A comparison of logarithmic and floating-point number systems implemented on Xilinx Virtex-II field-programmable gate arrays

Barry Roland Lee. University of Wales, Cardiff. 2004.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

DECLARATION

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... 2/4/04 .....(candidate)

Date.....

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... 2/4/04 .....(candidate)

Date.....

STATEMENT 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... 2/4/04 .....(candidate)

Date.....

Summary

The aim of this thesis is to compare the implementation of parameterisable LNS (logarithmic number system) and floating-point high dynamic range number systems on FPGA. The Virtex/Virtex-II range of FPGAs from Xilinx, which are the most popular FPGA technology, are used to implement the designs. The study focuses on using the low level primitives of the technology in an efficient way and so initially the design issues in implementing fixed-point operators are considered. The four basic operations of addition, multiplication, division and square root are considered. Carry-free adders, ripple-carry adders, parallel multipliers and digit recurrence division and square root are discussed. The floating-point operators use the word format and exceptions as described by the IEEE std-754. A dual-path adder implementation is described in detail, as are floating-point multiplier, divider and square root components. Results and comparisons with other works are given. The efficient implementation of function evaluation methods is considered next. An overview of current FPGA methods is given and a new piecewise polynomial implementation using the Taylor series is presented and compared with other designs in the literature.
In the next section the LNS word format, accuracy and exceptions are described and two new LNS addition/subtraction function approximations are described. The algorithms for performing multiplication, division and powering in the LNS domain are also described and are compared with other designs in the open literature. Parameterisable conversion algorithms to convert to/from the fixed-point domain from/to the LNS and floating-point domain are described and implementation results given. In the next chapter MATLAB bit-true software models are given that have the exact functionality as the hardware models. The interfaces of the models are given and a serial communication system to perform low speed system tests is described. A comparison of the LNS and floating-point number systems in terms of area and delay is given. Different functions implemented in LNS and floating-point arithmetic are also compared and conclusions are drawn. The results show that when the LNS is implemented with a 6-bit or less characteristic it is superior to floating-point. However, for larger characteristic lengths the floating-point system is more efficient due to the delay and exponential area increase of the LNS addition operator. The LNS is beneficial for larger characteristics than 6-bits only for specialist applications that require a high portion of division, multiplication, square root, powering operations and few additions.

Thank you...

In memory of my father, who passed away on the 8th February 2003.

My girlfriend Asmah, for her love, support and sanity checking.

My mum, for not asking stupid questions, financial support and for being who she is so I can achieve my goals.

My brother, Councillor Rob Lee for more sanity checking and a cheap place to stay.

Neil Burgess (Icera Semiconductor and Cardiff University), for having faith in my ability when all seemed lost and for the support provided in every aspect of my PhD 'life'.

Richard Walke (QinetiQ Limited), I am very grateful for the many research area suggestions and for the many helpful discussions. I feel lucky to have had his support, as this thesis would *definitely not* have been possible without the level of motivation and dedication he promotes.

Sponsors QinetiQ (formerly, DERA Defence Research and Evaluation Agency), for the three years of financial support and for the excellent company work experience.

RTSL team and everyone else at QinetiQ, who made my periods of work at QinetiQ a very enjoyable experience.

Lai Sze Au (Fellow PhD student and friend).

Nikos Mallios (Fellow PhD student and friend), for many late night discussions, words of encouragement and excellent hospitality.

Wayne Luk (Imperial College, London), for being a friendly face, and offering much support at many PLD conferences. A true research ambassador.

David Defour (ENS, Lyon), for being a good friend on my visit to ENS, Lyon, France.

Ken Lever (Cardiff University), for many interesting and help mathematical and DSP discussions, and for the presentation contribution at Asilomar 2003 (a close one!).

Arnold Tisserand (ENS, Lyon), for allowing me to have a month stay at ENS Lyon (away from the rain), which inspired many a good idea.

Jean-Michel Muller (ENS, Lyon), the head of ENS Lyon.

Braden Philips (Cardiff University), a nice guy with many good suggestions.

Matthew Patterson, George Blackwell, Wing Yau and Ross Townend.

Tempus fugit

Ergo ad rem, mox nox.

Contents

Acronyms

$\oplus$	Logical XOR function.
+	Addition for numeric operations. OR for logical operations.
.	Logical AND function when used in logic expressions.
*	Arithmetic multiplication.
[a, b]	A range including the limits a and b.
(a, b]	A range not including the limit a.
[a, b)	A range not including the limit b.
(a, b)	A range not including the limits a and b.
$\vee$	Logical OR function.
add/sub	Component that performs addition and subtraction.
AND	Logical AND function.
ALU	Arithmetic Logic Unit.
ASIC	Application Specific Integrated Circuit.
BlockRAM	Memory components found on Xilinx Virtex FPGAs.
BTFP	Better Than Floating-Point.
cc	Carry-chain.
Ceil(.)	Mathematical ceiling function.
CLB	Configurable Logic Block.
CORDIC	Coordinate Rotation Digital Computer.
CPA	Carry Propagate Adder.
CPU	Central Processing Unit.
CSA	Carry Save Adder.
dbz	Divide by zero.
DSP	Digital Signal Processing/Processor.
EDIF	Electronic Design Interchange Format.
Embed Mults	18x18-bit Embedded Multipliers found on Xilinx Virtex-II FPGAs.
exp	Exponent of a floating-point number.
ff	Flip-flop (D-type).
flp	Floating-point.
FPGA	Field-Programmable Gate Array.
IEEE	Institute of Electrical and Electronics Engineers.
inf	Infinity.
kcm	Constant coefficient multiplication.
LC	Logic Cell.
LCPQ	LUT Columns Per Quotient Digit.
LCPQP	LUT Columns Per Quotient Digit Pipelined.
LE	Logic Element.
LNS	Logarithmic Number System.
LPQ	Latency Per Quotient bit.
LSB	Least Significant Bit.
LSDF	Least Significant Digit First.
LUT	Look Up Table.
LZA	Lead Zero Anticipator.
LZD	Lead Zero Detector.
MAC	Multiply Accumulate.
mant	Mantissa.
max(.)	Function to calculate the maximum of a group of arguments.
min(.)	Function to calculate the minimum of a group of arguments.

MHz	Megahertz.
MSB	Most Significant Bit.
MSDF	Most Significant Digit First.
mux	Multiplexer.
NaN	Not-a-Number.
NOR	Logical NOR function.
ns	Nanoseconds.
OR	Logical OR function.
ov	Overflow.
P&R	Place and Route.
PC	Personal Computer.
PP	Partial Product.
R4	Radix-4.
RAM	Random Access Memory.
RCA	Ripple Carry Adder.
RHS	Right Hand Side.
RISC	Reduced Instruction Set Computer.
RLOC	Relative location.
ROM	Read Only Memory.
RTN	Round To Nearest.
RTNE	Round To Nearest Even.
RTMI	Round To Minus Infinity.
RTPI	Round To Plus Infinity.
SBTM	Symmetrical Bipartite Table Method.
SRT	Sweeny, Robertson, Tocher.
SRAM	Static Random Access Memory.
STAM	Symmetrical Table Addition Method.
TRUNC	Truncation.
UART	Universal Asynchronous Receive Transmit.
ulp	Unit in Last place (the weighting of the LSB).
un	Underflow.
UUT	Unit Under Test.
VHDL	Very high speed integrated circuit Hardware Description Language.
VLSI	Very Large Scale Integration.
XOR	Logical XOR function.

Chapter 1

Introduction

1.1 Programmable logic

The FPGA (Field-Programmable Gate Array) technology sector has grown immensely over the past decade as is evident from the number of conferences, companies and research centres dedicated to or involved with programmable logic. FPGAs are silicon programmable logic devices that can be reprogrammed numerous times (some companies claim an infinite number) to perform many different logic functions. Re-programmability and flexibility are the key factors to the success of the FPGA as they allow simple design enhancements, such as bug fixes, or a total algorithm overhaul with no hardware cost and minimal time penalty. Furthermore, changes can be done remotely, very rapidly and with FPGA devices still in system. Prototyping and rapid implementation of algorithms and custom designs with hardware speeds is possible, and with high level programming languages the ease and speed of design implementation is increasing. However, the high level languages do come at a price and control over certain design aspects, such as placement and low level primitive use, are lost to the synthesis and place and route (P&R) tools. Although slower than major commercial processors the custom data-path configuration and reduced overhead means that FPGA hardware implementations can speed up many software algorithms implemented on general-purpose processors or DSPs (Digital Signal Processors). FPGAs cannot match the speed and low power consumption of full custom ASIC (Application Specific Integrated Circuit) design, but such procedures are very expensive to undertake and are generally only used for mass produced or specialist devices. However, low cost high volume FPGA devices such as the Spartan-3 from Xilinx [32] using the latest 90 nanometre technology are trying to break into this high volume market for certain applications.

1.2 A brief FPGA technology history and overview

The traditional FPGA architecture consisted of a rectangular array of identical logic elements constructed from small LUTs (Look Up Tables) or from a selection of combinatorial logic gates. Logic elements typically contained a flip-flop or could be configured as a flip-flop. Similarly to current FPGAs the traditional FPGAs were SRAM based, which allowed them to be reconfigured. SRAM cells, which reside throughout the FPGA, store the configuration bits that determine the function a LUT stores, the routing of a signal, the constant select signal value of a multiplexer, and so on. Logic elements, registers and the I/Os were routed together in the traditional FPGA architecture by using a programmable interconnect matrix in much the same way as current FPGAs, however current FPGA routing matrices are much more complex. The first FPGAs contained up to 600 logic elements with around 70 I/Os and operated at a maximum clock speed of 75 MHz. They were typically implemented with 1.0 μm VLSI technology and operated at 5V.

As the technology size has decreased in various steps from 1.0 μm (0.6, 0.5, 0.45 ... 0.15, 0.13, 0.12 μm) to the current 90 nm technology of today the density of devices has increased. Current devices Xilinx [36] and Altera [48] have up to 180K logic elements and up to 1100 I/Os compared to the 600 logic elements and 70 I/Os of the first FPGAs. The successive technology size decrease has also brought about a successive speed increase and a voltage decrease as current FPGAs can operate with internal clock speeds of over 450 MHz and a core voltage of just 1.2V.

The applications of FPGAs have demanded additional dedicated features be incorporated into the FPGA structure. The additional features reduce the burden and inefficiencies of their implementation with basic logic elements. The first additional feature on FPGAs was dedicated blocks of memory. The memory blocks started off small at 128 to 256-bits and with a limited dual-port functionality of 1 read and 1 write port. A single FPGA could implement a total of 3K to 18Kbits of memory. The size of the memory blocks quickly grew from 1K to 2K to 4K to 9K to 18Kbits and most recently Altera have included a 590K RAM block in their Stratix-II device [48]. Currently the largest FPGA can implement around 8.1Mbits of dedicated block memory. Configuration and functionality of the memory blocks has also grown and most memory blocks can implement true dual-port memory, where both ports can be configured to read and/or write. The second additional feature to appear on FPGAs

was dedicated carry logic. This logic enables very fast ripple-carry adders, comparators and wide logic gates to be created. The third feature to appear on FPGAs was dedicated multipliers [35] and DSP blocks [25] and [48]. FPGAs are increasingly being used in DSP applications where multiplication and multiply-accumulate (MAC) operations are required and so dedicated logic for these operations has been included. The dedicated multiplier blocks follow a de facto standard of being 18x18-bit signed two's complement. All the embedded DSP blocks contain a number of 9x9-bit multipliers and these can be configured to produce the de facto 18x18-bit and 36x36-bit multipliers in a single block. The DSP blocks also provide support for multiply accumulate and complex multiplication functions at higher clock rates and with fewer pipeline stages than using logic elements to implement the same functions. The fourth feature to appear on FPGAs was the high performance transceiver block [36], which allows very fast serial communication of up to 10Gb/s with a single I/O line. Data is serialised and de-serialised on chip in dedicated hardware. Finally the newest feature to appear on FPGAs is the dedicated RISC CPU. The Virtex-II PRO [36] contains up to 2 IBM PowerPC 405 RISC CPU cores.

1.3 FPGA design flow

Initially when creating a design a high level model of the system is created in a high level programming language such as C or Java or in a mathematical environment such as MATLAB. This allows the concept and validity of an algorithm to be tested in software before any hardware design is undertaken. The high-level software design is usually written to match the hardware design exactly and so is referred to as a 'bit-true' design. This provides the ultimate layer of testing, as the processed data that the hardware returns should match the data returned by the software model exactly. The hardware design is written in a hardware description language such as VHDL or Verilog. Certain vendors and research institutes also offer other high level languages and design environments such as handel-C (C based), the float environment (written in Perl), the PAM Blox-II environment, C++ integration via the ACS stream complier and Jbits design environment (Java), which are more familiar to software designers. The idea is to allow the designer to focus on the algorithm implementation and not to worry about the underlying hardware structure. The hardware description is run through a synthesis tool, which translates the design into gates and optimizes it for the

target FPGA architecture and outputs an industry standard EDIF (Electronic Design Interchange Format) netlist. After synthesis the design is placed and routed (P&R), which decides the layout of the logic components and the routing structure to connect them. Specifically the Xilinx P&R flow, which will be used in this work, contains three phases of 'translation', 'mapping' and 'place and route'. The translation phase combines multiple netlists and specific design constraint information. The mapping phase maps the logic to the components (logic cells, I/O cells, and other components) in the target FPGA. The place and route phase is as described above. After P&R a configuration bitstream is generated that can be downloaded onto a device to program it with the functionality of the created design. At different stages of the design different levels of testing can be carried out. After the design has been written in VHDL or Verilog a functional test can be undertaken. Post P&R a full back-annotated test can be performed using the FPGA vendors' hardware libraries. This test provides the most accurate picture of how the final implemented design should perform. Finally there is a system test, where the final design running on hardware can be compared against the bit-true software model or any other successful test layer.

1.4 Aim of thesis: FPGA and high dynamic range number systems

Many algorithms use high dynamic range number systems to increase accuracy, increase the signal-to-noise ratio or to reduce the number of underflow and overflow exceptions. Application areas for high dynamic range number systems include digital signal processing applications such as FFT (Fast Fourier Transform) calculation, FHT (Fast Hartley Transform) calculation, digital sine-cosine generation, FIR (Finite Impulse Response) filtering and RLS (Recursive Least Squares) filtering; Image processing applications include video conferencing, compression and decompression in HDTV (High Definition Television), K-Means distance calculation, MPEG (Motion Pictures Expert Group) decoding, DCT (Discrete Cosine Transform) and IDCT (Inverse Discrete Cosine Transform) calculation; scientific applications include matrix operations, gravitational N-body interaction calculation, heat transfer in a 2D surface and seismic data processing; Telecommunication applications include network packet switching and cellular base station algorithms. Clearly there are many applications of high dynamic range computing, most of which are candidates for hardware acceleration using FPGA systems.

The I/O transfer rates, I/O pin counts and the density of current FPGAs have advanced sufficiently to accommodate these high dynamic range applications with ease. Furthermore the implications of Moore's law suggest that the density of FPGA technology will further increase in the next few years (until a technology wall is hit). This leads to the aim of this thesis, which is to investigate the trade offs in implementing two popular high dynamic range number systems on FPGAs, namely floating-point and logarithmic number system (LNS) arithmetic. LNS and floating-point are very different in their implementation but both have the capabilities of providing high dynamic range arithmetic. The question is, 'which one is most suitable for FPGA implementation?' Current FPGAs contain a restricted variety of logic elements compared to ASIC design and this poses quite a design challenge as the many efficient algorithms that have been developed for ASIC implementation do not necessarily map well to this design environment. An example of this is the compound adder that can calculate two different results of $a+b$ and $a+b+1$ with a small delay and area overhead. Implementing such an adder on FPGA requires three times as much logic as a conventional adder (i.e. two adders and a multiplexer to select the output). A compound adder calculates two different results simultaneously while the decision whether to select $a+b$ or $a+b+1$ is being performed, hence it is different to (faster than) calculating $a+b$ then injecting '1' or '0' into the carry-in input as desired. The focus and challenge of this work is therefore to look at new ways to efficiently implement basic operations and components and then to use these components to implement floating-point and LNS operations. Furthermore a low-level design strategy will be adopted that will utilise the low-level primitives of the technology in an efficient way and will increase the design challenge. It is appreciated that this design strategy affects the portability of the design and goes against the high-speed rapid prototyping methodology of reconfigurable design, but it is felt that the knowledge gained of the underlying FPGA hardware and the smaller and faster designs that the proposed strategy leads to is worth the extra effort. The task of implementing efficient fixed-point, floating-point and LNS units is non-trivial due to the huge variety of techniques that need to be considered. In this work some of the more common ASIC techniques are considered and new techniques are developed in an attempt to provide the first steps in a fair comparison of LNS and floating-point arithmetic on FPGA. We say first steps, as the array of techniques is so vast that many substantial studies have been included in the future work section due to time

restrictions. Due to the influence of the sponsors, and of the technology available the study will focus primarily on using the Virtex-II range of FPGAs from Xilinx [35]. The Virtex-II FPGA is very representative of current technology because it contains features such as memory blocks, multipliers, fast carry-logic and an array of small lookup tables that are commonly found in all current FPGA architectures.

1.5 Structure of the Xilinx Virtex and Virtex-II FPGAs

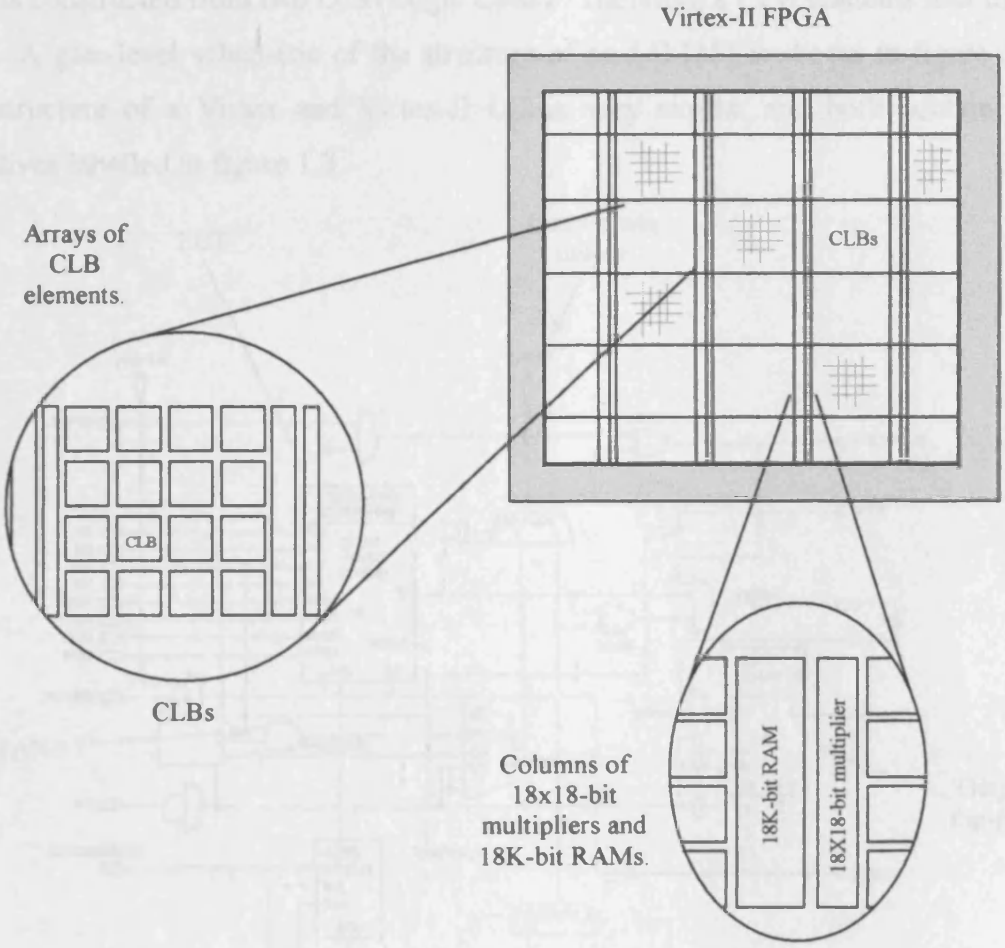


Figure 1.1. Logic structure of a Xilinx Virtex-II FPGA

Figure 1.1 shows the underlying structure of a Xilinx Virtex-II [35] FPGA. It can be seen that the chip consists of arrays of CLB (Configurable Logic Blocks) cells separated by columns of multiplier and RAM blocks. The structure of a Virtex [34] FPGA is very similar except that there are only two columns of RAM blocks on either side of the chip and there are no embedded multiplier blocks.

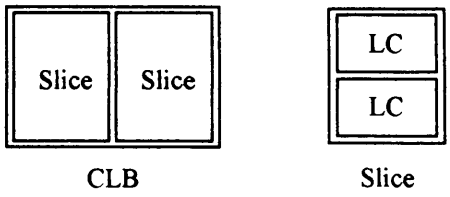


Figure 1.2. CLB and Slice structure of Xilinx Virtex and Virtex-II FPGAs

From figure 1.2 it can be seen that a CLB is constructed from two Slices and that a Slice is constructed from two LCs (Logic Cells). Therefore a CLB contains four logic cells. A gate-level schematic of the structure of an LC [35] is shown in figure 1.3. The structure of a Virtex and Virtex-II LC is very similar and both contain the primitives labelled in figure 1.3.

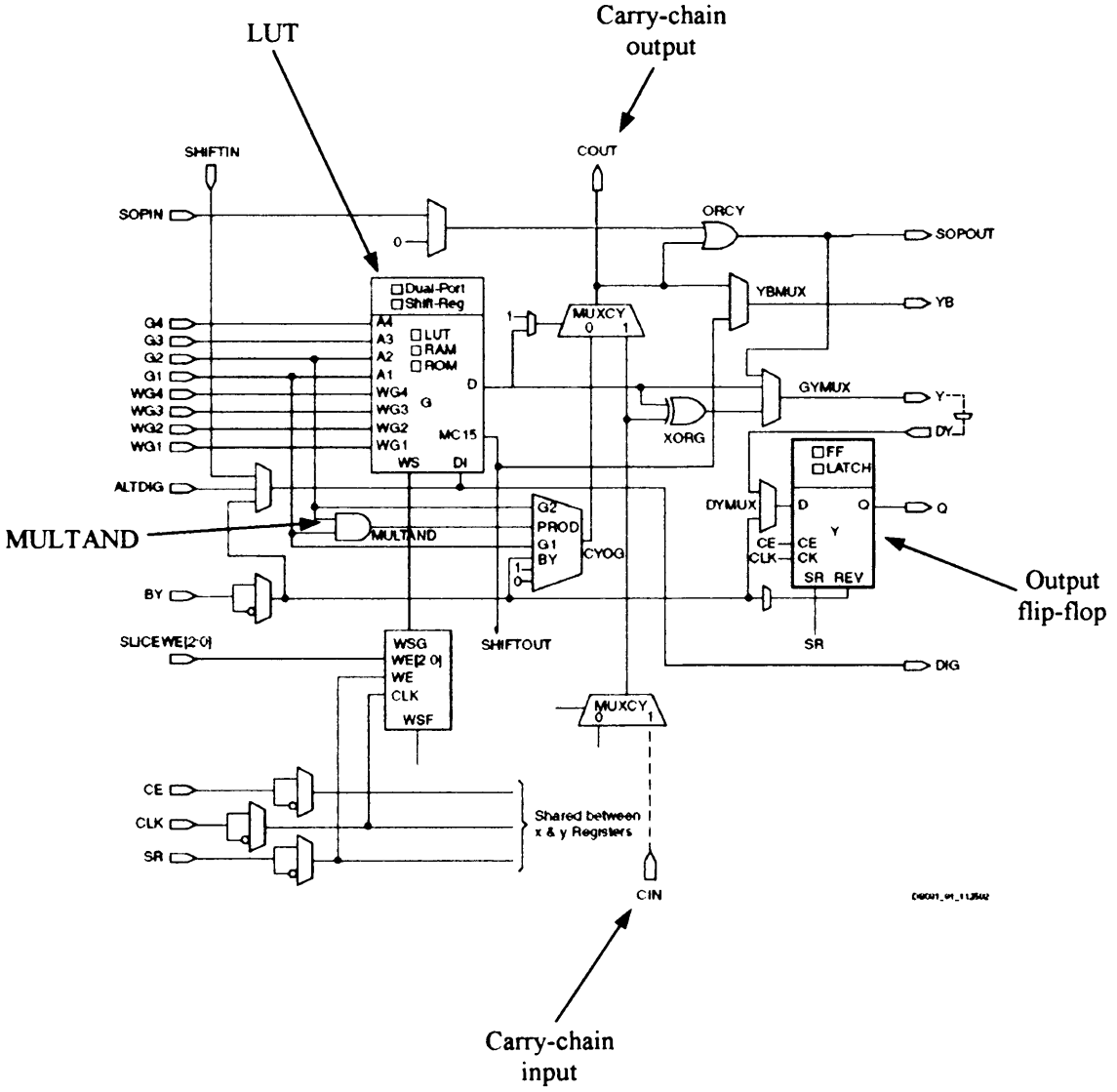


Figure 1.3. The structure of a Virtex-II logic cell (LC).

