A comparative study of synchronous and self-timed systolic array architectures.

HOGG, R. S.

A COMPARATIVE STUDY OF SYNCHRONOUS AND SELF-TIMED SYSTOLIC ARRAY ARCHITECTURES

R. S. HOGG

DEGREE OF DOCTOR OF
PHILOSOPHY

1997

ABSTRACT

This thesis examines systolic array architectures and their methods of control and communication synchronisation.

Systolic array processors suffer from synchronisation problems associated with the clocking mechanism that causally restricts their scalability. To overcome this problem both return-to-zero (RTZ) and non-return-to zero (NRTZ) delay-insensitive self-timed (ST) techniques can be used to realise architectures that operate correctly in the presence of arbitrary delays at all levels in their design. As a consequence, RTZ and NRTZ versions of an existing systolic array architecture, namely the Single instruction Systolic Array (SISA), have been developed in order to investigate the potential for realising architecturally scaleable systolic arrays. The new architectures, called the RTZ and NRTZ ST-SISAs, have been compared with each other and against their synchronous counterpart to establish their relative trade-offs. The new designs exhibit several novel features including: variable length bit-serial data words, average case processing speeds dependent on data word length as well as computational complexity, a novel autonomous inter-processor data communication mechanism and architectural scalability independent of fabrication technology.

This thesis introduces an implementation of the RTZ and NRTZ ST-SISA architectures, along with their performance and area characteristics. Guidelines have been developed from the resulting RTZ and NRTZ architectures allowing novel self-timed systolic architectures to be derived.

PUBLICATIONS

1. Hogg R. S., Lloyd D. W., Hughes W. I., "Using Occam and Transputers to Emulate Asynchronous Self-Timed Array Processors", Proceedings of the 15th International Conference on Information Technology Interfaces, Pula Croatia, ISSN 1330-1012, pp. 257-262, June 15-18 1993.

2. Hogg R. S., Lloyd D. W., Hughes W. I., "A Self-Timed Massively Parallel Architecture with Elastic Control Flow", Proceedings of the International Society of Computers and Their Applications Conference, Long Beach, California, USA, ISBN 1-880843-08-0, pp. 22-27, March 17-19 1994.

3. Hogg R. S., Lloyd D. W., Hughes W. I., "Communication Techniques for a Self-Timed Massively Parallel Architecture", Proceedings of the First International Conference on Massively Parallel Computing Systems: The Challenges of General-Purpose and Special-Purpose Computing, IEEE Computer Society Press, Ischia, Italy, ISBN 0-8186-6322-7, pp. 55-61, May 2-6 1994.

4. Hogg R. S., Lloyd D. W., Hughes W. I., "Self-Timed Communication Strategies for Massively Parallel Systolic Architectures", Parallel Processing: CONPAR 94 - VAPP VI, Proceedings of the Third Joint International Conference on Vector and Parallel Processing, Springer-Verlag Lecture Notes in Computer Science 854, Linz, Austria, ISBN 3-540-58430-7, pp. 557-567, September 1994.

5. Hogg R. S., Hughes W. I., Lloyd D. W., "A Novel Asynchronous ALU for Massively Parallel Architectures", Proceedings of the Fourth EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 282-289, 24-26 January 1996.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my parents who have supported my academic efforts over the past ten years. I wouldn't have been able to get to this stage in my life without their constant encouragement.

I wish to thank my supervisors, Dr. David Lloyd and Mr. Iain Hughes, for their guidance and support throughout the period of this research, working with them has proved a rewarding experience.

I would also like to thank my Director of Studies, Dr. Doug Bell, for his help and support during this project.

Finally, I gratefully acknowledge Dr. Manfred Schimmler and Dr. Hans-Werner Lang for an interesting visit to ISATEC.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI) have made possible the realisation of high performance massively parallel architectures on a single chip. This is due to the increased number of devices that can fit onto a large silicon area. One such massively parallel architecture is the Systolic Array (SA) [43] which achieves high processing speeds but suffers from problems attributed to the synchronous implementation technology that uses a global clocking mechanism. The main problem is that of clock skew [25, 89], which is the phase difference of the global synchronising signal as it arrives at the various system components, resulting in incorrect system operation. Clock skew becomes more apparent the greater the distance the clock signal must propagate. As a consequence, both the clock frequency and design scalability are limited as the number of devices and the size of the silicon area increase [14, 17], and will continue to worsen as technology evolves. Other problems associated with the global clocking mechanism include fixed instruction processing speed [16, 22] and fixed word length.

In response to the problems of clock skew both synchronous [34, 41, 42] and asynchronous self-timed design techniques [49, 74, 95] have been developed. The synchronous techniques adopt specially designed clock distribution networks that minimise the phase differences of the clock signal in the circuit, consequently, clock skew is only reduced and not eradicated. However, alternative asynchronous self-timed techniques exist that abolish the clock and instead use local data flows to co-ordinate correct circuit operation therefore removing clock skew.

1.2 The Systolic Array

Kung [47] proposed the SA as an elegant solution to computationally demanding applications. The high performance resulting from this approach has culminated in a large number of SA designs [39]. A typical SA [15] is a grid-like architecture of identical processing elements that process data much like an n×n dimensional pipeline. In physiology, the term 'systolic' describes the contraction of the heart, which regularly sends blood to all cells of the body through arteries and veins. Analogously, SAs perform operations in a rhythmic, regular and repetitive manner [44].

The progression of SA performance has been made possible through the rapid evolution of VLSI technology which has allowed a greater number of component devices to be fitted onto increasingly larger silicon chips. However, the smaller size of devices and the increasing speed at which they switch cannot be exploited because of clock skew [14]. These problems will continue to worsen with the advent of WSI [40, 55, 110] because a substantially larger number of devices can be fitted onto a substantially larger silicon area. As a consequence, the problems associated with clock skew are exacerbated because the clock signal must propagate for much longer distances. This increase in clock skew results in limited SA scalability. The alternative considered in this thesis is the asynchronous self-timed design method. The self-timed approach eliminates the need for a clock signal at any level in the design.

1.3 System Timing

The fundamental difference between synchronous and self-timed design is the manner in which circuit delays are handled as they pass through the system. These are considered as follows:

1.3.1 Synchronous Design

In a synchronous system responsibility for overcoming delays is passed to a global control centre. The worst case delay through the system is calculated and then a global signal, called the system clock, timed to match this delay is distributed to all modules. Consequently, all system functions operate to a worst case processing speed dictated by the slowest operation, the worst case signal delay and to restrictions imposed by clock skew.

1.3.2 Self-timed Design

In a self-timed system responsibility for overcoming delays is passed to the individual circuit modules themselves. The system modules co-ordinate computation both within themselves and communication with their immediate neighbours. This is achieved by each circuit determining when its computation is finished using special completion detection logic. Therefore, each stage of computation is triggered by local data flow rather than a global clock signal. This allows each computational module to operate at its own rate of computation resulting in task dependent processing speed.

1.4 Motivation

There exist many SA designs that are now well established as solutions to computationally demanding applications. However, future performance enhancement of SAs is restricted by clock skew. The alternative is the self-timed design method which removes the clock signal and its associated drawbacks. The self-timed approach potentially provides a solution to the restricted performance and scalability of SAs.

Self-timed design has now matured to a stage where the rules and techniques for constructing circuits are well established [35, 66, 88, 90, 102]. At the circuit design level, a wide range of circuit implementation techniques exist, such as the speed-independent [102], delay-insensitive [9] and bounded delay models [100], that vary in performance, circuit cost and synchronisation techniques. These circuits can be designed using a number of formal techniques, such as Communicating Sequential Processes (CSP) [28] and Signal Transition Graphs (STGs) [8], that adopt different design approaches to formally describe circuit operation. These formal descriptions can then be synthesised to realise the correct behaviour of the model in the circuit implementation. Alternatively, standard macro-module design libraries [58, 100] exist such that designers do not need to know about the low level implementation details of circuit modules and the design methodology with which they were developed.

At the system level, work has been undertaken by Unger [103], Meng [64] and Staunstrup [96, 97, 98, 99] concerned with establishing methods for connecting these self-timed circuits to form self-timed computing systems. However, the application of these methods is limited and mainly restricted to the research groups that first developed the methodologies. As a consequence, the macro-module approach used by designers of synchronous systems, using the Micropipeline library [100] of macro-modules, predominates.

Self-timed processors developed to date have focused on single processor designs [6, 18, 62]. Consequently, very little work has been undertaken on the design of self-timed SA processors. A self-timed special-purpose SA of simple construction has been developed by Lloyd [57] demonstrating that performance improvement is possible adopting the self-timing design technique when compared to the existing synchronous technique. What is required is an assessment of the wide range of the different self-timed design techniques for scaleable general-purpose SA design.

1.5 Aims and Objectives of the Thesis

This thesis has two main aims. The first is to establish a coherent design framework that allows the trade-offs of new architecturally scaleable self-timed massively parallel processor designs to be compared with each other and their existing synchronous counterparts. The second aim is to use this framework to establish new guide-lines that assist in the design of new self-timed scaleable massively parallel architectures.

The first aim is a direct result of the very limited amount of knowledge in the design of scaleable self-timed SA architectures. The vast majority of research to-date in self-timing has concentrated on formal methods [7, 28], synthesis techniques [8, 59, 65] and the construction of single-processor architectures [6, 19, 20, 61] and so little information exists of the system level construction of self-timed architectures. The designer is also unable to determine which self-timed techniques to use to realise truly scaleable SA architectures and their resulting costs and performances. Therefore, the first aim of this research is to establish the trade-offs of self-timed techniques with respect to each other and their synchronous equivalents through the development and exploration of a common research vehicle.

The first aim is to derive a coherent design framework to determine the differences between implementation techniques. Three objectives are identified.

1. To determine requirements for the implementation of self-timed architectures when used for SA design.
2. The development of novel self-timed architectural structures based on the requirements developed in (1) which lead to scaleable and modular array designs.

3. To characterise the implementation characteristics in terms of area and performance for the existing synchronous and novel self-timed architectural designs and to compare their relative trade-offs.

The second aim is to establish strategies for the design of self-timed architectures utilising the framework from the first aim. Two objectives are identified:

1. To produce guide-lines for the design of new self-timed architectures which do not require re-design when ported between architectures or circuit technologies.
2. To produce guide-lines for the selection of self-timed or synchronous circuit design techniques for a given set of criteria.

1.6 Structure of the Thesis

Chapter 1 has introduced, thus far, the subjects of self-timed design and systolic arrays and has provided an overview of their current status. The chapter has also presented the aims and objectives of this thesis. Finally, chapter 1 highlights the contributions to knowledge resulting from this work.

In **Chapter 2** SA architectures are introduced. From this an architectural taxonomy is used to categorise the different SA design approaches. The limitations of clocked control for SAs are then identified through a review of both the existing synchronous and self-timed techniques that attempt to alleviate or overcome these problems. The current status of self-timed design is then reviewed. This begins with an identification of the main methods of implementing self-timed circuit modules, followed by methods used to interconnect these modules to form system architectures. A review of the main methodologies used to design self-timed circuits is also undertaken. A number of key architectures that adopt these design methodologies are then reviewed and analysed. This chapter concludes by identifying the areas which require further investigation, leading to three experimental chapters and the identification of a research vehicle which together provide the basis for the research.

In **Chapter 3** three main experiments are proposed to address each of the experimental objectives identified in chapter 2. The experimental methodology adopted by each of these experiments is described and the assumptions shared by all the experiments are stated.

In **Chapter 4** the background to the adopted research vehicle, namely the Self-Timed Single Instruction Systolic Array (ST-SISA), is presented. It presents the existing synchronous and the new self-timed architectures which are described in terms of their array interconnection, processing element (PE) architecture and programmability.

In **Chapter 5** the data path experiment is presented. This chapter provides a detailed explanation of each of the data path components, which are used to realise data storage, data communication and arithmetic/logic processing, and analyses their implementation characteristics. The results for this study are used to derive a set of guide-lines to assist in the design of data path units.

In **Chapter 6** the control path experiment is presented. It provides a detailed explanation of each of the control path components which are used to realise instruction by-pass, instruction fetch, instruction decode and functions that are required to co-ordinate the processing cycles of the self-timed PEs. This is then followed by analysing their implementation characteristic relationships. The results for this study are used to derive a set of guide-lines to assist in the design of control path units.

In **Chapter 7** the integration of the control and data path units, in order to realise a working PE, is presented. This allows the relative circuit area and performance of each of the self-timed PEs to be compared against each other and with their synchronous equivalent. The ST-SISA PEs are then connected together to form a working array. This allows the effects on PE performance resulting from inter-PE instruction and data communication to be established.

In **Chapter 8** the conclusions from each of the previous experiments are summarised. The limitations of the work are examined and possible extensions to the existing investigations are discussed, as well as further areas of research. Finally, the thesis is concluded by summarising the main points of the research.

1.7 Contributions to Knowledge

This thesis does not attempt to resolve all of the problems, or design issues, associated with the implementation of SA architectures. Instead this work aims to provide a coherent framework in which to design truly architecturally scaleable self-timed massively parallel processors through the adoption of new self-timed design techniques and strategies. In addition, this design framework allows the cost/performance trade-offs of the new self-timed designs to be compared with their existing synchronous counterparts. The contributions to knowledge of this thesis are as follows:

- A methodological approach to the design of self-timed SA architectures.
- Novel scaleable self-timed SA architectures that exhibit common architectural structures and modularity.
- The characterisation of the new self-timed and existing synchronous SA architectures thus establishing their relative trade-offs in terms of cost and performance.
- Design guide-lines for the development of self-timed data path and control path architectures adopting new self-timed design techniques.
- Guide-lines for the selection of the appropriate self-timed or synchronous design method for SAs for given design criteria.

CHAPTER 2

REVIEW OF RELATED WORK

2.1 Objectives of the Chapter

This chapter introduces VLSI array architectures, primarily systolic array (SA) processors. This review will draw upon example architectures in order to illustrate their architectural, computational and communicational strategies and to identify the effects and trade-offs of adopting various synchronous and asynchronous control techniques. This chapter further investigates the techniques for self-timed control, and identifies those aspects that may prove appropriate for the exploitation by systolic architectures. Finally, this chapter proposes areas which require further research. These objectives are achieved by considering the following:

- Identify and categorise SAs, and introduce a taxonomy for their representation.
- Review the key SA architectures utilising the adopted taxonomy.
- Analyse the key aspects identified in the reviewed SA architectures.
- Review self-timed control techniques and design methodologies.
- Review the key self-timed architectures.
- Analyse the key aspects identified in the reviewed self-timed architectures.

2.2 Systolic Array Architectures

This section presents the background to SAs. From this a taxonomy will be developed which will be used to categorise the architectures according to their program control mechanism. Examples will be given of SA processors that achieve high performance through their use of modular design and localised communication.

2.2.1 Background to Systolic Arrays

The application domain of SA processors covers a wide variety of applications including image processing, sonar, radar, computer vision, speech and medical signal processing, the key characteristics of which are the enormous throughput rates and the huge amounts of data involved. Meeting these demands required a revolution in computing technology. This improvement in performance was only practicable with the development of smaller component devices on a larger chip area. As a consequence, high device speeds in conjunction with high degrees of on-chip parallelism have achieved the speeds required by modern day applications.

The degree of array concurrency is largely dictated by the underlying algorithm, and massive parallelism can only be achieved if the algorithm is designed to permit high degrees of pipelining and multiprocessing. Consequently, SA algorithms are dominated by convolution, matrix or transform algorithms which possess common properties such as regularity, recursiveness and locality [47]. The properties shared by these algorithms can be best exploited by SAs because of their intensive local communication, decentralised parallelism and modular, rhythmic, intensive and repetitive computation. The dominating aspects of this approach are elaborated in the following section.

2.2.2 Introduction to Systolic Array Architectures

A SA is an architecture constructed of identical processing elements, each synchronised by a clock signal. The array performs operations in a rhythmic, cellular and repetitive manner. Data is pumped through the array producing partial results as it is passed concurrently through the array from one processing element (PE) to the next. Consequently, SAs have a very high rate of throughput. A SA can take the form of a wide variety topologies [47]. The choice of topology is dependent on the application and the required performance.

The PEs of a SA can take one of three forms, differentiated by their programmability. An increase in programmability results in greater flexibility allowing the same array to execute a wider variety of algorithms. However, there is also a corresponding increase in the complexity and hence the size of the PE. The programmability of a SA can be differentiated as follows:

1. **A fixed-function PE** that contains hardwired functions. The size and simplicity of these PEs can mean that hundreds can fit on a single chip.
2. **A programmable PE** that bridges the gap between inflexible fixed-function and programmable architectures. This approach adopts a vector-like PE containing an

instruction decoding unit, without a control store, and a processing unit. This approach holds the instruction store external to the chip itself meaning that the circuit area consumed by each programmable PE is kept to a minimum.

3. **A fully programmable PE** complete with control store and processing unit. This PE essentially corresponds to the application generality of the von-Neuman type of computer architecture, though at the expense of increased PE circuit area, thus reducing the degree of on-chip array parallelism.

2.2.3 Taxonomy of Systolic Array Architectures.

There are a wide variety of SAs differing in their range of complexity and performance capabilities. In order to review the area a simple classification system has been adopted to identify key control and computational differences. The adopted classification focuses on the subdivision and elaboration of:

- the control mechanism.
- the functional complexity.

The taxonomy uses the Flynn classification notation [37] for various computer architecture types based upon the notion of instruction and data streams, namely; special-purpose, single instruction multiple data (SIMD) streams and multiple instruction multiple data (MIMD) streams. This is illustrated in figure 2.1. The distribution of control information can be classified as either central or local. From this the functional complexity can be divided into SIMD for centrally controlled SAs, and special-purpose or MIMD for locally controlled SAs.

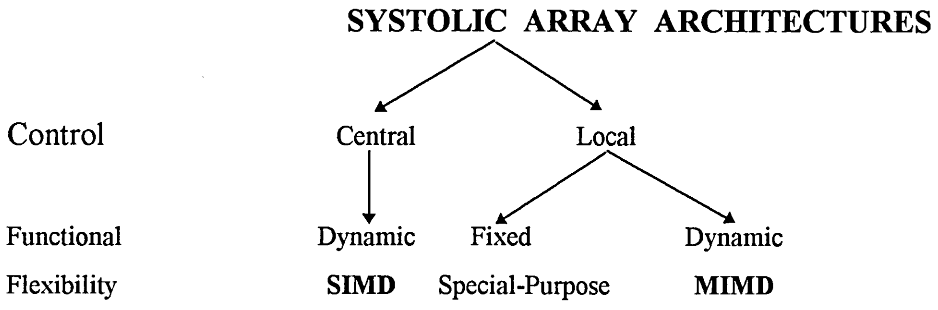


Figure 2.1. An array architecture taxonomy.

The special-purpose, SIMD and MIMD machine models are discussed in more detail in the following three sections.

Special-Purpose Systolic Arrays

Special-purpose systolic architectures need to be custom designed for each application because of their fixed-function. This type of SA architecture achieves high performance through simple design, and is very attractive from the viewpoint of simplicity and efficiency, but suffers from high design costs and lack of flexibility. Consequently, this type of architecture will not be considered in this work. Examples of special-purpose SA architectures can be found in [46, 47].

Single Instruction Multiple Data (SIMD) Systolic Arrays

SIMD processors provide a programmable alternative to special-purpose arrays. They are realised as an array of identical PEs, each having their own data memory which can be shared by immediate neighbours. Instructions are broadcast to all PEs such that they perform the same operation on different data, thus eliminating the need for on-chip control storage. The advantage of adopting this approach is that the PEs of a SIMD SA take up very little space on the silicon, therefore increasing the degree of on-chip parallelism. However, the generality of this type of architecture is restricted due to the broadcast nature of instructions. Examples of this type of architecture include Saxpy-1 [15] and the Brown Systolic array [36].

Multiple Instruction Multiple Data (MIMD) Systolic Arrays

MIMD processors are arranged as an array of identical PEs. Application generality is maximised as each PE has its own control unit, program and data store. Each PE, similar in design to that of a conventional von Neumann architecture, can be loaded with a different program or all PEs may loaded with an identical program. Data memory is usually not shared in a MIMD SA, but instead is passed along communication lines to the next PE. This results in problems of decreased processing throughput, due to the higher dependency on off-chip pin communications, and of synchronisation between multiple PEs. Examples of this type of architecture include iWarp [5, 24] and WAP [84].

2.2.4 Review of Key Array Architectures

The aim of this section is to review the current status of SA architectures. This review is broken down into two sections namely the SIMD and MIMD control classifications, with each review focusing on data and instruction communication, array synchronisation, and instruction set complexity. The Blitzen and Datawave processors reviewed in this section are not systolic architectures, however they exhibit architectural features that are relevant to the research and so are included for completeness.

2.2.4.1 SIMD Systolic Arrays.

2.2.4.1.1 Brown Systolic Array [36]

The Brown Systolic (B-SYS) array has a linear array architecture designed for sequence comparison and sorting applications. It has been designed to fit the maximum number of PEs onto a single VLSI chip. To achieve this each PE can only incorporate a very simple instruction set including basic Boolean, addition and comparison operations. To further maximise the degree of array parallelism each PE is only capable of storing eight bit words.

The B-SYS processor adopts a novel instruction communication mechanism where instructions are sent systolically through the linear array over a uni-directional control path. This is also the case for data interconnection. The instruction processing unit of each PE is of SIMD complexity though at the expense of extra circuitry as each PE requires its own dedicated instruction decoder.

Due to the systolic nature of instructions, and the array synchronisation mechanism that enforces global synchrony of PEs, read-after-write (RAW) hazards are handled by the notion of processing phases such that loads take place after all other computations have finished. Also, for more complex programs, such as those required for sequence comparison, alternate PEs are temporarily disabled to overcome RAW hazards. This is achieved through the use of a phase flag for each PE, which specifies its current phase, and a phase tag for each instruction. Therefore, PEs in phase *j* execute only those instructions tagged with the same *j* phase.

2.2.4.1.2 Saxpy Matrix-1 [15]

The Matrix-1 is a linear SIMD array processor designed to be used for a wide variety of signal processing algorithms. The array connects to a system memory which stores all the data arrays externally to the processor array; a system controller broadcasts Matrix-1 instructions over a combined data and control interconnect mechanism.

The array can consist of either 8, 16, 24, or 32 identical PEs, called computational zones, which are globally synchronised by a centralised clock. Each zone contains a pipelined 32-bit floating-point multiplier, a pipelined 32-bit floating-point adder with logic capabilities (its ALU), realising a complex instruction set, and a 4K-Word local store. The processor array can function in either the systolic mode or in block mode, in which all zones operate independently and use local data. Although the Matrix-1 is an SIMD architecture the programmer is allowed more freedom as computational zones may be disabled via the setting of a mask bit.

The Matrix-1 processor has three types of data communication mechanism each of which are one word wide and operate at the global clock speed. The first type of interconnect is the systolic data path which connects each zone to its right neighbour with the last zone connecting to the output buffer or, in a circular fashion, to the first zone. The second type of interconnect utilises the global buffer as the source to broadcast data to all the zones, though this conflicts with the systolic nature of the design. The third mechanism connects the local memory of each zone both to the zone's neighbouring PEs and to the processor interface.

2.2.4.1.3 Blitzen [4, 27, 38]

The objective of the Blitzen project was to construct a physically small high performance programmable massively parallel processor for use in real-time applications. The Blitzen chip achieves this by adopting a SIMD mesh array, consisting of 128 identical bit-serial PEs each with 1K Bytes of addressable memory, synchronised by a global clock. Each PE comprises:

- 16 logical operations.
- Single-bit full adder for bit-serial addition and two's complement subtraction.
- 32-bit variable length shifter.
- 6 single-bit registers.
- 1K byte of ram.

The variable length shift register can be configured to hold 4-, 8-, 12-, 16-, 20-, 24-, 28-, or 32-bit data words. An *n*-bit add (where *n* represents the data word length) is accomplished by specifying in the instruction the number of *n*- cycles to perform the load, and the number of *n*-cycles to perform the addition. As Blitzen is a SIMD machine the same instruction word is broadcast to each PE that performs the same operation on the same sized word. Two data communication interconnection mechanisms exist in the Blitzen array:

- PEs are inter-connected using an X-grid with nearest neighbour connection paths in eight directions. A routeing operation sends data out, bit-serially, in one direction and accepts data in from another. All PEs route in the same direction in one processing cycle.
- High system I/O bandwidth is provided by eight 4-bit I/O buses, one for each row, which shift data in from the west edge, and shift data out simultaneously along the east edge. In one processing cycle, 32 bits can be accessed at a column of eight PEs on a chip.

The Blitzen array processor consists of 16,384 PEs on 128 chips.

2.2.4.2 MIMD Systolic Arrays.

2.2.4.2.1 Instruction Systolic Array [53, 54, 78, 79, 80, 81, 82, 85, 86, 104, 105]

The ISA is a new variation on the systolic array architecture. In the ISA, instead of pumping data through the array like the SA, instructions called the top program and an orthogonal sequence of Boolean row selector bits called the left program are pumped through the array; resulting in the name - **instruction** systolic array. A PE receives an instruction and a selector bit in each cycle from its north and west neighbours respectively, therefore, each PE has a one word instruction register and two single bit selector stores. If an instruction meets a logic '1' selector bit then that instruction is executed, otherwise a no-operation is performed and the contents of the registers are left unchanged.

Data communication between neighbouring PEs is achieved on the execution of a read instruction. Each PE can read the communication register (CR), which can be read by five PEs including itself, of one of its four direct neighbours. To avoid RAW conflicts, the CR is read by one or more neighbouring PEs in the first half of the instruction cycle, and then each reading PE writes the data to its own CR in the second half of the cycle.

The ISA is closely related to the concept of wavefront array processing in that a set of instruction and selector diagonals propagate in wavefronts through the array from the top left-hand PE toward the bottom right-hand PE. However, the synchronisation of instruction and data transfer is regulated by a global timing signal.

The commercial ISA processor consists of 64 identical bit-serial PEs with a complex instruction set including:

- Logical operations.
- A full adder for bit-serial addition and two's complement subtraction.
- Comparator and multiplier.
- Floating-point unit.
- Shifter.

2.2.4.2.2 Single Instruction Systolic Array [53]

The SISA is closely related to the ISA except that it uses column selector bits in addition to row selector bits and has only one instruction input to the top left PE; hence the name - **single** instruction systolic array. The advantage of this variant on the ISA is the reduction of the overall amount of code by approximately m/2, m being the length of the instruction code. This reduction in code is only possible in programs where in each instruction diagonal there occurs only identical or no-operation instructions. As discovered by Lang [53] this is frequently the case in most practical ISA applications.

2.2.4.2.3 Wavefront Array Processor [48, 50, 51]

The Wavefront Array Processor (WAP) supports the same programmability and local interconnectivity on word-size data as that of a programmable SA. However, it fundamentally differs in that it is data driven rather than globally synchronised. The WAP makes use of local clocking strategies to synchronise operations internal to each PE, therefore each PE is driven by internal clocks. However, it uses asynchronous handshaking communication lines between PEs to control the flow of data. This allows data to be passed to successive PEs only when they are ready to accept new operands, or to suspend waiting PEs until the input data is present. The benefits of this structure are threefold:

- Data communication does not have to be synchronised or scheduled to meet at the correct PE at the correct time, but instead only the correct ordering of the data is necessary to ensure the correct data meets the correct PE.
- The asynchronicity of data communication means that data can be queued up between PEs of varying execution times, thus freeing up predecessor PEs for their next computation.
- The use of asynchronous communication means there is no need for a global clock and the effects of inter-PE clock skew are eliminated, though internal PE clock skew still remains a potential bottleneck.

A WAP is typically a grid-like array architecture, like the SA, in which computational 'wavefronts' [13] are pipelined through a localised communication network. Each PE consists of a simple modular architecture consisting of internal program memory, a control unit, an ALU, and a set of registers. Every PE has a bi-directional buffer and independent status and control flags that are used to handle inter-PE data communication for each of its four adjacent PE neighbours. To understand the concept of computational wavefronts, an N × N matrix multiplication algorithm is executed on a orthogonal WAP, as shown in figure 2.2.

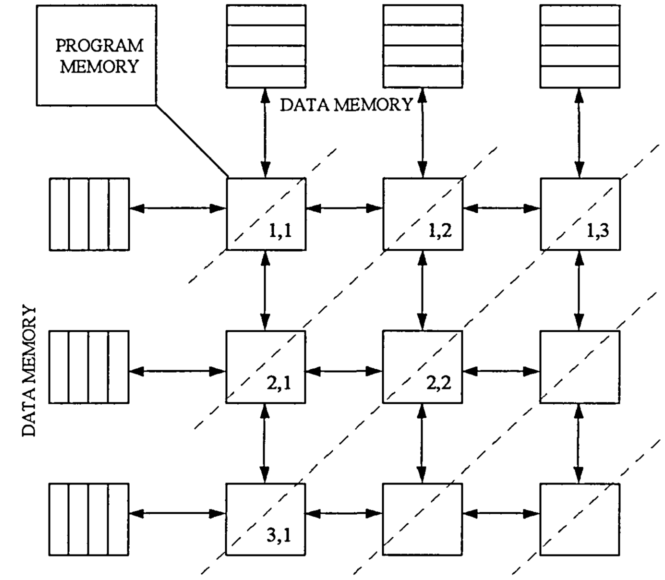


Figure 2.2. The wavefront array.

Consider three matrices where A=a_{ij}, B=b_{ij} and C= A×B = {c_{ij}}. Matrix A is decomposed into columns A_i and matrix B into rows B_i, therefore C=$A_1 \times B_1$ + $A_2 \times B_2$ + . . . + $A_N \times B_N$. The matrix multiplication can be carried out by N recursions of wavefronts on the WAP. The elements of A are stored in memory modules to the left, and those of B in memory modules to the top. The process begins with PE(1,1), where $c^{(1)}_{11} = c^{(0)}_{11} + a_{11} \times b_{11}$ is computed. The computational wavefront then propagates to the neighbouring PEs (1,2) and (2,1) which then execute their respective operations. The wavefront then propagates to PEs (3,1), (2,2) and (1,3). Thus, a computational wavefront proceeds through the array from the top left to the bottom right corresponding to one complete recursion. As the first wave propagates to the second diagonal an identical second recursion can be concurrently executed by pipelining it directly after the first one. This pipelining is feasible because two wavefronts are never allowed to intersect with one another, so different PE diagonals are used to execute different recursions at any given time.

2.2.4.2.4 Datawave[83, 84]

The Datawave multiprocessor chip has been developed by ITT Intermetall for High Definition TeleVision (HDTV) applications. The Datawave chip contains 16 mesh-connected PEs operating on 12-bit fixed-format data words. Each PE is based on the pipelined RISC concept with local control and data stores.

The processor array is globally synchronised by a 125-MHz clock which is distributed along an H-tree to minimise clock skew. Inter-PE data communication is possible over eight links configured as one read and one write bus at each of its four sides; each bus consists of 12 data and two handshake lines. All communication between PEs is data driven. Thus, only when all required data has arrived will processing commence. Local asynchronous handshaking protocols synchronise localised data transfers to the local PE clocks; where local PE clocks are derived from the global clock.

In order to absorb wait times, FIFO buffers are used between PEs to smooth out dataflow density fluctuations. These in-built mechanisms result in the programmer not needing to worry about timing or scheduling of data transfers, as long as the sequencing is correct. The Datawave PE architecture consists of the following modules:

- A multiplier-accumulator (MAC) which multiplies two 12-bit fixed point words producing a 24-bit product. The accumulator may output either a correctly scaled 12-bit product or in two-cycles as a 24-bit product (hi, lo) that includes over-flow bits.
- An ALU that works in parallel with the MAC, both of which are fully pipelined, and performs all arithmetic and logical operations on one or two operands. The ALU output is padded with five additional registers to match the latency of the MAC path. This ensures ALU and MAC results appear at the common output bus in the same order in which their corresponding instructions have been issued.
- A four-port register bank with 16 registers. In each cycle two registers can be read, and two can be written.
- A static RAM program store which can hold 64 instructions each of which is 46-bits wide.

2.2.4.2.5 iWarp [5, 24]

iWarp, developed by Carnegie Mellon University and Intel Corporation, has been specially designed for image and signal processing applications. The iWarp, the successor to Warp [2], is a single chip processor containing a single PE that must be connected to other iWarp PEs to form a systolic array. An iWarp PE consists of a computation agent and a communication agent that can carry out operations independently from one another. Therefore, a PE may perform a computation while communication through the PE is taking place with other PEs thus supporting loosely coupled message passing. Tightly coupled communication is also supported such that systolic algorithms can be executed. The computation agent consists of an IEEE standard 64-bit floating point adder and multiplier, a 32-bit integer/logic unit and a 128 word register file. The communication agent has four physical pairs of input and output ports thus allowing 1-D, 2-D and Torus networks to be configured.

2.2.5 Analysis of Key Systolic Array Architectures

The review section has briefly introduced the key architectures that exist in this area. However, an analysis must be undertaken to determine which features are common to these architectures, which features differ, and the rationale for the choice of a particular feature. The identified areas of analysis are split up as follows:

1. What are the relative trade-offs of the various **array features**; those of SIMD/MIMD control, array synchronisation and array topology? This will identify whether SIMD or MIMD architectures realise the highest PE density per chip and the extent of which data-driven mechanisms have been adopted.
2. What are the relative trade-offs of the various **data word features**; those of data orientation (serial or parallel) and data word size? This will establish the trade-offs between cost, performance and application generality.
3. What are the relative trade-offs of the various **PE functionality features**; those of instruction complexity, instruction timing and local storage? This will clarify how SA PEs can improve their performance and/or reduce their circuit cost through the implementation and control of their ALU.
4. What are the relative trade-offs of the various **data communication features**; those of the number and orientation of data ports, and the method of communication synchronisation? This will identify the degree of which communication can take place between neighbouring PEs and the communication mechanisms used to assist PE autonominity.

2.2.5.1. Analysis of Key Features

The following four sections analyse the array, data word, PE functionality and data communication areas. In each section the various features of the reviewed architectures are depicted in a tabular format so that the differences and commonalties of the various architectures can be clearly seen. The rationale for the adoption of a particular architectural feature and the reasons for the architectural differences of the reviewed key architectures are identified.

Array Features

Table 2.1 presents the array level viewpoint. The first column shows that the numbers of **PEs per chip** varies enormously. One of the reasons for this is attributed to the location of the control store. The B-SYS, ISA and SISA abolish the on-chip control storage and instead systolically pipeline the program through the array read from an external control store, whereas the WAP and iWarp store the program locally to each PE. Therefore, B-SYS, ISA and SISA maximise the number of PEs fit onto a chip.

The second column shows that **array topology** varies between the architectures. The reason for this is dependent on the application program [47].

The third column shows that the majority of the reviewed architectures adopt a globally clocked **timing** mechanism. However, each WAP PE chip is locally clocked. Therefore, WAP PEs are data driven. Blitzen PEs on each chip are also data driven, however, internal clocks are derived from the global signal.

	PEs Per Chip	**Topology**	**Timing**
B-SYS	Not available	Linear	Globally Synchronous
Matrix-1	Not available	Linear	Globally Synchronous
Blitzen	128	Mesh	Globally Synchronous
ISA/SISA	64	Mesh	Globally Synchronous
WAP	1	Mesh	Locally Synchronous
Datawave	16	Mesh	Globally Synchronous
iWarp	1	Any	Globally Synchronous

Table 2.1. Array features.

Data Word Features

Table 2.2 illustrates the difference in the data word features. It is important to note the **data orientation**, shown in the first column, has a significant effect on the number of wires and data throughput. Bit-serial orientated data is sent between circuit modules over a single wire, therefore consuming less circuit area than bit-parallel orientated data which requires one wire per bit. However, the speed at which bit-serial data can be communicated between circuit modules is slower than bit-parallel data. This is because bit-serial data is sent over a single wire requiring *n* clock cycles to communicate *n* data-bits, as compared to bit-parallel data in which all *n*-bits are communicated concurrently in the same clock cycle.

The second column shows the number of bits used to encode the data, or the **data size,** which is usually dependent on the application environment. For example, Datawave is specifically targeted at HDTV applications, which require only 12-bit data words, whereas more general-purpose architectures, such as the WAP and iWarp, adopt a larger data bandwidth to encompass a wider application environment. Another aspect of bit-serial data is the ability to **vary the length** of data words, as possible in the Blitzen processor.

	Data Orientation	**Data Size**	**Variable Length Support**
B-SYS	Bit-Parallel	8-bits	NO
Matrix-1	Bit-Parallel	32-bits	NO
Blitzen	Bit-Serial	24-bits	YES - 4-bit increments
ISA/SISA	Bit-Serial	16-bits	NO
WAP	Bit-Parallel	32-bits	NO
Datawave	Bit-Parallel	12-bits	YES - 12 or 24-bit products
iWarp	Bit-Parallel	32-bits	YES - 32 or 64-bit products

Table 2.2. Data word features.

Processing Element Functional Features

Table 2.3 shows the PE functional features. As can be seen in the first column, **ALU complexity** is simple for both the ISA and Blitzen architectures as they adopt a bit-serial

(1-bit data path) ALU due to the bit-serial orientated data. The bit-parallel ALUs, adopted for the Matrix-1 (32-bit), WAP (32-bit), Datawave (12-bit) and iWarp (64-bit) processors, are larger and more complex. The ISA adopts a bit-parallel multiplier due to the relative poor performance of bit-serial multipliers.

The second column shows that the **timing of instructions** can take one or more instruction cycles. All processors, except Matrix-1, execute some of their functions over multiple processing cycles. Executing slow computations such as multiply in two processor cycles allows the clock period to be reduced thus improving processor performance for faster instructions.

Table 2.3 also shows how data memory varies from processor to processor, though, only the MIMD processors contain a significant amount of control memory.

	ALU Complexity	**Timing of Instructions**	**Local Memory**
B-SYS	8-bit ALU & comparator	Single-cycle	8×8-bit data & control
Matrix-1	32-bit ALU & Multiplier	Single-cycle	4-K word data
Blitzen	1-bit ALU	Two-cycle	1-K bit data
ISA/SISA	1-bit ALU, comparator & 16-bit multiplier	Two-cycle for multiply & compare	11×16-bit data 2×16-bit inst.
WAP	32-bit ALU	Two-cycle	
Datawave	12-bit ALU & Multiplier	Two-cycle	16×12-bit data 64×46-bit inst.
iWarp	64-bit adder/multiplier & 32-bit logic unit	Single-cycle	128-words

Table 2.3. PE functional features.

Data Communication Features

Table 2.4 shows several communication rationales. It can be seen from the first column that the number of communication ports, or the **communications degree**, ranges from 2 to 8 ports. The communication degree for the mesh connected architectures allows bi-directional data communication between neighbouring PEs, whereas the B-SYS and Matrix-1 processor allow only FIFO pipelining.

The **orientation of the ports** are evenly spaced to allow communication with neighbouring PEs only.

Finally, the third column shows that the **method of passing data** for the more complex architectures is data-driven/asynchronous in order to aid PE autonomy and to facilitate array performance.

	Communications Degree	**Orientation of Ports**	**Method of Passing Data**
B-SYS	1 I/P & 1 O/P	Linear	Synchronous
Matrix-1	1 I/P & 1 O/P	Linear	Synchronous
Blitzen	4 I/P & 4 O/P	8-Directions	Synchronous
ISA/SISA	4 I/P & 4 O/P	NESW	Synchronous
WAP	4 I/P & 4 O/P	NESW	Data-driven
Datawave	4 I/P & 4 O/P	NESW	Data-driven
iWarp	4 I/P & 4 O/P	Any	Synchronous or asynchronous

Table 2.4. Data communication features.

2.2.5.2 Conclusions

An important observation is that SAs that do not contain on-chip control memory tend to accommodate more PEs per chip than those SAs containing control memory. However, other factors such as functional complexity and data path width also have a significant impact on PE circuit area.

MIMD architectures, such as Datawave, can reduce PE circuit area by limiting the complexity of the functional units to integer based computations, though at the expense of functional flexibility. Even further reductions can be made by adopting bit-serial PEs, such as the ISA, SISA and Blitzen, at the expense of computational performance. The

Blitzen architecture is attractive because it can vary the number of bits used to encode data, thus improving performance for small magnitude data.

A major disadvantage of SAs is their method of synchronisation which incurs problems associated with clock skew [25, 34]. This problem is caused by the effects of centralised control and becomes more apparent as the number of PEs in a network is increased. The next section discusses clock skew in more detail.

2.2.6 Timing Issues.

In the previous section clock skew was identified as major concern causing problems of synchronisation failure, limited scalability and fixed processing time. This section elaborates on the causes of clock skew and the problems it causes.

The feasibility of implementing a digital signal processing algorithm is dependent on the ability to realise the required computing power. For example, a typical HDTV application, such as image coding, requires the execution of 100 operations per sample at a 100-MHz sample rate therefore requiring a performance in excess of 10 giga operations per second. Realising this processing speed using Datawave [83, 84] would require the use of forty-eight PEs. Therefore, a typical high speed systolic network will usually contain 10's if not 100's of identical processing elements in order to achieve the processing speeds required by the application environment. However, global timing is often a limiting factor in realising high speed arrays, to quote S. Y. Kung [47]:

> "Whereas the asynchronous model in wavefront arrays incurs a fixed time delay overhead due to the handshaking processes, the synchronisation time delay in the systolic arrays is primarily due to the clock skew which changes dramatically with the size of the array. This latter phenomenon will be a potential barrier in the design of ultra-large-scale synchronous computing systems."

The problems of clock skew have become more significant with the developments of reduced device feature size, achieving higher switching speeds, on increasingly larger chip area. These developments are most apparent in the currently developing field of wafer scale integration (WSI) SA design [45, 56, 101]. The aim of this technology is to assemble an entire network of chips on a single wafer of silicon, with the objective of avoiding the costs associated with individual chip packaging and the loss of performance associated with off-chip pin communications. However, this technology adds to the problems of synchronising large processor arrays [14]. The two primary difficulties are the fact that the clock must propagate relatively greater distances and the large fan-out associated with the clock signal.

Ideally, the clock signal is expected to arrive at all the modules on a chip at the same time. However, the same pulse may arrive at different modules with a time offset; this time offset is dictated by the resistivity of the wires, together with their capacitance, determining the rate at which the voltage can be driven onto a wire across its whole length. Clock skew causes two main problems:

- Firstly, synchronisation failure of the circuit occurs, caused by the clock signal arriving at the array PEs at differing times, resulting in unreliable circuit operation. This can be avoided by lowering clock rates and/or adding circuit delays to the faster circuits, both at the expense of slowing computation.
- Secondly, circuit scalability is limited because the effects of clock skew become more significant when the clock signal has to propagate longer distances such as in large VLSI or WSI circuits.

Other factors that can contribute to clock skew include the shape of the clock distribution network, the number of modules, the speed of modules, the load of the modules, the direction of data transfer with respect to the clock and the clocking scheme used.

Another significant problem is that the clock period is fixed to the worst case delay through the system. In SA architectures this causes significant loss of performance because fast processing functions, such as addition or logical operations, are fixed to the same worst case speed as that of the slowest function, such as the multiplier. Multi-cycling of these complex operations, such as multiply, can reduce this problem but not eradicate it.

2.2.6.1 Methods of Overcoming Clock Skew

A number of clock distribution strategies have been developed for use in VLSI and WSI technologies. One technique that aims to overcome clock skew is called a Multi-Level Buffer Tree [25, 42], shown in figure 2.3. This technique splits a single clock signal progressively, level by level, into a regular clock distribution tree in which the points of the tree directly clock the storage elements. Buffering is required at each level of the tree to standardise the load, with the current drive capabilities for each buffer determined by the number of buffers in the next level of the tree. The drawback of this design is that it is difficult to scale [34].

A more scaleable approach is to divide the chip into regions, and to distribute the clock signal to all of the regions, namely an H-tree [14, 41, 42]. This method distributes the clock signal via a hierarchy of "H" structures as shown in figure 2.4. In this configuration the clock signal is distributed to the rest of the chip by a centralised "H" structure from which the signal is transmitted to the four corners of the main "H". These four signals arrive synchronously at the corners and form the inputs to the next level of hierarchy, represented by the four smaller "H" structures. The ends of the these "H"s are used to clock the local logic.

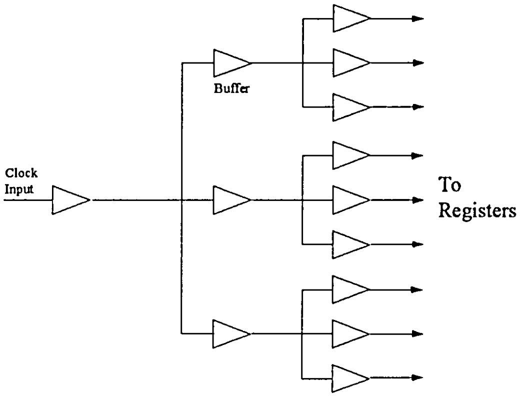


Figure 2.3. Multi-Tree buffer.

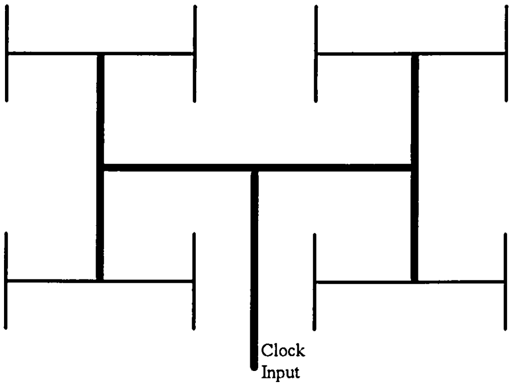


Figure 2.4. H-Tree structure.

It must be noted that by adopting a clock distribution strategy the effects of clock skew are reduced but not eradicated. As a consequence, an alternative asynchronous method of circuit timing known as self-timed control has evolved. The major difference between synchronous and self-timed control strategies is the way in which communicational and functional delays are handled as they propagate through a system [49].

In a synchronous design correct operation is ensured by asserting the signals within the system at a specific time, and for a specific duration. This time reference, known as the global clock, is calculated from and matched to the worst case delay through the system. The clock signal is distributed to all of the modules in order to ensure all computations and exchanges of information are synchronised.

In an self-timed design the clock is abolished and instead control is distributed to each individual circuit module. The modules themselves are responsible for correct system operation in the presence of communicational and computational delays, therefore signals within self-timed modules are asserted after some event, and are maintained until some other event. The next section presents the concepts that realise self-timed architectures.

2.3. Self-Timed Control

In this section a number of self-timed areas are reviewed and analysed. These are listed as follows:

- The handling of signals as they pass through a circuit module.
- The handling of signals as they propagate between circuit modules.
- The methodologies used to design self-timed architectures.
- The key self-timed architectures developed to date.

This section concludes with an analysis of the key self-timed architectures in order to establish the relative success and extent to which the techniques have been used to realise actual self-timed architectures.

2.3.1 Delays Within Circuits - Timing Models

There are three models of self-timed circuit-module operation, each exhibiting differences in the way signals are handled on their constituent gates and interconnecting wires:

1. The Bounded-Delay model assumes that all delays throughout the system are calculated and bounded.

2. The Speed-Independent model assumes arbitrary delays in gates but wires have no delay.
3. The Delay-Insensitive model assumes correct circuit operation in the presence of arbitrary delays in gates and on wires.

Each of these models are explained in more depth in the following sections.

2.3.1.1 The Bounded-Delay Model

This approach to circuit design assumes that all wire and gate delays are known, uniform, and bounded [26, 35]. This is the same model that is used for synchronous circuits. In this model only nodes at the boundary of the Finite State Machine (FSM), figure 2.5, are used to represent the state of the module. The boundary nodes for a combinational circuit include primary inputs, secondary state inputs (feedback from the previous process cycle), primary outputs and secondary state outputs (feedback for the next process cycle).

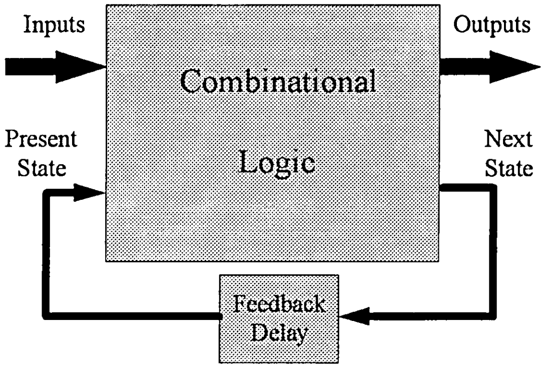


Figure 2.5. Sequential FSM.

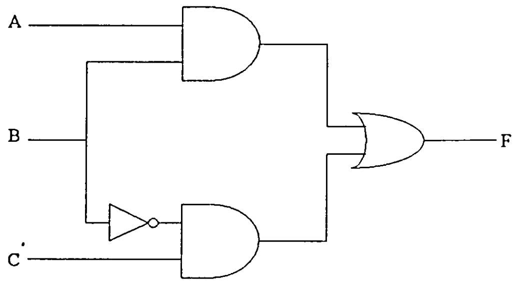


Figure 2.6. Circuit delay hazard.

There are several concerns when implementing self-timed bounded delay finite state machines which do not occur when implementing synchronous systems. Firstly, since there are no global clocks to synchronise input arrivals, signals can arrive at any time, and so the system must behave correctly in the presence of multiple input changes. For a two input circuit example, the system must not move directly from input '00' to '11' as this may cause the circuit to produce erroneous output, otherwise known as a dynamic hazard, but instead must briefly pass through '01' or '10'. Also, the output state must remain stable while these inputs transitions occur.

Secondly, care must be taken to ensure that delay-hazards are not present in the circuit. To illustrate a delay-hazard the following example is used. The circuit depicted in figure 2.6 will introduce a hazard when a change takes place on input B, assuming all gates have a gate delay associated with them. Assuming the current state of the circuit is (A,B,C)=(1,1,1), if we now set B='0' then the delay in the invertor will cause the top AND gate to become false, before the lower becomes true, and a '0' will propagate to the output, F. This is known as a delay-hazard and can be eliminated by ensuring the sum-of-products circuit doesn't contains a variable and its complement.

In order to cater for sequential circuits, a model similar to that used for synchronous circuits must be adopted thus forcing several requirements on the circuit. Firstly, the combinational logic must be stable in response to a new input before the present state entries change. This is achieved by placing delay elements on the present state feedback lines. This restriction applies also to the next-state inputs. The final requirement is that the next external input transition cannot occur until the entire system settles to a stable state. These restrictions are what define a fundamental mode circuit. The fundamental mode of operation [90] requires that the circuit may receive new inputs only after all of the circuit's nodes have stabilised. These rules are listed as follows (> means 'before'):

1. All inputs become defined > All outputs become defined
2. One input becomes undefined > All outputs remain defined
3. All inputs become undefined > All outputs become undefined
4. One input becomes defined > All outputs remain undefined

The bounded-delay model discussed has been successively applied to synchronous systems, however, there are some common problems that restrict its use in self-timed design. Firstly, the bounded-delay circuit modules must assume worst-case delays thus leading to worst case behaviour.

Secondly, the throughput of a bounded-delay self-timed system is extremely poor due to the severe timing restraints that must be enforced on the system components [26].

Finally, synchronisation failure in a synchronous system can be easily rectified by clocking the chip at a slower rate. However, in a self-timed system there is no clock to slow down and a delay fault can cause incorrect circuit operation that cannot be fixed. Techniques to ensure that no delay faults arise tend to be very complex and time consuming. As a consequence of these problems, the difficulty in designing a complex pipelined microprocessor is enormous, and the performance of such an implementation would be extremely poor.

2.3.1.2 The Speed-Independent Model

The first assumption of the Speed-Independent model [26, 66] is that all gates have arbitrary and unknown delays. This assumption means that the state of the circuit has to be represented in a different manner to the bounded delay model. In this approach signal values on all nodes within the combinatorial circuit represent the state of the circuit module and all gate transitions are carefully ordered to constrain the circuit's state transitions. This can be realised by constraining the interaction of the circuit with its environment, such that new values on the circuit's input can only be supplied in response to a change in its output. Sietz has specified a set of rules that describe a sequence of events, called the Input-Output Operational Mode [90]. These are listed below for a multiple input and output circuit module (> means 'before'):

1. One input becomes defined > One output becomes defined
2. All inputs become defined > All outputs become defined
3. All outputs become defined > One input becomes undefined
4. One input becomes undefined > One output becomes undefined.
5. All inputs become undefined > All outputs become undefined
6. All outputs become undefined > One input becomes defined

To ensure correct operation a second assumption states that within a circuit module the same signal propagating along different branches of a forked wire must proceed at the same rate. If this assumption was not enforced then a gate on one end of the fork may switch before the gate on the other fork, thus producing a delay-hazard.

Implementing this assumption, such that a signal propagates at the same speed along all wires, is complex. In response, a stronger assumption has been made stating that all wires within a circuit must incur negligible delays. A region within an integrated circuit in which all wires have negligible delays is called an isochronic region, and a branched wire within this region is called an isochronic fork [3, 90]. The rules for the input-output mode are relaxed in that a change on a single node on the output may cause a change on the input, therefore the adoption of an isochronic region for the entire circuit is essential. However, the realisation of isochronic delay assumptions can be difficult to realise in practice and is usually dependent on the implementation technology, wire lengths, gate construction and switching thresholds, therefore the isochronic constraint may require fault testing.

2.3.1.3 The Delay-Insensitive Model

Signals in a delay-insensitive circuit [9, 26, 76] are assumed to be arbitrary on both gates and wires. Therefore, this model requires no assumptions about isochronic regions or the delay of the combinational circuit to ensure correct operation. To signal the commencement of computation in a delay-insensitive module a supervisory sub-circuit is required that monitors each of the inputs and flags when each changes state [9]. A second component of this circuit then detects when all of the inputs are in the same state, then signals to the module to begin its function. Therefore, the operation of a delay-insensitive module must correspond to the fundamental mode of operation described earlier.

As no delay assumptions are applied to data as it propagates within or between gates, or between circuit modules, a new method of passing data is necessary. In a bounded-delay circuit the logic level of a wire is assumed to be correct by a given finite time, and thus can be processed at that given time. In a delay-insensitive system no assumptions can made as to when a wire will reach its desired value, thus data must be handled very carefully. A single wire cannot be used for level-encoded items, as two consecutive logic '0's cannot be distinguished from one another. Therefore, two wires are necessary to permit safe transfer of data. This approach not only transmits data but also communicates to self-timed modules to co-ordinate the progress of a computation. The handling of signals between circuit modules is considered in more detail in section 2.3.2.

The problem with the class of delay-insensitive circuits is that it is very limited because some of the more complex circuits are difficult to implement efficiently. However, any circuit is possible if the class of delay-insensitive modules is augmented with the isochronic forks [60, 67] assumption resulting in a reasonable quasi-delay-insensitive compromise. A C-element incorporating an isochronic-fork is investigated by Kees van Berkel [3].

2.3.1.4 Analysis of Models

This section has shown that there are a number of ways to implement self-timed circuits that have differing gate and wire assumptions. These assumptions are summarised below in table 2.5.

Model	**Gate Delays**	**Wire Delays**
Bounded	Bounded	Bounded
Speed-Independent	Unbounded	Bounded
Delay-Insensitive	Unbounded	Unbounded

Table 2.5. Summary of delay assumptions.

Of these methods the bounded-delay model, in its purest form, is the most difficult and time consuming model to implement because of the strict timing assumptions that are physically difficult to implement. However, a design method called micropipelines redresses the problem by combining a bounded-delay data path with a delay-insensitive control circuit (to be discussed in section 2.3.3.3). Also, the bounded delay model tends to limit concurrency, thus, efficient data paths in general cannot be built. Therefore, the bounded-delay model addressed here will no longer be considered.

The delay-insensitive model is the most attractive as it promotes truly scaleable circuits, in the presence of arbitrary unbounded delays at any level, and leads to easier design. The class of delay-insensitive circuits is limited, though a compromise is possible through the adoption of speed-independent techniques. Delay-insensitive circuits are also self-checking because any circuit faults or stuck-at-faults will eventually cause the entire system to lock up, thus automatically providing complete fault coverage [11].

2.3.2 Delays Between Circuits - Signalling Protocols

The correct handling of signals, as they propagate between circuit modules, is essential to guarantee the correct progression of computation and/or information. In a self-timed system where there is no global clock any transfer of information must be co-ordinated between the communicating circuits via the use of a signalling protocol. The reader is referred to the bounded signalling protocol [26, 75, 100], otherwise known as the data-bundled protocol, and the unbounded signalling protocols [1, 9, 12, 107].

2.3.2.1 Analysis of Signalling Protocols

There are two main ways to implement the co-ordination of data transfer between communicating modules. The data-bundled approach is attractive because it is closely related to the synchronous method in that it only requires one wire to encode each bit. However, a degree of re-design is necessary when scaling between implementation technologies, or porting modules to other designs, because of the need to re-calculate time delays for each request wire.

The second approach is attractive because it assumes unbounded delay assumptions for all wires, consequently no redesign is necessary when scaling or porting circuit modules. The disadvantage of this approach is that each bit must be encoded using two wires resulting in designs that consume more circuit compared with equivalent synchronous or data-bundled implementations.

2.3.3 Self-timed Design Methodologies

There are three main circuit design methodology classifications that cover the majority of self-timed circuits developed to date.

- The first uses a **graphical specification** [7, 8] as the starting point to eventually derive a verifiable set of logic equations and circuit structures.
- The second technique is based upon compiling a **high level language** [28, 59] down to a VLSI circuit.

The third is a more informal approach that uses a library of **macro-modules** [100] that can be connected together by a designer to form circuits of micro-processor complexity.

As there are numerous approaches to design self-timed circuits only the predominate method for each area is considered in depth.

2.3.3.1 Graphical Specification: Signal Transition Graphs

The signal transition graph (STG) [7, 8, 65, 108, 109] is presently the most amenable and widely used graphical approach to self-timed circuit design to date. It does not suffer from the same problems as the Petri-net which is restricted by a set of components (otherwise known as *places*) which are difficult to transform into an actual circuit realisation [26].

A STG describes circuit behaviour by specifying the precedence of transitions in a circuit, where transitions are interpreted as rising and falling transitions of signals of a control circuit. A control circuit can be said to consist of input signals which are caused by the environment, and output transitions that are caused by the circuit itself. STGs consist of a number of causal relations, for the various signal types defined above, where an arc is drawn between an input and output transition thus depicting the causal relation.

A hierarchical system design extension to the STG model has been developed that describes a composition procedure for the composition of deterministic STG circuit specifications. However, there is very little evidence to date that this work has been actually used to design complex microprocessor systems. The reader is referred to P.N.Lam *et al* [52].

2.3.3.2 High-Level Language : Communicating Sequential Processes (CSP)

In this method a computation is described as a set of communicating sequential processes (CSP) [59, 71], in a notation very similar to that to C.A.R Hoare's CSP [28]. The source program is then compiled in a series of semantic-preserving transformations resulting in a circuit which is correct by construction. The compilation process is made up of four steps; namely process decomposition, handshaking expansion, production rule expansion and operator reduction. These compilation steps are described as follows:

Process Decomposition

The first step replaces each process by a number of several semantically equivalent processes, resulting in a program of simple assignments and communication commands.

Handshaking Expansion

The second step replaces each channel by a pair of wire-operators.

Production-rule Expansion

This step compiles the handshaking expansion into a set of production rules from which all specific sequencing has been removed. By matching those rules to the operator semantics the program can be identified with networks of operators.

Operator Reduction

This last step consists of identifying sets of production rules in the program with sets of production rules which describe the operators. The program can then be identified as a set of circuit operators.

The system level design is completed by "mechanically" [61] interconnecting the various circuit modules together. As a result, there is no information available on the system level design using Martins CSP technique.

2.3.3.3 Macro-modules: Micropipelines

This approach to self-timed design is closely related to the widely-used hierarchical black box design approach adopted by most synchronous hardware engineers. It relies upon the existence of a library of basic building blocks, or standard cells, which can be assembled together by the designer to form much more complex self-timed systems. This requires that the functionality of each macro-module be exactly specified to the designer. The interface for each library module is usually considered delay-insensitive in order the keep the timing considerations away from the designer.

The predominant method in this category is the Micropipelined [75, 100] approach developed by Ivan Sutherland. Although the Micropipeline's data path is closely related to the bounded-delay model, it differs in that the data path is controlled by a delay-insensitive handshaking circuit. The micropipelined handshaking protocol adopts transition signalling, where the change between two logic levels is recognised as a signalling event, whereas the data is still represented using the conventional level-encoded logic states.

To implement the delay-insensitive handshaking protocol, two wires, called request and acknowledge, are used to connect the sender to the receiver in addition to an arbitrary number of data lines. In a communication, the sender places a value on its data lines and then sometime later produces an event on its request line. Some arbitrary time later the receiver accepts the data and then replies to the sender with an event on its acknowledge line. As can be deduced from this protocol the delay in transmitting data from the sender must be less than the delay in transmitting the request event; otherwise the request could arrive before the data is valid. This convention is said to be bundled because the request, acknowledge and data lines must be treated together when considering the delays in the circuit.

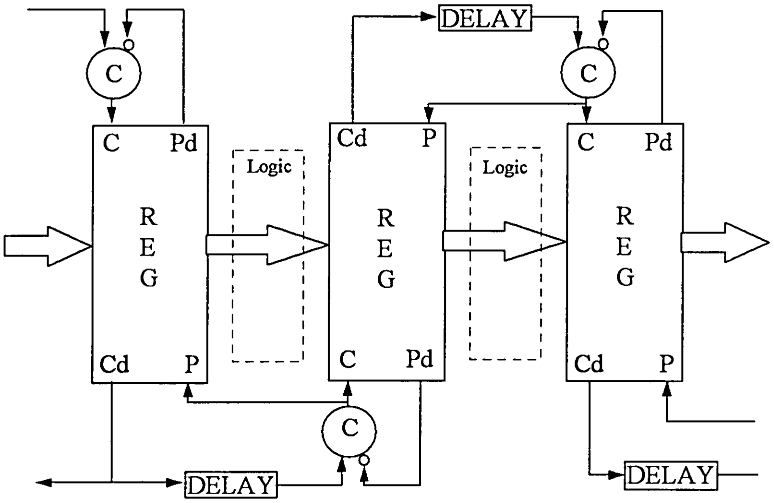


Figure 2.7. Micropipeline containing storage elements and optional combinational logic.

The other major element required for a micropipeline design is the event-driven storage device. Figure 2.7 shows several instances of the storage element each labelled 'REG'. Each storage element has two event inputs called capture (C), which render it transparent, and pass (D), which latches a data bit, and two outputs called capture done (Cd) and pass done (Pd) which are corresponding completion signals for C and D. In the event of a capture, a Cd event from the predecessor must be synchronised with the Pd event from the current capturing element. To pass data, the Cd from the successor must monitored by the current passing element.

In order to perform an AND on events the conventional set of Boolean logic gates cannot be used, as these devices cannot remember their previous inputs. Instead, a C-element [100] is used that only outputs an event when all inputs enter the same logic state. The C-element is utilised to synchronise the arrival of the Cd and Pd signals to signal the commencement of a capture phase. Figure 2.7 also shows how these storage elements are connected together to form a register pipeline. The hashed boxes between the storage elements labelled 'Logic' show how processing logic can be added to form a computational pipeline.

Interconnecting Micropipelines is based on the modular building block, or ad-hoc, approach, consequently, there is no information available regarding the system level design of micropipelined architectures.

2.3.3.4 Analysis of Self-timed Design Methodologies

In this section three of the main methods of designing self-timed circuits have been presented. Making a clear comparison between the various methods in terms of the resultant circuit area and performance is difficult, however, there are a number of clear differences that can be distinguished.

Firstly, only the STG and CSP methodologies allow the specification to be transformed into actual circuit realisations. Efficient algorithms for STG synthesis [106] and CSP synthesis [59] are well established and understood. Complex system-level design using the STG method results in over complicated graphical specifications [26].

Micropipelined design suffers in that there is no guidance on how to design complex systems. Current micropipelined architectures are generally designed in an 'ad-hoc' fashion, as no synthesis technique is currently available.

In conclusion, it is very difficult to assess which methodological approach is the best to adopt for the design of circuit modules and system-level architectural design. As a consequence, the extent to which these methods have been used to realise actual processor implementations must be addressed. This will establish the current state and extent to which the various self-timed design methodologies have been used in 'real-world' practical design situations.

2.3.4 Review of Key Self-timed Architectures

This section reviews a number of key architectures that use the previously presented self-timed design methods. The review of each architecture will concentrate upon the design (or specification) method, the data encoding technique, circuit implementation and architectural complexity.

2.3.4.1 The First Asynchronous Microprocessor [61, 62]

The pioneer asynchronous self-timed processor architecture was developed and transformed using the CSP [28, 59] design method explained in section 2.3.3.2. The focus of the research was to demonstrate the automatic compilation technique, rather than the processor architecture. Therefore, the architectural complexity is straightforward containing one simple 16-bit processing unit RISC processor, including 12 registers, four buses and an ALU. Other features include separate data and control paths, no delayed branches, and globally shared buses, resulting in a tightly-coupled architecture. Circuit implementation was realised using speed-independent circuit elements, and the data encoding technique was realised using four-phase dual-rail encoding. The processor was fabricated in 1.6μm technology achieving a speed of 18 MIPS using 20,000 transistors.

2.3.4.2 The Self-Timed RISC Processor [10]

The Self-Timed RISC processor is an advanced delay-insensitive architecture using the same CSP design method developed by Martin. The architectural complexity is more complicated than the first asynchronous microprocessor including a data processor (including register bank, ALU and decoder), and branch processor (including two decoders and program counter) that communicate through self-timed FIFOs serving as decoupling buffers. Both the branch processor and data processor communicate with the instruction memory, while only the data processor communicates with the data memory. The data encoding technique uses four-phase dual-rail signalling protocol for both data and control. Circuit modules are realised using delay-insensitive combinational logic, master-slave registers, finite state machines and FIFO register elements, as described in [9]. The advantages of this architecture include decoupled data and branch processing, variable delayed branches, acknowledge-less channels to remove redundant control signals and combined data-path and control. No information relating to cost or performance was available at the time of printing this thesis.

2.3.4.3 Amulet 1 [18, 19, 20]

The Amulet 1 project was established to investigate the potential power reduction due to an asynchronous implementation. The main goal of the project was to apply the micropipelined design method, adopting the two-phase data-bundled encoding technique, to an existing synchronous design of microprocessor architectural complexity, namely the ARM6; a RISC chip developed by ACORN Computers Limited.

The Amulet 1 consists of a number of pipelined sub-units, namely: the address interface (issues read and write requests), the register bank [73] (localised data storage that utilises a novel register locking mechanism), the execution unit [21] (ALU and multiplier that operate at an average rate of computation) and the data interface (reads data from memory and redirects it to the instruction pipeline, register bank or address interface). The major difference between the amulet's pipelining and that of the ARM6 is that the pipelining is elastic due to the asynchronous nature of operation. This elasticity allows the supply and demand of information between the various sub-units to be decoupled, thus allowing a high degree of overlapping execution.

The Amulet 1 circuit has been implemented, using the library of macro-modules defined by Sutherland [100], in 1μm technology achieving a performance of 9K dhrystones at 83mW using 58,374 transistors, as compared to the ARM6 which achieves 14K dhrystones at 75mW using 33,494 transistors.

2.3.4.4 The Non-Synchronous RISC (NSR) Processor [6, 77]

The NSR processor is a general-purpose 16-bit RISC machine developed using the Micropipeline design method. Circuit implementation is realised using Micropipelined macro-modules that adopt the two-phase data-bundled data encoding technique. The architectural complexity corresponds to a conventional RISC machine such that the instruction fetch, instruction decode, execute, memory and register bank operations are pipelined. In addition, each of these units is decoupled, like Amulet 1, through self-timed FIFO queues allowing a high degree of overlapping instruction execution. The NSR processor circuit implementation was realised using seven field programmable gate arrays with a best-case performance of 1.3 MIPS draining between 40.8mA and 53.1mA.

2.3.4.5 TITAC [68]

The architectural complexity of TITAC is simple in that it corresponds to a conventional load, compute and store 8-bit von Neumann microprocessor developed using a macro-module design method. TITAC circuit implementation is realised using quasi-delay-insensitive (adopting the isochronic-forks assumption) circuit elements, using the four-phase dual-rail data encoding technique, though some control-path elements adopt a simple request/acknowledge wire pair.

Two control structures were implemented for comparison. The first is a hardwired controller (C1) where the control structure was implemented by replacing, or mapping, the primitive elements of the dependency graph with one of five primitive circuit-elements (refer to page 56 of [68]). The second technique adopted a microprogrammed controller (C2), though, as no delay-insensitive memory was available 1-bit to dual-rail conversion was necessary. Combinational circuits, in the data path and control path, are implemented by transforming a binary decision diagram (BDD) representation of a function into a multi-levelled dual-rail implementation. Results have shown that C2 requires almost three times the number of gates as its synchronous counterpart. There is no synchronous counterpart for C1.

TITAC uses 5,517 cells and has been fabricated using 1 μm technology achieving a peak-speed of 11.2 MIPS with a peak power consumption of 212mW for C1, and 1.8

MIPS with 70mW for C2. A by-product of this research produced a library of building blocks for design automation.

2.3.4.6 The CounterFlow Pipeline Processor (CFPP) [94]

A CFPP, currently being developed by Sproul et al, uses a bi-directional Micropipeline design, adopting the two-phase data-bundled data encoding technique method, in which partially executed instructions and results move in counter flowing directions. A general CFPP architecture connects an instruction fetch unit at the bottom with a register unit at the top with a decoupled instruction execute pipeline in-between. Advantages of this architecture include localised control, regular structure, local communication, modularity and simplicity. This architecture is still under development.

2.3.5 Analysis of Key Self-Timed Architectures

This section analyses the key features of the previously presented architectures for the following issues:

1. The extent to which the various design methodologies have been used.
2. Identification of relationships between the processor complexity and the design methodology.
3. The scalability of the architectures.

The self-timed review section has briefly introduced the key architectures. This section presents an analysis that represents the features in a more accessible manner. **Table 2.6** shows the general features for the reviewed architectures. From this it can be seen that the micropipeline design method seems to be the most widely adopted approach for various processor complexities, with very little application of the formal methods in microprocessor design. The micropipeline architectures predominately adopt two-phase signalling protocols using techniques developed by Sutherland [100].

No evidence as yet has been found of purely delay-insensitive processor architectures, however, TITAC re-addresses this balance by adopting quasi-delay-insensitive circuit modules. Consequently, due to the bounded or isochronic timing assumptions, none of these architectures are truly architecturally scaleable without a degree of re-design.

Finally, it is extremely difficult to identify which performance characteristics are due to the design method and which are due to the architecture and/or technology.

	Design Method	**Processor Complexity**	**Signalling Protocol**	**Circuit Delay Assumptions**	**Fabricated ?**
FAM	CSP	Simple	Four-phase	Speed-Independent	Yes
ST-RISC	CSP	Complex	Four-phase	Delay-Insensitive	No
Amulet 1	Micro-pipeline	Very complex	Two-phase	Bounded	Yes
NSR	Micro-pipeline	Complex	Two-phase	Bounded	Yes
TITAC	Macro-module	Complex	Four-phase	Quasi-Delay-Insensitive	Yes
CFPP	Micro-pipeline	Not known	Two-phase	Bounded	No

Table 2.6. General features.

2.3.5.2 Conclusions from Analysis

The analysis of the key architectures has established that there has been little widespread progress made in designing and implementing complex programmable processor designs using formal design methodologies [7, 59]. CSP and STGs have been used to implement a small number of simple processor designs, however the vast majority of the complex architectures have been implemented using the macro-module approach. In fact, N.Paver [72], a co-designer of Amulet 1, identified a number of drawbacks with formal self-timed design techniques which are still apparent today. The drawbacks of formal techniques are as follows:

- They are limited to the size, type and complexity of circuit they can process.
- The resulting circuit carries significant overhead relating to area and performance.

The application of these techniques has not progressed sufficiently so that they can be generally adopted for the design of complex self-timed architectures.

The evidence shows that adopting any of the formal design methodologies will hinder the development of a novel complex self-timed array architecture at this time. These formal methods are still in their infancy in terms of their application to complex real-world architectures. Therefore, a less formal approach, based on a macro-module methodology, must be adopted to realise a complex self-timed SA architecture.

2.4 Thesis Objectives

The review chapter has shown the need for a study in the design of self-timed SA architectures as a result of the problem of array scalability, attributed to clock skew, and the current lack of work in self-timed SA processor design. This study must provide a basis for the evaluation of existing synchronous and self-timed design techniques for the implementation of array processing elements.

No information could be gained during the review regarding the trade-offs of self-timed design methods, compared with traditional synchronous techniques, for SA architecture realisation. It is therefore important to perform a study, comprising a series of investigations, to establish self-timed design strategies for the implementation of SA architectures. The study must characterise the nominated self-timed implementations in terms of circuit area, performance and design trade-offs compared with one another, and also against their synchronous counterparts.

A significant number of self-timed architectures have been constructed. However, it is very difficult to establish the reasons for the variations in cost and performance. To perform this study it is essential to perform investigations upon a single SA architecture, or research vehicle, thus providing a fair comparison of the existing synchronous and the new self-timed strategies. The suitability of the research vehicle must be categorised in terms of its ability to fully investigate the potential benefits of self-timed design.

Identification of Objectives

Most complex self-timed architectures to date have been developed using ad-hoc or macro-module methods, while recent research has focused on circuit synthesis and formal specification techniques. Consequently, there is very little understanding of the trade-offs between the various design strategies adopted for array architectures.

To address these issues this thesis presents a methodological study of array and processing element architectural design. Two primary aims have been identified:

The first aim is to derive a coherent design framework to determine the differences between implementation techniques. Three objectives are identified.

1. To determine requirements for the implementation of self-timed architectures when used for SA design.
2. The development of novel self-timed architectural structures based on the requirements developed in (1) which lead to scaleable and modular array designs.
3. To characterise the implementation characteristics in terms of area and performance for the existing synchronous and novel self-timed architectural designs and to compare their relative trade-offs.

The second aim is to establish strategies for the design of self-timed architectures utilising the framework from the first aim. Two objectives are identified:

1. To produce guide-lines for the design of new self-timed architectures which do not require re-design when ported between architectures or circuit technologies.
2. To produce guide-lines for the selection of self-timed or synchronous circuit design techniques for a given set of criteria.

This thesis aims to provide a detailed analysis in the design of self-timed array architectures using a common design framework and methodological study. This will lead to an understanding of the potential of self-timed techniques for general array implementation.

2.5 Conclusion

This chapter has provided a review of systolic array architectures and self-timed design techniques. A number of conclusions can be made from this review:

1. There are a wide variety of SA architectures that differ in circuit cost and performance, the reasons for which are due to their method of programming (SIMD or MIMD), data path implementation (Serial or Parallel), PE interconnection (Mesh or Linear) and synchronisation of data transfer (Synchronous or Data-driven).
2. A SA can be either globally or locally controlled resulting in globally synchronous or locally controlled data driven PEs, respectively.
3. Global timing of SAs causes problems of clock skew and limited scalability.
4. Self-timed control abolishes the clock therefore overcoming clock skew and its associated problems.
5. An architectural scaleable SA, that does not require redesign when ported between implementation technologies, can only be realised with delay-insensitive self-timed techniques.
6. Formal methods for self-timed design are still in their infancy consequently the macro-module approach predominates.

The merits of adopting self-timed techniques for SA design is unclear. As a consequence, a number of research objectives issues were identified in section 2.4 for further study, so that the potential of self-timed SA design can be more clearly understood.

CHAPTER 3

EXPERIMENTAL OVERVIEW

3.1 Objectives of the Chapter

The purpose of this chapter is to identify and describe experiments to address the thesis objectives. The second purpose of this chapter is to describe the experimental methodology and experimental assumptions used and referenced from the later investigative chapters. The chapters aims are as follows:

- A description of the experimental objectives is provided.
- The experimental methodology to be utilised by each investigation is described.
- The assumptions shared by all the experiments are stated.
- The Mentor Graphics tool suite and VHDL language used to model and simulate the newly constructed architectural models is presented.
- The experimental chapters aims, procedures and intended outcomes are summarised.

3.2 Objectives and Methodology of the Experiments

In the review chapter a number of fundamental areas were identified where very little knowledge exists in the systolic self-timed processor design domain. As a consequence a number of topics must be considered, listed below, that address the thesis objectives identified in chapter 2:

1. The requirements and methods for implementing self-timed SA architectures.
2. The realisation of self-timed SA architectures using the requirements and methods established in (1).
3. The circuit area and performance characteristics of self-timed SA architectures.
4. The circuit area and performance trade-offs between self-timed and synchronous SA architectures.
5. Guide-lines for the design of self-timed SA architectures.
6. Guide-lines for the selection of self-timed or synchronous circuit design techniques for SA design for a given set of criteria.

3.2.1 Scope of the Study

One important aspect of the review chapter is the wide variety of different systolic array implementations that are available. Most of these architectures, due to historic developments, are synchronous in nature, therefore the advantages and disadvantages of self-timed systolic array architectures are not known.

Another important key aspect of the review chapter is the number of self-timed design methodologies available, again each having their relative advantages and disadvantages. An exhaustive study comprising all of these architectures and self-timed design methods would be beyond the scope of this project. Thus, the scope of this study is limited to a single architecture, or research vehicle, that is the most amenable to the strictest self-timed design technology.

There are two levels at which the study can take place - at the hardware level and at the application level. The aim of this project is to propose design strategies for self-timed systolic array architectures, not to analyse the relative effects of self-timed implementation on the large number of systolic algorithms. Thus, the investigation is limited to hardware implementation.

There are a wide variety of self-timed implementation techniques available to realise self-timed circuits, however the majority of these assume that each circuit module resides within an isochronic region. Such techniques [60] would be unsuitable for this project because they are not easily scaleable between different fabrication technologies.

Only techniques that realise truly scaleable unbounded intra- and inter-module delays will be considered. Recent work by Lloyd [57, 58, 111, 112] has proposed a number of scaleable dual-rail RTZ and NRTZ modules that correspond to the implementation conditions of our self-timed architecture. Therefore, both of these implementation techniques have been adopted for this research resulting in RTZ and NRTZ versions of the research vehicle. This will enable a comparison to be made between these two technologies in terms of cost and performance.

3.2.2 Experimental Methodology

In this section the experimental methodology employed throughout the project is described.

The research aim is to establish the trade-offs of adopting self-timed techniques as compared with synchronous techniques in systolic arrays. This can only be achieved through adopting a common research vehicle architecture to ensure a 'level-playing field' in which to perform the investigations. The research vehicle, namely the Self-Timed Single Instruction Systolic Array (ST-SISA), is based on that of an existing synchronous architecture called the Single Instruction Systolic Array (SISA); the reasons for its selection are explained in chapter 4. There are a number of areas such as data signalling, data communication and architectural structure that differ from the SISA implementation due to the self-timed nature of operation; also considered in chapter 4. These differences require analysis and investigation and thus are addressed in the appropriate experimental chapters.

