Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A VERY LARGE SCALE INTEGRATION DESIGN OF AN ONLINE ALGORITHM FOR THE COMPUTATION OF ROTATION FACTORS

Stephen George Faris

April 1989
CSD-890015

UNIVERSITY OF CALIFORNIA

Los Angeles

A Very Large Scale Integration Design of an Online Algorithm for the Computation of Rotation Factors

A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Stephen George Faris

1989

© Copyright by

Stephen George Faris

1989

The thesis of Stephen George Faris is approved.

L. P. McNamee

Tomas Lang, Committee Co-Chair

Milos D. Ercegovac, Committee Co-Chair

University of California, Los Angeles

1989

TABLE OF CONTENTS

page

1. Introduction

1.1 The Computation of Rotation Factors 3
1.2 The Integrated Algorithm for Computation of Rotation Factors 5
1.3 An Integrated Architecture for Computation of Rotation Factors 7

2. The Generalized Architecture of an Online Unit

2.1 General Structure of the Online Algorithms 9
2.2 Design of a Generalized Online Unit
2.2.1 Logical Design of a Generalized Online Unit 11
2.2.2 Logical Design of an Operand Section 12
2.2.3 Logical Design of an Arithmetic Section 16
2.2.4 Logical Design of a Selection Section 17

3. High-level Functional Description of the Chip

3.1 Input, Intermediate, and Output Operands 19
3.2 System Signals and their Relationship 21
3.3 Fundamental Design Approach 23
3.4 Fundamental Building Blocks of the Implementation 25

4. Alignment Unit Architecture

4.1 Algorithmic Analysis 27
4.2 Logical and Physical Design of Sub-sections
4.2.1 The Arithmetic Section 28
4.2.2 The Shifting Section 30
4.3 The Resulting Structure 31

5. Sum-of-Squares Unit Architecture

5.1 Algorithmic Analysis 35
5.2 Logical and Physical Design of Sub-sections
5.2.1 The Operand Section 36
5.2.2 The Arithmetic Section 39
5.2.3 The Selection Section 41
5.3 The Resulting Structure 42

6. Square-Root Unit Architecture

6.1 Algorithmic Analysis 46
6.2 Logical and Physical Design of Sub-sections
6.2.1 The Operand Section 48
6.2.2 The Arithmetic Section 53
6.2.3 The Selection Section 55

6.3 The Resulting Structure....57

7. Division Unit Architecture

7.1 Algorithmic Analysis....62
7.2 Logical and Physical Design of Sub-sections
7.2.1 The Operand Section....63
7.2.2 The Arithmetic Section....66
7.2.3 The Selection Section....69
7.3 The Resulting Structure....71

8. Architectural, Performance, and Functional Evaluation

8.1 Chip Floorplan and Area Characteristics....75
8.2 Timing Analysis: Online Delay and Cycle Time....81
8.3 Operational Testing Mechanisms....84

LIST OF FIGURES

page

Figure 1.1 - Input Digit Consumption/Output Digit Production: Single Unit............... 2
Figure 1.2 - Input Digit Consumption/Output Digit Production: Multiple Units 2
Figure 1.3 - Operand and Result Number Formats... 4
Figure 1.4 - Algorithmic Breakdown into Sub-algorithms and Assignment to Units 6
Figure 2.1 - General Structure of the Online Algorithms 9
Figure 2.2 - Logical Design of a Generalized Online Unit.................................... 12
Figure 2.3 - Logical Design of an Operand Section 13
Figure 2.4 - Bit Pair/Bit Train Usage Methods.. 15
Figure 2.5 - Logical Design of an Arithmetic Section 17
Figure 2.6 - Logical Design of a Selection Section 18
Figure 3.1 - High-level Logical Organization of the Chip 20
Figure 3.2 - System Signal Line Relationships 23
Figure 3.3 - Bit-slice Design of Online Units 24
Figure 3.4 - The Standard Cell Library Summarized... 26
Figure 4.1 - Logical Structure of the Arithmetic Section: Alignment 29
Figure 4.2 - Logical Structure of the Shifting Section: Alignment........................... 31
Figure 4.3 - The Alignment Unit (Diagram).. 33
Figure 4.4 - The Alignment Unit (Plot)... 34
Figure 5.1 - Logical Structure of the Operand Section: Sum-of-Squares 37
Figure 5.2 - Physical Structure of the Operand Section: Sum-of-Squares.................. 39
Figure 5.3 - Logical Structure of the Arithmetic Section: Sum-of-Squares................. 40
Figure 5.4 - Physical Structure of the Arithmetic Section: Sum-of-Squares............... 41
Figure 5.5 - Logical Structure of the Selection Section: Sum-of-Squares 42
Figure 5.6 - The Sum-of-Squares Unit (Diagram)... 44

Figure 5.7 - The Sum-of-Squares Unit (Plot)....45
Figure 6.1 - Logical Structure of the Operand Section: Square-Root....49
Figure 6.2 - Bit Patterns Produced as a Function of d_j: Square-Root....50
Figure 6.3 - Physical Structure of the Operand Section: Square-Root....52
Figure 6.4 - Logical Structure of the Arithmetic Section: Square-Root....54
Figure 6.5 - Physical Structure of the Arithmetic Section: Square-Root....55
Figure 6.6 - The **dsel** Selection Function: Square-Root....56
Figure 6.7 - Logical Structure of the Selection Section: Square-Root....57
Figure 6.8 - The Square-Root Unit (Diagram)....59
Figure 6.9 - The Square-Root Unit (Plot)....60
Figure 7.1 - Logical Structure of the Operand Section: Division....64
Figure 7.2 - Bit Patterns Produced as a Function of $d_j.hold$: Division....64
Figure 7.3 - Physical Structure of the Operand Section: Division....65
Figure 7.4 - Logical Structure of the Arithmetic Section: Division....67
Figure 7.5 - Physical Structure of the Arithmetic Section: Division....68
Figure 7.6 - The **qsel** Selection Function: Division....69
Figure 7.7 - Logical Structure of the Selection Section: Division....70
Figure 7.8 - The Division Unit (Diagram)....73
Figure 7.9 - The Division Unit (Plot)....74
Figure 8.1 - The Online Rotation Chip (Diagram)....77
Figure 8.2 - The Online Rotation Chip (Plot)....78
Figure 8.3 - Timing Analysis: Individual Unit and Chip-wide Online Delays....82

ACKNOWLEDGEMENTS

Foremost I would like to thank my project partner David Loh. From hours to months we slugged away at what appeared (at times) to be insurmountable obstacles, and together we shared the lonely satisfaction of overcoming them. I trust that our qualities of endurance and dedication, so tempered by this experience, will serve us well in professional life.

I would also like to extend a kind note of appreciation to my advisors, Prof. Milos D. Ercegovac and Prof. Tomas Lang, for their patience, encouragement, and sustaining confidence in our ability to bring the project to completion. Their technical guidance and help were made available to us at all times and their contributions are reflected in all aspects of the final design.

ABSTRACT OF THE THESIS

A Very Large Scale Integration Design of an Online Algorithm for the Computation of Rotation Factors

by

Stephen George Faris
Master of Science in Computer Science
University of California, Los Angeles, 1989
Professor Milos D. Ercegovac, Co-Chair
Professor Tomas Lang, Co-Chair

A VLSI design of an online algorithm for the computation of rotation factors is presented. The purpose of this design is to show that an integrated online approach can result in a modular, expansible, and high-speed VLSI implementation, and therefore is particularly well-suited to the hardware design of special-purpose algorithms. The algorithm is broken down into a series of sub-algorithms, each of which is realized in a separate hardware unit on a single chip, allowing highly parallel and efficient computation while minimizing overall complexity. A generalized design for an online arithmetic unit is developed, followed by an in-depth analysis of each individual unit and unit interconnections. Area, speed, and simulation results, as well as a fully-integrated testing scheme for the chip, are presented at the end.

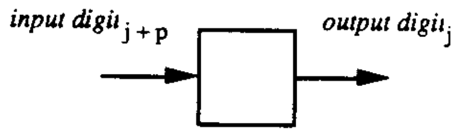
CHAPTER 1

INTRODUCTION

The purpose of this design is to show that an integrated online approach can result in a modular, expansible, and high-speed VLSI implementation, and therefore is particularly well-suited to the hardware design of special-purpose algorithms. In addition, a generalized design of an online unit is developed and shown to have broad applicability to the hardware design of a class of functions whose algorithms are describable in an online fashion.

Online arithmetic methods are characterized by digit-serial consumption of input digits and production of result digits in a most-to-least significant order, one digit per algorithmic step.[1,2,3] An online delay of p steps exists such that the jth digit of the result is produced after $j + p$ digits of the corresponding operands have been input, where p is determined by speed-complexity tradeoffs in the design process. Figure 1.1 illustrates this relationship. A typical online algorithm uses a recurrence equation to generate a residual quantity as a function of all previous result digits produced plus arriving operand digits. New result digits are selected (one per step) from a redundant digit set based upon the residual value. The final result may be converted to conventional form (if needed) by an on-the-fly conversion[4] of the redundant-digit results into the desired number base and format.

Single Online Unit **Timing Relationship**



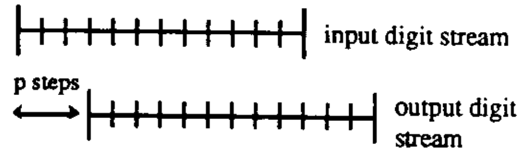
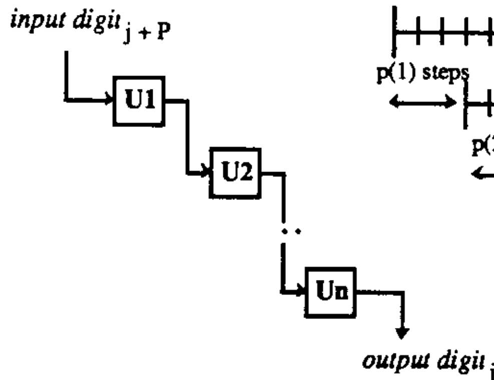


Figure 1.1 - Input Digit Consumption/Output Digit Production: Single Unit

A particularly complex algorithm may be broken down into a number of online sub-algorithms, each of which advances the process of computation from consumption of the original operand digits to production of final result digits. Each sub-algorithm may then be realized in a separate hardware unit functioning in parallel with all others. The challenge to the designer consists of determining how to decompose the algorithm into individual sub-algorithms and then how to integrate the digit consumption and production of each to provide for coordinated communication between predecessor and successor units. Figure 1.2 details how concurrent consumption of input digits and production of result digits occurs when multiple units are employed.

Multiple Online Units **Overall Timing Relationship**



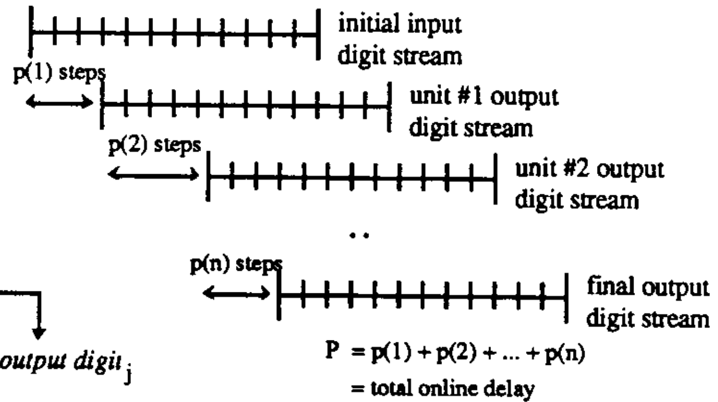


Figure 1.2 - Input Digit Consumption/Output Digit Production: Multiple Units

The characteristics and properties of an online algorithm for the computation of rotation factors are outlined in a previous paper by Ercegovac and Lang[5], which also describes potential application areas (namely matrix transformations) for such an implementation. The chip described in this thesis is based upon the design described therein with noted modifications. Detailed design information covering every aspect of the resulting chip's architecture, operation, and performance can be found in a separate document entitled *Detailed Design Specification - A VLSI Design of an Online Algorithm for the Computation of Rotation Factors*[6].

1.1 THE COMPUTATION OF ROTATION FACTORS

The algorithm in question computes the rotation factors of the Givens matrix transformation[7], defined as

$$C = \frac{G}{(G^2 + H^2)^{1/2}} \qquad S = \frac{H}{(G^2 + H^2)^{1/2}} \tag{1.1}$$

where $G = g \cdot 2^{e_g}$, $H = h \cdot 2^{e_h}$, $C = c \cdot 2^{e_c}$, and $S = s \cdot 2^{e_s}$. For this implementation a precision of 8 fractional binary digits was selected, corresponding to n=8 in the foundation paper by Ercegovac[1]. G and H are normalized floating-point numbers with 8-bit fractions in sign-and-magnitude and 4-bit exponents in 2's complement format, while C and S consist of a single integer digit (as explained later) and 8 fractional digits in sign-and-magnitude and a 4-bit exponent again in 2's complement format. The signs of all input operands are assumed positive, and the chip by definition produces strictly positive results, but it is possible to maintain the signs of G and H externally for signed operation. The

special cases $G = 0$ (producing $C = 0$ and $S = 1$) and $H = 0$ (producing $C = 1$ and $S = 0$) are neither permitted nor discussed. Figure 1.3 summarizes operand and result formats.

$G = g * 2$^eg	$g = .\,g1\ g2\ g3\ g4\ g5\ g6\ g7\ g8$	(normalized sign-and-mag.)
	$eg = eg3\ eg2\ eg1\ eg0$	(2's complement)
$H = h * 2$^eh	$h = .\,h1\ h2\ h3\ h4\ h5\ h6\ h7\ h8$	($g \neq 0$ and $h \neq 0$ guaranteed)
	$eh = eh3\ eh2\ eh1\ eh0$	
$C = c * 2$^ec	$c = c0\,.\,c1\ c2\ c3\ c4\ c5\ c6\ c7\ c8$	
	$ec = ec3\ ec2\ ec1\ ec0$	
$S = s * 2$^es	$s = s0\,.\,s1\ s2\ s3\ s4\ s5\ s6\ s7\ s8$	
	$es = es3\ es2\ es1\ es0$	

Figure 1.3 - Operand and Result Number Formats

The original algorithm called for the production of normalized C and S as the result; however, it was discovered during design that in fact c (of C) and/or s (of S) may possess a single integer digit under two separate sets of circumstances. The first occurs when the absolute exponent difference $|e_g - e_h|$ is so large that either c (of C) and/or s (of S) should equal exactly 1. Here, a result magnitude of 1.00000000 may be obtained. The second derives from the use of the original magnitudes g and h in off-line or full-parallel form in the division units of the final implementation. In fact, the proper quantities for use in the division unit should have been x and y, which represent the magnitudes g and h after shifting to equate their exponents e_g and e_h to the larger of the two values. The net effect is that whenever the larger of the original operands g or h is paired with the smaller exponent e_g or e_h, a non-zero integer digit may appear in c (if g was larger and e_g smaller) or s (if h was larger and e_h smaller), depending on the actual values of g and h and the absolute exponent difference $|e_g - e_h|$.

Result digits are produced in a digit-serial manner, meaning that additional online units may be connected to use chip outputs for the computation of even more complex algorithms having the equations of (1.1) as subcomponents. For this particular implementation, in fact, the serial outputs of all individual units comprising the computation are available to external units for further computation or testing purposes. In addition, chip inputs and outputs are received and transmitted in full-parallel form, with circuitry included for interface to a standard 32-bit CMOS microprocessor bus.

1.2 THE INTEGRATED ALGORITHM FOR COMPUTATION OF ROTATION FACTORS

The derivation of a single-step online algorithm for the production of final result digits was not undertaken, because the complexity of the resulting algorithm would lead to an implementation having an unacceptably long cycle time. Rather, the algorithm was decomposed into a series of sub-algorithms with a specialized unit assigned to execute each. The execution of each sub-algorithm is initiated sequentially, after a pre-determined online delay p_{unit}, which is fixed by the designer and based upon internal design characteristics and speed considerations within a unit. The individual chapters on each unit provide a full discussion of these issues. The breakdown of the full algorithm into sub-algorithms is given in Figure 1.4[1]:

Sub-algorithm	Range of Result	Hardware Unit
1. Computation of exponents e_c and e_s Computation and bit-serial production of the aligned fractions x (from g) and y (from h)	$-8 \le e_c, e_s \le 0$ $0.5 \le x < 1$ $0.5 \le y < 1$	Alignment (Arithmetic §) Alignment (Shifting §)
2. Computation of $z = x^2 + y^2$	$0.25 \le z < 2.0$	Sum-of-Squares (All §)
3. Computation of $d = z^{1/2}$	$.5 \le d < 2^{1/2}$	Square-Root (All §)
4. Computation of $c = g/d$ and $s = h/d$	$0.5 \le c, s < 2$	Division (x 2 units, All §)

Figure 1.4 - Algorithmic Breakdown into Sub-algorithms and Assignment to Units

Although the initiation of each unit's operation is sequential, once a unit has begun to operate it continues to produce result digits in parallel with the production of previous units as well as subsequent units whose operation has begun. During certain stages of execution, every unit is operating in parallel, consuming and producing result digits at the rate of one per step, and maximal parallelism is achieved. This particular implementation utilizes 8-bit mantissas (n=8 for the algorithm); an implementation corresponding to, say, n=16 would realize maximal parallelism for a longer time, and overall computational efficiency would be higher.

Each stage described above is realized in a single corresponding unit on the chip, with the exception of the division stage, which requires two identical units to compute the two result values. Because the range of possibilities for the base, format, and digit sets of each is so varied, the choices made must lead to the formation of an integrated algorithm where the outputs of any predecessor unit are produced in a form acceptable as inputs to its successor unit, and where overall chip input and output formats are compatible with the external world.

To arrive at an integrated algorithm, the representation format, range, and time availability of both input and outputs digits arriving and leaving a unit must be selected to achieve a balance between the performance of that unit and the requirements imposed by the intermediate role it plays in the computation, as well as the performance of the overall chip. Representation formats for input and output digits may include signed and unsigned binary, signed-digit, binary-coded, etc. Further, the formats used on and off-chip may vary depending on the characteristics of the system in which the chip operates. The range of input and output digits may be extended into the negative integers in the case of a redundant digit set. Finally, result digits must be produced at the point in time when they can be used immediately by a successor unit, else delay or conversion circuitry must be introduced that may slow throughput.

1.3 AN INTEGRATED ARCHITECTURE FOR COMPUTATION OF ROTATION FACTORS

In concert with the above statements, the division of labor on the chip closely matches the division of the algorithm into sub-algorithms, with each executed in a separate unit. The input and output digits sets for each were chosen to achieve interconnection compatibility with predecessor and successor units, and the online delay p_{unit} of each was chosen to minimize the total online delay p_{chip}. Figure 1.4 (see earlier) provided an overview of this division of labor.

The resulting calculation consists of 4 separate sub-algorithms which are then realized in 5 VLSI hardware units in a single-chip implementation. The alignment sub-algorithm is the only one of the four which does not operate in an online-fashion; rather, conventional arithmetic and shifting methods are sufficient to describe its operation. All

other units have a sub-algorithm which can be described in a standard online manner that leads naturally to a generalized online unit architecture.

CHAPTER 2

THE GENERALIZED ARCHITECTURE OF AN ONLINE UNIT

2.1 GENERAL STRUCTURE OF THE ONLINE ALGORITHMS

In general, the structure of the online algorithms found in this implementation can be described by the following model:

Algorithm *General*

1. Initialization Section: INITIALIZE registers to 0, or load their initial values
2. Recurrence Section: GENERATE the residual recurrence value $W[j]$;

   SELECT a result digit for this step, and
   UPDATE locally-held on-the-fly converted values,

where:

$W[j] = 2 \cdot W[j-1] + g$(converted previous input digits, converted previous result digits, and inputs during step j).

Figure 2.1 - General Structure of the Online Algorithms

During the high-level register-transfer design process for an algorithm of this form, one must select from two alternative approaches to the design of internal arithmetic and operand storage structures: the signed-digit and carry-save methods. Each method provides the major benefit of carry-free addition which is sought to minimize cycle times. The signed-digit approach leads to a more complex implementation because of the necessity to provide special hardware for operand storage and arithmetic operations. By contrast, the

carry-save approach minimizes the complexity of these structures, at the cost of having to include special hardware for internal on-the-fly conversion of previous result digits and perhaps arriving input digits. Arithmetic is then done in conventional manner. The carry-save approach was chosen for its overall simplicity and the perceived area savings of its structures.

Further, one notes that the computation can be divided into three separate tasks. First, a set of registers and logic is required to carry out the on-the-fly conversion, storage, and preparation of values to appear in the operand g of the recurrence equation. Online algorithms typically dictate that either arriving input digits, previous result digits, or both be locally on-the-fly converted and stored in registers to be used in the computation of subsequent residuals. The paper by Ercegovac and Lang provides details concerning on-the-fly conversion schemes. Further preparation logic is then added to combine these on-the-fly converted values with the value(s) of the current input digit(s) and produce the actual operands required in g.

Second, a set of registers and summing logic is required to execute the addition forming the residual $W[j]$ by carrying forward a portion of the previous residual $W[j-1]$ as $2 \cdot W[j-1]$, and combining it with the input g. This is best realized by integrating a set of registers into a carry-save adder structure and introducing a feedback loop such that $2 \cdot W[j-1]$ is re-combined with the newly-produced g at every step.

Finally, summing and selection logic are necessary for the selection of result digits from a redundant digit set based upon the residual value or an estimate of it. Throughout the chip, residual values are computed using carry-save adders; thus, the residual contains both a partial-sum and a carry-save component. If result digit selection required knowledge

of the residual's exact value, then a 2-1 reduction along the entire length of these components would be necessary preceding selection, introducing delay into the step time of the algorithm. The sub-algorithms used in this implementation require only an estimate of the residual value, meaning that a 2-1 reduction of only a few of the most significant bits of the partial-sum and carry-save components of $W[j]$ is necessary. Selection logic connects directly to the reduced residual estimate.

