1993

# BiCMOS implementation on DSP arithmetic blocks.

Henry Hin Hai. Chan
*University of Windsor*

# BiCMOS IMPLEMENTATION ON DSP ARITHMETIC BLOCKS

by

Henry Hin Hai Chan

A Thesis
Submitted to the Faculty of Graduate Studies through the
Department of Electrical Engineering in Partial Fulfillment
of the Requirements for the Degree of
Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

April, 1993

ISBN 0-315-83059-X

Name Henry Hin Hai Chan

*Dissertation Abstracts International* is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

~~VLSI Bit-Mod Arithmetic Block~~
Electronics and Electrical Engineering

SUBJECT TERM

0544

SUBJECT CODE

U·M·I

Subject Categories

## THE HUMANITIES AND SOCIAL SCIENCES

### COMMUNICATIONS AND THE ARTS

| | |
|---|---|
| Architecture | 0729 |
| Art History | 0377 |
| Cinema | 0900 |
| Dance | 0378 |
| Fine Arts | 0357 |
| Information Science | 0723 |
| Journalism | 0391 |
| Library Science | 0399 |
| Mass Communications | 0708 |
| Music | 0413 |
| Speech Communication | 0459 |
| Theater | 0465 |

### EDUCATION

| | |
|---|---|
| General | 0515 |
| Administration | 0514 |
| Adult and Continuing | 0516 |
| Agricultural | 0517 |
| Art | 0273 |
| Bilingual and Multicultural | 0282 |
| Business | 0688 |
| Community College | 0275 |
| Curriculum and Instruction | 0727 |
| Early Childhood | 0518 |
| Elementary | 0524 |
| Finance | 0277 |
| Guidance and Counseling | 0519 |
| Health | 0680 |
| Higher | 0745 |
| History of | 0520 |
| Home Economics | 0278 |
| Industrial | 0521 |
| Language and Literature | 0279 |
| Mathematics | 0280 |
| Music | 0522 |
| Philosophy of | 0998 |
| Physical | 0523 |
| Psychology | 0525 |
| Reading | 0535 |
| Religious | 0527 |
| Sciences | 0714 |
| Secondary | 0533 |
| Social Sciences | 0534 |
| Sociology of | 0340 |
| Special | 0529 |
| Teacher Training | 0530 |
| Technology | 0710 |
| Tests and Measurements | 0288 |
| Vocational | 0747 |

### LANGUAGE, LITERATURE AND LINGUISTICS

| | |
|---|---|
| Language | |
| General | 0679 |
| Ancient | 0289 |
| Linguistics | 0290 |
| Modern | 0291 |
| Literature | |
| General | 0401 |
| Classical | 0294 |
| Comparative | 0295 |
| Medieval | 0297 |
| Modern | 0298 |
| African | 0316 |
| American | 0591 |
| Asian | 0305 |
| Canadian (English) | 0352 |
| Canadian (French) | 0355 |
| English | 0593 |
| Germanic | 0311 |
| Latin American | 0312 |
| Middle Eastern | 0315 |
| Romance | 0313 |
| Slavic and East European | 0314 |

### PHILOSOPHY, RELIGION AND THEOLOGY

| | |
|---|---|
| Philosophy | 0422 |
| Religion | |
| General | 0318 |
| Biblical Studies | 0321 |
| Clergy | 0319 |
| History of | 0320 |
| Philosophy of | 0322 |
| Theology | 0469 |

### SOCIAL SCIENCES

| | |
|---|---|
| American Studies | 0323 |
| Anthropology | |
| Archaeology | 0324 |
| Cultural | 0326 |
| Physical | 0327 |
| Business Administration | |
| General | 0310 |
| Accounting | 0272 |
| Banking | 0770 |
| Management | 0454 |
| Marketing | 0338 |
| Canadian Studies | 0385 |
| Economics | |
| General | 0501 |
| Agricultural | 0503 |
| Commerce-Business | 0505 |
| Finance | 0508 |
| History | 0509 |
| Labor | 0510 |
| Theory | 0511 |
| Folklore | 0358 |
| Geography | 0366 |
| Gerontology | 0351 |
| History | |
| General | 0578 |
| Ancient | 0579 |
| Medieval | 0581 |
| Modern | 0582 |
| Black | 0328 |
| African | 0331 |
| Asia, Australia and Oceania | 0332 |
| Canadian | 0334 |
| European | 0335 |
| Latin American | 0336 |
| Middle Eastern | 0333 |
| United States | 0337 |
| History of Science | 0585 |
| Law | 0398 |
| Political Science | |
| General | 0615 |
| International Law and Relations | 0616 |
| Public Administration | 0617 |
| Recreation | 0814 |
| Social Work | 0452 |
| Sociology | |
| General | 0626 |
| Criminology and Penology | 0627 |
| Demography | 0938 |
| Ethnic and Racial Studies | 0631 |
| Individual and Family Studies | 0628 |
| Industrial and Labor Relations | 0629 |
| Public and Social Welfare | 0630 |
| Social Structure and Development | 0700 |
| Theory and Methods | 0344 |
| Transportation | 0709 |
| Urban and Regional Planning | 0999 |
| Women's Studies | 0453 |

## THE SCIENCES AND ENGINEERING

### BIOLOGICAL SCIENCES

| | |
|---|---|
| Agriculture | |
| General | 0473 |
| Agronomy | 0285 |
| Animal Culture and Nutrition | 0475 |
| Animal Pathology | 0476 |
| Food Science and Technology | 0359 |
| Forestry and Wildlife | 0478 |
| Plant Culture | 0479 |
| Plant Pathology | 0480 |
| Plant Physiology | 0817 |
| Range Management | 0777 |
| Wood Technology | 0746 |
| Biology | |
| General | 0306 |
| Anatomy | 0287 |
| Biostatistics | 0308 |
| Botany | 0309 |
| Cell | 0379 |
| Ecology | 0329 |
| Entomology | 0353 |
| Genetics | 0369 |
| Limnology | 0793 |
| Microbiology | 0410 |
| Molecular | 0307 |
| Neuroscience | 0317 |
| Oceanography | 0416 |
| Physiology | 0433 |
| Radiation | 0821 |
| Veterinary Science | 0778 |
| Zoology | 0472 |
| Biophysics | |
| General | 0786 |
| Medical | 0760 |

### EARTH SCIENCES

| | |
|---|---|
| Biogeochemistry | 0425 |
| Geochemistry | 0996 |
| Geodesy | 0370 |
| Geology | 0372 |
| Geophysics | 0373 |
| Hydrology | 0388 |
| Mineralogy | 0411 |
| Paleobotany | 0345 |
| Paleoecology | 0426 |
| Paleontology | 0418 |
| Paleozoology | 0985 |
| Palynology | 0427 |
| Physical Geography | 0368 |
| Physical Oceanography | 0415 |

### HEALTH AND ENVIRONMENTAL SCIENCES

| | |
|---|---|
| Environmental Sciences | 0768 |
| Health Sciences | |
| General | 0566 |
| Audiology | 0300 |
| Chemotherapy | 0992 |
| Dentistry | 0567 |
| Education | 0350 |
| Hospital Management | 0769 |
| Human Development | 0758 |
| Immunology | 0982 |
| Medicine and Surgery | 0564 |
| Mental Health | 0347 |
| Nursing | 0569 |
| Nutrition | 0570 |
| Obstetrics and Gynecology | 0380 |
| Occupational Health and Therapy | 0354 |
| Ophthalmology | 0381 |
| Pathology | 0571 |
| Pharmacology | 0419 |
| Pharmacy | 0572 |
| Physical Therapy | 0382 |
| Public Health | 0573 |
| Radiology | 0574 |
| Recreation | 0575 |
| Speech Pathology | 0460 |
| Toxicology | 0383 |
| Home Economics | 0386 |

### PHYSICAL SCIENCES

**Pure Sciences**

| | |
|---|---|
| Chemistry | |
| General | 0485 |
| Agricultural | 0749 |
| Analytical | 0486 |
| Biochemistry | 0487 |
| Inorganic | 0488 |
| Nuclear | 0738 |
| Organic | 0490 |
| Pharmaceutical | 0491 |
| Physical | 0494 |
| Polymer | 0495 |
| Radiation | 0754 |
| Mathematics | 0405 |
| Physics | |
| General | 0605 |
| Acoustics | 0986 |
| Astronomy and Astrophysics | 0606 |
| Atmospheric Science | 0608 |
| Atomic | 0748 |
| Electronics and Electricity | 0607 |
| Elementary Particles and High Energy | 0798 |
| Fluid and Plasma | 0759 |
| Molecular | 0609 |
| Nuclear | 0610 |
| Optics | 0752 |
| Radiation | 0756 |
| Solid State | 0611 |
| Statistics | 0463 |

**Applied Sciences**

| | |
|---|---|
| Applied Mechanics | 0346 |
| Computer Science | 0984 |
| Engineering | |
| General | 0537 |
| Aerospace | 0538 |
| Agricultural | 0539 |
| Automotive | 0540 |
| Biomedical | 0541 |
| Chemical | 0542 |
| Civil | 0543 |
| Electronics and Electrical | 0544 |
| Heat and Thermodynamics | 0348 |
| Hydraulic | 0545 |
| Industrial | 0546 |
| Marine | 0547 |
| Materials Science | 0794 |
| Mechanical | 0548 |
| Metallurgy | 0743 |
| Mining | 0551 |
| Nuclear | 0552 |
| Packaging | 0549 |
| Petroleum | 0765 |
| Sanitary and Municipal | 0554 |
| System Science | 0790 |
| Geotechnology | 0428 |
| Operations Research | 0796 |
| Plastics Technology | 0795 |
| Textile Technology | 0994 |

### PSYCHOLOGY

| | |
|---|---|
| General | 0621 |
| Behavioral | 0384 |
| Clinical | 0622 |
| Developmental | 0620 |
| Experimental | 0623 |
| Industrial | 0624 |
| Personality | 0625 |
| Physiological | 0989 |
| Psychobiology | 0349 |
| Psychometrics | 0632 |
| Social | 0451 |

## ABSTRACT

This thesis presents an improved VLSI architecture to perform different arithmetic operations, multiplication, division and square rooting, along with addition and subtraction. The architecture is highly regular, requires only three control bits to choose among five different operations. Through the use of a redundant binary number system and pipelining, the execution time for each operation is identical and is independent of the wordsize of the array. Moreover, the improved architecture is capable of being implemented using the dynamic switching tree technique. Finally, the improved architecture has been designed utilizing a 0.8 micron BiCMOS technology and has a throughput rate of 100 Megasamples per second for each operation.

## *ACKNOWLEDGEMENTS*

*I would like to express my sincere thanks and appreciation to* ***Dr. G.A. Jullien*** *and* ***Dr. W.C. Miller*** *for their tremendous support and guidance throughout the progress of this theis. I would like to give my special thanks to* ***Bruce Erickson*** *for his help and experiences. Thanks also go to* ***Andy Hung, Raymond Lei, Sam Lai*** *and* ***H.M. Chan*** *for their help and stay at school during night time. Special thanks must be given to* ***Teresa Chan*** *for her understanding, patience and support during the writing process. Last but far from least, I would like to thank my parents and my brother,* ***Mr. & Mrs. Tsang*** *and* ***Peter Tsang****, for without them none of this would be possible.*

# TABLE OF CONTENTS

## LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# INTRODUCTION

## 1.1 INTRODUCTION

In digital signal procesing (DSP), there is a well established need for high performance implementations of the basic arithmetic operations. All DSP systems require fast multiply and add/subtract operations while the more complex systems have a requirement for division and square root. It is evident that the speed of the basic arithmetic operations has considerable consequences on the performance of current DSP systems. With the rapid growth of integrated circuit technology, complex DSP systems which were prohibitively expensive in hardware terms, are usually implemented on a single chip. Furthermore, many arithmetic accelerating schemes (e.g. redundant number systems) are taking this advantage to improve the performance of current arithmetic algorithms.

Several architectures which share hardware to perform multiplication, division and square root have been proposed [1] [2] [3]. The architecture described by Kamal [1], exhibits regularity and local communication. However, its throughput rate is wordlength dependent and severely limited by carry propagate arithmetic. A similar array architecture was proposed by Agrawal [2]. The carry-save method is engaged instead of the carry propagate arithmetic. However, the result digits are in conventional binary form and result in lengthy communication paths for long wordlengths. The bit-serial architectures described by Zurawski and Gosling [3] and Ercegovac and Lang [4] do not address any regularity and local communications which are suited for VLSI implementation. Additionally, relatively complex control is required and the throughputs are dependent on the wordlength.

The VLSI architecture proposed by McCanny and McQuillan [5] is capable of performing combined multiply-accumulate, division and square root operations at a very high throughput rate by employing pipelining and redundant arithmetic system. Moreover, the architecture is highly regular, requires minimal control and can be reconfigured on every cycle. The execution time for each operation is the same and the throughput rate is independent of the wordsize of the array. The focus of this thesis is to implement an improved architecture which has a better performance, and utilizes less number of standard cells than the previous architecture. This improved architecture employs many of the characteristics from the original architecture, including regularity, equal execution time for each operation, and precision independent throughput and simple control. Furthermore, add and subtract operations are combined with the existing operations on the improved architecture. The final VLSI implementation utilizes 0.8μm BiCMOS technology, one of the most advance technologies available nowadays.

## 1.2 ORGANIZATION OF THESIS

This thesis consists of five chapters. The first chapter serves to briefly present the contents and then to lay out the structure of the remaining chapters. Chapter 2, is divided into two sections. The first section is devoted to a review of Signed Digit Number Representation (SDNR) [6], in particular, the Signed Binary Number Representation (SBNR) [7]. Different radix-2 redundant adders are also discussed. In the second part of the chapter, multiply, division and square root algorithms which employ the redundant arithmetic system are briefly discussed through the use of examples.

In Chapter 3, a unified algorithm to perform multiply, divide and square root operations and VLSI architecture to implement the algorithm purposed by McCanny and McQuillan [5] are discussed in detail. Based on the original architecture, an improved architecture is proposed in this chapter. Comparisons between two architectures are made in terms of the number of standard cells required and performance.

Chapter 4 briefly describes the Hardware Description Language (HDL), Verilog, which used to describe the function of the architectures and perform the switch-level simulation. VLSI implementation of the architectures are presented in this chapter.

The final chapter, Chapter 5, concludes the work with a summary and few suggestions are also made to give direction to future research in this area.

# Chapter 2

# REDUNDANT NUMBER SYSTEMS AND ALGORITHMS

## 2.1 INTRODUCTION

Over the years, the demand for higher performance and higher functionality VLSI processors has increased substantially. Therefore, there is also need to improve the performance, area, efficiency, and functionality of the arithmetic units contained within these VLSI processors. The basic arithmetic function, addition, is often implemented by using the ripple carry adder, which uses a minimum number of gates, but forces a long delay in producing the sum since the carry must be propagated through the entire number. Several methods are described in the literature [8] [9] that overcome the problem generated by carry propagation. One of the techniques is to introduce redundancy into the number systems to accelerate arithmetic operations. Redundancy can also provide structural flexibility to these number systems. This chapter will be divided into two main sections. The first section deals with the general concept of Signed Digit Number Representation (SDNR) [6], which is the basic fundation of the SBNR adder. The first section will also examine the general characteristic properties of signed-binary number. Methods for redundant addition will be discussed through the use of examples. The second section of this chapter will deal with the algorithms that are used as the basis of the work. This section will discuss the division, square root and multiply algorithms.

## 2.2 SIGNED DIGIT NUMBER REPRESENTATION[6]

The Signed Digit Number Representation (SDNR) was originally proposed by Avizienis to eliminate carry propagation chains in operations such as addition, subtraction, multiplication and division. Signed-digit numbers differ from conventional numbers in that the individual digit comprising a number allows both positive and negative digit values and since the individual digit contains all the sign information, there is no need for an explicit mechanism (such as 2's complement) to handle the overall sign of a number. For example, in a radix-2 SDNR, the individual digit may assume values 1, 0 or -1 ( denoted by $\bar{1}$ ). Signed-digit representations are redundant, that is, each radix r digit $z_i$ assumes more than r different values. In conventional (non-redundant) number representation, only r values of a digit (0, 1, ..., r-1) are allowed. In signed-digit (redundant) number representation, the value a (where a is the maximum digit magnitude) is chosen from the following range:

$$-(r-1) \le a \le r-1 \qquad \text{for radix r} \qquad (2.1)$$

The smallest set of digits is termed the minimally redundant set and contains at least r+1 values ( where r is the radix ). The largest set of digits is termed the maximally redundant set and contains at most 2r-1 values. For example, in higher radices, there is some choice available in the digit set that can be chosen: symmetric digit sets for radix-4 can be chosen as either $\{\bar{2},...2\}$ or $\{\bar{3},....3\}$. The other reason that signed-digit numbers are termed as redundant is that any given algebraic value may have several possible representations. For example, value 13 may be represented several ways in radix-2 SDNR as 1101, or $111\bar{1}$, or $100\bar{1}\bar{1}$, etc. The characteristic properties of signed-digit representations are listed below.

1. The algebraic value Z of the number z composed of n+m+1 digits ($z_n \cdots z_1 z_0 z_{-1} \cdots z_{-m}$) is given by the conventional expression:

$$Z = \sum_{i=n}^{-m} z_i r^i \qquad \text{where radix r is a positive integer} \qquad (2.2)$$

2. The algebraic value Z=0 has a unique representation, if and only if all $z_i$ =0.
3. The sign of the algebraic value Z is given by the sign of the most significant (left most) nonzero digit.
4. To form the representation of the additive inverse -Z, the sign of every nonzero digit $z_i$ is changed individually.
5. The addition and subtraction of two signed-digit operands Z and Y satisfies $s_i = f(z_i, y_i, z_{i-1}, y_{i-1}, z_{i-2}, y_{i-2})$ $\forall i$, where $s_i$ are digits in the representation of the sum or difference $s_i = z_i \pm y_i$. In other words, $s_i$ is a function of the sum or difference of three adjacent operands.

In a conventional binary number system, the addition of two binary numbers requires a computation time at least proportional to the logarithm of the wordlength of the operands, because of carry propagation from Least-Significant Bit (LSB) through Most-Significant Bit (MSB). This is usually the bottleneck for speed improvement in digital integrated systems. The addition of two binary numbers is sketched in Figure 2.1.

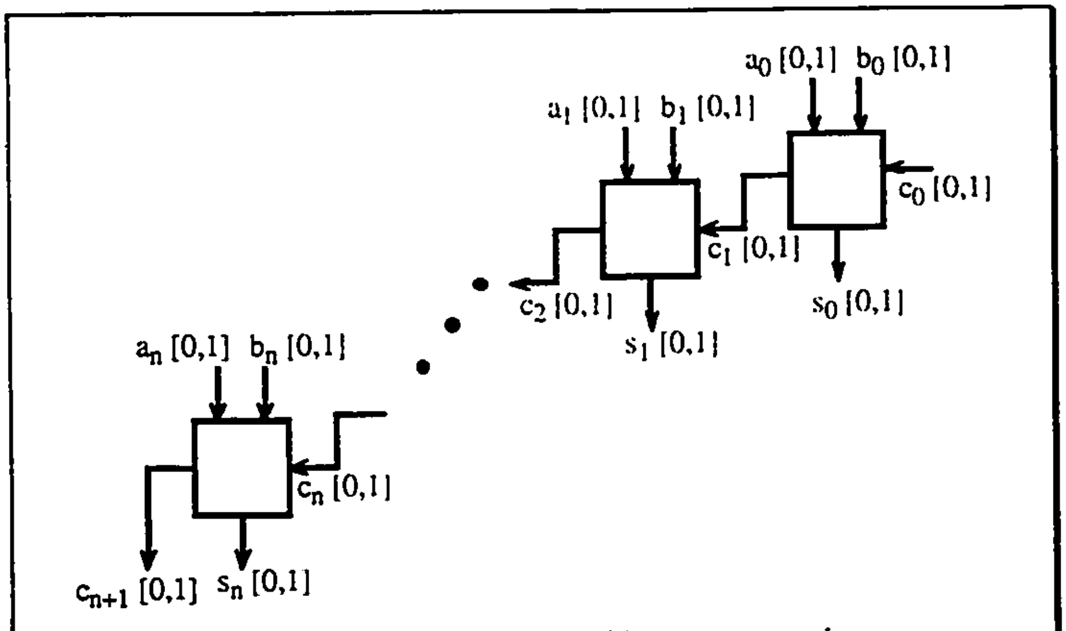


Figure 2.1: Classical addition with carry propagation

As the global delay of the adder depends on the carry delay, the basic full adder cell is optimized regarding this criterion. Redundancy in a number system allows methods of addition to be devised in which each digit of the result is a function only of the digits in a few adjacent positions of the operands and does not depend on the other digits in any way. Thus the features of redundancy in a number system has several important consequences. It allows parallel arithmetic operations to be performed completely without full carry propagation from the Least-Significant Bit (LSB) through to the Most-Significant Bit (MSB). Thus the time required for an operation such as parallel addition or subtraction is constant and does not dependent on the wordlength. In other words, it provides flexibility to the structure of the arithmetic units. A corollary of this is that it is possible to pipeline such an adder so that it operates from the most significant digit first.

## 2.2.1 SIGNED BINARY NUMBER REPRESENTATION

Of particular interest in this thesis is the Radix-2 SDNR known as the Signed Binary Number Representation (SBNR) which has a digit set $\{\bar{1},0,1\}$. Each SBNR digit requires a two bit representation implying a multiplicity of possible encoding schemes. Two encoding schemes which are particularly useful are the sign-and-magnitude and the (+,-) schemes which are defined in Table 2.1.

| Digit $d^*$ | sign-and-magnitude $d_s$ $d_m$ | (+,-) $d^+$ $d^-$ |
|---|---|---|
| 0 | 0 0 | 0 1 |
| 1 | 0 1 | 1 1 |
| -1 | 1 1 | 0 0 |
| 0 or d (d doesn't exist in (+,-)) | 1 0 | 1 0 |

Table 2.1: Encoding schemes of SBNR digits
Note: d: don't care

In the sign-and-magnitude scheme, the SBNR digits are coded as the sign and magnitude bits, that is $d^*=(d_s,d_m)$. In the (+,-) scheme, an SBNR digit is coded as $d^*=(d^+,d^-)$ where $d^*=d^+ + (d^- -1)$. The $d^+$ bit is coded such that $d^+ =0$ implies 0 and $d^+ =1$ implies 1, while the $d^-$ bit is coded such that $d^- =0$ implies -1 and $d^- =1$ implies 0. An important property of the (+,-) scheme is that the SBNR adders can be constructed from simple binary full adders. The structure of the (+,-) scheme adders will be discussed later in this section.

## 2.2.2 REDUNDANT BINARY ADDER

A redundant binary or SBNR adder in which all digit sets are $\{\bar{1},0,1\}$ was first discussed by Avizienis[6]. He showed that the parallel addition or subtraction dictates a three level structure. The result digit $s_i$ depends only on the three adjacent operands $z_{i-2},y_{i-2},z_{i-1},y_{i-1},z_i$ and $y_i$. The operation of the redundant binary adder is described by the following equations(2.3a - 2.3c):

$$z_i + y_i = 2t_i + w_i \qquad (2.3a)$$
$$w_i + t_{i-1} = 2t'_i + w'_i \qquad (2.3b)$$
$$s_i = w'_i + t'_{i-1} \qquad (2.3c)$$

where $z_i$ and $y_i$ are the operands, $w_i$ and $w'_i$ are defined as intermediate sum digits, $t_i$ and $t'_i$ are the transfer digits, and $s_i$ is the sum digit. The term transfer digit is used here instead of the commonly used terms "carry" or "borrow" for two reasons. First, the transfer digit may assume both positive and negative values in either addition or subtraction. Second, unlike the "carry" or "borrow" of conventional addition or subtraction, the transfer digit is never propagated past the first adder position on the left. The structure of the adder is shown in Figure 2.2. The sum of operands is realized in three steps. In the first step, $t_i$ is 1 whenever $z_i + y_i > 1$. In the second step, transfer digit $t'_i$ is 1 only if $t_i + w_i = 2$. This ensures that $t_i$ and $t'_i$ cannot be equal to +1 or -1 at the same time. At the last step, the sum digit, $s_i$, is simply obtained by the free addition of transfer digit $t'_{i-1}$ from the adjacent digit and sum digit $w'_i$.

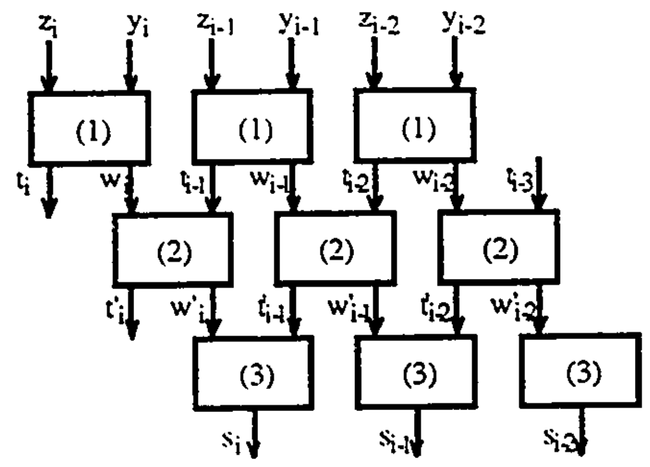


Figure 2.2: Redundant binary adder

## 2.2.2 MODIFIED REDUNDANT BINARY ADDER

Based on the concept introduced by Avizienis, the logic design and implementation of redundant binary adders have been investigated by several other authors[10] [11] [5]. Here, only the logic design by Robertson [10] will be discussed. The logic design will be used as the basic foundation in the improved arithmetic architecture in the following chapter.

The logic design introduced by Robertson [10] is very similar to the design by Avizienis [6], but with the combination of the first two stages, it results in a relatively simple logic design compared to the previous one. The redundant binary adder constructed with two inputs and one output, in the digit set $\{\bar{1},0,1\}$. The structure of the adder is shown in Figure 2.3, and the operation of the redundant binary adder is shown in eqn. 2.4:

$$l_i^* + k_i^* = 2m_{i+1} + a_i^* \tag{2.4a}$$
$$a_i^* + m_{i-1} = 2h_{i+1} + d_i \tag{2.4b}$$
$$s_i^* = d_i + h_{i-1} \tag{2.4c}$$

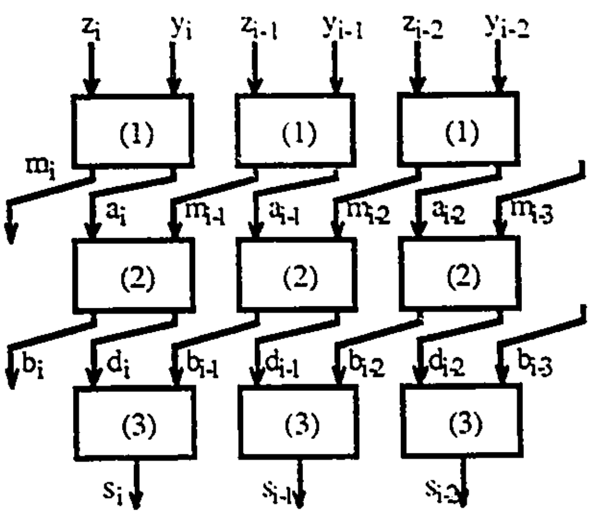


Figure 2.3: Modified redundant binary adder

At ith position, inputs $l_i^*$, $k_i^*$ are the operands of the adder/subtractor structure, and with output operates at one non-redundant $m_{i+1}$ and one redundant digit $a_i^*$ (an asterisk beside a symbol denotes a redundant binary digit chosen from digit set $\{\bar{1},0,1\}$). The sum of $l_i^*$ and $k_i^*$ equals $(2m_{i+1}+a_i^*)$ (2.4a) must be chosen from one of the following combinations (2.5a - 2.5c):

$$m_{i+1} \in \{\bar{1},0\},\ a_i^* \in \{0,1,2\} \tag{2.5a}$$

$$m_{i+1} \in \{0,1\},\ a_i^* \in \{\bar{2},\bar{1},0\} \tag{2.5b}$$

$$2m_{i+1} \in \{\bar{1},1\},\ a_i^* \in \{\bar{1},0,1\} \tag{2.5c}$$

Combining the first two levels of the original structure does result in a relatively simple logic design. With the combination of the first two stages, the format of transfer digit $a_i^*$ no longer needs to be considered. For the symmetric adder, the algebraic relationships are listed below and Figure 2.4 [10] depicts the operations shown below:

$$d_i + b_i = s_i^* \quad \text{for the final block} \tag{2.6}$$

$$l_i^* + k_i^* + m_{i-1} = 2m_i + 2b_i + d_i \quad \text{for the combined block} \tag{2.7}$$

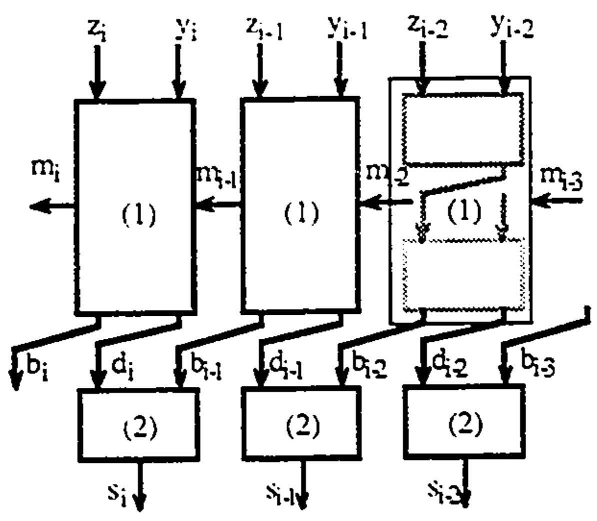


Figure 2.4: Modified 2-level redundant binary adder

In spite of equations 2.6 and 2.7, $m_i$ can be made independent of $m_{i-1}$. The final result digit $s_i^*$ is still a function only of the digits in three adjacent digital positions of the operands. The chosen sets for $m_i$, $b_i$, $d_i$ are $\{0,1\}$, $\{\bar{1},0\}$, $\{0,1\}$ respectively. Before proceeding further, one has to fix the binary representation for the redundant binary digit set $\{\bar{1},0,1\}$. The term redundant is used here because there exists more than one way to represent a redundant binary number by using the two-bit binary number. Robertson has shown that there exists only nine distinct ways, under permutation and negation, of representing three values $\bar{1}$ ,0 and 1 from the redundant digit set with a two-bit binary number. The nine formats are shown in Table 2.2. Since it is sometimes necessary to feed the output of the redundant adder as an operand to the input of the next redundant adder, the result digit $s_i^*$ should have the same binary representation as the operand digits $l_i^*$ and $k_i^*$.

| $s_i^n$ | $s_i^p$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | d | 0 | 0 | 1 | $\bar{1}$ |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | $\bar{1}$ | d | 0 | $\bar{1}$ | 0 | 1 | $\bar{1}$ | 0 | 0 |
| 1 | 1 | 0 | $\bar{1}$ | $\bar{1}$ | d | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ |

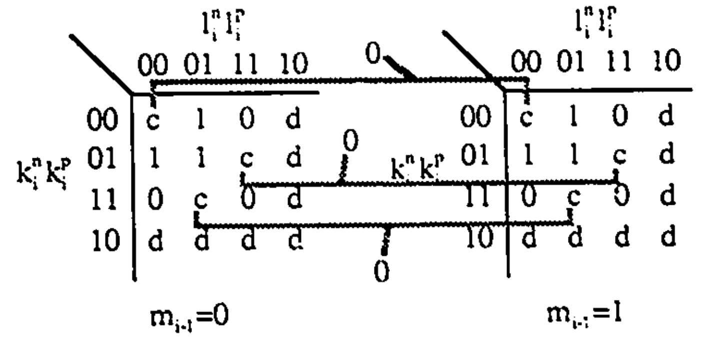
Table 2.2: Nine distinct formats of representing a redundant binary digit with two bits
Note : $s_i^* = [s_i^n, s_i^p]$, d : don't care and $\bar{1}$ : -1

Some rather interesting details are encountered in the design of this redundant binary adder. From equation 2.7, whenever $(l_i^* + k_i^* + m_{i-1})$ is algebraically 0 or 1, $-b_i$ and $m_i$ can be both 0 or both 1. This results in the introduction of "Coupled don't care" cases in the truth table. Table 2.3 shows the truth table for the combined block of the redundant binary adder using format 2 from Table 2.2. $l_i^*$, $k_i^*$ and $s_i^*$ are represented by pairs of bits $(l_i^n, l_i^p)$, $(k_i^n, k_i^p)$ and $(s_i^n, s_i^p)$ respectively. The horizontal pairs of c's across $b_i$ and $m_i$ are the "coupled don't cares". A horizontal pair of c's can be both 0 or both 1.

The complexity introduced by the "coupled don't cares" is greatly reduced by the important constraints that the transfers $b_i$ and $m_i$ be non-propagating. This means that $m_i$ must be independent of $m_{i-1}$ from the adjacent unit. Therefore, in Table 2.3, the "coupled don't cares" for minterms 1, 4, 19 and 28 must have the values 1, 1, 0, and 0 respectively. This requires the upper 16 function values of $m_i$ to match the lower 16 function values of $m_i$. With four minterms of "coupled don't cares" being fixed, there remain only six "coupled don't cares" of the values of $m_i$ needed to fill the table. Figure 2.5 shows the Karnaugh maps for $m_i$ from Table 2.3. Similar karnaugh maps for $b_i$ and $d_i$ are shown in Figure 2.6 and Figure 2.7 respectively.

| | $m_{i-1}$ | $l_i^n$ | $l_i^p$ | $k_i^n$ | $k_i^p$ | value | $m_i$ | $b_i$ | $d_i$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | c | c | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | c (1) | c (1) | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 | d | d | d | d |
| 3 | 0 | 0 | 0 | 1 | 1 | $\bar{1}$ | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 | 1 | c (1) | c (1) | 1 |
| 5 | 0 | 0 | 1 | 0 | 1 | 2 | 1 | 0 | 0 |
| 6 | 0 | 0 | 1 | 1 | 0 | d | d | d | d |
| 7 | 0 | 0 | 1 | 1 | 1 | 0 | c | c | 0 |
| 8 | 0 | 1 | 0 | 0 | 0 | d | d | d | d |
| 9 | 0 | 1 | 0 | 0 | 1 | d | d | d | d |
| 10 | 0 | 1 | 0 | 1 | 0 | d | d | d | d |
| 11 | 0 | 1 | 0 | 1 | 1 | d | d | d | d |
| 12 | 0 | 1 | 1 | 0 | 0 | $\bar{1}$ | 0 | 1 | 1 |
| 13 | 0 | 1 | 1 | 0 | 1 | 0 | c | c | 0 |
| 14 | 0 | 1 | 1 | 1 | 0 | d | d | d | d |
| 15 | 0 | 1 | 1 | 1 | 1 | $\bar{2}$ | 0 | 1 | 0 |
| 16 | 1 | 0 | 0 | 0 | 0 | 1 | c | c | 1 |
| 17 | 1 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 0 |
| 18 | 1 | 0 | 0 | 1 | 0 | d | d | d | d |
| 19 | 1 | 0 | 0 | 1 | 1 | 0 | c (0) | c (0) | 0 |
| 20 | 1 | 0 | 1 | 0 | 0 | 2 | 1 | 0 | 0 |
| 21 | 1 | 0 | 1 | 0 | 1 | 3 | 1 | 0 | 1 |
| 22 | 1 | 0 | 1 | 1 | 0 | d | d | d | d |
| 23 | 1 | 0 | 1 | 1 | 1 | 1 | c | c | 1 |
| 24 | 1 | 1 | 0 | 0 | 0 | d | d | d | d |
| 25 | 1 | 1 | 0 | 0 | 1 | d | d | d | d |
| 26 | 1 | 1 | 0 | 1 | 0 | d | d | d | d |
| 27 | 1 | 1 | 0 | 1 | 1 | d | d | d | d |
| 28 | 1 | 1 | 1 | 0 | 0 | 0 | c (0) | c (0) | 0 |
| 29 | 1 | 1 | 1 | 0 | 1 | 1 | c | c | 1 |
| 30 | 1 | 1 | 1 | 1 | 0 | d | d | d | d |
| 31 | 1 | 1 | 1 | 1 | 1 | $\bar{1}$ | 0 | 1 | 1 |

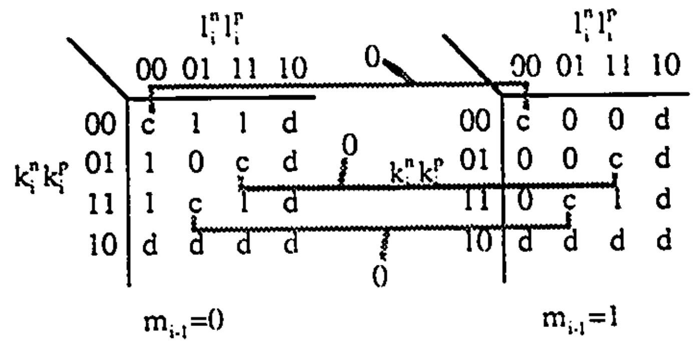
Table 2.3: Truth Table for the combined block of format 2



| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | c | 1 | 0 | d |
| 01 | 1 | 1 | c | d |
| 11 | 0 | c | 0 | d |
| 10 | d | d | d | d |

| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | c | 1 | 0 | d |
| 01 | 1 | 1 | c | d |
| 11 | 0 | c | 0 | d |
| 10 | d | d | d | d |

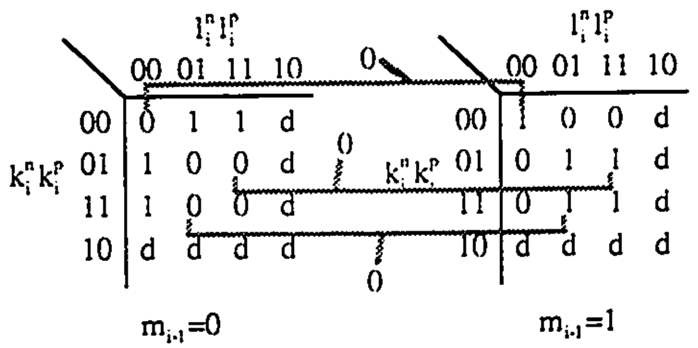
Figure 2.5: Karnaugh maps for $m_i$ for the redundant adder with format 2



| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | c | 1 | 1 | d |
| 01 | 1 | 0 | c | d |
| 11 | 1 | c | 1 | d |
| 10 | d | d | d | d |

| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | c | 0 | 0 | d |
| 01 | 0 | 0 | c | d |
| 11 | 0 | c | 1 | d |
| 10 | d | d | d | d |

Figure 2.6: Karnaugh maps for $b_i$ for the redundant adder with format 2



| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | d |
| 01 | 1 | 0 | 0 | d |
| 11 | 1 | 0 | 0 | d |
| 10 | d | d | d | d |

| $k_i^n k_i^p \backslash l_i^n l_i^p$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | d |
| 01 | 0 | 1 | 1 | d |
| 11 | 0 | 1 | 1 | d |
| 10 | d | d | d | d |

Figure 2.7: Karnaugh maps for $d_i$ for the redundant adder with format 2

With the introduction of "coupled don't cares", the logical design of the redundant binary adder will yield a simpler structure. The boolean functions of the redundant binary adder of format 2 are listed below:

$$d_i = m_{i-1} \oplus l_i^p \oplus k_i^p \qquad d_i \in \{0,1\} \qquad (2.6a)$$
$$m_i = \overline{l_i^n}\,\overline{k_i^n}(l_i^p \vee k_i^p) \qquad m_i \in \{0,1\} \qquad (2.6b)$$
$$b_i = \overline{m}_{i-1}\overline{l_i^p}k_i^p \vee \overline{m}_{i-1}l_i^p\overline{k_i^p} \vee l_i^n k_i^n \qquad b_i \in \{\bar{1},0\} \qquad (2.6c)$$
$$s_i^n = \bar{d}_i b_i \qquad (2.6d)$$
$$s_i^p = d_i \oplus b_i \qquad s_i^* \in \{\bar{1},0,1\} \qquad (2.6e)$$

The other formats for the redundant binary adder will not be discussed here as they were fully studied by Robertson [10]. The structure of the redundant binary adder is regular. It provides flexibility to the structure of the adder, in other words, the wordlength of the operands can be expanded easily without changing the addition time. This feature of the redundant binary adder is shown in Figure 2.8. And this particular feature is illustrated with an example on redundant binary addition as shown in Figure 2.9. This redundant binary adder structure will be used as the basis of the work in the following chapter.

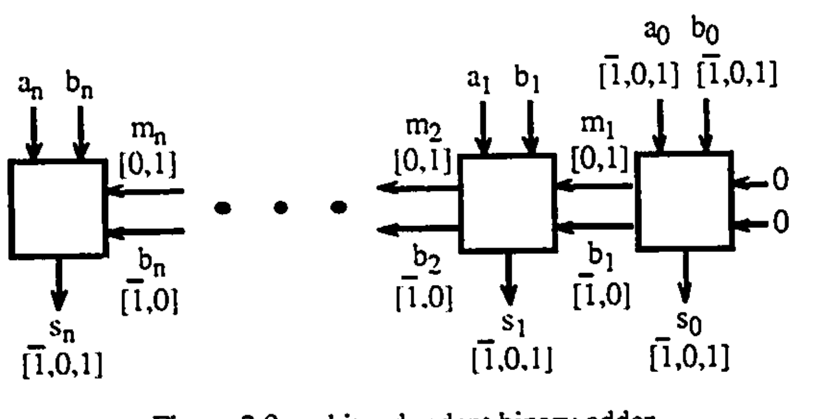
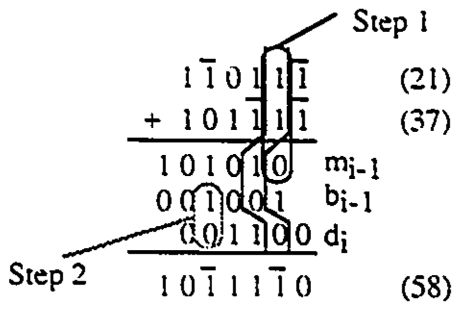


Figure 2.8: n-bit redundant binary adder

$l^* = 1\bar{1}011\bar{1}$ $k^* = 101\bar{1}1\bar{1}$

$m_{i-1}=0$ $b_{i-1}=0$ $i=0$


Step 1
$1\bar{1}01\bar{1}\bar{1}$ (21)
$+ 1011\bar{1}\bar{1}$ (37)
$10101 0$ $m_{i-1}$
$0011001$ $b_{i-1}$
$0011100$ $d_i$
Step 2
$10\bar{1}11\bar{1}0$ (58)

Step 1: obtain $m_i$, $b_i$ and $d_i$ from the truth table(Fig 3.) when $m_{i-1}=0$, $l_i^n=0$, $l_i^p=1$, $k_i^n=0$ and $k_i^p=1$ (i=1)

Step 2: addition of $b_{i-1}$ and $d_i$

Figure 2.9: Example of a redundant binary addition

## 2.2.3 (+,-) SCHEME SBNR ADDER

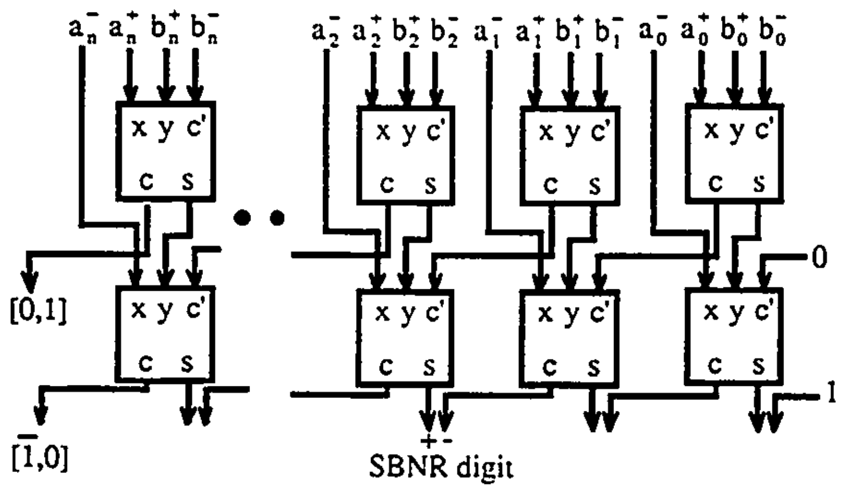

$a_n^-$ $a_n^+$ $b_n^+$ $b_n^-$
$a_2^-$ $a_2^+$ $b_2^+$ $b_2^-$ $a_1^-$ $a_1^+$ $b_1^+$ $b_1^-$ $a_0^-$ $a_0^+$ $b_0^+$ $b_0^-$
x y c'
c s
0
[0,1]
1
[$\bar{1}$,0]
+ -
SBNR digit

Figure 2.10: (+,-) scheme SBNR adder

As mentioned earlier in the section, (+,-) coding scheme SBNR adders [12] can be constructed from simple binary full adders. The circuit to add two SBNR operands $a=(a^+,a^-)$ and $b=(b^+,b^-)$ is shown in Figure 2.10 [12].