A brief description of each of the primitives labelled is given below:

- LUT. The LUT (Look-up Table) primitive is very versatile. It can be configured as a 4-bit input 1-bit output function generator, as a 16-bit memory (4-bit address, 1-bit data) or as a 16-bit shift register.
- Carry-chain. The carry-chain allows high-speed ripple-carry adders and wide logic gates to be produced. Each LC contains one 'MUXCY' carry-chain multiplexer. The bottom 'MUXCY' primitive shown in figure 1.3 is situated in the LC below.
- Output flip-flop. The output flip-flop allows the output of the LC to be registered without using any extra LCs. The register can be bypassed and can also be configured in numerous ways.
- MULTAND. The MULTAND primitive is a logical AND gate that facilitates the creation of efficient multipliers (the AND function is the same as a single bit multiplication).

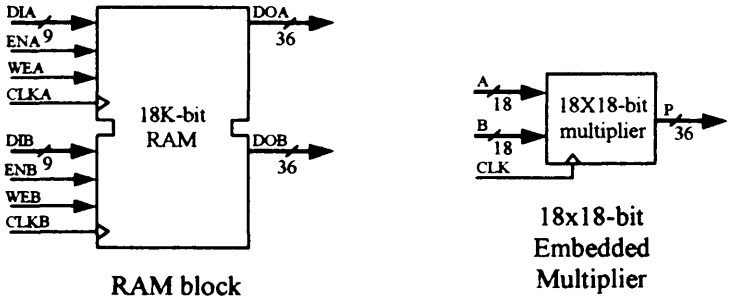


Figure 1.4. The interfaces of the Virtex-II RAM block and embedded multiplier

The Virtex-II RAM blocks are dual-port RAMs as shown in figure 1.4. Being dual-port means that there are two ports that access the same memory space. The embedded multipliers are 18X18-bit signed two's complement multipliers that produce a 36-bit signed two's complement output. The multipliers can be used synchronously or asynchronously. The interface of the 18X18-bit multiplier is shown in figure 1.4.

1.6 Thesis organisation and chapter contribution summary

In this section the outline of the thesis is given, as is the contribution made within each chapter. The publications where the contributions of the chapter were initially made are listed.

Chapter 2

Chapter 2 looks at the implementation of fixed-point operators. Fixed-point is the basis of all arithmetic so in this chapter the focus is on efficient implementation of the four common fixed-point operators of addition, multiplication, division and square root. A new 4:2 adder mapping is shown in the addition section and the advantages and disadvantages of carry-free addition are discussed. A few ripple-carry adder configurations are then presented some of which are not discussed in the open literature. The multiplication section looks at the implementation of Booth encoding, which is not considered elsewhere in the open literature for implementation on the Virtex-II FPGA. Two new partial product generators are presented in the multiplier section. A new way of expanding the embedded multipliers is discussed in this section and is compared with other methods [50]. The new multiplier expanding technique is also applied to the special multiplication case of squaring [50]. The division section looks at the implementation of digit-recurrence style dividers, not previously considered in the way presented. New implementations of the non-restoring, radix-4 SRT [51] and radix-8 SRT [50] dividers are given that are not previously given in the open literature. The prescaling technique, which has not before been considered for implementation on FPGA, is applied to the minimally redundant radix-8 design. A parameterisable non-restoring square root design is given in the final fixed-point section.

Chapter 3

Floating-point operations of addition, multiplication, division and square root are considered in this chapter. The chapter begins with a brief overview of floating-point and the IEEE std-754 format. The first published FPGA implementation of the dual-path (far, near) floating-point addition algorithm is presented [51]. A new carry-chain unit for performing IEEE 4-mode rounding, a new shifter mapping, a new method for performing lead zero detection and new methods for performing rounding in FPGA floating-point units are described. The floating-point multiplier makes use of the new embedded multiplier expanding. The first floating-point divider using the radix-4 SRT division algorithm is presented [50].

Chapter 4

Chapter 4 looks at ways of performing function approximation on FPGA. A new piecewise Taylor series approximation unit suitable for short to mid-length operands is described [52].

Chapter 5

The implementation of the four LNS operators of addition, multiplication, division and square root are considered in this chapter. The word format, accuracy and operator structures are given. Two new methods for performing LNS addition are described and compared in this chapter [53], [54].

Chapter 6

Chapter 6 deals with conversion from fixed-point to floating-point and vice-versa and from fixed-point to LNS and vice-versa. The structure of the conversion from fixed-point to LNS and vice-versa [55] is not discussed elsewhere in the open literature.

Chapter 7

The MATLAB software libraries and function interfaces are briefly discussed in this chapter, as is the implementation of the UART serial interface to allow a low speed system test of the modules to be carried out.

Chapter 8

In this chapter the logarithmic number system, floating-point operators and conversion components are compared in terms of area and delay. Some composite functions involving multiple operators are also compared to give a range of results.

Chapter 9

Chapter 9 concludes the findings of the results and comparisons. The suitability of the logarithmic and floating-point systems for the processing of different algorithms is discussed and further studies are proposed.

Chapter 2

Fixed-point

In this chapter the focus will be on how to efficiently implement fixed-point operators on FPGA and more specifically on Xilinx Virtex FPGAs. Fixed-point arithmetic is the underlying type of arithmetic used in high dynamic range arithmetic schemes so we will begin by considering ways of implementing on FPGA the four basic operations of addition, multiplication, division and square root using fixed-point.

2.1 Formats

Fixed-point formats generally consist of an integer and a fractional field although sometimes the format is restricted to just an integer or fraction type. There are two main ways of implementing signed values: sign-magnitude and two's complement. Sign-magnitude is a symmetrical format, which contains a single sign bit, an i-bit integer part and an f-bit fraction part. The value of a p-bit $(1+i+f)$ sign-magnitude value X is mathematically described in (2.1). For radix-2 (base-2) each bit x, of X, in equations (2.1) and (2.2) takes the value 0 or 1.

$$X = (-1)^{x_{p-f-1}} \left(\sum_{i=-f}^{p-f-2} x_i.2^i \right) \quad \text{(2.1)}$$

The MSB (most significant bit) of a two's complement number is a magnitude bit that is negatively weighted. Two's complement is a non-symmetrical fixed-point format with an i-bit integer part and an f-bit fraction. The value of a p-bit $(i+f)$ two's complement number X is given by (2.2).

$$X = -x_{p-f-1}.2^{p-f-1} + \sum_{i=-f}^{p-f-2} x_i.2^i \quad \text{(2.2)}$$

2.2 Distribution of fixed-point values

Consider a 6-bit fixed-point format with 3 integer bits and 3 fraction bits. Figure 2.1 illustrates the number line for the 6-bit two's complement format. From figure 2.1 it can be seen that the distribution of fixed-point values is even throughout the range of values.

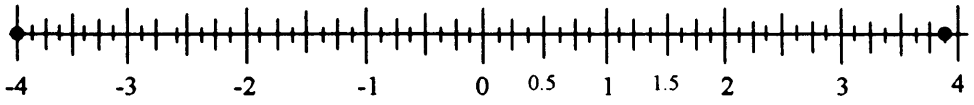
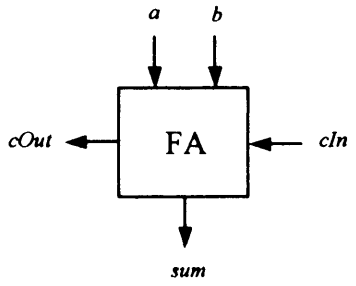


Figure 2.1. Number line for a 3-bit integer and 3-bit fraction two's complement number

The dots in figure 2.1 illustrate the location of the maximum and minimum values.

2.3 Fixed-point addition

Fixed-point addition is the most basic arithmetic operation. The basic building block of the fixed-point adder is the full adder shown in figure 2.2. Full adder cells can be considered as 3-bit counters, where the inputs all have an equal weighting of '1', the sum has a weighting of '1' and the carry has a weighting of '2'. Full adder cells can be chained together to create RCAs (ripple-carry adders), which are a special type of CPA (carry propagate adder) where two non-redundant inputs are added (reduced) to a single non-redundant value.



$$sum = a \oplus b \oplus cIn$$

$$cOut = a.b + a.cIn + b.cIn$$

Figure 2.2. A full adder cell

2.4 FPGA addition

The dedicated carry-chain embedded in the FPGA fabric allows very fast and efficient ripple-carry adders to be implemented. The carry-chain logic is so fast that alternative

adder designs Omondi [56], Parharmi [59] and Ercegovac [61], as used in ASIC devices, do not offer any implementation benefits. A survey of different FPGA CPA adder implementations is given in Xing [80] and Perri [81] and they show that the RCA is the most efficient adder design for FPGA. In this section we look at the basic RCA implementation and some very useful alternative configurations that are not available in the open literature. All the alternative structures use the same amount of logic as the basic adder design but the major difference is in the number of inputs a component requires and the logic function of the basic LUT elements.

2.4.1 Add (addition)

The full adder is the basic building block of the ripple-carry adder. The resources of a single LE (logic element) can be configured in such a way as to produce a full adder. Figure 2.3 illustrates a single slice, which contains two LEs configured as a two bit unsigned adder. The slice boundary is shown in figure 2.3 by the dashed box.

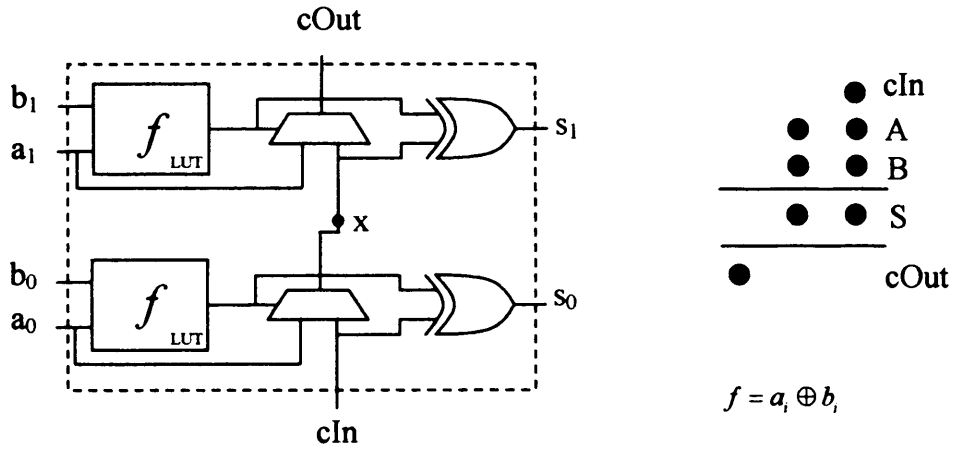


Figure 2.3. 2-bit adder implemented on a Virtex FPGA

A single full adder cell cannot be implemented on its own because there is no access to the carry-chain that joins the two LEs shown as point 'x' in figure 2.3. This has the repercussion that efficient [3:2] carry-save adders, a common carry free adder design, cannot be implemented efficiently on Virtex FPGAs.

2.4.2 Sub (subtraction)

True subtraction cannot be implemented efficiently on Virtex and Virtex-II FPGAs and so a two's complement system is used. The two's complement system involves

bit-inverting the operand to subtract, injecting a '1' into the carry-in (cIn) signal and then adding the two operands using a conventional ripple-carry adder. The LUT element needs to be configured with a logic function as shown in table 2.1.

2.4.3 Add/sub (addition/subtraction)

The addition and subtraction components can be combined into one macro that can do both operations as determined by a single selection signal. A 'sel' (selection) signal forces subtraction when it is at the logical value '1', addition when it is '0' and feeds the carry-in input of the adder. The LUT element logic function is shown in table 2.1.

2.4.4 Add/sub/zero (addition/subtraction/addition of zero)

In the add/sub component there is a single input into the LUT element that is not used. This line can be used to force a 'no operation' or an 'addition of zero'. Therefore a useful component that allows the operations A+B, A-B or A+0 can be implemented. The LUT element logic function is shown in table 2.1.

2.4.5 AddMux

Often when designing arithmetic components on FPGA the case where a choice of two values is to be added to another operand occurs. This can be pictured as an adder with a multiplexer on one of the inputs. A multiplexer consists of three inputs and an adder consists of two inputs. One of the adder inputs is fed by the multiplexer, thus four inputs are required for each bit, which fits into a LUT. The addMux component therefore calculates A+X or A+Y depending on the value of the 'sel' (selection) signal. Table 2.1 shows the LUT element logic function.

2.4.6 SubMux

The subMux component is a variation on the addMux component. Instead of A+X or A+Y the component performs A-X or A-Y depending on the 'sel' signal. See table 2.1 for the LUT element logic function.

2.4.7 AddSubMuxConst

It is not possible to fit an AddSubMux component, which adds or subtracts a choice of two operands from a given operand, into one LUT column. This is because such a component requires a five input LUT function. However it is possible to create such a

component if one of the operands to be subtracted or added is a constant and can be set at design time. The LUT function is then set at compile time and only a four input function is required to add/subtract a choice of a variable operand B or a constant operand K to/from a variable operand A. The component that performs this function is called an AddSubMuxConst component and the LUT element logic function for such a component is shown table 2.1.

2.4.8 Other configurations

There is scope to implement other types of adder that can blend logic functions that involve a particular input operand into the LUT used to implement the adder. This is a very useful way of reducing the delay and area of designs. Synthesis tools usually do not take advantage of this logic reduction technique and so it is advantageous to hand code macros to perform custom logic reduction functions.

Component name	Number of LUT inputs	Input names	Carry-chain input	Component function(s)	LUT function
Add	2	A, B, cIn	cIn	A+B	$a_i \oplus b_i$
Sub	2	A, B	'1'	A-B	$a_i \oplus \overline{b_i}$
Add/sub	3	A, B, sel	sel	A+B, A-B	$a_i \oplus b_i \oplus sel$
Add/sub/zero	4	A, B, sel, zero	sel	A+B, A-B, A+0, A-0	$a_i.\overline{b_i}.\overline{sel} + \overline{a_i}.b_i.\overline{sel}.\overline{zero} + \overline{a_i}.\overline{b_i}.sel + \overline{a_i}.sel.zero + a_i.b_i.sel.\overline{zero} + a_i.\overline{sel}.zero$
AddMux	4	A, X, Y, sel, cIn	cIn	A+X, A+Y	$a_i.x_i.\overline{sel} + \overline{a_i}.x_i.\overline{sel} + a_i.\overline{y_i}.sel + \overline{a_i}.y_i.sel$
SubMux	4	A, X, Y, sel	'1'	A-X, A-Y	$\overline{a_i}.\overline{x_i}.\overline{sel} + a_i.x_i.\overline{sel} + a_i.\overline{y_i}.sel + \overline{a_i}.\overline{y_i}.sel$
AddSubMuxConst	4	A, B, sel, const	sel	A+B, A-B, A+K, A-K	$(sel \oplus a_i \oplus b_i).\overline{const} + (sel \oplus a_i \oplus k_i).const$

Table 2.1. The configuration details of various FPGA adder implementations

2.4.9 Compact serial adders

By performing an addition in a number of steps and reusing the same hardware in each step a compact adder can be produced. If we assume two operands A and B of length N that need to be added are supplied LSB first one bit at a time then a single full adder cell can be used to sum both operands, Smith [82] and Andraka [83] show FPGA implementations. N cycles are needed to sum both operands and such an

algorithm belongs to the class of LSDF (least significant digit first) bit-serial algorithms Ercegovac [61]. The main problem with the bit-serial approach is the high latency. To reduce the latency a higher radix mode of operation called digit-serial can be used, Valls [93] and Lee [94] give FPGA implementations. In general a radix-r architecture calculates $\log_2(r)$ bits of the result per cycle. The usage of bit and digit-serial adders in a design depends on the availability of the operands to process. For this work we assume operands are supplied in parallel and so do not further investigate the use of this compact adder structure.

2.5 FPGA carry free addition

Carry free addition allows a number of operands to be reduced to a fewer number of operands without a carry propagating the full length of the addition. There are two popular types of carry free addition namely [3:2] carry-save addition and signed-digit addition. In this section the feasibility of the implementation of each is considered. [3:2] carry-save addition reduces three standard operands to two standard operands. It can also be considered as reducing a standard operand and a carry-save operand to a single carry-save operand. [4:2] carry-save addition is an extension of the [3:2] carry-save adder and can be seen as reducing four standard operands or reducing two carry-save operands to a single carry-save operand. Similar adders can be generated to add signed-digit operands without a propagation of a carry.

2.5.1 [3:2] CSA (carry-save adder)

To implement an n-bit ripple-carry adder n LUTs are required. However, to implement an n-bit [3:2] radix-2 carry-save adder $2n$ LUTs are required figure 2.4. The carry-save adder implementation only requires a single LUT delay, which is significantly less than the delay for a ripple-carry adder and it is independent of the precision. So the [3:2] carry-save adder has a place in high-speed applications where area is not a significant design issue.

2.5.2 Carry-save adder/subtracter

A carry-save structure that adds/subtracts a conventional operand to/from a carry-save operand depending on a select signal can be implemented using the same amount of

logic as the carry-save adder shown in figure 2.4. A 3-bit carry-save adder/subtracter with its corresponding dot diagram is shown in figure 2.5.

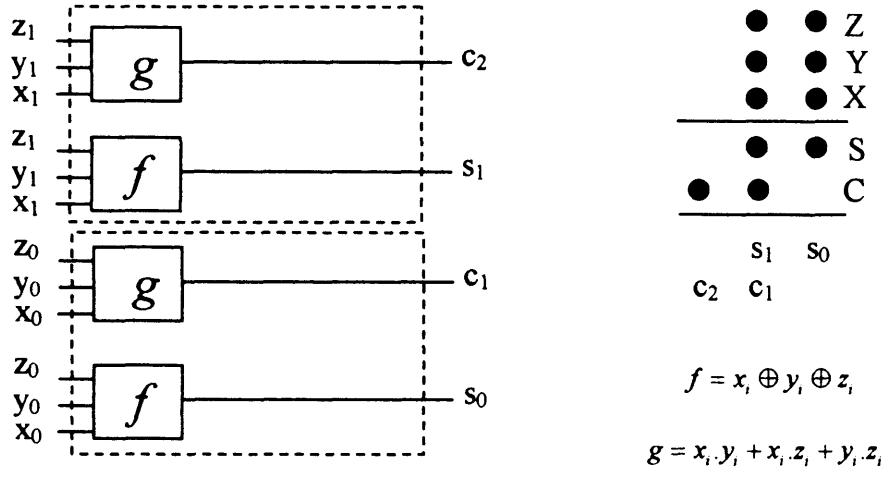


Figure 2.4. A 2-bit carry-save adder design

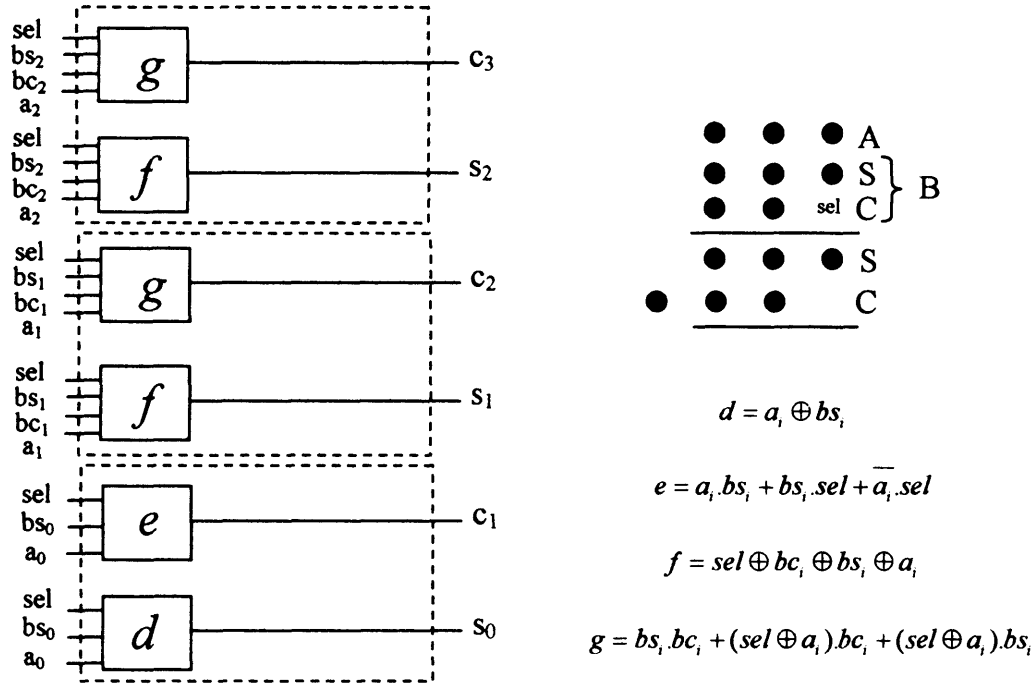


Figure 2.5. 3-bit addition/subtraction carry-save adder

2.5.3 High radix carry-save adders

Higher radix carry-save adder algorithms are possible and in particular the radix-4 algorithm leads to an efficient implementation in Virtex FPGAs. However, the higher

radix algorithms have the disadvantage over the [3:2] CSA in that they have to have one operand in high radix carry-save from. Figure 2.6 shows the implementation of a 4-bit radix-4 carry-save adder on Virtex FPGAs and it's corresponding dot diagram.

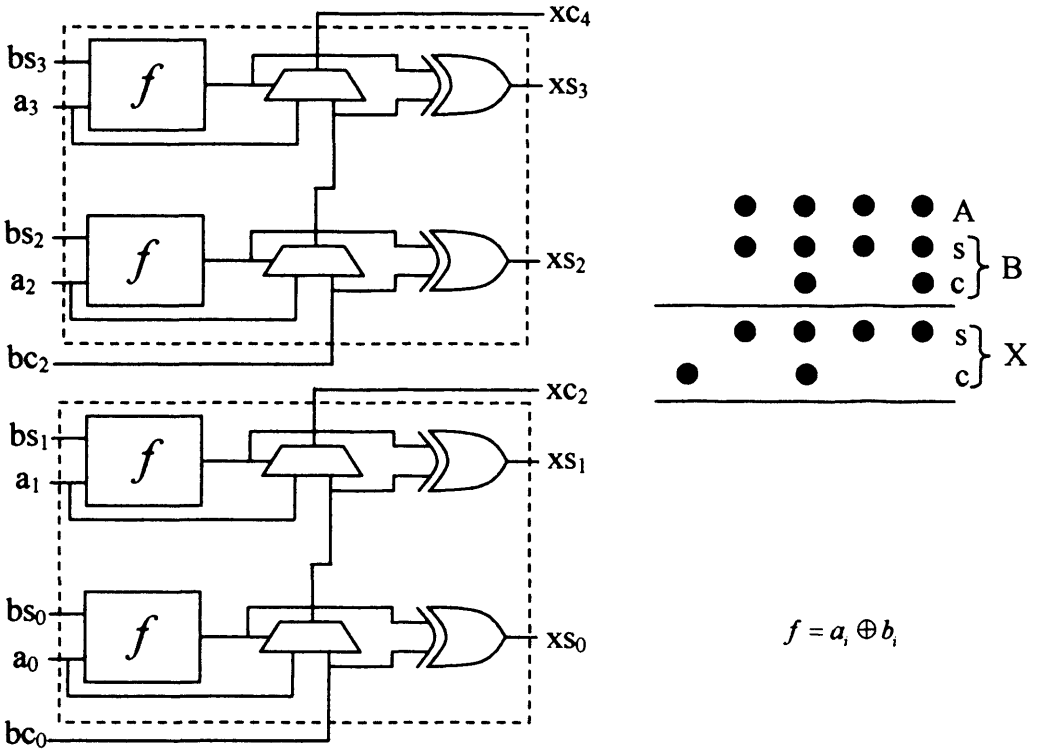


Figure 2.6. 4-bit radix-4 carry-save adder implementation

The adder in figure 2.6 is essentially a ripple-carry adder with a cut at every other carry-chain 'link' to produce an extra output and input. The output 'saves' the carry for the next stage of addition and the input takes the saved carry from the previous stage. High radix CSAs are useful in accumulation algorithms where the output of the adder is fed back into the inputs and summed with an operand in standard form. This is a popular operation in area efficient iterative algorithms such as digit-serial multiplication and division. Comparing figure 2.6 and figure 2.4 it can be seen that the radix-4 carry-save adder can sum a carry-save value and a standard operand using half as much logic as a radix-2 carry-save adder. A radix-4 carry-save adder/subtracter can also be implemented and requires the same amount of logic as figure 2.6. The downside to the efficient implementation of the high radix CSA is the extra delay caused by the XOR gate and carry-chain routing delay (see table 2.12).

2.5.4 [4:2] CSA (carry-save adder)

The [4:2] carry-save adder cell with carry-in and carry-out signals can be thought of as a [5:3] counter. Figure 2.7 shows the structure of a [4:2] adder. The adder is composed of two full adder (FA) cells, which are the basic [3:2] carry-save adder building blocks. From the structure of the [4:2] adder cell it can be seen that the carry-in signal has no effect on the carry-out signal giving a carry free adder.

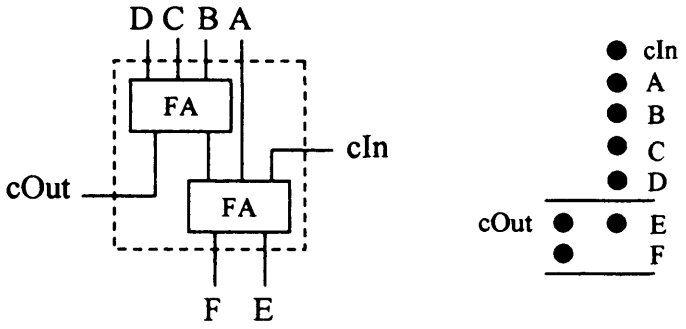


Figure 2.7. [4:2] carry-save adder and corresponding dot diagram

A basic mapping of an FPGA structure Luo [139] with the same functionality as figure 2.7 is shown in figure 2.8.

