A Globally Asynchronous Locally Synchronous Configurable Array Architecture for Algorithm Embeddings

Bo Gao

A thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science

University of Edinburgh

August, 1996

A thesis for the degree of Doctor of Philosophy

Department of Computer Science

University of Edinburgh

I declare that this thesis has been composed by myself and that the work described within this thesis is entirely my own except where clearly indicated otherwise in the text.

Bo Gao.

Abstract

Advanced VLSI/ULSI technologies have made it possible to realise parallelism and pipelining processing principles at affordable cost. One of the consequences is that more and more algorithms are now directly implemented in hardware. The configurable hardware algorithm approach has the potential to combine the performance of hardware algorithms and the flexibility of software algorithms at the user level. On the other hand, system timing design problems become one of the determining factors on design complexity, correct system function and high performance. This timing problem plays an even more important role in configurable systems. There are two typical system timing control design approaches, the synchronous timing design and the asynchronous timing design. This thesis investigates and demonstrates the idea and feasibility of applying asynchronous timing control at the system level and synchronous timing control to system composition modules, namely a Globally Asynchronous Locally Synchronous (GALS) design approach, for very large scale configurable hardware algorithms.

A systematic approach has been adopted in this thesis to develop a configurable GALS array architecture. With the analysis of general algorithmic properties, a novel multiple threads computation model consisting of an architecture with a pool of programmable hardware operators having configurable interconnections and a GALS system timing control structure is first established. The multiple threads computation model bridges algorithms and the architecture for efficient algorithm embeddings. The GALS timing control makes this threads model practical. A novel and fast event-driven GALS data transfer interface is developed upon which a bit-serial configurable GALS array system for algorithm embeddings is designed. Some good average performance results are obtained with a polynomial evaluation algorithm embedded as a frame buffer. The work on the GALS system timing design principle can be easily extended to the design of general GALS systems.

Acknowledgements

The work described in this thesis was inspired by Dr Thomas Kean's work on cellular Configurable Array Logic (CAL). Many useful suggestions were also obtained from direct discussions with Dr Thomas Kean. Dr David Rees has very patiently supervised me and given me a lot of encouragement all the way through my research work. He also carefully proof read the draft of this thesis and has made many useful comments. Special thanks also go to Professor David Kinniment from University of Newcastle upon Tyne who has helped me to clarify the synchronisation issue discussed in this thesis.

I would like to thank the department of Computer Science of Edinburgh University where an excellent research environment and computing facilities are provided, and from where I have obtained substantial knowledge on computer architectures and skills on programming.

I would also like to acknowledge the support of Sino-British Friendship Scholarship Scheme which made my research in Britain a reality.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we investigate the issue of implementing algorithms directly in configurable hardware architectures (configurable hardware algorithms). We aim to establish a proper computation, architecture model and system timing control strategy for configurable hardware algorithms, and to construct a system based on the models established. A configurable system can be used as an attached subsystem to a computer where it can be configured to run a computation intensive task. Therefore it can be regarded as an algorithm memory and data are processed on-the-fly when they flow through such an algorithm memory so as to achieve high computation throughput on the task and improve the system performance of the host computer. It can also be used as a testbed to test high level algorithm designs at hardware level. This is of particular interest to software-hardware co-designs where a complex algorithm is partly solved by software and partly by hardware to meet some special criteria such as real-time response and the cost.

There are several driving factors which inspired this research when we took a brief look at the evolution history of computing systems and how complex computation problems are solved.

1.1 Computing Systems

From the hardware point of view, a traditional computing system (a von Neumann computer) in general consists of a computation part (a central processing unit or CPU), a data storage part (memory), a control part, and a communication structure which connects these parts together. Each of these hardware parts implements some essential functions, such as basic arithmetic and logic operations in an Arithmetic and Logic Unit (ALU), addressed data read/write in a memory, or instruction decoding in a control part. A complicated computation task is decomposed into a sequence of essential functions directly supported by these parts. Functions which require one hardware part, for example the ALU, have to be evaluated one after another by sharing the same hardware in the time domain (sequential model).

The evolution of computing systems is two fold. Firstly, the performance and reliability of hardware components have been greatly improved over the years. Secondly, hardware costs are decreasing rapidly. Hence, it is possible to design and implement more complicated and faster computing systems by exploring novel system architectures different from the traditional sequential model to solve many difficult computation problems quickly which were almost impossible or very slow to do before.

1.2 Algorithms

An algorithm defines a computational method which solves a target problem in finite steps for all of the possible inputs of the problem. Different algorithms may be designed to solve a given problem. Performance (speed and hardware resource requirements) of these algorithms, however, will probably be very diverse. The design of efficient and high performance algorithms is highly system dependent.

1.2.1 Software Solutions

If a complex algorithm is to be solved on a computing system with limited hardware resources, for example one ALU which supports only one arithmetic or logic operation at a time point, a software solution is required to decompose complex functions, procedures, and data access operations defined in the algorithm into a proper sequence of basic operations supported by the existing hardware. A control sequence has to be generated while the algorithm is decomposed. The sequence of the decomposed operations will be executed in the system in accordance with the control sequence to obtain the required results.

Software solutions are the basis for the efficient handling of problems on a von Neumann type computing system. A von Neumann computer is a general purpose architecture developed with restrictions on hardware costs. No data dependencies of any algorithms can be reflected in the von Neumann architecture. The control sequence generated from an algorithm holds all the required data dependencies for solving a problem on a von Neumann computer. The performance of von Neumann computers replies heavily on the improvement of the single CPU operation speed. But as a matter of fact, the pace of improvements on the operation speed of hardware devices is usually behind requirements. The performance of von Neumann computers is inherently limited by the sequential computing bottle-neck because many properties of algorithms are simply ignored. New computing architectures and models are the ultimate choice which can take full advantage of many algorithmic properties and can result in a real leap in the system performance with existing micro-electronic technologies.

1.2.2 Hardware Solutions

It is possible to implement high level algorithm specifications and schedule their control tasks directly in hardware (hardware algorithms) by eliminating the sin-

gle CPU bottle-neck with a properly established computation model, a system architecture, and a control scheme to achieve a high system performance.

It becomes evident, if we take the design evolution history of microprocessors as an example, that more and more computing functions are directly implemented in hardware. The only arithmetic operation implemented in the first microprocessor design was addition; any operations more complicated than addition had to be done by software methods. For example, the instruction set of a Zilog Z80 did not have multiplication which had to be realised as an algorithm when invoked. Then there have been lots of efforts made in implementing various multiplication algorithms as hardware multipliers [44,13,140,86,122,133,16,19,18]. In the subsequent generations of microprocessor designs, multiplication is eventually included in the instruction set. Hardware accelerators for multiplication had been integrated into the designs of many microprocessors, for instance, Intel 80486, Motorola MC68040, and INMOS T800. Coprocessor approaches which can further extend the CPU instruction set to trigonometric, logarithmic, exponential and other floating-point arithmetic instructions were developed, examples are Intel 80387 and Motorola MC68881 math coprocessor. More complicated algorithms, such as sorting and Fast Fourier Transform (FFT), were also implemented directly in hardware. The implication of this development is that it is now very practical to implement many algorithms directly in hardware which previously had to be done with software solutions and this is the route that we follow to develop high performance systems.

The coprocessor approach is one of the ways to implement many functions in hardware. However, this approach is still limited in that there is only small amount of hardware resource to be sequentially programmed in a user transparent way. Another common approach is by parallel processing where data dependencies of algorithms can be reflected to some degree in the actual hardware. Systolic arrays are one of the intensively studied hardware solutions to a class of regularly structured algorithms. Systolic algorithms are designed in such a way that adequate hardware resources are provided and data movements are properly scheduled at

the design stage to meet the entire optimized computation requirements instead of sequencing data through limited hardware blocks many times. Many Application Specific Integrated Circuits (ASIC) can also be classified as hardware solutions to specific application problems.

A hardware system can be classified as a hardware algorithm if it has a large proportion or all of the data dependence structures of a class of algorithms and can directly output required results after a finite latency upon the presence of valid inputs. Unlike software solutions, control sequencing and scheduling are all hard-wired in hardware algorithms.

There are two ways to design hardware algorithms. The ASIC design is a popular approach to implement a hardware algorithm exactly as the algorithm specification. Another way is to design a blank hardware system which can be configured by end users. By blank system we mean a system that does not perform any specific functions before it is configured. A user has the freedom to design an application algorithm for the system. The specification of the user's algorithm will be mapped into such a blank hardware system assisted by an algorithm mapping tool. The algorithm is said to be embedded into the system after it is configured according to the map generated from the mapping tool.

The ASIC approach provides some advantages in design and performance. It is relatively easy to automate ASIC design procedures by taking advantage of both special silicon architectures, such as gate-array, and particular data dependencies of the target applications. Algorithms implemented in ASICs are faster than their configurable counter parts. However, ASICs also suffer from some drawbacks.

1. Turn-around time: The design time for an ASIC based on the gate-array structure is faster compared with custom Very Large Scale Integrated Circuit (VLSI) designs since many Computer Aided Design (CAD) systems are available. However, they still have to go through a design phase, fabrication phase, test phase and shipping phase which often take at least 3 to 4 months

to complete. It may take more than one such a cycle to get a final correct design.

2. High cost: The cost of designing an ASIC may be high because of the above mentioned multi-phase design and manufacturing process and sometimes the relatively small number of chips produced.

3. Low user controllability: The algorithmic aspects of an ASIC design are usually determined at the design phase and cannot be changed easily once chips are fabricated. This low user controllability means that ASICs cannot accept any minor revisions without substantial efforts and costs. This low flexibility renders ASICs not suitable for applications at early stages of development which will often undergo modifications.

On the other hand, a configurable system provides a fast design turn-around time and flexibility for easy design modifications. As far as costs are concerned, it may appear that a single configurable chip may cost more because of the extra configuration logic. The average cost can be brought down when large numbers of chips are produced since they can be used for a wide range of applications. For example, Field Programmable Gate Array (FPGA) products with configurable architectures are growing rapidly in recent years because of these attractions. Therefore a configurable architecture for algorithm embeddings has advantages of high performance by running algorithms directly in hardware and the flexibility of configurable logic.

1.2.3 Parallelism and Pipelining

A proper computation model is required to transform a user defined algorithm into a form which can be embedded efficiently into a configurable hardware system to achieve the best performance on the algorithm.

Parallel and pipeline processing are effective ways to increase the system performance. Parallel processing is to process non-dependent computing tasks in a set of processing elements (PE) simultaneously so that the single CPU bottle-neck in von Neumann computers is eliminated. The concept of pipelining came from industrial assembly lines through which end products are consecutively assembled step by step. Each step will always be kept busy with partly assembled parts continuously fed from the previous step and will feed newly assembled parts to the next step. It is obvious that each product only goes through an assembly line once. After the first product is output from the assembly line, there will be the same number of products being assembled as the number of steps of the line, while each product is at a different stage of its final completion. Pipelining techniques in computing systems are exactly the same as assembly lines, simply replacing parts and products with intermediate and final data values, and steps with processing blocks as pipeline stages. The outstanding properties of pipeline processing techniques are highly efficient utilization of hardware resources and application level parallelism.

These concepts are not new to scientific researchers at all. People, including von Neumann, had already realised the potential of parallel and pipeline processing as early as in 1950's [131]. Many parallel and pipeline processing systems were developed 25 years ago. However, it may involve some substantial software programming work for users on some of these parallel computers because it is often the user's responsibility to determine the parallel properties of an algorithm, decompose and schedule (similar to the control sequence on von Neumann computers) process controls on these parallel systems. It is often difficult for a user to figure out some implicit or run-time parallel properties of a problem. Users are often required to be aware of the parallel architecture of a system in order to use it efficiently. The application of these techniques is relatively new in configurable architectures. A parallel and pipeline processing computation model for a configurable hardware algorithm system is established in this thesis. This will

enable the system to make use of the parallel properties, especially the run-time properties, of an embedded algorithm automatically.

1.3 Regular and Modular Architectures

Hardware algorithms are mostly based on regular VLSI architectures. Examples are array multipliers and matrix multipliers. Advanced VLSI technologies have made it possible to investigate and develop various novel architectures for a wide range of applications. One of the most popular regular architectures is the array architecture because it is very good for VLSI implementation and algorithm embedding. Array architectures can be classified according to the granularity of elements, programmability, array processing timing control strategy and array operation control strategy.

There are some common characteristics among VLSI arrays.

- Regularity: An array system is formed by the duplication of one or several very limited types of cells in a very regular way normally in a two dimensional plane. Some three dimensional arrays also exist. The interconnections between array elements are also distributed in a regular pattern.

- Simplicity: Although an entire VLSI array may consist of a large number of duplicated elements, the design complexity of a large array system is often proportional to the complexity of the building elements of the array.

- Scalability: Due to the duplication nature in arrays, it is very easy to shrink or expand the size of an array so as to fit it to certain particular application requirements.

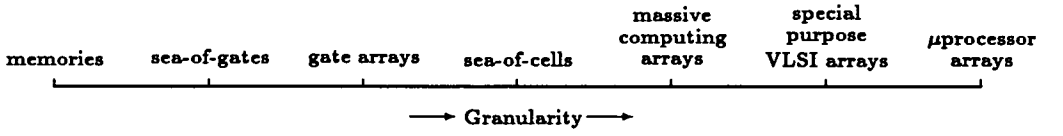


Figure 1–1: The spectrum of arrays

1.3.1 Granularity of Array Element

VLSI array architectures can be found spreading over a wide spectrum as shown in figure 1–1. VLSI arrays with the most fine-grained elements are memory arrays. The granularity of array elements is increased following sea-of-gates arrays, gate arrays, sea-of-cells arrays, massively parallel processing arrays, special purpose VLSI arrays, and very expensive microprocessor arrays where each processing element can be used as a CPU in a von Neumann computer. We choose the massively parallel processing array as the foundation for our configurable hardware algorithm architecture. This is a good compromise of system functionality, flexibility, size and rational cost.

1.3.2 Array Configurability

Array configurability reflects the flexibility with which an array system can be applied to embed just one or a class of algorithms, that is, the generality of a system. There are many different technologies to choose for the implementation of a configurable system. Some of these technologies have advantages in area and performance, but they require special processing techniques, such as laser structuring or fuse-blowing, which permanently change the physical structure of a hardware system and the cost can be high. Configurable systems using conventional low cost circuit switch devices can offer much higher user flexibility and reusability with slight degradation in performance and increase in silicon area.

The choice of a configuration method is determined by the design purpose and the target implementation technology for the design. Our system design

will be based on a normal CMOS process technology and it requires higher user configurability and reusability, so MOSFETs will be used as the basic switch devices in our configurable array architecture for algorithm embeddings.

1.3.3 Array System Timing and Control

There is a classification of arrays based on the type of instruction and data flow. An array whose elements operate on one instruction at a time to process one data stream is called a Single Instruction Single Data flow (SISD) array. Similarly, an SIMD array has Single Instruction and Multiple Data flow. If elements in an array operate on different instructions to process multiple data streams, the array is called a Multiple Instruction and Multiple Data flow (MIMD) array. An MISD array has Multiple Instruction and Single Data flow. A configurable array architecture for hardware algorithms can be regarded as a two phase MIMD array. A configuration phase is a Multiple Instruction flow (MI) phase. Once the array is configured, it runs in Multiple Data flow (MD) phase.

Because an array system can usually be divided into two types of essential parts: computation modules and a communication network, different timing control methods may be applied separately to computation modules and the communication network. A best match between these two timing control methods on computation modules and the communication network will make it possible to achieve an optimal performance and efficient system resource utilization.

It is a common practice to select either a synchronous or an asynchronous timing control method as the basis for an entire system timing control. In synchronous systems where all system operations are lock-stepped with a central global control clock, it is impossible to consider individual timing control for computation modules and the communication network. The speed of the central control clock in a synchronous system is determined by the worst possible case so as to secure the correct data movement in computation modules and the communication network.

The worst possible case is jointly determined by the worst clock distribution skew, the delay of the longest communication path and the slowest sequential logic in the computation modules.

There are no global system control clocks in asynchronous systems. The control of data movement and operation in an asynchronous system is completely localised. The performance of an asynchronous system is data dependent and measured by an average instead of the worst case for a maximum clock frequency as in a synchronous system.

We argue in this thesis that an asynchronous timing control can make the best out of a communication network, particularly a configurable communication network. It is also an excellent choice in general for timing control at the system level. We also argue that the synchronous timing control with clocks is still a very good choice for computation modules of sufficient complexity. Based on this argument, we look at the possibility of combining these two timing control methods together. A system timing design approach for a Globally Asynchronous communication network and Locally Synchronous computation modules (GALS) is established in the thesis. A configurable array architecture for algorithm embeddings is designed by applying our parallel multiple threads computation model and the GALS system timing control approach in the rest of the thesis. This architecture can easily accommodate any new technology, system design and user design changes. It also offers a solution to timing problems of the immediate future in the design of Ultra Large Scale and Wafer Scale Integrated (ULSI/WSI) systems.

1.4 Overview of the Thesis

Chapter 1: The essential concepts and ideas of hardware algorithms, configurable architectures and system timing control methodologies are introduced. The aim of this project is elaborated: the establishment of a proper computation model, configurable architecture and system timing control approach for high performance algorithm embeddings

Chapter 2: Some typical massively parallel computing systems are analysed and compared in this chapter. The purpose of these comparisons is to look at the common characteristics and the problems in these existing systems so that we can establish a proper computation model, interconnection network structure, and an overall system architecture for configurable hardware algorithms.

Chapter 3: In this chapter, a multiple threads computation model for irregular algorithms is established for algorithm embeddings. A configurable architecture template with a connected pool of hardware operators and a globally asynchronous locally synchronous (GALS) system timing control approach is proposed for algorithmically configurable architectures. A configurable GALS array system will be designed based on the ideas elaborated in this chapter.

Chapter 4: A Pseudo Nearest neighbour Configurable Array (PNCA) architecture with some constraints is proposed in this chapter. By combining the PNCA and GALS approach, a top level configurable GALS array topology and the logical structure of the interconnection network and a programmable hardware operator PH_{op} are illustrated. A guarded asynchronous handshaking communication protocol is also described.

Chapter 5: Various issues concerning the implementation of the GALS array system are discussed in this chapter. The design of a bit-serial configurable GALS array (GALSA) is presented. Some key components, such as an event-driven GALS data transfer interface, a novel tri-state register, the synchronisation issue, the design of a PH_{op} and a routing network, are described in detail. An example 4×4 GALS array is given. The event-driven GALS data transfer interface can also be used to construct general GALS systems.

Chapter 6: In this chapter, simulation results of the designs described in chapter 5 are presented. Three algorithm to the GALSA system mapping examples are given. The performance of the GALSA system is analysed and compared with some other similar systems. Although there are some extra delays caused by configuration switches in the routing network, the performance of the GALSA system is still very good because the system can process tasks based on the multiple threads computation model.

Chapter 7: This chapter summarises all the work presented in this thesis and the author's contributions. Further development of the current work is outlined. The prospects for configurable hardware algorithms in the future are also discussed.

Chapter 2

Massively Parallel Computing Systems

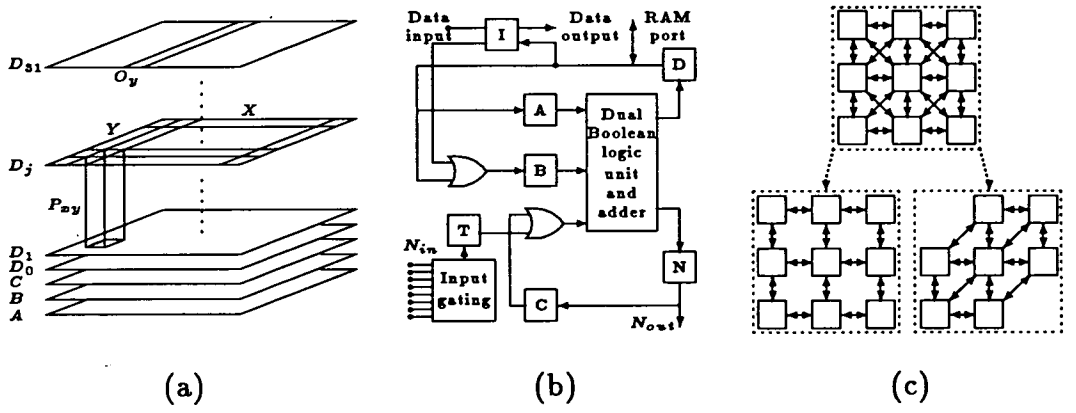
The work described in this thesis is closely related to a class of architectures called massively parallel computing arrays. A massively parallel processing array normally consists of one or a few types of processing elements (PE) which are duplicated as many times in a two dimensional plane and connected by a network as an application task requires. Some existing systems already have an array of 16K or 64K PEs. We shall analyse and compare some typical existing systems and other related work in this chapter. While some common points in these system designs are found, distinctions are also drawn between the research work carried out in this project and the existing systems.

Since 1970, the dramatic reduction in costs of integrated circuits and increasing requirements for high performance computing systems have stimulated the research and development of many parallel array architectures. Many of them have been implemented on Large Scale Integrated (LSI) circuit technologies. Newer generations of these systems and many other new systems are now mostly implemented in VLSI technologies. This makes it possible to accommodate more PEs in a silicon chip.

2.1 Cellular Logic Image Processor

The Cellular Logic Image Processor (CLIP) [24] was the first bit-serial array processor chip designed and fabricated. The development of the CLIP architecture can be traced back to 1973. It started at the time with many technology constraints which no longer apply. Although the early CLIP chip implementation is in current terms inefficient, the CLIP architecture is very heuristic. People are also trying to update the CLIP design with the latest state-of-art technology. The history of CLIP evolved from the the first prototype to CLIP4 (with 8 PEs on one chip in 1978) [24,35], CLIP5 (with 16 PEs on one chip in 1981) [34] and CLIP6 (1983) [33]. The CLIP5 PE uses essentially the same logic as the CLIP4 PE, while the chip configuration has been improved in several aspects, such as the use of a larger package which enables more pins to be assigned to control functions and more PEs to be integrated on one chip; local data storage has been removed to off-chip RAM and the data path design has been improved. There were substantial changes in the CLIP6 design principles. The major differences in CLIP6 are that all data paths and functional blocks are bit-parallel in operation on 8-bit of data. A single multi-bit ALU instead of a dual Boolean/adder PE is used, a multi-bit multiplexer for input selection replaces input gating circuitry, and a local condition code register supports a degree of PE autonomy control.

The data structure in an $n \times n$ CLIP array can be visualised as a stack of bit-planes as shown in figure 2–1(a). Each bit-plane is composed of an array of $n \times n$ data. One data bit in such a structure is represented as $D_j = \{d_{j,x,y} : x, y = 1, 2, \cdots, n\}$, where $d_{j,x,y}$ is a bit located at (x, y) in the jth plane. A data word can be stored in either binary stack format (vertical format) or binary column format (horizontal format). In binary stack format, a datum is uniquely addressed by (x, y) and passing through g bit-planes: $P_{x,y} = \{d_{j,x,y} : j = k, k+1, \cdots, k+g-1\}$. The binary stack format is very suitable for storage and processing



(a) (b) (c)

Figure 2–1: CLIP5: (a) Bit-planes, (b) PE, (c) Array interconnections

of pixels in an image. When numerical calculations are to be performed, it is sometimes convenient to represent data in binary column format: $O_y = \{d_{j,x,y} : y = 1, 2, \cdots, n\}$ in which there will be no one-to-one correspondence between data and n^2 PE addresses. The data structure in most of the two dimensional regular massively processing arrays can be modelled by this stacked bit-plane structure.

An old CLIP PE consists of a dual Boolean and a full adder processing unit, input gating circuitry, three registers (A, B, C) corresponding to a bit location in A, B, C bit-planes, a memory D for $D_{j,x,y}$ locating at (x, y) in D_j $(j = 1, 2, \cdots, n)$ bit-plane, and some glue logic. The PE can perform all the 16 Boolean functions with two variables and bit-serial addition which are all controlled by signals fed from an external control unit. The input gating however can be individually set to meet special application communication requirements. The CLIP PE schematic is shown in figure 2–1(b). Each CLIP PE is physically connected to its eight neighbours, but the actual CLIP logical connectivity among PEs can be configured as hexagonal, 4 nearest neighbour, or 8 neighbour connection to reflect the requirements of a specific application data structure (c.f. figure 2–1(c)). The input gating and full neighbourhood connectivity also make it possible to complete many operations in only one cycle for which other later designs may require as many as twelve cycles.

The CLIP system was specially designed for high speed image processing applications. A system with total of 9216 (96 × 96) PEs has been built with CLIP array chips. Data captured and A/D converted from a video camera are processed through the CLIP array and results are again D/A converted and output to a monitor. Some typical applications of such a system are simple edge detection, labeling, two-dimensional filtering operations on images, image enhancement, and skeletalisation.

2.2 Distributed Array Processor

The prototype Distributed Array Processor (DAP) [51] was designed and constructed in 1976 by Reddaway and others at International Computers Limited (ICL) as an enhanced memory module for ICL2900 series mainframes, and eventually evolved to an independent parallel processing system. The development of the DAP system was later separated from ICL to an independent company called Active Memory Technology Ltd (AMT). The first of the second generation DAP systems, built on LSI technology, was delivered in 1985. A prototype mini-DAP was built from a gate array chip which integrated 16 PEs together on one chip, and the whole system consists of an array of 32 × 32 PEs. A 64-PE custom VLSI chip is used in a version of the re-engineered 64 × 64 AMT DAP [55]. A DAP PE, as depicted in figure 2–2(a), is designed with a bit-serial full adder supported by a set of registers (A, C, Q, S, D), signal multiplexers, and an external local memory port which can address up to a maximum of 1M bits of RAM in the present architecture. The A register is used for PE "activity control" which can inhibit memory write operations in certain instructions in a PE. The activity control is also important in cases such as inhibiting PE operations at the predefined boundaries of a problem, or in conditional data-dependent operations. Q can be regarded as an accumulator and C as a carry register. The third input to the adder is selected by a multiplexer from PE memory, Q, A, data broadcast by the

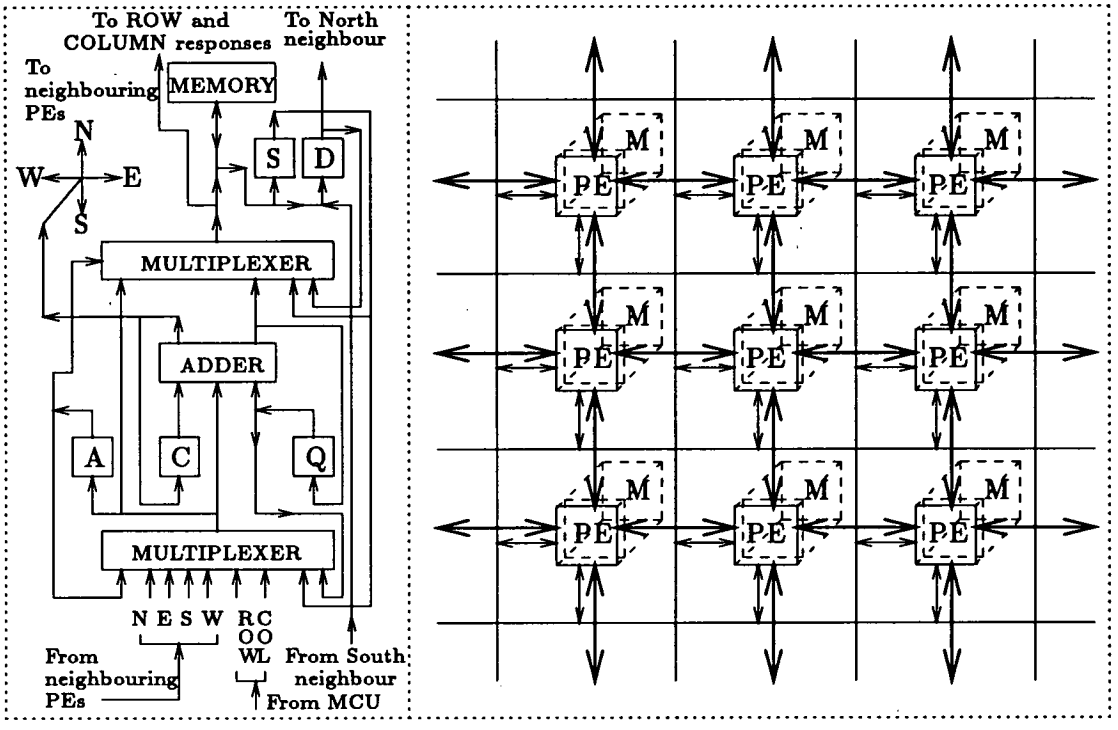


Figure 2–2: AMT DAP: (a) PE, (b) Array organisation

MCU (Master Control Unit), or the carry output of a neighbouring PE. S and D register are transparent to users, but play a very important role in assisting data movement. D is used as a buffer for data input/output through a fast interface unit, and S is a buffer for such instructions that need to read from and write to a memory. When A, C, Q, D and memory are viewed as an abstract data structure, they can also be represented as bit-planes as illustrated by figure 2–1(a) but with at least 32K array memory planes. Hence data in a DAP machine can be stored in either vertical or horizontal format.

Interconnections among PEs are essentially nearest neighbour connection but enhanced with X- and Y- buses for fast data broadcasting to the PE array or fast data retrieving from the array. Data in a register of a PE can move in any

of the four directions (North, South, East, West) to a corresponding register in a neighbouring PE. It is also possible to extract data from a specified row or column of PEs, or to AND together data from all of the rows or all of the columns. PEs at the edge of a DAP array are simply connected to their counterparts at the opposite edges, thus allowing shifts to "wrap-around" if required.

The DAP system has been efficiently applied to areas where large volumes of regularly structured data have to be processed, for instance, matrix manipulations, image processing, and sorting.

2.3 Massively Parallel Processor

The Massively Parallel Processor (MPP) project was initiated in 1971 and the construction of a real MPP system started from 1979 under a contract awarded to Goodyear Aerospace by NASA [7]. The first delivered MPP system consists of a physical array of 16896 PEs which can be logically configured as an array of 16384 (128×128) PEs with 512 (128×4) redundant PEs for the fault-tolerant purpose.

The interconnection topology in an MPP PE array is a simple conventional 4 nearest neighbour connection. The choice of this simple interconnection pattern is determined by the target application area of the MPP — two-dimensional image data processing, and by the huge number of PEs required. However, the edge topology of the MPP array is made very flexible as shown in figure 2–3(b). Programmable switch circuits are located on the four array edges. The connectivity between the right and left edge can be set as one of the following four states: open (no connection); cylindrical, $PE_{i,1}$ is connected to $PE_{i,127}$, $i = 0, 1, 2, \cdots, 127$; open spiral ($PE_{i,1}$ is connected to $PE_{i-1,n}$ for $1 \leq i \leq 127$); and closed spiral (similar to open spiral but $PE_{0,1}$ is connected to $PE_{127,127}$). The top and bottom edges of the array can be either connected or left open. When both left and right, top and bottom edges are connected, a ring configuration is formed. Since an

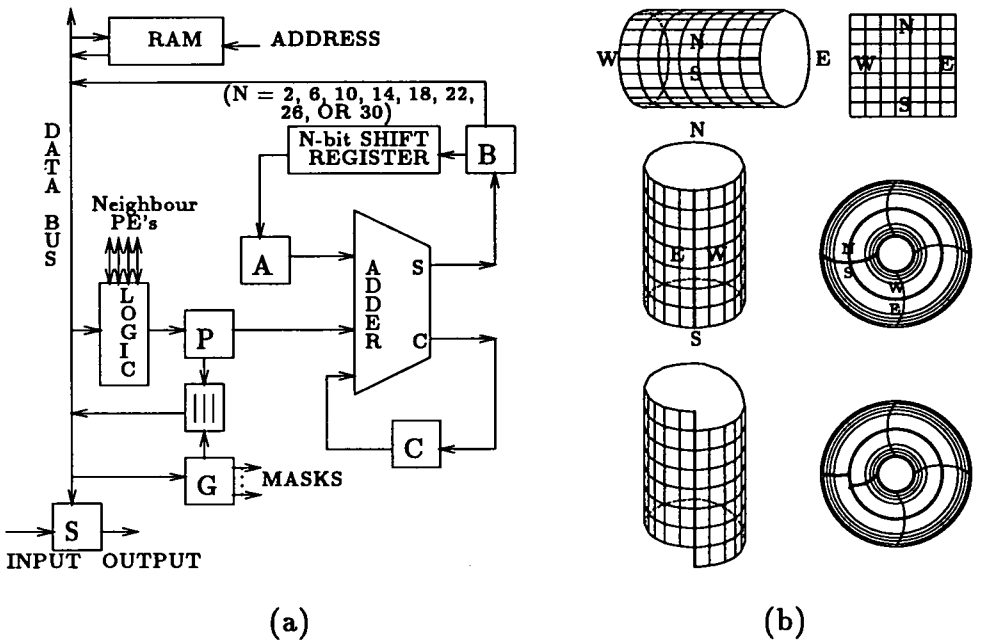


Figure 2–3: MPP: (a) PE, (b) Array edge topologies

MPP array is composed of a large number of PEs, fault-tolerance and reliability become important issues in the MPP system. Two measures are adopted. One is the simple group redundancy scheme by which a group of 128 × 4 redundant PEs is added. The entire array is divided into 33 groups of 4 × 128 PEs together with group bypassing gates in the routing network. The array can survive PE faults in one group by disabling (bypassing) the entire faulty group and activating the redundant group. The other technique is the parity error detection to find memory faults. One parity bit is combined with eight data bits of every 2 × 4 PE subarray. Whenever a fault is discovered, the group redundancy control will be used to disable the group containing the error.

Because the target workload of an MPP array is image processing in which the resolution of input pixels may vary from 6 to 12 bits, and intermediate results can be of length from 6 to more than 30 bits, the MPP PE had to be custom designed based on the bit-serial processing principle. This is very efficient to process operands of varying length. The actual PE structure as shown in figure 2–

3(a) was optimized for bit-serial arithmetic operations by combining a single bit full adder, a variable length shift register, and six single bit registers (A, B, C, P, G, S). Different from many other bit-serial PE design, a local data bus (D) is also used to provide a convenient way for data movement among PE registers and local RAM. All types of arithmetic operations are supported in the MPP PE array, such as integer/floating-point addition/subtraction, multiplication/division. The MPP PE logic circuit can also perform all 16 Boolean functions with two input variables from P and D register. A special routing operation which can shift the state in P to one of its four neighbours is included. The G register holds a mask bit so that masked operations are only performed in those PEs whose G is set to 1. The S register is used for shifting input and output data to/from the MPP PE array. A local RAM of 1K bits, from where operands are fetched and results are stored in a one-bit operation, is attached to each PE.

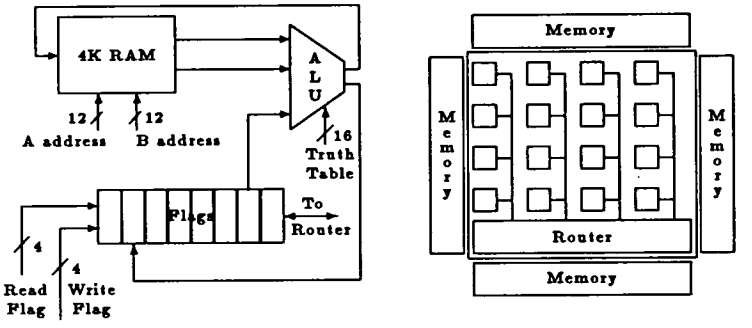
The implementation of the MPP PE array used a rather outdated technology. A subarray of 8 (2×4) PEs is integrated on one chip [110] designed with CMOS/SOS technology and packed in a 52-pin flatpack.

In 1986, a new project called BLITZEN motivated at miniaturising the physical size of the MPP system was started at the Microelectronics Center of North Carolina [10,47,21]. Years later, a custom VLSI CMOS chip containing 128 (8×16) PEs on an $11.0mm \times 11.7mm$ die was fabricated and mounted on a 176 pin PGA. The BLITZEN PE design is basically the same as the MPP PE but enhanced with more control functions such as local control of masking, local condition testing which may lead to alternative local processing actions and local modification of global addresses. But the interconnection pattern among PEs is very different from the MPP in that an "X" interconnection and rows of I/O buses are used. The "X" configuration enables each PE to communicate directly with its eight nearest neighbours and allows data to be routed along diagonals in an array which is faster than Manhattan routes.

2.4 Connection Machine

The Connection Machine (CM) is the largest massively parallel array processing system built so far. The architecture was originally conceived by Hillis in his thesis [49] at MIT, and was constructed by Thinking Machines Corporation. The major part of the CM has an array of 64K (2^{16}) data processors (PE) and a complex data communication network among these PEs. The communication network in the CM is completely different from other systems, using a packet-switched network instead of circuit-switched network. One of the goals of the CM design is to construct a very flexible architecture so that many different application types can be processed by the system. Operations based on data-parallelism make it possible for the CM to exhibit very high performance on processing massive amounts of data concurrently in each PE. The CM is well developed at the user interface level with a virtual-machine model which presents users with an abstract machine architecture to ease programming tasks for users on the machine. A virtual-processor model makes it possible to solve problems of sizes larger than the physical size of a CM.

The unique flexible communication network of the CM is supported by routers and a NEWS (North, East, West, South) grid. Every subset of 16 PEs shares one router while a total of 4,096 (2^{12}) routers is hard-wired in the pattern of a Boolean n-cube ($n = 12$). Therefore any router can be reached from any other router by travelling over no more than 12 wires. Each router handles messages for its subordinate 16 PEs and also serves as the interface with other routers. The operations of the router can be divided into five types: injection, delivery, forwarding, buffering, and referral. Injection is the process of sending new messages into the network from a subset of 16 PEs. The process by which a router removes a message from the network and sends it to a destination PE is called delivery. If an injected message is going to somewhere outside the cluster of 16 PEs, it must be forwarded. When several messages are delivered at once or several messages



(a) (b)

Figure 2–4: Connection Machine (a) PE, (b) A subarray of 16 PEs

are to be forwarded over the same wire, they will be buffered by the router. If a router's buffer is full, the router may refer a message to another router. Each router is uniquely assigned with a 12-bit binary address for the convenience of routing algorithms. The NEWS grid provides a two-dimensional Cartesian direct connection for nearest-neighbour communication which is faster than router communication for simple and regular data structures. Other communication modes such as broadcast communication and global OR of ALU carry output from all PEs are also supported in the CM.

A CM PE, which is shown in figure 2–4(a), consists of a bit-serial ALU unit, a local bit-addressable RAM, eight one-bit flag registers, a router interface, and a NEWS grid interface. The ALU is a logic element which can compute any two Boolean functions with three inputs and two outputs. The actual function of the ALU is determined by 8 bits stored in a function table. The A and B address specify the external memory locations from where the first and second operands to the ALU are read. The A-address is also the memory location to where the memory output of the ALU is written. The read flag selects one of the 16 (8 general purpose, 8 special purpose) flags from which the F input of the ALU is taken. The write flag selects one of the 16 flags to which the flag output of the

ALU is written. Flags in each PE are used to set communication modes and for memory error detection/correction.

The implementation of the CM architecture has evolved from CM-1 to CM-2 [130]. Both CM-1 and CM-2 use off-the-shelf RAM chips and a custom designed CMOS VLSI chip which contains a subarray of 16 PEs with one router and a control unit (c.f figure 2–4(b)). CM-2 has been improved in many aspects while keeping the same essential architecture as CM-1. The major differences in CM-2 are: 64K bits instead of 4K bits memory for each PE, four flag registers instead of eight, an optional floating point accelerator for every group of 32 PEs, increased error detection circuitry, redesigned router with improved reliability, diagnostic capability and performance, replacement of the two-dimensional NEWS grid with a more general n-dimensional grid on top of Hypercube, and a high speed I/O system.

The CM is a very flexible architecture which can be applied both in numeric and symbolic processing to a very broad range of applications. For instance, grid-based communication finds primary application in regularly structured problems such as particle simulations and matrix manipulations, while the general packet routing supports varying topologies in circuit simulation and computer vision.

2.5 Adaptive Array Processor

The Adaptive Array Processor (AAP) [64] was developed at NTT in Japan. The AAP architecture was designed to be used as a high performance system with a certain degree of flexibility for various two-dimensional data processing applications and small overhead for inter-PE communications over long distances. The adaptability of the AAP to applications is supported by three special features:

- Duplicated communication paths. Each PE is connected with its eight neighbours, and a duplication of connection between upper and lower PEs makes it possible to implement hierarchical bypass for flexible and fast data transfer.

- A complex data transfer unit in each PE. Since the physical interconnections among AAP PEs are fixed, a complex data transfer unit is used in each PE to support flexible data routing and hierarchical data bypass. The data transfer unit is implemented with various multiplexers and some control registers. Combined with duplicated interconnection paths, data can be routed rather freely among PEs in the AAP.

- Local modification of global controls. The local adaptability of each PE is determined by the contents of local control registers which can modify common control signals fed from a single global control unit outside the PE array.

The AAP PE comprises a 16-function bit-serial ALU for two inputs and a set of supporting registers. One unusual feature is the use of one of the data registers to hold a control signal which determines the storage destination for a result. A 64-bit register file is used to hold temporary data and makes it possible to implement various operation modes on an AAP. Data can also be structured in either vertical or horizontal format in an AAP as shown in the stacked bit-plane of figure 2–1(a). However, the AAP extends the stacked bit-plane structure with more operation modes. Bit-serial operations can be carried out in each PE on entire words stored in vertical format. The PE array can also be structured to either word-unit or block-unit operation modes. When part or the entirety of a row or column of PEs are combined to process data in horizontal format, the AAP is in word-unit mode. If a block of PEs (a subarray) is grouped together, horizontal words can also be processed in block-unit mode. The AAP PE schematic is shown in figure 2–5. The first prototype AAP chip integrates 64 (8×8) PEs together with 6K bits ($64 \times (64 + 32)$) of memory on a die of nearly $1cm^2$. The second generation of the AAP — AAP2 [65] is designed with an external RAM port of up to 1M bits logical address space in each PE. The AAP2 has one 40-bit microinstruction modifier which can support wideband modifiable PE operations in the array (called pseudo MIMD by AAP designers). The interconnection paths are also enhanced

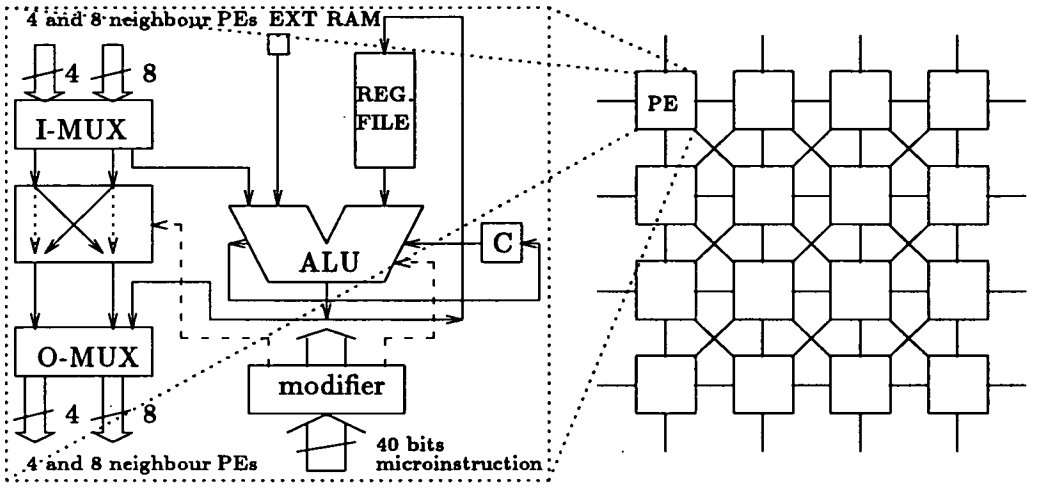


Figure 2–5: AAP: (a) PE, (b) Array interconnection paths

by a duplicated 4 neighbour connection network (c.f. figure 2–5(b)) instead of the upper-lower path in the AAP prototype. One external 64K bytes (8 × 64K bits, static) RAM module and one AAP2 64 PE array chip can be mounted on one special package.

The AAP system can be applied to many two-dimensional data processing applications. Examples are grey tone level histogram calculation, distorted image correction, feature extraction of character images, and logic simulation of electronic circuits.