## 2.2.4 CONVERSION BETWEEN REDUNDANT BINARY AND BINARY NUMBERS

An n-bit unsigned binary number $[x_{n-1}x_{n-2}\ldots x_0]_2(x_i \in \{0,1\})$ and an n-digit redundant binary number $[x_{n-1}x_{n-2}\ldots x_0]_{SD2}(x_i \in \{\bar{1},0,1\})$ have the same value $\sum_{i=0}^{n-1} x_i 2^i$, since the value of binary number $\{0,1\}$ is a sub-set of the redundant binary number $\{\bar{1},0,1\}$. Therefore, no computation is required to convert an unsigned binary number into an equivalent redundant binary integer.

A conversion from an n-digit redundant binary number $X=[x_{n-1}x_{n-2}\ldots x_0]_{SD2}(x_i \in \{\bar{1},0,1\})$ into the equivalent binary number $Y=[y_{n-1}y_{n-2}\ldots y_0]_2(y_i \in \{0,1\})$ has to be performed, because the binary system is the standard representation used externally in most systems. The result, Y, is generated by subtracting $X^-$ from $X^+$, where $X^-$ and $X^+$ are n-bit unsigned binary integers formed from the positive digits and the negative digits in X respectively. This conversion can be performed easily by the following equation (2.7):

$$Y\left(=\sum_{i=0}^{n-1} y_i 2^i\right) = X^+\left(=\sum_{i=0,\, x_i=1}^{n-1} x_i 2^i\right) - X^-\left(=\sum_{i=0,\, x_i=\bar{1}}^{n-1} x_i 2^i\right) \tag{2.7}$$

$X = 1\bar{1}101\bar{1}$ (25)

$$\begin{tikzcd} & X^+\ 101010\ (42) \\ X\ \ 1\bar{1}101\bar{1}\ (25) \arrow[ur] \arrow[dr] & \\ & X^-\ 010001\ (17) \end{tikzcd}$$

$X^+ = 101010$ (42) $X^- = 010001$ (17)

$Y = X^+ - X^- = 42 - 17 = 25$

Figure 2.11: Example on SBNR-binary conversion

The conversion can be performed in a computation time proportional to $\log_2 n$ by means of a carry look ahead adder. The number of computation elements of a carry look ahead adder is proportional to n.

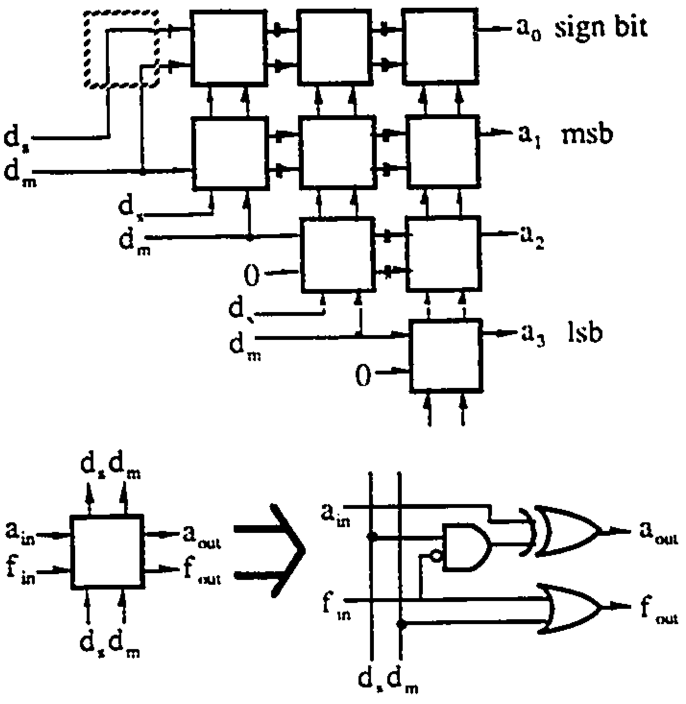


Figure 2.12: Conversion circuitry

Although this approach achieves the desired objective, the subtractor increases the delay through the chip. Therefore, an alternative scheme is needed which carries out the conversion while the data are being deskewed, and without affecting the latency. The 'on-the-fly' conversion was first proposed by Ercegovac and Lang [13] [14] and a modified version was later proposed by Knowles and McWhirter [15]. The conversion circuitry is shown in Figure 2.12, and the corresponding equations describing the conversion circuitry are:

If $f_{i(in)} = 1$: $a_{i(out)} = a_{i(in)}$; $f_{i(out)} = 1$ (2.8)

$$\text{If } f_{i(in)}=0:\ a_{i(out)}=\begin{cases} a_{i(in)} & \text{if} \quad d_j=0,1 \\ \overline{a_{i(in)}} & \text{if} \quad d_j=\overline{1} \end{cases} ;\ f_{i(out)}=\begin{cases} 0 & \text{if} \quad d_j=0 \\ 1 & \text{if} \quad d_j=\overline{1},1 \end{cases} \tag{2.9}$$

The operation of the array is explained in terms of columns of cells. As each digit can affect those of higher significance, each digit is broadcast back up to the relevant column. The estimate is represented by *a*-lines in each column. These bits are clocked across the array together with the control bit *f*. This control bit is initially set to 0. Whenever a nonzero digit appears at a lower significance, determining the value of the *a*-bits above it in the column, all the *f*-bits in these cells are set to 1. When the *f*-bit equals to 1, the estimate value of *a* cannot be altered. This ensures that a lower digit only affects the bits in an MSB direction up to the next 1, but not any higher significance bits. If the next lower significance is zero, then both value of *f* and *a* are unchanged. In any case, $|d_j|$ is appended to $a_{j-1}$. An example of the step by step conversion is given in Figure 2.13.

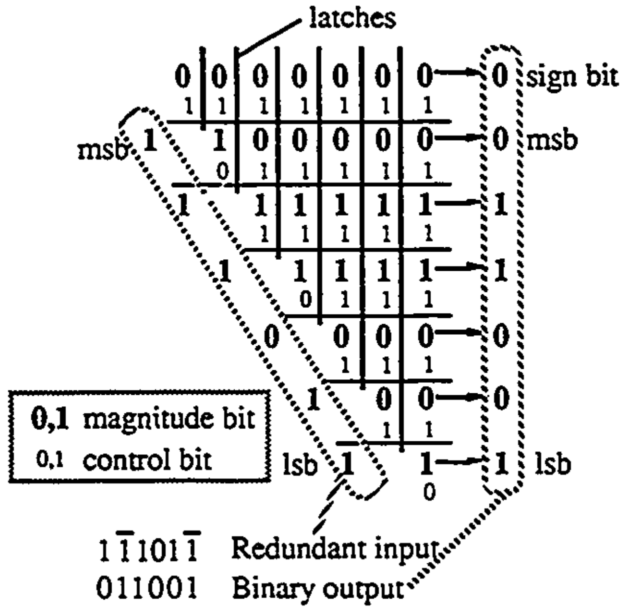


Figure 2.13: Step by step SBNR-binary conversion

### 2.2.5 REDUNDANCY OVERFLOW

One problem that arises with the SBNR is redundancy overflow. Redundancy overflow occurs when a SBNR number occupies more than the minimum number of signed digit positions. In other words, alternative representations of that number exist. For example, adding a zero to an n digit SBNR operand can yield an n+1 digit result, but it is obvious that an n digit result is sufficient. However, it is possible to compress the result back to an n digit word. In some cases, redundancy overflow can be accommodated by increasing the wordlength to n+1 digits or by extending the degree of redundancy occupied by the MSD, that is increasing the digit range of the MSD. An example of redundancy overflow in addition is shown in Figure 2.14.

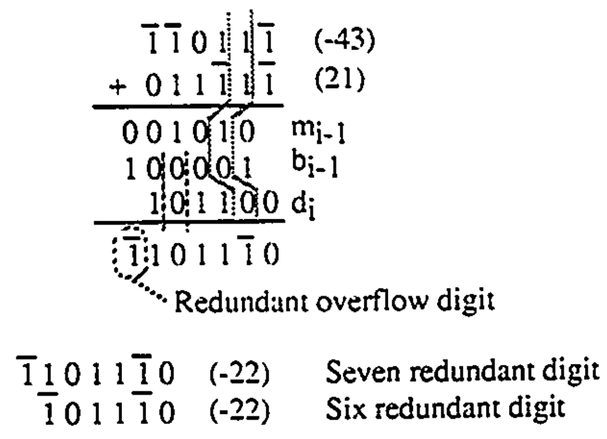

$\bar{1}\,\bar{1}\,0\,1\,1\,\bar{1}$ (-43)
$+\;0\,1\,1\,\bar{1}\,1\,\bar{1}$ (21)
$0\,0\,1\,0\,1\,0$ $m_{i-1}$
$1\,0\,0\,0\,0\,1$ $b_{i-1}$
$1\,0\,1\,1\,0\,0$ $d_i$
$\bar{1}\,1\,0\,1\,1\,\bar{1}\,0$
Redundant overflow digit
$\bar{1}\,1\,0\,1\,1\,\bar{1}\,0$ (-22) Seven redundant digit
$\bar{1}\,0\,1\,1\,\bar{1}\,0$ (-22) Six redundant digit

Figure 2.14: Example of Redundancy Overflow

## 2.3 REDUNDANT COMPUTER ARITHMETIC

Since the inception of computers, much effort has been expended in search of fast arithmetic techniques. These fast arithmetic techniques have been put into compact high-speed circuits as the computation units in various VLSI systems for real-time applications. One of the speed-up techniques is to introduce redundancy in the implementation of computer arithmetic. The judicious application of redundancy to the systems can increase the speed of operations and provide structural flexibility. Three algorithms that employ redundancy will be studied in the following sections. They are division, multiplication and square-root. Methods of operation will be discussed through the use of examples.

## 2.3.1 Division

Division is one of the most complex basic binary arithmetic operations. Several algorithms for fast computation of division of binary numbers have been proposed in the literature. They can be broadly classified into restoring and non-restoring algorithms. The restoring algorithm is another name for the pencil and paper division method that is usually taught in grade school. The restoring method of division requires the subtraction of the right shifted divisor from the scaled remainder at each step. If the partial remainder is negative, a quotient bit '0' is selected as the quotient bit and the original remainder is restored as the new remainder. Otherwise, a' 1' is selected as the quotient bit and the partial remainder is the new remainder. In any case, the divisor is right shifted one position. In the non-restoring algorithm, both addition and subtraction are used to avoid the restoring step when the partial remainder yields a negative number. Each step of the non-restoring division method requires the right shifted divisor to be either added to or subtracted from the scaled remainder depending on whether the quotient bit generated in the preceding row was a '0' or a '1' respectively. The introduction of redundancy into the algorithm can speed up the computation time. First, less time is needed to form the partial remainder since the carry propagation is limited. Also, with the quotient digit represented by redundancy resulting in a comparison between the partial remainder and the divisor need not be at the full precision. The most well-known non-restoring with redundancy method is the SRT division[16] [17]. SRT division originated from three initial proposers; D. W.

Sweeney of IBM, J. E. Robertson of the University of Illinois and K. D. Tocher of Imperial College. The first letter of each of their last names forms the acronym.

Performing division requires making a choice of quotient digits starting with the most significant, and proceeding to the least significant digits. A quotient digit is determined by estimating the partial remainder at each stage. The value of the quotient digit is chosen from the redundant digit set $\{\bar{1}, 0, 1\}$. The complete quotient is accumulated by the following equation:

$$Q = \sum_{i=0}^{n-1} q_i r^{-i} \tag{2.10}$$

| | |
|---|---|
| r: | radix |
| n: | number of quotient digits calculated |
| Q: | Accumulated quotient result |
| $q_i$: | quotient digit |

The quotient digit chosen at each stage in the division determines the operation of computing the next partial remainder according to the equation:

$$R_{i+1} = rR_i - Dq_i \tag{2.11}$$

| | |
|---|---|
| r: | radix |
| $R_i$: | partial remainder at stage i |
| D: | Divisor |
| $q_i$: | quotient digit |

The dividend is initialized with $rR_0$. In this method, the divisor and dividend must be normalized to the same binary range, and the valid quotient digits are in the set {-p,..,0,..,p} where p is restricted to be in the range of $\frac{r}{2} \le p \le r-1$ according to Atkins [37].

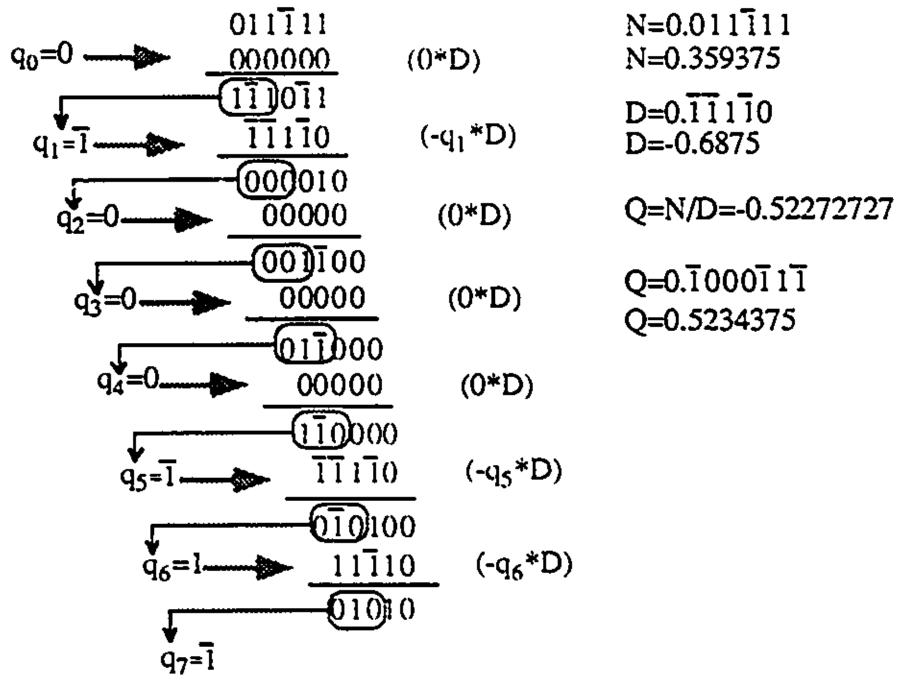


Figure 2.15: Division example

With redundancy introduced into the SRT division, each quotient digit need only use an approximation of the partial remainder, because small errors may be corrected with less significant quotient bits of the opposite sign. Since an approximation of the partial remainder is needed for the quotient bit selection, only a small number of the most significant bits need to be examined, and it can be proved that it is onlt necessary to examine 3 MSD's of the partial remainder [18]. A division example is shown in Figure 2.15.

## 2.3.2 Square Root

Various rapid square-rooting algorithms, based on the classical non-restoring method, are described by Metze [19] and Oklobdzija [20]. Metze's binary algorithms also give the square-root value in the notation with the digits -1, 0, 1. They are specially fitted to obtain the result with the minimal possible number of non-zero digits. Algorithms described in [20] also cover non-binary number notations and take into account a bigger number of various digits of the redundant notations of the square root.

Classical binary non-restoring square-rooting is very similar to classical binary non-restoring division method. In both cases, numbers are subtracted or added in successive process steps to decrease the successive partial remainders. The main difference among them is the way that these subtracted or added numbers are formed. But despite this difference, many division methods, being modifications of the classical non-restoring division, can be respectively adapted and used as square-rooting methods. Therefore, the concept of redundancy in division can be applied to the square-rooting method in a similar manner. As in the division method, the square-rooting method is accelerated by decreasing the time to form a partial remainder. In addition, only part of the partial remainder is needed to examine for the root digit extraction. The algorithm described by Majerski [21] is described in this section.

The $R_j$ is the radicand and assumed to be in the range $\frac{1}{4} \le R < 1$. and consequently the square root S is normalized: $\frac{1}{2} \le S < 1$. The accuracy of the partial root, $S_j$, at the jth step is given by:

$$\left|\sqrt{R_j} - S_j\right| < 2^{-j} \tag{2.12}$$

The scaled remainder at the jth step can be defined as follows:

$$Z_j = 2^j(R - S_j^2) \tag{2.13}$$

Therefore, the recurrence for computing successive remainders is:

$$Z_j = 2Z_{j-1} - s_j(2S_{j-1} + s_j 2^{-j}) \quad \text{where } j=1,2,3,..n \tag{2.14}$$

At the j-1th step, $s_j$, the root digit wiil be determined according to some criterion and the partial remainder $Z_j$ is formed by the equation specified above. The term in brackets defines the root digit extractor. The initial remainder $Z_0$, is set to the radicand R and the initial estimate of the square root is $S_1 = \frac{1}{2}$. The successive root digit extractors are in the form specified by Table 2.4 given below:

| Step | Root digit extractors |
|---|---|
| j=1 | $0.s_1 0000....$ |
| j=2 | $s_1.0s_2000....$ |
| j=3 | $s_1.s_2 0s_3 00....$ |
| j=4 | $s_1.s_2 s_3 0s_4 0....$ |

Table 2.4: Format of root digit extractors

A complete example is given below in Figure 2.16 to illustrate the procedures:

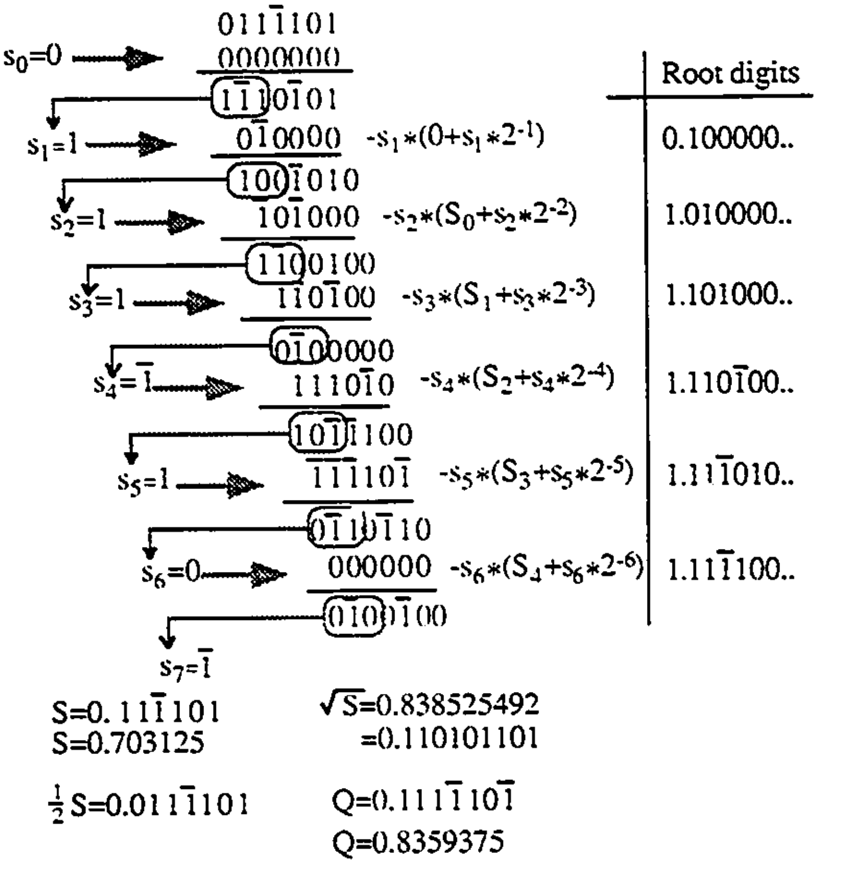


Figure 2.16: Example of square-root

### 2.3.3 Multiplication

Multiplication is one of the vital computer arithmetic operations in many digital applications such as digital signal processing, process control and computer graphics. High-speed multipliers are essential in real time signal processing systems providing filtering, correlation, and range measurement. As a result, various high-speed multipliers have been proposed and designed on a single-chip LSI [11] [22].

In practice, there are three common multiplier schemes: array multiplier, redundant binary tree, and Wallace tree[23]. The shift-add algorithm is a familiar multiplication method. Parallel multipliers based on this algorithm have been widely used, i.e. the array multiplier. Again, employing redundancy into the multiplication can speed up the computation time, since the partial product is formed independently of the wordlength. Based on the shift-add algorithm, Ercegovac [13] proposed a similar algorithm to perform the multiply-accumulate operation. The general multiply and add operation can be expressed as:

$$M = X \cdot Y + A \tag{2.15}$$

Here, it is assumed that the multiplicand, X, is known at full precision at the start of computation, whereas the multiplier Y and the addend A are assumed to be available in a digit-by-digit manner. For simplicity, it is assumed that all the operands are in the range specified by Table 2.5.

| Operands | Ranges |
|---|---|
| Multiplicand X | $\frac{1}{4} \le \lvert X\rvert < \frac{1}{2}$ |
| Multiplier Y | $\frac{1}{2} \le \lvert Y\rvert < 1$ |
| Addend A | $\frac{1}{4} \le \lvert A\rvert < \frac{1}{2}$ |

Table 2.5: Ranges of the operands

The partial multiplier $Y_j$, accumulated at the jth step is defined as:

$$Y_j = Y_{j-1} + y_j 2^{-j} = \sum_{i=1}^{j} y_i 2^{-i} \qquad (2.16)$$

$Y_j$ is a word representing the digits of the multiplier available at the jth step. The partial addend $A_j$ is similarly defined. The residual function at the jth step, $Z_j$ is defined as:

$$Z_j = 2^{j-\delta}(X \cdot Y_j + A - M_{j-\delta}) \quad j=1,2,3,..n \qquad (2.17)$$

where $M_{j-d}$ is the partial result and n is the number of digits in the multiplier word, the scaling factor $2^{j-\delta}$ is introduced for convenience. The resulting radix 2 recurrence for computing the multiply-accumulate expression $M = X \cdot Y + A$ is listed below:

$$Z_j = 2Z_{j-1} + 2^{-\delta} X \cdot y_j - m_{j-\delta} \quad j=1,2,3,..n \qquad (2.18)$$

$Z_j$ is the residual at the jth step, $m_{j-\delta}$ is the result digit determined on the preceding step where the digit is selected from SBNR digit set $\{\bar{1},0,1\}$, $y_j$ is the jth multiplier digit and the initial residual is $Z_0 = 2^{-\delta}A$. The latency of the algorithm $\delta$, for a redundant representation of the residual is given by:

$$\delta > \log_2(4 \cdot |X|_{max}) \qquad (2.19)$$

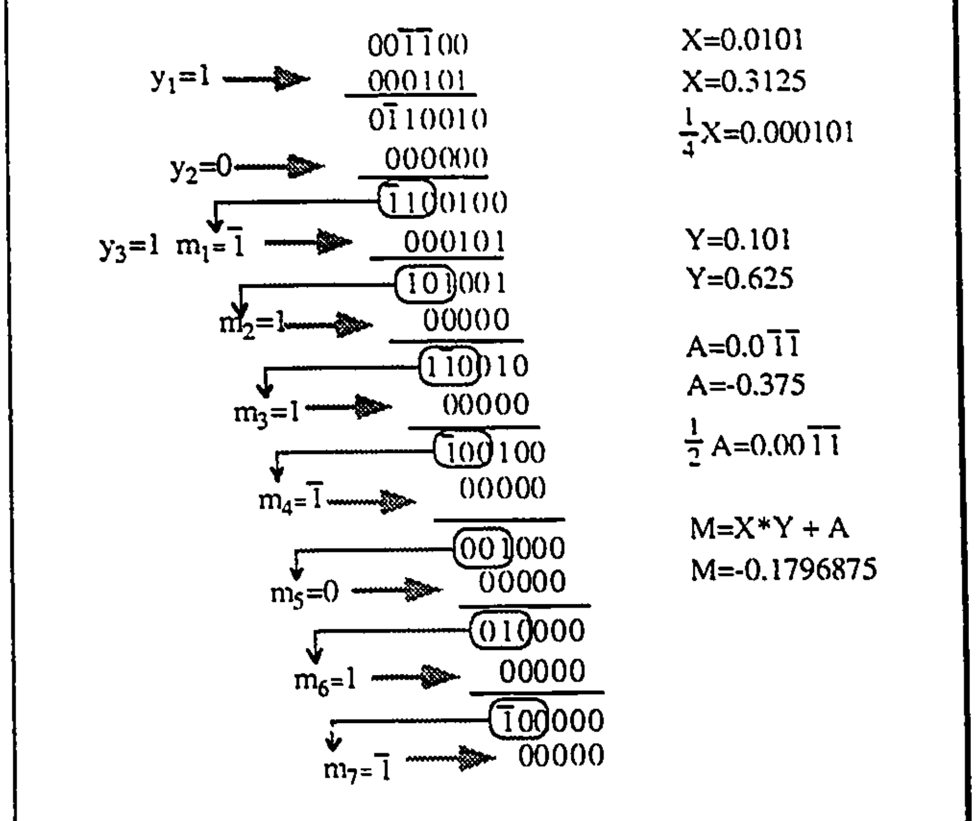


Figure 2.17: Example on Multiply-Accumulate

where $|X|_{max}$ is the maximum value of the multiplicand. Result digits can be determined by examining a low precision estimate of the residual at each step. For the ranges of operands specified in reference [13], since $|X|_{max} = \frac{1}{2}$ then from equation 2.19, the latency of the multiply-accumulate operation is $\delta=2$. A complete example is given in Figure 2.17 to illustrate the procedure.

## 2.4 SUMMARY

The purpose of this chapter is to introduce the basic concept of Signed-Digit Number Representation proposed by Avizienis [6]. Three redundant binary adders have been studied in detail, but only the redundant binary adder proposed by Robertson will be used as the basic foundation of the arithmetic architecture in the following chapter. Methods to handle redundancy overflow and conversion between signed-binary number to conventional binary number have been addressed. Three algorithms, multiplication, division and square rooting, which employ redundancy have been discussed and examples have been given.

# Chapter 3

# VLSI ARCHITECTURES FOR ARITHMETIC OPERATIONS

## 3.1 INTRODUCTION

In digital signal processing(DSP), there is an established need for fast and efficient hardware implementations of multiplication, division and square root operations. Over the years, a substantial effort has been expended in the design of fast hardware multipliers, since multiplication is the most frequently used operation in real-time digital signal processing. In this chapter, we will present a detailed discussion of the design of multiple operation arithmetic blocks.

This chapter can be classified into three main sections. The first section will briefly discuss the basic concept of systolic arrays. In the second section, a published architecture that performs multiply-accumulate, divide and square-root operations will be discussed. The architecture has a high throughput rate; the execution time is the same for each operation and is independent of wordlength. In the last section, an improved architecture that provides multiply-accumulate, divide, square-root, as well as addition and subtraction is presented. The throughput rate is estimated at 100 megasamples per second, utilizing a 0.8μm BiCMOS standard cell library. The improved architecture requires lower area, fewer cells, and is faster than the original architecture. In addition, the improved architecture is capable of being implemented using the dynamic switching tree technique, one of the major focus areas which members of the VLSI Group at the University of Windsor are actively pursuing.

## 3.2 BIT-LEVEL SYSTOLIC ARRAY

In digital design techniques, much attention has been paid to network design in the form of a repeated pattern of identical circuits. Kung and Leserson [24] has made a notable contribution of a class of pipelined, parallel processing arrays known as systolic arrays. A simple systolic array comprises a regular array of modules, each module containing limited memory and being connected to its nearest neighbours. These modules are usually identical, although some may differ on the boundary of the array. Three typical connected patterns are shown in Figure 3.1.

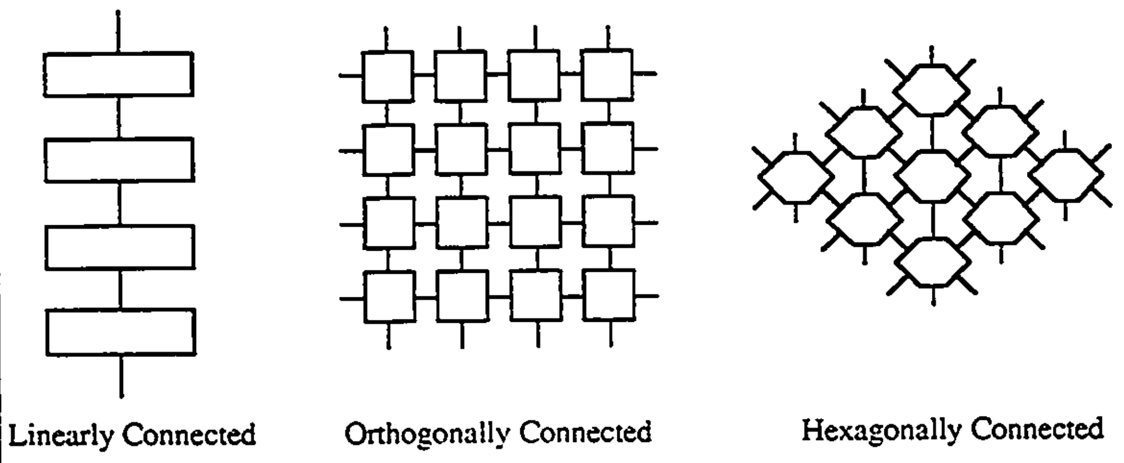


Figure 3.1: Three typical connected systolic arrays

They are the linearly connected, the orthogonally connected and the hexagonally connected array. On each cycle, each module or processing element (PE) performs a specific computation and stores the result, and the result previously stored in each PE is sent out to a neighbouring PE. All operations within the systolic array are synchronized and provided by a global clock. In this action, data are pumped rhythmically across the array. This is similar to the pumping action of the heart, hence the term "systolic".

The systolic array concept has since been extended to a higher level of granularity by McCanny and McWhirter who introduced the concept of bit-level systolic arrays [25]. In the latest designs, McCanny, Woods, Knowles and McWhirter present a number of papers dealing with the bit-level systolic architecture on different DSP applications including convolution [26] and correlation [27]. Bit-level systolic arrays retain the temporal and spatial locality characteristics associated with word-level systolic arrays and are particularly well suited for large-scale integration. Each PE in the architecture is a very simple operation and local memory is provided by latches.

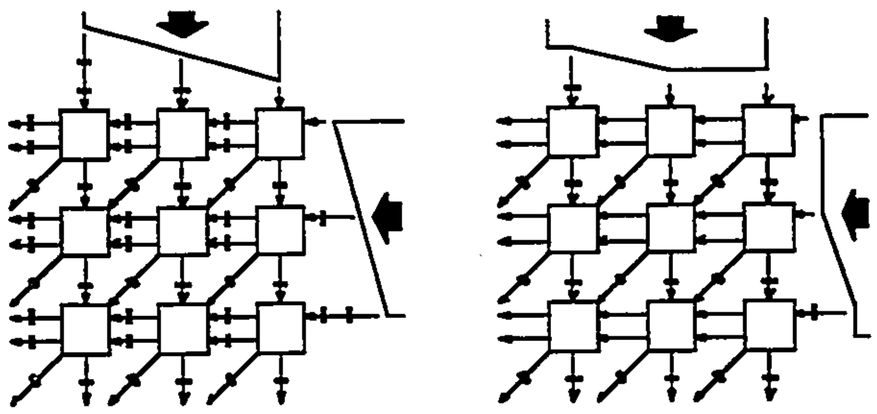


Figure 3.2: Bit-Level Systolic and Pipeline Arrays

The substantial throughput rates offered by systolic arrays can only be utilized, however, when the whole system can match similar levels of performance. For VLSI designs, many architectures are designed as pipelined array and linear array processors. Such arrays illustrate the desirable features of bit-level systolic arrays, including regularity and locality of communication. One of the important consequences is that data are now allowed to be broadcast in each pipelined stage. Pipeline arrays rather than bit-level systolic arrays are more commonly present in modern commercial DSP chips. Figure 3.2 shows the difference between the two arrays.

## 3.3 ADD-SHIFT-EXTRACT CONFIGURATION

In this section we discuss the design of pipeline arrays to perform add, shift and extract operations. The shift and add function is basic, to many arithmetic operations. For example, in the division method, the divisor is subtracted from the dividend and the divisor is shifted one bit towards the lower-order significant (one bit to the right) before entering the next stage. The quotient digit can be determined from the partial remainder and this operation will continue until the desired result is achieved. This method is shown in Figure 3.3. The other arithmetic operations proceed in a similar fashion.

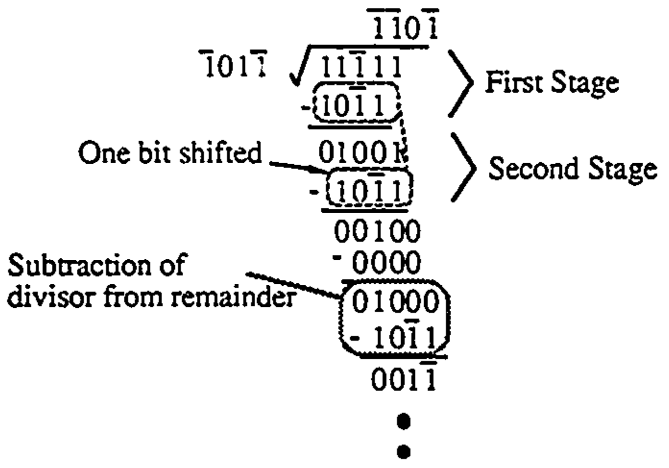


Figure 3.3: Division method

In order to perform these computer arithmetic operations on a VLSI architecture, it is necessary to map the add-shift configuration into the hardware. The first step is to partition the word level operand into the digit level operand. For an n-digit precision operand and result, it is obvious that the minimum requirement is an array of $n \times n$ blocks in which each block can perform digit level addition between two digit operands. Each row of the array can be pipelined to maximize the throughput rate and this results in a digit-by-digit operation at the end of each pipelined stage. In many of the architectures, data are broadcast into each pipelined stage of the array, so that multiplication and addition can be performed concurrently in every pipelined stage. The connection between each pipelined stage allows shifting to be performed one-bit to the left when data are passing from the preceding stage to the succeeding stage. The configuration of the add-shift-extract operation is illustrated in Figure 3.4. A number of architectures dealing with this configuration have been proposed [5] [11].

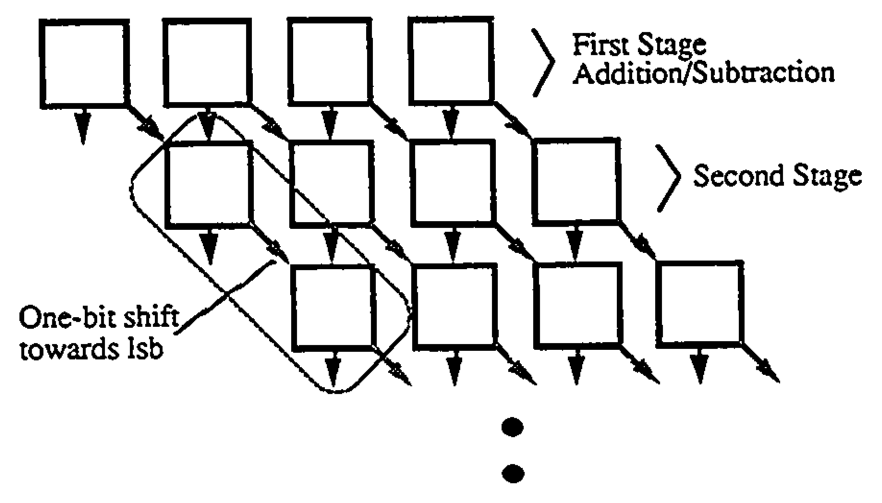


Figure 3.4: Add-Shift-Extract configuration

## 3.4 THE UNIFIED ALGORITHM

In order to develop a VLSI architecture for the combined operations of multiply-accumulate, divide and square root, an common algorithm, that performs these three operations, must first be established. We will use the previously discussed reduncdant arithmetic in order to achieve high performance by computing from the most-significant digit first.

## 3.4.1 RATIONALE FOR REDUNDANT ARITHMETIC

Numerous algorithms of division and square root are inherently computed most-significant bit first, since bits of higher significance must be computed first before lower-order bits can be determined. Therefore, it is possible to implement these algorithms using a conventional (i.e. non redundant) arithmetic. However, one major disadvantage of using conventional arithmetic is that it is intrinsically slow for such operations. At each iteration of the divide/square root algorithm, carry digits must propagate from the most-significant bit (MSB) to the least-significant bit (LSB) before the output digit can be determined. The problem associated with carry propagation is shown in Figure 3.5. As discussed previously, this carry propagation from MSB to LSB can be eliminated by employing arithmetic redundancy, e.g. carry-save or the signed binary number representation [6], and performance can be increased through parallel addition/subtraction.

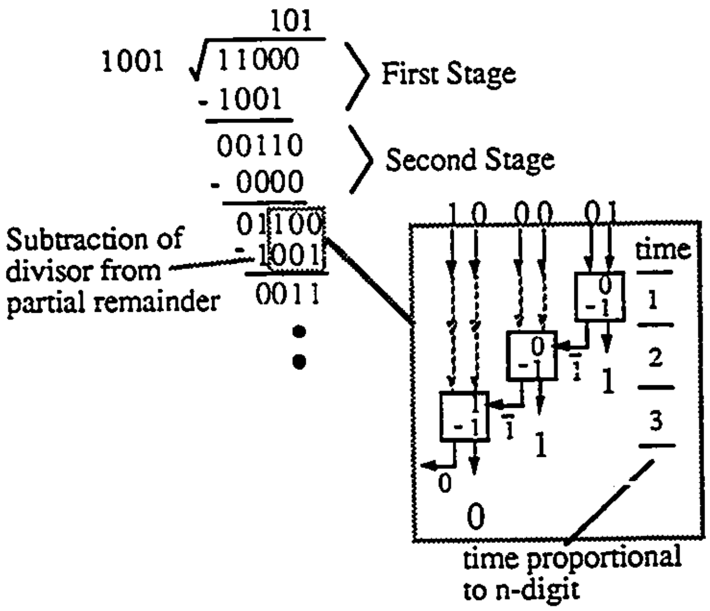


Figure 3.5: Divide/Square root (conventional) carry-propagate

In a conventional binary repeated-addition algorithm for multiply-accumulate operation, usually operating under least-significant bit fashion, the computation of most-significant bit involves the carry digits to be propagated from the lower order partial product. This operation is fully demonstrated in Figure 3.6.

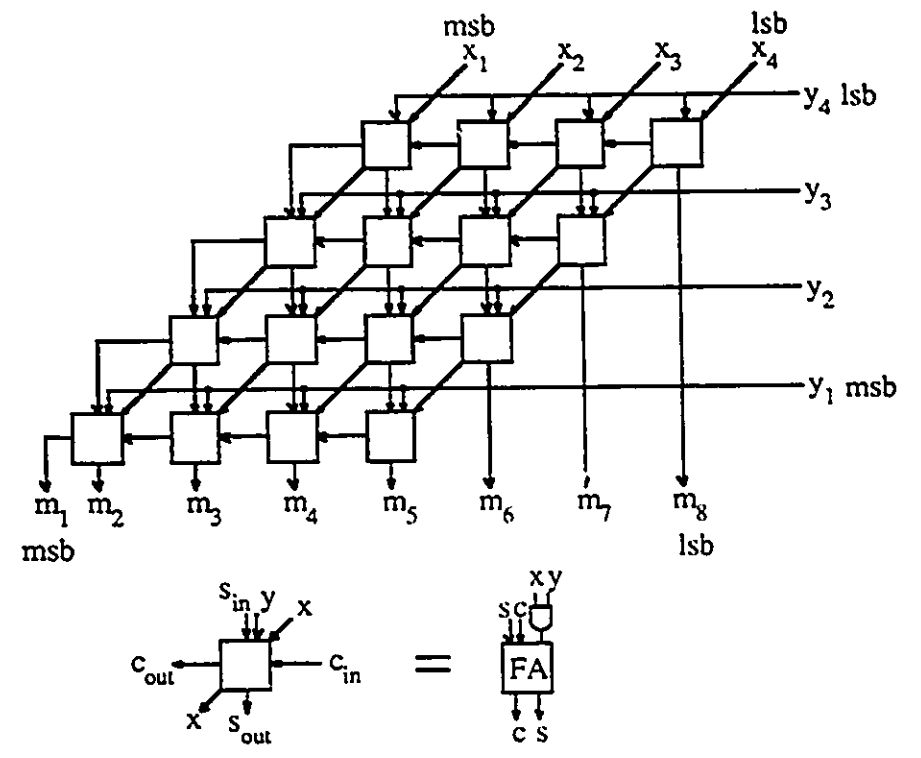


Figure 3.6: Conventional method for multiply-accumulate

In order to unify three operations in one algorithm, it is advantageous to compute the result of the multiply operation iteratively, most significant digit (MSD) first. MSD computation is also useful if only half of the most significant digits are required. To compute the most significant digit in a digit-by-digit manner, MSD first, sufficient flexibility must be introduced into the accumulating result to allow the effect of lower-order partial products which are not known yet. In other words, provision must be established so that initial over/under estimates of the result can be compensated for later in the computation when lower-order partial products become available. This computation flexibility is obtainable by computing the result in redundant form.

