NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Fabricating and Testing a VLSI Systolic Convolution Cell for Image Processing

Anthony Botzas

B. Eng., (McGill University), 1990

Department of Electrical Engineering

McGill University

Montréal

July, 1994

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master's Degree of Engineering

ISBN 0-315-99957-8

Fabricating and Testing a VLSI Systolic Convolution Cell

Anthony Botzas

B. Eng., (McGill University), 1990

Department of Electrical Engineering

McGill University

Montréal

July, 1994

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master's Degree of Engineering

Abstract

The two-dimensional discrete convolution operator is targeted for performance improvement in order to speed up image processing work loads. Since the large computation requirements for this operation are especially taxing to single processor computers, the approach is to consider parallel processing alternatives. Of the parallel processor classes considered, systolic arrays are singled out as the preferred parallel processing solution for the convolution problem.

Therefore, the design of a pipelined double precision floating point VLSI systolic cell for convolution is described. The arithmetic operations are distributed into three pipelined stages, enabling the cell to process each set of operands within 16 clock cycles. Once fabricated and tested, the systolic chip yielded an 80 MFLOPS performance which is a remarkable improvement over available general purpose computers.

Résumé

L'objectif du circuit pour la convolution bi-dimensionelle est d'accélérer le traitement des images digitales. Étant donné la complexité de l'opération, la convolution à deux dimensions place un lourd fardeau sur les processeurs séquentiels. Nous visons donc les méthodes de calculs parallèles pour la tâche. De tous les modèles de parallelisme considérés, nous démontrons les tableaux systoliques comme étant la solution de choix.

Pour cette raison, nous présentons la conception d'un circuit VLSI pour la convolution en pipeline utilisant des éléments systoliques. L'arithmétique requise, qui est à point flottant et de double précision, est distribuée sur trois stages de la pipeline. Ceci permet à chaque élément systolique de procéder à l'intérieur de seulement seize coups de l'horloge digitale. Une fois fabriqué et vérifié, le circuit à présenté une performance de 80 MFLOPs. Ceci démontre une accélération considérable par rapport aux ordinateurs conventionnels.

Acknowledgements

The author would like to thank Professor A. S. Malowany for his supervision in the research and the writing process of this thesis.

Thanks are also due to the McGill Research Centre for Intelligent Machines (McRCIM), the VLSI laboratory of McGill University and the Canadian Microelectronics Corporation (CMC) for providing the tools with which this research was conducted.

The author would like to thank the "veterans" (J. Drolet, J.F. Panisset, J.F. Côté and F. Larochelle) of the convolution project for laying the foundation on which this thesis stands.

The author is greatful to Prof. N. C. Rumin and Prof. G. Roberts, for their VLSI tutelage and to Eric Masson for a partnership that not only fostered a love of VLSI design but engendered a career.

The author is also immensely thankful to his family, friends, and co-workers for their constant encouragement and support in completing his thesis while working in industry.

Table of Contents

Chapter 1 Introduction 1

1.1 The Convolution Problem 3

1.1.1 Convolution in Signal-Processing 3

1.1.2 Convolution in Image-Processing 5

1.1.3 Applying the Convolution Operator 7

1.2 Computational Considerations 10

1.2.1 Uniprocessor Performance 11

1.2.2 Transforming the Domain of the Convolution Problem 15

1.2.3 Frequency Domain Versus Spatial Domain Processing 17

1.3 Summary 20

Chapter 2 Parallel Processing Solutions 21

2.1 Inherent Parallelism 22

2.2 Basics of Parallel Algorithms 24

2.3 General-Purpose Parallel Processing Options 25

2.3.1 Architecture Taxonomy 26

2.3.2 MIMD Parallel Processors 27

2.3.3 MIMD-based Parallel Processing Paradigms 29

2.3.4 Synchronous Parallel Processing Architectures 30

2.4 Systolic Array Architectures 35

2.4.1 Basic Architecture 35

2.4.2 Key Architectural Considerations 37

2.4.3 Balanced Systems 38

2.4.4 Convolution Revisited 40
2.5 Summary 41

Chapter 3 A Systolic Solution 44
3.1 System Architecture View 44
3.1.1 The Sensor Computing Environment 45
3.1.2 Interfacing Requirements 46
3.1.3 Data Stream Manipulation 46
3.2 Systolic Convolution Array 48
3.3 Systolic Cell 49
3.3.1 X Input Register 50
3.3.2 Multiplication Stage 1 51
3.3.3 Addition Stage 2 53
3.3.4 Normalization Stage 3 53
3.3.5 Partial Sum Transmission Unit 54
3.4 Summary 54

Chapter 4 Fabricating and Testing the Systolic Convolution Cell 55
4.1 Fabrication Process 55
4.1.1 CMOS3 DLM Processing Steps 55
4.1.2 CMOS3 DLM Design Rules 57
4.1.3 Packaging and Bonding 58
4.1.4 CADENCE VLSI Design Software 60
4.1.5 Design Files and File Hierarchy 60
4.1.6 Edge Database Format 61
4.1.7 Schematic Representations 62
4.1.8 Custom Layout 63

4.1.9 Layout Submission 66
4.2 Testing 67
4.2.1 Functional Testing 67
4.2.2 Manufacturing Testing 68
4.2.3 Testing Process 70
4.2.4 I/O Specification 75
4.2.5 Manufacturing-Test Principles 86
4.2.6 Manufacturing-Test Strategies for the Systolic Cell 88
4.2.7 Design for Testability 91
4.3 Summary 93

Chapter 5 Conclusion **94**

References **96**

Appendix A Convolution Benchmark **102**

Appendix B Parallel Convolution Benchmark **105**

Appendix C Overview of the IEEE Floating-Point Standard **108**

Appendix D VHDL Simulations **110**

Appendix E VLSI Design Life Cycle Activity **119**

Appendix F Design Block Hierarchy **121**

Appendix G Automated Testing Environment **145**

List of Figures

1.1 Example of a two-dimensional convolution. 6

1.2 Laplacian, $\nabla^2 W'$, kernel used in edge-detection. 9

1.3 Example of edge-detection with a Laplacian filter. 10

1.4 Frequency domain approach to convolution. 15

1.5 Frequency domain approach versus spatial domain approach. 18

1.6 Maximum kernel size for which convolution in the spatial domain used less flops than convolution done in the frequency domain. 19

2.1 Parallel computer architectures according to Duncan's taxonomy. 27

2.2 MasPar MP-1 interconnection network. 34

2.3 Two-dimensional systolic array connection topologies. 36

2.4 Comparison between (a) the traditional SISD computation model and (b) the systolic computation model. 39

2.5 Design SYST1: systolic convolution array. 42

3.1 Architecture of the systolic convolution processing system. 45

3.2 Systolic convolution array. 49

3.3 Systolic cell architecture. 50

3.4 Timing diagram for the systolic cell. 51

4.1 Top and bottom view of package. 59

4.2 Top-level-block schematic representation illustrating the floor plan of the systolic cell. 62

4.3 Sample cell layout of a flip-flop. 64

4.4 Layout representation. 65

4.5 VHDL simulation of the SYST2 design. 69

4.6 The test environment. 71

4.7 A.C. testing load circuit. 74

4.8 Oscilloscope trace of 1MHz CLK signal used in A.C. tests. 79

4.9 Switching waveforms at initialization. 81

4.10 Coefficient switching waveforms in coefficient-load mode. 82

4.11 Input and output switching waveforms. 85

C.1 Double precision floating-point representation. 108

E.1 Levels of abstraction in VLSI design. 119

E.2 VLSI design life cycle activity. 120

F.1 *fullcircuit2*: Top-level-block schematic representation of systolic cell. 127

F.2 *fullcircuit2/X_reg_28x4* : 28 × 4 shift register circuit for the X input data. . . 128

F.3 *fullcircuit2/X_reg_28x4/X_line* : Example of an iterative specification of an array of flip-flops. 129

F.4 *fullcircuit2/X_reg_28x4/X_line/ff_1* : Example of a typical delay flip-flop with "hold" capability. 130

F.5 *fullcircuit2/X_reg_28x4/X_line/ff_1/fff* : Example of a typical one-phase, master-slave, flip-flop implementation. 131

F.6 *fullcircuit2/X_reg_28x4/X_line/ff_1/inv_pass* : Example of an inverting pass gate implementation. 132

F.7 *fullcircuit2/Yin_exp* : Exponent portion of Y input circuit. 133

F.8 *fullcircuit2/Yin_man* : Mantissa portion of Y input circuit. 134

F.9 *fullcircuit2/Yout_exp* : Exponent portion of Y output circuit. 135

F.10 *fullcircuit2/control* : Global control circuitry. 136

F.11 *fullcircuit2/stage1_exp* : Exponent portion of first stage multiplication circuit. 137

F.12 *fullcircuit2/stage1_man* : Mantissa portion of first stage multiplication circuit.138

F.13 *fullcircuit2/stage2_exp* : Exponent portion of second stage addition circuit. . 139

F.14 *fullcircuit2/stage2_man* : Mantissa portion of second stage addition circuit. . 140

F.15 *fullcircuit2/stage3_exp* : Exponent portion of third stage normalization circuit. 141

F.16 *fullcircuit2/stage3_man* : Mantissa portion of third stage normalization circuit. 142

F.17 *fullcircuit2* : Top-level-block layout representation of systolic cell. 143

F.18 *fullcircuit2* : Top-level-block "exploded" layout representation of systolic cell. 144

List of Tables

1.1 Results of the CONV benchmark on the spectrum of available uniprocessors currently used to perform image-processing tasks. 13

2.1 Cost of various parallel processing options. 25

3.1 Features of the convolution system. 47

4.1 Pin description. 76

4.1 Pin description (continued). 77

4.2 D.C. characteristics. 78

4.3 Modes of operation in the systolic cell. 78

4.4 Initialization switching characteristics. 80

4.5 Coefficient switching characteristics. 83

4.6 Input and output switching characteristics. 86

Chapter 1 Introduction

Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used [Amdahl, 1967]. That is to say that perhaps the most important and pervasive principle of computer design is to make the common case fast: In making a design tradeoff, favour the frequent case over the infrequent case. This principle also applies when determining how to spend resources since the impact on making some occurrence faster is higher if the occurrence is frequent. Thus, improving the frequent event, rather than the rare event, will obviously help performance and increase "speedup" [Hennessy and Patterson, 1990].

Having impressed on the reader this keystone notion of performance improvement and taking into account the author's affinity toward image-processing and VLSI design, this thesis will aim to investigate and demonstrate a cost-effective means by which one can significantly improve the performance of image-processing tasks.

As practiced at present, image-processing relies to a great extent on two major, effectively distinct and self-contained domains of activity, computational algorithms and processor architectures -especially those architectures facilitated by VLSI technology [Offen, 1985]. Given the goal above, the scope of this thesis will lie in the latter domain and will focus on the design and test of a high-performance image-processing system.

The approach herein is to first isolate the most frequent operations employed in "typical" image-processing programs, and as per Amdahl's Law, to markably reduce the associated execution times of these operations so as to have the greatest impact on the total execution time of a given program. Of course, there is no such

thing as a typical image-processing program owing to the vast diversity of image-processing applications. And aside from the Abingdon Cross benchmark survey [Uhr, 1986] [Preston, 1989], little has been published about the workloads of image-processing systems.

The Abingdon Cross, which is presented [Lindskog, 1988] as a benchmark method for comparing the performance of image-processing architectures, has proved valuable to this thesis. Briefly put, the task is to find the medial axis of a cross in a noisy background where the signal to noise ratio is 0 dB. But unlike benchmarks for general-purpose computers [Lubeck *et al.*, 1985] [McMahon, 1989] [Berry *et al.*, 1988] which require the participant to compile and execute specific lines of Fortran code, the Abingdon Cross benchmark presents the participant with an image-processing problem without specifying the algorithm to be employed in its solution. Without a predefined algorithm and code, one cannot gain any specific knowledge with regard to instruction mix or instruction frequency, yet one can still draw some general but valuable information with regard to workload in image-processing systems. The common trait in almost all the algorithms used by participants of the Abingdon Cross benchmark survey is the utilization of spatial-domain linear operators for noise reduction and edge-detection.

Hence, if one desires to build an image-processing system suited to fast execution of the Abingdon Cross benchmark, one would likely begin by implementing in speedy hardware those frequently used operators such as the aforementioned noise filters and edge-detectors. Indeed, a general-purpose image-processing system should also be designed in this vein since a major preoccupation in image-processing is the filtering-out of noise and the enhancement of (edge) detail in order to emphasize certain specific properties [Levine, 1985] as is apparent in applications involving computer vision [Hall, 1979] [Ballard and Brown, 1982], robot vision [Briot, 1986], remote sensing [Mulders, 1987], acoustical imaging [Shimizu *et al.*, 1988], and tomography [Axel *et al.*, 1983]. Moreover, one can appreciate that these operators are also of prime concern to biological visual systems

[Hartline, 1949] [Hartline and Ratliff, 1954] [Hartline and Ratliff, 1957].

Most existing code for noise reduction and edge-detection relies on one linear operator known as the convolution operator. Thus the underlying premise of this thesis is that targeting the convolution operator for high-performance hardware implementation will have a high impact on the execution time of image-processing programs. And in the next section, the convolution operator is introduced and will remain as the "problem" on which this research effort will concentrate.

1.1 The Convolution Problem

The first priority in the quest to reduce image-processing execution time must be the speeding up of convolution computations. But a discussion on computational considerations cannot proceed until the operator itself is thoroughly defined. To start, the definition of convolution is first examined from a one-dimensional signal processing perspective. The context is then shifted to two-dimensional image-processing, and subsequently some examples are comprised so as to reinforce the definitions presented.

1.1.1 Convolution in Signal-Processing

Practitioners of discrete-time signal processing will attest that a linear time-invariant system is completely characterized by its impulse response $h[i]$, in the sense that, given $h[i]$ it is possible to compute the output sequence $y[i]$ due to any input $x[i]$ using

$$y[i] = \sum_{n=0}^{N-1} h[n]\, x[i-n] \tag{1.1}$$

Here,

$h[i]$ represents the N-length system's response to the unit sample sequence $\delta[i]$ also known as the discrete impulse function [Proakis and Manolakis, 1988]. As well, all signals are assumed to be causal and of finite length. Equation (1.1) is commonly called the "convolution sum". If $y[i]$ is a sequence whose values are related to the values of two sequences $h[i]$ and $x[i]$ as in Eq. (1.1), $y[i]$ is said to be the convolution of $x[i]$ with $h[i]$ and is represented by the notation:

$$y[i] = h[i] * x[i]. \tag{1.2}$$

It is apparent in Eq. (1.1) that in forming the input sequence $x[i-n]$, one must fold the sequence $x[n]$ about its origin to produce $x[-n]$ and displace it by i. Evaluating $y[i]$ then requires multiplying each of the overlapping samples of the $h[n]$ and $x[i-n]$ sequences and subsequently summing these products. Moreover, the commutativity of this operation implies that one can also choose to fold the impulse response $h[i]$ instead of $x[i]$.

Equation (1.1) is readily extensible to two dimensions. Clearly, the two-dimensional convolution of an input sequence $x[i,j]$ with an $N \times M$ impulse response $h[i,j]$ is given by:

$$y[i,j] = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} h[n,m]\, x[i-n, j-m] \tag{1.3}$$

Thus the fundamental expressions for discrete convolution have been presented, and it remains to be seen how these expressions are manipulated from an image-processing perspective.

1.1.2 Convolution in Image-Processing

In digital image-processing, the two-dimensional signals in Eq. (1.3) are said to represent "images". In a strict theoretical sense, an image is defined as a two-dimensional, almost invariably Cartesian, array of data resulting from sampling the projected instantiation of a local variable, the scene brightness function, obtained via a sensing device. The function values are either brightness values or vectors of brightness values sensed in different spectral bands, e.g. colour images. In the black-and-white case these values are usually called grey levels. The array values are typically real, non-negative, bounded, and implicitly zero outside the field of view bounded by the array dimensions. In addition, these digitized array elements are referred to as picture elements or "pixels".

In this thesis, however, the tendency is to view images as matrix entities. And hence the above Cartesian constraint on images is relaxed to include only those two-dimensional sequences whose pixels are addressed by positive integer indices. One such loose but preferred representation of Eq. (1.3) is the matrix element expression which follows:

$$Y[i,j] = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} W[n+1, m+1]\, I[i+n-N/2, j+m-M/2] \qquad (1.4)$$

In Eq. (1.4), W is an $N \times M$ matrix which is usually referred to as a "convolution kernel" and acts like an impulse response which is convolved with an input image I to produce the output image Y. Each pixel $Y[i,j]$ is simply the weighted sum of the corresponding pixel value $I[i,j]$ and the values stored in neighbouring pixels. The weights and number of neighbouring pixels that are included in the weighted sum are determined by the size and contents of the kernel W. Fig. 1.1 illustrates an example of how a 3×3 convolution is computed by sliding a kernel W over an image I, multiplying each kernel coefficient with the underlying pixel value, summing the products, and storing this sum in an image buffer Y.

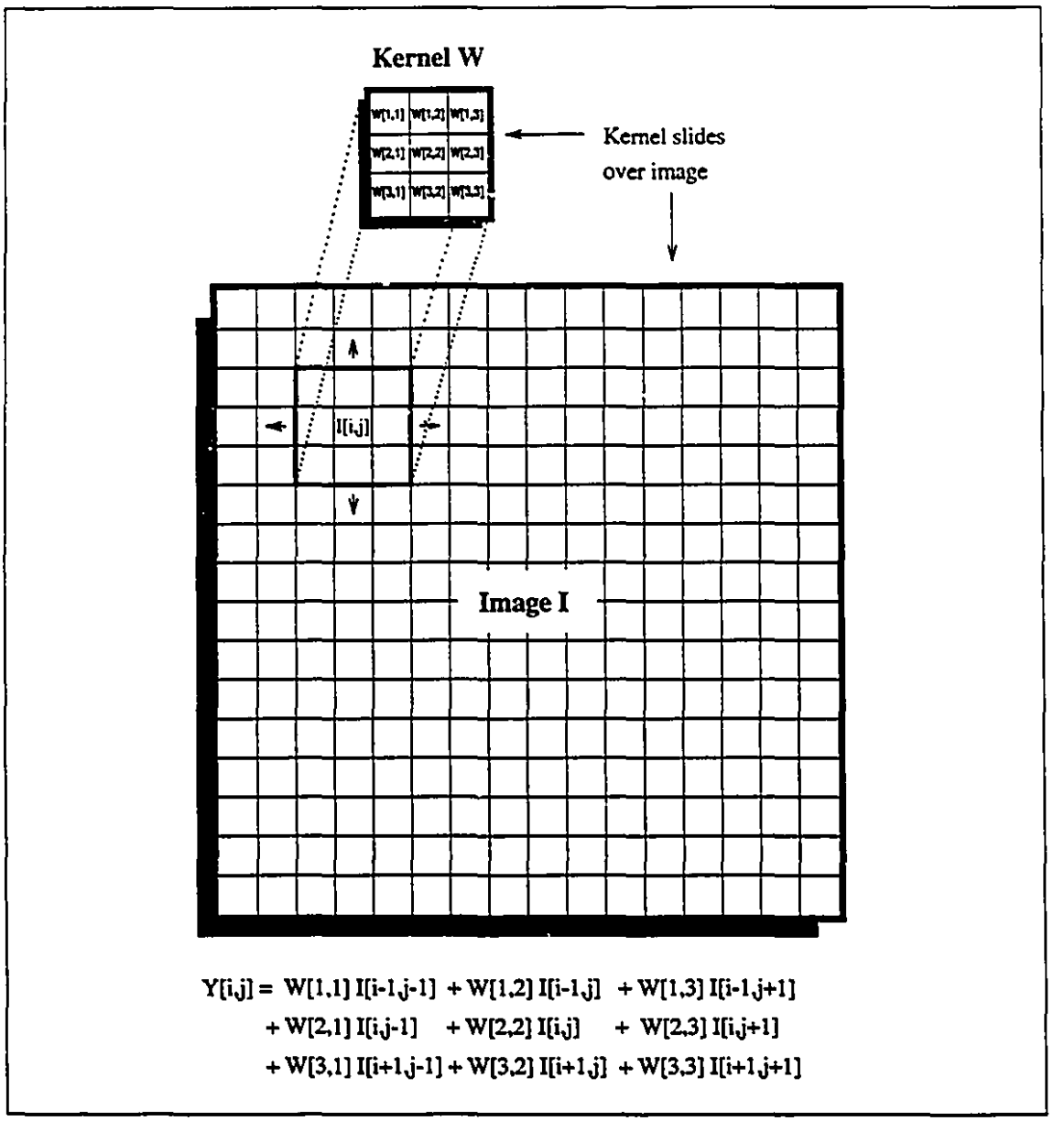

Figure 1.1: Example of a two-dimensional convolution of an image I with a 3×3 kernel W resulting in an image $Y = W' * I$.

Note that the row and column indices of I in Eq. (1.4) do not contain a $-n$ and a $-m$ respectively which implies that there is no folding of the image in each dimension. And neither is there a folding of W. Without a folding of either I or W, it seems that the expression in Eq. (1.4) violates the mathematical integrity of Eq. (1.3). However, it should be stated that the kernel W is often radially symmetrical and for all intents and purposes can be considered "pre-folded". For the cases where the kernel is asymmetrical, it becomes necessary to qualify Eq. (1.4) by adding that the W is considered to be the folded version of an original kernel W'. In other words, a convention is adopted where a given kernel W' should be rotated by 180 degrees to yield W prior to going through the convolution operator as defined in Eq. (1.4) and Fig. 1.1. The reason for separating the folding process from the definition of convolution has to do with facilitating hardware implementation, which will become more evident in Chapter 3. With this definition now in place, the classification and application of the convolution operation is discussed next.

1.1.3 Applying the Convolution Operator

Image-processing operations can be classified as either "point", "local" or "geometric" operations. Point operations involve transforming single pixels in a way that does not depend on any neighbouring pixels. Local operations involve transformations on pixels so that the new value of each pixel depends also on the values of pixels in some neighbourhood. Geometric operations involve pixel values at some other point, defined by a geometric transformation, or in a neighbourhood of that point.

Convolution, then, is defined as a local linear operator, and as such, it is very frequently used in the first stage of image analysis [Levine, 1985]. As was noted before, most initial filtering, edge-detection, correlation and compression relies to a large extent on convolution. At this point it would be instructive to reinforce the definition above with some practical examples of convolution kernels and their ef-

fect on images.

Consider first the case of a simple averaging filter. If all the weights were the same in a 3×3 kernel, then a given pixel in a convolved image Y would be the equally-weighted sum of the corresponding pixel in I and its eight immediate neighbours. The overall convolution effect would be simply to average or "smooth" abrupt spatial changes in pixel intensity.

An example of an edge-detection kernel proves a little more involved. Yet such an example is worth presenting, for it demonstrates that if the kernel is carefully chosen, a single convolution "pass" can do a substantial portion of image-processing work. The edge-detection problem is to outline discontinuities or abrupt spatial changes in pixel intensity [Schumann, 1904]. Marr and Hildreth [Marr and Hildreth, 1980] confront the issue by first proposing a Gaussian operator as an optimal smoothing filter. This implies convolving an image with a kernel W' whose weights are determined by the following two-dimensional Gaussian function:

$$W'(i,j) = \frac{1}{2\pi\sigma^2} \exp(-\frac{(i^2+j^2)}{2\sigma^2}) \tag{1.5}$$

The intensity changes in an image I will manifest themselves in the outputs $W' * I$ as peaks in the first derivative $D(W' * I)$, or as zero-crossings in the second derivative $D^2(W' * I)$ in the appropriate direction. In other words, the original edge-detection problem may be replaced by an equivalent one in which the zero-crossings of $D^2(W' * I)$ or, what is equivalent, $D^2W' * I$ are sought. Thus the kernel used for the edge-detection becomes:

$$W(i,j) = D^2W'(i,j) \tag{1.6}$$

and assuming linear local intensity variations near a zero-crossing it can be

shown [Marr and Hildreth, 1980] that a Laplacian operator may be employed in place of Eq. (1.6) such that the kernel W is acquired by $\nabla^2 W'$, an orientation independent second-order differential operator:

$$W(i,j) = \frac{1}{2}\,[2 - \frac{(i^2+j^2)}{2\sigma^2}]\,\exp[-\frac{(i^2+j^2)}{2\sigma^2}] \tag{1.7}$$

Equation (1.7) is plotted in Fig. 1.2 with $\sigma = 1$ and a domain translation to facilitate the mapping to a 9×9 kernel matrix W.

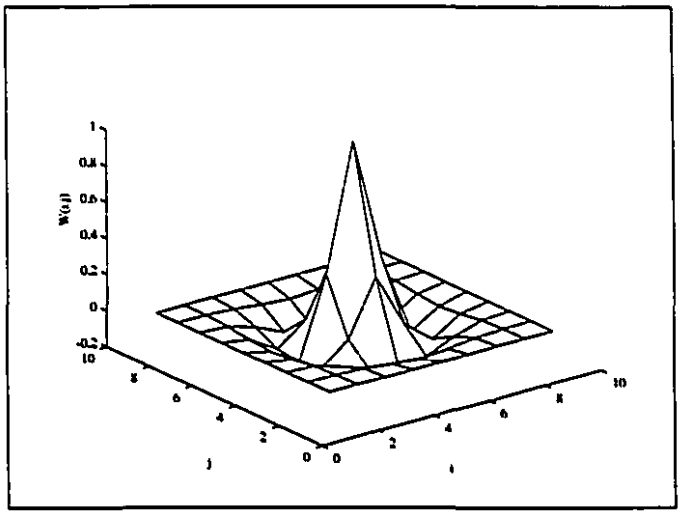

Figure 1.2: Laplacian, $\nabla^2 W'$, with a (low) $\sigma = 1$. The domain has been translated such that $1 \leq i \leq 9$ and $1 \leq j \leq 9$ resulting in a function that maps accordingly into a 9×9 kernel W.

Hence the inherently pre-folded, edge-detecting kernel as plotted in Fig. 1.2 is coded into a convolution program which also takes as its input a real image in the PGM (portable graymap) file format. The results are revealed in Fig. 1.3 and depict the effect of a Laplacian filter and the subsequent detection of zero-crossings where zero is represented by some intermediate gray-level in the output. In addition, using a larger kernel in the convolution (with a correspondingly larger σ) yields an edge-map with less detail. The chosen kernel size of 9×9 seems sufficiently large for extracting the right amount of detail from the given real image, though perhaps not so for other images. Mainly, it is hoped that the above discussion has adequately defined convolution and has alluded to the pivotal role that this mathematical op-

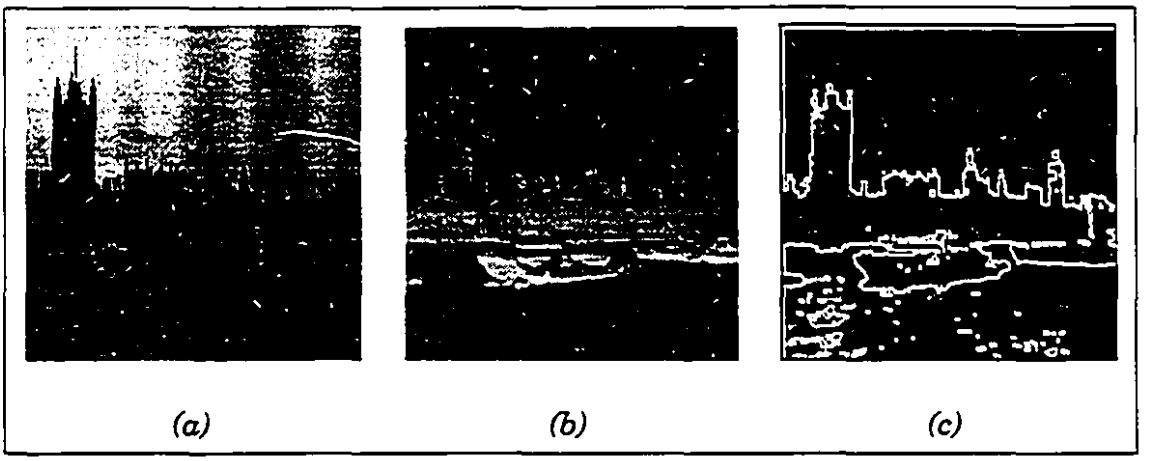

Figure 1.3: Example of edge-detection with a Laplacian filter. (a) Original image. (b) Output image after convolution with a Laplacian kernel. (c) Detection of zero-crossings.

erator plays in the realm of image-processing.

It shall be seen shortly that convolution's formidable image-processing power comes at the expense of a relatively lengthy execution time when it is performed on general-purpose uniprocessor architectures. Parallel processing architectures, on the other hand, will then be presented to be much more suitable for convolution implementation allowing for a considerable amount of speedup.

1.2 Computational Considerations

The time taken by a computer to complete a convolution operation is perhaps the most important consideration in this thesis. The sheer volume of data and floating-point arithmetic calculations needed to convolve a standard sized image can be overwhelming with regard to the available computational power and memory bandwidth of single processor machines used to run image-processing programs.

Firstly, consider the convolution of a $K \times L$ image with an $N \times M$ kernel. The weighted sum of an $N \times M$ neighbourhood of pixels requires NM multi-

plication operations and NM additions. Therefore, performing these operations for each of the KL (overlapping) neighbourhoods necessitates $2KLNM$ operations in all, excluding any arithmetic overhead required in resolving software loop counters. Techniques that take advantage of kernels with certain properties exist [Gonzalez and Wintz, 1987], yet for the general case the number of arithmetic operations in a convolution operation remains in the order of $KLNM$. This amount of computation is truly significant. For instance, the edge-detection example in Fig. 1.3 employed a 9×9 convolution on a standard 512×512 image and demanded over 21 million floating-point multiplications and 21 million floating-point additions.

1.2.1 Uniprocessor Performance

It should come as no surprise that all these calculations take a relatively long period of time to execute on general-purpose uniprocessor workstations on which most image-processing programs are currently being developed. However, just how long a convolution operation takes to execute is not easily reported since execution time is dependent on a myriad of factors. For example, the total elapsed time of a convolution routine is clearly code dependent, compiler dependent, machine dependent, I/O dependent, and also operating system dependent and is therefore given to large fluctuations owing to the wide range of influence of these numerous factors. In view of its varying nature, convolution execution time can only be meaningful if related in a very definitive context with deference to the above factors. Hence, it is undertaken to establish a well defined framework for presenting the elapsed time of a convolution operation.

First, one must carefully define what one means by elapsed time. This is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead – basically everything. However, since with multiprogramming the CPU works on another program while waiting for I/O and

may not necessarily minimize the elapsed time of one program, there needs to be a term to take this activity into account. "CPU time" recognizes this distinction and means the time the CPU is computing *not* including the time waiting for I/O or running other programs. Hence, CPU time seems a fair way to cite the true duration of a convolution routine since it excludes I/O and "load" factors which can account for unpredictably large latencies that are usually unrelated to the particular task at hand. CPU time can be further divided into the CPU time spent in the program, called "user CPU time", and the CPU time spent in the operating system performing tasks required by the program, called "system CPU time". And thus to further weed out the effects of random system elements such as time spent paging image arrays, the system CPU time is neglected and only user CPU time is considered as an "honest" way to report uniprocessor performance on the convolution operation.

In coding up a simple convolution routine in order to measure the elapsed user CPU time, it was realized that the goal, in effect, was to construct a benchmark program that would not only relate a typical convolution latency but would also compare the suitability of different machines in performing convolution. The resulting benchmark code which was dubbed the "CONV" benchmark is included in Appendix A. Expressly, this benchmark records the bare processing time needed to execute the double-precision floating-point process code within the main loops of a typical convolution routine. Results of the CONV benchmark were gathered from every available workstation and are ordered and presented in Table 1.1. Since a benchmark's most important feature is reproducibility, all information relating to its compilation and execution need also be reported. The computers tested with this benchmark are specified by model and associated microprocessor.

From Table 1.1, it is noticeable that the convolution benchmark was run on the gamut of Silicon Graphics and Sun workstations. The best performance was achieved with a Silicon Graphics Indigo™ station which required at least 21 seconds to perform the convolution. At the other extreme, a Sun 3 architecture faired the worst requiring at least 15 minutes 53 seconds to accomplish the same task! Ergo,

Machine Type	Microprocessor	User CPU Time
SGI INDIGO XS	MIPS R4000 - 50 MHz	21sec
SPARCstation 10/30	SuperSPARC - 33MHz	29sec
IRIS 4D/420VGX	MIPS R3000 - 40 MHz	36sec
PERSONAL IRIS	MIPS R3000 - 36 MHz	40sec
SPARCstation 2	SPARC - 40MHz	51sec
SPARCstation ELC	SPARC - 33MHz	1min 1sec
SPARCserver 470	SPARC - 33MHz	1min 6sec
SPARCstation IPC	SPARC - 25MHz	1min 30sec
SPARCstation 1+	SPARC - 20MHz	1min 37sec
SPARCstation 1	SPARC - 20MHz	1min 58sec
SPARCstation SLC	SPARC - 20MHz	2min 7sec
Sun 3/50	Motorola 68020	15min 53sec

Table 1.1: Results of the CONV benchmark on the spectrum of available uniprocessors currently used to perform image-processing tasks. GCC version 2.3.3 was used to compile the benchmark for each architecture with exception to the SUN 3 which only supported GCC version 2.0.

even with the best machine, the requisite 42 million FLOPS in the CONV benchmark took 21 seconds to complete which amounts to nothing more than 2 double-precision MFLOPS of delivered performance. However, going through lengths to highly optimize the code in the CONV benchmark it is possible to further improve this figure of performance on the R4000-based Indigo. For instance, setting machine dependent compiler options to schedule instructions specifically for the R4000 chip and to issue instructions from level 2 of the MIPS ISA (Instruction Set Architecture) with branch prediction, unrolling all loops, and using temporary local registers to store partial convolution results, a CONV performance of 7 MFLOPS was attained. But, this is still worse than the 9.4 MFLOPS quoted for an Indigo running the LINPACK benchmark [Wilson, 1993] and worse still than the 16 MFLOPS rate published by Silicon Graphics for this same machine. The disparity can be attributed to memory latency which is discussed next.

The architectures tested above are all high-performance RISC (Reduced Instruction Set Computer) architectures each with a pipelined FPU (Floating Point Unit)

which relies on a continuous stream of incoming data to sustain maximum processing throughput. No matter how well the compiler schedules instructions, there will invariably be times when the FPU is stalled for a period, typically 2 or 3 clock cycles, awaiting input data that is in main memory. In convolution, the problem of pipeline stalls due to memory latency is especially aggravated by the vast number of inherent load (and store) instructions, limiting the already limited computational power available in single processor architectures. For example, in the CONV benchmark, each store instruction issued corresponds to a pixel result destined for the image buffer Y; yet, each of these 2^{18} results requires 81 load instructions to access the neighbourhood of pixels in I and 81 loads for the weights in W. Despite the prevalence of cache systems in all architectures tested, this volume of loads puts overwhelming demands on the CPUs' memory substems engendering the somewhat debilitated performance in uniprocessor machines that was observed with the CONV benchmark.

Generally, the general-purpose single-processor systems studied are not very well-suited to the sort of numerical processing that the convolution operation calls for. Parallel processors, alternatively, will soon be shown to more readily conform to the task at hand. But first there remains an important issue with regard to convolution or rather the computational considerations of convolution that needs addressing. Above the focus was on spatial domain computation of the convolution operator; however, in certain instances convolution is best carried out in the frequency domain. It is essential to investigate the frequency domain approach since in the cases where it proves less computationally intensive than the spatial domain approach, frequency domain calculations instead of spatial domain calculations should serve as the basis for computing convolution.

1.2.2 Transforming the Domain of the Convolution Problem

It is sometimes more efficient to compute convolution by transforming the image and kernel arrays to the frequency domain, multiplying the transforms point by point, and then inverse transforming the result. This is the thrust of the frequency domain approach which is also represented by Fig. 1.4, and in this subsection, a theoretical discussion will unfold the mysteries behind the requisite frequency domain transformations.

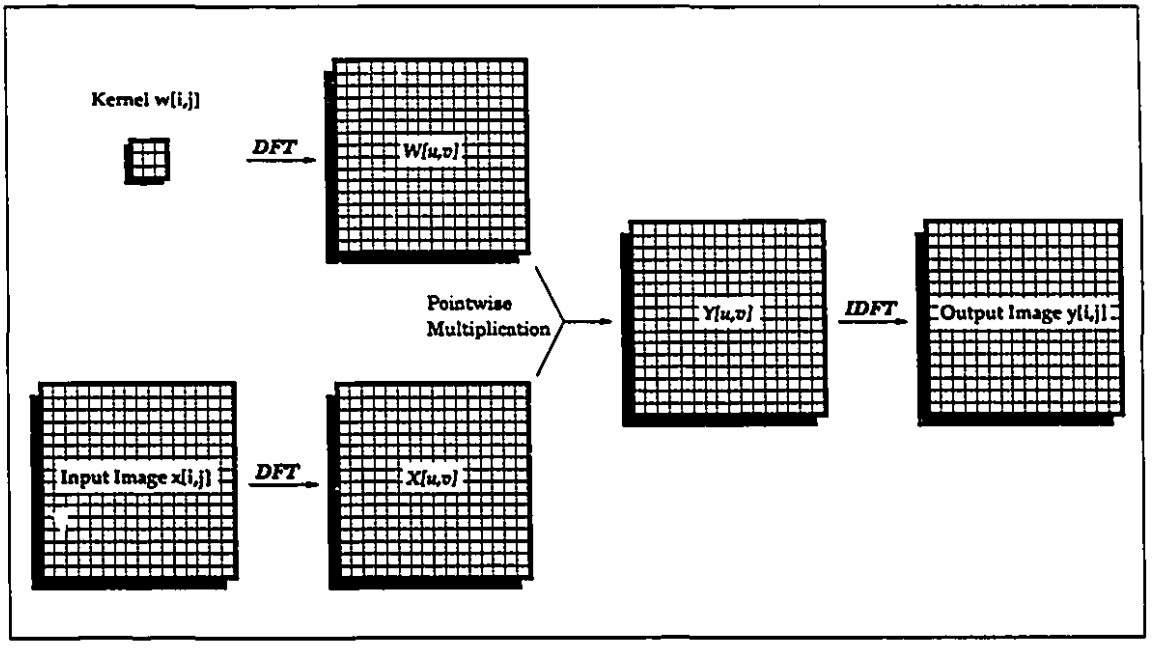

Figure 1.4: Frequency domain approach to convolution $y = w * x$. The input image $x[i, j]$ and kernel $w[i, j]$ are each passed through a DFT routine which transforms the spatial domain $[i, j]$ to the frequency domain $[u, v]$. The transforms, $W[u, v]$ and $X[u, v]$, are then multiplied point by point to yield $Y[u, v]$. The inverse transform is then applied to Y to recover the output image $y[i, j]$.

The frequency domain approach takes advantage of the Fourier duality of the convolution and multiplication operators. Specifically, by the Convolution Theorem and the Modulation or Windowing Theorem [Oppenheim and Schafer, 1989], discrete-time convolution of sequences is equivalent to multiplication of corresponding periodic Fourier transforms, and likewise, multiplication of sequences is equivalent to periodic convolution of corresponding Fourier transforms. Hence, a

simple pointwise multiplication operator should be used in lieu of the more laborious spatial convolution operator provided the discrete Fourier transform, or DFT, and its inverse, the IDFT, are painlessly computed.

When calculated by the brute-force method the two-dimensional DFT of an $N_1 \times N_2$ image which is defined by the following complex valued equation [Press *et al.*, 1988] [Burrus and Parks, 1985]

$$X[n_1, n_2] = \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2 / N_2) \exp(2\pi i k_1 n_1 / N_1)\, x[k_1, k_2] \qquad (1.8)$$

necessitates in the order of ${N_1}^2 {N_2}^2$ operations or K^4 operations if one assumes a square $K \times K$ image. This amount of computation is hardly painless, and if it were not for the recursive Fast Fourier Transform or FFT algorithm, one surely would never consider doing anything in the frequency domain. Fortunately, the FFT algorithm, whose "discovery" was credited to Cooley and Tuckey in 1965, can be used to compute the DFT in much fewer operations.

The FFT [Brassard and Bratley, 1988] is basically a divide and conquer algorithm which relates the transform of a one-dimensional sequence of N points to two sequences of $N/2$ points. It can be used recursively to subdivide the data all the way down to transforms of length 1 which are simply identity operations. The algorithm works well only when the original N is an integer power of 2; hence, most data sets which are not powers of two are padded with zeros up to the next power of two. The points as given are therefore just the one-point transforms. One combines adjacent pairs to get two-point transforms, and so on, until the first and second halves of the whole data set are combined into the final transform. Each combination takes of order N complex number multiplications and additions, and there are evidently $\log_2 N$ combinations, so the whole algorithm is of order $N \log_2 N$.

Now, by pulling the exponential of "subscript 2" outside of the sum over k_1 in Eq. 1.8, one can see instantly that the two-dimensional FFT can be computed

by taking one-dimensional FFTs sequentially on each index of the original two-dimensional array. Symbolically,

$$X[n_1, n_2] = \text{FFT-on-index-2}(\text{FFT-on-index-1}(x[k_1, k_2])) \tag{1.9}$$

Thus it is shown that a two-dimensional FFT can be efficiently implemented in this manner by first taking transforms along each row of an image and subsequently along each column of the resulting intermediate data representation. Ergo, the transform of a $K \times K$ image would be expected to take in the order of $K\log_2 K$ operations for each of the K rows plus K columns. Thus, the number of operations in a two-dimensional DFT using the FFT algorithm would be proportional to $K(K\log_2 K)$ or $K^2\log_2 K$, a considerable improvement over the brute-force DFT algorithm which requires in the order of K^4 flops.

Further along these theoretical lines, it can also be shown that the frequency domain approach for the convolution problem involves K^2 operations to carry out the pointwise multiplication of the transformed image and kernel, and subsequently, a certain number of operations for the inverse transform of this result. Since the inverse Fourier transform can also be calculated (with minor modifications in the input) by using the $K^2\log_2 K$ algorithm designed for the forward transform, then the entire convolution can be performed in that same order of $K^2\log_2 K$. However, as will be seen next, the multiplicative constant for the latter "order" expression can prove significant and that transforming the domain of the convolution problem is advantageous only for larger kernels.

