UNIVERSITY OF CALIFORNIA

Los Angeles

Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Coprocessor Architectures

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Alexandre Ferreira Tenca

1998

The dissertation of Alexandre Ferreira Tenca is approved.

Prof. Dr. Willian Newman

Prof. Dr. David Rennels

Prof. Dr. Jason Cong

Prof. Dr. Milos D. Ercegovac, Committee Chair

University of California, Los Angeles

1998

ABSTRACT OF THE DISSERTATION

Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Coprocessor Architectures

by

Alexandre Ferreira Tenca

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1998

Professor Prof. Dr. Milos D. Ercegovac, Chair

This is the abstract .....

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

Thanks to .....

VITA

1981	B.S., Electrical Engineering, University of São Paulo, São Paulo, Brazil
1990	M.S., Electrical Engineering, University of São Paulo, São Paulo, Brazil
1994	M.S., Computer Science, University of California, Los Angeles

PUBLICATIONS

Alexandre F. Tenca and Milos D. Ercegovac, "A High-Radix Multiplier Design for Variable Long-Precision Computations", 31st Asilomar Conference on Signals, Systems and Computers, Nov. 1997.

Alexandre F. Tenca and Milos D. Ercegovac, "Synchronous Up/Down Binary Counter for LUT FPGAs with Counting Frequency Independent of Counter Size", FPGA97 - ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 159-165, Feb. 1997, Monterey.

Alexandre F. Tenca and Milos D. Ercegovac, "High-radix Digit-slices for On-line Computations", Proceedings of SPIE Conference on High Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic, vol. 2994, 1996, pp. 14-25.

Chapter 1

Introduction

Programmable logic has provided a new hardware design space that allows the exploration of new techniques to obtain more performance. The most important programmable device is the Field-Programmable Gate Array (FPGA) that combine the regularity of gate arrays and the programmability of random access memories. FPGAs have evolved in different directions, dictated by the final application of the devices. One of these applications is the "reconfigurable computing." The name suggests that a task or part of a task performed in the general purpose computer can be downloaded into the programmable devices and be executed at hardware speed, which may be many times faster than the software speed (program execution in the computer).

A Reconfigurable Coprocessor Architecture (RCAr) is a digital system that combines FPGAs, fixed logic and memory. It is attached to the main processor and is able to handle special tasks faster than the processor.

The organization of the coprocessor may also be tailored to the particular application. In this thesis we study and propose an RCAr that supports the efficient implementation of circuits for variable long-precision arithmetic (VLPA). VLPA is used to improve the accuracy of computer calculations, vital in many areas, such as: computational geometry, N-body problem and criptography.

A few research works have produced hardware solutions for the VLPA problem. Only one work was done by this time that propose a hardware structure for VLPA in a reconfigurable architecture. In this dissertation we present the research on the extension of the on-line arithmetic concepts to the design of hardware algorithms for VLPA, which is also suitable for implementation using RCArs.

1.1 The need for VLPA

Throughout this work, the hardware precision, or sometimes called the width of the hardware data path, is represented by n. The precision required in an operation is represented by m.

VLPA encompass solutions for long precision computation that can go beyond the limits of the available hardware, or $m > n$. The objective of the research is to propose arithmetic algorithms that allow scalability by reutilization of the available hardware resources until the desired precision is attained.

IEEE Std. 754 [75485, Coo80], for example, has floating-point (FP) formats with few precision alternatives for the exponent and significand (mantissa). Computer systems that conform to this standard often have only 64-bit data paths. This number of bits is not needed in many applications, and yet it is insufficient for other applications in which the computation can produce results completely inaccurate without warning. Examples of this problem are presented in the literature [Lyn95, Sch96]. Let us reproduce one of them as an illustration. Consider the dot-product computation of two vectors A and B^T using IEEE double-precision arithmetic and *exact* arithmetic:

$$A = [-10^{18}, 2246, 10^{27}, 10^{25}, 22, 10^5]$$

$$B^T = [10^{38}, 33, 10^{29}, -10^{22}, 1044, 10^{42}]$$

$$IEEE \rightarrow A \times B^T = 0$$

$$exact \rightarrow A \times B^T = 97,086$$

the result of the computation using IEEE has a large error compared to the exact result.

Real numbers cannot be always precisely represented by floating-point numbers, with the exact value being between two consecutive FP numbers. The mapping from the real number to a machine representable floating-point number is done by rounding. This process creates errors that propagate in the computation and can

make the results of some calculation meaningless. The utilization of FP numbers leads to two main problems: catastrophic cancellation and roundoff errors, caused by the discrete nature of the FP representation. Roundoff error occurs because the significand of a FP number has a limited number of bits. Thus, a real number must be rounded in order to be represented with the limited precision significand. Catastrophic cancellation results from the subtraction of FP numbers with close values, that results in a value with less significant bits than the initial operands. One of the goals of numerical analysis is to determine the accuracy of numerical methods, and evaluate analitically if the result of a computational algorithm can be trusted or not. This is so called "problem of computer credibility" [Knu69, Lyn95]. Accuracy cannot be obtained in some cases if there is not enough precision in the arithmetic operations and operands. A statement found in many papers in the area is that "if more accuracy is needed, more bits must be employed".

Clearly, computer systems should be able to work with *variable precision*, in order to have efficient implementations. The term is used in the sense that the user or software should be able to adjust the precision of an arithmetic operation, ideally constrained only by the available memory in the system.

To solve the problem of computer accuracy, researchers proposed alternative number systems and arithmetic concepts [AH83, Vui90, MM94, Neu90, Moo79] (Section 1.2). These approaches are implemented in the digital computer in the form of software libraries, programming languages or coprocessors. Various implementations already available to deal with long-precision computations are presented in Sections 1.3 and 1.4. The classification of the available solutions for this problem is presented in Figure 1.1. VLP applications are built using high-level languages and libraries that provide data types and operations that support variable precision calculations. Coprocessors are used by these software modules to improve the most time consuming operations.

Exact arithmetic based on arithmetic libraries are slow. These libraries define the long-precision floating-point numbers in two different ways: as a *multiple digit*

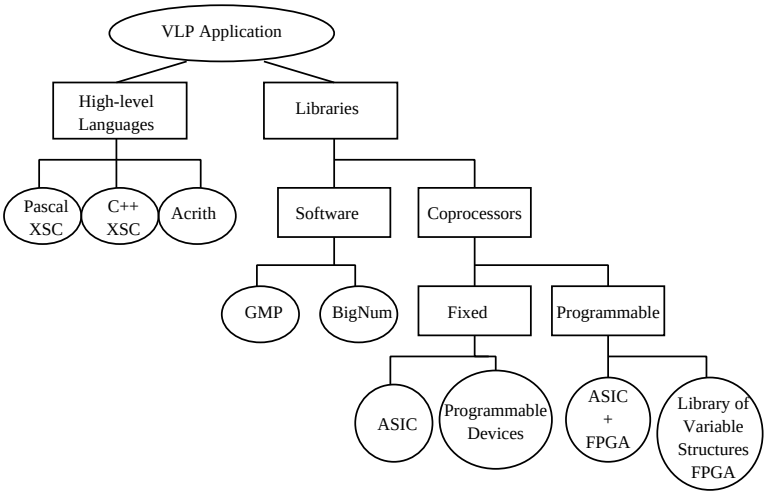

Figure 1.1: Solutions given to Very Long Precision Computation

format or as a *multiple term format*. In the multiple digit format the number is represented as a sequence of digits that form the significand and a signed exponent. The multiple term format considers the long-precision number as a collection of ordinary floating-point numbers, each one with its own significand and exponent. There are advantages and disadvantages in each scheme, but in particular, the multiple digit format is the one that can more compactly represent most numbers, since only one exponent is stored.

The next sections present commonly used solutions for the accuracy problem in computer systems, focusing on the aspect of long-precision provision that is required.

1.2 Alternative Arithmetic Systems

Some of the arithmetic systems proposed to solve the accuracy problem are:

- Interval arithmetic: considers each number as a pair of FP numbers, the upper and lower bounds for the exact value, called an *interval*. This concept, combined with the control over the precision, allows the user to keep the cal-

culations inside reasonable error bounds. Interval arithmetic is discussed in [AH83, KM81, KM81, Ral90, Neu90, Moo79]. All operations are defined over intervals, and the results are guaranteed to be inside the resulting interval. The advantage of this system is to allow the automatic verification of computed results. A disadvantage of this system is the need to perform the same operation many times in order to obtain the two FP values for the result interval. Interval division, for example, requires up to 4 FP divisions and some comparisons to obtain the result interval. Testing the sign of the interval endpoints it is possible to obtain the interval with only 2 FP operations in most of the cases. Another disadvantage is that interval arithmetic by itself does not provide mechanisms for variable precision computation, but it is a tool that allow the user to verify, at run time, if the present precision is acceptable or not. If the precision is not acceptable, using variable precision arithmetic, the user can increase it and redo the calculations. There is also the problem that "interval numbers" grow too fast, so that the results are too pessimistic.

- Continued fractions: real numbers are exactly represented as fractions of integer values, i.e., as rational numbers. The precision of the real number, depends on the precision of the integers used in the representation. The problem with this concept is the high complexity of the basic arithmetic algorithms [Vui90]. The advantage is the ability to represent real numbers without requiring rounding.

- Staggered arithmetic: considers long-precision numbers (LP) as a set of non-overlapping FP numbers. The resulting number has at least as many significant bits as all significand bits of the FP terms put together. The operations are executed over the set of FP numbers that represent the LP number and generate another LP number. This system takes advantage of the high efficiency of FP arithmetic units [Pri91]. Numbers that have large groups of zeroes are represented by few FP numbers, covering only the non-zero signifi-

cant digits that are far apart, therefore reducing the storage requirement and computation time. The disadvantage of this approach is the complexity of the algorithms for even the basic arithmetic operations. An example can be found in [Pri91] for the addition of two "digits" represented as FP numbers a and b. The complete addition algorithm involves operations over multiple digits. The digit addition is performed by the following algorithm, which satisfies: $a + b = c + d$, where $d = 0$ or $c + d$ is a valid *expansion* (c and d represent the sum values without overlapping). The word fl represents a floating point operation. For each digit, seven floating point operations are executed.

procedure $sum\_term(a, b)$
begin
if $|a| < |b|$
  swap(a,b)
$c := fl(a + b); e := fl(c - a)$
$g := fl(c - e); h := fl(g - a); f := fl(b - h)$
$d := fl(f - e)$
if $fl(d + e) \neq f$
  $c := a; d := b$
return (c,d)
end procedure;

A numeric example follows for 3 digits of precision ($p = 3$), base 10, $a = 42.3$ and $b = 0.324$. The sequence of operations gives the following values: $c = 42.6$, $e = 0.300$, $g = 42.3$, $h = 0.00$, $f = 0.324$ and $d = 0.0240$. Thus, the floating point value $a + b = 42.624$ is represented in precision $p = 3$ by the pair: $(42.6, 2.40 \times 10^{-}2)$. Note that $fl(a + b) = c = 42.6$ has an error or 0.024.

1.3 Software Languages and Libraries for VLPA

Variable precision libraries and languages provide an abstraction level that allows the user to use data types and operations for long and variable precision numbers. They extend the power of an existing hardware creating an environment for larger precision that makes use of the fixed precision available in the processor. Some examples of software language extensions and libraries for long-precision computation are:

- C++ XSC - extended precision C++ language [Wie];
- Numerical Recipes book [P+94] provides several routines for variable precision computation on large precision integer operands;
- GMP - Gnu Multiple Precision Library [Tor93]: a multiprecision package available as public-domain software. A manual of the available routines can be found with the software;
- Maple, Mathematica and Mathlab: these are general computer algebra systems that provide an interactive and easy to program environment. These packages allow the manipulations of numbers with variable precision.
- BigNum [Zim92]: a portable Le-Lisp package for arbitrary-precision integer arithmetic.

Long-precision numbers are represented as arrays of machine words. A word value is considered as a high-radix digit. Therefore, operations performed by the processor become digit operations. Algorithms to deal with digit operations are used to manipulate the long-precision numbers.

The flexibility required in a good software implementation, together with the portability needs, leads to large time overhead that could be avoided in hardware approach. Thus, it is usually the case that an algorithm implemented in hardware will perform better than the same algorithm implemented in software.

The performance is strongly dependent on the selection of the algorithm for the desired precision. It may happen that an algorithm is very efficient for high-precision numbers (hundreds of digits) and performs poorly for low-precision cases. Because of that, libraries select the algorithm to be used, based on the operands' precision.

In general, the long-precision libraries have routines that allow the user to work on different number types (integers, floating point, rational numbers, interval arithmetic numbers) with variable precision. Operations involving long-precision integers are the fastest and the most optimized ones. For efficiency, the inner loop of LP operations may be written in assembly language for each different type of processor. Other data types are manipulated using the efficient long-precision integer routines.

1.4 Existing Coprocessors for Long Precision Computations

There are few coprocessors presented in the literature for long and variable precision computations. Specific machines also exist to deal with long precision numbers for particular applications, like cryptosystems and dot-product computation. Many FPUs use extended precision internally. We are more interested in coprocessors that were designed for a more general application in the VLP arena.

Some of the coprocessors primarily used to increase the performance of arithmetic algorithms are discussed in this section.

1.4.1 Chow's VP Processor

In her thesis work, Catherine Y. Chow [Cho80, Cho83], presented the design of a general purpose variable precision processor for floating-point operations. The main aspects of her design are:

- Generalized the architecture for any digit set, radix and precision, allowing the adjustment of the number of digit slices to the available hardware. This

approach gives a high degree of flexibility and the units (VP modules) can be combined to operate in different modes, from totally parallel to totally serial, depending on the number of modules used. The operations are executed Most Significant digit first.

- The design uses digit recurrent algorithms assuming that one of the operands is available in parallel form.

- Instructions are defined at the VP module to support shifting, rotation, initialization, communication between VP modules, etc. Each VP module has 16 digit slices. Many VP modules can be combined to increase the hardware parallelism. The available hardware can be used repeatedly to obtain the desired precision.

An implementation of this design generated a coprocessor named Cascade [Car89]. This cpprocessor works with radix-16 digit slices. There is not much data on the performance of the processor.

1.4.2 CADAC — Controlled Precision Decimal Arithmetic Unit

This project was developed at University of Toronto [CHH83, HCH91]. CADAC was designed to work with decimal number system (BCD representation) in floating-point format. The arithmetic unit work on decimal digits. The approach avoids I/O errors that occur when converting from decimal to binary. For example: the number 0.31 in decimal is not representable in binary. The use of decimal notation is also more convenient to the user, however the arithmetic circuits for decimal numbers are not as efficient as circuits for binary.

The design uses pipelined stages. The multiplication of 32-digit numbers is done in 30μsec, by the execution of 276 steps, with a cycle time of 108.7 ns. The coprocessor can execute multiplication, addition, subtraction and division.

1.4.3 Coprocessor for Pascal-XSC

Pascal-XSC is an extended precision Pascal language. A chip was implemented by Baumhof [Bau95] to reduce the accumulation of errors during dot-product computation, that is provided by the language as an instruction. It accepts IEEE floating-point format and the most important feature of the processor is a module called *Long Accumulator* (LA) [MRR91, Kno91]. This accumulator is used to store partial results without rounding. A single and final rounding is necessary before the result is delivered to the user.

The performance improvement using this dedicated hardware is significant. However, this hardware solution is limited to the particular case of dot-product computation. Other types of operations cannot take advantage of the coprocessor. Besides that, it uses long precision only internally. Conventional FP number representation is used to exchange data at the user interface.

1.4.4 Interval Arithmetic Coprocessor

A work presented in [SSJ95b, SSJ95a, Sch96] describes algorithms, hardware organization and performance measurements of a coprocessor designed to speedup the execution of a Variable Precision Interval-Arithmetic Package called VPI [Ely93]. Interval arithmetic was already described in Section 1.2. The coprocessor is called VPI Arithmetic Coprocessor (VPIAC).

The architecture of the VPIAC is based on a conventional floating-point unit with a more sophisticated controller (scheduling algorithms) and Long-Accumulator, to avoid propagation of rounding errors in intermediated floating-point operations.

The performance figures shown in Table 1.1, obtained from [SSJ95a], compare the execution of VPI package with and without the coprocessor. The cycle count assumes the operands already in the register file.

Another important information is the number of cycles necessary to perform a single point operation or an interval operation in the coprocessor [SSJ95a]. This

Interval addition execution times (sec)			
Precision (bits)	VPIAC	VPI	Speedup
32	$2.6 \cdot 10^{-7}$	$9.6 \cdot 10^{-5}$	369
64	$3.2 \cdot 10^{-7}$	$1.1 \cdot 10^{-4}$	343
256	$7.0 \cdot 10^{-7}$	$1.9 \cdot 10^{-4}$	271
1024	$2.2 \cdot 10^{-6}$	$5.0 \cdot 10^{-4}$	227
Interval Multiplication execution times (sec)			
Precision (bits)	VPIAC	VPI	Speedup
32	$4.2 \cdot 10^{-7}$	$1.1 \cdot 10^{-4}$	262
64	$5.4 \cdot 10^{-7}$	$1.9 \cdot 10^{-4}$	352
256	$2.7 \cdot 10^{-6}$	$1.7 \cdot 10^{-3}$	630
1024	$3.3 \cdot 10^{-5}$	$2.6 \cdot 10^{-2}$	788
Interval Division execution times (sec)			
Precision (bits)	VPIAC	VPI	Speedup
32	$8.3 \cdot 10^{-7}$	$5.6 \cdot 10^{-4}$	675
64	$1.2 \cdot 10^{-6}$	$1.1 \cdot 10^{-3}$	917
256	$7.8 \cdot 10^{-6}$	$8.2 \cdot 10^{-3}$	1051
1024	$1.0 \cdot 10^{-4}$	$1.0 \cdot 10^{-1}$	1000

Table 1.1: Performance measurement of interval arithmetic on VPI, with and without the arithmetic coprocessor (VPIAC) [SSJ95a]

Operation	Point	Interval
Add/subtract	$2n + 8$	$4n + 12$
Multiply	$n^2 + n + 12$	$2n^2 + 2n + 22$
Square	$n^2/2 + 2n + 12$	$n^2 + 4n + 22$
Divide	$3n^2 + 4n + 20$	$6n^2 + 8n + 38$
Square Root	$3n^2 + 6n + 26$	$6n^2 + 12n + 48$

Table 1.2: Cycle counts for various operations in the VPIAC [SSJ95a]

information is given in Table 1.2. Variable n denotes the number of 32-bit words in the significand of the input operands. The number of cycles includes instruction fetch, data read from internal registers, operation, rounding the result and storing the result in the data registers. The time for data transfer between the coprocessor and the main memory was not included.

No special algorithms for long-precision computation are used. The algorithms presented in the work deal with necessary steps to maintain intervals to the basic arithmetic operations. At the the low level, the algorithms to add, multiply and divide long precision numbers are the classical ones. More details on the algorithm used for VLP operations in this coprocessor are given in Chapter 3.

The coprocessor has a 64×64-bit multiplier, a 128-bit adder, a Long Accumulator of 64 128-bit segments and two 128-bit shifters.

1.4.5 JANUS

Another variable-precision coprocessor was proposed in [GHM89]. This coprocessor uses the on-line arithmetic [Erc84, ET87] to obtain a simple and regular structure that can be used for all basic arithmetic operations. The design considered serial links to interconnect the coprocessor to transputers and ASIC design style for a maximum precision of 600 decimal digits. There is no provision in the chip to allow an extension of precision which makes operations with more than 600 digits very inefficient. Only a 64-digit SBD (radix-2 signed binary digit) multiplier

has been designed. The resulting clock speed was 30 MHz.

1.4.6 VLP Coprocessor for the TM-1

A VLPA coprocessor designed for the Transmogrifier-1 (TM-1) [GKC+94] was presented in [Hsu96]. This coprocessor also uses standard hardware algorithms with modifications for long-precision computations. The work concentrates in the arithmetic structures and coprocessor organizations that would be adequate for the structure of the TM-1. The number of cycles presented in [Hsu96] are reproduced in the next table.

Operation	Number of Cycles
multiplication	$7n^2 + 5n + 20$
division	$6n^2 + 21n + 17$

The design uses 50-bit digits and each cycle takes 130ns.

1.5 VLP Computation and RCArs

It is a fact that Central Processing Units or Arithmetic Coprocessors for general applications are designed for best performance in the most frequent operations. It is not feasible to have a processor that has the best performance for all possible applications. That is the main reason why Reconfigurable Architectures (RArs) are becoming popular. These machines are flexible enough to implement custom solutions, tailored for the particular user problem or application [AH93, VdB92, Wau91, CB92], and for this reason they are also called Custom Computing Machines (CCM). It is very common to see large speedups obtained in RArs, although slow devices are used (price paid for reconfigurability and flexibility). The speedup results from the exploitation of the high level of parallelism that can be attained in hardware, reduced software system overhead and the specialization of the circuits to the task on hand. Some examples of RArs are Anyboard [VdB92], Splash [AH93, Wau91] and Ganglion [CB92]. A list of many programmable boards and systems can be found in [Gui].

One of the first systems using FPGAs and designed for general use were developed at DEC Paris Reseach Laboratory [VBD+96]. The DEC PeRLe-0 and PeRLe-1 are FPGA boards that are attached to the system I/O bus of a host workstation. A mesh topology (nearest-neighbour) is used to interconnect the FPGA chips. Typical applications in the PAM systems are computationally complex but require low communication bandwidth. Some examples are: Long Integer Multiplication, Discrete Cosine Transform (DCT) and an RSA Decrypter.

Splash is another FPGA-based system designed at the Supercomputing Reseach Centre (SRC) that obtained significant speedups in gene pattern matching [AACE94, Hoa93], text searching [PTS93] and heat transfer problem solving based on finite difference method [PA96].

Recent studies consider the integration of a reconfigurable coprocessor with the general processor [RV96]. The work presented in [RV96] concentrates on the properties of the processor/FPGAs interface. The study shows that the speedup obtained with coprocessors is less sensitive to the communication delays when a DMA type of interface is used and most of the task is done by the reconfigurable architecture without CPU intervention. The CPU manipulates the data used by the coprocessor in the next iteration or manipulates the data generated by the coprocessor.

Other more theoretical studies analyze the application of FPGAs to speedup computations in general terms. The applicability of programmable coprocessors is discussed in [ACC96]. It considers the coprocessor connected directly to the main processor. The reconfigurable coprocessor is compared to a Very Long Instruction Word (VLIW) coprocessor architecture. The research results shows that the implementation of addition and multiplication for integer and floating point numbers on FPGA based coprocessors are not the best choice in most of the cases. The applications that show significant speedup are the ones with a limited number of multipliers and with no floating-point operations. The work does not consider the VLP operations. It concludes that the range of applications that can take

advantage of FPGA based coprocessors increase with the chip logical resources available. This tendency is a reality for the future, since the feature size is always being reduced and the size of the chip is also increasing. But it is important to pick up applications that have a reasonable amount of parallelism.

The potential of FPGA-based systems is studied in [DeH96]. The work analyzes the capacity of FPGA organizations, and the results show again that FPGAs will perform much better than general purpose computers in tasks that have significant degree of parallelism.

VLP computation is one of the tasks that fits in the category of applications that are well suited for FPGAs. The same digit operations are executed over and over again, until the final result is obtained. The exact number of digits that becomes the threshold between hardware and software execution depends on particular implementations. It is the type of application that has potential to run much faster in a specific hardware than in a general case processor running a software solution.

1.6 Research Objectives

The main research objectives of this work are:

- Investigation of alternatives for the design of hardware algorithms for VLPA computations focusing in Reconfigurable Architectures (RA).
- Development of VLP arithemtic algorithms using on-line arithmetic.
- Development of area/time estimates for the proposed VLP algorithms for RAs. The model is used as the base for the evaluation of possible alternatives in the implementation of the algorithms and also a first order approximation for the performance evalution suggested in this thesis.
- Proposal of a reconfigurable coprocessor architecture that considers the characteristics of the VLP algorithms and a particular communication mechanism

with the host computer.

- Performance evaluation of the pair host+coprocessor compared with the execution time of the long-precision computation by the host running a software package.

- Definition of a mechanism to intefrate the host and coprocessor in the execution of FP operations. Tasks related to FP operations such as normalization and exponent manipulation may be done by the host processor for performance reasons.

- study the parameters that are necessary to design a coprocessor which allows adjustment of the operation precision. The maximum precision is determined by available memory resources only [CHH83, HCH91].

1.7 Dissertation Outline

The remaining of this dissertation is organized as follows:

- Chapter 2 gives an high-level model for the coprocessor architecture, presents the FPGA architecture considered in the thesis and provides an initial discussion on the software interface that allows the use of the coprocessor operations.

- Chapter 3 presents the variable long-precision algorithms used in most of the software packages. Discuss the hardware implementation of VLP operations in available coprocessors and justifies the use of on-line arithmetic for VLP operations.

- Chapter 4, 5 and 6 describe the algorithms used for VLP multiplication, division and square root. Convergence conditions and implementation issues are discussed.

- Chapter 7 constains details of the coprocessor implementation, such as: digit recoding, number conversion and floating-point operations using the host and coprocessor.

- Chapter 8 gives the area/time estimates of the components used in the VLP algorithms for the particular case of FPGA technology.

- Chapter 9 describes the performance model for the VLP algorithms and provides an evaluation of the system composed of host+coprocessor.

- Chapter 10 present the conclusions for this work and future research.

Chapter 2

Reconfigurable Coprocessor Architecture

This chapter presents the architectural issues of the reconfigurable coprocessor proposed for VLP computations. The coprocessor architecture illustrates the target system for the arithmetic algorithms presented in this thesis and also provides the basis for making a first order performance estimate of the proposed approach. The coprocessor is modeled in terms of required logic, memory, interconnect resources and I/O capabilities. The reconfigurable coprocessor may have its datapath and control modified, as needed, in order to implement directly a particular algorithm.

We first present the coprocessor model architecture, give a general view of the target programmable devices and describe the interface between the coprocessor and the user software.

2.1 Reconfigurable Coprocessor Model

The coprocessor model presented in this section is not designed for general use but for the arithmetic operations proposed in the thesis. Its architecture reflects the characteristics of the algorithms for VLP arithmetic.

The model is shown in Figure 2.1 and considers the following functions:

- **Arithmetic operations**: basic arithmetic operators and other operations required in the arithmetic algorithms, such as number conversion. This function is implemented in the reconfigurable hardware of the coprocessor (Field Programmable Gate Arrays - FPGAs).

- **Local storage**: memory modules necessary to store intermediate results in

VLP operations. Memory resources may be available in the FPGA chip or may be included as extra chips in the coprocessor. The first solution provides a better access time, since the logic resources and memory cells are on the same chip. However, the use of internal memory resources reduces the area available for logic circuits used in VLP operations. Also, the internal memory elements have a small addressing space. The second solution has more delay in data transfer due to crossing chip boundaries. However, for long sequences of intermediate results this is the only practical solution. The transfer of more than one digit per memory access may be used to reduce the transfer delay. Memory elements that has two ports for simultaneous accesses are assumed for two reasons: (1) need for data buffers between host and coprocessor to store data that is generated faster than it is consumed and (2) capability to allow read and write of temporary data in the same cycle in order to maximize data throughput. Data channels between the memory elements and the host or coprocessor have different bandwidths represented as B_H and B_{LM}.

- **Data interface (I/O)**: this function is related to the transfer of operands and results to the system main memory or local memory. The coprocessor has communication channels to the host and local memory. The circuits that implement the communication protocol and DMA are in this category.

- **Memory Interface**: provides the functionality to read/write information to/from the local memory.

- **Control**: responsible for synchronization of tasks performed by all other functions.

- **Configuration**: responsible for donwnloading configuration files into the FPGAs. These files define the circuits that implement the coprocessor functions fully or in part. The function also includes storage for configuration files. The files are transferred by the host in advance, and when needed, they are written

to the FPGA devices in the coprocessor. If the FPGA chips permit partial reconfiguration [?], the bitstreams required to modify the configuration from one operation to the other are also stored in the configuration memory. Partial reconfiguration allows the modification of parts of the logic or modification of interconnections only. This process is faster than the static reconfiguration used in many FPGAs, when the whole circuit must be downloaded. Reconfiguration time is of main importance, since the application program may require successive and different operations in the VLP coprocessor.

The reconfigurable part of the system is composed by the FPGA chips and can accommodate all of these functions. For performance reasons, some of the functions, or parts of them, can be implemented in dedicated hardware. Dedicated circuits should be used to implement functions that are not modified through time. The use of dedicated hardware is not discussed in this thesis.

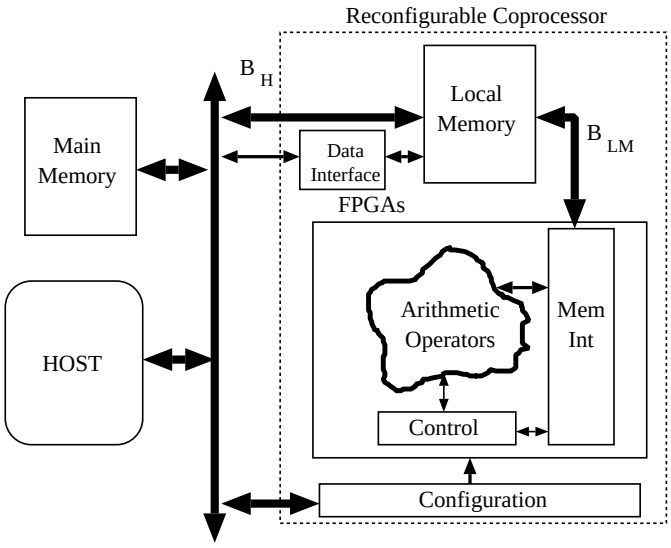

Figure 2.1: Coprocessor model

Once configured for a particular VLP operation, the coprocessor organization for VLP arithmetic is defined as shown in Figure 2.2. In the diagram we abstract the

complexity of the interface between the host and the coprocessor and concentrate on the architectural and design issues of the arithmetic coprocessor. The host is able to access the local memory to transfer input operands and read the result. We are omitting details to increase legibility and present only the main concepts.

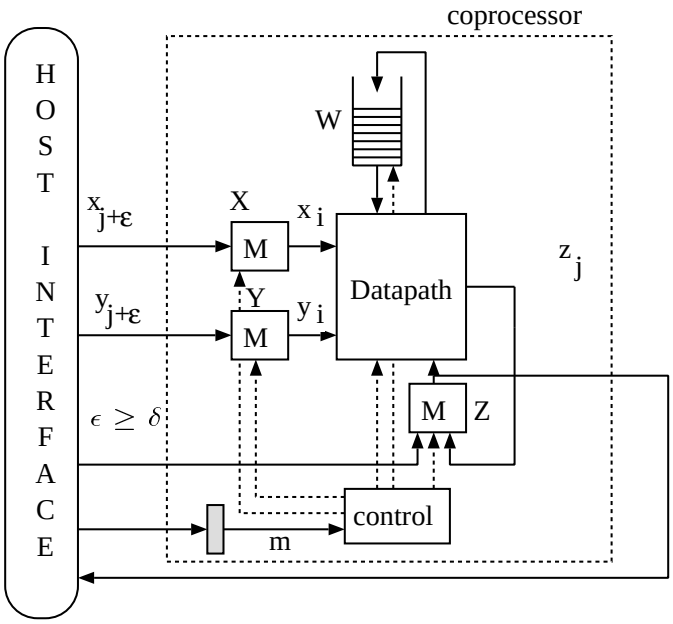

Figure 2.2: Coprocessor Organization for VLP operation

The memory components named X,Y,Z and W are dual-port RAMs for concurrent memory access. Specific locations of these memories store information on the last digits received from the host. Thus, the coprocessor operates based on the availability of input data. If the data is not available, the coprocessor stalls. The same is true for the host, which reads the result digits independently of the coprocessor operation. The memory elements have arbitration to avoid conflicts during memory accesses.

The coprocessor local memory is mapped into the host memory space. The host sees the addresses of the memory elements as a sequence of addresses. Internally, the coprocessor is able to access multiple words of the local memory, in order to

have sufficient data throughput to match the circuit operation.

A word transferred on the bus holds several digits to be manipulated in the coprocessor. The host processor program must pack digits into the bus transfer word to reduce the communication overhead.

2.1.1 FPGA Architecture

FPGAs are modeled as an array of cells or Configurable Logic Units (CLU). A CLU is composed of a Look Up Table (LUT) and a flip-flop. The LUT is configured to implement a particular logic function. In the Xilinx family of programmable devices, the LUT is called a *function generator* (FG). The number of inputs in the table change from one manufacturer to another, or from one family of devices to another. A n-input LUT can implement any logic function of n binary variables. Some FPGAs contain more than one type of LUTs. For example, in the Xilinx XC4000 devices, it is possible to implement the following function types in the CLU:

- $f_1(x_1, x_2, x_3, x_4)$, with $x_i \in \{0, 1\}$;
- $f_2(x_1, x_2, x_3, x_4, x_5)$, with $x_i \in \{0, 1\}$;
- $f_3(f_1(x_1, x_2, x_3, x_4), g_1(y_1, y_2, y_3, y_4), z)$, with x_i, y_i and $z \in \{0, 1\}$.

The flip-flop (FF) is independent from the LUT and can be used by the local LUT or by another logic in another cell. The interconnections are defined at configuration time. Figure 2.3 shows the model of one of these cells. One or more CLUs can be grouped into a Configurable Logic Block (CLB). In the Xilinx XC4000 series of devices we may have 1 or 2 CLUs in one CLB, depending on the function type being implemented. The multiplexer symbol (in gray) is programmed at configuration time, and allows the FF to receive data from the LUT or from another input, called DIN. Both combinational (X) and registered (XQ) outputs are provided.

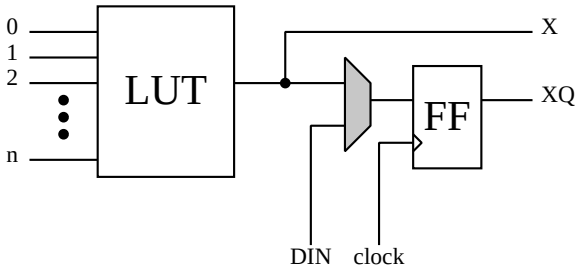

Figure 2.3: Model for a Look-up Table based configurable cell

CLUs are connected to each other by a programmable interconnect as shown in Figure 2.4 The flexible interconnect system allows a high degree of freedom for

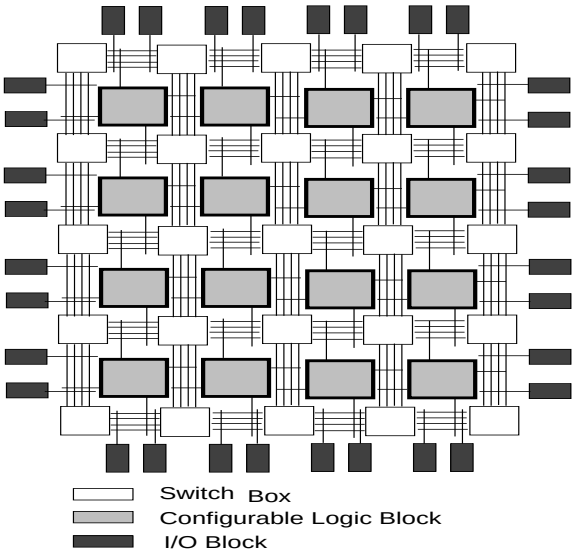

Figure 2.4: FPGA structure

placement and routing of circuits but also causes more delay than interconnects used in ASIC technology. The same way as ASICs, the communication line delay is sensitive to the load on the line and the length of it.

Regarding support for arithmetic operations, some FPGAs have special circuitry to reduce the time to propagate carries/borrows in adders and subtractors. This

fast carry logic (FCL) makes possible the implementation of fast and area efficient adders for small number of bits. It expands the functionality of the FG using an extra input and output (called CIN and $COUT$). These input/output pairs are connected by a dedicated network, faster than the one that interconnects CLUs. For this reason the FCL reduces the total addition time significantly.

The structure of a Configurable Logic Block (CLB) for the Xilinx XC4000 series is presented in Figure 2.5.

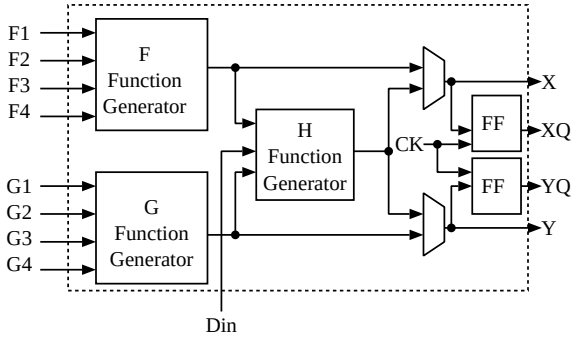

Figure 2.5: Structure of a CLB in the Xilinx XC4000 series

2.1.2 FPGA Array

If many FPGA chips are used, the interconnection between them will significantly impact the overall performance of the VLP algorithms. Flexible interconnects between the chips imply lower bandwidth and larger delay. For the VLP algorithms proposed in this thesis, its is important to have high data bandwidth, so a dedicated interconnect is assumed for highly demanded data and bus connections are considered for lower demanded data. After we explain the VLP algorithms these options will become clear.

The most adequate interconnection between FPGAs is the linear array, as shown in Figure 2.6. The local interconnections have small delays and allow pipelining between circuits working in neighboring chips.