## 3.4.2 UNIFIED ALGORITHM

In the previous chapter (Chapter 2), three algorithms have been introduced through the use of examples. The radix 2 SRT division method[16], the analogous square root algorithm[21] and the multiply-accumulate algorithm[28] have been used as the basis of the architecture. All algorithms exploit redundancy in the representation of the operands and results digits. Three algorithms 3.1, 3.2 and 3.3 can be re-written in a similar format as follows:

$$Z_j = 2Z_{j-1} + y_j \cdot \tfrac{1}{4}X - m_{j-2} \qquad \text{if mult-accum} \qquad (3.1)$$
$$Z_j = 2Z_{j-1} - q_{j-1} \cdot D \qquad \text{if division} \qquad (3.2)$$
$$Z_j = 2Z_{j-1} - q_{j-1} \cdot (Q_{j-2} + q_{j-1}2^{-j}) \qquad \text{if square root} \qquad (3.3)$$

These three algorithms for computing successive residuals can then be written collectively in a simple unified algorithm as follows:

$$Z_j = 2Z_{j-1} - b_j \cdot C_j + n_j \qquad (3.4)$$

where $Z_j$ is the residual and $C_j$ is the result digit extractor at jth step. The recurrence parameters $C_j$, $b_j$ and $n_j$ are defined in Table 3.1. The product $b_j \cdot C_j$ defines the result digit extractor multiple. Note that upper case variables refer to a complete word while lower case variables refer to a individual digit of a word.

| Operation | $C_j$ | $b_j$ | $n_j$ |
|---|---|---|---|
| Division | D | $q_{j-1}$ | 0 |
| Mult-accum | $\frac{1}{4}X$ | $-y_j$ | $-m_{j-2}$ |
| Square root | $Q_{j-2} + q_{j-1}2^{-j}$ | $q_{j-1}$ | 0 |

Table 3.1: Definition of recurrence parameters

The ranges of the operands for each operation are as shown in table 3.2. Note that each operand is a SBNR number, therefore both positive and negative operands are permitted.

| Operation | Operands | |
|---|---|---|
| Mult-accum $M = X \cdot Y + A$ | $\frac{1}{4} \le \lvert X \rvert, \lvert A \rvert < \frac{1}{2}$ | $\frac{1}{2} \le Y < 1$ |
| Division Q=N/D | $\frac{1}{4} \le \lvert N \rvert < \frac{1}{2}$ | $\frac{1}{2} \le \lvert D \rvert < 1$ |
| Square Root $Q = \sqrt{S}$ | $\frac{1}{4} \le S < 1$ | |

Table 3.2: Ranges of Operands

The initial values for the unify algorithm 3.4, parameters $2Z_0$, $C_0$, $b_0$ and $n_0$ are summarized in table 3.3.

| Operation | $2Z_0$ | $C_0$ | $b_0$ | $n_0$ |
|---|---|---|---|---|
| Division | N | D | 0 | 0 |
| Mult-accum | $\frac{1}{2}A$ | $\frac{1}{4}X$ | $-y_1$ | 0 |
| Square Root | $\frac{1}{2}R$ | 0 | 0 | 0 |

Table 3.3 Initial values of the parameters

## 3.5 ORIGINAL ARCHITECTURE

This architecture was originally proposed by McQuillan and McCanny [5]. The architecture is an array that performs multiplication, division and square-root operations.

The basic function of the main array is to perform addition; with some special connections and the use of multiplexing, different operations can be performed. In addition, with the use of a redundant number system and pipelining, the execution time to perform each operation is identical and does not depend on the wordlength. The circuit is highly regular, requires only two control bits and can be reconfigured on each cycle. There are six different types of cell modules in the architecture, and each cell module serves a different function.

## 3.5.1 REDUNDANT ADDITION

Based on the concept purposed by Avizienis [6], McQuillan and McCanny [5] proposed another redundant addition method. Now, all operands and results from the array have a digit set $\{\bar{2},\bar{1},0,1\}$. Each SBNR digit requires a 2-bit representation, namely sign bit and magnitude bit. The SBNR digit is encoded similarly to the sign-magnitude scheme described in Chapter 2, but with (1,0) denoting $\bar{2}$ instead of X or 0. The encoded scheme is given in Table 3.4.

| Value | Code (t s) |
|---|---|
| 0 | (0 0) |
| 1 | (0 1) |
| -1 ($\bar{1}$) | (1 1) |
| -2 ($\bar{2}$) | (1 0) |

Table 3.4: Encoding scheme for the original architecture

The result digit $z_{out}(t_{out}, s_{out})$ depends only on two adjacent operands. And the operation of the redundant binary adder is described by the following equations (3.5a-3.5b):

$$2t'_{out} + w = p + s_{in} + t_{in} \qquad p = -b_{in} \cdot c_{in} \qquad (3.5a)$$

$$2t_{out} + s_{out} = w + t'_{out} \qquad (3.5b)$$

where $w \in \{-2,-1,0\}$, $t' \in \{0,1\}$, $t_{out} \in \{-1,0\}$

$s_{out} \in \{0,1\}$, $b_{in}, c_{in} \in \{\bar{1},0,1\}$

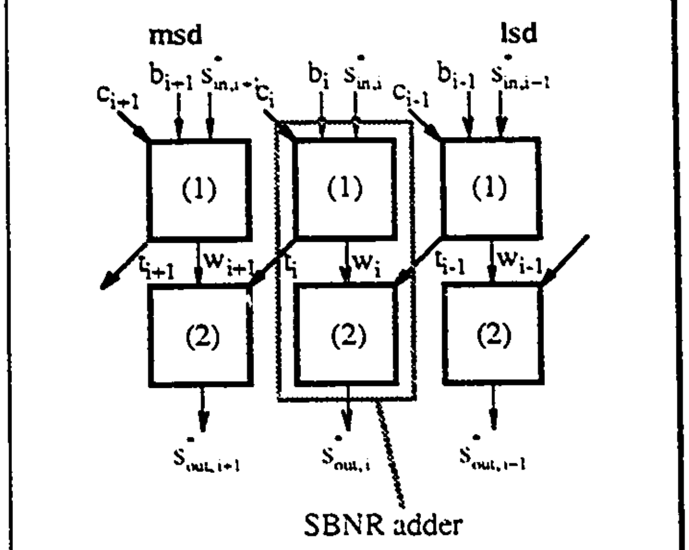


Figure 3.7: Structure of the redundant adder

This parallel addition is performed in two stages. The first stage of the adder cell accepts operands $p$, a sum digit $s_{in}$ and a transfer digit $t_{in}$ and performs the addition in the form of equation 3.5a. Note that $p$ is the multiplication between two SBNR operands, $b_{in}$ and $c_{in}$. The interim transfer digit $t'_{out}$ is passed to the second stage of the adjacent adder cell. In the second stage, the adder cell computes the output sum $s_{out}$ and transfer $t_{out}$ digits by adding interim transfer digit $t'_{in}$ from the right adjacent cell and the interim sum $w$ in the form of equation 3.5b. The structure of this adder is shown below in Figure 3.7

| $t_{in}$ | $s_{in}$ | p | $t'_{out}$ | $w$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0<br>1 | 0<br>-2 |
| 0 | 0 | 1 | 1 | -1 |
| 0 | 1 | 0 | 1 | -1 |
| 0 | 0 | -1 | 0 | -1 |
| 0 | 1 | -1 | 0<br>1 | 0<br>-2 |
| $\bar{1}$ | 0 | 0 | 0 | -1 |
| $\bar{1}$ | 0 | 1 | 0<br>1 | 0<br>-2 |
| $\bar{1}$ | 1 | 0 | 0<br>1 | 0<br>-2 |
| $\bar{1}$ | 1 | 1 | 1 | -1 |
| $\bar{1}$ | 0 | -1 | 0 | -2 |
| $\bar{1}$ | 1 | -1 | 0 | -1 |

Table 3.5: Values for interim transfer and interim sum

| $w$ | $t'_{out}$ | $t_{out}$ | $s_{out}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| -1 | 0 | -1 | 1 |
| -1 | 1 | 0 | 0 |
| -2 | 0 | -1 | 0 |
| -2 | 1 | -1 | 1 |

Table 3.6: Values of Transfer and Sum digit

The interim transfer $t'_{out}$ and sum $w$ digits can be computed according to equation 3.5a, and the values are shown in Table 3.5. In the second stage of the adder, values of the sum $s_{out}$ and transfer digit $t_{out}$ are computed by free addition of the interim digit $w$ and interim transfer digit $t'_{out}$, and the results of $s_{out}$ and $t_{out}$ are computed as illustrated in Table 3.6 respectively.

## 3.5.2 ARCHITECTURE

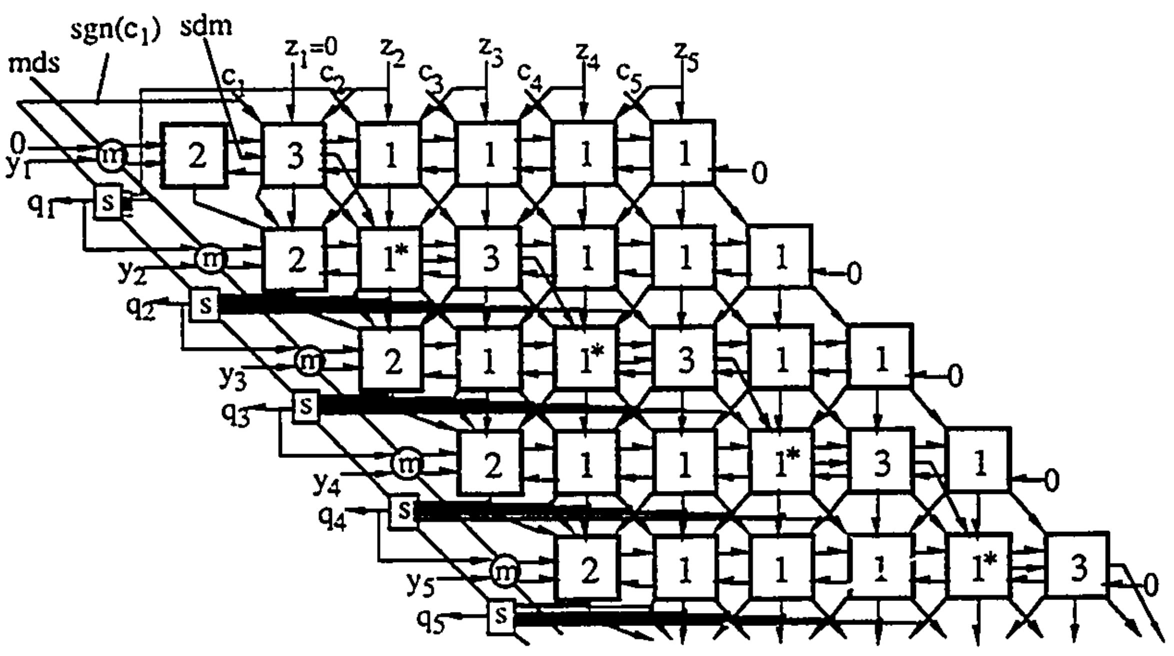


Figure 3.8: VLSI Architecture for Multiplication, Division and Square Root

An architecture to implement the unified algorithm is illustrated in Figure 3.8. Note that this architecture is performing 5x5 bit operations. The circuit comprises a pipelined array of SBNR multiply-add cells (cell types **1**, **1*** and **3**) bounded at the left hand edge by type **2** cells. Cells **1*** have additional multiplexing circuitry required for the square root operation. The S cells on the periphery of the main array implement the result digit selection function given in Table 3.7.

| Range of Partial Remainder | $Z \le -\frac{1}{2}$ | $\lvert Z\rvert \le -\frac{1}{4}$ | $Z \ge \frac{1}{2}$ |
|---|---|---|---|
| Result digit ($C \ge 0$) | $\bar{1}$ | 0 | 1 |
| ($C < 0$) | 1 | 0 | $\bar{1}$ |

Table 3.7: Selection function for Original Architecutre

The functional descriptions of the basic processing cells are given in Figure 3.9. The small black squares represent pipelining delays (one latch). Two control bits, denoted by sdm and mds are required to select between the three operations. Figure 3.8 illustrates a 5x5 bit architecture, but can be expanded to any wordlength by incorporating the appropiate cells.

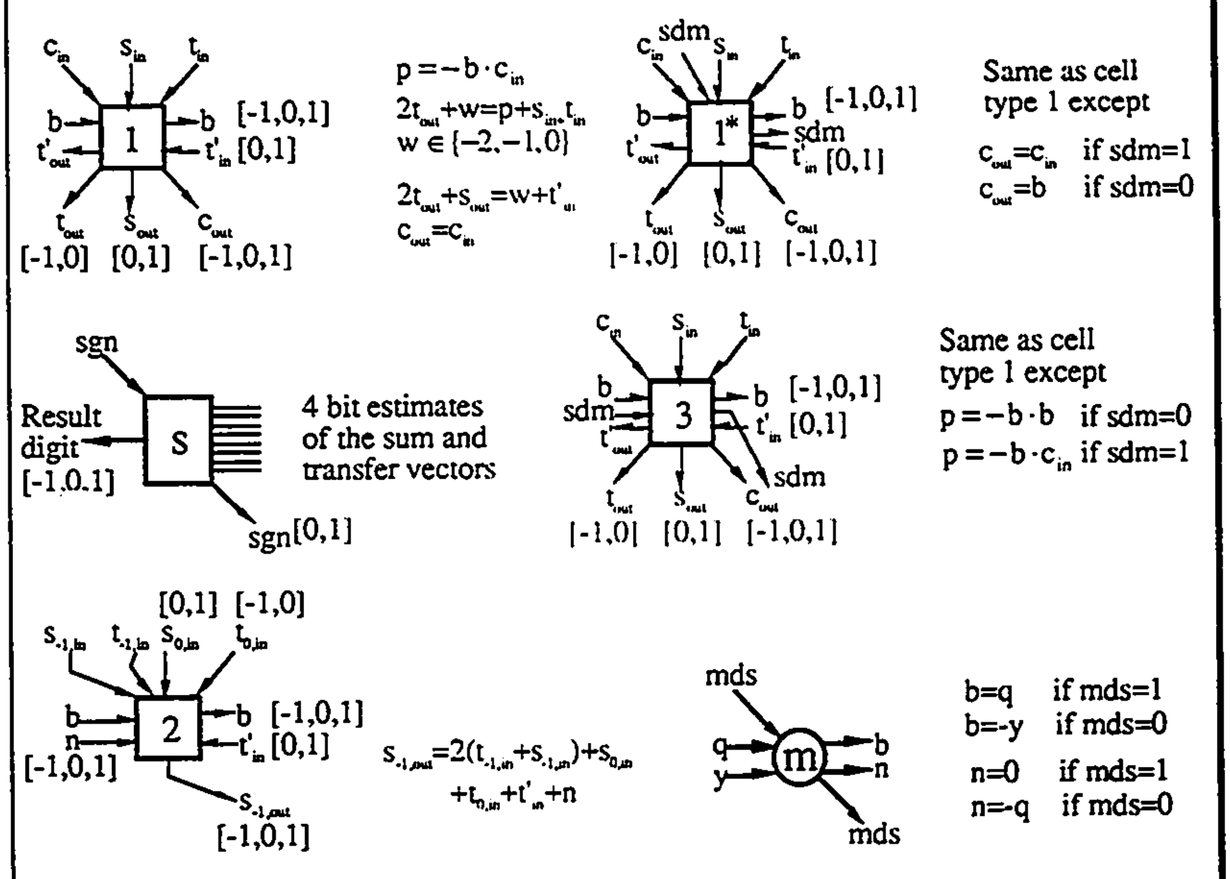

$c_{in}$ $s_{in}$ $t_{in}$
b [-1,0,1]
$t'_{out}$ $t'_{in}$ [0,1]
1
$t_{out}$ $s_{out}$ $c_{out}$
[-1,0] [0,1] [-1,0,1]
$p = -b \cdot c_{in}$
$2t_{out} + w = p + s_{in} + t_{in}$
$w \in \{-2, -1, 0\}$
$2t_{out} + s_{out} = w + t'_{in}$
$c_{out} = c_{in}$
sdm
1*
Same as cell type 1 except
$c_{out} = c_{in}$ if sdm=1
$c_{out} = b$ if sdm=0
sgn
Result digit [-1,0,1]
S
4 bit estimates of the sum and transfer vectors
sgn[0,1]
3
Same as cell type 1 except
$p = -b \cdot b$ if sdm=0
$p = -b \cdot c_{in}$ if sdm=1
[0,1] [-1,0]
$s_{-1,in}$ $t_{-1,in}$ $s_{0,in}$ $t_{0,in}$
2
n
[-1,0,1]
$s_{-1,out}$
$s_{-1,out} = 2(t_{-1,in} + s_{-1,in}) + s_{0,in} + t_{0,in} + t'_{in} + n$
mds
q
y
m
b
n
b=q if mds=1
b=-y if mds=0
n=0 if mds=1
n=-q if mds=0

Figure 3.9: Functional description of the basic cells

All operands and results from the array are signed binary numbers, each digit assuming values from the digit set $\{\bar{2},\bar{1},0,1\}$. The array architecture accepts three input words, namely, the multiplier Y, the initial extractor $C_0$ and an initial residual $Z_0$; this is illustrated in Figure 3.10.

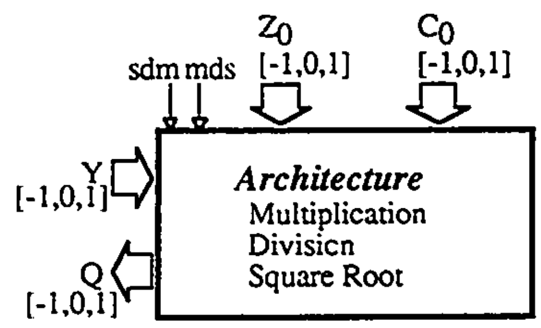


Figure 3.10: Original architecture functionality

These are set equal to the appropriate operands to enable the array to perform one of the three operations. The correspondence between the operation of the array and the control bits is summarized in Table 3.8.

| Operation | M/DS | S/DM | sgn($C_0$) |
|---|---|---|---|
| Mult-Accum | 0 | 1 | 0 |
| Division | 1 | 1 | sgn(D) |
| Square Root | 1 | 0 | 0 |

Table 3.8: Control bits of the original architecture

Each iteration of the algorithm is implemented by one row of the array, and the number of rows and columns determine the digit of precisions. Consider the jth row: this row accepts two SBNR input words, the scaled residual $2Z_{j-1}$ and the extractor $C_{j-1}$, from the preceding row. Moreover, two SBNR digits $b_j$ and $n_j$ are broadcast by type **m** cells appended to the jth row. The digit $b_j$ is broadcast to all cells in jth row enabling the extractor multiple to be determined. Concurrently, digit $n_j$ is subtracted from the msd of the residual by a type 2 cell. One important issue which has to be addressed is that of redundancy overflow in each step from the addition of the redundant numbers. Although in each iteration, the residual is only a fractional value, that is $|Z_j| < 1$, the residual representation of the residual can still overflow into the integer positions (see Chapter 2). In the architecture, redundancy overflow is accommodated by extending the number of integer positions from one to two digits in the range of the msd of the residual $Z_0$. The initial $Z_0$, which is derived from the array inputs, is expressed in transfer-sum form by treating the sign bit as transfer digit and the magnitude bit as sum digit. This is clearly illustrated in Figure 3.8. To ensure valid result digit selection, the S cell, in general, examines the four MSDs from each of the sum and transfers vectors (9 binary lines in total) together with the coefficient sign bit $sgn(c_1)$. However, for the first S cell, only one transfer and one sum digit are needed to be examined, namely the most significant sign bit of the initial residual $Z_0$ and the M/DS control bit. This ensures that for division and square root, $|q_0| = 1$ can be achieved and for multiply-accumulate, $q_1$=0 as required. Finally, all the results are computed in a digit skewed-parallel manner. The specific operation of the array is now outlined for each operation.

## Division:

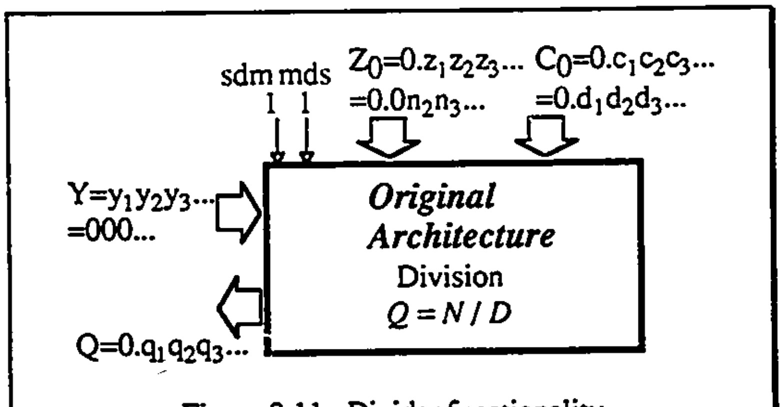


Figure 3.11: Divider functionality

To perform a division operation, S/DM and M/DS are both set to 1. This configures the cell types **1*** and **3** as type **1** cells enabling the divisor to be passed to each row of the array unchanged. Two input operands, N and D are needed for the operation; this is illustrated in Figure 3.11.

The scaled residual $2Z_0$ is initialized to the dividend $N = 0.0n_2n_3...$ and the quotient digit extractor $C_0$ is set to the divisor D. The S cell control bit $sgn(c_1)$ then equals the sign of the divisor. Each row operates as follows: the quotient digit $q_{j-1}$ from the previous j-1th row is selected by the multiplexer **m** as the broadcast digit for the jth row of cells and quotient extractor $-q_{j-1}D$ is computed. The new residual $Z_j$ is then determined by parallel addition of the quotient extractor and the previous sum and transfer digit vectors. And this parallel addition is performed as described in the previous section 2.2.2.

## Square Root:

The square root operation is selected when the control bits M/DS and S/DM are set to 1 and 0 respectively. The root digit extractor C is initialized to 0.00... while the scaled residual $2Z_0$ assumes the value $\frac{1}{2}R = 0.0r_1r_2....$ and this is clearly illustrated in Figure 3.12.

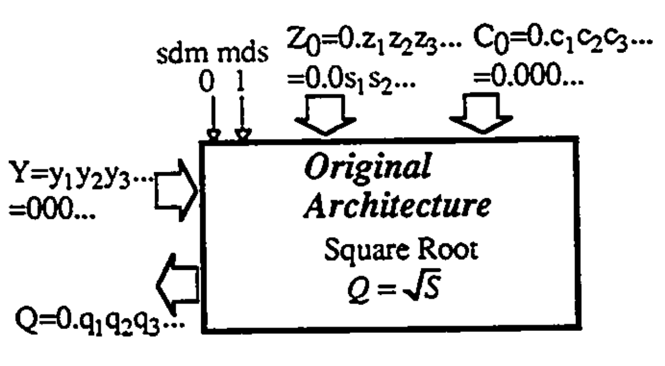


Figure 3.12: Square Root functionality

The extractor $C_j$ must be updated at each j th step as indicated in Table 2.4. Consider the j th row: the root digit $s_{j-1}$ from the preceding j-1th row is broadcast to all cells in the j th row and the extractor $-s_{j-1} \cdot S_{j-2} - s_{j-1}^2 \cdot 2^{-j}$ is computed. At this stage, the extractor $C_{j-1}$ will have been updated to store only the accumulating root $S_{j-2}$ in signed binary form. The S/DM control bit configures the type 3 cell enabling the term $-s_{j-1}^2 \cdot 2^{-j}$ to be computed. As with division, the new residual $2Z_j$ is determined by the parallel addition of the root extractor and the previous sum and transfer digit vectors. The S cells then select a valid root digit $q_j$ from estimates of the residual. The coefficient $C_j$ is updated to hold the partial root $S_{j-2} + q_{j-1} \cdot 2^{-j+1}$. This updating is performed by the multiplexers in the $1^*$ cells when S/DM is set to 0 and does not contribute to the critical path through the array. All cell outputs and control bits are latched and passed to the following row of the array.

## Multiply-Accumulate:

For the multiply-accumulate operation, the control bit M/DS and S/DM are set to 0 and 1 respectively. This enables the multiplier digit $y_j$ to be negated and broadcast to all cells in the j th row. The previous result digit, $m_{j-2}$, is also negated and passed to the type 2 cell. Three input operands are needed for the operation, namely X, Y and A. The functionality of this operation is shown in Figure 3.13.

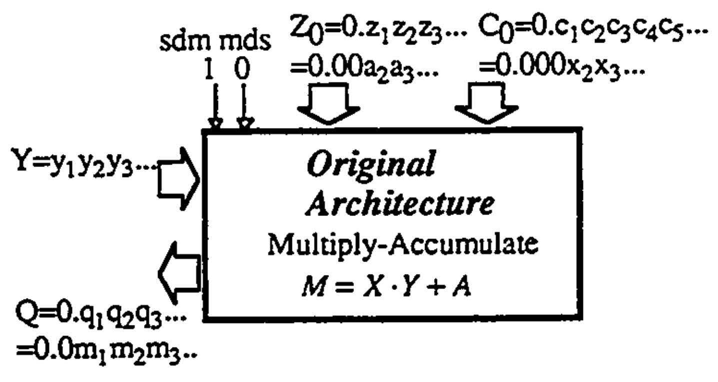


Figure 3.13: Multiply-Accumulate functionality

The quotient digit extractor $C_0$ is set to the scaled multiplicand $\frac{1}{4}X = 0.000x_2x_3...$ while the scaled residual $2Z_0$ is initialized to $\frac{1}{2}A = 0.00a_2a_3....$ Note that $sgn(c_1)=0$. Since the multiplicand is to pass through the array unchanged, the type **1*** and 3 cells must be configured as type **1** cells. This is achieved by setting S/DM to 1. The j th row computes the partial product terms $-y_j\frac{1}{4}X$. The new residual is then obtained by subtracting the previous result digit $m_{j-2}$ from the sum of the partial product terms and the previous sum and transfer digit vectors. The next result digit is determined by the S cells. As before, all cell outputs and control bits are latched and passed to the following row of the array.

The architecture has several desirable features which make it suitable for VLSI implementation including regularity, local interconnections and simple, local control. The critical path between latches has been estimated at 13 gate delays, and the simulation result is shown in the following chapter (Chapter 4). The gate count per row has been estimated at 32n+72 gate equivalents where n is the wordlength of input operand.

## 3.6 IMPROVED ARCHITECTURE

Based on the architecture purposed by McQuillan and McCanny [5]. An improved architecture is proposed in this section. The improved architecture can perform multiplication, division, and square-root, along with addition and subtraction. Despite of the extra operations, the critical path has been eliminated from 13 down to 10 gate delays. The array architecture on which the chip is based is the hardware description of the combined algorithm of the operations. Again, with the employment of a redundant number system and pipelining, the execution time for each operation is identical and independent of the wordlength. The architecture uses the redundant binary adder that is described in the previous chapter (chapter 2) as the basis foundation of the main array. The architecture requires only 3 control bits to select among the 5 different operations.

### 3.6.1 THE UNIFIED ALGORITHM

The algorithm, which is similar to the one described in the previous section (3.4.2), but with the capability to perform addition and subtraction, is described in this section. The algorithm combines the SRT division, square root and msd first binary multiplication described in the previous chapter (chapter 2). Also, by utilizing the feature of carry-free addition provided by the redundant arithmetic, addition and subtraction operations can be included in the unified algorithm. The corresponding unified algorithm equation (3.4), that is derived in Chapter 2 is stated here:

$$Z_j = 2Z_{j-1} - b_j \cdot C_j + n_j \tag{3.6}$$

where $Z_j$ is the residual and $C_j$ is the result digit extractor at jth step. The parameters $C_j$, $b_j$ and $n_j$ are summarized in the following Table 3.9:

| Operation | $C_j$ | $b_j$ | $n_j$ |
|---|---|---|---|
| Division | D | $q_{j-1}$ | 0 |
| Mult-accum | $\frac{1}{4}X$ | $-y_j$ | $-m_{j-2}$ |
| Square root | $Q_{j-2} + q_{j-1}2^{-j}$ | $q_{j-1}$ | 0 |
| Addition | 0 | 0 | $-a_{j-2}$ |
| Subtraction | 0 | 0 | $-a_{j-2}$ |

Table 3.9: Definition of recurrence parameters for Improved Architecture

The accuracy of an accumulating result is governed by $|Z - Z_i| < 2^{-i}$, in which subscript $i$ indicates the number of the digit and $Z_i$ is the partial product. The result is accumulated by appending successive result digits to the partial result without a carry propagation to previously determined result digits, that is $Z_i = Z_{i-1} + z_i \cdot 2^{-i}$. The initial values of the parameters are tabulated in Table 3.10.

| Operation | $2Z_0$ | $C_0$ | $n_0$ | $b_0$ |
|---|---|---|---|---|
| Division | N | D | 0 | 0 |
| Mult-accum | $\frac{1}{2}A$ | $\frac{1}{4}X$ | 0 | $-y_1$ |
| Square Root | $\frac{1}{2}S$ | 0 | 0 | 0 |
| Addition | $\frac{1}{4}A$ | $\frac{1}{4}B$ | 0 | 0 |
| Subtraction | $\frac{1}{4}A$ | $\frac{1}{4}B$ | 0 | 0 |

Table 3.10: Initial values of the parameters for Improved Architecture

For the unified algorithm, the ranges of the operands for each operation are as shown in Table 3.11. The ranges of input operands are similar to the ones shown in Table 3.2), except for the addition of input operands for addition and subtraction operations.

| Operation | Operands |
|---|---|
| Division<br>Q=N/D | $\frac{1}{4} \le \lvert N \rvert < \frac{1}{2} \quad \frac{1}{2} \le \lvert D \rvert < 1$ |
| Mult-accum<br>$M = X \cdot Y + A$ | $\frac{1}{4} \le \lvert X \rvert, \lvert A \rvert < \frac{1}{2} \quad \frac{1}{2} \le Y < 1$ |
| Square Root<br>$Q = \sqrt{S}$ | $\frac{1}{4} \le S < 1$ |
| Addition<br>Q=A+B | $0 \le \lvert A \rvert, \lvert B \rvert < \frac{1}{2}$ |
| Subtraction<br>Q=A-B | $0 \le \lvert A \rvert, \lvert B \rvert < \frac{1}{2}$ |

Table 3.11: Ranges of Operands for Improved Architecture

At each iteration, a simple digit result selection function is required for these five operations. The corresponding selection function can be expressed as:

$$\begin{matrix} q_{j-1} \\ m_{j-2} \\ a_{j-2} \end{matrix} = \begin{cases} \alpha & if \quad D \ge 0 \\ -\alpha & if \quad D < 0 \end{cases} \quad \text{where } \alpha = \begin{cases} +1 & if \quad \hat{Z} \ge \frac{1}{2} \\ 0 & if \quad -\frac{1}{4} \le \hat{Z} \le \frac{1}{4} \\ -1 & if \quad \hat{Z} \le -\frac{1}{2} \end{cases} \tag{3.7}$$

where $\hat{Z}$ is an estimate comprising the three MSDs of the residual $2Z_{j-1}$ and the result digits at jth step are espressed as $q_{j-1}$, $m_{j-2}$ and $a_{j-2}$. For division and square root operations, the msd is computed at the end of the first iteration whereas for the multiply-accumulate, addition and subtraction operations, the msd is not generated until the end of second iteration. The result digits have the following definition as tabulated in Table 3.12:

| Result word | Outputs | Actual Number Representation |
|---|---|---|
| Q | $q_1q_2q_3 \ldots q_j$ | $0.q_1q_2q_3 \ldots q_j$ |
| M | $0m_1m_2m_3 \ldots m_{j-1}$ | $0.m_1m_2m_3 \ldots m_{j-1}$ |
| A | $0a_1a_2a_3 \ldots a_{j-1}$ | $a_1.a_2a_3 \ldots a_{j-1}$ |

Table 3.12: Definition of result digits

The number representation of the residual and the result are signed binary numbers, assuming the values from the digit set $\{\bar{1},0,1\}$. Also, each SBNR digit is encoded according to the sign-magnitude scheme described in the preceding chapter (Chapter 2). Although the (+,-) scheme might yield a simple structure, periphery circuitry is needed to convert the conventional binary number to SBNR number. The advantage in using the sign-magnitude scheme is that conventional binary numbers can be entered directly from the magnitude digits of the architecture while the sign digits are connected to ground.

## 3.6.2 REDUNDANT ADDITION

The main array, which consists of multiply-add cells, is based on the modified redundant addition scheme described in Chapter 2. All operands and results from the array are SBNR numbers and have a digit set of $\{\bar{1},0,1\}$. Each digit is represented by two bits (2 binary lines), and is encoded according to the sign-magnitude scheme as described in chapter 2. The equations for the modified redundant addition algorithm derived in Chapter 2 are listed below:

$$d_i + b_i = s_i^* \quad \text{for the final block} \quad (2.6)$$
$$l_i^* + k_i^* + m_{i-1} = 2m_i + 2b_i + d_i \quad \text{for the combined block} \quad (2.7)$$
$$l_i^*, k_i^*, s_i^* \in \{\bar{1},0,1\},\ d_i, m_i \in \{0,1\},\ b_i \in \{\bar{1},0\}$$

In order to perform multiply-add functions, one of the operands has to be initialized as $k_i^* = b_i^* \cdot c_i^*$, in which $b_i^*$ and $c_i^*$ are both SBNR digits. As described previously, the parallel multiply-add function is performed in two stages. The first stage of the SBNR adder accepts operands $b_i^*$, $c_i^*$, $l_i^*$ and an interim transfer digit $m_{i-1}$ from the adjacent cell on the right, which is lower order significant, performs multiplication between $b_i^*$ and $c_i^*$, and addition in the form of equation (3.8):

$$l_i^* + (b_i^* \cdot c_i^*) + m_{i-1} = 2m_i + 2b_i + d_i \quad \text{for the combined block} \quad (3.8)$$
$$\text{where } b_i^*, c_i^* \in \{\bar{1},0,1\}$$

The interim transfer digit $b_i$ is passed to the second stage of the adjacent adder cell on its left or its higher order significant which computes the output sign $s_{i+1}^n$ and magnitude $s_{i+1}^p$ digits by adding the interim sum $d_{i+1}$ and the interim transfer from the adder cell on the right. The sign and magnitude output digits are computed in Table 3.13 as follows:

| Interim transfer $b_i$ | Interim sum $d_i$ | Output $(s_i^n, s_i^p)$ |
|---|---|---|
| 0 | 0 | (0,0) |
| 0 | 1 | (0,1) |
| $\bar{1}$ | 0 | (1,1) |
| $\bar{1}$ | 1 | (0,0) |

Table 3.13 Second stage of multiply-add cell

The values of $m_i$, $b_i$ and $d_i$ can be computed according to Table 2.3 in the previous chapter (Chapter 2). The structure of the multiply-add cell described equation 3.8 is shown in Figure 3.14.

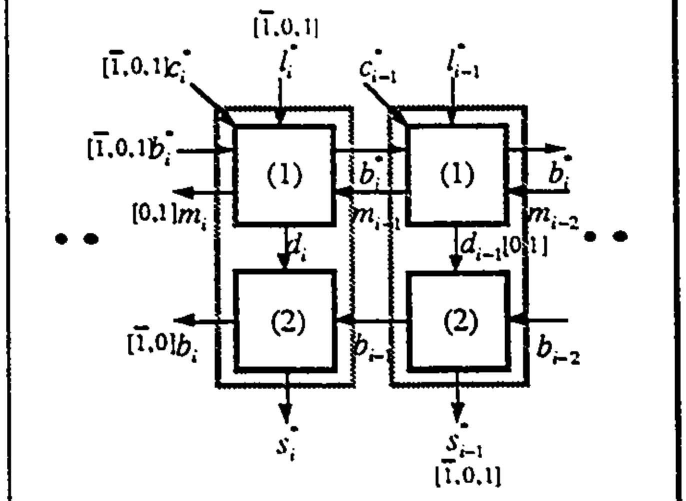


Figure 3.14: Structure of the multiply-add cell

## 3.6.3 ARCHITECTURE

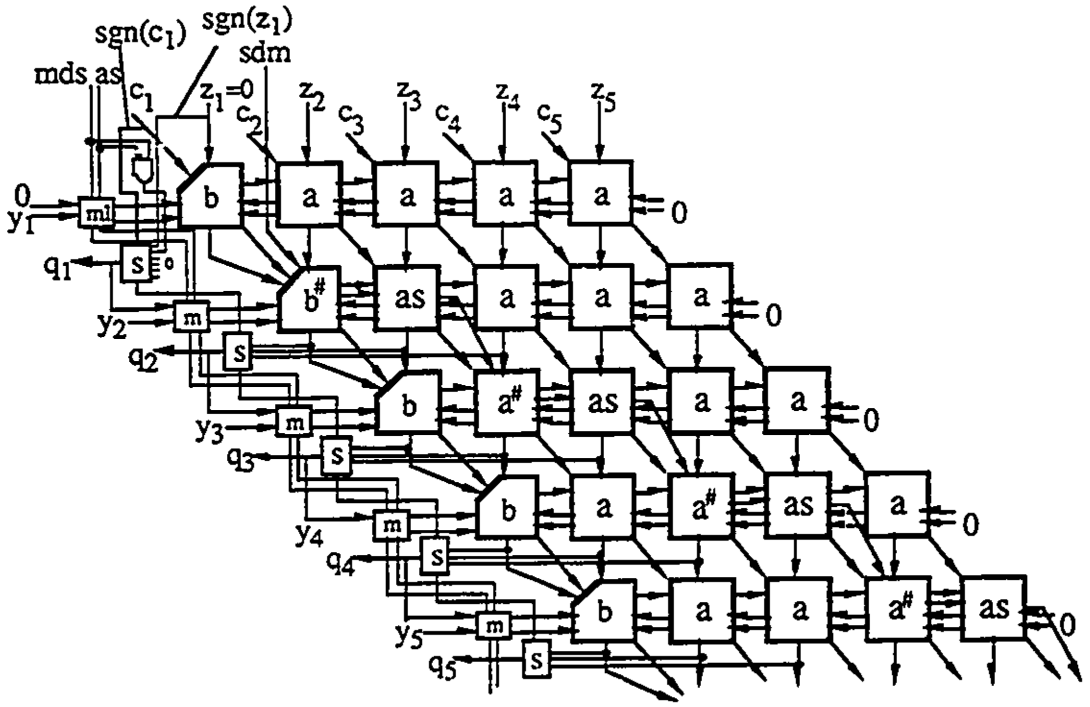


Figure 3.15: Modified VLSI Architecture for Multiplication, Division, Square Root Addition and Subtraction

The architecture to implement the unified algorithm is illustrated in Figure 3.15. The architecture, which is similar to the previous architecture, comprises a pipelined array of SBNR multiply-add cells (cell types a, a#, as), bounded at the left hand edge by type **b** cells. Cell types a# and **b**# have additional multiplexing circuitry required for the square root operation only. The default operation of these cells is the same as a and **b** cells respectively. The S cells on the periphery of the main array implement the result digit selection function, which is given in equation 3.7. Note that this architecture has the same performance as the previous one, both performing 5x5 bit operations.

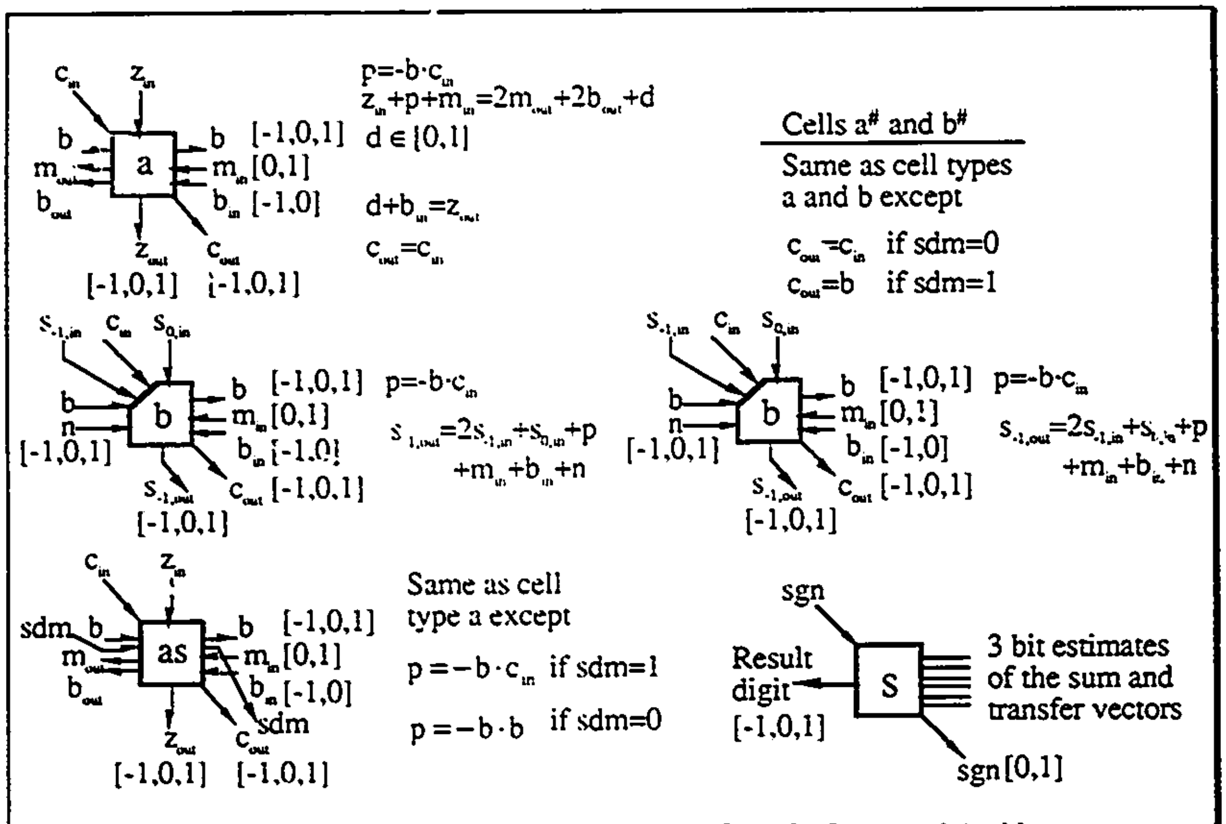

$c_{in}$ $z_{in}$
b [-1,0,1]
$m_{in}$ [0,1]
$b_{in}$ [-1,0]
$z_{out}$ $c_{out}$ [-1,0,1] [-1,0,1]
$p=-b\cdot c_{in}$
$z_{in}+p+m_{in}=2m_{out}+2b_{out}+d$
$d\in\{0,1\}$
$d+b_{in}=z_{out}$
$c_{out}=c_{in}$
Cells a# and b#
Same as cell types a and b except
$c_{out}=c_{in}$ if sdm=0
$c_{out}=b$ if sdm=1
$s_{-1,in}$ $c_{in}$ $s_{0,in}$
b n [-1,0,1]
$s_{-1,out}$ $c_{out}$ [-1,0,1] [-1,0,1]
$p=-b\cdot c_{in}$
$s_{-1,out}=2s_{-1,in}+s_{0,in}+p+m_{in}+b_{in}+n$
Same as cell type a except
$p=-b\cdot c_{in}$ if sdm=1
$p=-b\cdot b$ if sdm=0
sdm b
sgn
Result digit [-1,0,1]
3 bit estimates of the sum and transfer vectors
sgn [0,1]

Figure 3.16: Functional description of the Basic Cells for Improved Architecture

However, the total number of cell modules involved in the architecture is 35 compared to 40 from the previous architecture. The functional descriptions of the basic processing cells are given in Figure 3.16. Three control bits, denoted by sdm, mds and as are required to select between the five operations. The addition of an extra control bit does not contribute any delay to the critical path. The control bits of the improved architecture are summarized in Table 3.14.

| Operation | M/DS | S/DM | AS | sgn($C_0$) |
|---|---|---|---|---|
| Mult-Accum | 0 | 1 | 0 | 0 |
| Division | 1 | 1 | 0 | sgn(D) |
| Square Root | 1 | 0 | 0 | 0 |
| Addition | 0 | 1 | 1 | 0 |
| Subtraction | 1 | 1 | 1 | 0 |

Table 3.14: Control bits of the Improved Architecture

Each row of the array implements one iteration of the algorithm in equation 3.6, again the number of rows and columns determine the number of precision digits. For illustration, consider the jth row. The row accepts two SBNR input words from the preceding row, namely, the scaled residual $2Z_{j-1}$ and the extractor $C_{j-1}$. Each row also accepts two SBNR digits, $b_j$ and $n_j$ from the type **m** cell. According to control bits M/DS and AS, $b_j$ is broadcast to all cells in the row and digit $n_j$ is subtracted from the msd of the residual by boundary cell type **b**. Again, the redundancy overflow is presented in each iteration of addition and is compensated by the type **b** cell. But in contrast to the design of the type **2** cell, in the original architecture, the type **b** cell is a combination of the type **2** and type **1** cells from the msd side. Since in each iteration the residual is a fractional value, that is $|Z_j| < 1$, it is possible to limit the wordlength growth towards the integer position. This is illustrated in Figure 3.17.