The experiments can be sub-divided into the data path, control path and processing element investigations. Each experiment was undertaken according to a standard methodological approach tailored to fulfil the thesis objectives. The methodology describes an ordered procedure in which each thesis experiment was undertaken (step 2 is omitted if the experiment does not require any hardware models to be constructed).

1. Identify the fundamental requirements of the experiment and the investigations that need to be performed to satisfy these requirements.
2. Develop hardware models of the architectures according to common design rules and assumptions (described in section 3.3). Adopt the following standard developmental procedure to produce these models.
    a) Identify the requirements of the function independent of implementation technique.
    b) Identify the requirements of the function when considering the self-timed design techniques. This is sub-divided as follows:

i) Identify the requirements of the function when considering effects of the fundamental mode of self-timed operation and variable length words.

ii) Identify the requirements of the function when considering the RTZ/NRTZ self-timed encoding and end of word (EOW) tagging techniques (see section 3.4.1).

c) Derive the new RTZ and NRTZ architectures using the requirements derived in b(i) and b(ii).

3. Establish the existing synchronous implementation for the function.
4. Simulate the newly built architectural models in order to verify their correctness.
5. Measure the cost and performance characteristics of the simulated architectural models.
6. Determine design and selection guide-lines.

The techniques and tools used to model the newly developed self-timed architectures are presented in the following section.

3.3 Experimental Assumptions

The purpose of this section is to state the assumptions to be adhered to by the experimental chapters. Therefore, a number of issues have to be addressed in order to satisfy the following:

1. Selection of a suitable research vehicle in which to perform the proposed investigations.
2. Selection of a suitable macro-module library in order to realise the new self-timed architectures.
3. A method of modelling the macro-modules using a high-level behavioural language.
4. An environment in which to construct the new architectures.
5. A simulator in which to test the newly built architectures and to establish their performance characteristics.
6. Selection of a suitable implementation technology in order to characterise the self-timed and synchronous architectures such that their performance and area characteristics can be easily compared.

3.3.1 Simulation Assumptions

The self-timed designs are constructed from a library of building block macro-modules (Appendix D) previously specified by Lloyd [57, 58, 111, 112]. The reasons for the selection of this library are as follows:

- The library is readily available.
- The library is tried and tested.
- The circuit speed and area of the library modules are known.
- Simulation of the macro-modules is straightforward.

The synchronous SISA model used in this project is taken from the original SISA constructed by ISATEC. However, the SISA circuit designs illustrated at the level of abstraction shown in this thesis cannot be referenced as they have never been published by ISATEC. The information for these illustrations was obtained during a two week visit to ISATEC. Therefore, references are made, where possible, to circuit designs that relate closely to the SISA circuits depicted in this thesis. The bit-serial latching mechanism adopted for the SISA conforms to the single-phase clocking technique described by Schimmler [81], and is further elaborated in [63, 91].

The simulation of the library modules was achieved by specifying the behaviour of each module using the VHDL language provided in the Mentor Graphics tools suite, namely the IEEE standard System-1076 (VHDL) Very high speed integrated circuit Hardware Description Language [69]. The timings extracted from the standard library were used as parameters in the behavioural description for each module, thus allowing the timing of each function to be specified, therefore any timing source, such as manufacturer specific timings, could also be used

The self-timed architectures were realised by interconnecting the appropriate macro-modules, using a standard system construction methodology (Appendix D), in a schematic design environment called Design Architect; a package in the Mentor Graphics CAD suite. This allowed the whole system architecture to be described pictorially. Sample schematic diagrams for the RTZ and NRTZ ST-SISA architectures are shown in Appendix A and Appendix B, respectively. Sample Macro-module VHDL code is shown in Appendix C. The self-timed architectures were then simulated using Quicksim II in order to verify their correctness and to establish their performance characteristics.

The timings for each module were specified using a relative unit of measurement, namely the delta (Δ) delay, where a Δ delay is equivalent to the average switching time (low-to-high and high-to-low transitions) for a standard two-input NAND gate. Therefore, the timing of the system components and architecture is not dependent on a particular implementation technology. Another reason for using the Δ delay is that the latency of all the modules in the adopted library have been measured in these delays therefore accurate performance simulations are possible.

3.3.2 Design Assumptions

Delay Assumptions

The choice of delay assumptions is essential to the design of self-timed architectures. These assumptions are critical to the cost, performance and ease of design. So far this project has highlighted the delay-insensitive timing model as the strictest and most easily scaleable model to realise self-timed circuits. Therefore, self-timed architectures implemented using delay-insensitive timing operate correctly in the presence of unbounded inter- and intra-module line delays and unbounded intra-module gate delays; the fundamental mode of operation.

The reasons for adopting delay-insensitive timing are as follows:

- The resulting circuits are technology independent, therefore the design will be scaleable between different technologies.
- The resulting architecture operates correctly independent of time delays at any level, thus resulting in a straightforward and portable design.
- Co-ordination and control of the circuit elements is sequenced and not time dependent allowing the replacement of any circuit module without any form of system redesign.

As a consequence of adopting this timing model all information must be dual-rail encoded in order to satisfy the delay assumptions. There are currently two methods of encoding data on dual-rail lines, namely the RTZ and NRTZ encoding techniques. The macro-module library consists of modules for both of these techniques. Both the RTZ and NRTZ techniques are used to implement the ST-SISA, thus resulting in two versions of the ST-SISA.

For many implementation technologies, such as CMOS, the signal delay through any circuit element is dependant on the direction of the signal transition; a high-to-low transition takes a different amount of time than a low-to-high transition. As the time taken to produce new output is determined by the direction of a transition then the magnitude of the data on the dual-rail input(s) determines the speed of the circuit. Therefore, the switching speed of a circuit is determined by the logical state of the input and so is data-dependent. In this work it is assumed that the number of high-to-low and low-to-high transitions averages out over time, consequently, all delays are average case measurements.

Instruction Set Selection

The SISA instruction set includes a wide variety of arithmetic, logical and comparison instructions. However, due to the time constraints of a three year one man research project it was not possible to implement the complete SISA instruction set for the ST-SISA. Therefore, a survey was undertaken to establish the most commonly used SISA programs.

This survey concluded that matrix transposition and matrix sorting constitute the majority of SISA programs, with a smaller number of computational based programs such as matrix multiplication. As a result of this survey, only the essential instructions such as READ, AND, OR, ADD, MIN and MAX compare were implemented. For simplicity it is assumed that these functions operate upon unsigned integer data. This instruction set covers a wide enough range of applications to make the ST-SISA viable.

3.4 Introduction to the Investigations

This section summarises the investigations that address the thesis objectives, identifying the questions for each investigation.

3.4.1 The Research Vehicle Study

This study was undertaken in order to select an existing synchronous architecture that would benefit from self-timed implementation, thus forming the self-timed research vehicle, namely the ST-SISA. This suitability study involved identifying an existing architecture that exhibited clock skew, fixed worst case performance and restricted scalability in the expectation of resolving these problems via self-timed implementation. In addition, due to the circuit area overhead of self-timed circuits an architecture was selected that would not significantly add to overall cost of the circuit design. Therefore, it is essential to adopt a bit-serial architecture. However, this led to a problem of how to signal the end of a bit-serial word in the absence of a global timing mechanism.

As a consequence this investigation required the identification and development of EOW tagging techniques, thus, allowing the length of a bit-serial word to be varied dynamically. Other important factors to be considered by this chapter include identifying any differences required in the instruction set, programming and architecture of the self-timed research vehicle in comparison to its synchronous counterpart.

These studies enabled a common architectural design to be adopted throughout the remaining investigative chapters. Also, the main architectural features of the research vehicle's processing elements constitute the remaining experimental chapters.

3.4.2 The Data Path Investigation

The data path investigation consists of three main experiments namely the register bank unit (RBU), data output unit (DOU) and execute unit (EXU) experiments. Each of these experiments has four main aims, explained as follows. The first aim is to develop self-timed and encoding guide-lines that aid in the design of the data path function. The second main aim is to develop new RTZ and NRTZ architectural structures that realise the function. The third main aim is to determine circuit area and performance trade-offs between the new RTZ and NRTZ architectures, and existing synchronous architectures. The final aim is to establish selection guide-lines that aid in the selection of the appropriate design technique for the function for a given set of criteria.

The main design issues particular to each of the three above mentioned data path experiments is elaborated in the following sections.

The Register Bank Unit

The main design issues of the RBU are concerned with methods of controlling and synchronising the reading of data from, and writing data to, the register bank, and writing data to, and reading data from, its registers. These issues must consider the effects of the self-timed operation, the encoding and tagging techniques, and the dynamic length of data words.

The Data Output Unit

The main design issue of the DOU experiment is the transfer of the existing SISA data communication protocol to the ST-SISA when considering the effect of self-timed operation and RTZ/NRTZ signalling techniques. Therefore, the main objective is to establish self-timed communication strategies and protocols for interfacing the internal PE data path to its external outputs accounting for the time variations associated with the unbounded timing model at both the PE and array levels.

The Execute Unit

The EXU has many functional operations including, assignment, AND, OR, ADD, MIN and MAX. As a consequence, this experiment consists of three sub-experiments that cover the development of each of these functions: logical (AND, OR), addition and comparison (MIN, MAX) experiments. The main design issues of each of these experiments is examination of the effects of the self-timed techniques on the functionality of each instruction, thus ascertaining any possible changes that needed to be made to the design functionality. In doing this the following questions were addressed for each function, when adopting the RTZ and NRTZ encoding and tagging techniques:

- What are the possible effects when varying word size in successive execution cycles?
- What are the effects when varying the length of the input data words in the same execution cycle?
- Is it possible that the length of the output word is in excess of the length of the input words?

- Is it possible to produce an output bit for every bit input?

3.4.3 The Control Path Investigation

The control path consists of six main functions. Three of these functions are common to both the new ST-SISA and the existing synchronous SISA architectures: instruction by-pass unit (IBU), instruction fetch unit (IFU) and instruction decode unit (IDU). The other three functions are only used by the ST-SISA: the overwrite unit (OWU), result control unit (RCU) and processing element status unit (PSU). These latter three functions are required because the ST-SISA operates in instruction cycles where each cycle is sub-divided into a number of dynamic bit-cycles. The number of bit-cycles is determined by the length of the longest data word operated upon by the data path. As a consequence, these functions are required to co-ordinate the various sub-functions of the ST-SISA for both bit and instruction cycles.

The development of these functions constitutes the six main experiments of this investigation. Each of these experiments has four main aims, similar to those stated for the data path experiments. The first aim is to develop self-timing and encoding guide-lines that aid in the design of the control function. The second main aim is to develop new RTZ and NRTZ architectural structures that realise the function. The third main aim is to determine circuit area and performance trade-offs between the new RTZ and NRTZ architectures, and existing synchronous architectures for each of the IBU, IFU and IDU functions. However, no synchronous comparisons are possible for the OWU, RCU and PSU. The final aim is to establish selection guide-lines that aid in the selection of the appropriate design technique for the function for chosen design criteria.

The main design issues for each of the six experiments is briefly explained in the following six sections.

The Instruction By-pass Unit

The function of the IBU is to send the instruction word to the IFU only if two corresponding control bits it also receives are set to the appropriate logic value, otherwise the instruction is deselected and by-passes the PE. The main design issue is concerned with the IBU receiving a differing number of bits on its instruction and selector inputs, causing problems of dead-lock in the fundamental mode of circuit operation, because each input must receive an equal number of state changes.

The Instruction Fetch Unit

The IFU must first convert the bit-serial instruction into the equivalent bit-parallel instruction to correspond to the synchronous SISA. It must also handle the regeneration of this bit-parallel word in order to control the various sub-functions of the data path for each and every bit-cycle of an instruction cycle. The main design issues here are concerned with the storage and regeneration of the instruction word for a dynamic number of bit-cycles until the end of the instruction cycle is detected.

The Instruction Decode Unit

The IDU controls the various functions of the data path. The main design issues are that the IDU must monitor the status of the data path in order to disable the appropriate functions which have completed their functions. However, the IDU must continue to control other functions that require to continue processing in the same instruction cycle; these status signals are provided by the PSU.

The Over Write Unit

The function of the OWU is to dispose of the old data item from the selected destination register in order to make space for the new data item. This function is considered to be part of the control path because it must provide a status signal on each bit-cycle to the PSU to identify whether the old data item has been discarded or not. The main design issue here is the continuance of signalling a new state change on the status output when no more old data item bits can be read from the selected destination register.

The Result Control Unit

The RCU monitors the result path between the EXU and the RBU looking for the end of the bit-serial result operand so that it can signal to the PSU the end of the EXU execution cycle. The main design issue of the RCU is similar to that stated for the OWU in that it must continue signalling a new state change on its status output to the PSU on every PE bit cycle though no result bits can be read.

The Processing Element Status Unit

The PSU unit monitors the status outputs from the OWU and RCU, in addition to the source data operand outputs from the RBU in order to signal the end of the instruction cycle. This function also provides status bits to the IDU indicating which parts of the data path it should disable and continue to control. The main self-timed design issue of the PSU is to continue sending state changes to the IDU in order to signal the status of the RBU source operand outputs which have been disabled by the IDU.

3.4.4 The Processing Element Investigation

The main aim of this experiment is to establish that the PE operates correctly when the data and control paths are connected together. The second objective of this experiment is to compare the cost and performances of the RTZ, NRTZ and synchronous PE architectures. This will provide valuable information on the overall processor performance for each executable instruction, assessing the trade-offs of the various design techniques and identifying any bottle-necks in the self-timed architectures. The third aim is to analyse the effect on RTZ and NRTZ processor performance in terms of the effect on PE instruction latency, when taking into account the variation in data communication and instruction execution latency of neighbouring PEs. The final aim is to suggest guide-lines to aid the designer in the selection of the appropriate design technique for a SA for required performance and area characteristics.

3.5 Conclusion

This chapter has identified the experimental objectives. It has stated that a common research vehicle must be used to ensure a 'level-playing field' in which to investigate these objectives. The simulation and design assumptions to which the experiments must adhere to have been established. Finally, the investigations that address the experimental objectives have been described.

CHAPTER 4

THE RESEARCH VEHICLE

4.1 Objectives of the Chapter

This chapter provides the background to the research. The research vehicle, the Self-Timed Single Instruction Systolic Array (ST-SISA), is described in terms of its control mechanisms, array structure, processing element structure, application and design methodology.

The ST-SISA is based on the architecture of the synchronous Single Instruction Systolic Array (SISA). This chapter begins by identifying the functional properties of the SISA architecture that must be included within the ST-SISA implementation. The encoding and end-of-word tagging techniques adopted for the self-timed implementations are defined. The ST-SISA is introduced including the array level interconnection, processing element (PE) functionality, programming and the adopted instruction set.

4.2 The Single Instruction Systolic Array (SISA)

The Single Instruction Systolic Array (SISA) [53] provides enormous computing power for mathematical based applications such as image processing and ray tracing by use of massive parallelism. The SISA architecture is a bit-serial mesh connected array where application flexibility is provided by the programming of three streams of control information which are systolically pumped through the array, as shown in figure 4.1; instructions along the columns and selector bits along the columns and rows.

A single instruction (made up of two words) is input to the top left PE from where it moves, step by step, in the horizontal and vertical directions through the array forming a computational wavefront. This guarantees within each instruction diagonal that the same instruction is active before being passed onto the next diagonal (E.g. Figure 4.1 - Shows the SISA containing, from right to left, read_west, read_north, add, read_west and read_north instructions). Instructions are selected or de-selected for execution by the logic value of the column and the row selector bits, if both are logic '1' the instruction is executed, if either are logic '0' the instruction is transformed into a NOP.

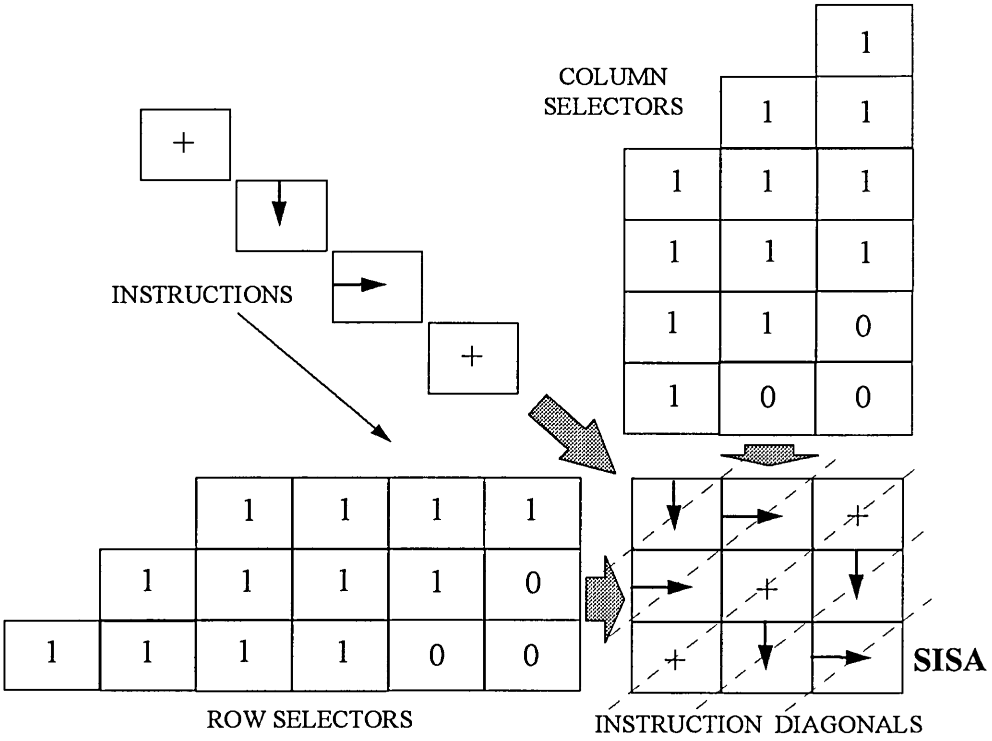


Figure 4.1. SISA control flow.

4.2.1 SISA Array Level Data and Control Interconnection

This section presents the interconnection schemes that implement control and data communication in the synchronous SISA. Figure 4.2 shows the input and output requirements for each SISA PE: north, east, south and west bi-directional data ports, instruction input and output, row and column selector input and output, word and bit clock inputs. Almost all information transfers are synchronised by the globally distributed bit-clock, except for the row and column selector transfers which are synchronised by the globally distributed word clock. Figure 4.3 shows how the PEs are interconnected to form an array.

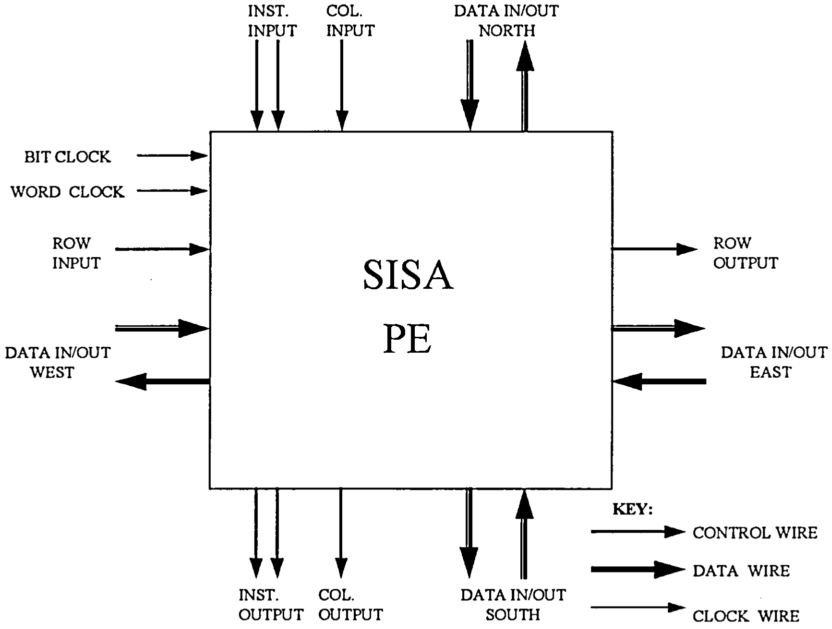


Figure 4.2. SISA PE input/output.

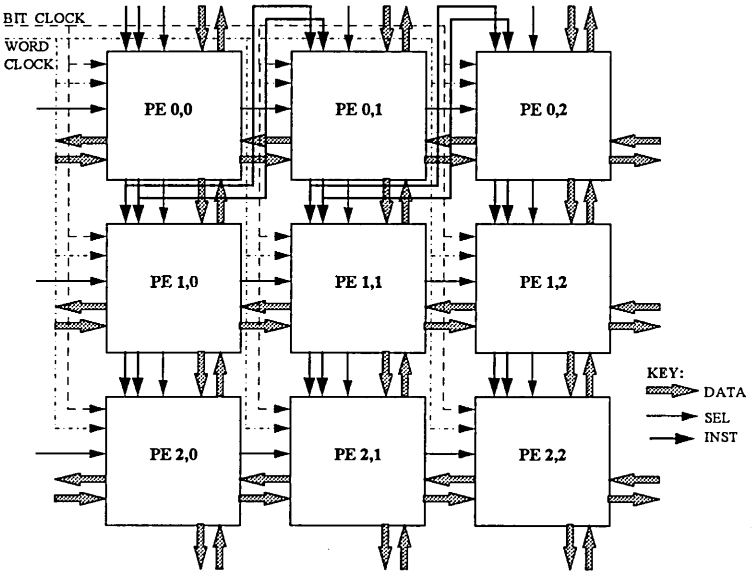


Figure 4.3. The SISA array.

4.2.2 SISA Processing Element Architecture

The following describes some of lower level functional details of the SISA, as depicted in figure 4.4. The execute unit (EXU) implements a wide variety of instructions including register assignment, logical, addition and comparison. The execution of each instruction takes one word clock, which consists of 16 bit clocks. The word clock is used to synchronise operations, such as fetch and decode, which only need to be done once for each word cycle. The bit clock is used to synchronise bit by bit operations, such as addition and register assignment. This necessitates that a bit clock and a word clock are broadcast to all PEs to synchronise internal bit by bit operations and instruction cycles, respectively. Consequently, each instruction takes the same execution time. Also, since every instruction is propagated to the neighbouring PE directly after its execution, PEs only need to buffer the present instruction. Therefore, they do not need their own control stores.

Each PE has a register bank unit (RBU) containing 11 registers each laid out as a 16-bit shift register. It passes on one or two operands, via its two addressable multiplexors RBU MUX A and RBU MUX B, to the EXU and receives results from the EXU via the RBU demultiplexor (RBU DMUX). The first register of the RBU is the communication register which can be read by the PE itself or by one, some or all of its four neighbouring PEs. Therefore, data communication is possible in the north, south, east and west directions on the execution of a read instruction. To avoid read/write conflicts read access is always performed before write access within the same clock cycle.

The instructions are read bit-serially and are kept in two shift registers, storing word 0 and word 1, from which control information is distributed to where it is used in the PE. The instruction buffers are also fed with the status of the row- and column-selector bits which mask their output to the instruction decoder. Only when both these selectors are logic '1' will control information be passed to the other functions of the PE, otherwise, no control is produced and effectively a no-operation (NOP) is performed.

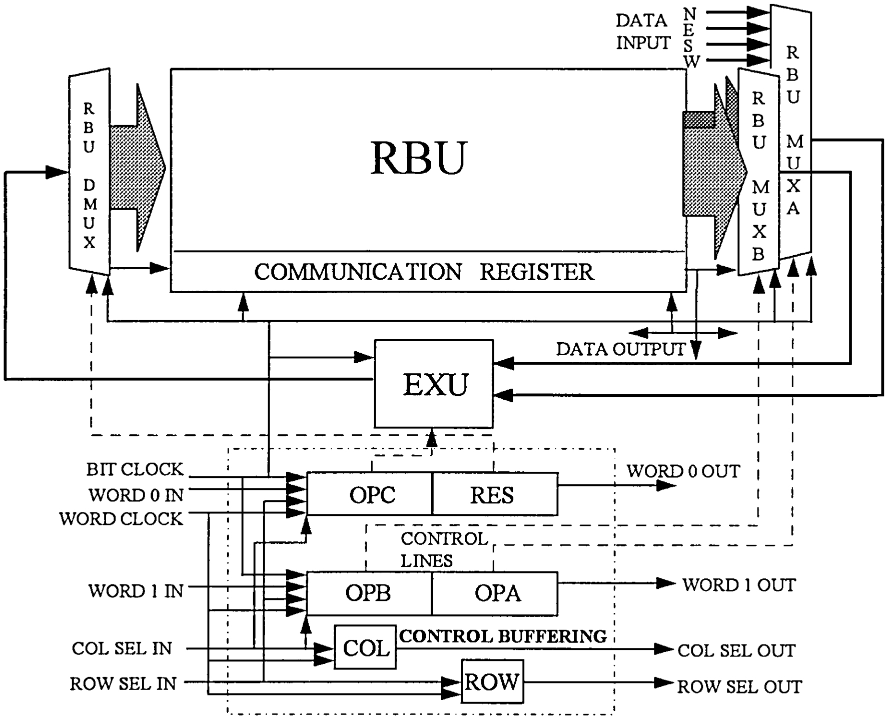


Figure 4.4. The SISA PE top level functions.

The advantage of the SISA over the ISA [81, 78, 105] is that it has a reduced instruction bandwidth because only one instruction is input into the array on each word cycle, as compared to many instructions for the ISA. However, the PE architectures for the SISA and ISA are exactly the same. This construction leads to a very flexible architecture which creates the possibility of very efficient solutions for a large variety of applications.

The SISA was selected as the research vehicle for this project for the following reasons:

1. High degree of parallelism - Facilitates high performance.
2. Locality of communication - Fits in well with the self-timing philosophy.
3. Fixed worse case processing speed - Due to the globally synchronous pipelined nature of the SISA all operations take the same worst case time to process. Adoption of self-timed control techniques should eliminate this problem.
4. One instruction input as opposed to *n* for the ISA.
5. Bit serial operation - Minimises interconnection lines and keeps circuit size down to a minimum.
6. Multi-purpose architecture.

4.3 The Self-Timed Single Instruction Systolic Array (ST-SISA)

This section begins by introducing the bit-serial encoding and tagging techniques adopted for the ST-SISA. The interconnection schemes required to implement control and data communication in the ST-SISA array are then presented. The processing element architecture and its major functional blocks are elaborated. The ST-SISA instruction set and field format are defined. Finally, an example ST-SISA program is explained.

4.3.1 Data and Control Communication Techniques

Two problems need to be addressed in order to realise self-timed communication in a bit-serial architecture. Firstly, encoding techniques must be identified that encode bit-serial data in a purely delay-insensitive manner; so that the arrival of new data can be detected in an environment where unbounded delays are assumed. Secondly, because there is no global timing reference in self-timed architectures the end of a bit-serial word cannot be signalled as for the conventional clocked SISA. Therefore, end-of-word (EOW) tagging techniques must be adopted in order to signal the end of a self-timed bit-serial word.

4.3.1.1 Encoding Techniques

Two dual rail encoding techniques could be used to realise delay-insensitive data communication in a self-timed bit-serial array, namely; return-to-zero (RTZ) [9] and non-return-to-zero (NRTZ) [12].

4.3.1.2 Bit-Serial End of Word Tagging Techniques

RTZ and NRTZ encoding provide self-timed communication techniques for synchronising bit by bit data transfer operations, therefore replacing the function of the global bit clock. The main use of these above mentioned techniques is in bit-parallel systems. However, neither of these approaches provides a facility to signal the end of a bit-serial word. Consequently, EOW tagging techniques are required for both RTZ and NRTZ encoding in order to make the ST-SISA feasible, and are considered in the following two sections.

4.3.1.2.1 RTZ End of Word Tagging

In the RTZ encoding technique valid data is encoded on two lines, using two-bit representations for data-true (1,0), data-false (0,1), and the empty state (0,0 = E). This leaves the unused (1,1) state to be utilised as a EOW Tag bit. This approach is attractive as it allows the **(T)**ag bit to be placed after any (V)alid bit in the stream, and before the (V)alid bit of the successive word, thus promoting variable length words (Example 4.1.).

```
TVEVEVTVEVEVEVEVEV
100101110001000100 -> F0
110000100100010001 -> F1
◄-W2-►◄----W1-----►
```

Example 4.1. RTZ EOW Tagging.

The conventional RTZ register cannot accommodate the illegal (1,1) state, so an alternative FIFO register design must be adopted. An appropriate alternative is the NRTZ register [58], comparable in size to the RTZ register, which can accommodate all dual-rail bit combinations. The advantages of this approach is that the EOW Tag can be placed at any position in the bit-serial transmission and taking the logical AND of the dual-rail is all that is required to detect the EOW Tag. Also, there is no encoding overhead associated with this tagging technique as a spacer token is simply replaced by an EOW Tag.

4.3.1.2.2 NRTZ End of Word Tagging

Signalling the end of a NRTZ bit-serial word is a much more complex problem than for the RTZ protocol, as no spare bit representation can be used. As a result the task of finding the end of a NRTZ stream is reduced to a process of counting bits or imposing a second encoding technique such as Huffman encoding [87]. Huffman Encoding is unattractive because data would have to be encoded and subsequently decoded at some later point resulting in a loss of performance and requiring extra circuitry. This overhead is present in all forms of encoding techniques. Consequently, for this work only the first approach is explored.

The method identified periodically reserves a bit position, of any EVEN phase, in the data stream to indicate the EOW Tag. For example, every fourth bit in the data stream in the example below (Example 4.2.) is reserved to indicate the EOW (T)ag. If this is set to logic '1' then the EOW is found, otherwise if it is set to logic '0' then more (D)ata bits for that word are expected. In this approach every fourth bit is simply ignored by arithmetic/logical functions and so the extra decoding circuitry required by the Huffman approach is not needed. In addition, unlike the RTZ method extra registers are not required to store empty states.

The disadvantage of this approach is that the word size is a multiple of the tag period. Therefore, in some cases bits preceding the EOW Tag bit are not used to encode actual data. These unused bits are simply encoded as logic '0' data bits and so can be ignored by arithmetic/logic functions.

```
TDDDTDDDTDDDTDDDTDDD
11000010101000110000 -> F0
10010111111101100101 -> F1
◄—W2—►◄—— W1——►
```

Example 4.2. NRTZ EOW Tagging.

By using a latched pipeline, with at least three latches, it is possible to connect the output of the last stage to the input of the first, forming a delay-insensitive ring [92, 93]. Therefore, if the ring is initialised to contain a pre-determined bit sequence consisting of a series of logic zeros ending with an EVEN phase logic '1' tag bit, a simple counter circuit is realised, as shown in figure 4.5. If the number of bits in the ring is matched to the tag period then the EVEN phase logic '1' tag bit in the ring can be used to detect the end of a word. For example, for each data bit processed the contents of the ring are also rotated by one bit. If both the ring output bit and the data bit are set to an EVEN phase logic '1' EOW Tag bit then the end of the word is detected. The main advantage of this approach is that for short tagging periods a saving in silicon area is made over that of conventional counters.

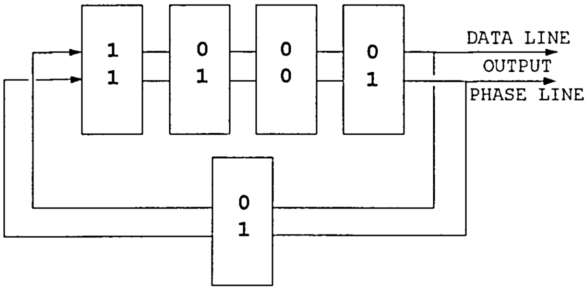


Figure 4.5. An NRTZ delay-insensitive ring for tag detection.

4.3.1.2.3 NRTZ End of Word Tag Period Selection.

To ensure data items are not inefficiently encoded, careful selection of the appropriate tag period is crucial. For example, if the magnitude of the data to be encoded is small then adopting a large tag period would be inappropriate as there would be a large number of unused bits.

Selection of the correct tag period is dependent on the maximum data word-length of the desired architecture. For the ST-SISA we have chosen to operate on small magnitude words, up to a maximum of 16-bits, therefore the tag period is expected to be small. However, the exact tag period to adopt is not clear, therefore it is essential to assess the efficiency of adopting various tagging periods for a 16-bit system. This involves calculating the percentage efficiency based on the ratio of the number of bits used to represent the magnitude of the data to the total number of bits, including EOW Tag(s) and unused data bits, required to encode the entire word. This information is depicted in figure 4.6 which shows the percentage efficiency of word lengths ranging from 1 incrementally to 16-bits adopting 4-, 6- and 8-bit tagging periods.

Figure 4.7 shows the average efficiency over various data bit ranges, including 1- 4 (ranging over the low order nibble), 13 -16 (ranging over the high order nibble), 1-8 (ranging over the low order byte), 9-16 (ranging over the high order byte) and 1-16 (the complete word range).

It can be seen from this graph that the 4-bit tagging period is the most efficient for the low order nibble and byte ranges. For the high order nibble and byte sized words the 6-bit tagging period is the most efficient. For the complete range of bits the 4-bit tagging period performs best on average. Therefore, it was decided to adopt the 4-bit tagging period for our design as this performs best in the average case, and on words that fit within the low-order byte and nibble ranges.

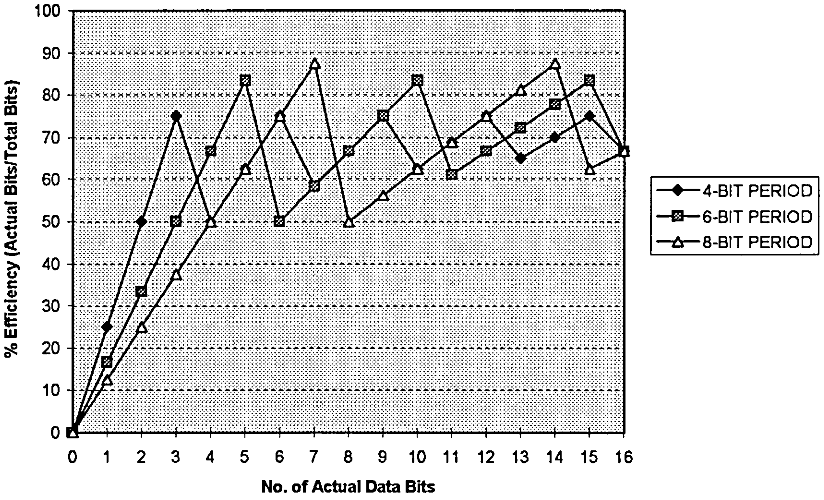


Figure 4.6. Percentage efficiencies using 4-,6- and 8-bit tagging periods.

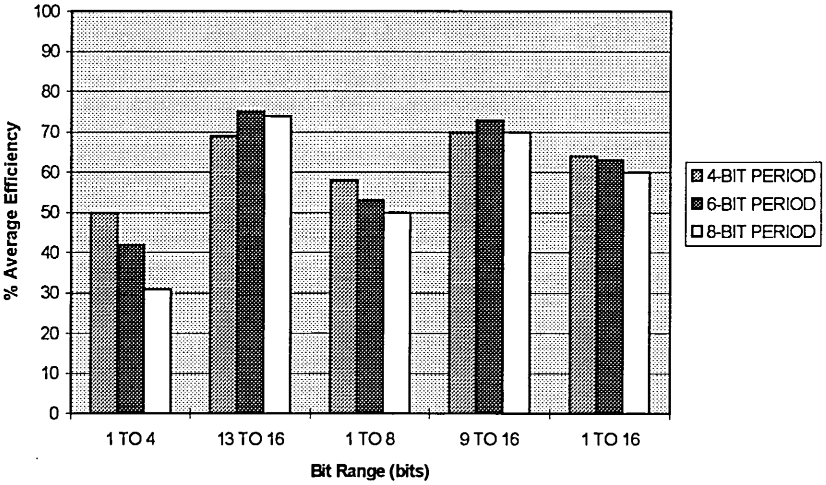


Figure 4.7. Average percentage efficiencies.

4.3.2 ST-SISA Array Level Data and Control Interconnection

The ST-SISA has three streams of control information which are pumped through the array, these being the instructions, and the row and column selector bits [30]. Data communication between neighbouring PEs is possible bi-directionally in the north, east, south and west directions [31]. In order to implement these communication channels a dual-rail line (two wires), and an acknowledge wire, is required for each PE bit-serial input and output. Figure 4.8 shows the input and output requirements for each ST-SISA PE and figure 4.9 shows how the PEs are interconnected to form an array.

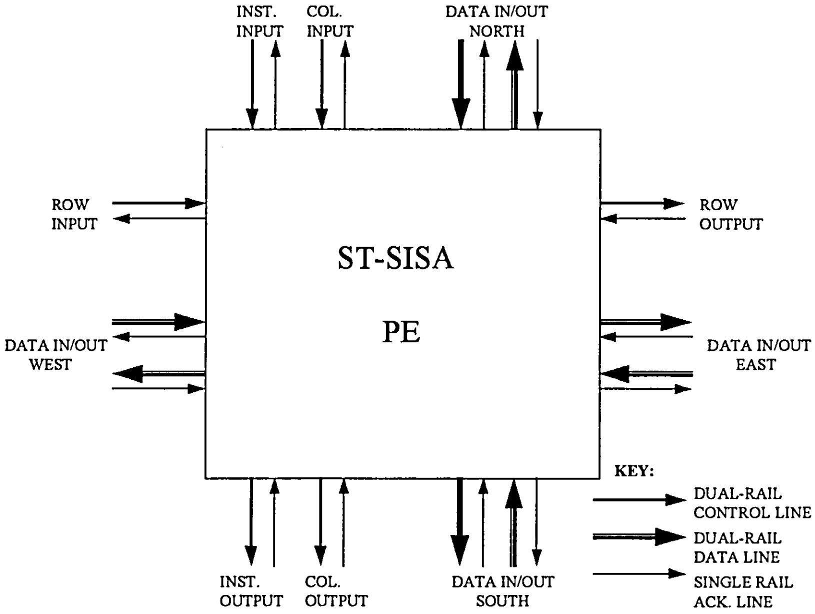


Figure 4.8. The ST-SISA PE.

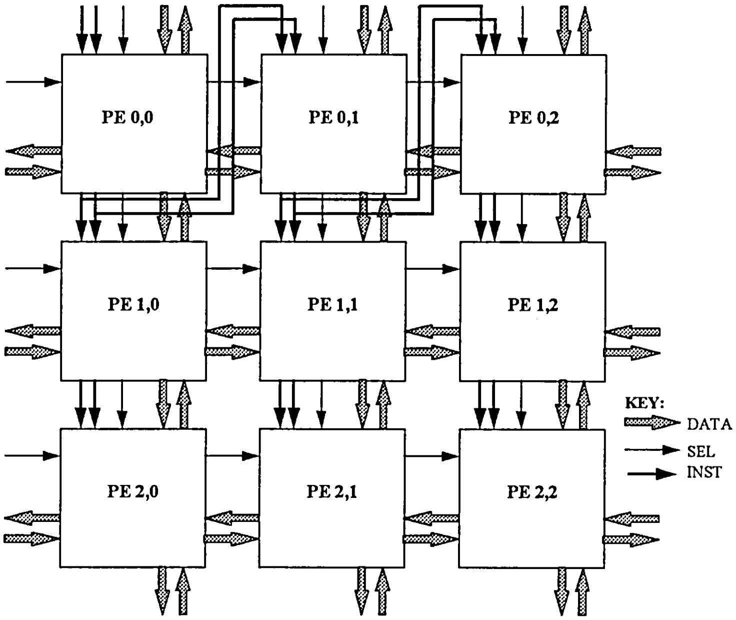


Figure 4.9. The ST-SISA array.

4.3.3 ST-SISA Processing Element Architecture

The ST-SISA PE is made up of several blocks as shown in figure 4.10. Each block of the processor accepts data from its inputs, processes it in accordance with its function and outputs its data in First-In-First-Out (FIFO) order.

These blocks produce output for each and every state change on their inputs. Therefore, to correspond to the rules of delay-insensitive design each function must receive a state change on all its inputs before an output can be produced, otherwise, the function remains in its current state. As a consequence this protocol requires that control information be re-generated for all the functions of the processor for each and every bit-cycle. The process of fetching and regenerating the control word is a function of the instruction fetch unit (IFU).

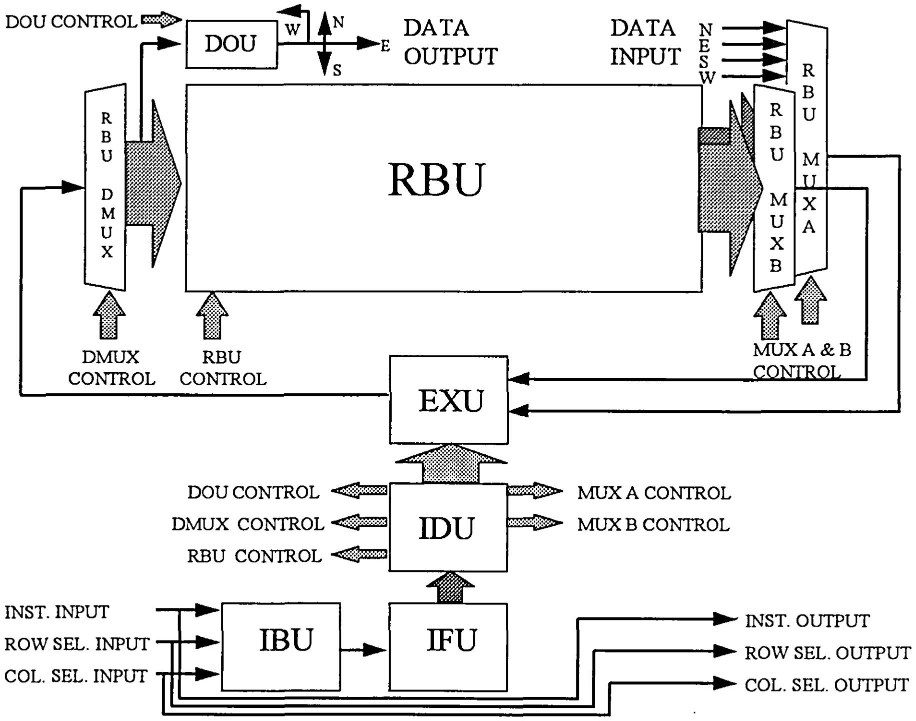


Figure 4.10. The ST-SISA processing element architecture.

A brief overview of the operation of the processor reveals a FIFO data path under the direct control of a FIFO control path. The operation of these is elaborated as follows:

The Control Path

The instruction bypass stage (IBU) reads the status of the column and row selectors, if both are set to logic ‘1’ then a copy of the instruction is taken from the instruction stream and passed on to the IFU, hence the instruction is selected for execution. If either or both of the selector bits are logic ‘0’, meaning that the instruction is not selected for execution, then the instruction is simply ignored. The IFU takes the bit-serial instruction and converts it into a bit-parallel representation before outputting it to the instruction decode unit (IDU). The IFU unit also stores the bit-parallel control word in order to re-send it to the IDU in successive bit-cycles. The IFU sends the next control word only when the appropriate completion signal is received. The IDU sends decoded register information to the register bank unit (RBU) and decoded instruction information to the execute unit (EXU) and data output functions.

The Data Path

One or both of the RBU multiplexors (RBU MUX A and RBU MUX B) read a source operand from one of the RBU registers before sending it on to the EXU. The EXU stage then performs an arithmetic, logical, comparison or assignment operation on the source operand/operands producing a result. The RBU's demultiplexor takes the result and then directs it to the appropriate register of the register bank. In the case when the result is written to the communication register (R0) a copy is also output to the appropriate neighbouring PEs via the data output unit (DOU).

ST-SISA Processing Element Functional Units

Details of the individual blocks of the ST-SISA PE are as follows:

The Instruction Bypass Unit (IBU)

The IBU takes as its input the row, column and instruction bit-serial control streams. If the instruction is deselected for the PE then one or both of the selector bits will be set to logic ‘0’. In this case the instruction is simply ignored by the IBU, therefore by-passing the rest of the PE architecture entirely. If both the row and column selector bits are set to logic ‘1’ then the instruction is passed onto the instruction fetch unit bit by bit. Also, each input to the IBU is forked and connected to the next respective successor PE for each respective control stream, thus, implementing the array level control interconnect.

The Instruction Fetch Unit (IFU)

The IFU takes the bit-serial instruction and converts it into a bit-parallel representation before sending it on to the IDU. This functionality corresponds to the SISA implementation. The IFU unit also stores the bit-parallel control word in order to re-send it to the IDU in successive bit-cycles. As outlined earlier, each function of the PE must receive a state change on all inputs before a new output is produced. Consequently, for each bit-cycle of the PE the IFU regenerates the same control word so that the IDU can re-apply decoded control information to each PE block. The IFU fetches the new control word when the appropriate status control signal is received from the register bank.

The Instruction Decoder Unit (IDU)

The IDU takes the parallel instruction from the IFU and generates the necessary control signals for the RBU, EXU and DOU. The IDU also accepts control status input from the RBU in order to disable registers that have been read and are no longer needed.

The Register Bank Unit (RBU)

The RBU (containing four registers for the purposes of this research) receives address information from the IDU. It passes on one or two operands, via its two addressable multiplexors, to the EXU and receives results from the EXU. Register 0 doubles as a communication register in that any data written to it is also sent to the DOU. Data stored in each individual general purpose register can vary in length from 1 to 16 bits.

The EXecute Unit (EXU)

The EXU receives its instructions in pre-decoded form from the IDU. It uses that information, and the necessary source operands from the RBU, to produce a result. The result is then routed back to the RBU. The execute unit performs several functions: AND, OR, ADD, ASSIGN, MIN and MAX. All except the MIN and MAX instructions produce a single output bit for every input bit. For these special instructions both operands are completely consumed before the result is generated and output to the RBU. This is because the appropriate word cannot be selected for output until the most significant bit of both source operands are compared. Therefore, both the MIN and MAX units require internal storage for two 16-bit data words.

The Data Output Unit (DOU)

The DOU takes whatever is input to the communication register R0 and outputs copies along dedicated output paths to the appropriate neighbouring destination PEs. It obtains its destination control information from the result operand field. In the event of a write instruction being issued each bit of the result address field is used to represent a directional select/deselect flag (bit0-North, bit1-East, bit2-South, bit3-West). Whenever a flag is set to logic '1' data is sent to the PE in that direction. This necessitates that a write instruction is identified by the IDU in order to address only the communication register.

4.3.4 ST-SISA Instruction Set

The ST-SISA is a 16-bit machine, implementing the simple instruction set shown in Table 4.1. Each ST-SISA PE operates solely on data stored in registers and data read from its four direct neighbours. There are four general purpose registers, one of which acts as the communication register (R0). Any data written to this special register, via executing the write instruction, is also copied and output to the appropriate neighbouring PEs, via the DOU.

Most of the instructions operate on two operands, addressable by Ra and Rb as shown in Table 4.1, including; AND, OR, ADD, MIN and MAX, with the WRITE, ASSIGN and READ instructions requiring only one source operand. The ASSIGN and READ instructions are realised using the same opcode since both assign data to an internal PE register. The only difference is that the source operand address field of the ASSIGN instruction references one of the internal PE registers whereas the READ instruction references an input from one of the neighbouring PEs. All instructions, except no-operation (NOP) which does not affect the state of the PE, produce a result (Rd).

No.	Mnemonic	Encoding	Action
0	NOP	000	NO ACTION
1	ASSIGN/READ Rd, Ra	001 -Rd- -Ra-	Rd ← Ra (internal or external)
2	AND Rd, Ra, Rb	010 -Rd- -Ra- -Rb-	Rd ← Ra AND Rb
3	OR Rd, Ra, Rb	011 -Rd- -Ra- -Rb-	Rd ← Ra OR Rb
4	ADD Rd, Ra, Rb	100 -Rd- -Ra- -Rb-	Rd ← Ra ADD Rb
5	MIN Rd, Ra, Rb	101 -Rd- -Ra- -Rb-	Rd ← Ra IF Ra < Rb ELSE Rb
6	MAX Rd, Ra, Rb	110 -Rd- -Ra- -Rb-	Rd ← Ra IF Ra > Rb ELSE Rb
7	WRITE R0, Ra	111 -R0- -Ra- -Rb-	R0 ← Ra

Table 4.1 - The ST-SISA instruction set.

The operations of the EXU stage are performed at a varying rate of computation dependent on the data word length and the complexity of the instruction executed.

Bit-serial data words input to the EXU stage not only vary in length from one instruction cycle to the next, but can vary in length with respect to one another in the same execution cycle. For example, an addition operation can add a word of 4 bits to a word of 12 bits producing a 13-bit result. This is the case for all arithmetic instructions. However, all functions must receive a state change on all their inputs every bit cycle in order for their function to be performed. This is clearly not the case if words vary in length, and will result in the circuit locking up because one of the inputs will cease to receive data bits for that word cycle. Therefore, dummy bits must be appended to the end of short words to ensure each input receives a state change on every bit cycle. A solution to this problem is presented in chapter 5.

ST-SISA Instruction Format

A ST-SISA instruction contains an operation code and address information for accessing register memory and performing I/O transfers. Figure 4.11 shows the format of the 16-bit instruction separated into two 8-bit words each containing two encoded sub-fields.

W0	**OPCODE** bits 0 to 3	**RESULT** bits 4 to 7
W1	**OPERAND B** bits 0 to 3	**OPERAND A** bits 4 to 7

Figure 4.11. The ST-SISA instruction format.

The first word (W0) encodes the instruction op-code and the address to which the result is to be written; whether it be an internal general purpose register or one or more of the four neighbouring PEs. Bits 0 to 3 encode the instruction to be executed. When a write instruction is executed the result field bits, 4 through to 7, are used to represent the North, East, South and East output directions of the PE, respectively. Therefore, logic '1' bits in the result field indicate to the DOU which neighbouring PEs are to receive the data.

The second word (W1) encodes the address of two source operands which are processed by the EXU. Operand A can address either one of the registers of the RBU or one of four external data inputs to the PE, whereas operand B can address only one register of the RBU. Figure 4.12 shows an example in which the result, produced by adding together the content of registers 1 and 2, is written to register 3.

W0 - 0100 0011

W1 - 0001 0010

Figure 4.12. Instruction encoding example.

4.3.5 ST-SISA Programming

Programming the ST-SISA replaces the requirement of correct timing and sequencing, necessary for synchronous architectures such as the SISA, with that of correct sequencing [29]. As can be seen from the example SISA program, depicted in figure 4.1, leading logical '0' selector bits are placed before the actual program bits; in selector rows 2 and 3, and selector columns 2 and 3. This reason for this is to maintain the correct timing of selector bits.

For ST-SISA programs, no leading logic '0' selector bits are necessary because PEs cannot perform their operation until all control information on their inputs arrives. For example, the first row selector bit for PE 0,1 (PE arrangement as seen in figure 4.9) and the first column selector bit for PE 1,0 cannot be read until PE 0,0 passes the column selector bit to PE 0,1 and the row selector bit to PE 1,0. In this respect, the ST-SISA operates much like the Wavefront Array in that control wavefronts propagate from the top left PE towards the bottom right PE. However, trailing selector bits in columns and rows 1 to *n* are required to ensure all control information is removed from the ST-SISA array before the next program is issued.

A study of SISA programs has identified that SISA programs fall into three main categories:

1. Communicational - Matrix transposition.
2. Computational - Matrix addition.
3. Conditional - Bubble sort.

4.4 Conclusion

This chapter has identified an architecture, deemed suitable for self-timed implementation, that exhibits all the problems associated with synchronous control. The new self-timed architecture, the ST-SISA, has been described in the following terms:

- Techniques to encode and signal the end of bit-serial words.
- Wire interconnection of the PEs.
- PE architecture.
- Programming.

The architecture described in this chapter is independent of the circuit implementation technology, assuming the technology uses dual-rail encoded circuit interfacing. However, the low-level implementation of the described architecture is dependent on the adopted dual-rail encoding technique; whether it be RTZ or NRTZ. Therefore, the effects of adopting the encoding and EOW tagging techniques introduced in this chapter must be investigated in the following experimental chapters in order to determine the differences of the ST-SISA architectures, and also for means of cost and performance comparison with the ST-SISA's synchronous counterpart.

CHAPTER 5

THE DATA PATH

5.1 Objectives of the Chapter

The objective of this chapter is to derive design guide-lines for the development of RTZ and NRTZ data path functions. These design guide-lines are developed through the design of the register bank unit, data output unit and execute unit architectural structures, constituting the second main aim of this investigation. The relative cost and performance characteristics for the new RTZ, NRTZ and existing synchronous data path functions can then be established. The final aim is to analyse these results in order to establish selection guide-lines that allow the identification of the most efficient design method for data path functions for a given set of criteria.

5.2 Introduction to the Data Path

The ST-SISA data path is shown in figure 5.1. It consists of several functions: a register bank unit (RBU), an execute unit (EXU) which includes ADD, AND, OR, MIN and MAX functions, and the data output unit (DOU). The operation of these data path units has been described previously in chapter 4.

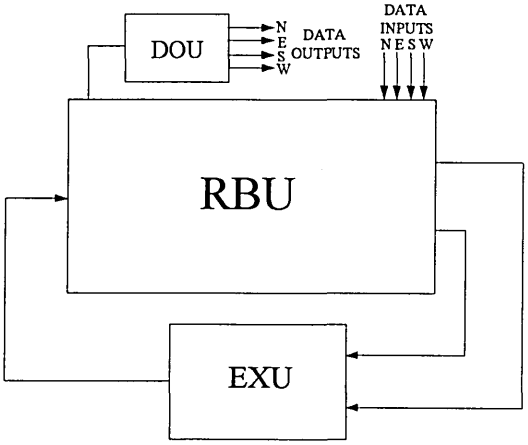


Figure 5.1. The data path.

The development of these self-timed RTZ and NRTZ architectures, and the comparison of their characteristics against each other and their synchronous counterparts, constitute the three main experimental sections of this chapter.

5.3 Experimental Methodology and Assumptions

To ensure consistency throughout the three proposed investigations in this chapter, and the following experimental chapters, the same experimental and design methodologies are adopted. The standard experimental methodology can be found in chapter 3 and the design methodology in Appendix D.

The RTZ and NRTZ ST-SISA architectural features investigated in this chapter are established on their corresponding synchronous SISA counterpart. However, it is not always possible to base the RTZ and NRTZ architectures directly on their synchronous counterpart due to the timing constraints of the self-timed design technique; as will be seen in the comparator experiment. Therefore, in such a case the new RTZ and NRTZ architectures have been developed to match the synchronous counterpart as closely as possible.

The self-timed architectures shown in this chapter have been modelled and simulated using the Mentor Graphics tool suite, discussed in chapter 3, in order to verify their correctness and to establish their performance characteristics.

The metrics used to assess the implementation characteristics of the RTZ, NRTZ and synchronous designs are listed below:

Latency delay: The time taken for a function to complete a word processing cycle (measured in delta delays (Δ)); for variable length data words for both RTZ and NRTZ architectures, and fixed length data words for synchronous architectures.

Throughput rate: The number of words that can be processed by a function in one Δ delay unit (1Δ/latency) for each possible word length.

Hardware: The area of the circuit in terms of the number of two-input NAND gate cell primitives required to implement the function.

It is important to note that the delay-insensitive functions addressed in this chapter are performed at a varying rate of computation. The rate of computation is not only dependent on the complexity of the operation but also on the length of the bit-serial data word. Therefore, the bit-serial data words input to a function can vary in length from one instruction cycle to the next, and with respect to each other within the same execution cycle. For example, an addition operation could be required to add a word of 4-bits to a word of 12-bits. Consequently, due to the variable nature of the input operands, and the fundamental mode of circuit operation, a common problem is recognisable in each function. This problem is stated as follows:

> All RTZ and NRTZ functional circuit modules must receive a state change on all their inputs in every bit cycle in order for their function to be performed. This is not the case, if words vary in length in the same instruction cycle, and thus results in circuit dead-lock because one of the inputs will cease to receive data bits for that word cycle.

In order to overcome this problem, in the fundamental mode, the input that reads the shorter word must still receive signal changes in order for the circuit module to produce its function. Therefore, dummy bits must be appended to the end of the shorter word therefore making both words of the same length.

5.4 The Register Bank Unit (RBU)

The RBU is shown in figure 5.2. Its role is to store variable length bit-serial data words in the PE, direct output data to the DOU and input data from one of the neighbouring PEs. It receives address information from the instruction decode unit in order to instruct it to read either one or two source operands via its two addressable multiplexors. The source operand provided by RBU MUXA is either read from one of the internal registers, or one of four inputs from the four neighbouring PEs, whereas RBU MUXB can read only one of the internal registers. The RBU addressable demultiplexor RBU DMUX receives the result operand from the EXU which is either written to one of the internal registers or to the DOU for external PE output.

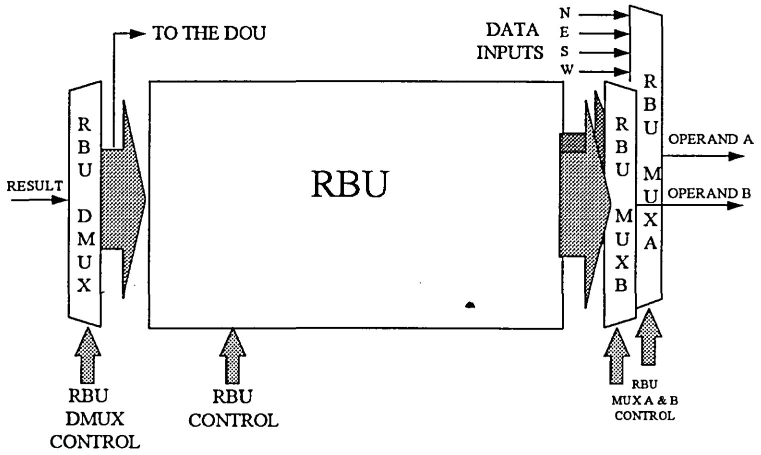


Figure 5.2. The register bank unit.

5.4.1 The Self-timing Requirements

To fulfil the functional requirements it is necessary to select the path of the data through the RBU and to synchronise the appropriate acknowledge signals between its various sub-functions. For example, if a register acts as a source to both RBU MUXA and RBU MUXB it must know when both multiplexors have read its output, so it can synchronise the appropriate acknowledge signals received from RBU MUXA and RBU MUXB. However, if a register acts as the source to only RBU MUXA then only the acknowledge signal from RBU MUXA needs to be monitored, not the acknowledge from RBU MUXB. Consequently, each register needs to know whether it is to act as the source to RBU MUXA and/or RBU MUXB, and/or the destination to RBU DMUX.

Due to this functional complexity each register requires local control circuitry to co-ordinate the synchronisation of the appropriate acknowledge signals for each of the above mentioned operational instances. There are eight combinations in which a register can operate in conjunction with MUXA, MUXB and the DMUX. These are as follows:

1. Register acts as a source to RBU MUXA.
2. Register acts as a source to RBU MUXB.
3. Register acts as a source to both RBU MUXA and RBU MUXB.
4. Register acts as a destination to RBU DMUX.
5. Register acts as a source to RBU MUXA and as a destination to RBU DMUX.

6. Register acts as a source to RBU MUXB and as a destination to RBU DMUX.
7. Register acts as a source to both RBU MUXA and RBU MUXB, and as a destination to RBU DMUX.
8. Register not used as either source and/or destination.

5.4.2 The RTZ Experiment

The RTZ Requirements

Data path components that operate on the EOW Tag bit are realised using NRTZ circuitry because of the need to cater for the EOW Tag bit. However, all local control circuitry operates on and generates RTZ encoded control information. The reason for adopting RTZ control codes is because the path of the data through the RBU cannot change until the first bit cycle of a new instruction, which is always a data valid bit cycle. As a consequence, because all control inputs to the data path functions are RTZ encoded, an empty phase control code automatically selects the same data path as that previously selected by the data valid control input.

The RTZ RBU Architecture

The RTZ RBU architecture is shown in figure 5.3.

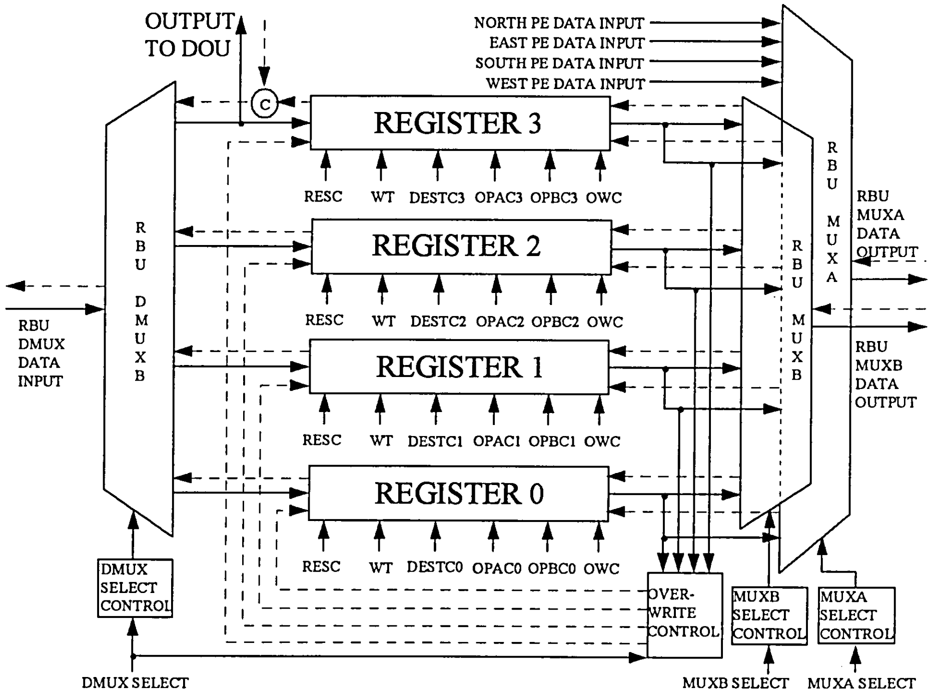


Figure 5.3. The RTZ register bank unit.

REGISTER (0, 1, 2 & 3): The purpose of each RBU register is to store a single data word, up to a maximum length of 16-bits. Data can be written to its input and read from its output. Specific control circuitry is required to co-ordinate data transfer between the register's write input and RBU DMUX, and/or the register's read output and RBU MUXA and/or RBU MUXB. Due to the complexity of a register, its architecture is shown separately in figure 5.4.

RBU MUXA: The role of RBU MUXA is to direct data from one of the RBU registers, or from one of the external inputs of the four neighbouring PEs, to the EXU, depending on the address placed on its control input.

RBU MUXB: This device directs data from one of the RBU registers to the EXU, depending on the address placed on its control input.

RBU DMUX: The role of this device is to direct the resultant data word received from the EXU to one of the internal registers for storage, or to the DOU for output to one or more neighbouring PE(s). This is controlled by the address placed on its control input.

DMUX/MUXA/MUXB SELECT CONTROL: The purpose of this device is to pass the address received from the IDU to the controlled RBU MUX/DMUX only if it is required to pass data. If the RBU MUX/DMUX is not used then the address is discarded to ensure only the required components of the data path are selected.

The RTZ Register Architecture

The RTZ Register Architecture is shown in figure 5.4.

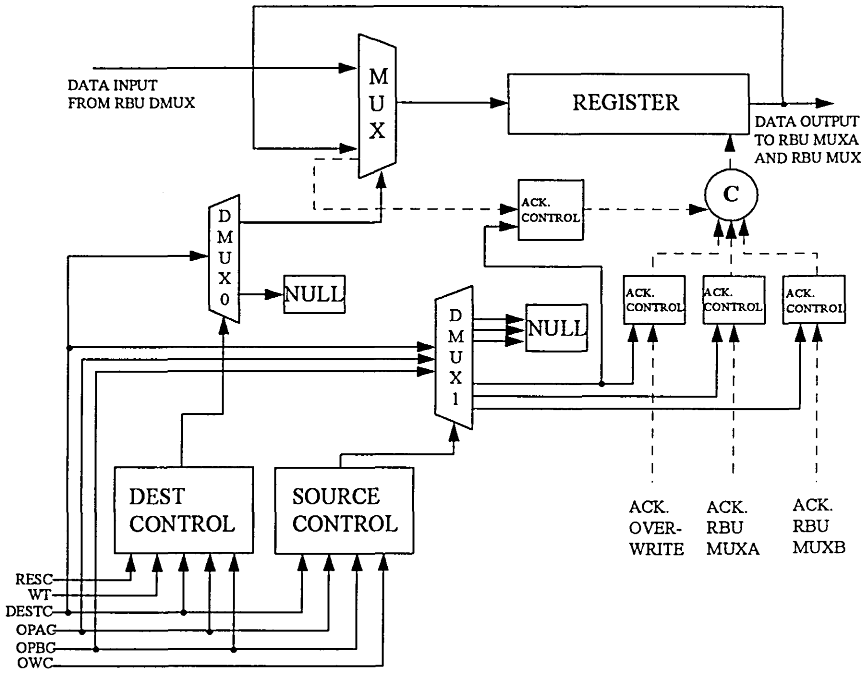


Figure 5.4. The RTZ register.

MUX: The role of MUX is to direct either the new data into the register, if it is selected as the destination for the result, or to re-write the register's old data back into itself, if it acts as the operand source; to maintain data integrity. Its operation is as follows. When the control input, controlled by control code DESTC, which is received from the IDU, is set to logic '1' the new result data received from the RBU's DMUX is written into the register. Otherwise, when it is set to logic '0' the original data stored in the register is fed back and re-written.

Destination (DEST) CONTROL: The role of this device, if the register is used as the source and/or destination, is to control the passing of the MUX control bit though DMUX0 to control MUX, otherwise the control bit is discarded in the null circuit. This ensures that MUX is not inappropriately enabled if the register is not used as the source or destination. Its operation is as follows. If any of the DESTC, OPAC or OPBC (all received from the IDU) control inputs are set to logic '1', indicating that the register is used as the destination and/or one or both source registers, then the control output is set to logic '1'; indicating to DMUX0 that DESTC is to be passed to the MUX control input. If DESTC, OPAC and OPBC are all set to logic '0', indicating the register is not used as either the source or destination, the control output is set to logic '0'; this signals to DMUX0 that DESTC is not required by MUX and can therefore be discarded.

If either WT or RESC are set to logic '1', independent of the logic level of DESTC, OPAC and OPBC, then the control output is set to logic '0' indicating to DMUX0 to discard DESTC. Otherwise, if both are logic '0' then the operation explained in the previous paragraph applies.

DMUX0: The function of this device is to pass DESTC to either the control input of MUX, if the control input is set to logic '0', or to the null circuit where it is discarded, if the control input is set to logic '0'.

SOURCE CONTROL: The purpose of this device is similar to that of DEST CONTROL except that it controls the passing of the control bits through DMUX1 to the three ACK CONTROL functions or the NULL circuit. Its operation is as follows. If DESTC, OPAC or OPBC are set to logic '1', indicating that the register is used as the source and/or destination, then the control output is set to logic '1' signalling to DMUX1 to send DESTC, OPAC and OPBC to their respective acknowledge select circuits. If DESTC, OPAC and OPBC are all set to logic '0' then the control output is set to logic '0' signalling to DMUX1 to discard all three control signals.

This circuit also monitors OWC. This signal when set to logic '1 indicates that the old data has not been overwritten in the destination register, otherwise when it is set to logic '0' this indicates that the destination register's old data has been overwritten. If the OWC input is set to logic '1' then the control output is set to logic '1' independent of the logic level of DESTC, OPAC and OPBC, otherwise when it is set to logic '0' then the operation in the previous paragraph applies.

DMUX1: The role of this component is similar to that of DMUX0 except that it passes the three ACK CONTROL control bits (DESTC, OPAC and OPBC), in parallel, to either the ACK CONTROL circuits, if the select input is set to logic '1', or to a NULL circuit, if the control input is set to logic '0'.

Acknowledge (ACK) CONTROL: The role of each of these circuits is to pass the acknowledge signal to the synchronising C-element, if the control input is set to logic

‘1’, or the internally produced pseudo acknowledge signal, if the control input is set to logic ‘0’.

5.4.3 The NRTZ Experiment

The NRTZ Requirements

All data path components are realised using NRTZ circuitry in order to cater for the NRTZ encoded and tagged words. In addition, all local control circuitry operates on and generates RTZ encoded control codes thus resulting in all control path components being realised in RTZ circuitry. The reason for adopting RTZ control codes is because the path of an NRTZ data word cannot change until the first bit cycle of a new instruction cycle, which is always an ODD phase. Therefore, a logic ‘0’ EVEN phase control input into a data path component automatically selects the same input/output path as that previously selected by the ODD phase control bit. As a result, all EVEN phase control bits are logic ’0’ because only ODD phase control bits differ in logic level.

The NRTZ RBU Architecture

The NRTZ RBU operates on NRTZ encoded data and is controlled by RTZ control signals as described in the previous section. Consequently, the NRTZ RBU architecture is almost the same as that produced for the RTZ RBU, as shown in figures 5.3, except that each NRTZ RBU register, shown in figure 5.4, is 24-bits long to accommodate the maximum length NRTZ encoded and tagged data word.

5.4.4 The Synchronous RBU

The synchronous SISA RBU [54] is shown in figure 5.5. It consists of two multiplexors, a demultiplexor, and four general purpose registers, one of which doubles as the communication register. An address is supplied to either one or both multiplexors, depending on the instruction, in order to select the source operand(s), and the demultiplexor is supplied with the destination address of the resultant operand.

Each selected source register, unless it also acts as the destination register, rotates its contents on each word-cycle in order to re-write its old data and to allow the data to be bit-serially read by the multiplexor(s), if required. The selected destination register discards its old data thus permitting the new data to overwrite it. Each register has a multiplexor at its input, which is controlled by a function within the instruction decoder, to direct the appropriate old (rewritten) or new (overwritten) data into the register.

The source data's least significant bit (LSB) is always read from the third latch pair, from the register input, two-bit cycles prior to the end of the previous word-cycle. Otherwise, if the LSB were taken directly from the register output then it would take 18-bit cycles, rather than the required 16-bit cycles, to pass the LSB from the register output through to the LSB of the destination register, because of the two full latches in the EXU's data path. As a consequence, the instruction source operand address fields must be pre-decoded 2-bits before the start of the next word-cycle; as will be discussed in chapter 6.

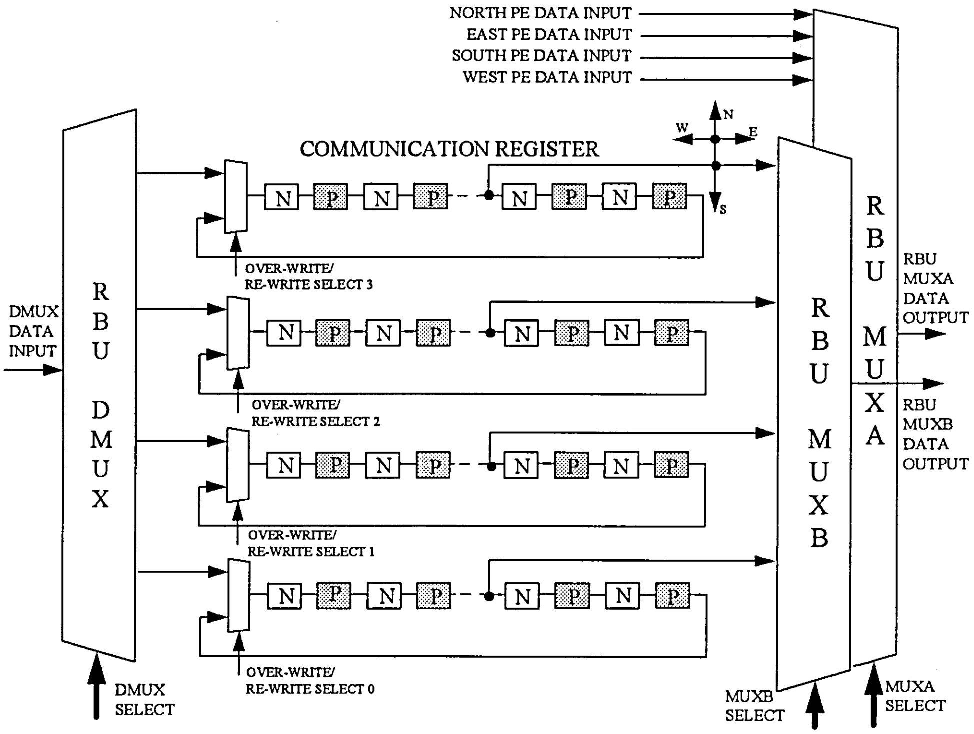


Figure 5.5. The synchronous register bank unit.

5.4.5 Results

Figure 5.6 illustrates the latencies of the RTZ and NRTZ RBU registers when operating on word lengths ranging from 1 to 16 bits, and the latency of the synchronous register when operating on its fixed length 16-bit word. **Figure 5.7** illustrates the latencies of the RTZ, NRTZ and Synchronous RBUs. **Figures 5.8** and **5.9** illustrate the throughput rates of the RTZ, NRTZ and synchronous registers and RBUs, respectively. **Table 5.1** shows gate count results for the RTZ, NRTZ and synchronous RBUs.

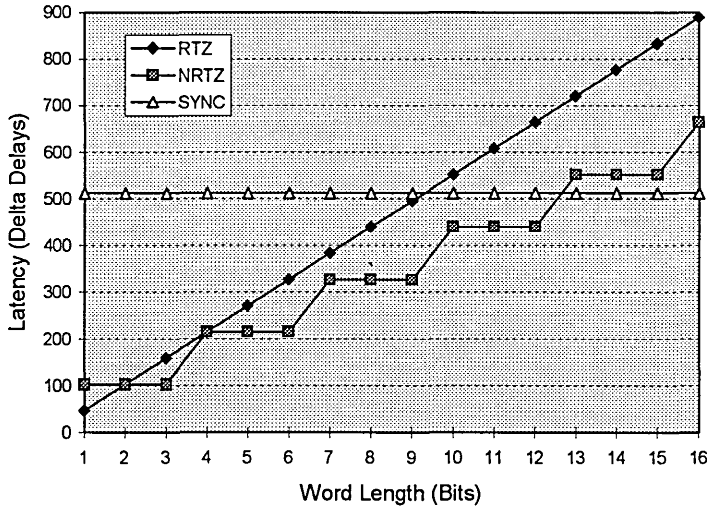


Figure 5.6. The RTZ, NRTZ and synchronous register latencies.

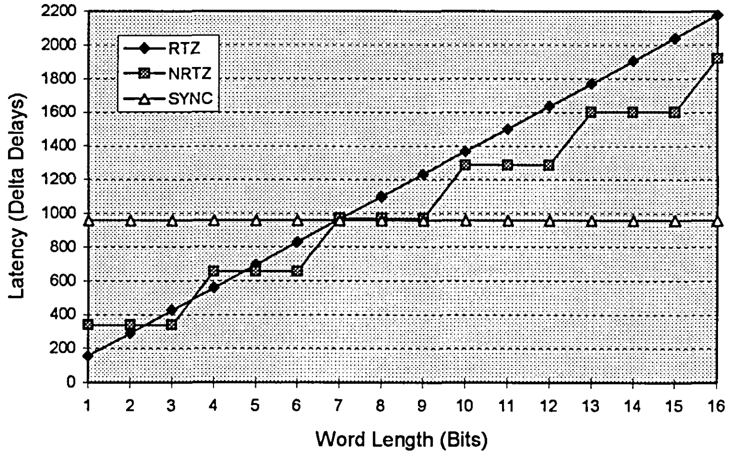


Figure 5.7. The RTZ, NRTZ and synchronous RBU latencies.

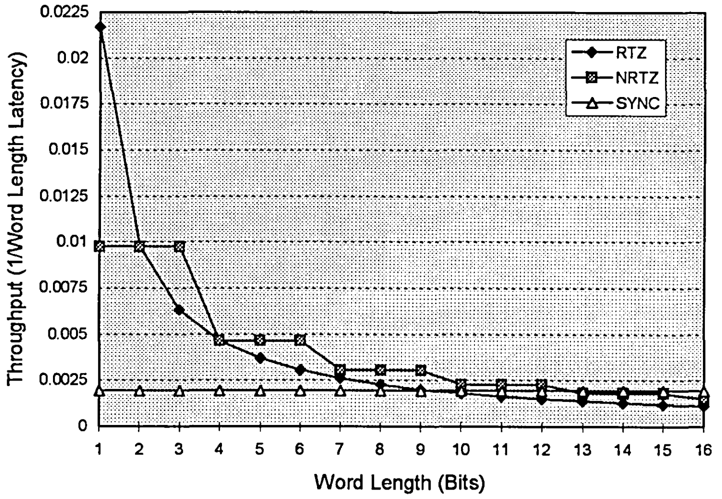


Figure 5.8. The RTZ, NRTZ and synchronous register throughputs.

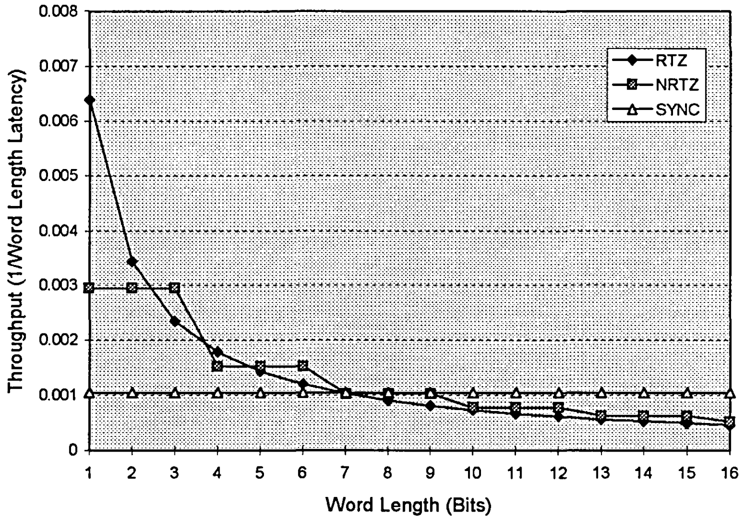


Figure 5.9. The RTZ, NRTZ and synchronous RBU throughputs.

	RTZ	NRTZ	SYNC
RBU	11121 gates	10209 gates	1175 gates

Table 5.1. The RTZ, NRTZ and synchronous RBU gate counts.

The following observations can be made from the above figures:

Figure 5.6 shows that the latency of the RTZ register is less than that of the synchronous register for words less than 9-bits in length. The NRTZ register is faster than the synchronous register when operating on word lengths of 12-bits and less, whereas the NRTZ is consistently faster than the RTZ except for a 1-bit word.

Figure 5.7 shows that the RTZ and NRTZ RBU latencies are faster than the Synchronous RBU for word lengths of less than 7-bits. The NRTZ RBU gradually improves its performance over that of the RTZ RBU as the word size increases.

Figure 5.8 shows that the throughputs of the RTZ and NRTZ registers are approximately four and nine times greater than the synchronous register, respectively, for a 1-bit word. It can also be seen that the throughputs of both the RTZ and NRTZ registers decreases gradually as the length of the word is increased, with the RTZ register's throughput less than the NRTZ and Synchronous register throughputs for long word lengths. The synchronous register's throughput remains constant for its fixed word size.

