

SERIAL-DATA COMPUTATION IN VLSI

by

Stewart G. Smith

A thesis submitted to the Faculty of Science,
University of Edinburgh, for the degree of
Doctor of Philosophy

Department of Electrical Engineering
1987



UNIVERSITY OF EDINBURGH

ABSTRACT OF THESIS

(Regulation 7.9)

Name of Candidate Stewart Gresty Smith

Address 12 [REDACTED]

Degree Ph.D. Date 26th January 1987

Title of Thesis Serial-Data Computation in VLSI

.....

No. of words in the main text of Thesis 38,000

This thesis describes novel advances in serial-data architectures, which achieve area-savings and performance improvements over conventional approaches. Computational structures which exploit these advances are specified in terms of fundamental serial-data elements, or ‘atoms’. Methods are detailed to specify these structures in technology-independent form, along with composition procedures for their automatic assembly into powerful functional primitives. Initial progress on a full-span structural silicon compiler is reported, which will provide area-efficient full-custom realisations of functional primitives from technology-independent specifications. The resulting library of modules is *functionally*, *technologically* and *operationally* flexible, promising an effective route from structural intent into silicon.

A fundamental theory of serial-data computation is expounded, and a set of methodical systems design techniques is identified. From these concepts a practical framework is constructed for serial-data systems design, and a rigorous case study in the form of a polyphase-network filterbank specified by the European Space Agency illustrates many of these ideas in practice. The knowledge gained from this and previous studies reveals the need for advances in serial-data architectures at the functional component level.

Reported architectural novelty at this level falls into two main areas: techniques for area-savings, and techniques for performance enhancement. The former category is based around an architectural synthesis method for matrix-vector computational modules such as complex multipliers. Two’s complement serial/parallel carry-save accumulation provides performance, while the use of symmetric-coded distributed arithmetic eliminates redundant computation to effect area-savings. The latter category comprises three architectural techniques which accelerate bit-serial computation without compromising its favourable advantages. In essence they rely on multi-wire representations of serial data - a step towards bit-parallelism. Interfacing techniques are developed to support the existence of domains of different throughput within a system, thereby enhancing the range of bandwidth-matching techniques available to the systems designer, and realising the potential to mix processing wordlengths within a serial-data system. These innovations address the most common criticisms of the bit-serial approach.

The novel architectural techniques described in this thesis may be used to enhance the function libraries of design automation tools such as the silicon compiler, providing systems designers with a flexible set of serial-data computational components with which to realise efficient solutions to digital signal processing problems.

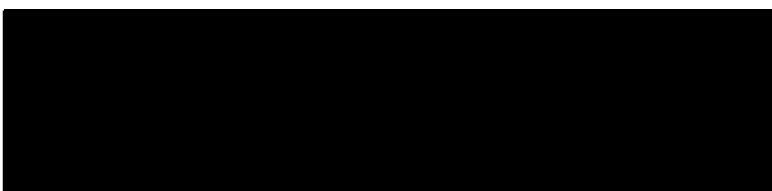
Declaration of Originality

The material contained herein, except where stated below, was researched and composed entirely by myself in the Department of Electrical Engineering at the University of Edinburgh, between January 1983 and January 1987. The exception to this statement is the work on software tool development and manual layout reported in Chapter 8, which is the product of a significant team effort to which the author has contributed at the level of architectural specification.

Acknowledgements

I wish to acknowledge the inspiration and guidance of my supervisor Prof. Peter Denyer, the creator of the *FIRST* silicon compiler and the prime mover of the VLSI group here, I wish to thank Profs. Jeff Collins and John Mavor, who as successive Heads of Department have done so much to create the infrastructure which I take for granted, my current colleagues David Renshaw and Alan Murray, who have helped make *FIRST* what it is, also Mike McGregor for the twin-pipe concept and multiplier layout effort, my erstwhile colleagues Neil Bergmann, Jim Nash, Dave Fletcher and Steve Belcher for software support, and Prof. Kunihiro Asada for *MOSYN*. Next I wish to thank those friendly residents of California who gave me so much of their valuable time in one-way technical discussions as well as hospitality above and beyond the call of duty - these include Milos Ercegovic (UCLA), Dick Lyon (Schlumberger SPAR), Carver Mead (Caltech), Greg Nash (Hughes), Earl Swartzlander (TRW) and Stan White (Rockwell). Their words and deeds have greatly influenced my technical direction. Thanks also to Bob Mhar and Harry Paul, who as M.Sc. students produced multiplier layout to my specification. Finally I wish to thank my wife Lesley for her patience as I prepared this thesis.

Signed



Stewart Gresty Smith

Index of abbreviations

In the interests of brevity, some acronyms and abbreviations peculiar to this thesis are coined and subsequently cited. These are listed below:

CSA	Carry-save adder	PIP	Partial inner-product
CSAS	Carry-save add-shift	PIPO	Parallel-in-parallel-out
DA	Distributed arithmetic	PISO	Parallel-in-serial-out
IP	Inner-product	PP	Partial product
JKM	Jackson, Kaiser & McDonald	PPP	Parallel partial product
LS	Least-significant	PPS	Partial product sum
LSB	Least-significant bit	SIPO	Serial-in-parallel-out
MB	Modified-Booth	SISO	Serial-in-serial-out
MS	Most-significant	SPP	Serial partial product
MSB	Most-significant bit	S/P	Serial/parallel
OB	Offset-binary	2C	Two's complement

Contents

Abstract	I
Declaration of originality	II
Acknowledgements	II
Index of abbreviations	III
Contents	IV
 Preamble	 1
 Chapter 1: Introduction to serial-data computation in VLSI	 2
The case for serial-data techniques	2
The case against serial-data techniques	4
Historical overview of bit-serial techniques	4
Programmable serial machines	5
Fixed-function serial machines: the ‘approach’	6
Final comments	7
 Chapter 2: Rudiments of serial-data computation	 8
Issues of space and time	8
Control	9
Two’s complement integer coding	9
Fundamental building blocks - the atoms	11
Latches	11
Logic	15
Numerical principles of serial-data additive operations	20
Partitioning issues	24
 Chapter 3: Two’s complement serial-data multiplication	 26
Derivation from bit-parallel architectures	27
Scrutiny of two serial-data multiplication architectures	28
Bit-serial derivation of the architectures	29
Bit-parallel derivation of the S/P architecture	37
The serial/parallel flush multiplier	38
Word-level equivalent architectures	39
Comparison of the S/P and Lyon multipliers	40
Serial/parallel multiplier environments	43

Other approaches	43
Quasi-serial multiplication	44
Incremental multiplication	44
Storage sharing	47
Chapter 4: The Edinburgh Tools	48
Background	48
<i>FIRST</i>	49
The <i>FIRST</i> primitive set	51
Case studies	52
<i>MOSYN</i>	52
The promise of full-span structural silicon compilation	54
Chapter 5: Bit-serial systems design: methodology and case study	55
Bit-serial systems design	55
Functional design - the soft model	56
Physical design - the hard model	58
Postscript	60
Case study system description	60
FFT subsystem design	61
FFT overview	61
Vector rotation	62
The pipeline FFT	63
Filterbank subsystem design	63
System specifications	64
Initial design issues and decisions	64
Resolving the issues	66
Additional specifications	67
Functional design - the soft model	69
The FIR section	70
The FFT section	72
The polyphase-network filterbank	74
Physical design - the hard model	74
The FIR section	76
The FFT section	78
Test strategy and confidence levels	80

A critical appraisal of <i>FIRST</i>	82
Chapter 6: Special serial-data techniques for area reduction	84
Overview of vector computation	84
Some carry-save approaches to vector computation	85
Serial/parallel symmetric-coded distributed arithmetic	86
The symmetric-coded serial/parallel multiplier	86
The serial/parallel inner-product computer	89
Architectural case studies	92
Architectural synthesis	97
Comparison with conventional approaches	97
Distributed arithmetic in context	98
Cascading	99
Incremental computation of squares and sums of squares	100
Incremental squaring	101
Incremental sums-of-squares computation	104
Bidirectional incremental multiplication	105
Application to the on-line algorithms	106
Final comments	109
Chapter 7: Special serial-data techniques for throughput enhancement	110
Twin-pipe	110
Twin-pipe building blocks	111
Architectural implications of twin-pipe techniques	116
Radix-4	117
Radix-4 building blocks	118
Architectural implications of radix-4 techniques	125
Multi-precision	125
Multi-precision building blocks	126
Architectural implications of multi-precision techniques	128
Interfacing between operational domains	128
Stripping down the CSAS computer - the automultiplier	131
Automultiplier summary	134
Final comments	135

Chapter 8: Full-span structural compilation of serial-data hardware	137
<i>FIRST</i> in perspective	138
Synthesis of Elementary Circuits ON Demand	139
Functional components of <i>SECOND</i>	141
PRIMITIVE specification (design capture)	142
PRIMITIVE verification (behavioural simulation)	145
PRIMITIVE implementation (physical assembly)	146
Example PRIMITIVE design - the serial/parallel multiplier	147
Other approaches	153
Final comments	154
 Chapter 9: Concluding remarks	 156
 References	 158
Appendix A - <i>FIRST</i> description of the soft model	171
Appendix B - <i>FIRST</i> description of the hard model	180
Appendix C - <i>RNL</i> description of the twin-pipe complex multiplier	189
Appendix D - Author's publications	202
Appendix E - Included publications	204

Preamble

This thesis is concerned with advances in serial-data computational architectures. The fundamental operational principles behind bit-serial computation are detailed, and the bit-serial field reviewed, in Chapters 1 to 3. Chapter 4 introduces *FIRST* and *MOSYN*, two design automation software packages developed at the University of Edinburgh. A design methodology for bit-serial systems is outlined, and *FIRST* is evaluated in a rigorous systems case study of a satellite communications signal processor which follows in Chapter 5. Chapters 6 and 7 are devoted to novel techniques for enhancing efficiency of bit-serial architectures, and finally in Chapter 8 a programme for automation of primitive design is outlined, including as an example the design and test of a serial/parallel multiplier chip which embodies many of the techniques of Chapters 6 and 7. Chapter 9 consists of concluding remarks.

Appendices A and B are *FIRST* codings of functional and physical models of the system specified in Chapter 5, and Appendix C summarises a primitive design example, the twin-pipe complex multiplier. This device exhibits many of the architectural innovations of Chapters 6 and 7, and illustrates the design capture concepts of Chapter 8. Appendix D lists the author's publications, and finally Appendix E reproduces some earlier, published case studies of *FIRST*. At key points throughout the thesis, author's publications are marked with the dagger symbol [†] and footnote.

Chapter 1

Introduction to serial-data computation in VLSI

The term 'serial-data' is used to describe the computational style presented in this thesis. We differentiate between this, and the expressions 'bit-serial' or 'bit-sequential', because, while many of the concepts follow directly from traditional bit-serial techniques, several forms of bit-parallelism are exploited in the proposed serial-data architectures. Serial-data techniques thus may be viewed as a superset of bit-serial techniques.

1.1. The case for serial-data techniques

The demands of modern high-performance numerical applications are such that they must be met in principle by special-purpose computational architectures [1,2]. Computational styles fall into two broad areas - serial-data and parallel-data (bit-parallel). In the applications area addressed by this thesis, namely fixed-function, real-time computation, serial-data machines exhibit many advantages over equivalent bit-parallel machines. Some of these advantages are listed below.

Performance and efficiency

Serial-data hardware consists of pipelines of small, combinatorial logic elements followed by latches. These elements are heavily exercised, and may be clocked at high rates. Numerical addition is performed in carry-save form - this moves carry-propagation into the time-domain, bringing the full performance benefits of pipelining. As multipliers consist mainly of arrays of adders, it follows that serial-data multipliers are also fast and hardware-efficient.

Functional parallelism

Functional parallelism, i.e. the use of parallel arrays of high density, low-cost, lower performance devices to obtain a high performance function [3], is a natural way to exploit the concurrency inherent in many real-time computational problems. More functional parallelism means less control overhead - in the limit, an

'isomorphic' hardware flowgraph has no control overhead at all [4]. The lower computational 'grain-size' of serial-data elements makes them better suited to mass instantiation than their bit-parallel functional equivalents.

Physical partitioning

A further consequence of small grain-size is ease of physical partitioning. Furthermore, single-wire communication (as opposed to buses) eases pin-out problems in arbitrarily partitioned systems of chips. A similar advantage may be measured on-chip, where serial communication does not dominate chip area as do parallel buses. Fine-grain implementations usually result in smaller collections of wires running for smaller distances.

Testability

Serial-data elements are amenable to testing by random patterns, a testing technique which is most appropriate when logical fan-in and combinatorial depth are low [5]. Test costs in serial-data hardware are a simple pseudo-random binary sequence generator beside each data input pad, and data-compression register beside each output. Entire systems may be tested in this manner, without including testability hardware *within* processors, which adversely affects performance and area-efficiency. Fault coverage is achieved by brute force, obviating the need for expensive fault simulation and automatic test pattern generation [6].

Yield and fault-tolerance

A final consequence of low grain-size is the potential for fault-tolerance through redundancy. Serial elements are exponentially better yielding than their parallel counterparts, and the lower cost of providing redundant serial elements is proportional to the product of area and yield - a sharply exponential function which weighs against the parallel case [4]. Moreover, the cost in switching and bussing of routing round faulty elements is considerably less in the serial case.

1.2. The case against serial-data techniques

Serial-data techniques are not proposed as the solution to all computational problems. There are applications at many levels which are better suited to bit-parallel realisation. A list of serial-data disadvantages follows.

Fixed word length

A fundamental parameter in serial-data design is the system wordlength. Data must be represented in words of this length throughout the system (other than in word-organised memories [7]), regardless of local needs. This is in contrast to bit-parallel realisations, where wordlength may be tailored to local requirements [8].

Data-independent operation

Many computational algorithms contain data-dependent operations, whether at system level (e.g. conditional branching), or at the ‘advanced’ level of arithmetic operation (e.g. division, square-root extraction). In both cases comparison of 2 operands is required, a function which cannot be pipelined in serial manner. Data-dependent operations must be implemented in bit-parallel fashion, and such operators with serial interfaces come at a heavy cost in area and time [7].

The advanced arithmetic operations are best suited to bit-parallel operation, although an interesting serial-data approach is apparent in the ‘on-line’ algorithms [9, 10]. Here computation proceeds MSB-first, in the same direction as the natural flow of the algorithm. Redundant data representations permit fast, carry-free parallel addition [11] internally, and computational latency is low.

1.3. Historical overview of bit-serial techniques.

Bit-serial computation and communication have traditionally been employed for the most part in dedicated real-time systems. The tools and architectures proposed in this thesis address this applications area. However there exists another class of bit-serial machines, aimed at general parallel processing, built around arrays of single-bit processing elements (PEs) as proposed by Unger in 1958 [12] for pattern recognition problems. We briefly review this class of machine.

1.3.1. Programmable serial machines

Programmable serial machines process arrays of data concurrently, one bit at a time, thus achieving considerable speed up over bit-parallel, sequential von-Neumann machines for large arrays. Such machines can exhibit greater storage and processing efficiency than their bit-parallel counterparts. They can handle data elements of any length, operating on reduced word-segments when necessary (e.g. single-cycle sign-testing on data elements).

Much of the groundwork for such machines was performed at Goodyear Aerospace, whose original STARAN project led to the development of the airborne associative processor (2000 PEs) for advanced radar systems, and the Massively Parallel Processor (16384 PEs) for satellite image processing [13,14]. The Connection Machine [15] is based on the concept of cellular automata, and is aimed at supercomputing and AI applications. It contains some 64000 fine-grain processing elements connected in a hypercube arrangement. In the UK, ICL developed the distributed array processor (DAP) [16]. Current versions operate with a 32 x 32 array of bit-serial full adders.

With the advent of VLSI, we are seeing the integration of arrays of single-bit PEs on a single piece of silicon, based on the Single-Instruction Multiple-Data (SIMD) concept [17]. Many of these are optimised for image-processing tasks, e.g. Brunel University's SCAPE [18], University College's CLIP machines [19], GEC's GRID [20] and NTT's adaptive array processor [21]. These devices contain square arrays of PEs (a fairly comprehensive review of such architectures appears in [19]).

More recent machines include the ITT cellular array processor (CAP) and NCR's geometric array parallel processor (GAPP). The CAP [22] is an expandable array processor (up to 16384 PEs), while the GAPP [23] is an integrated circuit containing 72 PEs. A new company, AMT, is developing a VLSI DAP-3. Reeves & Bruner proposed a logic synthesis methodology for machines such as these [24], where complex functions are decomposed in heuristic manner for direct execution as a sequence of 2-input operations.

In marked contrast to these minimal grain-size machines, Lyon has demonstrated MSSP, a programmable bit-serial speech processor [25]. MSSP represents a

‘reconfigurable flowgraph’, using multi-port RAM simultaneously as a storage and a switching medium in concert with a limited set of serial operators which includes larger elements (such as multipliers and limiters) along with adders and an ALU.

1.3.2. Fixed-function serial machines: the ‘approach’

The first realisation of the bit-serial approach to implementation of fixed-function machines was reported by Jackson, Kaiser and McDonald (JKM) in 1968 [26]. They described a set of ‘building-blocks’ which could be configured as a second-order filter section [27]. This in turn could be multiplexed with signal state memory to realise cascades of second-order sections for speech-bandwidth processing. More recently, several examples of this architecture have been implemented in LSI [28, 29, 30].

Around 1980 Powell proposed a technique known as functional design [31], where systems are constructed entirely from bit-serial functional operators of a common generic type. At the same time Lyon was extending the building-block concept by introducing standard interfacing conventions, and structured design management (nesting building blocks) [32]. The *FIRST* silicon compiler [7] was a logical extension to Powell and Lyon’s approach, automating the design of bit-serial signal processors. By combining a language compiler, simulator, physical cell-library and floorplanner, entire systems of bit-serial chips could be rapidly specified and accurately realised.

Fault-tolerant architectures

For reasons listed earlier, bit-serial elements exhibit great potential for fault tolerance. The ‘superchip’ architecture developed at the University of Edinburgh uses a large crossbar switch-matrix [33] to route traffic in a reconfigurable array of bit-serial processors. The *RVSLI* (restructurable VLSI) project at MIT uses lasers to make and break links around faulty processors on a wafer, and has seen success in diverse applications such as radar signal processing and speech processing [34, 35].

Floating-point

To date, most bit-serial hardware has used fixed-point, or integer arithmetic. The CUSP project at Cornell University employed a block floating-point number format [36] - this technique has since been extended to embrace genuine floating-point operation [37].

1.4. Final comments

Despite the criticisms levelled above, the case for using serial-data techniques in fixed-function architectures is a strong one. Later in this thesis, techniques are described which go some way towards answering these criticisms. In the next Chapter, we proceed to examine the fundamental principles behind serial-data architectures.

Chapter 2

Rudiments of serial-data computation

This chapter introduces the basic theory of serial-data computation, and the fundamental building blocks, or ‘atoms’, required for its execution.

2.1. Issues of Space and Time

A binary-coded, single-source digital signal may be viewed as a 2-dimensional array of bits. When the signal is represented in bit-parallel form, one dimension is space (spanned by the finite bit-index), and one is time (spanned by the potentially infinite sample-index). When the same digital signal is represented in bit-serial form, both dimensions are temporal, and some means must be found of marking the boundaries of samples (also known as *words*).

Each bit of a binary-coded signal has a weight associated with it, i.e. the bit-index is linked to a power of 2. In bit-parallel signal representations, index is implicit in the spatial distribution of bits - similarly, in bit-serial representations index is implicit in the temporal distribution of bits. A corollary of this is that any bit of a bit-parallel signal has equal weight to its temporal neighbours (where a temporal neighbour is defined as a bit existing at the same point in space at a different point in time). Similarly any bit of a bit-serial signal has equal weight to its spatial neighbours (where a spatial neighbour is defined as a bit existing at the same point in time at a different point in space). Should it be desired to introduce relative inequalities in weight, explicit arithmetic shifting hardware must be provided. This has important consequences in the design of serial-data computational elements, as will shortly be demonstrated.

Finally in the bit-serial case, there are two distribution options for the bits of a data word. The word may be transmitted and processed either LSB-first, or MSB-first [39]. For now we assume the former scheme to be adopted, as practised in the ‘approach’ [26, 33, 7].

2.2. Control

As just described, serial data computation and communication requires the representation of a two-dimensional entity (the signal bit-pattern) in one-dimension (time). The distinction between data *words* is not clear in a single bit-stream. A common solution to this problem [33, 7] is to provide a separate control pulse which coincides with the end (or more usually the beginning) of a signal sample.

Thus a bit-serial signal requires 2 wires for meaningful interpretation - one for *data*, and one for *control*. The information content of the latter does not extend beyond delineation of words. It controls the loading and unloading of registers, and instigates operation of the single-bit arithmetic shifters crucial to bit-serial computational elements. A fundamental task in bit-serial design is to ensure, by use of delay elements and arithmetic shifters, that all spatial neighbours are in synchronism with their associated control signal, and hence with each other.

LSB-control may be derived by counting the clock, generating a pulse on overflow/reset. Serial-data control at system level may be derived from a hierarchy of counters [7]. Arbitrary levels of control may be obtained by counting the previous level control signal (the clock acts as 'level 0').

2.3. Two's complement integer coding

The hardware employed in any computational system is closely related to the number system employed. Following the convention of *FIRST*, we employ two's complement (2C) number coding as an interfacing convention, although we may depart from this convention *internal* to computational structures if it is expedient to do so.

An n -bit 2C number is represented by a pattern of bits distributed spatially (if bit-parallel) or temporally (if bit-serial). An n -bit 2C number A may be expressed as follows:

$$A = -a_0 + \sum_{i=1}^{n-1} a_i 2^{-i}, \quad (a_i \in 0,1), \quad 2.1$$

The MSB a_0 is given negative weight, allowing representation of negative numbers when MSB is 1. Note the inclusion of a binary point in eqn. 2.1.

Binary point

For an integer representation to have any relative meaning, the position of the binary point must be fixed. In the above example, the binary point is situated to the right of the MSB, as we read from left to right with decreasing bit-significance. Thus:

$$1 - 2^{1-n} \leq A \leq -1$$

An n -bit 2C number A may be expanded in either direction by arbitrary amounts, by padding zeroes to the right (in the direction of decreasing significance) and by padding sign-repetitions to the left (in the direction of increasing significance). To see this, inspect the expression

$$A = -a_0 \left[-2^y + \sum_{i=0}^{y-1} 2^i \right] + \sum_{i=1}^{n-1} a_i 2^{-i} + \sum_{i=n}^{x+n} (0) \cdot 2^{-i} \quad 2.2$$

for equivalence with eqn. 2.1, where we include y sign-repetitions and x trailing zeroes, x and y arbitrary.

Arithmetic shifting

Arithmetic shifting may be employed to effect multiplication and truncating division by powers of 2, i.e. to alter the weight associated with the bits of an input operand. If we imagine an n -bit window on the extended bit-field described by eqn. 2.2, then moving the bit-field in one direction while the window *and* the binary point remain stationary effects an arithmetic shift in that direction. While the effect of right-shifting is a gradual loss of accuracy and precision, left-shifting beyond existing sign-repetitions or 'guard-bits' causes numerical overflow, a catastrophic situation which must either be accommodated by numerical limiting, or avoided altogether.

2.4. Fundamental building blocks - the atoms

As stated earlier, a wide range of functional architectures may be composed from a small set of serial-data building blocks. These building blocks are themselves composed of an even smaller set of elements, or 'atoms'. The elements fall into two categories, logic (for evaluation of Boolean functions), and latches (for stable storage of results). We address the latter category first.

2.4.1. Latches

The successful design of VLSI systems requires careful management not only of the 2-D silicon area but also of the operation of the system in the time dimension [40]. The use of latches (clocked storage elements) is key to the management of signal timing in serial-data systems. The main goal of clock scheme and latch design is to control the propagation of signals in synchronous systems. This thesis is not primarily concerned with circuit engineering issues, and latches are abstracted to simple behavioural models.

Serial-data hardware consists of small pieces of random logic with latched outputs, connected in cascade. Performance comes from the intrinsic pipelining of these structures. Logic blocks evaluate while their inputs are latched, and in turn pass their outputs to be latched on the next clock cycle. The maximum operating clock rate for any serial-data element is then the sum of the settling times of a latch plus that of the slowest constituent logic block. A secondary limiting factor on clock rate at system level is the propagation delay of communication channels - such propagation may itself be pipelined to maintain performance [7].

Latch operation

Latch operation, clocking schemes, and the type of logic (static or dynamic) employed are determined by considerations of circuit and processing engineering. A wide variety of combinations are used in practice. Mead and Conway [41] advocated a simple two-phase non-overlapping clock scheme for nMOS technology (this was adopted in *FIRST*). A pseudo 2-phase scheme may be adopted for CMOS, where inverses to the two clock signals are generated locally [42]). 4- and even 6-phase schemes have been proposed [42].

Recent work at the University of Edinburgh has resulted in a novel single-phase clocking technique and half-latch pair construct [43]. The devices fabricated to verify the architectural ideas of later chapters employ this clocking style (see Chapter 8), and throughout this thesis any references to clock or latches pertain to the above.

The concept of single-phase clocking is simple. Conventional 2-phase systems use (say) ϕ_1 as load signal for an input register, and ϕ_2 as load signal for an output register (Figure 2.1(a)). These registers may be viewed as identical *half-latches*, and the structure of Figure 2.1(a) without combinational logic forms a *full-latch*. Single phase systems (Figure 2.1(b)) use ϕ as load signal to both half-latches, necessitating two complementary half-latch types.

The clock signal ϕ is considered to be a square wave at the clocking frequency. When high the clock is said to be in the ϕ phase, and when low it is in the $\bar{\phi}$ phase [43]. A half-latch is associated with each phase. A π -latch is so called because it holds data stable during ϕ , and a μ -latch because it holds data stable during $\bar{\phi}$. The half-latches are illustrated in Figure 2.2. Greek letters π and μ symbolise the holding phases of the two half-latches [43]. A signal emanating from a π -latch is said to be π -timed (similarly μ). A μ -timed full-latch may then be realised by connecting a π -latch and a μ -latch in cascade (vice-versa for π -timing).

The benefits of single-phase clocking are fourfold [43]:

- 1) Only a single global ϕ need be generated and distributed.
- 2) Neither generation or distribution of ϕ is likely to be a limiting factor on maximum sustainable clock frequency.
- 3) ϕ may be generated off-chip, with attendant savings in chip area and power consumption.
- 4) ϕ cannot easily be skewed with respect to itself.

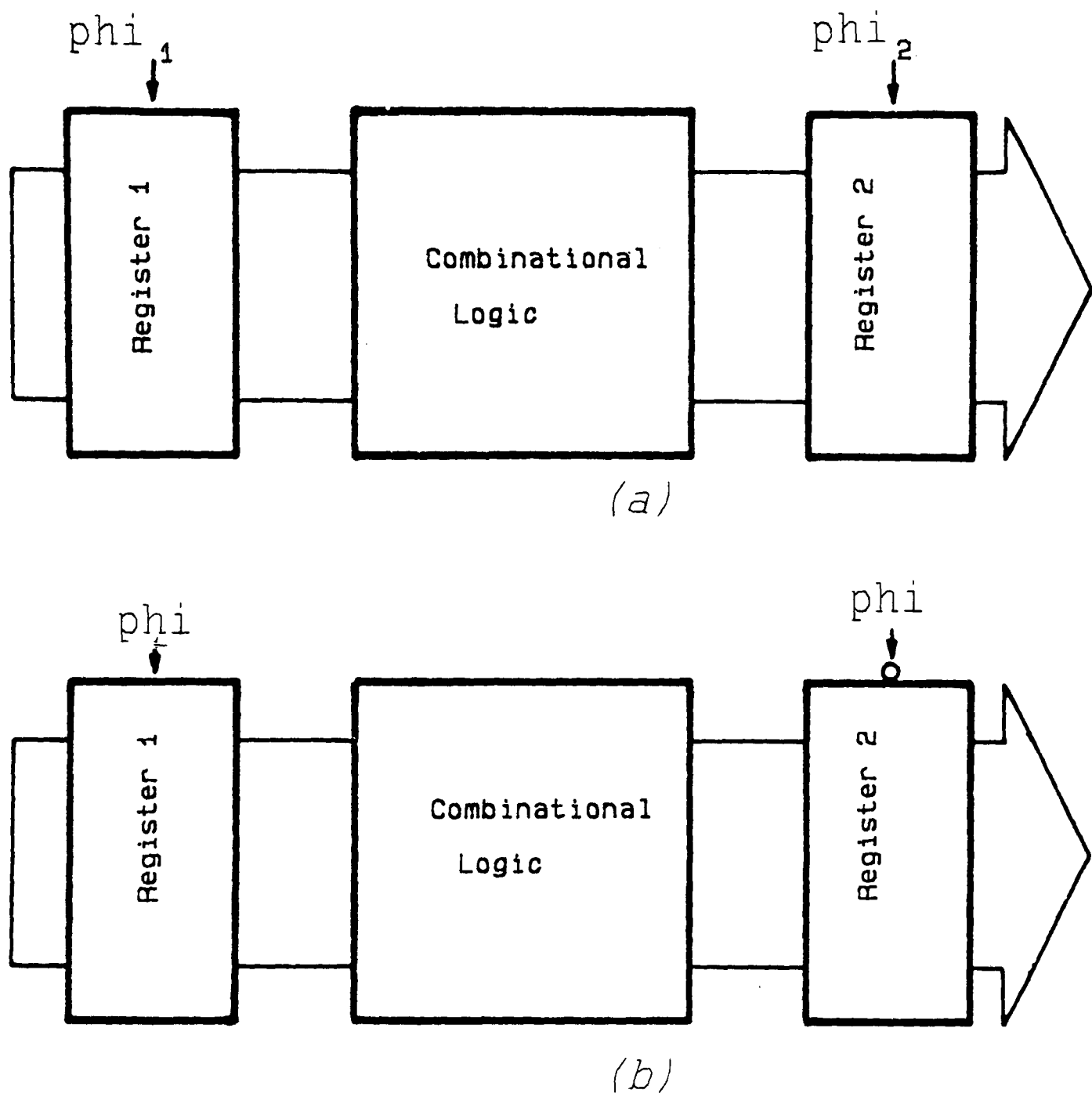
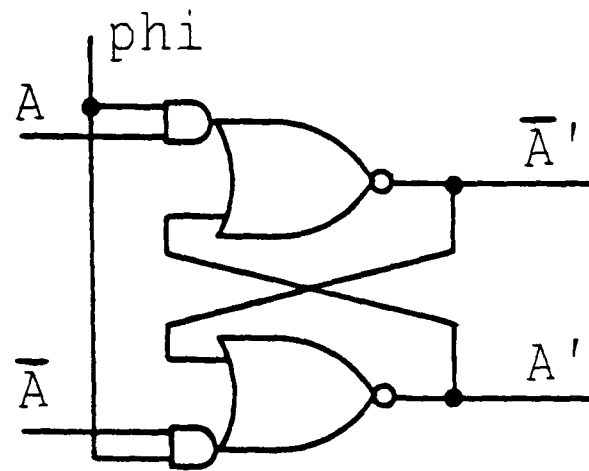


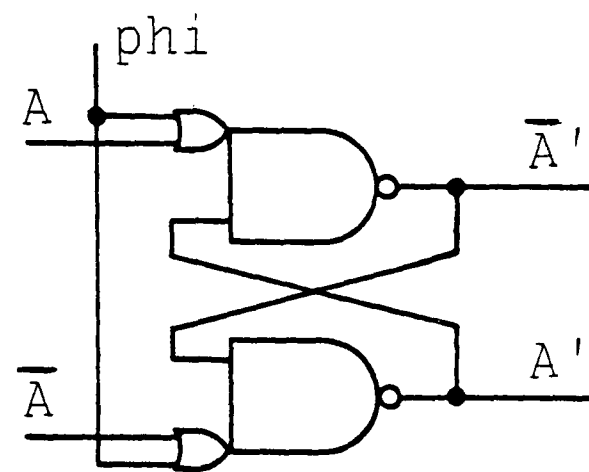
Figure 2.1: (a) conventional 2-phase (b) single-phase clocking

Holding latches

Some enhancements to the basic latch may improve efficiency of designs. In some cases we may wish to hold data steady for arbitrary periods longer than one clock cycle. A holding latch features a second control input signal (ENABLE in Figure 2.3). The holding operation is disabled (i.e. the register is loaded) only on certain combinations of clock and control signal. Figures 2.3(a) and (b) show fully-static π - and μ -latches, and Figures 2.3(c) and (d) their dynamic equivalents. Latches may even contain built-in function, e.g. set, load and clear.



(a) Fully Static mu Latch



(b) Fully Static pi Latch

Figure 2.2: fully-static half-latches

Registers

Connection of latches in cascade forms a register - a common structure in serial-data architectures, as serial-data computation is performed in pipelined registers which contain combinatorial function. An n -bit register contains n latches, and accordingly stores n bits of data. These registers may have parallel or serial inputs and outputs, in all combinations. In the following register acronyms, we use the letter P for parallel, S for serial, I for in and O for out. Four fundamental components are the PISO, PIPO, SISO (sometimes known as synchronous FIFO - first-in-first-out register) and SIPO. These structures are illustrated in Figure 2.4. In all but the logic-free case (the SISO - Figure 2.4(c)), the combinatorial element is a multiplexer (described shortly), which is used to load the register.

Note that each stage of the PO structures recirculates a local bit, while in the

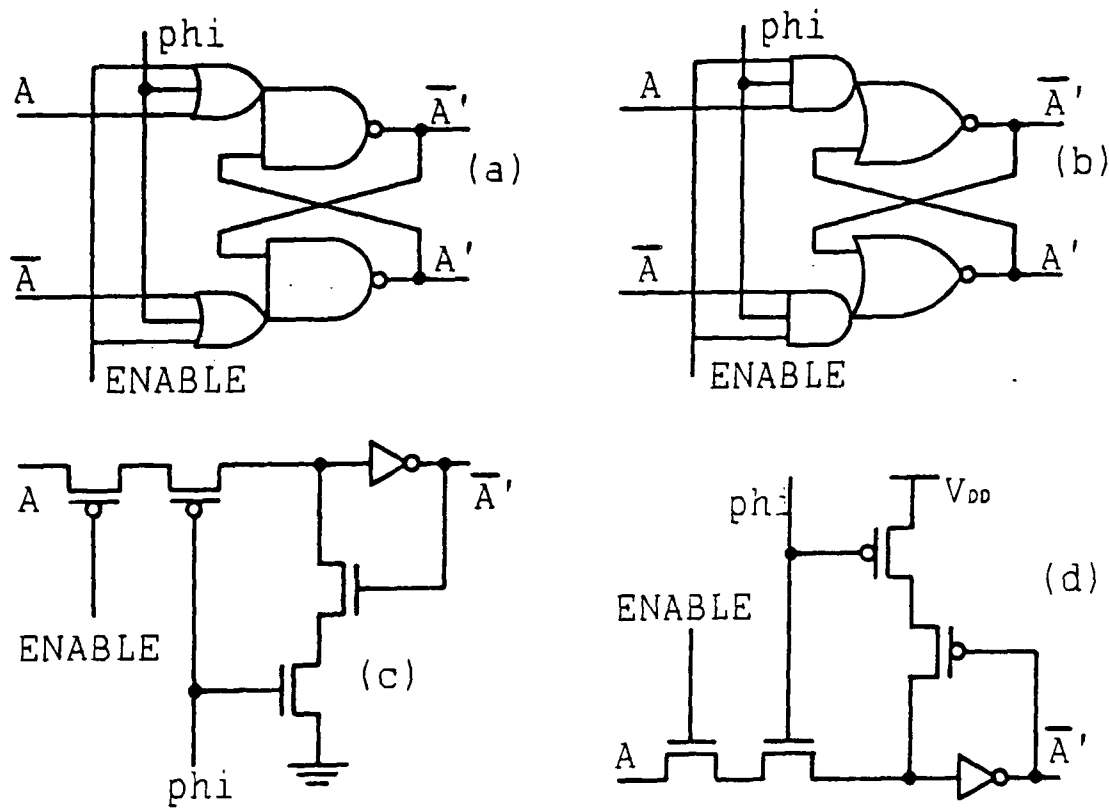


Figure 2.3: static and dynamic holding latches

SO structures it is passed to the neighbour. The difference between a SISO and a SIPO is in the holding action of the latter. A SISO may pass its contents in bit-parallel 'snapshot' form to a PI structure, if the PI structure only requires to read the SISO contents once every word cycle.

2.4.2. Logic

Having described elements for storage and delay, it remains to introduce the three fundamental structural 'atoms' which comprise a serial-data cell-library.

Multiplexers

A multiplexer (see symbol in Figure 2.4) is a simple switching primitive which selects one of its two data inputs as directed by a third, control input. When its output is latched, it forms a 1-bit PISO. As a stand-alone serial-data computational element, the multiplexer is employed at *word*-level, so the control signal may only change state between discrete serial-data input words. The higher levels of control described earlier are used here. However isolated instantiations of multiplexers (with or without latch) frequently find use at *bit*-level in the composition of other

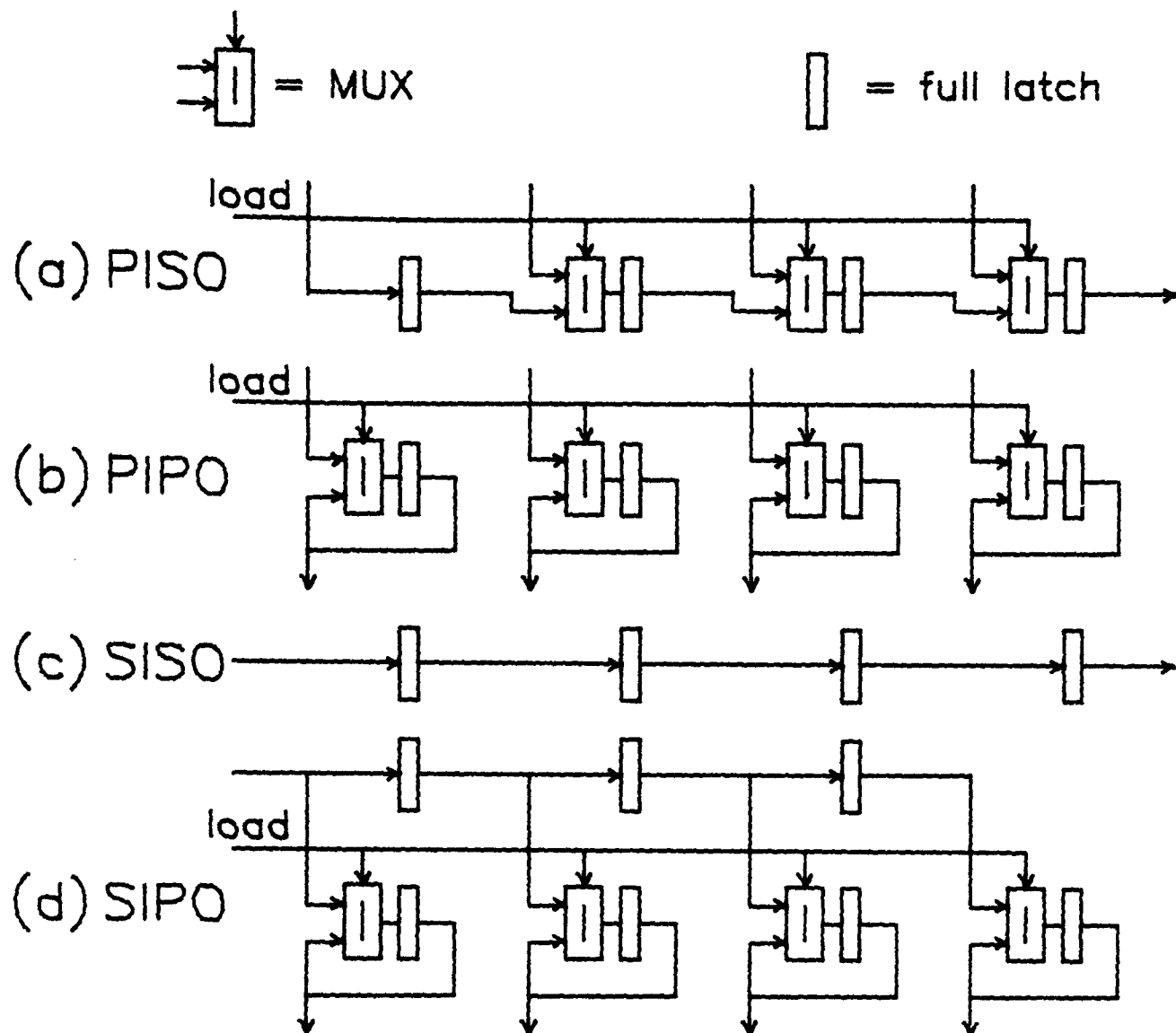


Figure 2.4: register components

functional primitives.

Arithmetic shifters

Arithmetic shifting is required whenever the weight of a word is to be altered with respect to its spatial neighbours. It is not possible to effect this shift merely by delaying some words with respect to others, as this causes a synchronisation error. Hardware must mimic the effect of moving the 'window' described earlier, introducing either sign-repetitions or trailing zeroes depending on shift direction. As they alter the weight associated with individual bits, arithmetic shifters serve as boundaries between 'neighbourhoods'.

2C arithmetic shifting is implemented by a 1-bit PISO, whose control signal decides the shift distance. The parallel input is either grounded for left-shift, or connected to the output for right-shift (the latter structure is in effect a 1-bit PIPO). We illustrate the operation of arithmetic shifting on a stream of 4-bit words $a_i b_i c_i d_i$, where a_i is MSB and d_i is LSB. Operational latencies are represented by the difference in boundary position between words at input and output. Two cases exist:

Right-shift

An arithmetic right-shift of 1 bit implements truncating *division* by 2. A 1-bit PIPO is used (Figure 2.5(a)). The bits of the input operand are left untouched, except for the LSB, which is replaced by the (sign-extended) MSB of the previous word. The weight of each bit at the output is halved in comparison with that at the input. Functional latency of this operation is 1 clock cycle.

input	d_4	a_3	b_3	c_3	d_3	a_2	b_2	c_2	d_2	a_1	b_1	c_1	d_1
output	a_3	a_3	b_3	c_3	a_2	a_2	b_2	c_2	a_1	a_1	b_1	c_1	a_0

Arithmetic right-shifting action

Left-shift

An arithmetic left-shift of 1 bit implements *multiplication* by 2. A 1-bit PISO is used, with grounded parallel input (Figure 2.5(b)). The bits of the input operand are left untouched, except for the MSB, which is replaced by logical 0 (selection of logical 0 may possibly be more efficiently realised by a 2-input gate than by a dedicated multiplexer). The weight of each bit at the output is doubled in comparison with that at the input. Functional latency of this operation is -1 clock cycles. This operation can cause overflow.

input	d_4	a_3	b_3	c_3	d_3	a_2	b_2	c_2	d_2	a_1	b_1	c_1	d_1
output	d_4	0	b_3	c_3	d_3	0	b_2	c_2	d_2	0	b_1	c_1	d_1

Arithmetic left-shifting action

In general, to implement an x -bit shift the multiplexer selects its top input for x clock cycles, starting at the beginning (LSB) of the input word.

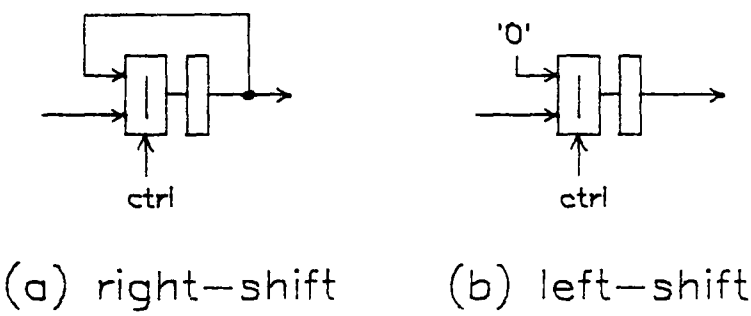


Figure 2.5: bit-serial arithmetic shifters

In the case of the right-shifter, 1 bit of state memory (i.e. a latch) is essential to recirculate the output bit - however this latch need not be situated in the through path. Inclusion of the latch is optional in the case of the left-shifter. Note that the shifters shown in Figure 2.5 have latched outputs - this adds 1 clock cycle to the functional latencies described above.

Adders

So far we have described simple 1-bit registers, and their adoption in the realisation of multiplexing and shifting elements. We now proceed to build in combinatorial logic to the 1-bit registers for the fundamental computational operation of addition.

The task of a bit-serial adder is to assimilate two equally weighted input signals, and produce their sum as output. At bit-level, the sum function is not closed,

i.e. the output may take a numeric value outside the (binary) set of values associated with its input. This requires provision of a second output, the carry, which is given twice the weight of the sum signal. The two outputs (sum and carry) are spatial neighbours, thus the carry must be explicitly left-shifted (as described above) to impart the correct weight. The LSB-control signal instigates this operation. As the weight of incoming data bits in the following iteration is, by earlier definition, double that of their predecessors in an LSB-first computational scheme, the left-shifted carry signal is correctly weighted with respect to its spatial neighbours (the incoming data bits), and may be summed together with them. Provision of the carry output and input stabilises the operation of the bit-serial adder, in that it is now capable of correctly summing all combinations of its 2 external inputs and fed-back carry.

Bit-serial addition hardware consists of a 1-bit PISO and a 1-bit PIPO, with sum-function and carry-function logic respectively built-in. The sum signal is output in SO fashion, while the carry signal recirculates in PO fashion. The sum output is the exclusive-OR of the 3 input operands, and the carry output is their ‘majority’ function (Table 2.1).

Table 2.1: Truth table for bit-serial adder				
Inputs			Outputs	
A	B	C	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The ‘carry-save’ computational style is key to the high performance/area ratio of bit-serial hardware - carry-propagation is temporal (not spatial as in the parallel case), and is naturally pipelined. Figure 2.6(a) shows the bit-serial adder in symbol

form (as used in later examples), and Figure 2.6(b) is expanded to include the left-shifter in the carry path.

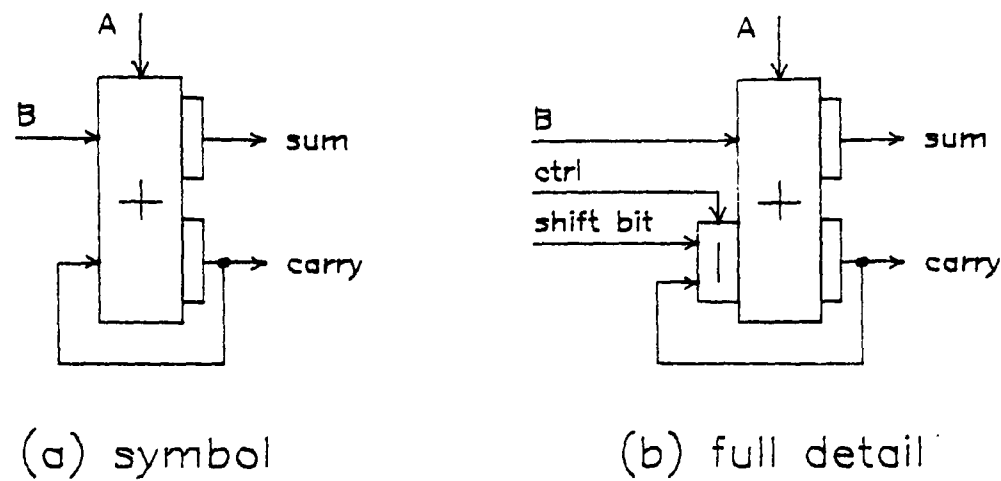


Figure 2.6: bit-serial adder

The 'shift-bit' of Figure 2.6(b) is included for generality - in multi-precision use of the bit-serial adder, the MS-carry out is passed from the neighbouring adder, while in the LS-multi-precision or single-precision case, this bit is logical zero and the multiplexer may be replaced by a simple gate as described above. The more usual single-precision case appears frequently throughout this thesis - multi-precision computation is covered in Chapter 7.

The bit-serial subtracter is very closely related to the adder. In the following section we discuss in greater depth the arithmetic principles behind 2C addition, negation and subtraction.

2.5. Numerical principles of serial-data additive operations

This section details the basic principles of high-performance integer arithmetic which lie behind the design of function libraries for real-time applications. The bit-serial adder has already been introduced, and its bit-level operation described - here the wider, word-level issues of addition, negation and subtraction are addressed.

2's complement addition

Bit-serial architectures employ carry-save adders [44, 45] for performance. As described earlier, a carry-save adder cell has 2 data inputs, one carry input, one sum output and one carry output, where the carry output is given twice the weight of the other signals.

Carry-save adders allow high clocking rates by avoiding the carry-propagation problems which plague conventional bit-parallel adders. In the bit-serial case, carry-propagation reduces to a local recirculation of the carry, with unit delay (a latch) and a shifter in the path. Bit-serial carry propagation is temporal (instead of spatial). Note that word growth of 1 bit is possible under the addition operation. If overflow is to be avoided, addend and augend words should contain one guard-bit (sign-repetition).

2C coding has many advantages, not least of which is the fact that a carry-save adder cell is small and fast. However the MSB of a 2's complement word has opposite weight to the others. This is not a problem in 2C addition (assuming no overflow), as 2C addition is performed as if the two input operands were unsigned numbers [46] - no correction is required for MSB weighting. The MS-carry (overwritten in the left-shifting process) is simply discarded.

2's complement negation

Negation in 2C may be performed by bit-inversion (1's complementing), and adding logical 1 at the LSB position (incrementing) [47]. To see this, consider eqn. 2.1 with inverted bits:

$$\begin{aligned}\bar{A} &= -(1 - a_0) + \sum_{i=1}^{n-1} (1 - a_i) 2^{-i} \\ &= a_0 - \sum_{i=1}^{n-1} a_i 2^{-i} - 1 + \sum_{i=1}^{n-1} 2^{-i} \\ &= -A - 2^{1-n}\end{aligned}$$

i.e.

$$-A = \bar{A} + 2^{1-n}$$

Now consider the case where the bottom z bits of input operand A are zero.

$$\bar{A} = -(1 - a_0) + \sum_{i=1}^{n-1-z} (1 - a_i) 2^{-i}$$

By change of variable from n to $m = n - z$, the above derivation may be applied once again, implying that negation may be accomplished by passing trailing zeroes then inverting all remaining bits, with the increment applied along with the first non-zero input bit. Furthermore, incrementing the (inverted) non-zero bit results in a non-zero output - thus the algorithm may be further modified to pass all trailing zeroes *and* the first non-zero bit, then invert all remaining bits. JKM proposed just such a building block [26], where the input stream is used to set a flip-flop, which in turn controls a switch which selects either inverted or non-inverted input bits as output.

Another method of achieving negation is the modified bit-serial adder of Figure 2.7, where the sum bit recirculates as well as the carry.

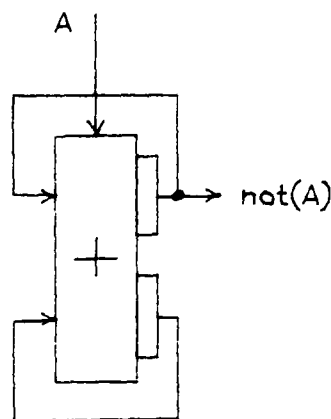


Figure 2.7: bit-serial negator as adder with fed-back sum

In this structure, recirculating bits are zero until the first non-zero input bit arrives. This bit produces a non-zero sum, which is output and recirculated. Subsequent input bits if zero will produce a recirculating non-zero sum-bit, or if non-zero will produce a non-zero recirculating carry-bit, but never both. The effect of one (and

only one) recirculating non-zero bit is to invert the input stream, via the sum function. Thus this structure mimics the JKM negator. Note that in Figure 2.7 a left-shifter is implicit in the sum path as well as the carry path.

2's complement subtraction

The operation of subtraction is so similar to that of addition that it may be regarded as a 'twin' to the adder building-block. From the operation of 2C negation initially described above, it follows that 2C subtraction (the operation $B - A$ in Figure 2.8) may be performed by negating the subtrahend, then adding. This technique is referred to as *implicit* subtraction [48][†], as the structure is really a modified adder (Figure 2.8(a)). The recirculated carry signal still has positive weight.

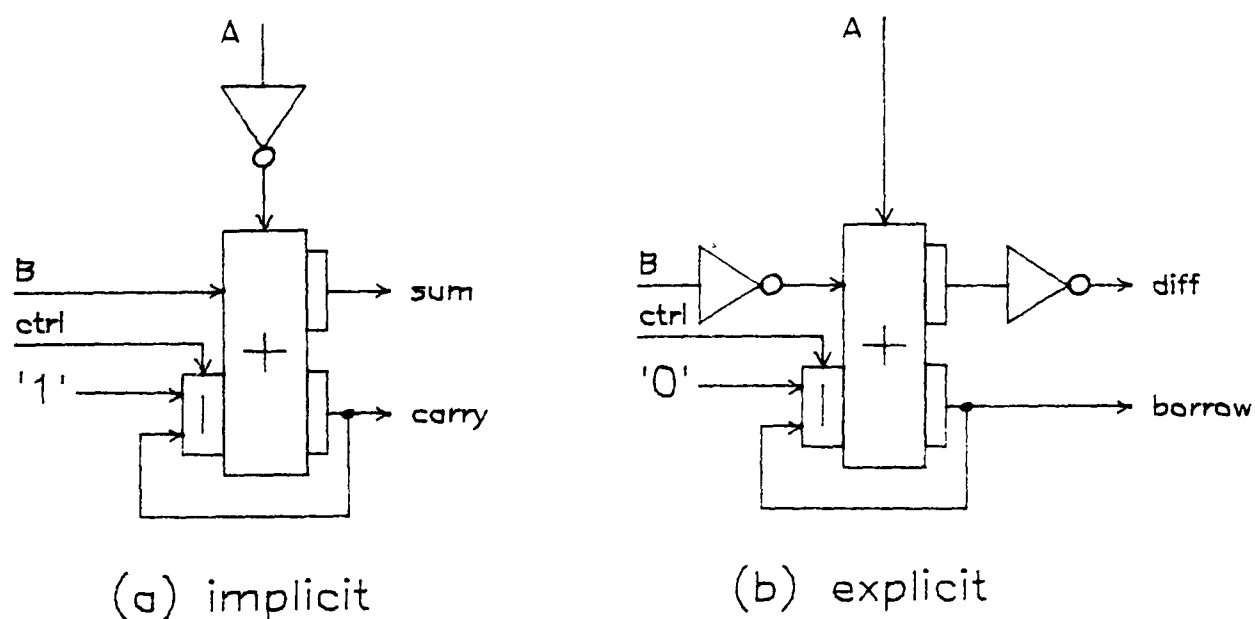


Figure 2.8: (a) implicit, and (b) explicit subtracters

To implement a true, or *explicit* subtracter [48], the augend (minuend) and sum (difference) signals in the standard carry-save adder must be inverted (Figure 2.8(b)). Incrementing is not necessary, as the incorrect internal representation caused by the first inversion operation is cancelled by the second. Here the recirculated carry (i.e. borrow) signal has negative weight. Should sum and carry signals be generated by separate logic networks, the conversion from an adder to an

[†] published work by the author.

explicit subtracter is accomplished by inverting the augend input to the carry network. Sum and difference functions are identical.

It is important to differentiate between the two subtracter types, as the weight of recirculating carry/borrow signal can be crucial in certain cases. Carry/borrow conflicts must be avoided when passing carries to other logic blocks, as standard carry-save adder/subtracters can only handle binary carries.

Addition with 3 or more inputs

Sometimes it may seem desirable to perform addition on more than two input variables. The bit-serial adder described above, with 3 inputs and 2 outputs (including carry signals) cannot absorb a further input operand, as an input pattern of all ones would cause overflow. This problem is solved by introducing a second carry output, with a relative weight of twice the original carry out, and four times the sum out. This new output demands the incorporation of an extra input, as two carry signals are now associated with each stage. The result is a 5-input, 3-output adder, sometimes known as a (5,3) counter. This structure can be extended to accept up to 7 inputs - in general a counter with $\log_2 n$ outputs may handle up to $2^n - 1$ inputs. Swartzlander [49] and Meo [50] have formalised the theory of counter network synthesis, which has important consequences in the design of bit-parallel multipliers.

In the highly-pipelined operational environment of a serial-data architecture, the increased complexity of these structures gives cause for concern, as it is unlikely that they can sustain as high a clock rate as the (3,2) counter.

2.6. Partitioning issues

Along with latches, the three atomic logic blocks (multiplexer, shifter and adder) described earlier are sufficient to realise a comprehensive bit-serial arithmetic cell-library. However if chips are to be assembled automatically, conveniently and efficiently from cell-library elements, some pre-packaging should be performed as a one-off exercise by a logic designer. In addition to efficient *functional* partitioning, the identification and provision of higher-level functional primitives as *system-level* building-blocks leads to efficient *physical* partitioning, through the

deployment of appropriate local assembly procedures in each case. Physical partitioning strongly affects performance and area-efficiency.

Chapter 3

Two's complement serial-data multiplication

Germane to the realisation of fixed-function, real-time computational systems is the operation of multiplication. Reflecting its importance in the context of this thesis, this chapter addresses the historical and theoretical background to serial-data multiplication. A short (and by no means comprehensive) overview of bit-parallel multiplication is included for contextual reasons. More detailed treatment of the subject may be found in [47, 51, 52, 46, 53].

A multiplication is an unconstrained two-dimensional sum of weighted cross-products of the constituent bits of two input operands (often known as *data* and *coefficient* words). Figure 3.1 shows the matrix of bit-product formation values in the multiplication of two 4-bit binary integers $a_0a_1a_2a_3$ and $b_0b_1b_2b_3$, where a_0 and b_0 are MSB, after eqn. 2.1.

	a_3	a_2	a_1	a_0
b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3
b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2
b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1
b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0

Figure 3.1: matrix of bit-product values (4×4 -bit example)

If a partial product (PP) is defined as the product of one input word with a single bit from the other, then a multiplication is an unconstrained one-dimensional sum of weighted PPs. Note that two classes of PP may be defined, as the multiplier bit may derive from either data or coefficient. These correspond to rows and columns of the bit-product matrix (Figure 3.1).

3.1. Derivation from bit-parallel architectures

Bit-parallel multiplication may be executed on $n \times m$ rectangular arrays of gated adders, each row forming a PP, adding it to the partial product sum (PPS) passed to it from the previous row, and passing the new PPS to the next row. In general, one processor exists for each matrix element in Figure 3.1, the task of which is to form the designated bit-product, and sum with the PPS bit of equivalent weight from the previous row. This function may be achieved using a gated full-adder cell.

Carry-save adders may be employed to speed up PP computation in rows [54]. Here carry outputs connect forward to carry inputs of higher-significance cells in the next row, while sums are passed as before - this avoids carry propagation in rows. However after the last row, recombination of sums and carries inevitably involves carry propagation in a bit-parallel adder.

Certain means may be employed to further speed the operation of bit-parallel multiplication. Various levels of pipelining [55, 56] may be utilised - resulting in impressive throughputs in the extreme [57]. PPs may be reduced in number by modified-Booth recoding [58, 59], or further-modified Booth recoding [60] the coefficient. Alternatively, concurrency of PP summation may be increased by the use of (5,3) adders [61], or by merging in tree-structures. The Wallace tree [62] merges 3 PPs into 2 using carry-save techniques, while 2 PPs may be merged into one using redundant-data addition, e.g. signed-digit [63] or twin-carry [64].

Each of these implementations must face the carry-propagation problem in some form before eventual product formation. Even in combinatorial array multipliers [65] this overhead is considerable, but in pipelined implementations the carry-propagation problem is acute, demanding area-expensive fast adders to maintain performance. In the case of fully-pipelined 'bit-systolic' architectures [66], the hardware overhead for carry-propagation is around 50%. Triangular array architectures have been proposed, which pipeline carry-propagation in the product [67, 68]. However the added complexity of cells throughout the array cancels the advantage of pipelined carry-propagation in triangular arrays.

In the simplest sense, serial-data multiplication architectures are a mapping of

the two-dimensional bit-parallel multiplier into a linear (one-dimensional) array. The second dimension becomes that of time. Only the well-structured bit-parallel architectures undergo such a mapping with any elegance - the standard rectangular array of carry-save adders is an ideal candidate, although the modified-Booth algorithm has been employed in serial-data multipliers to effect savings in both time [69] and area/latency [44, 7].

3.2. Scrutiny of two serial-data multiplication architectures

A set of serial-data multiplication architectures, based on the linear carry-save add-shift (CSAS) arrays described above, has received greatest attention from the hardware design community. This set was formally catalogued by Danielsson [70]. From it, two main architectures have emerged to dominate 2C applications - these are the serial-pipeline (Lyon) multiplier [44], and the serial/parallel (S/P) multiplier [71]. We describe the operation of these two architectures, highlighting their respective merits.

In the simplest sense, the function of each multiplier is to load the coefficient word (1 bit per hardware stage), broadcast the data word in LSB-first serial form, form partial products (PP) by gating with the resident coefficient bit in each stage, and shift-accumulate the PPs in carry-save form to compute the product. However the methods employed to achieve this end differ greatly between the two architectures. We begin by tabulating the operational similarities, assuming for now that coefficient bits are resident in their respective locations.

Similarities

In each case, the computational hardware consists of a linear array of CSA cells, with each sum output connected through an arithmetic shifter to one input of its downstream neighbour. In each case, the carry signal recirculates locally. In each case, the coefficient word is distributed in ordered fashion throughout the array, one bit per stage. In each case, a PP-bit is formed in every clock cycle by the AND function of the passing data bit and the resident coefficient bit. Finally in each case the cell at the end of the array in which the coefficient MSB is stored contains hardware to account for the negative weight of the 2C coefficient MSB.

Differences

In many respects the multipliers exhibit behavioural duality - this allows contrasts to be succinctly highlighted. In the following discussion, a concise notation is employed to highlight differences between S/P and Lyon multipliers. Following the notation of [72], references to the Lyon multiplier are enclosed in curly brackets immediately following references to the S/P.

The fundamental difference between the two multiplier types may now be revealed. In the S/P {Lyon} multiplier, the coefficient word is latched in the same {opposite} sense as the PPS signals in the array. Here 'sense' means 'order of decreasing bit-significance' - if PPS flow is left-to-right then the S/P {Lyon} latches the coefficient with LSB to the right {left}. Arithmetic shifting hardware gives each cell (and hence each coefficient bit) an associated 'static' weight - in the S/P {Lyon} multiplier this weight decreases {increases} in the direction of PPS flow. The shifting hardware is essential to impart weight to the coefficient bits, which would otherwise be spatial neighbours.

3.2.1. Bit-serial derivation of the architectures

The operational aspects of the two multiplier types are by no means trivial - their clarification requires detailed architectural analysis. The derivation is in terms of 'serial partial products' (SPP), where an SPP is defined as a bit-serial word which represents the product of a coefficient bit with a bit-serial data word. Given that carry signals recirculate locally and are correctly weighted as demonstrated earlier, they may be ignored for the time being.

The task of each spatial unit (computational stage) is in this case to add two words (one is the SPP, one is the PPS). Each cell consists mostly of a stand-alone bit-serial adder, which produces the sum of its two input words (local PP and 'upstream' PPS) to be passed as 'downstream' PPS to the next stage in the array. A fundamental serial-data requirement is that input operands to any computational element be correctly synchronised - this restriction reduces the distribution options (in time and space) of coefficient and data bits.

Shift-accumulation

In order to accomplish correct shifting accumulation, it is not enough merely to offset one adder input operand by one bit with respect to the other. Such an action causes the LSB of one word to be added to the MSB of the other. Synchronism at word-level is violated. For correct operation, arithmetic shifting hardware (as described in Chapter 2) must be provided between stages. The desired sense of coefficient storage decides which type of shifter should be inserted.

In the S/P {Lyon}, bit-significance decreases {increases} in the direction of PPS flow, therefore the PPS input to each adder should be of greater {lesser} significance than the PP input, and must be left- {right-}shifted accordingly. The functional delay through these elements - known to be -1 { 1 } bits in the S/P {Lyon} - affects the synchronism of downstream hardware. In the S/P {Lyon} multiplier, the functional delay of the shifter cancels {doubles} the delay through the previous adder. The synchronism of adder inputs may then be maintained by broadcasting data bits to the array with 0 { 2 } bits of delay per stage in the data path. Figures 3.2 show the matrices of bit-product formation *times* of S/P and Lyon multipliers, and Figure 3.3 the computational element at each stage.

	a_3	a_2	a_1	a_0
b_3	1	1	1	1
b_2	2	2	2	2
b_1	3	3	3	3
b_0	4	4	4	4

Figure 3.2(a): matrix of bit-product formation-times, S/P multiplier

Overflow handling

Together with the use of 2C adders in each stage comes the possibility of numerical overflow. Separate strategies are employed in each multiplier to avoid overflows. Recall that the Lyon multiplier right-shifts the PPS word in each stage,

	a_3	a_2	a_1	a_0
b_3	1	3	5	7
b_2	2	4	6	8
b_1	3	5	7	9
b_0	4	6	8	10

Figure 3.2(b): matrix of bit-product formation-times, Lyon multiplier

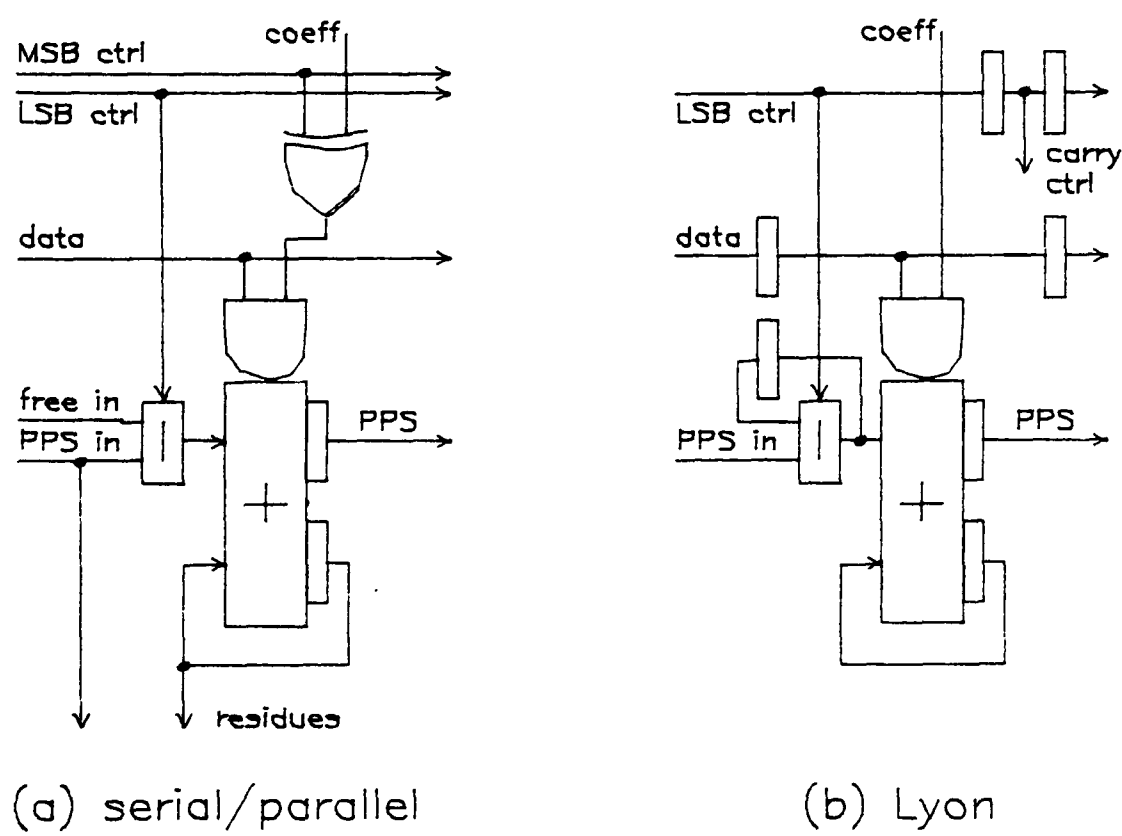


Figure 3.3: (a) S/P CSAS cell (b) Lyon CSAS cell

thereby ensuring that the PPS contains a guard-bit (sign-extension) on entry to the downstream adder cell. By imposing the constraint that the data word - and as a result the SPP - also contain one guard bit, it is ensured that normal 2C addition rules are obeyed at each adder in the array. Overflow conditions cannot arise, and the MS-carry may be discarded at each stage in the standard fashion of 2C addition. To circumvent the data guard-bit constraint, expensive overflow detection and correction circuitry must be employed [73].

The S/P multiplier left-shifts the PPS word in each stage, thereby inviting the possibility of overflow even before the addition operation. To counteract this

possibility, an extra addition operator is provided to process the PPS MSB (which is lost to the adder through the left-shifting process). Thus the bit-serial adder contained in each stage does not complete the calculation - it passes its final carry (the effective MS-carry-in of the full SPP-PPS sum calculation) to the extra addition stage. To complete the computation of the SPP, this carry signal is combined in the extra 'residue' adder with the final sum output from the upstream stage adder. Depending on the operational environment (shortly to be discussed) the residue adder may be a stand-alone serial-data adder (fed by a PISO), or a parallel carry-propagate adder (fed by a PIPO). Indeed should enough data guard-bits be provided, the overflow problem and the requirement for residue addition circuitry may be obviated [74].

MSB treatment

As detailed above, the MSB of a 2C word has negative weight, in contrast to the other bits. A property of 2C addition is that the MS-carry may be discarded, allowing the MSB to be treated in identical manner to the others. Multipliers may not always enjoy this favourable property, however.

The negative weight of the coefficient MSB may be handled simply by using a subtracter to derive the SPP associated with this bit. The data MSB, however, is more problematic. Booth's algorithm was proposed as a method of circumventing the data MSB-weight problem - by recoding data in ternary, symmetrical form (i.e. data bits may take the value 0, ± 1), the MSB can be processed in the same manner as the others [75]. Modified-Booth (MB) coding (data bit-pairs are recoded into 5-level, symmetrical form 0, ± 1 , ± 2 [58]), and offset binary (OB) coding (bits take value ± 1 [76]) also bring this advantage. Without resorting to recoding techniques, the Lyon multiplier handles the data-MSB with ease - as each adder computes the *complete* sum of the local SPP and the right-shifted upstream PPS, the MS-carry may be discarded in the standard fashion at each stage.

In the S/P multiplier, however, each adder computes the *incomplete* sum of the local SPP and the left-shifted upstream PPS. The final iteration is passed in the form of sum and carry residues to the residue adder described above - as the final carry from the stage adder is retained it must therefore be correctly weighted. To

accomplish correct weighting, genuine unsigned integer arithmetic must be performed throughout, which is hampered by inconsistently erroneous interpretation of the 2C-coded SPP MSB in the stage adders. This problem may be eased by inversion of the MSB, fixing the MSB error in the SPP at a constant one bit, independent of the value of the MSB. Computation may now proceed as if the adder inputs were unsigned. All carry signals generated are recombined with equivalently-weighted sum signals in the residue adder, except for that generated by the MS-SPP calculation, which may be discarded in the standard 2C fashion.

The effect of introducing a constant single-bit error throughout the array is to introduce a constant error in the product. This error may be viewed as a bit-parallel, 2C word consisting of all one's. As negation is achieved by bit-inversion plus increment, the additive inverse of the bit-parallel error is all zeroes, except the LSB which is one. Accordingly, cancellation of the error word is easily accomplished by incrementing the residue sum, i.e. by setting the LS-carry-in of the residue adder.

Control

The shifters contained in each multiplier require control, as described in Chapter 2. The control signal is broadcast to the array in along with the LSB data word, i.e. the control path is identical to the data path. The control signal is also used in data loading and unloading operations.

In the S/P multiplier, the MSB-inversion process requires a second control signal, in synchronism with the data-MSB (Figure 3.3(a)). If this signal is supplied externally, the LSB-control may be derived internally from it by simple bit-delay.

Output format

Each multiplier is capable of producing a full-precision product from m -bit coefficient and n -bit data (in general $m \leq n$). The full product is $n + m - 1$ bits long, i.e. word-growth by $m - 1$ bits is possible in the multiplier. In the S/P {Lyon} multiplier, the top {bottom} m bits of the product appear in parallel form throughout the array at the end {beginning} of each stage sum calculation (these are the bits which are overwritten in the arithmetic shifters). The bottom {top} $n - 1$

bits are output serially from the end of the array.

In many applications full-precision products are required - this may be achieved by formatting the product output as a double-precision fractional word, following bit-serial conventions for multi-precision word formatting [7] where multi-precision words are arranged in staggered form on multiple wires. Low-order bits are naturally output in serial form from the S/P multiplier, while in the Lyon those low-order bits lost in the shifters must be stored and shifted out in serial form - extra hardware must be included for this purpose [73, 77]. In both cases, steering logic and associated control must be provided.

In a single-precision fractional bit-serial environment, only the top n bits of the product are of interest - the product format is the same as that of the input data. The simplest way to achieve this is by truncation, i.e. discarding the bottom $m - 1$ bits of the product. The multipliers described above operate in this manner. As they introduce negative bias into representational errors in products, single-precision truncating multipliers have limited application in the real world.

More usual is the single-precision rounding mode (whose attendant representational error is almost exactly zero-mean). Rounding may be achieved by adding one half-LSB into the single-precision product. In the S/P multiplier, rounding may be achieved by setting (rather than clearing) the PPS input to the 2nd-MS stage at LSB-time (i.e. incrementing the left-shifted PPS output word from the topmost stage). This operation is independent of coefficient length.

At first sight, the Lyon cannot so easily assimilate the rounding bit, requiring hardware (in the form of a delay line driven by the LSB-control signal) at the front of the array to insert this bit, appropriately timed, at the free sum input to the array [78]. Although it has no free sum input at the half-LSB position, the Lyon multiplier does have a free carry input (normally cleared at LSB-time). This bit may be set to accomplish rounding operation.

Computational latencies

In some DSP applications, for instance those which involve short, high-speed computational loops, the computational latency (i.e. the time lapse between arrival of the input LSB and the appearance of the output LSB) of multipliers may be crucial. Examples include recursive [79] and wave digital [80][†] filters. It is important to point out the different uses of data and coefficients in DSP, as data and coefficient latencies are not necessarily identical. Firstly, the coefficient is often known *a priori* (unlike data which either arrive from external sources or are freshly-derived from previous computations). Secondly, the precision of representation may differ (the coefficient is often represented by fewer bits).

In nearly all cases the critical latency parameter is *data* latency. Assuming fractional operation, the overall data latency (i.e. the time lapse between arrival of data LSB and appearance of fractional product LSB) of each multiplier is approximately the sum of the latency in the PPS path ($0 \{2m\}$ bits for the S/P {Lyon}) and the distance of the MS-stage from the input end of the array ($m \{0\}$ bits for the S/P {Lyon}). These quantities are approximate in that they neglect unit hardware costs, e.g. pipelining latches, output buffers etc.

The quantity of *coefficient* latency, which is deemed less critical, is in fact identical at $2m$ bits for both multipliers, given the above approximations. The reason for the discrepancy in S/P data and coefficient latencies is that, while the Lyon multiplier accepts data and coefficient simultaneously, the S/P requires advance loading of the coefficient, adding m bits to the coefficient latency figure.

Free inputs

It is well known that use may be made of free inputs to multiplication hardware, to achieve accumulation 'on the fly' [45, 81]. The use of free inputs to achieve rounding operation was described earlier. However if rounding is foregone or postponed, n -bit serial words and/or m -bit parallel words may be accommodated.

Both multipliers clear their carry-loops at LSB-time. The carry-loops offer the

[†] published work by the author.

potential for a free input if, instead of clearing, these loops load the bits of some parallel input operand. The Lyon multiplier has one free serial input, whose weight corresponds to the *bottom* n bits of the full-precision product. As n and m differ in general, this input is of limited use. In contrast the equivalent free serial input to the S/P multiplier carries a weight which corresponds to the *top* n bits of the full-precision product, in other words an external fractional serial operand may be directly absorbed in a multiply-accumulate operation. The S/P multiplier has the added advantage of a second free parallel input (Figure 3.3(a)), as the left-shifters in the sum-path clear at LSB-time under normal circumstances, in identical manner to the carry-loops. This opens the possibility of performing multiplication on multi-precision data words, by passing intermediate results between multipliers in carry-save form.