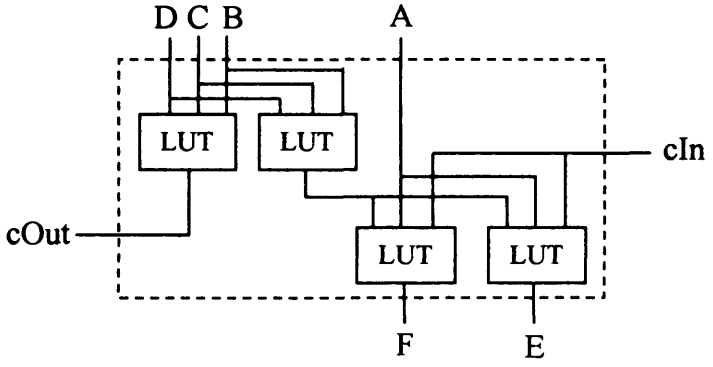


Figure 2.8. A basic mapping of a [4:2] adder to FPGA

2.5.5 New [4:2] CSA mapping

The structure of figure 2.8 can be improved upon and a novel FPGA mapping is shown in figure 2.9. The implementation allows a [4:2] adder that can compress four operands to two to be implemented in the same area as two conventional ripple-carry adders. From the adder structure of figure 2.9 it looks as if the [4:2] adder suffers from the same carry-chain access problem as the [3:2] carry-save adder. However, by

chaining the [4:2] adders together it can be seen that for any length adder only one logic element is wasted in accessing the carry-chain as shown in figure 2.10.

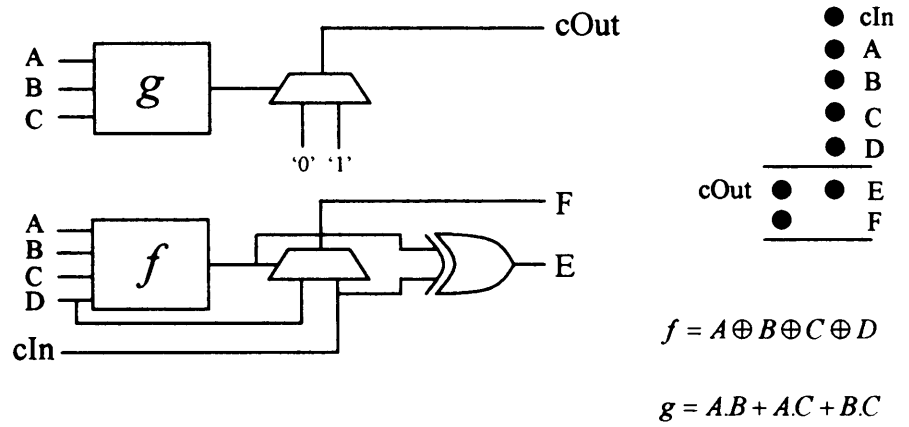


Figure 2.9. A novel [4:2] adder structure for Virtex FPGAs

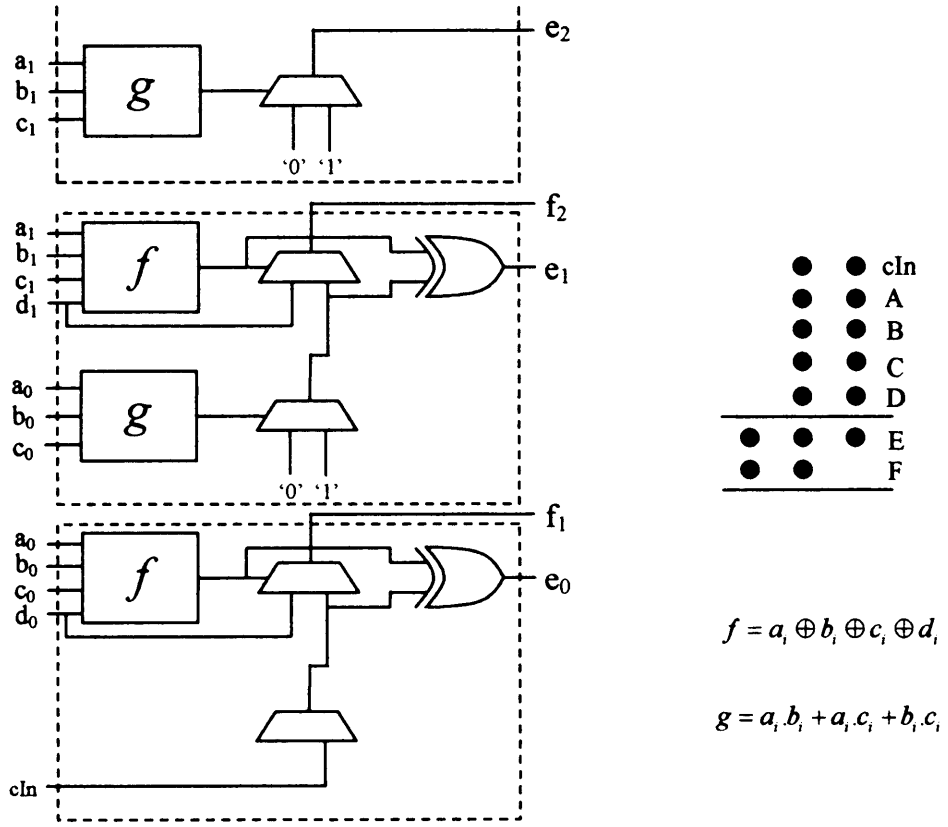


Figure 2.10. 2-bit [4:2] adder implemented on a Virtex FPGA

The main problem with the implementation of carry-save addition comes when trying to pipeline a design. Generally speaking carry-save arithmetic involves cutting the carry-chain to avoid a ripple of the carry. When pipelining a design the carry-chain

bit needs to be latched. The problem is that the Virtex FPGA does not have dedicated latches to register the carry bit (i.e. the 'Xc' operand in figure 2.6 and the 'F' operand in figure 2.10) and thus create efficient pipelined designs. Due to routing limitations a single logic element will be wasted when pipelining a carry bit.

2.5.6 Signed-digit addition

A signed-digit set Avizienis [63] allows a digit to take on positive and negative values. For example we will consider the radix-2 signed-digit set {-1,0,1}. Using a digit set with more consecutive digits (3 in this case) than the radix (2 in this case) causes the digit set to have redundancy, which allows carry free addition. A radix-2 signed-digit x consists of two bits: an x^- bit, which is the negative bit and an x^+ bit, which is the positive bit. This is known as 'borrow-save' encoding Ercegovac [61]. The coding of the signed-digit is shown in table 2.2.

x^+	x^-	x
0	0	0
0	1	-1
1	0	1
1	1	0

$$x = x^+ - x^-$$

Table 2.2. Coding of radix-2 signed digits

An n digit signed-digit integer is represented as (2.3).

$$X = \sum_{i=0}^{n-1} x_i r^i \qquad \text{(2.3)}$$

There are two different types of adder to consider for implementation. The first involves adding a signed-digit operand to a conventional non-redundant radix-2 operand. The second case requires the addition of two signed-digit operands.

2.5.7 Signed-digit to conventional digit addition

The addition/subtraction of a conventional digit y to/from a signed-digit x is best illustrated by a table showing all the possible results, which is shown in table 2.3. Implementing the penultimate two columns of table 2.3 as functions of the first three columns produces a carry free signed-digit and conventional digit adder with result in signed-digit form. An implementation of a 2-digit signed-digit and conventional digit

adder on a Virtex FPGA is shown in figure 2.11. A single cell of such an adder is known as a 'ppm' cell as two input operands are plus and one operand is minus. Similar mappings are given by Girau [101], McIlhenny [103] and Valls [107].

Operands						addition		subtraction	
x^+_i	x^-_i	y_i	x_i	$x_i + y_i$	$x_i - y_i$	s^+_{i+1}	s^-_i	s^-_{i+1}	s^+_i
0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	-1	1	1	1	1
0	1	0	-1	-1	-1	0	1	1	1
0	1	1	-1	0	-2	0	0	1	0
1	0	0	1	1	1	1	1	0	1
1	0	1	1	2	0	1	0	0	0
1	1	0	0	0	0	0	0	0	0
1	1	1	0	1	-1	1	1	1	1

Table 2.3. Signed-digit and conventional digit addition/subtraction results

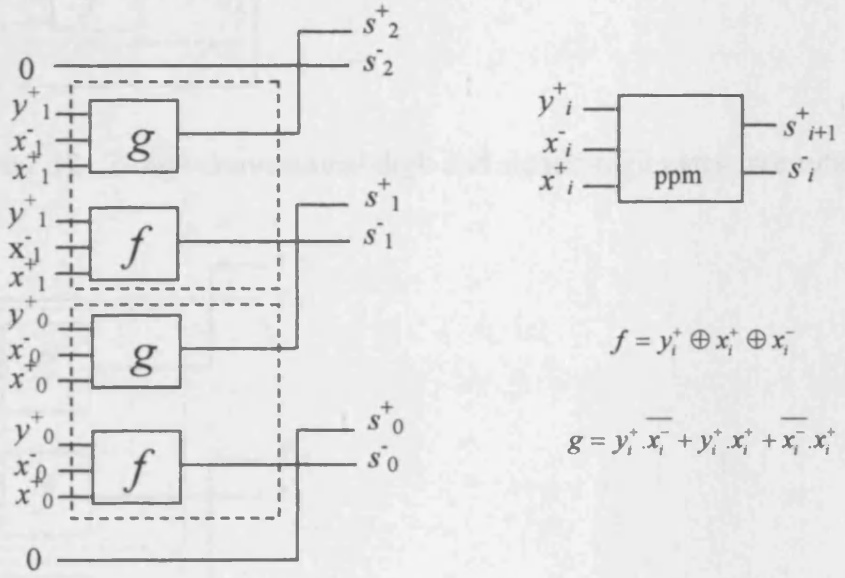


Figure 2.11. 2-digit signed-digit and conventional digit carry free adder

Implementing the final two columns of table 2.3 as functions of the first three columns produces a carry free signed-digit and conventional digit subtracter with result in signed-digit form. An implementation of a 2-digit subtracter on a Virtex FPGA is shown in figure 2.12. A single cell of such an adder is known as an 'mmp' cell as two input values are minus one is plus.

2.5.8 Signed-digit and conventional digit adder/subtracter

As for carry-save addition an adder that adds/subtracts a conventional operand to/from a signed-digit operand depending on a select signal can be implemented. Figure 2.13 shows a diagram of a 2-digit signed-digit and conventional digit adder/subtracter.

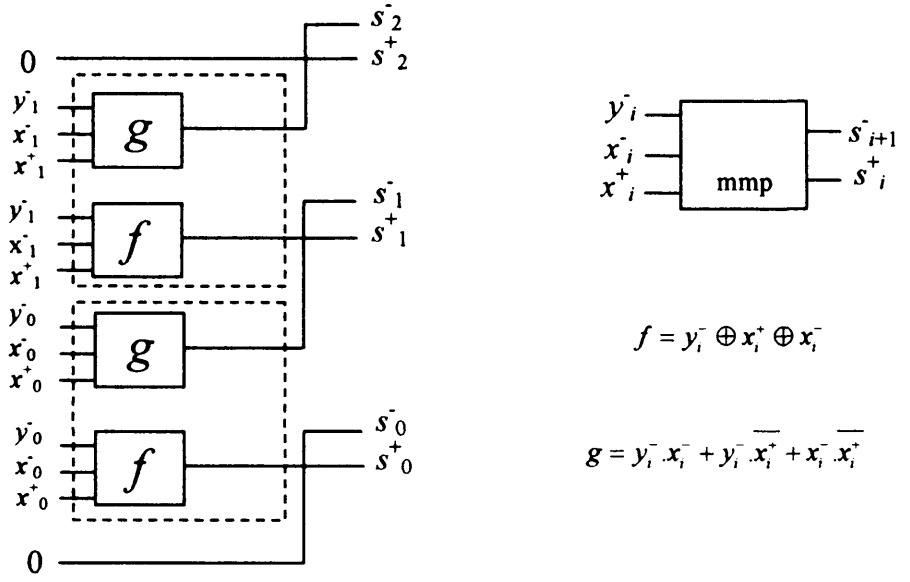


Figure 2.12. 2-digit conventional digit and signed-digit carry free subtracter

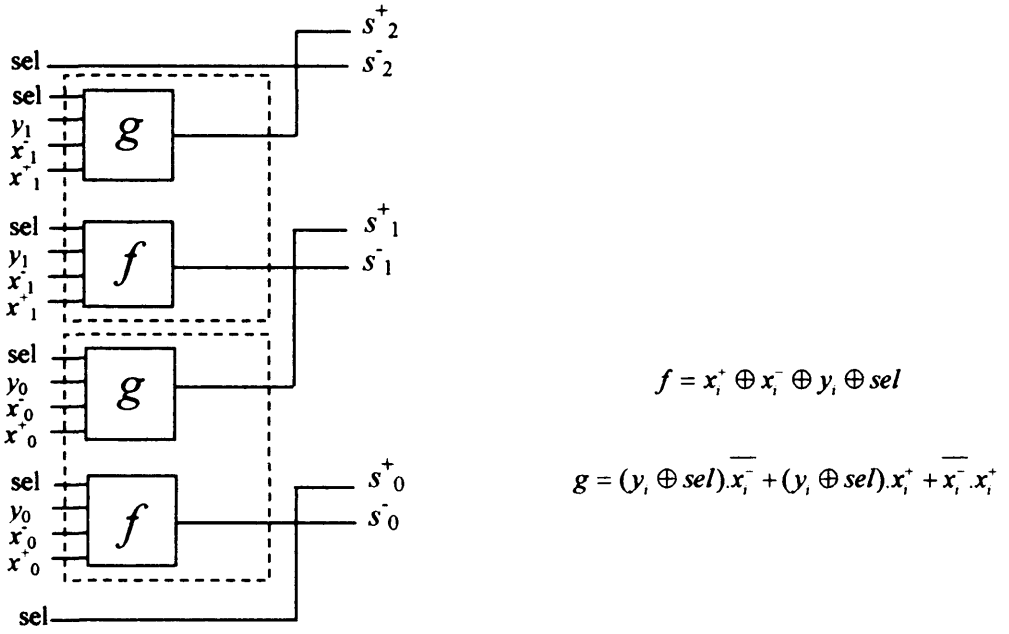


Figure 2.13. 2-digit signed-digit and conventional digit carry free adder/subtracter

2.5.9 Signed-digit to signed-digit addition/subtraction

The final case of carry free addition that will be considered is a component that allows the addition/subtraction of two signed-digit operands depending on a select signal Valls [107]. This is the most complicated carry free component that is considered in this section. The single cell that adds and subtracts two signed digits is constructed from the ppm and mmp cells. A diagram of the cell structure is shown in figure 2.14.

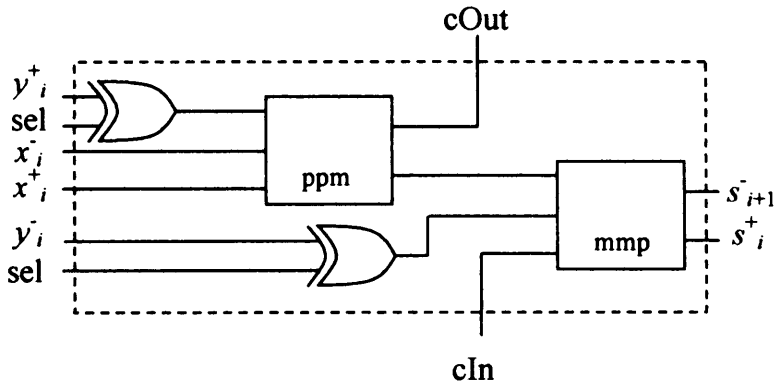


Figure 2.14. Signed-digit to signed-digit adder/subtracter cell (ssas)

A Virtex FPGA implementation of the cell in figure 2.14 requires 4 LUTs. 2 LUTs are used for the ppm cell as shown in figure 2.11 and 2 LUTs are used for the mmp cell as shown in figure 2.12. The sel line is blended into the LUTs for the ppm and mmp cells as only four inputs are required. Figure 2.15 illustrates the structure of a 2-digit signed-digit adder/subtracter that uses the cell structure shown in figure 2.14.

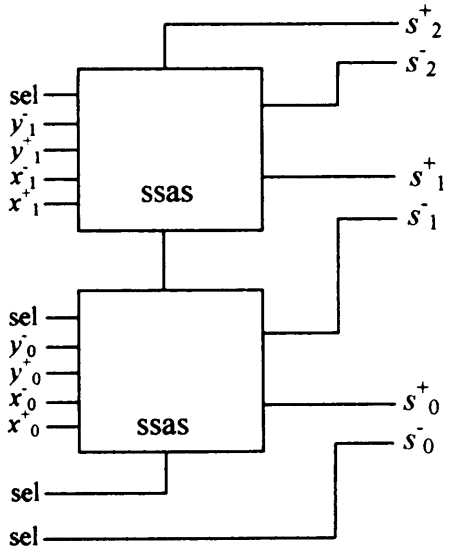


Figure 2.15. 2-digit signed-digit adder/subtracter

Signed-digit operations are most commonly used in online arithmetic Ercegovac [61], which is the name given to most significant digit first (MSDF) digit-serial arithmetic, where operands are fed into online operators one digit at time in a serial manner. Online arithmetic allows the calculation of operators (addition, multiplication, division, square root) to start while only knowing a limited number of the most significant digits of the input operands, which is particularly beneficial to the implementation of serial division and square root operations. The exact operation result cannot be known by only using a few of the most significant digits and therefore the redundancy in the digit set allows the running result to be corrected as more digits of the operands are fed into the system. Studies have been done on implementing online arithmetic on FPGA Dumas [100], Girau [101], Tisserand [102], McIlhenny [103], Tenca [104, 110], Lau [105, 106], Schneider [108] and P-Pascual [109] and we will not consider it further as it is outside the scope of this work.

2.5.10 Signed-digit and carry-save conversion to conventional representation

To convert from carry-save representation to conventional form a standard CPA such as a RCA can be used. The 'carry' operand is fed into one input of the CPA and the 'save' operand is fed into the other the result of the addition is the conventional representation. To convert from signed-digit representation a subtraction component is used. The subtrahend is the negative terms x^- and the minuend is the positive terms x^+. The result of the subtraction is a two's complement number of correct sign.

2.5.11 Conclusion

Table 2.4 summarises the area and simple delay calculations of a range of adder implementations. The carry-chain registering issue for pipelining adds irregularity to the efficient CSA designs and increases the area of the components, therefore the implementation will not be considered in creating other fixed-point components. The basic CSA designs are twice the size of the RCA designs and so the speed increase they offer is not felt to be worth the area sacrifice (especially for the word length of operands that will be required). The sign-digit adders, similarly to the basic CSAs, are faster than the basic RCA design but are twice the size so are disregarded. Carry-free addition components have a delay that is independent of operand width and that is less than equivalent RCA structures, however their size means they are only suited

to very high performance/long word-length FPGA designs where area is not a significant design issue. Carry-free addition techniques will not be considered further.

Adder type	Operand length	Area (slices)	Registered output area	Delay (components)	Latency
Ripple carry adder	N-bits	Ceil[1](N/2)	Ceil(N/2)	1 LUT, N cc[2], 1 XOR	1
LSDF bit-serial adder	N-bits	1	n/a	1 LUT	N
LSDF digit-serial adder (digit size D)	N-bits	Ceil(D/2)	n/a	1 LUT	Ceil(N/D)
[3:2] CSA basic	N-bits	N	N	1 LUT	1
Radix-4 CSA efficient	N-bits	Ceil(N/2)	Ceil(N/2) + Ceil(N/4)	1 LUT, 1 cc, 1 XOR	1
[4:2] CSA basic	N-bits	2N	2N	2 LUTs	1
[4:2] CSA efficient	N-bits	N	3N/2	1 LUT, 1 XOR	1
Signed-digit to conventional	N-digits	N	N	1 LUT	1
Signed-digit to signed-digit	N-digits	2N	2N	2 LUTs	1

[1] Ceil( ) is the mathematical ceiling function. [2] cc is an acronym for carry-chain.

Table 2.4. The delay and area of some carry propagate and carry free adders

2.6 Fixed-point multiplication

2.6.1 Unsigned multiplication

Consider two n and m-bit unsigned integer vectors shown in equations (2.4) and (2.5) respectively. X is the multiplier and Y is the multiplicand.

$$X = \sum_{i=0}^{n-1} x_i.2^i \qquad \text{(2.4)}$$

$$Y = \sum_{j=0}^{m-1} y_j.2^j \qquad \text{(2.5)}$$

The product of the values is shown in (2.6).

$$P = X * Y = \sum_{i=0}^{n-1} 2^i (x_i . \sum_{j=0}^{m-1} y_j.2^j) \qquad \text{(2.6)}$$

From (2.6) an n-bit by m-bit multiplication can be seen as the sum of n correctly scaled bitwise AND functions. All the bits that need to be summed in a multiplication can be arranged in an array called a partial product array. A partial product array and its corresponding dot diagram for a 4 by 4-bit multiplication of two unsigned operands X and Y are shown in figure 2.16.

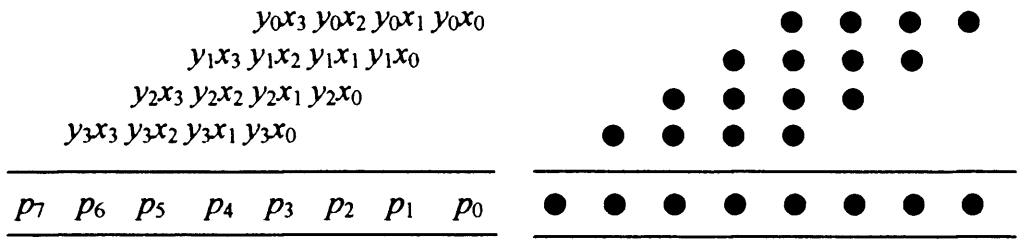


Figure 2.16. A 4 by 4-bit partial product array and dot diagram

An n by m-bit unsigned multiplication generates an $(n+m)$-bit product P, where P takes the value of (2.7).

$$P = \sum_{k=0}^{n+m-1} p_k.2^k \quad \text{(2.7)}$$

2.6.2 Signed multiplication

2.6.2.1 Sign-magnitude multiplication

Sign-magnitude values consist of a sign bit, which has the convention of being '1' for negative and '0' for positive and a magnitude part. To multiply two sign-magnitude values the sign-bit is XORed and the magnitudes are multiplied using an unsigned multiplier.

2.6.2.2 Two's complement multiplication

The most significant bit of a two's complement number is assumed to have a negative weighting. Equation (2.8) resembles an n-bit two's complement integer operand.

$$X = -x_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} x_i.2^i \quad \text{(2.8)}$$

When the leading bit is '1' the number is a negative and when it is '0' the number is positive. The multiplication of two two's complement numbers has two major

differences to unsigned multiplication. Firstly when summing the next partial product to the current partial result the partial result must be sign extended as opposed to being extended by adding zeros as in unsigned multiplication. Secondly the final partial product is subtracted from the partial result in two's complement multiplication, which is due to the sign of the final bit of the multiplier. The partial product array for two's complement multiplication is slightly different than for unsigned multiplication Ercegovac [61], and a 4 by 4-bit example is shown in figure 2.17.

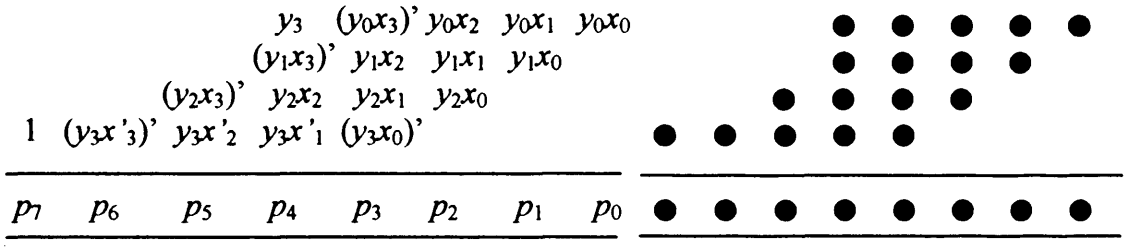


Figure 2.17. Transformed 4 by 4-bit two's complement partial product array

2.6.3 High radix partial product generation

To generate fewer partial products to sum, two or more multiplier bits can be multiplied with the multiplicand to create a single partial product. In general using N-bits of the multiplier results in a radix-2^N mode of operation. The problem with high radix operation is the multiples of the multiplicand that need to be generated. The radix-4 algorithm requires the multiples of $0Y$, $1Y$, $2Y$ and $3Y$ to be generated. The $1Y$ and $2Y$ multiples can easily be generated using shifting, but the $3Y$ multiple requires a carry propagate adder in its production. The radix-8 algorithm requires the multiples $0Y$, $1Y$, $2Y$, $3Y$, $4Y$, $5Y$, $6Y$ and $7Y$ to be able to be generated efficiently. Increasing the radix clearly increases the number of different possible multiples and the difficulty in generating them.