The S cells on the periphery of the main array are used for result digit selection. Each S cell examines only three MSDs from the sign and magnitude vectors together with coefficient sign bit sgn($c_j$), that is only 7 binary lines in total (6 MSD binary bits plus one sign bit). Obviously the design of the S cell is in contrast to the design of the original architecture, where the estimate comprises the four MSDs of the residual. With the connections as shown in Figure 3.16, this ensures that for division and square root, $|q_0| = 1$ and for multiply-accumulate, addition and subtraction, $q_1=0$ as required. The **m1** cell and **m** cells on the boundary along the side of the S cells are used as multiplexers to broadcast multiplier digits into the array. With control bit AS set to 0, this configures the type **m1** cell as a type **m** cell. As control bit AS is set to 1, this enables the addend to be passed in the first row of the array and negated thereafter.

## Division:

To perform division operations, S/DM and M/DS are both set to 1 and AS is set to 0. This configures cell types $a^{\#}$ and as as type **a** cells enabling the divisor to be passed to each subsequent row of the array unchanged. The quotient digit extractor $C_0 = c_1c_2c_3...$ is set to the divisor $D = d_1d_2d_3...$ and the scaled residual $2Z_0 = z_1z_2z_3...$ is set to $N = 0n_2n_3...$ correspondingly; this is illustrated in Figure 3.18.

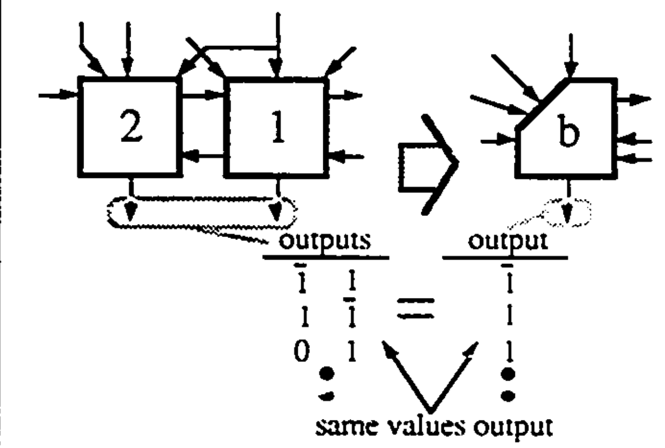


Figure 3.17 Difference between type 2 cell and type b cell

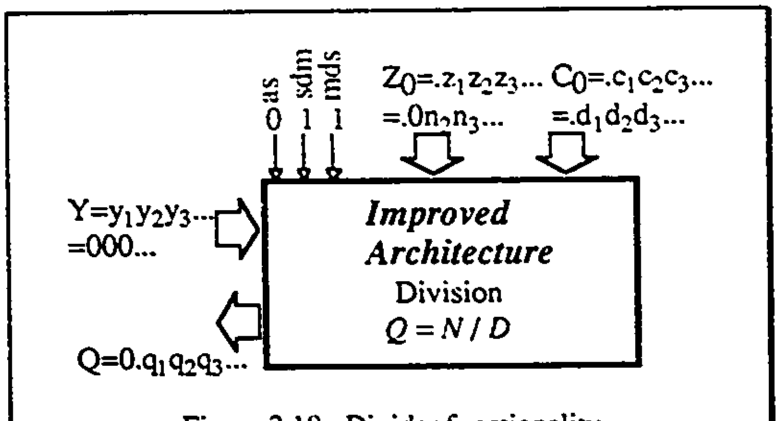


Figure 3.18: Divider functionality

The S cell control bit equals the sign of the divisor. Like the previous architecture, the control bits and the divisor are latched and passed through the array unchanged. The procedures are the same as described in the previous architecture.

## Square Root:

In order to perform the square root operation, S/DM must be set to 0, M/DS and AS are set to 1 and 0 respectively. The root digit extractor $C_0 = c_1c_2c_3...$ is initialized to 0 while the scaled residual $2Z_0=z_1z_2z_3...$ is set to $\frac{1}{2}R = .0r_1r_2...$, as illustrated in Figure 3.19.

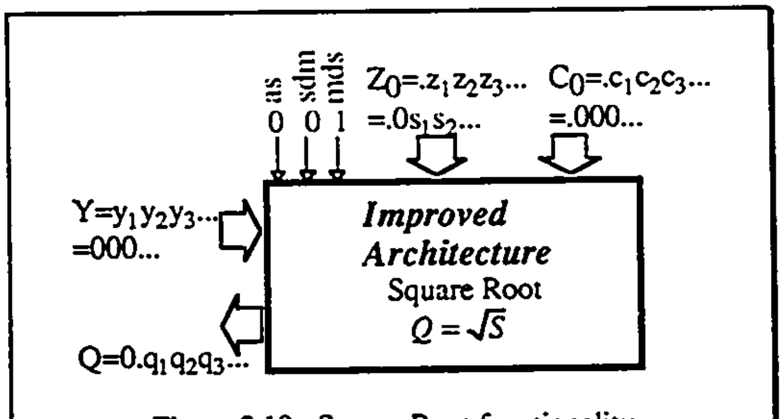


Figure 3.19: Square Root functionality

The procedures of the operation are the same as described in the previous architecture, and details will not be discussed here. The root digit is updated by the multiplexers in types 2#

cell and **1#** cells. All the control bits and cell outputs are latched and passed through the array.

## Multiply-Accumulate:

The M/DS and AS signals which control the type **m** and **ml** cells are both set to 0 thereby enabling the multiplier digit $y_j$ to broadcast to all cells in the jth row and the result digit, $m_{j-2}$, to be passed to the type **b** cell. With S/DM set to 1, this configures type **a#** and type **as** cells as type **a** cells enabling the multiplicand to pass through the array unchanged.

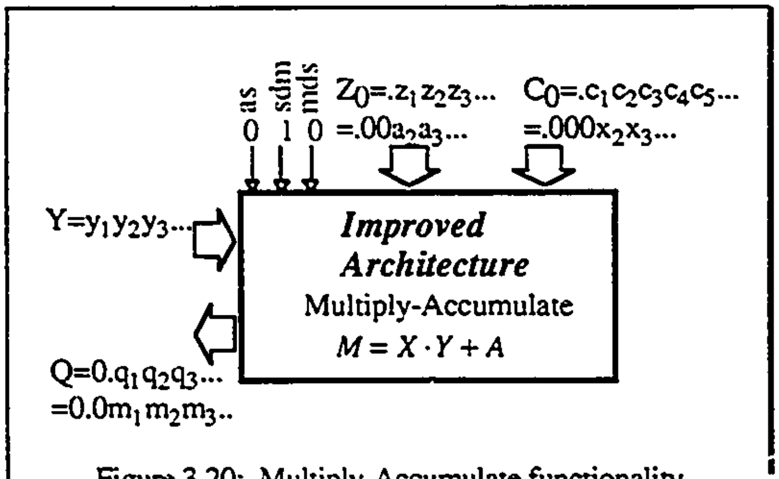


Figure 3.20: Multiply-Accumulate functionality

The quotient digit extractor $C_0 = c_1c_2c_3...$ is set to the scaled multiplicand $\frac{1}{4}X = .000x_2x_3...$ and the scaled residual $2Z_0 = z_1z_2z_3...$ is initialized to $\frac{1}{2}A = .00a_2a_3....$The functionality of multiply-accumulate is illustrated in Figure 3.20. All cell outputs are latched and passed through the array as specified previously.

## Addition/Subtraction:

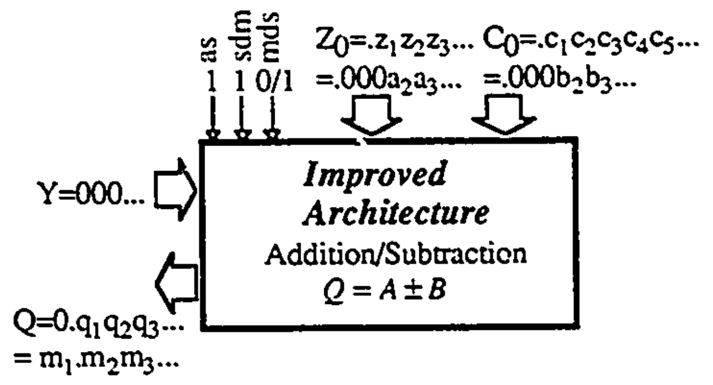


Figure 3.21: Addition/Subtraction functionality

To perform addition/subtraction operation, AS is set to 1 thereby enabling addend passes into the first row of the array and performing addition with the augend. $C_0 = c_1c_2c_3...$ is initialized to the scaled addend $\frac{1}{4}B = .000b_2b_3...$ and the scaled residual $2Z_0 = z_1z_2z_3...$ is set to $\frac{1}{4}A = .000a_2a_3...$; this is illustrated in Figure 3.21. Control bit M/DS is set to 0 or 1 regarding the operations of addition or subtraction. The result digit is determined by the S cells and each result digit $a_{j-2}$ is subtracted from the sum of product at each iteration.

This improved architecture employs many of the characteristics from the original architecture, including regularity, equal execution times for each operation, precision independent throughput and simple control. Futhermore, add and subtract operations are involved, and fewer cell modules are required to implement the improved architecture. Finally, the critical path for each pipelined row is estimated at 10 gate delays, and spice simulation results will be shown in the next chapter to verify the actual timing delay. The gate count per row for the improved architecture has been estimated at 21n+64 gate equivalents.

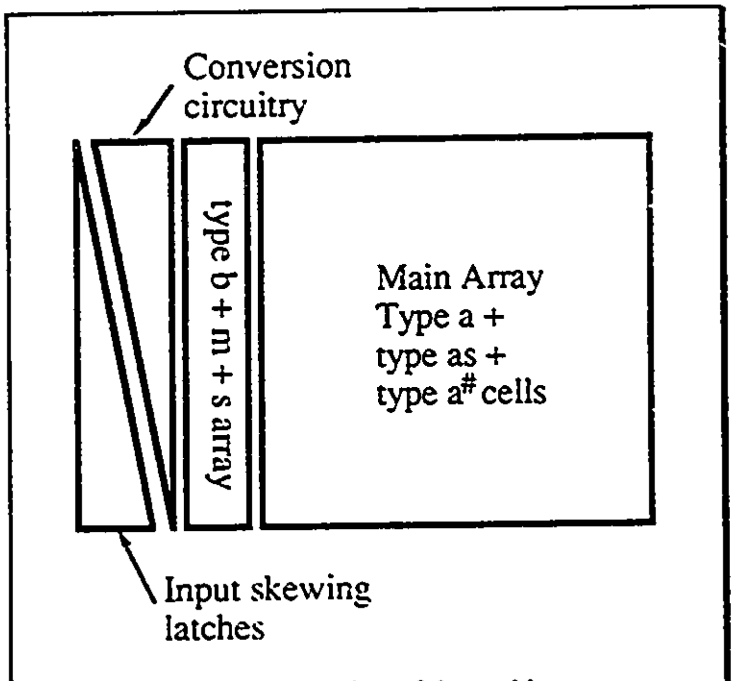


Figure 3.22 Floorplan of the architecture

As a result of pipelining, the array generates the outputs in a skewed-parallel manner. Moreover, the output data are in sign-magnitude format (sign bit + magnitude bit), therefore it is necessary to convert and deskew the data; this process can be achieved by using the conversion scheme described in Chapter 2. This scheme allows the data to be converted to a conventional binary form during the deskewing process. Before the input operands enter the architecture, in a bit parallel manner, the multiplier operands must be skewed. The multiplier skewing circuitry comprises a triangle array of latches and this triangular configuration allows the multiplier operand to enter the circuitry in a bit parallel manner but is accepted by the main array in a skewed-parallel manner. The overall floorplan of the arithmetic architecture with output deskewing and input skewing circuitry is illustrated in Figure 3.22.

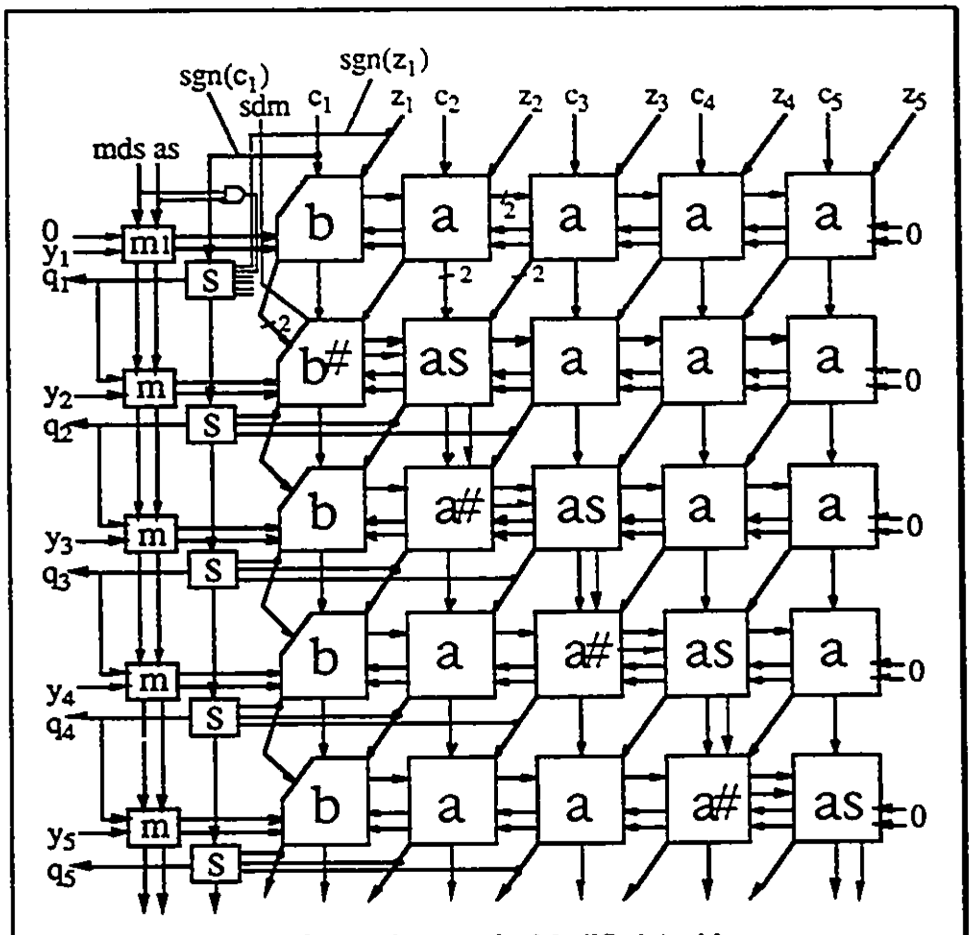


Figure 3.23 Architecture for Modified Architecture

The arrangement of cell modules shown in Figure 3.16 previously can be reconfigured as shown in Figure 3.23. The structure of the main array is highly regular and is very suitable for VLSI implementation.

## 3.7 COMPARISONS

In this chapter, an improved architecture to perform multiply-accumulate, division, square-root and addition/subtraction operations has been proposed. The major differences between the two architectures are summarized in Table 3.15.

| | Previous Architecture | Improved Architecture |
|---|---|---|
| Cell count/row (n bit operand) | n+1 | n |
| Critical path/row | 13 | 10 |
| Boundary cell | Overflow bound | Overflow + multiply-add |
| Result estimates ( S cell ) | 4 MSDs ( 10 bits ) | 3 MSDs ( 7 bits ) |
| SBNR digits | $\{\bar{2},\bar{1},0,1\}$ | $\{\bar{1},0,1\}$ |

Table 3.15 Comparisons between two architectures

It is clear that the improved architecture exhibits a better performance over the original architecture in terms of gate delay. Circuit comparisons and hspice simulations will be discussed in the next chapter.

## 3.8 DYNAMIC SWITCHING TREE

In the preceding sections, the original architecture and the improved architecture have been discussed. The two architectures utilize a small number of cell modules to perform different arithmetic operations. In digital design techniques, these cell modules can be either static or dynamic logic. Dynamic logic has a number of advantages over the static logic, including low power consumption, less area and performance, yet the drawback is timing constraints and charge sharing must be compensated in such design. Details on designing dynamic switching tree will not be discussed here, but it is of interest to compare the tree height between two architectures. Normally, a two input circuit component will have a tree height of two. As the inputs increase, the tree height grows correspondingly. Often, however, common output nodes can be merged together and the transistor can be replaced by a wire, thus causing the tree height to be decreased [29] [30].

Table 3.16 shows a comparison of the cell modules between the two architectures using the switching tree technique.

| | Number of Inputs | Maxiumum Tree Height |
|---|---|---|
| Type 2 | 10 | 6 |
| **Type b** | 12 | 5 |
| Type 1 | 7 | 5 |
| **Type a** | 8 | 5 |
| Type 3 | 8 | 5 |
| **Type as** | 9 | 5 |
| s cell | 10 | 8 |
| **select cell** | 7 | 5 |

Table 3.16: Comparisons of two architecture under switching tree technique
Note: Bold variables refer to the improved architecture

## 3.9 SUMMARY

This chapter briefly discussed the subject of bit-level systolic arrays and the configuration of the structure that is common in the hardware implementation of basic computer arithmetic operations. Algorithms and architectures have also been studied and researched in details. A comparison between a recently published architecture, for multiplication, division and square root, and a new architecture, proposed in this thesis, has shown that the improved architecture has a better performance in terms of latency and number of cell modules needed. Finally, the improved architecture is capable of being implemented using the switching tree technique with a maximum tree height of five.

# Chapter 4

# VERILOG MODELING AND VLSI IMPLEMENTATIONS

## 4.1 INTRODUCTION

In the previous chapter, an architecture to perform different arithmetic operations was studied, and based on the original architecture, an improved architecture to perform multiply-accumulate, division, square root and addition/subtraction was proposed. In designing such a complex system, a hierarchical design methodology is employed. In this chapter, Verilog [31], a hardware description language (HDL) and simulator will be used to build and simulate the fundamental blocks up to the system level of the architecture. We will also use both switch level and circuit level simulations to verify the functional correctness of the system and the timing delays of the architecture. With the proposed switching tree based cells, briefly discussed in the last chapter, SPICE simulations of those cells will be perfcrmed in the following sections.

## 4.2 DESIGNING WITH VERILOG

Verilog is an extremely rich language, with a variety of language features that supports the development of large scale designs. The features of the language governing design decomposition allow partitioning of a large design into several small modules, each of which is independently designed and described, and assembling of these modules together into a whole configuration. Details on the basic concepts will not be discussed here. For a more detailed discussion on the subject, the reader should refer to references [31] [32].

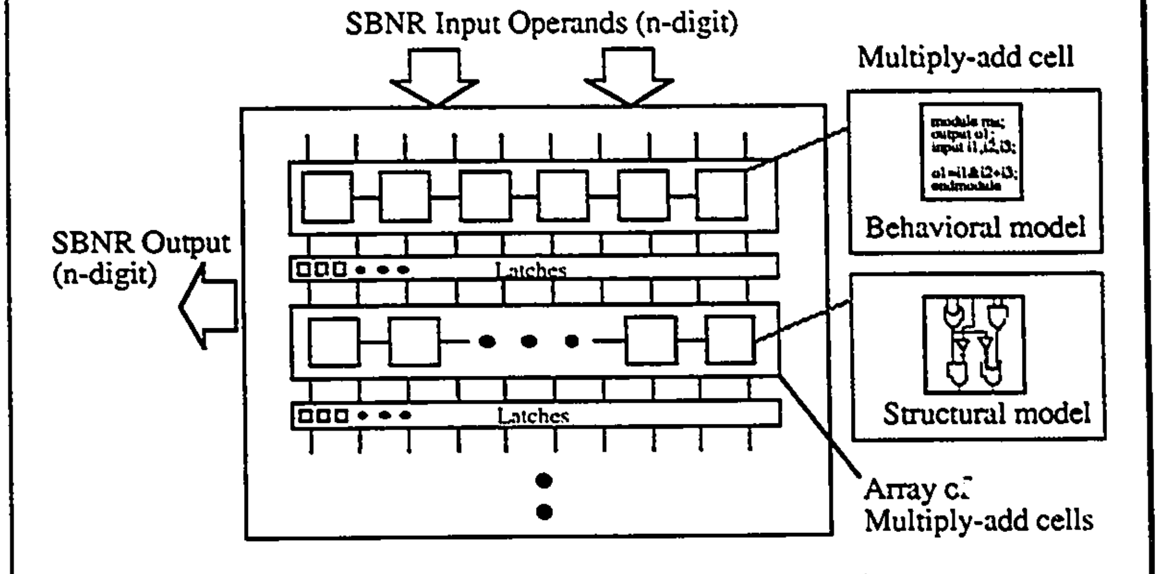
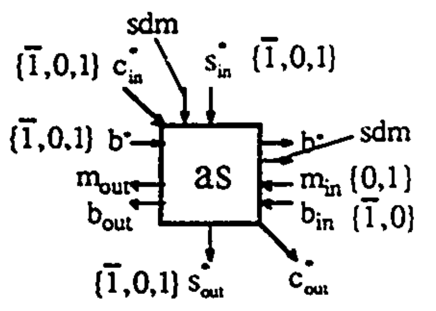


Figure 4.1: Hierarchical Structure of Improved Architecture

In order to design a system, it is essential to first decompose the system into a number of modules, in which each module can be either behavioral or structural. In general, behavioral module is described by functions of blocks and structural module is described by actual connections from nodes to nodes. For the improved architecture, it is an array architecture and composed of many pipelined rows, and each row comprises a number of cell modules. The hierarchical design structure is clearly illustrated in Figure 4.1.

## 4.2.1 BEHAVIORAL MODELING OF ARCHITECTURE

Considering the improved architecture, it is composed of six individual cell modules and each module has its own functionality. For the type as cell, as illustrated in Figure 4.2, the operations of the cell are described by its functional description: one control bit, three SBNR input numbers and two interim transfer input numbers are required to produce two interim transfer output numbers and two SBNR output numbers.


sdm
$\{\bar{1},0,1\}$ $c^{*}_{in}$
$s^{*}_{in}$ $\{\bar{1},0,1\}$
$\{\bar{1},0,1\}$ $b^{*}$
$m_{out}$
$b_{out}$
as
$b^{*}$
sdm
$m_{in}$ $\{0,1\}$
$b_{in}$ $\{\bar{1},0\}$
$\{\bar{1},0,1\}$ $s^{*}_{out}$
$c^{*}_{out}$

Functional description

$$p = b^{*} \cdot c^{*}_{out}$$
$$s^{*}_{in} + p + m_{in} = 2m_{out} + 2b_{out} + d$$
$$b_{in} + d = s^{*}_{out}$$
$$c^{*}_{out} = b^{*} \quad \text{if sdm=0}$$
$$c^{*}_{out} = c^{*}_{in} \quad \text{if sdm=1}$$

Figure 4.2: Type as cell and its functional description

Each SBNR number has a 2-bit binary representation, hence there is a total number of 11 input bits and 6 output bits for type as cell. From truth tables and Karnaugth maps, boolean expressions can be obtained. These boolean expressions can be used as module definitions for a behavioral model. The structure of the behavioral model will not be described here, but basically the behavioral model has input and output declarations, module interconnections and module definitions. Here, module definitions are expressed as boolean equations. Figure 4.3 shows the behavioral description for the type as cell; the behavioral descriptions for the other cell modules are shown in Appendix A.

```
module
cellas(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo);
input sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1;
output sno,spo,mi,bi,scno,scpo;
wire sno,spo,mi,bi,di,scno,scpo;

  assign mi= (sbn&~scn&scp) | (~sbn&sbp&scn) | (~sn&(~sbp|~scp));
  assign bi= ~( (mi_1&(sp^(sbp&scp))) | (~sn&sp&sbp&scp);
  assign di= (sbp&scp) ^ (mi_1 ^ sp);
  assign sno= ~di & bi_1;
  assign spo= di ^ bi_1;
  assign scno= scn&sdm | sbn&~sdm;
  assign scpo=scp&sdm | sbp&~sdm;

endmodule
```

Figure 4.3 Verilog behavioral description for type as cell

We now show simulation results that verify the functional correctness of the model. These simulations are not concerned with the timing or delays associated with the circuit components. The simulation of type as cell was performed under Verilog software, and it is shown in Figure 4.4. All the SBNR input and output numbers have the same binary representations, they are defined as (0,0)=0, (0,1)=1, (1,1)=$\bar{1}$ and (1,0)=X. The rest of the cell modules are simulated in a similar fashion.

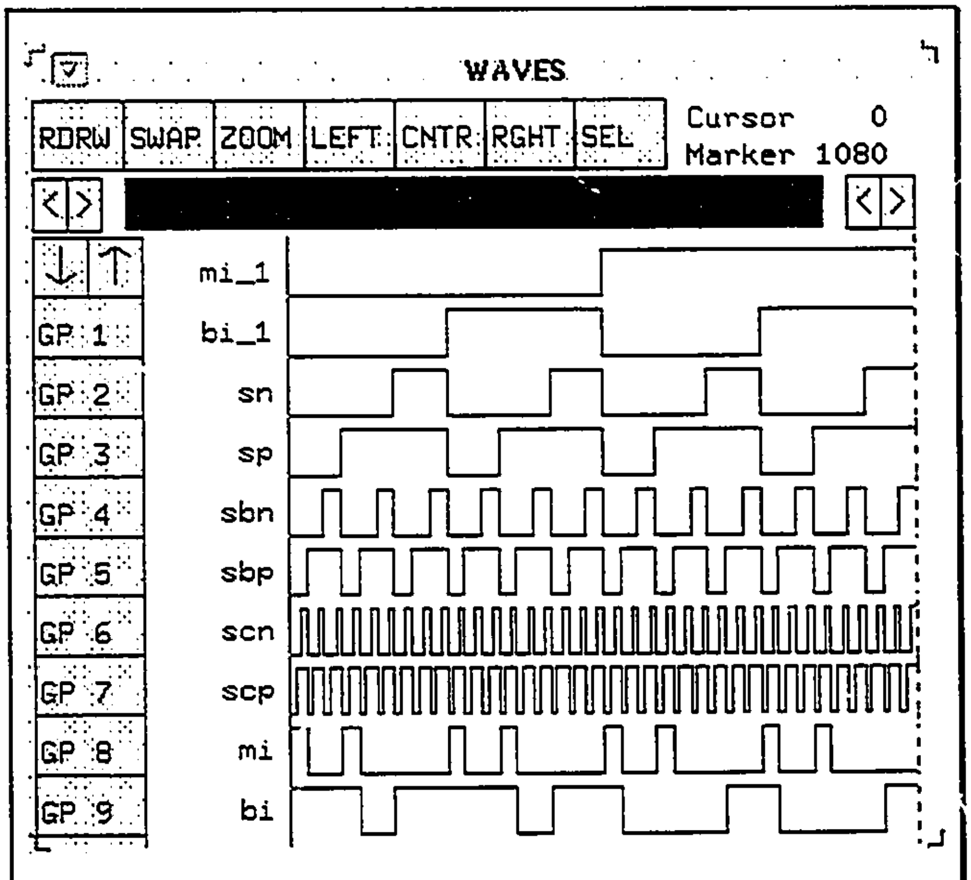


Figure 4.4: Graphic simulation of type as cell

With all the cell modules successfully simulated according to their functional requirements, the system is now constructed by incorporating all the cell modules in the structure shown in Figure 3.16 in Chapter 3. For example, an 8x8 bit architecture is partitioned into nine pipelined rows, and each pipelined row consists of eight individual cell modules.

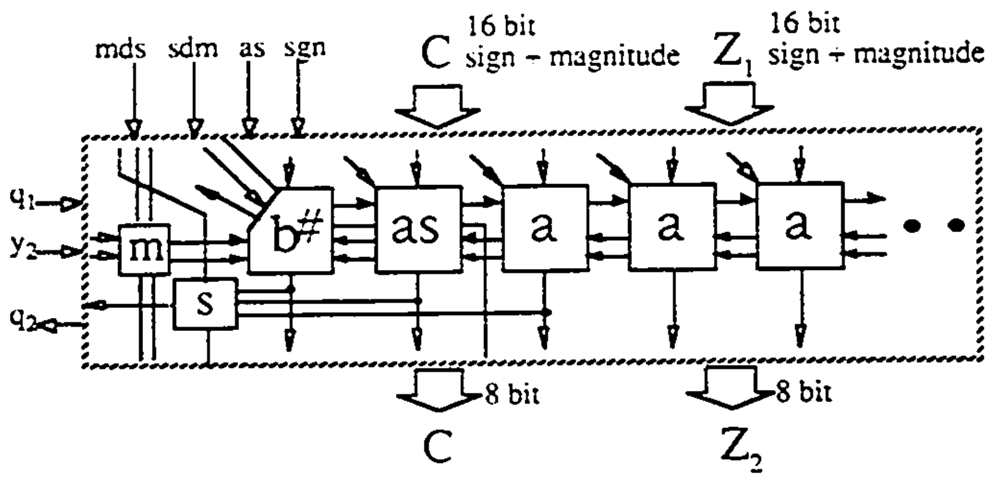


Figure 4.5: Structure configuration of row 2

Each pipelined row comprises one S-cell (result digit selection cell), one **m**-cell (multiplexer cell), one **b**-cell (boundary cell) and several numbers of type a cell and type as cell (multiply-add cell). Vectors of SBNR numbers are also involved in the inputs and outputs of the model. The structure configuration of row 2 from an 8x8 bit architecture is shown in Figure 4.5.

```
module
row2(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);
cella c5(1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
mi[5],bi[5],scno[4],scpo[4]).
c4(sn[5],sp[5],bn,bp,....
...
cellas c2(sdm,sn[3],sp[3],bn,bp,scn[2],scp[2],...);
cellam c1(sdm,sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
sno[0],spo[0],scno[0],scpo[0]):
mcell1 mc(mds,as,qn,qp,...):
select sel1(scn[0],sno[0],spo[0],...):
endmodule
```

Figure 4.6: Behavioral description of row 2

The behavioral model of each row evokes the cell modules required, and the interconnections between each cell module. The behavioral structure of row 2 is described in

Figure 4.6. The rest of the pipelined rows are described in a similar fashion and shown in Appendix A.

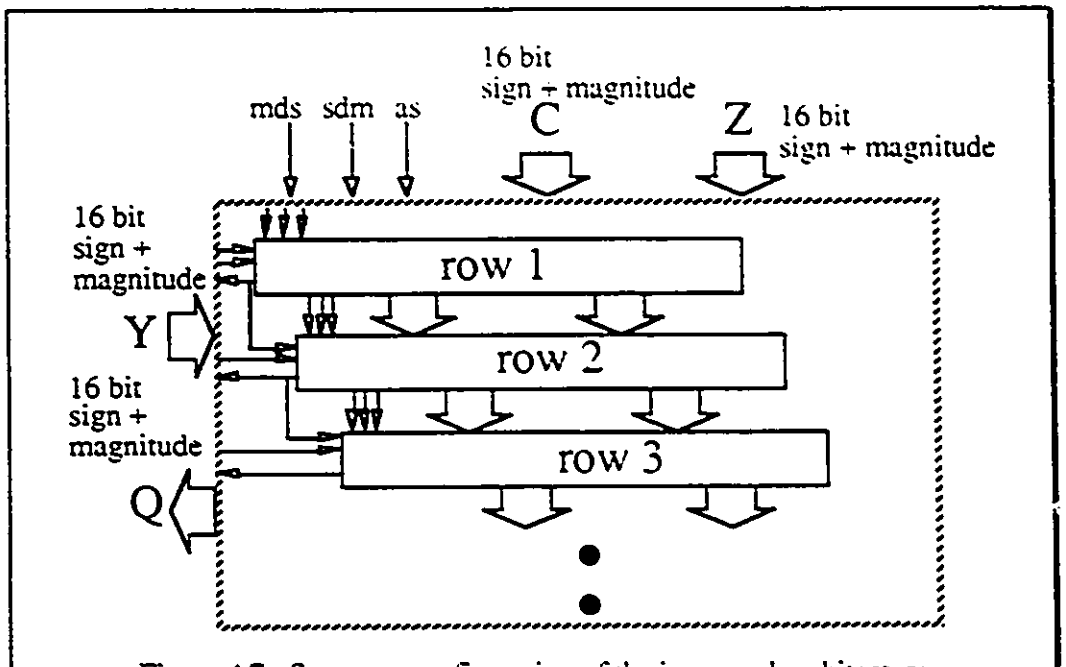


Figure 4.7: Structure configuration of the improved architecture

As previously discussed, Verilog software is used to perform the switch-level simulations, after each row is successfully simulated according to the configuration. The final step is to construct the array architecture by linking every row together according to the structure presented in Figure 3.16. The timing and delays associated with the entire system are not of concern. The purpose is to verify the functional correctness of the complete system. The structure configuration of the improved architecture is shown in Figure 4.7.

The final behavioral model of the improved architecture describes the inter-connections between each pipelined row; vectors of input and output SBNR numbers are also declared in the model. The behavioral description of the architecture is shown in Figure 4.8. For the entire description of the improved architecture, the reader should refer to Appendix A.

```
module allrow(sn,sp,scn,scp,mds,sdm,as,yn,yp,sno,spo,qno,qpo,clk);
input [0:5] sn,sp,scn,scp,yn,yp;
input mds,sdm,as,clk;
output [0:5] sno,spo;
output [0:6] qno,qpo;
wire [0:6] qno,qpo;
wire [0:5] sn,sp,.....;
row1 r1(sn,sp,scn,scp,mds,sdm,as,yn[0],yp[0],sno1,spo1,scno1,scpo1,
        qno[0],qpo[0]);
row2 r2(sno1,spo1,scno1,scpo1,mds,sdm,as,yn[1],yp[1],qno[0],qpo[0],
        sno2,spo2,scno2,scpo2,qno[1],qpo[1]);
row3 r3(sno2,spo2,scno2,...);
...
endmodule
```

Figure 4.8 Behavioral description of the improved architecture

For the 8x8 bit improved architecture, three input operands: C, $2Z_0$, and Y, and three control bits: M/DS, S/DM and AS are needed for five different operations. A random SBNR number generator written in 'C' is generated to produce the necessary SBNR input operands for the switch-level simulations.

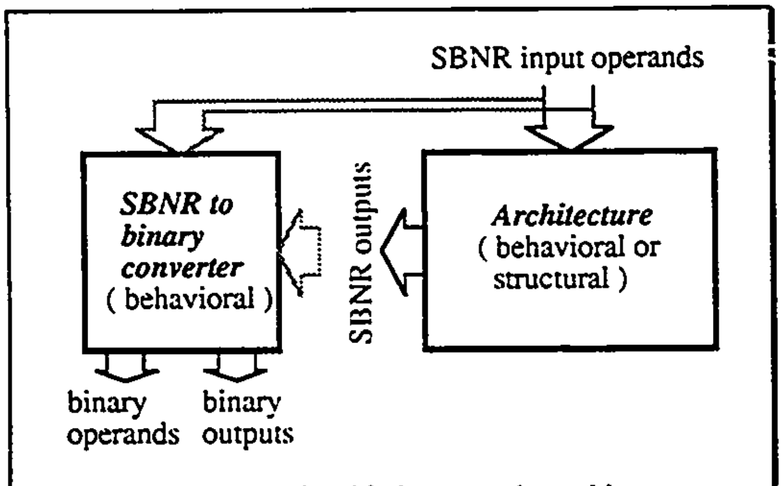

SBNR input operands
SBNR to binary converter ( behavioral )
SBNR outputs
Architecture ( behavioral or structural )
binary operands
binary outputs

Figure 4.9: Relationship between the architecture and the converter

Each SBNR operand is composed of sign and magnitude vectors, and hence in the verification process, many SBNR operands are needed for each operation. As a result, difficulty and confusion may prevent the designer from verifying the architecture. Therefore, as a convenience for the designer, a SBNR-binary converter is constructed utilizing the behavior model of verilog. The behavior model is written in a fashion similar to conventional imperitive programming languages, and it converts all the SBNR numbers to binary numbers. The relationship between the architecture and the converter is shown in Figure 4.9; the outputs from the converter are shown in Appendix B.

## 4.2.2 MIXED STRUCTURAL AND BEHAVIORAL

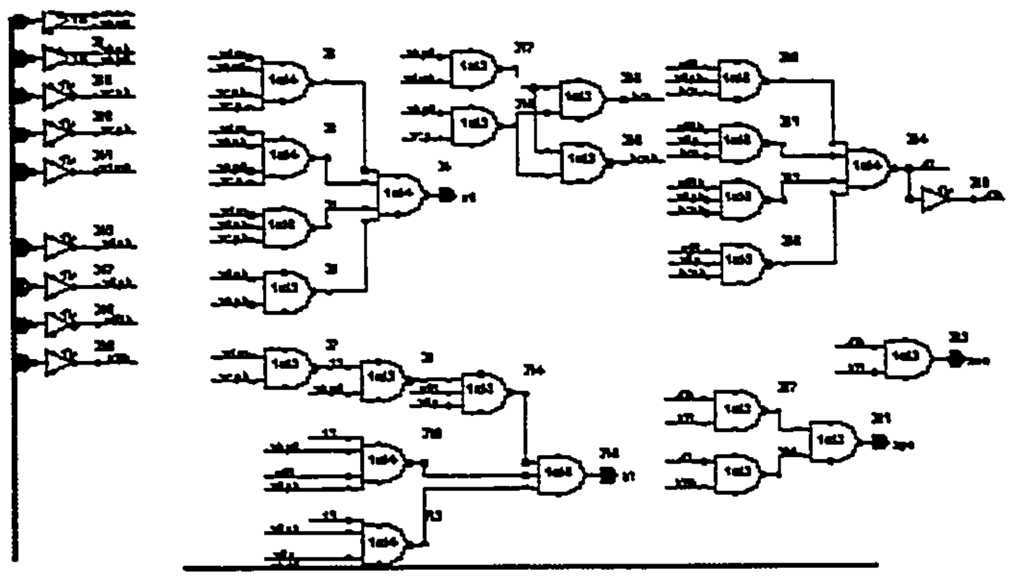


Figure 4.10: Schematic diagram of type as cell

In the previous section, every cell module in the architecture, to the highest level of hierarchy, has been verified and simulated under the behavioral model. Since six different types of cell modules are used as the basic elements of the construction, these cell modules can be implemented in terms of a logic gate network.

In this section, the structural model of the cell modules is constructed. All the cell modules use primitive gates as the building blocks. Figure 4.10 shows the schematic diagram of the type as cell. The cell has the functional description presented in Figure 4.3, and the input and output terminals are same as the ones described in the behavioral model of Figure 4.3. The other cell schematic diagrams are in Appendix C.

The logical description of the type as cell is associated with primitive gates, and a simulation must be performed to verify the functionality of the circuit component. One of the advantages of designing the system hierarchically is that the abstract of the fundamental cell module can be changed easily. The cell module as described in the previous behavioral level model, can now be replaced by a structural model. From the structural configuration of each pipelined row, the abstract of the cell module is just like a socket with the ability to bind a component, either structural or behavioral. The structure of the entire architecture is a mixture of structural and behavioral level descriptions; each cell module is implemented by primitive gates, and the interconnections between the cell modules and pipelined rows are described by the behavioral model.

## 4.3 STRUCTURAL ARCHITECTURE

The gate level structure of the architecture will be presented in this section. The architecture has been pipelined in which a number of latches are required for every row. In this design, a scan path approach to testing has been employed as it provides excellent observability and controllability at all points in the circuit. The pipeline latches are of the negative-edge-triggered scan latch and occupy approximately about 40% of the total area of the chip. To enable scan path testing, a global distributed testing signal is used to enable the latches to be connected together as a shift register. The test vectors can be clocked into the design via a test input pin and delivered to the inputs of any cell or row. The array can then be set into normal operation mode and a test performed. Resetting the array into the test mode enables the result of the test to be clocked out via a test output pin. The operations of scan path testing are shown in Figure 4.11a and Figure 4.11b respectively.

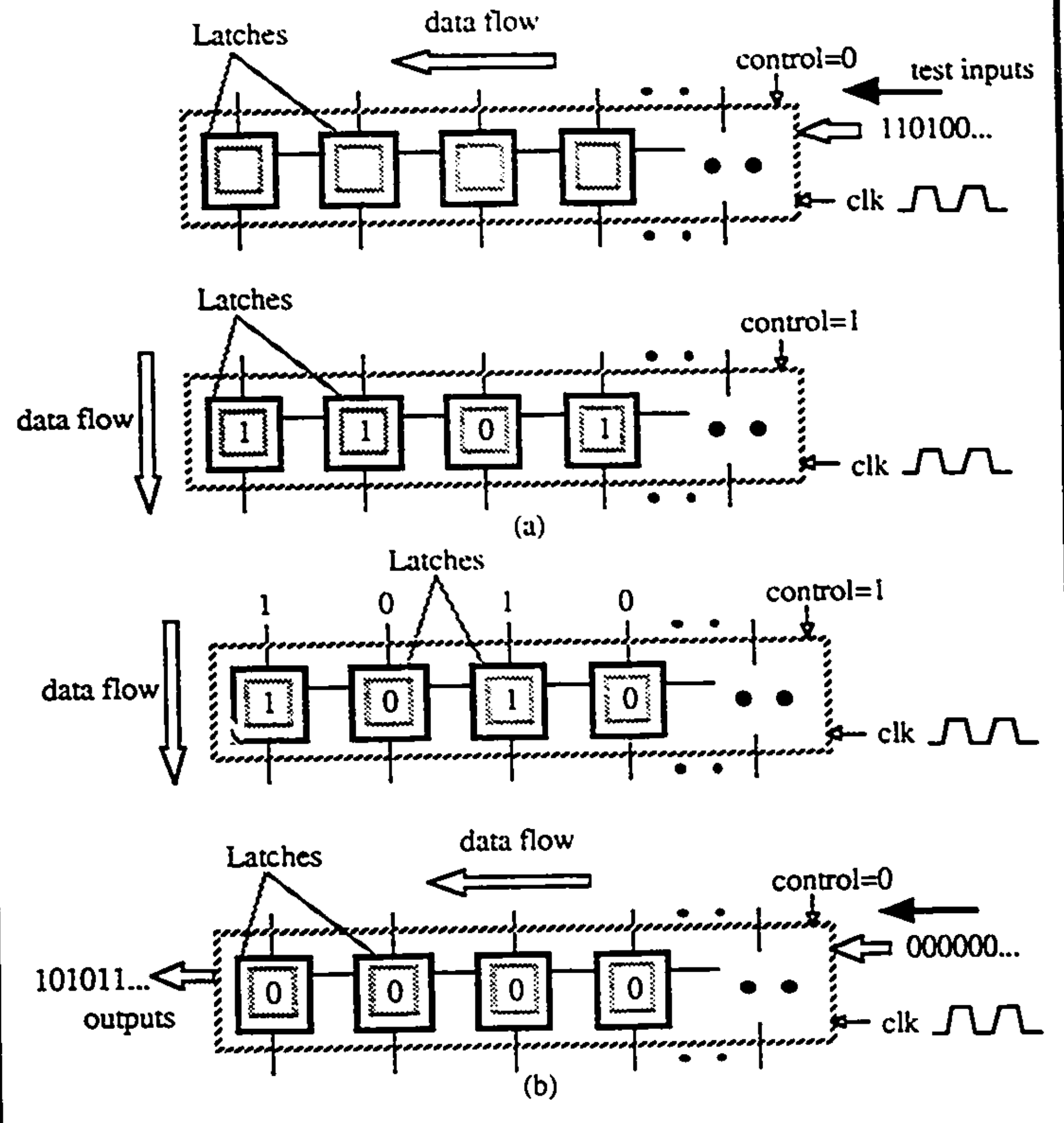


Figure 4.11: Scan path testing

The Verilog output waveform of the architecture is shown in Figure 4.12. Because of the pipelining, the output emerges in a skewed-parallel manner, and conversion circuitry is needed to convert the skewed SBNR outputs to conventional binary outputs. The circuitry comprises a pipelined triangular array of cells, the one described in Figure 2.13, and is appended to the output of the main array, which is the left hand edge of the array architecture. The method employed to convert the signed binary output of the main array to a conventional binary form is described in section 2.2.4. The conversion circuitry also deskews the output which is originally computed in a skewed-parallel form.