2.6 A Data-Driven VLSI Array

I. Koren and B. Mendelson [66] developed a Data Driven VLSI Array (DDVA) for embedding arbitrary algorithms. The DDVA is different from most of the massively parallel processing array architectures in that it operates on a data-driven principle. With the elimination of global control on data-flow, correct operation of the system will be guaranteed by the availability and presence of matching input data to each PE in a DDVA. The DDVA is designed with a hexagonal interconnect architecture where each PE is connected to six nearest neighbours. There are

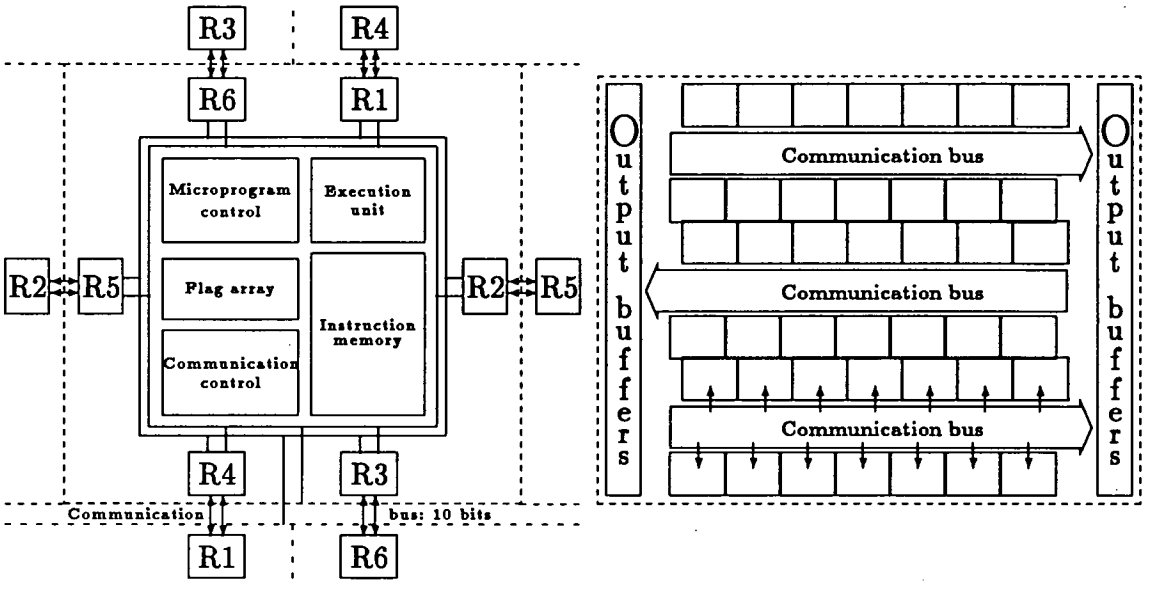


Figure 2–6: DDVA: (a) PE, (b) array architecture

rows of communication I/O buses of 10-bit wide (figure 2–6(b)). Communications among PEs are all through some of the six registers in each PE as shown in figure 2–6(a). A data flow graph for an algorithm can be mapped into a DDVA where nodes (vertices) of the graph are embedded into the PEs of the DDVA and edges (arcs) of the graph are mapped either as a connection between two communicating registers or a series of PEs which are only set as communication path if connection resources are not enough. Each DDVA PE processes 8-bit operands and is data-driven, i.e. a PE instruction is initiated only when all of its required operands are available and its destination registers are empty. Each PE can be loaded with at most six instructions, the execution order of instructions is completely data-driven instead of depending on a program counter. The complexity of the prototype PE is about 9,000 transistors in a NMOS technology. The microprogram control unit is the largest block in the PE which requires about 4,500 transistors. This unit translates instructions stored in the PE instruction memory into sequences of control signals that control the operation of the execution unit. This microprogram control unit is the fundamental feature of a DDVA that makes it possible for the DDVA to embed arbitrary algorithms and operate on the data-driven principle.

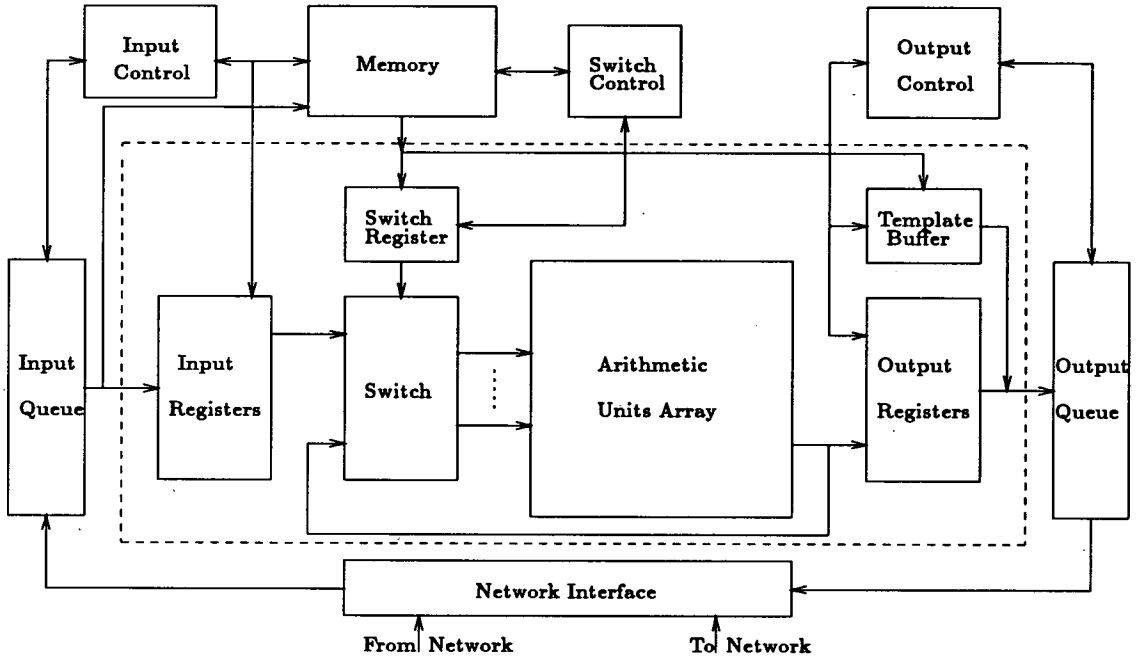


Figure 2–7: RAP system architecture

The DDVA architecture has very low hardware efficiency for arbitrary algorithms because it uses PEs as routing cells with very limited routing resources. It is too expensive to use such a complicated PE for just a simple routing function.

2.7 Reconfigurable Arithmetic Processor

The Reconfigurable Arithmetic Processor (RAP) [32] is actually developed as an arithmetic processing node for an MIMD concurrent computer. The aim of the RAP design is to reduce the amount of off-chip and memory data transfer traffic by evaluating an entire arithmetic formula directly in an RAP upon the configuration of the RAP to the structure of the formula to be evaluated. Only final evaluation results are sent back to a host computer after a set of inputs is presented to the configured RAP. All of the intermediate data are calculated, referred to and eliminated locally; no global memory references which will otherwise require expensive high speed I/O channels that are often a performance bottle-neck. All of the external communication requirements of an RAP are the input of configuration control data (which will be done prior to calculations), input of coefficients and values of

variables, and the output of final results. The schematic of the RAP architecture is illustrated in figure 2–7. The major building blocks in an RAP are a set of arithmetic processing units which include adders/subtractors and multipliers, a switching network, and a switch configuration control unit. Partial bit-parallel arithmetic units are designed to process 4-bit of operands in one operation or a serial-of 4-bit if longer operands are presented. The switching network in the RAP can be configured with the switch configuration control unit to interconnect arithmetic units in a way which represents the data-dependencies of the formula.

One of the most important features of the RAP architecture is the highly reduced memory communication flow requirements. Once all of the required data is retrieved from the memory, the entire evaluation of an arithmetic formula will be carried out without interaction with the host system until final results are obtained.

2.8 Reconfigurable Parallel Array Processor

The Reconfigurable Parallel Array Processor (RPAP) was developed by Rushton and Jesshope [117] [52, Section 3.5.4] at Southampton University with the aim of implementing a more general-purpose flexible architecture assuming Wafer Scale Integration techniques. The RPAP architecture is very flexible in that it can be logically configured to perform from bit-serial, through partial bit-parallel, to word operations. To minimise the communication problem between a host and an RPAP array, a shared memory between the two is used so that the address of the array memory can be directly mapped into the address space on the host. The RPAP is not a real SIMD system because some fields of the instruction words distributed across an RPAP array can be locally modified in a PE.

An RPAP is physically a two dimensional four nearest neighbour interconnected (NNI) array of simple bit-serial PEs with wrap-around at both opposite edges. The whole system is synchronously operated. The RPAP chip VLSI floor-

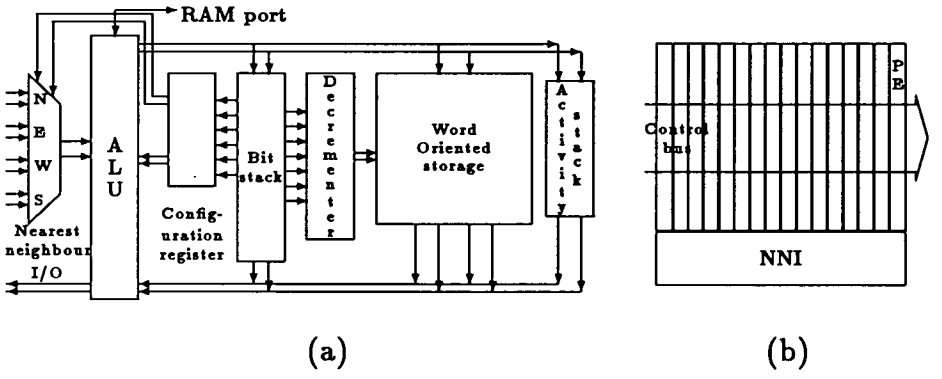


Figure 2–8: RPA: (a) PE, (b) The array floor plan

plan is also arranged as a square array of PEs. Because of synchronous timing control difficulties, the RPAP architecture can support up to 256×256 PEs in an array. A physical PE is a single bit-slice ALU. A wider virtual processing unit can be formed by combining a group of physical PEs. A virtual $R_v \times C_v$ array of $r \times c$-bit virtual PEs can be logically constructed out of an RPAP of $n \times n$ physical PEs, where R_v, C_v, r, c must satisfy:

$$R_v \times r = n \tag{2.1}$$

$$C_v \times c = n \tag{2.2}$$

$$r \times c \leq n \times n \tag{2.3}$$

$$1 \leq r,\ c \leq n \tag{2.4}$$

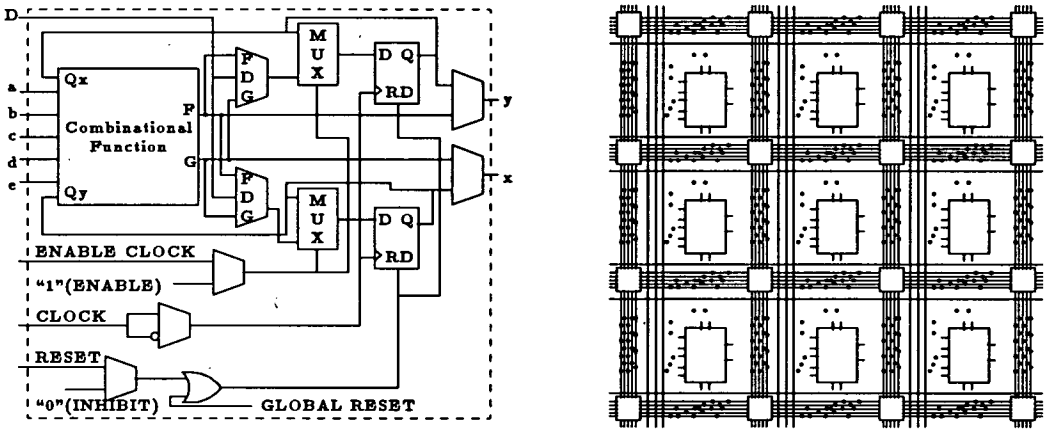
For example, from a 32×32 RPAP, a virtual single processor of 1024-bit is formed when $r = c = 32, R_v = C_v = 1$. Similarly, $r = 16, c = 4, R_v = 2, C_v = 8$ forms a 2×8 array of 64-bit virtual PEs. There are no restrictions on the direction of data propagation in the system, but each connected successive bit-slice PE should be physically adjacent to form a virtual PE. The structure of a physical PE, as shown in figure 2–8, is similar to one bit-slice in a conventional bit-slice processor. The special feature of the RPAP PE is that it has two-bit lines which allows simultaneous transfer of two operands to each PE. Both bit-serial and bit-parallel operations are well supported in an RPAP PE. A bit-stack and an activity stack can preserve the bit ordering required for arithmetic operations, besides which

they are able to provide parallel-to-serial and serial-to-parallel conversions. The internal storage has a dual control mechanism capable of stack or random access to the bytes of a word.

2.9 Field Programmable Gate Arrays

Gate array and standard cell architectures are developed for implementing VLSI system designs with fast turn-around time. Compared with custom design methodology, gate array and standard cell approaches have satisfied this fast turn-around time to some extent. It is also much easier to automate system design based on these predefined chip architectures. However, this improvement on turn-around time is still fundamentally limited by the procedure of chip fabrication. On the other hand, designers have no control over their designs once their designs start to be fabricated, not to mention extra costs to modify a design after chips are fabricated.

Field Programmable Gate Arrays (FPGA) pioneered by Xilinx Inc.[139] represent a different design methodology which resulted in a very versatile architecture that gives truly fast turn-around time and complete user freedom to implement a system design, test and modify the design easily. Following the success of Xilinx, there are now many companies also producing FPGA chips. The Xilinx XC series are also called Logic Cell Arrays (LCA), which in general are composed of a two dimensional array of logic cells surrounded by an interconnect area which is rich in memory controlled switches and wires. All of the logic cells in an XC array, as depicted by figure 2–9(a), are the same and can be configured to any logic function of up to 5 input logic variables (a, b, c, d, e). Functions with more than 5 inputs can be formed by combining cells together. D is used for a direct data input. Other inputs to a cell are used for cell operation controls. A special feature of this LCA cell is that two combinatorial functions of 4 inputs which share a common input can be configured within one cell. Thus there are two outputs x and y. Interconnections among logic cells can be set by configuring various switches around



(a) (b)

Figure 2–9: Xilinx LCA: (a) Function cell, (b) Interconnect resources

each logic cell to match the structure of a target system. The LCA provides plenty of programmable interconnect resources which are classified as general purpose interconnect, direct connection, and long lines, as shown in figure 2–9(b). General purpose interconnects consist of a grid of 5 horizontal and vertical metal tracks segmented by switch matrices and scattered programmable interconnect points (PIP, small spots in figure 2–9(b)). Direct interconnects provide the most efficient network interconnection between adjacent logic cells or I/O blocks. Long lines bypass switch matrices and are intended primarily for signals which must travel a long distance and must have minimum skew among multiple destinations. FPGAs have been successfully used in many designs. Because FPGAs are intended for use as general purpose devices, most of these designs are based on random logic which ignores the architecture regularity inside FPGAs.

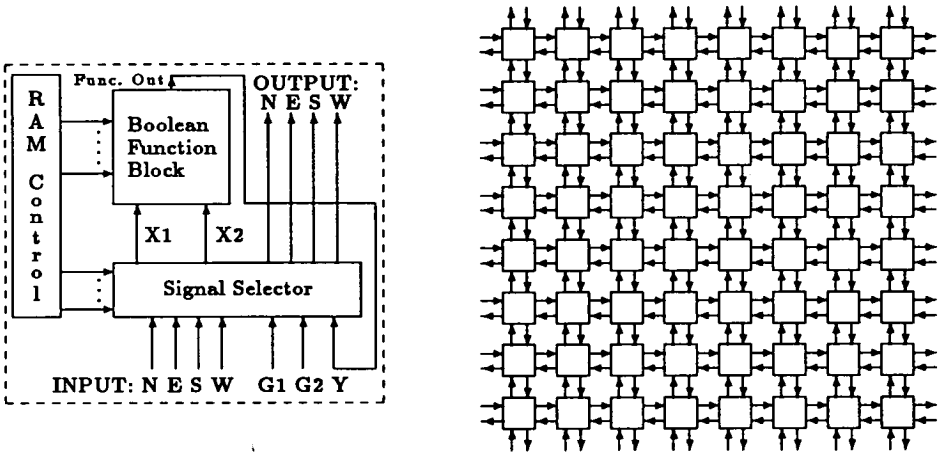
2.10 Cellular Array Logic

An early systematic description of the cellular logic design methodology can be found in [103] in 1971. Because of technology limitations, no real cellular arrays had been integrated on silicon chips. In 1977, Manning [87] further extended the concept to programmable cellular logic arrays for arbitrary logic implementations.

However, at that time it was still unrealistic in practice to implement such kinds of system with medium scale integration technologies. Xilinx is the first company to design and market user soft programmable gate arrays [139]. The success of Xilinx and the emergence of sea-of-gates architectures aroused researchers' interests in cellular logic. Kean [59,58] has designed his configurable Cellular Array Logic (CAL) architecture based on previous work on cellular logic architectures. The CAL became a standard product of formerly Algotronix Ltd which is part of Xilinx now.

The CAL architecture is very simple. The entire system consists of a 2-D array of a simple function cell which is bidirectionally interconnected with its four neighbours, as depicted in figure 2–10(b). Each cell in the array can be configured to either: one of the 16 two-variable one-bit logic functions, or as a routing cell which can route input values to another cell. The CAL cell function can be represented as Y = F(X1, X2), where X1, X2 and Y are Boolean variables. Compared with sea-of-gates architectures, CAL can actually be classified as a sea-of-cells architecture. There are 20 bits of RAM in each cell as shown in figure 2–10(a) for cell functionality, signal selection, and routing control. The configuration control bits for all cells in an array are loaded *a priori*. The current CAL chip contains 64×64 cells on one VLSI chip.

Since CAL cells manipulate only a single bit of data and do not have local memories for intermediate data, CAL is most suitable for bit-level applications, for instance, encryption/decryption, or binary image manipulation. It is not efficient to use CAL for applications with wide word width. Because CAL cells can also be used for routing purposes, the cell utilization of a CAL will be low if the data structure of a target system is of low regularity or communication intensive. This situation is more likely to happen for configurable systems which are supposed to be able to solve various and perhaps irregular algorithms. Another problem that the CAL may suffer is synchronous timing control difficulties. The CAL architecture is designed as a synchronous system; worst case timing requirements



(a) (b)

Figure 2–10: CAL: (a) Cell, (b) Cellular array

are different from one configuration to another. As a result, system control clocks have to be determined after each configuration, or simply imposed with the worst case clocks. This timing problem will be even severe when more chips have to be connected to form a larger CAL system.

2.11 Comparisons and Remarks

We are by no means trying to list all parallel processing systems. Systems described in previous sections are very representative for massively parallel processing systems in several areas: PE structure and memory requirement, interconnection network structure and data routing mechanism, abstract system data structure, system timing control strategy, and application objectives. There is no systematic performance comparison of these systems because they are designed for different purposes with substantial difference in costs. Performance, cost and generality of systems are often the factors that have to be compromised one for another by system designers and end users. We try to make good judgements from these existing systems and develop a new framework for configurable hardware algorithms.

All massively parallel computing systems can be classified as either programmable or configurable hardware algorithms. This is because the architecture of these systems explicitly or implicitly reflects the data structure and dependency of its target applications, there is no time sharing for users on these systems (running only one application algorithm at any time). The system operation schedule control is built into hardware. Data dependencies and functions of PEs can be dynamically changed in programmable systems such as DAP, MPP, CM, and DDVA. CLIP, AAP, RAP, RPAP, LCA and CAL are configurable systems which are configured *a priori* for a particular application in the configuration phase. Data can only be processed by the configured system in the run phase. A reconfiguration has to be carried out for a different algorithm.

There are two ways to run these systems. In stand-alone mode, they can be the centre for digital data processing. Inputs to such a system may be captured by cameras or sensors and A/D converted. Outputs may be directly used or D/A converted to control other devices, for example a robot to operate. They can also be used as attached systems to a host computer. The host computer acts as a front-end as the user interface and schedules system tasks by setting up the attached systems, down loading control instructions and data into them, and collecting results from them. CLIP, AAP, RAP, RPAP, LCA, and CAL can be used as either stand-alone or attached systems. But programmable DAP, MPP, CM, and DDVA are more suitable to run as attached systems. This is because these systems can be dynamically programmed at run-time and can be managed more efficiently by a host computer.

The Connection Machine is the most general purpose and expensive massively parallel processing system among those we described. Its strength lies in the combination of a packet switched router network and a circuit switched NEWS grid network to connect 16K simple PEs together, while other systems only use the circuit switching technique in their interconnection networks. This system demonstrates the importance of communication network on performance and adaptability

of a large parallel system. It is a powerful system for problems with high dynamic run-time properties, such as analysing particle system state in particle physics, or solving astronomy problems.

The CLIP is a rather special purpose system for fast image processing applications. However, it can be seen that the stacked bit-plane data structure in the CLIP is adopted by many other systems such as DAP, MPP, and AAP by using a local memory in each PE. The reason is that these systems share common target applications with very regular two-dimensional data arrays. The processing of one element in such a data array is often associated only with its 4 or 8 neighbouring elements in the array. Images consisting of pixels and matrices are two typical examples that can be represented by data arrays. The stacked bit-plane structure is also a concise abstract representation for an array of bit-serial PEs common in these systems. There is at least a nearest neighbour interconnection with neighbour selection logic in each PE to match the nearest neighbour data dependency, for example in image processing applications. However, variations in network structure exist, for instance, the choice of 4, 6, or 8 neighbour connection in a CLIP array, the X and Y bus for fast data broadcasting to and retrieving from a DAP array, the duplicated communication paths between PEs and the data transfer unit in a PE to route data beyond nearest neighbours in an AAP to increase the AAP's adaptability to different applications. In general, three phases are required to operate these systems: loading data into local memories in each PE, data processing in vibrating mode where data may be moving forward and backward only one PE distance, retrieving results from the array. It is noticed that a large local PE memory is required to reduce the time overhead in the array data loading phase. This is why newer generations of these systems tend to be designed with an external RAM port in a PE in order to use high density off-the-shelf RAM chips.

The CLIP is the best for image processing applications. DAP and MPP are the best for matrix manipulation tasks while the MPP demonstrates its best support

for floating-point operations with its variable length shift register in an MPP PE. The AAP has the best adaptability to a wide range of applications.

The RPAP is designed as an attached system. It avoids the PE local memory loading overhead by sharing memory modules with its host computer so that both the RPAP and its host computer can address the same memory space. The new problem with this shared memory will be the routing overhead from shared memory modules to each PE in an RPAP array and the memory read/write control where writing to a memory address can only be performed after all the reading requirements to the same address are completed. The RPAP does not provide routing adaptability on its nearest neighbour interconnection network; it depends on its virtual PE and virtual array principle where a virtual PE capable of N-bit processing can be configured from N bit-serial physical PEs and a virtual array is formed by virtual PEs. The RPAP's application is still limited by its nearest neighbour interconnection network in its physical and virtual arrays.

A completely different system architecture approach is adopted in [123] by using a binary-tree interconnection structure because many algorithms can be transformed and fitted into a binary tree. However, this algorithm transformation is by no means a trivial work. On the other hand, it is often difficult to preserve existing parallel properties of an algorithm after this transformation.

The DDVA is one of the architectures that aims at implementing arbitrary hardware algorithms. The idea of applying the data-driven computation principle to massively parallel processing arrays in the DDVA is very important for array architectures and in particular for systems to embed arbitrary algorithms. This avoids the difficulties of system level synchronous timing control in other systems where the actual physical size of a synchronous array system is limited by the problem of distributing the system global clocks. For example the largest physical size of an RPAP array is limited at $256 \times 256 = 65K$. Algorithms represented by Data Flow Graphs (DFG) can be mapped into a DDVA array. However, the DDVA architecture is poorly devised by using expensive PEs for the routing.

The RAP represents another approach to implement arithmetic algorithms in hardware. An algorithm is firstly decomposed into such a level that all arithmetic operations are supported by the available basic arithmetic processing units in the RAP array. The RAP interconnection network is then configured to the corresponding data dependency of the algorithm at this level. The RAP is designed with synchronous system timing control. This also imposes difficulties on the timing schedule every time the configuration is changed and always limits the system performance to the worst case critical path in a configured RAP.

DDVA and RAP have some common properties. They do not need large local memory allocation to their PEs because data is processed on-the-fly instead of sitting in PEs. They can significantly reduce memory access traffic which only happens when they read input data and output final results. Although the bit-serial processing principle is most commonly adopted, it can be seen that partially bit-serial and partially bit-parallel (a PE processing 2^n bit in parallel) is a way of further increasing system performance. The RAP uses 4-bit partial parallel arithmetic units, a DDVA PE and the PE in [123] process 8-bit operands, and an RPAP virtual PE can process operands of any word length up to the total number of physical bit-serial PEs in an array.

FPGAs are not particularly developed for parallel processing tasks. The major target of FPGAs is fast implementation of designs which otherwise have slow design turn-around time if they are to be implemented in gate arrays or as ASICs. Xilinx's LCA and Algotronix's CAL are two typical representatives for FPGA architectures. The LCA is designed with very rich routing resources and relatively complex functional cells. The CAL consistently uses an array of simple cells which can be configured to perform either a logic function or data routing. There are no other routing resources in a CAL array. One of the criteria for FPGA systems is the equivalent gate count to a conventional gate array by ignoring configuration logic and the percentage of the gate utilisation because some of the blocks may not be routable when an algorithm is embedded. In general, both systems are

similar in these two aspects for a given algorithm. However, it is much simpler to map an algorithm into a CAL. But a CAL can be slower than an LCA because there are fast routing resources, such as long lines and direct connections, in the LCA while a long distance routing in a CAL must go through a series of cells. The CAL is designed with synchronous system timing control. The worst case routing delay can restrict the system running at very slow speed. FPGAs are widely accepted nowadays, as is the technique to set the system configuration by using static memories. Theoretically speaking, any complex systems can be constructed from FPGAs. In practice, a complex system requires many FPGAs where synchronous system timing difficulties will restrict the size of the system to be built. The CAL is a suitable candidate for bit-level parallel processing tasks.

Fault-tolerance is a very important issue in massively parallel processing array systems. This is only considered in the MPP design which uses a very simple group redundancy scheme and parity error detection for memory faults. The CM-2 is also designed with error detection and diagnostic logic to improve the system reliability. In configurable systems, such as FPGAs or DDVA and RAP where irregular applications are to be embedded, a fault-tolerance technique called graceful degradation [29] can be used. Graceful degradation does not try to recover a system's size to its originally intended as redundancy techniques do; it simply discards faulty PEs and make use of the rest of good PEs in the system. The system is rendered unusable by the graceful degradation technique if there is no way to route around a faulty PE. This may happen when there are too many faulty PEs in one fault cluster or the routing logic around the PE is faulty.

2.12 Impacts on Configurable Hardware Algorithms

We shall discuss impacts of these typical systems on the development and design of our configurable architecture for hardware algorithms. The problems in these existing systems that we try to solve in our approach are also illustrated.

2.12.1 Circuit Switching vs. Packet Switching

Apart from the CM, which uses the packet switching technique for its router network, all the rest employ various circuit switching networks. One of the advantages of packet switching is that logical data routings are localised and time dependent. Since every packet contains a complete message of the destination address and data to be routed, a packet can be easily relayed from source through physical paths segmented by packet switches, to the destination. A packet switch is capable of temporarily storing a packet, determining its immediate next relay stage, and transmitting the packet to it. Logically, each packet only occupies one segment of a physical communication path with two packet switches on its ends. The physical path can be released for the routing of another packet every time a previous packet is absorbed by a packet switch. Therefore, in a packet switching network, each packet has a logical routing path, and all of the packet logical routing paths are overlapped on one physical network either simultaneously or sequentially in time. Hence there is much less network congestion possibility in a packet switched network than in circuit switching networks limited by the availability of physical network routing resources. A packet switching physical network does not necessarily represent any application data dependencies.

In circuit switching networks, instead of holding routing information which flows with data in packets, data routing controls are all fixed into the physical network. Every physical path is uniquely assigned for one source to destination communication. Thus enough physical communication paths must be provided to meet application requirements. Once set, a circuit switching network will surely

represent the data dependency of a particular application. It is a common practice to sacrifice some PE resources to gain more routing resources if there are congestions in a physical network.

Packet switching networks are very flexible and have an excellent dynamic routing capability for the fairly wide range of applications with completely different data dependencies on top of a limited number of physical communication paths. Although the number of physical communication paths has been kept minimal, the complexity of the packet switches is much higher than circuit switches. Therefore, the packet switching technique is a favorite choice for general purpose supercomputing systems. On the other hand, the systems illustrated in previous sections are mostly special-purpose-oriented, it is important to design the architectures as simple as possible to keep the system costs under control. Therefore, circuit switching networks are overwhelmingly used in massively parallel systems.

The circuit switching technique will be adopted in the design of our configurable architecture for embedding algorithms because of its simplicity.

2.12.2 PE local memory

As we analysed in the last section, a relatively large addressable local memory is required in a PE in array architectures for applications with regular data structures which involve minimum data movements when they are processed. However, DDVA, RAP, LCA, and CAL are targeting at different types of algorithms that are irregular, computation intensive and have substantial data movement but not data intensive when they are processed. Hence these systems put more emphasis on the network routing capabilities and the PE functionalities. Because the target applications are not data intensive, only some local registers are required in a PE to hold a small amount of the intermediate data generated during operations.

We target our configurable architecture for irregular and computation intensive algorithms. Data will also be processed on-the-fly when they flow through the

system. Therefore, the PE in our design will only need a small number of registers instead of a large block of local memory.

2.12.3 PE Degree

A PE degree D_{PE} is defined as the number of neighbours with which a PE is directly connected. The D_{PE} in a blank architecture is called physical D_{PE}; the D_{PE} in a system embedded with an algorithm is called logical D_{PE}. The logical D_{PE} in a system may differ from its physical D_{PE}. The physical D_{PE} of each PE in a system can also be different, for example, the physical D_{PE} on edges and corners of an array is often different from that inside the array. D_{PE} is an important factor in circuit switching networks because it represents the number of channels which are available for a PE to communicate with others.

The most popular physical network topology is the nearest neighbour interconnection. However, there are still many variations of PE degree, for instance, one PE can be connected to 2, 3, 4, 6, 8 immediate neighbours (physical $D_{PE} = 2, 3, 4, 6, 8$), in two dimensional arrays. Horizontal rows and vertical columns of broadcasting buses, as used in DAP and DDVA, are often added for fast global data transfer or improvement of interconnection flexibility. Hierarchical interconnections can also offer a great deal of flexibility for an architecture. There are two forms of interconnect hierarchies: the first consists of a set of different interconnect networks such as local connections, short and long lines in the LCA, and Boolean n-cube and NEWS grid in the CM; the other consists of the duplication paths as used in the AAP. An effective D_{PE} can be found in the case of both buses and hierarchical interconnects. Generally speaking, the higher the physical D_{PE} is, the more expensive but also more flexible will an array architecture be. For systems targeting at image and matrix processing applications, physical $D_{PE} \leq 8$ will usually be sufficient. For configurable systems, the choice of a phys-

ical D_{PE} is a compromise among implementation cost, system configurability and the percentage of PE utilisation.

Although the physical D_{PE} of an architecture can be fixed, it is usually possible to get a logical D_{PE} either smaller or larger than the physical D_{PE}. Examples can be found in most of the architectures in previous sections. In figure 2–1(c), the CLIP's physical $D_{PE} = 8$ while its logical D_{PE} can be set as (2, 4, 6, 8). The physical $D_{PE} = 3$ on the edges of the MPP can be logically set to $D_{PE} = 2$ *or* 4 as depicted in figure 2–3(b). The logical D_{PE} of CM, AAP, DDVA, RAP, RPAP, LCA, and CAL can be very flexible in that non-uniform D_{PE} in an array may even be possible by merging a group of PEs or cells together. This capability is required when an irregular algorithm is to be embedded in these systems.

The routing structure used in the AAP has a very good adaptability at a reasonable cost and still keeps a very good system regularity. We shall use a similar approach to design a PE with a routing part and a function processing part. The idea of our approach is to connect an array of the routing parts to form a two dimensional configurable array network, and a processing part is attached to each routing part to form a processing array. The physical D_{PE} will be determined by the design of the routing part which will be discussed in detail later in this thesis.

2.12.4 PE Functionality

A PE can be viewed as a much simplified version of a complex microprocessor. The simplification is often done in several ways:

Elimination of the control path

The control path in a normal microprocessor can consume more than half of its design complexity [2]. The major role of this control path is instruction fetching, decoding, sequencing, and the control of other system level signals, such as interrupts and bus access. Since most massively parallel systems are based on the

SIMD principle, there is no necessity to give each individual PE such a strong autonomous-control capability. A single central control unit, which is normally external to an array, will be able to meet most of the array control requirements. Instructions broadcast from the central control unit to the array are usually at the lowest level which can be directly executed by the data path in PEs. A PE can have some degree of low level autonomy, such as disable/enable or local address modifications. This is often desirable to increase the adaptability of a system to slight variations in applications. For example, the disable/enable register in CLIP, DAP, MPP. However, an AAP PE is capable of more complicated local modifications to globally broadcast microinstructions, and a DDVA PE uses a complex microprogram control unit similar to a conventional control path, which converts instructions in a PE local memory into sequences of control signals. In configurable systems, such as RAP, LCA, CAL, instead of concurrent instruction flow and data flow, "instructions" are loaded into control memories in each PE *a priori* which statically set both network communication patterns and computations to be performed. There will be computation data flow only in the normal operation mode. A new configuration has to be loaded to run a new algorithm.

Because it is not intended to implement dynamically programmable hardware algorithms at this stage, the simple "configure then run" two phase approach used by RAP, LCA and CAL will also be used as the operation principle for our configurable architecture. The network and PE functionality are set in the configuration phase. There will be no global instruction broadcasting in the run phase.

Low level computation capability

The most commonly supported logic functions in a PE are the full set of all 16 Boolean functions with two input variables, or a full adder for arithmetic operations. A separate logic unit and a full adder can be found in the CLIP and MPP PE. Only a full adder is used in the DAP PE. AAP, CM, and DDVA use conventional ALU designs. A CAL cell can only perform 16 Boolean functions, while an

LCA cell is designed for Boolean functions of up to 5 input variables. However, the RAP, exceptionally, has a pool of arithmetic computation elements for additions, subtractions, multiplications, and divisions for calculating arithmetic formulas. Another special array system, described in [78], is designed without the ALU in a PE. Instead, a sophisticated table-look-up mechanism is used to implement all of the computation and routing functions.

The configurable hardware algorithms to be developed are mostly arithmetic oriented, but also require to perform some simple logic functions such as AND, OR, XOR occasionally which is different from the RAP. At the same time, the system regularity is to be kept. Therefore, a configurable hardware operator will be designed. It can be configured to perform some basic arithmetic and logic functions.

Bit-serial operations

Most of the existing massively parallel processing systems are designed based on bit-serial word-parallel (BSWP) principles. This choice is again largely determined by technology limitations rather than application requirements, since most data being processed are multi-bit (except binary images). One advantage of bit-serial processings is flexibility in choosing an appropriate word length and precision at much lower hardware cost than the bit-parallel processing principle. For example, it usually takes 3 clock cycles to process 1-bit in a PE (retrieve operands, process them and store results). In a conventional single CPU sequential computer, data are processed in the bit-parallel word-serial (BPWS) principle. It usually takes 5 clock cycles (instruction fetch, decode, access operands, operation code execution, store results) to complete one arithmetic instruction. Assuming that these five operations can be carried out in a pipeline, the processing time depends on a fixed pipeline latency and the number of data and instructions fed into the pipeline. For example, 1K cycles (pipeline latency is not considered) are required to process a set of 1K 32bit data in a BPWS computer. It takes only $3 \times 32 = 96$ cycles to complete on an 32×32 BSWP PE array (plus some extra cycles for data access).

The performance of a bit-serial PE array mainly depends on the width of the data word (16bit, 32bit or 64bit) and the number of data set. Although fast carry techniques are used in BPWS processings, clocks can run at a faster speed in the simple bit-serial PEs than in a complicated 32bit CPU. The bottle-neck between the CPU and other parts in a sequential system can also set a restriction on the overall system speed. For example, a Digital Alpha processor [23] may run at a 175MHz clock, but systems built with one Alpha CPU can not achieve substantial system performance improvements because of other slower devices around the CPU. This simple comparison also shows that the speed up factor by the BSWP processing principle also depends on the amount of data that can be processed concurrently.

With advances in technologies, it becomes practical to implement partial bit-parallel and word-parallel (BPWP) PE arrays, such as the 4-bit serial arithmetic elements in the RAP and the 8-bit serial PEs in the DDVA. The BPWP processing principle can still keep the advantages of the BSWP processing principle, but improve the performance by a few factors. It now takes only $3 \times 8 = 32$ cycles to process the same set of 1K 32bit data on a 4-bit partial BPWP PE array.

A bit-serial programmable PE for a configurable BSWP processing array will be designed for this project to demonstrate the viability of our ideas. A 4-bit partial BPWP PE having the same functions as this bit-serial version is intended to be developed for a partial BPWP array in the future.

2.12.5 System Timing Control Strategies

Since most of the massively parallel processing systems are designed with system level synchronous timing control, this inevitably sets an upper limit on the physical sizes of these systems that can be built without timing problems. Even within such a limit, it can be a difficult task to configure a system with global time constraints. The performance of a configured system has to be set by the worst case delay in the system. Asynchronous timing control without clocks at the system level offers

a prospective way to build a true scalable architecture. As will be illustrated later in this thesis, once a proper communication protocol and interface are designed, it becomes an easy task to scale a system up or down. The performance of an asynchronous system will be data dependent and can be measured on average. However, we also appreciate the advantages brought by the synchronous control method. Therefore, a timing control structure with a Globally Asynchronous communication network and Locally Synchronous computation modules (GALS) is proposed and implemented in this thesis.

Our approach is different from the DDVA architecture, which is based on the data-driven operation principle, in that we try to construct a simple and general GALS interface between a PE and a proper interconnection network. This GALS interface can be used to construct a GALS scalable system by connecting many existing synchronous logic modules together. But we also use data flow graphs (DFG) as an intermediate representation for algorithms because a DFG clearly describes a data flow procedure when data are processed on-the-fly in our hardware algorithms.

2.13 Summary

A wide diversity of typical massively parallel computing systems have been examined in this chapter. Most of the systems discussed have already had VLSI and system implementations. All these analysed systems demonstrate considerable interests in hardware algorithms. The emphasis of our analysis and comparison was on the interconnection networks, the PE functionalities, and the timing control of these systems. Efforts were made to find the common properties of configurable hardware algorithms. Problems in the design and application of these systems were also identified. In the rest of this thesis, a configurable architecture based on a parallel computation threads model and a GALS system timing control approach for algorithm embeddings will be proposed, and a VLSI implementation of such an architecture will be presented.

Chapter 3

Algorithmically Configurable Architectures

We have noted that many massively parallel processing systems are developed by the application demands. Therefore people tend to design an architecture more or less based on the empirical knowledge from the target applications. The result of this approach is often a special-purpose system which suits the object applications well. In our approach, we shall first look at the abstract theoretical computation and architecture aspects at the top system level, then we start to design our configurable array architecture for algorithm embeddings. This top-down approach will establish a clear guideline that we shall follow in the design of our target system and identify the problems and objective. As a result of this approach, a computation model, an architecture template, and a system level control strategy are established in this chapter.

3.1 Towards Algorithmically Structured Systems

As it has already been analysed, the performance of computers with the von Neumann model which ignores any algorithmic properties for simplicity and low system costs is fundamentally limited by the single CPU computation and the CPU–memory communication bottle-neck. All algorithms have to be designed in

a sequential way to allow only one operation at each step to go through the single CPU bottle-neck in these von Neumann computers.

It becomes clear in these massively parallel processing systems, described in chapter 2, that the development of algorithmically structured architectures is an effective solution to the inherent bottle-neck in the von Neumann computer model. Algorithmically structured architectures achieve high performance in three major ways:

1. Processing an algorithm with an array of PEs in parallel instead of a single fast CPU;

2. Setting up an interconnection network to connect these PEs to match the data and computation dependency of the algorithm so that any parallel properties of the algorithm are explicitly exposed. In contrast to the sequential algorithms for von Neumann computers, it is often required to flatten the algorithm to expose its parallel properties maximally;

3. Minimising the communication traffic between the main system memory and the algorithmically structured computation array.

The process of flattening an algorithm and setting up an array system to run the algorithm with the same data and computation dependency is called algorithm mapping or embedding. The system is said to be algorithmically structured by this algorithm mapping process.

It is worth noting that these systems are mostly used as attached single-tasking systems to a host computer of von Neumann type. This is because the physical architecture of these systems can only be algorithmically structured to one particular logical architecture at a time that will uniquely match the data and computation dependency of one algorithm. Computers based on the von Neumann model are still very efficient in the multi-tasking and time-sharing applications, particularly for the user interface.

Hence, principles of algorithm designs for algorithmically structured architectures differ very much from those for sequential machines. From the algorithm point of view, an algorithm for the solution of a problem should be designed in such a way that the inherent parallel properties of the problem will be exposed as much as possible so that these properties can be effectively made use of when the algorithm is mapped to its hardware counterpart. It is still possible that different algorithms may be found to solve the same problem, and different hardware systems can be implemented for one algorithm. From the architecture point of view, there are two ways to design algorithmically structured hardware. One is special hardware implementation by which an algorithm is first carefully designed and verified, then faithfully and exactly implemented through circuit design. It is important that algorithms designed for the special implementation are flattened enough so that hardware system designs are straightforward and can be easily tested. The other is to embed algorithms into an existing system. The architecture of the system is designed with some degree of adaptability to variations in applications so that the system can be algorithmically structured to run a particular algorithm.

In between algorithmically specialised architectures, which are tailored for one or a family of closely related algorithms, and general purpose systems, there are restructurable, reconfigurable, and programmable architectures. The system configuration (both functionalities and interconnections) of restructurable architectures can be set *a priori* by using special laser/fuse techniques [112,39,25] or PROM/EPROM writing mechanisms [105,56,36]. The usage of restructurable architectures is limited in that they are not reusable because of the nonrecoverable configurability, and thus they are not suitable for large scale algorithmically structurable architectures. Restructurable methods are often used in the RAM/ROM design to improve yield and in PROMs for permanent down-loading of code. Configurable architectures can be physically and repeatedly configured to various topologies. Logical configurations are implemented by changing actual

physical paths for data propagation through a series of circuit switches set open or closed either externally or internally, by changing the content of the static switch controls. There is intensive research carried out on this class of architectures. However, many architectures are proposed from the fault-tolerance point of view, and some specifically for systolic and wavefront arrays [85,29,79,108,61, 102,119,113,141,77,62,124,12]. It would be nice if the results achieved in fault-tolerant research could also be applied and further developed in algorithmically structurable architectures, so that flexibility, reliability and survivability could be integrated into one system. The flexibility of programmable architectures is often realised by methods similar to those used in programming, such as addressing, bussing, register/memory transfer [7], and packet-switching [49], instead of configuration via circuit switching. There are no physical changes in wires, paths, and switches. Thus, programmable architectures are logically configurable. Because of the complicated control required, the complexity of programmable architectures is higher than configurable architectures. A configurable architecture is an ideal compromise for application adaptability and moderate cost although there will be some performance penalties because of the delays caused by configuration switches. This delay penalty can be overcome by parallel processing performance advantages from hardware algorithms. The configurable approach also fits the objective of this project: the development of a high performance algorithmically structurable system which can be used as a testbed for high level algorithms or an attached system to perform computation intensive tasks for a host computer.

3.2 Hardware Algorithms

Generally speaking, a hardware algorithm is a hardware system where there is a direct dependency correspondence to the data, control and computation dependency of an algorithm described in another way, for example in a Data Flow Graph. A hardware algorithm system is represented as a pool of connected hard-

ware operators H_{op}. Each connection forms a communication path and represents a dependent relationship between two connected H_{op}s, one of which serves as a data source and the other as the destination for data output from the source H_{op}. There may also exist open connections to an H_{op}. An open connection has one end connected to an H_{op} and the other end open, which can be connected to other external systems. Hardware operators are the hardware counterparts of a set of arithmetic and logic operators, such as $+, -, \times, \div, NOT, AND, OR, XOR, comparator$ or even more complex functions such as $\sqrt{\ }$, logarithm, and exponent. Any complex computations can be decomposed into a set of orderly connected lower level H_{op}s given that a group of sufficient low level H_{op}s is available. An H_{op} can also be regarded as a function stored in circuits called function caches.

A hardware algorithm can be mathematically defined as a mapping function — $HA = \{F : \{0,1\}^n \rightarrow \{0,1\}^m\}$, where F may consist of a set of subfunctions $f_i = \{f_i : \{0,1\}^{n_i} \rightarrow \{0,1\},\ i = 0,1,2,\cdots,m\}$. The domain $\{0,1\}$ may be logical 0 and 1, true and false, or binary numbers. Each mapping subfunction f_i, no matter how simple or complex it may be, is completely implemented in hardware as a hardware operator, i.e. $H^i_{op} = \{f_i,\ i = 0,1,2,\cdots,m\}$.