1.2.3 Frequency Domain Versus Spatial Domain Processing

The results of an experiment to measure the computational intensity of each approach to the convolution problem are presented in Fig. 1.5. Convolution imple-

mentations using the frequency domain and spatial domain approaches were carried out on a uniprocessor machine and the number of floating point operations, or flops, were tallied at run-time.

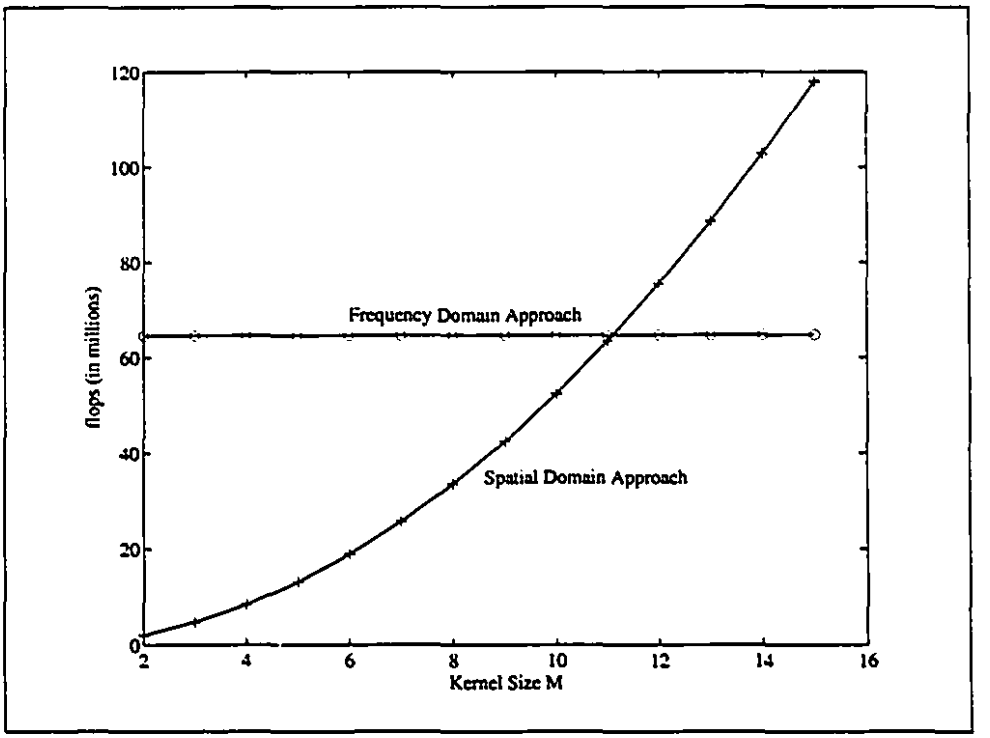

Figure 1.5: Frequency domain approach versus spatial domain approach. The number of floating-point operations required are plotted against kernel size M.

If the order expressions established above are taken as exact indications of the number of flops required for each approach, the breakeven point would be expected to occur when the K^2M^2 spatial domain flops equaled the $K^2\log_2 K$ flops. (Recall that a $K \times K$ image is being convolved with an $M \times M$ kernel.) Therefore, for a standard $K = 512$ image size, the two approaches would be of equal computational intensity when $M^2 = \log_2 K$, or simply when $M = 3$. However, the undergone experiment in which complex multiplications were counted as 6 flops and complex additions were counted as 2 flops suggested that the frequency domain approach did not become computationally expedient until the kernel size M exceeded 11. Fig. 1.6 delineates this breakeven point and summarizes the breakeven points for different values of image size K.

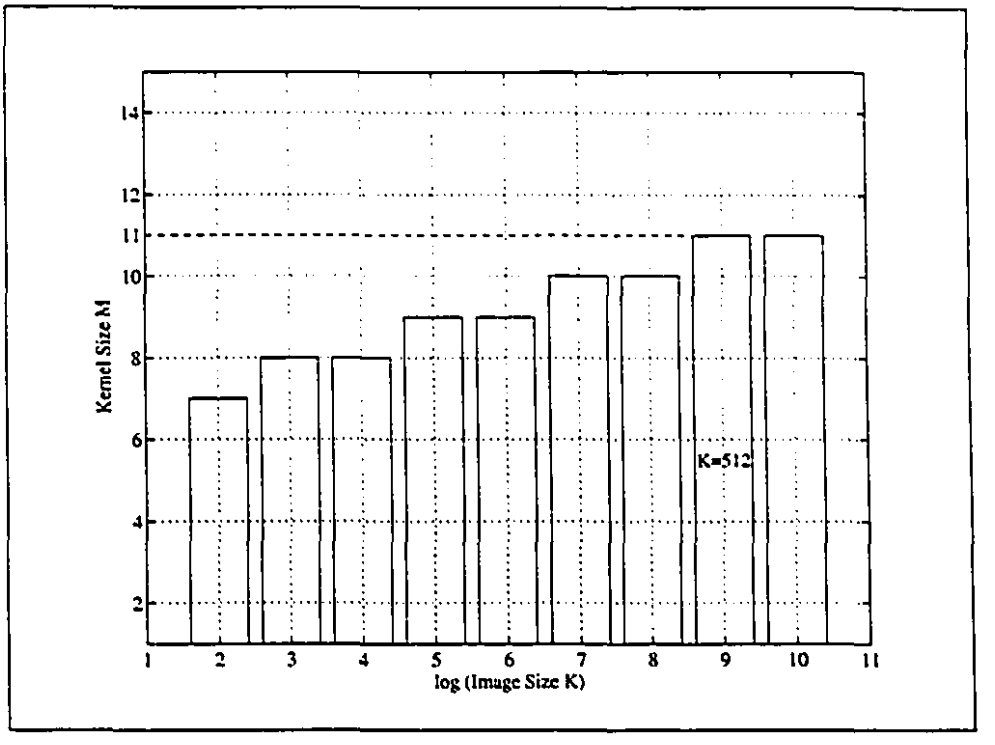

Figure 1.6: Maximum kernel size M for which convolution in the spatial domain used less flops than convolution done in the frequency domain. Values of M are plotted for different values of image size K.

It should be cautioned that these results are largely implementation dependent, yet they are interesting because they serve to dispel any misconceptions that the frequency domain approach might be a panacea for the convolution problem. Quite to the contrary, all observations indicate that the frequency domain approach is only effective for convolutions that employ large kernels. The great overhead in flops and the multitude of calls to memory needed to transform the domain of the problem cannot be justified for convolutions that use small kernel sizes. In these cases where the kernel is much smaller than the input image, which will also be a requirement for the hardware implementation to come, convolution is best handled in the spatial domain.

1.3 Summary

To summarize, the convolution operation was targeted for performance improvement because it was perceived as the most frequently used operation in image-processing tasks. The convolution operator was therefore introduced and defined and followed by some practical examples which reinforced the latter definitions. Furthermore, since much of the development of image processing programs occurs on relatively low-cost, general-purpose, uniprocessor workstations, convolution performance was first evaluated on such machines. In addition, whether domain transformations are employed or not, the essential point remains that the huge volume of data and floating-point arithmetic calculations needed to convolve a standard sized image can be overwhelming with regard to the available computational power and memory bandwidth of single processor machines. Alternatively, in the next chapter, parallel processing will be presented as the logical answer to overcoming the tremendous computing requirements of the convolution operation.

Chapter 2 Parallel Processing Solutions

The point of parallel processing is to reduce the elapsed time to complete the job at hand [Karp, 1987]. Executed on a uniprocessor that can sustain a 1-MFLOPS computation rate, an implementation of a sequential algorithm for convolution that engenders say 42,000,000 floating point operations can run to completion in about 42 seconds. Now, if the convolution problem can be reformulated into a parallel algorithm and implemented to run concurrently on say 10 processors each capable of 1 MFLOPS, the convolution job should ideally take one tenth the processing time or 4.2 seconds. Although it is implied that x processors give a speedup of x, real systems usually fall short of such ideal speedups hindered by inefficiencies due to synchronization, communication, or work imbalances among the multiple parallel processors. Essentially however, in spite of the realities imposed by coding style, the architecture of the machine, and the specific hardware implementation, parallel processing stands as the only feasible means by which one can attain dramatic performance improvement [van Zee and van de Vorst, 1989].

Expressly, the convolution problem, which is inherently parallel in nature, can be quite easily "parallelized", and as such can be readily implemented with considerable speedup on various general-purpose parallel processing computers. However, this chapter will attempt to instill the notion that a general-purpose parallel computer is not a cost-effective option for the rather specialized problem of convolution, and that maximum gain and cost-efficiency can best be achieved with a special-purpose hardware accelerator employing a type of synchronous parallel architecture known as the systolic array.

2.1 Inherent Parallelism

The convolution computational problem is inherently parallel in nature. This characterization simply implies that by virtue of its iterative nature, the convolution task is very easily divided into smaller subtasks which can be completed in parallel. Observe the sequential algorithm which is typified by the main loops in the CONV benchmark:

```
for (i=IBEGIN; i<=IEND; ++i)
    for (j=JBEGIN; j<=JEND; ++j)
        for (m=MBEGIN; m<=MEND; ++m)
            for (n=NBEGIN; n<=NEND; ++n)
                Y[i][j]+= W[m][n] * I[i+m-IOFFSET][j+n-JOFFSET];
```

This algorithm basically uses two pairs of nested loops to index the convolution sum iteratively. Moreover, indices $[i, j]$ index the output image data, $[m, n]$ index the kernel data, and the input image data is indexed by a combination thereof. This algorithm can clearly be divided (and conquered) in countless ways. One important approach finely partitions the compute work into small identical and independent subtasks which operate concurrently on different portions of data. This innate fine-grain data parallelism is readily demonstrated by unrolling the outer pair of loops of the algorithm above to yield a sequence of copies of the inner double-loops.

```
for (m=MBEGIN; m<=MEND; ++m)
    for (n=NBEGIN; n<=NEND; ++n)
        Y[0][0]+= W[m][n] * I[0+m-IOFFSET][0+n-JOFFSET];
for (m=MBEGIN; m<=MEND; ++m)
    for (n=NBEGIN; n<=NEND; ++n)
        Y[0][1]+= W[m][n] * I[0+m-IOFFSET][1+n-JOFFSET];
for (m=MBEGIN; m<=MEND; ++m)
    for (n=NBEGIN; n<=NEND; ++n)
        Y[0][2]+= W[m][n] * I[0+m-IOFFSET][2+n-JOFFSET];
...

for (m=MBEGIN; m<=MEND; ++m)
    for (n=NBEGIN; n<=NEND; ++n)
        Y[511][511]+= W[m][n] * I[511+m-IOFFSET][511+n-JOFFSET];
```

Here, each double-loop is identical in that each serves to accumulate the weighted sum of a given neighbourhood of input pixels, and each is responsible

for a one pixel result in the output buffer Y. Hence, each double-loop, which represents a rather small number of iterations (81 for the CONV benchmark), may be executed in parallel each on a separate processor. If there existed a parallel computer with as many processors as double-loops in the above routine, a CONV benchmark parallelized in this fashion would exhibit an ideal speedup of $512^2 = 262,144$!

By virtue of its inherent parallelism, the above sequential routine was quite effortlessly shown to be data-parallelizable in a fine-grain manner; yet attempts to break down the compute work further into finer, identical grains may present more of a challenge. Even though more "available parallelism" has already been demonstrated than current parallel computers (*c*1993) can exploit, it is worth briefly trying to discern still a higher degree of parallelism because the attempt points to a general problem in the art and science of converting sequential algorithms into parallel algorithms –the problem of data dependence. First, a basic premise for the concurrent operation of an array of parallel processors is that no processor's current computations should depend on the current computations of another processor. Now, consider unrolling each one of the double loops above in hopes of running each atomic iteration on a single processor. For instance, unrolling only the fist double-loop gives:

```
Y[0][0]+= W[0][0] * I[0+0-IOFFSET][0+0-JOFFSET];
Y[0][0]+= W[0][1] * I[0+0-IOFFSET][0+1-JOFFSET];
Y[0][0]+= W[0][2] * I[0+0-IOFFSET][0+2-JOFFSET];
...
Y[0][0]+= W[8][8] * I[0+8-IOFFSET][0+8-JOFFSET];
```

And it could be seen that each of these iterations/instructions operates on different input data, yet each stores its cumulative result in the same $Y[i][j]$. This implies an output data dependence. Generally, if each iteration within each double-loop is run in parallel on independent processors, then each processor will try (undesirably so) to update a global value of $Y[i][j]$ at the same time. Rather, each of these processors should have to wait its turn to update Y if the correct weighted sums are to be accumulated. Thus the sequential nature of the latter accumulation defeats the purpose of concurrent processing. Indeed, the pervasive point to be made is that most problems of interest, convolution included, possess certain inherently

sequential algorithmic components [Kuck, 1980] that present themselves in terms of data dependencies which invariably complicate or even limit the parallelization process. Section 2.4 will elaborate on how parallelization of such sequential components can be performed.

2.2 Basics of Parallel Algorithms

Thus far, parallelism has been investigated with no mention as to parallel architecture specifics. Merely the form of the sequential algorithm and the sort of parallelization that it conjectures have been presented. Hence, before exact models of parallel processing are introduced, it would be appropriate to touch upon, in some semblance of rigour, two important parameters with which parallel algorithms have and will be compared.

Within this thesis, T is denoted as the execution time for the "best" serial algorithm, and T_p as the execution time for a parallel algorithm using p processors. The speedup, S_p, can therefore be defined as

$$S_p = \frac{T}{T_p} \tag{2.1}$$

And the "efficiency" or "utilization", E_p, of the parallel algorithm is given by

$$E_p = \frac{S_p}{p} \tag{2.2}$$

If a parallel algorithm is 100% efficient, then one observes "linear" speedups. As intimated at the outset of this chapter, however, efficiency is practically never 100%, due to synchronization or communication costs. And in those cases where task granularity is irregular, suboptimal load balancing among parallel processors also works to further erode this figure.

2.3 General-Purpose Parallel Processing Options

The road to parallelizing convolution is a convoluted one. To begin, parallel processing solutions offer large-scale speedups for the convolution problem and many other inherently parallel problems. From a practical standpoint, commercially available, general-purpose, parallel processing computers can indeed be very effective, but it is most consequential that their high price tags place them beyond the reach of many academic and industrial organizations. Table 2.1 provides a cursory look at the cost of the state of the art in parallel computing. The cheapest option is shown to be a network of uniprocessor workstations. Albeit for communication intensive tasks such as the convolution of an image, the extensive communication overhead in using network constructs such as sockets [Horspool, 1986] or higher-level remote procedure calls (RPCs) negates the associated increase in computing power. At the other end of the spectrum, supercomputers are recognized as often the most appropriate resource for performing certain complex and important tasks [Rattner, 1985], but evidently they are prohibitively expensive.

Parallel Processing Options	Examples	Cost
Network of workstations	Network of SPARCstations	Lowest cost
Multiprocessor workstations	DEC Firefly, Apollo DN 10000, Solbourne, Xerox Dragon, SUN SPARCstation 20	$60,000.
Shared memory multiprocessors	Sequent Symmetry, Encore Multimax	$200–400,000.
Distributed memory multiprocessors	Intel iPSC hypercube, NCUBE	$200–400,000.
Supercomputers	Connection Machine CM-5, Intel Paragon, Kendall Squares KSR-1	$5,000,000.

Table 2.1: Cost of various parallel processing options.

Having secured the significance of the cost factor in the search for optimal par-

allel solutions for convolution, what remains is to establish an order of logic with which to lead into one such cost-effective solution. Finding one's way through the "megalopolis" of parallel processing options requires some sort of map. Hence, this section will take on the "big picture" of parallel computing, putting order to the parallel-processing options via a taxonomic survey. The parallel architectures to be surveyed in this section are mostly general-purpose in nature, hence the title above. Yet the subclass of architecture that will prove most cost-effective and thus most fundamental to this thesis will be special-purpose, and its discussion is deferred for the subsequent section (Section 2.4).

2.3.1 Architecture Taxonomy

The diversity of recently introduced parallel computer architectures confronts the taxonomist with what R.W. Hockney felicitously terms "a confusing menagerie of computer designs" [Hockney and Jesshope, 1988]. Placing the architectural alternatives in a coherent framework requires the adoption of an uptodate taxonomic system. According to Flynn's taxonomy [Flynn, 1966] which classifies computers based on their instruction and data streams, parallel architectures would fall under the multiple-instruction, multiple-data (MIMD) and single-instruction, multiple-data (SIMD) classifications. Although these distinctions provide a useful shorthand for characterizing architectures, they are insufficient for classifying various modern computers. Other taxonomies exist [Shore, 1973], but favoured in this thesis is a more contemporary grouping proposed by Duncan [Duncan, 1990]. This classification scheme, sketched in Fig. 2.1 leads one to consider processors in terms of MIMD, MIMD-based paradigms, and synchronous architectures.

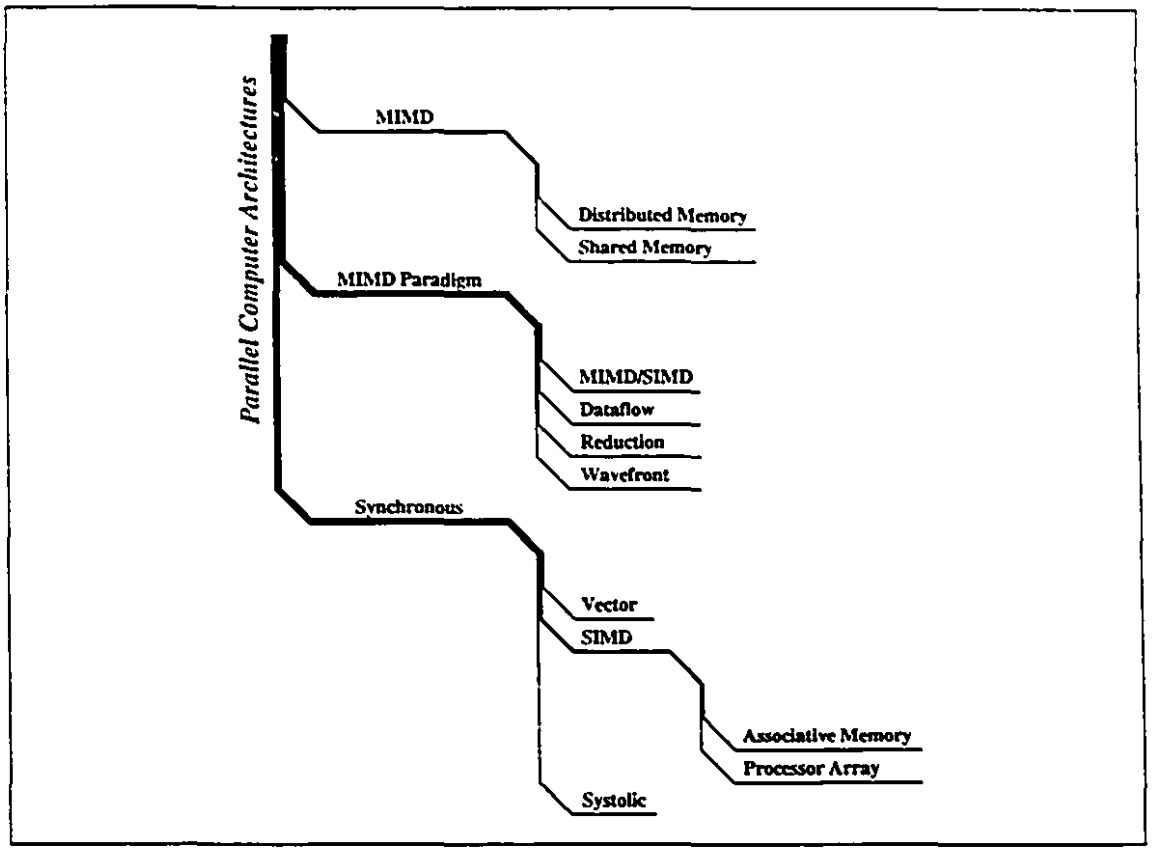

Figure 2.1: Parallel computer architectures according to Duncan's taxonomy.

2.3.2 MIMD Parallel Processors

MIMD architectures employ multiple processors that can execute independent instruction streams, using local data. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Rattner refers to this kind of parallel execution as "concurrency" [Rattner, 1985] which is the highest level form of parallelism, denoting independent operation of a collection of simultaneous computing activities. That is to say that a given task is divided coursely into a number of sizable subtasks which are executed asynchronously on independent processors. Depending on how their associated subtasks (software processes) interact, these processors can be classified as loosely coupled, interacting by passing messages to one another, or tightly-coupled, interacting via shared memory.

Loosely coupled systems are also referred to as distributed memory architectures because memory is not centralized but distributed locally on each processor. The only way for the application to share data among processors in these systems is for the programmer to explicitly code commands to move data from one processor to another. Examples of recent distributed memory, message passing machines are the Parsytec GC [Par, 1991], the nCube 2 [nCU, 1992] and the iPSC 860 [Int, 1993]. Although many processor interconnection schemes exist [Feng, 1981], the nCUBE and iPSC systems are noted for exploiting powerful hypercube interconnection networks [Seitz, 1985] [Palmer, 1986] [Freer, 1987] which boast the densest and most efficient inter-processor wiring.

Alternatively, shared-memory architectures such as the Sequent Symmetry [Lovett and Thakkar, 1988] accomplish inter-processor coordination by providing a global, shared memory that each processor can address. While computers in this subclass do not have some of the problems encountered by message-passing architectures, such as message sending latency as data is queued and forwarded by intermediate processor nodes, they do suffer from other problems such as data access synchronization and cache coherency. Also, the aggregate memory bandwidth will limit the number of processors that can be accommodated on these systems.

Naturally, the third form in MIMD architectures is manifested in hybrid systems such as the IBM RP3 [Pfister, 1985], the BBN Butterfly [Crowther, 1985], and the Cedar [Gajski, 1986]. A machine in this subclass has some of the properties of shared memory systems and some of those of message passing. Though all memory is actually local to a given processor, the operating system makes the machine look like it has a single, global memory. Thus, programs are written as if for a shared memory system; however, the performance considerations resemble those of a message passing machine.

MIMD computers are generally considered course-grain machines that perform well on problems that have a low degree of available parallelism and a low propor-

tion of communication among subtasks. As such, their design philosophy does not seem particularly suited to the inherent, fine-grain, data parallelism in the convolution problem and other fine grain tasks prevalent in image processing algorithms. For instance, convolution can be handled by dividing the input image into chunks to be worked upon separately amongst the n available processors of a MIMD system. Explicitly, a MIMD convolution program would likely "fork" n corresponding processes that would operate concurrently on n "sub-images". Since n tends to be relatively small (< 1000), the speedups for MIMD convolution are less than dramatic. Moreover, efficiency would presumably be poor as MIMD parallel convolution incurs a profound communication cost due to either an excessive number of shared-memory accesses (shared-memory contention) or an immoderate number of messages for the access of border data in neighbouring sub-images.

2.3.3 MIMD-based Parallel Processing Paradigms

MIMD/SIMD hybrids, dataflow architectures, reduction machines, and wavefront arrays are a some of the parallel processing models that do not readily befit Flynn's MIMD categorization. Although these paradigms are predicated on MIMD principles of asynchronous operation and concurrent manipulation of multiple instruction and data streams, each of these architectures is also based on a distinctive organizing principle as fundamental to its overall design as its MIMD characteristics.

Firstly, MIMD/SIMD hybrid models [Lipovski and Malek, 1987] allow selected portions of a MIMD architecture to be controlled in SIMD fashion. Although these models are interesting from an image-processing perspective, it will be argued shortly that convolution and other low-level vision algorithms are best supported by maximizing the SIMD resource.

Dataflow architectures [Srini, 1986] feature an execution paradigm in which instructions are enabled for execution as soon as all of their operands be-

come available. Thus, these "data-driven" processors such as the Datawave [Schmidt and Caesar, 1991] fair well for asynchronous tasks with numerous data dependencies and have been shown to achieve high degrees of concurrency. But despite all their merits, dataflow remains the architecture of choice only when simultaneity is low, irregular, and run-time dependent [Briggs and Hwang, 1984] – characteristics which are the antithesis of low-level image processing.

Reduction architectures, also referred to as "demand-driven" architectures [Treleaven *et al.*, 1982], implement an execution paradigm in which an instruction is enabled for execution when its results are required as operands for another instruction already enabled for execution. Like the data-driven architectures above, reduction architectures are not a practical choice for the low-level processing of images.

Lastly, wavefront array processors [Kung, 1987] are based on a MIMD architectural paradigm which combines an asynchronous dataflow execution model with "systolic data pipelining". Pipelines and systolic arrays will be the focus of Section 2.4 wherein it will be seen that simple, synchronized dataflow contributes to the many desirable properties of systolic architectures. However, wavefront arrays replace the global clock and explicit time delays used for synchronizing systolic data pipelines with asynchronous handshaking as the mechanism for coordinating inter-processor data movement. Thus wavefront arrays, despite their close resemblance to the impending systolic solution, increase control complexity and will be forsaken in favour of their simpler sibling -the systolic architecture.

2.3.4 Synchronous Parallel Processing Architectures

Synchronous computers essentially include vector, SIMD and systolic processors all of which perform concurrent operations in lockstep fashion, synchronized with either control units or global clocks. Hence, unlike the autonomous MIMD proces-

sors, synchronous processors perform in a very deterministic manner. The goal of synchronous architectures appears to be the fine division of compute work among multiple processors which makes them excellent candidates for fine-grain image-processing problems such as convolution.

Pipelined vector processors [Briggs and Hwang, 1984] such as the Cray X-MP [Larson, 1984] [Robbins and Robbins, 1989] and the IBM 3090 [IBM, 1985] are characterized by multiple, pipelined functional units which implement arithmetic and Boolean operations for both vectors and scalars and which can operate concurrently. Since such architectures can support task-level parallelism, they could arguably be termed MIMD architectures, although vector processing capabilities are the fundamental aspects of their design. The pipelined processing units in vector machines are said to exploit low-level "temporal parallelism" by performing a vector operation in stages and by overlapping execution such that at any given time each stage in the vector pipeline is processing a different element in the vector stream.

SIMD computers, on the other hand, will be shown to exploit a higher level "data-parallelism" [Hord, 1990]. These architectures, which can be further classified in terms of associative memory systems and and processor arrays, have traditionally employed a central control unit, multiple processors, and an interconnection network for either processor-to-processor or processor-to-memory communications. The control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. Individual processors may be allowed to disable or "sit out" the current instruction.

Computers built around associative memories [Kohonen, 1987] form a distinctive type of SIMD architecture that uses special comparison logic to access stored data in parallel according to its contents. Modern associative memory processors are not particularly suited to image-processing applications; rather, they have naturally been geared to database-oriented applications, such as tracking and surveil-

lance.

Finally, image-processing seems to have gained much from the SIMD architectural paradigm as realized by general-purpose, processor array architectures. Historically, processor arrays such as the Illiac IV [Barnes, 1968], the MPP (massively parallel processor) [Batcher, 1980], the ICL DAP (distributed array processor) [Hockney and Jesshope, 1988], and more recently, machines such as the ACMAA (access constrained memory array architecture) [Balsara and Irwin, 1991] and the Connection Machine [Hillis, 1985] [Thi, 1992], have often been used as image-processing "engines". These parallel computers typically employ a high number (> 1000) of simple processors or PE's (processing elements). Although they lack the autonomy and computing power of MIMD processors, the PE's within large processor arrays have been shown to make short work of inherently, fine-grain, data-parallel problems.

In keeping with the "practical" nature of this thesis, it is highly desirable to digress somewhat in order to substantiate this latter claim by demonstrating the efficacy of an array processor in solving the convolution problem. To this effect, an experiment was conducted that does indeed demonstrate the effectiveness of one array processor in as much as it was found to yield the best convolution performance presented thus far in this thesis. But when the cost of this array processor is factored in, this parallel solution can hardly be deemed "cost-effective".

The MasPar Experiment

The MasPar experiment entailed the parallelization of the sequential convolution algorithm as suggested in Section 2.1 and the subsequent implementation on an available MasPar MP-1 [Mas, 1992] array processor. The resulting parallel code constitutes a new benchmark called "PARCONV" and is featured in Appendix B. The MP-1 employs 2,048 PE's configured in a 64×32 toroidal mesh shown in Fig. 2.2. As was done in the CONV benchmark, the time taken to execute the

double-precision floating-point process code within the computational portion of the PARCONV benchmark was measured, although instead of user CPU time, this figure is referred to as "DPU time" to reflect the isolated processing time of the Data Parallel Unit (See Appendix B). Hence, the DPU time (or T_{2048}) was observed to be 3.69 seconds implying a processing rate of 11.4 MFLOPS. Numerous attempts were made to further optimize the code in the PARCONV benchmark yet the fastest version that utilized specialized "uni-directional" routines for inter-processor pixel transfers presented a mere 13.4 MFLOPS. When compared to the sequential algorithm executed on the Silicon Graphics Indigo station this implies a speedup, S_{2048}, of about 2 with a near nil efficiency. In all fairness, however, it is important to remark that this speedup is substantial when one considers that the MP-1 employs very simple 4-bit processing elements each running at 1.8 MIPS (millions of non-floating-point instructions per second) and that the 64-bit, double-precision, floating-point performance of one PE is orders of magnitude below that of the Indigo's R-4000 processor.

But perhaps the most important consideration is the cost-effectiveness of the MP-1 processor array. Given that the MasPar MP-1's price was $100,000 four years ago and assuming it is the same today, one can estimate the cost per MFLOPS of convolution processing power to be $7,500/MFLOPS. Hence, with regard to this cost/performance metric, it is highly questionable whether floating-point convolution on the MP-1 is a cost-effective operation.

Taking a step back to review the architectures surveyed, it is clear that of the three classes of parallel computer architecture studied, synchronous architectures conform most to the fine-granularity of image-processing tasks. In particular, SIMD array processors were recognized as the natural choice in implementing inherently data-parallel problems. The MasPar experiment illustrated how an array processor befits a parallelized convolution routine and how the MP-1 supports the highest convolution computation rate reported so far. However, there remains one last subclass of synchronous architecture which has yet to be presented, namely

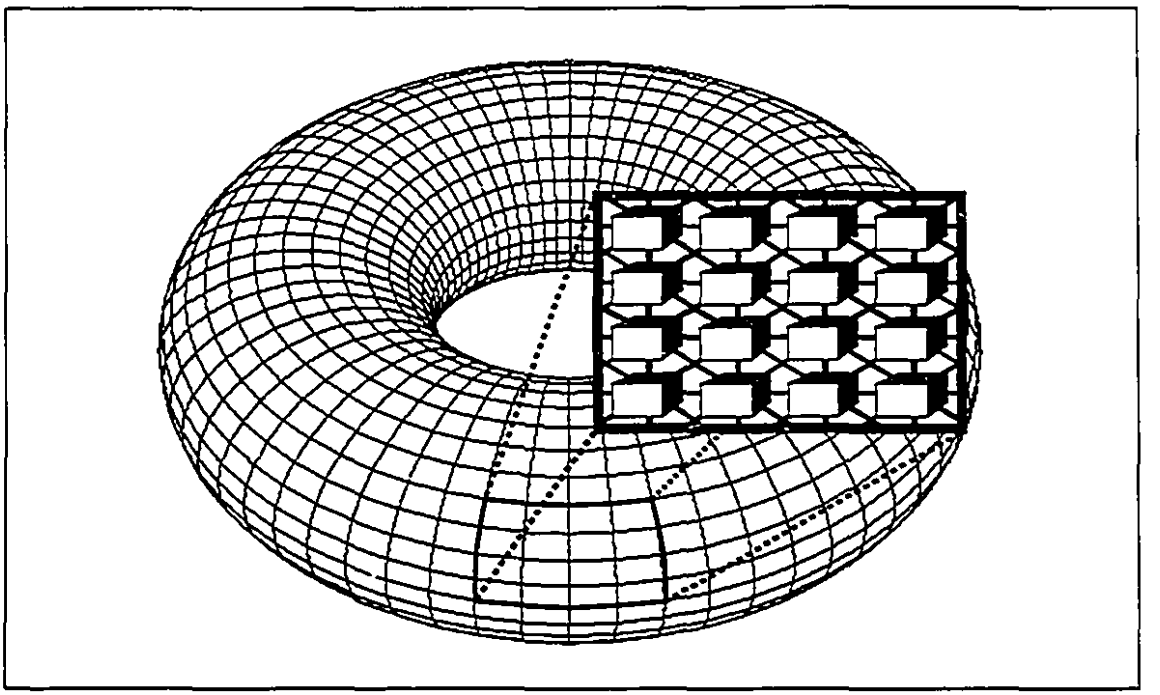

Figure 2.2: MasPar MP-1 interconnection network. The 2,048 PE's are configured in a 64 × 32 toroidal mesh. Each PE in the array is connected to its eight nearest neighbours.

the systolic array. It is introduced next as a special-purpose synchronous architecture which has dramatic performance potential. More importantly however, parallelization by means of a systolic array will be offered as an inexpensive, and therefore highly cost-effective, parallel processing solution which will be adopted in a high-performance image-processing system thereby forming the foundation for the main work in this thesis.

2.4 Systolic Array Architectures

Traversing the highways and byways of parallel computing in search of an architecture which will support a high performance, cost-effective convolution implementation, one ultimately ends up in the domain of systolic array architectures. First proposed by H.T. Kung [Kung, 1982], the systolic array is a type of parallel, synchronous architecture whose use has been chiefly application-specific. Thus, the context is being switched from general-purpose computer systems to special-purpose computer systems or to what will be alternately referred to in this thesis as "hardware accelerators" implying high-performance, special purpose systems which are typically used in meeting specific application requirements or in off-loading computations that are especially taxing to general-purpose computers. Principally, the systolic array architecture will be presented as a general methodology for mapping high-level computations into high-performance, cost-effective hardware structures.

2.4.1 Basic Architecture

Pipelines and systolic arrays represent a general means by which sequential algorithms can be parallelized. A pipelined system, for instance, operates like an automobile assembly line in which different people work on the same car at different times and many cars are assembled simultaneously. Analogously, input data enters at one end of a computation pipeline and partial results flow from stage to stage until the last stage completes the computation and yields the result. Systolic arrays function much in the same manner yet they are not constrained to operate in linear schemes characteristic of pipelines. Rather, as shown in Fig. 2.3, they can be rectangular, triangular, or hexagonal to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple direc-

tions – both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. Examples of two-dimensional systolic arrays are plentiful [Guibas *et al.*, 1979] [Lehman and Kung, 1980] [Kung and Leiserson, 1978] [Gentleman and Kung, 1981].

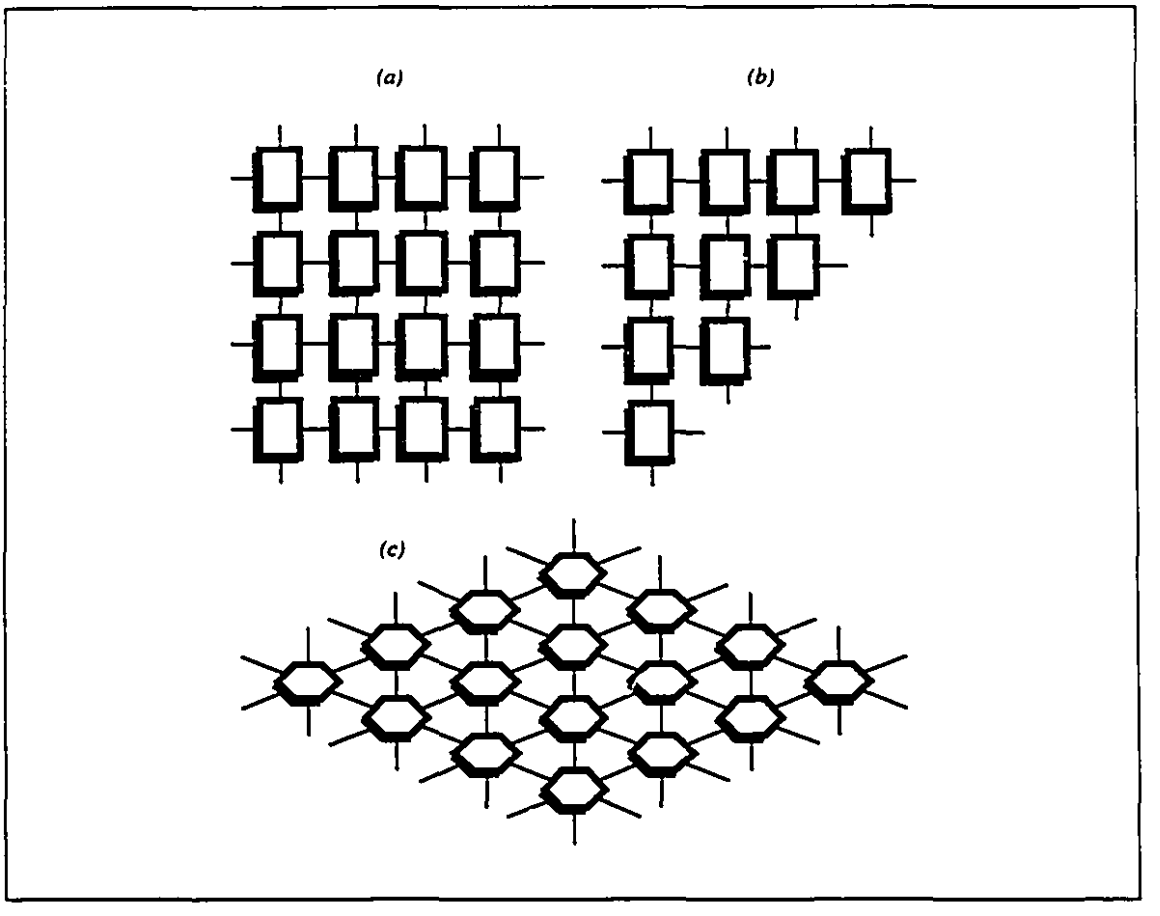

Figure 2.3: Two-dimensional systolic array connection topologies are typically (a) rectangular, (b) triangular, or (c) hexagonal.

A systolic system primarily consists of a set of interconnected cells, each capable of performing some simple operation. Simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation; therefore, cells in a systolic system, which are basically simple processing elements, are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows among cells in a pipelined fashion, and communication with the outside world occurs only at the boundary cells. Kung likens this 'rhythmically recurrent' flow of data to the flow of blood within the body and

thus adopted the medical term 'systolic' for this subclass of architectures.

> "...data flows from the computer memory in a rhythmic fashion, passing though many processing elements before it returns to memory, much as blood circulates to and from the heart." [Kung, 1982]

2.4.2 Key Architectural Considerations

Discussion on special purpose computer systems in this thesis represents a narrowing of scope or search space for systems which can handily overcome the convolution problem. To high-light the allure of the systolic subclass of architectures, we must first review some key architectural factors that constrain the design space of a special-purpose system. These elements, which will ultimately lead us toward a systolic approach, are essentially cost-effectiveness, computation rate, and I/O bandwidth.

Firstly, cost-effectiveness is a crucial consideration. The cost of a special-purpose system must be low enough to justify its limited applicability. Especially in VLSI designs, where a single chip comprises hundreds of thousands of components, great savings can be achieved if we were to choose an architecture that readily lends itself to decomposition into a few types of simple substructures or building blocks, which are used repetitively with simple interfaces. The proposed systolic architecture would be quite appropriate in this regard. In addition, such special-purpose systems based on simple, regular designs are likely to be modular and therefore easily adjustable to various cost/performance goals.

Secondly, as we have mentioned before, a high computation rate, which is a constant challenge to the special-purpose system designer, must be derived from the concurrent use of many processing elements. Though the degree of concurrency in special-purpose systems is largely determined by the underlying algorithm which should be designed to introduce high degrees of pipelining and multiprocessing,

the burden of fully utilizing the available parallelism rests on architectures such as the systolic type which can realize massive concurrency in a relatively straight forward and facile manner. In turn, the consequences of massive parallelism are manifested in terms of synchronization problems. When a large number of processing elements work simultaneously, coordination and communication become significant and more so in VLSI technologies where routing costs dominate the power, time and area required to implement a computation. Hence the simple, regular communication and control found in systolic systems is extremely welcome.

Finally, I/O considerations greatly influence overall performance. The ultimate goal in any special-purpose system is to balance the computation rate with that of the available host I/O bandwidth. Furthermore, in Fig. 2.4 we may observe that if the performance of a traditional computation model is limited by its I/O bandwidth, then no matter how fast the special-purpose system operates, there can be no improvement in throughput. However, orders of magnitude in throughput can be gained if multiple computations are performed per I/O access such as in the systolic approach. If the small systolic array in Fig. 2.4 uses each data item six times within six different overlapped operations, then the throughput or the amount of work done in a given time will increase six fold. Lastly, since an accurate a priori estimate of available I/O bandwidth in a complex system is usually impossible, a modular or systolic design will certainly prove auspicious as it is more easily adjusted to match a variety of I/O bandwidths.