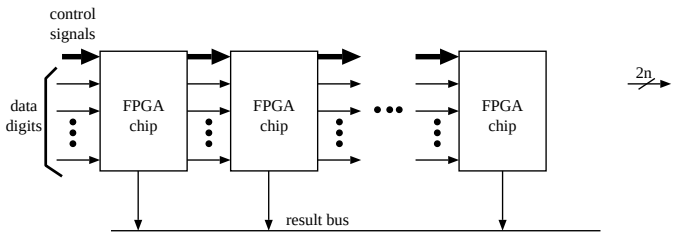

Figure 2.6: Linear Array of FPGAs

The communication across chips may take longer than the operation cycle. A strategy to balance the chip I/O bandwidth and the circuit speed was discussed in [Lou94] and may also be used here. This work will concentrate on single chip implementation, but also provides a preliminary discussion on the implementation of the VLP algorithms over multiple chips.

2.2 Software and Hardware Interface

We propose to make the new VLP arithmetic algorithms available to the VLP application or library in the form of a hardware object. The object hides the hardware implementation details from the user. The VLP application or library makes use of the coprocessor operation through a procedure call. There is a technology already available, named Hardware Object Technology (HOT) that uses this concept [Cor95b].

The VLP algorithm implemented in the coprocessor requires that the host executes some tasks that do not benefit from hardware implementation, as for example: digit adjustments, number conversion and exponent manipulation. Thus, only the kernels of the VLP operations are implemented in the coprocessor.

In this work we selected the Gnu Multi-precision Library (GMP) as the software package to exemplify the application of the coprocessor and also compare performance. The application software uses data types and VLP operations defined in the library. When using the coprocessor, the VLP operation is replaced by

a procedure, the the same interface, that is able to manipulate the reconfigurable hardware.

An example of VLP addition using GMP library with and without the coprocessor is shown in Figure 2.7. The procedure named *mpz_add* receives three pointers: one for the result (*sum*) and two for the operands (*int*1 and *int*2). The path on top of the figure shows the application program activating the procedure, that is executed by the software library and returns the control to the application. When the coprocessor is used, the activation of the procedure passes control to a software routine that interfaces the coprocessor hardware, doing basic operations such as: transferring data to the coprocessor's local memory, starting the operation in the coprocessor and reading the results. Using this procedure, the application program is not aware if the operation was performed by a GMP original routines or by the VLP coprocessor.

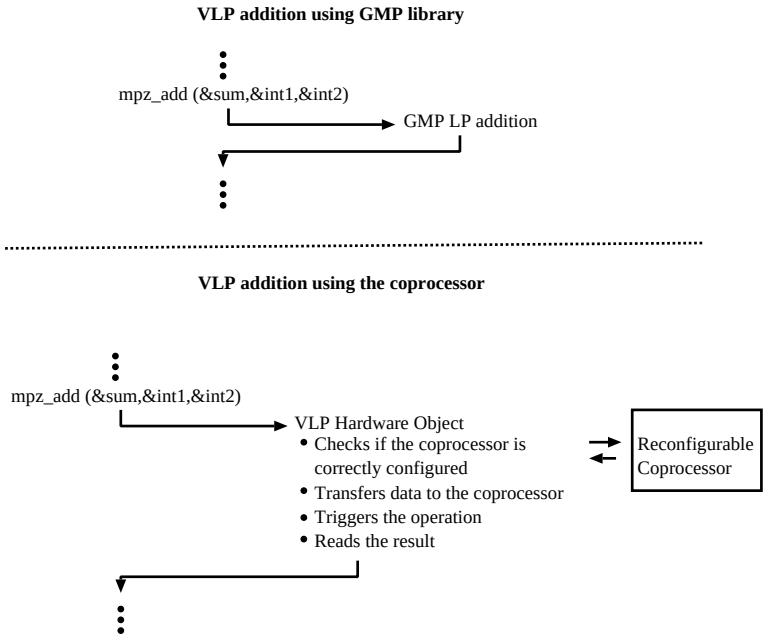

Figure 2.7: Software/Hardware interface for LP addition

Chapter 3

Variable Long-precision Arithmetic (VLPA) Algorithms

In this chapter we present commom algorithms used in software solutions for long-precision computation, review the hardware algorithms used in present coprocessors for long-precision calculations and introduce the concept of on-line arithmetic as a solution for VLP operations. The presentation of software algorithms provide the main concepts involved in long-precision calculations. We also justify the main reasons that make some of the algorithms preferable for hardware implementation.

By definition, one cannot have direct (full) hardware implementation for VLP arithmetic operations. The objective in any case is to find an optimal partitioning and allocation strategy for the harware resources available. The problem is involved since the hardware resources, although limited, are reconfigurable. The circuit design for FPGAs is done depending on the problem at hand and the resources available.

VLP operations are done serially, digit by digit. The number of bits in each digit is adjusted according to system characteristics. When using a software library to perform this task in a general purpose processor, the digit corresponds to the machine word. Using RAs one can adjust the digit size in order to get the most performance in the available area. Area utilization is not a serious problem for serial addition, since serial addition units use small chip area and the area depends only on the digit size, not the the precision of input operands or result. The problem in the use of limited precision units is exposed in other operations, for which the digits already received by the unit are important for the computation of the next output digits. Thus, these serial operators for long-precision computation

need hardware resources proportional to the precision of the result. This is really a problem of the internal state. Some operations, like addition, have a constant-size state while the others, like multiplication, have a state size proportional to the operands' precision.

Solutions to this problem are usualy inspired in algorithms for long-precision operations implemented in software.

3.1 VLP Algorithms used in Software

In this section we present algorithms used in software packages for variable long-precision computations. Sometimes these packages or libraries use the term *multiple precision* to indicate the fact that the operation's precision varies in multiples of a basic precision allowed by the processor.

By analyzing GMP [Tor93], Numerical Recipes routines [P+94] and MPFUN software packages [Bai93], we could determine the algorithms used for the four basic operations. The main aspects of most of the algorithms are also presented in detail in [Knu69, P+94]. They are used in the software packages mentioned in this work, with some modifications.

3.1.1 Notation and Conventions

Consider that each word in the machine holds one digit in a very high radix (r). A long precision number is represented as a sequence of words (digits). Usually the value of the radix is a function of the machine word length in bits n, i.e. $r = 2^n$, such that each word represents a digit, u_i.

Only positive numbers are considered based on the fact that most libraries use sign and magnitude representation, where the sign of the number can be treated separately. The magnitude value represented by a vector of digits $(u_1 u_2 u_3 ... u_m)$ is obtained in the conventional positional notation as:

$$u = \sum_{i=1}^{m} u_i r^{m-i}, \text{ with } 0 \leq u_i < r$$

The number of digits (m) in the vector is variable and it represents the *precision* of the number in radix r.

The long-precision operations are mapped into *primitive operations* (instructions) that are available in the computer:

- addition/subtraction of one-digit integers (radix r) giving a one-digit integer and a carry as a result;
- multiplication of a one-digit integer by a one-digit integer, giving a two-digit integer as a result.

Although for some operations the discussion is focused on integers, the application of the VLP algorithms to floating-point numbers is straightforward.

3.1.2 Software Algorithms for VLP Addition

The basic algorithm used for addition of extended precision numbers is the serial addition. Each step is performed in a machine word that stores a digit in base $r = 2^n$.

Addition and subtraction in these "classical algorithms" always start from the least-significant digit. The high-precision numbers are stored in memory in an array of memory words.

The software routine monitors the carry out in each digit addition and uses it as carry in for the next digit addition. The process stops as soon as the precision of one of the operands is exhausted and there is no carry out to be assimilated in next steps.

3.1.3 Software Algorithms for VLP Multiplication

VLP multiplication is performed by different algorithms depending on the precision of the operands. Simple algorithms are faster than more elaborated ones when the precision is around tens of digits (few hundred to a thousand bits), that

we call in this section as "low precision". After a limit that is defined experimentally for each software package implementation, the precision is considered "high precision", and a more elaborated algorithm (with better asymptotic time) is used.

The threshold between "low" and "high" precision was evaluated in [Com90] to be around 30 words (approximately 900 bits). If the elaborate algorithm is applied for operands with less than 30 words, the overhead involved in the data manipulation makes it worse than a simple algorithm. In the GMP library, the threshold has a similar value.

3.1.3.1 VLP Multiplication of Low Precision Numbers

For the multiplication of small precision numbers, the simplest algorithm for multiplication, equivalent to the classical "paper and pencil algorithm", is more efficient than other more sophisticated methods.

To multiply two operands $u = u_1u_2...u_p$ and $v = v_1v_2...v_m$, without loss of precision, the result $w = u.v = w_1w_2...w_{m+p}$ must have $m + p$ digits. The value of the operands and result are:

$$\begin{aligned} u &= \sum_{i=1}^{p} u_i r^{p-i} \\ v &= \sum_{i=1}^{m} v_i r^{m-i} \\ w &= \sum_{i=1}^{m+p} w_i r^{m+p-i} \end{aligned}$$

Algorithm 1 was adapted from [Knu69]. The algorithm executes $m.p$ digit multiplications and $mp - 1$ double precision additions. The number of single precision (digit) additions depends on the generation of carries in the addition of the first (least significant) words. If a carry is not generated, there is no need to continue with the second word. Some savings result from this simple test. The inclusion of a test for a zero condition of v_j is not worthwhile since the probability of having $v_j = 0$, with a high value of r, is very small. The asymptotic time of this is algorithm is $O(n^2)$.

Algorithm 1 *: low-precision-multiplication (classical)*

S1: *set* $w_m w_{m+1} ... w_{m+p} = 0$. *Initialize the outer loop counter* $j = m$.

S2: *initialize the inner loop counter* $i = p$ *and set variable* $carry = 0$.

S3: $w_j = (u_i.v_j + w_{i+j} + carry) \bmod r$ *and* $carry = \lfloor \frac{u_i.v_j + w_{i+j} + carry}{r} \rfloor$

where $0 \leq carry < r$ *is the carry-out digit at position* j.

Notice that $u_i.v_j$ *generates a double precision number, and thus, all additions are double precision.*

S4: $i = i - 1$. *If* $i > 0$ *go to S3. Otherwise* $w_j = carry$ *and continue to S5.*

S5: $j = j - 1$. *If* $j > 0$ *go to S2, otherwise stop.*

3.1.3.2 Multiplication of High-Precision numbers

One can calculate the product of two operands by breaking them in half and combining the result of the multiplication of these halves. An algorithm called Ofman-Karatsuba [Knu69] reduces the number of digit multiplications used in the classical algorithm. This scheme is used by GNU MP library when the number of digits is larger than a certain threshold value. The algorithm uses the following equation to obtain the product of U and V:

$$U.V = (r^m + r^{m/2})U_1V_1 + r^{m/2}(U_1 - U_0)(V_0 - V_1) + (r^{m/2} + 1)U_0V_0$$

Using this transformation, only 3 multiplications with half of the precision are needed. The number of additions/subtractions is: 2 subtractions in precision $m/2$, and 4 additions in precision m, besides some shift operations.

Assuming that both operands have the same precision, and it is a power of 2, the algorithm to multiply high-precision numbers is defined recursively as shown in Algorithm 2. Operands and result are: $U = u_1u_2...u_m$, $V = v_1v_2...v_m$ and $P = p_1p_2...p_{2m}$, with $m = 2^k$. The asymptotic time of this algorithm is $O(n^{1.6})$.

3.1.3.3 FFT-based VLP Algorithm

Another way to multiply high-precision numbers in software uses the FFT process, or Schönhage and Strassen algorithm [Knu69, Bai93]. The algorithm has a better asymptotic time of O(nlog(n)log(log(n))).

Examining the multiplication procedure, we can see that the product of two vectors of digits is the convolution of these vectors. That is, the multiplication can be represented by the expression:

$$p_n = \sum_{k=0}^{n} x_{n-k}y_k = x_n * y_n$$

where p_n is an element of the product vector and the symbol $*$ is the convolution operation. For example, consider the multiplication of two vectors X=(4,5,6) and Y=(7,8,9) [P+94]:

Algorithm 2 *: high-precision-multiplication (Ofman-Karatsuba)*

S1: *if* $(size < threshold)$

use classical multiplication algorithm and stop. Otherwise continue.

the stop in this step may represent the return from a recursive call.

S2: *split the operands in two halves* V_1, V_0, U_1 *and* U_0

where the vectors with indices 1 are the most-significant halves.

S3: *recursively call this algorithm to multiply*

$pp1 = U1 \times V1$,

$pp3 = U0 \times V0$, *and*

$pp2 = (U1 - U0) \times (V0 - V1)$

S4: *combine the partial products obtained in S3, according to the expression:*

$p = (r^{2c} + r^c) \times pp1 + r^c \times pp2 + (r^c + 1) \times pp3$

where $c = \lceil size/2 \rceil$

stop.

		4	5	6
	x	7	8	9
		36	45	54
	32	40	48	
28	35	42		
28	67	118	93	54

that by reduction to digits in radix 10 results in the vector $(3, 5, 9, 7, 8, 4)$.

The convolution theorem says that the Fourier transform of the convolution of two functions is equals to the product of their individual Fourier transforms. This theorem is true for the continuous and discrete (sampled) cases. Thus, antitransforming the product of the Fourier transforms of the two operand vectors, it is possible to get the convolution of these vectors. If operands are g and h, with m digits, and their Fourier transforms are $G(f)$ and $H(f)$, respectively, the theorem says that:

$$g * h \Longleftrightarrow G(f) \cdot H(f)$$

where the symbol $\Longleftrightarrow$ denotes the transformation or antitransformation between the two systems. Double precision numbers are necessary to store the accumulation of digits in the convolution to avoid overflow. Notice that this is a limitation for the number of digits that can be used in the multiplication.

To transform a vector with m digits, $mlog_2m$ steps are necessary.

The multiplication of the transformed vectors (imaginary numbers), requires $4m$ real multiplications and $2m$ additions. A better solution would be to use only $3m$ multiplications, as follows:

$$(a + bi)(c + di) = ac + (ad + bc)i - bd = (ac - bd) + i[(a + b)(c + d) - ac - bd]$$

Notice that the multiplication of two discrete functions $G(f)$ and $H(f)$ is obtained by multiplying the values of the functions for each possible value of the input.

After getting the transform of the convolution, the antitransform process needs $mlog_2m$ steps to obtain the value of the result. The obtained vector does not have

values that satisfy the values of the digits, thus, a final pass to adjust digit values and propagate the carries if necessary.

Some problems for hardware implementation are:

- use the values of *sin* in the process. These values are usually rounded, and stored in tables in the hardware.
- even though the number of steps was reduced to $mlog_2m$, the procedure is time consuming since each step is complex. A high level of parallelism must be used to make the procedure attractive.

This algorithm is used by MPFUN (a FORTRAN multiprecision package), a software library implemented in CRAY machines. The convolution process is adequate for vector computations. Values for the execution time of this algorithm in CRAY machines are presented in [Bai93].

3.1.4 Software Algorithms for VLP Division

The most used algorithm for long-precision division is the Newton-Raphson algorithm [P+94]. It makes use of one VLP multiplication algorithm. To obtain the quotient X/Y compute the reciprocal of the divisor by iteration of the Newton's rule:

$$U_{i+1} = U_i(2 - YU_i)$$

where U_0 is an approximation of $1/Y$, and the algorithm has a quadractic convergence to the correct value of $1/Y$. The flowchart of the complete algorithm is shown in Figure 3.1. The asymptotic execution time is $O(MloglogN)$, where M is the asymptotic time of the long-precision multiplication algorithm and $loglogN$ is the factor related to the convergence of the Newton's rule.

An efficiente implementation of VLP multiplication is critical for the overall performance of this division method. Many variable long-precision multiplications are executed and at least two of them using full precision of the operands.

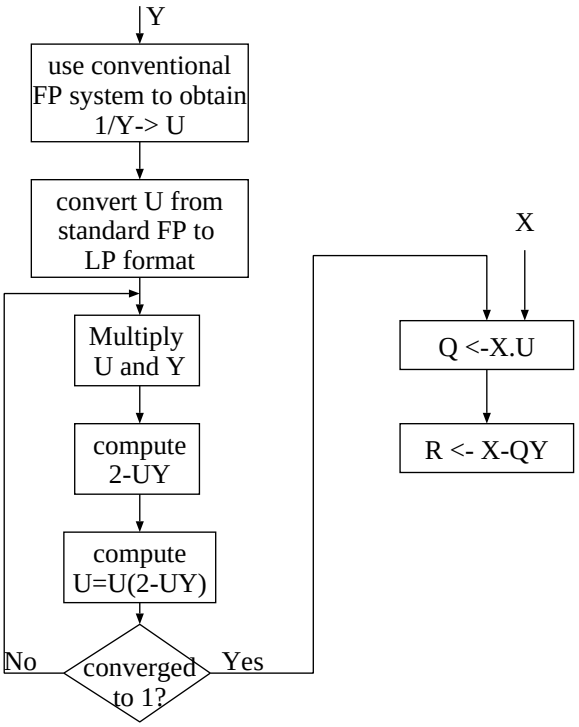

Figure 3.1: Flowchart of Newton-Raphson algorithm for division

3.1.5 Software Algorithms for VLP Square-Root

The Newton-Raphson square root method first computes the reciprocal of the square root and them multiply the result by the initial operand. The reciprocal of the square root of a long-precision number V is obtained iteratively using the recurrence equation:

$$U_{i+1} = \frac{1}{2}U_i(3 - VU_i^2) \tag{3.1}$$

where U_0 is an approximation of $\frac{1}{\sqrt{V}}$ and U_∞ converges quadraticaly to the full precision reciprocal value. Intermediate steps are done in precision that doubles in each iteration of the algorithm. A final full-precision multiplication by V is done to obtain the correct result. The initial approximation of $\frac{1}{\sqrt{V}}$ is obtained by table lookup or using an available square root instruction in the processor. The asymptotic time for the algorithm is the same as the division, using the same

method. The flowchart of the square-root computation using Newton-Raphson method is shown in figure 3.2.

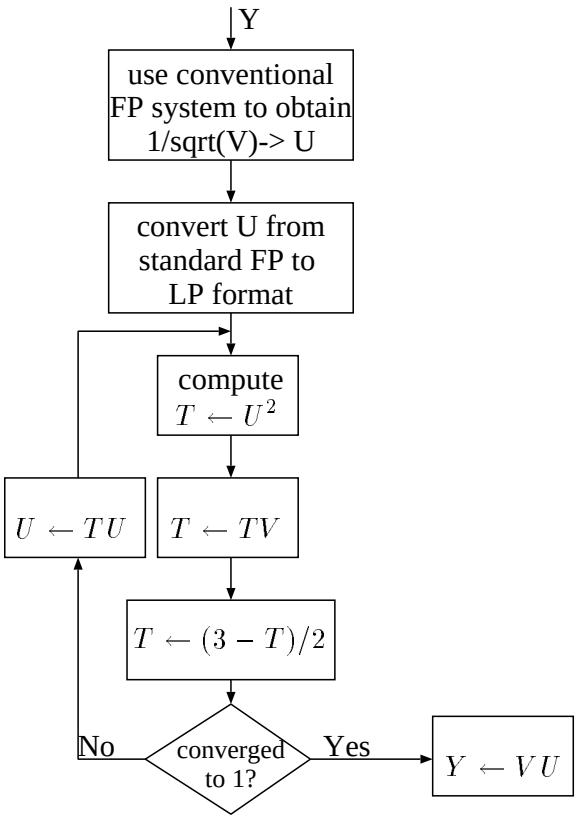

Figure 3.2: Flowchart of square-root computation using Newton-Raphson method

3.2 Hardware Implementation of VLP Algorithms

We discuss in this section the algorithms used in available coprocessors.

3.2.1 Multiplication in VLP Coprocessors

VLP arithmetic coprocessors use the classical algorithm for multiplication for long-precision numbers. Although the classical algorithm has a time complexity of $O(n^2)$, that is the worst of all software algorithms, the choice is justified by a

simpler control and data transformation circuits. The justification for using this type of algorithm and not another one with a better asymptotic time is given in [Zur93]. The approach was used in the design of the arithmetic unit of the VPIAC [SSJ95b, SSJ95a] and the VLPA coprocessor designed for the Transmogrifier-1 (TM-1) [Hsu96]. More details in the implementation of the arithmetic circuits of these machines are shown next.

The VLP multiplication is done in two different ways: computing/ accumulating long-precision partial products (by rows) [SSJ95b, Sch96] or computing a particular digit of the result (by columns) [Hsu96].

In the VPIAC [SSJ95b, SSJ95a] the multiplication algorithm used is a variation of the classical algorithm used in software. A $n \times n$ bit multiplier (digit of n bits) generates the digit multiples that are shifted and added to previously accumulated products using an internal long-accumulator. Figure 3.3 illustrates this algorithm. The words of the multiplicand are accessed from the least to the most significant digit. Each digit of B is multiplied by one digit of A. If the number of digits in the multiplicand is even (as shown in the figure) the digits of A with even indexes are multiplied first (i.e., A_4B_4 and A_2B_4). If the number of digits in the multiplicand is odd, odd indexes are used first. The use of a long accumulator increases performance and reduces data transfers at the cost of more area, however, this feature also limits the maximum precision to the length of the accumulator. The long-precision multiplication in this processor takes m^2+m+12 cycles, where m is the number of operand digits. The duration of each cycle depends on the precision of the digit size. The cycle times for a 64-bit, 32-bit, and 16-bit units are 22ns, 16ns and 12.5ns, respectively.

The design presented in [Hsu96] uses a different approach. It keeps inside the chip the partial accumulation of the digits in each column of the partial product matrix. The precision of the required adder is $log_2(\lceil m/n \rceil) + 2n$, representing two n-bit digits and some extra space for carries. The partial products of column i are generated by multiplying the words x_{i-k} and y_k, for k from 0 to i, and $0 \leq i < m$.

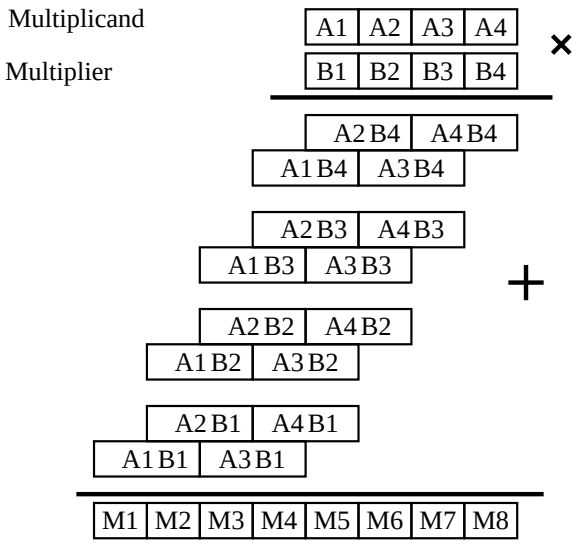

Figure 3.3: Multiplication method in the VPIAC

When $m \leq i < 2m$, then k varies from $i-(m-1)$ to $(m-1)$. Different words of the operands are read in every step, opposite to what is done in the VPIAC. The performance of this algorithm is given as a function of the memory accesses to memory: $2m^2 + 2m$ cycles, with a cycle time of 130ns.

3.2.2 Division in VLP Coprocessors

Hardware long-precision division is usually performed by modifications of the Newton-Raphson method [SSJ95b, SSJ95a] or digit-recurrence division algorithm [Hsu96]. The digit-recurrence division algorithm [EL94] generates each digit of the quotient based on the scaled residual:

$$w[j+1] = rw[j] - q_{j+1}d \tag{3.2}$$

where x is the dividend, d is the divisor, w is the residual and q is the quotient digit. Initially $w[0] = x$. The quotient digit is obtained by a selection function, which is more complex for higher radices. Pre-scaling of the operands simplifies

the selection function but adds extra steps to obtain the multiplication constant and scale the operands.

3.2.3 Square-root in VLP Coprocessors

The coprocessor proposed in [SSJ95b, SSJ95a] computes the square-root using an algorithm similar to Newton-Raphson, described in section 3.1.5. The square-root operation is not considered in [Hsu96].

3.3 On-line Algorithms for VLP Computations

At the conceptual level all VLP algorithms manipulate a processor word as a high-radix digit. The operations are carried digit by digit, serially. On-line algorithms [Erc84, TE77] are the only ones that allow serial computation of all arithmetic operations.

The main reasons for using on-line algorithms for VLP arithmetic are:

1. Inputs and outputs are handled *serially*, most-significant digit first allowing variable-precision operation and overlap between successive operations.

2. On-line algorithms work with fixed point numbers. When the algorithm is used to compute the significand part of floating-point product only the most significant digits of the product are computed without wasting cycles on the least significant digits.

3. Circuits for on-line operations have regular structure, which can be easily extended, and a simple algorithm step.

We first review the concepts of on-line arithmetic, and later we present and discuss the hardware organization and VLP algorithms based on on-line arithmetic.

3.3.1 General Concepts and Scheduling Strategies

The basic ideas and algorithms of on-line arithmetic are presented in [Erc84, TE77] and a design methodology in [EL88]. As mentioned above, the result digits are produced serially, most significant digit first, after a few cycles (on-line delay δ). On-line multiplication ($Z = XY$), division ($Z = X/Y$) and square root ($Z = \sqrt{X}$) are defined by the recurrence equations shown in Table 3.1, with the following convention for the digit vectors:

$$\begin{aligned} X[j] &= \sum_{i=0}^{j+\delta-1} x_i r^{-i} = X[j-1] + x_{j+\delta-1} r^{j+\delta-1}, \\ Y[j] &= \sum_{i=0}^{j+\delta-1} y_i r^{-i} = Y[j-1] + y_{j+\delta-1} r^{j+\delta-1} \\ W[j] &= \sum_{i=0}^{j} w_i r^{-i} \text{ and} \\ Z[j] &= \sum_{i=0}^{j} z_i r^{-i} \end{aligned}$$

where W is the *scaled residual* and any digit is in the set $\{-a, ..., -1, 0, 1, ..., a\}$, $r/2 \leq a \leq (r-1)$.

The on-line delays in each expression are slightly different and depend on the radix used. The scaled residual is kept inside bounds by subtracting the output digit z_j that is selected based on a *selection function* that considers the value of the present residual and other parameters. The input operands are usually in the range $[\frac{1}{r}, 1)$ but in some special cases, the operands are scaled to be in a different range to simplify the algorithm implementation. These cases are presented in Chapters 5 and 6 (VLP division and square root).

The theory of on-line operations was developed for any radix. However, the implementation of on-line operators is usually proposed for small radices (2 and 4) [SG93, TE87, TE89, TE86]. The evaluation of an on-line multiply-add operation using high radix digits is done in [TE96]. The work proposes structures that can be easily adjusted to different precision and shows the best cost/performance

On-line multiplication
$W[j] = r(W[j-1] - z_{j-1}) + r^{-\delta+1}(x_{j+\delta-1}Y[j] + y_{j+\delta-1}X[j-1])$
$W[0] = X[0]Y[0]$

On-line division
$W[j] = r(W[j-1] - z_{j-1}Y[j-1]) + x_{j+\delta-1}r^{-\delta+1} - y_{j+\delta-1}Z[j-1]r^{-\delta+1}$
$W[0] = X[0]$

On-line square root
$W[j] = r(W[j-1] - z_{j-1}(z[j-1] + z[j-2])) + x_{j+\delta-1}r^{-\delta+1}$
$W[0] = X[0]$

Table 3.1: Recurrence equations for on-line arithmetic

relation for the design of the on-line multiply-add operation. The use of high-radix structures always leads to better performance over small radix. For this reason we concentrate the study on high-radix on-line algorithms and circuits.

Unlike the software algorithms, on-line operators generate results in redundant form that must be converted to conventional representation. This may be considered as a drawback of the method at first. However, the concurrent execution of an on-the-fly conversion algorithm (similar to the algorithm presented in [EL87]) can minimize the performance impact of the conversion step. The conversion is discussed in Section 7.2.2.

The recurrence equations in Table 3.1 indicate the use of two digit by vector multiplications and long-precision additions. The vectors obtained by the digit by vector multiplications (such as $x_{j+\delta}Y[j]$ and $y_{j+\delta}X[j-1]$) must be properly aligned (by shifting the digits by a fixed amount) before the long precision addition takes

place. Variables such as $Y[j]$ and $X[j-1]$ represent vectors that increase in length as new digits are received. Conventional on-line operators use Append Registers to store these vectors. We propose later another method to handle incoming digits in VLP operations.

The discussion that follows presents two alternatives to implement on-line VLP operations which are applicable to all on-line operations. The first technique considers the use of a digit slice to obtain the long-precision result. The second technique consists of computing the on-line recurrence equation using serial modules.

3.3.1.1 The use of digit-slices

Figure 3.4 shows the space complexity of the recurrence equation computation. The parallelograms represent the digit by vector computation and the rectangles represent the long-precision additions.

When there is not enough hardware to implement the full precision of the operands, only a *section* of the total space is implemented in hardware (d radix 2^n digit slices). The input operands are received serially, digit by digit. While the input precision is less than d, the section is used once, for input precision in the range $[d, 2d)$ the section is used twice, and so on. Registers are placed in the circuit in order to store the intermediate bits (carries) that are generated from one activation of the section to the next.

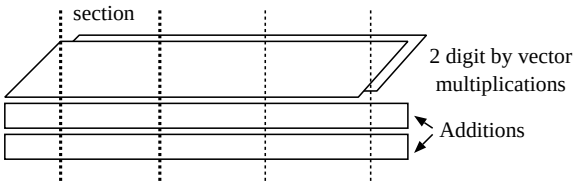

Figure 3.4: Spatial representation of on-line recurrence equation computation

To illustrate the approach, consider the case of $n = 3$, radix 8 digits, as shown

in Figure 3.5. Vector A represents the product $y_{j+3}X$ and B represents $x_{j+3}Y$. W represents $W[j-1]$ and W' represents the $W[j]$. C_i is the carry out digit of each section. A most significant digit first (MSDF) accumulation of the residual is done in this case.

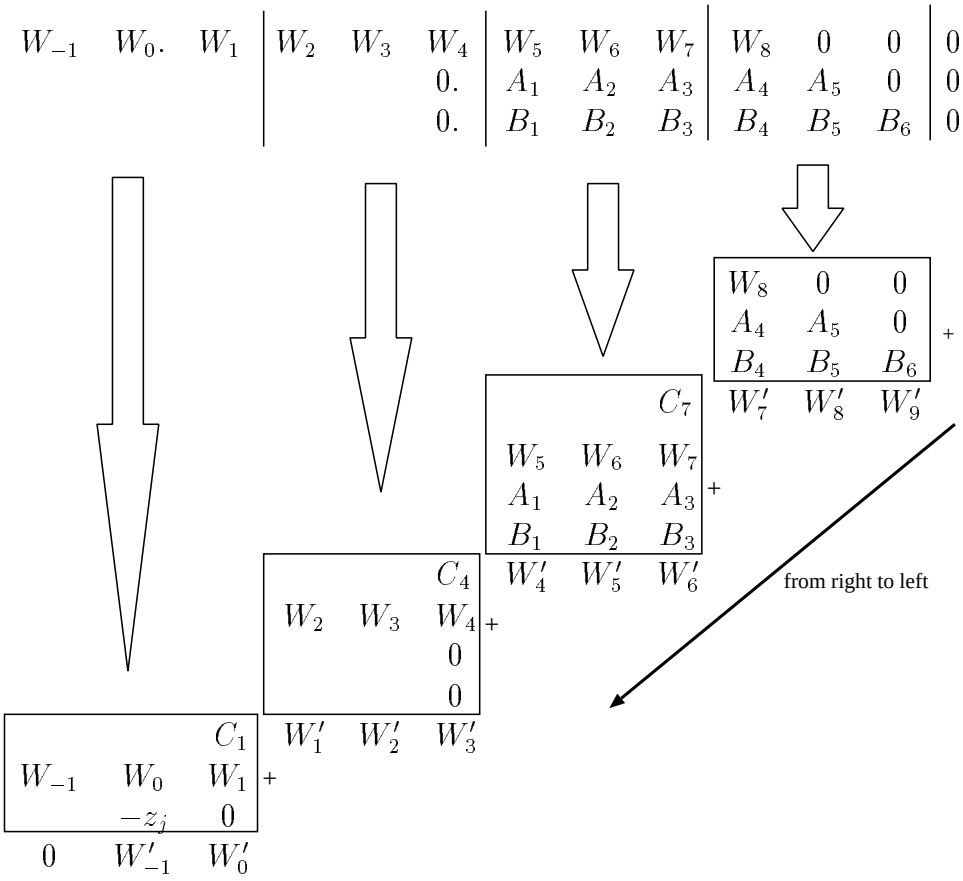

Figure 3.5: Multiple-precision computation of the recurrence equation

A general structure of the on-line multiplication slice is shown in Figure 3.6. Thicker lines represent paths more than one digit wide. A group of d digits of X, Y or W are read at once and applied to the group of digit-slices (section), *before* one computation cycle begins. Registers (dark boxes) are used to store the carries from one iteration to the next. Sections of multiplers and adders are represented as a box with a marked corner. These components are not complete functional

operators, they are part of larger operators.

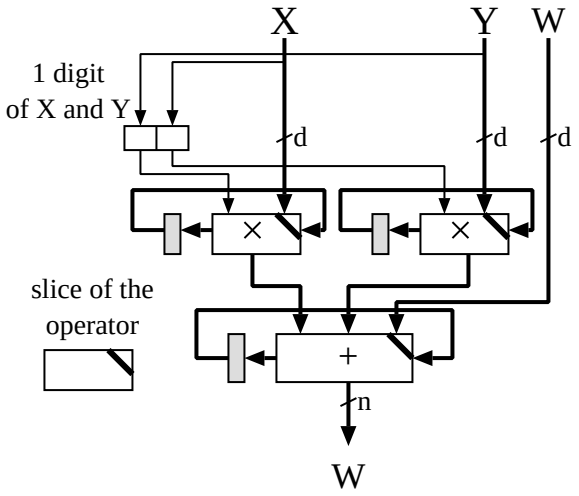

Figure 3.6: Digit-slices for VLP multiplication

3.3.1.2 Serial Computation of Recurrence Equation

This technique considers digit operators in high-radix to compute the on-line recurrence equation serially, instead of using digit slices. Both conventional or on-line serial modules can be used, and the residual can be computed in LSDF or MSDF mode.

The output digit (z_{j-1}) used in the recurrence equation is selected from the most significant digits of the residual. The LSDF mode of operation will force the output digit selection to the end of the serial computation of the residual. However, the iteration to compute the next residual cannot begin until the output digit is selected (based on the present residual), and this dependency will slow down the computation.

The MSDF mode of operation uses on-line units. The most significant digits of the residual are computed first, leaving more time to select the output digit before the next iteration is about to begin.

By construction, all recurrence equations created for on-line computations use digit by word multiplication, addition/subtraction and shifting. The digit by word multiplier has a fixed area which depends only on the digit size. The time to obtain the result depends on the precision of the vector containing the previously received digits, which can be as large as needed. The complexity of the circuit is adjusted based on the radix of the digit being used. Addition and subtraction also have an area dependent only on the radix of the digits being added.

The network of serial modules for the particular case of the recurrence equation used in multiplication is presented in Figure 3.7. The multiplication nodes are serial digit by vector multiplier modules. As shown in the figure, the multiplication nodes are composed by a digit by digit multiplier, a delay block (shaded box) and a serial adder (in the figure, an on-line adder). One of the inputs is kept with a fixed value during one recurrence equation iteration (digits $x_{j+\delta-1}$ and $y_{j+\delta-1}$). The digits stored in the digit vectors $X[j]$ and $Y[j]$, each digit represented in the figure as $X_i[j]$ and $Y_i[j]$, are fed serially into the multipliers. The same notation is used for the scaled residual. The output digit z_j is used in only one cycle to correct the residual. More details of this circuit are given in Chapter 4.

This strategy has the following advantages over the previous one:

- The network that implements the recurrence equation is composed of standard modules: serial-parallel multipliers, adders/subtractors, and delay blocks, and not by slices of operators.

- The basic blocks that constitute the data path to compute recurrence equation can be easily made high-radix. This increases the speed of this type of circuit to a maximum allowed by the communication bandwidth between the data path and the memory elements. For digit slices the implementation of high-radix can be very complex.

- When using on-line modules in the data path, the selection of the output digit (selection function) is going to be in the critical path only at the beginning

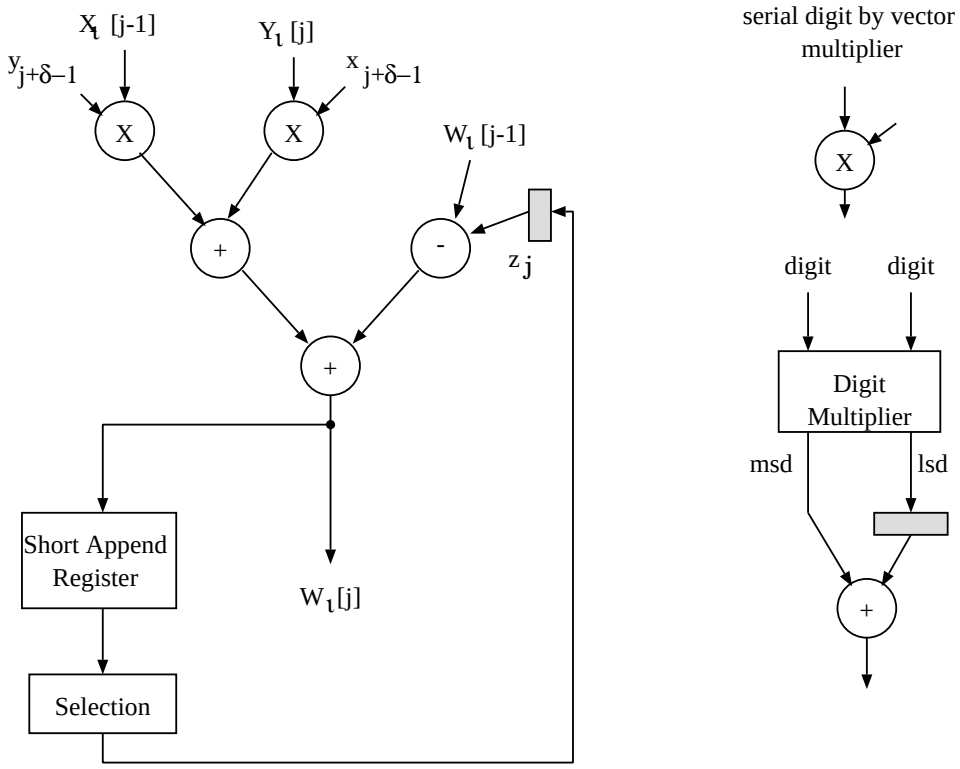

Figure 3.7: On-line computation of the recurrence equation for multiplication

of the operation, when the precision of the residual is small. There will be enough time to compute the output digit while the network is computing other digits of the recurrence equation.