Regularity and locality

These properties are generally accepted as desirable in the context of VLSI circuits [41]. Both multipliers may be constructed by cascading (almost) identical hardware stages. Each must contain a subtracter to account for the weight of the 2C coefficient MSB. However, at the cost of the free serial input, the S/P multiplier may in fact be constructed from identical hardware stages, the only difference being that the sum output from the MS-stage is fed back to its input to form a negator, as described in Chapter 2. The Lyon multiplier must contain a subtracter (explicit or implicit) in its MS-stage.

The Lyon multiplier is however truly local, as all signals are distributed locally in time-skewed manner. Lyon multipliers of arbitrary length may be composed by simple abutment of the repeated main stage - only the clock and power supplies are global. The S/P multiplier broadcasts data and control signals, and 'real-world' physical effects such as the rise time associated with charging/discharging long metal lines [82] may limit the size and performance of S/P multiplier realisations in practice. Pipelined buffers must be provided to drive these long lines, adding to the size, power consumption and latency of S/P multipliers. These also destroy regularity (another irregularity in the S/P case is the residue adder).

Serial-data loading operations

In the S/P multiplier, the coefficient word drops into the coefficient register in parallel fashion at data LSB-time (similarly the high order product bits drop out at MSB-time). Thus the entire coefficient must be available in bit-parallel form at data LSB-time. In a serial-data environment, the coefficient must be clocked into place during the preceding calculation, i.e. it must be known in advance of the associated data word.

The Lyon multiplier on the other hand is able to accept the coefficient in alignment with the data - it is a truly serial multiplier. By inserting one bit of delay in the coefficient path, and two bits in the control path, the coefficient stream is effectively reversed, dropping into place in a time-skewed manner as the control pulse moves through the array. Thus, in the Lyon multiplier, more than one computation may be in process at one time.

Serial-data unloading operations

Whether in single- or double-precision mode, the fractional S/P multiplier requires twin PISOs to clock the sum and carry residues through the residue adder. These PISOs are loaded at LSB-time in the following product calculation - the same LSB-control signal may be used to load the PISOs.

The Lyon multiplier only requires unloading hardware in double-precision mode. Here the low-order bits are unloaded in similar fashion to the loading of the coefficient bits - a PISO is provided which is loaded by the LSB-control in time-skewed manner, the result of which is correct ordering of the low-order product bits at the exit of the PISO.

3.2.2. Bit-parallel derivation of the S/P architecture

As references to bit-parallel entities have cropped up in recent paragraphs with increasing frequency, it is prudent to look at the S/P architecture from a different viewpoint - that of parallel partial products (PPP).

A PPP is defined as a bit-parallel word, representing the product of a data-bit

with a bit-parallel coefficient word. The operation of the S/P multiplier, along with the reason for its name, now becomes clear. The computational engine consists of a linear array of carry save adders, which acts as a bit-parallel right-shifting accumulator or CSAS computer. Shifting is accomplished by simple abutment of cells, in accordance with the implicit spatial weighting of bits in a bit-parallel word, as described in Chapter 2. This accumulator is cleared at LSB-time, and gates are provided for this purpose in both sum and carry paths, activated by the LSB-control signal. The data-word is broadcast to the array in LSB-first serial fashion, and bit-parallel PPs are formed and accumulated. The last PPP has negative weight, and accordingly is implicitly subtracted from the PPS (instead of being added). Implicit subtraction requires bit-inversion of the parallel word, as directed by the MSB-control signal. Finally (as in the bit-parallel case) the sum and carry residues must be combined - in this case by clocking through a bit-serial adder. Correct implicit subtraction is ensured by setting the carry-loop of this adder.

Although this description of the S/P multiplier is in a sense more natural and intuitive than the SPP description (particularly with regard to data-MSB treatment), it was instructive to compare the operation of the two seemingly disparate multiplier architectures, highlighting their construction and behaviour in a unified theoretical framework. Unfortunately the Lyon multiplier defies description in terms of PPPs. When it comes to developing more advanced computational architectures in later chapters, we shall use abstractions of the PPP model and CSAS computer.

3.3. The serial/parallel flush multiplier

A full-precision variation of the S/P multiplier [74] dispenses with a considerable portion of the hardware contained in the fractional model. No such variation exists in the Lyon. This multiplier - dubbed the S/P 'flush' multiplier and depicted in Figure 3.4 - allows word growth by $m - 1$ bits in the product, accordingly $m - 1$ guard bits must be provided in the input data. In applications where the loss of throughput resulting from unnecessary processing of guard-bits may be tolerated, hardware savings are considerable.

First of all, the twin PISOs and residue adder are no longer required, as the full-precision product appears in serial form at the end of the array. Furthermore,

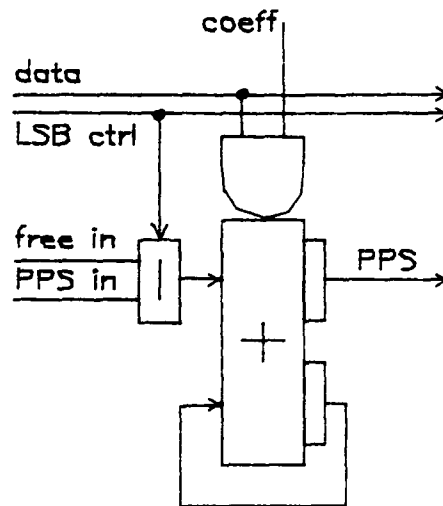


Figure 3.4: S/P flush multiplier CSAS cell

the MSB-control signal is also no longer required, as unsigned integer arithmetic is protected from incorrect interpretation of the data-MSB (the harmful effect of the data-MSB cannot propagate to the serial output of the array). The result of the latter improvement is that the computational elements in each stage, having less inputs and simplified internal function, might be expected to sustain higher clocking rates than the equivalent in the fractional S/P multiplier, recovering at least some of the throughput lost.

3.4. Word-level equivalent architectures

Systolic array architectures have received considerable attention in recent years. They too exhibit regularity, modularity and locality, in accepted VLSI practice. H. T. Kung, a major proponent of the systolic array concept, has outlined linear systolic array architectures for convolution and Fourier transformation, whose word-level operation is remarkably similar to the bit-level operation of the S/P and Lyon multipliers. An architecture for computation of the DFT by Horner's method [83], or more accurately its modification by Allen for continuous operation [84], is directly analogous to Lyon's multiplier. Meanwhile the systolic FIR-filter [85] (the erstwhile 'iterative computational network' [86]) with broadcast data vector, moving output vector and static coefficient vector, exactly parallels the operation of the S/P multiplier. The only fundamental difference is in the implicit weighting of input operands - in the bit-level case this weight increases in factors of 2, requiring

arithmetic shifters at each stage for correct operation.

3.5. Comparison of the S/P and Lyon multipliers

There are advantages and disadvantages associated with the use of both multipliers, and a comparison is in order. The comparison is complicated by the existence of two greatly contrasting S/P multiplier types (fractional and flush).

Area

Inspection of Figures 3.3 and 3.4 reveals the 'computation' latches, required for correct internal operation of the CSAS computers. Not shown are the 'communication' latches - these comprise (in all cases) 2 in the coefficient SIPO, and (in the fractional S/P case only) 2 more in the residue PISOs.

The Lyon multiplier is the most area-expensive - this is due mostly to the 2-bit delays in the data and control paths in every stage. A latch count in each stage reveals that the Lyon requires 9 latches, and the S/P 6. The S/P stage adder (requiring MSB-treatment) is slightly more complex than the Lyon, requiring more transistors for realisation. Furthermore, the S/P exhibits 'one-off' costs such as the residue adder and the data and control signal broadcast buffers. The clear winner in area is the S/P flush multiplier, whose stage adder is similar to the Lyon and whose latch requirement is only 4.

Throughput

The Lyon multiplier might be expected to exhibit better throughput, for two reasons. Firstly, as it handles data MSB in similar fashion to other data bits, it does not require circuitry for MSB processing. Computational circuits are correspondingly smaller, and therefore faster. Secondly, all communication is local - no rise times due to long wires hamper performance. The S/P flush multiplier shares the former advantage.

Dynamic range

Due to its use of the extra residue adder, the S/P multiplier is capable of handling full-range data. The Lyon on the other hand requires 1 guard-bit, or must suffer the time and area costs of overflow detection and correction. The S/P flush multiplier appears to come a poor third here, demanding m guard-bits for correct operation. However it could be argued that, by presenting the full product, the S/P flush multiplier in fact exhibits superior dynamic range (and inferior throughput) to the fractional multipliers. It must be concluded that the parameters of throughput and dynamic range are strongly linked.

Data latency

The S/P multiplier is considerably less latent than the Lyon, although the latter latency may be shortened via the modified-Booth algorithm. Use of the MB algorithm does not affect the S/P latency. It should be noted that the Lyon latency figure as presented above is more accurate than that attributed to the S/P, as the Lyon multiplier features local signalling and direct output from the array. A more accurate figure for the S/P latency is $m + 3$ bits - here we account for pipeline-buffered data broadcast and residue addition. Thus latency differences are less dramatic at low m .

Regularity

The Lyon multiplier requires a subtracter cell to handle $2C$ coefficient MSB weight. The S/P only requires the equivalent subtracter when supporting multiply-addition (otherwise it uses the negator circuit, which consists of a standard adder with fed-back sum). However the one-off circuits (e.g. residue adder) in the S/P multiplier conspire against regularity. The S/P flush multiplier without addend input is perhaps the most regular of all.

Locality

Clear winner in this case is the Lyon, exhibiting genuinely local communication throughout. The S/P multipliers suffer from the requirement to broadcast data and control signals (the 'through-wires' in Figures 3.3(a) and 3.4).

Free inputs

Clear loser in this case is the Lyon, with one free parallel input and one free serial, both at low significance. The S/P multiplier has two free parallel inputs, and one free serial at fractional significance. Multi-precision operation and multiply-addition are practicable in the fractional S/P case.

Rounding operation

Rounding is easily accomplished in either architecture, by incrementing in the second top stage. The cost of rounding is a slight decrease in regularity, and compromise of a free parallel input.

Double-precision operation

The fractional S/P multiplier is best suited to conversion to double-precision operation, as the PISOs required for clocking out parallel residues are already in place. All that is required is a final multiplexer for selecting high and low-order output streams. The $m - 1$ low-order bits, normally rejected in fractional operation, are available free of charge at the array output. These signals form the parallel residue in the Lyon multiplier - an extra PISO must be included if they are to be made available externally [73, 77].

Summary of the comparison

It seems from the above arguments that the S/P multiplier (with its choice of forms, lower latency, lower latch count and free inputs) is slightly more desirable than the Lyon. However that is not to say there is no place for the Lyon multiplier - its properties of locality and logical simplicity make it the choice for many applications, particularly those involving high performance multiplication by long coefficients. For the remaining chapters the S/P multiplier (and its CSAS computer) will act as basic multiplicative building-block, however most of the concepts to be introduced are equally applicable to the Lyon.

3.6. Serial/parallel multiplier environments

S/P structures find application in both serial-data and parallel-data architectures. We have studied in detail the former environment (Figure 3.5(b)) - now it is instructive to examine the latter.

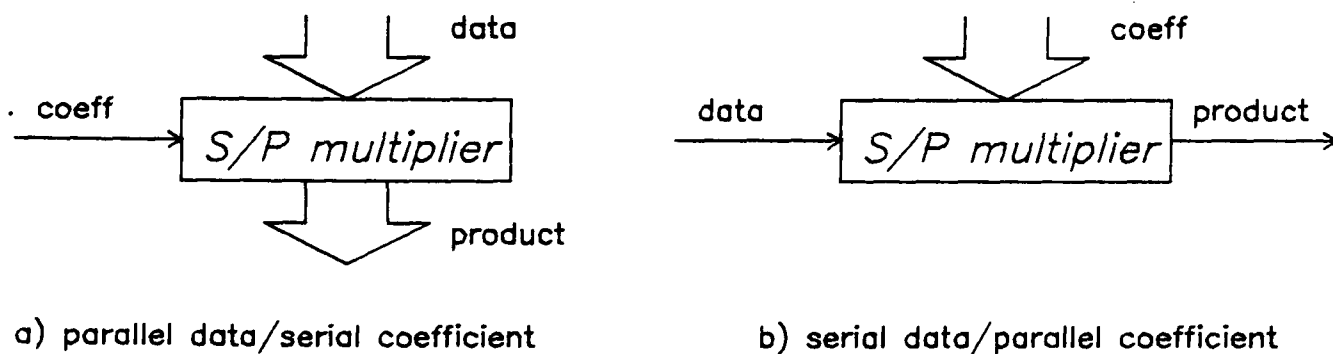


Figure 3.5: serial/parallel multiplier environments

In the bit-parallel, datapath environment (Figure 3.5(a)), one of the input operands enters a PISO to be broadcast to the array. When computation is completed, the carry-save product residue loads a parallel adder to be merged into a single word (this may be overlapped with the next product calculation). However the final (MS) PP must be subtracted to account for 2C MSB-weighting in the data - a control signal and some extra circuitry must somehow be provided for this purpose. Examples of this architecture appear in [87, 88] and (using modified-Booth coding to double throughput) [89, 69].

3.7. Other approaches

Before discussion of more advanced serial-data computation, a review of some other approaches to serial-data multiplication is appropriate. Serial-data multiplication has a rich history, dating back to the iterative, discrete-state automata of Atrubin and Knuth [90, 91]. These machines remained in the realms of theory, due mostly to the complexity of the automata (each had 11 inputs).

In 1968 JKM proposed a forerunner to the Lyon multiplier [26] - at the same time multiplier designs based on the shift-add principle began to appear [71]. By the mid-seventies, silicon realisations of these architectures were becoming familiar

[92, 93, 94, 95], and a standard-part S/P multiplier soon arrived in the marketplace [89].

3.7.1. Quasi-serial multiplication

In 1973 Swartzlander published an unusual serial-data multiplication architecture, based on the use of analog techniques for fast counting of ones in the columns of the bit-product matrix [96]. This architecture was subsequently modified by McDanel and Guha to handle 2C operands [97], using an adaptation of the Baugh-Wooley algorithm for 2C bit-parallel multiplication [98]. Although the quasi-serial multiplier exhibits limited performance in comparison to fully-pipelined serial-data multipliers, it offered a low-cost alternative to the ripple-carry add-shift implementations prevalent at that time.

3.7.2. Incremental multiplication

In 1979, a different class of pipelined serial-data multiplier, for full-precision unsigned operation, was proposed by Chen & Willoner [99]. This multiplier architecture operates in a markedly different manner to the Lyon & S/P architectures - the S/P flush multiplier is only slightly more closely related. Instead of dedicating one processor to the computation of each SPP, the Chen & Willoner (C&W) multiplier dedicates one processor (in this case a (5,3) counter) to the computation of each product bit. This gives the C&W multiplier the LSB-first 'on-line' property - at each iteration, one bit from both data and coefficient is input, and one product bit is output. However this facility carries considerable penalties in area and time - an $n \times m$ product calculation requires $n + m$ (5,3) counters and $n + m$ clock cycles for completion.

Chen and Willoner neglected many implementation details, and in many respects left great scope for improvements to their algorithm. Shortly afterwards, Sips identified one such improvement [100]. By introducing a shifting action on sum bits, the number of computational cells could be reduced from $n + m$ to n , increasing the usage-efficiency of the (5,3) adders. The resulting structure in many ways resembles a serialisation of the triangular Guild array multiplier [67]. Independently, Strader & Rhyne made the same discovery [101], naming the

improved structure the canonical bit-sequential (CBS) multiplier. The operation of this multiplier is covered in Chapter 6.

Gnanasekaran was the third author to point out this improvement, however he was the first to modify the structure for 2C operation, by including MSB-inversion logic [103]. A recent further modification to the CBS multiplier uses Booth's algorithm [75] to achieve 2C multiplication [102]. Buric & Mead published a full-precision incremental multiplier tailored as an inner-product step processor [104], which includes two (3,2) adders per stage instead of the single (5,3) adder of the CBS multiplier. The resulting two partial-product streams are combined into one in a final adder.

Apart from improved coefficient latency (and data latency in the case of the Lyon), these structures do not seem to offer any advantage over the Lyon and S/P multipliers, in which all stage adders achieve 100% hardware utilisation. Recently Scanlon & Fuchs proposed a modification to the CBS multiplier [77], which achieves full module utilisation by introducing bidirectional operation to the array, with attendant control overhead (see Chapter 6). They compare this architecture to a double-precision realisation of the Lyon multiplier, concluding that the area cost of the bidirectional CBS multiplier is approximately twice that of Lyon's. However another more interesting comparison might have been between the CBS multiplier and the S/P flush multiplier, whose latch count is considerably less than that of Lyon's.

The S/P flush multiplier, at considerably more hardware efficiency, also forms a $2n$ -bit product in $2n$ clock cycles, using n hardware stages. The one visible advantage of incremental multipliers over the flush multiplier is that they are 'on-line' with respect to both data and coefficient, whereas the flush multiplier is on-line only with respect to data. It is unusual (but not impossible) to find instances of coefficients in real-time computational networks resulting from immediately preceding computation (or external input), although this is often the case where data are concerned. Normally the coefficient is known *a priori* (even if only by n clock cycles - enough time to load the flush multiplier). This makes the advantage a tenuous one indeed.

One benefit of on-line arithmetic comes from the low latency, and consequently high overlapping potential, of chained computations on sequential networks of processors [10]. In this case, fractional operation (single- or multi-precision) is essential, as the word growth resulting from full-precision operations cannot be accommodated indefinitely. The FFT [27] is a good example of such a network - here additive and multiplicative operations are alternately chained. Bit-serial realisations may leave computational headroom for single-bit growth in adders (or round/truncate the results of these additions during the transform) but tolerate no word growth in multipliers [72]. However the on-line property and word-growth are inseparable in LSB-first schemes, where the 'early bits' output form the least-significant part of the product, which is discarded in fractional operation. Thus any latency advantages associated with LSB-first on-line hardware fade rapidly when confronted with networks of chained computational elements, such as the FFT.

Sips arrived at similar conclusions about LSB-first on-line architectures [105], and has more recently turned his attention to *MSB*-first on-line schemes. Incremental multiplication is operationally compatible with the advanced arithmetic operations possible in an *MSB*-first on-line computational environment [10]. Here fractional computation is performed on redundant-coded data to permit operations such as division and square-root extraction. The early extraction of fractional outputs is both feasible and desirable in this case, as the early bits are those of interest in fractional computation, i.e. the most-significant part of the product. Thus *MSB*-first on-line operations may be chained *without* the problem of word-growth. *MSB*-first on-line multiplication is performed in exactly analogous manner to the CBS algorithm (on redundant-coded digits).

According to Trivedi and Ercegovac, who proposed on-line architectures for multiplication and division in 1977 [10], incremental multiplication techniques were "well known" at that time, and were in use as far back as 1962 [106]. Chen and Willoner have not added to their original work - meanwhile Gnanasekaran has published a fast, modular serial/parallel multiplier design, using (3,2) counters [88]. From this comparison it is concluded that incremental multipliers find use only in the limited subset of applications which demand full-precision products to be output at minimal coefficient latency - other applications are better served by variations of Lyon or S/P multipliers [107].

A symmetry property of incremental multipliers may be exploited to achieve area-savings in the computation of squares and sums of squares [108]. The Lyon and S/P multipliers cannot put this property to use. The technique is described in Chapter 6.

3.7.3. Storage sharing

In DSP applications, few multiplication operations are performed in isolation. In many cases, identically formatted products are summed immediately after formation, or one datum is multiplied simultaneously by two coefficients (or vice-versa). Cheng modified JKM's second-order section to exploit this saving [78]. Murray & Denyer report a complex multiplier which shares data, coefficient and control among four double-precision Lyon multipliers [73], while CUSP, a radix-4 FFT processor, effects the same saving on four groups of four S/P flush multipliers [37].

Wawrzynek & Mead have implemented a bit-serial 'universal processing element' (UPE) for music synthesis, capable of interpolating multiplication calculations of the form $A + B \times M + D \times (1 - M)$ [109]. This consists of a modification to a 32-bit Lyon multiplier, and uses the approximation $\bar{M} = 1 - M$. As the bits of M and \bar{M} always differ, the pair of AND functions normally required to form two bit-products may be replaced by a single multiplexer, and the normal (3,2) adder of the Lyon multiplier may be employed for effective pairwise PP-summation. This is a good example of identifying redundant operations, and reducing hardware complexity by simplifying computational procedures. In Chapter 6, more such examples will be discussed.

Chapter 4

The Edinburgh tools

Novel contributions in the field of bit-serial systems architecture will be included in Chapter 5. Extensive use will be made of *FIRST*, a silicon compiler for bit-serial signal processors developed at the University of Edinburgh. Furthermore, Chapter 8 describes *SECOND*, a new 'full-span' structural silicon compiler, which uses the logic synthesiser *MOSYN* (also developed at the University of Edinburgh) to extend the range of *FIRST*. Through the exercise of these tools, the author has also made contributions to their development. As constant references are made to *FIRST* and *MOSYN* throughout the remainder of this thesis, they need to be introduced at this stage.

4.1. Background

Major advances in technology for the fabrication of VLSI integrated circuits have resulted in the potential to realise complex real-time computational algorithms which were previously of only theoretical interest [7]. With the integration of hundreds of thousands of devices on a single silicon surface arises the considerable likelihood of implementation errors - a 'complexity crisis' confronts VLSI designers [110]. To combat complexity, structured design styles and software tools [41] have emerged, culminating in the structural silicon compiler [111, 7, 112]. Structural silicon compilers guarantee working parts from high-level structural descriptions, through automatic assembly of known-good modules by known-good techniques. Low-level design errors are avoided by forcing the designer to follow this route.

Structural silicon compilers are not yet complete system synthesisers, as their power to implement function is limited by the flexibility of their underlying cell libraries. The inclusion of logic circuit synthesisers will allow the designer to carry his functional intent right down to the realisation in silicon, with no software-imposed architectural barriers. Accordingly, silicon compilers based on fixed cell libraries may be expected to give way to a new, more flexible generation of compilers, with the ability to translate the designer's intent efficiently at all levels of the

design process and the ability to incorporate new processes with minimal effort.

The University of Edinburgh has been active in the area of silicon compilation and design automation for some time. Two tools in particular are germane to this thesis: the *FIRST* silicon compiler [7] and *MOSYN* [113], a logic circuit synthesiser. A new tool, known as *SECOND* [114], is under development, drawing from the other two to realise a full-span structural silicon compiler. *SECOND* is introduced in Chapter 8, meanwhile this chapter gives a brief overview of *FIRST* and *MOSYN*.

4.2. *FIRST*

Building on the architectural methodology proposed by Lyon [33], and Buchanan's ideas on hardware description languages [115], *FIRST* [7, 116, 117] emerged as a joint project between the Departments of EE and CS at Edinburgh in 1982. Restricting its application to a specific class of problems (signal processing), and using bit-serial computation and communication, the designers of *FIRST* were able to come up with a very efficient layout strategy. Although potentially technology-independent, *FIRST* has been built around a 5- μ m nMOS cell library, and a two-phase non-overlapping clock scheme. Several functional primitives were also developed in 2.5- μ m double-metal CMOS [73].

The user is offered a single, high-level interface to *FIRST* in the form of a network description language. *FIRST* has (unseen by the user) a library of hand-designed 'leaf-cells', and software procedures for their assembly into parameterised 'primitives' (multiply, add etc.). Primitives form the lowest level of hierarchy available to the user, and may be used to construct flow-graphs of higher level objects (operators, chips, subsystems and finally systems) to solve specific DSP problems. An analogy to software programming languages would cast primitives in the role of 'instructions', while the other hierarchical levels correspond to user-defined 'subroutines'.

Chips are assembled according to a simple layout procedure, where two ranks of bit-serial processors communicate via a central channel. Pads are sited in an external rectangular 'ring', with space for optional linear feedback shift registers and control circuitry for purposes of self-test [5]. Serial communication allows most of the chip area to be devoted to computation, and routing of the central channel is a

relatively simple problem.

Figure 4.1 shows a bit-serial flowgraph, and Figure 4.2 shows the resulting *FIRST* chip, which performs the radix-2 FFT butterfly function [72]. The complex multiplication function is realised by 4 real multipliers and 2 real adders. A further 4 real adders complete the butterfly function.

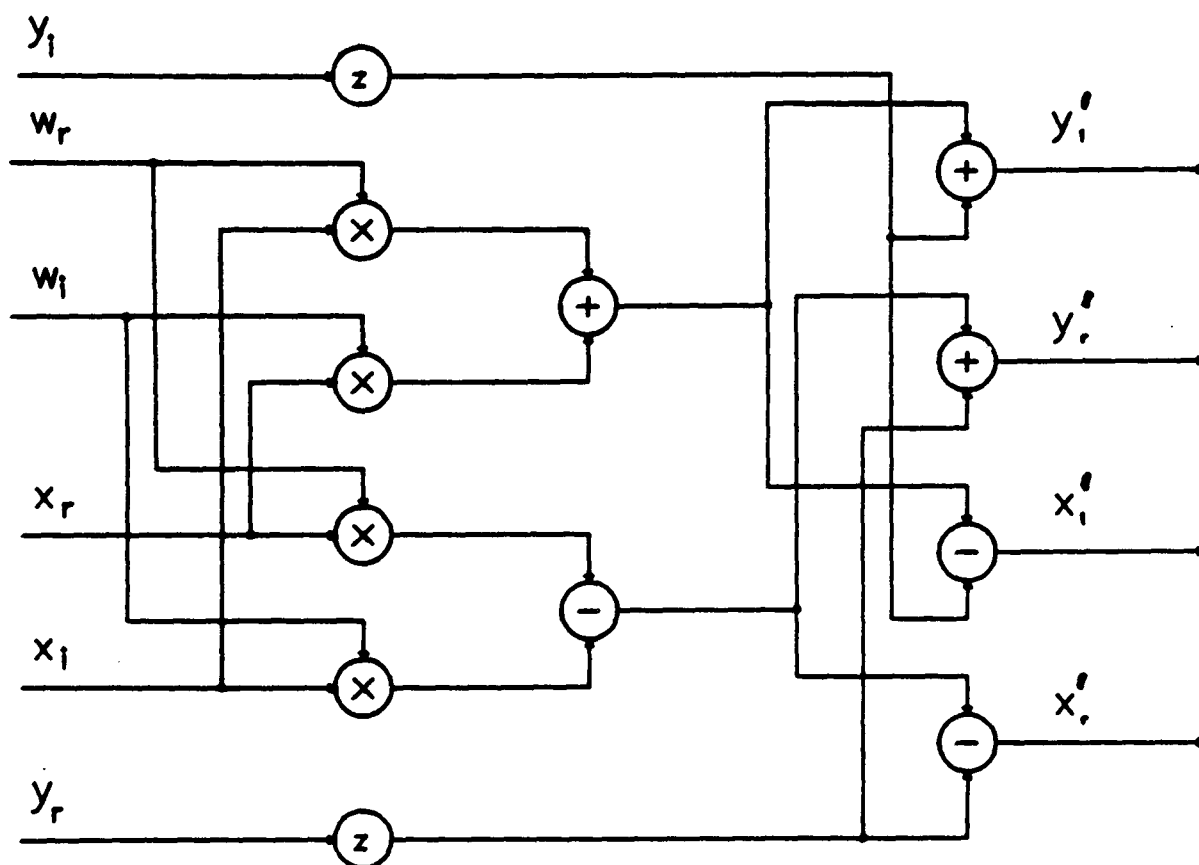


Figure 4.1: bit-serial flowgraph for FFT butterfly processor

No silicon compiler should allow the specification of low-level design errors (e.g. timing, race-conditions, design-rule violation, etc.). *FIRST* has the facility to warn the user when he strays from the path of design integrity in an obvious manner (e.g. nodes overdriven or left floating, word-synchronisation errors etc.). However high-level errors may creep in, which cannot be flagged by either language compiler or placement software (e.g. inputs to a subtracter reversed). Each primitive has a behavioural model which is exercised by a word-level, event-driven simulator for functional verification throughout the design process. It is crucial that behavioural duality is maintained between hardware and software models - this is ensured from the user's point of view by driving the simulator and layout software from the same source file.

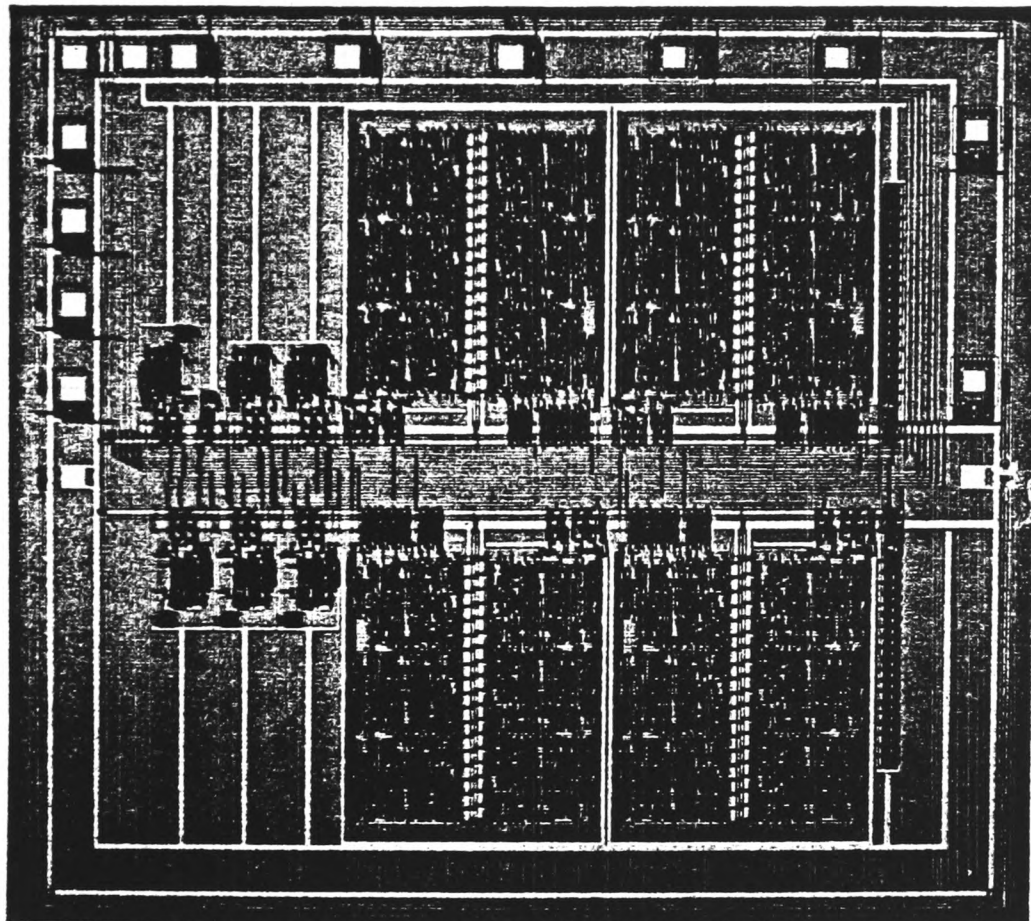


Figure 4.2: *FIRST-compiled FFT butterfly processor chip*

FIRST guarantees not only functional correctness but also performance, due to the low pin/buffer count benefits of serial communication. Each cell in the library is designed to work at a fixed, process-dependent clock-rate. The floorplan guarantees data transfers both between primitives and between chips at this rate. A recent comparison study [118] between *FIRST* and commodity 'micro-DSP' implementations of a large signal processor showed impressive savings in design time, board size, power consumption, chip and pin count.

4.2.1. The *FIRST* primitive set

FIRST offers the user around 20 functional elements, or 'primitives', with which to construct bit-serial signal processors. The flexibility of each primitive is greatly enhanced by parameterisation, where the name identifies a *family* of related functional elements rather than a single instance (each individual is identified by unique parameters). Primitives fall into 5 main categories - arithmetic, storage, control, format and pads [7].



Arithmetic: everyday arithmetic elements such as adders, subtracters and multipliers (single and double precision) are supplied, along with arithmetic shifters. Data-dependent operations such as absolute (modulus of a number) and order (sort two inputs by value) are supported, along with multiplex (1-from-2 data select) and a single-word ROM.

Storage: FIFO memories are provided for storage of both data and control signals. Simple shift registers serve for bit-level storage. Word-level memories may take advantage of the periodicity of control signals to save area, while data words may be stored in RAM structures with sign-repetitions removed.

Control: as well as the control memories, a source of control waveforms is provided.

Format: multi-precision representation of data is supported, and formatting blocks translate between representations. Numerical limiting is part of this process, and is also provided as a stand-alone operation.

Pads: clocked input and output pads are provided.

4.2.2. Case studies

Chapter 5 describes a detailed case study of *FIRST*. Appendices A and B contain the *FIRST* language codings related to this study, and Appendix E reproduces some earlier, more didactic case studies.

4.3. *MOSYN*

A logic circuit synthesis package known as *MOSYN* [113] has been developed at Edinburgh by a visiting professor from Tokyo. *MOSYN* produces a set of MOS circuit topologies (and terminal orderings) for realisation of input specifications, using a 3-way decomposition and reduction procedure recursively. Logic functions are manipulated using a 7-valued logic system. Target technology is static or dynamic CMOS or nMOS, as well as cascode voltage switch logic [119].

MOSYN allows the user to describe his desired function in 3 different ways: in truth table notation, in cube notation (both allowing specification of 'don't cares')

and in a concise logic description language. *MOSYN* produces in response a set of complex gates (e.g. nMOS pull-down trees) with comprehensive statistics, from which the designer may select the best candidate. *MOSYN-2* has the added capability to produce gate-matrix layout [120,42] of the selected tree, thus freeing the designer from the layout task *at the logic level*. Multiple-output functions may be generated, by synthesising individual circuits then merging and eliminating redundant portions.

MOSYN employs two distinct types of logic function to synthesise MOS transistor networks. The 'request' function specifies what is to be synthesised, while the 'realised' function represents one of the possible realisations. Both functions are 7-valued.

An algebra is included for the manipulation of these logic values. The operations of addition, subtraction and multiplication are supported. Addition has a direct physical representation (wired-or), while subtraction is the conceptual inverse of this. Multiplication operators are required to model pMOS and nMOS FETs, and different operators manipulate request and realised functions. Multiplication operators have a 'gate function' as first operand (*MOSYN* uses only 'hard' binary gate functions valued $\{0,1\}$). Thus the filtering action of a FET on a request/realised function is modelled by the multiplication operator. The request function is initially decomposed into three functions: a gate function, a terminal function and a wired function. By hierarchical repetition of this process, a tree-like circuit is finally obtained. Twin stack structures are used, one for processed FETs and one for pending FETs (a FET being represented by a data structure containing information such as FET type, request and realised functions, status and linkage information etc.). Decomposition is terminated when the function is empty, or realised by either a constant (0 or 1), an input variable or an already synthesised node.

Unlike other logic synthesisers, which use heuristics to arrive at near-optimal solutions [121, 122], *MOSYN* employs a certain amount of brute force in permuting input variables, although equivalences are detected before this process. The brute force approach is particularly appropriate in a bit-serial environment, where logical fan-in is low, limiting the dimension of search spaces.

4.4. The promise of full-span structural silicon compilation

The pairing of a 'half-span' silicon compiler such as *FIRST*, and a logic circuit synthesiser such as *MOSYN* promises powerful advantages to systems designers. Although *FIRST* supports an expandable cell-library, the task of designing and including new 'primitives' requires circuit design skills. Currently the system designer who lacks access to these skills must realise functional intent with the available primitive set.

With the availability of *MOSYN*, the systems designer may commission the design of his own *FIRST* primitives (on the understanding that they obey *FIRST*'s signalling and interfacing conventions). Not only may the existing library be extended - entire new libraries may be specified, using a different set of interfacing conventions if desired.

The primitive design task is reduced to that of specifying the primitive in terms of logic and latches, running the logic through *MOSYN*, and writing simple physical assembly and behavioural procedures for *FIRST* to call. These procedures are unified and automated in *SECOND*, using the user's 'typed' physical hierarchy to direct automatic assembly.

Although the gate-matrix layout produced by *MOSYN* may not be as dense as the hand-designed layout of the existing cell library, the functional power provided by tailored primitives more than compensates for this. Moreover, a cell-library may be described in *MOSYN* input specification form, greatly increasing its portability between processes and even technologies. Primitive assembly procedures may easily be specified in technology-free manner, and formalised for the general case.

The automation of primitive generation from user-specifications is the last major barrier to full-span structural silicon compilation. To this end, the *SECOND* project is described in Chapter 8. By Synthesis of Elementary Circuits ON Demand [114], the three forms of flexibility outlined in Chapter 1 should be realised.

The next chapter presents new material in serial-data systems design, exercising *FIRST* in the systems arena. The experiences gained suggest new architectural techniques, and also serve as a precursor to next-generation tool development.

Chapter 5

Bit-serial systems design: methodology and case study

This chapter describes a methodical approach to bit-serial systems design, which is illustrated through implementation in *FIRST* of a computationally-intensive digital signal processor. Potential improvements to *FIRST* are identified in the course of this study. The system to be designed is a polyphase-network filterbank for use as a satellite communications transmultiplexer, as specified by the European Space Agency (ESA) for evaluation of *FIRST*. Use is made of experiences from previous investigations in the architectural areas of Fourier transform machines [72][†] and transversal filters [123][†] - the former study is reproduced in Appendix E. A target architecture is identified, and the system is initially described in a functional manner, then verified via high-level simulation. Partitioning issues are then addressed to produce an optimal chip set. Testing strategies are outlined, and some weaknesses of *FIRST* exposed by this study discussed.

We begin by briefly outlining the author's personal approach to bit-serial systems design, which was developed during previous case studies including those of Appendix E.

5.1. Bit-serial systems design

The favoured approach to bit-serial systems design is the popular method of structured design, using top-down specification and bottom-up implementation [41, 7]. It is important that tools used for specification support hierarchical descriptions - *FIRST* adequately satisfies this requirement, supporting physical as well as functional hierarchy through 'typing' of hierarchical levels. Figure 5.1 illustrates *FIRST*'s typed hierarchy - arrows indicate the legal call structure.

Systems implementation proceeds in two main stages - description and verification of a *functional* model of the system, then translation of this functional model into a sensibly-partitioned *physical* model. We refer to these two models as the *soft*

[†] published work by the author.

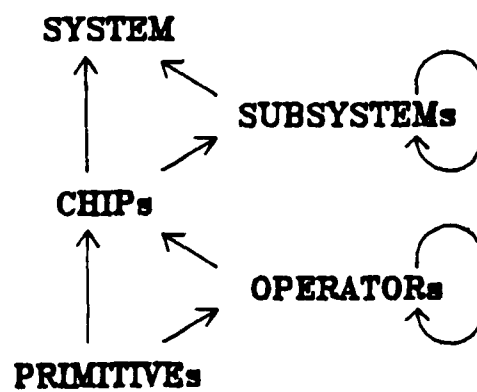


Figure 5.1: *FIRST's typed hierarchy*

model and the *hard* model respectively.

5.1.1. Functional design - the soft model

The soft model makes full use of the facility of primitive parameterisation through arithmetic expressions supported by *FIRST*, allowing key systems parameters (such as system wordlength, multiplier resolution etc.) to be varied at will. The *FIRST* behavioural simulator is used to decide on optimal values of these parameters. No attention is paid to physical issues at this stage, leading to such physical absurdities as unpartitioned BITDELAY primitives of many hundred stages in length. However the length parameter may be a complicated function of many other parameters, which change frequently as a soft design evolves. Arithmetic expressions for parameters, once fixed, allow the designer to make such changes without concern for their effect on unrelated areas of the design.

Multiplexing

It is seldom the case that computational bandwidth (a function of the technology) and signal bandwidth (a function of the application) are the same. A set of techniques has been developed whereby a single physical processor services the role of n virtual processors (where $n \leq$ the ratio of computational to signal bandwidth [7]) - this is known as (time-division) multiplexing. By multiplexing, the two bandwidth quantities identified above may be matched.

The choice of multiplexing scheme must be decided at an early stage in design, however its implementation may be postponed until the design of the hard

model. The size of multiplexing state memories depends not only on system parameters such as system wordlength and multiplexing level, but also on the task of the memory (whether samples recirculate indefinitely or not). Data and coefficient state memories in transversal filter design provide an example of this [123] - while coefficients remain static in relation to their associated virtual processor, data samples step through the virtual processors, one per sample period. The state memory requirements differ in each case. The design of these memories is complicated by the fact that loops may have multiple exits, forcing particular partitions even in the soft model.

Processor design

Another early task is to identify the processing elements required by the application. *FIRST* provides single-operation primitives (multipliers, adders etc.) - it is the task of the designer to bundle these primitives into custom arithmetic engines (e.g. inner-product step processors, FFT butterflies etc.) which may tackle identifiable sub-tasks of the application.

Time-aligning

FIRST primitives are supplied with 'time-aligned' inputs and outputs where possible - this encourages their use as functional 'black boxes' by the designer. He only needs to remember what a primitive does (its function), and how long it takes to do it (its latency). It is useful to extend this concept to the design of custom arithmetic operators, arranging for all operator inputs (and indeed outputs) to be mutually synchronous. Once again, the designer may treat operators as 'black boxes' with function and a single latency parameter.

Time-tagging

The practice of time-tagging (latency accounting) reduces the possibility of synchronisation errors as network branches are combined or loops closed. This aids the management of data-distribution, both globally (at system level) and locally (within processors or subsystems). Compensating delays may be specified in terms of differential time-tags. Time-tagging is useful in both soft and hard models - tags

may be arithmetic expressions or integers. Figure 5.2(a) shows a soft, open critical path (matched filter example), and Figure 5.2(b) shows a hard, closed critical path (adaptive filter example).

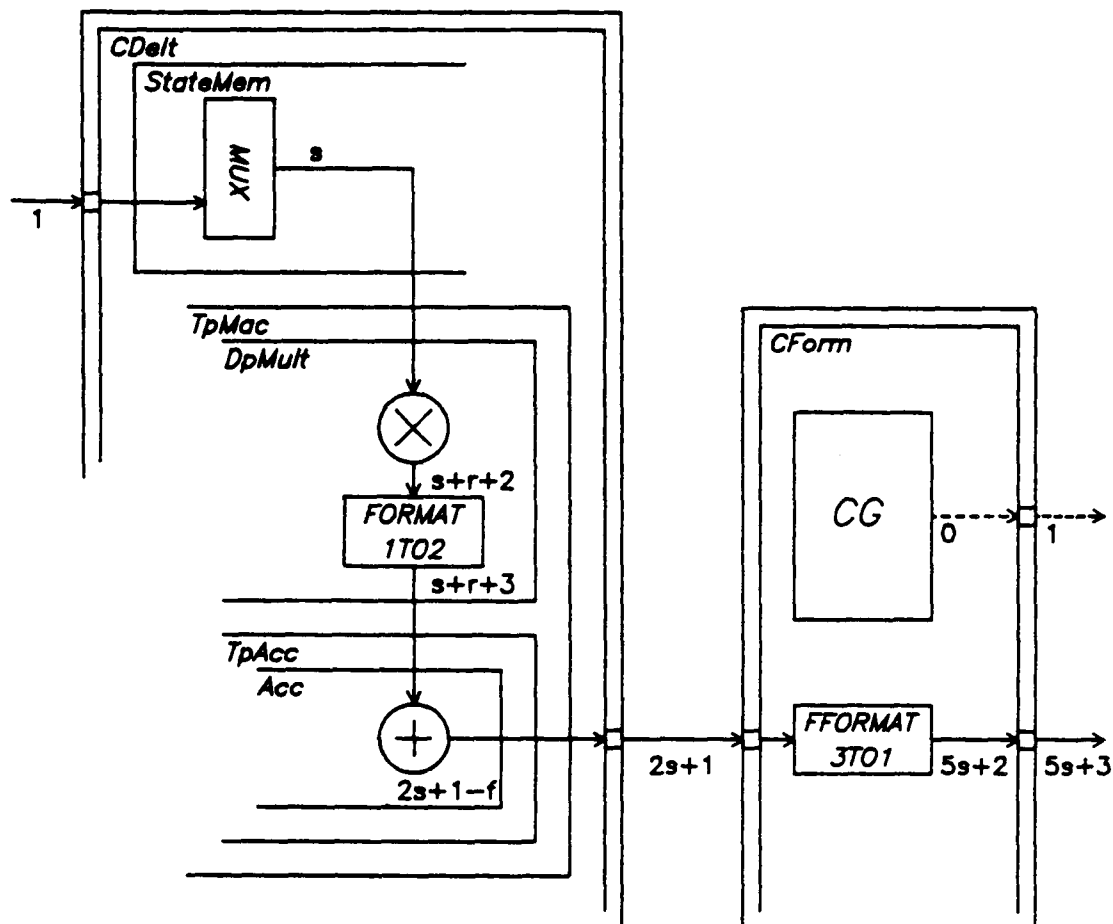


Figure 5.2(a): 'soft' critical path

5.1.2. Physical design - the hard model

When the optimal values of all parameters have been fixed and the behavioural model of the system meets specifications, the designer may address physical partitioning issues.

Physical partitioning

Partitioning involves dividing up the system and inserting chip boundaries to produce the final chip set. Intelligent use of parameters in the soft model can greatly simplify this task. Arithmetic expressions are replaced where possible with integer values. Time-aligned operators may have this restriction (carefully) relaxed thereby eliminating unnecessary delays. Large memories may be partitioned to

5.1.3. Postscript

The above discussion may give the impression that systems design is a sequential process, which little or no interaction between stages. This is not really the case, as mixtures of top-down and bottom-up techniques are usually employed, and soft models are seldom created with no regard to physical considerations. The designer may estimate partitioning requirements from the outset - the 'hardening' process is a gradual one.

The following case study illustrates the use of the soft model for functional verification, and the subsequent process of hardening.

5.2. Case study system description

The polyphase network (PPN) filterbank has applications as a transmultiplexer for satellite communications. The PPN [124, 125] performs efficient sample-rate reduction on high-frequency signals. It consists mainly of a set of small filters into which the input data stream is 'corner-turned' in blocks. When a PPN is used in conjunction with a DFT back-end (Figure 5.1), short-term spectral analysis is possible at reduced computational bandwidth. The Fast Fourier Transform (FFT) is commonly used to realise the DFT in practice. The PPN design stems from decimation of a prototype lowpass filter response which, in combination with the rotations inherent to the DFT, results in a set of frequency-shifted basic filter equivalents spanning the spectrum, i.e. a uniform filter bank [126].

Bit-serial techniques have been used in transmultiplexer design before now. Freeny et al. implemented a transmultiplexer in 1971 [127] built from bit-serial modules in discrete ECL, very much in the spirit of JKM's approach [26]. More recently, Jain et al. [128] chose a similar transmultiplexer to demonstrate the use of a silicon compiler in systems design. Neither uses PPN techniques. The ESA specification considerably exceeds the computational demands of these systems, mostly due to its greater operational signal bandwidth and channel count.

5.3. FFT subsystem design

FFT architectures are dominated by multipliers, adders, switches and delays, and fixed-point realisations contain no data-dependent operations. They are thus ideal candidates for bit-serial implementation. Some experience of FFT systems was gained in early case studies of *FIRST* [72] - see Appendix E. A full array architecture was studied, then row and column multiplexing schemes were introduced.

5.3.1. FFT overview

The Discrete Fourier Transform (DFT) plays a significant role in the field of spectral analysis, and is a common tool for mapping between time and frequency domains in digital signal processing [27, 129]. The DFT transforms an N -point sampled time series into an equivalent N -point frequency series. In equation form, the value of the transform X at the k^{th} frequency point, or 'bin', is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2\pi/N)nk} \quad 5.1$$

where both $X(k)$ and $x(n)$ are assumed periodic with period N , and k ranges from 0 to $N-1$. The factor $2\pi/N$ normalises the argument of the complex exponential, allowing k to span (at discrete intervals) the frequency range from zero up to, but not including, the sampling frequency. The variable n represents time, and so the quantity nk represents normalised angular displacement. As angles can be reduced modulo 2π , so can normalised angles be reduced modulo N . The ability to reduce angles modulo some sequence length is a cornerstone of the Fast Fourier Transform (FFT) algorithm [130, 131]. The FFT takes advantage of the properties of symmetry and periodicity of the DFT weights to reduce the computational complexity of the DFT to order($N \log N$). Eqn. 5.1 implies that the FFT can be computed using only the plane-vector operations of addition (i.e. real addition of vector components) and rotation.

5.3.2. Vector rotation

Fourier Transform machines require the operation of vector rotation, often (but not always) performed by complex multiplication. Dedicated complex multipliers are relatively rare, and usually evaluate the complex product using four real multipliers, minimising communication and sharing storage of operands in parallel [132, 133] or serial [134, 37, 73] 2C architectures. Just as the FFT uses the commonality of coefficients to make computational savings over the DFT by combining before rotating, so it is possible to perform addition before multiplication to reduce the number of real multiplies in the complex multiplier. The 3-multiplier solutions of Golub and Buneman [135, 136] reduce computation - however these approaches increase storage costs and adversely affect dynamic range in bit-serial realisations [72].

The four carry-propagate adders required by the conventional bit-parallel approach may be reduced to two using 'merged' arithmetic [137]. Alternatively, a reduced form of binomial expansion may approximate the trigonometric functions [138], reducing the number of required shifts and adds to effect the transformation. However none of these approaches make full use of the cross-symmetry of operands to simplify calculation of the complex product.

CORDIC processors [139, 135] have been suggested as an alternative to complex multipliers as a vector rotation medium - however (like 2's complement dividers) they contain 'conditional' operations which hamper performance. Limited success has been achieved in pipelining CORDIC processors [140] - nevertheless CORDIC offers the flexibility to tackle computational areas such as advanced function generation and transformation [141, 142].

In the more down-to-earth problem of vector plane rotation, the most common solution is the complex multiplier with unity-modulus coefficient. White [143] suggested an area-efficient symmetric-coded distributed arithmetic solution to the complex multiplication problem - this has since reappeared in bit-parallel [144] and serial-pipeline [48][†] form. In Chapter 6 White's model will be encompassed with a theoretical framework for synthesis of small matrix-vector computers.

[†] published work by the author.

5.3.3. The pipeline FFT

The FFT can be implemented in a fully word-parallel manner, using a large number - $N/R \log_R N$ - of butterfly processors, where N is the transform length and R is the radix. It follows that the hardware cost is of order $N/R \log_R N$ (there are $\log_R N$ columns, each containing N/R processors). The full array structure is capable of block transforms in unit word time. The area-time product is therefore also of order $N/R \log_R N$.

Area may be traded against time by a factor of $\log_R N$ to realise the column machine, which features N/R processors [72]. When N is large, this scheme still leads to fast but hardware-intensive systems. A more common area-time tradeoff is to divide area, and multiply time, by N/R , i.e. to employ a multiplexing level of N/R . The same transform may then be performed in a word time of N/R , using only $\log_R N$ processors. This is in effect implementing just one row, and multiplexing down the columns. The resulting row machine is a pipeline FFT [145], and the most advanced FFT machine known to the author employs this architecture (with radix $R = 4$) to realise 4096-point transforms on 40MHz data [146].

The area-time tradeoff mentioned above is a little more complicated in reality. The variable network topology required to 'perfect shuffle' data blocks in the FFT [147, 72] must be implemented through an arrangement of memories and switching elements [145, 72, 146]. The area cost of these elements must be taken into account. Appendix E contains details of a pipeline FFT realisation in *FIRST*.

5.4. Filterbank subsystem design

A filterbank is usually composed of several identical transversal filter components. Row and column multiplexing schemes may also be employed in filterbank design. Array topology is more regular than in the FFT, hence there is more scope for efficient multiplexing schemes. As the FFT is the more difficult to design, the multiplexing scheme appropriate to the FFT may also be used in the filterbank.

5.5. System specifications

The specifications of the system were as shown in Table 5.1.

Table 5.1: Initial specifications	
sampling frequency	= 1.28 MHz
input signal resolution	= 8 bits (real)
number of channels	= 32
channel bandwidth	= 10
transition bandwidth	= 5 kHz
passband ripple	= 0.5 dB
stopband attenuation	= 30 dB
channel separation	= 20 kHz
prototype lowpass filter length	= 64 x 6 = 384 points
subsampling ratio	= 32

It was agreed with ESA that the process bandwidth restriction of the available 5 μ m nMOS function library (maximum 8 MHz clock rate) could be relaxed within reason for the purposes of this study. The overall latency of the implementation is not of concern, which allows some extra design freedom. System outputs are in rectangular (real/imaginary) form.

5.6. Initial design issues and decisions

The PPN filterbank consists mainly of a bank of short transversal filters whose outputs form the inputs to an FFT subsystem. In this case there are 64 finite impulse-response (FIR) sections of length 6 points. This immediately suggests a distinct functional partitioning of the PPN filter into an FIR section and an FFT section.

One *FIRST* system or two

The input signal is in 8-bit form, and this might be expected to grow to 10 or 11 bits in the FIR filters (a 6-point filter with full-scale coefficients can cause growth of $\log_2 6$ bits - in practice this would be less). The 64-point FFT section would take this extended wordlength as input, producing a further maximum 6 bits ($\log_2 64$) of growth. Thus it is apparent that the necessary serial wordlength for the FFT section is larger than that for the FIR, and the first design issue arises, namely:

- 1 To design two separate *FIRST* systems, of differing wordlengths, with some buffer interface between them, or (more simply) implement as one unified system, with some wasted wordlength in the FIR section? The tradeoff here is extra *FIRST* FIR chip count against the cost in design time and board space of the interface and additional control requirement.

Efficiency and modularity of FFT computation

As the FFT input is real, a full complex implementation results in unnecessary computation [131]. Another design option is to use a full complex FFT processor to perform the FFTs of 2 real sequences concurrently [129], with a final bank of adders and subtractors to unravel the 2 output sequences. The next design issue arises, namely:

- 2 To employ a modular FFT to transform single real sequences, accepting a 50% inefficiency in hardware use, or to transform double sequences? The tradeoff here is the improvement from 50% to 100% utilisation of modular hardware, against the extra cost in design time and board space of the buffers which interleave FFT input sequences and de-interleave FFT output sequences, plus the extra computation for output recovery.

Multiplexing scheme

The word-rate supported by the *FIRST* bit-serial system can be found by dividing the process clocking rate by the system wordlength. This figure can be expected to lie between 200 kHz and 1 MHz. As the word-rate at the inputs to the FIR section is only 40kHz, it is apparent that a degree of hardware time-sharing

(multiplexing) may be employed, at a level M of between 5 and 25 from the above figures.

The specification calls for 64 6-point filters, and a 64-point FFT. Computation in the FIR section may be multiplexed 'column-wise' (each FIR filter using one physical filter point), or 'row-wise' (realising a virtual filter-bank by multiplexing one physical filter). Similarly computation in the FFT section may be multiplexed column-wise (implementing one column of a constant-geometry FFT processor [27]) or row-wise (implementing one row of a standard FFT and effecting network topology by storage and switching - a pipeline FFT [145]).

It is interesting to note that a 64-point radix-2 FFT has $\log_2 64 = 6$ stages - the same as the number of FIR filter points. Both FFT implementations require complicated switching and memory wedges - external to the column machine and internal to the pipeline [7]. However the column machine requires considerably more storage than the pipeline machine. A third issue has now been identified:

- 3 To use a column multiplexing scheme, with $M = 6$ or 12, or a row scheme, with $M = 8$ or 16? One goal here is to find a value of M which allows circuit clocking at optimal frequency. Also the ease of communication between FIR and FFT sections must be considered.

5.6.1. Resolving the issues

On the basis that interfacing memory between FIR and FFT is expensive, we start by admitting some bias towards a solution which avoids the use of such memory. In the case of issue 1, we choose the unified system, whose wordlength was 16 bits, and for issue 2, we choose the 50% efficient FFT.

If we choose $M = 12$ (the column scheme) we must use a serial clock at $40 * 16 * 12 = 7.68$ MHz. The row scheme with $M = 16$ gives a clocking rate of 10.24 MHz, which is better, and still within specification. If $M = 16$, the FFT machine radix is $64/16 = 4$, which is said to be the optimal radix for a pipeline machine [27, 145].

The row scheme is thus initially more attractive, but when we consider the

communication between FIR and FFT sections with $M = 16$, it appears more attractive still. If the system input interface is arranged to translate a single 1280 kHz wordstream into two 640 kHz wordstreams as depicted in Figure 5.3, the FIR section may be realised as two physical 6-point filter pairs (each pair sharing storage), the FIR signal storage may be completely merged with its multiplexing state memory, and no interfacing memory is required between FIR and FFT sections.

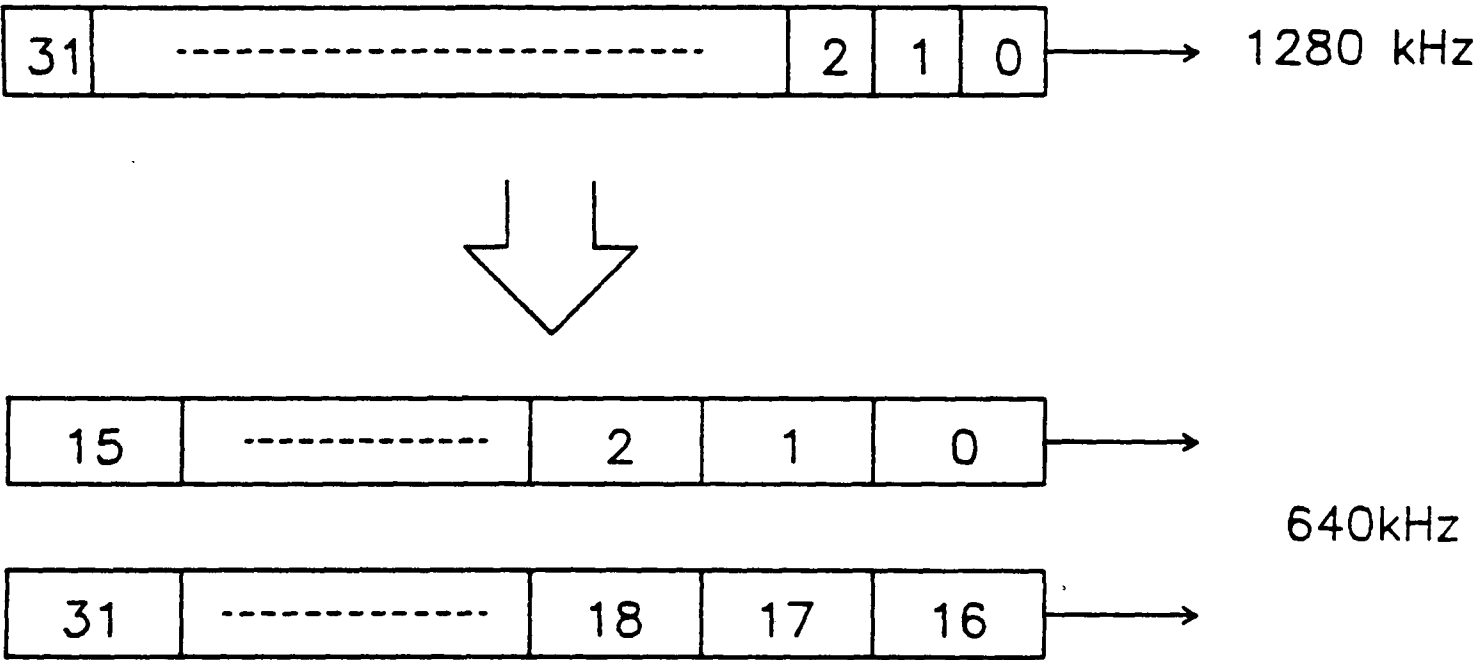


Figure 5.3: input interface memory

5.6.2. Additional specifications

We may now add to the system specifications (Table 5.2).

Table 5.2: Additional specifications	
system wordlength	= 16
multiplexing level	= 16

These quantities may remain as ‘soft’ *FIRST* parameters in subsequent descriptions.

Input interface

The input interface mentioned above should be built from standard parts. Its design therefore falls outside the scope of this study, although we note that it might consist of a 32 x 8-bit RAM, with a linear address counter for writing data, and a 'perfect shuffle' counter for reading data. The read pairs should be aligned, and sign extensions packed on before being presented in an LSB-first serial manner to the *FIRST* system.

The RAM must undergo one read and one write at a rate of 1.28 MHz, pointing to a cycle time of around 390 ns, well within the limits of current static RAM technology.

Coefficients

Coefficients are required in both FIR and FFT sections, although their use is somewhat different. In the former case, we may wish the prototype impulse response to be programmable (potentially at run-time), and accordingly provide the facility to read this impulse response externally on power-up or reset, thereafter storing the coefficients in a loop of state memory local to each multiplier. The coefficient vector should undergo the same block transformation as the signal does in the input interface, namely each 32-word block is transformed into 2 concurrent 16-word blocks.

FFT coefficients (twiddle factors) on the other hand are not usually 'variables' - we would supply them in the form of an on-board ROM which continually outputs the current coefficient bit-pattern, obviating any *FIRST* storage of twiddle factors. For purposes of simulation however, we create a notional *FIRST* chip which generates twiddle factors.

A similar strategy could of course be used in the FIR section, using RAM instead of ROM. The decision whether to use *FIRST* memory elements or commercial RAMs is a difficult one, and cannot be properly resolved until modularity and partitioning issues are addressed later in the design process.

5.7. Functional design - the soft model

We may now commence the functional design, starting with a top-down specification. As the two sections so far identified are fairly complex in themselves, we choose to design and simulate each one separately, confident in the simple technique outlined earlier for their ultimate interconnection. We have the opportunity to carry out some design exploration at this stage, and so we leave many system parameters (e.g. FIR coefficient resolution) 'soft' for the time being. We pay little attention to physical issues at this stage.

The system is partitioned into the FIR section and the FFT section (Figure 5.4). The FIR section forms one notional 'CHIP', whilst the FFT section is in the form of a 2-'CHIP' SUBSYSTEM: one CHIP generates and distributes twiddle factors and the other performs the FFT computation. The *FIRST* description of the soft model is listed in Appendix A, and should be referred to as an aid to understanding the following sections.

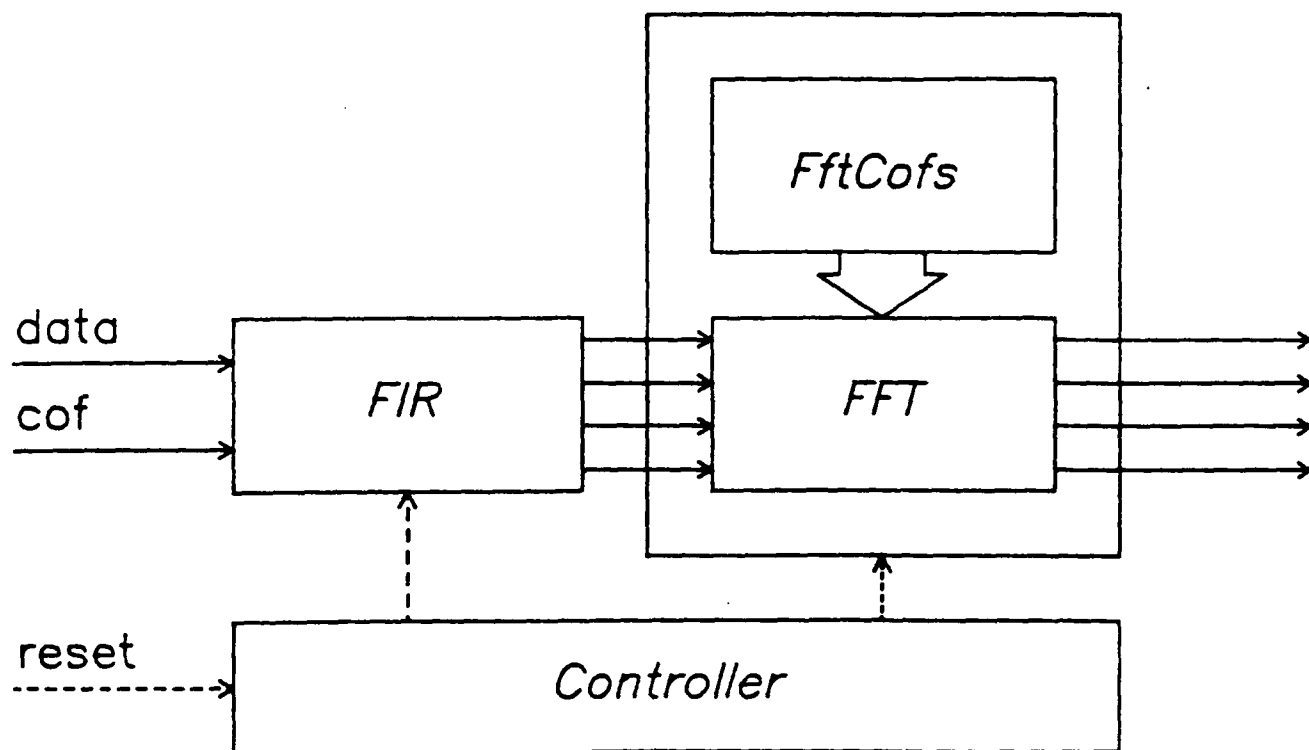


Figure 5.4: high level system plan

5.7.1. The FIR section

As stated above, the FIR section consists of 2 pairs of 6-point filters, each filter point having the facility to capture M coefficients, and having local data storage for M signal words. We say 2 pairs (rather than 4) because we may take advantage of the fact that samples are 'bounced' between filter pairs (the subsampling ratio is 32 and there are 64 filters), and storage may be shared. Figure 5.5 illustrates this.

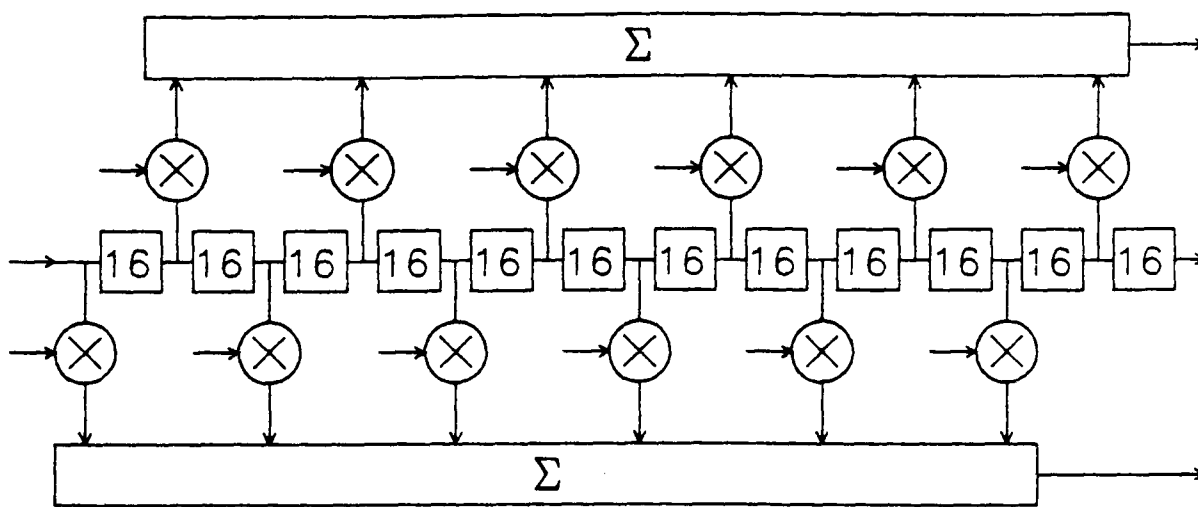


Figure 5.5: sample-bouncing filter pair

The signal memory is simple first-in-first-out (FIFO), but the coefficient memory must include the ability to recirculate M samples once they have been read. It therefore consists of a memory loop containing a multiplexer under control of an event pulse of length M words.

The basic component of the FIR stage - a filter-multiplier with its associated memory, may now be described as an OPERATOR in the *FIRST* language. The OPERATOR *FilterSection* takes 4 parameters: these are system wordlength *swl*, coefficient resolution *cofres*, input signal significance *signif* and multiplexing level *muxlevel*. We use the 1-bit predelay option on the signal input to the multiplier, to compensate for the 1 bit delay incurred on passing through the coefficient

multiplexer. We must delay control input to the multiplier accordingly. The OPERATOR is 'time-aligned', i.e. all its inputs are synchronous [7].

The next task is to construct a pair of physical 6-point filters, with signal and event delays cascaded through. To do this, we need 12 filter-multipliers, and two 6-input adder trees. We design the adder tree *Adder* using 5 adders with 'fanned-in' sums. The parameter *del* will be used to balance latencies later in the design process. Again we have used the predelay option for latency compensation.

We may next describe the filter pair *FilterCascade*, which consists of a regular, cascaded connection of OPERATORS. *FIRST* has a useful shorthand syntax for describing such repetitions. We finally produce a notional CHIP *FilterOut* containing the entire FIR section. At this stage we assign values to parameters via the CONSTANT statement.

Simulation

We may check the functionality of the FIR section by applying a 16-word block of dc to the input of each filter-pair. This is equivalent to applying a 32-word dc block (effectively an impulse at the sub-sampling frequency) to the input interface memory.

The response to this should show contiguous 32-word segments of the prototype lowpass filter response in the form of parallel pairs of 16-word blocks. Figure 5.6 illustrates the *FIRST* simulator output.

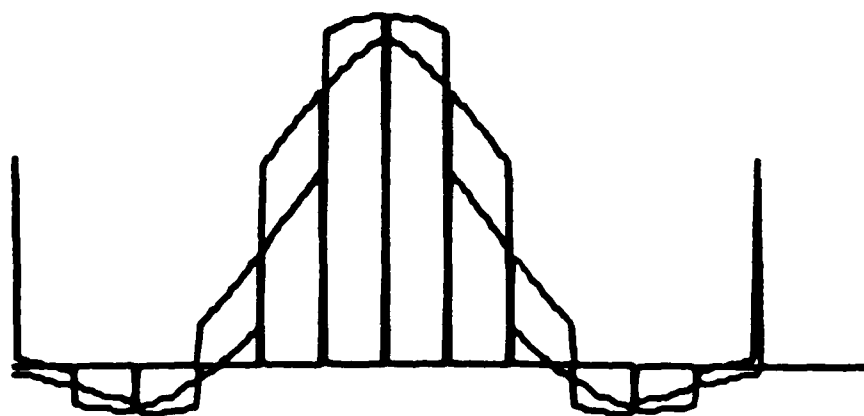


Figure 5.6: segmented block impulse response

5.7.2. The FFT section

The FFT computation is performed on a radix-4 64-point DIT pipeline machine. This has $\log_4 64 = 3$ arithmetic stages, with perfect-shuffle networks between stages. The shuffles are performed by a wedge-commutator-wedge arrangement (a commutator being a complex 4-pole switch).

The arithmetic at each stage is in the form of a 4-point FFT, which can be performed by adders and subtracters. All but the first stage are preceded by a twiddling (vector rotation) block. Figure 5.7 shows the FFT structure.

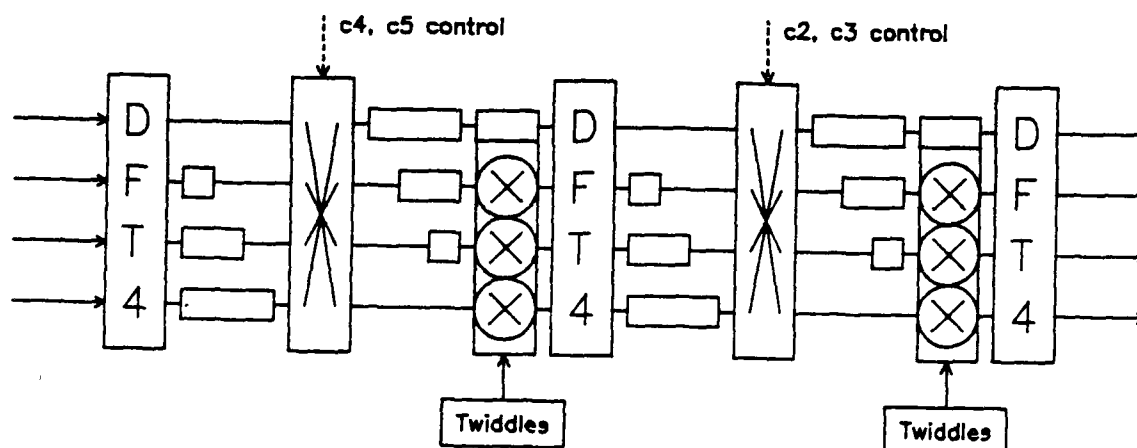


Figure 5.7: 64-point radix-4 DIT pipeline FFT

In order to facilitate the merging of FIR and FFT sections, a dummy chip containing 2 words of delay is inserted at the front of the FFT pipeline. This allows timing strategies to be developed for the FFT pipeline which do not require subsequent alteration on implementation of the full PPN filter bank. We may now proceed with a bottom up implementation, starting with the vector rotation.

Vector rotation

Vector rotation may be carried out using a complex multiplier. However *FIRST* does not currently provide a complex multiplier primitive. We must therefore construct an OPERATOR *CmplxMul* using real arithmetic elements - 4

multipliers, an adder and a subtracter. We use the latency parameters in the adder and subtracter to set the complex multiplier latency to two words, whilst controlling multiplier resolution via the parameter *co*.

The twiddling block in a radix- R FFT contains $R-1$ complex multipliers [27]. Here $R = 4$, so we must provide 3 complex multipliers and a complex unit word compensating delay for the twiddle free leg of OPERATOR Twiddle.

Radix-4 FFT

A radix-4 FFT may be constructed from 4 radix-2 DFTs. Although it requires an internal multiplication by $-j$, this can be accomplished by modifying the routing around one of the radix-2 DFTs. OPERATOR *Dft4* contains 4 instances of OPERATOR *Dft2*, which consists of a complex adder and subtracter. Adder latency parameters are used to set the OPERATOR latency to one word.

Perfect shuffler

The pipeline machine effects topological changes dynamically, using an arrangement of memory wedges separated by a commutator switch. The size of the wedges is proportional to the period of the commutator switching pattern, and we use the parameter *l* to vary this at different stages in the machine. OPERATOR *LineDel* is a basic parameterised complex wedge.

OPERATOR *Commutator* consists of 2 instances of OPERATOR *ComReal*, the 4-pole switch. This is in turn constructed from 8 MULTIPLEX primitives, under control of a 2-wire code. In a manner similar to *Dft4*, latency parameters are used to set the OPERATOR latency to one word.

Pipeline FFT

The pipeline FFT is encapsulated in CHIP *Pipe*, and consists of a cascade of previously declared word-synchronous OPERATORs, following Figure 5.7. The remaining tasks are to provide a correct control network, and ensure that the twiddle factors are distributed correctly in both time and space.

Simulation

We may save later design effort by arranging the FFT timing to be appropriate for the entire PPN system, and to this end we insert a dummy CHIP with a latency of 2 words to mimic the FIR section. This chip can subsequently be replaced by the FIR section to implement the entire PPN system without disturbing the FFT timing.

We split the FFT verification into 2 parts: first of all the timing of the commutator switching, then the correctness of the vector rotation. We may effect the former by applying dc at the 4 real inputs of the FFT. We should observe accumulation of energy at the initial (dc) output of each radix-4 DFT in the pipeline. The height of each block should increase, and its duration decrease (by a factor of 4) as we move through the transform, until all energy has been gathered in the dc output bin.

Having verified topology, we may now verify the computational scheme by applying a pure tone at the fundamental frequency (found by dividing sampling frequency by transform length), and observing energy accumulation in bins 1 and 63 (63 being the alias of 1).