2.4.3 Balanced Systems

In the above discussion we have alluded that the systolic architecture which possesses the said features of simplicity, regularity, and modularity, is quite amenable to the requirements of our special-purpose system. A systolic architecture would indeed seem optimal if only the algorithm that we hope to implement also lends itself to a systolic solution. The next subsection will show that this is truly the case,

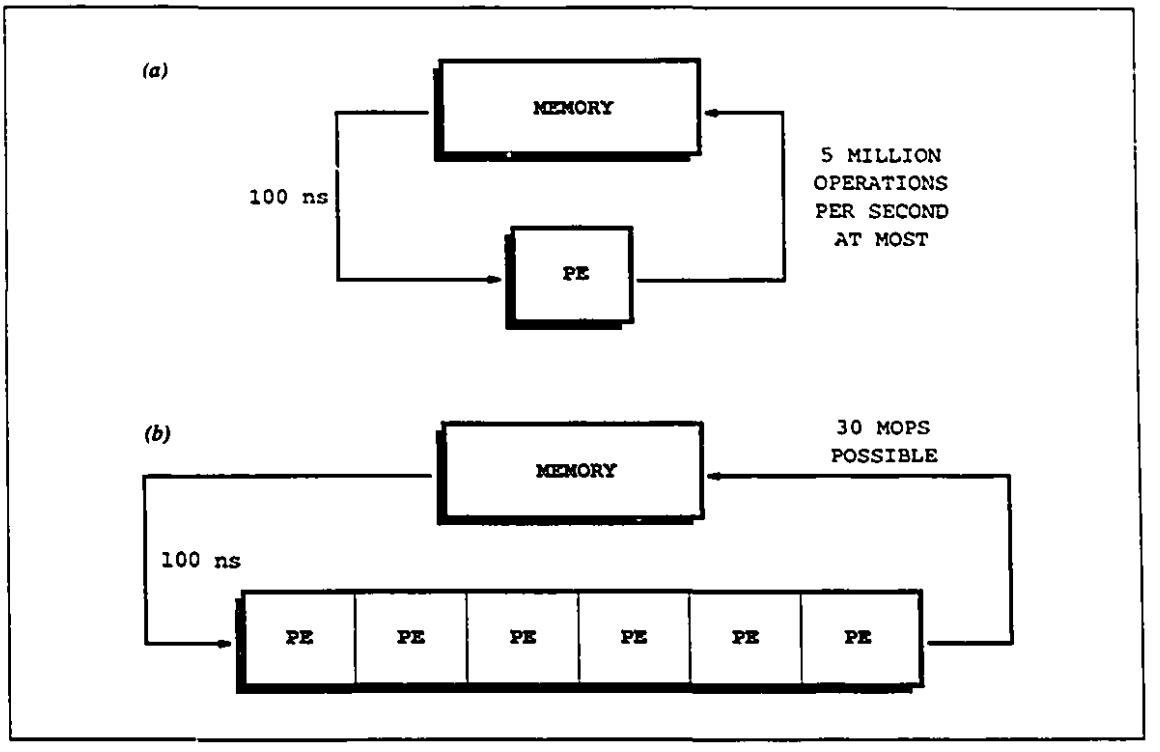

Figure 2.4: Comparison between (a) the traditional SISD computation model and (b) the systolic computation model.

but before we proceed in this vain let us further develop the concept of a balanced system.

Recall our earlier discussion about balanced special-purpose systems. We stated that balancing computation rate and I/O bandwidth is the highest goal in special-purpose system design. Ideally, we would like the system to fetch and write data at the fastest possible rate, while all the processing occurs simultaneously. Since I/O bandwidth is usually very difficult to improve without a great increase in cost, it is the computation rate that designers usually try to improve upon so as to make I/O bandwidth the only rate determining factor in system throughput. We shall call such ideal systems optimally balanced because they achieve the maximum throughput allowed by the I/O bottleneck.

Now, depending on the nature of the memory, the special-purpose processor, and the type and number of elementary operations to be performed to complete a certain routine, the throughput may be either limited by the computation rate or

the memory access rate. The former implies an 'I/O-bound' problem and the latter is 'compute-bound'. In I/O-bound tasks the memory cannot feed the processor fast enough whereas compute-bound tasks are characterized by the processor's inability to compute at the rate dictated by the I/O bandwidth.

2.4.4 Convolution Revisited

The convolution problem can be viewed as an algorithm which combines two data streams, namely the kernel weights W(i,j) and the pixel values I(i,j), in a certain manner to form the resultant data stream Y(i,j). Moreover, convolution and numerous other routines such as filtering, pattern matching, correlation, interpolation, polynomial evaluation (including discrete Fourier transforms), and polynomial multiplication and division can be classified as being compute-bound tasks and can thus be sped up with a systolic approach.

The notion of a compute-bound task is defined by Kung as a task in which the total number of elementary operations performed is greater than the total number of input and output elements. Otherwise the task is considered to be I/O-bound. This definition presupposes that whenever the number of operations exceeds the memory accesses, the special-purpose processor will be compute-bound and will not be able to compute results as fast as the memory is able to read and write. We may accept Kung's definition as a rough measure for the purposes of general problem classification, but we must always keep in mind that an a priori classification cannot be accomplished without the actual system parameters.

Hence, using Kung's definition, we may classify convolution and other pixel-group routines as compute-bound tasks. In the convolution of an $N \times N$ image with an $M \times M$ kernel, for example, we must load $M^2 + N^2$ operands (and store the N^2 results), yet perform considerably more ($2M^2N^2$) elementary operations on the incoming data. Thus the throughput is likely to be constrained by the computation

rate.

Speeding up compute-bound routines such as convolution, may often be accomplished in a relatively simple and inexpensive manner, that is, by the systolic approach. By replacing a single processing element with an array of PE's or cells, as illustrated in 2.4, the computation throughput can be increased until the system is optimally balanced. As Kung points out, the crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell as it is "pumped" from cell to cell in the array. And this is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner.

As an example of the systolic approach, Kung's "W2" architectural design is implemented at the register transfer logic level in a VHDL (Very High Speed Integrated Circuit Hardware Description Language) simulator and the results are represented in Fig. 2.5. The VHDL implementation is referred to as SYST1 and appears in Appendix D. The approach here is to configure an array of processing cells as shown in Fig. 2.5(a) each of which has the characteristic functions for Y_{out} and X_{out} shown in Fig. 2.5(b). Though the weights W (which are equal to 2 for this example) stay stored in the individual cells, it is important to note that the partial results Y move twice as fast as the inputs X yielding the correct first convolution sum ($X1W1 + X2W2 + X3W3$) after an initial latency as shown Fig. 2.5(c).

2.5 Summary

Of the parallel processor classes considered, the synchronous class appears to be the best suited for low-level image-processing applications. Specifically, since systolic arrays are simple, modular, expandable, and yield high performance, they meet the architectural challenges of special-purpose systems, and are therefore the preferred parallel processing solution for the convolution problem.

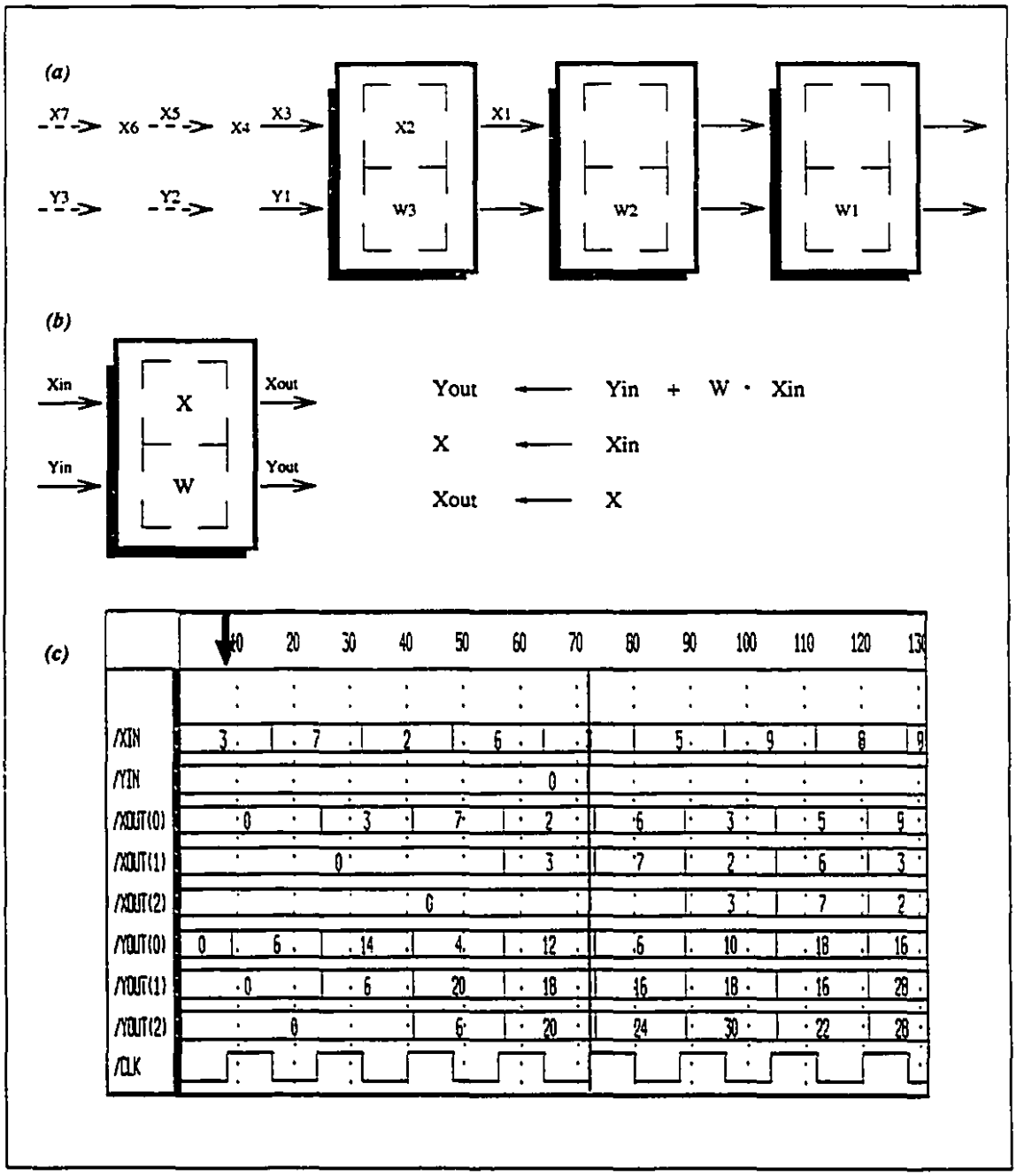

Figure 2.5: (a) Design SYST1: systolic convolution array (a) and cell (b) where weights stay and inputs, X, and partial results, Y, both move systolically in the same direction but at different speeds. A VHDL simulation (c) on the Vantage simulator verifies that this architecture yields valid results after $2 \times (\text{ARRAYLENGTH} - 1) + 1$ clock edges.

To end, we should note that systolic architectures are also advantageous in terms of higher level concerns such as scalability, software overhead, and usability. A unique characteristic of the systolic approach is that as the number of cells expands, the system cost and performance increases proportionally, provided that the size of the underlying problem is sufficiently large. For example, a systolic convolution array can use an arbitrarily large number of cells cost-effectively, if the kernel size is large. This is in contrast to other parallel architectures which are seldom cost-effective for more than a small number of processors. Furthermore, software overhead associated with operations such as address indexing are totally eliminated in systolic systems. This advantage alone can mean a substantial performance improvement over conventional general-purpose computers. Lastly, from a user's point of view, a systolic system such as the one we will present in the next chapter is easy to use –one simply pumps in the input data and then receives the results "on-the-fly".

Chapter 3 A Systolic Solution

In Chapter 2 the requirement for a systolic approach is arrived at largely in answer to the problem of parallelizing convolution. In essence, the research performed in Chapters 1 and 2 forms the "bedrock" of the requirements definition of a convolution processing system. These chapters give an answer to the question "Why?" which is often the first question asked in any system design review. Despite the longevity of the convolution project [Boudreault and Malowany, 1986] [Haule, 1990] [Panisset *et al.*, 1990] [Côté, 1990] [Larochelle, 1991] [Drolet, 1992], the purpose or driving force behind the project has never been covered at any great detail. Hence the attempt was made to address the highest level of abstraction of a convolution system design as graphed in Appendix E. What would naturally follow as a topic for this chapter is the system specification and the implementation of one systolic solution [Malowany *et al.*, 1990] [Malowany *et al.*, 1991] for the convolution problem. This will serve to pave the way for discussion on the results of the laboratory work required to fabricate and test a VLSI systolic cell that is to constitute the core of the systolic solution.

3.1 System Architecture View

The core of the convolution processing system is the systolic array of processing elements shown in Fig. 3.1. For maximum performance to be obtained each processing element should be a high-speed custom VLSI chip which performs the basic multiply and accumulate convolution operations in double precision IEEE format. Configured in 9 rows of 9 custom chips, the array allows a 9×9 convolution kernel to be applied in a single pass. The array can also be configured for one dimensional data, in which case an FIR filter with 81 coefficients is implemented.

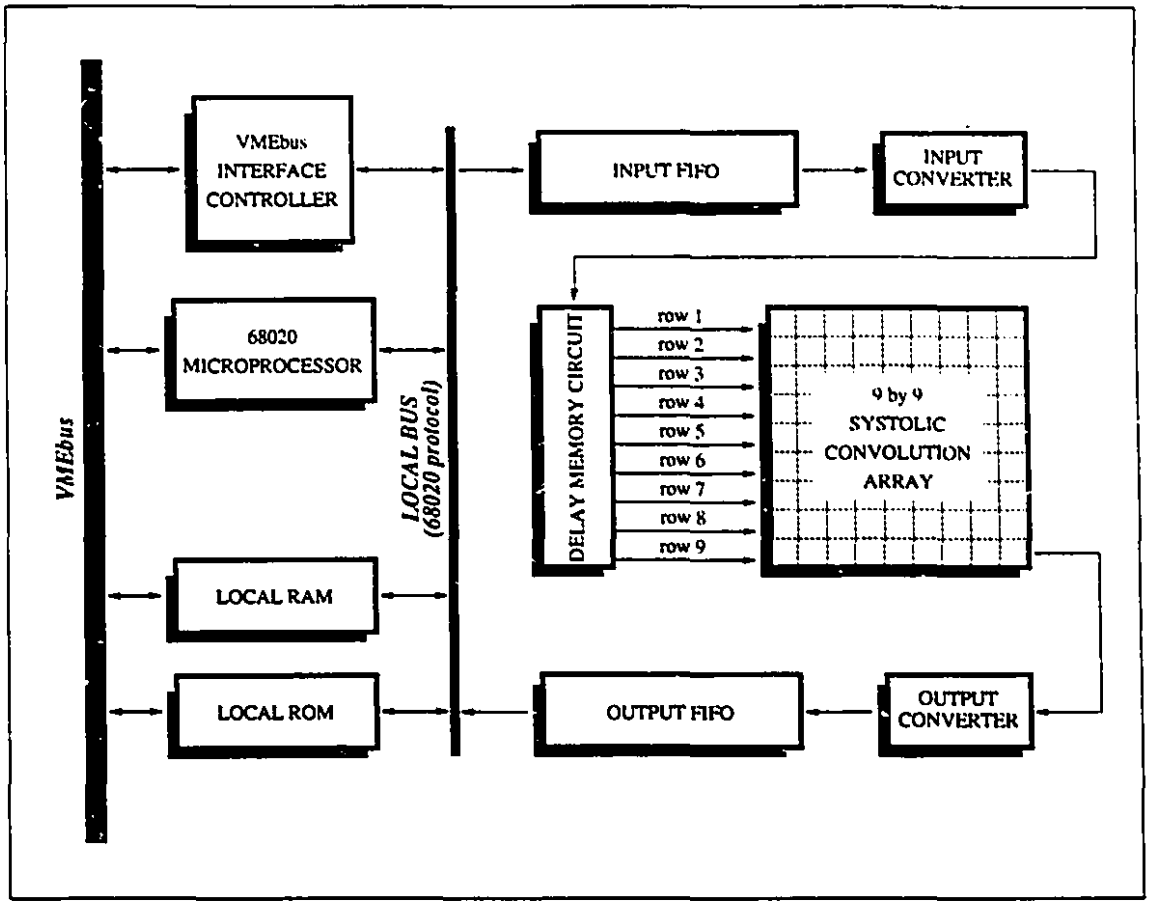

Figure 3.1: Architecture of the systolic convolution processing system.

Clearly, the dedicated systolic processing cells need much additional support circuitry that will supply the data sequence to the rows of the array in a timely fashion and that will store it after processing. Details of the board-level work for the convolution project can be found in [Panisset *et al.*, 1990] [Drolet, 1992].

3.1.1 The Sensor Computing Environment

The convolution processing system is to be implemented as an "intelligent" peripheral that can be easily integrated in the local image-processing environment which is referred to as the Sensor Computing Environment [McRCIM, 1990]. The Sensor Computing Environment is a multiprocessor VMEbus based system which incor-

porates a number of single-board computers and peripheral boards such as a laser range-finder and a variable camera. They run under VxWorks (WindRiver Software), a real-time "flavour" of the UNIX operating system.

3.1.2 Interfacing Requirements

To meet the VMEbus interfacing requirements, the convolution processing system includes a DMA (Direct Memory Access) engine built from an embedded Motorolla 68020 microprocessor and a VTC VIC-068 VMEbus interface controller. The DMA engine is responsible for fetching the image data from the host computer memory and for writing back the convolved image.

3.1.3 Data Stream Manipulation

The source image is read by the DMA engine in 4Kbyte bursts and transferred into the source FIFO (first-in first-out) queue [Botzas and Masson, 1990] where it is handed out in piecemeal fashion to the input converter.

The input converter subsequently takes the data from the FIFO and optionally converts it into the double-precision floating-point format suitable for the systolic array processing elements. Input conversion is required since image data often originates in integer format. For instance, frame-grabbers typically generate 8-bit integer data and devices such as the laser range-finder used in the sensor computing environment produce 16 bit integer values. Moreover, the input converter can also pass along data already in floating-point format -data which may have originated from the results of a previous convolution routine.

Next, a delay memory circuit (DMC) then takes care of feeding the lines of the image to the convolution array in the proper sequence. Each line is sent to the array 9 times, once for each row in the array. In addition, the DMC handles the border

effects by extending the source image with a border of zero-valued samples. It may also be used to up-sample the data, priming it for interpolation.

The data coming out of the systolic array is processed by the output converter which maps the floating-point numbers back into integer format if required. This proves to be useful when the resulting image is destined for display on an RGB monitor which typically accepts 8-bit intensity values. The output converter may be bypassed if magnitudes or numerical precision are unnecessarily sacrificed in the integer conversion, or if the data is to be fed back for another round of floating-point processing.

The output data is then written to the output FIFO after which the DMA engine ensures that convolved samples are written back to host memory whenever the FIFO is half-full. In other words, when there is a sufficient amount of processed data available, it is transferred back to the host computer which initiated the convolution operation.

Table 3.1 summarizes some features of the proposed systolic convolution processing system.

Feature	Description
Architecture	Systolic
Signal type	1-D or 2-D
Number of processors	81
Kernel configuration	9×9 (2-D) 81×1 (1-D)
Arithmetic	Double precision floating-point (IEEE 754)
Interpolation	Up-sampling (2× or 4×)
DMA engine	MC68020 with VTC VIC-068
Estimated performance	126 MFLOPS with 12.5 MHz clock

Table 3.1: Features of the convolution system.

3.2 Systolic Convolution Array

Fig. 3.2 illustrates how the systolic processing elements or cells are to be configured to perform the discrete convolution of image data. For simplicity, a 3×3 array is shown, however the reader can well extrapolate the information for the proposed 9×9 array size. Firstly, each systolic cell in the two-dimensional array is loaded with coefficients C_1 through C_9 that correspond to entries in the convolution kernel matrix. For optimal performance the array size should match the dimensions of the kernel. Furthermore, the first (top left) cell in the array is supplied with a zero valued Y_{in} representing an initialized partial convolution sum. The remaining task is to feed each row of the array with pixel intensity values that come from consecutive lines in the image. Input pixel data then flows through the X_{in} and X_{out} ports of each cell and is multiplied with the local coefficient and added to the incoming partial convolution sum from the Y_{in} ports. The partial convolution results thus flow out from the Y_{out} ports until each cell has made its contribution and a complete weighted sum of all pixels involved is produced at the last (bottom right) cell in the array. It is important to note that inputs and results flow at different speeds and that each input pixel is combined with each partial result at some point in the array so that convolution results are produced at every pipeline cycle after the initial latency. If the delay memory circuit does its job, the rows in the array are constantly fed with serial image data and the effect is like sliding a convolution operator over the breadth of the image –although it would appear more like the image is being squeezed through the convolution operator. Lastly, the array can be configured in a linear scheme which implies that the multiplexers at the left array boundary are simply made to select a one-dimensional data stream rather than the consecutive-line data from a two-dimensional image. This systolic convolution array functionality will be revisited when data flow validation is done in the following chapter.

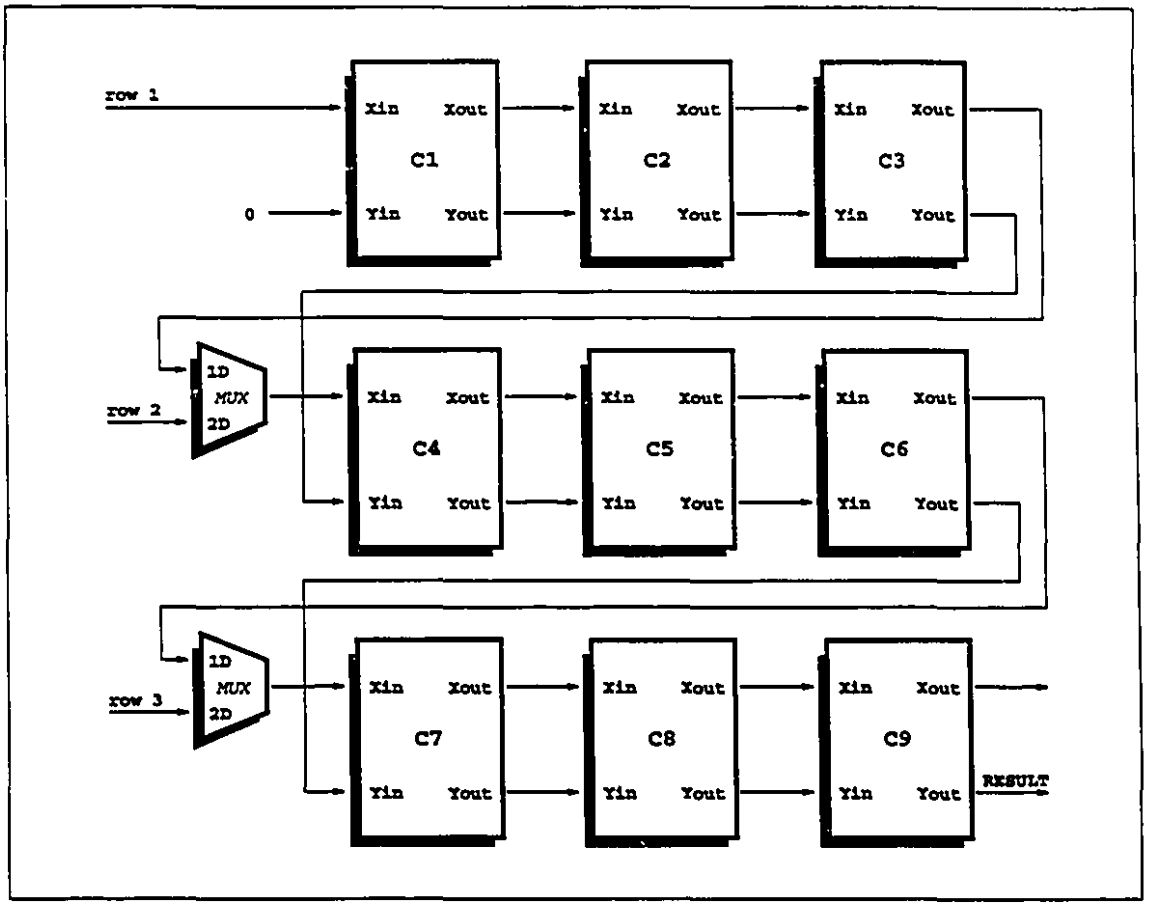

Figure 3.2: Systolic convolution array.

3.3 Systolic Cell

Fig. 3.3 and Fig. 3.4 represent the systolic cell architecture and associated timing of the individual processing stages. The systolic cell is designed [Côté, 1990] [Larochelle, 1991] to multiply a pixel intensity by a given coefficient and add this product to a partial sum. The architecture is organized into three stages which perform the multiplication, addition, and normalization operations in a pipelined fashion. Each operation requires exactly 16 clock cycles (labeled clock 0 through clock 15) for completion. The incoming pixel intensity is multiplied by the coefficient value in Stage 1 and the result is transmitted to Stage 2 at clock cycle 15. While Stage 1 begins its operations on the next pixel intensity, Stage 2 adds the previous result from Stage 1 to the incoming partial sum in the following 16 clock cycles. Fi-

nally, in Stage 3 the floating-point sum from Stage 2 is normalized during the subsequent 16 clock cycles. All internal stage data transfers are 64-bit parallel transfers whereas input
output communication to and from the cell is done on 4-bit (nibble) serial basis. The key design features of the individual components of the systolic cell are presented next.

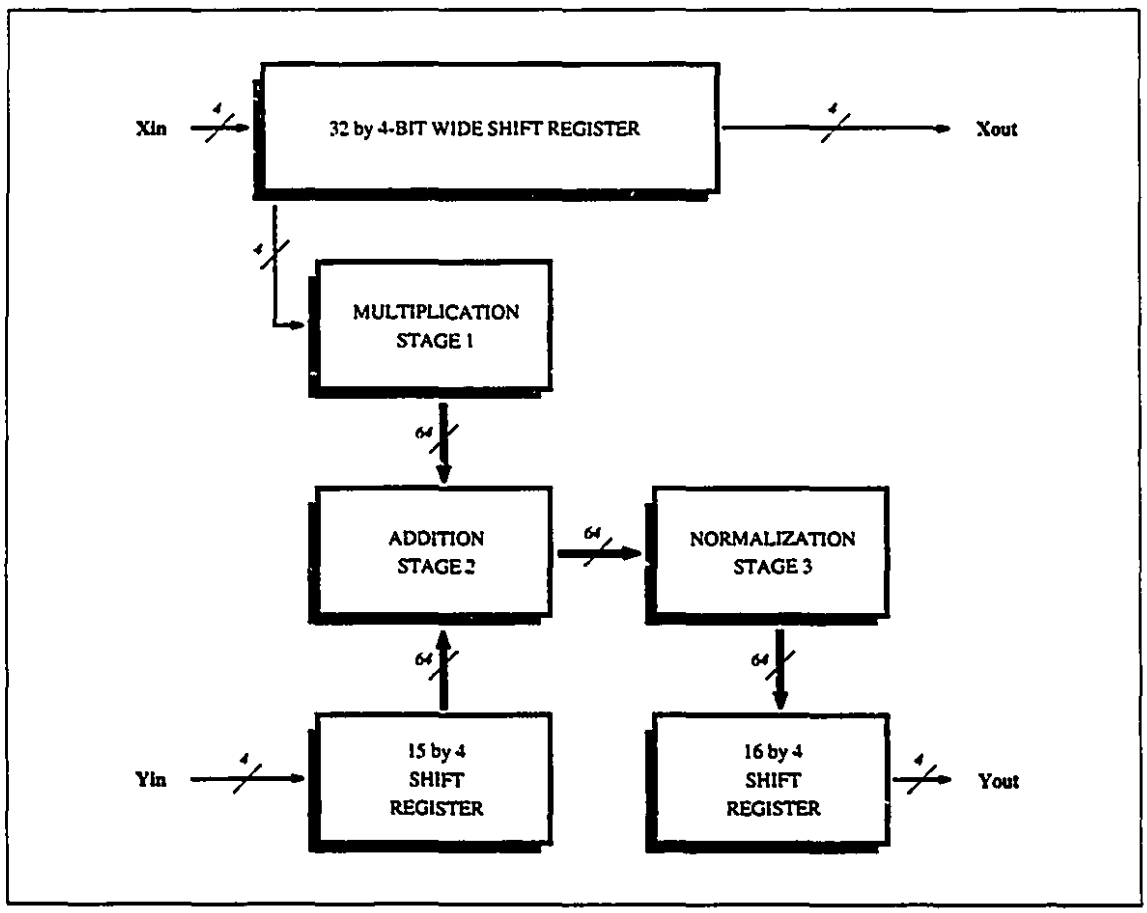

Figure 3.3: Systolic cell architecture.

3.3.1 X Input Register

The X input register is a 32 by 4-bit wide shift register which delays the incoming pixel intensities by 32 clocks cycles before transmitting them to the next systolic cell.

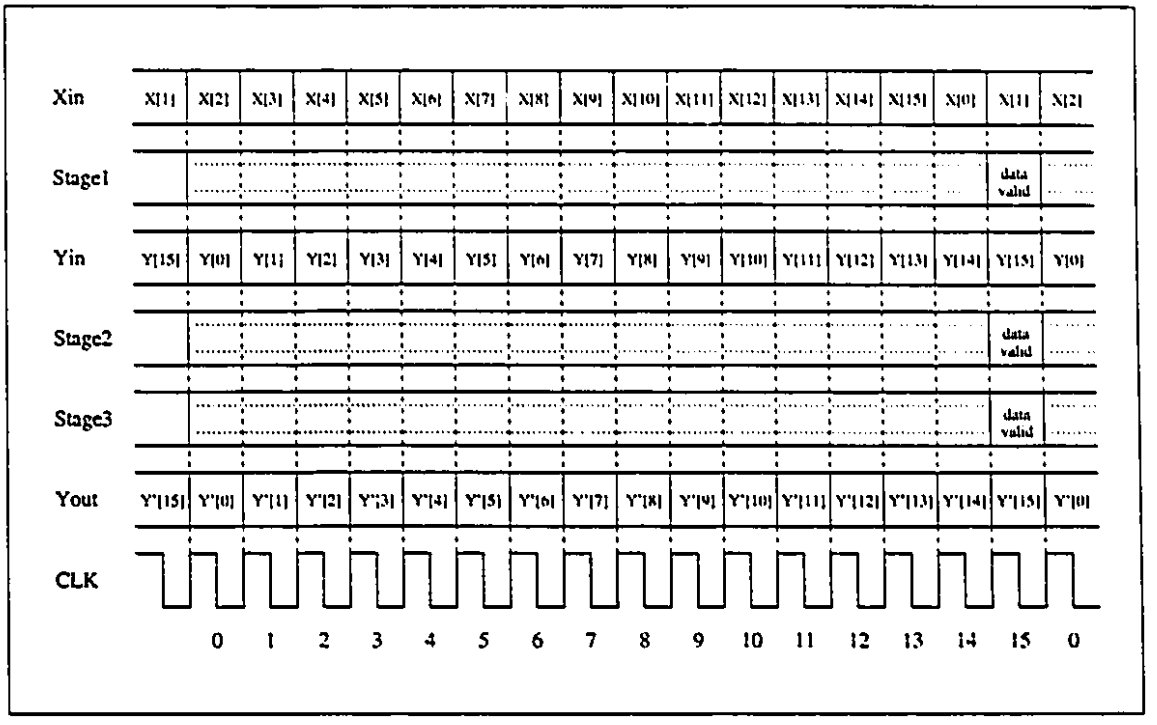

Figure 3.4: Timing diagram for the systolic cell. The X and Y data streams are merged to form the output stream Y', and each stream is shown indexed with respect to nibble. Each stage yields valid data on cycle 15 whereupon all transfers occur.

3.3.2 Multiplication Stage 1

Coefficient Register

The coefficient values must be loaded before any convolution processing can occur. Therefore, a systolic cell has a mode in which it can bit-wise load a double-precision floating-point coefficient value into a 64-bit shift register called the coefficient register. By connecting the C_{out} of each cell to the C_{in} of an adjacent cell in the array, the coefficient registers can be chained together forming a single shift register from which all coefficients in the array can be clocked in serially.

Mantissa Multiplication

As is customarily taught in elementary schools, long-multiplication can be facilitated by considering the multiplier in smaller chunks (or digits) which individually are easier to manipulate since the pupil (or machine) can refer to a short pre-memorized multiplication table. In the stage 1 mantissa multiplication of the coefficient with the input X_{in} the scenario is much the same. The X_{in} binary input is considered in 4-bit (nibble) chunks. Because the coefficient is constant during the entire convolution process, it is possible to compute in advance a multiplication look-up table that stores the multiplication results of the coefficient multiplied with all possible numbers represented by a 4-bit binary number. There are 16 such multiplication results that are computed once at initialization of the chip. During run-time, the incoming X_{in} nibble indexes one of these multiplication results via a multiplexer. All that remains then is the accumulation of the left-shifted multiplication results using a carry-look-ahead adder. One multiplication result is accumulated per cycle to a register called the product register. The multiplier X_{in} is cycled in nibble-by-nibble and the overall multiplication result is only valid once all 13 nibbles of the incoming 52 bit mantissa have been shifted in and processed.

Exponent Addition

The last three nibbles in the 64-bit floating-point X_{in} input contain the the exponent and the sign-bit. During the last (16th) cycle of the clock, the exponent circuitry reaches into the X_{in} shift register and adds the exponent bits with those of the stored coefficient exponent. As well, the resulting sign is generated.

3.3.3 Addition Stage 2

The addition of the incoming partial sum Y_{in} and the product from Stage 1 is performed in Stage 2. Stage 2 receives both of its operands at the rising edge of the first clock (clock 0). The first 60 bits of the partial sum Y_{in} input have been loaded in a 15×4 shift register during the previous 15 clock cycles. The remaining 4 bits are already present at the Y input pins.

Since this numerical processing is being conducted in floating point arithmetic the operands must be aligned before addition can take place. Floating point addition involves determining which operand is bigger, aligning the smaller number to the bigger one and then summing the two mantissas. The exponent value of the bigger number is unchanged and will be the exponent of the sum. The alignment takes place by shifting right the mantissa value of the smaller number by the number of bits corresponding to the difference between the two exponents. If the difference between the two exponents exceeds 54, the mantissa of the smaller number will be completely shifted out and the bigger number will be completely unaffected by the addition.

3.3.4 Normalization Stage 3

Stage 3 is responsible for normalizing the results from Stage 2. A floating-point number is normalized when the left-most 1 in the mantissa is exactly to the left of the binary point. The mantissa is therefore shifted left until the latter condition is met and the exponent is decremented by the same value since each shift left operation is like multiplying by two and therefore the whole number must be divided by two to retain its numerical correctness which means subtracting 1 from the exponent.

3.3.5 Partial Sum Transmission Unit

The 16x4 shift register is responsible for the transmission of the partial sum to the next cell. The normalized number from Stage 3 is loaded into the register on the rising edge of Pulse 0. It is shifted out four bits at a time during the following 16 clock cycles. Note that this nibble-wise serialization of I/O data greatly reduces the amount of interconnecting pins and wires between cells.

3.4 Summary

The problem of parallelizing the two-dimensional convolution operator requires a systolic solution. Hence, a board-level solution is presented that is to be integrated as a peripheral in the Sensor Computing Environment. The core of this systolic system architecture employs a 9×9 systolic array of processing cells. Each systolic cell is implemented on a VLSI chip and is organized into three stages which perform the double precision floating-point multiplication, addition, and normalization operations in a pipelined fashion.

Chapter 4 Fabricating and Testing the Systolic Convolution Cell

Although many design issues for the systolic convolution cell have been covered thus far, the cell design and its VLSI implementation are certainly not new and have been addressed before [Côté, 1990] [Larochelle, 1991]. What this thesis is exclusively responsible for, however, is bringing about a manufactured and tested systolic chip that could be used in high performance image processing systems. This chapter directs its attention on the author's practical work in achieving these goals. In terms of the VLSI design life cycle activity as charted in Appendix E, the author was chiefly responsible for all activities following and partially including "layout/unit simulation".

4.1 Fabrication Process

Preparing the systolic convolution cell for fabrication requires an intimate familiarity not only with the design but with the CMOS process used, with the rules that constrain the process mask and packaging features, with the design and validation tools that are required, and with the design submission process.

4.1.1 CMOS3 DLM Processing Steps

Prototype integrated circuit fabrication was available through the Canadian Microelectronics Corporation (CMC) using a process called CMOS3 DLM – a Northern Telecom Electronics 3-micron single polysilicon, double metal P-well CMOS process. Because of the 3-micron minimum feature size, the supply voltages of the finished devices are limited to 5 volts. The CMOS3 DLM process is actually quite a

unique and elaborate process and it needed to be thoroughly understood prior to the commencement of any layout related work.

CMOS3 DLM has 13 processing steps each with its corresponding mask listed below:

- P-well
- N-well
- Device well
- P-guard
- Capacitor P-doping
- Polysilicon
- N+
- P+
- Contact
- Metal1
- Via
- Metal2
- Passivation

Once P-well and N-well regions are defined, the device well processing step defines the regions which will become drains, sources, channel regions, diffusion interconnects, and capacitors if present.

The next photolithographic step uses the P-guard mask to accomplish a P-implant into P-well regions which eventually defines the threshold voltage of the P-channel transistors.

After defining the bottom plates of capacitors (there are none in the systolic cell) with heavily doped P+ using the capacitor mask, the polysilicon (SiO2) mask is

used to define the gate regions, poly interconnections, and the upper plates of capacitors if present.

The subsequent N+ and P+ processing steps are used to further define source and drain regions and contact regions.

Once contact regions have been etched out, the first layer of aluminum is deposited over the entire surface and the metal1 mask is used in a photolithographic operation to define the metallization lines.

A second layer of SiO2 is deposited over the entire surface and the "via" mask is used to etch out via regions which are used to connect metal1 to metal2.

Once the second layer of aluminum has been evaporated over the wafer, the metal2 mask defines the metalization lines whose purposes include bonding and probe pad placement.

Lastly a passivation layer is deposited over the entire surface and the passivation mask is used to etch out regions over the metal bonding pads and probe pads.

4.1.2 CMOS3 DLM Design Rules

To make the layout representation of the systolic cell design suitable for fabrication, more than two weeks were spent entirely devoted to fixing masks that violated the Northern Telecom Electronics 3-micron CMOS process design rules. The design rules that were breached included those that specified minimum feature size for each layer (mask), those rules that specified minimum spacing between features of the same layer, the rules that constrained the geometry of a layer that either completely surrounded or partially overlapped the geometry of another layer, and the rules for the 68-pin PGA package. All these design rules were specified using a 5-micron scale (design scale microns) for the minimum feature size. The designs are scaled down at Northern Telecom to 60% of the specified dimensions prior to the

creation of the pattern generator (PG) tapes.

4.1.3 Packaging and Bonding

The final steps in manufacturing chips are packaging and bonding. Once the the systolic cell is fit into the cavity of a package, bonding is achieved by connecting the fingers surrounding the package cavity to a standard pad frame supplied by the CMC or it may be custom designed. In the case of the convolution cell chip, the huge transistor-count (nearly 50 thousand) necessitated a special request for an "oversized" pad frame that exceeded the dimensions of the standard size-A (full-size) pad frame.

Also requested was a 68-pin grid array (PGA) package with a removable lid that permits physical access to the chip and bonding wires, to allow for probing or other diagnostic procedures. Once the chips returned from fabrication, all the lids were indeed removed for a microscopic inspection of the silicon. No visible defects were observed.

In Fig. 4.1 the 68-pin PGA package has been reproduced perfectly to scale. The top view is shown with the lid removed while the bottom view portrays the pin numbering scheme. The cavity size of the PGA is approximately 1 square centimeter or 410 square mils where 1 mil is the popular unit of VLSI dimensions representing 1 millionth of an inch.

There are 17 metal fingers on each side of the square for bonding wire connections. These bonding wires connect to bonding pads on the pad frame, and like the rules which govern mask geometry features there are also rules for pad placement. The sharper the angle of the bonding wire (for example, greater than 45 degrees), the greater the strain on the wire and the more unreliable the connection. The longer the bonding wire, the greater the risk that the wire will droop and possibly cause a short circuit to other wires. The risk is compounded if all the bonding

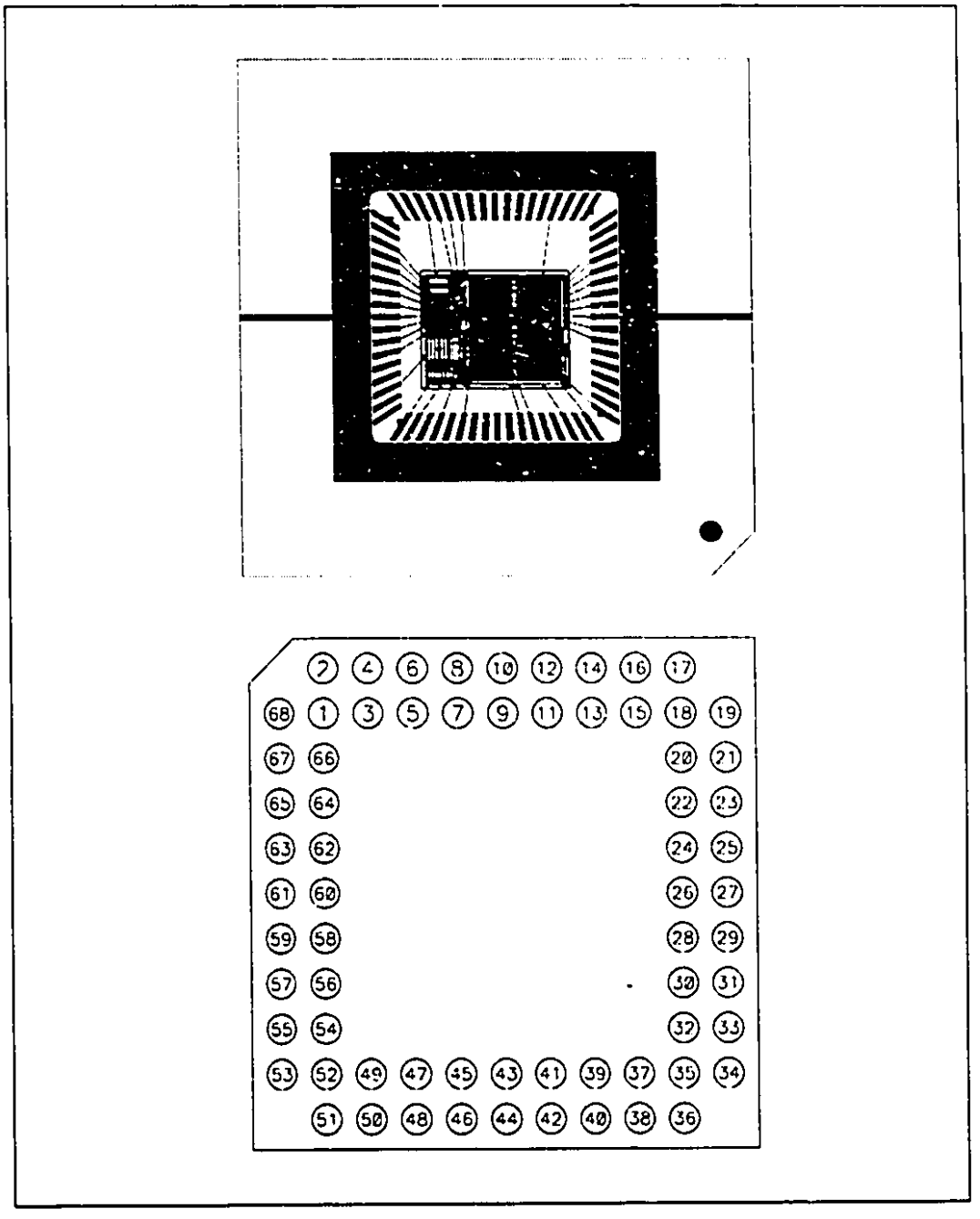

Figure 4.1: Top and bottom view of package.

pads are placed close together on one side of the design. Moreover, the pads should be collinear in placement to prevent bonding wire paths from crossing. The insight gained with respect to pad placement can be summarized as follows: place each pad so that the length of the bonding wire to the corresponding connection finger on the package is minimized and the separation between adjacent bonding wires is maximized.