The on-line algorithm using these scheduling strategies will have an asymptotic time complexity of $O(m^2)$ steps, where m is the number of operand digits.

3.4 Summary

In this chapter we discussed the software and hardware algorithms commonly used for long-precision computations. There are software algorithms that have a better asymptotic time than the algorithms used in hardware for long-precision. The hardware algorithms for VLP arithmetic have been based on conventional

and simple software algorithms to avoid the complexity in the implementation. We have also discussed the choice of on-line arithmetic for our VLP algorithms.

The number of cycles for the available hardware implementations are summarized in the following table. The equations shows that the asymptotic time of the VLP operations is $O(n^2)$, the same asymptotic time of the on-line algorithms.

Operation	VPIAC [SSJ95a]	TM-1 [Hsu96]
Multiplication	$n^2 + n + 12$	$7n^2 + 5n + 20$
Division	$3n^2 + 4n + 20$	$6n^2 + 21n + 17$
Square Root	$3n^2 + 6n + 26$	not implemented

However, the on-line algorithms have important features that allow the implementation of very efficient designs for long-precision computation. We explore the use of serial modules to compute the on-line recurrence equations based on the discussion presented in the previous section. VLP operation using on-line will allow the overlap of operations between the host and the coprocessor increasing the performance of the host+coprocessor architecture. When more than one coprocessor is used, ovelap of VLP operations executed in different processors are also possible. The algorithms for VLP multiplication, division and square root are described in the next chapters.

Addition and subtraction are not considered for implementation in the coprocessor for the following reasons:

1. The time complexity of addition and subtraction is linear. The gain in using the hardware for these operations over the software implementation (at the host level) is minimal. The algorithm is not complex and the host is very efficient to perform addition of integers.

2. The on-line approach for addition has no advantage over the traditional right to left addition, with carry propagation. The on-line generates the digits in redundant format, what requires a Carry Propagate addition by the end to convert to conventional number system.

Chapter 4

VLP Multiplier

This chapter presents the design of a VLP multiplier that uses the on-line arithmetic approach [Erc84, TE77]. The use of on-line arithmetic method to implement VLP operations was alredy justified in the previous chapter.

We first present the VLP multiplication algorithm considering the operation over long-precision integers, where the product has twice the precision of the operands. Then, we discuss the case of the multiplication of significands of floating-point numbers, where the output and operands' precision is the same. Main issues of the implementation of this particular algorithm are presented and discussed.

4.1 The VLP Multiplication Algorithm

On-line multiplication $Z = XY$ is defined by the following recurrence equation [EL88]:

$$W[j] = r(W[j-1] - z_{j-1}) + r^{-\delta+1}(x_{j+\delta-1}Y[j] + y_{j+\delta-1}X[j-1]), j \geq 1 \quad (4.1)$$

with

$$X[j] = \sum_{i=0}^{j+\delta-1} x_i r^{-i} \text{ and } Y[j] = \sum_{i=0}^{j+\delta-1} y_i r^{-i} \quad (4.2)$$

where x_i, y_i and $z_i \in \{-a, ..., -1, 0, 1, ..., a\}$ with $r/2 \leq\ a \leq\ (r-1)$.

The product $Z[j] = \sum_{i=1}^{j} z_i r^{-i}$ is obtained in redundant form. The variable W is called the *partial residual*. The initial condition for the algorithm is $W[0] = X[0]Y[0]$. Partial products are added in each iteration to the scaled residual rW, that is kept inside bounds by subtracting the output digit z_j. The on-line operands $Y[j]$ and $X[j-1]$ increase in precision as new digits are received.

Algorithm 3 corresponds to the VLP multiplication algorithm. The operands' precision in digits is represented by m. Each digit d_i is in the digit set $D_\rho = \{-\rho, \cdots, \rho\}$ with $\rho = r-1$ and $r = 2^n$. The on-line delay assumed in the algorithm is 3 cycles. We use output digit selection by truncation [TE96]. This selection function generates over-redundant output digits that must be recoded in order to generate output digits in the set D_ρ. The algorithm generates the product of $2m$ digits (double-precision).

The *RECODE* function converts the output stream formed by digits $z_j^* \in \{-(2r-1), ..., (2r-1)\}$ into a stream formed by $z_j \in \{-\rho, ..., \rho\}$. The analysis of the digits that form z_j^* shows that $W_0[j] \in \{-1, 0, 1\}$ and $W_1[j] \in \{-\rho, ..., \rho\}$. The sequence of over-redundant output digits z_j^* can be seen as two separate sequences:

$$(W_0[0], W_0[1], ..., W_0[m-2], 0)$$

and

$$(0, W_1[0], W_1[1], ..., W_1[m-2])$$

An on-line adder reduces these two sequences to one that corresponds to the vector of recoded values z_j. The recoder circuit is described in Chapter 7.

The first $m-2$ digits of the product are obtained during the recurrence iterations. The remaining least significant digits are in the residual vector, such that the final result is represented as:

$$Z = (z_1, ..., z_{m-2}, W_1[m-2], ..., W_{m+2}[m-2])$$

A variation of on-the-fly conversion method [EL87] is used to transform the output digit set from redundant to non-redundant (NR) representation. Further details are discussed in Section 7.2.2.

4.1.1 Data Path for VLP Multiplication

As discussed in chapter 3 the data path for VLP algorithms is implemented as a network of serial modules capable of performing: digit by vector multiplication,

Algorithm 3 *:VLP Multiplication Algorithm*

1. Initialization:

a) input digits x_1, x_2, y_1 *and* y_2 *and compute the initial residual:*

$W[0] = \sum_{i=1}^{2} \sum_{j=1}^{2} x_i y_j r^{-(i+j)}$

b) initialize the product:

$z_0^* = rW_0[0] + W_1[0]$

$z_{-1}^* = 0$

where W_0 *and* W_1 *represent the most significant digits of the residual (one integer and one fractional digit).*

c) compute the scaled residual P*:*

$P[0] = r(W[0] - z_0^*)$

2. Iteration: *for* $j = 1$ *to* $m - 2$

(a) compute serially the digits of the next residual as:

$W[j] = P[j-1] + (MA + MB)r^{-2}$

where MA *and* MB *correspond to the vectors* $x_{j+2}Y[j]$ *and* $y_{j+2}X[j-1]$*, respectively.*

(b) during the serial computation execute:

i. Selection of output digit:

$z_j^* = rW_0[j] + W_1[j]$

ii. Computation of scaled residual P:

$P[j] = r(W[j] - z_j^*)$

iii. Recoding: $z_{j-2} = RECODE(z_j^*, z_{j-1}^*, z_{j-2}^*)$

3. Obtain the last two recoded digits*:* z_{m-2} *and* z_{m-3}*, assuming* $z_{m-1}^* = z_m^* = 0$.

addition or subtraction. It is more adequate to implement the network using on-line operators. The sooner the most significant digit of the residual comes out, the sooner the selection of the output digit can be done. This feature also allows the utilization of multiple networks, each one working in a different iterations.

The data path for VLP multiplication is shown in figure 4.1. The on-line adder considered in this work is described in [DMV94, TE96] and Section 8.2.1. The digit by vector multiplier (DV Multiplier), combined with on-line adders constitute the main data transformation network to implement the serial computation of the recurrence equation for VLP multiplication. Other modules shown in the figure will be discussed later.

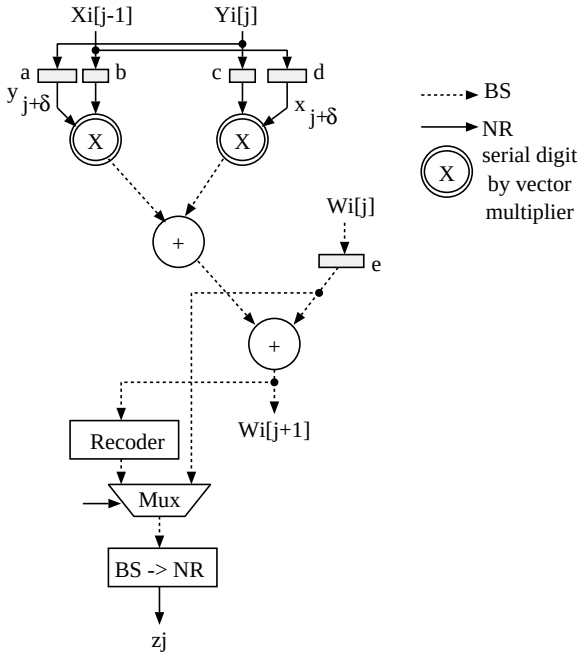

Figure 4.1: Data path for VLP multiplication

The DV multiplier circuit shown in Figure 4.2 works in on-line mode. One n-bit digit of each operand is inserted into the multiple generation and reduction block. The inputs of the multiplier are assumed to be in non-redundant two's complement

format (NR), since the operands are signed-digit vectors. This format uses the least number of bits, making it more adequate for storage and transmission. During the operation of this component one of the inputs is kept fixed while the other changes in each clock cycle (vector digits). The product of a digit x and a vector Y is computed serially as follows:

$$xY = x(.y_1\ y_2\ y_3\ \cdots\ y_k) =$$

$$xY = (.xy_1\ xy_3\ xy_5\ \cdots\ ) + r^{-1}(.xy_2\ xy_4\ \cdots\ )$$

where each xy_i corresponds to two digits. For example: $x = 7$, $Y = (.932)$, $r = 10$, $xY = (.6314) + 0.1(.21) = (.6524)$. The addition is done serially and since both digits of the product xy_i are generated at the same time, it is necessary to delay the least significant digit before it is applied to the serial adder. This operation is performed by the digit alignment section.

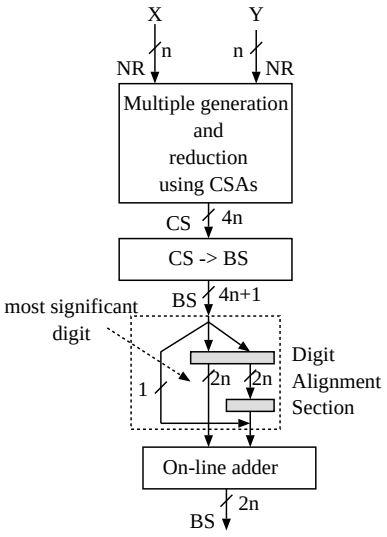

Figure 4.2: Digit by vector multiplier in on-line mode

The multiple generation and reduction block is basically a linear array multiplier that is implemented using radix-4 Booth recoding and carry save adders (CSA)

to accumulate the partial products. Both a linear-array or a tree of adders can be used for partial product reduction. The linear array is not as fast as the tree structure, however it is regular and easy to implement for different operand sizes. A slice of the multiple generation and reduction structure is shown in figure 4.3. Using Booth recoding the bit vector of one operand is recoded into signed digits in radix 4. These recoded digits are used to generate the proper multiple of the other operand. The multiple is then added to the partial product that was accumulated in the layers above.

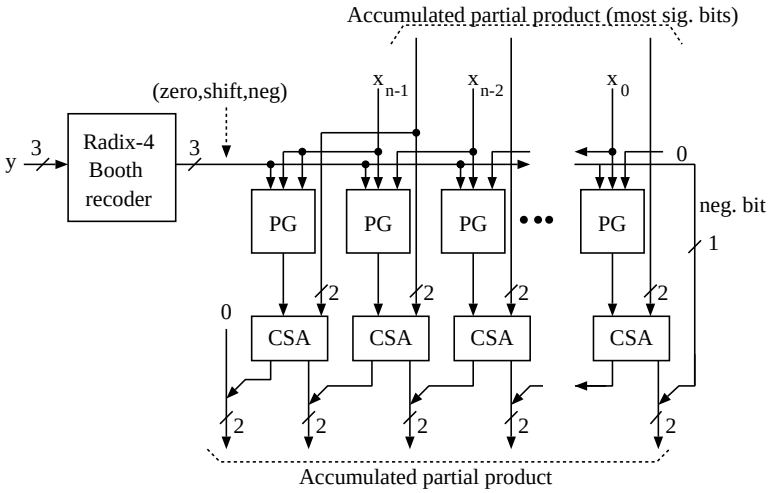

Figure 4.3: One layer of the reduction structure

In a regular multiplier, the CS value obtained from the reduction block would be converted to NR representation before being used by the next module. However, the module that uses the output of the reduction block is an on-line adder which is designed for inputs in BS code. Thus, the reduction block output is converted from CS to BS code directly. The CS to BS converter is discussed in Section 7.1.1. It is a circuit that does not have carry propagation and thus, a delay that does not depend on the digit size.

Since the output of the reduction block is used by the on-line adder that uses

BS code, signed-digit adders (SDA) would be a natural choice to avoid conversion of the output. However, the area of the multiplier would increase with the use of these adders. Each SDA uses 2 full adders (FA) while each CSA uses 1 FA per bit. Using Booth recoding, an 8-bit multiplier would need $(10 * 1 * 4) = 40$ FAs using CSAs, and $(10 * 2 * 4) = 80$ FAs using SDAs. Booth recoding and multiple generation would consume an area of 5*(10+1.5)=57.5 FAs. On the other hand, with CSAs it is necessary to have the CS→BS converter which adds 8 CLBs to the multiplier area. Thus the area of the multipier using SDAs is:

$$\frac{80 - (40 + 8)}{57.5 + 40 + 8} = 30\%$$

larger than the area of the multiplier using CSAs.

The on-line adder works with single digits and the output of the reduction block has two digits of precision. So the two-digit product representation must be transformed into two separate digit representations before it is used by the on-line adder. Using BS code the product can be easier split into two separate digit representations. Let $x = (-13)_{10}$ and $y = (12)_{10}$ be two signed digits in radix 16. The product $xy = (-156)_{10} = (101100100)_2 = (00000100, 10100000)$, where the second representation corresponds to two's complement of the product and the last representation corresponds to BS code representation of the same number. It can be seen that splitting the BS representation into two halves results in: $(0000, 1010) = -10$ and $(0100, 0000) = 4$, that corresponds to the value: $16 * (-10) + 4 = -156$.

Besides the main components for data transformation, other components used in the data path are: the recoder circuit, multiplexer and converter. The recoder circuit is an simplified on-line adder (shown in section 8.3). The multiplexer is used at the end of the VLP operation, when the digits in the residual vector are transferred to the result vector. The conversion from BS code to NR representation is done before the digits are stored into the result memory element. The BS to NR converter is discussed in Section 7.1.2.

Minimum delays of the datapath are shown in Figure 4.4. The minimum delay is caused by the use of serial modules in the data path network. The serial digit by vector multiplier has a delay of 3 cycles: 2 cycles in the on-line adder and 1 cycle in the digit alignment section (between the CS→BS and on-line adder modules, Figure 4.2).

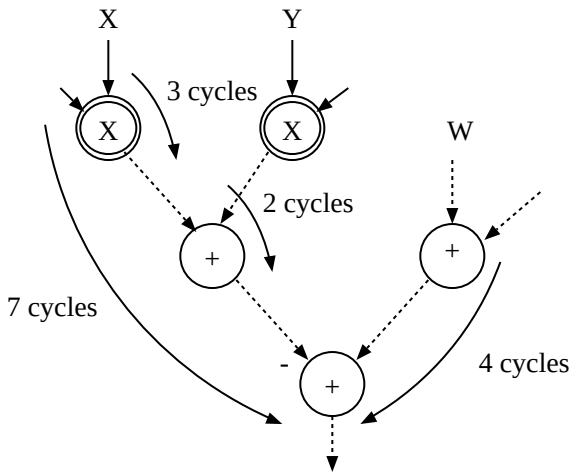

Figure 4.4: Data path delays

4.1.2 Data Arrangement for Serial Computation

The vectors used in VLP multiplication are:

- X and Y are the operands composed of m fractional digits. The vector position i contains x_i and y_i, respectively. A pointer named *opt* indicates the digit being considered at step j.
- Z vector holds the output digits generated by the RECODER. In order to avoid using another pointer just for this vector, *opt* may also be used to reference the vector position of the last written output digit.
- W is the vector that holds the residual digits. In some point in time the vector will hold some digits of $W[j]$ and some digits of $W[j-1]$. The pointer

p is used to read digits of the residual $W[j-1]$. The same pointer, less an offset, is used to write digits of $W[j]$ into the vector.

The digit vectors are shown in Figure 4.5. In order to simplify the manipulation

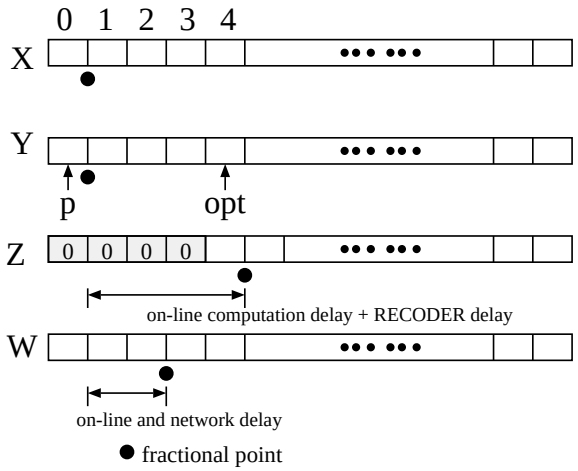

Figure 4.5: Data vectors for VLP multiplication

of the long-precision digit vectors, the alignment of values is performed shifting the digit vectors relatively to each other. The amount of shifting depends on the recurrence equation and the delay of arithmetic modules used in the VLP division data path. The case shown in the Figure considers a non-pipelined data path. The use of pipelining is discussed in section 4.2. Considering the recurrence equation (4.1) both X and Y, should be aligned and the displacement between W and X (or Y) is given by $\delta = 3$. However, as the paths involving X/Y and W have a difference of 5 cycles, W must be inserted 2 cycles after the digits in X/Y begin to be inserted in order to get a delay of 3 cycles between them (δ), as shown in Figure 4.5. One integer digit is kept for vectors X and Y, because a redundant representation can be used.

4.1.3 Serial Computation of the Residual

Algorithm 4 gives the steps required to manipulate the data path and data vectors in order to execute serial computation of the residual, which is the main part of the VLP multiplication operation. The algorithm is described using pseudo-code. In the pseudo-code we refer to the data vectors as done in C-code ($W[i]$, for example, is the vector position with index i). We assume that the registers a and d contain the value of the operand digits $x_{j+\delta-1}$ and $y_{j+\delta-1}$. This condition must be true after the initialization phase and after each iteration.

4.2 Pipelined Data Path

All components in the data path use redundant number representation that allows the implementation of operators with fixed delay, independent of the digit radix that is being used. The delay of these components can be as low as 1 4-input LUT delay, plus interconnect and FF delay.

Table 4.1 shows the maximum number of pipeline stages that can be added to each component of the data path. We use the term "added" because the on-line adder and the digit by vector multiplier (DV) already have registers in the non-pipeline implementation. In this evaluation we assume that 4-input LUT FPGAs are used. The use of k-input FPGAs, with $k < 4$ would incur in more logic levels, and more stages could be created.

The data path already has a delay of 7 cycles in the non-pipelined implementation. The total delay of the pipelined version is $7 + p$ cycles. The maximum degree of pipelining is computed as:

$$p_{max} = 6 + \left\lfloor \frac{n}{2} \right\rfloor \tag{4.3}$$

The data path has a critical path delay of $13 + \lfloor \frac{n}{2} \rfloor$ cycles. In the case of signed-digits in radix 2^{16}, 17 bits are used and the delay is 21 cycles with $p_{max} = 14$.

When using a deeply pipelined structure it is possible to start inserting data of

Algorithm 4 *: Serial computation of the residual for VLP Multiplication*

1. $p = 0$
2. *loop while* $p \leq opt + 7$ *(comment: 7 is the delay of the non-pipelined data path network):*
   - *if* $p < opt$ *read* $X[p]$ *and store into register b, otherwise clear register b*
   - *if* $p \leq opt$ *read* $Y[p]$ *and store into register c, otherwise clear register c*
   - *if* $p = opt + 1$ *store X[p] into register d and Y[p] into register a (store the input digits for next iteration, corresponding to* $x_{j+\delta-1}$ *and* $y_{j+\delta-1}$*)*
   - *read* $W[p]$ *and store into register e*
   - *if* $p = 4$ *store output of the data path into the RECODER circuit (most significant digit of* z_j^**) (comment: RECODER circuit is defined in section 8.3)*
   - *if* $p = 5$ *store output of the data path into the RECODER circuit (second digit of* z_j^**)*
   - *if* $p \geq 6$ *write the output of the data path to* $W[p-3]$ *(comment: the number 3 comes from the delay of the path between* $W_i[j-1]$ *and* $W_i[j]$ *(2) plus effect of scaling W by r in each iteration, that corresponds to one more cycle).*
   - *increment p*
3. *adjust pointers to vectors:* $opt = opt + 1$

Component	Stages
DV Multiplier - BR	1
DV Multiplier - PP	1
DV Multiplier - reduction	$\lfloor \frac{n}{2} \rfloor$ (for n-bit digits)
DV Multiplier - CSBS converter	1
On-line adder	1

Table 4.1: Number of stages in the digit by vector multiplier

the next iteration some cycles after the data for the present iteration is applied. Dependencies in the path force the insertiong of a bubble of 5 cycles. There is a 1 cycle dependency on the DV multiplier (digit alignment) and 1 cycle dependency in each on-line adder. So, after the digits of iteration j are inserted, zeros are applied for 5 cycles at the DV multiplier inputs, before the digits of iteration $j+1$ begin to be used as inputs. The longest pipelined path in the network used for VLP multiplication is shown in the figure 4.6.

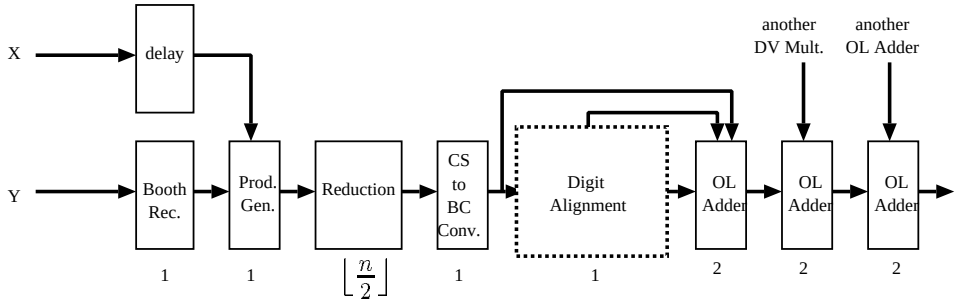

Figure 4.6: Pipelined Data Path

4.3 VLP Multiplication with Precision less than $2m$

The multiplication of two integer operands with m digits generates a full-precision result with $2m$ digits. When working with fractions, it is sometimes desirable to obtain a product that has less than $2m$ digits, usually with the same

precision of the operands (m digits). In this case, the computation of the full-precision result, followed by rounding can imply in twice the work that would be required in fact.

The discussion on long-precision computation in [Knu69] states that it is possible to reduce the amount of computation discarding multiples that are not going to affect the result significantly. Assume that $x = \sum_{i=1}^{m} x_i r^{-i}$ and $y = \sum_{i=1}^{m} y_i r^{-i}$ with $x_i, y_i \in \{-\rho, ..., -1, 0, 1, ..., \rho\}$, and $\rho = r - 1$. Multiples that belong to the same column p in the multiplication matrix are generated by the multiplication of digits x_i and y_j, such that $i + j = p$. Consider the situation when multiplies in columns $p > m + k$, with $k \geq 0$, are removed from the final product. It is possible to prove that the maximum error in this truncation process is $\epsilon_k = (m - k)r^{-(m+k-1)}$ The multiples that are discarded for $p = 9$ in a 6x6 multiplication are shown in Figure 4.7.

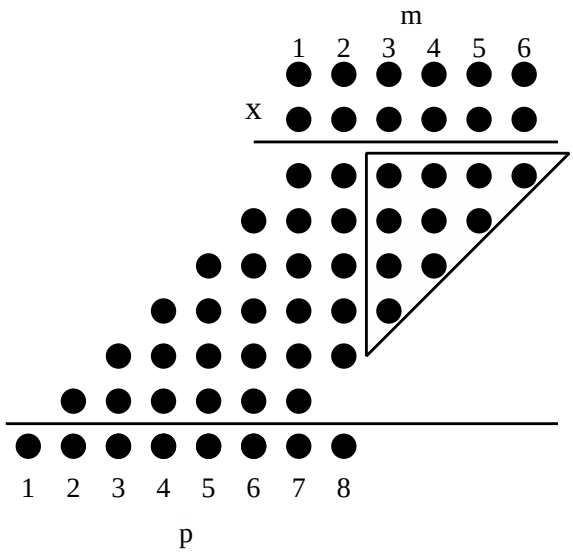

Figure 4.7: Truncated multiplication result

Proof 1 *generates the following maximum error ϵ:*

$$\epsilon = \left[\sum_{i=1}^{2m-(m+k)} i(r-1)^2 r^{i-1} \right] r^{-2m}$$

$$\epsilon = \left[\sum_{i=1}^{m-k} i r^i \right] \frac{(r-1)^2}{r} r^{-2m}$$

$$\epsilon = \frac{r\{1-(m-k+1)r^{m-k}+(m-k)r^{m-k+1}\}}{(1-r)^2} \frac{(r-1)^2}{r} r^{-2m}$$

$$\epsilon = r^{-2m} - (m-k+1)r^{-m-k} + (m-k)r^{-m-k+1}$$

Based on the fact that

$$r^{-2m} - (m-k+1)r^{-m-k} + (m-k)r^{-m-k+1} < (m-k)r^{-m-k+1}$$

for $0 \leq k \leq m$, the following upper bound on the maximum error is obtained:

$$\epsilon < \epsilon_k = (m-k)r^{-m-k+1} \tag{4.4}$$

Notice that k is related to the output digit position, not the input digit. It would be better to associate the error with the input digit position. Assuming that the multiplication algorithm starts to disregard multiples of digits of X and Y starting after input digit position t, the algorithm should discard multiples in columns to the right of column $2t$, that corresponds to column $p = 2t+1$. Since, from the previous expressions, $p > m+k$, we have the following relation:

$$p = m+k+1 \rightarrow k = p-m-1$$

$$\epsilon_k = (m-k)r^{-(m+k-1)} = (2m-p+1)r^{-(p-2)}$$

as $p = 2t+1$, we get the error based on the input position t as:

$$\gamma_t = 2(m-t)r^{-(2t-1)}$$

Based on this maximum error, we may now obtain the minimum number of operand digits that are required to obtain a requested output precision.

4.3.1 Truncation point to Satisfy Output Precision

For VLP computation, rounding results is meaningless. If the error of a truncated result is not acceptable (for m fractional digits it is r^{-m}), a greater precision is used and the error can be reduced as much as desired. For this reason we do not consider rounding problems in this thesis.

So, when the requested output precision is $m \leq m' < 2m$ we want to determine the number of input digits such that the elimination of multiples is not going to cause an error that is greater than the truncation error of the result with m' fractional digits. The following relation must be satisfied:

$$\gamma_t = 2(m-t)r^{-(2t-1)} < r^{-m'} \tag{4.5}$$

From (4.5), as $0 < t < m$, we obtain:

$$1 < 2(m-t) < 2m \tag{4.6}$$

and thus

$$r^{-(2t-1)} < \frac{r^{-m'}}{2m} \tag{4.7}$$

that implies

$$-2t+1 < -m' - log_r(2m) \tag{4.8}$$

$$2t > m' + log_r(2m) + 1 \tag{4.9}$$

$$t > \frac{m'}{2} + \frac{log_r(2m)}{2} + \frac{1}{2} \tag{4.10}$$

As the relation $log_r(2m) \leq 2$ is true for large r and m, the error bound is satisfied for

$$t = \lceil \frac{m'}{2} \rceil + 2 \tag{4.11}$$

For the values of r considered in this thesis (above 128) the value of the maximum m that satisfies the condition is reasonably large $m_{max} = 8192$ digits, that corresponds to 57,344 bits of output precision.

So, for all pratical purposes, we are going to use the truncation point ginven in equation (4.11) as the input position for truncation, when the output precision is m'.

4.3.2 VLP Multiplication Algorithm for Truncated Results

The effect of disregarding multiples is obtained in the VLP multiplier making a slight modification in the original algorithm. The idea is to reduce the precision of the working vectors, one digit at a time. Assume that the algorithm starts to truncate the multiples after input digit position t. At step j of the VLP multiplication, the algorithm computes the multiples $x_i y_j$, with $i + j \leq 2t$. The input precision is m digits. The difference in this algorithm and the one for full precision resides in the computation of the vector $MA = x_{j+\delta-1}Y$ and $MB = y_{j+\delta-1}X$. Instead of using a vector $Y[j]$ with precision of $j + \delta - 1$ for all iterations, at step j MA is computed as ($\delta = 3$):

$$MA = \begin{cases} x_{j+2}Y[j] & \textbf{if } j \leq t-2 \\ x_{j+2}Y[2t-j-2] & \text{otherwise} \end{cases} \tag{4.12}$$

and MB is computed as:

$$MB = \begin{cases} y_{j+2}X[j-1] & \textbf{if } j \leq t-2 \\ y_{j+2}X[2t-j-3] & \text{otherwise} \end{cases} \tag{4.13}$$

where $X[j]$ and $Y[j]$ are defined in equation (4.2).

The description shows that after input digit t, the precision of the stored operands is reduced, one digit per iteration.

The implementation of this method is reflected in the serial computation of the recurrence equation shown in Algorithm 4 only by using a new pointer for the main loop, instead of *opt*. The pointer works like *opt* while the precision increases, and starts to be decremented when the condition $j = t - 2$ is reached.

The simulation of the algorithm execution is shown in figure 4.8. The shaded area shows the gain over full precision computation.

```
generation of product with the same precision as operands
give the precision (n) of operands in digits (less than 200): 20
give the precision (m) of the result in digits(n<=m<=2n): 20
trace in which step (200 if not wanted): 200
want random generation of operands?y
give a seed for the random number generator: 100
X = 82 82 6 106 66 19 73 1 44 8 28 70 113 15 118 29 122 116 24 23
0.645633
Y = 83 29 31 53 6 24 50 30 50 49 82 21 54 87 57 22 8 117 124 92
0.650222
Truncation point = 12
W =   0  53  22   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
W =  53  93 114  74   0   0   0   0   0   0   0   0   0   0   0   0   0   0
W =  94  10  63  29  58   0   0   0   0   0   0   0   0   0   0   0   0   0
W =  11  38  47  82  63 114   0   0   0   0   0   0   0   0   0   0   0   0
W =  38  94  55  54  52  41  12   0   0   0   0   0   0   0   0   0   0   0
W =  94  83  35  17  33 120  37  72   0   0   0   0   0   0   0   0   0   0
W =  83 114 112 127  67  19 119  42  66   0   0   0   0   0   0   0   0   0
W = 115   5   1 126   0  39 113  37  64  30   0   0   0   0   0   0   0   0
W =   5  62 112  13  99  93  33  68   2 107  24   0   0   0   0   0   0   0
W =  63  20 121   2  39  45  31   4 118  76 127   8   0   0   0   0   0   0
W =  21  64  21  34  77  10  98 102  46  64  39 125 120   0   0   0   0   0
W =  64  80  44   7  49  33   4 116 122  19  98  10  76   0   0   0   0   0
W =  81  24  50 104 111 103  36  95  39  20 103 124   0   0   0   0   0   0
W =  24 116  95   9  16 115  55  59  86  30 112   0   0   0   0   0   0   0
W = 117  80  76  80 115  98  80 111 108  41   0   0   0   0   0   0   0   0
W =  80 109  88  81   7 122  82  82 111   0   0   0   0   0   0   0   0   0
W = 110  45  16   8  41 110  64   7   0   0   0   0   0   0   0   0   0   0
W =  46  39   3 106  73 114  49   0   0   0   0   0   0   0   0   0   0   0
W =  39  99  63  70  41   9   0   0   0   0   0   0   0   0   0   0   0   0
W = 100   9 115  64  49   0   0   0   0   0   0   0   0   0   0   0   0   0
Zj = 53 94 11 38 94 83 115 5 63 21 64 81 24 117 80 110 46 39 100 9
0.419805
```

Figure 4.8: Variable output precision operation of the VLP multiplier ($r = 2^7$)

4.3.3 Gain in Performance

The number of multiples generated in a full precision computation is m^2, assuming operands of the same size (m digits).

Considering $m \leq m' \leq 2m-4$ (for $2m-3 \leq m' \leq 2m$ all multiples are computed), the number of input digits required is:

$$t = \lceil \frac{m'}{2} \rceil + 2$$

that corresponds to a truncation of multiples starting at output fractional position $m' = 2t$. The number of multiples that are computed in this case as a function of m' is

$$M = m^2 - \left\lfloor \frac{(2m-m')^2 + 3(2m-m') + 2}{2} \right\rfloor \tag{4.14}$$

The gain of this truncation method to obtain the required digits for rounding, over the full-precision case followed by rounding is:

$$S = \frac{m^2}{M} \tag{4.15}$$

A graph of the speedup obtained when operand precision $m = 250$ and output precision $250 \leq m' < 498$ is shown in Figure 4.9.

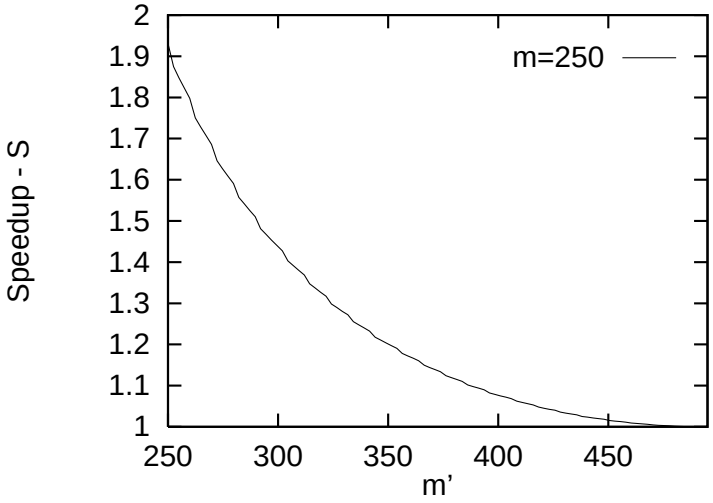

Figure 4.9: Speedup of VLP multiplier with variable output precision over full precision multiplication

4.4 Operands with Different Precision

In long precision computations it may happen that two operands have different precision, lets call them m_x and m_y, and assume that $m_x > m_y$. While $j + \delta \leq m_y$ we use the algorithm already described. After this point, digits of Y are all zeroes and the VLP multiplier can be modified to make use of its two digit by vector multipliers. Two output digits are computed in each iteration.

We modify the recurrence equation as follows, assuming that Y does not change from one iteration to the next, and the term $y_{j+\delta}$ of the original equation is zero.

$$W[j+1] = r(W[j] - z_j) + r^{-\delta}(x_{j+\delta}Y) \tag{4.16}$$

$$W[j+2] = r(W[j+1] - z_{j+1}) + r^{-\delta}(x_{j+\delta+1}Y) \tag{4.17}$$

replacing equation 4.16 into 4.17, and doing proper manipulation, we get:

$$W[j+2] = r^2(W[j] - (z_j + z_{j+1}r^{-1})) + r^{-\delta+1}(x_{j+\delta}Y) + r^{-\delta}(x_{j+\delta+1}Y) \tag{4.18}$$

the output is composed by digits z_j and z_{j+1}.

The data path is slightly modified to perform this operation such that the second multiplier begins to receive data from the input register of the first. Consecutive digits of X are used as the other inputs for the digit multipliers.

This modification makes the circuit operate two times faster than the original one, after the last digit of the shortest operand is received.

4.5 Execution time of the VLP Multiplier

In this estimate we consider m digit operands and a result precision m_p that is in the range $[m, 2m]$. The operand precision dictates the number of iterations to be executed, and the output precision defines the number of digits used in each iteration.

Using a pipelined data path with non-overlapped operation, the number of cycles to execute VLP multiplication for an output precision $m \leq m' \leq 2m$ is given in equation 4.19. The operands have 1 integer digit, as shown in figure 4.5. The initialization of the residual is done with 2 fractional digits. The precision increases until fractional digit t, as shown in section 4.3.1, when it starts to decrease one digit per iteration. Once the input digits were consumed, 2 digits are obtained from the RECODER and the remaining $m' - m + 2$ digits are transferred from the residual memory to the output, using the data path. As in this case, the BS→NR converter may have a delay that is larger then the coprocessor cycle time, we assume that the conversion time for each digit in the residual vector takes T_{conv}

coprocessor cycles.

$$T_{VLPmul} = \sum_{i=2}^{t}(i+7+p) + \sum_{i=t+1}^{m}(2t-i+8+p) + (m'-m+2)T_{conv} \quad (4.19)$$

For our purposes, as explained before, the value of t is:

$$t = \lceil \frac{m'}{2} \rceil + 2 \quad (4.20)$$

for $m \leq m' \leq 2m-4$. When $t = m$, the second summation is not computed. The value of t depends on the precision and the digit radix used, thus care must be taken to guarantee that the conditions for truncation are satisfied.