5.7.3. The polyphase-network filterbank

With two working subsections, we are now in a position to implement the entire *FIRST* system. We simulate the response to a chirp signal which spans the frequency range from dc to nyquist. As the *FIRST* output comes 4 real/imaginary pairs at a time, in blocks of 16 and with digit-reversed frequency index, a software package was written to display magnitude, log-magnitude and phase responses as a function of time and frequency. Figure 5.8 shows the magnitude response of the PPN system to the chirp input.

5.8. Physical design - the hard model

Having produced a functionally correct description of the PPN filter, we now turn our attention to partitioning issues. Appendix B is the *FIRST* description of the hard model, and should be referred to throughout this section.

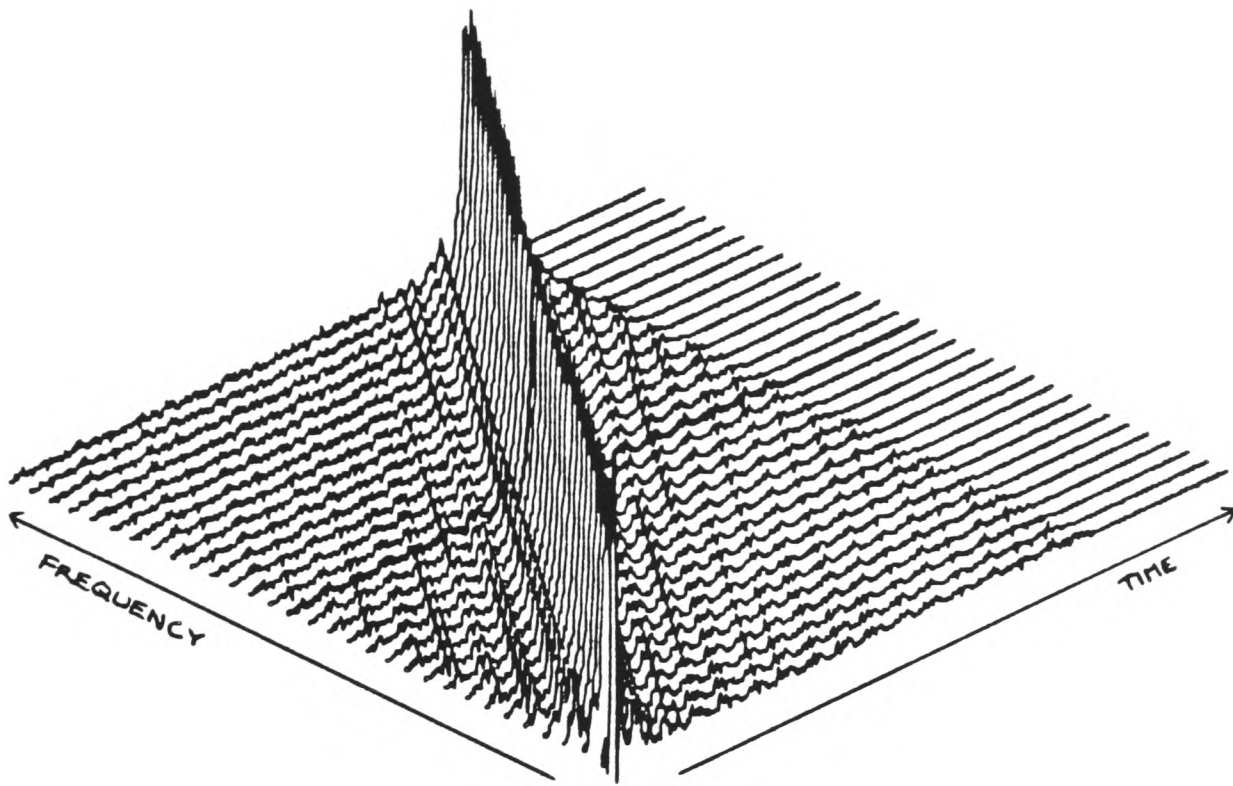


Figure 5.8: spectral evolution of chirp signal

System optimisation criteria

Attempts to optimise both parts count (number of chips in the system) and device count (number of different *FIRST* chip designs) may sometimes lead to conflicts. In a low-volume application such as that of the case study, production economies tend to favour a single multi-design wafer approach, rather than multiple single-design wafers. Therefore we are not so greatly concerned with minimising the device count, although we must bear in mind that test costs are related to device count.

The main optimisation criterion for this study is the number of pins in the system, as it is usually interconnect which provides problems of reliability in the field [148]. We should still pay some attention to parts count (which affects board space) and device count. Transistor count and physical sizes of chips should be restricted to maintain favourable manufacturing yields.

Chip optimisation criteria

A large part of our task here is to implement memory using more efficient WORDDELAY and CWORDDELAY primitives (where prudent) in place of the BITDELAY and CBITDELAY used throughout the soft model. WORDDELAY is a parallel store which discards input sign-extensions according to its parameters, and packs them on again on exit. CWORDDELAY uses the deterministic nature of control signals to synthesise a delayed version of its input.

Another issue is fan-out - the number of inputs driven by any particular output. In designs with word-synchronous control, for instance, we may find instances of the *cl* control driving more than the 6 inputs supported by the *FIRST* floorplan.

Finally, having settled on coefficient resolutions in FIR and FFT sections, we may relax the word-synchronous OPERATOR style used, re-implementing *Dft4* in minimal-latency form with its own dedicated *cl* line. This reduces transistor count by removing compensating delays, reduces system latency by 3 words to 23 words, and only increases pin count by 1 (on *CGen*).

5.8.1. The FIR section

The FIR section consists of 2 pairs of 6-point filter-multipliers (with associated state memory), and four 6-input adder trees. The data and event memory is shared between filter points in a pair, and so we couple one filter stage with its equivalent in the other filter in the pair. This allows us to minimise the number of pins in the data and event paths. We introduce a further parameter, *bound*, to allow OPERATOR *FilterSection* to contain optional pad delay compensation. All FIFO memory is re-implemented with word-oriented primitives for area efficiency, and further savings are realised by using the multiplier's facility to deliver its input data delayed by its latency. The CWORDDELAYs were partitioned to reduce their height to that of the multipliers.

Adder trees are small in terms of arithmetic hardware, but are large in terms of data transfers, having 6 inputs and one output. We could reduce the number of transfers by partitioning the FIR sections differently - if 2 contiguous filter points were together on a chip, then their products could be summed locally, and adder

trees would then have only 3 inputs. Unfortunately this compromises our data and event path, which must now enter and exit each chip twice, and in fact raises overall pin count. We choose the former scheme, which results in CHIP Section (Figure 5.9(a)).

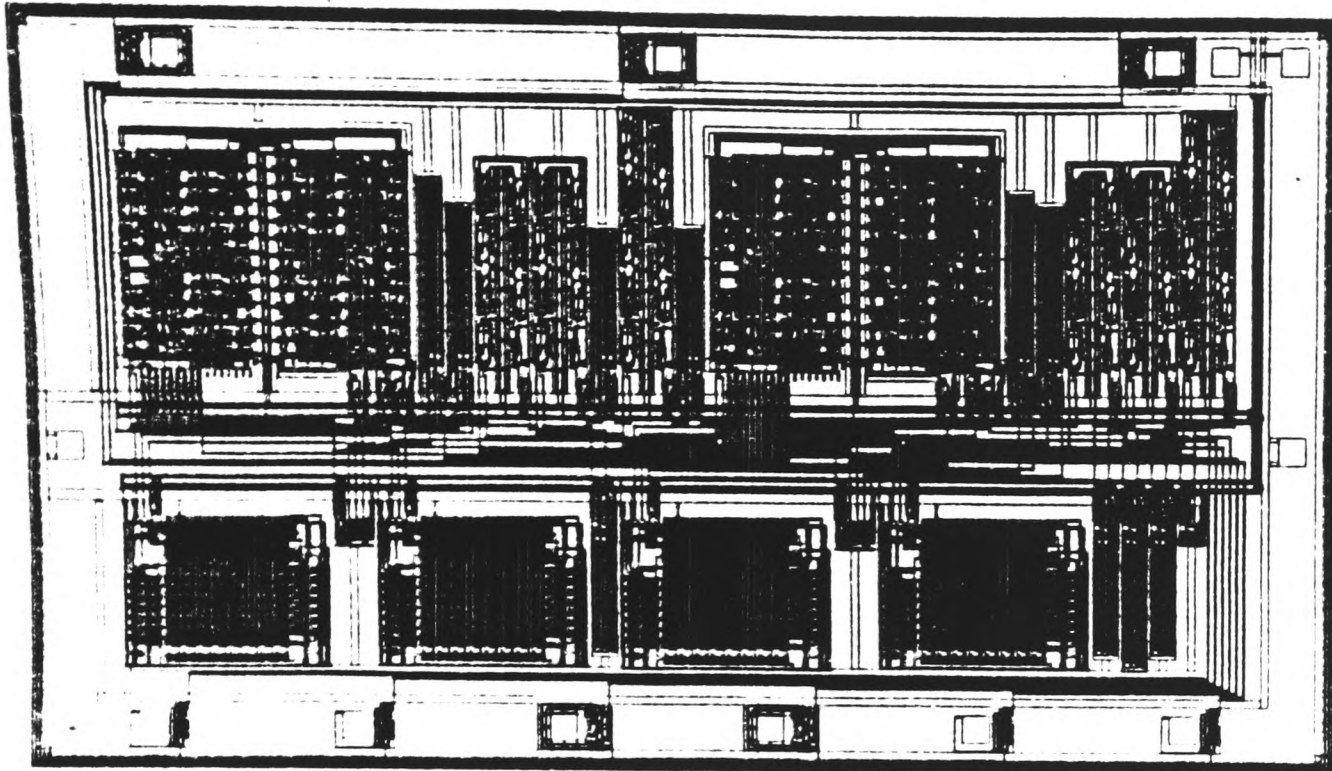


Figure 5.9(a): CHIP Section - size $6.40 \times 3.63\text{mm}$

The 4 adder trees could easily fit on one chip, but in this instance the pad count of 24 inputs and 4 outputs, with provision for associated linear-feedback shift-registers (LFSR) for self-test [7], is large enough to warrant partitioning of the trees into 2 CHIPS (*AddTree* - Figure 5.9(b)), each with 12 inputs and 2 outputs.

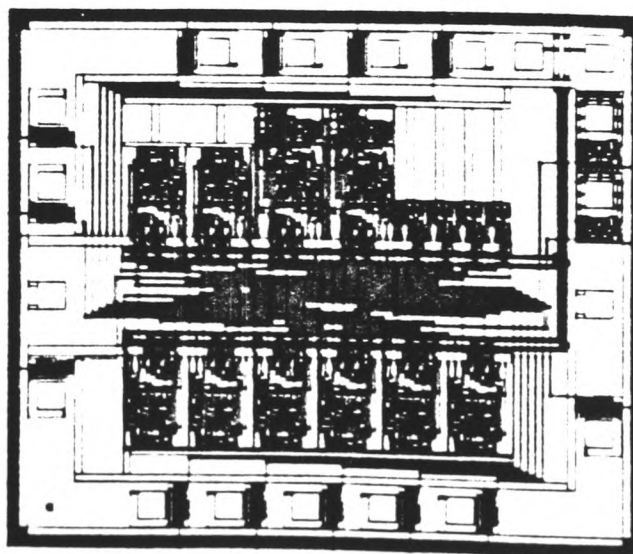


Figure 5.9(b): CHIP AddTree - size $3.07 \times 2.65\text{mm}$

This device exposes some of the drawbacks in using a simple floorplan such as that in *FIRST*.

5.8.2. The FFT section

The components of the FFT are complex multipliers, radix-4 DFTs, commutators and delays of various sizes.

The radix-4 Dft, like the adder tree, is small but pin-intensive. We produce the CHIP *Dft4* (Figure 5.9(c)) which contains a full complex radix-4 DFT, and for modularity reasons we choose to use it in the first stage even though inputs here are real. The latency is minimal at 4 bits (some control buffering was employed, and parameterised predelay pairs were passed to the component *Dft2* OPERATORS to maintain synchronism).

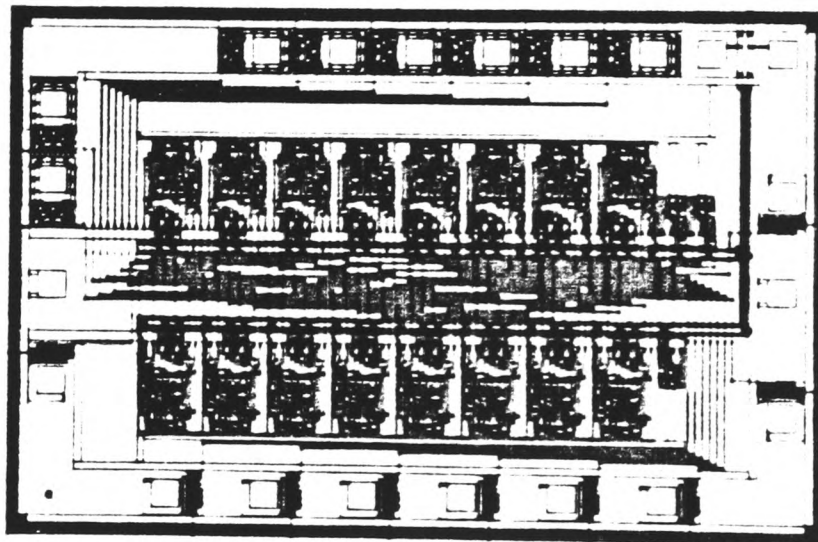


Figure 5.9(c): CHIP *Dft4* - size $3.95 \times 2.61\text{mm}$

One complex multiplier is chip-sized (CHIP *CmplxMul* - Figure 5.9(d)). By parameterising the latency of the ADD and SUBTRACT primitives on *CmplxMul*, we force the combined latency of *CmplxMul* and *Dft4* to be 2 words. Three instances of *CmplxMul* form the SUBSYSTEM *Twiddle*.

This leaves the question of how to partition the remainder of the system, which is mostly memory. Until now we have partitioned the pipeline into modular sections - however the shuffle networks are physically different at different stages (the pipes are different sizes). Where they exhibit symmetry is between real and imaginary data paths - these are identical. We therefore partition the shuffling

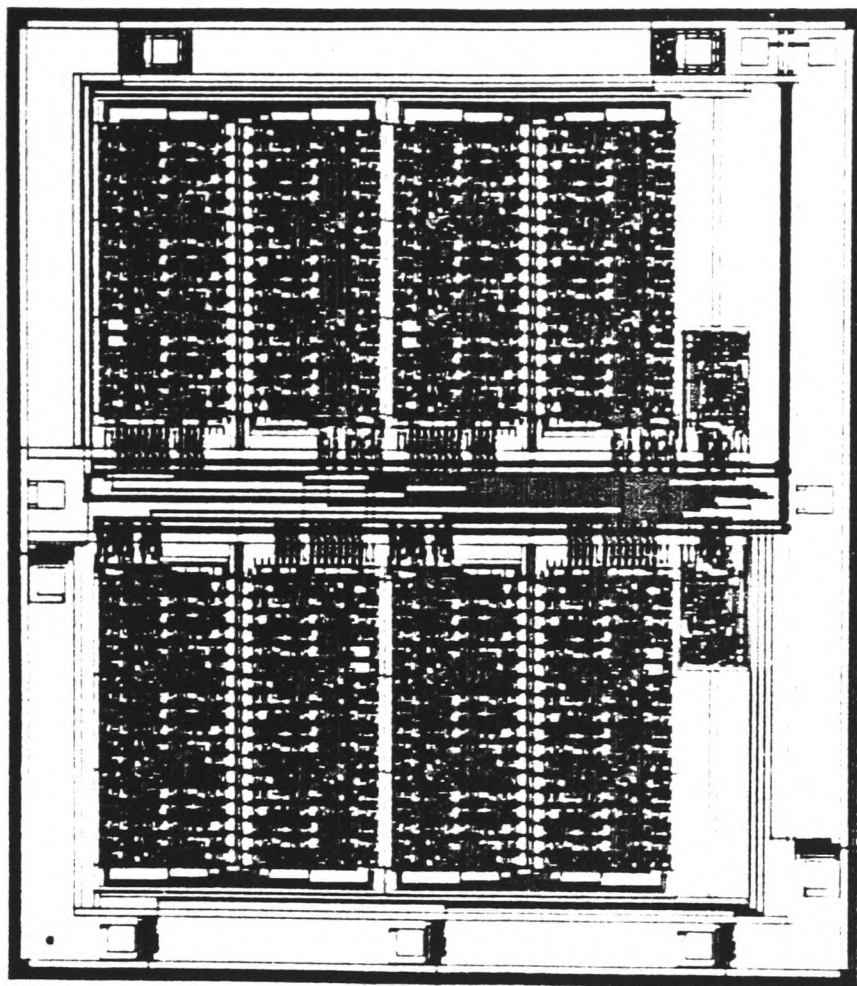


Figure 5.9(d): *CHIP CmplxMul* - size $4.26 \times 4.85\text{mm}$

(which also contains the twiddle compensation omitted from SUBSYSTEM *Twiddle*) in this manner, producing the *CHIP Commute* (Figure 5.9(e)). In this case we use sensibly partitioned BITDELAY primitives to implement FIFO memory, as use of WORDDELAY was seen to increase chip area and pin count.

Finally *CHIP Cgen* (Figure 5.9(f)) contains the CONTROLGENERATOR primitive which is the source of control waveforms in the system, along with delay elements which allow correct distribution of switching waveforms in the pipeline FFT. Levels 3 and 5 of control are required in a duplicate form, delayed by one quarter-period, by the FFT commutators. We implement this delay on *CHIP Commute* (thereby saving 4 pins at the cost of a little duplicated control delay).

The hard model was simulated in the same manner as the soft model, and behaved as expected.

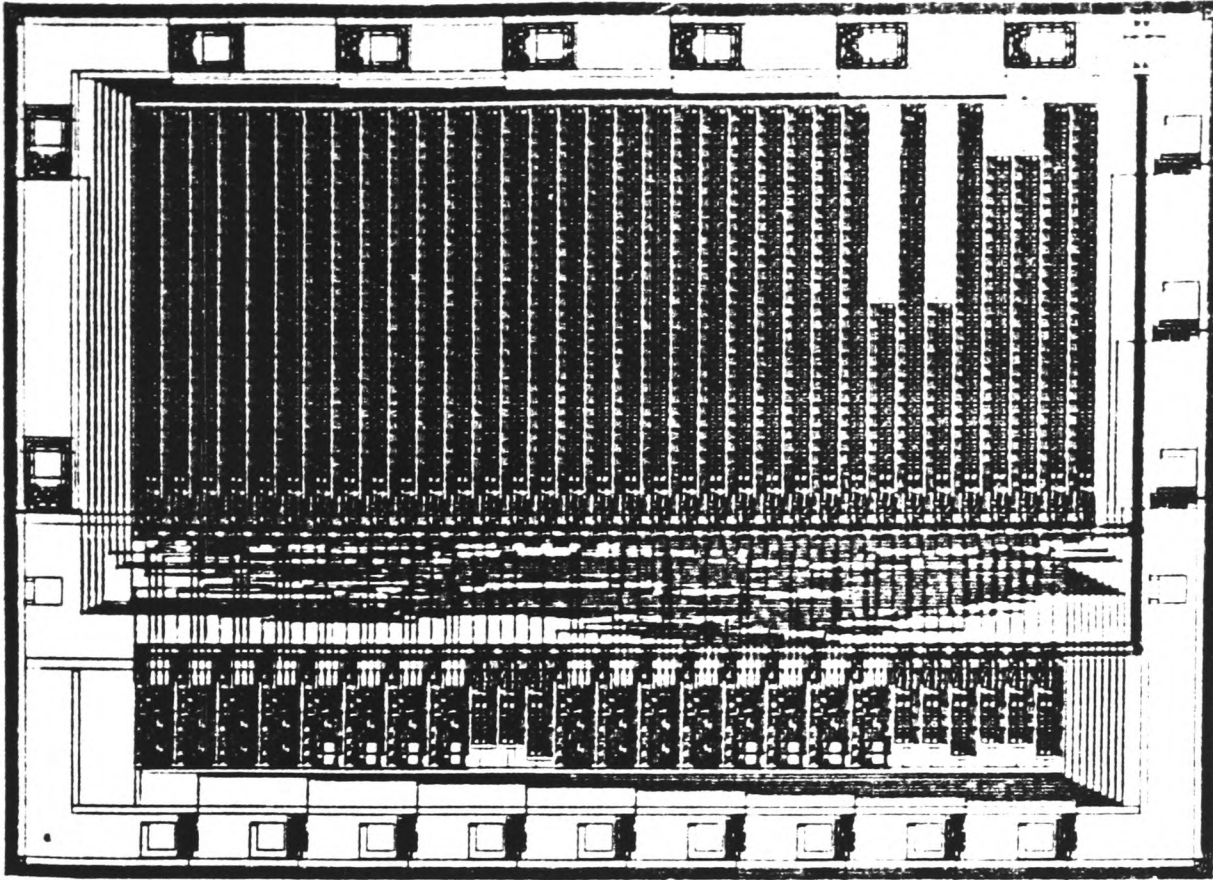


Figure 5.9(e): CHIP Commute - size $5.98 \times 4.31\text{mm}$

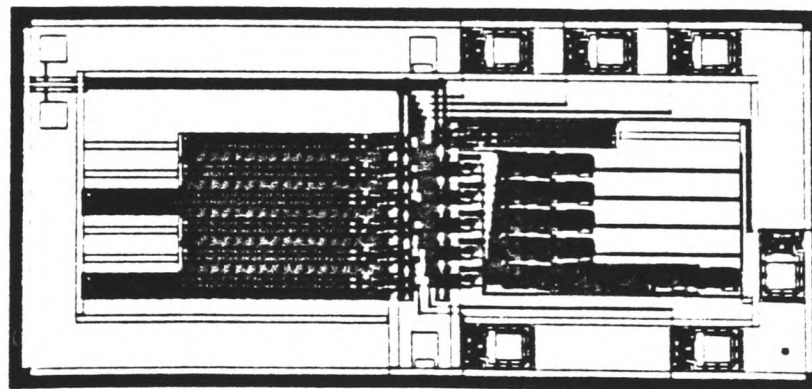


Figure 5.9(f): CHIP Cgen - size $1.90 \times 4.06\text{mm}$

5.9. Test strategy and confidence levels

Bit-serial computational elements are, by their nature, highly testable and straightforward to initialise (set to a known state) [5]. Pseudorandom test vector sets are therefore eminently suitable for testing bit-serial chips, provided:

- 1 All recursive loops in the data path are broken.
- 2 All bit-serial primitives in the system to be tested propagate random patterns.

The chip set has been designed and partitioned with these constraints in mind. The only loops in the system (coefficient loops in chip *Section*) can be broken by setting pin *pev* high. Furthermore, the system has been partitioned to maximise the access to internal points and primitives, thus increasing the diagnostic accuracy (at chip level) of a test program. All of the *FIRST* primitives used propagate random patterns, so the second testability requirement is met.

Test length

Figure 5.10 (taken from [5]) shows a series of testability curves for the single most random pattern resistant primitive in the chip set, the bit-serial multiplier. It can be seen that these curves are quite insensitive to multiplier coefficient length. In fact, for any length of multiplier (within reason), a 500-bit pseudorandom pattern test will result in a test confidence level in excess of 99% for the multiplier. As the other primitives will all be tested to an even higher degree, the overall test coverage may be regarded, in a probabilistic sense, to be 100%. Such a probabilistic measure is inherent in *FIRST*'s test strategy [5], as it avoids the crippling exercise of full fault simulation.

It is envisaged that this system will be subjected to an off-line test, to avoid the system control overhead implied by the scheduling of totally autonomous test. As mentioned above, the propagative randomness property of bit-serial systems obviates the need for automatic test-pattern generation. The *FIRST* simulator model may be driven with orthogonal pseudorandom sequences, and the activity on all output pins monitored. When testing hardware, the same stimuli will produce identical results to the simulator model unless a fault is present.

Inclusion of full self-test capability to *FIRST* systems has been investigated [5], and may be included in a manner transparent to the *FIRST* user. In this case pseudo-random sequences are generated local to each data input pin, and data compression registers at the data output pins record the test results. The system under study was partitioned with a view to supporting the eventual inclusion of

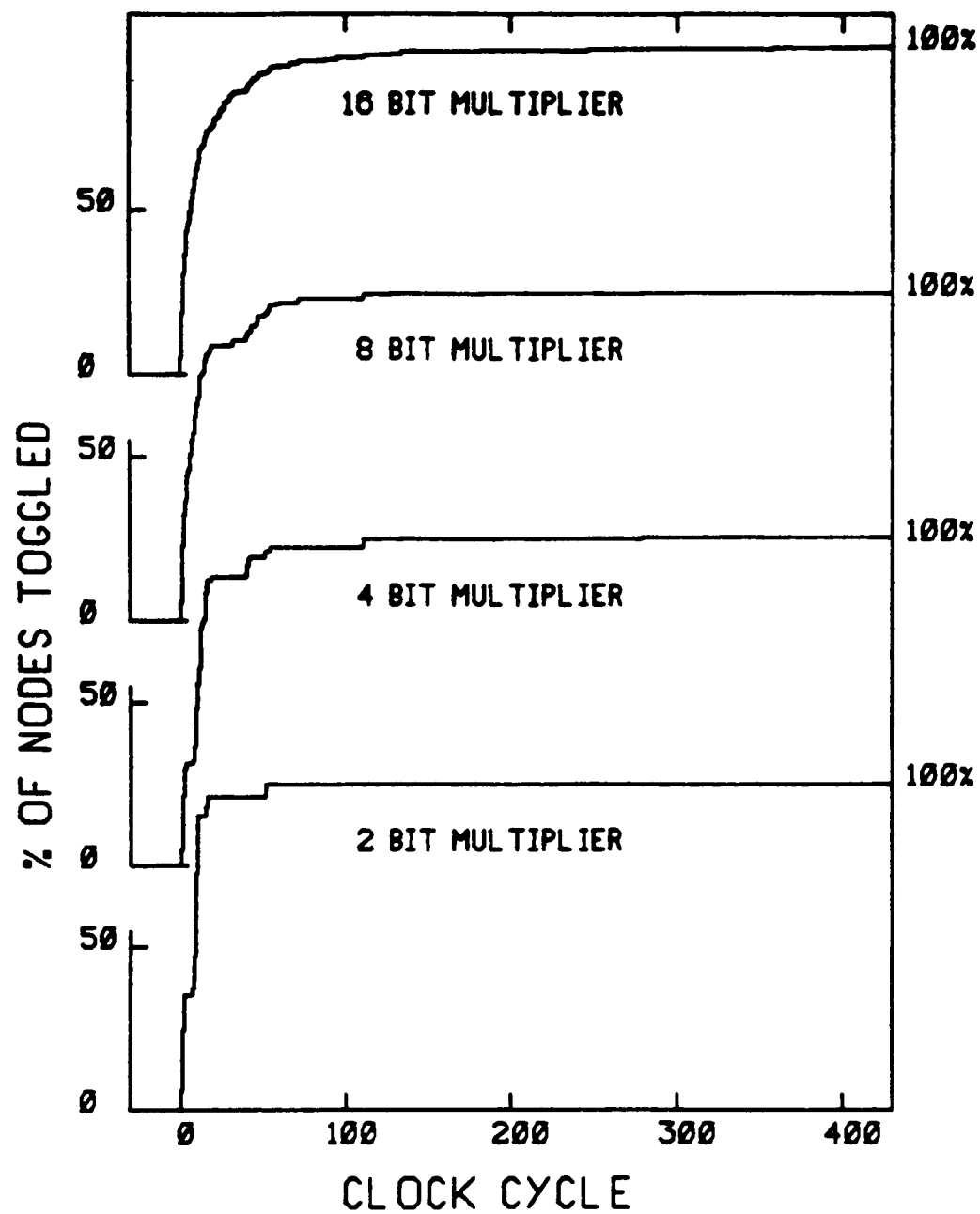


Figure 5.10: multiplier testability curves for various coefficient lengths

autonomous test.

5.10. A critical appraisal of *FIRST*

While the existing set of *FIRST* primitives is able to provide adequate architectural components for the PPN, there is a strong case for the development of further application-specific primitives. In this case custom complex multipliers and adders, or even full butterfly stages, may be merited. Some more sophisticated data

switching elements would also be advantageous here, as would be adders with built in truncation (i.e. 'averagers') to avoid word growth and improve dynamic range. Filterbanks might benefit from custom inner-product step processors. Although the *FIRST* function library is capable of providing the computational elements necessary to realise the PPN filterbank, the *functional* inflexibility of *FIRST* stands in the way of more efficient realisations. Chapters 6 and 7 describe advances in serial-data architecture which might lead to more efficient implementations of function.

The chip count of the PPN filterbank stands at around 25. It is estimated that a realisation in a more modern technology, 2- μ m double-metal CMOS, would consist of only 3 chips [118][†]. However current *FIRST* users are obliged to partition systems as dictated by the yield and integration levels of 5- μ m nMOS. The *technological* inflexibility of *FIRST* prevents advances in integration.

Although the best available mixture of FIR and FFT architectures was chosen (obviating memory interfaces), some inefficiency had to be accepted in the FIR filterbank. This subsystem could have been implemented in 8-bit arithmetic, however the FFT demanded 16-bit precision. Thus the filterbank was required to idle on sign-extensions for 50% of the time, compromising the elegance of the overall system architecture. Were variable-throughput elements available, the FFT could process 16-bit data at the rate at which the filterbank output 8-bit data. The filterbank could then be halved in size. Although individual computational elements such as complex multipliers and adders might increase in size, the FFT could in turn be converted from a radix-4 pipeline to a smaller radix-2 pipeline [145, 72] (here 'radix' refers to *transform* radix [130], not *number* radix), resulting in net area savings. The *operational* inflexibility of *FIRST* prevents architectural refinements of this nature.

This chapter has demonstrated advances in serial-data architectural knowledge, and the power of *FIRST* to realise efficient solutions to DSP problems despite the three limitations identified above. The remainder of this thesis addresses means to overcome the three inflexibilities of *FIRST*.

[†] published work by the author.

Chapter 6

Special serial-data techniques for area reduction

In the previous chapter, *FIRST* has been shown to be *functionally*, *technologically* and *operationally* inflexible. The first and third inflexibilities are addressed in this and the next chapter respectively. The concepts behind an advanced serial-data cell-library for DSP applications, based on the S/P multipliers introduced in Chapter 3, are outlined. A proposed solution to the technological inflexibility is the subject of Chapter 8.

This chapter presents novel, area-efficient adaptations of the serial-data multiplier form. Firstly an alternative functional partitioning of matrix and vector processing architectures is proposed, based on symmetric-coded distributed arithmetic. This technique offers area-savings at no loss in throughput [149, 150][†]. It is shown how such architectures may be cascaded. Finally a novel form of incremental multiplier is suggested, which is optimised for computation of squares and sums-of-squares [108].

6.1. Overview of vector computation

A multiplication is an unconstrained 1-D sum of (weighted) PPs. The dimension represents one of the two input operands - the weight of whose bits is a function of dimension index. We refer to the input operands as *data* and *coefficient* respectively, and note two common differences of usage in DSP applications. Firstly, the coefficient is often known *a priori* (unlike data which either arrive from external sources or are freshly-derived from previous computations). Secondly, the precision of representation may differ (the coefficient is often represented by fewer bits). In S/P architectures, one computational dimension (usually the data-bit index) lies along the time axis.

A fundamental form of vector product is the *dot*, or *inner* product (IP) [151]. The IP of two vectors is formed by summing the pairwise products of the vector

[†] published work by the author.

elements. An IP is then an unconstrained two-dimensional sum of PPs (the 2nd dimension being vector length). Once again, we note the difference of usage between data and coefficient (in this case vectors). By permuting and/or factoring summation indices, several different approaches are made possible. Classical multiply-accumulate techniques [27] put the vector index outermost, while DA [152,153] has the data-bit index outermost. By factoring the index of vector length, architectures may be realised which yield an optimal mixture of these two techniques.

Matrix-vector multiplication extends the list of favourable properties associated with coefficients, in that there are often further properties of symmetry in the coefficient matrix to be exploited. We shall demonstrate these advantageous properties in later examples.

6.1.1. Some carry-save approaches to vector computation

The properties of carry-save arithmetic seem particularly well suited to the computation of vector products, i.e. the unconstrained summation of single-bit products in three dimensions. Bit-level systolic arrays [154,66,155] are an efficient means of implementing such architectures. These architectures are characterised by minimal control and communication overheads, and potential clocking rates are limited only by the logical complexity of processing elements. A single-chip processor [156] has recently appeared, based on these principles.

Denyer and Myers [45] proposed arrays of carry-save adders which accumulated inner-products across each bit plane in bit-parallel carry-save fashion (MS-plane first), using the free inputs of the next-plane computer for accumulation. Cappello & Steiglitz [81] formalised this concept. Danielsson noted that families of convolvers could correspond to S/P multipliers, as convolution has the same structure at word-level as multiplication has at bit-level [70].

These architectures follow good VLSI practice, and result in area-efficient realisations of vector computers. All feature a high degree of modularity and regularity. However they exploit neither symmetries and redundancies in computation, nor preknowledge of coefficients, as do distributed arithmetic architectures. Although a combination of techniques might yield interesting results, we are more

concerned with implementing a set of serial-data modules which can be configured to realise arbitrary fixed-function computational networks. For this reason we eschew the systolic architectures, in deference to the 'approach'.

6.2. Serial/parallel symmetric-coded distributed arithmetic

This section presents a methodology for synthesis of area-efficient, high-performance VLSI architectures for vector and matrix multiplication. Use is made of distributed arithmetic techniques [152,153]. Three fundamental computational elements are employed in the composition of these architectures: memory register, multiplexer, and carry-save add-shift (CSAS) computer. 2C S/P carry-save accumulation provides performance, while the use of symmetric-coded DA in CSAS computers eliminates redundant computation to effect area-savings.

DA provides the facility to compute the sum of several products concurrently, in architectures which exhibit the same structure, regularity and modularity as do scalar multipliers. In fact a scalar multiplier is a trivial case of a DA architecture.

6.2.1. The symmetric-coded serial/parallel multiplier

Orthodox S/P multipliers, as described in Chapter 3, use 2C coding throughout. As we know of no better data-coding for the addition (accumulation) operation, this results in minimal hardware and maximal throughput. Figure 6.1 shows the 2C coded S/P flush multiplier (the following conversion process is equally applicable to fractional S/P and Lyon multipliers).

To assist in the development of matrix-vector architectures, we choose to code the data word in symmetric, offset-binary (OB) form, where logical 0 is interpreted as -1 [76]. This alters the role of data bits in the computation - logical 0 now effects the subtraction of the coefficient from the PPS, instead of the addition of zero.

Data conversion to OB is easily accomplished by MSB inversion. 2C data word A consisting of bits a_i converts to OB coded word A' as shown:

$$A = -a_0 + \sum_{i=1}^{n-1} a_i 2^{-i}, \quad a_i \in \{0,1\} ,$$

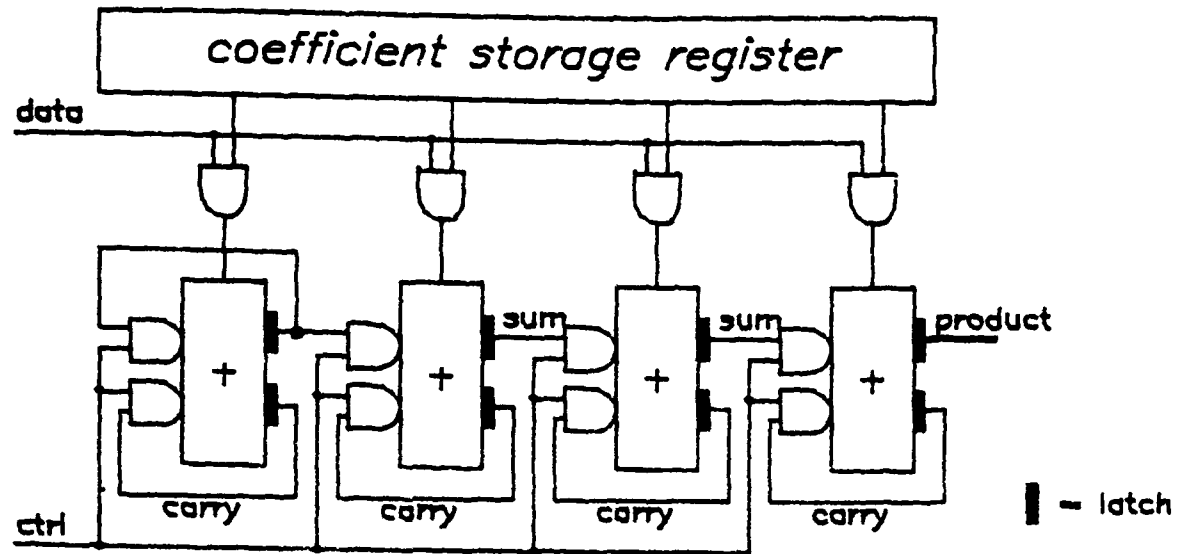


Figure 6.1: 2C coded S/P flush multiplier (4-bit coefficient)

$$A' = \sum_{i=0}^{n-1} a_i 2^{-i-1}, \quad a_i \in \{-1, 1\}$$

Error compensation by parallel load of the coefficient

It can be seen that

$$A' = A + 2^{-n} \quad 6.1$$

i.e. a small representational error results from the change of code to OB.

Consider the product P of 2C-coded data word A with coefficient C , and product P' of OB-coded data word A' with coefficient C .

$$\begin{aligned} P &= AC, & P' &= A'C \\ & & &= AC + 2^{-n}C \\ & & &= P + 2^{-n}C \end{aligned}$$

We see that the change of code from 2C to OB results in a representational error of $2^{-n}C$, which may be removed by subtracting the coefficient word at LSB-time (on

commencement of product computation). As the CSAS computer is incapable of explicit subtraction, this may be accomplished by adding the 2's complemented coefficient, i.e. by bit-inversion and incrementing (implicit subtraction). To this end, the inverted bits of the coefficient are used to load the free carry-loops in the main array at LSB-time (cf. clearing in the 2C version). The PPS input to the stage occupied by the coefficient LSB (i.e. the last stage) must be set at LSB-time to perform the necessary increment (all others are cleared).

As data is OB-coded, the coefficient word is either added to or subtracted from the PPS, depending on the broadcast data bit. Again, coefficient subtraction is performed implicitly (the incrementing bit is simply delayed, inverted data). Thus the OB S/P multiplier contains an XNOR-gate for bit-product formation (instead of the AND gates of the 2C version), and an extra CSAS cell at the end of the array for incrementing. This incrementing adder combines with the residue assimilation adder to form a double-precision adder from whose two output wires various product formats may be selected. Figure 6.2 shows the OB S/P multiplier.

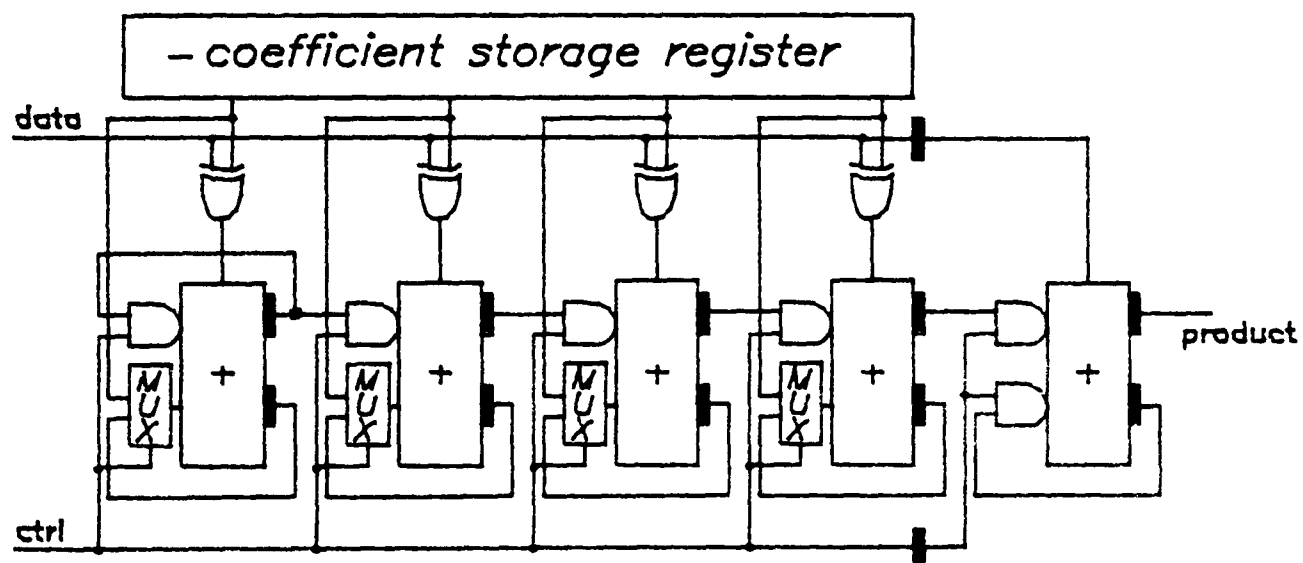


Figure 6.2: OB coded S/P flush multiplier (4-bit coefficient)

Note that the weight of data-bits is different between 2C and OB (the former is twice the latter). Care should be taken in aligning the addend input, and interpreting the weight of the coefficient and the output product.

Error compensation by left-shift and decrement

The above scheme necessitates loading of the coefficient into the carry loops at LSB-time. This causes a considerable increase of logical complexity in the basic computational cell, with corresponding area and performance costs. A second method of coding-error compensation cancels the data error directly, by pre-processing data bits before broadcast. Manipulation of eqn 6.1 produces the expression:

$$A = A' - 2^{-n}$$

$$\text{i.e. } 2A = 2A' - 2^{-n+1}$$

Thus preprocessing takes the form of a left-shift and decrement operation. Output data must be right-shifted for subsequent correct interpretation of the product.

This technique increases the latency of the OB multiplier, requires a guard-bit on input data thereby reducing dynamic range, and affects modularity in adverse manner. These points make it a less likely candidate for implementation. Pipelining of bit-product formation in the former scheme can be used to reduce cell complexity, leading to a more compact, modular architecture.

6.2.2. The serial/parallel inner-product computer

So far we have described a modification to the 2C S/P multiplier, allowing it to handle OB-coded data inputs. The price paid for this is the extra low-precision adder on the output of the main array, and increased cell complexity. However the S/P multiplier is now in the form where, with a little further modification [157],[†] it can compute inner products directly, using DA.

DA [153] replaces the multiplications involved in an IP computation with a series of memory look-ups. A set of 'partial inner products' (PIPs) made by convolving the coefficient vector with all possible bit-patterns from the data vector is precomputed and stored in memory. The PIPs are accessed (addressed) by the

[†] published work by the author.

actual bit-pattern across each bit-plane of the data vector, and accumulated (with the correct binary weight) to form the IP.

If data are coded in OB, the full PIP-set exhibits negative symmetry [76]. To exploit this property, we designate an arbitrary coefficient word as 'master', also referring to the data word associated with this coefficient in the IP computation as master. The master data-bit may then be removed from the address word and the memory-size halved. This bit instead serves as an 'add/subtract' instruction to the accumulator. Viewed in this light, the OB S/P multiplier contains a single-word 'memory', accessed by a 'zero-bit address word', i.e. look-up is trivial.

We now describe the conversion of the OB S/P multiplier into a 2-point IP computer. Instead of storing one coefficient word C , we introduce a second coefficient word D , and store the 2 PIPs K and K' , where

$$K = \frac{C + D}{2} , \quad K' = \frac{C - D}{2}$$

The factor of 2 prevents word-growth in PIPs, and compensates for the factor of 2 weight difference between OB and 2C data codes. We take 2 serial data words A and B as input, choosing (say) A as master. A is then broadcast as data to the CSAS array, and $(A \text{ XNOR } B)$ is used to select either K or K' . If the bits of A and B are equal in any bit-plane, the 'sum-PIP' K is selected - if unequal, the 'difference-PIP' K' is selected. If the 'master' bit is 1, the PIP is added, if not it is subtracted. Thus the modified S/P multiplier is capable of computing the inner-product step $AB + CD + E$ (where E is the addend), at little extra hardware cost.

Through a simple, recursive procedure, this principle can be extended to compute longer IPs:

For each additional data-coefficient pair:

Replace each register with a multiplexer and register-pair,

Load the register-pair with the old PIP \pm the new coefficient,

Select multiplexer output by XNOR of master data-bit and new data-bit.

Error compensation

Consider the IP $\sum P_x$ of 2C-coded data vector A_x with coefficient vector C_x , and product $\sum P'_x$ of OB-coded data vector A'_x with C_x .

$$\begin{aligned}\sum P_x &= \sum A_x C_x, & \sum P'_x &= \sum A'_x C_x \\ & & &= \sum (A_x C_x + 2^{-n} C_x) \\ & & &= \sum P_x + 2^{-n} \sum C_x\end{aligned}$$

Thus the 'sum-PIP' $\sum C_x/2$ loads the carry-loop on commencement to compensate for OB data-coding.

Figure 6.3 summarises the evolution of this class of DA architectures via the recursive composition procedure, starting from the S/P multiplier.

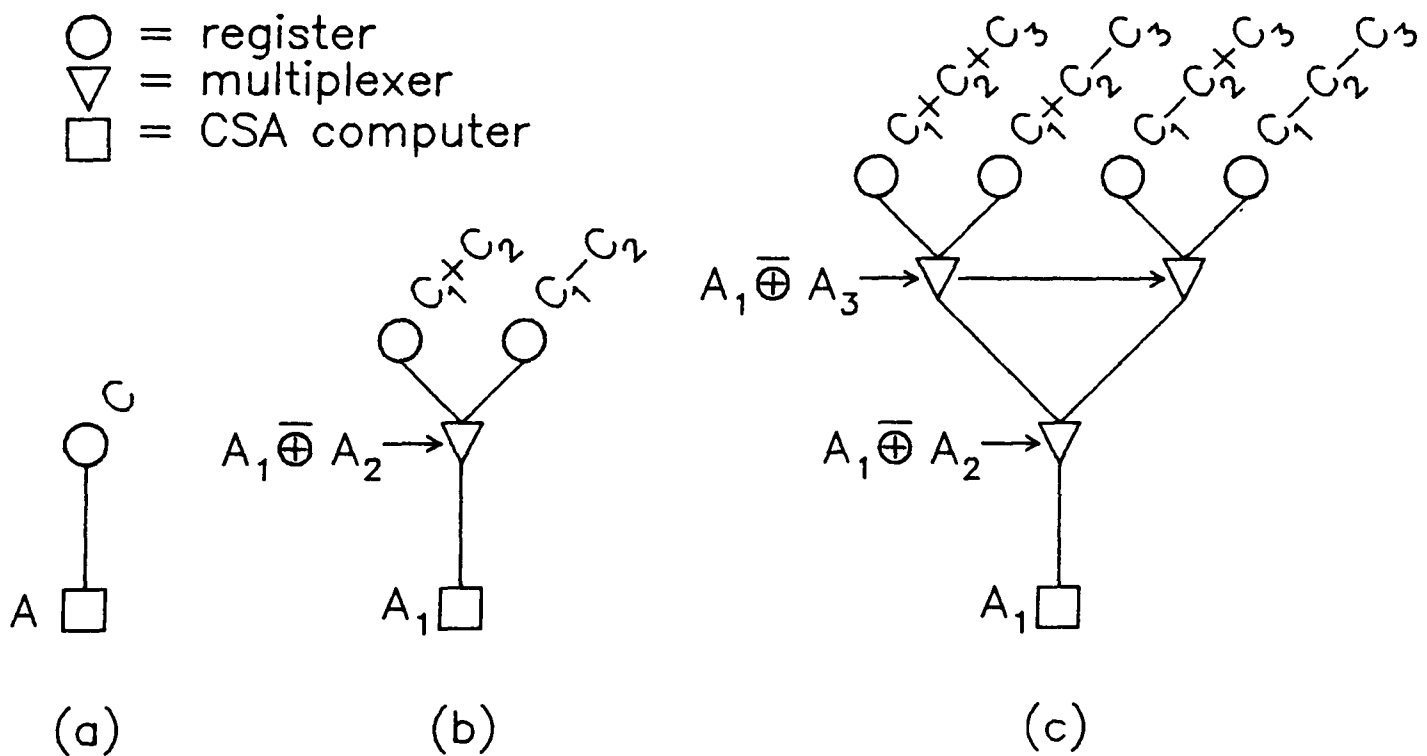


Figure 6.3: real IP computer evolution

Some abstraction is necessary to contain detail - for instance we are not concerned

with loading/unloading operations, carry-setting and cascadability (these are the same in all cases). We restrict ourselves to 3 architectural elements: registers, multiplexers and CSAS computers. The architectures are viewed 'end on', i.e. data flow is out of the page. Master bits are shown beside computers, while selection functions point at selectors.

Figure 6.3(a) is the abstracted version of the S/P multiplier of Figure 6.2, with a single register to hold coefficient word C and a single CSAS computer to form the product from data word A .

Figure 6.3(b) shows the modification of the S/P multiplier to form a 2-point IP $A_1A_1 + A_2C_2$. We replace the coefficient register with a multiplexer and register-pair. Here A is master, and the function $A_1 \text{ XNOR } B_1$ drives the multiplexer, which selects one of the PIPs (these are represented as $C_1 + C_2$ and $C_1 - C_2$, although PIPs are actually stored as half these values).

Figure 6.3(c) shows the extension of this technique to form the 3-point IP $A_1A_1 + A_2C_2 + A_3C_3$. Here A_1 is master and A_2 and A_3 are used for data selection, by XNOR with A_1 .

It should be noted however, that while savings in CSAS elements are linear, PIP storage costs grow exponentially. Depending on the technological implications of adding storage, a point will soon be reached where the DA approach is less attractive than the conventional [158]. For this reason, we propose a mixture of DA and conventional techniques for longer IP computations - this is effected by factoring the vector length index as described earlier.

6.2.3. Architectural case studies

Armed with the knowledge of how to construct IP computers, we may now review some of the matrix and vector architectures which can be synthesised with these techniques. Here we treat coefficients as known matrices operating on vectors of incoming data, exploiting where possible the properties of symmetry exhibited by these matrices.

Matrix-vector multiplication

The 2-point IP computation described above may be expressed in matrix notation as shown.

$$E = \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

While general matrix-vector computation of the form:

$$\begin{bmatrix} G \\ H \end{bmatrix} = \begin{bmatrix} C & D \\ E & F \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

may be executed on a pair of unrelated 2-point IP computers, the centrosymmetric matrix computation:

$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} C & D \\ D & C \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

may be performed on a simple variation of the architecture of Figure 6.3(b) (the addition of a second CSAS computer). Figure 6.4(a) depicts this architecture.

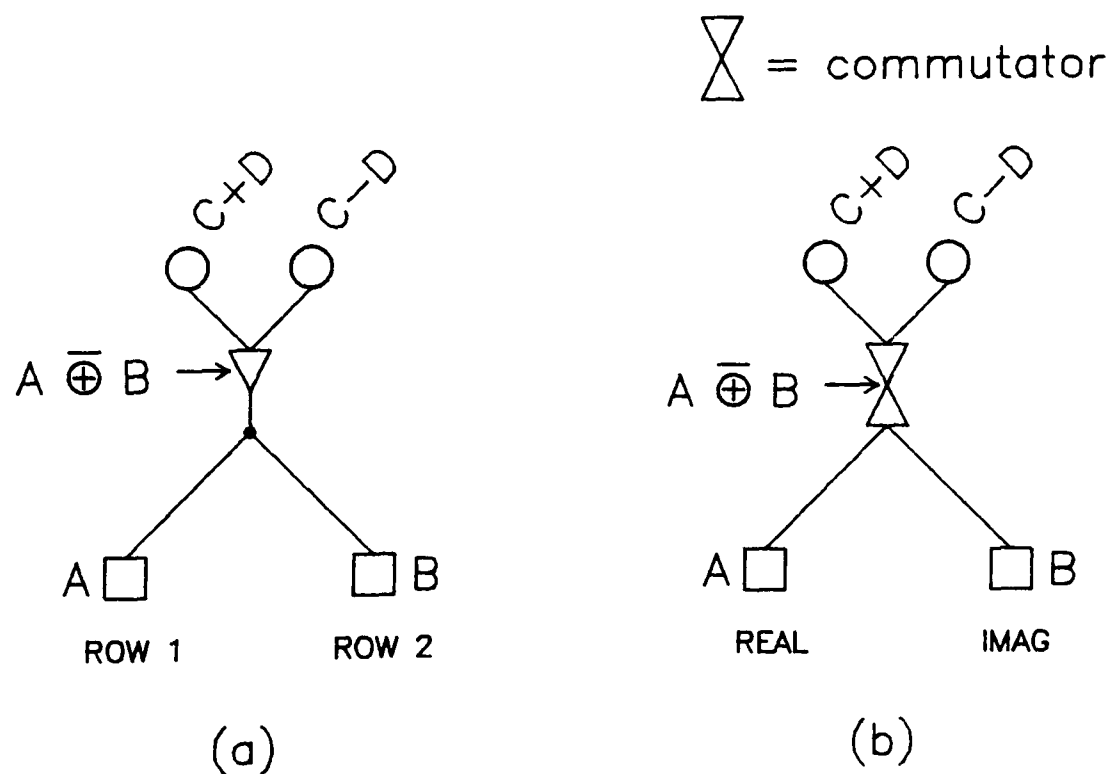


Figure 6.4: matrix-vector computer evolution

Due to centrosymmetry, PIP-selection is mutually *inclusive*, i.e. the selected PIP is

used in both computers. Note that B is master in the second computer, as B is associated with the master coefficient in the implicit IP computation which produces output F. While this structure finds limited application in DSP (hyperbolic rotation is one example of its use [141]), a further slight modification transforms it into a form of matrix-vector computer which is very common indeed.

Complex multiplication

The operation of complex multiplication (plane rotate and/or scale) occurs frequently in digital signal processing, for example in Fourier transformation [27], orthogonal filtering [159], and waveform generation [160]. Chapter 5 reviewed some of the many approaches to this computational problem.

If we designate data word-pair A,B and coefficient word-pair C,D as real and imaginary components of a complex numbers **A** and **C** respectively, a further slight modification to the architecture of Figure 6.4(a) may perform complex multiplication [161]. The complex product **E** = **CA**, where **E** = E + jF may be evaluated by the matrix computation:

$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} C & -D \\ D & C \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$$

Although the centrosymmetric property no longer holds, the coefficient matrix now displays the equally useful property of mutually *exclusive* PIP-selection. Here the sum-PIP in the implied imaginary IP computation equals the difference-PIP in the implied real computation (and vice-versa). Instead of selecting one PIP via a multiplexer for use in both computers, we steer both PIPs through a commutator (2-from-2 data selector). This architecture (Figure 6.4(b)) is similar to the complex multiplier embedded by White in an FFT processor [143]. This architecture has recently re-appeared in fully-parallel [144] and serial-pipeline [48][†] forms, this last example being a direct modification of the Lyon multiplier.

[†] published work by the author.

Complex inner-product

We may express the previous computation in complex notation:

$$\mathbf{E} = \mathbf{CA}$$

and extend the above concept to the complex inner-product computation:

$$\mathbf{E} = \begin{bmatrix} \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}$$

In similar fashion to the real arithmetic case, we form an IP computer by removing the register of a multiplier, and replacing it with a register-pair and multiplexer. Here registers are complex, i.e. they contain *pairs* of numbers. Thus a 'register-pair' in this case comprises 4 real registers, while a 'multiplexer' is a 1-from-4 data selector.

Extending the selection methods of the complex multiplier, the PIP-set may be split into a 'sum-set' and a 'difference-set', with mutually exclusive selection. However selection functions within these sets are complicated by the fact that selection *within sets* depends on the target computer. This follows from the asymmetry of the scalar coefficient matrix:

$$\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} c_1 & -d_1 & c_2 & -d_2 \\ d_1 & c_1 & d_2 & c_2 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ a_2 \\ b_2 \end{bmatrix}$$

By judicious arrangement of PIP storage, we may at least ensure that selection functions are shared between PIP-sets. These functions are 'quasi-exclusive', so-called because, like the exclusive-OR and -NOR functions, they form diagonals in the Karnaugh map (Figure 6.5). While exclusive functions form alternate diagonals, quasi-exclusive functions divide the map into two diagonal regions. Figure 6.6 shows the 2-point complex IP computer, with selection functions appropriate to the PIP arrangement.

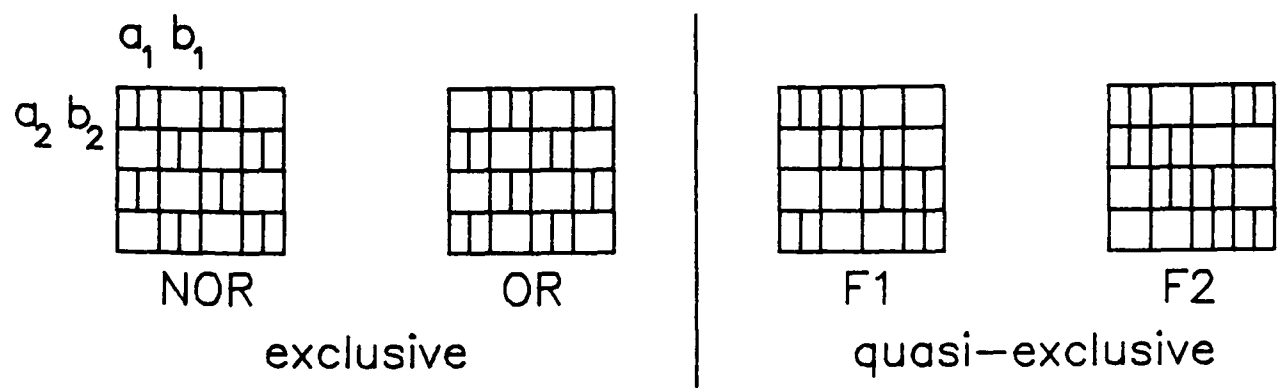


Figure 6.5: Karnaugh maps of exclusive and quasi-exclusive functions

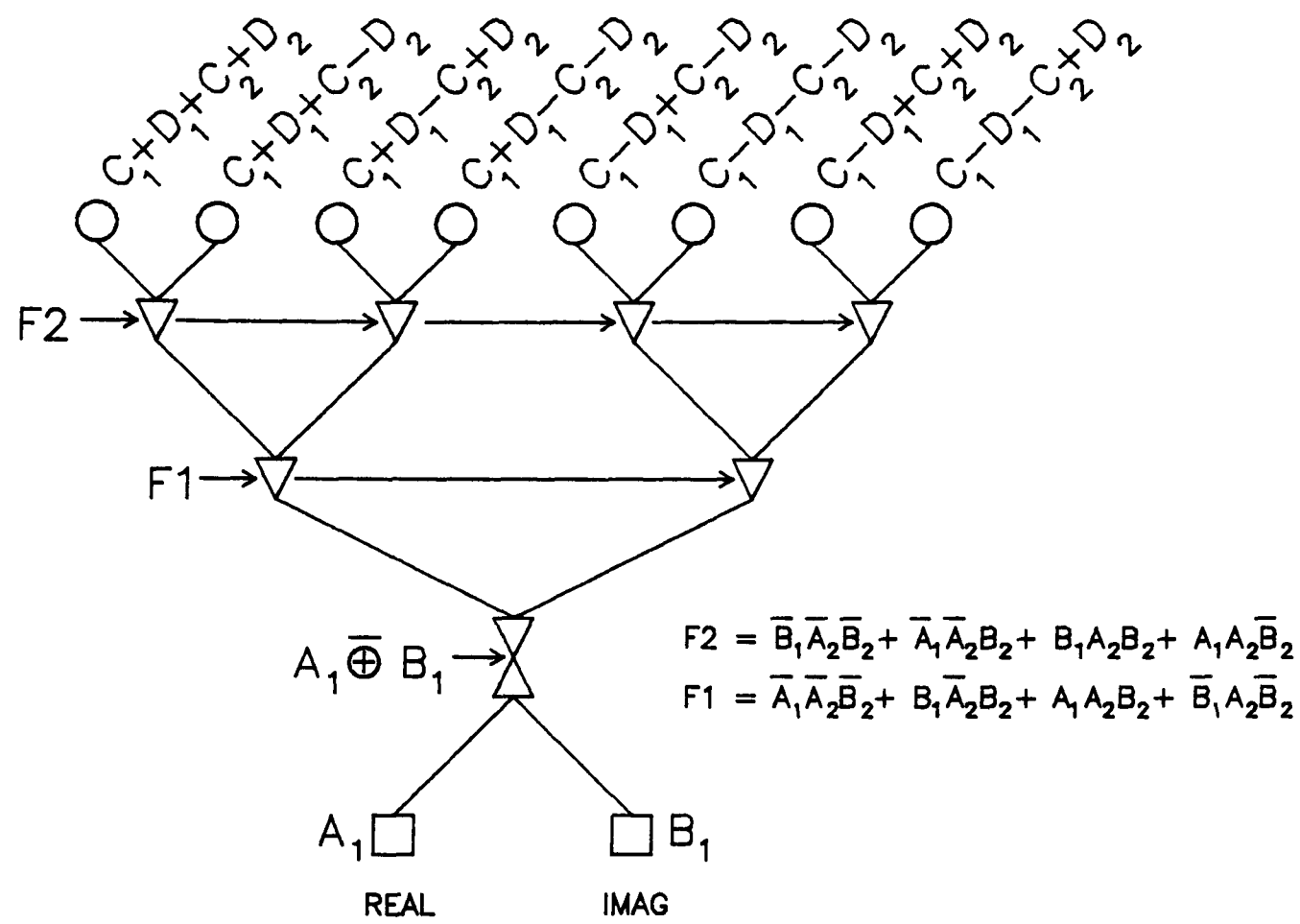


Figure 6.6: complex IP computer

A 3-point complex computer may be formed by replacing each register with a 4-register-3-multiplexer combination, and so on. However register count and multiplexing depth begin to dominate beyond the 2-point complex IP, as does the fan-in (hence area cost) of selection functions.

6.2.4. Architectural synthesis

We have seen how 3 simple elements may be combined in different ways to form various architectures for matrix and vector computation. Building on a single CSAS array, the construction rules for IP computers are procedural and recursive. Matrix computation follows easily, by addition of further CSAS computers and association of master data and coefficient in each implicit IP computation.

Procedural construction lends itself readily to computer automation in a silicon compilation environment [7]. With construction rules encapsulated in composition procedures, we envisage assembly of DA architectures in response to a single-line call in high-level language, rather than explicit calls to component modules.

6.2.5. Comparison with conventional approaches

To illustrate the area savings afforded by the DA approach, we compare the example architectures with the standard approach (SA), where storage is shared whenever possible (Table 6.1). S/P hardware costs are $O(m)$, where m is coefficient/PIP wordlength. We neglect $O(1)$ costs.

Table 6.1: Comparison of standard approach and DA						
function	register		mux		CSAS	
	SA	DA	SA	DA	SA	DA
2-point IP	2	2	-	1	2	1
3-point IP	3	4	-	3	3	1
2-pt. matrix-vector	2	2	-	1	4	2
complex multiply	2	2	-	1	4	2
2-pt. complex IP	4	8	-	7	8	2

The DA solution uses multiplexers and sometimes extra storage, but always less CSAS computers. The hardware costs of these elements vary with technology, nonetheless we suggest that the reduction in CSAS hardware afforded by the DA approach outweighs the increase in storage/selection costs over conventional

approaches.

Although area-efficient, DA appears to be 'storage heavy'. However if bit-serial throughput enhancement techniques [162] are employed, then the ratio of active logic to storage increases, accentuating the area savings of the DA approach.

DA requires the generation of some logic functions of input data bit-planes, and extra wires for their broadcast. Word-growth may occur *in PIPs*, necessitating perhaps one or two extra stages to maintain the accuracy of conventional realisations. Also it should be noted that the DA approach requires 'precomputed' PIPs to be available. If coefficients can only be provided in standard form, then a network of bit-serial adders and subtractors must be provided for PIP calculation, with attendant time and area penalties.

6.2.6. Distributed arithmetic in context

DA was proposed as a method for avoiding the use of standard-part multipliers in FIR/IIR filtering [153,163]. From the outset the two approaches were classed as diametrically opposite - they have been directly compared on several occasions, e.g. [164,165]. This has led to a general perception of DA as a 'ROM-accumulator' technique, involving memory technology in VLSI realisations with the area overhead of control and addressing logic.

We have demonstrated that 'memory addressing' (at least for small problem sizes) in VLSI is merely a data-steering operation, governed by simple logical functions on the incoming data bit-planes. Computation is performed on CSAS arrays, exactly as done in multipliers. Thus DA is more a *modification* than a *replacement* of multiplier technology.

Complexity of DSP algorithms is often expressed in terms of 'multiplier count'. We suggest that hardware partitioning into registers and CSAS computers (rather than multipliers) coupled with the use of DA in synthesis might result in improved complexity analysis techniques.

6.2.7. Cascading

The concept of DA has been introduced, and its usefulness in the computation of short vector and matrix products demonstrated. In many cases, these structures will be used in long cascades, e.g. in the computation of vector inner products [151]. Cascading issues relate closely to those of the conventional approach, as we propose to sum globally over the index of vector length. The only difference is that we have factored this index and nested one of those factors inside the bit-index - the other is outermost as usual.

Recall that these structures have one free input in the serial (temporal) dimension, and two potentially free inputs in the parallel (spatial) dimension, all of which can find use in cascading. The carry-loop could be freed up in all but the initial DA processor of the cascade, by lumping all the sum-PIPs local to each DA calculation into one global sum-PIP which represents the entire IP calculation. However there is only one free serial input, and as the addend word invariably extends up into the 'fractional' part of the addend, there is no obvious way to exploit the free carry-input. Thus local sum-PIPs will be loaded as before.

When cascading the flush IP computer, no free inputs may be exploited as the format of the output product is different from the input addend. That is not to say that the flush IP computer cannot be cascaded for computation of inner-products - on the contrary, it is best suited to such computation, if target data rates permit the computational inefficiencies resulting from the guard-bit requirement. Accumulation must be carried out on dedicated adders in this case.

Word growth can occur in long IP calculations - this may be accommodated in higher-order bit-serial accumulators [123]. Several architectural possibilities exist for inner-product calculations [86,85], which have direct relevance to serial-data realisations. Often the choice is decided by the allowable transform latency - two FIR filtering case studies in *FIRST* (matched filtering [166] and adaptive filtering [167]) yielded markedly different multiplexed architectural solutions. The former architecture was a pipelined, forward flowing cascade, whilst the latter, minimal-latency architecture fanned-in sums through a binary addition tree [123].

Formal mechanisms for *specifying* cascaded inner-product architectures are presented in [7]. Systems designers must take many factors into account when making these specifications - these include tolerable transform latency, signal statistics, signal bandwidth (hence multiplexing scheme), accuracy requirements, etc. However a formal mechanism is also required for *implementing* cascaded inner-product architectures. We present two such mechanisms, one which exploits the free input and one which does not.

Free accumulation

We envisage a cascade of IP computers, with the double-precision product output from each connecting to the addend input of the subsequent. The low-order m bits of this word are accommodated in bit-parallel form, and the remaining $qn - m$ bits in multi-precision serial form. One extra SIPO is required to convert the low-order bits into parallel form for loading at the free parallel input. The operation of this SIPO is identical to the SIPO used for coefficient loading, except that in this case no holding register is required. Word-growth beyond the range of double-precision may be accommodated in higher-order serial-data adders.

Adder-based accumulation

Here we make no use of the free input - double-precision outputs are fed directly to a multi-precision accumulator. The action of this accumulator beyond the double-precision range is identical to its action in the free-accumulation structure. The flush IP computer finds application in this cascading environment.

6.3. Incremental computation of squares and sums of squares

Here we present a final complexity-reduction technique, based on the incremental multipliers encountered in Chapter 3. Although it represents original work by the author [108], there is no immediate plan to adopt these architectures. The material serves mostly to illustrate another example of symmetrical computation.

This incremental algorithm is proposed for computation of squares or sums of squares, suitable for both MSB-first and LSB-first bit-sequential operation. The symmetry of the bit-product matrix associated with a squaring computation is key to

the operation of the algorithm. By permitting the elimination of redundant computation, existing hardware modules may be either reduced in size, or assigned to the evaluation of a second squaring computation. The corresponding hardware architectures may be derived from a simple conversion of existing incremental scalar multipliers.

Chapter 3 introduced the incremental multiplication technique, and concluded that it was area-expensive and of limited value. However incremental multipliers have one property which is not shared by the more conventional S/P and Lyon multipliers - they are 'on-line' with respect to both input operands. At each iteration, one bit from both operands can be input, and one product bit output. The case for their adoption is strengthened in applications which require 'data-data' type multiplications, instead of the more common 'coefficient-data' scheme. Squaring is one such application.

6.3.1. Incremental squaring

Figure 6.7 shows the matrix of bit-product formation *values* in the multiplication of 2 4-bit binary integers $a_0a_1a_2a_3$ and $b_0b_1b_2b_3$, where a_0 and b_0 are MSB, as dictated by eqn. 2.1. In the case of squaring (where $a_i = b_i$), this matrix is symmetrical. Figures 6.8(a-c) show the matrices of bit-product formation *times* of several multipliers. Only in the incremental case is this matrix also symmetrical.

	a_3	a_2	a_1	a_0
b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3
b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2
b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1
b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0

Figure 6.7: matrix of bit-product values (4 × 4-bit example)

This dual-symmetry allows a reduction of hardware. When the values *and* times of off-diagonal bit-products are identical to those of their reflection in the

	a_3	a_2	a_1	a_0
b_3	1	1	1	1
b_2	2	2	2	2
b_1	3	3	3	3
b_0	4	4	4	4

Figure 6.8(a): matrix of bit-product formation-times, S/P multiplier

	a_3	a_2	a_1	a_0
b_3	1	3	5	7
b_2	2	4	6	8
b_1	3	5	7	9
b_0	4	6	8	10

Figure 6.8(b): matrix of bit-product formation-times, Lyon multiplier

	a_3	a_2	a_1	a_0
b_3	1	2	3	4
b_2	2	2	3	4
b_1	3	3	3	4
b_0	4	4	4	4

Figure 6.8(c): matrix of bit-product formation-times, incremental multiplier

main diagonal of the matrix, their individual evaluation and accumulation is unnecessary. They may instead be merged pairwise into one evaluation and accumulation (at twice the weight, i.e. delayed by one clock cycle). On-diagonal elements must be treated as before, which slightly complicates matters of operand

distribution.

Figure 6.9(a) is an expanded version of Figure 6.8(c), showing the hardware activity in the 4 processors of the incremental multiplier. Each processor is capable of accumulating two bit-products, as shown. The 'z' symbol signifies processor inactivity, and the arrow symbol signifies passing of sums and assimilation of carries (the arrow is implied where accumulation of bit-products is performed). Figure 6.9(b) shows the same activity in the case of squaring, and Figure 6.9(c) the proposed modification to the algorithm. As each processor has only one bit-product to accumulate in this case, the (5,3) adders may be replaced by (3,2) adders.

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	$a_3b_3 + 0$	z	z	z
2	$a_3b_2 + a_2b_3$	$a_2b_2 + 0$	z	z
3	$a_3b_1 + a_1b_3$	$a_2b_1 + a_1b_2$	$a_1b_1 + 0$	z
4	$a_3b_0 + a_0b_3$	$a_2b_0 + a_0b_2$	$a_1b_0 + a_0b_1$	$a_0b_0 + 0$
5	↙	↙	↙	z
6	↙	↙	↙	z
7	↙	↙	z	z
8	↙	z	z	z

Figure 6.9(a): incremental multiplication

Note that the carry signal cannot be non-zero as a result of an on-diagonal calculation, and that in general no more than $n - 1$ processors are active at any time. Although this promises a reduction in the length of the array, the additional hardware cost for control and data routing would in all probability outweigh any saving.

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	$a_3a_3 + 0$	z	z	z
2	$a_3a_2 + a_2a_3$	$a_2a_2 + 0$	z	z
3	$a_3a_1 + a_1a_3$	$a_2a_1 + a_1a_2$	$a_1a_1 + 0$	z
4	$a_3a_0 + a_0a_3$	$a_2a_0 + a_0a_2$	$a_1a_0 + a_0a_1$	$a_0a_0 + 0$
5	↙	↙	↙	z
6	↙	↙	↙	z
7	↙	↙	z	z
8	↙	z	z	z

Figure 6.9(b): incremental squaring as a multiplication on equal data words

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	a_3a_3	z	z	z
2	z	a_2a_2	z	z
3	a_3a_2	z	a_1a_1	z
4	a_3a_1	a_2a_1	z	a_0a_0
5	a_3a_0	a_2a_0	a_1a_0	z
6	↙	↙	↙	z
7	↙	↙	z	z
8	↙	z	z	z

Figure 6.9(c): incremental squaring, exploiting symmetry of bit-product matrix

6.3.2. Incremental sums-of-squares computation

A second modification to the incremental multiplier allows computation of sums of squares. Figure 6.9(d) shows the introduction of a second 4-bit input operand $b_0b_1b_2b_3$. We revert to use of (5,3) adders, and compute the sum of squares as shown.

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	$a_3a_3 + b_3b_3$	z	z	z
2	↙	$a_2a_2 + b_2b_2$	z	z
3	$a_3a_2 + b_3b_2$	↙	$a_1a_1 + b_1b_1$	z
4	$a_3a_1 + b_3b_1$	$a_2a_1 + b_2b_1$	↙	$a_0a_0 + b_0b_0$
5	$a_3a_0 + b_3b_0$	$a_2a_0 + b_2b_0$	$a_1a_0 + b_1b_0$	↙
6	↙	↙	↙	z
7	↙	↙	z	z
8	↙	z	z	z

Figure 6.9(d): incremental sum-of-squaring

6.3.3. Bidirectional incremental multiplication

In all the above cases, the last part of the computational cycle is spent assimilating carries and clocking out results. The further a processor is from the output end of the array, the more time it spends idling. Scanlon & Fuchs made this observation [77], and proposed a further modification to the incremental multiplier architecture. By introducing bidirectional shifting to the array, they increased hardware usage to the point where a $2n$ -bit product could be produced every $n + 1$ clock cycles.

This structure is equally amenable to the proposed conversion process. Figure 6.10(a) shows the operation of the bidirectional multiplier, and Figures 6.10(b,c) its equivalent modification for computation of squares and sums of squares respectively.

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	$a_0b_0 + 0$	z	z	z
2	$a_0b_1 + a_1b_0$	$a_1b_1 + 0$	z	z
3	$a_0b_2 + a_2b_0$	$a_1b_2 + a_2b_1$	$a_2b_2 + 0$	z
4	$a_0b_3 + a_3b_0$	$a_1b_3 + a_3b_1$	$a_2b_3 + a_3b_2$	$a_3b_3 + 0$
5	↙	↙	↙	z
6	↙	↙	↙	$c_0d_0 + 0$
7	↙	↙	$c_1d_1 + 0$	$c_0d_1 + c_1d_0$
8	↙	$c_2d_2 + 0$	$c_1d_2 + c_2d_1$	$c_0d_2 + c_2d_0$
9	$c_3d_3 + 0$	$c_2d_3 + c_3d_2$	$c_1d_3 + c_3d_1$	$c_0d_3 + c_3d_0$
10	z	↘	↘	↘
11		↘	↘	↘
12			↘	↘
13				↘

Figure 6.10(a): bidirectional incremental multiplication

These structures as presented operate on unsigned integers. Two's complement operation is attainable by including hardware for either MSB-treatment [104, 103] or recoding [102].

6.3.4. Application to the on-line algorithms

Incremental techniques have been proposed as a method of implementing advanced arithmetic operations, such as division [10] and square-rooting [168, 169]. These algorithms exploit the fast, carry-free addition made possible by the use of redundant-coded data, processing operands and producing results on a digit-by-digit basis, most-significant digit first.