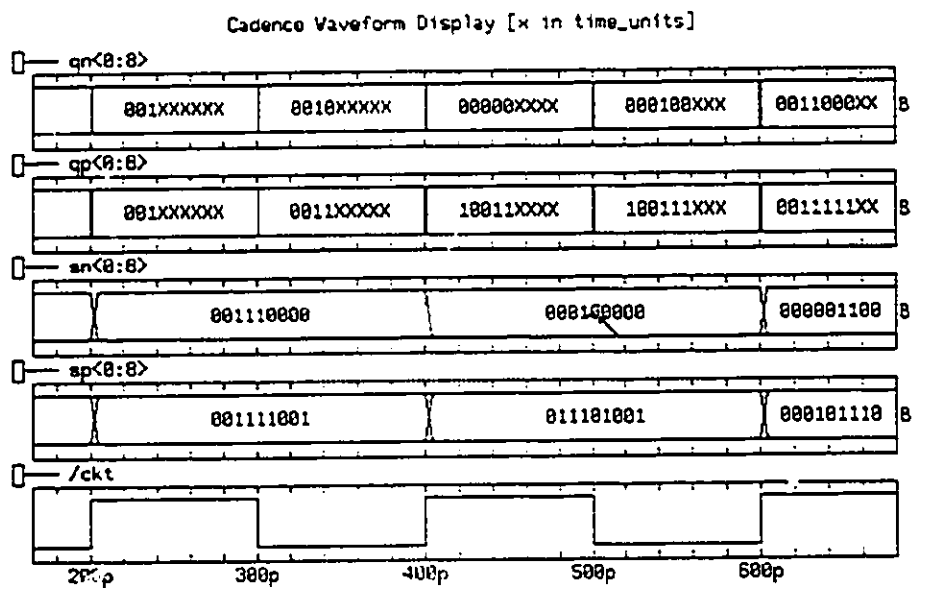


Figure 4.12: Switch level simulation of the improved architecture

For simplicity and convenience, the Automatic Placed and Rout method has been employed to layout the entire architecture. Custom layout is very time consuming and it is easy to make errors in interconnecting between cell modules. For a large design, it is very difficult and time-consuming to layout the architecture in a custom manner, even though a more compact design might result. The mask layout of the architecture, utilitzing the automatic place and rout method, is shown in Figure 4.13. The chip performs 8x8 bit arithmetic operations with equal execution time for each operation. The area occupied by the improved architecture is estimated at 3000x3000 $\mu m^2$, and it consists of 3,400 primitive gates.

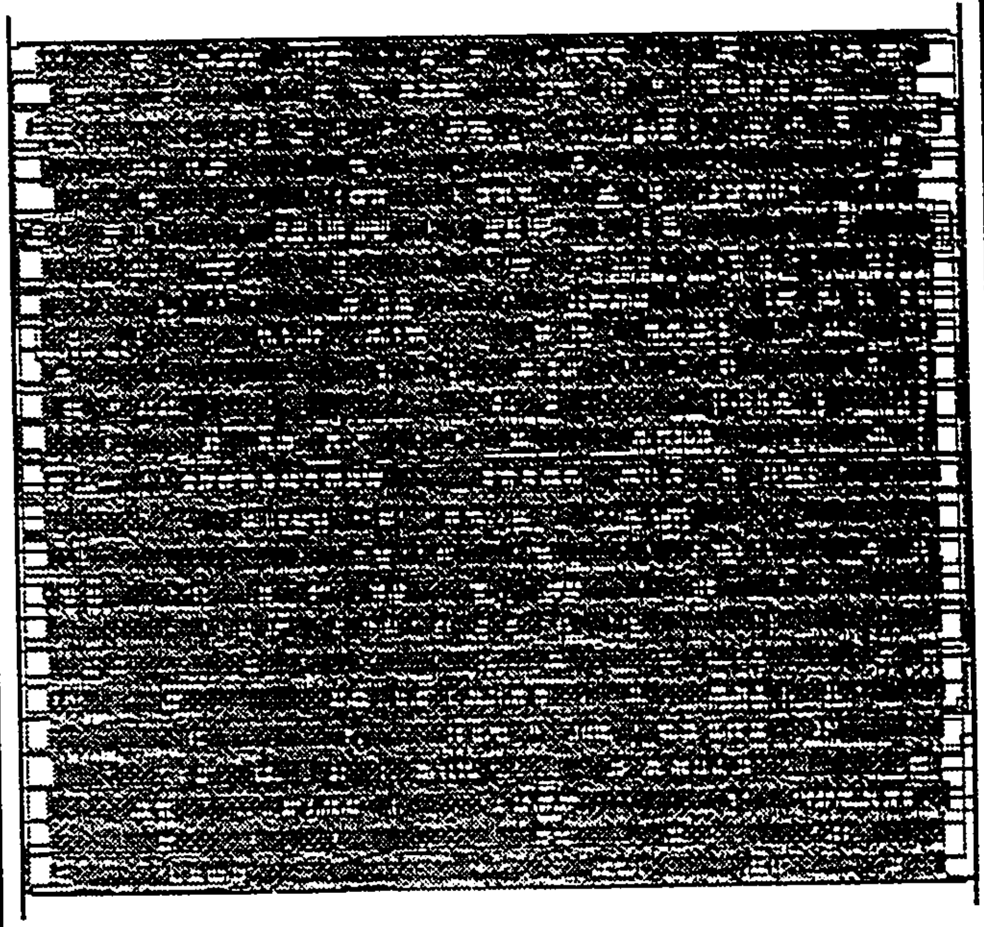


Figure 4.13: Mask Layout of the architecture

## 4.4 HSPICE SIMULATION

It is clear that the critical path of each row is governed by SBNR digit $q_j$ broadcast to the leftmost five cells of each row. The multiplixer circuitry for the square root operation in the main array does not contribute any timing delay to the critical path. In any case, the execution time for each operation is the same. The critical path delay under Hspice simulation is estimated at 10ns and it is clearly shown in Figure 4.14. The delay time is contributed by the data through the scan latch, type **m** cell, type **b** cell and select **s** cell.

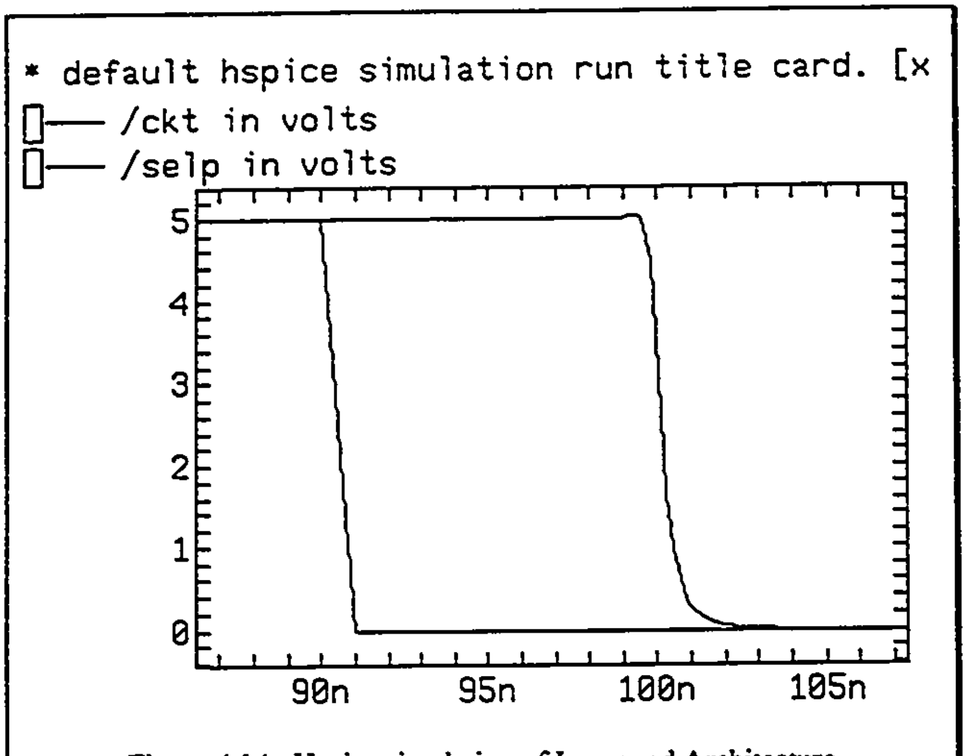


Figure 4.14: Hspice simulation of Improved Architecture

The Hspice [33] simulation results for the original architecture are not shown in this chapter, but the comparison of timing delays between two architecture are tabulated in Table 4.1.

| | Worst Case | Normal Case | Best Case |
|---|---|---|---|
| Original Architecture | 11.5 ns | 7.41 ns | 5.09 ns |
| Improved Architecture | 9.2 ns | 6.24 ns | 4.24 ns |

Table 4.1: Comparison of timing delays

From table 4.2 illustrated below, it is clear that the improved architecture has a number of advantages over the original architecture, for instance less standard gates are needed, less power dissipation occurs and better performance is given.

| | Original Architecture | Improved Architecture |
|---|---|---|
| Number of Standard Cells | 3871 | 3225 |
| Time delay/row ( Worst Case ) | 12.21 ns | 9.76 ns |
| Chip Area $\mu^2$ ( 8x8 bit ) | 3220 x 3150 | 2915 x 2706 |

Table 4.2: Comparison between two Architectures

As proposed in the last chapter, the architecture can be implemented by employing the dynamic switching tree technique. The schematic diagram of the magnitude bit of the type **b** cell is shown in Figure 4.15. The schematic diagrams of the rest cell modules constructed using the switching tree technique are shown in Appendix F.

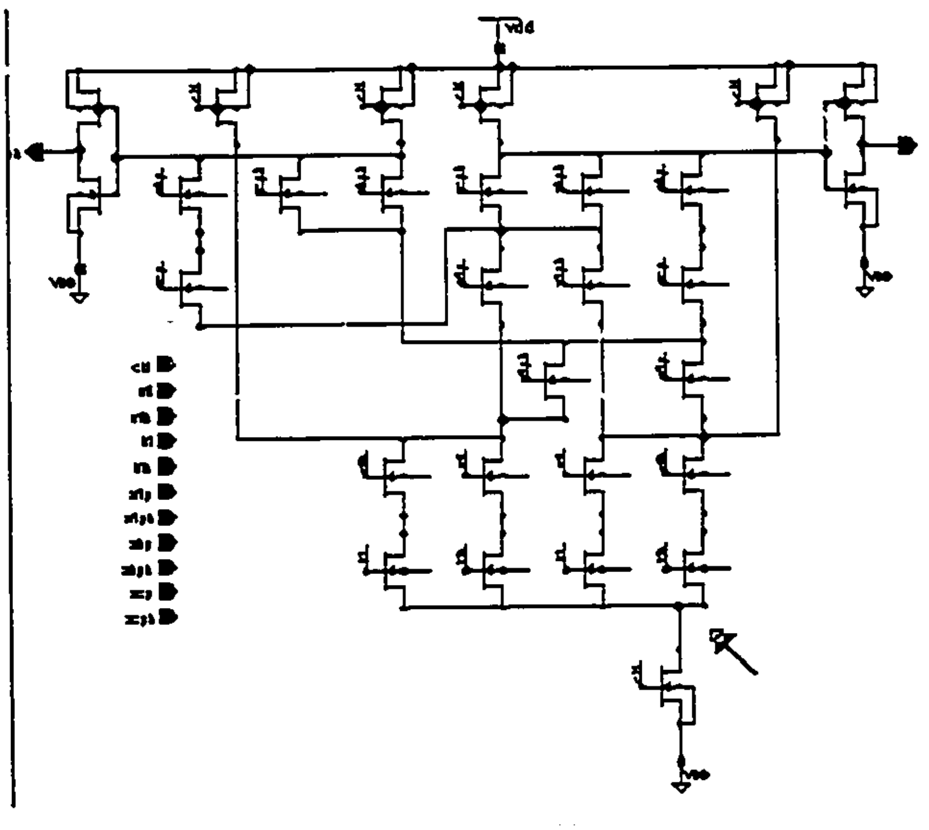


Figure 4.15: Switching tree diagram of magnitude bit of type b cell

Domino logic is chosen as the basic dynamic circuit structure. Domino logic has a number of advantages in terms of area, power dissipation and performance, yet the disadvantage is that it is not logically complete, so that the complementary function must be implemented separately. The timing delay of data through the multiply-add cells (type a) boundary cell (type b) and select cell (type s) is estimated at 5ns, and it is illustrated in Figure 4.16.

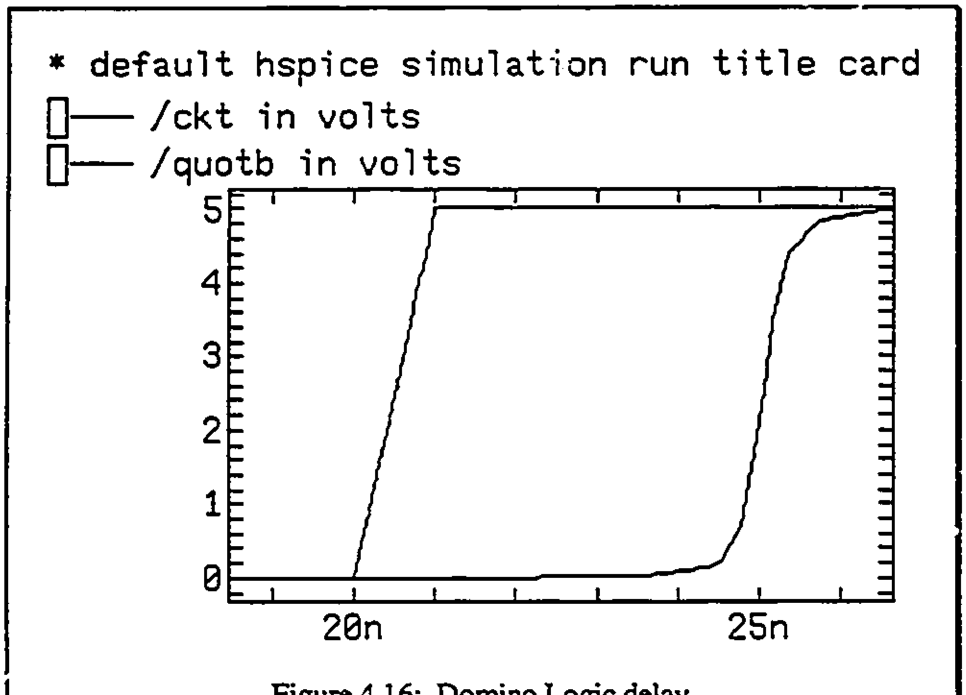


Figure 4.16: Domino Logic delay

As the value shows, it is a very attractive number. The height of the switching transistors of each block is at the acceptable range, so that it is very suitable for dynamic switching tree implementation, but charge sharing and timing constraints must be compensated in such a design. The charge sharing problem can be solved by charging some selective internal nodes by extra pre-charge transistor, but the selection of internal nodes has to be verified by simulation. Another solution is to ensure that the input signals are stable before entering the evaluation phase. This can be achieved by delaying the clock signal to the function block. By doing so, the internal nodes can be pre-charged in the pre-charging phase.

## 4.5 SUMMARY

The original and improved architectures were constructed and simulated using verilog software, which is a high-level hardware description language and switch-level simulator. Both architectures were constructed under a behavioral and structural model. Based on the simulation results under HSPICE software, it is indicated that the improved architecture is capable of reaching 100 Mhz throughput rates by utilizing 0.8 µm BiCMOS standard cells. By employing the switching tree technique, the improved architecture is capable of running at 200 Mhz throughput rates. The improved architecture is proven to utilize less cells, less chip area and is faster than the original architecture.

# Chapter 5

## CONCLUSIONS AND FUTURE DIRECTIONS

## 5.1 CONCLUSIONS

In this thesis, an improved VLSI architecture to perform combined multiply-accumulate, division, square-root and addition/subtraction operations has been implemented utilizing 0.8μm BiCMOS technology. The original architecture shows a combination of fine grain pipelining and a judicious use of redundant arithmetic results in the throughput rate of the architecture being independent of the operand wordlength. The circuit exhibits regularity and local communications, and can be easily extended to any wordlength due to its modular nature. Additionally, the execution time for each operation is the same, and the circuit requires only minimal control and can be reconfigured on every cycle. By retaining most of these characteristics from the original architecture, the improved architecture has a better performance in terms of latency and number of cell modules needed.

Signed Binary Number Representation (SBNR) and its usage to improve the performance of the arithmetic operations has been studied in detail. It has been shown that with the introduction of redundancy into the number representation, the time required for basic arithmetic operations can be accelerated. Additionally, redundancy can also provide structural flexiblity to the system. Redundant binary addition, in which all digit sets are

$\{\bar{1},0,1\}$ execute independently of the operand wordlength, in other words, parallel addition is achieved. Three algorithms that employ a redundant binary system have been studied, they are multiplication, division and square root. With the elimination of carry propagate during addition of two numbers, intermediate results can be obtained in a constant time independent of the wordlength. Thus the time required for the computation in each operation can be reduced and made independent of the operand wordlength.

Algorithms and architectures which perform combined arithmetic operations have also been studied and researched. The unified algorithm is a combination of the SRT division and square root schemes and msd first multiply-accumulate algorithm studied in this thesis. The algorithm is characterised by most-significant digit (MSD) first computation and equal execution time for each operation. The combination of three operations is the result of employing the concept of redundancy into the arithmetic system. The array architecture by McCanny and McQuillan has been carefully studied. The architecture operates on sign-magnitude operands and has a high throughput rate through a given level of pipelining. The improved architecture which employs many of the characterisitcs from McCanny and McQuillan's architecture has been presented. The architecture is capable of performing the three arithmetic operations offered by the original architecture with the addition of add/subtract operations. Furthermore, the improved architecture is capable of being implementing using the dynamic switching tree technique with a low tree height.

The original and improved architectures have been constructed and simulated by using Verilog, a high-level description language and simulator. Verilog provides the designer a technique to design the system hierarchically, by using different levels of abstract description. Thus the designer can determine the correctness of the system before actual VLSI implementation. Both architecutres have been simulated using Hspice, and the simulation result show that the improved architecture has an improved performance over the original architecture. Additionally, the simulation results of the imporved architecture have shown that a throughput rate of up to 100 Mhz can be achieved by using 0.8μm BiCMOS and the BNR standard cell library. Moreover, by employing switching tree techniques, speeds up to 200 Mhz can be reached with a tree height of five.

## 5.2 FUTURE DIRECTIONS

The improved architecture described in this thesis is for fixed point operations but can be easily extended to implement a floating point system. The IEEE standard 754 [34] can be used as the floating point standard since it is the widely acceptable floating point standard nowadays. In two references [35] [36], signed binary number representation (SBNR) is introduced into the floating point standard, therefore exponent handling, rounding, normalisation etc. are less complex than that in conventional binary units. However, a more detailed study should be undertaken in the area of applying the floating point standard to redundant binary implementations. The ultimate goal is to produce fast, area and power efficient designs for future technologies.

## REFERENCES

[1] A.K. Kamal, H. Singh, D.P. Agrawal, "A Generalized Pipeline Array", IEEE Trans. on Compt., Vol. C-23, pp. 533-536, 1974.

[2] D.P. Agrawal, "High Speed Arithmetic Arrays", IEEE Trans. on Compt., Vol. C-28, no. 3, 1979, pp. 215-224.

[3] J.H.P. Zurawski, J.B. Gosling, "Design of a High-speed Square Root, Multiply and divide Unit", IEEE Trans. on Compt., Vol. C-36, Jan 1987, pp. 13-23.

[4] M.D. Ercegovac, T. Lang, "Implementation of Module combining Multiplication, Division and Square Root", Proc. IEEE Intl. Symp. On Ccts and Systems., pp. 150-153, 1989.

[5] S.E. McQuillan, J.V. McCanny, "A VLSI Architecture For Multiplication, Division and Square Root", IEEE Intl. Conf. Acoustics, Speech and Signal Processing, May 14-17, pp. 1205-1208, 1991.

[6] A. Avizienis, "Signed Number Representations for Fast Parallel Arithmetic", IRE Trans. On Compt., Vol. EC-10, pp. 389-400, 1961.

[7] A. Avizienis, "Binary-Compatible Signed-Digit Arithmetic", Proceedings of Fall Joint Computer Conference, pp. 663-672, 1964.

[8] J. Sklansky, "Conditional-sum addition logic," IRE Trans. Electron. Computer", Vol. EC-9, pp. 226-231, June 1960.

[9] O.L. MacSorley, "High speed arithmetic in binary computers", Proc. IRE, Vol. 49, pp. 67-91, Jan. 1961.

[10] C.Y. Chow, J.E. Robertson, "Logical Design of A Redundant Binary Adder", IEEE Trans. on Computer, Vol. C-27, pp. 109-115, 1978.

[11] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa and N. Takagi, "A High-Speed Multiplier Using a Redundant Binary Adder Tree," IEEE Journal of Solid-State Circuits, Vol. SC-22, No. 1, February 1987.

[12] S.E. McQuillan, "Algorithms and Architectures for High Performance Arithmetic Processors", Faculty of Engineering, Queen's University of Belfast, September, 1992.

[13] M.D. Ercegovac and T. Lang, "Fast Multiplication without Carry-propagate Addition", Computer Science Department UCLA, Report CSD-870047, September 1987.

[14] M. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional number representations", IEEE Trans.on Computer, C-36, , pp. 895-897, 1987.

[15] S.C. Knowles, J.G. McWhirter, "An improved bit-level Systolic Architecture for IIR Filtering", Systolic Array Processors, eds. J.V. McCanny, J.G. McWhirter, E. Swartzlander, Prentice Hall, pp. 205-214, 1989.

[16] J. E. Robertson, "A New Class of Digital Division Methods", IRE trans. Electronic Computers, Vol. EC-7, pp. 218-222, September 1958.

[17] T. D. Tochner, "Techniques of Multiplication and Division for Automatic Binary Computers", Quarter J. Mech App. Math., Vol. 2, pt. 3, pp. 364-384, 1958.

[18] T. E. Williams, M. Horowitz, "SRT Division Diagrams and Their Usage in Designing Custom Integrated Circuits for Division," technical report CSL-TRL-87-326, Stanford University, November 1986.

[19] G. Metze, "minimal square rooting," IEEE Trans. Electron. Computer, vol. EC-14, pp. 181-185, April 1965.

[20] V. G. Oklobdzija and M. D. Ercegovac, "An on-line square root algorithm," IEEE Trans. Computer, vol. C-31, pp. 70-75, Jan. 1982.

[21] S. Majerski, "Square-Rooting Algorithms for High-Speed Digital Circuits", IEEE Trans. on Computers, Vol. C-34, No. 8, August, 1985.

[22] T.G. Noll, D.S. Landsiedel, H. Klar, G. Enders, "A Pipelined 330-MHz Multiplier", IEEE Journal of Solid-State Circuits, Vol. SC-21, No. 3, June, 1986.

[23] C.S. Wallace, "A Suggestion for a Fast Multiplier", IEEE trans. on Electronic Computer, Vol. EC-13, pp. 14-17, 1964.

[24] H.T. Kung, C. Leiserson, "Algorithms for VLSI Processor Arrays", Introduction to VLSI Systems, eds. C. Mead and L. Conway, Addison-Wesley, 1980.

[25] J.V. McCanny, J.G. Mcwhirter, "On the implementation of signal processing functions using one bit systolic arrays", Elect. Letter, Vol. 18, No. 6, pp. 241-243, 1982.

[26] J.V. McCanny, J.G. McWhirter, K.W. Wood, "An Optimized Bit Level Systolic Array for Convolution", IEE Proc., Vol. 131, Pt. F, No. 6, pp. 632-637, 1984.

[27] J.G. McWhirter, J.V. McCanny, "Systolic and Wavefront Arrays", VLSI Technology and Design, eds J.V. McCanny and J.C. Whiter, Academic Press, pp. 253-299, 1987.

[28] M.D. Ercegovac and T. Lang, "Implementation of module combining multiplication, division and square root," Proceedings of the International Symposium on circuits and Systems, pp. 150-153, 1989.

[29] R. Grondin, "WoodChuck, A Switching Tree Compiler Software", Faculty of Graduate Studies, University of Windsor, 1990.

[30] Zhang, personal communications, (July 1991).

[31] Cadence Design Systems, Inc., Verilog-XL Reference Manual Ver 1.6, March 1991.

[32] D.E. Thomas, P.R. Moorby. The Verilog Hardware Description Language, Kluwer Academic Publishers, 1991.

[33] Meta-Software, Inc., HSPICE User's Manual H9001, 1990.

[34] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE std 754, New York, The Institute of Electrical and Electronics Engineers, Inc., August, 1985.

[35] N. Quach, N. Takagi, M.J. Flynn, "On Fast IEEE Rounding", CSL-TR-91-459, Stanford University, Jan, 1991.

[36] N. Quach, M.J. Flynn, "Leading One Detection - Implementation, Generalization and Application", CSL-TR-91-463, March, 1991.

[37] D.E. Atkins, "Higher Radix Divison Using Estimates of the Divisor and Partial Remainders", IEEE Trans. on Computers, Vol. C-17, No. 10, October 1968.

# Appendix A

## BEHAVIORAL MODEL OF IMPROVED ARCHITECTURE

```
-------------------------- cell 1 ---------------------------------------
module cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo);

input sn,sp,sbn,sbp,scn,scp,mi_1,bi_1;
output sno,spo,mi,bi,scno,scpo;
wire sno,spo,mi,bi,di,scno,scpo;

 assign mi= (sbn&~scn&scp) | (~sbn&sbp&scn) | (~sn&(~sbp|~scp));

 assign bi= ~( (mi_1&(sp^(sbp&scp))) | (~sn&sp&sbp&scp) );

 assign di= (sbp&scp) ^ (mi_1^sp);

 assign sno= ~di & bi_1;
 assign spo= di ^ bi_1;

 assign scno=scn;
 assign scpo=scp;

endmodule

-------------------------- cell 1a ---------------------------------------
module cell1a(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo);

input sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1;
output sno,spo,mi,bi,scno,scpo;
wire sno,spo,mi,bi,di,scno,scpo;

 assign mi= (sbn&~scn&scp) | (~sbn&sbp&scn) | (~sn&(~sbp|~scp));

 assign bi= ~( (mi_1&(sp^(sbp&scp))) | (~sn&sp&sbp&scp) );

 assign di= (sbp&scp) ^ (mi_1^sp);

 assign sno= ~di & bi_1;
 assign spo= di ^ bi_1;

 assign scno=scn&sdm | sbn&~sdm;
 assign scpo=scp&sdm | sbp&~sdm;

endmodule

-------------------------- cell 3 ---------------------------------------
module cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo);

input sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1;
output sno,spo,mi,bi,scno,scpo;
wire sno,spo,mi,bi,di,scno,scpo;

 assign mi= sdm&((sbn&~scn&scp) | (~sbn&sbp&scn) | (~sn&~scp)) |
          (~sn&~sbp);
 assign bi= ~( (mi_1&sp&( (sdm&~scp) | ~sbp)) |
              (mi_1&~sp&sbp&(~sdm | scp)) |
              (~sn&sp&sbp&(scp | ~sdm)) );

 assign di= (sbp&(~sdm | scp)) ^ (mi_1^sp);

 assign sno= ~di & bi_1;
 assign spo= di ^ bi_1;

 assign scno=scn;
 assign scpo=scp;

endmodule

------------------------ cell comb ------------------------------------
module comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo);

input nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi;
output con,cop,scno,scpo;
wire con,cop,scno,scpo;
wire nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi bi;

 assign con= bi&( ~np&s0n&(~sbp|~scp|~s1p|~mi) |
          ~mi&~s1p&(np|~s0p) | ~mi&sbp&scp&(~s0p|s1n) |
          nn&~s0p | np&s0p&s1n | ~np&~s0p&s1n&(~scp|~sbp) ) |
          ~mi&( ~np&s0n&(~sbp|~scp|~s1p) |
          nn&~s0p | np&s0p&s1n | ~np&~s0p&s1n&(~sbp|~scp)) |
          ~np&s0n&(~s1p&~scp | s1n | ~s1p&~sbp);

 assign cop= ((s1p^(sbp&scp)) ^ mi) ^ bi;

 assign scno=scn;
 assign scpo=scp;

endmodule

------------------------ cell mcell1 ------------------------------------
module mcell1(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

  input mds,as,qn,qp,yn,yp;
  output bn,bp,nn,np;
  wire bn,bp,nn,np;

 assign bn=~mds&~yn&yp | mds&~as&qn&qp | ~mds&as;
 assign bp=~mds&yp | mds&qp | as;

 assign nn=~(qn | ~qp | mds&~as);
 assign np=~(mds&~as | ~qp);
endmodule

------------------------ cell mcell ------------------------------------
module mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np);
```

```
  input mds,as,qn,qp,yn,yp;
  output bn,bp,nn,np;
  wire bn,bp,nn,np,qn,qp,mds,as;

  assign bn=~mds&~as&~yn&yp | mds&~as&qn&qp;
  assign bp=~mds&~as&yp | mds&~as&qp;

  assign nn=~(qn | ~qp | mds&~as);
  assign np=~(mds&~as | ~qp);
endmodule

------------------------- select ------------------------------------------
module select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp);

  input aD,s0n,s0p,s1n,s1p,s2n,s2p;
  output seln,selp;
  wire seln,selp;
  wire aD,s0n,s0p,s1n,s1p,s2n,s2p;

  assign seln=(aD & ~s0p & ~s1n & s1p & ~s2n) |
            (aD & ~s0n & s0p &(~s2n | ~s1n)) |
            (~aD & ~s0p & s1n &(s2n | ~s2p)) |
            (~aD & s0n & (~s1p | ~s2p | s1n | s2n));

  assign selp=(~s0n&s0p&~(s1n&s2n)) | (~s0p&~s1n&s1p&~s2n) |
            (~s0p&s1n&(s2n|~s2p)) | (s0n&~(~s1n&s1p&~s2n&s2p));

endmodule

-------------------------- row1 ------------------------------------------
module row1(sn,sp,scn,scp,mds,sdm,as,yn,yp,sno,spo,scno,scpo,qno,qpo);

input [0:5] sn,sp,scn,scp;
input mds,sdm,as,yn,yp;
output [0:5] sno,spo,scno,scpo;
output qno,qpo;
wire qno,qpo,seltmp;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c5(sn[5],sp[5],bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
        mi[5],bi[5],scno[5],scpo[5]),

      c4(sn[4],sp[4],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
        mi[4],bi[4],scno[4],scpo[4]),

      c3(sn[3],sp[3],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
        mi[3],bi[3],scno[3],scpo[3]),
      c2(sn[2],sp[2],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
         mi[2],bi[2],scno[2],scpo[2]),

      c1(sn[1],sp[1],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
         mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,1'b0,1'b0,sn[0],sp[0],bn,bp,scn[0],scp[0],mi[1],bi[1],
           sno[0],spo[0],scno[0],scpo[0]);

// mcell1(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell1 mc(mds,as,1'b0,1'b0,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

assign seltmp=~as&mds;

select sel1(scn[0],sn[1],seltmp,1'b0,1'b0,1'b0,1'b0,qno,qpo);

endmodule

------------------------- row2 -------------------------------------------
module row2(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c5(1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
         mi[5],bi[5],scno[5],scpo[5]),

      c4(sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
         mi[4],bi[4],scno[4],scpo[4]),

      c3(sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
         mi[3],bi[3],scno[3],scpo[3]),

      c2(sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
         mi[2],bi[2],scno[2],scpo[2]);

// cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)
```

```
cell3 c1(sdm,sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
        mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
          sno[0],spo[0],scno0,scpo0);

assign scno[0]=sdm&scno0 | ~sdm&bn;
assign scpo[0]=sdm&scpo0 | ~sdm&bp;

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

---------------------------- row3 ---------------------------------------------
module row3(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c5(1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
        mi[5],bi[5],scno[5],scpo[5]),

     c4(sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
        mi[4],bi[4],scno[4],scpo[4]),

     c3(sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
        mi[3],bi[3],scno[3],scpo[3]);

// cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell3 c2(sdm,sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
        mi[2],bi[2],scno[2],scpo[2]);

// cell1a(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)
```

```
cell1a c1(sdm,sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
        mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
          sno[0],spo[0],scno[0],scpo[0]);

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

------------------------- row4 -------------------------------------------
module row4(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1  c5(1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
        mi[5],bi[5],scno[5],scpo[5]),

      c4(sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
        mi[4],bi[4],scno[4],scpo[4]);

// cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell3 c3(sdm,sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
        mi[3],bi[3],scno[3],scpo[3]);

// cell1a((sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1a c2(sdm,sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
        mi[2],bi[2],scno[2],scpo[2]);

cell1  c1(sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
        mi[1],bi[1],scno[1],scpo[1]);
```

```
// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
        sno[0],spo[0],scno[0],scpo[0]);

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

---------------------------- row5 --------------------------------------------
module row5(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c5(1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
       mi[5],bi[5],scno[5],scpo[5]);

// cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell3 c4(sdm,sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
       mi[4],bi[4],scno[4],scpo[4]);

// cell1a(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1a c3(sdm,sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
       mi[3],bi[3],scno[3],scpo[3]);

cell1 c2(sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
       mi[2],bi[2],scno[2],scpo[2]),

     c1(sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
       mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
          sno[0],spo[0],scno[0],scpo[0]);

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

------------------------- row6 -------------------------------------------
module row6(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell3(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell3 c5(sdm,1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
        mi[5],bi[5],scno[5],scpo[5]);

// cell1a(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1a c4(sdm,sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
        mi[4],bi[4],scno[4],scpo[4]);

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c3(sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
        mi[3],bi[3],scno[3],scpo[3]),

      c2(sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
        mi[2],bi[2],scno[2],scpo[2]),

      c1(sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
        mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
          sno[0],spo[0],scno[0],scpo[0]);

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)
```

```
mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)

select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

--------------------------- row7 --------------------------------------------
module row7(sn,sp,scn,scp,mds,sdm,as,yn,yp,qn,qp,sno,spo,scno,scpo,qno,qpo);

input [0:5] scn,scp;
input [0:5] sn,sp;
input mds,sdm,as,yn,yp,qn,qp;
output [0:5] scno,scpo,sno,spo;
output qno,qpo;
wire qno,qpo;
wire [0:5] sn,sp,scn,scp,scno,scpo,sno,spo;
wire [1:5] mi,bi;
wire bn,bp;

// cell1a(sdm,sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1a c5(sdm,1'b0,1'b0,bn,bp,scn[5],scp[5],1'b0,1'b0,sno[5],spo[5],
        mi[5],bi[5],scno[5],scpo[5]);

// cell1(sn,sp,sbn,sbp,scn,scp,mi_1,bi_1,sno,spo,mi,bi,scno,scpo)

cell1 c4(sn[5],sp[5],bn,bp,scn[4],scp[4],mi[5],bi[5],sno[4],spo[4],
        mi[4],bi[4],scno[4],scpo[4]),

     c3(sn[4],sp[4],bn,bp,scn[3],scp[3],mi[4],bi[4],sno[3],spo[3],
        mi[3],bi[3],scno[3],scpo[3]),

     c2(sn[3],sp[3],bn,bp,scn[2],scp[2],mi[3],bi[3],sno[2],spo[2],
        mi[2],bi[2],scno[2],scpo[2]),

     c1(sn[2],sp[2],bn,bp,scn[1],scp[1],mi[2],bi[2],sno[1],spo[1],
        mi[1],bi[1],scno[1],scpo[1]);

// comb(nn,np,s0n,s0p,s1n,s1p,sbn,sbp,scn,scp,mi,bi,con,cop,scno,scpo)

comb comb1(nn,np,sn[0],sp[0],sn[1],sp[1],bn,bp,scn[0],scp[0],mi[1],bi[1],
          sno[0],spo[0],scno[0],scpo[0]);

// mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np)

mcell mc(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

// select(aD,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp)
```

```
select sel1(scn[0],sno[0],spo[0],sno[1],spo[1],sno[2],spo[2],qno,qpo);

endmodule

---------------------------- allrow -------------------------------
module allrow(sn,sp,scn,scp,mds,sdm,as,yn,yp,sno,spo,qno,qpo,clk);

input [0:5] sn,sp,scn,scp;
input [0:5] yn,yp;
input mds,sdm,as,clk;
output [0:5] sno,spo;
output [0:6] qno,qpo;
wire [0:6] qno,qpo;
wire [0:5] sn,sp,sno1,spo1,sno2,spo2,sno3,spo3;
wire [0:5] sno4,spo4,sno5,spo5,sno6,spo6,sno7,spo7;
wire [0:5] scno1,scpo1,scno2,scpo2,scno3,scpo3;
wire [0:5] scno4,scpo4,scno5,scpo5,scno6,scpo6,scno7,scpo7;
wire clk;

row1 r1(sn,sp,scn,scp,mds,sdm,as,yn[0],yp[0],
        sno1,spo1,scno1,scpo1,qno[0],qpo[0]);

row2 r2(sno1,spo1,scno1,scpo1,mds,sdm,as,yn[1],yp[1],qno[0],qpo[0],
        sno2,spo2,scno2,scpo2,qno[1],qpo[1]);

row3 r3(sno2,spo2,scno2,scpo2,mds,sdm,as,yn[2],yp[2],qno[1],qpo[1],
        sno3,spo3,scno3,scpo3,qno[2],qpo[2]);

row4 r4(sno3,spo3,scno3,scpo3,mds,sdm,as,yn[3],yp[3],qno[2],qpo[2],
        sno4,spo4,scno4,scpo4,qno[3],qpo[3]);

row5 r5(sno4,spo4,scno4,scpo4,mds,sdm,as,yn[4],yp[4],qno[3],qpo[3],
        sno5,spo5,scno5,scpo5,qno[4],qpo[4]);

row6 r6(sno5,spo5,scno5,scpo5,mds,sdm,as,yn[5],yp[5],qno[4],qpo[4],
        sno6,spo6,scno6,scpo6,qno[5],qpo[5]);

row7 r7(sno6,spo6,scno6,scpo6,mds,sdm,as,1'b0,1'b0,qno[5],qpo[5],
        sno,spo,scno7,scpo7,qno[6],qpo[6]);

comp_res cr(sn,sp,qno,qpo,scn,scp,yn,yp,sdm,mds,as,clk);

endmodule
```

