HARDWARE ONLINE MULTIPLICATION-DIVISION: A DESIGN AND PERFORMANCE STUDY

BY

Abdul-Rahman Moustafa Elshafei

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

JUNE, 2009

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SUADI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis written by **ABDUL-RAHMAN MOUSTAFA ELSHAFEI** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.

Thesis Committee

Alaa Amin

Dr. Alaaeldin Amin (Thesis Advisor)

[illegible]

Dr. Adnan Gutub (Member)

[illegible] 17 JUN 2009

Dr. Talal M. Alkharobi (Member)

[illegible]

Department Chairman
Dr. Adnan Gutub

[illegible]

Dean of Graduate Studies
Dr. Salam A. Zummo

8/7/09

Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

Dedication

Dedicated

to

My Beloved Parents

Acknowledgment

All praise are due to Allah first and foremost for his countless blessings. Acknowledgment is due to King Fahd University of Petroleum & Minerals for supporting this research.

My unrestrained appreciation goes to my advisor, Dr. Alaaeldin Amin, for all the assistance, advice, motivation and invaluable support he has given me throughout the course of this work and on several other occasions. I also wish to thank my thesis committee members, Dr. Adnan Gutub and Dr. Talal Alkharobi, for their cooperation, support and critical review of the thesis.

.

Table of Contents

List of Figures

Thesis Abstract

NAME: Abdul-Rahman Moustafa Elshafei

TITLE: Hardware Online Multiplication-Division: A Design and Performance Study.

MAJOR FIELD: Computer Engineering.

DATE OF DEGREE: June 2009.

Digit-serial online arithmetic schemes are highly attractive since they allow successive processes to be computed in an overlapping manner. This results in high throughput operation with simple communication interface and reduced area overhead. Despite numerous works which target the development of online techniques for digital filters, none of these addressed the case of adaptive filters. In modern communication systems, adaptive filters play important role in the fields of equalization, noise cancellation, linear predication and system identification. In addition to the multiply-accumulate operations, many adaptive filter algorithms require a series of multiplication and division operations. Online implementation of these adaptive filters may not be feasible using cascading of conventional online multiplication and online division units due to the large hardware overhead and latency requirements. Fast implementations of these online multiplication and division operations at low area overhead are crucial for high-speed telecommunication networks. To date there has been no reported algorithms or hardware implementations for a unified online multiplication-division unit. In this thesis, two novel algorithms for online multiplication-division are introduced; Online Multiplication-Division with Constant Divisor, and Online Multiplication-Division Algorithms with full online inputs and output. The performance of both algorithms is studied and compared to a conventional sequence of independent online multiplication and division operations. Synthesis results are reported to compare area and delay aspects of these approaches.

خلاصة الرسالة

الإســـــم : عبد الرحمن مصطفى الشافعي

عنوان الرسالة : تصميم ودراسة أداء اجهزة عملية الضرب+القسم التسلسلي (online)

التخصص : هندسة الحاسب الآلي

تاريخ التخرج : جمادى الثاني 1430

الأساليب العددية الرقمية المتتابعة الحية (online) شائعة جداً نظراً لكونها تسمح بحساب العمليات المتتابعة بشكل متزامن. مما يؤدي إلى ارتفاع الحصيلة الإنتاجية وتقليل المساحة وزيادة كفاءة التواصل بين الأطـــراف. علـــى الرغم من تعدد الجهود والبحوث لتطوير التقنيات الحية للمصفيات الرقمية، لم تعتني أي من هذه الابحـــاث بدراســـة المصفيات المتكيفة والتي تلعب دوراً فعالا في مجالات علمية متعددة. وتتطلب المصفيات المتكيفة – بالإضافة إلى عمليت الضرب التراكمي – سلسلة منن عمليات الضرب والقسمة. الحلول الحية لهذه المصفيات المتكيفة باســـتخدام تـــراكم عمليات القسمة الحية واالضرب الحي قد لا تمثل حلا عملياً نظرا لارتفاع تكلفتها الناتج عن التأخير الـــزمني وزيـــادة المساحة. ولهذا فالحلول السريعة قليلة التكلفة لعمليات الضرب والقسمة الحية تعتـــبر ذات أهميـــة فائقـــة لشـــبكات الاتصالات فائقة السرعة. حتى اليوم، تشير التقارير إلى عدم تواجد أي خوارزميات أو أجهزة تقدم حلاً لوحدة الضرب والقسمة الحية الموحدة. هذه الرسالة تقدم نسختين من الخوارزميات لعملية الضرب والقسمة الحيـــة، الأول لعمليـــة الضرب والقسمة الحية إذا كان المقسوم عليه رقم ثابت، والثاني لعملية الضرب والقسمة الحيـــة لكامـــل المـــدخلات والنواتج الحية. وقد تم عمل مقارنات لأداء تلك الخوارزميات مع الأساليب التقليدية لعمليـــات الضـــرب والقســـمة المستقلة. بالإضافة إلى ذلك، تم ادراج نتائج التركيـــب (synthesis) لمقارنـــة المســـاحة والتـــأخيرالزمني لهـــذه الخوارزميات.

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 Introduction

Digital signal processing (DSP) is widely used today in data compression, neural networks, filtering and processing signals for communication, audio, speech, video, image, sonar, radar, seismic, sensor array and biomedical applications [1]. Digital signal processing has various advantages over analog signal processing such as less noise vulnerability, programmability and the ability to perform sophisticated mathematical operations which can be very difficult to perform using analog components [1]. Digital signals are transmitted as a sequence of samples from an analog-to-digital (A/D) converter. Each sample is represented as a sequence of digits arranged in a specific format [2].

In order to reduce or enhance certain aspects of a digital signal, a digital filter is used. Digital filtering plays a significant role in digital signal processing applications [3], virtual image synthesis [2], data transmission and reception [2], database management [2] and control applications [3]. A digital filter is a system that performs mathematical operations on the signal in which the output is a function of present and past inputs/outputs. The output is usually in a form of a weighted sum produced by multiplying a window of samples by fixed coefficients. Digital filters are generally characterized by a difference equation represented as follows:

$$y(n) = \sum_{k=0}^{\infty} h_k x(n-k) + \sum_{k=1}^{\infty} w_k y(n-k) \tag{1.1}$$

where,

n = sample number

y(n) = system output

x(n) = input sequence

h_k,w_k = pre-defined coefficients

A realization of the difference equation (Equation 1.1) is shown in Figure 1.1. According to the Figure, the data flow of a digital filter mainly consists of a series of multiplication and addition operations.

In the past, the designs of fast modules for digital filters were limited by two factors [4]. First, the sampling rate is affected by the delay between successive recurrence evaluations. Second, the handling of nonlinear oscillations without affecting the sampling rate. Both limitations can be eliminated by introducing online digit-serial implementations for digital filters [4]. Since the input operands and the result of the digital filter operation can be represented as digit vectors, both the inputs and the output may be delivered in a digit serial manner.

In serial operations, at every clock cycle one digit of each input operand is entered and one digit of the output is delivered. One of the main reasons for having serial input/output is to simplify the hardware interface and to reduce the number of signal lines connecting modules. This results in a reduction in the overall area as well as the total power consumption. One mode of digit-serial transmission is by most-significant-digit first (MSDF). In MSDF operations, both the input and output digits are applied or received starting with the most-significant digit. Arithmetic performed in this mode is known as online arithmetic.

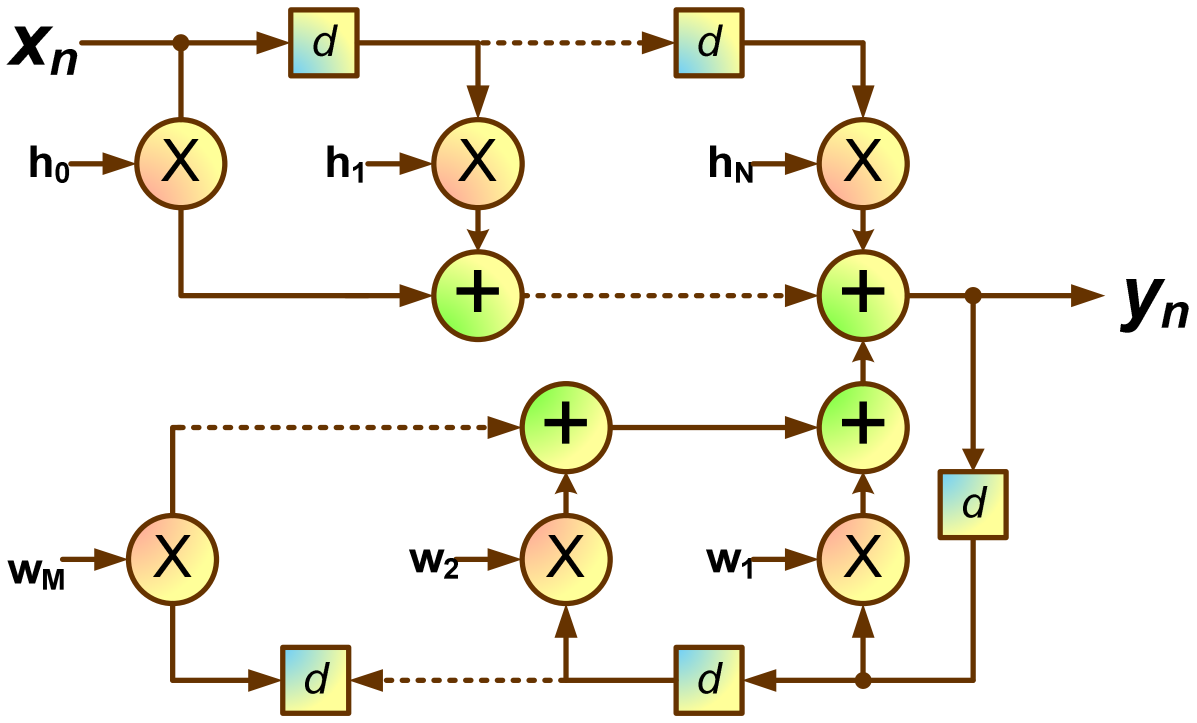


Figure 1.1 A Digital Filter Implementation

Online arithmetic architectures are highly attractive for digital filters because their ability to allow early start of subsequent computations, their compatibility with A/D & D/A converters, their shorter execution times of variable precision operations, and their simplified input/output interconnections [5]. Such benefits of online arithmetic have prompted many researchers in developing different online architectures for digital filters [2] - [11]. Online arithmetic has also been found useful for matrix-based operations, DCT, FFT and CORDIC algorithms [12].

Despite the numerous research which targets improving the throughput of DSP applications using online arithmetic, to our knowledge none has addressed the possible integration of online arithmetic with adaptive filters. Adaptive filters are digital filters in which the filter characteristics are modified and adapted to achieve desired objectives without user intervention [13]. They provide elegant solutions for non-stationary environments or environments with unknown statistics. In modern communication systems, adaptive filters play an important role in the fields of equalization, noise cancellation, linear predication, system identification, and control systems [14]. They are proven to be highly effective in achieving high efficiency, quality and reliability in ubiquitous telecommunication environments [13].

The purpose of the adaptive filter is to mimic the behavior of an unknown digital filter. This is done by having the adaptive filter coefficients $\hat{h}(n)$ equal those of the unknown filter, h(n) within a minimum number of samples. The output of the adaptive filter is subtracted from the output of the unknown system that has been affected with additive noise. Based on the result of the subtraction, the filter coefficients are updated for the next sampling period using some adaptive filter algorithms. Figure 1.2 shows a general outline for such adaptive systems [14].

However, unlike digital filters, the dominant operations of many adaptive filter algorithms consist of a series of multiplications and division operations. These additional mathematical operations are needed in order to update the filter coefficients.

Examples of such algorithms are: The Recursive Least Square Filter [16], the Normalized Least Mean Squares Filter [13] and the Mixed-Norm LMS-LMF filter [17]. The mathematical formulations of each of these adaptive filter algorithms are represented as follows:

1. The Recursive Least Square Filter:

$$\hat{\mathbf{h}}(n) = \hat{\mathbf{h}}(n-1) + \mathbf{k}(n)d(n) + \mathbf{k}(n)\hat{\mathbf{h}}(n-1)\mathbf{x}^{\mathrm{T}}(\mathrm{n})$$

$$\mathbf{k}(n) = \frac{\mathbf{P}(n\text{-}1)\mathbf{x}(n)}{1+\mathbf{x}^{T}(n)\mathbf{P}(n\text{-}1)\mathbf{x}(n)}$$

2. Normalized Least Mean Squares Filter:

$$\hat{\mathbf{h}}(n+1) = \hat{h}(n) + \mu\frac{\mathbf{e}(n)\mathbf{x}(n)}{\|\mathbf{x}(n)\|^{2}}$$

3. Mixed-Norm LMS-LMF Filter:

$$\hat{\mathbf{h}}(n+1) = \hat{h}(n) + [\alpha_n e(n) + 2(1-\alpha_n)e^{3}(n)]\frac{\overline{\mu}\cdot X(n)}{\|X(n)\|^{2}}$$

where,

$\hat{h}(n)$: the adaptive filter coefficients

$\mathbf{k}(n)$: the gain factor

d(n): output of the unknown system

x(n): the input vector

e(n): error

$\|X(n)\|$: Euclidean norm of the vector X(n)

μ: step size

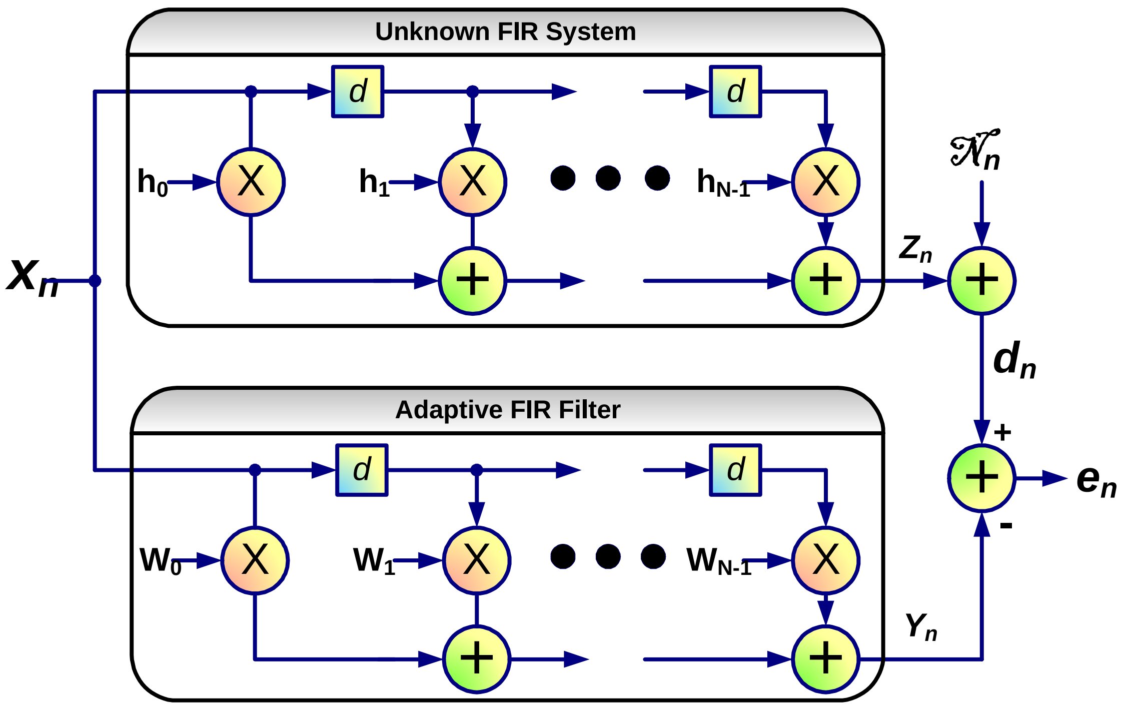


Figure 1.2 General Adaptive Filter System

For instance, the hardware architecture of the Normalized Least Mean Squares Filter is shown in Figure 1.3. Based on Figure 1.3, the adaptive filter arithmetic implementation mainly constitutes of multiplication operations followed by a division operation. The consecutive multiplication and division operations can be merged together into a single arithmetic unit called a multiplication-division operation. Multiplication-division operations are independent arithmetic units that computes Q = A * B / D, where A is the multiplier input, B is the multiplicand input, D is the divisor input and Q is the quotient output.

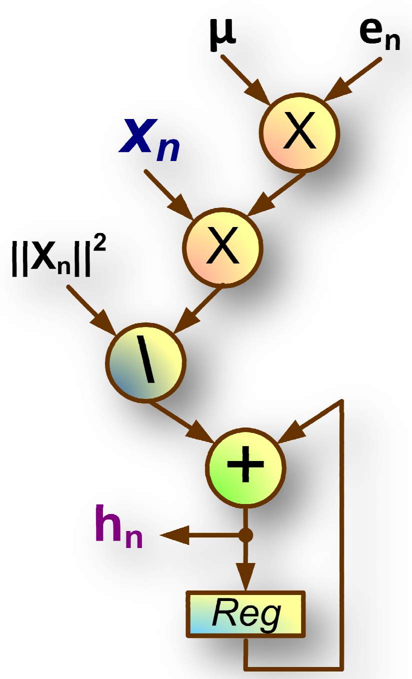


Figure 1.3 Data flow path for the Normalized Least Mean Square adaptive filter algorithm

An online implementation of such kind of adaptive filters requires an online implementation for a multiplication-division unit. To the best of our knowledge, there have been no reports of algorithms or implementations for online multiplication-division in any existing literature or patents. Although separate operations for online multiplication and online division operations do exist, implementing adaptive filters using these single online multiplication and division operations requires large amount of unnecessary hardware and have long latencies. Such implementations are inefficient especially in today's highly competitive market where applications such as cell phones, hearing aids, and digital audio devices require an ever increasing number of constraints on area, power and speed [14]. Therefore, in order to efficiently implement adaptive filters using online arithmetic architectures, a high-performance online multiplication-division algorithm is required.

1.2 Thesis Objectives and Organization

The purpose of this thesis is to devise an algorithm for an online multiplication-division arithmetic operation. In this thesis work, two novel algorithms for online multiplication-division are introduced. The first algorithm, *Online Multiplication-Division with Parallel Divisor*, computes Q = A*B/D, given that the multiplicand (A), the multiplier (B) and the quotient (Q) are online inputs/outputs while the divisor (D) input is received in parallel. The second algorithm, *Fully Online Multiplication-Division based on Composite Algorithms*, computes Q = A*B/D given that A, B, D and Q are all online inputs and outputs. Both algorithms deliver the remainder in parallel. Because the first algorithm has a shorter latency and higher throughput compared to the latter, it is useful for environments when the divisor is a constant or is readily available before the online operation begins. Since all online division and online multiplication-division algorithms may require a correction stage at the last clock cycle, a generic high-speed correction stage algorithm is also proposed in this thesis that can be consolidated with either of the two algorithms.

The objectives of the thesis are: (1) to study reported online arithmetic operations. (2) to devise an on-line algorithm for multiplication-division for both parallel input divisors and online input divisors. (3) Propose hardware implementations for both algorithms. (4) Model and simulate the proposed online multiplier-dividers using a

Hardware Description Language (HDL) such as Verilog. (5) Synthesize the HDL models and evaluate both algorithms performance/efficiency in terms of time and hardware complexities. (6) Model and synthesize an online multiplication-division arithmetic operation, consisting of cascaded online multiplication and online division units as a reference implementation for performance comparison in terms of timing and hardware costs.

The rest of the thesis is organized as follows. In chapter-2, online arithmetic background information is given. This chapter also includes literature survey of reported online arithmetic operations, e.g. online addition, multiplication, division and on-the-fly conversion. Chapter 3 provides a detailed discussion on the two proposed algorithms for online multiplication-division together with their mathematical proofs. The chapter first starts by describing the sequential modular multiplication algorithm of Koc and Hung [15] which is followed by a step by step explanation of how to convert the Koc-Hung Modular Multiplication into an online multiplication-division algorithm. Chapter 3 also includes a detailed description of a unified fully online multiplication-division algorithm in which the divisor is an online input. The algorithm is developed using composite algorithms based on the conventional methods of computing multiplication-division operations. Later in the same chapter, a high-speed correction stage is proposed.

Chapter 4 presents the hardware implementation of the two proposed algorithms compared to the conventional online multiplication-division implementation. Synthesis results and discussion are also presented in Chapter-4. Conclusion and future work are given in Chapter 5.

CHAPTER 2

BACKGROUND

2.1 Serial Arithmetic

In serial operations, at every clock cycle one digit of each input operand is entered and one digit of the output is delivered. One of the main reasons for having serial input/output is to simplify the hardware interface and to reduce the number of signal lines connecting modules. This results in a reduction in the overall area as well as the total power dissipation.

There are two modes of digit-serial transmission; the Least-Significant Digit First (LSDF) mode, and the Most Significant Digit First (MSDF) mode. In the LSDF mode, the digits of the operands are applied serially starting with the least-significant digit (Figure 2.1A) and the digits of the result are produced starting with the least significant digit first as well. This mode is also known as right-to-left mode. Conversely, in the MSDF mode, both the input and output digits are applied or received starting with the most-significant digit (Figure 2.1B). Arithmetic performed in this mode is known as online arithmetic, and the corresponding latency is known as the online delay [18].

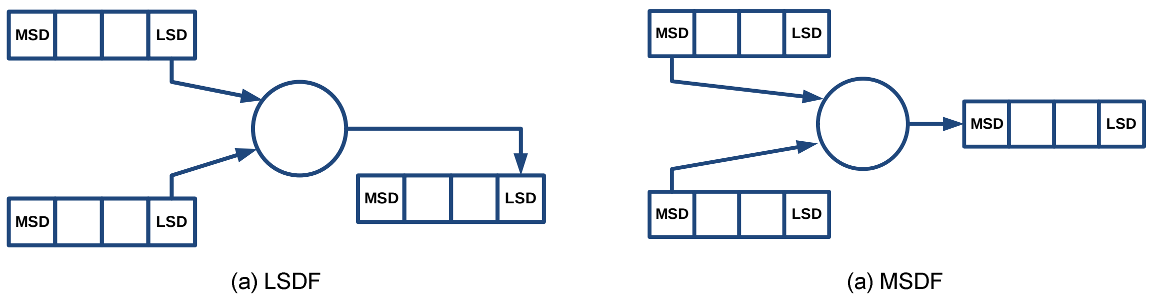


Figure 2.1 Data Flow during Digit Serial Operations

2.2 Online Arithmetic

On-line arithmetic, introduced by Ercegovac in 1977 [19], has been considered as an efficient solution to applications in communication and many signal processing problems [6], [9], [12] and [21]. Online arithmetic algorithms reduce the computation time of long sequences of arithmetic operations by allowing overlapped executions between sequential operations.

Generally, online schemes for digit-serial implementations are favored over LSDF schemes for several reasons. First, unlike LSDF operations, division and square-root calculations only work in the MSDF direction [22]. Second, most digital communication systems operate on data obtained from Analogue to Digital Converters (ADC). Low cost ADC's such as Successive Approximations (SA) produce the result one bit at a time starting with the most significant bit first (MSBF) [22]. Third, in some signal processing applications, results of arithmetic operations need not be calculated in full precision. Unlike LSDF, online arithmetic operations allow truncation and rounding of the result with the least number of clock cycles. Fourth, online arithmetic eliminates the time-consuming carry-propagation during addition [22].

An important characteristic of online arithmetic is the online delay or latency δ. The online delay is the minimum number of operand digits required to be received before the first digit of the output is generated. Therefore, the jth digit of the output result appears after receiving $j + \delta$ digits of the input operands. Figure 2.2 illustrates the case of an on-line delay of $\delta = 2$. Although this overhead time may appear as a disadvantage for implementing online arithmetic architectures, the delay is usually compensated by overlapping the execution of successive operations.

Input A:	a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
Input B:	b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
Output Y:	$\leftarrow\delta\rightarrow$ y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8
Example of $\delta = 2$	

Figure 2.2 An Online delay of 2

As shown in Figure 2.3, parallelism can be achieved by overlapping arithmetic operations at the digit level [23]. A short delay after the first digit of the result becomes available for the next operation. The operations which use as input the result of the preceding operations does not have to wait for the previous ones to finish in order to start execution. Accordingly, online arithmetic effectively overlaps arithmetic operations with strong data dependencies [23].

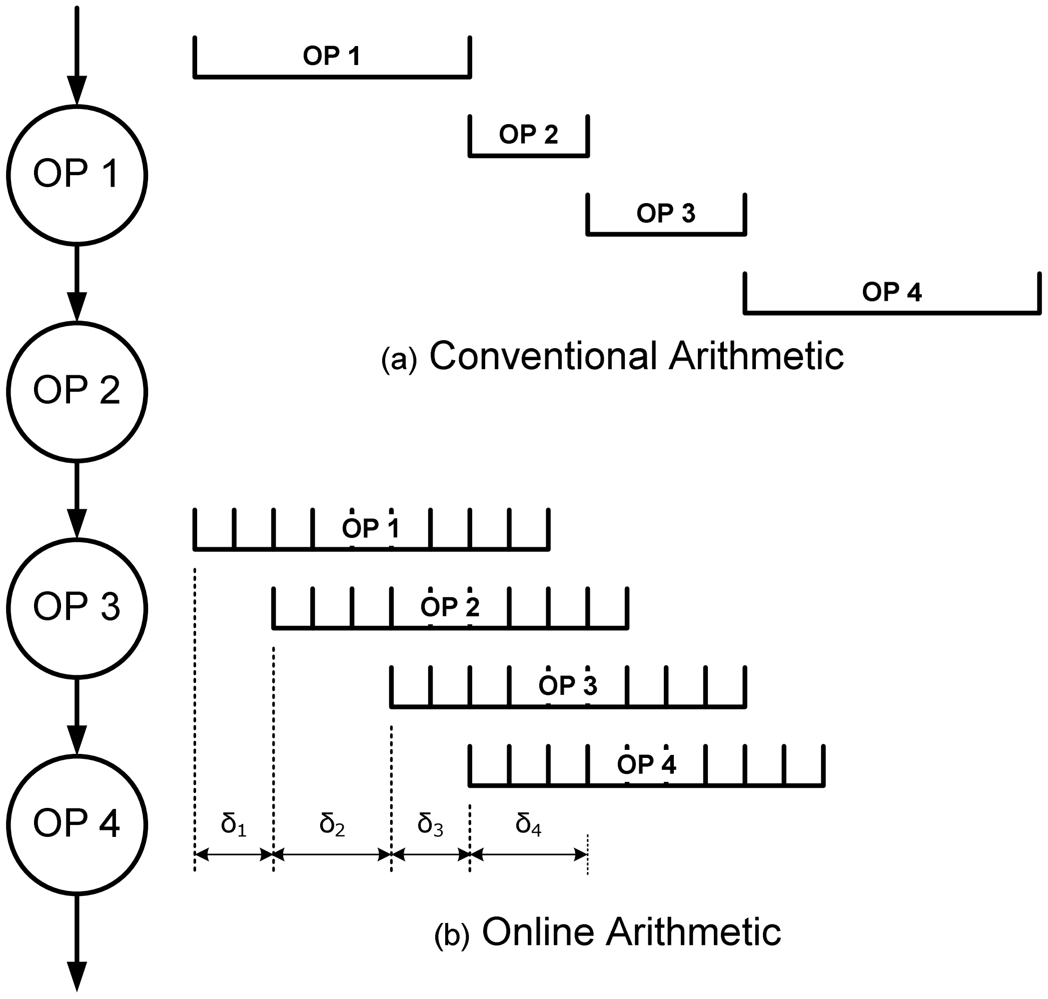


Figure 2.3 Timing of sequential operations using conventional (a) and online arithmetic (b)

Another important characteristic of online arithmetic is the use of redundant number representations. A given value may have several representations introducing flexibility in computing output result digits on the basis of the available partial information of the input operands. Redundancy also allows the possibility of limiting carry propagation up to only one digit position. There are two commonly used types of redundant number representations; signed digit and carry-save [18]. Signed-digit number representations are more commonly used for online arithmetic. Redundant signed-digit number representation permits negative digit values, for example $27\bar{3}$ can also be represented as 267, i.e. 270 – 3 = 267. The output of online arithmetic is represented in a radix-r redundant number system with a symmetric digit set {-*p*, ..., *p*}, where $r/2 \le p \le r-1$.

There are three main properties that generally characterize on-line arithmetic algorithms [5]:

(1) Operands and results should be in a fractional on-line form such that the values of inputs A, B and the output C at time j would be:

$$A[j] = \sum_{i=1}^{j+\delta} a_i r^{-i} = A[j-1] + a_{j+\delta} r^{-(j+\delta)}$$
$$B[j] = \sum_{i=1}^{j+\delta} b_i r^{-i} = B[j-1] + b_{j+\delta} r^{-(j+\delta)}$$
$$C[j] = \sum_{i=1}^{j} c_i r^{-i} = C[j-1] + c_{j+\delta} r^{-j}$$

(2) A residual function, G, determines the internal state, W^j, of the operation:

$$W^j = G(W^{j-1}, A[j], B[j], C[j])$$

(3) A selection function S that defines the output digit c_j which, in general, depends on the internal state, the inputs and/or the previous outputs:

$$c_j = S(W^{j-1}, A[j], B[j], C[j-1])$$

For k-bit inputs/outputs, the online algorithm iterates for a total of δ + k clock cycles. The input digits are introduced in the first k clock cycles and are 0 afterwards. On the other hand, the output digits are 0 in the first δ clock cycles. During each iteration, the online algorithm evaluates a recurrence expression and generates the next output digit by executing a digit selection function based on the result of the recurrence evaluation.

The following sections describe reported hardware algorithms for on-line addition/subtraction, on-line multiplication, on-line division and on-the-fly conversion.

2.3 Online Addition/Subtraction

The online addition algorithm performs serial addition of two signed-digit operands in the most-to-least direction. Addition is performed through a carry-free signed digit adder. Online addition for a radix-r, with r > 2 has an online delay of $\delta = 1$. Assuming a balanced digit set $[-\alpha, \alpha]$ and a radix r > 2, the online addition algorithms to compute Z = X+Y may be expressed using the following equation [18],

$$z_i = c_i + s_i$$

where c_{i+1} and s_i are given by:

$$(c_{i+1}, s_{i+2}) = \begin{cases} (0,\ x_{i+2} + y_{i+2}) & \text{if } \left|x_{i+2} + y_{i+2}\right| \leq a - 1 \\ (1,\ x_{i+2} + y_{i+2} - r) & \text{if } x_{i+2} + y_{i+2} \geq a \\ (-1, x_{i+2} + y_{i+2} + r) & \text{if } x_{i+2} + y_{i+2} \leq -a \end{cases}$$

where $r/2 \leq \alpha \leq r - 1$ and i ranges from 0 to n-1

Figure 2.4 illustrates the physical implementation of a radix-r online adder.

Table 1, demonstrates a radix-4 online addition example with digit set [-3, 3] operands:

$X = .12\bar{3}30\bar{1}$, $Y = .2\bar{1}\bar{3}322$.

The result is $Z = 1.\bar{1}0\bar{1}221$

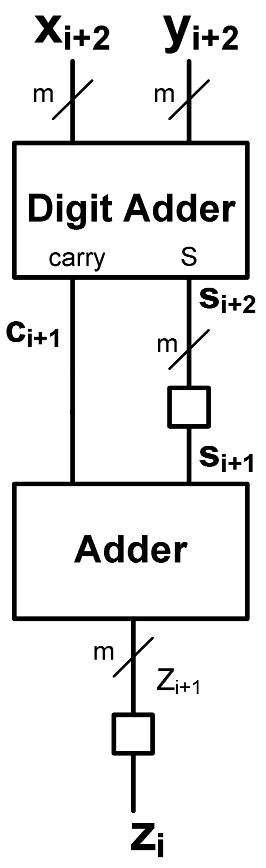


Figure 2.4 Radix-r online adder (r = 2^m)

Table 2.1 Example of radix-4 online addition

i	x_{i+2}	y_{i+2}	c_{i+1}	s_{i+2}	s_{i+1}	z_{i+1}	z_i
0	1	2	1	-1	0	1	0
1	2	-1	0	1	-1	-1	1
2	-3	-3	-1	-2	1	0	-1
3	3	3	1	2	-2	-1	0
4	0	2	0	2	2	2	-1
5	-1	2	0	1	2	2	2
6	0	0	0	0	1	1	2
8	0	0	0	0	0	0	1

2.4 Online Multiplication

Two types of online multiplication algorithms have been proposed by Ercegovac. The first algorithm is used only for non-redundant binary inputs. The other algorithm is used for inputs that can be represented in both binary and signed-bit number representation.