In an overlapped operation, the equation for the number of cycles is:

$$T_{VLPmul} = \sum_{i=2}^{t}(i+1+5) + \sum_{i=t+1}^{m}(2t-i+1+1+5) + (7+p-5) + (m'-m+2)T_{conv} \quad (4.21)$$

using a buble of 5 cycles. The term $(7+p-5)$ corresponds to the data path latency for the last iteration minus the the buble already considered in the summation term, for the last iteration.

The time for an iteration when the precision of vector Y is i fractional digits and successive iterations are not overlapped is given as:

$$T_{iter}(i) = \begin{cases} (i+1+7+p-1) = i+p+7 & \text{if } i \leq t \\ 2t-i+8+p & \text{otherwise} \end{cases} \quad (4.22)$$

Chapter 5

VLP Divider

This chapter presents an algorithm for VLP division and discusses main implementation issues. The on-line recurrence equation is obtained from [Tu90], for the on-line division operation ($\frac{N}{D} = Q$):

$$W[j] = r(W[j-1] - q_{j-1}D[j-1]) + n_{j+\delta-1}r^{-\delta+1} - d_{j+\delta-1}Q[j-1]r^{-\delta+1} \quad (5.1)$$

where

$$N[j] = \sum_{i=0}^{j+\delta-1} n_i r^{-i} \quad (5.2)$$

$$D[j] = \sum_{i=0}^{j+\delta-1} d_i r^{-i} \quad (5.3)$$

$$Q[j] = \sum_{i=0}^{j} q_i r^{-i} \quad (5.4)$$

$$W[j] = \sum_{i=0}^{j} w_i r^{-i} \quad (5.5)$$

and with the initial condition of $W[0] = N[0]$. The on-line delay (δ) for high-radix is 3, based on [Tu90].

Equation (5.1) implies that the value of the quotient digit (q_{j-1}) used in the iteration to obtain $W[j]$ must already be part of variable Q. It is more convenient for the implementation of the VLP algorithm presented in this chapter that the insertion of the new quotient digit be done at the end of the iteration. This is obtained by the following manipulation of equation 5.1:

$$W[j] = r(W[j-1] - q_{j-1}D[j-1]) + n_{j+\delta-1}r^{-\delta+1} - d_{j+\delta-1}(q_{j-1}r^{-j+1} + Q[j-2])r^{-\delta+1}$$

$$W[j] = r(W[j-1] - q_{j-1}D[j-1]) + n_{j+\delta-1}r^{-\delta+1} - d_{j+\delta-1}r^{-j-\delta+1}q_{j-1}r - d_{j+\delta-1}Q[j-2]r^{-\delta+1}$$

$$W[j] = r(W[j-1] - q_{j-1}(D[j-1] + d_{j+\delta-1}r^{-j-\delta+1})) + n_{j+\delta-1}r^{-\delta+1} - d_{j+\delta-1}Q[j-2]r^{-\delta+1}$$

that results in:

$$W[j] = rW[j-1] + n_{j+\delta-1}r^{-\delta+1} - d_{j+\delta-1}Q[j-2]r^{-\delta+1} - rq_{j-1}D[j] \qquad (5.6)$$

with the initial condition of $W[0] = N[0]$. We base our discussion of VLP division in equation (5.6). It is easier to obtain $D[j]$ than $Q[j-1]$ and the complexity of the recurrence equation is exactly the same.

The output digit q_j is selected based on a function of the residual value and the divisor. The selection function is represented as $\mathcal{SEL}(W[j], d)$, and for implementation, it should consider only a short precision value of both parameters. Also, when using high-radix digits, the selection function becomes very complex. One solution to simplify the function is to scale the divisor to a value close to 1 as done in [EL94] for digit recurrent algorithms. When the divisor is scaled, the selection is done based only on the short precision residual, as shown in equation (5.7). The conditions for this selection function are presented in section 5.2.

$$q_j = \mathcal{SEL}(\hat{W}[j]) \qquad (5.7)$$

In the next sections we present the algorithm for high-radix VLP on-line division, the data path organization, digit selection and timing characteristics.

5.1 VLP Divison Algorithm

Based on the general model of the reconfigurable coprocessor, the operands, result and residual data are stored in memory elements that are referenced in this

section as digit vectors. High-radix signed-digits are considered (radix $r = 2^n$). The operands and result have m digits of precision. The vectors used in VLP division are:

- N is the dividend vector and contains one integer digit and m fractional digits. The value of N corresponds to $\sum_{i=0}^{m} n_i r^{-i}$ and the vector position i contains n_i, starting from position $i = 0$.
- $Q[j]$ contains all quotient digits generated until and including iteration j.
- D is the divisor vector that holds one integer digit and m fractional digits. All digits of the divisor may be present in the memory, however, during the algorithm, only partial view of D is used, referenced as $D[j]$. $D[j]$ represents the most significant digits of the divisor, from the first digit to the one that is being used at iteration j, or $d_{j+\delta-1}$.
- $W[j]$ is the residual vector at step j.

These data vectors are shown in Figure 5.1. In order to simplify the manipulation of the long-precision digit vectors, the alignment of values is performed shifting the digit vectors relatively to each other. The same pointer p can be used to read all digit vectors. The amount of shifting depends on the recurrence equation and on the delay of arithmetic modules used in the VLP division data path. The case shown in the Figure considers a non-pipelined data path. The use of pipelining is discussed in section 5.7.

The pointer *opt* indicates the input operand digit being considered at step j. The same pointer is used to insert the quotient digit into Q.

Algorithm 5 shows the steps to perform VLP division. We assume that $N < D$ to avoid overflow.

The $APPEND(Q[j], q_j)$ function constitutes on concatenating the digit q_j to the vector $Q[j] = (q_0, q_1, q_2, \ldots, q_{j-1})$, such that the resulting vector is $Q[j] = (q_0, q_1, q_2, \ldots, q_{j-1}, q_j)$. The value of q_j is written to $Q[opt]$.

Algorithm 5 *: VLP division*

1. Initialization:

(a) $W_i[0] = n_i$, $i = 0, \ldots, \delta - 1$

$W_i[0] = 0$, $i > (\delta - 1)$

(b) $q_0 = \mathcal{SEL}(\hat{W}[0])$

(c) $Q[-1] \leftarrow 0$

(d) $opt = \delta$

2. Iteration: *for* $j = 1$ *to* m

(a) compute serially the digits of the next residual as:

$W[j] = rW[j-1] + n_{j+\delta-1} r^{-\delta+1} - d_{j+\delta-1} Q[j-2] r^{-\delta+1} - r q_{j-1} D[j]$

(b) update quotiend vector: $Q[j-1] \leftarrow APPEND(Q[j-2], q_{j-1})$

(c) Selection of quotient digit:

$q_j = \mathcal{SEL}(\hat{W}[j])$

3. update quotiend vector: $Q[m] \leftarrow APPEND(Q[m-1], q_m)$

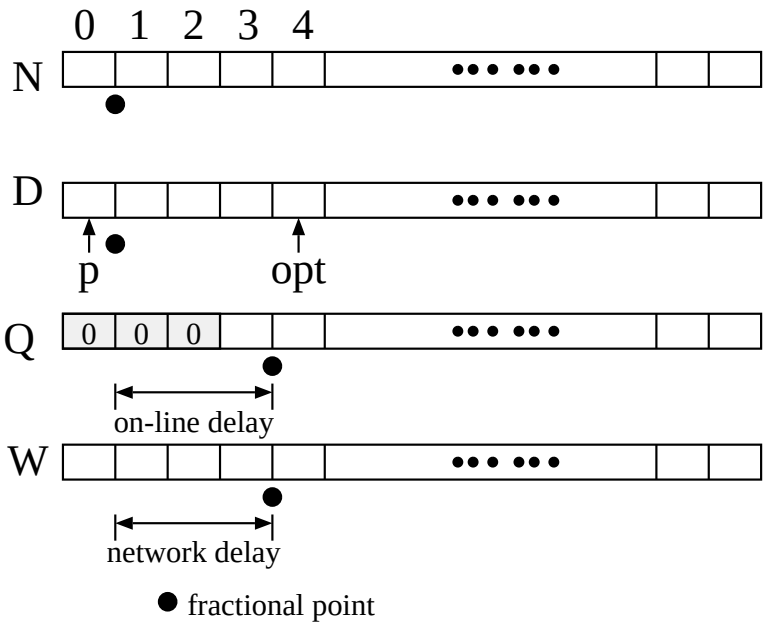

Figure 5.1: Digit vector for VLP Division

The VLP division datapath shown in Figure 5.2 uses basically the same arithmetic operators used in the VLP multiplier data path presented in Chapter 4. The the same type of arithmetic operators are used. Compared to the VLP multiplier this data path uses a serial subtractor to generate $W_i[j]$, the inputs of the modules come from different memory elements and the selection function changed.

The relative position of digits in each vector is explained in terms of the network delays and the recurrence equation. As digits from vectors D and Q are inserted at the same height of the network tree, the difference between them is only the on-line delay (based on the recurrence equation). In a non-pipelined network, the digit by vector multipliers have a delay of 3 cycles. The on-line adders (or subtractor) have a delay of 2 cycles. Considering the recurrence equation the residual W must be aligned with vector D. However, the network delay in the path from input D to the level where W is inserted corresponds to 3 cycles. Thus, W must wait for 3 cycles before it is applied to the network. That means the digits of W must be

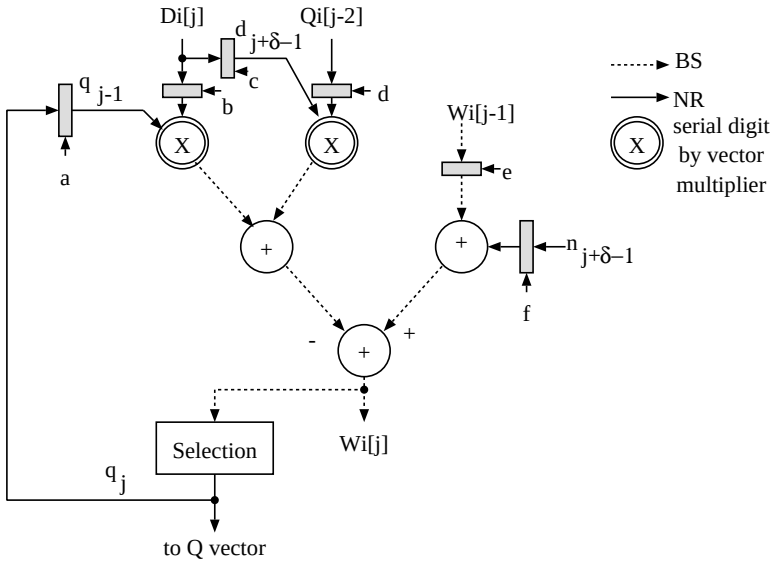

Figure 5.2: Data path for VLP on-line division

displaced 3 positions to the right of D. In a pipelined implementation, the distance increases by the number of stages inserted in the DV multiplier.

Proper control over the data path registers (marked in the data path with small letters) allows the division control circuit to generate zeroes as inputs in the following cases:

- input n_j must have the digit of the dividend at step j, for only one cycle. For the rest of the time, this input has a zero value;
- although all (or most) digits of the divisor are already in the coprocessor memory, digits that are not included in $D[j-1]$ should not be loaded into the input register of $D_i[j]$.

other vectors can be continuously read, since the other digit positions will contain only zeroes.

A pseudo-code that details the serial computation of the residual in the VLP algorithm is shown in Algorithm 6. We assume that the value in register c is

already correct (contains the value of the present divisor digit).

5.2 Selection Function

The traditional selection of the quotient digit is done based on the values of the residual and the divisor. For small radices, a quotient digit is selected by comparing the most significant digits of the residual and divisor with constants. Depending on the range of them, the proper output digit is selected.

When high-radix digits are used, the implementation of the selection function using table lookup is very expensive, there are many constants to compare and the size of the required table is prohibitive. Solutions for the problem were given in [EL94]. The one that we investigate in this thesis is based on scaling the divisor to a range that allows the selection of the quotient digit based only on a rounded value of the residual. This operation is easy to implement.

In order to have convergence conditions satisfied in the recurrence equation, the operands must be pre-scaled to a pre-defined range. The dividend and divisor are scaled by a constant M such that the scaled divisor d is close to 1, in the range: $1 - \alpha \leq d \leq 1 + \alpha$.

This method was used in digit-recurrence algorithm for high-radix ($2^9 - 2^{18}$) and for small radix on-line division [TE87]. In this section we derive the convergence bounds for this approach for VLP division using high-radix. The error analysis considers the selection function by rounding and maximal redundancy of the digit set. For the VLP division implementation we use signed-digits in BS code with maximal redundancy (redundancy factor K=1) for the digit set $\mathcal{D}_\rho = \{-\rho, ..., \rho\}$. The redundancy factor is defined as $K = \frac{\rho}{r-1}$.

The quotient digit is obtained as:

$$q_j = \mathcal{S}(\hat{W}[j]) = \left\lfloor \hat{W}[j] + \frac{1}{2} \right\rfloor \tag{5.8}$$

where $\hat{W}[j]$ is the estimate of the residual.

Algorithm 6 *: serial computation of the residual - VLP division*

1. *initialization*
   - $p = 0$
   - *store* q_j *into register a*
2. *loop while* $p \leq opt + 7$ *(comment: 7 is the delay of the non-pipelined data path network):*
   - *if* $p > opt$ *clear register b, otherwise read* $D[p]$ *and store into register b*
   - *if* $p = opt + 1$ *read* $D[p]$ *and store into register c*
   - *read* $Q[p]$ *and* $W[p]$ *from memory and store into registers d and e*
   - *if* $p = \delta + 3$ *store* $N[opt]$ *into register f, otherwise clear register f*
   - *issue control signals to the selection function to store the first most significant digits of the residual (comment: the selection function circuit is defined in section 5.5)*
   - *if* $p \geq 5$ *write the output of the data path to* $W[p-5]$ *(comment: the number 5 comes from the delay of the path between* $W_i[j-1]$ *and* $W_i[j]$*, plus an extra cycle to scale the residual).*
   - *increment p*
3. *adjust pointers to vectors:* $opt = opt + 1$

Adopting the same terminology used in [EL94] we compute the *remanent* γ as:

$$\gamma = W[j] - q_j \tag{5.9}$$

Assuming that the residual is represented with signed-digits and the estimate of the residual is obtained based on truncation of the residual value at the t^{th} fractional bit, the bounds for the remanent are:

$$-\frac{1}{2} - 2^{-t} \leq \gamma < \frac{1}{2} + 2^{-t} \tag{5.10}$$

The residual value is obtained as a function of the remanent by manipulation of equation (5.1), as follows:

$$W[j+1] = r(W[j] - q_j D[j]) + n_{j+\delta} r^{-\delta+1} - d_{j+\delta} Q[j] r^{-\delta+1} \tag{5.11}$$

$$W[j+1] = r(W[j] - (q_j - q_j) - q_j D[j]) + n_{j+\delta} r^{-\delta+1} - d_{j+\delta} Q[j] r^{-\delta+1} \tag{5.12}$$

Inserting equation (5.9) into equation (5.12):

$$W[j+1] = r\gamma + r q_j (1 - D[j]) + n_{j+\delta} r^{-\delta+1} - d_{j+\delta} Q[j] r^{-\delta+1} \tag{5.13}$$

One condition for convergence comes from the recurrence equation of on-line division (5.1) considering $d_{j+\delta-1} = 0$ and $n_{j+\delta-1} = q_{j-1} = \rho$, with the value of the residual represented as l:

$$l \geq r(l - \rho d) + \rho r^{-\delta+1}$$

$$l \leq rKd - Kr^{-\delta+1}$$

that after substitution of the variable l by $W[j]$ results in the upper bound:

$$|W[j]| \leq K(rd - r^{-\delta+1}) \tag{5.14}$$

It is also necessary to avoid that the selection function generates a digit that is greater or equal to r. The following relation applies:

$$|W[j]| < r - \frac{1}{2} - 2^{-t} \tag{5.15}$$

Combining equations (5.14) and (5.15) for the case $K = 1$, the bound for the residual becomes:

$$max(-r + \frac{1}{2} + 2^{-t}, -rd + r^{-\delta+1}) < W[j] < min(r - \frac{1}{2} - 2^{-t}, rd - r^{-\delta+1}) \quad (5.16)$$

As the bounds are symmetrical we analyze only the upper bound based on the conditions imposed by equations (5.16) and (5.13):

$$r(\frac{1}{2} + 2^{-t}) + r(r-1)|1-d| + (r-1)r^{-\delta} < min(rd - r^{-\delta+1}, r - \frac{1}{2} - 2^{-t}) \quad (5.17)$$

or

$$(\frac{1}{2} + 2^{-t}) + (r-1)|1-d| + (r-1)r^{-\delta-1} < min(d - r^{-\delta}, 1 - \frac{1}{2r} - \frac{2^{-t}}{r}) \quad (5.18)$$

The solutions for this relation is calculated dividing the range of values in three regions as follows:

- $d \geq 1$

  $$(\frac{1}{2} + 2^{-t}) + (r-1)(d-1) + (r-1)r^{-\delta-1} < 1 - \frac{1}{2r} - \frac{2^{-t}}{r} \quad (5.19)$$

  that results in

  $$d < 1 + \frac{1}{2r} - \frac{2^{-t}(r+1)}{r(r-1)} - r^{-\delta-1} \quad (5.20)$$

- $1 > d \geq 1 - \frac{1}{2r} - \frac{2^{-t}}{r} + r^{-\delta-1}$

  $$(\frac{1}{2} + 2^{-t}) + (r-1)(1-d) + (r-1)r^{-\delta-1} < 1 - \frac{1}{2r} - \frac{2^{-t}}{r} \quad (5.21)$$

  that imposes the bound:

  $$d > 1 - \frac{1}{2r} + \frac{2^{-t}(r+1)}{r(r-1)} + r^{-\delta-1} \quad (5.22)$$

- for $d < 1 - \frac{1}{2r} - \frac{2^{-t}}{r} + r^{-\delta-1}$ the expression (5.18) results in contradiction

Combining the possible bounds from expressions (5.20) and (5.22) we obtain the range for the pre-scaled divisor as:

$$1 - \frac{1}{2r} + \frac{2^{-t}(r+1)}{r(r-1)} + r^{-\delta-1} < d < 1 + \frac{1}{2r} - \frac{2^{-t}(r+1)}{r(r-1)} - r^{-\delta-1} \tag{5.23}$$

For $\delta \geq 3$ it is easy to show that $t \geq 2$ would be enough to obtain consistent upper and lower limits for the scaled divisor. When working with high-radix digits, the number of bits in one fractional digit is $\lceil log_2(r) \rceil$, that is more than enough to obtain a reasonable range for the pre-scaled divisor, in order to allow this selection function. So, assuming that $t = log_2(r)$, we get the following bounds on the scaled divisor:

$$1 - \frac{1}{2r} + \frac{r+1}{r^2(r-1)} + r^{-\delta-1} < d < 1 + \frac{1}{2r} - \frac{r+1}{r^2(r-1)} - r^{-\delta-1}$$

that could be rewritten as:

$$1 - \alpha < d < 1 + \alpha \tag{5.24}$$

where

$$\alpha = \frac{1}{2r} - \epsilon \tag{5.25}$$

and

$$\epsilon = \frac{r+1}{r^2(r-1)} + r^{-\delta-1} \tag{5.26}$$

and only one fractional digit is used for selection.

An example of the on-line division using pre-scaled operands is shown in Table 5.1.

5.3 Scaling Factor (M)

From the bounds of the scaled divisor, we can determine the scaling factor M that is used to scale the divisor and dividend. Basically, the value of M is obtained by the reciprocal of the divisor, computed using some of the most significant digits.

This short precision reciprocal value could be obtained by the host processor, using the FP arithmetic unit, or by a dedicated circuit in the coprocessor. Since

$r = 64$ and $t = 3$ and $\delta = 3$					
$0.992439 < d < 1.007560$					
$N = (0.56547772)_8$ and $D = (1.00172700)_8$=1.0037498					
$W[0] =$		56.	54	00	$q_0 = 1$
$rW[0] =$	56	54.	00	00	
$-q_0 D[0]r =$	1	00.	17	00	
$n_3 r^{-2} =$				77	
$-d_3 Q[0] =$	00	00.	00	27	
$W[1] =$	00	$\overline{21}$.	$\overline{42}$	$\overline{30}$	$q_1 = \overline{22}$
$rW[1] =$	$\overline{21}$	$\overline{42}$.	$\overline{30}$	00	
$-q_1 D[1]r =$	22	04.	24	36	
$n_4 r^{-2} =$				72	
$-d_4 Q[1] =$	00	00.	00	00	
$W[2] =$	00	41.	75	30	$q_2 = 42$
$rW[2] =$	41	75.	30	00	
$-q_2 D[2]r =$	42	10.	12	16	
$n_5 r^{-2} =$				00	
$-d_5 Q[2] =$	00	00.	00	00	
$W[3] =$	00	$\overline{12}$.	$\overline{62}$	$\overline{16}$	$q_3 = \overline{13}$
$rW[3] =$	$\overline{12}$	$\overline{62}$.	$\overline{16}$	00	
$-q_3 D[3]r =$	13	02.	50	75	
$n_6 r^{-2} =$				00	
$-d_6 Q[3] =$	00	00.	00	00	
$W[4] =$	00	20.	32	75	$q_4 = 20$
$Q = (1.\overline{22}42\overline{13}20)_8 = (0.56416520)_8$					

Table 5.1: Example of high-radix on-line division using pre-scaled operands

the reciprocal is computed in short precision, the best solution would be the use of the host FP unit. We explore this option in the next section. A dedicated circuit for this purpose would consume space in the reconfigurable hardware (if the circuit is kept during the operation) or force a reconfiguration of the coprocessor during the operation. The circuit would be designed for small radix that would be slower than the FP unit in the processor. In both cases the performance would suffer.

5.3.1 Computation of the Scaling Factor at the Host

Using the maximum value of ϵ we compute a more restrictive bounds for the scaled divisor: $1 - \alpha' \leq d \leq 1 + \alpha'$, with:

$$\alpha' = \frac{1}{2r} - \frac{2}{r^2} \tag{5.27}$$

based on the observation that:

$$\alpha = \frac{1}{2r} - \epsilon \quad \text{and} \quad \epsilon = \frac{r+1}{r^2(r-1)} + r^{-4} < \frac{2}{r^2} = \epsilon_{max} \tag{5.28}$$

for $r \geq 4$. The equation already includes the on-line delay of 3. The same result would be valid for $\delta > 3$.

The bit patterns for the upper and lower limits of the interval are shown in Figure 5.3. The vertical bars separate bits from different radix-r digits.

As shown in the figure, the scaled divisor must have the first fractional digit equals to 0 or $r-1$. The second fractional digit is $0 \leq d_2 \leq \frac{r-1}{2}$ for the case $1 + \alpha'$ or $0 \leq d_2 \leq (\frac{r}{2} + 3)$ for the case $1 - \alpha'$.

Consider the original divisor to be y. The host has an FP unit that is capable of computing a reciprocal approximation of y with a precision of k coprocessor digits. If $k \geq 2$, the reciprocal of $\hat{y}_k = M$ has 2 or more digits, and the multiplication of the scaling factor by y will generate a value close to 1, with an error caused by the limited precision of $\hat{y}_k$.

We know show that the truncation error introduced to obtain $\hat{y}$ does not cause the final scaled divisor to fall outside the expected bounds.

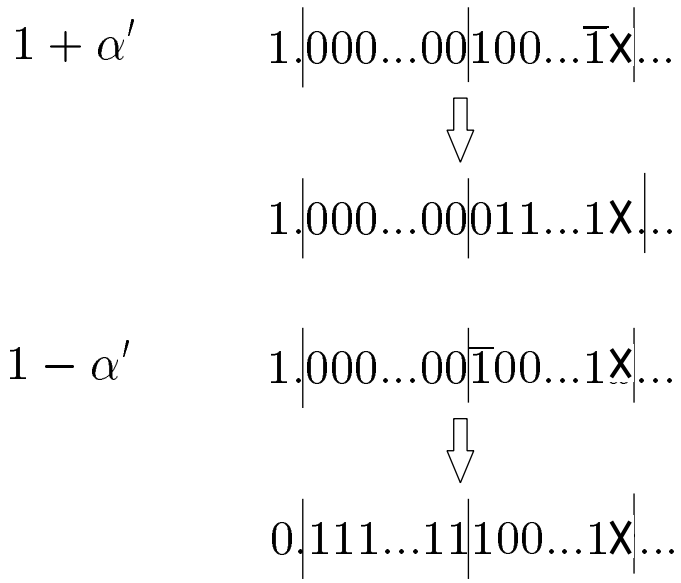

Figure 5.3: Divisor bounds

The approximation $\hat{y}_k$ of the divisor y has an error of r^{-k}. Assuming that y is a normalized number, $0.5 \leq y < 1$, the maximum difference between the obtained scaled divisor and 1 is:

$$yM - 1 \leq (\hat{y} + r^{-k})M - 1 \tag{5.29}$$

Since the scaling factor comes from a limited precision computation (k digits), the product $\hat{y}M$ also has an error, and corresponds to $(1 + r^{-k})$, thus:

$$\hat{y}M - 1 \leq \hat{y}M + Mr^{-k} - 1 = 1 + r^{-k} + Mr^{-k} - 1 \leq 3r^{-k} \tag{5.30}$$

Assuming the minimum value of $k = 2$, this error will affect the 2 least significant bits of the second fractional digit of the scaled divisor. Therefore, the scaled divisor will be in the desired range for digit radices above 16.

In conclusion, using at least 2 digits of the divisor y we obtain the scaling factor M in the host FP unit by calculating the reciprocal of the truncated divisor $\hat{y}$. The value d obtained from the multiplication of M and y is guaranteed to satisfy the the bounds: $\alpha < \alpha' \leq d \leq \alpha' < \alpha$, for proper selection of the quotient digit by

rouding.

Observe that the number of bits used in the coprocessor digit must be less than the number of bits in a host processor word, such that at least two coprocessor digits can fit in one host processor word.

5.4 Pre-scaling of Operands

During the pre-scaling phase, both dividend and divisor must be multiplied by the scaling factor M. This task can be acomplished by an on-line pre-scaler hardware or by software at the host level. In the next section we present the on-line pre-scaler.

5.4.1 On-line Pre-scaler

The on-line pre-scaler is shown in Figure 5.4. Since the original divisor is normalized, the scaling factor is in the range $1 < M \leq 2$. For $M = 2$, the host can perform the pre-scaling operation, that consists of one bit left shift of each operand. Other values of M are going to have an integer bit (b) and two fractional digits (d_1 and d_2). The on-line circuit to compute this operation contains two digit-by-vector multipliers and two adders. One of the registers shown in the figure is controlled by the most significant bit of the scaling factor (b). If b is one, the input digits are delayed and passed to the last adder, otherwise the register controlled by b is kept with a zero value. The delay depends on the degree of pipelining in the pre-scaler network.

The structure is similar to the same as the data path circuit shown in previous sections. The basic differences are some of the connections to the memory components and the delay block of $3 + p'$. These differences force the reconfiguration of the FPGA for each pre-scaling phase.

Another alternative is to use the same data path used for VLP division plus some extra circuitry. This solution is shown in Figure 5.5. The extra circuits are

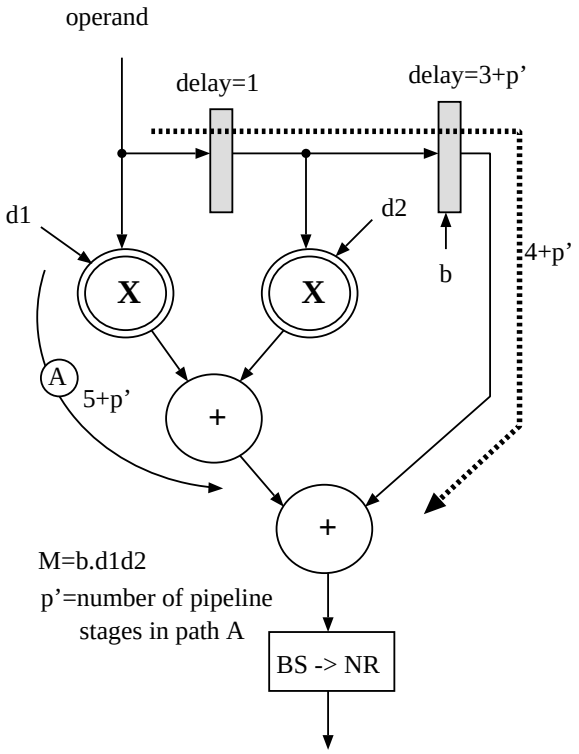

Figure 5.4: On-line pre-scaler

shown inside dashed boxes. The pre-scaling is done in 2 phases, assuming that both operands are placed into the same vector D:

1. The control circuit passes the digits through the data path with $rega = 1$, registers c, d, e and f cleared. A copy of the operand is this way stored into the residual vector.

2. load $rega = d_1$ and $regc = d_2$. The operand is inserted again into the data path. If $b = 1$ (most significant bit of the scaling factor) the copy of the operand in the residual vector is also used as input, with adequate delay. The network output is transferred to the proper output vector (D or N). During this transfer, the digits are converted from BS to NR representation.

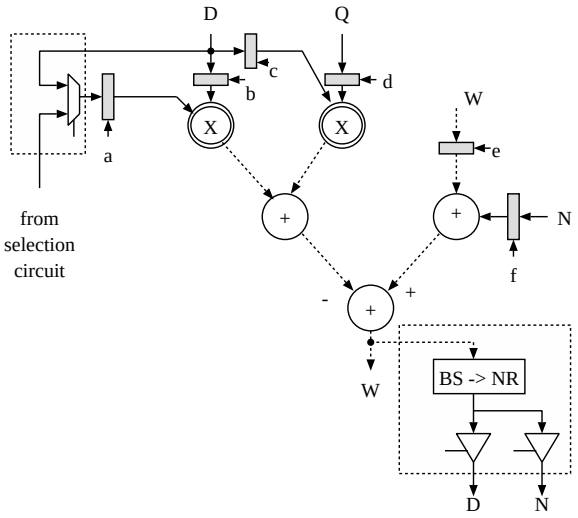

Figure 5.5: Pre-scaling using the data path of the VLP division

With an efficient implementation, the circuit will take $3(7+p+m-1)$ cycles per operand, where p is the level of pipelining and m is the precision of the operands.

5.5 Selection Circuit

The most significant digits of the residual that come out of the data path are stored inside the selection circuit in a small append register. The stored value is converted from BS code to non-redundant two's complement form before the rounding process takes place. A block diagram of the selection circuit is shown in the Figure 5.6.

For proper selection, only one fractional digit is required. The digit conversion stage has the capacity for 2 digits in high-radix r (one integer and one fractional digit) plus one bit, and is responsible to convert the redundant digit output of the datapath (BS code) into a non-redundant representation. The rounding takes place after the conversion, considering only the integer digits of the non-redundat residual and one fractional bit. Since the non-redundant representation is two's complement, the addition of 0.5 followed by truncation produces the desired quo-

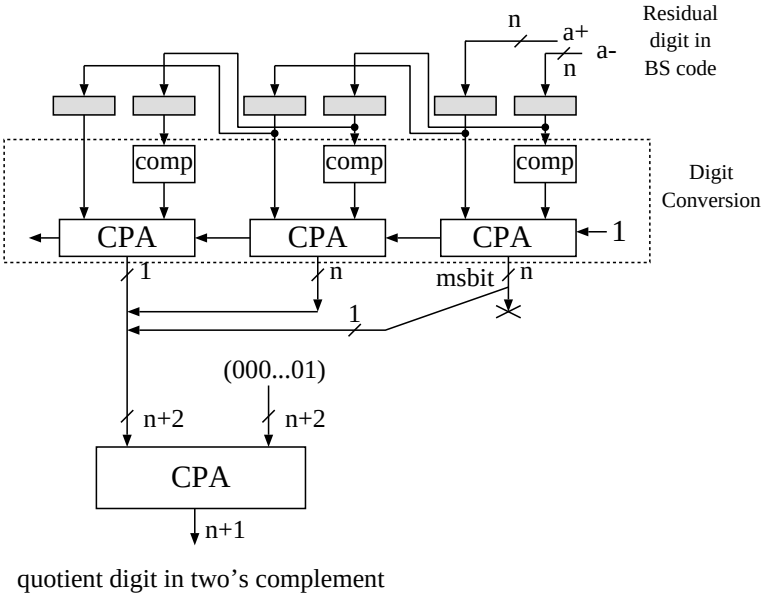

Figure 5.6: Selection circuit for VLP division

tient digit in non-redundant format.

The most significant digit of the residual (leftmost CPA) must be zero, thus, only one bit is used to keep the sign of the non-redundant residual, i.e., $n+1$ bits of the output are used as the quotient digit.

The selection time may force the data path to stall when the precision of the operands is small (beginning of the operation). Since the three digits are stored into the unit, the time to compute the quotient digit corresponds to:

$$T_{sel} = \frac{T_{CPA}(2n+1) + T_{CPA}(n+2)}{T_{cp}}$$

where $n = log_2(r)$, and T_{cp} is the coprocessor cycle time.

The complementation block (COMP) will be absorbed by the CPA modules in a FPGA synthesis process. A more elaborate estimate of the selection function time is given in Chapter 8, when we discuss the design of arithmetic operators for FPGAs. The influence of the selection function in the VLP division time is given in section 5.8.

5.6 Reducing the Number of Cycles

Based on the same idea considered for the VLP multiplication, the computation of the recurrence equation does not need to be done in full precision of the residual all the time. The algorithm may be modified to reduce the precision of the data vectors in one digit per step, after a certain step j. Assume that the first truncation of the data vectors occurs at position t of the input vector D. We consider in this analysis only the vector D because it is the one that has more weight in the recurrence equation. The same truncation in vector Q would result in a smaller error, since this vector is multiplied by $r^{-\delta+1}$.

Disregarding digits from D after the digit at position t results in an error ϵ in the scaled residual that is bounded as:

$$\epsilon \leq |r(r-1)r^{-t}|$$

One more iteration will result in an error that is composed of the residual error plus the divisor error (that now has one less digit):

$$r(r-1)r^{-t} + (r-1)r^{-t+1} \rightarrow (scaled) \rightarrow 2r^2(r-1)r^{-t}$$

and for q iterations the equation for the error is:

$$\epsilon_q \leq |qr^q(r-1)r^{-t}| \tag{5.31}$$

For correct selection of the output digit, the error inserted by the limited precision computation of the recurrence equation must be less than r^{-1}, since only one fractional digit is used in the truncated residual for output selection. Thus:

$$qr^q(r-1)r^{-t} < r^{-1} \tag{5.32}$$

In VLP division, the number of iterations executed after digit t is processed, for a final precision of m digits is $q = m - t + \delta - 1$.

The value q is bounded as

$$q \leq \frac{m}{2} \tag{5.33}$$

Using equation (5.33) in (5.32), we obtain:

$$\frac{m}{2} r^{\frac{m}{2}} (r-1) r^{-t} < r^{-1} \tag{5.34}$$

$$r^{t} \geq \frac{m}{2} r^{\frac{m}{2}+2} \tag{5.35}$$

$$t \geq log_r(\frac{m}{2}) + \lceil \frac{m}{2} \rceil + 2 \tag{5.36}$$

As the term $log_r(\frac{m}{2}) < 2$ is true for a large range of values, as explained for VLP multiplication, we assume:

$$t = \lceil \frac{m}{2} \rceil + 4 \tag{5.37}$$

Four digits after the middle of the output vector, the precision of the recurrence equation can start to decrease, without compromising the final result of the computation.

5.7 Pipelined Operation

The effect of pipelining in the data path for VLP division impacts the performance in two ways. The increase in the pipeline level reduces the cycle time which, for a long sequence of digits, results in less total time. However, the increase in the number of pipeline stages also increases the latency. As the selection function is executed only after the first residual digits are generated, delaying the generation of these digits will also delay the generation of the next quotient digit. While the precision of the residual is small, the generation of the quotient digit will suspend the beginning of the next iteration.

The delay of the data path is $7 + p$, where p is the degree of pipelining in the implementation. The minimum number of cycles in the data path is 7. Considering the overlap of iterations, and looking at the input sequences, the data path receives k input digits and a bubble of 5 cycles (minimum), before the next iteration. If the quotient digit is not ready after $k + 5$ cycles, the new input digits cannot be applied. The quotient digit is generated in cycle $7 + p + 2 + T_{sel}$, related to the

cycle when the iteration begins. A new iteration cannot begin before the quotient digit of the present iteration is generated, that results in $k+5 \leq 9+p+T_{sel}$. The number of cycles per iteration, as a function of the number of digits in the input vectors (k) is given as:

$$cycles(k) = \begin{cases} 9+p+T_{sel} & \text{if} \quad k \leq 4+p+T_{sel} \\ k+5 & \text{otherwise} \end{cases} \tag{5.38}$$

The value of the T_{sel} is discussed in section 5.5.

