroaki Kunieda, "Automatic Synthesis of a Serial Input Multiprocessor Array", *IEICE TRANS. Fundamentals* , Vol. E79-A, No.12, pp.2097-2105, Dec. 1996.

[67] K.K. Parhi. "Systematic synthesis of DSP data format converters using lifetime analysis and forward-backward register allocation" in *IEEE Transactions on Circuits and Systems-II:Analog and Digital Signal Processing* , Vol.39, No.7, pp.423-440, 1992

[68] M. Hatanmian and G.I. Cash. "A 70-MHz 8-bitx8-bit parallel pipelined multiplier in 2.5-μm CMOS ". In *IEEE Journal of Solid-State Circuits* , Vol. SC-21, No. 4, pp.505-513, August 1986

[69] LYON, R.E. "Two's complement pipeline multipliers" in *IEEE Transactions on Communications* , COM-24, pp.418-425, April 1976

[70] SMITH, S.G. "Signal/parallel automultiplier" in *Electron. Letters* , Vol.23, No.8, pp.413-415, 1987

[71] B. Bi and E.B. Jones. "High-performance bit-serial adders and multipliers" in *IEEE Proc. Circuit, Devices and Systems* , Vol.139, No.1, pp.109-113, Feb 1992

[72] V.K. Prasnna Kumar and Yu-Chen Tsai, " On Synthesizing Optimal Family of Linear Systolic Arrays for Matrix Multiplication", *IEEE Trans. on Computers* , Vol.40, No.6, pp.770-774, June 1991.

[73] J. Gong, D.D. Gajski, and A. Nicolau, "Performance Evaluation for Application-Specific Architectures", *IEEE Trans. on Very Large Scale Integration (VLSI) System* , Vol.3, No.4, pp.483-490, Dec. 1995.

[74] Dongju Li and Hiroaki Kunieda, "Memory Sharing Processor Array (MSPA) and its Design Methodology" *ICCI'96 (1996 Int. Conf. on Computing and Information)*, June, 1996.

[75] K.Ganapathy and B.W. Wah, "Optimizing general design objectives in processor-array Design" in *Pro. IEEE Int'l Parallel Processing Symposium* , pp.295-302, April 1994.

[76] K. Ganapathy and B.W. Wah, "Optimal Design of Lower Dimensional Processor Arrays for Uniform Recurrences", *Proceedings International Conference on Application-Specific Array Processors* , IEEE, August 1992.

[77] Dongju Li and Hiroaki Kunieda, “Memory Sharing Processor Array(MSPA) Architecture”, *IEICE TRANS. Fundamentals* , Vol. E79-A, No. 12 pp.2086-2096, Dec. 1996.

[78] Y.L. Jeang. Y.C. Hsu, J.F. Wang, and J.Y. Lee, “High Throughput Pipelined Data Path Synthesis by Conserving the Regularity of Nested Loops”, IEEE, 1063-6757/93, pp.450-453, 1993.

[79] S.Y. Kung, “VLSI Array Processors”, Englewood Cliffs, NJ: Prentice Hall, 1988.

[80] J. Vlontzos. ”A wavefront array processor using dataflow processing elements” *In intl’ Conf. on Supercomputing* Athens, Greece, June,1987.

[81] T. Komarek and P. Pirsch, “Array Architectures for Block Matching Algorithm”, *IEEE Trans. on CAS* , Vol.36, No.10, pp.1301-1308, 1989.

[82] H.T.Kung. ”Why systolic architecture?” *Compt. Mag* , 15(l), pp.37-46, Jan.1982.

[83] H.T.Kung, R.L.Picard. ”One-dimensional systolic array for multidimensional convolution and resampling” *VLSI for Pattern Recognition and Image processing* , Fu K.S.(editor), Spring-verlag, pp.9-24, 1984.

[84] P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi, “ Parallel Implimentation of the Full Search Block Matching Algorithm for Motion Estimation”, *International Conference on Application-Specific Array Processors*, IEEE, 1995.

[85] William K.Pratt. ”Digital Image Processing” *A Wiley-interscience Publication John Wiley & Sons* Inc, Sun Microsystems, Inc. Mountain View, California, 1991.

[86] C.I.Brown, N.A. Thacker and R.B.Yates. “A Data Path for the Efficient Implementation of Sparse Matrix Multiplication. By Internet.

[87] C.H.Hsieh and T.P.Lin, ”VLSI Architecture for Block Matching Motion Estimation Algorithm”, *IEEE Trans on Circuits and Systems for Video Technology* , Vol. 2, No.2, pp. 169-175, June 1992.