2.4.1 Online Multiplier for Normal-Digit Inputs

Ercegovac and Trivedi proposed the Online Multiplier for Normal-Digit Inputs algorithm in 1977 [20]. The n-digit input operands A and B are received serially with most significant-digit first. The digit of the product C is produced serially after a delay of $\delta = 2$. The output is represented in a radix-r redundant number system with a symmetric digit set of {-p, ... , p}. The full precision of the output is generated after 2n + 2 clock cycles. The algorithm defines a residual function $W^{(j)}$ which is computed according to the following recurrence relation with j ranging from 1 to 2n + 2:

$$W^{(j)} = r(W^{(j-1)} - c_{j-1}) + A[j]b_j + a_j B[j-1] \tag{2.1}$$

where $W^{(0)} = 0$, $A[0] = B[0] = 0$ and $c_0 = 0$

A[j] and B[j] are vectors that are represented by the following equations:

$$A[j] = \sum_{i=0}^{j} a_i 2^{-i}, \; B[j] = \sum_{i=0}^{j} b_i 2^{-i}$$

The digit serial output is based on the following digit selection function:

$$cj = \begin{cases} sign(W^{(j)}) * \left|W^{(j)}\right| + 1/2 & if \left|W^{(j)}\right| \leq p \\ sign(W^{(j)}) * \left\lfloor W^{(j)} \right\rfloor & otherwise \end{cases}$$

<u>Example</u> of radix-2 operation:

Assume that the inputs are: A = 0.01011101,   B = 0.01110101

The computed output is C = $0.0010101011\bar{1}0000011\bar{1}$

C is then converted to conventional format: C = 0.0010101010000001

Table 2.2 Example of Ercegovac online multiplication for normal digits

j	*A[j]*	*B[j-1]*	a_j	b_j	$A[j]b_j+B[j-1]a_j$	*W[j]*	C_j	$2(W[j]-c_j)$
1	0.0	0.0	0	0	0.0	0.0	0	0
2	0.01	0.0	1	1	0.01	0.01	0	0.1
3	0.010	0.01	0	1	0.01	0.11	1	-0.1
4	0.0101	0.011	1	1	0.1011	0.0011	0	0.011
5	0.01011	0.0111	1	0	0.01110	0.11010	1	-0.0110
6	0.010111	0.01110	1	1	0.110011	0.011011	0	0.11011
7	0.0101110	0.011101	0	0	0.0	0.11011	1	-0.0101
8	0.01011101	0.0111010	1	1	0.11010001	0.10000001	1	-0.1111111
9	0.01011101	0.01110101	0	0	0.0	-0.1111111	-1	0.000001
10	0.01011101	0.01110101	0	0	0.0	0.000001	0	0.00001
11	0.01011101	0.01110101	0	0	0.0	0.00001	0	0.0001
12	0.01011101	0.01110101	0	0	0.0	0.0001	0	0.001
13	0.01011101	0.01110101	0	0	0.0	0.001	0	0.01
14	0.01011101	0.01110101	0	0	0.0	0.01	0	0.1
15	0.01011101	0.01110101	0	0	0.0	0.1	1	-1
16	0.01011101	0.01110101	0	0	0.0	-1	-1	0

2.4.2 Online Multiplier for signed-digit inputs

This second type of online multiplier was also proposed by Ercegovac and Lang [18]. The algorithm accepts inputs represented in either binary or signed-digit representations. The compatibility with signed-digit inputs provides an advantage for recursive applications where the signed-digit outputs can be used directly for further operations without conversion to binary representation. However, this new approach increases the on-line delay to $\delta = 3$. The total number of clock cycles required to compute the output in full precision is $T = 2n + 3$.

The residual function is decomposed into the following two functions:

$$V^{(j)} = 2 * W^{(j-1)} + 2^{-3} * (b_j A[j] + a_j B[j-1])$$

$$W^{(j)} = V^{(j)} - c_j$$

where $W^{(0)} = 0$, $B[0] = 0$, $1 \le j \le 2n+3$.

The serial output depends on the product-digit selection function:

$$c_j = \begin{cases} 1 & if \quad \tfrac{1}{2} \le V^{(j)} \le \tfrac{7}{4} \\ 0 & if \quad -\tfrac{1}{2} \le V^{(j)} \le \tfrac{1}{4} \\ -1 & if \quad -2 \le V^{(j)} \le -\tfrac{3}{4} \end{cases}$$

The hardware architecture of this type of online multiplication algorithm is shown in Figure 2.5. A [4:2] compressor is used to speed up the addition,

$$2W^{(j-1)} + 2^{-3}b_jA[j] + 2^{-3}a_jB[j-1]$$

Since the [4:2] compressor produces a sum in a form of a carry-sum pair (VC, VS), a 4-bit estimate of V is calculated by adding the 4 most significant bits of VS and VC. This estimate is produced by the **V** block. The **M** block then performs subtraction of c_j to obtain W.

<u>Example</u> for radix-2 inputs:

$A = 0.101\bar{1}\bar{1}110$ $\qquad$ $B = 0.110\bar{1}10\bar{1}1$

Table 2.3 shows that the computed product (truncated) is $c = 0.10\bar{1}01\bar{1}10$ while the true product in conventional form is c = 0.0110010110000010

Table 2.3 Example of online multiplication for signed digits

j	a_i	b_i	*A[j]*	*B[j-1]*	*V[j]*	c_i	*W[j]*
1	1	1	.1	.0	00.0001	0	00.0001
2	0	1	.10	.1	00.00110	0	00.00110
3	1	0	.101	.11	00.011110	0	00.011110
4	-1	-1	.1001	.110	00.1100011	1	11.1100011
5	-1	1	.10001	.1011	11.10000111	0	11.10000111
6	1	0	.100011	.10111	11.001001010	-1	00.001001010
7	1	-1	.1000111	.101110	00.0100111101	0	00.0100111101
8	0	1	.10001110	.1011011	00.1011000001	1	11.1011000001
9	0	0	.10001110	.10110111	11.011000001	-1	00.011000001
10	0	0	.10001110	.10110111	00.11000001	1	11.11000001
11	0	0	.10001110	.10110111	11.1000001	0	11.1000001

2.5 Online Division

An on-line division unit computes Q = N/D such that the quotient digits are generated at the same rate as the operand digits are received. The online division is implemented using the following recurrence equation [24]:

$$W^{(j)} = 2(W^{(j-1)} - q_{i-1}D[j-1]) + n_{j+4}2^{-4} - d_{j+4}Q[j-1]2^{-4} \qquad (2.2)$$

where,

$q_0 = Q[0] = 0$, $W^{(0)} = N[0]$ and $n_k = d_k = 0 \ \forall k \geq n$

The following conventions are used for the digit vectors:

$$N[j] = \sum_{i=1}^{j+\delta} n_i 2^{-i},\ D[j] = \sum_{i=1}^{j+\delta} d_i 2^{-i},\ W[j] = \sum_{i=1}^{j} w_i 2^{-i}, \text{ and } Q[j] = \sum_{i=1}^{j} q_i 2^{-i}$$

The quotient digit is produced by the selection function:

$$q_j = Sel(W) = \begin{cases} 1 & if\ W \geq 1/4 \\ \overline{1} & f\ W < -1/4 \\ 0 & otherwise \end{cases}$$

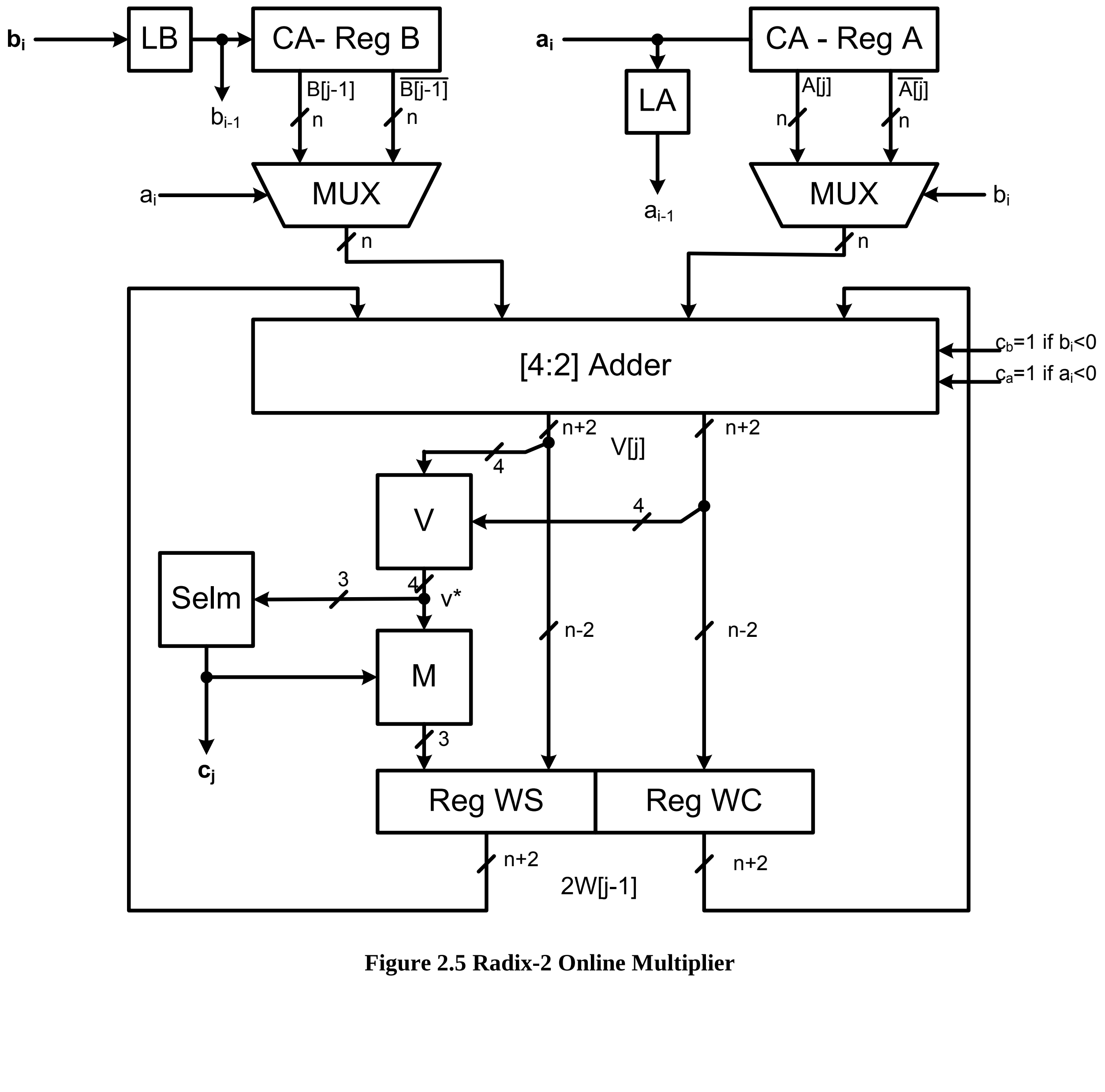


Figure 2.5 Radix-2 Online Multiplier

However, the input operands must satisfy these two conditions:

(i) $N < D$

(ii) $½ \leq D < 1$

The on-line delay of the above algorithm is $\delta = 4$ and accordingly, the recurrence relation only works after the first four digits are received. Thus the first 4 clock cycles are labeled by the clock cycles -3 to 0. The recurrence relation then starts from clock cycle 0 to n. All operations are performed in fractions and the result is delivered in signed-bit-representation.

Example for 6-bit radix-2 inputs:

N = 0.010011

D = 0.110110

Table 2.4 Example of a radix-2 online division

j	n_j	d_j	N[j]	D[j]	W[j]	q_j	Q[j]
-3	0	1	0.0	0.1	0	0	0
-2	1	1	0.01	0.11	0	0	0
-1	0	0	0.010	0.110	0	0	0
0	0	1	0.0100	0.1101	00.0100	0	0
1	1	1	0.01001	0.11011	00.1001	1	0.1
2	1	0	0.010011	0.11011	11.1001	$\bar{1}$	$0.1\bar{1}$
3	0	0	0.010011	0.11011	00.1101	1	$0.1\bar{1}1$
4	0	0	0.010011	0.11011	11.1111	0	$0.1\bar{1}1$
5	0	0	0.010011	0.11011	11.111	0	$0.1\bar{1}1$
6	0	0	0.010011	0.11011	11.11	0	$0.1\bar{1}1$

From Table 2.4, the output result is $Q = 0.1\bar{1}1000 = 0.011000 = 0.375$ which is the closest result equal to 0.351851851 that can be expressed within a 6-bit number.

According to [18] the online division algorithm can be efficiently implemented in hardware by reducing the data dependencies in equation 2.2, using carry-save adders, and by merging the terms $n_{j+4}2^{-4}$ with $d_{j+4}Q[j-1]2^{-4}$ into a single unified number. The data dependency in equation 2.2 can be reduced by splitting the equation into two recurrence functions:

$$V^{j} = 2W^{j} + n_{j+4}2^{-4} - d_{j+4}Q[j-1]2^{-4} \tag{2.3}$$

$$W^{j+1} = V^{j} - q_{j}D[j]$$

with the selection function depending on V:

$$q_{j} = Sel(V) = \begin{cases} 1 & if\ V \geq 1/4 \\ \overline{1} & if\ V < -1/4 \\ 0 & otherwise \end{cases} \tag{2.4}$$

As a result, in each iteration the output q_j only depends on the calculation of V^j i.e. the sum of $2W^{j} + n_{j+4}2^{-4} - d_{j+4}Q[j-1]2^{-4}$ instead of the lengthy summation of W^j as in equation 2.2. Due to the use of carry-save adders, V and W are then represented as n+6 bit carry-sum pairs defined as (VC, VS) and (WC, WS). The first 2 bits are integer bits while the rest are fractional bits. The 2 integer bits consist of a sign bit and the other is for accommodating the left-shift of W in equation 2.3. The four additional fractional bits are for accommodating the multiplication process of $2^{-4} * (n_{j+4} - d_{j+4}Q[j-1])$.

Since the selection function depends on V in its conventional form, an estimate of V, denoted as $\hat{V}$ is calculated from (VC, VS) . The estimation is performed using a 5-bit carry-propagate adder which sums the 5 most-significant bits of VC, i.e. VC_1 VC_0. VC_{-1} VC_{-2} VC_{-3} with the 5 most-significant bits of VS, i.e. VS_1 VS_0. VS_{-1} VS_{-2} VS_{-3}. The

4 most significant bits of the estimate is then compared with ±00.01 using the selection function.

Furthermore, to reduce the number of input operands for carry-save addition, the terms $n_{j+4}2^{-4}$ and $d_{j+4}Q[j-1]2^{-4}$ are merged together into a single value. Since $n_{j+4}2^{-4}$ can be either (00.0001), (11.1111) or (0), and $d_{j+4}Q[j-1]2^{-4}$ can be either (0), (00.0000Q[j-1]) or (11.1111$\overline{Q[j-1]}$), it is possible to unify the 6 most-significant bits using a combination of n_{j+4} and d_{j+4}. This is done by using a **U**-block that produces a unified 6-bit value based on Table 2.5. The U-block is then concatenated with 2^{-4}Q[j-1] and passed as a single input to the carry-save adder for calculating V.

Table 2.5 The U-Block Function

n_{j+5}	d_{j+5}	U	n_{j+5}	d_{j+5}	U
1	1	00.0000	1	$0,\bar{1}$	00.0001
0	1	11.1111	0	$0,\bar{1}$	00.0000
$\bar{1}$	1	11.1110	$\bar{1}$	$0,\bar{1}$	11.1111

The overall online division algorithm using carry-save adders is summarized in Algorithm 2.1.

Algorithm 2.1 Online Division Algorithm

```
1.[initialize]
D[0]=Q[0]=W$^0$=0
for j = -4 to -1
  D[j] ← CA(D[j-1],$d_{j+4}$)
  $V^j$ = 2$W^j$ + $n_{j+4}2^{-4}$
  $W^{j+1}$ ← $V^j$
end for
```

```
2. [Recurrence]
for j = 0 … k – 1
  D[j] ← CA(D[j-1],$d_{j+4}$)
  $V^j$ =2$W^j$ + $n_{j+4}$•$2^{-4}$ – $d_{j+4}$•Q[j-1]•$2^{-4}$
  $\hat{V} = VC_{1:-4} + VS_{1:-4}$
  $q_j$ = SELD($\hat{V}$)
  $W^{j+1}$ ← $V^j$ – $q_j$D[j]
  Q[j] ← CA(Q[j-1],$q_j$)
  Output $q_j$
end for
```

The corresponding hardware implementation for Algorithm 2.1 is illustrated in Figure 2.6. The **CA** blocks are for converting the accumulated values of d_j and q_j to 2's complement representation using on-the-fly conversion (explained in Section 2.6). The **LN** and **LD** blocks are single bit flip-flops. The two 6-bit carry-save adders are for calculating V^j – q_jD[j] and $2W^j + n_{j+4}2^{-4} - d_{j+4}Q[j-1]2^{-4}$. The two **MUX** block are multiplexers to compute q_j.D[j] and d_{j+4}•Q[j-1]. The **V** block is a 5-bit carry-propagate adder to determine the estimate of $V^{(j)}$. Finally, the **Selm** block performs the selection function to produce the output q_j.

2.6 On-The-Fly Conversion

On-the-fly is a conversion algorithm developed by Ercegovac and Lang [25] to convert binary signed digit online inputs into conventional representation. The algorithm receives a k-bit input q_j, one bit at each clock cycle starting with the most-significant bit. After k iterations, the algorithm produces a k-bit output Q[j] in 2's complement format. To avoid carry propagation, an intermediate result called QM[j] is used. At each iteration, both Q[j] and QM[j] are updated using appending operations based on the following conditions:

$$QM[j+1] = \begin{cases} (Q[j], 0) & \text{if } q_j > 0 \\ (QM[j], 1 + q_j) & \text{if } q_j \le 0 \end{cases}$$

$$Q[j+1] = \begin{cases} (Q[j], |q_j|) & \text{if } q_j \ge 0 \\ (QM[j], 1) & \text{if } q_j < 0 \end{cases}$$

For example, the input $0.10\bar{1}1\bar{1}\bar{1}01$ can be converted to conventional representation as shown in Table 2.6. The solution is determined at the last clock where Q[j] = 0.01100101.

Table 2.6 On-the-fly Conversion Example

j	**q_j**	**QM[j]**	**Q[j]**
0	1	0.0	0.1
1	0	0.01	0.10
2	$\bar{1}$	0.010	0.011
3	1	0.0110	0.0111
4	$\bar{1}$	0.01100	0.01101
5	$\bar{1}$	0.011000	0.011001
6	0	0.0110001	0.0110010
7	1	0.01100100	0.01100101

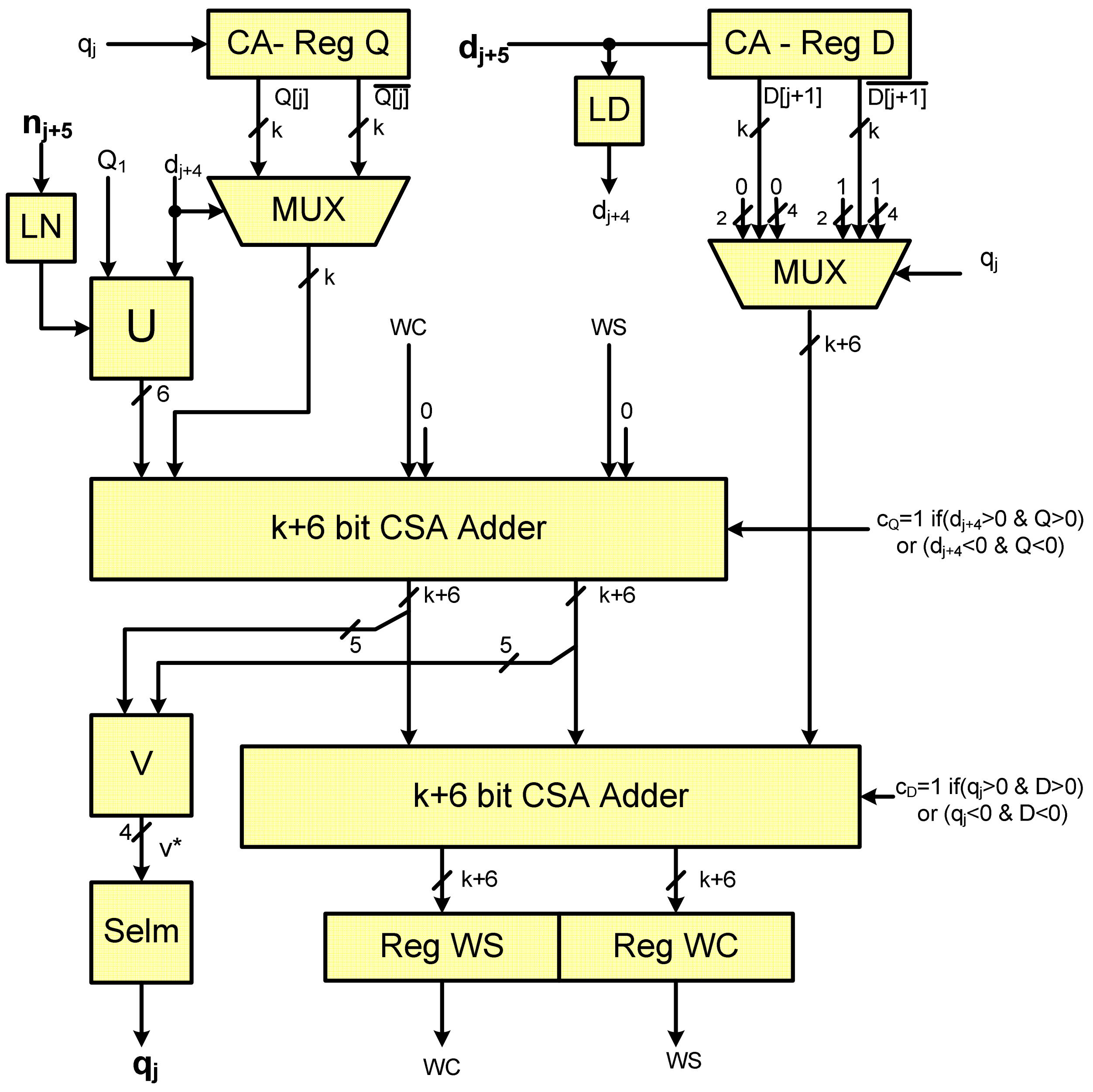


Figure 2.6 Hardware Architecture for a Radix-2 Online Division

CHAPTER 3

ONLINE MULTIPLICATION-DIVISION

3.1 Koc-Hung Modular Multiplication

The modular multiplication algorithm proposed by Koc and Hung [15] performs modular multiplication by interleaving the shift-and-add procedure for multiplying two numbers with a customized modular reduction steps. The modular reduction steps are based on a sign estimation technique that estimates the sign of a given number represented by a carry-sum pair produced by a carry-save adder. Carry-save adders are used to eliminate the problem of carry-propagation leading to more efficient hardware implementations [15].

In a modular reduction operation (X mod M), X is compared with some integer multiple q of M. Hence, the result of the comparison is determined by calculating the sign of (X - qM). For non-redundant numbers represented in 2's complement, the sign is indicated by the most significant bit. However, when carry-save addition is used, the sign of the resulting carry-sum pair may not be readily available and the precise determination of the sign requires the computation of the total sum. Computing the sum may take O(n) time with a carry-propagate adder which significantly increases the delay.

Instead, the authors developed a sign estimation technique for detecting the sign of the carry-sum pair.

The sign estimation algorithm computes the sign of a number only when the number is large enough in magnitude. Given two binary numbers C and S representing the carry-sum components of an (n + 4)-bit 2’s complement number, the sign estimation function ES(S,C) adds the 5 most significant bits of the sum (S) and carry (C) components of the number, i.e. $ES(S,C) = S_{n+3:n-1} + C_{n+3:n-1}$. The sign estimation function returns of the 3 possible results:

1. The sign is positive (+): if $ES(S,C) \geq 1$
2. The sign is negative (-): if $ES(S,C) \leq -2$
3. The sign is unsure (±): if $-2 < ES(S,C) < 1$

Algorithm 3.1 computes P = AB mod M, where M is a k-bit number. The most significant bit of the modulus must be 1, i.e. $2^{k-1} \leq M < 2^k$. The inputs A and B are positive numbers that are less than M. The partial product is represented in signed 2's complement carry-save pair format (PC, PS) using k+4 bits, i.e. 4 additional bits are needed. The left-most bits PS_{k+3} and PC_{k+3} are used as sign bits while the other three bits are required to accommodate the scaling factor of M.

In line 1, the modulus is scaled by a factor of 8. This ensures that the partial product (PS, PC) stays within the range $-2^{k-1} < (PS+PC)/8 < 2^{k-1}$ after the addition of A_iB (lines 4, 5 and 6) in each iteration. Lines 2 to 8 perform combined shift, add and modular reduction operations. The iteration index i counts from k-1 down to -3, thus A_{-1}, A_{-2} and A_{-3} are defined to be 0. The three extra iterations compensate for the use of the scaled up modulus. The assimilation of PS, PC and the correction step are in lines 9 and 10. The correction step ensures that the generated remainder lies between 0 and M, i.e. P ϵ [0,M). Finally in line 11, the result is scaled back by 8 before being returned.

Algorithm 3.1 Koc-Hung Modular Multiplication Algorithm

```
1.  M` := 8M, PS := 0, PC := 0
2.  For i = k - 1, k - 2, …, 0, -1, -2, -3 do
3.    begin
4.    IF  ES(PS, PC) ≥ 1  then  (PS, PC) := 2PS + 2PC + $A_iB$ – M`
5.    ELSEIF ES(PS, PC) ≤ -2 then (PS, PC):= 2PS + 2PC + $A_iB$ + M`
6.    ELSE  (PS, PC) := 2PS + 2PC + $A_iB$
7.    ES(PS, PC) := $PS_{k+3:k-1}$ + $PC_{k+3:k-1}$
8.    End
9.  P = PS + PC
10. IF P < 0  then  P := P + M`
11. Return P/8.
```

The correctness of the modular multiplication algorithm can be proven by showing throughout the for loop:

$$PS + PC \in \left[-\frac{3M`}{4}, \frac{7M`}{8} \right) \tag{3.1}$$

The correction stage (lines 9-11) returns a final result within the range [0,M)

In line1, PS and PC are initially set to 0, thus the condition of equation 3.1 is true. To show that equation 3.1 remains true after every iteration, the three cases of the sign estimation need to be considered:

- **ES (PS, PC) = (+):** in this case PS + PC ≥ 2^{k-1} > M`/8, and thus PS + PC is within (M`/8, 7M`/8). After execution of line 4 in Algorithm 3.1 by shifting PC/PS, addition of $A_iB$ and the subtraction of M`, PS + PC $\in$ [-3M`/4, 7M`/8).

  Proof: Assume that (PC, PS) at the beginning of iteration i is equal to the maximum theoretical value i.e. (PC + PS) = 7M`/8. Since the inputs A and B should be less than M, we can also take into consideration that at iteration i $A_iB$ is equal to its maximum value M`/8 -1.

Hence, at line 4,

$$(PS + PC) = 2*7M`/8 +( M`/8 -1) – M` = (14M`+M`-8M`)/8 – 1$$

$$= 7M`/8 – 1 < 7M`/8$$

On the other hand, by considering $A_iB = 0$ and (PC+PS) equals to its minimum value, i.e. (PC+PS) = M`/8, the execution of line 4 results in:

$$(PS + PC) = 2*M`/8 + 0 – M` = (2M`-8M`)/8$$

$$= -6M`/8 = -3M`/4$$

Thus the conditions for equation 3.1 remains true after executing line 4.

- **ES(PS, PC) = (-):** in this case PS + PC $\leq -2^k$ < -M`/8, and thus PS + PC is within [-7M`/8, -M`/8). Once shifting, addition of $A_iB$ and the addition of M` are done according to line 5, PS + PC $\in$ [-3M`/4, 7M`/8).

  Proof: Assume that PC, PS at the beginning of iteration i is equal to the maximum theoretical value i.e. (PC + PS) = -M`/8. Since the inputs A and B should be less than M, we can also take into consideration that at iteration i $A_iB$ is equal to its maximum value M`/8 -1. Hence the upper bounds of equation 3.1 are satisfied after executing line 5,

$$(PS + PC) = -2*M`/8 + (M/8 -1) + M = (-2M`+M`+8M`)/8 – 1$$

$$= 7M`/8 – 1 < 7M`/8$$

On the other hand, by considering $A_iB = 0$ and (PC+PS) equals to its minimum value, i.e. (PC+PS) = -7M`/8, the execution of line 5 results in:

$$(PS + PC) = -2*7M`/8 + 0 + M` = (-14M`+8M`)/8$$

$= -6M`/8 = -3M`/4$

Thus the conditions for equation 3.1 remains true after executing line 5.

- **ES(PS, PC) = (±):** in this case, it is guaranteed that PS + PC $\in [-3\cdot 2^{k-1}, 3\cdot 2^{k-1}) \subseteq [-3M`/8, 3M`/8)$. Once shifting and addition of A_iB are done, PS + PC$\in [-3M`/4, 7M`/8)$.

  Proof: Assume that PC, PS at the beginning of iteration i is equal to the maximum theoretical value i.e. (PC + PS) = 3M`/8. Since the inputs A and B should be less than M, we can also take into consideration that at iteration i $A_iB$ is equal to its maximum value M`/8 -1. Hence the upper bounds of equation 3.1 are satisfied after executing line 6 as follows,

  $$(PS + PC) = 2*3M`/8 + (M`/8 -1) = (6M`+M`)/8 – 1$$
  $$= 7M`/8 – 1 < 7M`/8$$

  On the other hand, by considering $A_iB = 0$ and (PC+PS) equals to its minimum value, i.e. (PC+PS) = -3M`/8, the execution of line 6 also satisfies the lower bounds of equation 3.1 as follows:

  $$(PS + PC) = -2*3M`/8 + 0 = -6M`/8 = -3M`/4$$

  Thus the conditions for equation 3.1 remains true after executing line 6.

After completing the for-loop, the sum PS + PC $\in [-3M`/4, 7M`/8) \subseteq [-M`, M`)$. The final summation and selection of lines 7 to 9 ensures that the result produced is in $[0, M`) \equiv [0, 8M)$.

Example :

Assume that we need to compute 13 * 14 mod 15.

Let, A = $(13)_{10}$ = $(1101)_2$, B = $(14)_{10}$ = $(1110)_2$, M = $(15)_{10}$ = $(1111)_2$, k = 4 bits

Initialization: M` = 8M = 0111_1000, PS = 0000_0000, PC = 0000_0000, ES = 00000

For loop from i = 3 to -3 is given in Table 3.1.