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	a_3a_3	z	z	z
2	z	a_2a_2	z	z
3	a_3a_2	z	a_1a_1	z
4	a_3a_1	a_2a_1	z	a_0a_0
5	a_3a_0	a_2a_0	a_1a_0	z
6	↙	↙	↙	b_3b_3
7	↙	↙	b_2b_2	z
8	↙	b_1b_1	z	b_3b_2
9	b_0b_0	z	b_2b_1	b_3b_1
10	z	b_1b_0	b_2b_0	b_3b_0
11		↘	↘	↘
12			↘	↘
13				↘

Figure 6.10(b): bidirectional incremental squaring

In the simplest sense, such algorithms rely on producing an output estimate E , and maintaining an internal ‘scaled partial remainder’. This remainder is the result of an incremental multiply-subtract ‘check computation’ involving the input operand(s) and the output estimate. If the subtracter is prevented from overflowing, the output estimate converges towards the correct result. Overflow prevention is problematic where redundant coded data are concerned – a complicated digit-selection process must be performed during each iteration to prevent overflow.

A Multiplication ($E = A \times B$) is characterised by the expression:

$$E - A \times B \rightarrow 0$$

time	Proc. 1	Proc. 2	Proc. 3	Proc. 4
1	$a_3a_3 + b_3b_3$	z	z	z
2	↙	$a_2a_2 + b_2b_2$	z	z
3	$a_3a_2 + b_3b_2$	↙	$a_1a_1 + b_1b_1$	z
4	$a_3a_1 + b_3b_1$	$a_2a_1 + b_2b_1$	↙	$a_0a_0 + b_0b_0$
5	$a_3a_0 + b_3b_0$	$a_2a_0 + b_2b_0$	$a_1a_0 + b_1b_0$	↙
6	↙	↙	↙	$c_3c_3 + d_3d_3$
7	↙	↙	$c_2c_2 + d_2d_2$	↘
8	↙	$c_1c_1 + d_1d_1$	↘	$c_3c_2 + d_3d_2$
9	$c_0c_0 + d_0d_0$	↘	$c_2c_1 + d_2d_1$	$c_3c_1 + d_3d_1$
10	↘	$c_1c_0 + d_1d_0$	$c_2c_0 + d_2d_0$	$c_3c_0 + d_3d_0$
11		↘	↘	↘
12			↘	↘
13				↘

Figure 6.10(c): bidirectional incremental sum-of-squaring

i.e. the product and the estimate are finally one and the same. No feedback of E is employed.

B Division ($E = A / B$) is characterised by the expression:

$$A - E \times B \rightarrow 0$$

i.e. E is internally multiplied with B as check computation.

C Square-rooting ($E = \sqrt{A}$) is characterised by the expression:

$$A - E^2 \rightarrow 0$$

i.e. E is internally squared as check computation.

The conversion from multiplier to squarer adds little to the work already published by Ercegovic and colleagues [168, 169]. However the conversion from multiplier to sum-of-squarer gives rise to the possibility of computing functions such as root-of-sum-of-squares, e.g. magnitude of a plane vector or complex number, in a single on-line structure. This points to a new on-line algorithm:

D Magnitude extraction, where $E = \sqrt{A^2 + B^2}$, characterised by the expression:

$$A^2 + B^2 - E^2 \rightarrow 0$$

Here efficiencies of squaring and sum-of-squaring may be exploited to effect a considerable reduction in hardware cost, leading to a structure only slightly larger than the single-operation processors.

6.4. Final comments

This chapter has introduced some novel serial-data computational elements. The simpler distributed arithmetic elements have been proved in simulation, first at high-level in purpose-built *C* language [170] codings, then in transistor-level *RNL* [171] descriptions (see Appendix C). Incorporation of these elements into a function library such as that of *FIRST* might well increase its *functional* flexibility. *Operational* flexibility, i.e. flexibility of throughput and/or dynamic range exhibited by computational elements, is the subject of the next chapter.

Chapter 7

Special serial-data techniques for throughput enhancement

Although bit-serial architectures exhibit excellent properties for VLSI implementation of fixed-function DSP machines, two criticisms which are often levelled concern the high cost of operand storage, whether in memories, registers or pipelining latches, and the low processing throughput in relation to bit-parallel functional equivalents. For instance a bit-serial multiplier must firstly load then store the coefficient word in a static register for the duration of a product calculation. Storage cost, internal latching, computational hardware and computational time are all $O(n)$. A pipelined parallel multiplier in contrast uses $O(n^2)$ internal latching and computational hardware, but still only $O(n)$ operand storage, while computing in $O(1)$ time. Thus operand storage is less dominant in the parallel case than in the serial.

In this chapter we describe techniques to increase the throughput of bit-serial computational networks, while retaining the many advantages associated with this architectural approach. In essence these techniques rely on multi-wire representations of serial data - a step towards bit-parallelism. As the cost of data storage associated with bit-serial architectures is not increased by these techniques, it has a favourable effect on overall area-time product. Procedures and interfacing conventions are outlined, which allow the synthesis of serial-data library elements using these techniques. By offering a range of performance/area options, we may bring the operational flexibility defined earlier. The proposed data formats are summarised in Figure 7.1.

7.1. Twin-pipe

Compatible with any double-phase clocking technique is a novel serial-data architectural technique which achieves double the throughput of conventional bit-serial architectures, without suffering a proportional increase in circuit area [172,162][†]. This technique features a two-wire representation of serial data.

[†] published work by the author.

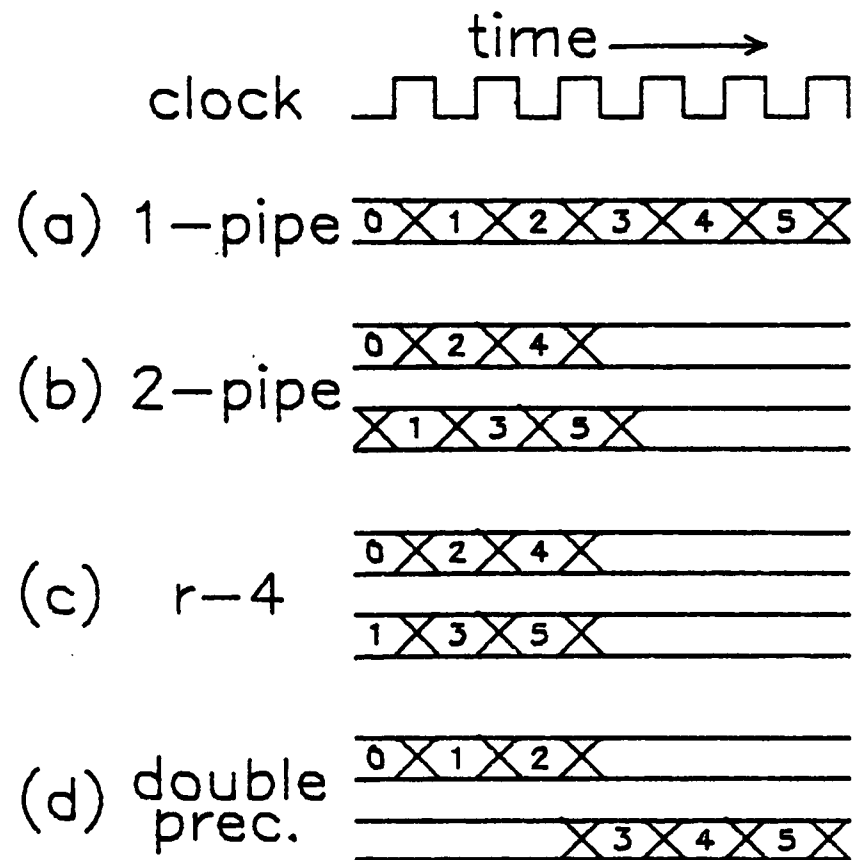


Figure 7.1: serial-data formats

Figures 7.1(a) and 7.1(b) illustrate single-pipe and twin-pipe data formats. Twin-pipe operation requires that the serial-data wordlength is an even number. Even numbered data bits are processed on different clock phases and in different circuits from odd-numbered bits, and logical function is included only where necessary.

As a full-latch is equivalent to two half-latches, storage costs remain constant. Thus area-time product is improved by this technique. We propose to implement twin-pipe architectures using the single-phase clocking technique described in Chapter 2 [43].

7.1.1. Twin-pipe building blocks

In order to map the serial-data cell library into twin-pipe, we repeat the process of defining registers and computational 'atoms'. Once these object are defined, the construction of higher-level functional elements such as multipliers may proceed in the same manner as before. We begin with registers.

Registers

In Chapter 2 we encountered the single-phase clocking technique, and the π and μ latches. The π and μ latches have direct physical significance - the former latches data when the clock is high, the latter when it is low. In the following discussion, the data bit-types *even* and *odd* are used, and a computational pipe is associated with each. Throughout this chapter LSB is associated with the even pipe, and MSB with the odd. Even- and odd-pipe latches may take either π or μ form, as long as they are different.

A simplistic view of the mapping of single-pipe architectures into twin-pipe indicates that the area cost of combinatorial logic should double under such a mapping. This view is somewhat pessimistic. In most cases the asymmetry inherent in 2C computation demands that different logic functions operate on even and odd data bits. While the single-pipe realisation must merge the two functions, resulting in one large logic block, the twin-pipe equivalent may separate the functions, yielding two different logic blocks each of which may exhibit lower fan-in and combinatorial complexity than the single-pipe block.

Figure 7.2 shows the twin-pipe equivalents of the 4 basic registers. As the loading action is only required in one pipe, no increase in circuit area is apparent (we neglect the area cost of wires). The separation of logic function into even and odd-pipe components in this manner may be exploited in many cases to reduce the complexity of twin-pipe hardware.

Multiplexer

As data-selection imposes the same logic function on all data bits, the twin-pipe multiplexer is one element which cannot exploit computational asymmetry. Thus the twin-pipe multiplexer (Figure 7.3) consists of two multiplexing blocks and two half-latches. Odd-pipe control may be generated locally from even-pipe control (as shown), or brought in on a second control wire.

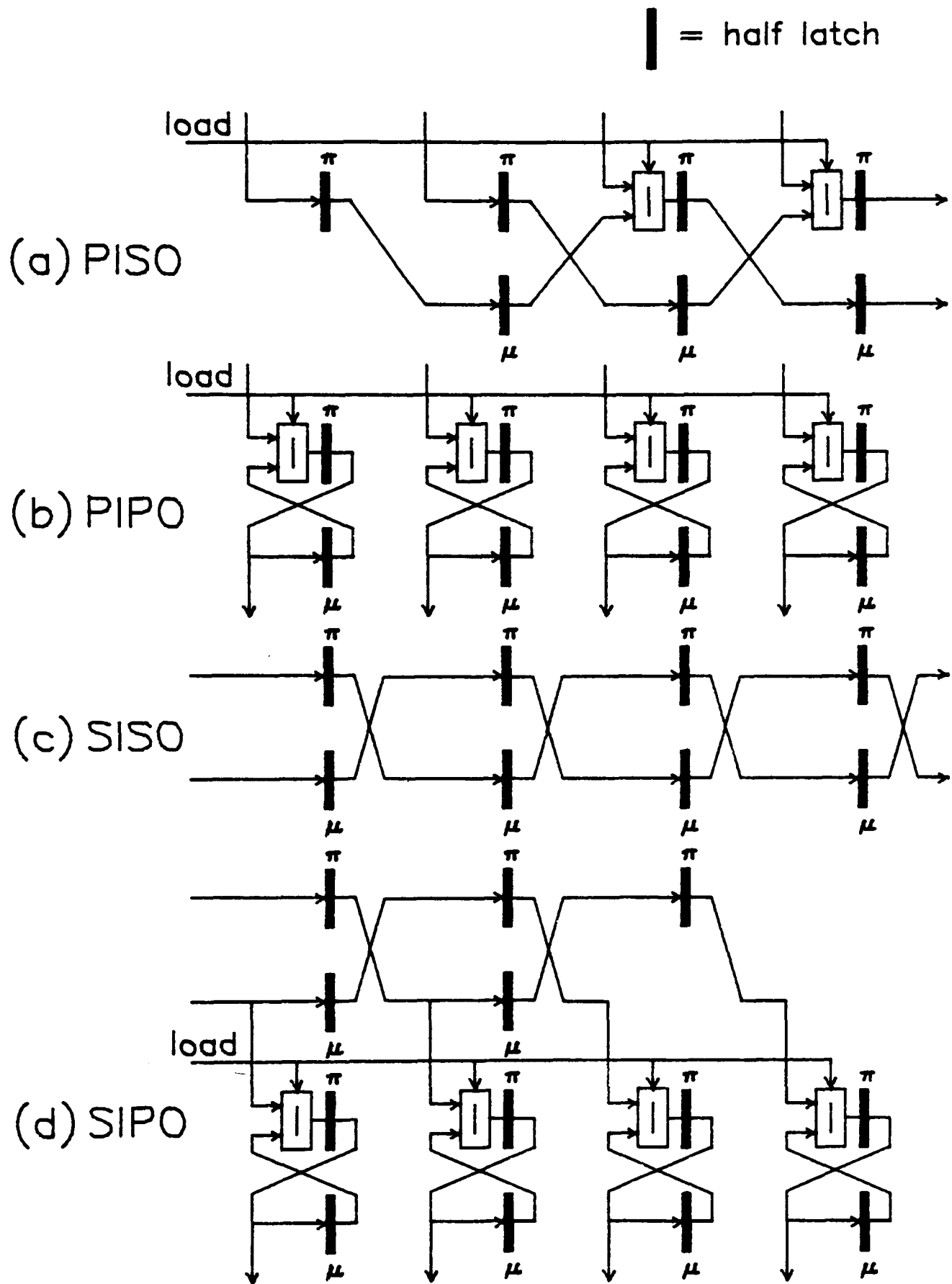


Figure 7.2: twin-pipe forms of the 4 basic registers

Arithmetic Shifters

Four cases of arithmetic shifting exist in the twin-pipe case, as left- and right-shifting may be performed over even and odd bit-distances, requiring different

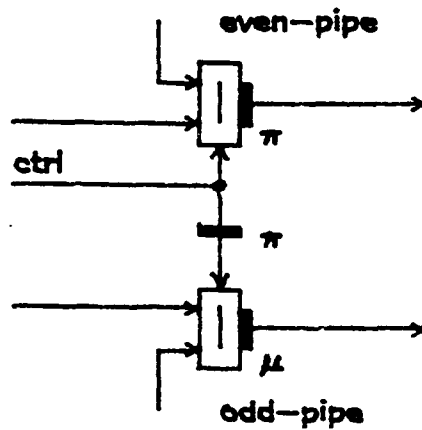


Figure 7.3: twin-pipe multiplexer

hardware structures in each case. Odd bit-distances require a crossover of pipes, while even distances do not. In the case of a 1-bit shift, no multiplexing logic is required on the odd-indexed pipe (accordingly, no odd-pipe control signal is required). As will shortly be demonstrated, this has important consequences in twin-pipe multiplier design. Figure 7.4 shows the 4 twin-pipe shifters.

Adder

The twin-pipe adder is another example of a structure which exploits computational asymmetry. The function of the adder is to produce sum and carry functions from all operand bits, with a left-shift at the carry-input (active only at LSB-time, on the even pipe). Accordingly, the shifter is included only on the even pipe - the odd-pipe logic block merely performs the raw sum and carry functions. Figure 7.5 shows the twin-pipe adder. Note the crossed over carry configuration, reminiscent of the 1-bit PIPO.

CSAS computer

The CSAS computer is a linear array of bit-serial adders, with left-shifters in the sum path as well as in the carry path. Thus we see a crossover of pipes in the sum path. Once again, left-shifting (LSB treatment) is an even-pipe function. Meanwhile the odd pipe must handle MSB-inversion. Here the disjoint functions for LSB- and MSB-treatment may be separated, resulting in logic blocks with 4 inputs in both pipes (the equivalent single-pipe block has 5). If the bit-product formation function (necessary in both pipes) is included in the logic blocks, these

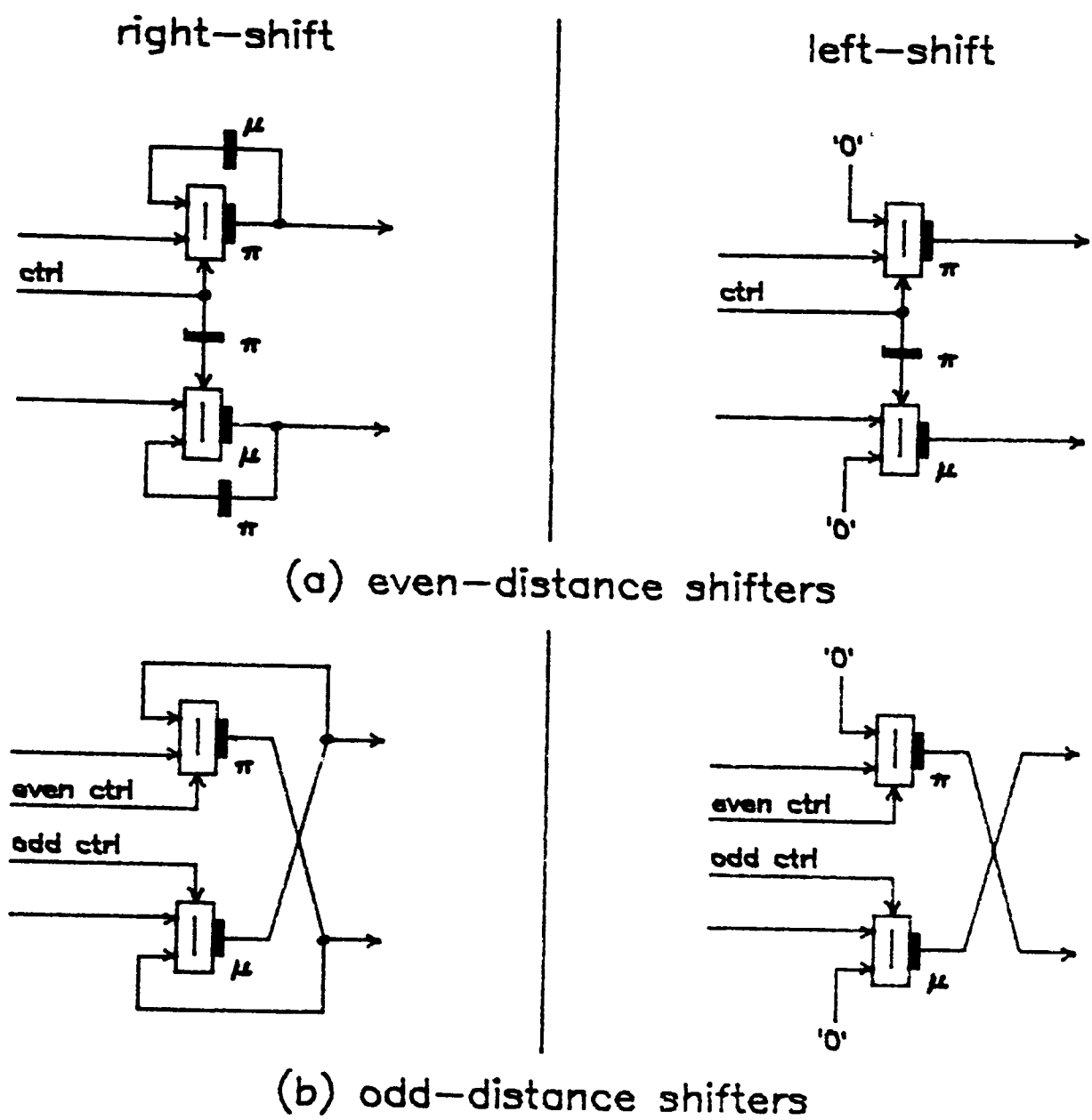


Figure 7.4: twin-pipe shifters

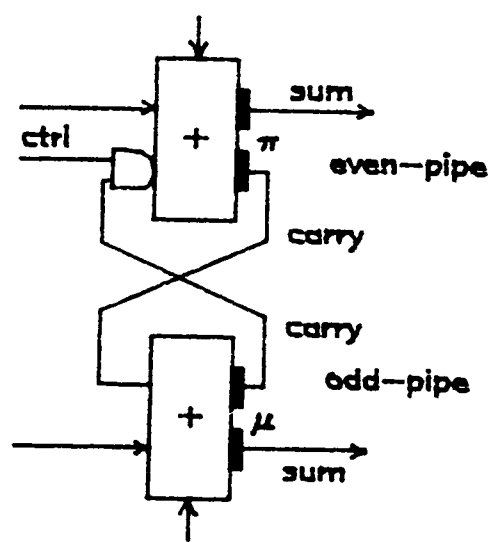


Figure 7.5: twin-pipe adder

figures rise to 5 and 6 respectively. Figure 7.6(a) shows the single-pipe CSAS computer, and Figure 7.6(b) the equivalent twin-pipe CSAS computer.

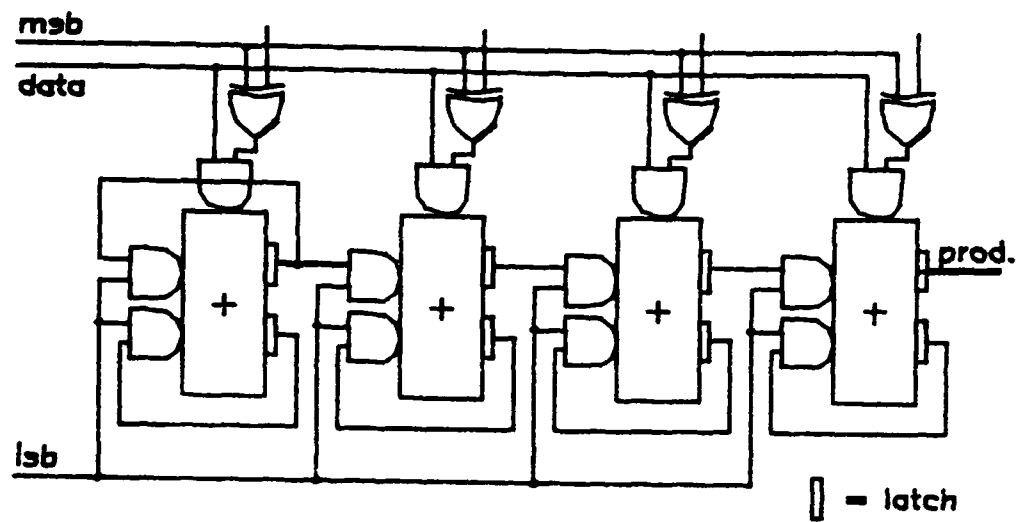


Figure 7.6(a): single-pipe CSAS computer

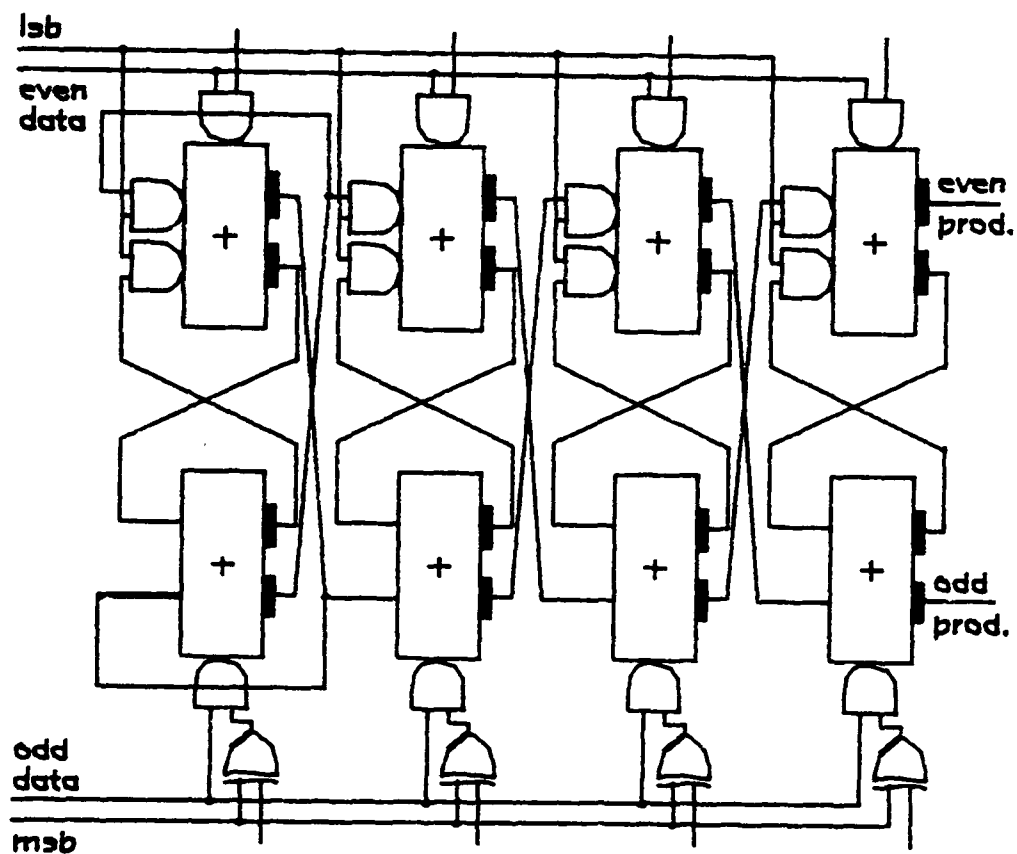


Figure 7.6(b): twin-pipe CSAS computer

7.1.2. Architectural implications of twin-pipe techniques

Together with the single-phase circuit techniques outlined earlier, twin-pipe represents an efficient method of doubling the throughput of bit-serial architectures. Storage costs, in both registers and latches, are no greater than those of conventional single-pipe realisations, while the costs of logic function are usually considerably less than double. The effect on area-time product is beneficial.

In most cases, a twin-pipe solution offers an area-efficient solution to a computational problem. However that is not to say that single-pipe techniques should be discarded - many situations will arise which do not require the throughput of twin-pipe. In addition, a mixture of the two techniques may prove beneficial.

7.2. Radix-4

A second throughput enhancement technique [173,162][†] relies on higher-level architectural advances, involving techniques emanating more from the field of computer arithmetic than from circuit engineering. This technique is compatible with twin-pipe, and if used in concert with twin-pipe results in a factor of 4 speed increase. Here we use single-pipe structures to illustrate radix-4 computation.

In radix-4 structures data are processed and transmitted in contiguous pairs, even bits on one wire and odd bits on the other. However the pairs are transmitted and processed concurrently, not on opposite clock phases as is the case in twin-pipe. Computational elements have *two* logical inputs per input operand, and an n -bit (i.e. $n/2$ -digit) data word is processed in $n/2$ clock cycles. The pairwise linking of data bits in this manner implies that the basic 'bundle' of information is no longer a bit - rather it is a radix-4 *digit*. By performing all computation and communication in radix-4, we are once again able to decrease the unfavourable ratio of storage to logic, doubling throughput for no increase in operand storage. However, unlike twin-pipe, the cost of data latching doubles in radix-4.

Figure 7.1(c) illustrates the radix-4 data format. Data are transmitted LSB-first. It should be noted that numbers are still represented in standard, two's complement form, and that only the distribution of bits in space and time is altered. We use the term 'radix-4' mostly to convey the concept of pairwise transmission and processing of data bits. Radix-4 data are transmitted LS-*digit*-first, and the data wordlength *in digits* is fixed and constant in each system. Only the stand-alone adder is in reality a radix-4 structure, as use of the modified-Booth (M-Booth) multiplier recoding technique [59] allows a radix-2 structure to perform multiplication, resulting in further area savings.

[†] published work by the author.

7.2.1. Radix-4 building blocks

We repeat the mapping of the serial-data atoms, this time into radix-4, starting again with the registers. Unfortunately, as will be shown, separation of logic function into even and odd-pipe components cannot be exploited to reduce the complexity of radix-4 hardware. This is because even and odd pipe components are processed concurrently in the same logic block, and not sequentially in different logic blocks as is the case in twin-pipe schemes.

Registers

Figure 7.7(a-d) shows the radix-4 equivalents of the 4 basic registers. Here the loading action is distributed between pipes, and once again no increase in circuit area is apparent, neglecting the area cost of wires.

Multiplexer

Like the twin-pipe multiplexer, the radix-4 multiplexer consists of two multiplexing blocks and two latches (full latches here).

Arithmetic Shifters

Again, radix-4 shifters are similar to twin-pipe shifters, in that hardware realisations differ for even and odd shift distances. An even-distance shift may be performed by a pair of radix-2 shifters, sharing control. However an odd-distance shift requires separate control signals on each multiplexer, and crossover of the output wires for correct subsequent interpretation. Figure 7.8 shows the 4 radix-4 shifters.

Adder

A radix-4 adder [173] may be constructed by cascading two radix-2 adders (included for comparison in Figure 7.9(a)), removing the latching from the low-order carry output (Figure 7.9(b)). Alternatively, the low-order inputs may be used directly in combinatorial odd-index sum and carry generation circuits (Figure 7.9(c)). However neither of these arrangements will match the radix-2 adder for performance - the former because it is cascaded but not pipelined, and the latter

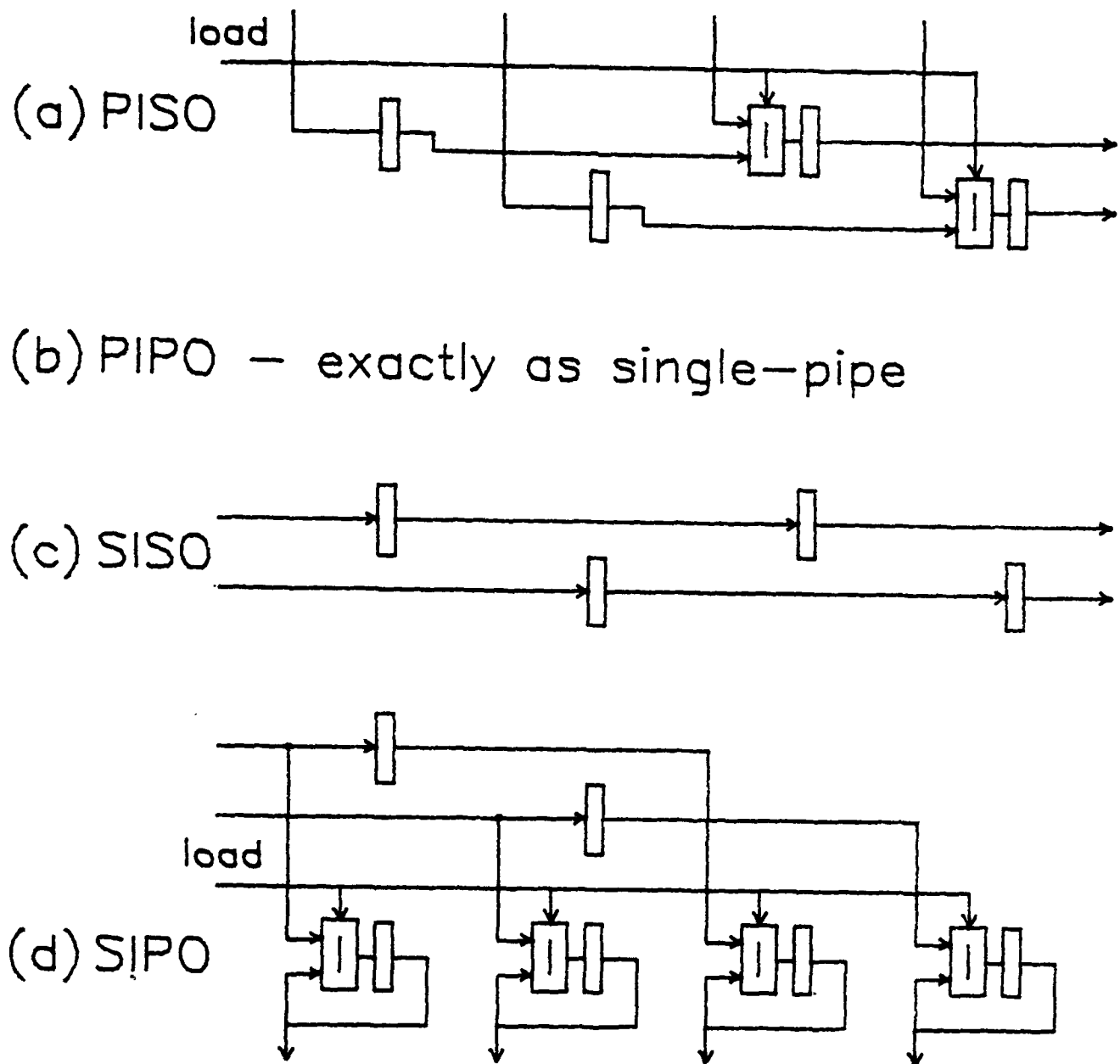


Figure 7.7: radix-4 register types

due to the excessive size (5 inputs), and hence evaluation time, of the odd-index sum and carry logic networks. The more complex architecture of Figure 7.9(d) pipelines the recombination of the carry, limiting the number of inputs to an acceptable 4 in each logic block.

The three radix-4 adders of Figure 7.9(b-d) offer different performance-area options. We shall proceed with the largest and fastest, Figure 7.9(d). We note that the radix-4 adder is somewhat more than twice the size of a radix-2 adder. This architecture finds use as a stand-alone adder, where the increased latency resulting from the extra pipelining is of little concern. However adders find frequent use as constituent parts of the multipliers which tend to dominate DSP applications. Here adders are used in linear carry-save arrays, and their size and latency directly affects

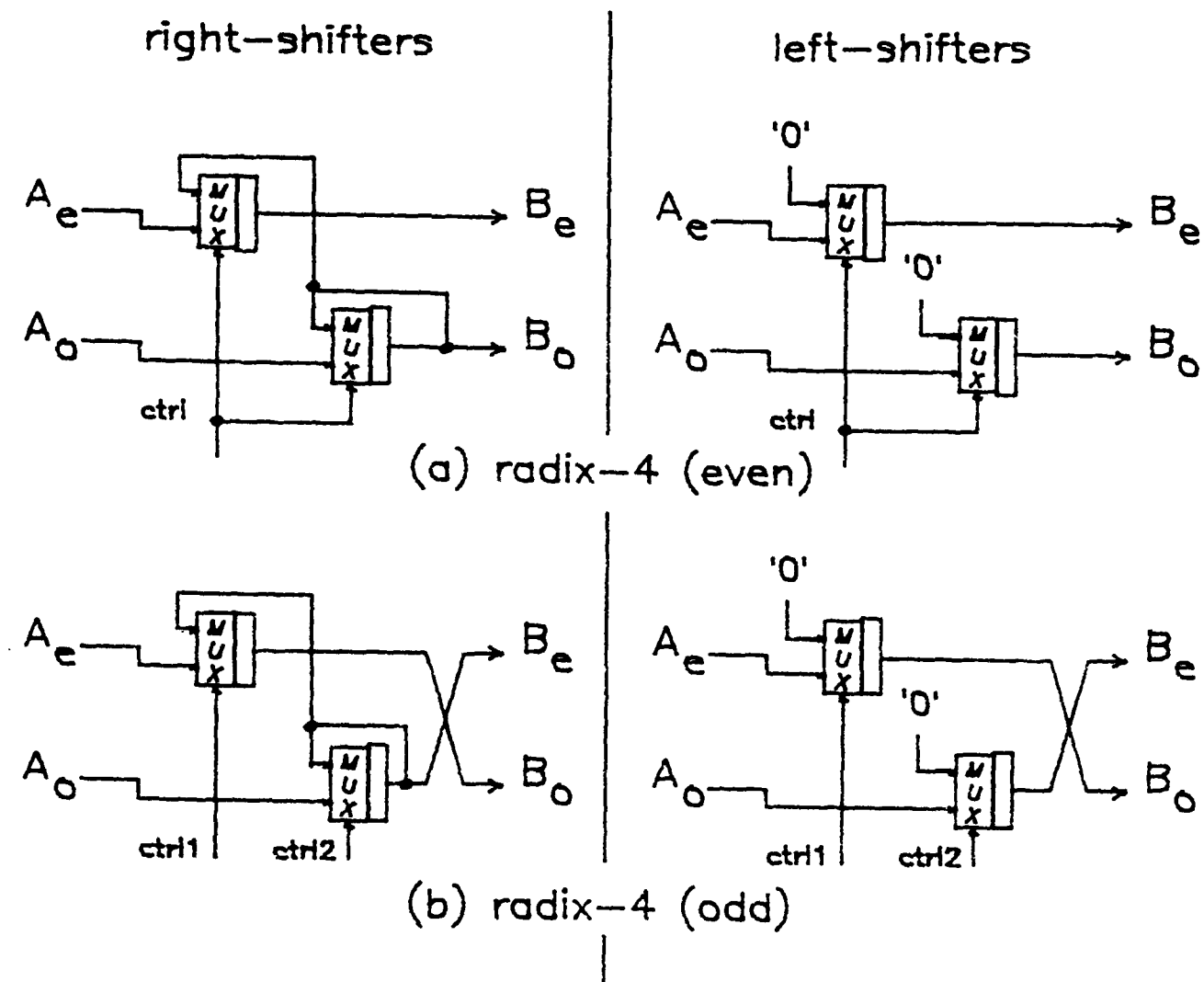


Figure 7.8: radix-4 shifters

the size and latency of the multiplier. Fortunately, as is about to be revealed, radix-4 multipliers may be constructed from radix-2 adders, obviating this problem.

CSAS computer

The CSAS computer could be built from a linear array of bit-serial adders, as before. However we choose not to implement the radix-4 CSAS computer, turning instead to a well-known complexity reduction technique, the modified Booth recoding algorithm [58, 59]. Radix-2 bit-multiplication has the beneficial effect of *closure*, i.e. the product of two bits may itself be expressed as a bit. As radix-4 digit-multiplication is not closed, a radix-4 multiplier would require a twin array of carry-save cells, were it not for M-Booth recoding.

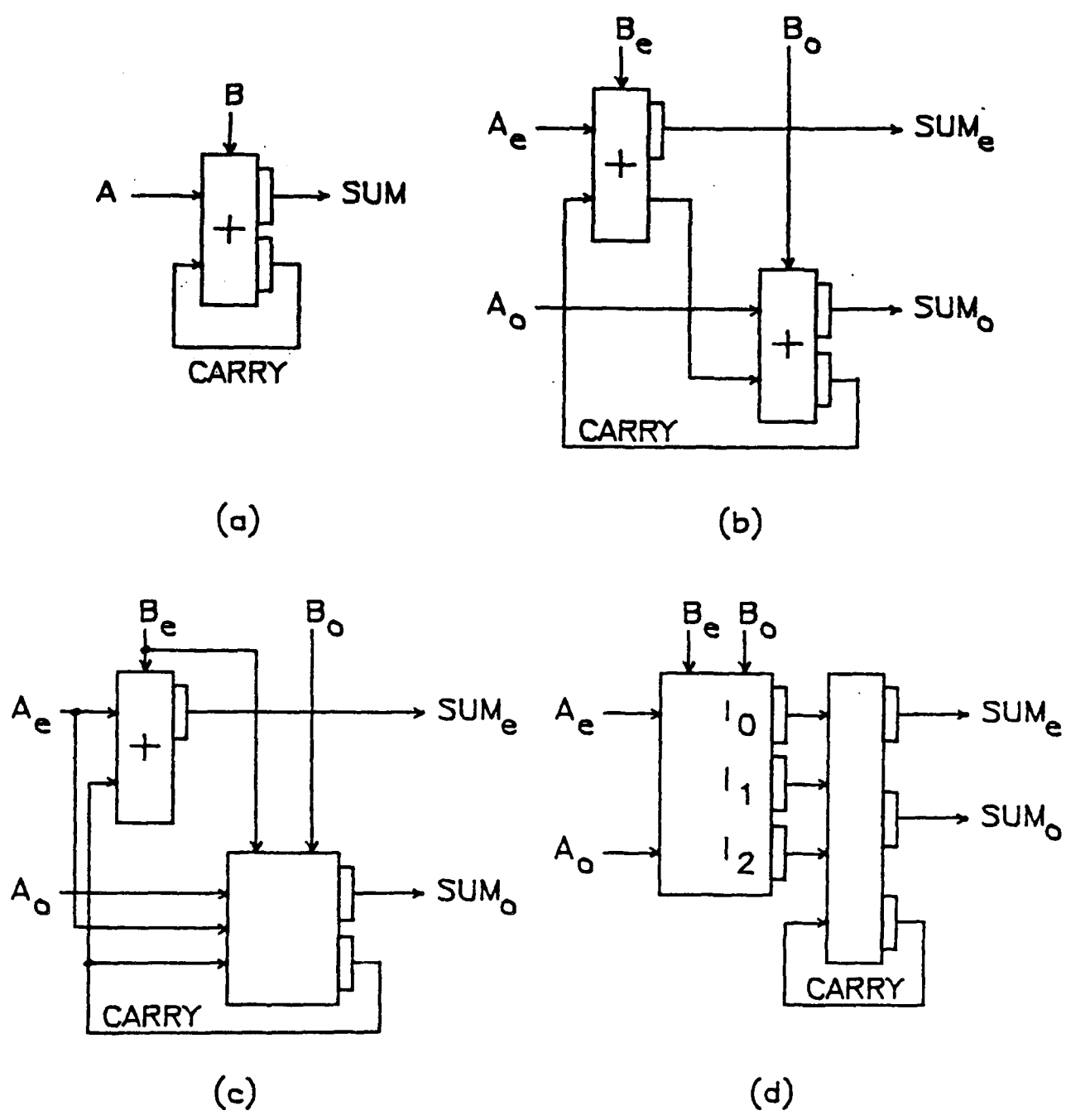


Figure 7.9: radix-4 adders

The radix-4 modified-Booth multiplier

The M-Booth algorithm [59] uses a 5-level recoding scheme (Table 7.1) to effect savings in area or time. The former scheme relies on recoding coefficient bits, thereby halving the number of cells in the multiplier array [44, 7]. The latter processes data bits in concurrent pairs, thereby performing multiplication in half the time [69]. This relates closely to our radix-4 computational format.

Data are recoded in overlapping triplets, each pair sharing with the MS member of the next-MS pair (the LS pair uses logical 0 as third input). As recoded data are in the form 0, ± 1 or ± 2 , they form a radix-4 PP directly from a radix-2

coefficient by gating and/or shifting.

Table 7.1: Modified Booth recoding			
i2	i1	i0	output
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

As the quantity zero may take either positive or negative sign, data bit *i2* may be used directly as a sign-bit. We generate the functions *zero* and *shift* and broadcast these functions along the multiplier array, along with data bit *i2* (functions *z*, *s* and *m* in Figure 7.10). To simplify the function *shift*, we generate the ‘zero PP’ in shifted form. As the purpose of *i2* is to invert the selected PP, the other functions must precede it for correct PP generation.

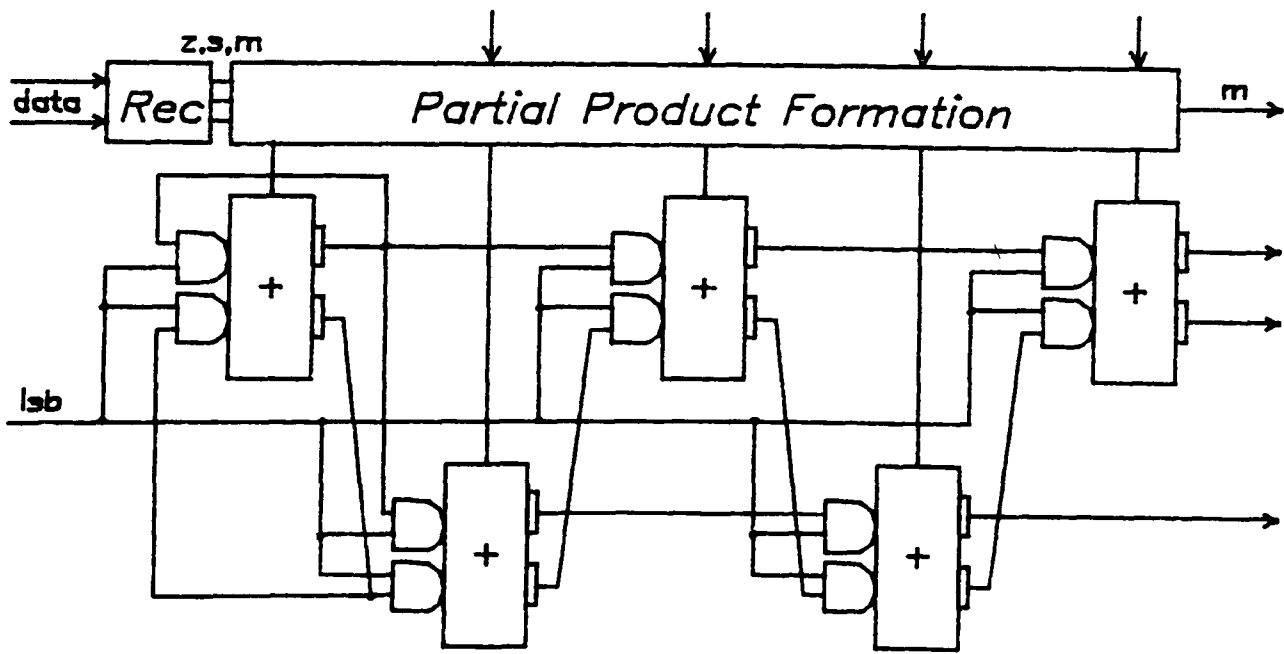


Figure 7.10: radix-4 CSAS computer

Internal operation of the radix-4 CSAS computer (Figure 7.10) is similar to that of [69]. However that radix-4 multiplier is intended for a datapath (parallel-data) environment, where low order product bits are discarded (including the least-significant carry), and high-order bits are subsequently merged in a carry-propagate adder.

As we saw, the serial-data multiplier must also assimilate residual carries. In the radix-4 case carry residues appear in serial (low-order) and in parallel (high-order) formats. Provision of an extra, double-precision radix-4 adder on the output of the array ensures correct product formation. All inputs of this structure are used. In the low-order adder, one addend is the radix-4 sum output, the other is made up from the serial carry output (the high order bit) and a correction bit for true two's complementing of the negated PPs (the low-order bit). This last bit is simply $i2$ delayed by one clock cycle. A similar extra adder cell is used for true two's complementing in the symmetric coded distributed arithmetic architectures of Chapter 6. Figure 7.11 shows the fractional radix-4 multiplier architecture, including residue adder (gating internal to the CSAS computer is suppressed in Figure 7.11, and the serial addend input is included in contrast to Figure 7.10).

The computational engine of the radix-4 multiplier is, once again, a logic/shift-register arrangement which computes partial sums. However there is an important operational difference between this structure and the radix-2 CSAS computer. The radix-4 (M-Booth) multiplication algorithm achieves its computational speed increase by halving the number of PPs to be shift-accumulated in the product calculation. Each PP carries 4 times the weight of its predecessor, as opposed to 2 in the radix-2 case. Thus the bit-stream entering each full-adder is in a hybrid-radix form, in that it consists of a series of radix-4 digits represented by bits. The sum-bits must accordingly be left-shifted by 2 bits in a radix-4 PISO structure, while the carry-bits are still only left-shifted by one bit.

The topological similarity to the even-distance radix-4 shifter is apparent in the sum-path. However the carry-path (including odd-distance shifter) must account for the hybrid radix addition modules, in which carry signals cannot recirculate. Thus each carry-output connects to the input of the next-MS hybrid adder.

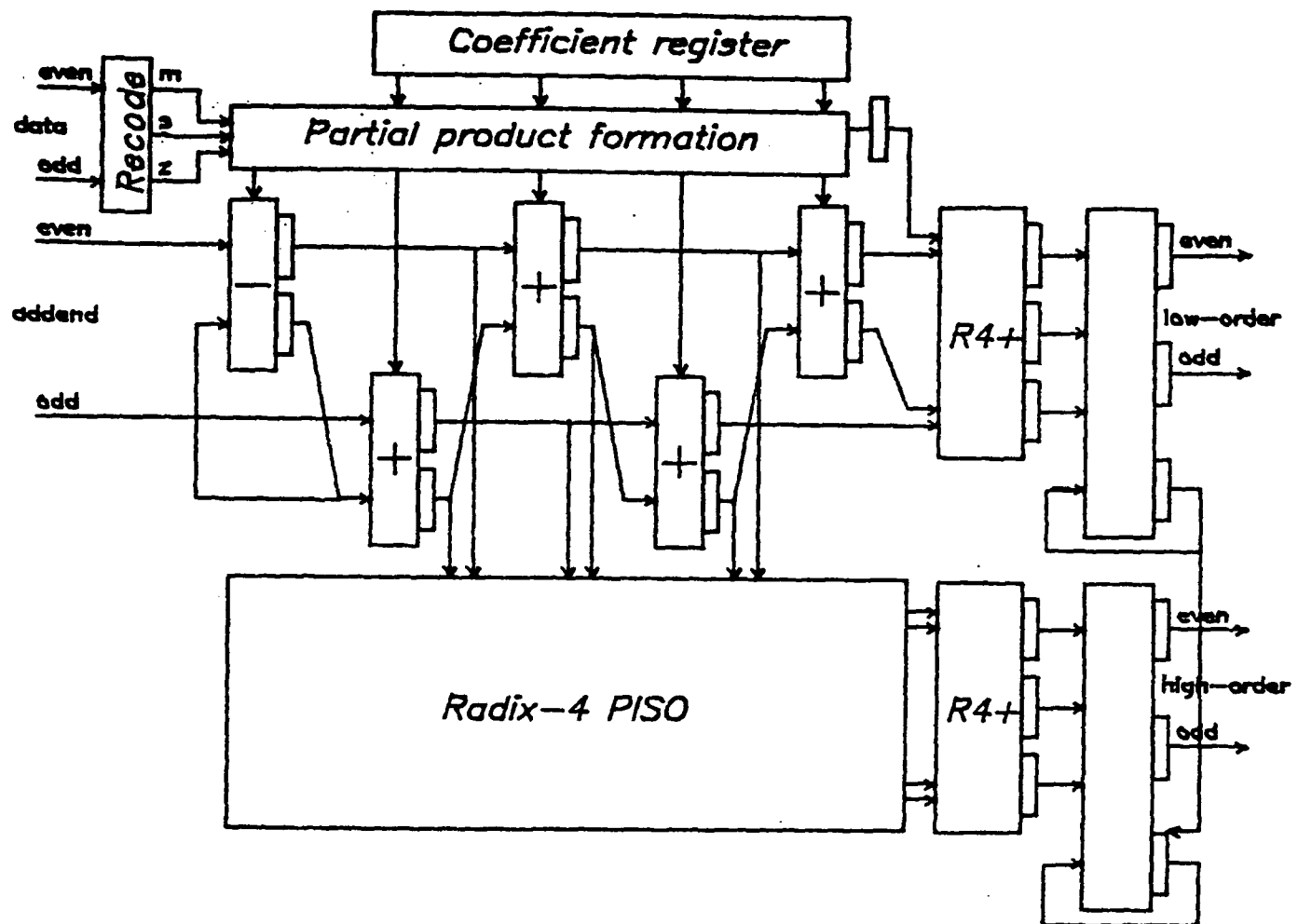


Figure 7.11: radix-4 fractional S/P multiplier

PPs are formed in 3 conceptual stages, one for each of the recoded-data functions specified earlier. Function *shift* drives an array of multiplexers, *zero* an array of AND-gates, and *i2* an array of exclusive-ORs (this has the effect of inverting the PP bits - recall the 2's complement correction bit on the output adder). These operations may be merged or pipelined to arbitrary depth.

Choice of the M-Booth algorithm for multiplicative operations throws up the first incompatibility issue in the architectural developments of this and the preceding chapter. The DA algorithm and the M-Booth algorithm, relying on different techniques for exploitation of coding symmetry, cannot be used together. In a v -point vector computation, there are 2^v possible patterns in radix-2, 4^v in radix-4 and 5^v where M-Booth is used. In radix-4, either storage and selection costs become prohibitively high, or a twin array must be used (negating the original area-savings). Because of this, IP architectures in radix-4 are constructed in the traditional manner, using multipliers and adders.

7.2.2. Architectural implications of radix-4 techniques

We have seen how a radix-4 cell library for DSP applications may be rapidly specified, given knowledge of standard bit-serial techniques. One-to-one mappings exist between radix-2 elements and their radix-4 counterparts. Active logic parts are in many cases twin instances of the radix-2 part - only the adder is significantly different.

Use may be made of the radix-4 recoding properties of the M-Booth multiplier to realise an area-efficient multiplication part, although one unfortunate drawback of the M-Booth algorithm is a fundamental incompatibility with DA architectures for IP computations.

Although it improves area-time product, we propose radix-4 hardware as an alternative to, not a replacement for, radix-2. Radix-4 techniques offer higher throughput at the expense of increased circuit area, but this increase is considerably greater than twin-pipe. Having a wide range of throughputs at their disposal should give systems designers greater flexibility when it comes to matching signal and computational bandwidths in particular applications, complementing the range of architectural techniques detailed in [7], and answering the operational inflexibility charge against serial-data computation.

7.3. Multi-precision

The final throughput enhancement technique allows speed increases by arbitrary factors. This is a direct step towards bit parallelism - a factor of x speed increase causes a factor of x growth in area. This technique may be combined with twin-pipe and radix-4, together or separately.

FIRST uses a multi-precision format [7], as detailed in Figure 7.1(d). Here data are transmitted and processed on multiple wires, in word-staggered fashion. *FIRST* allows multiplexing addition and formatting operations to be performed on multi-precision data, but not multiplication. We show how multiplication may be performed on multi-precision data. Advantage is taken of the free parallel inputs to the CSAS array, and of the fact that coefficients may be treated differently from data.

7.3.1. Multi-precision building blocks

Before this, and for the last time, we specify the basic building blocks in multi-precision.

Registers

Registers are easily realised by multiple instantiations of single-precision registers.

Multiplexer

The multi-precision multiplexer consists of x instantiations of the single-precision multiplexer. For correct operation, the control waveform must be staggered in the same fashion as the data.

Arithmetic Shifters

Multi-precision shifters are similar to their single-precision counterpart, except that only the MS-shifter operates on 2C-coded data (the others process unsigned data). Unsigned right-shifters propagate the rejected bit from their higher-significance neighbour. Only the MS-shifter generates sign repetitions. Similarly left-shifters propagate the rejected bit from their lower-significance neighbour. Only the LS-shifter generates trailing zeroes.

Adder

The multi-precision adder consists of x full-adders. The left-shift of the carry function extends for the full, multi-precision length of the word, thus the trailing zero is generated only in the LS-adder. Elsewhere the carry-loops are loaded with the MS-carry-out from the lower-significance neighbour.

CSAS computer

The multi-precision CSAS computer consists of x linear arrays of bit-serial adders, with left-shifters in the sum path as well as in the carry path. Once again, left-shifting extends over the full length of the word. Multiplication of multi-precision data by a single-precision coefficient may be performed by passing the coefficient from LS-multiplier upwards at LSB-time, at the same time using the free parallel inputs to load the PPS in carry-save form from one multiplier to the next (the PPS in carry-save form consists of the reject bits from the left-shifters throughout the array). Figure 7.12 shows the general case multi-precision CSAS computer.

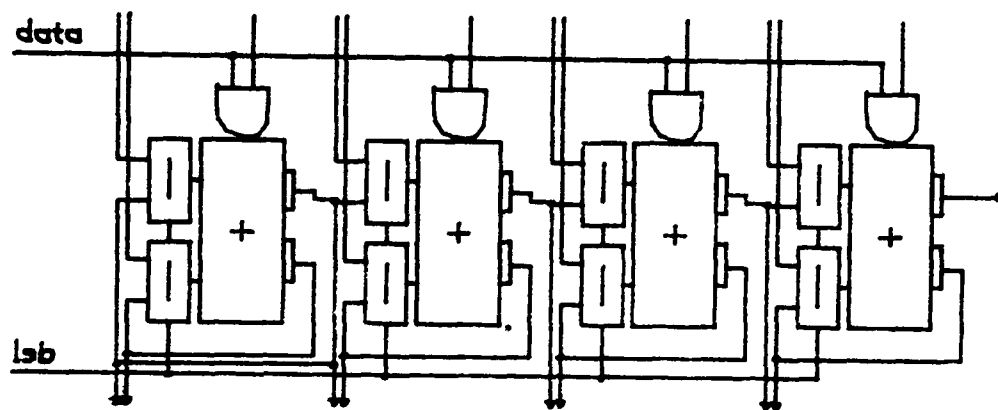


Figure 7.12: multi-precision CSAS computer, general case

Multiplication of multi-precision data by a multi-precision coefficient may be performed simply by concatenating the coefficient into one long parallel word. However this exposes the major weakness of the S/P architecture - the broadcasting of data and control signals. For a large CSAS array, the RC load resulting from the long physical wire and the large number of gates driven becomes unacceptably high. It is necessary to pipeline this broadcast, by including buffers after every m stages of the array. Performance may be maintained in a 2-dimensional array of m -bit CSAS computers, using the free serial input to those CSAS computers which are not in the MS-column. The MS carry from each computer is used by the equivalent in the higher-significance row and column - this signal must be delayed to balance the pipelining delay of the broadcast signals.

7.3.2. Architectural implications of multi-precision techniques

Multi-precision is a sheer brute-force throughput enhancement technique, spanning the spectrum of architectures between bit-serial and bit-parallel. This opens up the possibility of performing large integer multiplications in S/P architectures to address applications such as RSA encryption [174].

In the limit, at a serial wordlength of 1 bit, the multi-precision data, single-precision coefficient serial multiplication architecture is equivalent to a pipelined bit-parallel architecture [53]. In this case, carries are not recirculated and sums are not down-shifted, thus full-adder realisations need not include extra gating. Such structures may exhibit formidable throughputs [57]. Bit-parallel pipelined architectures employ wedging memories on at least data, if not on coefficient as well. The product is re-aligned by passing through another wedging memory. Wedging has the effect of imposing the staggered transmission format in data - in serial-data realisations such a format may be maintained in networks of processors, eliminating the need for wedging other than at system interfaces.

7.4. Interfacing between operational domains

A feature of the bit-serial approach is that interfacing between computational elements may be facilitated by insisting that each element obeys rigid communication protocols [33, 7]. In contrast, the techniques outlined above lead to realisations which feature strikingly different protocols for computation and communication. It may be expedient to mix techniques in a serial-data system, producing 'operational domains' within a system which are incompatible without special interfacing circuitry. There are two distinct reasons for doing this. The first is to realise different throughputs (i.e. word *rates*) within the same system, and the second is to realise different dynamic ranges (i.e. word *sizes*) within the same system. An example of the latter requirement arose during the *FIRST* satellite transmultiplexer case study of Chapter 5.

Even if multi-precision structures are generalised from the double-precision case, three distinct architectural styles have been identified above, on top of the conventional (single-precision, single-pipe) bit-serial architecture. As each-way interfaces are required for all pairwise combinations, a substantial number of

interface structures require to be defined. We seek to demonstrate methods of assembling interfaces, instead of producing an *ad hoc* structure for each. In the following discussion, we refer to *fast* and *slow* structures, and describe methods of interfacing between the two. The example given is the single-to-twin-pipe conversion, and its inverse.

Increasing dynamic range

The structure of Figure 7.13(a) performs slow-to-fast conversion. It consists of a slow SISO whose contents load a fast PISO. The fast PISO contains an extra connection to perform sign-repetition. One slow word is input, and one fast word of doubled size is output every word cycle. The low-order bits of the fast word consist of the bits of the slow word, and the high-order bits are sign-repetitions. The technique could easily be adjusted to provide trailing zeroes instead of sign repetitions, or even a mixture of the two.

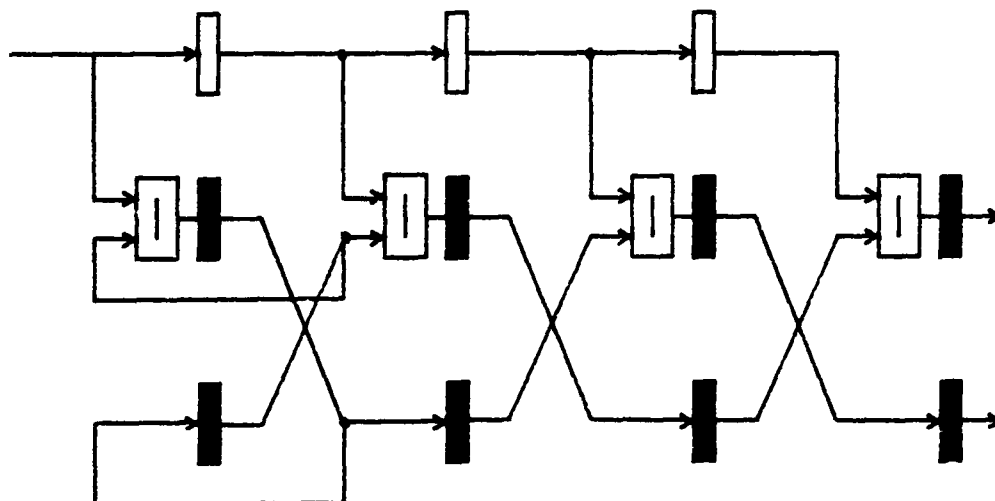


Figure 7.13(a): increasing dynamic range

Decreasing dynamic range

This structure (Figure 7.13(b)) performs the inverse operation of the previous. Here a fast SISO loads a slow PISO, and the top half of the fast word is discarded. The technique could easily be adjusted to discard trailing zeroes instead of sign repetitions, or even a mixture of the two.

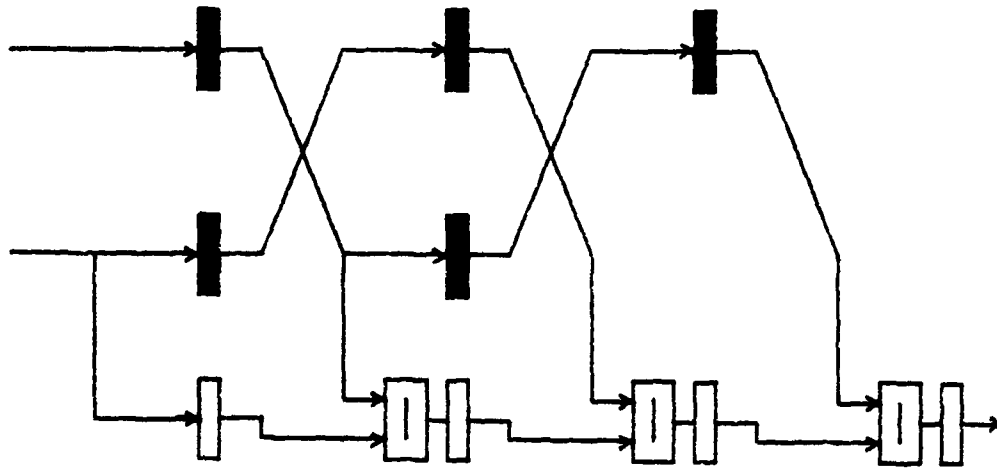


Figure 7.13(b): decreasing dynamic range

Increasing throughput

The above techniques have no effect on word-rate. This technique does - here two slow words are input in half-staggered form, and are output consecutively in a single fast stream. To achieve this, we use separately-controlled multiplexers (requiring twin control inputs) in the fast PISO, to allow alternate loading of the two slow words. Figure 7.13(c) shows this structure.

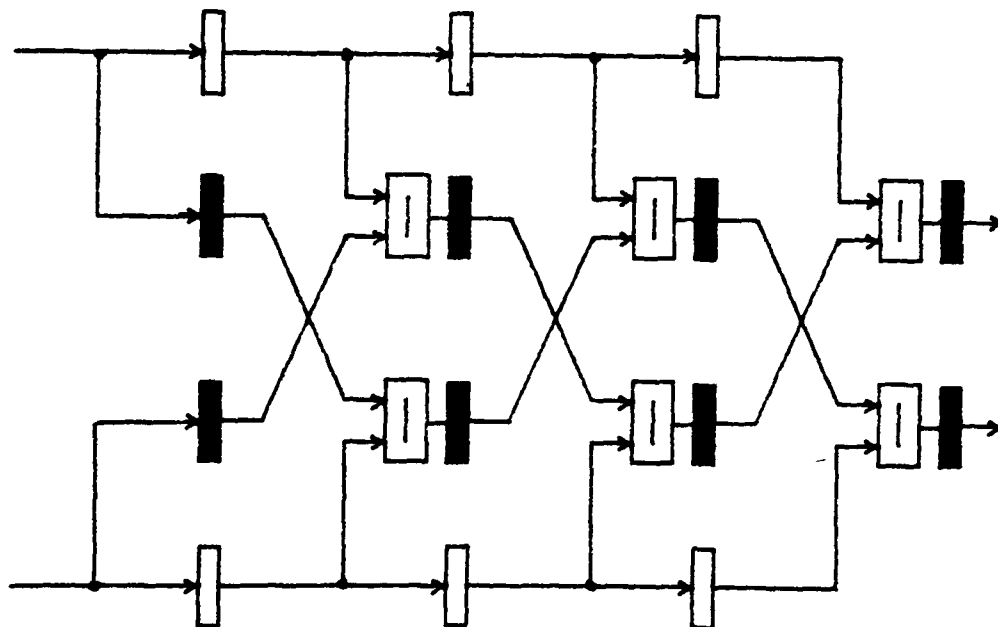


Figure 7.13(c): increasing throughput

Decreasing throughput

This structure (Figure 7.13(d)) performs the inverse operation of the previous. Here a fast SISO loads two slow PISOs, which output two slow words in the same half-staggered form as before.

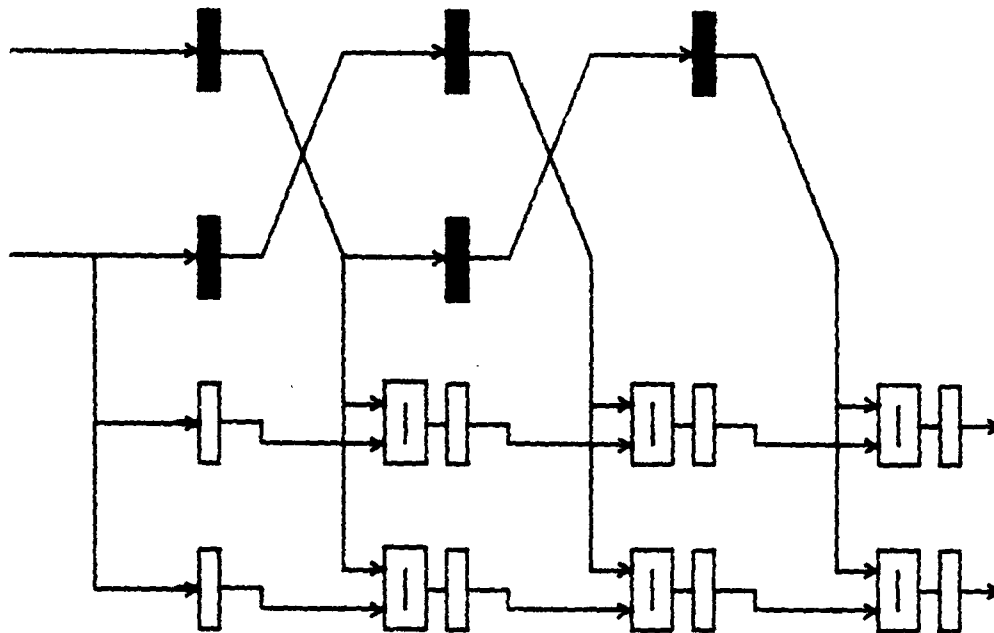


Figure 7.13(d): decreasing throughput

It should be noted that the area costs of these interfacing elements are considerable, and their use must be justified in terms of overall system costs.

7.5. Stripping down the CSAS computer - the automultiplier

Chapter 3 introduced the S/P flush multiplier, a variation of the S/P multiplier architecture which produces full-precision serial output at low area cost. However logic gates included to form PPs and clear the accumulator between product calculations contribute to hardware complexity, and impair performance. A novel multiplication architecture - the 'automultiplier' [175] - pipelines the formation of PPs and dispenses with gating in the critical sum and carry paths internal to the array, reducing the computational element to the minimum full-adder at each stage. The automultiplier is so-called because its accumulator is automatically reset in the final cycles of a product calculation, and thus requires no internal hardware for initialisation. The resulting low-complexity multiplier array may sustain maximally high

clocking rates.

Automultiplier architecture

Figure 7.14 shows the flush multiplier - it can be seen that each computational element has 5 inputs.

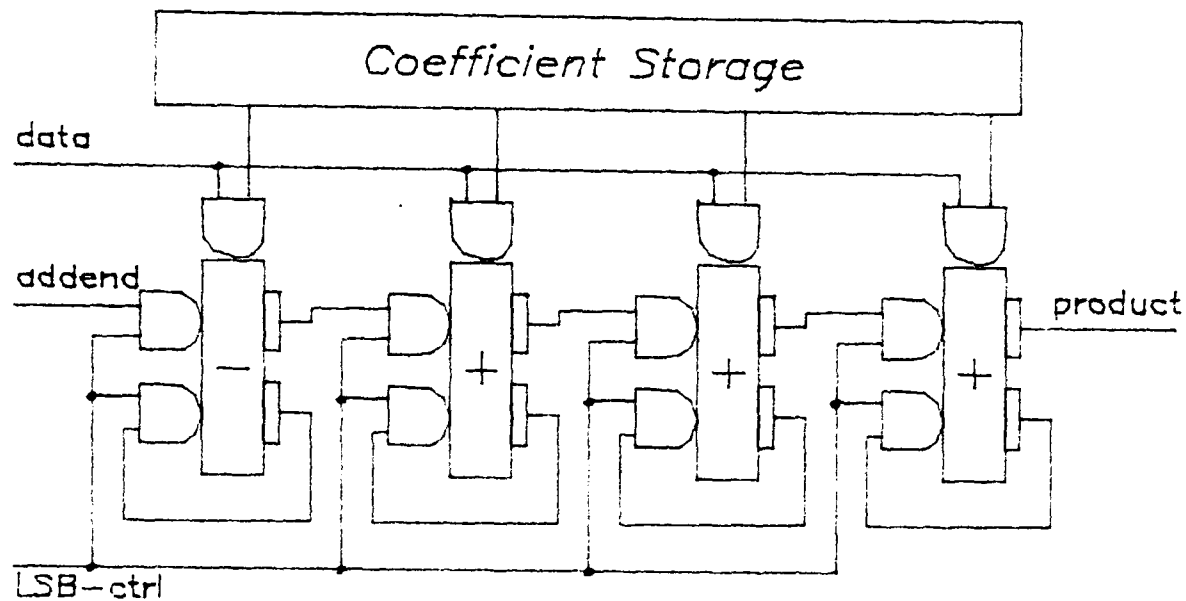


Figure 7.14: the S/P flush multiplier

By pipelining PP-formation, we may reduce this figure to 4, at the cost of a latch per stage and one clock cycle of latency. As evaluation-time of a logic block is related to fan-in and logical complexity, this simple step has a beneficial effect on potential maximum clock rate.

Conventional wisdom dictates that the accumulator in a S/P multiplier should be cleared between product calculations. As the accumulator contents are in carry-save form, this does not necessarily mean that all signals should be forced to logical zero. The requirement is actually that the residue sum and carry words be additive inverses. The S/P flush multiplier has the beneficial property that, if the final PP is correctly interpreted (i.e. subtracted), the carry-save residue consists of product sign-repetitions. The residue will be (effectively) all zeroes in the case of positive product, and (effectively) all ones in the case of negative product. The residue may be easily cancelled, by insertion of an incrementing 'cancel-bit' which is conditional on product sign, i.e. the exclusive-OR of data and coefficient MSBs.

Three hardware conversions must be performed on the structure of Fig. 7.14 -

a) re-introduction of final-PP subtraction logic, b) generation of the cancel-bit, and c) pipelining of PP formation. The subtraction may occur at any time during broadcast of data sign-repetitions, providing all subsequent PPs are zero. We choose to subtract the final PP at time $n+1$, forcing all subsequent PPs to zero by clearing the coefficient store (the storage latches must have 'clear' facility - however these latches are not in a critical-timing path). Figure 7.15 shows the automultiplier.

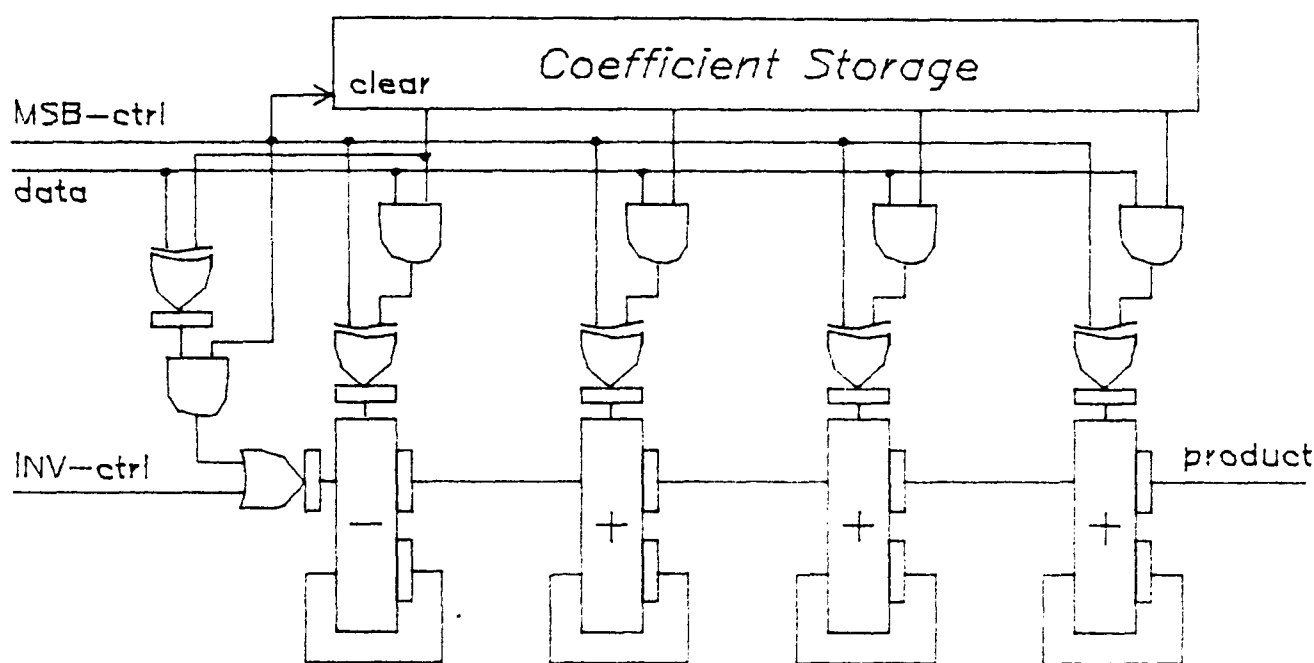


Figure 7.15: the S/P automultiplier

Cell complexity in the main array is now at a minimum, as there is no gating on any input of the full-adder. This structure may sustain clock rates comparable to fully-pipelined parallel multipliers, e.g. bit-systolic arrays [66].

Control

In steady-state operation, two control signals are required. These we name *INC* and *MSB* - each is a normally-low signal which goes high for one clock cycle per multiplication (i.e. one may be generated locally by delaying the other). The difference between the automultiplier and other bit-serial multipliers is that control signals are *not* required in the critical logic blocks (which cannot be pipelined) internal to the accumulator. Thus the critical logic blocks, which compute sum and carry signals, exhibit minimum size and delay, while minimal latches may be used throughout. Conventional bit-serial multipliers must pay for accumulator control in

either logic or latch complexity.

Control is required for 4 purposes - to invert the bits of the final PP (*MSB*), to clear the coefficient store (*MSB*), to increment the PPS for true subtraction of the final PP (*INC*), and to cancel residual sign-repetitions if the product is negative (*MSB*). The additional logic at the front of the array performs the latter two actions. The effect of *MSB* must appear at the leading end of the CSAS array $n+1$ clock cycles after commencement of a product calculation, and that of *INC* m clock cycles in advance of this.

Initialisation

Correct functioning of the automultiplier depends on the ability to force a numerical zero (in carry-save form) into the accumulator, while the last $m-1$ product bits are clocked out. There is the possibility that spurious states may form in the accumulator on power-up. These may be 'flushed out' by performing the calculation 0×0 , with *MSB* (but not *INC*) inhibited. Only one pathological state may arise - if a logical one should form in the borrow-path of the initial subtracter, the action of the subtracter (with zeroes on other inputs) is to sustain it. Should such a state exist, provision of *INC* will cancel it - otherwise *INC* will propagate down the array and disappear off the end, as do other spurious states.