4.1.4 CADENCE VLSI Design Software

A large part of this thesis involved learning to use CADENCE's VLSI Design software tools. The EDGE Design Framework™ provides a completely integrated set of VLSI design and verification tools in a menu-driven, graphical environment. A few basic concepts with regard to the VLSI design tools used will lend insight as to how a project of this magnitude is realized.

4.1.5 Design Files and File Hierarchy

The overall design of a chip can be looked upon as a hierarchy of blocks, with each block representing some functional unit of the complete chip. The block size is dependent upon the design approach taken for the chip and could be anything from a single transistor to a complete microprocessor. Each block may incorporate symbolic instances of other blocks, in which case a hierarchy of blocks will be formed.

The blocks themselves may have many different representations depending on the role of the block in the overall design. There are representations for "schematic", "symbolic", "layout", "extracted", "silos", "spice", and "autoLayout", to name just a few. The representations are further divided into revisions. The "current" revision contains the most recently saved revision. There can be many "backup" revisions as well.

Within the UNIX file system, all the blocks used in a design, except those from some common libraries, are usually created in a common directory. The blocks are simply subdirectory entries in this common directory. The representations are sub-directory entries in the block directories. The revisions are the actual binary design files, which are stored in the representation sub-directories.

4.1.6 Edge Database Format

In the EDGE environment, all design files, independently of what type of representation they are intended for, share a common database format. The database format is flexible enough to be able to describe representations which contain only geometry, such as "layout", representations which contain only electrical connectivity, such as "extracted", representations which contain only textual information, such as "silos" and "spice", and representations which contain all three, such as "schematic" and "symbol". The database format also provides the means to describe the instance hierarchy and attach properties to any of the database objects.

Geometry, instances, pins terminals, nets, labels and properties are all objects in the EDGE database format. Geometrical objects are stored as sets of coordinate points which describe boxes, polygons or other shapes drawn on particular layers. Instances are pointers to symbol representations of other blocks. Each instance is assigned a unique name within the schematic to distinguish it from other instances of the same block. Pins and terminals are used to describe input/output terminals of the schematic and of each instance in the schematic. Nets are used to describe the connectivity between all the terminals in the block. Labels are used to put text strings in the design and properties are to describe characteristics of the entire design file or the specific objects in the design file.

4.1.7 Schematic Representations

The schematic representations of a block is one in which the electrical functionality of the block is described using instances of symbols, pins for the input and output terminals, and wires that represent the interconnection nets. Fig. 4.2 shows the top-level-block schematic representation which also illustrates the floor plan of the systolic cell. Further, in Appendix F an effort was made using the SKILL programming language native to CADENCE tools to capture the complete hierarchy of schematic representations. The schematics of some of the key blocks in this hierarchy are also included.

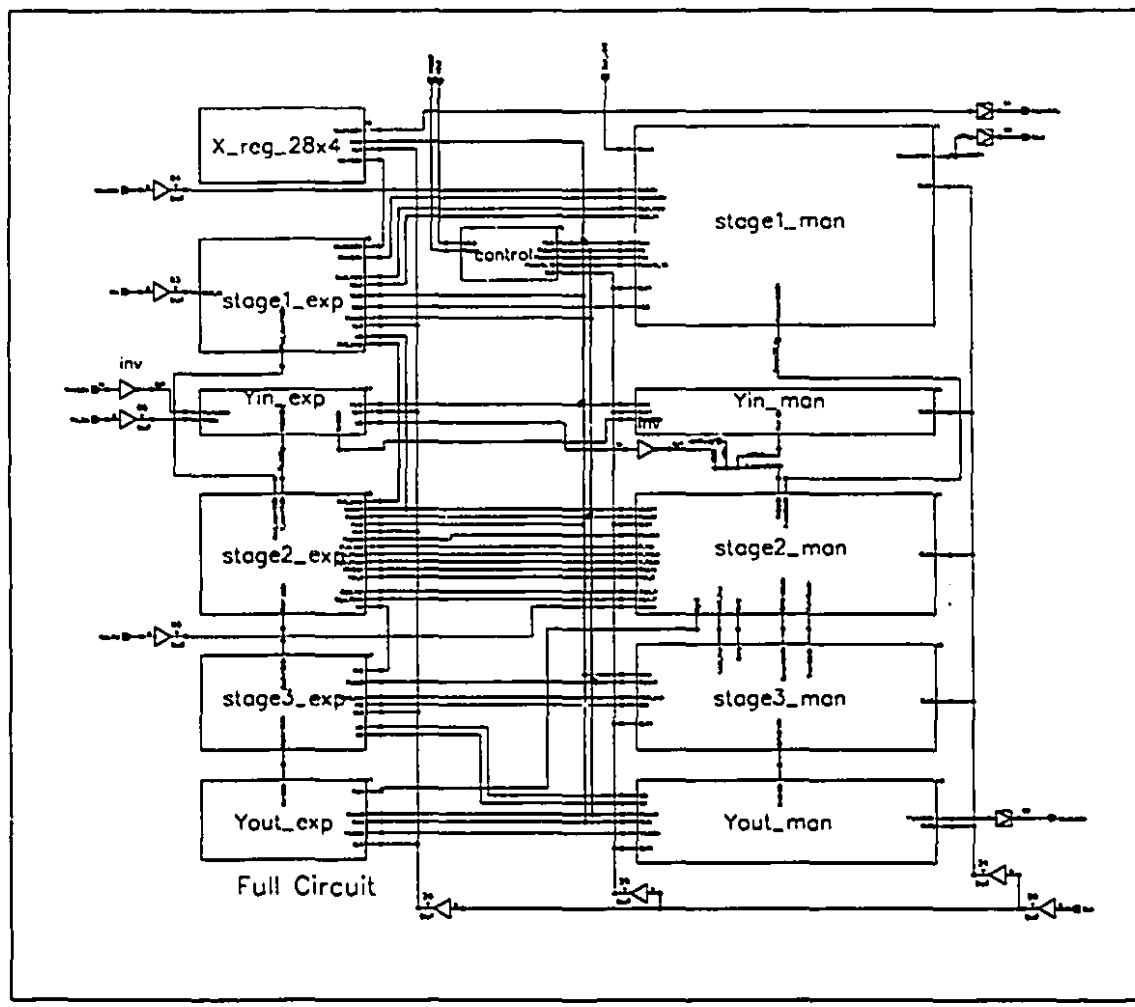

Figure 4.2: Top-level-block schematic representation illustrating the floor plan of the systolic cell.

4.1.8 Custom Layout

The layout representation of a block is one in which the masks used for the fabrication process are described geometrically using rectangles and polygons on different process related layers. The layers (masks) used depend on the technology in which the block is being implemented. In our case the layers file was setup to contain all the layers necessary for the CMOS3 DLM technology.

As in the case of hierarchical schematics, hierarchical layouts are used to reduce the complexity of large layouts. Custom layout is a tedious and involved task, hence one tries to create a small core set of essential layout blocks that can be instantiated together in more complex arrangements to build the major blocks of the layout. Layout revision files tend to be enormous, due to the amount of geometrical information being stored. Since instances are simply pointers to the master block (not copies) significant memory savings can be achieved and the computer can work with the data far more efficiently. In addition, if a change is made within a layout block, the change is reflected immediately in all instances of that block.

The layout for the systolic cell was totally "custom" which does mean that the layout representation of each and every block was created manually. A sample cell layout of a flip-flop is shown in Fig. 4.3. The dark patches in this figure highlight the placement of P-type (pin) metal1 and metal2 layers which are used to impress or probe signals on desired nodes during simulation. Indeed since the designers of some of the lower level blocks had neglected to save the corresponding simulation files, many such lower level layout representations required revalidation with the HSPICE simulator before the chip could be sent out for fabrication. Fortunately, only a few small clock buffers had to be "squeezed" into the top-level layout to ensure sufficient clock drive. The overall layout representation both in its hierarchical form and its "exploded" form appears in Fig. 4.4 and in Appendix F.

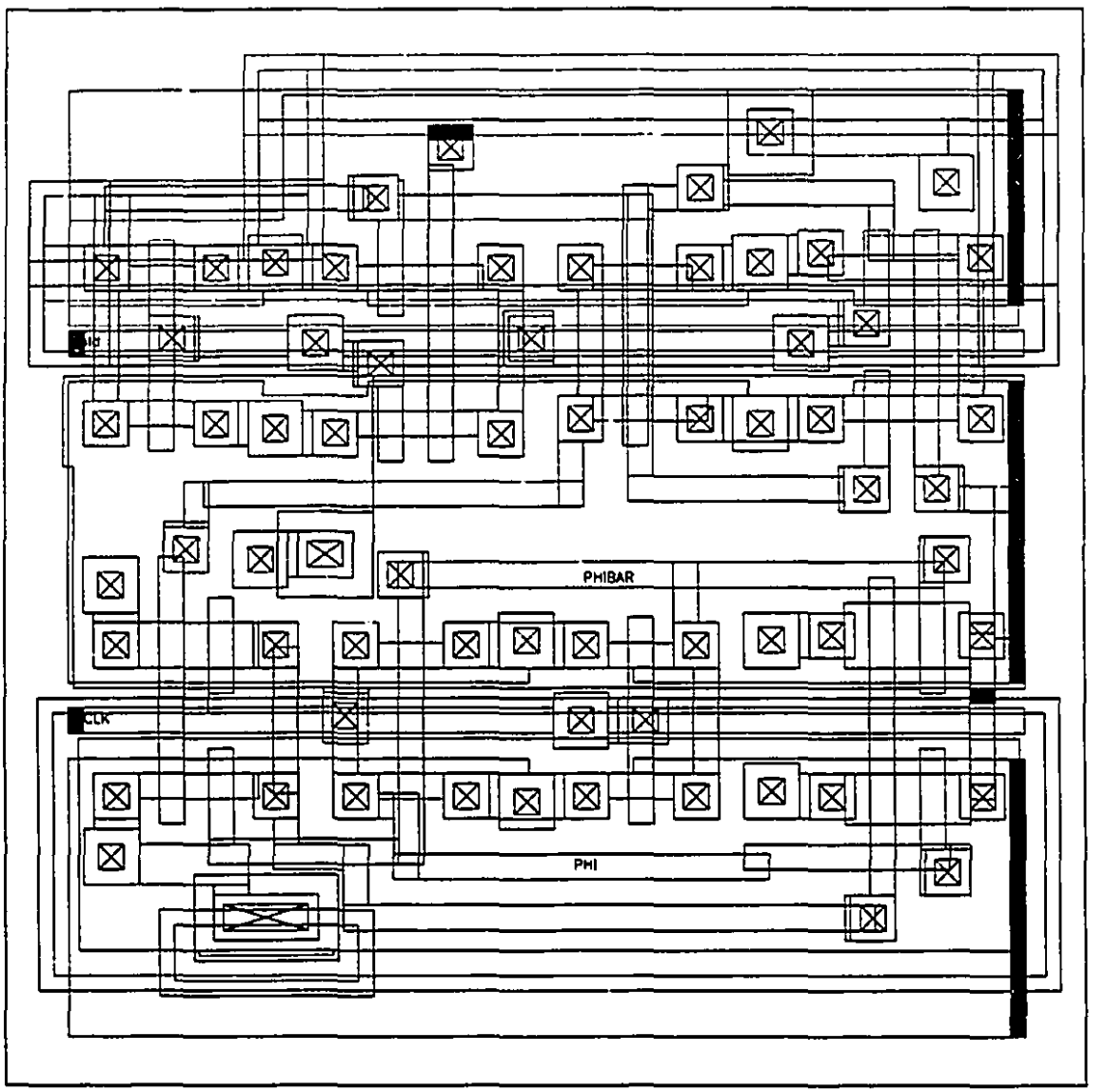

Figure 4.3: Sample cell layout of a flip-flop. The dark patches highlight the placement of P-type (pin) metal1 and metal2 which is used to impress or probe signals on desired nodes during simulation.

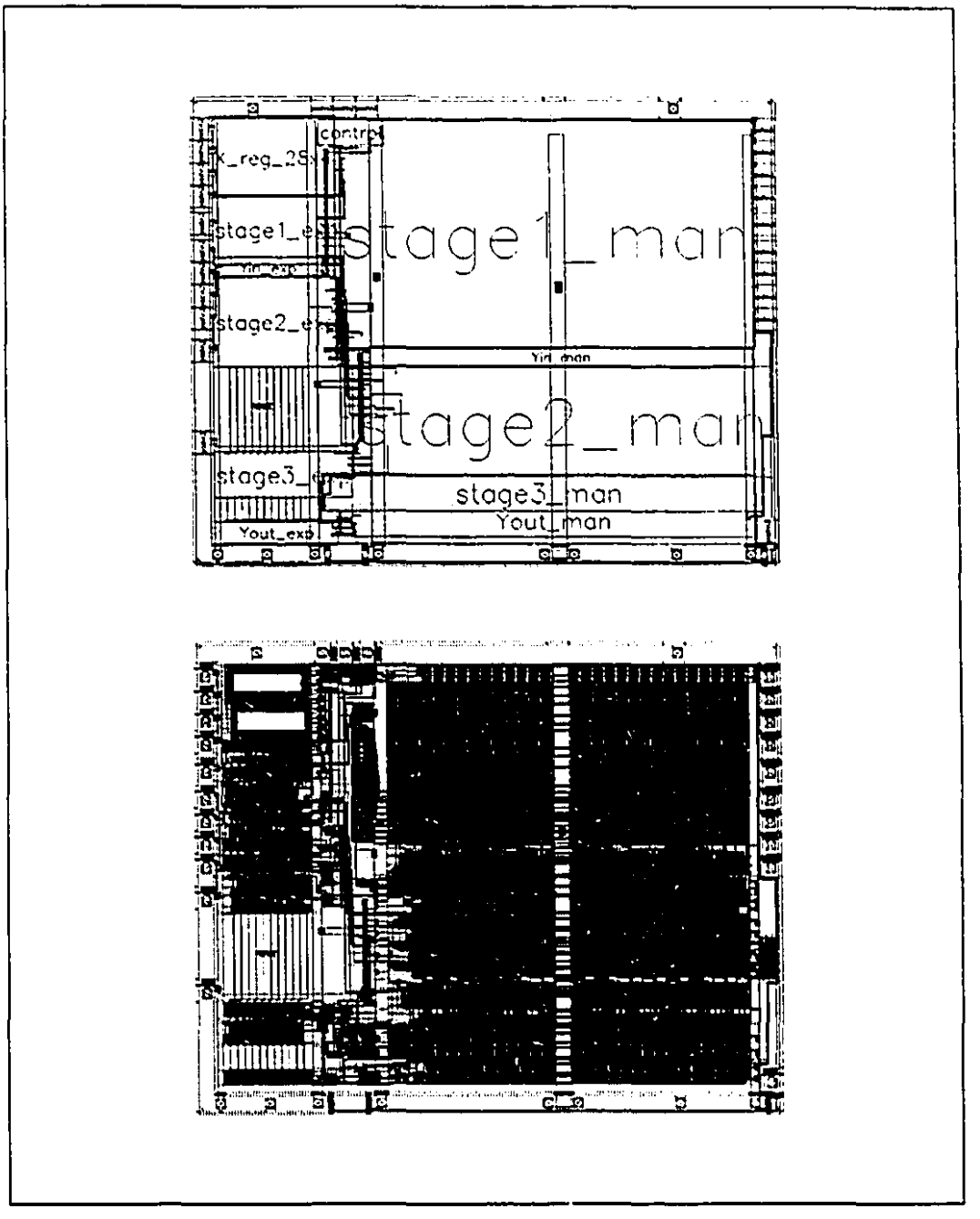

Figure 4.4: Layout representation.

4.1.9 Layout Submission

Once a valid layout was produced that passed all design and packaging rules, it was a challenge to export the design data from the CADENCE environment to the site of the Canadian Microelectronics Corporation. All design data submitted to the CMC must be in the Caltech Intermediate Form (CIF 2.0). A complete description of CIF syntax is given in [Mead and Conway, 1980]. The CMC checks submitted designs with their own design rule checker called DRACULA™ which utilizes more conservative rules than those used in the CADENCE environment. The increased constraints imposed by the CMC had caused further rule violations, and the design had to be resubmitted 6 times before it was accepted for fabrication. After 6 months, 5 manufactured chips returned from the Northern Telecom Electronics fabrication laboratory. The focus then turned to the anxiously awaited though momentous task of testing the systolic convolution chip.

4.2 Testing

While in real estate the refrain is "Location! Location! Location!", the comparable advice in IC design according to [Weste and Eshraghian, 1993] should be "Testing! Testing! Testing!". Indeed, we will now look upon the testing of the systolic convolution cell under a structured framework in which the key aspects of VLSI testing are also surveyed.

Due to the complexity of the manufacturing process not all die on a wafer correctly operate. Small imperfections in starting material, processing steps, or in photomasking may result in bridged connections or missing features. Hence it is the aim of a test procedure to determine which die are good and should be used in end systems.

Tests may fall into two main categories. The first set of tests verifies that the chip performs its intended function. In our case, the intended function is the basic floating point multiplication, addition, and normalization required of the systolic cell. These initial tests assert that all the components in the chip, acting in concert, achieve the desired function. These tests are usually used early in the design cycle to verify the functionality of the circuit and are called functional tests. They may be lumped into the verification activity.

The second set of tests verifies that every gate and register in the chip functions correctly. These tests are used after the chip is manufactured to verify that the silicon is intact. These are called manufacturing tests. Generally speaking, the nature of design usually leads one to consider function before manufacturing concerns.

4.2.1 Functional Testing

For most systems, functionality tests involve proving that a circuit is functionally equivalent to some specification. That specification might be a verbal description,

a plain-language textual specification, a description in some high-level computer language such as C, FORTRAN, PASCAL, or Lisp or in a hardware-description language such as VHDL, ELLA, or Verilog, or simply a table of inputs and required outputs. Functional equivalence may be carried out at various levels of the design hierarchy. If the description is in a behavioural language, the behaviour at a system level may be verifiable.

For the systolic convolution array, a top-level, behavioural and structural VHDL specification is included in Appendix D. A sample simulation result shown in Fig. 4.5 verifies the intended functionality of a simple three-cell systolic array. By virtue of the row latency in the SYST2 systolic design, accurate convolution sums of the current pixel intensity X_{in} and its two serial predecessors occur seven pipeline-clock edges after the current sample. Note that all weights in this example are all equal to 2, and that the initial partial sum input Y_{in} to the left-most boundary of the array is fixed to zero. The first convolution sum, hence, for the Xin values of 3, 7, and 2 is 24. The convolution sum for the next sequence 7, 2, and 6 is 30, and so on. This kind of functional verification at the system specification level was precursor to basic functional testing of the systolic cell itself. These basic functional tests were conducted on the manufactured chip itself and were used not only to verify basic functionality but to provide valuable input/output specifications of all ports on the chip.

4.2.2 Manufacturing Testing

Whereas functionality tests seek to verify the function of a chip as a whole, manufacturing tests are used to verify that every gate operates as expected. The need to do this arises from a number manufacturing defects that might occur during chip fabrication. Typical defects may include layer-to-layer shorts (ie. metal-to-metal), discontinuous wires, or thin-oxide shorts to the substrate or well. These in turn lead to particular circuit maladies, including nodes shorted to power or ground,

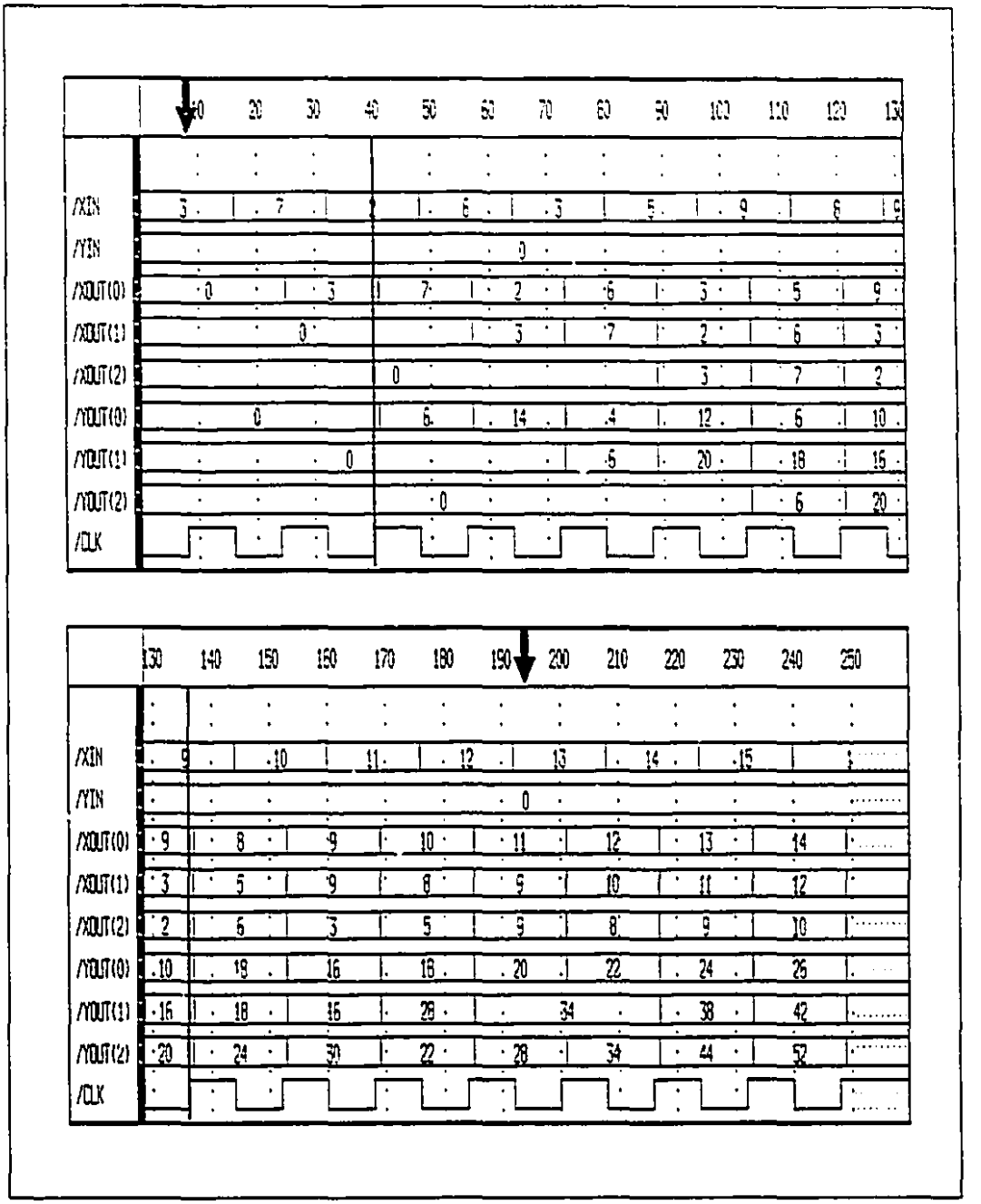

Figure 4.5: VHDL simulation of the SYST2 design on the Vantage simulator demonstrates valid results after $2 \times (\text{ARRAYLENGTH} - 1) + 3$ clock edges.

nodes shorted to each other, floating inputs, or disconnected outputs. In general, manufacturing-test generation assumes that the chip functions correctly, and ways of exercising all gate inputs and of monitoring all gate outputs are required.

4.2.3 Testing Process

Having introduced the concept of functionality testing and manufacturing testing, it should be made apparent that conducting these tests on a chip once it has returned from the fabrication laboratory can be a long and arduous process. However, if time is spent to suitably prepare and automate frequently performed tasks then the tests can be made less overwhelming.

The first concern in the mechanics of testing the systolic convolution cell was in developing a good testing environment. The test environment constituted those devices which were used to stimulate the chip under test as well as those which were used to analyze the resulting responses. Conventional methods were first considered in which the chip would be "bread-boarded" and manually tested with a data generator/data analyzer pair. Yet in light of the lengthy iterative testing process at hand, it was concluded that this approach would prove far too time consuming and laborious.

The availability of "networked" testing equipment opened up the much more attractive possibility of a "fully-automated" testing environment. The testing equipment which still comprised primarily of a data generator and data analyzer were brought under remote-control from a series of centralized test programs that ensured expediency in the test process. Tests could be run and rerun with all but a few seconds delay. Hence, as will be stated shortly, much software development effort was invested in the ways of test environment automation, and this work was as much a part of this thesis as it was a direct contribution to the testing community of the VLSI laboratory of McGill University.

Automated Test Environment

The overall block diagram of the automated test environment is shown in Fig. 4.6. As can be seen, the controlling computer is linked to the test equipment through an HP-IB bus. This bus allows two-way communication between the computer and other devices in the form of commands and data transport. In this way, all data gathered by the test equipment is accessible by the controller and can be processed using a variety of software tools in a UNIX environment. This importing of "measured" results into the computer workstation also greatly facilitated the testing process.

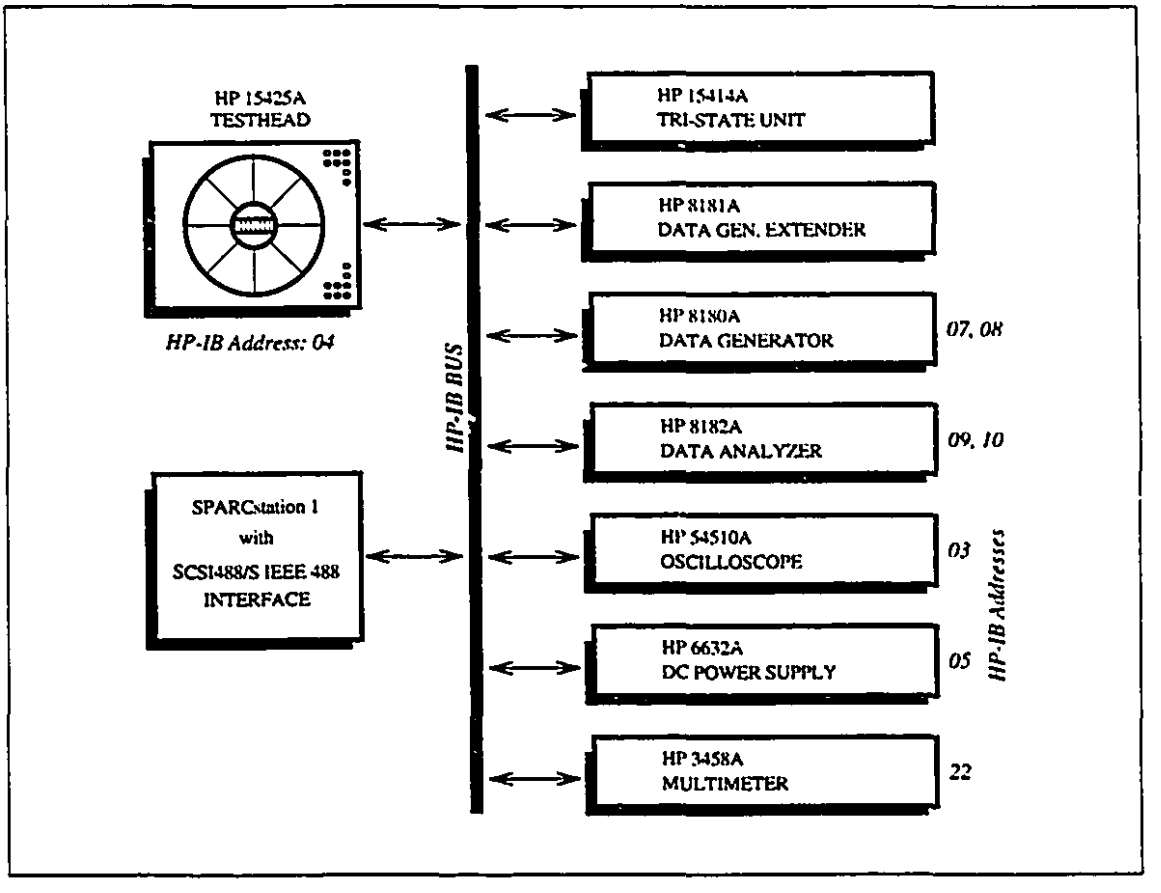

Figure 4.6: The test environment was automated via a SPARCstation 1 and an IEEE488 interface to the HP-IB bus.

Testhead Layout

The connection between the test equipment and the device under test (DUT) was accomplished with a Hewlett-Packard 15425A testhead. The outer ring of the testhead can accommodate up to 84 connections of which 27 are double (bidirectional) channels. These channels are connected to 84 "pogo" pins at the inner ring through optional load resistors and relays. Finally, the pogo pins are connected to a 40-pin socket which receives the DUT.

A list of channel assignments for the testhead appears in the section entitled "Testhead Configuration" under Appendix G. This table maps the channels that connect to the data analyzer and data generator to the pogo pins, to the pin numbers on the testhead socket which receives the test-board, to the corresponding signal names and pin numbers of the actual chip.

In addition to the AC-measurement channels, the HP 15425A has connections for DC power and ground. Excluding the DC measurement ring, there are three separate DC rails labeled DPS1, DPS2, and DPS3, all sharing a common ground. These supplies occupy channels 55, 54, and 53 respectively and can be accessed through "banana" plugs on the top face of the unit. The ground pin appears on channel 56 and is also common to the DC-measurement ring.

Test Process Software Development

The controlling host SPARC workstation depicted in Fig. 4.6 was interfaced with the rest of the HP test equipment via an HP-IB bus. Specifically used were a SCSI IEEE controller by IOtech™ consisting mainly of the SCSI488 IEEE488 interface card and the Driver488 software which provided the core software interface between UNIX and the IEEE interface. Thus given this rudimentary network connectivity and a basic set of communication directives for the HP-IB bus, it was undertaken to develop a complete software base for higher-level utility functions including a

front-end user interface for the test process. To this, a script writing mechanism accompanied by a parser and syntax checker was also created for users who wished to write their own scripts in the native language of the data generator and data analyzer and the other devices on the right side of Fig. 4.6. Sample initialization scripts are included in Appendix G.

Device-Under-Test Board

In order to test the sytolic cell in its square 68-pin package, a device-under-test board was built to fit over the rectangular 40-pin socket of the testhead. The board was first populated and tested with a few standard CMOS parts such as shift-registers and 4-bit adders so that confidence could be gained in the testing process prior to strapping in the real systolic part. Once the test environment itself was tested and tuned to the CMOS technology, the board was rewired and populated solely with the systolic chip and some of the associated capacitors drawn in Fig. 4.7. The purpose of the capacitors were chiefly for output load equivalents and for noise reduction from the power supply. C_L includes the probe and jig capacitance and the $22\mu F$ (electrolytic) and $0.1\mu F$ capacitors filter out low and high frequency noise respectively. It is regrettable that the latter noise reduction capacitors were not used at the outset of the testing process, and it is suspected that at least 4 systolic cell chips may have fallen victim to "latch-up" as a result. Since the latch-up phenomenon is blamed for destroying all but one of five chips manufactured, a brief account follows.

Latch-Up

Latch-up is an undesirable effect that plagues CMOS technologies. The result of this effect is the shorting of the VDD (5v) and Vss (0v ground) lines, usually resulting in chip self-destruction. The source of the latch-up effect [Troutman, 1986] [Estreich and Dutton, 1982] may be explained by parasitic bipolar devices which

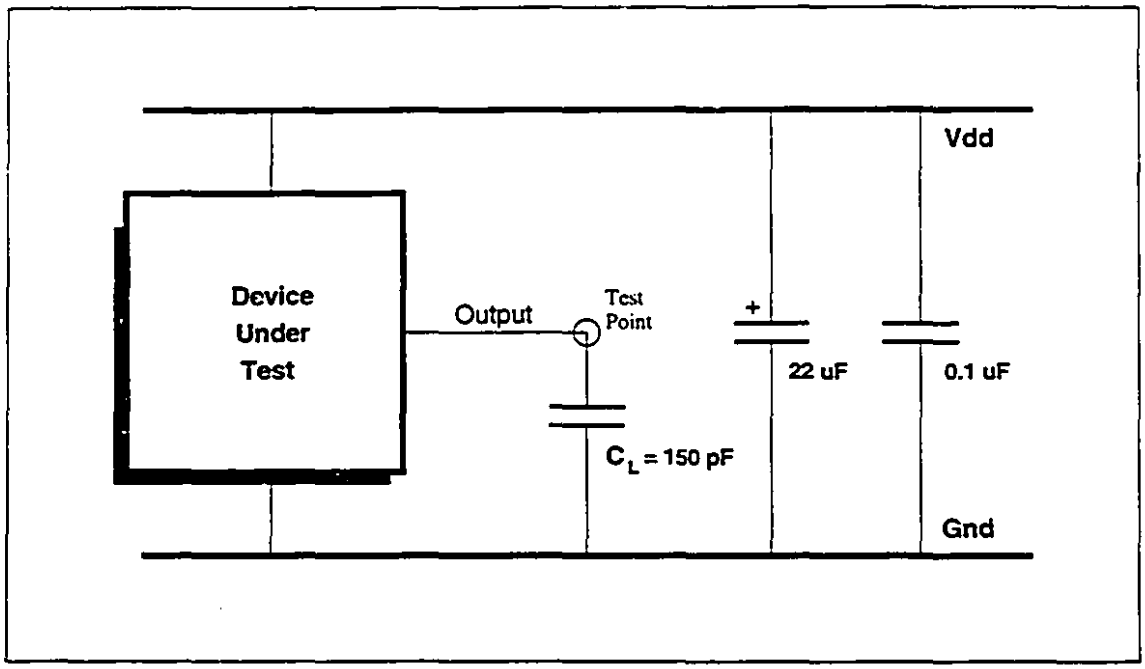

Figure 4.7: A.C. testing load circuit. C_L includes the probe and jig capacitance. The $22\mu F$ (electrolytic) and $0.1\mu F$ capacitors filter out low and high frequency noise respectively from the voltage source.

are an unwanted byproduct of producing pMOS and nMOS transistors. Under the right conditions the latent parasitic circuit becomes active or "snaps" and draws a large current while maintaining a low voltage across its terminals. Latch-up can be triggered by transient currents or voltages that may occur internally to a chip during power-up or externally due to voltages or currents beyond normal operating ranges. Radiation pulses can also cause latch-up; however, this was not considered the likely factor in our case.

Since current has to be injected into the emitters of the parasitic device for latch-up to occur, such a condition is likely to befall the I/O circuits employed on a CMOS chip, where the internal circuit voltages meet the external world and large currents can flow. It is therefore suggested that in future "spins" of the systolic cell that extra precautions be taken with the peripheral circuits. For instance, placing an abundance of substrate contacts over the peripheral transistors serves to short out parasitic devices rendering them harmless. It is also highly advised that the aforementioned noise reduction capacitors not be omitted from the DUT board.

4.2.4 I/O Specification

In addition to providing functional specification verification of a chip, functionality tests may also serve to extract valuable timing information which is part and parcel of an input/output port specification. An I/O specification is indeed required if the chip is to be used as a module in a systolic system. However, before I/O timing is discussed it is essential that each port on the chip be thoroughly described first. Table 4.1 presents a complete description of the 24 pins of the double-precision floating-point systolic convolution cell chip.

DC Characteristics

Once the chip was socketed and powered up, the DC parameters were recorded. Table 4.2 lists some DC characteristics that were measured during the testing process. Digital input voltages ranged from -0.4 volts for a logic zero (LOW) to 6.0 volts for a logic one (HIGH). The latter HIGH voltages may seem excessive for the data generator but an excess 0.5 volts can at least be accounted for by the 10% fluctuation of the available power supply. Furthermore, the maximum power supply current that was drawn by the chip recorded as high as $20.2mA$ which indicates an unusually high power dissipation in the order of $100mW$. Lastly, the 1MHz test clock supplied to the circuit is traced in Fig. 4.8. The duty cycle is 50% and the transitions are driven sharply at roughly $12ns$ a piece. It is important to note that these transition times are further eroded as the clock is routed internally through the chip's clock-buffer tree.

Operating the Systolic Cell

True to the philosophy of systolic designs, the systolic cell chip has simple control mechanisms that mitigate the global synchronization overhead required of large arrays. Once the chip has been set to a known state using the PRESET signal it is al-

Symbol	Pin No.	Type	Name and Function
CLK	1	Input	**Clock:** This line provides the basic timing for the systolic cell. A symmetric clock signal (50% duty cycle) is input and used in a single phase clocking scheme.
Yout0-Yout3	7-10	Output	**Y output data:** The Y output data bus outputs the correctly normalized partial convolution sum. Along with the Yin bus, the Yout bus provides a data path for a systolic array which enables partial results to move systolically from cell to cell.
Cout	11	Output	**Coefficient output:** This is the output of the kernel coefficient register. In a systolic array the coefficient inputs and outputs are chained together so that each cell can be serially loaded via the boundary cell.
Xout0-Xout3	12-15	Output	**X output data:** The X output data bus along with the input data bus provides a data path for a systolic array which enables the inputs to move systolically from cell to cell.
C_LOAD	29	Input	**Coefficient load:** When held HIGH this line enables the kernel coefficient register to shift-right synchronously unless HOLD=1. Coefficient data enters from Cin and exits from Cout. When C_LOAD is LOW, the coefficient bits are held in place. This control signal must not be left floating.
HOLD	30	Input	**Hold:** When this line is set HIGH, every flip-flop in the systolic cell including those in the coefficient register is held in its current state until HOLD is reset to LOW. HOLD effectively "stalls" the systolic cell. HOLD must not be left floating.

Table 4.1: Pin description.

Symbol	Pin No.	Type	Name and Function
PRESET	31	Input	**Preset**: When asserted, the PRESET line presets the state counter in the control circuit to state 14. This provides a time reference for subsequent operation. PRESET must not be left floating.
Xin3-Xin0	37-40	Input	**X input data:** These lines constitute the incoming pixel-data bus. Floating-point data is fed in to the systolic cell nibble by nibble in run mode with HOLD=0 and C_LOAD=0.
Cin	41	Input	**Coefficient input:** This line is the input to the serial bit register that holds the systolic cell's kernel coefficient which is loaded in during coefficient-load mode when HOLD=0 and C_LOAD=1.
Yin3-Yin0	42-45	Input	**Y input data:** This is the incoming partial-sum data-bus. Floating-point nibble-data is entered into the systolic cell in run mode unless gated by Yin_Dis.
Yin_Dis	46	Input	**Yin disable:** When held HIGH this line asserts zeros on the Yin data bus. When held LOW the regular partial-sum inputs are fed in. This line should not be left floating. This function is especially useful at the input boundary of a systolic array.
Abs_Val	49	Input	**Absolute value:** When held HIGH this line asserts a zero on the latch which stores the sign-bit in the normalization stage 3 of the systolic cell. When held LOW the regular sign result from the addition stage 2 is stored. This line should not be left floating. This function is useful for processing pixel data where only the magnitude information is required.

Table 4.1: Pin description (continued).

Symbol	Parameter	Min	Max	Units
V_{IL}	Input LOW voltage	-0.4	+0.6	V
V_{IH}	Input HIGH voltage	4.0	$Vcc + 0.5$	V
V_{OL}	Output LOW voltage	—	0.2	V
V_{OH}	Output HIGH voltage	4.75	—	V
I_{CC}	Power supply current	—	20.2	mA
V_{CL}	Clock input LOW voltage	-1.9	0.2	V
V_{CH}	Clock input HIGH voltage	4.7	6.4	V

Table 4.2: D.C. characteristics at ambient room temperature and $Vcc = 5V \pm 10\%$.

lowed to operate in only one of three modes given in Table 4.3. The HOLD signal determines whether the chip is stalled or not; for example, if HOLD is held HIGH then every memory element in the chip keeps its current state. The hold mode is especially useful when synchronizing the individual systolic cells with other devices in a systolic system. Moreover, when HOLD is not asserted then the chip may operate in one of two remaining modes which is selected by the coefficient load (C_LOAD) signal. If HOLD is LOW, setting C_LOAD to HIGH effectively stalls every memory element in the chip except the serial coefficient register which is enabled to load a coefficient bit-stream. For apparent reasons, this mode is called the coefficient-load mode and can be entered without regard to the state of the chip. When HOLD is LOW and C_LOAD is LOW then all sequential elements in the chip are enabled and the coefficient loads are dissabled. This latter mode is the main mode of systolic operation and is thus named "run" mode. More will be said of these modes as the timing of the input and output switching waveforms is reviewed next.

Signals	Mode of Operation
$HOLD = 1$	hold mode
$HOLD = 0,\ C\_LOAD = 0$	run mode
$HOLD = 0,\ C\_LOAD = 1$	coefficient-load mode

Table 4.3: Modes of operation in the systolic cell.