Assimilation: P = PS + PC = 1001_1000

Correction: P = P + M` = 1001_1000 + 0111_1000 = 0001_0000

Result: P = P/8 = 00_0010 = $(2)_{10}$

Table 3.1 Example of Koc-Hung Modular Multiplication

i	A_i B	ES(PS, PC)	PS, PC
3	0000_1110	0_0000	2PS: 0000_0000 2PC: 0000_0000 A_i B: 0000_1110 PS: 0000_1110 PC: 0000_0000
2	0000_1110	0_0001	2PS: 0001_1100 2PC: 0000_0000 -M`: 1000_1000 A_i B: 0000_1110 PS: 1001_1010[1] PC: 0001_1000
1	0000_0000	1_0110	2PS: 0011_0100 2PC: 0011_0000 +M`: 0111_1000 A_i B: 0000_0000 PS: 0111_1100 PC: 0110_0000
0	0000_1110	1_1011	2PS: 1111_1000 2PC: 1100_0000 +M`: 0111_1000 A_i B: 0000_1110 PS: 1101_1110 PC: 0110_0000
-1	0000_0000	0_0111	2PS: 1011_1100 2PC: 1100_0000 -M`: 1000_1000

[1] It is assumed that a 4:2 compressor is used to sum 2PS + 2PC + A_iB ± M`

			A_i B: 0000_0000 PS: 1111_0100 PC: 0001_0000
-2	0000_0000	0_0000	2PS: 1110_1000 2PC: 0010_0000 A_i B: 0000_0000 PS: 1100_1000 PC: 0100_0000
-3	0000_0000	0_0001	2PS: 1001_0000 2PC: 1000_0000 -M`: 1000_1000 A_i B: 0000_0000 PS: 1001_1000 PC: 0000_0000

3.2 Multiplier-Divider based on Koc-Hung Modular Multiplication

The modular multiplication algorithm presented in the previous section can be slightly modified to also yield the quotient resulting from dividing (A*B) by M. As a result the modified algorithm implements a multiplier-divider which computes (A·B/M) producing both a quotient Q and a remainder R such that A*B = Q*M + R. Since A and B are less than M, the resultant Q will always be less than M. Hence, the maximum size of Q is k-bit long and an overflow cannot possibly occur. Algorithm 3.2 shows the pseudo-code of a multiplier-divider based on Koc-Hung modular multiplication approach.

Algorithm 3.2 Multiplier-Divider Algorithm

```
1. M′ := 8M, RS := 0, RC := 0, ES := 0, Q := 0
2. For i = k - 1, k - 2, …, 0, -1, -2, -3 do
3.  begin
```

```
4.    ES(RS, RC) := $RS_{k+3:k-1} + RC_{k+3:k-1}$
5.    IF  ES(RS, RC) ≥ 1  then
6.       (RS, RC) := 2RS + 2RC + $A_iB$ – M’
7.       Q := 2Q + 1
8.    ELSEIF  ES(RS, RC) ≤ -2  then
9.       (RS, RC) := 2RS + 2RC + $A_iB$ + M’
10.      Q := 2Q - 1
11.    ELSE
12.      (RS, RC) := 2RS + 2RC + $A_iB$
13.      Q := 2Q
14. End
15. R = RS + RC
16. IF R < 0  then
17.    R := R + M’
18.    Q := Q - 1
19. Return R/8, Q.
```

Example

Assume we would like to compute: 15Q + R = 13*14/15

Let, A = $(13)_{10} = (1101)_2$, B = $(14)_{10} = (1110)_2$, M = $(15)_{10} = (1111)_2$, k = 4 bits

Initialization: M` = 8M = 01111000, RS = 00000000, RC = 00000000,

ES = 00000, Q = 0000

For loop from i = 3 to -3,

Table 3.2 Example of Multiplication-Division Based on Koc-Hung Modular Multiplication

i	A_i B	ES(RS, RC)	RS, RC	Q
3	0000_1110	0_0000	2RS: 0000_0000 2RC: 0000_0000 A_i B: 0000_1110 RS: 0000_1110 RC: 0000_0000	0000
2	0000_1110	0_0001	2RS: 0001_1100 2RC: 0000_0000 -M`: 1000_1000 A_i B: 0000_1110 RS: 1001_1010 RC: 0001_1000	0001

1	0000_0000	1_0110	2RS: 0011_0100 2RC: 0011_0000 +M`: 0111_1000 A_j B: 0000_0000 RS: 0111_1100 RC: 0110_0000	0001
0	0000_1110	1_1011	2RS: 1111_1000 2RC: 1100_0000 +M`: 0111_1000 A_j B: 0000_1110 RS: 1101_1110 RC: 0110_0000	0001
-1	0000_0000	0_0111	2RS: 1011_1100 2RC: 1100_0000 -M`: 1000_1000 A_j B: 0000_0000 RS: 1111_0100 RC: 0001_0000	0011
-2	0000_0000	0_0000	2RS: 1110_1000 2RC: 0010_0000 A_j B: 0000_0000 RS: 1100_1000 RC: 0100_0000	0110
-3	0000_0000	0_0001	2RS: 1001_0000 2RC: 1000_0000 -M`: 1000_1000 A_j B: 0000_0000 RS: 1001_1000 RC: 0000_0000	1101

Assimilation: R = RS + RC = 1001_1000

Correction: R = R + M` = 1001_1000 + 0111_1000 = 0001_0000, Q = Q – 1 = 1100

Result: R = R/8 = 00_0010 = $(2)_{10}$, Q = 1100 = $(12)_{10}$

3.3 Online Multiplier-Divider with a Constant Divisor

The multiplier-divider based on the Koc-Hung modulo multiplication presented in the previous section can be converted to a single multiplication-division online module with online inputs A and B and an online quotient output (Q). However, the input divisor D (output remainder R) is assumed to be received (produced) in parallel. Converting the Koc-Hung multiplier-divider to online is performed in 3 stages:

1. Conversion of the multiplication procedure used by Koc-Hung modular multiplication to the online form.
2. Modification of the sign-estimation technique to accommodate changes caused by the online nature of the multiplication procedure.
3. Converting the quotient Q to produce the result in an online manner.

3.3.1 Converting the Multiplication Procedure to Online

The Koc-Hung modular multiplication uses a left-to-right serial-parallel multiplication algorithm to multiply A*B, where A is the serial multiplier and B is the parallel multiplicand. By using this approach, A_iB is added to the partial product and then the partial product is shifted to the left in each clock cycle for k-1 iterations. The final product P is achieved at the kth iteration resulting in,

$$\begin{aligned}P &= a_{k-1}2^{k-1}(2^{k-1}b_{k-1}+2^{k-2}b_{k-2}+\cdots+b_0)+\ \ldots\ +a_1 2(2^{k-1}b_{k-1}+2^{k-2}b_{k-2}+\cdots+b_0)\\ &\quad +a_0(2^{k-1}b_{k-1}+2^{k-2}b_{k-2}+\cdots+b_0)\\ &=\sum_{i=0}^{k-1}\sum_{j=0}^{k-1}a_i b_j 2^{i+j}\end{aligned}$$

where, a_i and b_i are the ith bits of A and B respectively.

As already illustrated in Chapter 2, inputs to an online system are represented in fraction format even though the actual inputs may be integers. In this case, the input operands, A and B are represented as:

$A = \sum_{i=1}^{k} a_i 2^{-i}, B = \sum_{i=1}^{k} b_i 2^{-i}$,

Accordingly, the final product of the serial-parallel multiplication procedure becomes:

$$P \quad = \sum_{i=1}^{k} a_i 2^{-i} * \sum_{j=1}^{k} b_j 2^{-j}$$

$$= \sum_{i=1}^{k} \sum_{j=1}^{k} a_i b_j 2^{-(i+j)} \qquad (3.2)$$

The serial-parallel multiplication scheme, however, would not work when both inputs are online since input B is no longer fully available in each iteration. In order to get a mathematical formulation for multiplying two online inputs we expand equation 3.2 in a visual hierarchy as illustrated in Figure 3.1(a). The rows represent the partial product A_iB for every iteration. Each bit of the 2k-bit product is determined by the sum of all multiplicative pairs found within the same column and any carries generated from prior sums. Figures 3.1-b to 3.1-e, demonstrate the case for online inputs, where only one bit from both inputs are available at a time.

Figure 3.1 (b): In the first iteration i.e. j =1, only the most significant bits of the inputs are received, i.e. a_1 and b_1. Accordingly, the only multiplicative pair that can be processed is $2^{-2}a_1b_1$.

Figure 3.1 (c): in the second iteration the partial product becomes,

$\hat{P} = 2^{-4} (a_2b_2) + 2^{-3} (a_2 b_1 + a_1b_2) = 2^{-3} [a_2(2^{-1} b_2+b_1) + a_1b_2]$

Figure 3.1 (d): in the third iteration the partial product becomes,

$\hat{P} = 2^{-4} [a_3(2^{-2} b_3+2^{-1} b_2+b_1) + b_3(2^{-1}a_2+a_1)]$

Figure 3.1 (e): in the fourth iteration the partial product becomes,

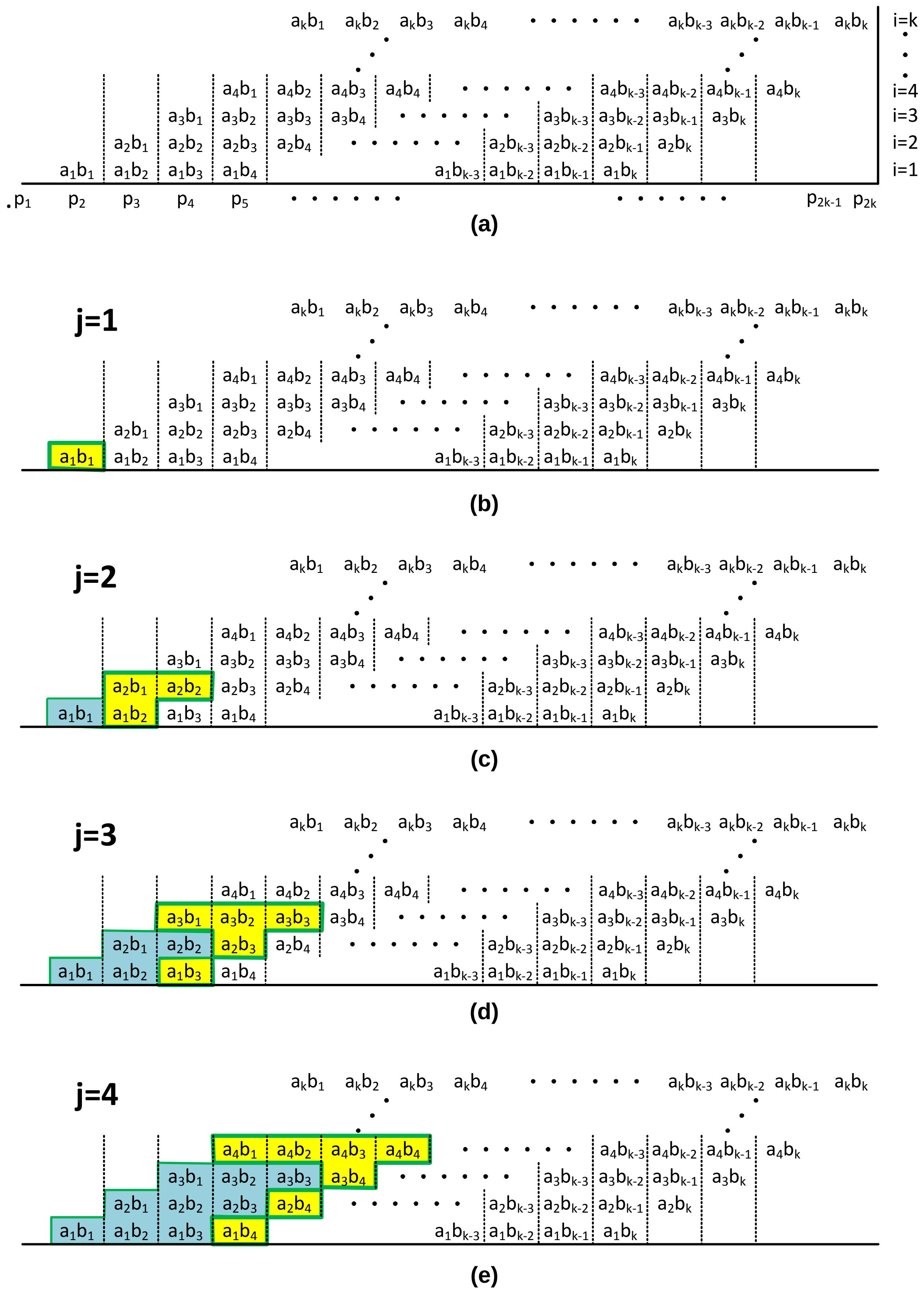

(a)
(b)
(c)
(d)
(e)
j=1
j=2
j=3
j=4
i=k
i=4
i=3
i=2
i=1
p1
p2
p3
p4
p5
p2k-1
p2k
akb1 akb2 akb3 akb4 akbk-3 akbk-2 akbk-1 akbk
a4b1 a4b2 a4b3 a4b4 a4bk-3 a4bk-2 a4bk-1 a4bk
a3b1 a3b2 a3b3 a3b4 a3bk-3 a3bk-2 a3bk-1 a3bk
a2b1 a2b2 a2b3 a2b4 a2bk-3 a2bk-2 a2bk-1 a2bk
a1b1 a1b2 a1b3 a1b4 a1bk-3 a1bk-2 a1bk-1 a1bk

Figure 3.1 Converting parallel-serial multiplication to online by observing available multiplicative pairs during iterations 1-4

$\hat{P} = 2^{-5} [a_4(2^{-3}b_4+2^{-2} b_3+2^{-1} b_2+b_1)+ b_4(2^{-2} a_3+2^{-1}a_2+a_1)]$

In general, the partial product at iteration j becomes:

$$\hat{P} = 2^{-j} [a_j(2^{-j}b_j + \dots + 2^{-2}b_2 + 2^{-1}b_1)+ b_j(2^{-j+1}a_{j-1} + \dots + 2^{-2}a_2 + 2^{-1}a_1)]$$

$$= 2^{-j}\left[a_j\sum_{i=1}^{j} b_i 2^{-i} + b_j\sum_{i=1}^{j-1} a_i 2^{-i}\right] \tag{3.3}$$

Since left-shifting is naturally inherent in the multiplier-divider (Algorithm 3.2), the factor 2^{-j} is dropped from equation 3.3. Thus the multiplicative additive A_iB can be converted to the following online form:

$$\hat{P} = a_j\sum_{i=1}^{j} b_i 2^{-i} + b_j\sum_{i=1}^{j-1} a_i 2^{-i} = a_j B[j] + b_j A[j-1]$$

and the recurrence relation for multiplication with online inputs becomes:

$$P^{(i)} = P^{(i-1)} + 2^{-i}(a_i B[i] + b_i A[i-1]) = 2P^{(i-1)} + a_i B[i] + b_i A[i-1]$$

where, i = 1, 2, … k and the final product P = $P^{(k)}$ for inputs of size k-bits.

3.3.2 Modifying the Sign-Estimation Technique

Since online inputs are represented as fractions, the final product will also be in a fractional form. With the partial product $\hat{P}$ having k+4 bits, the extra 4-bits which include the sign bit are to the left of the binary point. Thus, the partial remainder has the form: $r_3r_2r_1r_0.r_{-1}r_{-2}\dots r_{-k}$ with r_3 being the sign bit. Accordingly, the sign estimate procedure ES(RS,RC) of the partial remainder represented in carry-save form: $\acute{r}_3\ \acute{r}_2\ \acute{r}_1\ \acute{r}_0.\acute{r}_{-1}$. The pseudo-code of the sign estimate procedure is shown in Algorithm 3.3.

Algorithm 3.3 Pseudo-code for the Sign-Estimation Technique

```
[D′ := 8D]
ES(RS, RC) ← $RS_{3:-1}$ + $RC_{3:-1}$
IF ES(RS, RC) = (+) then
  (RS,RC) ← 2RS + 2RC + $\hat{P}$ - D′;
ELSEIF ES(RS, RC) = (-) then
  (RS,RC) ← 2RS + 2RC + $\hat{P}$ + D′;
ELSE (±) then (RS,RC) ← 2RS + 2RC + $\hat{P}$
```

1. **The sign is positive (+).** To detect if ES(RS, RC) is positive, $(RS+RC)_{3:-1} \geq 0.5$. This is almost the same condition as used in Algorithm 3.1, line 4. The difference is that the least significant bit of estimate ES(RS,RC) is a fraction. Hence, the condition that was used by Koc-Hung to determine whether the sign of the estimate is positive, is divided by 2. In this case $RS + RC \geq 2^{-1} > D`/8$, and thus RS + RC is within (D`/8, 7D`/8). After shifting, addition of $a_iB[i] + b_iA[i-1]$ and the subtraction of M, $RS + RC \in (-3D`/4, 7D`/8)$.

2. **The sign is negative (-).** With online inputs, the addition of the partial product can yield a number that is greater than the modulus. To be exact, the maximum value of $\hat{P}$:

$$MAX(a_iB[i]+b_iA[i-1]) = (D - 2^{-k}) + [(D - 2^{-k})-2^{-k}] = 2D - 3\cdot2^{-k}$$

Notice that the modulus is represented as a fraction since the inputs A and B are also fractions. To ensure that the partial remainder RS+RC stays within the range $2^{-k}\cdot[-2^{k-1} < (RS+RC)/8 < 2^{k-1}]$ after shifting, addition of $MAX(a_iB[i]+b_iA[i-1])$ and the addition of D`, a new set of conditions is needed to determine when the estimate ES(RC,RS) is negative. Let x be the value of (RS, RC) before the start of Algorithm 3.3. If x is negative, the following condition should be met:

$$2x + (2D - 3\cdot 2^{-k}) + 8D < 8\cdot 2^{-1} \; = \; 2x < -(2^{2} - 10D + 3\cdot 2^{-k})$$

$$= \; x < -2^{-1}[10D - 2^{2} - 3\cdot 2^{-k}] \qquad (3.4)$$

If we consider that the largest possible Modulus is used, then $D = 1 - 2^{-k}$. In this case, equation 3.4 yields:

$$x < -2^{-1}[10 - 2^{2} - 13\cdot 2^{-k}] = \; x < -2^{-1}[6 - 13\cdot 2^{-k}] \qquad (3.5)$$

Since the estimate ES(RC,RS) is based on the five most significant bits of (RS,RC), equation 3.5 becomes:

$$x < -2^{-1}[(0110.0 + \varepsilon) - 13\cdot 2^{-k}]$$

where, ε is the error of estimating RS+RC when using only the 5 most significant bits.

In the worst case, subtraction of $13\cdot 2^{-k}$ may result in a borrow from bit position 2^{-1}. On the contrary, the estimate error ε may result in an estimate error of $+2^{-1}$. Thus the estimate of equation 3.5 becomes:

$x < -2^{-1}[(0110.0 + 0000.1) - 0000.1] = x < - 0011.0 = x \leq - 0010.1$

3. **The sign is unsure (±).** in this case, it is guaranteed that $RS+RC \in [-3\cdot 2^{k-1}, 3\cdot 2^{k-1})$ $\subseteq [-3D`/8,\ 3D`/8)$. Once shifting and addition of $a_iB[i]+b_iA[i-1]$ are done, $RS + RC \in [-3D`/4,\ 7D`/8)$.

Based on the threshold values that we have selected to estimate the sign of (RC, RS), the value (RC + RS) will always be less than 2^{3} (or equivalently 8D) at the end of each iteration. Thus, no overflow may occur if the estimate of (RC, RS) becomes a large positive number, i.e. (RC, RS) > 0111.1 + ε which may result in a sign change of (RC, RS). To prove that (RC, RS) < 0111.1 at the end of each iteration, we need to determine the maximum value of (RC, RS) can be achieved after shifting, addition of

$a_iB[i] + b_iA[i-1]$ and the addition of M. Let X be defined as the maximum value of (RC, RS) from the previous iteration. Since the addition of M only occurs if the estimate of (RC, RS) from the previous iteration is negative, thus X equals -2.5, which is the threshold value we calculated to determine when (RC, RS) is negative. Hence,

$$\begin{aligned} \mathrm{MAX}(RC, RS) &= 2X + D` + \mathrm{MAX}(a_iB[i]+b_iA[i-1]) \\ &= -5 + 8D + 2D - 3\cdot 2^{-k} = -5 + 10D - 3\cdot 2^{-k} \end{aligned}$$

Since, the maximum value of $D = 1 - 2^{-k}$

$$\therefore \ \mathrm{MAX}(RC, RS) = -5 + 10 - 10\cdot 2^{-k} - 3\cdot 2^{-k} = 5 - 3\cdot 2^{-k}$$

Therefore, MAX(RC, RS) < 0101.0, based on the 5-bit estimate of (RC, RS) at any iteration.

3.3.3 Converting the Quotient to Online

In the multiplier-divider algorithm based on the Koc-Hung modular multiplication technique (Algorithm 3.2), the quotient is accumulated and produced in parallel. In lines 6 and 9 of Algorithm 3.2, the quotient is accumulated in the form of increments and decrements of Q. By adapting Binary signed Digit (BSD) set for the quotient, increments and decrements in each iteration of Algorithm 3.2, directly translates to a quotient digit. Thus, Q+1 increment results in $q_i = +1$, and a decrement of

Q-1, yields $q_i = \bar{1}$. This allows the quotient to be delivered in an online form. The quotient bit q_i is coded with two bits (q_p, q_n) according to Table 3.3.

Table 3.3 Signed bit representation of q_i

q_i	q_p	q_n
0	0	0
1	1	0
$\bar{1}$	0	1

The signed-bit representation ensures a carry-free accumulation of the quotient. Consequently, the quotient increments/decrements do not affect the quotient bits generated from previous clock cycles. Accordingly, the online delay is δ = 3 and the total number of clock cycles = k + 3.

3.3.4 The Online Multiplier-Divider Algorithm for Constant Divisors

By combining the results obtained from the 3 stages of converting the multiplier-divider to online, we develop an overall online multiplier-divider algorithm which is represented in Algorithm 3.4.

Algorithm 3.4. Proposed Online Multiplier-Divider Algorithm with δ = 3

```
(I) Initialization:
              RS, RC ← 0;  ES ← 0; D′ ← 8D;
              VS, VC ← 0;  A[0] ← 0;  B[0] ← 0;

(II) Loop:
```

```
For i = 1, 2, …, k+3  do begin

   a. Shift:  RS ← 2RS; RC ← 2RC;
              A ← A + $a_i \cdot 2^{-i}$;  B ← B + $b_i \cdot 2^{-i}$;
   ---------------------------------------------------------------
   b. Add:    IF ES(RS, RC) ≥ 0.5  then
                  (VS,VC) ← $a_i$·B[i] + $b_i$·A[i-1] + $\overline{D'}$; // k+4 CSA
                  $VC_0$ ← 1;
                  $q_i$ ← 1;
              ELSEIF  ES(RS, RC) ≤ -2.5  then
                  (VS,VC) ← $a_i$·B[i] + $b_i$·A[i-1] + D′;  // k+4 CSA
                  $q_i$ ← $\overline{1}$;
              ELSE  then
                  (VS,VC) ← $a_i$·B[i] + $b_i$·A[i-1];      // k+4 CSA
                  $q_i$ ← 0;
              (RS,RC) ← VS + VC + RC + RS    // [4:2] Compressor
              ES(RS, RC) ← $RS_{k+3:k-1}$ + $RC_{k+3:k-1}$       // 5-bit CPA
   ---------------------------------------------------------------
   c. Output: out ($q_i$)
              IF i = k+3  then
                  return (RS/8, RC/8)
End For
```

Example A*B/D

A = $(53)_{10}$ = $(110101)_2$, B = $(56)_{10}$ = $(111000)_2$, D = $(63)_{10}$ = $(111111)_2$, k = 6 bits

Initialization: D` = 8D = 0111.111000, RS = 0000.000000, RC = 0000.000000,

ES = 0000.0,

For loop from i = 1 to 9 is shown in Table 3.4.

Table 3.4 Example of the proposed online multiplier-divider algorithm for constant divisors

i	**a_i**	**b_i**	**A[i-1]**	**B[i]**
1	1	1	0000.000000	0000.100000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	0000.0	a_iB[i]: 0000.100000 b_iA[i-1]: 0000.000000 0 : 0000.000000 VS: 0000.100000 VC: 0000.000000	2RS: 0000.000000 2RC: 0000.000000 VS: 0000.100000 VC: 0000.000000 RS: 0000.100000 RC: 0000.000000	0

2	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	1	1	0000.100000	0000.110000
	ES(RS, RC)	**VS, VC**	**RS, RC (Add)**	**q_i**
	0000.1	a_iB[i]: 0000.110000 b_iA[i-1]: 0000.100000 $\overline{D}$` : $\underline{1000.000111}$ VS: 1000.010111 VC: 0001.000001	2RS: 0001.000000 2RC: 0000.000000 VS: 1000.010111 $\underline{\text{VC: 0000.000000}}$ RS: 1010.010100 RC: 0000.000100	1
3	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	0	1	0000.110000	0000.111000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1010.0	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.110000 $\underline{\text{+D`: 0111.111000}}$ VS: 0111.001000 VC: 0001.100000	2RS: 0100.101000 2RC: 0000.001000 VS: 0111.001000 $\underline{\text{VC: 0001.100000}}$ RS: 1001.011000 RC: 0100.000000	$\bar{1}$
4	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	1	0	0000.110000	0000.111000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1101.0	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.111000 $\underline{\text{+D`: 0111.111000}}$ VS: 0111.000000 VC: 0001.110000	2RS: 0010.110000 2RC: 1000.000000 VS: 0111.000000 $\underline{\text{VC: 0001.110000}}$ RS: 1011.100000 RC: 1000.000000	$\bar{1}$
5	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	0	0	0000.100000	0000.111000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	0011.1	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.000000 $\overline{D}$` : $\underline{1000.000111}$ VS: 1000.000111 VC: 0000.000001	2RS: 0111.000000 2RC: 0000.000000 VS: 1000.000111 $\underline{\text{VC: 0000.000001}}$ RS: 1111.000100 RC: 0000.000100	1

6	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	1	0	0000.110000	0000.111000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1111.0	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.111000 0 : 0000.000000 VS: 0000.111000 VC: 0000.000000	2RS: 1110.001000 2RC: 0000.001000 VS: 0000.111000 VC: 0000.000000 RS: 1110.101000 RC: 0000.100000	0
7	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	0	0	0000.000000	0000.000000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1111.0	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.000000 0 : 0000.000000 VS: 0000.000000 VC: 0000.000000	2RS: 1101.010000 2RC: 0001.000000 VS: 0000.000000 VC: 0000.000000 RS: 1100.010000 RC: 0010.000000	0
8	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	0	0	0000.000000	0000.000000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1110.0	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.000000 0 : 0000.000000 VS: 0000.000000 VC: 0000.000000	2RS: 1000.100000 2RC: 0100.000000 VS: 0000.000000 VC: 0000.000000 RS: 1100.100000 RC: 0000.000000	0
9	**a_i**	**b_i**	**A[i-1]**	**B[i]**
	0	0	0000.000000	0000.000000
	ES(RS, RC)	**VS, VC**	**RS, RC**	**q_i**
	1100.1	a_iB[i]: 0000.000000 b_iA[i-1]: 0000.000000 +D`: 0111.111000 VS: 0111.111000 VC: 0000.000000	2RS: 1001.000000 2RC: 0000.000000 VS: 0111.111000 VC: 0000.000000 RS: 1100.111000 RC: 0100.000000	$\bar{1}$

The resultant quotient becomes: $Q = 1\bar{1}\bar{1}1\_000\bar{1} = 10\_1111 = (47)_{10}$

The remainder is obtained in parallel at iteration k+3:

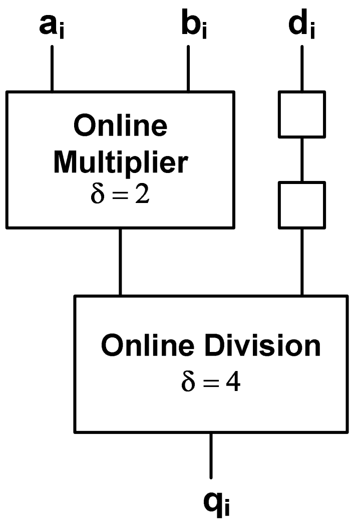
$R = RS/8+RC/8 = 110\_0111 + 010\_0000 = 000\_0111 = (7)_{10}$

3.4 Online Multiplier-Divider Algorithm with an Online Divisor Input

In the previous algorithm, we presented an online scheme for performing multiplication-division with online multiplier and multiplicand inputs while the input modulus is received in parallel. However, such a scheme may not be suitable for applications with a fully online environment where the divisor is received online rather than in parallel. For such cases, a realization of an algorithm that performs Q = A*B/D is needed, where all input operands A, B and D are online digit-serial inputs.

3.4.1 Cascaded Implementation of a Fully Online Multiplier-Divider

The conventional method for performing the fully online multiplier-divider operation is by cascading single-operation online modules. Such a network would consist of an online multiplication unit (section 2.4) followed by an online division unit (section 2.5). The online multiplication unit computes a signed 2k-bit product after an online delay of $\delta = 2$ clock cycles. Once the product bits are generated, the product bits are then delivered to the online division unit. Since the product bits are produced after 2 clock cycles, the divisor d_i should be delayed by two clock cycles such that the online division unit receives properly synchronized product and divisor input bits. The online division then produces the quotient after a delay of 4 clock cycles. Overall, the online latency for the entire multiplier-divider operation is δ = 6 and the total number of clock cycles to produce the quotient is k + 6 cycles. The hardware implementation for performing conventional online multiplication-division is shown in Figure 3.2.



3.2 Architecture of the Conventional Fully Online Multiplier-Divider

3.4.2 Proposed Fully Online Multiplier-Divider Algorithm Based on composite Algorithms

The conventional cascaded multiplier-divider, despite its simplicity, requires a large amount of hardware (detailed analysis of the hardware and timing performance are presented in Chapter 4 and 5). In this section, we present an alternative implementation for a fully online multiplier-divider which reduces the amount of hardware required by combining several operations used in the online multiplication unit and the online division unit into a single composite unit. Thus, we propose a composite online multiplier-divider algorithm based on new online multiplication-division recurrence relation.

The recursive recurrence relation used by online multiplication (P = A*B) described in Chapter 2 is:

$$\dot{W}^{(j)} = 2(\dot{W}^{(j-1)} - p_{j-1}) + A[j]b_j + a_j B[j-1]$$

Similarly, the recurrence relation used by online division (Q = N/D) is:

$$\ddot{W}^{(j)} = 2(\ddot{W}^{(j-1)} - q_{j-1}D[j-1]) + n_j 2^{-4} - d_j Q[j-1]2^{-4} \tag{3.6}$$

Notice that, if the output result of the online multiplication unit is delivered to the online division unit, then n_j in equation 3.6 represents bit j of the 2k-bit product P. To produce n_j, the online multiplication algorithm uses a selection function to find an estimated signed-bit value from the partial residue $\dot{W}^{(j)}$. Instead of estimating the value of the partial residue, the online multiplication selection function can be dropped and the entire value of the multiplicative partial product i.e. $A[j]b_j + a_j B[j-1]$ can be used to replace n_j. Thus, the recurrence relation used by online multiplication can be combined with the online division recursive relation by replacing n_j with the partial product. Since shifting of the residue function is already inherent in the online division recurrence relation, the shifting of the previous state of the online multiplication residue function $2\dot{W}^{(j-1)}$ is dropped and is not dded to n_j.