2.6.4 Multiplier structure

There are various options for the structure of a multiplier. These include: Serial-serial multiplication, Nibouche [98] gives an example of the FPGA implementation of serial-serial multiplication; Bit-serial parallel multiplication, several different FPGA implementations of bit-serial multiplication are given in Andraka [83], He [85], Li [87], Valls [93] and Adaos [95]; Digit-serial parallel multiplication, different FPGA

implementations of digit-serial multiplication are given in Smith [82], Satyanarayana [84], Rao [88], Lee [89, 91], Valls [92, 93], Kahalil [90] and Sansaloni [96]; Several array structure algorithms have been proposed for FPGA implementation including: The Majithia [70] and Bandhopadhyay [72] arrays by Saha [112]; The Pezaris [71] array by Saha [113], Stohmann [120] and Thornton [124]; The Baugh-Wooley [73] array by Saha [113] and Canik [117]; The Guild [69] array by Ruiz [119]. In parallel multiplication the multiplier and multiplicand arrive simultaneously and all partial products are generated in parallel MacSorley [65] and then summed in a tree structure using carry-save adders Wallace [67], Dadda [68] or ripple carry adders. This structure has the lowest latency and can be fully pipelined to allow a single cycle issue (i.e. a new pair of operands can be supplied to the multiplier on each clock cycle). Many authors have investigated the classic modified-Booth-encoding-Wallace-tree multiplier implementation on FPGA including Saha [112], Kumar [116], Singh [118], Ruiz [119], Tagzout [122], Laurent [123], Thornton [124] and Shah [125]. In the next section the feasibility of the implementation of Booth encoding will be briefly analysed and then the remainder of the multiplication section will focus on other methods of implementing parallel multipliers.

2.7 FPGA multiplication

2.7.1 Parallel multiplication

In parallel multiplication all partial products are generated in parallel and then summed in a tree structure using carry-save adders or ripple carry adders. Clearly the tree will be bigger and have more levels the more partial products there are to sum. A method of reducing the number of partial products is to use more bits of the multiplier in generating each partial product, which results in a high radix method of operation. Using one bit of the multiplier to generate the partial product results in the radix-2 algorithm, two bits results in a radix-4, three in a radix-8 and so on.

2.7.2 Dedicated Virtex FPGA multiplier logic

FPGAs are very popular arithmetic intensive algorithmic platforms. With this in mind the Virtex FPGA designers realised the importance of multiplication in designing efficient algorithms and therefore provided special logic elements to facilitate fast multiplication. The Virtex FPGA is built out of LUT columns with fast carry logic to

allow the efficient implementation of adders. The LUT columns can be configured as radix-2 partial product generators and can generate a partial product and sum it to another operand. However the designers also supplied a dedicated AND gate primitive to enable radix-4 partial product generators to be implemented in a single LUT slice. The radix-4 partial product generators can be configured in many different ways and in figures 2.18 and 2.19 the configurations that multiply a multiplicand by the digit sets {0,1,2,3} and {-2,-1,0,1} respectively are shown.

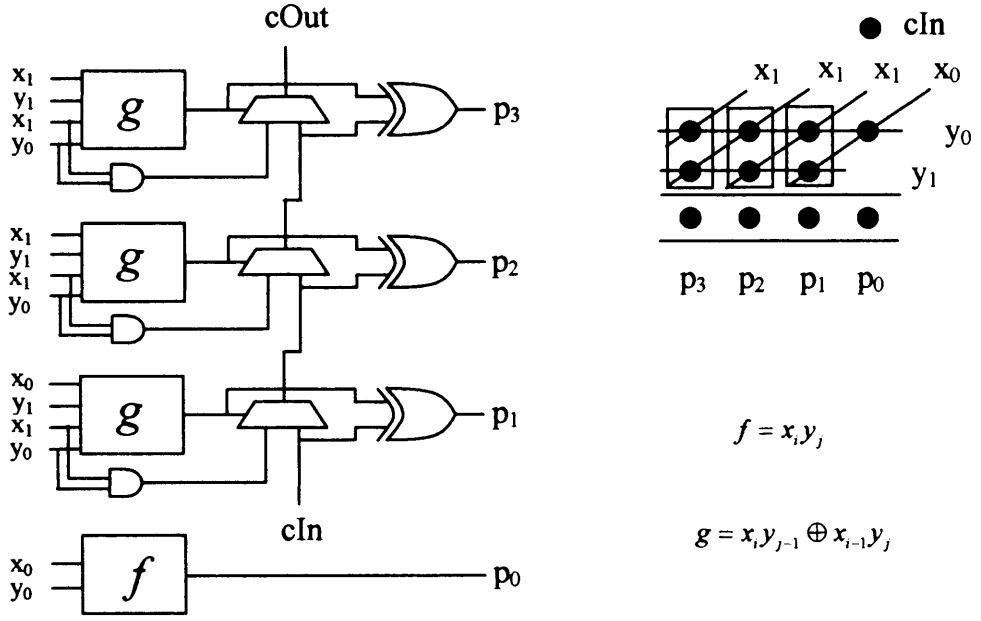


Figure 2.18. Signed two's complement Radix-4 partial product generator for the multiples (0,1,2,3) (y_1y_0)=(00,01,10,11)

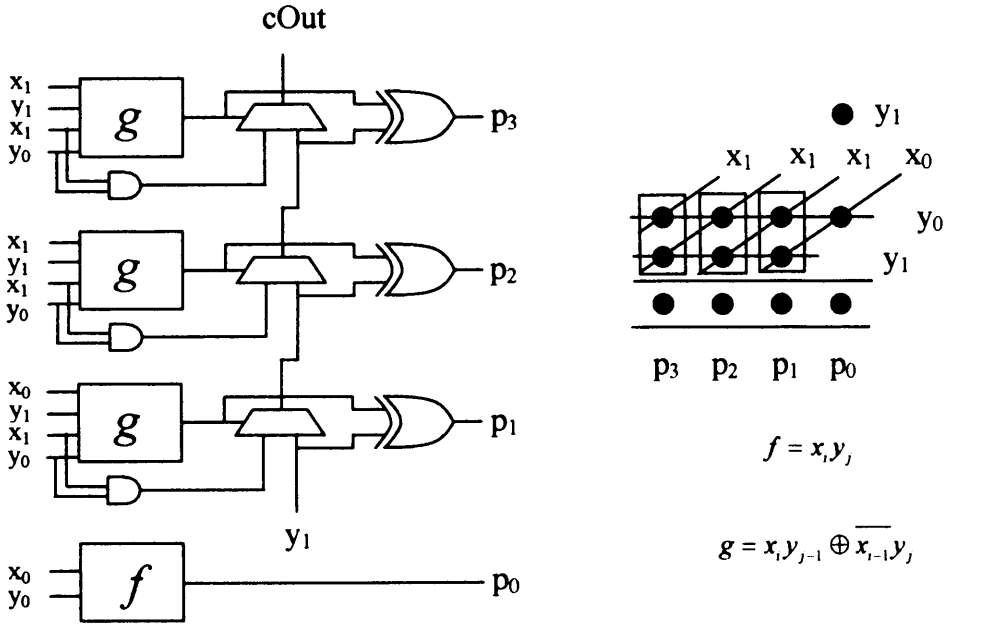


Figure 2.19. Signed two's complement radix-4 partial product generator for the multiples (-2,-1,0,1) (y_1y_0)=(10,11,00,01)

2.7.3 New radix-4 partial product generator mapping

A new partial product generator mapping that can multiply a multiplicand by a value from the digit set {1,2,3,4} is shown in figure 2.20.

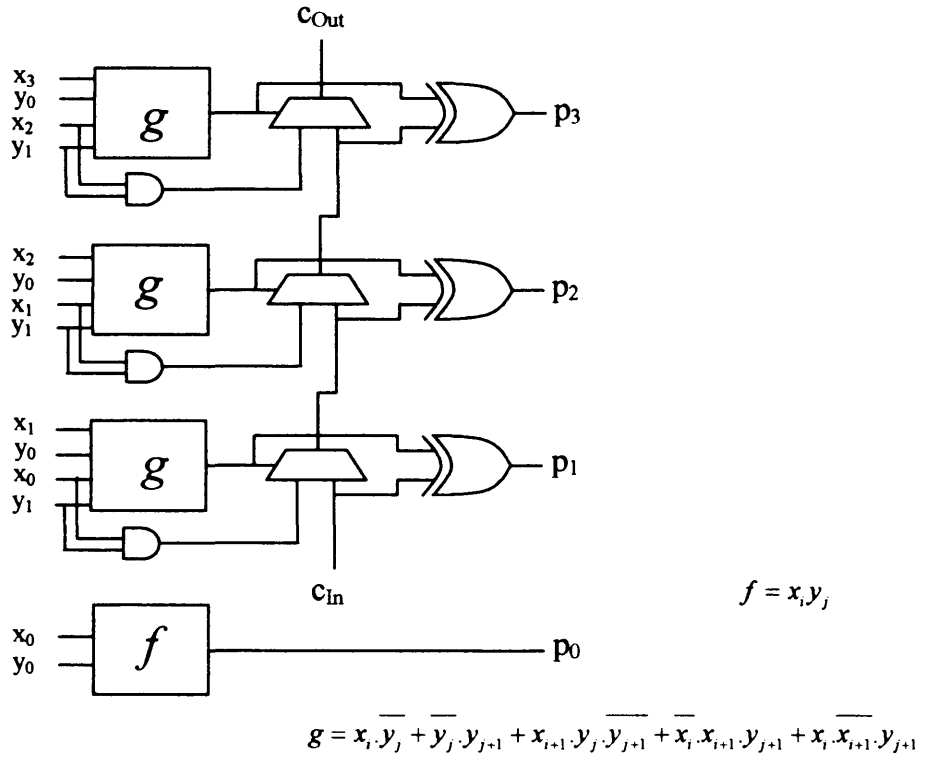


Figure 2.20. Radix-4 partial product generator for the multiples (1,2,3,4) (y_1y_0)=(01,00,11,10)

Multiplication where the multiplier is split up into two bit radix-4 digits fits very well on Virtex FPGAs as the underlying hardware, shown in figures 2.18 and 2.19, is designed to support it. However there is a logic technique called Booth encoding Booth [64], MacSorley [65] that is commonly used in ASIC designs to reduce the number of different multiples to develop in high radix multiplication.

2.7.4 Booth encoding

Booth encoding is a technique to recode the multiplier operand into a different digit set, which simplifies the multiples of the multiplicand that need to be created. The parallel recoding algorithm recodes an operand into a redundant digit set, which allows the recoding to be done in parallel. It must be stressed that Booth encoding does not reduce the number of partial products that need to be created, as many authors suggest, it merely simplifies their creation.

2.7.4.1 Radix-4 algorithm (Booth-2)

The radix-4 algorithm, assuming the multiplier is split into radix-4 digits, recodes the multiplier from the digit set {0,1,2,3} to the digit set {-2,-1,0,1,2}. Each recoded digit is produced by scanning three bits of the multiplier at a time overlapping the end scan bit of one group with the first scan bit of the next. Figure 2.21 illustrates the 3-bit scanning technique and table 2.5 illustrates the recoded digits depending on the scanned bits.

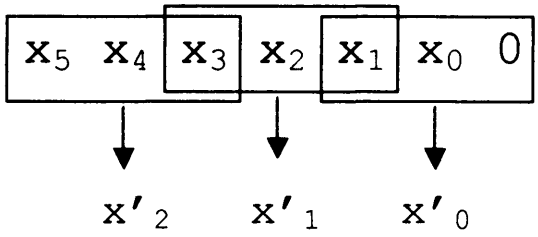


Figure 2.21. Booth-2 recoding of a 6-bit multiplier

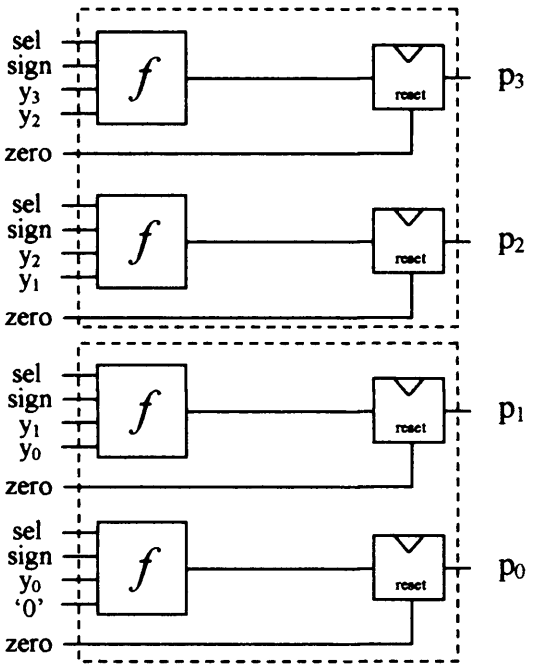
x_{i+1}	x_i	x_{i-1}	x'_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 2.5. Recoded digits of an operand according to the Booth-2 algorithm

2.7.4.2 New Booth-2 partial product generator mapping

The multiples of $\{-2Y,-Y,0,Y,2Y\}$ can be produced carry-free by configuring a LUT column as a signMux component to calculate the multiples $\{-2Y, -Y, Y, 2Y\}$ and utilising the dedicated register in each logic element to force a zero multiple. The hardware configuration to implement the novel signMuxZero component is shown in figure 2.22. Because the negative multiples are generated by inverting the positive multiples an extra correction bit is required in the LSB position, whose value is determined by the 'sign' input (this is the basic two's complement inversion technique). To continue the carry-free property of the multiplier the partial products

can be summed using carry-save adders in a tree structure (Wallace, Dadda). The most efficient carry-save adder is the [4:2] adder (see section 2.5.5) in which four partial products can be reduced to two in each level. Due to the structure of the [4:2] adder the most efficient numbers of partial products to sum are multiples of 2 e.g. 4,6,8... etc. Other quantities of partial products require [3:2] carry-save adders at some stage of the summation, which are less efficient than [4:2] adders.



$$sign = x_{i+1}$$

$$sel = x_i \oplus x_{i-1}$$

$$zero = \overline{x_{i+1}}.\overline{x_i}.\overline{x_{i-1}} + x_{i+1}.x_i.x_{i-1}$$

$$corrBit = \overline{x_{i-1}}.x_{i+1} + x_{i+1}.\overline{x_i}$$

$$f = y_i.\overline{sign}.\overline{sel} + \overline{y_i}.sign.\overline{sel} + \overline{y_{i+1}}.sign.sel + y_{i+1}.\overline{sign}.sel$$

Figure 2.22. The Virtex FPGA implementation of a new signMuxZero component

Figure 2.22 gives the equations for the selection bit, '*sel*', which selects the multiples of 1 or 2; the sign bit, '*sign*', which determines the sign of the 1 or 2 multiples; the zero bit, '*zero*', which determines whether the multiple is zero or not; and finally the correction bit, '*corrBit*', which is used in the two's complement operation to generate the negative multiples (note. when performing Booth encoding bit x_{i+1} is normally added in at the LSB position to perform the two's complement operation. We cannot use this principle here because we use registers to generate the zero multiple, thus an extra bit '*corrBit*' is used instead). The *sel*, *sign*, *corrBit* and *zero* bits are all based on the three bits of the multiplier currently being scanned and are the recoded bits. To implement the signMuxZero component without the registers a 5-bit function is required, which would double the logic required to implement the component.

2.7.4.3 Radix-8 algorithm (Booth-3)

The radix-8 algorithm requires a 4-bit scan of the multiplier to convert the digit set form {0,1,2,3,4,5,6,7} to {-4,-3,-2,-1,0,1,2,3,4}, thus eliminating the need to generate the $5Y$, $6Y$ and $7Y$ multiples. Figure 2.23 illustrates the 4-bit scanning technique of a 12-bit operand. Table 2.6 illustrates the Booth-3 recoded digits depending on the 4 scanned bits.

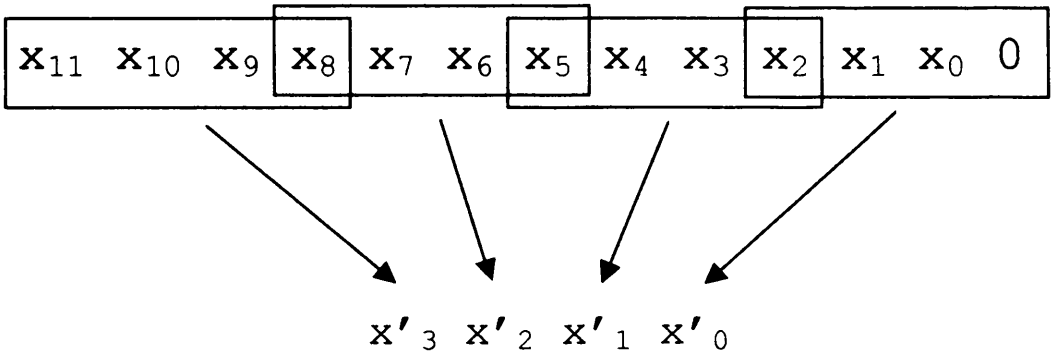


Figure 2.23. The Booth-3 scanning of a 12-bit operand

x_{i+2}	x_{i+1}	x_i	x_{i-1}	x'_i
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	2
0	1	0	0	2
0	1	0	1	3
0	1	1	0	3
0	1	1	1	4
1	0	0	0	-4
1	0	0	1	-3
1	0	1	0	-3
1	0	1	1	-2
1	1	0	0	-2
1	1	0	1	-1
1	1	1	0	-1
1	1	1	1	0

Table 2.6. Recoding of an operand according to the Booth-3 algorithm

A radix-4 partial product generator can be used to generate the multiples of {1,2,3,4}. As for the signMuxZero component each register can be configured to generate the 0 multiple, which allows all the positive multiples to be created. The main disadvantage

of the radix-8 Booth-3 algorithm is that a double inversion cannot be implemented in a single logic element. As an example consider the situation where two multiples $-4Y.8^i$ and $-4Y.8^{i+1}$ are to be generated and summed. If two radix-4 partial product generators supply the multiples $4.Y.8^i$ and $4.Y.8^{i+1}$ to an adder then the adder must negate the two operands and sum them. However due to a limitation in the structure of a Virtex logic element the double negation and corresponding addition of two operands is not possible. This is known as the double subtract problem described by Courtney [126] and cannot be solved without using extra hardware in the adder tree or leaving the result incorrect and correcting it later in another operation. Due to this fact the radix-8 algorithm is disregarded.

2.7.4.4 Higher radix algorithms

Higher radix algorithms do not have any benefit due to the difficulty of generating the multiples and the extra logic required to recode the multiplier.

2.7.5 Basic Virtex FPGA multiplication structure

The Virtex radix-4 partial product generators can implement multiplication by the digit set {0,1,2,3}, which is needed when multiplying a multiplicand by any of the radix-4 digits of a two's complement multiplier except the most significant digit. The most significant digit can belong to the digit set {-2,-1,0,1} and therefore needs a slightly modified partial product generator. Using the two partial product generators and a carry propagate adder tree gives the standard multiplier implementation. Consider an 8 by 8-bit signed multiplier where the multiplier A is split into four radix-4 digits. Figure 2.24 shows the multiplier structure, which consists of four partial product generators and a 3-ripple-carry adder tree.

2.7.6 Summary

Despite the benefits of Booth encoding it is not beneficial to Virtex FPGA designs for three reasons. Firstly a radix-4 partial product generator can already be implemented in a single LUT column. A sign/mux/zero component that allows the multiples {-2,-1,0,1,2} to be generated must always be registered and occupies the same area as the radix-4 partial product generator. The second reason is that a double inversion cannot be implemented in a single logic element, which prevents the Booth-3 algorithm from being implemented efficiently. Thirdly the extra logic to implement

the Booth recoding adds to the design area. However, in implementing a pure carry-free multiplier the Booth-2 encoded multiplier with subsequent Wallace tree is the best implementation structure, but it does have pipelining issues that cause an area penalty over the basic multiplier structure.

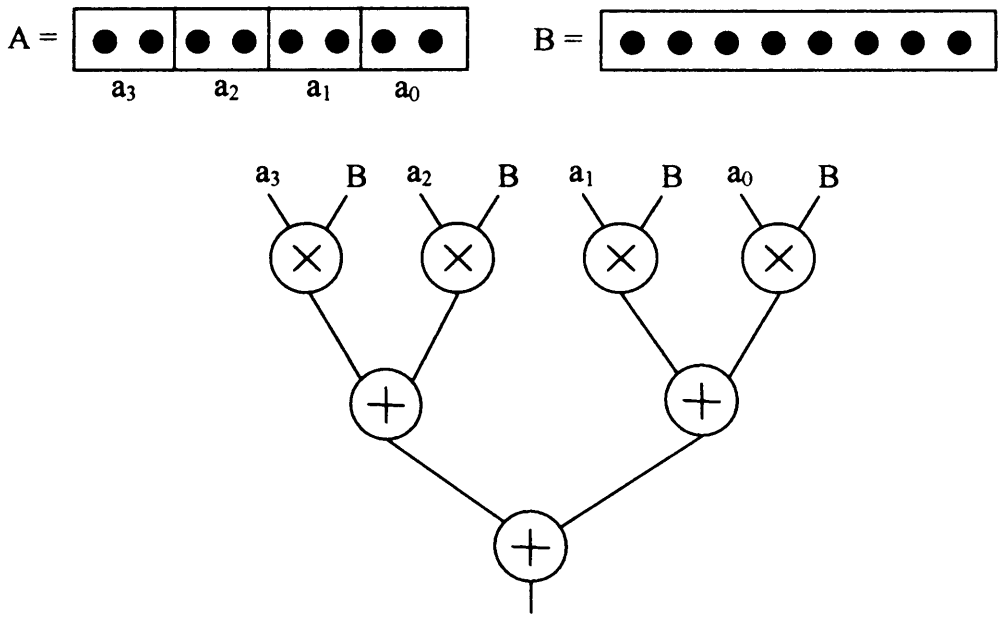


Figure 2.24. Basic Virtex FPGA implementation of an 8 by 8-bit signed multiplier

2.7.7 Virtex-II FPGA embedded multipliers

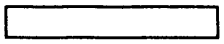
The Virtex-II and other FPGA families include dedicated multiplier blocks as a hardware feature that is embedded in the FPGA fabric. The multiplier blocks of the Virtex-II FPGA allow up to 18 by 18-bit signed two's complement multiplication or up to 17 by 17-bit unsigned multiplication. For many DSP applications the 18 and 17-bit operand width of the embedded blocks is sufficient, however it is desirable to have larger operand widths. For example, in the IEEE 754 standard [141] the single precision floating-point format requires a 24 by 24-bit multiplier to multiply two 24-bit significand values. Several options for the expansion of the embedded multiplier blocks are possible and in the next section the expansion techniques are studied.

2.7.7.1 Splitting large multiplications across multiple multiplier blocks

Clearly a 24 by 24-bit unsigned multiplication cannot be implemented with a single 18 by 18-bit multiplier. However, such a multiplication can be split amongst a

number of multiplier blocks using a divide and conquer technique. Consider an n-bit unsigned integer and an n-bit two's complement integer defined by (2.9) and (2.10) respectively.

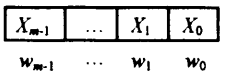
$$X = \sum_{i=0}^{n-1} x_i.2^i \quad \text{(2.9)}$$

$X =$ 

$$X = -x_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} x_i.2^i \quad \text{(2.10)}$$

X can be split into m appropriately weighted integers of widths $\{w_{m-1}, \ldots, w_0, w_{-1}\}$, where $w_{-1} = 0$, so that X can be written as in (2.11).

$$X = X_{m-1} + \ldots + X_0 \quad \text{(2.11)}$$

$X =$ 

A single term of the split operand is defined by (2.12) where u and v take the values defined by (2.13) and (2.14) respectively.

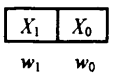
$$X_j = 2^u * \sum_{k=u}^{v-1} x_k.2^{k-u} \quad \text{(2.12)}$$

$$u = \sum_{l=-1}^{j-1} w_l \quad \text{(2.13)}$$

$$v = \sum_{l=-1}^{j} w_l \quad \text{(2.14)}$$

Consider two unsigned n-bit integers X and Y as defined in (2.9). X (Y) is split into two vectors X_0 and X_1 (Y_0 and Y_1) of length w_0 and w_1 respectively such that,

$$X = 2^{w_0}.X_1 + X_0 \quad \text{(2.15)}$$

$X =$ 

$$Y = 2^{w_0}.Y_1 + Y_0 \quad \text{(2.16)}$$

The product of X and Y can be written as (2.17) and expanded into (2.18).

$$X * Y = (2^{w_0}.X_1 + X_0)(2^{w_0}.Y_1 + Y_0) \quad \text{(2.17)}$$

$$X * Y = X_0Y_0 + 2^{w_0}X_0Y_1 + 2^{w_0}X_1Y_0 + 2^{2w_0}X_1Y_1 \quad \text{(2.18)}$$

Providing w_0 and w_1 are less than the maximum unsigned embedded multiplier width (17-bit in the Virtex-II FPGA case) the four products of (2.18) can be produced with

four multiplier blocks. The largest unsigned multiplication that can be performed with a two part split is a 34 by 34-bit multiplication. Larger operands require more splits and thus more products are developed, which implies that more embedded multiplier blocks are required.