# Appendix B

## SBNR-BINARY CONVERTER & ITS OUTPUT

```
---------------------------- SBNR-BINARY converter --------------------------------

module comp_res(sn,sp,qno,qpo,scn,scp,yn,yp,sdm,mds,as,clk);

input [0:5] sn,sp,scn,scp;
input [0:6] qno,qpo;
input [0:5] yn,yp;
input sdm,mds,as,clk;
real itmp,ktmp,ktmpc,tmp1,tmp2,tqp,ts,tc,typ,org,result;
integer i,k;

always @(posedge clk)
begin
tqp=0;
org=0;
tc=0;
ts=0;
typ=0;
result=0;
if (mds==1 && sdm==1 && as==0)
begin
 itmp=0;
 ktmp=0;
end
if (mds==1 && sdm==0 && as==0)
begin
 itmp=0;
 ktmp=1;
 ktmpc=1;
end
if (mds==0 && sdm==1 && as==0)
begin
 itmp=1;
 ktmp=1;
 ktmpc=2;
end
if (as==1)
begin
 itmp=2;
 ktmp=2;
 ktmpc=2;
end

 for(i=0; i<=5; i=i+1)
 begin
  if(yp[i]==1)
  begin
   tmp1=1.0;
   for(k=itmp; k<=i+1; k=k+1)
   tmp1=tmp1*(1.0/2.0);
   if(yn[i]==1)
    typ=typ-tmp1;
   else
```

```
     typ=typ+tmp1;
   end
 end

 for(i=itmp; i<=6; i=i+1)
 begin
  if(qpo[i]==1)
  begin
    tmp1=1.0;
    for(k=itmp; k<=i; k=k+1)
    tmp1=tmp1*(1.0/2.0);
    if(qno[i]==1)
        tqp=tqp-tmp1;
    else
      tqp=tqp+tmp1;
  end
 end

 for(i=itmp; i<=5; i=i+1)
 begin
  if(sp[i]==1)
  begin
    tmp1=1.0;
    for(k=ktmp; k<=i; k=k+1)
    tmp1=tmp1*(1.0/2.0);
    if(sn[i]==1)
        ts=ts-tmp1;
    else
        ts=ts+tmp1;
  end
  if(scp[i]==1)
  begin
    tmp2=1.0;
    for(k=ktmpc; k<=i; k=k+1)
    tmp2=tmp2*(1.0/2.0);
    if (scn[i]==1)
        tc=tc-tmp2;
    else
        tc=tc+tmp2;
  end
 end

 if(mds==1 && sdm==1 && as==0)
 begin
  org=ts/tc;
  $display("*** Division (N/D) ***");
   $display("sn=%b sp=%b scn=%b scp=%b",sn,sp,scn,scp);
   $display("qn=%b qp=%b",qno,qpo);
  $display("N= %g\tD= %g",ts,tc);
  $display("Result= %f\tOriginal= %f\n",tqp,org);
 end
 if(mds==1 && sdm==0 && as==0)
 begin
```

```
  org=tqp;
  $display("*** Square Root ( S=Q*Q ) ***");
   $display("sn=%b sp=%b",sn,sp);
   $display("qn=%b qp=%b",qno,qpo);
   $display("S= %g",ts);
  $display("Result= %f\tOriginal= %f\n",tqp*tqp,ts);
 end
 if(mds==0 && sdm==1 && as==0)
 begin
  org=ts+tc*typ;
  $display("*** Multiplication-accumulate ( X*Y + A ) ***");
   $display("sn=%b sp=%b scn=%b scp=%b",sn,sp,scn,scp);
   $display("qn=%b qp=%b yn=%b yp=%b",qno,qpo,yn,yp);
  $display("X= %g\tY= %g\tA= %g",tc,typ,ts);
  $display("Result= %f\tOriginal= %f\n",tqp,org);
 end
 if(sdm==1 && as==1)
 begin
  org=ts+tc;
  $display("*** Addition & Subtraction ( X + A or X - A) ***");
   $display("mds=%b as=%b sn=%b sp=%b scn=%b scp=%b",mds,as,sn,sp,scn,scp);
   $display("qn=%b qp=%b yn=%b yp=%b",qno,qpo,yn,yp);
  $display("X= %g\tY= %g\tA= %g",tc,typ,ts);
  $display("Result= %f\tOriginal= %f\n",tqp,org);
 end
end

endmodule
```

```
------------------------- Division ----------------------------

Host command: /cmc1/verilogXL/sun4_4.1_1.6.0.1/exe/verilog-grx
Command arguments:
   -p2e0d6d30
   test-all.v
   -y /homes/hchan/Veritools/Verilog/NCA
   +libext+.v

VERILOG-XL 1.6.0.1 log file created Dec 20, 1992  05:53:30
 * Copyright Cadence Design Systems, Inc. 1985, 1988.   *
 *    All Rights Reserved.      Licensed Software.    *
 * Confidential and proprietary information which is the *
 *      property of Cadence Design Systems, Inc.        *
Compiling source file "test-all.v"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Highest level modules:
test

*** Division (N/D) ***
sn=000000 sp=000000 scn=000000 scp=000000
qn=0000000 qp=1000000
N= 0   D= 0
Result= 0.500000      Original= NaN

*** Division (N/D) ***
sn=000010 sp=011111 scn=000000 scp=100111
qn=0001000 qp=1101001
N= 0.421875  D= 0.609375
Result= 0.695312      Original= 0.692308

*** Division (N/D) ***
sn=011000 sp=011111 scn=000000 scp=100110
qn=1000001 qp=1001001
N= -0.265625           D= 0.59375
Result= -0.445312     Original= -0.447368

*** Division (N/D) ***
sn=000000 sp=011000 scn=000001 scp=100101
qn=0001000 qp=1101000
N= 0.375       D= 0.546875
Result= 0.687500      Original= 0.685714

*** Division (N/D) ***
sn=000000 sp=011110 scn=000000 scp=100000
qn=0000000 qp=1111000
N= 0.46875   D= 0.5
Result= 0.937500      Original= 0.937500

*** Division (N/D) ***
sn=011100 sp=011110 scn=000000 scp=100010
qn=1100010 qp=1100010
```

```
N= -0.40625  D= 0.53125
Result= -0.765625    Original= -0.764706

*** Division (N/D) ***
sn=000000 sp=011011 scn=000000 scp=100010
qn=0000010 qp=1101010
N= 0.421875  D= 0.53125
Result= 0.796875     Original= 0.794118

*** Division (N/D) ***
sn=000100 sp=011100 scn=000000 scp=100101
qn=0000011 qp=1001011
N= 0.3125     D= 0.578125
Result= 0.539062     Original= 0.540541

*** Division (N/D) ***
sn=011000 sp=011101 scn=000000 scp=100011
qn=1000110 qp=1000110
N= -0.296875          D= 0.546875
Result= -0.546875    Original= -0.542857

*** Division (N/D) ***
sn=000110 sp=011111 scn=000000 scp=100000
qn=0000100 qp=1010100
N= 0.296875  D= 0.5
Result= 0.593750     Original= 0.593750

*** Division (N/D) ***
sn=000000 sp=011000 scn=000000 scp=100011
qn=0001000 qp=1101000
N= 0.375      D= 0.546875
Result= 0.687500     Original= 0.685714

*** Division (N/D) ***
sn=000000 sp=011010 scn=000000 scp=100110
qn=0001000 qp=1101000
N= 0.40625   D= 0.59375
Result= 0.687500     Original= 0.684211

*** Division (N/D) ***
sn=000001 sp=011101 scn=100100 scp=100100
qn=1100000 qp=1100000
N= 0.421875  D= -0.5625
Result= -0.750000    Original= -0.750000

*** Division (N/D) ***
sn=000000 sp=011111 scn=100001 scp=100001
qn=1111000 qp=1111000
N= 0.484375  D= -0.515625
Result= -0.937500    Original= -0.939394

*** Division (N/D) ***
sn=000000 sp=011100 scn=100001 scp=100001
```

```
qn=1110000 qp=1110100
N= 0.4375    D= -0.515625
Result= -0.843750    Original= -0.848485

*** Division (N/D) ***
sn=000010 sp=011011 scn=000011 scp=100111
qn=0000000 qp=1011001
N= 0.359375  D= 0.515625
Result= 0.695312     Original= 0.696970

*** Division (N/D) ***
sn=000100 sp=011100 scn=000001 scp=100111
qn=0000100 qp=1001101
N= 0.3125    D= 0.578125
Result= 0.539062     Original= 0.540541

*** Division (N/D) ***
sn=011001 sp=011101 scn=000000 scp=100000
qn=1010100 qp=1010100
N= -0.328125         D= 0.5
Result= -0.656250    Original= -0.656250

*** Division (N/D) ***
sn=011100 sp=011101 scn=100001 scp=100001
qn=0000000 qp=1101000
N= -0.421875         D= -0.515625
Result= 0.812500     Original= 0.818182

*** Division (N/D) ***
sn=000000 sp=011000 scn=100100 scp=100100
qn=1100000 qp=1101010
N= 0.375     D= -0.5625
Result= -0.671875    Original= -0.666667

*** Division (N/D) ***
sn=000000 sp=011000 scn=000000 scp=100110
qn=0000000 qp=1010001
N= 0.375     D= 0.59375
Result= 0.632812     Original= 0.631579

L38 "test-all.v": $stop at simulation time 210
Type ? for help
C1 > $finish;
C1: $finish at simulation time 210
11117 simulation events
CPU time: 0 secs to compile + 0 secs to link + 4 secs in simulation
End of VERILOG-XL 1.6.0.1   Dec 20, 1992  05:53:51


------------------------ Square Root ---------------------------

Host command: /cmc1/verilogXL/sun4_4.1_1.6.0.1/exe/verilog-grx
Command arguments:
  -p2e0d6d30
  test-all.v
  -y /homes/hchan/Veritools/Verilog/NCA
  +libext+.v

VERILOG-XL 1.6.0.1 log file created Dec 20, 1992 05:55:21
 * Copyright Cadence Design Systems, Inc. 1985, 1988.   *
 *    All Rights Reserved.     Licensed Software.   *
 * Confidential and proprietary information which is the *
 *     property of Cadence Design Systems, Inc.         *
Compiling source file "test-all.v"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Highest level modules:
test

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=000000
qn=0101111 qp=1101111
S= 0
Result= 0.017639     Original= 0.000000

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011001
qn=0000001 qp=1110011
S= 0.78125
Result= 0.779358     Original= 0.781250

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011001
qn=0000001 qp=1110011
S= 0.78125
Result= 0.779358     Original= 0.781250

*** Square Root ( S=Q*Q ) ***
sn=000100 sp=011110
qn=0001010 qp=1111110
S= 0.6875
Result= 0.685791     Original= 0.687500

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011100
qn=0000000 qp=1111000
S= 0.875
Result= 0.878906     Original= 0.875000

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011100
qn=0000000 qp=1111000
S= 0.875
Result= 0.878906     Original= 0.875000

*** Square Root ( S=Q*Q ) ***
```

```
sn=000100 sp=011101
qn=0001000 qp=1111000
S= 0.65625
Result= 0.660156      Original= 0.656250

*** Square Root ( S=Q*Q ) ***
sn=000010 sp=011011
qn=0000100 qp=1110101
S= 0.71875
Result= 0.725159      Original= 0.718750

*** Square Root ( S=Q*Q ) ***
sn [illegible]001 sp=011111
q[illegible] 00010 qp=1111110
S[illegible] 90625
Result= 0.908447      Original= 0.906250

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011001
qn=0000001 qp=1110011
S= 0.78125
Result= 0.779358      Original= 0.781250

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011011
qn=0000101 qp=1111111
S= 0.84375
Result= 0.835510      Original= 0.843750

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011001
qn=0000001 qp=1110011
S= 0.78125
Result= 0.779358      Original= 0.781250

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011100
qn=0000000 qp=1111000
S= 0.875
Result= 0.878906      Original= 0.875000

*** Square Root ( S=Q*Q ) ***
sn=000100 sp=011100
qn=0000101 qp=1101111
S= 0.625
Result= 0.622620      Original= 0.625000

*** Square Root ( S=Q*Q ) ***
sn=000010 sp=011010
qn=0001010 qp=1111110
S= 0.6875
Result= 0.685791      Original= 0.687500
```

```
*** Square Root ( S=Q*Q ) ***
sn=000011 sp=011011
qn=0001000 qp=1111000
S= 0.65625
Result= 0.660156     Original= 0.656250

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011000
qn=0000010 qp=1110011
S= 0.75
Result= 0.752014     Original= 0.750000

*** Square Root ( S=Q*Q ) ***
sn=000001 sp=011101
qn=0000101 qp=1111111
S= 0.84375
Result= 0.835510     Original= 0.843750

*** Square Root ( S=Q*Q ) ***
sn=000101 sp=011101
qn=0000010 qp=1100111
S= 0.59375
Result= 0.598206     Original= 0.593750

*** Square Root ( S=Q*Q ) ***
sn=000001 sp=011101
qn=0000101 qp=1111111
S= 0.84375
Result= 0.835510     Original= 0.843750

*** Square Root ( S=Q*Q ) ***
sn=000000 sp=011100
qn=0000000 qp=1111000
S= 0.875
Result= 0.878906     Original= 0.875000

L38 "test-all.v": $stop at simulation time 210
Type ? for help
C1 > $finish;
C1: $finish at simulation time 210
8480 simulation events
CPU time: 0 secs to compile + 0 secs to link + 2 secs in simulation
End of VERILOG-XL 1.6.0.1   Dec 20, 1992  05:55:32


---------------------- Multiplication --------------------------

Host command: /cmc1/verilogXL/sun4_4.1_1.6.0.1/exe/verilog-grx
Command arguments:
   -p2e0d6d30
   test-all.v
   -y /homes/hchan/Veritools/Verilog/NCA
   +libext+.v
```

```
VERILOG-XL 1.6.0.1 log file created Dec 20, 1992  05:58:48
 * Copyright Cadence Design Systems, Inc. 1985, 1988.   *
 *    All Rights Reserved.      Licensed Software.   *
 * Confidential and proprietary information which is the *
 *     property of Cadence Design Systems, Inc.         *
Compiling source file "test-all.v"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA"
Highest level modules:
test

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=000000 scn=000000 scp=000000
qn=0000000 qp=0000000 yn=000000 yp=000000
X= 0   Y= 0   A= 0
Result= 0.000000       Original= 0.000000

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001000 sp=001000 scn=000001 scp=000111
qn=0000100 qp=0000110 yn=000000 yp=101101
X= 0.3125      Y= 0.703125  A= -0.25
Result= -0.031250    Original= -0.030273

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001000 scn=000000 scp=000110
qn=0000110 qp=0101111 yn=000101 yp=111101
X= 0.375       Y= 0.796875  A= 0.25
Result= 0.546875      Original= 0.548828

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001000 scn=000100 scp=000100
qn=0001100 qp=0011110 yn=001000 yp=111001
X= -0.25       Y= 0.640625  A= 0.25
Result= 0.093750      Original= 0.089844

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000010 sp=001110 scn=000000 scp=000110
qn=0000011 qp=0100111 yn=001100 yp=111100
X= 0.375       Y= 0.5625     A= 0.3125
Result= 0.515625      Original= 0.523438

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001110 scn=000000 scp=000110
qn=0000000 qp=0110000 yn=000000 yp=110101
X= 0.375       Y= 0.828125  A= 0.4375
Result= 0.750000      Original= 0.748047

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001001 sp=001001 scn=000001 scp=000111
qn=0010010 qp=0011110 yn=001100 yp=111110
X= 0.3125      Y= 0.59375    A= -0.28125
Result= -0.093750    Original= -0.095703
```

```
*** Multiplication-accumulate ( X*Y + A ) ***
sn=001010 sp=001011 scn=000110 scp=000111
qn=0100011 qp=0100111 yn=000000 yp=101000
X= -0.3125    Y= 0.625       A= -0.28125
Result= -0.484375    Original= -0.476562

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001111 scn=000110 scp=000111
qn=0010010 qp=0110111 yn=000000 yp=100100
X= -0.3125    Y= 0.5625      A= 0.46875
Result= 0.296875     Original= 0.292969

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001100 sp=001100 scn=000001 scp=000111
qn=0100010 qp=0110110 yn=000000 yp=100000
X= 0.3125     Y= 0.5         A= -0.375
Result= -0.218750    Original= -0.218750

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001100 sp=001100 scn=000100 scp=000100
qn=0101010 qp=0101111 yn=000000 yp=110101
X= -0.25      Y= 0.828125  A= -0.375
Result= -0.578125    Original= -0.582031

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001101 sp=001101 scn=000000 scp=000111
qn=0001001 qp=0001111 yn=000001 yp=110111
X= 0.4375     Y= 0.828125  A= -0.40625
Result= -0.046875    Original= -0.043945

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001001 scn=000000 scp=000110
qn=0001000 qp=0111000 yn=000010 yp=111110
X= 0.375      Y= 0.90625   A= 0.28125
Result= 0.625000     Original= 0.621094

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001001 scn=000000 scp=000101
qn=0001001 qp=0101111 yn=000000 yp=100011
X= 0.3125     Y= 0.546875  A= 0.28125
Result= 0.453125     Original= 0.452148

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001000 sp=001000 scn=000101 scp=000101
qn=0100010 qp=0100011 yn=000100 yp=111110
X= -0.3125    Y= 0.84375     A= -0.25
Result= -0.515625    Original= -0.513672

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001001 sp=001001 scn=000101 scp=000101
qn=0100000 qp=0100001 yn=001001 yp=111101
X= -0.3125    Y= 0.671875  A= -0.28125
```

```
Result= -0.484375     Original= -0.491211

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001010 sp=001011 scn=000000 scp=000100
qn=0010000 qp=0011000 yn=001001 yp=111011
X= 0.25         Y= 0.640625  A= -0.28125
Result= -0.125000     Original= -0.121094

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000010 sp=001111 scn=000000 scp=000101
qn=0000100 qp=0101110 yn=000000 yp=110011
X= 0.3125      Y= 0.796875  A= 0.34375
Result= 0.593750      Original= 0.592773

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000000 sp=001110 scn=000111 scp=000111
qn=0011010 qp=0111111 yn=000010 yp=101011
X= -0.4375     Y= 0.609375  A= 0.4375
Result= 0.171875      Original= 0.170898

*** Multiplication-accumulate ( X*Y + A ) ***
sn=001010 sp=001011 scn=000001 scp=000111
qn=0001000 qp=0001101 yn=000000 yp=101110
X= 0.3125      Y= 0.71875    A= -0.28125
Result= -0.046875     Original= -0.056641

*** Multiplication-accumulate ( X*Y + A ) ***
sn=000001 sp=001011 scn=000000 scp=000110
qn=0001101 qp=0111111 yn=000100 yp=111100
X= 0.375       Y= 0.8125     A= 0.28125
Result= 0.578125      Original= 0.585938

L58 "test-all.v": $stop at simulation time 210
Type ? for help
C1 > $finish;
C1: $finish at simulation time 210
9753 simulation events
CPU time: 0 secs to compile + 0 secs to link + 3 secs in simulation
End of VERILOG-XL 1.6.0.1  Dec 20, 1992  05:59:02


------------------------ Addition ---------------------------

Host command: /cmc1/verilogXL/sun4_4.1_1.6.0.1/exe/verilog-grx
Command arguments:
   -p1e387bf1
   tmp.v
   -y /homes/hchan/Veritools/Verilog/NCA/Arch_behave
   +libext+.v
   -y /homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows
   +libext+.v

VERILOG-XL 1.6.0.1 log file created Feb 24, 1993  07:09:22
```

```
 * Copyright Cadence Design Systems, Inc. 1985, 1988.  *
 *    All Rights Reserved.     Licensed Software.   *
 * Confidential and proprietary information which is the *
 *     property of Cadence Design Systems, Inc.        *
Compiling source file "tmp.v"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Highest level modules:
test

*** Addition (A+X) ***
mds=0 as=1 sn=000000 sp=000000 scn=000000 scp=000000
qn=000000 qp=000000 yn=000000 yp=000000
X= 0   Y= 0   A= 0
Result= 0.000000     Original= 0.000000

*** Addition (A+X) ***
mds=0 as=1 sn=001110 sp=001110 scn=000100 scp=000100
qn=010100 qp=010110 yn=100000 yp=100000
X= -0.125      Y= -0.25      A= -0.4375
Result= -0.562500    Original= -0.562500

*** Addition (A+X) ***
mds=0 as=1 sn=000100 sp=000111 scn=000110 scp=000110
qn=010001 qp=011011 yn=100000 yp=100000
X= -0.1875    Y= -0.25      A= -0.03125
Result= -0.218750    Original= -0.218750

*** Addition (A+X) ***
mds=0 as=1 sn=000101 sp=001111 scn=001110 scp=001111
qn=010000 qp=011000 yn=100000 yp=100000
X= -0.40625  Y= -0.25      A= 0.15625
Result= -0.250000    Original= -0.250000

*** Addition (A+X) ***
mds=0 as=1 sn=000001 sp=000101 scn=000000 scp=000100
qn=001010 qp=011011 yn=100000 yp=100000
X= 0.125       Y= -0.25      A= 0.09375
Result= 0.218750     Original= 0.218750

*** Addition (A+X) ***
mds=0 as=1 sn=000000 sp=001101 scn=001100 scp=001100
qn=000001 qp=000011 yn=100000 yp=100000
X= -0.375      Y= -0.25      A= 0.40625
Result= 0.031250     Original= 0.031250

*** Addition (A+X) ***
mds=0 as=1 sn=000000 sp=000110 scn=000000 scp=001101
qn=000110 qp=011111 yn=100000 yp=100000
X= 0.40625   Y= -0.25      A= 0.1875
```

```
Result= 0.593750      Original= 0.593750

*** Addition (A + X) ***
mds=0 as=1 sn=000001 sp=000101 scn=000000 scp=000101
qn=001000 qp=011000 yn=100000 yp=100000
X= 0.15625   Y= -0.25       A= 0.09375
Result= 0.250000      Original= 0.250000

*** Addition (A + X) ***
mds=0 as=1 sn=000100 sp=001100 scn=000101 scp=000101
qn=000010 qp=000011 yn=100000 yp=100000
X= -0.15625  Y= -0.25       A= 0.125
Result= -0.031250     Original= -0.031250

*** Addition (A + X) ***
mds=0 as=1 sn=000000 sp=001101 scn=000000 scp=000100
qn=000001 qp=010011 yn=100000 yp=100000
X= 0.125       Y= -0.25       A= 0.40625
Result= 0.531250      Original= 0.531250

*** Addition (A + X) ***
mds=0 as=1 sn=000000 sp=001111 scn=000101 scp=000101
qn=001010 qp=011110 yn=100000 yp=100000
X= -0.15625  Y= -0.25       A= 0.46875
Result= 0.312500      Original= 0.312500

*** Addition (A + X) ***
mds=0 as=1 sn=000001 sp=001111 scn=000000 scp=000111
qn=000100 qp=011100 yn=100000 yp=100000
X= 0.21875   Y= -0.25       A= 0.40625
Result= 0.625000      Original= 0.625000

*** Addition (A + X) ***
mds=0 as=1 sn=001100 sp=001110 scn=001111 scp=001111
qn=011010 qp=011011 yn=100000 yp=100000
X= -0.46875  Y= -0.25       A= -0.3125
Result= -0.781250     Original= -0.781250

*** Addition (A + X) ***
mds=0 as=1 sn=001101 sp=001101 scn=000000 scp=000101
qn=010000 qp=011000 yn=100000 yp=100000
X= 0.15625   Y= -0.25       A= -0.40625
Result= -0.250000     Original= -0.250000

*** Addition (A + X) ***
mds=0 as=1 sn=001100 sp=001101 scn=001100 scp=001100
qn=011001 qp=011011 yn=100000 yp=100000
X= -0.375      Y= -0.25       A= -0.34375
Result= -0.718750     Original= -0.718750

*** Addition (A + X) ***
mds=0 as=1 sn=000100 sp=001111 scn=001110 scp=001110
qn=010001 qp=011011 yn=100000 yp=100000
```

```
X= -0.4375    Y= -0.25       A= 0.21875
Result= -0.218750    Original= -0.218750

*** Addition (A + X) ***
mds=0 as=1 sn=000110 sp=000111 scn=000001 scp=000111
qn=000000 qp=000000 yn=100000 yp=100000
X= 0.15625   Y= -0.25       A= -0.15625
Result= 0.000000     Original= 0.000000

*** Addition (A + X) ***
mds=0 as=1 sn=000100 sp=000101 scn=000000 scp=001110
qn=001010 qp=011111 yn=100000 yp=100000
X= 0.4375     Y= -0.25       A= -0.09375
Result= 0.343750     Original= 0.343750

*** Addition (A + X) ***
mds=0 as=1 sn=000100 sp=001101 scn=000101 scp=000101
qn=000000 qp=000000 yn=100000 yp=100000
X= -0.15625  Y= -0.25       A= 0.15625
Result= 0.000000     Original= 0.000000

*** Addition (A + X) ***
mds=0 as=1 sn=000000 sp=000110 scn=000100 scp=001100
qn=001010 qp=011110 yn=100000 yp=100000
X= 0.125      Y= -0.25       A= 0.1875
Result= 0.312500     Original= 0.312500

*** Addition (A + X) ***
mds=0 as=1 sn=000101 sp=001111 scn=000000 scp=001110
qn=000110 qp=011111 yn=100000 yp=100000
X= 0.4375     Y= -0.25       A= 0.15625
Result= 0.593750     Original= 0.593750

L58 "tmp.v": $stop at simulation time 210
Type ? for help
C1 > $finish;
C1: $finish at simulation time 210
6296 simulation events
CPU time: 0 secs to compile + 0 secs to link + 1 secs in simulation
End of VERILOG-XL 1.6.0.1  Feb 24, 1993  07:09:40


---------------------- Subtraction ---------------------------

Host command: /cmc1/verilogXL/sun4_4.1_1.6.0.1/exe/verilog-grx
Command arguments:
   -p1e387bf1
   tmp.v
   -y /homes/hchan/Veritools/Verilog/NCA/Arch_behave
   +libext+.v
   -y /homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows
   +libext+.v
```

```
VERILOG-XL 1.6.0.1 log file created Feb 24, 1993 07:13:08
 * Copyright Cadence Design Systems, Inc. 1985, 1988.  *
 *    All Rights Reserved.      Licensed Software.    *
 * Confidential and proprietary information which is the *
 *     property of Cadence Design Systems, Inc.        *
Compiling source file "tmp.v"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave/Rows"
Scanning library directory "/homes/hchan/Veritools/Verilog/NCA/Arch_behave"
Highest level modules:
test

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=000000 scn=000000 scp=000000
qn=000000 qp=000000 yn=000000 yp=000000
X= 0   Y= 0   A= 0
Result= 0.000000     Original= 0.000000

*** Subtraction (A - X) ***
mds=1 as=1 sn=001100 sp=001100 scn=000100 scp=000100
qn=010000 qp=011000 yn=000000 yp=100000
X= -0.125     Y= 0.25        A= -0.375
Result= -0.250000    Original= -0.250000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000010 sp=000010 scn=000000 scp=000101
qn=010001 qp=011011 yn=000000 yp=100000
X= 0.15625    Y= 0.25        A= -0.0625
Result= -0.218750    Original= -0.218750

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=001101 scn=000001 scp=000111
qn=001000 qp=011000 yn=000000 yp=100000
X= 0.15625    Y= 0.25        A= 0.40625
Result= 0.250000     Original= 0.250000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000100 sp=001100 scn=000100 scp=000100
qn=001000 qp=011000 yn=000000 yp=100000
X= -0.125     Y= 0.25        A= 0.125
Result= 0.250000     Original= 0.250000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000101 sp=000101 scn=000000 scp=000100
qn=010010 qp=011011 yn=000000 yp=100000
X= 0.125      Y= 0.25        A= -0.15625
Result= -0.281250    Original= -0.281250

*** Subtraction (A - X) ***
mds=1 as=1 sn=001110 sp=001110 scn=000000 scp=000100
qn=010100 qp=010110 yn=000000 yp=100000
```

```
X= 0.125       Y= 0.25        A= -0.4375
Result= -0.562500     Original= -0.562500

*** Subtraction (A - X) ***
mds=1 as=1 sn=001100 sp=001100 scn=001100 scp=001100
qn=000000 qp=000000 yn=000000 yp=100000
X= -0.375      Y= 0.25        A= -0.375
Result= 0.000000      Original= 0.000000

*** Subtraction (A - X) ***
mds=1 as=1 sn=001000 sp=001100 scn=000001 scp=000111
qn=010010 qp=011011 yn=000000 yp=100000
X= 0.15625     Y= 0.25        A= -0.125
Result= -0.281250     Original= -0.281250

*** Subtraction (A - X) ***
mds=1 as=1 sn=000110 sp=000010 scn=001101 scp=001101
qn=001010 qp=011111 yn=000000 yp=100000
X= -0.40625    Y= 0.25        A= -0.0625
Result= 0.343750      Original= 0.343750

*** Subtraction (A - X) ***
mds=1 as=1 sn=000001 sp=000101 scn=000000 scp=000111
qn=001000 qp=001100 yn=000000 yp=100000
X= 0.21875     Y= 0.25        A= 0.09375
Result= -0.125000     Original= -0.125000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=000111 scn=000000 scp=001101
qn=001010 qp=001110 yn=000000 yp=100000
X= 0.40625     Y= 0.25        A= 0.21875
Result= -0.187500     Original= -0.187500

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=000011 scn=000000 scp=000101
qn=000100 qp=000110 yn=000000 yp=100000
X= 0.15625     Y= 0.25        A= 0.09375
Result= -0.062500     Original= -0.062500

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=000010 scn=000100 scp=001100
qn=000100 qp=000110 yn=000000 yp=100000
X= 0.125       Y= 0.25        A= 0.0625
Result= -0.062500     Original= -0.062500

*** Subtraction (A - X) ***
mds=1 as=1 sn=001100 sp=001100 scn=000100 scp=000100
qn=010000 qp=011000 yn=000000 yp=100000
X= -0.125      Y= 0.25        A= -0.375
Result= -0.250000     Original= -0.250000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=000101 scn=000000 scp=001100
```

```
qn=010001 qp=011011 yn=000000 yp=100000
X= 0.375      Y= 0.25       A= 0.15625
Result= -0.218750    Original= -0.218750

*** Subtraction (A - X) ***
mds=1 as=1 sn=001100 sp=001100 scn=000000 scp=000101
qn=010010 qp=010011 yn=000000 yp=100000
X= 0.15625    Y= 0.25       A= -0.375
Result= -0.531250    Original= -0.531250

*** Subtraction (A - X) ***
mds=1 as=1 sn=000010 sp=000110 scn=000100 scp=001100
qn=000100 qp=000110 yn=000000 yp=100000
X= 0.125      Y= 0.25       A= 0.0625
Result= -0.062500    Original= -0.062500

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=001100 scn=000000 scp=001100
qn=000000 qp=000000 yn=000000 yp=100000
X= 0.375      Y= 0.25       A= 0.375
Result= 0.000000     Original= 0.000000

*** Subtraction (A - X) ***
mds=1 as=1 sn=000110 sp=001110 scn=000001 scp=000111
qn=000101 qp=000111 yn=000000 yp=100000
X= 0.15625    Y= 0.25       A= 0.0625
Result= -0.093750    Original= -0.093750

*** Subtraction (A - X) ***
mds=1 as=1 sn=000000 sp=001010 scn=000100 scp=000100
qn=000100 qp=010110 yn=000000 yp=100000
X= -0.125     Y= 0.25       A= 0.3125
Result= 0.437500     Original= 0.437500

L58 "tmp.v": $stop at simulation time 210
Type ? for help
C1 > $finish;
C1: $finish at simulation time 210
5968 simulation events
CPU time: 0 secs to compile + 0 secs to link + 1 secs in simulation
End of VERILOG-XL 1.6.0.1   Feb 24, 1993  07:13:17
```