Since a complex function can always be decomposed into a set of ordered simple functions, it will be very useful to construct a finite set of primitive functions, which are chosen from a set of most frequently used essential functions. An essential function is a function from which outputs can be obtained through only a single logical transformation step from the values of the input variables, without caring about the internal detailed physical steps. Accordingly, a finite set of hardware operators $P = \{P_i | P_i : \{0,1\}^{n_i} \rightarrow \{0,1\},\ i = 0,1,2,\cdots,p\}$ called primitive hardware operators (H_{pop}), corresponding one for one with the selected primitive functions, can also be defined. A primitive hardware algorithm system can be built with this set of H_{pop}. An H_{op} of higher complexity required by an algorithm can be constructed from this finite set of primitive H_{pop}s. Thus, a primitive hardware algorithm system can be represented by a primitive system construction

graph (SCG) in which nodes represent primitive hardware operators H_{pop} and edges correspond to the connections among H_{pop}s.

The cost, system complexity, and the requirement for regular architectures suggest that it is not viable to build a system with arbitrary H_{op}s for algorithm embeddings. A compromise approach is to select such a set of primitive H_{pop}s with reasonable and similar complexity to keep the system cost, complexity and regularity under control. Any high level SCG can be initially drawn based on the normal H_{op}s and then refined to a primitive SCG in which only H_{pop}s in the primitive H_{op} set P are used. In this way, algorithms of any complexity can still be mapped into a primitive hardware algorithm system provided that there are enough H_{pop} resources.

3.3 Computation Architectures

Architectures for configurable hardware algorithms can be constructed with a set of hardware operators $H_{op} = \{H_{op}^i : domain1^i \rightarrow domain2^i,\ i = 1, 2, 3, \cdots, n\}$ which are connected by an interconnection network. Either the connection among H_{op}s or the functionality of H_{op}s or both can be designed configurable in order to embed the data, control and computation dependency of a particular algorithm.

3.3.1 Dimensionality and Connectivity

The network in which hardware operators in a system are connected can be analysed from two aspects: dimensionality and connectivity, ($\mathcal{D}$, $\mathcal{C}$). Dimensionality $\mathcal{D}$ is a base vector upon which all of the elements in the system can be logically addressed: $\mathcal{D} = (x_0, x_1, x_2, \cdots, x_d)$. Connectivity $\mathcal{C}$ is a transformation that gives the addresses of some elements with which an element addressed by $Y = (y_0, y_1, y_2, \cdots, y_d)$ is directly connected: $\mathcal{C} = \{\mathcal{C}_i(Y) \mid Y \xrightarrow{\mathcal{C}_i^j} N_i^j,\ N_i^j =$

$(n_{ij}^0, n_{ij}^1, n_{ij}^2, \cdots, n_{ij}^d)$, $i = 1, 2, \cdots, k_c$, $j = 0, 1, 2, \cdots, m_i\}$. Thus, the total number of neighbours of the element Y is $\sum_{i=1}^{k_c} m_i$. The connectivity of elements on boundaries may be different from that inside a system. $\mathcal{C}$ may also be different due to different orientation of $\mathcal{D}$ and network routing/layout-plan. A transformation can be found if merely $\mathcal{D}$ is rotated. If elements in an array are not connected on a regular basis, the $\mathcal{C}$ of each element may vary one from another.

This concept can even be extended to a conventional single processor system which can be viewed as a point in the space domain and as a linear array in the time domain. The processor is shared by multiple tasks in the time domain that is divided into small segments of non-overlapped periods. A traditional processor will process only one task in one time segment. Pipelines are used extensively in modern processors where multiple operations can be carried out in a pipeline. This can still be modelled by a two dimensional array in the time domain. A linear array of connected n elements is one dimensional system, $|\mathcal{D}| = 1$. Each element inside the linear array is connected only with its immediate previous and next element while there are two boundary elements each of which is connected with only one element in the array i.e. $\mathcal{D}_1 = x$, $\mathcal{C}_2 = \{\mathcal{C}_i(x) \mid \mathcal{C}_i = x \pm 1,\ i = 1, 2\ \ x = 2, 3, \cdots, (n-1)\}$. Boundary conditions sometimes are different. In the above example, boundary conditions are $\mathcal{C} = x + 1,\ if\ x = 1$, and $\mathcal{C} = x - 1,\ if\ x = n$. There will be many different possible topologies if a set of hardware operators are to be arranged in a two dimensional space, $\mathcal{D}_2 = (x, y)$. For example, $\mathcal{C}_4 = \{\mathcal{C}_i(x, y) \mid (x, y) \xrightarrow{\mathcal{C}_i(x,y)} [(x \pm 1, y),\ (x, y \pm 1)],\ i = 1, 2, 3, 4\}$ is a mesh. $\mathcal{C}_8 = \{\mathcal{C}_i(x, y) \mid (x, y) \xrightarrow{\mathcal{C}_i(x,y)} (x \pm 1, y \pm 1),\ i = 1, 2, 3, \cdots, 8\}$ represents an array of 8 nearest neighbour connections extended from $\mathcal{C}_4$. $\mathcal{C}_6 = \{\mathcal{C}_i(x, y) \mid \mathcal{C}_4,\ (x, y) \xrightarrow{\mathcal{C}_i(x,y)} [(x-1, y-1),\ (x+1, y+1)],\ i = 1, 2\}$ is a hexagonal array. A four level complete binary tree can be simply laid out as a H-tree: $\mathcal{C}_{H-tree} = \{\mathcal{C}_i(x, y) \mid (x, y) \xrightarrow{\mathcal{C}_i^0(x,y)} (x, y \pm 2),\ (x, y) \xrightarrow{\mathcal{C}_i^1(x,y)} (x \pm 2, y),\ (x, y) \xrightarrow{\mathcal{C}_i^2(x,y)} (x, y \pm 1),\ (x, y) \xrightarrow{\mathcal{C}_i^3(x,y)} (x \pm 1, y),\ i = 1, 2\}$. A cube in the three dimensional space $\mathcal{D}_3 = (x, y, z)$ can be expressed as $\mathcal{C}_3 = \{\mathcal{C}_i(x, y, z) \mid (x, y, z) \xrightarrow{\mathcal{C}_i(x,y,z)} [(x, y \pm$

$1, z)$, $(x \pm 1, y, z)$, $(x, y, z \pm 1)]$, $i = 1, 2, \cdots, 6\}$. Most of the systems built so far are in the $|\mathcal{D}| = 2$ space. There are also attempts to construct real $|\mathcal{D}| = 3$ systems by developing novel three dimensional semiconductor processing techniques [45]. For systems in higher logical dimension spaces ($|\mathcal{D}| > 3$), a common approach is to map the space with $|\mathcal{D}| > 2$ into a planar space $|\mathcal{D}| = 2$ which is often inefficient in hardware utilisation because of the higher $|\mathcal{C}|$.

In configurable systems, the placements of H_{op}s are fixed with an initial interconnection network. The initial physical dimensionality and connectivity of a system will certainly affect the adaptability and efficiency of algorithm embeddings. The logical dimensionality and connectivity of a configured system for an algorithm are by no means the same as the initial physical ones. An architecture with unconfigured initial physical dimensionality and connectivity is called a blank architecture.

There is still a long way to go before three dimensional processing technologies can be practically used. Linear arrays are not suitable for embedding higher $|\mathcal{D}|$ structures. Therefore a $|\mathcal{D}| = 2$ configurable array will be designed. The only way to increase the adaptability of a two dimensional array is to increase $|\mathcal{C}|$ for each element in the array. If there is a total of N elements in an array, $|\mathcal{C}| \leq (N - 1)$. A completely connected system, where $|\mathcal{C}| = N - 1$, is too expensive to implement for a large N. We shall choose a proper $\mathcal{C}$ in conjunction with the establishment of a computation model.

3.3.2 Configuration Methods

From the configuration viewpoint, two kinds of configurable hardware algorithm system are identified.

- Statically configurable systems: in which the physical configuration of a system, e.g. the embedding of an algorithm, can only be done in a sepa-

rate configuration phase before the execution of the algorithm, or after the completion of the execution of an algorithm. Configuration data are loaded *a priori* and cannot be changed either partially or completely during the execution of an algorithm. Statically configurable systems are simple and not prone to hazards caused by configuration actions. Most of the existing flexible systems for hardware algorithms are of this type.

- Dynamically configurable systems: in which partial or complete configuration of a system can be done dynamically during the execution of an algorithm. A dynamic configuration approach can be adopted in both logical (programmable) and physical configurable architectures. Dynamically configurable architectures are very suitable for running multiphase algorithms whose data and computation dependencies may be different from one phase to another. However, it is expensive and difficult to design dynamically configurable systems. Extensive fault-tolerance techniques must be used in such systems because they are prone to malfunctions. Architectures proposed in [90,81,125,49] are designed for dynamic configurability.

To run an algorithm in a statically configurable system, three distinct phases are required. The mapping phase is a procedure that takes an algorithm specification either in a high level programming language form or an algebraic form, and converts, under a set of constraints, the data and computation dependency of the algorithm into a configuration data stream which sets the logical network topology and functions of H_{op}s in the target architecture. The configuration phase is an algorithm embedding process which loads the generated configuration data stream into the system and sets a blank system to a specific logical architecture to run the algorithm. The actual execution of the algorithm will be carried out in an execution phase which is initiated when the computation is needed and all of the inputs to the computation are ready. The only interaction between a configured

system and its host computer or other external devices is data inputs and results output.

The execution of algorithms in dynamically configurable architectures is much more algorithm-dependent. After the initial configuration of an architecture and the start of the execution of an algorithm, the logical architecture of the system at any time point may be determined by the results generated before this time point, or by globally broadcast instructions, or there may be a special configuration cycle between two consecutive phases during a multiphase execution.

The statically configurable method is chosen for our configurable hardware algorithms because the emphasis of this research is on the architecture issue instead of dynamic system scheduling, and we are targeting those applications which are mostly single phased.

3.4 Computation models for Hardware Algorithms

Designs of hardware algorithms can vary over a very wide range of applications. A general algorithmic approach is preferred over the empirical architecture approach for the design of our configurable hardware algorithms. In this algorithmic approach, a basic computation model is defined as a bridge between an architecture and applications. This computation model is an abstract representation of the architecture. It also sets an abstract algorithmic structure that a specified algorithm is transformed before it is embedded. This approach is most likely to get the best from both the architecture and algorithms. There are some computation models established for the design of hardware algorithms. Some of these models can be used generally in many different applications. Some of them, however, can only be used under certain special conditions. Systolisation of algorithms is a typical example of such an approach.

3.4.1 Combinational Hardware Algorithms

This is the simplest type of hardware algorithm which can be implemented simply in pure combinational logic. In this kind of system, there will be no timing and clocking controls over signal and data flows. Output data will emerge and become steady after a finite period of delay (latency) once input data are validated. An n-bit combinational adder is such an example. H_{op}s implemented in combinational logic are called combinational H_{op}s.

3.4.2 Systolic Algorithms

Systolic algorithms are also frequently called systolic arrays, referring to the mixture of a computation model and an architecture model. Since this model was proposed by Kung in late 70's [68,69], there have been enormous efforts made to systolise many application algorithms, and various systolic systems have been designed and implemented for these algorithms. The systolic model, as depicted in figure 3–1, was named after the blood circulation system in a human body. The memory corresponds to the heart, PEs are similar to organs, and communication channels like blood vessels. Data are circulated from the central memory to PEs, processed by consecutive PEs either in linear or two-dimensional forms, and only final results are returned to the central memory. All of the data movement is pulsed by a central global clock in a lock-step manner analogous to the blood flow pumped by heart-beats.

The function of a PE can be expressed as $\mathbf{Y} = F_{PE}(\Phi, \mathbf{X})$ where Φ is the synchronisation clock vector, $\mathbf{X} = (x_0, x_1, x_2, \cdots, x_n)$ is a set of inputs to the PE, and $\mathbf{Y} = (y_0, y_1, y_2, \cdots, y_m)$ is a set of outputs from the PE. The central memory accesses can be expressed as $\mathbf{D} = M_R(\Phi, \mathbf{A})$ and $M_W(\mathbf{A}) = \mathbf{D}(\Phi)$ respectively, where $\mathbf{A} = (a_0, a_1, a_2, \cdots, a_l)$ is the memory address and R indicates a memory read and W a memory write operation, $\mathbf{D}$ is the data to be read from or written

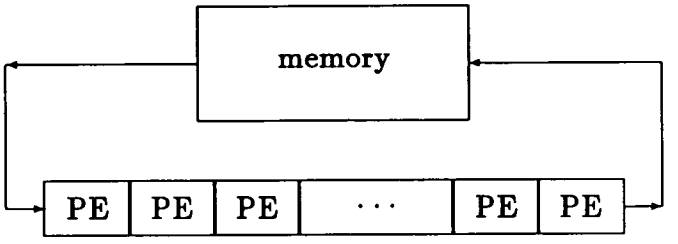


Figure 3–1: A linear systolic computation model

into the memory at address **A**. Then, systolic computations performed in a linear array of $(n+1)$ PE's can be represented as follows:

$$\mathbf{Y} = F_{PE_n}(\Phi, \cdots F_{PE_3}(\Phi, F_{PE_2}(\Phi, F_{PE_1}(\Phi, F_{PE_0}(\Phi, M_R(\Phi, \mathbf{A}_1)))))\cdots) \quad (3.1)$$

$$M_W(\mathbf{A}_2) = \mathbf{Y}(\Phi) \quad (3.2)$$

Note that function 3.1 is not a recursive form since each PE may perform different operations or perform the same operations on different sets of input data simultaneously. The essential principle behind this computation model is that once a datum is retrieved from the memory, it and its relevant intermediate results will be used and reused as much as possible, while they are flowing through the systolic array, by all of the operations which need them as operands, without any further access to the memory until final results are obtained. The systolic model enables a system to achieve a sustainable high computation bandwidth while only moderate I/O bandwidth is required. Instead of exclusive memory read and write in the conventional single memory single CPU model, it is necessary to be able simultaneously to read from and write to the memory module (at different address) in the systolic model. The systolic model is particularly suitable for computation intensive algorithms with regular data structures. It has been demonstrated to be more efficient to process matrices in this on-the-fly mode than processing matrices sitting in PE local memories as in DAP and MPP.

The systolic model is different from simple pipeline processing techniques in that it can be used in a multi-dimensional system with multiple data-flow directions. Central memory access is strictly not allowed for any intermediate data.

3.4.3 Computational Wavefronts

Because the systolic model is synchronously timed, it is very important to keep the correct data flow pace at each PE so that the right data can encounter each other at the right time and in the right PE with the right control. Great care must be taken in the design and implementation of practical systolic systems to ensure this strict lock-step time requirement is satisfied. Clock skews are one of the major factors that may cause malfunctions [31]. It is a common practice to distribute clocks with great care and slow down clocks at the expense of reduced performance [74]. Even worse, these measures sometimes cannot solve the problem caused by timing mismatch [46,135]. To overcome these synchronous design difficulties, a computational wavefront model has been proposed [73,75,72,76].

The wavefront model is based on wave propagation phenomena. A wavefront propagation example is waves in water where vibrations from a source will generate a wavefront immediately around the source; this wavefront will again activate another wavefront next to it and so on. Even if the source stops vibrating, wavefronts should continue to propagate further away from the source provided that there is no energy dissipation. Therefore, wavefront propagation is naturally an asynchronous model. Figure 3–2 depicts diagonal computational wavefronts in a two dimensional computing array where the central control, acting as the source of vibration, is located at the upper-left corner of the array. The central control initialises wavefronts and issues instructions to the array. One computational wavefront w can be generated along a diagonal of $\{PE_{ij} : i + j = w,\ i, j = 0, 1, 2, \cdots, N,\ w = 0, 1, 2, \cdots, (2N - 1)\}$ if and only if there are no computations existing in the $w + 1$ wavefront and computations in the $w - 1$ wavefront are

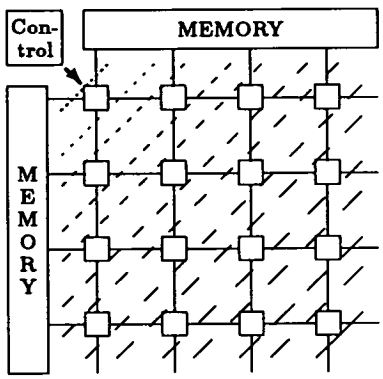


Figure 3–2: Computational wavefronts and their propagation

all completed. Thus the propagation of computational wavefronts is completely localised by asynchronous wavefront generation.

Note that this simple wavefront model strictly confines only one "source" and one set of regular dependent wavefronts to exist, so that interferences among different set of wavefronts can be avoided. This is also the fact which restricts the application of this model to only a very limited range of problems which can be solved by the propagation of regular computation wavefronts upon regular data structures.

Because the wavefront computation model does not use global clocks to control the data movement, data flow will be determined by states in PEs and wavefronts. A PE state s_{PE} is determined by its input state s_i, output state s_o and computation state s_c. The input and output state in a PE depend on whether there are valid data presenting at its inputs and outputs respectively. A PE's computation state reflects whether the main functional block in the PE is busy or idle. So, s_{PE} has eight possible states as shown in table 3–1. It must be ensured that all PEs on the same wavefront w have the same state to propagate a regular wavefront. This can be expensive to implement. Hence the state of a wavefront w can be defined as $s_w = s_{PE_w}$.

s_i	s_c	s_o	s_{PE}
1	0	0	I
0	1	0	C
0	0	1	O
1	1	0	I + C
1	0	1	I + O
0	1	1	C + O
1	1	1	I + C + O
0	0	0	idle

Table 3–1: PE states, I = INPUT, C = COMPUTE, O = OUTPUT

The function of each PE on a wavefront w can generally be defined as $\mathbf{Y} = F_{PE}(\mathcal{W}, \mathbf{X}, M(\mathbf{A}))$, where $\mathbf{X} = (x_0, x_1, x_2, \cdots, x_n)$, and $\mathbf{Y} = (y_0, y_1, y_2, \cdots, y_m)$, $M(\mathbf{A})$ is the memory access operation at address $\mathbf{A} = (a_0, a_1, a_2, \cdots, a_k)$, $\mathcal{W}$ is determined by the state of $(w-1)$, w, $(w+1)$ wavefronts respectively, $\mathcal{W} = S(\mathbf{s}_{w-1}, \mathbf{s}_w, \mathbf{s}_{w+1})$.

One diagonal computational wavefront can be described as a set of functions conditioned by the computation status of itself, and its previous and next wavefront:

$$\begin{aligned} F_w \;=\; & \{F_{PE_{ij}} : (0,1)^{n_{ij}} \xrightarrow{\mathcal{W}} (0,1)^{m_{ij}}, \\ & i+j=w,\ i,j=0,1,2,\cdots,N,\ w=0,1,2,\cdots,(2N-1)\} \end{aligned} \tag{3.3}$$

3.4.4 Non Control-Driven Computations

Just like the global clock that keeps the pace of data flow in a synchronous system, a central control unit in a control-driven system controls the flow of every computation step i.e. a computation will not be started until a control signal from the

central control unit is issued for it. The program counter in conventional sequential computers is a typical control-driven example, where counter registers are used to keep the current execution point and relevant information to determine what the next step is. Control-driven computations are very common in computing systems because it is simple to design a control schedule at system level.

When the control-driven method is applied in parallel processing architectures, it still works well for problems with regular data structure. However, if data structures with a certain degree of irregularity are met, a control-driven parallel processing system will have low execution efficiency and low hardware efficiency because of the idle waiting states required to match irregularities in data dependencies with time. To overcome these difficulties, other computation driven methods have been proposed, e.g. data-driven, demand-driven, and message-driven [5,1,136,66, 20]. All of these approaches can be described by a general firing rule: an action (firing) will be activated if and only if a set of conditions is satisfied. In the control-driven method, the conditions are all held and checked in a control unit and firing signals (maybe accompanied by instructions) are issued to execution units. In a data-driven computing system, provided that there is a set of connected function modules, each module has a set of instructions to be executed, and the order in which instructions are executed is data-dependent, i.e. the firing (execution) of an instruction in a module is uniquely determined by the condition that all of the operands required by the instruction are available and the destination for the results of that instruction is ready to accept the results generated after the firing. Thus condition checking and action firing are localised to each module. Concurrent action firing in several modules can be expected. Pipelining techniques can still be applied in data-driven systems. With the control-driven method, it is user's responsibility to uncover inherent parallel properties of an algorithm which is often not a trivial task. The data-driven computation matches, and can make good use of, the parallel properties of algorithms. The demand-driven computation is similar to the data-driven principle but the condition for an instruction firing in a

module is a request for output data from the next module. The advantage of the demand-driven computation is that, except for the required actions to be done, no other unnecessary actions are activated at all. Demands are first traced from requests for some results back to the right inputs, after which real computations start to flow along the activated paths until the requested results are obtained.

3.4.5 Multiple Threads Computations

Besides applications with regular data structures and regular communication structures, there is still a large class of computation-intensive applications which do not possess such regularity in their data dependencies, communication structures and computation distribution. There are no transformations to convert them to be processed on architectures designed for regular algorithms, neither is it possible to fit them into computation models such as systolic or wavefront model. However, these computation-intensive tasks still have high degrees of inherent parallelism which can be exploited to reduce processing time.

A proper computation model, which can make the best use of parallel properties of irregular algorithms and result in an efficient embedding of an algorithm into such a primitive configurable system in the light of irregularities, is required. A close examination of the characteristics of irregular algorithms is carried out in the following. As a result, a multiple threads computation model is established for irregular algorithm embeddings.

There are three different types of irregularity in an algorithm:

1. Irregular data distributions. Data are distributed irregularly either in the space or time domain. Therefore they cannot be vectorised to generate regular data flow patterns;

2. Non-uniform computation distribution. Computations which can be carried out concurrently at any step of an algorithm may be substantially differ-

ent from each other and distributed randomly in a parallel system. Thus, regularly shaped computational wavefronts cannot be found at one step or throughout the entire computation process;

3. Non-regular communication patterns. Occasional feedback, conditional execution, and non-neighbour operand requests result in irregular communication patterns that create the toughest task for efficiently parallelising an algorithm with the control-driven method.

In order to exploit maximally inherent implicit parallel properties in such applications, algorithms may be designed or flattened on the basis of the fine-grained parallel processing principle which processes complex tasks with a group of simple basic arithmetic and logic functions. This fine-grained parallel processing principle matches exactly the primitive configurable hardware algorithm system construct illustrated in section 3.2.

Data Flow Graphs

A data flow graph (DFG) is a very flexible, clear and yet abstract representation for the visualisation and analysis of various dependencies in an algorithm. Its flexibility lies in that it is very easy to divide a DFG into a set of disconnected subgraphs or merge several DFGs into one large DFG, and the level of abstraction of a DFG can also be easily scaled by simply merging subsets of nodes to supernodes of coarser granularity or by expanding a node to a subgraph composed of nodes with finer granularity. On the other hand, it is not a difficult task to convert an algorithm specified in a high level language or algebraic form to a DFG, and then optimise or transform the algorithm based on its DFG. Thereafter, it is relatively simple to map the DFG to a hardware system. Therefore, the data flow graph is chosen as an intermediate representation for the analysis, optimisation and embedding of irregular algorithms into configurable architectures.

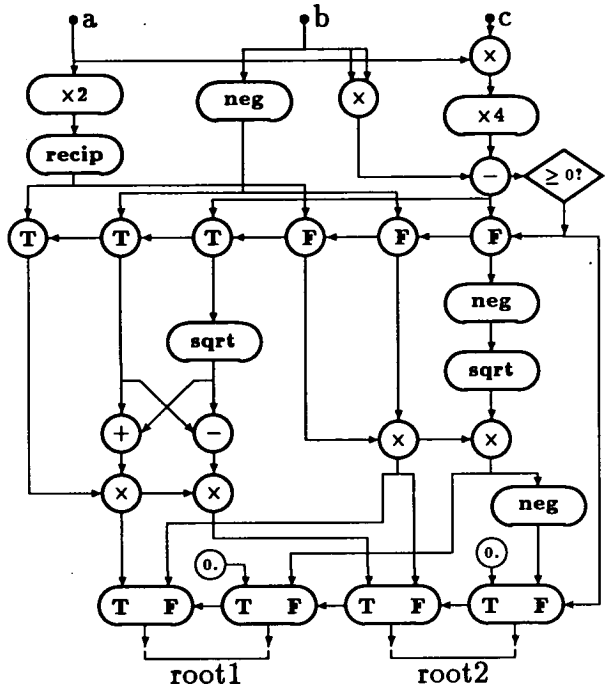


Figure 3–3: The data flow graph of equation 3.5

A DFG is a tuple $G = (V, E)$ where V is a set of vertices (often called nodes) in a graph G, and E is a set of edges connecting a subset of nodes $v \in V$. A DFG is also a directed graph so that every $e \in E$ has an associated direction. A node in a DFG denotes actions to be done to data on its input edges, and directed edges represent paths through which data can flow and in which data are held. Figure 3–3 is a DFG example which shows an algorithm for the solution of a quadratic equation:

$$ax^2 + bx + c = 0 \tag{3.4}$$

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad i = 1, 2 \tag{3.5}$$

Provided that there is a finite complete set of primitive nodes V_p from which any nodes of higher complexity can be constructed, then a DFG $G = (V, E)$ can be flattened to an equivalent primitive graph $G_f = (V_f, E_f)$, $V_f \subseteq V_p$, by expanding every node $(v \in V) \wedge (v \notin V_p)$ to a subgraph $G_v = (V_v, E_v)$, $V_v \subseteq V_p$

so that the function of the subgraph G_v is equivalent to the function of node v, $F_v = F_{G_v}$. A DFG can also be compacted to a more abstracted equivalent form by merging subsets of nodes $V_i \subset V$ and edges $E_i \subset E$ which comprise subgraph $G_i = (V_i, E_i)$ into a supergraph $G_s = (V_s, E_s)$ so that $F_{v_s^i} = F_{G_i}$, $v_s^i \in V_s$, $E_s \subset E$, $E_s \cup E_i = E$, $i = 1, 2, 3, \cdots, \sum_i \| V_i \| = \| V \|$.

Computation Threads

From the simple example illustrated by equation 3.5 and the corresponding DFG shown in figure 3–3, some interesting characteristics can be noted.

- Nonuniform node granularities. A finite set of nodes with different granularities in circuit complexity and time complexity can be found in the DFG in figure 3–3 since they are directly converted from equation 3.5. Even if an original DFG is flattened to a primitive DFG upon V_p, granularities of nodes $v_p \in V_p$ may still be different in circuit and time complexity.

- Irrelevant computation wavefronts. At any time during the computation, it is often the case that a twisted computation front is found. A wavefront may cross another wavefront. Mixed sequential and parallel dependencies are found, for instance, the calculation of $\frac{1}{2a}$, $-b$, b^2, $4ac$ can be performed concurrently, but the square root of $b^2 - 4ac$ has to be computed after the evaluation of b^2 and $4ac$.

- High ratio between the amount of intermediate and I/O data. Only three inputs, coefficients (a, b, c), and the output of two roots, each consisting of a real and an imaginary part, are required for I/O. The amount of intermediate data depends on the number of internal nodes and the level of abstraction in a DFG. There are 23 intermediate data in figure 3–3. The large amount of intermediate data will impose a high I/O bandwidth requirement and will

be inefficient if they are stored and retrieved to and from a global memory every time they are generated and needed.

- Optimisations. It is possible to optimise and decompose some of the nodes in figure 3–3. For example, $\boxed{\times 2}$ and $\boxed{\times 4}$ node can be simplified to a shifter. A multiplier node can be decomposed into a subgraph which implements multiplications by additions.

It is apparent that there are always some sequential data dependencies in an irregular algorithm. It is possible to extract a sequential data dependency into an abstract form called thread. If there is only one thread that can be extracted from an algorithm, it is a pure sequential algorithm. The only technique to improve the throughput of a system running this algorithm is to process these sequential steps in a pipeline. However, if multiple threads can be extracted from an algorithm, it is highly likely to run the steps in different threads concurrently.

A computation thread is defined as a single sequence of connected nodes $T = \{ (V_T, E_T) \mid (V_T \subseteq V) \wedge (E_T \subseteq E) : v_i \xrightarrow{e_i} v_{i+1},\ i = 1, 2, \cdots, (n_T - 1),\ v_i \in V_T,\ e_i \in E_T,\ n_T = \parallel V_T \parallel = (\parallel E_T \parallel + 1) \geq 1\}$. Outputs from a thread may not necessarily come from the last node v_{n_T}, and if inputs to a thread are not to the first node v_1, it must be ensured that they are present at the node before the computation front propagating in the thread reaches the node. Thus, a tight dependency is defined as an edge $(e \in E) \wedge (e \in E_T)$, a loose dependency is defined as an edge $(e \in E) \wedge (e \notin E_T)$. A pair of tightly coupled nodes are two nodes in V_T connected by an edge in E_T. According to these definitions, the DFG in figure 3–3 can be abstracted to a computation thread graph (CTG) as shown in figure 3–4 where dash lines represent loose dependencies, circles and ovals connected by solid lines comprise threads. The length of an oval represents the computation time complexity of a node.

There are some important properties in computation threads. The number of nodes in a thread may be $1 \leq \parallel V_T \parallel \leq \parallel V \parallel$. A thread does not necessarily

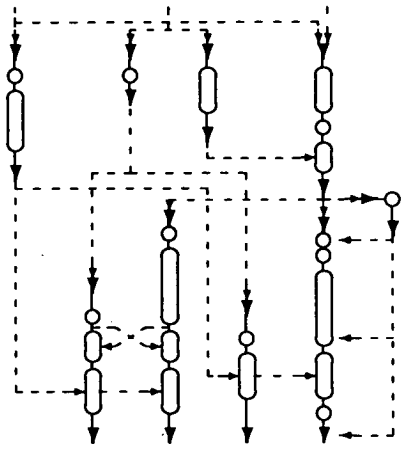


Figure 3–4: The computation thread graph for figure 3–3

run from an input node to an output node in a DFG; it can start from any node and terminate at any node. One thread may fork to several threads, and several threads may end up with one thread by joining at one node. Every thread is directed, i.e. there is an unique data and computation propagation direction in one thread. Selected sequential steps of an algorithm have corresponding nodes inside a thread which have to be activated one after another. The activation of a thread may depend on the arrival of outputs from some other threads. Once a thread is activated, it runs on its own and multiple threads may run concurrently. Therefore parallel scheduling can be automatically detected and carried out at run-time with this multiple threads computation model. Another important indication is that when a DFG is to be mapped into a hardware system, tightly coupled nodes on a thread should be put in physically close positions.

Two parameters are defined for a CTG. The length of a thread T, $length(T)$, is defined as the time taken from the activation of T to when outputs are available from T. $length(T)$ is a measure of the computation time of T. $length(T)$ is determined by the number and time complexity of nodes, and the data propagation delay between nodes in the thread. The critical path $Cpath$ of a CTG is defined as an input to output path comprised of a set of dependent threads T_i having

maximum $\sum_i length(T_i)$. Thus, *Cpath* measures the computation time of an entire DFG. There can be more than one CTGs extracted from a DFG. Two constraints, cost or time, can be used when a CTG is extracted. With the cost constraint, the amount of hardware resource is set. Some of the threads have to be merged to share the available hardware resource. With the time constraint, a reasonable time length is set. The *Cpath* of an extracted CTG must be within the time constraint. In practice, a DFG should be flattened to a primitive DFG first before a constrainted CTG extraction is carried out.

As an abstract representation, the multiple threads computation model clearly splits the sequential and parallel properties mixed in a DFG and simplifies nonuniform node granularities to the time complexity of nodes. Sequential dependencies are explicitly expressed by threads and parallel processing is automatically scheduled at run-time. This model is a bridge between a DFG and an algorithm mapping process. It provides a framework to optimise easily an algorithm with constraints. It also contains the architecture information implying placements of nodes with delay constraints. It describes the principle of processing data on-the-fly for algorithms concisely in both space and time domain. Because the multiple threads computation model is much more relaxed compared with the many restrictions in other computation models for regular algorithms, such as the systolic and wavefront model, it can be applied to model arbitrary algorithms. This model is therefore the basis for the analysis and optimisation of algorithms to be mapped into our algorithmically configurable architecture.

3.5 Timing Control Structures

The choice and design of a proper timing control structure for a system is a vital and yet a very practical issue. Some general considerations on synchronous and asynchronous timing methods can be found in [120,95,135,46]. Before designing

an architecture based on the multiple threads computation model, it is necessary to examine further the specific timing requirements from the model and the architecture. Discussion will thus be focussed on the best timing strategy for the multiple threads computation model and large scale modular VLSI architectures.

3.5.1 Clocks and Clock Skews

An ideal clock ϕ is a periodic function of time $\phi(t)$ which can be defined as following:

$$\phi(t) = \begin{cases} 1 & t_0 + nT \leq t \leq t_0 + nT + t_H \\ 0 & t_0 + nT + t_H \leq t \leq t_0 + (n+1)T \end{cases} \quad . \tag{3.6}$$

where $n = 0, 1, 2, \cdots$, $T = t_H + t_L$ is one clock period in which t_H is the time interval for one clock pulse at logical value 1, and t_L is the time interval between two consecutive pulses. For non-ideal clocks, there is a continuous transition function between logical 1 and 0 instead of an abrupt change of levels.

In synchronous systems, all of the operations and data movements are synchronised with a system wide global reference, usually a system wide global clock. A global clock must satisfy that clock signals at any two physical points in a synchronous system are logically equivalent at any time. Global clock events in a synchronous system serve two purposes as sequence references and also time references. As a sequence reference, a transition (event) between two levels of a clock defines the instance at which the system may change state so that random state changes and interferences can be eliminated. As a time reference, the interval between clock level transitions defines a time region during which data can either move between successive processing stages or are processed in stages isolated from others. In other words, a clock signal can be viewed as a guard which controls when and what is to be done or not to be done in a synchronous system. Two-phase nonoverlapping clocks ϕ_1 and ϕ_2, which always satisfy $\phi_1(t) \cdot \phi_2(t) = 0$ at any time t, are one of the most commonly used clock control schemes.

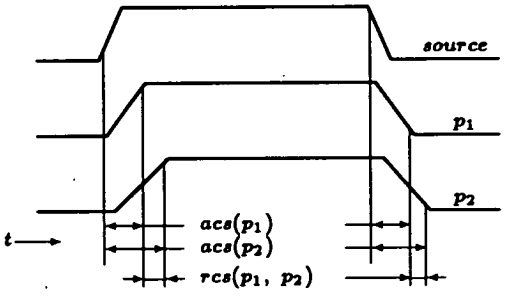


Figure 3–5: Absolute and relative clock skews

An optimised clock distribution scheme must be used to generate logically equivalent clock signals across an entire system to ensure the correct system operations with a clock speed as fast as possible. However, no matter what kind of clock distribution scheme is used in integrated circuits, clock skews are physically unavoidable. Clock skews can be caused by many factors such as signal propagation delays on wires, capacitive loading variations at different points on clock distribution paths, and variations in device and process parameters e.g. the MOS transistor threshold voltage V_T. A general clock distribution principle is that the clock from a root or source clock generator is equally buffered and extended with equal wire length to each synchronous area with roughly the same capacitive load. A clock distribution tree is such an example. From a clock distribution, an absolute clock skew (ACS) can be defined as the maximum time difference of the high/low state at a point p on a clock distribution path from the low/high state at the clock source s (this includes the propagation delay). A relative clock skew (RCS) can be defined as the maximum time difference of the high or low state at two different points (p_1, p_2) on clock distribution paths as shown in figure 3–5. Thus,

$$ACS(p) = max(t_p^h - t_s^l,\ t_p^l - t_s^h) \tag{3.7}$$

$$RCS(p_1, p_2) = \mid max(t_{p_2}^h - t_{p_1}^h,\ t_{p_2}^l - t_{p_1}^l) \mid \tag{3.8}$$

$$RCS(p_1, p_2) = \mid ACS(p_2) - ACS(p_1) \mid \tag{3.9}$$

where t^h and t^l are the time points when the clock goes high and low; subscript s is the clock source; p, p_1 and p_2 are physical points on clock distribution paths.

The worst case ACS reflects the time interval required to charge or discharge a complete clock distribution network from one clock source, while the data communication rate between two connected modules in a synchronous system is heavily influenced by the RCS in the two modules. To determine the maximum frequency of a synchronous clock, both worst case ACS and RCS should be considered. In [74], an $O(n^3)$ clock skew is derived from an $n \times n$ 2-D processing element array. But this clock skew is in fact an ACS. When the size of integrated systems increases, the ACS increases much faster than the RCS. However, RCS plays the major role in determining the timing performance of a synchronous system. An optimised clock distribution must guarantee that the RCS between two clock signals at any two physical locations in a system is negligible compared with the clock period T.

3.5.2 Computing without Clocks

People recognise that there are many advantages in self-timed logic systems with asynchronous timing control to replace global control clocks in synchronous systems [95,128]. However, most designers tend to choose to design synchronous systems. This is mainly because it is more expensive to design an asynchronous system than a synchronous one with medium system complexity. It is also simpler to schedule system operations with a periodic global reference. Because it is becoming more and more difficult to distribute a proper global clock network over a large area of silicon and it is increasingly expensive to design an efficient schedule for a synchronous system with millions of transistors [23], interest has revived in asynchronous design methods [98,57,92,91,50,3,27,6,26,138,82]. The latest effort is the AMULET project [38,106] to design an asynchronous version of the ARM (Advanced Risc Machine) processor. Methodologies are also developed for the automatic synthesis of asynchronous systems [99,11,43].

A fundamental difference in asynchronous design methods from synchronous

designs is that they permit "asynchronous thinking" in design processes. This allows a designer to focus on the functionality of a module and not its timing details. If correctness and reasonable execution time were the only criteria for the acceptance of a design, the asynchronous design approach would be a very good choice over synchronous designs [42]. A self-timed system is built by decomposing the system into a set of combinational logic blocks and inserting an asynchronous hand-shaking control between each pair of connected blocks. The decomposition of such a system often descends to a level that each of its building blocks performs relatively simple functions with few inputs and outputs because the complexity of hand-shaking circuits increases drastically with the number of inputs and outputs [116]. There are two consequences of this design principle. First of all, since there is no global clock control in an entire system, the system performance is data dependent at run-time. The performance of self-timed systems is measured by an average instead of maximum clock frequency for worst case delays in synchronous systems. Secondly, there are extra delays caused by hand-shaking logic at each logic block in a self-timed system. The circuit complexity ratio between the hand-shaking control logic and the actual computation block is also high which implies a relatively low area efficiency for computation logic.

3.5.3 Separately Timed Communications and Computations

A system can usually be decomposed into two essential parts: a set of computation modules and a communication network connecting these modules. It is a heuristic that the highest performance of a system can be achieved if both computation modules and the communication network are running at their highest possible bandwidths and these two bandwidths are well matched with each other. If computational modules and the communication network of the system are timed separately, there is a better chance to achieve this goal. The feasibility of meeting

such a requirement depends not only on the timing scheme established but also on the architecture of a system.

There are many alternative timing strategies and their variants which can possibly be used in an architecture to control the data flow between modules and the computations inside modules at the correct time scale. However, most of them are not able to support the requirement for separately timed communications and computations. Four possible types of timing control schemes are examined in order to select an adequate scheme.

- Synchronously timed architectures. All of the data movement and data processing operations are synchronous in lock-step manner to global clock events. The clock frequency is pre-determined close to the highest possible margin so that high system performance can be achieved while correct system actions are still ensured,

- Clock period programmable synchronous architectures. This scheme is used in the polymorphic-torus architecture described in [90,89,80]. While still remaining as a synchronous architecture, the period of the central clock can be programmed so as to adapt to various configurations of the architecture,

- Self-timed control architectures. There will be neither global nor local clocks existing in such architectures consisting of combinational logic blocks communicating with each other through an asynchronous hand-shaking protocol. Therefore, the state of a self-timed system and actions to be done at a time point are completely determined by previous system states and signals generated from the hand-shaking control logic,

- Globally Asynchronous Locally Synchronous (GALS) architectures. "Synchronous" and "asynchronous" design approaches represent two extremes. It is possible to combine these two methods together. One interesting combination is to use clocks local to individual logic modules for syn-

chronous computation in each module, and an asynchronous hand-shaking protocol between logic modules for asynchronous communications in an interconnection network. Thus the controls on communications between a pair of connected modules are hand-shaking signals strictly local to the two communicating modules. System level global signals (if there are any) do not have any effects on this localised communication. The synchronous clock is also localised to the internal logic inside a module only, that is, the RCS between two locally synchronous logic modules does not affect the correct data transfer communications. However, a special mechanism is required to synchronise the events in an asynchronous hand-shaking protocol at the input of a synchronous module with the local clock in the module.

The synchronous timing scheme is often the first choice in the system design because of the low hardware complexity and logic design simplicity. This is true in the sense of conventional technology and architecture design principles which can be described as always designing systems of fixed physical topologies interconnected with, wherever and whenever possible, materials of highest conductivity. Therefore it is normally possible to design carefully a particular clock distribution network for a particular system so that the clock skews (particularly the RCS) across the entire system are minimised. The speed of the global control clock must be determined from the combinations of all the worst possible cases in the ACS, RCS, the longest communication path delay, and the slowest delay among all logic blocks. With the rapid increase in the complexity of integrated systems, the advantage of synchronous design simplicity is transferred to the overhead difficulties of designing the clock distribution network which is more closely related with the low level physical properties of integrated circuits. Because of some random characteristics from process technologies, these low level physical properties are much more difficult to manage and control than complex logic designs. To take the developing trend of VLSI as an example, while minimum feature sizes are decreasing (scaling down), the complexity (number of devices in a system) and

physical size (chip size) are increasing, and many previously negligible second-order physical effects cannot be ignored. This makes the clock distribution in a system even more difficult. The clock period programmable scheme merely makes it convenient to fit the global clock into a particular system configuration while it does not have any performance improvements on the worst case situation. Even worse, it has been shown that the measure of slowing clocks down sometimes still cannot compensate for clock skews [135]. The system fails in this case.

With the emergence of configurable architectures, the synchronous timing method suffers more problems. Expandability and flexibility are two of the characteristics of configurable systems. The expandability means that the size of a system can be varied to suit different application requirements, and the flexibility means the logical topology and the actual functionality of the system can be repeatedly configured for different applications. Therefore, the delay characteristic (computation delays and communication delays) in the system may be very different with different configurations and cannot be estimated in advance. The final system logical structure may be substantially different from the initial blank architecture in that previously dependent blocks may become independent or vice versa after the configuration, and the distribution of clock skews (RCS) will vary as well. It will be very difficult, if not impossible, to layout a fixed clock distribution network valid for all of the possible configurations of a system. It will also not be easy to determine an appropriate clock speed even if such a distribution exists.

In a system with a large number of logic modules, these synchronous timing difficulties are mostly at the system level which can affect the communication correctness and performance between connected modules. Once a clock is distributed into a logic module, its skews are unlikely to affect the correct functioning of the module. This discussion suggests that it is a good choice to apply the GALS timing scheme at the system level. As far as the data transfer is concerned, an asynchronous hand-shaking protocol is indifferent to varying delays in the com-

munication path between two modules [4,101,116,53]. Therefore, changing the configuration of a system with the GALS control will not have effects on the communication function of the network. Only the speed of communications will vary which is automatically adjusted by the protocol. The correctness of communications is assured by the protocol. As we shall see in the next section, the GALS timing scheme matches well to the architecture proposed, the requirement to time the communication network and logic modules separately, the DFG representation and the multiple threads computation model.

3.5.4 Communicating Synchronous Logic Modules

We proposed a general design framework in [40] for the construction of large scale modular systems by communicating synchronous logic modules. This framework is based on the GALS scheme and a GALS hand-shaking interface. A complex system is decomposed into a group of connected logic modules. These modules can be combinational or controlled by clocks local to each module. A GALS interface is attached to each module. These GALS interfaces are then connected to keep the dependencies in this group of modules.

A very important constraint in decomposing a system to impose a GALS modular structure is the level of complexity in each decomposed module. The *equipotential region* defined in [95, section 7.6] is a good constraint for the decomposition. An equipotential region is the size of an area in which all the signals are treated as identical at all the points on a wire in the region, that is, the delay associated with equalising the potential across one wire in the region is small and negligible in comparison with the device switching delays or signal transition times. The size of an equipotential region can always be estimated given a choice of processing technology [95]. There will be no difficulties in distributing a clock and implementing synchronous logic controlled by the clock local to the logic inside an equipotential

region. Hence the complexity of a decomposed module should be such that it can be fitted into an equipotential region.

The GALS design approach offers a solution to the problems in the design and operation of massively parallel processing systems and large scale configurable systems in the following areas.

- Easier modular design through the GALS interface. The design of a complex integrated system is made much easier by interfacing system modules through the GALS interface. The difficulties of distributing global clocks and system level synchronous timing design are eliminated by the GALS interface. Each individual module can be designed, reused and modified independently from the rest of the system at any time.

- Scalability. The size of a GALS system can be easily scaled up or down by adding or removing modules without concern over the global system timing. The system is also operationally scalable, i.e. if one module becomes slower or the delay in a communication path is increased, the system slows down but will not fail, and the system will run faster if some modules or communications become faster. This is a very important property for configurable systems.