Figure 5.9 shows that the throughput rates of the RTZ and NRTZ RBUs are approximately six and three times greater, respectively, than that of the synchronous RBU for a 1-bit word. The throughputs of all three implementations are comparable over mid-range word lengths (7 to 9-bits), but the throughputs of the RTZ and NRTZ RBUs decrease below the throughput of the synchronous RBU as the word length is increased to the maximum size.

Table 5.1 shows that the RTZ RBU is approximately 9% bigger than the NRTZ RBU. However, both the RTZ and NRTZ RBUs are approximately nine times bigger than the synchronous RBU.

5.4.6 Analysis

The RTZ and NRTZ registers and RBU implementations differ to their synchronous counterparts in that the length of the word they operate upon can vary from word to word. This has a significant effect on the latency and throughput of these functions when they operate on short length words. The latencies of RTZ and NRTZ RBUs, as compared to the latencies of the RTZ and NRTZ registers, respectively, increases significantly due to the inclusion of the large RBU multiplexors and demultiplexor in the data path.

The circuit area required to implement both the RTZ and NRTZ RBUs is considerably larger than the area required by the synchronous RBU. This is due to the extra circuitry required by the self-timed circuit implementation technique to accommodate dual-rail encoded words. In addition, the large amount of local register control circuitry and the large amount of circuitry required to implement the RBU 8-input multiplexor add to the circuit overhead.

Any clock skew that might detrimentally effect the processing speeds of the synchronous register and RBU would have to increase their latency by approximately 76% and 124%, respectively, to make them as slow as their RTZ counterparts. This assumes that all operate on the same maximum length 16-bit word.

5.5 The Data Output Unit (DOU)

The DOU is required to send data from the transmitting PE to one or more receiving PEs.

5.5.1 The Self-timing Requirements

The self-timed DOU is required to control and synchronise the asynchronous transfer of data from transmitting PE to one or more receiving PEs. This is because problems arise when multiple asynchronous PEs attempt to read the data from the transmitting PE at different instances in time [31, 32]. Therefore, a data bit output from the DOU must be read by all the appropriate receiving PEs before the next bit is output. This is achieved by synchronising the acknowledge signals, received from the reading PEs, on each and every bit transfer.

Each PE in the ST-SISA array can communicate with one, some or all of its four neighbouring PEs. Consequently, data might be sent to only one or two neighbouring PEs, therefore only the PEs that read the data need to be synchronised, the remainder can be ignored. Therefore, the DOU must know which neighbouring PEs are to read the data item in order to synchronise only the PEs involved in the data transfer. This is achieved by monitoring the appropriate bits in the result field, on the execution of a write instruction, where the logic '1' bit(s) identify the receiving PE(s), as previously described in section 4.3.4.

The rendezvous function performed by the C-element is ideally suited to the synchronisation of multiple acknowledge signals. However, problems arise when only one, two or three of the neighbouring PEs read a data item because only signal changes will only take place on some, not all, of the C-element's input lines. Consequently, the C-element's output cannot signal the completion of the synchronisation because it must receive signal changes on all of its inputs. Therefore, a mechanism is required to force a signal change on each acknowledge line from each PE which does not read the data item. This enforcement of a signal change on an acknowledge line is termed a 'pseudo write'.

Data output to the receiving PE(s) must be buffered in order to absorb processing time differences between the various PE(s) involved in the communication. For example, if a write instruction is executed by PE1, before the read instruction of PE2, PE1 must wait for PE2 to execute the read instruction, thus slowing PE1. By inserting a buffer in each output direction of the DOU these variations in PE processing time can be absorbed.

The DOU only monitors the operand result field when a write instruction is executed, or until the EOW has been output from the DOU, otherwise data will be unnecessarily output resulting in incorrect array operation.

5.5.2 The RTZ Experiment

The RTZ Requirements

The RTZ requirements for the DOU are the same as those stated for the RTZ RBU in section 5.4.2.

The RTZ DOU Architecture

The RTZ DOU architecture is shown in figure 5.10.

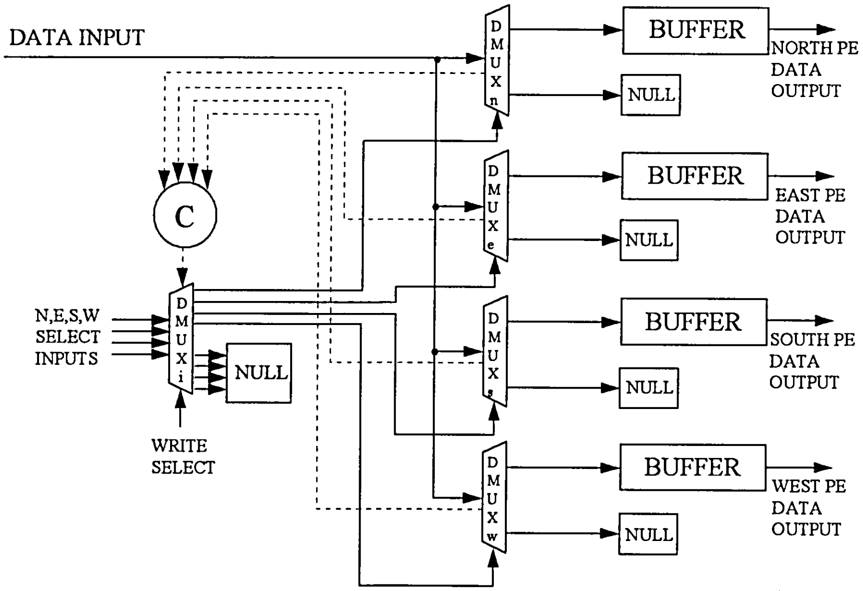


Figure 5.10. The RTZ data output unit.

DMUXi: This role of this device is to pass the (N)orth, (E)ast ,(S)outh ,(W)est write select bits to the corresponding DMUXn, DMUXe, DMUXs and DMUXw data output devices, when the write control input is set to logic '1', or to the NULL circuit, when the select input is set logic '0'. This device also re-reads and outputs the N,E,S,W write select bits, when it receives an acknowledge signal from the C-element, on each and every data bit output from the DOU until the write control input is set logic '0'.

DMUX(n, e, s, w): The purpose of each of these devices is to pass the data bit to the corresponding receiving PE, if the control input is set to logic '1', or to pass the data bit to the null circuit, if the control input is set to logic '0', where the data bit is discarded thus performing a pseudo write. The pseudo write enables an acknowledge signal to be sent to the C-element.

C: When all four acknowledge signals have been received from DMUXn, DMUXe, DMUXs and DMUXw, the C-element signals the end of the bit transfer to DMUXi.

Buffer: Stores the bit-serial data word on route to its receiving PE. This enables the DOU to continue to output data even if the receiving PE is not ready to read the data, however, when the register is full the DOU must wait.

Null: This device simply discards its input.

5.5.3 The NRTZ Experiment

The NRTZ Requirements

The NRTZ requirements for the DOU are the same as those stated for the RTZ DOU in section 5.5.2 because both must operate on two-phase data and RTZ control information.

The NRTZ DOU Architecture

The NRTZ DOU architecture is almost the same as that of the RTZ DOU, figure 5.10, except each output buffer register is 24-bits long to accommodate the maximum length NRTZ tagged data word.

5.5.4 The Synchronous DOU

Each SISA PE has its own dedicated DOU [54], or communication register (CR), as shown in figure 5.11, that can be read directly by one, some or all of its neighbouring PEs. The CR is realised using a simple general purpose register except it has four extra output connections, one to each of the four neighbouring PEs. Therefore, the CR operates in exactly the same way as the general purpose register described in section 5.4.2, except that it can be read by up to four extra PE receivers.

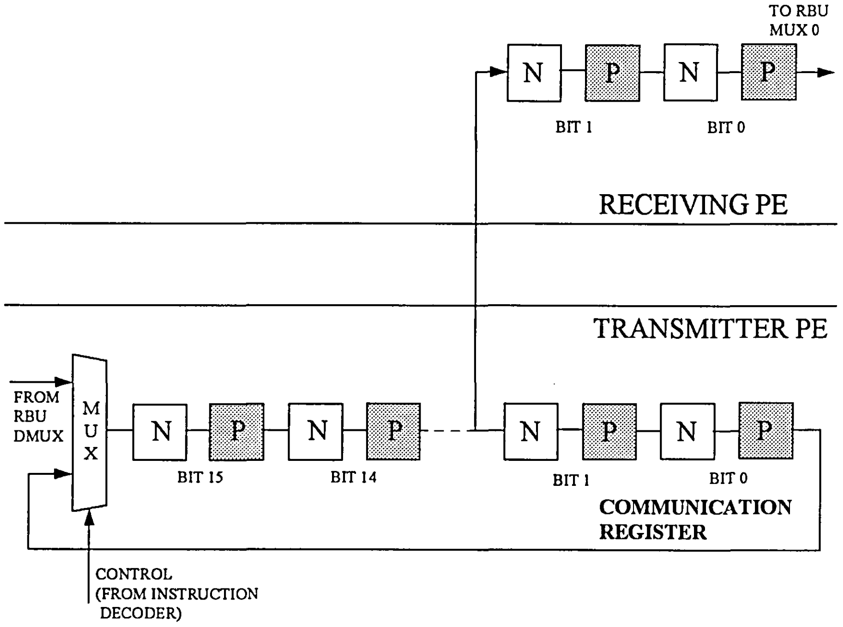


Figure 5.11. The synchronous data output unit.

5.5.5 Results

Figures 5.12 and **figure 5.13** show the latency and throughput results, respectively, for the RTZ and NRTZ DOUs when outputting variable length bit-serial words ranging from 1 to 16 bits in length. **Table 5.2** shows the gate count for each of the new RTZ and NRTZ DOUs as well as for the existing synchronous DOU.

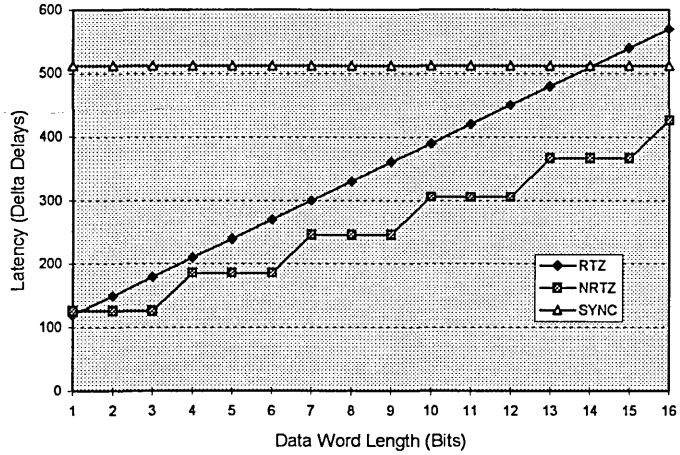


Figure 5.12. Latency Results for the RTZ, NRTZ and synchronous DOUs.

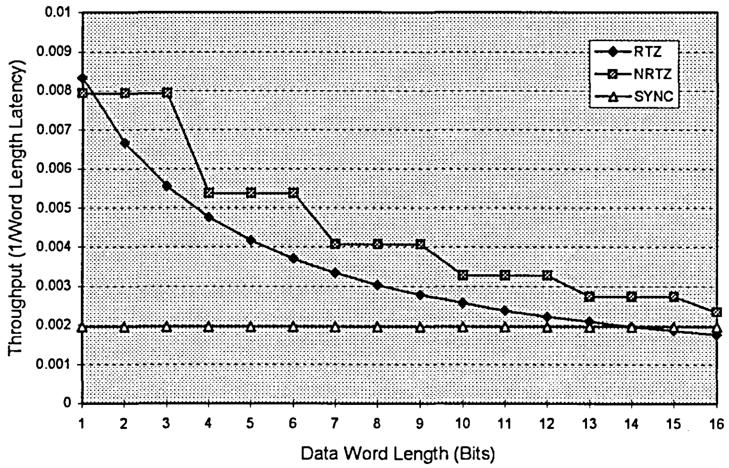


Figure 5.13. Throughput results for the RTZ, NRTZ and synchronous DOUs.

	RTZ	NRTZ	SYNC
DOU	2950 gates	2342 gates	292 gates

Table 5.2. The RTZ, NRTZ and synchronous DOU gate counts.

Figure 5.12 shows that the latency of the NRTZ DOU is less than that of the RTZ DOU for all word lengths, assuming both operate on the same length word. They are both considerably faster than the synchronous DOU for short and mid-range length words (1 to 14-bits words), with the NRTZ DOU faster for long words (15 and 16-bit words).

Figure 5.13 shows that the throughput of the NRTZ DOU is considerably greater than that of the synchronous DOU for all word lengths, whereas the throughput of the RTZ DOU is greater than the synchronous DOU for all word lengths of up to 14-bits.

Table 5.2 shows that both the RTZ and NRTZ DOUs are considerably larger than their synchronous counterpart, whereas the RTZ DOU is 26% larger than the NRTZ DOU.

5.5.6 Analysis

The RTZ and NRTZ DOUs have variable processing rates due to the variable length of the data words. The RTZ and NRTZ DOUs are significantly faster than their synchronous counterpart because of their simple design. The RTZ and NRTZ DOUs are both considerably larger than the synchronous DOU because of the extra circuitry associated with self-timed circuit implementation and because of the large circuit area associated with the inter-PE data buffers.

Any clock skew that might detrimentally effect the performance of the synchronous DOU would have to increase its latency by approximately 15% to make it as slow as the RTZ DOU when operating on the same maximum length 16-bit word.

5.6 The Execute Unit (EXU

The EXU [33], illustrated in figure 5.14, performs several functions: ASSIGN, AND, OR, ADD, MIN and MAX. The desired function is determined by the op-code, which controls the path taken by the source operand(s) through the EXU's input demultiplexor(s) and the path taken by the result through the output multiplexor to the output of the EXU.

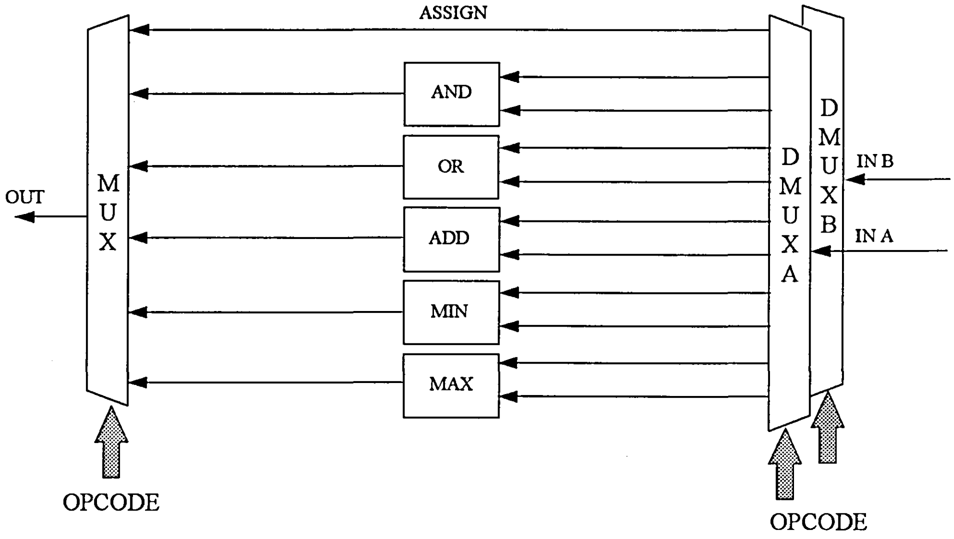


Figure 5.14. The execute unit.

The functions of the EXU can be divided into four main categories, namely assignment, logical, addition and comparison operations. It is not necessary to perform an experiment for the ASSIGN instruction because no special circuitry needs to be designed. All that is required is a single dual-rail line to connect the ASSIGN output from DMUX A to the corresponding input of MUX, as illustrated in figure 5.14. However, the logical, addition and comparison categories require more in-depth analysis and are considered in the following three main sections.

5.6.1 The Logical Function Experiment

The role of the logic processing function is to perform logical operations such as AND on two bit-serial operands. This is achieved by ANDing the *n*th bit (where *n* is the bit position) of the first operand (operand A) to the corresponding *n*th bit of the second operand (operand B), for all *n* up to the end of the longest operand. For example, bit 0 of operand A must be ANDed with bit 0 of operand B then bit 1 of operand A with bit 1 of operand B, and so on until the end of the longest operand.

5.6.1.1 The Self-timing Requirements

Due to the variable length of operand words there is a strong possibility that the AND circuit will operate on operands of differing lengths thus presenting problems of circuit deadlock, as discussed in section 5.3. In order to overcome this problem extra logic bits, called pseudo bits, must be appended to the shorter operand to make both operands the same length, enabling the data-driven logic function to continue processing the longer operand.

It is possible to deduce from these requirements that the length of the resultant operand does not increase in length over that of the longer input operand. Also, for every input bit an output result bit is produced. The next section details the realisation of the RTZ logic function.

5.6.1.2 The RTZ Experiment

The RTZ Requirements

The AND function can only operate on RTZ encoded words (i.e. data-true, data-false and empty), the illegal EOW Tag bit (i.e. F0='1', F1='1') cannot be processed. Therefore, an EOW Tag must be converted to an empty (E) value before it enters the AND function and then converted back to an EOW Tag only when both input data bits are EOW Tags, thus indicating the end of the operation.

The pseudo data valid bits for the RTZ circuit are assumed to have a logic '0' value in order that the magnitude of the shorter operand is left unchanged, however each valid (V) bit must be separated by an EOW (T) Tag bit. This enables the pseudo EOW Tag bit of the shorter operand to coincide with the EOW Tag of the longer operand.

All data path components, unless stated otherwise, are realised using NRTZ circuitry to cater for the EOW Tag, whereas all control components are realised using RTZ circuitry. The reasons for this have been explained previously in section 5.4.2.

The RTZ AND Architecture

The RTZ AND architecture is shown in figure 5.15.

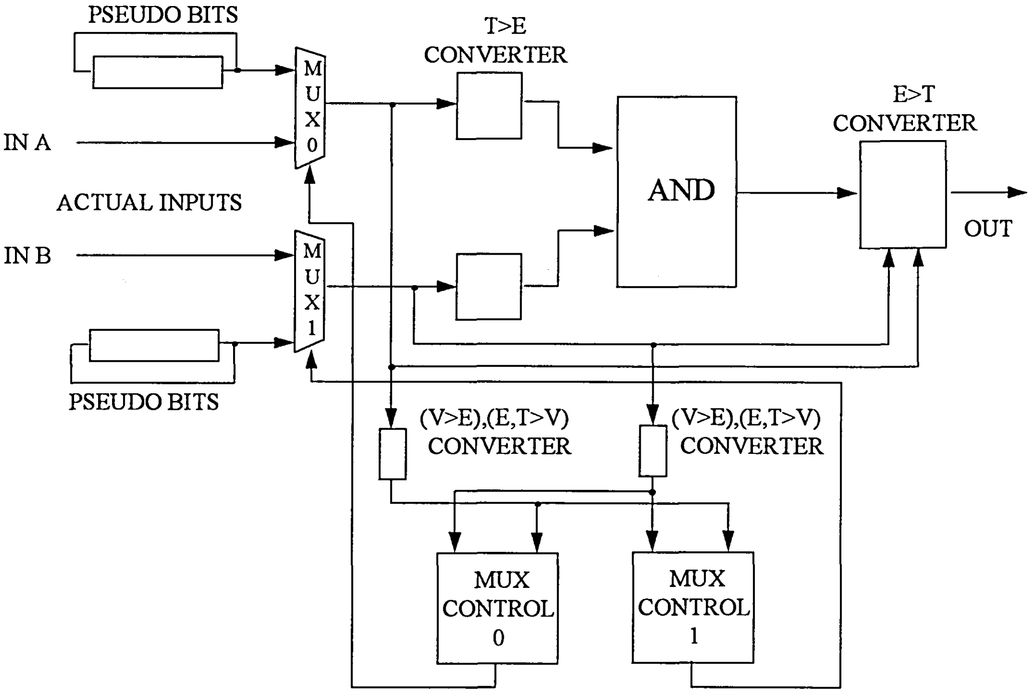


Figure 5.15. The RTZ AND.

MUX0/MUX1: The role of MUX0/MUX1 is to pass either an actual data bit to the AND function when its control input is logic '0', indicating that the EOW has not been detected, or a pseudo data bit, if its control input is set to logic '1', indicating that the EOW has been found. The pseudo data bits, fed by a FIFO ring buffer, enable dummy bits to be appended to the end of short words thus enabling the AND output to change state.

(V>E), (E, T>V) CONVERTER: The purpose of this circuit is to convert each monitored data bit into the equivalent RTZ code, of the correct phase, required by the two monitoring MUX CONTROL circuits. This enables local control functions to be realised in RTZ circuitry thus reducing their complexity. In addition, it converts the phase of the data so that the output from MUX CONTROL 0/MUX CONTROL 1 is in the correct phase for the control input of MUX0/MUX1. It behaves as follows. The device converts any valid data bit (logic '0' or logic '1') to an empty bit, an empty to a logic '0' data bit (data-false), and an EOW Tag bit to a logic '1' data bit (data-true).

MUX CONTROL(0,1): The role of this circuit is to control the passing of the actual or pseudo data bits through the controlled MUX circuit. This device monitors the output bits from MUX0 and MUX1, for each bit cycle, which are termed the primary multiplexor, the controlled device relative to MUX CONTROL 0/MUX CONTROL 1, and the secondary multiplexor, which is the controlled device of the other MUX CONTROL circuit. Each MUX CONTROL circuit behaves as follows. When the secondary input (SI) is a tag and the primary input (PI) is an empty (i.e. the secondary word is shorter than the primary word), or both the SI and PI are tag bits, MUX CONTROL selects the actual data input on the next bit cycle. When the PI is a tag and the SI is empty bit (i.e. the primary word is shorter than the secondary word) the multiplexor's pseudo input is selected in order to attach a dummy bit to the end of the word. The pseudo input will continue to be selected on each bit cycle until the SI is an EOW Tag.

T>E CONVERTER: Whenever an EOW Tag bit is input this circuit converts it to an empty value. All other states are passed unchanged.

E>T CONVERTER: Converts an empty state, output from the AND function, into an EOW Tag bit whenever the two input bits to the AND are EOW Tags.

AND: This is a conventional RTZ logical AND function.

5.6.1.3 The NRTZ Experiment

The NRTZ Requirements

Like the RTZ implementation, the NRTZ AND function must not operate on logic '1' EOW Tag bits (EOW1), which indicate the end of an word, otherwise the logic value of an EOW Tag bit might be changed. Therefore, an EOW1 must be converted to a logic '0' EOW Tag (EOW0), an unused EOW Tag bit, before it enters the AND and then converted back to an EOW1 only when both inputs are EOW1.

Pseudo bits must be appended to the end of the shorter word to make both words equal in length. For the NRTZ ST-SISA presented in this work, *n* multiples of four pseudo bits (where *n* is the number of extra tagging periods) are appended to the end of the shorter word. The first three bits are set to logic '0' so as not to change the value of the shorter word, the fourth bit, or the pseudo EOW Tag bit, must be set to a logic '1' to indicate the EOW.

All data path components, unless stated otherwise, are realised using NRTZ circuitry, whereas all control components are realised using RTZ circuitry, for reasons outlined in section 5.4.3.

The NRTZ AND Architecture

The NRTZ AND architecture, shown in figure 5.16, is similar to that of the RTZ, except that it operates using the NRTZ data encoding and tagging scheme. The MUX0 and MUX1 data input devices are the same as used for the RTZ version, therefore only the other functions are considered.

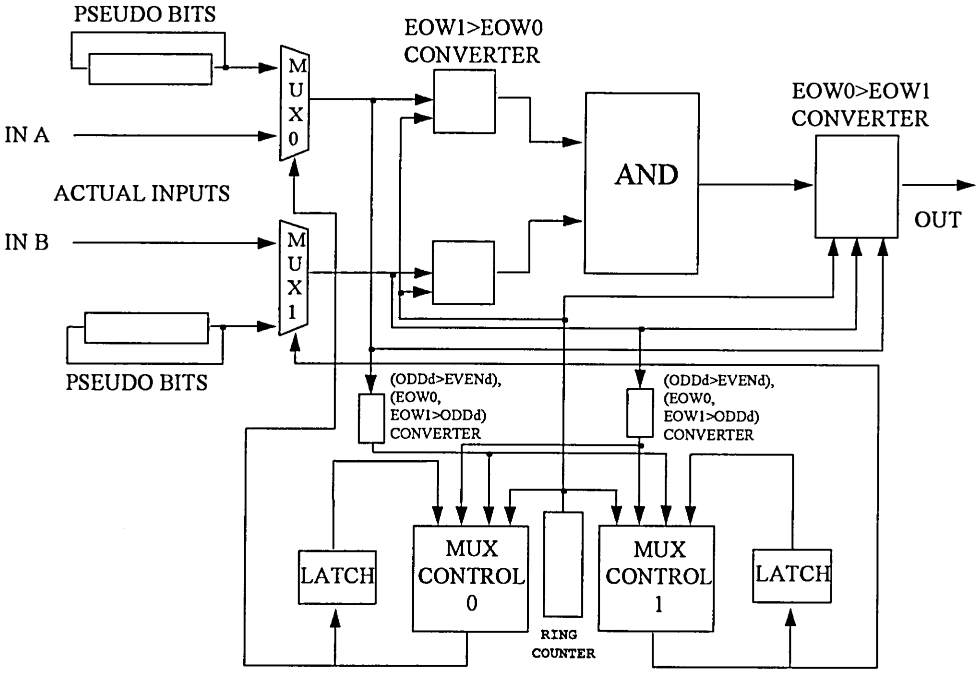


Figure 5.16. The NRTZ AND.

Ring Counter: A counting mechanism, realised by a ring buffer, presented in section 4.3.1.2.2, is used by other components to differentiate between the EOW Tag and data bits in the bit-serial data stream.

MUX CONTROL 0/MUX CONTROL 1: The role of this device is the same as that used for the RTZ AND architecture. Consequently, it operates in a similar manner to the previously explained MUX CONTROL, except it differentiates between an EOW1 or an EOW0 and a EVEN data bit (EVENd) via monitoring the ring counter output. Feedback from the previous output state is used to indicate whether an actual data input or a pseudo data input was selected in the last bit cycle thus maintaining the same MUX CONTROL output status between EOW Tag bits. The operation of this circuit is explained as follows.

In the event of an EOW1 on the ring counter input (RCI), and a data bit (EVENd) set to logic '0' on the feedback input (FI), MUX CONTROL selects the actual input of the multiplexor if PI is an EOW0 and SI is either an EOW1 or EOW0, or if both PI and SI are EOW1. If PI is an EOW1, independent of the logic value of the FI, and SI is a EOW0, then the pseudo input is selected in order to append dummy bits on to the end of the word.

In the event of an ODD data bit (ODDd) or EVENd on the RCI, MUX CONTROL selects the actual input of the multiplexor if FI is logic '0', or the pseudo input if FI is logic '1'. Therefore, the intermediary bits, between EOW Tag bits, of either the pseudo word or actual word are selected.

EOW1>EOW0 CONVERTER: This devices converts an EOW1 to an EOW0, when the output from the ring counter is an EOW1, before it is input into the AND to ensure the AND does not change an EOW0 to an EOW1.

EOW0>EOW1 CONVERTER: An EOW0 output from AND is converted to back to an EOW1 only when both data inputs into the two EOW1>EOW0 CONVERTERS are EOW1 and the RCI is an EOW1.

(ODDd>EVENd), (EOW0, EOW1 > ODDd) CONVERTER: This device allows MUX CONTROL (0 & 1) to be realised using RTZ circuitry. It achieves this by converting an ODDd of any logic value to an logic '0' EVENd and converting an EOW0 or EOW1 to the corresponding logic level ODDd.

5.6.1.4 Synchronous AND

The synchronous AND, figure 5.17, is realised by a simple AND gate latched at its inputs and output.

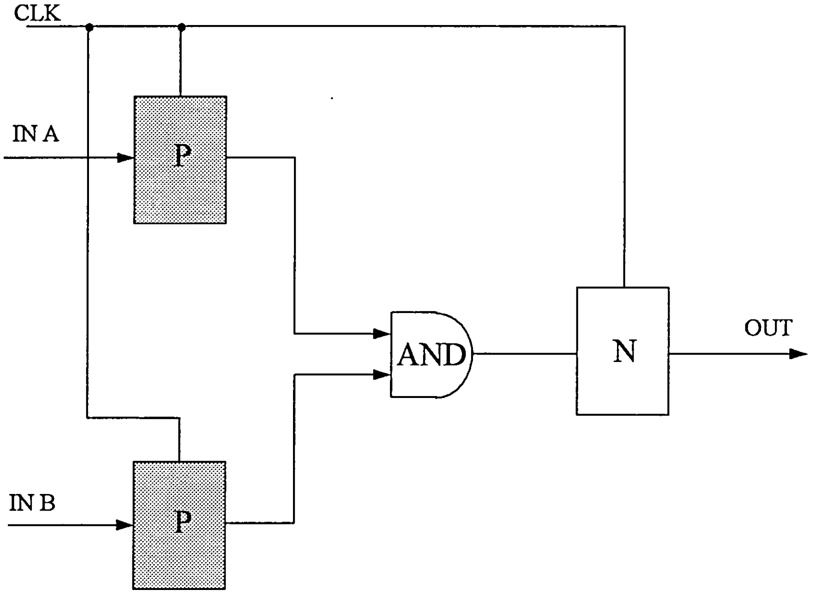


Figure 5.17. The synchronous AND.

5.6.1.5 Results

Figures 5.18, Figure 5.19 and **Table 5.3** illustrate the latency, throughput and gate count results, respectively, for the RTZ and NRTZ AND functions, that operate on variable length words, and the fixed word size synchronous AND function.

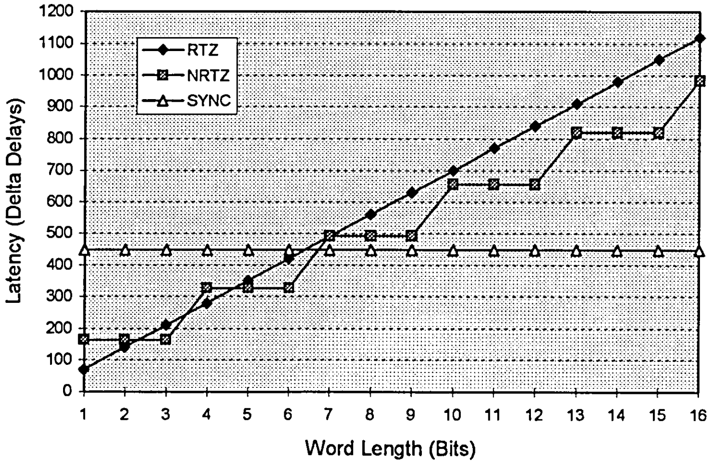


Figure 5.18. The RTZ, NRTZ and synchronous AND function latencies.

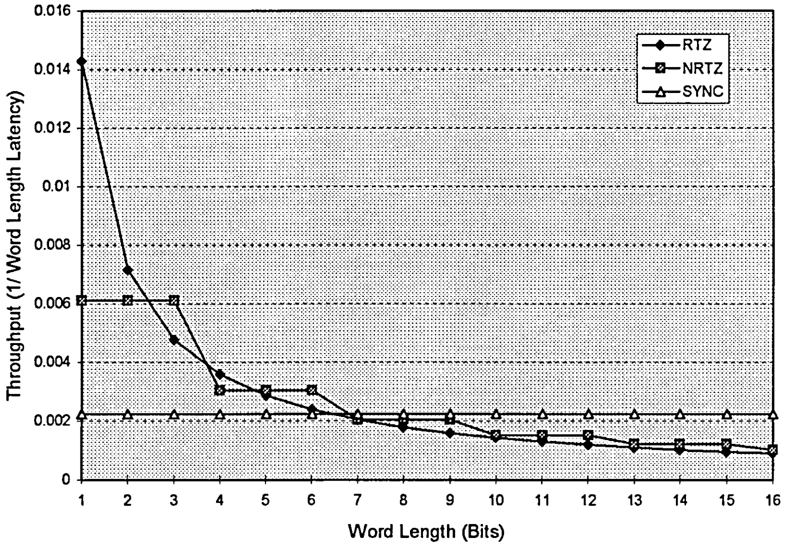


Figure 5.19. The RTZ, NRTZ and synchronous logic function throughputs.

	RTZ	NRTZ	SYNC
LOGIC	746 gates	1599 gates	27 gates

Table 5.3. The RTZ, NRTZ and synchronous logic function gate counts.

The following observations have been made:

Figure 5.18 indicates that the NRTZ AND operation is slightly faster than the RTZ equivalent for all word lengths except for 1,2,4 and 7 bits long. In addition, the graph shows that significant performance improvements are available for small magnitude data using variable word methods when compared with the same fixed word-length synchronous method. However, the RTZ and NRTZ implementations cease to perform better than the equivalent synchronous AND for word lengths greater than 6-bits.

Figure 5.19 shows that the throughput of the RTZ AND is approximately two times greater than the NRTZ AND function for a 1-bit word, however throughputs are comparable as the word size is increased. Both the RTZ and NRTZ AND throughputs are significantly greater than that of the synchronous AND for short words (1-5 bits), comparable to the synchronous AND over the mid range (6-9 bits) and approximately 50% less efficient than the synchronous AND for long words (10 -16 bits).

Table 5.3 shows that the NRTZ AND is just over twice as large as the RTZ equivalent. However, the RTZ and NRTZ AND circuits are approximately twenty-eight and sixty times larger, respectively, than the equivalent synchronous AND.

5.6.1.6 Analysis

The NRTZ latency is greater than that of the RTZ AND for 1,2,4 and 7-bit word lengths because it requires more bits to encode the data. However, all other word lengths require less bits to encode and tag the data resulting in the NRTZ AND being consistently faster than the RTZ AND. Both the RTZ and NRTZ AND functions are faster than the fixed speed synchronous AND function for short word lengths because they require fewer bit cycles to process data.

The greater circuit area of the NRTZ AND compared with that of the RTZ AND can be attributed to the increased complexity of the multiplexor controllers, including the ring buffer counter, and as an effect of the extra expense of the multiplexor control circuits as the number of inputs is incrementally increased.

Any clock skew that might detrimentally effect the synchronous AND's processing speed would have to increase its latency by approximately 146% to make it as slow as the RTZ AND when it operates on the same maximum length 16-bit word.

5.6.2 The ADD Function Experiment

The ADD function adds together two bit-serial words such that the *n*th bit of operand A is only added to the same *n*th bit of operand B, and the carry-propagation bit from the previous bit addition, up to the end of the longest word. Each bit addition produces a sum bit and a carry bit where the carry bit is fed back as input to the next bit addition.

5.6.2.1 The Self-timing Requirements

The major difference between a variable length bit-serial addition function and the previous logical function is that the word size in the former may increase as a result of carry propagation or overflow. This can be catered for, in the event of overflow, by simply appending pseudo bits to the end of both source operands in order to propagate the carry to the sum output, thus increasing the length of the word. If the bit-serial word increases in length from the maximum length of 16 bits to 17-bits then the ST-SISA PE will continue operating correctly because the RTZ RBU and NRTZ RBU registers can accept a maximum word length of 20-bits and 24-bits, respectively; anything beyond these ranges for the respective implementations would cause the entire PE to deadlock.

5.6.2.2 The RTZ Experiment

The RTZ Requirements

The requirements of the RTZ ADD function are the same as those stated for the logic function. However, there is an extra requirement as a consequence of the possibility of overflow in the ADD function. This means that the empty bit output from the ADD cannot be converted to an EOW Tag bit when both the inputs are tag bits because the overflow bit must be appended to the end of the sum word before the EOW Tag. Therefore, only when there is no overflow can the empty value be converted back to an EOW Tag bit.

All data path components, unless stated otherwise, are realised using NRTZ circuitry, and all control circuitry is realised using RTZ circuitry, for reasons outlined earlier in section 5.4.2.

The RTZ ADD Architecture

The RTZ ADD, figure 5.20, is similar to the RTZ AND except for changes to the two MUX CONTROL circuits, output tag converter and the substitution of the AND with the RTZ full adder.

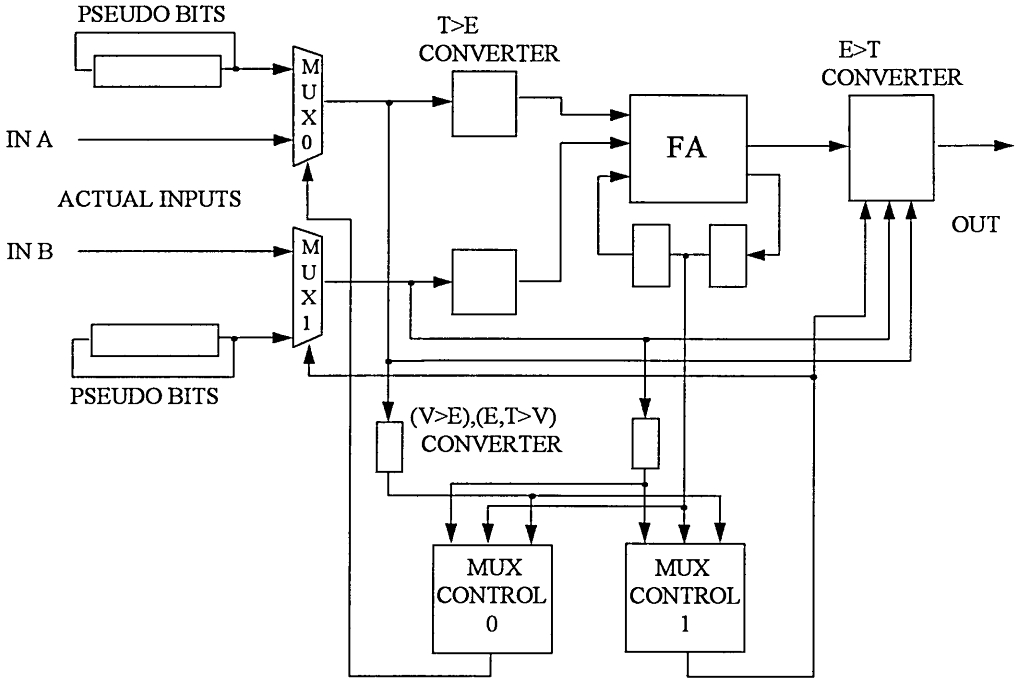


Figure 5.20. The RTZ ADD.

MUX CONTROL 0/MUX CONTROL 1: The role and operation of this circuit is the same as that used in the AND function, except that it monitors the carry-out bit produced by the full adder. Consequently, if the carry-out bit is logic '1' and the present input operand bits are EOW Tag bits then the pseudo input is selected in the next bit cycle thus appending a logic '0' and a tag bit, in consecutive bit cycles, to the end of the input operand.

E>T CONVERTER: The role of this function is to ensure that the empty bit output from the FA is converted to an EOW Tag bit only when both inputs are EOW Tags and both MUX CONTROL outputs are selecting the next actual data input; when the MUX CONTROL output is logic '0' this indicates that either there is no overflow or the overflow has been previously handled.

Full Adder (FA): This is a conventional RTZ full adder function [70].

5.6.2.3 The NRTZ Experiment

The NRTZ Requirements

The NRTZ requirements are similar to those stated for the NRTZ logic function, except that carry feedback in the NRTZ full adder introduces two problems, which are elaborated below.

The first problem is associated with the propagation of the carry output bit to the carry input of the next bit addition. For example, if an EVENd phase carry output bit is fed back to the carry input for addition with the next operand data bits, of ODD phase, then the FA will deadlock because the three inputs into the FA function are not in the same phase. To overcome this problem an EVENd phase carry output bit must be converted to an ODDd when passed back to the carry input, and vice versa for an ODDd carry output bit to an EVENd carry input bit.

The second problem is concerned with the propagation of an ODDd carry bit, from the addition preceding the input of EOW0 tag bits, to the addition of the ODDd data bits succeeding the EOW0 tag bits. This is illustrated below in figure 5.21.

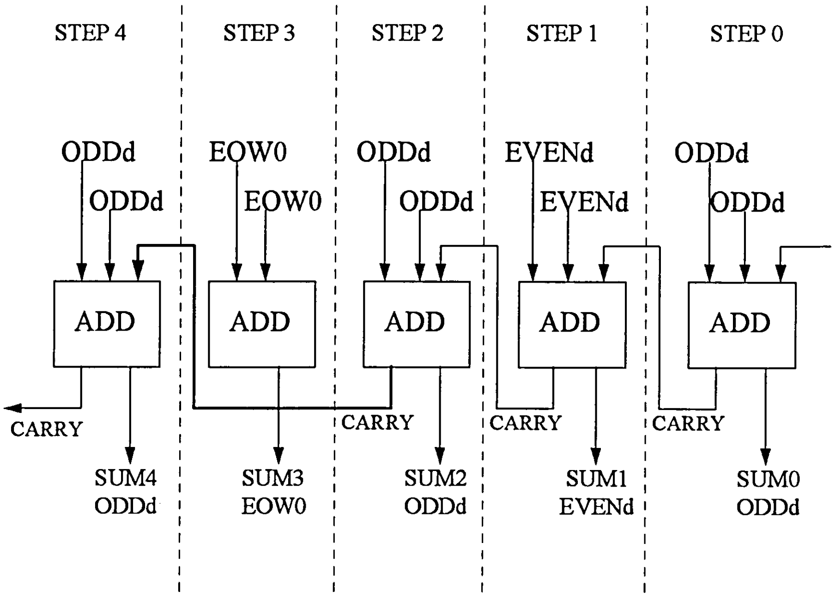


Figure 5.21. The carry propagation problem.

The figure shows that the carry output in step 2 must propagate to the addition in step 4, otherwise the addition of a logic '1' carry bit to the two EOW0 operand bits in step 3 would result in an undesirable EOW1 sum output, signalling the EOW prematurely. In other words, the two EOW0 tag bit inputs must be output from the ADD as an EOW0 sum, therefore not changing the logic value of the EOW0 in the output stream, in conjunction with passing the ODDd carry of the previous addition through the FA in readiness for the succeeding ODD phase data bit addition. This is achieved by adding one of the EOW0 bits to two copies of the EVENd (phase changed) carry bit, such that the carry bit is propagated to the next addition. Therefore, if the carry input is logic '1' the sum is logic '0' and the carry produced is logic '1', otherwise, if the carry input is logic '0' both the sum and carry outputs are logic '0'.

All data path components are realised using NRTZ circuitry, whereas all control components are realised using RTZ circuitry, for reasons explained earlier in section 5.4.3.

The NRTZ ADD Architecture

The NRTZ ADD architecture is shown in figure 5.22.

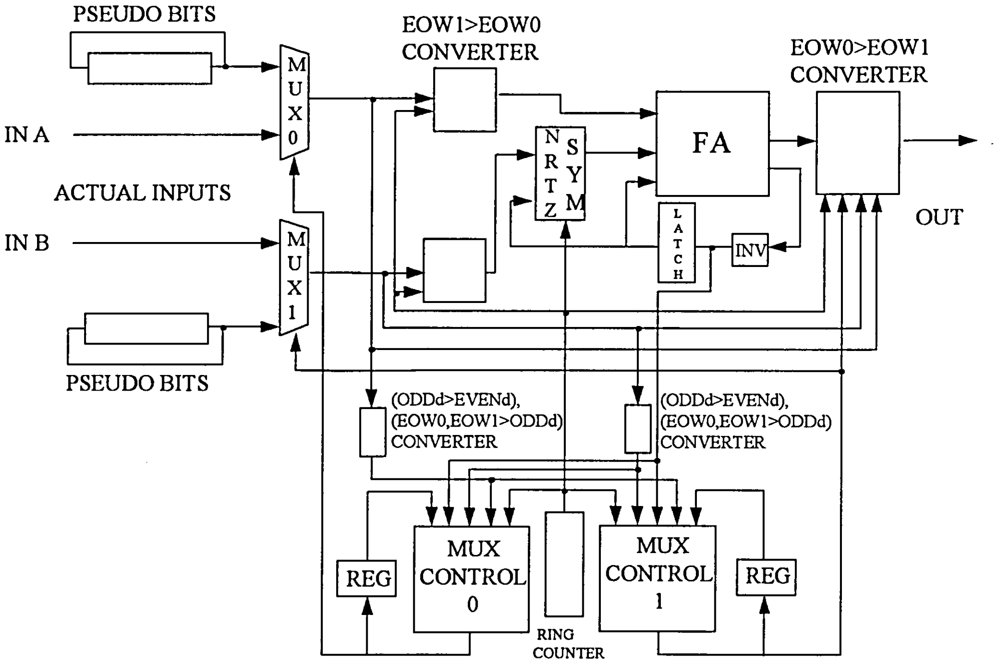


Figure 5.22. The NRTZ ADD.

Phase Invertor (INV): The role of this device is to invert the phase of the output carry bit before it is input into the next addition. This is realised by placing an invertor gate in the phase wire of the carry output from the full adder. This ensures that the phase of the carry toggles without it affecting its logic level.

NRTZ Symmetric (SYMM) Switch: The role of this device is to propagate the carry bit produced by the addition before the EOW0 tag bits to the addition following the EOW0 tag bits. The functionality of the NRTZ symmetric switch, as depicted in figure 5.23, is as follows. The switch accepts operand and carry bit inputs and a control input. Whenever the control input is logic '0' the operand and carry bits are passed to their respective outputs thus allowing the FA to add together the two operand data bits and carry bit. However, when the control input is logic '1' the inputs are passed to each other's respective output. In other words they are crossed over, resulting in one of the EOW0 bits being discarded, by the null circuit, and another copy of the carry bit being sent to the FA. Therefore, if a carry bit set to logic '1' is added to itself and the EOW0 then the sum will be logic '0' and the carry output logic '1'. Otherwise, if a carry bit set to logic '0' is added to itself and the EOW0 then both the sum and carry output will be logic '0'. This operation allows the carry bit to propagate unchanged through the EOW0 addition phase, to the next addition, without changing the value of EOW0 on output.

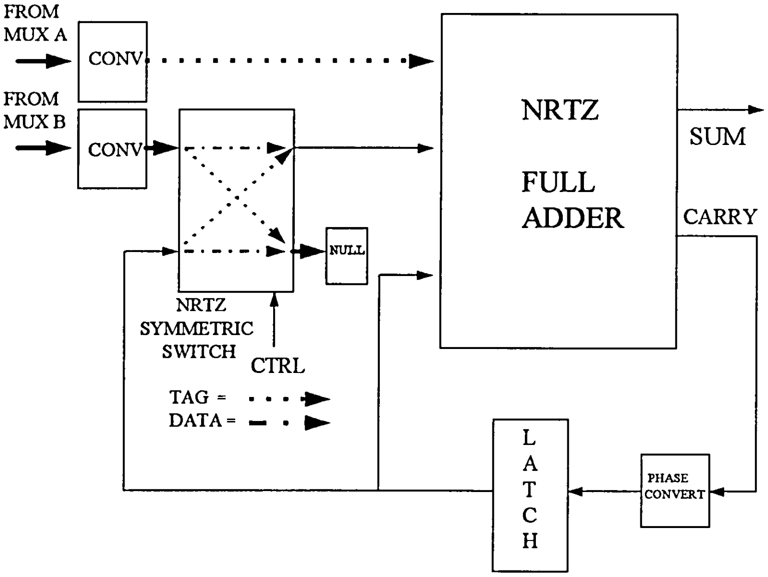


Figure 5.23. The NRTZ full adder.

MUX CONTROL (0 & 1): The role and operation of the MUX CONTROL is similar to that used for the AND circuit, except it monitors the carry-out bit produced by the full adder. Consequently, if the previous addition produces a carry and the present input operands bits are both EOW1 then the pseudo input is selected in the next bit cycle.

EOW0>EOW1 CONVERTER: The purpose of this device is to convert an EOW0 output from the FA to an EOW1 only when both inputs are EOW1, the ring counter is an EOW1 and both multiplexor controller outputs are selecting the next actual data input. This indicates that either there is no overflow or the overflow bit has been previously output.

5.6.2.4 The Synchronous ADD

The synchronous ADD function [91], figure 5.24, is realised by a single FA which is latched at its inputs and outputs. It operates as follows. The two source operands are fed bit serially into the FA where they are added together with the carry bit of the previous bit addition, consequently producing a sum bit, which is fed to the destination register, and a carry bit, which is fed into the next addition.

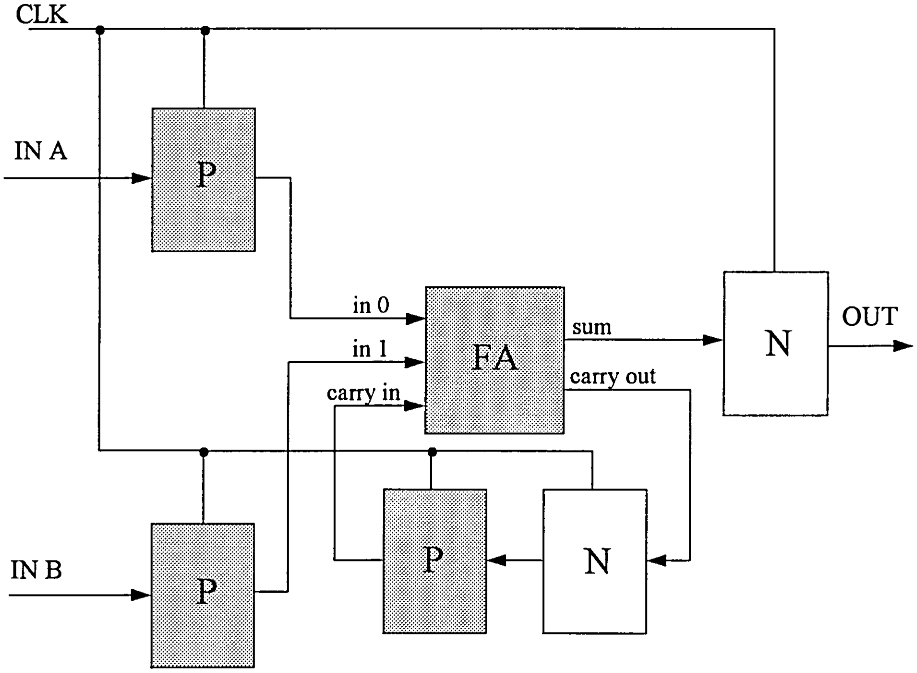


Figure 5.24. The synchronous ADD.

5.6.2.5 Results

Figures 5.25, **Figure 5.26** and **Table 5.4** illustrate the latency, throughput and hardware results, respectively, for the RTZ, NRTZ and synchronous ADD function.

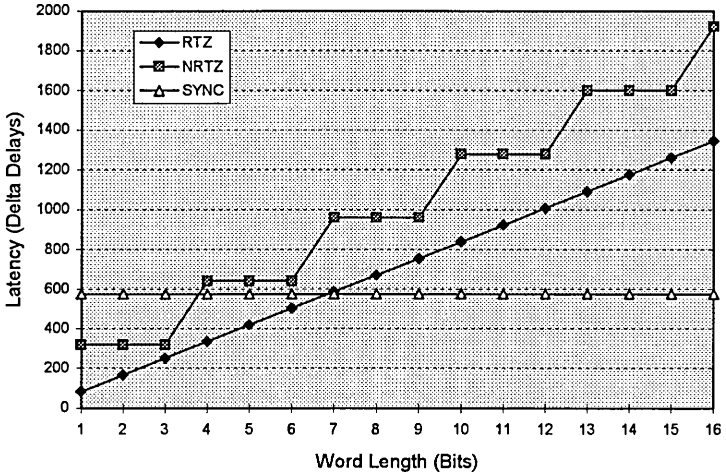


Figure 5.25. The RTZ, NRTZ and synchronous ADD latencies.

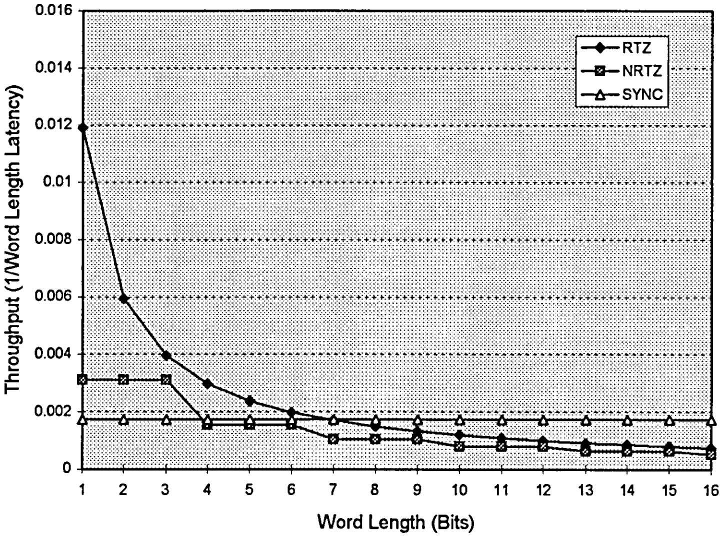


Figure 5.26. The RTZ, NRTZ and synchronous ADD throughputs.

	RTZ	NRTZ	SYNC
ADD	1309 gates	2142 gates	62 gates

Table 5.4. Gate counts for the RTZ, NRTZ and synchronous ADD functions.

The following observations can be made from these results:

Figure 5.25 shows that the RTZ ADD is faster than the synchronous ADD for word lengths less than 7-bits, as compared to the NRTZ adder which is faster when operating on words lengths less than 4 bits. It is also noticeable that the RTZ is significantly faster than the NRTZ over the complete word length range even though the NRTZ requires less bits to encode its data in most cases.

Figure 5.26 illustrates that the throughput of the RTZ ADD is six times greater than the synchronous ADD for short word lengths, but only four times greater than the NRTZ ADD. However, the throughput of the NRTZ ADD is approximately two times greater than the synchronous ADD for 1 to 3 bit length words. The synchronous ADD is approximately 100% more efficient than the RTZ and NRTZ ADD for long words.

Table 5.4 shows that the RTZ and NRTZ ADD functions are approximately twenty-one and thirty-four times larger, respectively, than the synchronous ADD. The NRTZ ADD is approximately 64% bigger than the RTZ ADD.

5.6.2.6 Analysis

In this case the RTZ ADD is faster than the NRTZ ADD, whereas the NRTZ AND was faster than the RTZ AND. The main reason for this reversal can be attributed to the NRTZ symmetric switch which increases the latency of the NRTZ data path as compared with the RTZ equivalent. The NRTZ ADD is significantly bigger in circuit area than the RTZ because of the extra circuitry required to implement the NRTZ ADD function, symmetric switch and the increased complexity of the multiplexor control circuits.

Any clock skew that might detrimentally effect the synchronous ADD's performance would have to increase its latency by approximately 222% to make it as slow as the NRTZ ADD when it operates on the same maximum length 16-bit word.

5.6.3 The Comparator Experiment

The comparator function is considered here by the development of the MAX comparator, which outputs the larger of two input variable length bit-serial operands. The MIN comparator design is virtually the same as the MAX comparator, except it outputs the smaller of two input operands. Therefore, only the implementation characteristics, and not the design, of the MIN comparator are considered in this section (The RTZ and NRTZ MIN comparator implementations can be seen Appendix A12 and Appendix B12, respectively).

5.6.3.1 The Self-timing Requirements

The self-timed MAX comparator compares the *n*th bit of the first operand (operand A) to the same *n*th bit of the second operand (operand B), and sets a status flag that records which of the two operands is the greater for that *n*th bit position. This process of comparing the same *n*th bit position for the two operands is repeated up until the end of the longest word. For example, for two 8-bit operands, if the 4th bit of operand A is set to logic '1' and the 4th bit of operand B is logic '0' then the status flag is set to indicate that operand A is the greater up until the 4th bit position. If both bits are set to the same logic level then the status flag is left unchanged. The status flag setting at the end of the last bit comparison indicates which of the two words has the greater magnitude. The greater of the two operands can then be output. The advantage of a self-timed MAX comparator is that the functions of comparing the two operands and assigning the larger to the destination RBU register can take place in the same instruction cycle, as compared to the synchronous comparator which requires two instruction cycles.

Pseudo bits must be appended to the shorter word therefore enabling the data-driven comparator logic function to continue processing the longer word. This is because of the fundamental mode of operation which would cause the circuit to deadlock if one operand were shorter than the other.

All data path components, unless stated otherwise, are implemented with NRTZ circuitry, and all control circuitry is realised using RTZ circuitry, for reasons outlined previously in section 5.4.2.

5.6.3.2 The RTZ Experiment

The RTZ Requirements

The requirements of the RTZ MAX function are the same as those stated for the RTZ logic function except that the data path is only monitored and not manipulated. Therefore, requirements relating to the manipulation of the data path operands do not apply to the RTZ MAX comparator.

The RTZ MAX Comparator Architecture

The RTZ MAX function is shown in figure 5.27. The functionality of MUX0, MUX1, MUX CONTROL 0 and MUX CONTROL 1 are the same as those explained previously for the RTZ AND function.

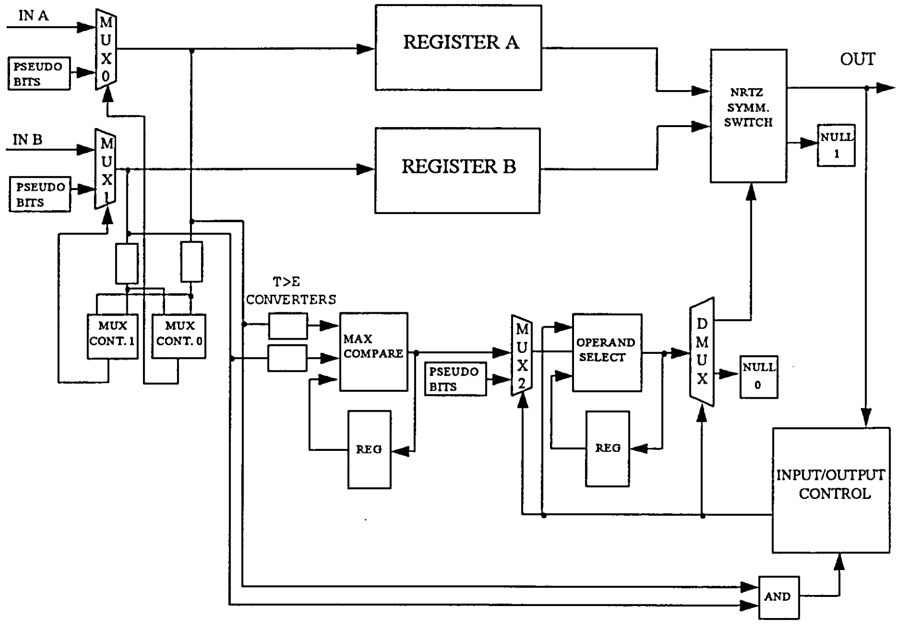


Figure 5.27. The RTZ MAX comparator.

REGISTER (0 & 1): These registers store the operands as they are input because the larger operand cannot be output until after the last bit comparison.

MAX COMPARE: The role of this circuit is to perform the comparison of the two bit-serial operands. It sets its output, called the operand select bit, to indicate which is the larger. It operates as follows. The *n*th bit of operand A is compared to the same *n*th bit of operand B (for all *n*) where the output, or the operand select bit, is set to either logic ‘0’, indicating operand A is the greater, or logic ‘1’, indicating that operand B is greater. If both *n*th bits are the same then the result of the previous comparison, fed back as input into the present comparison, is output.

NRTZ Symmetric (SYMM) SWITCH: The purpose of this device is to output the greater operand from the comparator and send the smaller operand to a null circuit where it is discarded. This circuit is controlled by DMUX. This device operates as follows. If the control input is set to logic ‘0’, indicating that operand A is the larger, then the contents of register A are sent to the output of the EXU, and the contents of register B are sent to the null circuit where they are discarded. If the control input is set to logic '1' then the opposite occurs.

DMUX: The purpose of this circuit is to pass the operand select bits to the control input of the NRTZ SYMM SWITCH if the last bit comparison has completed. If the words are still in the process of being compared then the operand select bit is sent to a null circuit where it is discarded, so that the operand cannot be output before the last bit comparison has taken place.

MUX2: The role of this device is to pass either the actual or pseudo operand select bits to the operand select circuit. The actual operand select bits are passed when the operands are still in the process of being compared (control input = logic '0'). Pseudo operand select bits are output when the last bit of both operands have been compared (control input = logic '1') until the greater operand has been output from the EXU (control input returns to logic '0').

OPERAND SELECT: This device outputs each operand select bit to DMUX, if the last bit comparison has not taken place. Otherwise, it passes the pseudo operand select bits, after the last bit comparison, which are set to the logic level of the previous actual operand select bit; to do this the output is fed back.

AND: The role of this device is to signal, to the INPUT/OUTPUT CONTROL, when the EOW Tag of both words have been detected. In valid data phase only a data-false value is output independent of the input logic level of each input.

INPUT/OUTPUT CONTROL: This circuit indicates to MUX2, DMUX and OPERAND SELECT whether the input operands are still in the process of being compared (output = logic '0') or the greater of the two has been determined (output = logic '1').

5.6.3.3 The NRTZ Experiment

The NRTZ Requirements

The requirements of the NRTZ MAX function are the same as those stated for the NRTZ logic function except that the data path is only monitored and not manipulated.

In addition, each logic '1' EVENd bit output from MAX COMPARE is converted to logic '0', except in feedback. This is because the last bit comparison, used to select the operand to be output, will always be of an ODDd phase. This allows the circuitry succeeding the MAX COMPARE to be the same as that used for the RTZ.

The NRTZ MAX Comparator Architecture

The NRTZ MAX Comparator function, shown in figure 5.28, is very similar to the RTZ equivalent except that the MUX CONTROL 0 & 1 circuits are the same as those used in the NRTZ AND architecture, and the NRTZ implementation of the comparator operator function.

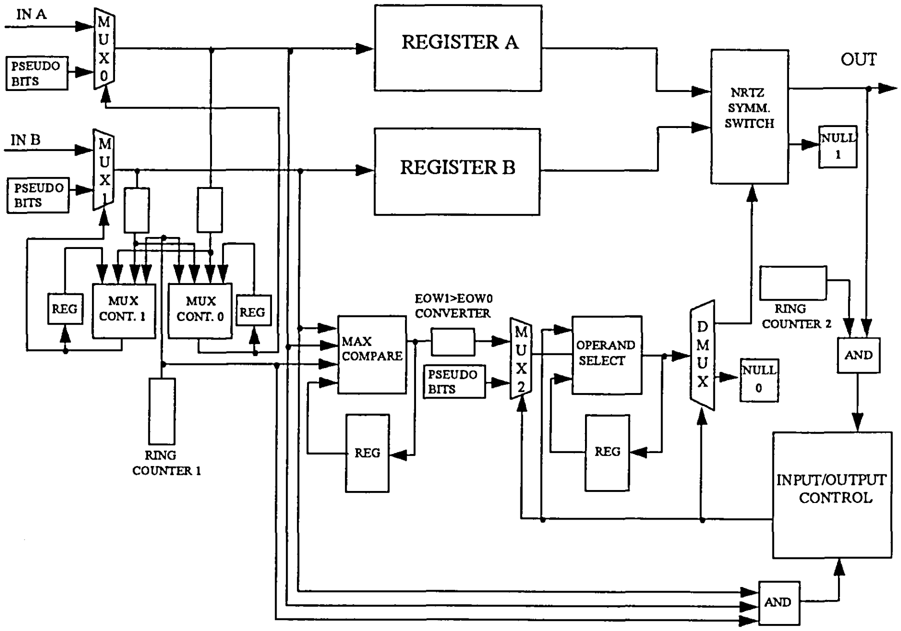


Figure 5.28. The NRTZ MAX comparator.

In order to differentiate between EOW0 or EOW1 and data bits, two ring counters, explained previously in chapter 4, are required so that control, and comparison, of the input operands and the output of the largest operand can operate independently. Consequently, RING COUNTER 1 feeds MAX COMPARE, MUX CONTROL 0 & 1 and the three input AND circuit, where the AND signals to the input/output controller when the EOW1 is detected for both input operands. RING COUNTER 2 feeds the two input NRTZ AND which in turn signals to the INPUT/OUTPUT CONTROL when the end of the output operand is detected.

MAX COMPARE: This circuit is an NRTZ implementation of the previously explained RTZ equivalent. The other main difference of this circuit is that it incorporates an extra input that reads RING COUNTER 1. In the event that the RING COUNTER input is an EOW1 then the MAX comparator simply ignores the values on its data inputs and outputs the value read from its feedback input, therefore the value of the operand select bit is unchanged.

AND (Three-input): Signals to INPUT/OUTPUT CONTROL when both the input operands are EOW1 tag bits. This is achieved by ANDing each operand input with the RING COUNTER output. If the RING COUNTER output value is EOW1 and both operand inputs are also EOW1 then an EOW1 is output to INPUT/OUTPUT CONTROL signalling the end of both input operands. If any of the inputs are EOW0 then an EOW0 is output to INPUT/OUTPUT CONTROL indicating that more data is to arrive.

AND (Two input): Operates in the same manner as the 3-input AND gate except that it monitors only the comparator resultant operand output and the output of ring counter 2.

5.6.3.4 The Synchronous MIN/MAX Comparator

The Synchronous MIN/MAX comparator [82], shown in figure 5.29, performs its function over two instruction cycles. In the first instruction cycle the two input bit-serial operands are compared such that the *n*th bit of the first operand (operand A) is compared to the same *n*th bit of the second operand (operand B) for the entire 16-bit word length; thus taking one complete instruction cycle. On each bit cycle the comparator output is set to logic '1' if the *n*th bit of operand B is the greater, logic '0' if the *n*th bit of operand A is the greater, or to the logic level of the previous comparator output, if the *n*th bits of the operands are the same. After the last bit comparison the greater of the two operands is then known.

In the second cycle the comparator circuit is disabled though its output logic level is maintained. The multiplexor (MUX), controlled by the comparator and the MIN/MAX control input from the instruction decoder, then re-reads the two operands and outputs either the maximum operand signalled by the comparator, if the MIN/MAX control input is to logic '1', or the minimum operand if the MIN/MAX control input is set to logic '0'.

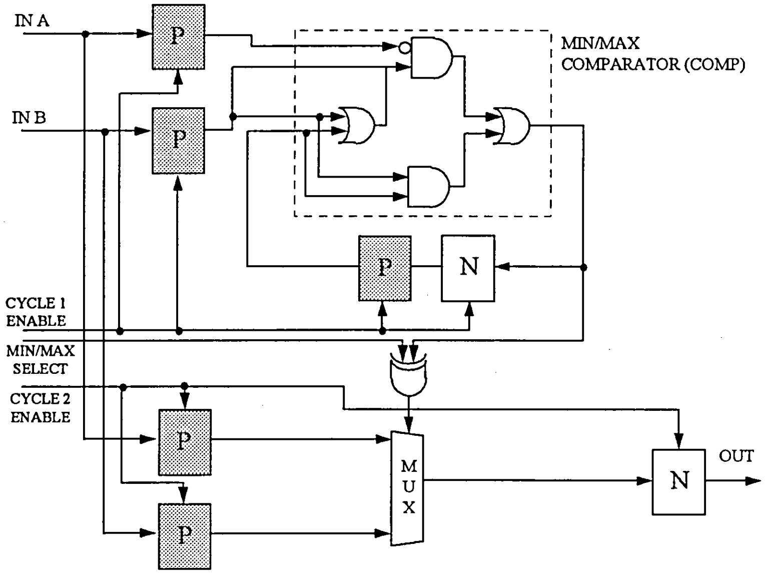


Figure 5.29. The synchronous MIN/MAX comparator.

5.6.3.5 Results

Figure 5.30 illustrates the latency results for the comparator function. **Figure 5.31** and **Figure 5.32** present the throughputs of the RTZ and NRTZ MIN and MAX functions, respectively, compared with the synchronous MIN/MAX comparator. Finally, **Table 5.5**, lists the hardware results for all three versions of the MIN and MAX functions.

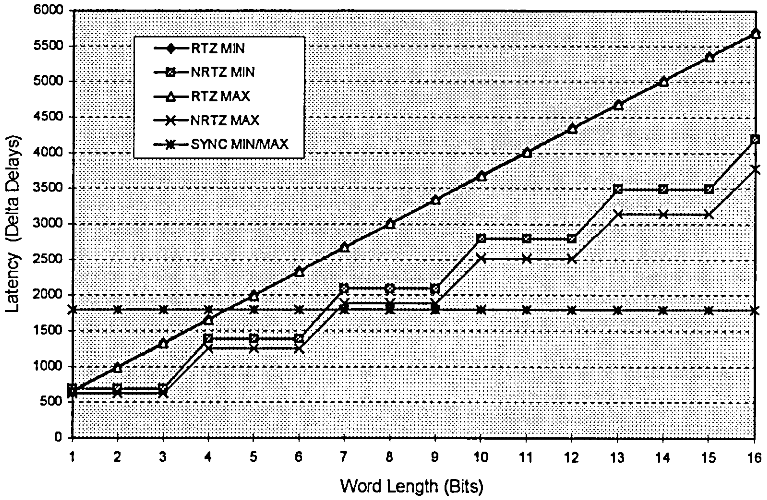


Figure 5.30. The RTZ, NRTZ and synchronous MIN and MAX latencies.

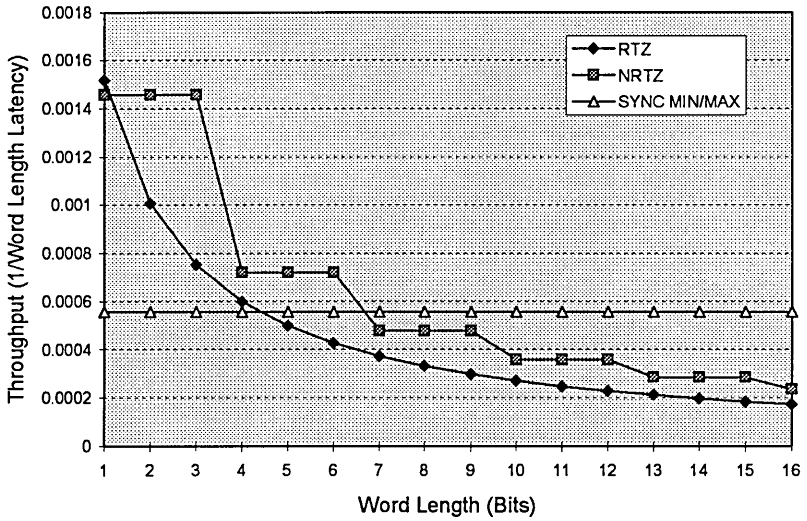


Figure 5.31. The RTZ, NRTZ and synchronous MIN throughputs.

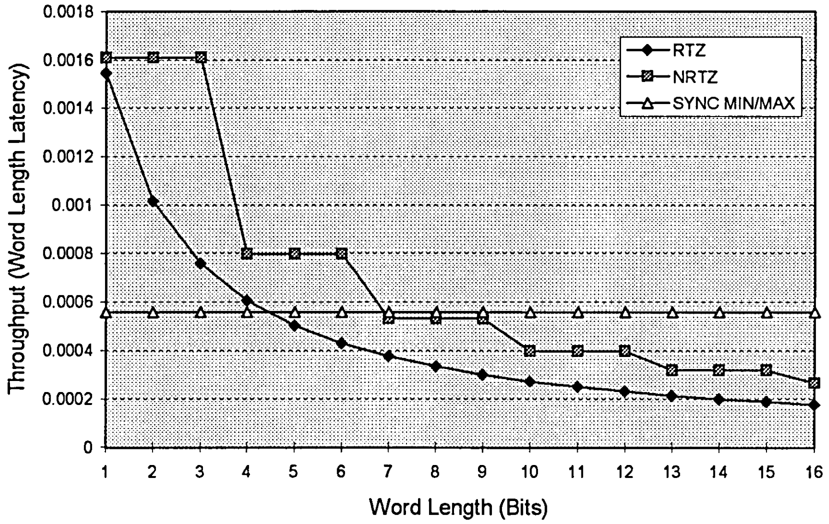


Figure 5.32. The RTZ, NRTZ and synchronous MAX throughputs.

	RTZ	NRTZ	SYNC
MAX	2905 gates	3086 gates	89 gates
MIN	3264 gates	3558 gates	As above

Table 5.5. Gate counts for the RTZ, NRTZ and synchronous MIN and MAX functions.

The following observations have been made:

Figure 5.30 shows that the RTZ MIN and MAX are faster than the Synchronous MIN/MAX comparator for words of up to 4 bits in length. The NRTZ MIN and MAX comparators are faster still than the synchronous equivalent for words of up to 6 bits in length, and are significantly faster, over the complete range of bits, than the RTZ MIN and MAX functions.

Figures 5.31 and **5.32** show that the throughputs of the RTZ and NRTZ MIN and MAX comparators are comparable. The throughputs of the RTZ and NRTZ implementations are greater than the synchronous equivalent for short word lengths, but becomes less than half as efficient than the synchronous MIN/MAX when the word size increases to the maximum word length.

Table 5.5 shows that the RTZ and NRTZ MAX functions are respectively thirty-three and thirty-five times larger than the synchronous MIN/MAX, whereas the RTZ MIN and MAX functions are thirty-seven and forty times bigger than the synchronous MIN/MAX, respectively. The NRTZ MAX and NRTZ MIN comparators are approximately 6% and 9% bigger than their RTZ counterparts, respectively.

5.6.3.6 Analysis

The NRTZ MIN and MAX are faster than their RTZ counterparts because the same data is encoded using a smaller number of bits.

The RTZ and NRTZ comparator designs are significantly slower and larger than the other EXU functions because of their significant control circuit overhead.

Any clock skew that might detrimentally effect the synchronous MIN/MAX performance would have to increase its latency by approximately 229% to make it as slow as the RTZ MIN or MAX comparators when they operate on the same maximum length 16-bit word.

5.7 Self-timing Design Guide-lines

This section presents the design guide-lines for the implementation of self-timed data path architectures: data communication (DOU), data storage (RBU) and data execution (EXU) functions. These guide-lines were determined during the previous experiments in this chapter for each data path unit and can be sub-divided into guide-lines for general self-timed implementation (independent of the encoding technique) and RTZ and NRTZ encoding and tagging guide-lines.

Design Guide-line Assumptions

The RTZ and NRTZ data path functions share two main design commonalties. Both these commonalties, stated below, have been explained in sections 5.4.2 & 5.4.3 and are common to each of the RTZ and NRTZ data path functions addressed in this section:

1. The data path must be realised using NRTZ circuitry.
2. All local control is to be realised using RTZ circuitry.

In a few cases an amendment to these guide-lines is required and will be highlighted, in the appropriate sections, in *italic type*.

Data Communication

General Self-timing Guide-lines

1. The transmitting PE must synchronise the acknowledge signals received from multiple destination PEs on each bit cycle.
2. If the transmitting PE is to communicate with a select number of its total receivers then only the acknowledge signals from the selected receivers must be synchronised on each bit cycle.
3. The receiving PEs selected to read the data must be identified on each bit cycle to the transmitting PE in order to synchronise the appropriate acknowledge signals.
4. Due to the fundamental mode of operation pseudo acknowledge signals must be enforced on the acknowledge feedback lines from non-reading PEs, on each bit cycle, to the synchronising C-element thus allowing it to change state.
5. Each direction of data communication from the outputting device is buffered in order to absorb processing latency differences between the multiple receivers.

Data Storage

General Self-timing Guide-lines

1. Each register must have prior knowledge of whether it is to communicate with the RBU demultiplexor and/or one or both RBU multiplexors in order that the appropriate acknowledge signals between these units can be synchronised.
2. Due to the fundamental mode of operation, if a register is to synchronise with only one of the RBU multiplexors then a pseudo acknowledge signal must be enforced on the input into the synchronising C-element from the multiplexor which is not to read the data.
3. If a register is to act as a source, and not the destination, then the data must be re-written into the register in order to maintain data integrity. This necessitates the following:
   a) The acknowledge signal received from the register's data input must be synchronised with the actual and pseudo acknowledge signal(s) of the receiving and non-receiving multiplexors, respectively.
4. If a register is to act as the destination only, then the old data in the register must be pseudo read, in other words discarded, in order to maintain data integrity. This necessitates the following:
   a) A pseudo acknowledge signal must be sent to the register's output stage to imitate a read. This signal must be synchronised with the pseudo acknowledge signals of both multiplexors and pseudo re-write acknowledge.
5. If the register is to act as a destination and a source then the old data automatically vacates the register, thus the old data is not re-written.

Execution

This section is sub-divided into the logical, addition and comparison design guide-lines.

Logical

General Self-timing Guide-lines

1. Pseudo bits must be appended to the end of the shorter data word input into the function.

RTZ Guide-lines

1. The data path is implemented using NRTZ circuitry, *except for the operator function which is implemented using an RTZ circuit.*
2. Local control is realised using RTZ circuitry, except for the following NRTZ requirement:
   a) Valid data bits monitored by the local control circuitry are converted to empty bits before they enter the RTZ control circuitry. In addition, each empty bit is converted to the equivalent logic '0' data valid bit and each EOW tag bit is converted to the equivalent logic '1' data valid bit.
3. Pseudo bits alternate between a logic '0' data valid bit, to coincide with data valid bits of the longer word, and an EOW Tag bit, to coincide with the empty bits or tag bit of the longer word, until the EOW is detected for both input operands.
4. An EOW Tag bit must be converted to empty bit before it is input into the logic function.
5. As a consequence of guide-line 4, an empty bit output from the logic function is converted back to an EOW Tag bit only when both inputs into the logic function are EOW Tag bits.

NRTZ Guide-lines

1. Local control is realised using RTZ circuitry, except for the following NRTZ requirement:
   a) Data path bits must be converted into their RTZ equivalent encoding before they enter the RTZ control circuit function. This means ODD phase data bits are converted empty bits, EVEN phase bits are converted to logic '0' data valid bits, and EOW0 and EOW1 tag bits are converted to their equivalent logic '0' and logic '1' data valid bits.
2. Pseudo bits must alternate between logic '0' ODDd, logic '0' EVENd, logic '0' ODDd and EOW1 bits, matching the 4-bit tagging period, such that each bit coincides with the same phase of bit of the longer operand.
3. Each EOW1 bit must be converted to an EOW0 bit before it enters the logic function.

4. An EOW0 output from the logic function must be converted to an EOW1 only when both inputs into the logic function are EOW1.

Addition

General Self-timing Guide-lines

The general guide-lines of the logic function also apply to the addition function. The following guide-line is also necessary:

2. Pseudo bits must be appended to the end of both operands in the event of carry overflow.

RTZ Guide-lines

Logic function guide-lines 1 through 4 also apply to the addition function. However, guide-line 5 must be amended for the addition function, described as follows:

5. As a consequence of guide-line 2, an empty bit output from the logic function is converted back to an EOW Tag bit only when both inputs into the logic function are EOW Tag bits and there is no carry overflow.