# Appendix C

## SCHEMATIC DIAGRAMS OF IMPROVED ARCHITECTURE & ORIGINAL ARCHITECTURE

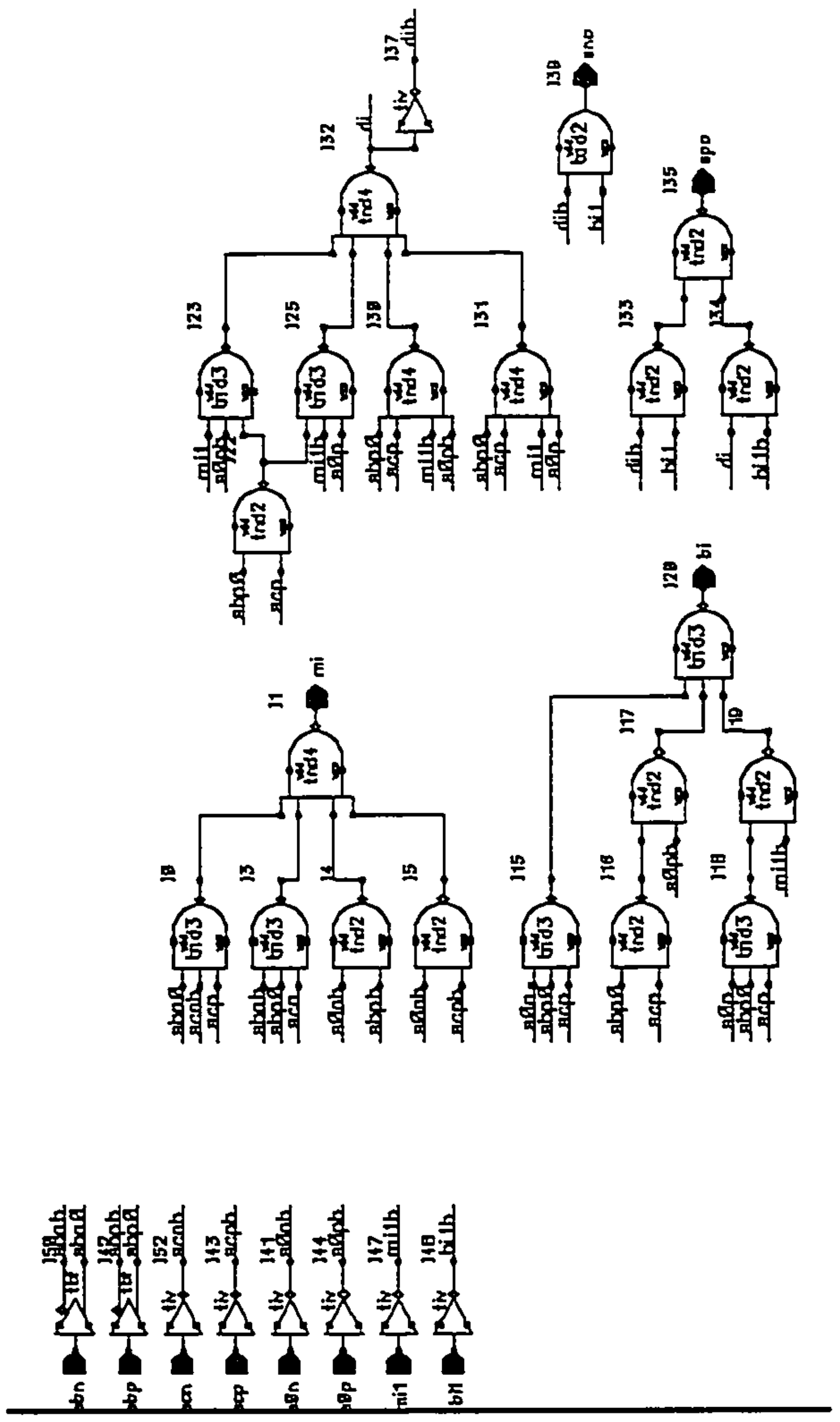


Figure C.1 Type *a* Cell of Improved Architecture

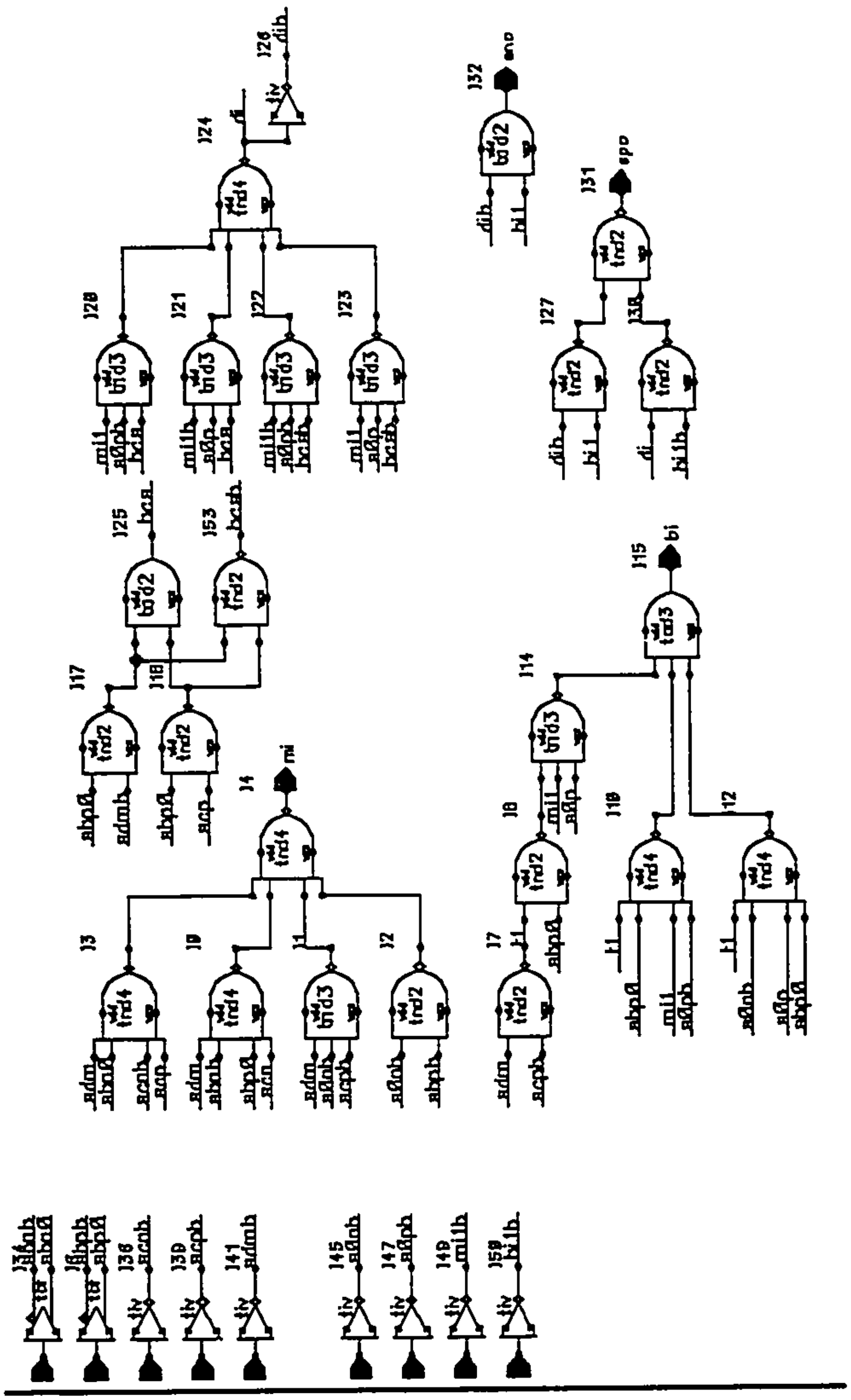


Figure C.2 Type *as* Cell of Improved Architecture

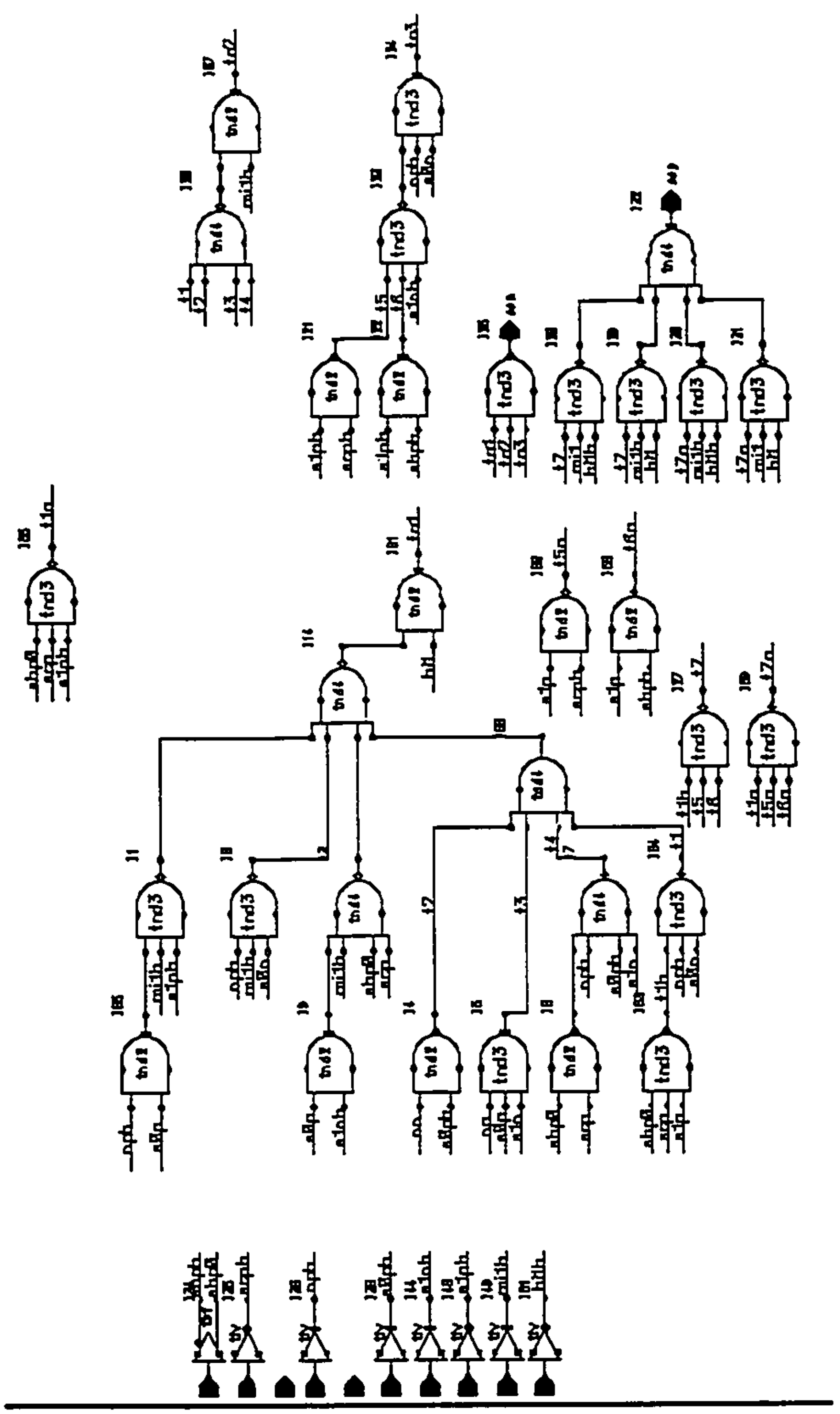


Figure C.3 Type *b* Cell of Improved Architecture

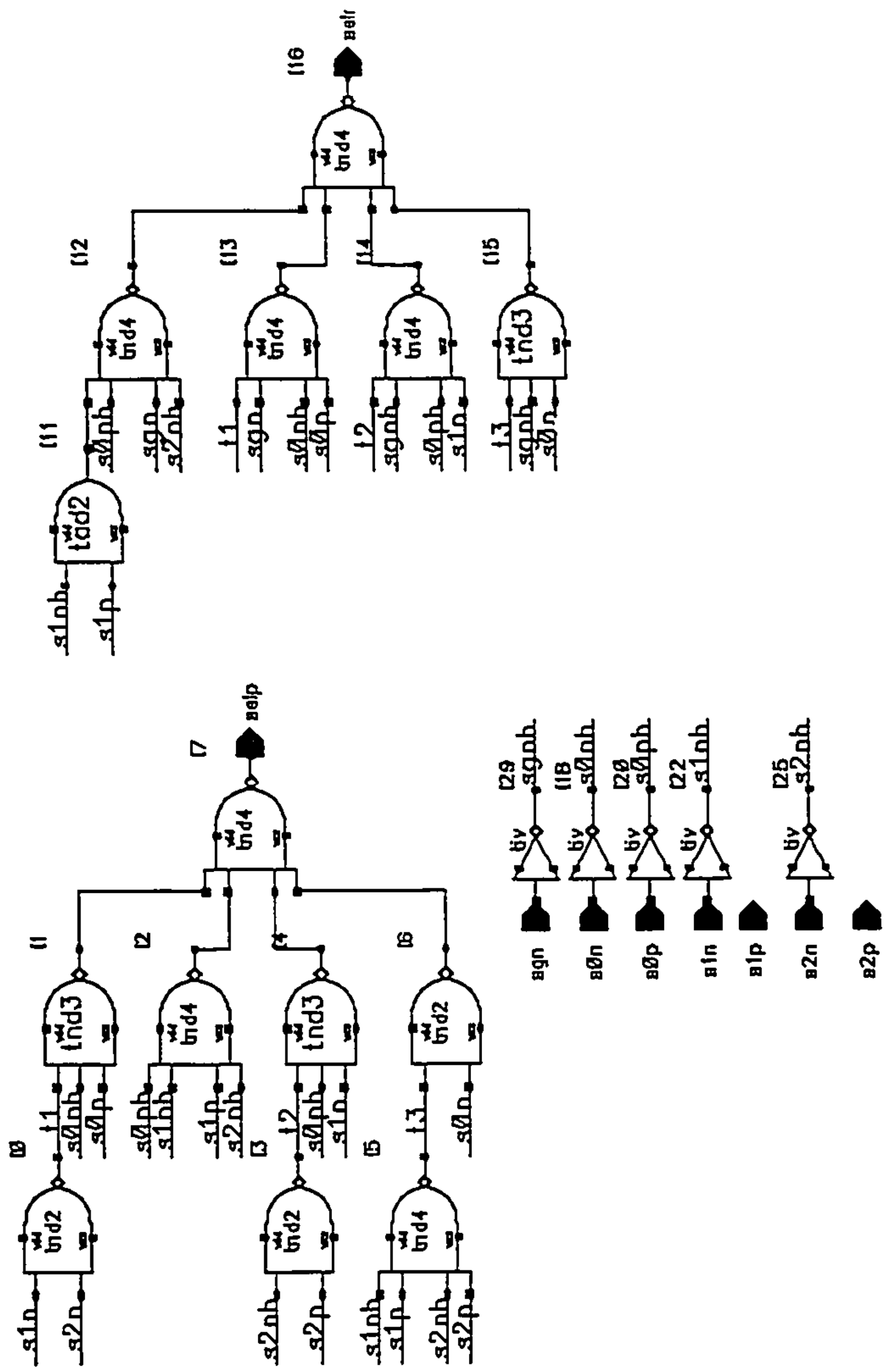


Figure C.4 Type *s* Cell of Improved Architecture

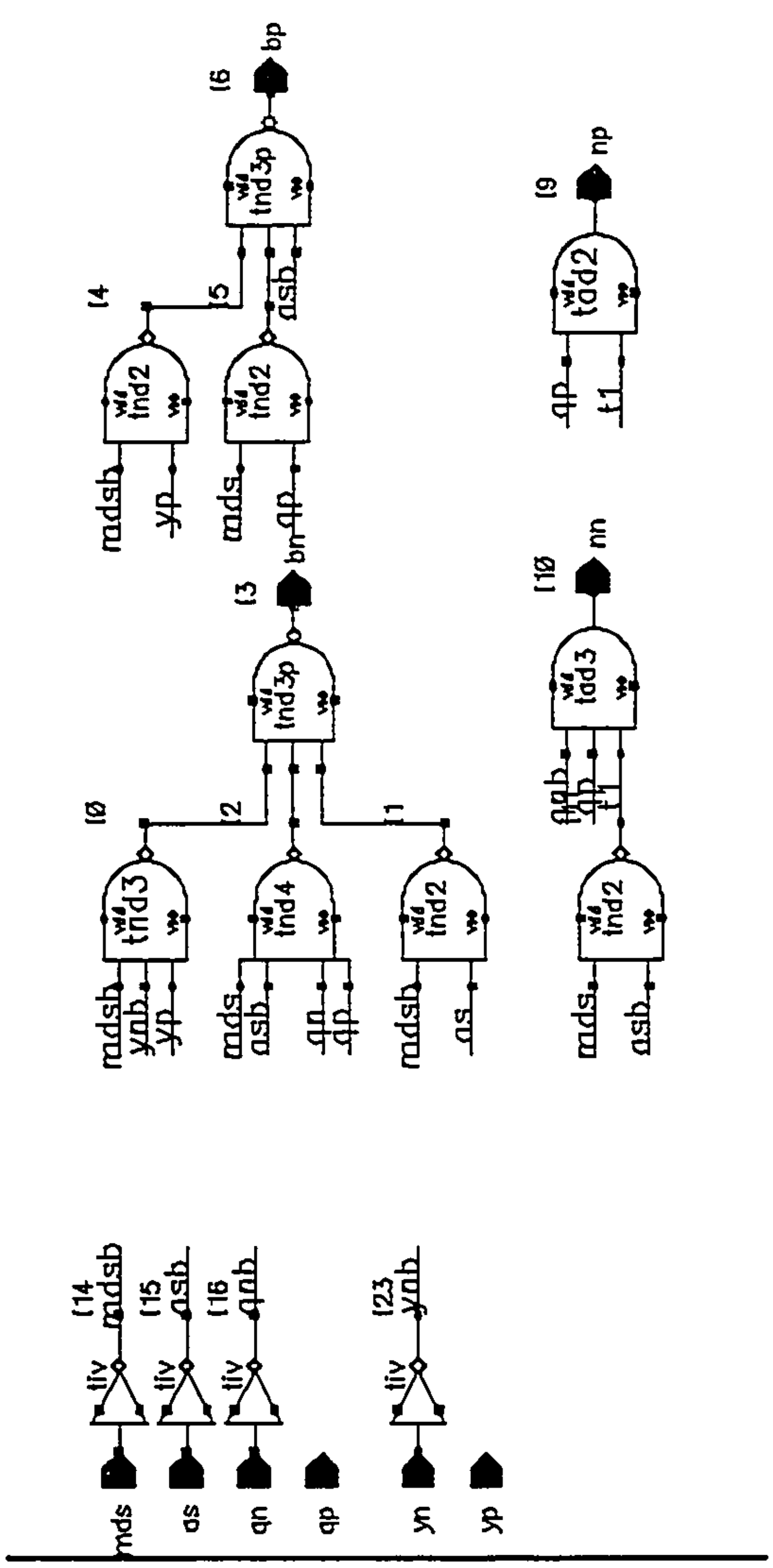


Figure C.5 Type *m* Cell of Improved Architecture

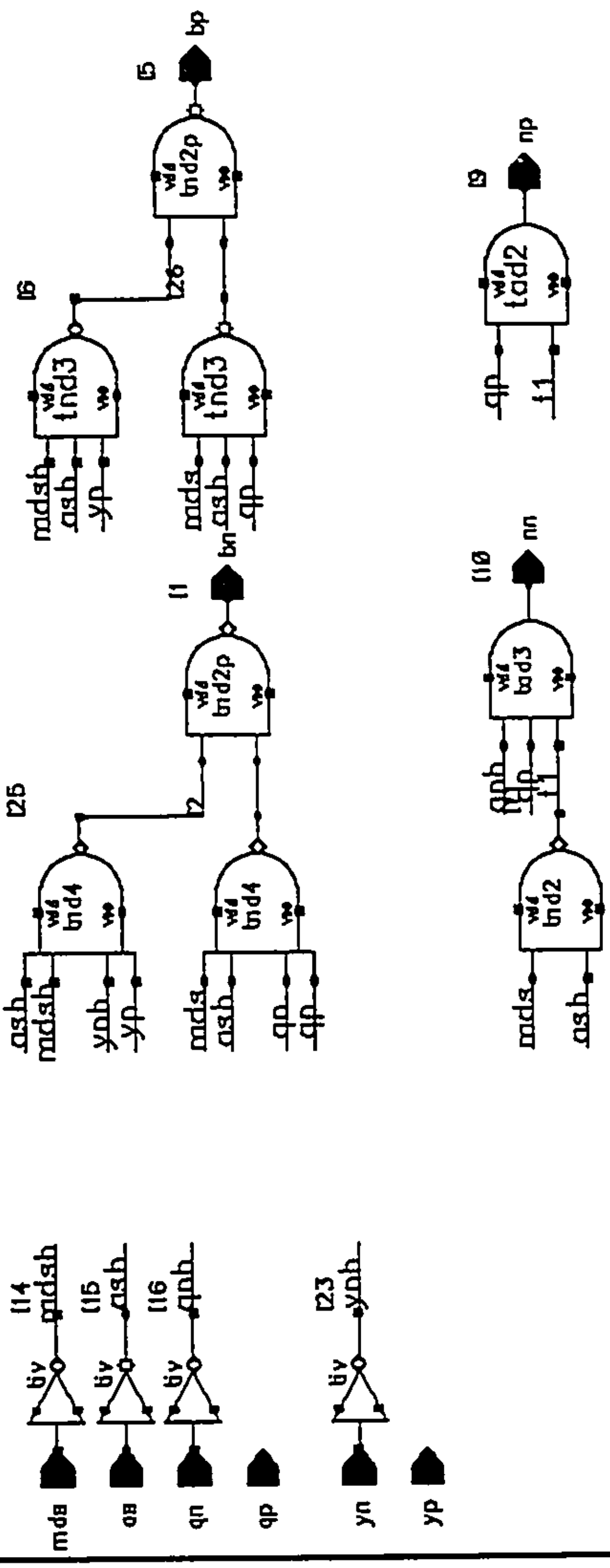


Figure C.6 Type *m*1 Cell of Improved Architecture

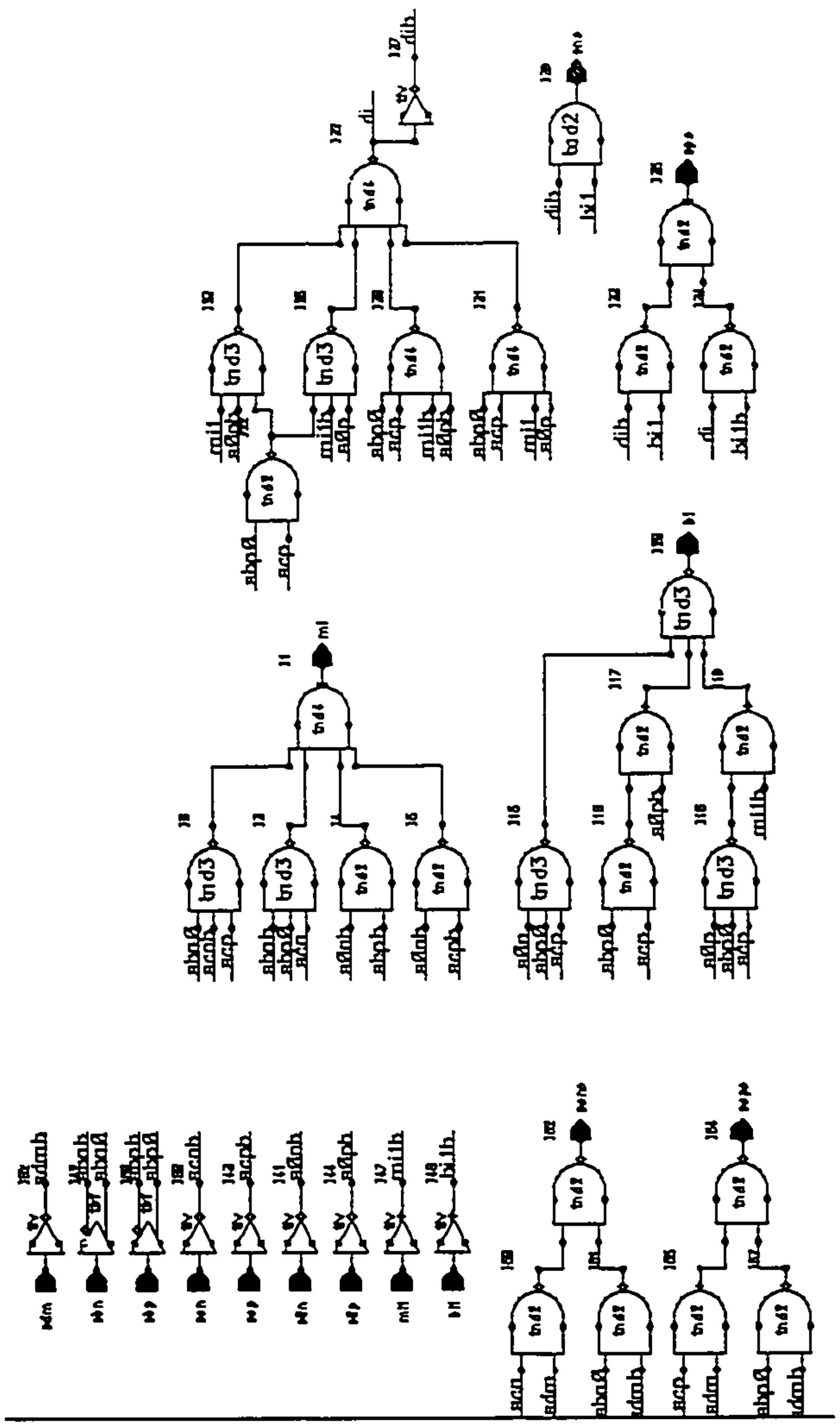


Figure C.7 Type $a^{\#}$ Cell of Improved Architecture

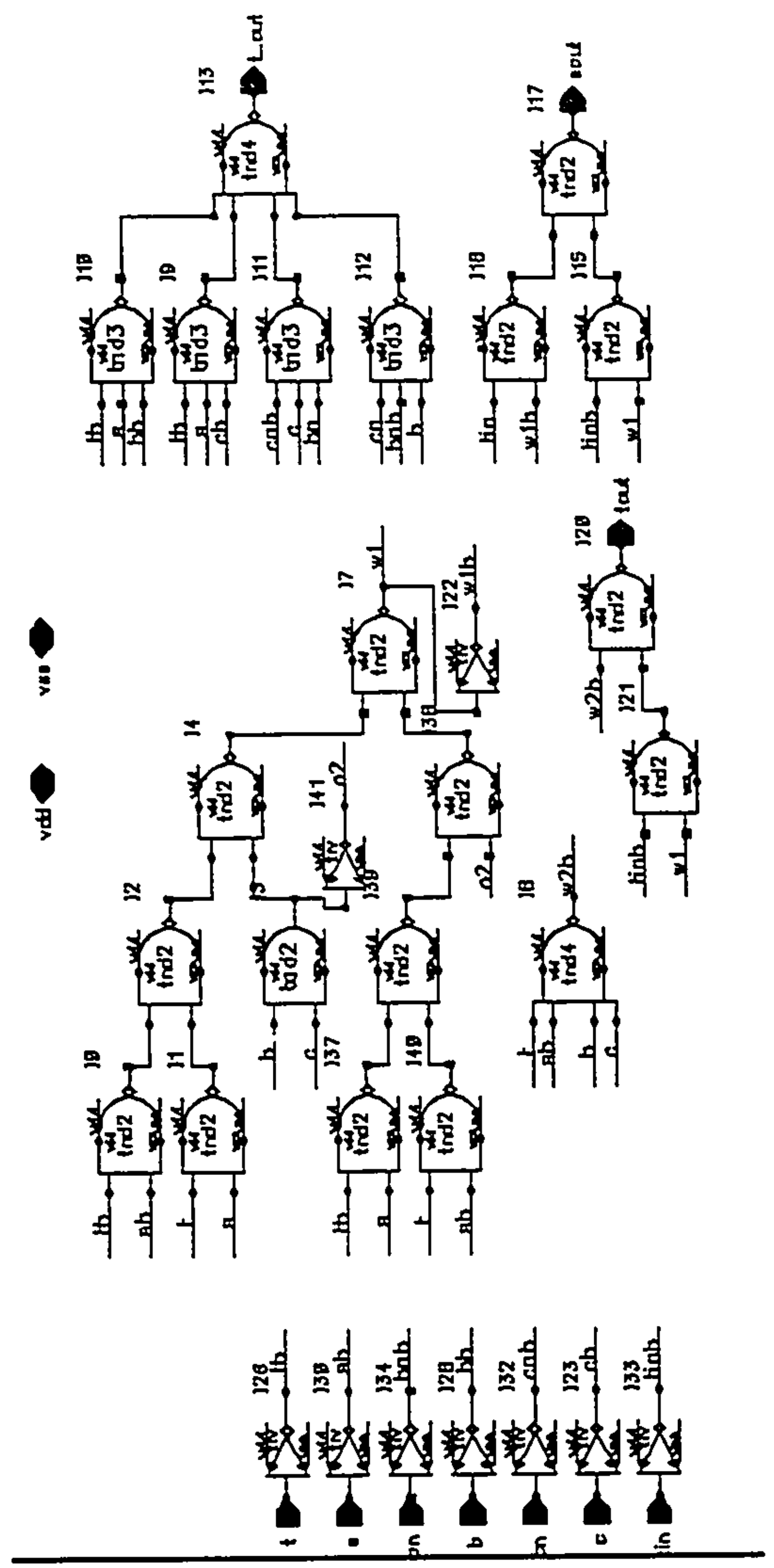


Figure C.8 Type 1 Cell of Original Architecture

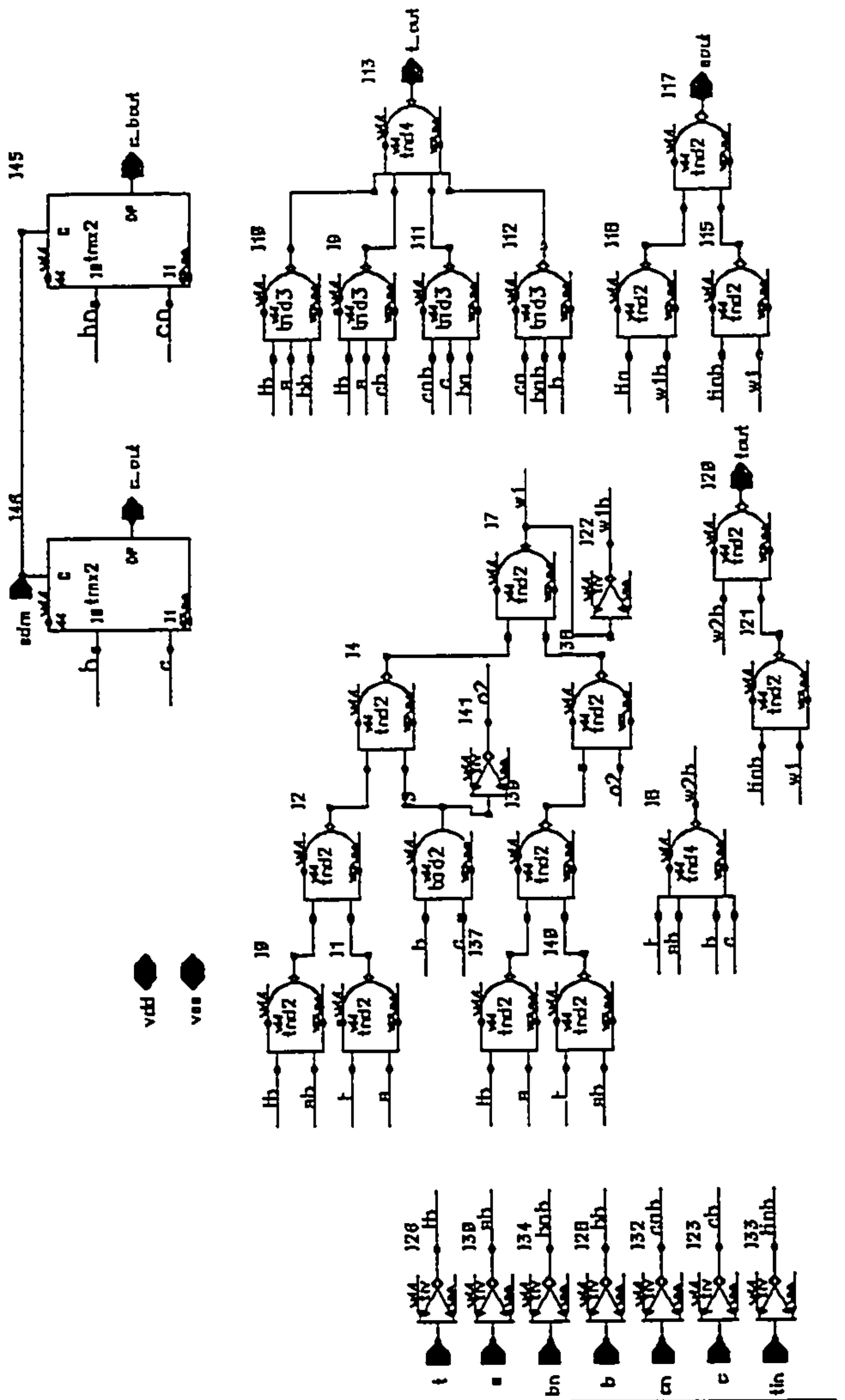


Figure C.9 Type 1* Cell of Original Architecture

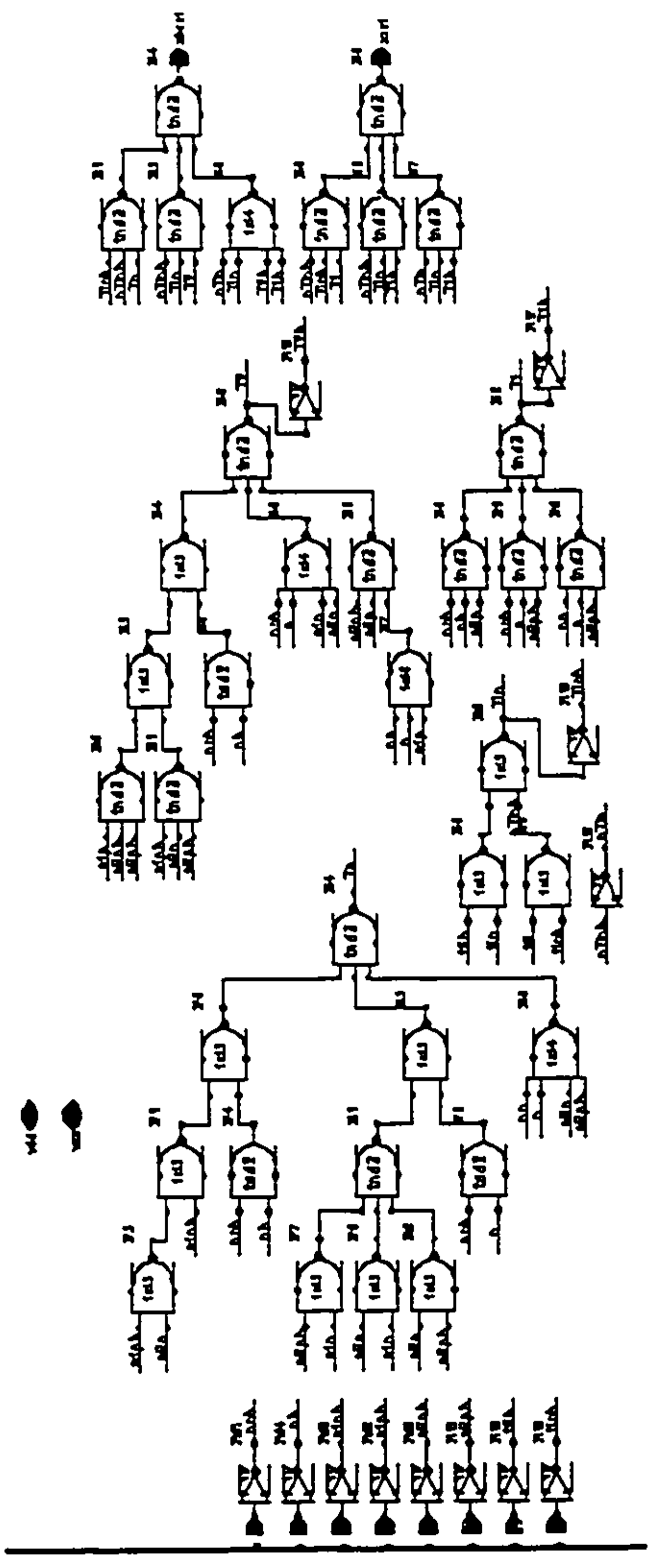


Figure C.10 Type 2 Cell of Original Architecture

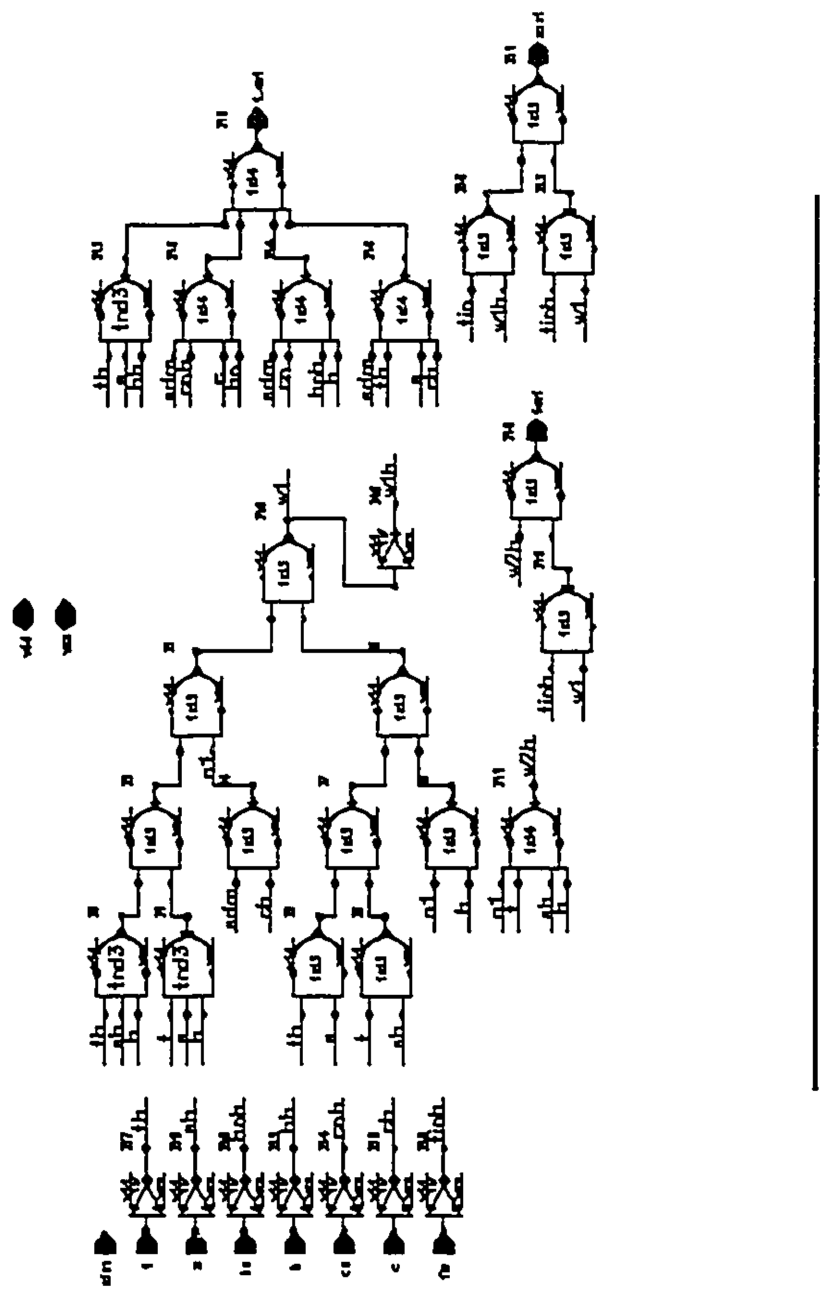


Figure C.11 Type 3 Cell of Original Architecture

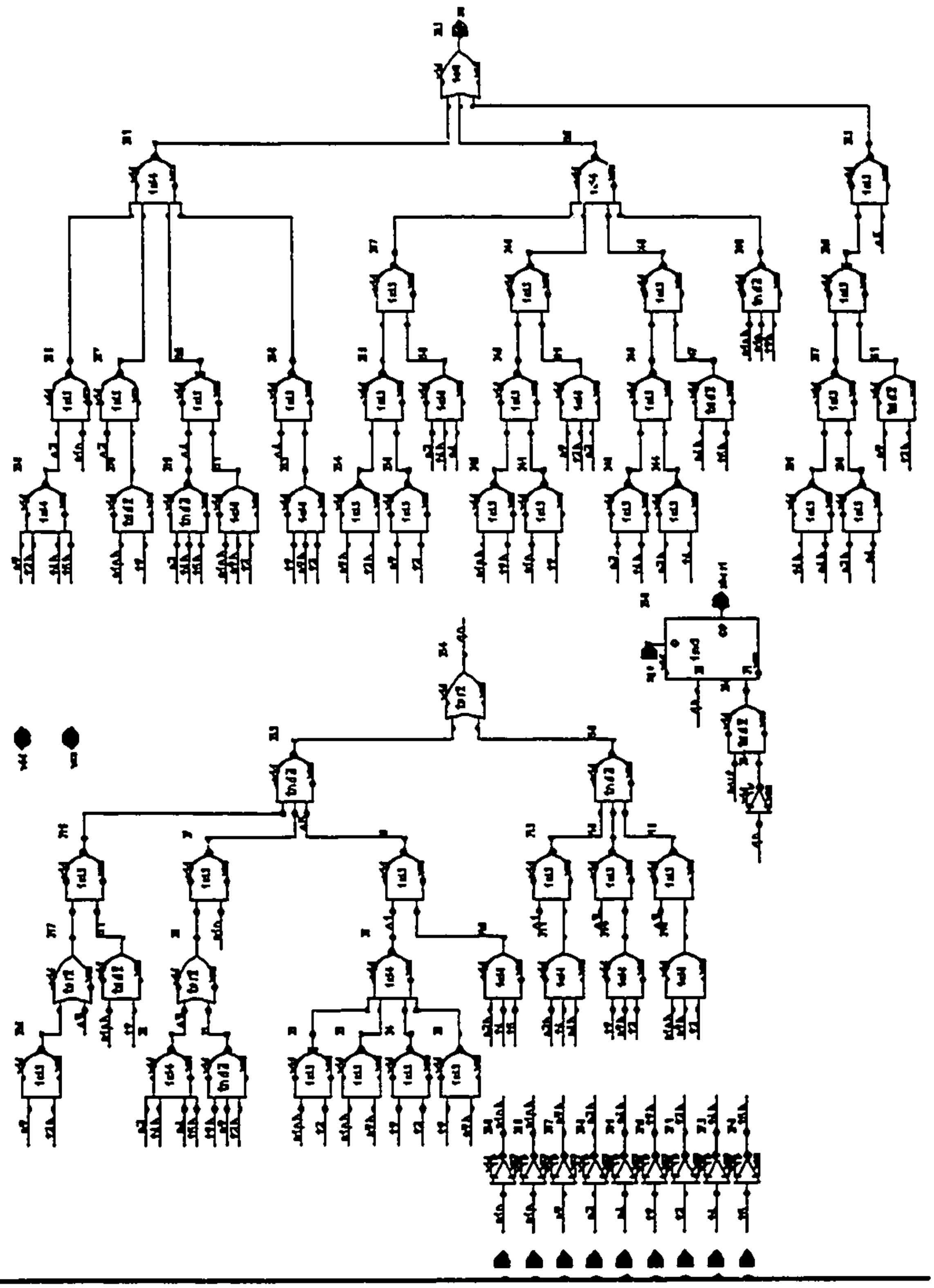


Figure C.12 Type *s* Cell of Original Architecture

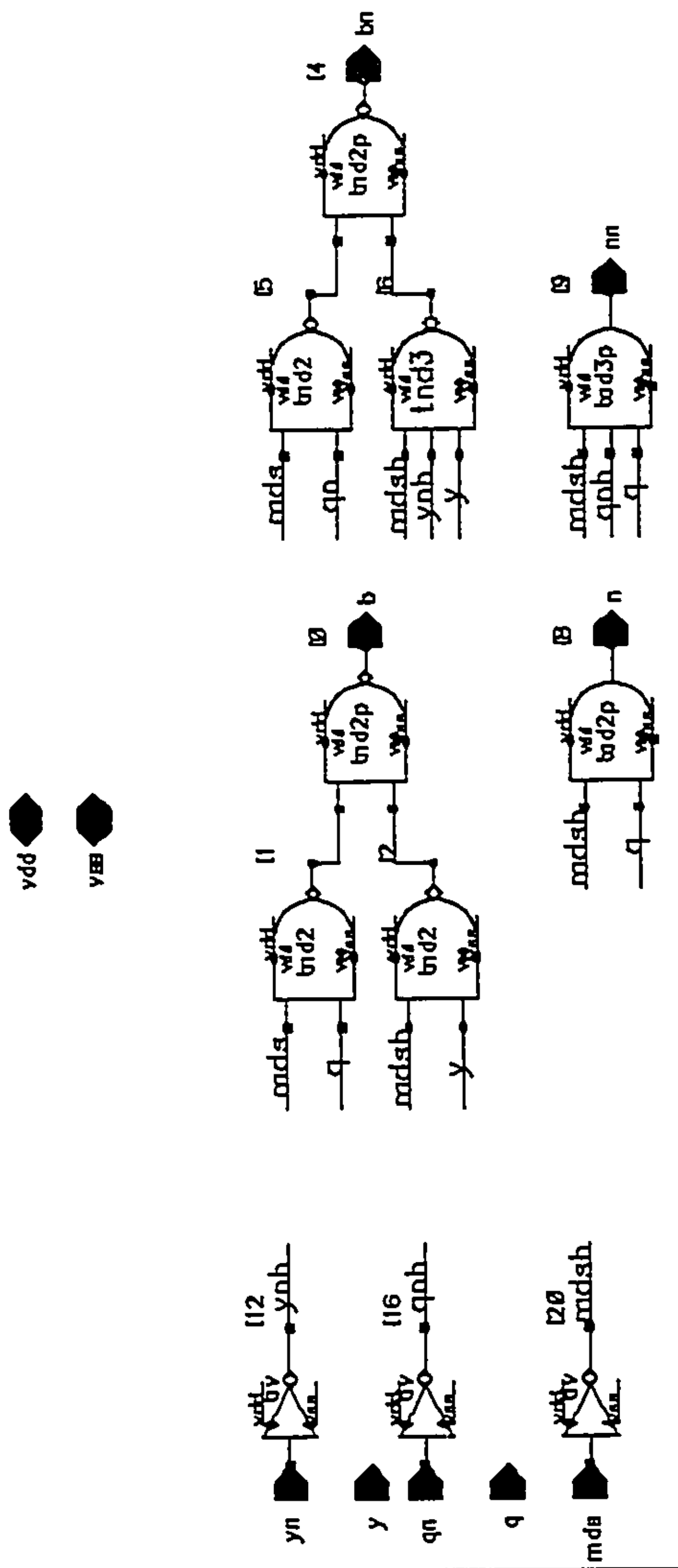


Figure C.13 Type *m* Cell of Original Architecture

# Appendix D

## MASK LAYOUT OF IMPROVED ARCHITECTURE & ORIGINAL ARCHITECTURE

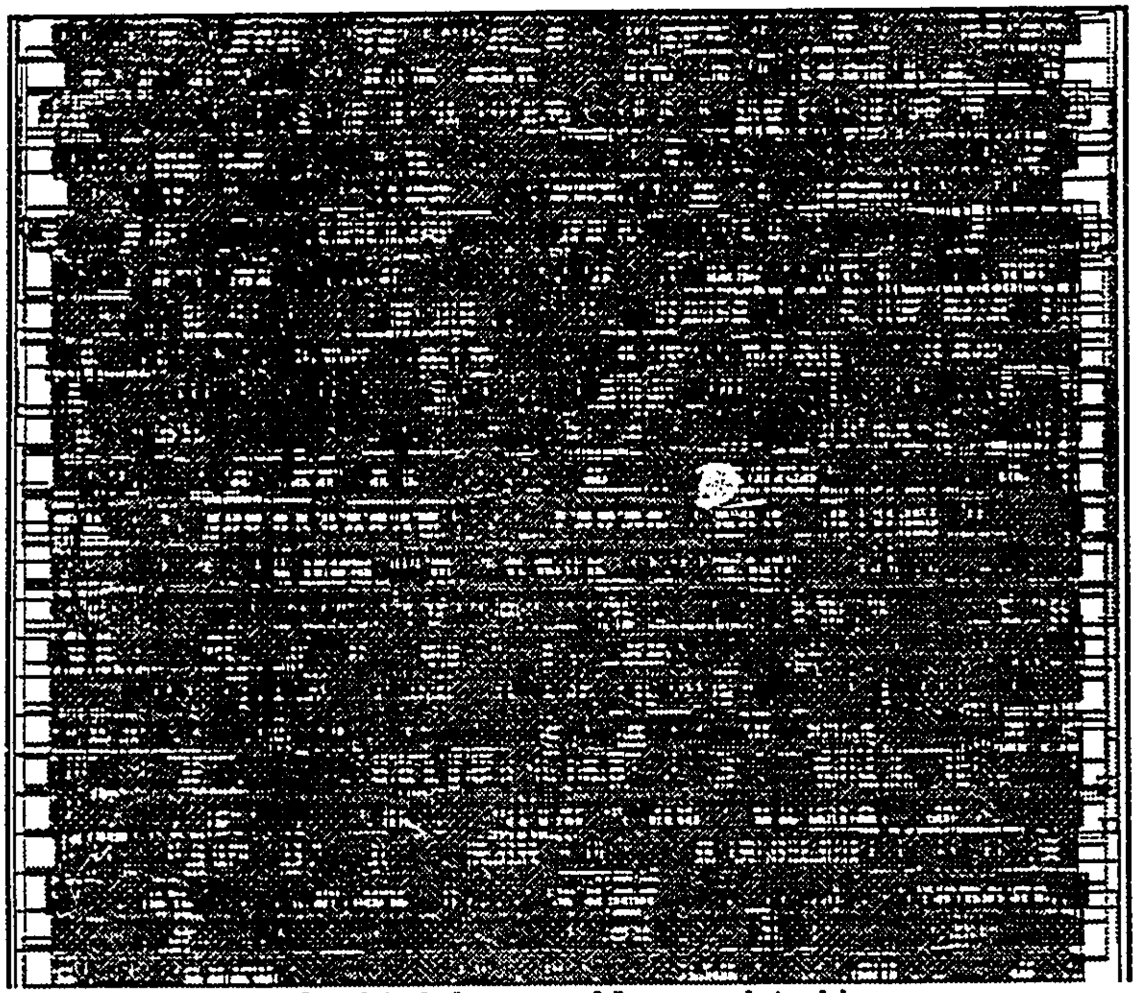


Figure D.1 Mask Layout of Improved Architecture

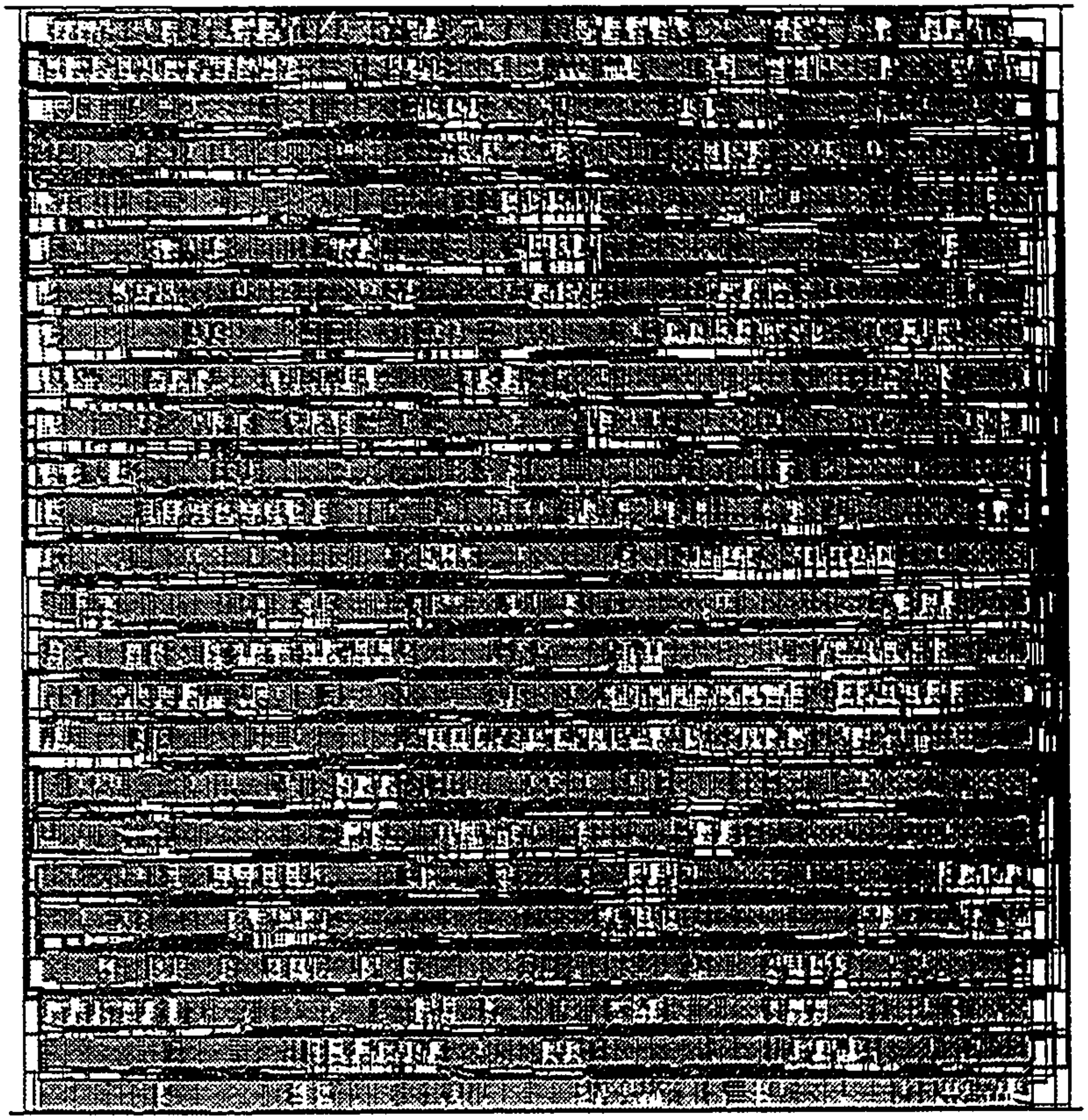


Figure D.2 Mask Layout of Original Architecture

# Appendix E

## VERILOG NETLIST OF IMPROVED ARCHITECTURE

```
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/cella"

// HDL models


// End HDL models


// module cella(bi, mi, sno, spo, bi1, mi1, s0n, s0p, sbn, sbp, scn, scp);
module cella(s0n,s0p,sbn,sbp,scn,scp,mi1,bi1,sno,spo,mi,bi);
output bi, mi, sno, spo;
input bi1, mi1, s0n, s0p, sbn, sbp, scn, scp;
supply1 vdd;
supply0 vss;
and I39(sno, bi1, dib);
not I52(scnb, scn);
not I50(sbnb, sbn);
not I48(bi1b, bi1);
not I47(mi1b, mi1);
not I44(s0pb, s0p);
not I43(scpb, scp);
not I42(sbpb, sbp);
not I41(s0nb, s0n);
not I37(dib, di);
nand I35(spo, hnl_0, hnl_1);
nand I34(hnl_0, bi1b, di);
nand I33(hnl_1, bi1, dib);
nand I22(hnl_2, scp, sbp);
nand I19(hnl_3, mi1b, hnl_4);
nand I17(hnl_5, s0pb, hnl_6);
nand I16(hnl_6, scp, sbp);
nand I5(hnl_7, scpb, s0nb);
nand I4(hnl_8, sbpb, s0nb);
nand I32(di, hnl_9, hnl_10, hnl_11, hnl_12);
nand I31(hnl_9, s0p, mi1, scp, sbp);
nand I30(hnl_10, s0pb, mi1b, scp, sbp);
nand I1(mi, hnl_7, hnl_8, hnl_13, hnl_14);
nand I25(hnl_11, s0p, mi1b, hnl_2);
nand I23(hnl_12, hnl_2, s0pb, mi1);
nand I20(bi, hnl_3, hnl_5, hnl_15);
nand I18(hnl_4, scp, sbp, s0p);
nand I15(hnl_15, scp, sbp, s0n);
nand I3(hnl_13, scn, sbp, sbnb);
nand I0(hnl_14, scp, scnb, sbn);
endmodule

--- cella1 ----
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/cella1"
```

```
// HDL models


// End HDL models


// module cella1(bi, mi, scno, scpo, sno, spo, bi1, mi1, s0n, s0p, sbn, sbp, scn, scp,
sdm);
module cella1(sdm,s0n,s0p,sbn,sbp,scn,scp,mi1,bi1,sno,spo,mi,bi,scno,scpo);
output bi, mi, scno, scpo, sno, spo;
input bi1, mi1, s0n, s0p, sbn, sbp, scn, scp, sdm;
supply1 vdd;
supply0 vss;
and I39(sno, bi1, dib);
not I62(sbpb, sbp);
not I60(scnb, scn);
not I58(sdmb, sdm);
not I48(bi1b, bi1);
not I47(mi1b, mi1);
not I44(s0pb, s0p);
not I43(scpb, scp);
not I42(sbnb, sbn);
not I41(s0nb, s0n);
not I37(dib, di);
nand I57(hnl_0, sdmb, sbp);
nand I55(hnl_1, sdm, scp);
nand I54(scpo, hnl_0, hnl_1);
nand I52(scno, hnl_2, hnl_3);
nand I51(hnl_2, sdmb, sbn);
nand I50(hnl_3, sdm, scn);
nand I35(spo, hnl_4, hnl_5);
nand I34(hnl_4, bi1b, di);
nand I33(hnl_5, bi1, dib);
nand I22(hnl_6, scp, sbp);
nand I19(hnl_7, mi1b, hnl_8);
nand I17(hnl_9, s0pb, hnl_10);
nand I16(hnl_10, scp, sbp);
nand I5(hnl_11, scpb, s0nb);
nand I4(hnl_12, sbpb, s0nb);
nand I32(di, hnl_13, hnl_14, hnl_15, hnl_16);
nand I31(hnl_13, s0p, mi1, scp, sbp);
nand I30(hnl_14, s0pb, mi1b, scp, sbp);
nand I1(mi, hnl_11, hnl_12, hnl_17, hnl_18);
nand I25(hnl_15, s0p, mi1b, hnl_6);
nand I23(hnl_16, hnl_6, s0pb, mi1);
nand I20(bi, hnl_7, hnl_9, hnl_19);
nand I18(hnl_8, scp, sbp, s0p);
nand I15(hnl_19, scp, sbp, s0n);
nand I3(hnl_17, scn, sbp, sbnb);
nand I0(hnl_18, scp, scnb, sbn);
endmodule
```

```
---- cellas -----
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/cellas"

// HDL models


// End HDL models


// module cellas(bi, mi, sno, spo, bi1, mi1, s0n, s0p, sbn, sbp, scn, scp, sdm);
module cellas(sdm,s0n,s0p,sbn,sbp,scn,scp,mi1,bi1,sno,spo,mi,bi);

output bi, mi, sno, spo;
input bi1, mi1, s0n, s0p, sbn, sbp, scn, scp, sdm;
supply0 vss;
supply1 vdd;
and I15(bi, hnl_0, hnl_1, hnl_2);
and I32(sno, bi1, dib);
and I25(bcs, hnl_3, hnl_4);
not I50(bi1b, bi1);
not I49(mi1b, mi1);
not I47(s0pb, s0p);
not I45(s0nb, s0n);
not I41(sdmb, sdm);
not I39(scpb, scp);
not I36(scnb, scn);
not I34(sbnb, sbn);
not I26(dib, di);
not I6(sbpb, sbp);
nand I53(bcsb, hnl_3, hnl_4);
nand I31(spo, hnl_5, hnl_6);
nand I30(hnl_5, bi1b, di);
nand I27(hnl_6, bi1, dib);
nand I18(hnl_3, scp, sbp);
nand I17(hnl_4, sdmb, sbp);
nand I8(hnl_7, sbp, t1);
nand I7(t1, scpb, sdm);
nand I2(hnl_8, sbpb, s0nb);
nand I23(hnl_9, bcsb, s0p, mi1);
nand I22(hnl_10, bcsb, s0pb, mi1b);
nand I21(hnl_11, bcs, s0p, mi1b);
nand I20(hnl_12, bcs, s0pb, mi1);