2.2 DESIGN OF A GENERALIZED ONLINE UNIT

By dividing the hardware architecture of an online unit into three separate sections corresponding to the three tasks outlined above, a generalized model emerges having broad applicability to the class of algorithms used in this implementation. In the sections below, this architecture is described and the general functions of the Selection, Arithmetic, and Operand sections are discussed.

Logical Design of a Generalized Online Unit

Figure 2.2 details the spatial arrangement of the Operand, Arithmetic, and Selection sections that results when inter-unit connection and intra-unit data flows are taken into consideration.

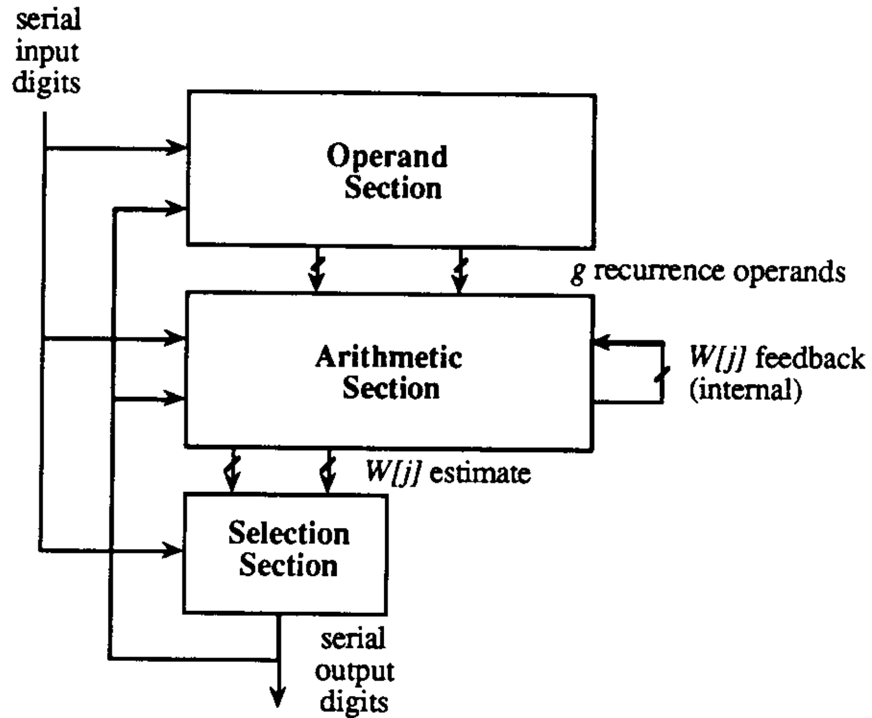


Figure 2.2 - Logical Design of a Generalized Online Unit

Logical Design of an Operand Section

The Operand section is generally responsible for producing all the operands appearing in the recurrence equation for $W[j]$ except for that portion dependent upon the previous $W[j]$, namely $2 \cdot W[j-1]$, that is stored and fed back internally within the Arithmetic section. The functionality of the Operand section is logically divided across three different sub-sections, each of which contributes to operand production. Figure 2.3 details the arrangement of these sub-sections.

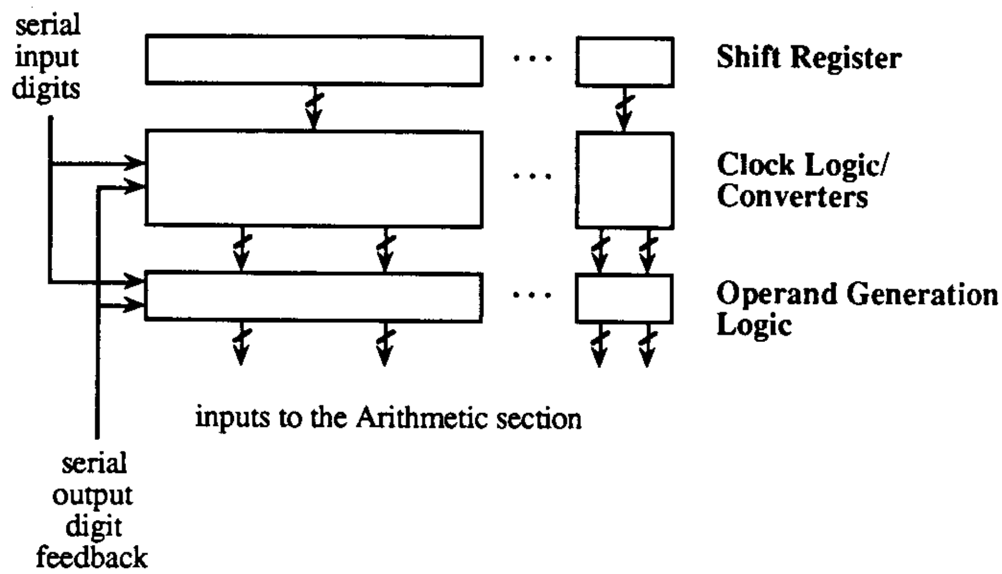


Figure 2.3 - Logical Design of an Operand Section

A shift-register across the top comprises the first sub-section; its purpose is to carry either a travelling bit pair or a bit train across the unit in order to mark those positions where on-the-fly converting should take place. In some cases, the input digit set for a value to be on-the-fly converted is non-redundant, and conversion degenerates into a simple appending of the arriving digits onto the end of the converted value. Here, a single conversion register and a bit pair are sufficient, with the pair marking the position where appending should take place. In other cases, the input digit set is redundant, and conversion may require all or any number of digits of previously converted values to be swapped between two conversion registers. Here, the bit train marks those positions where parallel-loading should take place.

Clocking logic and the actual converters comprise the second sub-section. Figure 2.4 details the operation of each of two conversion methods used assuming a two-phase clocking scheme. In the first example, only a single traveling bit *pair* is needed, because on-the-fly conversion requires only an appending operation in position i after step i of the

algorithm. In the second example a bit *train* is necessary, since on-the-fly conversion requires that all previous bit-positions be loaded between two conversion registers. In this case, the appending position is marked by logic which detects the forward edge of the train, while loading positions trail behind. Both markers are then added to the appropriate load-triggering signal and synchronized by clocking logic to a system clock. The converters themselves consist merely of flip-flops; where only an appending operation is required, simple resettable flip-flops are adequate, while flip-flops with multiplexed inputs are required for the case where parallel loading occurs.

Note that the logic as shown in the diagram will deliver a clock to a particular position in either register of the converter subject to two situations. When a full parallel load from the opposite register is required, the load signal will be active, and all positions will receive a clock. In this case, the end-detection logic is ignored. However, in the case when only an append operation takes place, no load signal will appear, and the end detection logic provides the only clock activation signal, namely to the latch in position j of the converter.

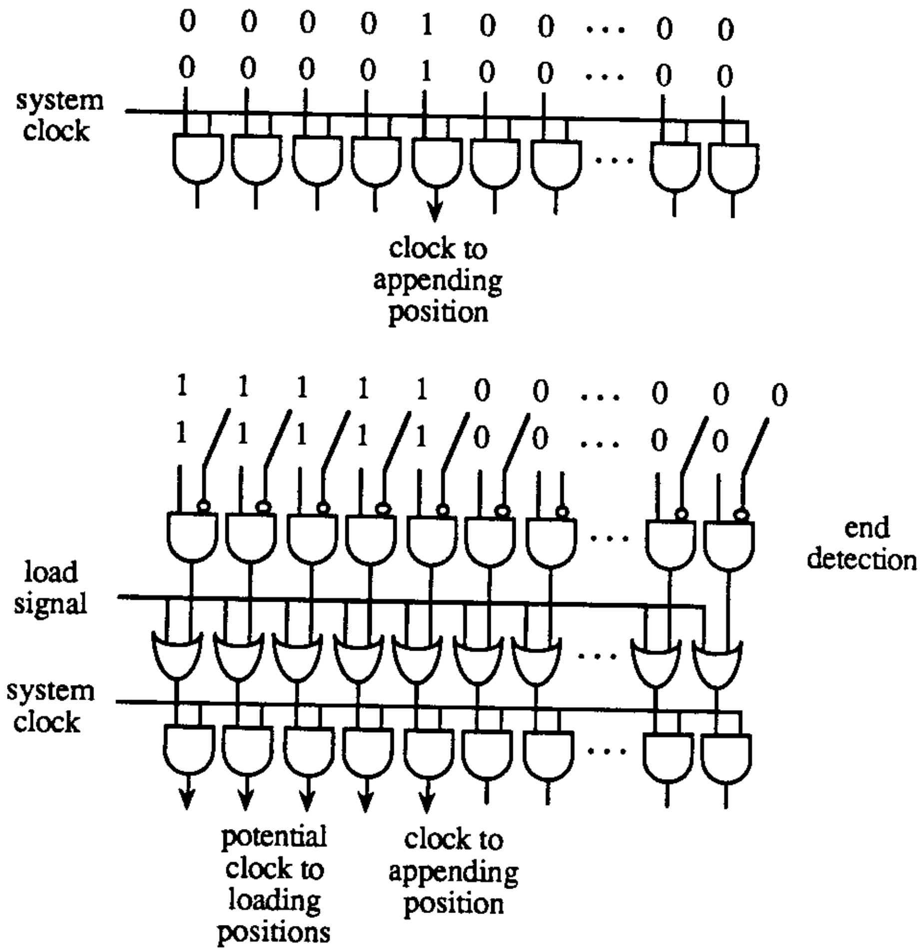


Figure 2.4 - Bit Pair/Bit Train Usage Methods

Finally, some additional logic is required to produce the operands which comprise g in their final form. This typically involves modifying the on-the-fly converted values or appending additional digits to them according to a set of rules to factor in the contribution of arriving input digits. This logic accepts the input digits and the output of the converters above, modifies the converted values to produce the operands of g, then passes the result down to the Arithmetic section.

Logical Design of an Arithmetic Section

The Arithmetic section is responsible for actual calculation of the residual value *W[j]*, and consists of a set of registers and summing logic, specifically carry-save adders. Internally, 3-2 and 4-2 reductions are continually performed, producing the carry-save form of the residual from the carry-save form of the previous residual plus the two newly introduced operands from the Operand section above. This is accomplished through a feedback loop which re-introduces the previous residual *W[j]* and multiplies it by 2 as required by the recurrence equation. All reductions are performed using two levels of cascaded 3-2 carry-save adders. Figure 2.5 details the resulting structure.

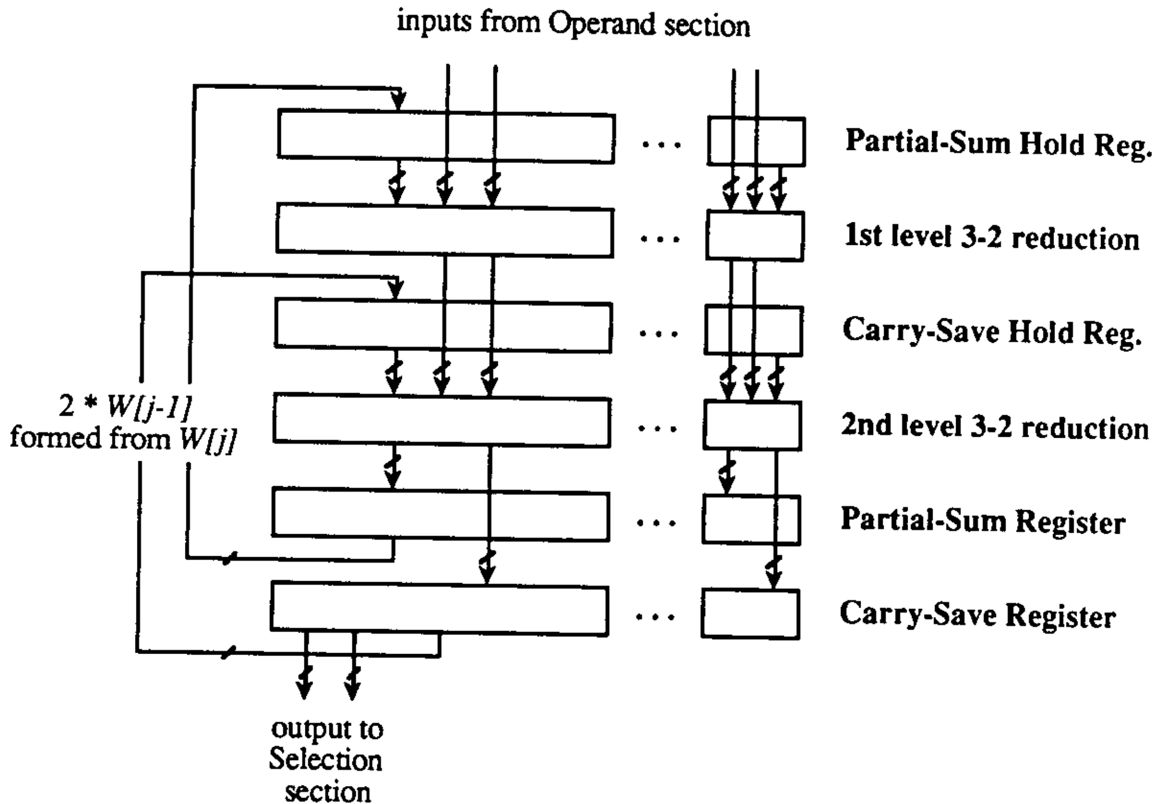


Figure 2.5 - Logical Design of an Arithmetic Section

Logical Design of a Selection Section

The Selection section is responsible for the selection of result digits based upon an estimate of the value of *W[j]* and (in some cases) a current input digit. The section is comprised of summing logic, which reduces the input and the carry-save form of *W[j]* to straight binary, and selection logic, which assigns digits based upon the value obtained. In one case, only the two parts of the carry-save form of *W[j]* need be reduced, and a simple carry-propagate adder is adequate for the task. When an input digit is assimilated into the selection quantity at this point an additional addition operation may be required. This is accomplished by inserting a carry-save adder to reduce the three operands (the input digit,

plus the partial-sum and carry-save forms of $W[j]$) to two, which the carry-propagate adder can then reduce to straight binary.

Figure 2.6 details the resulting structure. Note that the section is positioned under only a few of the most significant digits of $W[j]$; the full precision reduction of $W[j]$ is not required. Instead, only an estimate of the true value of $W[j]$ is used; the extent to which the section extends into lower-order digits is a measure of the accuracy of the estimate of $W[j]$ needed. As a rule, lesser accuracy in the estimate decreases the step time yet requires a greater online delay p_{unit}. Hence, a tradeoff exists between the increasing the online delay of the unit, a one-time delay incurred before result digit production begins, and increasing the step time of the algorithm, which must be paid in every cycle of the chip's operation.

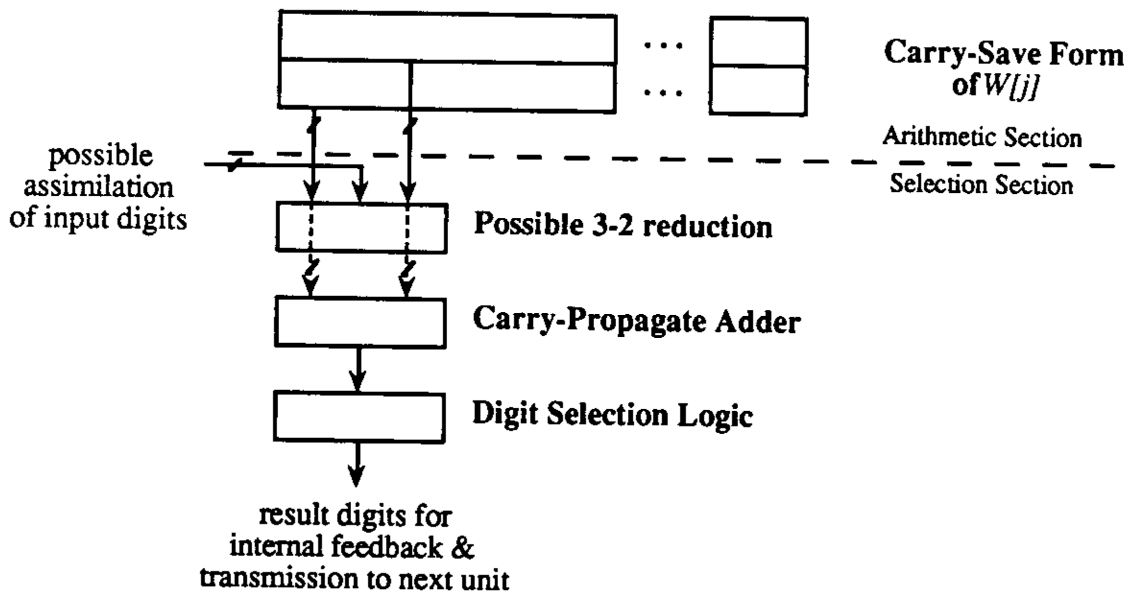


Figure 2.6 - Logical Design of a Selection Section

CHAPTER 3

HIGH-LEVEL FUNCTIONAL DESCRIPTION OF THE CHIP

3.1 INPUT, INTERMEDIATE, AND OUTPUT OPERANDS

The complete functionality of the chip is divided into 5 different units, each of which advances the progress of computation from consumption of the original input digits to production of the final result digits. Figure 3.1 gives a high-level logical description of the resulting architecture, which includes an Alignment Unit, a Sum-of-Squares Unit, a Square-Root Unit, and two identical Division Units.

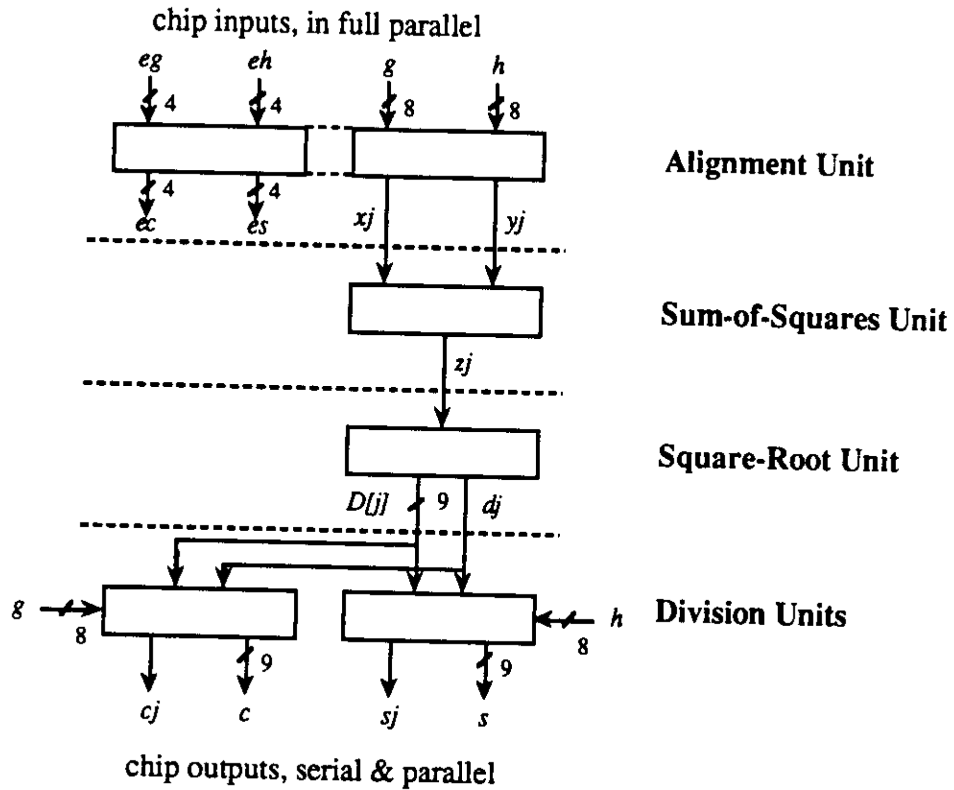


Figure 3.1 - High-level Logical Organization of the Chip

The chip receives operand inputs and transmits result outputs in full-parallel form, consistent with its use in a 32-bit CMOS microprocessor-based system. All internal transmission, however, occurs digit-serially in online fashion. The Alignment Unit is responsible for the initial translation of g and h in parallel form to x_j and y_j in bit-serial form, as well as for the production of e_c and e_s in full-parallel form, while the Division Units output both a digit-serial and a full-parallel form of their outputs c_j and c, and s_j and s, respectively. No provision for serial inputs to the chip was made.

3.2 SYSTEM SIGNALS AND THEIR RELATIONSHIP

In addition to the operands mentioned above, 11 system-wide signal lines are utilized to manage all activities that take place on the chip. These lines are comprised of power and ground leads, 4 clocking inputs, 4 system reset and initiation lines, and finally an output sampling line.

The four clocking inputs supply true and complement forms of a two-phase non-overlapping clock, and are designated Ø1, Ø1bar, Ø2, and Ø2bar. The barred inputs could have been generated from the true inputs on-chip; however, for simplicity we decided to generate these externally, where greater flexibility exists to ensure that any skew or overlap are avoided. The final implementation includes full-buffering of all clock inputs on-chip to completely eliminate the possibility of skew.

The two-phase clocking scheme was chosen for this implementation for several reasons. First, designs based upon a two-phase scheme guarantee a race-free implementation where the designer need not worry about the relative length of logic delays between latches. Second, a two-phase scheme permits use of level-triggered flip-flops, which are relatively smaller and faster than edge-triggered, master-slave ones. Finally, each step in the online algorithms implemented, as well as in the operation of the Alignment Unit, which is non-online, divides naturally into two different sets of tasks, one of which must be completed before the other. One of the clocks is used to initiate the operation of each set, and hence a full algorithmic step for each unit consists of a pulse from both Ø1 and Ø2.