7.5.1. Automultiplier summary

A novel serial/parallel multiplication architecture has been described, of similar hardware complexity to existing full-precision serial/parallel multipliers, but featuring a balanced distribution of logical complexity, so that no logic block has more than three inputs, and minimal latches may be employed. No control functions are performed inside the carry-save add-shift array. As a result, this architecture is capable of sustaining clock rates comparable to fully-pipelined parallel multipliers.

In the context of a serial-data function library, enhanced clock rate offers no advantage if other library elements cannot keep pace. Thus the automultiplier departs from the serial-data conventions established throughout this thesis, forming an evolutionary branch away from the mainstream. This is not to imply imminent

extinction - it might find application in reduced, high-performance serial-data function libraries.

7.6. Final comments

Elementary constituents of a bit-serial cell-library for digital signal processing have been specified, using several formats for computation and communication. The automultiplier represents a novel departure from conventional multiplier design, centred on a control-free accumulator which promises high clocking potential at low area cost. However the bulk of the novelty in this chapter stems from the throughput enhancement techniques, and the operational choices which arise from their use.

Three such techniques have been proposed. Twin-pipe and radix-4 realisations dramatically improve area-time product. Static operand storage costs are the same as those associated with conventional bit-serial architectures - only active logic increases in area. Multi-precision techniques, while offering no improvement in area-time product, allow arbitrary tradeoffs in these two dimensions.

Figure 7.16 shows an area-time plane, over which a designer may roam in search of the optimal mix of techniques for his application. Diagonal lines depict constant area-time product. The twin-pipe architecture serves as a starting-point, or 'default' option, from which to select architectures. Note the incompatibility of DA and radix-4 techniques.

Using the techniques outlined in this chapter, a function library may contain several versions of the same functional element, tailored to different performance requirements. Access to such *operationally* flexible library elements increases the architectural options available to systems designers, leading to potentially more efficient systems implementations. Arbitrary tradeoffs between area and time are possible, even within the same serial-data system.

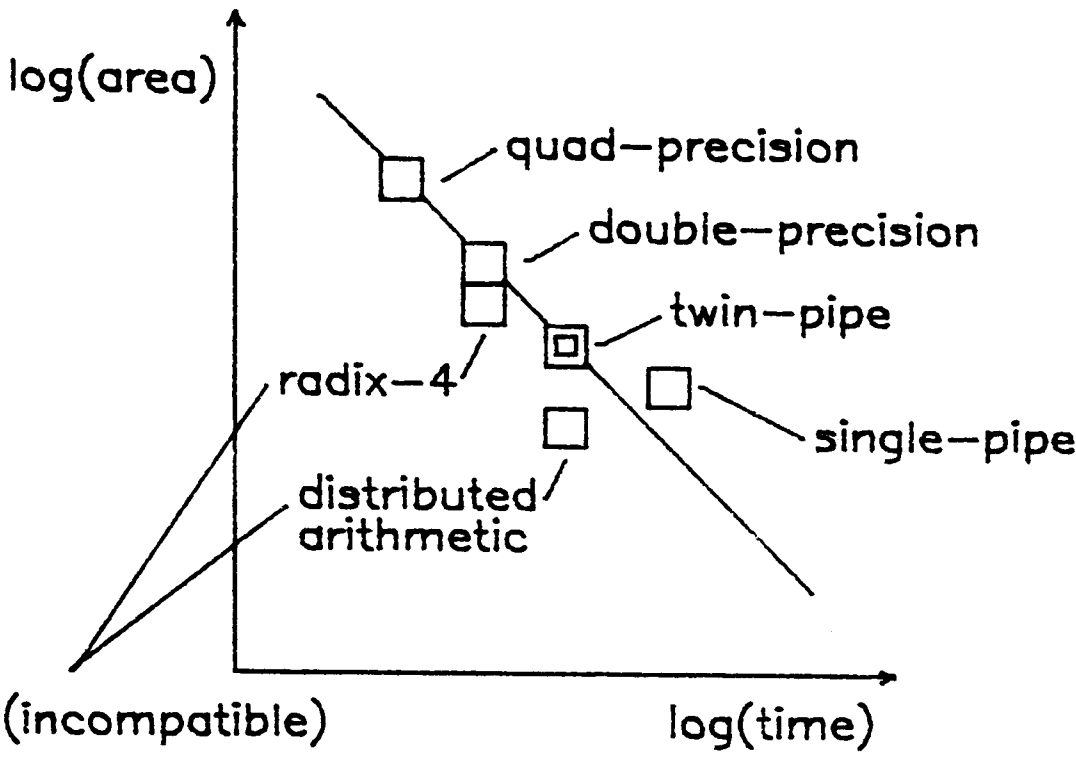


Figure 7.16: designer options on the area-time plane

Chapter 8

Full-span structural compilation of serial-data hardware

Throughout this thesis it has been stated that the *FIRST* function library suffers from three instances of inflexibility - *functional*, *technological* and *operational*. Provision of flexibility in these three areas should greatly enhance the ability of users to realise efficient solutions to real-time computational problems. Architectural techniques which promise flexibility in the first and third areas have been outlined in the previous two chapters - however the means of incorporating these suggestions into a unified design environment have not yet been disclosed. Provision of flexibility should not re-impose the burden of complexity on the designer. Software tools must manage the increased flexibility, leaving the designer free to reap its benefits.

The development of design automation software to overcome the three inflexibilities [114][†] is underway. The software, known as '*SECOND*' (*Synthesis of Elementary Circuits ON Demand*), allows a library of 'leaf-cells' to be specified in technology-free form (i.e. as small blocks of logic function plus latches), along with composition procedures for automatic assembly of functional primitives. *SECOND* provides the user with the facility not only to call on previously defined functional primitives, but also to specify his own, in both cases with technology-independence - the resulting library will be *functionally*, *technologically* and *operationally* flexible. It is intended to support single-pipe and twin-pipe representations of each primitive from the outset.

A serial/parallel multiplier has been used as a trial primitive for the new design route. It was successfully fabricated and tested. This device illustrates many of the concepts of this and the previous two chapters.

[†] published work by the author.

8.1. *FIRST* in perspective

Underneath the *FIRST* user-interface is a *fixed* library of around twenty parameterisable bit-serial function blocks, or 'primitives'. These in turn are composed of configurations of hand-crafted 'leaf-cells', bound to a *fixed* technology and maximum clocking rate. Computational and signal bandwidths may be matched through a variety of multiplexing techniques, but the throughput of each primitive element is *fixed*.

FIRST has a 'typed' hierarchy, as shown in Figure 8.1 (the legal call structure was illustrated in Figure 5.1). Here upper-case letters represent levels with physical significance.

<i>FIRST</i>	SYSTEM subsystem CHIP operator PRIMITIVE
<i>SECOND</i>	PRIMITIVE subprimitive SLICE subslice BLOCK subblock DEVICE

Figure 8.1: typed hierarchies in *FIRST* and *SECOND*

Lower-case levels are notional, used for design management only. The construction rules are simple:

- 1 Physical objects can only contain objects at lower levels, no lower than the next physical level.
- 2 Notional objects can contain objects at equal or lower levels, no lower than the next physical level.

Each physical level in the hierarchy has an associated assembly methodology. Assembly in *FIRST* is 'blind', i.e. assembly procedures at different physical levels of hierarchy are unconnected. In *FIRST*, SYSTEM assembly was never implemented, as commercial board layout packages abound. However, CHIP interfaces are arranged to guarantee communication at the function library clock rate, up to a predefined fan-out limit.

CHIPS are assembled by place-and-route software, using a simple floorplanner driven by a flattened netlist of PRIMITIVEs resulting from language compilation. The *FIRST* floorplanner treats PRIMITIVEs merely as rectangles whose behaviour at the interfaces is well-defined. PRIMITIVE interfaces are also arranged to guarantee communication at the function library clock rate, up to a predefined fan-out limit.

PRIMITIVEs are functional elements such as multipliers, serial memories etc., whose flexibility is greatly increased by parameterising many features. Each PRIMITIVE instance is assembled, according to its parameters, using ad hoc composition procedures and hand-crafted layout cells, details of which are hidden from the *FIRST* user. The production of both layout cells and composition procedures is time-consuming. In a sense, yesterday's chip-level problems have become today's cell-level problems. Through automation of cell synthesis and primitive composition, the same benefits may be brought to function library design as are currently brought by silicon compilers to chip design.

8.2. Synthesis of Elementary Circuits ON Demand

A typed hierarchy like that of *FIRST* is useful, both for the designer's management of information, and for capturing physical partitioning information. To implement full-span structural compilation, we extend this concept down to the device level (Figure 8.1). A secondary software architecture underneath *FIRST*, as

shown in Figure 8.2, gives the sophisticated user increased power by allowing him direct access to the function library.

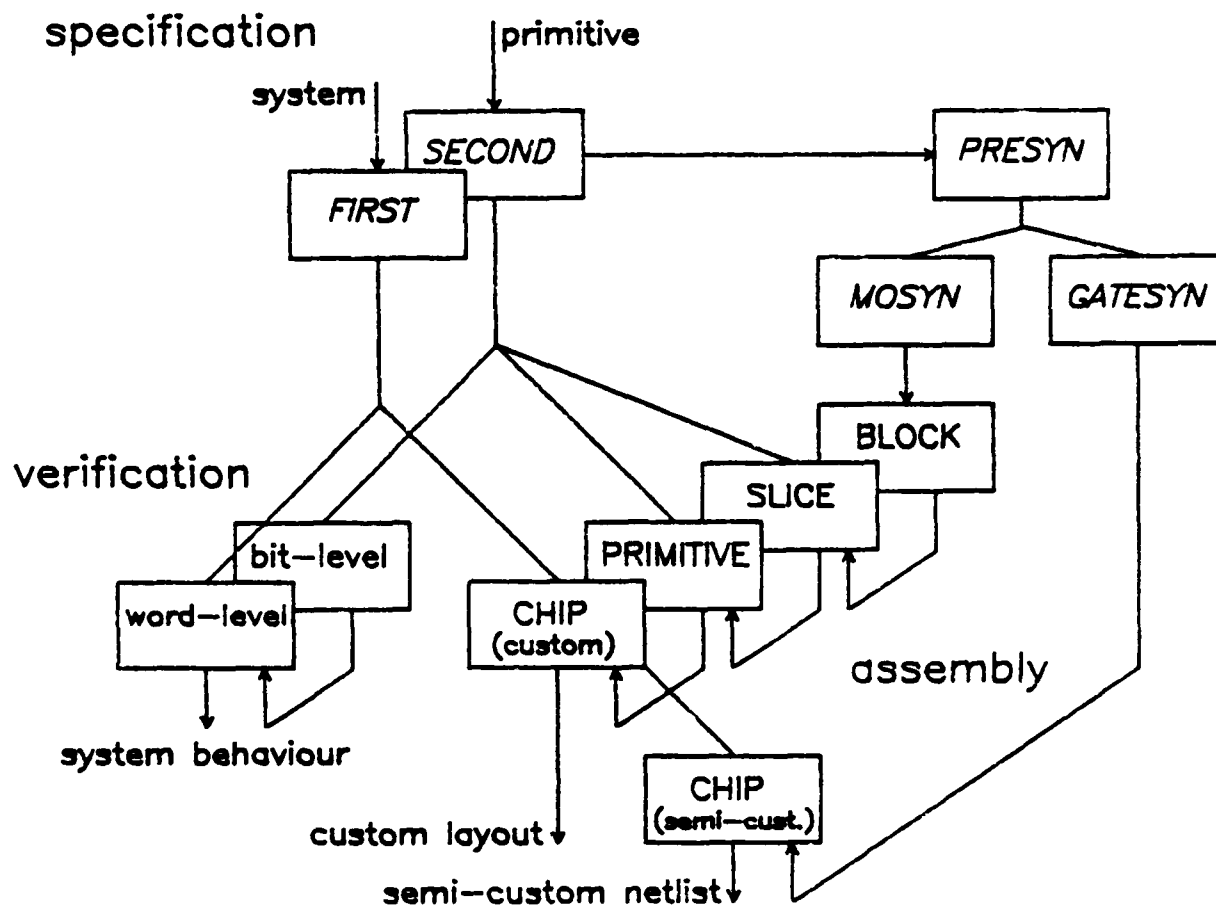


Figure 8.2: software architecture

SECOND allows the designer to specify custom primitives, as a hierarchical list of logic blocks and latches. Each physical level of hierarchy has its own assembler (floorplanner), but only at DEVICE level are ad hoc techniques employed. Logic function is only captured at BLOCK level - above BLOCK level we are only concerned with 'boxes and wires'. Logic BLOCKs (or subblocks) are described in terms of functional 'atoms' (AND, NOR etc.), or as binary cubes identifying the on-set, off-set and don't cares of the required function [121, 113].

As well as customised logic blocks, the designer may call on either previously used blocks or pieces of hand-crafted layout. The former facility is analogous to software programmers using libraries of known-good high-level coded subroutines to improve productivity, while the latter facility is analogous to software programmers calling known-good machine-code subroutines to improve performance. Underlining the analogy, the latter comes with a loss of design portability. Latches may be

treated as special BLOCKs themselves, rather than being incorporated in BLOCKs. Frequently-used objects such as these may benefit from inclusion in the microcell-library, at the small investment of an expert's time to hand-pack layout in the target technology. Failing this, they may be specified in identical manner to logic blocks. In order to maintain performance, we forbid the cascading of BLOCKs, i.e. BLOCK outputs connect to latches only.

The user of *FIRST* requires expertise in architectures, certainly at the systems level. Should his expertise extend down to the logic gate level, *SECOND* will allow extensions of the basic library in a manner analogous to the production and maintenance of the subroutine libraries produced by software programmers. If not, he will still have the original function library at his disposal, but with the important difference that he will be able to take advantage of improved silicon processing when the opportunity arises.

SECOND promises an effective route from *structural* intent into silicon. The route from *behavioural* intent into silicon - the 'holy grail' of systems designers [176] - is outwith the immediate scope of this project, however future behaviour-to-structure compiler developments will be able to call on the techniques proposed.

It is our wish that *SECOND* support not only custom, but semi-custom realisations of user-defined function. *MOSYN* is useful only in the former context. A tool is under development which produces optimised netlists from BLOCK specifications, targetted at semi-custom implementation media such as libraries of standard cells. *GATESYN* is to be compatible with *MOSYN* at the user interface.

8.3. Functional components of *SECOND*

The purpose of *SECOND* is to allow the specification of PRIMITIVEs, their functional verification, and their implementation in custom or semi-custom form in arbitrary technologies. To this end, a software architecture similar to that of *FIRST* is required (Figure 8.2), but with some important differences.

8.3.1. PRIMITIVE specification (design capture)

Design capture in *FIRST* is via a single, high-level language interface. The *FIRST* user specifies systems as hierarchical lists of functional objects ('boxes and wires'), with PRIMITIVES at the lowest level of hierarchy. The *SECOND* user, on the other hand, is involved in implementation details somewhat closer to technological interfaces. In particular, he has to support multiple representations of PRIMITIVES. As PRIMITIVE specification above BLOCK level is still a matter of 'boxes and wires', a *FIRST*-like interface may be extended part of the way down the *SECOND* hierarchy. VX is a graphical design tool under development for just such a purpose (a similar tool was developed for *FIRST* [177][†] but not adopted).

A representational dichotomy occurs at BLOCK level, not just between custom and semi-custom forms, but between operational forms as well. While the custom circuit techniques used in *SECOND* rely on conductance of transistor networks [43], semi-custom versions of logical operators are implemented as collections of logic gates or standard cells. Moreover, 3 representations of a BLOCK must be maintained if *SECOND* is to support both single-pipe and twin-pipe operational modes. These are the single-pipe version, and π - and μ -timed pairs of (reduced) twin-pipe functions.

To illustrate the use of *PRESYN*, *MOSYN* and *GATESYN*, we pursue the design capture of the CSAS computational element (gated full-adder) from the 2C fractional S/P multiplier discussed in Chapters 3 and 6. Only twin-pipe realisations are shown (single-pipe realisations follow more directly from the specification). As semi-custom implementations cannot support the performance of custom, we choose to relax the restriction on cascading BLOCKs in semi-custom assembly, thus avoiding the duplication of logic function necessary in the custom case. Such duplication exists at several levels in the cited example.

The software architecture of Figure 8.2 presents a single interface to the *SECOND* user for BLOCK specification. *PRESYN* takes as input the specification of a BLOCK in single-pipe form, but with identification of even-pipe and odd-pipe control signals. *PRESYN* produces *MOSYN* language listings of the various BLOCK

[†] published work by the author.

representations, which drive *MOSYN* or *GATESYN* to produce custom and semi-custom representations respectively.

The user might specify as follows:

```
EVEN: lsb
ODD: msb
NAME: sum
FUNCTION: xor (and (xor (msb, cof), dat), and (pps, lsb), and (cin, lsb))
NAME: carry
FUNCTION: major (and (xor (msb, cof), dat), and (pps, lsb), and (cin, lsb))
```

PRESYN separates even- and odd-pipe control, producing *MOSYN* language listings of even- and odd-pipe functions:

```
*function sum_ev
C1: xor (and (cof, dat), and (pps, lsb), and (cin, lsb))
else C0;
*function carry_ev
C1: major (and (cof, dat), and (pps, lsb), and (cin, lsb))
else C0;

*function sum_od
C1: xor (and (xor (msb, cof), dat), pps, cin)
else C0;
*function carry_od
C1: major (and (xor (msb, cof), dat), pps, cin)
else C0;
```

These listings drive *MOSYN* and *GATESYN*. *MOSYN* produces four full-custom transistor networks, each of which is dedicated to the evaluation of one logic variable. *GATESYN* on the other hand produces only two semi-custom netlists (one per pipe), and gates are shared where possible to minimise area. Figure 8.3 illustrates possible full-custom and semi-custom realisations of the even-pipe functional blocks. In the custom (high-performance) implementation (Figure 8.3(a)), three AND-gates are functionally duplicated in SUBBLOCKs for both sum and carry formation - in semi-custom, they could be shared (as indeed could some gates in sum and carry SUBBLOCKs). Figure 8.3(b) shows a potential semi-custom gate-level implementation, minimised and optimised for 2-input NOR/NAND usage. Ideally *GATESYN* should minimise gate count in this manner, without user-intervention.

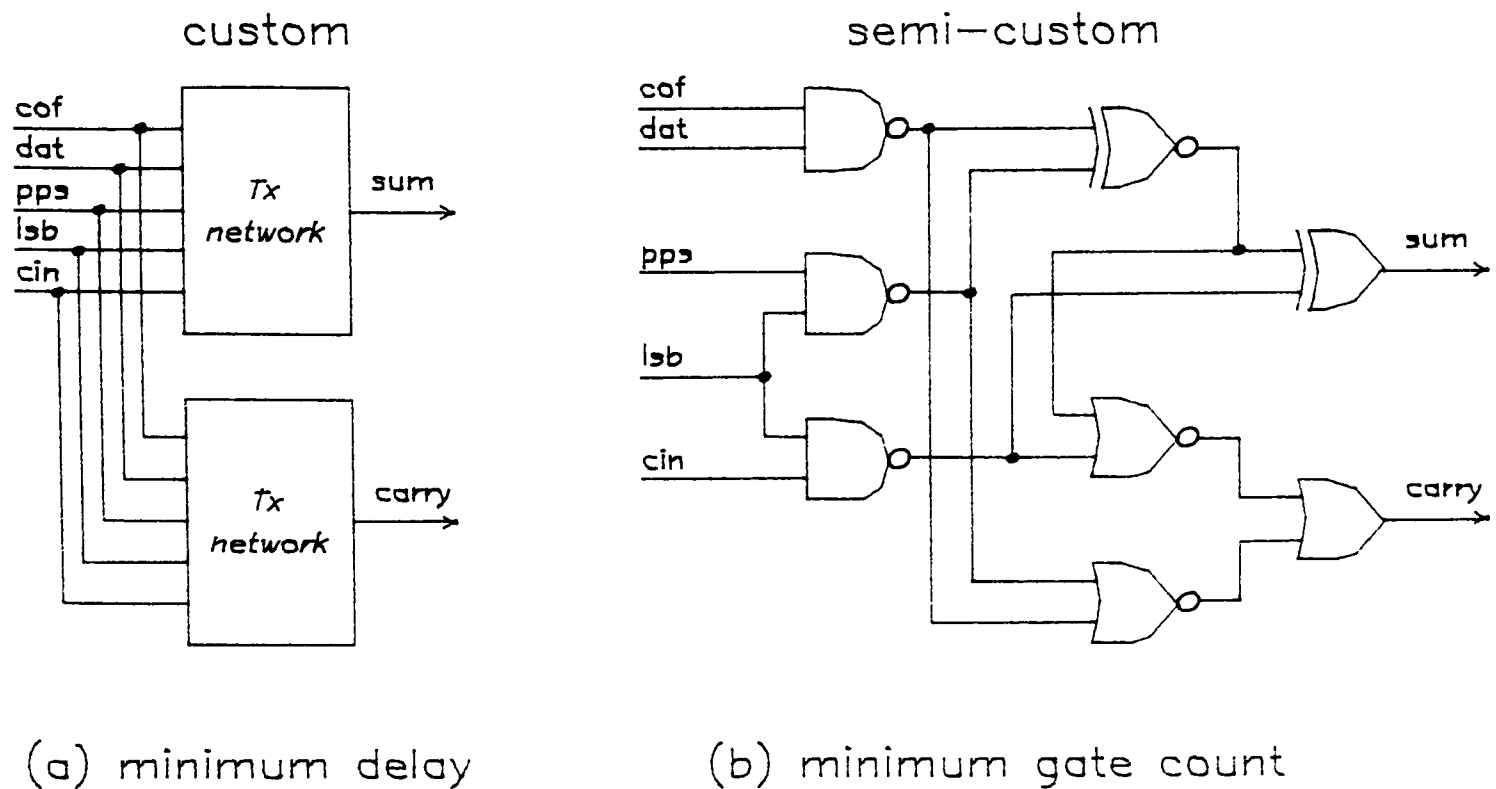


Figure 8.3: a) custom, and b) semi-custom gated full-adder realisations

Parameterisation

As in *FIRST*, the function library may be made more flexible by including families of functionally 'soft' (i.e. programmable at compile-time) PRIMITIVEs such as multipliers and memories, instead of fixed instances of each. The family is identified by name and the individual by parameters.

Chapter 7 emphasised that bit-serial hardware consists mainly of shift-register structures with built-in function. These can be single-stage structures (e.g. adder), linear arrays (e.g. multiplier) or even two-dimensional arrays (e.g. word-size memory [7]). In the multiple-stage cases a PRIMITIVE consists of repetitions or cascades of arbitrary instances of hardware SLICEs whose internal content and external connectivity are *mostly* identical. Parameterisation of these aspects of SLICEs greatly increases the descriptive power of the design capture medium, allowing one SLICE description to service the entire PRIMITIVE family, thereby maximising designer efficiency. Two high-level programming constructs are useful here. The provision of *repeat* statements is mandatory, allowing variable-length arrays of SLICEs to be specified. The provision of *conditional* statements is also

desirable, allowing minor differences between SLICE instantiations to be specified.

Unification

We anticipate the eventual unification of *FIRST* and *SECOND* into a single user interface. Although design capture would then be uniform across the full span of systems design, the less sophisticated user need not deploy *SECOND*, instead using *FIRST* with an existing function library. Once again, we compare this with the novice software programmer who calls standard library subroutines, and the expert who writes his own. Each uses the same design medium, and both benefit from an efficient compilation environment.

8.3.2. PRIMITIVE verification (behavioural simulation)

An important difference arises when we compare tools required for verification in *FIRST* and *SECOND*. *FIRST* revolves round a floorplanner and function library which guarantees performance of CHIPS and SYSTEMs. Responsibility for this guarantee rests in the first case with the creator of the floorplanning strategy, and in the second with the individual designer of each PRIMITIVE. The user is only responsible for the *functional* correctness of his design - to this end he is provided with a word-level, event driven simulator. The domain of the simulator is sufficiently abstract (and run-times short) to allow algorithmic exploration and verification.

SECOND lets the user don the mantle of responsibility not only for functional correctness of PRIMITIVEs, but also for their *performance*. Thus circuit simulation must be employed at the BLOCK level to ensure that the logical depth between latches never exceeds the target for the particular function library in question. However this is only required in marginal cases, which are highlighted by *MOSYN*'s statistical information output.

Behavioural simulation now spans the full design space. In *FIRST*, a further responsibility of the PRIMITIVE designer is to provide a high-level language coding of the primitive's word-level behaviour. This responsibility passes to the *SECOND* user, as does the responsibility at the lower level of describing logical behaviour of BLOCKs. The latter task is made simple by the fact that BLOCKs are necessarily

small objects - logical simulators at higher levels may 'look-up' the verificational truth-tables generated by *MOSYN*. However at present the abstraction of *PRIMITIVE* behaviour to word-level remains a manual task, and is correspondingly a potential source of errors.

8.3.3. *PRIMITIVE* implementation (physical assembly)

At this stage the differences between custom and semi-custom implementations become clearly defined. While three physical hierarchical 'types' are provided for *PRIMITIVE* specification, these types are only significant in the custom case.

Semi-custom

Semi-custom design descriptions have no need for typed physical levels of hierarchy [178], although hierarchy still has an important part to play in efficient assembly of semi-custom chips. Here the responsibility of *SECOND* is to produce functionally correct netlists for manufacture - assembly is the responsibility of whichever semi-custom IC fabrication house is employed.

Custom

In the custom case, hierarchical typing allows the designer to dictate how the primitive is logically partitioned and physically assembled. In this way he is able to control the floorplanning of the primitive. At the lowest user level, logical *BLOCKS* are specified in *MOSYN* language, and *MOSYN* returns layout with no regard to external connectivity. Figure 8.4(a) shows symbolic gate-matrix layout of the even-pipe gated carry-block example produced by *MOSLAY* (the physical netlist assembler of *MOSYN*), and Figure 8.4(b) this layout fleshed out for particular design rules (in this case the 3- μ m, double-level metal CMOS supported by the MCE brokerage service).

SLICES are assembled by placing blocks, and *PRIMITIVES* by stacking slices. If genuine full-span compilation is to be achieved, it is imperative that these tasks be automated. While the latter task is mostly the fairly straightforward abutment of rectangles, we perceive the former as the single most difficult barrier between half-span and full-span structural compilation. Figure 8.5 shows the effect of simple

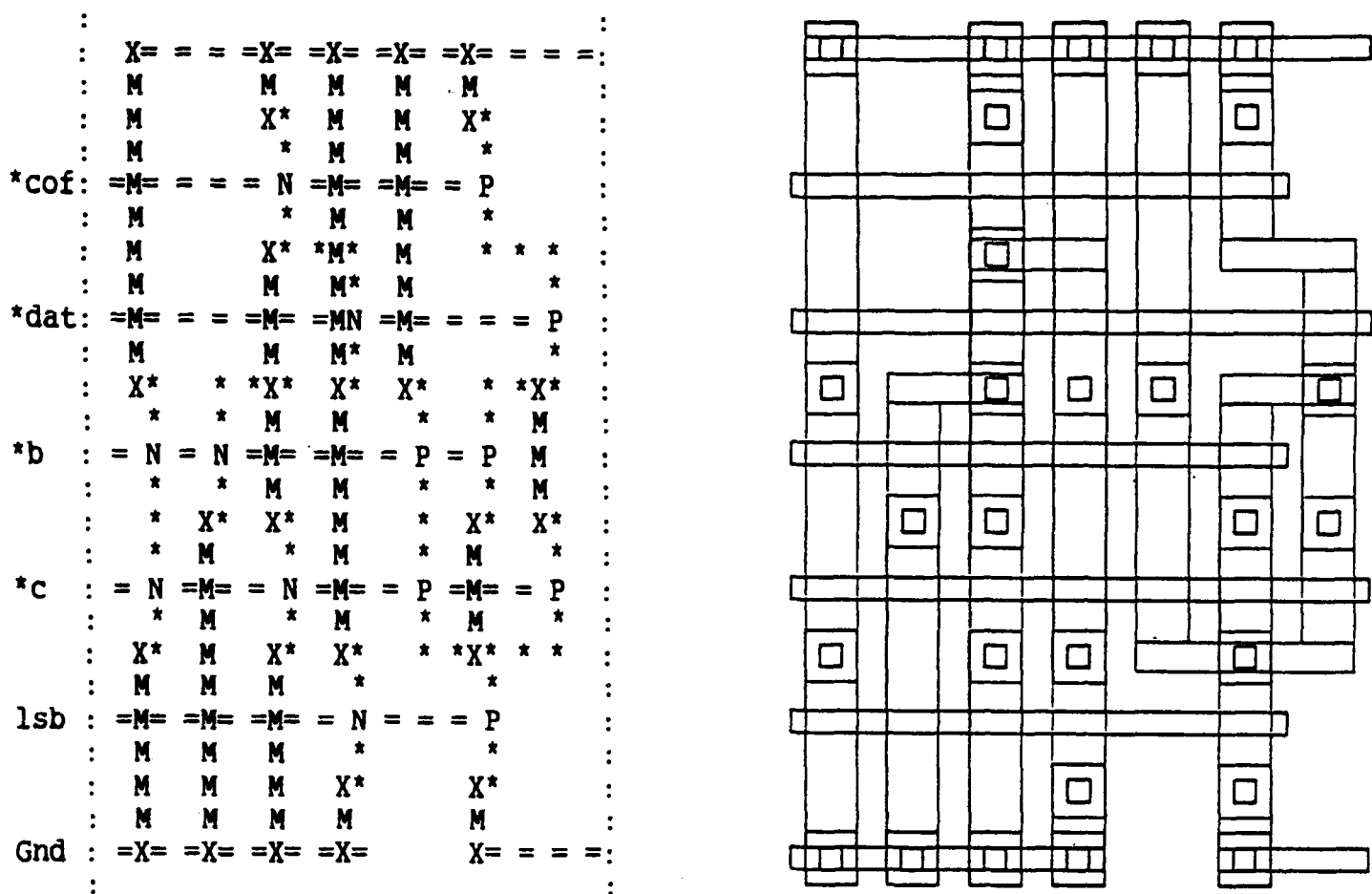


Figure 8.4: a) symbolic, and b) fleshed-out gate matrix layout

partitioning - in Figure 8.5(a) *MOSLAY* has been allowed to lay out an entire subslice, whereas in Figure 8.5(b) the subslice has been reduced to constituent BLOCKS and latches, these elements laid out by *MOSLAY*, and the results hand-assembled. It is intended that SLICE assembly in *SECOND* produce layout automatically.

SLICE assembly entails optimal placement and routing (in metal 2) of BLOCKs. BLOCKs and latches may share wells and supplies in a 'back-to-back' manner. Efficient SLICE assembly may require that BLOCK assembly use external connectivity information, compromising the elegance of 'blind' typed assembly.

8.4. Example PRIMITIVE design - the serial/parallel multiplier

To illustrate some proposed concepts and anticipated problems concerning *SECOND*, we report two exercises in the design of a 6-bit serial/parallel (S/P) multiplier for use in a datapath environment. The multiplication architecture was

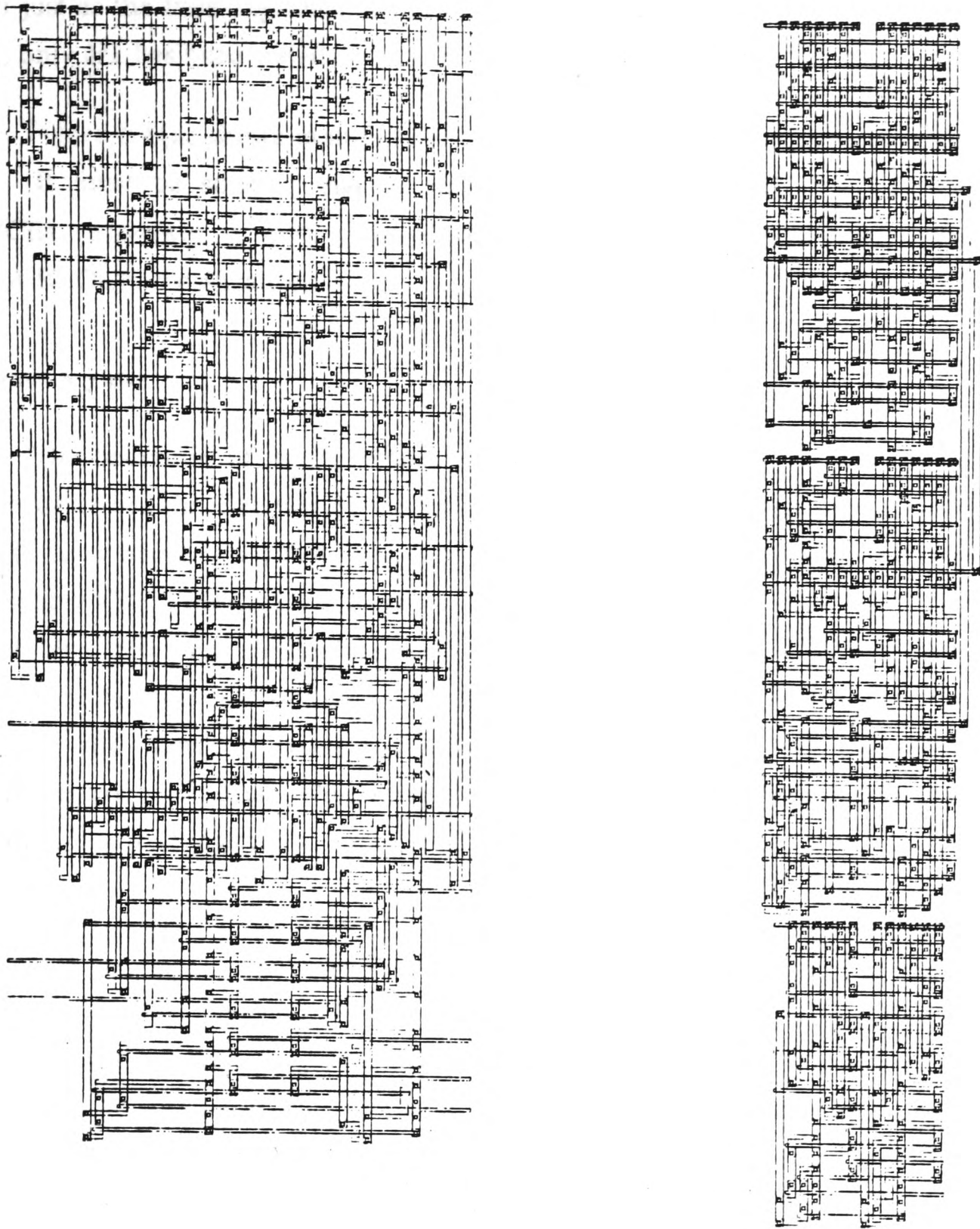


Figure 8.5: a) unpartitioned, and b) partitioned subslice

defined by the author, using the twin-pipe technique described in Chapter 7. The first multiplier, *M1*, was assembled manually by two M.Sc. students (J. I. Mhar and H. Paul), as part of their course project work. The second, *M2*, was assembled by M. S. McGregor, with some help from the author. *M2* returned from fabrica-

tion just as the finishing touches were being put to this thesis. At the time of writing, only *M1* has been tested.

Figure 8.6(a) shows multiplier *M1*, and 8.6(b) the freshly-delivered multiplier *M2*. It must be stressed that in neither case was *SECOND* used to assemble SLICES. However in both cases *MOSYN* was used to generate logic trees from input BLOCK specifications.

Layout styles

The operation of the multiplier subject is similar to the fractional S/P multiplier described in Chapter 3, except that interfaces are bit-parallel. A parallel ripple-carry adder is used to form the product (similar to the complex multiplier of Appendix C). Twin-pipe architecture is implemented in the case of *M1* using static logic trees and dynamic latches, and in the case of *M2* using dynamic logic trees and static latches. *M1* occupies an area of 1.85 mm^2 in $3\text{-}\mu\text{m}$ double-level metal CMOS, and transistor count is 1200. *M2* occupies an area of 3.45 mm^2 , and transistor count is 1500. Thus the cost of fully static latches outweighs the savings in using dynamic logic.

M1 was laid out manually, using the same gate-matrix style as is envisaged for *SECOND* (the layout of Figure 8.5(b), corresponding to about half of a SLICE, was produced by J. I. Mhar in similar fashion). *M2* was laid out in a simple 'tiled' layout style, where transistors are pre-deposited in columns of 4 (P-N-N-P). Custom logic is realised by routing, or 'personalising' these tiles, using as few columns as possible, in a manner similar to gate array personalisation (except that polysilicon and diffusion as well as metal may be used as conducting media). This ease of layout comes at the cost of wasted silicon area, as can be seen from Figure 8.6. *M1* features 650 transistors/ mm^2 , *M2* only 435.

Test results of *M1*

Out of 20 fabricated parts, 6 were found to be functional. These were found to have an operating range (in terms of clock rate) of 1 Hz to 12 MHz. Functional testing was performed on a Tektronix Digital Analysis System (*DAS 9100*), up to a clocking rate of 10 MHz. Devices which passed this test were then monitored on

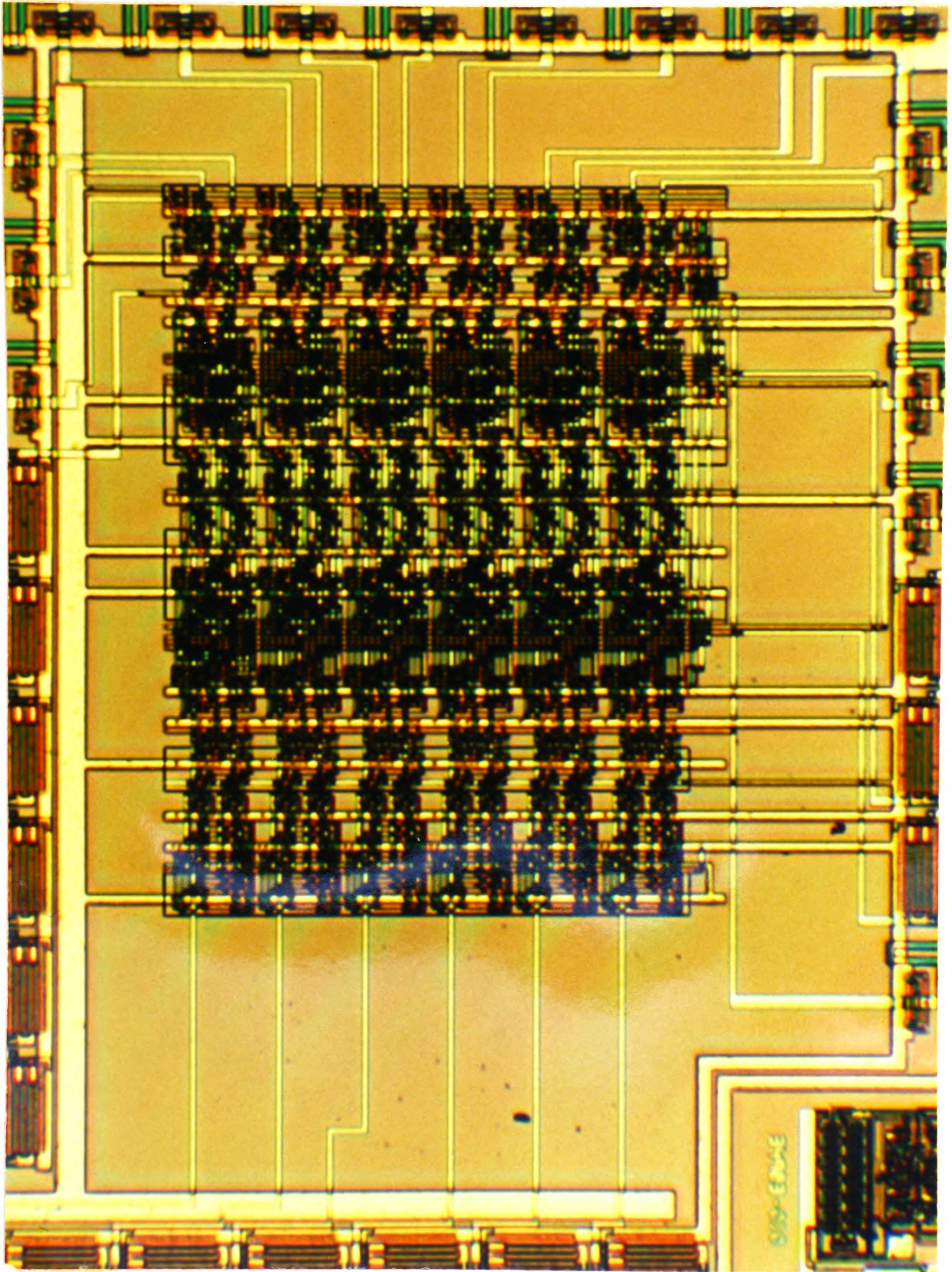


Figure 8.6(a): multiplier M1

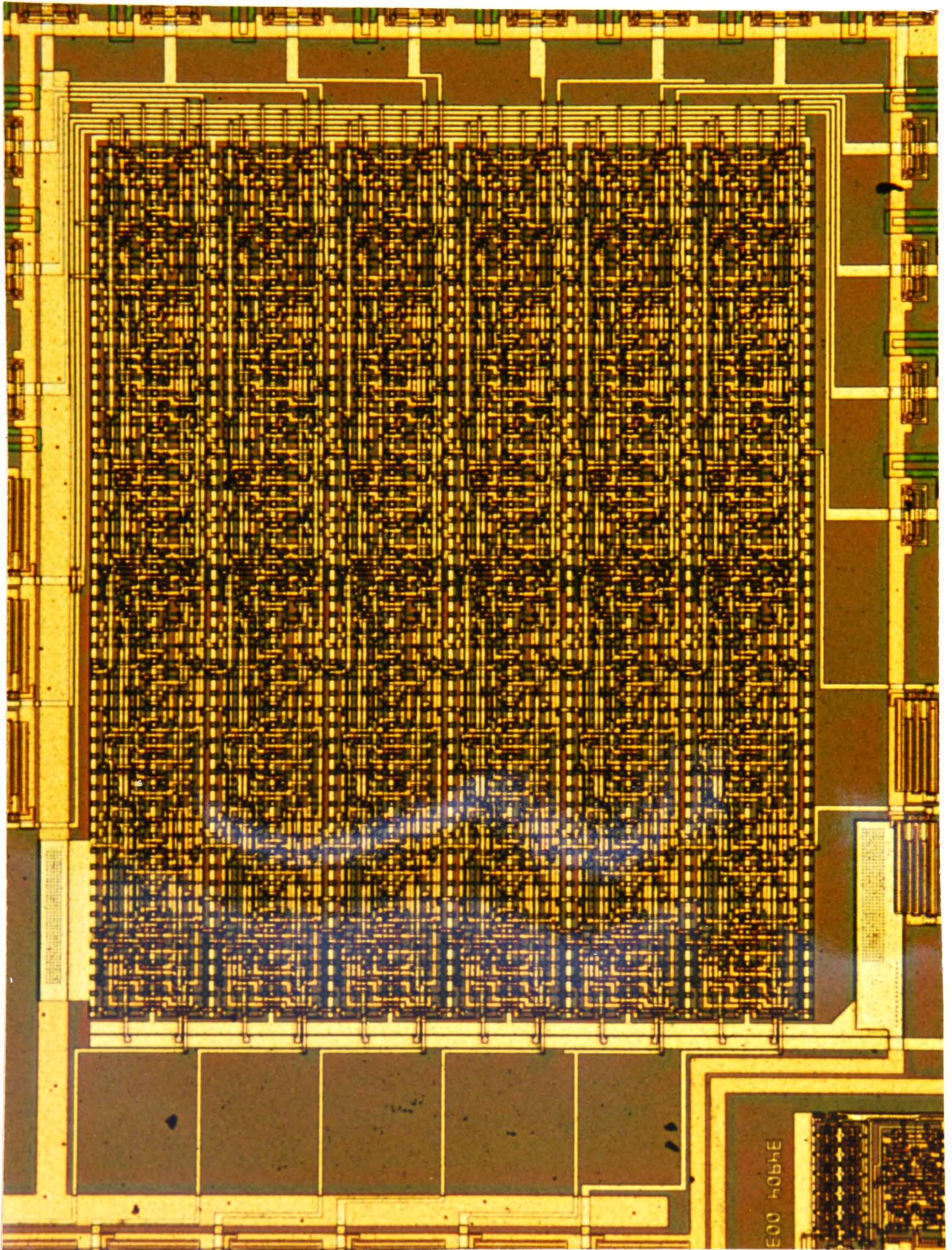


Figure 8.6(b): multiplier M2

an oscilloscope, and the clocking rate increased until failure was observed.

Figure 8.7 shows *RNL* waveforms which were produced by circuit-extracting from the layout (the glitches on the output lines are apparent while the ripple-adder is settling).

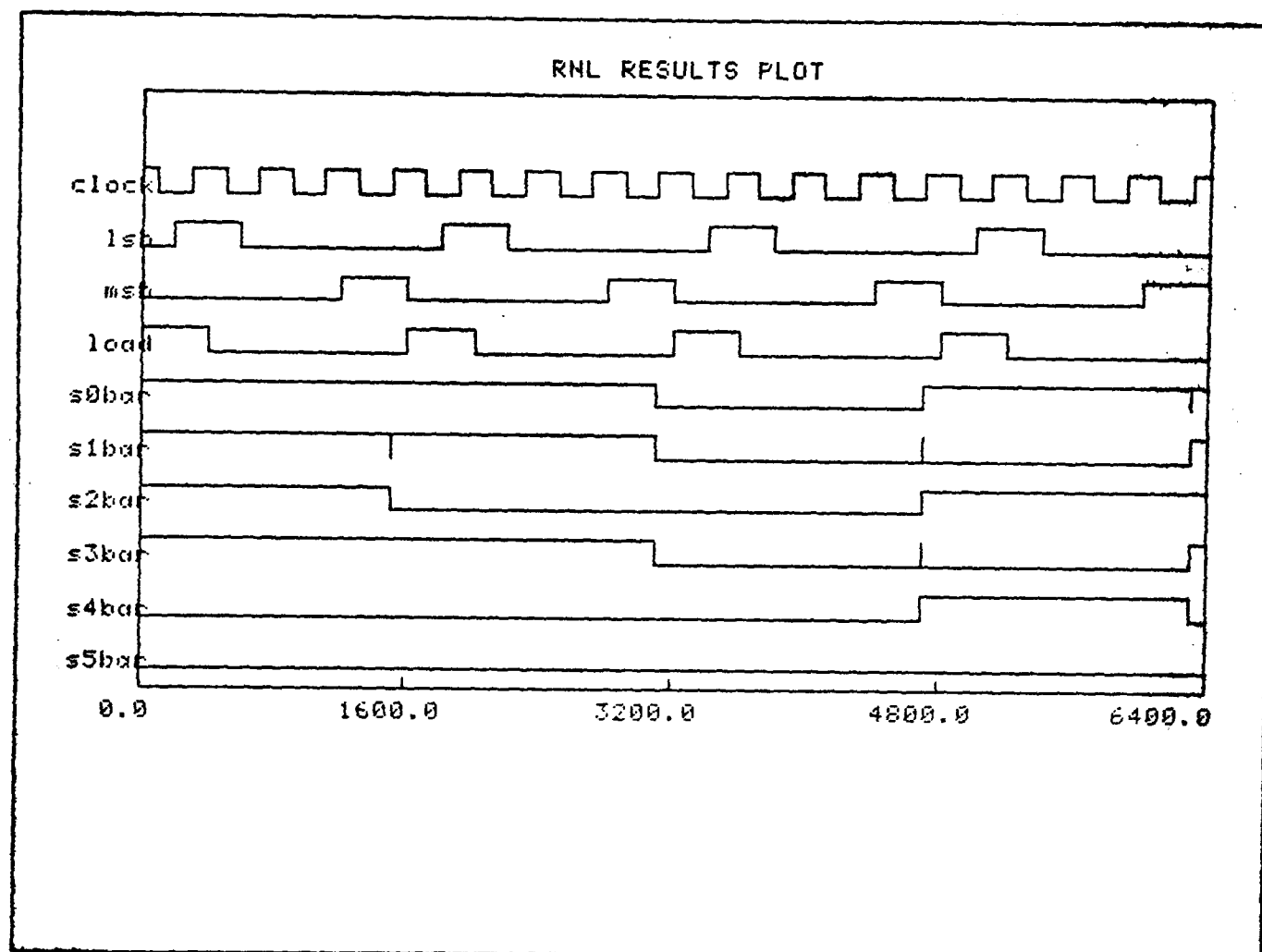


Figure 8.7: *RNL* simulation waveforms, multiplier *M1*

Figure 8.8 shows *DAS* waveforms produced by the device under test, using the same stimuli as the *RNL* simulations. The region marked in Figure 8.8 corresponds to the time interval spanned by Figure 8.7.

Some identifiable inefficiencies in circuit engineering limited *M1*'s maximum clocking rate - these included failure to buffer the broadcast data and control bits (thus incurring long rise times on global wires), long runs of polysilicon in the clock path (causing clock skew), and cascading of BLOCKs (instead of unique BLOCKs connected to latches only). We anticipate around a twofold performance increase in *M2*, which uses dynamic logic trees, metallised clock path, pipelined buffering of broadcast signals and no BLOCK cascading.

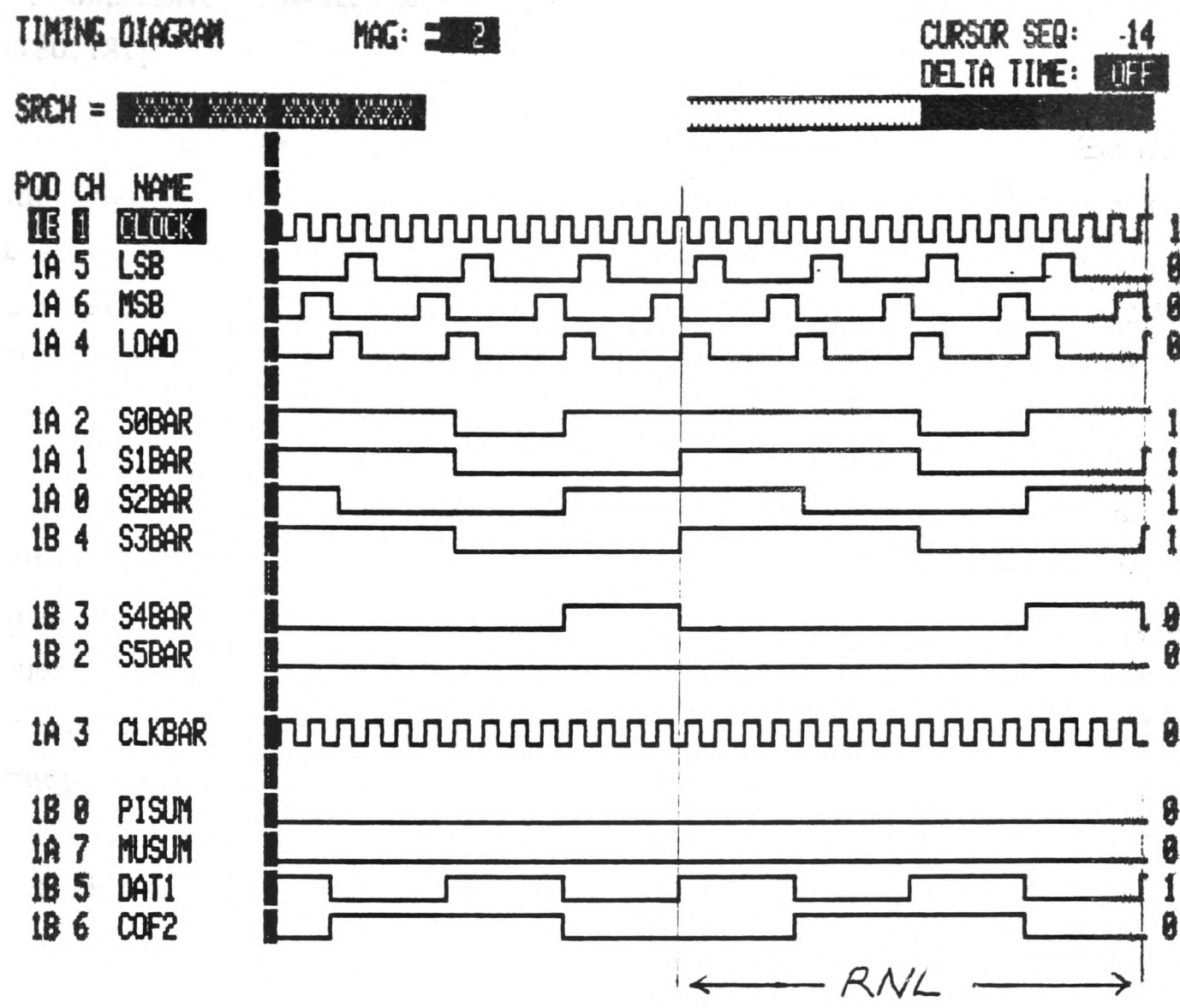


Figure 8.8: DAS test waveforms, multiplier M1

8.5. Other approaches

The implementation of full-span structural silicon compilation is in progress, and the researchers who are undertaking it (including the author) have much to learn about the tools and techniques of design automation at cell level. Good work on interactive module generation is under way at IMEC in Belgium [179], drawing on the resources of the *CATHEDRAL* project [128]. *CATHEDRAL* is a bit-serial silicon compiler which uses more sophisticated floorplanning than *FIRST*, and higher-level methods of design specification, but is targetted at a narrower applications area - digital filtering. Filters may be synthesised using 8 small, hand-crafted cells of common height. Collaboration between IMEC and UC Berkeley has resulted in

an impressive cross-fertilisation of ideas in DSP architectural synthesis, e.g. [180, 181].

Another bit-serial compiler project, *BSSC*, is in progress at General Electric in the USA [182]. Here systems are specified as algorithms in a high-level language closely related to *C* [170]. Control paths and synchronising delay networks (manually specified by *FIRST* users) are synthesised automatically. Like *CATHEDRAL*, *BSSC* uses a linearised, folded layout style similar to standard-cell placement systems.

Clearly the problem of module generation or cell synthesis is not peculiar to bit-serial architectures. Much work has been targetted at module generation in its own right, e.g. [183, 184, 185], and many of the ideas proposed may prove useful in the *SECOND* project. The incorporation and extension of these ideas is however outside the realm of this thesis and the current state of layout synthesis at Edinburgh University.

8.6. Final Comments

Silicon compilers have been proposed as the answer to the custom IC design 'bottleneck', by giving systems designers access to the skills of circuit and layout experts. However compilers based on function libraries merely move the bottleneck from chip level to cell level. Full-span compilation moves the bottleneck down to the device level. Circuit and layout expertise is only required in the creation of tiny 'microcells' such as transistors and contacts. In the semi-custom implementation route, such details are of no concern.

SECOND (Synthesis of Elementary Circuits *ON Demand*) is a design capture, simulation and layout package which will extend *FIRST*'s typed hierarchy down to the device level. Instead of the current assembly of *FIRST* primitives using ad hoc composition procedures and hand-crafted cells, *SECOND* supports the logic-level capture of cells and their automatic synthesis and (eventually) placement. The user controls hardware partitioning through hierarchical 'typing' - each hierarchical type corresponds to a physical level with associated assembler/floorplanner. Only at the device level are process-dependent, ad hoc techniques employed.

While the area-cost of chip designs produced by full-span compilation may at times exceed that of manual layout, the reduction in design effort afforded more than compensates for this. In custom realisations, area-efficiency may be improved by identifying distinct levels of hardware hierarchy, and invoking appropriate assembly procedures for each. Semi-custom place and route software may also exploit intelligent partitioning. A sub-optimal design realisation in today's technology may then compare favourably with an efficient realisation in yesterday's, at a fraction of the cost.

Chapter 9

Concluding remarks

The combination of design automation (via full-span structural silicon compilation) and the elegance and simplicity of serial-data computation promises designers rapid access to powerful systems solutions in the field of real-time, fixed-function numerical computation, e.g. digital signal processing. This thesis builds on work previously carried out at the University of Edinburgh in design automation using bit-serial architectures. The subject matter spans the complete design space from systems down to silicon, as well as issues of software for automation of the design process. As a result of this, some of the treatment (particularly at the silicon end) is necessarily shallow. On the other hand, the areas of serial-data arithmetic theory and architecture are covered in some depth.

A balanced examination of the pros and cons of the serial-data approach concluded that there are many advantages to be found in the serial-data execution of algorithms which exhibit no data-dependence. It was shown that serial-data architectures may be constructed from a small set of computational 'atoms' (latches, multiplexers, shifters and adders), although for efficiency it is prudent to bundle function into a larger set of larger elements (e.g. complex multipliers). The view of serial-data elements as shift-registers containing logic function succeeded in simplifying the specification of these larger elements. The fundamental operation of multiplication was examined in some detail, the field reviewed, and an optimal candidate (the serial/parallel multiplier in flush and fractional forms) selected for development of more advanced architectures.

The *FIRST* silicon compiler was discussed, and the basic concepts of a serial-data systems design methodology outlined. Compiler and methodology were then thoroughly examined in a case study of a satellite communications transmultiplexer, commissioned by the European Space Agency. This study represents a major systems design exercise, and makes a significant contribution to bit-serial architectural knowledge. It was concluded that *FIRST* might be improved by overcoming three types of inflexibility - *functional*, *technological* and *operational*. The remainder of

the thesis was devoted to providing flexibility in these three areas.

Novel complexity reduction techniques were demonstrated in the specification of architectures aimed at computational problems such as matrix-vector multiplication, which exploit several symmetry properties to effect savings in silicon area. A simple synthesis procedure was demonstrated for these architectures. Next a variety of techniques were shown, which bring performance increases over conventional serial architectures, using multi-wire computation and communication in a step towards bit-parallelism which loses none of the advantages associated with bit-serial architectures. The latter range of techniques make possible mixtures of operational domains (marked by throughput and/or dynamic range) within a single system, bringing *operational* flexibility to serial-data architectures, and answering two of the more common charges against bit-serial architectures - namely slower throughput in relation to bit-parallel realisations, and fixed system wordlength.

Further novelty was demonstrated in area-efficient incremental multiplication architecture which computes squares and sums-of-squares, and the 'automultiplier', a potentially high-performance serial/parallel multiplier based on a control-free accumulator.

Finally, *functional* and *technological* inflexibility were addressed by *SECOND*, a design automation route from systems specification into silicon, whose initial progress and goals were outlined. Methods were proposed which allow the design capture of computational primitives, in the form of logic and latches (i.e. in technologically independent form), physical assembly being directed by the designer's hierarchical 'typing'. A library of such elements may not only be moved between processes or technologies, but also expanded to include new elements with ease. The design, fabrication and test of a serial/parallel multiplier were reported, verifying some of the proposed serial-data architectures and exemplifying the approach of *SECOND*.

It is hoped that the ideas and techniques contained in this thesis may point the way ahead for researchers wishing to put the favourable properties of serial-data computation to use.

References

1. J. Allen, "Computer Architecture for Digital Signal Processing," *Proc. IEEE* **73** pp. 852 - 873 (May 1985).
2. K. Bromley and H. J. Whitehouse, "Signal Processing Technology Overview," *Proc. SPIE 298 - Real Time Signal Processing IV* pp. 102 - 106 (1981).
3. N. R. Powell and J. M. Irwin, "Signal Processing with Bit-Serial Word-Parallel Architectures," *Proc. SPIE 154 - Real Time Signal Processing* pp. 98 - 104 (1978).
4. P. B. Denyer and S. G. Smith, "Bit-Serial Architectures for Parallel Arrays," *Proc. SPIE 614, Highly Parallel Signal Processing Architectures* pp. 66 - 73 (Los Angeles, January 1986).
5. A. F. Murray, "Testing and Self-Testing," pp. 120 - 144 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
6. R. G. Bennetts, *Design of Testable Logic circuits*, Addison-Wesley (1984).
7. P. B. Denyer and D. Renshaw, *VLSI Signal Processing - A Bit-Serial Approach*, Addison-Wesley (1985).
8. S. P. Pope and R. W. Brodersen, "Macrocell Design for Concurrent Signal Processing," pp. 395 - 411 in *3rd Caltech Conf. on VLSI*, ed. R. Bryant, Computer Science Press (1983).
9. M. D. Ercegovic, "On-Line Arithmetic: an Overview," *Proc. SPIE 495 - Real Time Signal Processing VII* pp. 86 - 93 (1984).
10. K. S. Trivedi and M. D. Ercegovic, "On-Line Algorithms for Division and Multiplication," *Trans. IEEE C-26* pp. 681 - 687 (July 1977).
11. A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *Trans. IRE EC-10* pp. 389 - 400 (September 1961).
12. S. H. Unger, "A Computer Orientated Towards Spatial Problems," *Proc. IRE* **46** pp. 1744 - 1750 (1958).
13. K. E. Batcher, "Design of a Massively Parallel Processor," *Trans. IEEE C-29* pp. 836 - 840 (September 1980).
14. J. L. Potter, *The Massively Parallel Processor*, MIT Press (1985).

15. W. D. Hillis, *The Connection Machine*, MIT Press (1985).
16. P. M. Flanders et al., "Efficient High Speed Computing with the Distributed Array Processor," pp. 113 - 128 in *High Speed Computer and Algorithm Organisation*, ed. D. J. Kuck et al., Academic Press (1977).
17. M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE* **54** pp. 1901 - 1909 (December 1966).
18. R. M. Lea, "SCAPE: A Single-Chip Array Processing Element for Image Analysis," *Proc. VLSI '83*, pp. 285 - 294 (Trondheim, August 1983).
19. T. J. Fountain, "A Survey of Bit-Serial Array Processor Circuits," pp. 1 - 14 in *Computing Structures for Image Processing*, ed. M. J. B. Duff, Academic Press (1983).
20. D. K. Arvind, I. N. Robinson, and I. N. Parker, "A VLSI Chip for Real-Time Image Processing," *Proc. ISCS'83*, pp. 405 - 408 (Newport Beach, CA, May 1983).
21. T. Kondo et al., "An LSI Adaptive Array Processor," *J. IEEE SC-18* pp. 147 - 156 (April 1983).
22. J. M. Cotton and G. E. Masterson, "Computation of the Discrete Fourier Transform on the Cellular Array Processor," *Electrical Communication* **59** pp. 306 - 311 (June 1985).
23. R. Davis and D. Thomas, "Systolic Array Chip Matches the Pace of High-Speed Processing," *Electronic Design*, (October 31, 1984).
24. A. P. Reeves and J. D. Bruner, "Efficient Function Implementation for Bit-Serial Parallel Processors," *Trans. IEEE C-29* pp. 841 - 844 (September 1980).
25. R. F. Lyon, "MSSP: A Bit-Serial Multiprocessor for Signal Processing," pp. 263 - 276 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
26. L. B. Jackson, J. F. Kaiser, and H. S. McDonald, "An Approach to the Implementation of Digital Filters," *Trans. IEEE AU-16* pp. 413 - 421 (September 1968).
27. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall (1975).
28. P. F. Adams, J. R. Harbridge, and R. H. MacMillan, "An nMOS Integrated Circuit for Digital Filtering and Level Detection," *J. IEEE SC-16* pp. 183 - 190 (1981).
29. G. P. Edwards, P. J. Jennings, and T. Preston, "A MOS LSI Double Second Order Digital Filter Circuit," *Proc. IEEE ISSCC '75*, pp. 20 - 21 (February, 1975).

30. R. F. Lyon, "Filters: An Integrated Digital Filter Subsystem," pp. 253 - 262 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
31. D. J. Myers and P. A. Ivey, "STAR - A VLSI Architecture for Signal Processing," *Proc. MIT Conf. on Adv. Res. in VLSI*, pp. 179 - 183 (Cambridge, MA, January 1984).
32. N. R. Powell, "Functional Parallelism in VLSI Systems and Computations," pp. 41 - 49 in *VLSI Systems and Computations*, ed. H. T. Kung, R. F. Sproull & G. Steele Jr., Springer-Verlag (1981).
33. R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," pp. 131 - 140 in *VLSI 81*, ed. J. P. Gray, Academic Press (1981).
34. W. Chen et al., "Fault-Tolerant Wafer Scale Architectures Using Large Crossbar Switch Arrays," pp. 113 - 124 in *Wafer Scale Integration*, ed. C. R. Jesshope & W. R. Moore, Adam Hilger (1986).
35. S. L. Garverick and E. A. Pierce, "A Single Wafer 16-Point 16-MHz FFT Processor," *Proc. 1983 CICC*, (Rochester, NY, May 1983).
36. F. M. Rhodes, "Applications of RVLSI to Signal Processing," pp. 223 - 235 in *Wafer Scale Integration*, ed. C. R. Jesshope & W. R. Moore, Adam Hilger (1986).
37. R. W. Linderman et al., "CUSP: a 2- μ m CMOS Digital Signal Processor," *J. IEEE SC-20* pp. 761 - 769 (June 1985).
38. P. M. Chau and W. H. Ku, "A VLSI Floating-Point Signal Processor," pp. 293 - 305 in *VLSI Signal Processing, II*, ed. S. Y. Kung, R. E. Owen and J. G. Nash, IEEE Press (1986).
39. D. Cohen, "On Holy Wars and a Plea for Peace," *Computer* 14 pp. 49 - 54 (October 1981).
40. C. L. Seitz, "System Timing," pp. 218 - 262 in *Mead & Conway, "Introduction to VLSI Systems"*, Addison-Wesley (1980).
41. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).
42. N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley (1985).
43. M. S. McGregor, P. B. Denyer, and A. F. Murray, "A Single-Phase Clocking Scheme for CMOS VLSI," *Proc. 1987 VLSI Research Conf.*, (Palo Alto, CA, March 1987).

44. R. F. Lyon, "Two's Complement Pipeline Multipliers," *Trans. IEEE COM-24* pp. 418 - 425 (April 1976).
45. P. B. Denyer and D. J. Myers, "Carry-Save Adders for VLSI Signal Processing," pp. 151 - 160 in *VLSI '81*, ed. J. P. Gray, Academic Press (1981).
46. S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart and Winston (1982).
47. K. Hwang, *Computer Arithmetic Principles, Architecture, and Design*, Wiley (1979).
48. S. G. Smith and P. B. Denyer, "Efficient Bit-Serial Complex Multiplication and Sum-of-Products Computation Using Distributed Arithmetic," *Proc. IEEE-IECEJ-ASJ ICASSP'86*, pp. 2203 - 2206 (Tokyo, April 1986).
49. E. E. Swartzlander Jr., "Parallel Counters," *Trans. IEEE C-22* pp. 1021 - 1024 (December 1973).
50. A. R. Meo, "Arithmetic Networks and Their Minimisation Using a New Line of Elementary Units," *Trans. IEEE C-24* pp. 258 - 280 (March 1975).
51. O. Spaniol, *Computer Arithmetic*, Wiley (1981).
52. E. E. Swartzlander Jr. (ed.), *Computer Arithmetic*, Dowden, Hutchinson & Ross (1980).
53. S. Waser, "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing," *Computer* 11 pp. 19 - 29 (October 1978).
54. A. Habibi and P. A. Wintz, "Fast Multipliers," *Trans. IEEE C-19* pp. 153 - 157 (February 1970).
55. T. G. Hallin and M. J. Flynn, "Pipelining of Arithmetic Functions," *Trans. IEEE C-21* pp. 880 - 886 (August 1972).
56. J. R. Jump and S. R. Ahuja, "Effective Pipelining of Digital Systems," *Trans. IEEE C-27* pp. 855 - 865 (September 1978).
57. T. G. Noll et al., "A Pipelined 330 MHz Multiplier," *J. IEEE SC-21* pp. 411 - 416 (June 1986).
58. S. F. Anderson et al., "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Res. Develop.* 11 pp. 34-53 (January 1967).
59. L. P. Rubinfield, "A Proof of the Modified Booth's Algorithm for Multiplications," *Trans. IEEE C-24* pp. 1014 - 1015 (October 1975).
60. L. De Vos et al., "A Fast Adder-Based Multiplication Unit for Customised Digital Signal Processors," *Proc. IEEE-IECEJ-ASJ ICASSP'86*, pp. 2163 - 2166 (Tokyo, April 1986).

61. L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza* **34** pp. 349 - 356 (May 1965).
62. C. S. Wallace, "A Suggestion for a Fast Multiplier," *Trans. IEEE EC-13* pp. 14 - 17 (February 1964).
63. N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *Trans. IEEE C-34* pp. 789 - 796 (September 1985).
64. J. E. Vuillemin, "A Very Fast Multiplication Algorithm for VLSI Implementation," *Integration* **1** pp. 39 - 52 (April 1983).
65. S. D. Pezaris, "A 40-ns 17-Bit by 17-Bit Array Multiplier," *Trans. IEEE C-20* pp. 442 - 447 (April 1971).
66. J. V. McCanny and J. G. McWhirter, "Implementation of Signal Processing Functions Using 1-Bit Systolic Arrays," *Electronics Letters* **18** pp. 241 - 242 (March 18, 1982).
67. H. H. Guild, "Fully Iterative Fast Array for Binary Multiplication," *Electronics Letters* **5** p. 263 (June 12, 1969).
68. J. Deverell, "Pipeline Iterative Arithmetic Arrays," *Trans. IEEE C-24* pp. 317 - 322 (March 1975).
69. M. D. Ercegovac and J. G. Nash, "An Area-Time Efficient VLSI Design of a Radix-4 Multiplier," *Proc. IEEE ICCD '83*, pp. 684 - 687 (Port Chester, NY, 1983).
70. P. E. Danielsson, "Serial/Parallel Convolvers," *Trans. IEEE C-33* pp. 652 - 667 (July 1984).
71. S. A. White and T. Mitsutomi, "The IC Digital Filter: a New Low-Cost Signal Processing Tool," *Control Eng.* **43** pp. 58 - 68 (June 1970).
72. S. G. Smith, "Fourier Transform Machines," pp. 147 - 199, Chapter 8 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
73. A. F. Murray and P. B. Denyer, "A CMOS Design Strategy for Bit-Serial Signal Processing," *J. IEEE SC-20* pp. 746-753. (June 1985).
74. D. J. Myers and P. A. Ivey, "Circuit Elements for VLSI Signal Processing," *Br. Telecom Technol. J.* **2** pp. 67 - 77 (July 1984).
75. A. D. Booth, "A Signed Binary Multiplication Technique," *Q. J. Mech. Appl. Math.* **4** pp. 236 - 240 (1951).

76. C. F. N. Cowan, S. G. Smith, and J. H. Elliott, "A Digital Adaptive Filter using a Memory-Accumulator Architecture: Theory and Realisation," *Trans. IEEE ASSP-31* pp. 541 - 549 (June 1983).
77. J. T. Scanlon and W. K. Fuchs, "High-Performance Bit-Serial Multiplication," *Proc. IEEE ICCD'86*, (Rye Brook, NY, October 1986).
78. E. K. Cheng and C. A. Mead, "A Two's Complement Pipeline Multiplier," *Proc. IEEE ICASSP '76*, pp. 647 - 650 (Philadelphia, April 1976).
79. L. E. Turner, P. B. Denyer, and D. Renshaw, "A Bit Serial LDI Recursive Digital Filter," *Proc. IEEE ICASSP'84*, pp. 41A.3.1 - 41A.3.4 (San Diego, March 1984).
80. N. Petrie, J. Mavor, and S. G. Smith, "General-Purpose Adaptor Structure for Wave-Digital-Filter Realisation," *Electronics Letters* **19** pp. 1038 - 1039 (November 24, 1983).
81. P. R. Cappello and K. Steiglitz, "A Note on 'Free Accumulation' in VLSI Filter Architectures," *Trans. IEEE CAS-32* pp. 291 - 296 (March 1985).
82. C. A. Mead and M. Rem, "Minimum Propagation Delays in VLSI," *J. IEEE SC-17* pp. 773 - 775 (August 1982).
83. H. T. Kung, "Special-Purpose Devices for Signal and Image Processing: an Opportunity in Very Large Scale Integration (VLSI)," *SPIE Real-Time Signal Processing III* **241** pp. 76 - 84 (1980).
84. G. H. Allen, P. B. Denyer, and D. Renshaw, "A Bit-Serial Linear Array DFT," *Proc. IEEE ICASSP'84*, pp. 41A.1.1 - 41A.1.4 (San Diego, March 1984).
85. H. T. Kung, "Why Systolic Architectures?," *Computer* **15** pp. 37 - 46 (January 1982).
86. D. Cohen, "Mathematical Approach to Iterative Computational Networks," *Proc. 4th IEEE Symp. on Computer Arith.*, pp. 226 - 238 (Santa Monica, CA, October 1978).
87. R. V. Donthi, M. Saleem, and H. Singh, "On Bit-Sequential Multipliers," *Proc. 6th IEEE Symp. on Computer Arith.*, pp. 104 - 108 (1983).
88. R. Gnanasekaran, "A Fast Serial-Parallel Multiplier," *Trans. IEEE C-34* pp. 741 - 745 (August 1985).
89. J. Mick, J. Springer, and C. Ghest, "A High-Speed Serial/Parallel Multiplier: the Am25LS14," pp. 2-15 - 2-23 in *Bipolar Microprocessor and Logic Interface Data Book*, Advanced Micro Devices (1982).

90. A. J. Atrubin, "A One-Dimensional Real-Time Iterative Multiplier," *Trans. IEEE EC-14* pp. 394 - 399 (June 1965).
91. D. E. Knuth, *The Art of Computer Programming Vol. 2 / Seminumerical Algorithms*, Section 4.3.3.E, Addison-Wesley (1969).
92. G. L. Baldwin et al., "A Modular High-Speed Serial Pipeline Multiplier for Digital Signal Processing," *J. IEEE SC-13* pp. 400 - 408 (June 1978).
93. D. Hampel, K. E. McGuire, and K. J. Prost, "CMOS/SOS Serial-Parallel Multiplier," *J. IEEE SC-10* pp. 307 - 314 (October 1975).
94. J. Kane, "A Low-Power, Bipolar, Two's Complement Serial Pipeline Multiplier Chip," *J. IEEE SC-11* pp. 669 - 678 (October 1976).
95. J. R. Verjans, "A Serial-Parallel Multiplier Using the NENDEP Technology," *J. IEEE SC-12* pp. 323 - 325 (June 1977).
96. E. E. Swartzlander Jr., "The Quasi-Serial Multiplier," *Trans. IEEE C-22* pp. 317 - 321 (April 1973).
97. T. G. McDanel and R. K. Guha, "The Two's Complement Quasi-Serial Multiplier," *Trans. IEEE C-24* pp. 1233 - 1235 (December 1975).
98. C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *Trans. IEEE C-22* pp. 1045 - 1047 (December 1973).
99. I.-N. Chen and R. Willoner, "An $O(n)$ Parallel Multiplier with Bit-Sequential Input and Output," *Trans. IEEE C-28* pp. 721 - 727 (October 1979).
100. H. J. Sips, "Comments on 'An $O(n)$ Parallel Multiplier with Bit-Sequential Input and Output'," *Trans. IEEE C-31* pp. 325 - 327 (April 1982).
101. N. R. Strader and V. T. Rhyne, "A Canonical Bit-Sequential Multiplier," *Trans. IEEE C-31* pp. 791 - 795 (August 1982).
102. T. Rhyne and N. R. Strader, "A Signed Bit-Sequential Multiplier," *Trans. IEEE C-35* pp. 896 - 901 (October 1986).
103. R. Gnanasekaran, "On a Bit-Serial Input and Bit-Serial Output Multiplier," *Trans. IEEE C-32* pp. 878 - 880 (September 1983).
104. M. R. Buric and C. A. Mead, "Bit-Serial Inner Product Processors in VLSI," *Proc. 2nd Caltech Conf. on VLSI*, pp. 155 - 164 (1981).
105. H. J. Sips, "Bit-Sequential Arithmetic for Parallel Processors," *Trans. IEEE C-33* pp. 7 - 21 (January 1984).
106. E. L. Braun, *Digital Computer Design*, McGraw-Hill (1962).