NRTZ Guide-lines

NRTZ logic function guide-lines 1 through 4 apply to the NRTZ addition function, however, guide-line 5 must be amended. In addition, three further guide-lines specific to the NRTZ addition function are required.

5. An EOW0 bit output from the logic function must be converted to an EOW1 only when both inputs into the logic function are EOW1 and there is no carry overflow.
6. An ODD phase carry bit preceding the input of EOW0 bits must be propagated to the addition of the ODD phase data bits succeeding the EOW0 bits.
7. An ODD phase carry bit must be converted to the equivalent EVEN phase data bit in readiness for the next bit addition cycle, and vice versa for EVEN to ODD conversion.

Comparison

General Self-timing Guide-lines

Logic function guide-line 1 also applies to the comparator. In addition, a second guide-line is required for the comparator function:

2. If the comparison is to made in one instruction cycle the data operands must be internally stored in the comparator until the last bit comparison, after which the appropriate operand can be output.

RTZ Guide-lines

RTZ Logic guide-lines 1 through 4 also apply to the RTZ comparator.

NRTZ Guide-lines

NRTZ Logic guide-lines 1 through 4 also apply to the NRTZ comparator.

5.8 The Selection Guide-lines

This section proposes guide-lines to aid computer architects in the selection of the appropriate RTZ, NRTZ or synchronous design technique, depending on the required SA data path implementation characteristics. These selection guide-lines are based on the circuit characteristics produced for each of the RTZ, NRTZ and synchronous data path functions and are categorised into performance, based on the latency and throughput results, and gate count selection guide-lines, discussed as follows.

Performance Guide-lines

The performance of self-timed data path functions is dynamic in that it is dependent on the length of the data words operated upon. The performance of the synchronous data path functions is fixed to a static worst case speed, even if the magnitude of the data item is less than the magnitude represented by the fixed-position most significant bit (MSB); in this case one or more trailing logic '0' bits follow the encoded data for each bit position up to the pre-determined MSB.

The following guide-lines propose the appropriate technique to adopt to maximise a function's performance when considering the number of bits required to encode the data item independent of the design technique:

- If the word length of the environment data items varies over a range of 0 to 8-bits then NRTZ data path functions are the most appropriate to adopt in order to benefit most from the fast processing speed of short length words.
- If the data set magnitude varies between 8 to 16-bits then synchronous data path functions should be adopted because their worst case speed performs consistently better than both the RTZ and NRTZ data path functions operating on longer data words.
- If the data word length varies over the mid range, of between 4 to 12-bits, or the complete range, of between 1 to 16-bits, then the selection of the appropriate design technique is more difficult because the processing speed of the RTZ and NRTZ data path functions over these ranges are very similar to the fixed worst case speed synchronous data path functions. The required circuit area is therefore the deciding factor.

The guide-lines suggested above are based on the best case performance for each of the RTZ, NRTZ and synchronous data path functions, however, their final performance is dependent on other units that supply data and control information to their inputs. Therefore, selection of the appropriate design technique is directly related to the overall performance of the data path and control path when connected together. Consequently, final decisions on the selected implementation technique for the data path functions cannot be made until the performances of the RTZ, NRTZ and synchronous PEs are determined; presented in chapter 7.

Circuit Area Guide-lines

The following guide-lines propose the appropriate technique to use for different circuit area requirements:

- The gate counts of the RTZ and NRTZ data path functions are considerably larger than the synchronous equivalents. Therefore, if the desired circuit area of the processor is required to be small in order to make benefit of the maximum number PEs that can be fit onto a single chip, then the synchronous design technique is the best technique to use.

- However, if circuit area is not an imperative issue then the NRTZ design approach, which is only slightly bigger than the RTZ, is the best to adopt in order to make use of the average case processing speed.

5.9 Conclusion

In this chapter three experimental studies were presented realising new self-timed RTZ and NRTZ architectures for the register bank unit, data output unit and execute unit data path functions. Each of the three above stated studies followed a standard experimental methodology that resulted in three important results:

- New guide-lines to aid in the design of RTZ and NRTZ self-timed bit-serial data path functions. This included general guide-lines that consider delay-insensitive self-timed design issues for the three afore-mentioned data path functions. In addition, guide-lines that consider encoding and tagging design issues were presented.
- New guide-lines to aid in the selection of the appropriate design technique for a given set of criteria.
- Comparison of the implementation characteristics of the RTZ, NRTZ and synchronous data path functions. This has shown that the performance of the RTZ and NRTZ data path functions is better than the synchronous equivalent for short data word lengths. However, the circuit area of the RTZ and NRTZ data path functions are significantly greater than their equivalent synchronous counterparts.

CHAPTER 6

THE CONTROL PATH

6.1 The Objectives of the Chapter

The development of design guide-lines for RTZ and NRTZ control path functions is the main aim of this chapter. This aim is achieved through the design of the instruction bypass unit (IBU), instruction decode unit (IDU) and instruction fetch unit (IFU) control structures. The performance and cost characteristics for the RTZ and NRTZ control path functions are compared against each another and with their synchronous counterpart. From these results selection guide-lines are determined to identify the most efficient design method for each control path functions for a chosen application environment.

In addition to the control path structures identified above, three further control path functions, presented in the following section, are required to control the bit and instruction cycles of the ST-SISA. The aims identified above also apply to these three functions, however, because these functions are specific to the ST-SISA, comparisons with the SISA are not necessary.

6.2 Introduction to the Control Path

The ST-SISA control path is shown in figure 6.1. It contains several functions: the IBU, IFU, and IDU, and the newly presented over-write unit (OWU), result control unit (RCU) and the PE status unit (PSU). The role of the three fore mentioned control path units has been previously described in chapter 4.

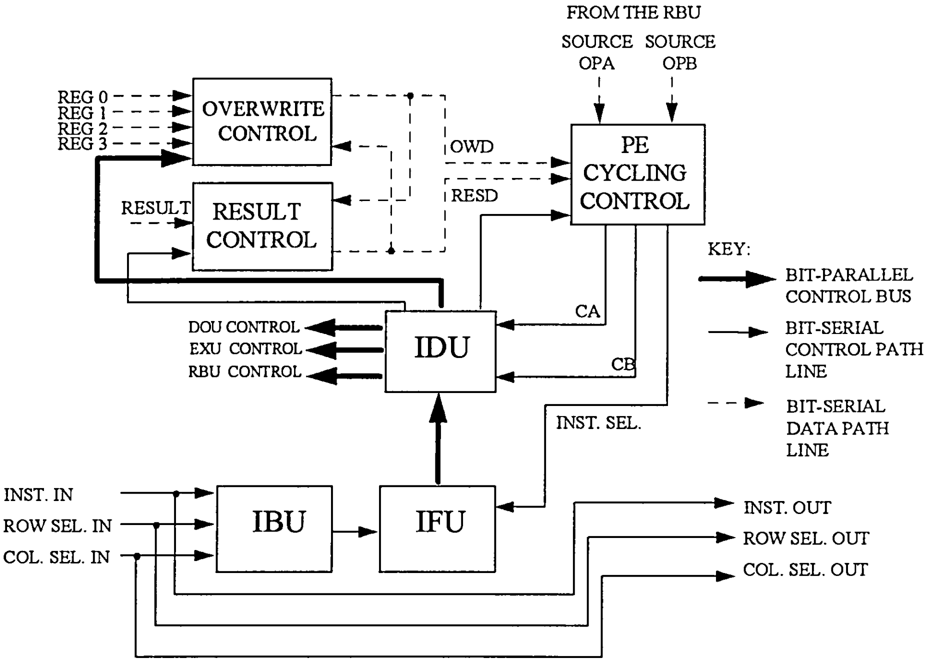


Figure 6.1. The control path.

The latter three control path functions are required because the ST-SISA operates in instruction cycles where each instruction cycle is sub-divided into a dynamic number of bit-cycles. The number of bit-cycles is determined by the length of the longest data word operated upon in the instruction cycle. As a consequence, it is necessary to co-ordinate the various sub-functions of the ST-SISA for both bit and instruction cycles. The RCU, OWU and PSU provide the required functionality, explained as follows:

Over-Write Unit (OWU): The role of the OWU is to allow the destination register of the RBU to dispose of its old data, by performing a pseudo read of its output, thus making space in the register for the new result data from the EXU. The OWU also signals to the PSU on each bit cycle whether or not the EOW Tag of the destination register's old data has been detected, indicating the end of the over-write operation.

Result Control Unit (RCU): The purpose of the RCU is to monitor the result output from the EXU in order to signal on every bit cycle to the PSU whether it has detected the EOW Tag bit, thus indicating that the selected EXU function has completed its operation, or if it has detected a data bit (or otherwise), indicating that the selected EXU function is not yet complete.

PE Status Unit (PSU): The role of this unit is to monitor the two RBU source operand outputs in order to send their status, indicating if either or both are EOW Tags or not, on every bit cycle to the IDU and IFU. This information enables the IDU to correctly disable the source operand data path that is no longer required; when one word is longer that the other. Both source operand data paths are disabled when a MIN/MAX operation completes the comparison of the source operands before the result can be written to the destination register. The PSU in addition monitors the status signals from the OWU and RCU, such that when these and both the source operand outputs indicate they have detected a tag, the PSU can then signal to the IFU to accordingly select or re-select the next or current instruction for transmission to the control path.

6.3 Experimental Methodology and Assumptions

To ensure consistency throughout this thesis the proposed investigations in this chapter adopt the same experimental and design methodologies as detailed in chapter 3 and Appendix D, respectively.

The experimental assumptions and adopted metrics are the same as those specified in previous chapter, in section 5.3.

6.4 The Instruction Bypass Unit (IBU)

The function of the IBU is to output the two instruction words (fixed to eight bits in length) to the IFU if both the row and column selector bits are set to logic '1'. Otherwise, if either or both selector bits are set to logic '0', the instruction words are simply discarded by the IBU.

6.4.1 Self-timing Requirements

The row and column selectors are input into the IBU with the first bit of each instruction word. Therefore, the row and column selectors must not only control the path taken through the IBU for the first instruction bit of both words but for all succeeding bits of both words in order to correctly select or deselect the entire instruction. However, this causes problems because the fundamental mode of operation dictates that the row and column selector bit inputs must transpire through a state change for each instruction bit input, otherwise the IBU will deadlock. In order to overcome this problem the IBU must internally regenerate the row and column selector bits for each bit cycle of the IBU.

6.4.2 The RTZ Experiment

The RTZ Requirements

The EOW Tag bit is used to signal the end of each instruction word. Therefore, when the EOW Tag bit is detected by the IBU, this indicates the next row and column selector bits can be read by the IBU.

All instruction path components are realised using NRTZ circuitry to cater for the EOW Tag (as described in section 5.4.2). In addition, because the selectors are RTZ encoded and untagged, all selector path components are realised using RTZ circuitry.

The RTZ IBU Architecture

The RTZ IBU Architecture is shown in figure 6.2.

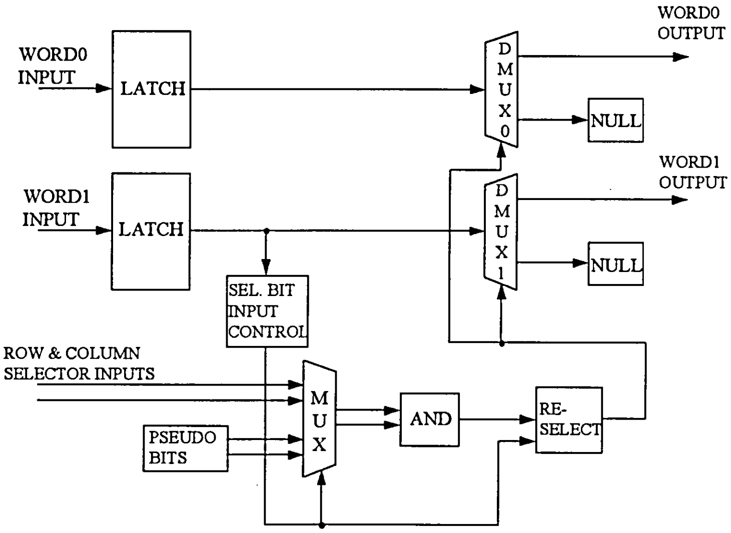


Figure 6.2. The RTZ IBU.

MUX: The role of this device is to pass the actual row and column selector bits to the AND circuit, associated with first bit of an instruction, or internally generated pseudo row and column selector bits. It operates as follows. Whenever the control input is set to logic '1', indicating that the present instruction bit is an EOW Tag, the next row and column selector bits are passed to the AND circuit which consequently controls the selection of the next instruction word. However, if the control input is set to logic '0', then row and column pseudo bits are passed to the output as no more selector bits can be read for the current input instruction.

SELECT (SEL) BIT INPUT CONTROL: The purpose of this device is to signal to MUX the type of instruction bit as each passes through the IBU. It outputs a logic '1' bit when an EOW Tag has been detected, or a logic '0' when it detects an empty bit, and an empty bit when it detects a valid bit of either logic level.

RE-SELECT: This device changes the logic level of the ANDed pseudo row and column selector bits to the logic level of the previous ANDed actual row and column selector bits until the end of the instruction word. It operates as follows. Whenever the control input is set to logic '1' the bit received from AND is output unchanged. If the control input is set to logic '0' then the bit received from AND is set to the logic level of the previous RE-SELECT output.

DMUX(0, 1): The purpose of this device is to output its respective instruction word to the IFU when the row and column selector bits are both set to logic '1', indicated when the control input is set to logic '1'. If the control input is set to logic '0' then the instruction word is sent to a null circuit where it is discarded.

LATCH: Buffers each instruction bit, taken from the instruction stream, before they are passed to the demultiplexor.

NULL: Simply discards the received instruction bit in the event one or both of the row and column selector bits are set to logic '0'.

6.4.3 The NRTZ Experiment

The NRTZ Requirements

The requirements of the NRTZ circuit are very similar to the requirements of the RTZ circuit, except that it operates on an untagged NRTZ instruction stream. The untagged approach was adopted because instruction words are fixed to a length of eight bits. Therefore, if the NRTZ tagging technique were used to signal the end of a word then four extra bits (two EOW0, one EOW1 and an used data bit) would be required to encode each and every instruction word. So instead, an NRTZ ring counter is used to count each instruction bit input into the IBU for every instruction word. When the eighth bit is reached, the next row and column selector bits and instruction are input to the IBU.

All instruction path components are implemented with NRTZ circuitry and the selector path components are realised using RTZ circuitry (as described in section 5.4.3).

The NRTZ IBU Architecture

The NRTZ IBU Architecture, shown in figure 6.3, is the same as the RTZ IBU architecture except for the inclusion of the 8-bit ring counter instead of the select input control circuit.

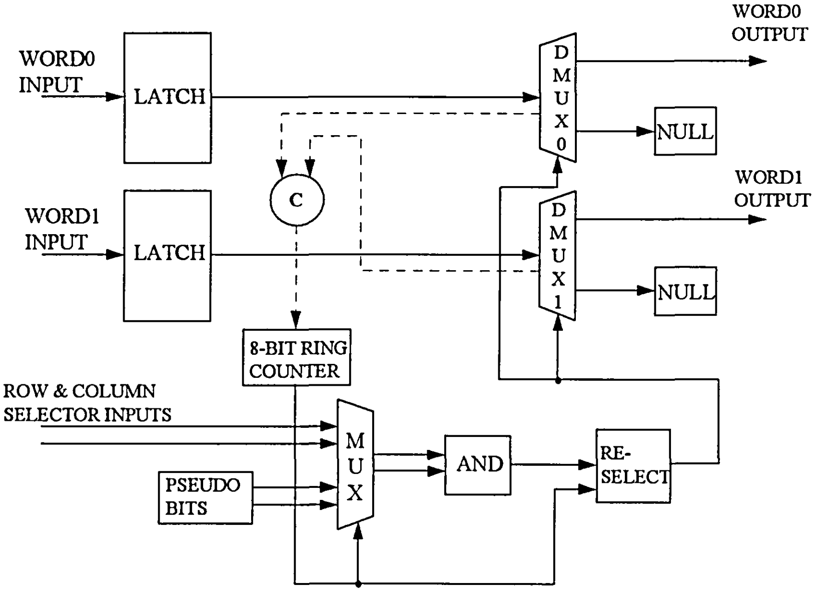


Figure 6.3. The NRTZ IBU.

8-BIT RING COUNTER: The function of the ring counter is to count each instruction bit input into the IBU in order to signal the receipt of the eighth bit. It operates as follows. Whenever a instruction word bit is output from the IBU the ring counter rotates its internal buffer. When the eighth bit is read the output from the ring counter is set to logic '1', otherwise it is logic '0', thus signalling to the MUX to read in the next row and column selector bits.

6.4.4 The Synchronous IBU

The role of the synchronous IBU [82], figure 6.4, is slightly different to that of the ST-SISA IBU in that it inputs the control information in parallel, converted from serial to parallel by the IFU. It then passes the bit-parallel instruction information to the IDU either unchanged or converted into a NOP to deselect it if either or both of the row and column selector bits are set to logic '0'.

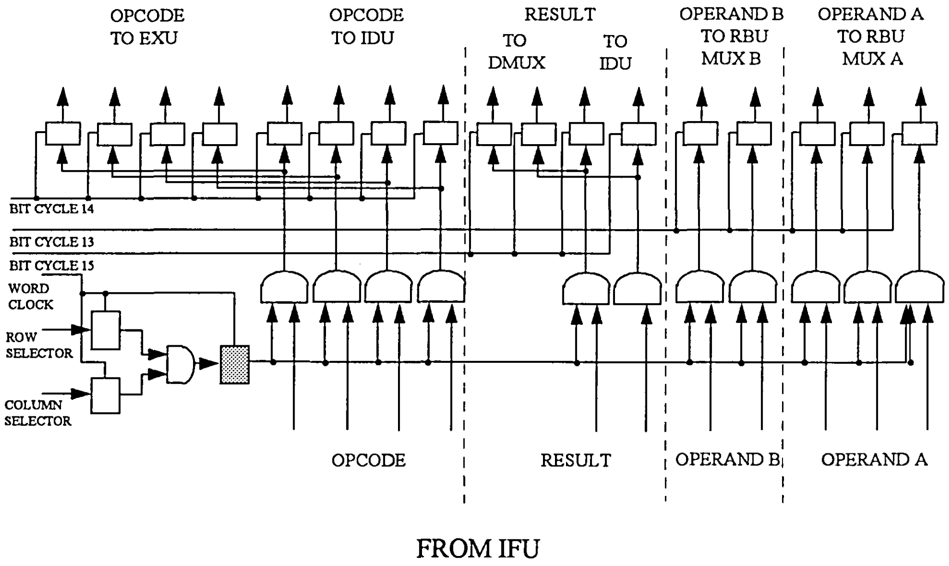


Figure 6.4. The synchronous IBU.

The IBU operates as follows. The row and column selector bits, associated with each instruction, are read and ANDed together on each word cycle. The output from the row and column AND gate is then used as a select mask bit by each of the instruction bit AND gates, such that each gate outputs its respective input instruction bit, if the select mask bit is set to logic '1', or outputs a logic '0', if the select mask bit is set to logic '0'. Each of the instruction fields are then latched at the IBU output, so that the appropriate instruction fields can be read by the IDU, RBU and EXU.

6.4.5. Results

Figure 6.5, **Figure 6.6** and **Table 6.1** illustrate the latency, throughput and hardware results, respectively, for the RTZ, NRTZ and Synchronous IBU functions.

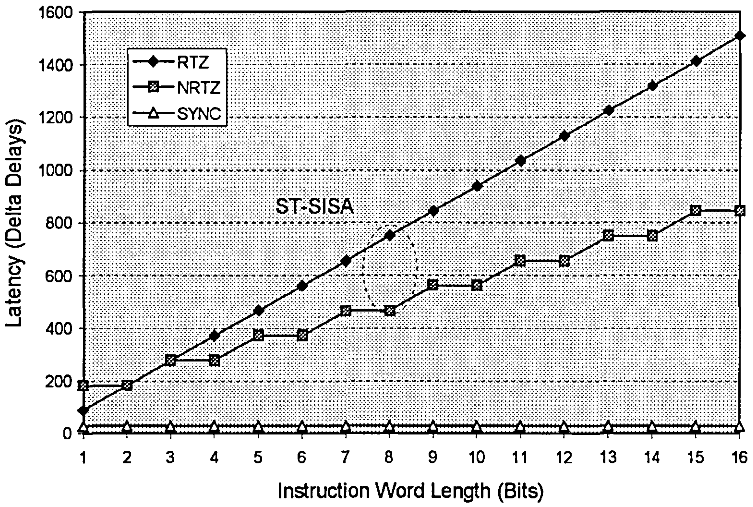


Figure 6.5. The RTZ, NRTZ and synchronous IBU latencies.

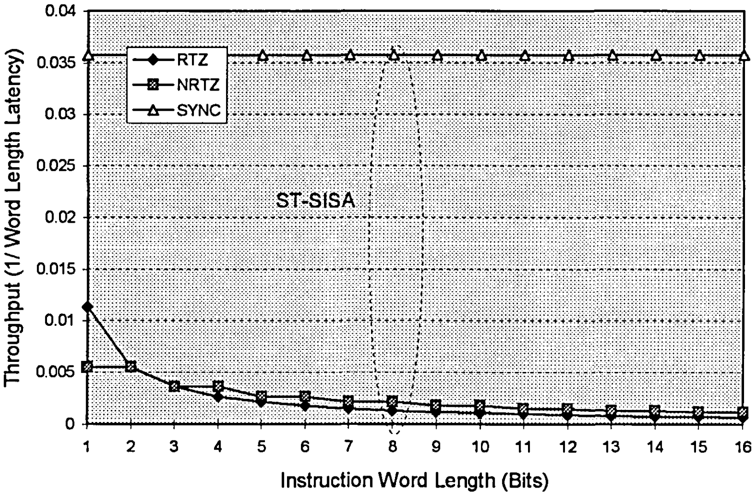


Figure 6.6. The RTZ, NRTZ and synchronous IBU throughputs.

	RTZ	NRTZ	SYNC
IBU	446 gates	538 gates	422 gates

Table 6.1. The RTZ, NRTZ and synchronous IBU gate counts.

The following observations have been made:

Figure 6.5 shows that the latency of the synchronous IBU is considerably less than that of the RTZ and NRTZ designs. It can also be seen that the NRTZ IBU performs better than the RTZ IBU.

Figure 6.6 shows that the throughput of the synchronous IBU is considerably greater than both the RTZ and NRTZ IBUs. The throughput of the NRTZ IBU is slightly greater than the RTZ IBU's throughput.

Table 6.1 shows that the RTZ and NRTZ IBUs are approximately 6% and 27% bigger than their synchronous counterpart. The NRTZ IBU is approximately 21% bigger than the RTZ IBU.

6.4.6 Analysis

The self-timed and synchronous designs differ in that the latter operates in parallel as opposed to the former which operates in serial. Therefore, because of the inherent large size of delay-insensitive circuits, and the bit-parallel implementation of the synchronous design, the three circuits are comparable in circuit area. In addition, because the synchronous design operates in parallel it is considerably faster than the RTZ and NRTZ designs. The NRTZ RBU latency and throughput is greater than the RTZ IBU because it operates on less instruction bits because of the fewer bits required by the NRTZ encoding technique.

Any clock skew that might detrimentally effect the performance of the synchronous IBU would have to increase its latency by approximately 6000% to make it as slow as the RTZ IBU when it operates on the same maximum length 16-bit word.

6.5 The Instruction Fetch Unit (IFU)

The purpose of the IFU is to convert the bit-serial instruction into the equivalent bit-parallel format before sending it to the IDU; in order to conform with the existing synchronous SISA design.

6.5.1 The Self-timing Requirements

The IFU takes the bit-serial instruction and then converts it into the bit-parallel equivalent before sending the result and operand address codes to the RBU and IDU, and the instruction code to the EXU and the IDU. It then re-sends the bit-parallel instruction on every ST-SISA bit cycle until the end of the instruction cycle. This is a result of the fundamental mode of circuit operation which requires that each control circuit module receives control information each time it guides a single data bit through its controlled device. Consequently, the IFU must store and re-send the bit-parallel instruction until the EOW Tag bit has been processed by the RBU's DMUX. The IFU can then fetch the new instruction from the IBU when the appropriate control signal is received.

6.5.2 The RTZ Experiment

The RTZ Requirements

In order to transmit the bit-parallel representation of the RTZ instruction to the relevant functional units, each (V)alid data bit, and its succeeding (E)mpty token, in each bit-serial instruction word must be separated and distributed over their own dedicated control line. In addition, because the control path is completely RTZ encoded, the (T)ag bit, succeeding the eighth bit, must be converted to an empty token before it is transmitted to its appropriate function(s). This functionality is illustrated in figure 6.7. As a consequence of this functionality all the RTZ IFU circuitry, after the tag is converted to empty bit, is implemented using RTZ circuitry.

The RTZ IFU Architecture

The RTZ IFU Architecture, depicted in figure 6.7, fetches eight instruction bits, therefore two IFUs, one for each instruction word, are required to fetch all sixteen instruction bits.

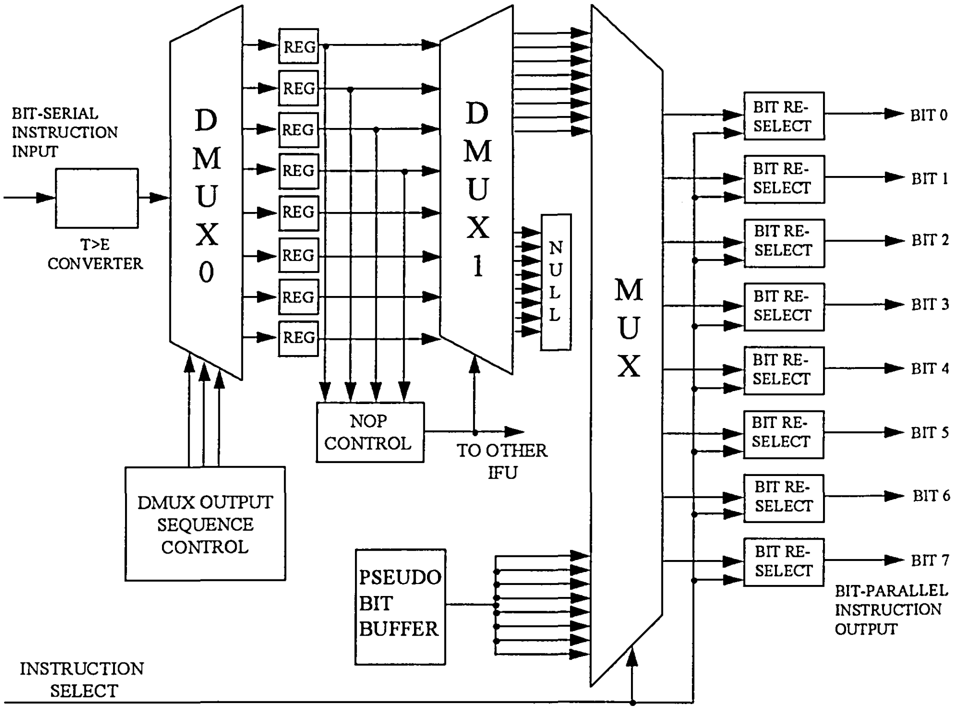


Figure 6.7. The RTZ IFU.

DMUX0: The role of this device is to convert the bit-serial instruction word into an equivalent bit-parallel representation by sending the *n*th valid bit (where *n* is the bit position), with its corresponding empty token, to the corresponding *n*th output. Therefore, bit0 is passed to DMUX output 0, bit1 to output 1, bit2 to output 2 and so on, in ascending order, until the end of the word. This process is repeated for every input instruction word. This device is controlled by DMUX OUTPUT SEQUENCE CONTROL, explained in the next paragraph.

DMUX OUTPUT SEQUENCE CONTROL: The purpose of this device is to control DMUX0 enabling it to perform its function as stated in the previous paragraph. It operates as follows. A bit-parallel address incremented on each bit cycle is output to DMUX0, so that the corresponding 0 through to 8 outputs of DMUX0 are selected in ascending order.

DMUX1: The function of this device is to discard instructions, if the select input is set to logic '1' indicating a NOP instruction has been detected. Otherwise, if the instruction is something other that a NOP, the control input is set to logic '0', then it is sent to MUX for output to the IDU.

NOP CONTROL: The purpose of this circuit is to differentiate between a NOP and all other instructions in order that NOPs are discarded and all other instructions are sent to the IDU. It is realised using a simple 4-input NAND circuit that monitors the opcode field of the instruction. It sends a logic '1' control bit to DMUX1 when the current op-code is a NOP, otherwise logic '0' is sent indicating a new instruction is to be executed by the PE. Only one NOP control function is required for the two IFUs.

MUX: The role of this circuit is to send either the new bit-parallel instruction word, when the previous instruction is finished, or pseudo bits to the IDU, on each bit cycle of the current instruction. It operates as follows. If the control input is set to logic '1', the new instruction word is sent to the IDU. Otherwise, if the control input is set to logic '0', then pseudo bits are passed through MUX.

BIT RE-SELECT: The role of this device is to set the logic level of pseudo bits to the logic level of the first control bit so that the instruction can be regenerated on each bit cycle, following the first bit cycle, of an instruction. It operates as follows. A new instruction bit is output if the select input is set to logic '1'. Otherwise, if the select input is set to logic '0' then the pseudo bit, received from MUX, is set to the logic level of the previous control bit which was output to the IDU.

6.5.3 The NRTZ Experiment

The NRTZ Requirements

The requirements of the NRTZ IFU are very similar to the RTZ IFU except for the need to convert the NRTZ instruction into the equivalent RTZ instruction because the NRTZ ST-SISA incorporates an RTZ control path.

Two problems must be considered to convert an NRTZ bit-serial instruction into a equivalent bit-parallel representation. Firstly, EVEN phase instruction bits, 1, 3, 5 & 7, see figure 6.8, after being converted to parallel must then be converted to (O)DD phase instruction bits, otherwise they will be in the incorrect phase for the IDU. Consequently all circuitry that precedes and includes this functionality is implemented using NRTZ circuitry to cater for the NRTZ instruction. All circuitry succeeding the conversion of the NRTZ instruction to its RTZ equivalent is implemented in RTZ circuitry.

Secondly, an empty token must be appended to the end of each newly converted bit-parallel data-valid instruction bit to ensure the correct RTZ sequence of data-valid and empty states. The required functionality is illustrated in figure 6.8, showing the transformation of the NRTZ bit-serial word into the RTZ bit-parallel word.

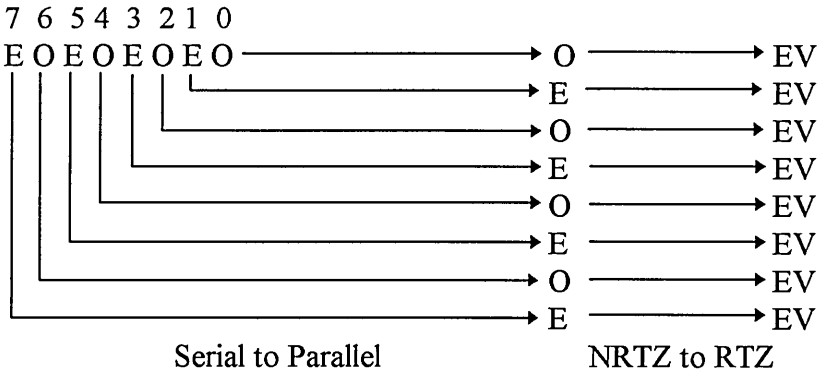


Figure 6.8. NRTZ bit-serial to RTZ bit-parallel.

The NRTZ IFU Architecture

The NRTZ IFU architecture is shown in figure 6.9.

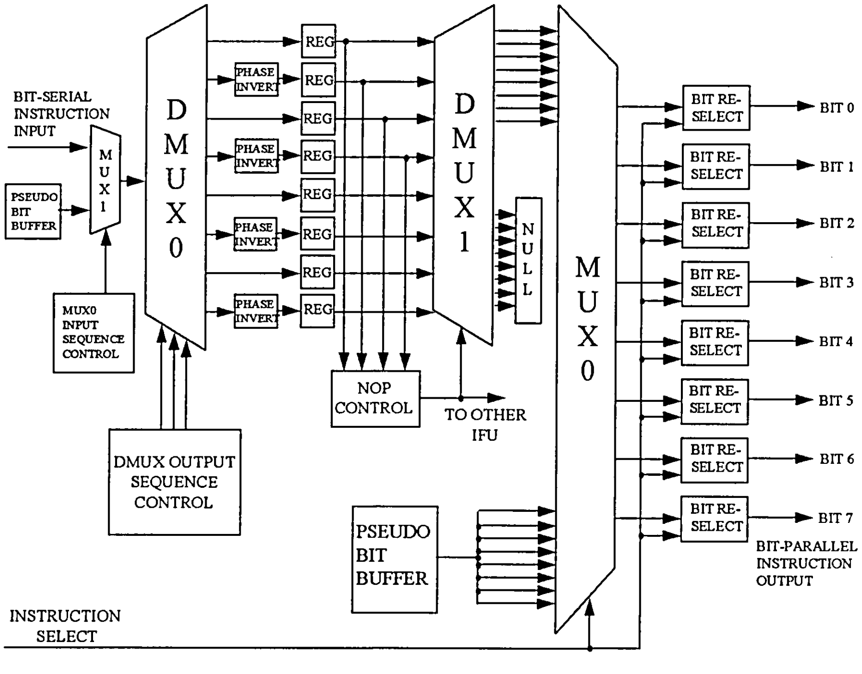


Figure 6.9. The NRTZ IFU.

DMUX OUTPUT SEQUENCE CONTROL: The role of this device is to control the outputs of DMUX0 such that instruction bits are passed to the outputs in the order in which they were input. This is followed by passing alternate data (logic '0') and empty pseudo bits in descending order to the DMUX0 outputs; which are then appended to the new bit-parallel instruction word. In other words, the output bit-parallel address word is incremented on each bit cycle, 0 through to 7, before being decremented on each bit cycle to 0 (e.g. 0,1,2,3,4,5,6,7,7,6,5,4,3,2,1,0).

MUX1: The role of this device is to send either the NRTZ encoded instruction, or a sequence of eight NRTZ logic '0' pseudo bits, to DMUX0.

MUX0 INPUT SEQUENCE CONTROL: The role of this device is to select the input port of MUX1. It does this by outputting a sequence of eight NRTZ encoded logic '1' bits, which correspondingly cause MUX1 to pass the eight instruction bits to DMUX0, followed by a sequence of eight NRTZ logic '0' bits, which cause MUX0 to send eight logic '0' pseudo bits to DMUX1.

PHASE INVERT: The role of this device is to change the phase of an EVEN phase instruction bit to an ODD phase instruction bit and vice versa.

6.5.4 The Synchronous IFU

The role of the IFU [81], figure 6.10, is to convert the input bit-serial instruction into the equivalent bit-parallel instruction for output to the IBU. The IFU also passes the unchanged bit-serial instruction on to the next PE. The synchronous IFU acts as a source to the IBU whereas the ST-SISA IBU acts as a source to the IFU.

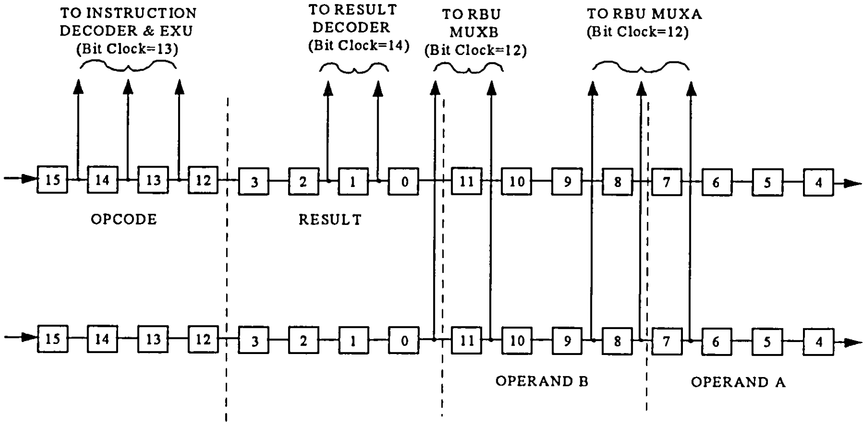

TO INSTRUCTION DECODER & EXU (Bit Clock=13)
TO RESULT DECODER (Bit Clock=14)
TO RBU MUXB (Bit Clock=12)
TO RBU MUXA (Bit Clock=12)
15
14
13
12
3
2
1
0
11
10
9
8
7
6
5
4
OPCODE
RESULT
OPERAND B
OPERAND A

Figure 6.10. The synchronous IFU.

The IFU is realised by two simple 16-bit bit-serial FIFO pipelines in which some of the latch outputs, as seen in the figure 6.10, are fed to the IBU in addition to the succeeding FIFO latch. Each instruction word is divided into two fields, therefore only after the sixteenth bit cycle does each field lie in the appropriate latches.

6.5.5 Results

Figure 6.11 shows the serial to parallel conversion time of the first instruction for the three versions of the IFU. It also shows the time taken to regenerate the bit-parallel instruction for successive data bit cycles; the synchronous IFU has no regenerate cycles.

Figure 6.12 illustrates the time taken by the RTZ and NRTZ IFUs to convert the next instruction succeeding the first instruction from serial to parallel,. This aims to show any reduction in the IFU processing time of the second instruction through the overlapping of regenerate cycles, of the first instruction, with the serial to parallel conversion time of the second instruction.

Figure 6.13 shows the throughput of the RTZ, NRTZ and synchronous IFUs. This figure only shows the results for non-overlapping instructions.

Finally, **Table 6.2** shows the gate count results for the RTZ, NRTZ and synchronous IFU functions.

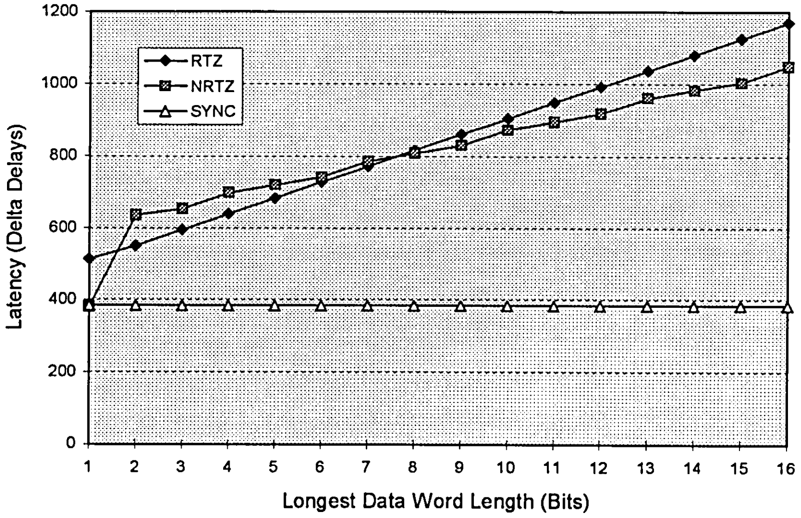


Figure 6.11. The RTZ, NRTZ and synchronous IFU first instruction latencies.

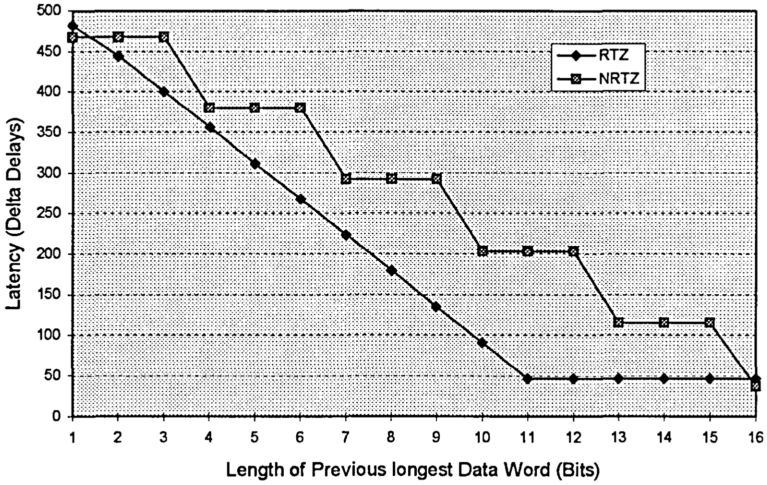


Figure 6.12. The RTZ, NRTZ and synchronous IFU successive instruction latencies.

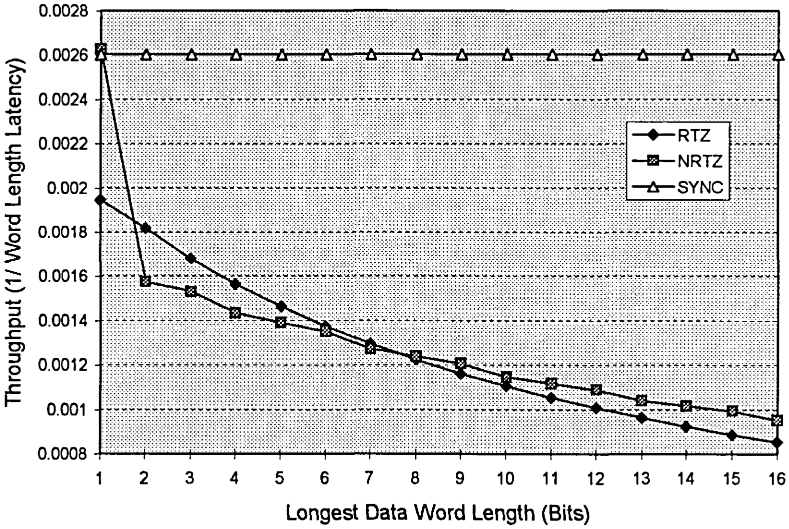


Figure 6.13. The RTZ, NRTZ and synchronous IFU throughputs.

	RTZ	NRTZ	SYNC
IFU	5104 gates	5512 gates	512 gates

Table 6.2. The RTZ, NRTZ and synchronous IFU gate counts.

The following observations can be made:

Figure 6.11 shows that the time taken by the NRTZ IFU to convert the bit-serial instruction from serial to parallel is approximately 100 Δ faster than the time taken by the RTZ IBU. However, the time taken to append empty bits onto the NRTZ bit-parallel instruction, controlling the second NRTZ PE bit cycle, is significantly longer than the RTZ IFU takes to regenerate its instruction code, for its second PE bit cycle. It can also be seen that the NRTZ IFU gradually increases its performance over that of the RTZ IFU as the number of bit cycles increases from 8-bits upwards. The synchronous IFU does not need to regenerate control information and so its latency remains constant.

Figure 6.12 shows that the greater the number of regenerate cycles for the first instruction, the greater the parallel overlap with serial to parallel conversion of the second instruction.

Figure 6.13 shows that the throughput of the RTZ IFU is greater than the NRTZ IFU throughput for short length data words, however, the NRTZ IFUs throughput is greater the RTZ IBU's throughput for long data words.

Table 6.2 shows that both the RTZ and NRTZ IFUs are ten and eleven times larger in circuit area, respectively, than their synchronous counterpart, with the NRTZ IFU requiring 8% more circuit area than the RTZ IFU.

6.5.6 Analysis

The RTZ and NRTZ IFUs are significantly slower than the synchronous IFU due to their increased functionality and high gate count. However, the latency of the RTZ and NRTZ designs can reduced by up to 50 Δ through the parallel execution of the serial to parallel conversion and regenerate functions.

Any clock skew that might detrimentally effect the synchronous IFU's processing speed would have to increase its latency by approximately 200% to make it as slow as the RTZ IFU when it operates on the same maximum length 16-bit word.

6.6 The Instruction Decode Unit (IDU)

The IDU takes the complete instruction from the IFU and then converts into a suitable format in order to co-ordinate and control the operation of the various sub-units of the data path.

6.6.1 The Self-timing Requirements

A number of the required control signals have been previously identified in the DOU and RBU experiments: the source operand A & B register selects, destination register select, wait select and over-write select for each register and the write select for the DOU.

In addition to these control requirements two other control signals are necessary. The first is used to automatically select the DOU output of the RBU DMUX in the event of a write operation, because the result field is used to select the direction(s) in which the DOU is to output data. Consequently, when a write instruction is issued the RBU DMUX control input word must be forcibly set to 0 in order to select the DMUX's DOU output.

The second control signal is necessary to disable the source data path, for operand B, when only a single operand instruction is executed, such as an ASSIGN or WRITE instructions. Otherwise, the PE would deadlock because operand B's data path would not be able to operate upon any data. The sub-units that must be disabled are the RBU MUXB and EXU MUXB.

6.6.2 The RTZ Experiment

The RTZ Requirements

The entire RTZ IDU is implemented in RTZ circuitry because it only ever operates upon RTZ encoded bit-parallel instructions.

The RTZ IDU Architecture

The RTZ IBU architecture is shown in figure 6.14.

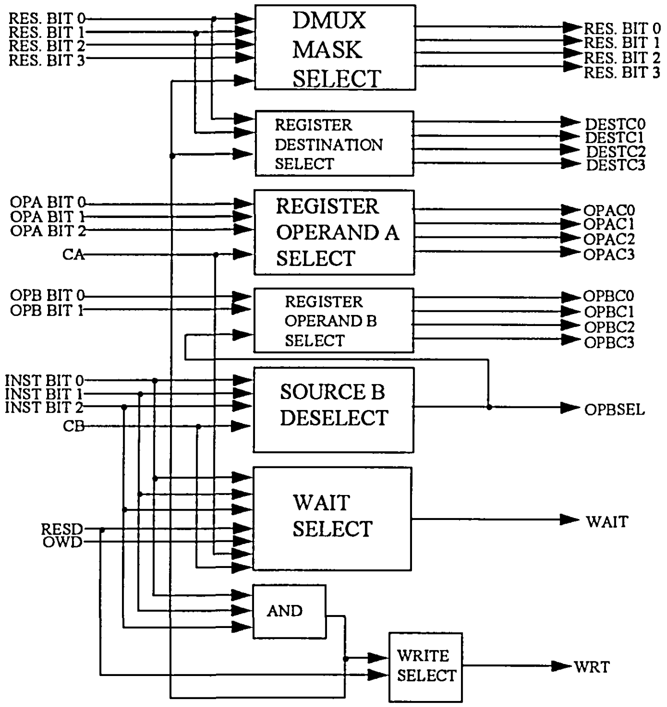


Figure 6.14. The RTZ IDU.

OPERAND A SELECT: The role of this device is to decode operand A read from the IFU so that the appropriate control signals are generated for the control of the four registers. It operates as follows. It decodes operand A and sets to logic '1' one of four dedicated register control lines (OPAC0, OPAC1, OPAC2 or OPAC3), thus signalling to the corresponding register that it is read by RBU's MUXA; all other lines are set to logic '0' signalling to their corresponding registers that they are not read by MUXA. When the entire data word has left the selected register, CA is set to logic '1', signalling to the decoder to reset the previously selected line, indicating to the corresponding register to cease communicating with RBU MUXA.

DESTINATION SELECT: Same operation as the OPERAND A SELECT except it monitors the result field and the AND of the instruction op-code. If the result of the AND of the instruction is logic '1', indicating that a write instruction has been issued, then the destination select control outputs are set to logic '0' indicating to all registers that they are not used to store the resultant data word.

OPERAND B SELECT: Same operation as above except it monitors OPBSEL, and supplies control bits to the corresponding OPBC0, OPBC1, OPBC2 or OPBC3 outputs.

DMUX MASK SELECT: This device automatically selects the first output of the RBU DMUX, to the DOU, in the event of a write instruction. In this case the control input of the RBU's DMUX is set to 0_{10}, in order to automatically select the DOU output, because the result field is instead referenced by the DOU.

SOURCE B DISABLE: This device disables operand B's data path in the event of a write or assign instruction. It operates as follows. It sets its output to logic '1' to disable the various sub-units of operand B's data path. Otherwise, for any other instruction besides a write or assign it outputs logic '0' indicating to operand B's data path units that they are required.

WAIT SELECT: The role of this device is to disable the result data path in the event of a MIN or MAX operation. If this instruction is executed this device outputs a logic '1' control signal to the EXU and RBU indicating that they must wait for the result data operand from the comparison function. If this signal was not issued then the corresponding sub-units would expect the arrival of a result data bit, which would not arrive, causing the entire ST-SISA to deadlock.

WRITE SELECT: The purpose of this device is to signal to the DOU to read the result address field from the IFU. It operates as follows. When its output is set to logic '1' the DOU reads the instruction result field. Otherwise, when it is set to logic '0' the DOU will ignore the result field.

6.6.3 The NRTZ Experiment

The NRTZ Requirements

The requirements for the NRTZ IDU are the same as the RTZ IDU because both adopt RTZ encoded control information.

The NRTZ IDU Architecture

The NRTZ IDU architecture is exactly the same as the RTZ IDU for reasons explained previously in section 6.6.2.

6.6.4 The Synchronous IDU

The synchronous IDU [81], figure 6.15, has two main sub-functions. The first is to decode the result field for use by the RBU. The second is to identify the occurrence of a MIN or MAX instruction and generate controls signals for use by the comparator.

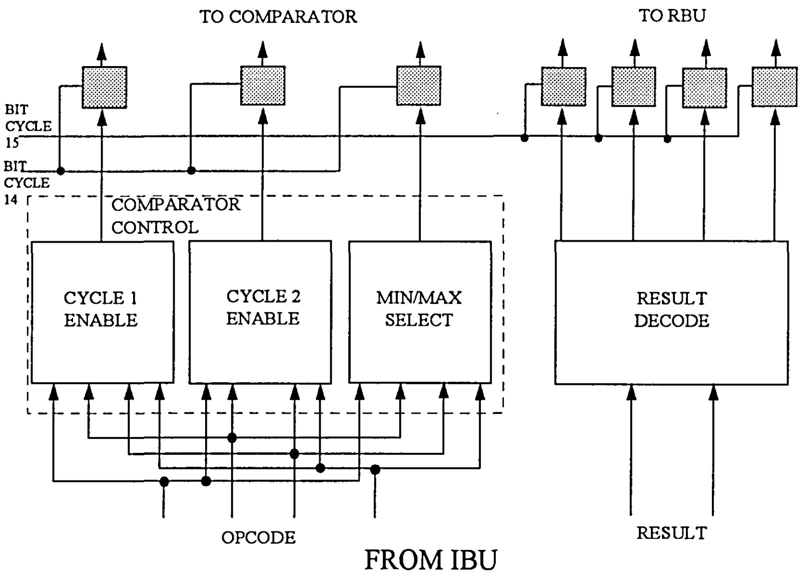


Figure 6.15. The synchronous IDU.

6.6.5 Results

Figure 6.16 and **Figure 6.17** illustrate the latency and throughput, respectively, for the synchronous IDU, and for the RTZ IDU, when it controls RTZ and NRTZ data path circuits. **Table 6.3** shows the gate count results for the RTZ and synchronous IDU functions.

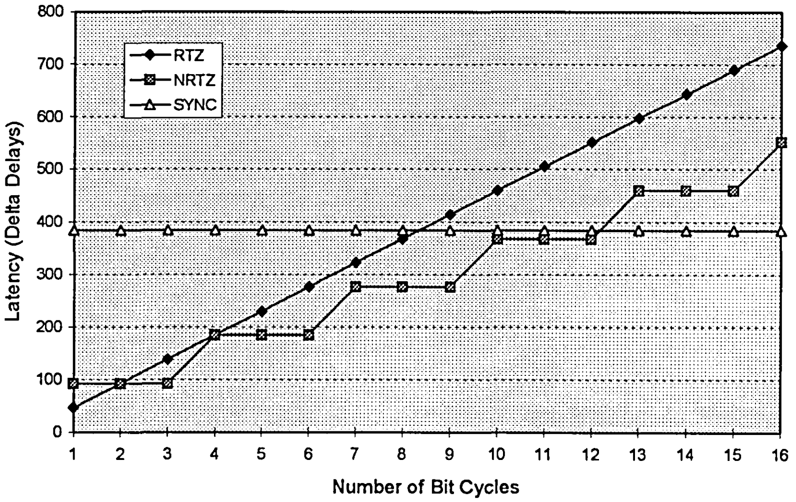


Figure 6.16. The RTZ/NRTZ and synchronous IDU latencies.

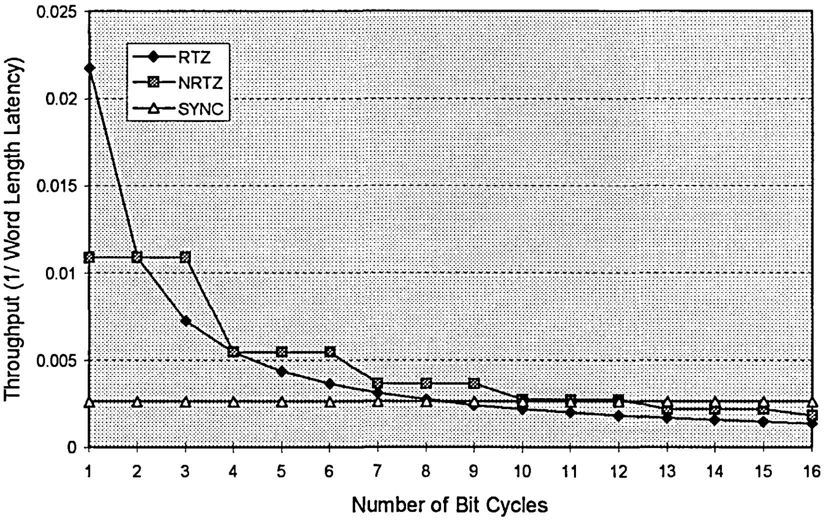


Figure 6.17. The RTZ/NRTZ IDU throughputs.

	RTZ	SYNC
IDU	393 gates	124 gates

Table 6.3. The RTZ, NRTZ and synchronous IDU gate counts.

The following observations have been made:

The RTZ/NRTZ IDU must decode the instruction, and PE status information, on each and every bit cycle. **Figure 6.16** shows that the RTZ/NRTZ IDU is faster than its synchronous counterpart if the longest data word in the instruction cycle is 8-bits or less. However, for data words longer than 8-bits the RTZ/NRTZ IDU takes longer to decode the required control information for the complete instruction cycle than the synchronous IDU.

Figure 6.17 shows that the throughput of the RTZ/NRTZ IDU decreases as the word length is increased. The RTZ/NRTZ IDU throughputs are greater than that of the synchronous IDU for short length words, however for long words the synchronous IDU's throughput is greater. The throughput of the RTZ/NRTZ IDU is greater when it controls an NRTZ data path than a RTZ data path.

Table 6.3 shows that the RTZ/NRTZ IDU is approximately three times bigger than the synchronous IDU.

6.6.6 Analysis

The RTZ/NRTZ IDU must decode the instruction it receives on each and every PE bit cycle, consequently, its performance is relative to the synchronous IDU is dictated by the length of the longest data operand in the instruction cycle. The RTZ/NRTZ IDU latency is reduced when the IDU is used to control the NRTZ data path, because the NRTZ encoding technique requires less bits to encode the same data item than the RTZ encoding technique.

The RTZ IDU has a much higher gate count than the synchronous IDU because of the extra functions it implements, namely the mask select, register control, write select and wait select functions, in addition to the inherent larger size of self-timed circuits.

If clock skew were to detrimentally effect the performance of the synchronous IDU it would have to increase its latency by approximately 92% to make it as slow as the RTZ/NRTZ IDU when it operates on an RTZ 16-bit word.

6.7 The Result Control Unit (RCU)

The Result Control Unit (RCU) monitors the result operand as it is passed bit-serially from the EXU to the RBU. Its main function is to signal to the PSU the type of the bit it is monitoring, differentiating between an EOW Tag bit and a data bit. This enables the PSU to identify if the current bit represents the end, or continuance of the currently selected EXU function.

6.7.1 The Self-timing Requirements

The ST-SISA operates in fundamental mode, consequently, in the event that the destination register's old data has not been overwritten the RCU must continue sending result status signals to the PSU even if the instruction cycle is not yet complete. The RCU is reset once the OWU status signal indicates that its EOW Tag has been detected. Consequently, the RCU, in addition to the PSU, must monitor the OWU status signal.

Another important factor of the RCU to consider is that when a MIN or MAX instruction is executed the RCU will not receive any result bits from the EXU until the EXU has completely read both source operands. This will cause problems because the RCU must produce status output signals on every PE bit cycle otherwise the PSU will deadlock as it will not receive any signal changes on one of its inputs. Therefore, the IDU sends a wait signal to the RCU signalling that it must produce pseudo result status signals before the actual result operand can be read from the EXU.

6.7.2 The RTZ Experiment

The RTZ Requirements

Any pseudo status bits produced after the result EOW has been detected must be set to logic '1' to ensure that the RCU continues to signal the EOW Tag has been detected. Any pseudo status bits produced before a MIN or MAX operation has completed its comparison must be set to logic '0' indicating the end of the word has not been detected.

All data path components are realised using NRTZ circuitry, whereas all control components are realised using RTZ circuitry for reasons explained in section 5.4.2.

The RTZ RCU Architecture

The RTZ RCU Architecture is shown in figure 6.18.

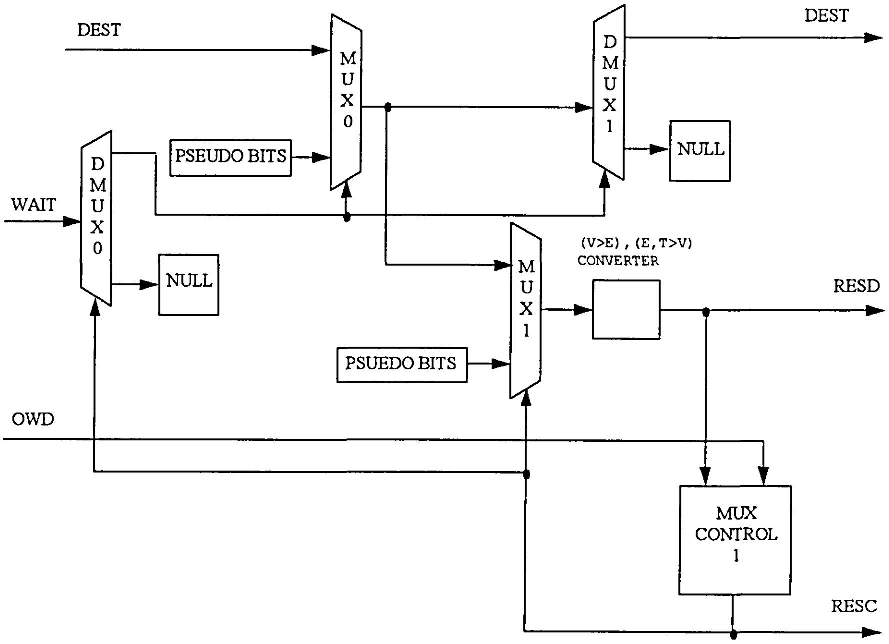


Figure 6.18. The RTZ RCU.

DMUX0: This device controls the passing of the wait control signal, produced by the IDU, to MUX0. It operates as follows. If the control input into DMUX0 is set to logic '0', indicating that the result EOW Tag has not been detected by MUX CONTROL 1, then the wait control signal is sent to MUX0. If the control input is set to logic '1', indicating that the EOW has been detected for the current instruction cycle, then the wait signal is discarded.

MUX0: The role of this device is to pass pseudo bits to MUX1, in the event of a MIN or MAX instruction, to ensure that the controlling device of MUX1 (MUX CONTROL 1) continues to receive state changes on all its inputs to avoid circuit deadlock. It operates as follows. If the wait signal is set to logic '0' then the instruction currently being executed is either an instruction besides MIN or MAX, or a MIN or MAX instruction that has finished processing its operands, and so the result operand can be passed to DMUX1. If the wait signal is set to logic '1' then pseudo bits, set to logic '0', are sent to MUX1 which feeds MUX CONTROL 1. This is necessary because MUX CONTROL 1 must receive state changes on all its inputs.

DMUX1: This device is used to discard the pseudo data bits, sent by MUX0, because they are not read by the RBU. It operates as follows. If the wait signal is set to logic '0', indicating the instruction currently being executed is an instruction besides MIN or MAX, or a MIN or MAX instruction that has finished processing its operands, then the result is passed to the RBU. Otherwise, if the wait signal is set to logic '0' then the pseudo bits received from MUX1 are discarded to ensure the RBU does not receive unwanted pseudo bits.

MUX CONTROL 1: The role of this device is to control the selection of the appropriate output of DMUX0 and input of MUX1. It does this by monitoring the result operand and the over-write data (OWD) from the register bank. In the event the result is an EOW Tag, independent of the logic level of OWD, it selects the next actual result input of MUX1, and allows DMUX0 to pass the wait signal to MUX0 and DMUX1. If the result input is an EOW Tag, and the OWD is not an EOW Tag, then the pseudo input of MUX1 is selected, in order that the MUX control can still receive state changes on its result input while it continues monitoring OWD. Otherwise, if both result and

OWD are EOW Tag bits then the actual input of MUX1 is selected. In addition it allows DMUX0 to discard the wait signal.

MUX1: The role of this multiplexor is to feed either the result word or pseudo data bits to MUX CONTROL 1, OWU and PSU. It operates as follows. If the control input is set to logic '0' then an actual result bit is sent to MUX CONTROL 1, otherwise if the control input is set to logic '1' a pseudo bit is sent.

6.7.3 The NRTZ Experiment

The NRTZ Requirements

Every fourth pseudo status bit produced after the result EOW Tag has been detected must be set to an EOW1 to ensure that the RCU continues to signal that the EOW has been detected. Every fourth pseudo status bit produced before MAX completes its comparison must be set to EOW0 indicating the EOW has not been detected.

All data path components are realised using NRTZ circuitry and all control components are realised using RTZ circuitry.

The NRTZ RCU Architecture

The NRTZ architecture, figure 6.19, is similar to the RTZ architecture except for the need to differentiate between EOW Tag bits and EVEN phase data bits.

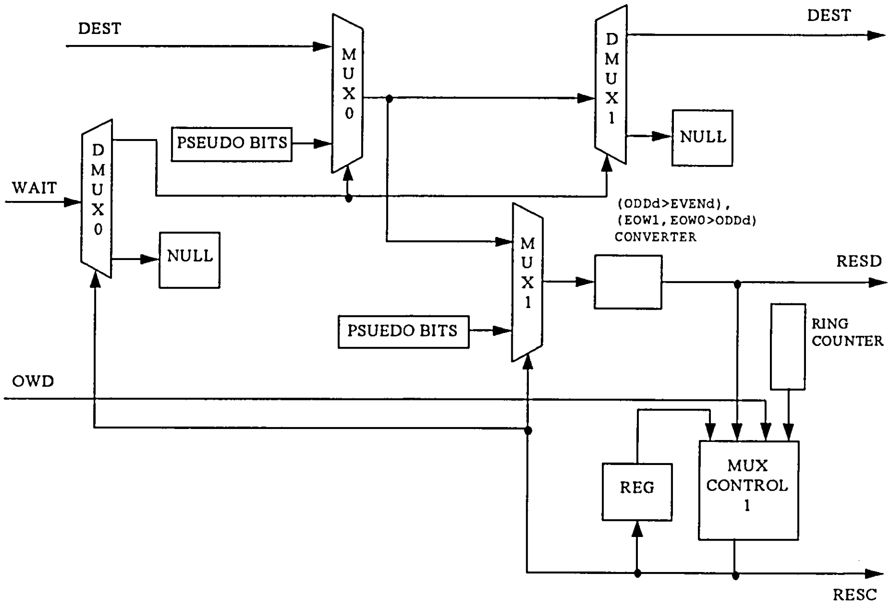


Figure 6.19. The NRTZ RCU.

MUX CONTROL 1: The role of this device is exactly the same as its previously explained RTZ counterpart, only its operation differs. It operates as follows. In the event of a EOW1 on the ring counter input (RCI), and a logic '0' EVENd on the feedback input (FI), MUX CONTROL 1 selects the actual input of MUX1 if the result is an EOW0 and if OWD is either an EOW1 or EOW0, or if both the result and OWD are EOW1 bits. If the result is an EOW1, independent of the logic value on the FI, and OWD is an EOW0, then the pseudo input is selected in order to continue sending bits to the result input of MUX CONTROL 1.

In the event of an ODDd or EVENd on the RCI, MUX CONTROL 1 selects the actual input of the multiplexor if FI is set to logic '0', or the pseudo input if FI is set to logic '1'. Therefore, the intermediary bits, between EOW Tag bits, of either the pseudo or actual result word are selected.

6.7.4 Results

Figure 6.20, Figure 6.21 and **Table 6.4** show the latency, throughput, and gate overhead results, respectively, for both the RTZ and NRTZ RCU functions.

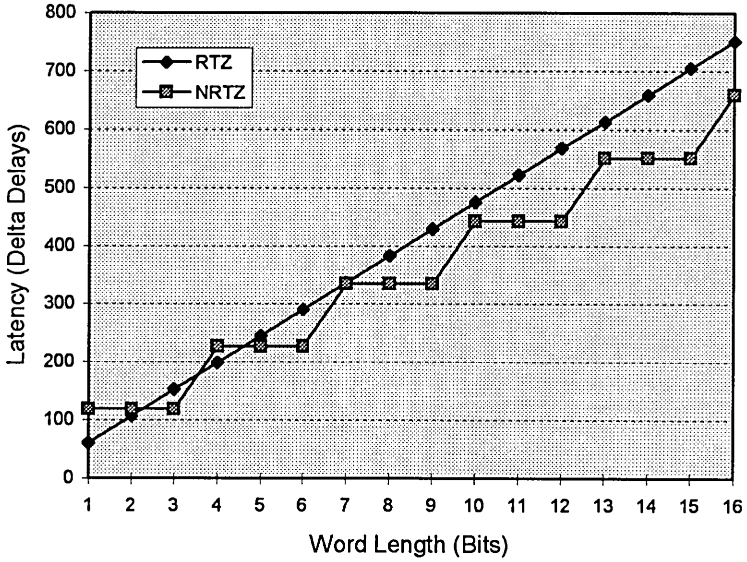


Figure 6.20. The RTZ and NRTZ RCU latencies.

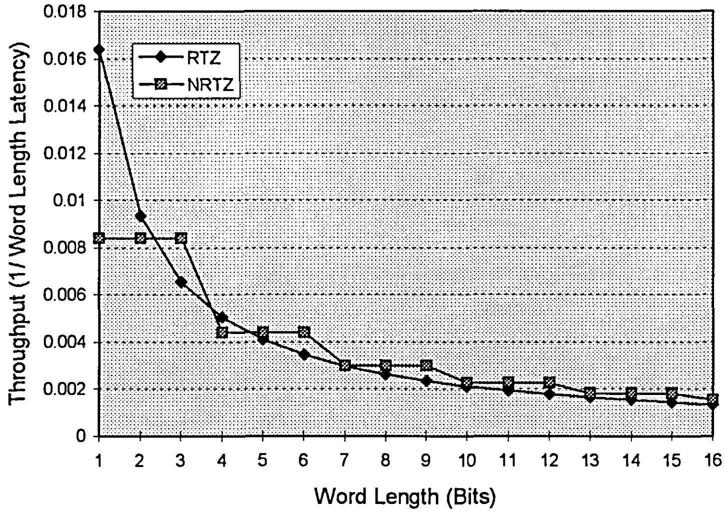


Figure 6.21. The RTZ and NRTZ RCU throughputs.

	RTZ	NRTZ
RCU	407 gates	536 gates

Table 6.4. The RTZ and NRTZ RCU gate counts.

Figure 6.20 shows that the latencies for the RTZ and NRTZ functions are comparable for short and mid-size word lengths. However, the NRTZ RCU performs slightly better than the RTZ RCU as the word size approaches the maximum length.

Figure 6.21 shows that the throughput of the NRTZ RCU is slightly greater than the NRTZ RCU's throughput.

Table 6.4 shows that the NRTZ function is approximately 32% bigger than the RTZ.

6.7.5 Analysis

The NRTZ is slightly faster and its throughput greater than the RTZ because it requires less bits to encode the data that it operates upon. The NRTZ RCU has a higher circuit overhead than the RTZ RCU because of extra circuitry required to implement the mux control circuit, ring buffer counter and feedback storage.

6.8 The OverWrite Unit (OWU)

The function of the OWU is to remove the old data word from the selected destination register, thus creating space in the register for new data.

6.8.1 The Self-timing Requirements

The OWU removes old data from the RBU destination register by supplying an acknowledge signal to the LSB register latch each time a data bit is to be overwritten, thus allowing the current output bit to be replaced by the succeeding bit. This process is repeated until the EOW Tag bit of the old data is detected and discarded. Therefore, the OWU must read the result field address on every bit cycle so it can read the appropriate RBU register.

If the RBU destination register also acts as a source then overwrite acknowledge signals are synchronised with the acknowledge signals from the appropriate RBU multiplexor(s); though it is not necessary to do this, extra control circuitry would have been required to deselect the OWU when the destination register also acts as the source.

The OWU must signal to the PSU, on each bit cycle, whether the overwritten word's EOW Tag has been detected or not. This is required by the PSU to identify the end of the instruction cycle. However, when the overwritten data word is shorter than the new word problems arise because the PSU, like all the circuits, operates in fundamental mode and therefore requires a signal on all of its inputs on each and every bit cycle, otherwise it will deadlock. Consequently, the OWU must continue to provide EOW status signals to the PSU after it has detected the EOW Tag of the overwritten data. The OWU EOW status signal can only be reset after the result EOW Tag bit has been detected, indicating the end of the instruction cycle. Therefore, the OWU must also monitor the status of the result word.

6.8.2 The RTZ Experiment

The RTZ Requirements

The RTZ requirements of the OWU are the same as those specified for the RCU except they apply to the overwrite status signal instead of the result status signal.

The RTZ OWU Architecture

The RTZ OWU Architecture is shown in figure 6.22.

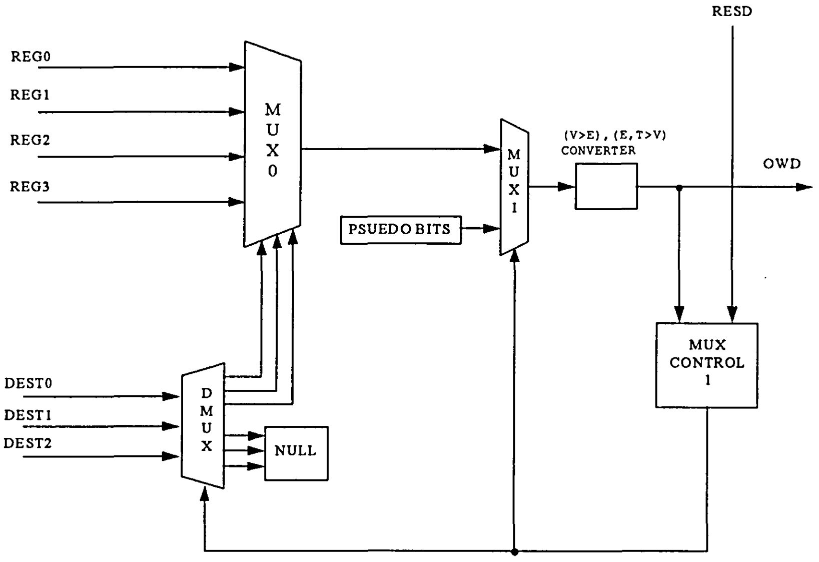


Figure 6.22. The RTZ OWU.

DMUX: The role of this bus demultiplexor is to pass the result address, from the IDU, to the control input of MUX0 to allow it to read the appropriate RBU register output. It operates as follows. If the control input into DMUX is set to logic '0', indicating that the old data in the destination register has not been overwritten, then the result address is passed to MUX0. If the control input is set to logic '1', indicating that the destination register's EOW Tag has been detected, in the current instruction cycle, then the result address is no longer needed and is therefore discarded.

MUX0: The role of this device is to guide the data bit, on each bit cycle, from the appropriate RBU register to MUX CONTROL 1 depending on the address supplied on its control input. It then acknowledges the receipt of each data bit it receives thus allowing the destination register to discard its old data. MUX0 also passes each data bit to MUX1 which feeds the EOW monitoring device (MUX CONTROL 1).

MUX CONTROL 1: The purpose of this module is to control the passing of the result address through DMUX to MUX0, and to control the passing of the data bits or pseudo bits through MUX1 to MUX CONTROL 1. It operates as follows. In the event OWD is not set to an EOW Tag bit, independent of the logic level of RESD, then the next actual result input of MUX1 is selected, allowing DMUX to pass the result address to

MUX0. If the OWD input is an EOW Tag, and the RESD is not an EOW Tag, then the pseudo input of MUX1 is selected, in order that MUX CONTROL 1 can continue to receive state changes on its OWD input while it continues monitoring RESD, and so that DMUX can discard the result address. Otherwise, if both OWD and RESD are EOW Tags then the actual input of MUX1 is selected, and the result address of the next instruction is passed through to MUX0.

MUX1: The function of this multiplexor is to pass the current overwritten data bit or a pseudo data input, under the control of the selector input, to MUX CONTROL 1. If the control input is set to logic '0' then the OWD bit is sent to MUX CONTROL 1, otherwise if the control input is set to logic '1' a pseudo bit is sent instead.

6.8.3 The NRTZ Experiment

The NRTZ Requirements

The NRTZ requirements for the OWU are the same as those specified for the RCU except they apply to the overwrite status signal instead of the result status signal.

The NRTZ OWU Architecture

The NRTZ architecture, shown in figure 6.23, is very similar to the RTZ architecture except for the need to differentiate between EOW Tag bits and EVEN phase data bits. Therefore, changes needed to be made to the mux control circuit, explained as follows:

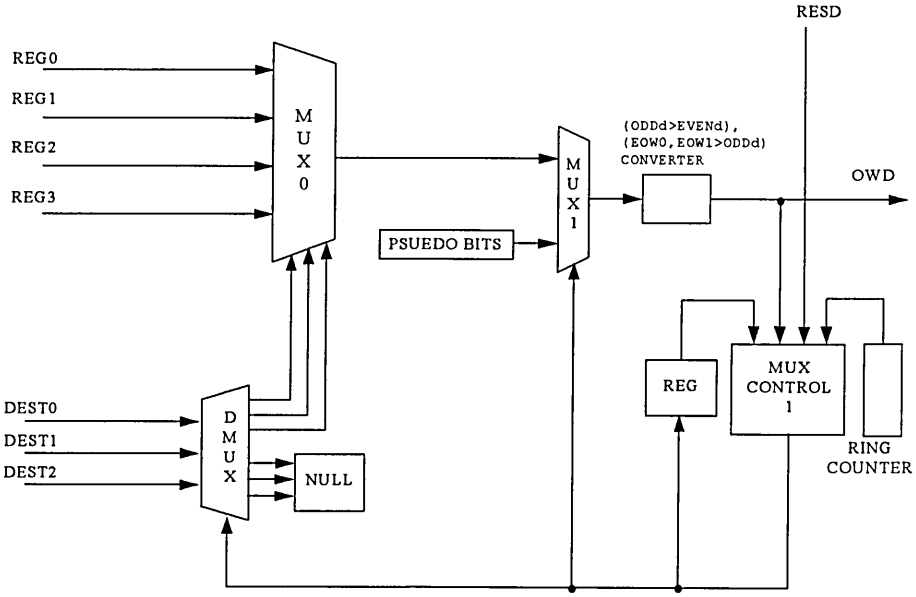


Figure 6.23. The NRTZ OWU.

MUX CONTROL 1: The function of this device is exactly the same as that previously described for the equivalent RTZ OWU, only its operation differs. It operates as follows. In the event of an EOW1 on the ring counter input (RCI), and a logic '0' EVENd on the feedback input (FI), MUX CONTROL selects the actual input of MUX1 if the OWD is a EOW0 and RESD is either an EOW1 or EOW0, or if both the OWD and RESD are set to an EOW1. If OWD is an EOW1, independent of the logic value of the FI, and RESD is an EOW0, then the pseudo input is selected in order to continue sending bits to the result input of MUX CONTROL 1.

In the event of an ODDd or EVENd on the RCI, MUX CONTROL 1 selects the actual input of the multiplexor if FI is logic '0', or the pseudo input if FI is logic '1'. Therefore, the intermediary bits, between EOW Tag bits, of either the pseudo word or the actual OWD are selected.

6.8.4 Results

Figure 6.24 and **Figure 6.25** illustrate the latency and throughput results, respectively, for the RTZ and NRTZ OWUs. **Table 6.5** shows the gate overhead for both the RTZ and NRTZ OWU functions.

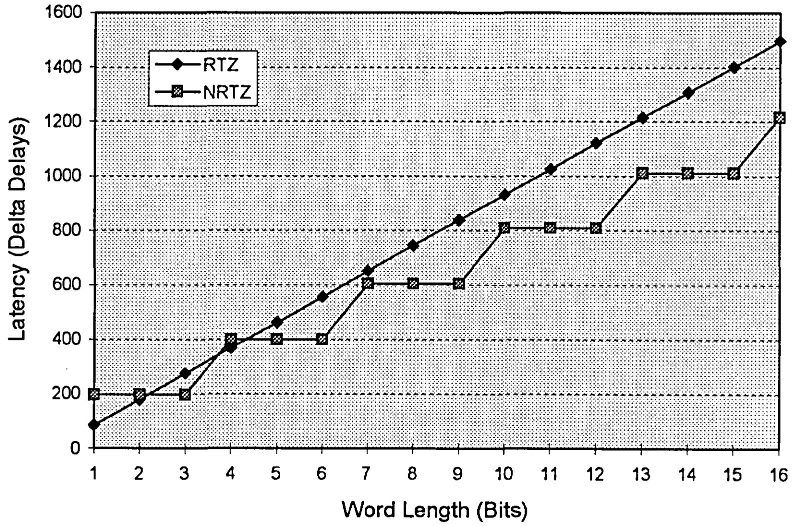


Figure 6.24. The RTZ and NRTZ OWU latencies.

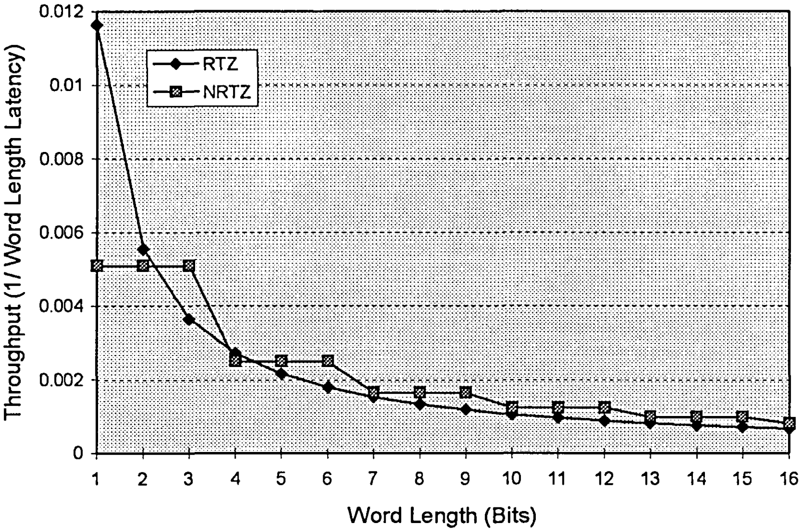


Figure 6.25. The RTZ and NRTZ OWU throughputs.