Next, the system calculation initiation lines, ENABLE and ENABLEbar, are used to time the input of the original operands into the latches of the Alignment and Division Units. In the Alignment Unit, the operands just input are used immediately in the production of result exponents, as well as in the production of x_j and y_j from g and h, while in the Division Unit g and h are stored for a number of steps ($= p_{\text{div}}$) before use. The next pair of lines, GRESET (for Global Reset) and GRESETbar, are used to preset all latches on the chip to all zeros, corresponding to the initiation sections of each online sub-algorithm.

The final system input, READ, is used to trigger output of the full-parallel results of the chip's calculation, c and e_c of C, and s and e_s of S. The usual time to do so would be after the chip has fully completed the calculation; however, the READ signal can be used at any time to sample the exponent outputs e_c and e_s, or the mantissa outputs c and s. One possible advantage of this capability is for testing, where c and s could be sampled at intermediate points to determine their values during any step of the algorithm.

Figure 3.2 below shows in detail the relationship between the system clocking, calculation initiation, and reset lines. For brevity, only their true forms are shown, the complement forms are easily deduced. Note that all activity in the system is synchronized to one of the two system clocks, even the behavior of ENABLE and GRESET. ENABLE is defined to rise and fall in sync with the leading edge of Ø2, while GRESET is similarly defined with respect to Ø1. The interspersed lines are meant to indicate that any period of time may elapse with system signals held at that level before calculation begins. ENABLE and GRESET are only active for this brief period of time at the beginning of a calculation, thereafter they have no function and remain at the 0 level.

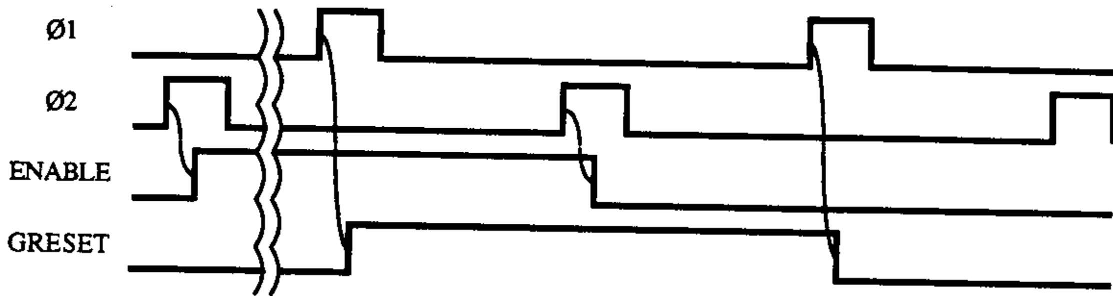


Figure 3.2 - System Signal Line Relationships

3.3 FUNDAMENTAL DESIGN APPROACH

Fundamentally, the designs used for all units on the chip (with the exception of Alignment) are based upon the generalized model for an online unit outlined in Chapter 2. In every case, these designs were implemented in a bit-sliced manner, to facilitate customization of the resulting units to whatever precision desired of the chip's outputs. In the case of this implementation, a value of n=8 was adhered to, meaning that the results of all units can be guaranteed to have an accuracy of 8 fractional digits after conversion to conventional form by the on-the-fly method.

The bit-sliced methodology used brings several valuable advantages to the processes of design and customization. First, from a designer's point of view, a bit-slice approach is ideal because it allows design changes to be implemented with only a few changes to a single module, and causes a minimum of disruption to the remainder of the design. The units of this implementation lend themselves very naturally to this approach, because in most cases a consistent amount of hardware, connected in a consistent manner, is devoted to each slice. The logical design presented in Figure 2.2 is simply sliced vertically, with each slice roughly corresponding to one of the n positions required to

produce results with an accuracy of n fractional digits (in some cases, an extra position or two may be required). Figure 3.3 displays the resulting structure.

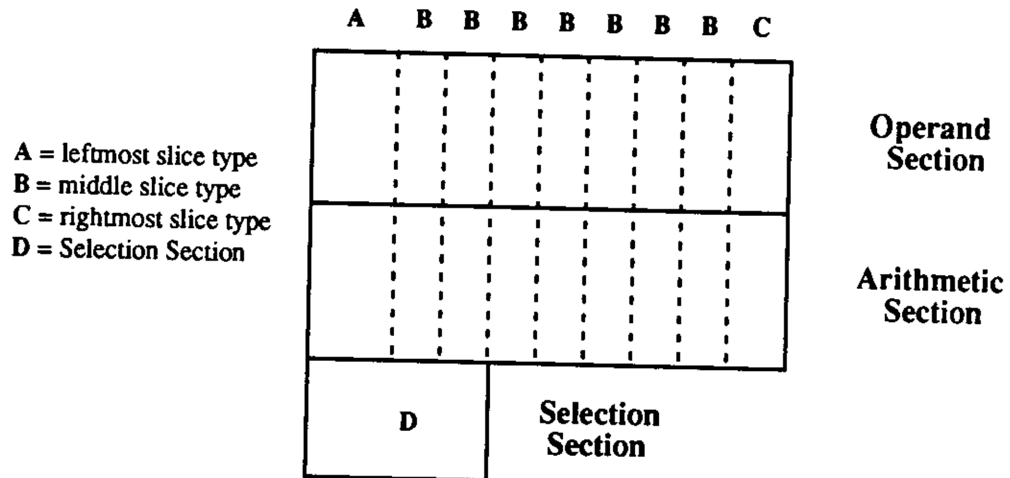


Figure 3.3 - Bit-slice Design of Online Units

In terms of implementation, a bit-sliced approach is of great benefit for the customization capabilities it offers. Theoretically, any number of digits of accuracy required in the result can be achieved by simply extending the internal accuracy with which each online unit operates. In a bit-sliced design, this is accomplished by simply inserting more of the type B slices into the design of each unit. Of course, a practical limit exists as to the number of slices which can be inserted, primarily due to electrical considerations such as the ability of busses to meet current supply requirements, etc.

Taking this approach one step further, it is theoretically possible for the type A, B, and C bit-slices of all online units to be incorporated into a single type A, type B, and type C slice (3 total), with the differing Selection sections incorporated into the type A slice. This would have resulted in a tall, thin rectangular implementation. This approach was not

chosen primarily because the resulting dimensions do not fit practically onto the project area available on a typical-sized chip, which is usually roughly square. Another objection raised was that the width of each of these "mega-slices" would be dominated by the width of the widest slice from among all units, and lead to inefficient use of project area in the thinner contributions from other units.

3.4 FUNDAMENTAL BUILDING BLOCKS OF THE IMPLEMENTATION

At the circuit level, the fundamental building blocks of the implementation consist of a set of logic, arithmetic, flip-flop, shifting, and buffer cells which together comprise the standard cell library. Figure 3.4 summarizes these cells. The physical and electrical characteristics of each are discussed in a later chapter - they are introduced here to facilitate the circuit level descriptions of all units in succeeding chapters. For convenience, the delay and area (in lambda^2, so as to be technology-independent) are provided also. The actual implementation was done using a 3 micron CMOS p-well process, and strictly static designs were used to accomodate the capabilities of the simulation tools at our disposal.

Type	Name	Description	Delay (ns)	Area (lambda^2)
logic	inv	simple inverter	1.5/1.25	752
	nand2	2-input nand gate	2.0/2.5	1175
	nand3	3-input nand gate	2.75/4.75	1551
	nor2	2-input nor gate	3.5/1.75	1175
	nor3	3-input nor gate	7.0/1.5	1551
	mux	2-1 multiplexor	6.75/7.5	3975
arithmetic	fad	full adder	8.25/8.25	4872
	inv.fad	simple inverter (same height as a fad)	1.25/1.0	1740
flip-flop	dff	D-type flip-flop	6.0/3.0	2440
	rsff	resettable D-type flip-flop	8.25/7.75	3264
	sff	settable D-type flip-flop	6.5/7.75	3944
	muxff	D-type flip-flop with 2-way	na/na	6283

		multiplexed input		
	rsmuxff	resettable version of the above	na/na	7590
shift	rshift	resettable master-slave flip-flop	na/na	7216
	sshift	settable master-slave flip-flop	na/na	8036
	muxshift	master-slave flip-flop w/ 2-way multiplexed input	na/na	11700
buffer	buff	inverting buffer	5.75/5.5	1175

Figure 3.4 - The Standard Cell Library Summarized

The functionality of each cell is self-explanatory. One point that should also be mentioned here regards the flip-flop and shift-type cells of the library. All flip-flop cells were designed to be level-triggered, as opposed to edge-triggered, and utilize only one of the two system clocks, plus its complement, as the strobe input. The shift cells are composed of two simple flip-flops in a master-slave configuration, and hence require both clocks plus their complements to operate.

CHAPTER 4

ALIGNMENT UNIT ARCHITECTURE

The Alignment Unit is the first major unit in the calculation chain and performs two main tasks. First, it calculates the result exponents e_C and e_S, given the original operand exponents e_g and e_h. Second, it aligns the original mantissa values g and h by delaying the introduction of either one into the x_j and y_j digit streams, respectively, (inserting zeros) to equate the exponents e_g and e_h to the larger of the two values.

4.1 ALGORITHMIC ANALYSIS

Calculation of the exponent difference is performed in bit-parallel manner using a conventional two's complement adder:

algorithm Align (Result Exponent Calculation)

$e_{diff} = e_g - e_h;$

if ($e_{diff} \geq 0$) **then**
{$e_C = 0;$
$e_S = -e_{diff}$}
else
{$e_C = e_{diff};$
$e_S = 0$}
endif

end *Align*

(4.1)

The alignment of g and h to form x_j and y_j is similarly simple, although an iterative algorithm is required to produce two values at every step j.

algorithm Align (Alignment Operation)

for $j = 1,2, \dots , n = 8$

if $e_{diff} \geq 0$ **then**
$\{x_j = g_j;$
if $j \leq e_{diff}$ **then** $y_j = 0;$
else $y_j = h_{(j-e_{diff})}\}$ (4.2)
else
$\{y_j = h_j;$
if $j \leq |e_{diff}|$ **then** $x_j = 0;$
else $x_j = g_{(j-|e_{diff}|)}\}$

end *Align*

4.2 LOGICAL AND PHYSICAL DESIGN OF SUB-SECTIONS

The Arithmetic Section

The Arithmetic section of the unit contains all the circuitry needed to calculate result exponents, and in addition executes the iterative algorithm for alignment of the mantissas. Shift control signals are fed to the shift registers of the Shifting section below, where mantissa values g and h are initially latched in parallel, then shifted out as x_j and y_j. Figure 4.1 details the logical structure of the Arithmetic section.

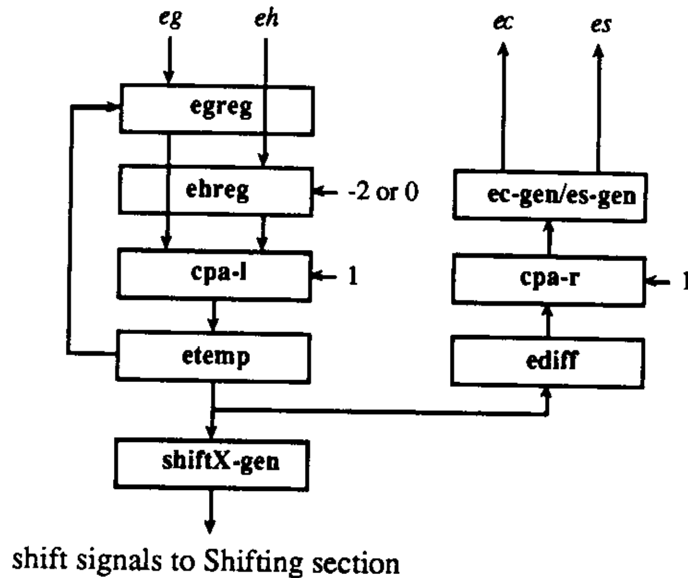


Figure 4.1 - Logical Structure of the Arithmetic Section: Alignment

At the beginning of the chip's operation, the ENABLE signal is used to direct the inputs of **egreg** and **ehreg** to accept the initial inputs *eg* and *eh* . A constant 1 is forced at the right end of **cpa-l**, expressly for the purpose of forming $-eh = ehbar + 1$. e_g and $-e_h$ are then added in **cpa-l** to form $e_{temp} = e_{diff} = e_g - e_h$. **ediff** receives only a single clock pulse, because it's only function is to store the original exponent difference for computation of e_s and e_c. To accomplish this, **cpa-r** first generates $-e_{diff} = e_{diffbar} + 1$. Immediately thereafter, **es-gen** selects either $e_s = -e_{diff}$ ($e_{diff} \geq 0$) or $e_s = 0$ ($e_{diff} < 0$), while **ec-gen** selects $e_c = 0$ ($e_{diff} \geq 0$) or $e_c = e_{diff}$ ($e_{diff} < 0$).

After the initial computation, the tasks turns to counting the original exponent difference, now stored in **etemp**, either down to 0 ($e_{temp} > 0$) or up to 0 ($e_{temp} < 0$) by one, while supplying the proper shift signals to the Shifting section. Counting down is accomplished by continually adding -1 to the difference in **cpa-l** until 0 is reached, while counting up is done by continually adding 1. Since a constant 1 is fed into the right end of

cpa-l, we must form either -2 or 0 which, when added to this constant 1, will form the -1 or 1 desired. The inputs of **egreg** and **ehreg** are re-directed, and these registers are now utilized as a temporary intermediary register and as a source of either the -2 or 0 as required. During the ensuing steps of the algorithm, the value in **etemp** is continually added to this -1 or 1 value until 0 is reached; at this point, clocking of **etemp** ceases.

The shift signals *shiftG* and *shiftH* are formed in **shiftX-gen** by ORing the ENABLE signal (to parallel load the original mantissas), z_d (a signal that becomes valid when $e_{temp} = 0$ is reached), and either *passG* or *passH*. *passG* is valid immediately iff the original exponent difference (e_{diff}) was positive, while *passH* is valid iff the difference was negative. *shiftG* is synced to a system clock to produce a shift operation in **gshift**, and *shiftH* is applied to **hshift** in the same manner.

Physically, both **egreg** and **ehreg** are implemented using muxff's, since their inputs must be switched from the original exponents e_g and e_h to their inputs when used as intermediate registers. **cpa-l** and **cpa-r** are comprised exclusively of **fad**'s, and **etemp** and **ediff** immediately following are implemented using plain dff's. Both **ec-gen/es-gen** and **shiftX-gen** are composed of logic gates.

The Shifting Section

The Shifting section of the unit contains all the circuitry needed to store the initial mantissa values at the beginning of a calculation, and thereafter to shift out g and h as x_j and y_j, respectively, depending on the shift signals received from the Arithmetic section above. The section consists of two shift-registers, **gshift** and **hshift**, composed of muxshift's. Initially, the ENABLE signal is used to trigger parallel loading of the

mantissas; thereafter, the section relies on signals from the Arithmetic section to ensure that valid x_j's and y_j's are shifted out of **gshift** and **hshift**, respectively, at their proper times. Figure 4.2 details the logical structure of the Shifting section.

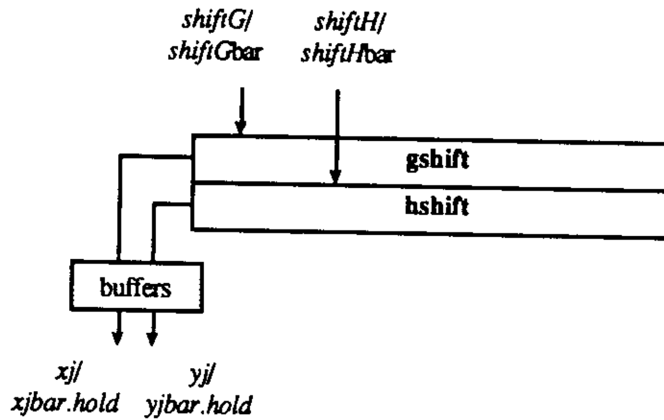


Figure 4.2 - Logical Structure of the Shifting Section: Alignment

In addition to x_j and y_j, two additional serial outputs, *xjbar.hold* and *yjbar.hold*, are produced in this section. These two signals represent the inverted values of x_j and y_j delayed for a half cycle to arrive at the proper point in time for use in the next unit. This delay circuitry was included here for convenience only - it could as well have been incorporated into the Sum-of-Squares unit.

4.3 THE RESULTING STRUCTURE

No attempt was made to design the unit in a bit-sliced manner, although in fact the Shifting section naturally lends itself to a bit-sliced design, with each slice corresponding to a single bit position in both g and h. The Arithmetic section, unfortunately, consists mostly of ad-hoc register, arithmetic, and logic circuitry, and a bit-sliced design would have been a

much more difficult task. In view of this, a standard design was undertaken, with exponent widths chosen at 4 bits so that exponent differences from -8 to +7 could be represented, as mentioned earlier. This range of allowable exponent differences provides for maximum flexibility in delaying significant bits of either mantissa in the formation of x_j and y_j.

The physical structure of the entire unit consists of the Arithmetic and Shifting sections stacked one upon the other and aligned at the left. In the vertical dimension, the height of the unit was determined by the heights of both sub-sections. The width of the unit is dominated by the width of the shift registers required to hold both mantissa values.

In the final implementation, the Arithmetic section measured 932 microns high by 1131 microns wide, while the Shifting section measured 656 microns high by 1662 microns wide, yielding dimensions for the entire unit of 1588 by 1662 microns. With respect to the Arithmetic section, one can only estimate how these figures would change in a larger implementation; however, expansion would only be required in the horizontal direction, and approximately 530 microns of excess area exists there already on the right-hand side of the unit.

With respect to the Shifting section, an additional pair of stacked muxshift cells would be required for every 1-bit increase in fractional length n, each of which measures 150 microns in width, and hence the entire unit would increase 150 microns in width (to the right) for every bit added.

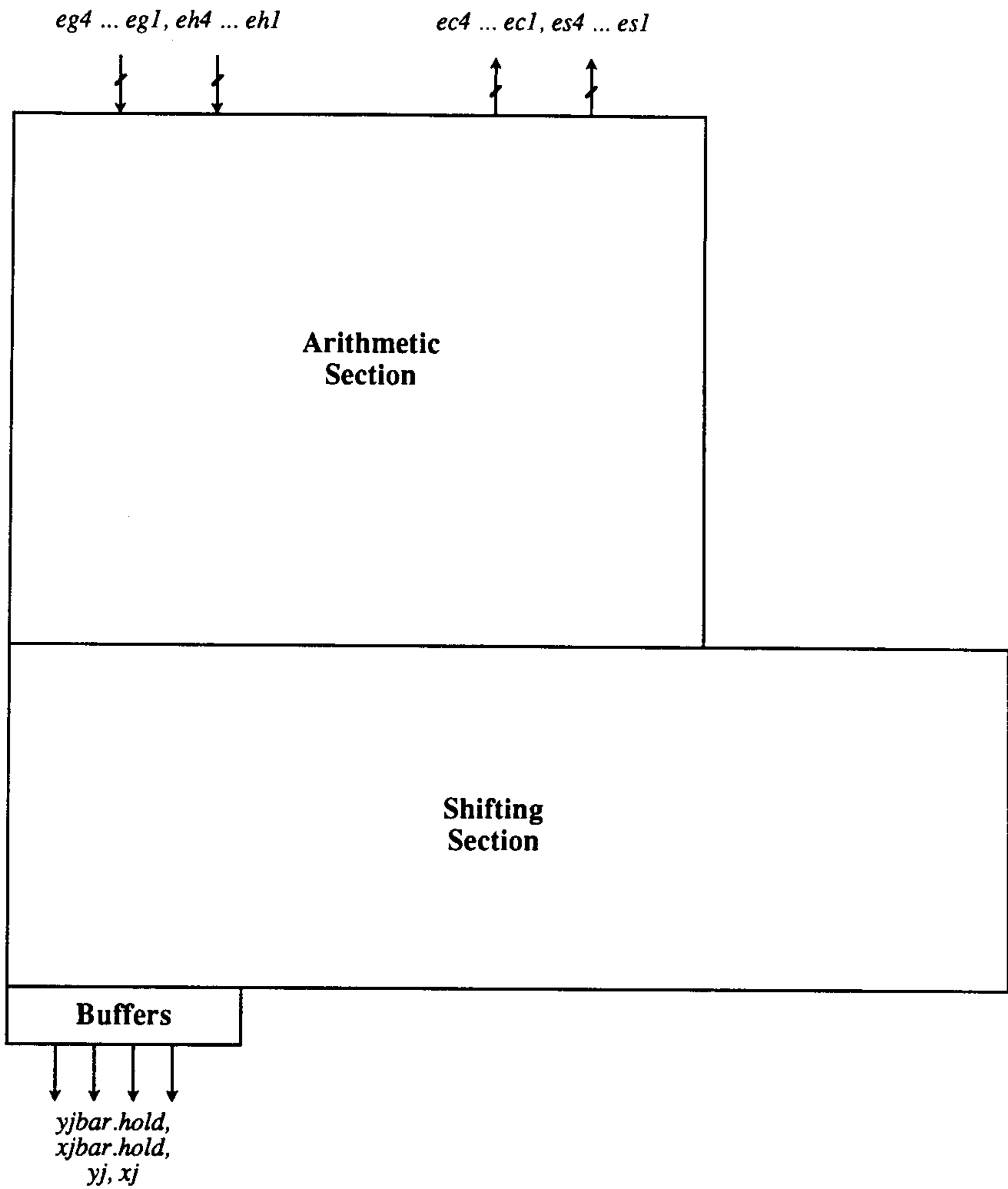


Figure 4.3 - The Alignment Unit (Diagram)