Proof: Assume that we use the online multiplication recurrence equation in parallel which does not require estimating product bit p_j using an estimating function. The online multiplication recurrence becomes:

$$\dot{W}^{(j)} = 2\dot{W}^{(j-1)} + A[j]b_j + a_j B[j-1]$$

At iteration,

j = 1: $\dot{W}^{(1)} = A[1]b_1 + a_1 B[0]$

j = 2: $\dot{W}^{(2)} = 2A[1]b_1 + 2a_1 B[0] + A[2]b_2 + a_2 B[1]$

j = k:

$\dot{W}^{(k)} = A[k]b_k + a_k B[k-1] + 2A[k-1]b_{k-1} + a_{k-1}B[k-2] + \ldots + 2^{k-1}A[1]b_1 + 2^{k-1}a_1 B[0]$

$$= \sum_{j=1}^{k} 2^{k-j}(A[j]b_j + a_j B[j-1]) \tag{3.7}$$

By merging the online multiplication residue function with the online division residue function without the online multiplication shifting operation we get:

$$W^{(j)} = 2W^{(j-1)} - 2q_{j-1}D[j-1] + 2^{-4}(b_j A[j] + a_j B[j-1]) - d_j Q[j-1]2^{-4}$$

Let, $\alpha_j = -2q_{j-1}D[j-1] - d_j Q[j-1]2^{-4}$

Hence W$^{(j)}$ becomes:

$$W^{(j)} = 2W^{(j-1)} + 2^{-4}(b_j A[j] + a_j B[j-1]) + \alpha_j$$

At iteration,

j = 1: $W^{(1)} = 2^{-4}(A[1]b_1 + a_1 B[0]) + \alpha_1$

j = 2: $W^{(2)} = 2^{-4}(A[2]b_2 + a_2 B[1]) + 2^{-4}(2A[1]b_1 + 2a_1 B[0]) + \alpha_2 + 2\alpha_1$

j = k: $W^{(k)} = \sum_{l=1}^{k} \alpha_l 2^{k-l} + 2^{-4}\left[\sum_{j=1}^{k} 2^{k-j}(A[j]b_j + a_j B[j-1])\right]$

By substituting equation 3.7 into iteration k, we get:

$$W^{(k)} = \sum_{l=1}^{k} \alpha_l 2^{k-l} + 2^{-4}\dot{W}^{(k)}$$

Therefore, the shifting operation of $2\dot{W}^{(j-1)}$ could be dropped and the dividend bit n_j replaced by $A[j]b_j + a_j B[j-1]$ to convert the online division to an online multiplication-division recurrence equation. The resultant recurrence relation for an online multiplier-divider is given by:

$$W^{(j)} = 2(W^{(j-1)} - q_{j-1}D[j-1]) + 2^{-4}(b_j A[j] + a_j B[j-1]) - d_j Q[j-1]2^{-4} \qquad (3.8)$$

Since the remainder (R) equals to W after the last iteration, equation 3.8 can be represented as:

$$R^{(j)} = 2(R^{(j-1)} - q_{j-1}D[j-1]) + 2^{-4}(b_j A[j] + a_j B[j-1]) - d_j Q[j-1]2^{-4}$$

where, $R = R^{(k+\delta)}$

The number of bits representing the residue function $R^{(j)}$ and the online delay remains the same as in online division and are unaffected by changes made to the recurrence relation. Therefore, the online delay of the fully online multiplication-division algorithm is $\delta = 4$ and the total number of clock cycles to compute the quotient is k + 4. Further, similar to online division, the most significant bit of the divisor must be 1, thus $0.5 \leq D < 1$. The corresponding algorithm for a fully online multiplier-divider computing Q = A*B/D is given in Algorithm 3.5.

The residue function, the partial product and the internal variables are represented as a k+6 bit carry-sum pairs in the form of (RC, RS), (PC, PS) and (VC, VS). Two of the (k+6)-bits are to the left of the binary point; one to account the left shift operation and the other is a sign bit. When subtracting $2^{-4}d_i \cdot \overline{Q[i-1]}$ from (PC, PS) we added an extra precondition when Q[i-1]! = 0 to avoid an unnecessary addition of all 1's by the carry-save adder. The addition of all 1's when Q=0 could result in producing inaccurate results when the intended outcome should be 0 such as 0.01*0.00/0.11.

When $q_i = 1$, $q_i \cdot D[i]$ is subtracted from (VC, VS) by adding the inverse of D[i] i.e. $\overline{D[i]}$ with (VC, VS) using a carry-save adder and setting the least significant bit of RC to 1. i.e. $RC_{-k-4} = 1$. This would avoid using extra hardware for an inverted on-the-fly converter to determine (-D[i]) in 2's complement format as we accumulate new values of d_i in each clock cycle.

A 5-bit estimate of V called $\hat{V}$ is produced by summing the 5 most-significant bits of VC and VS. Based on $\hat{V}$, a selection function similar to online division is used to produce q_i. Since the boundaries of the selection function are from $\hat{V}$ = (-½ to ¼], only 4-bits of $\hat{V}$ is needed to select q_i. At the end of the Add stage in Algorithm 3.5, we store the quotient bits q_i in binary representation using an on-the-fly conversion which is denoted as CA. Finally, the remainder of the multiplication-division is determined by delivering the contents of (RC, RS) at the last clock cycle.

Algorithm 3.5 Proposed Fully Online Multiplier-Divider Algorithm with δ = 4

```
(I) Initialization:
              RS, RC ← 0;  PS, PC ← 0; VS, VC ← 0;
              D[0] ← 0; A[0] ← 0; B[0] ← 0; Q[0] ← 0;

(II) Loop:
For i = 1, 2, …, k+4  do begin

   a. Shift:  RS ← 2RS; RC ← 2RC;
              A ← A + $a_i \cdot 2^{-i}$;  B ← B + $b_i \cdot 2^{-i}$; D ← D + $d_i \cdot 2^{-i}$;
   b. Add:   (PC,PS) ← RS + RC + $2^{-4} a_i \cdot B[i]$ + $2^{-4} b_i \cdot A[i-1]$
                                                        -- 4:2 Compressor
```

```
            IF Q[i-1]≠0 && d_i≠0 then
               (VS,VC) ← PS + PC + $2^{-4}d_i\cdot\overline{Q[i-1]}$   -- k+6 bit CSA
               $VC_{-k-4}$ ← 1
            ELSE (VS,VC) ← PS + PC
            $\hat{V} = VS_{1:-3} + VC_{1:-3}$                  -- 5-bit CPA
            IF i > 4 && $\hat{V}_{1:-2} > 1/4$ then
               $q_i$ ← 1
            ELSE IF i > 4 && $\hat{V}_{1:-2} < -1/2$ then
               $q_i$ ← $\overline{1}$
            ELSE  $q_i$ ← 0;
            (RS, RC) ← VS + VC - $q_i\cdot D[i]$        -- k+6 bit CSA
   c. Output:
            Q ← CA(Q[i-1],$q_i$)            -- on-the-fly conversion
            out ($q_i$)
End For
(III) Remainder:
            Return (RS, RC)
```

Example A*B/D

A = $(53)_{10} = (110101)_2$, B = $(56)_{10} = (111000)_2$, D = $(63)_{10} = (111111)_2$, k = 6 bits

Initialization: RS = 0000.000000, RC = 0000.000000, Q = 0.000000,

The details of the For loop from i = 1 to 10 is shown in Table 3.5

Table 3.5 Example of the proposed fully online multiplication-division algorithm using composite online algorithms

i	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]
1	1	0.000000	1	0.100000	1	0.100000	00.000	0	0.000000
	PS, PC			**VS, VC**			**RS, RC**		
	2RS: 00.0000000000			PS: 00.0000100000			VS: 00.0000100000		
	2RC: 00.0000000000			PC: 00.0000000000			VC: 00.0000000000		
	$2^{-4}a_iB[i]$: 00.0000100000			$-2^{-4}d_i\overline{Q[i-1]}$: 00.0000000000			$-q_iD[i]$: 00.0000000000		
	$2^{-4}b_iA[i-1]$: 00.0000000000			VS: 00.0000100000			RS: 00.0000100000		
	PS: 00.0000100000			VC: 00.0000000000			RC 00.0000000000		
	PC: 00.0000000000								

2	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]
	1	0.100000	1	0.110000	1	0.110000	00.001	0	0.000000
	PS, PC			VS, VC			RS, RC		
	2RS: 00.0001000000 2RC: 00.0000000000 $2^{-4}a_iB[i]$: 00.0000110000 $2^{-4}b_iA[i-1]$: 00.0000100000 PS: 00.0000010000 PC: 00.0010000000			PS: 00.0000010000 PC: 00.0010000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 00.0010010000 VC: 00.0000000000			VS: 00.0010010000 VC: 00.0000000000 $-q_iD[i]$: 00.0000000000 RS: 00.0010010000 RC 00.0000000000		
3	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]
	0	0.110000	1	0.111000	1	0.111000	00.010	0	0.000000
	PS, PC			VS, VC			RS, RC		
	2RS: 00.0100100000 2RC: 00.0000000000 $2^{-4}a_iB[i]$: 00.0000000000 $2^{-4}b_iA[i-1]$: 00.0000110000 PS: 00.0101010000 PC: 00.0000000000			PS: 00.0101010000 PC: 00.0000000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 00.0101010000 VC: 00.0000000000			VS: 00.0101010000 VC: 00.0000000000 $-q_iD[i]$: 00.0000000000 RS: 00.0101010000 RC 00.0000000000		
4	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]
	1	0.110000	0	0.111000	1	0.111100	00.101	0	0.000000
	PS, PC			VS, VC			RS, RC		
	2RS: 00.1010100000 2RC: 00.0000000000 $2^{-4}a_iB[i]$: 00.0000111000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 00.1011011000 PC: 00.0000000000			PS: 00.1011011000 PC: 00.0000000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 00.1011011000 VC: 00.0000000000			VS: 00.1011011000 VC: 00.0000000000 $-q_iD[i]$: 00.0000000000 RS: 00.1011011000 RC 00.0000000000		
5	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]
	0	0.110100	0	0.111000	1	0.111110	01.011	1	0.100000
	PS, PC			VS, VC			RS, RC		
	2RS: 00.1011011000 2RC: 00.0000000000 $2^{-4}a_iB[i]$: 00.0000000000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 01.0110110000 PC: 00.0000000000			PS: 01.0110110000 PC: 00.0000000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 01.0110110000 VC: 00.0000000000			VS: 01.0110110000 VC: 00.0000000000 $-q_iD[i]$: 11.0000011111 RS: 10.0110101111 RC 10.0000100001		
6	a_i	A[i-1]	b_i	B[i]	d_i	D[i]	$\hat{V}$	q_i	Q[i]

	1	0.110100	0	0.111000	1	0.111111	00.111	1	0.110000
	PS, PC			**VS, VC**			**RS, RC**		
	2RS: 00.1101011110 2RC: 00.0001000010 $2^{-4}a_iB[i]$: 00.0000111000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 00.1100010100 PC: 00.0011000100			PS: 00.1100010100 PC: 00.0011000100 $-2^{-4}d_iQ[i-1]$: 11.1111011111 VS: 11.0000001111 VC: 01.1110101001			VS: 11.0000001111 VC: 01.1110101001 $-q_iD[i]$: 11.0000001111 RS: 01.1110101001 RC 10.0000011111		
7	a_i	**A[i-1]**	b_i	**B[i]**	d_i	**D[i]**	$\hat{V}$	q_i	**Q[i]**
	0	0.110101	0	0.111000	0	0.111111	11.110	0	0.110000
	PS, PC			**VS, VC**			**RS, RC**		
	2RS: 11.1101010010 2RC: 00.0000111110 $2^{-4}a_iB[i]$: 00.0000000000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 11.1101101100 PC: 00.0000100100			PS: 11.1101101100 PC: 00.0000100100 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 11.1101001000 VC: 00.0001001000			VS: 11.1101001000 VC: 00.0001001000 $-q_iD[i]$: 00.0000000000 RS: 11.1100000000 RC 00.0010010000		
8	a_i	**A[i-1]**	b_i	**B[i]**	d_i	**D[i]**	$\hat{V}$	q_i	**Q[i]**
	0	0.110101	0	0.111000	0	0.111111	11.110	0	0.110000
	PS, PC			**VS, VC**			**RS, RC**		
	2RS: 11.1000000000 2RC: 00.0100100000 $2^{-4}a_iB[i]$: 00.0000000000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 11.1100100000 PC: 00.0000000000			PS: 11.1100100000 PC: 00.0000000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 11.1100100000 VC: 00.0000000000			VS: 11.1100100000 VC: 00.0000000000 $-q_iD[i]$: 00.0000000000 RS: 11.1100100000 RC 00.0000000000		
9	a_i	**A[i-1]**	b_i	**B[i]**	d_i	**D[i]**	$\hat{V}$	q_i	**Q[i]**
	0	0.110101	0	0.111000	0	0.111111	11.100	$\bar{1}$	0.101110
	PS, PC			**VS, VC**			**RS, RC**		
	2RS: 11.1001000000 2RC: 00.0000000000 $2^{-4}a_iB[i]$: 00.0000000000 $2^{-4}b_iA[i-1]$: 00.0000000000 PS: 11.1001000000 PC: 00.0000000000			PS: 11.1001000000 PC: 00.0000000000 $-2^{-4}d_iQ[i-1]$: 00.0000000000 VS: 11.1001000000 VC: 00.0000000000			VS: 11.1001000000 VC: 00.0000000000 $-q_iD[i]$: 001111110000 RS: 11.0110110000 RC 01.0010000000		
10	a_i	**A[i-1]**	b_i	**B[i]**	d_i	**D[i]**	$\hat{V}$	q_i	**Q[i]**
	0	0.110101	0	0.111000	0	0.111111	01.000	1	0.101111

	PS, PC	VS, VC	RS, RC
	2RS: 10.1101100000	PS: 00.1001100000	VS: 00.0001100000
	2RC: 10.0100000000	PC: 00.1000000000	VC: 01.0000000000
	$2^{-4}a_iB[i]$: 00.0000000000	$-2^{-4}d_iQ[i-1]$: 00.0000000000	$-q_iD[i]$: 11.0000001111
	$2^{-4}b_iA[i-1]$: 00.0000000000	VS: 00.0001100000	RS: 10.0001101111
	PS: 00.1001100000	VC: 01.0000000000	RC 10.0000000001
	PC: 00.1000000000		

3.5 Correction Stage

So far we have presented three different techniques for online multiplication-division. The first technique performs multiplication-division provided that the Modulus (Divisor) is available in parallel. The second technique performs a conventional fully online multiplication-division using an online multiplier followed by an online divider. The third technique performs a fully online multiplication-division operation using composite algorithms. All three algorithms produce a 2k-bit result that consists of a combined quotient and reminder. The remainder can be either delivered in parallel at the last iteration as described in Algorithms 3.4 and 3.5, or produced as subsequent quotient bits q_i using the selection function for an additional k clock cycles. Such an approach is however shadowed by two main problems:

1. It is possible that the remainder delivered at the last iteration is negative. Although the remainder is within range $-M < R < M$, a negative remainder may not be desired in certain applications.
2. The k-bit quotient Q is dependent on the remainder. If the remainder is negative, the actual k-bit quotient is 2^{-k} less than the generated k-bit quotient. This problem, however, will not arise if the quotient and the remainder are delivered by q_i using k

additional clock cycles. For instance, if one of the online multiplication-division algorithms produced a 10-bit quotient $Q_{out} = 10010\_0\bar{1}010$ using 5-bit inputs. If the 5-bit quotient Q is taken independently then Q = 10010 and $R = 0\bar{1}010$. Q in this case would be incorrect since the actual Q is 10001. But since Q is dependent on R, we can only get the correct 5-bit value of Q after converting Q_{out} from signed bit representation to binary.

In order to produce a k-bit quotient Q that is independent of the remainder and to deliver the remainder within the range $0 \leq R < M$, a correction step is needed. The correction step may be used at the last clock cycle for all online multiplication-division implementations. A theoretical description of the correction step algorithm is given in lines 15 to 18 in Algorithm 3.2. To integrate the correction step with the online algorithms, specifically Algorithms 3.4 and 3.5, the following challenges need to be investigated:

1. Quotient correction by decrementing the quotient in the last clock cycle if the remainder is negative.
2. Accurately detecting the sign of the remainder and correcting the remainder

3.5.1 Quotient Correction

In Algorithms 3.4 and 3.5, the quotient Q is produced in an online manner by generating one quotient bit q_i in each iteration. In case of a correction step, if the remainder is negative at the last iteration, q_i will then need to be decremented by 1 before being delivered to the output. However, if $q_i = \bar{1}$, then decrementing q_i will result in a 2-bit value of $\bar{2}$. Since we can only output a one bit value at a time, an intermediate variable is needed to temporarily accumulate the values of q_i. We define this intermediate value as the partial quotient denoted as $\hat{Q}$ to accumulate the increments and decrements of Q during each iteration. The maximum number of bits of $\hat{Q}$ is determined by the maximum number of quotient increments/decrements that may occur at any clock cycle. This limitation holds true as long as the values of the shifted out bits of $2\hat{Q}$ will not be changed due to the quotient increments/decrements at any subsequent clock cycle. Therefore, the size of $\hat{Q}$ can be limited to 2 bits.

To ensure that no overflow may occur when accumulating the quotient increments/decrements a secondary selection function is needed. When iterating through the for-loops of Algorithms 3.4 and 3.5, we shift $\hat{Q}$ and may either add 1 or $\bar{1}$ and thus overflow can never occur. However in the last iteration, there is a possibility that $\bar{2}$ could be added to $\hat{Q}$. If the partial quotient from the previous iteration is $\hat{Q} = \bar{1}$, then after shifting $\hat{Q}$ and adding $\bar{2}$, an overflow occurs since $\hat{Q}$ should be only 2 bits. For that reason, whenever $\hat{Q} = 0\bar{1}$, we change $\hat{Q}$ to an equivalent representation as $\hat{Q} = \bar{1}1$. The resulting selection function to determine the corrected quotient bits ($\hat{q}_i$) becomes:

$$\hat{q}_i = \begin{cases} 1 & if \quad \hat{Q} > 1 \\ \bar{1} & if \quad \hat{Q} \leq -1 \\ 0 & otherwise \end{cases}$$

$\hat{q}_i$ is then subtracted from $\hat{Q}$ before passing $\hat{Q}$ to the next iteration. Notice that by having the size of $\hat{Q}$ = 2-bits, an extra clock cycle is needed to output the last quotient bit. As a result, the online delay of the online multiplier-divider algorithms increases by 1 due to the correction step.

3.5.2 Remainder Sign Detection and Correction

Based on the correction stage algorithm, assimilation of the carry-sum pair of R is required to get the full precision of the remainder and accurately determine the actual sign of R. This addition may have a delay of order O(n) if carry-propagation is used. Since the speed of the online multiplier-divider algorithms depends heavily on the constant delay of carry-save adders, the assimilation of R will consequently have an adverse effect on the critical path of the circuit. So as to maintain a minimum delay within all operations, the correction stage for the online multiplier-divider algorithms constitutes the following:

1. The remainder is corrected using a carry-save adder and delivered at the last iteration in carry-save representation without assimilation. Thus the remainder output remains in the form of RS and RC. For conventional and composite online multiplication-division, if the remainder is negative during the last iteration, the correction is performed by the following addition:

   $$(RC, RS) := RC + RS + D$$

2. Instead of using a carry-propagate adder to determine the sign of (RC, RS), a Brent-Kung Adder is used. The Brent-Kung Adder [26] technique is used to

determine when a carry-out is generated if $(RS, RC)_{L-2:0}$ are added together, where L is the size of R depending upon the online multiplier-divider algorithm used. The carry bit is then added to $(RS_{L-1} + RC_{L-1})$ by using an XOR gate to determine the actual sign of (RC, RS). The Brent-Kung Adder has a delay which is of order $O(\log_2 L)$. A detailed discussion of the Brent-Kung Adder follows.

The Brent-Kung Adder:

The Brent- Kung adder is a type of carry-lookahead adder that uses a generalized technique to determine when a carry is generated (known as *generate*), and when a carry generated by a previous bit/block can be propagated (known as *propagate*) [26]. A generate signal, G_i equals 1 when both inputs of a full adder stage, x_i and y_i are 1, i.e $G_i = x_i \cdot y_i$. On the other hand, the propagate P_i equals 1 when $x_i y_i = 01$ or $x_i y_i = 10$., i.e. $P_i = x_i + y_i$. Accordingly, the Boolean expression for the carry-out can be written as:

$$C_{i+1} = x_i \cdot y_i + C_i(x_i + y_i) = G_i + C_i \cdot P_i \qquad (3.8)$$

Figure 3.3(a), shows the generate and propagate circuit diagram for the initial adders input.

In 1982, Brent and Kung introduced a generalized P_i and G_i [27]:

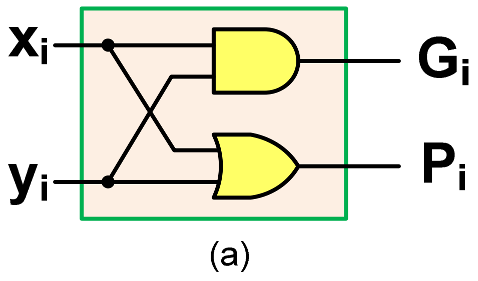
1. A composite propagate, P_α^β : is a carry that can propagate between stages α and β of the adder. If $P_\alpha^\beta = 1$, then $C_{\beta-1}$ is propagated to C_α i.e. $C_\alpha = C_{\beta-1}$ by propagating through the adder stages that produce $G_\beta, G_{\beta+1}, \ldots, G_\alpha$ and $P_\beta, P_{\beta+1}, \ldots, P_\alpha$. The composite propagate is defined as:

$$P_\alpha^\beta = P_\alpha \cdot P_{\alpha-1} \cdot \ldots P_\beta$$

2. A composite generate, G_α^β : a carry generated between stages α and β of the adder. If $G_\alpha^\beta = 1$, then a carry was generated within a sequence of adder stages

that produce G_β, $G_{\beta+1}$, …, G_α and P_β, $P_{\beta+1}$, …, P_α, and propagated to the output. The composite generate is defined as:

$$G_\alpha^\beta = G_\alpha + P_\alpha G_{\alpha-1}^\beta$$



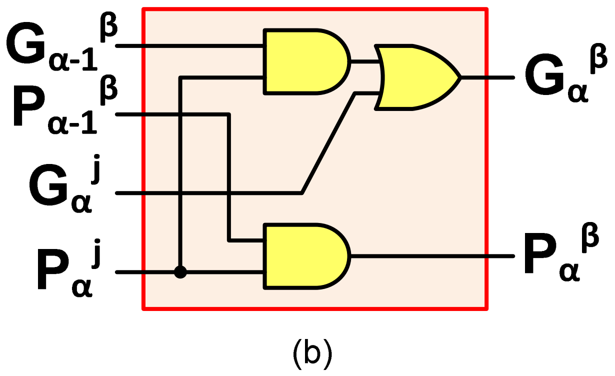


Figure 3.3 a) G, P signals from initial adder inputs. (b) Generate-Propagate Operator Unit

Figure 3.3(b), shows a Generate-Propagate Operator (GPO) unit that merges between two adjacent G and P pairs. By using the G_i and P_i signals produced by the initial adder inputs combined with GPO units, an 8-bit architecture of the Brent-Kung carry-lookahead adder is shown in Figure 3.4.

The area complexity of the Brent-Kung Adder:

- Area of the G_i, P_i unit: 2 gates
- Area of the GPO unit: 3 gates

- Total Area for L-bit R = L•(2) + (L-1)•(3) = 5L-3 gate

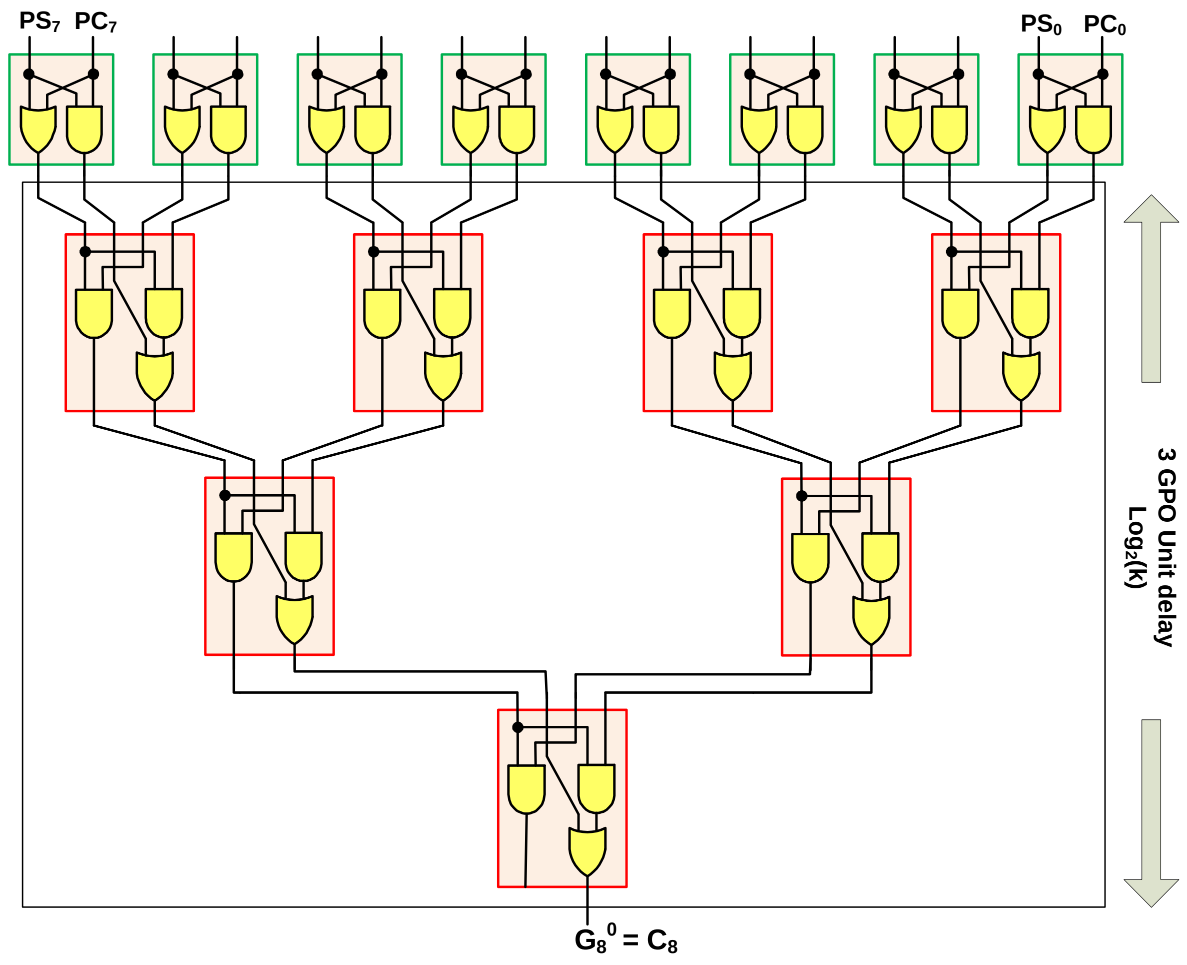


Figure 3.4 Brent-Kung Adder architecture for finding the carry-out when adding two numbers

For the sake of comparison, if a Carry-Propagate Adder (CPA) is used, the total area becomes,

(Area of full-Adder) * L = 5L gates

The delay of the Brent-Kung adder:

- Delay of the G_i, P_i unit: 1 gate
- Delay of the GPO unit: 2 gate

- Total Delay for L-bit R = $1+2\lceil \log_2 L \rceil$ gate delays

In comparison, the delay of a CPA is: 2L gate delays

Example of generating the carry-out of 48-bit addition:

- Delay of Brent-Kung Adder = $1+2\lceil 5.58 \rceil = 13$ gate delay
- Delay of CPA adder = 2*48 = 96 gate delays

3.5.3 Correction Stage Algorithm

To conclude this section, the algorithm for the correction step required for all of the online multiplication-division implementations is summarized in Algorithm 3.6.

Algorithm 3.6 Correction Stage Algorithm

```
(I) Initialization:   $\hat{Q} \leftarrow 0$;
-----------------------------------------------------------------
(II) Loop:
For i = 1, 2, …, k+δ+1  do begin
   a. Shift: $\hat{Q} \leftarrow 2\hat{Q}$;
   --------------------------------------------------------------
   b. Add:  $\hat{Q} \leftarrow \hat{Q} + q_i$;                 -- 2-bit sign adder
            IF i = k+δ then
               carry ← $RS_{L-2:0}$ + $RC_{L-2:0}$; -- (L-1)bit Brent-Kung Adder
               sign ← $RS_{L-1} \oplus RC_{L-1} \oplus$ carry;          -- XOR gate
               IF sign = 1 then
                  (RS,RC) ← RS + RC + D[i];          -- L-bit CSA
                  $\hat{Q} \leftarrow \hat{Q} + \bar{1}$;
            IF $\hat{Q} > 1$ then $\hat{q}_i \leftarrow 1$
            ELSE IF $\hat{Q} \leq -1$ then $\hat{q}_i \leftarrow \bar{1}$
            ELSE $\hat{q}_i \leftarrow 0$
            $\hat{Q} \leftarrow \hat{Q} - 2\hat{q}_i$;
   --------------------------------------------------------------
   c. Output:
            out ($\hat{q}_i$);
            IF i = k+δ then
```

```
                out (RC, RS);
End For
```

The **sign** signal in Algorithm 3.6 is a single bit signal that indicates if the remainder is negative. Also, the loop of the correction stage iterates concurrently with the internal loop of Algorithms 3.4 and 3.5 . Thus the correction stage only uses one additional clock cycle for the overall online multiplication-division operation.

CHAPTER 4

HARDWARE IMPLEMENTATION & SYNTHESIS RESULTS

This chapter covers the hardware implementation for the online multiplication-division algorithm with constant divisors (Algorithm 3.4), the conventional cascaded online multiplication-division and the fully online multiplication-division algorithm based on composite algorithms (Algorithm 3.5). The hardware implementations were modeled in the Verilog hardware description language. Theoretical analysis of area and delay costs are presented and compared to obtained synthesis results. Details of the hardware models and testbenches are given in Appendix A.

4.1 Theoretical Analysis

In order to perform a technology-independent cost analysis of the online multiplication-division algorithms an area and delay cost models are adapted [28]. The area and timing costs of these basic components are given in Table 4.1. The area and costs are normalized to the area and delay costs of simple basic 2-input gates [28], e.g. AND, NAND, OR or a NOR gate. The area and delay costs are assumed to be g and μ respectively and other modules area and delay costs are given in terms of these basic quantities (Table 4.1).

To derive technology-independent area and delay estimates for the multiplier-divider hardware implementation, the following assumptions are made:

- A 4-to-1 MUX has twice the area and delay of a 2-to-1 MUX
- The delay of the **U**-block = delay of an 8-to-1 MUX = 6μ
- The area of an n-bit CPA/CSA consists of n Full-Adders (FA)
- The delay of an n-bit CPA = (n-1)*(delay of 2 simple gates) + (delay of FA)
- The delay of an n-bit CSA = delay of a single Full Adder
- The area of an n-bit optimized [4:2] compressor[1] = 2*(area of n-bit CSA)
- The delay of an n-bit optimized [4:2] compressor = 6μ
- The delay of an on-the-fly (CA) unit = delay of a 2-to-1 MUX = 1.5μ

Table 4.1 Area cost g (gate equivalent) and delay μ of the basic components

component	Area (g)	Delay (μ)
Simple gates	1	1
XOR/XNOR	2	2
2-to-1 MUX	1.5	2
Flip-Flop	4	2
Full Adder	7	4

[1] The optimized [4:2] compressor reported in [29] has the same area as a normal [4:2] compressor but has a 25% shorter delay

4.1.1 Online Multiplier-Divider with Constant Divisor (Algorithm 3.4)

The data path of the online multiplication-division (Algorithm 3.4) based on the Koc-Hung modular multiplication algorithm is shown in Figure 4.1. In total the hardware implementation requires a k+4 carry-save adder (CSA), a k+4 [4:2] compressor, two 2-to-1 multiplexers, a 4-to-1 multiplexer, two k-bit registers (Reg), two k+4 bit register, a 5-bit carry-propagate adder (CPA) and a sign estimation unit ES(RC, RS) which can be implemented using an encoder. The two k-bit registers are used to accumulate the values received for the online inputs a_j and b_j. The two 2-to-1 multiplexers are used for the multiplication of $a_j \cdot B[j]$ and $b_j \cdot A[j-1]$. Depending on the sign estimation of ES(RC, RS), the 4-to-1 multiplexer selects the addition of D, $\overline{D}$ or 0 to $a_j \cdot B[j] + b_j \cdot A[j-1]$using the k+4 carry-save adder. The k+4 [4:2] compressor then adds (VC, VS) with the shifted values of (RC, RS). The 5-bit CPA is used to compute $RS_{k+3:k-1} + RC_{k+3:k-1}$. The ES(RS, RC) block decides whether the 5-bit sum from the CPA is negative, positive or undecided. The ES(RS, RC) block also produces the ouptut quotient bit q_j. (RC, RS) are finally stored in the k+4 –bit registers.