	RTZ	NRTZ
OWU	578 gates	707 gates

Table 6.5. The RTZ and NRTZ OWU gate counts.

Figure 6.24 shows that both the RCU RTZ and NRTZ implementations are comparable over short and mid-range word lengths. However, as the word size approaches the maximum length the NRTZ OWU performance is better than the RTZ OWU.

Figure 6.25 shows that both the NRTZ and RTZ OWU throughputs are comparable for short length words, however, for long words the NRTZ OWU throughput is greater.

Table 6.5 shows that the NRTZ is approximately 22% larger the RTZ.

6.8.5 Analysis

The reasons for the better performance and higher circuit overhead of the NRTZ OWU compared with the RTZ OWU are the same as those previously explained for the RCU function.

6.9 The Processing Element Status Unit (PSU)

The PSU has two main functions. The first is to signal the detection of the EOW for both source operands to enable the IDU to cease activating the selected RBU register(s) when more data is to be processed in the current instruction cycle. The second function is to signal the end of an instruction cycle. These two functions are explained in more detail in the following section.

6.9.1 The Self-timing Requirements

The first function of the PSU is necessary because if one source operand is shorter in length than the other then the IDU must cease sending control signals to the register that supplied the shorter operand. This is because improper circuit operation will arise as one data path will be expecting more data when none is available for that instruction cycle. This is also the case if a MIN or MAX instruction is executed in that the source operand register control signals must be disabled before the result can be written into the destination register. If the source registers continue to receive activation signals after both have output their data otherwise then incorrect circuit operation will occur.

The second main function of the PSU is to monitor the EOW Tag for each of the OWD, RESD, operand A and operand B status signals so it can signal the start of the next instruction cycle.

6.9.2 The RTZ Experiment

The RTZ Requirements

All pseudo operand A or operand B status bits produced, after the result EOW Tag has been detected, must be set to logic '1' to ensure that the PSU continues to signal the EOW to the IDU until the end of the instruction cycle.

All data path components are implemented in NRTZ circuitry to cater for the NRTZ EOW tagged data. The control is implemented in NRTZ circuitry.

The RTZ PSU Architecture

The RTZ PSU architecture is shown in figure 6.26.

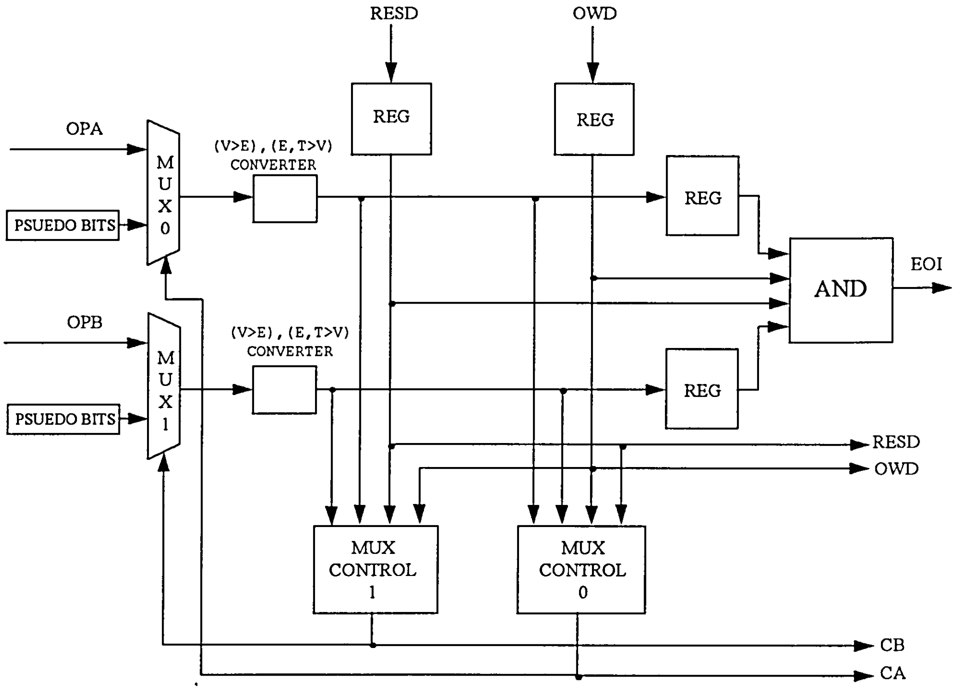


Figure 6.26. The RTZ PSU.

MUX (0, 1): The function of MUX is to pass to the receiving modules either an actual bit, indicating the status of the operand bit, or a pseudo bit, to ensure the receiving functions do not deadlock. It operates as follows. Each MUX has two data inputs, a source operand data input from the RBU and a pseudo data input, which are under the control of the selector input, which is fed by MUX CONTROL. If the control input is set to logic '0' then the source operand data input is passed to the mux output, otherwise if the control input is set to logic '1' then a pseudo bit is passed to the mux output. The pseudo input is fed by a FIFO ring buffer.

MUX CONTROL: The role of this device is to control the associated MUX through monitoring the RBU's source operand A and B output bits and the RESD and OWD status signals, so that the appropriate output control signal can be supplied to the controlled MUX(0, 1). The two monitored multiplexors are termed the primary multiplexor (the controlled device) and the secondary multiplexor, which is simply monitored with respect to the multiplexor control. The controller behaves as follows.

When the secondary input (SI) is a tag and the primary input (PI) is an empty bit (i.e. the secondary word is shorter than the primary word), or both the SI and PI are tag bits, and both RESD and OWD are tag bits, indicating the end of the instruction cycle, then MUX CONTROL selects the PI on the next bit cycle. If the PI and SI are both tags and OWD and RESD are not both tags then the pseudo input is selected.

When the PI is set to a EOW Tag and the SI is an empty value (i.e. the primary word is shorter than the secondary word) then the multiplexor's pseudo input is selected in order to attach a dummy bit to the end of the word. The pseudo input will continue to be selected on each bit cycle until the SI and the RESD and OWD signals are EOW Tags, indicating that the current instruction cycle has finished.

The outputs from each multiplexor controller, CA for operand A and CB for operand B, are also fed to the IDU to accordingly disable the data path for operand A or operand B.

AND: This circuit signals when both the operand, OWD and RESD EOW Tag bits have been detected thus signalling the end of the instruction (EOI) cycle. If all the inputs into the 'AND' are set to logic '1' then the output is set to logic '1' signalling the end of the instruction cycle. Otherwise, a logic '0' is output indicating that there is more data to be processed for the current instruction cycle.

6.9.3 The NRTZ Experiment

The NRTZ Requirements

Every pseudo status bit produced for operand A and operand B, after the result EOW for either been detected, must be set to logic '1' to ensure that the PSU continues to signal the EOW to the IDU.

The NRTZ PSU Architecture

The NRTZ architecture, shown in figure 6.27, is similar to the RTZ architecture except for the need to differentiate between EOW Tag bits and EVEN phase data bits. Therefore, only needed changes to be made to the MUX CONTROL, explained as follows:

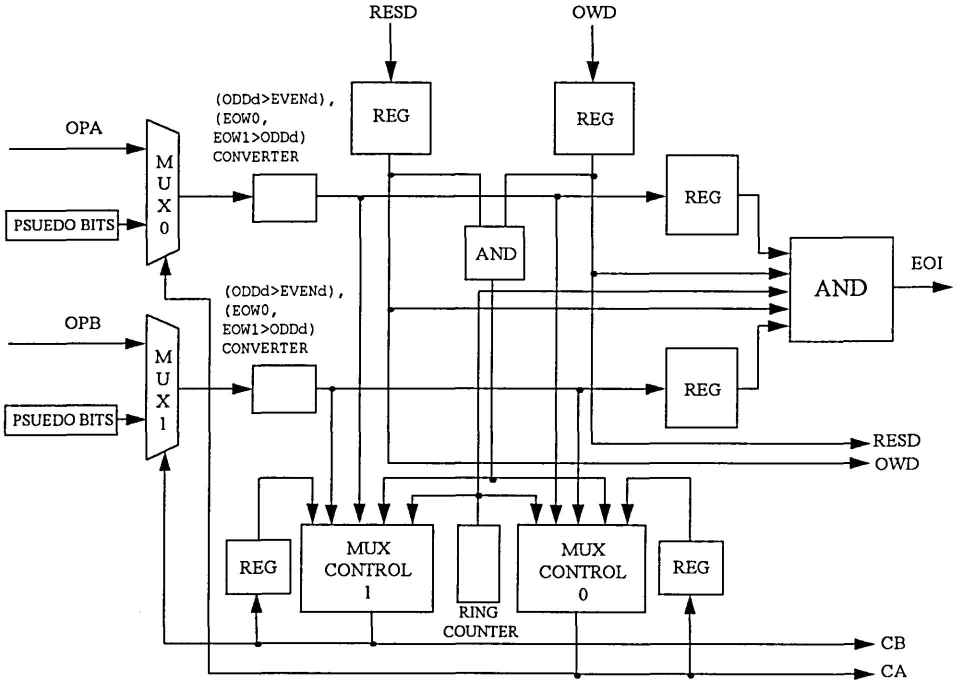


Figure 6.27. The NRTZ PSU.

MUX CONTROL: This circuit operates in a similar manner to the previously explained MUX CONTROL, except it differentiates between an EOW1 or an EOW0 and an EVENd via monitoring the RING COUNTER output. Feedback from the previous output state is used to indicate whether the actual data input or the pseudo data input was selected in the last cycle. The operation of this circuit is explained as follows.

In the event of an EOW1 on RING COUNTER input (RCI), and a logic ‘0’ EVENd on the feedback input (FI), MUX CONTROL selects the actual input of the multiplexor if PI is an EOW0 and SI, RESD and OWD are either an EOW1 or EOW0, or if PI, SI, OWD and RESD are EOW1. If PI is an EOW1, independent of the logic value of FI, RESD and OWD, and SI is an EOW0, then the pseudo input is selected in order to append dummy bits to the end of the word. If PI and SI are EOW1, and OWD and RESD are EOW0, then the pseudo input is selected. Otherwise, if PI and SI are EOW1, and OWD and RESD are EOW1, then the next actual input is selected, thus indicating the end of the current instruction cycle.

In the event of an ODDd or EVENd on the RCI, MUX CONTROL selects the actual input of the multiplexor if FI is set to logic ’0’, or the pseudo input if FI is set to logic

'1'. Therefore, the intermediate bits, between EOW Tag bits, of either the pseudo word or actual word are selected.

6.9.4 Results

Figure 6.28, figure 6.29 and **table 6.6** show the latency, throughput and gate overhead results, respectively, for both the RTZ and NRTZ PSU functions.

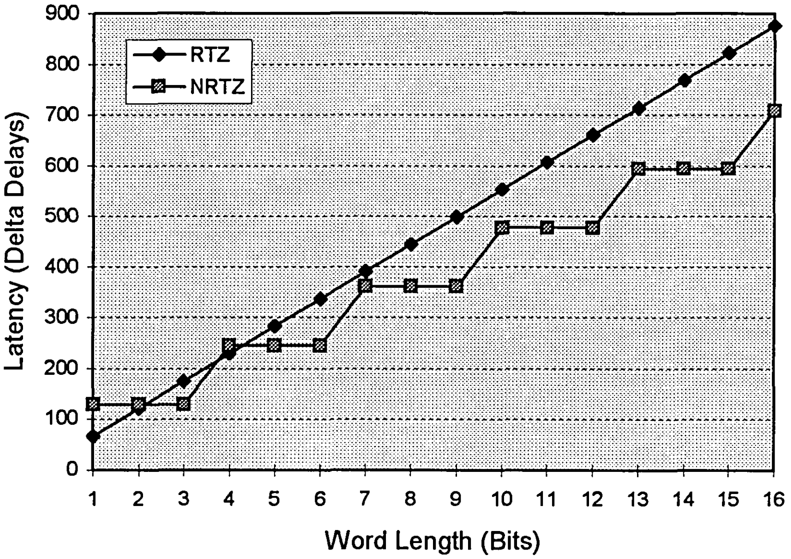


Figure 6.28. The RTZ and NRTZ PSU latencies.

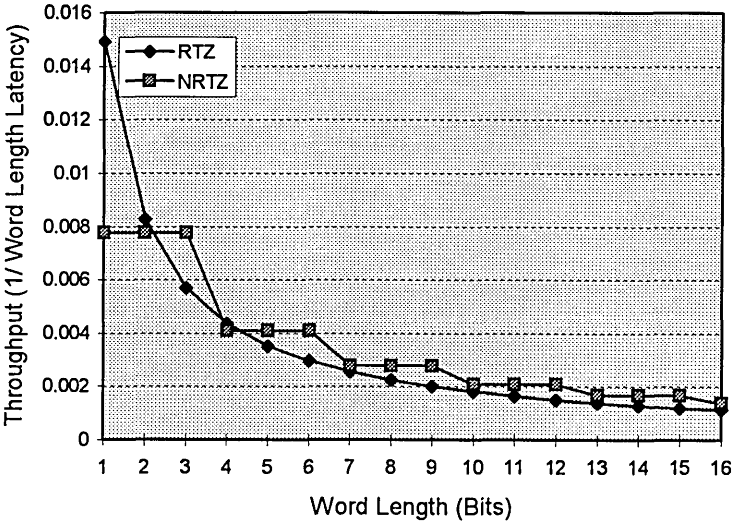


Figure 6.29. The RTZ and NRTZ PSU throughputs.

	RTZ	NRTZ
PSU	612 gates	1075 gates

Table 6.6. The RTZ and NRTZ PSU gate counts.

Figure 6.28 shows that the NRTZ and RTZ latencies are comparable over the short and mid-range length words, however, the NRTZ PSU is slightly faster as the word length approaches the maximum.

Figure 6.29 shows that the RTZ and NRTZ PSU throughputs are comparable for short length words, however, the NRTZ PSU throughput is greater for medium to long length words.

Table 6.6 shows that the NRTZ PSU is approximately 76% bigger than the RTZ PSU.

6.9.5 Analysis

The reasons for the higher performance of the NRTZ PSU compared with the RTZ PSU is due to the same reason as the RCU and OWU functions. In addition, the circuit overhead of NRTZ PSU is significantly larger than the RTZ PSU due to the extra circuitry required by the ring buffer, the two MUX CONTROL circuits, and the extra AND function.

6.10 The Self-timing Design Guide-lines

This section summarises the design guide-lines for the implementation of self-timed control path structures: instruction bypass (IBU), instruction fetch (IFU), instruction decode (IDU), and PE bit and instruction cycle control functions (RCU, OWU, PSU). These guide-lines were determined during the previous experiments in this chapter for each of control path functions and can be sub-divided into guide-lines for general self-timed implementation, independent of the encoding technique, and RTZ and NRTZ encoding and tagging guide-lines.

Design Guide-line Assumptions

The RTZ and NRTZ control path functions share two main design commonalities previously explained in sections 5.4.2 and 5.4.3, respectively. These two guide-lines are stated below and are common to each of the RTZ and NRTZ control path functions addressed in this section:

1. The data path must be realised using NRTZ circuitry.
2. All local control is be realised using RTZ circuitry.

In a few cases a modification to these guide-lines will be necessary and will be highlighted, in the appropriate sections, in *italic type*.

Instruction By-pass

General Self-timing Guide-lines

1. The single bit row and column selectors, that determine whether a bit-serial instruction is input into a PE, must coincide with the first bit of the bit-serial instruction.
2. As a consequence of the previous guide-line and the fundamental mode of operation the IBU would dead-lock because the succeeding instruction bits would not be input with selector bits. Therefore, selector bits must be internally re-generated in order to appropriately select/deselect the remaining instruction bits.

The RTZ Guide-lines

3. The EOW Tag bit, in the instruction stream, indicates that the next row and column selector bits and instruction can be input.

The NRTZ Guide-lines

3. The NRTZ instruction stream is untagged and fixed in length, and consequently each instruction bit is counted as it is input. When a maximum count is reached, the next row selector bit, column selector bit, and instruction words, can be input.

Instruction Fetch

General Self-timing Guide-lines

1. The bit-serial instruction is converted into the equivalent bit-parallel representation and transmitted to the decoder on each PE bit cycle.
2. The bit-parallel instruction must be internally stored and re-generated on each bit cycle.

RTZ Guide-lines

1. Each valid instruction bit and its corresponding succeeding empty bit must be passed to the instruction decoder over a separate dedicated line.
2. The instruction EOW Tag bit, succeeding the last valid bit of the instruction, must be converted to an empty bit before it sent to the instruction decoder.
3. All data path circuitry preceding and including the conversion of the EOW Tag to an empty bit must be realised using NRTZ circuitry.
4. *All data path circuitry succeeding the conversion of the EOW Tag to an empty bit is realised using RTZ circuitry.*

NRTZ Guide-lines

1. As a consequence of the control circuitry of the NRTZ PE being completely implemented in RTZ circuitry the NRTZ bit-serial instruction must be converted to the equivalent RTZ bit-parallel representation. This necessitates the following requirements:
    a) Each EVEN bit in the bit-parallel instruction must be converted to the equivalent ODD phase bit, or RTZ valid bit, to be in the correct format for the instruction decoder.
    b) An empty bit must be inserted between valid instruction bits in each parallel line to realise the RTZ encoding.
2. *All data path circuitry preceding the operation described above in guide-line 1 must be realised using NRTZ circuitry.*
3. *All data path circuitry succeeding the operation described in guide-line 1 must be realised using RTZ circuitry.*

Instruction Decode

General Self-timing Guide-lines

1. The IDU must provide control information to each of its controlled functions indicating whether they are enabled or disabled in each bit cycle; this is necessary because of the fundamental mode of operation.
2. The status (whether the EOW Tag has been detected or not) of the two source operand data paths from the RBU (sent by the PSU), and the result from the EXU, must be monitored in each bit cycle in order that the IDU can correctly disable the appropriate parts of the data path.

RTZ Guide-lines

1. *The RTZ decoder is realised completely in RTZ circuitry.*

NRTZ Guide-lines

1. *The NRTZ decoder is exactly the same as the RTZ decoder.*

The Result Control Unit

General Self-timing Guide-lines

1. The RCU must transmit a status bit, for each result bit that is passed between the EXU and RBU, to differentiate between the EOW Tag (data-true), empty bit (data-false) and valid data bit (empty).
2. The RCU must continue sending status bits in the event the result EOW Tag is detected and the destination register's old data has not been overwritten; therefore the RCU must monitor the OWU status bit.
3. If a MIN or MAX instruction is executed the RCU must continue to output status bits, set to indicate that the EOW Tag has not been detected for each bit cycle, until the first result data bit is output from the EXU, after which guide-lines 1 and 2 above, apply.

The OverWrite Unit

General Self-timing Guide-lines

1. The OWU removes old data from the destination register by supplying an acknowledge signal every time a data bit is read.

2. A status bit must be sent to the PSU, for each old data bit overwritten, in order to differentiate between the EOW Tag bit, empty bit and data bit.
3. If the destination register also acts as a source then overwrite acknowledge signals must be synchronised with the acknowledge signals of the appropriate RBU multiplexor(s).
4. The OWU must continue sending status signals to the PSU after it has detected the EOW of the overwritten data if the result status bit is not set to signify the EOW Tag.

The Processing Element Status Unit

General Self-timing Guide-lines

1. The PSU monitors each bit in both bit-serial operand streams as each are passed from the RBU and EXU. It transmits a corresponding status bit for both operand bits as they are monitored, to the IDU, differentiating between the EOW Tag, empty bit and valid data bit.
2. The PSU must monitor for the EOW Tag bit for each of the OWD, RESD, operand A and operand B status signals so it can signal the start of the next instruction cycle or regenerate the present instruction to the IFU.

6.11 Selection Guide-lines

This section proposes guide-lines to aid computer architects in the selection of the appropriate RTZ, NRTZ or synchronous design technique, depending on the required SA control path implementation characteristics. These guide-lines are directly related to the selected data path implementation technique, for example, if the selected data path implementation technique is NRTZ then the control path functions must also be NRTZ and not RTZ functions. The following performance and circuit area guide-lines are based, respectively, on the latency and throughput, and gate count characteristics produced for the RTZ, NRTZ and synchronous data path functions.

Performance Guide-lines

The performances of the IFU and IDU control path functions are dependent on the length of the data word operated upon by the data path, whereas the IBU's performance is fixed to a worst case speed dependent on the pre-determined instruction word length.

The performance of all the synchronous data path functions is fixed to a static worst case speed determined by the system clock.

The following guide-lines propose the appropriate technique to adopt to maximise a control path function's performance when considering the number of control bit cycles produced for each of the functions:

- The RTZ and NRTZ IFU and IBU are slower than the fixed speed synchronous equivalents for all word lengths, and the IDU is slower for medium to long word lengths. Therefore, considering the performance of the RTZ and NRTZ control path functions independently, from their respective data path functions they control, means that the synchronous technique is the best to adopt.

The performance of the self-timed IFU and IDU functions is directly related to the number of bits processed by, and the performance of, the data path functions. Also, the performance of the synchronous control path functions given in this chapter are assumed to be best case processing speeds and not the fixed worst case PE speed as dictated by the system clock. As a consequence, final decisions on the selected implementation technique for a control path function for a required performance cannot made until the performances of the completed RTZ, NRTZ and synchronous PE are determined; presented in the next chapter.

Circuit Area Guide-lines

The gate counts of the RTZ and NRTZ IFU and IDU functions are considerable larger than their synchronous equivalents, whereas the IBU are approximately the same. In addition, the RTZ and NRTZ control paths require three extra functions due to the self-timed implementation, thus, adding considerably to the overall gate count of the control path. Therefore, the synchronous design technique is the best technique to adopt because of the large size of the RTZ and NRTZ control path functions.

6.12 Conclusion

This chapter has presented two new self-timed methods for realising control circuitry for the ST-SISA. These methods have been characterised in terms of circuit performance and area and have been compared with each other and their synchronous counterpart. This comparison has showed us that the RTZ and NRTZ circuits are architecturally scaleable but at the cost of considerably larger circuit area and reduced performance as compared with the synchronous counterparts.

This chapter has provided guide-lines to aid computer architects in the design of RTZ and NRTZ control path functions for use in bit-serial architectures. These guide-lines support the designer by proposing considerations to take into account when applying delay-insensitive assumptions, and RTZ and NRTZ encoding and tagging techniques to control path functions for use in bit-serial architectures.

CHAPTER 7

THE PROCESSING ELEMENT

7.1 Objectives of the Chapter

The first main aim of this chapter is to demonstrate that the new RTZ and NRTZ data path and control path functions, presented in the two previous chapters, operate correctly when connected together to form the complete RTZ and NRTZ ST-SISA processing element (PE) architectures. This aim will be achieved through the test and analysis of the RTZ and NRTZ PE architectures when operating on each of the pre-defined instructions for differing data bandwidths. The second main aim is to demonstrate that both instruction and data inter-PE communication operates correctly in the ST-SISA array. This aim is satisfied through the test and analysis of a number of representative programs executed by the ST-SISA model. The successful completion of both these objectives will enable the relative circuit area and performance trade-offs between the RTZ and NRTZ ST-SISA and the synchronous SISA PEs to be established.

7.2 Objectives of the Experiments

Two main experiments have been proposed, stated in the following two sections.

7.2.1 The PE Experiment

The aim of this experiment is to demonstrate that the RTZ and NRTZ ST-SISA PEs operate correctly. It also compares the circuit area and performance trade-offs between the RTZ and NRTZ ST-SISA PEs and the existing synchronous SISA PE. The metrics used are as follows:

1. **Latency Experiment (1):** Time between the start of an instruction operation and it's subsequent completion.
2. **Throughput Experiment (2):** The total amount of work completed in one delta delay (1Δ/latency).
3. **Hardware Overhead Experiment (3):** The number of PE gate primitives.

7.2.2 The Communication Experiment

The aim of this experiment is to demonstrate that ST -SISA instruction and data inter-PE array communication operates correctly. This experiment also compares the effects of instruction and data communication on PE instruction latency for the RTZ and NRTZ ST-SISA arrays and SISA array. The metric used is stated as follows:

1. **Latency Experiment**: Time taken by each array PE from the start of an communication instruction execution and it's subsequent completion.

7.3 The PE Experiment

The ST-SISA PE architecture, common to both the RTZ and NRTZ implementations, is shown below in figure 7.1. It illustrates how the functional units of the control and data paths, presented and discussed in the previous chapters, are connected together to form the finished ST-SISA PE architecture. The role of each of the data and control path functions has been described previously, in chapters 5 and 6, respectively.

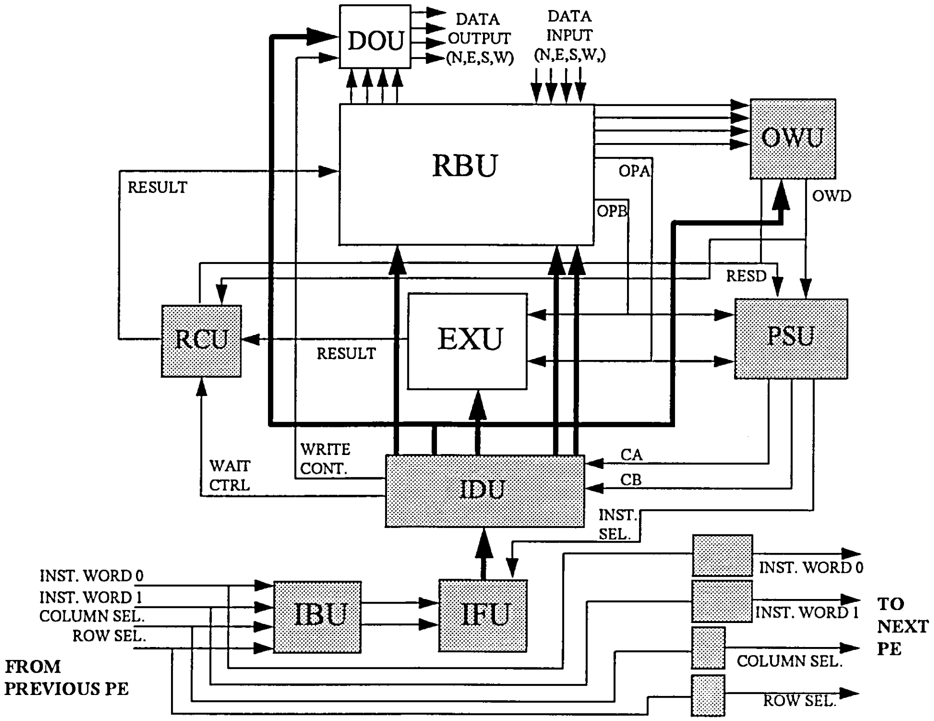


Figure 7.1. The complete ST-SISA PE.

The INST. SEL. line signals the status of the PE after the previous bit cycle, in that a logic '1' signals the end of the instruction cycle, whereas a logic '0' signals that there is at least one more bit cycle in the instruction cycle to take place. Therefore, measurements can be taken from the time when all the instruction information is placed on the PE instruction inputs until the time the INST. SEL. line is set to logic '1'.

7.3.1 Latency Experiment (1)

Introduction

The latency of each of the ST-SISA instructions is defined as the time taken from the start to the completion of an instruction cycle. This measurement is taken as the greatest time interval between the receipt of a new instruction, on the PE instruction input lines, and the eventual signalling of the end of the instruction cycle, on the INST. SEL. line. This definition assumes that an instruction cycle can vary depending on the length of the longest data operand, therefore, measurements are taken for all possible word lengths for each instruction; this definition also assumes that RTZ empty values and NRTZ unused EOW Tag bits are included in these measurements.

Experiment

The experimental method adopted was to determine the time taken to process an instruction for the full range of data word lengths for each of the RTZ, NRTZ and synchronous PE implementations.

7.3.1.1 Latency Results

The results presented in **Figures 7.2** through to **7.6** represent the trends that occur in the latency for the READ/WRITE/ASSIGN, AND, ADD, MIN and MAX instructions, respectively, for the RTZ, NRTZ and synchronous PEs.

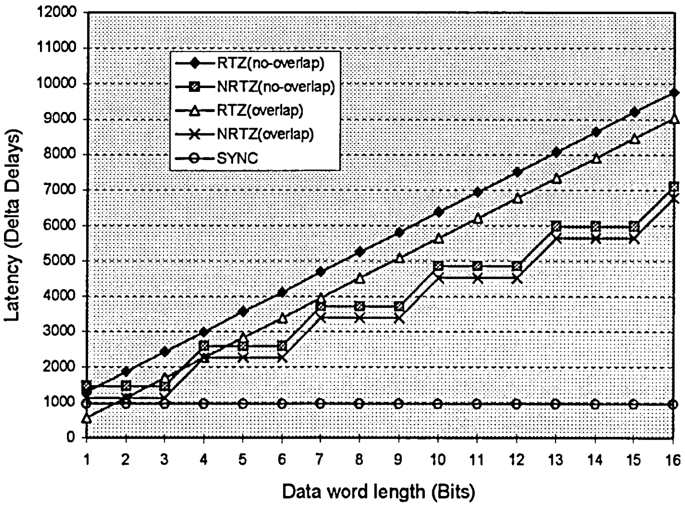


Figure 7.2. READ, WRITE and ASSIGN.

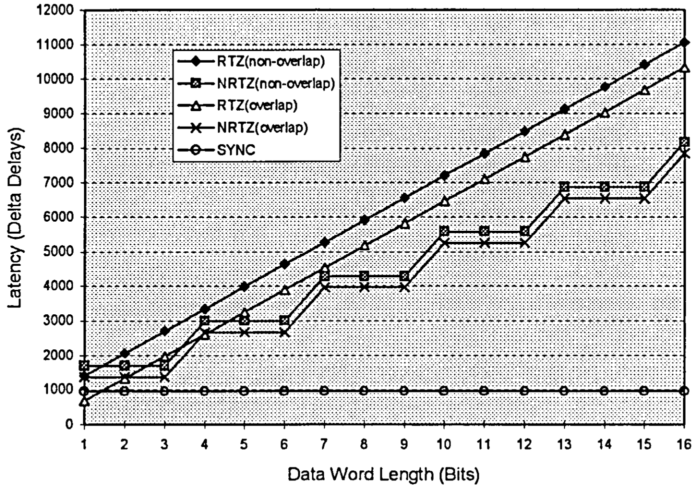


Figure 7.3. AND.

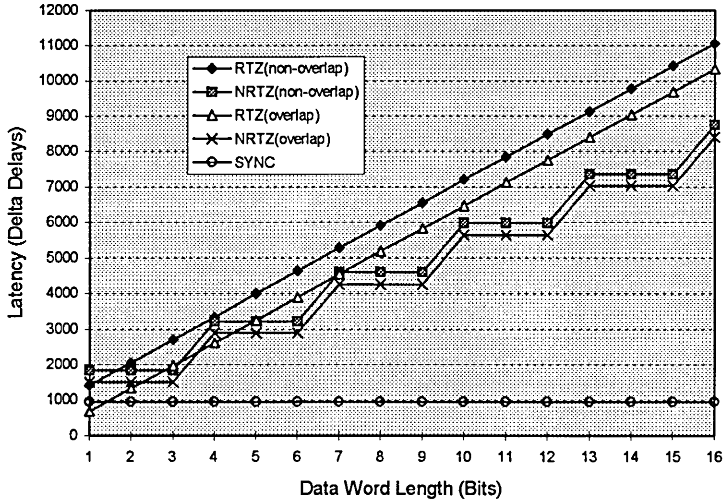


Figure 7.4. ADD.

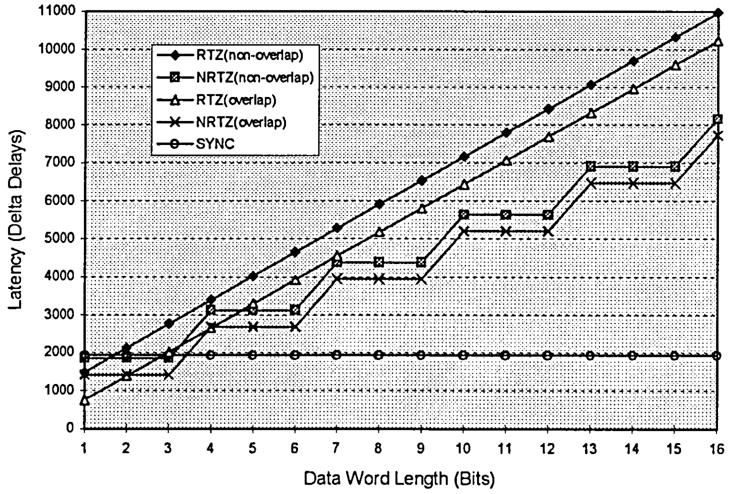


Figure 7.5. MAX Comparator.

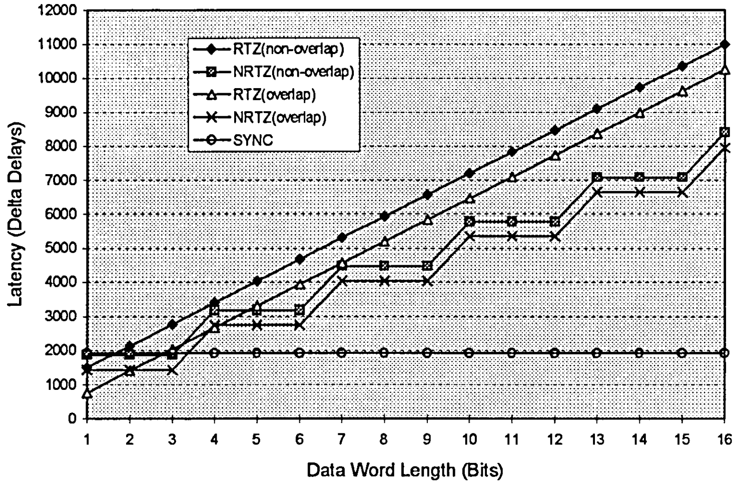


Figure 7.6. MIN Comparator.

The following trends can be observed from **Figures 7.2** through to **7.6:**

- The latency of overlapping instructions is less than non-overlapping instructions.
- The latency of each instruction increases as the word length of the longest operand is increased.
- Almost all synchronous instructions produce shorter latencies than the self-timed equivalents, except for a 1-bit RTZ word length for the READ, AND and ADD overlapped instructions. In addition, 1 and 2 bit word lengths for RTZ designs, and words lengths up to 3-bits for NRTZ designs, are faster than the synchronous design for the MIN and MAX overlapping instructions.
- The latency of each NRTZ instruction is less than the equivalent RTZ PE instruction.
- Each instruction executed on the same ST-SISA PE implementation (i.e. RTZ), assuming the same word size, takes approximately the same time.

7.3.1.2 Latency Analysis

These results show several significant features. Firstly, the overlapping of instructions allows an instruction to be converted from serial to its corresponding parallel equivalent while the preceding instruction is still being executed, therefore reducing the total latency of the succeeding instruction. However, if there are idle times between the PE executing successive instructions then the non-overlapping, or worst case, latency must be assumed for each instruction.

Secondly, the greater the length of the longest operand then the greater the number of bit-cycles required to complete the instruction cycle therefore increasing instruction latency.

Thirdly, the NRTZ instruction latencies are less than the equivalent RTZ instruction latencies because of the fewer number of bits required by the NRTZ encoding technique compared with the RTZ technique to encode the same data value.

Finally, the latency of each self-timed instruction, assuming they operate on the same word length, are approximately the same. This is because the next cycle of control information for the RBU and EXU can only be output by the IFU when it receives the instruction select status signal from the PSU. The PSU can only send this control information signal after it receives the result EOW Tag status bit from the RCU. This is because the RBU and EXU operate in serial.

7.3.2 Throughput Experiment (2)

Introduction

The aim of this experiment is to establish the trends that occur in the throughput of each of the self-timed and synchronous PE implementations when operating on the pre-defined instruction set. These trends will reveal the PE performance for the three adopted design methods.

The throughput is defined as the total amount of work done in a given time. The rate at which the synchronous PE performs is always constant because of the fixed clocking rate, however, the performance of the two self-timed PE designs depends on the length of data operated upon for each instruction and the number and variance of the different instructions executed. These two factors concerned with the self-timed designs are directly related to the algorithm executed and the data operated upon. As a consequence there are no strict rules that can be adopted to determine the exact throughput of the RTZ and NRTZ PEs when working in an array of PEs. Therefore, in this experiment it is assumed that the environment always has the correct data and instructions ready to supply to a PE's inputs for both the RTZ and NRTZ PE designs. From this the worst, best and average case throughputs for a single PE can be determined.

Experiment

The throughput of each PE was obtained by measuring the time taken for each PE design to process each instruction (i.e. latency) and then taking the reciprocal of these measurements. Using these results it is possible to determine the following:

1. The best (i.e. overlapping instructions), worst (i.e. non-overlapping instructions) and average case (i.e. for both non-overlapping and overlapping instructions) throughputs of each PE for each individual word length.
2. The best, worst and average throughputs for each PE averaged over the entire word length range. In addition, the absolute best throughput (i.e. the repeated execution of the fastest instruction operating on the shortest possible word lengths) and the worst case throughput (i.e. the repeated execution of the slowest instruction operating on the longest possible word lengths) are established.

7.3.2.1 Throughput Results

Figure 7.7 depicts the best, worst and average case throughputs for each of the RTZ, NRTZ and synchronous PEs for all possible word lengths. **Figure 7.8** illustrates the best, worst and average case throughputs averaged over the entire word length range for each PE. This figure also shows the absolute (ABS) best throughput and ABS worst throughput of each of the RTZ, NRTZ and synchronous PEs.

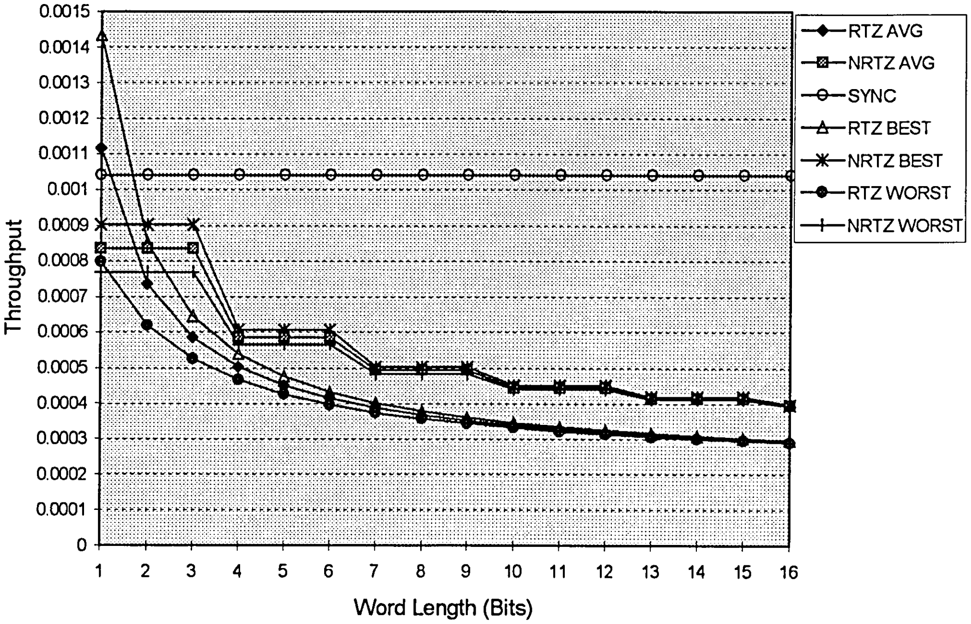


Figure 7.7. The average, best and worst case throughputs.

The following trends can be derived from **Figure 7.7**.

- The throughput of the self-timed PEs decreases as the word length increases.
- The throughput of the synchronous PE is constant.
- The best and average case throughputs of the RTZ PE for a 1 bit word length are greater than the fixed constant throughput of the synchronous PE by approximately 43% and 5%, respectively.
- The throughput of the synchronous PE is greater than the throughput of the NRTZ PE for all words lengths.
- The throughput of the synchronous PE is greater than the best throughput of the RTZ PE, by 30% to 250%, and the best throughput of the NRTZ PE, by 17% to 163%, as the word length is increased from 2- to 16-bits.
- The average, best and worst case throughputs of the RTZ and NRTZ PEs converge to the same value as the word size is increased.

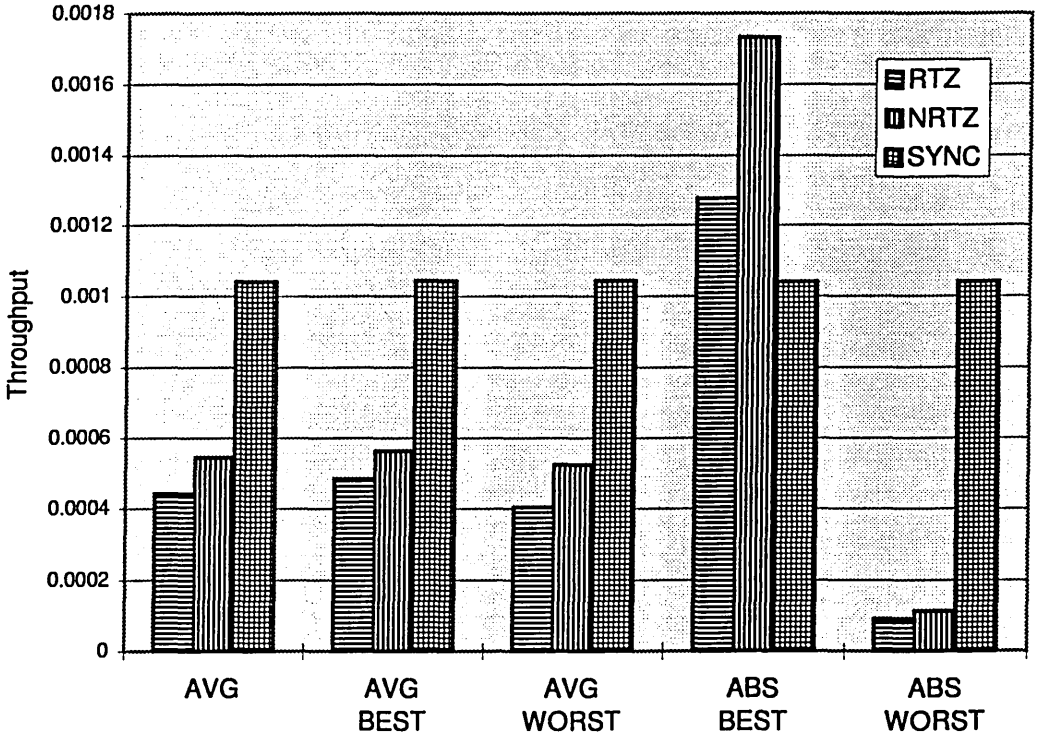


Figure 7.8. The average, average best and average worst case throughputs over the entire word range and the absolute best and worst throughputs.

The following trends can be distinguished from **Figure 7.8**.

- The average, best and worst case throughputs of the RTZ and NRTZ PEs, respectively, averaged over the entire word range are approximately the same.
- The throughput of the synchronous PE is approximately two times greater than the average, average best and average worst case throughputs of the RTZ and NRTZ PEs over the entire word range.
- The absolute best throughputs of the RTZ and NRTZ PEs are approximately 23% and 62% greater, respectively, than the throughput of the synchronous PE.
- The throughput of the synchronous PE is approximately ten times greater than both the absolute worst throughputs of the RTZ and NRTZ PEs.

7.3.2.2 Throughput Analysis

These results show several important features. Firstly, the length of the data word operated upon in each bit cycle has a direct effect on the throughput of the RTZ and NRTZ PEs. For 1-bit data words the RTZ and NRTZ perform slightly better than the constant synchronous throughput. However, as the word length is increased the throughputs of both the self-timed PEs drops dramatically as the word size is increased. The reasons for this are as a direct result of the sequential operation of the data path, previously explained in section 7.3.1.2. The sequential operation of the data path also has a detrimental effect on the average throughput of the all the instructions over the entire word length range.

Finally, the absolute best throughputs of the RTZ and NRTZ PEs are significantly greater than the synchronous PE's fixed constant throughput. This can only occur when a high degree of NOPs are executed. Therefore, this demonstrates the benefit of the variable length processing speed of the RTZ and NRTZ instructions as compared to the synchronous instructions, which all take the same fixed worst case processing speed. However, the absolute worst case throughputs of the RTZ and NRTZ PEs, in other words, the repeated execution of the worst case instruction assuming the longest possible word length, is far worse than the fixed synchronous throughput. This again can be attributed to the relative processing speed of the self-timed data path.

7.3.3 Circuit Overhead Experiment (3)

Introduction

This experiment obtains the trends that occur in the circuit area required to realise the RTZ, NRTZ and synchronous PEs. In addition, the proportion of circuit area required to implement the control and data paths has been established revealing their relative costs.

Experimental Methodology

The circuit area overhead for the data path and control paths has been compiled with reference to chapters 5 and 6, respectively.

7.3.3.1 Circuit Area Results

Figures 7.9, 7.10 and **7.11** show the total data path, control path and PE circuit area gate counts, respectively, for the RTZ, NRTZ and Synchronous implementations.

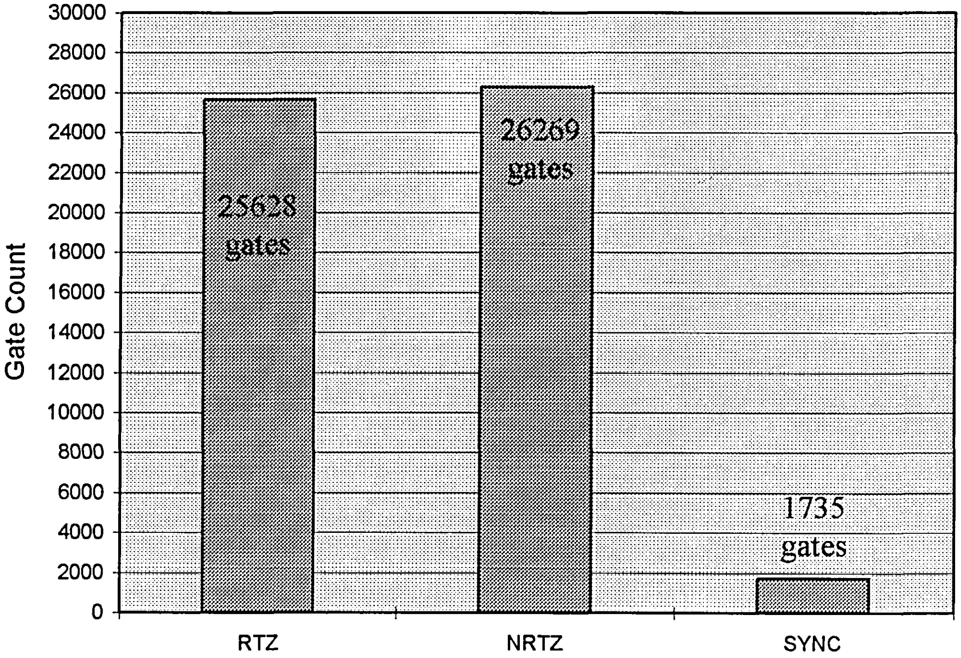


Figure 7.9. RTZ, NRTZ and synchronous data path circuit areas.

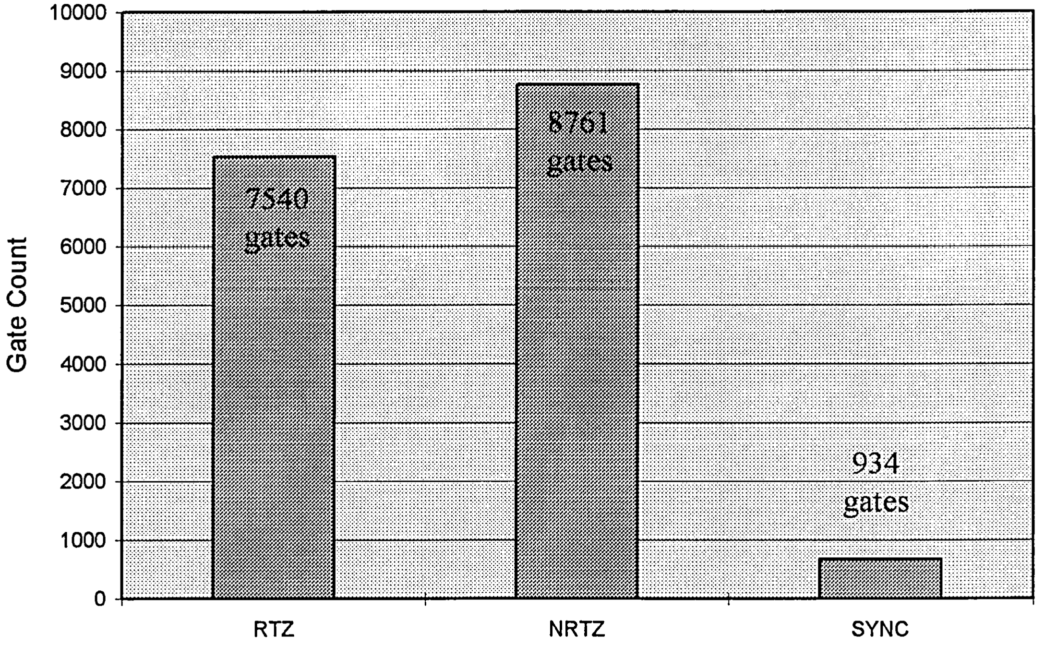


Figure 7.10. RTZ, NRTZ and synchronous control path gate counts.

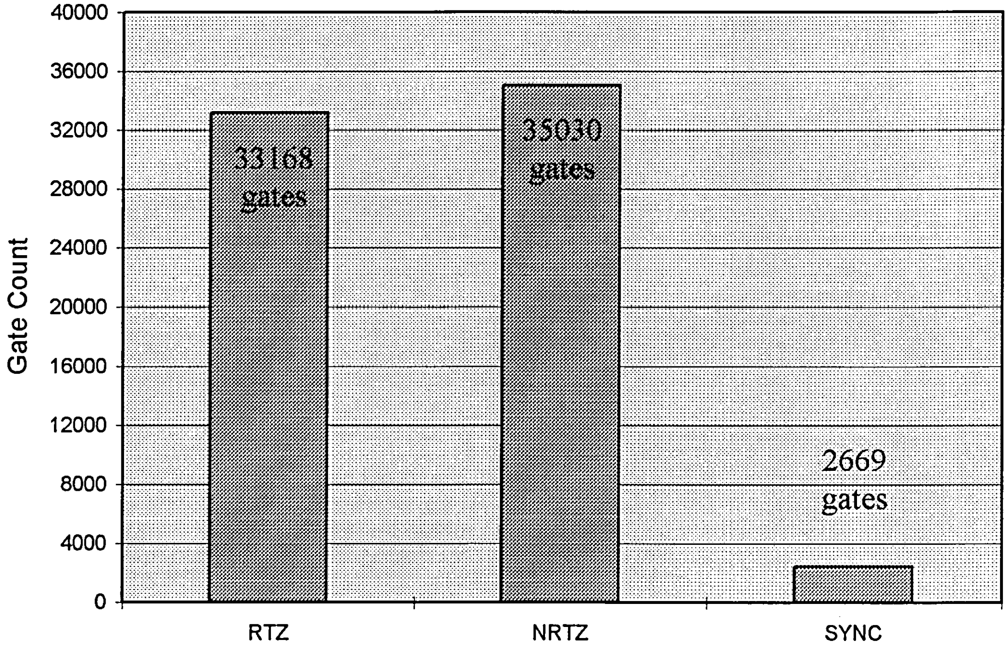


Figure 7.11. RTZ, NRTZ and synchronous PE gate counts.

Examination of **Figures 7.9, 7.10** and **7.11** reveals the following trends:

- The RTZ and NRTZ data paths are approximately fourteen and fifteen times bigger, respectively, than the synchronous data path.
- The NRTZ data path is approximately 6% bigger than the RTZ data path.
- The RTZ and NRTZ control paths are approximately eight and nine times bigger, respectively, than the synchronous control path.
- The NRTZ control path is approximately 16% larger than the RTZ control path.
- The RTZ and NRTZ processing elements are respectively twelve and thirteen times bigger than the synchronous processing element.
- The NRTZ PE is approximately 6% bigger than the RTZ PE.

7.3.3.2 Circuit Area Analysis

There are several reasons for the above observations. Firstly, the RTZ and NRTZ delay-insensitive circuit implementations are well known to consume a higher degree of circuit area than their equivalent synchronous counterparts thus resulting in their significant circuit overhead. In addition, the local control circuitry required by the various data path and control path functions also adds to this overhead.

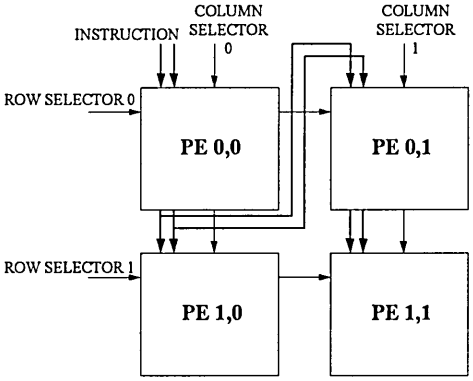
Secondly, the relative circuit area required by the RTZ and NRTZ control path functions is much greater than the RTZ and NRTZ data path functions when they are compared to their synchronous equivalents. This is as a result of the large amount of self-timed circuitry required to convert the bit-serial instruction into parallel and to regenerate instruction codes.

Finally, the NRTZ control path, data path and final PE are larger than their RTZ counterparts because of the extra circuitry required to implement the ring buffer counters and the extra inputs required by the various combinational functions that operate on the ring buffer outputs.

7.4 The Communication Experiment

The ST-SISA array has three uni-directional streams of control information, figure 7.12: instructions along the columns and selector bits along the columns and rows, respectively. Therefore, instruction flow is only possible in one direction starting from the top left hand PE (0,0) proceeding diagonally towards the bottom right PE (1,1).

Data communication in a ST-SISA array, figure 7.13, is possible in the North, South, East and West directions on the execution of a read instruction by the receiving PE and a write instruction by the transmitting PE. Therefore, data communication between any two neighbouring processing elements can only take place over the same dedicated lines connecting the two PEs.



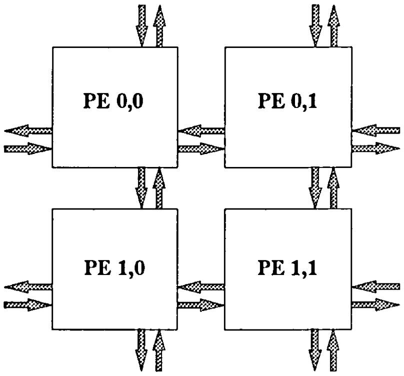


Figure 7.12. The ST-SISA control flow. **Figure 7.13. ST-SISA data flow.**

Testing both instruction and data communication in the ST-SISA array is straightforward. Firstly, any correct ST-SISA program will test control communication. Secondly, testing data communication between two PEs is the same as testing data communication between a PE and any one of it's four neighbours. Therefore, instruction and data communication can be fully tested through the execution of a program that exchanges data in both communicational directions between two neighbouring PEs, as shown in figure 7.14.

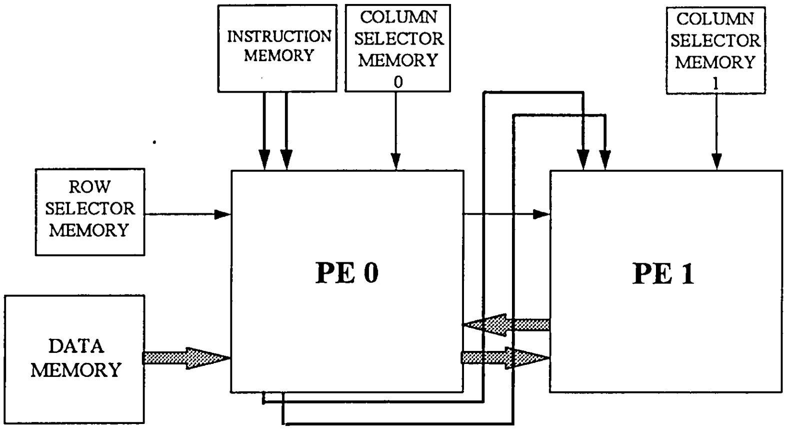
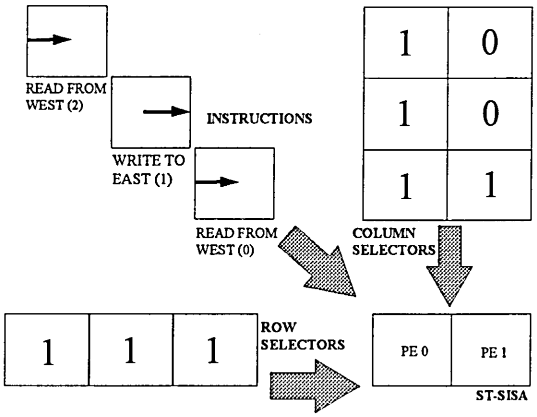


Figure 7.14. The ST-SISA test array.

Three programs can be used to test ST-SISA instruction and data communication. Uni-directional data communication can be tested through the execution of a simple program that assigns a data item, read from the westerly situated data memory, to each of the two PEs in the array, figure 7.15 (program 1 [54]); therefore, PE(1) must pass the first data item read on to PE(2). Bi-directional data communication can be tested through the execution of a program that then swaps these two data items between the two PEs, figure 7.16 (program 2 [53]). Bi-directional communication can also be tested through the execution of a data sort program which swaps the two PE data items only if the magnitude of PE 1's data is less than the magnitude of PE 0's data, figure 7.17 (program 3 [82]).



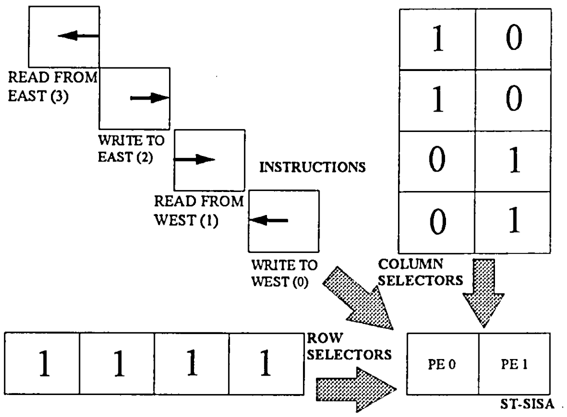


Figure 7.15. Program 1 - Assign. **Figure 7.16. Program 2 - Transposition.**

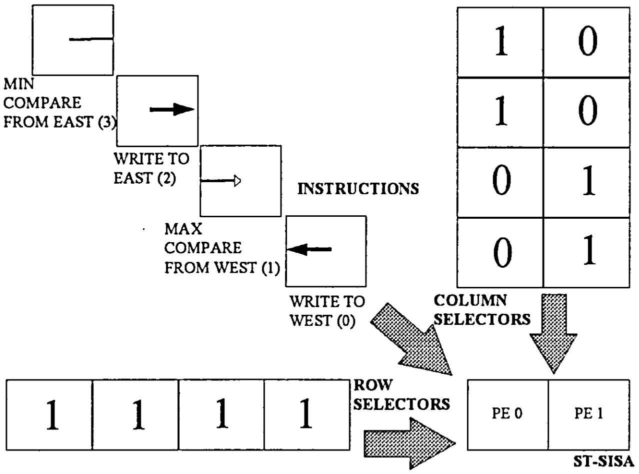


Figure 7.17. Program 3 - Sort.

7.4.1 Latency Experiment

Introduction

This experiment has two main objectives. The first aim is to ensure that both instruction and data communication work correctly for both the RTZ and NRTZ ST-SISAs. The second main objective is to observe the effects of instruction and data communication on the execution times of the RTZ and NRTZ ST-SISA PEs for each of the communication instructions in the three programs described earlier.

Experiment

The first objective is satisfied through the successful execution of the three above ST-SISA programs. The second objective is achieved through comparing the differences in PE instruction execution times (latency) of the single PE execution times detailed earlier (section 7.3.1.3) with the latencies of the instructions performed by each PE, when executed by the two PE array models, for each of the three programs. Therefore, the effects of RTZ and NRTZ self-timed inter-PE instruction and data communication on ST-SISA PE and array performance can be compared against each other and against their existing synchronous counterpart.

7.4.1.1 Latency Results

The results in **Figures 7.18** and **7.19** show the latency trends for the RTZ and NRTZ array simulations for the executed instructions in program 1.

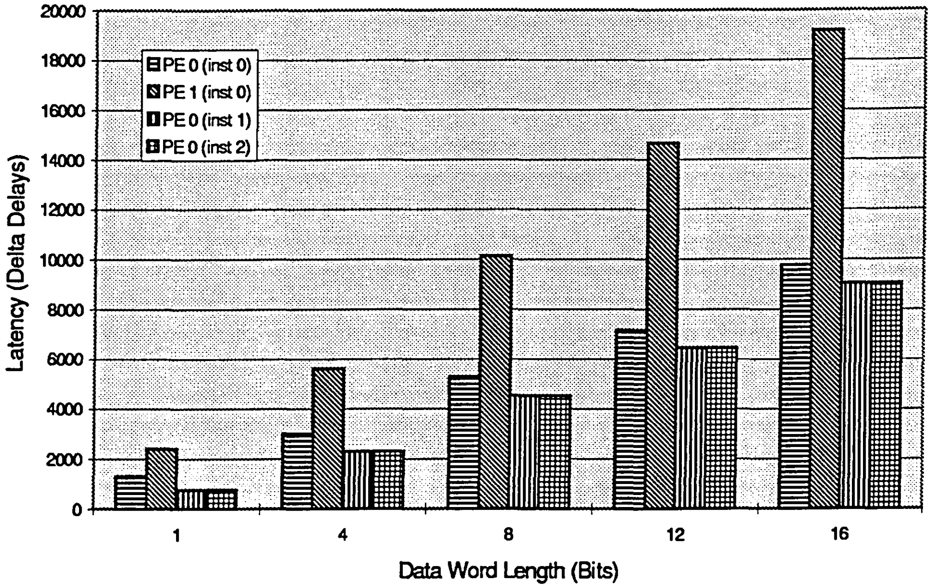


Figure 7.18. RTZ ST-SISA array results for program 1.

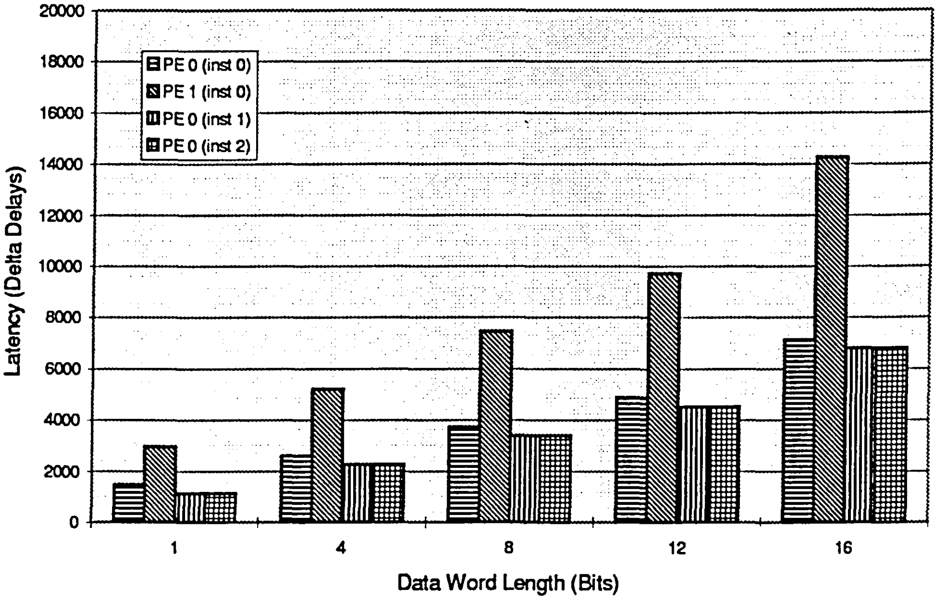


Figure 7.19. NRTZ ST-SISA array results for program 1.

The following trends can be observed from **Figures 7.18** and **7.19** for both the RTZ and NRTZ arrays, respectively, when executing program 1:

- The latency of the first read instruction (0) is greater than the time taken by the second read instruction (2) when both are executed by PE 0.

- The latency of the write instruction (1) is the same as the second read instruction (2) when executed by PE 0.
- The latency of the first read instruction (0) when executed by PE 1 is twice times as great as the latency of the same read instruction (0) when executed by PE 0.

The results in **Figures 7.20** and **7.21** show the latency trends for the RTZ and NRTZ array simulations, respectively, for the executed instructions in program 2. It is assumed that program 1 has been previously executed in order to assign data to the array.

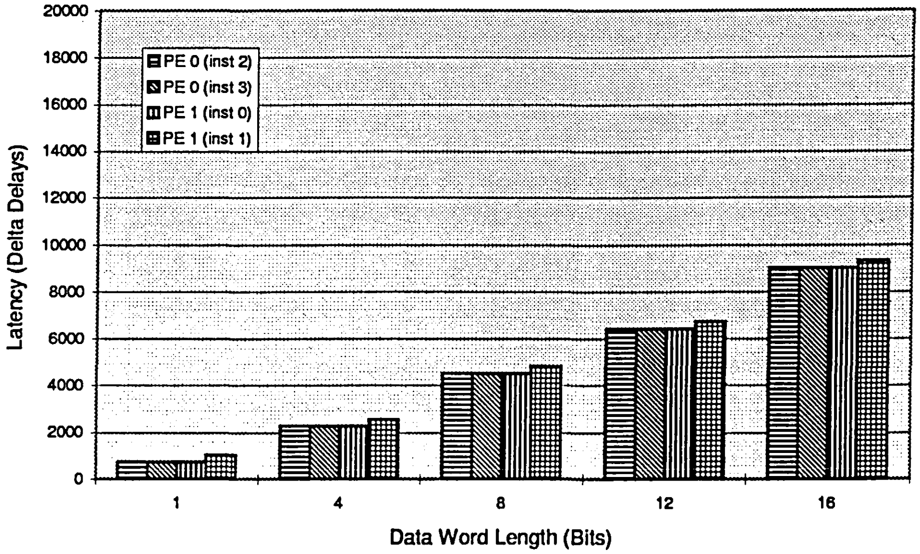


Figure 7.20. RTZ ST-SISA array results for program 2.

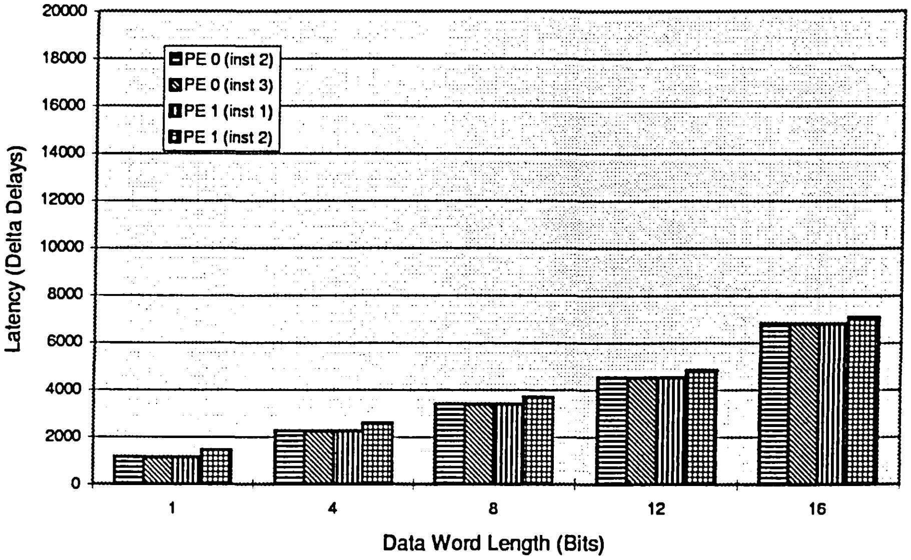


Figure 7.21. NRTZ ST-SISA array results for program 2.

The following trends can be observed from **Figures 7.20** and **7.21** for both the RTZ and NRTZ arrays when executing program 2:

- The latency of the read instruction (1) executed by PE1 is greater than the latency of the read instruction (3) executed by PE 0.
- The latencies of instructions 1, 2 and 3 are the same.

The results in **Figures 7.22** and **7.23** show the latency trends for the RTZ and NRTZ array simulations, respectively, for the executed instructions in program 3. It is assumed that program 1 has been previously executed to assign data to the array.

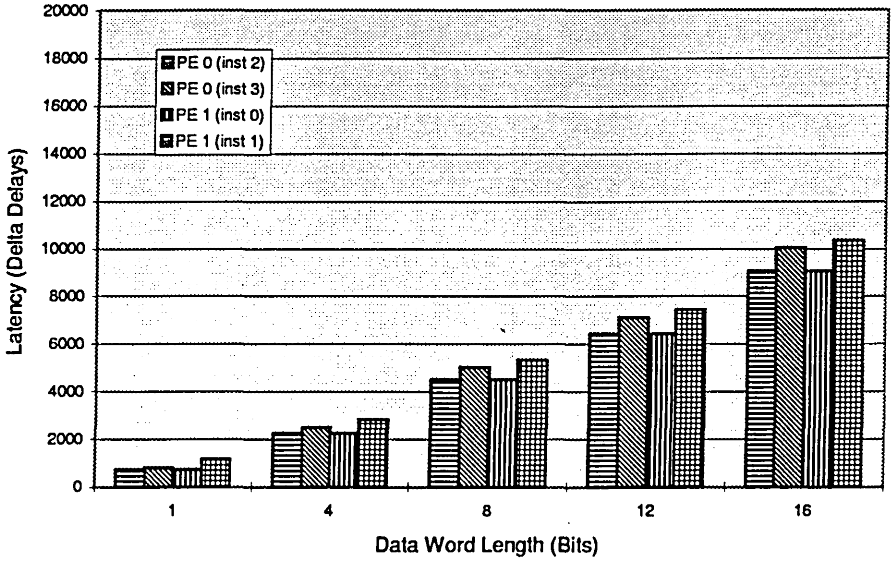


Figure 7.22. RTZ ST-SISA array results for program 3.

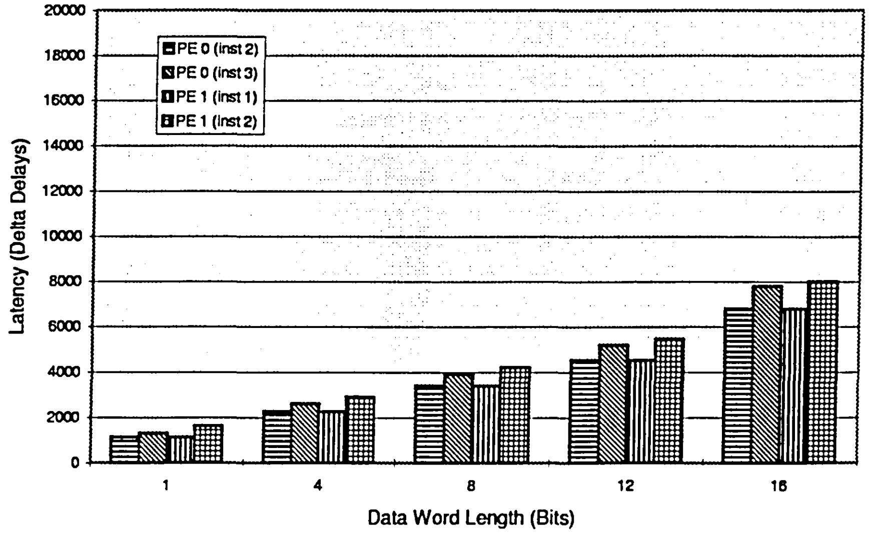


Figure 7.23. NRTZ ST-SISA array results for program 3.

The following trends can be observed from **Figures 7.22** and **7.23** for both the RTZ and NRTZ arrays when executing program 3:

- The latency of the MAX instruction (1) executed by PE1 is greater than the latency of the MIN instruction (3) executed by PE 0.
- The latencies of the write instructions (1 and 3) are the same.

7.4.1.2 Analysis

The above results show several features. Firstly, the first instruction executed by PE 0 and PE 1 take longer to execute than subsequent instructions because they cannot overlap serial-parallel instruction conversion with a preceding instruction. Secondly, the write and read instruction take the same time to execute because both are based on the assignment instruction. Finally, the first read instruction in program 1 when executed by PE 1 takes twice as long to execute because it must first wait for PE 0 to finish executing its read instruction and then begin executing the write instruction which then passes the appropriate data to PE 1.

7.5 Selection Guide-lines

This section provides guide-lines to aid computer architects in the selection of the appropriate RTZ, NRTZ or synchronous design technique to adopt, depending on the required SA PE implementation characteristics. These selection guide-lines are based on the circuit characteristics produced for each of the RTZ, NRTZ and synchronous PEs and are categorised into performance, based on the latency and throughput results, and gate count selection guide-lines, discussed as follows.

Performance Guide-lines

The performance of self-timed PEs is dependent on the length of the longest data word operated upon and the instructions executed. The performance of the synchronous PE is fixed to a static worst case speed dictated by the system clock, and is therefore not dependent on the instruction executed or data operated upon. The following guide-lines propose the appropriate technique to adopt to maximise a PEs performance when considering for differing data word lengths and instructions:

- The performances of the RTZ and NRTZ PEs are worse than the fixed worst case speed synchronous PE for all EXU instructions for almost all word lengths except a 1-bit word length. Therefore the synchronous design technique should be adopted to achieve maximum performance for algorithms which execute a high degree of EXU instructions, such as READ, WRITE, AND, OR, ADD, MIN and MAX.

- If the executed algorithm contains a high number of NOPs then the NRTZ PE should be adopted because it processes NOPs faster than the RTZ PE and considerably faster than synchronous PE.

Circuit Area Guide-lines

The synchronous PE is considerably smaller than both the RTZ and NRTZ PEs therefore it is the one to adopt if the maximum number of PEs are to be fitted onto a chip.

7.6 Conclusion

In this chapter the latency, throughput and circuit area of the RTZ and NRTZ PE architectures have been compared against one another, and their synchronous counterpart. It has been shown that in a small number of cases that the RTZ and NRTZ PE implementations can improve their performance as compared with their synchronous equivalent, however, this is at the cost of a significant amount of extra circuitry. Instruction and data communication has been proved to work correctly for the selected algorithms. Through executing these programs it has been shown that the self-timed nature of circuit operation also has an effect on PE instruction latency.

CHAPTER 8

CONCLUSIONS

8.1 Objectives of the Chapter

The main aim of this chapter is to conclude the thesis. It begins by summarising the main aims and conclusions from each of the previous investigations. The chapter also addresses the limitations of the work and the possible extensions to the existing investigations, as well as further areas of research. Finally, the thesis is concluded by assessing the success of the research by establishing whether the research aims have been achieved.

8.2 Review of Aims and Objectives

The body of the thesis was divided into six main areas. Firstly, in **chapter 2**, the principles of systolic array architectures were introduced. A taxonomy was presented which was used to classify the various reviewed key SA architectures. Analysis of these key SAs identified which features were common to these architectures, which features differed, and the significance for the choice of a particular feature. This was followed by addressing the main drawback of synchronously controlled SAs, that of clock skew, and two existing synchronous techniques that attempt to overcome this problem. An in-depth review of the alternative self-timed techniques, which use local data flows to co-ordinate correct circuit operation, was then undertaken. This review identified the main implementation techniques adopted to realise self-timed circuits, methodologies used to formally describe their operation and techniques of interconnecting circuit modules together. A representative range of key self-timed architectures was then reviewed and analysed in order to assess the relative successes of the various formal circuit design methodologies used in existing 'real-world' self-timed processor designs. The conclusion of this chapter highlighted the following main points:

1. There are a wide variety of SA architectures that differ in circuit cost and performance, the reasons for which are due to their method of programming (SIMD or MIMD), data path implementation (Serial or Parallel), PE interconnection (Mesh or Linear) and synchronisation of data transfer (Synchronous or Data-driven).

2. A SA can be either globally or locally controlled, resulting in globally synchronous or locally controlled data driven PEs, respectively.
3. Global Timing of SAs causes problems of clock skew and limited scalability.
4. Self-timed control abolishes the clock therefore overcoming clock skew and its associated problems.
5. An architectural scaleable SA, that does not require redesign when ported between implementation technologies, can only be realised with delay-insensitive self-timing techniques.
6. Formal methods for self-timed system design are still in their infancy consequently the macro-module approach predominates.

Almost no information could be gained from the review chapter regarding the trade-offs of self-timing design methods, compared with traditional synchronous techniques, for SA architecture realisation. As a consequence an investigation was proposed that was to be performed upon a single SA architecture, the ST-SISA, thus providing 'a level playing field' in which a fair comparison of the existing synchronous and the new self-timing strategies could be undertaken. To address these issues a methodological study comprising two primary aims were identified:

The first aim was to derive a coherent design framework to determine the differences between implementation techniques. Three objectives were identified.

1. To determine requirements for the implementation of self-timed architectures when used for programmable SA design.
2. The development of novel self-timed architectural structures based on the requirements developed in (1) which lead to scaleable and modular array designs.
3. To characterise the implementations in terms of area and performance for the existing synchronous and new self-timed architectural designs and to compare their relative trade-offs.

The second aim was to establish strategies for the design of self-timed architectures utilising the framework from the first aim. Two objectives were identified:

1. To produce guide-lines for the design of new self-timed architectures which do not require re-design when ported between architectures or circuit technologies.
2. To produce guide-lines for the selection of self-timed or synchronous circuit design techniques for a given set of criteria.

In **Chapter 3**, three main experiments were proposed: the data path, control path and processing element experiments. Their aims, procedures, and intended outcomes were also summarised. The experimental methodology to be utilised by each of these experiments was described and the assumptions of all the experiments was stated.

In **Chapter 4,** the research vehicle was introduced and described in terms of its array interconnection, processing element architecture and programmability for both the existing synchronous and the new self-timed architectures. Two methods of delay-insensitive self-timed information encoding were introduced. To realise variable length bit-serial words two, new end-of-word (EOW) tagging techniques were conceived.

In **Chapters 5** and **6** the main thesis objectives were addressed through the undertaking of the data path and control path experiments, respectively. Both chapters presented the requirements for each of their respective functions. This allowed the data path and control path architectures to be realised. Circuit area and performance trade-offs were established and then compared for the self-timed and synchronous architectures. Guide-lines were developed to aid in the design of self-timed data path and control path functions. Finally, guide-lines for the selection of the appropriate implementation technique for a given set of criteria were presented.

In **Chapter 7,** for each architecture, the data path and control path were combined to realise a complete PE. The circuit area and performance characteristics were subsequently established for the self-timed and synchronous PEs, thus allowing their relative trade-offs to be determined.

8.3 Conclusions of the Experiments

This section summarises the conclusions obtained from the three experimental studies in this thesis. The conclusions are addressed according to the two principal aims: construction of the design framework and the formulation of self-timing strategies.