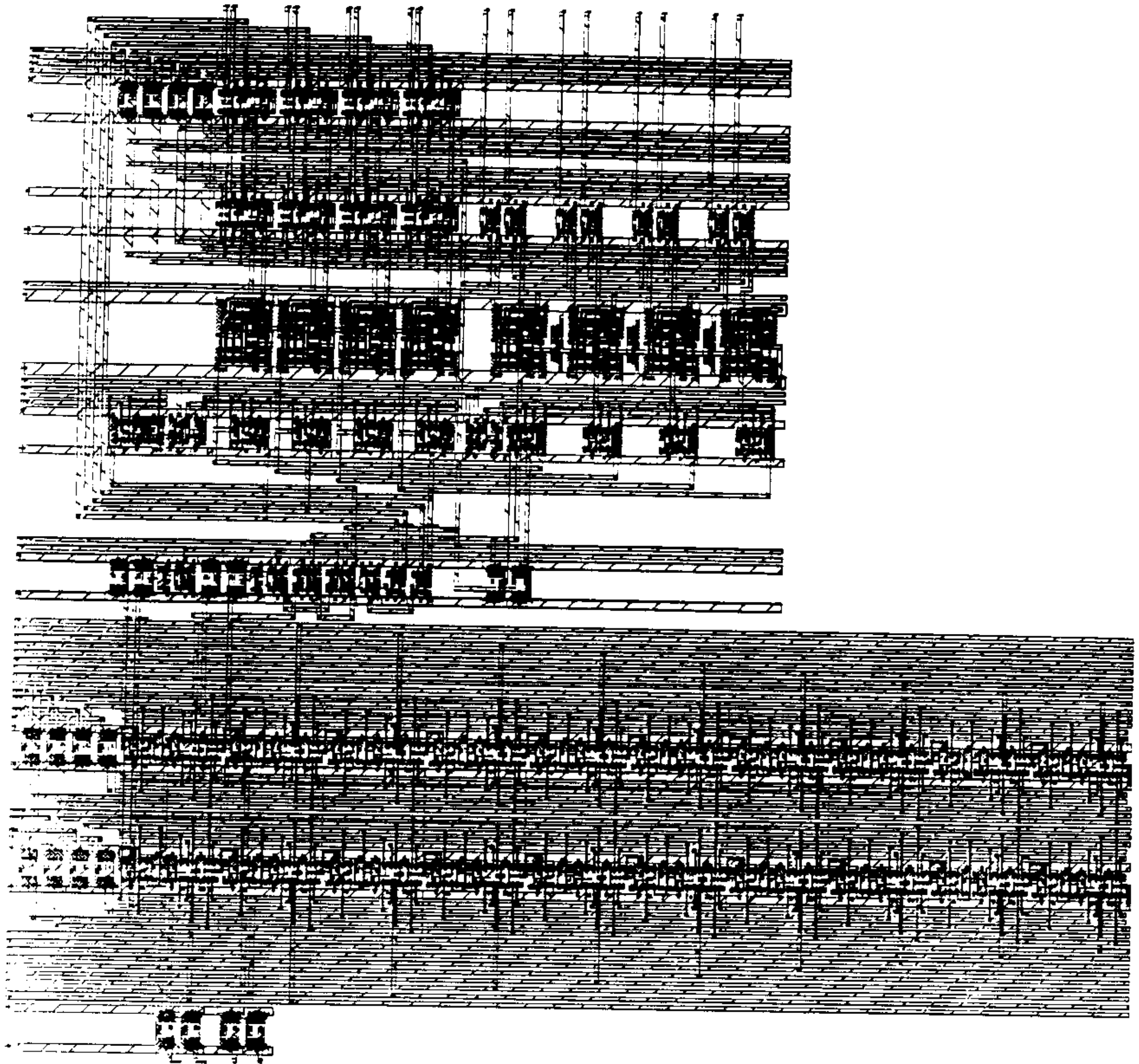


Figure 4.4 - The Alignment Unit (Plot)

CHAPTER 5

SUM-OF-SQUARES UNIT ARCHITECTURE

The Sum-of-Squares Unit computes the sum of the squares of the aligned fractions x and y, whose digits are received serially as x_j and y_j from the Arithmetic Unit above. The design of this unit is based upon the generalized model of an online unit outlined in Chapter 2. The unit's output consists of a digit stream z_j, $0 \le z_j \le 6$, such that the equation $z = x^2 + y^2$ is satisfied, with n=8 fractional digits of precision.

5.1 ALGORITHMIC ANALYSIS

This algorithm is the first of three to be discussed whose structure matches that of the generalized model of an online algorithm outlined in Chapter 2. In this case, $W[j]$ is the residual whose value is repeatedly generated at every step, z_j represents the result digit produced during step j, and $X[j]$ and $Y[j]$ represent the full parallel, on-the-fly converted values produced from x_j and y_j, respectively, after every step. Since the range of input digits found in x_j and y_j is [0,1], and no negative digits are possible, on-the-fly conversion defaults to a simple appending operation.

For the sake of brevity in the discussion that follows, the algorithm is annotated at right and some quantities that appear in the recurrence equation are given pseudonyms. Later, when the process by which these quantities are produced is detailed, the pseudonyms are used to avoid repeating unnecessary complex formulae.

algorithm Sum-of-Squares (5.1)

```
X[0] = .000000000;                         /* INITIALIZE internal registers */
Y[0] = .000000000;                              /* for online operands */

W[0] = 00.00000000;                        /* INITIALIZE the residual value */

for j = 1,2, ... , n+1 = 9 {               /* GENERATE recurrence value */

W[j] = 2 • csfract W[j-1] +
       2 • { X[j-1] + xj • 2^-(j+1) } • xj +          /* 2 • {Qx} • xj */
       2 • { Y[j-1] + yj • 2^-(j+1) } • yj;           /* 2 • {Qy} • yj */

zj = csint W[j];                                /* SELECT result digit */
X[j] = convert (X[j-1], xj);               /* UPDATE online operands */
Y[j] = convert (Y[j-1], yj);
}

end Sum-of-Squares
```

5.2 LOGICAL AND PHYSICAL DESIGN OF SUB-SECTIONS

The Operand Section

The Operand section of the unit contains all the circuitry required to generate those operands of the recurrence equation, namely $\{Q_x\} \cdot x_j$ and $\{Q_y\} \cdot y_j$, whose values depend upon the input digit streams x_j and y_j received from the Alignment Unit. In effect, this task can be reduced to producing the binary bit patterns that represent these quantities, then ensuring that they are fed into the Arithmetic section aligned at the proper position with respect to the binary point. Because these quantities depend upon the on-the-fly converted values of previous input digits x_j and y_j, namely $X[j-1]$ and $Y[j-1]$, the converters are physically placed in the Operand section also. Figure 5.1 details the logical structure of the Operand section.

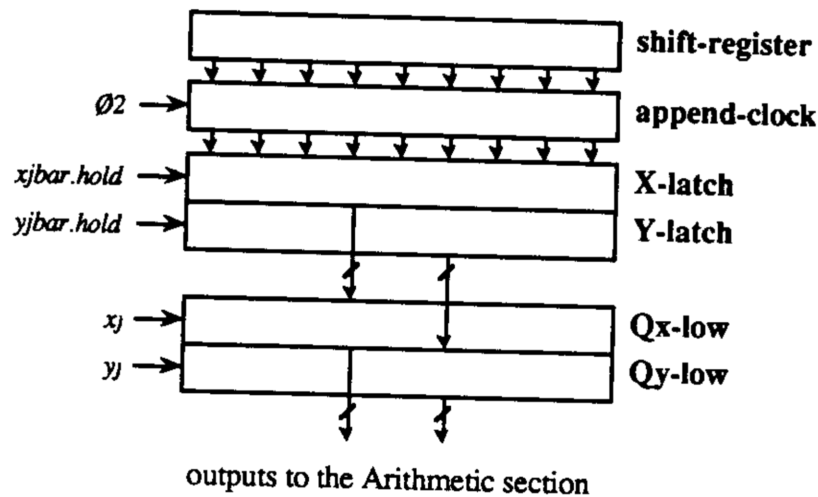


Figure 5.1 - Logical Structure of the Operand Section: Sum-of-Squares

The section operates as follows. At the beginning of the chip's operation, the GRESET signal is used to preset all latches to 0, except for the leftmost master flip-flop in **shift-register**, which is seeded with a 1. This one is the source of a bit pair that travels across the section, moving one position per step of the algorithm, and marking the current position j. A bit pair is required because each stage of the shift register is composed of two flip-flops in a master-slave configuration, with all masters using one system clock and all slaves the other clock. This bit pair marker is used during every step of the algorithm to mark where operations in the jth and j+1st positions should take place.

The section's first objective is to produce the bit patterns corresponding to Q_X = < $X[j\text{-}1]$, 0, $xj_{(j+1)}$, 0, ... , 0 > and Q_Y = < $Y[j\text{-}1]$, 0, $yj_{(j+1)}$, 0, ... , 0 >. However, rather than forcing the values of x_j and y_j into the j+1st positions as dictated, a 1 was forced, producing Q_X = < $X[j\text{-}1]$, 0, 1_{j+1}, 0, ..., 0 > and Q_Y = < $Y[j\text{-}1]$, 0, 1_{j+1}, 0, ..., 0 >. Forcing in this fashion was mathematically equivalent for this case, where the range of

x_j and y_j is [0,1], because the next step involves an ANDing operation with the true x_j and y_j anyway, yielding x_j AND 1 (= x_j AND x_j), y_j AND 1 (= y_j AND y_j), respectively, in position j+1. The ANDing operation applied across the entirety of Q_x and Q_y produces $\{Q_x\} \cdot x_j$ and $\{Q_y\} \cdot y_j$, and operation generation is complete. Both of these operations take place in **Qx-low** and **Qy-low**, respectively. Since *X[j-1]* and *Y[j-1]* are stored in **X-latch** and **Y-latch**, they flow directly into these logic levels.

Note that the bit patterns produced have not included a multiplication by 2, since such multiplication can be accomplished by introducing the patterns into the Arithmetic section shifted one position to the left. Thus, the outputs of the Arithmetic section, which consist of 9-bit fractions with no integer magnitude, arrive in the Operand section as a single integer digit and 8 fractional digits.

On the next half cycle, the Operand section is responsible for the on-the-fly conversion of *X[j-1]* and x_j, *Y[j-1]* and y_j to form *X[j]* and *Y[j]* for use in the next step. Since the range of the input digits of x_j and y_j is [0,1], on-the-fly conversion amounts to simple appending of the arriving input digits onto their converted quantities in position j. This is accomplished by feeding the value of the input digits *xjbar.hold* and *yjbar.hold*, held over for a half cycle in the Alignment Unit, to all positions in their respective latches, then strobing only the proper position j, which is marked by the bit pair traveling across the top of the section.

Physically, the Operand section is divided into bit-slices, with each slice containing a single cell or a pair of cells to accomplish the tasks of every register/latch and logic level. The composition of every slice is consistent with the exception of the leftmost one, where extra circuitry is necessary for two reasons. First, buffers for system signals and arriving

input digits are located in this slice, where their signal levels are restored and strengthened for transmission across the width of the unit. Second, an extra position is added to the shift register in this slice to delay the beginning of the unit's operation until the proper moment in time. Figure 5.2 details the resulting physical structure of the Operand section portion of a bit-slice, with shaded cells representing the extra hardware required in the leftmost slice.

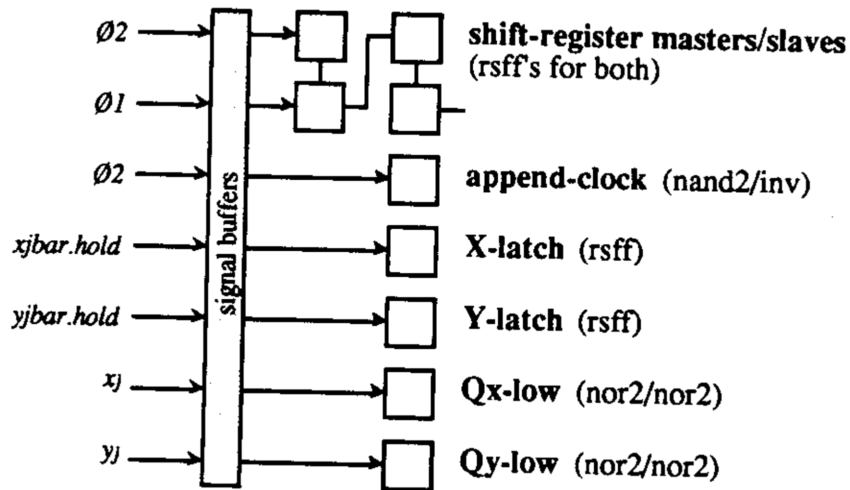


Figure 5.2 - Physical Structure of the Operand Section: Sum-of-Squares

The Arithmetic Section

The Arithmetic section calculates the residual value *W[j]*, and consists of a set of registers and summing logic as detailed in Figure 5.3. The carry-save form of 2 • csfract *W[j-1]* is formed by routing the fractional portions of its partial-sum and carry-save components, held in **W-latch.ps** and **W-latch.sc** respectively, up and to the left one bit position to accomplish the multiplication by 2. $Q_x \cdot x_j$ and $Q_y \cdot y_j$ are introduced from the Operand section above, and are shifted one position to the left to form $2 \cdot Q_x \cdot x_j$ and 2 •

$Qy \cdot yj$ as called for in the recurrence equation. A 4-2 reduction of these quantities in **csa-level1** and **csa-level2** then produces $W[j]$ for use in result digit selection.

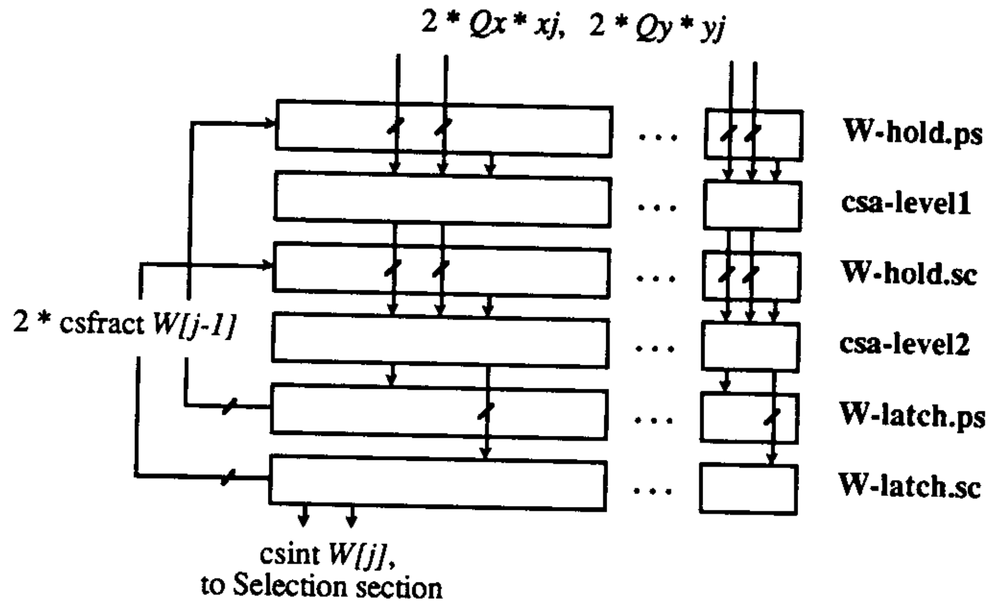


Figure 5.3 - Logical Structure of the Arithmetic Section: Sum-of-Squares

Physically, the Arithmetic section is also divided into bit-slices, with each slice containing 6 individual cells to accomplish the tasks of the four register/latch structures and two 3-2 carry-save reduction levels. The composition of every slice is consistent with the exception of the leftmost one, where extra circuitry is required for system signal buffering. Further, an extra position is added at the left end of the **W-latch.ps** and **W-latch.sc** structures to allow for two potential integer digits in the carry-save forms of $W[j]$. Figure 5.4 details the resulting physical structure of the Arithmetic section portion of a bit-slice, with shaded cells representing the extra hardware required in the leftmost slice.

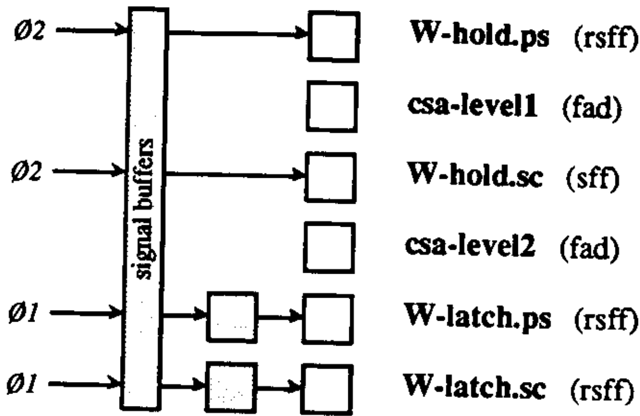


Figure 5.4 - Physical Structure of the Arithmetic Section: Sum-of-Squares

The Selection Section

The Selection section is responsible for the selection and latching of result z_j digits from the integer portion of the residual *W[j]* such that z_j = csint *W[j]*. Figure 5.5 details the logical structure that results. This involves a simple 2-1 reduction in **cpa** of the integer portions of the partial-sum and carry-save components of *W[j]*, each of which contain two integer positions. Because each of these integer portions can have a minimum value of 0 and a maximum value of 3, the result digit selection range is $0 \leq z_j \leq 6$. In this case, no additional logic is required after the reduction to select digits, and hence only a register **z-latch** is included to hold the result. An inverting buffer level placed immediately after **z-latch** inverts the sense of the unit's outputs, which are actually transmitted as *z2jbar*, *z1jbar*, and *z0jbar*.

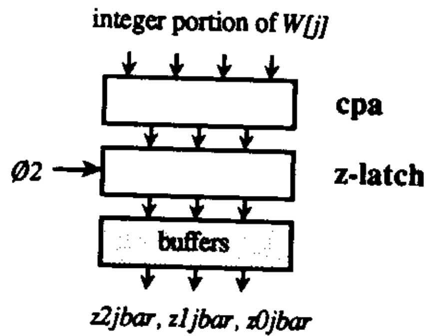


Figure 5.5 - Logical Structure of the Selection Section: Sum-of-Squares

5.3 THE RESULTING STRUCTURE

The physical structure of the entire unit consists of the Operand, Arithmetic, and Selection sections stacked one upon the other and aligned at the left. The functions of the Operand and Arithmetic sections were integrated into type-A, type-B, and type-C bit-slices, with a single type-A slice running along the leftmost edge of the unit, 7 type-B slices inserted along the width, and finally a single type-C slice capping off the right end. The Selection section forms the type-D portion and fits snugly below the leftmost slice.

In the final implementation, a type-A bit-slice measured 1445 microns high by 290 microns wide, a type-B slice had the same height and was 123 microns wide, and a type-C slice again had the same height and was 126 microns wide. The extra width of the type-A slice is due to the extra circuitry incorporated therein and described earlier. The type-D Selection section measured 327 microns by 341 microns.

In the vertical dimension, the height of the unit was 1772 microns, dominated by the height of a bit-slice. In the horizontal dimension, the width of the unit was 1277 microns for this n=8 implementation. To accommodate a larger precision n, an additional

type-B slice, at a width of 123 microns, would be required for each additional fractional digit of precision desired.