107. S. G. Smith, "Comments on 'A Signed Bit-Sequential Multiplier'," *submitted to IEEE Trans. Comput.*, (November 1986).
108. S. G. Smith, "Incremental Computation of Squares and Sums of Squares," *submitted to IEEE Trans. Comput.*, (November 1986).
109. J. Wawrzynek and C. A. Mead, "A Bit-Serial Architecture for Sound Synthesis," pp. 277 - 297 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
110. G. E. Moore, "VLSI: Some Fundamental Challenges," *IEEE Spectrum* 16 pp. 30 - 37 (April 1979).
111. R. F. Ayers, *VLSI Silicon Compilation and the Art of Automatic Chip Design*, Prentice-Hall (1983).
112. S. G. Smith, "Silicon Compilers - Design Synthesis Beyond CAD," pp. 135 - 147, Chapter 11 in *Computer-Aided Tools for VLSI Systems Design*, ed. G. Russell, Peter Peregrinus (1987).
113. K. Asada and J. Mavor, "Area Optimised MOS Circuit Generation Using the Circuit Synthesis Program MOSYN-2," *Proc. ESSCIRC '86*, pp. 71 - 73 (Delft, September 1986).
114. S. G. Smith et al., "Full-Span Silicon Compilation of Digital Signal Processors," *Proc. IEEE ICASSP'87*, (Dallas, TX, April 1987).
115. I. Buchanan, *Modelling and Verification in Structured Integrated Circuit Design*, Ph.D. Thesis, University of Edinburgh (1980).
116. N. W. Bergmann, "A Case Study of the FIRST Silicon Compiler," pp. 413 - 430 in *3rd Caltech Conference on VLSI*, ed. R. Bryant, Computer Science Press (1983).
117. P. B. Denyer, D. Renshaw, and N. Bergmann, "A Silicon Compiler for VLSI Signal Processors," *Proc. ESSCIRC'82*, pp. 215 - 218 (Brussel, September 1982).
118. S. G. Smith et al., "A Comparison of Micro-DSP and Silicon Compiler Implementations of a Polyphase-Network Filter Bank," *Proc. IEEE-IECEJ-ASJ ICASSP'86*, pp. 2207 - 2210 (Tokyo, April 1986).
119. R. K. Brayton et al., "The YORKTOWN Silicon Compiler," *Proc. ISCAS '85*, pp. 391 - 394 (Kyoto, June 1985).
120. A. D. Lopez and H.-F. S. Law, "A Dense Gate-Matrix Layout Method for MOS VLSI," *Trans. IEEE ED-27* pp. 1671 - 1675 (August 1980).
121. R. K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers (1984).

122. A. R. Newton and A. L. Sangiovanni-Vincentelli, "Computer-Aided Design for VLSI Circuits," *Computer* **19** pp. 38 - 60 (April 1986).
123. S. G. Smith, "Transversal Filters," pp. 200 - 235, Chapter 9 in *P. B. Denyer & D. Renshaw, "VLSI Signal Processing - A Bit-Serial Approach"*, Addison-Wesley (1985).
124. M. G. Bellanger, G. Bonnerot, and M. Coudreuse, "Digital Filtering by Polyphase Network: Application to Sample-Rate Alteration and Filter Banks," *Trans. IEEE ASSP-24* pp. 109 - 114 (April 1976).
125. U. Heute and P. Vary, "A Digital Filter Bank with Polyphase Network and FFT Hardware: Measurements and Applications," *Signal Processing* **3** pp. 307 - 319 (October 1981).
126. R. W. Schafer and L. R. Rabiner, "Design of Digital Filter Banks for Speech Analysis," *Bell Syst. Tech. J.* **50** pp. 3097 - 3115 (December 1971).
127. S. L. Freeny et al., "Design of Digital Filters for an All Digital Frequency Division Multiplex-Time Division Multiplex Translator," *Trans. IEEE CT-18* pp. 702 - 711 (November 1971).
128. R. Jain et al., "Custom Design of a VLSI PCM-FDM Transmultiplexer from System Specifications to Circuit Layout Using a Computer-Aided Design System," *J. IEEE SC-21* pp. 73 - 85 (February 1986).
129. A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, Prentice-Hall (1975).
130. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comp.* **19** pp. 297 - 301 (April 1965).
131. E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall (1974).
132. J. Deverell, "Multiplication of Complex Numbers Using Iterative Arrays," *Electronics Letters* **7** pp. 205 - 207 (6th May, 1971).
133. J. Fox et al., "A Self Testing 2 Micron CMOS Chip Set for FFT Applications," *Proc. ESSCIRC '85*, pp. 13 - 24 (Toulouse, September 1985).
134. A. Greiner et al., "A Flexible High Performance Serial Radix-2 FFT Butterfly Unit," *Proc ESSCIRC '85*, pp. 25 - 28 (Toulouse, September 1985).
135. A. M. Despain, "Fourier Transform Computers using CORDIC Iterations," *Trans. IEEE C-23* pp. 993 - 1001 (October 1974).
136. P. S. Moharir, "Extending the Scope of Golub's Method Beyond Complex Multiplication," *Trans. IEEE C-34* pp. 484 - 487 (May 1985).

137. E. E. Swartzlander Jr., "Merged Arithmetic," *Trans. IEEE C-29* pp. 946 - 950 (October 1980).
138. R. F. Eschenbach and B. M. Oliver, "An Efficient Coordinate Rotation Algorithm," *Trans. IEEE C-27* pp. 1178 - 1180 (December 1978).
139. J. E. Volder, "The CORDIC Trigonometric Computing Technique," *Trans. IRE EC-8* pp. 330 - 334 (August 1959).
140. E. D. Deprettere, P. Dewilde, and R. Udo, "Pipelined Cordic Architectures for Fast VLSI Filtering and Array Processing," *Proc. IEEE ICASSP'84*, pp. 41A.6.1 - 41A.6.4 (San Diego, March 1984).
141. H. M. Ahmed, J. -M. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer* **15** pp. 65 - 82 (January 1982).
142. D. T. L. Lee and M. Morf, "Generalised CORDIC for Digital Signal Processing," *Proc. IEEE ICASSP'82*, pp. 1748 - 1751 (Paris, May 1982).
143. S. A. White, "A Simple FFT Butterfly Arithmetic Unit," *Trans. IEEE CAS-28* pp. 352 - 355 (April 1981).
144. I. R. Mactaggart and M. A. Jack, "A Single Chip Radix-2 FFT Butterfly Architecture Using Parallel Data Distributed Arithmetic," *J. IEEE SC-19* pp. 368 - 373 (June 1984).
145. J. H. McClellan and R. J. Purdy, "Applications of Digital Signal Processing to Radar," pp. 239 - 329 in *Applications of Digital Signal Processing*, ed. A. V. Oppenheim, Prentice-Hall (1978).
146. E. E. Swartzlander Jr. and G. Hallnor, "Fast Transform Processor Implementation," *Proc. IEEE ICASSP'84*, pp. 25A.5.1 - 25A.5.4 (San Diego, March 1984).
147. H. S. Stone, "Parallel Processing with the Perfect Shuffle," *Trans. IEEE C-20* pp. 153 - 161 (February 1971).
148. H. C. Rickers and P. F. Manno, "Microprocessor and LSI Microcircuit Reliability-Prediction Model," *Trans. IEEE R-29* pp. 196 - 202 (August 1980).
149. S. G. Smith and P. B. Denyer, "Serial/Parallel Architectures for Area-Efficient Vector Multiplication," *Proc. IEEE ICASSP'87*, (Dallas, TX, April 1987).
150. S. G. Smith and P. B. Denyer, "Synthesis of Area-Efficient VLSI Architectures for Vector and Matrix Multiplication," *Proc. 8th IEEE Symp. on Computer Arith.*, (Como, Italy, May 1987).
151. E. E. Swartzlander Jr., B. K. Gilbert, and I. S. Reed, "Inner Product Computers," *Trans. IEEE C-27* pp. 21 - 31 (January 1978).

152. A. Croisier et al., *Digital Filter for PCM Encoded Signals*, U.S. Patent 3 777 130 (December 4, 1973).
153. A. Peled and B. Liu, "A New Hardware Realisation of Digital Filters," *Trans. IEEE ASSP-22* pp. 456 - 462 (December 1974).
154. P. R. Cappello and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," pp. 245 - 254 in *VLSI Systems and Computations*, ed. H. T. Kung, R. F. Sproull & G. Steele Jr., Springer-Verlag (1981).
155. J. G. McWhirter, J. V. McCanny, and K. W. Wood, "Novel Multibit Convolver/Correlator Chip based on Systolic Array Principles," *SPIE Real-Time Signal Processing V 341* pp. 66 - 73 (1982).
156. J. C. White et al., "A High-Speed CMOS/SOS Implementation of a Bit Level Systolic Correlator," *Proc. IEEE-IECEJ-ASJ ICASSP'86*, pp. 1161 - 1164 (Tokyo, April 1986).
157. S. G. Smith, "Efficient Serial/Parallel Inner-Product Computation," *Electronics Letters 22* pp. 750 - 752 (July 3, 1986).
158. S. A. White, "On Mechanization of Vector Multiplication," *Proc. IEEE 63* pp. 730 - 731 (April 1975).
159. S. K. Rao and T. Kailath, "Orthogonal Digital Filters for VLSI Implementation," *Trans. IEEE CAS-31* pp. 933 - 945 (November 1984).
160. S. G. Smith, "Modelling Musical Instruments in the Digital Domain," *Proc. IEEE ICASSP'84*, pp. 19.7.1 - 19.7.4 (San Diego, March 1984).
161. S. G. Smith, "Serial/Parallel Modules for Complex Arithmetic," *Electronics Letters 22* pp. 1256 - 1257 (November 6, 1986).
162. S. G. Smith, M. S. McGregor, and P. B. Denyer, "Techniques to Increase the Computational Throughput of Bit-Serial Architectures," *Proc. IEEE ICASSP'87*, (Dallas, TX, April 1987).
163. C. S. Burrus, "Digital Filter Structures Described by Distributed Arithmetic," *Trans. IEEE CAS-24* pp. 674 - 680 (December 1977).
164. T. A. C. M. Claasen, W. F. G. Mecklenbrauker, and J. B. H. Peek, "Some Considerations on the Implementation of Digital Systems for Signal Processing," *Philips Res. Repts. 30* pp. 73 - 84 (1975).
165. H. J. De Man, C. J. Vandebulcke, and M. M. Van Cappellen, "High-Speed NMOS Circuits for ROM-Accumulator and Multiplier Type Digital Filters," *J. IEEE SC-13* pp. 565 - 572 (October 1978).
166. G. L. Turin, "An Introduction to Digital Matched Filters," *Proc. IEEE 64* pp. 1092 - 1112 (July 1976).

167. C. F. N. Cowan and P. M. Grant, *Adaptive Filters*, Prentice Hall (1985).
168. M. D. Ercegovic, "An On-Line Square Rooting Algorithm," *Proc. 4th IEEE Symp. on Computer Arith.*, pp. 183 - 189 (Santa Monica, CA, October 1978).
169. V. G. Oklobdzija and M. D. Ercegovic, "An On-line Square Root Algorithm," *Trans. IEEE C-31* pp. 70 - 75 (January 1982).
170. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).
171. C. Terman, *Logic Level Simulation for VLSI Circuits*, Kluwer Academic Publishers (to appear).
172. M. S. McGregor, S. G. Smith, P. B. Denyer, and A. F. Murray, "Serial-Data Computation on Twin Pipelines," *submitted to Electronics Letters*, (December 1986).
173. S. G. Smith and P. B. Denyer, "Radix-4 Modules for High-Performance Bit-Serial Computation," *submitted to Proc. IEE Part E*, (October 1986).
174. R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Crypto-Systems," *C. ACM* 21 pp. 121 - 126 (February 1978).
175. S. G. Smith, "The Serial/Parallel Automultiplier," *submitted to Electronics Letters*, (January 1987).
176. P. B. Denyer, "System Compilation," pp. 3 - 13 in *VLSI Signal Processing, II*, ed. S. Y. Kung, R. E. Owen and J. G. Nash, IEEE Press (1986).
177. J. H. Nash and S. G. Smith, "A Front End Graphic Interface to the FIRST Silicon Compiler," *Proc. IEE Electronic Design Automation (EDA84)*, pp. 120 - 124 (Warwick, UK, March 1984).
178. J. P. Gray, I. Buchanan, and P. S. Robertson, "Designing Gate Arrays Using a Silicon Compiler," *Proc. 19th DA Conf.*, pp. 377 - 383 (Las Vegas, 1982).
179. P. Six, I. Vandeweerd, and H. De Man, "An Interactive Environment for Creating Module Generators," *Proc. ESSCIRC'86*, pp. 65 - 67 (Delft, September 1986).
180. J. Rabaey, S. P. Pope, and R. W. Brodersen, "An Integrated Automatic Layout Generation System for DSP Circuits," *Trans. IEEE CAD-4* pp. 285 - 296 (July 1985).
181. R. Jain, P. A. Ruetz, and R. W. Brodersen, "Architectural Strategies for Digital Signal Processing Circuits," pp. 361 - 372 in *VLSI Signal Processing, II*, ed. S. Y. Kung, R. E. Owen and J. G. Nash, IEEE Press (1986).

182. J. R. Jasica, S. Noujaim, and R. Hartley, "A Bit-Serial Silicon Compiler," *Proc. ICCAD'85*, pp. 91 - 93 (Santa Clara, November 1985).
183. M. R. Buric and T. G. Matheson, "Silicon Compilation Environments," *Proc. IEEE CICC*, pp. 208 - 212 (Portland, OR, May 1985).
184. C. Piguet, E. Dijkstra, and G. Berweiler, "Automatic Generation of CMOS Layout Cells under Topological Constraints," *Proc. ESSCIRC'86*, pp. 68 - 70 (Delft, September 1986).
185. P. W. Kollarisch and N. H. E. Weste, "TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout," *J. IEEE SC-20* pp. 799 - 804 (June 1985).

Appendix A

FIRST Description of the Soft Model

! FIR section for PPN filter

!

OPERATOR FilterSection [swl,cofres,muxlevel] (c1, ev -> clout, evout) -
datain, cof -> dataout, multout

SIGNAL datadel, cofsel, cofdelay
CONTROL c11

CONSTANT multdel = ((3 * cofres) / 2) + 2

MULTIPLY [1,cofres,1,0] (c11 -> clout) datain, cofsel -> multout, datadel
BITDELAY [(swl*muxlevel)-(multdel+1)] datadel -> dataout
MULTIPLEX [1,0,0] (ev) cofdelay, cof -> cofsel
BITDELAY [(swl*muxlevel)-1] cofsel -> cofdelay

CBITDELAY [1] (c1 -> c11)
CBITDELAY [swl*muxlevel] (ev -> evout)

END

!

!

OPERATOR FilterCascade [swl,cofres,muxlevel] (c1, ev -> clout) datain,
cof -> proda0 THROUGH 5, prodb0 THROUGH 5

SIGNAL multout, dataout
CONTROL cldummy, evout

FilterSection [swl,cofres,muxlevel] (c1, ev -> cldummy, evout) -
datain, cof -> dataout, multout TIMES 12 WITH -
(evout -> ev) dataout -> datain
multout = proda0, prodb0, proda1, prodb1, proda2, prodb2,
proda3, prodb3, proda4, prodb4, proda5, prodb5
cdummy = clout, NC, NC, NC, NC, NC, NC, NC, NC, NC, NC, NC

END

!

!

OPERATOR Adder [del] (c1) in0 THROUGH 5 -> out

SIGNAL s0 THROUGH 3
CONTROL c11, c12

ADD [1,0,0,0] (c1) in0, in1, GND -> s0, NC
ADD [1,0,0,0] (c1) in2, in3, GND -> s1, NC

```

ADD [1,0,0,0] (c1) in4, in5, GND -> s2, NC
ADD [1,0,0,0] (c11) s0, s1, GND -> s3, NC
ADD [del-2,0,1,0] (c12) s3, s2, GND -> out, NC

CBITDELAY [1] (c1 -> c11)
CBITDELAY [1] (c11 -> c12)

END
!=====
!=====
OPERATOR FilterAdd [swl,cofres,muxlevel,del] (c1, ev) datain, cof -> -
    out0, out1

    SIGNAL proda0 THROUGH 5, prodb0 THROUGH 5
    CONTROL clout

    FilterCascade [swl,cofres,muxlevel] (c1, ev -> clout) datain,
        cof -> proda0 THROUGH 5, prodb0 THROUGH 5
    Adder [del] (clout) proda0 THROUGH 5 -> out0
    Adder [del] (clout) prodb0 THROUGH 5 -> out1

END
!=====
!=====
CHIP FilterOut (pcl, pev) pdat0, pdat1, pcof0, pcof1 -> pout0 THROUGH 3

    SIGNAL dat0, dat1, cof0, cof1, out0 THROUGH 3
    CONTROL cl, ev

    PADIN (pcl, pev -> cl, ev)
    PADIN pdat0, pdat1, pcof0, pcof1 -> dat0, dat1, cof0, cof1
    PADOUT out0 THROUGH 3 -> pout0 THROUGH 3
    PADORDER VDD, pcl, pev, pdat0, pdat1, pcof0, pcof1,
        pout0 THROUGH 3, GND, CLOCK
    CONSTANT swl = 16, cofres = 8, muxlevel = 16
    CONSTANT multdel = ((3 * cofres) / 2) + 2
    CONSTANT del = (2 * swl) - (multdel + 2)

    FilterAdd [swl,cofres,muxlevel,del] (c1, ev) dat0, cof0 -> -
        out0, out1
    FilterAdd [swl,cofres,muxlevel,del] (c1, ev) dat1, cof1 -> -
        out2, out3

END
!=====
!=====
OPERATOR Dft2 [del] (c1) rin0,rin1,iin0,iin1 -> rout0,rout1,
iout0,iout1

    ADD [del,0,0,0] (c1) rin0,rin1,GND -> rout0,NC
    SUBTRACT [del,0,0,0] (c1) rin0,rin1,GND -> rout1,NC
    ADD [del,0,0,0] (c1) iin0,iin1,GND -> iout0,NC

```



```
SUBTRACT [del,0,0,0] (c1) iin0,iin1,GND -> iout1,NC

END
=====
=====
OPERATOR Dft4 [sw1] (c1) rin0,rin1,rin2,rin3,iin0,iin1,iin2,iin3 -> -
rout0,rout1,rout2,rout3,iout0,iout1,iout2,iout3

SIGNAL r0,r1,r2,r3,i0,i1,i2,i3
CONTROL c11

Dft2 [sw1/2] (c1) rin0,rin2,iin0,iin2 -> r0,r1,i0,i1
Dft2 [sw1/2] (c1) rin1,rin3,iin1,iin3 -> r2,r3,i2,i3
Dft2 [sw1-(sw1/2)] (c11) r0,r2,i0,i2 -> rout0,rout2,iout0,iout2
Dft2 [sw1-(sw1/2)] (c11) r1,r3,i1,r3 -> rout1,rout3,iout3,iout1

CBITDELAY [sw1/2] (c1 -> c11)

END
=====
=====
OPERATOR CmplxMul [sw1,co] (c1) rin,iin,rcof,icof -> rout,iout

SIGNAL s0,s1,s2,s3
CONTROL c11

CONSTANT multdel = ((3 * co) / 2) + 2

MULTIPLY [1,co,0,0] (c1 -> c11) rin,rcof -> s0,NC
MULTIPLY [1,co,0,0] (c1 -> NC) iin,icof -> s1,NC
MULTIPLY [1,co,0,0] (c1 -> NC) rin,icof -> s2,NC
MULTIPLY [1,co,0,0] (c1 -> NC) iin,rcof -> s3,NC
SUBTRACT [(2*sw1)-multdel,0,0,0] (c11) s0,s1,GND -> rout,NC
ADD [(2*sw1)-multdel,0,0,0] (c11) s2,s3,GND -> iout,NC

END
=====
=====
OPERATOR Twiddle [sw1,co] (c1) rin0,rin1,rin2,rin3,iin0,iin1,iin2,iin3,
rcof1,rcof2,rcof3,icof1,icof2,icof3 -> rout0,rout1,rout2,rout3,
iout0,iout1,iout2,iout3

BITDELAY [2*sw1] rin0 -> rout0
BITDELAY [2*sw1] iin0 -> iout0
CmplxMul [sw1,co] (c1) rin1,iin1,rcof1,icof1 -> rout1,iout1
CmplxMul [sw1,co] (c1) rin2,iin2,rcof2,icof2 -> rout2,iout2
CmplxMul [sw1,co] (c1) rin3,iin3,rcof3,icof3 -> rout3,iout3

END
=====
=====
```

```
OPERATOR ComReal [swl] (cn,cnplus1,cnplus1del) in0,in1,in2,in3 -> -
out0,out1,out2,out3
```

```
SIGNAL r0,r1,r2,r3
```

```
MULTIPLEX [swl/2,0,0] (cnplus1) in2,in0 -> r0
MULTIPLEX [swl/2,0,0] (cnplus1del) in3,in1 -> r1
MULTIPLEX [swl/2,0,0] (cnplus1) in0,in2 -> r2
MULTIPLEX [swl/2,0,0] (cnplus1del) in1,in3 -> r3
MULTIPLEX [swl-(swl/2),0,0] (cn) r1,r0 -> out0
MULTIPLEX [swl-(swl/2),0,0] (cn) r0,r1 -> out1
MULTIPLEX [swl-(swl/2),0,0] (cn) r3,r2 -> out2
MULTIPLEX [swl-(swl/2),0,0] (cn) r2,r3 -> out3
```

```
END
```

```
=====
```

```
OPERATOR Commutator [swl] (cn,cnplus1,cnplus1del) rin0,rin1,rin2,rin3,
iin0,iin1,iin2,iin3 -> rout0,rout1,rout2,rout3,iout0,iout1,iout2,iout3
```

```
ComReal [swl] (cn,cnplus1,cnplus1del) rin0,rin1,rin2,rin3 -> -
rout0,rout1,rout2,rout3
```

```
ComReal [swl] (cn,cnplus1,cnplus1del) iin0,iin1,iin2,iin3 -> -
iout0,iout1,iout2,iout3
```

```
END
```

```
=====
```

```
OPERATOR LineDel [swl,len] rin1,rin2,rin3,iin1,iin2,iin3 -> -
rout1,rout2,rout3,iout1,iout2,iout3
```

```
BITDELAY [swl*len] rin1 -> rout1
BITDELAY [2*swl*len] rin2 -> rout2
BITDELAY [3*swl*len] rin3 -> rout3
BITDELAY [swl*len] iin1 -> iout1
BITDELAY [2*swl*len] iin2 -> iout2
BITDELAY [3*swl*len] iin3 -> iout3
```

```
END
```

```
=====
```

```
CHIP Pipe (pc1,pc2,pc3,pc4,pc5) prin0,prin1,prin2,prin3,
prcof1,prcof2,prcof3,picof1,picof2,picof3,prcof11,prcof12,prcof13,
picof11,picof12,picof13 -> prout0,prout1,prout2,prout3,piout0,piout1,piout2,piout3
```

```
SIGNAL rin0,rin1,rin2,rin3,rout0,rout1,rout2,rout3,iout0,iout1,iout2,iout3
SIGNAL rcof1,rcof2,rcof3,icof1,icof2,icof3,rcof11,rcof12,rcof13,
icof11,icof12,icof13
SIGNAL r0,r1,r2,r3,i0,i1,i2,i3
SIGNAL r10,r11,r12,r13,i10,i11,i12,i13
SIGNAL r20,r21,r22,r23,i20,i21,i22,i23
SIGNAL r30,r31,r32,r33,i30,i31,i32,i33
SIGNAL r40,r41,r42,r43,i40,i41,i42,i43
SIGNAL r50,r51,r52,r53,i50,i51,i52,i53
```

```

SIGNAL r60,r61,r62,r63,i60,i61,i62,i63
SIGNAL r70,r71,r72,r73,i70,i71,i72,i73
SIGNAL r80,r81,r82,r83,i80,i81,i82,i83
SIGNAL r90,r91,r92,r93,i90,i91,i92,i93
CONTROL c1,c2,c3,c4,c5,c3del,c5del, c2a, c3a, c4a, c5a

PADIN (pc1,pc2,pc3,pc4,pc5 -> c1,c2,c3,c4,c5)
PADIN prin0,prin1,prin2,prin3 -> rin0,rin1,rin2,rin3
PADIN prcof1,prcof2,prcof3,picof1,picof2,picof3,prcof11,prcof12,prcof13,
picof11,picof12,picof13 -> rcof1,rcof2,rcof3,icof1,icof2,icof3,
rcof11,rcof12,rcof13,icof11,icof12,icof13
PADOUT rout0,rout1,rout2,rout3,iout0,iout1,iout2,iout3 -> prout0,prout1,prout2,pro
piout0,piout1,piout2,piout3
PADORDER VDD,pc5,pc1,pc2,pc3,pc4,prin0,prin1,prin2,prin3,
GND,CLOCK,prout0,prout1,prout2,prout3,piout0,piout1,piout2,piout3,
prcof1,prcof2,prcof3,picof1,picof2,picof3,prcof11,prcof12,prcof13,
picof11,picof12,picof13

CONSTANT swl = 16,co = 12

Dft4 [swl] (c1) rin0,rin1,rin2,rin3,GND, GND, GND, GND -> -
r0,r1,r2,r3,i0,i1,i2,i3
LineDel [swl,4] r1,r2,r3,i1,i2,i3 -> -
r11,r12,r13,i11,i12,i13
Commutator [swl] (c4a,c5a,c5del) r0,r11,r12,r13,i0,
i11,i12,i13.-> r20,r21,r22,r23,i20,i21,i22,i23
LineDel [swl,4] r22,r21,r20,i22,i21,i20 -> -
r32,r31,r30,i32,i31,i30
Twiddle [swl,co] (c1) r30,r31,r32,r23,i30,i31,i32,i23,
rcof1,rcof2,rcof3,icof1,icof2,icof3 -> r40,r41,r42,r43,
i40,i41,i42,i43
Dft4 [swl] (c1) r40,r41,r42,r43,i40,i41,i42,i43 -> -
r50,r51,r52,r53,i50,i51,i52,i53
LineDel [swl,1] r51,r52,r53,i51,i52,i53 -> -
r61,r62,r63,i61,i62,i63
Commutator [swl] (c2a,c3a,c3del) r50,r61,r62,r63,i50,
i61,i62,i63 -> r70,r71,r72,r73,i70,i71,i72,i73
LineDel [swl,1] r72,r71,r70,i72,i71,i70 -> -
r82,r81,r80,i82,i81,i80
Twiddle [swl,co] (c1) r80,r81,r82,r73,i80,i81,i82,i73,
rcof11,rcof12,rcof13,icof11,icof12,icof13 -> r90,r91,r92,r93,
i90,i91,i92,i93
Dft4 [swl] (c1) r90,r91,r92,r93,i90,i91,i92,i93 -> -
rout0,rout1,rout2,rout3,iout0,iout1,iout2,iout3
CBITDELAY [swl+(swl/2)] (c2 -> c2a)
CBITDELAY [3*swl] (c3 -> c3a)
CBITDELAY [swl] (c3a -> c3del)
CBITDELAY [3*swl+(swl/2)] (c4 -> c4a)
CBITDELAY [3*swl] (c5 -> c5a)
CBITDELAY [4*swl] (c5a -> c5del)

```

END

```
=====
!
OPERATOR CofSource [n,swl] (c1, e2) in -> out
! basic component of circular ROM, with associated read-only register
! latency is swl
```

SIGNAL const

```
CONSTGEN [swl,n] (c1) -> const
MULTIPLEX [swl-1,1,0] (e2) in, const -> out
```

END

```
=====
!
!
=====
OPERATOR MuxCofs [a0 THROUGH 15, swl] (c1, e2 -> ) -> sout1, sout2
! circular ROM with outputs arranged for the 2 stages of a 64-pt FFT
```

SIGNAL s0 THROUGH 15

```
CofSource [a0,swl] (c1, e2) s1 -> s0
CofSource [a1,swl] (c1, e2) s2 -> s1
CofSource [a2,swl] (c1, e2) s3 -> s2
CofSource [a3,swl] (c1, e2) s4 -> s3
CofSource [a4,swl] (c1, e2) s5 -> s4
CofSource [a5,swl] (c1, e2) s6 -> s5
CofSource [a6,swl] (c1, e2) s7 -> s6
CofSource [a7,swl] (c1, e2) s8 -> s7
CofSource [a8,swl] (c1, e2) s9 -> s8
CofSource [a9,swl] (c1, e2) s10 -> s9
CofSource [a10,swl] (c1, e2) s11 -> s10
CofSource [a11,swl] (c1, e2) s12 -> s11
CofSource [a12,swl] (c1, e2) s13 -> s12
CofSource [a13,swl] (c1, e2) s14 -> s13
CofSource [a14,swl] (c1, e2) s15 -> s14
CofSource [a15,swl] (c1, e2) s0 -> s15
! tap out at correct points to line up coeffs with RP at start of transform
! BITDELAY [swl-1] s14 -> sout1
! BITDELAY [swl-1] s10 -> sout2
! BITDELAY [swl-1] s2 -> sout1
! BITDELAY [swl-1] s11 -> sout2
```

END

```
=====
!
!
=====
CHIP FftCofs (pc1, pe2) -> ps1t1r, ps1t1i, ps1t2r, ps1t2i,
ps1t3r, ps1t3i, ps2t1r, ps2t1i, ps2t2r, ps2t2i, ps2t3r, ps2t3i

SIGNAL s1t1r, s1t1i, s1t2r, s1t2i, s1t3r, s1t3i, s2t1r, s2t1i,
s2t2r, s2t2i, s2t3r, s2t3i
CONTROL c1, e2, e2a
```

```
PADIN (pcl, pe2 -> c1, e2)
PADOUT slt1r, slt1i, slt2r, slt2i, slt3r, slt3i, s2t1r, s2t1i,
      s2t2r, s2t2i, s2t3r, s2t3i -> pslt1r, pslt1i, pslt2r, pslt2i,
      pslt3r, pslt3i, ps2t1r, ps2t1i, ps2t2r, ps2t2i, ps2t3r, ps2t3i
PADORDER VDD, pcl, pe2, GND, CLOCK, pslt1r, pslt1i, pslt2r, pslt2i,
      pslt3r, pslt3i, ps2t1r, ps2t1i, ps2t2r, ps2t2i, ps2t3r, ps2t3i
```

```
CONSTANT swl = 16
CONSTANT r0 = 2047, i0 = 0
CONSTANT r1 = 2038, i1 = 65335
CONSTANT r2 = 2009, i2 = 65136
CONSTANT r3 = 1960, i3 = 64941
CONSTANT r4 = 1892, i4 = 64752
CONSTANT r5 = 1806, i5 = 64571
CONSTANT r6 = 1703, i6 = 64398
CONSTANT r7 = 1583, i7 = 64237
CONSTANT r8 = 1448, i8 = 64088
CONSTANT r9 = 1299, i9 = 63953
CONSTANT r10 = 1138, i10 = 63833
CONSTANT r11 = 965, i11 = 63730
CONSTANT r12 = 784, i12 = 63644
CONSTANT r13 = 595, i13 = 63576
CONSTANT r14 = 400, i14 = 63527
CONSTANT r15 = 201, i15 = 63498
CONSTANT r16 = 0, i16 = 63488
CONSTANT r18 = 65136, i18 = 63527
CONSTANT r20 = 64752, i20 = 63644
CONSTANT r21 = 64571, i21 = 63730
CONSTANT r22 = 64398, i22 = 63833
CONSTANT r24 = 64088, i24 = 64088
CONSTANT r26 = 63833, i26 = 64398
CONSTANT r27 = 63730, i27 = 64571
CONSTANT r28 = 63644, i28 = 64752
CONSTANT r30 = 63527, i30 = 65136
CONSTANT r33 = 63498, i33 = 201
CONSTANT r36 = 63644, i36 = 784
CONSTANT r39 = 63953, i39 = 1299
CONSTANT r42 = 64398, i42 = 1703
CONSTANT r45 = 64941, i45 = 1960
```

```
! stage 1 twiddles
MuxCofs [r0,r0,r0,r0,r4,r4,r4,r4,r8,r8,r8,r8,r12,r12,r12,r12,
      swl] (c1, e2a) -> slt1r, NC
MuxCofs [i0,i0,i0,i0,i4,i4,i4,i4,i8,i8,i8,i8,i12,i12,i12,i12,
      swl] (c1, e2a) -> slt1i, NC
MuxCofs [r0,r0,r0,r0,r8,r8,r8,r8,r16,r16,r16,r16,r24,r24,r24,r24,
      swl] (c1, e2a) -> slt2r, NC
MuxCofs [i0,i0,i0,i0,i8,i8,i8,i8,i16,i16,i16,i16,i24,i24,i24,i24,
      swl] (c1, e2a) -> slt2i, NC
MuxCofs [r0,r0,r0,r0,r12,r12,r12,r12,r24,r24,r24,r24,r36,r36,r36,r36,
      swl] (c1, e2a) -> slt3r, NC
```

```

MuxCofs [i0,i0,i0,i0,i12,i12,i12,i12,i24,i24,i24,i24,i36,i36,i36,i36,
swl] (c1, e2a) -> slt3i, NC
! stage 2 twiddles
MuxCofs [r0,r4,r8,r12,r1,r5,r9,r13,r2,r6,r10,r14,r3,r7,r11,r15,
swl] (c1, e2a) -> NC, s2t1r
MuxCofs [i0,i4,i8,i12,i1,i5,i9,i13,i2,i6,i10,i14,i3,i7,i11,i15,
swl] (c1, e2a) -> NC, s2t1i
MuxCofs [r0,r8,r16,r24,r2,r10,r18,r26,r4,r12,r20,r28,r6,r14,r22,r30,
swl] (c1, e2a) -> NC, s2t2r
MuxCofs [i0,i8,i16,i24,i2,i10,i18,i26,i4,i12,i20,i28,i6,i14,i22,i30,
swl] (c1, e2a) -> NC, s2t2i
MuxCofs [r0,r12,r24,r36,r3,r15,r27,r39,r6,r18,r30,r42,r9,r21,r33,r45,
swl] (c1, e2a) -> NC, s2t3r
MuxCofs [i0,i12,i24,i36,i3,i15,i27,i39,i6,i18,i30,i42,i9,i21,i33,i45,
swl] (c1, e2a) -> NC, s2t3i
CBITDELAY [1] (e2 -> e2a)

```

END

```

=====
!
=====
CHIP CGen (-> pc1, pc2, pc3, pc4, pc5)

```

CONTROL c1, c2, c3, c4, c5

PADOUT (c1, c2, c3, c4, c5 -> pc1, pc2, pc3, pc4, pc5)
PADORDER VDD, pc1, pc2, pc3, pc4, pc5, GND, CLOCK

CONTROLGENERATOR (-> NC, c1, c2, c3, c4, c5)
CYCLE [16]
CYCLE [2]
CYCLE [2]
CYCLE [2]
CYCLE [2]
ENDCONTROLGENERATOR

END

```

=====
!
=====
CHIP FirDummy ps0, ps1, ps2, ps3 -> ps00, ps01, ps02, ps03

```

SIGNAL s0, s1, s2, s3, so0, so1, so2, so3
PADIN ps0, ps1, ps2, ps3 -> s0, s1, s2, s3
PADOUT so0, so1, so2, so3 -> ps00, ps01, ps02, ps03
PADORDER VDD, ps0, ps1, ps2, ps3, GND, CLOCK, ps00, ps01, ps02, ps03
CONSTANT swl = 16

BITDELAY [(2*swl)-1] s0 -> so0
BITDELAY [(2*swl)-1] s1 -> so1
BITDELAY [(2*swl)-1] s2 -> so2
BITDELAY [(2*swl)-1] s3 -> so3

END

```
!=====
SYSTEM x (c1, ev2, ev6) dat0, dat1, cof0, cof1 -> rout0 THROUGH 3,
      iout0 THROUGH 3

  SIGNAL rcof1, icof1, rcof2, icof2, rcof3, icof3,
      rcof11, icof11, rcof12, icof12, rcof13, icof13
  SIGNAL r0 THROUGH 3
  CONTROL c2, c3, c4, c5

! note o/p wires 1 & 2 crossed
  FilterOut (c1, ev6) dat0, dat1, cof0, cof1 -> r0, r2, r1, r3
  Pipe (c1, c2, c3, c4, c5) r0 THROUGH 3, rcof1, rcof2,
      rcof3, icof1, icof2, icof3, rcof11, rcof12, rcof13, icof11, icof12,
      icof13 -> rout0, rout1, rout2, rout3, iout0, iout1, iout2, iout3
  FftCofs (c1, ev2) -> rcof1, icof1, rcof2, icof2, rcof3, icof3,
      rcof11, icof11, rcof12, icof12, rcof13, icof13
  CGen (-> c1, c2, c3, c4, c5)

END
endofprogram
```

Appendix B

FIRST Description of the Hard Model

```
! FIR section for PPN filter
! NB - notional CHIP FftCofs not included (see soft model)
!=====
OPERATOR FilterSection [swl,cofres,muxlevel,signif,bound] (c1,
    c11, ev -> clout, evout) datain, cof -> dataout, multout

    SIGNAL datadel, datadel2, cofsel, cofsel2, cofdelay
    CONTROL ev1, ev2, ev3

    CONSTANT multdel = ((3 * cofres) / 2) + 2

    MULTIPLY [1,cofres,1,0] (c11 -> clout) datain, cofsel -> multout, datadel
    WORDDELAY [muxlevel-2,signif,0] (clout) datadel -> datadel2
    BITDELAY [(2*swl)-(multdel+bound+2)] datadel2 -> dataout
    MULTIPLEX [1,0,0] (ev) cofdelay, cof -> cofsel
    WORDDELAY [muxlevel-1,cofres,0] (c11) cofsel -> cofsel2
    BITDELAY [swl-2] cofsel2 -> cofdelay

    CWORDDELAY [7,0] (c1, ev -> ev1)
    CBITDELAY [swl-7] (ev1 -> ev2)
    CWORDDELAY [7,0] (c1, ev2 -> ev3)
    CBITDELAY [swl-(7+bound)] (ev3 -> evout)

END
!=====
!=====
CHIP Section (pcl, pev -> pclout, pevout) pdatain, pcof -> -
    pmult0, pmult1, pdataout

    SIGNAL datain, cof, mult0, mult1, dataout, dataint
    CONTROL c1, c11, clout, ev, evout, evint

    PADIN (pcl, pev -> c1, ev) pdatain, pcof -> datain, cof
    PADOUT (clout, evout -> pclout, pevout) mult0, mult1,
        dataout -> pmult0, pmult1, pdataout
    PADORDER VDD, pcl, pev, pclout, pevout, pdatain, pcof, GND, CLOCK,
        pmult0, pmult1, pdataout

    CONSTANT swl = 16, cofres = 8, muxlevel = 16, signif = 8

    FilterSection [swl,cofres,muxlevel,signif,0] (c1,
        c11, ev -> clout, evint) datain, cof -> dataint, mult0
```



```
FilterSection [swl,cofres,muxlevel,signif,1] (c1,  
    c11, evint -> NC, evout) dataint, cof -> dataout, mult1  
CBITDELAY [1] (c1 -> c11)
```

END

```
!=====
```

```
SUBSYSTEM FilterCascade (c1, ev -> clout) datain,  
    cof -> proda0 THROUGH 5, prodb0 THROUGH 5
```

```
SIGNAL mult0, mult1, dataout  
CONTROL cldummy, evout
```

```
Section (c1, ev -> cldummy, evout) -  
    datain, cof -> mult0, mult1, dataout TIMES 6 WITH -  
    (evout -> ev) dataout -> datain  
    mult0 = proda0 THROUGH 5  
    mult1 = prodb0 THROUGH 5  
    cldummy = clout, NC, NC, NC, NC, NC
```

END

```
!=====
```

```
OPERATOR Adder [del] (c1, c11, c12) in0 THROUGH 5 -> out
```

```
SIGNAL s0 THROUGH 3
```

```
ADD [1,1,1,0] (c1) in0, in1, GND -> s0, NC  
ADD [1,1,1,0] (c1) in2, in3, GND -> s1, NC  
ADD [1,1,1,0] (c1) in4, in5, GND -> s2, NC  
ADD [1,0,0,0] (c11) s0, s1, GND -> s3, NC  
ADD [del-3,0,1,0] (c12) s3, s2, GND -> out, NC
```

END

```
!=====
```

```
CHIP AddTree (pc1) pina0 THROUGH 5, pinb0 THROUGH 5 -> psum0, psum1
```

```
SIGNAL ina0 THROUGH 5, inb0 THROUGH 5, sum0, sum1  
CONTROL c1, c11, c12, cla, clb
```

```
PADIN (pc1 -> c1) pina0 THROUGH 5, pinb0 THROUGH 5 -> -  
    ina0 THROUGH 5, inb0 THROUGH 5  
PADOUT sum0, sum1 -> psum0, psum1  
PADORDER VDD, pc1, pina0 THROUGH 5, GND, psum0, psum1,  
    CLOCK, pinb0 THROUGH 5
```

```
CONSTANT swl = 16, cofres = 8  
CONSTANT multdel = ((3 * cofres) / 2) + 2  
CONSTANT del = (2 * swl) - (multdel + 7)
```

```

Adder [del] (c1a, c11, c12) ina0 THROUGH 5 -> sum0
Adder [del] (c1b, c11, c12) inb0 THROUGH 5 -> sum1
CBITDELAY [1] (c1 -> c1a)
CBITDELAY [1] (c1 -> c1b)
CBITDELAY [1] (c1a -> c11)
CBITDELAY [1] (c11 -> c12)

END
!=====
!=====
SUBSYSTEM Fir (c1, ev) dat0, dat1, cof0, cof1 -> out0 THROUGH 3

SIGNAL a0 THROUGH 5, b0 THROUGH 5, z0 THROUGH 5, d0 THROUGH 5
CONTROL clout

FilterCascade (c1, ev -> clout) dat0,
    cof0 -> a0 THROUGH 5, b0 THROUGH 5
FilterCascade (c1, ev -> NC) dat1,
    cof1 -> z0 THROUGH 5, d0 THROUGH 5
! outputs 1 & 2 crossed over
AddTree (clout) a0 THROUGH 5, b0 THROUGH 5 -> out0, out2
AddTree (clout) z0 THROUGH 5, d0 THROUGH 5 -> out1, out3

END
!=====
! Radix-4 64-point pipeline FFT
! Dft4 is minimal-latency
!=====
OPERATOR Dft2 [pre0,pre1] (c1) rin0,rin1,iin0,iin1 -> rout0,rout1,
iout0,iout1

ADD [1,pre0,pre1,0] (c1) rin0,rin1,GND -> rout0,NC
SUBTRACT [1,pre0,pre1,0] (c1) rin0,rin1,GND -> rout1,NC
ADD [1,pre0,pre1,0] (c1) iin0,iin1,GND -> iout0,NC
SUBTRACT [1,pre0,pre1,0] (c1) iin0,iin1,GND -> iout1,NC

END
!=====
!=====
CHIP Dft4 (pc1) prin0 THROUGH 3, piin0 THROUGH 3 -> -
    prout0 THROUGH 3, piout0 THROUGH 3

SIGNAL rin0 THROUGH 3, iin0 THROUGH 3,
    rout0 THROUGH 3, iout0 THROUGH 3
SIGNAL r0,r1,r2,r3,i0,i1,i2,i3
CONTROL c1, c11, c12a, c12b
PADIN (pc1 -> c1) prin0 THROUGH 3, piin0 THROUGH 3 -> -
    rin0 THROUGH 3, iin0 THROUGH 3
PADOUT rout0 THROUGH 3, iout0 THROUGH 3 -> -
    prout0 THROUGH 3, piout0 THROUGH 3
PADORDER VDD, prin0 THROUGH 3, piin0 THROUGH 3, GND,

```

```

    pc1, CLOCK, prout0 THROUGH 3, piout0 THROUGH 3

CONSTANT swl = 16

Dft2 [0,0] (c1) rin0,rin2,iin0,iin2 -> r0,r1,i0,i1
Dft2 [1,1] (c11) rin1,rin3,iin1,iin3 -> r2,r3,i2,i3
Dft2 [1,0] (c12a) r0,r2,i0,i2 -> rout0,rout2,iout0,iout2
Dft2 [1,0] (c12b) r1,i3,i1,r3 -> rout1,rout3,iout3,iout1

CBITDELAY [1] (c1 -> c11)
CBITDELAY [1] (c11 -> c12a)
CBITDELAY [1] (c11 -> c12b)

END
!=====
!=====
CHIP CmplxMul (pc1) prin,piin,prcof,picof -> prout,piout

    SIGNAL rin,iin,rcof,icof,rout,iout
    SIGNAL s0,s1,s2,s3
    CONTROL c1, c11

    PADIN (pc1 -> c1) prin,piin,prcof,picof -> rin,iin,rcof,icof
    PADOUT rout, iout -> prout, piout
    PADORDER VDD, pc1, prin,piin,prcof,picof,GND,CLOCK, prout,piout

    CONSTANT swl = 16, co = 12
    CONSTANT multdel = ((3 * co) / 2) + 2

    MULTIPLY [1,co,0,0] (c1 -> c11) rin,rcof -> s0,NC
    MULTIPLY [1,co,0,0] (c1 -> NC) iin,icof -> s1,NC
    MULTIPLY [1,co,0,0] (c1 -> NC) rin,icof -> s2,NC
    MULTIPLY [1,co,0,0] (c1 -> NC) iin,rcof -> s3,NC
    SUBTRACT [(2*swl)-(multdel+5),0,0,0] (c11) s0,s1,GND -> rout,NC
    ADD [(2*swl)-(multdel+5),0,0,0] (c11) s2,s3,GND -> iout,NC

END
!=====
!=====
SUBSYSTEM Twiddle (c1) rin1 THROUGH 3, iin1 THROUGH 3,
    rcof1,rcof2,rcof3,icof1,icof2,icof3 -> rout1 THROUGH 3,
    iout1 THROUGH 3

    CmplxMul (c1) rin1,iin1,rcof1,icof1 -> rout1,iout1
    CmplxMul (c1) rin2,iin2,rcof2,icof2 -> rout2,iout2
    CmplxMul (c1) rin3,iin3,rcof3,icof3 -> rout3,iout3

END
!=====
!=====
OPERATOR ComReal [swl] (cn,cnplus1,cnplus1del) in0 THROUGH 3 -> -

```

out0 THROUGH 3

SIGNAL r0,r1,r2,r3

```
MULTIPLEX [swl/2,0,0] (cnplus1) in2,in0 -> r0
MULTIPLEX [swl/2,0,0] (cnplus1del) in3,in1 -> r1
MULTIPLEX [swl/2,0,0] (cnplus1) in0,in2 -> r2
MULTIPLEX [swl/2,0,0] (cnplus1del) in1,in3 -> r3
MULTIPLEX [swl-((swl/2)+1),0,0] (cn) r1,r0 -> out0
MULTIPLEX [swl-((swl/2)+1),0,0] (cn) r0,r1 -> out1
MULTIPLEX [swl-((swl/2)+1),0,0] (cn) r3,r2 -> out2
MULTIPLEX [swl-((swl/2)+1),0,0] (cn) r2,r3 -> out3
```

END

=====

OPERATOR Del12 [swl] in -> out

SIGNAL s0 THROUGH 4

```
BITDELAY [2*swl] in -> s0
BITDELAY [2*swl] s0 -> s1
BITDELAY [2*swl] s1 -> s2
BITDELAY [2*swl] s2 -> s3
BITDELAY [2*swl] s3 -> s4
BITDELAY [2*swl] s4 -> out
```

END

=====

OPERATOR Del8 [swl] in -> out

SIGNAL s0 THROUGH 2

```
BITDELAY [2*swl] in -> s0
BITDELAY [2*swl] s0 -> s1
BITDELAY [2*swl] s1 -> s2
BITDELAY [2*swl] s2 -> out
```

END

=====

=====

OPERATOR Del4 [swl] in -> out

SIGNAL s0

```
BITDELAY [2*swl] in -> s0
BITDELAY [2*swl] s0 -> out
```

END

=====

=====

OPERATOR Del3 [swl] in -> out

SIGNAL s0

BITDELAY [2*swl] in -> s0

BITDELAY [swl] s0 -> out

END

=====

OPERATOR Del2 [swl] in -> out

BITDELAY [2*swl] in -> out

END

=====

OPERATOR Del1 [swl] in -> out

SIGNAL s0, s1

BITDELAY [swl/3] in -> s0

BITDELAY [swl/3] s0 -> s1

BITDELAY [swl-(2*(swl/3))] s1 -> out

END

=====

OPERATOR Shuffle4 [swl] (c4, c5, c5del) in0 THROUGH 3 -> -
out0 THROUGH 3

SIGNAL a1 THROUGH 3, b0 THROUGH 2

Del4 [swl] in1 -> a1

Del8 [swl] in2 -> a2

Del12 [swl] in3 -> a3

ComReal [swl] (c4,c5,c5del) in0, a1 THROUGH 3 -> -
b0 THROUGH 2, out3

Del12 [swl] b0 -> out0

Del8 [swl] b1 -> out1

Del4 [swl] b2 -> out2

END

=====

OPERATOR Shuffle1 [swl] (c2, c3, c3del) in0 THROUGH 3 -> -
out0 THROUGH 3

SIGNAL a1 THROUGH 3, b0 THROUGH 2

Del1 [swl] in1 -> a1

```
Del2 [sw1] in2 -> a2
Del3 [sw1] in3 -> a3
ComReal [sw1] (c2,c3,c3del) in0, a1 THROUGH 3 -> -
    b0 THROUGH 2, out3.
Del3 [sw1] b0 -> out0
Del2 [sw1] b1 -> out1
Del1 [sw1] b2 -> out2

END
!=====
CHIP Commute (pc2, pc3, pc4, pc5) pina0 THROUGH 3,
    pinb0 THROUGH 3 -> pouta0 THROUGH 3, poutb0 THROUGH 3

    SIGNAL ina0 THROUGH 3, inb0 THROUGH 3,
        outa0 THROUGH 3, outb0 THROUGH 3, a0, b0
    CONTROL c2, c3, c4, c5, c31, c32, c3del, c51, c5del

    PADIN (pc2, pc3, pc4, pc5 -> c2, c3, c4, c5)
    PADIN pina0 THROUGH 3, pinb0 THROUGH 3 -> -
        ina0 THROUGH 3, inb0 THROUGH 3
    PADOUT outa0 THROUGH 3, outb0 THROUGH 3 -> -
        pouta0 THROUGH 3, poutb0 THROUGH 3
    PADORDER VDD, pc2, pina0 THROUGH 3, pinb0 THROUGH 3, GND,
        pc3, pc4, pc5, CLOCK, pouta0 THROUGH 3, poutb0 THROUGH 3

    CONSTANT sw1 = 16

    Shuffle4 [sw1] (c4, c5, c5del) ina0 THROUGH 3 -> -
        a0, outa1 THROUGH 3
    Shuffle1 [sw1] (c2, c3, c3del) inb0 THROUGH 3 -> -
        b0, outb1 THROUGH 3
! twiddle compensation for line 0
    BITDELAY [(2*sw1)-4] a0 -> outa0
    BITDELAY [(2*sw1)-4] b0 -> outb0

!    CBITDELAY [sw1] (c3 -> c3del)
    CBITDELAY [sw1/3] (c3 -> c31)
    CBITDELAY [sw1/3] (c31 -> c32)
    CBITDELAY [sw1-(2*(sw1/3))] (c32 -> c3del)
    CBITDELAY [2*sw1] (c5 -> c51)
    CBITDELAY [2*sw1] (c51 -> c5del)

END
!=====
!=====
CHIP Dummy (pc1) -> pout
    SIGNAL out
    CONTROL c1
    PADIN (pc1 -> c1)
    PADOUT out -> pout
    PADORDER VDD, GND, CLOCK, pc1, pout
```

```

    ADD [15-4,0,0,0] (c1) GND, GND, GND -> out, NC
END
=====
!=====
SUBSYSTEM Pipe (c1,c1d,c2,c3,c4,c5) rin0 THROUGH 3, rcof1,rcof2,rcof3,
    icof1,icof2,icof3,rcof11,rcof12,rcof13,icof11,icof12,
    icof13 -> rout0 THROUGH 3, iout0 THROUGH 3

SIGNAL r0,r1,r2,r3,i0,i1,i2,i3
SIGNAL r10,r11,r12,r13,i10,i11,i12,i13
SIGNAL r21,r22,r23,i21,i22,i23
SIGNAL r30,r31,r32,r33,i30,i31,i32,i33
SIGNAL r40,r41,r42,r43,i40,i41,i42,i43
SIGNAL r51,r52,r53,i51,i52,i53
SIGNAL dumgnd

Dummy (c1) -> dumgnd
Dft4 (c1d) rin0 THROUGH 3, dumgnd, dumgnd, dumgnd, dumgnd -> -
    r0 THROUGH 3, i0 THROUGH 3
Commute (c2, c3, c4, c5) r0 THROUGH 3,
    r30 THROUGH 33 -> r10 THROUGH 13, r40 THROUGH 43
Commute (c2, c3, c4, c5) i0 THROUGH 3,
    i30 THROUGH 33 -> i10 THROUGH 13, i40 THROUGH 43
Twiddle (c1) r11 THROUGH 13, i11 THROUGH 13,
    rcof1,rcof2,rcof3,icof1,icof2,icof3 -> r21 THROUGH 23,
    i21 THROUGH 23
Dft4 (c1d) r10, r21 THROUGH 23, i10, i21 THROUGH 23 -> -
    r30 THROUGH 33, i30 THROUGH 33
Twiddle (c1) r41 THROUGH 43, i41 THROUGH 43,
    rcof11,rcof12,rcof13,icof11,icof12,icof13 -> r51 THROUGH 53,
    i51 THROUGH 53
Dft4 (c1d) r40, r51 THROUGH 53, i40, i51 THROUGH 53 -> -
    rout0 THROUGH 3, iout0 THROUGH 3

END
=====
!=====
CHIP CGen (-> pc1, pc1d, pc2, pc3, pc4, pc5)

CONTROL c1, c1d, c2, c3, c4, c5, c41, c51, c2i, c3i, c4i, c5i

PADOUT (c1, c1d, c2, c3, c4, c5 -> pc1, pc1d, pc2, pc3, pc4, pc5)
PADORDER VDD, pc1, pc1d, pc2, pc3, pc4, pc5, GND, CLOCK

CONSTANT sw1 = 16

CONTROLGENERATOR (-> NC, c1, c2i, c3i, c4i, c5i)
    CYCLE [sw1]
    CYCLE [2]
    CYCLE [2]
    CYCLE [2]

```

```

    CYCLE [2]
    ENDCONTROLGENERATOR
! special c1 line for Dft4 chips
    CBITDELAY [swl-4] (c1 -> c1d)
! c2 delayed by 1.5 words
    CBITDELAY [swl+(swl/2)] (c2i -> c2)
! c3 delayed by 1 words
    CBITDELAY [swl] (c3i -> c3)
! c4 delayed by 2.5 words
    CBITDELAY [swl] (c4i -> c4i)
    CBITDELAY [swl+(swl/2)] (c4i -> c4)
! c5 delayed by 2 words
    CBITDELAY [swl] (c5i -> c5i)
    CBITDELAY [swl] (c5i -> c5)

END
=====
!
=====
SYSTEM x (c1, ev2, ev6) dat0, dat1, cof0, cof1 -> -
    rout0 THROUGH 3, iout0 THROUGH 3

    SIGNAL rcof1, icof1, rcof2, icof2, rcof3, icof3,
        rcof11, icof11, rcof12, icof12, rcof13, icof13
    SIGNAL r0 THROUGH 3
    CONTROL c1d, c2, c3, c4, c5

    Fir (c1, ev6) dat0, dat1, cof0, cof1 -> r0 THROUGH 3
    Pipe (c1, c1d, c2, c3, c4, c5) r0 THROUGH 3, rcof1, rcof2,
        rcof3, icof1, icof2, icof3, rcof11, rcof12, rcof13, icof11, icof12,
        icof13 -> rout0, rout1, rout2, rout3, iout0, iout1, iout2, iout3
    FftCofs (c1, ev2) -> rcof1, icof1, rcof2, icof2, rcof3, icof3,
        rcof11, icof11, rcof12, icof12, rcof13, icof13
    CGen (-> c1, c1d, c2, c3, c4, c5)

END
endofprogram
```


Appendix C

***RNL* description of the twin-pipe complex multiplier**

The following functional description of a complex multiplier primitive reflects many of the concepts contained in this thesis. This architecture uses distributed arithmetic for savings in transistor count, and twin-pipe operation for increased throughput. Interfaces in this case are bit-parallel. Many of the design capture techniques proposed for *SECOND* are illustrated by this example.

The design contains 4 levels of functional hierarchy. Each level starts with a call to a 'library' at the next lower level, and a declaration of internal nodes.

At the top level (LEVEL 0), such nodes are abstract - they contain no information about the number of bits in the signal, or whether it is single-pipe, twin-pipe etc. Thus at this level the design is captured not only in technologically-independent form, but also in operationally-independent form. The parameter n is passed to each constituent element (here value assignment to n is accomplished via the 'setq' statement, although in actuality a higher level of hierarchy would pass this parameter down). The *RNL* code relates closely to Figure 6.3(b) (with the addition of PISOs for data distribution). This is an appropriate level of abstraction for manipulation of matrix-vector architectures - for instance this complex multiplier description was derived from a 2-point inner-product computer description by simply instantiating a second CSAS computer (one line of *RNL* code). Only *functional* binding is performed at this level.

The next level down (LEVEL 1) reveals the contents of the high-level computational elements, and it is here that *operational* binding (to the twin-pipe architecture) occurs. The parameter n is used as final value for a loop construct, with i as the variable. The abstract nodes may take two extensions - the first denotes μ and π -timing (0 and 1 respectively), and the second corresponds to some function of the current value of i .

The next level (LEVEL 2) reveals the logic blocks (as mixed transistor trees), and the half-latch associated with each block. An exception is the ripple-carry

adder, which is asynchronous and fully static. The final hierarchical level (LEVEL 3) is the latch library.

Note that this hierarchy relates to the functional manipulation of logic elements. Physical manipulation requires a different hierarchy, as described in Chapter 8. User-inserted 'type' parameters in the functional hierarchy may be used by physical assembly routines for efficient primitive composition. However this facility has not yet been implemented.

```
;=====
; LEVEL 0
(load "xmultlib.net")
(node dlod clod hold lsb msb rd rc id ic gdat1 gdat2 dat1 dat2
pip1 pip2 sel1 sel2 rp rlop ip ilop clk)
(setq n 6)
(piso n gdat1 rd dlod clk)
(piso n gdat2 id dlod clk)
(pipo n pip1 rc clod clk)
(pipo n pip2 ic clod clk)
(select n sel1 sel2 dat1 dat2 gdat1 gdat2 pip1 pip2)
(csas n rp rlop dat1 sel1 pip1 lsb msb hold clk)
(csas n ip ilop dat2 sel2 pip2 lsb msb hold clk)

;=====
; LEVEL 1
(load "/isg/sgs/rnl/logic/logiclib.net")
(macro piso (n out pipar load clk)
(local dat)
  (repeat i 1 n
; PISO stage
    (mux_mu dat.0.i pipar.i load dat.1.(- i 1) clk)
    (piso1o dat.1.i dat.0.(- i 1) clk)
  )
; broadcast buffers
  (musolo out.0 dat.1.n clk)
  (piso1o out.1 dat.0.n clk)
; sign-repetition connections
  (connect dat.1.0 dat.1.1)
  (connect dat.0.0 dat.0.1)
)
(macro pipo (n out pipar load clk)
  (repeat i 1 n
; PIP0 stage
    (mux_mu out.0.i pipar.i load out.1.i clk)
    (piso1o out.1.i out.0.i clk)
```

```
)
)
(macro select (n sel1 sel2 dato1 dato2 dat1 dat2 pip1 pip2)
(local d1 d2)
  (repeat i 1 n
; input data buffering
    (musolo d1.0.i dat1.1 clk)
    (musolo d2.0.i dat2.1 clk)
    (pisolo d1.1.i dat1.0 clk)
    (pisolo d2.1.i dat2.0 clk)
; delay for output data
    (pisolo dato1.1.i d1.0.i clk)
    (pisolo dato2.1.i d2.0.i clk)
    (musolo dato1.0.i d1.1.i clk)
    (musolo dato2.0.i d2.1.i clk)
; selection
    (select_pi sel1.1.i d1.0.i d2.0.i pip1.0.i pip2.0.i clk)
    (select_pi sel2.1.i d1.0.i d2.0.i pip2.0.i pip1.0.i clk)
    (select_mu sel1.0.i d1.1.i d2.1.i pip1.1.i pip2.1.i clk)
    (select_mu sel2.0.i d1.1.i d2.1.i pip2.1.i pip1.1.i clk)
  )
)
(macro csas (n p lop dat sel erin lsb msb hold clk)
(local pps car bp a b er ripcar lsbl msbl dd inc inccar lsbi)
  (repeat i 1 n
; local control buffering and delay
    (pisolo msbl.i msb clk)
    (musolo lsbl.i lsb clk)
; bit-product formation
    (xor_pi bp.1.i sel.0.i dat.0.i clk)
    (xor_mu bp.0.i sel.1.i dat.1.i clk)
; delayed error compensation signal
    (musolo er.i erin.1.i clk)
; main CSAS computer
    (olscar_pi car.1.i bp.0.i pps.0.(- i 1) er.i car.0.i lsbl.i clk)
    (olssum_pi pps.1.i bp.0.i er.i pps.0.(- i 1) car.0.i lsbl.i clk)
    (omscar_mu car.0.i bp.1.i msbl.i pps.1.(- i 1) car.1.i clk)
    (omssum_mu pps.0.i bp.1.i msbl.i pps.1.(- i 1) car.1.i clk)
; ripple-adder with holding latches
    (pihold a.i pps.0.(- i 1) hold clk)
    (pihold b.i car.0.i hold clk)
    (ripcaa p.i ripcar.i a.i b.i ripcar.(+ i 1))
  )
; sign-repetition connections for main CSAS computer
  (connect pps.0.0 pps.0.1)
  (connect pps.1.0 pps.1.1)
; data delays and msb-treatment on feed to incrementing adder
  (musolo dd.0 dat.1.n clk)
  (pisolo inc.1 dd.0 clk)
  (pisolo dd.1 dat.0.n clk)
  (xor_mu inc.0 dd.1 msbl.n clk)
```

```
; delayed control feed to incrementing adder
  (pisolo lsbi lsbl.n clk)
; incrementing adder
  (evcar_mu inccar.0 inccar.1 lsbi inc.1 pps.1.n clk)
  (evsum_mu lop.0 inc.1 pps.1.n inccar.1 lsbi clk)
  (car_pi inccar.1 inc.0 pps.0.n inccar.0 clk)
  (sum_pi lop.1 inc.0 pps.0.n inccar.0 clk)
; final ripple-adder stage and holding latches
  (pihold ripcar.(+ n 2) inc.0 hold clk)
  (pihold a.(+ n 1) pps.0.n hold clk)
  (pihold b.(+ n 1) inccar.0 hold clk)
  (ripcsa p.(+ n 1) ripcar.(+ n 1) a.(+ n 1) b.(+ n 1) ripcar.(+ n 2))
)
```

```
;=====
```

```
; LEVEL 2
(load "/isg/sgs/rnl/latch/latchlib.net")
(macro ripcsa (sum car a b c)
  (local n6 n9 n8 p4 n20 n19 p15 p18 p17 p14)
  (ptrans a sum n6 4 3)
  (etrans a sum n20 4 3)
  (ptrans a sum p15 4 3)
  (etrans a sum p4 4 3)
  (ptrans b n6 n9 4 3)
  (etrans b n6 n8 4 3)
  (ptrans c n9 Gnd 4 3)
  (etrans c n8 Gnd 4 3)
  (ptrans b p4 p17 4 3)
  (etrans b p4 p18 4 3)
  (ptrans a car n20 4 3)
  (ptrans b car n19 4 3)
  (etrans a car p15 4 3)
  (etrans b car p14 4 3)
  (ptrans b n20 n8 4 3)
  (etrans b n20 n9 4 3)
  (ptrans c n19 Gnd 4 3)
  (ptrans b p15 p18 4 3)
  (etrans b p15 p17 4 3)
  (etrans c p18 Vdd 4 3)
  (ptrans c p17 Vdd 4 3)
  (etrans c p14 Vdd 4 3)
)
(macro olscar_pi (out PIP B ER C LSB clk)
; selection (3) from olscar_pi.tre
  (local top bot N4 N8 N7 N2 N3)
  (etrans PIP top N4 4 3)
  (etrans B top N2 4 3)
  (etrans B N4 N8 4 3)
  (etrans ER N4 N7 4 3)
  (etrans C N4 N8 4 3)
)
```

```
(ptrans LSB N8 bot 4 3)
(etrans LSB N7 bot 4 3)
(etrans C N2 N3 4 3)
(ptrans LSB N3 bot 4 3)
(pi out top bot clk)
)
(macro olssum_pi (out PIP ER B C LSB clk)
; selection (10) from olssum_pi.tre
(local top bot N5 N11 N10 N8 N9 N2)
(ptrans PIP top N5 4 3)
(etrans PIP top N2 4 3)
(etrans ER N5 N11 4 3)
(ptrans B N5 N10 4 3)
(etrans B N5 N8 4 3)
(etrans LSB N11 bot 4 3)
(etrans C N10 N9 4 3)
(ptrans C N8 N9 4 3)
(ptrans LSB N9 bot 4 3)
(ptrans ER N2 N11 4 3)
(ptrans B N2 N8 4 3)
(etrans B N2 N10 4 3)
(pi out top bot clk)
)
(macro car_mu (out A B C clk)
; selection (0) from car_mu.tre
(local top bot N3 N2)
(ptrans A top N3 4 3)
(ptrans B top N2 4 3)
(ptrans B N3 bot 4 3)
(ptrans C N3 bot 4 3)
(ptrans C N2 bot 4 3)
(mu out top bot clk)
)
(macro sum_mu (out A B C clk)
; selection (0) from sum_mu.tre
(local top bot N4 N7 N6 N2)
(ptrans A top N4 4 3)
(etrans A top N2 4 3)
(ptrans B N4 N7 4 3)
(etrans B N4 N6 4 3)
(ptrans C N7 bot 4 3)
(etrans C N6 bot 4 3)
(ptrans B N2 N6 4 3)
(etrans B N2 N7 4 3)
(mu out top bot clk)
)
(macro mux_mu (out HISEL CTRL LOSEL clk)
; selection (0) from mux_mu.tre
(local top bot N3 N2)
(ptrans HISEL top N3 4 3)
(ptrans CTRL top N2 4 3)
```

```
(etrans CTRL N3 bot 4 3)
(ptrans LOSEL N2 bot 4 3)
(mu out top bot clk)
)
(macro select_pi (out DATA1 DATA2 PIP1 PIP2 clk)
; selection (5) from select_pi.tre
(local top bot N4 N7 N6 N2)
(ptrans DATA1 top N4 4 3)
(etrans DATA1 top N2 4 3)
(ptrans DATA2 N4 N7 4 3)
(etrans DATA2 N4 N6 4 3)
(etrans PIP1 N7 bot 4 3)
(etrans PIP2 N6 bot 4 3)
(ptrans DATA2 N2 N6 4 3)
(etrans DATA2 N2 N7 4 3)
(pi out top bot clk)
)
(macro select_mu (out DATA1 DATA2 PIP1 PIP2 clk)
; selection (5) from select_mu.tre
(local top bot N4 N7 N6 N2)
(ptrans DATA1 top N4 4 3)
(etrans DATA1 top N2 4 3)
(ptrans DATA2 N4 N7 4 3)
(etrans DATA2 N4 N6 4 3)
(ptrans PIP1 N7 bot 4 3)
(ptrans PIP2 N6 bot 4 3)
(ptrans DATA2 N2 N6 4 3)
(etrans DATA2 N2 N7 4 3)
(mu out top bot clk)
)
(macro car_pi (out A B C clk)
; selection (0) from car_pi.tre
(local top bot N3 N2)
(etrans A top N3 4 3)
(etrans B top N2 4 3)
(etrans B N3 bot 4 3)
(etrans C N3 bot 4 3)
(etrans C N2 bot 4 3)
(pi out top bot clk)
)
(macro sum_pi (out A B C clk)
; selection (0) from sum_pi.tre
(local top bot N4 N7 N6 N2)
(ptrans A top N4 4 3)
(etrans A top N2 4 3)
(ptrans B N4 N7 4 3)
(etrans B N4 N6 4 3)
(etrans C N7 bot 4 3)
(ptrans C N6 bot 4 3)
(ptrans B N2 N6 4 3)
(etrans B N2 N7 4 3)
```

```
(pi out top bot clk)
)
(macro evcar_mu (out C LSB A B clk)
; selection (5) from evcar_mu.tre
(local top bot N4 N3)
(ptrans C top N4 4 3)
(etrans LSB top N4 4 3)
(ptrans A top N3 4 3)
(ptrans A N4 bot 4 3)
(ptrans B N4 bot 4 3)
(ptrans B N3 bot 4 3)
(mu out top bot clk)
)
(macro evsum_mu (out A B C LSB clk)
; selection (0) from evsum_mu.tre
(local top bot N4 N8 N6 N7 N2)
(ptrans A top N4 4 3)
(etrans A top N2 4 3)
(ptrans B N4 N8 4 3)
(etrans B N4 N6 4 3)
(ptrans C N8 bot 4 3)
(etrans LSB N8 bot 4 3)
(etrans C N6 N7 4 3)
(ptrans LSB N7 bot 4 3)
(ptrans B N2 N6 4 3)
(etrans B N2 N8 4 3)
(mu out top bot clk)
)
(macro xor_mu (out A B clk)
; selection (0) from xor_mu.tre
(local top bot N3 N2)
(ptrans A top N3 4 3)
(etrans A top N2 4 3)
(ptrans B N3 bot 4 3)
(etrans B N2 bot 4 3)
(mu out top bot clk)
)
(macro xor_pi (out A B clk)
; selection (0) from xor_pi.tre
(local top bot N3 N2)
(ptrans A top N3 4 3)
(etrans A top N2 4 3)
(etrans B N3 bot 4 3)
(ptrans B N2 bot 4 3)
(pi out top bot clk)
)
(macro not_pi (out A clk)
; selection (0) from not_pi.tre
(local top bot)
(ptrans A top bot 4 3)
(pi out top bot clk)
```

```
)
(macro omscar_mu (out A MSB B C clk)
; selection (0) from omscar_mu.tre
(local top bot N5 N8 N7 N2 N4)
(ptrans A top N5 4 3)
(etrans A top N2 4 3)
(ptrans MSB N5 N8 4 3)
(ptrans B N5 N7 4 3)
(ptrans B N8 bot 4 3)
(ptrans C N8 bot 4 3)
(ptrans C N7 bot 4 3)
(etrans MSB N2 N8 4 3)
(ptrans B N2 N4 4 3)
(ptrans C N4 bot 4 3)
(mu out top bot clk)
)
(macro omssum_mu (out A MSB B C clk)
; selection (0) from omssum_mu.tre
(local top bot N4 N10 N6 N9 N8 N2)
(ptrans A top N4 4 3)
(etrans A top N2 4 3)
(ptrans MSB N4 N10 4 3)
(etrans MSB N4 N6 4 3)
(ptrans B N10 N8 4 3)
(etrans B N10 N9 4 3)
(ptrans B N6 N9 4 3)
(etrans B N6 N8 4 3)
(etrans C N9 bot 4 3)
(ptrans C N8 bot 4 3)
(ptrans MSB N2 N6 4 3)
(etrans MSB N2 N10 4 3)
(mu out top bot clk)
)
```

;

```
; LEVEL 3
(macro pilatch (out in clk)
(local outbar mid)
(ptrans clk in outbar 4 3)
(etrans out outbar mid 4 3)
(etrans clk Gnd mid 4 3)
(ptrans outbar Vdd out 4 3)
(etrans outbar Gnd out 4 3)
)
(macro mulatch (out in clk)
(local outbar mid)
(etrans clk in outbar 4 3)
(ptrans out outbar mid 4 3)
(ptrans clk Vdd mid 4 3)
(ptrans outbar Vdd out 4 3)
```



```
(etrans outbar Gnd out 4 3)
)
(macro pisense (out top bot clk)
(ptrans clk Vdd top 4 3)
(etrans clk Gnd bot 4 3)
(ptrans out bot top 4 3)
(ptrans bot Vdd out 4 3)
(etrans bot Gnd out 4 3)
)
(macro musense (out top bot clk)
(ptrans clk Vdd top 4 3)
(etrans clk Gnd bot 4 3)
(etrans out bot top 4 3)
(ptrans top Vdd out 4 3)
(etrans top Gnd out 4 3)
)
(macro pi (out top bot clk)
(local inv)
(pisense inv top bot clk)
(pilatch out inv clk)
)
(macro mu (out top bot clk)
(local inv)
(musense inv top bot clk)
(mulatch out inv clk)
)
(macro pihold (out in hold clk)
(local inv held)
(ptrans in inv Vdd 4 3)
(etrans in inv Gnd 4 3)
(etrans hold inv held 4 3)
(pilatch out held clk)
)
(macro muhold (out in hold clk)
(local inv held)
(ptrans in inv Vdd 4 3)
(etrans in inv Gnd 4 3)
(ptrans hold inv held 4 3)
(mulatch out held clk)
)
(macro pisolo (out in clk)
(local inv)
(ptrans in inv Vdd 4 3)
(etrans in inv Gnd 4 3)
(pilatch out inv clk)
)
(macro musolo (out in clk)
(local inv)
(ptrans in inv Vdd 4 3)
(etrans in inv Gnd 4 3)
(mulatch out inv clk)
```

Next we list the *MOSYN* descriptions of the logic blocks used. In this case *PRESYN* was not used, i.e. the descriptions are operationally fixed (as twin-pipe). In the case of π -timed output, conductance of the logic tree is required when the function is true. In the case of μ -timed output, conductance of the logic tree is required when the function is false.

```
; mu carry function
*function carmu
CH: major(A,B,C)
else C0

; pi carry function
*function carpi
C0: major(A,B,C)
else CH

; mu carry function with clearable C
*function evcarmu
CH: major(A,B,and(C,not(LSB)))
else C0

; mu sum function with clearable C
*function evsummu
CH: xor(A,B,and(C,not(LSB)))
else C0

; mu multiplexer
*function muxmu
CH: or(and(HISEL,CTRL),and(LOSEL,not(CTRL)))
else C0

; pi not function
*function notpi
C0: not(A)
else CH

; carry function with clearable B and loadable C
*function olscarpi
C0: major(PIP,and(B,not(LSB)),or(and(ER,LSB),and(C,not(LSB))))
else CH

; sum function with clearable B and loadable C
*function olssumpi
C0: xor(PIP,and(B,not(LSB)),or(and(ER,LSB),and(C,not(LSB))))
else CH
```

```
; A + B carry function with ripple-carry
*function car
C1: major(A,B,C)
else C0

; A + B sum function with ripple-carry
*function sum
C1: xor(A,B,C)
else C0


; mu PIP selector
*function selectmu
CH: or(and(PIP1,not(xor(DATA1,DATA2))),and(PIP2,xor(DATA1,DATA2)))
else C0


; pi PIP selector
*function selectpi
C0: or(and(PIP1,not(xor(DATA1,DATA2))),and(PIP2,xor(DATA1,DATA2)))
else CH


; mu sum function
*function summu
CH: xor(A,B,C)
else C0


; pi sum function
*function sumpi
C0: xor(A,B,C)
else CH


; mu xor function
*function xormu
CH: xor(A,B)
else C0


; pi xor function
*function xorpi
C0: xor(A,B)
else CH
```

Simulation

The model was exercised under various input patterns. The task of the complex multiplier was to perform the calculation: $E = AC - BD$, $F = AD + BC$. In each case, the correct result was observed. One such example is given below, for the conditions:

variable	sim.name	expression	binary	decimal
A	rd		001100	12
B	id		001101	13
PIP1	ic		001110	14
PIP2	rc		001111	15
C		$-\frac{1}{2}(PIP1 + PIP2)$		-29
D		$-\frac{1}{2}(PIP1 - PIP2)$		-1
E	rp	AC-BD	111010110001	-335
F	ip	AD+BC	111001111011	-389

The drive file is shown below, followed by the simulation output. The top 7 bits of real and imaginary product terms are bunched, parallel outputs, while the bottom 5 bits are in twin-pipe serial form, starting with the μ -timed LSB (*rlop.0* and *ilop.0*) at time = 90.

```
(load "uwstd.l")
(load "eustd.l")
(read-network "xmult.bin")
(setq incr 100)
(log-file "xmult.res")
I
sim-init
V clk lhlhlhlhlhlhlh
V dlod hlllllllllll
V clod llhhlllllllll
V lsb hhhhhhllllhhhh
V msb lllllllllhllll
V hold llllllllllhlh
l rd.1 rd.2 rd.5 rd.6 id.1 id.2 id.5 rc.1 rc.2 rc.6 ic.1 ic.2
h rd.3 rd.4 id.3 id.4 id.6 rc.3 rc.4 rc.5 ic.3 ic.4 ic.5 ic.6
w clk rp.1 rp.2 rp.3 rp.4 rp.5 rp.6 rp.7 rlop.0 rlop.1
w ip.1 ip.2 ip.3 ip.4 ip.5 ip.6 ip.7 ilop.0 ilop.1
g rp.1 rp.2 rp.3 rp.4 rp.5 rp.6 rp.7
g ip.1 ip.2 ip.3 ip.4 ip.5 ip.6 ip.7
R
```

Switch-level init : 2094

RNL simulation results : SWITCH LEVEL

		r r			i i	
		l l			l l	
		rrrrrrrr	o o	iiiiiii	o o	
	c	pppppppp	p p	pppppppp	p p	
	l	
TIME	k	1234567	0 1	1234567	0 1	
(ns)						
0	0	0000001	1 1	0100001	1 1	
10	0	1111111	1 1	1111111	1 1	
20	1	1111111	1 1	1111111	0 1	
30	0	1111111	1 0	1111111	0 1	
40	1	1111111	1 0	1111111	0 1	
50	0	1111111	1 0	1111111	0 1	
60	1	1111111	0 0	1111111	0 1	
70	0	1111111	0 0	1111111	0 0	
80	1	1111111	1 0	1111111	1 0	
90	0	1111111	1 0	1111111	1 1	
100	1	1111111	0 0	1111111	0 1	
110	0	1111111	0 0	1111111	0 1	
120	1	1111111	1 0	1111111	1 1	
130	0	1110101	1 1	1110011	1 1	
140	1	1110101	0 1	1110011	0 1	
q						

Appendix D

Author's publications

The following is a list of the author's relevant publications, in chronological order. Appendix E is a reproduction of a book chapter (ref. 7 below), which contains detailed case studies of *FIRST* which were performed prior to the work reported in Chapter 5.