8.3.1 Construction of the Design Framework

The conclusions can be further sub-divided into design requirements for self-timed architectures, circuit construction and circuit characterisation.

8.3.1.1 Design Requirements for Self-Timed Architectures

The design requirements for the self-timed architectures were determined prior to the design of each of the data and control path functions. The requirements for each of these functions covered two main areas. Firstly, general self-timing design requirements considered the effects of self-timed operation on a function. Secondly, the encoding requirements considered the effects of adopting the selected encoding and EOW tagging techniques on a function. The main conclusions drawn from these two sets of requirements are presented in the following two sections:

General Self-timing Requirements: There are a number of requirements that are common to all the self-timed data path and control path functions considered in this thesis. The reason for these commonalities is due to the fundamental mode of circuit operation that requires that each circuit module, whether a control or data path module, must receive state changes on all inputs before its function is performed. The general self-timing requirements are as follows:

In order to guide each data bit through the data path, the control path must re-send the same control information to the control inputs of data path modules on each bit cycle until the end of the instruction cycle is encountered. If this requirement was not enforced then the entire PE would dead-lock because not all inputs into a data path module would receive a state change for each data bit processed, thus violating the fundamental mode of operation.

A device called a C-element was used to synchronise multiple acknowledge signals sent by several receiving circuit modules to a single transmitting module. The C-element operates in fundamental mode therefore its output can only change state when all of its acknowledge inputs change state. This causes problems when a circuit module sends data to only one or some of its possible receivers because not all acknowledge signals into the C-element will change state, resulting in circuit dead-lock. This problem was overcome by enforcing a pseudo-acknowledge signal on each acknowledge input not involved in the synchronisation of data transfer, thus allowing the output to change state. This requirement is demonstrated in the ST-SISA's DOU which can synchronise data transfer between one or more neighbouring PEs. This is achieved by setting a user-defined control code that selects which acknowledge inputs to monitor for synchronisation and which inputs are to have pseudo acknowledge signals enforced upon them.

The dynamic length of bit-serial data words will mean that EXU functions which operate upon two operands of differing lengths, such as the ADD function, will dead-lock because one input will cease to receive state changes before the other. However, this problem was overcome by appending logic '0' pseudo data bits to the end of the shorter operand such that the function can continue operating on the longer word without changing the output result.

Encoding Requirements: Two self-timed versions of the research vehicle were developed, namely the RTZ and NRTZ ST-SISA processors. The processors differ in their adopted data encoding and end of word tagging techniques. Both the RTZ and NRTZ codes use two bits to encode their data resulting in four possible code combinations for each technique. The RTZ encoding technique uses two codes to represent data, a third code, called the empty state, to separate data items in the data stream and the fourth code to signal the end of a data word.

The NRTZ encoding technique uses all four possible states to encode the data, leaving no available code to signal the EOW as in the RTZ. Signalling the EOW in an NRTZ bit-serial word was achieved by periodically reserving a bit position in the bit-steam (every fourth bit). Setting the EOW bit position to logic '0' denotes that more data bits for the present word are to arrive. When the EOW bit position is set to logic '1' the end of the word is signalled.

Although the encoding approaches are very different in the way they encode and tag data they are very similar in that both utilise all four possible code combinations. As a result, all NRTZ data path functions, and the majority of the RTZ data path circuit modules, were implemented using NRTZ circuitry. The only exceptions were for the RTZ arithmetic and logic functions which were implemented using RTZ circuitry. The problem with using NRTZ equivalents is that they produce incorrect results when processing RTZ encoded and tagged data. For the RTZ arithmetic/logic circuits the EOW Tag is converted into an empty bit before it enters the function and then converted back into a EOW Tag only when both inputs are EOW Tags and when there is no carry propagation.

All local control circuit modules in the RTZ and NRTZ ST-SISAs were implemented using RTZ circuitry as each data bit in an instruction cycle always takes the same route through the data path. As a consequence, a new data path route can only be set up in the first bit cycle of a new instruction cycle because it is the only time that a new instruction can be decoded. It was possible therefore to use an RTZ empty control code to select the route taken through the data path by an empty spacer bit or an EOW Tag bit. Both of these can take the same route as the previous data valid bit. Similarly, a logic '0' EVEN phase NRTZ control bit, which has the same encoding as the RTZ empty bit, was used to select the route taken through the data path by an EVEN phase data bit or EOW Tag bit. Both of these can take the same route as the previous NRTZ ODD phase data bit.

The NRTZ ADD function had two main problems associated with the NRTZ EOW tagging technique. If one data word is shorter than the other, in other words when a logic '1' EOW Tag is added to a logic '0' EOW Tag, the ADD function outputs the logic '1' EOW Tag prematurely. This problem was solved by converting the logic '1' EOW Tag into a logic '0' EOW Tag before passing it into the ADD function, and then converting the result into a logic '1' EOW Tag when both inputs are logic '1' EOW Tags, and there is no carry overflow.

The second problem with the NRTZ ADD is concerned with the propagation of the carry bit from the bit addition before the input of the unused (logic '0') EOW Tag bits to the next data bit addition in the instruction cycle. For example, if a logic '1' carry bit from the previous bit addition is added to two unused EOW Tag operand bits then a logic '1' EOW Tag sum bit is output from the adder and a logic '0' carry bit is passed to the next bit addition. These are both unwanted occurrences. Firstly, changing the input logic '0' EOW Tag bit to a logic '1' EOW Tag after addition would signal the EOW prematurely. Secondly, the logic level of carry bit would change from a logic '1' to a logic '0' thus causing the final result to be incorrect. These problems were overcome by adding one of the logic '0' EOW Tag bits to two copies of the carry bit, such that if the carry input is logic '1' the sum produced is '0' and the carry produced is '1', similarly if the carry input is '0' both the sum and carry outputs are '0'.

8.3.1.2 Development of Novel Self-timed Architectures

The second part of the design process involved building the RTZ and NRTZ data and control path architectural structures. This was achieved by connecting together the appropriate elements from the standard macro module library using the system construction methodology, defined in Appendix D, to realise each main data and control path function. The resultant architectures exhibit a number of common features as a consequence of their common self-timing and encoding requirements, such as the local control circuitry used for each arithmetic and logical function.

8.3.1.3 Design Characterisation

The final part of the design process was to establish the latency, throughput and circuit area characteristics of the data path, control path and PE architectures in order to compare the RTZ and NRTZ implementation characteristics against one another and with their synchronous counterpart. The conclusions are summarised as follows:

Data Path: Analysis revealed that the latency and throughput of each of the self-timed data path components is dependent on the length of the bit-serial data processed such that the shorter the length of the longest operand the higher their performance. Consequently, the RTZ and NRTZ data path functions have been shown to have lower latencies and higher throughputs than their fixed speed synchronous counterparts for short length words. However, the synchronous data path functions perform better than both the RTZ and NRTZ designs for long bit-serial data words.

Almost all the NRTZ data path functions achieve lower latencies and higher throughputs than their equivalent RTZ data path functions because they require fewer bit cycles to complete the same operation when performed on the same data values. However, the latency and throughput of the NRTZ ADD is worse than the equivalent RTZ ADD because of the extra circuitry required to cater for carry propagation.

The RTZ and NRTZ data path functions are considerably larger in circuit area than their equivalent synchronous functions because of the inherently larger size of self-timed circuits and the extra circuitry required to implement the self-timing and encoding requirements. NRTZ data path functions are larger than their RTZ counterparts. This is due to the increased complexity of the NRTZ multiplexor control circuits, as a consequence of the greater number of inputs, the ring buffer counters, and the increased complexity of the NRTZ data path architectures.

Control Path: Both the RTZ and NRTZ IBUs have fixed worst case latency and throughput because both operate on fixed length instruction words. As a consequence there was no leeway in which to improve their performance over that of the synchronous IBU. It was demonstrated that the NRTZ IBU's performance is better than the RTZ IBU since it requires fewer bit cycles to process each instruction word.

Analysis of the IFU and IDU modules has revealed that their latency and throughput is dependent on the length of the longest data word processed by the data path. This is because the IFU must regenerate, and the IDU must subsequently re-decode the data path control codes for each and every bit cycle until the end of the instruction cycle.

The latency and throughput of the RTZ and NRTZ IFU have been shown to be worse than their synchronous counterpart because of the considerable degree of processing time required to convert the bit-serial instruction into the equivalent bit-parallel instruction. In addition, the time required to re-generate the instruction codes for each bit cycle adds to the relative poor performance of the RTZ and NRTZ IFUs. The performance of the NRTZ IFU is worse than that of the RTZ IFU for short instruction cycles, however, it improves over that of the RTZ IFU when the length of bit-serial data operated upon increases. This is due to the NRTZ IFU initially taking longer than the RTZ IFU to convert the bit-serial instruction into the equivalent parallel instruction, but then requiring fewer re-generate cycles than the RTZ IFU to control each bit propagating through the data path.

The NRTZ control path uses the RTZ IDU as a result of the NRTZ IFU generating RTZ instruction codes, therefore, its performance is dependent on the number of bit-cycles required to complete the instruction cycle.

The RTZ and NRTZ IFUs have a considerably larger circuit area than their synchronous counterpart because of the extra circuitry associated with the re-generate function and the inherent larger size of self-timed circuits. The NRTZ IFU is larger than the RTZ IFU because of the extra circuitry required to convert the NRTZ word into the equivalent RTZ word.

Both the RTZ and NRTZ IBUs have a slightly larger circuit area compared to their synchronous counterpart due to their complex implementation. The NRTZ IBU is larger than the RTZ IBU because it uses a counter circuit to count in the bits of each instruction word.

The RTZ/NRTZ IDU is larger than its synchronous counterpart due to the extra circuitry required to monitor the data path status signals and the extra circuitry associated with self-timed circuit technology.

PE: The latency and throughput of both the RTZ and NRTZ PEs is directly related to the number of bit cycles each requires to complete an instruction cycle. Therefore, the greater the number of bit cycles the longer the latency and the lower the throughput. The RTZ and NRTZ PEs processing speeds are lower than the equivalent synchronous PE because their main data path units operate in serial as a direct result of the way the data path must be controlled. In addition, the inherent slow operating speed of self-timed circuits, due to the extra circuitry required to realise a circuit, is also a contributing factor to the poor performance. The NRTZ instruction latencies in general are less than the equivalent RTZ instruction latencies because of the fewer number of bit cycles required by the NRTZ PE compared to the number of bit cycles required by the RTZ PE to process the same data values.

The latency of each instruction, assuming the same self-timed circuit technology (i.e. RTZ or NRTZ) for the PE, and the same data word length, is approximately the same. This is attributed to the sequential operation of the data path. The conversion of a bit-serial instruction into the corresponding bit-parallel instruction while the preceding instruction is still being executed, in other words the overlapping of instructions, improves PE performance, however, this improvement is only small.

The best case throughputs of the RTZ and NRTZ PEs are significantly greater than the synchronous PE's fixed constant throughput. This can only occur when a high degree of no-operations are executed. However, the average and worst case throughputs of the RTZ and NRTZ PEs are much worse than the fixed synchronous throughput. Once again this is attributed to the sequential operation of the self-timed data path.

Both the RTZ and NRTZ PEs are both significantly larger in terms of circuit area than their synchronous counterpart for reasons previously explained in the data and control path conclusions.

8.3.2 Formulation of Self-timing Development Strategies

The second main research aim was to develop design guide-lines for use by computer architects to aid in the design of SAs. This involved two main tasks. Firstly, to develop guide-lines for the design of scaleable self-timed SA architectures. Secondly, to establish guide-lines for the selection of appropriate self-timed or synchronous design techniques to be used for the design of a particular SA given a set of design criteria. These are presented accordingly.

8.3.2.1 Design Guide-lines

Design guide-lines have been developed for each of the data and control path functions. The application of these guide-lines have been proved to be successful through the realisation of several new RTZ and NRTZ self-timed architectural designs.

8.3.2.2 Selection Guide-lines

Selection guide-lines have been proposed that aid the designer in the choice of the appropriate design technique to realise the required SA architecture. The guide-lines were sub-divided into performance and circuit area guide-lines for the data path, control path and final SA PE architecture. In conclusion, the guide-lines suggested that if a large number of PEs are to be fitted onto the same chip and/or if the lengths of the data words to be operated upon range between 8 and 16-bits then the synchronous design techniques should be used to realise the SA architecture. However, if the application algorithms include a high degree of no-operations and the length of the data to be operated upon ranges between 0 to 8 bits then the NRTZ should be adopted if circuit area is not a restrictive factor.

8.4 Limitations of the Work

There are a number of limitations considered as follows. Firstly, there are a wide variety of self-timed implementation techniques available each having their relative advantages and disadvantages. An exhaustive study comprising all of these self-timed techniques would be beyond the scope of this project. Therefore, only two self-timed design techniques were investigated.

The second experimental limitation has been the restriction of the investigation of the self-timed design techniques in a single SA architecture. However, generic guide-lines have been developed for the design of other self-timed SA architectures.

The third experimental limitation has been the restriction of the research to the hardware implementation of the SA architectures. Further work could consider the execution of algorithms on the self-timed SA architectures.

8.5 Further Work

Further work includes extending the current investigations presented in this thesis, and additional investigations that can be undertaken. These are described in the following two sections.

8.5.1 Extending the Current Investigations

Fabrication of the RTZ and NRTZ ST-SISA chips: Characterisation of the two versions of the ST-SISA have been achieved through the simulation and analysis using tools from the Mentor Graphics CAD suite. These tools have provided models for analysis, but the performance characteristics obtained have not been verified through the physical implementation of the RTZ and NRTZ ST-SISAs. However, because both the self-timing models in this thesis have been simulated using this same model then their performances are equally compromised.

Minimisation of Duplicated circuitry: The EXU has a high proportion of circuitry that does not need to be duplicated. Reducing this circuitry to one common function available for use by all the EXU functions requires further investigation.

8.5.2 Additional Investigations

Maximisation of Self-timing Benefits: This research has demonstrated that a scaleable SA architecture can be built, however, the resulting architecture consumes a large amount of circuit area. The reason for this is partially attributed to the large amount of control circuitry required to co-ordinate local and global operations for the adopted research vehicle, and is therefore not completely due to the self-timed design technique. However, the extra control circuit area overhead could be reduced through the development of a new SA PE architecture especially designed to make maximum efficient utilisation of the new self-timed techniques presented in this thesis.

Dynamic Length Instruction Words: This research has presented two self-timing techniques that signal the end of bit-serial data words. As a result, design guide-lines were developed that have been used to realise architectural structures that operate on variable length data words. However, no work has been undertaken on the development of techniques to signal the end of variable length bit-serial instruction words or design guide-lines required to realise structures that operate on variable length instruction words. Therefore, it is not clear to what degree the existing EOW tagging techniques and design guide-lines could be used, or what new design guide-lines, if any, must be developed.

8.6 Summary

This thesis has presented new RTZ and NRTZ design methods for the design of scaleable systolic array architectures. These methods were proven through the realisation of new RTZ and NRTZ self-timed architectures based on a common research vehicle. Five objectives were proposed and each was fulfilled resulting in the following:

1. A methodological approach to the design of scaleable bit-serial RTZ and NRTZ SA architectures.
2. New scaleable self-timed architectural solutions that exhibit common circuit architectures and modularity.
3. The determination of implementation characteristics for the new RTZ and NRTZ self-timed architectural designs. In addition, this has shown that the RTZ and NRTZ designs can increase their performance when operating on short length data words, or when executing simple instructions such as no-operations, when compared to the fixed performance of the synchronous design. This is at the expense of a substantial amount of extra circuitry.
4. Design guide-lines for the development of self-timed SA architectures verified through the realisation of the new self-timed RTZ and NRTZ ST-SISA processors.
5. Guide-lines for the selection of appropriate self-timed or synchronous techniques for given design criteria. These guide-lines were based on the characteristics obtained for the new RTZ and NRTZ ST-SISA and existing synchronous SISA processors.

REFERENCES

[1] Albicki A., "Self-Timed Digital Systems", Proceedings of the 4th Annual IEEE International ASIC Conference, pp TS-1/1-4, 1991.

[2] Annaratone M., Arnould E., et al, "Architecture of Warp", Proc. of Compcon, IEEE Compt. Soc. Press, pp 264-267, February 1987.

[3] Berkel (van) K., "Beware the Isochronic Fork", INTEGRATION, the VLSI journal 13, pp 103-128, 1992.

[4] Blevins D. W., Davis E. W., et al, "BLITZEN: A highly Integrated Massively Parallel Machine", Journal of Parallel and Distributed Computing, Vol. 8, No. 2, pages 150-160, February 1990.

[5] Borkar S., Cohn R., et al, "iWarp: An Integrated Solution to High-Speed Parallel Computing", IEEE Proceedings on the Supercomputing Conference, pp 330-339, November 1988.

[6] Brunvand E., "The NSR Processor", Proceedings of the 26th Hawaii International Conference on Systems Sciences, Vol. 1, pp 428-435, January 1993.

[7] Chu T. A., "On the Models for Designing VLSI Asynchronous Digital Systems", Integration, the VLSI Journal 4, pp 99-113, 1986.

[8] Chu T. A., "Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications", PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1987.

[9] David I., Ginosar R., Yoeli M., "Implementing Sequential Machines as Self-Timed Circuits", IEEE Transactions on Computers, Vol. 41, No. 1, pp 12-17, January 1992.

[10] David I., Ginosar R., Yoeli M., "Self-Timed Architecture of a Reduced Instruction Set Computer", Manchester IFIP Working Conference on Asynchronous Design Methodologies, IFIP Transactions, Vol. A-28, Elsevier Science Publishers, pp. 29-43, 1993.

[11] David I., Ginosar R., Yoeli M., "Self-Timed is Self-Checking", Journal of Electronic Testing: Theory and Applications, 6, Kluwer Academic Publishers, pp 219-228, 1995.

[12] Dean M. E., Williams T. E., "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)", MIT Conference on Advanced Research in VLSI, March 1991.

[13] Distante F., Giovanna Sami M., "A Protocol for Asynchronous Wavefront-Computation Arrays", From Systolic Arrays, in Proc. of First International Workshop on Systolic Arrays, Oxford, pp 249-258, 2-4 July 1988.

[14] Fisher A. L., Kung H. T., "Synchronizing Large VLSI Array Processors", IEEE Transactions on Computers, Vol. c-34, No. 8, pp 734-740, August 1985.

[15] Foulser D. E., Schreiber R., "The Saxpy Matrix-1: A General-Purpose Systolic Computer", IEEE Computer Magazine 17, 1, pp35-43, July 1987.

[16] Franklin M., Pan A. T., "Clocked and Asynchronous Instruction Pipelines", Proceedings of the 26th Annual International Symposium on Microarchitecture, pp 177-184, 1-3 December 1993.

[17] Furber S. B., "Breaking Step: The Return of Asynchronous Logic", IEE Review, Vol. 39, No. 4, pp. 159-162, July 1993.

[18] Furber S. B., "Computing Without Clocks: Micropipelining the ARM Processor", taken from The SUN Annual Lecture Notes in Computer Science at the University of Manchester, Computing Without Clocks: Asynchronous Microprocessor Design, 13-14 September 1994.

[19] Furber S. B., Day P., et al, "A Micropipelined ARM", Proceedings of the IEEE Computing Conference, IEEE Computer Society Press, March 1994.

[20] Furber S. B., Day P., et al, "The design and Evaluation of an Asynchronous Microprocessor", Proceedings of Computer Design Conference: Computers and Processors, IEEE Computer Society Press, pp 217-220, October 1994.

[21] Garside J. D., "A CMOS VLSI Implementation of an Asynchronous ALU", Manchester IFIP Working Conference on Asynchronous Design Methodologies, IFIP Transactions, Vol. A-28, Elsevier Science Publishers, pp. 181-207, 1993.

[22] Ginosar R., Michell N., "On the Potential of Asynchronous Pipelined Processors", Computer Architecture News, pp. 27-34, December 18, 1990.

[23] Greenstreet M. Steiglitz R., K., "Bubbles Can Make Self-Timed Pipelines Fast", Journal of VLSI Signal Processing, 2, pp 139-148, 1990.

[24] Greer B., Webb J., "Real-Time Supercomputing on iWarp", SPIE Vol. 1659 Image Processing and Interchange, pp 12-23, 1992.

[25] Hatamian M., "Understanding Clock Skew in Synchronous Systems", Concurrent Computations, pp 87-96, 1988.

[26] Hauck S., "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol. 83, No. 1, pp 69-93, January 1995.

[27] Heaton R., Blevins D., "A Bit-Serial Array Processing Chip for Image Processing" IEEE Journal of Solid-State Circuits, Vol. 25, No. 2, pp 364-368, April 1990.

[28] Hoare C. A. R., "Developments in Concurrency and Communication", Addison and Wesley Publishing Company, ISBN 0-201-17232-1, 1990.

[29] Hogg R. S., Lloyd D .W., Hughes W. I., "Using Occam and Transputers to Emulate Asynchronous Self-Timed Array Processors", Proceedings of the 15th International Conference on Information Technology Interfaces, Pula Croatia, ISSN 1330-1012, pp. 257-262, 15-18 1993.

[30] Hogg R. S., Lloyd D .W., Hughes W. I., "A Self-Timed Massively Parallel Architecture with Elastic Control Flow", Proceedings of the International Society of Computers and Their Applications Conference, Long Beach, California, USA, ISBN 1-880843-08-0, pp. 22-27, March 17-19 1994.

[31] Hogg R. S., Lloyd D .W., Hughes W. I., "Communication Techniques for a Self-Timed Massively Parallel Architecture", Proceedings of the First International Conference on Massively Parallel Computing Systems: The Challenges of General-Purpose and Special-Purpose Computing, IEEE Computer Society Press, Ischia, Italy, ISBN 0-8186-6322-7, pp. 55-61, May 2-6 1994.

[32] Hogg R. S., Lloyd D .W., Hughes W. I., "Self-Timed Communication Strategies for Massively Parallel Systolic Architectures", Parallel Processing: CONPAR 94 - VAPP VI, Proceedings of the Third Joint International Conference on Vector and Parallel Processing, Springer-Verlag Lecture Notes in Computer Science 854, Linz, Austria, ISBN 3-540-58430-7, pp. 557-567, September 1994.

[33] Hogg R. S., Hughes W. I., Lloyd D .W., "A Novel Asynchronous ALU for Massively Parallel Architectures", Proceedings of the Fourth EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 282-289, 24-26 January 1996.

[34] Horowitz M., "Clocking Strategies in High Performance Processors", Symposium on VLSI Circuits Digest of Technical Papers, pp 50-53, 1992.

[35] Huffman D., "The Synthesis of Sequential Switching Circuits", Journal of the Franklin Institute, Vol 257, No 3, pp. 161-304, March 1954.

[36] Hughey R., Lopresti D. P., "Architecture of a Programmable Systolic Array", International Conference on Systolic Arrays, IEEE, pp 41-49, 1988.

[37] Hwang K., "Computer Arithemetic: Principles, Architecture and Design", Jon Wiley & Sons, ISBN 0-471-06076-3, 1979.

[38] Jennings J. M., Heaton R. A., "Comparative Performance Evaluation of a New SIMD Machine" 3rd Symposium on the Frontiers of Massively Parallel Computation, pp 255-258, October 1990.

[39] Johnson K. T., Hurson A. R., "General-Purpose Systolic Arrays", IEEE Computer Magazine, pp 20-31, November 1993.

[40] Jones S., "Wafer-Scale Integration: Status of an Art", Proceedings of the Second IFIP WG 10.5 Workshop on Wafer Scale Integration, pp 19-27, 23-25 September 1987.

[41] Keezer D. C., Jain V. K., "Design and Evaluation of Wafer Scale Clock Distribution", Proceedings of the International Conference on Wafer Scale Integration, pp 168-175, 1992.

[42] Keezer D. C., Jain V. K., "Clock Distribution Strategies for WSI: A Critical Survey", International Conference on Wafer Scale Integration, pp 277-283, 1991.

[43] Kung H. T., "Why Systolic Architectures?", IEEE Computer Magazine 15, 1, pp 37-46, January 1982.

[44] Kung H. T., "Systolic Communication", International Conference on Systolic Arrays, IEEE, pp 695-703, 1988.

[45] Kung H. T., Lam M. S., "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays", Journal of Parallel and Distributed Computing 1, pp 32-63, 1984.

[46] Kung S. Y., "VLSI Array Processors", From Systolic Arrays, in Proc. of First International Workshop on Systolic Arrays, Oxford, pp 7-24, 2-4 July 1988.

[47] Kung S. Y., "VLSI Array Processors", Prentice Hall Publishers, ISBN 0-13-942749-X, 1988.

[48] Kung S. Y., Arun K. S., et al, "Wavefront Array Processor: Language, Architecture, and Applications", IEEE Transactions on Computers, Vol. c-31, No. 11, pp 1054-1066, November 1982.

[49] Kung S. Y., Gal-Ezer R. J., "Synchronous versus Asynchronous Computation in VLSI array Processors", SPIE Vol. 341 Real Time Signal Processing V, pp 53-65, 1992.

[50] Kung S. Y., Jean S. N., et al, "Wavefront Array Processors - Concept to Implementation", IEEE Computer Magazine, pp 18-33, July 1987.

[51] Kung S. Y., Prassanna Kumar V. K., "Wavefront Array Processor and Beyond", IEEE, pp 176-179, 1985.

[52] Lam P. N., Li H. F., "Hierarchical Design of Delay-Insensitive Systems", IEE Proceedings, Vol. 137, Pt. E, No. 1, pp 41-56, January 1990.

[53] Lang H. W., "ISA and SISA: Two Variants of a General Purpose Systolic Array Architecture", Proc. Second International Conference on Supercomputing, Vol. 1, pp 460-467, 1987.

[54] Lang H. W., "Das befehlssystolische Prozessorfeld - Architektur und Programmierung", Dissertation zur Erlangung des Doktorgrades De Mathematisch-Naturwissenschaftlichen Fakulät der Christian-Albrects-Universität zu Kiel, 1989.

[55] Lea R. M., "Wafer-Scale Integration: Motivation, Perspective and Potential", Proceedings of the Second IFIP WG 10.5 Workshop on Wafer Scale Integration, pp 3-17, 23-25 September 1987.

[56] Leighton T., Leiserson C. E., "Wafer-Scale Integration of Systolic Arrays", IEEE Transactions on Computers, Vol. c-34, No. 5, May 1985.

[57] Lloyd D., Jones S., "Self-Timed Fine-Grained Parallel Processing Array Design", Proceedings on High Performance Special Purpose Architectures, International Symposium on Computer Architecture, 23-24 May 1992.

[58] Lloyd D., "Design Methods For Asynchronous Circuits", PhD Thesis, Department of Electrical and Electronic Engineering, University of Nottingham, March 1995.

[59] Martin A. J., "Compiling Communicating Processes in Delay-Insensitive VLSI Circuits", Distributed Computing 1, pp 226-234, 1986.

[60] Martin A. J., "The Limitations to Delay-Insensitivity in Asynchronous Circuits" Proceedings of the 6th MIT Conference on Advanced Research in VLSI, MIT Press, 1990.

[61] Martin A. J., Burns S. M., "The First Asynchronous Microprocessor: The Test Results", Caltech Technical Report CS-TR-89-6, April 1989.

[62] Martin A. J., Burns S. T., et al, "The Design of an Asynchronous Microprocessor", Proceedings of Decennial Caltech Conference on VLSI, MIT Press, March 1989.

[63] McGregor M. S., Denyer P.B., Murray A.F., "A Single-Phase Clocking Scheme for CMOS VLSI", VLSI Research Conference, pp. 257-271, March 1987.

[64] Meng T. H., "Synchronization Design for Digital Systems", Kluwer Academic Publishers, ISBN 0-7923-9128-4, 1991.

[65] Moon C. W., Stephan P. R., et al, "Specification, Synthesis, and Verification of Hazard-Free Asynchronous Circuits", Journal of VLSI Signal Processing, 7, pp 85-100, 1994.

[66] Muller R., "Speed Independent Switching Circuit Theory", in Chapter 10, Switching Theory, Vol. 2, John Wiley & Sons, New York, 1960.

[67] Nanya T., Kuwako M., "On Signal Transition Causality for Self-Timed Implementation of Boolean Functions", Proceedings of the 26th Hawaii International Conference on Systems Sciences, Vol. 1, pp 359-368, January 1993.

[68] Nanya T., Ueno Y., et al, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor", IEEE Design and Test of Computers, pp 50-63, Summer 1994.

[69] Navabi Z., "VHDL: Analysis and Modelling of Synthesis Systems", McGraw-Hill, ISBN 0-07-046472-3, 1993.

[70] Nielsen C. D., "Design of Delay-Insensitive Circuits Using Synchronised Transitions", MSc Thesis, Department of Computer Science, Technical University of Denmark, Lyngby, January 1990.

[71] Nielsen C. D., Martin A. J., "Design of a Delay-Insensitive Multiply-Accumulate Unit", Proceedings of the 26th Hawaii International Conference on Systems Sciences, Vol. 1, pp 379-388, January 1993.

[72] Paver N. C., "The Design and Implementation of an Asynchronous Microprocessor", PhD Thesis, Department of Computer Science, University of Manchester Institute of Science & Technology, 1994.

[73] Paver N. C., Day P., et al, "Register Locking in an Asynchronous Microprocessor", Proceedings of the International Conference on Computer Design in Computers and Processors, IEEE Computer Society Press, pp. 351-357, October 1992.

[74] Poole N. R., "Self-Timed Logic Circuits", Electronics and Communication Engineering, pp 261-270, December 1994.

[75] Pountain D., "Computing Without Clocks", Byte Magazine, pp 145-150, January 1993.

[76] Rem M., "The Nature of Delay-Insensitive Computing", IV High Order Workshop, pp 105-122, September 1991.

[77] Richardson W., Brunvand F. E., "The NSR Processor Prototype", Technical Report UUCS-92-029, August 14 1992.

[78] Sarkar S., "Enhanced Systolic Arrays - A New Idea", Canadian Conference on Electronics and Computer Engineering, Vol. 2, pp 857-860, 14-17 September 1993.

[79] Sarkar S., Majumdar A. K., "An Instruction Systolic Array Implementation of the Two-Dimensional Fast Fourier Transform", Microprocessing and Microprogramming 33, North-Holland Publishers, pp 101-110, 1991.

[80] Schimmler M., Lang H. Maaβ W., R., "The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as an Add-On Board for Personal Computers", Proc. of the International Conference on High Performance Computing and Networking, pp 487-488, April 1994.

[81] Schimmler M., Schmeck H., "A Fault Tolerant and High Speed Instruction Systolic Array", VLSI 91, Elsevier Science Publishers, pp 471-480, 1992.

[82] Schmeck H., "A Comparison-Based Instruction Systolic Array", Parallel Algorithms and Architectures, North-Holland Publishers, pp 281- 292, 1986.

[83] Schmidt U., Caesar K., "Datawave: A Single-Chip Multiprocessor for Video Applications", IEEE Micro, pp 22-94, June 1991.

[84] Schmidt U., Mehrgardt S., "Wavefront Array Processor for Video Applications", IEEE International Conference on Computer Design, pp 307-10, September 1990.

[85] Schroder H., "Top-Design Designs of Instruction Systolic Arrays for Polynomial Interpolation and Evaluation", Journal of Parallel and Distributed Computing 6, pp 692-703, 1989.

[86] Schroder H., Strazdins P., "Program Compression on the Instruction Systolic Array", Parallel Computing 17, pp 207-219, 1991.

[87] Sham L., "The Evolution of Data Compression Techniques", Proceedings of the International Society of Computers and Their Applications Conference, Long Beach, California, USA, ISBN 1-880843-08-0, pp. 135-139, March 17-19 1994.

[88] Sietz C. L., "Asynchronous Machines Exhibiting Concurrency", ACM Conference on Concurrent Systems and Parallel Computation, pp 93-106, 1970.

[89] Sietz C. L., "Self-Timed VLSI Systems", Proceedings of Caltech Conference on VLSI, pp 345-354, January 1979.

[90] Sietz C. L., "System Timing", Chapter 7, Introduction to VLSI Systems, C. Mead and L. Conway, Addison-Wesley, ISBN 0-201-04358-0, pp 218-262, 1990.

[91] Smith S. G., Denyer P. B., "Serial-Data Computation", Kluwer Academic Publishers, ISBN 0-89838-253-X, 1988.

[92] Sparsø J., Staunstrup J., et al, "Design of Delay-Insensitive Circuits using Multi-Ring Structures", Proceedings of EURO-DAC, European Design Automation Conference, IEEE Comput. Soc. Press, pp 15-20, September 7-10 1992.

[93] Sparsø J., Staunstrup J., "Design and Performance Analysis of Delay-Insensitive Mult-Ring Structures", Proceedings of the 26th Hawaii International Conference on Systems Sciences, Vol. 1, pp 349-358, January 1993.

[94] Sproull R. F., Sutherland I. E., Molnar C., "Counterflow Pipeline Processor Architecture", Sun Microsystems Laboratories Inc. Technical Report TR-94-25, taken from The SUN Annual Lecture Notes in Computer Science at the University of Manchester, Computing Without Clocks: Asynchronous Microprocessor Design, 13-14 September 1994.

[95] Sridhar R., "Asynchronous Design Techniques", International ASIC Conference, IEEE, pp 296-299, 1992.

[96] Staunstrup J., Greenstreet M. R., "Designing Delay-Insensitive Circuits using Synchronised Transitions. Part I: Motivation", Technical Report 1989-05-10, 1989.

[97] Staunstrup J., Greenstreet M. R., "Designing Delay-Insensitive Circuits using Synchronised Transitions. Part II: The Formal Model", Technical Report 1989-05-28, 1989.

[98] Staunstrup J., Greenstreet M. R., "From High-Level Descriptions to VLSI Circuits", BIT 28, pp 620-638, 1988.

[99] Staunstrup J., Greenstreet M. R., "Synchronized Transitions", IFIP WG 10.5 SUMMER SCHOOL on Formal Methods for VLSI Design, Lecture Notes.

[100] Sutherland I. E., "Micropipelines", Communications of the ACM, Vol. 32, No. 6, pp 720-738, June 1989.

[101] Swartzlander E. E., "Systolic Arrays for Wafer Scale Integration", Contributions by Speakers at the International Conference on Systolic Arrays, pp 179-84, June 1989.

[102] Unger S. H., "Asynchronous Sequential Switching Circuits", John Wiley & Sons, ISBN 471 89632 2, 1969.

[103] Unger S. H., "A Building Block Approach to Unclocked Systems", Proceedings of the 26th Hawaii International Conference on Systems Sciences, Vol. 1, pp 339-349, January 1993.

[104] de Vel O., "Concurrent Simulation of Instruction Systolic Array Structures", International Workshop on Modelling, Analysis and Simulation of Computing and Telecommunication Systems, pp 38-43, 17-20 January 1993.

[105] de Vel O., Murthy V. K., "ISA - A New Parallel Architecture for Real-Time Robot Control and High-Speed Vision Systems", Automated Inspection and High Speed Vision Architectures II, SPIE Vol. 1004, pp 230-236, 1988.

[106] Venbekbergen P., Catthoor F., "Optimized Synthesis of Asynchronous Control Circuits from Graph-theoretic Specifications", Proceedings of the IEEE International Conference on Computer Aided Design Digest of Technology, pp 184-187, 11-15 November 1990.

[107] Verhoeff T., "Delay-Insensitive Codes - An Overview", Distributed Computing 3, pp 1-8, 1988.

[108] Yakovlev A. V., "On Limitations and Extensions of STG Model for Designing Asynchronous Control Circuits", Proceedings of the International Conference on Computer Design in Computers and Processors, IEE Computer Society Press, pp. 396-400, October 1992.

[109] Yakovlev A. V., Petrov A., et al, "Synthesis of Asynchronous Control Circuits from Symbolic Signal Transition Graphs", Manchester IFIP Working Conference on Asynchronous Design Methodologies, IFIP Transactions, Vol. A-28, Elsevier Science Publishers, pp. 71-85, 1993.

[110] Zhang, Wang C. J., "Wafer Scale Integration of one- and two-Dimensional Linear Arrays", Proceedings of the International Conference on Wafer Scale Integration, pp 65-74, 1992.

[111] Lloyd D. W, Jones S. R, "Digital Test Methodology", UK Patent No. 9225317.8, Assigned to the British Technology Group and the University of Nottingham, 1991.

[112] Lloyd D. W, Jones S. R, "Improved Self-Timed Circuit Design Method", IEE Electronic Letters, 78 (9), pp 492-493, 1991.

APPENDICES

This section shows sample schematic diagrams of the RTZ ST-SISA (Appendix A) and NRTZ ST-SISA (Appendix B) processing elements, sample VHDL code of selected library macro-modules (Appendix C) and the adopted design methodology (Appendix D).

APPENDIX A :
THE RETURN-TO-ZERO (RTZ) ST-SISA ARCHITECTURE

APPENDIX B :

THE NON-RETURN-TO-ZERO (NRTZ) ST-SISA ARCHITECTURE

APPENDIX C :

SAMPLE VHDL MACRO-MODULES.

A select number of sample VHDL macro-modules, listed below, are included in this section showing the approach adopted to modelling their operation.

C1. C-element.

C2. RTZ AND Function.

C3. NRTZ AND Function.

C4. RTZ Register Control Function.

C5. NRTZ Full Adder.

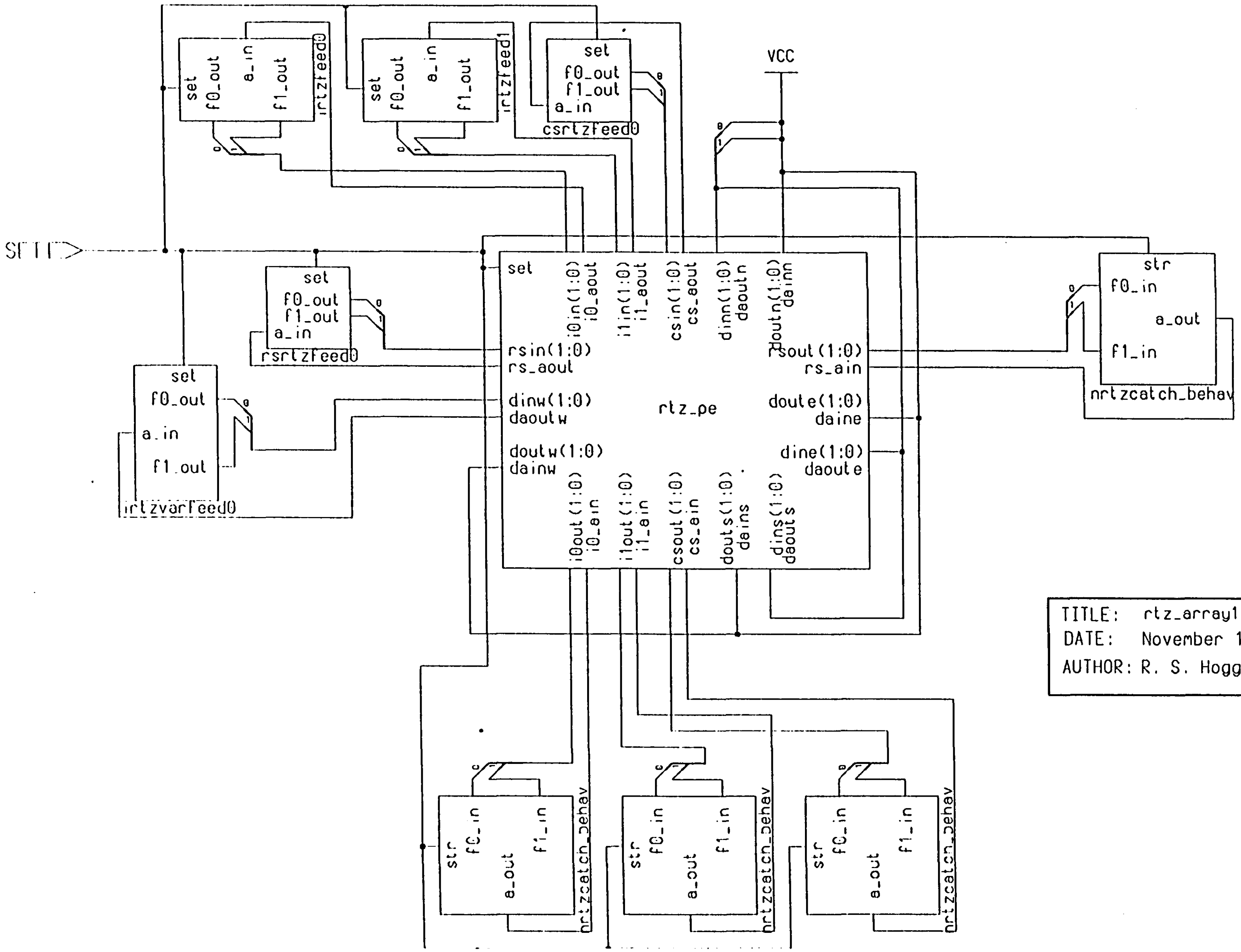
C6. RTZ Multiplexor.

APPENDIX D:

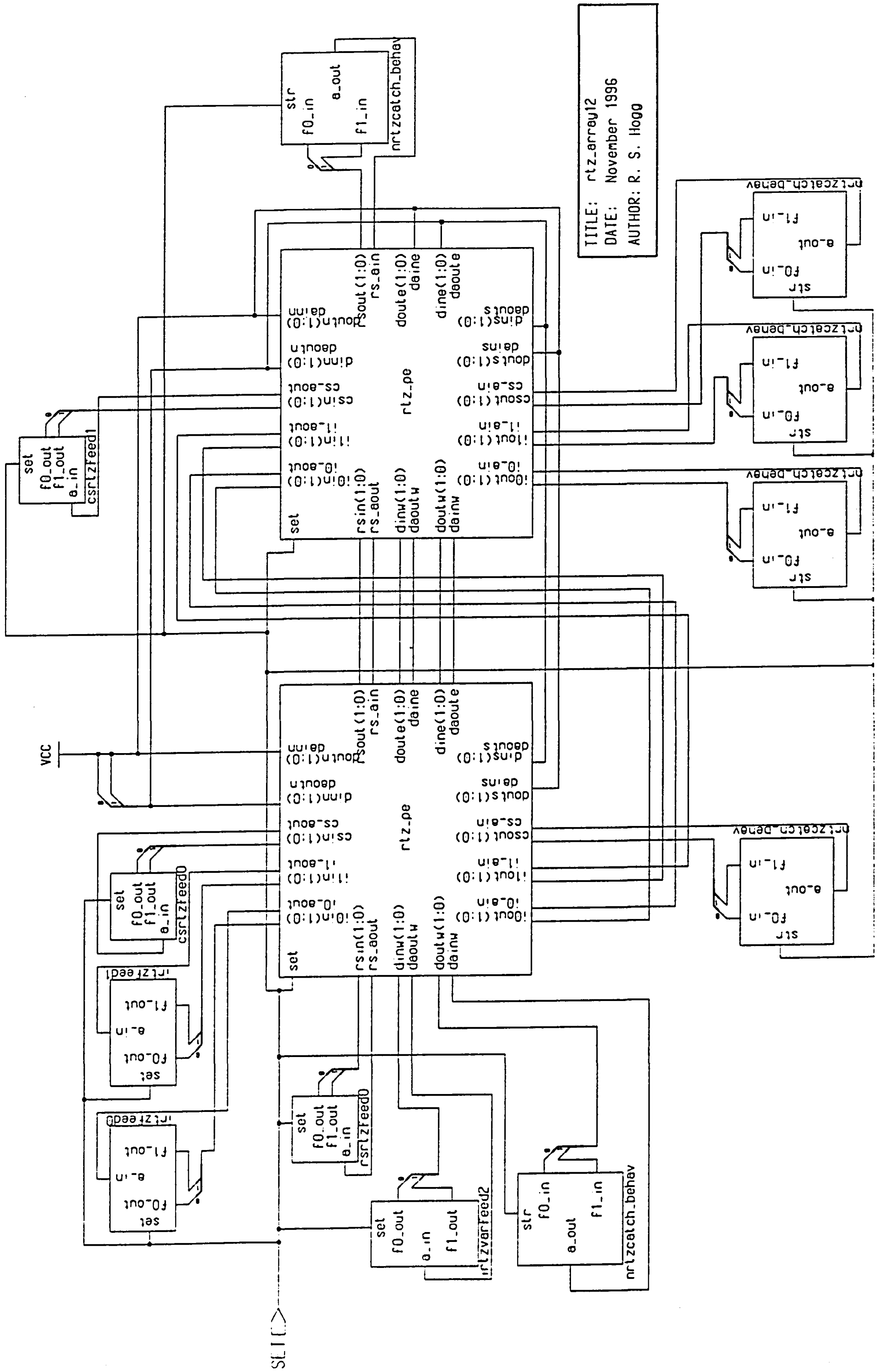
DESIGN METHODOLOGY.

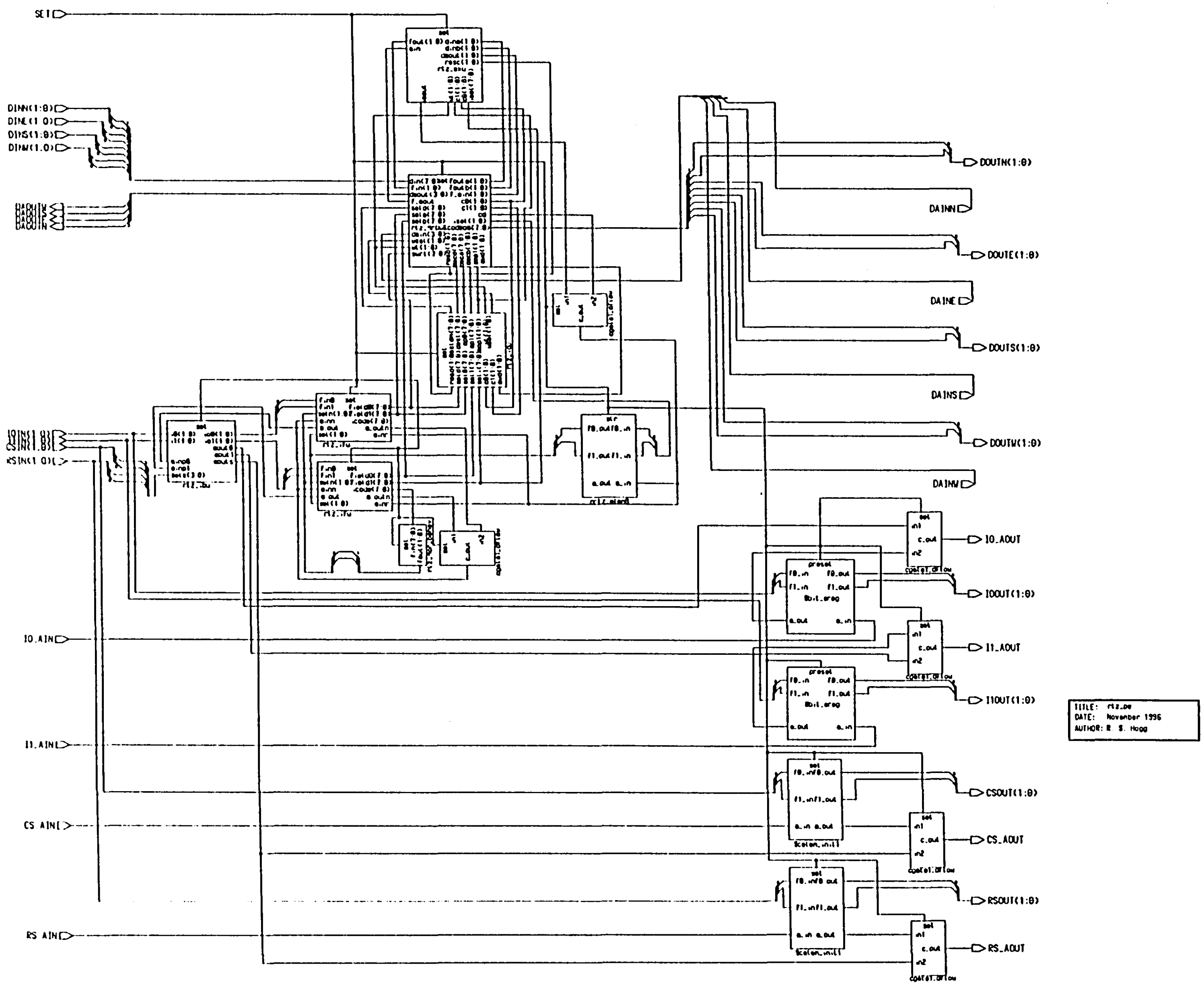
D1 - D7. Macro-Module Library.

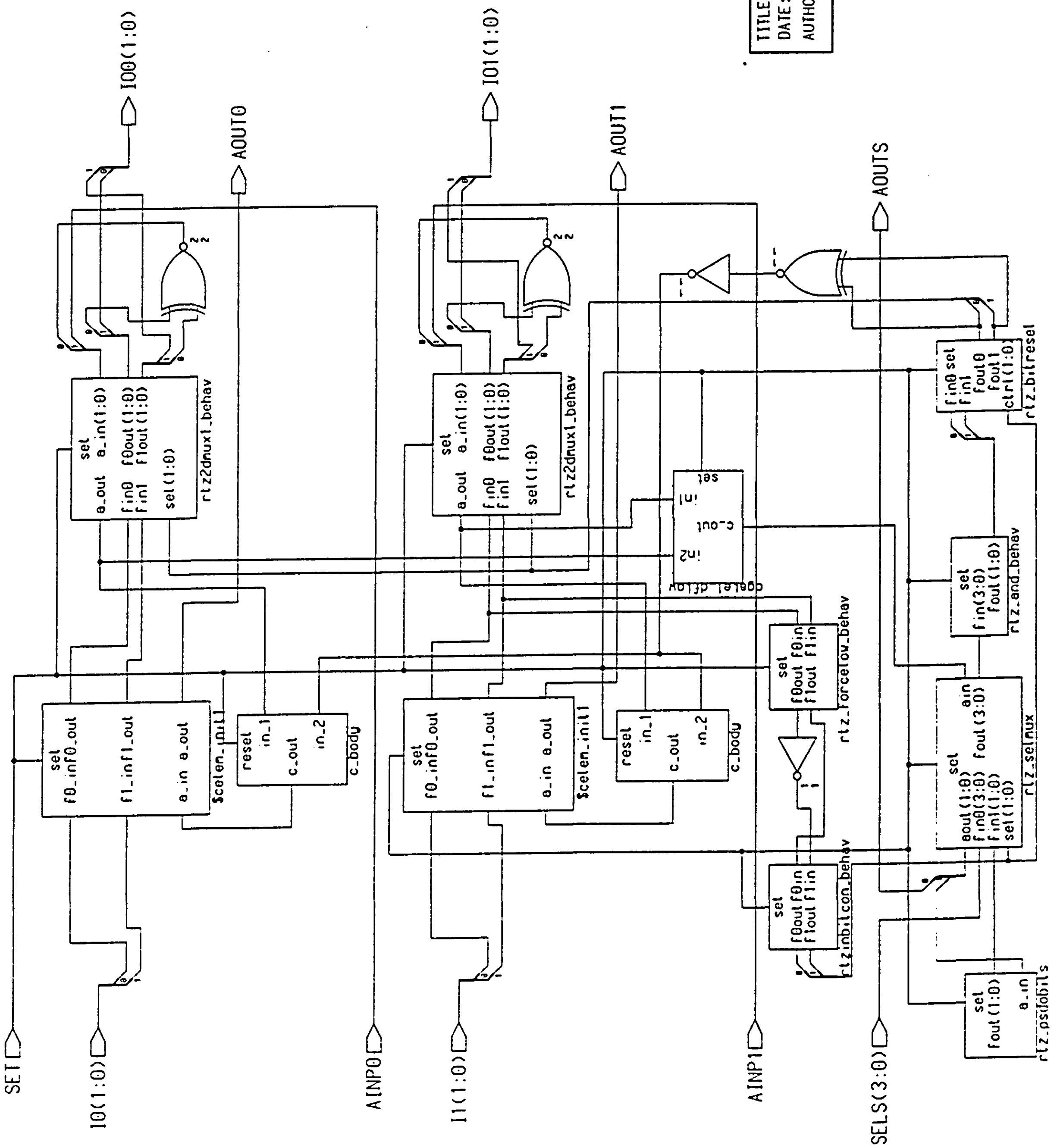
D8 - D11. System Construction Methodology.



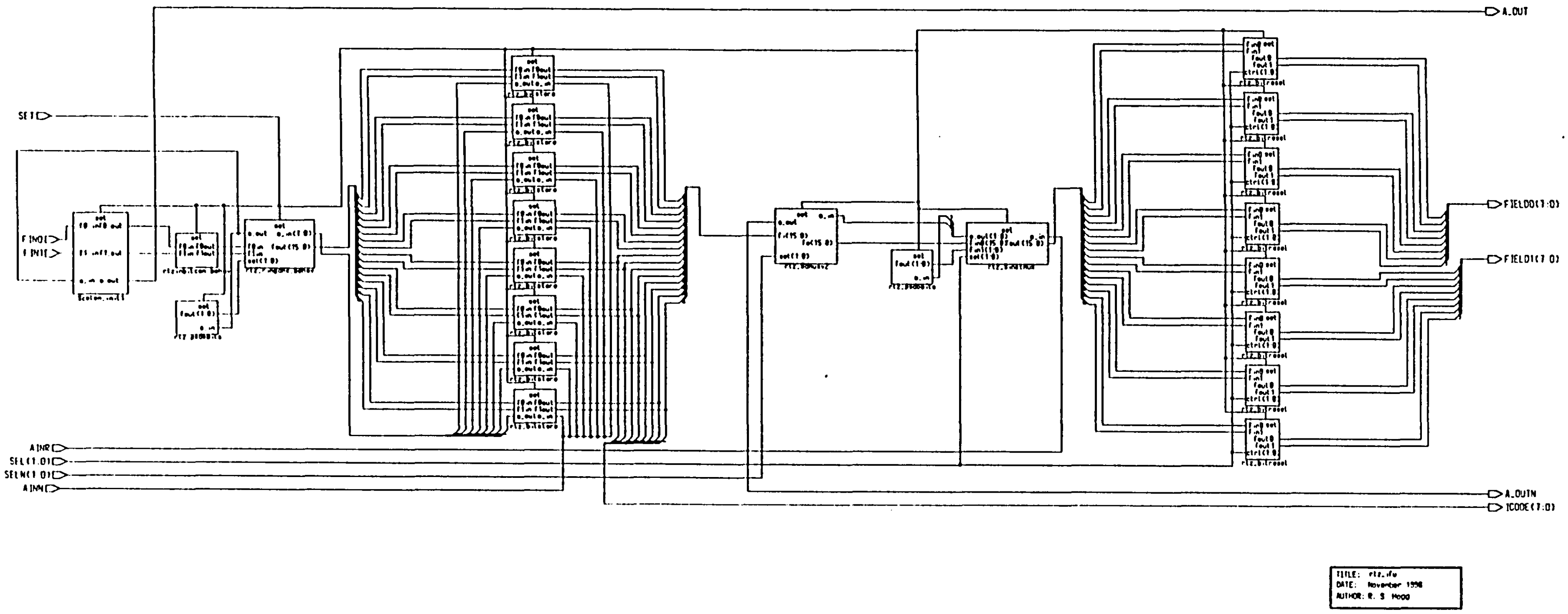
TITLE: rtz_array11
DATE: November 1996
AUTHOR: R. S. Hogg

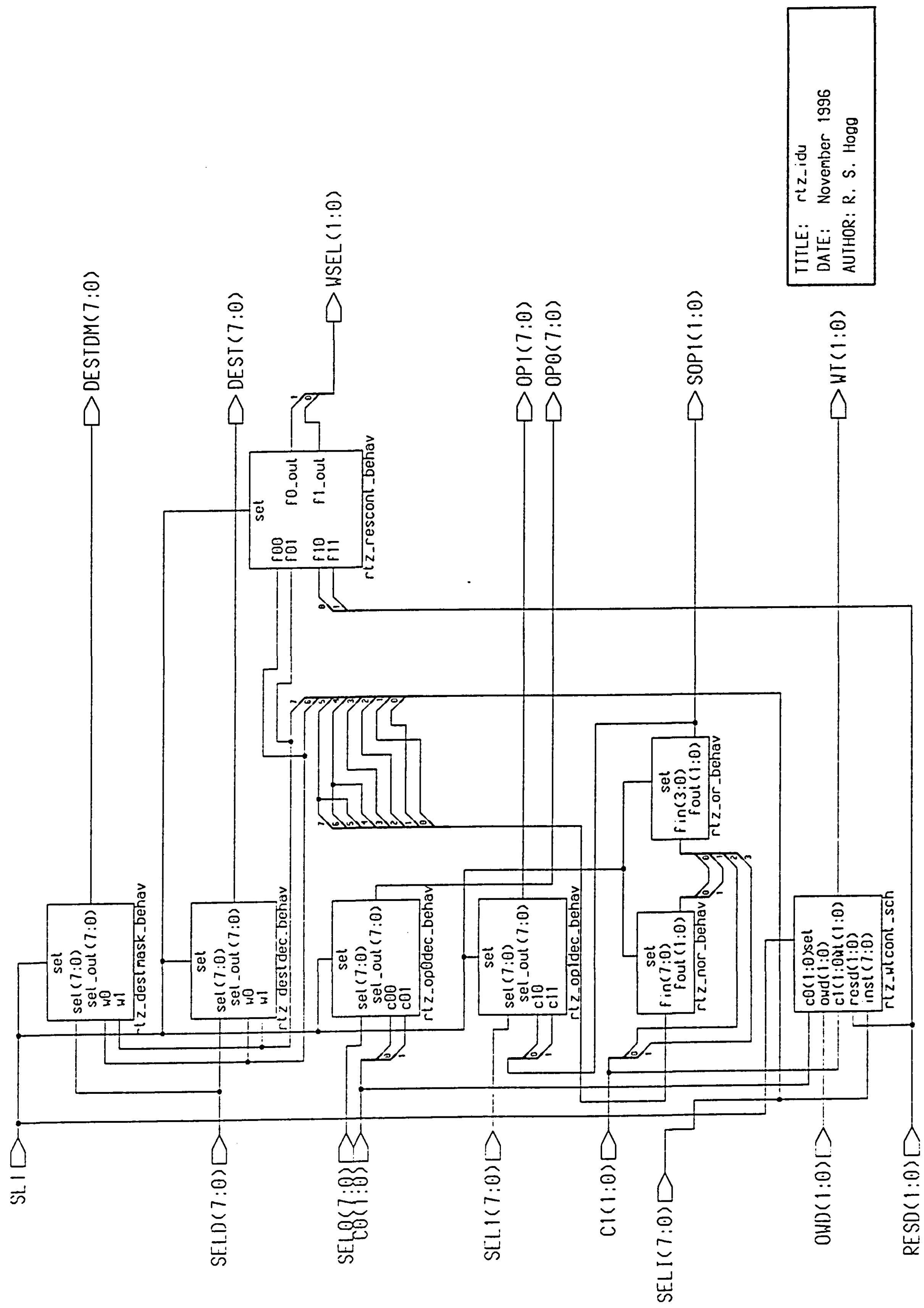

TITLE: rtz_array12
DATE: November 1996
AUTHOR: R. S. Hogg
rtz_pe
doutn(1:0) daoutn
dinn(1:0) daoutn
csin(1:0) cs_aout
i1in(1:0) i1_aout
i0in(1:0) i0_aout
set
rsout(1:0) rs_ain
doute(1:0) daine
dine(1:0) daoute
dins(1:0) daouts
douts(1:0) dains
csout(1:0) cs_ain
i1out(1:0) i1_ain
i0out(1:0) i0_ain
rsin(1:0) rs_aout
dinw(1:0) daoutw
doutw(1:0) dainw
nrtzcatch_behav
str
f0_in
f1_in
a_out
csrtzfeed1
csrtzfeed0
set
f0_out
f1_out
a_in
nrtzfeed1
nrtzfeed0
rsrtzfeed0
nrtzvarfeed2
VCC
SET

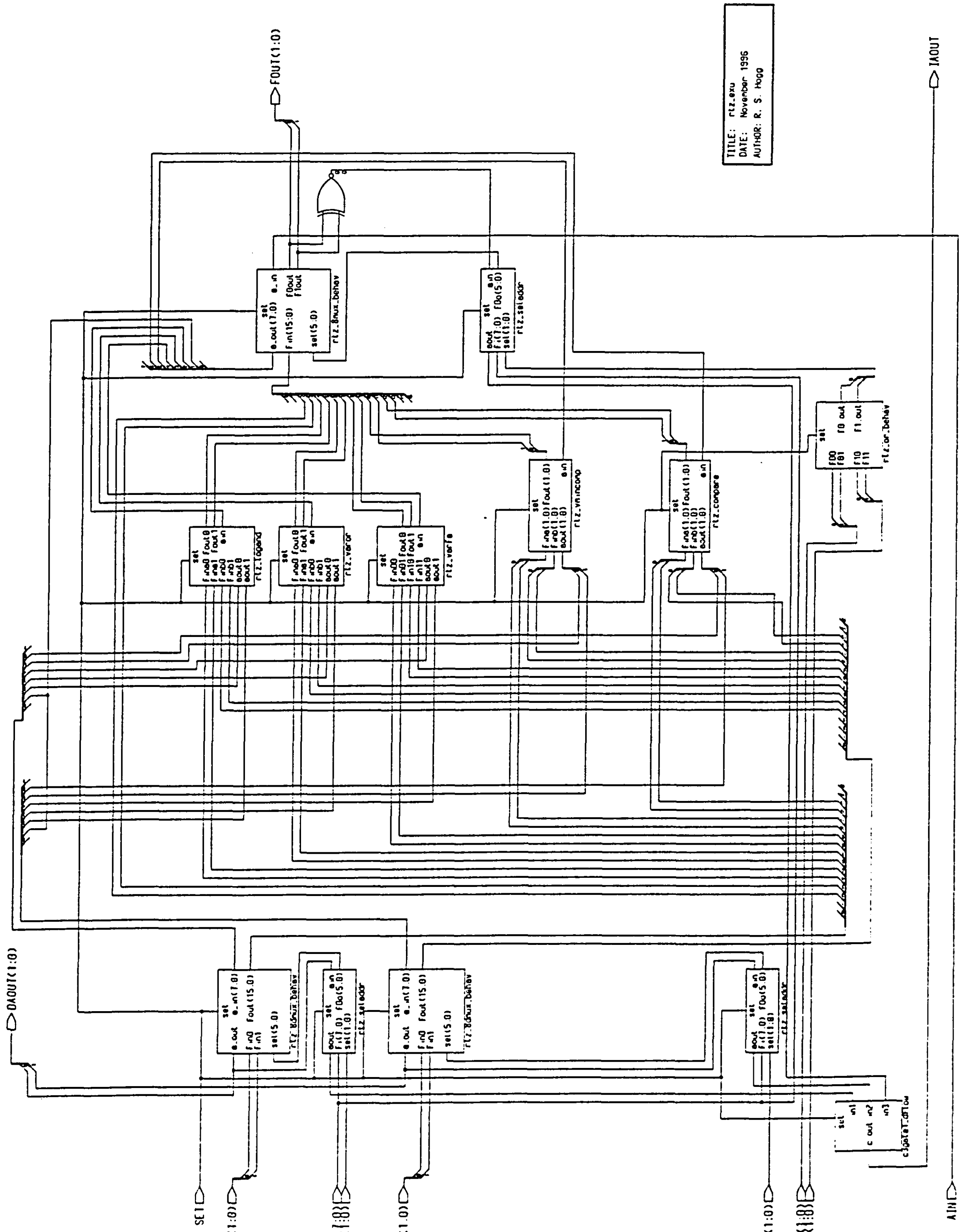

SET
DINN(1:0)
DINE(1:0)
DINS(1:0)
DINW(1:0)
DAOUTW
DAOUTS
DAOUTE
DAOUTN
IOIN(1:0)
I1IN(1:0)
CSIN(1:0)
RSIN(1:0)
IO_AIN
I1_AIN
CS AIN
RS AIN
fout(1:0)
a_in
d_ne(1:0)
d_nb(1:0)
dbout(1:0)
resc(1:0)
rlz_alu
f_out
rlz_fu
rlz_mixed
set
f0_out f0_in
f1_out f1_in
a_out a_in
cgate1_drlow
preset
f0_in
f1_in
f0_out
f1_out
8bit_areg
a_out
a_in
in1
c_out
in2
f0_in f0_out
f1_in f1_out
a_in a_out
8colon_unit
DOUTN(1:0)
DAINN
DOUTE(1:0)
DAINE
DOUTS(1:0)
DAINS
DOUTW(1:0)
DAINW
IO_AOUT
IOOUT(1:0)
I1_AOUT
I1OUT(1:0)
CSOUT(1:0)
CS_AOUT
RSOUT(1:0)
RS_AOUT
TITLE: rlz_pe
DATE: November 1996
AUTHOR: R. S. Hogg

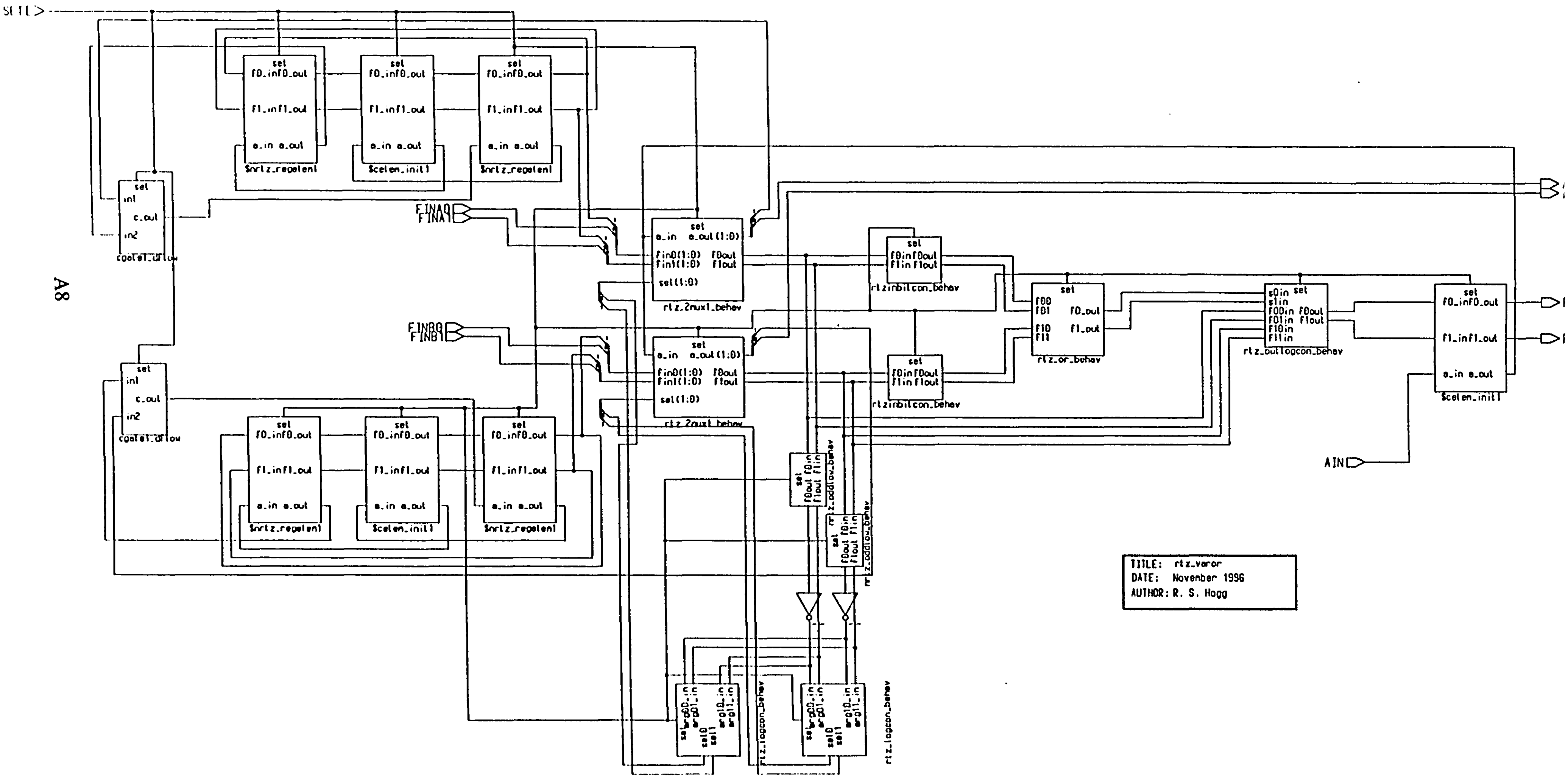


TITLE: rtz.ibu
DATE: November 1996
AUTHOR: R. S. Hogg

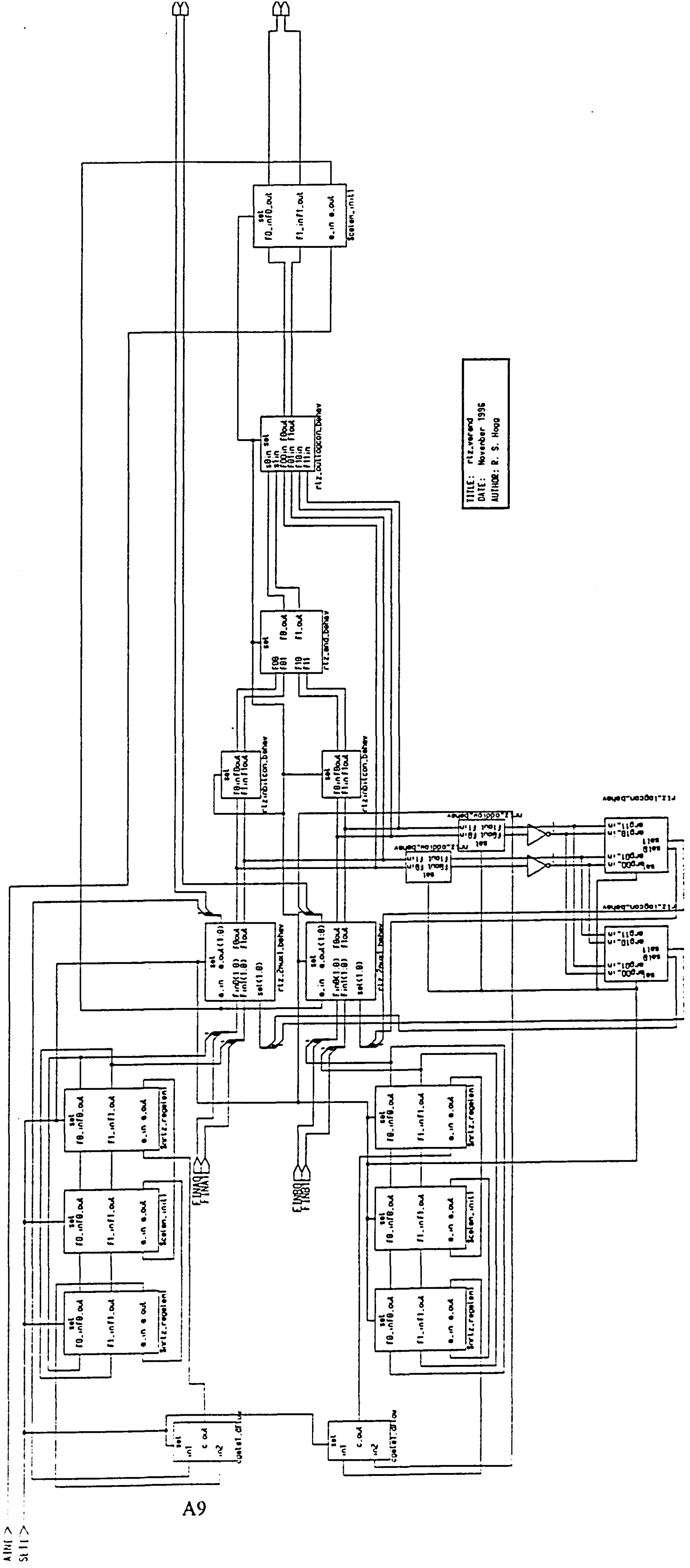

SET
FIN0
FIN1
AINR
SEL(1:0)
SELN(1:0)
AINN
A.OUT
FIELD0(7:0)
FIELD1(7:0)
A.OUTN
ICODE(7:0)
sel
f0.in f0.out
f1.in f1.out
o.in o.out
sel
f0.in f0out
f1.in f1out
sel
fout(1:0)
o.in
sel
o.out o.in(1:0)
f0.in fout(15:0)
f1.in
sel(1:0)
sel
f0.in f0out
f1.in f1out
o.out o.in
store
sel o.in
o.out
fi(15:0)
fo(15:0)
sel(1:0)
sel
fout(1:0)
o.in
sel
o.out(1:0) o.in
fin0(15:0) fout(15:0)
fin1(1:0)
sel(1:0)
fin0 sel
fin1
fout0
fout1
ctrl(1:0)
reset
TITLE: rtz.ifu
DATE: November 1998
AUTHOR: R. S. Mooo


TITLE: rtz_idu
DATE: November 1996
AUTHOR: R. S. Hogg
DESTDM(7:0)
DEST(7:0)
WSEL(1:0)
OP1(7:0)
OP0(7:0)
SOP1(1:0)
WT(1:0)
SL1
SELD(7:0)
SEL0(7:0)
C0(1:0)
SEL1(7:0)
C1(1:0)
SELI(7:0)
OWD(1:0)
RESD(1:0)
set
sel(7:0)
sel_out(7:0)
w0
w1
rtz_destmask_behav
set
sel(7:0)
sel_out(7:0)
w0
w1
rtz_destdec_behav
set
sel(7:0)
sel_out(7:0)
c00
c01
rtz_op0dec_behav
set
sel(7:0)
sel_out(7:0)
c10
c11
rtz_op1dec_behav
set
fin(7:0)
fout(1:0)
rtz_nor_behav
set
fin(3:0)
fout(1:0)
rtz_or_behav
set
f00
f01
f0_out
f10
f11
f1_out
rtz_rescont_behav
c0(1:0)
set
owd(1:0)
c1(1:0)
wt(1:0)
resd(1:0)
inst(7:0)
rtz_wtcont_sch
0
1
2
3
4
5
6
7

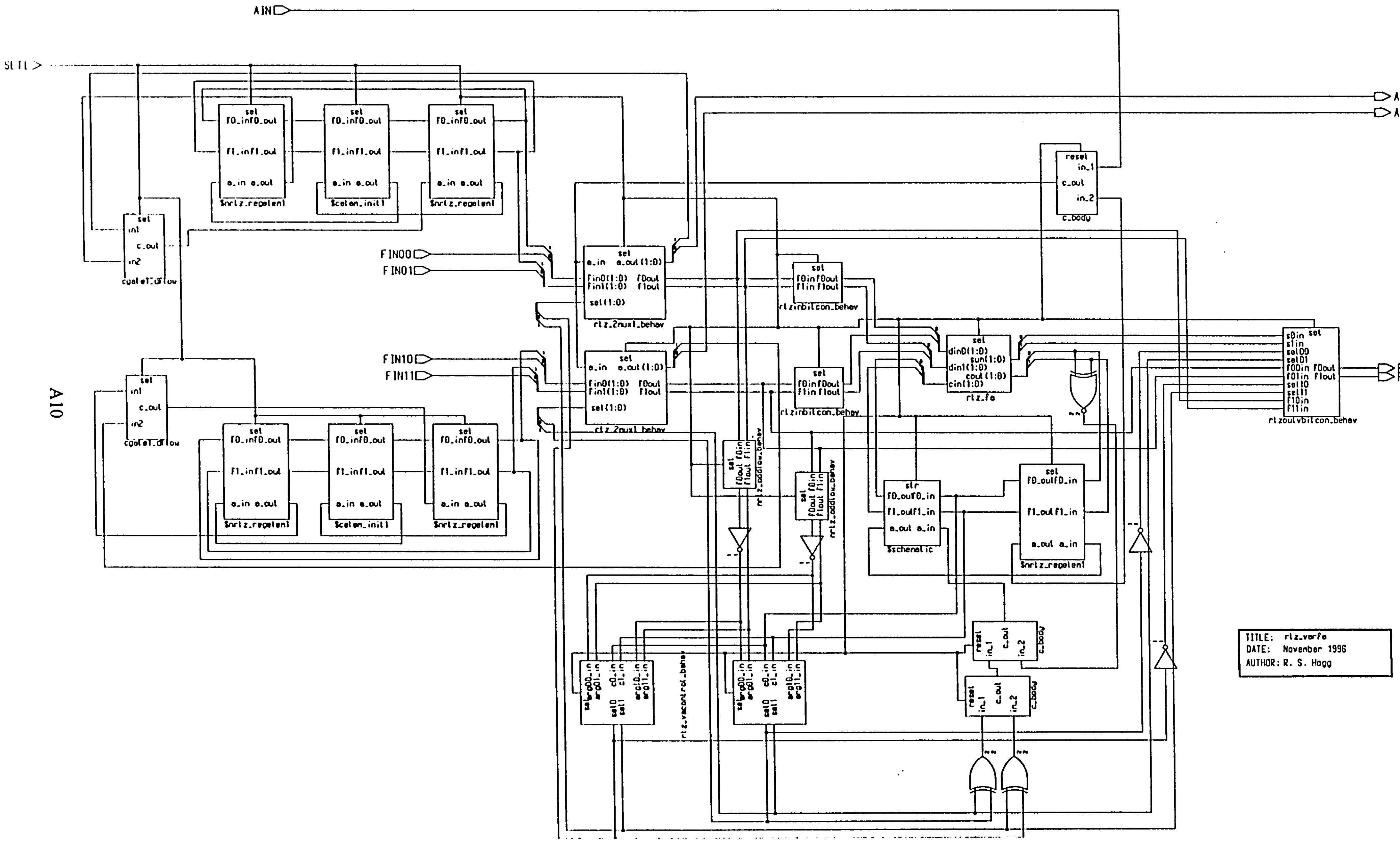

TITLE: rtz.exu
DATE: November 1996
AUTHOR: R. S. Hogg
FOUT(1:0)
IAOUT
DAOUT(1:0)
set
e.out(7:0) e.n
f.in(15:0) f0out f1out
sel(5:0)
rtz.8mux_behav
aout set e.n
f.i(7:0) f0o(5:0)
sel(1:0)
rtz.selector
set
f.na0 fout0
f.na1 fout1
f.nb0 e.n
f.nb1
aout0
aout1
rtz.logand
rtz.veror
f.n00
f.n01
f.n10
f.n11
rtz.verfe
set
f.na(1:0) fout(1:0) e.n
f.nb(1:0)
aout(1:0)
rtz.vmincomp
rtz.compare
set
f00 f0 out
f01
f10 f1.out
f11
rtz.or_behav
e.out e.n(7:0)
f.n0 fout(15:0)
f.n1
sel(5:0)
rtz.8mux_behav
set n1
c out n2
n3
c3gateT.dflow
SET
DINA(1:0)
ISEL(7:0)
CU(1:0)
DINB(1:0)
C1(1:0)
MI(1:0)
RESC(1:0)
AIN