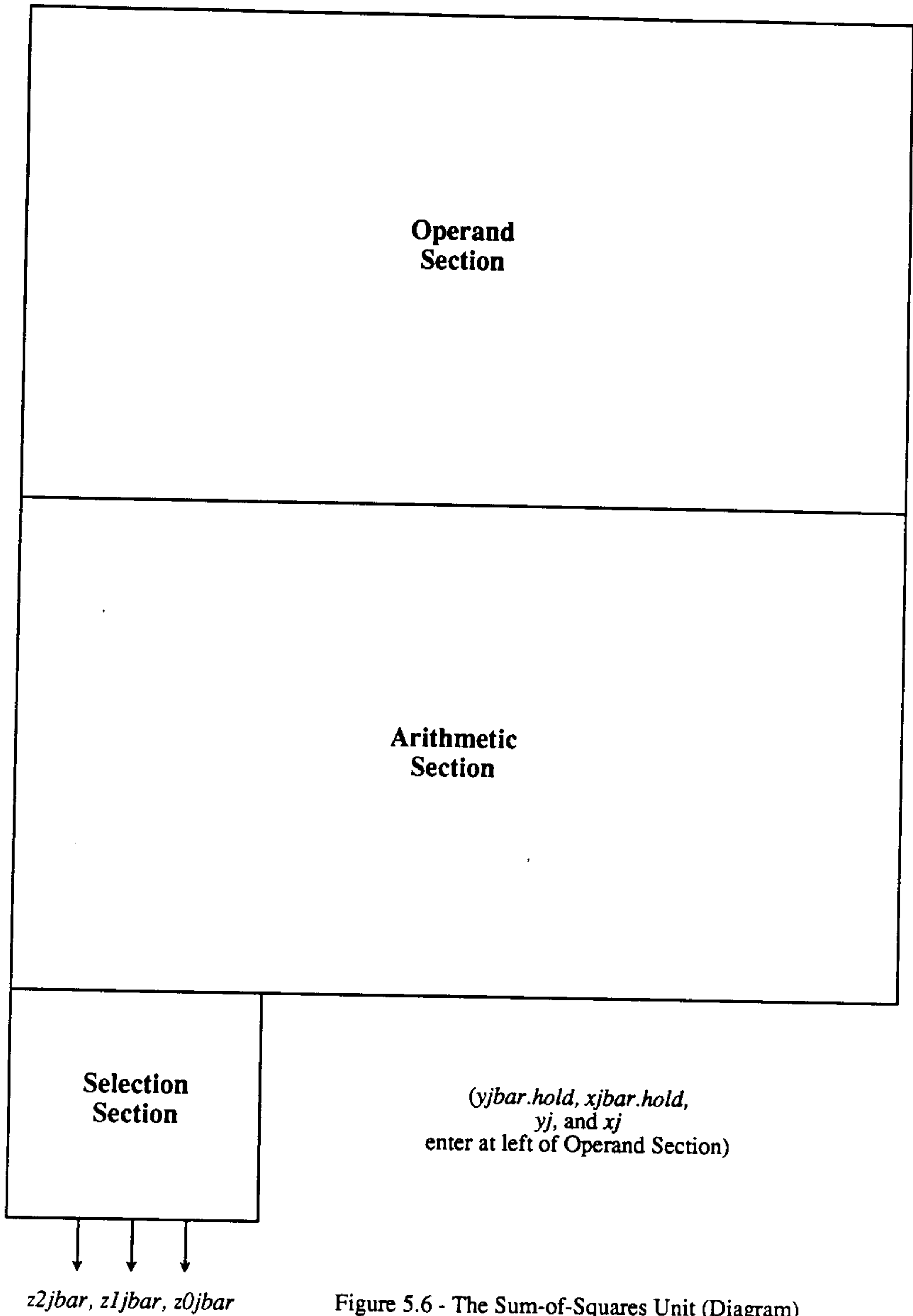


Figure 5.6 - The Sum-of-Squares Unit (Diagram)

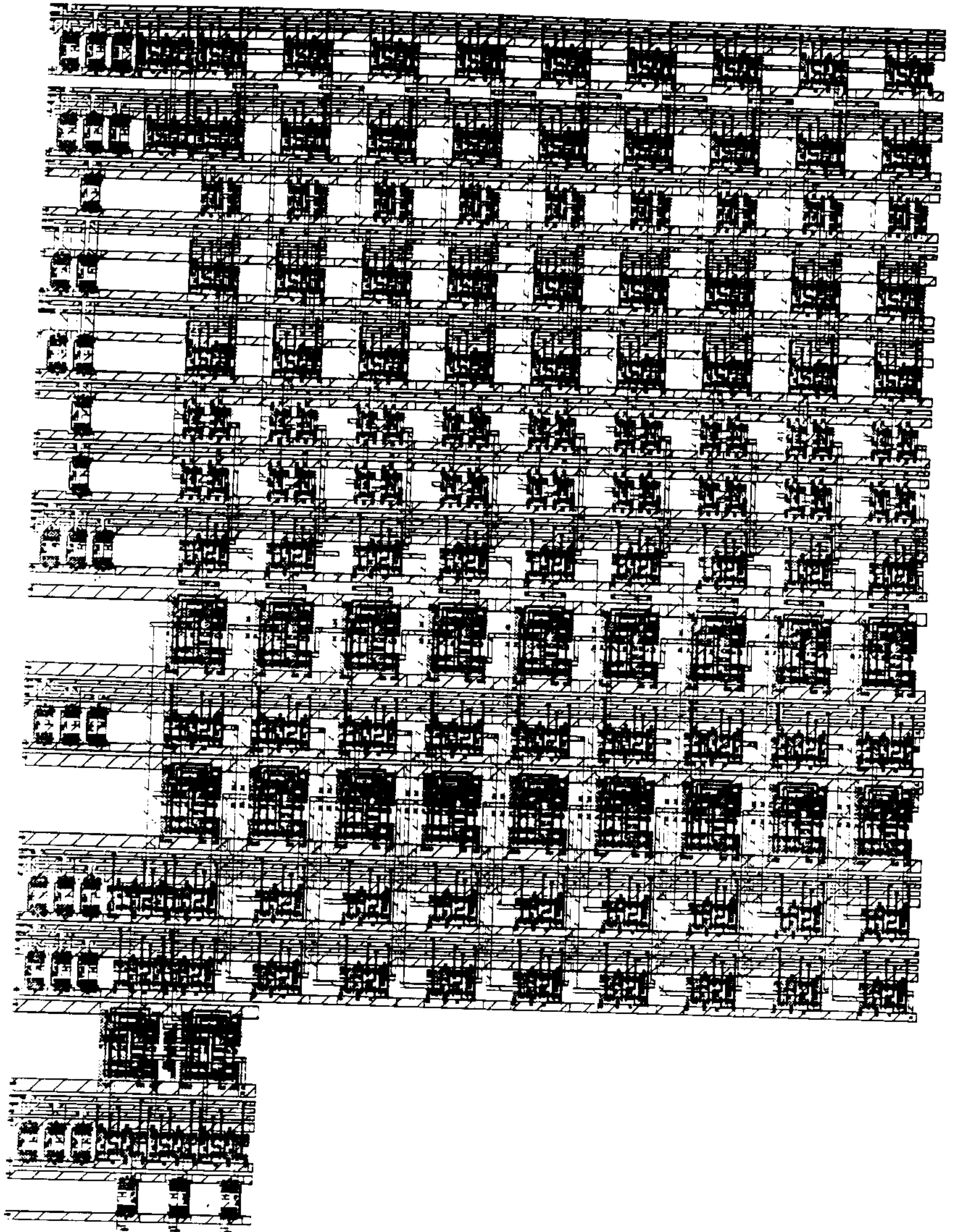


Figure 5.7 - The Sum-of-Squares Unit (Plot)

CHAPTER 6

SQUARE-ROOT UNIT ARCHITECTURE

The Square-Root Unit computes the square root of $z = x^2 + y^2$, whose digits are received serially as *z2jbar*, *z1jbar*, and *z0jbar* from the Sum-of-Squares Unit above. The architecture of this unit is also based upon the generalized model of an online unit outlined in Chapter 2. The unit's output consists of a digit stream d_j, $d_j \in [-1,0,1]$, such that the equation $d = z^{1/2}$ is satisfied, with n=8 fractional digits of precision. Further, the algorithm used in the immediately succeeding Division Units requires in every step the full parallel form of the on-the-fly converted value $D[j]$, which is produced from d_j and stored internally in the Square-Root Unit.

6.1 ALGORITHMIC ANALYSIS

The structure of this algorithm also matches that of the generalized model of an online algorithm outlined in Chapter 2. In this case, $R[j]$ is the residual whose value is repeatedly generated at every step, d_j represents the result digit selected during step j, and $D[j]$ represents the full parallel, on-the-fly converted value produced from the d_j result digit stream. Since $d_j \in [-1,0,1]$, d_j may assume a negative value, and true on-the-fly conversion involving two registers representing $D[j]$ and $D^*[j]$ is required.

Again, for the sake of brevity, the algorithm is annotated at right and some quantities that appear in the recurrence equation are given pseudonyms which are used later to avoid repeating complex formulae.

```
algorithm Square-Root                                                    (6.1)
    D[-2] = 00.00000000;
    D*[-2] = 00.00000000;                         /* INITIALIZE conversion regs */

    d-1 = 0;                                   /* INITIALIZE result digit stream */

    R[-5] = 000.0000000;                             /* INITIALIZE the residual */

    for j = -4, -3, ... , n-1 = 7 {            /* GENERATE recurrence value */
        if (j ≤ -2) then
            R[j] = 2 • R[j-1] +
                   { zj+4 • 2^-4 };                               /* {Qz} */
        else
            {R[j] = 2 • R[j-1] +
                   { zj+4 • 2^-4 } +                              /* {Qz} */
                   2 • { -dj • D[j-1] + dj^2 • 2^-(j+1) };    /* 2 • {Qd} */
            Rhat-star[j] = Rhat[j] + zj+5 • 2^-5;     /* SELECT result digit */
            dj+1 = dsel Rhat-star[j];
            D[j] = convert (D[j-1], dj)};             /* UPDATE local values */
    }
end Square Root
```

While this algorithm may appear at first to differ from the generalized model due to the **if** ... **then** ... **else** construct, from an implementation standpoint only the statements in the **else** clause, which do match the standard form, are actually executed during every step. While $j \leq -2$, the value of Q_d, and hence of $2 \cdot \{Q_d\}$, is held at 0, and the recurrence equation defaults to the form given in the **if** clause. The values of the digits d_j prematurely

selected during these steps are ignored, and on-the-fly conversion to produce $D[j]$ does not commence until after the first valid digit is produced.

6.2 LOGICAL AND PHYSICAL DESIGN OF SUB-SECTIONS

The Operand Section

The Operand section of the unit contains the circuitry required to generate those operands of the recurrence equation, namely $\{Qd\}$ and $\{Qz\}$, that are newly-introduced at every step. Their values depend upon the on-the-fly converted value $D[j-1]$ of the previous result d_j digits produced and the z_j input digit stream received from the Sum-of-Squares Unit, respectively. Again, this task can be reduced to producing the binary bit patterns that represent these quantities, then ensuring that they are fed into the Arithmetic section aligned at the proper position with respect to the binary point. Converters are physically placed in the Operand section, and a feedback loop from the Selection section returns result d_j digits here for conversion. Figure 6.1 details the logical structure of the Operand section.

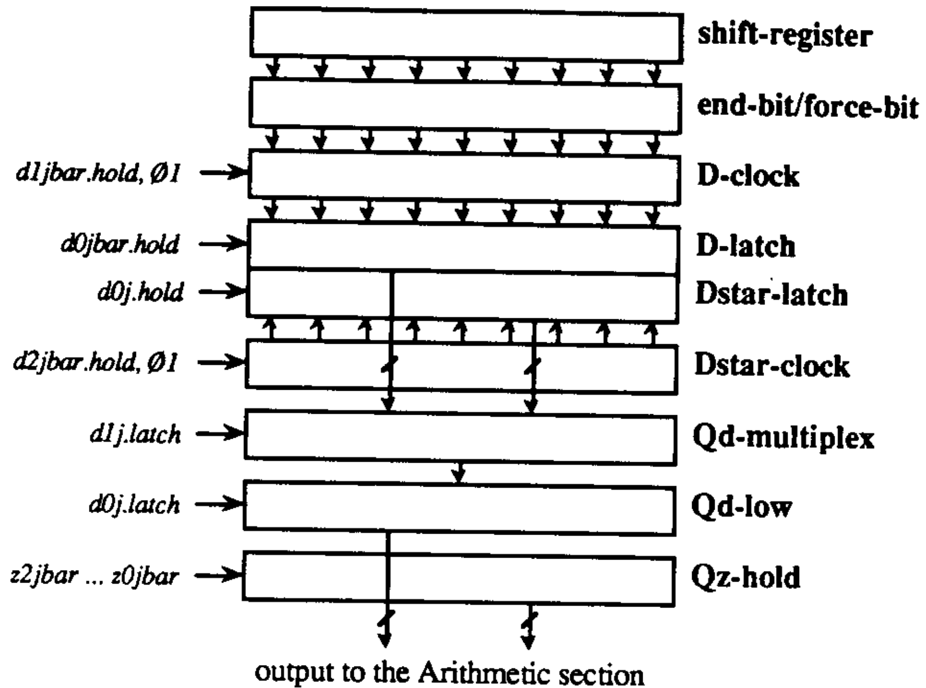


Figure 6.1 - Logical Structure of the Operand Section: Square-Root

The section operates as follows. At the beginning of the chip's operation, the GRESET signal is used to preset all latches to 0. The input to the leftmost master flip-flop in the shift-register is wired directly to a 1, and hence a bit train is seeded that advances across the section at the rate of one position per step of the algorithm. A bit train is used, rather than a traveling bit pair, so that position j, at the forward edge of the train, as well as all positions $\leq j$ can be discerned from each other and from uncrossed positions further ahead.

The section's first objective is to produce the bit patterns corresponding to Q_d, subject to two sets of constraints. First, it was necessary to ensure that the value of Q_d remained equal to 0 for the first three steps of the algorithm, to ensure that the recurrence equation given in the **else** clause would default to the form given in the **if** clause. To do

so, it was necessary to suppress on-the-fly conversion and production of *D[j]* to maintain its value at 0, and also to ignore the premature d_j values received from the Selection section. Both tasks were accomplished by introducing delay stages into **shift-register**, since both on-the-fly conversion and d_j insertion were dependent on the presence of the bit train in slice *j*.

Second, the Q_d bit pattern must be formed, where Q_d may take on one of three different values (and hence one of three different patterns) depending upon the value of the d_j input. All of the hardware found in the Operand section that was not dedicated to on-the-fly conversion was required to produce these bit patterns. Figure 6.2 details these patterns as a function of the d_j input.

Qd =	[*Dbar* , 1 , 1(*j*+1) , 0 , 0 , ... , 0]	if	*d(j)* = 1
Qd =	[0 , ... , 0 , 0(*j*+1) , 0 , 0 , ... , 0]	if	*d(j)* = 0
Qd =	[*Dstar* , 1 , 1(*j*+1) , 0 , 0 , ... , 0]	if	*d(j)* = -1

Figure 6.2 - Bit patterns produced as a function of d_j: Square-Root

For the $d_j = 1$ case, the bit train across the top of the section is used to selectively invert the bits of *D[j-1]*, held in **D-latch**, to form *Dbar* (= *D[j-1]* logically inverted) as shown. The bit train covers the positions which have already been crossed, and hence marks exactly those positions of **D-latch** which contain significant on-the-fly converted digits to be inverted. Any trailing zeros in *D[j-1]* which lie to the right of significant digits are passed through unaffected. For the $d_j = -1$ case, *D*[j-1]*, held in **Dstar-latch**, may be used directly as *Dstar* (= *D*[j-1]* true). The logic required to select either *D[j-1]* or *D*[j-1]*, and further to invert the significant bits of *D[j-1]* as required is contained in **Qd-multiplex**. To complete the pattern, logic in **force-bit** detects the forward edge of the bit

train corresponding to position j, and the two low order bits of Q_d in positions j and $j+1$ are forced to 1.

As in the Sum-of-Squares case, the Q_d bit pattern produced is introduced into the Arithmetic section shifted one position to the left to accomplish the multiplication by 2. Thus, the Q_d output, which consists of 2 integer and 8 fractional digits, arrives in the Arithmetic section as 3 integer and 7 fractional digits. The binary point of the Arithmetic section falls one position to the right of the binary point maintained in the Operand section.

Production of the other major operand of the recurrence, Q_z, consists of latching the z_{j+4} digits arriving from the Sum-of-Squares unit in **Qz-hold** in the proper position to accomplish a multiplication by 2^{-4}, while all other positions are latched with a constant 0. These are then held across the same half cycle as Q_d for production of the next $R[j]$ in the Arithmetic section.

On the next half cycle, the Operand section is responsible for the on-the-fly conversion of $D[j]$ using the result d_j digit from the previous step in the algorithm. Since this d_j value may be negative, true on-the-fly conversion using two registers is required. **D-latch** and **Dstar-latch**, respectively, are used to convert $D[j]$ and $D^*[j]$. Parallel loading between the conversion registers is triggered according to the value of d_j when non-zero. When $d_j = 1$, positions 0 .. j-1 of $D^*[j-1]$ are loaded with the values residing in the same positions of $D[j-1]$, and when $d_j = -1$, loading takes place in the reverse direction.

Regardless of loading, a digit must be appended onto both quantities in position j, which is marked by the logic in end-bit that detects the forward edge of the bit train. When $d_j = 1$ or -1, a 1 is appended onto $D[j-1]$ and a 0 onto $D^*[j-1]$, and when $d_j = 0$ a 0 is

appended to $D[j-1]$ and a 1 to $D^*[j-1]$. This completes preparation of $D[j]$ for use as $D[j-1]$ in the next step of the algorithm.

Physically, the Operand section is again broken into bit-slices. More complex circuitry is required in each slice than in the Sum-of-Squares Unit because of the extra logic required for bit train mapping, true on-the-fly conversion, and more complex bit-pattern generation. Figure 6.3 details the physical structure of the Operand section portion of a bit-slice, with shaded cells representing the extra circuitry required in the leftmost slice.

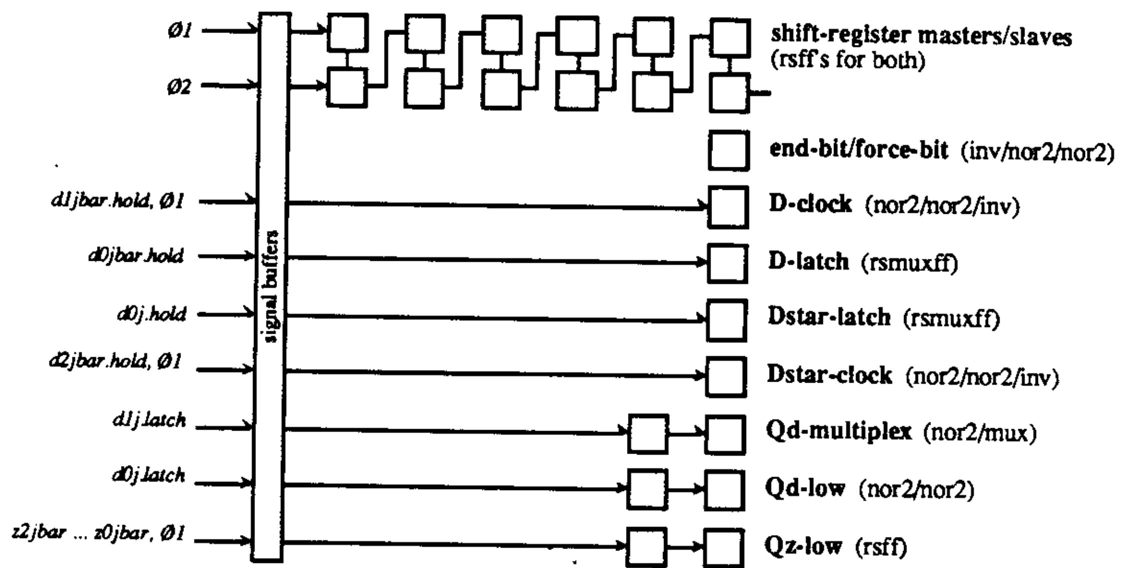


Figure 6.3 - Physical Structure of the Operand Section: Square-Root

shift-register is responsible for carrying the bit train across the entire section. Note the extra delay stages included in the leftmost slice; these are required to delay on-the-fly conversion until valid d_j digits are produced by the algorithm. **end-bit/force-bit** contains the logic required to detect the leading edge of the bit train for use in conversion and Q_d production, respectively. **D-clock** and **Dstar-clock** contain the logic required to

trigger and synchronize parallel loading between the conversion registers **D-latch** and **Dstar-latch**. **Qd-multiplex** and **Qd-low** select either *D[j-1]* or *D*[j-1]* from **D-latch** or **Dstar-latch**, then append a pair of 1's to produce the Q_d bit pattern. Finally, **Qz-hold** holds the *zj+4* value arriving from the Sum-of-Squares Unit across the next half cycle, when *R[j]* is produced.

The Arithmetic Section

The Arithmetic section is responsible for actual calculation of the residual value *R[j]*, and consists of a set of registers and summing logic as detailed in Figure 6.4. The carry-save form of 2 • *R[j-1]* is formed by routing its partial-sum and carry-save components, held in **R-latch.ps** and **R-latch.sc**, respectively, up and to the left one bit position to accomplish the multiplication by 2. Q_d and Q_z are introduced from the Operand section above, with Q_d shifted one position to the left to form 2 • $\{Q_d\}$. A 4-2 reduction in **csa-level1** and **csa-level2** then produces *R[j]*. Because Q_z only extends to the 2-4 position, that portion of **csa-level1** to the right of Q_z is actually not required. However, since the cells composing **csa-level1** in the bit-sliced design were still present, they were included and one input of the full-adders in these positions was wired to 0.

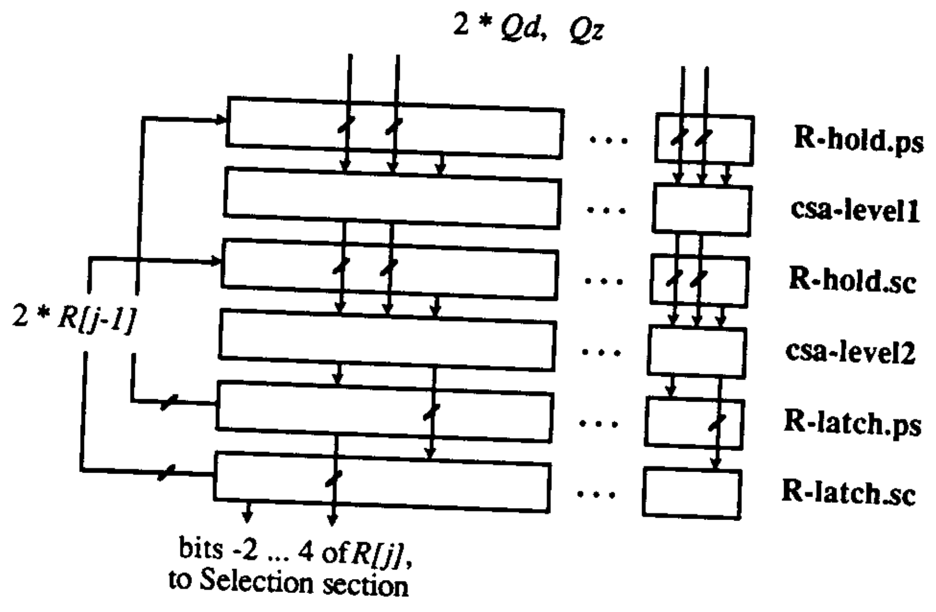


Figure 6.4 - Logical Structure of the Arithmetic Section: Square-Root

Physically, the Arithmetic section is divided up into bit-slices in exactly the same manner as for the Sum-of-Squares Unit with the exception of the leftmost slice. Here, extra circuitry is required both for system signal buffering and to provide an extra integer bit position to cover the entire range of $R[j]$ values possible. In total, three integer and four fractional positions are transmitted to the Selection section. Figure 6.5 details the resulting physical structure of the Arithmetic section portion of a bit-slice, with shaded cells representing the extra hardware required in the leftmost slice.