2.7.7.2 Signed multiplication

Splitting signed two's complement multiplication across multiple multiplier blocks is very similar to unsigned multiplication except that partial products with a negatively weighted MSB must be sign extended when added. The multiplier blocks are 18 by 18-bit signed and this must be considered when tiling the blocks because there are different increase points where the number of multiplier blocks required to cover an area increases. Table 2.7 illustrates the increase points for signed and unsigned multiplication.

Size	Signed	Unsigned
17X17	1	1
18X18	1	4
19X19	4	4
:	:	:
34X34	4	4
35X35	4	9
36X36	9	9
:	:	:
51X51	9	9
52X52	9	16
53X53	16	16

Table 2.7. Divide and conquer signed and unsigned multiplier quantity increase points

Using multiple multiplier blocks to perform a multiplication is very efficient in terms of the logic elements. However using only the embedded multiplier blocks to perform the partial product generation can be very inefficient in terms of the utilisation of the embedded multipliers. Consider an unsigned 20 by 20-bit multiplication. If 4 multiplier blocks were used to perform the multiplication then, assuming we are not evenly tiling the multipliers on the partial product array, one multiplier would be used to perform a 3 by 3-bit multiplication and two of the multipliers would be performing

3 by 17-bit multiplications. Only one multiplier is being fully utilised while the others are being used with 4 or 18% efficiency.

2.7.7.3 Improving the utilisation of embedded multipliers for large multiplications

There are a number of ways to improve the utilisation of the embedded multipliers in Virtex-II FPGA designs. In this section the following four methods are described:

i) The two-way method originally proposed by Karatsuba [66]. It is commonly used in complex multiplication implementation.

ii) A method proposed by Chapman [127] of Xilinx, which uses a single embedded multiplier block and some LUT based logic.

iii) A method proposed by Beuchat [128], which also uses a single embedded multiplier block and some LUT based logic.

iv) Finally a new improved method, which uses a single embedded multiplier block and some extra LUT based logic.

2.7.7.3.1 The two-way method

The two-way method, also known as Karatsuba multiplication, is based on an identity that allows the number of multiplications used in the two split divide and conquer method to be reduced from four to three. Consider the multiplication of two n-bit operands X and Y each split into two sections of length w_0 and w_1. The multiplication of two such operands can be written as shown in (2.19), which is a repeat of equation (2.18).

$$X * Y = X_0Y_0 + 2^{w_0} X_0Y_1 + 2^{w_0} X_1Y_0 + 2^{2w_0} X_1Y_1 \quad \text{(2.19)}$$

Using the 'trick' of adding on a value and also adding on the negative of the same value and thus keeping the same value of the equation we can transform equation (2.19) via equation (2.20) into (2.21) or (2.22).

$$X * Y = X_1Y_1 2^{2w_0} + X_1Y_0 2^{w_0} + X_0Y_1 2^{w_0} + X_0Y_0 + X_0Y_0 2^{w_0} - X_0Y_0 2^{w_0} + X_1Y_1 2^{w_0} - X_1Y_1 2^{w_0} \quad \text{(2.20)}$$

$$X * Y = X_1Y_1(2^{2w_0} - 2^{w_0}) + (X_1 + X_0)(Y_1 + Y_0)2^{w_0} + X_0Y_0(1 - 2^{w_0}) \quad \text{(2.21)}$$

$$X * Y = X_1Y_1(2^{2w_0} + 2^{w_0}) - (X_1 - X_0)(Y_1 - Y_0)2^{w_0} + X_0Y_0(1 + 2^{w_0}) \quad \text{(2.22)}$$

There is a draw back in implementing equation (2.21) in that the sum of X_1 and X_0 or Y_1 and Y_0 could overflow and require an extra bit in its representation. To solve this problem Knuth [62], equation (2.22) can be implemented and X_1 and X_0 or Y_1 and Y_0 can be swapped as desired to avoid the overflow problem. The swapping of the operands saves a single multiplier bit. The extra logic required to swap the operands outweighs the multiplier savings, so for simplicity sake we will not consider the operand swapping. Figure 2.25 illustrates the weighting of the products in equation (2.21) and shows the duplication of the X_0Y_0 and X_1Y_1 products. Two of the products can be concatenated and this allows the products to be accumulated with three adders.

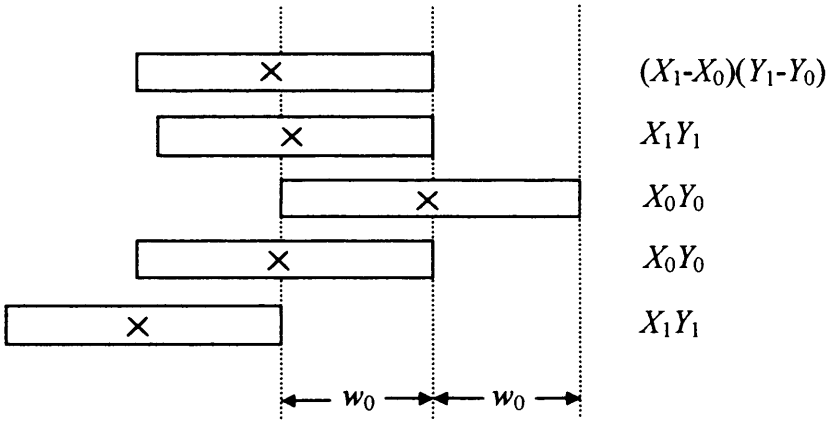


Figure 2.25. The two-way method partial products and their weightings

Figure 2.26 shows the hardware needed to implement the two-way multiplication. Compared to the divide and conquer method one multiplication has been traded for three much less costly adder components.

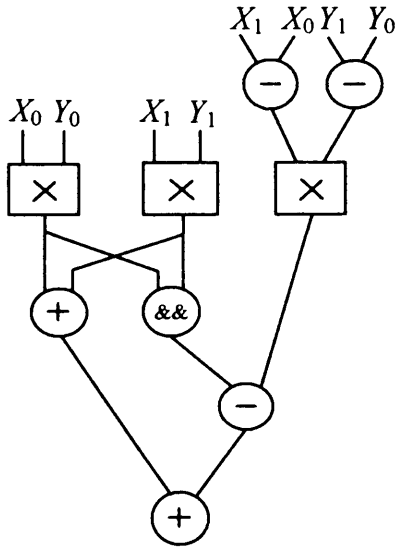


Figure 2.26. Hardware implementation of the two-way method

The Karatsuba technique can be applied to bigger multiplications. Consider a multiplication in which each operand would require four splits and thus 16 multiplications (e.g. a 60 by 60-bit unsigned multiplication) if implemented using the divide and conquer technique. Using one pass of the Karatsuba technique requires 3 multiplications. However the multiplications are too big to be implemented using single embedded multipliers. Each of these multiplications can be done using the Karatsuba technique thus the whole system requires 9 embedded multipliers and 16 adders. The Karatsuba technique is superior to the divide and conquer technique for multiplications that utilise the embedded multipliers with 100% efficiency. For example consider a 34 by 34-bit unsigned multiplication implemented with the divide and conquer algorithm, which uses 4 embedded multipliers each with 100% efficiency. The Karatsuba technique requires just 3 embedded multipliers each utilised with 100% efficiency, demonstrating the improvement of the technique.

2.7.7.3.2 Chapman method

The Chapman method is very similar to the divide and conquer method except that products that do not utilize the embedded multipliers with high efficiency are constructed from LUT based multiplications. Consider a 22 by 22-bit signed two's complement multiplication shown in figure 2.27.

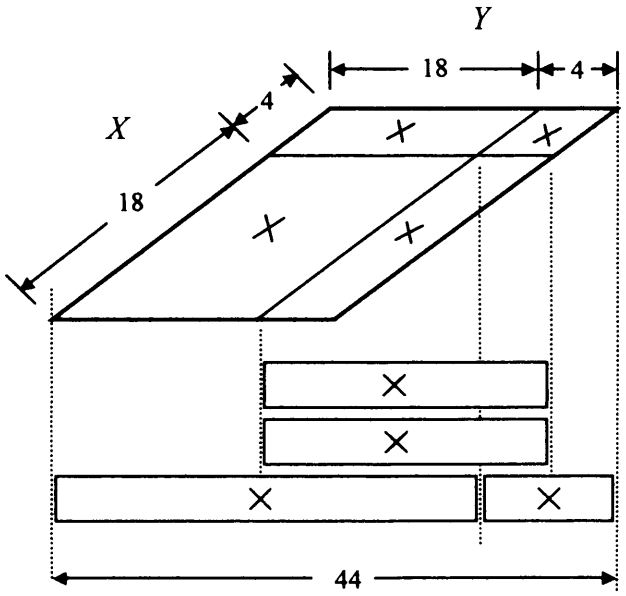


Figure 2.27. 22 by 22-bit signed two's complement multiplication

The 22 by 22-bit signed multiplication is split across four multipliers. The 18 by 18-bit multiplication is assigned to an embedded multiplier, the two 18 by 4-bit multiplications are assigned to two LUT based multipliers and finally the 4 by 4-bit multiplication is assigned to a LUT based multiplier. As for the divide and conquer method two adders are needed to sum the partial products to the final result. The scheme can easily be applied to signed and unsigned multiplication of different operand lengths and allows more multiplications to be placed on an FPGA device.

2.7.7.3.3 Beuchat method

The dots in a partial product array column have the same weighting regardless of the row they are in. This property can be used to transform the partial product array into a different shape. The partial product array for an unsigned 20 by 20-bit multiplication is shown in figure 2.28. The partial product array can be rearranged as shown in figure 2.29.

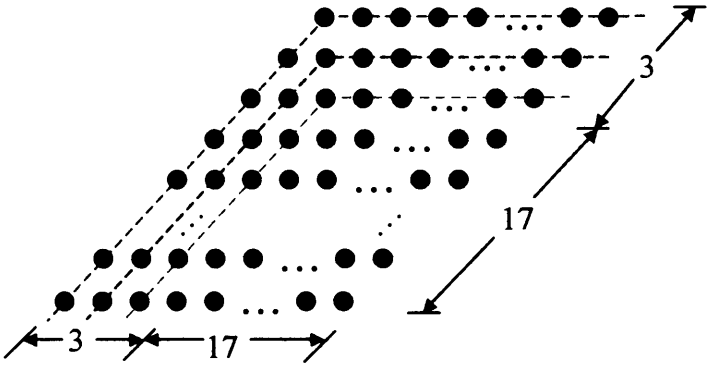


Figure 2.28. 20 by 20-bit partial product array with green, blue and red grouped bits

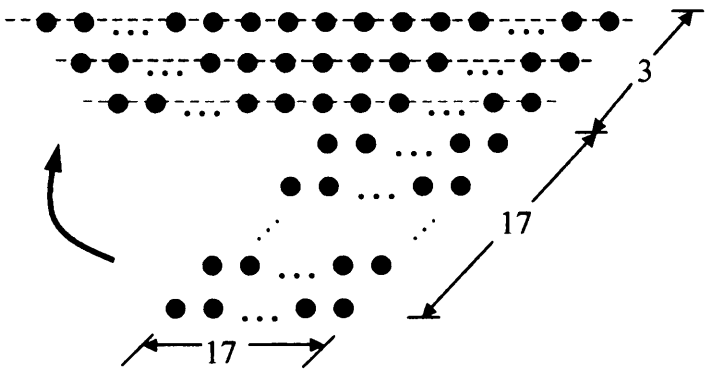


Figure 2.29. The folded partial product array with grouped bits

The 17 by 17-bit partial product array at the bottom of figure 2.29 can be produced and reduced to a single product by using a single embedded multiplier. The trapezoidal array of products can be generated and reduced to a single value using LUT based logic. It is unclear the precise way the authors reduce the trapezoidal array of products, but from their report of the method it appears they write high level VHDL to infer a tree reduction scheme consisting of carry propagate adders. They then let the synthesis tool map this high level structure to the FPGA primitives. It is unclear whether the synthesis tool takes advantage of the dedicated logic in the Virtex-II FPGA to facilitate partial product generation. The result form the trapezoidal partial product reduction and from the embedded multiplier can be added with a single adder and thus produce the final product.

2.7.7.3.4 New method

Unsigned multiplication

This new method builds on the work of Beuchat by improving the partial product arrangement and the partial product generation. Figure 2.30 reiterates the 20 by 20-bit unsigned partial product array of figure 2.28. Figure 2.31 shows the new partial product arrangement.

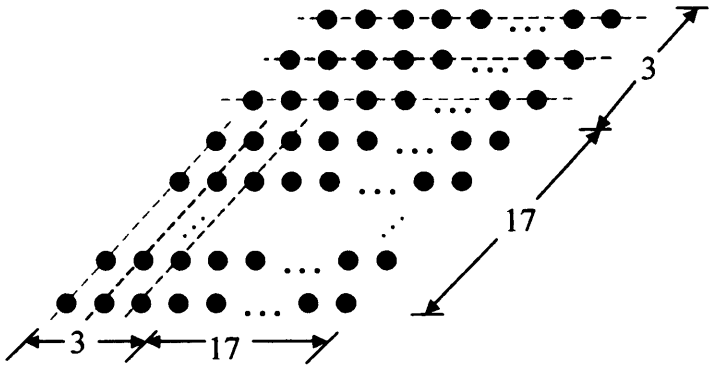


Figure 2.30. 20 by 20-bit unsigned partial product array

The 17 by 17-bit partial product array at the bottom of figure 2.31 can be generated and reduced with a single embedded multiplier. The parallelogram group of partial products can be generated and reduced using LUT based logic. The arrangement improves on the method of Beuchat because all the partial products are the same length, which improves the layout and reduces the area of the component. The

dedicated logic of the FPGA can be taken advantage of to produce the partial products in a radix-4 fashion and thus increase the speed of the design while using the minimum amount of logic. The method can be expanded for multiplications larger than 34 by 34-bits where the embedded multiplier is assumed to be 34 by 34-bits and is constructed using either the two-way or divide and conquer method. The method is most suitable for multiplications where the multiplier and multiplicand are the same length. The method can be used for signed multiplication by modifying the partial product array.

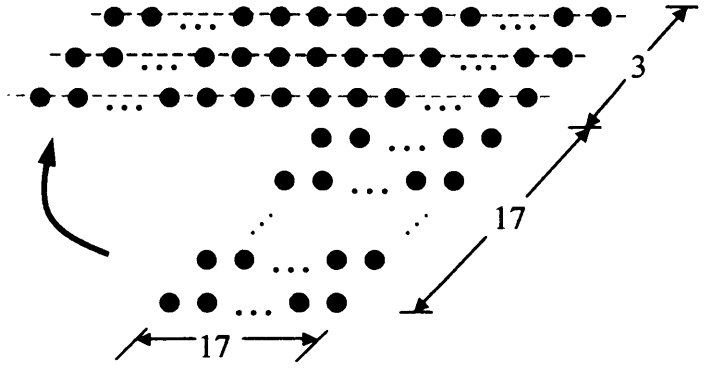


Figure 2.31. New folded 20 by 20-bit unsigned partial product array

Signed multiplication

The 4 by 4-bit multiplication shown in section 2.6.2.2 can be modified to perform unsigned multiplication so advantage can be taken of a dedicated signed multiplier to generate some of the partial product bits and sum them into a single product. Figure 2.32 shows the signed two's complement array after modification.

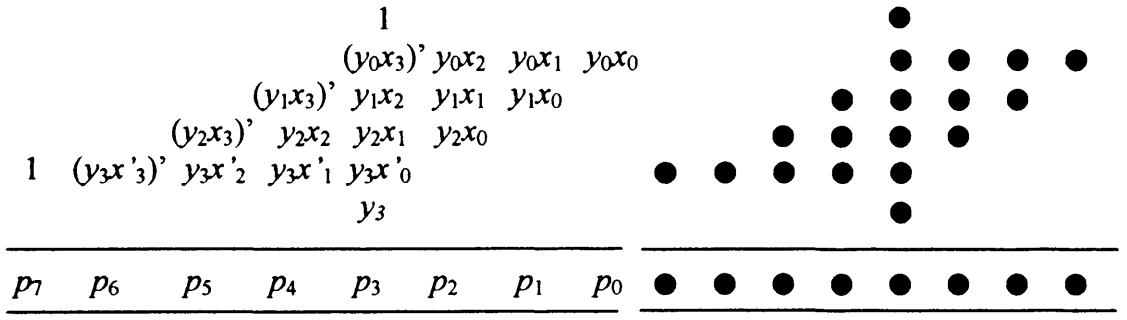


Figure 2.32. Modified signed 4 by 4-bit partial product array

Unfortunately the bit at the top of the array in figure 2.32 cannot be removed using an identity and therefore an extra adder is needed for the signed multiplication design that is not needed in the unsigned multiplier. The single bit at the bottom of the partial product array in figure 2.32 is the extra bit associated with the two's complement operation performed on the final row. This extra bit is automatically generated and handled internally by the dedicated signed multiplier. In the third column from the left an operation to generate two bits and then sum them is required. The logic structure of the Virtex family of FPGAs does not allow the elements in the third column to be generated and summed in a single logic element, thus an extra LUT is required to perform the function $(y_{n-1}x_m)'$ or $y_n x'_{m-1}$. The extra LUT is clearly a very small and justifiable logic increase, but it does add some irregularity to the multiplier structure.

The partial product array of a 20 by 20-bit signed multiplication is shown in figure 2.33. The extra bit at the bottom the array is automatically handled by the multiplier performing the 17 by 17-bit multiplication and can be ignored.

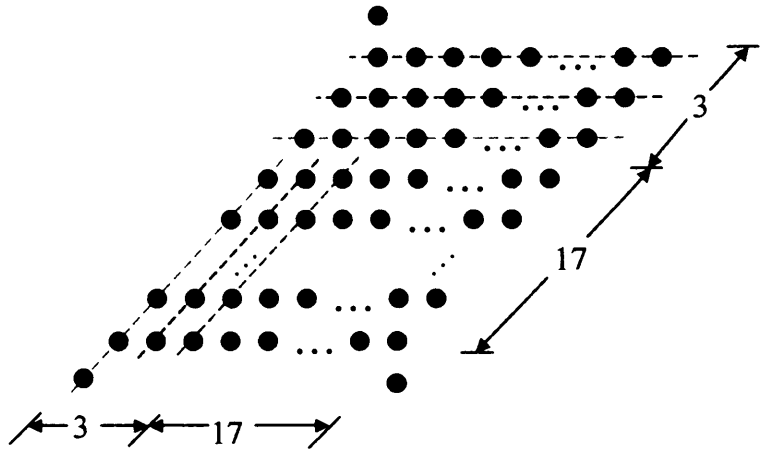


Figure 2.33. 20 by 20-bit signed multiplication partial product array

The partial product array can be folded as for the unsigned multiplication case shown in figure 2.31. The folded array is shown in figure 2.34. To ensure a correct final result the $m+n-1$ LSBs of the m by n-bit embedded multiplier result are added to the partial result generated by the parallelogram products reduced with the LUT based logic.

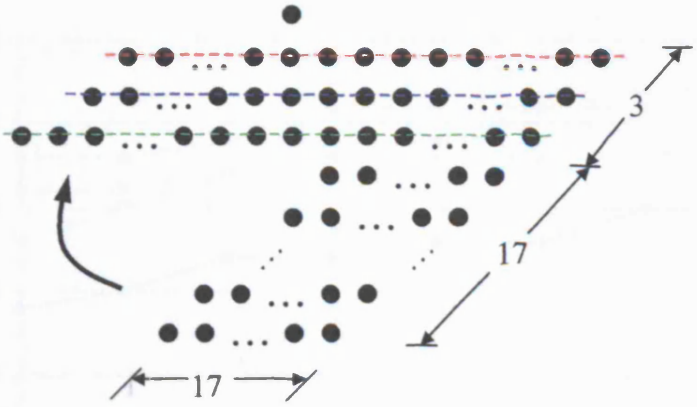


Figure 2.34. Folded 20 by 20-bit partial product array

2.7.7.4 Implementation results

Area and delay results for the Karatsuba and divide and conquer methods are given by Beuchat [128] and these results are used in the comparison of the different methods. Only unsigned multiplication will be considered. Signed multiplication for the proposed method has a very similar area and delay except that an extra adder is needed to add in the extra bit at the top of the partial product array shown in figure 2.34. Figures 2.35 and 2.36 show the area and delay of the three methods given by Beuchat [128] and the proposed method.

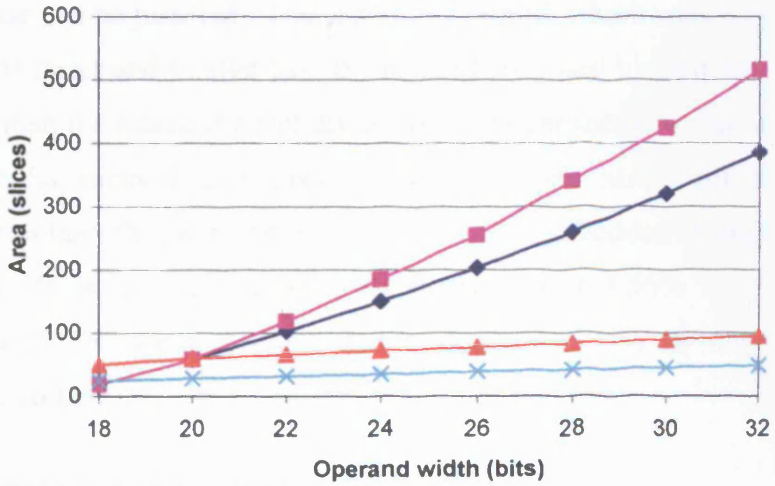


Figure 2.35. Area comparisons of various multipliers implemented on a Virtex-II FPGA: [■] Beuchat method; [x] Divide and conquer method; [▲] Two-way method; [♦] new method

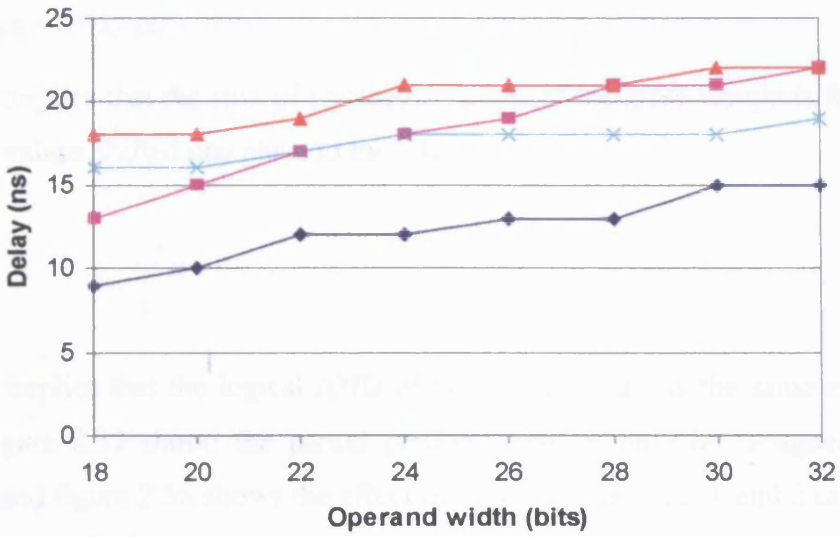


Figure 2.36. Delay comparisons of various multipliers implemented on a Virtex-II FPGA: [■] Beuchat method; [x] Divide and conquer method; [▲] Two-way method; [♦] new method

2.7.7.5 Comparison

The Karatsuba multiplication method uses three embedded multipliers, which is one less multiplier than the divide and conquer method but it is slightly slower and uses more slices (approximately 40). 40 slices is 0.78% of the total slices on a 1-Million gate Xilinx XC2V1000 FPGA and 1 multiplier is 2.5% of the total multipliers so the slice increase can be justified. The proposed method, which uses only one embedded multiplier, is faster and smaller than the method proposed by Beuchat. The method is also faster than the Karatsuba and divide and conquer methods, but uses more slices. The Karatsuba method uses approximately 80 less slices for a 24 by 24-bit multiplication than the proposed method but uses 2 embedded multipliers more. The slice trade off is justified because 80 slices is just 1.56% of the slices of an XC2V1000 FPGA, while 2 embedded multipliers is 5% of the total embedded multiplier quantity.

2.7.8 Squaring: a special case of multiplication

Squaring is a special case of multiplication Ercegovac [61] where both operands are the same length and take the same value. Two simple identities of the partial product array can be used to halve the logic required in implementing a squaring multiplier.

Identity 1

$x_i x_{i+1} + x_{i+1} x_i = 2^* x_{i+1} x_i$

Identity 1 implies that the sum of two equal values of the same weight is the same as one of the values shifted one place to the left.