- Easy Design Automation. A library of logic modules and the GALS interface blocks can be set up. The advantage of this library is that previously proven modules from both synchronous and asynchronous design can all be adopted. The task of the design automation system is to decompose, with certain constraints, a complete system into modules by referring to the module library, optimising the placement and routing connections of these modules, and connecting them through the GALS interface properly.

- Performance. The overall performance of a GALS system is also data dependent instead of depending on the worst case delay in synchronous sys-

tems. The GALS approach also provides an independent time control on the module functions and communications between modules. It is possible to optimise a system decomposition by matching the synchronous module operation bandwidth with the asynchronous communication bandwidth.

- Reduced Power. Since there are no signal transitions in an inactive module guarded by a GALS interface, the overall system power consumption can be reduced.

When the GALS scheme is used in our configurable architecture, it forms a GALS system template. It is a straightforward task to map a CTG and a DFG representation into such a template. The firing rule for a thread and a node in data-driven computations is checked and activated by the GALS interface. Parallel processing in different threads and nodes can be automatically scheduled and activated by the GALS scheme at run-time.

3.6 Algorithm Embeddings

Automatic algorithm mapping tools are required and are relatively simple to develop for configurable architectures [139,109,58,118,96,67,97,126,60] because the mapping only needs to convert an algorithm specification to a set of configuration data instead of detailed circuit level or layout level implementations. It is probably hard (very time consuming) to find an optimal mapping, but near optimal mappings can usually be found rather quickly. The mapping algorithms will also be architecture dependent; different configurable architectures would require their own mapping algorithms and mapping systems. The mapping process to embed an algorithm into our GALS configurable system involves generating a DFG from an algorithm specification, flattening the DFG, extracting a CTG, placements

of nodes and routing according to the CTG and the flattened DFG, and finally generating configuration bit-streams.

Since the main purpose of this thesis is to investigate the feasibilities of configurable architectures for algorithm embeddings and the design of such an architecture, the development of algorithm mapping theories and an automatic mapping system will be the task of a future project.

3.7 Summary

In this chapter, some fundamental issues for algorithmically structurable architectures have been discussed. A multiple threads computation model has been established for algorithms of irregular type. This computation model enables parallel processing of irregular algorithms to be scheduled automatically and carried out at run-time. It also facilitates node placement in an algorithm mapping process. An algorithmically structurable architecture template of connected hardware operators was illustrated. After detailed analysis of various timing control schemes for large scale configurable or modular systems, a GALS timing control scheme has been proposed. This GALS scheme effectively links the multiple threads computation, the principle of processing data on-the-fly, and algorithmically configurable architectures for convenient and efficient algorithm embeddings. The GALS scheme also exhibits its prospects to the design of future ULSI and WSI modular systems.

Chapter 4

A Configurable GALS Array

An algorithmically configurable architecture can generally be regarded as an ensembled architecture [121] in which a set of hardware operators are aggregated together and arranged with an initial physical topological relationship in a two dimensional plane. These hardware operators are logically connected to embed an algorithm.

Because an algorithm can be transformed into a primitive DFG representation, we propose a modular architecture with a pool of connected hardware operators H_{op}s for the embedding of irregular algorithms. These H_{op}s can be programmed to form the nodes in a primitive DFG of an algorithm. The architecture also has a configurable interconnection network to facilitate the mapping of tight and loose interconnections in a multiple threads computation graph. The GALS scheme will also be combined into this architecture. A top-down hierarchical overview of the architecture is presented in this chapter.

4.1 Basic Architecture Constraints

An H_{op} is a logic module with a few input and output ports on its boundaries for interconnection with other H_{op}s. A pool of such H_{op}s can be physically placed and connected in many different ways. A few basic architecture constraints are

considered to make it easy and efficient for VLSI implementation and suitable for irregular algorithm embeddings.

4.1.1 Architecture Regularity

A highly regular system structure is very suitable for the VLSI implementation. There are two possible approaches in the design of hardware operators for algorithmically configurable architectures. One approach, as used in [32,50,14], is to design a set of totally different hardware operators, each of which has fixed functionality, and connect these operators by a switched network. Two problems may be found in this approach; the limited availability of a particular type of hardware operator and the irregularity of modules due to the difference in the complexity of hardware operators. Another approach, which is much more common, is the design of a programmable hardware operator PH_{op} or processing element (PE). The PH_{op} can be programmed to perform a set of different functions as different H_{op}s. A system is then composed of a pool of the same PH_{op}. Architectures developed by this approach can support higher flexibility requirements. More importantly, a very regular structure can be obtained because only one logic module is used throughout the system. A PH_{op} will be designed and used repeatedly in our configurable system.

The design complexity of such a system can be reduced to the design of a communication network, a programmable PH_{op}, and some system I/O modules. The complexity of a PH_{op} will be much lower, and the design of a PH_{op} can still follow the conventional hierarchical approach by further decomposition. The design of the communication network will heavily affect the performance of the final system. This clear distinction between communications and computations makes it possible to investigate the two issues separately.

4.1.2 Architecture Scalability

Because the size of an actual sub-system, i.e. the number of PH_{op}s, that could be integrated on a single silicon chip is very limited, while a practical algorithm may require more PH_{op}s, it is important that an algorithm can be decomposed into several sub-algorithms of a smaller scale so that each can be embedded into one on-chip sub-system. To put this in another way, the boundary of a sub-system should be designed in a way making it easy to expand to a larger system of the same type for algorithms of larger scales.

The system scalability can thus be defined as the capability of a system whose size can be expanded to a larger scaled system of the same type by directly connecting a set of duplicated sub-systems or vice versa. The interfaces between the communication network and computation blocks are always on the boundaries of such scalable systems.

Regularity and scalability are two closely related issues. The placement of logic modules and the interconnection network determine the scalability of a system. A better scalability can be obtained with a regular structure. 2-D arrays and binary trees are two examples of a regular structure with a very good scalability.

4.1.3 Communication Overheads

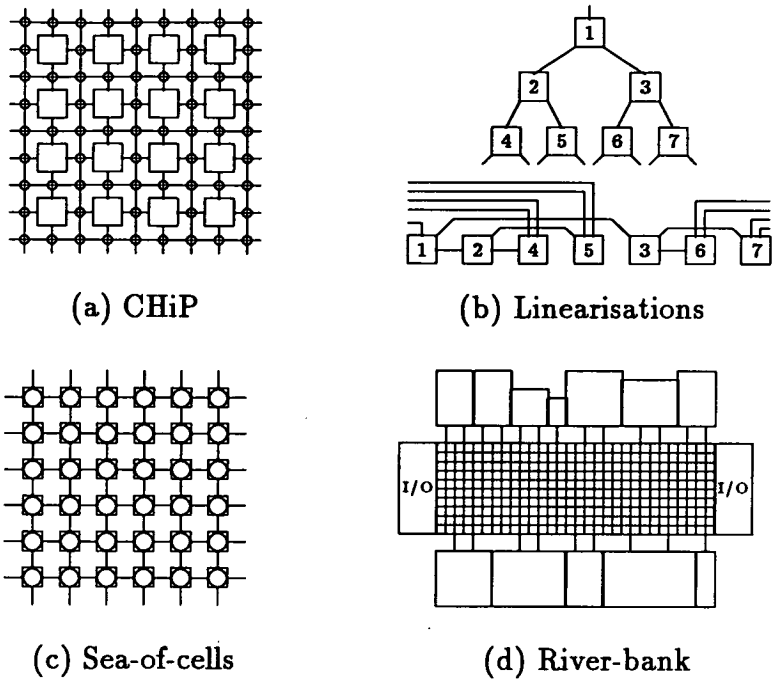
No matter what kind of approach is adopted, there are always communication difficulties in a configurable network because hardware operators and the communication network are physically pre-defined. That is, the total communication or routing resources have an upper bound RR_{cc}. If the overall communication requirement to embed an algorithm AR_{cc} is lower than or equal to the upper bound RR_{cc}, the communication network is said to be under utilised or fully utilised respectively. If $AR_{cc} > RR_{cc}$, the communication network is said to be congested

in which case the most common measure is to sacrifice some hardware operators to increase the RR_{cc}.

The AR_{cc} includes algorithm communication and embedding overheads. Algorithm communication requirements, which are represented by the edges in a primitive DFG are application dependent. Embedding overheads are caused by either the mapping of an algorithm with a higher dimensionality to a 2-D plane with lower routing capability or the irregularity in a DFG. The choice of the RR_{cc} for a configurable system is a compromise between silicon area occupied by the communication network, communication network utilisation, and hardware operator utilisation. Since many systems developed so far are for algorithms of regular data structures, the RR_{cc} of these systems is normally fixed and low with connectivity between C_2 to C_8. For irregular algorithms, architectures supporting flexible C and with a medium RR_{cc} are needed because the embedding overheads may be heavy due to the irregularity, albeit the communication requirement of these algorithms may not be very high.

4.2 System Level Physical Topology

The choice and design of a system level physical topology for a configurable system with a pool of PH_{op}s forms the essential framework in which application algorithms can be embedded. A selected physical topology will eventually determine the efficiency and performance of the embedded algorithm. The most important aspect of a physical topology is the design of a switched interconnection network and a placement scheme for the pool of PH_{op}s. Several typical system top-level topologies, which are depicted in figure 4–1, are considered and a Pseudo Nearest neighbour Configurable Array architecture (PNCA) is designed for the mapping of DFGs and computation threads.



(a) CHiP (b) Linearisations

(c) Sea-of-cells (d) River-bank

Figure 4–1: Typical switched interconnection schemes

4.2.1 Interstitial of a Switch Lattice and PE Array

The essential feature of this scheme, well illustrated by the CHiP architecture [125] as shown in figure 4–1(a), is an interconnected switch lattice and a 2-D array of PEs which are nested and intermingled with each other so that there is at least one switch isolating the connection between any two neighbouring PEs. Every PE is surrounded and isolated from each other by a set of switches. The configurability in this kind of architectures largely depends on the switch lattice, particularly the *corridor width* W_c which is the number of parallel routing paths available in one horizontal or vertical channel. Theoretical research points out that $W_c \propto \log N$ are necessary and sufficient to embed all planar interconnection patterns of N nodes with a reasonable PE utilisation [132]. A wider W_c is required to embed a complex pattern efficiently such as a shuffle-exchange graph. The W_c must be at least proportional to $N/\log N$ on an average [129]. This architecture is very regular because both PEs and switches are the same across the array. However, the

scalability of this architecture is spoiled by the surrounding switches and routing channels on the boundaries of the array.

As far as the communication overheads are concerned, it is difficult to select an appropriate W_c, especially as it is dependent on N for many algorithms, if this is not associated with a particular computation model. When different data structures are embedded into a system with a selected W_c, routing paths may be under utilised, near optimally utilised, or congested. The number of I/O points on the array boundaries can be calculated as

$$P_{I/O} = [W_{row} \times (m+1) + W_{col} \times (n+1) + m + n] \times w_P \tag{4.1}$$

where w_P is the number of bits in one path, m and n are the number of array element rows and columns, $m \times n = N$, W_{row} and W_{col} are row and column corridor width respectively. If $W_{row} = W_{col} = W_c$, $m = n$, equation 4.1 can be simplified to

$$P_{I/O} = [2W_c(n+1) + 2n] \times w_P \tag{4.2}$$

which is proportional to n. Therefore, when a W_c is selected, trade-offs must be made among routing channel area overheads, corridor utilisations, PE utilisations, and boundary I/O counts. It is noted that the practical implementation of the architecture based on this scheme had some modifications [48,88] because it will be very area inefficient if switches are scattered as sparsely as in CHiP.

4.2.2 Linearisation

Another distinct approach, known as the Diogenes scheme [114,115], is to linearise higher dimensional topologies to a physical linear array. All of the PEs in this scheme are placed as a linear array or a snake shaped linear array; a routing track consisting of segmented parallel paths is stacked above or below the linear array. A switch set is located at each PE location so that the I/O of the PE can be connected to a set of paths in the routing track. A specific PE dependency structure can be embedded into such a system by connecting PEs via paths and

switches in the track. Figure 4–1(b) shows a linearised example of a 7-node binary tree.

This is a very flexible scheme as long as there are sufficient paths in the routing track, almost any kind of topology can be embedded in such a scheme. The number of I/O points on the boundaries of this scheme tends to be much lower than those in 2-D physical arrays.

The problem of this scheme is that the average connection length increases with the increased array size if the dimensionality of the embedded logical topology is higher than linear. Folding of a long linear array and adding some vertical connections [115] can only alleviate the problem in a very limited way. The minimum number of paths in a track also depends on algorithm data dependencies and the size of the linearised array.

4.2.3 Overlapped Communications and Computations

This can be well illustrated by sea-of-cells approaches [87,59,58] as shown in figure 4–1(c). In this scheme, the communication network and computation lattice are logically overlapped with each other. The basic idea of this approach is that large number of identical fine-grained cells are connected in a 2-D plane with $C_n, n = 4, 6, 8$. The functionality of a cell can be set to perform only one of: logical/arithmetic computations, or routing operations at a time. Thus, if all of the cells in such a system are set to routing operations, by which each cell only passes incoming data to its prescribed neighbours, the entire system becomes a routing network. It may also be sufficient for some applications requiring only $C_n \leq 8$ connections for communications so that all of the cells can be set to various logical or arithmetic computations.

If C_{rout} is the hardware complexity required for routing operations in a cell and C_{cell} is the hardware complexity of the cell, in order to achieve reasonable cell utilisations and hardware efficiency, the granularity of cells, i.e. C_{cell}, must be

sufficiently small, and the ratio of C_{rout}/C_{cell} should be designed as a reasonable value so that there is no heavy routing hardware overhead when a cell is set as a routing node. Suppose $C_{rout}/C_{cell} < 0.5$, cell utilisation U_{cell} can be defined as

$$U_{cell} = 1 - \frac{N_{rout}}{N} \cdot \frac{C_{comp}}{C_{cell}} \tag{4.3}$$

$$N_{comp} + N_{rout} = N \tag{4.4}$$

where N_{comp} is the total number of cells used as computation nodes, N_{rout} is the total number of cells used as routing nodes. U_{cell} can range from 0% to 100%. When this scheme is used in [66] where the PE complexity is very high, C_{rout}/C_{cell} is very low, and C_{comp}/C_{cell} is high. Hence, U_{cell} of the array can be very low for irregular algorithms (high N_{rout}).

Because PEs are arranged as a 2-D array and the interconnections between PEs are direct wires without inserted switches, theoretically, the connectivity of PEs under this interconnection scheme can be anything between a linear connection, $\mathcal{C}_2$, through $\mathcal{C}_4$, $\mathcal{C}_6$, $\mathcal{C}_8$, to a complete connection $\mathcal{C}_{N-1}$ in a system of N PEs. The dimensionality $\mid \mathcal{D} \mid$ can be from 1 to N respectively. It is possible to increase routing resources with complex PEs by increasing $\mathcal{C}_n$. Practically, the interconnection cost in a planar space with very limited number of interconnection layers increases drastically with $\mathcal{C}_n$ when $\mathcal{C}_n > 8$. Therefore in planar silicon implementation, nearest neighbour interconnection schemes, $\mathcal{C}_n \in [2, 4]$, are very common choices because only one interconnection layer is needed for these nearest neighbour schemes. It is anticipated that this overlapped communication and computation scheme is not suitable for PH_{op}s with the GALS interface because it is relatively expensive to use such a module as a simple routing cell only.

4.2.4 Aggregated Switched Communication Network

In the above interconnection schemes, routing functions are distributed, together with PEs, over a complete system. They are suitable for array architectures composed of uniform PE. Another equally important and yet very flexible switched

interconnection scheme is to design a central communication network and connect functional modules around the boundaries of the network. This scheme is more preferred in the PCB (Printed Circuit Board) level implementation. Advantages are obvious: complete separation of the communication network and functional modules leading to the possibility of optimising both the switched network and functional modules, no restrictions on the module type and physical size, modules can be easily added/removed, no pre-defined physical topologies among attached modules. Considerable general research work has been done in the field of interconnection networks independent of any specific functional modules.

There are also attempts to implement this kind of interconnection scheme on chip or wafer level [50,14]. Advantages are higher flexibility and reduced number of I/O points. Figure 4–1(d) depicts the river bank architecture used in [14] which is similar to the floorplan generated from the FIRST [8] silicon complier. The grid block represents a cross-point switch network, blocks on the top and bottom of the network are functional modules connected to the terminals of the network. Communications between other systems and the external world are through the right and left edges of the network.

Some problems arise in the silicon implementation of this scheme. Although the central switching network can be designed totally independently of the functional modules, from a configurability viewpoint, it is expected that a set of hardware operators (H_{op}'s) of different functionalities can be readily integrated and connected to the network, via which the logical connection topology of these hardware operators can be set to the specific data dependency of an algorithm. The centrally designed network, which is very uniform, prefers the attached H_{op}s having the same width, while a VLSI floorplan prefers H_{op}s of the same height as in the standard cell approach. The consequence may be either low silicon efficiency or mismatch between H_{op}s' I/O ports and the network terminals, i.e the mismatch between the regular switch network and the irregular H_{op}s. The average connection length is longer too since data communication between two H_{op}s on the

upper and lower bank of the network have always to travel through a number of switches at least as same as the number of rows in the network. The scalability is also limited in that a system can only be extended in one dimension. One of the consequences is that worst case long connections may cross from one end to the other of the extended network. The routability of the network depends on the number of H_{op}s and the number of rows. If the network is only expanded in one dimension without increasing the number of rows, the routability of the expanded network will decrease drastically. Thus this scheme is not suitable for the silicon implementation of large configurable systems.

4.2.5 A Pseudo Nearest neighbour Configurable Array

As illustrated in previous sections, none of the four common system level physical topologies is a good choice for the DFG and computation threads mapping. Therefore, a Pseudo Nearest neighbour Configurable Array architecture (PNCA) is devised to make an efficient use of the precious two dimensional space with limited interconnection layers available under current integration technologies and to facilitate the mapping of the DFG and computation threads of an algorithm. The system top-level physical topology of a PNCA with 4×4 Routing Cells (RCs) and programmable H_{op}s is depicted in figure 4–2. The architecture is composed of a two dimensional regular array with identical PH_{op}s connected by a circuit switched configurable interconnection network.

The Physical Topology

It can be seen from figure 4–2 that a pool of PH_{op}s are arranged in a 2-D plane as a rectangular array. A PH_{op} can be programmed to perform a primitive logic or arithmetic function. PH_{op}s are not directly connected with each other. Instead, each of them is attached to one node in a configurable interconnection network. Each network node, which is located at the crossing point between a horizontal

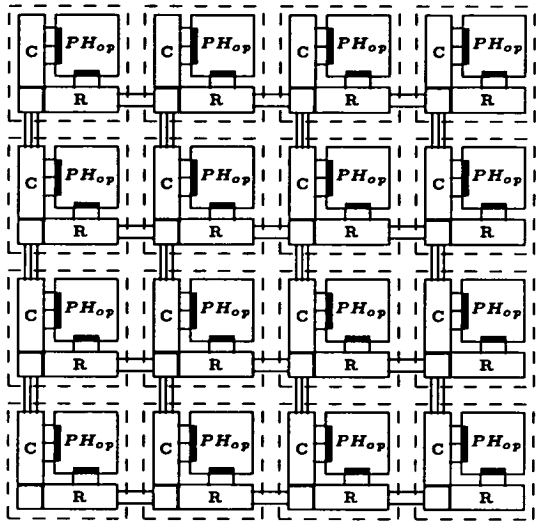


Figure 4–2: A Pseudo Nearest neighbour Configurable Array

and a vertical routing channel (H and V channel), is called a Routing Cell (RC). An RC is shown as an L shaped polygon in the figure. Every RC is physically connected to its four nearest neighbour RCs on its North, East, West and South (NEWS). An H and V channel can have multiple routing paths in parallel. Hence, an RC may have multiple ports on its NEWS boundary. The function of an RC is to select a path from either an H or V channel as an input or output port for the PH_{op} attached to it, and/or bypass a signal from one port to another. Several independent paths may exist at the same time in one RC so that multiple data can be routed through one RC simultaneously without interfering with each other. The routing capability of the network is mainly supported by the RCs.

An RC and its attached PH_{op} are regarded as one Multi-ported PE (MPE) which is shown as a dash box in figure 4–2. Each MPE is physically connected to its four nearest NEWS neighbours. But a PH_{op} can be logically connected to another PH_{op} beyond its NEWS neighbours through the RC settings. Therefore this architecture is called Pseudo Nearest neighbour Configurable Array (PNCA) for its physical nearest neighbour MPE connection and its configurability to support logical connections beyond NEWS neighbours.

The Network Configurability

The configurable network in the proposed PNCA architecture can be analysed independently from the PH_{op} array as a 2-D grid of a connected RC array. There are two factors that determine the configurability of this RC network: the number of independent routing paths, also called corridor width W, in an H and V channel W_h, W_v, and the number of switching states in an RC. The selection of W_h and W_v is a trade-off between routing capability and area efficiency. Note that the area of an RC will be increased in proportion to $W_h \times W_v$. An RC has a number of finite switching states either dependent or independent of W_h, W_v, which will be determined by the design of switch structure. As a constraint to keep the regularity of the array, W_h and W_v are not changed in any segments of the H and V channels and only one type of an RC is used. Provided that there are P_{RC} independent ports in an RC,

$$P_{RC} = 2 \times (W_h + W_v) + W_{H_{op}} = 4 \times (W_h + W_v) \tag{4.5}$$

where $W_{H_{op}}$ is the number of ports between one RC and its attached H_{op}, and here, $W_{H_{op}} = 2 \times (W_h + W_v)$. The upper bound on the number of switching states SS_{RC} of the RC is determined by the number of ports P_{RC} in the RC:

$$\begin{aligned} SS_{RC} &\leq (\sum_{i=0}^{n} C_n^i - n) + \frac{1}{2} \sum_{i=2}^{n-2} \sum_{j=2}^{n-i} C_n^i C_{n-i}^j \\ &= (\sum_{i=0}^{n} C_n^i - n) + \frac{1}{2} \sum_{i=2}^{n-2} C_n^i (2^{n-i} - 1 - n + i) \end{aligned} \tag{4.6}$$

where $n = P_{RC}$. According to equation 4.6, the upper bound of SS_{RC} will increase drastically with P_{RC}. It is worth noting that the complexity of an RC will also increase quickly if SS_{RC} is increasing towards the upper bound. An RC with the upper bound SS_{RC} has a complete port connection where each port is connected through switches to the rest of ports in the RC.

Because there are a finite number of RCs in a network, $N_{RC} = N_{H_{op}} = N_{MPE}$, in a PNCA, there will also be an upper bound on the number of interconnection

states S_{INT} realisable in a PNCA. S_{INT} is determined by the SS_{RC} and N_{RC},

$$S_{INT} = SS_{RC}^{N_{RC}} \tag{4.7}$$

From equation 4.7, it is clear that S_{INT} will increase exponentially with N_{RC} on the basis of SS_{RC}. Since the area efficiency of a high SS_{RC} RC will be very low, SS_{RC} should be selected much less than the upper bound while S_{INT} can still be maintained sufficiently high by increasing N_{RC}.

The highest possible requirements for W_h and W_v are that they are able to support a complete connection where every PH_{op} can be connected to all of the rest of the PH_{op}s in a PNCA. To connect a row with n elements completely, $\frac{n(n-1)}{2}$ segments of wires are needed. Several wire segments may be embedded in one segmented path in the H-channel of the row. Thus, the corresponding channel width W_h may be smaller than $\frac{n(n-1)}{2}$ by the sharing of some wire segments in one path. For instance, $n = 5$, $\frac{n(n-1)}{2} = 10$, $W_h = 6$ is sufficient. This can also be applied to the connection of a column of elements. However, because only orthogonal H and V channels are available, wider W_h and W_v are required to embed diagonal connections by jogging through the H and V channels if a 2-D array of elements is to be completely connected. Even if interconnect wires can share paths in channels, the complete connection is very expensive and seldom useful in practice. The W_h and W_v also depend on the number of elements in an array to accommodate a complete connection. It is not necessary to waste much area to support such expensive high connectivity in algorithmically configurable architectures. The selection of a compromise W_h and W_v will be discussed with DFG characteristics.

4.3 The GALS Scheme in PNCA

After the PNCA top-level physical topology is set up, a system level timing control scheme must be combined into the architecture to regulate data flows. As illustrated in section 3.5, the GALS system timing scheme is chosen for the PNCA architecture.

4.3.1 Synchronous Regions in PNCA

It is important to select an adequate logic complexity so that logic modules within this complexity can be fitted into an equipotential region with the local synchronous timing control. Although the equipotential region is technology dependent, it will make the system design and algorithm mapping much easier if a proper synchronous region for a configurable architecture is selected in advance. There are two possible choices of local synchronous regions in a PNCA. One is to include all the nodes on one thread as one synchronous region. The other is to confine a synchronous region to one PH_{op} only. The choice of a thread synchronous region is algorithm dependent. It can be reckoned that this choice will impose many difficulties in the algorithm embedding process because nodes, which are not in a synchronous region in one algorithm, may be in the same synchronous region in another algorithm. This uncertainty of synchronous regions will also impose more hardware requirements because of the switching over between the synchronous and asynchronous timing mechanism. Therefore, an algorithm independent GALS structure is desired in the PNCA architecture. The choice of one PH_{op} as a synchronous region will be algorithm independent. This synchronous PH_{op} selection also matches very well to the array hierarchical structure of the PNCA. Communications between the synchronous PH_{op}s are through an asynchronous hand-shaking protocol.

4.3.2 Communicating Synchronous PH_{op}s

Most of the asynchronous hand-shaking designs are developed for systems consisting of irregular blocks connected by fixed interconnections. A widely used three element structure is to insert an asynchronous hand shaking element, such as a Muller C-element [100] or its modified form, in-between every pair of communicating modules. For architectures with configurable interconnections and requirements for scalability, this approach appears less attractive. This is because in a configurable system, the dependency between a pair of modules may vary with the embedded algorithm, i.e. two dependent modules for an algorithm may become independent for another algorithm. Therefore, it is difficult with this three element structure to distribute and connect the hand-shaking blocks among logic modules in a configurable system for different algorithms. If this three element structure is used in an array, it will also destroy the scalability of the array on the array boundaries.

Asynchronous Guarded Communications

To preserve the array uniformity and scalability, a two element asynchronous communicating structure is desirable in configurable systems. Instead of inserting a separate hand-shaking element, the asynchronous hand-shaking is split into an input and an output guard logic, which are added before inputs and after outputs in a module, so that the communication between any pair of modules are controlled by the input and output guard in the two modules. A two element structure is obtained with this guarded communication scheme. Figure 4–3 depicts this two element communication structure with guards represented by black strips.

This guarded communication protocol can be clearly illustrated by the bundled data interface [128], as shown in figure 4–3, with two communicating modules. A bundled data interface has an arbitrary number of data bits on a D line accompanied by two communication control signalling lines called Request (R) and

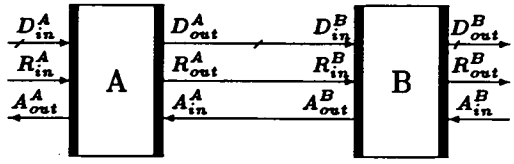


Figure 4–3: Asynchronous guarded communications

Acknowledge (A). The functionality of an Input Guard (IG) is to detect the R^B_{in} line to see if there is a new datum presented on the input D line, to decide whether to accept the new datum according to the state of the guarded module (B), and to acknowledge, through the A^B_{out} line, the sender module (A) after the new datum is stable in the B module. The functionality of an Output Guard (OG) is to send a request (R^A_{out}) to a receiver module (B) when a new datum is generated at the output, and to reset the output to null, i.e. free the guarded module (A) to the next operation when an acknowledgement (A^A_{in}) is detected from the receiver module (B). Thus, the input and output guard logic, G_{in} and G_{out}, are:

$$G^B_{in} = \begin{cases} (R^B_{in} \wedge \overline{S_B} = 1) \rightarrow (D^B_{in} \Rightarrow (S_B = 1)) \rightarrow (A^B_{out} = 1) \rightarrow (R^A_{out} = 0) \\ (R^B_{in} \wedge \overline{S_B} = 0) \rightarrow G^B_{in} \end{cases} \tag{4.8}$$

$$G^A_{out} = \begin{cases} (\overline{A^A_{in}} \wedge S_A = 1) \Rightarrow D^A_{out} \rightarrow (R^A_{out} = 1) \\ (A^A_{in} \wedge S_A = 1) \rightarrow (R^A_{out} = 0) \rightarrow \begin{pmatrix} D^A_{out} = NULL \\ S_A = 0 \\ A^B_{out} = 0 \end{pmatrix} \\ (A^A_{in} \wedge S_A = 0) \rightarrow G^A_{out} \end{cases} \tag{4.9}$$

where R_{in}, R_{out}, A_{in}, A_{out}, D_{in}, and D_{out} are data sending request (new valid input data), data receiving acknowledgement, and data at the input and output of a module respectively; S is the state of a module. $S = 1$ means an engaged module where inputs are prohibited.

There is a basic constraint with this bundled data interface. Data (D^A_{out}) can only change when the A module is acknowledged (A^A_{in}), and D^B_{in} must become

stable before the B module detects a new request (R^B_{in}). That is, an R^A_{out} signal must be sent out after D^A_{out} is stable and the delay on the D line must not exceed that on the R line. Under this constraint, the communication between the two modules is insensitive to delays on the interconnect paths by which is meant that the change of delays on the interconnect paths will not affect the correctness of the data transfer, only the speed of transfer will vary.

Special Considerations

There are several special issues worth noting when this guarded GALS control scheme is applied and special care must be taken when designing the guard logic.

1. Different communication modes. Three general communication modes are identified. One-to-one communication is exactly shown in figure 4–3. The generation of communication guarding signals in this case is straightforward. One-to-many broadcast is the mode in which one common source module is feeding data to more than one destination module. The acknowledge signal back to the source module is the logical AND of all the acknowledges from the destination modules. Many-to-one assembly is a communication pattern when several modules are supplying operands to one module simultaneously. In this mode, the Input Guard of the destination module must ensure that all of the input operands are taken before prohibiting its inputs and generating an acknowledge back to all the source modules.

2. Deadlock avoidance. A typical problem in asynchronous hand-shaking is deadlock. Deadlock is a phenomenon when operation processes are halted in an endless waiting state. This is a case often arising in certain loop configurations where some inputs are waiting for results from some outputs while the generation of these outputs depends on these inputs. To take the two communicating modules in figure 4–3 as an example, a deadlock loop is formed if R^B_{out} is connected to R^A_{in}, A^A_{out} is connected to A^B_{in}, and D^B_{out} is connected to D^A_{in}. Special measures must be taken to avoid deadlock [71].

For example, one or several conditional fork structures can be added to break possible deadlocks in a loop configuration.

3. Synchronisation. A modular system timed with the GALS scheme requires a synchronisation mechanism to coordinate the asynchronous input data with the local clock in a synchronous logic module. One of the most common ways is to use a synchroniser to synchronise the asynchronous input data to the event of the local clock in a module. An ideal synchroniser should be completely reliable. However, because synchronisers are usually implemented with bistable structures, there is a probability of synchronisation failure in which the output of a synchroniser stays in a metastable state for an indefinite length of time instead of settling to one of the two stable states [63]. If this metastable state persists for too long, incorrect state may be resolved from the synchroniser. This is called a synchronisation failure. The synchronisation failure is caused by the physical nature of the continuous transition between bistable states; it is inevitable. But, various techniques can be used in synchroniser design so as to reduce the probability of synchronisation failure to an acceptable low level. This problem is discussed in more detail with the design of a synchroniser in the next chapter.

4.3.3 A Configurable GALS Array

A configurable GALS Array (GALSA) system is constructed by placing an input and output guard with a synchroniser between an RC and its attached synchronous PH_{op} in each MPE in the PNCA. This structure enables the system level asynchronous communication between locally synchronous PH_{op}s. But this will not change the physical boundary of an MPE nor the RC interconnection network. Therefore the top-level system physical topology of the PNCA architecture is well preserved in the GALSA system.

The GALSA system has all the properties discussed so far for a configurable system to embed irregular algorithms:

1. Regularity and scalability. The GALSA system is highly regular with a 2-D array of MPEs. If the ports are properly placed on the boundaries of an MPE layout, a GALSA can be easily obtained by abutting the MPE in the horizontal and vertical direction. The system can be easily enlarged by directly connecting small sub-systems together. There are no system timing difficulties with different network configurations and it is operationally scalable because of the GALS timing scheme.

2. Computation threads. The PNCA architecture is designed with the multiple threads computation model in mind. Therefore, threads can be easily formed in a GALSA. A thread is formed with a row or a column, or part of a row or column of RCs. The PH_{op}s in a thread are physically placed one by one and connected in shortest paths corresponding to the tightly coupled nodes in a thread of a CTG. The communication between threads may be routed through unused paths in the H and V channels as loose connections.

3. Primitive DFG mappings. Because a PH_{op} can be programmed to one of the logical/arithmetic functions defined in a primitive DFG, the programming of the PH_{op}s in a GALSA system is straightforward to embed the nodes in a primitive DFG.

4. Data Flow Computations. A GALSA system is a data flow computing engine by its GALS timing nature. A PH_{op} is ready to "fire" when its outputs are cleared and matching data are presented at all of its inputs. This also applies to the firing of a computation thread, which enables the automatic detection and scheduling for parallel processing among threads and PH_{op} nodes.

5. Minimum memory accesses to a host system. Once a GALSA system is configured, the memory accesses of the GALSA system to its host computer are minimised to getting the initial input data from the memory and saving the final results to the memory.

The design of the GALSA system is decomposed to the design of an RC, a PH_{op} with a GALS interface, the configuration data stream loading circuitry, and an array I/O interface. Because the system is timed with the GALS scheme, the design of these components is independent from each other from the time point of view. The complexity of each decomposed component is relatively low and thus well manageable in the full-custom design method.

4.4 RC and PH_{op}

The design of an RC and PH_{op} for a GALSA system is closely related to some DFG properties from the computation viewpoint. It is necessary to analyse these DFG properties in order to establish proper logical structures for the RC and PH_{op}.

4.4.1 DFG Computation Properties

A general classification on types of nodes in DFGs can give a clear indication on the RC and PH_{op} design. Three general types of nodes are classified in a DFG by examining the DFG example in figure 3–3.

1. Arithmetic nodes (A-nodes).

   Arithmetic functions can be evaluated in arithmetic nodes, for example, the addition/subtraction, multiplication, and the square rooting nodes used in figure 3–3. The hardware complexity corresponding to these kinds of nodes is the highest, and computation delays through them are inevitably long because of the multi-step operations required in many complex arithmetic functions. It is always possible to decompose a complex arithmetic function to a set of simple arithmetic or even logical functions. Hence, a set of primitive arithmetic nodes can be defined so that arithmetic functions of higher complexity can be decomposed to this set of primitive arithmetic nodes, and evaluated through this set of primitive nodes connected in a specific way.

Primitive arithmetic nodes are hardware counterparts of some unary and binary arithmetic operations; the number of required input operands ≤ 3, and the number of outputs ≤ 2. Thus, the number of I/O ports of these primitive arithmetic nodes P_A will be $1 \leq P_A \leq 5$.

2. Boolean nodes (B-nodes).

   These are similar to arithmetic nodes, but the operations to be carried out in Boolean nodes are all Boolean type functions such as AND, OR, XOR, bitwise operations, comparisons. The $\boxed{\geq 0?}$ node in figure 3–3 is a Boolean node. A set of primitive Boolean nodes can also be defined so that complex Boolean functions can be decomposed and evaluated through this set of primitive Boolean nodes. Primitive Boolean nodes implement a set of Boolean functions of single opcode with one or two input operands. The output of a Boolean node is a logic value true or false. The number of I/O ports of primitive Boolean nodes P_B will be $1 \leq P_B \leq 3$.

3. Data flow control nodes (C-nodes).

   This is a set of nodes in which input data are diverted or selected in a way determined by the value of Boolean control input variables. At least one Boolean control input is presented to this type of node. Typical data flow control nodes are merger, gate, router, and self-iterator.

   - Merger. The function of a merger node is to select, according to the pattern of control inputs, one data item to output from a set of data inputs. In general, a merge node can have 2^n data inputs, one output, and n bits of Boolean input, $P_M = 2^n + n + 1$. A primitive merger is defined as $n = 1$, $P_M = 4$, that is, one of the two inputs to a primitive merger is selected by the value of a Boolean input to the output. In figure 3–3, the last four nodes, which generate *root1* and *root2*, are primitive mergers. A merger having arbitrary input m can be constructed from $(m-1)$ primitive mergers with a Boolean input of

c bits and a $c \longrightarrow (m-1)$ bits Boolean decoder node to decode the c control bits, where

$$c = \begin{cases} \log_2(m-1) & if\ \log_2(m-1) = \lceil \log_2(m-1) \rceil \\ \lceil \log_2(m-1) + 1 \rceil & otherwise \end{cases} \quad (4.10)$$

- Gate. A gate node has one data input, one data output and one pattern input, $P_G = 3$. There is a special pattern attached to a gate node, for instance the nodes with a single Boolean true (T) or false (F) pattern in figure 3–3. Data presented at the input of a gate node will be passed to its output if and only if the input pattern matches the pattern attached to the node.

- Router. A Routing node is the inverse of a merger node. An input data to a router is passed to only one of a set of its outputs, as determined by the pattern of the Boolean control input. A general router has one data input, 2^n outputs, and n bits of Boolean input, $P_R = 2^n + n + 1$. A primitive router is defined as $n = 1$, $P_R = 4$, so that one Boolean variable input can select one of the two outputs, to which the single input datum is to be routed. Similarly, a router with arbitrary m outputs can be built from $(m-1)$ primitive routers and a $c \longrightarrow (m-1)$ bits Boolean decoder node to decode the c control bits. c here also follows equation 4.10.

- Iterator. The output of an iterator, which is controlled by certain conditions, can be either fed back to its own input or directed to some other node. The input to an iterator must be selected either from the output from another node or the feedback from its own output. Therefore, an iterator can actually be composed from a primitive merger at its input, a primitive router at its output, and an operational node in between. One of the outputs from the router is connected to one of the inputs of the merger. Conversion nodes may also be needed to convert other iterating conditions to Boolean control conditions.

Edges in DFGs are all directed, to represent data flow direction. Frequently encountered edge types are one-to-one and one-to-many edges. One-to-one edges do not impose any implementation or embedding difficulties. Many-to-one edges are prohibited in direct implementations because of the uncertainty caused by wired-or connections. Besides the special care required in handling the asynchronous hand-shaking of the one-to-many type, as described in section 4.3.2, there are also many physical and technical factors to be considered, such as excessive capacitive loads, non-ideal switching behavior, signal propagation degradations/delays when an edge of a DFG is mapped into a GALSA system. If one data source is to be shared by too many destinations, exceeding the fan-out capability of the driving node, buffer stages must be added.

4.4.2 The Routing Cell

The configurability of a GALSA system will be mainly determined by the design of the Routing Cell. More importantly, the performance of the RC will heavily influence the performance of an overall system. It is a compromise between routing capability and area efficiency to select a number of ports P_{RC} and a number of states SS_{RC} for an RC. P_{RC} is determined by W_h and W_v by equation 4.5. Therefore the selection of a P_{RC} is the problem of determining a W_h and W_v.

According to equation 4.6, the upper bound of SS_{RC} increases quickly with P_{RC}. So if a higher routing capability is required, P_{RC} must be increased. However, a large P_{RC} will be very area inefficient. The final system configurability depends not only on RC routing states but also on the size of an application problem. It is unrealistic and not necessary to select a very large W_h and W_v to achieve a very high configurability at the expense of area. On the other hand, interconnection resources can be significantly saved when the computation threads model is applied because only the shortest local connections are needed in each thread. A modest W_h and W_v, that can support configurability higher than nearest neighbour

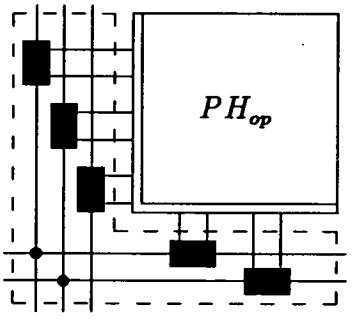


Figure 4–4: The channel width and RC ports

interconnection, much lower than a complete interconnection, will be sufficient in most cases. A system composed of an array of primitive type PH_{op}s is a good compromise for PH_{op} granularity and area efficiency. As analysed in section 4.4.1, the number of I/O ports of all primitive DFG nodes is in the range $[1, 5]$. This indicates $W_h + Wv = 5$ as a reasonable choice without imposing too much area requirement for the H and V channels, and a primitive PH_{op} can communicate its 5 I/O data with 5 paths in the H and V channels. We set $W_v = 3$ and $W_h = 2$ to give the V channel more routing capability, so that threads extracted from an algorithm will be mainly mapped into the vertical direction in a GALSA array. $W_h = 2$ allocates a spare routing path in the H channel because one path may often be used for neighbour connections. Because data can enter and leave a PH_{op} in four NEWS directions, there are $2 \times (W_v + W_h) = 10$ channel I/O ports in an RC to interface its neighbouring 4 RCs. The inputs and outputs in a PH_{op} are tapped from these 10 channel I/O ports. This assignment of channel width, RC ports and an RC/PH_{op} interface is illustrated in figure 4–4. The black boxes in the figure are switch units which can divide a channel path into two segments and tap both path segments for the PH_{op}.

So we have $W_{PH_{op}} = 10$ and $P_{RC} = 20$ for this RC design from equation 4.5. If a higher configurability is required, some PH_{op}s may be left unused to get more RCs, and H and V channel segments. Hence, routing congestions unsolvable with the 100% PH_{op} utilization constraint may still be solved at the cost of a decreased PH_{op} utilization.

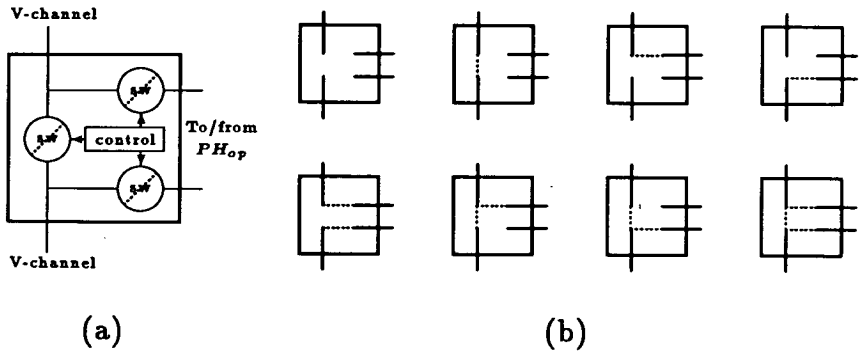


Figure 4–5: (a) A switch unit in an RC, and (b) Switching states of (a)

The upper bound of routing states for the RC design can be obtained from equation 4.6: $SS_{RC} \leq \sum_{i=0}^{20} C_{20}^{i} - 20 + \frac{1}{2}\sum_{i=2}^{18} C_{20}^{i}(2^{20-i} + i - 21)$. To implement all the routing states in this upper bond, the RC switching mechanism will be very complicated and expensive because of the high P_{RC}. Therefore, a simple switch unit, as shown in figure 4–5(a), has been designed.

The switch unit is a four terminal device consisting of three switch elements and a switch control. One switch element divides a channel path into two segments and the other two switch elements tap both segments of the path for a PH_{op}. The orientation of the switch unit in the figure is for the V-channels. Switch units used in the H-channels are obtained by rotating the unit anti-clockwise by 90°. Compared with the switch structures proposed in [15,88], this switch unit is much simpler while the routing capability is still sufficiently flexible because the segmentation of channel paths can improve routing capability efficiently. The tapping of the segmented paths makes it much easier to form computation threads and communication paths between threads. The interaction between a V channel and an H channel may only happen at their cross points. The two dots in figure 4–4 are two switches linking two paths in the H channel to two paths in the V channel respectively. It is not necessary to have cross-point links (6 switches) because the 5 switch units in an RC can tap any of the path segments for the PH_{op}.

Since there are three switch elements in one switch unit, all of the possible routing patterns in one switch unit can be found by various combinations of ON

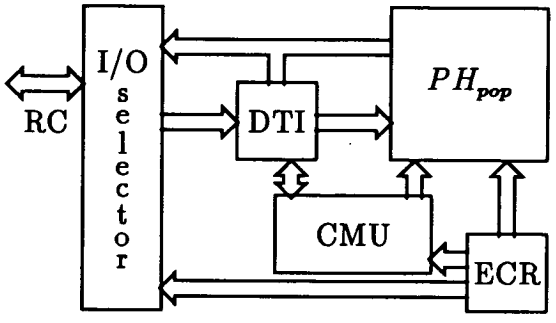


Figure 4–6: PH_{op} block diagram

and OFF state of the three switch elements. Thus, the complete number of routing states of a switch unit can be calculated as: $SS_{su} = C_3^0 + C_3^1 + C_3^2 + C_3^3 = 8$. These eight possible routing states are depicted in figure 4–5(b). From a performance viewpoint, it is best to make use of the states involving only one or at most two conducting switch elements. The last state in which three switch elements are all closed is not allowed because a wired-or condition may occur.