In a non-overlapped mode of operation, the constrain is $k+7+p \leq 9+p+T_{sel}$, that implies the equation:

$$cyclesn(k) = \begin{cases} 9+p+T_{sel} & \text{if} \quad k \leq 2+T_{sel} \\ k+7+p & \text{otherwise} \end{cases} \tag{5.39}$$

The extra cycles imposed by the selection function may be seem as an overhead over the number of cycles of the VLP division without selection function. This number of extra cycles are not easily obtained since VLP division starts to reduce the precision of intermediate calculations after t operand digits. Thus, the selection function is going to affect the beginning and the end of the VLP division operation. An approximation of the overhead caused by the selection may be obtained as twice the extra cycles when the working precision increases:

$$T_{ovh} = 2(\sum_{k=\delta}^{\beta} 9+p+T_{sel}-(k+5)) \tag{5.40}$$

where $\beta = min(4+p+T_{sel}, t)$, with t representing the input truncation point, makes the summation items always positive or zero. The same overhead for the case of non-overlapped operation is given as:

$$T_{ovhn} = 2(\sum_{k=\delta}^{\gamma} 2+T_{sel}-k) \tag{5.41}$$

where $\gamma = min(2+T_{sel}, t)$.

These equations show that the non-overlapped mode of operation reduces the impact of the selection function in the total time of the operation because it takes longer to complete each iteration.

5.8 Execution Time

This section presents the execution time of the VLP division for two operands with m digits of precision, generating a quotient of m digits of precision. The degree of pipelining in the data path is represented by p.

Due to pre-scaling, the dividend and divisor may have one integer digit. We consider this worst case situation in the following equations. Pre-scaling itself will take:

$$T_{pre}(m,p) = 3(7+p+m-1) \quad \text{cycles} \tag{5.42}$$

for each input operand and $7+p$ is the data path delay in cycles, as explained in Chapter 4.

The overhead imposed by the selection function in a pipelined data path is given in equation (5.40) as T_{ovh} orT_{ovhn} . Each iteration with k fractional digits will take k+6 cycles (one extra integer digit included) in a pipelined and overlapped operation, until digit $d = \lceil \frac{m}{2} \rceil + 4$. After fractional digit d, the precision of the data vectors used in the serial computation decrease one digit per iteration, as explained in section 5.6, and the number of cycles for each iteration involving the new fractional digit ($k > d$) is $(2d-k+1)+1+5$ cycles (composed by the number cycles needed for k fractional digits in the iteration, one integer digit and a bubble of 5 cycles).

Putting it all together we obtain the expression for the number of cycles for VLP division as:

$$T_{VLPdiv} = 2T_{pre}(m,p) + T_{ovhn} + \sum_{j=3}^{d}(j+6) + \sum_{j=d+1}^{m+2}(2d-j+7) + (7+p-5) \tag{5.43}$$

Without overlap of iterations it is necessary to insert the operand digits and wait for the data to go through the data path, before the next begins. The number of cycles for VLP division in this case, can be approximated by the expression:

$$T_{VLPdiv} = 2T_{pre}(m+p) + T_{ovh} + \sum_{j=3}^{d}(j+7+p) + \sum_{j=d+1}^{m+2}(2d-j+1+7+p) \tag{5.44}$$

The basic difference between the VLP division time and VLP multiplication time is the pre-scaling of operands and the extra cycles when the data path does not have the output of the selection function available (during the first iterations).

The cycle time to execute one iteration with input operands in precision i digits after the point when the selection function is affecting the beginning of the next iteration is given as:

$$T_{iter}(i) = i + 7 + p \text{ [cycles]} \tag{5.45}$$

Chapter 6

VLP Square Root

This chapter presents the design aspects of the VLP square root operation. The recurrence equation for on-line square-root computation $Y = \sqrt{X}$ is:

$$W[j] = r(W[j-1] - y_{j-1}(Y[j-1] + Y[j-2])) + x_{j+\delta-1}r^{-\delta+1} \tag{6.1}$$

or

$$W[j] = r(W[j-1] - y_{j-1}(2Y[j-2] + y_{j-1}r^{-j+1})) + x_{j+\delta-1}r^{-\delta+1} \tag{6.2}$$

where

$$X[j] = \sum_{i=0}^{j+\delta-1} n_i r^{-i} \tag{6.3}$$

$$Y[j] = \sum_{i=0}^{j} q_i r^{-i} \tag{6.4}$$

$$W[j] = \sum_{i=0}^{j} w_i r^{-i} \tag{6.5}$$

and with the initial condition of $W[0] = X[0]$. The on-line delay for high-radix square root computation is 3 cycles, based on [Tu90].

Lets compare the pros and cons of using one or another equation. If the main selection factor is to have a data path similar to the VLP multiplication and division, the first recurrence equation (6.1) is more adequate. The implementation of this recurrence equation will have both digit by vector multipliers doing the same task most of the time (multiplication of y_{j-1} and $Y[j]$). The other equation (6.2) leads to an implementation which has a data path where one of the digit by vector (DV) multipliers is used as a digit by digit (DD) multiplier only. The difference in area between the two implementations is small (only an on-line adder

and some registers), since the DD multiplier consumes most of the area in the DV multiplier. The control circuit would be more complex for the second approach, since the correct time to insert the value y_{j-1}^2 in the network would change from one iteration to the other, and the insertion would need to be done for only 2 or 3 cycles (digit product in redundant representation). The rest of the time, the circuit is not used at all. The shift left operation ($2Y[j-2]$) would also force a small increase in the circuit area of the DV multiplier in the implementation of the second equation.

For these reasons we are going to use recurrence equation 6.1 as the basic equation for our VLP square root implementation.

6.1 VLP Square-Root Algorithm

In this section we describe the algorithm to compute the VLP square root ($Y = \sqrt{X}$) of a LP number, with $0.25 \leq X < 1$ with m digits. The serial computation of the recurrence equation is performed by the data path shown in Figure 6.1.

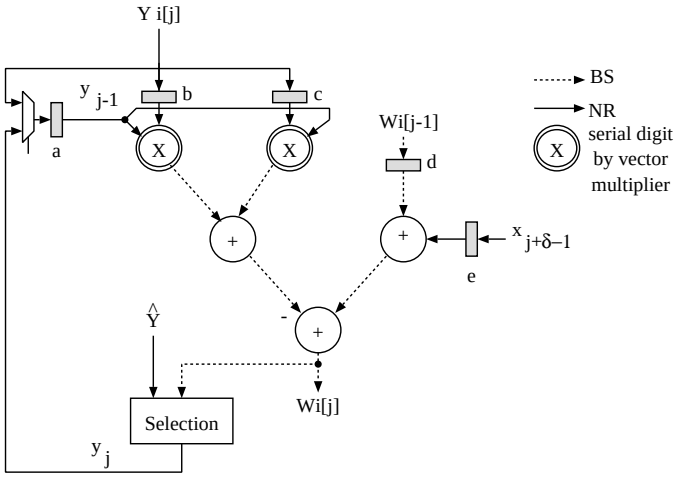

Figure 6.1: Data path for VLP on-line square-root operation

The same type of modules used in VLP division are used in this data path. The difference between this circuit and the other is the interconnection of the DV multpliers and the memory elements, the selection function and a multiplexer for register (a).

The data vectors used are:

- X is the input operand digit vector and contains one integer digit and m fractional digits. The value of X corresponds to $\sum_{i=0}^{m} x_i r^{-i}$ and the vector position i contains digit x_i, starting from position $i = 0$.
- $Y[j]$ contains all output digits generated by the algorithm, including the digit selected in step j.
- W is the scaled residual digit vector at step j.

The VLP square root algorithm to compute a number in precision m, using the on-line recurrence equation is described in Algorithm 7.

The selection function $\mathcal{S}el$ is described in the next section. Other values used in the algorithm, such as the number of fractional digits in the output estimate $\hat{Y}$, will become clear later in this chapter.

The data vectors manipulated by the algorithm are shown in figure 6.2. The big dots represent the fractional points in the digit vectors. The pointer xp references the input operand digit being considered at step j. Pointer *opt* shows the position of the last output digit. After each iteration a new digit is generated and inserted in Y, followed by an increment of *opt*. The distance between xp and *opt* correspond to the on-line delay.

In order to simplify the manipulation of the long-precision digit vectors, the alignment of values is performed shifting the digit vectors relatively to each other. The amount of the shift depends on the network and recurrence equation. The same pointer is used to read digit vectors W and Y.

The delays of the datapath are shown in Figure 6.3. Both Y and W, based on the recurrence equation, should be aligned. But, as the paths involving Y and W

Algorithm 7 *: VLP square root*

1. Initialization:

- *transfer the first δ most significant digits of X to W, making the proper alignment in the digit vector:*
  $W_i[0] = x_i,\ i = 0, \ldots, \delta - 1$
  $W_i[0] = 0,\ i > \delta - 1$
- *initialize pointers to data vectors: $xp = \delta$ and $opt = 0$.*
- *copy to vector Y the estimate of the output $\hat{Y} = (\hat{y}_0, \hat{y}_1, \hat{y}_2)$ that was provided by the host (this step may be avoided if the host is able to access the memory element that stores Y directly) forcing $Y[2] = \hat{Y}$.*
- *generate the initial residual based on the output estimate ($\hat{Y} = (\hat{y}_0, \hat{y}_1, \hat{y}_2)$)*
  for j=*1* to 3
  compute serially the recurrence equation:
  $W[j] = rW[j-1] + x_{j+\delta-1} r^{-\delta+1} - r\hat{y}_{j-1} Y[j-2] - r\hat{y}_{j-1} Y[j-1]$
  where t is the number of fractional digits in the output estimate $\hat{Y}$.
- *select the output digit for the first time:*
  $y_3 = \mathcal{S}el(\hat{W}[3], \hat{Y})$
- *update Y vector: $Y[3] \leftarrow APPEND(Y[2], y_3)$*

2. Iteration: *for $j = 4$ to m*

(a) compute serially the digits of the next residual as:
$W[j] = rW[j-1] + x_{j+\delta-1} r^{-\delta+1} - ry_{j-1} Y[j-2] - ry_{j-1} Y[j-1]$

(b) Select the output digit
$y_j = \mathcal{S}el(\hat{W}[j], \hat{Y})$

(c) update quotiend vector $Y[j] \leftarrow APPEND(Y[j-1], y_j)$

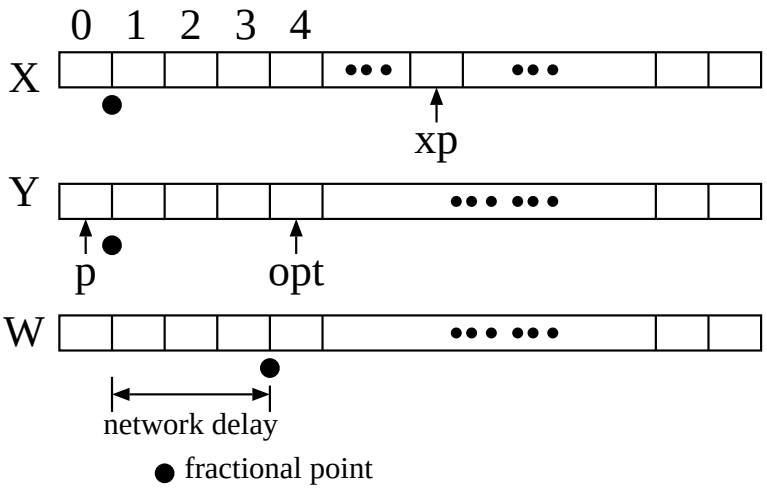

Figure 6.2: Digit vector for VLP Square Root

have a difference of 3 cycles, W must be shifted 3 digits related to Y, as shown in Figure 6.2.

The pseudo-code for the serial computation of the recurrence equation is shown in Algorithm 8.

We now analyze the conditions to have selection function using rounding. Contrary to the VLP division, the square root imposes more constraints to the application of this method.

6.2 Convergence conditions for Output Selection

This section shows the derivation of the bounds for the operand to allow rounding as the selection function. Constrasting with the division recurrence equation, square-root equation corrects the residual value by multiplying the partial result of the operation by the new output digit. More than that, as the operation output is used in the recurrence equation, the leading digits must be computed by another circuit, or obtained from table look up. The option of table lookup is not considered in this evaluation, since the table for high radix is too large. We consider

Algorithm 8 *: serial computation of the residual*

1. *initialization*
   - $p = 0$
   - *store* $Y[opt]$ *into register* a
   - *set the limit for the next loop. It is necessary when the precision of input* Y *is less than the precision of the residual* W.
     if $(opt < \delta)$ $limit = \delta + 7$ *else* $limit = opt + 7$
     (comment: 7 is the delay of the non-pipelined data path network)
2. *loop while* $p \leq limit$ *do:*
   - *if* $p > opt$ *clear register* b*, otherwise read* $Y[p]$ *and store into register* b
   - *if* $p \geq opt$ *clear register* c*, otherwise read* $Y[p]$ *and store into register* c
   - *read* $W[p]$ *from memory and store into register* d
   - *if* $p = \delta + 3$ *store* $X[xp]$ *into register* e*, otherwise clear register* e
   - *issue control signals to the selection function to store the first most significant digits of the residual (comment: the selection function circuit is defined in section 6.2.2)*
   - *if* $p \geq 5$ *write the output of the data path to* $W[p-5]$ *(comment: the number 5 comes from the delay of the path between* $W_i[j-1]$ *and* $W_i[j]$ *and the scaling of the residual).*
   - *increment* p
3. *update pointers to the vectors:* $xp = xp + 1$ *and* $opt = opt + 1$.

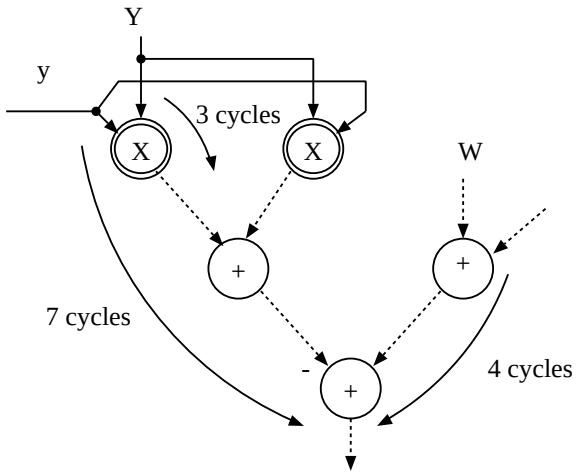

Figure 6.3: Data path delays

the case when the host FP arithmetic unit is used to compute a short precision estimate of the output, and provides the digits obtained in this process to start up the computation of the VLP square root.

One way to attack the problem is to consider an scaling factor the same way as we did for division. In this case the output Y may be close to 0.5, or more specifically in the range $0.5 - \alpha < Y < 0.5 + \alpha$ in order to allow the rounding function for selection of the output digit. This solution requires correction of the result. With X as the input, and a scaling factor of M, the result of the square root operation is $\sqrt{MX} = \sqrt{M}\sqrt{X}$, and it must be multiplied by $\frac{1}{\sqrt{M}}$. This correction factor is a long-precision number, which forces the application of other long-precision computations, including a long-precision square-root. For this reason, this approach is not further analyzed in this thesis.

Another idea is to use a modified selection function, that is not based on the residual value only, but in a combination of the residual and the output. In this case, a larger interval of input/output values may be considered and it is not necessary to scale the input operand or correct the output. This approach is discussed in the next section.

6.2.1 Selection function with Compensated Residual

The pre-scaling of the input operand causes many problems. The solution proposed and studied in this section uses a more elaborated selection function based on the residual value and an estimate of the output, as follows:

$$y_{j-1} = \mathcal{S}el = \left\lfloor \frac{\hat{W}[j-1]}{2\hat{Y}} + \frac{1}{2} \right\rfloor \tag{6.6}$$

where $\hat{W}$ is an limited precision value of the residual and $\hat{Y}$ is an approximation of the value $\sqrt{X}$ computed by the host, based on the most significant digits of the input operand X. In [Mat91] it was proposed a high-radix square-root unit that would use the same selection method, but truncation instead of rounding. This work was cited in [LM92].

The conditions assumed are:

- $0.25 \leq X < 1$, that results in $0.5 \leq Y < 1$.
- the host is capable of handling k coprocessor digits;
- the host generates a rounded limited precision estimate of the output based on limited precision input values, up to t fractional digits.

First, observe that using the proposed selection function the output digit set may not be maximally redundant sometimes. It depends on the value of the output Y. If we consider $|W| < r$, for example, the selection function reduces to:

$$y_{j-1} = \mathcal{S}el \leq \left\lfloor \frac{r}{2\hat{Y}} + \frac{1}{2} \right\rfloor \tag{6.7}$$

Based on this equation the digit set for y_j changes depending on $\hat{Y}$. For example, if $Y = 0.98$ and $r = 10$, the possible values of y_j are in the set $\{-6, -5, \ldots, -1, 0, 1, \ldots, 5, 6\}$. For Y close to 0.5 the maximally redundant digit set is possible. We assume that the host generates an approximation of the output with t fractional digits (rounded), thus $k \geq t$ digits. The utilization of tables would require a large space

in memory. On the other hand, this assumption limits the width of the coprocessor word to a fraction of the host processor word.

The error in the truncated low-precision value $\hat{X}$ is given by:

$$|X - \hat{X}| < \frac{r^{-t}}{2} \tag{6.8}$$

The estimate of the output is computed based on $\hat{X}$. We need to determine the error in the estimate, defines as $|Y - \hat{Y}|$. The relative error of $\hat{X}$ when the truncation is done at the t^{th} fractional digit is specified as:

$$\delta_x = \frac{r^{-t}}{X} \tag{6.9}$$

As $0.25 \leq X < 1$ the error has the following upper bound:

$$\delta_x < 4r^{-t} \tag{6.10}$$

and from this equation we obtain:

$$\hat{X} = X(1 + \delta_x) \tag{6.11}$$

and thus

$$\hat{Y} = \sqrt{\hat{X}} = \sqrt{\hat{X}}.\sqrt{1 + \delta_x} = Y\sqrt{1 + \delta_x} \tag{6.12}$$

Using the property that $\sqrt{1 + \alpha} < 1 + \sqrt{\alpha}$ for $\alpha < 1$, we obtain the equation:

$$\sqrt{1 + \delta_x} < \sqrt{1 + 4r^{-t}} < 1 + 2r^{-t/2} \tag{6.13}$$

thus

$$\hat{Y} = Y(1 + 2r^{-t/2}) \tag{6.14}$$

and the relative error is computed as:

$$\delta_y = 2r^{-t/2} \tag{6.15}$$

as $0.5 \leq Y < 1$, the maximum absolute error is

$$|Y - \hat{Y}| < r^{-t/2} \tag{6.16}$$

We define the variable γ as:

$$\gamma = \frac{W[j-1]}{2Y} - y_{j-1} \tag{6.17}$$

Based on the selection function (equation (6.6)) that considers a truncted value of W and the error incurred in the use of $\hat{Y}$ instead of the real Y we obtain the bounds on γ as:

$$|\gamma| < \frac{1}{2} + r^{-t} + r^{-t/2} \tag{6.18}$$

where the value r^{-t} corresponds to the residual truncation error and $r^{-t/2}$ is the error in $\hat{Y}$.

To obtain the value of W as a function of γ we combine equations (6.1) and (6.17) using the maximum values of the possible terms in the equations. The result is:

$$w \leq 2rY\gamma + (r-1)r^{-\delta+1} \tag{6.19}$$

where w is the value of the residual and Y is the output value.

For convergence of the recurrence equation (6.1) it is necessary that:

$$r(w - 2\rho Y) + (r-1)r^{-\delta+1} \leq w \tag{6.20}$$

where ρ is the maximum value of digit y_{j-1}, such that $y_{j-1} \in \{-\rho, ..., \rho\}$. The minus sign in the equation was used because the output Y is always positive and the output digit has the same sign of the residual. We also considered that the maximum input digit value is $(r-1)$. From equation (6.20):

$$w \leq \frac{2r\rho Y - (r-1)r^{-\delta+1}}{r-1} \tag{6.21}$$

To have the output in a non-redundant digit set we must impose:

$$\left|\frac{W}{2\hat{Y}}\right| < r - \frac{1}{2} - r^{-t} \tag{6.22}$$

From this equation, combined with equation 6.16 we obtain:

$$|W| < 2(Y - r^{-t/2})(r - \frac{1}{2} - r^{-t}) \tag{6.23}$$

Equations (6.21) and (6.23) give the upper bound on the residual value as:

$$W < min(2(Y + r^{-t/2})(r - \frac{1}{2} - r^{-t}), \frac{2r\rho Y - (r-1)r^{-\delta+1}}{r-1}) \quad (6.24)$$

Combining equation (6.19) and (6.24) we get:

$$2rY\gamma + (r-1)r^{-\delta+1} < min(2(Y - r^{-t/2})(r - \frac{1}{2} - r^{-t}), \frac{2r\rho Y - (r-1)r^{-\delta+1}}{r-1}) \quad (6.25)$$

As these functions are continuous and monotonic in the interval of interest, lets analyze the condition for the extreme values only and test the condition for the values $\delta = 3$ and $t = 2$. The value $t = 2$ is used because the VLP division algorithm already limited a minimum of 2 coprocessor digits per host word.

- $Y = 0.5$: in this case $\rho = r - 1$, and the condition reduces to:

$$r(\frac{1}{2} + r^{-2} + r^{-1}) + (r-1)r^{-2} < min((1 - 2r^{-1})(r - \frac{1}{2} - r^{-2}), r - (r-1)r^{-2}) \quad (6.26)$$

- $Y = 1$: in this case $\rho = \frac{r}{2} + 1$:

$$2r(\frac{1}{2} + r^{-2} + r^{-1}) + (r-1)r^{-2} < min(2(1 - r^{-1})(r - \frac{1}{2} - r^{-2}), \frac{2r(\frac{r}{2} + 1) - (r-1)r^{-2}}{r-1}) \quad (6.27)$$

These conditions are satisfied for values of $r > 8$.

An example of the VLP square root computation using the proposed selection function is shown in Table 6.1. The output digits generated at steps 1 and 2 are obtained from the estimate of the output value, provided by the host. The selection function is applied in all other steps to obtain y_j.

6.2.2 Selection Circuit

The circuit used by the selection function of the VLP square root is shown in Figure 6.4. The selection function requires the multiplication of a truncated residual by the reciprocal of the output estimate $\hat{Y}$. The reciprocal $(\hat{Y})^{-1}$ is a value in the range $(1, 2]$ and has 2 fractional digits, a total of $(2n + 2)$ bits. The value

$X = 0.732198$ in radix 10
$\hat{X} = 0.73$ (truncated value of X, two digits)
$\hat{Y} = \sqrt{\hat{X}} = 0.85$ (approximation of the result Y)
$\frac{1}{\sqrt{\hat{X}}} = 1.18$
$\delta = 3$ (on-line delay)
$W[0] = 0.73$

j	x	$W[j]$	$\frac{W}{2\hat{Y}}$	y_j
0		$W[0] = 0.73$	0.43	0
1	2	$W[1] = 10(0.73) + 0.02 = 7.32$	4.31	8 (from $\sqrt{\hat{X}}$)
2	1	$W[2] = 10(7.32 - 8 \times 0.8) + 0.01 = 9.21$	5.42	5 (from $\sqrt{\hat{X}}$)
3	9	$W[3] = 10(9.21 - 5(0.85 + 0.8)) + 0.09 = 9.69$	5.71	6
4	8	$W[4] = 10(9.69 - 6(0.856 + 0.85)) + 0.08 = -5.38$	-3.17	-3
5	0	$W[5] = 10(-5.38 - (-3)(0.8557 + 0.856)) = -2.449$	-1.43	-1
6	0	$W[6] = 10(-2.449 + (0.85569 + 0.8557)) = -7.3761$	-4.34	-4

$Y = 0.856\overline{314} = 0.855686$
$expected = 0.855686$

Table 6.1: Example of VLP Square Root in radix 10

of $(\hat{Y})^{-1}$ is represented in conventional number system. The multiplier and CPA modules perform the scaling of the residual value and rounding. The division by 2 is obtained by proper interconnection between the multiplier and CPA (represented by an right arrow crossing the multiplier output represented in the Figure). The rounding circuit could be incorporated to the CPA stage of the multiplier (internally) reducing the area and total delay, but for our estimates we consider them as separate modules.

Since on-line modules are used in the serial computation of the recurrence equation, the most significant digits of the residual are obtained first, and while the remaining digits of the residual are computed, the selection function works on the generation of the next output digit. The timing diagram in Figure 6.5 shows the time used by an hypothetical selection function that consumes 4 cycles, and the serial computation of the recurrence equation. Observe that the selection function stalls the operation of the coprocessor when the precision of the operands is low. It will not be in the critical path after the first most significant digits of

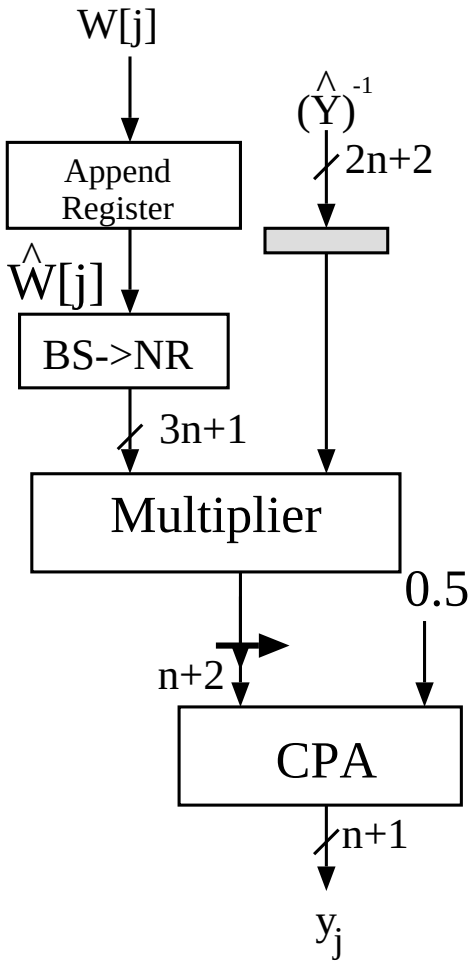

Figure 6.4: Circuit used for output selection in VLP Square Root

the output are generated, since the latency to generate each digit increases one cycle per iteration. In the Figure, after iteration 3, the selection function does not limit the speed of the datapath. This feature allows the utilization of cheaper and slower components in the implementation of the selection function circuit, such as: a parallel serial multiplication module and Carry Propagate Adders.

The same problems with the level of pipelining and initial operation of the unit that happened in VLP division are going to be more pronunciated in VLP square root, since the selection function for VLP square root is more complex. However,

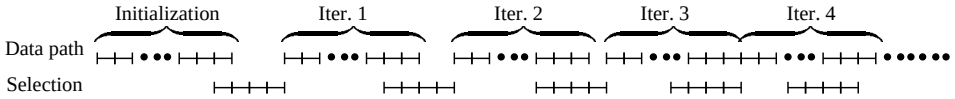

Figure 6.5: Timing of selection function and data path

equation (5.40) used to computer the overhead imposed by the selection function in a pipelined data path is also valid for VLP square root.

The time for selection depends on the selection of components and technology. An estimate of the circuit delay for FPGAs is shown in section 8.5.2.

6.3 Performance Evaluation

6.3.1 Optimization of the Number of Cycles

For an output of precision m digits, it is not necessary to compute the recurrence equation in full precision all the time. If we apply the same idea presented for the VLP multiplier, we may work with a precision for vectors Y and W that is reduced in one digit in each step after a certain step n. Observe that the reduction of Y's precision is only for residual calculation. The precision of Y continues to increase in each new iteration, but not all digits are read in each iteration. Assume that the first truncation of vector Y is done at digit position k. This action inserts an error ϵ in the scaled residual that is bounded as:

$$\epsilon \leq |2r(r-1)r^{-k}|$$

One more iteration will result in an error that combines the error in the short precision Y value (one less digit) and the error in the previous residual (ϵ):

$$\epsilon + 2(r-1)r^{-k+1} \rightarrow (scaled) \rightarrow 4r^2(r-1)r^{-k}$$

and for q iterations the equation for the error is:

$$\epsilon_q \leq |2qr^q(r-1)r^{-k}| \tag{6.28}$$

For correct selection of the output digit, the error inserted by the limited precision computation of the recurrence equation must be less than r^{-t}, where t is the number of fractional digits in the truncated residual used in the selection function. Thus:

$$2qr^{q}(r-1)r^{-k} < r^{-t} \tag{6.29}$$

In VLP square root calculation, the number of iterations executed after input digit k was processed, for a final precision of m is $q = m - k + \delta - 1$. However, a simpler upper bound for q is:

$$q \leq \frac{m}{2} \tag{6.30}$$

thus:

$$2\frac{m}{2}r^{\frac{m}{2}+t}(r-1) < r^{k} \tag{6.31}$$

and using the condition that $t = 2$ digits, we obtain:

$$mr^{\frac{m}{2}+2}(r-1) < mr^{\frac{m}{2}+3} \leq r^{k}$$

$$log_r(m) + (\frac{m}{2} + 3) \leq k$$

As $log_r(m) < 2$ is true for a large range of input precision values (m), based on the same discussion presented for VLP multiplication, we obtain:

$$k \geq \left\lceil \frac{m}{2} \right\rceil + 5 \tag{6.32}$$

Five digits after the middle of the output vector, the precision of the recurrence equation can start to decrease, without compromising the selection function of output digits.

6.3.2 Execution time

During initialization, some digits of X are copied to vector W (Algorithm 7). This task can be done by the host before the data is inserted in the coprocessor,

without consuming coprocessor cycles Also during initialization phase, $t+1$ iterations are performed to generate the initial residual. During this phase δ digits of the residual are necessary (one integer and $\delta-1$ fractional digits). Each iteration takes $\delta+2$ cycles (bubble of 2 cycles in the path of the residual), with a setup time (to transfer digits from Y) of $t_{dp}-t_w$, where t_{dp} is the total delay in the data path and t_w is the delay in the branch used by the scaled residual W. The selection function is activated only in the iteration that issues the last digit of the estimate (iteration that generates y_{t+1}). Total number of cycles during initialization is:

$$T_{init} = (t_{dp} - t_w) + (t+1)(\delta+2) + N + T_{sel} \text{ [cycles]} \tag{6.33}$$

where N considers the number of cycles required to obtain the $t+1$ most significant digits of the residual at iteration $t+1$, after all digits of the previous residual and a bubble of 2 cycles were inserted. Since we are considering the input behavior and the selection function depends on the output, the value of N is inserted to compensate for this change in the point of reference. Thus $N = t_w-(\delta+2)-1+3$ where the value 3 corresponds to the number of digits used in the selection function, and the rest of the equation corresponds to the time to get the first residual digit at the output.

For other iterations to compute output digit y_i, with $t+1 \leq i \leq m$, the pipeline structure and the time of the selection function circuit will increase the total number of cycles in T_{ovh}, as presented in equation (5.40), related to the zero delay selection function case.

The total number of cycles necessary in VLP square root, considering the truncation point in digit $d = \lceil \frac{m}{2} \rceil + 5$ is approximately given by the equation (with $\delta = 3$):

$$T_{VLPsqrt} = T_{init} + T_{ovh} + \sum_{j=3}^{d}(j+6) + \sum_{j=d+1}^{m+2}(2d-j+7) + (7+p) - 5 \tag{6.34}$$

considering one integer digit for the operand, a pipelined data path with delay $7+p$ and an overlapped operation between iterations. In the equation, one integer

digit is considered for the operand and result. This assumption is necessary given the redundant representation that can be used for X and Y.

For a non-overlapped operation the equation is modified to:

$$T_{VLPsqrt} = T_{init} + T_{ovhn} + \sum_{j=3}^{d}(j + 7 + p) + \sum_{j=d+1}^{m+2}(2d - j + 8 + p) \qquad (6.35)$$

Comparing the number of cycles of VLP square root and the previous VLP operations, the VLP square root operation behaves like VLP division, without the need for scaling.

Chapter 7

Implementation Aspects and Host Tasks for VLP Operation

This chapter describes the implementation aspects of the VLP algorithms in the RAC and also present operations that should be implemented at the host level. Regarding the VLP algorithms, we discuss in detail the arithmetic modules that were used in the previous chapters, more specifically, number conversion modules.

Aspects of the implementation associated to host tasks are: floating-point (FP) number format, digit conversion and FP algorithms using the coprocessor VLP operations.

7.1 Digit Code Conversion

All digits used in VLP algorithms are signed-digits (SD) in the maximally redundant set $\{-(r-1), ..., -1, 0, 1, ..., r-1\}$. The SDs are represented in this work using: BS code, non-redundant two's complement (NR) or CS form.

Redundant number system is considered in all internal operations of the network of modules to compute the on-line recurrence equation. The utilization of different codes in different stages of the computation is the key for efficient implementations. We present in this section the converters used in the VLP data path networks presented in the previous chapters.

7.1.1 CS to BS Converter

The output of multipliers is usually in CS code. CS adders are the ones that give the best time and area relation. It is important to have a parallel conversion method from CS to BS code, avoiding the need to assimilate the bits in CS code before the equivalent BS code is obtained. In this section we present the circuit

and proof for a parallel convertion method.

The n-bit input number in CS code is represented by $\underline{x} = (\underline{c}, \underline{s})$, where $\underline{c} = (c_{n-1}, ..., c_1, c_0)$ and $\underline{s} = (s_{n-1}, ..., s_1, s_0)$, with $c_i, s_i \in \{0, 1\}$. Both vectors are in two's complement form. Thus, $c = -c_{n-1}2^{n-1} + \sum_{i=0}^{n-2} c_i 2^i$ and $s = -s_{n-1}2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$. We assume that no overflow is allowed.

Following the procedure described in [EL96] we generate another digit vector v_i adding each digit of the CS representation in parallel.

$$\begin{array}{ccccccc} s_{n-1} & : & s_{n-2} & ... & s_1 & s_0 \\ c_{n-1} & : & c_{n-2} & ... & c_1 & c_0 \\ \hline v_{n-1} & : & v_{n-2} & ... & v_1 & v_0 \end{array}$$

such that $v_i \in \{0, 1, 2\}$, for $i = 0, 1, ..., n-2$ and $v_{n-1} \in \{0, \overline{1}, \overline{2}\}$.

Recoding $v_i = 2p_{i+1} - m_i$ and $b_i = p_i - m_i$ we get the following table for values of v_i and b_i:

v_i	0	1	2
p_{i+1}	0	1	1
m_i	0	1	0

b_i	0	0	1	$\overline{1}$
p_i	0	1	1	0
m_i	0	1	0	1

From the table is easy to see that $p_{i+1} = (c_i \text{ or } s_i)$ and $m_i = (c_i \text{ xor } s_i)$. We also know that:

$$u = \sum_{i=0}^{n-2} (s_i + c_i)2^i = p_{n-1}2^{n-1} + \sum_{i=0}^{n-2} (p_i + m_i)2^i \geq 0 \qquad (7.1)$$

when combined with v_{n-1} the following options are possible, assuming that no overflow occurs:

v_{n-1}	$p_{n-1} = 0$	$p_{n-1} = 1$
0	0	$1^{(b)}$
$\overline{1}$	$\overline{1}$	0
$\overline{2}$	$\_\_^{(a)}$	$\overline{1}$
	b_{n-1}	

In case (a), if $v_{n-1} = -2$, then $u \geq 2^{n-1}$ which implies $p_{n-1} \neq 0$ (i.e. $p_{n-1} = 1$). When $v_{n-1} = 0$ then $u < 2^{n-1}$, so either $p_{n-1} = 0$ or $p_{n-1} = 1$ and the next non-zero b_i is negative.

The circuit of the CS→BS converter is shown in Figure 7.1. The circuit operates without carry propagation. Only 1 CLB per signed bit.

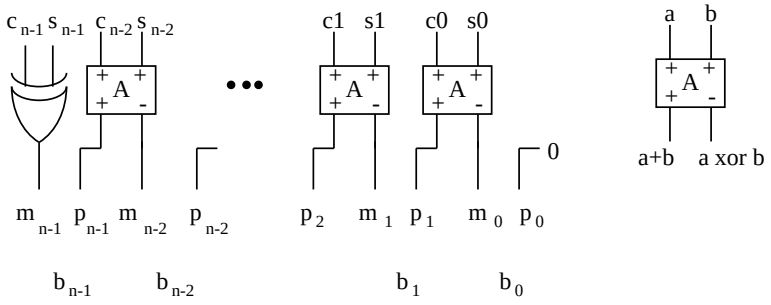

Figure 7.1: CS→BS converter

7.1.2 BS to NR Converter

The circuit that performs $BS \rightarrow NR$ conversion is shown in Figure 7.2. The conversion of redundant representation to NR always imply in the use of a carry or borrow propagation. In this case, the fastest and area efficient circuit in FPGAs should use the Fast Carry Logic (FCL). The circuit shown in the figure is composed of a chain of Full Subtractors (FS). Each FS with inputs a, b and c_{in}, and output c_{out} and s, computes the following expression: $a - b - c_{in} = -2c_{out} + s$. The circuit uses the FCL in the FPGA.