Identity 2

$x_i x_i = x_i$

Identity 2 implies that the logical AND of two equal values is the same as the input value. Figure 2.37 shows the partial product array of an 8-bit unsigned squaring operation and figure 2.38 shows the effect of applying identities 1 and 2 to the partial product array of figure 2.37.

							x_0x_7	x_0x_6	x_0x_5	x_0x_4	x_0x_3	x_0x_2	x_0x_1	x_0x_0
						x_1x_7	x_1x_6	x_1x_5	x_1x_4	x_1x_3	x_1x_2	x_1x_1	x_1x_0	
					x_2x_7	x_2x_6	x_2x_5	x_2x_4	x_2x_3	x_2x_2	x_2x_1	x_2x_0		
				x_3x_7	x_3x_6	x_3x_5	x_3x_4	x_3x_3	x_3x_2	x_3x_1	x_3x_0			
			x_4x_7	x_4x_6	x_4x_5	x_4x_4	x_4x_3	x_4x_2	x_4x_1	x_4x_0				
		x_5x_7	x_5x_6	x_5x_5	x_5x_4	x_5x_3	x_5x_2	x_5x_1	x_5x_0					
	x_6x_7	x_6x_6	x_6x_5	x_6x_4	x_6x_3	x_6x_2	x_6x_1	x_6x_0						
x_7x_7	x_7x_6	x_7x_5	x_7x_4	x_7x_3	x_7x_2	x_7x_1	x_7x_0							

Figure 2.37. 8 by 8-bit squaring partial product array

							x_0x_7	x_0x_6	x_0x_5	x_0x_4	x_0x_3	x_0x_2	x_0x_1		x_0
						x_1x_7	x_1x_6	x_1x_5	x_1x_4	x_1x_3	x_1x_2		x_1		
					x_2x_7	x_2x_6	x_2x_5	x_2x_4	x_2x_3		x_2				
				x_3x_7	x_3x_6	x_3x_5	x_3x_4		x_3						
			x_4x_7	x_4x_6	x_4x_5		x_4								
		x_5x_7	x_5x_6		x_5										
	x_6x_7		x_6												
x_7															

Figure 2.38. The reduced partial product array for squaring

Applying identities 1 and 2 has halved the number of partial product bits in the partial product array and will clearly require less logic to generate the square of a number in comparison to using a regular multiplier.

2.7.8.1 New FPGA squaring method using a single embedded multiplier

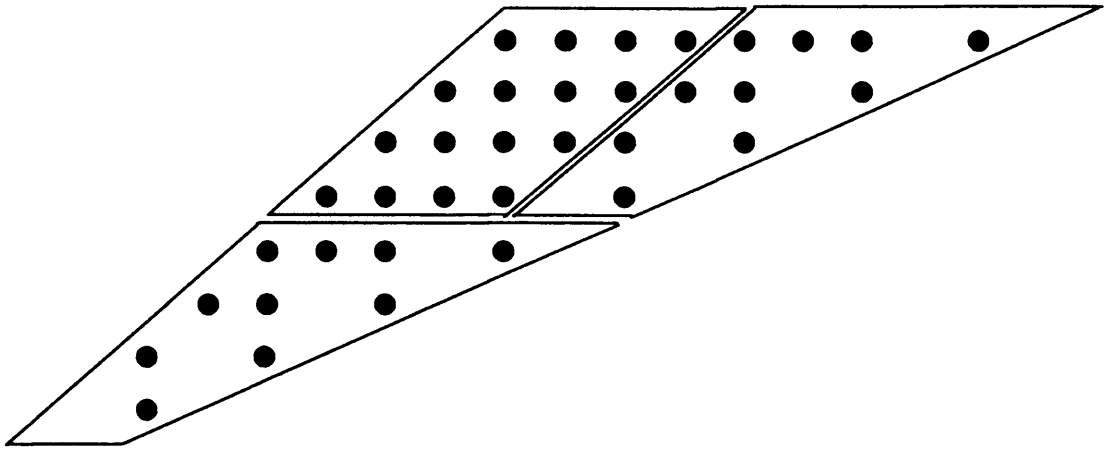


Figure 2.39. The reduced partial product dot diagram for squaring

As illustrated in the dot diagram of figure 2.39 the squarer partial product array can be decomposed into three separate sections. If the input operand to the squarer is split into two equal or almost equal sections (because of an odd number of bits) then the input operand squared is given by (2.23).

$$X^2 = 2^{2w_0} X_1^2 + 2(X_0 2^{w_0} X_1) + X_0^2 \qquad (2.23)$$

Equation (2.23) mathematically describes the partitioning of the partial product array shown in figure 2.39. The top left parallelogram shape of partial product bits can be generated and summed with a single 4 by 4-bit unsigned fixed-point multiplier. The embedded multiplier of the Virtex FPGA can be used to perform this reduction for unsigned squarers of between 18 and 34-bits. Adjacent to the parallelogram array are two triangular arrays, which require 4 by 4-bit 'squaring' multipliers to generate the partial product bits and reduce them to a single product. The array folding technique described previously can be used to simplify the structure of the 'squaring' multipliers and in figure 2.40 an example of how the technique can be applied to an 8 by 8-bit 'squaring' multiplier is given. The single bit at the bottom of the partial product array of figure 2.40 can be moved into the next column by using identity 3, which combines it with the bit directly above it.

Identity 3

$x_i x_{i+1} + x_{i+1} = 2x_i x_{i+1} + x'_i x_{i+1}$

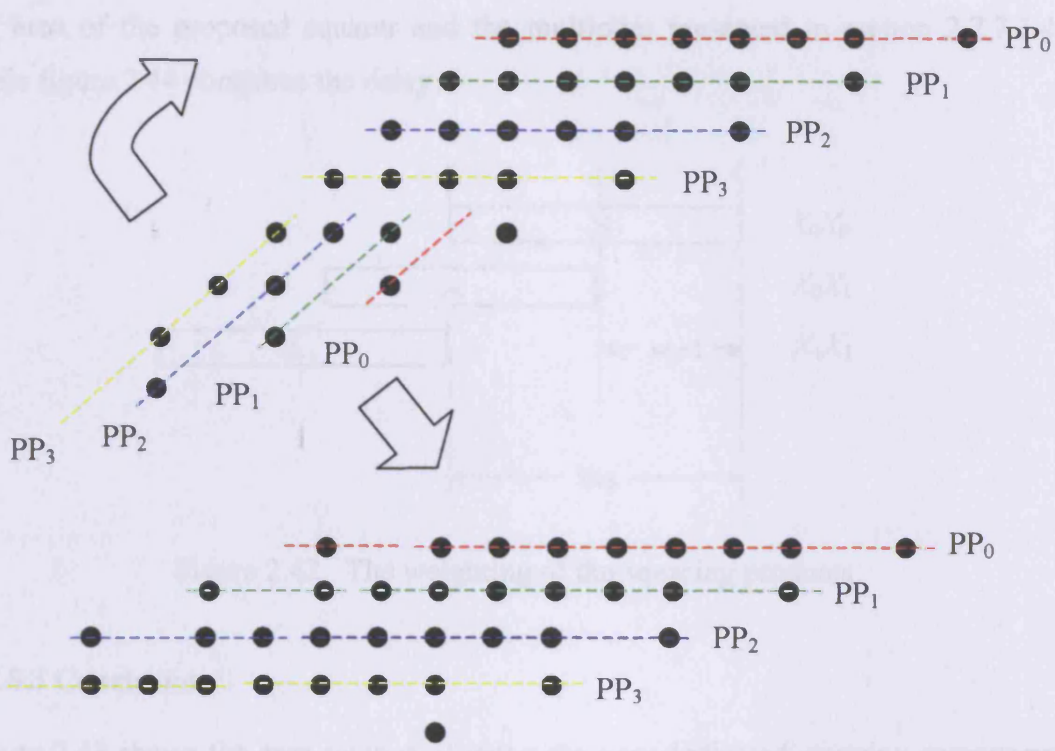


Figure 2.40. Applying the array folding technique to squaring

Using identity 3 transforms the partial product array of figure 2.40 into the array shown in figure 2.41. The partial product bits of the array in figure 2.41 have a regular layout and can be succinctly generated by the FPGA LUT based logic. Two 'squaring' multipliers and a single fixed-point multiplier are required in the implementation of the original squaring multiplier and these three products can be accumulated with a single adder because the two squaring multiplier products can be concatenated as shown in figure 2.42.

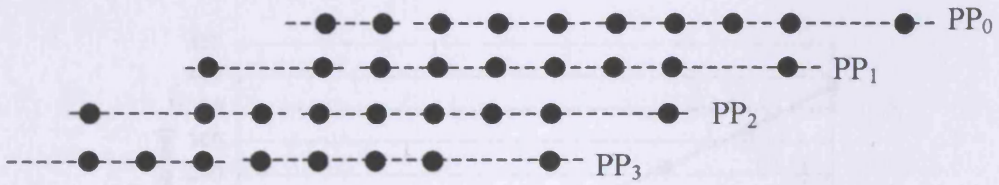


Figure 2.41. Transforming the partial product array of figure 2.40 by applying identity 3

2.7.8.2 Implementation results

The squaring component is compared against the proposed multiplier of section 2.7.7.3.4 to show the area savings that the component offers. Figure 2.43 compares

the area of the proposed squarer and the multiplier presented in section 2.7.7.3.4, while figure 2.44 compares the delay.

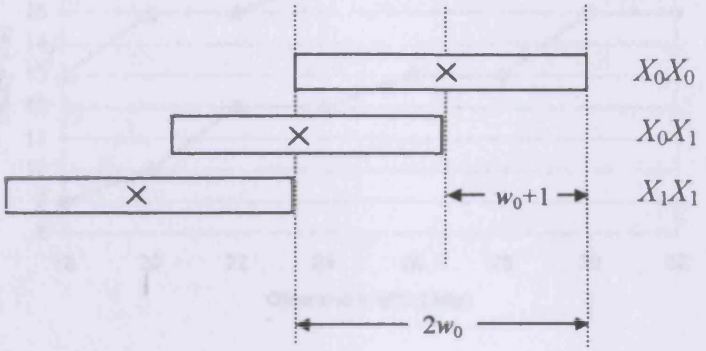


Figure 2.42. The weighting of the squaring products

2.7.8.3 Conclusion

Figure 2.43 shows the area savings of using the new dedicated squaring component. Interestingly there is a trade off point at the length 22-bits where below 22-bits it is more economical to use the regular multiplier, while above 22-bits the dedicated squarer offers the minimal area solution. Figure 2.44 shows that the area improvements are traded for a delay increase and above 22-bits a possible 33% delay increase is observed. The delay increase is primarily caused by the irregularity of the of the squaring component's adder tree. Other squaring components have been proposed for FPGA including Klotchkov [137] and Al-Kahalili [129], but these designs differ from the proposed design, as the FPGA technology does not contain embedded multipliers.

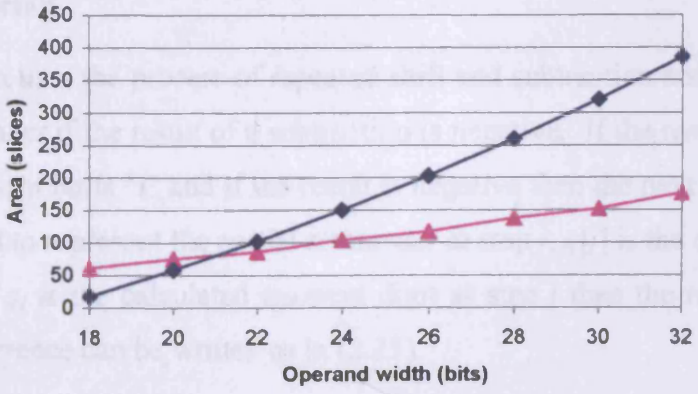


Figure 2.43. The area of the proposed single embedded-multiplier squarer [▲] and multiplier [♦]

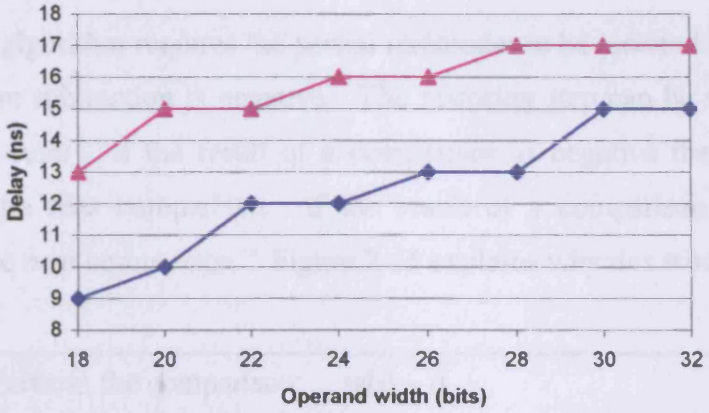


Figure 2.44. The delay of the proposed single embedded-multiplier squarer [▲] and multiplier [♦]

2.8 Fixed-point division

x/d is the division operation where x is the dividend and d is the divisor. The process of division calculates a quotient q and a remainder rem such that equation (2.24) is satisfied.

$x = q*d + rem$, where $rem<d$ ___(2.24)

2.8.1 Digit recurrence (sequential) division

This work considers the design of floating-point units so we are interested in fractional division. We will assume $x \in [0,1)$, $d \in [0.5,1)$ and follow the constraint that $x<d$, thus $q \in [0,1)$.

2.8.1.1 Restoring

Basic division uses the process of repeated shift and subtraction and a restoring of the partial remainder if the result of a subtraction is negative. If the result is positive then the next quotient bit is '1' and if the result is negative then the next quotient bit is '0'. If $w[j]$ is used to represent the partial remainder at step j, $q[j]$ is the calculated quotient at step j and q_j is the calculated quotient digit at step j then the restoring algorithm division recurrence can be written as in (2.25).

$w[j+1] = 2*w[j] - d*q_{j+1}$ ___(2.25)

2.8.1.2 Non-restoring

The restoring algorithm requires the partial remainder to be restored if the outcome of the comparison subtraction is negative. The restoring step can be avoided by using the following rule. "If the result of a comparison is negative then add instead of subtract for the next comparison. If the result of a comparison is positive then subtract for the next comparison." Figure 2.45 explains why this works.

(1) Perform the comparison: $w[j] - d$

(2) If the result is negative then we add on the next step. Remember the result is not restored: $(w[j] - d) + 0.5*d$

(3) The addition can be seen as a speculative subtraction of half the original divisor: $w[j] - 0.5*d$

Figure 2.45. An explanation of the non-restoring technique

Using the rule described in figure 2.45 leads to a non-restoring division algorithm. With the non-restoring algorithm the next quotient digit is selected as '0' if the comparison result is negative and '1' if the comparison result is positive (note. It is commonly assumed that the digit '-1' is used if the result is negative. We avoid using '-1' explicitly here by noting that $2^{-i} + 2^{-i-1} = 11_2*2^{-i-1}$ for positive results and $2^{-i} - 2^{-i-1} = 01_2*2^{-i-1}$ for negative results. Therefore the LSB is always '1' so can be made implicit and the bit juxtaposed to the LSB is created by using the rule described). The final remainder of a non-restoring algorithm could be negative where a correction of the remainder by adding on the value of the divisor is required (note. This is before shifting the final remainder to correct the subtraction). The recurrence for the non-restoring divider is the same as in (2.25).

2.8.1.3 SRT digit recurrence division

Restoring and non-restoring algorithms require a full-length comparison of the divisor with the partial remainder to determine the next quotient digit. A class of algorithms known as SRT, after Sweeny [306], Robertson [307] and Tocher [308] who independently proposed the algorithm, allow only a few of the MSBs of the partial remainder and divisor to be checked to determine the next quotient digit. A speedup

is thus obtained because the quotient digit selection no longer waits for the carry of the comparison to ripple to the MSB. The SRT class of algorithms requires the quotient to be selected from a redundant digit set. The redundancy allows a degree of freedom in selecting the next quotient digit as a wrong selection can be corrected in later iterations. Furthermore the SRT algorithm allows the partial remainder to be stored in redundant form (such as carry-save or signed-digit). This means the partial remainder can be updated at each algorithm step with a carry-free adder. For this work the partial remainder will be updated using a CPA and we will not consider the implications of keeping the partial remainder in redundant form. We now describe the fundamentals of the SRT algorithm for a general radix r omitting derivations. As stated the division process x/d produces a quotient and remainder that satisfy (2.24). Preferred division algorithms require equations (2.26) and (2.27) to hold. Thus the quotient has a range [0,1).

$$\begin{aligned} 0.5 \le d < 1 \\ 0 \le x < 1 \end{aligned} \quad \text{(2.26)}$$

$$x < d \quad \text{(2.27)}$$

The value of the quotient after j steps is given by (2.28), where $q[0]$ is determined by initialisation and r is the chosen radix of the implementation.

$$q[j] = q[0] + \sum_{i=1}^{j} q_i r^{-i} \quad \text{(2.28)}$$

The quotient is selected form a symmetrical signed digit set, which is shown in (2.29).

$$q_j \in \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a-1, a\} \quad \text{(2.29)}$$

Different digit sets for different radices have different redundancies. Based on the radix r, and the maximum digit set value a, the redundancy of a particular digit set for a particular radix can be defined by (2.30).

$$\rho = \frac{a}{r-1} \text{ where, } 1/2 < \rho \le 1 \quad \text{(2.30)}$$

Consider a radix-4 digit set. If a takes the value of 2 then the redundancy of the digit set {-2,-1,0,1,2} is given in (2.31).

$$\rho = \frac{2}{3} \qquad \text{(2.31)}$$

However if a takes the value 3 then the redundancy of the digit set, which is {-3,-2,-1,0,1,2,3} is given in (2.32).

$$\rho = \frac{3}{3} = 1 \qquad \text{(2.32)}$$

ρ cannot be less than or equal to 1/2 which implies that the digit set must contain more than r values. The general recurrence for SRT algorithms is given in (2.33).

$$w[j+1] = r.w[j] - d.q_{j+1} \qquad \text{(2.33)}$$

The digit selected for q_{j+1} depends on the truncated value of the shifted partial remainder and the divisor (2.34).

$$q_{j+1} = sel(truncate(r.w[j]), truncate(d)) \qquad \text{(2.34)}$$

The question now is "how do we know which value of the quotient digit set to select for q_{j+1}?" The selection intervals for a particular quotient digit value k i.e. $q_{j+1}=k$ are given by (2.35) and (2.36).

$$U_k = (k+\rho).d \qquad \text{(2.35)}$$

$$L_k = (k-p).d \qquad \text{(2.36)}$$

In equations (2.35) and (2.36) U_k and L_k are values of the shifted partial remainder $r.w_j$. So (2.35) and (2.36) can be read as: "Depending on the value of the divisor, the quotient digit q, and the redundancy of the digit set ρ the upper limit that the partial remainder can take to select q_{j+1} as k is given by $(k+\rho)*d$. Similarly the lower limit is $(k-\rho)*d$."

The upper and lower limits can be plotted on a graph of shifted partial remainder versus divisor called a pd-plot. The pd-plot helps determine the selection function and provides a value of how many bits of the shifted partial remainder and divisor need to be checked in determining the next quotient digit. An example of the use of a pd-plot is given in Appendix A where radix-2 and maximally redundant radix-4 SRT algorithms are constructed.

2.9 FPGA digit recurrence division

Xilinx Virtex FPGAs are constructed from columns of LUTs with a dedicated carry chain to allow fast carry propagation. The columns of LUTs can be used to implement ripple-carry adders in numerous configurations, multiplexers and unsigned radix-4 partial product generators. All these components can be used in digit recurrence division to generate the multiple of the divisor and add it to the shifted partial remainder (2.37). Different radices r, calculate $\log_2(r)$ bits of the quotient per cycle. If initially we ignore the quotient digit selection we can use a metric of LUT columns per quotient bit (LCPQ) to measure the efficiency of the multiple generation and shifted partial product update for a particular radix and redundancy.

$$w[j+1] = rw[j] - q_{j+1}d \quad \text{(2.37)}$$

2.9.1 LCPQ (LUT columns per quotient bit)

2.9.1.1 Radix-2

There are three main radix-2 division algorithms: Restoring, Non-restoring and radix-2 SRT. The restoring algorithm involves subtracting the divisor from the shifted partial remainder. If the result is positive then it is fed into the next stage, however if the result is negative then the input shifted partial remainder is fed into the next stage. This algorithm can be implemented using an adder and a multiplexer as shown in figure 2.46.

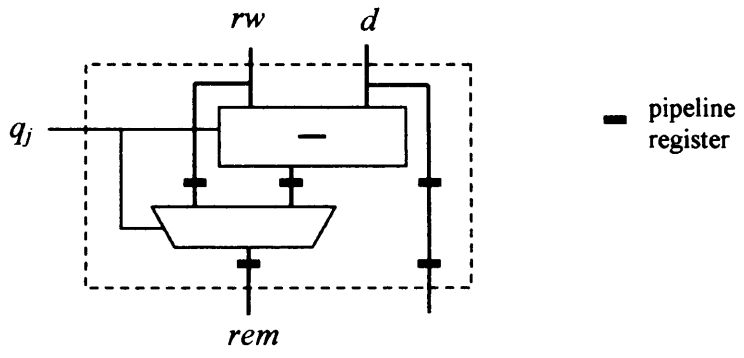


Figure 2.46. Hardware implementation of a restoring algorithm stage

The contents of the dashed box in figure 2.46 shows that two LUT column components are required per quotient digit produced. This gives restoring division an LCPQ value of 2. Non-restoring division does not require the multiplexer restoring

stage and instead the addition or subtraction of the divisor in each step is determined by the sign of the previous result. The hardware diagram of non-restoring division is shown in figure 2.47.

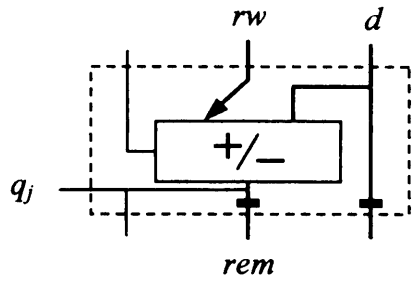


Figure 2.47. Hardware implementation of a non-restoring algorithm stage

The contents of the dashed box in figure 2.47 shows that one LUT column component is required per quotient digit produced. This gives non-restoring division an LCPQ value of 1. Radix-2 SRT division requires the divisor multiples of $\{-d,0,d\}$ to be created and added to the shifted partial remainder. A special adder configured as an addSubZero component, where zero means add zero, can be used to generate the divisor multiples and add them to the shifted partial remainder. Figure 2.48 shows the hardware implementation of a radix-2 SRT divider stage. The contents of the dashed box in figure 2.48 shows that one LUT column component is required per quotient digit produced. This gives radix-2 SRT division an LCPQ value of 1.

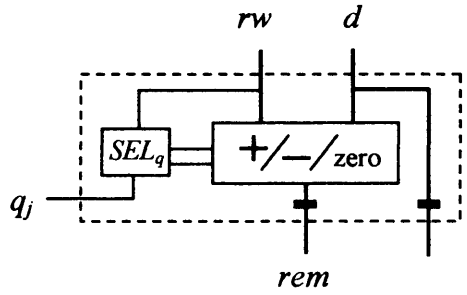


Figure 2.48. Hardware implementation of a radix-2 SRT algorithm stage

2.9.1.2 Radix-4

We consider two radix-4 algorithms. The first algorithm is the minimally redundant radix-4 SRT algorithm with non-redundant residual. The minimal redundancy means a quotient digit can be selected from the set {-2,-1,0,1,2} and thus the following multiples of the divisor need to be generated $\{-2d,-d,0,d,2d\}$. A 2-to-1 multiplexer

can be used to select between the multiples of d and $2d$. An add/sub/zero component can be used to add or subtract the selected multiple or add zero. The radix-4 algorithm produces one radix-4 digit per iteration and so in effect produces two bits of the result per iteration. The hardware diagram of the minimally redundant radix-4 algorithm is shown in figure 2.49.

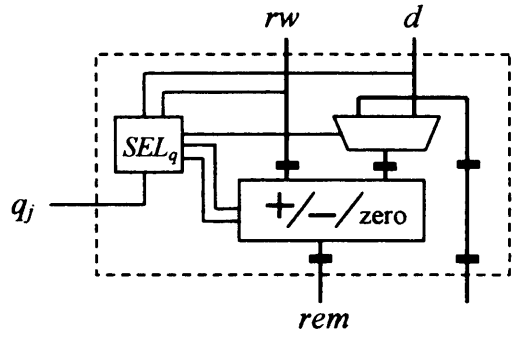


Figure 2.49. Hardware implementation of a min radix-4 SRT algorithm stage

The contents of the dashed box in figure 2.49 shows that two LUT column components are required per quotient digit produced. This gives radix-4 minimally redundant SRT division an LCPQ value of 1. The second radix-4 algorithm considered is the maximally redundant radix-4 SRT algorithm with non-redundant residual. The maximal redundancy means the quotient digits are selected from the set {-3,-2,-1,0,1,2,3} and thus the multiples of $\{-3d,-2d,-d,0,d,2d,3d\}$ need to be able to be created. The multiples $\{0,d,2d,3d\}$ can be generated by configuring a LUT column as a radix-4 partial product generator. An add/sub component can then be used to add or subtract the multiples from the shifted partial remainder. A diagram of the hardware required for the maximally redundant radix-4 divider implementation is shown in figure 2.50.

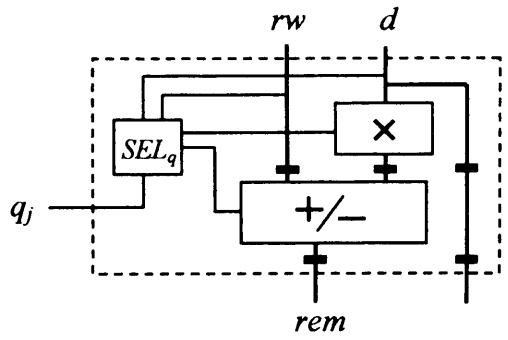


Figure 2.50. Hardware implementation of a max radix-4 SRT algorithm stage

The contents of the dashed box in figure 2.50 shows that two LUT column components are required per quotient digit produced. This gives radix-4 maximally redundant SRT division an LCPQ value of 1.