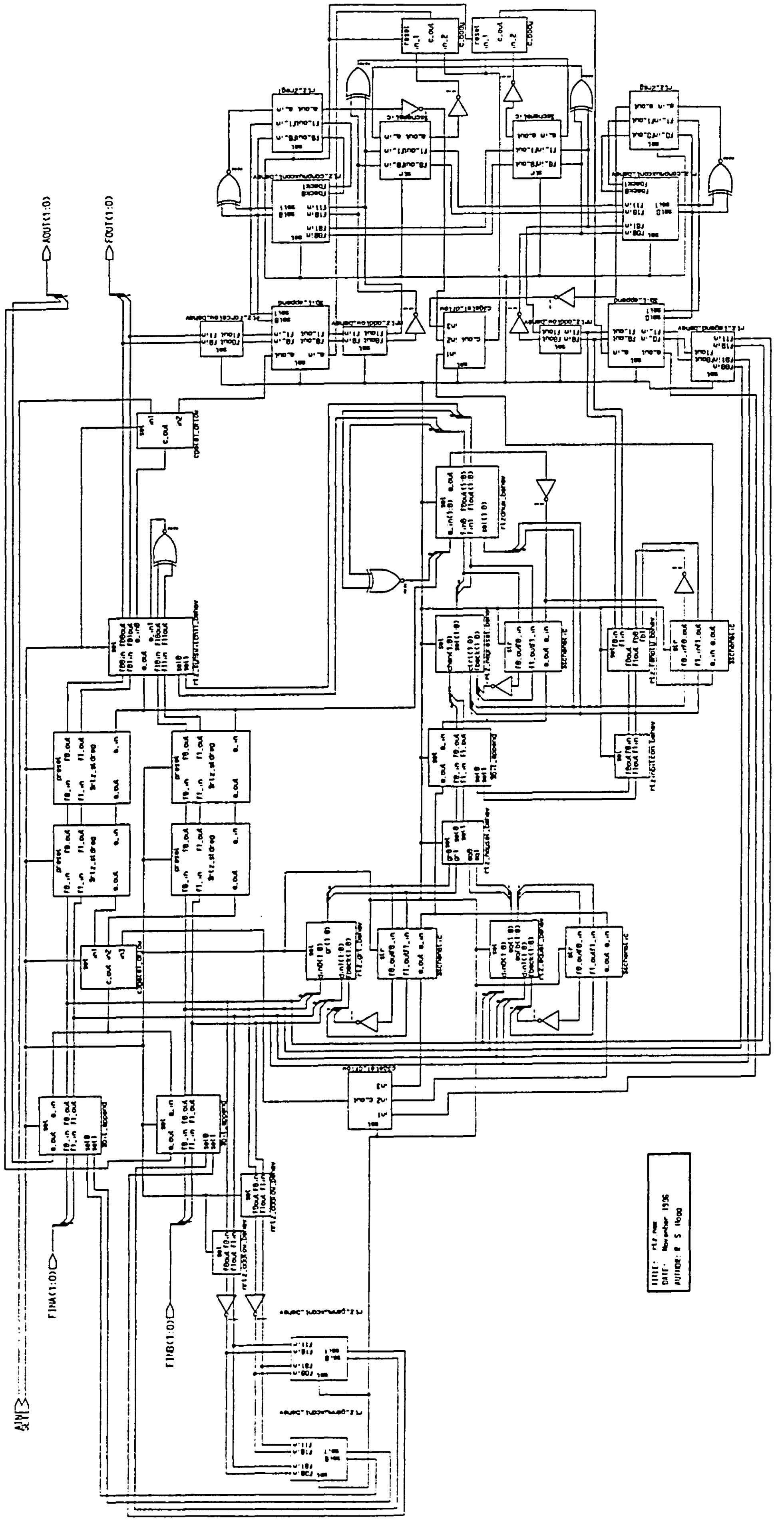
TITLE: rtz_veror
DATE: November 1996
AUTHOR: R. S. Hogg

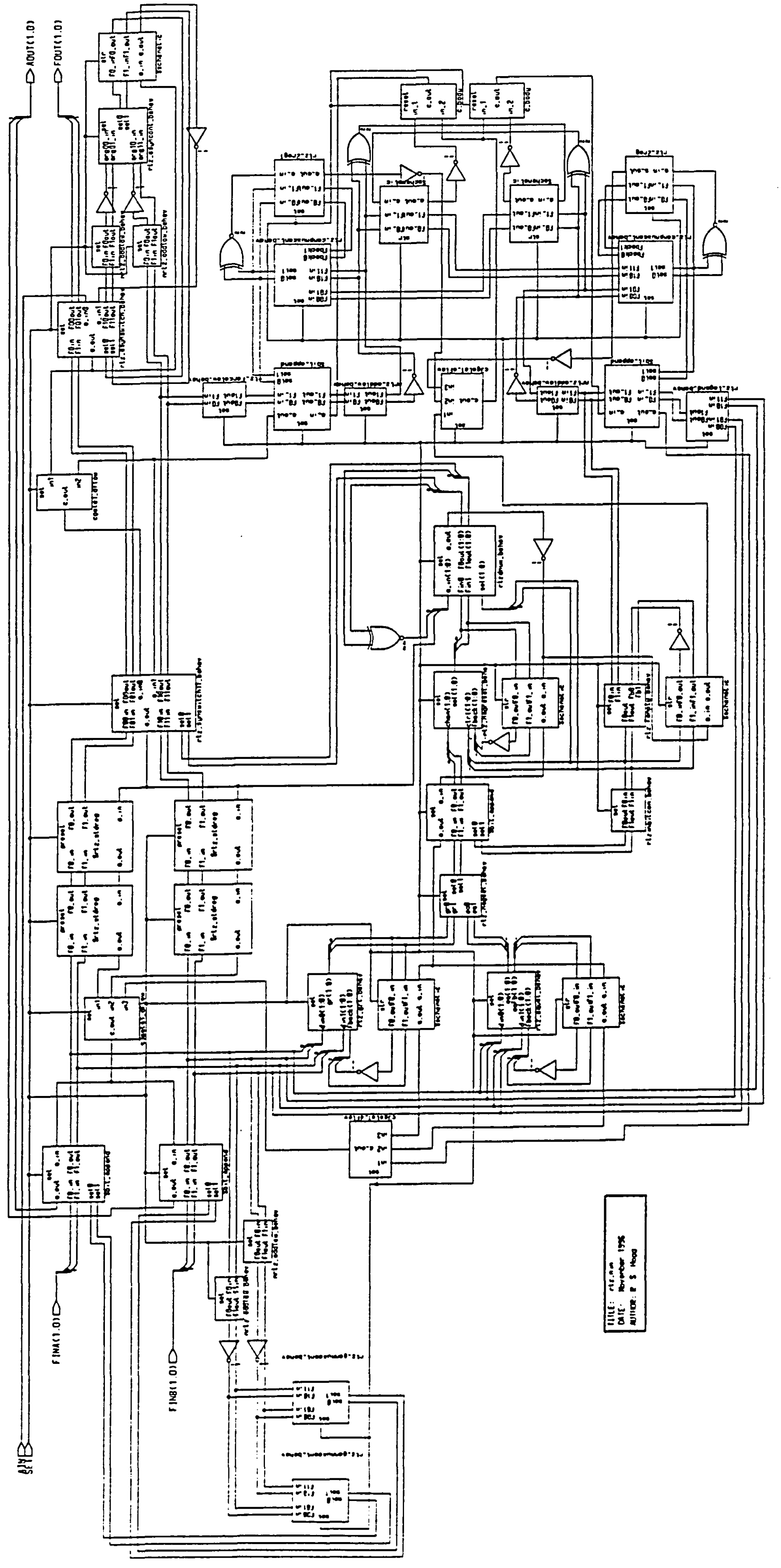

TITLE: rtz_verand
DATE: November 1996
AUTHOR: R. S. Hogg
sel f0_inf0_out f1_inf1_out e_in e_out
$celem_init1
s0in sel s1in f00in f0out f01in f1out f10in f11in
rtz_outlogcon_behav
sel f00 f0_out f01 f10 f1_out f11
rtz_and_behav
sel f0_inf0out f1_inf1out
rtzinbitcon_behav
rtz_oddinv_behav
sel f0out f0in f1out f1in
rtz_logcon_behav
sel0 sel1 erg00_in erg01_in erg10_in erg11_in
sel e_in e_out(1:0) fin0(1:0) f0out fin1(1:0) f1out sel(1:0)
rtz_2mux1_behav
$rtz_regelem
FINA0
FINA1
FINB0
FINB1
sel in1 c_out in2
cgatel_dflow
Ain[>
Stil >

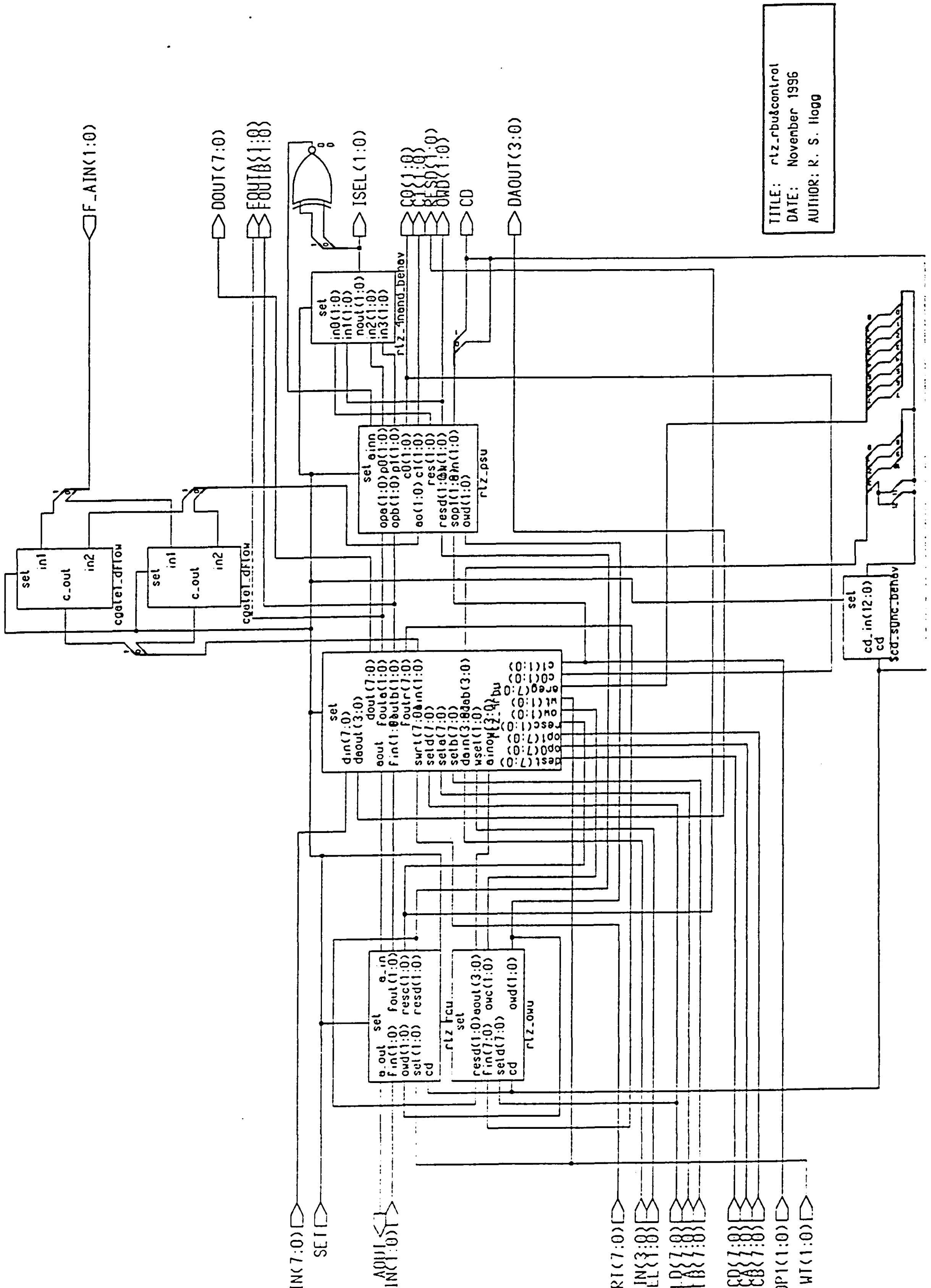
A9



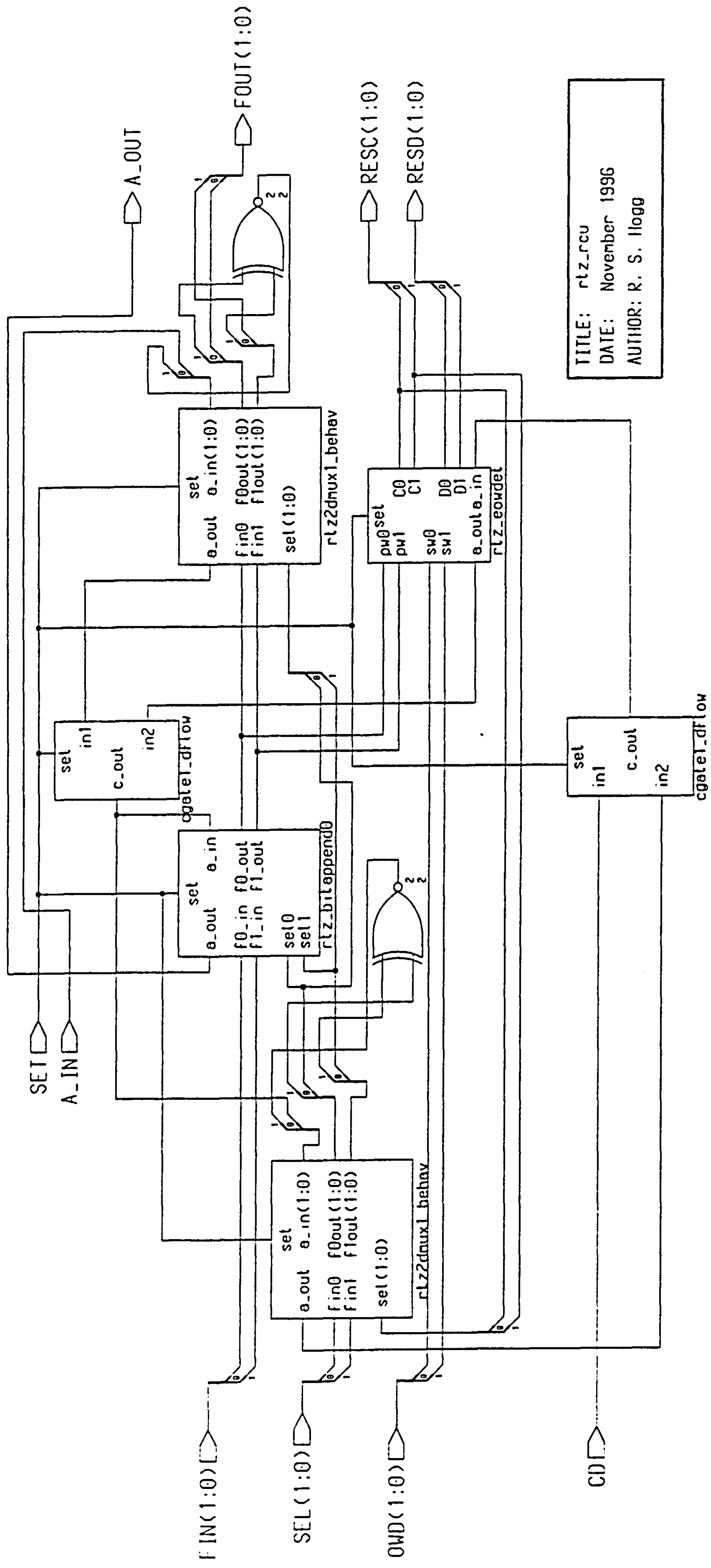
TITLE: rlz_verfe
DATE: November 1996
AUTHOR: R. S. Hogg

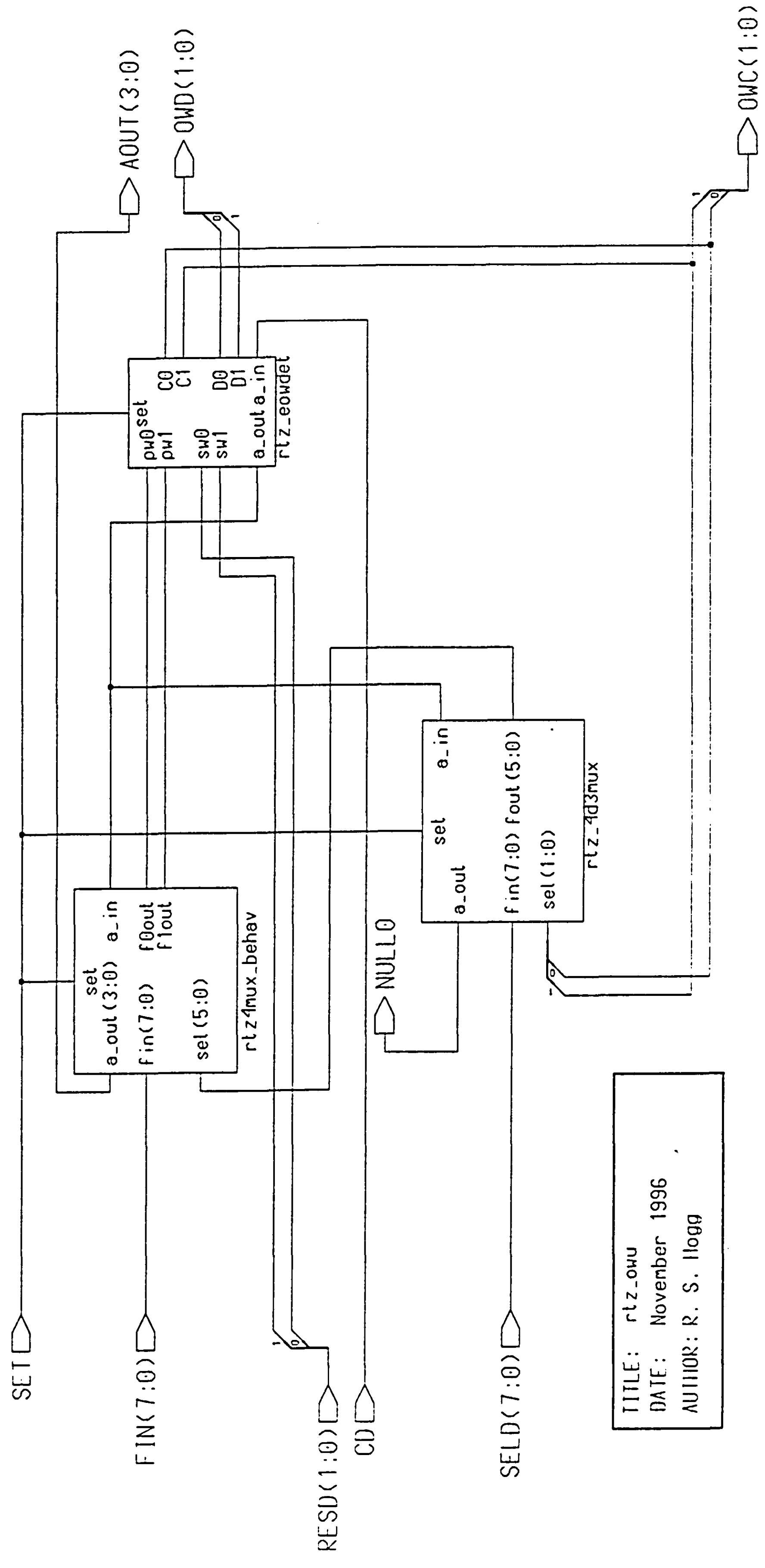

AOUT(1:0)
FOUT(1:0)
FINA(1:0)
FINB(1:0)
rtzdmux_behav
3bit_append
3bit_register
2bit_append
celement
c_bool
TITLE: rtz_mac
DATE: November 1996
AUTHOR: R S Hogg


AOUT(1:0)
FOUT(1:0)
FINA(1:0)
FINB(1:0)
SEL
TITLE: rtz.fsm
DATE: November 1996
AUTHOR: R S Hogg

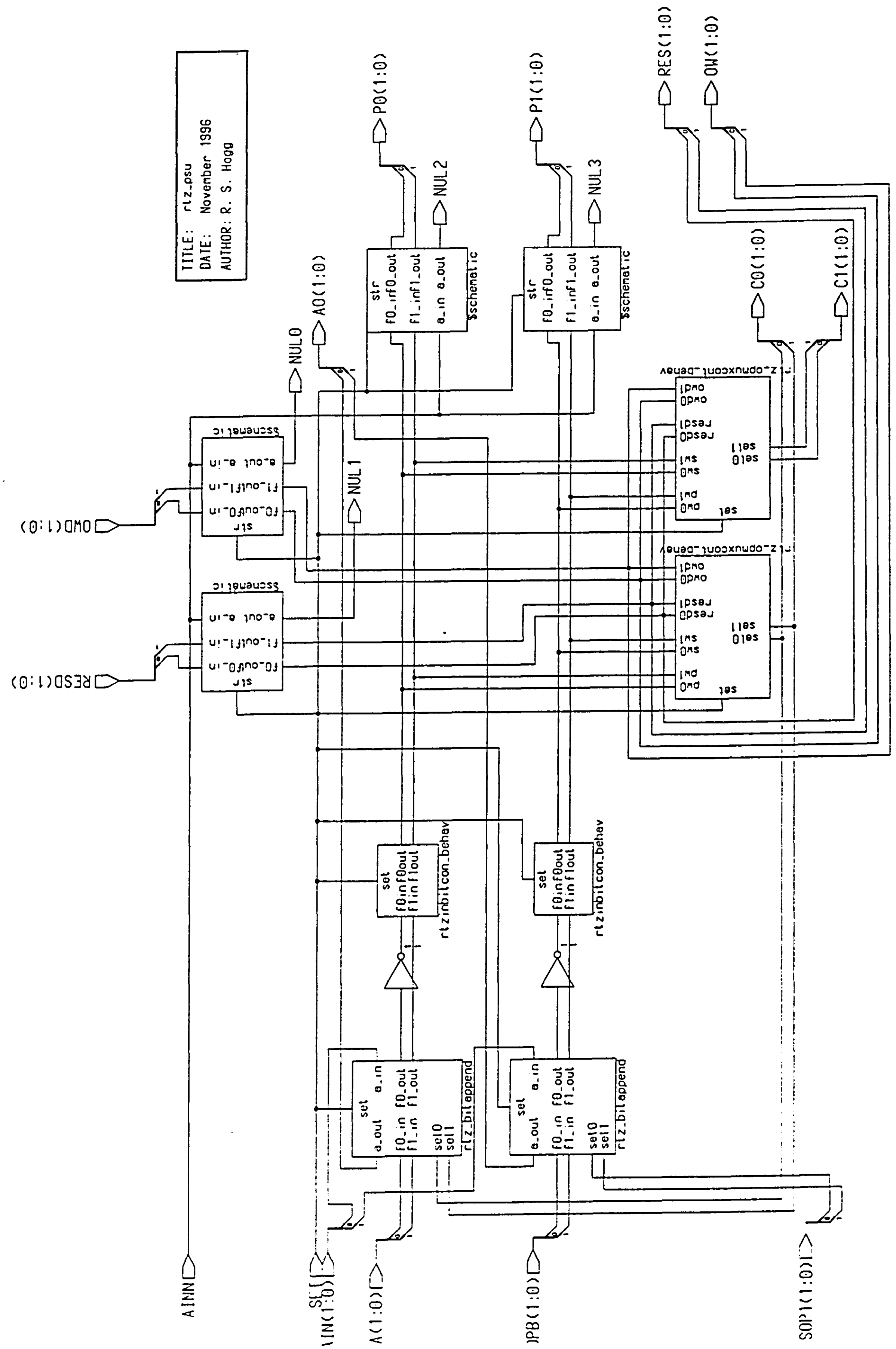


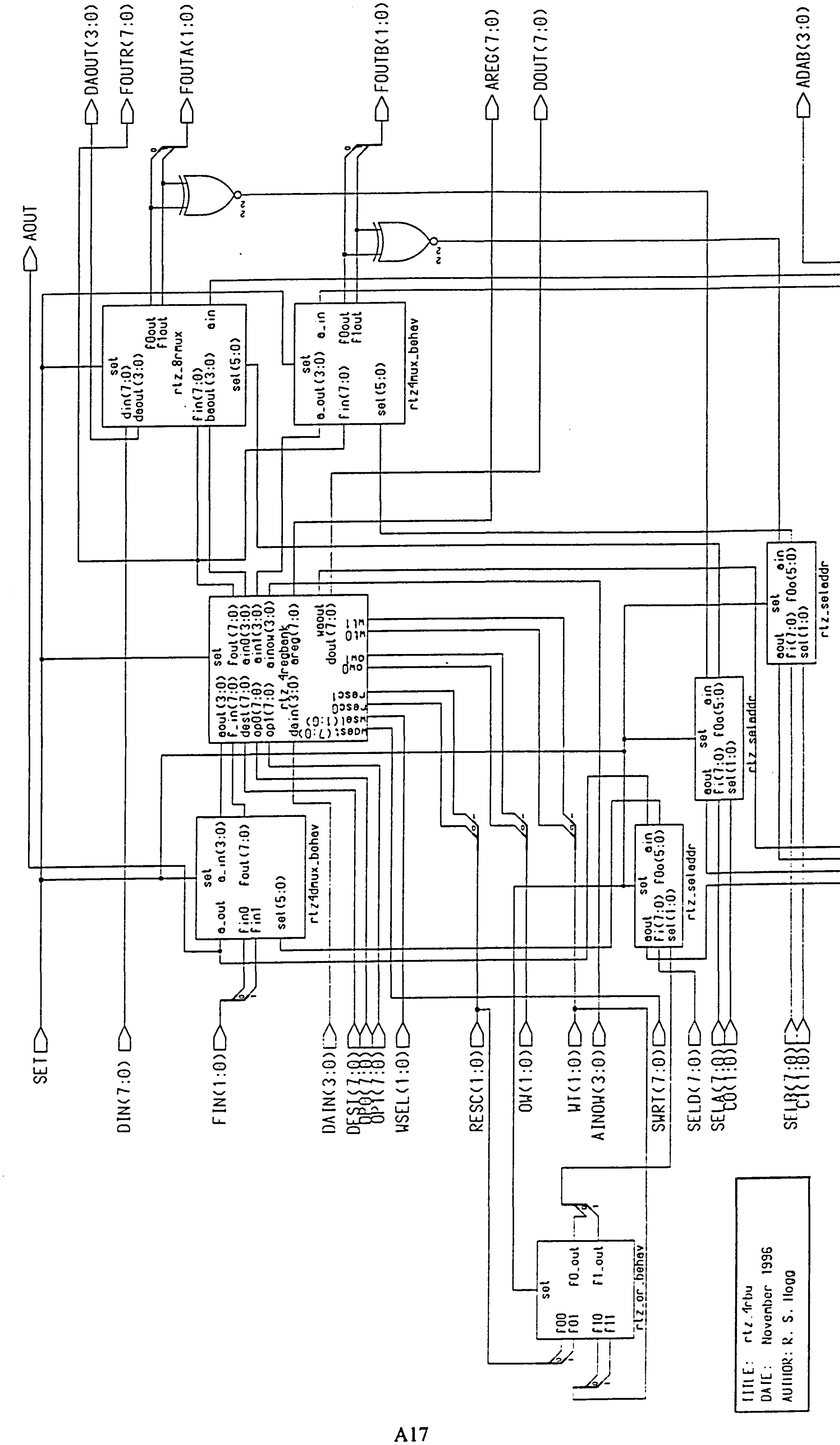
TITLE: rlz_rbu&control
DATE: November 1996
AUTHOR: R. S. Hogg

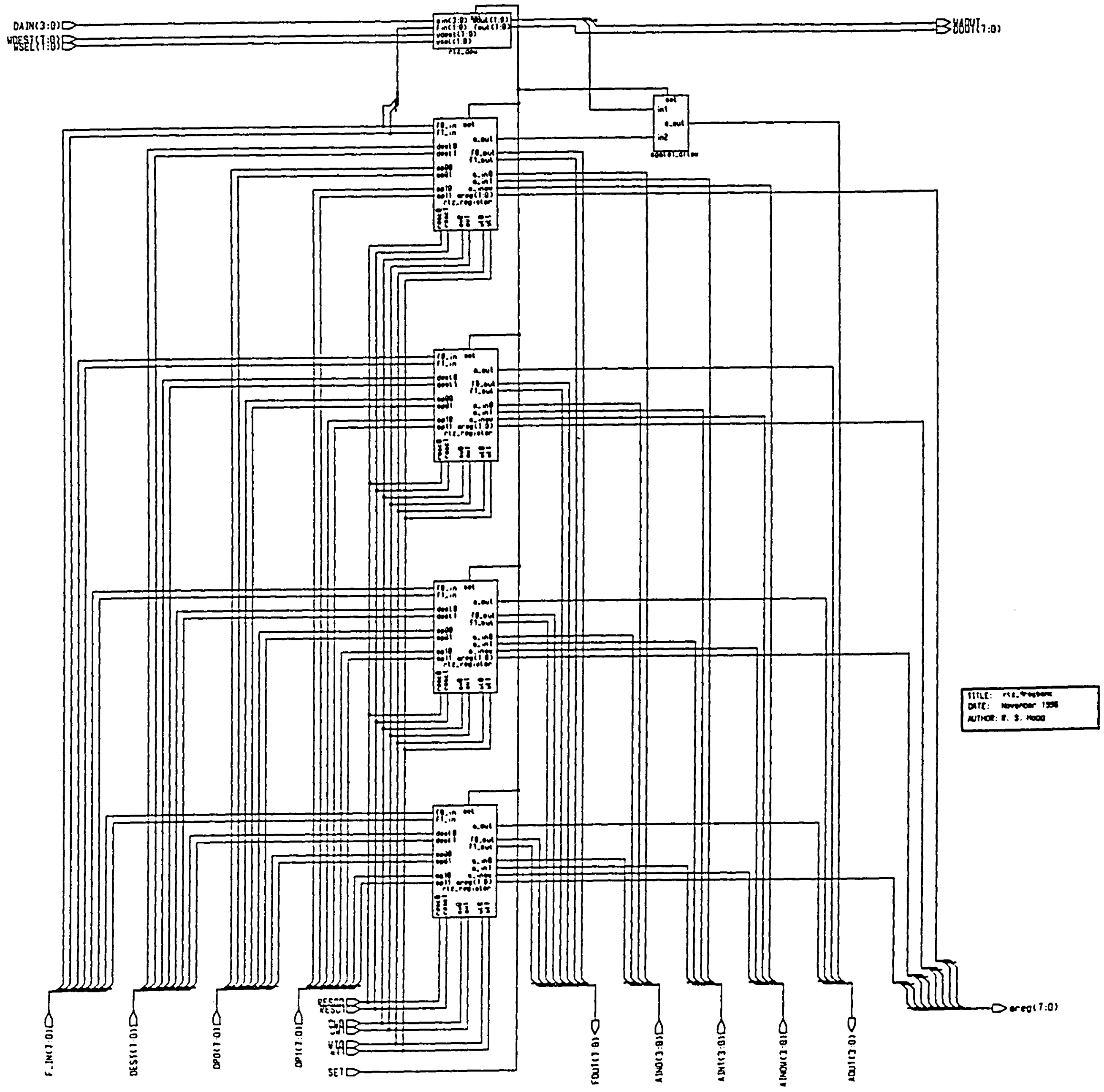

A_OUT
FOUT(1:0)
RESC(1:0)
RESD(1:0)
TITLE: rtz_rcu
DATE: November 1996
AUTHOR: R. S. Hogg
rtz2dmux1_behav
set
a_in(1:0)
a_out
fin0
fin1
f0out(1:0)
f1out(1:0)
sel(1:0)
rtz_eowdet
pw0
set
pw1
C0
C1
sw0
sw1
D0
D1
a_out
a_in
cgate1_dflow
set
in1
c_out
in2
cgate1_dflow
set
in1
c_out
in2
rtz_bitappend0
set
a_in
a_out
f0_in
f1_in
f0_out
f1_out
sel0
sel1
rtz2dmux1_behav
set
a_in(1:0)
a_out
fin0
fin1
f0out(1:0)
f1out(1:0)
sel(1:0)
SET
A_IN
FIN(1:0)
SEL(1:0)
OWD(1:0)
CD

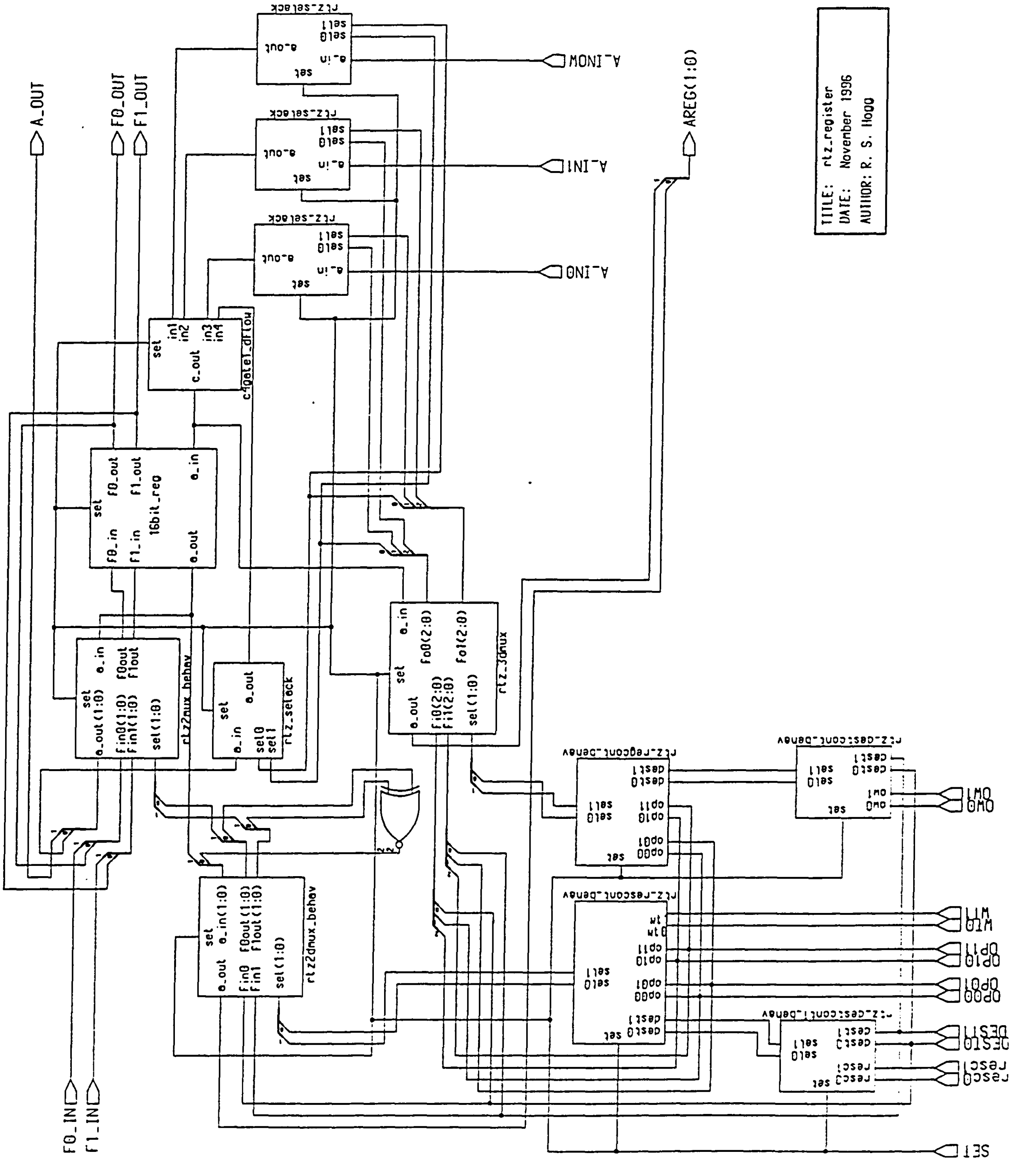


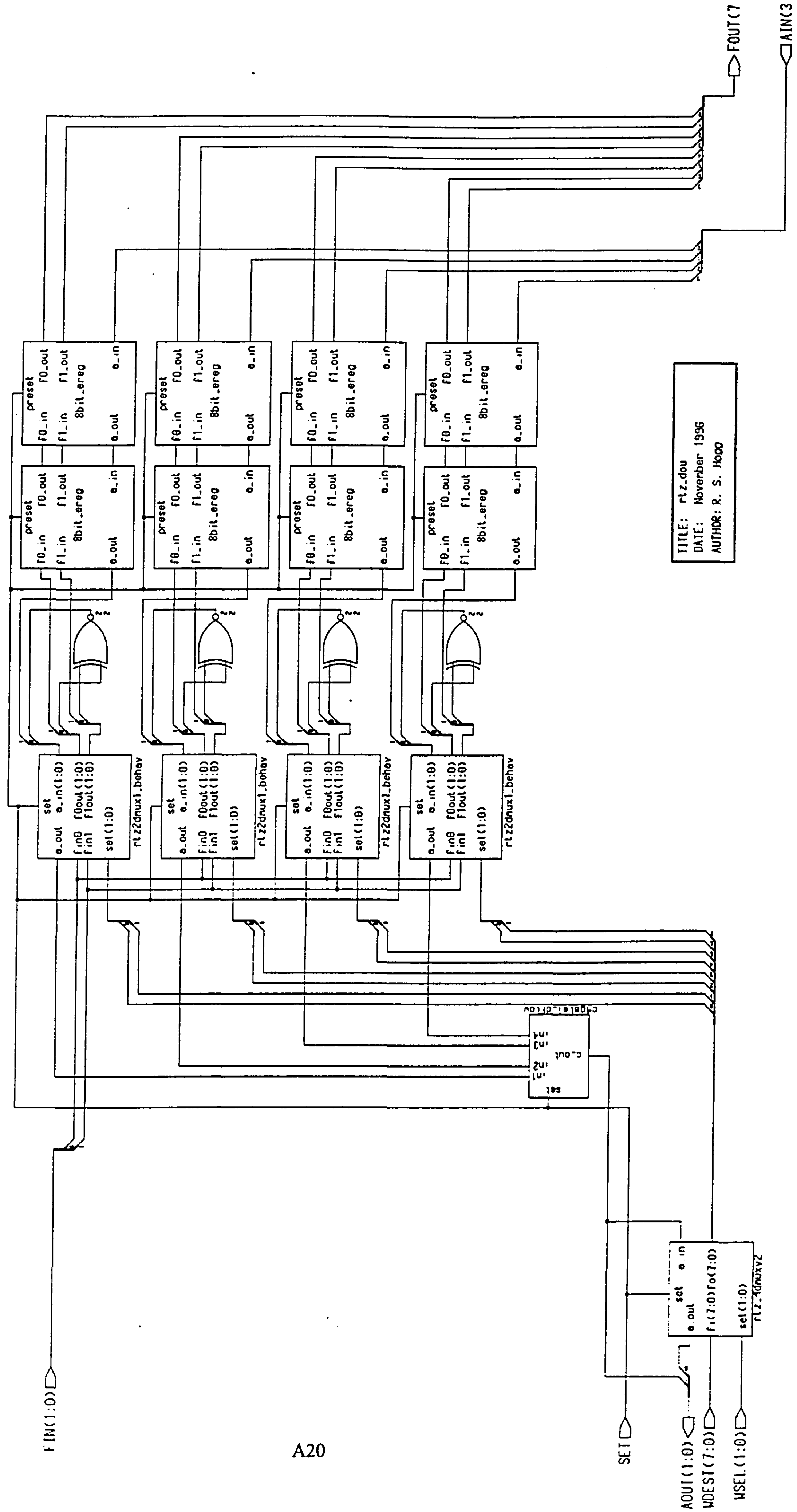
TITLE: rtz_owu
DATE: November 1996
AUTHOR: R. S. Hogg

TITLE: rtz_psu
DATE: November 1996
AUTHOR: R. S. Hogg
P0(1:0)
NUL2
P1(1:0)
NUL3
RES(1:0)
OW(1:0)
A0(1:0)
NUL0
C0(1:0)
C1(1:0)
NUL1
OMD(1:0)
RESD(1:0)
str
f0_inf0_out
f1_inf1_out
a_in a_out
$schematic
str
f0_inf0_out
f1_inf1_out
a_in a_out
$schematic
str
f0_outf0_in
f1_outf1_in
a_out a_in
$schematic
str
f0_outf0_in
f1_outf1_in
a_out a_in
$schematic
owd1
owd0
resd1
resd0
sw1
sw0
pw1
pw0
sel1
sel0
sel
rtz_opmuxcont_behav
owd1
owd0
resd1
resd0
sw1
sw0
pw1
pw0
sel1
sel0
sel
rtz_opmuxcont_behav
sel
f0inf0out
f1inf1out
rtzinbitcon_behav
sel
f0inf0out
f1inf1out
rtzinbitcon_behav
sel
a_out a_in
f0_in f0_out
f1_in f1_out
sel0
sel1
rtz_bitappend
sel
a_out a_in
f0_in f0_out
f1_in f1_out
sel0
sel1
rtz_bitappend
AINN
SL[(1:0)]
\IN(1:0)
A(1:0)
)PB(1:0)
SOP1(1:0)

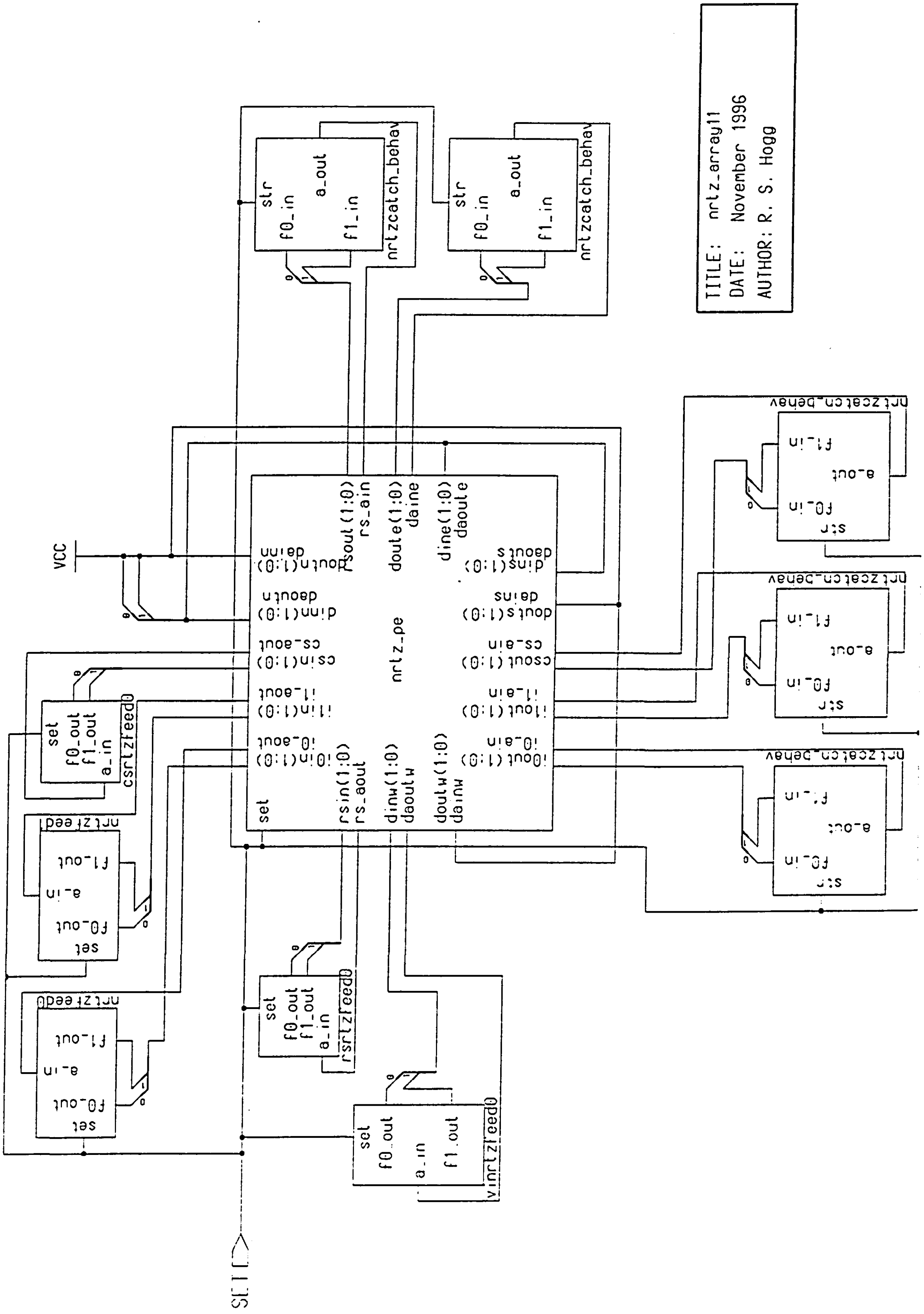

DAOUT(3:0)
FOUTR(7:0)
FOUTA(1:0)
FOUTB(1:0)
AREG(7:0)
DOUT(7:0)
ADAB(3:0)
AOUT
set
din(7:0)
daout(3:0)
fin(7:0)
baout(3:0)
f0out
f1out
ain
sel(5:0)
rtz_8rmux
a_out(3:0)
a_in
rtz4mux_behav
rtz_4regbank
aout(3:0)
f_in(7:0)
dest(7:0)
op0(7:0)
op1(7:0)
dain(3:0)
fout(7:0)
ain0(3:0)
ain1(3:0)
ainow(3:0)
areg(7:0)
dout(7:0)
woout
resc1
resc0
wsel(1:0)
wdest(7:0)
a_out
fin0
fin1
a_in(3:0)
fout(7:0)
rtz4dmux_behav
aout
fi(7:0)
f0a(5:0)
sel(1:0)
rtz_seladdr
f00
f01
f10
f11
f0_out
f1_out
rtz_or_behav
SET
DIN(7:0)
FIN(1:0)
DAIN(3:0)
DEST(7:0)
OP0(7:0)
OP1(7:0)
WSEL(1:0)
RESC(1:0)
OW(1:0)
WT(1:0)
AINOW(3:0)
SWRT(7:0)
SELD(7:0)
SELA(7:0)
C0(1:0)
SELB(7:0)
C1(1:0)
AIN(1:0)
TITLE: rtz_4rbu
DATE: November 1996
AUTHOR: R. S. Hogg


DAIN(3:0)
WDEST(1:0)
WSEL(1:0)
ain(3:0)
fin(1:0)
wdest(1:0)
wsel(1:0)
aout(1:0)
fout(1:0)
rtz_daw
WAOUT
DOUT(7:0)
set
in1
in2
a_out
f0_in
f1_in
dest0
dest1
op00
op01
op10
op11
f0_out
f1_out
a_in0
a_in1
a_new
areg(1:0)
rtz_register
TITLE: rtz_fregbank
DATE: November 1996
AUTHOR: R. S. Hood
F_IN(7:0)
DEST(1:0)
OP0(7:0)
OP1(7:0)
RESET
SET
FOUT(1:0)
AIN0(3:0)
AIN1(3:0)
AINOW(3:0)
AOUT(3:0)
areg(7:0)

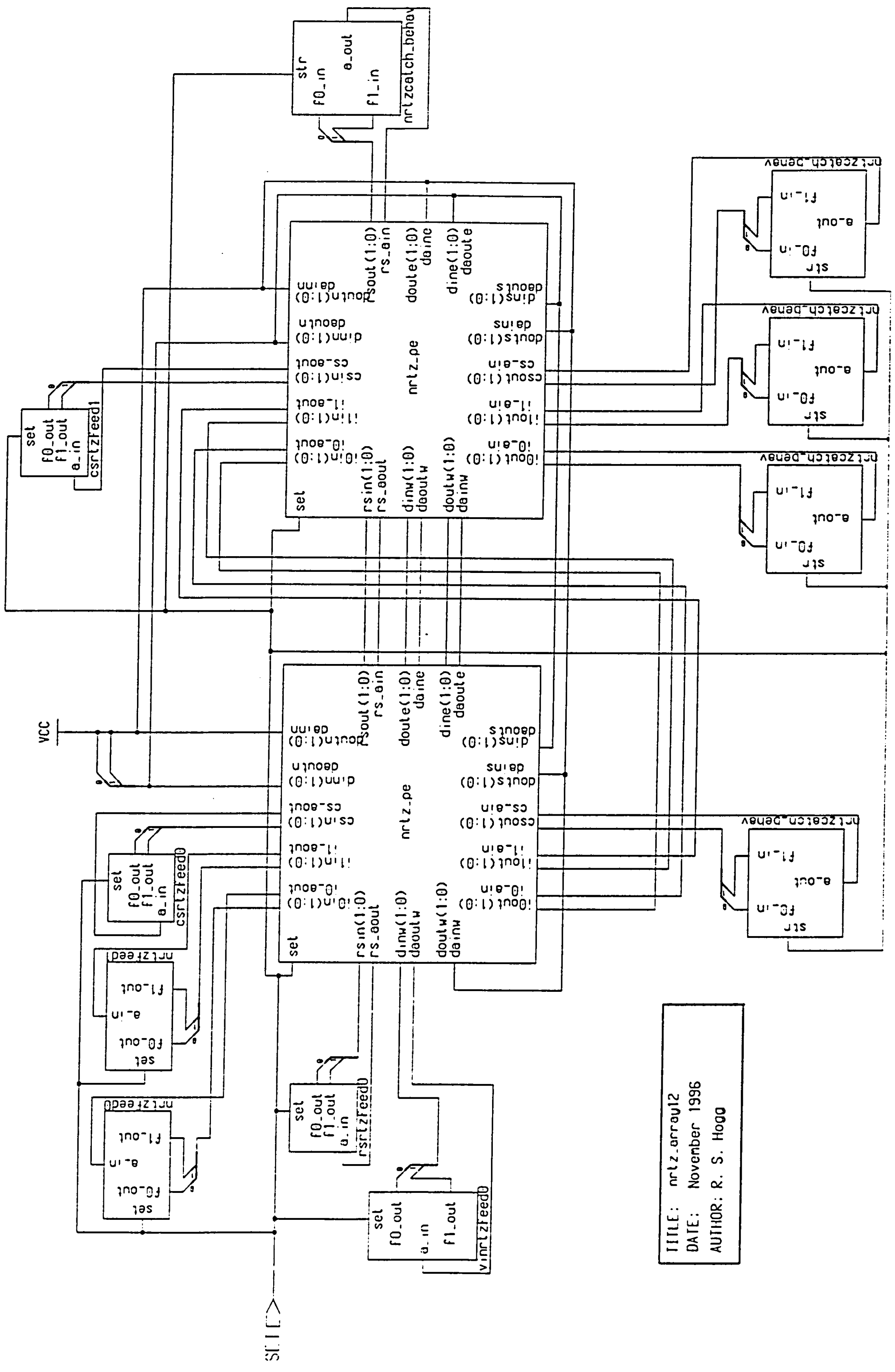

TITLE: rtz_register
DATE: November 1996
AUTHOR: R. S. Hogg
A_OUT
F0_OUT
F1_OUT
A_INOW
A_IN1
A_IN0
AREG(1:0)
rtz_selack
rtz_selack
rtz_selack
rtz_selack
c4gatel_dflow
16bit_reg
rtz2mux_behav
rtz_3dmux
rtz2dmux_behav
rtz_regcont_behav
rtz_rescont_behav
rtz_des:cont_behav
rtz_des:cont1_behav
OW1
OW0
WT1
WT0
OP11
OP10
OP01
OP00
DEST1
DEST0
resc1
resc0
SET
F0_IN
F1_IN



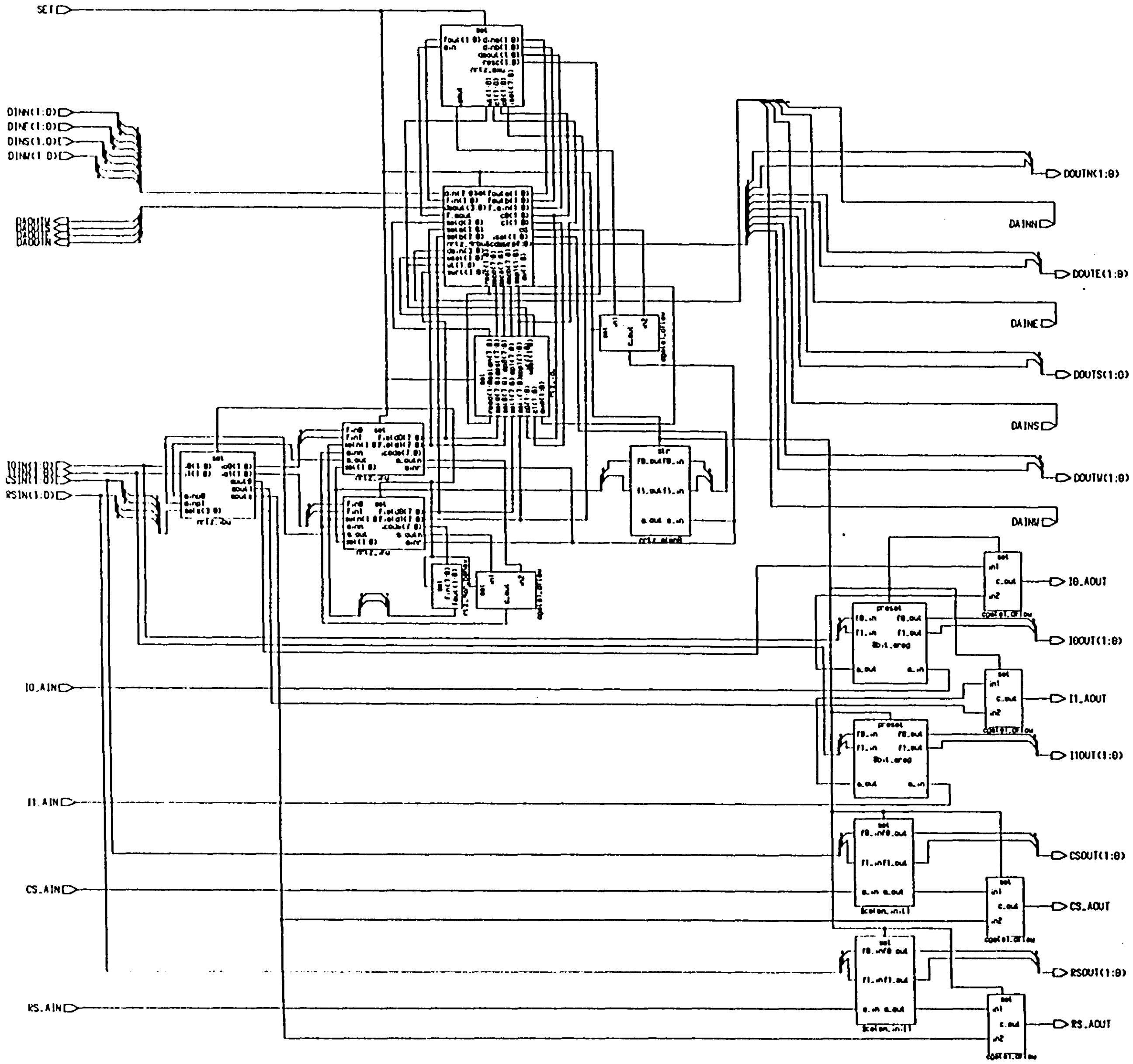
TITLE: rtz.dou
DATE: November 1996
AUTHOR: R. S. Hood



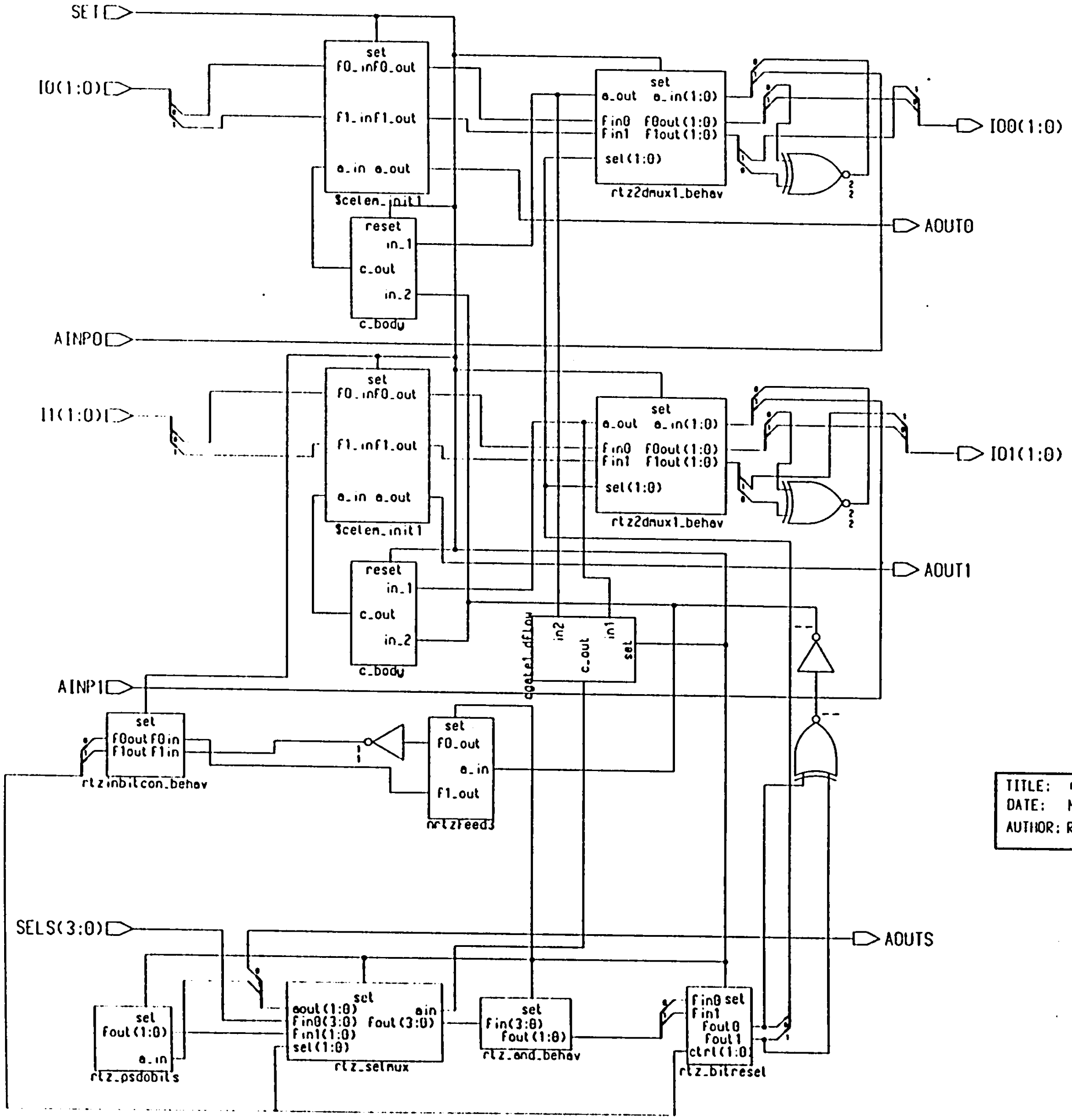
TITLE: nrtz_array11
DATE: November 1996
AUTHOR: R. S. Hogg



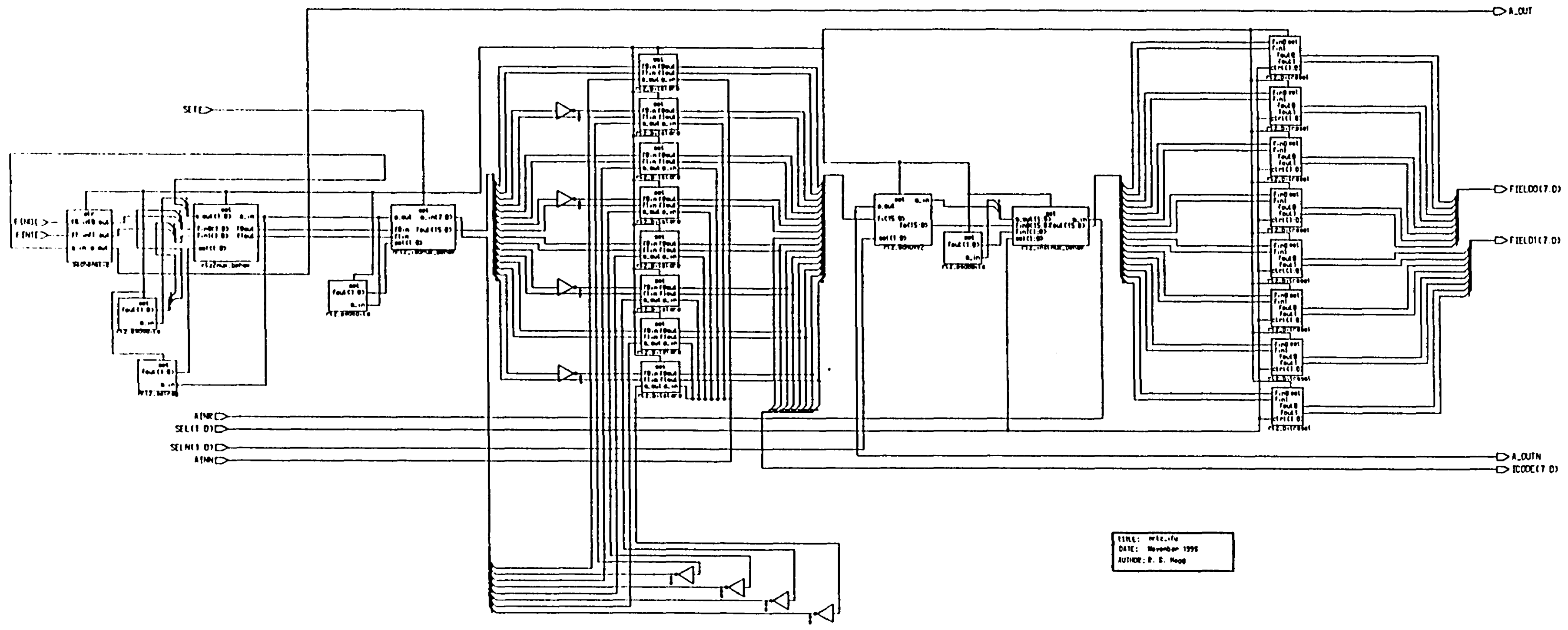
TITLE: nrtz_array12
DATE: November 1996
AUTHOR: R. S. Hogg



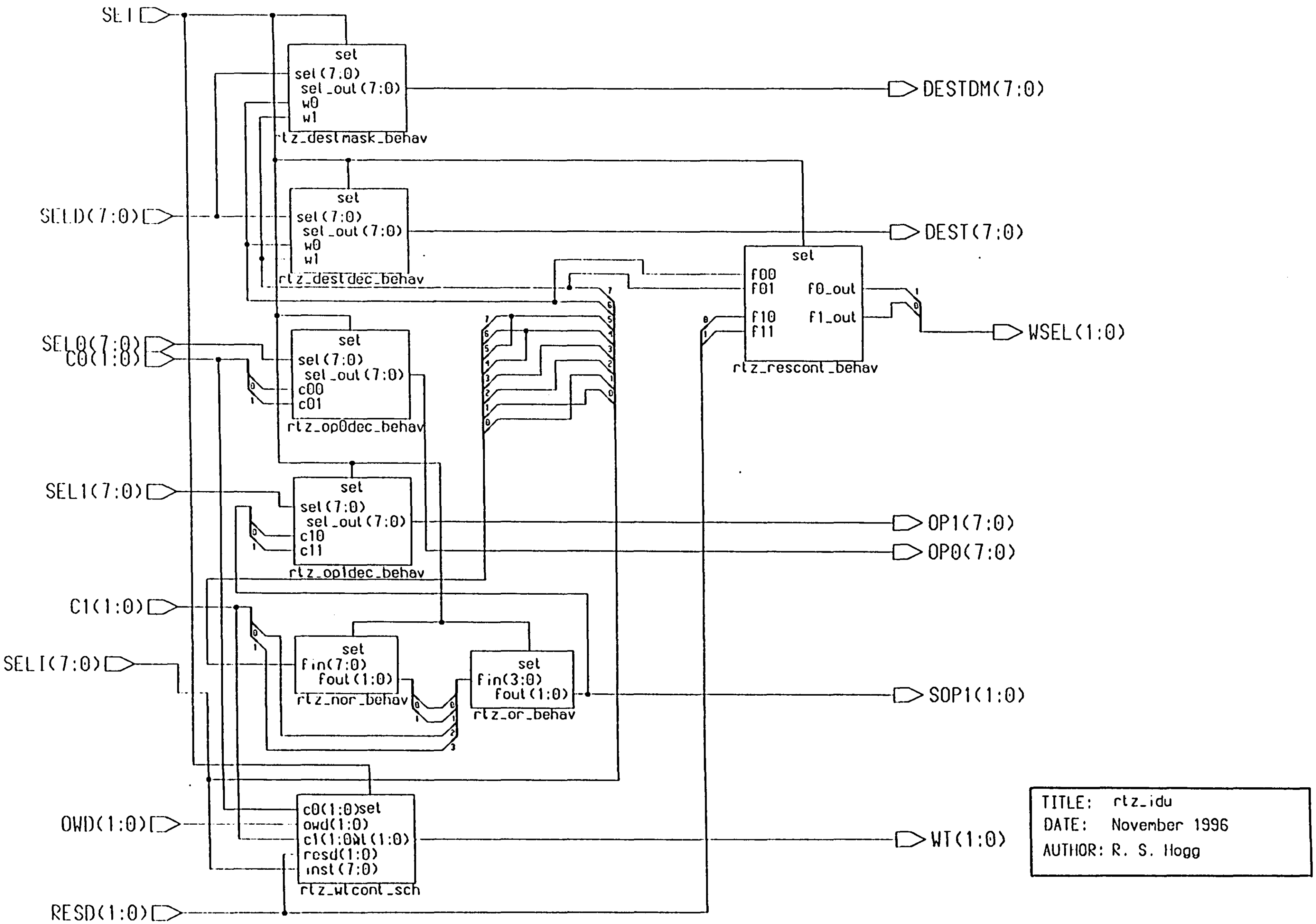
TITLE: rrtz.pe
DATE: November 1996
AUTHOR: R. S. Hogg



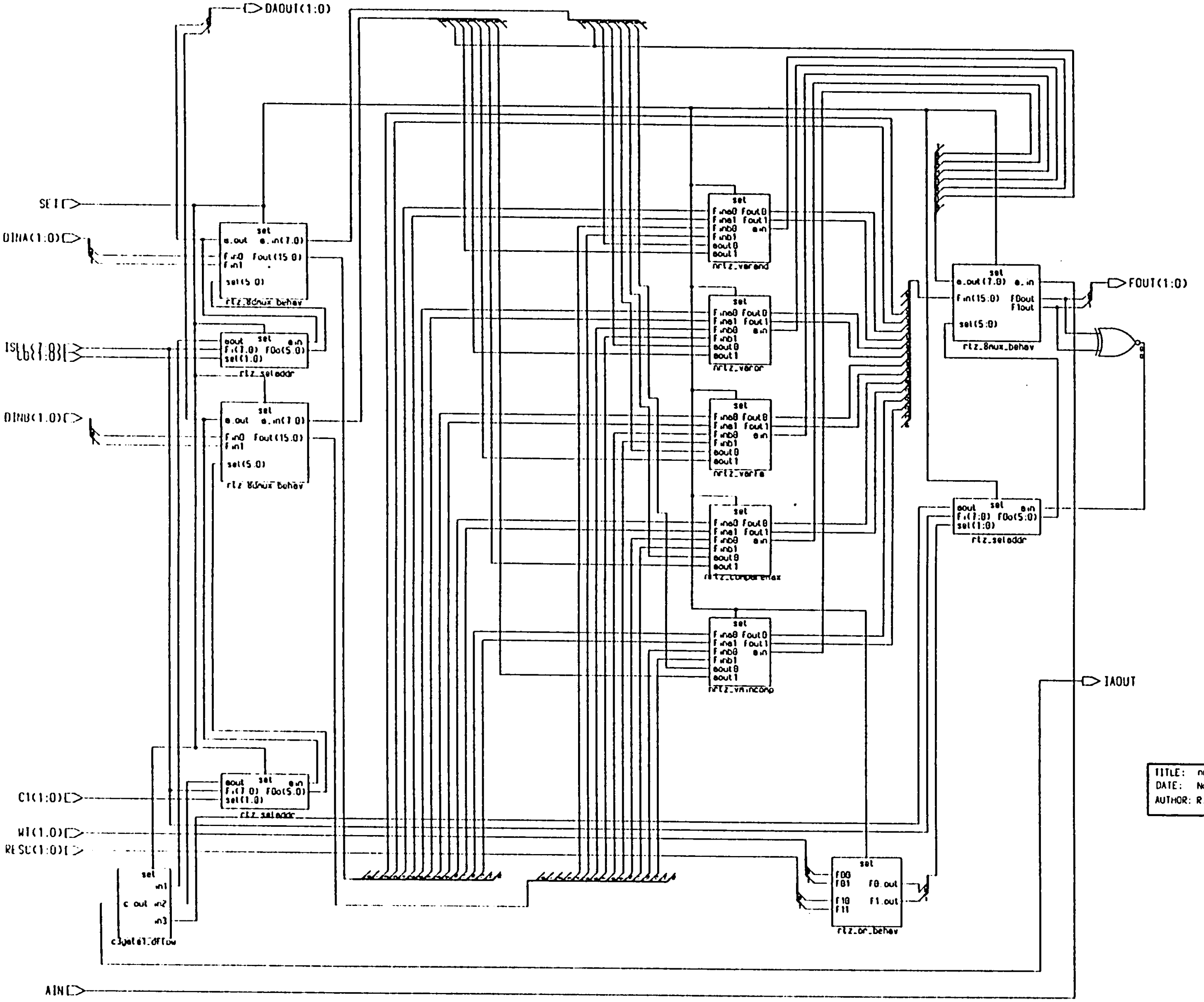
TITLE: nrtz_ibu
DATE: November 1996
AUTHOR: R. S. Hogg



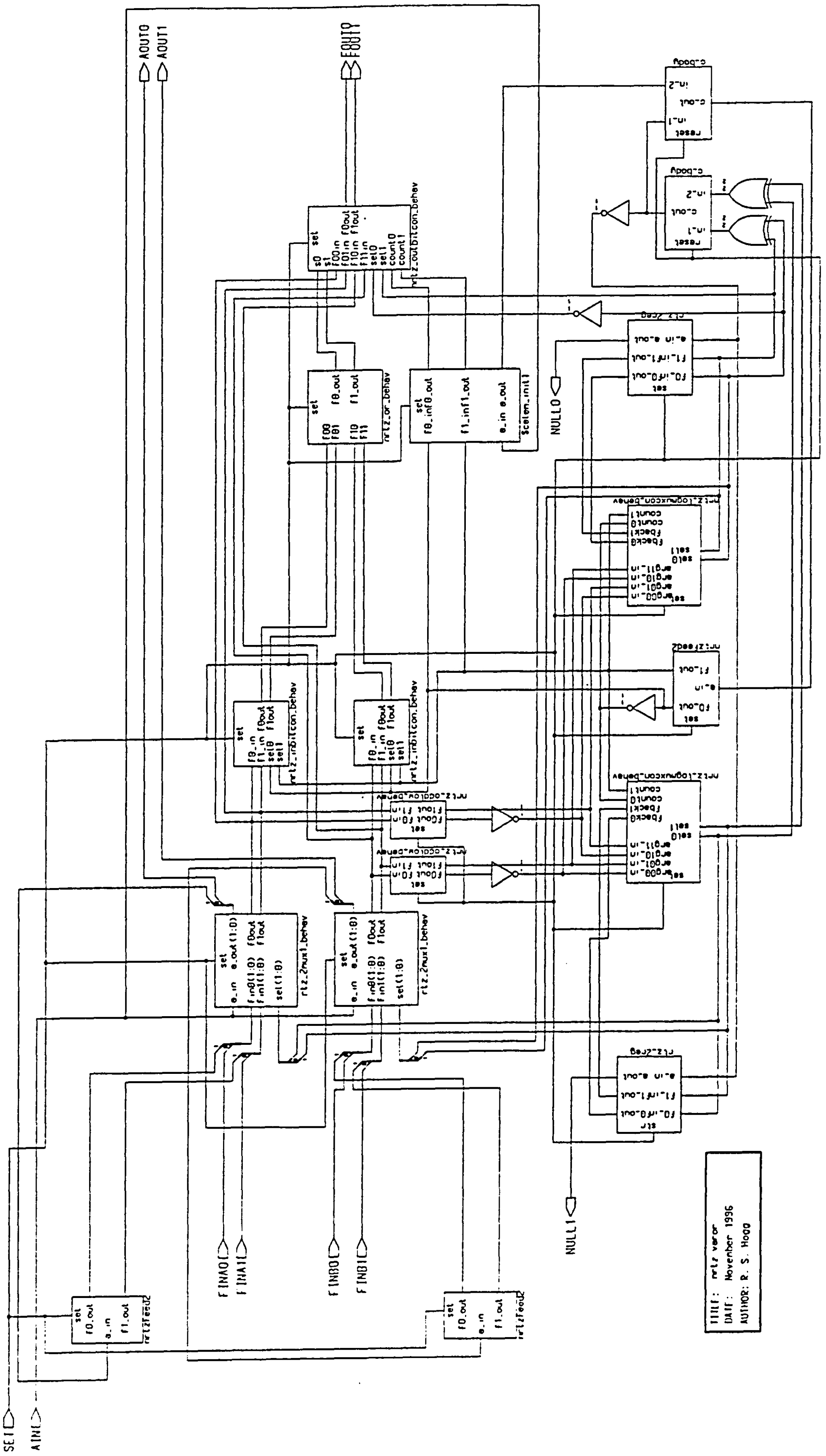
TITLE: mrlz_ifu
DATE: November 1995
AUTHOR: R. S. Hogg

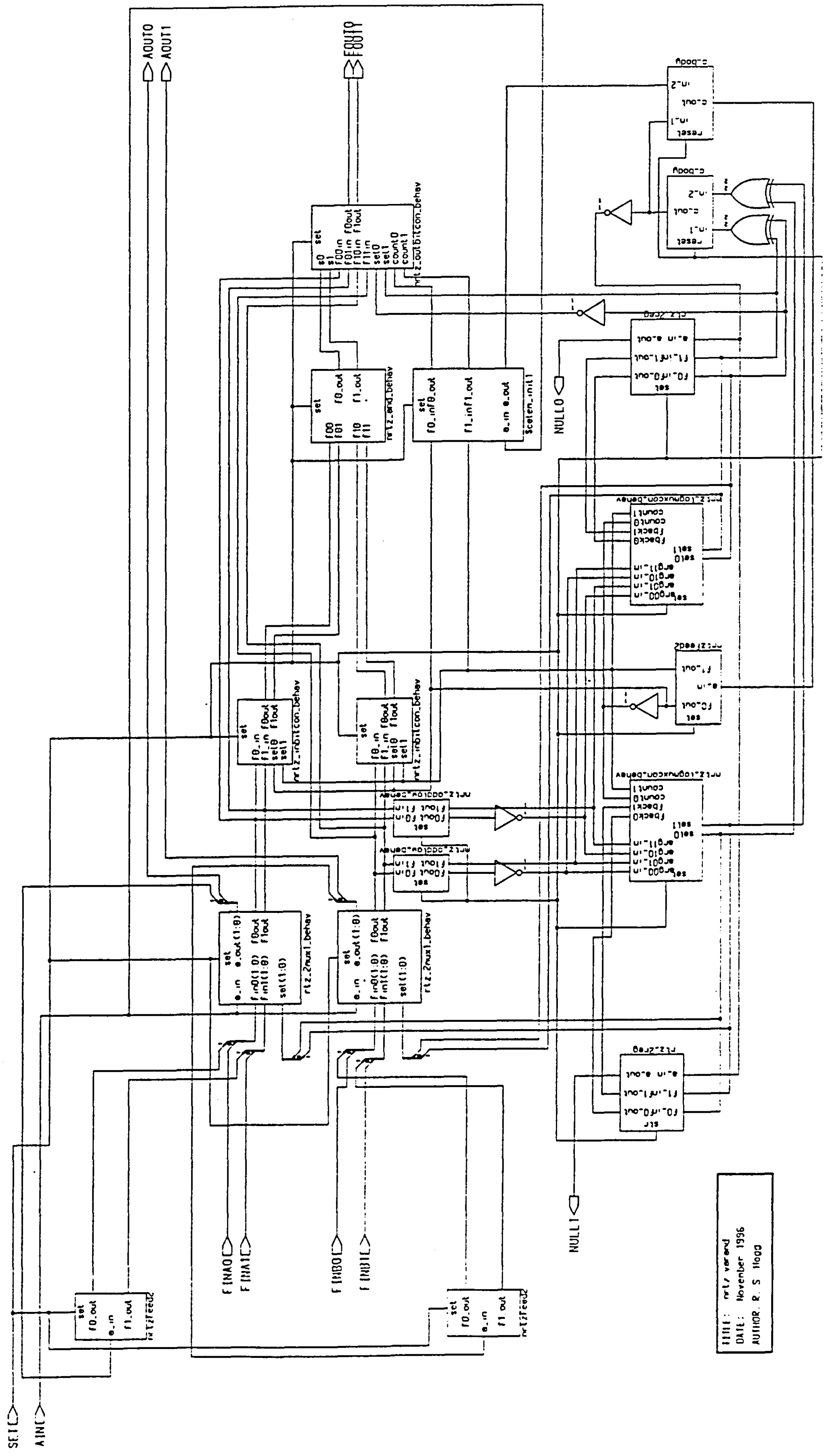


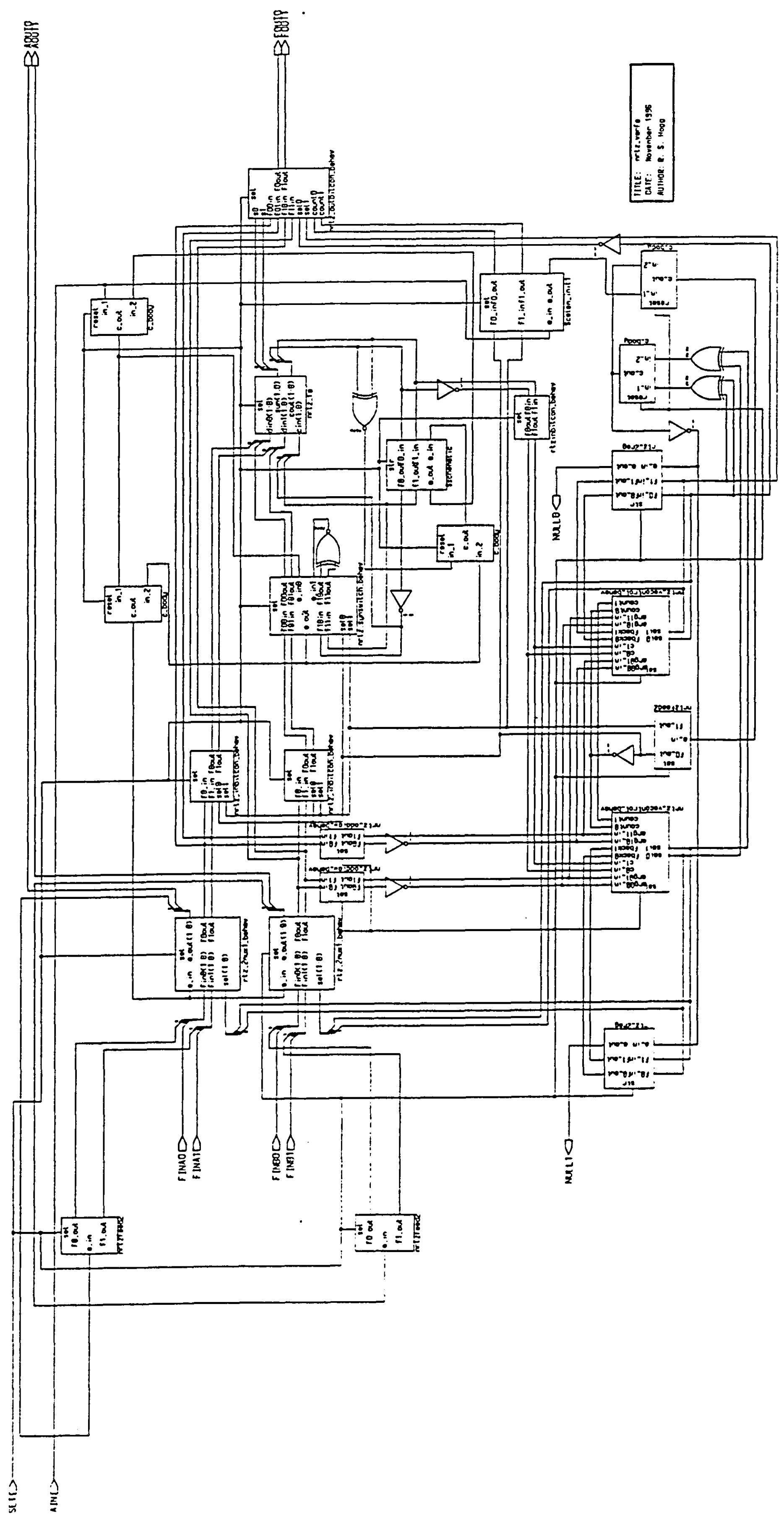
TITLE: rtz_idu
DATE: November 1996
AUTHOR: R. S. Hogg