The area analysis of Algorithm 3.4 is calculated based on Table 4.1 as follows:

- area of the two k-bit Reg: 8k g
- area of k+4 –bit Reg: (4k+16) g
- area of k+4 CSA: (7k+28) g
- area of k+4 [4:2] Adder: (14k+56) g
- area of the two 2-to-1 MUX (implemented as AND-gates since one input is 0): 2k g

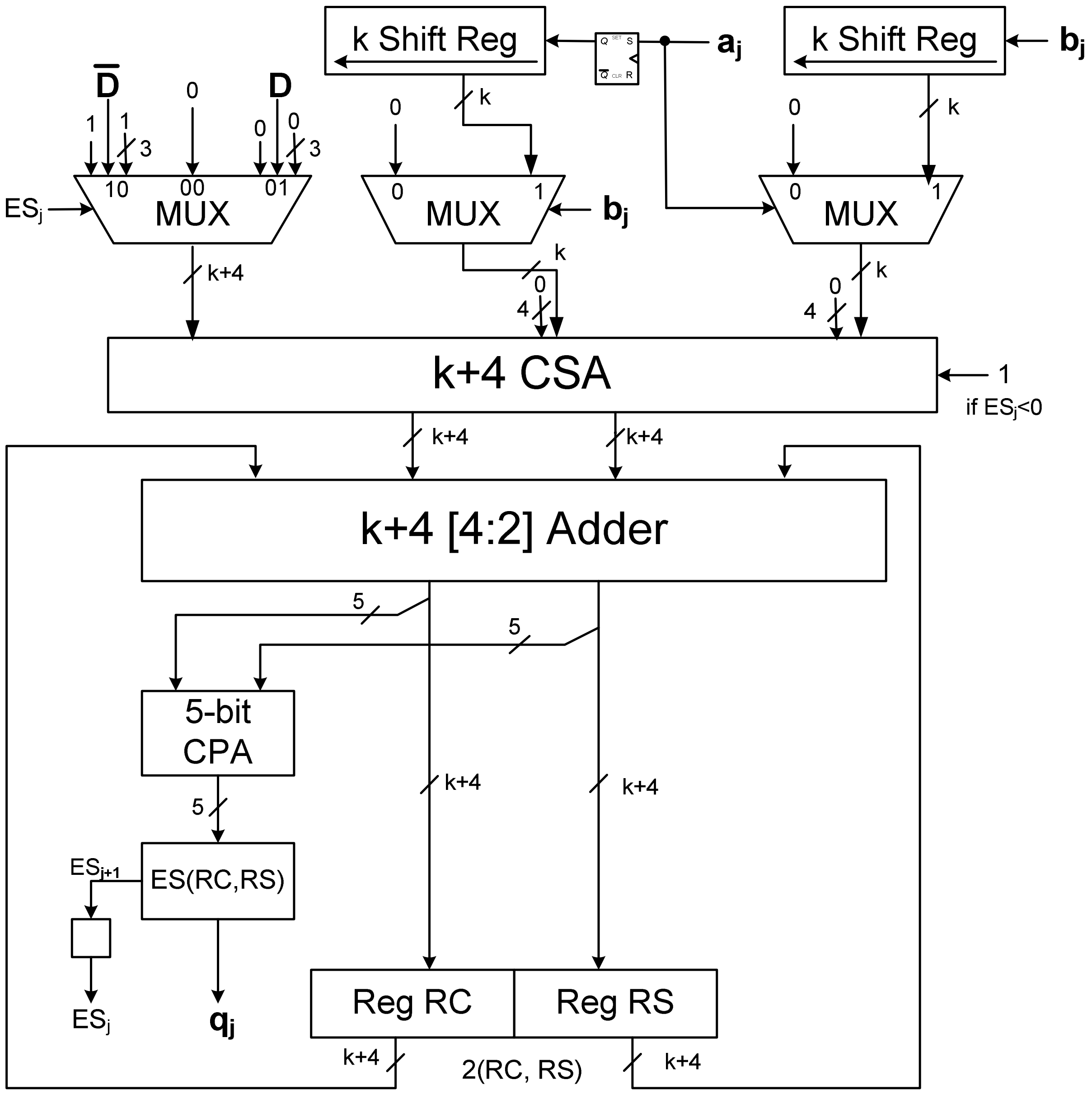


Figure 4.1 Path of the radix-2 online multiplication division with constant input divisor

- area of k+4 bit 4-to1 MUX: (3k+12) g

- area of 5-bit CPA: 35 g

- area of ES(RC, RS): 10g

Total area complexity = (38k + 157) g ≈ 38k g

Delay Analysis of the above circuit can be calculated based on Table 4.1:

- delay of 4-to-1 MUX: 4 μ

- delay of k+4 bit CSA: 4 μ

- delay of k+4 bit [4:2] Adder: 6 μ

- delay of 5-bit CPA: 12 μ

- delay of ES(RC, RS) unit: 4 μ

- delay of registers = delay of flip-flop: 2 μ

Total path delay of Algorithm 3.4: 32 μ.

4.1.2 Cascaded Online Multiplier-Divider

There are two possible implementations for the conventional online multiplier-divider algorithm. The first implementation which was represented earlier in Figure 3.2, the online multiplication unit and the online division unit are used in sequence without any intermediate delay. The second implementation uses an extra single bit flip-flop for pipelining the online multiplication unit and the online division unit as shown in Figure 4.2.

4.1.2.1 Implementation 1

In this implementation the online product bits produced by the online multiplication unit are sent directly to the online division unit yielding an online delay of $\delta = 6$. The critical path of such an implementation is the sum of the critical paths of the online multiplication unit and the online division unit. In total, the hardware implementation of the online multiplication unit and online division unit requires a k+4 bit [4:2] compressor, two 2k+2 bit CSA's, five k-bit 2-to-1 MUX's, a k-bit 4-to-1 MUX, a 6-bit 4-to-1 MUX (U block), two 1-bit 4-to-1 MUX's (c_d and c_q), five k-bit registers, two k+4 bit registers, an 8-bit Reg (CA for n_j), two 2k+2 bit registers, a 4-bit CPA, a 5-bit CPA and two selection function units.

Area analysis for implementation 1:

- area ofthe two 2k+2 CSA: (28k+28) g
- area of the k+4 bit [4:2] compressor: (14k+28) g
- area of the five k-bit Reg: 20k g
- area of five k-bit 2-to-1 MUX: 7.5k g
- area of three k-bit 2-to-1 MUX (implemented as AND-gates since one input is 0): 3k g
- area of k bit 4-to-1 MUX: 3k g
- area of 6-bit 4-to-1 MUX: 18 g
- area of two 1-bit 4-to-1 MUX: 6 g
- area of two k+4 –bit Reg: 8k+32 g
- area of two 2k+2 –bit Reg: 16k+16 g

- area of 8-bit Reg: 32 g
- area of 5-bit CPA: 35 g
- area of 4-bit CPA: 28 g
- area of **Selm**: ~10g
- area of **Seld**: ~10g

Total area complexity = (99.5k + 243) g ≈ 99.5k g

Delay analysis for implementation 1:

- delay of 2-to-1 MUX (an AND gate is assumed to be used): 1 μ
- delay of k+6 bit [4:2] Compressor: 6 μ
- delay of 4-bit CPA: 10 μ
- delay of **Selm** unit: 4 μ
- delay of the division **U**-block: 6 μ
- delay of 2k+2 bit CSA: 4 μ
- delay of 5-bit CPA: 12 μ
- delay of **SELD** unit: 4 μ
- delay of the 4-to-1 MUX: 4 μ
- delay of 2k+2 bit CSA: 4 μ
- delay of the registers: 2 μ

Total path delay: 57 μ

4.1.2.2 Implementation 2

The critical path of conventional online multiplication-division can be reduced by using an additional flip-flop to pipeline the stages between the online multiplication and the online division. The flip-flop temporarily stores the product bits produced by the online multiplication unit for one clock cycle before passing the product to the online division unit. Thus the critical path of the convention online multiplication-division depends upon the delay of the online unit that has the maximum delay path. Since the online division unit has a longer path, the clock period will be limited by the critical path of the division unit. However, the addition of the flip-flop between the single-operation online units increases the latency by 1 clock cycle. Thus, the online delay of the conventional online multiplication-division using this implementation becomes $\delta = 7$. Besides the extra flip-flop, the area complexity remains the same as in implementation 1.

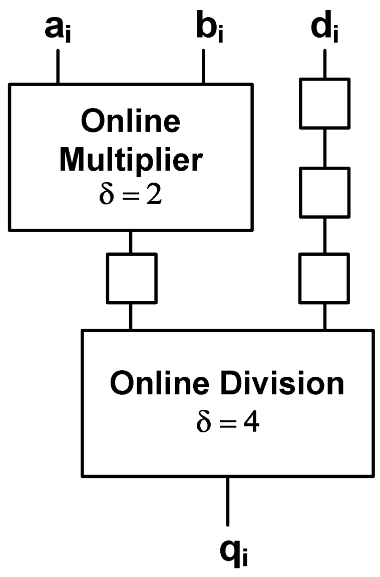


Figure 4.2 Alternative implementation of the conventional radix-2 fully online multiplication-division.

Theoretical analysis of the path delay for implementation 2 can be determined by calculating the delay of an online division operation as follows:

- delay of the division **U**-block: 6 μ

- delay of 2k+2 bit CSA: 4 µ
- delay of 5-bit CPA: 12 µ
- delay of **SELD** unit: 4 µ
- delay of the 4-to-1 MUX: 4 µ
- delay of 2k+2 bit CSA: 4 µ
- delay of the registers: 2 µ

Total path delay: 36 µ

4.1.3 Fully Online Multiplier-Divider Using Composite Algorithms

The data-path implementation of the Fully Online Multiplier-Divider Using Composite Algorithms (Algorithm 3.5) is shown in Figure 4.3. Three k-bit registers are needed to store the values of the online input bits of operands A, B and the divisor D. Three 2-to-1 multiplexers are needed for the bit-by-vector multiplication of $a_j \cdot B[j]$, $b_j \cdot A[j-1]$ and $d_j \cdot \overline{Q[j-1]}$. A k+6 bit [4:2] compressor and a k+6 carry-save adder is then used to sum $a_j \cdot B[j]$, $b_j \cdot A[j-1]$ and $d_j \cdot \overline{Q[j-1]}$ with the values of (RC, RS) from the previous iteration to produce (VC, VS). A 4-bit estimate of (VC, VS) is then produced at block **V** using a 5-bit carry-propagate adder. The 4-bit estimate of (VC, VS) is then passed to the selection function block labeled **Selm** to generate the online quotient output q_j. The q_j bits are stored in signed 2's complement format using a k-bit on-the-fly converter (**CA**). As for the binary signed bit multiplication of $q_j \cdot D[j]$, a 4-to-1 multiplexer is required to produce D[j], 0 or $\overline{D[j]}$ depending on q_j. (VC, VS) is then added with -q_i·D[i] to produce (RC, RS) using a k+6 bit carry-save adder. In total, the implementation of Algorithm 4 requires five k-bit registers (two k-bit registers for CA), two k+6 bit registers, two k+6 bit CSA, a k+6 bit [4:2] compressor, 5-bit CPA, four k-bit 2-to-1 MUX (two k-bit MUX's for CA), a k+6 bit 2-to-1 MUX, a k+6 bit 4-to-1 MUX and a selection function.

By using the area of a 2-input gate as a baseline called g, a theoretical analysis of the area complexity of Algorithm 4 can be calculated as follows:

- area of the five k-bit Reg: 20k g
- area of two k+6 –bit Reg: (8k+48) g
- area ofthe two k+6 CSA: (14k+84) g
- area of k+6 [4:2] Adder: (14k+84) g
- area of two k-bit 2-to-1 MUX (can be implemented as AND-gates since one input is 0): 2k g
- area of two k-bit 2-to-1 MUX (for **CA**): 3k g
- area of a k+6 bit 2-to-1 MUX (can be implemented as AND-gates since one input is 0): (1k+6) g
- area of k+6 bit 4-to-1 MUX: (3k+18) g
- area of 5-bit CPA (**V**): 35 g
- area of **Selm**: ~10g

Total area complexity = (65k + 285) g ≈ 65k g

A theoretical analysis of the critical path of Algorithm 3.5 can also be calculated in terms of the 2-input gate delay μ as follows:

- delay of 2-to-1 MUX (an AND gate is assumed to be used): 1 μ

- delay of k+6 bit [4:2] Compressor: 6 μ

- delay of k+6 bit CSA: 4 μ

- delay of 5-bit CPA: 12 μ

- delay of **Selm** unit: 4 μ

- delay of 4-to-1 MUX: 4 μ

- delay of k+6 bit CSA: 4 μ

-delay of registers: 2 μ

Total path delay of Algorithm 3.5 : 37 μ

4.1.4 The Correction Stage

In Chapter 3, we presented an analysis and hardware design of the correction stage. The main operation of the correction stage is to determine the sign of the remainder represented as a carry-sum pair by using a Brent-Kung Adder. The $\log_2(k)$ binary-tree architecture (Figure 3.4) of the Brent-Kung Adder represented a challenge to create a parameterized hardware model in Verilog. Thus a C++ code was used to create a verilog module called "signDetect.v" based on a user-defined input for the size of the online multiplication-division operands. The C++ code is attached in Appendix B with instructions on using the compiled program.

To demonstrate the importance of the correction step, we modeled and simulated the online multiplication-division algorithm with parallel modulus input (Algorithm 3.4) using the correction stage algorithm (Algorithm 3.6) incorporated within the design. Details of the Verilog model and testbench simulation files are given in Appendix A.

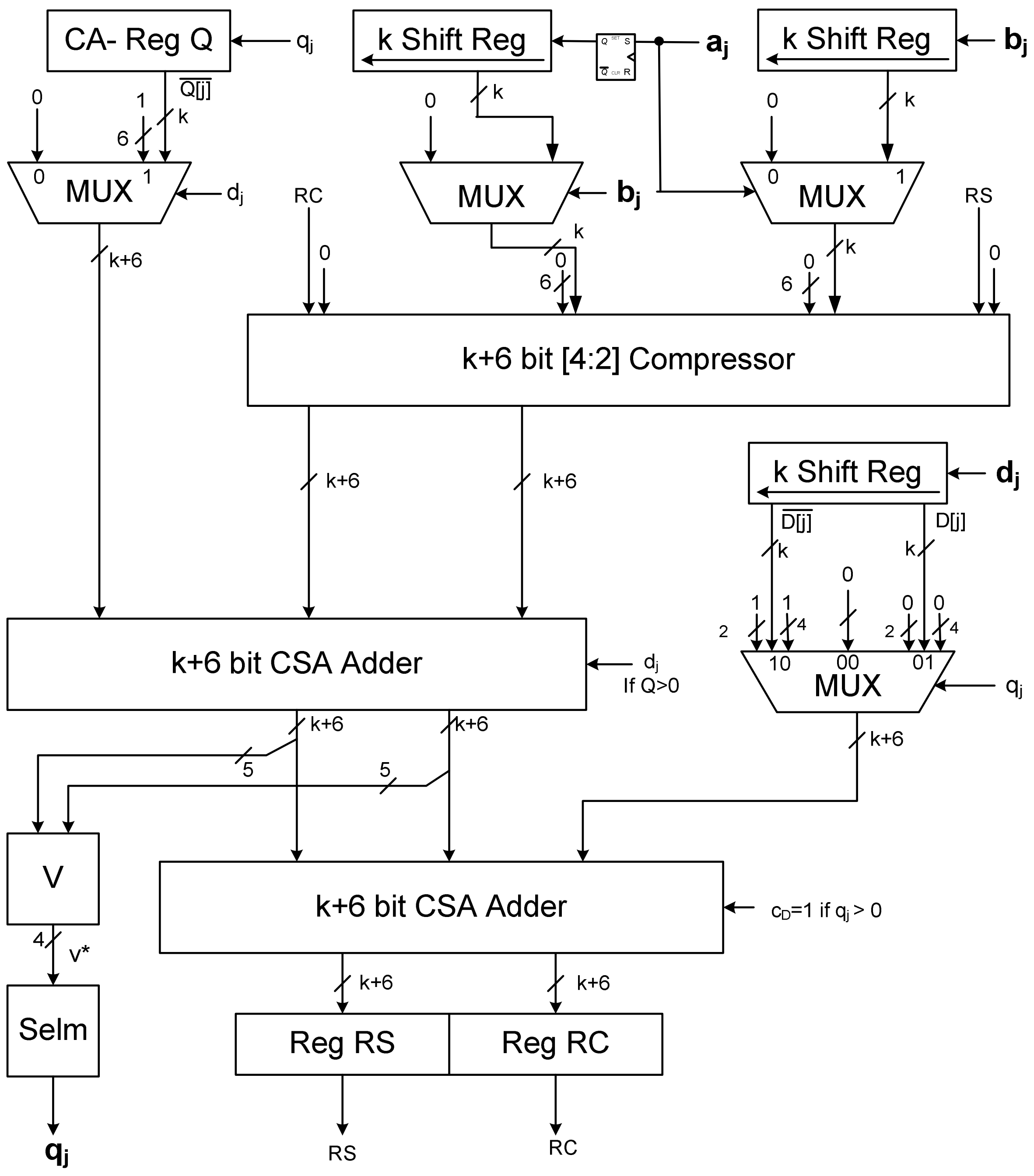


Figure 4.3 Architecture of the radix-2 fully online multiplication division using composite algorithms.

4.2 Synthesis Results and Performance Comparison

The developed Verilog models of the online multiplier-divider algorithm with constant divisor (Algorithm 3.4), the conventional cascaded online multiplier-divider approach and the fully online multiplier-divider using the composite recurrence relations (Algorithm 3.5) were synthesized using Xilinx ISE 8.1 onto a Virtex-4 XC4VSX35 FPGA. The synthesis area and delay results of the three algorithms at different operand sizes (k) are shown in Table 4.2. The area complexity is measured in terms of the number of slices utilized by each algorithm. The synthesis report for the three algorithms at k = 64 bit is given in Appendix C.

Table 4.2 Delay and area synthesis results of online multiplicaiton-division algorithms

	Online Multiplier-Divider with constant divisor		Conventional Cascaded Online Multiplier-Divider			Online Multiplier-Divider with Composite algorithms	
k	Delay	Area (slices)	Delay (Impl. 1)	Delay (Impl. 2)	Area (slices)	Delay	Area (slices)
8	7.904ns	64	17.54ns	9.590ns	186	11.622ns	145
16	8.085ns	103	17.657ns	9.676ns	322	11.412ns	275
32	8.291ns	195	18.672ns	10.408ns	613	12.210ns	506
64	8.653ns	379	20.014ns	11.339ns	1199	12.976ns	771
128	10.690ns	744	22.602ns	12.008ns	2350	15.279ns	1530
256	11.230ns	1478	22.921ns	11.784ns	4703	15.678ns	3008
512	11.660ns	2945	26.61ns	15.036ns	9698	15.996ns	5852
1024	11.949ns	5874	30.172	18.309ns	19538	17.561ns	12136

The total time and area required to complete a 64 bit online multiplication-division operation using the three algorithms is summarized in Table 4.3. The total time indicates the time required to complete a 64-bit online multiplication-division operation.

Table 4.3 Summary of the total time and area for the online multiplication-division algorithms for k = 64 bits

Algorithm	Online Delay	Clock Period	Total Time	Area (slices)
Online Multiplier-Divider with constant divisor	δ = 3	8.653ns	0.58μs	379
Cascaded Online Multiplier-Divider (Implementation 1)	δ = 6	20.014ns	1.4 μs	1199
Cascaded Online Multiplier-Divider (Implementation 2)	δ = 7	11.339ns	0.805 μs	1199
Online Multiplier-Divider with Composite algorithms	δ = 4	12.976ns	0.88 μs	771

CHAPTER 5

CONCLUSION AND FUTURE WORK

One of the major contributions in this work is the development of the first fully online multiplication-division algorithm. This novel algorithm computes Q = A*B/D given that A, B, D and Q are all online inputs and outputs. The algorithm consumes about 40% less area compared to a conventional approach for computing online multiplication-division using a sequence of independent online multiplication and division operations. The proposed algorithm also has a smaller online delay of $\delta = 4$ compared to the conventional cascaded method which requires an online delay of either 6 or 7 depending upon the type of hardware implementation used.

For applications where the input divisor (D) is a constant, we have proposed a second novel online multiplication-division algorithm that computes Q=A*B/D, given that A, B and Q are online inputs/outputs with the divisor (D) being a constant or an input which is available in parallel. This algorithm has an online delay of only $\delta = 3$ and requires approximately 70% less hardware compared to the conventional cascaded online multiplication-division algorithm. The algorithm also has at least 24% less total path delay compared to the conventional method.

Since all online division and multiplication-division algorithms may require a correction stage at the last clock cycle, we have also proposed a generic high-speed correction stage that can be integrated with any online division or multiplication-division algorithm. For k-bit input operands, the proposed correction stage has a total path delay which is O(log k). In contrast, conventional methods for implementing the correction stage usually have a total path delay of O(k).

To verify the correctness of the proposed algorithms, we have modeled and simulated the online multiplication-division based on composite algorithm and the online multiplication-division with constant divisor algorithm in Verilog. The algorithms have worked flawlessly under exhaustive simulation which covers all possible combinations for A, B and D using k = 4, 6 and 8 bit inputs. As for verifying the correction stage, a separate Verilog model was developed for the online multiplication-division with constant divisor algorithm with the correction stage integrated. Successful operation has also been proven through exhaustive simulation with k = 4, 6, 8 and 10 bits.

Performance comparisons between the proposed algorithms and the conventional method for online multiplication-division were performed using both theoretical analysis and by synthesizing the Verilog models using a Virtex-4 Xilinx FPGA. The hardware synthesis results are summarized in Table 4.2.

The following are some possible future directions for further research related to this work:

- The hardware costs of the proposed online multiplication-division algorithms can be reduced by using bit-slice or systolic hardware architectures.

- A formulation of a high-radix algorithm based upon the proposed online multiplication-division algorithms is needed to further improve the total time required to complete the online multiplication-division operation.
- Possible improvements to the speed of the correction stage by pipelining the addition unit as reported in [30].
- Using the proposed online multiplication-division algorithms for possible online implementation of adaptive filters such as the Normalized Mean Least Squares filter. The speed performance of the adaptive filter can then be evaluated and compared to conventional implementations of adaptive filters.

Appendix A

Verilog Code

A.1 Online Multiplication-Division with Constant Divisors

To efficiently model the algorithm, separate Verilog sub-modules were created for both the CSA and the [4:2] compressor. The Verilog files are attached in Appendix A. Each of the Verilog files calls a common file called 'common.v'. This file specifies the N parameter defining the size of the input bits. The main module is called 'onlineKoc'. The test bench file called onlineKocTB.v can be used for exhaustive simulation by covering all possible input values for A and B, or it can be used to test the algorithm using single pre-defined inputs for A and B.

A.1.1 Synthesizable RTL Code

Listing 1: common.v

```
//number of bits of the input operands and the modulus
`define N 64
//Number of bits for the clock counter
`define m 6
```

Listing 2: csa.v

```
`include “common.v”

//Carry-Save Adder
module CSA(A,B,C,FS,FC,cin);

/****************************************************
inputs and outputs of CSA
* A, B, C inputs operands of compressor
* cin carry-in bit input
* FS,FC output in carry-save representation
*/
input [`N+3:0] A, B, C;
output [`N+3:0] FS, FC;
input cin;

//internal 84orrection84
reg [`N+3:0] YS;
integer j;
reg [`N+4:0] YC;

//internal assignment
assign FS = YS;
assign FC = YC[`N+3:0];

//c,s = A+B+C
always@(A or B or C or cin)
begin
YC[0]=cin;
for(j=0;j<`N+4;j=j+1)
    begin
       {YC[j+1],YS[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: compressor2.v

```
`include “common.v”

//Generic N+4 bit 4:2 compressor
module compressor(A,B,C,D,FS,FC, cin, cin2, cout);

/****************************************************
inputs and outputs of 4:2 Compressor
* A, B, C, D inputs operands of compressor
* cin, cin2 are carry-in inputs
* FS,FC output in carry-save representation
* cout carry-out bit
*/
input [`N+3:0] A, B, C, D;
output [`N+3:0] FS, FC;
output cout;
input cin, cin2;

//declaration of intermediate values
reg [`N+3:0] YS, sum;
integer i,j;
reg [`N+4:0] carry, YC;

//internal assignments
assign FS = YS;
assign FC = YC[`N+3:0];
assign cout=carry[`N+4]^YC[`N+4];

//adds D+c+s = C,S using CSA
always@(sum or D or carry or cin2)
begin
YC[0]=cin2;
for(i=0;i<`N+4;i=i+1)
    begin
       {YC[i+1],YS[i]}=D[i]+carry[i]+sum[i];
    end
end

// add A+B+C = c,s  using CSA
always@(A or B or C or cin)
begin
carry[0]=cin;
for(j=0;j<`N+4;j=j+1)
    begin
       {carry[j+1],sum[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 4: onlinekoc.v

```
`include "common.v"

module onlineKoc(clk, reset, x, y, n, q, q_n, RS, RC);

/*****************************************************
inputs and outputs of the online modular 86orrection86ion
* x,y inputs operands
* n modulus
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* RS,RC remainder output in carry-save representation
* q,q_n online output bits of Q
*/
input clk, reset;
input x,y;
input [`N-1:0] n;
output q,q_n;
output [`N+3:0] RS,RC;

/*************************
internal declarations
***********************/
//Vectors X[i] and Y[i] for accumulating x,y
reg [`N-1:0] X, Y;
//x*Y[i] and y*X[i-1]
wire [`N-1:0] xY, yX;
//converting inputs x,y to fractional inputs
wire [`N-1:0] 86orrection, 86orrection, 86orre;
reg [`N-1:0] shifter;
wire [`N-2:0] zeroes;
//sign estimate
wire [4:0] ES;
reg [4:0] ES1;
//residue function
wire [`N+3:0]  PS, PC, VS, VC;
reg [`N+3:0] PS1, PC1, add;
//iteration counter
reg [`m:0] counter;
//intermediate variables
reg g,g_n;
wire negflag1, dummy;
//8M and -8M
wire [`N+3:0] n_p, n_n;


//creating constants and shifting the modulus
assign zeroes =0;
assign n_p = {1'b0,n,3'b000};
assign n_n = {1'b1,~n,3'b111};

//all registers and flip-flops used are assigned here
always@(posedge clk or posedge reset)
begin
//reseting all flip-flops
if (reset) begin
       PS1<=0;
       PC1<=0;
       counter<=1;
       ES1<=0;
       Y<=0;
       X<=0;
       shifter<={1'b1,zeroes};
       end
else
     begin
     //shift the partial residue to the left for the next clock cycle
       PS1<={PS[`N+2:0],1'b0};
       PC1<={PC[`N+2:0],1'b0};

     //counter to keep track the number of iterations
       counter<=counter+1;

     //saving the estimate for the next clock cycle
       ES1<=ES;

     //storing all previous values of the input 87orrecti X,Y in a shift
register
       Y<=87orre;
       X<=X|87orrection;
       shifter <= {1'b0,shifter[`N-1:1]};
       end
end

//storing the inputs in fractional format since all internal operations are
done
//in fractions
assign 87orrection = x? shifter:0;
assign 87orrection = y? shifter:0;

//multiplying x_i*Y[i]  and y_i*X[i-1]
assign 87orre = Y|87orrection;
assign xY = x? 87orre: 0;
assign yX = y? X: 0;

//The sign estimate procedure
always@(ES1 or n_n or n_p)
begin
  if(ES1>=1 && ES1 <=15)
   begin
       add = n_n;
       g = 1; g_n=0;
   end
  else if(ES1>=16 && ES1<=27)
   begin
       add = n_p;
       g = 0; g_n=1;
   end
  else
   begin
       add = 0;
       g = 0; g_n=0;
   end
end

assign negflag1 = g;

//sum xy+yX +/- M +PS+PC using a compressor and CSA
```

```
CSA adder1(.A({4’b0000,xY}),.B({4’b0000,yX}),.C(add),.FS(VS),.FC(VC),
.cin(negflag1));
compressor compressor1(.A(VC),.B(PS1),.C(VS),.D(PC1),.FS(PS),.FC(PC),
.cin(1’b0), .cin2(1’b0), .cout(dummy));

//find the estimate of PS,PC
assign ES = PS[`N+3:`N-1] + PC[`N+3:`N-1];

//output the remainder after shifting back to the correct positions
assign RS = PC;
assign RC = PS;

//output the quotient bits
assign q = g;
assign q_n = g_n;

endmodule
```

A.1.2 Testbench

Listing 5: onlinekocTB.v

```
`include “common.v”

//Test Bench for online multiplication-division for parallel input divisor
module onlinemultdividerTB;

reg clk,rst;
reg x,y;
reg [`N-1:0] n;
wire q, q_n;
reg [`N+2:0] pos, neg;
reg [`N-1:0] D;
wire [`N+3:0] RS, RC;
reg [2*`N-1:0] C;
integer i,j, error;
reg [`N+3:0] temp1,temp2;
reg [5:0] m;

//recieve and store the values of q from the online koc multiplier-divider
//storing the positive and negative values in seperate registers
always@(posedge clk or posedge rst)
begin
if (rst) begin
 pos <= 0; neg <= 0; end
else
  begin
  pos<= {pos[`N+1:0],q};
  neg<= {neg[`N+1:0],q_n};
  end
end

//simulate clock signal with period of 20 time steps
always #10 clk<=~clk;

//initialize all signals
initial begin
clk=0;
rst=0;
m = `N-1;
i=0;
j=0;
error=0;

n=131; // <- set modulus N here

C=0;
D=0;
x=0;
y=0;
end

/****************
//This section of the code is used for brute-force testing
// De-comment this section when needed
initial begin
```

```
for(i=0;i<=n-1;i=i+1)
begin
  for(j=i;j<=n-1;j=j+1)
  begin
    C=j*i;
    D=C/n;
    #10 rst=1;
  @(posedge clk) x=i[7]; y=j[7];    #10; rst =0;
  @(posedge clk) x=i[6]; y=j[6]; // #10; rst =0;
  @(posedge clk) x=i[5]; y=j[5];  // #10; rst =0;
  @(posedge clk) x=i[4]; y=j[4]; // #10; rst =0;
  @(posedge clk) x=i[3]; y=j[3]; //  #10; rst =0;
  @(posedge clk) x=i[2]; y=j[2];
  @(posedge clk) x=i[1]; y=j[1];
  @(posedge clk) x=i[0]; y=j[0];
  @(posedge clk) x=0; y=0;
  #80; temp1=RS; temp2=RC;
if((D –(pos-neg))) begin
       $display(“%d * %d /%d = %b,   RS=%b, RC=%b”,i[`N-1:0],j[`N-1:0],n,(pos-
neg), temp1, temp2); //RS,RC);
        error=error+1;
        end
   #10 rst=1;
   #20 rst=0;
   #100;
   end
end
if(!error) $display(“No errors. All 2^%d * 2^%d  combinations are
successful!”,m,m);
else $display(“Correction Stage Required!  Number of Errors = %d”, error);
***********/

/**/
//Testing using selective numbers:
initial begin
#10 rst=1;
@(posedge clk) x=1; y=1; #10; rst =0;
#5; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=1; y=1;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=1; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=1;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
```

```
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=1;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk)  x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,yX=%b,xY=%b”,multdivider.ES,multdivider.yX,multdivider.xY);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
q,multdivider.q_n);

#20; $display(“A:101110 * B:011010 / N:%b = Q:%b”,n,(pos-neg));
/**/

#200 $finish;
end

onlineKoc multdivider(.clk(clk), .reset(rst), .x(x), .y(y), .n(n), .q(q),
.q_n(q_n), .RS(RS), .RC(RC));

endmodule
```

A.2 Conventional Fully Online Multiplication-Division Based on Single Operation Online Algorithms

Implementation 1 was also modeled in hardware using Verilog. Separate Verilog top modules were created for the online multiplication unit and the online division unit. The top module for the online multiplication is called “onlineMult.v”. The top module calls an instance of the sub-module called “compressor.v” which implements a k+4 bit [4:2] compressor. The online multiplication Verilog files were tested using the testbench file “multb.v”. On the other hand, the top module for the online division unit is called “onlineDiv.v”. The top module calls instances of the sub-modules “Ublock.v”, “CSA.v”, “onthefly.v” and onthefly2.v”. The Ublock submodule is the Verilog model of the **U-**block as in Figure 4.3. Two instances of the CSA sub-module are used to model the two 2k+2 bit carry-save adders. The onthefly module performs on-the-fly conversion for storing q_j in 2’s complement format. The onthefly2 sub-module performs on-the-fly conversion for storing n_j in 2’s complement format. Finally, the online division unit was tested using the testbench file “onlineDivTB.v”.