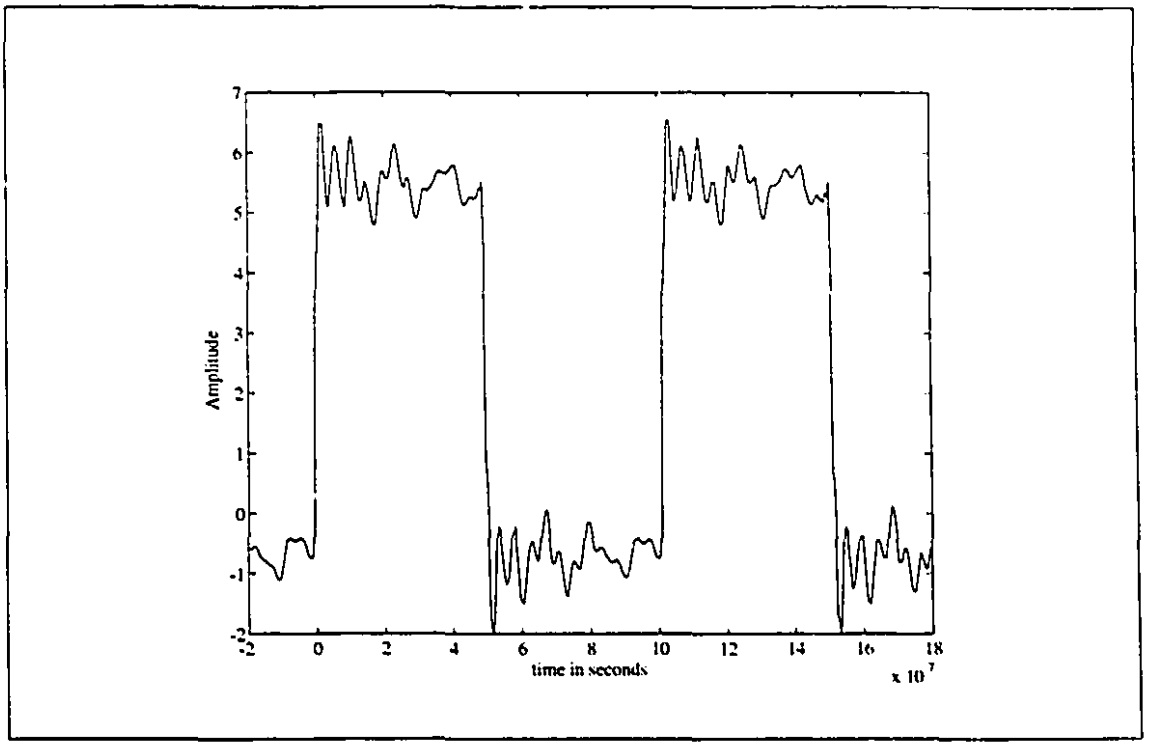

Figure 4.8: Oscilloscope trace of 1MHz CLK signal used in A.C. tests. Rise and fall times are approximately 12ns. f_{max} is slightly less than 8MHz.

I/O Timing Specification

Manufacturing tests rely on accurate timing specifications so that input signals are strobed in at the right time and output signals are sampled correctly. In the following sections input/output timing specifications are presented with reference to rising clock edges and signal transitions which indicate the beginning or ending of a "nibble-slice" of data. The term "word" is used to reference a particular 64-bit quantity of data that is further segmented into 16 "nibbles". Note that for the X_{in} and Y_{in} data streams, the last three nibbles sampled in represent the 12 bits of sign and exponent data whereas the first 13 nibbles embody the mantissa data.

Furthermore, the ability to vary timing on a per-pin basis with the data generator allows a process known as "schmooing" to be carried out. The "schmoo" tests in this thesis skewed the timing on inputs with respect to the chip clock so that setup and hold violations were detected from observed changes in correlated output sig-

nals. For instance, if ample setup time was given to an input signal which directly affected the value of a particular output signal, then all one would have to do is to decrease the setup for that input signal until a violation occurred that changed the expected output result.

Chip Initialization

The controlling state machine in the systolic cell must be initialized to a known state if it is to be synchronized with the incoming X_{in} and Y_{in} signals. For this reason a PRESET signal exists for the purposes of initializing a Johnson or "ring" counter which keeps track of the internal states of the chip. The counter is decoded to produce strobes for internal synchronization, and the effects of the PRESET signal is tightly coupled with the observability of correct results at the output port Y_{out}. For this reason, more will be said of the PRESET signal when the Y_{out} timing is discussed. Fig. 4.9 and Table 4.4 specify the required timing of the PRESET signal which effectively defines clock cycle 14. Inputs such as X_{in} must follow initialization to be synchronized to clock cycle 15. Furthermore, inputs such as Yin_Dis, Abs_Val, and Y_{in} should be synchronized to clock cycle 1.

Symbol	Parameter	Min	Units
t_{per}	CLK period	126	ns
t_{sup}	PRESET HIGH to CLK setup time	30	ns
t_{hp}	CLK to PRESET HIGH hold time	0	ns
t_{sud}	Yin_Dis HIGH to CLK setup time	50	ns
t_{sua}	Abs_Val HIGH to CLK setup time	50	ns

Table 4.4: Initialization switching characteristics. Note tnat although it is recommended that Abs_Val be synchronized to the "first" clock cycle 1, this is not a critical requirement since Abs_Val is only used 32 clock cycles afterward in the normalization stage.

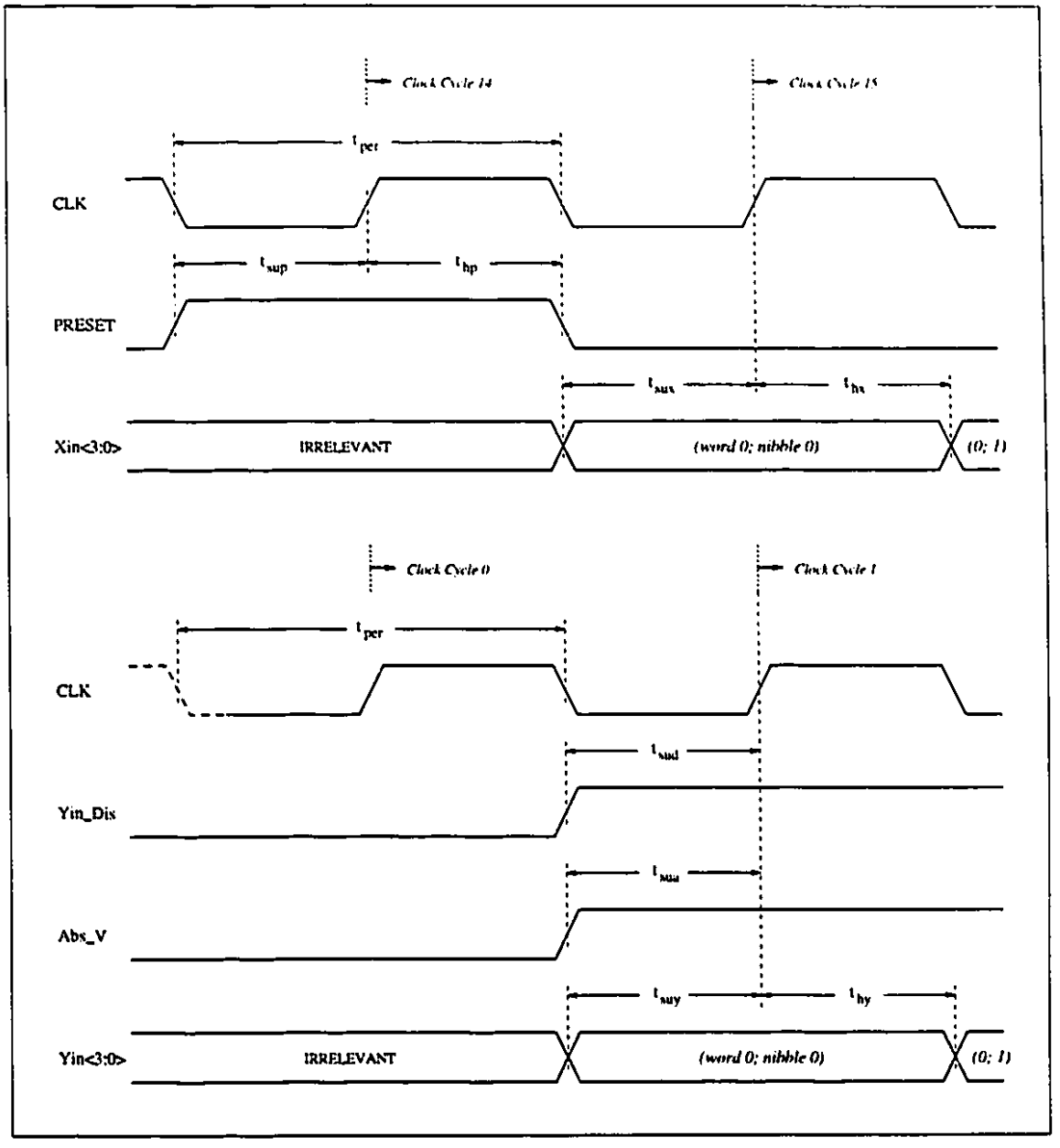

Figure 4.9: Switching waveforms at initialization. The $PRESET$ signal initializes the Johnson counter in the control block causing the next clock cycle to be defined as clock cycle 14. Xin should be synchronized to the subsequent clock cycle 15, and Yin to clock cycle 1. If asserted, Yin_Dis and Abs_Val should be synchronized to clock cycle 1.

Coefficient Loading

To serially load 64-bits into the coefficient register, the HOLD signal must be reset while the C_LOAD signal is asserted. Allowing for setup and hold times as indicated in Table 4.5, the propagation delay for a valid Cout signal as illustrated in Fig. 4.10 ranges from 35 to 45 nanoseconds. Note that propagation delays are minimum for a HIGH to LOW transition and maximum for a LOW to HIGH transition. Also note that Cout shifts out a bit 63 periods after it was first sampled at Cin.

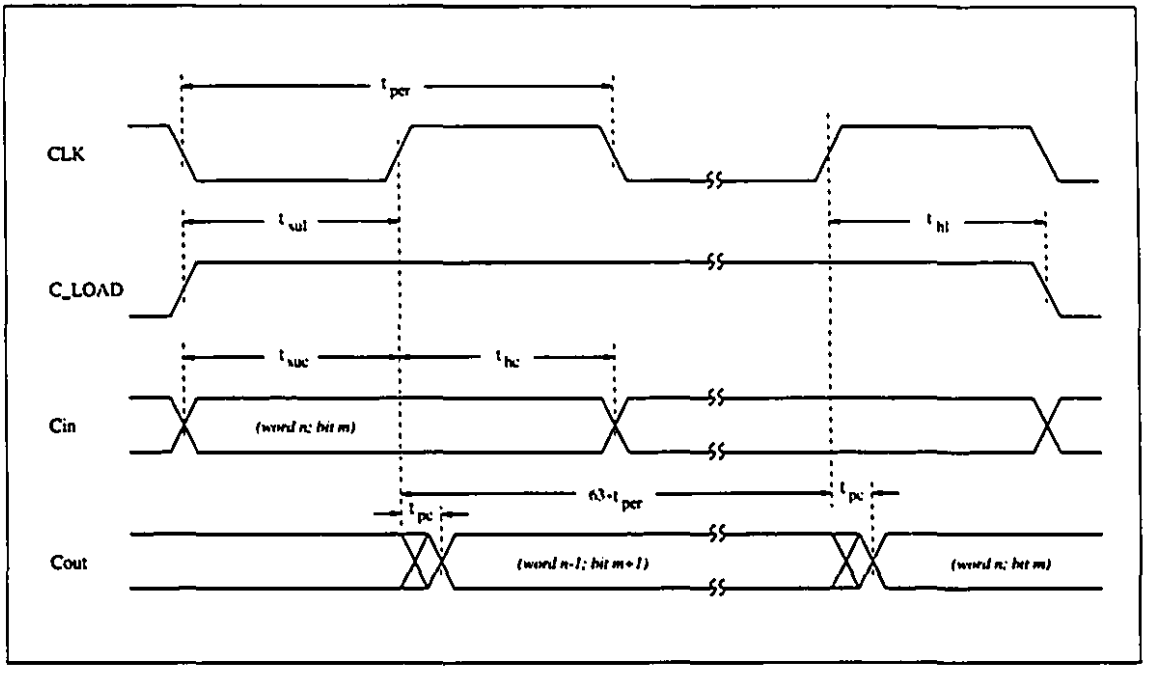

Figure 4.10: Coefficient switching waveforms in coefficient-load mode ($HOLD = 0, C\_LOAD = 1$).

Output Results

In run mode, 64-bit words are continuously being fed nibble by nibble at the input ports of X_{in} and Y_{in}. The initial X_{in} nibble is synchronized to clock cycle 15 and the associated Y_{in} nibble is synchronized to clock cycle 1. 47 clock periods after a Y_{in} nibble has been sampled, the associated processed nibble is output at the Y_{out} port as shown in Fig. 4.11.

Parameters	Description	Min	Max	Units
t_{per}	CLK period	126	—	ns
t_{sul}	C_LOAD HIGH to CLK setup time	30	—	ns
t_{hl}	CLK to C_LOAD HIGH hold time	20	—	ns
t_{suc}	Cin to CLK setup time	30	—	ns
t_{hc}	CLK to Cin hold time	0	—	ns
t_{pc}	Propagation delay from CLK to Cout Valid	35	45	ns

Table 4.5: Coefficient switching characteristics. Propagation delays are minimum for a HIGH to LOW transition and maximum for a LOW to HIGH transition.

The Y_{out} signal proved to be very difficult to test. The crux of the problem was that the last three nibbles of each Y_{out} word which represented the exponent and sign bit of the double-precision floating-point format were all set to HIGH indicating an overflow where none was expected. The mantissa seemed correct for most cases yet even that too was incorrect for certain input values. Much effort was invested in making sure that no setup or hold times were violated, yet all Y_{out} tests still failed consistently. However, enough correct values for the mantissa field were obtained to adequately perform schmoo tests with regard to the PRESET and Yin_Dis signal. In addition, the Abs_Val signal could not be accurately tested since the sign bit of the Yout words was not operational, hence a conservative estimate was entered in Table 4.4.

Fig. 4.11 and Table 4.6 display the input and output switching characteristics. The X registers function perfectly shifting out a given nibble, 31 clock periods after it was sampled. As mentioned before, Y_{out} mantissa nibbles were only accurate for certain values of X_{in}, C_{in}, and Y_{in}, and furthermore no pattern was apparent for choosing inputs that would pump out valid outputs. Naturally, this leads one to speculate whether there was a design error, an implementation error, or a fault in the silicon. The latter scenario is a likely one and will be elaborated when dealing with the issue of fabrication faults.

Lastly, the maximum clock frequency for which valid mantissa nibbles were

recorded was slightly less than 8MHz which corresponds to a 126 nanosecond period. According to Eq. 4.1 and Eq. 4.1 this corresponds to a systolic array latency of 0.000342 seconds which is insignificant to the 0.524 seconds of total execution time required (Eq. 4.1 to process a 512×512 image. Hence, a systolic system that uses an array of cells similar to the one that was fabricated and tested in this thesis is expected to have a peak performance of approximately 80 MFLOPS (Eq. 4.1).

$$\text{row latency} = [2 * (\text{ARRAYLENGTH} - 1)] + 3 = 19 \text{ pipeline cycles}$$

$$\begin{aligned}\text{array latency} &= (\text{ARRAYHEIGHT}) * (\text{row latency}) \\ &= 171 \text{ pipeline cycles} \\ &= 2,736 \text{ clock cycles} \\ &= 342,000 \text{ ns} = 0.000342 \text{ seconds}\end{aligned}$$

$$\begin{aligned}\text{execution time} &= (\text{image size}) * (1 \text{ pipeline cycle per result}) \\ &= 256\text{K pipeline cycles} \\ &= 4,194,304 \text{ clock cycles} * 125\text{ns per clock cycle} \\ &= 0.524 \text{ seconds}\end{aligned}$$

$$\begin{aligned}\text{computation rate} &= 42,000,000 \text{ FLOPs}/(\text{execution time}) \\ &= 80.2 \text{ MFLOPS}\end{aligned}$$

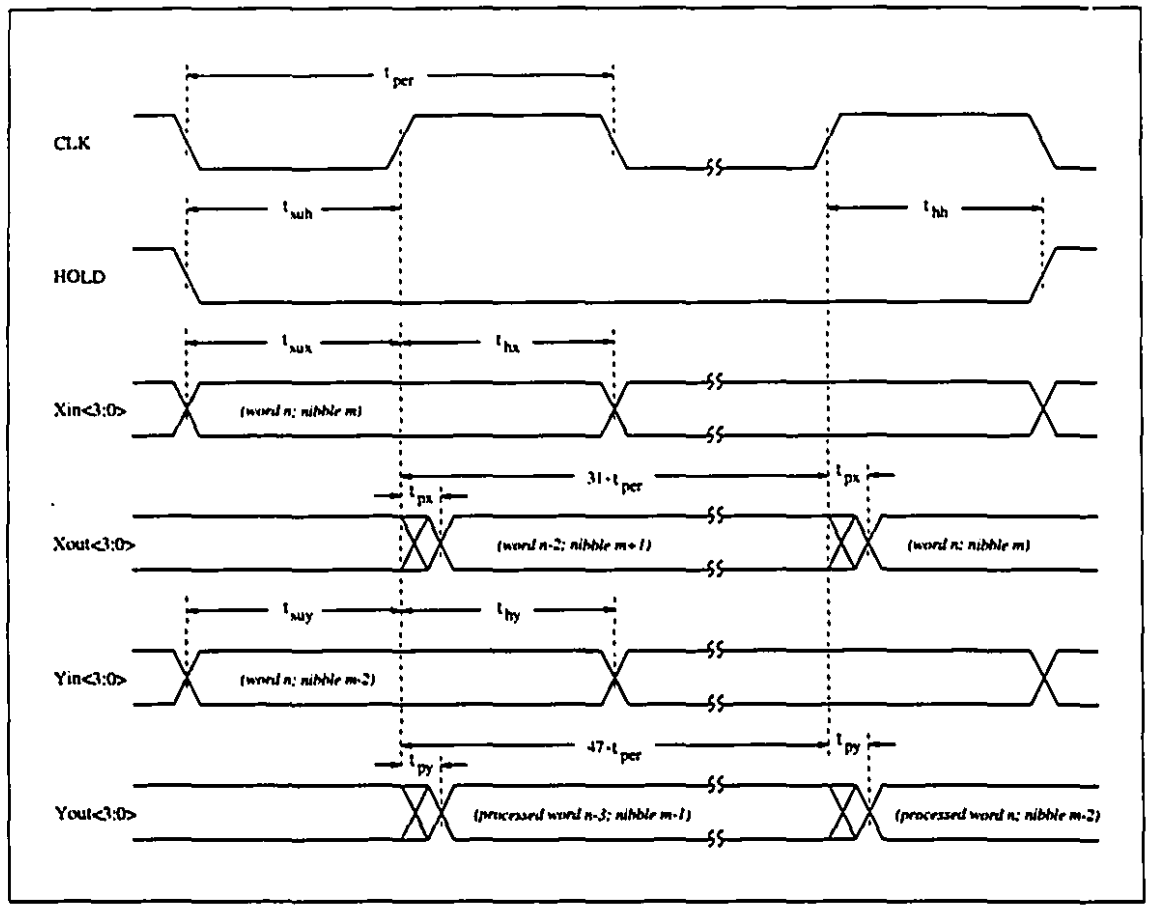

Figure 4.11: Input and output switching waveforms in run mode ($HOLD = 0$, $C\_LOAD = 0$).

Parameters	Description	Min	Max	Units
t_{per}	CLK period	126	—	ns
t_{suh}	HOLD LOW to CLK setup time	25	—	ns
t_{hh}	CLK to HOLD LOW hold time	10	—	ns
t_{sux}	Xin to CLK setup time	30	—	ns
t_{hx}	CLK to Xin hold time	0	—	ns
t_{px}	Propagation delay from CLK to Xout Valid	35	45	ns
t_{suy}	Yin to CLK setup time	30	—	ns
t_{hy}	CLK to Yin hold time	0	—	ns
t_{py}	Propagation delay from CLK to Yout Valid	38	50	ns

Table 4.6: Input and output switching characteristics. Input rise and fall times are typically 15ns. Output propagation delays are maximum for LOW to HIGH transitions.

4.2.5 Manufacturing-Test Principles

The probability of a particular transistor being defective once fabricated is minuscule. However, if one sums up these defect probabilities for all transistors in a complex chip then it is common to obtain a probability of a chip defect in the proximity of fifty percent. The proportion of fabricated chips that are fault free is referred to as the yield. Typical fabrication yields for mature processes tend to fluctuate around the 40 percent mark. It cannot be overemphasized that manufacturing tests play a critical role in VLSI design and must be given sufficient attention in this thesis. However, before such testing can be addressed with respect to the systolic convolution cell design, some testing principles are first defined.

Fault Models

To deal with the existence of good and bad parts one requires models for how faults occur and their impact on circuits. The most popular model is called the "Stuck-At" model. A faulty gate input is modeled as a "stuck at zero" or "stuck at one". These faults most frequently occur due to thin-oxide shorts (the n-transistor gate shorting with VSS or the p-transistor gate shorting with VDD) or metal-to-metal

shorts which cause the output of a gate to be "stuck at" a 0 or 1 value. Other models include "stuck-open" (open-circuit) or "shorted" (short-circuit) models.

Observability

The observability of a particular circuit node is the degree at which one can observe that node at the output pins of an integrated circuit. This measure is important when one desires to measure the output of a gate within a large circuit to check that it operates correctly. Ideally, one would like to be able to observe directly or with moderate indirection (ie. one may have to wait a few cycles) every gate output within an integrated circuit.

Controllability

Correspondingly, the controllability of an internal node within a chip is a measure of the ease of setting that node to a 0 or 1 state. An easily controllable node would be directly settable via an input pad. A node with little controllability might require thousands of cycles to get it to the right state. Recommendations will be made on how to increase the observability and controllability of the systolic cell when design for testability is examined.

Fault Coverage

Fault coverage is a formal measure of the goodness of a test program. It expresses the percentage of the chip's internal nodes that were checked for faults with the applied test vectors. The method of calculation is as follows. In a gate-level model simulation, each node is taken in sequence and held to a 0. For each node that is artificially set to 0, the simulation is run and the outputs are compared with the outputs of a known "good machine" - circuit with no nodes artificially set to 0. When a discrepancy is detected between the faulty machine and the good machine, the

fault is marked as detected and that particular simulation is stopped and the next simulation to check the next node begins. Thus one can see what percentage of the chip's internal nodes have stuck-at-0 detectability. And a similar approach can be performed for a measure of stuck-at-1 detectability. Both approaches are part of a process known as "fault-grading".

4.2.6 Manufacturing-Test Strategies for the Systolic Cell

Although the chip has already been shown to fail some basic functional tests for certain input vectors, this section will nevertheless recommend strategies for obtaining high fault coverage with the existing design. The proposed strategies are meant to test each part of the chip independently. Since the chip is partitioned into three stages, testing for each stage will be discussed in turn.

Stage 1 Mantissa Circuit

The first stage multiplication circuit is comprised of a product generator, a mantissa carry-look-ahead adder, and the exponent adder. The product generator is composed of multiplexers and ripple-carry adders. Since every multiplexed bit slice is independent from the other bit-slices, the multiplexers are easily tested by multiplexing 1 and 0 values from each of their inputs. Next, the ripple-carry adders have the desirable property of being testable with random vectors. It can be shown that a uniform distribution of input vectors ($P(inputs) = 0.5$) should yield a uniform distribution of output sums and ripple-carry-outs. The inputs of the adders are driven by the coefficient register which allows for complete controllability. Moreover, observability is relatively easy if one controls the X_{in} signal to multiplex the outputs of the seven ripple carry adders. Subsequently, with a null Y_{in} signal the content of the product register can be propagated through the second and third stages.

The mantissa carry-look-ahead adder of the first stage in the systolic cell is much

harder to test than the ripple-carry adders. The carry-look-ahead circuitry is much more complex than a straightforward ripple. For example, the carry out corresponding to bit 56 of the adder is a function of 113 inputs! A study of using random patterns to test a carry-look-ahead scheme was conducted by [Larochelle, 1991] in which the probability for controlling and observing the carry out corresponding to bit 3 was minute. The implication was that nearly a billion vectors would have to be applied before the probability of detecting a fault became acceptable.

To circumvent this impasse, Larochelle presented an alternate (deterministic) approach in which the carry-look-ahead circuit is partitioned with respect to its building blocks and tested on a block by block basis. The pyramidal structure of the adder is constructed using 4-bit carry-look-ahead building blocks. As such, a 16-bit adder is constructed with four 4-bit adders connected together with an extra level of look-ahead circuitry. Likewise, a 64-bit adder requires a third level of look-ahead circuitry and four 16-bit adders. Hence the problem breaks down to testing the basic full adder and the three levels of the look-ahead circuit. The full-adder is easily tested by applying all eight combinations of the two 1-bit inputs and the carry-in. Testing each level of carry-look-ahead circuitry is also straightforward if one realizes that the carry-propagate and carry-generate outputs of each 4-bit carry-look-ahead building block are easily controlled through the inputs to the adder. To create input test vectors for the second level look-ahead circuitry one need only replicate each bit in the initial 4-bit test vector such that a "0" becomes a "0000" and a "1" becomes a "1111" thus creating a 16-bit test vector. Similarly, the third-level is tested by repeating each bit sixteen times instead of four.

One drawback to testing the 57-bit carry-look-ahead adder as described above is that one of its addends is a bus that is fed back from the partial-product register, and this creates a controllability problem. Since some of the test vectors must be applied through this feedback path, careful planning is required in loading the partial product register with the required value.

Stage 1 Exponent Circuit

The first stage exponent circuit is composed of an 11-bit ripple adder an 11-bit carry-look-ahead adder. Controllability and observability do not require much effort for these components since the inputs are driven directly by two shift registers and the outputs are directly transmitted to State 2. Again the testing approach is the same as for the adders in the mantissa but without the controllability problems.

Stage 2 Mantissa Circuit

The second stage mantissa addition circuit is made up of multiplexed registers, a look-ahead adder, and a look-ahead two's complementer. The multiplexers of the shift registers are tested during the testing of the Stage 2 exponent circuit. The two's complementer is a look-ahead adder that adds 1 to the inverted output of the other adder. Both adders are tested using the look-ahead adder approach described above. Controllability is achieved using the Y_{in} register and the content of the product register in Stage 1. Observability is realized once the result is propagated through Stage 3.

Stage 2 Exponent Circuit

The second stage exponent circuit constitutes two registers and an 11-bit carry-look-ahead adder. One of the adder inputs is derived from a multiplexer which selects a hard-wired "+8" value or a "-1" value. The result from Stage 1 as well as the Y_{in} input can be used to load the content of the registers. The test results get transmitted to Stage 3 every 16 clock cycles.

Stage 3 Normalization Circuit

The third stage mantissa circuit has no combinational logic. The registers with their multiplexed inputs are tested at the same time as the exponent circuitry. The exponent circuitry houses a single register and a carry-look-ahead adder which is configured to add "+8" or "-1" until the leading 1 in the mantissa register is immediately to the left of the binary point. Y_{in} is used to load the contents of the exponent and mantissa registers, and loading a zero valued X_{in} will propagate the same Y_{in} value to stage 3 which is directly observable from Y_{out}.

4.2.7 Design for Testability

A major shortcoming in this systolic cell design was that it was not designed for testability. This section surveys the design techniques that may be used to make all nodes in a chip both controllable and observable, and thus testable. There are four main approaches which are categorized as ad-hoc, scan-based, self-test, and IDDQ. Once these categories are surveyed, recommendations will be made for redesigning the systolic cell chip for increased testability.

Ad-hoc testing is a collection of design ideas aimed at reducing the combinational explosion of testing. One common technique is to partition large sequential circuits in order to reduce the number of cycles required for testing - the approach being one of divide and conquer where a big circuit with many inputs is made easier to test by reducing it into multiple smaller sub-circuits each with fewer inputs. A second ad-hoc approach is to add extra test-points for improved observability. Still another method is to add multiplexers to provide alternate signal paths that can be enabled during a "test mode" of a circuit. Lastly, one can provide for easy state reset for improved controllability.

Scan-based approaches convert all registers in the circuit into serial shift regis-

ters which are "chained" together via their serial inputs and outputs. Testing proceeds by serially clocking the data through the "scan-chain" to the right point in the circuit, running a single system clock cycle and serially clocking the data out for observation.

Self-test and built in test techniques rely on augmenting circuits to allow them to perform operations on themselves that prove correct operation. One method is signature analysis [Frowerk, 1977] [Nadig, 1977]. This involves the use of a pseudo-random sequence generator to generate the input signals for a section of combinational circuitry and then using a signature analyzer to observe the output signals.

One increasingly popular method of testing for bridging faults is called IDDQ (VDD supply current Quiescent) or current-supply monitoring [Acken, 1983] [Lee and Breuer, 1992]. This relies on the fact that when a CMOS logic gate is not switching, it draws no DC current (except for leakage). When a bridging fault occurs, for some combination of input conditions a measurable DC I_{DD} will flow. Testing consists of applying the normal vectors, allowing the signals to settle, and then measuring I_{DD}. It is highly likely that given the unusually high power supply current recorded in the DC characteristics Table 4.2 that the tested systolic cell chip did indeed have a bridging fault. As just demonstrated, this kind of testing gives a form of indirect massive observability at no cost in terms of circuit overhead.

Having encountered the difficulties in devising testing schemes for circuits such as those found in Stage 1 of the systolic cell, it is now clear that a partial scan chain would greatly simplify test vector generation. For example, the Stage 1 controllability and observability problems would be completely resolved by simply configuring the partial product register into a shift register during a newly created test-mode. Moreover, if it were chained to the X register such that X_{in} and X_{out} pins could be used to control and observe the contents of the partial-product register during test-mode then only a few very minor gate changes and the addition of one pin to designate the test-mode would have to be incurred.

4.3 Summary

Before the systolic convolution chip could be fabricated a deep-seated knowledge was required of the systolic design, the CMOS3 DLM process, the associated design rules, the CADENCE design and validation tools, and the submission process. Once the chip was validated in the CADENCE environment and the Vantage VHDL simulator, and after it was sent for fabrication, the testing environment was sufficiently automated such that testing process could be performed in an efficient manner. Functional tests were run that spawned an I/O and timing specification for the sytolic cell. These tests exposed a defect with the fabricated chip. Furthermore, manufacturing test strategies were given and approaches for re-designing the systolic convolution chip for testability were recommended.

Chapter 5 Conclusion

The convolution operation was perceived as the most frequently used operation in image-processing tasks and as such it was targeted for performance improvement. Due to the huge volume of data and floating-point arithmetic calculations needed to convolve a standard sized image, convolution was found to be overwhelming with regard to the available computational power and memory bandwidth of single processor machines.

However, the convolution operator can be easily "parallelized" and sped up on various general-purpose parallel processing computers. Of the parallel processor classes considered, the synchronous class appeared to be the best suited for low-level image-processing applications. Specifically, since systolic arrays are simple, modular, expandable, and yield high performance at a lower relative cost, they meet the architectural challenges of special-purpose systems, and are therefore the preferred parallel processing solution for the convolution problem.

Hence, a systolic solution is presented that is to employ a 9×9 systolic array of processing cells. Each systolic cell is implemented on a VLSI chip and is organized into three stages which perform the required double precision floating-point multiplication, addition, and normalization operations in a pipelined fashion.

Once the chip was validated in the CADENCE™ environment and on the Vantage VHDL simulator, and after it was sent for fabrication, the testing environment was sufficiently automated such that the testing process could be performed in an efficient manner. Functional tests were run that were used to generate I/O port timing specifications. Unfortunately, some of the input vectors for these tests exposed a defect with the fabricated chip. Since the IDDQ tests pointed in the direction of a bridging fault, a manufacturing error is suspected. Lastly, manufacturing

test strategies were given and approaches for re-designing the systolic convolution chip for testability were recommended.

References

[Acken, 1983] J. M. Acken, "Testing for bridging faults (shorts) in cmos circuits," in *Proceedings of the 20th IEEE/ACM Design Automation Conference*, (Miami Beach, Fla.), pp. 717–718, June 1983.

[Amdahl, 1967] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS 1967 Spring Joint Computer Conf. 30*, (Atlantic City, N.J.), pp. 483–485, April 1967.

[Axel *et al.*, 1983] L. Axel, P. H. Arger, and R. A. Zimmerman, "Applications of computerized tomography to diagnostic radiology," *Proceedings of the IEEE*, vol. 71, pp. 291–431, March 1983.

[Ballard and Brown, 1982] D. H. Ballard and C. M. Brown, *Computer Vision*. Prentice Hall, 1982.

[Balsara and Irwin, 1991] P. T. Balsara and M. J. Irwin, "Image processing on a memory architecture," *Journal of VLSI Signal Processing*, vol. 2, pp. 313–324, May 1991.

[Barnes, 1968] G. H. Barnes, "The Illiac IV Computer," *IEEE Transactions on Computers*, vol. C-17, pp. 746–757, 1968.

[Batcher, 1980] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, vol. C-29, p. 836, 1980.

[Berry *et al.*, 1988] M. Berry, D. Chen, and D. K. P. Koss, "The perfect club benchmarks: Effective performance evaluation of supercomputers," tech. rep., Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, Illinois, November 1988.

[Botzas and Masson, 1990] A. Botzas and E. L. Masson, "First in first out (fifo) queue," tech. rep., McGill University, 1990.

[Boudreault and Malowany, 1986] Y. Boudreault and A. Malowany, "A VLSI convolver for a robot vision system," *Proceedings of the Canadian Conference on Very Large Scale Integration*, pp. 265–270, 1986.

[Brassard and Bratley, 1988] G. Brassard and P. Bratley, *Algorithmics, Theory and Practice*. EngleWood Cliffs, New Jersey: Prentice Hall, 1988.

[Briggs and Hwang, 1984] F. Briggs and K. Hwang, *Computer Architectures and Parallel Processing*. New York, N.Y.: McGraw-Hill, 1984.

[Briot, 1986] M. Briot, *Robot Vision and Sensory Controls*. Springer-Verlag, Berlin: IFS (Publications) Ltd., 1986.

[Burrus and Parks, 1985] C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms: Theory and Implementations*. New-York: Wiley Press, 1985.

[Côté, 1990] J.-F. Côté, "The design of a testable floating point convolution processor," Master's thesis, McGill University, November 1990.

[Crowther, 1985] W. Crowther, "Performance measurements on a 128-node butterfly parallel processor," in *Proceedings of the International Conference on Parallel Processing*, pp. 531–535, IEEE Computer Society Press, 1985.

[Drolet, 1992] J. Drolet, "The design of a floating point convolution system," Master's thesis, McGill University, August 1992.

[Duncan, 1990] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, pp. 5–16, February 1990.

[Estreich and Dutton, 1982] D. B. Estreich and R. W. Dutton, "Modeling Latch-Up in CMOS Integrated Circuits and Systems," *IEEE Transactions on CAD*, vol. CAD-1, pp. 347–354, October 1982.

[Feng, 1981] T. Feng, "A survey of interconnection networks," *IEEE Computer*, pp. 12–27, December 1981.

[Ferguson, 1991] W. Ferguson, Jr., "Selecting math coprocessors," *IEEE Spectrum*, vol. 28, pp. 38–41, July 1991.

[Flynn, 1966] M. J. Flynn, "Very high speed computing systems," *Proceedings of the IEEE*, vol. 54, pp. 1901–1909, December 1966.

[Freer, 1987] J. Freer, *Systems Design with Advanced Microprocessors*. Howard W. Sams & Co., 1987.

[Frowerk, 1977] R. A. Frowerk, "Signature Analysis – A New Digital Field Service Method," *Hewlett Packard Journal*, pp. 2–8, May 1977.

[Gajski, 1986] D. Gajski, "CEDAR," in *Digest of Papers, Compcon* (A. G. Bell, ed.), IEEE Computer Society Press, 1986.

[Gentleman and Kung, 1981] W. Gentleman and H. Kung, "Matrix triangularization by systolic arrays," *Proc. SPIE, Real-Time Signal Processing IV, Society of Photo-optical Instrumentation Engineers*, vol. 298, 1981.

[Gonzalez and Wintz, 1987] R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Reading, Massachusetts: Addison-Wesley, 1987.

[Guibas *et al.*, 1979] L. Guibas, H. Kung, and C. Thompson, "Direct VLSI implementation of combinatorial algorithms," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication*, pp. 509–525, January 1979.

[Hall, 1979] E. L. Hall, *Computer Image Processing and Recognition*. Academic, 1979.

[Hartline and Ratliff, 1954] H. R. Hartline and F. Ratliff, "Spatial summation of inhibitory influences in the eye of limulus," *Science*, vol. 20, no. 3124, p. 781, 1954.

[Hartline and Ratliff, 1957] H. R. Hartline and F. Ratliff, "Inhibitory interaction of receptor units in the eye of the limulus," *Journal of General Physiology*, vol. 40, no. 3, pp. 357–376, 1957.

[Hartline, 1949] H. R. Hartline, "Inhibition of activity of visual receptors by illuminating nearby retinal elements in the limulus eye," *Federation Proceedings*, vol. 8, no. 3, p. 69, 1949.

[Haule, 1990] D. D. Haule, "Design of a VLSI system for image processing," Master's thesis, McGill University, March 1990.

[Hennessy and Patterson, 1990] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Inc., 1990.

[Hillis, 1985] W. D. Hillis, *The Connection Machine*. Massachusetts Institute of Technology Press, 1985.

[Hockney and Jesshope, 1988] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2, Architecture, Programming and Algorithms*. Bristol, England, and Philadelphia: Adam Hilger Ltd., 1988.

[Hord, 1990] R. M. Hord, *Parallel Supercomputing in SIMD Architectures*. Boca Raton, Ann Arbor, Boston: CRC Press, 1990.

[Horspool, 1986] R. N. Horspool, *C Programming in the Berkeley Unix Environment*. Prentice Hall, 1986.

[IBM, 1985] IBM Corp., *IBM 3090 System Summary–Engineering/Scientific*, 1985.

[IEE, 1985] IEEE, *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, 1985.

[Int, 1993] Intel Corporation, *Parallel Supercomputers*, 1993.

[Iverson, 1992] L. Iverson, *McImage Library: McRCIM Image Processing Library and Development Environment for the MasPar MP-1*. Montreal, Quebec, Canada, 1992.

[Karp, 1987] A. H. Karp, "Programming for parallelism," *IEEE Computer*, pp. 43–57, May 1987.

[Kohonen, 1987] T. Kohonen, *Content-Addressable Memories*. New York, N.Y.: Springer-Verlag, 1987.

[Kuck, 1980] D. Kuck, "High speed multiprocessing and compilation techniques," *IEEE Transactions on Computers*, vol. C-29, pp. 763–776, 1980.

[Kung and Leiserson, 1978] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," *Sparse Matrix Symposium (SIAM 1979)*, pp. 256–282, 1978.

[Kung, 1982] H. T. Kung, "Why systolic architectures?," *Computer*, vol. 15, pp. 37–46, January 1982.

[Kung, 1987] S.-Y. Kung, "Wavefront array processors–concept to implementation," *IEEE Computer*, vol. 20, pp. 18–33, July 1987.

[Larochelle, 1991] F. Larochelle, "VLSI design of a double precision floating point convolution systolic cell," Master's thesis, McGill University, March 1991.

[Larson, 1984] J. L. Larson, "An introduction to multitasking on the Cray X-MP2 multiprocessor," *IEEE Computer*, pp. 62–69, July 1984.

[Lee and Breuer, 1992] K.-J. Lee and M. A. Breuer, "Design and Test Rules for CMOS Circuits to Facilitate IDDQ Testing of Bridging Faults," *IEEE Transactions on CAD*, vol. 11, pp. 659–670, May 1992.

[Lehman and Kung, 1980] P. Lehman and H. Kung, "Systolic (VLSI) arrays for relational database operations," *Proc. ACM - Sigmod 1980 Int'l Conf. Management of Data*, pp. 105–116, May 1980.

[Levine, 1985] M. D. Levine, *Vision in Man and Machine*. McGraw-Hill Inc., 1985.

[Lindskog, 1988] B. Lindskog, *PICAP3*. Dissertations no. 176, Linkoping Studies In Science And Technology, Linkoping, Sweden, 1988.

[Lipovski and Malek, 1987] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*. New York, N.Y.: Wiley and Sons, 1987.

[Lovett and Thakkar, 1988] T. Lovett and S. Thakkar, "The sequent symmetry multiprocessor system," in *Proceedings of the 1988 International Conference of Parallel Processing*, (University Park, Pennsylvania), pp. 303–310, 1988.

[Lubeck *et al.*, 1985] O. Lubeck, J. Moore, and R. Mendez, "A comparison of three supercomputers: Fujitsu VP-200, Hitachi 5810/20, and Cray X-MP/2," *Computer*, pp. 10–24, December 1985.

[Malowany *et al.*, 1990] A. S. Malowany, J. Drolet, J. Panisset, J. F. Côté, and F. Larochelle, "A double precision floating point convolution system," in *Proceedings of the ASME International Computers in Engineering Conference, Vol. 2*, (Boston, Mass.), pp. 1–6, American Society of Mechanical Engineers, August 1990.

[Malowany *et al.*, 1991] A. S. Malowany, J. Drolet, and J. F. Panisset, "Design of a floating point convolution processor," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, (Quebec City, Canada), pp. 13.5.1–13.5.4, Canadian Society for Electrical and Computer Engineering, September 1991.

[Marr and Hildreth, 1980] D. Marr and E. Hildreth, "Theory of edge detection," in *Proceedings of the Royal Society of London, Ser. B, Vol 207*, pp. 187–217, 1980.

[Mas, 1992] MasPar Computer Corporation, *MasPar MP-1 Hardware Manuals*, 1992.

[McMahon, 1989] F. McMahon, "The Livermore FORTRAN kernels: A computer test of numerical performance range," tech. rep., Lawrence Livermore National Laboratory, Univ. of California, Livermore, California, December 1989.