2.9.1.3 Radix-8

There are four different digit sets, not including over redundant and non-redundant digit sets that can be chosen for a radix-8 implementation. We will only consider the extremes of maximal and minimal redundancy as the two other choices in between do not offer any benefit for FPGA implementation. The minimally redundant radix-8 algorithm with non-redundant residual requires the quotient digit to be selected form the digit set {-4,-3,-2,-1,0,1,2,3,4} and thus the multiples of $\{-4d,\ldots,-d,0,d,\ldots,4d\}$ need to be able to be created. The radix-4 partial product generator can be configured to generate the multiples of $\{d,2d,3d,4d\}$. Using this in conjunction with an add/sub/zero component allows all the multiples to be generated. Figure 2.51 illustrates the hardware required for the minimally redundant radix-8 algorithm.

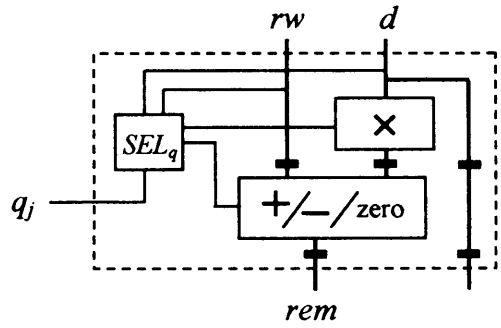


Figure 2.51. Hardware implementation of a min radix-8 SRT algorithm stage

The contents of the dashed box in figure 2.51 shows that two LUT column components are required per quotient digit produced. This gives radix-8 minimally redundant SRT division an LCPQ value of 2/3. Maximally redundant radix-8 algorithm with non-redundant residual requires the quotient digit to be selected from the digit set {-7,...,-1,0,1,...,7} and thus the multiples of $\{-7d,\ldots,-d,0,d,\ldots,7d\}$ need to be able to be created. A conventional LUT based multiplier can be used to generate the multiples and an add/sub component can be used to add or subtract the multiples of the divisor from the shifted partial remainder. Figure 2.52 shows the hardware implementation of the maximally redundant radix-8 algorithm. The contents of the dashed box in figure 2.52 shows that three LUT column components are required per

quotient digit produced. This gives radix-8 maximally redundant SRT division an LCPQ value of 1.

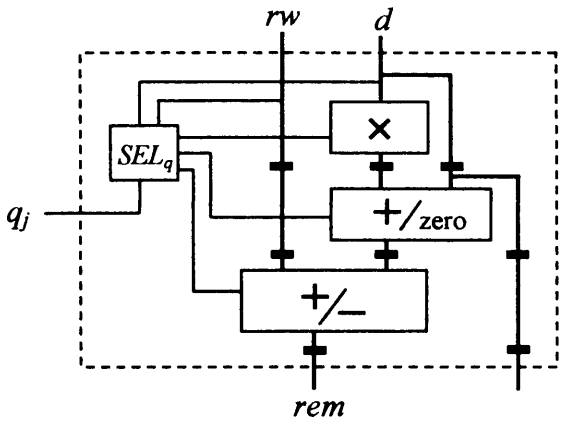


Figure 2.52. Hardware implementation of a max radix-8 SRT algorithm stage

2.9.1.4 Radix-16

As for radix-8 we will only consider the maximally and minimally redundant quotient digit set implementations. The minimally redundant radix-16 algorithm with non-redundant residual requires the quotient digit to be selected from the digit set $\{-8,\ldots,-1,0,1,\ldots,8\}$ and thus the multiples of $\{-8d,\ldots,-d,0,d,\ldots,8d\}$ need to be able to be created. By using the modified partial product generator the multiples $\{d,2d,3d,4d\}$ can be generated. Using an add/zero component the multiple $4d$ or 0 can be added on giving the multiples $\{d,\ldots,8d\}$. Finally an add/sub/zero component is used to update the shifted partial remainder. Figure 2.53 shows the hardware implementation of the minimally redundant radix-16 algorithm.

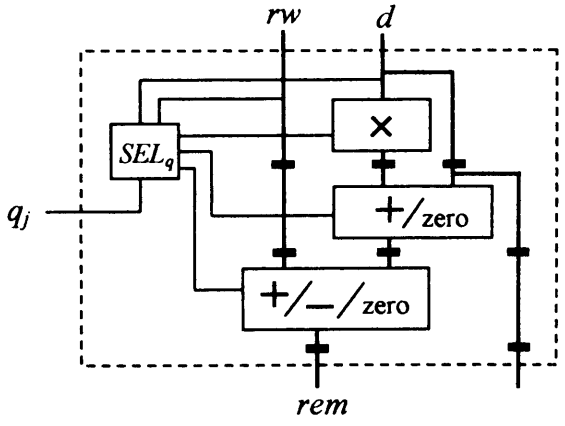


Figure 2.53. Hardware implementation of a min radix-16 SRT algorithm stage

The contents of the dashed box in figure 2.53 shows that three LUT column components are required per quotient digit produced. This gives radix-16 minimally redundant SRT division an LCPQ value of 3/4. The maximally redundant radix-16 algorithm with non-redundant residual requires the quotient digit to be selected from the digit set {-15,...,-1,0,1,...,15} and thus the multiples of {-15*d*,...,-*d*,0,*d*,...,15*d*} need to be able to be created. The multiples can be created using a LUT-based multiplier and a final add/sub component. Figure 2.54 shows the hardware implementation of the maximally redundant radix-16 algorithm. The dashed box in figure 2.54 shows that four LUT column components are required per quotient digit produced. This gives radix-16 maximally redundant SRT division an LCPQ value of 1.

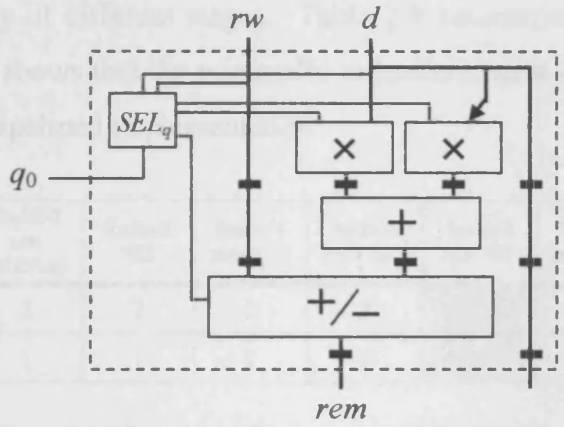


Figure 2.54. Hardware implementation of a max radix-16 SRT algorithm stage

2.9.1.5 Higher radices

Radices higher than 16 are not considered due to the complex nature of the quotient digit selection function, which would require very large memory components.

2.9.1.6 Summary

The LCPQ (LUT column per quotient bit) study is concluded by summarising the results as shown in table 2.8. From table 2.8 it can be seen that the minimally redundant radix-8 implementation offers the most efficient LCPQ value.

	Radix-2 restoring	Radix-2 non restoring	Radix-2 SRT	Radix-4 min SRT	Radix-4 max SRT	Radix-8 min SRT	Radix-8 max SRT	Radix-16 min SRT	Radix-16 max SRT
LCPQ	2	1	1	1	1	2/3	1	3/4	1

Table 2.8. LCPQ values for various divider implementations on Virtex FPGAs

2.9.2 Pipelining issues

To increase the throughput of the design stages, pipelining can be used. It is convenient to introduce pipelining stages at the output of each LUT column macro as registers can be incorporated into the LUT macros without increasing the logic area. The divisor needs to be pipelined, which can be done using a register for radix-2 stages and a shift register for higher radix stages. The register or shift register component adds a LUT column component to the area of each stage. In certain cases the shifted partial remainder also needs to be pipelined. The LCPQ metric can be modified to include the pipeline logic, which is denoted LCPQP (LUT columns per quotient bit pipelined). A metric of LPQ (latency per quotient bit) can be used to compare the latency of different stages. Table 2.9 summaries the LCPQP and LPQ metrics. Table 2.9 shows that the minimally redundant radix-8 implementation offers the most efficient pipelined implementation.

	Radix-2 restoring	Radix-2 non restoring	Radix-2 SRT	Radix-4 min SRT	Radix-4 max SRT	Radix-8 min SRT	Radix-8 max SRT	Radix-16 min SRT	Radix-16 max SRT
LCPQP	4	2	2	2	2	4/3	4	6/4	6/4
LPQ	2	1	1	1	1	2/3	1	3/4	3/4

Table 2.9. LCPQP and LPQ values for various divider implementations on Virtex FPGAs

2.9.3 Quotient digit selection

The quotient digit selection function takes as input the most-significant bits of the divisor and shifted partial remainder. From the inputs the function works out the next quotient digit. A pd-plot can be drawn and the selection region boundaries that determine the next quotient digit can be calculated by hand. The boundaries are chosen to lie in between upper and lower boundary lines, which cut across the pd-plot. Calculating the selection region boundaries can be complicated for high radix algorithms. A simple search program that takes the upper bounds and lower bounds of a selection interval and calculates the staircase selection function has been developed. The program is used to produce the selection boundaries in an automated fashion. Table 2.10 illustrates the number of shifted partial remainder bits *rw* and the number of divisor bits *d* that need to be used to determine the next quotient digit. The program uses a non-redundant residual.

Radix-2 SRT		Radix-4 min SRT		Radix-4 max SRT		Radix-8 min SRT		Radix-8 max SRT		Radix-16 min SRT		Radix-16 max SRT		Radix-32 min SRT		Radix-32 max SRT	
rw	d	rw	d	rw	d	rw	d	rw	d	rw	d	rw	d	rw	d	rw	d
3	0	6	3	4	1	9	6	6	3	11	9	8	4	13	11	10	5

Table 2.10. Precision requirements for SRT comparisons with non-redundant residual

The restoring and non-restoring algorithms do not require a selection function as the sign of the subtraction/addition result determines the next operation and quotient bit. Assuming a pure ROM is used for the quotient selection function (i.e. no logic minimisation) then the radix-2 SRT selection function requires a single LUT per selection bit and the maximally redundant radix-4 selection function requires one slice (2 LUTs) per selection bit, which are very small logic requirements. However the minimally redundant radix-4 and maximally redundant radix-8 algorithms require 16 slices per selection bit, which in comparison is the same amount of logic used to create a 32-bit adder. Furthermore 2(3)-bits are needed for the radix-4(8) quotient digit, which means a substantially extra amount of logic is needed for the quotient digit selection function. The logic requirement is expected to reduce with logic minimisation techniques but will compromise the delay of the circuit.

2.9.4 Basic stage delay analysis

Estimating the pre-implementation delay of FPGA designs is difficult due to the unpredictability of the place and route delays. In the basic delay analysis we only consider the 'logic delay' and ignore routing and fan out delay. Table 2.11 summarises the LUT, XOR, CC (carry-chain) and ROM delays of each stage. Table 2.12 provides delay information for the different primitives.

	Radix-2 restoring	Radix-2 non restoring	Radix-2 SRT	Radix-4 min SRT	Radix-4 max SRT	Radix-8 min SRT	Radix-8 max SRT	Radix-16 min SRT	Radix-16 max SRT
LUT	2	1	1	2	2	2	3	3	3
CC	1	1	1	1	1	1	1	1	1
XOR	1	1	1	1	2	2	3	3	3
ROM address	NA	NA	3	9	5	15	9	20	12

Table 2.11. LUT, CC and XOR delays for various divider stages

	LUT	CC	XOR
Delay	0.44ns	0.05ns/bit	1.27ns

Table 2.12. Estimated LUT, CC and XOR delays for Virtex II-4 FPGAs based on results from the Xilinx ISE Timing Analyzer software tool

Address width (bits)	4	5	6	7	8	9	10	11	12	13	14	15
Delay (ns)	0.44	0.7	0.89	1.38	1.70	2.14	2.57	2.76	3.34	3.6	3.95	4.3

Table 2.13. Estimated ROM delays based on address width

Table 2.13 shows how the delay of a ROM varies with address width. Combining the data in tables 2.11, 2.12 and 2.13 a delay prediction for the dividers can be created. Table 2.14 illustrates the logic delay prediction for 12, 24 and 36 bit operand dividers.

	Radix-2 restoring	Radix-2 non restoring	Radix-2 SRT	Radix-4 min SRT	Radix-4 max SRT	Radix-8 min SRT	Radix-8 max SRT	Radix-16 min SRT	Radix-16 max SRT
12 bits	33	28	33	30	28	33	31	>30	27
24 bits	80	70	80	66	64	71	68	>64	58
36 bits	142	126	142	110	107	114	109	>101	92

Table 2.14. Logic delay (ns) prediction for 12, 24 and 36 bit operand dividers

2.9.5 Summary

In summary the direct implementation of the minimally redundant radix-4 algorithm and the algorithms of radices above 4 are not feasible due to the very large selection function memory requirements. The most efficient radix-2 algorithm is the non-restoring algorithm, which has the lowest delay and smallest area. So the algorithm choice is narrowed to the non-restoring radix-2 and the maximally redundant radix-4. The area and delay results of the non-restoring radix-2 and maximally redundant radix-4 implementations are shown in figures 2.55 and 2.56 respectively. The graphs of figures 2.55 and 2.56 confirm the theory and show that the radix-4 algorithm has the lower delay but requires slightly more logic to handle the quotient digit selection function.

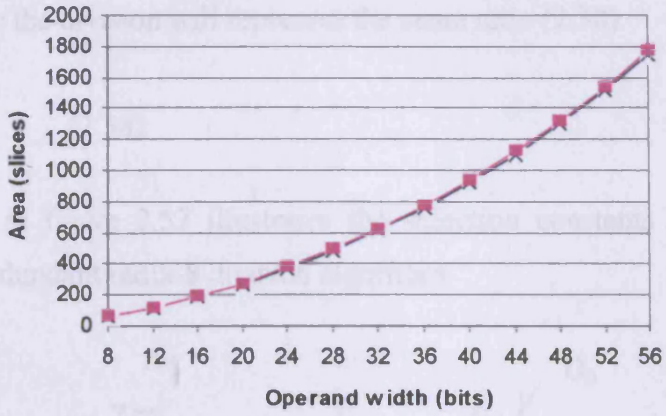


Figure 2.55. The area of some non-restoring radix-2 [x] and maximally redundant radix-4 [■] divider implementations

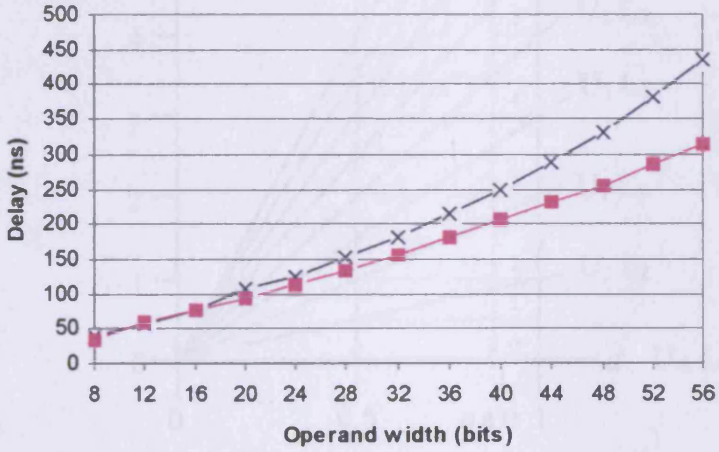


Figure 2.56. The delay of some non-restoring radix-2 [x] and maximally redundant radix-4 [■] divider implementations

2.9.6 Pre-scaling

The number of bits that need to be checked to determine the next quotient digit can be reduced by using a technique known as pre-scaling. By looking at the pd-plot (e.g. figure 2.57) it can be seen that if the divisor is constrained to be close to 1 then the quotient digit selection function can be made independent of the divisor and thus reducing the number of bits to supply to the function. The number of shifted partial remainder bits that need to be checked can also be reduced. To constrain the divisor to be close to 1 it needs to be multiplied by a constant c, that is determined by

inspecting the most significant bits of the divisor. If the dividend is also multiplied by the constant c the division will represent the same ratio (2.38).

$$\frac{x}{d} \equiv \frac{x \times c}{d \times c} \qquad (2.38)$$

The pd-plot in figure 2.57 illustrates the selection constants as red lines for the maximally redundant radix-8 division algorithm.

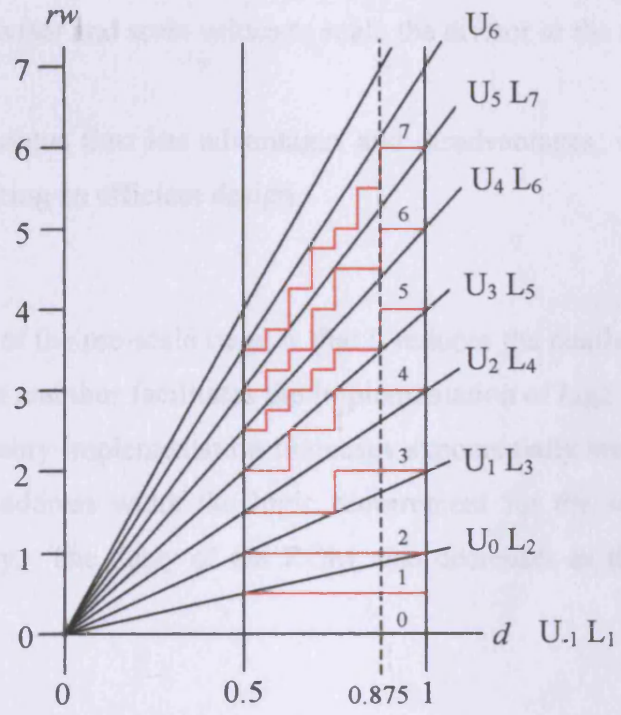


Figure 2.57. Maximally redundant radix-8 pd-plot with non-redundant residual

From figure 2.57 it can be seen that if the divisor is scaled to the region [0.875,1) then the selection function would be independent of the divisor and furthermore would only require four bits of the shifted partial remainder to be checked. Table 2.15 shows that three bits of the divisor need to be checked to calculate the scaling constant, which could very efficiently be implemented in a ROM with only a single LUT delay. The scaling value requires a 4-bit content ROM (the MSB is always '1' so does not need to be stored) and the scaling multiplier needs to be 5-bits in length.

Divisor MSBs	Scale value
0.1000	1.1100
0.1001	1.1001
0.1010	1.0111
0.1011	1.0101
0.1100	1.0011
0.1101	1.0010
0.1110	1.0000
0.1111	1.0000

Table 2.15. The divisor and scale values to scale the divisor to the range [0.875,1)

The pre-scaling technique thus has advantages and disadvantages, which need to be weighed up in producing an efficient design.

Advantages

The main advantage of the pre-scale stage is that it reduces the number of input bits to the selection function and thus facilitates the implementation of high radix algorithms. The LUT based memory implementation increases exponentially with address width, so by reducing the address width the logic requirement for the selection function reduces exponentially. The delay of the ROM also decreases as the address width decreases.

Disadvantages

The pre-scaling benefits are not for free. Firstly a ROM is needed to determine the scaling value. Secondly two multipliers are needed to scale the divisor and dividend. The scaling multiplication increases the length of the divisor and dividend, which in turn increases the area of the adders and partial product generators in each stage. Thirdly the remainder produced by the algorithm is incorrect and to obtain the correct remainder it must be multiplied by the reciprocal of the scaling factor Burgess [305].

2.9.6.1 A minimally redundant radix-8 divider with non-redundant residual and pre-scaling

From the LCPQ, LCPQP and LPQ metrics the minimally redundant radix-8 implementation is the most efficient implementation. The advantage of the minimally redundant radix-8 algorithm is compromised by the 15-bit selection function

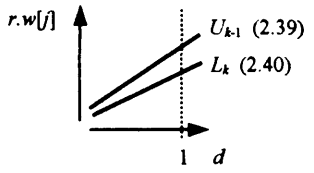
requirement. The following section details the application of the pre-scaling technique to the minimally redundant radix-8 algorithm.

2.9.6.2 Divisor scaling region

The selection region that determines the largest (magnitude) selection quotient digit also determines the region that the divisor needs to be scaled to to make the selection independent of the divisor. (U_{k-1}, L_k). We can develop the bounds for the divisor scaling interval. The upper bounds and lower bounds for the selection interval are described by (2.39) and (2.40) respectively.

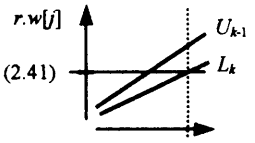
$$U_{k-1} = (k-1+\rho)d \quad \text{(2.39)}$$

$$L_k = (k-\rho)d \quad \text{(2.40)}$$



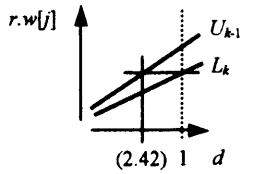
The shifted partial remainder value when d is 1, which is the upper range for the scaled divisor, is given in (2.41) by using (2.40).

$$L_k = (k-\rho) \quad \text{(2.41)}$$



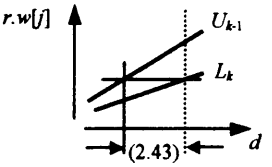
Setting U_{k-1} equal to L_k and solving (2.39) to calculate d, which is the lower range for the scaled divisor leads to (2.42).

$$d = \frac{(k-\rho)}{(k-1+\rho)} \quad \text{(2.42)}$$



The difference between the upper and lower bounds of the range for the scaled divisor is given in (2.43).

$$range = 1 - \frac{(k-\rho)}{(k-1+\rho)} \quad \text{(2.43)}$$



If the range is truncated after the most significant non-zero bit (so it is still greater than zero) then the quantity of fractional bits b that remain can be used to generate the range that the divisor must be scaled. This is shown in algorithm 1.

Algorithm 1

```
b := 0;
while (truncate(range,b) = 0)
      b := b+1;
end
```

Using the value of b generated by algorithm 1 the lower bound on the divisor can be created as shown in (2.44).

$$d \in \left[1-2^{-b},1\right) \quad \text{(2.44)}$$

Expression (2.44) shows the largest range that d can be scaled. However, this can require full width comparisons of the shifted partial remainder that defeats the purpose of the scaling. To over come this problem we need to generate an alternative, more convenient value for (2.41). This is done by calculating U_{k-1} using (2.39) and then successively truncating the value from 0 fractional bits to n fractional bits until the truncated term is greater than or equal to L_k. This procedure is shown in algorithm 2.

Algorithm 2

```
upper = U_{k-1};
lower = L_k;

b := 0;
while (truncate(upper,b) < lower)
      b=b+1;
end
```

Equation (2.45) shows the value of U_{k-1} that is truncated to b fractional bits of precision, where b is defined by algorithm 2.

$$\hat{U}_{k-1} = truncate(U_{k-1}, b) \quad \text{(2.45)}$$

Substituting (2.45) into (2.39) and rearranging creates a lower bound on the divisor scaling interval (2.46).

$$d = \frac{\hat{U}_{k-1}}{(k-1+\rho)} \quad \text{(2.46)}$$

The difference between the upper and lower bounds of the range for the scaled divisor is given in (2.47).

$$range = 1 - \frac{\hat{U}_{k-1}}{(k-1+\rho)} \quad \text{(2.47)}$$

Following the previous procedure (algorithm 1 and equation (2.44)) the range that the divisor can be scaled to can be developed.

2.9.6.3 Radix-8 divider scale region and scale values

Using the above process the interval bounds for the scaled divisor of the minimally redundant radix-8 algorithm have been developed and are shown in (2.48).

$$d \in \left[\frac{63}{64}, 1\right) \quad \text{(2.48)}$$

Calculating the scale values as shown in table 2.15 for the maximally redundant radix-8 case is a very time consuming task if done by hand. Instead a search program that determines the divisor splits and calculates the scale values has been written to automate the process. The program returns the split and scale values in a similar format to table 2.15 so they can be used in an FPGA memory component. Running the program to scale the divisor as shown in (2.48) shows that 6-bits of the divisor need to be checked to determine the scale value, which is 9-bits in length. Table 2.16 shows a snippet of the divisor selection intervals and the corresponding scaling value.

Divisor MSBs	Scale Value
0.1000000	1.11111000
0.1000001	1.11110001
0.1000010	1.11101001
:	:
0.1111101	1.00000100
0.1111110	1.00000000
0.1111111	1.00000000

Table 2.16. A selection of divisor and corresponding scale values for minimally redundant radix-8 division

A 64X8 bit ROM is required for the scaling function and can be implemented using 4*8=32 LUTs. Assuming the divisor and dividend are n-bits in length, two 9Xn bit multipliers are required to scale the divisor and dividend. These scaling multipliers can be constructed from the 18X18bit multipliers embedded in the FPGA fabric. From (2.45), b is 5, so 5-bits of the shifted partial remainder need to be checked for the quotient digit selection function. The quotient digit selection function input width

for each stage has been reduced from 15-bits to a much more feasible 5-bits. The 9X*n* bit multiplication increases the width of each multiple generator by 8-bits, which in turn increases the area of each stage by 16 LUTs.

2.9.6.4 Implementation results

The radix-8 divider has been implemented and is compared here with the maximally redundant radix-4 divider. Figure 2.58 compares the area of the two dividers and figure 2.59 compares the delay. It must be pointed out that the radix-8 divider also uses 2*((operand_width+16)/17) embedded multipliers.