A.2.1 Online Multiplication

A.2.1.1 Synthesizable RTL Code

Listing 1: common.v

```
//Please type in the number of bits in the input operands
`define N 64
```

Listing 2: compressor2.v

```
`include “common.v”

// Generic 4:2 compressor
module compressor(A,B,C,D,FS,FC, cin, cin2, cout);

input [`N+3:0] A, B, C, D;
output [`N+3:0] FS, FC;
output [1:0] cout;
input cin, cin2;

reg [`N+3:0] YS, sum;
integer i,j;
reg [`N+4:0] carry, YC;

assign FS = YS;
assign FC = YC[`N+3:0];
assign cout=carry[`N+1]+YC[`N+1];


//adds S+C+D = C,S
always@(sum or D or carry or cin2)
begin
YC[0]=cin2;
for(i=0;i<`N+4;i=i+1)
    begin
       {YC[i+1],YS[i]}=D[i]+carry[i]+sum[i];
    end
end

// add A+B+C = C,S
always@(A or B or C or cin)
begin
carry[0]=cin;
for(j=0;j<`N+4;j=j+1)
    begin
       {carry[j+1],sum[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: onlinemult.v

```
`include “common.v”

module CsonlineMultiplier(clk, rst, x, y, z, z_n);

/***************************************************
inputs and outputs of Online Multiplier using Carry-Save operations
* x, y  input bit operands of multiplier
* rst   signal to reset flip-flops and indicate start of operation
* z,z_n product output in BSD representation
*/
input clk, rst, x, y;
output z, z_n;
wire x,y;
reg z, z_n;

//declaring internal variables
reg  [`N-1:0] X, Y;
wire [`N-1:0] xY, yX;
wire [`N-1:0] 94orrection, 94orrection, 94orre;
wire [`N+3:0] PS, PC;
reg  [`N+3:0] PS1, PC1;
reg  [1:0] W1, W1_n;
wire  W;
wire [3:0] V;
wire [1:0] carry;
reg  [`N-1:0] shifter;

//constants
wire [`N-2:0] zeroes;
assign zeroes =0;

//assigning all registers and flip-flops
always@(posedge clk or posedge rst)
begin
if (rst)   // reseting all flipl-flops and registers
    begin
    PS1 <=0;
       PC1 <=0;
       Y<=0;
       X<=0;
       shifter<={1’b1,zeroes};
       end
else
    begin
    PS1 <= {W, V[1:0], PS[`N-1:0],1’b0};
       PC1 <= {3’b000, PC[`N-1:0],1’b0};
       Y<=94orre;
       X<=X|94orrection;
       shifter <= {1’b0,shifter[`N-1:1]};
       end
end

//calculating y*X[i-1] and x*Y[i]
assign 94orrection = x? shifter:0;
assign 94orrection = y? shifter:0;
assign 94orre = Y|94orrection;
assign xY = x? 94orre: 0;
assign yX = y? X: 0;
```

```
//4:2 compressor to add xY+yX+PS1+PC1
compressor
compressor1(.A({4’b0000,xY}),.B(PS1),.C({4’b0000,yX}),.D(PC1),.FS(PS),.FC(PC),
.cin(1’b0), .cin2(1’b0), .cout(carry));

//Calculating the estimate V
assign V = PS[`N+3:`N]+PC[`N+3:`N];

// performing the Ercegovac M-Block function to perform (PS, PC) = V – z
assign W = (z|z_n)^V[2];

//Selection Function to determine the online product bit
always@(V[3:1])
begin
 case(V[3:1])
 3’b000: begin z=0; z_n=0; end
 3’b001: begin z=1; z_n=0; end
 3’b010: begin z=1; z_n=0; end
 3’b011: begin z=1; z_n=0; end
 3’b100: begin z=0; z_n=1; end
 3’b101: begin z=0; z_n=1; end
 3’b110: begin z=0; z_n=1; end
 3’b111: begin z=0; z_n=0; end
 default: begin z=0; z_n=0; end
 endcase
end

endmodule
```

A.2.1.2 TestBench

Listing 4: multTB.v

```
`include “common.v”
// Online Multiplication Test-Bench
module multb;

//declaring variables
reg clk, a, b, rst;
wire z, z_n;
integer i,j, error;
reg [2*`N:0] pos, neg;

//clock speed assignment
always #10 clk<=~clk;

//initializing variables
initial begin
clk=0; a=0;
rst=0; b=0;
i=0; error=0; j=0;
end

//receiving ans storing the online output bits from the online multiplier
always@(posedge clk or posedge rst) begin
if (rst) begin
 pos <= 0; neg <= 0; end
else
  begin
  pos<= {pos[2*`N-1:0],z};
  neg<= {neg[2*`N-1:0],z_n};
  end
end

/**/
//Code for exhaustive testing the online multiplier covering all input
combinations
//comment this code if exhaustive testing is not needed
//code will need to be modified according the size of the operands
 initial begin
for(i=0;i<=255;i=i+1) begin
  for(j=i;j<=255;j=j+1) begin
#10 rst<=1;
  @(posedge clk) a<=i[7]; b<=j[7]; #10; rst <=0;
  @(posedge clk) a<=i[6]; b<=j[6];
  @(posedge clk) a<=i[5]; b<=j[5];
  @(posedge clk) a<=i[4]; b<=j[4];
  @(posedge clk) a<=i[3]; b<=j[3];
  @(posedge clk) a<=i[2]; b<=j[2];
  @(posedge clk) a<=i[1]; b<=j[1];
  @(posedge clk) a<=i[0]; b<=j[0];
  @(posedge clk) a<=0; b<=0;
  #210   if((i*j)!=(pos-neg)) begin
        $display(“%d multiply by %d = %b”,i[`N-1:0],j[`N-1:0],(pos-neg));
        error=error+1;
        end
  #10 rst=1;
  #20 rst=0;
  #50;  end
end
  if(!error) $display(“No errors. All 2^%d * 2^%d  combinations are 
successful!”,`N,`N);
  else $display(“Number of Errors = %d”, error);
  #400 $finish;
end
/**/

/**
//Testing the online multiplier using selective input values
//De-comment this code if selective testing is required
initial begin
  #10 rst<=1;
  @(posedge clk) a<=1; b<=1; #10; rst <=0;
  $display(“inc=%b, PS=%b, PC=%b”,CSOM.V, CSOM.PS, CSOM.PC);
  $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                                 $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                           $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                                 $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                           $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                                 $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                                  $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=1; b<=1; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
                                 $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  @(posedge clk) a<=0; b<=0; #10; $display(“inc=%b, PS=%b, PC=%b”,CSOM.carry, 
CSOM.PS, CSOM.PC);
      $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  #20; $display(“inc=%b, PS=%b, PC=%b”,CSOM.V, CSOM.PS, CSOM.PC);
     $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  #20;  $display(“inc=%b, PS=%b, PC=%b”,CSOM.V, CSOM.PS, CSOM.PC);
     $display(“W=%b, W_n=%b”,CSOM.z, CSOM.z_n);
  #160    $display(“pos=%b, neg=%b, F=%b”,pos, neg,(pos-neg));
  #400 $finish;
end
**/

//instantiating the online multiplier module
CsonlineMultiplier CSOM(.clk(clk),.rst(rst),.x(a), .y(b), .z(z), .z_n(z_n));
endmodule
```

A.2.2 Online Division

A.2.2.1 Synthesizable RTL Code

Listing 1: common.v

```
//number of bits of the input operands and the modulus
`define N 64
//Number of bits for the clock counter
`define m 6
```

Listing 2: csa.v

```
`include “common.v”

//N+2 bit Carry-Save Adder
module CSA(A,B,C,FS,FC,cin);

/***************************************************
inputs and outputs of CSA
* A, B, C inputs operands of compressor
* cin carry-in bit input
* FS,FC output in carry-save representation
*/
input [2*`N+1:0] A, B, C;
output [2*`N+1:0] FS, FC;
input cin;

//internal declaration
reg [2*`N+1:0] YS;
integer j;
reg [2*`N+2:0] YC;

//internal assignment
assign FS = YS;
assign FC = YC[2*`N+1:0];

//c,s = A+B+C
always@(A or B or C or cin)
begin
YC[0]=cin;
for(j=0;j<(2*`N+2);j=j+1)
    begin
       {YC[j+1],YS[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: onthefly.v

```
`include “common.v”

//On-the-fly conversion module for N bit variables
module ontheflyConversionN(clk, reset, q_p, q_n,  Q);

/***************************************************
inputs and outputs of the On-the-fly conversion
* q,q_n online input in signed-bit representation
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* Q parallel output in 2’s complement format
*/
input  q_p,q_n, clk, reset;
output [`N-1:0] Q;

// Internal Declarations
wire [`N-1:0] 100orrection, normalizedQM;
reg [`N-1:0] Q1, QM1;
wire [`N-1:0] A, B, fullQ, fullQM;
wire a,b;
reg [`N-1:0] shifter;

//constants
wire [`N-2:0] zeroes;
assign zeroes =0;

//assigning variables for combinational logic
assign a = q_p|q_n;
assign b = ~a;
assign Q = Q1;
assign A = (q_n)? QM1 : Q1;
assign B = (q_p)? Q1 : QM1;

//assigning the registers
always@(posedge clk or posedge reset)
begin
if (reset) begin
       Q1<=0;
       QM1<=0;
       shifter<={1’b1,zeroes};
       end
else
     begin
       Q1<=fullQ;
       QM1<=fullQM;
       shifter <= {1’b0,shifter[`N-1:1]};
       end
end

//representing variables in fractional format
assign 100orrection = a*shifter;
assign normalizedQM = b*shifter;
assign fullQ = A|100orrection;
assign fullQM = B|normalizedQM;

endmodule
```

Listing 4: onthefly2.v

```
`include “common.v”

//On-the-fly conversion module for 2N bit variables
module ontheflyConversion2N(clk,reset, q_p, q_n,Q );

/****************************************************
inputs and outputs of the On-the-fly conversion
* q,q_n online input in signed-bit representation
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* Q parallel output in 2’s complement format
*/
input  q_p,q_n, clk, reset;
output [2*`N-1:0] Q;

// Internal Declarations
wire [2*`N-1:0] 101orrection, normalizedQM;
reg [2*`N-1:0] Q1, QM1;
wire [2*`N-1:0] A, B, fullQ, fullQM;
wire a,b;
reg [2*`N-1:0] shifter;

//constants
wire [2*`N-2:0] zeroes;
assign zeroes =0;

//assigning variables
assign a = q_p|q_n;
assign b = ~a;
assign Q = fullQ;
assign A = (q_n)? QM1 : Q1;
assign B = (q_p)? Q1 : QM1;

//assigning the registers
always@(posedge clk or posedge reset) begin
if (reset) begin  //resetting the registers
       Q1<=0;
       QM1<=0;
       shifter<={1’b1,zeroes};
       end
else
    begin
       Q1<=fullQ;
       QM1<=fullQM;
       shifter <= {1’b0,shifter[2*`N-1:1]};
       end
end

//representing variables in fractional format
assign 101orrection = a*shifter;
assign normalizedQM = b*shifter;
assign fullQ = A|101orrection;
assign fullQM = B|normalizedQM;

endmodule
```

Listing 5: Ublock.v

```
//U-Block function to merge n*2^(-4) with –d*Q[i-1]
module Ublock(
   x,
   x_n,
   d,
   u
);

/****************************************************
inputs and outputs of the U-Block
* x,x_n online input in signed-bit representation representing the online
dividend
* d online single bit input representing the divisor
* u 6-bit output
*/

// Internal Declarations
input  x,d;
input  x_n;
output [5:0] u;
reg [5:0] U;

assign u =U;

//truth table covering all possible outputs of u
always @ (x or x_n or d) begin
case ({x_n,x,d})
 4’b000: begin U=6’b000000; end
 4’b001: begin U=6’b111111; end
 4’b010: begin U=6’b000001; end
 4’b011: begin U=6’b000000; end
 4’b100: begin U=6’b111111; end
 4’b101: begin U=6’b111110; end
 default: begin U=6’bxxxxxx; end
endcase

end

endmodule
```

Listing 6: onlineDiv.v

```
`include “common.v”

module onlineDivision(clk, reset, n, n_n, d, q, q_n);

/*****************************************************
inputs and outputs of the online divisor
* n,n_n dividend input in signed-bit representation
* d divisor
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* q,q_n online output bits of Q
*/
input clk, reset;
input n,n_n, d;
output q,q_n;

//internal declarations
wire [`N-1:0] Q, 103orrection, Dinv, D;
wire [2*`N-1:0] N;
wire [4:0] V;
wire [5:0] u, U;
wire [2*`N+1:0]  WS, WC, VS, VC;
reg [`m:0] counter;
reg [2*`N+1:0] WS1, WC1, qD;
reg [`N-1:0] D1;
reg q, q_n;
reg negflag1, negflag2;
reg [2*`N-5:0] Qinv;
reg [`N-1:0] shifter;
wire [`N-2:0] zeroes;
wire [`N-2:0] ones;
wire n2, n_n, clk, reset, d, d2;


//creating constants and inverting the divisor
assign zeroes =0;
assign ones = ~zeroes;
assign Dinv = ~D;


//all registers and flip-flops used are assigned here
always@(posedge clk or posedge reset)
begin
if (reset) begin
       WS1<=0;
       WC1<=0;
       counter<=1;
       D1<=0;
       shifter<={1’b1,zeroes};
       end
else
     begin
        //shift the partial residue to the left for the next clock cycle
       WS1<=(counter==4)? {1’b0,N,1’b0}:{WS[2*`N:0],1’b0};
       WC1<=(counter==4)? 0:{WC[2*`N:0],1’b0};
       D1<=D;
        //counter to keep track the number of iterations
       counter<=counter+1;
       shifter <= {1’b0,shifter[`N-1:1]};
       end
end

//storing the inputs in fractional format
assign 104orrection = d? shifter:0;
assign D = D1|104orrection;

//Calculating Qinv  = -d*Q[i-1]*2^(-4)
always@(Q or d2 or counter or ones)
begin
if(counter>4)
  begin
          Qinv = (d2)? {~Q,ones[`N-5:0]}:0 ;
          negflag1 = d2;
  end
else
  begin
  negflag1 = 0;
  Qinv = 0;
  end
end

//Calculating qD =  -q*D[i]
always@(D or Dinv or q or q_n or counter or zeroes or ones)
begin
if(counter>4)
  begin
  if(D>0) begin
    case ({q,q_n})
      4’b00: begin qD=0; negflag2=0; end
      4’b01: begin qD={2’b00,D,zeroes,1’b0}; negflag2=0; end
      4’b10: begin qD={2’b11,Dinv,ones,1’b1}; negflag2=1; end
      default: begin qD=0; negflag2=0; end
      endcase
    end
  else begin
       qD = 0 ;
       negflag2 = 0;
       end
end
else
  begin
  negflag2 = 0;
  qD = 0;
  end
end

//CSA adders to add Qinv + WS1 + WC1 + qD
CSA adder1(.A({U,Qinv}),.B(WS1),.C(WC1),.FS(VS),.FC(VC), .cin(negflag1));
CSA adder2(.A(VS),.B(VC),.C(qD),.FS(WS),.FC(WC), .cin(negflag2));

//calculating the estimate V
assign V = VS[2*`N+1:2*`N-3]+VC[2*`N+1:2*`N-3];

//Selection function to determine the online quotient bit output
always@(V[4:1])
begin
      case (V[4:1])
      4’b0000: begin q=0; q_n=0; end
      4’b0001: begin q=1; q_n=0; end
      4’b0010: begin q=1; q_n=0; end
      4’b0011: begin q=1; q_n=0; end
      4’b0100: begin q=1; q_n=0; end
      4’b0101: begin q=1; q_n=0; end
      4’b0110: begin q=1; q_n=0; end
      4’b0111: begin q=1; q_n=0; end
      4’b1000: begin q=0; q_n=1; end
      4’b1001: begin q=0; q_n=1; end
      4’b1010: begin q=0; q_n=1; end
      4’b1011: begin q=0; q_n=1; end
      4’b1100: begin q=0; q_n=1; end
      4’b1101: begin q=0; q_n=1; end
      4’b1110: begin q=0; q_n=1; end
      4’b1111: begin q=0; q_n=0; end
      default: begin q=0; q_n=0; end
      endcase
end

assign d2 = (Q>0)? D:0;
assign U = (counter>4)? U:0;
assign n2 = n;

//converting online variables to 2’s complement format
ontheflyConversionN onthefly1(.clk(clk),.reset(reset),.q_p(q), .q_n(q_n),
.Q(Q));
ontheflyConversion2N onthefly2(.clk(clk),.reset(reset),.q_p(n2), .q_n(n_n),
.Q(N));

//The U-block function determined here
Ublock Ublock1(.x(n), .x_n(n_n),.d(d2),.u(u));

endmodule
```

A.2.2.2 Testbench

Listing 8: onlineDivTB.v

```
`include “common.v”

//Test Bench to simulate ercegovac online division
module onlineDividerTB;

reg clk,rst;
reg x,y;
reg  n, n_n, d;
wire q, q_n;
reg [`N:0] pos, neg;
integer i,j, error;

//recieve and store the values of q from the online koc multiplier-divider
//storing the positive and negative values in seperate registers
always@(posedge clk or posedge rst)
begin
if (rst) begin
 pos <= 0; neg <= 0; end
else
  begin
  pos<= {pos[`N-1:0],q};
  neg<= {neg[`N-1:0],q_n};
  end
end

//simulate clock signal with period of 20 time steps
always #10 clk<=~clk;

//initialize all signals
initial begin
clk=0;
rst=0;
n=0;
n_n=0;
d=0;
end


//Testing using selective numbers:
initial begin
#10 rst=1;
@(posedge clk) n=0; n_n=0; d=1; #10 rst =0;
@(posedge clk) n=0; n_n=0; d=1; #10
@(posedge clk) n=0; n_n=0; d=1; #10
@(posedge clk) n=0; n_n=0; d=1; #10
@(posedge clk) n=0; n_n=0; d=1; #10
$display(“WS:%b, WC:%b, VS:%b, VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
$display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
   $display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
@(posedge clk) n=0; n_n=0; d=1; #10
display(“WS:%b, WC:%b, VS:%b, VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
$display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
$display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
```

```
@(posedge clk) n=0; n_n=0; d=0; #10 $display(“WS:%b, WC:%b, VS:%b,
VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
$display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
$display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
@(posedge clk) n=1; n_n=0; d=0; #10 $display(“WS:%b, WC:%b, VS:%b,
VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
   $display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
@(posedge clk) n=0; n_n=0; d=0; #10 $display(“WS:%b, WC:%b, VS:%b,
VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
$display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
$display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
@(posedge clk) n=0; n_n=0; d=0; #10 $display(“WS:%b, WC:%b, VS:%b,
VC:%b”,div1.WS, div1.WC, div1.VS, div1.VC);
$display(“qD:%b, Qinv:%b, U:%b, V:%b”,div1.qD, div1.Qinv, div1.U, div1.V);
$display(“D:%b, Q:%b, q:%b, q_n:%b”,div1.D, div1.Q, div1.q, div1.q_n);
@(posedge clk) n=0; n_n=0; d=0; #10; $display(“Q:%b”,(pos-neg));
@(posedge clk) n=0; n_n=0; d=0;
@(posedge clk) n=0; n_n=0; d=0;

#200 $finish;
end

onlineDivision div1(.clk(clk), .reset(rst), .n(n), .n_n(n_n), .d(d), .q(q),
.q_n(q_n));

endmodule
```

A.3 Fully Online Multiplication-Division using Composite Algorithms

To efficiently model algorithm 3.5, separate Verilog sub-modules were created for the k+6 bit CSA, the k+6 bit [4:2] compressor and the k-bit on-the-fly converter for Q. Each of the Verilog files calls a common file called ‘common.v’. This contains a changeable parameter to customize the size of the input bits denoted as N. N is currently set to 64 bits.The main module is called ‘hybridOnlineMultDiv’. The test bench file called onlineMultDivTB.v can be used to either for exhaustive simulation by covering all possible input values for A and B, or it can be used to test the algorithm using single pre-defined inputs for A and B.

A.3.1 Synthesizable RTL Code

Listing 1: common.v

```
//number of bits of the input operands and the modulus
`define N 64
//Number of bits for the clock counter
`define m 6
```

Listing 2: compressor2.v

```
`include “common.v”

// Generic N+6 bit 4:2 compressor
module compressor(A,B,C,D,FS,FC, cin, cin2, cout);

/***************************************************
inputs and outputs of 4:2 Compressor
* A, B, C, D inputs operands of compressor
* cin, cin2 are carry-in inputs
* FS,FC output in carry-save representation
* cout carry-out bit
*/
input [`N+5:0] A, B, C, D;
output [`N+5:0] FS, FC;
output [1:0] cout;
input cin, cin2;

//declaration of intermediate values
reg [`N+5:0] YS, sum;
integer i,j;
reg [`N+6:0] carry, YC;

//internal assignments
assign FS = YS;
assign FC = YC[`N+5:0];
assign cout=carry[`N+1]+YC[`N+1];


//adds D+c+s = C,S using CSA
always@(sum or D or carry or cin2)
begin
YC[0]=cin2;
for(i=0;i<`N+6;i=i+1)
    begin
       {YC[i+1],YS[i]}=D[i]+carry[i]+sum[i];
    end
end

// add A+B+C = c,s  using CSA
always@(A or B or C or cin)
begin
carry[0]=cin;
for(j=0;j<`N+6;j=j+1)
    begin
       {carry[j+1],sum[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: csa.v

```
`include “common.v”

//N+6 bit Carry-Save Adder
module CSA(A,B,C,FS,FC,cin);

/****************************************************
inputs and outputs of CSA
* A, B, C inputs operands of compressor
* cin carry-in bit input
* FS,FC output in carry-save representation
*/
input [`N+5:0] A, B, C;
output [`N+5:0] FS, FC;
input cin;

//declaring variables
reg [`N+5:0] YS; //, sum;
integer j;
reg [`N+6:0] YC;

//internal assignment
assign FS = YS;
assign FC = YC[`N+5:0];

//c,s = A+B+C
always@(A or B or C or cin)
begin
YC[0]=cin;
for(j=0;j<`N+6;j=j+1)
    begin
       {YC[j+1],YS[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 4: onthefly.v

```
`include “common.v”
//On-the-fly conversion module for N+6 bit variables
module ontheflyConversionN(
   clk,
   reset,
   q_p,
   q_n,
   Q
);
/*****************************************************
inputs and outputs of the On-the-fly conversion
* q,q_n online input in signed-bit representation
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* Q parallel output in 2’s complement format
*/
input  q_p,q_n, clk, reset;
output [`N+5:0] Q;
// Internal Declarations
wire [`N+5:0] 111orrection, normalizedQM;
reg [`N+5:0] Q1, QM1;
wire [`N+5:0] A, B, fullQ, fullQM;
wire a,b;
reg [`N+5:0] shifter;

//constants
wire [`N+2:0] zeroes;
assign zeroes =0;
//assigning variables for combinational logic
assign a = q_p|q_n;
assign b = ~a;
assign Q = Q1;
assign A = (q_n)? QM1 : Q1;
assign B = (q_p)? Q1 : QM1;

//Sequential Logic assignment
always@(posedge clk or posedge reset)
begin
if (reset) begin  //reset registry
       Q1<=0;
       QM1<=0;
       shifter<={3’b001,zeroes};
       end
else
     begin
       Q1<=fullQ;
       QM1<=fullQM;
       shifter <= {1’b0,shifter[`N+4:1]};
       end
end
//representing variables in fractional format
assign 111orrection = a*shifter;
assign normalizedQM = b*shifter;
assign fullQ = A|111orrection;
assign fullQM = B|normalizedQM;
endmodule
```

Listing 5: hybridOnlineMultDiv.v

```
`include “common.v”

//Online Multiplier-Divider based on Composite Algorithms
module hybridOnlineMultDiv(clk, reset, x, y, d, z, z_n, RS, RC);

/*****************************************************
inputs and outputs of Online Multiplier using Carry-Save operations
* x, y    online input operands
* d       online divisor input
* rst     signal to reset flip-flops and indicate start of operation
* z,z_n   product output in BSD representation
* RS, RC  Remainder output in form carry-sum pair representation
*/
input clk, reset, x, y, d;
output z, z_n;
output [`N+5:0] RS, RC;
reg q, q_n;
wire clk, reset, d;

/*************************
variable declarations
************************/
//d2  = d only if Q>0 – intermediate variable used later
wire d2;
//Vectors X[i] and Y[i] for accumulating x,y
reg  [`N-1:0] X, Y;
//Vector D[i]
reg  [`N-1:0] D1;
wire [`N-1:0] D;
//x*Y[i] and y*X[i-1]
wire [`N-1:0] xY, yX, correction;
//converting inputs x,y,d to fractional inputs
wire [`N-1:0] 112orrection, Dinv;
wire [`N-1:0] 112orrection, 112orrection, 112orre;
reg  [`N-1:0] shifter;
//residue function and vector Q[i], -Q[i]
wire [`N+5:0] PS, PC, Q;
reg  [`N+5:0] WS1, WC1, qD, Qinv;
wire [`N+5:0] WS, WC, VS, VC;
//5-bit estimate of the residue function
wire [4:0]    V;
//dummy variable
wire [1:0]    dummy;
//iteration counter
reg  [`m:0]   counter;
//constants
wire [`N-2:0] zeroes;
wire [`N-2:0] ones;
//flag indicates if an input to an adder is negative
//this will set the least significant carry bit of the sum to 1
//and hence 2’s complement addition can be performed
reg negflag1, negflag2;

//creating constants
assign zeroes =0;
assign ones = ~zeroes;

//inverse of D[i]
```

```
assign Dinv = ~D;

//assigning all registers and flip-flops
always@(posedge clk or posedge reset)
begin
if (reset) begin  // reseting all flipl-flops and registers
    WS1 <=0;
       WC1 <=0;
       Y<=0;
       X<=0;
       D1<=0;
       counter<=1;
       shifter<={1'b1,zeroes};
       end
else
    begin
       //storing the next state of recurrence relation
    WS1 <= {WS[`N+4:0],1'b0};
       WC1 <= {WC[`N+4:0],1'b0};

       //storinig the vectors D[i], X[i], Y[i]
       Y<=113orre;
       X<=X|113orrection;
       D1<=D;

    //counter to keep track the number of iterations
       counter<=counter+1;
       shifter<= {1'b0,shifter[`N-1:1]};
       end
end

//storing the online inputs in fractional format
assign 113orrection = d? shifter:0;
assign 113orrection = x? shifter:0;
assign 113orrection = y? shifter:0;
assign D = D1|113orrection;

//calculating x*Y[i] and y*X[i-1]
assign 113orre = Y|113orrection;
assign xY = x? 113orre: 0;
assign yX = y? X: 0;

//calculating Qinv = -d*Q[i-1] given that Q>0
always@(Q or d2)
begin
    Qinv = (d2)? ~Q:0;
    negflag1 = d2;
end

//calculating qD = -q*d[i]
always@(D or Dinv or q or q_n)
begin
    case ({q,q_n})
      4'b00: begin qD=0; negflag2=0; end
      4'b01: begin qD={2'b00,D,4'b0000}; negflag2=0; end
      4'b10: begin qD={2'b11,Dinv,4'b1111}; negflag2=1; end
      default: begin qD=0; negflag2=0; end
      endcase
end
```

```
//calculating the addition of W = Qinv + WS1 + WC1 + qD + 2^-4*(x*Y[i] + y*X[i-
1])
compressor
compressor1(.A({6’b000000,xY}),.B(WS1),.C({6’b000000,yX}),.D(WC1),.FS(PS),.FC(P
C), .cin(1’b0), .cin2(1’b0), .cout(dummy));
CSA adder1(.A(Qinv),.B(PS),.C(PC),.FS(VS),.FC(VC), .cin(negflag1));
CSA adder2(.A(VS),.B(VC),.C(qD),.FS(WS),.FC(WC), .cin(negflag2));

//estimation of V
assign V = (counter<=4)? 0: (VS[`N+5:`N+1]+VC[`N+5:`N+1]);

//check if Q!=0
assign d2 = (Q>0)? D:0;

//converting the 114orrection114 quotient bits into 2’s complement format
ontheflyConversionN onthefly1(.clk(clk),.reset(reset),.q_p(q), .q_n(q_n),
.Q(Q));

//selection function to produce the quotient output q
always@(V[4:1])
begin
      case (V[4:1])
      4’b0000: begin q=0; q_n=0; end
      4’b0001: begin q=1; q_n=0; end
      4’b0010: begin q=1; q_n=0; end
      4’b0011: begin q=1; q_n=0; end
      4’b0100: begin q=1; q_n=0; end
      4’b0101: begin q=1; q_n=0; end
      4’b0110: begin q=1; q_n=0; end
      4’b0111: begin q=1; q_n=0; end
      4’b1000: begin q=0; q_n=1; end
      4’b1001: begin q=0; q_n=1; end
      4’b1010: begin q=0; q_n=1; end
      4’b1011: begin q=0; q_n=1; end
      4’b1100: begin q=0; q_n=1; end
      4’b1101: begin q=0; q_n=1; end
      4’b1110: begin q=0; q_n=1; end
      4’b1111: begin q=0; q_n=0; end
      default: begin q=0; q_n=0; end
      endcase
end

//output the remainder only during the last iteration
assign RS = (counter==`N+4)? WS:0;
assign RC = (counter==`N+4)? WC:0;

//output the quotient
assign z   = q;
assign z_n = q_n;

endmodule
```

A.3.2 TestBench

Listing 6: onlineMultDivTB.v

```
`include “common.v”

//Test Bench to simulate online multiplication-division using composite
algorithms
module onlinemultdividerTB;

//variable declaration
reg clk,rst;
reg x,y;
reg [`N-1:0] n;
wire q, q_n;
reg [`N:0] pos, neg, quotient;
reg d;
wire [`N+5:0] RS, RC;
reg [`N-1:0] D;
reg [2*`N-1:0] C;
integer i,j, error;
reg [`N+5:0] temp1,temp2;
reg [`m:0] m;

//recieve and store the values of q from the online koc multiplier-divider
//storing the positive and negative values in seperate registers
always@(posedge clk or posedge rst)
begin
if (rst) begin
 pos <= 0; neg <= 0; end
else
  begin
  pos<= {pos[`N-1:0],q};
  neg<= {neg[`N-1:0],q_n};
  end
end

//simulate clock signal with period of 20 time steps
always #10 clk<=~clk;

//initialize all signals
initial begin
clk=0;
rst=0;
m = `N-1;
i=0;
j=0;
quotient=0;
error=0;

n = 63;   // <- Adjust modulus here
C=0;
D=0;
x=0;
y=0;
d=0;
end
```

```
/****************
//This section of the code is used for brute-force testing
// De-comment this section when needed
initial begin
for(i=0;i<=n-1;i=i+1)
begin
  for(j=i;j<=n-1;j=j+1)
  begin
    C=j*i;
    D=C/n;
    #10 rst=1;
//  @(posedge clk) x=i[7]; y=j[7];    #10; rst =0;
//  @(posedge clk) x=i[6]; y=j[6]; // #10; rst =0;
  @(posedge clk) x=i[5]; y=j[5]; d=n[5]; #10; rst =0;
  @(posedge clk) x=i[4]; y=j[4]; d=n[4]; // #10; rst =0;
  @(posedge clk) x=i[3]; y=j[3]; d=n[3]; //  #10; rst =0;
  @(posedge clk) x=i[2]; y=j[2]; d=n[2];
  @(posedge clk) x=i[1]; y=j[1]; d=n[1];
  @(posedge clk) x=i[0]; y=j[0]; d=n[0];
  @(posedge clk) x=0; y=0; d=0;
  #80; temp1=RS; temp2=RC;
  #20;   quotient=pos-neg;
  if((D –quotient)) begin
       $display(“%d * %d /%d = %b,   WS=%b, WC=%b”,i[`N-1:0],j[`N-
1:0],n,quotient, temp1, temp2);
        error=error+1;
        end
   #10 rst=1;
   #20 rst=0;
   #100;
   end
end
if(!error) $display(“No errors. All 2^%d * 2^%d  combinations are
successful!”,m,m);
else $display(“Correction Stage Required! Number of Errors = %d”, error);
***********/

/**/
//Testing using selective numbers:
initial begin
#10 rst=1;
@(posedge clk) x=1; y=1; d=n[5]; #10; rst =0;
#5; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=1; y=1; d=n[4];
#10; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=1; y=0; d=n[3];
#10; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
```

```
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=1; d=n[2];
#10; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=0; d=n[1];
#10; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk) x=0; y=1; d=n[0];
#10; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

@(posedge clk)  x=0; y=0; d=0;
#20; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

#20; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

#20; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

#20; $display(“qD=%b,PS=%b,PC=%b,
remainder=%b”,multdivider.qD,multdivider.PS,multdivider.PC, (RS+RC));
$display(“VS=%b,VC=%b,Qinv=%b”,multdivider.VS,multdivider.VC,multdivider.Qinv);
$display(“WS=%b,WC=%b,q=%b,q_n=%b\n”,multdivider.WS,multdivider.WC,multdivider.
q,multdivider.q_n);

#20; $display(“A:10100 * B:10100 / D:%b = Q:%b , R=%b”,n, (pos-neg),(RS+RC));
/**/

#200 $finish;
end

hybridOnlineMultDiv multdivider(.clk(clk), .reset(rst), .x(x), .y(y), .d(d),
.z(q), .z_n(q_n), .RS(RS), .RC(RC));

endmodule
```

A.4 Online Multiplication-Division with Constant Divisor Integrated with the Correction Stage Algorithm

The top Verilog module is called “onlineKoc.v”. The top module creates instances for the sub-modules “bitwiseGP.v”, “GPOperator.v”, “signDetect.v”, “signedAdder.v”, “compressor.v” and “CSA.v”. The sub-module bitwiseGP creates generate and propagate signals for each of the remainder (PC, PS) bits. The sub-module GPOperator implements a single generate-propagate operator unit as in Figure 3.3(b). The sub-module signedAdder is a 2-bit binary signed digit (BSD) adder for accumulating the partial quotient with q_i. The compressor and CSA sub-modules implements a k+4 bit [4:2] compressor and a k+4 bit carry-save adder as required by Algorithm 3.4. The algorithm was exhaustively tested using all possible input combinations with k = 4, 6, 8 and 10 using the testbench file “onlineKocTB.v”. Due to the incorporated correction stage, the online multiplication-division algorithm produced the correct results for all input combinations without errors.