[McRCIM, 1990] McRCIM, "Mcgill research center for intelligent machines annual report," tech. rep., McGill University, 1990.

[Mead and Conway, 1980] C. Mead and L. Conway, *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[Mulders, 1987] M. A. Mulders, *Remote Sensing in Soil Science*. New-York: Elsevier Science Pub., 1987.

[Nadig, 1977] H. J. Nadig, "Signature Analysis – Concepts, Examples, and Guidelines," *Hewlett Packard Journal*, pp. 15–21, May 1977.

[nCU, 1992] nCUBE Corporation, *nCUBE 2: Technical Overview*, 1992.

[Offen, 1985] R. J. Offen, *VLSI Image Processing*. Collins Professional and Technical Books, 1985.

[Oppenheim and Schafer, 1989] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. EngleWood Cliffs, New Jersey: Prentice Hall, 1989.

[Palmer, 1986] J. F. Palmer, "A VLSI Parallel Computer," in *Digest of Papers, Compcon* (A. G. Bell, ed.), IEEE Computer Society Press, 1986.

[Panisset *et al.*, 1990] J. F. Panisset, J. Drolet, J. F. Côté, F. Larochelle, and A. S. Malowany, "A floating point convolution system," in *33rd Midwest Symposium on Circuits and Systems*, (Calgary, Alb., Canada), pp. 397–400, IEEE Circuits and Systems Society, August 1990.

[Par, 1991] Parsytec GmbH, *Beyond the Supercomputer, Parsytec GC*, 1991.

[Pfister, 1985] G. F. Pfister, "The IBM Research Parallel Processor Prototype (RP3)," in *Proceedings of the International Conference on Parallel Processing*, IEEE Computer Society Press, 1985.

[Press *et al.*, 1988] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C, The Art of Scientific Computing*. Cambridge, New York, Port Chester, Melbourne, Sydney: Cambridge University Press, 1988.

[Preston, 1989] K. Preston, Jr., "The Abingdon Cross benchmark survey," *Computer*, pp. 9–18, July 1989.

[Proakis and Manolakis, 1988] J. G. Proakis and D. G. Manolakis, *Introduction to Digital Signal Processing*. New-York: Macmillan, 1988.

[Rattner, 1985] J. Rattner, "Concurrent processing: A new direction in scientific computing," in *AFIPS Conference Proceedings*, vol. 54, p. 157, 1985.

[Robbins and Robbins, 1989] K. A. Robbins and S. Robbins, *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing*. Springer-Verlag, 1989.

[Schmidt and Caesar, 1991] U. Schmidt and K. Caesar, "Datawave: A single-chip multiprocessor for video applications," *IEEE Micro*, vol. 11, pp. 22–25,88–93, June 1991.

[Schumann, 1904] F. Schumann, "Einige beobachtungen über die zusammenfassung von gesichtseindrucken zu einheiten," *Psychologische Studien*, vol. 1, pp. 1–32, 1904.

[Seitz, 1985] C. L. Seitz, "The cosmic cube," in *Comm. ACM*, vol. 28, (New York, N.Y..), pp. 22–23, 1985.

[Shimizu *et al.*, 1988] H. Shimizu, N. Chubachi, and J. Kushibiki, *Acoustical Imaging*. New York, London: Plennum Press, 1988.

[Shore, 1973] J. E. Shore, "Second thoughts on parallel processing," *Comput. Elect. Eng.*, vol. 1, pp. 95–109, 1973.

[Srini, 1986] V. Srini, "An architectural comparison of dataflow systems," *IEEE Computer*, vol. 19, pp. 68–88, March 1986.

[Thi, 1992] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, 1992.

[Treleaven *et al.*, 1982] P. Treleaven, D. Brownbridge, and R. Hopkins, "Data-driven and demand driven computer architectures," *ACM Computing Surveys*, vol. 14, pp. 93–143, March 1982.

[Troutman, 1986] R. R. Troutman, *Latch-Up in CMOS Technology: The Problem and Its Cure*. Boston, Mass.: Kluwer Academic Publishers, 1986.

[Uhr, 1986] L. Uhr, *Evaluation of Multicomputers for Image Processing*. Cambridge, Mass: Academic Press, 1986.

[van Zee and van de Vorst, 1989] G. A. van Zee and J. G. G. van de Vorst, *Shell Conference on Parallel Computing*. New York: Springer-Verlag, 1989.

[Weste and Eshraghian, 1993] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective, Second Edition*. Addison-Wesley, 1993.

[Wilson, 1993] D. Wilson, "The Silicon Graphics Indigo R4000 workstation," *Unix Review*, p. 53, January 1993.

Appendix A Convolution Benchmark

```
/*****************************************************************
 *
 * NAME:   conv - convolution benchmark
 *
 * DESCRIPTION: This benchmark measures the time used in executing
 *              the process code within the main loops of a convolution
 *              routine.  The defines below ensure maximum flexibility
 *              and serve to parametrize the benchmark.
 *              This benchmark exists solely to isolate performance of
 *              double-precision floating-point arithmetic found in
 *              the heart of image-processing programs that employ
 *              convolution operators.
 *
 * COMPILER: GCC Version 2.3.3
 * OPERATING SYSTEM: Sun UNIX 4.2 Release: 3.5
 *
 * DATE:     Program Updated 93/02/23.
 * AUTHOR:   Anthony Botzas
 *
 *****************************************************************/


#include <stdio.h>
#include <sys/time.h>
#include "cputime.h"


#define ISIZE 512
#define IBEGIN 4
#define IEND 507

#define JSIZE 512
#define JBEGIN 4
#define JEND 507

#define MSIZE 9
#define MBEGIN 0
#define MEND 8

#define NSIZE 9
#define NBEGIN 0
#define NEND 8

#define IOFFSET 4
#define JOFFSET 4


/* Global arrays for image data and kernel */
double I[ISIZE][JSIZE], Y[ISIZE][JSIZE], W[MSIZE][NSIZE];
```

```
main()
{
        int     i,j,m,n;
        double t1,t2;

        /* initialization */
        for (i=0; i<ISIZE; ++i)
            for (j=0; j<JSIZE; ++j)
                I[i][j]=3.14;
        for (m=0; m<MSIZE; ++m)
            for (n=0; n<NSIZE; ++n)
                W[m][n]=2.72;

        t1 = cputime(User);

        /* main loops of convolution operation */
        for (i=IBEGIN; i<=IEND; ++i)
            for (j=JBEGIN; j<=JEND; ++j)
                for (m=MBEGIN; m<=MEND; ++m)
                    for (n=NBEGIN; n<=NEND; ++n)
                        Y[i][j] += W[n][m] * I[i+n-IOFFSET][j+m-JOFFSET];

        t2 = cputime(User);

        /* print user mode seconds */
        printf ("time=%g secs.\n", t2-t1);

}


/*************************************************************************
 *
 * NAME:   cputime
 *
 * USAGE:  double cputime (option);
 *         enum cputime_option option;
 *
 * DESCRIPTION: Returns the time consumed by a process.
 *              The options are as follows:
 *
 *    System  : Time spent executing system calls for the process.
 *    User    : Time spent executing the processes own code.
 *    All     : All time spent executing for the process.
 *
 *************************************************************************/

#include <sys/time.h>
#include <sys/resource.h>
#include "cputime.h"

double cputime(option)
enum cputime_option option;
{
 static struct rusage ru;
 double result;

 getrusage(RUSAGE_SELF,&ru);

 result = 0;

 if (option == System || option == All)
    result = ru.ru_stime.tv_sec + (ru.ru_stime.tv_usec/1000000.);

 if (option == User || option == All)
    result = ru.ru_utime.tv_sec + (ru.ru_utime.tv_usec/1000000.);

 return (result);
}
```

```
/*****************************************************************
 *
 * HEADER: cputime.h
 *
 * DESCRIPTION: Definitions for the function cputime.
 *
 *****************************************************************/

double cputime ();

enum cputime_option
    { System,       /* Get time used executing system calls */
      User,         /* Get time used executing process code */
      All           /* Get all of the time used by a process */
    };
```

Appendix B Parallel Convolution Benchmark

```
/*****************************************************************
 *
 * NAME:   parconv - parallel convolution benchmark
 *
 * DESCRIPTION: This is parallelized version of the CONV benchmark.
 *              It runs ONLY on MasPar MP-x architectures.  However,
 *              the source code can conceivably be ported to other SIMD
 *              architectures.  The objective is to measure the time
 *              used in executing the double-precision floating-point
 *              process code within the computational portion of a
 *              parallel convolution routine.
 *
 * MASPAR MP-1: The MP-1 is an array processor comprised of a front
 *              end DECstation and a data parallel unit (DPU).  The
 *              DPU consists of the array control unit (ACU) and the
 *              processing element array (PE array). The PE's are
 *              arranged in a 64 by 32 grid with nearest-neighbour
 *              interconnects.  Thus there are 2K PE's on the MP-1.
 * VIRTUAL
 * PROGRAMMING: Optimal image processing performance would be achieved
 *              if the dimensions of the image array matched those of
 *              the PE array.  Then there would be one PE for each
 *              pixel in the image.  However, this is not the case
 *              for the MP-1 since the 64x32 PE array cannot
 *              accommodate the 512x512 image all at once.  Thus,
 *              virtualization techniques [Iverson, 1992] have been
 *              developed to present a virtual programming environment
 *              which detaches the MP-1 programmer from the physical
 *              PE array dimensions.  Hence, using "virtualization
 *              macros" such as "ImageOp1" and "ImageOp2" below, the
 *              programmer indicates that the instructions enclosed in
 *              braces, {}, should be performed in parallel on each PE
 *              of a virtual array that matches the dimensions of the
 *              image.  Ergo, virtualization is accomplished via the
 *              pre-processor which replaces each virtalization macro
 *              with a double loop over the image.  Conceptually, the
 *              macros break up the 512x512 image into 128 "tiles"
 *              each with pixel dimensions 64x32.  Therefore, each PE
 *              instruction is iterated once for every tile in the
 *              image.
 *
 *
 * COMPILER: AMPL_CC Version 3.0.12
 * OPERATING SYSTEM: ULTRIX  4.2a
 *
 * DATE:     Program Updated 93/02/23.
 * AUTHOR:   Anthony Botzas
 *
 *
 *****************************************************************/
```

```
/* INCLUDES */

#include <stdio.h>

#ifdef _MPL
#include <mpl.h>
#include <mprpc/rpc.h>
#else
#include <rpc/rpc.h>
#endif

#include <McImage/image.h>
#include <McImage/Dimage.h>
#include <McImage/Dinet.h>


/* DEFINES */
/* -consistent with the CONV benchmark */

#define ISIZE   512
#define JSIZE   512
#define MSIZE   9
#define NSIZE   9
#define IOFFSET 4
#define JOFFSET 4


main()
{


/* INITIALIZATION */

    /* Define input and output images. These are simple structures */
    /* which embody both the image and a description of the */
    /* virtualization mechanism which it uses. */
    doubleImage InputImage;
    doubleImage OutputImage;

    register double W[MSIZE][NSIZE];  /* ACU registers (CReg's) */
    register int    m, n;

    register plural double sum; /* PReg register defined on each PE */

    /* Allocate the input image and the convolution result image. */
    double_allocaImage(&InputImage, ISIZE, JSIZE);
    double_allocaImage(&OutputImage, ISIZE, JSIZE);

    /* Initialize kernel on CReg's. */
    for (m=0; m<MSIZE; ++m)
        for (n=0; n<NSIZE; ++n)
            W[m][n] = 2.72;

    /* Virtualization macro that initializes input image. */
    ImageOp1( &InputImage,double,In_p, {
        *In_p = 3.14; /* This instruction is executed on every PE. */
                      /* Note that virtualization will iterate this */
                      /* parallel instruction until the entire image */
                      /* dimensions have been traversed. */
    } );
```

```
/* COMPUTATION */

    /* Start a real time counter in the Data Parallel Unit (DPU). The */
    /* counter increments once every machine cycle (80ns for the MP-1).*/
    dpuTimerStart();


    /* Virtualization macro that weights the neighbouring pixels and */
    /* accumulates the result in the corresponding output image pixel. */
    ImageOp2( &OutputImage,double,Out_p, &InputImage,double,In_p,{

        sum = 0; /* temporary accumulation register */

        for (m=0; m<MSIZE; ++m)
            for (n=0; n<NSIZE; ++n)
                sum += W[m][n] *
                    double_inet(&InputImage,m-IOFFSET,n-JOFFSET);
                    /* inet fetches a neighbouring pixel given a */
                    /* relative displacement (dx,dy) */

        *Out_p = sum;  /* store value of convolution sum */

    } );

    printf("time=%g secs.\n", dpuTimerElapsed() );

}
```

Appendix C Overview of the IEEE Floating-Point Standard

Manufacturers have in the past employed proprietary formats to store real numbers. Some of those formats did not even ensure correctly rounded results of common operations [Ferguson, 1991]. Due to the increased use of floating-point arithmetic, the need for a standard representation became necessary. In 1985, an IEEE working group presented the IEEE 754 standard whose goal was to improve software and hardware portability. The standard describes such things as the floating-point format (single and double precision), the combination (rounding) of floating-point through common operations such as addition, multiplication and division, and behaviour under error conditions (division by zero, overflow, etc.).

The converters on the hardware accelerator board comply with this standard. Hence, an overview of the double precision floating-point representation is presented. For complete details on the IEEE 754 standard, the reader is referred to [IEE, 1985].

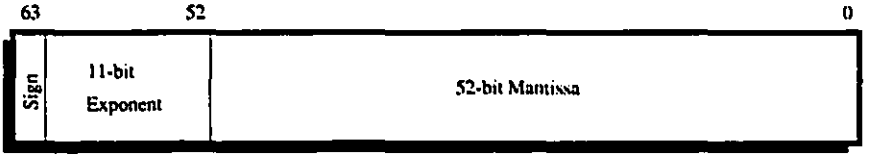

Figure C.1: Double precision floating-point representation.

As depicted in Figure C.1, a double precision number is 64 bits long: one bit for the sign (0 = positive, 1 = negative), 11 bits for the exponent and 52 bits for the mantissa. To ensure a unique internal representation for each floating-point number, the exponent is adjusted so that the mantissa has an implied 1 before its binary point. This "normalization" implies that the 1 in front of the binary point need not be stored since it is always present. So, although there are 52 bits in the mantissa, 53-bit precision is provided.

Moreover, the exponent value is represented using the "excess 1023 notation" which implies an exponent range of -1023 to 1024. The decimal value, d, of a floating-point number is hence given by

$$d = (-1)^s * (1.mant) * 2^{(exp-1023)} \tag{C.1}$$

where s is the sign, $mant$ and exp are the decimal equivalent of the mantissa and the exponent respectively. In excess 1023 notation, an exponent with the maximum value represents infinity (∞) only if the mantissa is zero otherwise it is NaN (not a number). An exponent with the minimum value represents a zero if the mantissa is null, otherwise it indicates an underflow.

```
-------------------------------------------------------------------------
--  TOPLEVEL.VHDL
--
--  This is a toplevel simulation of a linear systolic convolution array.
--  Inputs and partial results move systolicly in the same direction;
--  however, THE PARTIAL RESULTS MOVE FASTER THAN THE INPUTS .
--  The convolution weights do not move.
--
--  The test bench below automatically generates an array of length
--  ARRAYLENGTH and feeds it with a stream of input pixels.
--  2*(ARRAYLENGTH-1)+1 clock edges after an input sample enters the
--  first cell, a valid convolution sum of that sample and the previous
--  ARRAYLENGTH samples appears at the output of the last cell.
-------------------------------------------------------------------------


-------------------------------------------------------------------------
-- Clock Generator
-------------------------------------------------------------------------

entity clock_gen is
  generic (Tpw : time := 5 ns);   -- default clock pulse width
  port (phi : out bit);           -- one phase
end clock_gen;


architecture behaviour of clock_gen is
  constant CLOCK_PERIOD : time := 2*(Tpw);
begin

  clock_driver: process
  begin
    phi <= '0', '1' after Tpw;
    wait for CLOCK_PERIOD;
  end process clock_driver;

end behaviour;


-------------------------------------------------------------------------
-- Synchronous Register
-------------------------------------------------------------------------

entity synch_reg is
  generic (Tpd : time := 1 ns);       -- default propagation delay
  port (d : in integer;
    q : out integer := 0;         -- default register content
    clk : in bit);
end synch_reg;


architecture behaviour of synch_reg is
begin

  process
  begin
```

```
    wait until clk = '1';
    q <= d after Tpd;
  end process;

end behaviour;


-------------------------------------------------------------------------
-- Synchronous Convolver
-------------------------------------------------------------------------

entity convolver is
  generic (Tpd : time := 1 ns;        -- default propagation delay
       weight : integer := 1);    -- default convolution weight
  port (pixel  : in integer;
    insum  : in integer;
    outsum : out integer := 0;    -- default contents
    clk    : in bit);
end convolver;


architecture behaviour of convolver is
begin

  process
  begin
    wait until clk = '1';
    outsum <= insum + (weight * pixel) after Tpd;
  end process;

end behaviour;


-------------------------------------------------------------------------
-- systolic cell
-------------------------------------------------------------------------

entity syst_cell is
  port (Xin  : in integer;
    Xout : out integer;
    Yin  : in integer;
    Yout : out integer;
    clk  : in bit);
end syst_cell;


architecture structure of syst_cell is

  component synch_reg
    port (d   : in integer;
      q   : out integer;
      clk : in bit);
  end component;

  component convolver
    generic (Tpd : time;
         weight : integer);
    port (pixel  : in integer;
      insum  : in integer;
      outsum : out integer;
      clk    : in bit);
  end component;

  signal X : integer;

begin

  reg1: synch_reg
```

```
    port map (d => Xin, q => X, clk => clk);

  reg2: synch_reg
    port map (d => X, q => Xout, clk => clk);

  conv: convolver
    generic map (Tpd => 1 ns, weight => 2)
    port map (pixel => Xin, insum => Yin, outsum => Yout, clk => clk);

end structure;


-------------------------------------------------------------------------
-- Test Bench: systolic array
-------------------------------------------------------------------------


entity syst_array_test is
end syst_array_test;


architecture structure of syst_array_test is

  component clock_gen
    port (phi : out bit);
  end component;

  component syst_cell
    port (Xin  : in integer;
      Xout : out integer;
      Yin  : in integer;
      Yout : out integer;
      clk  : in bit);
  end component;

  constant ARRAYLENGTH : integer := 3;
  signal Xin, Yin : integer;
  type int_array is array (0 to ARRAYLENGTH-1) of integer;
  signal Xout, Yout : int_array;
  signal clk: bit;

begin

  cg : clock_gen
    port map (phi => clk);


  cell0: syst_cell
    port map (Xin => Xin, Xout => Xout(0),
          Yin => Yin, Yout => Yout(0),
          clk => clk);

  cell_array: for i in 1 to ARRAYLENGTH-1 generate
      cell: syst_cell
        port map (Xin => Xout(i-1), Xout => Xout(i),
          Yin => Yout(i-1), Yout => Yout(i),
          clk => clk);
  end generate cell_array;

  Xin <= 3, 7 after  16 ns,  2 after  32 ns,  6 after  48 ns,
            3 after  64 ns,  5 after  80 ns,  9 after  96 ns,
            8 after 112 ns,  9 after 128 ns, 10 after 144 ns,
           11 after 160 ns, 12 after 176 ns, 13 after 192 ns,
           14 after 208 ns, 15 after 224 ns, 16 after 240 ns;


--   Xin <= 1, 2 after 10 ns, 3 after 20 ns, 4 after 30 ns, 5 after 40 ns,
--              6 after 50 ns, 7 after 60 ns, 8 after 70 ns, 9 after 80 ns,
```

```
--            10 after 90 ns;

  Yin <= 0;

end structure;


configuration syst_array_structure_test of syst_array_test is

  for structure

      for cg : clock_gen
          use entity work.clock_gen(behaviour)
          generic map (Tpw => 8 ns);
      end for;

      for all : syst_cell
          use entity work.syst_cell(structure);
          for structure
              for all : synch_reg
                  use entity work.synch_reg(behaviour);
              end for;
              for conv : convolver
                  use entity work.convolver(behaviour);
              end for;
          end for;
      end for;

  end for;

end;
```

```
--------------------------------------------------------------------------
--  TOPLEVEL.VHDL
--
--  This is a toplevel simulation of a linear systolic convolution array.
--  Inputs and partial results move systolicly in the same direction;
--  however, THE INPUTS MOVE FASTER THAN THE PARTIAL RESULTS.
--  The convolution weights do not move.
--
--  The test bench below automatically generates an array of length
--  ARRAYLENGTH and feeds it with a stream of input pixels.
--  2*(ARRAYLENGTH-1)+3 clock edges after an input sample enters the
--  first cell, a valid convolution sum of that sample and the previous
--  ARRAYLENGTH samples appears at the output of the last cell.
--------------------------------------------------------------------------


--------------------------------------------------------------------------
-- Clock Generator
--------------------------------------------------------------------------

entity clock_gen is
  generic (Tpw : time := 5 ns);   -- default clock pulse width
  port (phi : out bit);           -- one phase
end clock_gen;


architecture behaviour of clock_gen is
  constant CLOCK_PERIOD : time := 2*(Tpw);
begin

  clock_driver: process
  begin
    phi <= '0', '1' after Tpw;
    wait for CLOCK_PERIOD;
  end process clock_driver;

end behaviour;


--------------------------------------------------------------------------
-- Synchronous Register
--------------------------------------------------------------------------

entity synch_reg is
  generic (Tpd : time := 1 ns);       -- default propagation delay
  port (d : in integer;
    q : out integer := 0;         -- default register content
    clk : in bit);
end synch_reg;


architecture behaviour of synch_reg is
begin

  process
  begin
    wait until clk = '1';
    q <= d after Tpd;
  end process;

end behaviour;


--------------------------------------------------------------------------
-- Multiplication Stage
--------------------------------------------------------------------------
```

```
entity mult_stage is
  generic (Tpd : time := 1 ns;        -- default propagation delay
       weight : integer := 1);    -- default convolution weight
  port (x  : in integer;
    xw : out integer := 0;
    clk: in bit);
end mult_stage;


architecture behaviour of mult_stage is
begin

  process
  begin
    wait until clk = '1';
    xw <= x * weight after Tpd;
  end process;

end behaviour;

-------------------------------------------------------------------------
-- Addition Stage
-------------------------------------------------------------------------

entity add_stage is
  generic (Tpd : time := 1 ns);       -- default propagation delay
  port (xw     : in integer;
        insum  : in integer;
    outsum : out integer := 0;
    clk    : in bit);
end add_stage;


architecture behaviour of add_stage is
begin

  process
  begin
    wait until clk = '1';
    outsum <= insum + xw after Tpd;
  end process;

end behaviour;


-------------------------------------------------------------------------
-- Normalization Stage
-------------------------------------------------------------------------

entity norm_stage is
  generic (Tpd  : time := 1 ns);      -- default propagation delay
  port (innorm  : in integer;
    outnorm : out integer := 0;
    clk     : in bit);
end norm_stage;


architecture behaviour of norm_stage is
begin

  process
  begin
    wait until clk = '1';
    outnorm <= innorm after Tpd;
  end process;

end behaviour;
```

```
-------------------------------------------------------------------------
-- Systolic Cell
-------------------------------------------------------------------------

entity syst_cell is
  port (Xin  : in integer;
    Xout : out integer;
    Yin  : in integer;
    Yout : out integer;
    clk  : in bit);
end syst_cell;


architecture structure of syst_cell is

  component synch_reg
    port (d   : in integer;
      q   : out integer;
      clk : in bit);
  end component;

  component mult_stage
    generic (Tpd : time;
      weight : integer);
    port (x      : in integer;
      xw     : out integer;
      clk    : in bit);
  end component;

  component add_stage
    port (xw     : in integer;
          insum  : in integer;
      outsum : out integer;
      clk    : in bit);
  end component;

  component norm_stage
    port (innorm  : in integer;
          outnorm : out integer;
      clk     : in bit);
  end component;


  signal X, X1, Y1, Y2, Y3 : integer;


begin

  xreg1: synch_reg
    port map (d => Xin, q => X, clk => clk);

  xreg2: synch_reg
    port map (d => X, q => Xout, clk => clk);

  stage1: mult_stage
    generic map (Tpd => 1 ns, weight => 2)
    port map (x => Xin, xw => X1, clk => clk);

  yregin: synch_reg
    port map (d => Yin, q => Y1, clk => clk);

  stage2: add_stage
    port map (xw => X1, insum => Y1, outsum => Y2, clk => clk);

  stage3: norm_stage
    port map (innorm => Y2, outnorm => Y3, clk => clk);
```

```
  Yout <= Y3;

-- yregout: synch_reg
-- port map (d => Y3, q => Yout, clk => clk);


end structure;


--------------------------------------------------------------------------
-- Test Bench: systolic array
--------------------------------------------------------------------------


entity syst_array_test is
end syst_array_test;


architecture structure of syst_array_test is

  component clock_gen
    port (phi : out bit);
  end component;

  component syst_cell
    port (Xin  : in integer;
      Xout : out integer;
      Yin  : in integer;
      Yout : out integer;
      clk  : in bit);
  end component;

  constant ARRAYLENGTH : integer := 3;
  signal Xin, Yin : integer;
  type int_array is array (0 to ARRAYLENGTH-1) of integer;
  signal Xout, Yout : int_array;
  signal clk: bit;

begin

  cg : clock_gen
    port map (phi => clk);


  cell0: syst_cell
    port map (Xin => Xin, Xout => Xout(0),
          Yin => Yin, Yout => Yout(0),
          clk => clk);

  cell_array: for i in 1 to ARRAYLENGTH-1 generate
      cell: syst_cell
        port map (Xin => Xout(i-1), Xout => Xout(i),
          Yin => Yout(i-1), Yout => Yout(i),
          clk => clk);
  end generate cell_array;


  Xin <= 3, 7 after  16 ns,  2 after  32 ns,  6 after  48 ns,
            3 after  64 ns,  5 after  80 ns,  9 after  96 ns,
            8 after 112 ns,  9 after 128 ns, 10 after 144 ns,
           11 after 160 ns, 12 after 176 ns, 13 after 192 ns,
           14 after 208 ns, 15 after 224 ns, 16 after 240 ns;

--  Xin <= 1, 2 after 10 ns, 3 after 20 ns, 4 after 30 ns, 5 after 40 ns,
--            6 after 50 ns, 7 after 60 ns, 8 after 70 ns, 9 after 80 ns,
--           10 after 90 ns;

  Yin <= 0;
```

```
end structure;


configuration syst_array_structure_test of syst_array_test is

  for structure

      for cg : clock_gen
          use entity work.clock_gen(behaviour)
          generic map (Tpw => 8 ns);
      end for;

      for all : syst_cell
          use entity work.syst_cell(structure);
          for structure
              for all : synch_reg
                  use entity work.synch_reg(behaviour);
              end for;
              for stage1 : mult_stage
                  use entity work.mult_stage(behaviour);
              end for;
              for stage2 : add_stage
                  use entity work.add_stage(behaviour);
              end for;
              for stage3 : norm_stage
                  use entity work.norm_stage(behaviour);
              end for;
          end for;
      end for;

  end for;

end;
```

Appendix E VLSI Design Life Cycle Activity

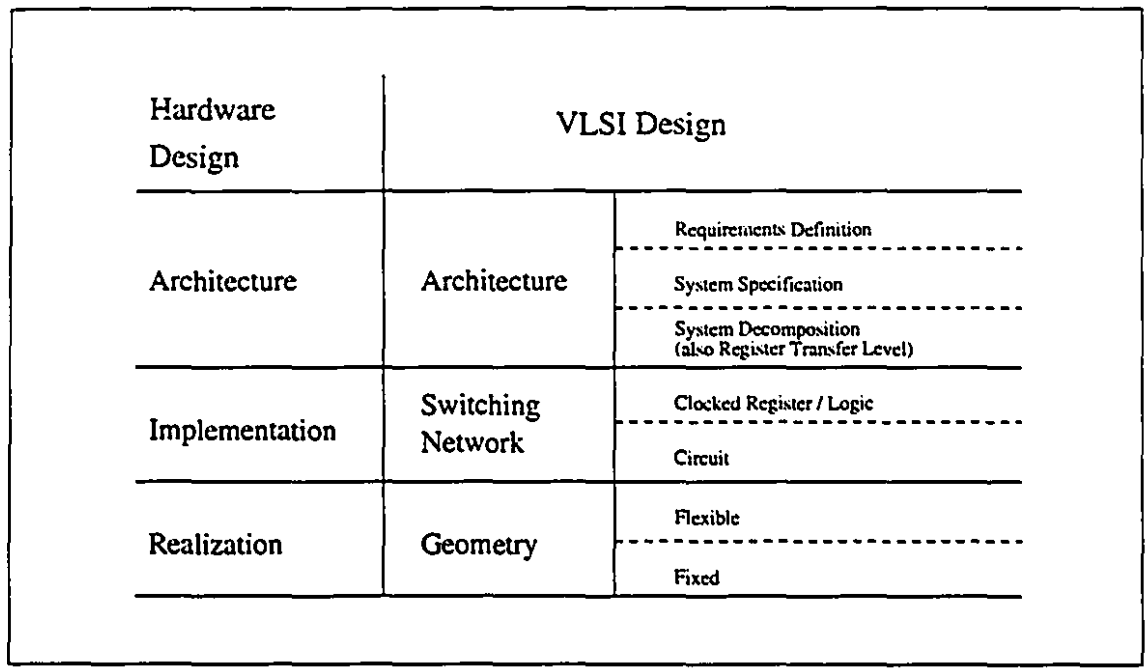

Figure E.1: Levels of abstraction in VLSI design.

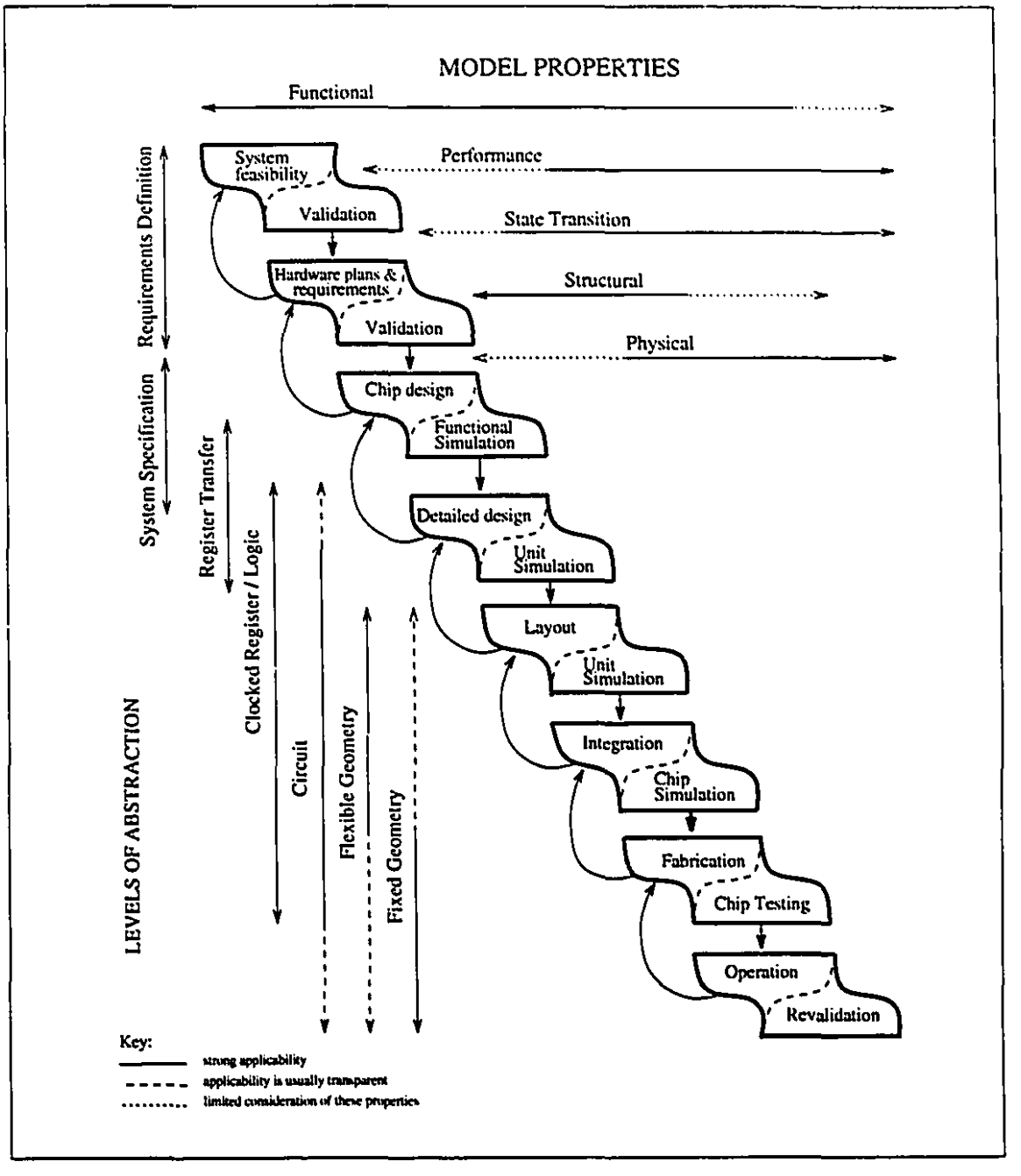

Figure E.2: VLSI design life cycle activity.

Appendix F Design Block Hierarchy

```
;; PrintHierarchy.skill
;;
;; This is my recursive SKILL routine that prints the complete block
;; hierarchy starting from the top level block "fullcircuit2".
;; It traverses the schematic representations in a depth-first
;; manner printing out the current block hierarchy in a format similar
;; to UNIX directory paths.  For brevity, only instance masters are
;; printed (once).
;;
;; Author: Anthony Botzas
;; Date  : May 19th, 1993


;; global
dontExpandList = '("ipin" "opin" "gnd" "vdd" "nmos" "pmos"
                   "inv" "patch" "padout" "buffer"
                   "nor2" "nor3" "nor4"
                   "or2" "or3" "or4"
                   "nand2" "nand3" "nand4"
                   "and2" "and3" "and4"
                  )

procedure( PrintHierarchy()
   prog( (rep outPort blockList)

      outPort=outfile("Hierarchy")

      representation = nil
      while( representation == nil
         representation = dbOpen( "fullcircuit2 schematic current" )
         if( representation == nil then
            fprintf( outPort "Cannot open rep - %s\n" dbErrorIS () )
         )
      )
      currentHierarchyofBlocks = ncons(nil)

      fprintInstanceMasters( outPort representation currentHierarchyofBlocks )
      close( outPort )
   )
)

procedure( fprintInstanceMasters( outfile rep blockStack )
   prog( (revBlockStack instmast i)

      blockStack = cons( rep~>blockName blockStack )            ; push block

      revBlockStack = reverse( blockStack )
      for( i 1 length(revBlockStack)-1
         fprintf( outfile "%s/" nth(i revBlockStack) )
      )
      fprintf( outfile "\n" )


      foreach( instmast rep~>instanceMasters
         fprintf( outfile "%s  " instmast~>blockName )
      )
```

```
      fprintf( outfile "\n\n" )

      foreach( instmast rep~>instanceMasters
         rep = dbOpen( strcat( instmast~>blockName " schematic current" ))
         if( ( (rep != nil) &&
               (!member( rep~>blockName dontExpandList) ) ) then
            dontExpandList = cons( rep~>blockName dontExpandList )
            fprintInstanceMasters( outfile rep blockStack )
         )
      )
      blockStack = cdr(blockStack)                              ; pop block
   )
)

/*

---Output---

Format:  Block hierarchy/
           List of instance masters in block.

fullcircuit2/
   padout      stage2_exp  stage3_exp  X_reg_28x4  Yout_exp
   Yout_man    stage3_man  patch       inv         stage2_man
   Yin_man     Yin_exp     buffer      ipin        control
   stage1_exp  stage1_man  opin

fullcircuit2/stage2_exp/
   nor2        buffer      stage2_load  opin       s2_dec_ctrl
   inv         ipin        decline8     11_comp    stage2_exp_v
   stage2_exp_f

fullcircuit2/stage2_exp/stage2_load/
   buffer      11_comp     nor2         inv        opin
   nor3        ipin

fullcircuit2/stage2_exp/stage2_load/11_comp/
   Speedcomp   ipin        opin         Speedcomp3  Comp

fullcircuit2/stage2_exp/stage2_load/11_comp/Speedcomp/
   and4        ipin          inv        gnd        nmos
   pmos        vdd           opin

fullcircuit2/stage2_exp/stage2_load/11_comp/Speedcomp3/
   and3        ipin        inv          gnd        nmos
   pmos        vdd         opin

fullcircuit2/stage2_exp/stage2_load/11_comp/Comp/
   xnor2       opin        ipin         inv        nor2

fullcircuit2/stage2_exp/s2_dec_ctrl/
   buffer      nor3        nand3        nand2      opin
   ff_1        inv         nor2         ipin

fullcircuit2/stage2_exp/s2_dec_ctrl/ff_1/
   opin        ipin        inv          ff_D       inv_pass

fullcircuit2/stage2_exp/s2_dec_ctrl/ff_1/ff_D/
   inv         gnd         vdd          nmos       pmos
   opin        ipin

fullcircuit2/stage2_exp/s2_dec_ctrl/ff_1/inv_pass/
   opin        ipin        gnd          vdd        nmos
   pmos

fullcircuit2/stage2_exp/decline8/
   buffer      11_adder_2    opin         ipin       vdd
   patch
```

```
fullcircuit2/stage2_exp/declinc8/11_adder_2/
   ipin        opin          pmos          nmos          inv
   gnd         vdd           carryred      adder_red     adder
   4bitadder

fullcircuit2/stage2_exp/declinc8/11_adder_2/carryred/
   vdd         gnd           inv           opin          nmos
   ipin        pmos

fullcircuit2/stage2_exp/declinc8/11_adder_2/adder_red/
   inv         opin          vdd           ipin          gnd
   pmos        nmos

fullcircuit2/stage2_exp/declinc8/11_adder_2/adder/
   nor2        nand2         inv           opin          vdd
   ipin        gnd           pmos          nmos

fullcircuit2/stage2_exp/declinc8/11_adder_2/4bitadder/
   carryfull   opin          ipin          adder

fullcircuit2/stage2_exp/declinc8/11_adder_2/4bitadder/carryfull/
   and4        carry         inv           gnd           nmos
   pmos        vdd           ipin          opin

fullcircuit2/stage2_exp/declinc8/11_adder_2/4bitadder/carryfull/carry/
   inv         gnd           vdd           opin          ipin
   pmos        nmos

fullcircuit2/stage2_exp/stage2_exp_v/
   opin        ff_3          ipin          buffer

fullcircuit2/stage2_exp/stage2_exp_v/ff_3/
   opin        ipin          inv           inv_pass      ff_D

fullcircuit2/stage2_exp/stage2_exp_f/
   ff_2        nor3          nand4         opin          ipin
   buffer

fullcircuit2/stage2_exp/stage2_exp_f/ff_2/
   opin        ipin          inv           inv_pass      ff_D

fullcircuit2/stage3_exp/
   buffer      inc1_dec8     stage3_ov     opin          ipin
   s3_exp_reg

fullcircuit2/stage3_exp/inc1_dec8/
   11_adder_2  buffer        inv           opin          ipin
   patch       gnd

fullcircuit2/stage3_exp/stage3_ov/
   buffer      ff_1          inv           nand2         nand3
   opin        ipin

fullcircuit2/stage3_exp/s3_exp_reg/
   and2        nor4          nor3          nand3         nand4
   nor2        ff_2          opin          buffer        ipin

fullcircuit2/X_reg_28x4/
   X_line      opin          ipin          buffer

fullcircuit2/X_reg_28x4/X_line/
   opin        ipin          ff_1          buffer

fullcircuit2/Yout_exp/
   opin        nor2          ipin          ff_2          buffer

fullcircuit2/Yout_man/
```

```
   nand2       inv          ff_1        opin        nor2
   ipin        buffer       ff_2

fullcircuit2/stage3_man/
   ipin        s3_man_ctrl  opin        s3_man_reg

fullcircuit2/stage3_man/s3_man_ctrl/
   nor2        or2          opin        inv         ipin
   nor3

fullcircuit2/stage3_man/s3_man_reg/
   ff_4        nand2        nor3        nor4        opin
   ipin        buffer

fullcircuit2/stage3_man/s3_man_reg/ff_4/
   opin        ipin         inv         inv_pass    ff_D

fullcircuit2/stage2_man/
   twoscompl   stage3_sign  54_adder    s2_adder_ctrl  opin
   ipin        s2_man_v     s2_man_f

fullcircuit2/stage2_man/twoscompl/
   buffer      opin         ipin        xor2        vdd
   and3        and2         twoscarry

fullcircuit2/stage2_man/twoscompl/xor2/
   opin        ipin         gnd         nmos        vdd
   pmos

fullcircuit2/stage2_man/twoscompl/twoscarry/
   opin        ipin         and4        and3        and2

fullcircuit2/stage2_man/stage3_sign/
   opin        ipin         nor2        buffer      inv
   ff_1

fullcircuit2/stage2_man/54_adder/
   buffer      carry        carryred    16_adder    4bitadder
   ipin        opin

fullcircuit2/stage2_man/54_adder/16_adder/
   inv         carryfull    patch       4bitadder   opin
   ipin

fullcircuit2/stage2_man/s2_adder_ctrl/
   xnor2       nand2        nor2        or2         inv
   xor2        ipin         opin

fullcircuit2/stage2_man/s2_man_v/
   inv         nor3         ff_4_clear  ff_2        ff_1
   ipin        opin         nor2        buffer

fullcircuit2/stage2_man/s2_man_v/ff_4_clear/
   nor2        opin         ipin        inv         inv_pass
   ff_D

fullcircuit2/stage2_man/s2_man_f/
   inv         opin         xnor2       ipin        buffer
   ff_2

fullcircuit2/Yin_man/
   opin        ipin         buffer      ff_1

fullcircuit2/Yin_exp/
   and3        ff_1         buffer      nor2        nor3
   nor4        opin         ipin

fullcircuit2/control/
```

```
   ff_1p       ipin          buffer      nor3          inv
   opin

fullcircuit2/control/ff_1p/
   opin        ipin          inv         inv_pass      ffp

fullcircuit2/control/ff_1p/ffp/
   inv         gnd           vdd         opin          pmos
   nmos        ipin

fullcircuit2/stage1_exp/
   ov_stage1   ov_stage1-2   X_exp_reg   opin          ipin
   xor2        coef_exp      11_adder

fullcircuit2/stage1_exp/ov_stage1/
   buffer      or3           nor2        nand2         or2
   opin        ipin          nor3        nand3         nand4

fullcircuit2/stage1_exp/ov_stage1-2/
   opin        ipin          buffer      inv           ff_1

fullcircuit2/stage1_exp/X_exp_reg/
   nor2        nand2         buffer      UD-OV         opin
   ff_1        ipin

fullcircuit2/stage1_exp/X_exp_reg/UD-OV/
   opin        ipin          nor4        nor3          nand4
   nand3

fullcircuit2/stage1_exp/coef_exp/
   UD-OV       adder_402     ff_1        opin          ipin
   buffer

fullcircuit2/stage1_exp/coef_exp/adder_402/
   ipin        addercarry    opin

fullcircuit2/stage1_exp/coef_exp/adder_402/addercarry/
   inv         opin          vdd         ipin          gnd
   pmos        nmos

fullcircuit2/stage1_exp/11_adder/
   buffer      inv           gnd         patch         opin
   carryred    ipin          4bitadder

fullcircuit2/stage1_man/
   patch       inv           buffer      opin          ipin
   coef_man    prod_reg      57_adder    prodgen

fullcircuit2/stage1_man/coef_man/
   coef_ctrl   opin          ipin        buffer        ff_1

fullcircuit2/stage1_man/coef_man/coef_ctrl/
   nor2        buffer        opin        inv           ipin

fullcircuit2/stage1_man/prod_reg/
   prod_man    buffer        or2         opin          inv
   ipin

fullcircuit2/stage1_man/prod_reg/prod_man/
   ff_1m       ff_1          buffer      nor2          opin
   inv         ipin

fullcircuit2/stage1_man/prod_reg/prod_man/ff_1m/
   ff_D        inv_pass      and2        opin          inv
   ipin

fullcircuit2/stage1_man/57_adder/
   carry       gnd           carryred    addercarry    16_adder
```

```
   opin        ipin          patch       4bitadder

fullcircuit2/stage1_man/prodgen/
   inv         buffer        ff_1        opin          prodgen_ctrl
   productgen  ipin

fullcircuit2/stage1_man/prodgen/prodgen_ctrl/
   or4         buffer        opin        nand2         nor2
   inv         ipin

fullcircuit2/stage1_man/prodgen/productgen/
   prodslice   patch         opin        ipin

fullcircuit2/stage1_man/prodgen/productgen/prodslice/
   addercarry  patch         opin        inv           vdd
   pmos        inv_pass      ipin

*/
```