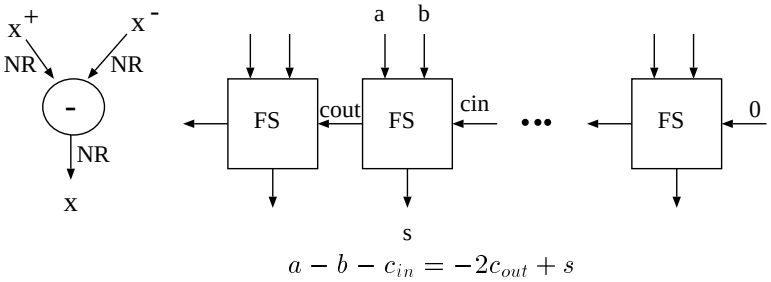

Figure 7.2: $BS \rightarrow NR$ converter

The truth table for the FS function is shown in Table 7.1. From the table we

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 7.1: Truth table for the FS function

obtain the following logical expressions for c_{out} and s:

$$c_{out} = c_{in}a' + c_{in}b + a'b$$

$$s = a \oplus b \oplus c_{in}$$

7.2 Tasks Performed at the Host

The details of the hardware implementation should be hidden from the user. When the software activates one of the VLP operations it does so through a procedure call. This procedure is responsible to check the present configuration of the coprocessor (and take any required action), perform tasks that are not efficiently done in the coprocessor (manipulation of significands and exponents in FP operations), transform digits in radix b (host digit size) to radix r (coprocessor digits) and vice-versa and perform on-the-fly conversion on the result generated by the coprocessor.

This section presents the data organization used in some of the VLP software and hardware systems discussed in Chapter 3, and also describes host tasks in more detail, preparing the reader for the performance evalution of the system composed of coprocessor and host.

7.2.1 VLP Number Format

Long-precision integers are stored in the host as vectors of integer variables. LP floating-point numbers are usually represented in: *multiple digit format* or *multiple term format*. In the *multiple digit format* the FP number is composed of a single exponent and a sequence of high-radix digits that form the significand. The *multiple term* format considers the long-precision number is expressed as a collection of ordinary floating-point numbers, each one with its own significand and exponent. See Figure 7.3.

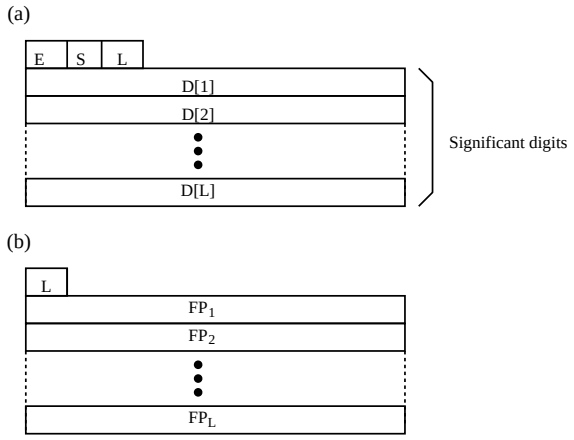

Figure 7.3: Formats of VLP numbers: (a) multiple digit and (b) multiple term

The multiple digit format has the fields: exponent (E), significand's sign (S), significand's length (L) and vector of machine words or high-radix digits ($D[i]$). The significand of the long-precision number is represented in sign and magnitude form.

The multiple term format is composed of a vector of FP numbers and a field that indicates the vector's length.

There are advantages and disadvantages in each scheme, but in particular, the multiple digit format is the one that can more compactly represent most numbers, since only one exponent is stored. The advantage of using multiple term is the

representation of only the significand digits that are different than zero. This feature would allow to skip over zeroes during the computation, what would imply in faster implementation, or even more compact number representation in some cases. However, the probability of having a long sequence of zeroes in the LP number is very small, what makes this "advantage" not significant. In this thesis we assume the multiple digit format.

7.2.2 On-the-Fly Conversion

On-the-fly [EL87] conversion of the result from signed digits to conventional representation (sign and magnitude) is not efficiently done in the coprocessor because it would be necessary to hold the digits until, in the limit, all of the result digits were defined. This condition would prejudice the digit transfer time to the host. Besides that, the result vector would have to be manipulated in the coprocessor local memory thus requiring special hardware in the FPGA to convert the result digits serially (similar to the on-line recurrence equation computation).

For these reasons, the OFC operation is better executed by the host, concurrently with VLP operation performed by the coprocessor. The host converts the result vector composed of SDs in NR representation (two's complement) into a digit vector in sign-and-magnitude form. Digits in NR representation are composed of two fields: S (sign in the most significant bit) and F (other bits of the representation). Field F is considered as a positive binary representation.

The host manipulates 3 vectors: RESULT (where the final representation is stored), Q and QM (where temporary values of the digits are stored). The algorithm shown in Algorithm 9. The overline symbol means bit complementation and SD represents the signed digit received from the coprocessor.

In particular, when the VLP multiplication algorithm generates an output with precision larger than m, some of the output digits are available in the residual vector after the operand digits are received. These digits can be transferred to the host very fast (see Section 4.1). One option to speedup the OFC consists in

Algorithm 9 *: OFC algorithm executed by the host processor*

```
/* receive the first digit different than zero */
/* SD is a signed digit read from the coprocessor */
while (SD = 0)
      write zero to RESULT
if (SD > 0)
  result is positive
  store F in Q and F − 1 in QM
  while the last is not received
       case the digit is
            positive:
                    move data on Q to RESULT
                    empty Q and QM
                    store F in Q and F − 1 in QM
            negative:
                    move data on QM to RESULT
                    empty Q and QM
                    store F in Q and F − 1 in QM
            zero:
                 append 0 to Q and r − 1 in QM
if (SD < 0)
  result is negative
  store F̄ in QM and F̄ + 1 in Q
  while the last is not received
       case the digit is
            positive:
                    move data on QM to RESULT
                    empty Q and QM
                    store F̄ + 1 in Q and F̄ in QM
            negative:
                    move data on Q to RESULT
                    empty Q and QM
                    store F̄ + 1 in Q and F̄ in QM
            zero:
                 append 0 to Q and r − 1 in QM
move all information in Q to RESULT.
STOP
```

performing the conversion in two phases: (1) the multiplier is generating z_j and (2) the multiplier is sending the digits stored in the last residual. In phase (1), the host executes the OFC algorithm generating two vectors Q and QM that correspond to two possible representations of the received digit vector. In phase (2), the multiplier sends the residual digits from least-significant to most-significant, and perform conversion as the digits are transferred. This operation is done by a serial BS→NR converter (serial subtractor). The subtraction of two k-bit vectors will result in another k-bit vector G and a borrow bit (b). The non-redundant result is obtained by the host selecting Q if $b = 0$, or QM if $b = 1$, and concatenating the selected vector with vector G.

For example, assume that the product is represented in redundant form, $r = 10$, as $(10\overline{2}8\overline{4}6)$, where $\overline{x} = -x$. Assume also that the host receives the first 3 digits during phase (1). The host creates the following two vectors, based on the OFC algorithm: (098) and (097). When the multiplier transmits the other digits it executes the serial operation $(806) - (040)$, that results in the digit stream: 6, 6, 7 and 0 (from least to most-significant digit, also including the borrow bit). As $b = 0$, the vector (098) is selected and the final converted product is: (098766).

7.2.3 Digit Expansion and Compression

Before the LP number is transfered to the coprocessor the host must adjust the digit radix. The host store the significand in sign and magnitude format, that consists of a sign bit and a long string of digits in radix $b = 2^w$ (D). The coprocessor, on the other hand works with signed digits in radix $r = 2^n$ that will require $n + 1$ bits each. During digit conversion from radix b to radix r, sign bits are introduced for each coprocessor digit. For high utilization of communication and storage resources an integer number of coprocessor digits should fit in one host word. That implies:

$$k(n+1) = w \quad \rightarrow \quad n = \left\lfloor \frac{w}{k} - 1 \right\rfloor \tag{7.2}$$

where k is the number of coprocessor digits per host word. VLP division and square root define a minimum k of 2. The value of n is defined as a function of the hardware resources and VLP algorithm.

For a host working with 32-bit digits in SM and $k = 2$ we obtain $n = 15$. Taking L as the length of the LP number in the host, after this transformation of digits, the LP number has $m = \lceil \frac{Lw}{n} \rceil$ coprocessor digits. The host executes the data transformation decribed in Procedure 1 to convert digits in radix b to radix r, for the particular case of $w = 32$, $k = 2$ adn $n = 15$. For other values of k,w and n other masks and shift operations must be used, that are easily deduced from the given procedure.

A similar operation is performed by the host to compress the digits already received from the coprocessor and converted to SM format (OFC). The procedure 2 shows the digit compression task for the same parameters used in the conversion from host to coprocessor.

7.2.4 VLP Floating Point Operations

The algorithms developed for multiplication, division and square-root over fixed point numbers are the main components for the development of VLP floating point operations that are executed by the host and coprocessor together. It is not worthwhile to implement in hardware the portions of the VLP computation that are executed only once. Some of these tasks are: manipulation of exponents, adjustment of significands to fit the assumptions used in the VLP algorithms, post-correction and normalization of results.

In the next sections we describe the operations executed at the host in order to implement VLP FP operations, making use of the coprocessor VLP operations.

Procedure 1 *: host executes base conversion (host → coprocessor)*

```
// example of digit expansion using k=2, w=32, n=15
mask1 = 0x7fff0000;
mask2 = 0x00007fff;
sb = 0; /* number of spill bits */
spill = 0;
for (i=1;i<=L;i++) /* D[1] is the most significant host digit */
{ /* insert the previous spill bits into the new processor word */
    word = (D[i] >> sb) | (spill << (w-sb));
    spill = (D[i] << (w-sb-k)) >> (w-sb-k);
    temp = (word >> 1) & mask1;
    temp2 = (word >> 2) & mask2;
    temp = temp | temp2;  /* two digits are mounted */
                          /* spill bits in the spill word */
    /* send temp to coprocessor */
    sb = sb + k;
  if (sb>=(k*n)) /* too many bits in the spill-bit register */
    {
      word = spill << (w-sb)
      temp = (word >> 1) & mask1;
      temp2 = (word >> 2) & mask2;
      temp = temp | temp2;
    /* send temp to coprocessor */
      sb = 0;
      spill = 0;
    };
};
/* insert the remainder spill bits into a blank word */
word = (spill << (w-sb))
temp = (word >> 1) & mask1;
temp2 = (word >> 2) & mask2;
temp = temp | temp2;
/* send temp to coprocessor */
```

Procedure 2 *Procedure executed at the host for base conversion (coprocessor → host)*

```
/* vector R stores the received and converted  */
/* coprocessor digits.                         */
mask1 = 0xfffe0000;
mask2 = 0x0001fffc;
sb = 0;
w = 32;
j = 1;
for (i=1;i<=m;i++) /* use 2m for integer multiplication */
{ /* compress the word */
    temp = ((R[i]<<1) & mask1) | ((R[i] & mask2)<<2);
  /* insert the value into register D */
    if (sb > 0)
      { D[j-1]=D[j-1] | (temp >> (w-sb));
        D[j] = temp << sb;
        sb = (sb + 2) % w; /* empty bit places in D[j] */
      }
     else
      { D[j] = temp;
        sb = 2;
      };
  if (sb > 0) j++;
};
```

7.2.4.1 Notation

To make a clear difference between the numbers manipulated at the host level and at the coprocessor level, we introduce the following notation. The host works with long-precision floating point numbers with the format presented in section 7.2.1. The format includes the long-precision significand and an exponent, that is unbiased. The long-precision FP number is represented as:

$$x_{FP} = x_{mfp} \cdot r_h^{x_{efpr}}$$

where x_{mfp} is the significand of the FP number (in the range $\frac{1}{r} \leq x_{mfp} < 1$) and x_{efpr} is the exponent in radix r_h. Lets assume that the exponent is not biased, that means, a positive or negative sign is explicitly assigned to it. The radix r_h is defined as a function of the number of bits used in each digit of the host processor (w), as:

$$r_h = 2^w$$

The coprocessor works in a different radix ($r = 2^n$) that depends on the available resources in the reconfigurable hardware.

The host manipulates numbers composed of digits in radix r_h. The digits are converted from radix r_h to radix r by the host and transferred to the coprocessor that manipulates digits in radix r. Operators sent to the coprocessor are considered as fixed point numbers. The range of these numbers depend on the VLP algorithm. It is convenient to have the FP number expressed as:

$$x_{FP} = x_{mfp} \cdot 2^{wx_{efpr}} = x_{mfp} \cdot 2^{x_{efp}} \tag{7.3}$$

this transformation provides a better format for the representation of bit shifts.

7.2.4.2 VLP FP Multiplication

The VLP FP multiplication $Z_{fp} = X_{fp}Y_{fp}$ is obtained as:

$$Z_{fp} = z_{mfp} \times r_h^{z_{efpr}}$$

$$z_{mfp} = x_{mfp} \times y_{mfp}$$

$$z_{efp} = x_{efpr} + y_{efpr}$$

The VLP multiplier will be better utilized if operands are in the range $[\frac{1}{r}, 1)$. However x_{mfp} and y_{mfp} are in the range $[\frac{1}{r_h}, 1)$. As $r_h \geq r^2$, there is a possibility that the most significant radix-r digits of the operands (x_{mfp} and y_{mfp}) are zeroes. Two options:

1. the significands are not scaled, and the multiplier waste some cycles with the leading zeroes.

2. normalize the significands to be in the range $[\frac{1}{r}, 1)$. This process requires bit shifts to scale the operands and bit-shifts to correct the result. For example: consider the numbers $x = (0.\ 03\ 47)_{100}$ and $y = (0.\ 89\ 83)_{100}$, with $r_h = 100$ and $r = 10$. The most significant digit of x represented in radix r_h is 03, and the most significant digit in radix $r = 10$ is zero. Performing a left shift of 1 digit in radix r we get $(0.347)_{10}$. The product is $(0.311710)_{10}$ but the correct result is $(0.03\ 11\ 71\ 00)_{100}$, thus a right shift in radix r is required.

For simplicity we assume the first case and do not consider that the host expend cycles in the process.

A post-correction is needed if the most significant digit of the result (z_{mfp}) is zero. In this case, the significand is shifted one host digit to the left (reducing the significand lenght by 1) and the exponent is decremented by one.

$$z_{mfp} \leftarrow z_{mfp} \times r_h$$

$$z_{efp} \leftarrow z_{efpr} - 1$$

In GMP there is no rounding of the significand after the multiplication computes z_{mfp} with twice the precision of x_{mfp} or y_{mfp} (whichever has the largest precision $= max(||x_{mfp}||, ||x_{mfp}||)$). As presented in Chapter 4, the VLP multiplication algorithm avoids unnecessary computation when the precision of the output is the

same as the operands and a significant speedup in the use of the coprocessor is expected in this case.

The host may also provide to the coprocessor only the necessary precision for the requested computation. If the output precision that was requested is m, and the the precision of the operands are: m_1 and m_2, the host may send to the coprocessor the precision m if $m < m_1 + m_2$, or the precision $m_1 + m_2$, otherwise.

7.2.4.3 VLP FP Division

Consider the FP division $Q_{fp} = N_{fp}/D_{fp}$. One condition for the VLP Division algorithm is to have the divisor d (fixed-point number) in the range $0.5 \leq d < 1$. However, the FP number D_{fp} has the significand in the range $\frac{1}{r} \leq d_{mfp} < 1$. Thus, the following operations are performed at the host:

1. shift d_{mfp} to the left until the first fractional bit is one. The normalized number is called d_n. The number of bit positions shifted is S.

2. compute the short precision scaling factor $M = \frac{1}{\hat{d}_n}$ using the FP ALU at the host (this operation is performed at the host considering a short precision reciprocal of the divisor's significand, as explained in Section 5.3.1). The truncated value of d_n ($\hat{d}_n$) has 2 fractional digits.

3. provide the value M to the coprocessor for on-line pre-scaling or perform prescaling at the host level to generate Md_n and Mn_{mfp}.

4. pass the operand digits (scaled or not, depends on previous step) to the coprocessor.

5. read the result digits performing the OFC algorithm described in Section 7.2.2. The converted result is q.

6. compute the exponent of the result $q_{efp} = n_{efp} - d_{efp} + S$. The value S was added to correct the resulting significand by the positions shifted to adjust the range of the divisor (step 1).

7. correct the exponent and significand depending on the most significant digit of q.

7.2.4.4 VLP FP Square Root

The condition for VLP square root computation was shown in Chapter 6. The fixed-point radicand X manipulated in the VLP operation is in the range $0.25 \leq x < 1$. The floating-point number X_{fp} must be manipulated by the host as follows, in order to compute $Z_{fp} = \sqrt{X_{fp}}$:

1. correct the significand: as $\frac{1}{r} \leq x_{mfp} < 1$, then x_{mfp} can be less than 0.25, and it is necessary to shift the significand to the left. Shift left operation of p bits on the significand is equivalent to multiply x_{mfp} by 2^p. Another reason to correct the significand is to have an odd value for x_{efp}. The new number is $x_{mfp}2^p \times 2^{x_{efp}-p}$ such that $(x_{efp} - p)$ is even.

2. compute the short precision $\hat{Y}$ based on a short precision computation of $\sqrt{x_{mfp}2^p}$. Compute $\hat{Y}^{-1}$. Send both values to the coprocessor (after digit conversion).

3. submit the corrected significand to the coprocessor.

4. read the result and perform OFC to obtain z_{mfp}.

5. compute the exponent

$$z_{efp} = \frac{x_{efp} - p}{2}$$

   If z_{efp} is not divisible by w ($r_h = 2^w$), correct z_{mfp} performing k right bit shifts until $z_{efp} = \frac{x_{efp}-p}{2} + k$ is divisible by w. Make $z_{efpr} = \frac{z_{efp}}{w}$

Consider the following example in radix $r = 8$: $X_{fp} = (0.2375)_8 \times 8^3 = (0.2375)_8 \times 2^9$. And $\sqrt{X_{fp}} = (0.1450)_8 \times 8^2$. Note that $x_{mfp} > 0.25$ already, however

$$\sqrt{(0.2375)_8 \times 2^9} = \sqrt{(0.2375)_8} \times 2^{4.5}$$

creates a non-integer power of 2, which is not easy to compute. Thus, the significand is shifted to the left one bit position to obtain the new FP number $(0.4772)_8 \times 2^8$. The significand of this new number is applied to the coprocessor that returns the result $(0.6242)_8$. The result exponent z_{efp} will be first computed as 4. Since 4 is not divisible by $w = 3$, two right shifts of the final result will result in the significand $(0.1450)_8$, $z_{efp} = 6$ and $z_{efpr} = 2$ ($2^6 = 8^2$).

Chapter 8

VLP Circuit Design for FPGAs

This chapter shows the estimates of area consumption for the data path, selection function and other modules used in the VLP circuits. We are not concerned with the control complexity in this study. The given estimates are upper bounds on the number of CLBs in the designs. Optimizations at the synthesis level are expected to reduce the actual number of CLBs.

VLPA operations are constructed based on the presented building blocks. Since we are considering reconfigurable architectures, the precision of these basic operators can be adjusted in order to satisfy the performance requirements of the higher level arithmetic algorithms for VLPA.

The first section discuss important aspects of the VLP algorithms in FPGAs. The next sections show the estimates of area and time of the arithmetic operators used in the VLP data path and selection functions.

8.1 Important Design Aspects

8.1.1 FPGA Time Parameters

In all estimates we consider 4-input LUT FPGAs. The main FPGA parameters that are considered in these estimates are shown in Table 8.1. Specific values for a particular device is shown in Appendix A. The delay of a circuit always includes the delay of the interconnect to deliver the output value.

8.1.2 Pipeline Degree

Different degrees of pipelining can be utilized to shorten the cycle time and improve the overall performance. As discussed in [TE96], a long precision on-line

CLB Switching Characteristics	
Description	Symbol
Combinational Delays	
F/G inputs to X/Y outputs	T_{ILO}
F/G inputs via H to X/Y outputs	T_{IHO}
C inputs via DIN through H to X/Y outputs	T_{HH2O}
CLB Fast Carry Logic	
Operand inputs to COUT	T_{OPCY}
Add/Subtract input to COUT	T_{ASCY}
Initialization inputs to COUT	T_{INCY}
CIN through FGs[1] to X/Y outputs	T_{SUM}
CIN to COUT, bypass FGs	T_{BYP}
Carry network delay (COUT to CIN)	T_{NET}
Sequential Delays	
Clock K to outputs Q	T_{CKO}
Setup time before Clock K	
F/G inputs	T_{ICK}
F/G inputs via H	T_{IHCK}
C inputs via DIN	T_{DICK}
Average interconnect delay	t_{interc}
Input/Output Timing Characteristics	
Description	Symbol
Global Low skew clock to Output using Output FF	T_{ICKOF}
Input Setup time, using Global Low Skew clock and Input FF	T_{SPD}
Input Hold Time, using Global Low Skew clock and IFF	T_{PHD}

1. Function Generators.

Table 8.1: FPGA Timing Parameters (adapted from Xilinx data book)

multiply-add module has the best performance for maximum degree of pipelining. A pipelined structure has more latency than a non-pipelined one. The impact of a large latency is not so significant for the arithmetic operations when hundreds or thousands of bits are considered. On the other hand, the pipelined structure has shorter cycle time, that affects all the digits computed by the unit (higher throughput), which reduces significantly the total operation time. Based on this observation we assume that a maximum degree of pipelining is used whenever possible.

Based on the cost/performance relation discussed in [Kwa93], lets consider t as the cycle time time required by a non-pipelined circuit. To execute the same task on a k-stage pipeline, with an equal flow through delay t one needs a clock period of

$$p = \frac{t}{k} + d = \frac{t + kd}{k}$$

that corresponds to a maximum throughput of

$$f = \frac{1}{p} = \frac{k}{t + kd}$$

The total pipeline cost is estimated by $c + kh$, where c is the cost of all logic stages and h is the cost of each latch. The parameter h for FPGAs is practically zero. Defining the performance/cost ratio as in [Kwa93]:

$$PCR = \frac{f}{c + kh} = \frac{k}{(t + kd)(c + kh)} \tag{8.1}$$

that has a maximum for

$$k_0 = \sqrt{\frac{tc}{dh}} \tag{8.2}$$

For a non-pipelined data path circuit ($r = 256$) implemented in a XC4013-5 we obtained the values: $c = 276$ CLBs, $t = 69.1ns$ and $d = 3ns$. With maximum pipeline, the data path has 22 stages. The increase in area due to pipelining is only 84 CLBs, thus, the cost of each latch is $84/22 = 3.8$. These numbers result in $k_0 = 40$, much above the maximum number of stages allowed in the architecture.

Another justification for the use of the largest level of pipelining is the total execution time. The stage delay grows in steps of the CLB delay, plus interconnect time, lets say, T_{FG}. With maximum pipelining of k stages the cycle time is given by the minimum step T_{FG}. With less stages, the next balanced pipelining structure has a cycle time $2T_{FG}$ and $k/2$ stages. The total task time for C inputs is $T_k = (k+C)T_{FG}$ with the maximum pipelined structure, and is $T_{k/2} = (k/2+C)2T_{FG} = (k+2C)T_{FG}$ for the other case. When, $C >> k$ (that is the case for VLP computation), the time to execute the task with half the maximum pipelining degree is basically twice the minimum time to execute the task in the architecture.

8.1.3 Digit Representation

Digits manipulated by arithmetic operators may be in conventional or redundant form.

For positive digits, the representation in conventional form uses $k = log_2(r)$ per radix-r digit and in redundant Carry Save (CS) form it uses $2k$ bits.

A radix-r signed digit in a maximally redundant set may use conventional two's complement and one's complement form or redundant representation, such as carry-save and borrow-save code. The conventional digit representation uses $k+1$ bits per radix r digit, and the redundant representation uses $2k$ bits per digit.

Borrow-save code has been used for a long time in the Illiac III [Atk70]. Using this code, the borrow-save representation of each signed-digit is a vector of *signed bits* ($\underline{b}$) represented by two binary variables, $\underline{b} = (b^+, b^-)$, such that the signed-bit value is evaluated as $b = b^+ - b^-$.

We make use of different digit representations in order to obtain the most efficient arithmetic structures for VLP computation.

8.2 Design of Arithmetic Operators for FPGAs

The main operators used in the design of VLP operations are conventional or on-line. In terms of the data interface, the conventional arithmetic operations can be designed for digit-parallel or digit-serial modes of computation. Digit-serial mode in conventional arithmetic is done Least Significant Digit First – LSDF. On-line algorithms use digit-serial interface and work in the Most Significant Digit First – MSDF – mode of computation.

The algorithm types are related to the way the operands and results are manipulated. The types are: sequential (serial-parallel), unfolded (fully-parallel, non-pipelined/pipelined) and serial algorithms. The fully-parallel type is the most area consuming but it usually has the highest speed. Serial and sequential types are implemented by circuits that use less area than the parallel type and have a lower I/O requirement. On the other hand, the number of cycles to complete an operation is larger for serial and sequential type than for parallel.

We consider in the next sections the features of the basic arithmetic operations in the organizations that are of interest for this work.

8.2.1 Addition

This section presents circuits for parallel and serial addition. The circuits are grouped as: digit-parallel and serial (conventional or on-line) addition.

8.2.1.1 Digit-parallel Addition

Experimental results for conventional addition in the XC4000 series of devices [YX96, Xil93] indicate that Ripple Carry Addition — RCA —, for operands with less than 56 bits, is the fastest approach. For more than 56 bits, a good choice is the Carry Select Adder. Both of the previous adders are also called Carry Propagate Adders (CPA).

Parallel addition using redundant adders, like Signed-Digit Adders — SDA —

would enable addition to be done in a fixed time, independent of the operand precision. The inconvenience of using SDAs is the generation of an output in redundant form. So, it is sometimes necessary to convert the output from redundant to conventional representation. A CPA is used for this task. Redundant adders are justified in the addition of many operands or in the cases when the output of the adder can be used without conversion by another arithmetic structure.

The use of the dedicated carry logic available in many FPGA devices is possible only if SD numbers are represented in two's complement form. In this case the area of a maximally redundant radix-r SD adder is the same of a RCA with $log_2 r + 1$ bits. The delay is proportional to the number of bits.

When signed digits are represented in Borrow Save – BS – code, the fast carry logic cannot be efficiently used since the chain of carries is very short (two radix-2 digits). The area of the SDA in this case is basically 4 times more than the one used by a conventional RCA (2 CLBs per bit), that can be estimated as $2n$ (CLBs). A design using two FAs would have a delay of 2 F-function-generators plus interconnect and it is independent of n (precision).

Another redundant adder that can be used is the Carry Save Adder — CSA. This adder uses half the area used in the SDA adder to combine one operand in CS form and another in conventional form with roughly half of the delay. To combine two operands in CS form, the CSA has the same area and delay of the SDA. Table 8.2 shows the area/delay relation for these adders, without registers, only combinational delay. The types of operands are also shown as NR - non-redundant, CSA - carry-save form and BS - borrow-save form.

8.2.1.2 Digit-Serial Addition

Conventional

Serial addition in the conventional number system receives operands least significant digit first. The basic organization is a short length CPA and a flip-flop. The carry out of one iteration is stored in the flip-flop and used as carry in of the

Adder type	Type of Op.	Area (CLBs)	Delay T_{PADD} [ns]
RCA	1CSA or 2NR	$\lceil \frac{n}{2} \rceil + 2$	$(T_{OPCY} + T_{NET})$ $+(T_{BYP} + T_{NET})\lfloor \frac{n-3}{2} \rfloor$ $+T_{SUM} + t_{interc}$
SDA	2BS code SDs	$2n$	$T_{IHO} + t_{interc}$
CSA	1CSA + 1NR	n	$T_{ILO} + t_{interc}$
	2CSA	$2n$	$2T_{ILO} + 2t_{interc}$

Table 8.2: Area/delay of digit-parallel adders in the XC4000

next iteration. We define the precision of the short length adder as n bits and the total precision of the operands as m bits. In each cycle, n bits are received and processed in parallel (see Figure 8.1). A group of n bits represents a digit in radix 2^n. The number of digits to be added serially is $d(m,n) = \lceil m/n \rceil$.

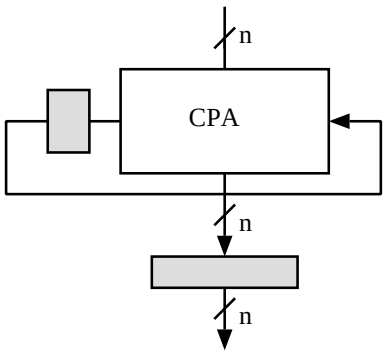

Figure 8.1: Conventional radix 2^n serial Adder

Using RCAs, the special cases of $n = 1$ and $n = 2$ can be implemented without the dedicated carry logic, with area 1 CLB and 2 CLBs, and delay $T_{ILO} + T_{CKO} + t_{interc}$ and $T_{ILO} + T_{HH2O} + 2t_{interc} + T_{CKO}$, respectively. For $2 < n \leq 56$, the use of the fast carry logic produces the best designs in terms of area and delay. Carry select adders should be considered only when $n > 56$.

The cycle time of a radix 2^n serial adder is:

$$t_{cycle}(n) = T_{PADD}(n) + T_{CKO}$$

where $T_{PADD}(n)$ is the delay of a conventional parallel adder of length n. The total time for serial addition of m bits using an adder of n bits is computed as:

$$T_{SADD}(m,n) = d(m,n)t_{cycle}(n)$$

The area of a serial adder dependents only on the digit radix. The inclusion of the FF for the state storage does not increase the area of this adder, in terms of the area already computed for the parallel adder for the same digit size.

On-line Addition

On-line operations rely on the utilization of redundant number system to represent the output. Cost/delay of on-line addition depends on the encoding of redundant operands.

The general structure of an on-line adder is presented in Figure 8.2. It is composed of a redundant adder, a selection function of output digits and registers. The circuit implements the following recurrence equation:

$$W[j+1] = r(W[j] - z_j) + r^{-\delta}(x_{j+\delta} + y_{j+\delta})$$

where variable W is the scaled residual, z_j is the output digit and $x_{j+\delta}$, $y_{j+\delta}$ are input digits in the set $\{-a, \ldots, -1, 0, 1, \ldots, a\}$, with $r/2 \leq a \leq r-1$. The on-line delay δ can be as low as 1 clock cycle, depending on r. The cycle time is a function of the selection function that by itself depends on the radix r. The complexity of the selection function increases as the radix increases.

The design of on-line adders using BS code is presented in [DMV94]. The on-line delay of this design is $\delta = 2$. A radix $r = 2^n$ signed-digit on-line adder is obtained as an extension of the signed-bit adder. The radix-r digit is formed concatenating n signed-bits. The delay of the implementation is independent of the radix. The design of a radix-8 on-line adder is shown in Figure 8.3 and uses full-adders as the

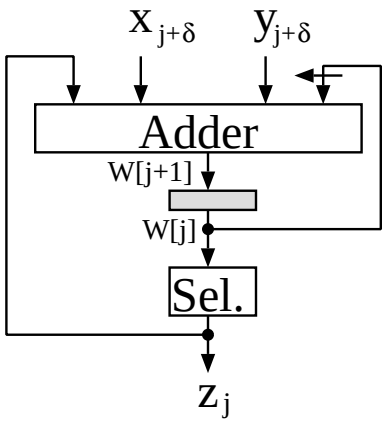

Figure 8.2: Basic on-line adder structure

building block. The delay if the implementation is $T_{OLA} = 2T_{FA} + T_{CKO}$, where $T_{FA} = T_{ILO} + t_{interc}$.

The area of a radix-r on-line adder is $2n$ CLBs. If we include a layer of registers between the layers of FAs, the delay increases in one cycle ($\delta = 3$), but the cycle time is reduced to:

$$T_{OLAp} = T_{FA} + T_{CKO}$$

The total time to perform serial radix 2^n on-line addition of operands with m bits is $T_{OLA}[d(m,n) + \delta - 1]$. The increase in area due to pipelining corresponds to 2.5 CLBs, independent of the radix.

The circuit output is a number in the redundant number system, The use of on-the-fly conversion (OFC) [EL87] reduces this disadvantage. That means, the conversion is done as the output digits are generated such that the output is already converted by the time the last digit is received.

8.2.2 Multiplication

In this section we present the area/time models of multiplier operators in fully-parallel and serial-parallel organizations. Only the most convenient components

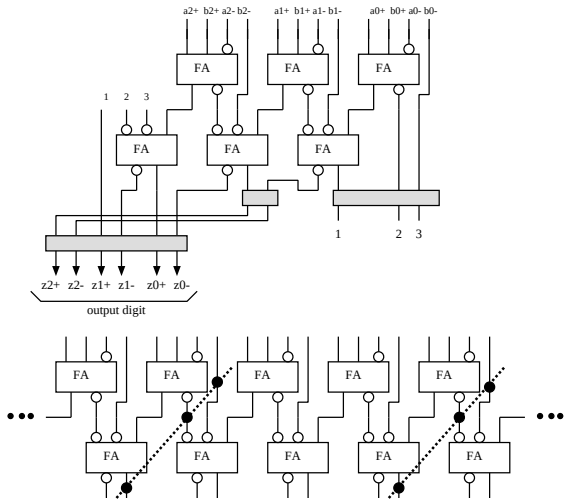

Figure 8.3: Radix-8 on-line adder

for FPGAs are shown.

A multiplier usually consists of multiple generators, a reduction structure that produces the product in a redundant form and a CPA to obtain the product in a conventional form.

8.2.2.1 Parallel multiplier

Parallel $n \times n$ multipliers are potentially the fastest but use the most area. The reduction structure can be a tree of FAs, a group of RCAs, a network of column compressors, etc. We describe in the following sections the organizations used in the VLP algorithms. A general description of multiplier structures is given in [Kor93].

In this thesis we use the multiplier operator in the implementation of the digit by vector multiplier (DV), already defined in previous chapters. The DV multiplier uses a linear-array structure that is described in the next.

Linear-Array multiplier

The linear-array multiplier uses CSAs to reduce the multiples and include a CPA

to obtain the final conventional product, see Figure 8.4. In this case, the area is $\frac{n}{2} + (n-1)n + \lceil \frac{n}{2} \rceil + 2$ CLBs and the delay is $n(T_{ILO} + t_{interc}) + T_{RCA}(n) + t_{interc}$. This implementation consumes more area but it has lesser delay than the first one.

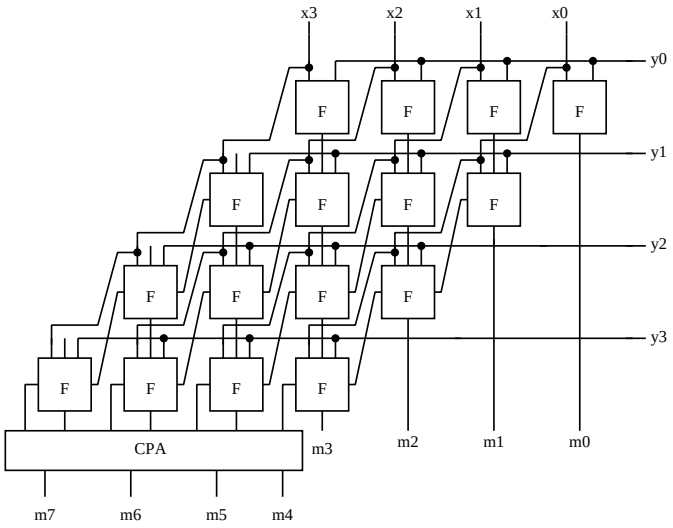

Figure 8.4: Array Multiplier using CSAs

Pipelining is used to increase the throughput and reduce the cycle time. A pipelined version of the array multiplier has a cycle time limited by the last CPA. In our designs, we are able to use a parallel multiplier that generates a redundant representation, and as a consequence, the CSA reduction without the CPA stage can be used. Without the CPA stage, the pipeline cycle time is as low as one CSA delay, plus flip-flop and interconnect delay. The area is affected by the required registers between stages to store the value X and bits of Y used by following stages.

Many methods can be used to improve the performance of the array multiplier, among them: Booth recoding and other alternatives for multiple reduction. Booth recoding is discussed next. Other alternatives for reduction were considered in the work, but were discarded based on the difficulty to have a single description that could be instantiated for various operand sizes. We looked for a generic description

of a parallel multiplier that could be easily adjusted to different operand sizes. A tree multiplier has an structure that dependents on the size of the operand. For this reason, we do not describe or analyze the tree reduction structures or column compression schemes in this thesis. A faster implementation could certainly be obtained if these other reduction schemes were applied to the particular implementation of the VLP algorithms described in this work.

Booth recoding [Kor93] is used in multiplication to transform the binary vector that represents the operand Y into another vector composed of radix-4 digits in the set $\{-2,-1,0,1,2\}$. The multiplication of a radix-4 digit in this digit set by X is obtained by simple operations like complementation and shifting. The output of the Booth recoding circuit uses a 3-bit code (z,s,c) that indicates the conditions: zero, shift ($\times 2$) and complement (negative multiple). Booth recoding can be used for signed or unsigned operands. When considering unsigned numbers, an extra most significant bit of 0 must be considered to compute the number of rows.

The structure of the multiplier using this technique is shown in Figure 8.5. Since the output of the recoder consists of 3 bits, 1.5 CLBs are necessary for each radix-4 digit. When applied to the array multiplier the number of rows is basically reduced in half. For an operand of n bits, only $\lceil \frac{n+1}{2} \rceil - 1$ addition stages are required. An estimate of the array multiplier area using Booth recoding and CSA reduction of multiples is presented in Table 8.3. As shown in Figure 8.7 the multiple generator uses 2 4-input LUTs and so 1 CLB per bit is required. If a multiplier with conventional representation is required we need to add the CPA area, shown in Table 8.4. Also, in the Table we present the extra area required for maximum pipelining.