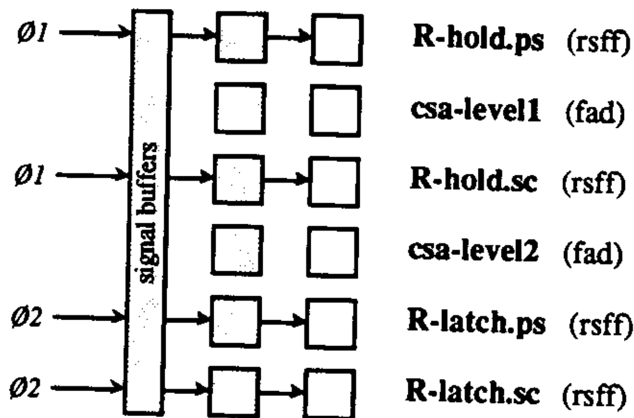


Figure 6.5 - Physical Structure of the Arithmetic Section: Square-Root

The Selection Section

The Selection section is responsible for the selection and latching of result d_j digits. Selection is performed on the quantity *Rhatstar[j]*, which represents the sum of a low-precision estimate *Rhat[j]* of *R[j]* plus a factor $z_{j+5} \cdot 2^{-5}$ (see the algorithm 6.1). The number of fractional bits t of *R[j]* which must be included in the estimate *Rhat[j]* varies according to the online delay p_{sqr} that the designer is willing to tolerate. For this implementation, an estimate *Rhat[j]* having $t = 3$ fractional bits was chosen, which corresponds to an online delay of $p_{sqr} = 4$.

The selection function **dsel** that determines result d_j digits operates on the value of *Rhatstar[j]* and is given in Figure 6.6. The range of *Rhatstar[j]* is [-2.5, 3.5], requiring three integer digits of *Rhatstar[j]* (and hence of *R[j]*), while **dsel** requires precision to 1/4's, mandating that two fractional bits and therefore a total of 5 bits be examined for selection. The selected d_j's are represented in binary as *d1j* and *d0j*. In addition, another specially-defined binary digit *d2j* is produced for convenience and is set equal to 1 in the d_j = 1 case only.

Range: Range R $^{\text{hat-star}}$[j] = [-2.5,3.5], therefore 3 integer bits needed for selection

Precision: to 1/4's, therefore 2 fractional bits needed for selection

dsel Selection: Table

					R $^{\text{hat-star}}$ [j	dj		d2j	d1j	d0j
0	1	1	1	0						
.	.	.	.	.						
0	0	0	0	1						
0	0	0	0	0	≥ 0	1	=	1	0	1
1	1	1	1	1	= -1/4	0	=	0	0	0
1	1	1	1	0						
1	1	1	0	1	≤ -1/2	-1	=	0	1	1
.	.	.	.	.						
1	0	1	1	0						

Figure 6.6 - The **dsel** Selection Function: Square-Root

The resulting logical structure of the Selection section is detailed in Figure 6.7. First, a 3-2 reduction in **3-2 reduce** of *Rhat[j]* (equal to **R-latch.ps** + **R-latch.sc** to t = 3 fractional digits of precision) and $z_{j+5} \cdot 2^{-5}$ is required to include the current input digit. The resulting carry-save forms are then assimilated in **cpa** into a single binary value equal to *Rhatstar[j]*. **d-latch** contains the selection logic as well as the latch structures required to select and hold d_j as *d2j.latch*, *d1j.latch*, and *d0j.latch*. Finally, **d-hold** serves as a delay latch to hold the d_j value produced for feedback to the on-the-fly converters of the Operand section during the next half cycle.

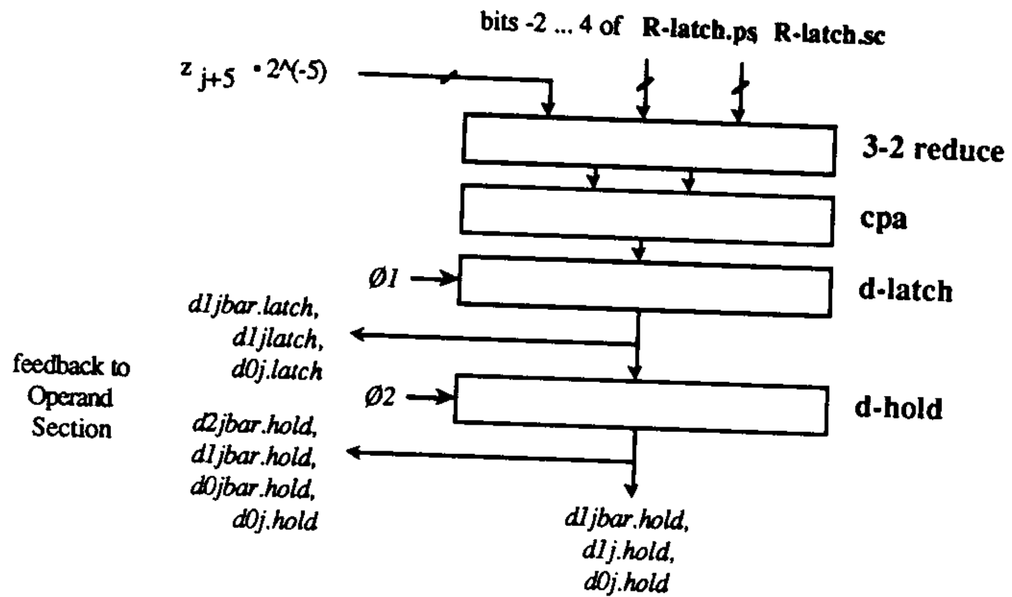


Figure 6.7 - Logical Structure of the Selection Section: Square-Root

6.3 THE RESULTING STRUCTURE

As was the case in the Sum-of-Squares unit, the physical structure of the entire unit consists of the Operand, Arithmetic, and Selection sections stacked one upon the other and aligned at the left. The Operand and Arithmetic sections were integrated into type-A, type-B, and type-C bit-slices, with a single type-A slice running along the leftmost edge of the unit, 7 type-B slices inserted along the width, and finally a single type-C slice capping off the right end. The Selection section again fits snugly underneath the leftmost slice as the type-D portion.

One difference in this case was that 3 special type-B slices, corresponding to bit positions 2, 3, and 4 of the *R[j]* residual, were required in order to latch and properly align Q_Z immediately above the Arithmetic section. These slices differed from the others in that

the flip-flops corresponding to these positions were wired to accept as inputs $z2j$, $z1j$, and $z0j$ of z_{j+4}, shifted to correspond to the multiplication by 2^{-4} as called for in the algorithm. The other slices, by contrast, had their flip-flip inputs wired to 0.

In the final implementation, a type-A bit-slice measured 2028 microns high by 597 microns wide, all type-B slices had the same height and were 147 microns wide, and a type-C slice again had the same height and was 213 microns wide. The extra width of the type-A slice is due to the extra circuitry incorporated therein and described earlier. The type-D Selection section measured 701 microns by 1248 microns.

In the vertical dimension, the height of the unit was 2729 microns, dominated by far by the height of a bit-slice. In the horizontal dimension, the width of the unit was 1839 microns for this n=8 implementation. To accommodate a larger precision n, an additional type-B slice, at a width of 147 microns, would be required for each additional fractional digit of precision desired.

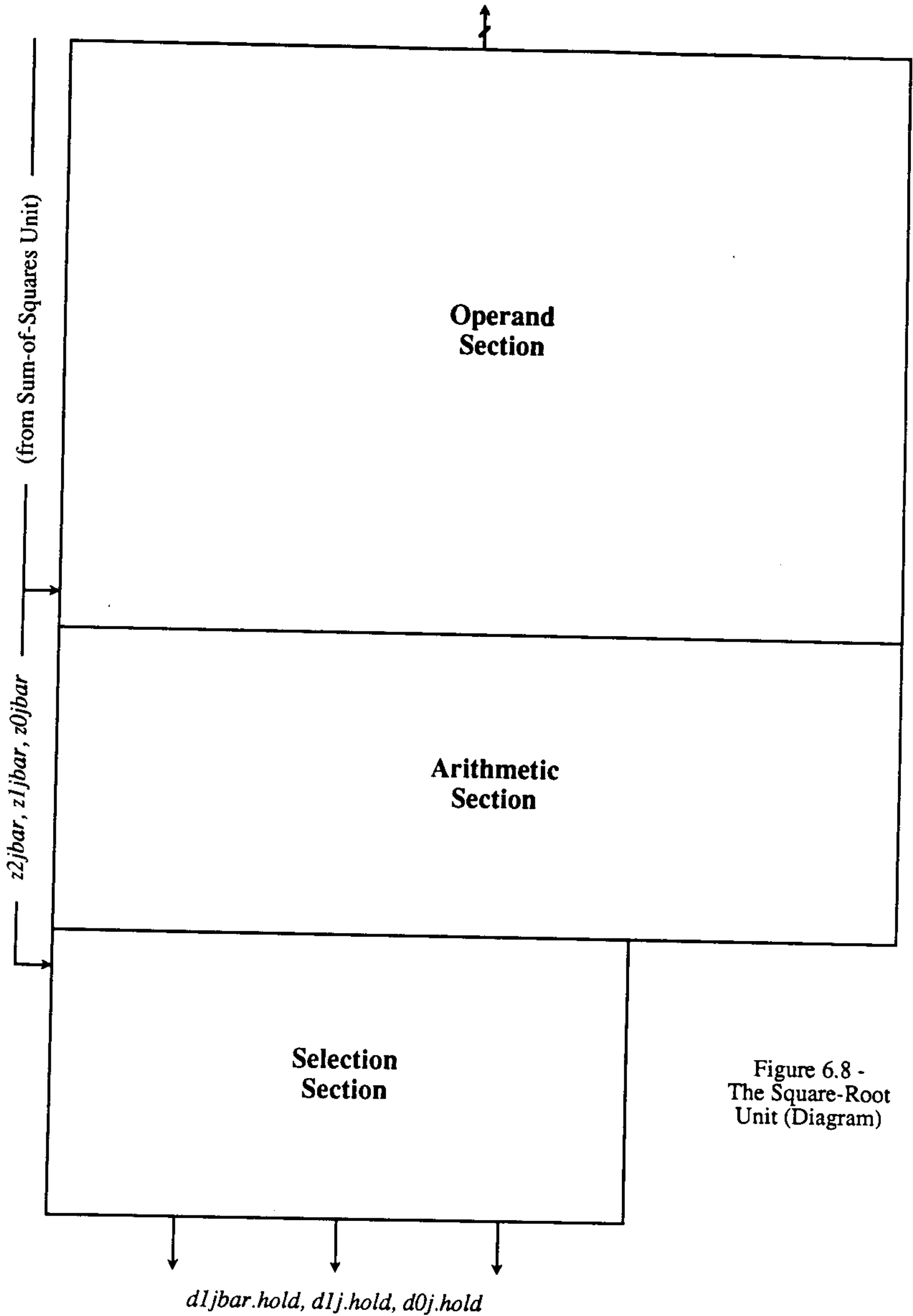


Figure 6.8 -
The Square-Root
Unit (Diagram)

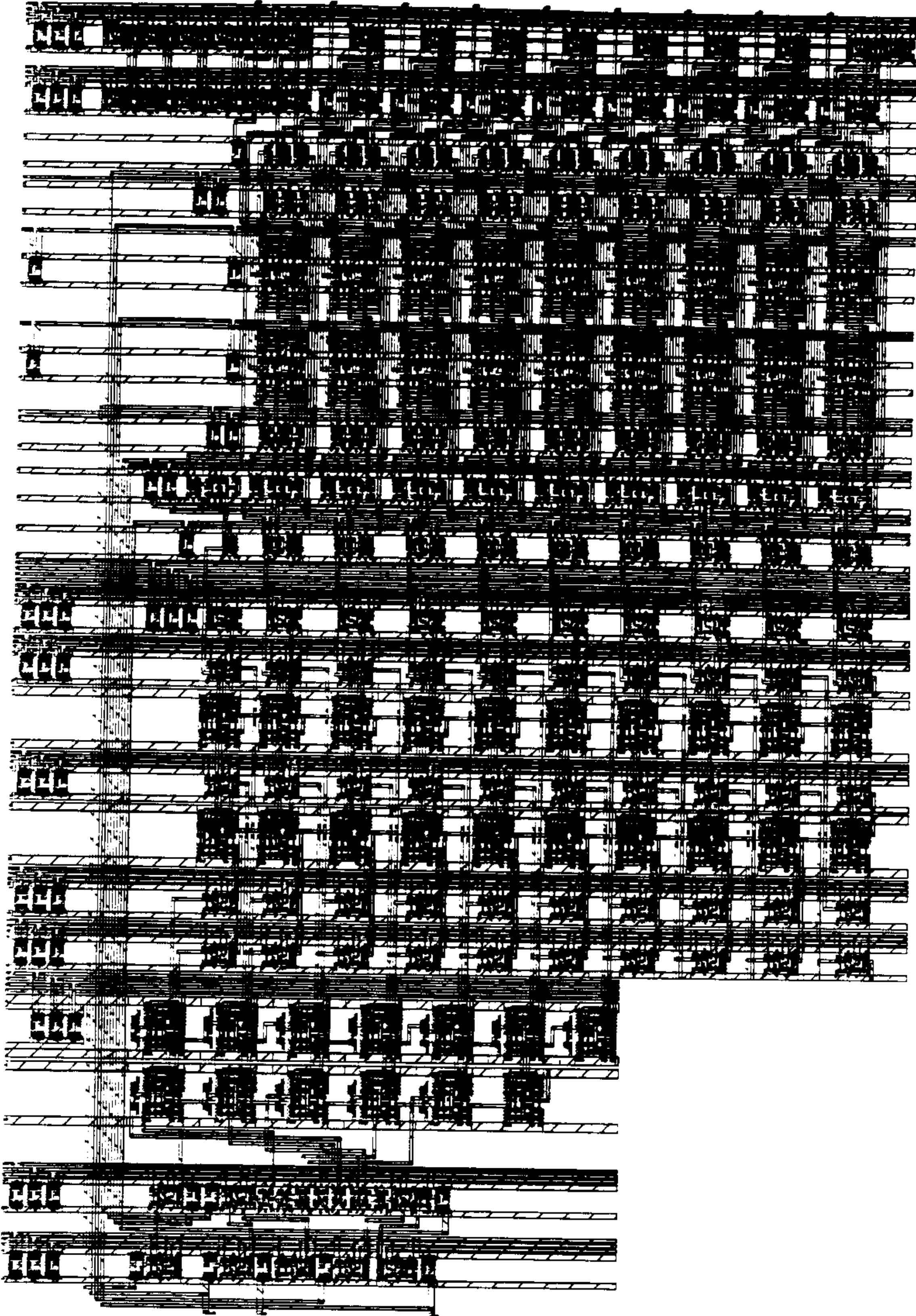


Figure 6.9 - The Square-Root Unit (Plot)

CHAPTER 7

DIVISION UNIT ARCHITECTURE

The twin Division Units, the final major units in the calculation chain, are charged with computation of the quotients $c = g/d$ and $s = h/d$. Inputs to the units consist of *dj.hold*'s, whose digits are received serially as *d1jbar.hold*, *d1j.hold*, and *d0j.hold*, *D[j]*'s, received in full-parallel as *D0[j]bar* ... *D8[j]bar* and representing the on-the-fly converted value of all d_j digits, and the initial values of g and h. Both quantities are produced in and obtained from the Square-Root Unit, with the *.hold* suffix indicating that a half-cycle delay was introduced between the production of these d_j's in the previous unit and their consumption here. The division algorithm is unique in that it calls for a non-zero initial condition in one of the variables involved. For the $c = g/d$ unit, g is pre-loaded into a register within the Arithmetic section, and likewise for the $s = h/d$ unit h is pre-loaded in an Arithmetic section register.

For the purpose of describing a unit's operation, the $s = h/d$ case in particular will be used since the operation of the unit producing $c = g/d$ is analogous. For this case, the unit's output consists of a digit stream s_j, $s_j \in [-1,0,1]$, such that the equation $s = h/d$ is satisfied, with $n = 8$ fractional digits of precision. Also, because this is the final unit in the calculation chain, the full parallel on-the-fly converted value *S[j]* of the s_j's produced is routed to output pads and may be sampled at any point during the chip's operation.

7.1 ALGORITHMIC ANALYSIS

The structure of this algorithm similarly matches that of the generalized model of Chapter 2. In this case, *W[j]* is the residual whose value is repeatedly generated at every step, s_j represents the result digit selected during step *j*, and *S[j]* represents the full parallel, on-the-fly converted value produced from the s_j result digit stream. Since $s_j \varepsilon$ [-1,0,1], s_j may assume a negative value, and true on-the-fly conversion involving two registers representing *S[j]* and *S*[j]* is required. In addition, an internal quantity *P[j]* is kept within the unit and is used during the opposite half cycle to assist in the production of *W[j]*.

One caveat in the algorithm below is that internally, rather than producing $s = h/d$ as dictated by inputs from the external world, the value of $s = 2^{-3} \cdot h/d$ is produced. This reflects a choice of an online delay $p_{div} = 3$, and the 2^{-3} factor is incorporated into the initial loading of h as $h \cdot 2^{-3}$. To compensate for this multiplication, the *S[j]* value routed off-chip is shifted left three bit positions, and thus the true $s = h/d$ is communicated to the outside world.

```
algorithm Division                                                      (7.1)

        S[0] = 00.00000000000;             /* INITIALIZE conversion regs */
        S*[0] = 00.00000000000;
        so = 0;                            /* INITIALIZE result digit stream */

        P[0] = 00.000 h1 ... h8;           /* INITIALIZE the residual */
        D[0] = do;                         /* INITIALIZE other inputs */

        for j = 1, 2, ... , n+3 = 11 {     /* GENERATE recurrence value */

                {W[j] = 2 • P[j-1] +
                        { -dj • S[j-1] };                /* {Qs} */
                sj = qsel W[j];                          /* SELECT result digit */

                P[j] = W[j] +                            /* UPDATE local values */
                        { -sj • D[j] } };                /* {Qd} */
                S[j] = convert (S[j-1], sj);

        }
end Division
```

7.2 LOGICAL AND PHYSICAL DESIGN OF SUB-SECTIONS

The Operand Section

The Operand section of the unit contains the circuitry required to generate the Q_s operand of the recurrence equation. The value of Q_s depends upon the $d_j.hold$ digit received from the Square-Root Unit and the on-the-fly converted value $S[j-1]$ of previous result s_j digits produced within this unit. Since $d_j.hold \in [-1,0,1]$, the task of producing the binary bit pattern representing Q_s reduces to multiplying $S[j-1]$ by 1, 0, or -1. Converters are physically placed within the Operand section again, and a feedback loop from the Selection section returns result s_j digits here for conversion. Figure 7.1 details the resulting structure of the Operand section.

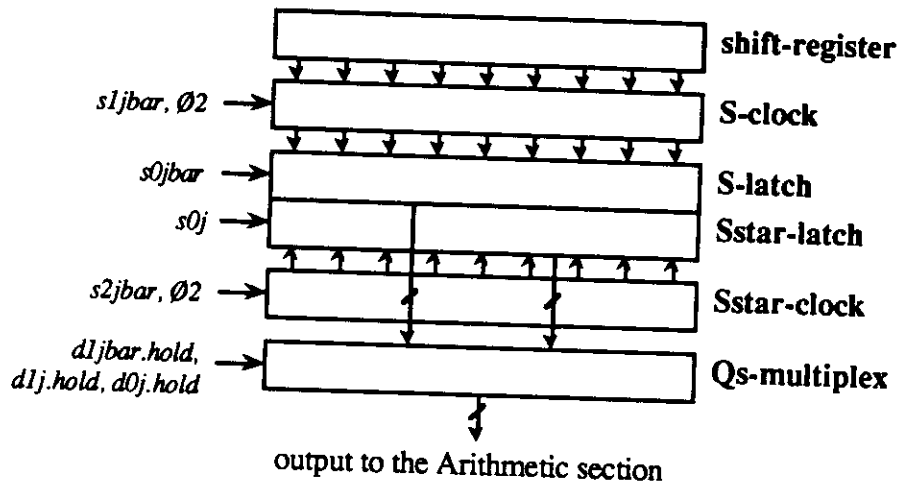


Figure 7.1 - Logical Structure of the Operand Section: Division

The section operates as follows. At the beginning of the chip's operation, the GRESET signal is used to preset all latches to 0, except for the leftmost master flip-flop in **shift-register**, which is seeded with a 1. This 1 is the source of a bit pair that travels across the section marking the current position *j* exactly as in the Sum-of-Squares Unit.

The section's first objective is to produce the bit pattern corresponding to Q_S. This is accomplished by multiplying *S[j-1]* by 1, 0, or -1 corresponding to *dj.hold* values of -1, 0, or 1, respectively. Figure 7.2 details the patterns that result as a function of the *dj.hold* input.

Qs = [*Sbar* , 1 , .. , 1 , ... , 1]	*d(j).hold* =	1 (1 inserted)
Qs = [0 , 0 , ... , 0 , ... , 0]	*d(j).hold* =	0
Qs = [*S* , 0 , ... , 0 , ... , 0]	*d(j).hold* =	-1

Figure 7.2 - Bit patterns produced as a function of d_j*.hold*: Division

On the next half cycle, the Operand section is responsible for the on-the-fly conversion of *S[j]* using the result s_j digit selected during this step of the algorithm. Since this s_j value may be negative, true on-the-fly conversion using two registers to convert *S[j]* and *S*[j]* is performed. Parallel loading between these registers is triggered by the value of s_j if non-zero and operates exactly as between the registers of the Square-Root Unit's converter.

Physically, the Operand section is again broken into bit-slices. Figure 7.3 details the physical structure of the Operand section portion of a bit-slice, with shaded cells representing the extra circuitry required in the leftmost slice.