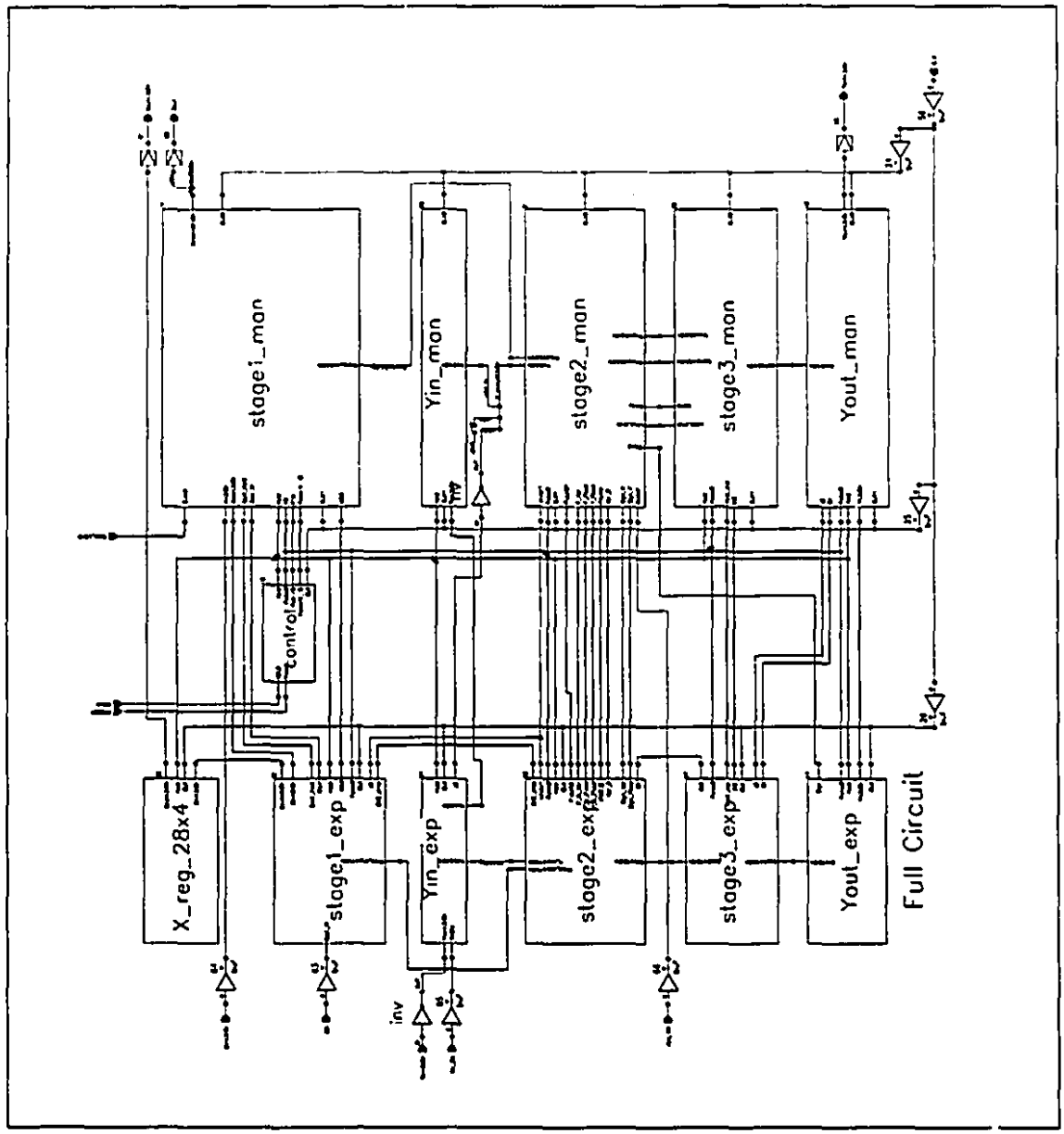

Figure F.1: *fullcircuit2*: Top-level-block schematic representation of systolic cell.

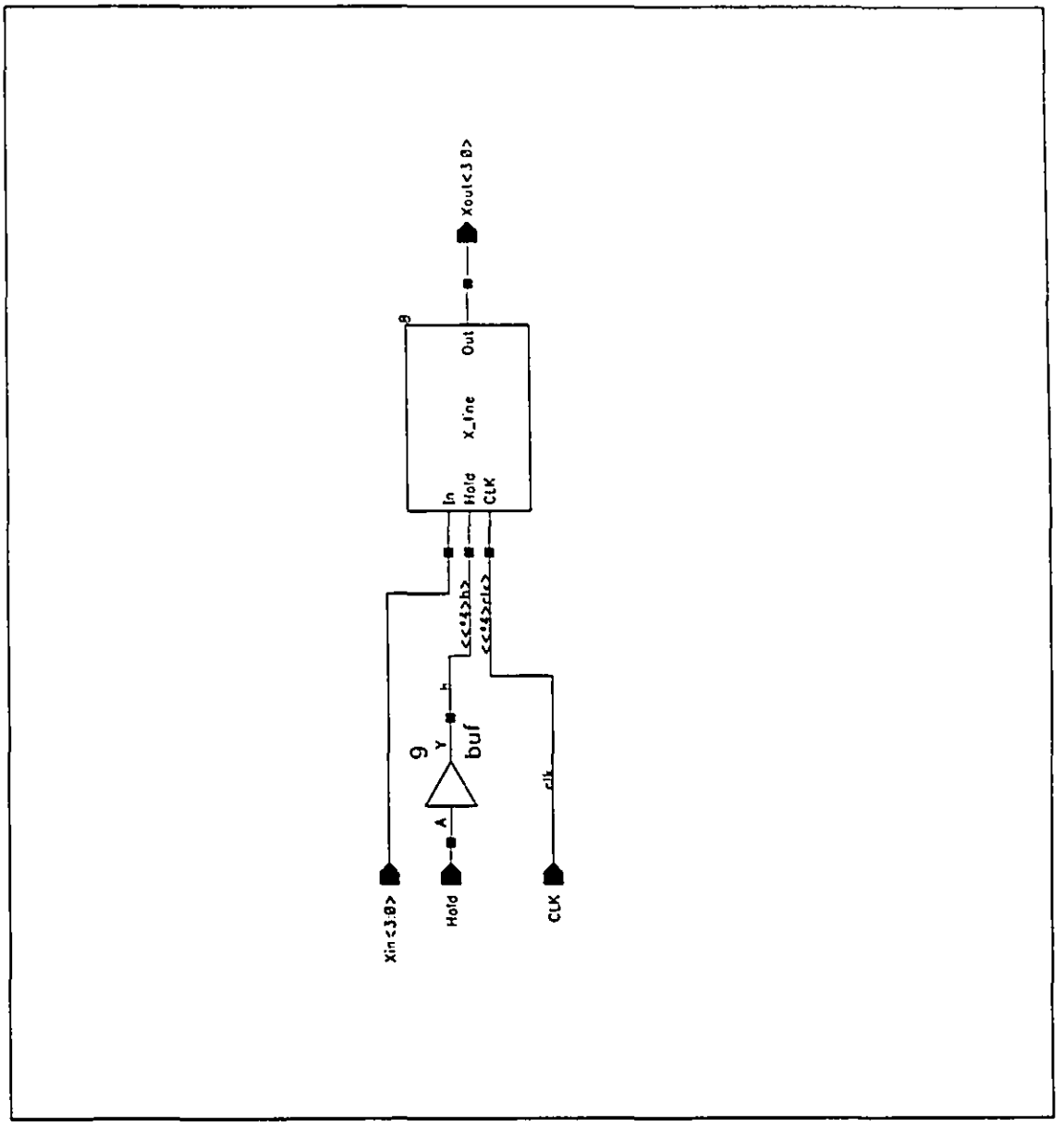

Figure F.2: *fullcircuit2/X_reg_28x4* : 28 × 4 shift register circuit for the X input data.

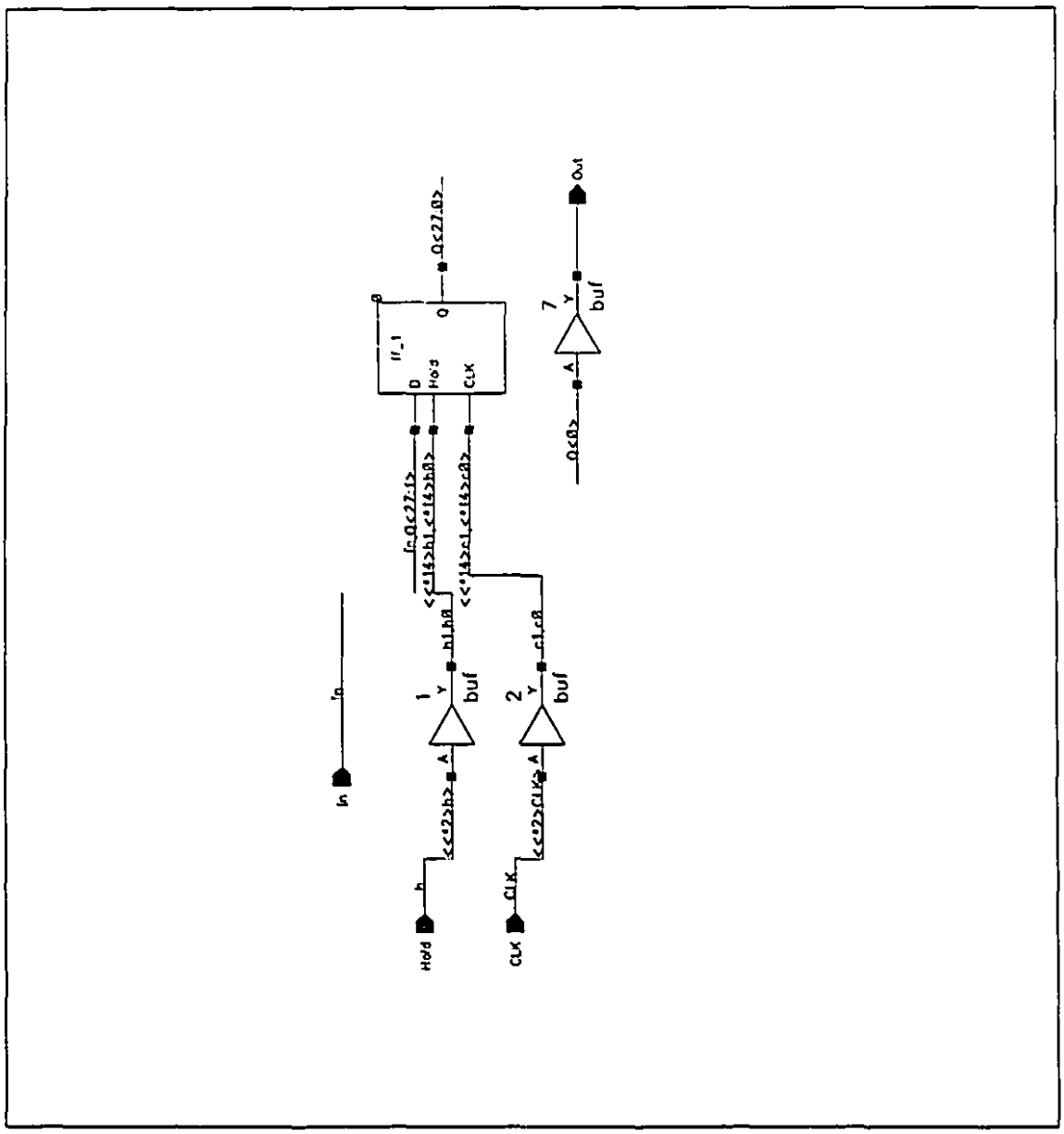

Figure F.3: *fullcircuit2/X_reg_28x4/X_line* : Example of an iterative specification of an array of flip-flops.

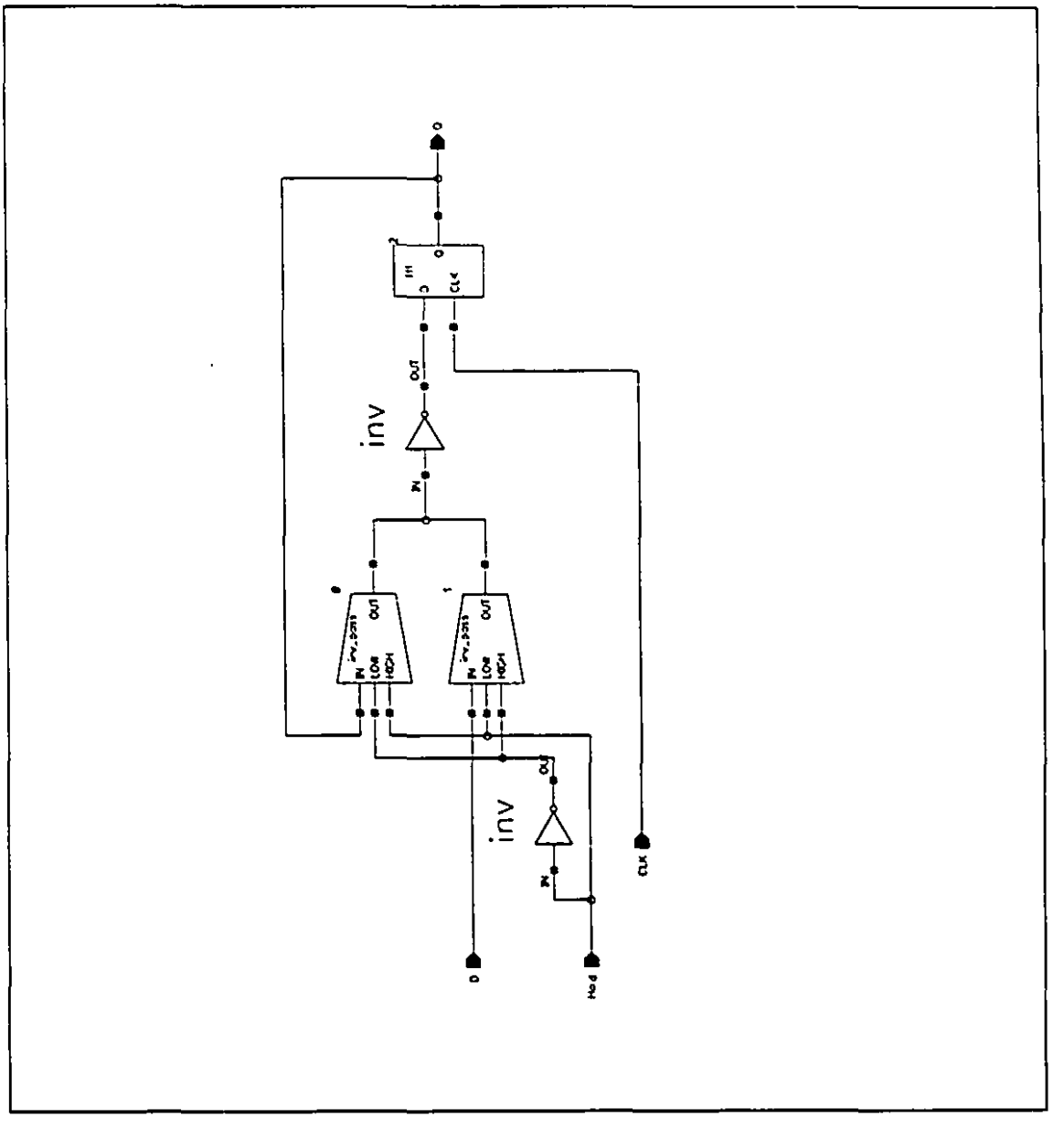

Figure F.4: *fullcircuit2/X_reg_28x4/X_line/ff_1* : Example of a typical delay flip-flop with "hold" capability.

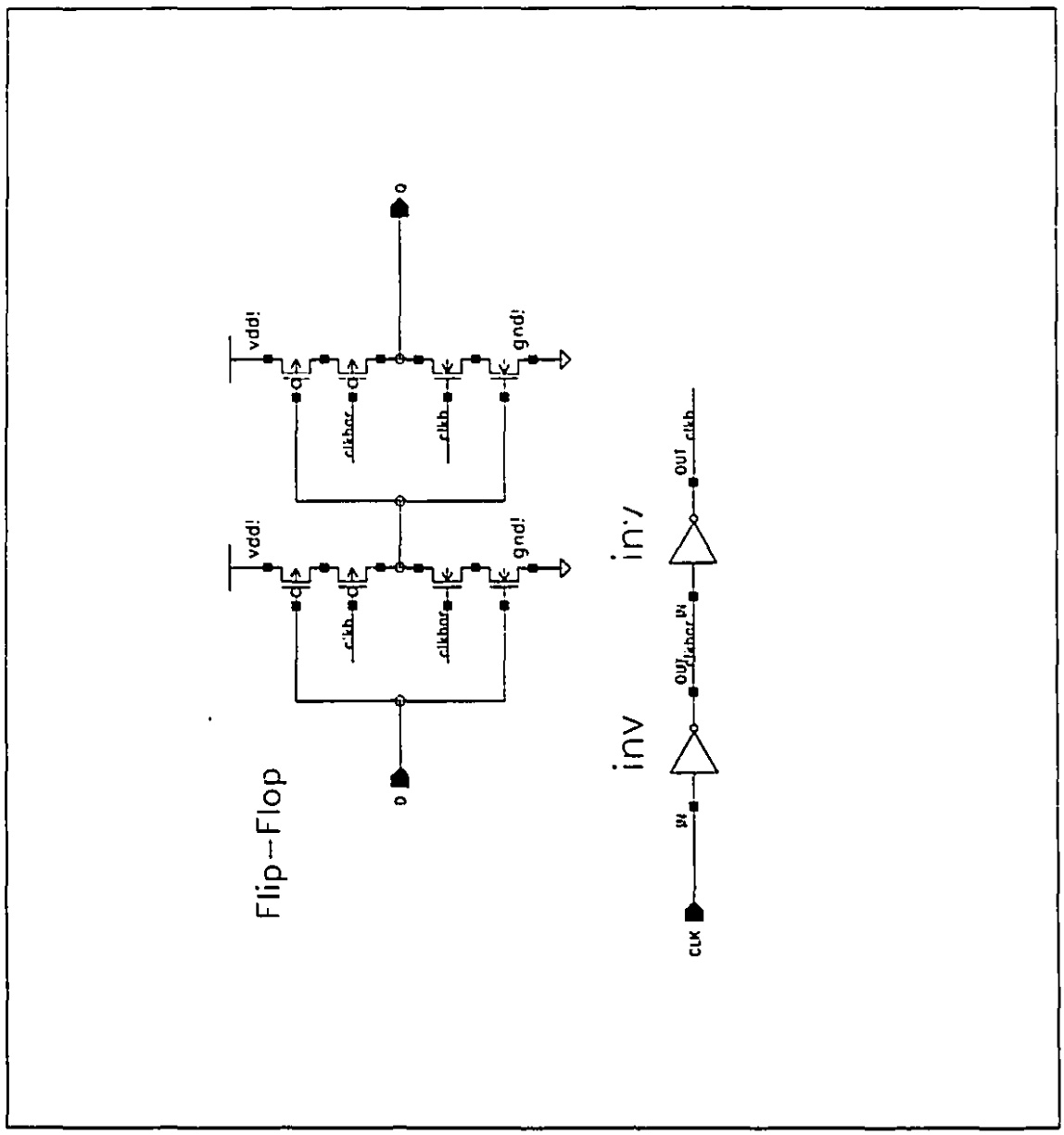

Figure F.5: *fullcircuit2/X_reg_28x4/X_line/ff_1/fff* : Example of a typical one-phase, master-slave, flip-flop implementation.

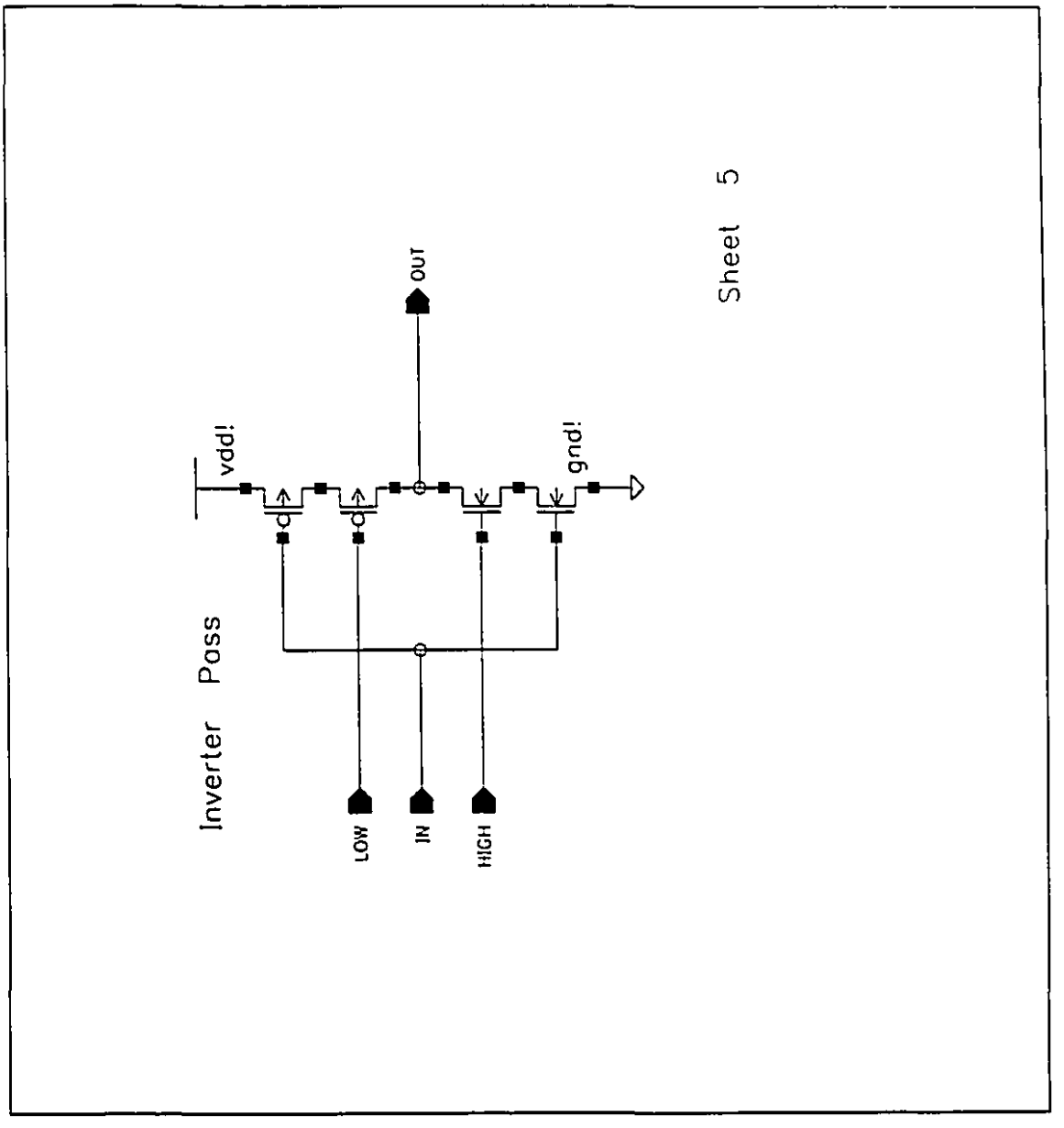

Figure F.6: *fullcircuit2/X_reg_28x4/X_line/ff_1/inv_pass* : Example of an inverting pass gate implementation.

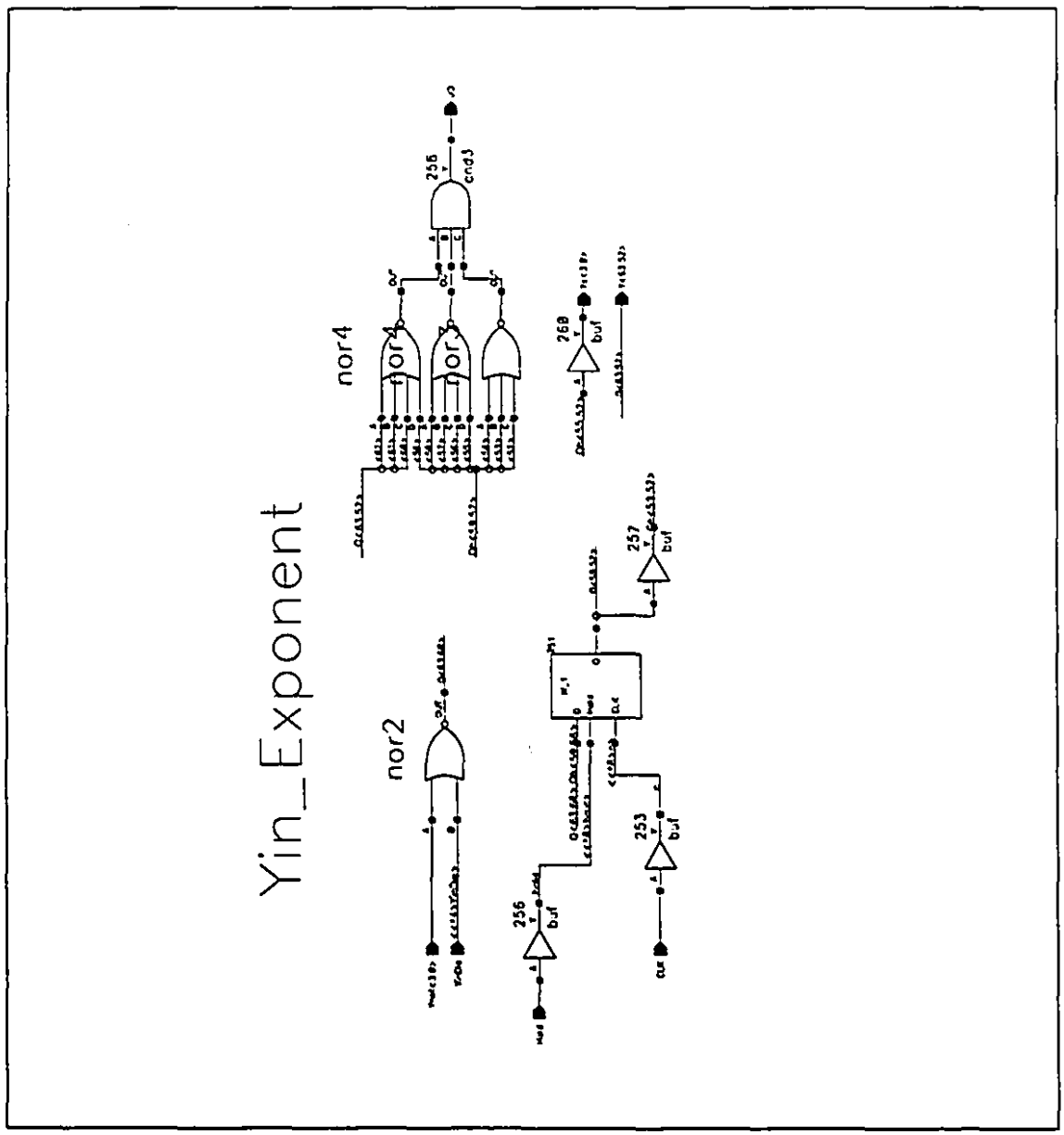

Figure F.7: *fullcircuit2/Yin_exp* : Exponent portion of Y input circuit.

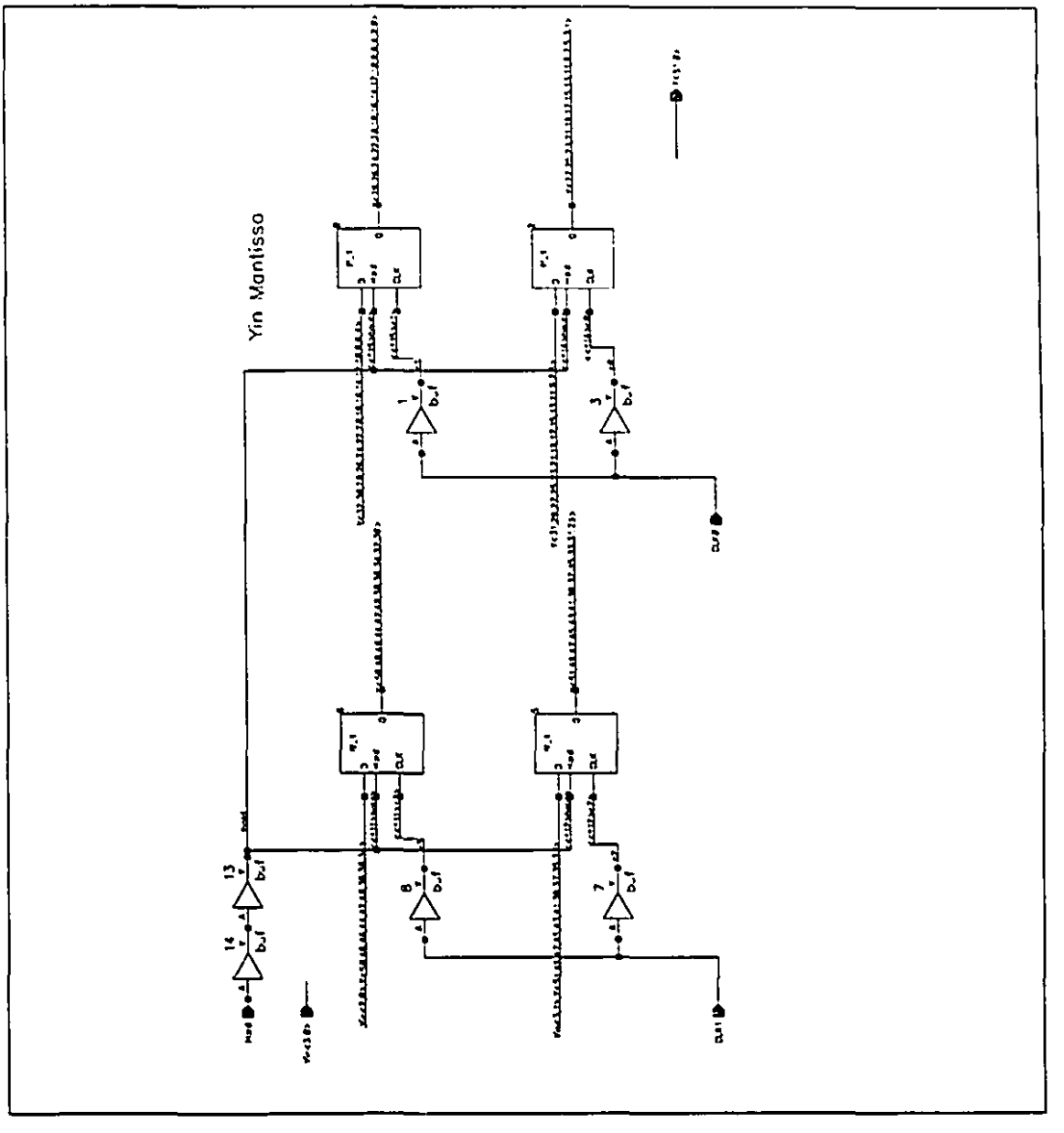

Figure F.8: *fullcircuit2/Yin_man* : Mantissa portion of Y input circuit.

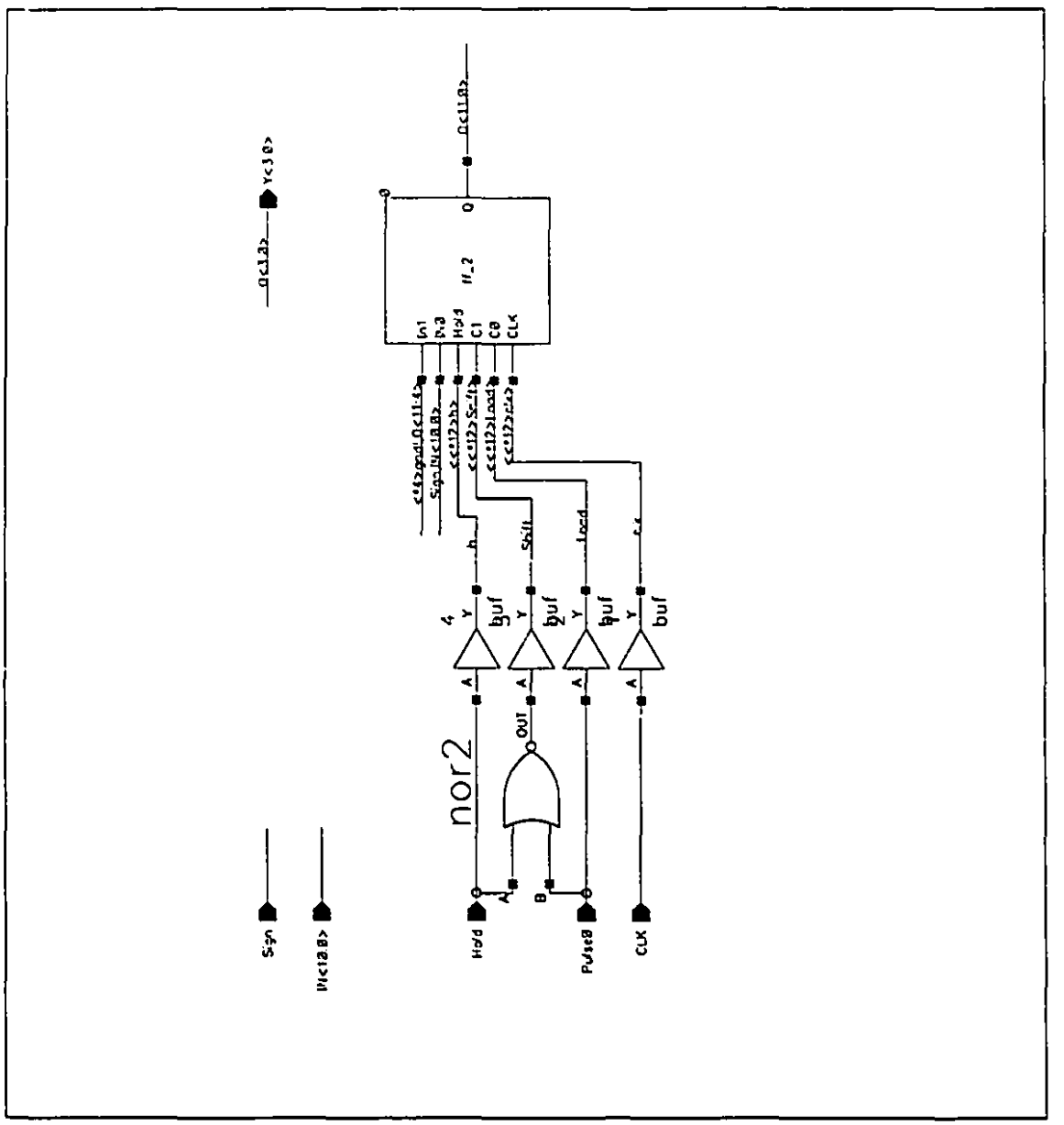

Figure F.9: *fullcircuit2/Yout_exp* : Exponent portion of Y output circuit.

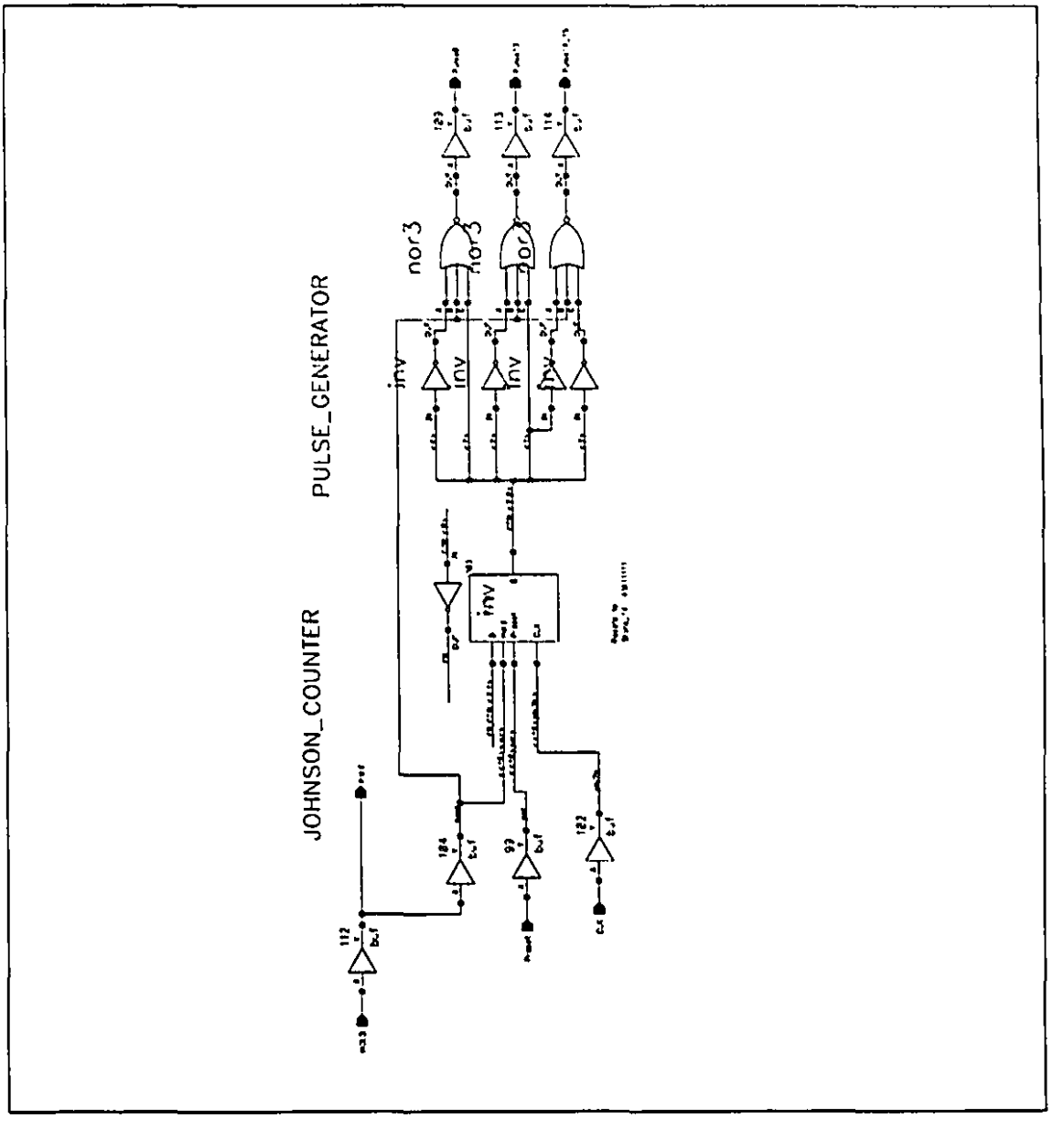

Figure F.10: *fullcircuit2/control* : Global control circuitry.