In available literature

1. S. G. Smith, C. F. N. Cowan and M. J. Rutter, "A New Structure for Adaptive Echo Cancellation," *Proc. IEEE ICASSP'83* pp. 49 - 52 (Boston, MA, April 1983)
2. C. F. N. Cowan, S. G. Smith and J. H. Elliott, "A Digital Adaptive Filter using a Memory-Accumulator Architecture: Theory and Realisation," *Trans. IEEE ASSP-31* pp. 541 - 549 (June 1983)
3. N. Petrie, J. Mavor and S. G. Smith, "General-Purpose Adaptor Structure for Wave-Digital-Filter Realisation," *Electronics Letters* **19** pp. 1038 - 1039 (November 24, 1983)
4. J. H. Nash and S. G. Smith, "A Front End Graphic Interface to the FIRST Silicon Compiler," *Proc. IEE Electronic Design Automation Conf. (EDA84)* pp. 120 - 124 (Warwick, UK, March 1984)
5. S. G. Smith, "Modelling Musical Instruments in the Digital Domain," *Proc. IEEE ICASSP'84* pp. 19.7.1 - 19.7.4 (San Diego, March 1984)
6. S. G. Smith, "A Silicon Compiler for Bit-Serial Signal Processors," *Proc. Computer Graphics User '85* pp. 67 - 76 (London, February 1985)
7. S. G. Smith, "Fourier Transform Machines," Chapter 8, pp. 147 - 199 in P. B. Denyer & D. Renshaw, *VLSI Signal Processing - A Bit-Serial Approach*, Reading, MA, Addison-Wesley, 1985
8. S. G. Smith, "Transversal Filters," Chapter 9, pp. 200 - 235 in P. B. Denyer & D. Renshaw, *VLSI Signal Processing - A Bit-Serial Approach*, Reading, MA, Addison-Wesley, 1985
9. P. B. Denyer and S. G. Smith, "Bit-Serial Architectures for Parallel Arrays," *Proc. SPIE 614, Highly Parallel Signal Processing Architectures* pp. 66 - 73 (Los Angeles, January 1986)

10. S. G. Smith and P. B. Denyer, "Efficient Bit-Serial Complex Multiplication and Sum-of-Products Computation Using Distributed Arithmetic," *Proc. IEEE-IECEJ-ASJ ICASSP'86* pp. 2203 - 2206 (Tokyo, April 1986)
11. S. G. Smith, A. Fitzgerald, P. B. Denyer, D. Renshaw, N. P. Wootton and R. Creasey, "A Comparison of Micro-DSP and Silicon Compiler Implementations of a Polyphase-Network Filter Bank," *Proc. IEEE-IECEJ-ASJ ICASSP'86* pp. 2207 - 2210 (Tokyo, April 1986)
12. S. G. Smith, "Efficient Serial/Parallel Inner-Product Computation," *Electronics Letters* **22** pp. 750 - 752 (July 3, 1986)
13. S. G. Smith, "Serial/Parallel Modules for Complex Arithmetic," *Electronics Letters* **22** pp. 1256- 1257 (November 6, 1986)
14. S. G. Smith "Silicon Compilers - Design Synthesis Beyond CAD," pp. 135 - 147, Chapter 11 in G. Russell (ed.) *Computer-Aided Tools for VLSI Design*, Peter Peregrinus (1987)

To appear

15. S. G. Smith and P. B. Denyer, "Serial/Parallel Architectures for Area-Efficient Vector Multiplication," *Proc. IEEE ICASSP'87* (Dallas, TX, April 1987)
16. S. G. Smith, M. S. McGregor and P. B. Denyer, "Techniques to Increase the Computational Throughput of Bit-Serial Architectures," *Proc. IEEE ICASSP'87* (Dallas, TX, April 1987)
17. S. G. Smith, P. B. Denyer, D. Renshaw, K. Asada, K. P. Coplan, M. Keightley and J. I. Mhar, "Full-Span Structural Compilation of DSP Hardware," *Proc. IEEE ICASSP'87* (Dallas, TX, April 1987)
18. S. G. Smith and P. B. Denyer, "Synthesis of Area-Efficient VLSI Architectures for Vector and Matrix Multiplication," *Proc. 8th IEEE Symp. on Computer Arith.* (Como, Italy, May 1987)

In review

19. S. G. Smith and P. B. Denyer, "Radix-4 Modules for High-Performance Bit-Serial Computation," submitted to *Proc. IEE Part E* (November 1986)
20. S. G. Smith, "Comments on 'A Signed Bit-Sequential Multiplier'," submitted to *IEEE Trans. Computers* (November 1986)
21. S. G. Smith, "Incremental Computation of Squares and Sums of Squares," submitted to *IEEE Trans. Computers* (November 1986)
22. M. S. McGregor, S. G. Smith, P. B. Denyer and A. F. Murray, "Serial-Data Computation on Twin Pipelines," submitted to *Electronics Letters* (December 1986)
23. S. G. Smith, "The Serial/Parallel Automultiplier," submitted to *Electronics Letters* (January 1987)

Fourier Transform Machines

Stewart G. Smith
Department of Electrical Engineering
University of Edinburgh

8.1 Introduction

As an introductory case study this chapter is somewhat more detailed than the remaining studies. In part this reflects the importance of the Fourier transform, in particular the Fast Fourier Transform (FFT), as a major field of application. However, this chapter also offers some practical tuition on the use of FIRST in constructing realistically large systems.

8.2 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) plays a significant role in the field of spectral analysis, and is a common tool for mapping between time and frequency domains in digital signal processing. The DFT transforms an N-point sampled time series into an equivalent N-point frequency series.

Consider the k-th member of the output series, X(k). X(k) is constructed by accumulating the element-by-element product of the input sequence x(n) with the particular complex exponential weighting sequence which repeats itself k times every N samples. In equation form, the value of the transform X at the k-th frequency point, or 'bin', is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)nk}$$

8.1

where both X(k) and x(n) are assumed periodic with period N (Rabiner and Gold, 1975) and k ranges from 0 to N – 1. The factor 2π/N normalises the argument of the complex exponential, allowing k to span (at discrete intervals) the frequency range from zero up to, but not including, the sampling frequency. The variable n represents time, and so the quantity nk represents normalised angular displacement. As angles can be reduced modulo 2π, so can normalised angles be reduced modulo N. The ability to reduce angles modulo some sequence length is a cornerstone of the Fast Fourier Transform (FFT) algorithm, as we shall see.

A unique property of the complex exponential function (of period N samples) is that uniformly spaced samples form orthonormal sequences. Thus

the N complex exponential weighting sequences are mutually orthonormal over the transform length, and as such form the basis vectors of an N -dimensional transform space. The DFT consists of the projection of the input sequence on to each of these basis vectors. Alternatively, the DFT is the evaluation of the z -transform of the input sequence $x(n)$ at N equally-spaced points on the unit circle in the complex plane. Whilst it is not absolutely necessary to absorb the underlying theory, the relevance of the unit circle to the DFT should not be missed.

Each complex exponential sequence may be realised by taking the unit vector in the complex plane, and repetitively rotating it by a constant angle until the N -point complex sequence has been generated. This process is carried out for the set of N different angles φ for which $N\varphi = 0 \bmod 2\pi$. Any vector of unit length whose angle exhibits this property is an N -th root of unity. Then equation (8.1) can be expressed:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad 8.2$$

where we take $W_N = \exp(-j2\pi/N)$ to be the principal N -th root of unity.[†] If we imagine the set of N -th roots of unity as forming the spokes of a wheel whose rim is the unit circle in the complex plane, then W is the clockwise neighbour to the spoke representing unity.

The input sequence to the DFT is assumed to consist of N rotating components, whose angular advances in one sample period (i.e. frequencies) correspond to the angles between the N wheel-spokes and unity. It is then periodic in N samples. Any component of normalised frequency k will be 'de-rotated' by the k -th complex exponential, and accumulation in the k -th bin will result in vector growth to a value proportional to the component.

Due to the orthogonality of the basis vectors, and the assumed orthogonality of the signal components, each signal component projects on to only one output bin. If the input signal is not periodic, it will contain components which project on to more than one basis vector, and the energy of such components will be distributed over several bins.

8.2.1 Some computational approaches to the DFT

To compute the DFT, we need only the operations of rotation and combination of vectors in the complex plane, which may be accomplished by complex multiplication and accumulation respectively. Computation of the DFT by straightforward means requires order (N^2) complex arithmetic operations (rotate-and-accumulates). This is equivalent to general matrix-vector multiplication. However, the DFT matrix may be factored in many ways, and if some combination is done before rotation, the amount of computation may be reduced.

[†] The reader should be aware of some inconsistency in the literature where the sign of the complex exponent is concerned. We follow the more usual convention $W_N = \exp(-j2\pi/N)$.

The Fast Fourier Transform (FFT) (Cooley and Tukey, 1965; Bergland, 1969) takes advantage of the properties of symmetry and periodicity of the DFT weights to reduce the computational complexity of the DFT to order($N \log N$).

Another class of DFT machines perform Number Theoretical Transforms (NTTs). These algorithms exploit number theoretical properties of either data or data addresses. The former class of algorithms treat data samples as elements in some finite computational structure, such as a field or a ring (McClellan and Rader, 1979). Hardware realisations make extensive use of residue arithmetic; for example Reed *et al.* (1983) detail a recent pipeline architecture for an NTT.

Examples of the latter algorithm class range from Rader's prime-length transform (Rader, 1968) to the Winograd Fourier Transform Algorithm (WFTA) (Winograd, 1978). The WFTA features a lower arithmetic complexity than the FFT, but at the expense of more complicated control arrangements. It is certainly a useful algorithm for computation of the DFT on mainframe computers, where complex data routing (still a matter of memory reads and writes) is almost free and multiplication is time-consuming. In VLSI, however, communication becomes a dominant algorithmic expense (Mead and Conway, 1980).

We restrict our studies to FFT and full DFT machines, staying within the mainstream of current systems implementation (Swartzlander and Hallnor, 1984; Linderman *et al.*, 1984). The first study consists of a full array realisation of a sixteen-point block FFT, using a constant geometry, decimation-in-time (DIT) radix-2 algorithm (Rabiner and Gold, 1975), but containing a hardwired shuffle to produce normally-ordered outputs. The use of multiplexing to reduce this machine to a single processor column is demonstrated, and partitioning issues are addressed to produce an optimised set of 4 chip types.

Next we describe a different multiplexing scheme, in the form of a radix-4, 64-point pipeline machine resembling that of McClellan and Purdy (1978) but with a shuffle network for output re-ordering. In contrast, our final study follows a modular, pipelined linear array architecture to compute the DFT as described by Allen (1984) after Kung (1980).

8.3 A DFT toolkit

It is normal design practice to specify from the top down, and implement from the bottom up. However, as this chapter serves to introduce the practical application of FIRST, we shall begin by creating some small modules which can be used later on to construct Fourier transform machines. This will necessarily lead to some 'acts of faith', as some computational elements are introduced before their theoretical background is discussed.

8.3.1 Vector rotation

It is clear from the preceding discussion that all Fourier transform machines

require the operation of vector rotation. CORDIC (for Co-Ordinate Rotation Digital Computer) hardware (Volder, 1959) is capable of either vector rotation or angle extraction, and has been used in the construction of FFT machines (Despain, 1974). However, being of iterative nature it is not well suited to bit-serial implementation.

The most common vector rotation element is the complex multiplier with unity-modulus coefficient. The FIRST primitive set outlined in Chapter 4 does not contain a complex multiplier *per se*, but its construction from real arithmetic elements is straightforward.

We need hardware to solve the following:

$$e + jf = (a + jb)(c + jd) \tag{8.3}$$

where $j^2 = -1$, i.e.

$$e = ac - bd, \qquad f = ad + bc \tag{8.4}$$

Equation (8.4) shows explicitly the requirements of complex multiplication using real arithmetic elements. We need four multipliers, an adder and a subtractor – all within the set of FIRST primitives. The code for this OPERATOR 'Xmult' is:

```
OPERATOR Xmult[coeff,lat](c1->c1out) a,b,c,d->e,f
latency is (3*coeff)/2 + 2 + lat, i.e. sum of multiplier and adder latencies

SIGNAL ac,bd,ad,bc
CONTROL c11

MULTIPLY[1,coeff,0,0](c1->c11) a,c->ac,NC
MULTIPLY[1,coeff,0,0](c1->NC) b,d->bd,NC
MULTIPLY[1,coeff,0,0](c1->NC) a,d->ad,NC
MULTIPLY[1,coeff,0,0](c1->NC) b,c->bc,NC
SUBTRACT[lat,0,0,0](c11) ac,bd,GND->e,NC
ADD[lat,0,0,0](c11) ad,bc,GND->f,NC

CBITDELAY[lat](c11->c1out)
```

END

This code declares an OPERATOR whose external input signal nodes a, b, c and d correspond to the initial variables of the right-hand side of equation (8.3); the elementary variables for the calculations. The external output signal nodes e and f represent the left-hand side of equation (8.3), which we wish to evaluate in terms of a, b, c and d. The internal signal nodes (ac, bd, etc.), on the other hand, are the intermediate terms in equations (8.4). We need neither remember how a complex multiplier works after we design it, nor have further dealings with internal nodes. These details are hidden in the hierarchy. We had

a problem (equation 8.3), and now have a functional element (OPERATOR Xmult) to solve it.

The only significant feature of Xmult, apart from its function, is its latency; the time delay in bits between inputs and outputs. When Xmult is used at a higher design level, its latency must be known. It is good practice to include the latency in a 'comment'.

For future flexibility we have parameterised the coefficient wordlength used in the various multipliers, and the latency of the adder and subtractor. This allows the resolution and hence the accuracy of the vector rotator to be varied, while the adder latency may control its overall latency. We do not anticipate using the input predelay option, so all predelays are set to zero.

Control for the adder and subtractor is derived from one of the multipliers (the multiplier outputs control as well as data). The multiplier control output is further delayed to achieve synchronous control and data outputs from the OPERATOR. Figure 8.1 shows the flow-graph for Xmult.

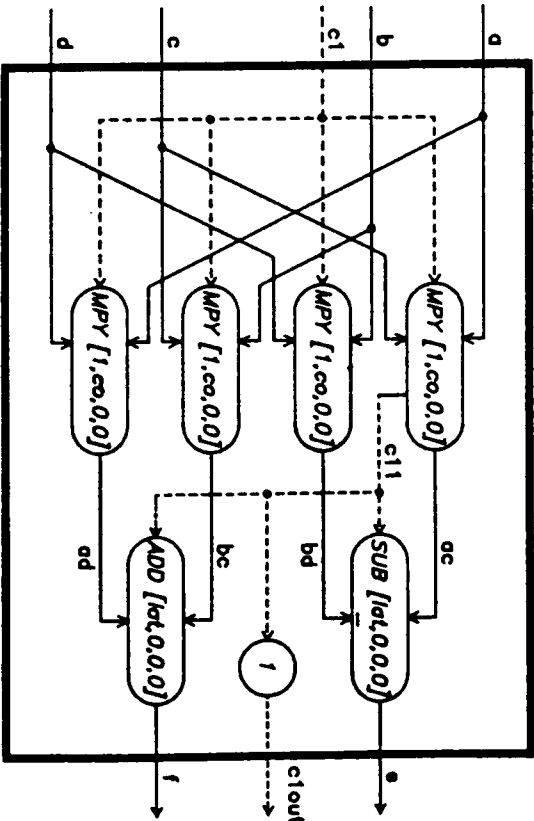


Fig. 8.1 Complex Multiplication OPERATOR 'Xmult'.

A better complex multiplier?

The components of Xmult are capable of generating the sum of one product pair and the difference of another. We have created a complex multiplier by cross-wiring the inputs. Just as the FFT uses the commonality of coefficients to make computational savings over the DFT by combining before rotating, so it is possible to perform addition before multiplication to reduce the number of real multiplies in the complex multiplier.

Using the commonality property, equation (8.3) may be expanded as:

$$e + jf = (ac - bd) + j((a + b)(c + d) - ac - bd)$$

or
$$e + jf = a(c + d) - d(a + b) + j(a(c + d) - c(a - b))$$

(Golub's method), or in the more specific case where $c = \cos\theta$ and $d = \sin\theta$ (vector rotation),
$$e + jf = (1 + \cos\theta)(a - b \tan(\theta/2)) - b + j((1 + \cos\theta)(a - b \tan(\theta/2)) \tan(\theta/2) + a)$$

(Buneman's method (Despain, 1974)).

Each expression involves only three multiplies, and five (Golub) or three (Buneman) add/subtracts. In terms of silicon area, these seem to be more attractive options. However, there are penalties to be paid for these savings.

Firstly multiplier coefficients have been allowed to exceed unity; some extra shifting must be incorporated to re-align products. Initial addition causes words to grow by one bit, causing loss of coefficient resolution which can only be regained by increasing the size and latency of the multipliers. The same loss (of a bit) applies to the data, and thus the system wordlength must be increased by one bit to compensate. Finally we must include some control delay primitives, as we cannot derive control output from an adder. These features make the three-multiplier rotators somewhat less attractive in this case.

The best of all solutions is to engineer a dedicated complex multiplier primitive. Murray *et al.* (1984) report an example of a bit-serial, modular design. Here advantage is taken of commonality of coefficients to produce a compact linear array complex multiplier, cascadable to any desired coefficient size.

8.3.2 Short DFT machines

Another building block common to most Fourier transform machines is the short DFT. This is used in the construction of 'butterflies' of various radix (Rabiner and Gold, 1975). A radix-2 butterfly, so called because of its conventional graphic representation (Figure 8.2), realises either:

$$v = y + wx \quad \text{and} \quad u = y - wx$$

if decimating in time (DIT), or

$$v = y + x \quad \text{and} \quad u = w(y - x)$$

if decimating in frequency (DIF) (Rabiner and Gold, 1975).

Here y and x are complex inputs, v and u are complex outputs, and w is a complex weight of unity magnitude, known as a 'twiddle factor' (Gentleman and Sande, 1966) – realisable using OPERATOR Xmult. Decimation in time (or frequency) is so called because the input (or output) sequence is split into R sub-sequences of $1/R$ times the original length in the process of replacing an N -point transform by R of length N/R , R being the machine radix (Rabiner and Gold, 1975). We discuss this later.

The radix-2 butterfly performs a 2-point DFT, with a twiddle (or vector

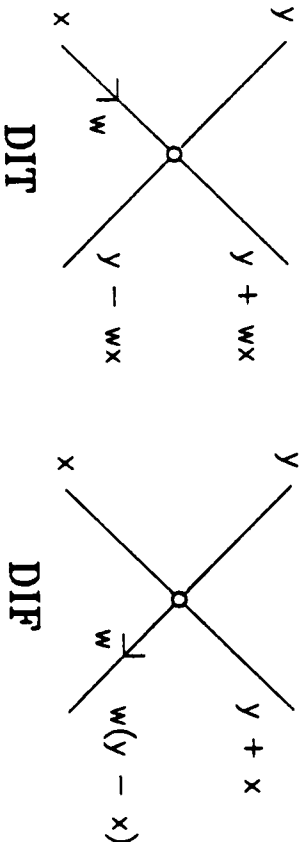


Fig. 8.2 Radix-2-DIT and DIF butterfly representations.

rotation) carried out on either the lower input (if DIT), or lower output (if DIF), *leg*. We have seen that an N -point DFT requires the generation of N distinct, equally-spaced unit vectors in the complex plane (recall the wheel-spokes analogy). Here $N = 2$, and the vectors are simply 1 and -1 . Thus no multipliers are required for a 2-point DFT. We need only add

$$e + jf = (a + c) + j(b + d)$$

and subtract

$$g + jh = (a - c) + j(b - d)$$

the two complex data inputs. Once again, we need to implement some complex arithmetic elements – call them 'Xadd' and 'Xsub'. Figure 8.3 shows OPERATOR Xadd.

OPERATOR Xadd [lat] ($c1$) $a, c, b, d \rightarrow e, f$
latency is lat

ADD [lat,0,0,0] ($c1$) $a, c, \text{GND} \rightarrow e, \text{NC}$
ADD [lat,0,0,0] ($c1$) $b, d, \text{GND} \rightarrow f, \text{NC}$

END

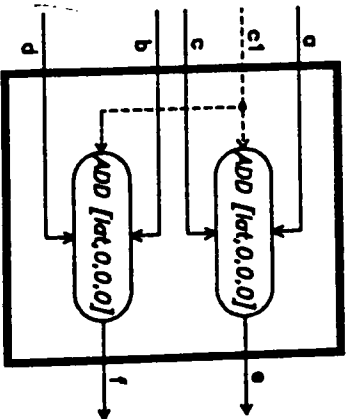


Fig. 8.3 Complex Addition OPERATOR 'Xadd'.

```
OPERATOR Xsub[lat](c1) a, c, b, d -> g, h
!latency is lat
```

```
SUBTRACT[lat,0,0,0](c1) a, c, GND -> g, NC
SUBTRACT[lat,0,0,0](c1) b, d, GND -> h, NC
```

END

We can now design a 2-point DFT machine, and call it 'Dft2' (Figure 8.4):

```
OPERATOR Dft2[lat](c1) a, c, b, d -> e, g, f, h
!latency is lat
```

```
Xadd[lat](c1) a, c, b, d -> e, f
Xsub[lat](c1) a, c, b, d -> g, h
```

END

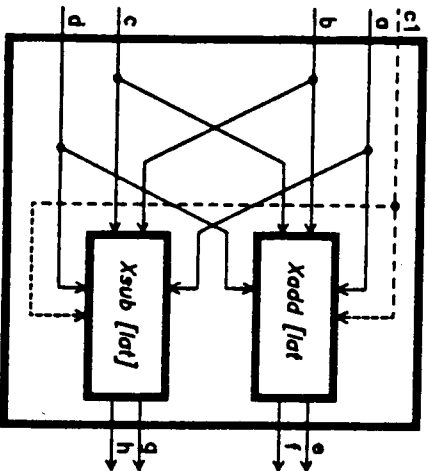


Fig. 8.4 2-point DFT OPERATOR 'Dft2'.

Here we have used the same parameter and node names in the declarations as in the instantiations. FIRST does not require this – however, it helps to clarify the mapping of 'algorithm' into FIRST source code. We leave the simulation of Dft2 until later.

Butterflies of any radix normally have twiddle-free top legs. Data on this leg must be delayed to compensate for the latency of any twiddles. Before constructing a radix-2 butterfly, we encapsulate Xmult and some delay to make a time-aligned twiddling block called 'Twiddle2', shown in Figure 8.5. The expression for compdel derives from the latency formulae for multiplier and adder in Xmult.

```
OPERATOR Twiddle2[coeff,lat](c1 -> c1o) rin1, rin2, iin1, iin2, wre,
wim -> rout1, rout2, iout1, iout2
!latency is compdel (see CONSTANT declaration below)
```

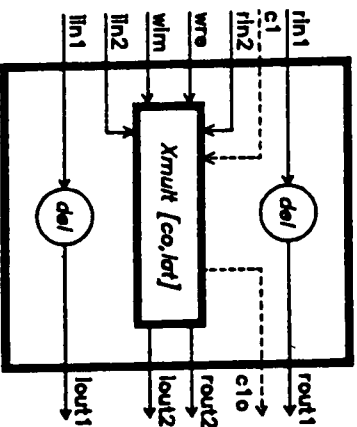


Fig. 8.5 Radix-2 twiddling OPERATOR 'Twiddle2'.

```
CONSTANT compdel = (3 * coeff)/2 + 2 + lat
BITDELAY[compdel] rin1 -> rout1
BITDELAY[compdel] iin1 -> iout1
Xmult[coeff,lat](c1 -> c1o) rin2, iin2, wre, wim -> rout2, iout2
;
```

END

It is now possible to construct both DIT and DIF radix-2 butterflies; called 'Rad2Dit' (Figure 8.6) and 'Rad2Dif' (Figure 8.7) respectively:

```
OPERATOR Rad2Dit[coeff,lat1,lat2](c1) rin1, rin2, iin1, iin2, wre,
wim -> rout1, rout2, iout1, iout2
!latency is compdel (see CONSTANT declaration in Twiddle2) + lat2
```

```
SIGNAL r1, r2, il, i2
CONTROL c11
```

```
Twiddle2[coeff,lat1](c1 -> c11) rin1, rin2, iin1, iin2, wre, wim -> r1,
r2, il, i2
Dft2[lat2](c11) r1, r2, il, i2 -> rout1, rout2, iout1, iout2
```

END

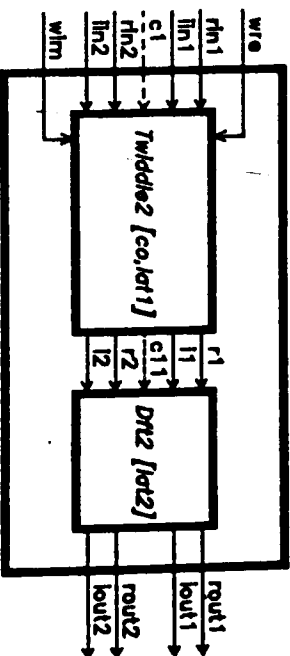


Fig. 8.6 Radix-2 DIT butterfly OPERATOR 'Rad2Dit'.

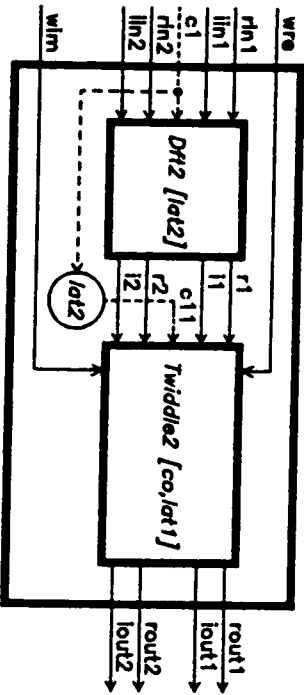


Fig. 8.7 Radix-2 DIF butterfly OPERATOR 'Rad2Dif'.

OPERATOR Rad2Dif[coeff,lat1,lat2](c1) rin1, rin2, iin1, iin2, wre,
wim-> rout1, rout2, iout1, iout2
!latency is lat2 + compdel (see CONSTANT declaration in Twiddle2)

SIGNAL r1, r2, i1, i2
CONTROL c11

Dft2 [lat2] (c1) rin1, rin2, iin1, iin2-> r1, r2, i1, i2
Twiddle2 [coeff,lat1] (c11-> NC) r1, r2, i1, i2, wre, wim-> rout1,
rout2, iout1, iout2

CBITDELAY [lat2] (c1-> c11)

END

These OPERATORS have time-aligned inputs and outputs (see Chapter 6). Whilst this makes them conceptually easier to handle later on in the design process, they contain delays which may impair their efficiency. It may be desirable to optimise these elements for a particular design.

Notice that we require a CBITDELAY in Rad2Dif, as we cannot take advantage of a control output from Dft2. Also the CBITDELAY primitive in Xmult is redundant here - a penalty of time-aligning.

8.3.3 A radix-4 DFT

The radix-2 machine Dft2 performs a parallel DFT on 2 points of complex data. It requires no multipliers because internal rotations are through 0 and 180 degrees only, which can be accomplished by adding and subtracting. The step to a radix-4 butterfly, which involves angular shifts of 0, 90, 180 and 270 degrees - none of which require multipliers - is a logical one. Furthermore, we can build a 2 x 2 array of Dft2 OPERATORS to implement 'Dft4', a little 4-point FFT machine. This is depicted in Figure 8.8.

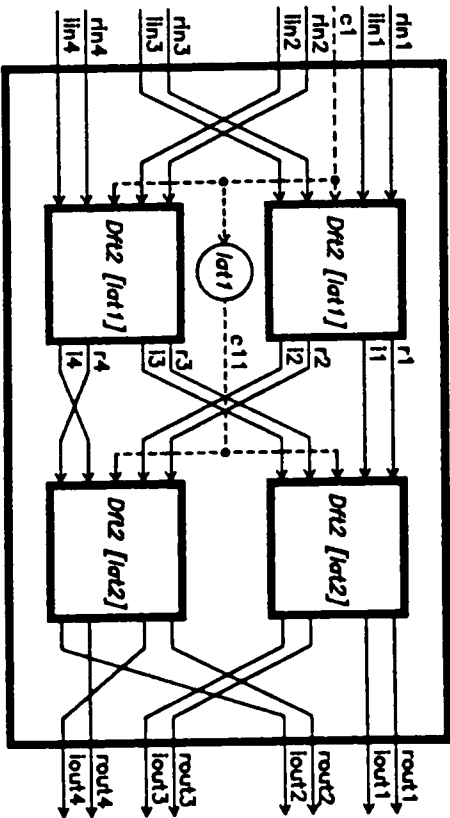


Fig. 8.8 4-point DFT OPERATOR 'Dft4'.

OPERATOR Dft4 [lat1,lat2] (c1) rin1 THROUGH 4, iin1 THROUGH 4 -
-> rout1 THROUGH 4, iout1 THROUGH 4

!latency is lat1 + lat2

SIGNAL, r1 THROUGH 4, i1 THROUGH 4
CONTROL c11

Dft2 [lat1] (c1) rin1, rin3, iin1, iin3-> r1, r2, i1, i2
Dft2 [lat1] (c1) rin2, rin4, iin2, iin4-> r3, r4, i3, i4
Dft2 [lat2] (c11) r1, r3, i1, i3-> rout1, rout3, iout1, iout3
Dft2 [lat2] (c11) r2, r4, i2, i4-> rout2, rout4, iout2, iout4
CBITDELAY [lat1] (c1-> c11)

END

Note the use of the shorthand THROUGH statement (Chapter 3) to avoid naming long node lists explicitly. The delays lat1 and lat2 are parameterised for the present, and the required control delay is built in to synchronise the second column of processors. The tortuous routing around the last Dft2 operator is to effect a premultiplication by -j, an internal 'twiddle' in the 4-point FFT machine.

Simulation

The structure has now become sufficiently complicated to merit simulation. We must create a simulation environment for Dft4. This entails creation of a SYSTEM containing a CHIP with Dft4 connected to a CONTROLGENERATOR primitive, and of course the inputs and outputs of Dft4 connected to pads.

Here we are using 'perfect' arithmetic, i.e. addition and subtraction, in which no numerical degradation occurs (provided overflow is avoided), and the input samples will not suffer from quantisation, being either zero or some constant amplitude. Therefore we need not be concerned with quantisation effects.

The OPERATOR Dft4 is a 4-point parallel DFT machine. Let us input parallel signal blocks of length 4 samples, containing single frequency components of 0, $\pi/2$, π and $3\pi/2$ radians/sample period, each starting with zero phase, and with some space between. In Figure 8.9 we see the response to each frequency appearing only in the correct single output wire (or bin), and the four distinct phases of each sample block.

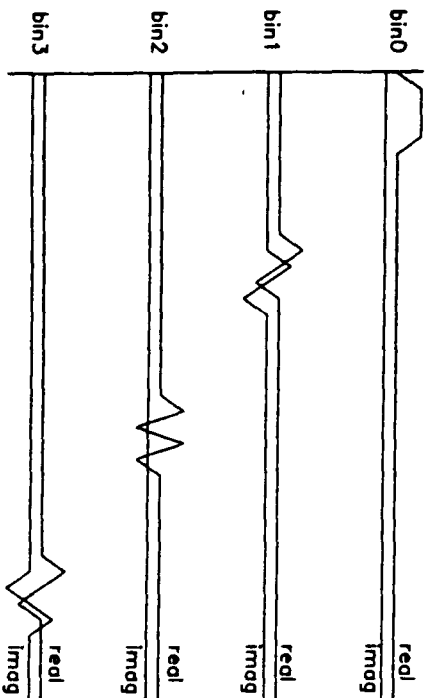


Fig. 8.9 Simulation of 'Dft4', showing complex amplitude response to stimulus at distinct output frequencies.

8.3.4 A radix-4 butterfly

To construct a radix-4 butterfly, we must add twiddles, just as in the case of radix-2. All but the top leg of the butterfly contain twiddles. Here we require a block of 3 Xmult operators and the compensating delays for the top (untwiddled) wire. We already have the components for this – Twiddle2 and Xmult. So the code for 'Twiddle4' (Figure 8.10) is:

```
OPERATOR Twiddle4[coeff,lat](c1->c10) rin1 THROUGH 4,
    rin1 THROUGH 4, wre1 THROUGH 3, wim1 THROUGH 3->-
    rout1 THROUGH 4, iout1 THROUGH 4
    //latency is constdel (as defined in Xmult) + lat

    Twiddle2[coeff,lat](c1->NC) rin1, rin2, iin1, iin2, wre1, wim1->-
    rout1, rout2, iout1, iout2
    Xmult[coeff,lat](c1->c10) rin3, rin3, wre2, wim2-> rout3, iout3
    Xmult[coeff,lat](c1->NC) rin4, rin4, wre3, wim3-> rout4, iout4
    END
```

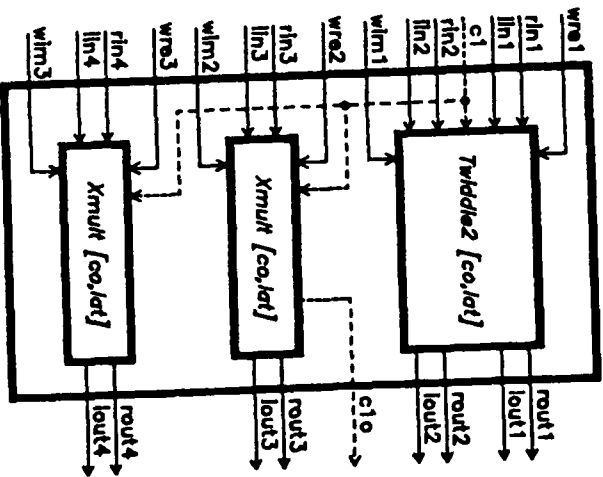


Fig. 8.10 Radix-4 twiddling OPERATOR 'Twiddle4'.

and 'Rad4Dif' (Figure 8.11) follows on easily:

```
OPERATOR Rad4Dif[coeff,lat1,lat2,lat3](c1->c10) rin1 THROUGH 4,
    rin1 THROUGH 4, wre1 THROUGH 3, wim1 THROUGH 3->-
    rout1 THROUGH 4, iout1 THROUGH 4
    //latency is constdel (as defined in Xmult) + lat2

    SIGNAL r1 THROUGH 4, i1 THROUGH 4
    CONTROL c11

    Dft4[lat2,lat3](c1) rin1 THROUGH 4, iin1 THROUGH 4->-
    r1 THROUGH 4, i1 THROUGH 4
    Twiddle4[coeff,lat1](c11->NC) r1 THROUGH 4, i1 THROUGH 4,
    wre1 THROUGH 3, wim1 THROUGH 3->-
    rout1 THROUGH 4, iout1 THROUGH 4
    CBITDELAY[lat2 + lat3](c1->c11)
```

END

'Rad4Dif' is constructed by cascading the two components of Rad4Dif in reverse order.

8.3.5 Windowing

The DFT operates on blocks of data. A block of data can be imagined as an infinite time series pointwise multiplied with a unit rectangular window function, that is a function which is zero outside some range, else unity. Since

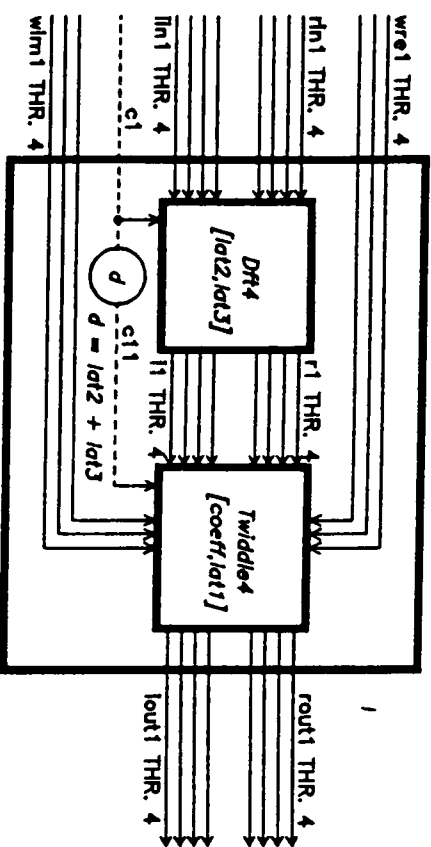


Fig. 8.11 Radix-4 DIF butterfly OPERATOR 'Rad4Dif'.

pointwise multiplication in the time-domain is equivalent to convolution in the frequency-domain (Rabiner and Gold, 1975), we find that the DFT output is convolved with the spectrum of a rectangular window, whose amplitude function is of the $\sin(x)/x$ type (Harris, 1978). Although it has value zero at all discrete observation frequencies, and accordingly does not affect on-bin components, this function has a symmetrical series of off-axis peaks (sidelobes).

A related problem in analysis of finite length sequences is that any signal components which are not periodic in the block length exhibit discontinuities at the boundaries of the observation, resulting in what is known as 'spectral leakage'. When a non-periodic component is convolved with the window spectrum, the result is a spread of energy over the range of bins. The normal solution to this problem is to window the input sequence with some function whose sidelobe behaviour in the frequency domain improves on that of the rectangular window.

All window design is an application-dependent compromise. Harris (1978) summarised many of the alternatives, and Nuttall (1983) reduced the design criteria to two parameters – sidelobe decay and mainlobe to peak-sidelobe ratio.

For now we will consider the design of a programmable window processor – we can worry about window functions later. The twiddles preceding the first column are trivial in the case of a DIT FFT, and non-existent in the case of DIF, so we can take advantage of this by building in the window multipliers instead of the twiddles. The window function has identical weights for real and imaginary signal components, leading to some sharing of coefficients. All that is required is a real weighting OPERATOR 'Xweight2' (Figure 8.12) which performs four real multiplications with two shared coefficients on two complex data points:

```
OPERATOR Xweight2[wcoeff](c1 -> c10) rin1, rin2, iin1, iin2, w1,
w2 -> rout1, rout2, iout1, iout2
latency is 3/2 * wcoeff + 2
```

```
MULTIPLY [1, wcoeff, 0, 0] (c1 -> c10) rin1, rin2, w1 -> rout1, NC
MULTIPLY [1, wcoeff, 0, 0] (c1 -> NC) rin2, w2 -> rout2, NC
MULTIPLY [1, wcoeff, 0, 0] (c1 -> NC) iin1, w1 -> iout1, NC
MULTIPLY [1, wcoeff, 0, 0] (c1 -> NC) iin2, w2 -> iout2, NC
```

END

Again we defer simulation – the effects of window type, quantisation, etc., can only be properly studied in the context of a real system simulation. The first-column processor 'Window2' will consist of Xweight2 and Dft2 in cascade:

```
OPERATOR Window2[wcoeff, lat] (c1) rin1, rin2, iin1, iin2, w1,
w2 -> rout1, rout2, iout1, iout2
latency is 3/2 * wcoeff + 2 + lat
```

```
SIGNAL r1, r2, i1, i2
CONTROL c11
```

```
Xweight2[wcoeff](c1 -> c11) rin1, rin2, iin1, iin2, w1, w2 -> r1, r2,
i1, i2
Dft2[lat](c11) r1, r2, i1, i2 -> rout1, rout2, iout1, iout2
```

END

The OPERATORs Window2 (not illustrated) and Rad2Dit are functionally dissimilar, but topologically identical.

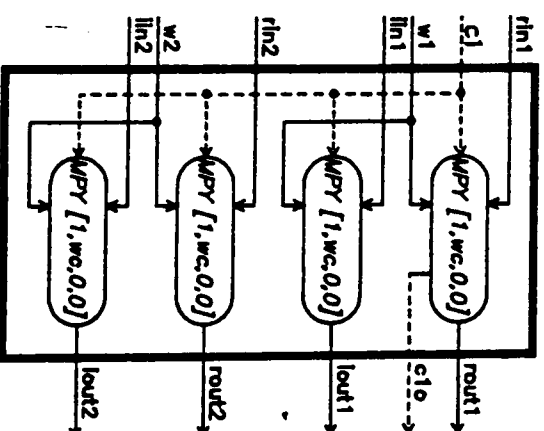


Fig. 8.12 Weighting OPERATOR 'Xweight2'.

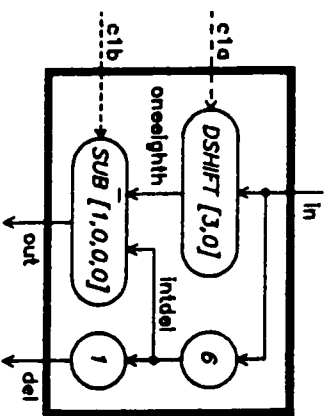


Fig. 8.13 7/8 scaling OPERATOR 'SevenEighths'.

8.3.6 Complex-to-magnitude conversion

In many cases only the magnitude output of an FFT processor is required. The magnitude of a complex number is the root of the sum of the squares of its components. There exist many approximation algorithms, summarised by Filip (1976), the simplest of which has already been introduced in Chapters 1 and 3. We structure the OPERATOR to show explicitly how the fraction 7/8 is realised. OPERATOR SevenEighths (Figure 8.13) is the time-aligned

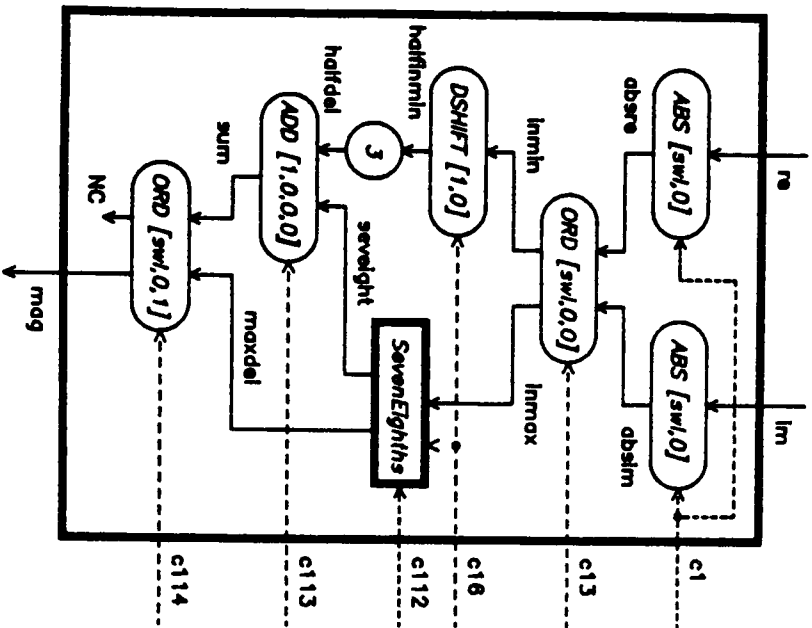


Fig. 8.14 OPERATOR 'ComplexToMagnitude'.

component which executes the 7/8 function. The two BITDELAYS compensate for the latencies of the DSHIFT and SUBTRACT primitives.

OPERATOR SevenEighths (c1a, c1b) in -> out, del
latency is 7

SIGNAL oneighth, intdel
DSHIFT[3,0](c1a) in -> oneighth
SUBTRACT[1,0,0,0](c1b) intdel, oneighth, GND -> out, NC
BITDELAY[6] in -> intdel
BITDELAY[1] intdel -> del

END

The full OPERATOR ComplexToMagnitude (Figure 8.14) contains OPERATOR SevenEighths. Note that use has been made of input predelay in the final ORDER primitive, allowing time aligning of SevenEighths outputs at no expense in waterfront. ComplexToMagnitude is an example of an OPERATOR which contains a mixture of elements at the same level and at the adjacent lower level in the hierarchy. FIRST allows infinite nesting of those levels of hierarchy (OPERATOR and SUBSYSTEM) which have no direct physical significance.

OPERATOR ComplexToMagnitude[swl](c1, c13, c16, c112, c113, c114) re,
im -> mag

latency is 3 * swl + 17

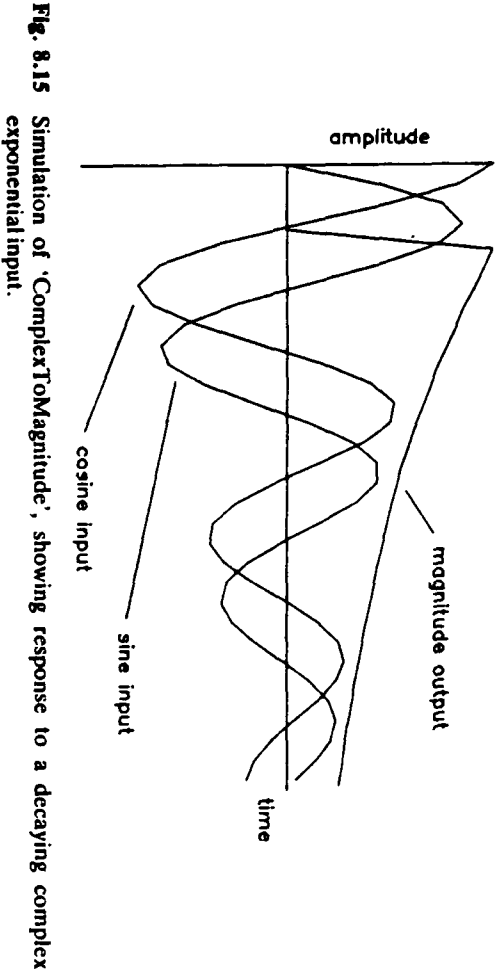
SIGNAL absre, absim, inmax, immin, halfmnin, seveight, sum,
maxdel, halfdel
ABSOLUTE[swl,0](c1) re -> absre
ABSOLUTE[swl,0](c1) im -> absim
ORDER[swl,0,0](c13) absre, absim -> inmax, immin
DSHIFT[1,0](c16) immin -> halfmnin
ADD[1,0,0,0](c113) seveight, halfdel, GND -> sum, NC
SevenEighths(c16, c112) inmax -> seveight, maxdel
BITDELAY[3] halfmnin -> halfdel
ORDER[swl,0,1](c114) sum, maxdel -> mag, NC

END

Simulation

We use the ComplexToMagnitude OPERATOR to evaluate the amplitude of a complex signal. We input a signal of the form $\exp(-(a + jb)t)$, where $a < 1$. This signal consists of a complex sinusoid with an amplitude envelope $\exp(-at)$, which we may extract with ComplexToMagnitude. Figure 8.15 shows the result. Note the delay in response due to the OPERATOR latency.

With this addition to the toolkit, we may consider it complete enough to start looking at some DFT systems.



8.4 FFT machines

For tutorial purposes, we commenced design in a bottom up manner; now we shall start to specify systems from the top down. The DFT toolkit is ready for the construction of an actual FFT machine. We have a vector rotator, and radix-2 and 4 DFT machines, which may be used to build the butterflies which are germane to the FFT computation. We also have windowing and magnitude conversion blocks for front and back end processing.

Before we can build an FFT machine, however, we must know a little more about the operation of the FFT. Those readers who are already familiar with the FFT, or indeed those who don't wish to be, may skip the next section. Alternative derivations and explanations abound in the literature (see References).

8.4.1 Indexing and shuffling in the FFT

It is instructive to investigate two types of shuffling transformation which occur naturally in the FFT. The first is the ideal, or perfect shuffle (Stone, 1971). A radix R, order M perfect shuffle σ_M has the effect of interleaving elements of a data sequence of length $N = R^M$ in such a manner as to bring together elements spaced apart by P samples, where $P = N/R = R^{M-1}$. The second shuffle is the digit reversing shuffle – the result of repeated application of M perfect shuffles of descending order.

Having defined R as a radix,[†] we may index data sequences in terms of R, giving an M-digit index. These shuffles serve to transform the indices of data sequences – the latter in a rather obvious manner. The effect of a perfect shuffle of order m is to circular left-shift the m least-significant digits of the index (Stone, 1971). Note that significance, or weight, of digits remains constant throughout transforms.

[†] Here R is constant throughout the transform, although mixed-radix transforms are possible (Rabiner and Gold, 1975).

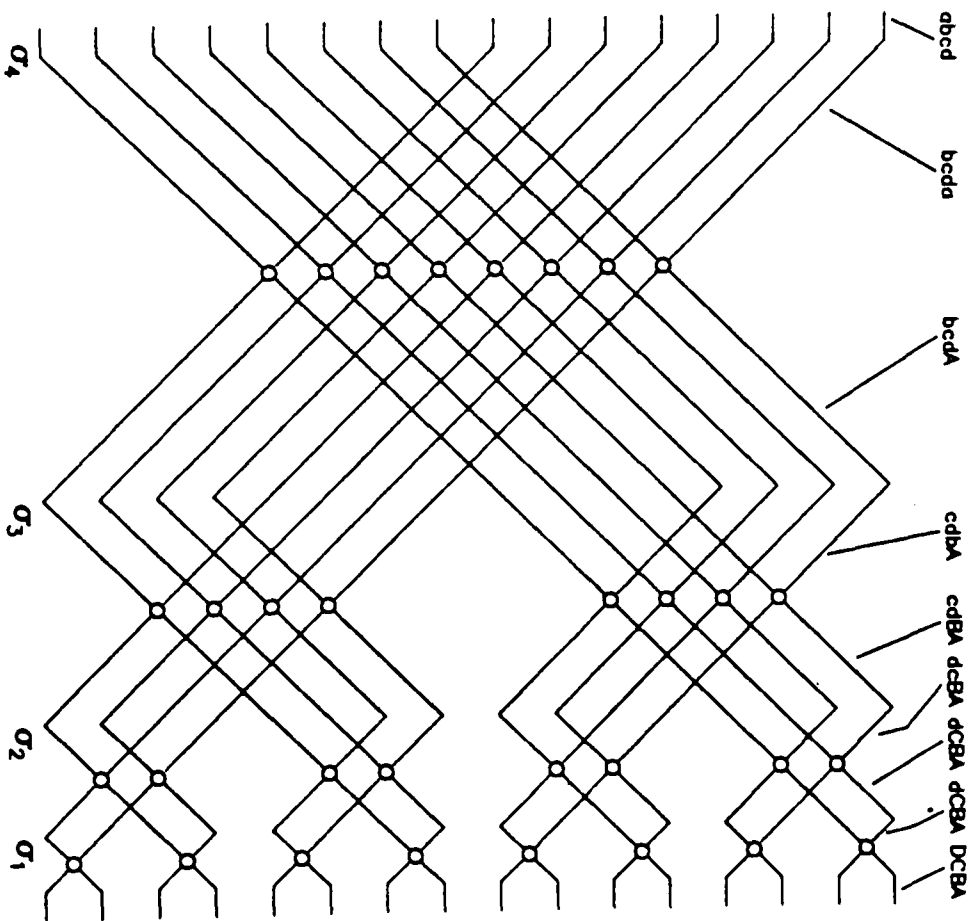


Fig. 8.16 16-point radix-2 Cooley-Tukey FFT network, showing index transformations across the array.

Figure 8.16 shows a network for computing the Cooley-Tukey FFT (Cooley and Tukey, 1965), with $R = 2$ and $M = 4$. The key property of this network is that each point in the input sequence is able to communicate through a radix-R, depth-M tree structure with each point in the output sequence. We refer to this property as 'universal connectivity'.

Digit reversal

Although the input sequence is correctly ordered, the topology of the network effectively perfect shuffles the sequence, bringing together elements spaced apart by $P = 8$ samples into groups of $R = 2$ before the first column of the machine. After the first column, we have R groups of P/R samples, which are in turn perfect shuffled by the network topology. The order of the perfect shuffle here is one less than before. The effect on the data index of repeated, decreasing order perfect perfect shuffles at the column interfaces is shown along the top of Figure 8.16.

It is important to note that the digit reversing shuffle inherent in the FFT is a result of the universal connectivity of the network, and nothing else. If correctly ordered output is required, a further digit reversing shuffle must be performed at some stage in the transform.

The butterflies have no topological effect (they are horizontally symmetrical), and only serve to alter the interpretation of the stage digit from time to frequency indexing (see lower and upper case letters in Figure 8.16). Each butterfly carries out a local R-point DFT on a time sub-sequence, summing over the stage digit, for all combinations of the other digits.

8.4.2 The Cooley–Tukey FFT algorithm

So far we have described a network which offers a constant, minimum length path from any input point to any output point, given a radix R. We have yet to implement the arithmetic which occurs in the nodes and branches of the network, although we hinted at its function.

The FFT can be described in theory as successively fracturing one-dimensional sequences into equivalent two-dimensional sequences (Gold and Blahut, 1973). This can be done in several ways, the two most common of which are decimation in time (DIT) and in frequency (DIF) (Rabiner and Gold, 1975). As we develop the two algorithms below, their similarity will become obvious.

Either DIT or DIF algorithms may be implemented on the network of Figure 8.16. A right-to-left mirror image of Figure 8.16 will produce correctly ordered output from digit-reversed input, again using either DIT or DIF. The algorithms and topologies merely decide the location and value of twiddle factors. The networks exhibit isomorphism (they are all different layouts of the same radix-R, universally connective tree structure). From a functional point of view they are identical.

Using the shorthand notation

$$z'_m = \sum_{i=0}^R R^i z_i$$

we index the input series $x(n)$ and output series $X(k)$ with

$$n = \sum_{m=0}^{M-1} R^m d_m = d'_{M-1} \quad k = \sum_{m=0}^{M-1} R^m c_m = c'_{M-1}$$

With this indexing notation in mind, we may now develop the FFT algorithm. We start with the DFT equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

Expressing n in terms of its component digits d_m allows splitting of the summation into M sums, each over R terms.

$$X(k) = \sum_{d_0=0}^{R-1} \sum_{d_1=0}^{R-1} \dots \sum_{d_{M-1}=0}^{R-1} x(n) W_N^{nk} \quad 8.5$$

The advantage in splitting into a nest of sums over R terms is that, if a rotation factor can be found which is periodic in R , the basic computational unit at each network node can be realised by an R-point DFT machine, which we have already seen to be a fast and compact unit (at least for $R = 2$ and 4). The mechanism of the FFT algorithm may now be revealed, as we manipulate the W term.

As a completely general algorithm development would lead to some rather unwieldy expressions, we shall develop the DIT and DIF algorithms in parallel. In the text, references to variables will appear in the form DIT (DIF). Equations are in pairs, DIT first.

First of all we separate the component digits of n (k).

$$W_N^{nk} = W_N^{d_0 k} W_N^{R d_1 k} \dots W_N^{R^{M-1} d_{M-1} k}$$

$$W_N^{nk} = W_N^{R^{M-1} c_{M-1} n} W_N^{R^{M-2} c_{M-2} n} \dots W_N^{c_0 n}$$

then we use the periodicity property to cancel powers of R in the exponent of W at each stage, thereby 'coarsening' its angular range where possible.

$$W_N^{nk} = W_N^{d_0 k} W_R^{d_1 k} \dots W_R^{d_{M-1} k}$$

$$W_N^{nk} = W_R^{c_{M-1} n} W_R^{c_{M-2} n} \dots W_R^{c_0 n}$$

Next we resolve k (n) into its component digits, noting that only those digits whose weight is less than the periodicity of W are effectively non-zero (again due to the periodicity property):

$$W_N^{nk} = W_N^{d_0 c'_{M-1}} W_R^{d_1 c'_{M-2}} \dots W_R^{d_{M-1} c'_0}$$

$$W_N^{nk} = W_R^{c_{M-1} d_0} W_R^{c_{M-2} d_1} \dots W_R^{c_0 d_{M-1}}$$

and finally we separate terms involving d (c) into a part which is periodic in R , and a part which isn't:

$$W_N^{nk} = W_R^{d_0 c_{M-1}} W_N^{d_0 c'_{M-2}} W_R^{d_1 c'_{M-2}} W_R^{d_1 c'_{M-3}} \dots W_R^{d_{M-1} c'_0}$$

$$W_N^{nk} = W_R^{c_{M-1} d_0} W_R^{c_{M-2} d_1} W_R^{c_{M-2} d_0} W_R^{c_{M-2} d_1} \dots W_N^{c_0 d_{M-2}} W_R^{c_0 d_{M-1}} \quad 8.6$$

Substituting equation (8.6) into equation (8.5) gives the DIT expression

$$X(k) = \sum_{d_0=0}^{R-1} W_R^{d_0 c_{M-1}} W_N^{d_0 c'_{M-2}} \sum_{d_1=0}^{R-1} W_R^{d_1 c'_{M-2}} W_R^{d_1 c'_{M-3}} \dots \sum_{d_{M-1}=0}^{R-1} x(n) W_R^{d_{M-1} c'_0} \quad 8.7$$

and the DIF expression

$$X(k) = \sum_{d_0=0}^{R-1} W_R^{c_{M-1} d_0} W_R^{c_{M-2} d_1} \sum_{d_1=0}^{R-1} W_R^{c_{M-2} d_0} W_R^{c_{M-2} d_1} \dots W_N^{c_0 d_{M-2}} \sum_{d_{M-1}=0}^{R-1} x(n) W_R^{c_0 d_{M-1}} \quad 8.8$$

The reader should attempt to establish a link between these equations and the topology of the FFT network. At each stage of the machine, a summation is performed over the associated index digit. This summation is performed for all combinations of the other digits, leading to an N/R by M array of processors. Each butterfly column effectively transforms the interpretation of the stage digit from the time domain to the frequency domain. As butterfly inputs and outputs are characterised by the stage digit, the 'span' of each column, i.e. the distance between these elements, equals the weight of the stage digit.

The rotation factors which are not periodic in R are the twiddle factors employed in network branches of the transform. Those which are periodic in R serve to complete the DFT definitions, whose necessity was underlined earlier. Wherever $d\{c\} = 0$, the twiddle factor corresponds to a multiplication by unity, and is omitted – hence the twiddle free input {output} top leg on all butterflies.

Whatever the geometry used, any twiddle factor may be evaluated by tracing connections to any elements in transform input and output sequences, and evaluating at the relevant stage in equation (8.7) {(8.8)}. Finally, note that both transforms render twiddle factors unnecessary outside the DFT columns – a fact which is often overlooked in the literature.

8.4.3 The constant-geometry algorithm

Many other topologies exist to satisfy the needs of FFT users. Constant geometry algorithms (Rabiner and Gold, 1975) allow one column of processors to be repeated across the array, thus allowing a standard part (whether a chip or a circuit board) to be used throughout the transform. The cost of this modular approach is the inclusion of trivial rotations by W^0 before the first (in DIT realisations) or after the last (in DIF) column, or some extra control and switching circuitry for their obviation.

To implement a constant-geometry algorithm, we must ensure that the spacing between elements in sequences (i.e. span) remains constant across the array. This may be achieved by including an order- M perfect shuffle after each stage. Figure 8.17 (a) and (b) illustrates (explicitly and implicitly) the right-hand side of one such stage, showing the action of the perfect shuffle. Spans are increased by a factor of R , which cancels the reduction by R inherent in the Cooley–Tukey FFT. The effect on DIT twiddles is merely to perfect shuffle them as well – this has the beneficial effect of ordering them correctly down the columns (cf. digit reversal in the Cooley–Tukey DIT FFT).

Note that in this network M extra shuffles are performed. They have no overall effect, as this corresponds to a complete circular shift of the index back to its original state. Given N and R , all FFT networks are isomorphous and differ only in the employment of twiddles and shuffles throughout. For instance, normally-ordered output can be produced from the mirror image of the network implied by Figure 8.17, if the input sequence index is digit-reversed. The perfect shuffles of the network, both inherent (for universal connectivity) and additional (for constant geometry), are in this case inverse shuffles, or 'unshuffles'.

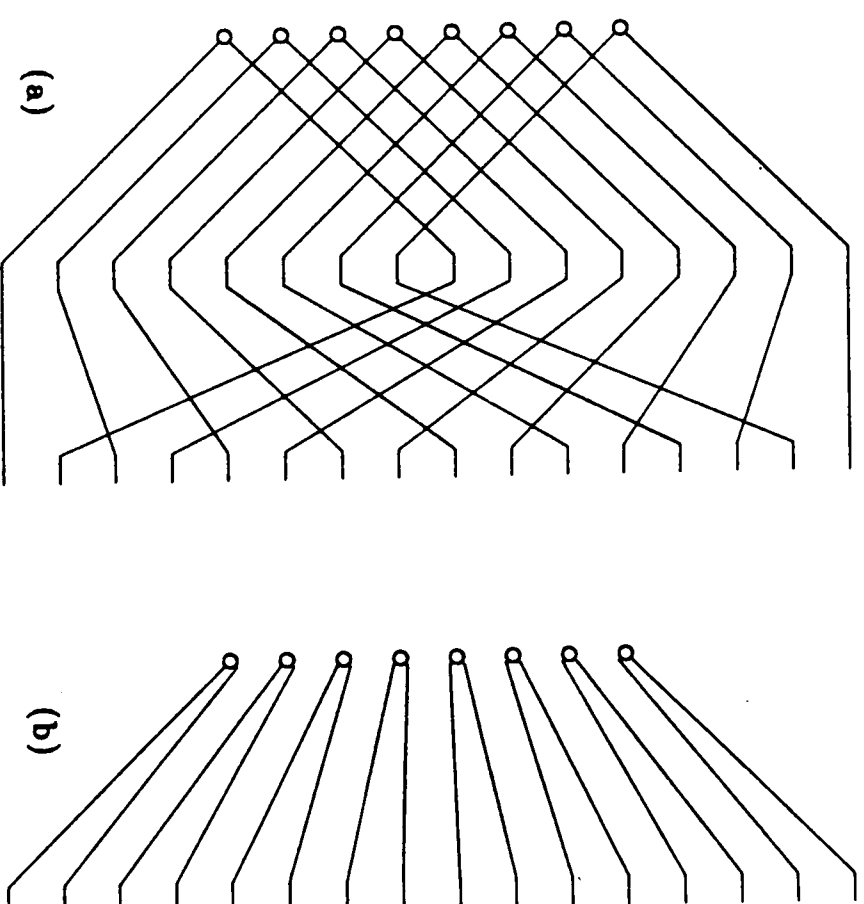


Fig. 8.17 Right-hand side of constant-geometry FFT stage, showing (a) explicitly and (b) implicitly the action of the additional order- M perfect shuffle.

We now have the tools and the knowledge to proceed with the design of an FFT machine.

8.5 The full array FFT machine

Our first exercise is a 16-point full array FFT machine with programmable input data windowing, implemented on the inverse constant-geometry network just described and operating on 12-bit data words. We include a hard-wired shuffle on the inputs, and arrange for the output of the machine to appear in magnitude form.

The heart of an N -point FFT machine may be represented at the highest level by a block with N time-domain inputs and N frequency-domain outputs. However, the input signal is normally a word-stream (e.g. the output from an A/D converter), and so we employ a corner-turning memory to load a block of N samples, and output each sample on its own wire (or wire pair if the input signal is complex). Although the inverse of this process can be performed on the transformer output, converting N parallel words into a block of N

contiguous frequency-domain samples on one wire, we choose to leave the transform output in word-parallel form. Figure 8.18 shows the high level system plan.

The corner-turning memory has not been studied as part of our methodology, but it is clear that this is no more than a sophisticated input primitive. The corner-turning memory is the word-stream equivalent of a serial-to-parallel converter (SIFO) acting on a bit-stream. Garverick and Pierce (1983) report the design of such a memory. For the purpose of this study we restrict ourselves to the main computational engine.

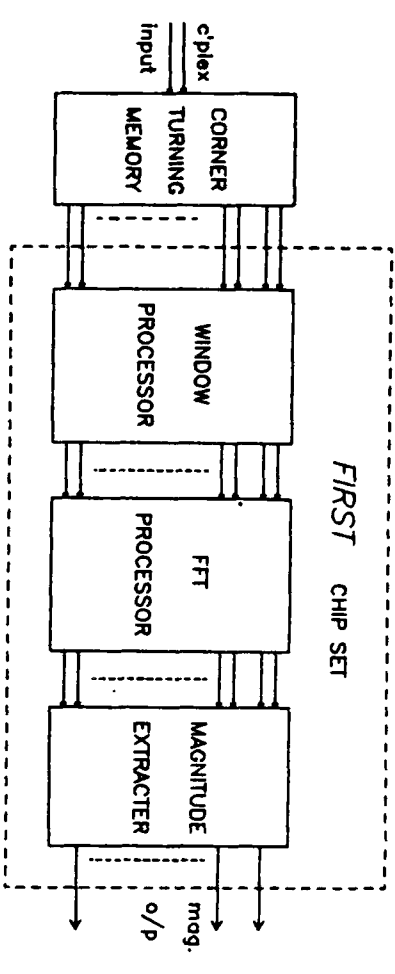


Fig. 8.18 High-level system plan of parallel FFT machine.

8.5.1 Preliminary architectural decisions

We wish to realise a 16-point FFT machine, containing a minimal number of different chip types. For the purpose of simulation, we shall create notional chips for generation of constants (window coefficients and twiddle factors – see below), and for the input corner-turning memory. The heart of the machine, in slightly expanded form, will consist of an array of 8 by 4 radix-2 ‘butterfly’ processors. The 16-point complex output will feed a column of complex-to-magnitude converters.

As we saw earlier, twiddle factors in the first column are not required, and the correspondingly reduced butterfly can be combined with the windowing coefficients and the data shuffling to form a first-column processor. The remaining three columns then consist of regularly connected butterfly processors. The block diagram for the target system is detailed in Figure 8.19.

Word ranges and precisions

Due to the additions and subtractions inherent in the butterfly, word growth occurs in the FFT as computation proceeds across the array. As this array consists of 4 columns, we need 4 guard bits on top of our system wordlength to accommodate word growth. This is a simpler approach, although one very slightly inferior in terms of noise performance, to any form of truncation within the process (Oppenheim and Weinstein, 1972). We also must observe the

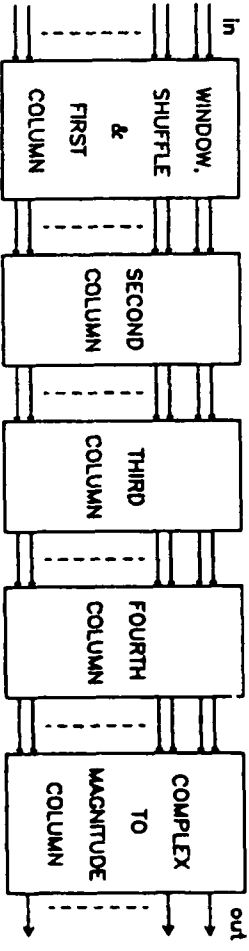


Fig. 8.19 Block diagram of parallel FFT machine.

requirement for the customary 2 guard bits in the data input to all multipliers, which ensure their correct operation.

The fact that the final butterfly stage follows the final multiplier in the data path allows us to ‘steal’ one of the two guard bits, potentially reducing the system wordlength swl by one bit. Although the magnitude converter can cause growth by one bit, given full-scale signals on both real and imaginary inputs, we feel that this is a sufficiently unlikely situation to warrant an extra bit of system wordlength. Thus swl must be at least five bits greater than the input signal wordlengths.

We present here a fully concurrent implementation of the FFT algorithm. The transform rate r_t for the block FFT is thus equal to the word rate, which we recall is the ratio of process clock rate r_p (8 MHz here) to system wordlength, i.e.

$$r_t = \frac{r_p}{s + 5}$$

The latency of a butterfly processor will depend on the resolution of the coefficients. This resolution is a fundamental system parameter. As our transform is a very short one (the wheel spokes are sparse), a considerable amount of coefficient inaccuracy can be tolerated. In any case, the 3% tolerance of the magnitude converters introduces a considerable amount of noise into the output. As 8-bit multipliers are physically compact, we select an 8-bit coefficient wordlength.

For a 8-bit coefficient, the minimum butterfly latency is 16 bits, the objects in the data path being an 8-bit multiplier (latency 14 bits), and two adders or subtractors (latency a minimum of 1 bit each). If, as seems likely, a chip boundary is to be included, this figure rises to 17. Happily, a 12-bit input signal wordlength requires at least a 17-bit system wordlength, as explained earlier. For a system wordlength of 17 and a bit-rate of 8 MHz, the word rate will be 471 kHz. Assuming this is adequate, we set the system wordlength at 17 bits, thereby forcing word synchronism between columns as well as rows in the array, which greatly simplifies control.

Thus the 16-point demonstration machine will perform block transforms at 471 kHz, on data sampled at 7.53 MHz. If this data rate seems unspectacular, it should be noted that throughput here is proportional to transform length.

Thus, by merely instantiating more identical processors, extending the rows and columns of the machine according to their formulae given above, we could commission a 256-point FFT processor capable of transforming data sampled at frequencies exceeding 80 MHz. In this case, the 8-bit coefficient word would severely degrade numerical accuracy, and an increase to 12 bits might be called for. The cost of this ploy would be increased chip area and transform latency, and the loss of column synchronism.

8.5.2 A twiddle factor strategy

We are using a fractional two's complement number system to implement twiddle factors, to 8-bit accuracy. This approach suffers from the inability to express unity, which is outside the range of the number system. Most rotations suffer from inaccuracy due to quantisation errors, but we can always choose a complex coefficient which is within 0.707 of a bit of its correct value in the complex plane (consider any point within a unit square, and its distance from the nearest corner of the square). When attempting to represent unity (or for that matter j), however, we must accept a constant 1-bit error in the representation, which notably degrades accuracy at short coefficient lengths. Alternatively, we can seek a method of avoiding unity in either real or imaginary coefficients, noting that the number -1 is exactly represented in fractional two's complement.

The transform is a parallel one, and reads all elements of its input vector simultaneously. In that case, it is a simple matter to reverse the input sequence to the transform, thereby replacing the sequence by its conjugate (the real part is an even function, the imaginary part odd). This conjugation can be cancelled by replacing all twiddle factors by their conjugates. Although this means that the transform output appears in conjugate form, this phase error is of no concern as we are extracting output magnitude. Commonality of real and imaginary twiddle coefficients has now increased, and the number -1 is no longer required as a coefficient.

We wish to avoid unity (not -1), so the next step is to negate the conjugate twiddle factors. The effect of this step can be immediately cancelled by adding instead of subtracting, and vice versa, in OPERATOR Dft2. Rather than disturb the low-level OPERATOR Dft2, which is useful in its own right, we create the modified OPERATOR Dft2swap, which we subsequently use in Rad2Dit and Window2. We are making one declarational change, and two instantational changes, but only from the point of view of renaming constituent elements of OPERATORS. This means that the topology of the machine remains undisturbed.

To sum up, by reversing the input sequence, replacing all twiddle factors by their negative conjugate, and effectively swapping the adds and subtracts in the 2-point DFTs, we may maximise coefficient commonality and represent unity in exact manner. Such a strategy is only feasible in parallel transforms, and only necessary where coefficient resolution is poor.

8.5.3 Implementation

We have completed the design of arithmetic elements to be employed, and have made the decisions on numerical issues. Now we can implement the remainder of the machine.

As stated earlier, our architecture is based on the use of regular columns of processors. This immediately suggests a SUBSYSTEM corresponding to each column. The machine then consists of one windowing and shuffling column, three butterfly columns, and one complex-to-magnitude column in cascade.

Estimates on chip sizes, based mostly on multiplier content, indicate that for a 5 μm nMOS process, one butterfly processor will fit on a chip, one windowing processor likewise, while two of the magnitude processors may share a chip. This means that the first column, the three central columns and the final column each contain 8 chips, making 40 chips in all. (It is reasonable to assume that each of these columns could fit on one chip using a more advanced technology with around 2 μm feature sizes.) We now formally specify the three CHIPS.

The CHIPS

To create the CHIP Bfly2 we have to encapsulate Rad2Dit in a CHIP, and assign values to the various parameters using the CONSTANT statement.

```
CHIP Bfly2 (pc1) prin1, prin2, piin1, piin2, pwre, pwim -> prout1,
          prout2, piout1, piout2
```

latency is 17 bits

```
SIGNAL rin1, rin2, iin1, iin2, wre, wim, rout1, rout2, iout1, iout2
CONTROL c1
```

```
PADIN (pc1 -> c1) prin1, prin2, piin1, piin2, pwre, pwim -> rin1,
          rin2, iin1, iin2, wre, wim
PADOUT rout1, rout2, iout1, iout2 -> prout1, prout2, piout1, piout2
PADORDER VDD, pc1, prin1, prin2, piin1, piin2, GND, pwre,
          pwim, CLOCK, prout1, prout2, piout1, piout2
```

```
CONSTANT coeff = 8, lat1 = 1, lat2 = 1
```

```
Rad2Dit[coeff, lat1, lat2] (c1) rin1, rin2, iin1, iin2, wre, wim -> rout1,
          rout2, iout1, iout2
```

END

Plate 3 is a photomicrograph of CHIP Bfly2 (here parameter lat2 = 2, and coeff = 12). Such a device might form a computational node in a much larger transform, e.g. 256-point. The CHIPS 'Window' and 'Magnitude' are equally easy to implement – Window contains one Window2 OPERATOR, and Magnitude two ComplexToMagnitude OPERATORS. As before, this task is merely one of encapsulation and parameter assignment.