There are four states for the H and V cross section links, so the actual RC routing states can be obtained as:

$$SS_{RC} = 4 \times SS_{su}^{N_{su}} \tag{4.11}$$

where N_{su} is the number of switch units in an RC. For $SS_{su} = 8$ and $N_{su} = 5$, we get $SS_{RC} = 131072$ for the RC in figure 4–4. This RC should be sufficiently rich in its routing capability for most arithmetic applications.

4.4.3 The Programmable H_{op}

The top-level block diagram of a PH_{op} is shown in figure 4–6. In the figure, the I/O selector selects input and output ports from an RC; the Data Transfer Interface (DTI) and the Clock Management Unit (CMU) form a GALS interface; the Execution Code Register (ECR) stores binary codes to control the I/O selector, CMU, and the function of a primitive PH_{op} (PH_{pop}).

Conventional microprocessors usually have a very complicated control part for instruction decoding, scheduling, sequencing and various event handling, and a data path for processing data. The structure of PEs in MIMD parallel processing systems is similar to conventional microprocessors, in particular, a complex control part in each PE is required because every PE may behave independently on different or even the same instructions decoded differently. The PEs in SIMD systems are usually much simpler in that there is no complex control part because all of the PEs are sharing a common central control part which sequences and broadcasts instructions executable in the data paths of all PEs. However, a simple control mechanism local to each PE can often be found very useful in SIMD architectures, to facilitate some local low level modifications to globally broadcasted instructions. Examples are the disable/enable mechanism and data memory address offsets. Dynamic programmability is another interesting approach in which interconnections and PE operations may be dynamically changed according to some conditions and data status generated during processing. Dynamic programmability will certainly require higher system complexity to implement. The dynamic control principle is very useful for applications with highly dynamic data dependencies, such as the region growing and the labelling problems in image processing, or dynamic particle movement in a particle system. Because the GALSA system is expected to be applied to arithmetic and logic evaluation applications, static programmability should be sufficient. As opposed to broadcasting instructions during data processing as in SIMD systems, different executable instructions for statically programmable PEs are all loaded into each PE prior to the execution phase. In our case, it is not necessary to design a complex instruction handling control part for a PH_{op}; the PH_{op} is programmed by loading codes into its local ECR at the same time as the array interconnection network is configured.

A set of proper primitive functions for the design of the PH_{pop} need to be selected, so that the circuit complexity of a designed PH_{pop} is not too high for the range of selected functions. Any other complex functions can be implemented by

node type	function	symbol	H_{op}	operands	outputs
Arithmetic	add/sub	+/-	full adder	A, B, C_{in}	S, C_{out}
Boolean	NOT/AND/OR	$\neg \wedge \vee$	Boolean functions	A, B	C
	XOR/XNOR	$\oplus$ $\odot$			
Merger/Gate	selection	$\triangledown$	2-1 multiplexer	L, R, C	O

Table 4–1: Selected primitive functions for PH_{pop}

connecting a group of PH_{pop}s in an appropriate way. According to the DFG node properties analysed in section 4.4.1, a PH_{pop} should be able to perform at least some simple A-node, B-node, merger and gate functions. All the other types of nodes or the same type of node with higher complexity can be built from these primitive nodes. For example, a 1-to-2 router can be formed with 2 gate nodes having the same input; an iterator node can be constructed with a merger and 2 gate nodes. The simplest arithmetic function is addition and subtraction, which will be supported by the PH_{pop}. NOT/AND/OR/XOR/XNOR are chosen as the available Boolean functions because these functions can be derived directly from an addition function which will be illustrated in section 5.4.3. Other Boolean functions can be obtained from these 5 primitive functions. The merger and gate can be implemented with a 2-to-1 multiplexer. These selected primitive functions are listed in table 4–1.

4.5 PH_{op} Local Memory

Since most of the massively parallel processing systems are designed for image processing or matrix related applications, a common approach is to load a complete set of data, such as an array of image pixels, into a processing array, and to save the intermediate and final results in the array. If there are enough PEs

in the array, each PE will hold only one data item, otherwise each PE needs to hold a subset of data items. Thus the minimum local memory requirement for a PE will be at least the word length of one data item. In actual system implementations, it is normally desirable to allocate as much local memory as possible so as to reduce local memory reading and writing traffic to a host computer during data processing. However, the choice of the local memory size is limited by the available chip size and technologies. One of the common solutions to this is to design an external RAM port in each PE so that the local memory can be easily expanded with the off-the-shelf RAM chips to meet application requirements. Besides local memories, a set of registers are often required to facilitate data manipulation operations. Because we adopt the data flow processing principle, data are processed on-the-fly. That is, every time a finite set of input data flows through a set of connected programmed PH_{pop}'s, they are modified through a sequence of intermediate data, and a final set of required results can be obtained with a finite number of processing steps. One important characteristic of this processing data on-the-fly approach is that once all of the subsequent modified data from the previous step are generated, there is no need to preserve their ancestor data. In [70,30], memory requirements for systolic arrays which are based on the processing on-the-fly computation principle are analysed. In general, the use of input/output and some temporary registers will be suitable for systems based on the processing on-the-fly computation principle. Thus, there is no requirement for high local memory for PH_{op}s in a GALSA system as in a conventional massively parallel computing system. As can be seen from table 4–1, there should be at least three input registers to hold input variables and two temporary registers for output results in a PH_{pop}.

4.6 Summary

In this chapter, some basic architecture constraints are discussed and several typical configurable interconnection topologies are analysed and compared. Based on this analysis and the data-flow computation principle, an algorithmically configurable array architecture called Pseudo Nearest neighbour Configurable Array (PNCA) is proposed for the multiple threads computation model. A top-level GALSA system is illustrated by imposing a guarded GALS timing scheme on top of this PNCA architecture. The dependency between two connected PH_{op}s is controlled through a guarded asynchronous hand-shaking protocol and each PH_{op} runs synchronously with a local clock. A VLSI implementation of this GALSA system will be described in the following two chapters.

Chapter 5

An Implementation of a GALSA

From the top-level GALSA system structure described in chapter 4, we are ready to implement the building blocks while moving down the system hierarchy. There are four major parts to be designed: a PH_{op}, an RC, a configuration data stream loading control and an I/O interface for the array boundaries. Because the communication between PH_{op}s are asynchronous, the design of each part is independent from the timing point of view, i.e. we focus on the design of functions for each part. There is no need for special timing considerations between any two parts.

5.1 Design Tools and Implementation Technology

During the progress of this project, there were several choices in the use of CAD systems. Available systems include the MAGIC interactive layout system with DRC check and netlist extraction capability, to interface with various simulators [17,104,22], and the ES2 Solo gate array compiler [28]. Later on, the Cadence Edge and latest Opus design system became available.

The ES2 Solo software is basically a gate array and macro cell silicon compilation system. A design is taken from either a schematic or netlist entry, through

simulation, automatic place and route for gate array style layout, post-layout simulation, package choice and design validation to obtain a chip design for fabrication.

Since the architecture of the GALSA system is highly regular, it can be implemented by the duplication of the RCs and PH_{op}s in horizontal and vertical directions as a two dimensional rectangular array. The area efficiency of an entire GALSA system will be largely determined by the design of the RC and PH_{op}. Hence, the ES2 Solo was not used. The logic of the RC and PH_{op} needed to be fully custom-designed and their layouts needed to be implemented with an interactive layout system. The Berkeley MAGIC CAD system has a set of assisting programs which can extract layout data and convert them into several netlist formats which can be used as inputs to some simulators. This makes the post-layout simulation possible. The Cadence Opus is a much more powerful system, which can perform many more functions, including most of the tasks that the ES2 Solo and MAGIC can do. Therefore, we migrated to use the Cadence Opus system to design our GALSA system. All logic functions are custom-built with MOS transistors so that the transistor sizes can be properly adjusted.

There are several simulators available, such as Crystal and ESIM from the UCB CAD package [17], RNL [104], and Spice [17,104]. Crystal is an interactive VLSI circuit timing analyser which can estimate the speed of a circuit and print out information about the critical paths. ESIM is an interactive event-driven switch level simulator. Both Crystal and ESIM take `.sim` format files extracted from layouts by MAGIC. Both use very simple models and are not appropriate for complex circuits. RNL is another switch level timing and logic simulator with a LISP based interface. Although the circuit model used in RNL is simple, it can be fine tuned at the user level so as to suit the different requirements of different types of circuit. RNL can take the circuit netlist extracted from the layouts designed with MAGIC and can be run either interactively or in batch mode. Spice is a circuit level simulator which has much more elaborate models for various devices and thus can get more accurate simulation results at the expense

of more CPU time than other higher level simulators. Spice is often used in simulating composition blocks or key components of a large VLSI system. At the outset, Spice 2G6 was used, but this is an old version which is slow and has very poor convergence behaviour. During the course of this project, Spice 3C1 was obtained and installed; this version has much better performance with improved numerical algorithms and an improved graphics interface for waveform outputs. Spice 3C1 can still take input data extracted by the MAGIC system. When we migrated to the Cadence Opus system, the Hspice circuit simulator, which is based on Spice, became available. The Hspice simulator has even better numerical convergence performance and a very impressive graphics user interface. Thus, the current design of the GALSA system is simulated with the Hspice simulator.

The fabrication technology and libraries, which are also integrated into the Opus system, are from Mietec 2μm N-well double poly-silicon, double metal CMOS process. Electrical parameters (NMOS and PMOS models) for Hspice simulation are provided by Mietec.

5.2 The Configuration Technique

There are many different configuration techniques to interconnect PH_{op}s in accordance with the data dependency of an algorithm. Each of these techniques has different system reusability, implementation requirement, switching performance, and silicon area requirement. The interconnection network in a GALSA system can be implemented with either non-volatile hard restructuring or soft configuring technique.

Non-volatile hard restructuring techniques are usually based on the physical blowing of a wire or connection between two points by special techniques such as laser, electron-beam or applying a programming voltage. Therefore a restructuring is non-volatile and permanent. A hard restructuring is often performed after wafer

fabrication to improve yields, for instance for RAMs, by disconnecting faulty cells and bringing in redundant fault free cells [111,94]. Some of these techniques are also used in Field Programmable Gate Arrays (FPGA) for final customisation. In [39,25], "anti-fuses", which can be irreversibly changed from high to low resistance when "blown" by applying a programming voltage across them, are used in an electrically configurable gate array design. The ON or OFF characteristics of a connection resulted from hard restructuring methods are the best, and they also take up the smallest area of the various configuration methods. However, hard restructurability can only be used once, i.e. it is impossible to "undo" a change. The implementation of a hard restructurable system normally requires a special and expensive processing technique.

As opposed to irreversible physical changes in hard restructurings, various electronic switches can be designed for soft configuration purposes. A soft configuration is defined as a configuration which can be done repeatedly without permanent physical changes to a configurable system, i.e. configuration changes are reversible. There are two common ways to control the ON/OFF status of a soft switch. One is the non-volatile switch control, in which switch settings can be retained even when the system power is switched off such as the EPROM or EEPROM techniques used in [56,105,54]. The other is to use static bistable elements, for instance using static RAM (SRAM), to control the ON/OFF settings of switches. The SRAM control is volatile because once the system power is off, all switch settings will disappear. There are many configurable systems implemented with the SRAM switch control technique [139,109,59,58,9,14,81,90,134].

Soft configurable systems provide much higher configuration flexibility at the user level, i.e. configurations are achieved by changing electronic control signals to switches, and can be done repeatedly at any time. An extra benefit of using SRAM controlled switches is that conventional common VLSI fabrication technologies can be used for silicon implementation, and it is fairly simple to configure such a system which is just the same as writing a normal RAM array. Therefore, a

soft configurable system with SRAM controlled switches has the best user configurability and reusability. It can be implemented with less expensive technologies while EPROM or EEPROM processing technologies are more complicated than a SRAM process. The flexibility of soft configuration techniques is obtained at the expense of more area consumed by switching logic than hard restructuring methods, and some extra delays introduced into communication paths by switching logic. The cost of larger area requirement is compensated by less expensive processing technologies. A high system throughput (bandwidth) can still be obtained because communications can be pipelined [88] to overcome latencies caused by the soft switching devices. The performance of switching devices is also being improved continuously.

Our GALSA system was to be designed with high user configurability, and to be implemented with a conventional CMOS processing technology and design tools. It is not suitable to use hard restructuring techniques because of the non-reusability after a restructuring; the restructuring process is slow and the implementation requires special and expensive processing technologies. We do not choose the EPROM or EEPROM approach because some special implementation techniques and programming process are also required. Therefore, the soft configuration approach with SRAM controlled switching logic was chosen in the design of the GALSA system.

As illustrated in chapter 4, the design of the GALSA system can be divided to the task of designing a PH_{op}, an RC, a configuration data stream loading circuit and an array boundary I/O interface. Although an asynchronous data transfer interface is included in the PH_{op} depicted in figure 4–6, we shall first describe the design of such an interface for its special and important role in our GALS approach to the design of configurable hardware algorithms.

5.3 Asynchronous Data Transfer Interface

The asynchronous data transfer interface (DTI) controls the data movement between two connected PH_{op}s. In our GALS approach, the DTI will be asynchronous following the guarded communication protocol illustrated in figure 4–3. This DTI consists of a hand-shaking Input Guard (IG), an Output Guard (OG) and a data status signal generator for the input and output of a PH_{op}.

5.3.1 Hand-Shaking Cycle

There are two basic control signals involved in the guarded communication between a pair of connected modules: a request to output data when available and an acknowledge to complete a hand-shaking cycle when input data are correctly accepted. A logic module can be in either input or output mode. When new data are valid and stable on output lines of a module, the module is said to be in an output mode and an R_{out} signal is sent to the module which will accept the output data. When the destination module is ready to accept the data, the module is said to be in an input mode, and the data presented on the input lines of the module are transferred to the module. Upon the completion of this data transfer, an A_{out} signal is sent back from the destination module to the source module. This A_{out} signal frees the source module from the output mode, and resets the data output request signal. A complete hand-shaking procedure can be regarded as a cycle: a data output request signal starts a hand-shaking cycle, and an acknowledge signal terminates a hand-shaking cycle. During one hand-shaking cycle, the module which outputs data is in control until an acknowledge signal is generated, at which time the control moves to the module receiving the data. The state of the DTI between two communicating modules is determined by the way in which the request and acknowledge signal are represented. The interface is

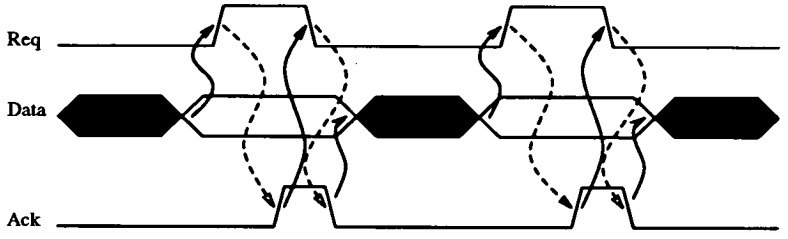


Figure 5–1: Level signalling

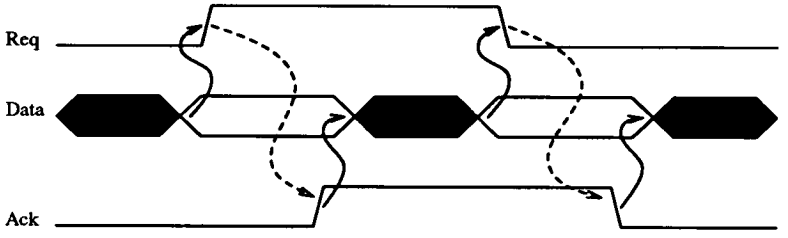


Figure 5–2: Transition signalling

said to be in a quiescent state if there are no active hand-shaking signals between them.

There are two common ways of representing the state of an asynchronous bundled data interface: level signalling and transition signalling. In the level signalling representation, which is shown in figure 5–1, both the request and the acknowledge signal are low in the quiescent state. A request event is signalled when the output module raises its R_{out} signal. The corresponding input module acts according to its own state and raises its A_{out} signal when input data are accepted. Upon receiving this acknowledge signal, the output module resets the R_{out} signal to indicate the completion of the current hand-shaking cycle, and finally the input module resets the A_{out} signal to return the interface to the quiescent state. This requires two round-loop trips between two communicating modules to complete one hand-shaking cycle. In the transition signalling representation in figure 5–2, level transitions in request and acknowledge signal are used to control hand-shaking and data communications. A data output request is signalled when the R_{out} signal is toggled. This is acknowledged by a following toggle in the A_{out} signal to complete one hand-shaking cycle. A transition in a signal is an

event, therefore a transition signalling interface is event-driven. If both the request and the acknowledge signal are initialised to low, a transition signalling interface is in the quiescent state when the request and the acknowledge signal have the same logic state. Hand-shaking Control signals require only one round-loop trip in the transition signalling interface. Thus, the time required to return to the low-low quiescent state in the level signalling interface is eliminated. The delays caused by configurable switches in interconnection paths can be very different with algorithm mappings and some of these delays may be comparable with gate delays. So this saving of time in one extra round-loop trip is very important in configurable systems which can improve the communication performance over the level signalling interface. The event-driven hand-shaking is also a concise protocol which fits well to the asynchronous guarded communication defined in section 4.3.2. Therefore the transition signalling representation is adopted for the design of our asynchronous DTI.

5.3.2 Data Status Signal

One of the key issues in designing asynchronous hand-shaking logic is to generate a data status signal which can represent the existence of valid data, for example, stable results after a data evaluation or a register write, at the output or input of a logic module. The asynchronous DTI will be activated and will generate a correct sequence of control signals upon the value of this data status signal. There are three basic structures which can be used to generate a data status signal for the output of an evaluation logic module: a pre-determined simple worst case delay unit, a data transition detector, or differential logic with complementary output data values.

Pre-determined logic module latency

It is possible to estimate a maximum latency for a logic module. Therefore, a delay unit designed with the same maximum possible latency can be inserted into

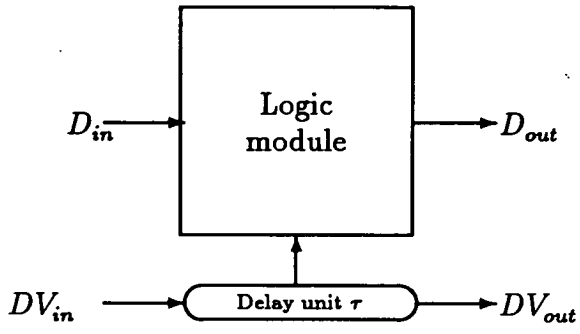


Figure 5–3: Data status signal from a pre-determined block latency

a hand-shaking path which is in parallel with a data flow path in logic modules as shown in figure 5–3. A data status signal is passed through the delay unit in parallel whilst the input data at D_{in} is processed through the logic module. The Data-Valid (DV) signal is passed from DV_{in} to DV_{out} with the same delay just as the output data becomes stable at D_{out}. This DV signal will activate an asynchronous DTI which in turn controls the data-flow rhythm in the data path. Because the delay unit reflects the worst possible latency in a logic module, the performance of this structure will also be the slowest. In our GALS approach to configurable hardware algorithms, the number of clock cycles required in each PH_{op} may be different depending on algorithms. This fixed delay unit structure is not a suitable choice for our design.

Data state transitions

The basic idea in this structure is to make use of data logic state transitions to generate a DV signal. There are many different ways to detect the logic state transition of a datum. In figure 5–4, the logic state of output data from a logic module is kept by a delay unit and then compared with the new output data logic value via an XOR tree, any logic transitions at the output of the previous stage will set the DV to high. In actual implementation, a lock mechanism is required to lock the DV at high until the current set of output data are no longer needed

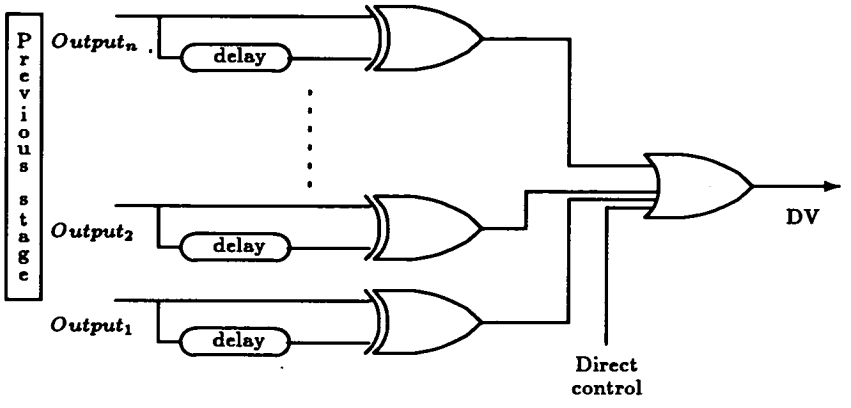


Figure 5–4: A data transition detector

by the next stage. There is one extreme situation which cannot be easily handled by this logic structure, i.e. all of the outputs ($output_1, output_2, ..., output_n$) remain unchanged but represent a new set of data. On the other hand, at least two stages of delay are introduced before a DV signal is generated. In the following sections, we will find that this structure can be used in our transition signalling DTI design where only request and acknowledge control signals are involved.

Exploitation of differential logic

Differential logic can be easily modified to generate a data status signal DV. Figure 5–5 shows such an example which is very similar to domino logic. A precharge mechanism is used in this differential logic structure with an NMOS evaluation tree and two PMOS pull-ups. Two distinct operation phases can be defined in the block: precharge/neutral and evaluation. When $I = 0$, the logic block is said to be in precharge/neutral phase since out and $\overline{out}$ are both precharged to 1 through the two PMOS pull-ups regardless the logic states of inputs to the NMOS tree. When $I = 1$, the NMOS tree is enabled and either out or $\overline{out}$ will be discharged to 0 which is determined by the input logic states to the NMOS tree. The logic block is thus said to be in evaluation state. The NAND gate in figure 5–5 acts as

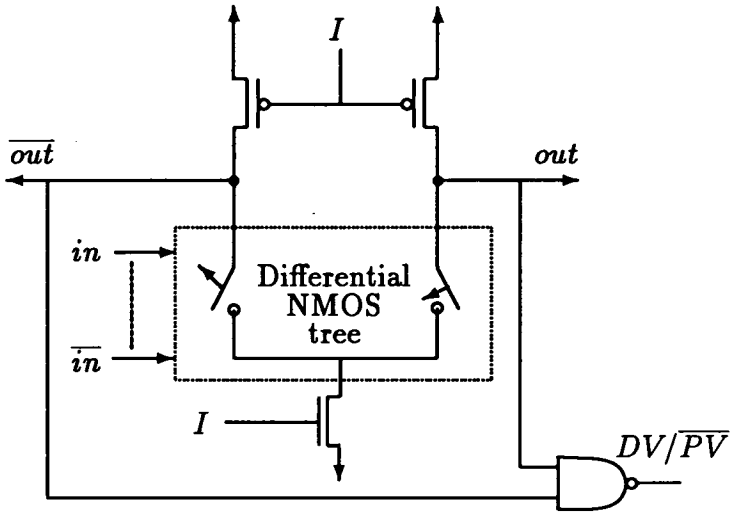


Figure 5–5: Hand-shaking signals generated from differential logic

an AND element for values at *out* and $\overline{out}$. Therefore, when (*out*, $\overline{out}$) $\longrightarrow$ 1, we have $\overline{PV} \longrightarrow 0$ which signals the establishment of a neutral state and the logic block is ready to evaluate a new set of input data. When one of the *out* and $\overline{out}$ is discharged to 0, we have $DV \longrightarrow 1$ which signals the completion of an evaluation and the availability of a stable result at (*out*, $\overline{out}$). A variation of this structure can be found in [41] for a dynamic self-timed adder design. The static NAND gate is replaced by a dynamic NAND tree and a dual-rail carry path is used to propagate a DV signal through the NAND tree for the adder.

The size of the NMOS tree depends on the number of inputs and the complexity of the logic function being implemented. The NMOS tree does not necessarily grow linearly with the complexity of the logic function because it can be optimized to share some transistors from *out* and $\overline{out}$ side of the tree.

In a GALSA system, each PH_{op} has a local clock to control its function. Different functions may require different numbers of clock cycles to complete. This differential logic structure cannot make use of this particular local clock property and can complicate the clocked logic design, so it is not used in our

GALS approach. A special Clock Management Unit (CMU) will be described later which makes use of this property to generate a DV signal at the output of the PH_{op} module.

A Tri-state Register

The data transfer between two connected PH_{op}s is through input registers. With input registers, it is possible to overlap some part of a hand-shaking cycle with the internal operations in PH_{op}s to improve the overall system performance further. For example, if a PH_{op} is activated after new data are stable in its input registers, the time required for the PH_{op} to send out an A_{out} to complete the current hand-shaking cycle is overlapped with the PH_{op}'s normal computation.

A data status signal for input registers is also required to enable the overlapping of a hand-shaking cycle and a computation. Because a normal register has two stable states 0 or 1, this makes it relatively simple to design a tri-state register with a third state similar to the DV signal of differential logic in the last section. This tri-state register can hold input data as a normal input register. It can also interact with the hand-shaking guard and the CMU to form a complete general GALS DTI.

When a normal register is in one of the two stable states, two circuit nodes can always be found whose logic levels are complementary to each other. A tri-state register can be formed by creating another stable state where these two nodes are set to the same logic level. Figure 5–6 shows the design of such a register. When it is stable, if a Data-Valid flag: $DV_R = \overline{Q \cdot \overline{Q}} = 1$, the register is in an occupied state where data must be kept and the register must be write-protected. If $DV_R = \overline{Q \cdot \overline{Q}} = 0$, the register is in an empty state and it is ready to accept a new datum.

Signal $\overline{R}$ comes from the hand-shaking guard which clears the register and keeps it in the empty state when an acknowledge event is received. The register

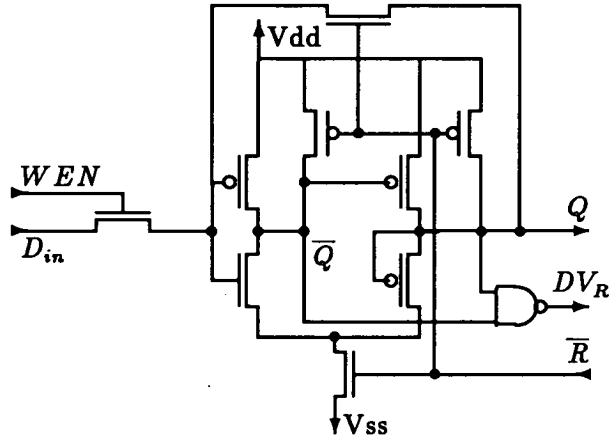


Figure 5–6: A tri-state register design with WEN, $\overline{R}$, DV_R

is write-enabled (WEN) when it is in the empty state. However, a datum can only be written into the register when an input request event is detected by the hand-shaking guard. Note that WEN and $\overline{R}$ should be mutually exclusive, i.e. they should not be active at the same time.

Although this tri-state register design requires 7 more transistors than a normal static register, it improves data transfer performance and reliability over the pre-defined worst case delay structure. On the other hand, because a delay unit needs at least 6 transistors, the cost of our tri-state register design is comparable in total. There are also two choices for multiple inputs in a logic module: to use tri-state registers for all inputs to gain maximum performance or to use one tri-state register and normal registers for the rest of inputs to minimise cost. In the latter case, the tri-state register acts as a register and a delay unit as well.

5.3.3 Event-Driven Hand-shaking

At the output side of an output module, three events can be identified in a transition signalling hand-shaking cycle: request, acknowledge, and clear. At the input side of an input module, three events can also be identified: request, input-setup,

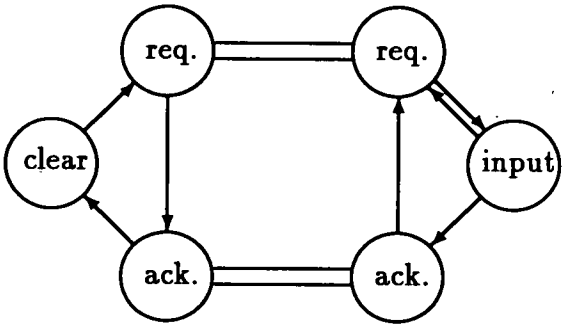


Figure 5–7: Event sequence in event-driven hand-shaking

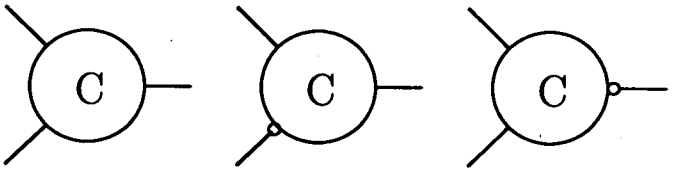


Figure 5–8: Muller C-element and its variations

and acknowledge. The request and acknowledge event are the same for a pair of communicating modules. Therefore there is a total of four events involved. The transition signalling hand-shaking logic derived from these events is event-driven hand-shaking. As illustrated in figure 5–7, events at an output always happen in the order of request $\longrightarrow$ acknowledge $\longrightarrow$ clear $\longrightarrow$ request and events at an input are in the order of request $\longrightarrow$ input-setup $\longrightarrow$ acknowledge $\longrightarrow$ request.

Six event processing elements are illustrated in [128]. Two most basic event elements are exclusive OR (XOR) and Muller C-element [100]. An XOR element implements the OR logic for events, that is, when either input of an XOR changes state, its output also changes state. The Muller C-element carries out the AND operation on events. When both inputs of a C-element are in the same logical

state, the C-element and its output copy that state. When the two inputs differ, the C-element uses its internal storage to keep its output on previous state. Thus only after an event takes place on both of its inputs will a C-element produce an event at its output. Both XOR and C-element can be easily generalised to multiple inputs. The XOR element can be represented by the conventional XOR symbol. The Muller C-element and its variations are shown in figure 5–8. Since the Muller C-element uses the concept of state, i.e. the logical levels of signals, the level signalling hand-shaking scheme can be easily implemented with the C-element [128,99,98]. In our transition signalling guarded communication approach, an event-driven Input Guard and Output Guard are to be designed whilst the C-element can be used to merge the multiple independent input request or input acknowledge events to save the cost of multiple input/output guards.

5.3.4 An Event-Driven Register Transfer Interface

We developed a transition signalling DTI for register data transfer based on the XOR data transition detection structure illustrated in section 5.3.2. One basic assumption, to ensure each new input state is captured for the structure, is that the logic state generated after an event on a signal always differs from its immediately previous logic state. This condition is guaranteed on the request and acknowledge control signal in the transition signalling hand-shaking representation described in section 5.3.1.

The schematic of the top level event-driven register transfer DTI is shown in figure 5–9. In the figure, there is an input guard (IG), an output guard (OG), a tri-state register (R3S) and some glue logic. The $\overline{DV_F}$ is a flag signal for the output data status in a function module PH_{pop}. T_{in} and T_{out} are input and output transition event signals. EVT_i and EVT_o are the flags for the input and output events respectively. The *init* signal initialises the whole interface after power on.

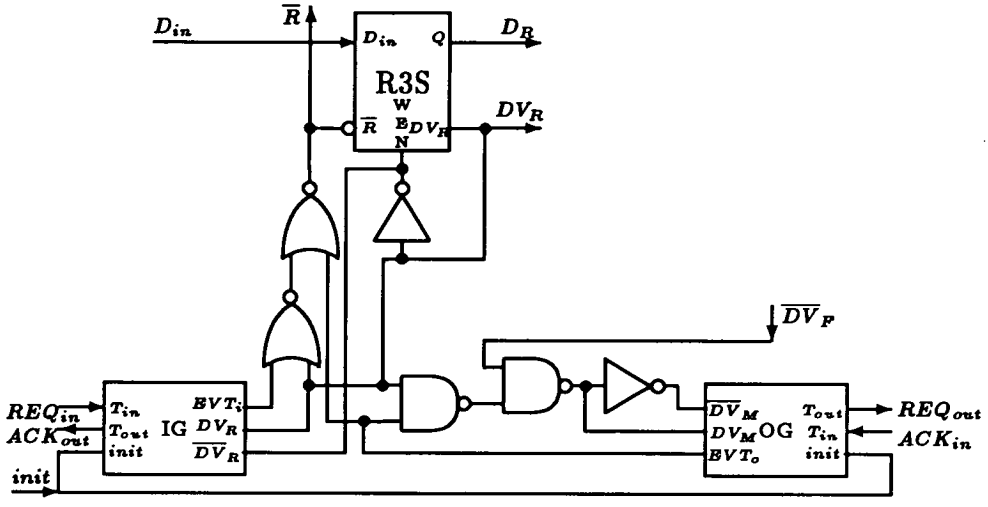


Figure 5–9: An event-driven DTI

After a further analysis on the event sequence illustrated in figure 5–7, detailed control signal state transition graphs for the IG and OG are generated as shown in figure 5–10. Throughout the rest of this thesis, superscripts $+$, $-$, T, $T+$, $T-$ to any signal name represent logic high, logic low, a transition event (any direction), a transition to high and a transition to low respectively. All input signals are initially set to low by an external *init* signal except ACK_{out} and ACK_{in} which are initially set high.

Figure 5–10(a) shows the signal state transitions for an input guard. An input communication cycle starts with a transition event REQ_{in}^{T} on T_{in} which sets EVT_i^+. If DV_R^+ by which is meant the tri-state input register R3S is in occupied state, EVT_i will wait until DV_R^{T-} to proceed. When DV_R^-, R3S starts to take in a new datum and resolves to a stable register occupied state with DV_R^{T+}. Once DV_R^+, an event ACK_{out}^{T} is generated on T_{out} and EVT_i is cleared. Another input cycle starts again when REQ_{in}^{T}. A similar signal state transition graph for an output guard is shown in figure 5–10(b). DV_M (Module Data-Valid) is another state flag similar to DV_R in a tri-state register which represents the state of an entire logic module PH_{op}. DV_M is derived from DV_R and DV_F. If there is no valid data at the output of a module, $DV_M = 0$. A DV_R^{T+} will activate a new computation

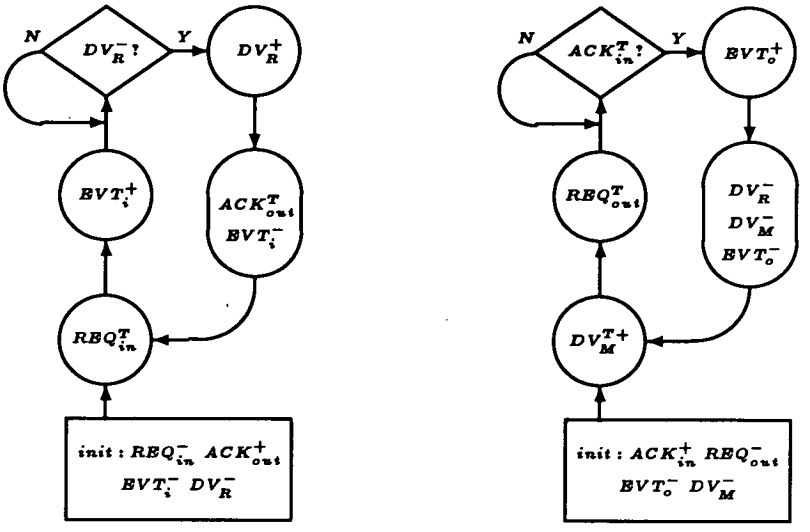


Figure 5–10: State transition graphs: (a) input guard, (b) output guard

in the PH_{op} and a $\overline{DV_F}^{T-}$ occurs after a pre-defined number of clocks. An output cycle starts with $\overline{DV_F}^{T-}$ which enables the OG to send a REQ_{out}^T event to its next module. The OG then waits for an ACK_{in}^T event from the next module. Once an ACK_{in}^T is received, the OG sends the clear signal $\overline{R}$ to clear the DV_R and $\overline{DV_F}$ flag in the current module. A new computation and output cycle starts again following a new DV_R^{T+} from the next input cycle.

The IG and OG are designed according to the signal state transition graphs in figure 5–10. The schematics of the designed IG and OG are shown in figure 5–11. The heart of this guard logic is a master-slave flip-flop which keeps T_{in}'s previous state. When $W = 1$, the slave Dlatch is open for writing; when $W = 0$, the master Dlatch takes T_{in}'s new state and the slave Dlatch is locked to keep T_{in}'s old state. The XOR element is used to set the input event flag EVT_i/EVT_o when T_{in} makes a transition.

A data transfer cycle in the event-driven DTI in figure 5–9 starts with REQ_{in}^T. Then the IG follows the state transition graph in figure 5–10(a). When DV_R^{T+}, the function module is activated by the CMU and $\overline{DV_F}^{T-}$ after a number of clock cycles following the activation of the function module. The OG is activated when

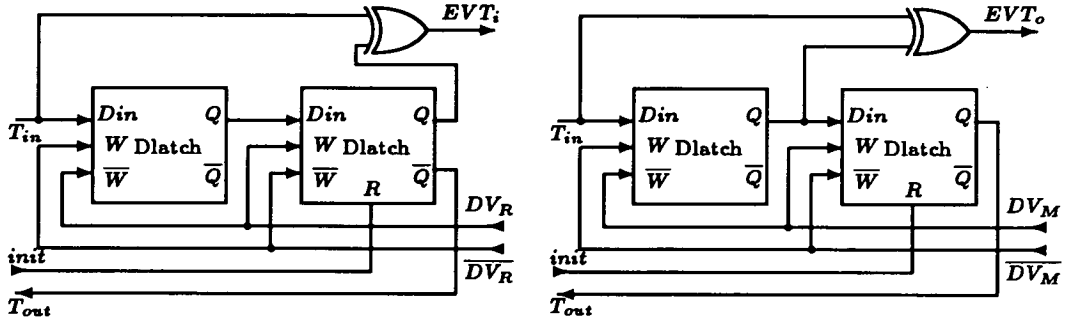


Figure 5–11: Event-driven (a) input guard, (b) output guard

$\overline{DV_F}^{T-}$ and follows the transition graph shown in figure 5–10(b). $\overline{R} = 0$ when EVT_i^- and DV_R^- (no REQ_{in} event, keeping R3S empty), or $\overline{R}^{T-}$ if an ACK_{in} event is received and EVT_o^{T+} by the OG (clear R3S and $\overline{DV_F}$). $\overline{R} = 1$ will keep the value in the R3S.

The advantage of this event-driven guarded DTI design is that it is now an easy task to design a total scalable modular system because a logic module with this DTI is completely self-contained and portable. A system constructed from such modules can be easily scaled up or down. A module can be easily adapted in any other GALS based system without worrying about the system level timing design and scheduling problem.

5.4 The Implementation of a PH_{op}

The abstract structure of a PH_{op} has been illustrated in section 4.4.3. The essential function of a PH_{op} is to transform, in its primitive programmable operator PH_{pop}, a set of input data selected from a set of input ports, and output modified data to a set of output ports. The data transformation function and I/O ports selections are directly programmed by bits stored in an execution code register (ECR). The data transfer interface (DTI) and the clock management unit (CMU) form a GALS

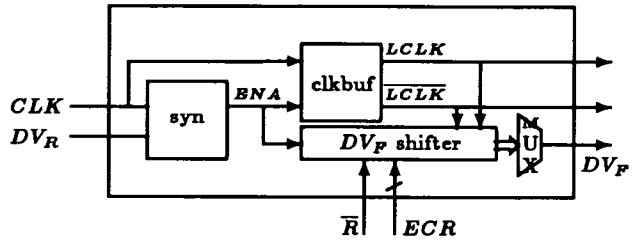


Figure 5–12: A Clock Management Unit

interface that connects and controls the data transfer between PH_{op}s. In this section, we describe the design and implementation of the PH_{op} in detail.

5.4.1 The Clock Management Unit

In a self-timed system, a combinational function module is activated immediately after a DV_R^{T+} in the tri-state input register, and the completion flag $\overline{DV_F}$ for a computation can be generated by adopting differential cascade voltage logic [99] or a dual-rail complementary carry chain [41]. Because a PH_{op} runs with a local clock and has asynchronous data input from the event-driven DTI, a clock management unit (CMU) is required to synchronise the asynchronous input data with the local clock and computation in the PH_{op}. The CMU also controls the DV_F flag for the PH_{op}. Figure 5–12 shows the block diagram of the CMU. syn is a synchroniser, clkbuf is a local clock buffer and DV_F shifter is a shifter that controls the DV_F flag.

Synchronisation

A common approach to interfacing asynchronous input data with a clocked logic module is to use a synchroniser. A synchroniser can be a type of flip-flop controlled by a local clock to synchronise an asynchronous input data to an event of the local clock (usually a rising clock edge). A particular problem in synchronisation is the possibility of entering a metastable state in a flip-flop. A metastable state may

happen when a data input changes just before the flip-flop write-enable goes low. If a metastable state appears the flip-flop may not be able to resolve to one of its two stable states for the input data for a long period of time. If this happens, this is called a synchronisation failure. Although various solutions, such as a stoppable clock, a pausable clock and extensible clock schemes, have been investigated [84,107,127,83], the possibility of synchronisation failures remains a fundamental problem in these approaches. Given a synchroniser design, a probability of synchronisation failure can be estimated [63]. Then, various design techniques can be used to reduce the synchronisation failure probability to an acceptable low level.

We propose a different approach to synchronisation in our implementation. Firstly, the tri-state input register, as described in section 5.3.2, is controlled by the event-driven DTI instead of the local clock. Thus, the asynchronous input data can be safely locked into the input register without the interference from the local clock. Secondly, the synchronisation between the asynchronous data in the input register and the local clock for a PH_{op} is controlled through the DV_R flag of the tri-state input register and the local clock. In this approach, the probability of generating wrong results caused by a synchronisation failure can be reduced further because data in the input register are always set up correctly before a possible synchronisation failure between the DV_R and the local clock, and the PH_{pop} can still process the correct input data to produce correct results.

Because a locally synchronous computation module is activated by the DV_R flag from the tri-state input register, a safer and faster design without using synchroniser is to assign an independent local clock generator in each module. The local clock generator and hence the computation function are activated by the DV_R^{T+} and stopped by the $\overline{DV_F}^{T+}$. In this way, each clock cycle can be efficiently used in the computation module and there will be no risk of synchronisation failure. However, this structure is very expensive in hardware when the complexity of a module is relatively low and the number of modules in a system is high. So it is not adopted in our configurable GALSA design.

Another choice is to distribute a global clock to each independent PH_{op} for local use. Different from the strict restriction on minimum clock skews imposed on distributing a global clock for a synchronous system, the only requirement to distribute a global clock in our GALSA system is that it has enough driving power to each PH_{op}. Clock skews caused by the distribution are no longer a fatal problem because the distributed clock is only local to a PH_{op}, i.e. clocks to different PH_{op}s can be regarded as independent whilst sharing a common clock source. Given the restriction that the physical size of a PH_{op} is either within the size of an equipotential region of a chosen technology or a scale that a local clock can be easily distributed inside it with the minimum clock skew restriction, the effort and cost required to distribute such a global clock for local use in a GALSA system is much less than in a synchronous system.

The synchronisation problem now is: DV_R^{T+} can happen at any time relative to the rising edge of a running local clock in a PH_{op}. Therefore, the activation of the PH_{op} has to be at the start of the first clock cycle after the DV_R^{T+}. Figure 5–13 shows the design of such a synchroniser for the asynchronous DV_R and a local clock CLK. ENA (enable signal) is a control signal which activates the PH_{op} timed by CLK when it goes high. CLK is a buffered local clock driven by a free-running system clock. When DV_R is low, ENA is held low which sets the PH_{op} idle. An ENA^{T+} follows the first CLK^{T+} after a DV_R^{T+}. The major part of a CLK^{T-} time interval can be safely used to sample the DV_R signal from the tri-state register. A sampling time interval is called a DV_R detection window in the CLK. The sampling rate is the same as the CLK frequency. Once a DV_R^{T+} is detected in a DV_R detection window, the A signal in figure 5–13 will be locked to low, and ENA will be raised to high at the immediate next CLK^{T+} edge after A^{T-}.