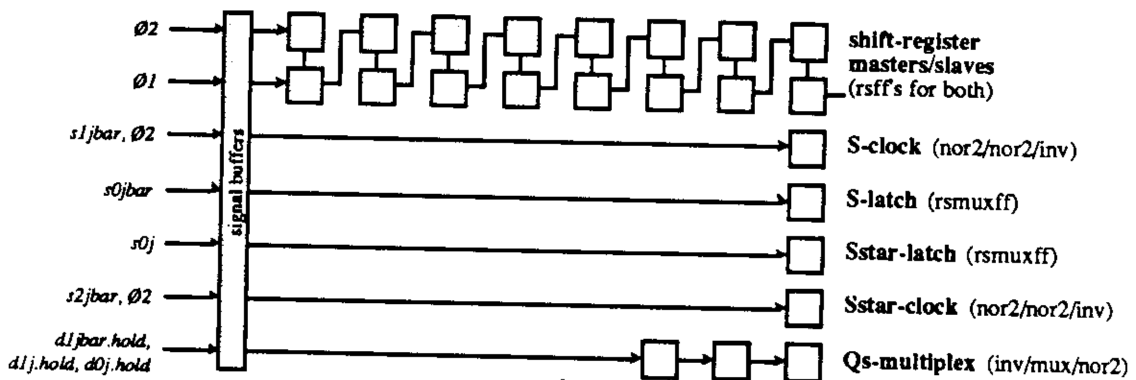


Figure 7.3 - Physical Structure of the Operand Section: Division

shift-register is responsible for carrying the bit pair across the entire section. Seven delay stages are included in the leftmost slice; these are required to delay on-the-fly conversion until valid s_j digits are produced. **S-clock** and **Sstar-clock** contain the logic required to trigger and synchronize parallel loading between the conversion registers S-

latch and **Sstar-latch**. Finally, **Qs-multiplex** selects *S[j-1]* from **S-latch**, then multiplies it by 1, 0, or -1 to produce the Q_d bit pattern.

The Arithmetic Section

The Arithmetic section is responsible for actual calculation of the residual value *W[j]*. In the Sum-of-Squares and Square-Root Units, this task consisted of reducing three operands of a single recurrence equation to produce the carry-save form of the residual. However, in the case of the Division Unit, this task consists of reducing a recurrence equation having <u>two</u> operands to produce the carry-save form of the residual *W[j]* during the first half-cycle, and then reducing the value so produced along with another operand, Q_d, to form *P[j]* (an internal quantity) during the next half-cycle. *P[j]* serves in turn as *P[j-1]* in the production of *W[j]* during the next step of the algorithm. Equations 7.2 (excerpted from 7.1) detail the events that transpire during each half phase of an algorithmic step.

```
W[j] = 2 • P[j-1] +                     /* first (Ø2 -> Ø1) phase */
       { -dj • S[j-1] };                               /* Qs */
sj = qsel W[j];

P[j] = W[j] +                           /* second (Ø1 -> Ø2) phase */      (7.2)
       { -sj • D[j] };                                 /* Qd */
S[j] = convert (S[j-1], sj);
```

Figure 7.4 details the logical structure of the Arithmetic section. During step *j*, the partial-sum and carry-save forms of *P[j-1]*, held in **P-latch.ps** and **P-latch.sc**, are routed up and to the left one position to accomplish the multiplication by 2. They are then combined with Q_s in **csa-level1** to form the partial-sum and carry-save forms of *W[j]*, kept in **W-latch.ps** and **W-latch.sc**. In the next half-cycle, Q_d is formed (according to

the value of the s_j selected) in **Qd-multiplex** in the exact same manner as Q_s was formed in the Operand section. Q_d and the partial-sum and carry-save forms of $W[j]$ are then combined to form $P[j]$ in **P-latch.ps** and **P-latch.sc**.

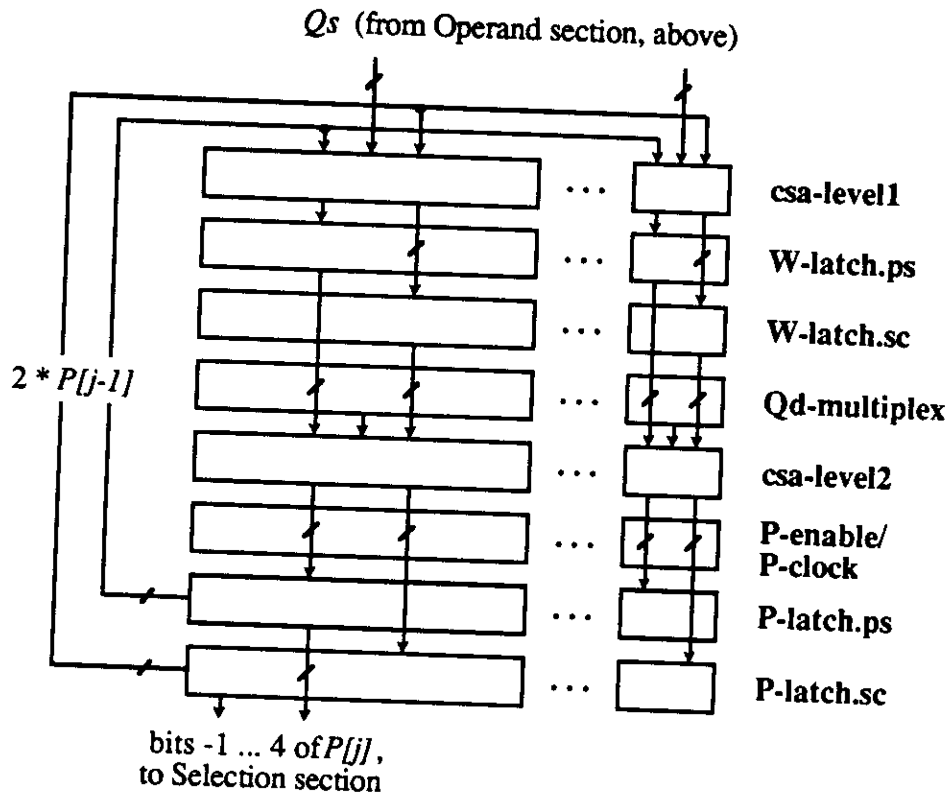


Figure 7.4 - Logical Structure of the Arithmetic Section: Division

The **P-enable/P-clock** level is included to provide gated clocks to the latches of **P-latch.ps** and **P-latch.sc**. Normally, the clocks to these registers are allowed to run free. Because latch inputs typically remain zero until a unit begins to produce significant digits, no added clocking logic is required. However, a special condition forces the inclusion of gated clocks in this case. The **P-latch.ps** register is utilized during the first step of the algorithm to latch and hold the initial value of h input from the external world,

and a single clock pulse must be applied at the proper moment to accomplish this. Thereafter, clocks must be inhibited until the global time step during which the first valid result digit is selected, otherwise the value held in **P-latch.ps** would be churned through a number of invalid iterations of the Division algorithm, producing erroneous s_j digits in advance of the proper moment.

Physically, the Arithmetic section is divided up into bit-slices in the same manner as for both previous units. As usual, the leftmost slice contains extra circuitry for system signal buffering, and in addition provides two extra integer bit positions to cover the entire range of *P[j]* values possible. In total, two integer and four fractional positions are transmitted to the Selection section. Figure 7.5 details the resulting physical structure of the Arithmetic section portion of a bit-slice, with shaded cells representing the extra hardware required in the leftmost slice.

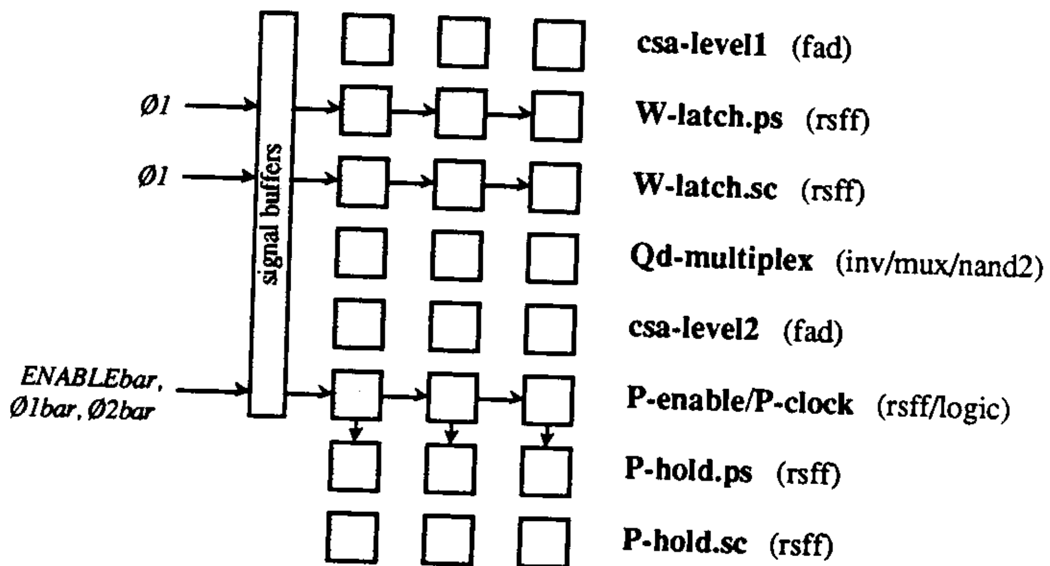


Figure 7.5 - Physical Structure of the Arithmetic Section: Division

The Selection Section

The Selection section is responsible for the selection and latching of result s_j digits and implements the selection function **qsel**. According to the algorithm, selection is performed on the value of the *W[j]* residual produced during the first half-cycle of an algorithmic step. However, it was noted that based upon the range and precision requirements of **qsel**, a low-precision estimate *What[j]* of *W[j]* could be used for selection instead, where *What[j]* requires only the values of *P[j-1]* and d_j, and not *S[j-1]*. Figure 7.6 details the operation of the selection function **qsel** based upon the value of *What[j]*. Note again that a specially-defined binary digit *s2j* is produced for convenience and is set equal to 1 in the $s_j = 1$ case only.

Range: Range W ^{hat} [j] = [-23/8,23/8), therefore 3 integer bits needed for selection

Precision: to 1/8's, therefore 3 fractional bits needed for selection

qsel Selection: Table							W ^{hat} [j	sj		s2j	s1j	s0j
	0	1	0	1	1	1						
	.	.	.	.	.	.						
	0	0	0	0	1	1						
	0	0	0	0	1	0	≥ 1/4	1	=	1	0	1
	0	0	0	0	0	1						
	0	0	0	0	0	0						
	1	1	1	1	1	1	[-1/4,1/8]	0	=	0	0	0
	1	1	1	1	1	0						
	1	1	1	1	0	1						
	1	1	1	1	0	0	≤ -3/8	-1	=	0	1	1
	.	.	.	.	.	.						
	1	0	1	0	0	1						

Figure 7.6 - The **qsel** Selection Function: Division

The derivation of the estimate *What[j]* from the recurrence equation for *W[j]* is straightforward and depends upon two factors. First, the value of *S[j-1]* in the recurrence

equation is guaranteed to be positive and further to be less than $2^{-P_{div}} = 2^{-3}$. Second, **qsel** only requires precision to 1/8's, or 3 fractional bit positions. Therefore, addition of Q_S to 2 • *P[j-1]* in the recurrence equation affects **qsel** only in the positions above and including 2^{-3}. Hence, the equation $What[j] = 2 \cdot P[j-1] + (\text{sign}(d_j) - 1) \cdot 2^{-3}$ is adequate to represent an estimate of the value of *W[j]* to the three fractional bit positions required by **qsel**. Elimination of the *S[j-1]* term leads to a much simpler implementation requiring less internal routing and addition circuitry than would be necessary otherwise.

The resulting logical structure of the Selection section is detailed in Figure 7.7. First, the binary point of the Selection section is placed one position to the right with respect to the binary points of **P-latch.ps** and **P-latch.sc**, effectively multiplying them by 2 to form 2 • *P[j-1]*. Next, these are combined with $(\text{sign}(d_j) - 1) \cdot 2^{-3}$ in **3-2 reduce** to produce the carry-save forms of *What[j]*. The forms are then reduced to straight binary in **cpa**, and finally **s-latch** contains the selection logic as well as the latch structures required to select and hold s_j as *s2j*, *s1j*, and *s0j*.

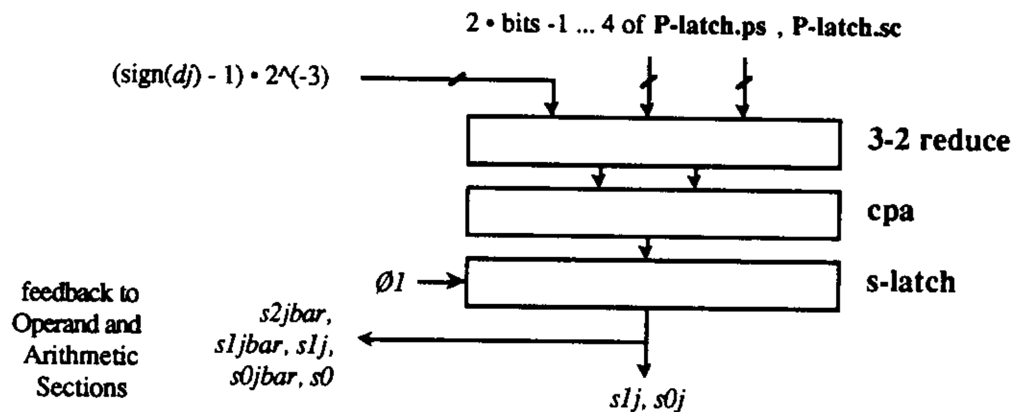


Figure 7.7 - Logical Structure of the Selection Section: Division

7.3 THE RESULTING STRUCTURE

The physical structure of the Division unit consists of the Operand, Arithmetic, and Selection sections stacked one upon the other and aligned at the left. The Operand and Arithmetic sections were integrated into type-A, type-B, and type-C bit-slices, with a single type-A slice running along the leftmost edge of the unit, 9 type-B slices inserted along the width, and finally a single type-C slice capping off the right end. The Selection section again fits snugly below the leftmost slice as the type-D portion.

As in the Square-Root Unit case, 2 special type-B slices, corresponding to bit positions 2 and 3 of the *P[j]* residual, were required in order to accept 0 as an initial condition during the loading of *h* into **P-latch.ps** at the beginning of the chip's operation. These slices differed slightly from the others in that the flip-flops of **P-latch.ps** corresponding to these positions were wired to ground. The other slices, by contrast, had their flip-flip inputs wired to the proper bit among *h1* ... *h8*.

In the final implementation, a type-A bit-slice measured 2024 microns high by 723 microns wide, all type-B slices had the same height and were either 167 (positions 2 and 3) or 179 microns wide (positions 4 ... 10), and a type-C slice again had the same height and was 222 microns wide. The type-D Selection section measured 581 microns by 1137 microns.

In the vertical dimension, the height of the unit was 2604 microns, dominated by far by the height of a bit-slice. In the horizontal dimension, the width of the unit was 2528 microns for this n=8 implementation. To accommodate a larger precision n, an additional

type-B slice, at a width of 179 microns, would be required for each additional fractional digit of precision desired.

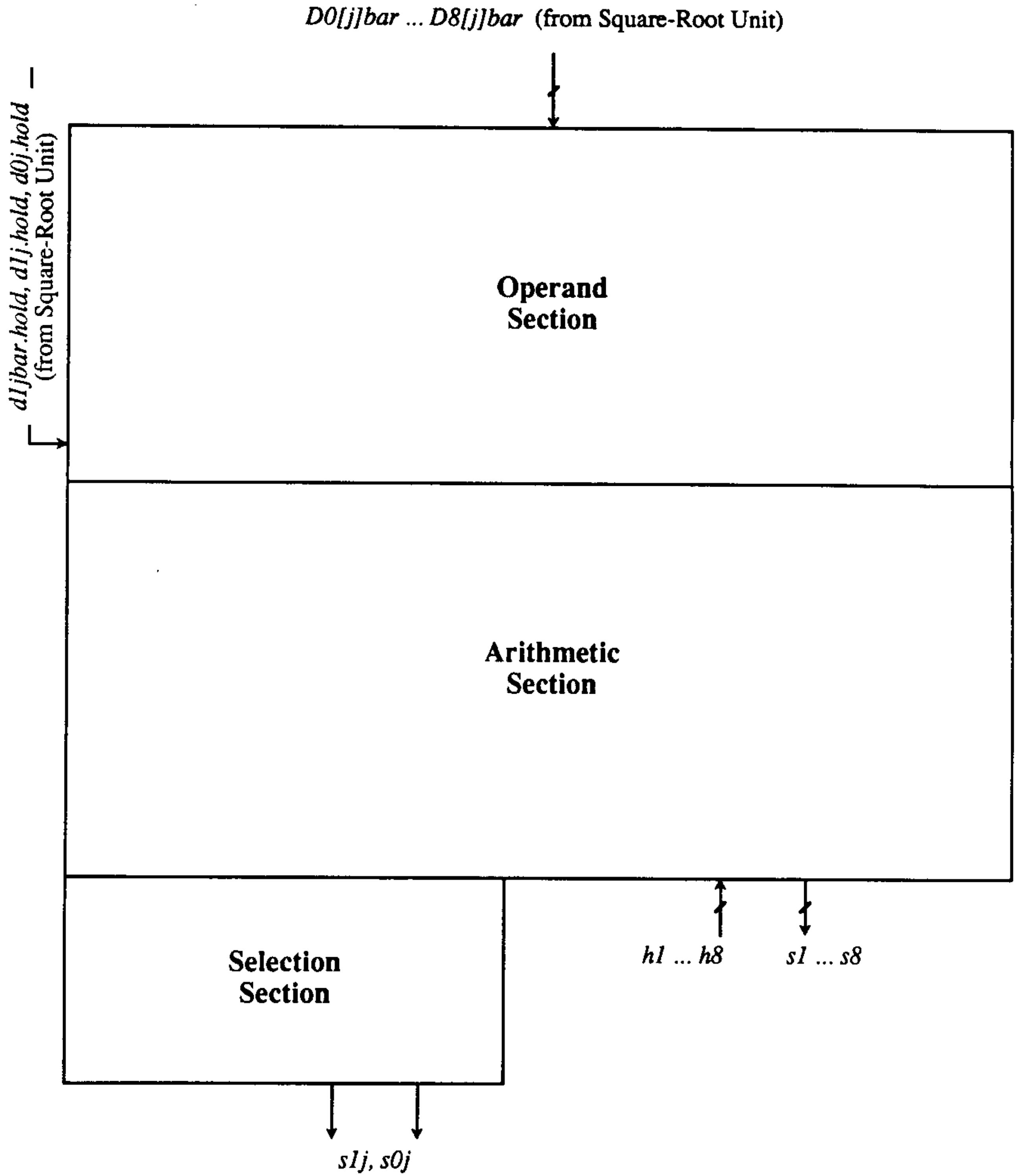


Figure 7.8 - The Division Unit (Diagram)

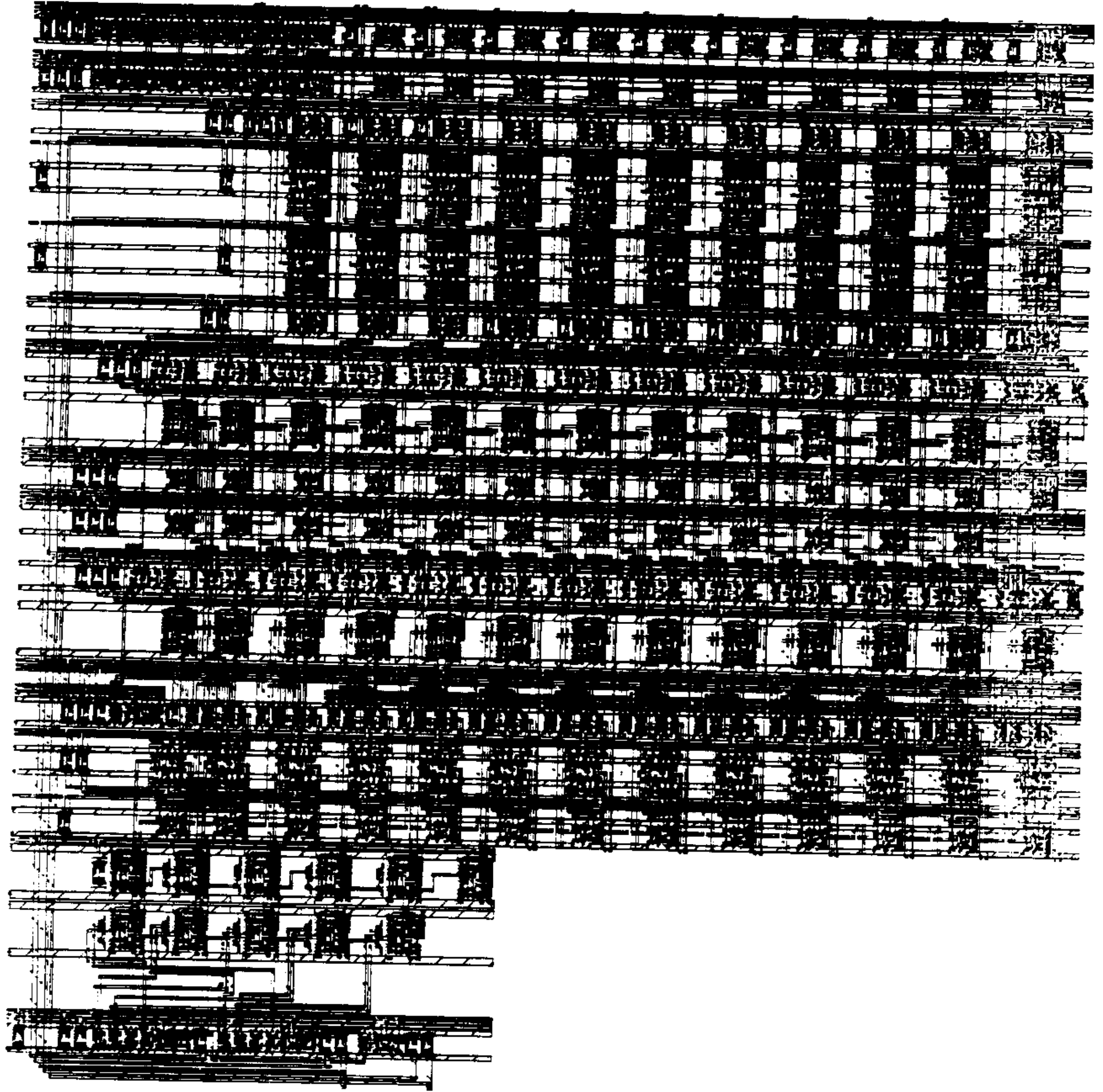


Figure 7.9 - The Division Unit (Plot)

CHAPTER 8

ARCHITECTURAL, PERFORMANCE, AND FUNCTIONAL EVALUATION

8.1 CHIP FLOORPLAN AND AREA CHARACTERISTICS

The 5 units of the final implementation were placed within the project area according to several sets of constraints. Calculation-dependent factors dictated that units be placed for smooth flow of operands between predecessor and successor units. Practically-oriented considerations included the need to attach units to a single, system-wide bus carrying power, clock, and control signals. Expansion-oriented considerations suggested that units be placed to accomodate the addition of interior bit-slices to increase the accuracy of their results in future implementations. When taken together, these considerations attempt to provide the fastest possible operation, most efficient use of project area, and greatest expansive flexibility while accommodating the unique input/output constraints imposed by each unit within the integrated architecture. Figure 8.1 details the resulting logical structure of the implementation, and Figure 8.2 provides the corresponding plot of the finished rotation chip.