nand I14(hnl_2, s0p, mi1, hnl_7);
nand I1(hnl_13, scpb, s0nb, sdm);
nand I24(di, hnl_9, hnl_10, hnl_11, hnl_12);
nand I12(hnl_0, sbp, s0p, s0nb, t1);
nand I10(hnl_1, s0pb, mi1, sbp, t1);
nand I4(mi, hnl_8, hnl_13, hnl_14, hnl_15);
nand I3(hnl_15, scp, scnb, sbn, sdm):
```

```
nand I0(hnl_14, scn, sbp, sbnb, sdm);
endmodule

------- comb -------
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/comb"

// HDL models


// End HDL models


// module comb(con, cop, bi1, mi1, nn, np, s0n, s0p, s1n, s1p, sbp, scp);
module comb(nn,np,s0n,s0p,s1n,s1p,sbp,scp,mi1,bi1,con,cop);

output con, cop;
input bi1, mi1, nn, np, s0n, s0p, s1n, s1p, sbp, scp;
supply1 vdd;
supply0 vss;
and I66(hnl_0, t1, t4, t3, t2);
not I51(bi1b, bi1);
not I49(mi1b, mi1);
not I46(s1pb, s1p);
not I44(s1nb, s1n);
not I38(s0pb, s0p);
not I36(npb, np);
not I35(scpb, scp);
not I34(sbpb, sbp);
nand I32(cop, hnl_1, hnl_2, hnl_3, hnl_4);
nand I20(hnl_5, t4, t3, t2, t1);
nand I14(hnl_6, hnl_0, hnl_7, hnl_8, hnl_9);
nand I7(t4, s1n, s0pb, npb, hnl_10);
nand I3(hnl_7, scp, sbp, mi1b, hnl_11);
nand I1(hnl_9, s1pb, mi1b, hnl_12);
nand I64(t1, s0n, npb, t1b);
nand I63(t1b, s1p, scp, sbp);
nand I59(t7a, t6a, t5a, t1a);
nand I55(t1a, s1pb, scp, sbp);
nand I31(hnl_1, bi1, mi1, t7a);
nand I30(hnl_2, bi1b, mi1b, t7a);
nand I29(hnl_3, bi1, mi1b, t7);
nand I28(hnl_4, bi1b, mi1, t7);
nand I27(t7, t6, t5, t1b);
nand I25(con, to3, to2, to1);
nand I24(to3, s0n, npb, hnl_13);
nand I23(hnl_13, s1nb, t6, t5);
nand I8(hnl_8, s0n, mi1b, npb);
nand I5(t3, s1n, s0p, np);
nand I67(to2, mi1b, hnl_5);
nand I65(hnl_12, s0p, npb);
nand I61(to1, bi1, hnl_6);
nand I60(t5a, scpb, s1p);
nand I58(t6a, sbpb, s1p);
nand I22(t6, sbpb, s1pb);
nand I21(t5, scpb, s1pb);
nand I6(hnl_10, scp, sbp);
nand I9(hnl_11, s1nb, s0p);
nand I4(t2, s0pb, nn);
endmodule

------ mcell1 ------
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/mcell"

// HDL models


// End HDL models


// module mcell1(bn, bp, nn, np, as, mds, qn, qp, yn, yp);
module mcell1(mds,as,qn,qp,yn,yp,bn,bp,nn,np);
output bn, bp, nn, np;
input as, mds, qn, qp, yn, yp;
supply1 vdd;
supply0 vss;
not I23(ynb, yn);
not I16(qnb, qn);
not I15(asb, as);
not I14(mdsb, mds);
and I10(nn, t1, qp, qnb);
and I9(np, t1, qp);
nand I2(hnl_0, qp, qn, asb, mds);
nand I11(t1, asb, mds);
nand I5(hnl_1, qp, mds);
nand I4(hnl_2, yp, mdsb);
nand I1(hnl_3, as, mdsb);
nand I6(bp, asb, hnl_1, hnl_2);
nand I3(bn, hnl_3, hnl_0, hnl_4);
nand I0(hnl_4, yp, ynb, mdsb);
endmodule

------ mcell ------
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/mcell1"

// HDL models


// End HDL models
// module mcell(bn, bp, nn, np, as, mds, qn, qp, yn, yp);
module mcell(mds,as,qn,qp,yn,yp,bn,bp,nn,np);

output bn, bp, nn, np;
input as, mds, qn, qp, yn, yp;
supply1 vdd;
supply0 vss;
not I23(ynb, yn);
not I16(qnb, qn);
not I15(asb, as);
not I14(mdsb, mds);
and I10(nn, t1, qp, qnb);
and I9(np, t1, qp);
nand I25(hnl_0, yp, ynb, mdsb, asb);
nand I2(hnl_1, qp, qn, asb, mds);
nand I11(t1, asb, mds);
nand I5(bp, hnl_2, hnl_3);
nand I1(bn, hnl_1, hnl_0);
nand I26(hnl_2, qp, asb, mds);
nand I6(hnl_3, yp, asb, mdsb);
endmodule

------ select ------
// Verilog netlist of
//"/home/engn/home5/hchan/batmos/Work/NCA/cells/select"

// HDL models


// End HDL models


// module select(seln, selp, s0n, s0p, s1n, s1p, s2n, s2p, sgn);
module select(sgn,s0n,s0p,s1n,s1p,s2n,s2p,seln,selp);

output seln, selp;
input s0n, s0p, s1n, s1p, s2n, s2p, sgn;
supply1 vdd;
supply0 vss;
not I29(sgnb, sgn);
not I25(s2nb, s2n);
not I22(s1nb, s1n);
not I20(s0pb, s0p);
not I18(s0nb, s0n);
and I11(hnl_0, s1p, s1nb);
nand I16(seln, hnl_1, hnl_2, hnl_3, hnl_4);
nand I14(hnl_2, s1n, s0pb, sgnb, t2);
nand I13(hnl_3, s0p, s0nb, sgn, t1);
nand I12(hnl_4, s2nb, sgn, s0pb, hnl_0);
nand I7(selp, hnl_5, hnl_6, hnl_7, hnl_8);
nand I5(t3, s2p, s2nb, s1p, s1nb);
nand I2(hnl_7, s2nb, s1p, s1nb, s0pb);
nand I15(hnl_1, s0n, sgnb, t3);
nand I4(hnl_6, s1n, s0pb, t2);
nand I1(hnl_8, s0p, s0nb, t1);
nand I6(hnl_5, s0n, t3);
nand I3(t2, s2p, s2nb);
nand I0(t1, s2n, s1n);
endmodule
```

# Appendix F

## SCHEMATIC DIAGRAM OF IMPROVED ARCHITECTURE USING SWITCHING TREE TECHNIQUE

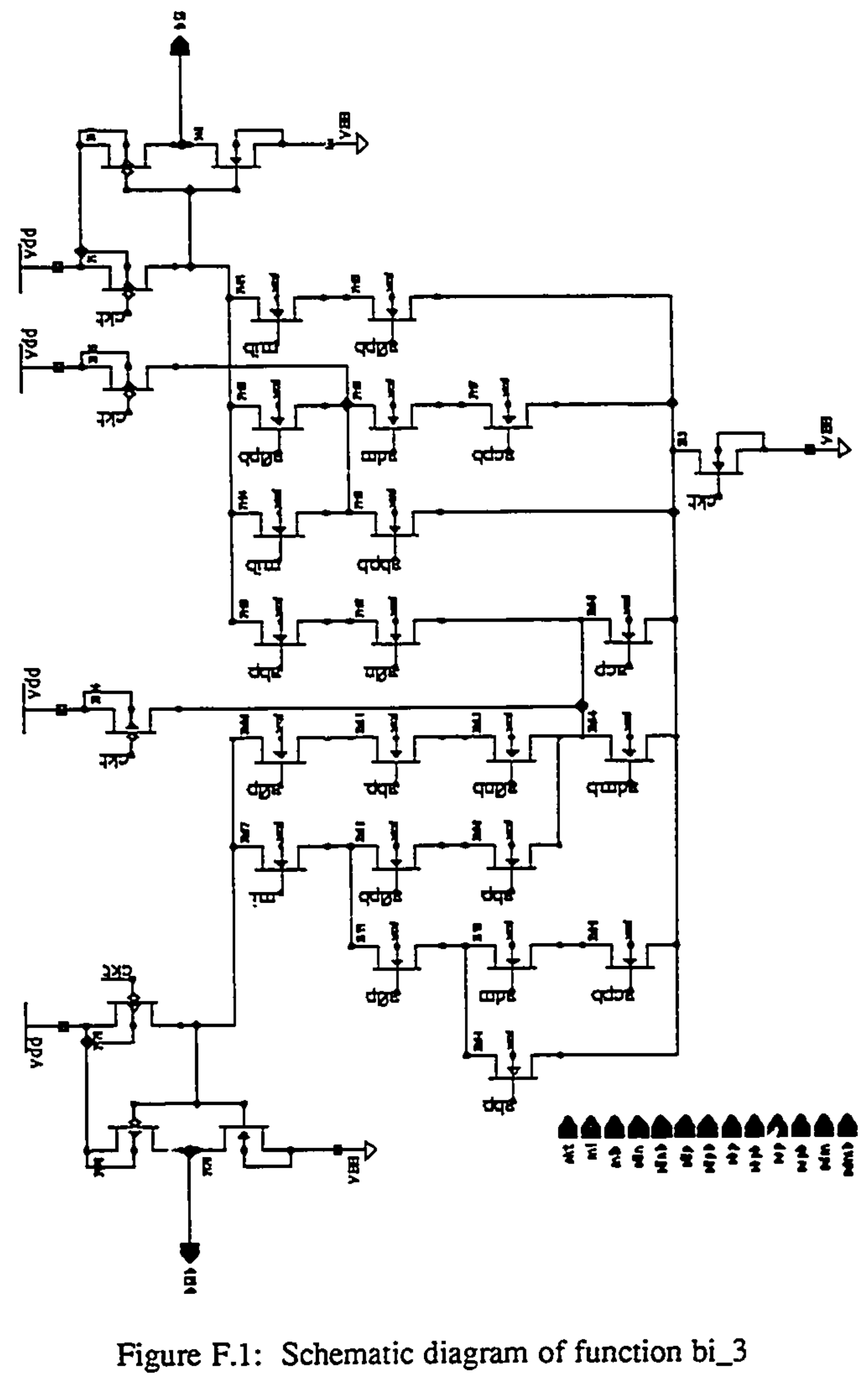


Figure F.1: Schematic diagram of function bi_3

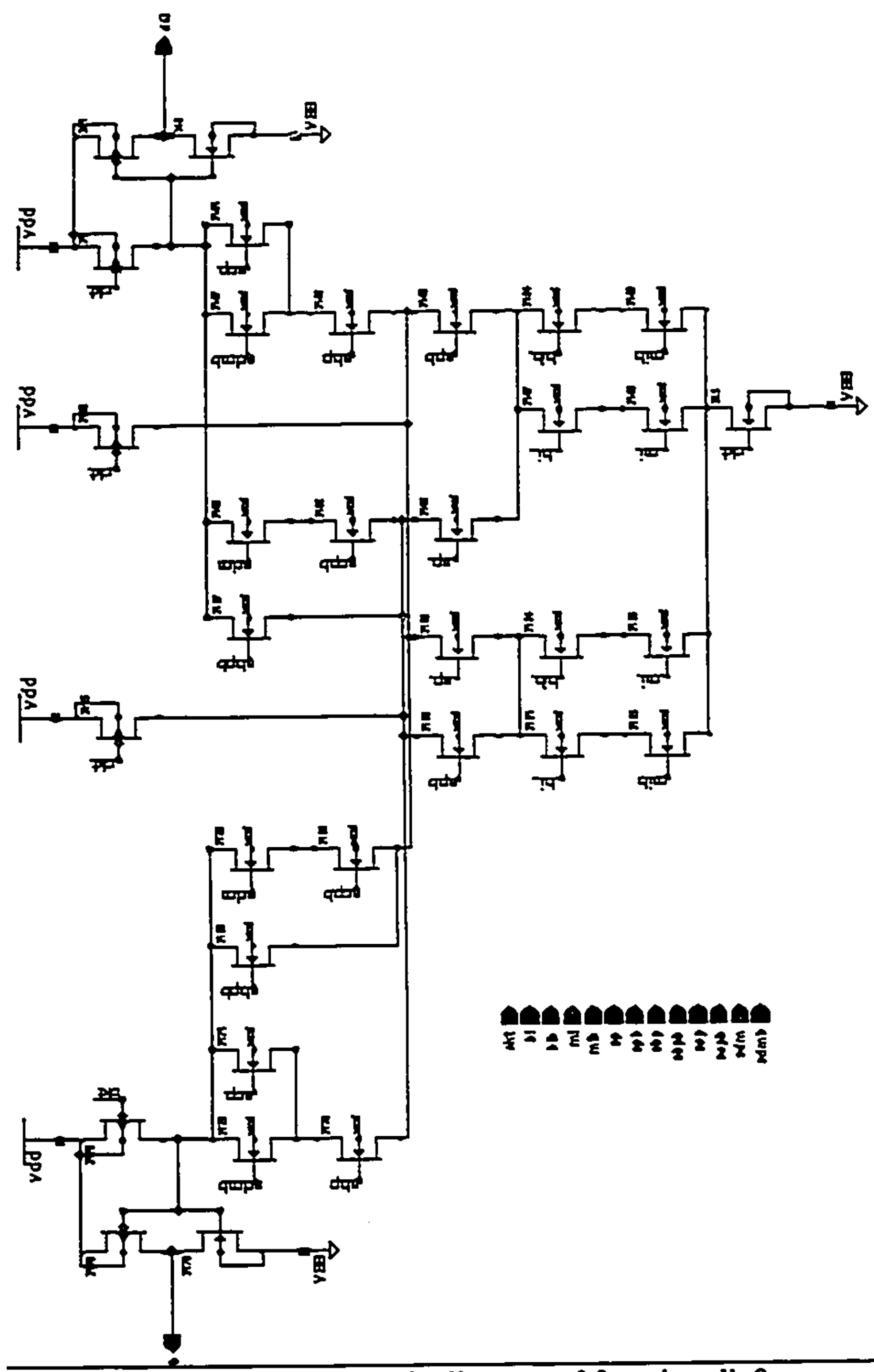


Figure F.2: Schematic diagram of function di_3

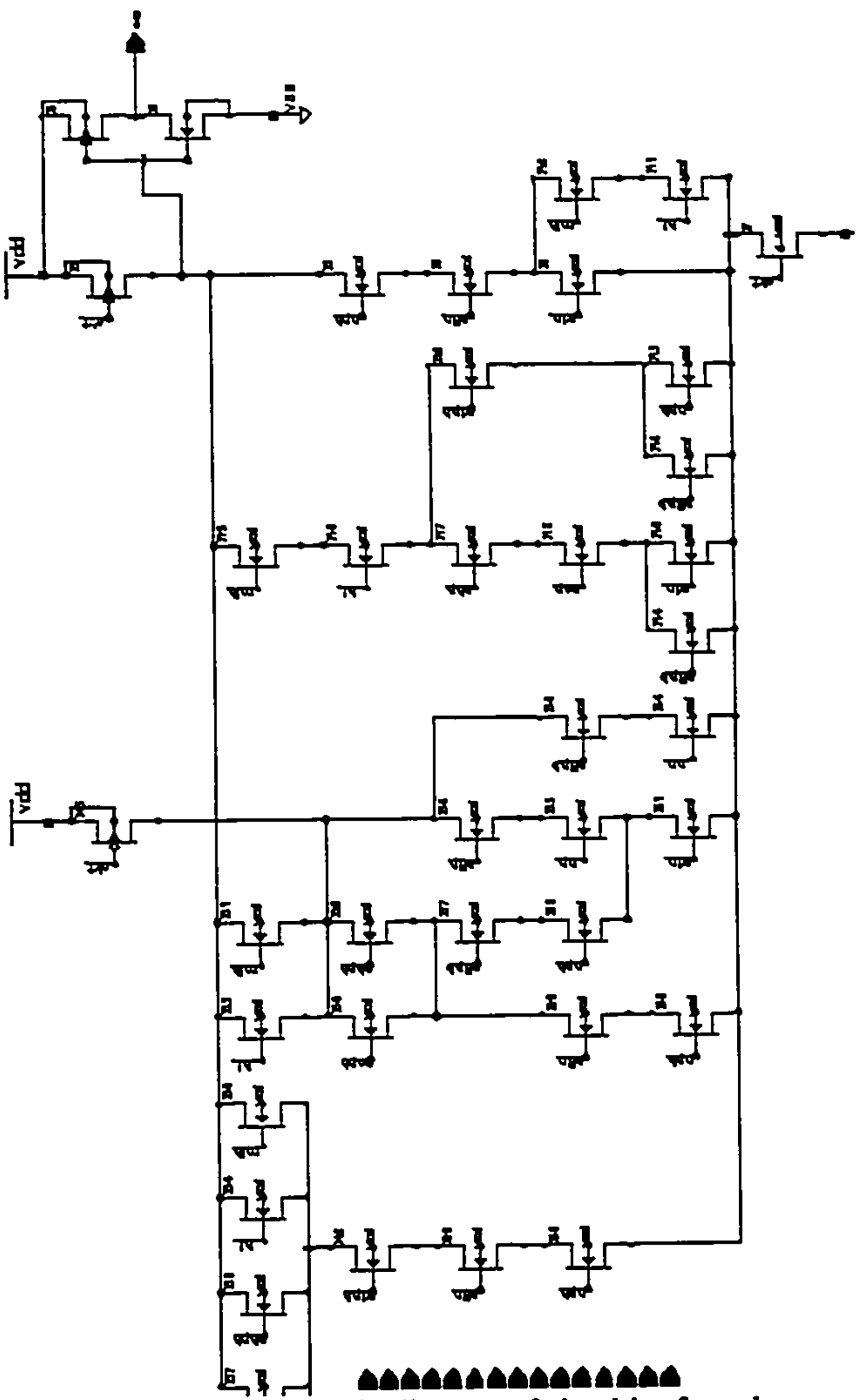


Figure F.3: Schematic diagram of sign bit of comb

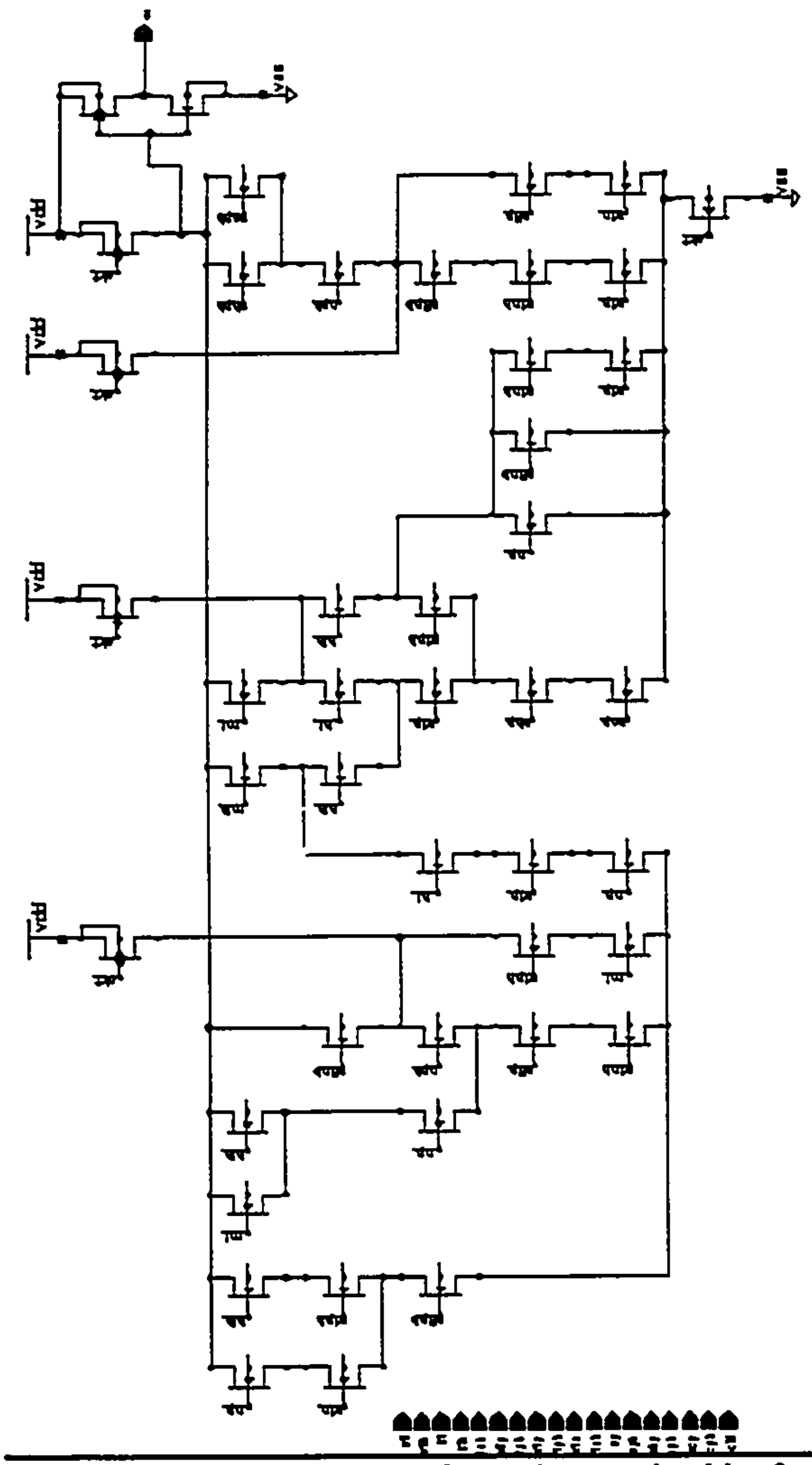


Figure F.4: Schematic diagram of complement sign bit of comb

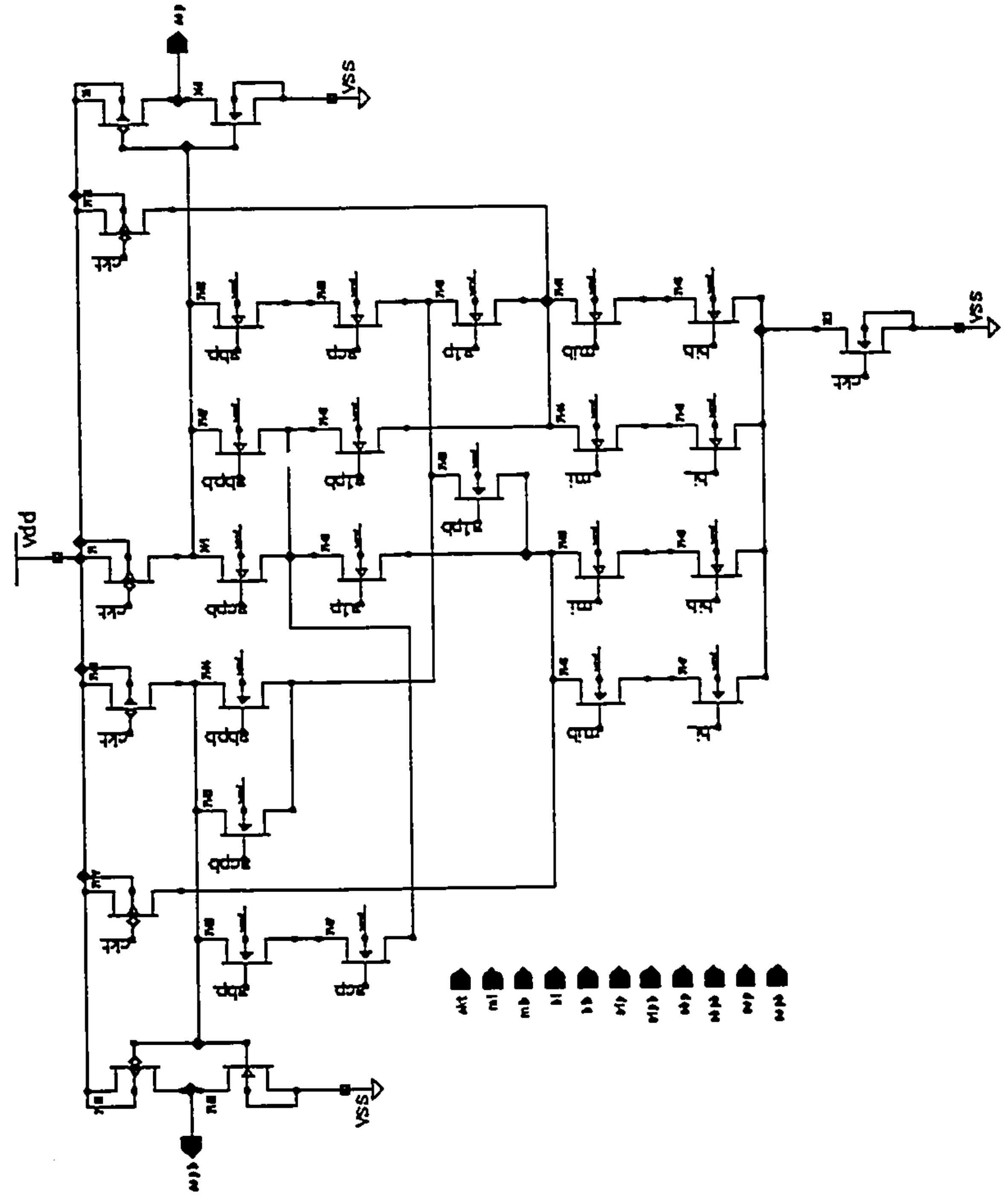


Figure F.5: Schematic diagram of magnitude bit of comb

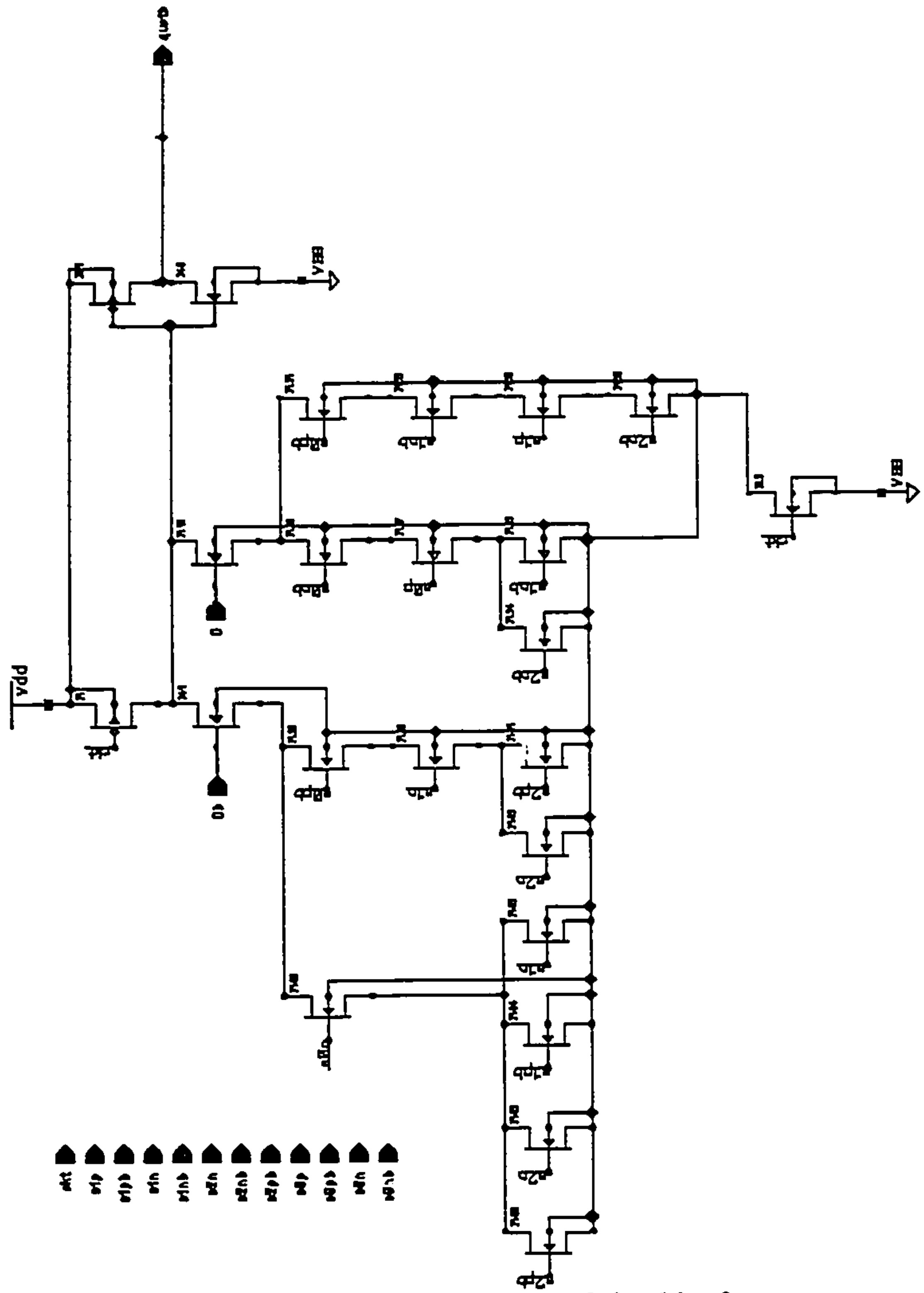


Figure F.6: Schematic diagram of sign bit of quot

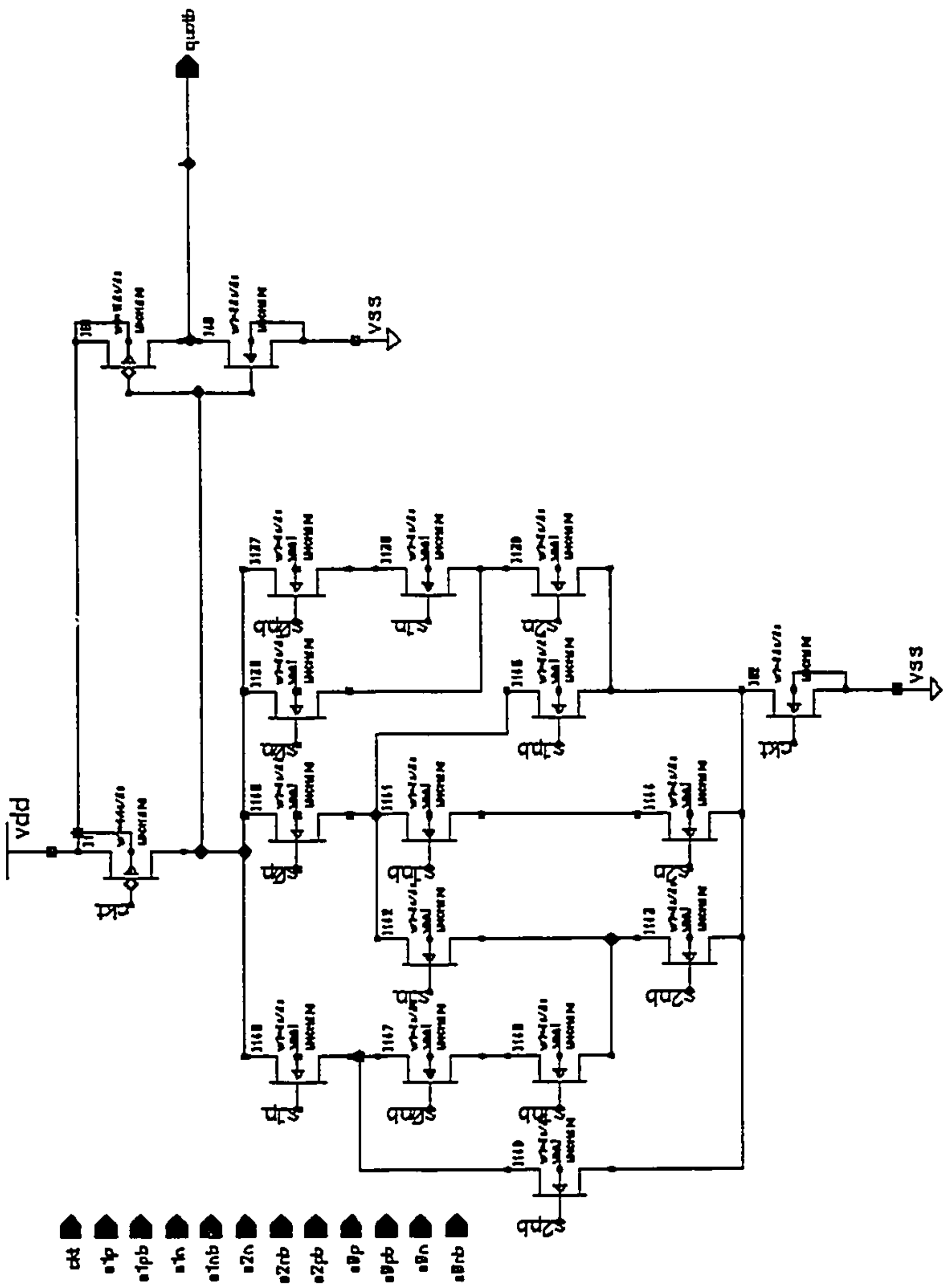


Figure F.7: Schematic diagram of magnitude bit of quot

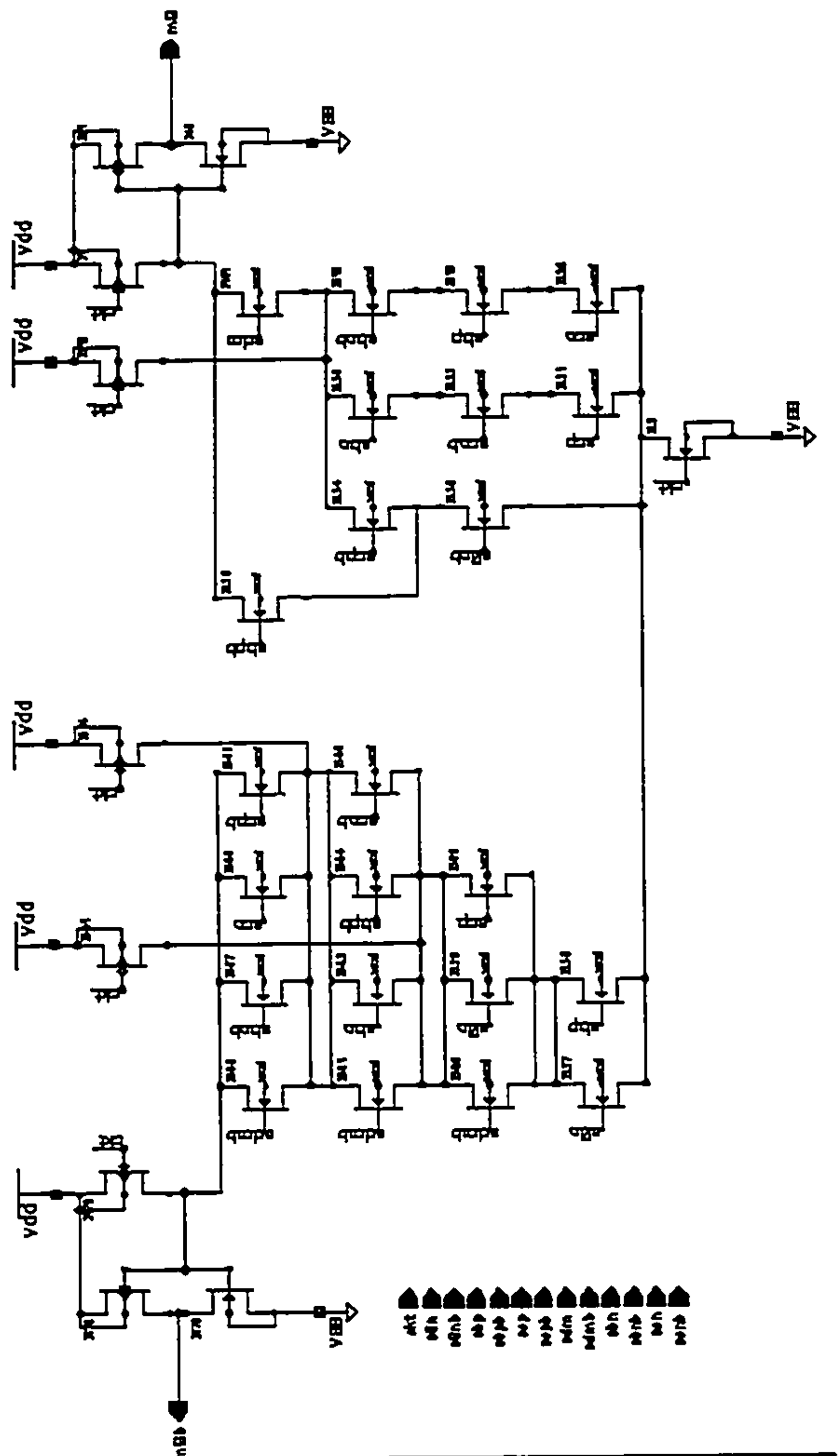


Figure F.8: Schematic diagram of function mi_3

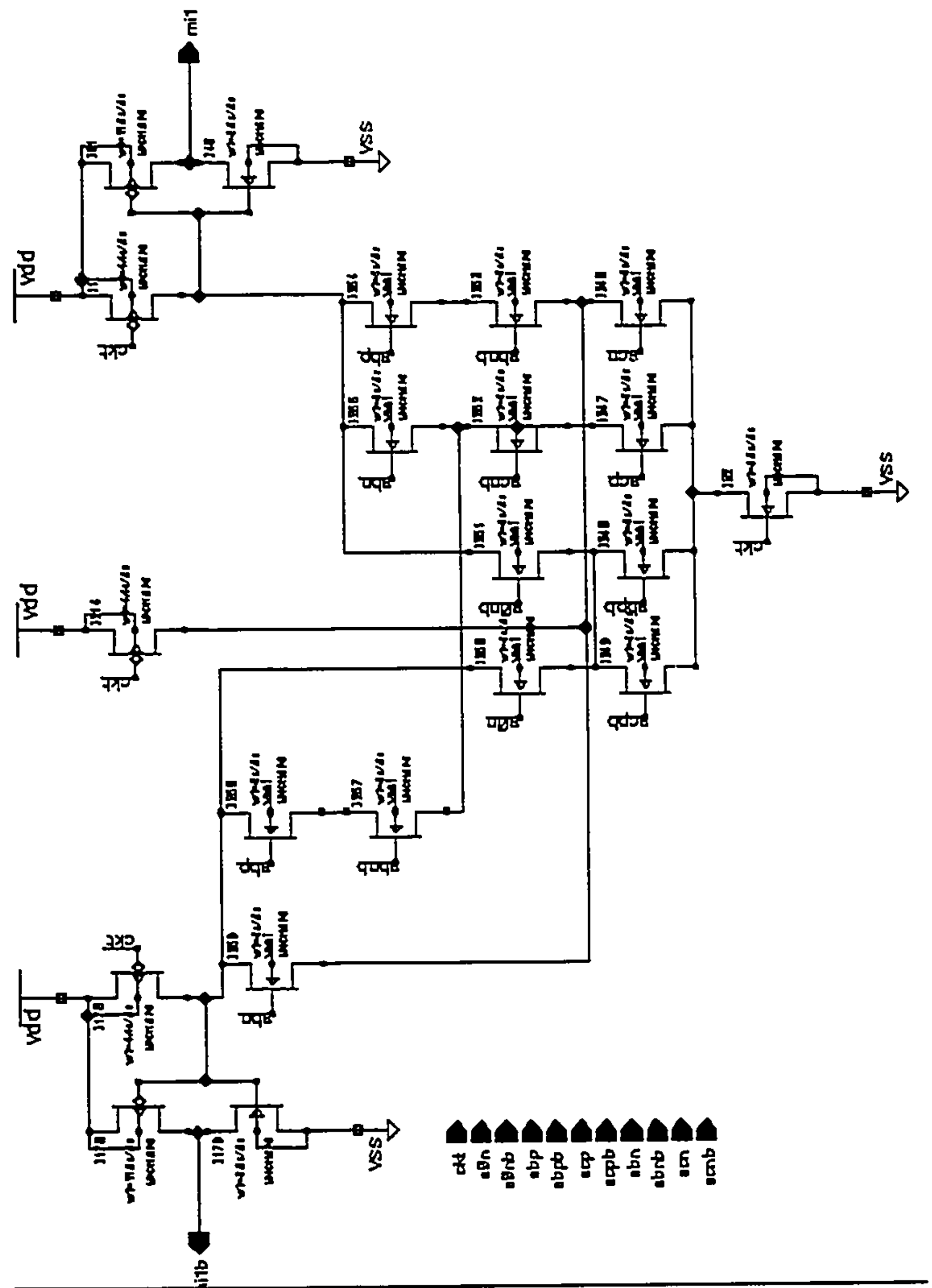


Figure F.9: Schematic diagram of function mi_1

# VITA AUCTORIS

|  |  |
|---|---|
| NAME: | Henry Hin Hai Chan |
| PLACE OF BIRTH: | Singapore |
| YEAR OF BIRTH: | 1966 |
| EDUCATION: | Hill Park High School, Hamilton, Ontario<br>1984-1986 |
|  | University of Windsor, Windsor, Ontario<br>1986-1990, B.A.Sc (Electrical Engineering) |
|  | University of Windsor, Windsor, Ontario<br>1990-1993, M.A.Sc (Electrical Engineering) |