There is also a probability of synchronisation failure with this synchroniser because the gate G1/G2 may enter the metastable state where signal A is balanced between making a decision to resolve to a logic high or low state. This is depicted

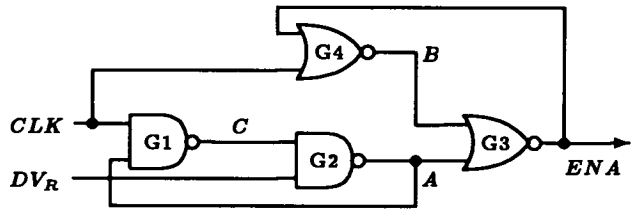


Figure 5–13: A synchroniser for DV_R and CLK

by VC and VA' on the G1/G2 input and output voltage transfer curves in figure 5–14, when both DV_R and CLK input are very close to the effective threshold voltage of G1/G2 at the same time. A metastable state may appear if a DV_R^{T+} happens at a very short time just before the end of a DV_R detection window or at the same time as a CLK^{T+}. This short period of time is called a synchronisation risk zone in a DV_R detection window. In practice, noise (switching and thermal) or a slight initial imbalance on signals can eventually push the signal A one way or the other. The time taken to reach an output decision is called the decision time t_d [63]. The synchroniser design shown in figure 5–13 allows a whole clock cycle T_C for the output A from G1/G2 to resolve. If $t_d > T_C$, a synchronisation failure may be caused. Using the formula given in [63], a Mean Time Between Failure ($MTBF$) can be estimated as:

$$MTBF = \frac{1}{2\tau \times f_C \times f_{DV_R}} \times e^{t_d/\tau} \tag{5.1}$$

where f_C and f_{DV_R} are clock and DV_R input frequency respectively, t_d is assumed as T_C, τ is a circuit parameter which is typically $0.2ns$ in the chosen 2μm CMOS technology. Given $f_C = 100MHz$, $f_{DV_R} = 25MHz$, we got $MTBF = 5.18 \times 10^{15}$ seconds. This is 10^7 times longer than the expected 15 years normal silicon chip's life span.

Circuit and layout design techniques can also be used to reduce the synchronisation failure probability to an even lower level if the NMOS and PMOS effective gains of G1, G2, G3, and G4 are designed to meet the following conditions:

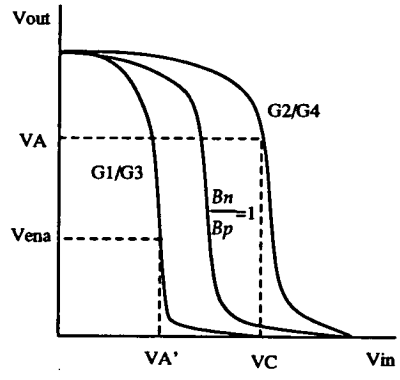


Figure 5–14: Voltage transfer curves for different $\frac{\beta_{neff}}{\beta_{peff}}$

$$\frac{\beta_{neff}}{\beta_{peff}} > 1 \text{ for G1 and G3} \tag{5.2}$$

$$\frac{\beta_{neff}}{\beta_{peff}} < 1 \text{ for G2 and G4} \tag{5.3}$$

so that the four gates effective threshold voltages will be set apart as shown in figure 5–14. If a metastable state appears in G1/G2, V_A may be vibrating in a small range around the VA point for G2 in figure 5–14, $VA' < V_A < 2VC - VA'$. For G3, the ENA signal will have much better chance to stay outside the metastable state by either keeping 0 or flipping to 1 state for the synchroniser. Therefore, the worst case delay after a DV_R^{T+} is one clock cycle before a PH_{op} is activated. With this analysis on $MTBF$ and this circuit design technique, this synchroniser should be safe enough to be used in our GALSA system.

To verify our analysis, an intensive simulation on the synchroniser design with the Hspice circuit simulator has been carried out. The simulation strategy is to move a DV_R^{T+} edge across a region from a CLK^{T-} edge to a point just after a CLK^{T+} edge. Although it is impossible to simulate a consecutive move of the DV_R edge in Hspice, we identified a synchronisation risk zone which is about a 10ps overlap when a DV_R^{T+} edge and CLK^{T+} edge are between 1.7v and 3.3v. If both DV_R^{T+} and CLK^{T+} fall in this voltage and time risk zone, it can take a longer time (about 1.7ns) than usual (0.4ns) for the synchroniser to resolve V_A to a

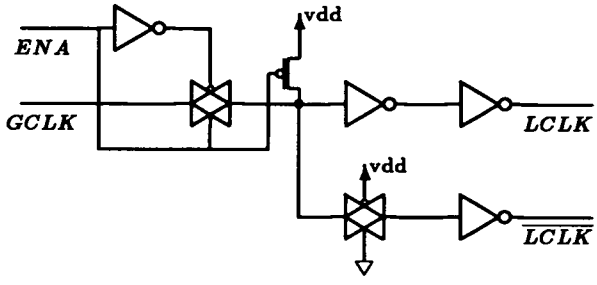


Figure 5–15: A local clock buffer

stable state. When this happens, V_{ENA} is still kept as 0 or changes to 1 after 1.5ns delay. Nevertheless, no synchronisation failure is seen in all simulation results and a worst case of one clock cycle delay does appear at a point in the synchronisation risk zone. Several typical simulation waveforms obtained by moving the DV_R^{T+} edge are given in the appendix.

Local clock buffer

The local clock to each PH_{op} is driven by a shared clock source in a GALSA system, and a PH_{op} is activated by the ENA signal from the CMU in the PH_{op}. A local clock buffer, as shown in figure 5–15, is used to generate a two phase local clock from a globally distributed clock $GCLK$. The two phase local clock is activated and stopped by the ENA signal. This buffer plays two important roles: it compensates the time lost in the synchroniser for the $LCLK$ phase, and it holds $LCLK$ and $\overline{LCLK}$ to minimise the power consumption in a PH_{op} when the PH_{op} is not active.

DV_F control

As illustrated in section 5.3.2, a data status flag DV_F for a locally synchronous PH_{op} can be generated by taking advantage of the number of clock cycles required to run a programmed function.

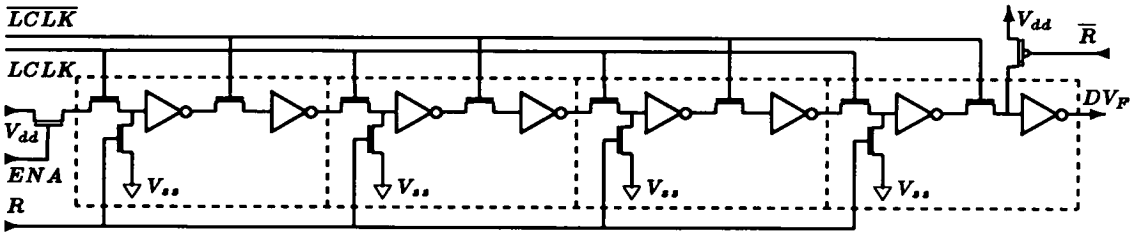


Figure 5–16: A dynamic shifter for DV_F^+ control

A DV_F^+ is generated through a simple dynamic linear shifter controlled by the local clock $LCLK$. A 4-stage dynamic shifter is shown in figure 5–16. The total length of the shifter L_s equals the possible maximum number of clock cycles required by a PH_{op}. A multiplexer is used to select an output from stage $n, 1 \leq n \leq L_s$, in the shifter. n is specified in a clock cycle field in the ECR. The shifter is activated by the ENA signal from the synchroniser. The DV_F flag is cleared by the $\overline{R}$ signal from the event-driven DTI.

5.4.2 An Event-Driven General GALS Logic Module

A complete event-driven GALS data transfer interface can now be constructed by combining the event-driven DTI described in section 5.3.4 and the CMU depicted in the last section. With this GALS data transfer interface, a general GALS logic module, which can be used to build a GALS system, is shown in figure 5–17. In the figure, the GALS interface is enclosed in the dash box and the logic module can perform arbitrary logic functions with the local clock control. This general GALS building module exhibits an excellent portability and timing independency which meets the requirement of developing configurable hardware algorithms we aim to achieve in this project. The issue of using this general GALS logic module to construct a general GALS system or a GALS pipeline is discussed in [40].

The GALS logic module works in three phases in a GALSA system: the pre-loading of ECR and other configuration bit streams, the initialisation ($init = 1$) of the GALS interface, and the normal system operation phase ($init = 0$) where

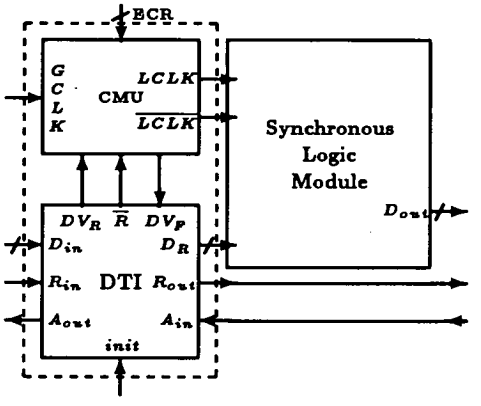


Figure 5–17: A general GALS logic module

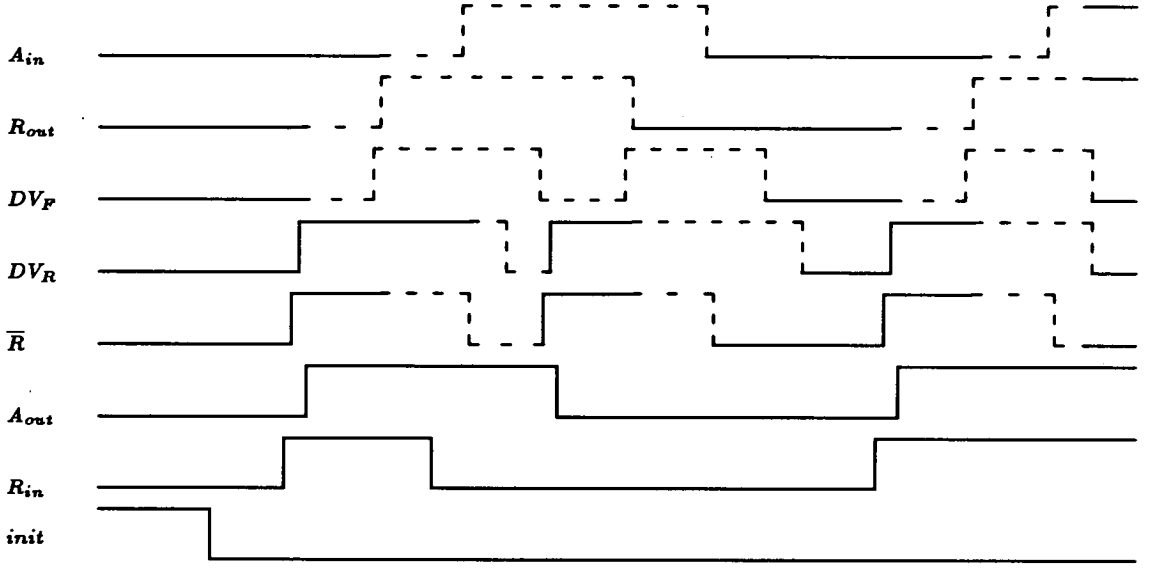


Figure 5–18: Waveforms for the event-driven GALS data transfer interface

sequences of asynchronous data transfer and computation events take place. At the boundaries of a GALSA system, R_{in} and A_{in} must be initialised from external sources.

Figure 5–18 depicts a group of typical waveforms for the GALS interface signals in figure 5–17. It is noted that all the rising and falling edges of the signals illustrated in figure 5–18 represent effective events. A complete hand-shaking cycle consists of a sequence of such events. Also the design in figure 5–17 is a delay-insensitive GALS system. That is, results from the synchronous logic module will

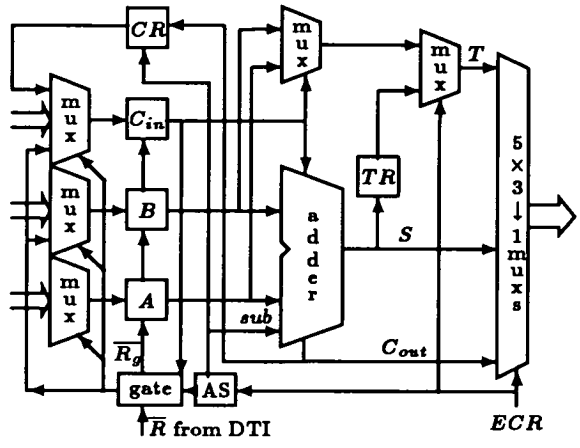


Figure 5–19: A PH_{pop} and its I/O multiplexers

not be affected no matter what the length of delay is involved in the communication path between two communicating modules and the logic module works correctly with a properly chosen clock speed. If the delay in a communication path changes, only the time to generate correct results will vary accordingly.

5.4.3 The PH_{pop} and I/O Selector

In a primitive hardware operator PH_{pop}, there are three input registers for A, B and C_{in} respectively. These input registers are controlled by the event-driven DTI. There is one carry register (CR) for C_{out} and one temporary register (TR) for S output. For the I/O selector in a PH_{op}, there are three 6-to-1 input multiplexers to select inputs for A, B and C_{in}, and five 3-to-1 multiplexers to select S, C_{out} or T to 5 output ports. Figure 5–19 depicts the block diagram of the PH_{pop} and its associated I/O selectors. All I/O multiplexers are controlled by the ECR.

In the heart of the PH_{pop} there is an Execution Unit (EU) which can carry out the primitive functions described in table 4–1. There are many different ways to implement such an EU. A variety of design structures was considered for the EU. One interesting ALU structure which is very flexible is given in Mead and Conway's book [95, chapter 5]. However, it requires 12 bits of control to form an

ALU with three general functional blocks consisting of pass transistors. Because our target GALSA system is arithmetic computation intensive algorithms oriented, we decided to design an EU based on a full adder similar to the PEs in most of the massively parallel processing systems.

The EU basically consists of a full adder and a 2-to-1 multiplexer. This is sufficient to run the primitive functions listed in table 4–1. We first analyse the functions that can be performed by a 1-bit full adder and determine how a full adder can be programmed to implement these functions. In the following sections, 1-bit full adder is assumed if not specified otherwise.

A full adder takes in three inputs A, B and a carry C_{in} from a lower bit, and outputs a sum S and a carry C_{out} to a higher bit. S and C_{out} are calculated by equation 5.4 and 5.5. From these two equations, it can be seen that a set of functions as depicted in table 5–1 can be realised by setting C_{in} and/or B input and choosing either S or C_{out} as output respectively.

$$\begin{aligned} S &= A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}} + \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} \\ &= A \oplus B \oplus C_{in} \end{aligned} \tag{5.4}$$

$$C_{out} = A \cdot B + B \cdot C_{in} + A \cdot C_{in} \tag{5.5}$$

There are many different ways to design a full adder [137, section 8.2]. An adder design based on transmission gates, as depicted in figure 5–20, is adopted in our implementation because this adder can be implemented efficiently in CMOS from the area point of view and it has an equivalent sum and carry delay with buffered and non-inverted sum/carry output. The part inclosed in the dash box in figure 5–20 is capable of generating the $A \oplus B$ and $\overline{A \oplus B}$ function. This adder has been intensively simulated with the Hspice simulator with the chosen Mietec CMOS technology, and proved correct. A *sub* control input to the adder selects either B or $\overline{B}$ for an addtion or a 2's complement subtract function. If the adder is programmed for a 2's complement subtract function, $\overline{B}$ is selected as an input

Function	inputs			outputs	
addition	C_{in}	A	B	S	C_{out}
subtract	C_{in}	A	$\overline{B}$	S	C_{out}
inverse	0	A	1	$\overline{A}$	×
OR	1	A	B	×	$A \vee B$
AND	0	A	B	×	$A \wedge B$
XOR	0	A	B	$A \oplus B$	×
XNOR	1	A	B	$\overline{A \oplus B}$	×

×: Don't care

Table 5–1: Possible functions from a full adder

to the adder and C_{in} is initialised to 1. The AS block in figure 5–19 sets the *sub* flag and initialises the carry register CR if a subtract op-code is set in the ECR. If a result S is to be saved into the TR, two clock cycles are required, one for add/sub function and one for TR write. This TR is included in our PH_{pop} design because it is anticipated that sometimes an S value may be required at the next cycle of computation.

There are up to five possible input request and five input acknowledge signals to each PH_{op}. Two 5-input Muller C-elements are used to AND together input request and acknowledge events to get one REQ_{in} event and one ACK_{in} event to the event-driven DTI. Figure 5–21 shows the design for a simple 5-input Muller C-element. The DFF, which is similar to the Dlatch used in the Input Guard and Output Guard but without the write control W, is initialised to 0. The other part of the PH_{op} ensures that at least one of the $R_i, i = 0, 1, \cdots, 4$ is initially zero. The ACK_{out} signal is looped back as a request signal from unused input ports, and the REQ_{out} signal is looped back as an acknowledge signal from unused output ports.

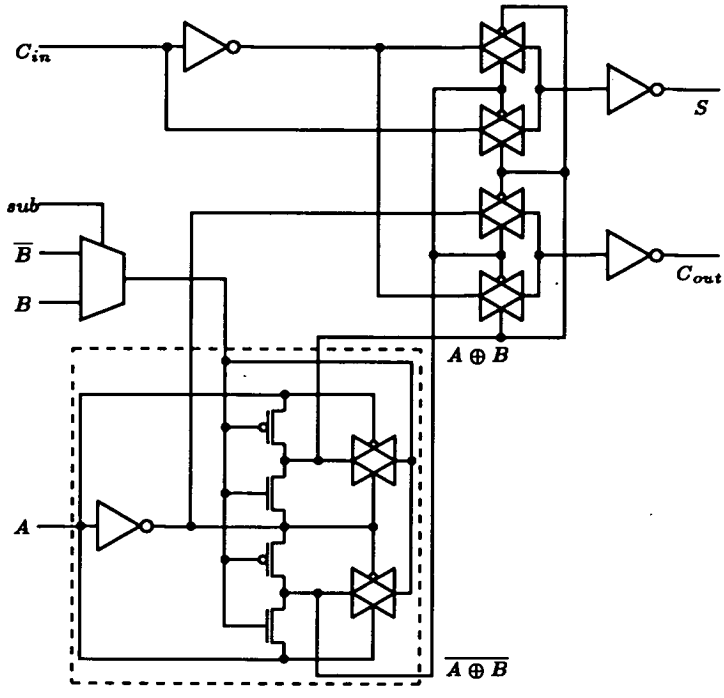


Figure 5–20: A transmission gate full adder

The ACK_{out} and REQ_{out} signal are also sent to the selected input and output ports respectively.

5.4.4 The Execution Code Register

The Execution Code Register sets the function of a PH_{pop} and the number of clock cycles required to run the function. It also sets input constants and selects I/O ports for the PH_{pop}. There are four fields in an ECR: op-code, N_{clk}, input constants, and I/O port select, as shown in figure 5–22. The content of the Execution Code Register is preloaded at the same time as the routing network is configured.

The first three bits b_0, b_1, b_2 in the op-code field define, in conjunction with the C_{in}^0 and B^0 constant field b_6, b_7, the function to be performed in a PH_{pop}. Functions and their corresponding op-codes are listed in table 5–2. b_3 is a pattern bit for the gate function. The next N_{clk} field has two bits b_4 and b_5. N_{clk} defines the number of clock cycles required by an EU function. In this case, at most 4 clock cycles can be accommodated. This N_{clk} field makes it possible to design

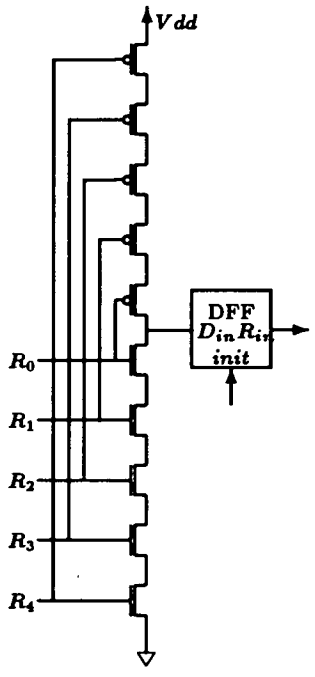


Figure 5–21: A 5-input Muller C-element

op-code ($b_2b_1b_0$)	function(s)
0 0 0	$\neg$ $\vee$ $\wedge$ $\oplus$ $\overline{\oplus}$
0 0 1	+
0 1 0	−
1 0 0	merge
1 1 1	gate

Table 5–2: Functions defined by op-code

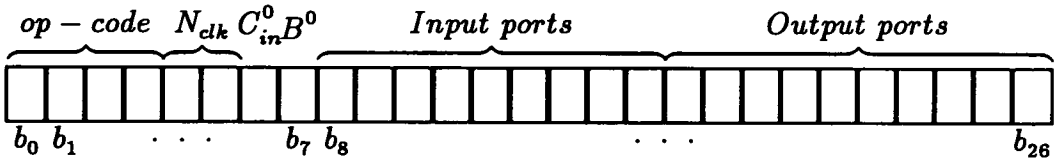


Figure 5–22: The Execution Code Register

more complicated synchronous EU functions for a PH_{pop} in the future, and more bits may be used in this field to allow more clock cycles for a function. The N_{clk} field is fed into the CMU in a PH_{op}.

Therefore for the *AS* block in figure 5–19, the *sub* flag is generated by equation 5.6 according to table 5–2:

$$sub = \overline{b_0} \cdot b_1 \tag{5.6}$$

Functions in table 4–1 that have to be implemented separately are the 2-to-1 merger and the gate. A 2-to-1 multiplexer is used for a 2-to-1 merge function with (A, B) as inputs and C_{in} as the merger control. The logic design for the *gate* block in figure 5–19 is shown in figure 5–23. The match flag $\overline{M} = 0$ and the *gate* is open if C_{in} matches the pattern b_3 in the ECR; otherwise $\overline{M} = 1$ will keep the *gate* closed and set the reset signal $\overline{R_g} = 0$ to clear the current node (DV_R and DV_F), and wait for the next cycle of input data and Boolean control input. For other functions, $\overline{R}$ from the DTI is passed directly to $\overline{R_g}$. The T output is selected from either the merger multiplexer output or the temporary register TR. If $b_2 = 1$, T selects the output from the merge or gate function.

There is one input and one output port in each routing channel, so we have a total of five input and five output ports in the three vertical and two horizontal channels: vch$< 0 : 2 >$ and hch$< 0 : 1 >$. b_8 to b_{16} are for A, B and C_{in} operands. Each operand needs 3 bits to control a 6-to-1 multiplexer which chooses an input from the five possible input ports or a constant. b_{17} to b_{26} are for the three output results, S, C_{out} and a T. Every 2 bits are used to control a 3-to-1 multiplexer which

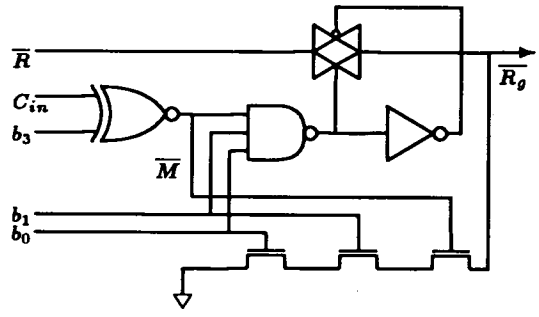


Figure 5–23: Gate logic block

A, B, C_{in}	vch0	vch1	vch2	hch0	hch1	constant/register
$b_{i+2}b_{i+1}b_i$	001	010	011	100	101	000

(a) Input port selection for A, $i = 8$; B, $i = 11$; C_{in}, $i = 14$

vch< 0 : 2 >, hch< 0 : 1 >	S	C_{out}	T	NIL
$b_{j+1}b_j$	01	10	11	00

(b) Selection of S, C_{out}, or T to output ports, $j = 17,\ 19,\ 21,\ 23,\ 25$

Table 5–3: Bit settings for I/O port selection in an ECR

selects one of the three outputs to one of the five output ports vch< 0 : 2 > and hch< 0 : 1 >. T is an output which is from either the 2-to-1 merger multiplexer or a PH_{pop} internal temporary register (TR) as shown in figure 5–19. With this output structure, it is possible to broadcast one output to up to five different PH_{op}s and process returned acknowledge signals properly. The I/O channel port selection table 5–3 elaborates b_8 to b_{26} settings for the I/O ports.

From table 5–2 and 5–3, the content of the ECR can also be expressed in text format, like an assembly language, for clarity and easy understanding. Two execution code examples are given in table 5–4. The first example is an execution code for an ADD, with A input from vch0, B input from vch1 and C_{in} input from an internal feedback carry register (CR); S output is duplicated to vch0 and hch0, T is output to vch2, and C_{out} to hch1. This function requires 2 clock cycles to

Function	ECR code	Text expression
$A+B$	00101000100010000100011100 1	ADD 0 2 0 0 vch0 vch1 CR S NIL T S C_{out}
$A \oplus B$	00000000001101111110000010 00	XOR 0 1 0 0 hch0 hch1 CST S NIL S NIL NIL

Table 5–4: ECR execution codes and text expressions

complete. The second example sets an XOR function with A input from hch0, B input from hch1, $C_{in} = 0$ as a constant (CST); the result S is duplicated to vch0 and vch2. The other output ports are not used.

5.4.5 Multiplexers

Multiplexers are heavily used in many array architectures and configurable architectures. The PH_{op} in a GALSA system is no exception as this can be seen from figure 5–19. A detailed analysis on various multiplexer designs based on the conventional CMOS technology can be found in [58].

Two particular multiplexer designs are preferable: one RAM control per switch or NMOS pass transistor trees. The multiplexer constructed from one RAM per switch scheme has the best performance because there is only one transistor switch to connect a path. The area of this type of multiplexers grows rapidly when the number of inputs increases. On the other hand, an NMOS pass transistor tree has the area advantage while its performance is slightly slower than that of the one RAM per switch scheme if the number of inputs is not too high. For example in an NMOS 6-to-1 multiplexer tree as depicted in figure 5–24, only two more pass transistor delays are added in each input to output path but saves 3 RAM bits compared with the direct RAM control design. We adopted a compromise approach for area and performance: directly RAM controlled transistor switches for the Routing Cells and NMOS pass transistor trees for the multiplexers in a PH_{op} design. The 6-to-1 multiplexer in figure 5–24 is used as the input selector in figure 5–19. A 2-to-1 or 3-to-1 multiplexer is just a small sub-tree (x_0, x_1) or

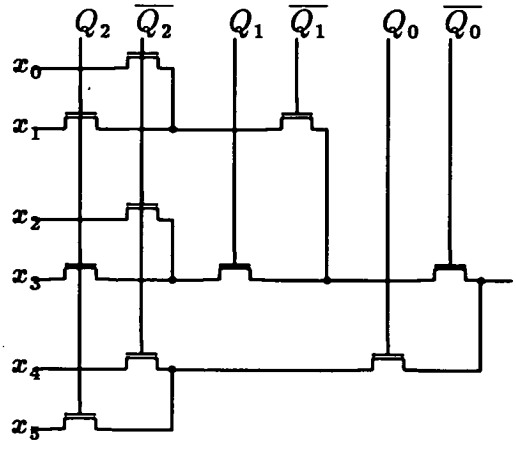


Figure 5–24: A 6-to-1 NMOS pass transistor tree multiplexer

(x_0, x_1, x_2) of the 6-to-1 multiplexer. Controls of these multiplexers come from the corresponding fields of the ECR as illustrated in the last section.

5.5 The Routing Network

The implementation of the routing network in a GALSA system is based on the discussion on network principles in section 4.2.5 and the decision on the Routing Cell capability made in section 4.4.2.

5.5.1 Switches

A basic switch device, that can be used to connect or disconnect two points, must be chosen before we design the switch unit for the Routing Cell. With conventional CMOS technologies, there are three possible choices for a basic switch device: a transmission gate, an NMOS pass transistor, or a tri-state non-inverting buffer.

A transmission gate or tri-state buffer switch requires Q and $\overline{Q}$ from a Boolean control, but a pass transistor switch needs only one Q from a Boolean control. Both transmission gate and pass transistor switches have native bi-directional switching capability whilst it is very expensive to have bi-direction flow capability with tri-state buffer switches. Among these three types of switches, a tri-state buffer switch has its own driving power at the expense of larger area. A pass transistor

switch is the smallest amongst the three switch types, so are parasitic capacitances in an NMOS switch. The NMOS threshold voltage (V_T) loss, ($V_{dd} - V_T$) when a V_{dd} passes through the NMOS pass transistor, is the major drawback. The "bootstrap" technique was considered and simulated for restoring the lost high voltage. However, this technique is not recommended in conventional CMOS processes because bootstrapping is very likely to the lead to devastating latch-up effect.

Further analysis and simulations show that if a set of NMOS pass transistors are connected in series, the high voltage degradation after the first pass transistor's V_T loss will be just the voltage drop through the effective "on" resistance of the conducting transistor. Because this resistance voltage drop is very small, the V_T voltage loss is no longer a concerning factor after the first pass transistor switch. The only problem left is that the more the pass transistors are in series, the slower will be the following waveform rising edges. Although there is almost no high voltage loss with transmission gate switches, the delays of both rising and falling edges are not greatly improved for a transmission gate switch because of the larger parasitic capacitances with one NMOS and one PMOS in parallel. Two serial switch chains with 8 NMOS pass transistors and 8 transmission gates (TG) are simulated with Hspice. The NMOS switch chain is also simulated with two power supplies: $5V$ for data signals and $7V$ for switch control. Table 5–5 shows the delays with one driver before and one driver after each switch chain. From this table, it is clear that the TG chain has poor low signal propagation speed and marginally better high signal propagation speed than the NMOS chain. The NMOS chain with a separate $V_{dd} + V_T$ switch control voltage supply shows the best signal propagation speed for both low and high signals. The transmission gate switch will have even worse performance with the large parasitic capacitances in our routing network because a switch chain can be branched. On the other hand, NMOS pass transistor switches, which will be scattered around asynchronously communicating PH_{op}s, meet the requirement of the delay-insensitive

Delays (ns)	TG chain	NMOS chain	NMOS chain ($5V + 7V$)
Low Signal	7.5	4.7	4.1
High Signal	9.1	10.7	4.9

Table 5–5: Delays in switch chains

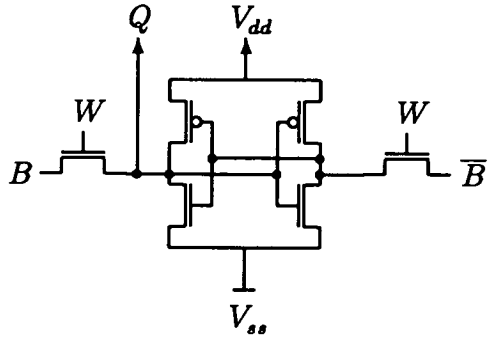


Figure 5–25: A six transistor static CCM

DTI described in section 5.4.2. Therefore, the NMOS pass transistor is chosen as the switch device in the switch unit design. We also designed the routing network and the configuration control memory with two separate power supply networks, this will enable us to test the system with either one power supply or two separate supplies for the routing network performance.

5.5.2 The Configuration Control Memory

Memories used in the GALSA system for storing configuration control data are called configuration control memory (CCM). N_{CCM}^{RC} is the number of CCM bits required to configure a Routing Cell.

A full detailed analysis and comparison of dynamic and static RAM designs can be found in [58]. With conventional CMOS technologies and our particular configuration requirements, a six transistor static RAM, as shown in figure 5–25, has been selected as a CCM in our GALSA implementation.

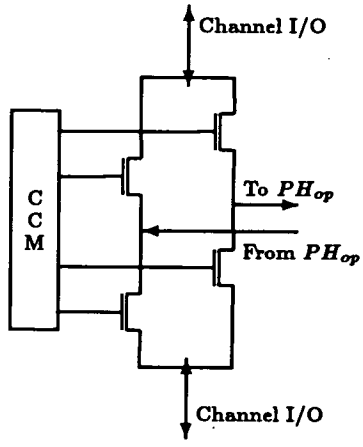


Figure 5–26: A switch unit with 4 bits CCM and 4 NMOS pass transistors

5.5.3 The Routing Cell

The basic element repeatedly used in an RC is a switch unit. As shown in figure 4–5, a switch unit is a four terminal device which can internally route data from one terminal to another terminal. The data routing is controlled by a CCM.

There are different ways available to implement the switch control. For example, central storage can be used for each RC and logic can be used to decode the actual controlling signal to each switch element. This method can reduce the number of CCM bits, at the expense of decoding logic. Another way is to store a control bit for each switch directly in a CCM. Thus a total of 4 bits of memory is needed in one switch unit. This direct switch control can retain the regularity of the RC and simplify the RC design, thus this control structure is used in the routing network. Figure 5–26 shows the design of the switch unit. The total number of direct control CCM bits required in an RC N_{CCM}^{RC} can be calculated from equation 5.7.

$$N_{CCM}^{RC} = 4 \times N_{chnl} + 2 \tag{5.7}$$

where N_{chnl} is the total number of routing channels to the RC and the extra 2 bits are used to control the two cross points between the two vertical and horizontal channels. Because 3 vertical and 2 horizontal routing channels are selected for each

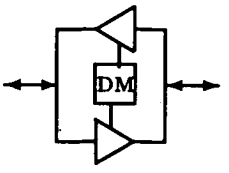


Figure 5–27: A bi-directional channel buffer

row and column of PH_{op}s, as illustrated in section 4.4.2, the total number of CCM bits in an RC is 22. For each channel, there are three switch units: one for data, two for request and acknowledge, which share one 4-bit CCM. Thus there are 15 switch units in an RC.

5.5.4 Routing Channel Buffers

Because of the delay and waveform distortions caused by the RCs, buffers are used in the routing channels to recover signal waveforms and add driving power to signals. Since signals may flow in both directions in a channel path depending on the mapping of an algorithm, bi-directional buffers are used in each routing path with every RC. The schematic of a bi-directional buffer is shown in figure 5–27. There are two tri-state buffers in the figure. The DM is 1 bit CCM which always sets one buffer ON and one OFF.

5.6 A GALSA System

The core of a GALSA array chip is formed by duplicating an array element which consists of a Routing Cell with channel buffers, a PH_{op} and a block of CCM, in the X and Y dimension. All ports of this array element will be properly positioned on the boundaries of a square area so that an $N \times N$ array core layout can be simply constructed by abutting this element layout in rows and columns. Other major considerations for the chip level implementation are peripheral control circuits around the core of the array. These peripheral circuits have three major functions:

distributing a global clock for each PH_{op} local use, preloading configuration bit streams into CCMs and ECRs, and providing I/O interface on the boundaries of a chip.

The clock distribution network is a tree structure with an external clock $GLCK$ at the root. This $GCLK$ drives a clock buffer at each row in the core array through one or two levels of buffers depending on the number of columns in the array. This row clock buffer then drives N clock buffers to N PH_{op}s in a row.

5.6.1 The Pre-loading Circuits

The operation of a GALSA system runs in three phases. A configuration phase must be performed to download network configuration and ECR bit streams for a particular algorithm once the system is powered on. The configuration phase is also called algorithm embedding phase for this reason. After this configuration phase, the system is initialised and then runs in the normal operation mode for the embedded algorithm.

In the configuration phase, the GALSA system is simply treated as a memory array. The way to write this "memory" array is similar to writing a word to a memory. For each write step, a row of CCMs is selected and written simultaneously. CCMs and ECRs are arranged in 2 columns in each array element, therefore, there are $2N$ bits in a "memory" row for an array with N columns. This requires $2N$ pins for downloading a sequence of configuration bit stream words. A column shifter which has the same number of stages as the total number of bits in a "memory" column is used to generate a sequence of row write control signals for "memory" writing. This shifter is controlled by a clock which also controls the flow of input configuration bit stream words so that the write of a "memory" row is lock-stepped with the configuration word flow. Therefore, downloading of configuration words is in synchronous mode in the configuration phase. This does not impose any difficulties in design because there are no complex data depen-

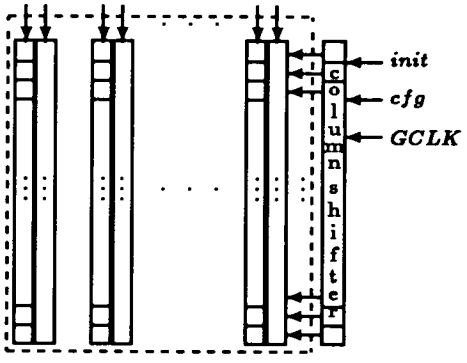


Figure 5–28: A configuration preloading structure

dencies with rows of write lines and two vertical clock lines which can be easily distributed to the column shifter. The clock does not need to run at a very fast speed. This will ensure that each write signal is enabled at the right time for an input configuration word. The clock input is shared with the clock input used in the GALS operation phase. Figure 5–28 shows this configuration preloading "memory" structure.

To save on the number of configuration word pins, an extra synchronous serial-in parallel-out row shift register of length equal to that of a "memory" word can be used. This shifter is placed on the top edge of the array. Only one configuration pin, which is the input to the row shifter, is required in this scheme. A sequence of configuration bits of a word length is first shifted into the row shifter, then a write line is enabled and the whole content of the row shifter is copied into a row of the "memory". This downloading procedure is repeated for all the rows to complete the configuration phase. An N-bit counter is required to generate a modulo-N control to the column shifter for the row writing signals.

There are two ways to to retain a system configuration. The easiest way is to keep the system power on once it is configured. The other way is to duplicate configuration vectors in an EEPROM which is connected to a system. Every time the system is powered on, the configuration vectors are automatically downloaded into the system from the EEPROM.

5.6.2 GALS Array I/O Interface

The GALSA system is a pad-limited architecture, namely the number of pads may determine the final size of a chip. Therefore slim pads are used for the pad ring. Multiplexers are also used at the ends of horizontal and vertical channels to multiplex the three vertical channels to one I/O pad group (3 pins), and also the two horizontal channels into one I/O pad group (3 pin). This may reduce the routability on the boundaries of a chip. However, efforts can be made to map sub-DFGs of an algorithm with less I/O requirements into a chip to avoid the I/O congestion on the boundaries of the chip. Figure 5–29 depicts the top level schematic of a 4×4 GALSA system with 8 configuration bit stream downloading pins.

5.7 Testability

It can be seen from the definition of the GALSA architecture that a GALS array can be tested in two phases. The test of the routing network is quite straightforward. First the conduction and switch ON function of each channel and switch are tested, then the independence of each channel and switch OFF function of each switch are tested. This will also test the function of CCM blocks. Once the routing network is tested, each PH_{op} can be tested in turn. The data transfer interface can be tested by setting all PH_{op}s to the merge function. The next step is to test all the primitive functions of each PH_{op}. This GALS array system has some degree of graceful degradation fault-tolerance on the array size. Malfunctioning PH_{op}s can be marked in a file and are avoided in the algorithm mapping procedure. As far as the routing network is concerned, some open connection faults can also be marked in the same file and avoided in algorithm mapping. However, if there are too many faults in the routing network, in particular a fault cluster

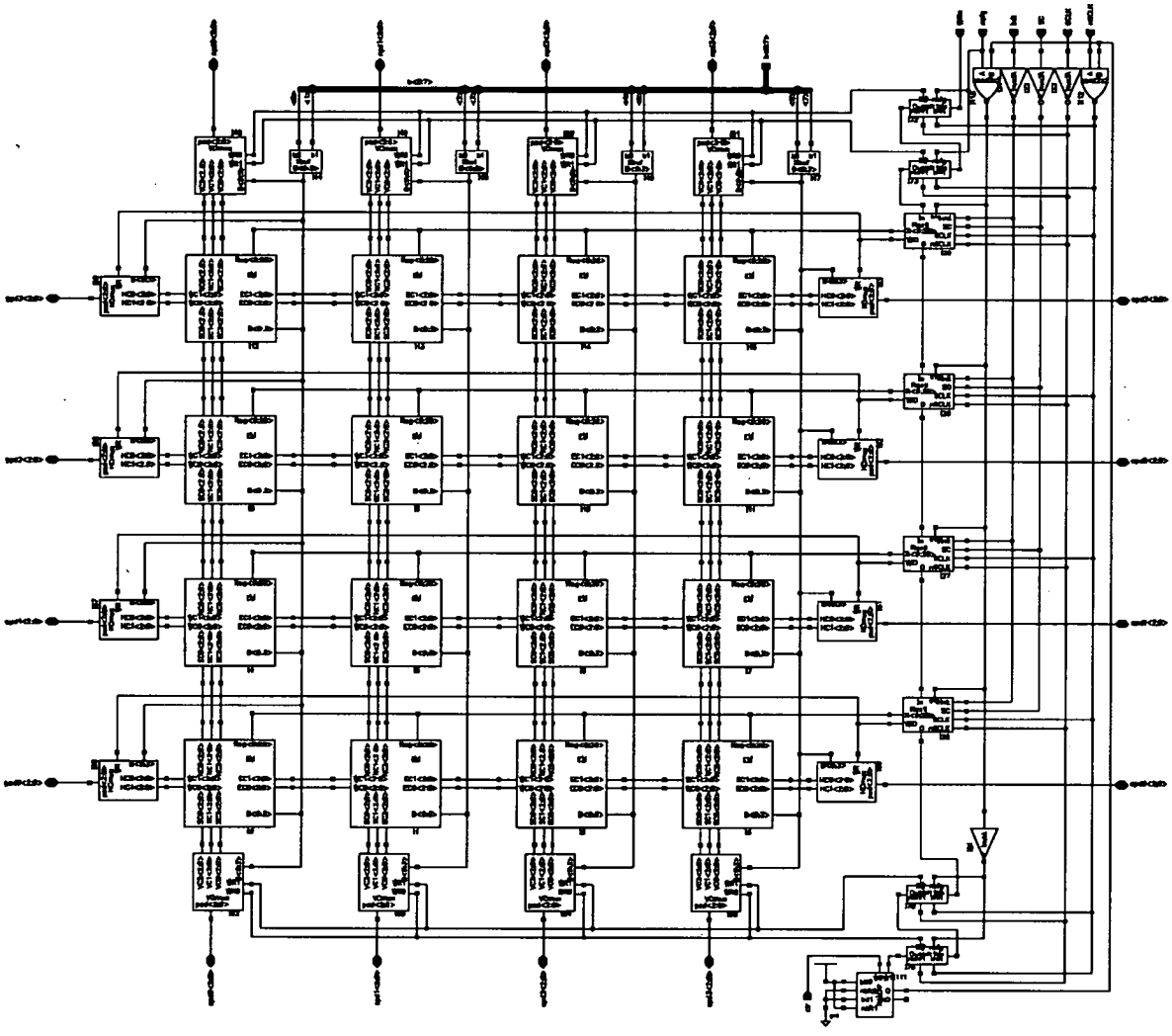


Figure 5–29: The schematic of a 4 × 4 GALS array

which cuts a whole section of an array, or some short circuit faults in a chip, the chip is rendered unusable.

5.8 Summary

The design of a configurable GALSA system is described in this chapter. One of the most important design blocks implemented is an event-driven GALS data transfer interface. A novel tri-state register and new synchronisation scheme for the GALS interface are presented. These designs are carefully analysed and proved working with simulated results by Hspice. The GALS approach presented here can also be extended to the design of general GALS systems. The design details of a programmable hardware operator (PH_{op}) and the routing network for the GALSA system architecture established in the preceding chapters are also elaborated in this chapter. In the next chapter, some simulation results on the performance of these designs will be presented and some algorithm mapping examples will be described. A performance comparison with some existing systems will also be made.

Chapter 6

Example Algorithms and Simulation Results

Intensive simulations with the Hspice circuit simulator on each individual design block described in chapter 5 were carried out to ensure that they function correctly, and to obtain typical timing characteristics of these elements. It is not easy to simulate an entire blank GALSA system, so we thoroughly tested the configuration vector pre-loading logic design, the routing network and a complete array element module. Several example application algorithms are studied and mapped into the GALSA system. Complete system level simulations were carried out for the configured GALSA systems with these embedded algorithms.

6.1 Typical Timing Characteristics

This section contains the typical timing characteristics of the key components designed for the GALSA system. Each component is analysed with the Hspice simulator under typical operating conditions which are described next. The components characterised are: the tri-state register, the input and output guard, the event-driven data transfer interface, the synchroniser, the transmission gate

	Q load	DV_R load	$\overline{R}^{T-} \rightarrow DV_R^{T-}$	$\overline{R}^{T+} \rightarrow DV_R^{T+}$
Q^{T+}	2ns	2ns	0.95ns	1.04ns
Q^{T-}	2ns	2ns	0.70ns	0.79ns

Table 6–1: The tri-state register timing

adder/subtractor, multiplexers, the Routing Cell, and the configuration bit-stream preloading circuit.

6.1.1 Simulation and Measurement Conditions

All Spice device models are taken from Mietec 2μm, double poly, double metal, N-well CMOS process. All circuits are simulated with $5V$ Vdd and $0V$ Vss at $27°C$.

All time measurements given in the following tables are in nanoseconds and the load at an output is the number of standard inverters unless otherwise stated. Input stimuli ramp $0 - 100\%$ in $1ns$. The propagation delay is measured from the 50% point of the input to the 50% point on the output. The rise/fall time is measured between the 10% and 90% value of the output.

6.1.2 The tri-state register and the GALS DTI

There are five signals in a tri-state register: D_{in}, WEN, DV_R, $\overline{R}$, and Q. Table 6–1 lists the timing characteristics for $\overline{R}^{T-} \longrightarrow DV_R^{T-}$ when WEN^-, and $\overline{R}^{T+} \longrightarrow DV_R^{T+}$ when WEN^+ with valid data at D_{in}.