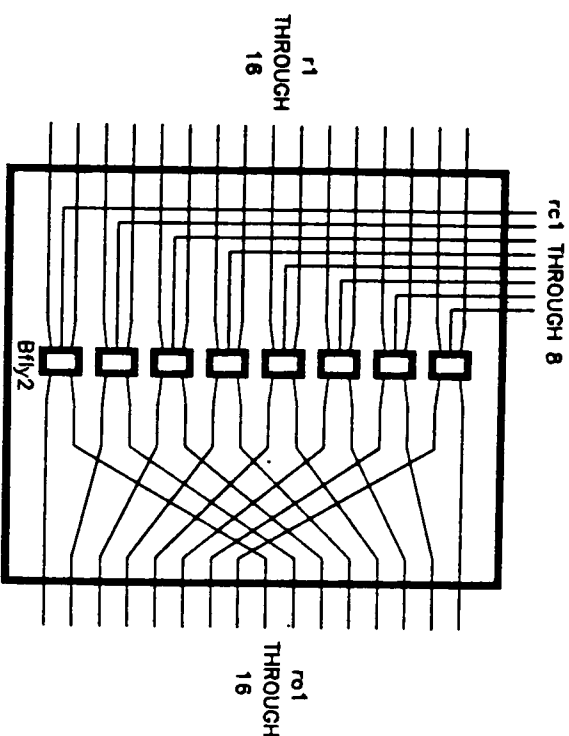


Fig. 8.20 Main butterfly processor column SUBSYSTEM 'MainColumn', showing perfect unshuffle on output. All data lines are complex.

Now we have the three chip types which we require to build the full array FFT. Our only remaining task is to implement the network topology in the SUBSYSTEMS.

The MainColumn and Heart subsystems

The Bfly2 CHIPS connect together in the constant-geometry FFT architecture depicted in Figure 8.20 to form the SUBSYSTEM MainColumn.

```
SUBSYSTEM MainColumn(c1) r1 THROUGH 16, i1 THROUGH 16,
rc1 THROUGH 8, ic1 THROUGH 8 -> ro1 THROUGH 16,
io1 THROUGH 16
SIGNAL yr, xr, yi, xi, wr, wi, yro, xro, yio, xio
Bfly2(c1) yr, xr, yi, xi, wr, wi -> yro, xro, yio, xio - TIMES 8 WITH
yr = r1, r3, r5, r7, r9, r11, r13, r15
xr = r2, r4, r6, r8, r10, r12, r14, r16
yi = i1, i3, i5, i7, i9, i11, i13, i15
xi = i2, i4, i6, i8, i10, i12, i14, i16
wr = rc1 THROUGH 8
wi = ic1 THROUGH 8
yro = ro1 THROUGH 8
xro = ro9 THROUGH 16
yio = io1 THROUGH 8
xio = io9 THROUGH 16
```

END

This code uses the shorthand TIMES statement, with assignment-replacement to wire up the Bfly2 chips. The network topology is contained in the ordering of the replacement lists following the TIMES statement. The list of internal nodes declared in the SIGNAL statement are in fact dummy nodes – they are replaced by the corresponding element of the list as the chips are instantiated. Every node here is distinct (no nodes connect to more than one chip). TIMES syntax allows many more connection types, as we saw in Chapter 3.

As we selected a constant-geometry algorithm, the SUBSYSTEM Heart may be constructed by simply cascading three MainColumn SUBSYSTEMS. Here the appropriate twiddle factors must be routed to the butterflies. In practice we would store these in ROM, but for the purpose of simulation we create a chip which contains CONSTGEN primitives for each numerically individual coefficient.

```
SUBSYSTEM Heart(c1) r1 THROUGH 16, i1 THROUGH 16,
wre0 THROUGH 7 - ro1 THROUGH 16, io1 THROUGH 16
SIGNAL rc1 THROUGH 8, ic1 THROUGH 8
MainColumn(c1) r1 THROUGH 16, i1 THROUGH 16, rc1 THROUGH 8,
ic1 THROUGH 8 - ro1 THROUGH 16, io1 THROUGH 16 -
TIMES 3 WITH ro1 THROUGH 16, io1 THROUGH 16 -
r1 THROUGH 16, i1 THROUGH 16
rc1 wre0, wre0, wre0
rc2 wre0, wre0, wre1
rc3 wre0, wre2, wre2
rc4 wre0, wre2, wre3
rc5 wre4, wre4, wre4
rc6 wre4, wre4, wre5
rc7 wre4, wre6, wre6
rc8 wre4, wre6, wre7
ic1 wre4, wre4, wre4
ic2 wre4, wre4, wre3
ic3 wre4, wre2, wre2
ic4 wre4, wre2, wre1
ic5 wre0, wre0, wre0
ic6 wre0, wre0, wre1
ic7 wre0, wre2, wre2
ic8 wre0, wre2, wre3
```

END

Note the extensive use of assignment-replacement in Heart (Figure 8.21), to route the appropriate twiddle factors to the Bfly2 CHIPS. We take full advantage of commonality between real and imaginary twiddle components.

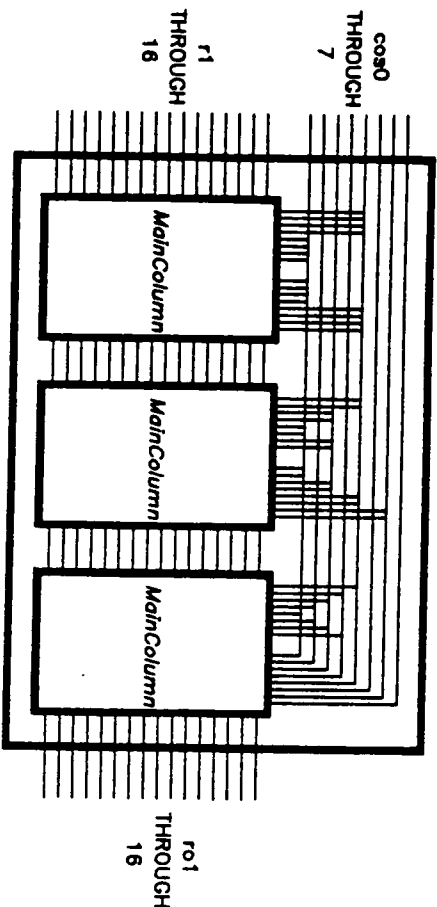


Fig. 8.21 SUBSYSTEM 'Heart', showing routing of twiddle factors. All data lines are complex.

Cascading is realised by the syntax immediately after the TIMES statement, which specifies how outputs of one column connect to the next. Any node not mentioned in either the cascading syntax or the assignments is connected globally – for instance the c1 inputs to these SUBSYSTEMs. Thus all output nodenames must appear in the syntax which follows the TIMES statement, if node conflicts are to be avoided.

The InColumn SUBSYSTEM

The construction of the SUBSYSTEM InColumn (Figure 8.22) is a more complex problem, as it is here that the index bit-reversal is implemented. We further complicate matters by using the symmetry of the windowing function to reduce the number of coefficient lines into the SUBSYSTEM. Again we use

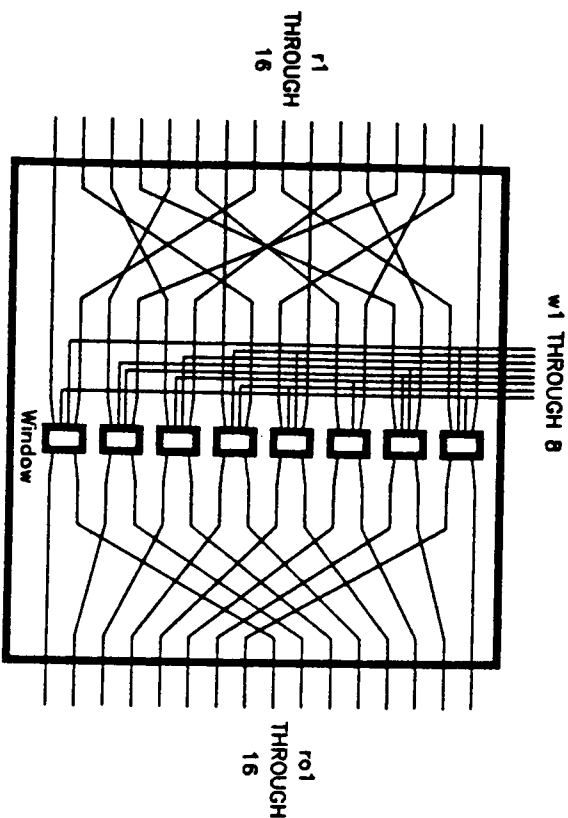


Fig. 8.22 Windowing SUBSYSTEM 'InColumn', showing bit-reversing shuffle on input and perfect unshuffle on output. All data lines are complex.

assignment-replacement for coefficient routing.

Here we are using a truly symmetric window, and not a DFT-symmetric window (Harris, 1978). The axis of symmetry of the latter falls on sample $N/2$, while that of the former falls half a sample period earlier. The DFT symmetric window requires one extra coefficient to be generated, increasing pin count, and the improvement in performance (over true symmetry) predicted by Harris is unlikely to be significant, given the noisy nature of our transform.

```
SUBSYSTEM InColumn (zc1) r1 THROUGH 16, i1 THROUGH 16,
w1 THROUGH 8 -> r01 THROUGH 16, i01 THROUGH 16
```

```
SIGNAL yr, yi, xr, xi, wy, wx, yro, xro, yio, xio
Window (zc1) yr, xr, yi, xi, wy, wx -> yro, xro, yio,
xio TIMES 8 WITH
  yr = r1, r5, r3, r7, r2, r6, r4, r8
  xr = r9, r13, r11, r15, r10, r14, r12, r16
  yi = i1, i5, i3, i7, i2, i6, i4, i8
  xi = i9, i13, i11, i15, i10, i14, i12, i16
  wy = w8, w4, w6, w2, w7, w3, w5, w1
  wx = w1, w5, w3, w7, w2, w6, w4, w8
  yro = r01 THROUGH 8
  xro = r09 THROUGH 16
  yio = i01 THROUGH 8
  xio = i09 THROUGH 16
```

END

The OutColumn SUBSYSTEM

Recall the OPERATOR ComplexToMagnitude from Chapter 3, which was duplicated and integrated on CHIP Magnitude. The construction of SUBSYSTEM Outcolumn (Figure 8.23) then consists of 8 distinct instantiations of Magnitude. The wiring topology used is somewhat arbitrary, as any scheme which maintains consistency of in, out and mag node numbering will do.

```
SUBSYSTEM OutColumn (c1) r01 THROUGH 16,
im1 THROUGH 16 -> mag1 THROUGH 16

SIGNAL rea, reb, ima, imb, maga, magb

Magnitude (c1) rea, reb, ima, imb -> maga, magb TIMES 8 WITH
  rea = r01 THROUGH 8
  reb = r09 THROUGH 16
  ima = im1 THROUGH 8
  imb = im9 THROUGH 16
  maga = mag1 THROUGH 8
  magb = mag9 THROUGH 16
```

END

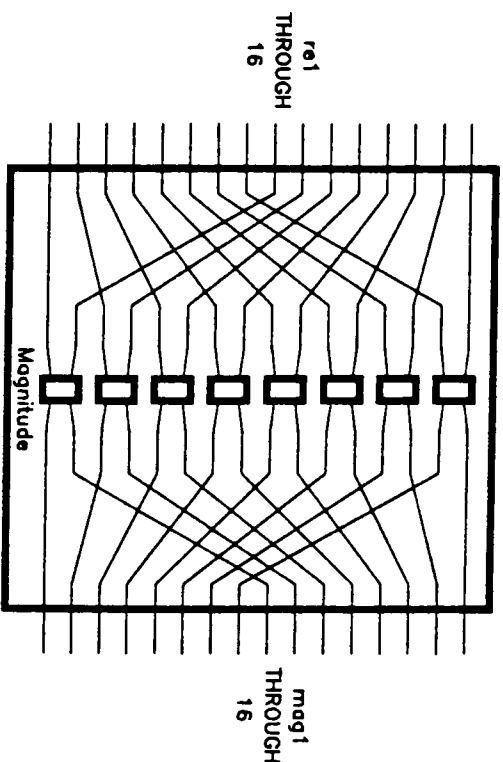


Fig. 8.23 Magnitude extraction SUBSYSTEM 'OutColumn', showing necessary symmetry. Input lines are complex, output real.

The FFT SYSTEM

We have now arrived at the top (SYSTEM) level in the hierarchy. The power of hierarchy lies in the ability of the designer to hide detail in lower levels. This we have already done within SUBSYSTEMs (topological details), and CHIPS (computational details). Thus putting together the SYSTEM FFT (Figure 8.24) is merely a matter of cascading the SUBSYSTEMs InColumn, Heart and OutColumn in a regular manner, and requires little effort.

```
SYSTEM FFT rin, iin -> out1 THROUGH 16
SIGNAL rin1 THROUGH 16, iin1 THROUGH 16,
r1 THROUGH 16, i1 THROUGH 16, ro1 THROUGH 16,
io1 THROUGH 16, cos0 THROUGH 7, w1 THROUGH 8
CONTROL c1
Corner rin, iin -> rin1 THROUGH 16, iin1 THROUGH 16
Twiddles (c1) -> cos0 THROUGH 7
HamCoeffs (c1) -> w1 THROUGH 8
Cgenerate (-> c1)
InColumn (c1) rin1 THROUGH 16, iin1 THROUGH 16,
w1 THROUGH 8 -> r1 THROUGH 16, i1 THROUGH 16
Heart (c1) r1 THROUGH 16, i1 THROUGH 16,
cos0 THROUGH 7 -> ro1 THROUGH 16, io1 THROUGH 16
OutColumn (c1) ro1 THROUGH 16, io1 THROUGH 16 -> -
out1 THROUGH 16
```

END

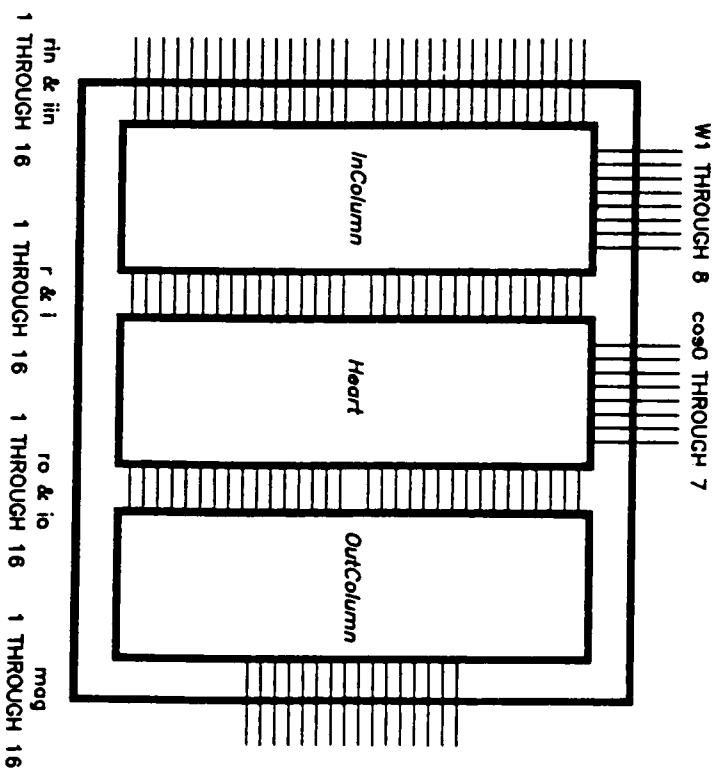


Fig. 8.24 SYSTEM FFT.

8.5.4 Simulation

The Fourier transformer is a machine which allows decomposition of a signal into an arbitrary number of spectral components or 'bins' - in this case 16. Windowing is performed prior to FFT processing to minimise the effects of 'off-bin' components on non-related bins. We employ Hamming and Blackman windows (Harris, 1978) to effect windowing.

To demonstrate the use of windows, we illustrate a simple two-tone detection problem. One tone is at full-scale amplitude, and the other is at 1% of this (i.e. 40 dB down). Figure 8.25 shows the log-magnitude output of the processor when both tones are at discrete observation frequencies, on bins 7 and 12. We observe the 40 dB difference between these bins, and note that transform noise due to poor arithmetic accuracy almost masks the minor tone. The worst offender is the partner tone to the dominant - this is bin 15 (7 + the span, which is 8 here). This bin accumulates energy (and errors) in identical manner to the dominant until the last butterfly column, where it is uniquely identified. The 6 dB loss of dynamic range necessary for correct multiplier operation can be seen in the dominant tone. The energy in bins 2, 6, 10 and 14 corresponds to one bit.

Next we move the dominant tone on to a position between bins, the new normalised frequency being 6.5. The spread of energy over the other bins is apparent, and the minor tone is no longer detectable (Figure 8.26).

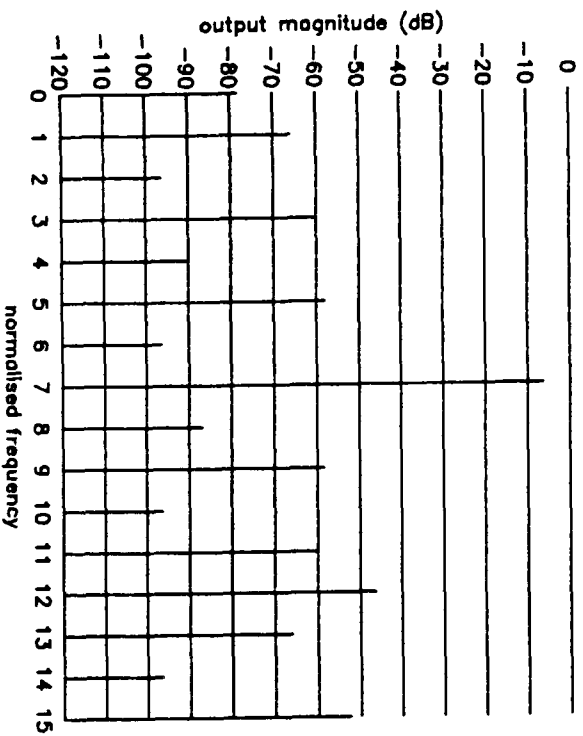


Fig. 8.25 FFT simulation, showing response to tones on bins 7 and 12, with 40 dB difference, no window used.

Introduction of the Hamming window, shown in Figure 8.27, reduces the leakage to a great enough extent to reveal the minor tone once more – a good 14 dB above its neighbours. The Blackman window has a wider main lobe, and lower sidelobe energy than the Hamming window. Its detection characteristics are displayed in Figure 8.28, showing higher energy in immediate neighbours of the two tones, but less in other bins.

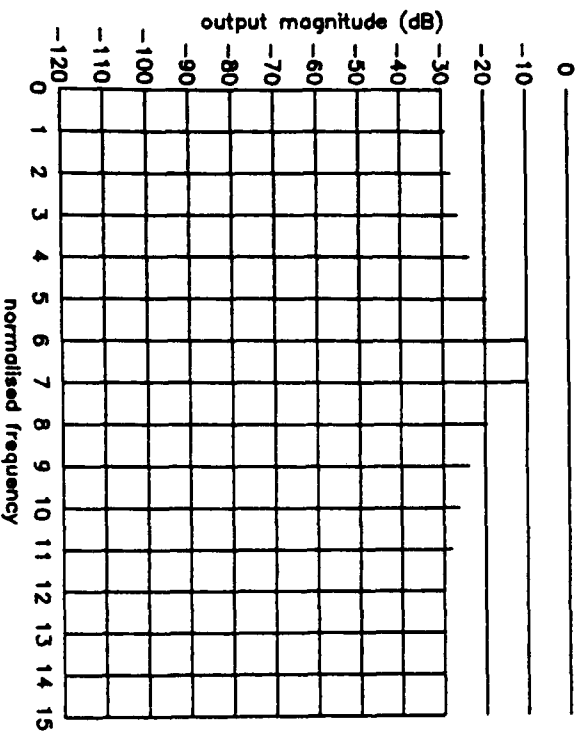


Fig. 8.26 Result of moving dominant tone to a mid-bin position (bin 6.5), no window used.

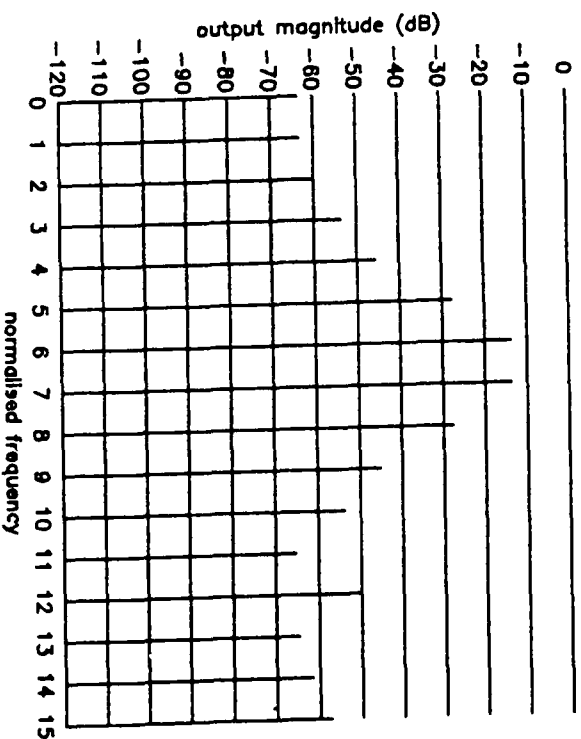


Fig. 8.27 Introduction of Hamming window reveals minor tone once more.

8.5.5 Some communications issues

The full array FFT machine is capable of one transform every word-time. If the input word rate to the entire machine – corner-turning memory included – is equal to the processor word-rate, then such a machine performs a 'sliding transform' on the input data. Each incoming word participates in N transforms before expiring, and the maximum amount of processing is carried out on the

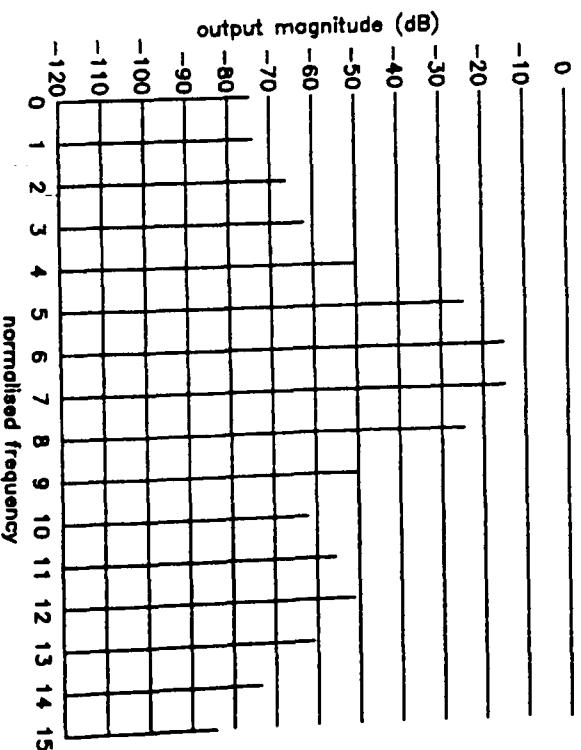


Fig. 8.28 Response of Blackman window in the same experiment.

input sequence. A sliding transform of the type just described, where phase information is neglected, can also be realised recursively by a column DFT machine with only order(N) processors (Ting, 1980). Although this machine accumulates an output phase error, this is of no importance if only output magnitude is of interest.

A more common transform is the 'block transform', whereby a block of entirely new samples is presented to the machine every word-time. Each input word then participates in just one transform. However, for such a scheme to be effective, the input word-rate should be N times the processing word-rate. Even for a modest machine such as the one just described, the input word-rate is 7.53 MHz. Higher transform lengths lead to unattainable input data rates, despite the ability of the processor to deal with the data once input. In/out bandwidth limits performance in this case.

A common compromise between the computational burden of the sliding transform and the communications burden of the block transform is the 'overlapped transform' (Rabiner and Gold, 1975). This is accomplished through the multiplexing technique described in Chapter 5, allowing matching of in/out and processing bandwidths. Alternatively, block transform machines may be realised with in/out data rates less severe than that of the parallel machine.

The FFT being implemented on a rectangular array of processors, two multiplexing schemes suggest themselves – a row scheme and a column scheme. We investigate both, but only produce a chip set for the former machine.

8.6 The column FFT machine

We mentioned earlier that the constant geometry network would allow one column of processors to be repeated spatially, i.e. cascaded, to implement the transform. If the repetition is made temporal rather than spatial, then only one physical column of processors is required. We are trading time against silicon area – a common technique in systems optimisation.

It is fortunate that in the case $N = 16$ (giving $M = 4$ for radix-2), the sub-block length ($= N/M, = 4$) is an integer. Thus we process 4 samples at a time, and carry out block transforms at a quarter of the rate of the full array machine, using approximately a quarter of the hardware. We say 'approximately' because the area-time tradeoff is more than just a matter of dividing down the hardware of the full array machine – some delay and switching elements must also be introduced for correct operation.

A further penalty, as mentioned earlier, is that we must implement the 'trivial' twiddles of the first column, as the same hardware realises all columns of the machine. Thus we cannot combine windowing with the first column as before. Not only is this wasteful of silicon, but it also causes arithmetic degradation due to the fact that unity cannot be represented. One way round this would be to negate all twiddle factors, and swap the complex adders and subtractors in the butterflies (one of the tricks used in the full array machine).

Hardware overview

Our initial concept of the column machine is a single column of butterfly processors, like MainColumn of the full array study, with a windowing front end and a magnitude-extracting back end. The front end can be a set of simple weighting blocks, like Xweight, and the back end can contain elements like ComplexToMagnitude. As the in/out data rate is one-quarter of that of the full array machine, only four instantiations of these OPERATORS are required. The corner-turning memory takes four samples from the input word-stream, and outputs them on four wire-pairs. Figure 8.29 shows a high level system plan.

Some design problems manifest themselves as we consider the operation of the butterfly column, and how it communicates with the front and back ends. This column must read in its 16-point windowed data block, then recirculate its outputs three times before reading the next block. Its inputs and outputs are 16-point, while the interfaces to the front and back ends are 4 x 4-point (i.e. four sequential blocks of four). Two actions are required from the designer.

Firstly we must 'time align' the four sub-blocks which are output sequentially from the weighting block. Similarly we must 'de-align' the column outputs, for correct presentation to the magnitude-extraction block. Secondly we must ensure that the column and the back end 'read' the correct data at the correct time.

The former strategy can be implemented using 'wedges' of delay – in the case of the front end interface with common inputs, and in the case of the back end with distinct inputs (Figure 8.30). The front end wedge is arranged to perform the required bit-reversing shuffle on the 4×4 -point signal block. The technique employed minimises the number of crossovers in signal paths, at no greater hardware cost than a hardwired shuffle.

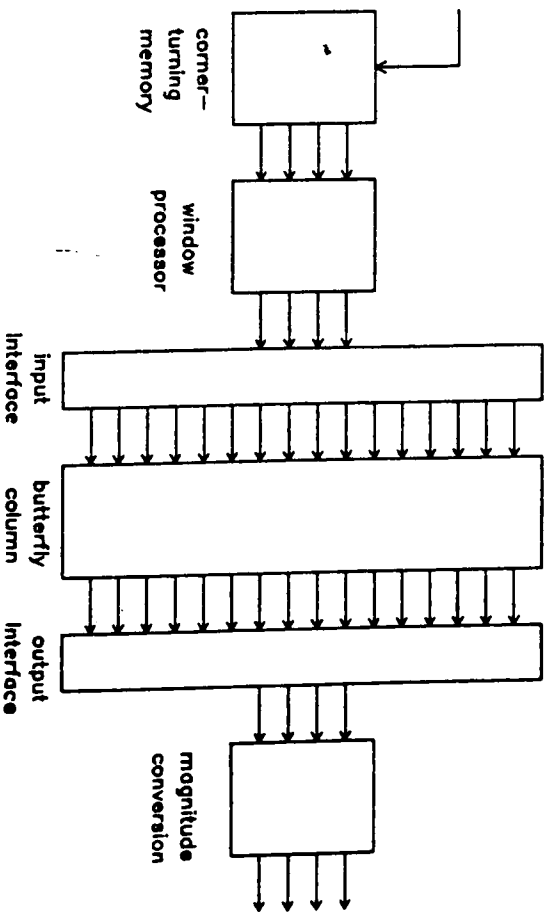


Fig. 8.29 Column FFT machine plan.

The latter strategy is accomplished by employing MULTIPLEX primitives – at the front end interface to read a valid input block once in four cycles, and at the back end to select one from four sub-blocks, ensuring both that selection order is correct, and that the valid machine output is read and not some intermediate inter-column result. The signals are passed by the MULTIPLEXers when valid (as shown by the node numbers in time sequences in Figure 8.30).

Another issue is the sharing of multiplier coefficients in the weighting block and in the FFT column engine. We must realise a memory loop of length 4 words for each processor, containing a MULTIPLEX switch to read in coefficients on start up. The coefficients will subsequently recirculate indefinitely, appearing in the correct time-slot at the multiplier. Thus the machine must run in two modes – start up and steady state.

Initial hardware decisions

We now have a much clearer idea of what comprises the column machine, and can commence its design. Much of this work has already been done in the full array study – we need only add the wedges, switches and memory loops described above. The front end wedge can be included at the outputs of each windowing multiplier, at the cost of increasing pin count on that chip. The back end wedge, having distinct inputs and outputs, should inhabit the same chip as the '1 from 4' data selector used by the magnitude extractors, thereby minimising the number of pin-expensive 16 channel transfers between chips.

The butterfly chips can be the same as used in the full array study, with the addition of MULTIPLEX primitives for input data selection, and memory

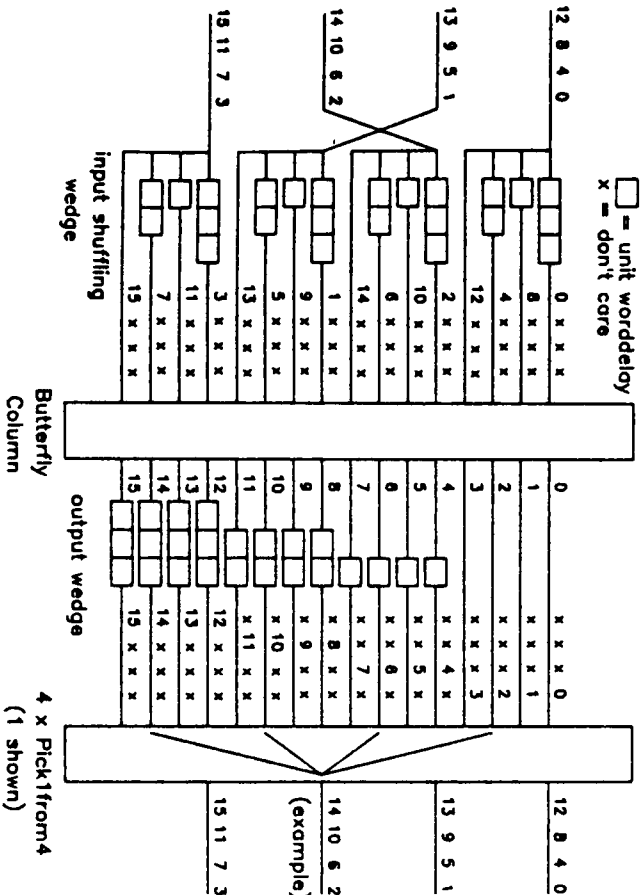


Fig. 8.30 Action of input and output interfacing wedges.

loops for storing real and imaginary coefficients. Note that the inclusion of MULTIPLEX in the signal path increases the sw1 by one bit, as the sw1 has a minimum value decided by the latency of the BFLY2 chip (for synchronisation reasons). Thus sw1 = 18 bits here. Finally the Magnitude chip, being small, can support the system CONTROLGENERATOR (CG).

Control and multiplexing issues

In the full array study, we ensured a simple control system by making all columns of the machine word synchronous. All we then needed to do was supply a c1 signal to each chip in the system. Here we are involved with a more complicated control network, and some care must be taken in the design process.

We are using a multiplexing regime, and accordingly must employ higher levels of control than c1. MULTIPLEX primitives are situated at the column inputs, and at the back end inputs. In the former task, they select one from two signals, recirculating outputs back to inputs for 3 out of 4 word cycles. This suggests a c2 CYCLE, or 'frame', of length 4. In the latter task they select one from four, in a cyclic fashion.

As we only have one-from-two MULTIPLEX primitives available to us, we could construct a one-from-four selector as a binary tree of three MULTIPLEXers. This necessitates a c2 CYCLE of 2 words and a c3 CYCLE of 2 frames, yielding a 2-bit control code for selection control. Unfortunately this compromises our earlier scheme for the butterfly data recirculation, and, as it is the latency of this loop which will decide our minimum system wordlength, we must seek another solution for the one-from-four selector. Figure 8.31 shows a scheme which employs a c2 CYCLE of 4, and a c3 CYCLE of 2.

We must also provide the facility for catching multiplier coefficients described earlier. A MULTIPLEX switch can be programmed to allow the coefficients to be loaded into the loop on start up. This action is under the

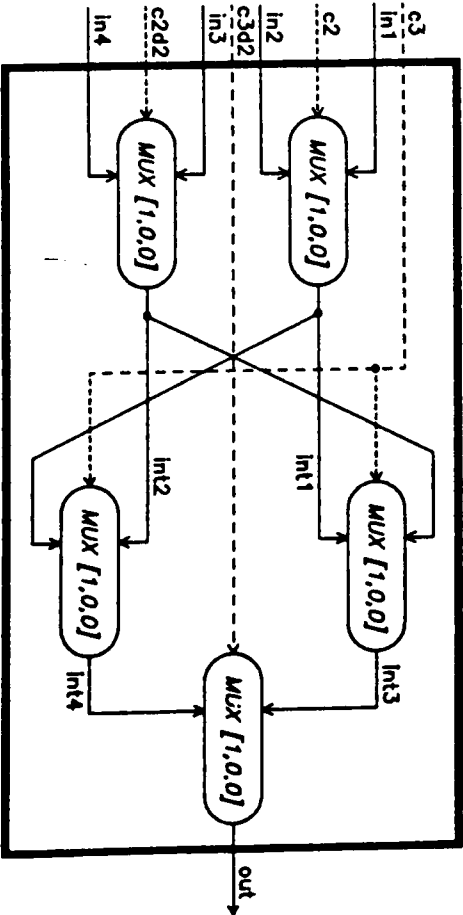


Fig. 8.31 Output selection OPERATOR 'Pick1from4'.

control of an event pulse, of length 4 words (the frame length), and thus we associate the EVENT with CYCLE c2.

A final design issue is the control timing. We maintain c1 synchronism at chip level throughout, at the cost of a 4-bit CBITDELAY on Magnitude CHiPs (which in turn allows some cyclic c1 optimisation on Magnitude). We have hidden c1 details in the hierarchy.

The next control level, c2, decides when the butterflies read the new input vector. As we are not concerned about how the input data stream is sectioned into blocks, we can treat the inputs to the Column SUBSYSTEM as a temporal 'reference point' (RP), and design the control network from there on. The controls input to OPERATOR 'Pick1from4' are word synchronous, and in fact are c2-frame synchronous as well (the Column output is valid at the same time as it reads its new input). Thus the control implementation of 'Pick1from4' reduces to the problem of wiring up its 4 input lines – see code and Figure 8.31.

OPERATOR Pick1from4 (c2, c2d2, c3, c3d2) in1 THROUGH 4 → out
latency is 3

```
SIGNAL int1 THROUGH 4
MULTIPLEX [1,0,0] (c2) in2, in1 → int1
MULTIPLEX [1,0,0] (c2d2) in4, in3 → int2
MULTIPLEX [1,0,0] (c3) int1, int2 → int3
MULTIPLEX [1,0,0] (c3) int2, int1 → int4
MULTIPLEX [1,0,0] (c3d2) int3, int4 → out
```

END

The event pulse for coefficient catching should be in synchronism with other control signals. Synchronism may be maintained by setting the latency of OPERATOR 'CofCatch' to sw1, via the MULTIPLEX latency parameter. (Signals passing through MULTIPLEX are delayed by at least 1 bit, thus 1 word is the cost of maintaining synchronism.) Now we must ensure that coefficients are presented to CofCatch in such a fashion as to coincide with the desired data samples at the multipliers.

OPERATOR CofCatch [sw1] (c3) in → out
latency is sw1

```
SIGNAL s1 THROUGH 3
MULTIPLEX [sw1,0,0] (c3) s3, in → out
BITDELAY [sw1] out → s1
BITDELAY [sw1] s1 → s2
BITDELAY [sw1] s2 → s3
```

END

Hardware partitioning

The description contains 4 SUBSYSTEMs, each of which contains multiple instances of a unique CHIP. Figures 8.32–8.35 show the floorplans of the four CHiPs employed.

The windowing CHIP Windo contains 4 multipliers, 4 wedges, and 4 CofCatch OPERATORS. The wedges, consisting of BITDELAYS of length 1, 2 and 3 words, are decomposed into 6 equal-length BITDELAYS to reduce chip height. Similarly CofCatch is partitioned into 4 word-sized blocks.

The butterfly CHIP Bfly is much the same as the parallel version, but includes 2 CofCatch OPERATORS, 4 MULTIPLEX primitives and output recirculation. The MULTIPLEXs are combined with Xmult to make the time-aligned block 'PickAndTwiddle' (like Twiddle2 of the previous study, with selection on all 4 inputs). Note the 1-bit predelay on all multiplexers, to compensate for pin delay. This is necessary because synchronous data recirculation is internal to the CHIP, and does not incur a pin delay. Delays required in the twiddle-free top leg are realised in the MULTIPLEX latency parameter.

OPERATOR PickAndTwiddle [coeff, lat] (c1, c2 → c1o) rin1, rin2, iin1, iin2,
rb1, rb2, ib1, ib2, wre, wim → rout1, rout2, iout1, iout2
latency is compdel + 1 (see expression below)

```
SIGNAL r, i
CONSTANT compdel = (3 * coeff)/2 + 2 + lat
MULTIPLEX [1 + compdel, 1, 0] (c2) rb1, rin1 → rout1
MULTIPLEX [1 + compdel, 1, 0] (c2) ib1, iin1 → iout1
MULTIPLEX [1, 1, 0] (c2) rb2, rin2 → r
MULTIPLEX [1, 1, 0] (c2) ib2, iin2 → i
Xmult [coeff, lat] (c1 → c1o) r, i, wre, wim → rout2, iout2
```

END

We partition the SUBSYSTEM WedgePair in a different manner from the rest of the machine - we separate real and imaginary parts, as this is the only form of symmetry displayed by the SUBSYSTEM. SUBSYSTEM WedgePair consists of two OutWedge CHiPs, one each for real and imaginary signals. OutWedge in turn consists of 4 Pick1from4 OPERATORS, and a wedge made with 6 DelayBlock OPERATORS, DelayBlock being a 4 channel parameterised delay element. We call DelayBlock 6 times, not 3, in order to restrict BITDELAYS to a size commensurate with the rest of the chip, just as in Windo and CofCatch.

Finally the Magnitude CHiPs are implemented much as before, but with the inclusion of a CONTROLGENERATOR primitive, and a 4-bit CBITDELAY to maintain c1 synchronism at chip level (4 bits is the latency of the previous CHIP in the path, OutWedge). The CG must be isolated from the rest of the chip, as FIRST allows the use of only one CG per system. Internal control on Magnitude is brought in via its pins.

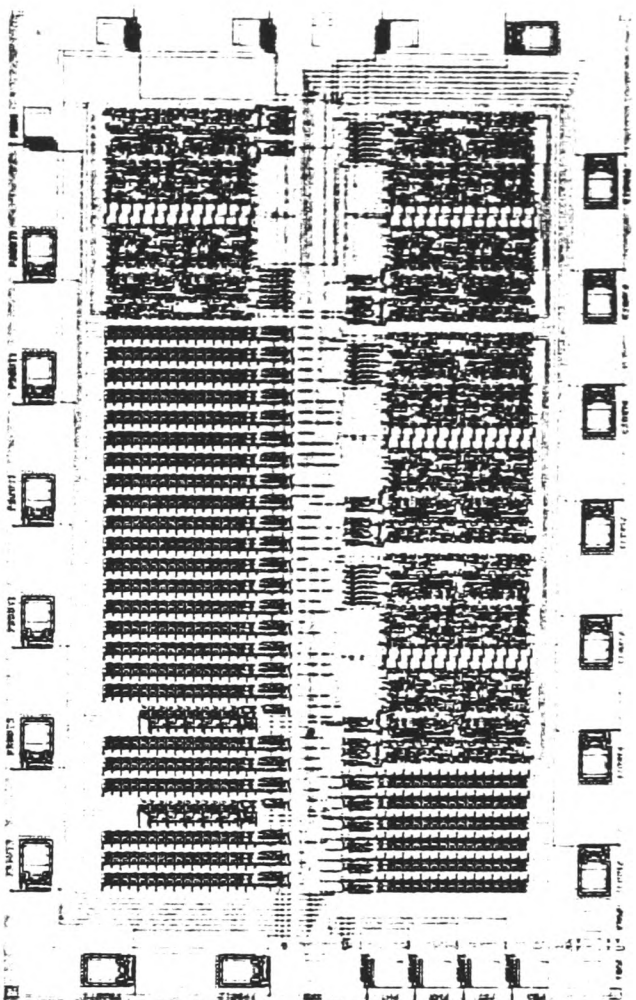


Fig. 8.32 Floorplan of input weighting CHIP 'Windo'.

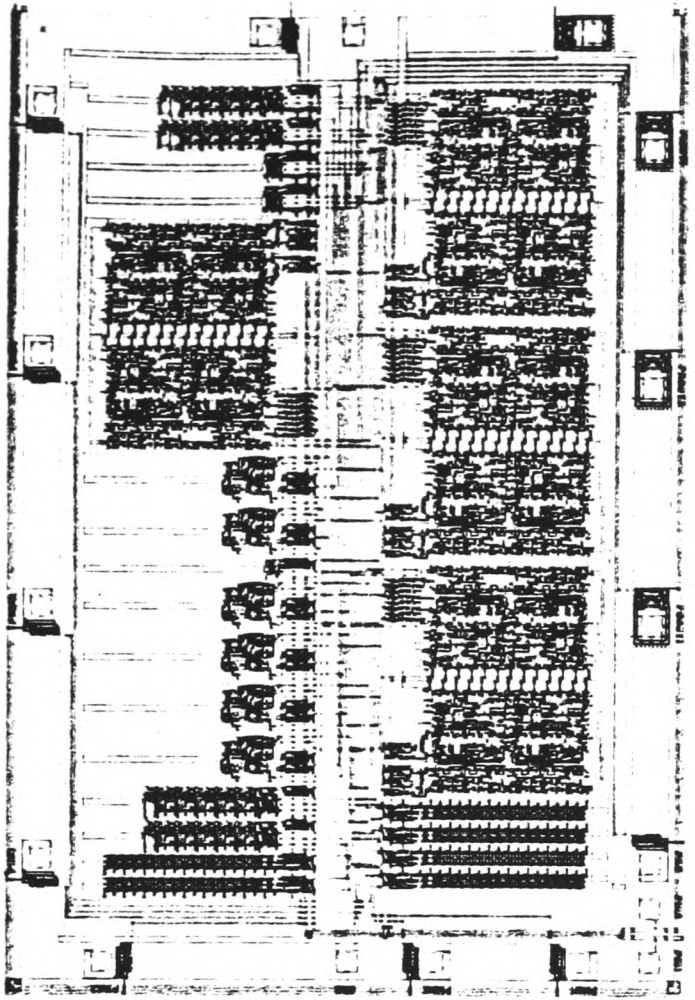


Fig. 8.33 Floorplan of butterfly processor CHIP 'Bfly'.

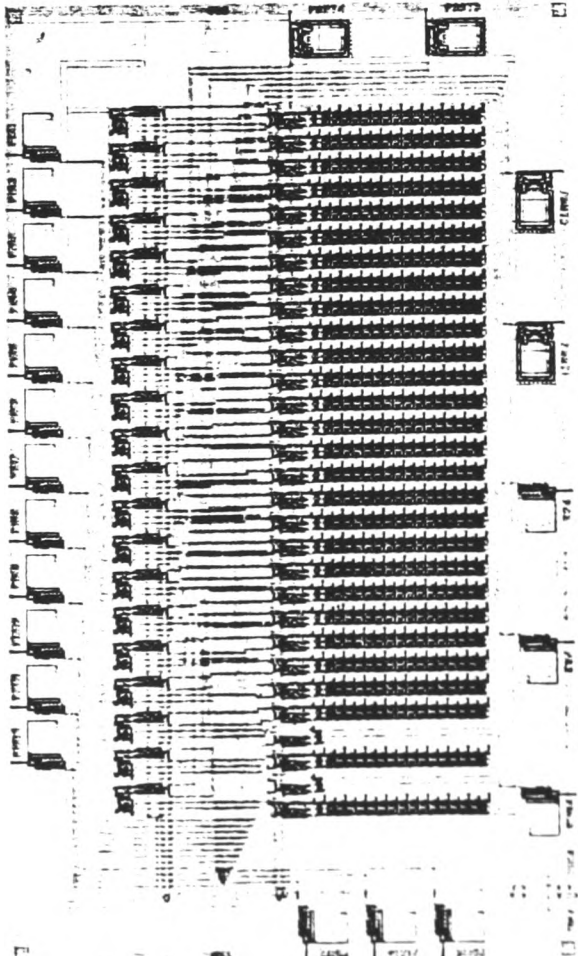


Fig. 8.34 Floorplan of output wedging and selecting CHIP 'OutWedge'.

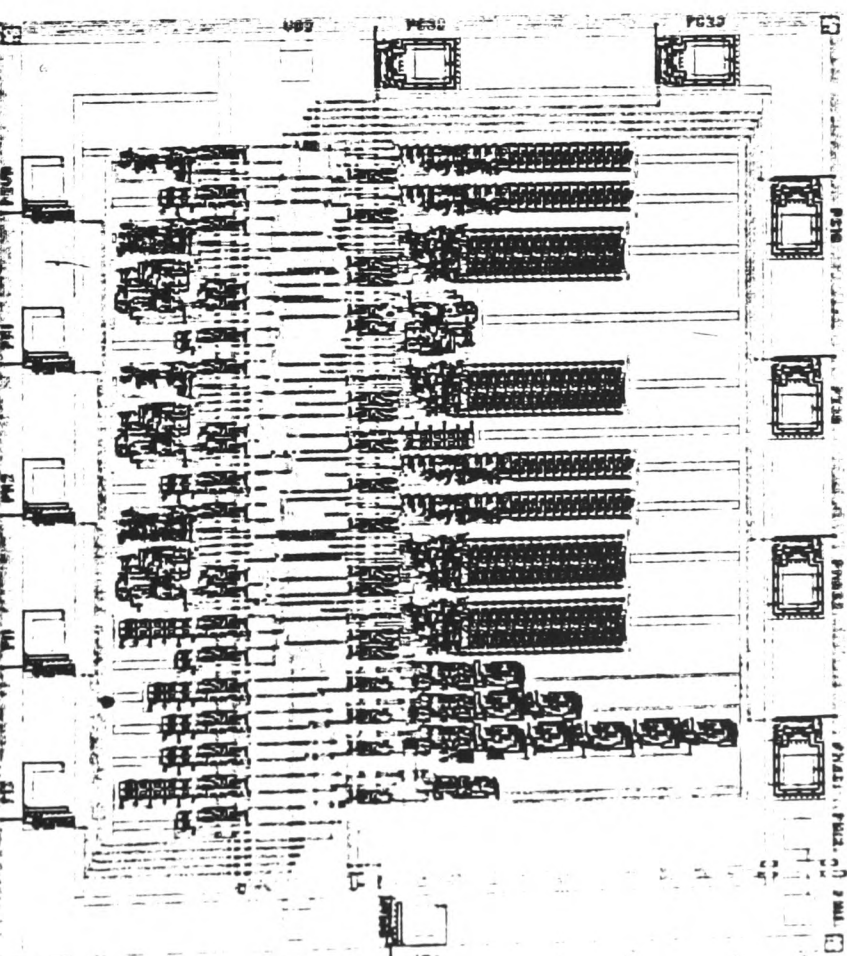


Fig. 8.35 Floorplan of magnitude extraction and control generation CHIP 'Magnitude'.

Figures 8.32 – 8.35 show the chips used – their sizes are as follows:

Chip	Width (λ)	Height (λ)	Aspect Ratio
Bfly	2461	1645	1.50
Windo	2657	1666	1.59
OutWedge	2237	1314	1.70
Magnitude	1691	1337	1.26

This study has shown that many design issues arise after floorplans have been produced, and some iteration is normally required to optimise any chip set. The silicon compiler lets the designer successively refine his work, allowing physical, as well as functional, considerations to come to bear on the design. We oversimplified the physical design issues in the full array study – it is unusual to compile and use an OPERATOR ‘toolkit’ without any regard to physical implications.

8.7 The pipeline FFT machine

We have seen (for radix-2) how the FFT can be implemented in a fully parallel manner, using a large number – $(N/R)\log_R N$ – of butterfly processors, where N is the transform length and R is the radix. It follows that the hardware cost is of order $(N/R)\log_R N$ (there are $\log_R N$ columns, each containing N/R processors). The parallel structure is capable of block transforms in unit word time. The area-time product is therefore also of order $(N/R)\log_R N$.

We have also seen how area could be traded against time to realise the column machine. When N is large, this scheme still leads to fast but hardware-intensive systems. A more common area-time tradeoff is to divide area, and multiply time, by N/R , i.e. to employ a multiplexing level of N/R . The same transform may then be performed in a word time of N/R , using only $\log_R N$ processors. This is in effect implementing just one row, and multiplexing down the columns. The resulting machine is a pipeline FFT (McClellan and Purdy, 1978), and the most advanced FFT machine known to the authors employs this architecture (with radix $R = 4$) to realise 4096-point transforms on 40 MHz data (Swartzlander and Hallnor, 1984).

8.7.1 Some radix issues

A radix- R DFT is capable of processing R points of complex data at a time, and a radix- R butterfly uses $(R - 1)$ twiddles. An N -point radix- R full array machine requires $(N/R)\log_R N$ butterflies, while an equivalent pipeline machine needs $\log_R N$ stages of processors (Rabiner and Gold, 1975). If we neglect the cost of adders, and also ‘trivial’ columns of twiddles (usually one column is twiddle-free), we can state that a radix- R machine will yield a factor of $R/2$ improvement in throughput over a radix-2 machine, whilst only costing $(R-1)/\log_2 R$ times more hardware.

It would seem from these figures that if the ratio of throughput to hardware cost were proposed as a figure of merit for FFT machines, then we should choose as high a radix as possible. This leads us to expose the fallacies of the approximations we made. At radices higher than 4, using standard arithmetic, the DFT machines used in the butterflies require internal multipliers, for instance 2 in the case of radix-8. It seems that radix-4 is in some sense optimal for long transforms on pipeline machines (McClellan and Purdy, 1978), and this is the radix we choose.

We shall demonstrate here a fairly modest machine; a 64-point, radix-4 transform. In this instance we omit the details of physical implementation, i.e. the partitioning of the system into chips.

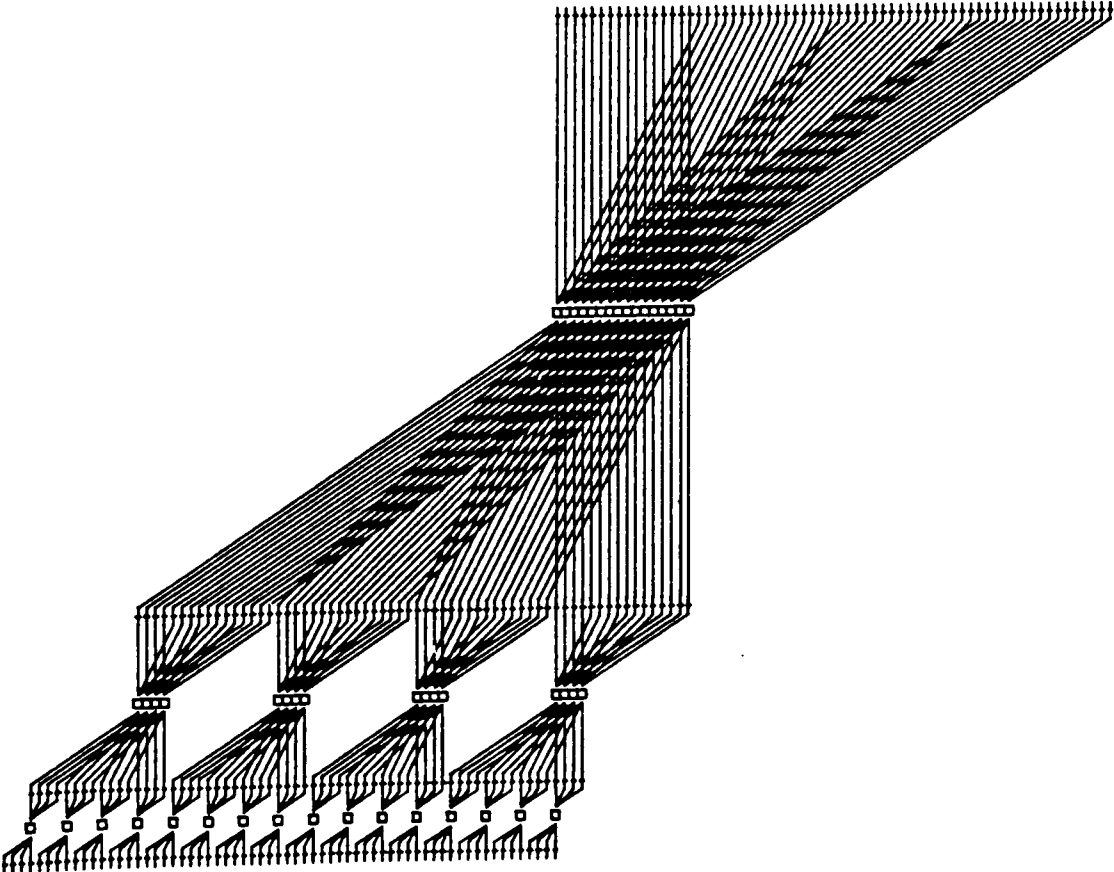


Fig. 8.36 64-point Radix-4 Cooley-Tukey FFT network, drawn in such a way as to indicate the position and size of delays in the equivalent pipeline machine.

8.7.2 Architecture

The area-time tradeoff mentioned above is a little more complicated in reality, as we neglected the cost of storage in the pipeline FFT. Storage is required as a result of the non-local routing in all FFT algorithms. We choose a network which has maximal locality of routing, and correspondingly requires minimal storage. This network implements the radix-4 Cooley–Tukey FFT.

Figure 8.36 shows the parallel version of the network, and Figure 8.37 the pipeline equivalent. Figure 8.36 has four columns of wiring posts interspersed with three columns of processors. Locality can be visualised as the vertical distance of connections from posts to processors in Figure 8.36. Note that the vertical distances from processors to posts have been distorted by a factor of R for diagrammatical convenience, resulting in the vertical gaps between subsequent processor groups. The vertical distances (which are proportional to the stage spans) decrease by a factor of R as we move from left to right through the parallel processor.

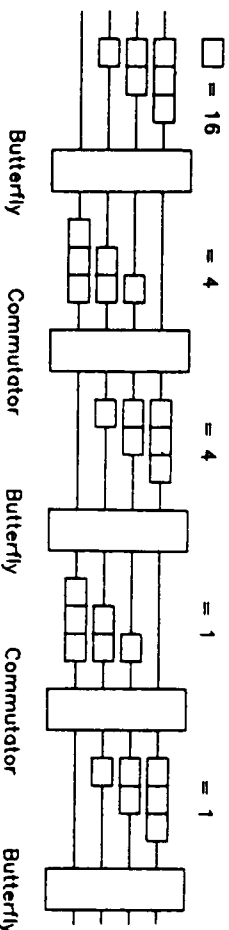


Fig. 8.37 Pipeline equivalent of radix-4 FFT machine, showing unit delays at each stage.

Pipe lengths

The decrease in span is accompanied by a factor of R increase in topological repetition. The quantity of span in the parallel machine has been traded for time delay in the pipeline machine. The time delays are implemented by wedges of delay which re-align samples spaced N/R words apart. The length of these wedge components corresponds to the spans of Figure 8.36, bearing in mind the distortion mentioned earlier. Accordingly, the pipe lengths decrease by a factor of R as we move through the pipeline processor (Figure 8.37).

The commutator

The pipeline FFT machine performs exactly the same transform as the parallel machine, but processes data in contiguous parallel sub-blocks of length R , where R is the transform radix (4 here). Although, as we saw in the previous example, the topology of FFT algorithms can be arranged to be constant across the array, it cannot be made constant in a downward direction, the direction in which we multiplex in the pipeline machine.[†]

[†] Note that the constant-geometry network is inappropriate for pipelining – locality is constantly poor.

The network topology, which was hardwired in the parallel and column machines, is implemented here using a dynamic routing device, or commutator (Rabiner and Gold, 1975), along with the antisymmetric wedges of memory on either side. Notice in Figure 8.36 that the first column topology repeats once, the middle column four times and the last column sixteen times. This property has an analogy in Figure 8.37 – the number of times the commutator runs through its switching pattern in a transform period.

The switching pattern itself relates to the algorithm topology, at the input to each processor group. Just as we replace the columns of processors in the parallel machine by a single processor in the pipeline machine, we replace the column of wiring posts with a wedge-commutator-wedge arrangement, which acts as a 'topological interface' between processors, performing the perfect shuffle which was hardwired in the previous machines. This arrangement will transpose an $R \times R$ matrix of signal sub-blocks. The length of these sub-blocks is the span at that stage in the transform, and forms the 'unit pipe length' for the particular transform stage. Figure 8.38 shows the action of the commutator in shuffling a 4×4 signal block.

Transform types and input buffering

The initial column of posts is replaced by the right-hand wedge only, as a commutator is unnecessary unless maximum block transfer rate is required. Simply tying the R input lines together allows execution of 'overlapped' transforms, useful in fast convolution (Rabiner and Gold, 1975), where the overlap ratio is $(R-1)/R$. We shall implement a machine of this type.

Full speed block transform capability requires fast input buffering. This usually takes the form of a 'swinging buffer' memory – a dual delay line of length N , with R equally spaced tap-out points. While one memory is filling at the input word-rate, the other is emptying on the R lines, at the word-rate supported by the bit-serial hardware it feeds. The input word-rate is thus R times the processing word-rate. Like the corner-turning memory of the parallel machine, we have not explicitly studied this memory.

Output data shuffling

The overlapped transform is frequently used in fast convolution, where the input signal undergoes an FFT, is term-by-term multiplied with a reference signal (the frequency domain equivalent of convolution), and then inverse FFT'd. This allows an order $(N/\log N)$ speed-up over conventional sum of lagged-products structures (Stockham, 1966). The inverse FFT (IFFT) is exactly equivalent to the FFT (save for a scaling factor and the use of conjugate twiddles (Rabiner and Gold, 1975)). Thus the topology is identical, and a double dose of digit reversal results in the correct ordering of the output. The reference sequence must be input in digit-reversed order.

We shall shuffle the outputs, however, to simplify their interpretation. To this end, we must implement a digit-reversing network. Digit-reversal can be embedded in the transform itself (Parker, 1980), but this leads to poor locality.

If the wedge-commutator-wedge arrangement which performs the perfect shuffle of order 2 after column 1 is replicated at the pipeline output, it will

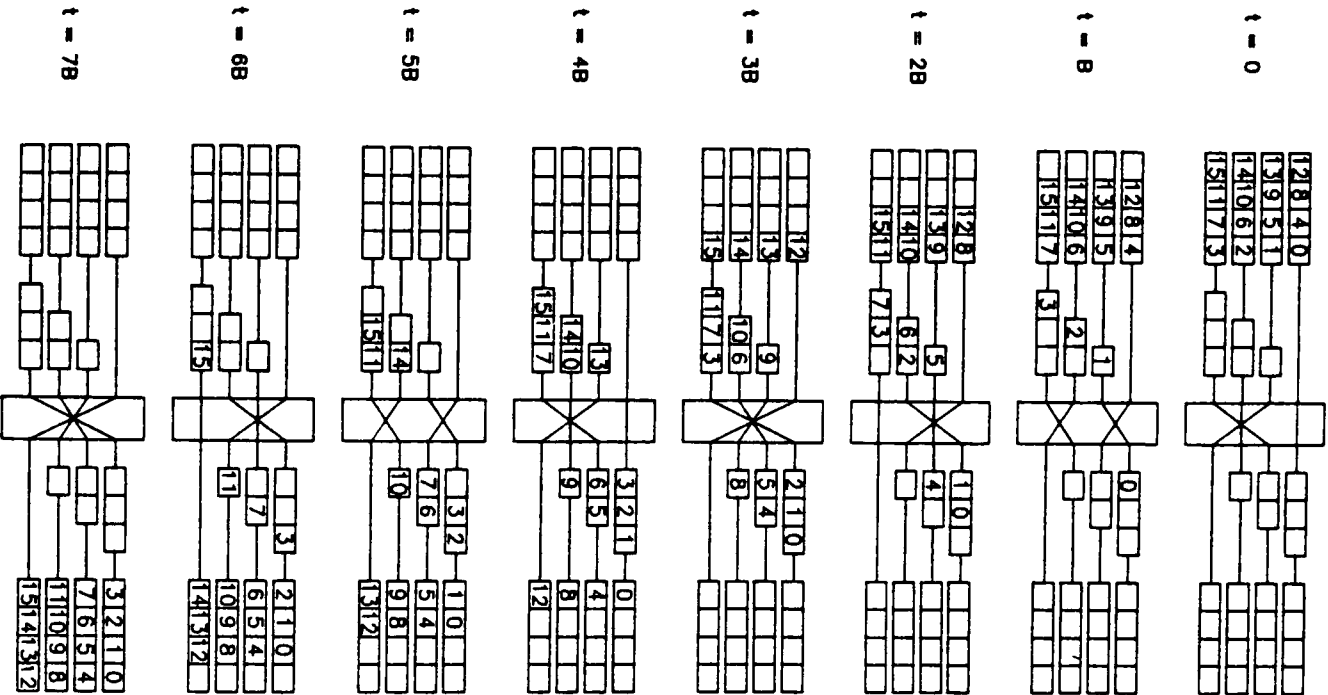


Fig. 8.36 Block shuffling action of the commutator. B is the unit delay in words, and switch pattern is shown before data transfer takes place.

perform a digit reversing shuffle. This is only possible because the relationship between processor output and effective network topology differs between stage 1 and stage 3 outputs.

A digit-reversing shuffle can be implemented as a cascade of decreasing-order perfect shuffles (or unshuffles). The order 1 shuffle is a null operation.

The order 2 perfect shuffle of the additional network, being a simple digit-swap, is identical to an order 2 unshuffle. The stage 3 topology is derived from that of stage 1 by a perfect unshuffle of order 3, as can be seen by inspection of the network in Figure 8.36. Thus we have a cascade of decreasing order unshuffles, which as stated earlier effects the digit-reversing shuffle.

Final design issues

We are already familiar with most of the elements which constitute the pipeline FFT – for instance the OPERATORS Xmult and Dft4. Although a dedicated primitive would be more elegant, OPERATOR ‘Commutator’ can be implemented as 2 instantiations of OPERATOR ‘FourPole’, which consists of 8 MULTIPLEX primitives controlled by a 2-bit code. The third control signal (cde1) is a delayed version of cnplus1.

OPERATOR FourPole[s] (cnplus1, cde1, cn) in1 THROUGH 4 -> -
out1 THROUGH 4

! latency is s

SIGNAL s1 THROUGH 4

MULTIPLEX [s/2,0,0] (cnplus1) in3, in1 -> s1
MULTIPLEX [s/2,0,0] (cde1) in4, in2 -> s2
MULTIPLEX [s/2,0,0] (cnplus1) in1, in3 -> s3
MULTIPLEX [s/2,0,0] (cde1) in2, in4 -> s4
MULTIPLEX [s-(s/2),0,0] (cn) s2, s1 -> out1
MULTIPLEX [s-(s/2),0,0] (cn) s1, s2 -> out2
MULTIPLEX [s-(s/2),0,0] (cn) s4, s3 -> out3
MULTIPLEX [s-(s/2),0,0] (cn) s3, s4 -> out4

END

The overall control problem reduces to one of generating a succession of binary counted word frames, and implementing the 2-bit control codes at each stage. The latency of processing elements in the pipeline should be taken into consideration when connecting up control.

It should be borne in mind that (neglecting processor latency) the first block of transform output appears in synchronism with the last block of transform input (see the horizontal lines in Figure 8.36, corresponding to zero pipe delay). Thus the control code for the final shuffle commutator should be presented one word early with respect to any reference point used for the rest of the machine.

8.8 The linear array DFT

So far we have described only FFT machines. Although these machines improve on the DFT in terms of arithmetic hardware usage (order(N log N) for the FFT, order(N²) for the DFT), hardware savings are not dramatic when N is low. Kung (1980) described a linear ‘systolic’ array of length N which

evaluates the DFT by Horner's method, using the input sequence $x(n)$ as polynomial coefficients. Equation (8.1) may be rewritten:

$$X(k) = ((\dots((x(N-1))W_N^k + x(N-2))W_N^k + \dots + x(1))W_N^k + x(0)) \quad 8.9$$

Although this method is prone to rounding errors from repeated multiplication of the partial results by W^k , this again is not so important when N is low. The advantage of the approach is the high degree of processor modularity, and the locality of communication afforded.

Allen (1984) modified Kung's array for continuous operation on real data, and implemented it in FIRST, producing a CHIP for each stage of the linear array. He also demonstrated the ease with which the array elements can be multiplexed for bandwidth matching, and produced a further floorplan using the Golub complex multiplier described earlier.

The processing module

The DFT values are computed sequentially by passing partial results, together with the appropriate value of W (these are input in ascending rotational order), down the array. At each point in the array, they meet up with the polynomial coefficients $x(n)$, which travel down the array at half the speed, and are latched in each processor for the transform duration. This ploy is used to reverse the input sequence while maintaining local communication. In cell n , the partial result $p(k)$ is multiplied by W^k , and added to the input sequence element $x(N-1-n)$ resident in the cell. At the end of the array, $p(k) = X(k)$, and the

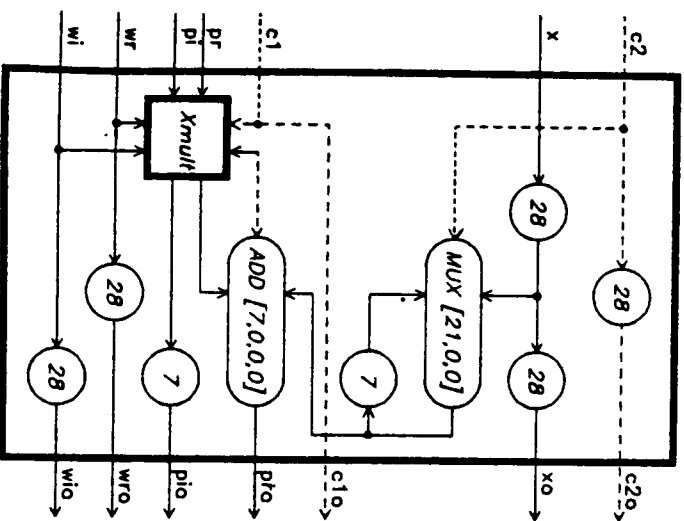


Fig. 8.39 Linear array DFT module.

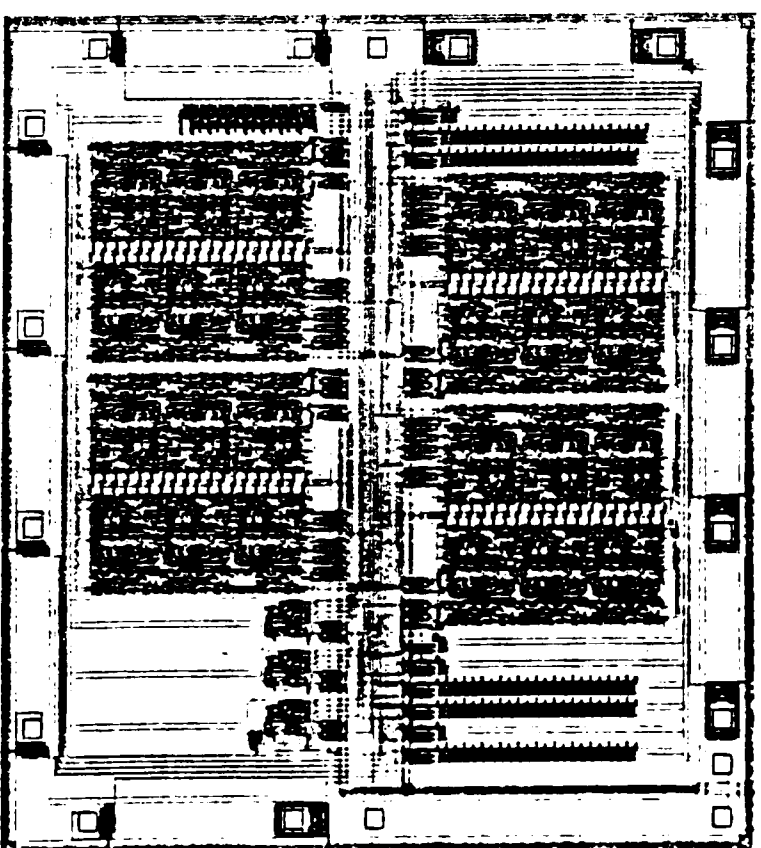


Fig. 8.40 Floorplan of linear array DFT CHIP.

transform latency is $2N$ words.

There is a strong parallel between the word-level operation of this array, and the bit-level operation of a serial multiplier (Lyon, 1976). Figure 8.39 shows the basic processor cell (with system wordlength of 28 bits), and Figure 8.40 the CHIP floorplan.

Control

Allen ensured that his design was c1-synchronous at CHIP level, save for the PAD delays incurred in communication between CHIPS. This allows c1 to be passed down the array with the other signals, maintaining local communication. Control is needed at c2 level, to mark 'start of block'. The c2 control signal, of cycle length N , is passed down the array at the same speed as $p(k)$ and W^k . At each processor, c2 'catches up' with a member of the input sequence $x(n)$, and the MULTIPLEX primitive controlled by c2 steers this element into the word-long loop where it recirculates for N word times while being added into $p(k)$.

8.9 Conclusions

We have implemented several types of Fourier transform machine, using only a small subset of the FIRST primitive set. The use of structure in design has

demonstrated that previous projects can provide much of the material for a new system. Complexity can be hidden at the appropriate level in the hierarchy employed. The availability of a parameterised, high-level language compiler plus simulator allows not only design verification, but architectural exploration as well. The time saved by automating the low-level tasks of IC design can be fruitfully spent working at the system level, where the designer uses his talents most productively.

The FFT is a prime case for the development of further application-specific primitives. In this case custom complex multipliers and adders, or even full butterfly stages, may be merited. Some more sophisticated data switching elements would be advantageous here.

References

- Allen, G. H., Denyer, P. B., and Renshaw, D. 'A Bit-Serial Linear Array DFT', *Proc. IEEE ICASSP'84*, pp 41A.1-4, San Diego, 1984.
- Bergland, G. D., 'A Guided Tour of the Fast Fourier Transform', *IEEE Spectrum*, vol. 6, no 7, pp 41-52, 1969.
- Cooley, J. W. and Tukey, J. W., 'An Algorithm for the Machine Calculation of Complex Fourier Series', *Math. Comp.*, vol. 19, pp 297-301, 1965.
- Despain, A. M., 'Fourier Transform Computers using CORDIC Iterations', *IEEE Trans. Computers*, vol. C-23, pp 993-1001, 1974.
- Filip, A. E., 'A Baker's Dozen Magnitude Approximation and Their Detection Statistics', *IEEE Trans. AES*, vol. AES-12, pp 87-89, 1976.
- Garverick, S. L. and Pierce, E. A., 'A Single Wafer 16-Point 16-MHz FFT Processor', *Proc. 1983 CICC*, Rochester, New York, 1983.
- Gentleman, W. M. and Sande, G., 'Fast Fourier Transforms - For Fun and Profit', *1966 Joint Fall Comput. Conf., AFIPS Proc.*, vol. 29, pp 563-578, Washington DC, 1966.
- Gold, B. and Bially, T., 'Parallelism in Fast Fourier Transform Hardware', *IEEE Trans. Audio Electroacoust.*, vol. AU-21, pp 5-16, 1973.
- Harris, F. J., 'On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform', *Proc. IEEE*, vol. 66, pp 51-83, 1978.
- la'ja', J., 'High-Speed Networks for Computing the Discrete Fourier Transform', *Proc. M.I.T. Conference on Advanced Research in VLSI*, pp 11-20, Cambridge, MA, 1984.
- Kung, H. T., 'Special-Purpose Devices for Signal and Image Processing: an Opportunity in Very Large Scale Integration (VLSI)', *Proc. SPIE*, vol. 241, Real-Time Signal Processing III, pp 76-84, San Diego, 1980.
- Linderman, R. W. *et al.*, 'Digital Signal Processing Capabilities of CUSP, a High Performance Bit-Serial VLSI Processor', *Proc. IEEE ICASSP'84*, pp 16.1.1-4, San Diego, 1984.
- Lyon, R. F., 'Two's Complement Pipeline Multipliers', *IEEE Trans. Communications*, vol. COM-24, pp 418-425, 1976.
- McClellan, J. H. and Purdy, R. J., 'Applications of Digital Signal Processing to Radar', pp 239-329 in A. V. Oppenheim (ed.), *Applications of Digital Signal Processing*, Prentice-Hall, 1978.
- McClellan, J. H. and Rader, C. M., (eds.) *Number Theory in Digital Signal Processing*, Prentice-Hall, 1979.

- Mead C. A. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.
- Murray, A. F., Denyer, P. B. and Donaldson, W., 'A CMOS Cell Library for Bit-Serial Signal Processing', *Proc. ESSCIRC'84*, pp 205-209, Edinburgh, 1984.
- Nuttall, A. H., 'A Two-Parameter Class of Bessel Weightings for Spectral Analysis or Array Processing - The Ideal Weighting-Window Pairs', *IEEE Trans. ASSP*, vol. ASSP-31, pp 1309-1312, 1983.
- Oppenheim, A. V. and Weinstein, C. J., 'Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform', *Proc. IEEE*, vol. 60, pp 957-976, 1972.
- Parker, D. S., 'Notes on Shuffle/Exchange-Type Switching Networks', *IEEE Trans. Computers*, vol. C-29, pp 213-222, 1980.
- Rabiner, L. R. and Gold, B., *Theory and Application of Digital Signal Processing*, Prentice-Hall, 1975.
- Rader, C. M., 'Discrete Fourier Transform When the Number of Data Samples is Prime', *Proc. IEEE*, vol. 56, pp 1107-1108, 1968.
- Reed, I. S., Yeh, C. -S. and Truong, T. K., 'A VLSI Architecture for Digital Filters Using Complex Number-Theoretic Transforms', *Proc. IEEE ICASSP'83*, pp 923-926, Boston, 1983.
- Stockham, T. G. Jr., 'High Speed Convolution and Correlation', *1966 Spring Joint Conf., AFIPS Conf. Proc.*, vol. 28, pp 229-233, 1966.
- Stone, H. S., 'Parallel Processing with the Perfect Shuffle', *IEEE Trans. Computers*, vol. C-20, pp 153-161, 1971.
- Swartzlander, E. E. Jr. and Hallnor, G., 'Fast Transform Processor Implementation', *Proc. IEEE ICASSP'84*, pp 25A.5.1-4, San Diego, 1984.
- Ting, C. -H., 'Fourier Transform Faster than Fast Fourier Transform (FFT)', *Proc. SPIE*, vol. 241, Real-Time Signal Processing III, pp 167-171, San Diego, 1980.
- Volder, J. E., 'The CORDIC Trigonometric Computing Technique', *IRE Trans. Electron. Comput.*, vol. EC-8, pp 330-334, 1959.
- Ward, J. S. *et al.*, 'Figures of Merit for VLSI Implementations of Digital Signal Processing Algorithms', *Proc. IEE Pt. F*, vol. 131, pp 64-70, 1984.
- Winograd, S., 'On Computing the Discrete Fourier Transform', *Math. Comp.*, vol. 32, pp 175-199, 1978.