[88] Thomas KOMAREK and Peter PERSCH, ”Array Architectures for Block Matching Algorithms,” *IEEE Trans* , CAS-36, No.10, pp.1301-1308, Oct. 1989.

[89] Andre DeHon “DPGA-Coupled Microprocessors: Commodity ICs for the Eearly 21st Century”, *T.I.T. Transit Project* , Transit Note No.100, Jan. 1994.

[90] J. Kneip, J.P. Wittenburg, M. Berekovic, K. Ronner, and Peter Pirsch, “An Algorithm Adapted Autonomous Controlling Concept for a Parallel Single-Chip Digital Signal Processor”, *IEEE 0-7803-2612-1/95* , pp. 41-50, 1995.

[91] P. Pirsch, N. Demassieux, W. Gehrke, "VLSI Architecture for Video Compresion - A Survey", *Proc. IEEE*, Vol.83, No.2, pp. 220-246, Feb. 1995.

[92] Burton Smith, "Opportunities for Growth in High Performance Computing", Tera Computer Company, by internet.

[93] Paul Bennett and Angelo Ramolini, "The PowerScale Architecture: A Technical Overview", Jan. 1995.

[94] Moore,C.R., Balser,D.M., Muhich,J.S., and East R.E.., "IBM Signal Chip RISC Processor (RSC)", *Proceedings of the 1992 International Conference on Computer Design* , 1992.

[95] Diefendorff, K. and Allen, M., "Organization of the Motorola 88110 Superscalar RISC Microprocessor", *IEEE Micro* , Vol.12, No.2, pp.40-63, April 1992.

Publications Related to the Dissertation

Full Papers

1. **Dongju Li** and Hiroaki Kunieda, "Memory Sharing Processor Array(MSPA) Architecture" *IEICE TRANS. Fundamentals*, Vol. E79-A, No.12, pp.2086-2096, Dec. 1996.

2. **Dongju Li** and Hiroaki Kunieda, "Automatic Synthesis of a Serial Input Multiprocessor Array" *IEICE TRANS. Fundamentals*, Vol. E79-A, No.12, pp.2097-2105, Dec. 1996.

3. **Dongju Li**, Tsuyoshi Isshiki, and Hiroaki Kunieda, "Array Processor Synthesis for Image Window Processing", *Submitted to IEEE Trans. CAS-II*, August, 1997.

4. Li, Jiang, **Dongju Li**, Shintaro Haba, Chawalit, and Hiroaki Kunieda, "Dedicated Design of Motion Estimator with Bits Truncation Fast Algorithm" *IEICE TRANS. Fundamentals, Special Section on Digital Signal Processing*, Vol.E79-A, No.8, August, 1998. conditional accepted.

International Conference

1. Hiroaki Kunieda, Yu-shong Liao, **Dong-ju Li**, Kazuhito Ito, "Automatic design for Bit-serials MSPA architecture", *Proceedings of ASP-DAC'95 /CHDL'95 /VLSI'95*, pp.27-32, Japan, August 1995.

2. **Dongju Li**, Kazuhito Ito, Hiroaki Kunieda, "Memory Sharing Processor Array(MSPA) and its Design Methodology", *Proceedings of The Eighth International Conference on Computing and Information*, Canada, June, 1996.

3. **Dongju LI, Hiroaki Kunieda**, "Programmable Design for Memory Sharing Processor Array (MSPA)", *Proceeding of IEEE, ISCAS-97*, Hong Kong, June 9-12, 1997.

4. **Dongju LI, Hiroaki Kunieda**, "Array Architecture and Design for Image Window Operation Processing ASICs", *ISCAS-98*, to appear.

5. Li, Jiang, **Dongju Li**, Shintaro Haba, Chawalit, and Hiroaki Kunieda, Towards One Chip HDTV MPEG2 Encoder LSI" *Custom Integrated Circuit Conference CICC'98*, Santa Clara, California, May 11-14, 1998, to appear.

Domestic Conference

1. **Dongju Li**, Kazuhito Ito, Hiroaki Kunieda, "ASIC Design Methodology for MSPA Architecture and its application to Data-serial Matrix Multiplier", IEICE Technical Report on VLD, VLD95-129, ICD95-2 29(1996-03), pp.85-92, 1996.

2. **Dongju Li, Hiroaki Kunieda, "Programmable Design for Memory Sharing Processor Array (MSPA)"**, *Technical Report of IEICE of the Institute of Electronics, Information and Communication Engineers* , Technical Report of IEICE, VLD96-51 (1996-10), pp.09-16, 1996.