There are 4 signals: R_{in}, A_{out}, EVT_i, and DV_R, in an input guard, and R_{out}, A_{in}, EVT_o, and DV_M, in an output guard. The timing characteristics for $R_{in}^T/A_{in}^T \longrightarrow EVT_i^{T+}/EVT_o^{T+}$ and $DV_R^{T+}/DV_M^{T+} \longrightarrow A_{out}^T/R_{out}^T$ are given in table 6–2. A complete shortest input cycle for an input guard, $R_{in}^T \rightarrow EVT_i^{T+} \rightarrow$

IG	A_{out} load	$R_{in}^{T} \rightarrow EVT_{i}^{T+}$	$DV_{R}^{T+} \rightarrow A_{out}^{T}$
R_{in}^{T+}	2ns	0.52ns	0.98ns
R_{in}^{T-}	2ns	0.61ns	0.67ns
OG	R_{out} load	$DV_{M}^{T+} \rightarrow R_{out}^{T}$	$A_{in}^{T} \rightarrow EVT_{o}^{T+}$
A_{in}^{T+}	2ns	0.98ns	0.47ns
A_{in}^{T-}	2ns	0.93ns	0.56ns

Table 6–2: The Input Guard and Output Guard

$\overline{R}^{T+} \rightarrow DV_{R}^{T+} \rightarrow A_{out}^{T}$, as illustrated in figure 5–10, is determined by the slowest $EVT_{i}^{T+} \rightarrow DV_{R}^{T+}$ when DV_{R}^{T-} is before EVT_{i}^{T+}. This is measured as $1.71ns$, therefore, $T_{min}^{IG} = 0.61 + 1.71 + 0.98 = 3.4ns$. A shortest output reset cycle for an output guard is $A_{in}^{T} \rightarrow EVT_{o}^{T+} \rightarrow \overline{R}^{T-} \rightarrow (DV_{M}^{T-} = DV_{R}^{T-} \cdot DV_{F}^{T-}) \rightarrow EVT_{o}^{T-}$. It takes one NMOS transistor delay, $0.21ns$, for $EVT_{o}^{T+} \rightarrow \overline{R}^{T-}$, and two AND gates delay, $0.62ns$, for $(DV_{R}^{T-} \cdot DV_{F}^{T-}) \rightarrow DV_{M}^{T-}$. Thus, we have $T_{min}^{OG} = 0.56+0.21+0.95+0.62 = 2.34ns$. A shortest $R_{in}^{T} \rightarrow DV_{R}^{T+} \rightarrow R_{out}^{T}$ path can be from a tri-state register to an output directly, $T_{in}^{out} = 0.61+1.71+0.98 = 3.30ns$, i.e. a data can pass through such a stage in only $3.30ns$. Because $DV_{R}^{T+} \rightarrow A_{out}^{T}$ is overlapped with $DV_{R}^{T+} \rightarrow R_{out}^{T}$ in time, an upper bound on a maximum sustainable data transfer rate can be estimated for the event-driven data transfer interface as: $1/(3.30 + 0.98 + 2.34) = 1/6.62ns = 151$MBit/s for the bit-serial data transfer. When this event-driven data transfer interface is used in a GALSA system, there are extra delay factors caused by the Routing Cell switches and multiplexers in PH_{op}s, and this will be configuration dependent.

The timing characteristics for the 5 input event-AND Muller C-element shown in figure 5–21 are in table 6–3. These parameters are the worst case figures after simulating different combinations of input event sequences.

	Output load	init	output
T_{out}^{T+}	2ns	0.23ns	1.54ns
T_{out}^{T-}	2ns	0.21ns	1.43ns

Table 6–3: Delays in the 5 input Muller C-element

TGadder	S load	C_{out} load	S	C_{out}
add	2ns	2ns	1.25ns	1.25ns
sub	2ns	2ns	1.32ns	1.32ns
MUXs	Output load	2-to-1	3-to-1	6-to-1
rising	2ns	0.29ns	0.61ns	0.88ns
falling	2ns	0.17ns	0.29ns	0.52ns

Table 6–4: Delays in the transmission gate adder and multiplexers

6.1.3 The Transmission Gate Adder and Multiplexers

There are three data inputs A, B, C_{in} and two outputs S, C_{out} in the transmission gate adder. The add/sub control is static before any inputs are applied to the adder. There are eight possible input combination situations which are all simulated for both *add* and *sub* function. Table 6–4 gives the worst case S and C_{out} output delays obtained from the simulation. This result conforms with the analysis of equal S and C_{out} delay with the transmission gate adder.

There are three types of multiplexers used in the PH_{op} design: 2-to-1, 3-to-1 and 6-to-1. The delays associated with these multiplexers are given in table 6–4.

From table 6–4, an estimation can be made on an upper bound for the local clock speed. This is 2 times the worst case multiplexers' delay plus the worst case adder delay. Allowing 30% process variations, we get an upper bound for the local clock as $210MHz$.

$5V$	Output load	NS/WE	NW/SE	NE	SW	BDCbuffer
rising	1.5ns	0.94ns	1.10ns	2.18ns	0.43ns	0.71ns
falling	1.5ns	0.67ns	0.85ns	1.92ns	0.23ns	0.68ns
$5V/7V$						
rising	1.5ns	0.72ns	0.89ns	1.96ns	0.31ns	0.64ns
falling	1.5ns	0.61ns	0.78ns	1.89ns	0.20ns	0.62ns

Table 6–5: Delays in a Routing Cell and bi-directional channel buffer

6.1.4 The Routing Cell and Channel Buffer

Because there is no direct interaction between V channels or H channels, we only give the delay factors associated with one V-channel (vh0) and one H-channel (hch0). The other channels have the same delay factors. There are four typical channel delays in the vch0 and hch0 routing: North-South (NS) or West-East (WE) involving only one switch unit of figure 5–26, NW/SE involving one switch unit and one pass transistor, NE involving two switch units and one pass transistor, SW involving only one pass transistor. The delays for these channel routing patterns are given in table 6–5. The table lists both delays when one $5V$ power supply is used and a separate $7V$ supply is used for the CCM. Each input is driven by a bi-directional channel buffer of figure 5–27, and each output drives another bi-directional channel buffer.

6.1.5 Array Element Test

It is important to test a GALSA array element as an integrated module. An array element is formed by connecting one PH_{pop}, as shown in figure 5–19, to one event-driven GALS DTI, one Routing Cell and a CCM module. The CCM module sets the Routing Cell, the I/O ports and function of the PH_{pop}.

The testing strategy is to simulate one such array element with each function op-code defined in table 5–2 one by one. In each op-code simulation, all possible input vectors to the function are tested and each input vector comes from a different set of input ports in the Routing Cell, and the corresponding outputs to a different set of output ports. For instance, there are eight possible input vectors to a full adder function. One of the first five possible A values is selected from vch0, vch1, vhc2, hch0, hch1 in turn to test the A input multiplexer function thoroughly. Similar I/O selections are also applied to B/C_{in} inputs and S/C_{out} outputs. When there are less than five possible input vectors, for example a two input Boolean function has only four possible input vectors, some randomly chosen input vectors are used to make up the five input test vectors. Therefore, there are five different CCM settings to simulate for one op-code function. The last CCM setting is also used for other input vectors after the first five. The A_{out} is looped back to R_{in}, and R_{out} is looped back to A_{in} in the GALS DTI in simulations.

No function errors were found with this intensive test on the schematic design of this array element, so we believe that the element functions correctly. The synchronisation mechanism in the GALS DTI also undergoes a test each time an input vector is applied. No metastable state is observed from these simulations.

6.1.6 Configuration Test

As illustrated in section 5.6.1, a sequence of configuration words is loaded into each row of CCMs in the core array of a GALSA similar to writing a memory array. The write signal to each CCM row is generated from a column shifter which is controlled by a configuration clock. Because the configuration phase runs in synchronous mode, we must test that the column shifter can generate a correct sequence of write signals, and all CCM rows can be correctly written using a proper configuration clock speed.

A long column shifter is simulated first to obtain a correct sequence of CCM row write signals. Then we simulated the configuration phase in a 4×4 GALSA by downloading two sequences of test configuration words. Each CCM bit is complemented in these two sets of configuration data to test all CCM bits. After the first sequence of configuration data is loaded, a reverse procedure is performed to read back the sequence of configuration data which is compared with the original one. This is repeated for the second sequence of complementary configuration data. Because both read-back data did not produce any differences from their original ones, this test indicates that the CCM array and the configuration pre-loading structure work correctly. The maximum configuration clock frequency that can ensure correct data loading to the CCM array depends on the size of the GALSA. Because all switches are statically driven by CCMs and a configuration phase is always carried out well in advance, the speed of a configuration process is not a particularly important factor as long as it is reasonably quick. Therefore, all transistors used in a CCM bit are the smallest allowed. For the 4×4 GALSA array, the simulation shows that configuration data can be loaded safely with a $20MHz$ configuration clock. It only takes approximately $6\mu s$ to configure a 4×4 array with the $20MHz$ configuration clock. We estimate that a 100×100 array can be safely configured, with a $1MHz$ configuration clock, within just $4ms$.

To test the connectivity of the channels and Routing Cells, we simulated the system with a configuration where the I/Os of all the PH_{op}s to Routing Cells are disabled and all the V and H channels are set as N $\rightarrow$ S and W $\rightarrow$ E conducting. A 0101 test vector is input into each channel after this configuration. This step is repeated for S $\rightarrow$ N and E $\rightarrow$ W channel settings so that the bi-directional channel buffers and Routing Cells are fully tested. The same 0101 output vector is observed at all channel outputs in all cases. This means the connections and the bi-directional channel buffers in each channel work properly.

6.2 A 4×4 Multiplier in a GALSA

This section describes an example of embedding a 4×4 bits integer multiplication function into a GALSA array.

6.2.1 Integer Multiplication

Suppose there are two integers X, Y,

$$X = \sum_{i=0}^{m-1} X_i \cdot 2^i \tag{6.1}$$

$$Y = \sum_{j=0}^{n-1} Y_j \cdot 2^j \tag{6.2}$$

The product of the X and Y will be

$$P = X \cdot Y \tag{6.3}$$

$$= (\sum_{i=0}^{m-1} X_i \cdot 2^i) \cdot (\sum_{j=0}^{n-1} Y_j \cdot 2^j) \tag{6.4}$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X_i \cdot Y_j) \cdot 2^{i+j} \tag{6.5}$$

$$= \sum_{k=0}^{m+n-1} P_k \cdot 2^k \tag{6.6}$$

where $P_k = \sum_{i,j} X_i \cdot Y_j$ for all (i, j) pairs that meet $i + j = k$. For a 4×4 multiplication, $m = n = 4$. Table 6–6 lists all the partial products generated. From this table it can be found that such a multiplication can be directly implemented as an array multiplier. A straightforward carry-save array multiplier is depicted in figure 6–2. Figure 6–1 shows the multiplier cell which is used in the array multiplier. One particular characteristic of this multiplier is that carries on each row are not added to the partial products of that row, instead carries are added to the partial products of the next row. This structure automatically eliminates the carry propagation delays in partial products generation apart from the last row.

				X_3	X_2	X_1	X_0
				Y_3	Y_2	Y_1	Y_0
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1	
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2		
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Table 6–6: A 4×4 integer multiplication

6.2.2 Embedding the 4×4 Array Multiplier into a GALSA

Generally speaking, an $n \times n$ array multiplier requires n^2 AND functions, $n(n-2)$ full adders, and n half adders. The carry to each partial product is added to it with a delay of one step, but carries have to be added at the same time when the final product is calculated. A carry-look-ahead technique can be used in this final product calculation. However, further analysis shows that for an $n \times n$ multiplication, only $(n-1)$ full adders are needed in the last stage which means that the worst case carry propagation delay is only n bits for the final $2 \times n$

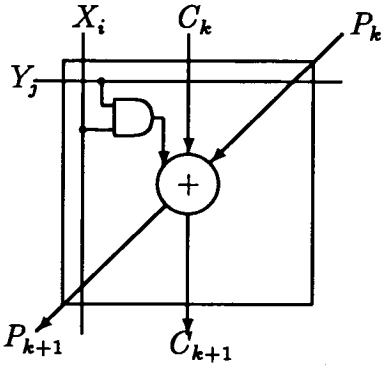


Figure 6–1: A multiplier cell for an array multiplier

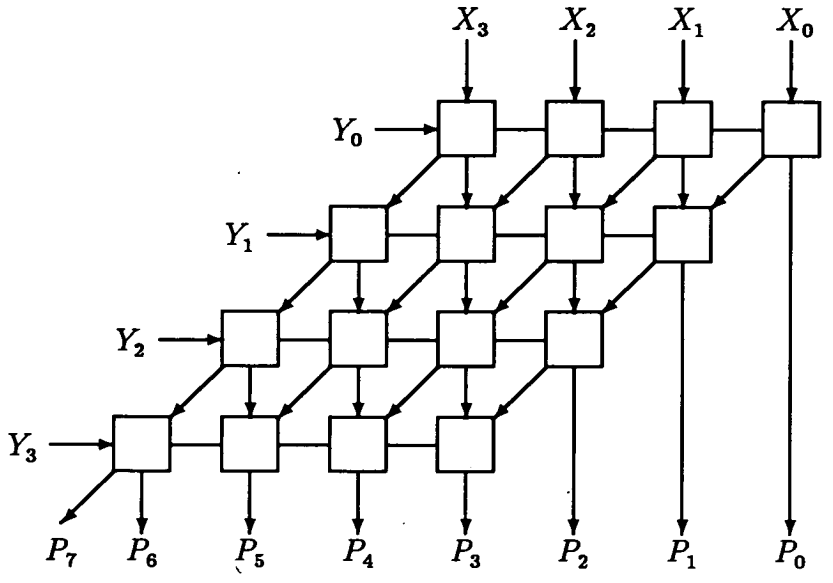


Figure 6–2: An array multiplier for a 4×4 multiplication

product terms if ripple-through carry adders are used. Therefore, three simple ripple-through adders are used in the final stage of the 4×4 multiplier.

The calculation of each product term is identified as a computation thread in an array multiplication function. Carries will not stall the generation of a current partial product in such a thread because they are generated one step earlier. The final row of ripple-through additions is another thread because the final higher n product terms are generated while carries are propagated through.

Four different macro-cells are composed for the array multiplier to match the elements used in figure 6–2 and the routing around these elements. Figure 6–3 depicts these four macro-cells. The BD macros are located on the top and left boundaries of the array to generate the first row and the most significant bits for partial products in table 6–6. The second row of partial products in table 6–6 is generated from the RHA macros. The third and forth row of partial products are generated from RFA macros. The RTCA macros calculate the final four higher bits for the final product.

Since routings are included in macro-cells, the 4×4 array multiplier is embedded into a GALSA by stacking and abutting these macros to form a rectangular

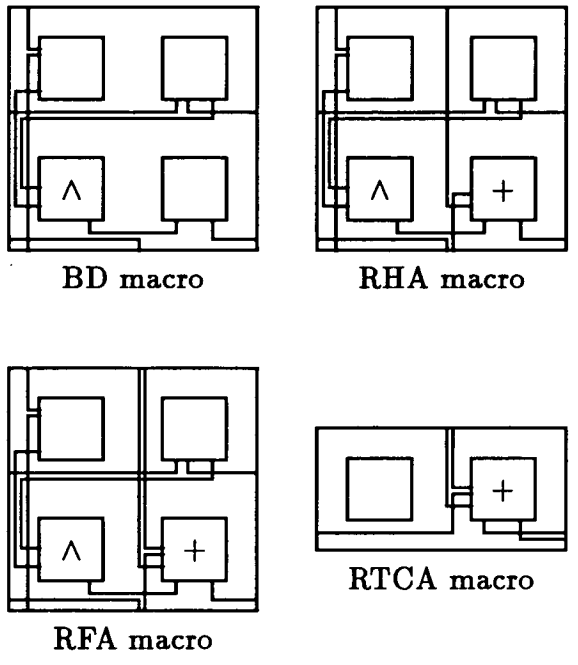


Figure 6–3: Macro-cells for an array multiplier

array as shown in figure 6–4. Macros on a diagonal (X_i, Y_j), where $i+j = k \in [0, 6]$, form a computation thread for a partial product term P_k. There are seven such diagonal computation threads. The last row of RTCA macros is another thread to compute the final P_4 to P_7 product terms.

There are 16 possible inputs for 4-bit integers X and Y. An average multiplication speed can be obtained if all 256 multiplications are simulated. This is too time consuming. Instead, five X values: 0011, 0110, 1001, 1100, 1111, and five Y values: 0010, 0101, 1011, 1101, 1111, are randomly chosen as inputs to simulate the embedded multiplication function. Because P_7 is the slowest product term from the multiplier, we measured the time taken to generate P_7 for each multiplication. Then an average $23.76ns$ multiplication time from these 25 multiplications is obtained.

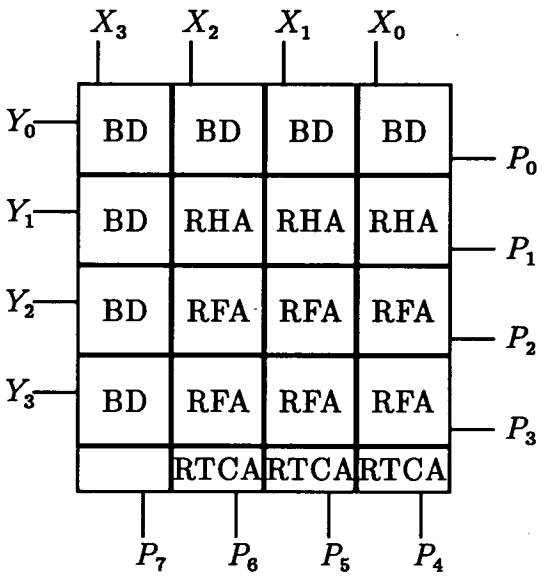


Figure 6–4: A 4×4 array multiplier in a GALSA array.

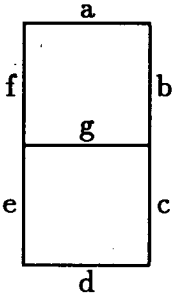


Figure 6–5: A seven segment display

6.3 A Seven Segment Display Decoder

In this algorithm mapping example, a seven segment display decoder function is mapped into a GALSA array.

A seven segment display as shown in figure 6–5 has seven independent segments with a unique label assigned to each segment. Each segment is controlled by a control signal. Different state combinations of the seven control signals will light the display as one of the digits of (0, 1, 2, ..., 9).

$X_3X_2X_1X_0$	a	b	c	d	e	f	g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
1010	×	×	×	×	×	×	×
1011	×	×	×	×	×	×	×
1100	×	×	×	×	×	×	×
1101	×	×	×	×	×	×	×
1110	×	×	×	×	×	×	×
1111	×	×	×	×	×	×	×

Table 6–7: Seven segment decoder truth table

There are 4 bits of input to a seven segment display decoder and 7 segment control signals as output from the decoder. The truth table for this decoder function is given in table 6–7.

Because of the large number of "don't care" states in table 6–7, the decoding function for each segment control signal can be substantially simplified. The optimised decoding functions for each segment control signal is listed in equation 6.7. Each function can be embedded into an independent computation thread. However, it can be seen from these functions that there are some terms which are shared by several control signals. Although these functions are independent from

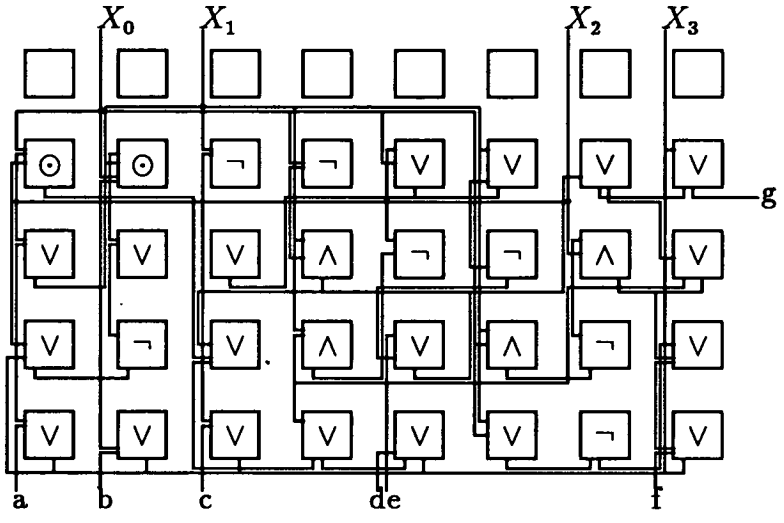


Figure 6–6: A seven segment display decoder in a GALSA array

each other, we tried to take advantage of these shared terms when mapping these equations into a GALSA array. While some array elements are saved as result of sharing some terms, more routings are needed. Figure 6–6 depicts the final mapping of the seven segment decoder into a GALSA array.

$$\begin{cases} a &= X_3 + X_1 + X_0 \odot X_2 \\ b &= X_3 + X_0 \odot X_1 + \overline{X_3} \cdot \overline{X_2} \\ c &= X_3 + X_2 + \overline{X_1} + X_0 \\ d &= X_3 + X_2 \odot X_0 + X_1 \cdot \overline{X_0} + \overline{X_2} \cdot X_1 \\ e &= X_1 \cdot \overline{X_0} + \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0} \\ f &= X_3 + \overline{X_1} \cdot \overline{X_0} + X_2 \cdot \overline{X_1 \cdot X_0} \\ g &= X_3 + X_1 \cdot \overline{X_0} + \overline{X_2} \cdot X_1 + X_2 \cdot \overline{X_1 \cdot X_0} \end{cases} \tag{6.7}$$

Ten possible inputs are all simulated with the embedded decoding functions. The average time to obtain all seven decoded control signals is $15.29ns$.

6.4 Evaluation of Polynomial Expressions

High speed evaluation of a large number of polynomial expressions has considerable application in the modelling and real-time display of objects in computer graphics. VLSI techniques have already been used for the design and implementation of frame buffers for computer graphics. A traditional frame buffer is usually a two dimensional memory array storing an array of picture elements (pixels) that are to be displayed on a bitmap screen. A frame buffer is one of the most important devices used in modern raster graphics displays. In this section, an algorithm for evaluation of polynomial expressions described in [93] is mapped into a GALSA system as a high performance frame buffer.

6.4.1 Display of Pixels for Different Objects

To display an object on a screen, the value of each pixel in the screen has to be calculated according to a certain function which is often a polynomial expression of various orders. Suppose a screen is represented as a fixed square grid of $(m+1) \times (m+1)$ pixels, a white line can be drawn by illuminating the pixels close to (X, Y) points in the screen coordinates which satisfy:

$$Ax + By + C = 0 \tag{6.8}$$

For a pixel at (x, y), $0 \leq x, y \leq m$, on the screen, the value $Ax + By + C$ is the perpendicular distance of the pixel to the line. An entire line can be drawn by highlighting all the pixels for which $|Ax + By + C| < W$, where $W \geq 1$ is a line width threshold. A half plane is turned on by illuminating all pixels for which $Ax + By + C < 0$. Generating arcs and circles involves evaluating quadratic polynomial expressions. A circle of radius r centered at point (a, b) can be drawn by illuminating all pixels at (x, y) for which $|r^2 - [(x-a)^2 + (y-b)^2]| \leq W$,

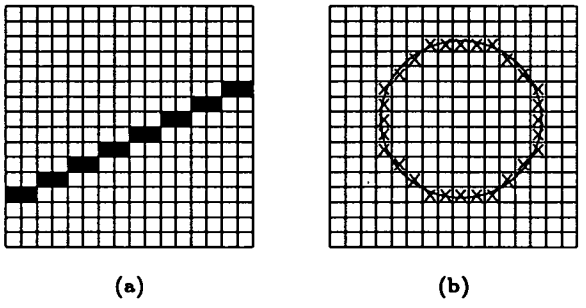


Figure 6–7: Pixel display: (a) a line; (b) a circle

where W is also a circle line width threshold value. Figure 6–7 shows a line and a circle drawn on a screen. If pixels satisfying $[r^2 - [(x-a)^2 + (y-b)^2] \geq 0$ are illuminated, a light dot of r radius centered at point (a, b) is turned on. For applications involving display of molecules, atoms are modelled as spheres, and the bonds between them as cylinders. The orthogonal projection of the atoms and their bonds reduces to evaluating quadratic expressions.

Thus, a large amount of calculation is required, for example to display a picture in a window area of 400 × 400 pixels. If these calculations were carried out in a CPU, which is also responsible for other computation tasks, the whole system speed and the graphics display would be very slow. Efforts have been made to design fast frame buffers with an array of identical simple processing elements [37, 93] to drive graphics displays directly.

6.4.2 Polynomials in Single Variable

A forward difference method is described in [93] to evaluate polynomials in single variable, and is extended to polynomials in two variables. The main property used by this method is that the nth difference for a polynomial of degree n is a constant.

Consider a polynomial of degree n given by

$$P_n(x) = \sum_{i=0}^{n} a_i x^i \tag{6.9}$$

The forward difference $P_n(x+1)-P_n(x)$ is a polynomial $P_{n-1}(x)$ in degree $(n-1)$. Applying the same reasoning repeatedly, we get

$$\begin{cases} P_n(x+1)-P_n(x) &= P_{n-1}(x) \\ P_{n-1}(x+1)-P_{n-1}(x) &= P_{n-2}(x) \\ P_{n-2}(x+1)-P_{n-2}(x) &= P_{n-3}(x) \\ \vdots & \\ P_1(x+1)-P_1(x) &= P_0(x) = constant \end{cases} \tag{6.10}$$

Thus, if all $P_i(0), 0 \leq i \leq n$, are known, the polynomial in equation 6.9 can be incrementally evaluated by equation 6.10 at all grid points $0 \leq x \leq m$. $[P_0(0), P_1(0), \cdots, P_n(0)]^T$ is defined as an initial vector. This initial vector can be obtained by using a linear transform from the coefficients of equation 6.9 as equation 6.11.

$$\begin{bmatrix} P_0(0) \\ P_1(0) \\ \vdots \\ P_n(0) \end{bmatrix} = \begin{bmatrix} & & & & w_{0,n} \\ & & & w_{1,n-1} & w_{1,n} \\ & & \cdot & & \vdots \\ w_{n,0} & w_{n,1} & \cdots & w_{n,n-1} & w_{n,n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \tag{6.11}$$

The elements $w_{i,j}, n \leq i+j \leq 2n$, in the lower right triangular transform matrix are integers which depend only on n so an initial vector can be precomputed once the coefficients of a polynomial in degree n are provided. A linear transform to obtain an initial vector for a general cubic polynomial $n = 3$ is shown in equation 6.12. The transform matrices for linear and quadratic polynomials are the lower left 2×2 and 3×3 sub-matrix in equation 6.12 respectively.

$$\begin{bmatrix} P_0(0) \\ P_1(0) \\ P_2(0) \\ P_3(0) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 6 \\ 0 & 0 & 2 & 6 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 6a_3 \\ 2a_2 + 6a_3 \\ a_1 + a_2 + a_3 \\ a_0 \end{bmatrix} \tag{6.12}$$

A sequential algorithm to compute a polynomial in n degree at grid points $0 \leq x \leq m$ is given below.

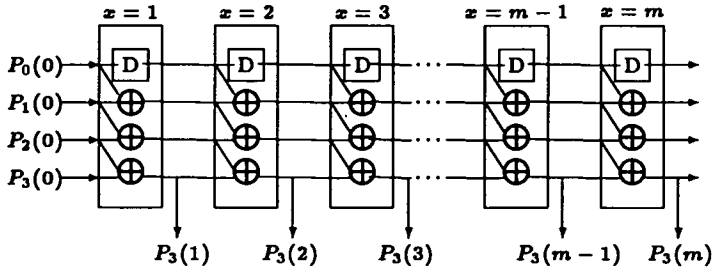


Figure 6–8: A linear array for evaluating cubic polynomials

```
Algorithm polynomial_in_x
          begin
step 1:   Compute the initial vector P_i(0), 0 ≤ i ≤ n;
step 2:   for x := 0 to m − 1 do
step 3:       {* compute the polynomial at x *}
              begin
              P_0(x + 1) := P_0(x);
step 4:       for i := 1 to n do
step 5:           P_i(x + 1) := P_{i−1}(x) + P_i(x);
              end;
          end;
```

From this sequential algorithm, the *for* loop can be easily unwound into an array organisation. This process also creates a pipeline structure without feedback so that the initial vectors of a large number of polynomials with different coefficients can be completely pipelined for evaluation. As a result, a linear array which evaluates cubic polynomials is shown in figure 6–8. The initial vector is calculated by a host computer and fed into the array. Because for an n degree polynomial, there are only $(n+1)$ elements in an initial vector, the calculation of these $(n+1)$ elements does not impose much load on the host computer. In the figure, D element is a delay unit.

6.4.3 Polynomials in Two Variables

A polynomial of degree n with two variables is given as

$$Q_n(x,y) = \sum_{0 \le j+j \le n} a_{i,j} x^i y^j \tag{6.13}$$

This $Q_n(x,y)$ can be treated as a single variable $P_n(x)$ of degree n

$$\begin{aligned} P_n(x) &= \textstyle\sum_{i=0}^{n} f_{n-i}(y) x^i \\ &= (a_{0,0} + a_{0,1}y + a_{0,2}y^2 + a_{0,3}y^3) \\ &\quad +(a_{1,0} + a_{1,1}y + a_{1,2}y^2)x \\ &\quad +(a_{2,0} + a_{2,1}y)x^2 + a_{3,0}x^3 \\ &= f_3(y) + f_2(y)x + f_1(y)x^2 + f_0(y)x^3 \end{aligned} \tag{6.14}$$

Each coefficient in equation 6.14 is a polynomial with single variable y of degree $n-i$. For a particular y, a set of coefficients can be obtained from $f_{n-i}(y)$ for a $P_n(x)$ at y. The initial vector for $P_n(x)$ can still be calculated from equation 6.12 by replacing a_i with $f_{n-i}(y)$

$$\begin{aligned} P_0(0) &= g_0(y) &&= 6a_{3,0} \\ P_1(0) &= g_1(y) &&= (2a_{2,0} + 6a_{3,0}) + 2a_{2,1}y \\ P_2(0) &= g_2(y) &&= (a_{1,0} + a_{2,0} + a_{3,0}) + (a_{1,1} + a_{2,1})y + a_{1,2}y^2 \\ P_3(0) &= g_3(y) &&= a_{0,0} + a_{0,1}y + a_{0,2}y^2 + a_{0,3}y^3 \end{aligned} \tag{6.15}$$

A separate y-array similar to the one shown in figure 6–8 to evaluate these $g_i(y)$ polynomials for $y \in [0,m]$ is required. Figure 6–9 shows one element of the y-array. The four outputs at each y point from this y-array, which are the initial vector for the x-array same as the one shown in figure 6–8 at y, are fed into the x-array for the evaluation of a $P_n(x)$ polynomial at that y. There are m x-arrays in total for the frame buffer. From equation 6.15 and 6.12, four initial vectors to the y-array are $[(6a_{0,3})\ (2a_{0,2}+6a_{0,3})\ (a_{0,1}+a_{0,2}+a_{0,3})\ (a_{0,0})]^T$ for $g_3(y)$, $[(2a_{1,2})\ (a_{1,1}+a_{2,1}+a_{1,2})\ (a_{1,0}+a_{2,0}+a_{3,0})]^T$ for $g_2(y)$, $[(2a_{2,1})\ (2a_{2,0}+6a_{3,0})]^T$ for $g_1(y)$, and $[6a_{3,0}]$ for $g_0(y)$.

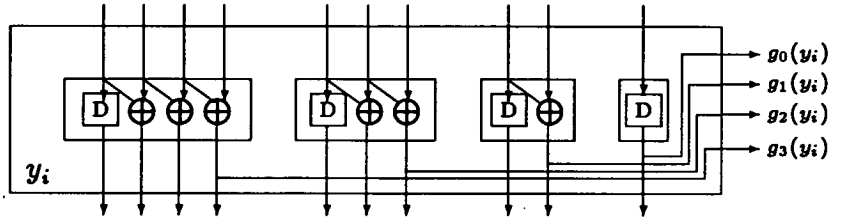


Figure 6–9: A y-array element at y_i

6.4.4 A Bit-Serial Frame Buffer

To reduce the cost, a bit-serial y-array and m bit-serial x-arrays are adopted in our design which matches nicely the structure of a GALSA system.

Referring back to figure 6–8, we can input each initial vector element $P_i(0)$ in bit-serial format from the lowest significant bit (LSB) to the most significant bit (MSB), use bit-serial adders with the carry output fed back to the carry input at each adder, and replace the delay element with a register. As a result, a bit-serial array for evaluating single variable cubic polynomials is obtained. One particular advantage of this bit-serial array organisation is that it can be used for any polynomials with different coefficients and variable word length. A complete bit-serial frame buffer organisation is shown in figure 6–10 with one y-array and three x-arrays as an example. The initial vectors are input into the y-array in bit-serial format.

6.4.5 Embedding the Frame Buffer into a GALSA

As is clearly shown in figure 6–10 for the bit-serial frame buffer, the y-array can be extracted as one vertical y-thread and each horizontal x-array can be extracted as a horizontal x-thread. The activation of the y-thread is by the input of initial vectors to the y-array. An x-thread is activated once the initial vector generated by the corresponding stage in the y-thread is available to the x-thread.

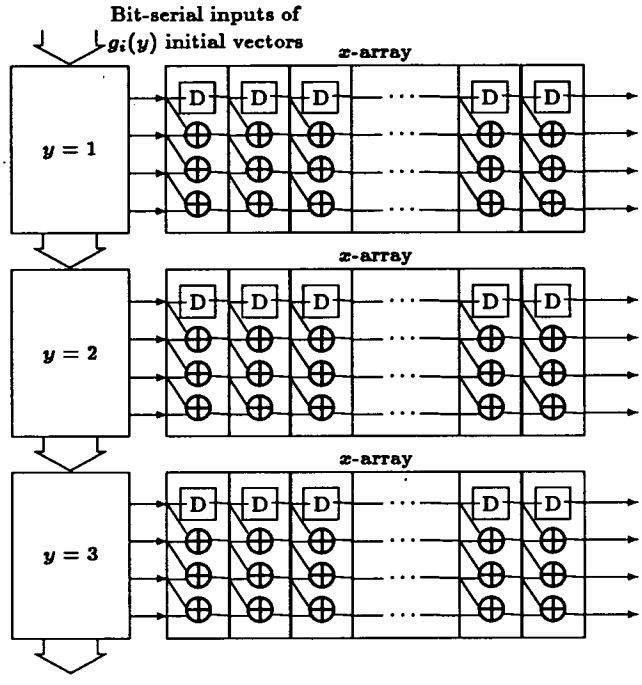


Figure 6–10: A bit-serial frame buffer processing array

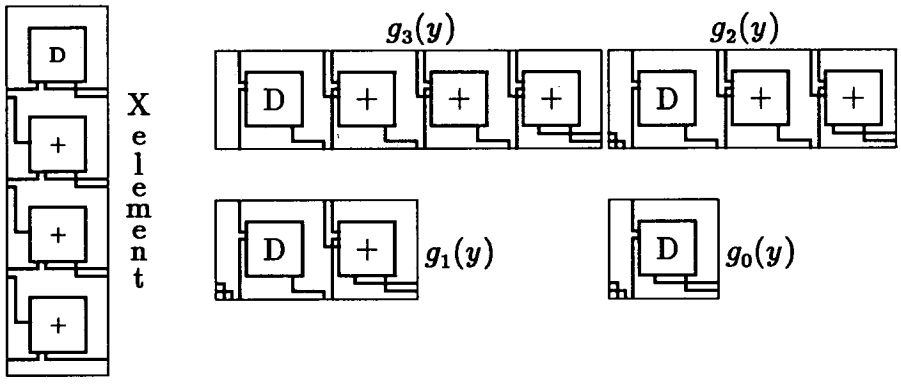


Figure 6–11: Macro cells for an X-element and $g_i(y)$

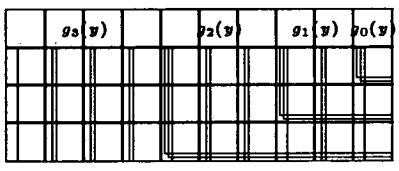


Figure 6–12: A Y-element for the y-thread

It is a fairly straightforward procedure to map an x-thread and a y-thread into a GALSA. Firstly, a macro cell called X-element for one stage in an x-thread can be generated as shown in figure 6–11, then this X-element is concatenated horizontally to form an x-thread. Secondly, four macro cells for each $g_i(y), i = 0, 1, 2, 3$ are generated for one stage in the y-thread. Figure 6–11 also depicts the four macro cells for $g_i(y)$. A Y-element as shown in figure 6–12 is obtained by concatenating these four cells horizontally with some additional routings. A y-thread is formed by concatenating this Y-element vertically. An entire frame buffer is embedded into a GALSA system by attaching an x-thread to the right of each Y-element in the y-thread.

It is too time consuming to simulate a real size, such as 400 × 400, embedded polynomial evaluation frame buffer. A much smaller 10 × 10 frame buffer was simulated instead. The upper left corner is the origin of the array coordinate $(x, y) = (0, 0)$. The first bit of the first pixel at (0,0) is generated just $14.70ns$ after initial vectors enter the y-thread. An average of $6.65ns$ is required to generate a bit for the pixel at $(x+1, 0)$ from the pixel at $(x, 0)$. It takes $117.2ns$ and $176.65ns$ to get an 8-bit pixel at $(0, 0)$ and $(9, 0)$ respectively. An average of $18.87ns$ was measured to obtain the first bit of the pixel at $(0, y + 1)$ after the first bit of the pixel at $(0, y)$ was generated. It takes $371.78ns$ to get the last bit of the pixel at (9,9).

6.5 Comparisons

We compared the performance of the 4 × 4 multiplier and seven segment display decoder embedded into the GALSA with Xilinx FPGA and CAL implementations. Table 6–8 compares the average performance on the same set of inputs. Functions in Xilinx FPGAs are simulated. The figures for a CAL system is estimated from its cell delay characteristics. It is apparent that the CAL has the best performance

	GALSA	Xilinx	CAL
multiplier	23.76ns	21.52ns	24.40ns
decoder	15.29ns	11.41ns	10.10ns

Table 6–8: Multiplier and decoder comparison

600×600	GALSA	MPP	AAP	RAP
Pipeline Latency	$78\mu s$	$240\mu s$	$120\mu s$	$180\mu s$
Max Refresh Rate	$0.13\mu s$	$1.6\mu s$	$0.8\mu s$	$0.3\mu s$

Table 6–9: Polynomial evaluation performance comparison

for the decoder function. This is because it is very good at bit-level combinational logic functions. Although the Xilinx FPGA has slightly better performance than our GALSA in this particular 4×4 multiplication case, we anticipate that our GALSA will perform better than the Xilinx FPGA on average when real long word multiplications, for example 512 bits, are carried out. The GALSA system is therefore not suitable for embedding combinational logic functions because of low performance and the high hardware cost compared with the CAL system.

We extended the simulation results for the polynomial evaluation in the 10×10 frame buffer to a 600×600 8-plane frame buffer. It is estimated that it would take approximately $78\mu s$ to fill up the last row of the x-thread to get a complete 8-bit pixel at $(599, 599)$. From this point, each pixel can be refreshed with a new pixel value as fast as $0.13\mu s$. We can compare the performance of this frame buffer with the estimated performance of the same algorithm on MPP, AAP and RAP. The estimation results were calculated according to the system clock speed and the number of clock cycles required to run the algorithm. MPP runs with a $10MHz$ clock. Both AAP and RAP run with a $20MHz$ clock. The results are given in table 6–9.

The frame buffer embedded into a GALSA system has an apparent advantage in performance when a large number of pixels are to be evaluated. This is mainly because of the asynchronous communication approach adopted at the system level. With such a large number of pixels in a large physical array, it is very difficult to run a faster clock for MPP, AAP or RAP system to improve their performance.

The Connection Machine and DDVA are too expensive to be used for this algorithm. On the other hand, they do not seem to be able to offer a substantial increase in performance for this algorithm.

6.6 Summary

In this chapter, we discussed the timing characteristics obtained from Hspice simulations on our designs. Three example algorithms were studied and embedded into our GALSA system to test the functionality of our design at system level. This also enables us to compare the performance with some existing systems for hardware algorithms. Our GALSA system has much better performance when the physical size of an array is very large where synchronous systems cannot operate with a fast global clock.

Chapter 7

Conclusions and Future Prospects

7.1 Overview of the Thesis

Chapter 1 introduced some fundamental concepts upon which our algorithmically configurable architecture principle is established. This chapter also introduced the basics on computation models, hardware algorithm models and system timing schemes from which a globally asynchronous locally synchronous configurable array system for algorithm embeddings was developed in the rest of this thesis.

In chapter 2, we took a detailed look at some typical array architectures developed by others for massively parallel processing. The analysis and comparison of these systems focused on the common characteristics and problems these systems have, and what are the problems and basic principles we need to solve in our approach to configurable hardware algorithms. Some practical implementation issues, such as interconnection structures, switching mechanisms, and processing element internal structures, were also discussed.

Chapter 3 presented and analysed in detail various computation, architecture and timing models. A multiple threads computation model for irregular algorithms, a configurable architecture of connected programmable hardware operators, and a globally asynchronous locally synchronous (GALS) system timing control strategy were proposed as the foundations for the development of a configurable GALS array system and for the efficient embedding of algorithms.

In chapter 4, some basic architecture constraints were discussed, and four typical circuit-switched interconnection topologies were classified and compared first. Then we proposed an array architecture composed of programmable hardware operators PH_{op} connected by a configurable interconnection network which is timed with the GALS system timing control method for the multiple threads computation model. The top-level system topology, the routing network structure, and the primitive functions of a PH_{op} for the proposed system were also presented.

The hierarchical design of a complete configurable GALS Array (GALSA) system was described in chapter 5. An important event-driven GALS data transfer interface was elaborated in great detail. A novel synchronisation scheme was proposed. Careful analysis and intensive simulations on this synchronisation scheme have been carried out to ensure an acceptable very low synchronisation failure rate. The design of the PH_{op} based on a transmission gate adder and multiplexers was presented. The system level routing network and the configuration data preloading structure were also illustrated.

All the system composition components were intensively simulated with the Hspice circuit simulator to ensure their correct functionality. In chapter 6, important timing characteristics obtained from Hspice simulations were given, and three algorithm examples: a 4×4 integer multiplication function, a seven segment display decoder, and a polynomial evaluation algorithm for a pixel frame buffer were studied and embedded into our GALSA system. The performance of these embedded algorithms based on the simulation results was compared with some existing systems. This comparison confirms our initial aim of this project: a high performance configurable system with GALS timing control was developed for hardware algorithms. The system can offer a much better performance than synchronous systems for data and computation intensive algorithms.

This concluding chapter clarifies the author's contributions and concludes the research presented in this thesis. Some related work to be done is illustrated.

Possible developments for configurable hardware algorithms in the future are also discussed.

In the appendix, more simulation results from the Hspice transient analysis on the synchroniser and the event-driven DTI design are given.

7.2 Achievements and the Author's Contributions

System timing problems attract more and more attention when the complexity of VLSI/ULSI systems is increasing so rapidly and the deep submicron technology is becoming mature. It is already a practical issue in deep sub-micron systems that the interconnection delays are one of the major factors that affects system performance or even the correct functioning. It will inevitably be an extremely difficult task to design such systems using a synchronous approach.

The author tried to look at the system timing problem by combining the advantages from both the synchronous and asynchronous design approach. At the same time, in the light of prospects in achieving high performance by parallel processing in hardware algorithms, the author adopted a systematic approach to develop a configurable GALS array system for algorithm embeddings following the proposal and establishment of an appropriate computation, architecture and system timing control model. The author's work on the event-driven GALS data transfer interface was also extended to a general GALS system design approach. The development of this configurable GALS array system enables us to perform some initial tests on the GALS design approach idea and do some comparisons of performance with other systems. The initial comparison is very encouraging with the embedded polynomial evaluation algorithm as a pixel frame buffer.

7.3 Other Work to Be Done

At the time this thesis is completed, a bit-serial GALSA system has been designed. However, there are still many ways in which the design itself can be improved. For instance, although the routing network can be easily tested, development of the testability for the overall system is still required.

One of the future tasks is to design the system with the partial Bit Parallel partial Word Parallel (BPWP) processing principle. For example, a PH_{op} should be capable of processing 4-bits of input operands at a time. Further engineering work is also required to get an actual working system on silicon. On the other hand, more testing algorithms need to be studied and embedded into the system to check the function and performance of the system further. Other work, such as automatic algorithm mapping and fault-tolerance, is equally important.

7.4 Automatic Configuration Vector Generation

Automatic design tools must be developed so as to get the full potential of a configurable system. It is by no means a simple task to transform an algorithm into a set of configuration vectors efficiently for a configurable system. Therefore an automatic algorithm mapping tool for configuration vector generation should be developed to ease the task. Two possible ways can be followed.