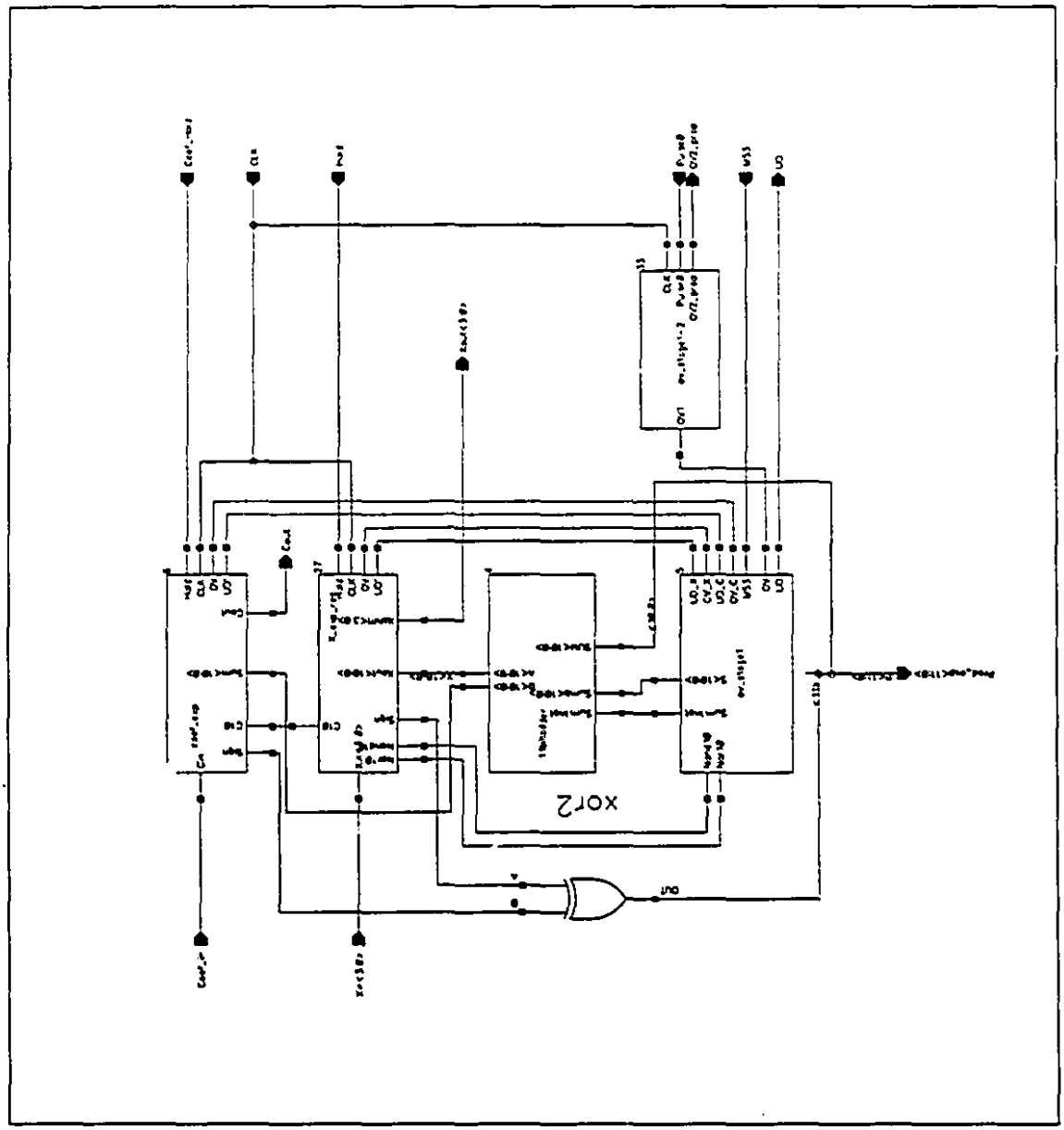

Figure F.11: *fullcircuit2/stage1_exp* : Exponent portion of first stage multiplication circuit.

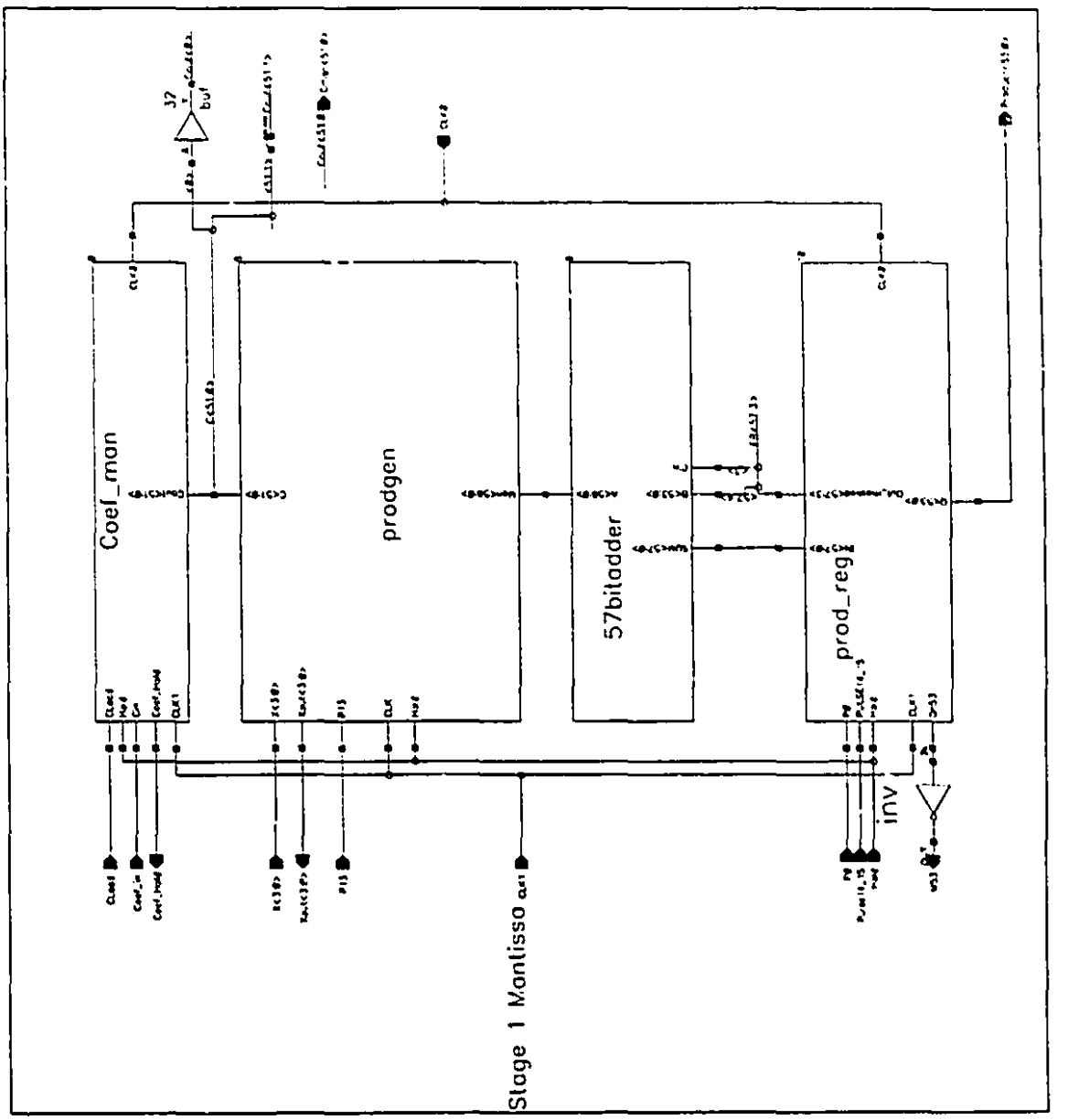

Figure F.12: *fullcircuit2/stage1_man* : Mantissa portion of first stage multiplication circuit.

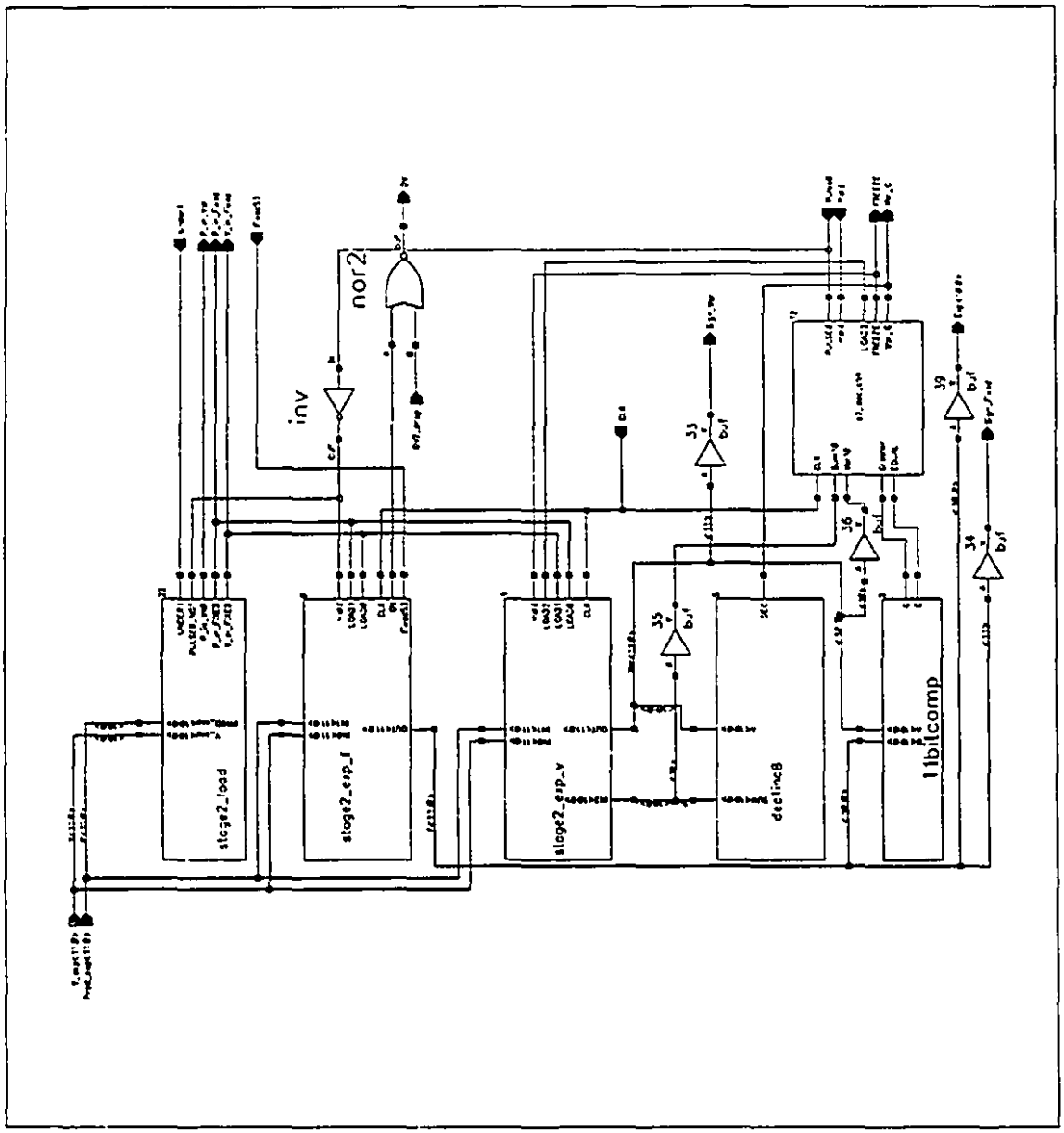

Figure F.13: *fullcircuit2/stage2_exp* : Exponent portion of second stage addition circuit.

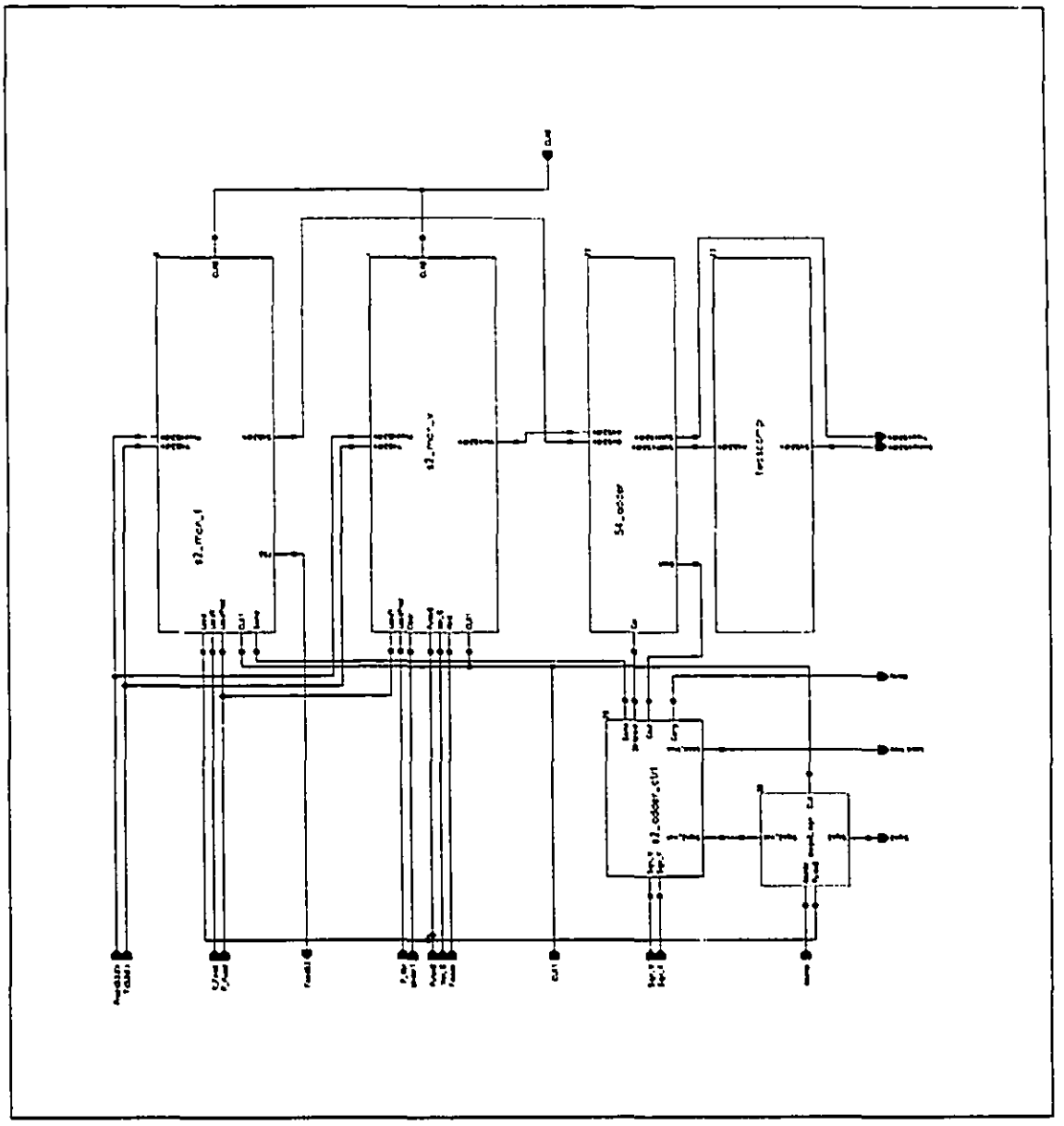

Figure F.14: *fullcircuit2/stage2_man* : Mantissa portion of second stage addition circuit.

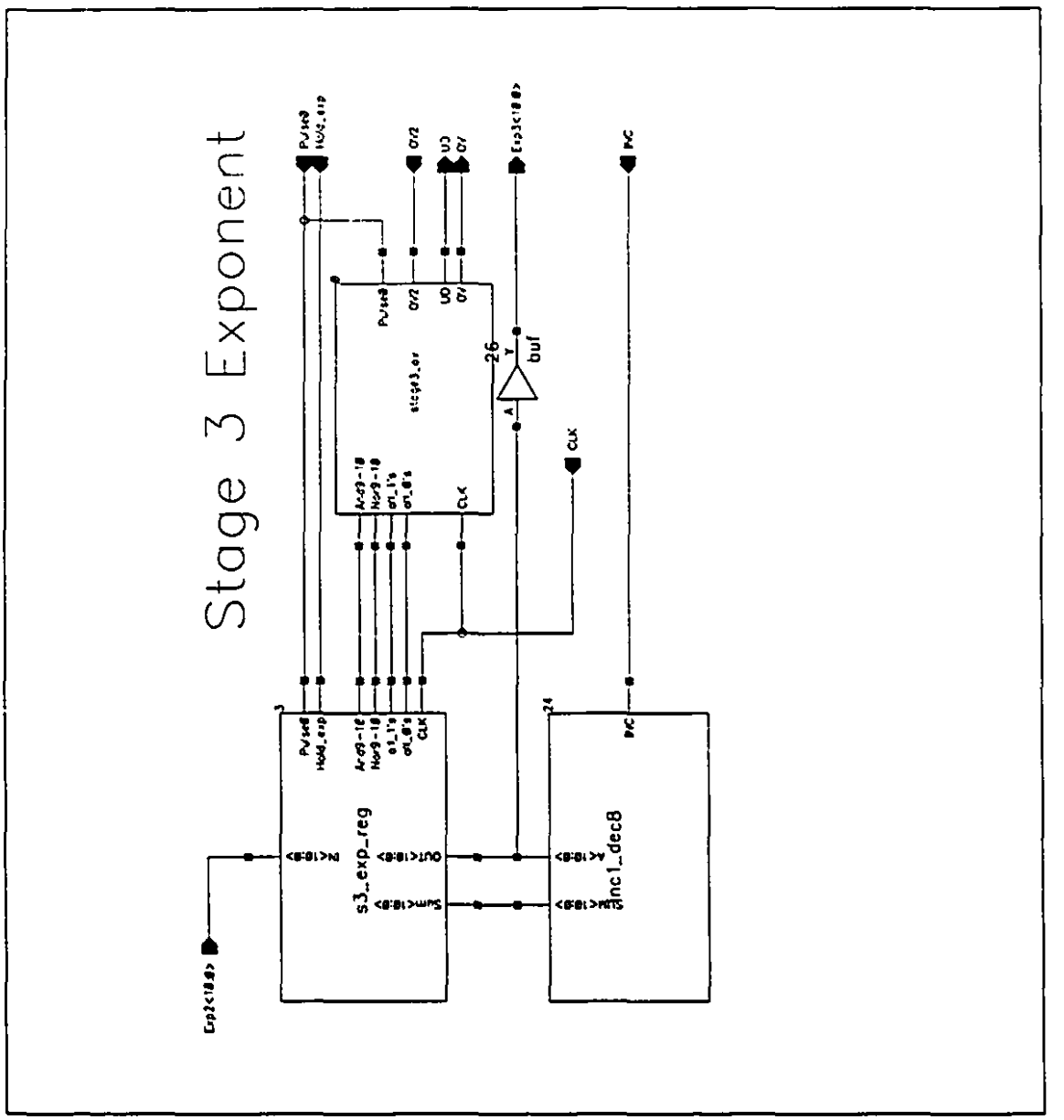

Figure F.15: *fullcircuit2/stage3_exp* : Exponent portion of third stage normalization circuit.

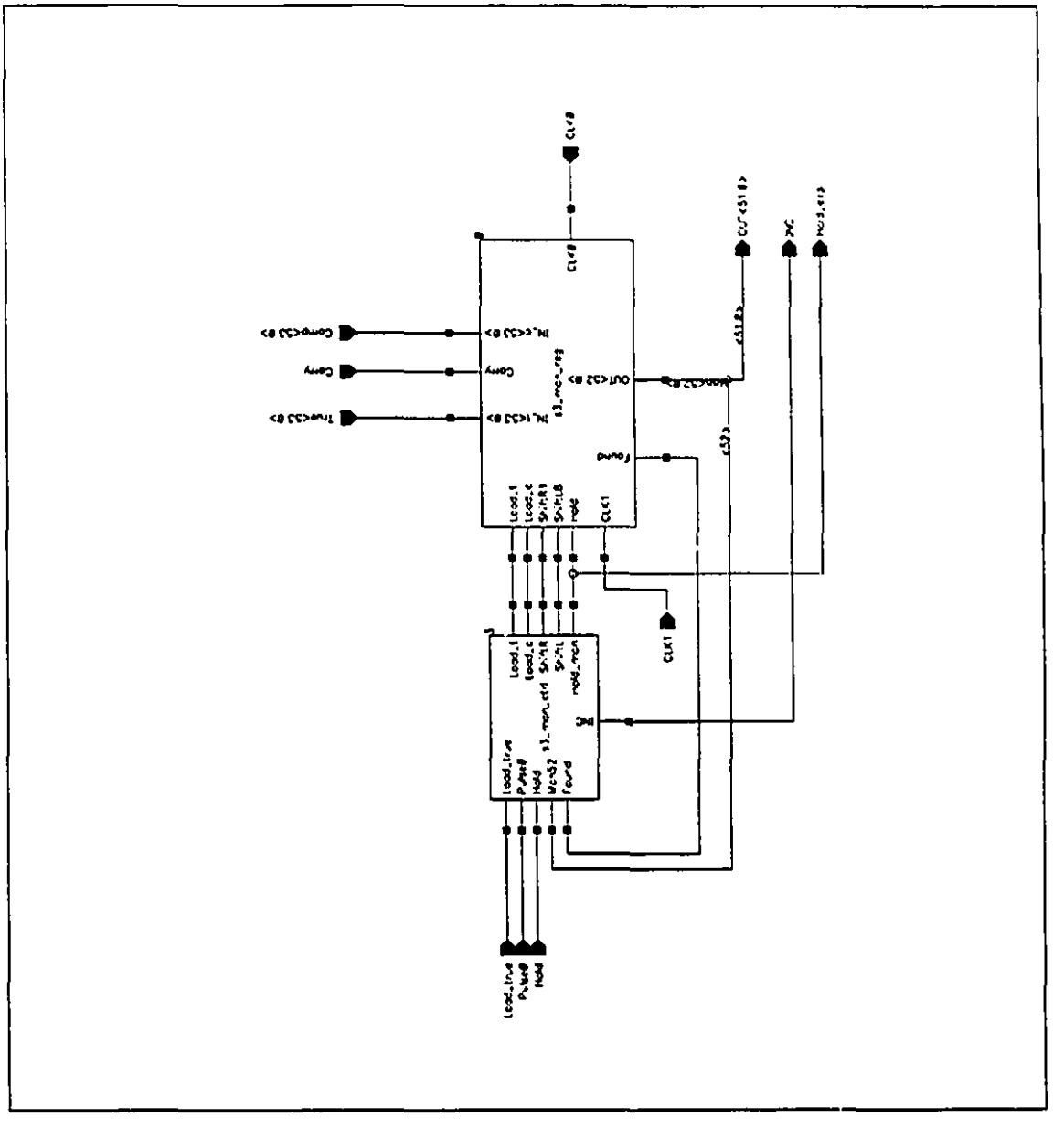

Figure F.16: *fullcircuit2/stage3_man* : Mantissa portion of third stage normalization circuit.

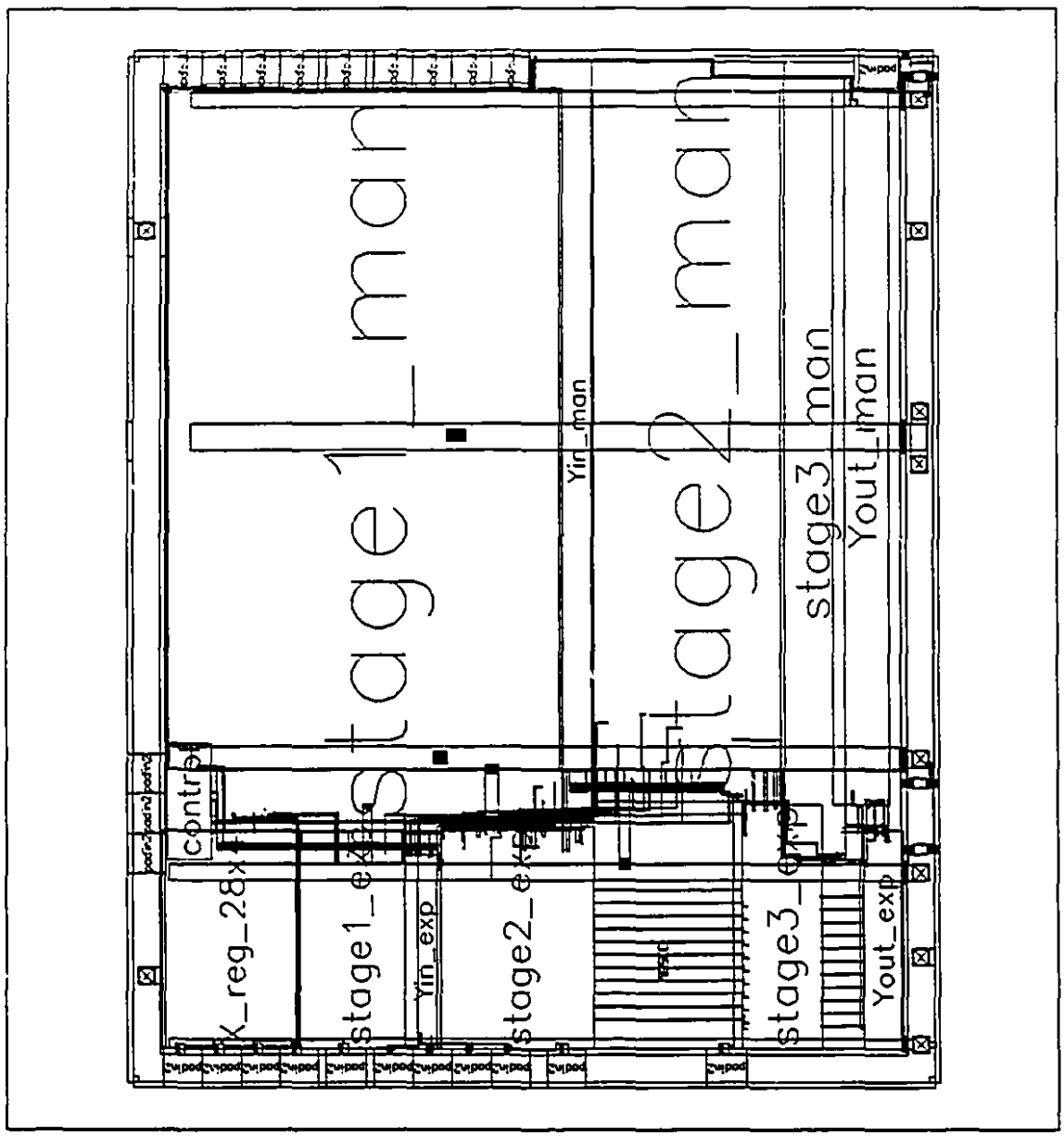

Figure F.17: *fullcircuit2* : Top-level-block layout representation of systolic cell.

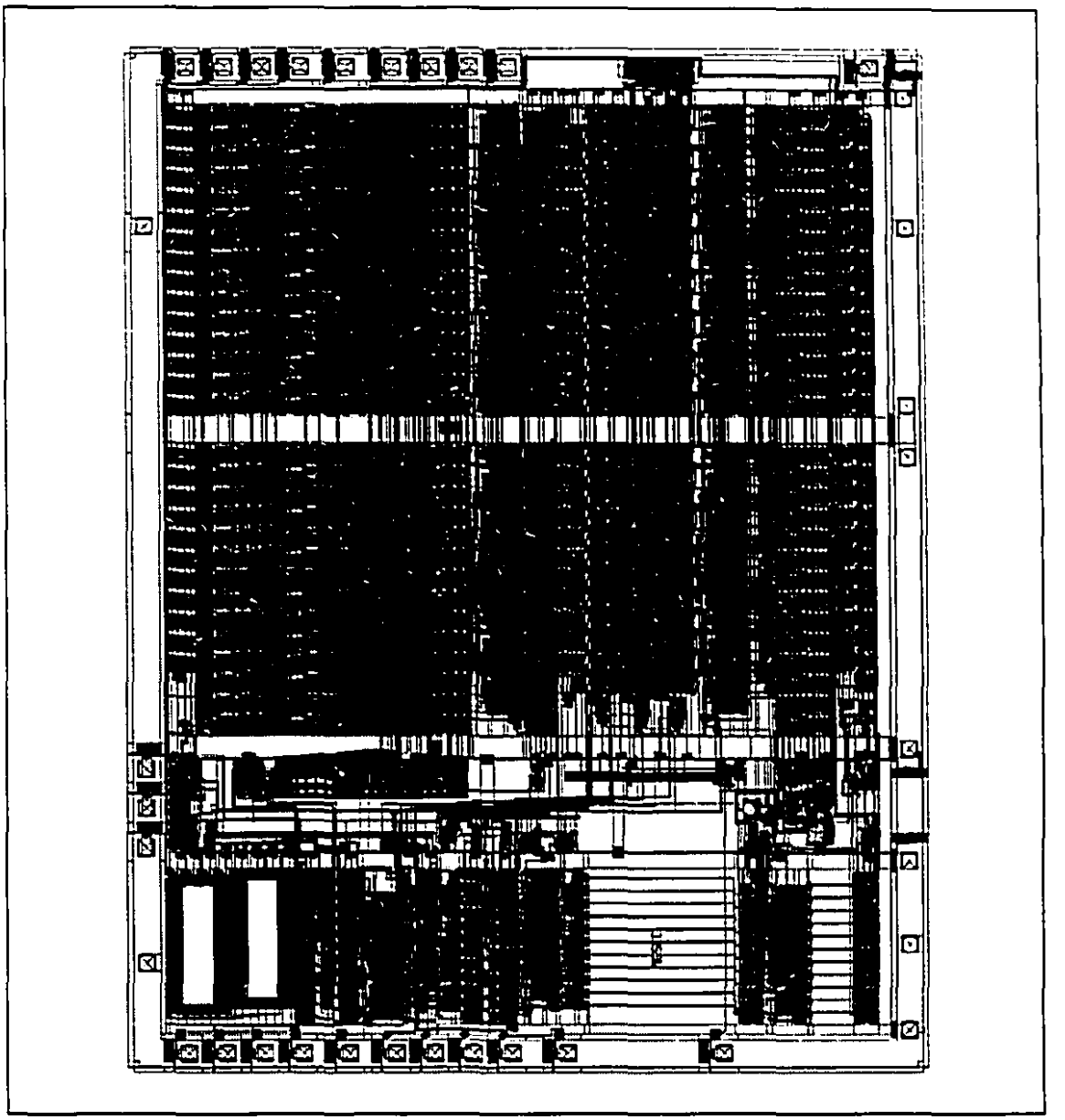

Figure F.18: *fullcircuit2* : Top-level-block "exploded"layout representation of systolic cell.

TEST HEAD CONFIGURATION

DIGITAL SIGNALS

Data Generator

Address	Pin		Socket	Signal	ICPin
0-0	7.		26.	Xin(0)	40
0-1	9.		27.	Xin(1)	39
0-2	12.		28.	Xin(2)	38
0-3	14.		29.	Xin(3)	37
1-0	16.		30.	Yin(0)	45
1-1	18.		31.	Yin(1)	44
1-2	20.		32.	Yin(2)	43
1-3	22.		33.	Yin(3)	42
2-0	25.		6.	PRESET	31
2-1	27.		7.	HOLD	30
2-2	29.		8.	C_LOAD	29
2-3	31.		10.		
4-0	33.		11.	Cin 41	
4-1	35.		12.	Yin_Dis	46
4-2	37.		13.	Abs_Val	49
4-3	40.		14.		
5-0	42.		15.		
5-1	44.		16.		
5-2	46.		17.		
5-3	48.				
6-0	50.				
6-1	52.				
6-2	57.				
6-3	59.				
Clock 1		84. <-- the clock output	5.	CLK 1	
Clock 2		3. <-- the clock output	19.		
Strobe		5. <-- the strobe output			

Data Analyzer

Address	Pin			
0-0	61.	34.	Xout(0)	12
0-1	10.	35.	Xout(1)	13
0-2	64.	36.	Xout(2)	14
0-3	65.	37.	Xout(3)	15
2-0	66.	38.	Xin(0)	40
2-1	67.	39.	Xin(1)	39
2-2	68.	40.	Xin(2)	38
2-3	69.	1.	Xin(3)	37
1-0	24.			

```
1-1     70.
1-2     71.
1-3     72.

3-0     73.
3-1     74.
3-2     75.
3-3     39.
4-0     76.
4-1     77.
4-2     78.
4-3     79.
5-0     80.
5-1     81.
5-2     82.
5-3     83.

Ground          56.                             23.
Ext Clock       60. <-- ext clock input         18. CLK 1

DPS1            55.                             22.
DPS2                                             4. Vdd 32,55,62,63,66
DPS3                                             9.
GND             56.                              2. Vss 21,52,53,57,68
```

```
/*********************************************************************
 *  .8180initrc
 *  -----------
 *
 *  HP 8180A initialization run commands.
 *  These commands are read in by my HP interface program HPINIT.
 *  My parser and syntax checker expects comments as in C language.
 *  It is also wise to delimit commands with semicolons.
 *  Note that the purpose of this file is to load an initial set of
 *  desirable parameters on each "page" of the data GENERATOR.
 *
 *  AUTHOR: ANTHONY BOTZAS
 *********************************************************************/


rss;    /* set generator to a known state */

/*********************************************************************
 *
 *  RSS
 *  ---
 *
 *  Recalls the standard parameter set written by the manufacturer for
 *  the HP8180 Data Generator. "rss" performs the following actions:
 *  [Note: I have put arrows next to the settings which are not
 *   particularly desirable to me.  I plan to reset these explicitly
 *   after the rss command.]
 *
 *  rss = Recall Standard Set
 *
 *  ...CONTROL PAGE...
 *
 *       stp                ; stop
 *
 *       fad 0              ; first address
 *  ---> lad 1023           ; last address
 *
 *       cyml               ; cycle mode auto
 *
 *       stbl               ; strobe breaks off
 *
 *       clkl               ; clock source internal
 *
 *       thr 0v             ; input threshold
 *       impl               ; input impedance 50 ohms
 *       ruil               ; run input off
 *       spil               ; stop input off
 *       bril               ; break input off
 *
 *   stol                ; strobe output as NRZ data channel
 *  ---> stl2               ; strobe level ECL (on output page)
 *
 *   outl                ; outputs off
 *
 *  ...TIMING PAGE...
 *
 *  ---> per 100ns          ; clock period (f=10MegHz)
 *
 *       del 1c 0ns         ; delay clock 1
 *   fmt 1c 1            ; format clock 1 RZ
 *  ---> wid 1c 40ns        ; width clock 1
 *
 *       del 2c 50ns        ; delay clock 2
 *   fmt 2c 2            ; format clock 2 RZ=50%
 *   wid 2c 10ns         ; width clock 2
 *
```

```
*       del 0xx 0ns             ; delay all channels
*    fmt xx 3                 ; format all channels NRZ
*    del yy 0ns               ; delay all extenders
*
*   ...OUTPUT PAGE...
*
*    lim1                     ; load impedance 50 ohms
*
*   ---> hilA 0.25v             ; high level label A
*   ---> lolA -0.25v            ; low level label A
*    hilB 5v                  ; high level label B
*    lolB 0v                  ; low level label B
*    hilC -0.8v               ; high level label C
*    lolC -1.8v               ; low level label C
*    hilD 2.4v                ; high level label D
*    lolD 0.8v                ; low level label D
*
*    lba xx                   ; select label A for all channels
*    nor xx                   ; select normal polarity for all
*                             ; channels
*
*   ...DATA PAGE...
*
*    ssc                      ; set standard configuration
*                                ; on the data page
*
*************************************************************************/


/*  TIMING PAGE */

    /* Clock signal */

    fmt 1c 1;           /* return to zero format */
    per 1us;            /* 1.0 MHz */
    wid 1c 500ns;    /* 500ns pulse width */
    del 1c 0ns;         /* no delay w.r.t. strobe */


    /* Xin(0..3) on Connector 0 */

    fmt 00 3;           /* non-return to zero format */
    del 00 0ns;

    fmt 01 3;
    del 01 0ns;

    fmt 02 3;
    del 02 0ns;

    fmt 03 3;
    del 03 0ns;


    /* Yin(0..3) on Connector 1 */

    fmt 10 3;
    del 10 0ns;

    fmt 11 3;
    del 11 0ns;

    fmt 12 3;
    del 12 0ns;

    fmt 13 3;
    del 13 0ns;
```

```
    /* PRESET, HOLD, and COEF_LOAD on Connector 2 */

    fmt 20 3;
    del 20 0ns;

    fmt 21 3;
    del 21 0ns;

    fmt 22 3;
    del 22 0ns;


    /* Cin, Yin_Dis, and Abs_Val on Connector 4 */

    fmt 40 3;
    del 40 0ns;

    fmt 41 3;
    del 41 0ns;

    fmt 42 3;
    del 42 0ns;


/*  OUTPUT PAGE */

    hilA 5v; lolA 0v;   /* set TTL levels for Label A */

    stll;               /* strobe level TTL */
                        /* (not really necessary) */


/*  DATA PAGE */

    pag 4; cld; cls;

    cas by 00 01 02 03; ads by 10 11 12 13;
    ads by 20 21 22;    ads by 40 41 42;


    fad 0000; lad 0008;
    tsa 0000;


/*                                  PRESET     Cin      *
 * ADDR          STR   DATA         HOLD       Yin_Dis  *
 *                     Xin  Yin     Coef_LOAD  Abs_Val */

/* 0000 */ for   1     0000 0000    000        000;
/* 0001 */ for   1     1000 0000    000        000;
/* 0002 */ for   1     0000 0000    000        000;
/* 0003 */ for   1     1000 0000    000        000;
/* 0004 */ for   1     0000 0000    000        000;
/* 0005 */ for   1     1000 0000    000        000;
/* 0006 */ for   1     0000 0000    000        000;
/* 0007 */ for   1     1000 0000    000        000;
/* 0008 */ for   1     0000 0000    000        000;

out2; /* outputs ON */
run;
```

```
/*********************************************************************
 *  .8182initrc
 *  -----------
 *
 *  HP 8182A initialization run commands.
 *  These commands are read in by my HP interface program HPINIT.
 *  My parser and syntax checker expects comments as in C language.
 *  It is also wise to delimit commands with semicolons.
 *  Note that the purpose of this file is to load an initial set of
 *  desirable parameters on each "page" of the data ANALYZER.
 *
 *  AUTHOR: ANTHONY BOTZAS
 *********************************************************************/

stp:    /* stop operation */

rcl4:   /* recall standard set */

/*********************************************************************
 *
 *  RCL4
 *  ----
 *
 *  Recalls the standard parameter set written by the manufacturer for
 *  the HP8182 Data Alalyzer. "rcl4" performs the following actions:
 *  [Note: I have put arrows next to the settings which are not
 *   particularly desirable to me.  I plan to reset these explicitly
 *   after the rcl4 command.]
 *
 *
 *  ...CONTROL PAGE...
 *
 *      opr 1       ; trigger start analysis
 *          ; for recording post-trigger data
 *  gld 2       ; glitch detect off
 *
 * ---> clk 1       ; clock source EXTERNAL
 *  cks 1       ; clock slope positive
 *  ckt 1.4v    ; clock threshold
 *  ckd 0.00ns  ; clock delay
 *
 *  cql 3       ; clock qualifier level don't care
 *  cqt 1.4v    ; clock qualifier threshold
 *  cqi 2       ; clock qualifier impedance 100Kohms
 *
 *
 *  ckw 10.0ns  ; clock width (for external clock
 *                  ; in TRG STRT COMP MODE)
 *  tas 3       ; trigger arm slope don't care
 *  tat 1.4v    ; trigger arm threshold
 *  tai 2       ; trigger arm impedance 100Kohms
 *
 *  twd XXX...  ; trigger word don't care
 *
 *      tql 3           ; trigger qualifier level don't care
 *      tqt 1.4v        ; trigger qualifier threshold
 *      tqi 2           ; trigger qualifier impedance 100Kohms
 *
 *      trc 01          ; trigger count
 *
 *      agc 2           ; allow gaps in count NO
 *
 *      trd 00000       ; trigger delay
 *
 *      sps 3           ; stop slope internal
 *      spt 1.4v        ; stop threshold
 *      spi 2           ; stop impedance 100Kohms
 * ---> spd 1023        ; stop delay
```

```
*                       ; stop occurs 1023 clock periods after triggering
*      spe 1            ; stop on error OFF (real time compare mode)
*      aad 4            ; autoarming OFF
*      cyp-1            ; cycling period OFF
*
*
*   ...INPUT PAGE...
*
*      c__ 1            ; all installed connecters SINGLE threshold
*                       ; 8182A software interrogates hardware to
*                       ; determine how many connectors are installed
*      si_ 1.40v        ; all labels single threshold value
*      lo_ 0.8v         ; all labels lower threshold value
*      up_ 2.00v        ; all labels upper threshold value
*
*
*      lbl aaa...       ; all channels label A
*
*  ---> cas b ...       ; 4 channels for every installed connector
*
*
*
*   ...EXPECTED DATA PAGE...
*
*      tad 0000         ; top address
*
*
*   ...STATE LIST PAGE...
*
*      dse 1            ; display errors YES (if user switches
*                       ; glitch detect ON)
*
*
*   ...TIMING DIAGRAMS PAGE...
*
*      cad 0000         ; cursor address
*
*      dse 1            ; display errors YES (if user switches
*                       ; glitch detect ON)
*      hoz 1            ; horizontal zoom factor = 1
*      vez 1            ; vertical zoom factor = 1
*
*
*   ...ERROR MAP PAGE...
*
*      cte 2            ; error count OFF
*
*      dsg 1            ; display glitches YES (if user switches
*                       ; glitch detect ON)
*
*
*************************************************************************/


/*  CONTROL PAGE */

    clk 1;          /* clock source EXTERNAL */


/*  INPUT PAGE */       /* these channels have to  */
                        /* appear in reverse order */

    cas B 03 02 01 00;   /* Xout(3..0) */
    ads B 23 22 21 20;   /* Xin(3..0)  */

pag 6;
run;
```