A.4.1 Synthesizable RTL Code

Listing 1: common.v – generated by main.exe

```
//number of bits of the input operands and the modulus
`define N 8
//Number of bits for the clock counter
`define m 3
```

Listing 2: csa.v

```
`include “common.v”

//N+4 bit Carry-Save adder
module CSA(A,B,C,FS,FC,cin);

/****************************************************
inputs and outputs of CSA
* A, B, C inputs operands of compressor
* cin carry-in bit input
* FS,FC output in carry-save representation
*/
input [`N+3:0] A, B, C;
output [`N+3:0] FS, FC;
input cin;

reg [`N+3:0] YS;
integer j;
reg [`N+4:0] YC;

assign FS = YS;
assign FC = YC[`N+3:0];

//c,s = A+B+C
always@(A or B or C or cin)
begin
YC[0]=cin;
for(j=0;j<`N+4;j=j+1)
    begin
       {YC[j+1],YS[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: compressor2.v

```
`include "common.v"

// Generic N+4 bit 4:2 compressor
module compressor(A,B,C,D,FS,FC, cin, cin2, cout);

/***************************************************
inputs and outputs of 4:2 Compressor
* A, B, C, D inputs operands of compressor
* cin, cin2 are carry-in inputs
* FS,FC output in carry-save representation
* cout carry-out bit
*/
input [`N+3:0] A, B, C, D;
output [`N+3:0] FS, FC;
output cout;
input cin, cin2;

//declaration of intermediate values
reg [`N+3:0] YS, sum;
integer i,j;
reg [`N+4:0] carry, YC;

//generating the output
assign FS = YS;
assign FC = YC[`N+3:0];
assign cout=carry[`N+4]^YC[`N+4];


//adds D+c+s = C,S using CSA
always@(sum or D or carry or cin2)
begin
YC[0]=cin2;
for(i=0;i<`N+4;i=i+1)
    begin
       {YC[i+1],YS[i]}=D[i]+carry[i]+sum[i];
    end
end

// add A+B+C = c,s  using CSA
always@(A or B or C or cin)
begin
carry[0]=cin;
for(j=0;j<`N+4;j=j+1)
    begin
       {carry[j+1],sum[j]}=A[j]+B[j]+C[j];
    end
end

endmodule
```

Listing 3: signedAdder.v

```
//Adds 2-bit redundant BSD numbers
module signedAdder(A, A_n, B, B_n, S, S_n);

/***************************************************
inputs and outputs of the BSD adder
* A,A_n input operand in BSD form
* B,B_n input operand in BSD form
* S,S_n output sum in BSD form
*/
input [1:0] A, A_n, B;
input [1:0] B_n;
output [1:0] S, S_n;
wire [1:0] A, A_n, B, B_n;
wire [1:0] S, S_n;

//internal variables
wire [2:0] AgB, AgB_n, BgA_n, BgA;
wire [2:0] C, C_n;
reg [1:0] t, t_n;

//add the two numbers
assign C=A+B;
assign C_n = A_n+B_n;

//selecton function that changes the Q bit if the sum is 0,-1 to -1,1
//this function 121orrect that there is no overflow ocurrs at last iteration
assign S =({t,t_n}==4'b0001)? 2'b01 :t;
assign S_n = ({t,t_n}==4'b0001)? 2'b10 :t_n;

//if A is greator than B
assign AgB = C+~C_n+1;
assign AgB_n = 0;

//if B is greator than A
assign BgA = 0;
assign BgA_n = ~C+C_n+1;

always@(C or C_n or AgB or BgA_n)
begin
//detect the sign of the addition
 if(C>=C_n) begin
   t= AgB[1:0];
   t_n=0;
 end
 else begin
  t=0;
  t_n=BgA_n[1:0];
 end
end

endmodule
```

Listing 4: bitwiseGP.v

```
`include “common.v”

//Bitwise Generate and Propagate signals for the remainder
module GP(
   PC,
   PS,
   P,
   G
);

/*****************************************************
inputs and outputs of the Bitwise G-P
* PC,PS inputs operands representing the remainder
* P     N-bit propagate signals output
* G     N-bit generate signal ouput
*/
input [`N-2:0] PC, PS;
output [`N-2:0] P,G;

// Internal Declarations
reg [`N-2:0] p,g;
integer j;

//produce the 122orrec
assign P = p;
assign G=g;

//Generating Propagate and Generate signals for each PS,Pc bit
always@(PS or PC)
begin
for(j=0;j<`N-1;j=j+1)
    begin
        p[j] = PS[j]|PC[j];
        g[j] = PS[j]&PC[j];
    end
end

endmodule
```

Listing 5: Gpoperator.v

```
//Generate-Propagate Operator
module GPO(
   G1,
   G2,
   G3
);

/*****************************************************
inputs and outputs of the Generate-Propagate Operator
* G1,G2 2-bit input operands representing consecutive G-P signals
* G3    2-bit Output representing the 123orrect G-P signals that covers G1, G2
*/

input [1:0] G1, G2;
output [1:0] G3;

//generalized carry operator
//merges two adjacent G,P pairs
assign G3[1] = G1[1]&G2[1];
assign G3[0] = G1[0]|(G1[1]&G2[0]);

endmodule
```

Listing 6: example of signDetect.v – generated by main.exe

```
`include “common.v”

module signDetector(PC,PS,Y);

// Internal Declarations
input  [`N-2:0] PS, PC;
output Y;

wire [`N-2:0] PS,PC,P,G;
wire Y,dummy;
wire [1:0] G1;
wire [1:0] G2;
wire [1:0] G3;
wire [1:0] G4;
wire [1:0] G5;
wire [1:0] G6;
GP GP1(.PS(PS),.PC(PC),.P(P),.G(G));

GPO GPO7(.G1({P[1],G[1]}),.G2({P[0],G[0]}),.G3(G1));
GPO GPO8(.G1({P[3],G[3]}),.G2({P[2],G[2]}),.G3(G2));
GPO GPO9(.G1({P[5],G[5]}),.G2({P[4],G[4]}),.G3(G3));
GPO GPO10(.G1(G2),.G2(G1),.G3(G4));
GPO GPO11(.G1({P[6],G[6]}),.G2(G3),.G3(G5));
GPO GPO12(.G1(G5),.G2(G4),.G3({dummy,Y}));
endmodule
```

Listing 7: onlinekoc.v

```
include "common.v"

//Online Multiplier-Divider with Correction stage for parallel input divisor
module onlineKoc(clk, reset, x, y, n, q, q_n, RS, RC);

/*****************************************************
inputs and outputs of the online multiplicaiton-division module
* x,y inputs operands
* n modulus
* reset is a reset or start signal to initialize fli-flops and indicate start
of operations
* RS,RC remainder output in carry-save representation
* q,q_n online output bits of Q
*/
input clk, reset, x,y;
input [`N-1:0] n;
output q,q_n;
output [`N-1:0] RS,RC;

/*************************
internal declarations
************************/
//Vectors X[i] and Y[i] for accumulating x,y
reg [`N-1:0] X, Y;
//x*Y[i] and y*X[i-1]
wire [`N-1:0] xY, yX;
//converting inputs x,y to fractional inputs
wire [`N-1:0] normalizedx, normalizedy, fullY;
reg [`N-1:0] shifter;
wire [`N-2:0] zeroes;
//sign estimate
wire [4:0] ES;
reg [4:0] ES1;
//residue function and remainder
wire [`N+3:0]  PS, PC, VS, VC, rs,rc;
reg [`N+3:0] PS1, PC1, correction, add;
//iteration counter
reg [`m:0] counter;
//intermediate variables
reg g,g_n;
wire dummy, carry;
//partial quotient
reg [1:0] Q1,Q1_n;
wire [1:0] Q,Q_n;
//quotient increments that are added to the partial quotient
wire [1:0] increment, increment_n, tmp;
//flag indicates if an input to an adder is negative
//this will set the least significant carry bit of the sum to 1
//and hence 2's complement addition can be performed
reg negflag2;
wire negflag1;
//8M and -8M
wire [`N+3:0] n_p, n_n;

//creating constants and shifting the modulus
assign zeroes =0;
assign n_p = {1'b0,n,3'b000};
assign n_n = {1'b1,~n,3'b111};
```

```
//all registers and flip-flops used are declared here
always@(posedge clk or posedge reset)
begin
//reseting all flip-flops
if (reset) begin
       PS1<=0; PC1<=0;
       counter<=1;
       ES1<=0;
       Y<=0;
       X<=0;
        Q1<=0; Q1_n<=0;
       shifter<={1'b1,zeroes};
       end
else
    begin
    //shift the partial residue to the left for the next clock cycle
       PS1<={PS[`N+2:0],1'b0};
       PC1<={PC[`N+2:0],1'b0};

    //shift the partial quotient to the left
       Q1<={Q[0],1'b0};
       Q1_n<={Q_n[0],1'b0};

    //counter to keep track the number of iterations
       counter<=counter+1;

    //saving the estimate for the next clock cycle
       ES1<=ES;

    //storing all previous values of the input opernds X,Y in a shift register
       Y<=fullY;
       X<=X|normalizedx;
       shifter <= {1'b0,shifter[`N-1:1]};
       end
end

//storing the inputs in fractional format since all internal operations are
// done in fractions
assign normalizedx = x? shifter:0;
assign normalizedy = y? shifter:0;

//multiplying x_i*Y[i]  and y_i*X[i-1]
assign fullY = Y|normalizedy;
assign xY = x? fullY: 0;
assign yX = y? X: 0;

//The sign estimate procedure
always@(ES1 or n_n or n_p)
begin
  if(ES1>=1 && ES1 <=15)  begin
       add = n_n;
       g = 1; g_n=0;
   end
  else if(ES1>=16 && ES1<=27)   begin
       add = n_p;
       g = 0; g_n=1;
   end
  else
   begin
```

```
      add = 0;
      g = 0; g_n=0;
   end
end

assign negflag1 = g;

//sum x*Y[i] + y*X[i] +/-M + PS + PC using a compressor and CSA
CSA adder1(.A({4'b0000,xY}),.B({4'b0000,yX}),.C(add),.FS(VS),.FC(VC),
.cin(negflag1));
compressor compressor1(.A(VC),.B(PS1),.C(VS),.D(PC1),.FS(PS),.FC(PC),
.cin(1'b0), .cin2(1'b0), .cout(dummy));

//dtermine if a carry is generated from the k-1 bits of PS and PC
signDetector SD1(.PS(PS[`N-2:0]), .PC(PC[`N-2:0]), .Y(carry));

//find the estimate of PS,PC
assign ES = PS[`N+3:`N-1] + PC[`N+3:`N-1];

//dtermine if the sum PS+PC is a positive or a negative number
always@(ES or carry) begin
if (ES[4]==0)
     negflag2=0;
else if(ES == 31)
     negflag2=!carry;
else negflag2=1;
end

//correction stage
always@(counter or negflag2) begin
if(counter == `N+3)  begin
  correction = (negflag2)? n_p:0;
 end
else  correction =0; end

//correct the remainder by adding correction value
CSA adder2(.A(PS),.B(PC),.C(correction),.FS(rs),.FC(rc), .cin(1'b0));

//output the remainder after shifting back to the correct positions
assign RS = rc[`N+2:3];
assign RC = rs[`N+2:3];

//finding the partial quotient Q based on the additions and subtractions of n
assign increment = (counter==`N+4)? 0:{1'b0,g};
assign tmp = (counter==`N+3)? (g_n + negflag2): {1'b0,g_n};
assign increment_n = (counter==`N+4)? 0:tmp;

//the selection function is inherent in the signed adder
signedAdder signed2(.A(increment), .A_n(increment_n), .B(Q1), .B_n(Q1_n),
.S(Q), .S_n(Q_n));

//output the quotient bits
assign q = Q[1];
assign q_n = Q_n[1];
endmodule
```

A.4.2 Testbench

Listing 8: onlinekocTB.v

```
`include “common.v”

//Test Bench for simulating the online multiplier-divider with correction stage
module onlinemultdividerTB;

reg clk,rst;
reg x,y;
reg [`N-1:0] n;
wire q, q_n;
// pos,neg are used to accumulate the online quotient bits from the 128orrec in
BSD format
reg [`N+2:0] pos, neg;
reg [`N-1:0] D;
wire [`N-1:0] RS, RC;
reg [2*`N-1:0] C;
integer i,j, error;
reg [`N-1:0] temp1,temp2;
reg [5:0] m;

//recieve and store the values of q from the online koc multiplier-divider
//storing the positive and negative values in seperate registers
always@(posedge clk or posedge rst)
begin
if (rst) begin
 pos <= 0; neg <= 0; end
else
  begin
  pos<= {pos[`N+1:0],q};
  neg<= {neg[`N+1:0],q_n};
  end
end

//simulate clock signal with period of 20 time steps
always #10 clk<=~clk;

//initialize all signals
initial begin
clk=0;
rst=0;
m = `N-1;
i=0;
j=0;
error=0;

n=131; // <- set modulus N here

C=0;
D=0;
x=0;
y=0;
end

/****************/
//This section of the code is used for brute-force testing
```

```
// De-comment this section when needed
initial begin
for(i=0;i<=n-1;i=i+1)
begin
  for(j=i;j<=n-1;j=j+1)
  begin
    C=j*i;
    D=C/n;
    #10 rst=1;
  @(posedge clk) x=i[7]; y=j[7];    #10; rst =0;
  @(posedge clk) x=i[6]; y=j[6]; // #10; rst =0;
  @(posedge clk) x=i[5]; y=j[5]; //  #10; rst =0;
  @(posedge clk) x=i[4]; y=j[4]; // #10; rst =0;
  @(posedge clk) x=i[3]; y=j[3];  // #10; rst =0;
  @(posedge clk) x=i[2]; y=j[2];
  @(posedge clk) x=i[1]; y=j[1];
  @(posedge clk) x=i[0]; y=j[0];
  @(posedge clk) x=0; y=0;
  #80; temp1=RS; temp2=RC;
  #20;   if((D –(pos-neg))) begin
       $display(“%d * %d /%d = %b,   RS=%b, RC=%b”,i[`N-1:0],j[`N-1:0],n,(pos-
neg), temp1, temp2); //RS,RC);
        error=error+1;
        end
   #10 rst=1;
   #20 rst=0;
   #100;
   end
end
if(!error) $display(“No errors. All 2^%d * 2^%d  combinations are
successful!”,m,m);
else $display(“Number of Errors = %d”, error);
/***********/

/**
//Testing using selective numbers:
initial begin
#10 rst=1;
@(posedge clk) x=1; y=1; #10; rst =0;

@(posedge clk) x=1; y=1;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,129orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) x=1; y=1;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,129orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);
@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,129orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
```

```
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,130orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,130orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk)  x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,130orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,130orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) x=0; y=0;
#10; $display(“add=%b,VS=%b,VC=%b,
remainder=%b”,multdivider.add,multdivider.VS,multdivider.VC, (RS+RC));
$display(“ES=%b,tmp=%b,130orrection=%b”,multdivider.ES,multdivider.carry,multdi
vider.correction);
$display(“PS=%b,PC=%b,q=%b,q_n=%b\n”,multdivider.PS,multdivider.PC,multdivider.
Q,multdivider.Q_n);

@(posedge clk) // x=0; y=0;
#80; $display(“A:101110 * B:011010 / N:%b = Q:%b”,n,(pos-neg));
**/
#200 $finish;
end

onlineKoc multdivider(.clk(clk), .reset(rst), .x(x), .y(y), .n(n), .q(q),
.q_n(q_n), .RS(RS), .RC(RC));
endmodule
```

Appendix B

C++ Code to Generate signDetect.v File for Implementing the Brent-Kung Adder

Listing 1: main.cpp

```
#include<stdlib.h>
#include<stdio.h>
#include<fstream>
#include<iostream>
#include<string>
#include<math.h>
#include<vector>
using namespace std;

void welcome(void)
{
    cout<<"\n
ÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜ\n";
    cout<<"  Ý
Þ\n";
    cout<<"  Ý   Copyright KOCH 2008, King Fahd University of Petroleum and
Minerals   Þ\n";
    cout<<"  Ý Logarithmic Sign Detection File Creation for Online Multiplier-
Divider Þ\n";
    cout<<"  Ý
Þ\n";
    cout<<"
ßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßß\n";
    cout<<endl;
}

void pause (void)
{
    system("PAUSE");
}

main()
{
    system("IF EXIST signdetect.v rename signdetect.v signdetect.bak");
    ofstream sd("signdetect.v");
    ofstream common("common.v");
    if(sd.fail()) cout<<"ERROR!\n";
    int N, depth, i,j, n, levelStart=0,temp, m;
    vector<int> *levelNums;
    int *nodes, *levelSize;
    float numGPO;
    welcome();

    cout<<"Please enter the maximum number of bits used by the modulus:\n";
    cout<<">> ";
    cin>>N;
    numGPO= (float)N-1.0;
    depth = ceil(log10(numGPO)/log10(2.0));
    n=N-1;

    sd<<"`include \"common.v\"\n\n";
    sd<<"module signDetector(PC,PS,Y);\n\n";
    sd<<"// Internal Declarations\n";
    sd<<"input  [`N-2:0] PS, PC;\n";
    sd<<"output Y;\n\n";
    sd<<"wire [`N-2:0] PS,PC,P,G;\n";
    sd<<"wire Y,dummy;\n";

    for(i=1;i<n;i++)
      sd<<"wire [1:0] G"<<i<<";\n";

    //getting bitwise g and p
    sd<<"GP GP1(.PS(PS),.PC(PC),.P(P),.G(G));\n\n";

    //Greating GPO tree
    levelNums = new vector<int>[depth+1];
    levelSize = new int[depth+1];
    nodes = new int[depth+1];

    levelStart=n;
    levelSize[0]=n;
    for(i=0;i<n;i++)
      levelNums[0].push_back(i);
    for(j=1;j<=depth;j++)
    {
        levelSize[j] = levelSize[j-1]/2;
        nodes[j]= levelSize[j];
        for(i=0;i<levelSize[j];i++) {
          levelNums[j].push_back(i+levelStart); }
        if(levelSize[j-1]%2)
        {
            temp= levelNums[j-1].back();
            levelNums[j].push_back(temp);
            levelStart=levelStart+levelSize[j];
            levelSize[j]++;
        }
        else {
          levelStart=levelStart+levelSize[j]; }
     }

     for(j=1;j<=depth;j++)
     {
         for(i=0;i<nodes[j];i++)
         {
             sd<<"GPO GPO"<<levelNums[j][i]<<"(.G1(";
             if(levelNums[j-1][2*i+1]<n)
                 sd<<"{P["<<levelNums[j-1][2*i+1]<<"],G["<<levelNums[j-
1][2*i+1]<<"]}";
             else
                 sd<<"G"<<levelNums[j-1][2*i+1]+1-n;
             sd<<"),.G2(";
             if(levelNums[j-1][2*i]<n)
                 sd<<"{P["<<levelNums[j-1][2*i]<<"],G["<<levelNums[j-
1][2*i]<<"]}";
             else
```

```
                  sd<<"G"<<levelNums[j-1][2*i]+1-n;
              if(j==depth)
                 sd<<"),.G3({dummy,Y}));\n";
              else
                 sd<<"),.G3(G"<<levelNums[j][i]+1-n<<"));\n";
          }
      }
      sd<<"endmodule\n";

     sd.close();
     cout<<"\nFile Created Successfully!\n";

     //creating common file
     common<<"//number of bits of the input operands and the modulus\n";
     common<<"`define N "<<N<<endl;
     common<<"//Number of bits for the clock counter\n";
     m = ceil(log10((float)N)/log10(2.0));
     common<<"`define m "<<m<<endl;
     common.close();
     pause();
     return 0;
}
```

Appendix C

Synthesis Reports for 64-bit Online Multiplication-Division Operations

C.1 Online Multiplication-Division with Constant Divisors

```
HDL Synthesis Report

Macro Statistics
# ROMs                                                  : 1
 4x1-bit ROM                                            : 1
# Adders/Subtractors                                    : 408
 1-bit adder carry out                                  : 204
 2-bit adder                                            : 203
 5-bit adder                                            : 1
# Registers                                             : 6
 5-bit register                                         : 1
 64-bit register                                        : 3
 68-bit register                                        : 2
# Comparators                                           : 4
 5-bit comparator greatequal                            : 2
 5-bit comparator lessequal                             : 2
# Multiplexers                                          : 1
 68-bit 4-to-1 multiplexer                              : 1
# Xors                                                  : 1
 1-bit xor2                                             : 1

*                       Advanced HDL Synthesis                          *

WARNING:Xst:1710 - FF/Latch  <PS1_0> (without init value) has a constant value of 0 in
block <onlineKoc>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <PC1_0> (without init
value) has a constant value of 0 in block <onlineKoc>.

Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                                  : 1
 4x1-bit ROM                                            : 1
# Adders/Subtractors                                    : 408
 1-bit adder carry out                                  : 204
 2-bit adder                                            : 203
 5-bit adder                                            : 1
# Registers                                             : 331
 Flip-Flops                                             : 331
# Comparators                                           : 4
 5-bit comparator greatequal                            : 2
 5-bit comparator lessequal                             : 2
# Multiplexers                                          : 1
 68-bit 4-to-1 multiplexer                              : 1
# Xors                                                  : 1
 1-bit xor2                                             : 1

*                         Low Level Synthesis                           *
Loading device for application Rf_Device from file '4vsx35.nph' in environment
C:\Xilinx.
```

```
Optimizing unit <onlineKoc> ...

Optimizing unit <compressor> ...

Optimizing unit <CSA> ...

Mapping all equations...
WARNING:Xst:1710 - FF/Latch  <PC1_1> (without init value) has a constant value of 0 in
block <onlineKoc>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <PC1_2> (without init
value) has a constant value of 0 in block <onlineKoc>.
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block onlineKoc, actual ratio is 2.

*                            Final Report                                    *
=========================================================================
Final Results
RTL Top Level Output File Name     : onlineKoc.ngr
Top Level Output File Name         : onlineKoc
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 206

Cell Usage :
# BELS                             : 690
#      GND                         : 1
#      LUT2                        : 5
#      LUT2_D                      : 1
#      LUT3                        : 139
#      LUT3_D                      : 299
#      LUT3_L                      : 4
#      LUT4                        : 140
#      LUT4_D                      : 80
#      LUT4_L                      : 12
#      MUXCY                       : 4
#      MUXF5                       : 1
#      XORCY                       : 4
# FlipFlops/Latches                : 329
#      FDC                         : 328
#      FDP                         : 1
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 205
#      IBUF                        : 67
#      OBUF                        : 138
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                      379  out of  15360     2%
 Number of Slice Flip Flops:            329  out of  30720     1%
 Number of 4 input LUTs:                680  out of  30720     2%
 Number of bonded IOBs:                 206  out of    448    45%
 Number of GCLKs:                         1  out of     32     3%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 329   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -12
```

```
   Minimum period: 4.638ns (Maximum Frequency: 215.596MHz)
   Minimum input arrival time before clock: 6.316ns
   Maximum output required time after clock: 7.947ns
   Maximum combinational path delay: 8.653ns

Timing Detail:
-------------------------------------------------------------------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock ‘clk’
  Clock period: 4.638ns (frequency: 215.596MHz)
  Total number of paths / destination ports: 7994 / 328
-------------------------------------------------------------------------
Delay:               4.638ns (Levels of Logic = 9)
  Source:            ES1_4 (FF)
  Destination:       ES1_4 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: ES1_4 to ES1_4
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q             13   0.272   0.713  ES1_4 (ES1_4)
     LUT4:I0->O            1   0.147   0.451  add<62>1_SW1 (N463)
     LUT4_D:I2->O          2   0.147   0.417  add<62>1 (add<62>)
     LUT4_D:I3->LO         1   0.147   0.157  adder1/Madd__AUX_63_Mxor_Result<1>_Result1
(N858)
     LUT3:I2->O            4   0.147   0.580
compressor1/Madd__AUX_200_Mxor_Result<1>_Result1 (compressor1/carry<64>)
     LUT3_D:I0->O          1   0.147   0.394
compressor1/Madd__AUX_133_Mxor_Result<0>_Result1 (RC_64_OBUF)
     MUXCY:DI->O           1   0.280   0.000  onlineKoc_ES<1>cy (onlineKoc_ES<1>_cyo)
     MUXCY:CI->O           1   0.034   0.000  onlineKoc_ES<2>cy (onlineKoc_ES<2>_cyo)
     MUXCY:CI->O           0   0.034   0.000  onlineKoc_ES<3>cy (onlineKoc_ES<3>_cyo)
     XORCY:CI->O           1   0.273   0.000  onlineKoc_ES<4>_xor (ES<4>)
     FDC:D                     0.297          ES1_4
    ----------------------------------------
    Total                      4.638ns (1.925ns logic, 2.713ns route)
                                       (41.5% logic, 58.5% route)

=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock ‘clk’
  Total number of paths / destination ports: 2788 / 265
-------------------------------------------------------------------------
Offset:              6.316ns (Levels of Logic = 10)
  Source:            y (PAD)
  Destination:       ES1_4 (FF)
  Destination Clock: clk rising

  Data Path: y to ES1_4
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           263   0.754   1.784  y_IBUF (y_IBUF)
     LUT4_D:I0->LO         1   0.147   0.157  XY<63>1 (N495)
     LUT3:I2->O            1   0.147   0.451  add<63>1_SW0 (N457)
     LUT4:I2->O            3   0.147   0.541  adder1/Madd__AUX_64_Mxor_Result<0>_Result1
(VS<63>)
     LUT3:I1->O            4   0.147   0.580
compressor1/Madd__AUX_200_Mxor_Result<1>_Result1 (compressor1/carry<64>)
     LUT3_D:I0->O          1   0.147   0.394
compressor1/Madd__AUX_133_Mxor_Result<0>_Result1 (RC_64_OBUF)
     MUXCY:DI->O           1   0.280   0.000  onlineKoc_ES<1>cy (onlineKoc_ES<1>_cyo)
     MUXCY:CI->O           1   0.034   0.000  onlineKoc_ES<2>cy (onlineKoc_ES<2>_cyo)
     MUXCY:CI->O           0   0.034   0.000  onlineKoc_ES<3>cy (onlineKoc_ES<3>_cyo)
     XORCY:CI->O           1   0.273   0.000  onlineKoc_ES<4>_xor (ES<4>)
     FDC:D                     0.297          ES1_4
    ----------------------------------------
    Total                      6.316ns (2.407ns logic, 3.909ns route)
                                       (38.1% logic, 61.9% route)

=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock ‘clk’
  Total number of paths / destination ports: 6048 / 136
-------------------------------------------------------------------------
Offset:              7.947ns (Levels of Logic = 7)
  Source:            ES1_1 (FF)
  Destination:       RS<10> (PAD)
  Source Clock:      clk rising
  Data Path: ES1_1 to RS<10>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              4   0.272   0.580  ES1_1 (ES1_1)
     LUT2_D:I0->LO         1   0.147   0.109  Ker3_SW0 (N855)
     LUT4:I3->O           58   0.147   1.008  Ker3_3 (Ker3_2)
     LUT3_D:I2->O          1   0.147   0.451  add<8>1 (add<8>)
     LUT4:I2->O            2   0.147   0.543  adder1/Madd__AUX_9_Mxor_Result<0>_Result1
(VS<8>)
     LUT3_D:I1->O          1   0.147   0.451
compressor1/Madd__AUX_145_Mxor_Result<1>_Result1 (compressor1/carry<9>)
     LUT3_D:I2->O          1   0.147   0.394
compressor1/Madd__AUX_78_Mxor_Result<1>_Result1 (RS_10_OBUF)
     OBUF:I->O                 3.255          RS_10_OBUF (RS<10>)
    ----------------------------------------
    Total                      7.947ns (4.409ns logic, 3.538ns route)
                                       (55.5% logic, 44.5% route)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 2067 / 134
-------------------------------------------------------------------------
Delay:               8.653ns (Levels of Logic = 7)
  Source:            y (PAD)
  Destination:       RC<64> (PAD)

  Data Path: y to RC<64>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           263   0.754   1.784  y_IBUF (y_IBUF)
     LUT4_D:I0->LO         1   0.147   0.157  xY<63>1 (N495)
     LUT3:I2->O            1   0.147   0.451  add<63>1_SW0 (N457)
     LUT4:I2->O            3   0.147   0.541  adder1/Madd__AUX_64_Mxor_Result<0>_Result1
(VS<63>)
     LUT3:I1->O            4   0.147   0.580
compressor1/Madd__AUX_200_Mxor_Result<1>_Result1 (compressor1/carry<64>)
     LUT3_D:I0->O          1   0.147   0.394
compressor1/Madd__AUX_133_Mxor_Result<0>_Result1 (RC_64_OBUF)
     OBUF:I->O                 3.255          RC_64_OBUF (RC<64>)
    ----------------------------------------
    Total                      8.653ns (4.744ns logic, 3.909ns route)
                                       (54.8% logic, 45.2% route)

=========================================================================
CPU : 25.22 / 25.53 s | Elapsed : 26.00 / 26.00 s
```

C.2 Fully Online Multiplication-Division using Composite Algorithms

```
HDL Synthesis Report

Macro Statistics
# ROMs                                                 : 2
 16x2-bit ROM                                          : 1
 4x1-bit ROM                                           : 1
# Adders/Subtractors                                   : 561
 1-bit adder carry out                                 : 281
 2-bit adder                                           : 279
 5-bit adder                                           : 1
# Counters                                             : 1
 8-bit up counter                                      : 1
# Registers                                            : 9
 64-bit register                                       : 4
 70-bit register                                       : 5
# Comparators                                          : 2
 70-bit comparator greater                             : 1
 8-bit comparator lessequal                            : 1
# Multiplexers                                         : 1
 70-bit 4-to-1 multiplexer                             : 1

*                       Advanced HDL Synthesis                          *

WARNING:Xst:1710 - FF/Latch  <WS1_0> (without init value) has a constant value of 0 in
block <hybridOnlineMultDiv>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <WC1_0> (without init
value) has a constant value of 0 in block <hybridOnlineMultDiv>.

Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                                 : 2
 16x2-bit ROM                                          : 1
 4x1-bit ROM                                           : 1
# Adders/Subtractors                                   : 561
 1-bit adder carry out                                 : 281
 2-bit adder                                           : 279
 5-bit adder                                           : 1
# Counters                                             : 1
 8-bit up counter                                      : 1
# Registers                                            : 604
 Flip-Flops                                            : 604
# Comparators                                          : 2
 70-bit comparator greater                             : 1
 8-bit comparator lessequal                            : 1
# Multiplexers                                         : 1
 70-bit 4-to-1 multiplexer                             : 1

*                            Final Report                               *
Final Results
RTL Top Level Output File Name     : hybridOnlineMultDiv.ngr
Top Level Output File Name         : hybridOnlineMultDiv
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 147

Cell Usage :
# BELS                             : 1513
#      BUF                         : 1
#      GND                         : 1
#      INV                         : 1
#      LUT2                        : 105
#      LUT2_D                      : 7
#      LUT2_L                      : 71
#      LUT3                        : 262
#      LUT3_D                      : 123
#      LUT3_L                      : 341
#      LUT4                        : 303
#      LUT4_D                      : 200
#      LUT4_L                      : 56
#      MUXCY                       : 22
#      MUXF5                       : 15
#      VCC                         : 1
#      XORCY                       : 4
# FlipFlops/Latches                : 546
#      FDC                         : 544
#      FDP                         : 2
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 146
#      IBUF                        : 4
#      OBUF                        : 142
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                     771  out of  15360     5%
 Number of Slice Flip Flops:           546  out of  30720     1%
 Number of 4 input LUTs:              1468  out of  30720     4%
 Number of bonded IOBs:                147  out of    448    32%
 Number of GCLKs:                        1  out of     32     3%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 546   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -12

   Minimum period: 7.616ns (Maximum Frequency: 131.299MHz)
   Minimum input arrival time before clock: 8.995ns
   Maximum output required time after clock: 11.597ns
   Maximum combinational path delay: 12.976ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 7.616ns (frequency: 131.299MHz)
  Total number of paths / destination ports: 1911631 / 545
-------------------------------------------------------------------------
Delay:               7.616ns (Levels of Logic = 27)
  Source:            onthefly1/Q1_0 (FF)
  Destination:       onthefly1/QM1_68 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: onthefly1/Q1_0 to onthefly1/QM1_68
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              7   0.272   0.619  onthefly1/Q1_0 (onthefly1/Q1_0)
     LUT4_L:I0->LO         1   0.147   0.000  Mcompar__n0005_norlut (N7)
     MUXCY:S->O            1   0.278   0.000  Mcompar__n0005_norcy
(Mcompar__n0005_nor_cyo)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_0
(Mcompar__n0005_nor_cyo1)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_1
(Mcompar__n0005_nor_cyo2)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_2
(Mcompar__n0005_nor_cyo3)
```

```
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_3
(Mcompar__n0005_nor_cyo4)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_4
(Mcompar__n0005_nor_cyo5)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_5
(Mcompar__n0005_nor_cyo6)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_6
(Mcompar__n0005_nor_cyo7)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_7
(Mcompar__n0005_nor_cyo8)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_8
(Mcompar__n0005_nor_cyo9)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_9
(Mcompar__n0005_nor_cyo10)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_10
(Mcompar__n0005_nor_cyo11)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_11
(Mcompar__n0005_nor_cyo12)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_12
(Mcompar__n0005_nor_cyo13)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_13
(Mcompar__n0005_nor_cyo14)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_14
(Mcompar__n0005_nor_cyo15)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_15
(Mcompar__n0005_nor_cyo16)
     MUXCY:CI->O          17   0.280   0.638  Mcompar__n0005_norcy_rn_16
(Mcompar__n0005_nor_cyo17)
     LUT4:I2->O            4   0.147   0.540
adder1/Madd__AUX_207_Mxor_Result<1>_Result1 (VC<65>)
     LUT2_L:I1->LO         1   0.147   0.000  hybridOnlineMultDivlut (N26)
     MUXCY:S->O            1   0.278   0.000  hybridOnlineMultDivcy
(hybridOnlineMultDiv_cyo)
     MUXCY:CI->O           1   0.034   0.000  hybridOnlineMultDiv__n0001<1>cy
(hybridOnlineMultDiv__n0001<1>_cyo)
     XORCY:CI->O          30   0.273   1.036  hybridOnlineMultDiv__n0001<2>_xor
(_n0001<2>)
     LUT2_D:I0->O         18   0.147   0.601  Mrom_data_Mrom__n00071_SW0 (N1774)
     LUT4_D:I3->O         69   0.147   1.043  Mrom_data_Mrom__n00071 (qD<0>)
     LUT3_L:I2->LO         1   0.147   0.000  onthefly1/QM1_69_rstpot (N1485)
     FDC:D                     0.297          onthefly1/QM1_69
    ----------------------------------------
    Total                      7.616ns (3.138ns logic, 4.478ns route)
                                       (41.2% logic, 58.8% route)

=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 836524 / 142
-------------------------------------------------------------------------
Offset:              11.597ns (Levels of Logic = 29)
  Source:            onthefly1/Q1_0 (FF)
  Destination:       RS<67> (PAD)
  Source Clock:      clk rising

  Data Path: onthefly1/Q1_0 to RS<67>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              7   0.272   0.619  onthefly1/Q1_0 (onthefly1/Q1_0)
     LUT4_L:I0->LO         1   0.147   0.000  Mcompar__n0005_norlut (N7)
     MUXCY:S->O            1   0.278   0.000  Mcompar__n0005_norcy
(Mcompar__n0005_nor_cyo)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_0
(Mcompar__n0005_nor_cyo1)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_1
(Mcompar__n0005_nor_cyo2)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_2
(Mcompar__n0005_nor_cyo3)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_3
(Mcompar__n0005_nor_cyo4)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_4
(Mcompar__n0005_nor_cyo5)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_5
(Mcompar__n0005_nor_cyo6)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_6
(Mcompar__n0005_nor_cyo7)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_7
(Mcompar__n0005_nor_cyo8)
```

```
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_8
(Mcompar__n0005_nor_cyo9)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_9
(Mcompar__n0005_nor_cyo10)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_10
(Mcompar__n0005_nor_cyo11)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_11
(Mcompar__n0005_nor_cyo12)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_12
(Mcompar__n0005_nor_cyo13)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_13
(Mcompar__n0005_nor_cyo14)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_14
(Mcompar__n0005_nor_cyo15)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0005_norcy_rn_15
(Mcompar__n0005_nor_cyo16)
     MUXCY:CI->O          17   0.280   0.638  Mcompar__n0005_norcy_rn_16
(Mcompar__n0005_nor_cyo17)
     LUT4:I2->O            4   0.147   0.540
adder1/Madd__AUX_207_Mxor_Result<1>_Result1 (VC<65>)
     LUT2_L:I1->LO         1   0.147   0.000  hybridOnlineMultDivlut (N26)
     MUXCY:S->O            1   0.278   0.000  hybridOnlineMultDivcy
(hybridOnlineMultDiv_cyo)
     MUXCY:CI->O           1   0.034   0.000  hybridOnlineMultDiv__n0001<1>cy
(hybridOnlineMultDiv__n0001<1>_cyo)
     XORCY:CI->O          30   0.273   1.036  hybridOnlineMultDiv__n0001<2>_xor
(_n0001<2>)
     LUT2_D:I0->O         18   0.147   0.601  Mrom_data_Mrom__n00071_SW0 (N1774)
     LUT4_D:I3->O         69   0.147   0.995  Mrom_data_Mrom__n00071 (qD<0>)
     LUT4_D:I3->O          1   0.147   0.529
adder2/Madd__AUX_195_Mxor_Result<0>_Result1 (WS<52>)
     LUT2:I1->O            1   0.147   0.394  RS<52>1 (RS_52_OBUF)
     OBUF:I->O                 3.255          RS_52_OBUF (RS<52>)
    ----------------------------------------
    Total                    11.597ns (6.243ns logic, 5.354ns route)
                                      (53.8% logic, 46.2% route)
```

C.3 Online Multiplication

```
HDL Synthesis Report

Macro Statistics
# ROMs                                                     : 1
 8x2-bit ROM                                               : 1
# Adders/Subtractors                                       : 273
 1-bit adder carry out                                     : 137
 2-bit adder                                               : 135
 4-bit adder                                               : 1
# Registers                                                : 5
 64-bit register                                           : 3
 68-bit register                                           : 2
# Xors                                                     : 1
 1-bit xor2                                                : 1

*                        Advanced HDL Synthesis                          *

WARNING:Xst:2404 -  FFs/Latches <PC1<67:65>> (without init value) have a constant value
of 0 in block <CsonlineMultiplier>.
WARNING:Xst:1710 - FF/Latch  <PC1_64> (without init value) has a constant value of 0 in
block <CsonlineMultiplier>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <PS1_0> (without init
value) has a constant value of 0 in block <CsonlineMultiplier>.

Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                                     : 1
 8x2-bit ROM                                               : 1
# Adders/Subtractors                                       : 273
 1-bit adder carry out                                     : 137
 2-bit adder                                               : 135
 4-bit adder                                               : 1
# Registers                                                : 323
 Flip-Flops                                                : 323
# Xors                                                     : 1
 1-bit xor2                                                : 1

*                          Low Level Synthesis                           *
Loading device for application Rf_Device from file '4vsx35.nph' in environment
C:\Xilinx.

Optimizing unit <CsonlineMultiplier> ...

Optimizing unit <compressor> ...

Mapping all equations...
WARNING:Xst:1710 - FF/Latch  <PC1_63> (without init value) has a constant value of 0 in
block <CsonlineMultiplier>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <PC1_62> (without init
value) has a constant value of 0 in block <CsonlineMultiplier>.
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block CsonlineMultiplier, actual ratio is 1.

*                             Final Report                               *
Final Results
RTL Top Level Output File Name     : CsonlineMultiplier.ngr
Top Level Output File Name         : CsonlineMultiplier
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 6

Cell Usage :
# BELS                             : 463
#      GND                         : 1
#      LUT2                        : 3
#      LUT2_L                     : 3
#      LUT3                       : 130
#      LUT3_L                     : 122
#      LUT4                       : 69
#      LUT4_D                     : 124
#      LUT4_L                     : 10
#      MUXF5                      : 1
# FlipFlops/Latches               : 321
#      FDC                        : 320
#      FDP                        : 1
# Clock Buffers                   : 1
#      BUFGP                      : 1
# IO Buffers                      : 5
#      IBUF                       : 3
#      OBUF                       : 2
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                    245  out of  15360     1%
 Number of Slice Flip Flops:          321  out of  30720     1%
 Number of 4 input LUTs:              461  out of  30720     1%
 Number of bonded IOBs:                 6  out of    448     1%
 Number of GCLKs:                       1  out of     32     3%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 321   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -12

   Minimum period: 3.388ns (Maximum Frequency: 295.120MHz)
   Minimum input arrival time before clock: 5.017ns
   Maximum output required time after clock: 7.046ns
   Maximum combinational path delay: 8.675ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock ‘clk’
  Clock period: 3.388ns (frequency: 295.120MHz)
  Total number of paths / destination ports: 1523 / 320
-------------------------------------------------------------------------
Delay:               3.388ns (Levels of Logic = 5)
  Source:            x_rnm0_63 (FF)
  Destination:       PS1_67 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: x_rnm0_63 to PS1_67
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              3   0.272   0.581  x_rnm0_63 (x_rnm0_63)
     LUT2:I0->O            3   0.147   0.541  yX<63>1 (yX<63>)
     LUT4_L:I1->LO         1   0.147   0.000
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111_F (N395)
     MUXF5:I0->O           4   0.291   0.414
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111 (PC<65>)
     LUT4:I3->O            1   0.147   0.403  Mxor_W_Result1_SW0 (N387)
     LUT4_L:I3->LO         1   0.147   0.000  Mxor_W_Result1 (W)
     FDC:D                     0.297          PS1_67
    ----------------------------------------
    Total                      3.388ns (1.448ns logic, 1.940ns route)
```

```
                                       (42.7% logic, 57.3% route)

=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 918 / 257
-------------------------------------------------------------------------
Offset:              5.017ns (Levels of Logic = 6)
  Source:            y (PAD)
  Destination:       PS1_67 (FF)
  Destination Clock: clk rising

  Data Path: y to PS1_67
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           258   0.754   1.728  y_IBUF (y_IBUF)
     LUT2:I1->O            3   0.147   0.541  yX<63>1 (yX<63>)
     LUT4_L:I1->LO         1   0.147   0.000
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111_F (N395)
     MUXF5:I0->O           4   0.291   0.414
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111 (PC<65>)
     LUT4:I3->O            1   0.147   0.403  Mxor_W_Result1_SW0 (N387)
     LUT4_L:I3->LO         1   0.147   0.000  Mxor_W_Result1 (W)
     FDC:D                     0.297          PS1_67
    ----------------------------------------
    Total                      5.017ns (1.930ns logic, 3.087ns route)
                                       (38.5% logic, 61.5% route)

=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 108 / 2
-------------------------------------------------------------------------
Offset:              7.046ns (Levels of Logic = 6)
  Source:            x_rnm0_63 (FF)
  Destination:       z_n (PAD)
  Source Clock:      clk rising

  Data Path: x_rnm0_63 to z_n
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              3   0.272   0.581  x_rnm0_63 (x_rnm0_63)
     LUT2:I0->O            3   0.147   0.541  yX<63>1 (yX<63>)
     LUT4_L:I1->LO         1   0.147   0.000
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111_F (N395)
     MUXF5:I0->O           4   0.291   0.580
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111 (PC<65>)
     LUT4:I0->O            2   0.147   0.543  CsonlineMultiplier_V<1>cy11
(CsonlineMultiplier_V<1>_cyo)
     LUT4:I1->O            1   0.147   0.394  Mrom_data_Mrom__n0001 (z_n_OBUF)
     OBUF:I->O                 3.255          z_n_OBUF (z_n)
    ----------------------------------------
    Total                      7.046ns (4.406ns logic, 2.640ns route)
                                       (62.5% logic, 37.5% route)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 52 / 2
-------------------------------------------------------------------------
Delay:               8.675ns (Levels of Logic = 7)
  Source:            y (PAD)
  Destination:       z_n (PAD)

  Data Path: y to z_n
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           258   0.754   1.728  y_IBUF (y_IBUF)
     LUT2:I1->O            3   0.147   0.541  yX<63>1 (yX<63>)
     LUT4_L:I1->LO         1   0.147   0.000
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111_F (N395)
     MUXF5:I0->O           4   0.291   0.580
compressor1/Madd__AUX_66_Mxor_Result<1>_Result1111 (PC<65>)
     LUT4:I0->O            2   0.147   0.543  CsonlineMultiplier_V<1>cy11
(CsonlineMultiplier_V<1>_cyo)
     LUT4:I1->O            1   0.147   0.394  Mrom_data_Mrom__n0001 (z_n_OBUF)
     OBUF:I->O                 3.255          z_n_OBUF (z_n)
    ----------------------------------------
    Total                      8.675ns (4.888ns logic, 3.787ns route)
                                       (56.3% logic, 43.7% route)

=========================================================================
```

C.3 Online Division

```
HDL Synthesis Report

Macro Statistics
# ROMs                                               : 3
 16x2-bit ROM                                        : 1
 4x1-bit ROM                                         : 1
 6x6-bit ROM                                         : 1
# Adders/Subtractors                                 : 521
 1-bit adder carry out                               : 260
 2-bit adder                                         : 260
 5-bit adder                                         : 1
# Counters                                           : 1
 8-bit up counter                                    : 1
# Registers                                          : 10
 128-bit register                                    : 3
 130-bit register                                    : 2
 64-bit register                                     : 5
# Comparators                                        : 3
 64-bit comparator greater                           : 2
 8-bit comparator greater                            : 1
# Multiplexers                                       : 4
 1-bit 4-to-1 multiplexer                            : 1
 124-bit 4-to-1 multiplexer                          : 1
 130-bit 4-to-1 multiplexer                          : 2


*                       Advanced HDL Synthesis                          *

WARNING:Xst:1710 - FF/Latch  <WS1_0> (without init value) has a constant value of 0 in
block <onlineDivision>.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch  <WC1_0> (without init
value) has a constant value of 0 in block <onlineDivision>.

Advanced HDL Synthesis Report

Macro Statistics
# ROMs                                               : 3
 16x2-bit ROM                                        : 1
 4x1-bit ROM                                         : 1
 6x6-bit ROM                                         : 1
# Adders/Subtractors                                 : 521
 1-bit adder carry out                               : 260
 2-bit adder                                         : 260
 5-bit adder                                         : 1
# Counters                                           : 1
 8-bit up counter                                    : 1
# Registers                                          : 962
 Flip-Flops                                          : 962
# Comparators                                        : 3
 64-bit comparator greater                           : 2
 8-bit comparator greater                            : 1
# Multiplexers                                       : 4
 1-bit 4-to-1 multiplexer                            : 1
 124-bit 4-to-1 multiplexer                          : 1
 130-bit 4-to-1 multiplexer                          : 2


*                            Final Report                               *
Final Results
RTL Top Level Output File Name     : onlineDivision.ngr
Top Level Output File Name         : onlineDivision
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# Ios                              : 7

Cell Usage :
# BELS                             : 1813
#      BUF                         : 1
#      GND                         : 1
#      INV                         : 1
#      LUT2                        : 206
#      LUT2_D                      : 4
#      LUT2_L                      : 184
#      LUT3                        : 194
#      LUT3_D                      : 2
#      LUT4                        : 491
#      LUT4_D                      : 263
#      LUT4_L                      : 425
#      MUXCY                       : 36
#      VCC                         : 1
#      XORCY                       : 4
# FlipFlops/Latches                : 842
#      FDC                         : 840
#      FDP                         : 2
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 6
#      IBUF                        : 4
#      OBUF                        : 2
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 4vsx35ff668-12

 Number of Slices:                      946  out of  15360     6%
 Number of Slice Flip Flops:            842  out of  30720     2%
 Number of 4 input LUTs:               1769  out of  30720     5%
 Number of bonded IOBs:                   7  out of    448     1%
 Number of GCLKs:                         1  out of     32     3%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 842   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -12

   Minimum period: 7.339ns (Maximum Frequency: 136.257MHz)
   Minimum input arrival time before clock: 8.812ns
   Maximum output required time after clock: 9.952ns
   Maximum combinational path delay: 11.339ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 7.339ns (frequency: 136.257MHz)
  Total number of paths / destination ports: 1414485 / 841
-------------------------------------------------------------------------
Delay:               7.339ns (Levels of Logic = 25)
  Source:            onthefly1/Q1_0 (FF)
  Destination:       WS1_6 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: onthefly1/Q1_0 to WS1_6
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              5   0.272   0.588  onthefly1/Q1_0 (onthefly1/Q1_0)
     LUT4_L:I0->LO         1   0.147   0.000  Mcompar__n0008_norlut (N30)
     MUXCY:S->O            1   0.278   0.000  Mcompar__n0008_norcy
(Mcompar__n0008_nor_cyo)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_0
(Mcompar__n0008_nor_cyo1)
```

```
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_1
(Mcompar__n0008_nor_cyo2)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_2
(Mcompar__n0008_nor_cyo3)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_3
(Mcompar__n0008_nor_cyo4)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_4
(Mcompar__n0008_nor_cyo5)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_5
(Mcompar__n0008_nor_cyo6)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_6
(Mcompar__n0008_nor_cyo7)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_7
(Mcompar__n0008_nor_cyo8)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_8
(Mcompar__n0008_nor_cyo9)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_9
(Mcompar__n0008_nor_cyo10)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_10
(Mcompar__n0008_nor_cyo11)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_11
(Mcompar__n0008_nor_cyo12)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_12
(Mcompar__n0008_nor_cyo13)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_13
(Mcompar__n0008_nor_cyo14)
     MUXCY:CI->O           7   0.280   0.619  Mcompar__n0008_norcy_rn_14
(Mcompar__n0008_nor_cyo15)
     LUT4:I0->O            2   0.147   0.583
adder1/Madd__AUX_128_Mxor_Result<0>_Result1 (VS<125>)
     LUT2_D:I0->LO         1   0.147   0.000  onlineDivisionlut (N2593)
     MUXCY:S->O            1   0.278   0.000  onlineDivisioncy (onlineDivision_cyo)
     MUXCY:CI->O           1   0.034   0.000  onlineDivision_V<1>cy
(onlineDivision_V<1>_cyo)
     XORCY:CI->O         146   0.273   1.409  onlineDivision_V<2>_xor (V<2>)
     LUT2_D:I0->O         13   0.147   0.547  Ker01_SW0 (N1870)
     LUT4_D:I3->O          8   0.147   0.524  Ker01_1 (Ker01)
     LUT4_L:I2->LO         1   0.147   0.000  _n0001<26> (_n0001<26>)
     FDC:D                     0.297          WS1_26
    ----------------------------------------
    Total                      7.339ns (3.070ns logic, 4.269ns route)
                                       (41.8% logic, 58.2% route)

=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 4488 / 2
-------------------------------------------------------------------------
Offset:              9.952ns (Levels of Logic = 23)
  Source:            onthefly1/Q1_0 (FF)
  Destination:       q (PAD)
  Source Clock:      clk rising

  Data Path: onthefly1/Q1_0 to q
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              5   0.272   0.588  onthefly1/Q1_0 (onthefly1/Q1_0)
     LUT4_L:I0->LO         1   0.147   0.000  Mcompar__n0008_norlut (N30)
     MUXCY:S->O            1   0.278   0.000  Mcompar__n0008_norcy
(Mcompar__n0008_nor_cyo)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_0
(Mcompar__n0008_nor_cyo1)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_1
(Mcompar__n0008_nor_cyo2)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_2
(Mcompar__n0008_nor_cyo3)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_3
(Mcompar__n0008_nor_cyo4)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_4
(Mcompar__n0008_nor_cyo5)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_5
(Mcompar__n0008_nor_cyo6)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_6
(Mcompar__n0008_nor_cyo7)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_7
(Mcompar__n0008_nor_cyo8)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_8
(Mcompar__n0008_nor_cyo9)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_9
```

```
(Mcompar__n0008_nor_cyo10)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_10
(Mcompar__n0008_nor_cyo11)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_11
(Mcompar__n0008_nor_cyo12)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_12
(Mcompar__n0008_nor_cyo13)
     MUXCY:CI->O           1   0.034   0.000  Mcompar__n0008_norcy_rn_13
(Mcompar__n0008_nor_cyo14)
     MUXCY:CI->O           7   0.280   0.453  Mcompar__n0008_norcy_rn_14
(Mcompar__n0008_nor_cyo15)
     LUT4_D:I3->O          9   0.147   0.495  Mrom_data_Ublock1/Mrom_u_rnm01 (N3)
     LUT4:I3->O            2   0.147   0.408
adder1/Madd__AUX_130_Mxor_Result<0>_Result1 (VS<127>)
     MUXCY:DI->O           1   0.280   0.000  onlineDivision_V<2>cy
(onlineDivision_V<2>_cyo)
     XORCY:CI->O         159   0.273   1.332  onlineDivision_V<3>_xor (V<3>)
     LUT4:I2->O           65   0.147   0.973  Mrom_data_Mrom__n00121 (q_OBUF)
     OBUF:I->O                 3.255          q_OBUF (q)
    ----------------------------------------
    Total                      9.952ns (5.702ns logic, 4.250ns route)
                                       (57.3% logic, 42.7% route)

CPU : 53.77 / 54.08 s | Elapsed : 54.00 / 54.00 s
```

Bibliography

[1] J.G. Proakis and D.G. Manolakis, Digital Signal Processing. 3rd edition, Prentice-Hall, NJ, 1998.

[2] W. G. Natter and B. Nowrouzian, "Digit-serial online arithmetic for high-speed digital signal processing applications," in Signals, Systems and Computers 2001, vol.1, pp. 171-176.

[3] M. Lapointe, H.T. Huynh, and P. Fortier,"Systematic design of pipelined recursive filters," IEEE Trans. Computers, vol. 42, no. 4, pp. 413-426, April, 1993.

[4] R. H. Brackert, M. D. Ercegovac, and A. N. Willson, "Design of an online multiply-add module for recursive digital filters," in Proc. 9th IEEE Symp. Computer Arithmetic, 1989, pp. 34-41.

[5] D. Lau , A. Schneider, M. D. Ercegovac , J. Villasenor, A FPGA-based Library for On-Line Signal Processing, Journal of VLSI Signal Processing Systems, v.28 n.1-2, p.129-143, May-June 2001.

[6] M. Ercegovac, T. Lang, On-line arithmetic for DSP applications, 32nd Midwest Symposium on Circuits and Systems, 1989, pp. 365–368.

[7] M.D. Ercegovac and T. Lang, "Most-significant-digit-first and online arithmetic approaches for the design of recursive filters," in Proc. 23rd Asilomar Conf. Signals, Systems, Computers, 1989, pp. 7-11.

[8] R.H. Brackert, A.N. Wilson, and M.D. Ercegovac. A high-speed recursive digital Fiter using on-line arithmetic. In Proc. Of the ISCAS'89, pages 1552{1555, Portland, 1989. IEEE Service Center, Picataway.

[9] J.S. Fernando and M.D. Ercegovac, "Conventional and Online Arithmetic Designs for High-Speed Recursive Digital Filters," Proc. Fifth IEEE Workshop VLSI Signal Processing, pp. 81-90, Oct. 1992.

[10] S E McQuillan and J V McCanny " A Systematic Methodology for the design of recursive Digital Filters", IEEE Trans. On Computers, Aug. 1995, vol. 44, no 8, pp. 971-982.

[11] J.S. Fernando and M.D. Ercegovac. On-line arithmetic modules for recursive digital fiters. In Proc. Of the 26th Annual Asilomar Conference on Signals, Systems and Computers, volume 2, pages 681-685. IEEE Society Press, Los Alamitos, 1992.

[12] D. Lau, A. Schneider, M.D. Ercegovac, J. Villasenor, FPGA-based structures for on-line FFT and DCT, Proceedings of the Seventh IEEE Symposium Field-Programmable Custom Computing Machines, 1999, pp. 310–311.

[13] M. Moinuddin, "Normalized Time Varying Mixed Norm LMS-LMF Adaptive Algorithm", master's thesis report, KFUPM, Saudi Arabia, May 2001.

[14] A. Elshafei, A. Zerquine and A. Bouhraoua, "A Scalable FPGA Implementation for Mixed-Norm LMS-LMF Adaptive Filters", IWCMC 2009 Wireless LANs and Wireless PANs Symposium, June 2009.

[15] C. K. Koc and C. Y. Hung. Fast algorithm for modular reduction. *IEE Proceedings – Computers and Digital Techniques,* 145(4):265-271, July 1998.

[16] M. H. Hayes, (1996). "9.4: Recursive Least Squares", *Statistical Digital Signal Processing and Modeling*. Wiley, pg.541.

[17] A. Zerguine and T. Aboulnasr, "A variable weight mixed-norm adaptive algorithm," International Conference on Acoustics, Speech and Signal Processing, ICASSP 2002.

[18] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[19] M.D. Ercegovac, "On-Line Arithmetic: An Overview", Proc. SPLE Conference on Real-Time Signal Processing, 1984.

[20] K. S. Trivedi and M. D. Ercegovac, "Online algorithms for division and multiplication," IEEE Trans. Comput., vol. C-26, July 1977

[21] S. Rajagopal, J. Cavallaro, On-line arithmetic for detection in digital communication receivers, 15th IEEE Symposium on Computer Arithmetic, 2001, pp. 257–265.

[22] M. Dimmler, A. Tisserand, U. Holmberg, and R. Longchamp. On-Line Arithmetic for Real-Time Control of Microsystems**.** *IEEE/ASME Transactions on Mechatronics*, 4(2):213-217, June 1999.

[23] P. Kang and G. Tu, Online Arithmetic Algorithms for Efficient Implementation, Comput. Sci. Dep., Univ. California, Los Angeles, 1990.

[24] A. F. Tenca, S. U. Hussaini, "A Design of Radix-2 On-line Division Using LSA Organization," Proceedings of the 15th International Symposium on Computer Arithmetic, June 11-13, 2001, Vail, Colorado, pp. 266-273

[25] M. D. Ercegovac , T Lang, On-the-fly conversion of redundant into conventional representations, IEEE Transactions on Computers, v.36 n.7, p.895-897, July 1987

[26] R. P. Brent and H. T. Kung, “A Regular Layout for Parallel Adders”, IEEE Trans. Comput., vol. C-31, no. 3, pp. 260-264, March 1982.

[27] J. Knight, “Logarithmic Circuits,” in *Lecture Notes: Digital Circuits.* Ottawa, ON: Carleton University, 2003, pp. 97-122.

[28] S. M. Muller , Wolfgang Paul, Computer Architecture: Complexity and Correctness, Springer-Verlag New York, Inc., Secaucus, NJ, 2000

[29] A. Amin, “Carry-Save Addition,” in *Lecture Notes: Computer Arithmetic.* Dhahran, Saudi Arabia: KFUPM, 2007, pp. 9-10.

[30] A. Gutub, M. Ibrahim, and M. A. Araman, "Super Pipelined Digit Serial Adders for Multimedia and e-Security", IEEE 1st International Computer Engineering Conference on New Technologies for the Information Society, Cairo, EGYPT, pp. 558-561, Dec 2004