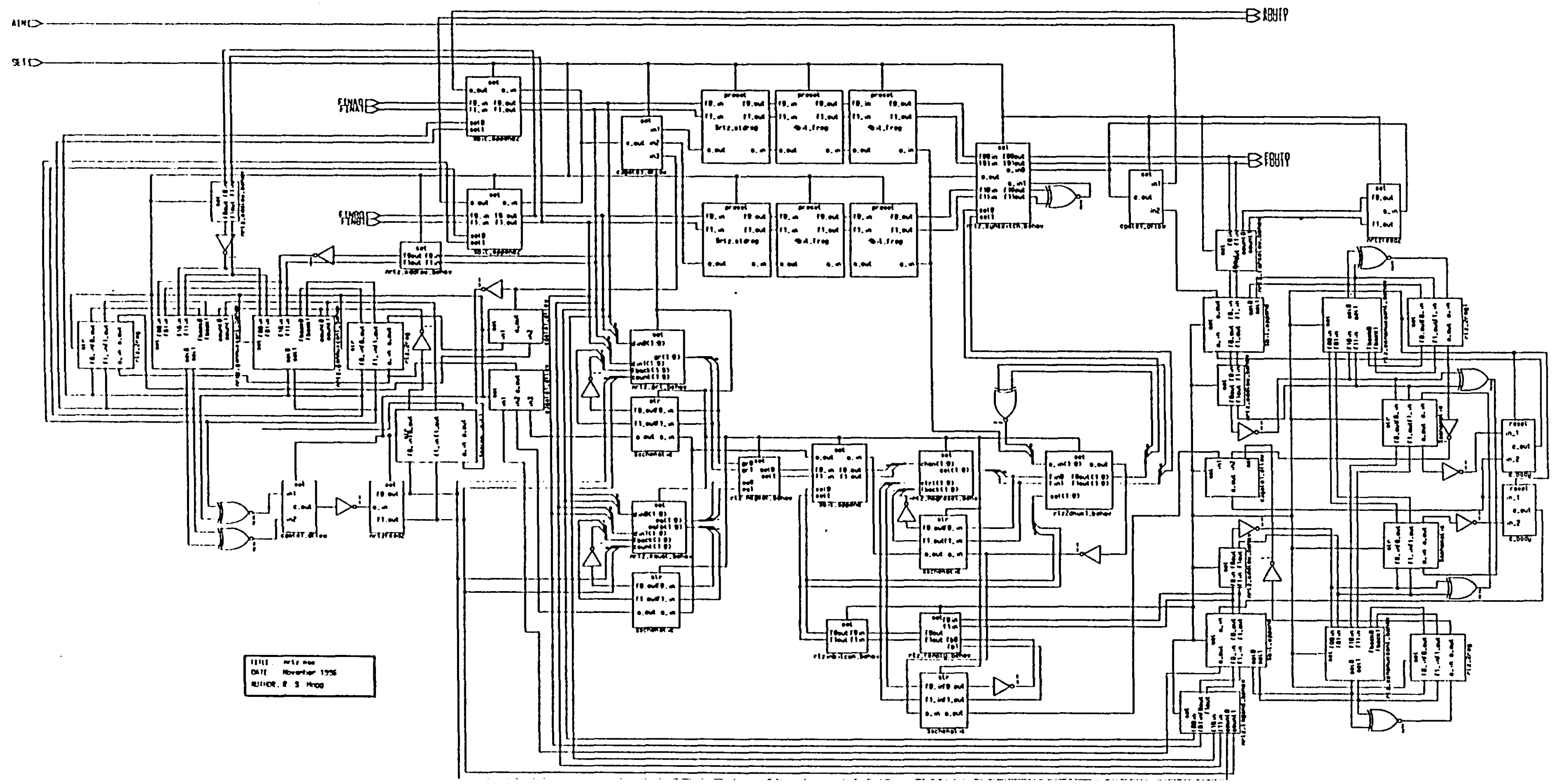


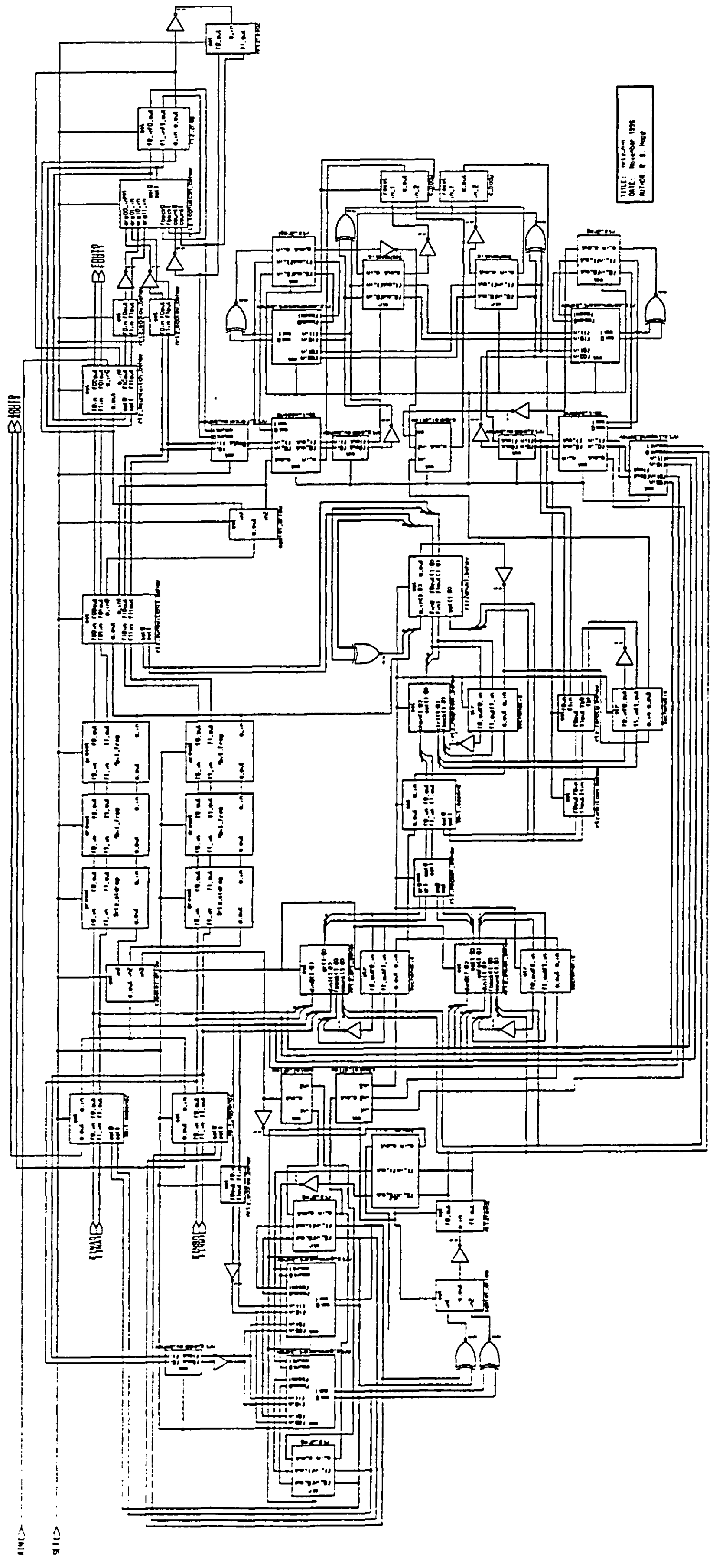
TITLE: nrtz.exu
DATE: November 1996
AUTHOR: R. S. Hogg

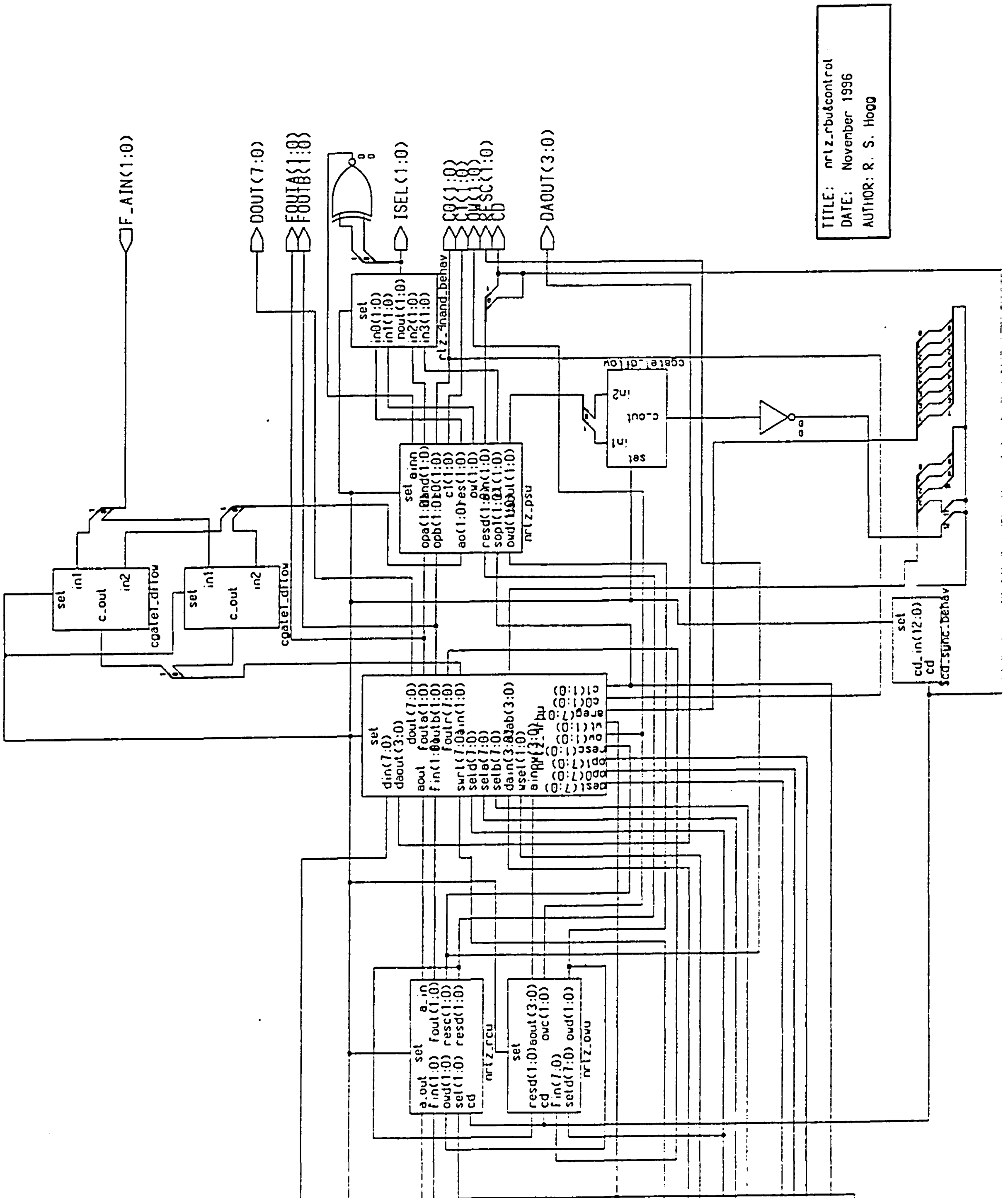

AOUT0
AOUT1
FOUT0
FOUT1
NULL0
NULL1
FINA0
FINA1
FINB0
FINB1
SEI
AIN
nrtz_outbitcon_behav
s0 s1 f00in f01in f10in f11in sel0 sel1 count0 count1 f0out f1out set
nrtz_or_behav
f00 f01 f10 f11 set f0_out f1_out
Scelen_init1
set f0_in f0_out f1_in f1_out e_in e_out
c_body
reset in_1 in_2 c_out
rtz_zeng
nrtz_logmuxcon_behav
sel arg00_in arg01_in arg10_in arg11_in sel0 sel1 fback0 fback1 count0 count1
nrtzfeed2
set f0_out e_in f1_out
nrtz_inbitcon_behav
set f0_in f1_in sel0 sel1 f0out f1out
nrtz_ocdlow_behav
set f0_in f1_in f0out f1out
rtz_2mux1_behav
set e_in e_out(1:0) fin0(1:0) fin1(1:0) sel(1:0) f0out f1out
rtz_zng
str f0_in f0_out f1_in f1_out e_in e_out
TITLE: nrtz veror
DATE: November 1996
AUTHOR: R. S. Hogg


AOUT0
AOUT1
FOUT0
FOUT1
nrtz_outbitcon_behav
nrtz_and_behav
Scelen_init1
nrtz_creg
c_body
NULL0
nrtz_logmuxcon_behav
nrtzfeed2
nrtz_inbitcon_behav
rtz_2mux1_behav
NULL1
FINA0
FINA1
FINB0
FINB1
nrtzfeed1
nrtzfeed2
SET
AINI
TITLE: nrtz vererd
DATE: November 1996
AUTHOR. R. S Hogg

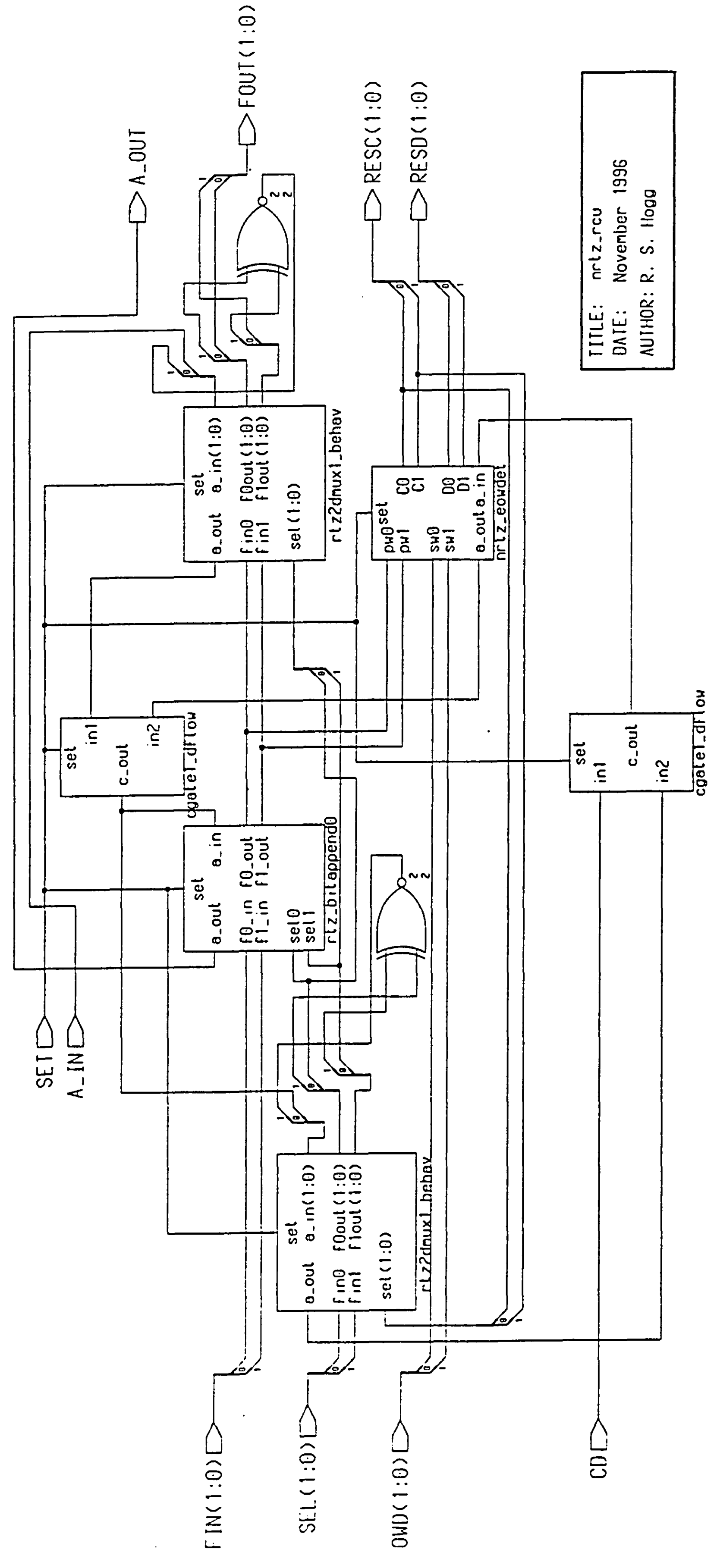

TITLE: nrlz.verfa
DATE: November 1996
AUTHOR: R. S. Hogg
AOUT0
AOUT1
FOUT0
FOUT1
SEL
AIN
FINA0
FINA1
FINB0
FINB1
NULL0
NULL1
c_body
rtz_fa
nrlz_2mux1_behav
nrlz_outbitcon_behav
nrlz_tgswitch_behav
nrlz_inbitcon_behav
reset
in_1
in_2
c_out
sel
e_in
e_out

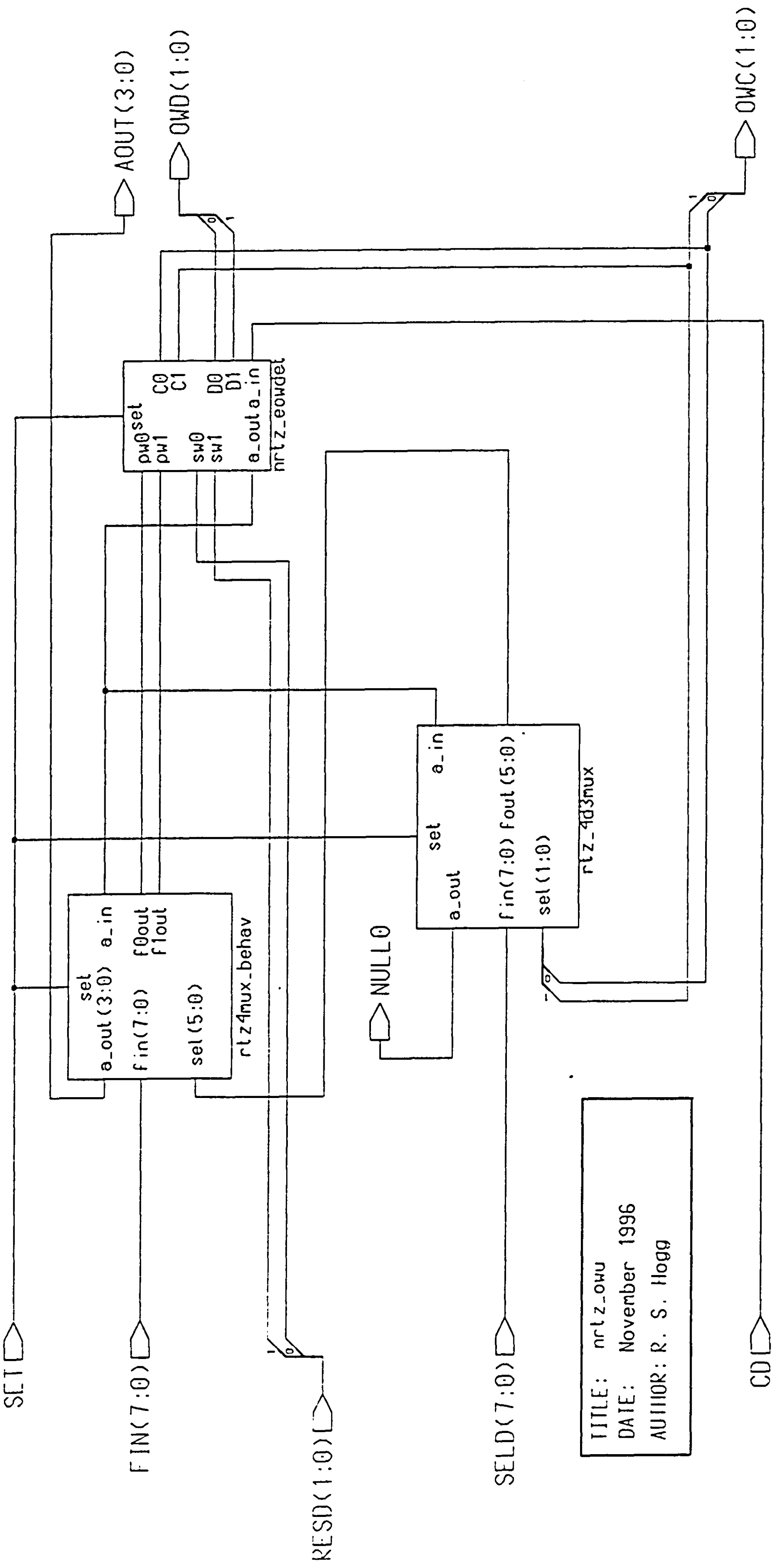

AIN
SEL
AOUTP
EOUTP
FINA0
FINA1
FINB0
FINB1
TITLE . artz.mae
DATE November 1996
AUTHOR. R. S Hogg

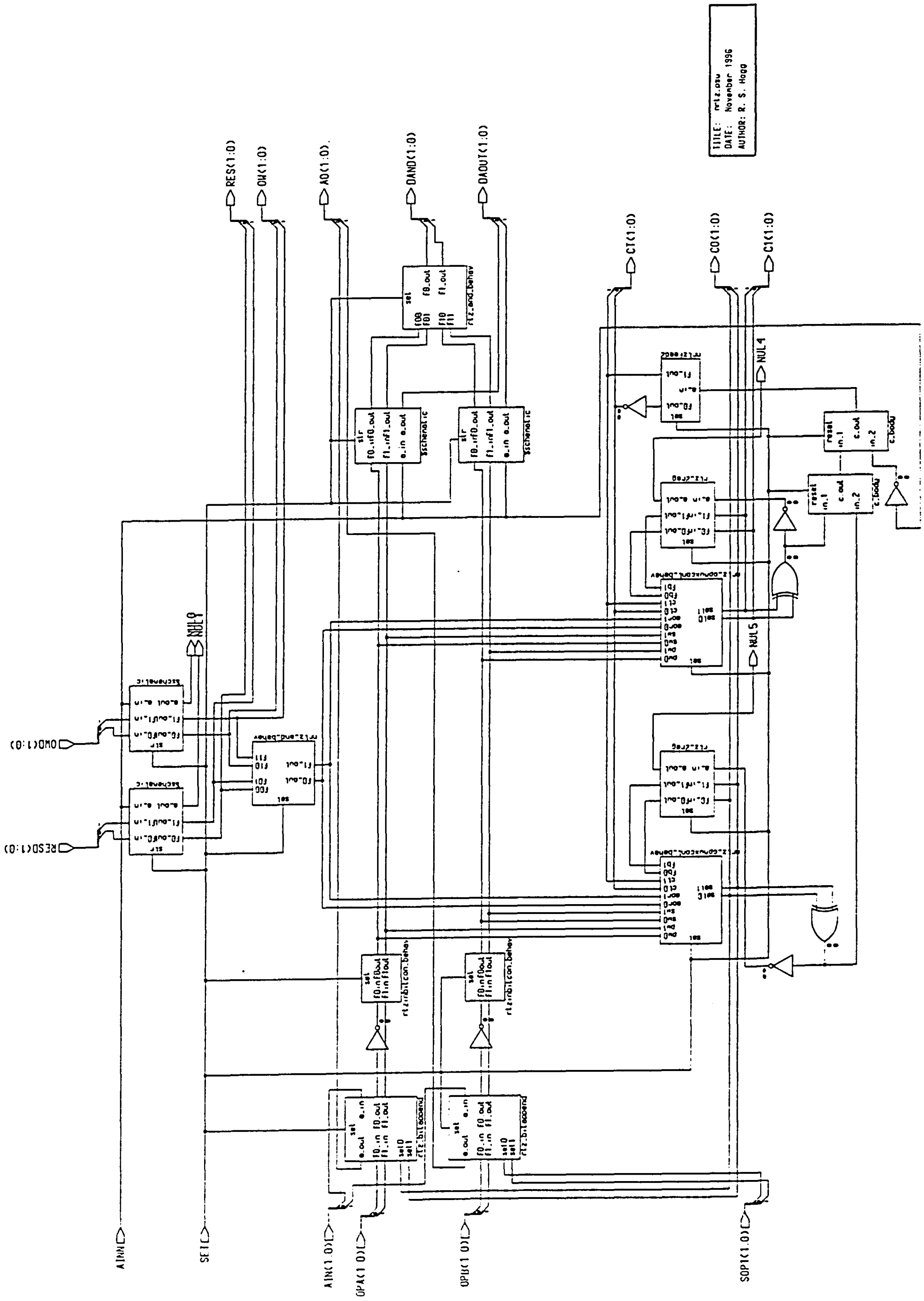

TITLE:
DATE: November 1996
AUTHOR: R S Hogg

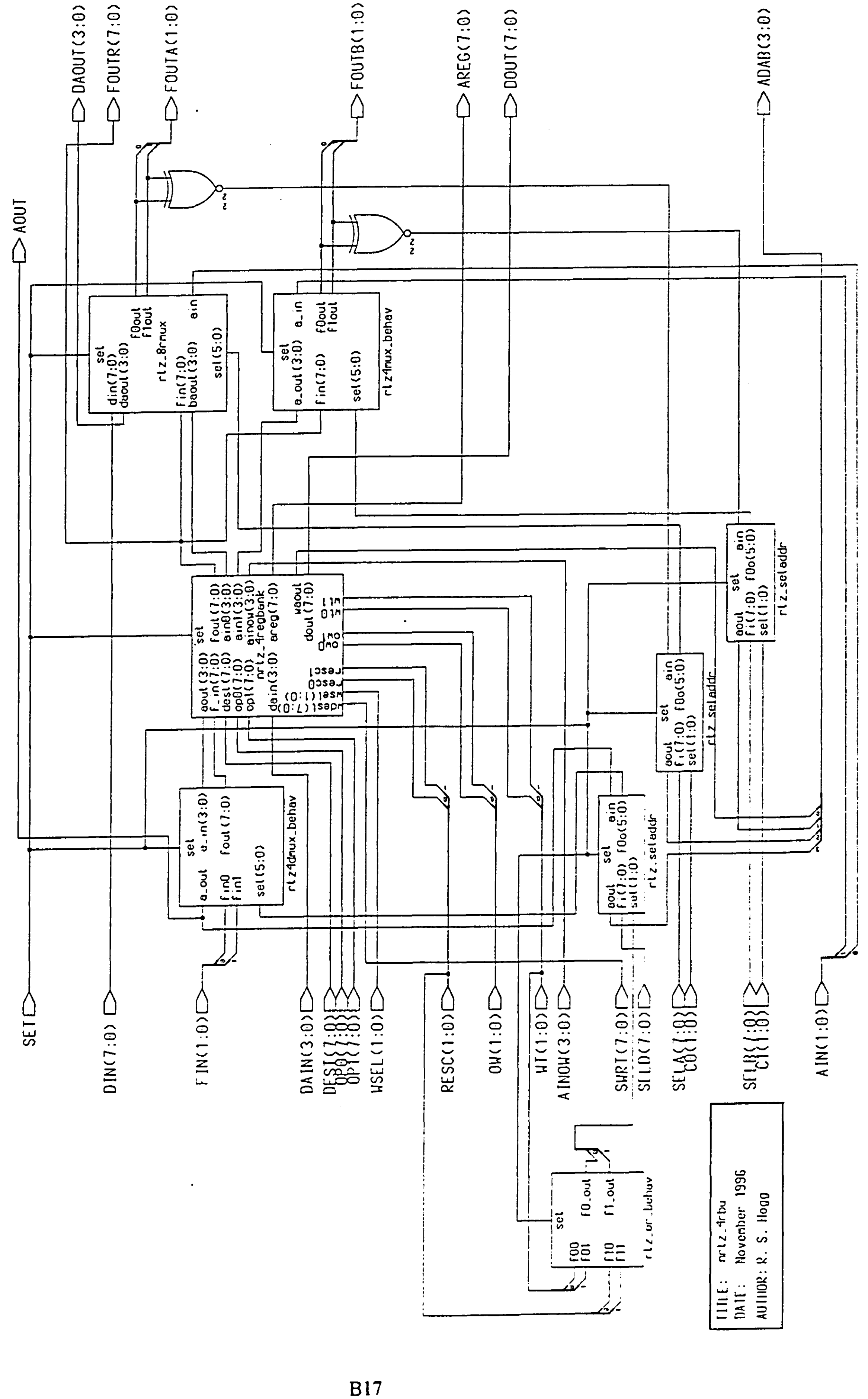


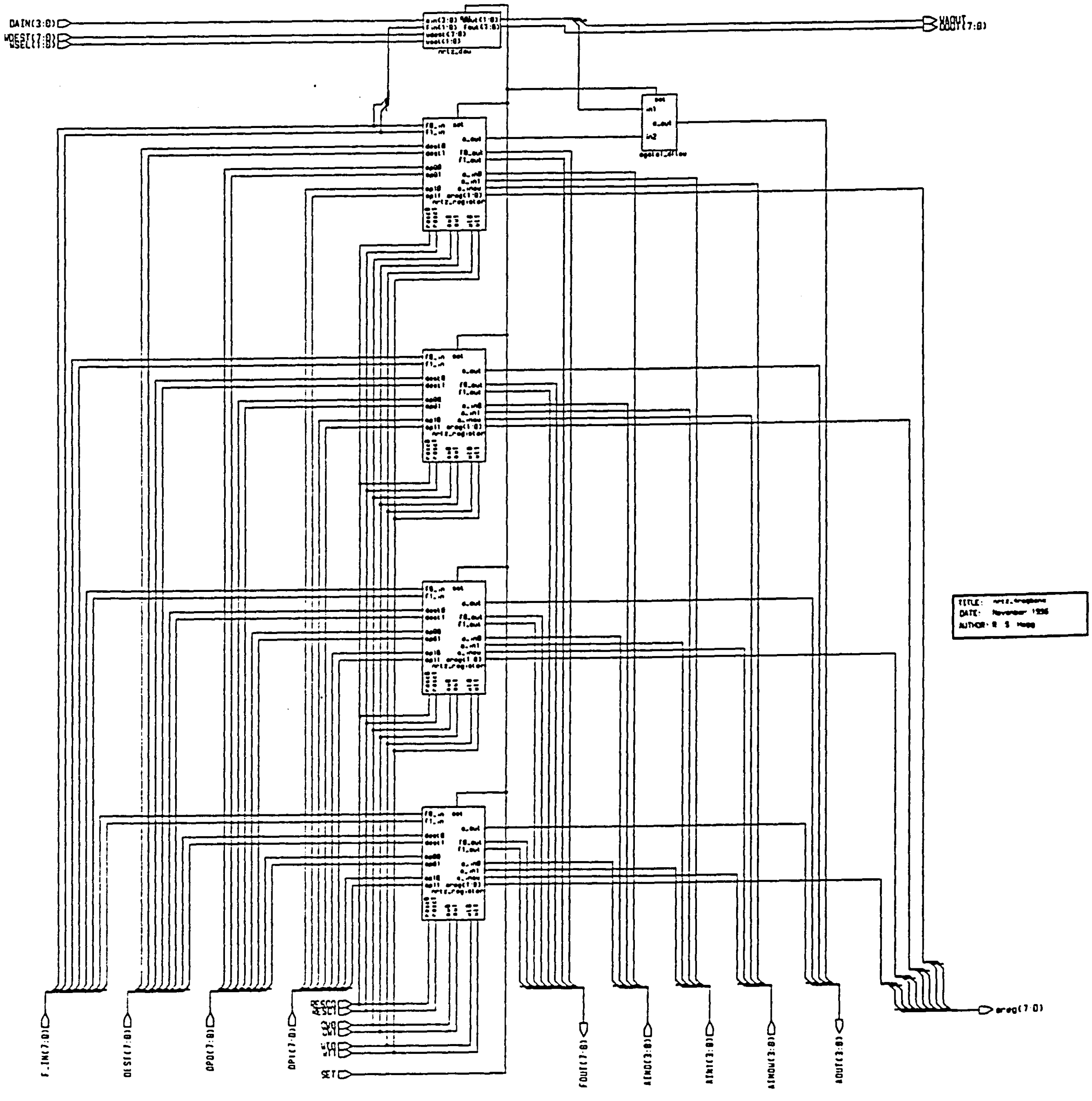
TITLE: nrtz_rbu&control
DATE: November 1996
AUTHOR: R. S. Hogg

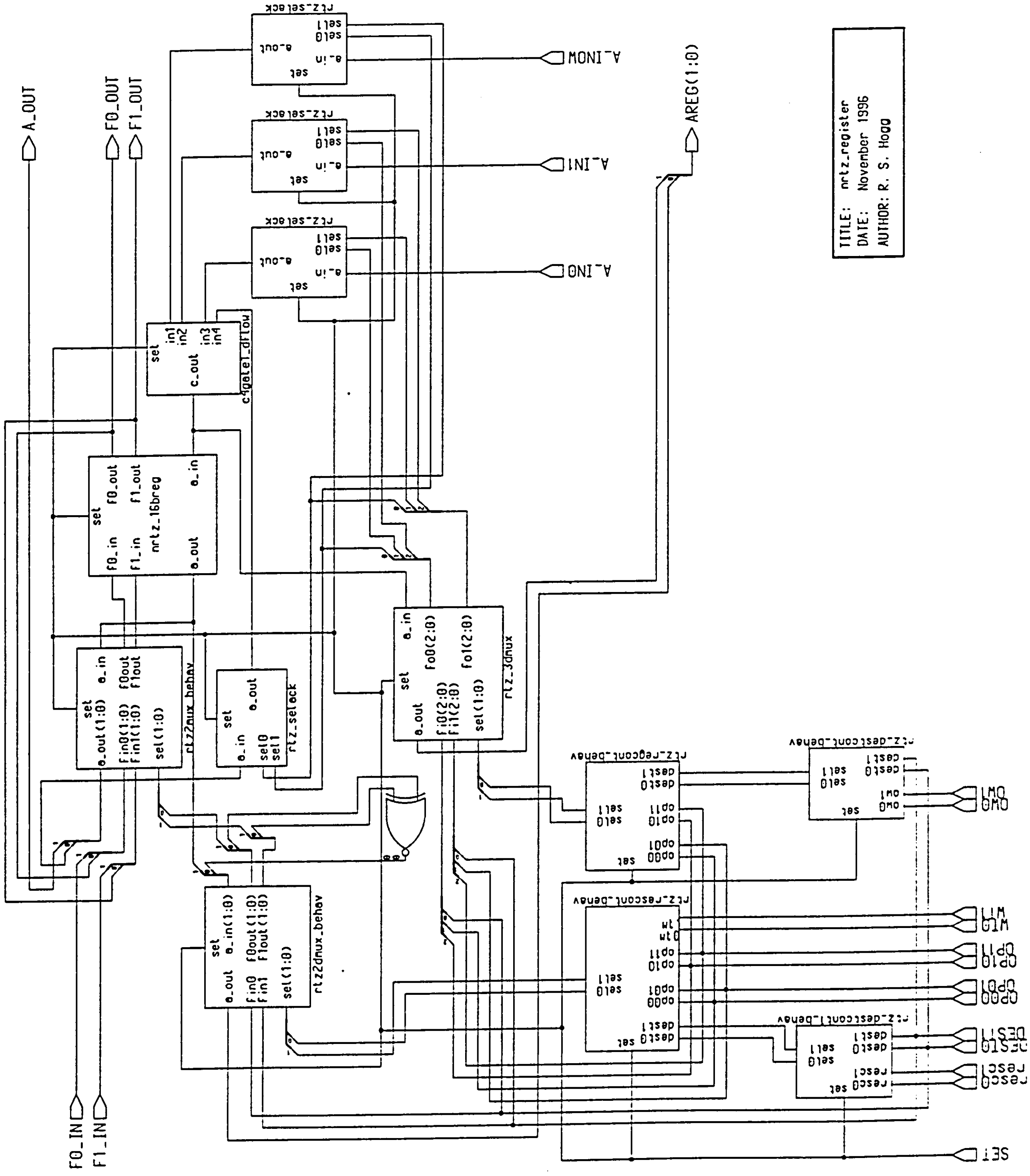

A_OUT
FOUT(1:0)
RESC(1:0)
RESD(1:0)
TITLE: nrtz_rcv
DATE: November 1996
AUTHOR: R. S. Hogg
set
a_out
a_in(1:0)
fin0
fin1
f0out(1:0)
f1out(1:0)
sel(1:0)
rtz2dmux1_behav
pw0
pw1
set
C0
C1
sw0
sw1
D0
D1
a_out
a_in
nrtz_eowdet
set
in1
c_out
in2
cgate1_dflow
set
a_out
a_in
f0_in
f1_in
f0_out
f1_out
sel0
sel1
rtz_bitappend0
set
in1
c_out
in2
cgate1_dflow
set
a_out
a_in(1:0)
fin0
fin1
f0out(1:0)
f1out(1:0)
sel(1:0)
rtz2dmux1_behav
SET
A_IN
FIN(1:0)
SEL(1:0)
OWD(1:0)
CD


AOUT(3:0)
OWD(1:0)
OWC(1:0)
pw0
set
pw1
C0
C1
sw0
sw1
D0
D1
a_out
a_in
nrtz_eowdet
set
a_in
a_out
fin(7:0)
fout(5:0)
sel(1:0)
rtz_4d3mux
NULLO
set
a_out(3:0)
a_in
fin(7:0)
f0out
f1out
sel(5:0)
rtz4mux_behav
SET
FIN(7:0)
RESD(1:0)
SELD(7:0)
CD
TITLE: nrtz_owu
DATE: November 1996
AUTHOR: R. S. Hogg

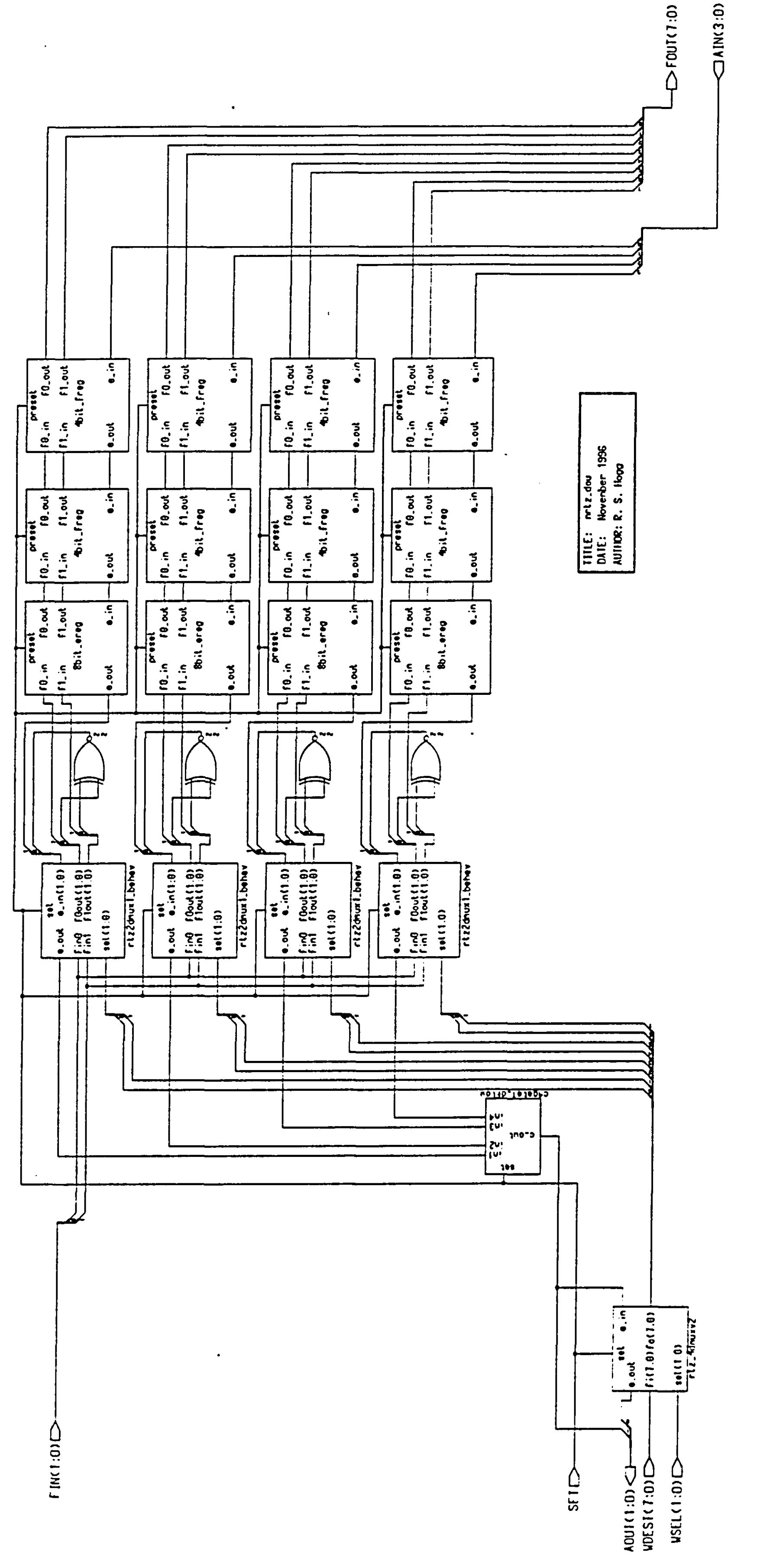

TITLE: nrtz.osu
DATE: November 1996
AUTHOR: R. S. Hogg
RES(1:0)
OK(1:0)
AO(1:0)
DAND(1:0)
DAOUT(1:0)
CT(1:0)
C0(1:0)
C1(1:0)
NUL4
NUL5
NUL0
OMD(1:0)
RESD(1:0)
AINN
SET
AIN(1:0)
OPA(1:0)
OPB(1:0)
SOP(1:0)
rtz_and_behav
$schematic
rtz_creg
rtz_opnxcont_behav
c_body
rtz_rbitcon_behav
rtz_bitopend


DAOUT(3:0)
FOUTR(7:0)
FOUTA(1:0)
FOUTB(1:0)
AREG(7:0)
DOUT(7:0)
ADAB(3:0)
AOUT
SET
DIN(7:0)
FIN(1:0)
DAIN(3:0)
DEST(7:0)
OP0(7:0)
OP1(7:0)
WSEL(1:0)
RESC(1:0)
OW(1:0)
WT(1:0)
AINOW(3:0)
SWRT(7:0)
SELD(7:0)
SELA(7:0)
C0(1:0)
SELB(7:0)
C1(1:0)
AIN(1:0)
set
din(7:0)
daout(3:0)
rlz_8rnux
fin(7:0)
baout(3:0)
sel(5:0)
f0out
f1out
ain
set
a_out(3:0)
a_in
fin(7:0)
f0out
f1out
sel(5:0)
rlz4mux_behav
set
aout(3:0)
f_in(7:0)
dest(7:0)
op0(7:0)
op1(7:0)
fout(7:0)
ain0(3:0)
ain1(3:0)
ainow(3:0)
nrlz_4regbank
areg(7:0)
dain(3:0)
dout(7:0)
woout
wsel(1:0)
desl(7:0)
resc0
resc1
set
a_out
a_in(3:0)
fout(7:0)
fin0
fin1
sel(5:0)
rlz4dmux_behav
set
aout
fi(7:0)
f0o(5:0)
sel(1:0)
ain
rlz_seladdr
set
fi(7:0)
f0o(5:0)
sel(1:0)
ain
rlz_seladdr
set
aout
fi(7:0)
f0o(5:0)
sel(1:0)
ain
rlz_seladdr
set
f00
f01
f10
f11
f0_out
f1_out
rlz_or_behav
TITLE: nrlz_4rbu
DATE: November 1996
AUTHOR: R. S. Hogg


DAIN(3:0)
WDEST(7:0)
WSEL(1:0)
ain(3:0)
fin(1:0)
wdest(7:0)
wsel(1:0)
fout(7:0)
nrtz_dew
WAOUT
DOUT(7:0)
set
in1
a_out
in2
f0_in
f1_in
dest0
dest1
op00
op01
op10
op11
a_out
f0_out
f1_out
a_in0
a_in1
a_inov
areg(1:0)
nrtz_register
TITLE:
DATE: November 1996
AUTHOR: R S Hess
F_IN(7:0)
DEST(7:0)
OP0(7:0)
OP1(7:0)
SET
FOUT(7:0)
AIN0(3:0)
AIN1(3:0)
AINOV(3:0)
AOUT(3:0)
areg(7:0)



TITLE: nrtz_register
DATE: November 1996
AUTHOR: R. S. Hogg


FOUT(7:0)
AIN(3:0)
preset
f0_in
f1_in
f0_out
f1_out
e_out
e_in
4bit_freg
8bit_ereg
set
e_in(1:0)
f0out(1:0)
f1out(1:0)
fin0
fin1
sel(1:0)
rtz2dmux1_behav
c_out
TITLE: nrtz.dow
DATE: November 1996
AUTHOR: R. S. Hogg
sel(1:0)
rtz_4dmux2
FIN(1:0)
SFT
AOUT(1:0)
WDEST(7:0)
WSEL(1:0)

```
HITECTURE c_body OF c_element IS

IN
out <=  '0' AFTER 3ns  WHEN reset = '1' ELSE
        in_1 AFTER 3ns WHEN in_1 = in_2 ELSE
        c_out;
 c_body;
```

```
HITECTURE rtz_and_behav OF rtz_2and IS

IN

lect_input: PROCESS
ARIABLE first_iterate, found: BOOLEAN := false;
ARIABLE i: INTEGER:=0;
ARIABLE fun0, fun1:qsim_state:='0';
ARIABLE next_state : qsim_state:='1';
GIN
F first_iterate = false  AND set='1' THEN
first_iterate := true;
fout(0)<='0';
fout(1)<='0';
LSE
IF (fin(0) XOR fin(1)) /= next_state OR (fin(2) XOR fin(3)) /= next_state  TH

 WAIT UNTIL (fin(0) XOR fin(1)) = next_state AND (fin(2) XOR fin(3)) = next_s
e;
END IF;
CASE fin(3 DOWNTO 0) IS
 --    "3210";
 WHEN  "0101" =>    fun1:='0'; fun0:='1';
 WHEN  "0110" =>    fun1:='0'; fun0:='1';
 WHEN  "1001" =>    fun1:='0'; fun0:='1';
 WHEN  "1010" =>    fun1:='1'; fun0:='0';

 WHEN OTHERS =>       fun1:='0'; fun0:='0';
END CASE;
fout(1)<=fun1 AFTER 6ns; fout(0)<=fun0 AFTER 6ns;
next_state:=NOT(next_state);
ND IF;
D PROCESS select_input;

 rtz_and_behav;
```

```
CHITECTURE nrtz_and_behav OF nrtz_and IS

GIN

rtz_and: PROCESS
TYPE mem6 IS ARRAY (0 TO 3) OF qsim_state;
 ARIABLE input:mem6;
 ARIABLE first_iterate: BOOLEAN := false;
 ARIABLE ftmp0, ftmp1: qsim_state:='0';
 ARIABLE next_state:qsim_state:='1';
EGIN
IF first_iterate = false  OR set='1' THEN
 first_iterate := true;
 f0_out<='0';
 f1_out<='0';
 LSE
 IF (f00 XOR f01) /= next_state OR (f10 XOR f11) /=next_state THEN
  WAIT UNTIL (f00 XOR f01) = next_state AND (f10 XOR f11) =next_state;
 END IF;
 input(0):=f00; input(1):=f01; input(2):=f10; input(3):= f11;
 CASE input(0 TO 3) IS
  WHEN "0000" => ftmp1:='0'; ftmp0:='0';
  WHEN "0011" => ftmp1:='0'; ftmp0:='0';
  WHEN "1100" => ftmp1:='0'; ftmp0:='0';
  WHEN "1111" => ftmp1:='1'; ftmp0:='1';
  WHEN "1010" => ftmp1:='0'; ftmp0:='1';
  WHEN "0110" => ftmp1:='0'; ftmp0:='1';
  WHEN "1001" => ftmp1:='0'; ftmp0:='1';
  WHEN "0101" => ftmp1:='1'; ftmp0:='0';
  WHEN OTHERS   => -- 'do nothing';
 END CASE;
 f0_out<=ftmp0 AFTER 10ns; f1_out<=ftmp1 AFTER 10ns;
 next_state:=NOT(next_state);
 ND IF;
 D PROCESS nrtz_and;

  nrtz_and_behav;
```

```
CHITECTURE rtz_regcont_behav OF rtz_regcont IS

GIN

elect_mux: PROCESS
TYPE mem6 IS ARRAY (0 TO 5) OF qsim_state;
VARIABLE input:mem6;
VARIABLE first_iterate: BOOLEAN := false;
VARIABLE s0, s1: qsim_state:='0';
VARIABLE next_state:qsim_state:='1';
EGIN
IF first_iterate = false  AND set='1' THEN
 first_iterate := true;
 sel0<='0';
 sel1<='0';
ELSE
 IF (op00 XOR op01) /= next_state OR (op10 XOR op11) /=next_state OR (dest0 XO
dest1)/=next_state THEN
  WAIT UNTIL (op00 XOR op01) = next_state AND (op10 XOR op11) =next_state AND
est0 XOR dest1)=next_state;
 END IF;
 input(0):=op00; input(1):=op01; input(2):=op10; input(3):=op11; input(4):=des
; input(5):=dest1;
 CASE input(0 TO 5) IS
  WHEN "101010" => s1:='1'; s0:='0';
  WHEN "101001" => s1:='0'; s0:='1';
  WHEN "100110" => s1:='0'; s0:='1';
  WHEN "100101" => s1:='0'; s0:='1';
  WHEN "011010" => s1:='0'; s0:='1';
  WHEN "011001" => s1:='0'; s0:='1';
  WHEN "010110" => s1:='0'; s0:='1';
  WHEN "010101" => s1:='0'; s0:='1';

  WHEN OTHERS   => s1:='0'; s0:='0';
 END CASE;

 sel0<=s0 AFTER 8ns; sel1<=s1 AFTER 8ns;
 next_state:=NOT(next_state);
 ND IF;
 D PROCESS select_mux;

  rtz_regcont_behav;
```

```
CHITECTURE nrtz_fa_behav OF nrtz_fa IS

GIN

rtz_fadder: PROCESS
TYPE mem6 IS ARRAY (0 TO 5) OF qsim_state;
VARIABLE input:mem6;
VARIABLE first_iterate: BOOLEAN := false;
VARIABLE s0, s1, c0, c1: qsim_state:='0';
VARIABLE next_state:qsim_state:='1';
EGIN
IF first_iterate = false  OR set='1' THEN
 first_iterate := true;
 sum(0)<='0'; sum(1)<='0';
 cout(0)<='0'; cout(1)<='0';
ELSE
 IF (din0(0) XOR din1(0)) /= next_state OR (din0(1) XOR din1(1)) /=next_state
 (cin(0) XOR cin(1))/=next_state THEN
  WAIT UNTIL (din0(0) XOR din1(0)) = next_state AND (din0(1) XOR din1(1)) =nex
state AND (cin(0) XOR cin(1))=next_state;
 END IF;
 input(0):=din0(0); input(1):=din1(0); input(2):=din0(1); input(3):= din1(1);
put(4):= cin(0); input(5):=cin(1);
 CASE input(0 TO 5) IS
  WHEN "000000" => s1:='0'; s0:='0'; c1:='0'; c0:='0';
  WHEN "101010" => s1:='0'; s0:='1'; c1:='0'; c0:='1';
  WHEN "101001" => s1:='1'; s0:='0'; c1:='0'; c0:='1';
  WHEN "100110" => s1:='1'; s0:='0'; c1:='0'; c0:='1';
  WHEN "100101" => s1:='0'; s0:='1'; c1:='1'; c0:='0';
  WHEN "011010" => s1:='1'; s0:='0'; c1:='0'; c0:='1';
  WHEN "011001" => s1:='0'; s0:='1'; c1:='1'; c0:='0';
  WHEN "010110" => s1:='0'; s0:='1'; c1:='1'; c0:='0';
  WHEN "010101" => s1:='1'; s0:='0'; c1:='1'; c0:='0';
  WHEN "110000" => s1:='1'; s0:='1'; c1:='0'; c0:='0';
  WHEN "001100" => s1:='1'; s0:='1'; c1:='0'; c0:='0';
  WHEN "111100" => s1:='0'; s0:='0'; c1:='1'; c0:='1';
  WHEN "000011" => s1:='1'; s0:='1'; c1:='0'; c0:='0';
  WHEN "110011" => s1:='0'; s0:='0'; c1:='1'; c0:='1';
  WHEN "001111" => s1:='0'; s0:='0'; c1:='1'; c0:='1';
  WHEN "111111" => s1:='1'; s0:='1'; c1:='1'; c0:='1';
  WHEN OTHERS   => -- 'do nothing';
 END CASE;
 sum(0)<=s0 AFTER 11ns; sum(1)<=s1 AFTER 11ns;
 cout(0)<=c0 AFTER 11ns; cout(1)<=c1 AFTER 11ns;
 next_state:=NOT(next_state);
 ND IF;
 D PROCESS nrtz_fadder;

  nrtz_fa_behav;
```

```
CHITECTURE rtz_2mux_behav OF rtz_2mux IS
SIGNAL portid_ack: qsim_state := '0';
SIGNAL port_id: integer:= -1;
GIN


elect_input: PROCESS
 ARIABLE first_iterate, found: BOOLEAN := false;
 ARIABLE i: INTEGER:=0;
 ARIABLE next_state : qsim_state:='1';
 GIN
IF first_iterate = false  AND set='1' THEN
 first_iterate := true;
 LSE
 IF (sel(i) XOR sel(i+1)) /= next_state THEN
  WAIT UNTIL (sel(i) XOR sel(i+1)) = next_state;
 END IF;
 IF sel(1)='1' THEN
  port_id<=1;
 ELSE
  port_id<=0;
 END IF;
 WAIT UNTIL portid_ack=next_state;
 next_state:=NOT(next_state);
 port_id<=-1;
 ND IF;
 D PROCESS select_input;


 ta_input: PROCESS
 ARIABLE bit_count, portid_temp:integer:=0;
 ARIABLE first_iterate: BOOLEAN:=false;
 ARIABLE fun0, fun1: qsim_state:='1';
 ARIABLE next_state : qsim_state:='1';
 GIN
 F first_iterate = false OR set='1' THEN
 first_iterate:=true;
 f0out<='0';
 f1out<='0';
 a_out(1)<='1';
 a_out(0)<='1';
 LSE
 WAIT UNTIL (port_id > -1) AND set='0' AND a_in = next_state;
 IF next_state='0' THEN
  portid_temp:=portid_temp+port_id;
 ELSE
  portid_temp:=port_id;
 END IF;
 IF (fin0(portid_temp) XOR fin1(portid_temp))/=next_state THEN
  WAIT UNTIL (fin0(portid_temp) XOR fin1(portid_temp)) = next_state;
 END IF;
 fun0:=fin0(portid_temp);
 fun1:=fin1(portid_temp);
 f0out<=fun0 AFTER 6ns;
 f1out<=fun1 AFTER 6ns;
 a_out(portid_temp)<= NOT(fun0 XOR fun1) AFTER 8ns;
 portid_ack<=next_state;
 next_state:=NOT(next_state);
 VAIT UNTIL port_id = -1;
 D IF;
  PROCESS data_input;
```

APPENDIX D

Design Methodology

The design method used for the realisation of the ST-SISA adopts delay-insensitive circuit modules such that the resulting architecture will assume both unbounded intra- and inter-module delays. Therefore, the circuit modules themselves are responsible for co-ordinating computation within themselves and with their immediate neighbours. In order to build delay-insensitive systems out of these circuit modules a design method must be adopted to ensure straightforward and correct circuit construction that is amenable to other designers. There are a number of ways to build delay-insensitive systems, as have been described in the review chapter, each having their relative advantages and disadvantages. In this project we have chosen the building block, or macro-module, approach to delay-insensitive design.

This section begins by describing the functionality of the library of circuit modules, or macro-modules, that compose delay-insensitive circuits. Secondly, the method of inter-connecting delay-insensitive macro-modules, to form circuits of greater complexity, is described.

The Macro-Module Library

This section describes the macro-modules that are used for constructing delay-insensitive circuits. For many of the macro-modules described, two different implementations are required that differ only in the signalling protocol adopted at the data-interface; data-path macro-modules use the two-phase (NRTZ) signalling protocol whereas control-path modules adopt the four-phase (RTZ) protocol. Although these modules adopt different signalling protocols, the number and positioning of input and output wires remains constant for both the two-phase and four-phase implementations, therefore, each module is depicted only once.

The library of macro-modules can be separated into four categories namely: functional blocks, latches, switches and interconnection elements.

Latch

A latch [58], figure 1, is used to store and pass data between communicating circuits. The operation of the latch is to allow a new value to overwrite the output only when both a new data value is placed on the input and an acknowledge is received from the succeeding register, indicating the register can load a new value, otherwise the latch is forced to hold its current value.

Every time a new value is placed on the output a transition is placed onto the acknowledge output wire indicating to the preceding circuit that the data has been received. Both two- and four-phase latches are required, however the operation for each is the same as described above only the encoding of the data placed on the data interface differs.

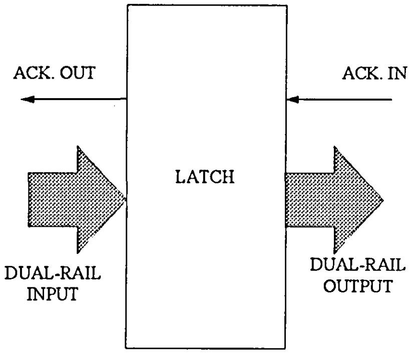


Figure 1. The Latch.

Functional Blocks

A functional block [58], figure 2, computes a specific combinational function upon dual-rail encoded data. It has no synchronising properties in relation to the control of the data paths. The type of combinational function varies on the type of computation. Typical functions include logical and arithmetic operators such as AND and ADD, in addition to more complex user specified functions such as those used in control circuitry.

As will be seen in the experimental chapters a wide variety of combinational functions are implemented, therefore, only the essential properties are discussed here. The main feature of a combinational circuit is that a new output cannot be produced until all the inputs have transpired to the same state, in other words it operates in fundamental mode.

For some of the functions used, both two- and four-phase implementations are required; such as the AND function used in the two- and four-phase ST-SISA ALU.

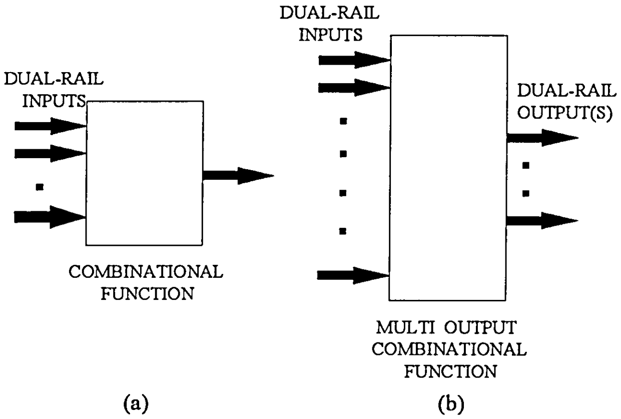


Figure 2. Single- (a) and Multi-output (b) combinational functions.

Switches

Switches [58, 93] are used as data flow control elements. There are primarily four different types of switch, namely: the multiplexor, demultiplexor, symmetric switch and asymmetric switch. These are explained as follows:

Demultiplexor

A demultiplexor, figure 3(b), has one data input which is output over one of 2^n data lines depending on the status of the n-input control bus. Therefore, a new data value can be placed on only one output and all other outputs maintain their current value.

Each demultiplexor, data interface is latched therefore the selected data path, with its associated acknowledge wires, operate in the same way as for the simple latch module. four- and two-phase implementations of the demultiplexor are needed, however the control inputs for both of these adopt the four-phase signalling protocol.

Multiplexor

A multiplexor, figure 3(a), has 2^n dual-rail inputs of which only one input line is selected, depending on the status of the n-input control bus, for connection to the output line. Therefore, a new data value can be placed on only one input and all other inputs must maintain their current value. Each data interface is latched in the same way as the demultiplexor. Two- and four-phase implementations are required.

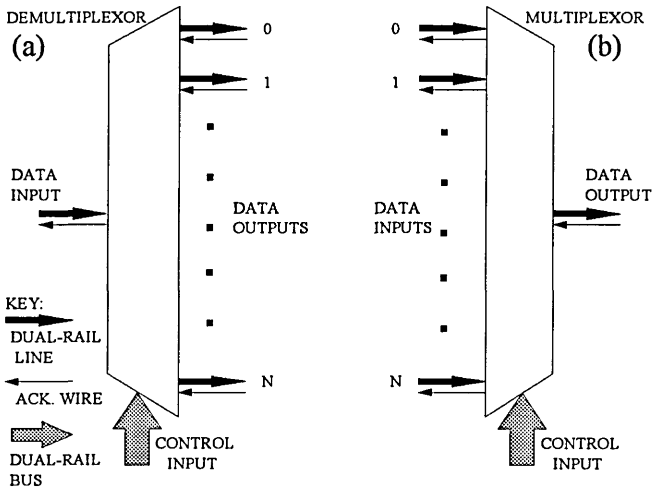


Figure 3. The multiplexor (a) and demultiplexor (b) modules.

Symmetric Switch

A symmetric switch, figure 4(a), has two data inputs where either both data values are passed through to their respective outputs or both of them are crossed over so that they are output to each others respective output. This function is controlled by a single four-phase dual-rail control input. Only a two-phase implementation of this module is required. This is because it is used to operate upon NRTZ data words that use all dual-rail logic bit combinations.

Asymmetric Switch

An asymmetric switch, figure 4(b), again has two data inputs where either both data values are passed to their respective outputs or only one of them is crossed over to the others output, therefore the other must wait. Only a two-phase implementation of this module, under the control of a four-phase control input, is required for the same reasons as discussed for the symmetric switch.

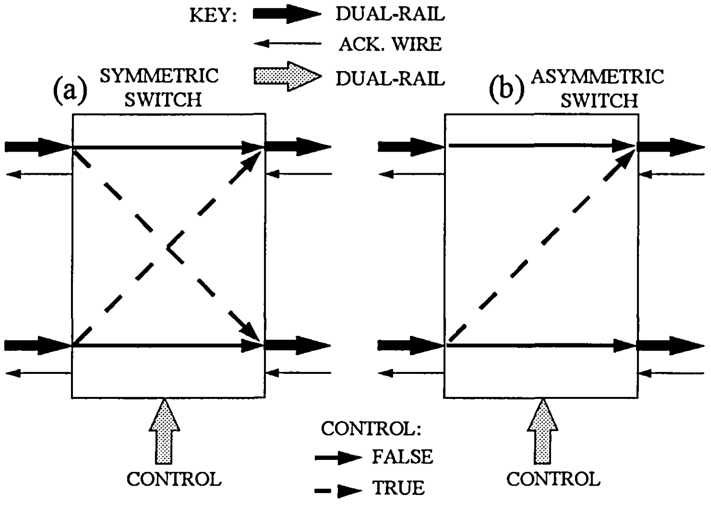


Figure 4. The symmetric (a) and asymmetric (b) modules.

The C-element

The C-element [100], figure 5, is used to AND signal events, such as acknowledge signals. When all of the inputs into a C-element are in the same logic state, the C-element's output changes to that state. When the inputs differ, the C-element retains its previous state and so its output is unchanged; the main use of the C-element in this project is to synchronise acknowledge signals between multiple communicating latching elements.

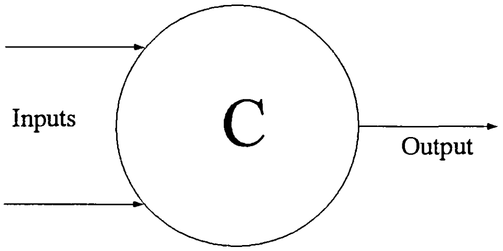


Figure 5. The C-element.

Fork and Join Elements

The structures used to interconnect multiple circuit modules are called fork and join [92, 93], as shown in figure 6, (a) and (b), respectively. The fork is used when the same signal is input to more than one circuit module. The join is used when signals from more than one source are input to a circuit module.

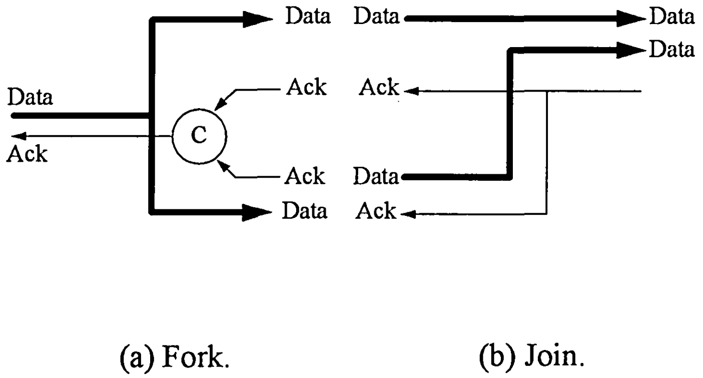


Figure 6. The fork (a) and join (b) interconnect structures.

An important aspect of both these structures is the handling of the acknowledge signals between the circuits involved. For the fork structure, each input module that accepts data from the output module must return an acknowledge signal to the output module. The output module must then synchronise all the acknowledge signals it receives such that the next data value cannot be accepted until all the acknowledge signals are in the same state. This is achieved by Anding all the acknowledge signals with a C-element. For the join structure, the acknowledge output produced by the input module must be forked to all the output modules that provide input data values.

Constructing Bus Macro-modules

It is possible to build bus orientated macro-modules, by connecting together a number of modules from those described earlier, such as a dual-rail bus demultiplexor. This type of bus function can be easily realised by connecting together a number of single dual-rail data-interface demultiplexors. For example, shown below in figure 7 is a circuit composed of two of the afore-mentioned demultiplexors which connect a two-input two bit wide data bus to two data bus outputs of the same bus width.

The circuit operates as follows. Whenever the control input status is set to false data on A0 and A1 is passed to outputs B0 and B1 respectively, otherwise, if the control input is true they are instead passed to outputs C0 and C1.

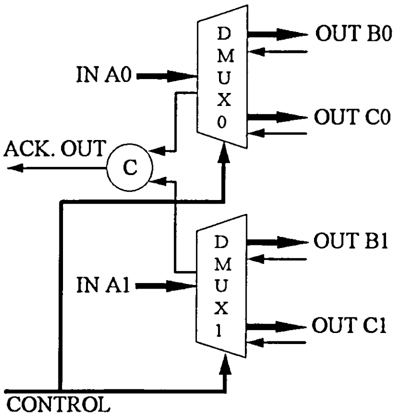


Figure 7. Bus demultiplexor.

Initialising Macro-modules

Each of above described macro-modules has another single rail input, not shown in the above figures, called either reset or set that is used to initialise a circuit module into a known state on power-up; both are global signals controlled by outside the system. The reset signal is used by RTZ modules to initialise their dual-rail outputs to logic '0', whereas the set signal is used by NRTZ modules to initialise their dual-rail outputs to logic '1'.

The drawback of the reset and set initialise signals is that neither can be adapted to the delay-insensitive protocol because both do not have accompanying acknowledge signals. However, the problem is limited by stating the assumption that a circuit module will settle into a known state within a finite period of time after the initialise signal has been asserted [70].

The System Construction Methodology

The design technique is based on the data-flow approach using pipelines and rings [92, 93] that are composed into larger structures through the joining and forking of data paths. The pipelines and rings are implemented by using a limited set of macro-modules consisting of latches, combinational circuits and switches that are inter-connected using the small set of simple interconnect elements described in the previous section.

The benefit of using this approach is that by limiting the class of circuit modules and interconnect structures it is possible to build complex designs that are correct according to simple construction rules. Another benefit is that the analysis of performance and the understanding of bottlenecks are made easier because of the simplicity of construction.

Pipelines

The composition of a FIFO pipeline can be realised if each stage of the pipeline includes a latch added to the output of a combinational function block, as seen in figure 8.

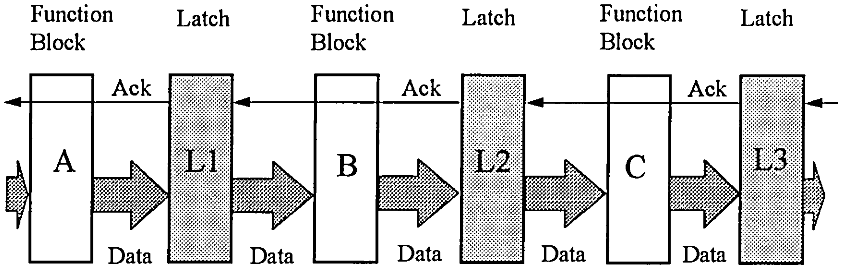


Figure 8. Delay-insensitive pipeline.

The function of the delay-insensitive latch is to prevent the incoming data from entering the combinational block until the successor latch has accepted the previous data. The latching of data is controlled by acknowledge signals from successor latches. For RTZ encoded data a latch may hold a new valid data item when its successor latch in the pipeline holds the empty value, and vice-versa for when the latch holds a empty spacer and the successor holds valid data. When the pipeline operates on NRTZ encoded data a latch may hold a new ODD valid data item when the successor latch holds a EVEN valid data item. The handshaking described ensures that the data flowing through the pipeline always consists of alternating valid and empty/tag values. This is the case for NRTZ encoded data in that data flowing through the pipeline always consists of alternating ODD and EVEN values, where some EVEN values may instead represent an EOW Tag.

Rings

Using a series of pipelined latches it is possible to connect the output of the last latch stage to the input of the first, forming a delay-insensitive ring. Such a ring is capable of performing an iterative computation.

Consider a ring containing three latches. The ring will always contain one of two patterns; either two valid and one empty element or two empty and one valid. Figure 9 (a,b,c,d) shows how the contents of a delay-insensitive ring iteratively changes as the empty and valid items are passed around. It can be seen when data propagates from one latch (the transmitter) to its successor (the receiver) the contents of the transmitting latch can be overwritten by data propagating from the preceding latch. Consequently, there are periods when the same data item occupies two stages, as shown in figure 9 (b) where the same empty (E) bit is copied into two neighbouring latches. This is called a bubble [23]. Bubbles are essential for propagating data so that when data propagates forwards though a pipeline, bubbles are created therefore creating space for new data.

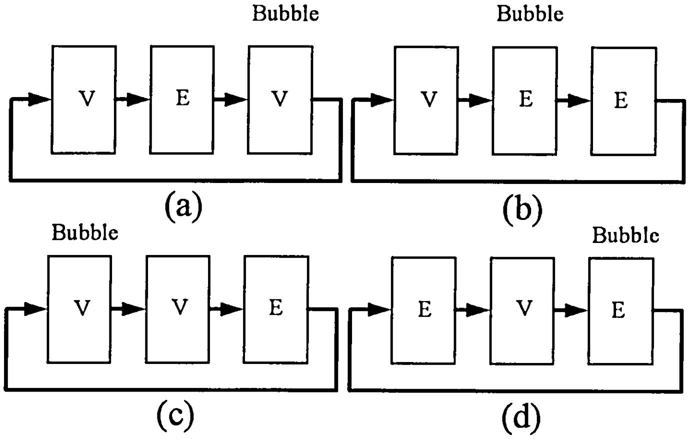


Figure 9. Sequence of computations in a ring.

Basic System Interconnection Structures

The fork and join circuit interconnect elements can be used to interconnect a number of separate data paths together thus forming more complex circuit structures.

There are three basic types of circuit structure [70]. The first is the join structure, figure 10, which joins two (or more) data paths together at a combinational function input. The second is the split structure, figure 11, which separates a single data path into two (or more) data paths from a combinational function. Finally, the join and split structure, figure 12, brings together two (or more) data paths at the function inputs and then immediately splits them into two (or more) data paths at the output. Each of these structures must be latched at both their input(s) and output(s).

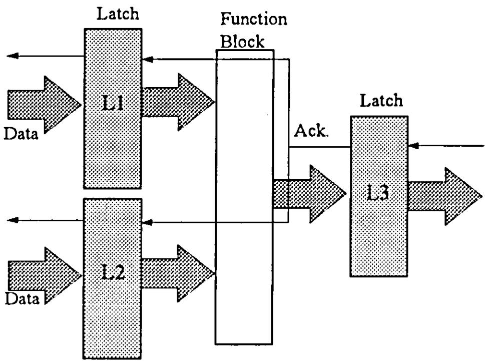


Figure 10. Join structure.

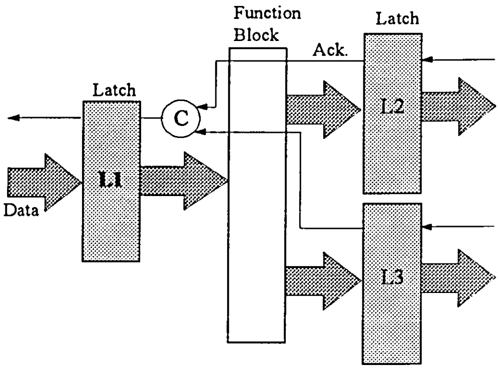


Figure 11 Split structure.

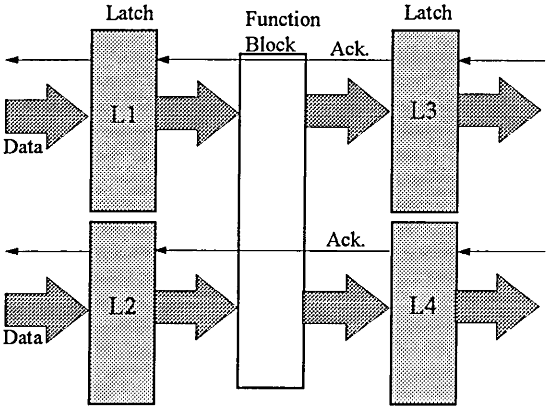


Figure 12. Join and split structure.

Multi-ring Structures

The ring and pipeline structures described earlier can be connected together to form more complex structures called a multi-ring structure [92, 93] using the three basic circuit structures described in the previous section. Figures 13, (a), (b) and (c), respectively show how the join, split, and join and split structures are used to form multi-rings.

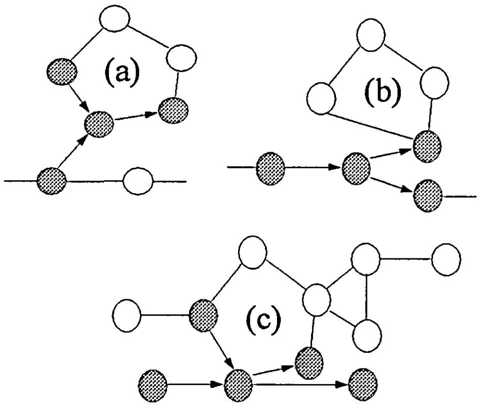


Figure 13. Multi-ring structures.

Finite State Machines

In order to build a delay-insensitive finite state machine (FSM) [70], as shown in figure 14, two or more latches otherwise known as register, must be combined with a functional logic block. It can be seen that the join element is used to synchronise the transfer of data from the primary and secondary input latches to the latch at the output of the functional logic.

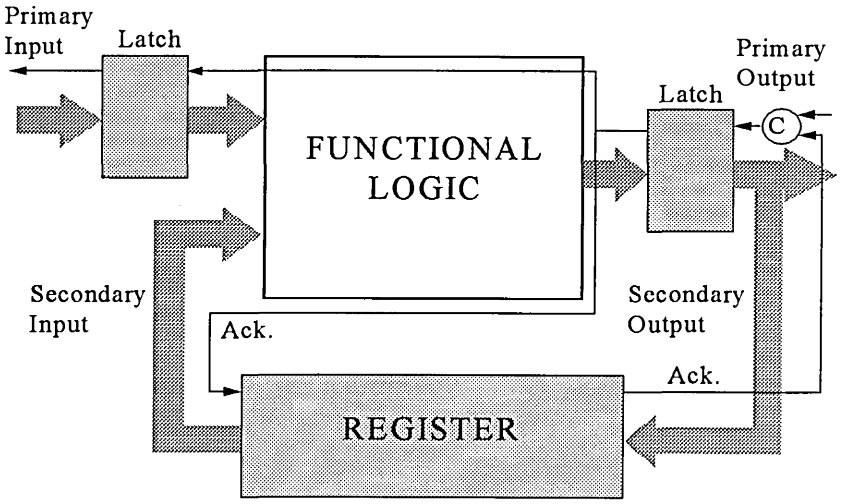


Figure 14. Delay-insensitive finite state machine.