The time to compute the multiplication is divided in 3 parts: Booth recoding time, multiple generation, CSA addition stages and CPA addition stage, that adds up to:

$$T_{mult}(n) = T_{BR} + T_{MG} + (\lceil \frac{n+1}{2} \rceil - 1)T_{CSA} + T_{RCA}(2n-2) \qquad (8.3)$$

Area of array multiplier for n-bit operands			
Component	CLBs per radix-4 digit	quantity	total
Booth recoder	1.5	$\lceil \frac{n+1}{2} \rceil$	$\frac{3}{2}\lceil \frac{n+1}{2} \rceil$
multiple generator	$n+1$	$\lceil \frac{n+1}{2} \rceil$	$(n+1)\lceil \frac{n+1}{2} \rceil$
CSAs	$n+1$	$\lceil \frac{n+1}{2} \rceil - 1$	$(n+1)(\lceil \frac{n+1}{2} \rceil - 1)$
Total			$(2n+\frac{7}{2})\lceil \frac{n+1}{2} \rceil - (n+1)$

Table 8.3: Array Multiplier with Booth Recoding (CS output)

CPA	$\frac{n}{2} + \lceil \frac{n+1}{2} \rceil + 1$
pipeline (add)	$\frac{1}{2}(\frac{n}{2}-1)(\frac{n}{2}-2) + \frac{n^2}{4}$

Table 8.4: Extra area for the Linear-Array Multiplier

where

$$T_{BR} = T_{ILO} + t_{interc}$$

and

$$T_{MG} = T_{IHO} + t_{interc}$$

For a multiplier that generates the output in CS form, the CPA time (T_{RCA}) should be removed from the previous equation.

The cycle time of a pipelined multiplier is dominated by the multiple generator delay. The clock cycle time reduces to $T_{IHO} + T_{CKO}$ and the total number of stages in the multiplier for an operator of n bits is $\lceil \frac{n+1}{2} \rceil + 1$.

8.2.2.2 Serial-parallel multiplier

When the speed of the component is not a vital factor, serial-parallel operators can be used. There are some possibilities for serial-parallel multipliers, however, for FPGA technology, the design proposed in [Lou94] is the most efficient one, beside having other features that are adequate for FPGAs.

Parallel-serial multiplication is obtained combining results of digit by word multipliers. Multiplier designs like the one shown in [Pet95, HP95] broadcasts the digit

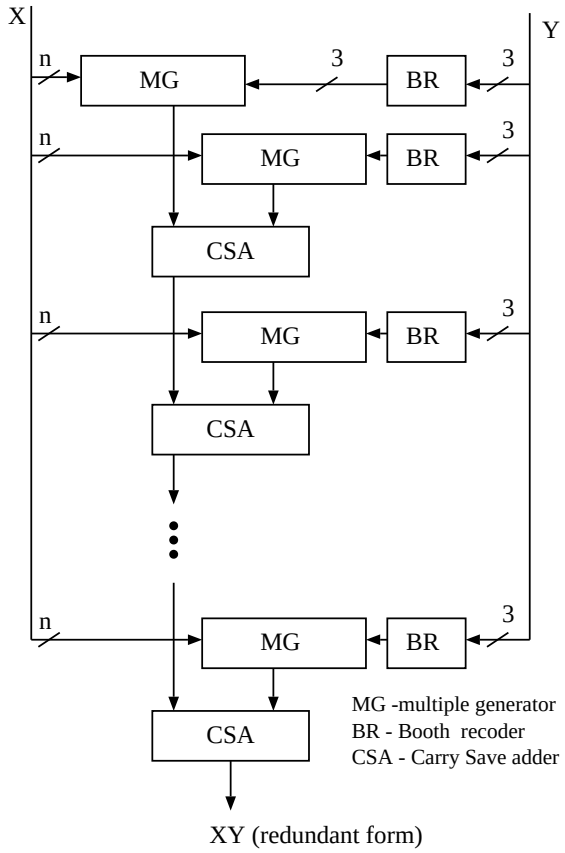

Figure 8.5: Array Multiplier using Booth recoding and CSAs

value to several arithmetic modules at the same time, such that they can compute the digit by word product in parallel. This type of solution implies a large fan-out for the digit communication line (broadcast of the signal).

The broadcast of signals is particularly expensive on FPGAs. As in ASICs, the delay of signals depends on the load and length of the interconnects. However, because of the flexibility of the interconnect network in FPGAs, the communication delays grow faster. The Linear Sequential Array [Lou94] is a generalization of the pipelining structure suggested in [Erc84] and it can be used effectively to reduce broadcast of signals and pipeline arithmetic data paths.

Figure 8.6 shows a radix-4 LSA multiplier with 2 bits per LSA module. Booth

recoding is used to transform radix-2 into radix-4 digit set. The input Y' is a subvector of Y containing only the bits necessary to perform the Booth recoding algorithm in each step. Variable z_j represents a radix-4 digit in the set $\{-2, -1, 0, 1, 2\}$. The particular case of the computation done in LSA1 is also shown in Figure 8.6.

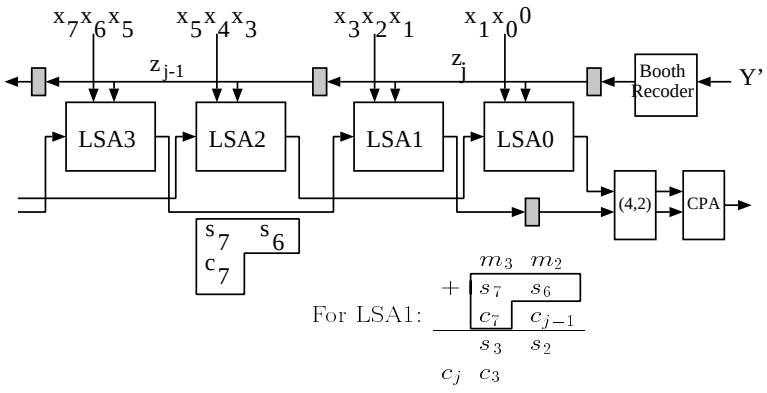

Figure 8.6: LSA multiplier (radix 4)

The addition of values is done in CS form. The carry out generated during addition in one clock cycle (c_j) is kept to be used in the next clock cycle (c_{j-1}). Because of that, the bits transferred from LSA3 do not include c_6. The values of m_i correspond to the bits of the product xz_j that are computed in the digit slice. The values obtained in LSA1, for example, are:

z_j	m_3m_2
-2	$x'_2x'_1$
-1	$x'_3x'_2$
0	00
1	x_3x_2
2	x_2x_1

The scheme of a radix-4 LSA is shown in Figure 8.7. It is composed of FAs and multiple generators. In the figure, these components were already mapped to CLBs. A general description of the operation of LSA multipliers is found in [Lou94].

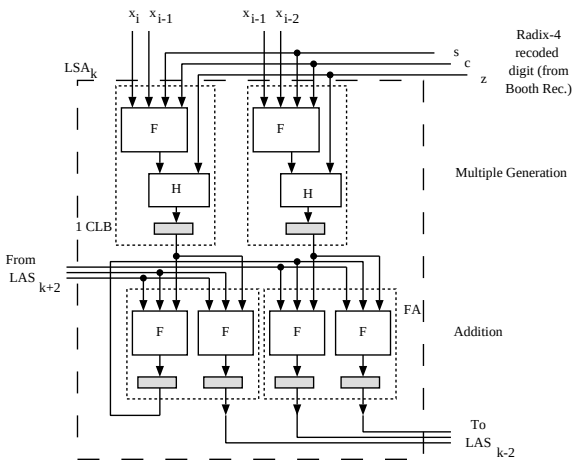

Figure 8.7: Radix-4 LSA module

A complete example of the radix-4 LSA multiplication algorithm with $X = 01101101$ and $Y = 01111101$ is shown in Table 8.5. The recoded value of Y is $Z = 20\overline{1}1$. The values inside boxes are carries, that are generated in one cycle and used in the next cycle, in the same LSA module. Variable A is set to 1 every time the value at LSA1 is negative (value 0 otherwise). This variable, combined with the complementation of digits performed in each LSA module, is used to generate the negative values of X or $2X$ (in the two's complement form). The most significant LSA is different from the others in the sense that it must control the sign extensions.

The area and delay of this design for $r = 16$ (using two radix-4 Booth recoders) is [Lou94]:

$$T_{lsa}(n) = (4 + \lceil \frac{n}{2} \rceil) t_{cycle} \tag{8.4}$$

$$A_{lsa}(n) = 30 + 18(\lceil \frac{n}{4} \rceil)(CLBs) \tag{8.5}$$

where $t_{cycle} = T_{IHO} + t_{interc} + T_{CKO}$ (from [Lou94]), $t_{cycle} = 12.7ns$ for the XC4010-5 and the estimate would generate $t_{cycle} = 12ns$). Each LSA module uses 18 CLBs. Four bits of the result are generated in each cycle.

	LSA3	LSA2	LSA1	LSA0
$X =$	01	10	11	01
$z =$	0	0	1	1
$xz =$			11	01
transfer bits			00	00
			00	00
CS form			11	01
			00	0A
$z =$	1	1	-1	-1
$xz =$	01	10	00	10
transfer bits	00	00	00	00
	0x	00	00	00
CS form	01	10	00	10
	0x	00	00	0A
$z =$	-1	-1	0	0
$xz =$	10	01	00	00
transfer bits	00	00	01	10
	0x	00	00	00
CS form	10	01	01	10
	0x	00	00	0A
$z =$	0	0	2	2
$xz =$	00	00	10	10
transfer bits	11	00	10	01
	0x	00	00	00
CS form	11	00	00	11
	0x	00	00	0A
$z =$	2	2	0	0
$xz =$	11	01	00	00
transfer bits	11	00	11	00
	0x	00	00	00
CS form	00	01	10	00
	1x	00	01	0A
$z =$	0	0	0	0
$xz =$	00	00	00	00
transfer bits	00	00	00	01
	0x	00	10	00
CS form	00	00	10	01
	0x	00	00	0A

Table 8.5: LSA Multiplication - radix 4

8.2.3 Summary of Results

A summary of the results discussed in this section is presented in Table 8.6 with the following conventions:

- n – precision of operators (input precision).
- $d(m,n) = \lceil m/n \rceil$ – number of input digits;

Time values are given as a function of the FPGA parameters. For serial or pipelined implementations, the delay is given in terms of the number of cycles and cycle time. Only the operators of interest for the circuits proposed in this thesis are listed in the Table.

8.3 Recoder Circuit for the VLP Multiplier

The digit recoder used in the VLP multiplier is implemented as a simplified radix-r on-line adder with one operand restricted to -1, 0, and 1 values. The most significant signed bits (*sbits*) of the digit are zeroes which causes the removal of most of the full adder (FA) modules in the on-line adder, with some some modification in the interconnections.

The upper row of modified FA modules used in the on-line adder produces 2 binary outputs, t and u, from 3 binary variables x, y, and z, such that $x + y - z = 2t - u$. The switching expressions are $t = xy + xz' + yz'$, and $u = x \oplus y \oplus z$. In the recoder case, since $x = z = 0$ for many of the inputs, we have that $t = u = y$. A radix-8 on-line recoder is shown in Figure 8.8. It is a simplification of the radix-8 on-line adder shown in section 8.2.1. The two least significant sbits are delayed to synchronize the values between different cycles. Another register is used for z_2 to align the sbits in the output digit.

The implementation of this module with FPGAs XC4000 uses $k + 1$ CLBs for a radix $r = 2^k$ recoder. The delay is equivalent to two F-function generators plus flip-flop and interconnection delay.

Components	Area (CLBs)	Time
Basic blocks		
BR	$\frac{3}{2}\lceil\frac{n+1}{2}\rceil$	$T_{BR} = T_{ILO} + t_{interc}$
MG	n	$T_{MG} = T_{IHO} + t_{interc}$
CSA	1	$T_{CSA} = T_{ILO} + t_{interc}$
Parallel Adder		
RCA	$\lceil\frac{n}{2}\rceil + 2$	$T_{RCA}(n) = (T_{OPCY} + T_{NET})+$ $(T_{BYP} + T_{NET})\lfloor\frac{n-3}{2}\rfloor+$ $T_{SUM} + t_{interc}$
Serial Adder		
OLA ($\delta = 2$)	$2n$	$2(T_{ILO} + t_{interc}) + T_{CKO}$ $(d(m,n) + 1$ cycles)
OLA (pipelined, $\delta = 3$)	$2n + 2.5$	$T_{ILO} + t_{interc} + T_{CKO}$ $(d(m,n) + 2$ cycles)
Conventional (using RCA)	$\lceil\frac{n}{2}\rceil + 2$	$T_{RCA}(n) + t_{interc} + T_{CKO}$ $(d(m,n)$ cycles)
Parallel Multiplier[1]		
Array multiplier	$(2n + \frac{9}{2})\lceil\frac{n+1}{2}\rceil - \frac{n}{2}$	$T_{BR} + T_{MG} + (\lceil\frac{n+1}{2}\rceil - 1)T_{CSA}$ $+T_{RCA}(2n-2)$
Array multiplier (no CPA)	$(2n + \frac{7}{2})\lceil\frac{n+1}{2}\rceil - (n+1)$	$T_{BR} + T_{MG} + (\lceil\frac{n+1}{2}\rceil - 1)T_{CSA}$
Pipelined multiplier (add)	$\frac{1}{2}(\lceil\frac{n}{2}\rceil - 1)(n - \lceil\frac{n}{2}\rceil - 2)$ $+\lceil\frac{n+1}{2}\rceil\lceil\frac{n}{2}\rceil$	$T_{IHO} + t_{interc} + T_{CKO}$ $(\lceil\frac{n+1}{2}\rceil + 1$ cycles)
Serial-parallel Multiplier		
LSA multiplier ($r = 16$)	$30 + 18(\lceil\frac{n}{4}\rceil)$	$(T_{IHO} + t_{interc} + TCKO)$ $((4 + \lceil\frac{n}{2}\rceil)$ cycles)

1. n includes the sign bit, thus it corresponds to the total number of bits in a digit NR representation.

Table 8.6: Area and time estimates for addition and multiplication of n-bit operators using 4-input LUT FPGAs

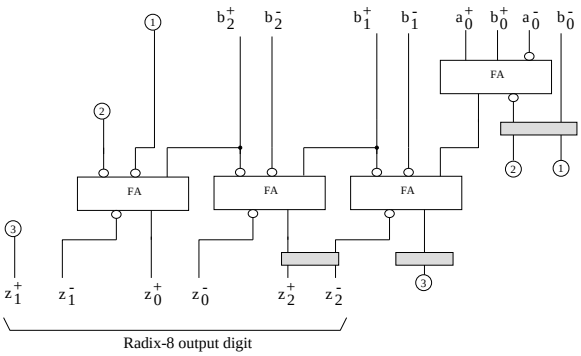

Figure 8.8: Radix-8 On-Line Recoder.

Component	Area (CLBs)
Input Registers	$3n$
CPA (BS to NR conversion of 3 digits)	$3(\lceil \frac{n}{2} \rceil + 2)$
CPA (rounding)	$\lceil \frac{n+2}{2} \rceil + 2$
TOTAL	$3n + 9\lceil \frac{n}{2} \rceil + 8$

Table 8.7: Area of the VLP division selection function (digits in radix 2^n)

8.4 VLP Data Path Area Estimates

The basic data path used in VLP algorithms presented in the previous chapters was designed using components listed in the previous sections. Other components not listed are digit recoders and basic logic components, such as multiplexers and shifters. The area used in the recoder of the VLP multiplier was presented in section 8.3. Other recoders were discussed in the previous chapter.

The selection function for VLP division has an area that includes the components shown in Table 8.7.

The area of the selection circuit for the VLP square-root is shown in Table 8.8. This selection function is more complex than the one for division. Based on the operation of the VLP square-root algorithm we select arithmetic operators that are inexpensive in terms of area. The truncated residual ($\hat{W}[j]$) has 2 fractional

Component	Area (CLBs)
$(\hat{Y})^{-1}$ Register	$n + 1.5$
BS to NR conversion (3 digits + 1 bit)	$(\lceil \frac{3n+1}{2} \rceil + 2)$
mult. input shifter	$n + 1$
multiplier (LSA_{16})	$30 + 18\lceil \frac{3n+2}{4} \rceil$
mult. output shifter	$\frac{n}{2} + 1$
CPA (rounding)	$\lceil \frac{n+2}{2} \rceil + 2$
Mux for register (a)	$\lceil \frac{n+1}{2} \rceil$
TOTAL	$3n + 38 + \lceil \frac{3n+1}{2} \rceil + 18\lceil \frac{3n+2}{4} \rceil + \lceil \frac{n+2}{2} \rceil$

Table 8.8: Area of the VLP square root selection function (digits in radix 2^n)

digits and one integer digit in BS code (each digit has $2n$ bits, $n = log_2 r$). The approximation of the output has $2n$ fractional bits plus 2 integer bits. A serial-parallel multiplier is used to scale the residual (LSA multiplier in radix 16), for this reason, a shifter is used to generate one of the inputs (the reciprocal of the input estimate $\hat{Y}^{-1}$). The other input (truncated residual) is loaded in parallel. Another shifter collects the serial output (only $n + 2$ bits are stored).

The area estimate of the data path with the extra selection circuit options is shown in Table 8.9.

Table 8.10 shows the area used by data paths for some values of n.

8.5 Delay of Selection Functions

Based on the circuits presented in previous chapters, and the time estimates developed, we are able to determine the delay of the selection functions. As the delays used in the previous chapters are normalized in terms of the coprocessor cycles (with cycle time T_{cp}), the values obtained in the previous sections must be divided by T_{cp} to obtain T_{sel}.

Component	Area (CLBs)	Quantity ×	/	√
digit by vector multiplier				
- digit by digit multiplier	$(2n+\frac{11}{2})\lceil\frac{n+2}{2}\rceil-(n+2)$			
- multiplier pipelining	$\frac{1}{2}(\lceil\frac{n+1}{2}\rceil-1)(n-\lceil\frac{n+1}{2}\rceil-1)+\lceil\frac{n+2}{2}\rceil\lceil\frac{n+1}{2}\rceil$			
- CS to BS converter	n			
- OL Adder	$2n+2.5$			
subtotal	$(2n+\frac{11}{2}+\lceil\frac{n+1}{2}\rceil)\lceil\frac{n+2}{2}\rceil+\frac{1}{2}(\lceil\frac{n+1}{2}\rceil-1)(n-\lceil\frac{n+1}{2}\rceil-1)+2n+0.5$	2	2	2
OL Adder	$2n+2.5$	2	3	3
Recoder+(BS→NR) +Mux (VLP Mult.)	$n+1+\frac{n}{2}+1+2\frac{n}{2}$	1		
Selection VLP division	from Table 8.7		1	
Selection VLP sqrt	from Table 8.8			1

Table 8.9: Data Path area for digits in radix 2^n (pipelined)

n	VLP Multiplication	VLP Division	VLP Sqrt
7	322.5	384.5	502.5
8	365	429.5	567.5
15	1010.5	1108.5	1314.5

Table 8.10: Area (# CLBs) of the VLP data path for some values of n

8.5.1 Delay of VLP Division Selection

The selection circuit for VLP division does not have any sequential circuit, only two CPAs. The total time is:

$$t_{divsel} = T_{RCA}(2n+1) + T_{RCA}(n+2)$$

$$t_{divsel} = 2(T_{OPCY} + T_{NET} + T_{SUM} + t_{interc}) + (T_{BYP} + T_{NET})(\lfloor \frac{2n-2}{2} \rfloor + \lfloor \frac{n-1}{2} \rfloor) \tag{8.6}$$

8.5.2 Delay of VLP Square Root Selection

The selection circuit for VLP square root is computed as:

$$t_{sqrtsel} = T_{BSNR}(f(n)) + T_{MULT}(f(n)) + T_{MUX} + T_{CPA}(n+2) \tag{8.7}$$

where $f(n) = 3n + 1$. The NS→BR converter was described in section 7.1.2 and corresponds to an RCA circuit in terms of area and time. T_{MULT} is going to be a sequential multiplier (LSA consumes less area and is more adequate for FPGAs). The delay of a 2-input vector multiplexer is $T_{MUX} = T_{ILO} + t_{interc}$. Equation (8.5.2) reduces to:

$$t_{sqrtsel} = 2T_{fclt} + (T_{BYP} + T_{NET})(\lfloor \frac{f(n)-3}{2} \rfloor + \lfloor \frac{n-1}{2} \rfloor) + (4 + \lceil \frac{f(n)+1}{2} \rceil)T_{cp} \tag{8.8}$$

and

$$T_{fclt} = T_{OPCY} + T_{NET} + T_{SUM} + t_{interc}$$

Chapter 9

Performance Evaluation

Given the VLP algorithms and area/time estimates of the circuit implementation into 4-input LUT FPGAs we analyze the performance impact of the reconfigurable arithmetic coprocessor in the overall performance of long-precision arithmetic. There are some types of machines that could take advantage of the coprocessor. We chose a high performance workstation as the host and compare the performance of the host alone and the pair host+coprocessor.

As explained before the host executes some tasks that are not worthy to be done at the coprocessor. This concurrent execution of host and coprocessor is difficult to be described by equations. For this reason we propose a model for the system and simulate the model based on the behavior of the VLP algorithms, the tasks performed by the host, and measurements on a real computer. Based on the simulation results we investigate how architecture parameters affect the performance, like bus communication and processor speed.

9.1 Coprocessor Reconfiguration

The reconfiguration time of the coprocessor can be significant. This time is incurred when it is necessary to switch between different parts of the design or when a different arithmetic function is required in the coprocessor. The reconfiguration time (T_{reconf}) in the Xilinx devices for example is several miliseconds in the XC4000 and in the range of μs to ns for the XC6200 series. The larger the FPGA, the larger the time for reconfiguration. Reconfiguration during the VLP operation with these large delays is unacceptable. The impact of the reconfiguration can be minimized by some of the following techniques:

- prediction (or preview) of the next VLP operation. The host processor, knowing that the coprocessor must be reconfigured, can trigger this operation in advance, while other tasks are executed.

- host takes over: knowing that the reconfiguration time is going to degrade the performance, the host goes ahead and executes the operation. During this time the coprocessor can be prepared for the next operation.

- partial reconfiguration: future 4-input LUT FPGAs may have partial reconfiguration feature. FPGAs such as the Xilinx XC6200 allows partial reconfiguration. This characteristic would reduced the reconfiguration time significantly, specially for the proposed VLP algorithms. The VLP operation data paths are alike and the change from one to another is done by proper rewiring and reconfiguration of the selection function circuits only.

For our estimates we assume that the VLP operation is already configured in the coprocessor. The performance impact of reconfigurations can be part of future investigations.

9.2 Coprocessor Model

A high-level model of the coprocessor was already given in Chapter 2. A more detailed block diagram of the coprocessor is shown in Figure 9.1. The main components are:

- *Memory Access Control Block*: this module controls the access to the coprocessor local memory. It is also able to perform DMA operations to read operands from main memory or another coprocessor' memory. This feature is useful when cascading VLP operations over multiple coprocessors. It also holds the access to memory locations that are being accessed by the coprocessor at the same time, based on the status signals from the memory (*busy*).

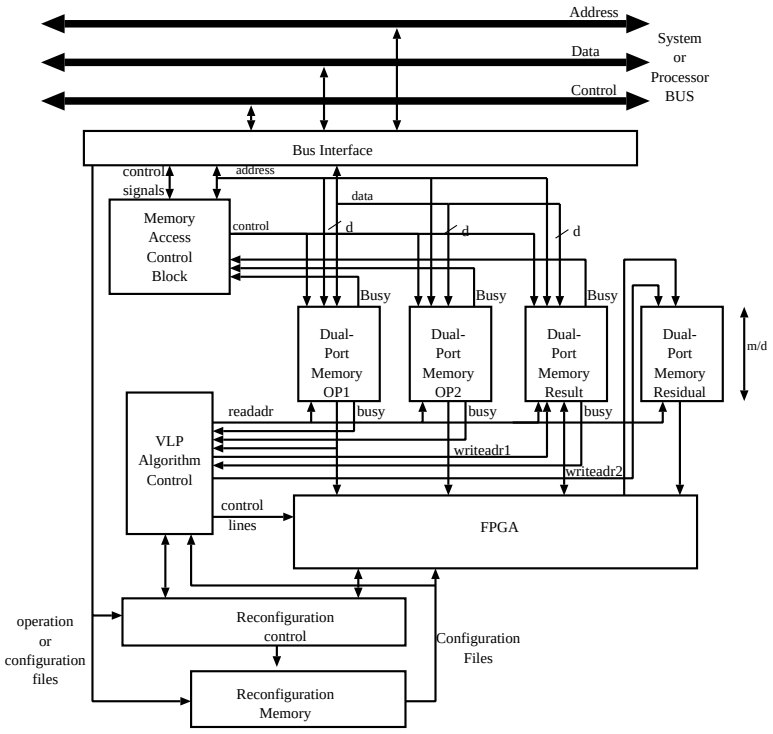

Figure 9.1: More detailed coprocessor model

- *Dual-port Memory elements (Local Memory):* the algorithms proposed for VLP computation will have the best performance if the operands and result be stored in memory components that allow simultaneous access to memory locations. For the operands, dual-port memory is important to decouple the speed of the host processor digit transfers and the coprocessor digit consumption rate. The same is valid for the result dual-port RAM. The residual dual-port RAM is used only by the coprocessor, but in each cycle it is necessary to read and write data to the residual memory, thus, in this case, the use of this type of memory simplifies the design. Each block is capable of access arbitration (signal *busy*) when the access occurs to the same memory address. These memories are also used to store information that are transferred to the coprocessor to execute the required computation like scaling factor for VLP

division and operation's precision. The memory space is viewed differently by the host and the coprocessor. The host sees the accessible memory of the coprocessor as a continuous space. The coprocessor sees each memory element as a separate space, that is accessed in parallel and have the same base address.

- *FPGA:* for this model we assume one FPGA chip. The case of multiple chips (as shown in section 2.1.2) may be considered for future investigation.

- *VLP Algorithm Control:* this block represents the control circuit of the VLP algorithm following the descriptions done in the previous chapters. It is responsible for issuing control signals to the FPGA chip (in order to load values in the proper registers at the proper time) and access the memory elements (reading operand digits and writing result or residual digits). The complexity of the control is slightly affected by the memory organization, that in the model will hold d digits.

- *Reconfiguration Control:* the reconfiguration control is the block responsible for downloading the correct reconfiguration files into the FPGA. The controller is activated by the host to force a reconfiguration, or it can be activated by the VLP Algorithm Control block, when the operation that was requested is not the one loaded into the FPGA.

- *Reconfiguration memory:* stores the configuration files of all VLP operations. It is loaded by the host computer with the possible files and accessed by the Reconfiguration Control block during reconfiguration time.

The model shows only the most important interconnections between the coprocessor component and system/processor bus.

9.3 Model Parameters

A VLP operation can be performed completely by the host in a time:

$$T_{LP} = T_{host} C_{prog} \tag{9.1}$$

where T_{host} is the host clock cycle time and C_{prog} is the average number of cycles needed to execute a program that performs the LP operation.

The bus bandwidth B_H is defined as:

$$B_H = \frac{d}{T_b} \text{ digits/cycle} \tag{9.2}$$

where d is defined based on the coprocessor hardware resources and the word size. The bus transfer unit is assumed to be the same size as the host word. T_b represents the number of processor cycles used for each bus transfer.

The coprocessor Dual-Port memory enables concurrent access to the same memory block when the memory addresses are not the same. We assume that the coprocessor is able to read/write d digits from/to each memory bank. So, based on the block diagram of the coprocessor (and the VLP algorithms described) the bandwidth with the local memory is:

$$B_{LM} = \frac{[d_{NR} + d_{BS}]d}{T_{LM}} \tag{9.3}$$

where

• d_{NR} is the number of digits in non-redundant form that are read(d_{NRR}) or written(d_{NRW}) from/to the local memory during the VLP operation in clock cycle. We consider $d_{NR} = d_{NRR} + d_{NRW}$.

• d_{BS} is the number of digits in redundant form consumed (d_{BSR}) or generated (d_{BSW}) by the coprocessor in each cycle (residual value).The relation $d_{BS} = d_{BSR} + d_{BSW}$ holds.

• T_{LM} is the local memory access time (memory cycle) combined with the FPGA delays.

The value of T_{LM} is defined as a function of the memory parameters and the FPGA I/O characteristics:

$$T_{LM} = max(T_{ICKOF} + T_{LMsetup}, T_{SPD} - T_{LMhold} + T_{LMread}) \tag{9.4}$$

where $T_{LMsetup}$ is the memory setup time during a write cycle,T_{LMhold} is the hold time of data read from the memory after the address change and T_{LMread} is the read access time. The other parameters are related to the FPGA I/O characteristics.

Another constrain relates the memory bandwidth with the data demanded by the VLP operator's data path:

$$B_{LM} \geq \frac{d_{NR} + d_{BS}}{T_{cp}} \tag{9.5}$$

where T_{cp} is the coprocessor cycle time based on the data path maximum delay. This minimum coprocessor cycle is obtained based on the FPGA device used and logical design of the arithmetic operators. Using the structures presented in this thesis with maximum degree of pipelining we obtain a cycle time that is:

$$T_{cpmin} = T_{IHO} + t_{interc} + T_{CKO} \leq T_{cp} \tag{9.6}$$

The combination of Equations (9.3) and (9.5) implies that

$$T_{LM} \leq dT_{cp} \tag{9.7}$$

that gives a lower bound on the number of digits that must be simultaneously read or written to the memory modules in one memory cycle.

An upper bound on the value of d is obtained from the maximum number of user I/O pins in the chip (N_p) as:

$$[(n+1)d_{NR} + 2nd_{BS}]d + N_c \leq N_p \tag{9.8}$$

that results in:

$$d \leq \frac{N_p - N_c}{(n+1)d_{NR} + 2nd_{BS}} \tag{9.9}$$

where N_c is the number of pins used for control signals. The NR representation of a signed digit in radix-r ($n = log_2 r$) takes $n+1$ bits and the redundant representation of the same digit (BS) takes $2n$ bits

In our model we assume that $T_{CP} = K_{CP} T_{host}$.

The number of cycles of each algorithm (C_{cp}) was given in the previous chapters and it is different for each VLP operation. It is computed in terms of the number of coprocessor digits that are required as output precision or the number of digits in the input operands (m).

The total operation time is composed of:

$$TT = C_{cp} T_{cp} + T_{waitCP} + T_{opt} + T_{res} + T_{FPO} \tag{9.10}$$

where

- T_{opt} represents the operand transfer time. It includes the time that the host needs to adjust the digit format and write to the coprocessor's local memory. This operation is overlapped with the VLP operation in the coprocessor. This parameter considers only the time when the coprocessor is waiting for data from the host before the operation starts.
- T_{res} is the result transfer time. Includes the time required by the host to read the coprocessor result, perform OFC and convert from expanded digit format to compact SM format (used in the software). Again, this operation can be overlapped with the coprocessor operation, so this parameter considers only the time that the host takes after the last result digit is generated by the coprocessor.
- T_{FPO} is the time for other tasks required for proper FP operation.
- T_{waitCP} includes the waiting time after the coprocessor starts its operation.

The coprocessor area depends on the VLP operation. We assume the control circuit is implemented in another space. The main components of the circuit synthesized into the FPGA is shown in Figure 9.2. The blocks in the figure are:

- input buffer: stores the digits received from the local memory until they are used by the data path. It is a dual register structure. While one register

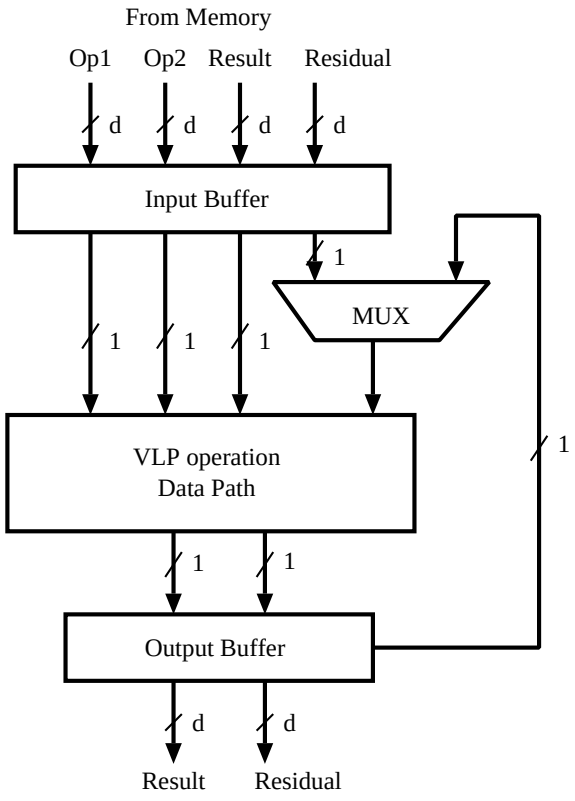

Figure 9.2: Block diagram of the circuits inside the FPGA

stores the information being read from the memory, another register is used to shift the digits into the data path. The number of input buffers depend on the number of digits read in each iteration: d_{NRR} and d_{BSR}.

- output buffer: stores the digits that are generated by the data path, until they form a group of digits that can be stored back into the memory. The output of the shift register is stored into another register that keeps the value during a writing cycle. The number of registers required depends on the number of digits to be written into memory: d_{NRW} and d_{BSW}.

- forwarding multiplexer: this component is required to forward the information from the output buffer to the data path inputs when digits already in the

Operation	d_{NRR}	d_{NRW}	d_{BSR}	d_{BSW}
VLP Multiplication	2	1	1	1
VLP Division	3	1	1	1
VLP Square Root	2	1	1	1

Table 9.1: Maximum number of digits read simultaneously in each iteration

Component	Area (CLBs)
Input buffer	$2(n+1)+2n$
Output buffer	$(n+1)+2n$
Forwarding Multiplexer	$\frac{n+1}{2}+n$
Data path of VLP SQRT	Table 8.9

Table 9.2: Maximum area required to implement the VLP algorithms in FPGAs

output buffer were not yet written to the local memory. Two-input vector multiplexers are used, and their size depend on the number of output digits that should feedback to the data path inputs.

- Data path: already discussed in previous chapters, constitutes the main data transformation element for the VLP algorithms.

Table 9.1 shows the maximum number of digits read or written in a single iteration for each VLP operation and Table 9.2 shows the maximum area required by the circuit components. The VLP square root was considered in the estimate since it is the one that consumes the larger area among the VLP operations. From the table we determine the digit radix (if the amount of reconfigurable resources is fixed) or the FPGA device that is more adequate for the coprocessor.

9.4 Measurements

The total time to execute a long precision computation in a general processor running a software library was obtained using Quantify [Pur], a tool to measure

limbs (32 bits)	Multiplication	FP Multiplication	FP Division	FP Square Root
15	9025	10137	12095	20944
20	15930	17371	19898	35275
30	35590	37690	40421	64343
40	48679	51319	69428	109848
50	75439	78761	106206	163864
100	228209	234684	400366	535943
200	688283	701350	1567276	1824409
300	1226662	1246286	3492902	3707245

Table 9.3: Number of cycles for long-precision operations in GMP (C_{prog})

the program run time in terms of the processor cycle. We selected an UltraSparc[1] machine as the host computer. The host is a RISC processor that runs at 167 MHz. The test programs use the GMP library version 2.0.2 [Tor93] that is claimed to be one of the fastest libraries for LP computations. These programs are listed in Appendix A. The number of cycles to execute the main routines for each LP algorithm in GMP are shown in Table 9.3. The same data is also plotted in Figure 9.3, converting limbs to digits in radix $r = 2^{15}$. The use of this radix for the coprocessor is justified later.

Based on the description of the FP tasks done by the host (section 7.2) and the analysis of the GMP library routines, we concluded that the manipulation of the significand and exponents is basically the same for GMP and the coprocessor+host system. The time related to this task was also measured, as shown in Table 9.4. The value of T_{FPO} depends on the operation and the number of coprocessor digits in the significand (m).

Measurement of other tasks performed by the host during VLP operation are shown in Table 9.5. These measures use the fact that limbs have 32 bits and the coprocessor digit has 15 bits. The reason for 15-bit digits will become clear later. The numbers on the table show that the average numbers of cycles per digit are:

[1]UltraSparc is a trademark of SUN Microsystems

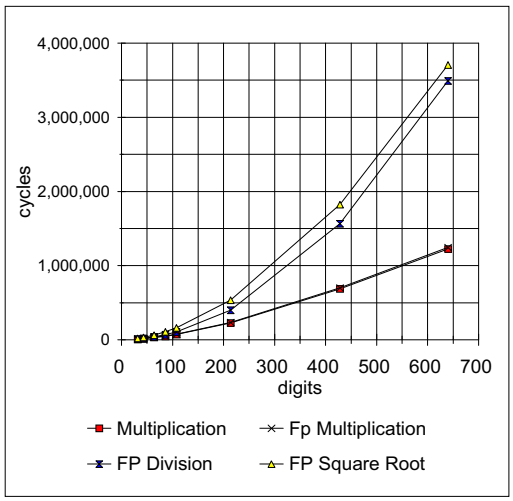

Figure 9.3: Number of cycles for Long-precision Operations

FP Operation	T_{FPO} (cycles)
Multiplication	$1.9m + 17$
Division	$4m + 82$
Square-root	$6m + 204$

Table 9.4: Significant and exponent manipulation time in GMP

$T_{compress} = 38$ cycles/dig,$T_{OFC} = 119$ cycles/dig and the expansion time $T_{gop} = 50$ cycles/dig.

The case of prescaling at the host (VLP division) was also implemented and measured in the host computer. An on-line prescaler was implemented in software, for digits in the same radix as the coprocessor digits. The pre-scaling has a setup time $T_{prescalsu} = 2257$ cycles and after that takes $T_{prescal} = 277$ cycles per coprocessor digit. The time to compute the scaling factor is given as $T_{factor} = 893$ cycles.