7.4.1 Automatic Data Flow Mapping

This method starts from the data flow description of an algorithm. An algorithm has to be either manually transformed into a data flow description or automatically transformed by other means.

Like other VLSI CAD systems, an automatic mapping system can take several forms of input, for example a front end data flow graph editor similar to the schematic capture front end in a CAD system or a data flow description similar to hardware descriptions to a VLSI CAD system.

A library of macro hardware operators can be developed from a set of primitive functions. A user can also set up his own macro operator library.

The automatic mapping system first decomposes the input data flow graph into a primitive data flow graph which only consists of connected primitive hardware operators and macro operators available from libraries. The dependencies and timing information can be extracted into a threads graph. Then the major task of the system is to map the edges and nodes in the primitive data flow graph and computation threads graph into the configurable array according to extracted threads, i.e. the allocation of operators and the routing of interconnections amongst these operators. Finally, a set of configuration vectors for Routing Cells and programming vectors for PH_{op}s can be generated from the mapping.

7.4.2 Automatic Algorithm Mapping

The ultimate solution is to describe an algorithm in a high level language such as C or C^{++}, or better still, a true data flow language. The automatic mapping system takes this high level algorithm description and transforms it into a primitive data flow description. From this point, the mapping method described in section 7.4.1 can be used to generate configuration and programming vectors.

7.5 Fault-Tolerance

Fault-tolerance is another important issue in massively parallel array architectures. There has been much research on fault-tolerance methodologies and schemes already carried out and published. Fault-tolerance for the GALSA architecture should be investigated in the future. The framework for this problem is outlined in this section.

Each fault-tolerance scheme usually assumes a fault model with some additional conditions before the scheme can be effectively applied. If a fault pattern is beyond the capability of the scheme, the fault pattern is said to be unrecoverable. It is a common practice to assume random faults instead of clustered faults. Another common assumption in most of the fault-tolerance schemes for array architectures is that faults most likely happen in processing elements. In the GALSA architecture, we need to consider the possibility of failure in both the hardware operators and the routing network. This is because more than two thirds of transistors in a Routing Cell are in the CCM which may have a higher failure possibility.

Generally speaking, there are three classes of methodology for fault-tolerance, i.e. redundancy, graceful degradation, and time sharing. Schemes based on the redundancy methodology use extra spare rows or columns of processing elements in an array, and try to replace the faulty processing elements in the array with spare processing elements so that the original physical size of the array can be restored. Rather than using physical redundancies, the time sharing methodology tries to restore an array logically equivalent to a physical array which has faulty processing elements, by time sharing a healthy processing element next to each faulty element. This methodology needs a complex swapping mechanism to make one element perform the role of two elements. The graceful degradation methodology adopts a completely different approach. Instead of trying to restore an array with faulty

elements to its original size, this method tries to establish a new largest possible rectangular sub-array out of the healthy elements from the physical array. The resultant sub-array will be smaller than its parent array but it is still usable.

The combination of redundancy and the graceful degradation methodology appears to be a very attractive fault-tolerance approach for the GALSA architecture under the multiple threads computation model. Still further, we do not even need to re-establish a sub-array of rectangular shape. This is because if a slightly larger physical GALSA array is used than the requirement of an application, as long as there are enough healthy routable PH_{op}s for the application in the array where faulty elements occur, it is not necessary to restore the array to its original physical size and shape. Routing resources can also be saved without re-establishing a rectangular sub-array.

7.6 Future Developments and Prospects

7.6.1 Taking Advantages of New Technologies

There are two trends in the development of microelectronic technologies. The evolution of conventional technologies is the way most of the commercial products take, that is, the continuous improvement of technologies which make it possible to integrate more and more devices of smaller and smaller feature sizes into silicon dies which are becoming increasingly larger. The Digital Alpha processor and the Intel Pentium processor are two such examples. The GALSA architecture can benefit from this evolution in two aspects, first, more PH_{op}s can be integrated; and second, the complexity of the routing network and PH_{op} can also be increased to be more powerful. However, the conventional approach eventually will approach the physical limits of silicon and conventional CMOS properties. Thus, the emerging revolutionary breakthroughs of new devices and materials are be-

coming more appealing in the longer term. Silicon MOSFETs, which change state in the nanosecond scale, are generally slow as switching devices. The GaAs metal semiconductor (GaAs MES) FET is capable of switching in less than $30ps$ and operating at frequencies in excess of $5GHz$. This is because electrons in GaAs have a significantly higher mobility than electrons in silicon. Once the GaAs material manufacturing technology is mature, GaAs integrated circuits will certainly play an important role. All current VLSI architectures will benefit from high speed GaAs technology. For the GALSA architecture, improvement in the switching speed in the routing network is very important. On the other hand, the GALSA architecture is readily designed to incorporate any new faster and better switching devices because of its GALS system timing scheme.

7.6.2 Multi-layer Metal and Three dimensional Structures

The GALSA architecture is very wire demanding. The two metal layer processing technology used in the design in this thesis cannot meet the GALSA requirement because a large amount of silicon area is wasted for routing connections. Therefore, a multi-layer metal processing technology is desirable so that the network routing layers can overlap active areas.

Many data flow graphs actually have three dimensional properties. Thus congestion in the routing network is unavoidable if these three dimensional graphs are to be mapped into a two dimensional array. Hence, the GALSA architecture should particularly benefit from three dimensional integration structures.

7.6.3 Wafer Scale Integrations

The most common way to manufacture VLSI chips is to put together a group of designs or the replication of a design in the form of a two dimensional array of dies on a silicon wafer. A test is performed on each die area after the wafer is

processed and faulty dies are marked. This array of dies is then cut into chips, and working chips are mounted and packed into packages.

The number of elements for an array on a die manufactured in such a way, which can range from a few to a couple of thousand depending on the size of the element, is very limited. In order to embed an application into such an array, it is often necessary to connect a set of such chips to form a larger array. Therefore, this die cutting and packaging procedure appears unnecessary for array architectures. One possible solution is to use Multi-Chip-Modules (MCM) by which multiple chips are mounted on a larger substrate and packed into one package. This technique can reduce the cost of packing individual chips and improve the communication speed among chips. However, the die cutting process is still required for MCMs. Wafer scale integration (WSI) technology is particularly attractive for array architectures. The idea of WSI is to make a complete system on an entire wafer; the wafer is packed in one package and used as one system. Thus, if an entire wafer were to be used to accommodate a complete array, the cost of putting a pad ring on each die, the wafer cutting, the multiple packaging of smaller chips and the extra connection delays between packages would be eliminated.

There are still many technical problems associated with WSI to be solved. For example, the power dissipation of an entire wafer tends to be very high, so the cooling of a whole wafer is very important and difficult. It is not an easy task to package a large wafer. A practical fault-tolerance technique must be applied to WSI because faults certainly will happen on some areas of a wafer. Current research has already made significant progress in WSI, but the three dimensional integration technology is still at an early stage of research.

The theories and design methodologies developed in this thesis can be easily adapted to WSI architecture. The major changes required will be in the peripheral and configuration vector preloading parts. The inclusion of fault-detection and fault-tolerance mechanisms is also needed.

7.7 Conclusions

A few years ago, the configurable logic methodology was still questioned by many people, now this methodology has been widely accepted and many commercial products on FPGAs and configurable logic arrays are already available. However, the asynchronous integrated circuit design methodology still needs more attention than it has so far received. System timing issues must be properly considered in VLSI/ULSI/WSI design, especially in large configuration systems. We believe that very large scale configurable logic is the area where asynchronous design methodology can be used as an effective system level timing control approach. The initial testing on the developed configurable GALSA system achieves a better average performance on large scale problems, such as polynomial evaluations as a frame buffer, than some other synchronous massively parallel processing systems.

But the configurable logic is by no means the only area where asynchronous design methodology can be applied. Two effects of the rapid advance of technology are that integrated systems are becoming ever more complex and more computing tasks are directly implemented in hardware. With this trend of continuous increase of system complexity and decrease of hardware costs, the asynchronous design methodology will play at least as important a role as the synchronous design framework, for example the GALS timing scheme proposed in this thesis.

Bibliography

[1] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[2] Francois Anceau. *The Architecture of Microprocessors.* Addison-Wesley, 1986.

[3] Y. Ansade, R. Cornu-emieux, B. Faure, and G. Mazare. WSI asynchronous cells network. In G. Saucier and J. Trilhe, editors, *Wafer Scale Integration*, pages 77–87, March 1986.

[4] D. B. Armstrong and A. D. Friedman. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Trans. on Computers*, c-18(12):–, December 1969.

[5] Arvind and Kim P. Gostelow. The U-interpreter. *Computer*, 15(2):42–49, February 1982.

[6] Katsuhiko Asada, Hiroadi Terada, Satoshi Matsumoto, Souichi Miyata, Hajime Asano, Hiroki Miura, Masahisa Shimizu, Shinji Komori, Takeshi Fukuhara, and Kenji Shima. Hardware structure of a one-chip data-driven processor: Q-p. In *IEEE Proc. 1987 Int'l Conf. on Parallel Processing*, pages 327–329, August 1987.

[7] K. E. Batcher. Design of a massively parallel processor. *IEEE Trans. on Computers*, c-29(9):836–840, September 1980.

[8] N. W. Bergmann. A case study of the FIRST silicon complier. In R. Bryant, editor, *3rd Caltech Conference on VLSI*, pages 413–430, 1983.

[9] Patrice Bertin, Didier Roncin, and Jean Vuillenmin. Introduction to Programmable Active Memory. Technical report, Digital Paris Research Laboratory, June 1989.

[10] Donald W. Blevins, Edward W. Davis, Robert A. Heaton, and John H. Reif. BLITZEN: A highly integrated massively parallel machine. *Journal of Parallel and Distributed Computing*, 8(2):150–160, February 1990.

[11] J. A. Brzozowski and J. C. Ebergen. Recent developments in the design of asynchronous circuits. Technical Report CS-89-18, Computer Science Department, University of Waterloo, 1989.

[12] Mengly Chean and Jose A. B. Fortes. A taxonomy of reconfiguration techniques for fault-tolerant processor arrays. *Computer*, 23(1):55–69, January 1990.

[13] Marina C. Chen. The generation of a class of multipliers: Synthesizing highly parallel algorithms in VLSI. *IEEE Trans. on Computers*, 37(3):329–338, March 1988.

[14] W. Chen. *A Reconfigurable Architecture for Very Large Scale Microelectronics Systems.* PhD thesis, Dept. of Electrical Engineering, University of Edinburgh, 1986.

[15] G. Chevalier and G. Saucier. A programmable switch matrix for the wafer scale integraton of a processor array. In Chris Jesshope and Will Moore, editors, *Wafer Scale Integration*, pages 92–100, July 1985.

[16] L. Ciminiera and A. Valenzano. Low cost serial multiplier for high-speed specialised processors. *IEE Proc.-E Computers and Digital Techniques*, 135(5):259–265, September 1988.

[17] Computer Science Division, EECS dept., University of California at Berkeley. *Berkeley 1986 VLSI Tools user's Manual*, 1986.

[18] Luigi Dadda. On serial-input multipliers for two's complement numbers. *IEEE Trans. on Computers*, 38(9):1341–1345, September 1989.

[19] William J. Dally. A high-performance VLSI quaternary serial multiplier. In *IEEE Int'l Conf. on Computer Design: VLSI in Computers and Processors*, pages 649–653, Oct. 1987.

[20] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *The 14th Annual Int'l Symp. on Computer Architecture*, pages 189–196, June 1987.

[21] Edward W. Davis and John H. Reif. Architecture and operation of BLITZEN processing element. In *Proc. Thrid Int'l Conference on Supercomputing*, pages 128–137, May 1988.

[22] Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley. *SPICE User's Guide.*

[23] Digital Equipment Corporation, Maynard, MA, U.S.A. *Alpha Architecture Handbook*, 1992.

[24] M. J. B. Duff. Review of the CLIP image processing system. In *Proc. AFIPS National Computer Conf.*, pages 1055–1060, June 1978.

[25] Khaled A. El-ayat, Abbas Gamal, Richard Guo, John Chang, Ricky K. H. Mak, Frederick Chiu, Esmat Z. Hamdy, John McCollum, and Amr Mohsen. A CMOS electrically configurable gate array. *IEEE J. of Solid-State Circuits*, 24(3):752–761, June 1989.

[26] Shinji Komori et al. An elastic pipeline mechanism by self-timed circuits. *IEEE J. of Solid-State Circuits*, 23(1):111–117, Feb. 1988.

[27] Shinji Komori et al. The Data-Driven microprocessor. *IEEE MICRO*, 9(3):45–58, June 1989.

[28] European Silicon Structure. *Solo1400 User Guide.*

[29] Richard A. Evans. A self-organizing, fault-tolerant, 2-dimensional array. In E. Hörbst, editor, *Proceedings of the IFIP, VLSI '85, VLSI design of digital systems*, pages 239–248. North-Holland, 1985.

[30] Allan L. Fisher. Memory and modularity in systolic array implementations. In *IEEE Proceedings of 1985 Int'l Conf. on Parallel processing*, pages 99–101, May 1985.

[31] Allan L. Fisher and H. T. Kung. Synchronizing large VLSI processor arrays. In *IEEE 10th Int'l Symposium on Computer Architecture*, pages 54–58, June 1983.

[32] Stuart Fiske and William J. Dally. The reconfigurable arithmetic processor. In *IEEE The 15th Annual Int'l Symp. on Computer Architecture*, pages 30–36, May 1988.

[33] T. J. Fountain. Towards CLIP6 – an extra dimension. In *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pages 25–30, 1981.

[34] T. J. Fountain. A survey of bit-serial array processor circuits. In M. J. B. Duff, editor, *Computing Structures for Image Processing*, pages 1–14, 1983.

[35] T. J. Fountain and V. Goetcherian. CLIP4 parallel processing system. *IEE Proc.-E Computers and Digital Techniques*, 127(5):219–224, September 1980.

[36] Jeff Fried, Elizabeth Daly, Ted Lyszczarz, and Michael Cooperman. A memory-controlled crosspoint switch using E-beam restructuring technology. *IEEE J. of Solid-State Circuits*, 24(1):183–187, February 1989.

[37] Henry Fuchs, John Poulton, Alan Paeth, and Alan Bell. Developing pixel-planes, a smart memory based raster graphics systems. In *1982 Conference on Advanced Research in VLSI, M.I.T.*, pages 137–146, January 1982.

[38] Steve Furber. Computing without clocks: Micropipelining the ARM processor. In *AMULET1 Modelling Workshop*, July 1994.

[39] Abbas El Gamal, Jonathan Greene, Justin Reyneri, Eric Rogoyski, Khaled A. El-ayat, and Amr Mohsen. An architecture for electrically configurable gate array. *IEEE J. of Solid-State Circuits*, 24(2):394–398, April 1989.

[40] Bo Gao and D. J. Rees. Communicating synchronous logic modules. In *21st Euromicro conference on Design of Hardware and Software Systems*, pages 708–715, September 1995.

[41] J.D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In *Proceedings of the IFIP Conference on Asynchronous Design Methodologies*, 1993.

[42] Ganesh Gopalahrishnan and Venkatesh Akella. VLSI asynchronous systems: specification and synthesis. *Microprocessors and Microsystems*, 16(10):517–527, October 1992.

[43] Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UU-CS-TR-90-016, Dept. of Computer Science, University of Utah, Oct. 1990.

[44] John B. Gosling. *Design of Arithmetic Units for Digital Computers.* The MacMilian Press Ltd, 1980.

[45] Jan Grinberg, Graham R. Nudd, and R. David Etchells. A cellular VLSI architecture. *Computer*, 17(1):69–81, January 1984.

[46] M. Hatamian and G. L. Cash. Parallel bit-level pipelined VLSI designs for high-speed signal processing. *Proceedings of the IEEE*, 75(9):1192–1202, September 1987.

[47] Robert A. (Fred) Heaton and Donald W. Blevins. BLITZEN: A VLSI array processing chip. In *IEEE Proc. Custom Integrated Circuits Conference*, pages 12.1.1–12.1.5, May 1989.

[48] Kye S. Hedlund. The design of a prototype WASP machine. In Chris Jesshope and Will Moore, editors, *Wafer Scale Integration*, pages 89–97, July 1985.

[49] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[50] Masaharu Hirayama. VLSI oriented asynchronous architecture. In *The 13th Int'l Symp. on Computer Architecture*, pages 290–296, June 1986.

[51] R. W. Hockney and C. R. Jesshope. *Parallel Computers – Architecture, Programming and Algorithms.* Adam Hilger Ltd., 1984.

[52] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2 – Architecture, Programming and Algorithms.* Adam Hilger Ltd., 1988.

[53] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Trans. on Computers*, c-31(12):1135–1141, Dec. 1982.

[54] Y. Hsia, G. Chang, and F.D. Erwin. Adaptive wafer scale integration. In *1979 Int'l Digest of Tech. Papers of the 11th Conf. on Solid State Devices*, pages 81–82, 1979.

[55] D. J. Hunt. *AMT DAP – a processor array in a workstation environment.* Active Memory Technology, Reading, Berks, UK, April 1989.

[56] Yasuo Ikawa, Kiyoshi Urui, Masashi Wada, Tomoji Takada, Masahiko Kawamura, Misao Miyata, Noboru Amano, and Tadashi Shibata. A one-day chip: An innovative IC construction approach using electrically reconfigurable logic VLSI with on-chip programmable interconnections. *IEEE J. of Solid-State Circuits*, sc-21(2):223–227, April 1986.

[57] G. Jacobs and R. W. Brodersen. Self-timed integrated circuits for digital signal processing applications. In *VLSI Signal Processing III*, pages 197–208. New York: IEEE Press, Nov. 1988.

[58] Thomas Andrew Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation.* PhD thesis, Dept. of Computer Science, University of Edinburgh, December 1989.

[59] Tom Kean and Genbao Feng. Configurable logic: An approach to the rapid implementation of ASIC's. Technical Report CSR-234-87, Dept. of Computer Science, Edinburgh University, June 1987.

[60] Anwar Khurshid and P. David Fisher. Algorithm implementation on reconfigurable mixed systolic arrays. In *Proceedings on 1985 Int'l Conf. on Parallel Processing*, pages 79–88, 1985.

[61] J. H. Kim and S. M. Reddy. On easily testable and reconfigurable two-dimensional systolic arrays. In *IEEE Proc. 1987 Int'l Conf. on Parallel Processing*, pages 101–109, 1987.

[62] Jun Hwan Kim and Sudhakar M. Reddy. On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement. *IEEE Trans. on Computers*, 38(4):515–525, April 1989.

[63] D. J. Kinniment and J. V. Woods. Synchronisation and arbitration circuits in digital systems. *IEE Proc.-E Computers and Digital Techniques*, 123(10):961–966, October 1976.

[64] Toshio Kondo, Takayoshi Nakashima, Makoto Aoki, and Tsuneta Sudo. An LSI adaptive array processor. *IEEE J. of Solid-State Circuits*, SC-18(2):147–156, April 1983.

[65] Toshio Kondo, Toshio Tsuchiya, Yoshihiro Kitamura, Yoshi Sugiyama, Takashi Kimura, and Takayoshi Nakashima. Pseudo MIMD array processor–AAP2. In *The 13th Annual Int'l Symp. on Computer Architecture*, June 1986.

[66] Israel Koren and Bilha Mendelson. A Data-Driven VLSI array for arbitrary algorithms. *Computer*, 21(8):30–43, October 1988.

[67] Israel Koren and Gabriel M. Silberman. A direct mapping of algorithms onto VLSI processor arrays based on the data-flow approach. In *IEEE Proceedings on 1983 Int'l Conf. on Parallel Processing*, pages 335–337, 1983.

[68] H. T. Kung. Let's design algorithms for VLSI systems. In *Proceedings of the Caltech Conf. on Very Large Scale Integration*, pages 65–90, Jan. 1979.

[69] H. T. Kung. Why systolic architectures. *Computer*, 15(1):37–46, January 1982.

[70] H. T. Kung. Memory requirements for balanced computer architectures. In *The 13th Annual Int'l Symposium on Computer Architecture*, pages 49–54, June 1986.

[71] H. T. Kung. Deadlock avoidance for systolic communication. In *The 15th Annual Int'l Symp. on Computer Architecture*, pages 252–260, May 1988.

[72] S. Y. Kung. On supercomputing with systolic/wavefront array processors. In *Proceedings IEEE*, pages 867–884, July 1984.

[73] S. Y. Kung, K.S. Arun, Ron J. Gal-ezer, and D.V. Bhaskar Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Trans. on Computers*, c-31(11):1054–1066, Nov. 1982.

[74] S. Y. Kung and R. J. Gal-Ezer. Synchronous versus asynchronous computation in very large scale integration array processors. In *SPIE, Real Time Signal Processing V*, pages 53–65, 1982.

[75] S. Y. Kung and R.J. Gal-Ezer. Wavefront array processor: Architecture, language and application. In *Conference on Advanced Research in VLSI, M.I.T.*, pages 4–19, January 1982.

[76] S. Y. Kung and S.C. Lo. Wavefront array processors – concept to implementation. *Computer*, 20(7):18–33, July 1987.

[77] Sun-Yuan Kung, Shiann-Ning Jean, and Chih-Wei Chang. Fault-tolerant array processors using single-track switches. *IEEE Trans. on Computers*, 38(4):501–514, April 1989.

[78] Bradley C. Kuszmaul and Jeff Fried. NAP No ALU Processor: The great communicator. *Journal of Parallel and Distributed Computing*, 8(2):169–179, February 1990.

[79] Myoung Sung Lee and Gideon Frieder. Massively fault-tolerant cellular array. In *IEEE Proceedings of 1986 Int'l Conf. on Parallel Processing*, pages 343–350, August 1986.

[80] Hungwen Li and Massimo Maresca. Polymorphic-Torus architecture for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3):233–243, March 1989.

[81] Hungwen Li and Massimo Maresca. Polymorphic-torus network. *IEEE Trans. on Computers*, 38(9):1345–1351, September 1989.

[82] Shih lien Lu. Implementation of iterative networks with CMOS differential logic. *IEEE J. of Solid-State Circuits*, 23(4):1013–1017, August 1988.

[83] W. Lim. Design methodology for stoppable clock systems. *IEE Proc.-E Computers and Digital Techniques*, 133(1):65–69, January 1986.

[84] Willie Y-P. Lim and Jr. Jerome R. Cox. Clocks and the performance of synchronisers. *IEE Proc.-E Computers and Digital Techniques*, 130(2):57–64, March 1983.

[85] Fabrizio Lombardi. Reconfiguration of hexagonal arrays by diagonal deletion. *Integration, the VLSI journal*, 7(6):263–290, 1988.

[86] Erl-Huei Lu, Lein Harn, Jau-Yien Lee, and Wen-Yih Hwang. A programmable VLSI architecture for computing multiplication and polynomial evaluation modulo a positive integer. *IEEE J. of Solid-State Circuits*, 23(1):204–207, February 1988.

[87] F. Manning. An approach to highly integrated computer maintained cellular arrays. *IEEE Trans. on Computers*, c-26(6):536–552, June 1977.

[88] G.H. Manolis and Miriam G. Blatt. Switch design for soft-configurable WSI system. In G. Saucier and J. Trilhe, editors, *Wafer Scale Integration*, pages 255–270, March 1986.

[89] Massimo Maresca, Mark A. Lavin, and Hungwen Li. Parallel architectures for vision. *Proceedings of the IEEE*, 76(8):970–981, August 1988.

[90] Massimo Maresca and Hungwen Li. Connection autonomy in simd computers: A VLSI implementation. *Journal of Parallel and Distributed Computing*, 7(2):302–320, October 1989.

[91] Alan J. Martin. The design of a self-timed circuit for distributed mutual exclusion. Technical Report 5178:TR:85, Computer Science Dept., California Institute of Technology, March 1985.

[92] Alan J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microproccessor. In C. L. Seitz, editor, *Proc. Decennial CalTech Conf. on VLSI*, pages 351–373, 1989.

[93] P. C. Mathias and L. M. Patnaik. Systolic evaluation of polynomial expressions. *IEEE Trans. on Computers*, c-39(5):653–665, May 1990.

[94] J. McDonald, E. Rogers, K. Rose, and A. Steckl. The trials of wafer-scale integration. *IEEE Spectrum Magazine*, pages 32–39, October 1984.

[95] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[96] Bilha Mendelson and Gabriel M. Silberman. An improved mapping of data flow programs on a VLSI array of processors. In *IEEE Proc. 1987 Int'l Conf. on Parallel Processing*, pages 871–873, 1987.

[97] Bilha Mendelson and Gabriel M. Silberman. Mapping data flow programs on a VLSI array of processors. In *The 14th Annual Symp. on Computer Architecture*, pages 72–80, June 1987.

[98] T. H.-Y. Meng, G. W. Jacobs, R. W. Brodersen, and D. G. Messerschmitt. Asynchronous processor design for digital signal processing. In *Proc. IEEE ICASSP*, pages 107–118, April 1988.

[99] Teresa H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Trans. on Computer-Aided Design*, 8(11):1185–1205, Nov. 1989.

[100] R. E. Miller. *Sequential Circuits*, volume 2. Wiley, NY, 1965. Chapter 10, In Switching Theory.

[101] David Misunas. Petri nets and speed independent design. *Communications of the ACM*, 16(8):474–481, August 1973.

[102] Will R. Moore. A review of fault-tolerant techniques for the enhancement of integrated circuit yield. *Proceedings of the IEEE*, 74(5):684–698, May 1986.

[103] Amar Mukhopadhyay, editor. *Recent Developments in Switching Theory*, chapter 7, 9. Academic Press, 1971.

[104] The Northwest Laboratory for Integrated Systems (LIS), Dept. of Computer Science, University of Washington. *VLSI Design Tools Reference Manual*, Feb. 1987.

[105] Jagdish Pathak, Hal Kurkowski, Robert Pugh, Ritu Shrivastava, and Frederick B. Jenne. A 19-ns 250-mW CMOS erasable programmable logic device. *IEEE J. of Solid-State Circuits*, sc-21(5):775–784, Oct. 1986.

[106] Nigel Charles Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, University of Manchester, 1994.

[107] M. Pechoucek. Anomalous response times of input synchronisers. *IEEE Trans. on Computers*, c-25(2):133–139, 1976.

[108] Vincenzo Piuri. Fault-tolerant hexagonal arithmetic array processors. In *Proc. Euromicro'88, Supercomputers: Technology and applications*, pages 629–636, August 1988.

[109] Plessey Semiconductors, Cheney Manor, Swindon, UK. *Electrically Reconfigurable Array - ERA*, 1990.

[110] J. L. Potter. Image processing on the massively parallel processor. *Computer*, 16(1):62–67, January 1983.

[111] J. Raffel, A. Anderson, G. Chapman, K. Konkle, B. Mathur, A. Soares, and P. Wyatt. A wafer-scale digital integrator. In *Proceedings, IEEE Int'l Conf. on Computer Design: VLSI in Computers (ICCD '84)*, pages 121–126, October 1984.

[112] J.I. Raffel. On the use of nonvolatile programmable links for restructurable VLSI. In *Proceedings of the Caltech Conf. on Very Large Scale Integration*, pages 95–104, Jan. 1979.

[113] A. L. Rosenberg. Graph-theoretic approaches to fault-tolerant WSI processor arrays. In Chris Jesshope and Will Moore, editors, *Wafer Scale Integration*, pages 10–23, July 1985.

[114] Arnold L. Rosenberg. The diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. on Computers*, c-32(10):902–910, October 1983.

[115] Arnold L. Rosenberg. Diogenes, circa 1986. In *VLSI Algorithms and Architectures, Aegean Workshop on Computing , Proceedings*, pages 96–106, July 1986.

[116] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-Modules: Internally clocked delay-insensitive modules. *IEEE Trans. on Computers*, 37(9):1005–1018, September 1988.

[117] A.J. Rushton and C.R. Jesshope. The reconfigurable processor array — an architecture in need of WSI. In Chris Jesshope and Will Moore, editors, *Wafer Scale Integration*, pages 149–158, July 1985.

[118] P. Sadayappan, Fikret Ercal, and Steven Martin. Mapping finite element graphs onto processor meshes. In *IEEE Proc. 1987 Int'l Conf. on Parallel Processing*, pages 192–195, 1987.

[119] Mariagiovanna Sami and Renato Stefanelli. Reconfigurable architectures for VLSI processing arrays. *Proceedings of the IEEE*, 74(5):712–721, May 1986.

[120] C. L. Seitz. Self-timed VLSI systems. In *Proceedings of the Caltech Conf. on Very Large Scale Integration*, pages 345–355, January 1979.

[121] Charles L. Seitz. An ensemble architecture for VLSI – a survey and taxonomy. In *1982 Conference on Advanced Research in VLSI, M.I.T*, pages 130–135, January 1982.

[122] Naresh R. Shanbhag and Pushkal Juneja. Parallel implementation of a 4x4-bit multiplier using modified booth's algorithm. *IEEE J. of Solid-State Circuits*, 23(4):1010–1013, August 1988.

[123] David Elliot Shaw and Theodore M. Sabety. The multiple-processor PPS chip of the NON-VON 3 supercomputer. *Integration, the VLSI journal*, 4(3):161–174, December 1985.

[124] Adit D. Singh. Interstitial redundancy: An area efficient fault tolerance scheme for large area VLSI processor arrays. *IEEE Trans. on Computers*, 37(11):1398–1410, Nov. 1988.

[125] Lawrence Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[126] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[127] M. J. Stucki and J. R. Cox. Synchronisation strategies. In *Proceedings of the Caltech Conference on VLSI*, pages 375–393, January 1979.

[128] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6), 1989.

[129] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Carnegie-Mellon University, Pittsburg, Pa., 1980.

[130] Lewis W. Tucker and Grorge G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, August 1988.

[131] S. H. Unger. A computer oriented toward spatial problems. *Proc. IRE*, 46:1744–1750, 1958.

[132] L. G. Valiant. Universal considerations in VLSI circuits. *IEEE Trans. on Computers*, c-30(2), February 1981.

[133] P.J. Varman. A fault-tolerant VLSI matrix multiplier. In *IEEE Proceedings of 1986 Int'l Conf. on Parallel processing*, pages 351–357, August 1986.

[134] Jouko Viitanen, Tapio Korpiharju, Jarmo Takala, and Hannu Kiminkinen. Mapping algorithms onto the TUT cellular array processor. In Sun-Yuan Kung, Earl E. Swartzlander, Jr, and Jose A. B. Fortes, editors, *Proc. of the*

Int'l Conf. on Application Specific Array Processors, pages 235–246, September 1990.

[135] D. F. Wann and M. A. Franklin. Asynchronous and clocked control structures for VLSI based interconnection network. *IEEE Trans. on Computers*, c-32(3):284–293, March 1983.

[136] Ian Watson and John Gurd. A practical data flow computer. *Computer*, 15(2):51–57, February 1982.

[137] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A System Perspective.* Addison-Wesley, 1985.

[138] T. E. Williams, M Horowitz, R. L. Alerson, and T. S. Yang. A self-timed chip for division. In Paul Losleken, editor, *Advanced Research in VLSI, Proc. 1987 Stanford Conf.*, pages 75–96, March 1987.

[139] Xilinx, Inc, San Jose, CA. *The Programmable Gate Array Data Book*, 1989.

[140] Hiroto Yasuura and Shuzo Yajima. Hardware algorithms for VLSI systems. In *VLSI Engineering – Lecture Notes in Computer Science*, pages 105–129, 1982.

[141] Hee Yong Youn and Adit D. Singh. On area efficient and fault tolerant tree embedding in VLSI. In *IEEE Proc. 1987 Int'l Conf. on Parallel Processing*, pages 170–177, 1987.

Appendix A

Hspice Transient Analysis

In this appendix, several typical input and output waveforms for the synchroniser and event-driven data transfer interface designs described in this thesis are given. These results were obtained by performing Hspice transient analysis on these circuits.

A.1 The Synchroniser

Four typical simulation results are given for the synchroniser design. Waveforms for a DV_R^{T+} happening after a CLK^{T-} and before a CLK^{T+}, a DV_R^{T+} happening at the start of a synchronisation risk zone are shown in figure A–1. In the figure, node 2 is a CLK input, and node 3 is a DV_R input. Node 5 is the A point in figure 5–13, and node 8 is the ENA output from the synchroniser as shown in figure 5–13. *si.tr0.55* is the simulation result for a DV_R^{T+} before a CLK^{T+}, and *si.tr0.4* is the simulation result where a synchronisation risk zone appears around $30ns$. It can be seen from the figure that it takes longer time for the A point to resolve to the 0 state in this synchronisation risk zone.

Waveforms for a DV_R^{T+} happening at the end of a synchronisation risk zone, and a DV_R^{T+} happening after a CLK^{T+} are shown in figure A–2. *si.tr00* is the

simulation result for a DV_R^{T+} at the end of the synchronisation risk zone around $30ns$, and *si.tr0.1* is the simulation result for a DV_R^{T+} after a CLK^{T+}. It is clearly shown in the *si.tr00* result that the A point is pull down to a middle value first in the synchronisation risk zone before it eventually resolves itself back to the 1 state.

The most important result obtained from these simulations is that the ENA can stay on low when the A resolves to one of the two stable states. The ENA either keeps low or flips to high after the A settles towards a stable state.

A.2 The Event-Driven DTI

Four simulated event-driven hand-shaking cycles are shown in figure A–3 for the event-driven DTI. This simulated event-driven DTI has an Input Guard, an Output Guard and a tri-state register. The *Din* is the data input to the tri-state register, and Q/nQ are the output from the register. The $DVIR$ is the status flag for the register. The $DVHop$ is the status flag for the function module. The node *net*13 is the reset signal $\overline{R}$.

The waveforms in the figure clearly show that this DTI works properly with the tri-state input register and is event-driven.

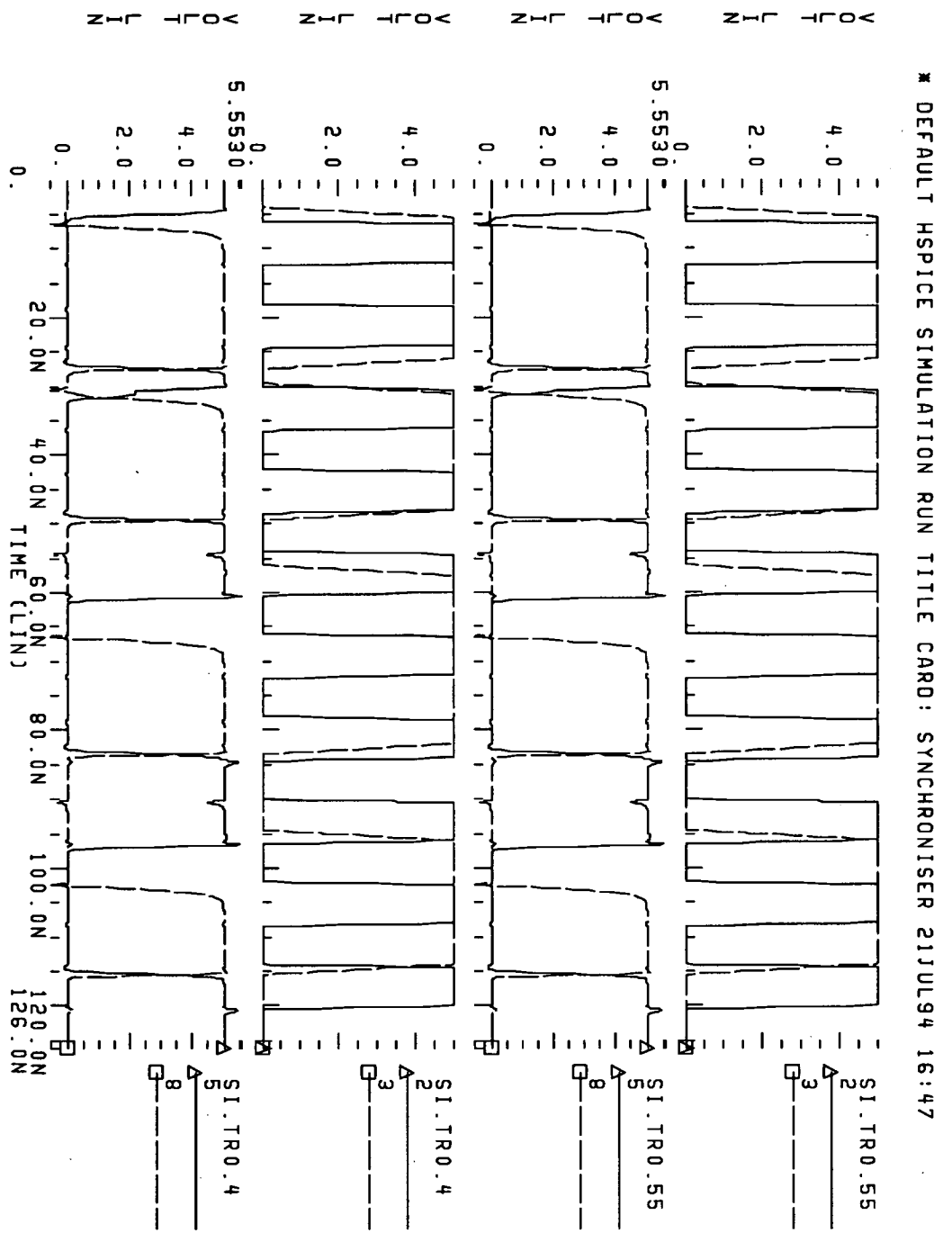


Figure A–1: A DV_R^{T+} before a risk zone

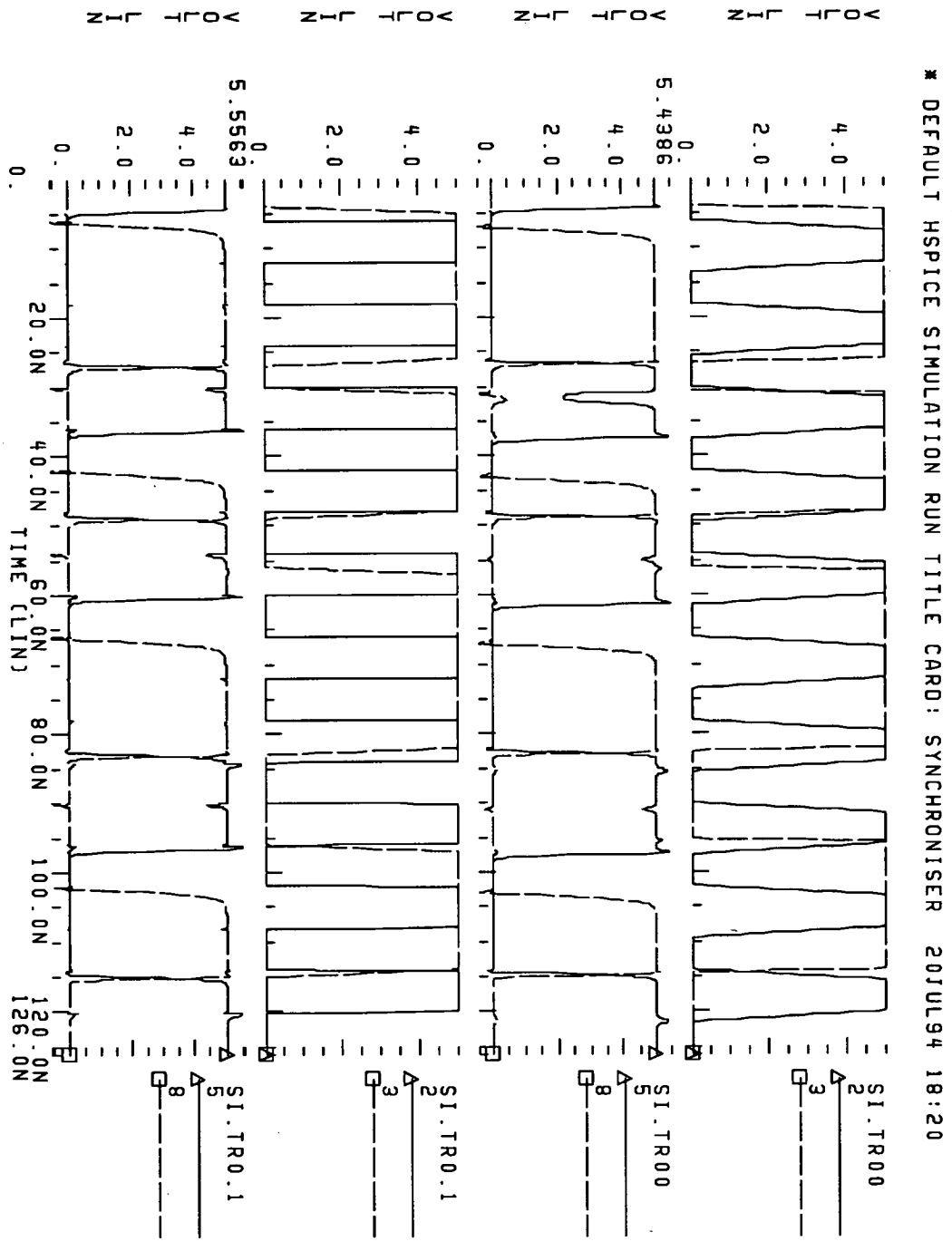


Figure A–2: A DV_R^{T+} after a risk zone

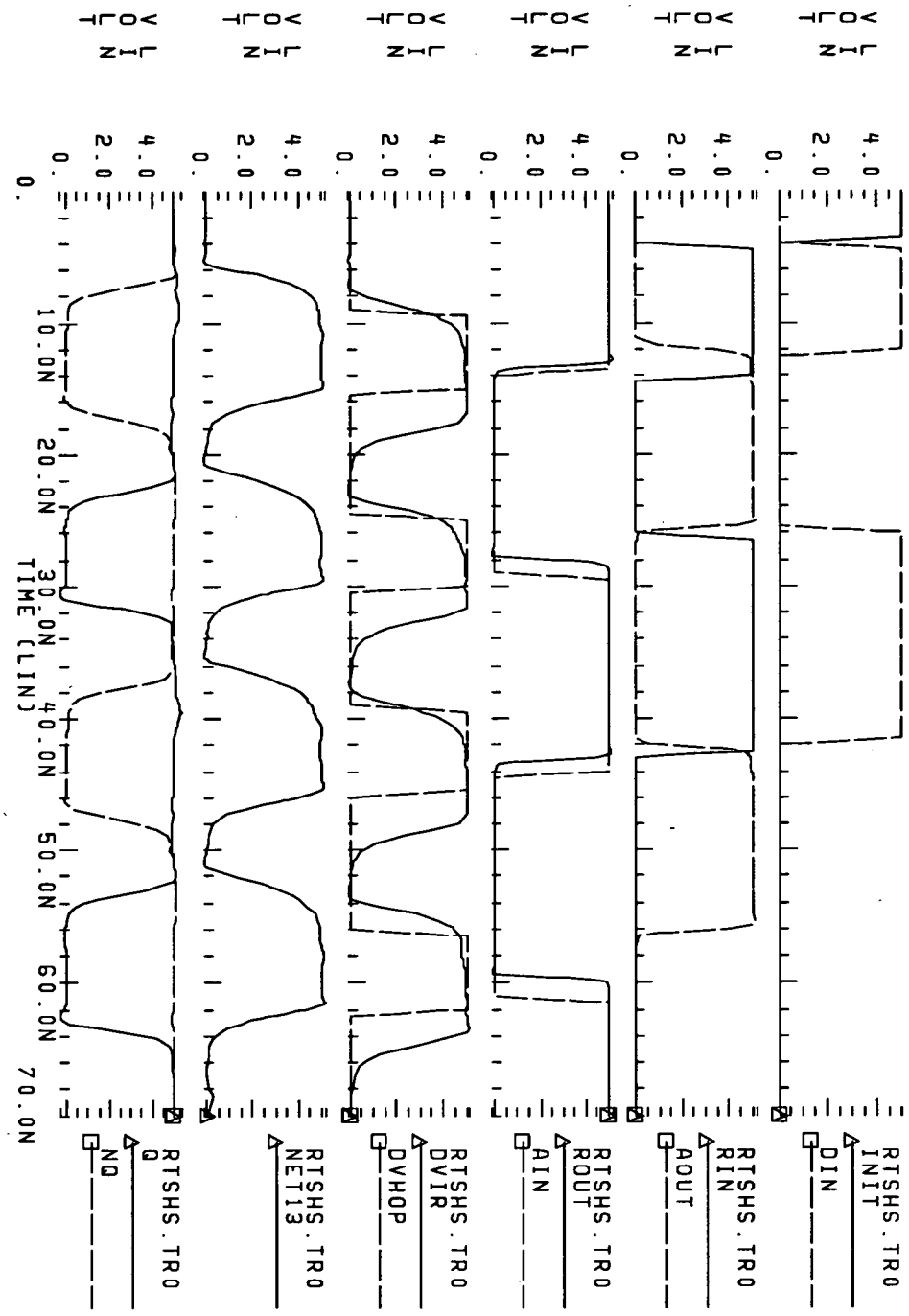


Figure A–3: The event-driven DTI