First, unit-to-unit distances between those units which produce and consume a particular result digit stream were optimized to minimize the delays incurred in transmission. From Alignment through Division, communicating units are placed adjacent to each other, resulting in a clockwise flow from upper right to upper left of inter-unit digit streams. Result x_j and y_j digits flow from the Alignment Unit directly into the Sum-of-

Squares Unit, whose z_j output flows in turn to the Square-Root Unit, whose d_j and $D[j]$ results flow finally into the twin Division Units.

Alignment Unit = 1100 transistors
Sum-of-Squares Unit = 1700 transistors
Square-Root Unit = 2865 transistors
Division Units = 2 * 3482 = 6964 transistors

Entire Chip (+ pads, etc.) = 14,000+ transistors

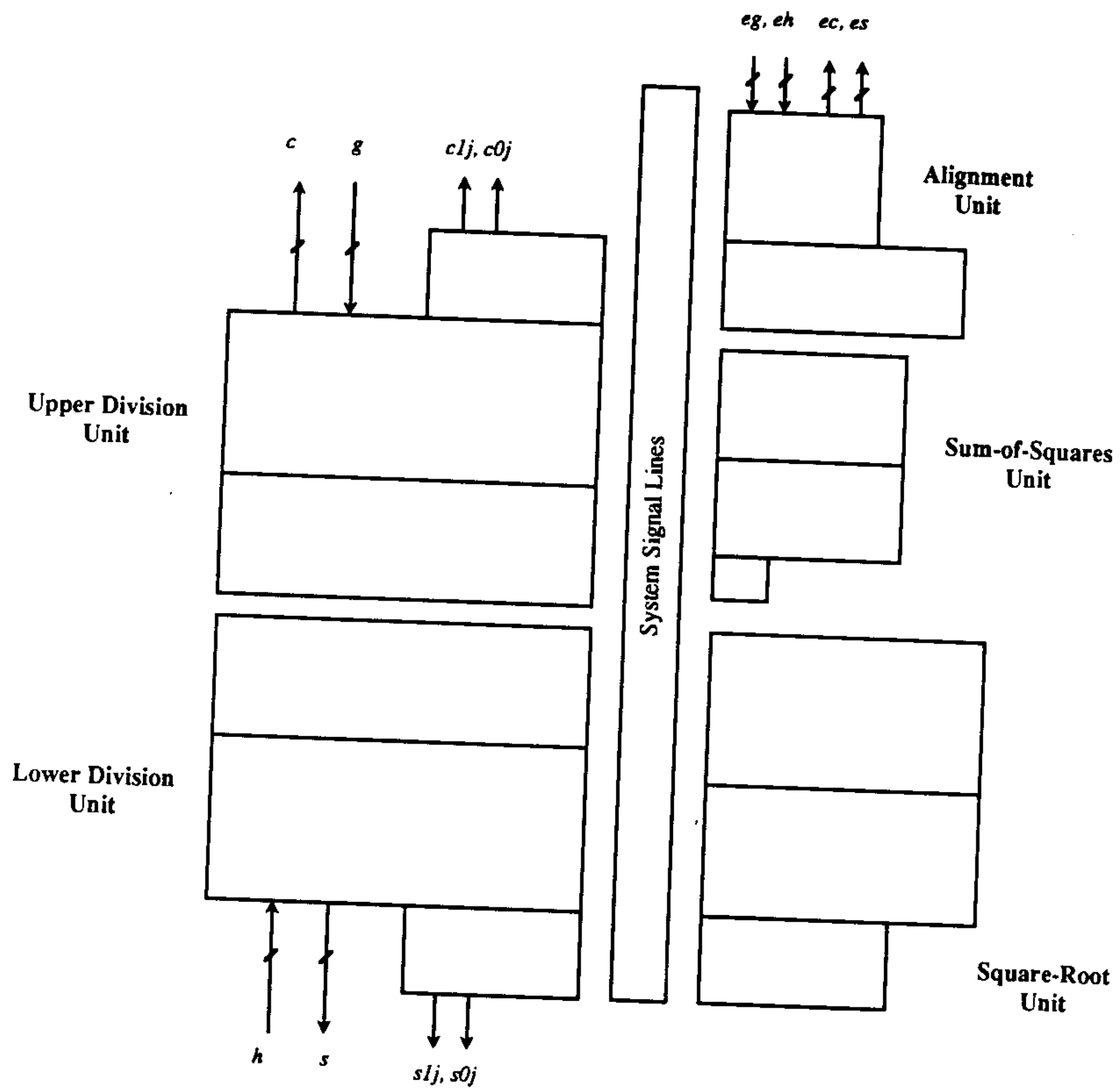


Figure 8.1 - The Online Rotation Chip (Diagram)

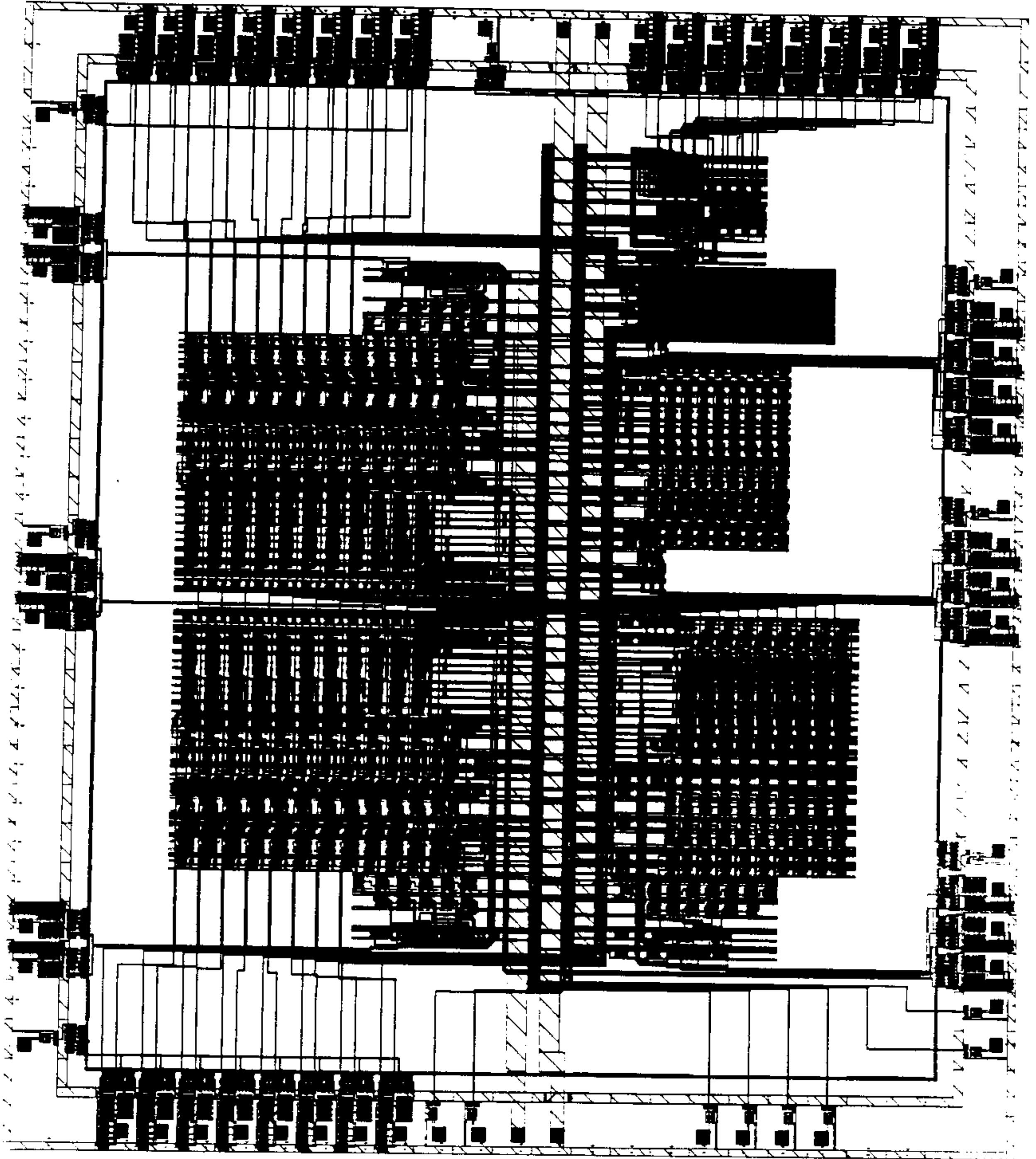


Figure 8.2 - The Online Rotation Chip (Plot)

Second, three of the units, namely the Alignment and twin Division Units, present unique circumstances because they receive and transmit inputs and results in full-parallel from the external world, and hence needed to be placed within close proximity to their appropriate bidirectional input/output pads. The Alignment Unit receives exponent inputs e_g and e_h and transmits exponent outputs e_c and e_s directly from pads located at the upper right. The mantissas g and h, however, are received from pads located at the upper and lower left, and must be bussed to the shifters of the Alignment Unit's Shifting section. Such placement was motivated by the fact that the Division Units required the original mantissa values as initial conditions for their operation, and as well produced the final mantissas c and s for output through the same pads. The final arrangement minimized the total amount of bussing required considering the needs of both units.

Third, all units required access to a common set of system power and signal lines, and by centralizing these the amount of area devoted to non-computational circuitry could be reduced. The bit-sliced design of every unit provides for the horizontal distribution of system power and signals from leads that extent to the edge of the leftmost (type-A) slice. Hence, a "signal highway" running through the center of the chip and feeding units to the left and right was used, distributing Vdd, GND, clocks, enabling, and reset signals. It was necessary to flip the orientation of the Division Units in the horizontal direction, and further to flip the top Division Unit upside down, to facilitate distribution while maintaining the ease of routing signals on and off chip.

Fourth, since the Division algorithm calls for transmission of the full value of $D[j]$ produced within the Square-Root unit at every step, a separate parallel data path had to be placed in a manner minimizing as well as equalizing the transmission delay incurred. Both

goals were accomplished by routing these lines up and out of the top of the Square-Root Unit, then horizontally to the left between the twin Division Units where individual lines were connected to the appropriate bit-slices therein.

Finally, placement was influenced by the potential for future expansion of the chip's precision via addition of extra bit-slices to each unit. As detailed in the chapters on individual units, extension of precision to greater than the $n = 8$ fractional digits provided by this implementation is easily accomplished by adding a single type-B bit-slice per unit increase in n desired to every unit. Because all units of the final implementation are oriented to expand away from the centrally-located signal highway, extension of any single unit in no way affects the placement or orientation of any other unit. Expansion, then, amounts simply to extending the horizontal dimensions of the project area, adding the appropriate bit-slices, and including extra input/output pads. See the last sections of the chapters on individual units for additional information regarding expansion.

The project area was divided into those zones encompassing the internal circuitry only (50% of the total chip area), the internal circuitry + the surrounding wire channel (71% of the total), and finally the entire chip (= internal circuitry + the surrounding wire channel + pads, 100% of the total). Of the 50% devoted to internal circuitry, 3.9% went to the Alignment Unit, 3.3% to Sum-of-Squares, 7.4% to Square-Root, and 19.3% to the twin Division Units, with the remaining 16.1% of the area dedicated to the system signal highway and routing lines.

Because the control structures required by the implementation were so simple, it was possible to optimize the use of the available chip area for computational structures that directly determine the precision that can be accomodated. Furthermore, the addition of bit

slices to the units to increase precision does not require any additional control circuitry, and therefore any additional area is used as efficiently as is possible. What little control there is consists of extensions to the shift registers that span across the tops of the Sum-of-Squares, Square-Root, and Division Units. These were extended to accommodate the online delay of each unit.

8.3 TIMING ANALYSIS: ONLINE DELAY AND CYCLE TIME

According to theory, the online delay (defined as the number of algorithmic steps between loading of the original operands and production of the first result digit) and latency (defined as the number of steps between loading and production of the final result digit) of the entire chip should be given by the equations below.

$$\begin{aligned} p_{chip} &= (p_{align} + 1) + (p_{sos} + 1) + (p_{sqr} + 1) + (p_{div} + 1) \\ &= (0 + 1) + (0 + 1) + (4 + 1) + (3 + 1) \\ &= 11 \text{ time steps, theoretically} \\ latency_{chip} &= p_{chip} + (n - 1) \\ &= 11 + (8 - 1) \\ &= 18 \text{ time steps, theoretically} \end{aligned} \tag{8.1}$$

However, two implementation-related factors led to an an actual online delay of p_{chip} = 11.5 real time steps and a $latency_{chip}$ = 19 real time steps. First, an extra time step was required at the beginning of the chip's operation for input of the original exponent and mantissa operands and was not included in the theoretical calculation. Second, given that the time steps of this implementation were divided into atomic half-cycles, it was found that a single half-cycle could be shaved off of the online delay p_{sqr}. Finally, an extra half-cycle is required at the end of the chip's operation to allow for the mantissa outputs to stabilize in their latches. Equations (8.2), representing the implementation, are given for comparison

to those of (8.1), while figure 8.3 gives a pictorial view of the online delays and overlapped operation of individual units as well as the online delay and latency of the entire implementation.

$$
\begin{aligned}
p_{chip} &= p_{input} + (p_{align} + 1) + (p_{sos} + 1) + (p_{sqr} + 1 - .5) + (p_{div} + 1) + p_{out} \\
&= .5 + (0 + 1) + (0 + 1) + (4 + 1 - .5) + (3 + 1) + .5 \\
&= 11.5 \text{ time steps, actually} \\
latency_{chip} &= p_{chip} + (n - 1) \\
&= 11.5 + (8 - 1) \\
&= 18.5 \text{ time steps, actually}
\end{aligned}
\tag{8.2}
$$

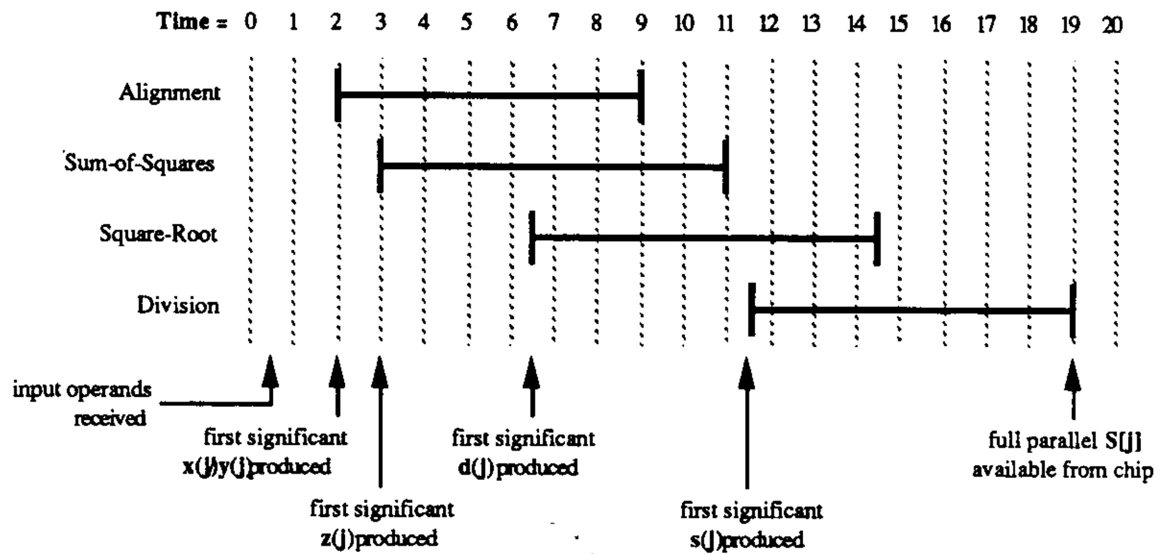


Figure 8.3 - Timing Analysis: Individual Unit and Chip-Wide Online Delays

The cycle time of the implementation was derived by locating the critical paths in both the Ø1 -> Ø2 and Ø2 -> Ø1 half cycles, then simulating them using worst-case inputs under SPICE to obtain accurate estimates of their delays. Worst-case inputs consisted of those input combinations leading to the longest propagation delays through buffers, gates,

flip-flops, and full-adders within any one unit. Note that the cycle-time of any implementation depends directly upon the technology used for the underlying circuitry; in this case, a 3 micron CMOS process was utilized. For this implementation, both critical paths are found in the Square-Root Unit.

In the Ø1 -> Ø2 half cycle, the critical path consists of three operations. First, *D[j]* and *D*[j]* are generated in the on-the-fly converter of the Operand section, with loading between the conversion registers triggered by the Ø1 pulse. Next, Q_d is formed by appending the pair of one's to the end of the converted value as detailed in chapter 6. Finally, a 4-2 reduction in the Arithmetic section forms *R[j]*. The signal path consists of a single buffer, 2 logic gates, a flip-flip, 4 more logic gates, and finally 2 full-adders, and the entire computation may take up to 50 nanoseconds.

For the Ø2 -> Ø1 half cycle, the critical path consists of two operations. First, a 3-2 reduction of the contents of **R-latch.ps**, **R-latch.sc**, and Q_z, is performed. This is then followed by a 2-1 reduction of the results prior to selection, which determines the result d_j digit. The signal path here consists of a buffer, a flip-flop, 2 more buffers, a 7-bit carry-propagate adder (composed of full-adders), and finally 3 logic gates. A worst-case delay of 55 nanoseconds was computed for this half-cycle.

Given the delays of each half-cycle in the two-phased operation of the chip, the total step time can be expressed simply as the sum of 50 and 55, or 105 nanoseconds, and correspondingly the expected speed of operation is about 9.5 megahertz. This implies that the total latency of this implementation is about (19 time steps) • (105 nanoseconds/step), or about 2 microseconds per calculation. Therefore, from a system point of view the chip

could be expected to calculate at the rate of $1/(2 \cdot 10^{-6})$ or about 0.5 MRPS (million rotations per second).

The best way to achieve faster execution speeds within this technology is to reduce the delay of each of the two critical paths. This can be done by replacing the carry-propagate adders present in both with higher-speed adder structures. The benefits of a faster cycle time may be multiplied by the number of steps involved in each calculation to determine the total time savings that may be realized. Since the carry-propagate portion of each critical path is fairly significant, substantial savings are possible.

8.3 OPERATIONAL TESTING MECHANISMS

In recognition of the difficulties involved in testing such a large chip, a number of built-in testing mechanisms were included to assist in the identification and isolation of non-working units. Although the functionality of the entire chip is lost if a non-working unit is present, it is still possible to test the other units individually for functionality. Because this is a "testbed" implementation, it was important that maximum flexibility to test individual units or groups of units be available.

The testing scheme incorporated into the design of the chip accomplishes these objectives by routing all unit-to-unit serial communications lines to a pad, where they can be sampled at desired points during the chip's operation to determine if any or all units are working properly. In fact, an override mechanism is also included, so that in addition the inputs to any one unit can be logically disconnected from the outputs of the previous unit and connected to off-chip input sources. Should any one unit be found defective, it can be bypassed and the integrity of the other units can still be established.

To assist in testing, extremely detailed records of the functional testing of the design were kept and compiled in a separate document[8]. Extensive switch-level simulation was done, with the goal of verifying the accuracy of every intermediate value ever produced within any unit. Test cases were chosen such that any and all boundary conditions that may be encountered by any single unit and the chip as a whole were checked to confirm that correct results were produced. For operational testing purposes, these results will be extremely valuable because they provide the exact 0 and 1 values expected at any point in time both within and at the output pads of the chip.

REFERENCES

1. Ercegovac, M.D., On-line arithmetic: An overview, *Proc. SPIE 1984*, Vol. 495, *Real Time Signal Processing VII*, 1984, pp. 86-93.

2. Ercegovac, M.D., and Lang, T., On-Line Arithmetic: A Design Methodology and Applications in Digital Signal Processing, *VLSI Signal Processing* III, Brodersen, R.W., and Moscovitz, H.S., editors, IEEE Press, 1988, pp. 252-263.

3. Irwin, M.J., and Owens, R.M., Digit-pipelined arithmetic as illustrated by the paste-up system: A tutorial, *IEEE Computer* (April 1987), pp. 61-73.

4. Ercegovac, M.D., and Lang, T., On-the-fly conversion of redundant into conventional representations, *IEEE Trans. Comput.* (July 1987), pp. 895-897.

5. Ercegovac, M.D. and Lang, T., On-Line Scheme for Computing Rotation Factors, *Journal of Parallel and Distributed Computing* 5, 1984, pp. 209-227.

6. Faris, S.G., *Detailed Design Specification - A VLSI Design of an Online Algorithm for the Computation of Rotation Factors*, available from the UCLA Computer Science Department.

7. Golub, G.H., and Van Loan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1983.

8. Faris, S.G., *Esim and Spice Simulation Results - A VLSI Design of an Online Algorithm for the Computation of Rotation Factors*, available from the UCLA Computer Science Department.