Limbs	Digits	Digit Compression	Digit Expansion	OFC
15	32	1230	1646	3813
20	43	1671	2161	5229
30	64	2437	3230	7589
40	86	3263	4260	10185
50	107	4085	5329	12781
100	214	8091	10596	25289
200	427	16155	21169	50541
300	640	24163	31742	75557

Table 9.5: Other tasks performed by the host during VLP operations

9.5 Performance Estimate

This performance estimate is done first determining the area and cycle time for the VLP operations at the coprocessor level. This estimate considers:

- one FPGA chip of a given technology, in this case Xilinx. Based on the model parameters presented in Section 9.3 we select the most adequate chip to fit the VLPA circuits. It is also possible to determine the number of coprocessor digits that must be accessed per memory cycle.
- dual port memory components with small access time. Dual port RAMs inside the chip would have a small access time but these memories would store a small number of digits, limiting the precision of the VLP operators.

The second phase of this estimate is done considering measurements of the other tasks done by the host: digit preparation, OFC and digit compression. These measures are simulated using the operation model shown in Figure 9.4. The VLP computation is performed by three different tasks:

1. the host translates the LP format to the coprocessor digit format and sends the information to the coprocessor. The bus transfer time is modeled as an extra delay for each word transferred to the coprocessor. Each word may carry 2 or more coprocessor digits. The exact number is defined later.

2. the coprocessor consumes the operand digits and generate result digits. If the operand digits are not available, the coprocessor computation waits until the necessary digits arrive. Result digits are immediately available to the host.

3. the host reads the result digits. This task starts only after task (1) is complete. For each result digit read, an extra time corresponding to the bus transfer delay is added. We assume the worst case situation when only one digit can be read in each cycle. It may be the case that more than one digit can be read at once, that is a better case them the one considered.

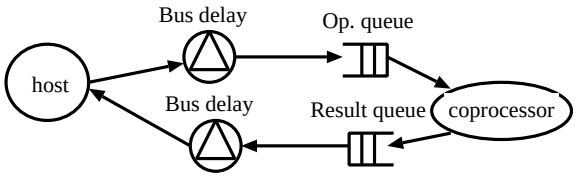

Figure 9.4: Model for host coprocessor operation

9.5.1 Coprocessor Area/Time

For this estimate we consider the components shown in Table 9.2. Based on the VLP algorithms shown in the previous chapters, these are the parameters for the system:

- the host is able to read/write 32-bit words to the coprocessor memory. Based on the discussion in section 7.2.3 the minimum number of digits per word is 2 and the largest coprocessor digit radix is 2^{15}.

- based on the choice of $n = 15$, the area needed by the VLP square root case (the most area consuming operation), is 1446 CLBs (from Table 9.2). For this area, the chip XC4044 can be used. We chose a fast component in this family, the XC4044XL, that has the parameters shown in Appendix A.

Parameter	Value
n	15
d	2
T_{cp}	10ns

Table 9.6: Coprocessor parameters

- from the data sheet and equation 9.6 we obtain $T_{cpmin} = 2.2 + 2 + 1.6 = 5.6ns$
- using a dual-port RAM as described in [CY795] the memory parameters are $T_{LMread} = 15ns$, $T_{LMhold} = 3ns$ and $T_{LMsetup} = 10ns$, that lead to the definition of

$$T_{LM} = max(7.8 + 10, 8.0 - 3 + 15) = 20ns \tag{9.11}$$

- based on equation (9.7) we obtain $d = 2$ and $T_{cp} = 10ns$. The value of $d = 2$ also satisfies equation (9.9) for the values $N_p = 320$, $d_{NR} = 4$ and $d_{BS} = 2$, leaving $N_c = 72$ pins for control signals.

The coprocessor parameters are summarized in Table 9.6.

All parameters listed in Table 9.7 are obtained from the VLP algorithms or the area/time estimates presented above. Other considerations about the performed evaluation are:

- The BS→NR converter used by the VLP multiplier to transmit the second half of the double precision product will have a delay of 9.35ns, that is less than T_{cp}. Thus, during the transmission of the residual to the host, a rate of one digit per cycle can be attained.
- T_{init} corresponds to the initialization time of the algorithms, before the main iteration steps take place. This time is used only for the VLP square root. It corresponds to the preparation of the residual, based on the estimate of the output. The initialization time is 63 coprocessor cycles, based on equation provided in Chapter 6 and the FPGA time characteristics ($63K_{cp}$ host cycles).

- The simulation considers pipelined data paths and non-overlapped iterations.
- The number of cycles to complete one iteration when the precision of the input operands is i digits is given by the equation (9.12), that is the same for all VLP operations.

$$T_{rg}(i) = \begin{cases} K_{cp}(i+21) & \text{if } i \leq d \\ K_{cp}(2d-i+22) & \text{otherwise} \end{cases} \quad (9.12)$$

- The delay of the selection functions for division and square root are estimated using the equations (8.6) and (8.8) as 23 coprocessor cycles and 30 coprocessor cycles, respectively.

We consider that the bus that interconnects the coprocessor and host has a delay of $T_b = 32$ cycles. This parameter reflects the Sbus transfer time. Each individual transfer, not in burst mode, takes 4 cycles [Lyl92]. The Sbus uses a 25MHz clock frequency that results in 160ns per 32-bit word transfer. This corresponds to 16 coprocessor cycles or $16K_{cp} = 32$ host cycles.

9.5.2 Model Simulation

A simulation program was implemented in C++, based on the model described and the coprocessor parameters listed in Table 9.7. Many parameters are based on measurements in the host computer as described in Section 9.4. The time is computed in terms of host cycles.

The simulation results for some values of precision are shown in Tables 9.8, 9.9, 9.10, 9.11 and 9.12. The VLP integer multiplication generates an output that is twice the precision of the operands. Floating-point operations generate results with precision equals to the precision of the operands. The first case of VLP FP division considers the scaling of operands at the host. The second case is the situation when the pre-scaling is done by the coprocessor.

The speedup obtained with the pair host+coprocessor is shown in Figure 9.5. It is computed in respect to the execution of a similar computation at the host,

Parameter	Description	Value (cycles)
T_{gop}	interval for coprocessor digit generation	50
N_{dig}	number of digits transferred in each bus cycle	2
T_b	number of host cycles for each bus cycle	32
K_{cp}	ratio between the host and coprocessor cycle time	2.0
m	number of digits in one operand or result	varies
T_{ofc}	average number of cycles per digit of OFC	119
$T_{compress}$	average time to compress one coprocessor digit	38
T_{init}	initialization phase of the VLP algorithm	varies
T_{sel}	selection function delay	varies
$T_{rg}(i)$	time between the generation of two result digits	Eq. (9.12)
T_{FPO}	time for significand and exponent manipulation	Table 9.4
$T_{prescalsu}$	set up time for prescaling of 1 operand (division)	2257
$T_{prescal}$	time for prescaling per digit	277
T_{factor}	computation of division scaling factor	893

Table 9.7: Parameters for the system

# digits ($r = 2^{15}$)	Total time	CP time (useful)	Waiting time	
			Host	CP
32	13405	2424	0	856
43	18059	3744	0	856
64	26653	6036	0	856
86	35761	11226	0	856
107	44555	16224	0	856
214	98161	55386	9408	856
427	285388	201504	108353	856
640	563353	438360	298236	856

Table 9.8: Host+coprocessor operation - Integer Multiplication

# digits ($r = 2^{15}$)	Total time	CP time (useful)	Waiting time	
			Host	CP
32	8302	2016	0	1188
43	11250	2946	0	1404
64	16587	5056	0	1348
86	22282	7740	0	902
107	27819	10754	0	902
214	55422	32956	0	902
427	130869	111234	20202	902
640	263014	234880	97301	902

Table 9.9: Host+coprocessor operation - VLP FP multiplication

# digits ($r = 2^{15}$)	Total time	CP time (useful)	Waiting time	
			Host	CP
32	31601	3008	0	18304
43	41220	4042	0	25588
64	57681	6236	0	37618
86	75611	9008	0	50620
107	93380	12106	0	63412
214	179931	34736	0	121468
427	354180	113866	0	221812
640	527121	238364	0	298258

Table 9.10: Host+coprocessor operation - VLP FP division(pre-scaling at the host)

# digits ($r = 2^{15}$)	Total time	CP time (useful)	Waiting time	
			Host	CP
32	9797	3632	470	2788
43	12834	4798	536	3922
64	18341	7244	662	5796
86	24215	10280	794	7864
107	29964	13630	962	9938
214	68330	37544	11501	19896
427	179084	119230	66562	40018
640	335006	246284	166991	59940

Table 9.11: Host+coprocessor operation - VLP FP division (pre-scaling at the coprocessor)

# digits ($r = 2^{15}$)	Total time	CP time (useful)	Waiting time	
			Host	CP
32	8620	3582	0	74
43	11613	4770	0	74
64	17036	7078	0	74
86	22822	9894	0	74
107	28445	13034	0	74
214	56486	35878	0	74
427	135183	115434	22578	74
640	269479	240358	100955	74

Table 9.12: Host+coprocessor operation - VLP FP square root

using GMP. Observe that the speedup for square root is the best among all VLP operations. This result is expected since the square root operation in GMP is the one that takes more time, while for the coprocessor, the VLP square root is just slightly slower than the others, caused by the more complex selection function.

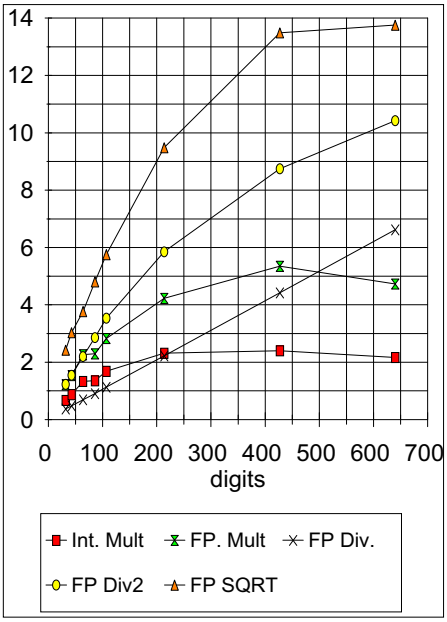

Figure 9.5: Speedup obtained with Host+coprocessor over Host alone

The second best speedup is obtained with VLP division when the coprocessor executes the prescaling locally. VLP division with pre-scaling done by the host will show some benefit only for more than 100 digits of precision. For low values of precision, the time used by the host to manipulate the coprocessor digits and transfer them, kills the advantage of using the coprocessor.

Observe that the simulation was done for $K_{CP} = 2$, that corresponds to a host computer running at 200MHz (and not 167MHz).

As the precision of the operands increase, more work is left for the coprocessor and the speedup increases. After a certain point, the speedup starts to drop. A qualitative view of the speedup behavior is shown in Figure 9.6. The figure

illustrates the fact that there is a range of application for the VLP algorithms. Software algorithms have better asymptotic time but more overhead than the VLP

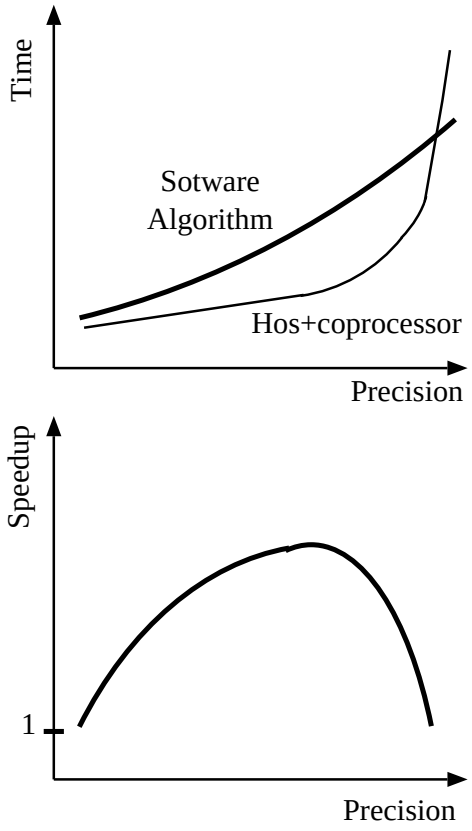

Figure 9.6: Qualitative behavior of the speedup

algorithms for a large range of precision. The speedup increases as the number of digits manipulated by the coprocessor increases. After a certain number of digits, the speedup starts to decrease as a result of the asymptotic time of the VLP hardware algorithm. Thus, given a pair host+coprocessor the software responsible for the manipulation of the coprocessor should decide if the coprocessor is going to be used.

Another reason for this behavior is the proportion of total time used by the coprocessor. For small number of digits, the host computation time dominates

the total time. In this case the total time has a linear behavior. As the precision increases, the coprocessor is responsible for most of the total time and the total computation delay shows a $O(n^2)$ behavior. The proportion of the total time used by the coprocessor, with its waiting time, is shown in Figure 9.7, for the case VLP multiplication.

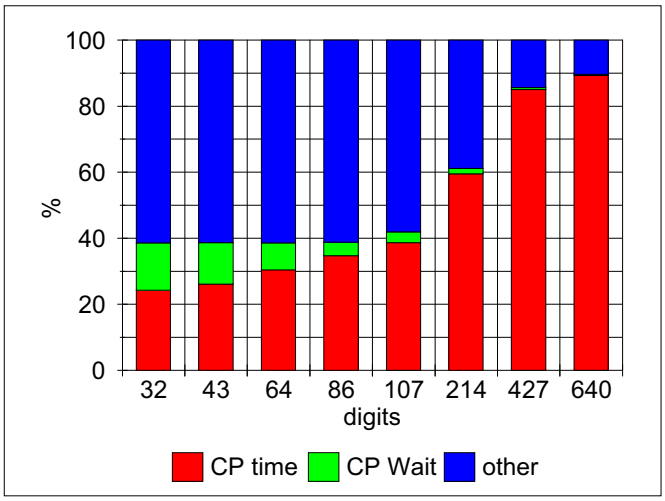

Figure 9.7: Proportion of time used by the coprocessor

Simulations performed with other values for the parameter T_b showed that the speedup is only marginally affected by variations in the bus transfer time. The variation of the parameter K_{CP} reflects the case when a faster host computer is used. Table 9.13 shows the system behavior for different values of K_{CP} for $m = 427$ digits and a bus with $T_b = 32$ cycles of transfer time. The VLP FP multiplication is considered. The coprocessor is kept at the same clock frequency. The data shows that even for very fast host computers, the use of the coprocessor will provide a reasonable speedup. Observe that division and square root have better performance than the VLP multiplication considered in this analysis.

K_{CP}	Total Time	Speedup	Host clock frequency (MHz)
2	130869	5.4	200
3	185025	3.8	300
4	240187	2.9	400
5	295766	2.4	500

Table 9.13: Variation in the Speedup with the Host speed

Chapter 10

Conclusion and Future Research

Throughout this thesis we have shown the advantages of using on-line arithmetic in the implementation of hardware algorithms for VLP computations. For the same digit size, the number of cycles required in VLP operations is less than the number of cycles of other algorithms used for hardware implementation.

The designs presented in this dissertation have been developed to be easily applied to any digit radix. Estimates of area and time provide the required tools to evaluate the applicability of the technique to a particular design space. Based on the estimates, the designer can select the proper chip to be used or select the digit radix that best fits in the hardware resources available.

A radix-256 VLP multiplier was implemented in the EVC1 board [Cor95a]. The design was specified in VHDL and sythesized using Powerview[1] and XACT[2] FPGA development software. The implementation was integrated to GMP version 2.0.2 using the H.O.T. technology [Cor95b]. The EVC1 board was connected to a Sun Sparcstation1. This experiment was used as a proof of concept and provided the required background for the preparation of the area/time estimates given in Chapter 8.

All algorithms were implemented and tested in C++. The VLP data paths were described as a network on on-line modules, the same way as in the hardware implementation. High-radix on-line operands were also developed. Using the simulation tools it was possible to verify most of the parameters for the VLP algorithms included in this thesis. Most tests were performed in radix 10 or 100

[1]CAD tool from Viewlogic.

[2]CAD tool for FPGA synthesis from Xilinx.

to simplify verification of the circuit operation.

The evolution of the technology leads to an increase in area and speed of FPGA devices. This tendency has been shown over the years. The extra space in the FPGA may be used in two different ways:

- store the residual value in the VLP computations or even the operands and result.

- extract more parallelism.

As discussed before, the implementation of dual-port RAM in FPGAs is practically limited to a small addressing space. More memory locations can be implemented at the cost of poor performance. However, an internal memory element would have a better access time and less communication delay. Thus, this option must be reevaluated for future technologies.

To extract more parallelism from the VLP algorithms the designer can use the extra hardware to increase the radix of the digits used in the coprocessor or have more than one iteration being executed at the same time. The strategy of unfolding iterations is possible for all VLP operations. The result digits produced by one data path is fed into another data path. Specially for VLP multiplication, which has a selection function that does not consume time from the iteration, this approach would be very efficient. The linear organization of FPGA chips proposed in Section 2.1.2 could be used to pipeline the various instances of the VLP multiplication data path. VLP division and square root would take less advantage of this strategy given that their selection functions are more complex than the one used for VLP multiplication, and would fatally affect the time to trigger trigger consecutive unfolded iterations.

Even though this study was concentrated on FPGA technology, the use of ASIC technology would be possible.

10.1 Research Contributions

This work provides the following main contributions:

1. provide algorithms for VLP operations based on high-radix on-line arithmetic.

2. proposes solutions for the problem of selection function in high-radix on-line operations. As a result of this study the dissertation provides the required conditions for VLP computations.

3. generalizes the estimation of area and time of the proposed algorithm implementations. Such estimates are used as the basis for performance evaluation of the coprocessor speed up working cooperatively with the host computer.

4. shows that the cooperative execution of tasks among the host and coprocessor results in significant overall speedup over the execution of the same task by software in the host alone, for a large range of precision.

5. proposes new logical designs for arithmetic structures such as the parallel CS to BS converter, the serial digit by vector multiplier and the on-line recoder for the VLP multiplier.

10.2 Future Research

During this research we observed some direction that could be followed in future research work:

- Software implementation of the on-line algorithm for VLP operations. Based on the investigation in the area of software for long-precision computation it became clear that there is a space for the application of the on-line technique for some ranges of precision. The implementation should be as efficient as the ones already available for other algorithms. This requirement would generate

an implementation that could be fairly compared with the others. The implementation of the on-line operations in assembly code for the target system would be important.

- Analyze the situation when more than one VLP data path is available for the computation or when the design must be partitioned into multiple chips. The behavior of the VLP on-line operations have a variable latency to generate and consume digits. This variable timing results in synchronization problems that should be studied.

- Investigate the case when more than one coprocessor is available. For this case the variable latency to generate output digits will also play a role. Coprocessors working in an overlapped mode would not have a good match of speed. Besides that, some tasks executed by the host would also affect the performed of this concurrent operation.

- Investigate the impact of partial reconfiguration in the system operation. Analysis the implementation of the same algorithms using another FPGA, such as the XC6200 would be interesting to obtain data for the evaluation of the time overhead to perform switching between VLP operations.

- Researchers claim that FPGAs will include special circuitry for multipliers embedded in the matrix of CLBs. Multiplication is the most area consuming circuit in the VLP data paths. It would be an important to investigate the impact of these dedicated multipliers in the design and performance of the VLP operations.

APPENDIX A

Timing Characteristics of the XC4000 FPGAs

XC4000XL-1		
CLB Switching Characteristics		
Description	Symbol	time [ns]
Combinational Delays		
F/G inputs to X/Y outputs	T_{ILO}	1.3
F/G inputs via H to X/Y outputs	T_{IHO}	2.2
C inputs via DIN through H to X/Y outputs	T_{HH2O}	2.0
CLB Fast Carry Logic		
Operand inputs to COUT	T_{OPCY}	2.0
Add/Subtract input to COUT	T_{ASCY}	2.5
Initialization inputs to COUT	T_{INCY}	1.5
CIN through FGs[1] to X/Y outputs	T_{SUM}	2.4
CIN to COUT, bypass FGs	T_{BYP}	0.2
Carry network delay (COUT to CIN)	T_{NET}	0.25
Sequential Delays		
Clock K to outputs Q	T_{CKO}	1.6
Setup time before Clock K		
F/G inputs	T_{ICK}	0.9
F/G inputs via H	T_{IHCK}	1.7
C inputs via DIN	T_{DICK}	0.7
Average interconnect delay	t_{interc}	2.0
Input/Output Timing Characteristics (XC4044XL[2])		
Description	Symbol	time [ns]
Global Low skew clock to Output using Output FF[3]	T_{ICKOF}	7.8
Input Setup time, using Global Low Skew clock and IFF[4]	T_{SPD}	8.0
Input Hold Time, using Global Low Skew clock and IFF	T_{PHD}	0

[1] Function Generators.

[2] XC4044 has 1600 CLBs and 320 user I/O pins.

[3] Load capacitance of 50pF.

[4] IFF - Input Flip-flop.

APPENDIX B

Test Program for LP Operations using GMP version 2.0.2

GMP uses limbs of 32 bits. A limb is a high radix digit that is manipulated by the arithmetic algorithms.

We have made four types of measurements: integer multiplication, floating point multiplication, division and square-root. Using Quantify we measure the number of cycles for the kernel of each operation.

B.1 Test Program for LP Integer Multiplication

This program computes the product of two integers with 100 limbs of precision (3200 bits). As the software routines are optimized to skip over zeroes, many runs are done to reduce obtain an average number of cycles for the program execution. The kernel of this operation is the procedure mpn_mul (called by mpz_mul).

B.2 Test Program for Floating Point Operations

This program computes the product of two floating point numbers with 100 limbs in the significand field (3200 bits). The randomization routines generate patterns with long sequences of zeroes, for this reason, the program uses the result of previous operations (that has shorter sequences of zeroes) as operands for the next ones. The difference between the FP multiplication, division and square root test program is the set of instructions inside the for loop. The test program used for FP multiplication is shown next and was used to measure the number of cycles in the mpf_mul routine.

To measure the number of cycles of FP division (routine mpf_div) a similar program was used replacing the the instructions {mpf_mul(x,u,v); mpf_mul(v,x,u);

```
#include <stdio.h>
#include "gmp.h"
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "gmp-impl.h"

void main() {
 int num_limbs_prog;
 int i, j, num_runs;
 MP_INT integ1, integ2;
 MP_INT prod;

num_limbs_prog=100;
num_runs=100;
 mpz_init (&integ1);
 mpz_init (&integ2);
 mpz_init (&prod);
for (i=0;i<num_runs;i++)
{
 /* initialize the random values of integ1 and integ2 */
 mpz_random(&integ1,num_limbs_prog);
 mpz_random(&integ2,num_limbs_prog);

 mpz_mul(&prod,&integ1,&integ2);
};

}
```

```
#include <stdio.h>
#include "gmp.h"
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "gmp-impl.h"
#include "urandom.h"

#ifndef SIZE
#define SIZE 16
#endif

main (argc, argv)
     int argc;
     char **argv;
{
  mp_size_t size = 300; /* size in limbs */
  mp_exp_t exp;
  int i;
  mpf_t u, v, x;
  int num_runs;

  mpf_set_default_prec (size*32); /* the library use lims of 32 bits */
  mpf_init (u);
  mpf_init (v);
  mpf_init (x);
  num_runs = 100;

  /* initialize operand u */
  exp = urandom () % SIZE;
  mpf_random2 (u, size, exp);

  /* initialize operand v */
  exp = urandom () % SIZE;
  mpf_random2 (v, size, exp);
  for (i=0;i<num_runs;i++)
{
      mpf_mul (x, u, v);
      mpf_mul (v, x, u);
      mpf_mul (u, x, v);
};

  exit (0);
}
```

mpf_mul(u,x,v);} with {mpf_div(x,u,v); mpf_div(v,x,u);}.

For LP square root we collected the number of cycles of the routine mpf_sqrt. The test program is similar to the one for FP multiplication replacing the three multiplication instructions with the instructions {mpf_sqrt(x,u); mpf_sqrt(u,x);}.

APPENDIX C

Digit radix transformation using BS Code

Theorem 1 *Given a value x represented by a signed-bit vector $(x_{n-1}, ..., x_1, x_0)$, with $x_i \in \{-1, 0, 1\}$, it is always possible to transform the representation of x to another radix $r = 2^k$, $k > 1$ just splitting the signed bits into groups of k bits, from right to left:*

$$x = (x_{n-1}, ..., x_1, x_0)_2 = (..., (x_{2k-1}, ..., x_k), (x_{k-1}, x_{k-2}, ..., x_1, x_0))_k$$

where each group $(x_{2ki-1}, ..., x_{ki})$, $0 \leq i \leq \lceil \frac{n}{k} \rceil$, represents a radix-r signed digit in the maximally redundat set $\{-(r-1), ..., (r-1)\}$.

Proof: When the signed-bit vector is split into groups of k signed bits, each group of bits will have a value:

$$y_i = \sum_{j=0}^{k-1} y_{ij} 2^j \tag{C.1}$$

with $y_{ij} \in \{-1, 0, 1\}$ and $y_i \in \{-(r-1), ..., (r-1)\}$, since $r = 2^k$.

Lets show that the value of the vector formed by digit y_i represent the value of x. The value of the vector $(y_d, ..., y_1, y_0)$ in radix r is computed as:

$$y = \sum_{i=0}^{d} y_i r^i = \sum_{i=0}^{d} y_i 2^{ki} \tag{C.2}$$

where $d = \lceil \frac{n}{k} \rceil - 1$.

Substituting the value of y_i in equation (C.2) by the value given in equation (C.1) we obtain:

$$y = \sum_{i=0}^{d} (\sum_{j=0}^{k-1} y_{ij} 2^j) 2^{ki} \tag{C.3}$$

From the relation between x and y we know that $x_{ki+j} = y_{ij}$, and based on the signed digit representation we have $x_i = 0$ for $i > n$. Thus,

$$y = \sum_{i=0}^{d} \sum_{j=0}^{k-1} x_{ki+j} 2^{ki+j} = \sum_{i=0}^{n} x_i 2^i = x \tag{C.4}$$

Bibliography

[75485] ANSI/IEEE Std 754-1985. Standard for Binary Floating-Point Arithmetic. In *SIGPLAN Notices*, volume 22-2, pages 7–18, 1985.

[AACE94] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding Lines and Building Pyramids with Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–161, Napa Valley, California, April 1994.

[ACC96] O. T. Albaharna, P. Y. K. Cheung, and T. J. Clarke. On the Viability of FPGA-Based Integrated Coprocessors. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 206–215, 1996.

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Computer Science and Applied Mathematics. Academic Press, 1983.

[AH93] P. M. Athanas and F. S. Harvey. Processor Reconfiguration through Instruction-Set Metamorphism. *IEEE Computer*, pages 11–18, March 1993.

[Atk70] D.E. Atkins. Design of the arithmetic units of ILLIAC III: use of redundancy and higher radix methods. *IEEE Transactions on Computers*, 1970.

[Bai93] D. H. Bailey. A Portable High Performance Multiprecision Package. Technical Report Technical Report RNR-90-022, RNR, 1993.

[Bau95] C. Baumhof. A New VLSI Vector Arithmetic Coprocessor for the PC. In *IEEE 12th Symposium on Computer Arithmetic*, pages 210–215, 1995.

[Car89] T. M. Carter. CASCADE: Hardware for High/Variable Precision Arithmetic. In *IEEE 9th Symposium on Computer Arithmetic*, pages 184–191, 1989.

[CB92] C. E. Cox and W. E. Blanz. Ganglion - A fast field-programmable Gate Array implementation of a Connectionist Classifier. *IEEE Journal of Solid-State Circuits*, 27(3):288–299, Mar. 1992.

[CHH83] M. Cohen, T. Hull, and V. Hamacher. CADAC: A Controlled-Precision Decimal Arithmetic Unit. *IEEE Transactions on Computers*, C-32(4):370–377, 1983.

[Cho80] C. Chow. *A Variable Precision Processor Module*. PhD thesis, University of Illinois at Urbana-Champaign, 1980. UIUCDCS-R-80-1032.

[Cho83] C. Chow. A Variable Precision Module. In *Int. Conference on Computer Design: VLSI in Computers*, pages 692–695, 1983.

[Com90] P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4), 1990.

[Coo80] J. T. Coonen. An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic. *IEEE Computer*, pages 68–79, January 1980.

[Cor95a] Virtual Computer Corporation. *Engineer's Virtual Computer - EVC1s Users Guide*, first edition, 1995.

[Cor95b] Virtual Computer Corporation. *HOT - Hardware Object Technology - Programming Guide*, first edition, 1995.

[CY795] Cypress Semiconductor Corporation, San Jose, CA. *Data Sheet: 1Kx8 Dual-Port Static RAM*, 1995.

[DeH96] DeHon, André. Reconfigurable Architectures for General-Purpose Computing. Technical Report 1586, MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996.

[DMV94] M. Daumas, J. M. Muller, and J. Vuillemin. Implementing On-Line Arithmetic on PAM. In *4th Int. Workshop on Field Programmable Logic and Applications*, pages 196–207, 1994.

[EL87] M. D. Ercegovac and T. Lang. On-the-fly Convertion of Redundant into Conventional Representations. *IEEE Transactions on Computers*, C-36(7):895–897, 1987.

[EL88] M. D. Ercegovac and T. Lang. On-line Arithmetic: A Design Methodology and Applications in Digital Signal Processing. In *IEEE Workshop on VLSI Signal Processing*. IEEE, 1988.

[EL94] M. D. Ercegovac and T. Lang. *Division and square root: digit-recurrence algorithms and implementations.* Kluwer Academic Publishers, 1994.

[EL96] M. D. Ercegovac and T. Lang. On Recoding in Arithmetic Algorithms. *Journal of VLSI Signal Processing*, (14):283–294, 1996.

[Ely93] J. E. Ely. The VPI Software Package for Variable Precision Interval Arithmetic . In *Interval Computations*, pages 135–153, 1993.

[Erc84] M. D. Ercegovac. On-line Arithmetic: An Overview. In *Real Time Signal Processing VII - 495*, pages 86–93. SPIE, 1984.

[ET87] M. D. Ercegovac and K. Trivedi. On-line Operations. *IEEE Transactions on Computers*, C-36(7):895–897, 1987.

[GHM89] A. Guyot, Y. Herreros, and J. M. Muller. JANUS, an On-line Multiplier/divider for manipulating large numbers. *IEEE 9th Symposium on Computer Arithmetic*, pages 106–111, 1989.

[GKC+94] D. Galloway, D. Karchmer, P. Chow, D. Lewis, and J. Rose. The Transmogrifier: The University of Toronto Field-Programmable System. Technical Report CSRI-306, University of Toronto, Canada, June 1994. ftp://ftp.csri.toronto.edu/csri-technical-reports/306.

[Gui] S. Guicionne. List of fpga-based computing machines. Hyper text link:http://uts.cc.utexas.edu/ guiccione/HW_list.html.

[HCH91] T.E. Hull, M. S. Cohen, and C. B. Hall. Specification for a Variable-Precision Arithmetic Coprocessor . In *IEEE 10th Symposium on Computer Arithmetic*, pages 127–131, 1991.

[Hoa93] D. T. Hoang. Searching Genetic Databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, April 1993.

[HP95] R. Hartley and K. K. Parhi. *Digit-Serial Computation.* Kluwer Academic Publishers, 1995.

[Hsu96] C.-Y. Hsu. Variable Precision Arithmetic Processor in FPGAs. Master's thesis, University of Toronto, 1996.

[KM81] U. W. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice.* Computer Science and Applied Mathematics. Academic Press, 1981.

[Kno91] A. Knofel. Fast Hardware Units for the Computation of Accurate Dot Products. In *IEEE 10th Symposium on Computer Arithmetic*, pages 70–74, 1991.

[Knu69] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Publishing Co., 1969.

[Kor93] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[Kwa93] K. Kwang. *Advanced Computer Architecture with Parallel Programming*. McGraw-Hill, Inc., preliminary edition edition, 1993.

[LM92] Tomás Lang and Paolo Montuschi. Higher Radix Square Root with Prescaling. *IEEE Transactions on Computers*, 1992.

[Lou94] M. E. Louie. *Variable Precision Arithmetic with Lookup Table Based Field Programmable Gate Arrays*. PhD thesis, UCLA, 1994.

[Lyl92] James D. Lyle. *Sbus: Information, Applications and Experience*. Springer-Verlag, 1992.

[Lyn95] T. Lynch. High Radix On-line Arithmetic for Credible and Accurate General Purpose Computing. In *Real Numbers and Computers... Les Nombres Reels et L'Ordinateur*, pages 78–89, Ecole des Mines de Saint-Etienne, France, 1995.

[Mat91] D. W. Matula. Design of a Highly Parallel IEEE Floating Point arithmetic unit. In *Proc. Symp. Combinatorial Optimization Sci. and Technology*, 1991.

[MM94] V. Menissier-Morain. *Arithmetic Exacte: Conception, Algorithmique et Performances d'une Implementation Informatique en Precision Arbitraire*. PhD thesis, L'Universite Paris VII, France, December 1994.

[Moo79] R. E. Moore. Methods and Applications of Interval Analysis. In *Studies in Applied Mathematics*, Philadelphia, 1979. SIAM.

[MRR91] M. Muller, C. Rub, and W. Rulling. Exact Accumulation of Floating-Point Numbers. In *IEEE 10th Symposium on Computer Arithmetic*, pages 64–69, 1991.

[Neu90] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

[P+94] W. H. Press et al. *Numerical Recipes in C: the art of Scientific Computing.* Cambridge University Press, 2nd edition, 1994.

[PA96] K. J. Paar and P. M. Athanas. Implementation of a Finite Difference Method on a Custom Computing Platform. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, pages 44–53, Boston, November 1996. SPIE.

[Pet95] Petersen, Russell J. An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing. Master's thesis, UCLA, Electrical Engineering, 1995.

[Pri91] D. M. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. In *IEEE 10th Int. Symposium on Computer Arithmetic*, pages 132–143, 1991.

[PTS93] D. Pryor, M. Thistle, and N. Shirazi. Text Searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[Pur] Pure Atria. *Quantify User's Guide.*

[Ral90] L. B. Rall. Tools for Mathematical Computation. In *Computer Aided Proofs in Analysis*, volume IMA volumes in Mathematics and Its applications, pages 217–228. Springer Verlag, 1990.

[RV96] S. Rajamani and P. Viswanath. A Quantitative Analysis of Processor-Programmable Logic Interface. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 226–234, 1996.

[Sch96] M. Schulte. *A Variable Precision, Interval Arithmetic Processor*. PhD thesis, University of Texas, Austin, 1996.

[SG93] A. Skaf and A. Guyot. VLSI Design of On-line Add/Multiply Algorithms. *IEEE Int. Conference on Computer Design*, pages 264–267, 1993.

[SSJ95a] M. J. Schulte and E. E. Swartzlander Jr. A Coprocessor for Accurate and Reliable Numerical Computations. In *IEEE Int. Conf. on Computer Design*, pages 686–691, 1995.

[SSJ95b] M. J. Schulte and E. E. Swartzlander Jr. Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor. In *IEEE 12th Symposium on Computer Arithmetic*, pages 222–229, 1995.

[TE77] K. S. Trivedi and M. D. Ercegovac. On-line Algorithms for Division and Multiplication. *IEEE Trans. on Computers*, C-26(7):681–687, 1977.

[TE86] D. M. Tullsen and M. D. Ercegovac. Design and VLSI Implementation of an On-line Algorithm. In *Real Time Signal Processing IX - 698*, pages 92–99. SPIE, 1986.

[TE87] P. K.-G. Tu and M. D. Ercegovac. A Radix-4 On-line Division Algorithm. In *IEEE 8th Symposium on Computer Arithmetic*, pages 181–187, 1987.

[TE89] P. K.-G. Tu and M. D. Ercegovac. Design of On-line Division Unit. In *IEEE 9th Symposium on Computer Artihmetic*, pages 42–49, 1989.

[TE96] A. F. Tenca and M. D. Ercegovac. High-radix digit-slices for on-line computations. In *Proceedings of SPIE Conference on High Speed Computing, Digital Signal Processing, and Filtering using reconfigurable logic*, volume 2914, pages 14–25, 1996.

[Tor93] G. Torbjon. *User Guide to GNU MP: Multiple Precision Library*, 1.3.2 edition, 1993. Available with the software.

[Tu90] P. K.-G. Tu. *On-line Arithmetic Algorithms for Efficient Implementation.* PhD thesis, University of California, Los Angeles, Sept 1990.

[VBD+96] J. E. Vuillemin, P. Bertin, Roncin D., M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[VdB92] D. et al. Van den Bout. Anyboard: An FPGA-Based Reconfigurable System. *IEEE Design and Test of Computers*, pages 21–30, September 1992.

[Vui90] J. E. Vuillemin. Exact Real Computer Arithmetic with Continued Fractions. *IEEE Trans. on Computers*, 39(8):1087–1105, 1990.

[Wau91] T. C. Waugh. Field Programmable Gate Array Key to Reconfigurable Array Outperforming Supercomputers. In *IEEE Custom Integrated Circuits Conference*, 1991.

[Wie] A. Wiethoff. A C++ Class Library for Extended Scientific Computing.

[Xil93] Xilinx. *The Programmable Logic Data Book*, 1993.

[YX96] W. W. H. Yu and S. Xing. Performance Evaluation of FPGA implementation of High-Speed Addition algorithms. In *Proceedings of SPIE Conference on High Speed Computing, Digital Signal Processing, and Filtering using reconfigurable logic*, volume 2914, pages 26–33, 1996.

[Zim92] P. Zimmerman. Comparison of Three Public-Domain Multiprecision Libraries: Bignum, GMP and Paris. Obtained through e-mail:zimmerman@inria.fr, 1992.

[Zur93] D. Zuras. On Squaring and Multiplying Large Integers. In *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 260–271, 1993.