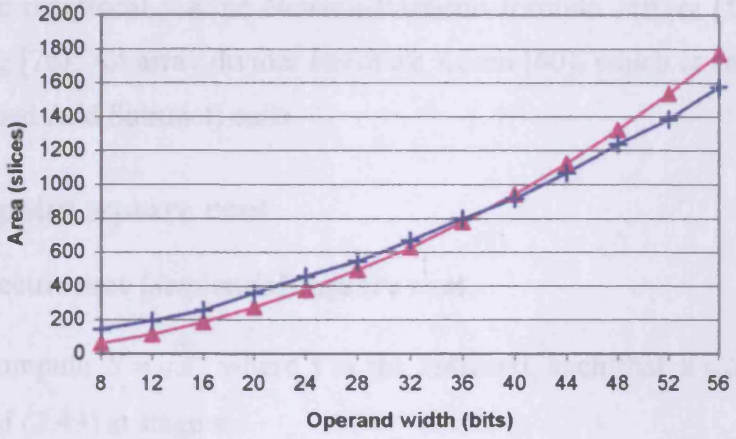


Figure 2.58. The area of some radix-4 divider [▲] and radix-8 divider [+] implementations

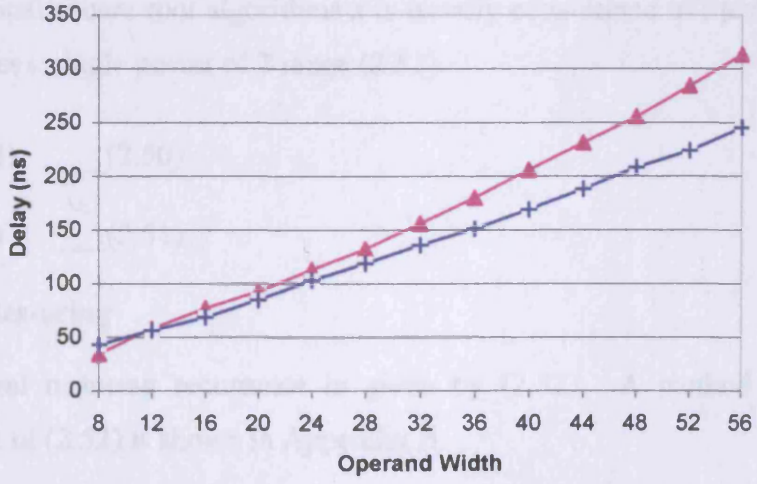


Figure 2.59. The delay of some radix-4 divider [▲] and radix-8 divider [+] implementations

It is clear that the radix-8 divider has superior speed, which increases with operand word length. The fact that the slice area of the two designs is equivalent and that the pipelining is more efficient are further plus point for the radix-8 divider. However, the radix-8 implementation requires a number of embedded multipliers, which might not be a worthwhile sacrifice for the speed gain.

2.9.7 Other division algorithms

Other forms of division exist Oberman [78], which have not been previously considered for FPGA implementation in the open literature. A few of these algorithms include: Convergent division proposed by Goldschmidt [74]; Division by calculating the reciprocal via the Newton-Raphson formula Jeffery [58]; High-radix division Wong [76]; An array divider structure Koren [60], which is constructed from CAS (Controlled Add Subtract) cells.

2.10 Fixed-point square root

2.10.1 Digit recurrence (sequential) square root

We wish to compute $S = \sqrt{x}$, where x is the radicand, such that $x = S^2 + rem$ and S has the form of (2.49) at stage n.

$$S[n] = 0.s_1 s_2 \ldots s_n = S[0] + \sum_{i=1}^{n} 2^{-i}.s_i \qquad \text{(2.49)}$$

For fractional square root algorithms x is usually constrained to the domain of (2.50), which gives a single power of 2 range (2.51).

$$x \in [0.25, 1) \qquad \text{(2.50)}$$

$$S \in [0.5, 1) \qquad \text{(2.51)}$$

2.10.1.1 Restoring

The general restoring recurrence is given by (2.52). A method of deriving the recurrence of (2.52) is shown in Appendix B.

$$w[j+1] = 2.w[j] - s_{j+1}.(2.S[j] + 2^{-(j+1)}) \qquad \text{(2.52)}$$

In (2.52) $w[j]$ is the partial remainder after j iterations, $S[j]$ is the partial result after j iterations, s_{j+1} is the $j+1^{th}$ digit of the result generated in cycle $j+1$. A speculative subtraction of $2.S[j]+2^{-(j+1)}$ from $2.w[j]$ is done to determine the next quotient bit s_{j+1}. If the result is positive then s_{j+1} is set to 1 and the result of the subtraction is kept as the next partial remainder. If the result is negative then s_{j+1} is set to 0 and the partial remainder result is not updated.

2.10.1.2 Non-restoring

As for division a non-restoring algorithm can be created. Here if the result of the speculative subtraction in cycle j is positive then s_{j+1} is set to '1' and in the next iteration a speculative subtraction of (2.53) is performed. However if the result is negative then s_{j+1} is set to '0' a speculative addition of (2.54) is performed.

$2.S[j]+2^{-(j+1)}$ ___(2.53)

$2.S[j]+3*2^{-(j+1)}$ ___(2.54)

If the final remainder of a non-restoring square root operation is negative then it needs correcting by adding on (2.55)

$2*S[n]+2^{-n}$ ___(2.55)

2.10.1.3 SRT square root

SRT square root is very similar to SRT division with the two main differences being the formation of the value to update the shifted partial remainder and the selection function for the next quotient digit. For the discussion that follows we assume a normalized fractional radicand with the range given in (2.50). The range of the argument given in (2.50) gives the result of the function $\sqrt{x}$ a range shown in (2.51). Each iteration step of the recurrence produces one digit of the result most-significant digit first. The value of the result after j iterations is given in (2.56).

$$S[j]=\sum_{i=0}^{n} s_i r^{-i} \quad \text{___(2.56)}$$

The general recurrence for the square root operation is given in (2.57).

$w[j+1]=rw[j]-2S[j]s_{j+1}-s_{j+1}^2 r^{-(j+1)}$ ___(2.57)

There are two important parts of the recurrence:

1. The first is the selection function that determines the next square root digit s_{j+1}. The next square root digit is chosen from a redundant digit set to allow reduced precision comparisons as was done for the division algorithm. The selection function depends on the most significant bits of the shifted partial remainder and the current square root result as shown in (2.58).

$$s_{j+1} = SELsqrt(truncate(r.w[j]),truncate(s[j])) \quad (2.58)$$

As for division the problem is in deciding the next quotient digit to select. The square root digit selection has boundaries just as in the division quotient digit selection. However, there is a small difference, which is the digit selection for the next square root digit varies for each iteration of the algorithm. Equations (2.59) and (2.60) show the upper and lower selection bounds for a particular square root digit value k and at a particular iteration j.

$$U_k[j] = 2S[j](k+\rho) + (k+\rho)^2 r^{-(j+1)} \quad (2.59)$$

$$L_k[j] = 2S[j](k-\rho) + (k-\rho)^2 r^{-(j+1)} \quad (2.60)$$

We can plot (2.59) and (2.60) on a pd-plot to highlight the overlap regions and develop a selection function for a particular radix and redundancy implementation. From (2.59) and (2.60) it appears that j curves need to be drawn for each interval bound. However by drawing the pd-plot with $U_k[\infty]$ and $L_k[1]$ a general selection function can be developed that holds for all iterations. If an array square root unit were to be implemented then the selection function could be programmed differently for each iteration stage and the problem of making a general selection function could be avoided.

2. The second important part of the recurrence is the formation of $F[j]$ that feeds the shifted partial remainder update adder (2.61).

$$F[j] = -(2.S[j].s_{j+1} + s_{j+1}^2.r^{-(j+1)}) \quad (2.61)$$

The formation of $F[j]$ can pose a problem and especially so for high radix unrolled implementations. Consider the radix-2 case where the next square root digit s_{j+1} is chosen from the digit set {-1,0,1}.

There are three possible scenarios:

$s_{j+1} = -1$

$$F[j] = +(2.S[j] - 2^{-(j+1)})$$

$s_{j+1} = 0$

$$F[j] = 0$$

$s_{j+1} = 1$

$$F[j] = -(2.S[j] + 2^{-(j+1)})$$

The addition or subtraction of the bracketed term is not a problem as the update adder can be configured in an add/sub style. The problem comes in creating the value inside the brackets, as depending on the value of the next quotient digit, $2^{-(j+1)}$ is added or subtracted from the current square root result. The addition of $2^{-(j+1)}$ is simply a concatenation step, but the subtraction of $2^{-(j+1)}$ requires a full-length subtracter. There is a method of overcoming this problem that is used for sequential structure square root units and is based on on-the-fly conversion where two running square root results of $s[j]$-2^{-j} and $s[j]$ are calculated in each iteration. The appropriate running square root is used in the $F[j]$ formation so it is always generated by concatenation alone and avoids the extra subtraction component. However, for a fully parallel implementation the extra hardware required to store and generate two current square root values is too great to make the implementation practical, as a basic non-restoring parallel structure would require only half the logic. Further problems arise for maximally redundant radix-4 and higher radix algorithms in terms of the $F[j]$ formation. Here the two terms in equation (2.63) overlap and cannot be generated by concatenation alone and therefore an adder component is required. A multiplier is required in the formation of the $F[j]$ term for high radix algorithms. In a digit recurrence implementation if the final remainder is negative then the quotient must be corrected by subtracting 1-ulp (unit in the last place). The un-scaled remainder must also be corrected by adding on the corrected quotient shifted one place to the left concatenated with 1-ulp. As for division the size of the quotient digit selection function increases with radix and reduced redundancy further prohibiting the implementation of high radix square root.

2.10.1.4 Summary

Due to the complexity and extra hardware associated with the $F[j]$ formation, the $S[j]$ formation, the quotient correction, the remainder correction and the quotient digit selection function it is felt that no benefit can be gained over the non-restoring algorithm by using the digit recurrence algorithm.

2.11 FPGA non-restoring square root

A very important feature of the non-restoring algorithm is that at each subsequent iteration the width of the value to add to or subtract from the shifted partial remainder increases by a single bit. For the first iteration the width of the add/sub is only 2-bits and this increases to n+2 for the n^{th} stage. The carryout of the add/sub component, which is the sign bit of the partial remainder inverted, is used to determine whether to add or subtract. The sign of the shifted partial remainder is used to determine the value to concatenate onto the end of the shifted current square root value. The current square root value is updated following the addition/subtraction. A hardware implementation of a single stage is shown in figure 2.60.

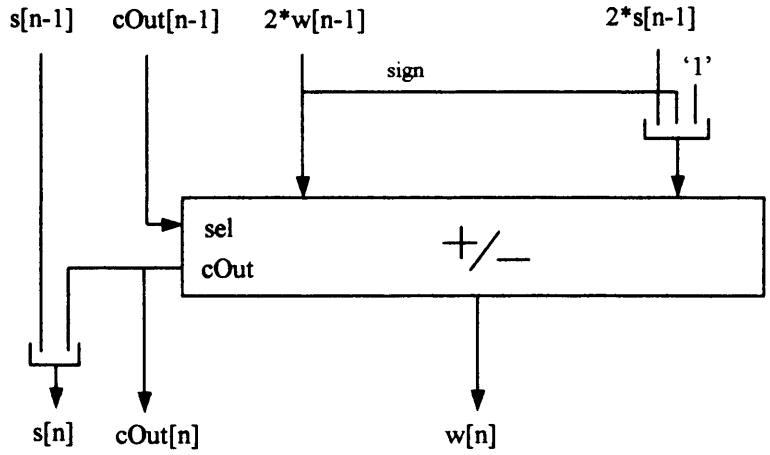


Figure 2.60. Implementation of a single radix-2 non-restoring square root stage

A hardware diagram of the complete radix-2 non-restoring divider is shown in figure 2.61. The multiplexer on the input is to shift the input value by one bit to the left. This is needed in a floating-point square root extractor if the exponent is odd.

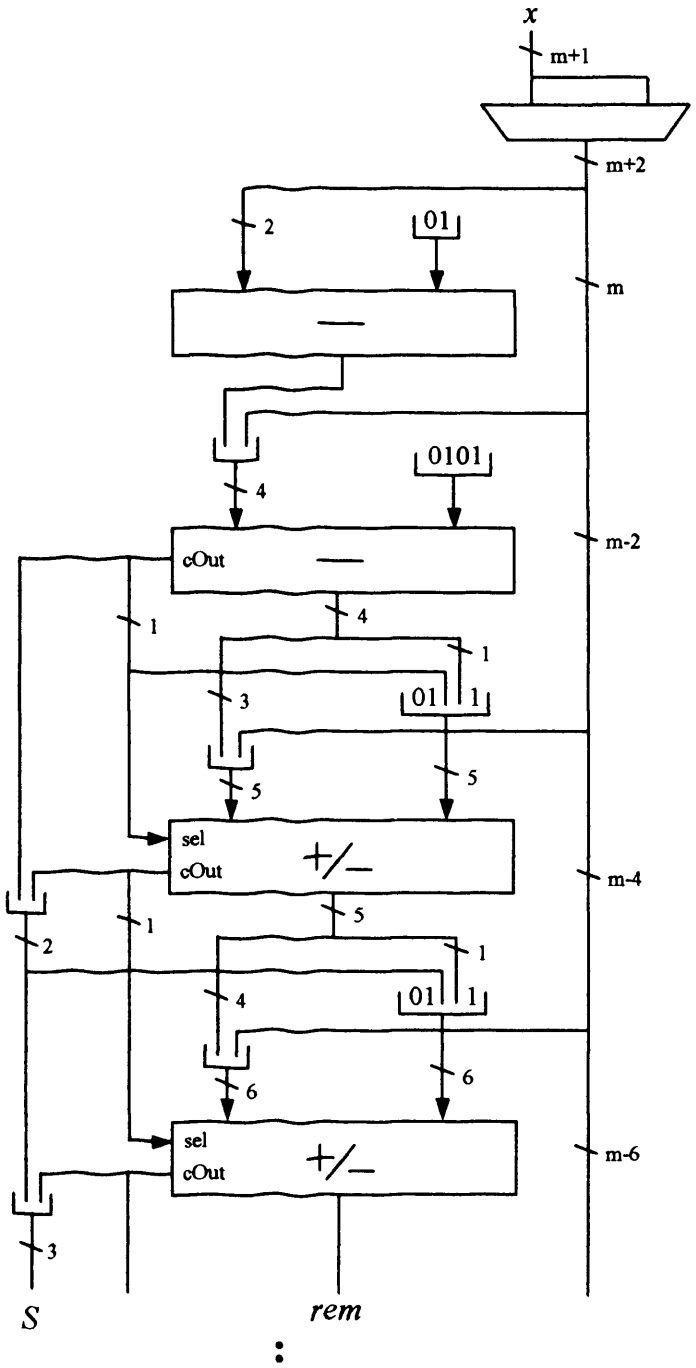


Figure 2.61. Hardware implementation of a non-restoring square root extractor

2.11.1 Other square root algorithms

Other forms of square root exist which have not been previously considered for FPGA implementation in the open literature. A convergent square root algorithm can be developed Ercegovac [61] as done for division. The Newton-Raphson formula Jeffery [58] can be used to generate the reciprocal square root and this can be multiplied by the radicand to produce the square root.

Chapter 3

Floating-point

3.1 Format

The floating-point format is a high dynamic range format that can reduce the number of overflow and underflow exceptions generated by a fixed-point system of equal word length. When compared to a fixed-point system the floating-point system trades accuracy and redundancy for dynamic range. The floating-point format consists of two main fields: an exponent and a significand. On its own the significand takes a fixed range of values and acts like a fixed-point value. The exponent part determines the position of the binary point in the significand value. Figure 3.1 illustrates the two's fields of a floating-point value.

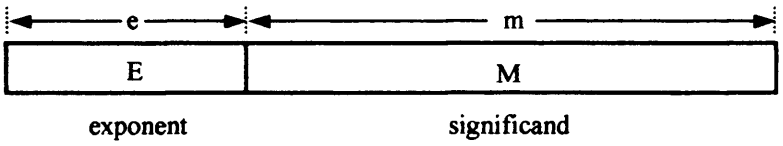


Figure 3.1. The two fields of a floating-point number

A radix-2 floating-point number X represented by the pair (E, M) takes the value of (3.1).

$$X = M.2^{E} \quad \text{(3.1)}$$

To represent signed and unsigned values the significand part must be a signed number. Two's complement or sign-magnitude representations can be used. We will consider the use of sign-magnitude as it has been adopted in the IEEE binary floating-point standard [141]. The exponent must also be a signed value so that the binary point can be left and right shifted. We will use a two's complement exponent representation.

3.2 Distribution of floating-point values

To illustrate the spread of numbers lets consider a 6-bit format where 2-bits are used to represent the exponent, 3-bits are used to represent the significand and a single bit is used for the sign. Using 2-bits for the two's complement exponent means it can take the values {-2, -1, 0, 1}. We will assume the 3-bit significand, without the influence of the exponent (i.e. when the exponent is 0), can take the values {0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5}. The 6-bit format has the spread of values as shown in figure 3.2.

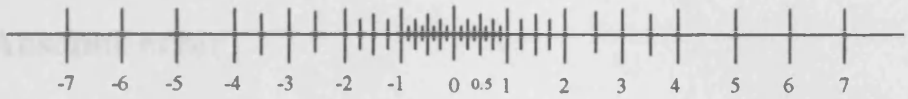


Figure 3.2. Number line for a 1-bit sign, 2-bit exponent and 3-bit significand floating-point number system

From figure 3.2 it can be seen that the distribution of floating-point numbers throughout the range is not uniform. Starting from the largest magnitude values and traversing towards zero shows that the concentration of values doubles at each power of 2 (the descriptive term often used in the literature for two consecutive powers of 2 is 'binade', although if derived from the term 'decade', which is a single power of 10, this is incorrect). The floating-point number system is a redundant system as table 3.1 shows. In table 3.1 all the different exponent-significand pairings are shown for the 6-bit number system. All the duplicate values are highlighted in grey.

	2^{-2}	2^{-1}	2^{0}	2^{1}
00.0	0.000	0.00	00.0	000.
00.1	0.001	0.01	00.1	001.
01.0	0.010	0.10	01.0	010.
01.1	0.011	0.11	01.1	011.
10.0	0.100	1.00	10.0	100.
10.1	0.101	1.01	10.1	101.
11.0	0.110	1.10	11.0	110.
11.1	0.111	1.11	11.1	111.

Table 3.1. All exponent-significand pair values for a 6-bit floating-point format

If we compare the 6-bit floating-point format to the 6-bit fixed-point format we see that the fixed-point format can represent values in the range 0.125_{10} to 3.875_{10} and can also represent zero. The floating-point format can represent values in the range

0.125_{10} to 7_{10} and can also represent zero and so has a larger dynamic range. The difference in dynamic range of the two number systems is made clearer when a larger word length format is considered. If we split a 32-bit word into a sign, an 8-bit integer and a 23-bit fraction the maximum fixed-point value is approximately 512_{10} and the minimum value just greater than zero is $2^{-23}{}_{10}$. If a 32-bit word is split into a sign bit, an 8-bit exponent and a 23-bit significand with range [0, 1) then the maximum floating-point value is approximately $2^{127}{}_{10}$ and the minimum value is $2^{-151}{}_{10}$.

3.3 Absolute error

The 6-bit floating-point and fixed-point number systems have a finite word length and clearly cannot represent every possible real number exactly. This means that converting a finite real number to one of the 6-bit formats will cause an error. Consider the conversion of the value 2.7 to the 6-bit fixed-point system. Here the nearest value is 2.75, which results in an absolute error of 0.05. For the 6-bit floating-point format the closest value is 2.5, which results in an absolute error of 0.2. Clearly the absolute error for this scenario is less for the range that can be represented by the fixed-point system demonstrating an advantage of the fixed-point system.

3.4 The IEEE floating-point standard

To unify the many floating-point standards the IEEE-754 standard for binary floating-point [141] was developed. The standard was developed from the rigorous analysis of many of the existing floating-point formats and the incorporation of their 'plus points' into an optimised floating-point format. Due to the popularity and quality of the standard it was decided that it was to be used as the basis for the floating-point format in this work. The aim of this work is to develop parameterisable exponent and significand word length floating-point units and so we will use general exponent and significand lengths for the IEEE floating-point standard discussion.

3.4.1 IEEE std-754 word format

The IEEE floating-point standard uses a three field format consisting of a sign bit, a biased exponent and a normalised significand with range [1, 2). The sign bit denotes the sign of the number with the convention that a value of '1' represents a negative

value and a sign of '0' represents a positive value. The exponent is a two's complement number that is biased by the addition of the value of 2^{e-1}-1 to the true exponent value, where e is the number of bits in the exponent. A biased exponent is sometimes described as being represented in the *excess* 2^{e-1}-1 method. Table 3.2 illustrates the true exponent and the biased exponent for a 4-bit *excess*-7 exponent representation format.

E_{true}		E_{bias}
1001	-7	0000
1010	-6	0001
1011	-5	0010
1100	-4	0011
1101	-3	0100
1110	-2	0101
1111	-1	0110
0000	0	0111
0001	1	1000
0010	2	1001
0011	3	1010
0100	4	1011
0101	5	1100
0110	6	1101
0111	7	1110
1000	8	1111

Table 3.2. 4-bit *excess*-7 exponent representation

The exponent biasing simplifies the comparison of two floating-point values as negative exponents after biasing have smaller (unsigned) values than positive exponents. Furthermore an all zero exponent and significand is used to represent zero and the biasing ensures that in comparisons zero is the smallest value. Infinity is encoded as an all ones exponent and so the biasing ensures that in comparisons infinity is the largest value. In the IEEE std-754 the significand is constrained to the range [1, 2). After an operation if the significand is not in the range [1, 2) it must be normalised to lie in this range. This is known as a normalised significand value and prevents multiple representations of the same value. The leading bit of the normalised significand is always '1' and is not explicitly required in the representation of a value (e.g. when storing values in memory), but it does need to be reintroduced when operating on the floating-point values. The leading bit is sometimes called the 'hidden bit'. Due to the leading bit of the significand always being '1' zero cannot be represented. Thus an all zero exponent represents zero when the significand is zero

and also represents a class of values called denormalized values when the significand is not zero. When an all zero exponent is detected the leading bit of the significand should be set to '0' and not '1'. A further point is that throughout this work the significand will have the meaning of being the value of the significand with the hidden bit revealed. On the other hand the mantissa will be used to describe the fractional part of the significand i.e. the part without the hidden bit.
The normal value of a floating-point triple (S, E, M), where E is the biased exponent value and M is the mantissa value is given in (3.2).

$$X = (-1)^S * 2^{E-bias} * (1+M) \quad \_\_\_(3.2)$$

3.4.2 IEEE std-754 value types

There are five different types of value that can be encoded in the IEEE format: Normalised values, Denormalized values, Zero, NaN (Not-a-Number) and Infinity.

Normalised values represent the standard set of values that can be represented by a biased exponent in the range $[1, 2^e-2]$ and any significand value in the range $[1, 2-2^{-m}]$, where e is the exponent length and m is the mantissa length.

Denormalized values are the values represented by a biased exponent of 0 and a significand in the range $[2^{-m}, 1-2^{-m}]$. The denormalized values allow gradual underflow to zero i.e. instead of the smallest non-zero number being 2^{1-bias}, the denormalized numbers allow the non-biased exponent of 0 to be used so the smallest value that can be represented is $2^{-m-bias}$. The idea is to reduce the number of underflow exceptions and to produce more accurate results.

Zero is encoded using an all zero biased exponent and an all zero significand.

NaN signifies that the result of an operation is not defined as a number. For example the subtraction of infinity from infinity has no definite answer and therefore the result is determined to be NaN. NaN is encoded as the all ones biased exponent and a mantissa that is not equal to zero.

Infinity is used to represent infinity and can be signed. The overflow of a calculation can be set to infinity. Infinity is encoded as the all ones biased exponent and a mantissa that is equal to zero.

Exception	Biased exponent	Hidden bit	Mantissa
Normalized 'Norm'	$[1, 2^e-2]$	'1'	$[0, 1-2^{-m}]$
Zero '0'	0	'0'	0
Denormalized 'Denorm'	0	'0'	$[2^{-m}, 1-2^{-m}]$
Infinity '∞'	2^e-1	'1'	0
NaN 'NaN'	2^e-1	'1'	$[2^{-m}, 1-2^{-m}]$

Table 3.3. Special value encodings for IEEE floating-point numbers with exponent width *e* and mantissa width *m*

The position of the special values on a number line is shown in figure 3.3.

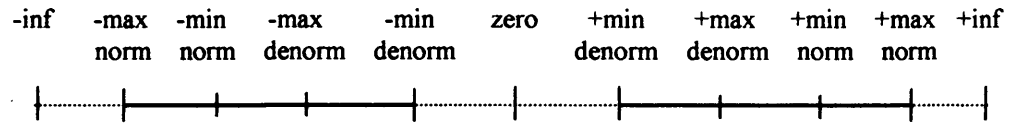


Figure 3.3. The position of the special values

3.4.3 IEEE std-754 rounding modes

There are four rounding modes defined for the IEEE floating-point standard: Round-to-nearest-even (the default rounding mode), Round-towards-zero, Round-to-plus-infinity and Round-to-minus-infinity.

RTNE (Round-to-nearest-even). This is basic rounding with a slight twist to prevent the bias that occurs. In basic rounding if we want to round a fixed-point value with integer and fractional sections to the nearest integer then any value with a fraction that is less than a half requires the fraction to be simply discarded (truncated). Any value with a fraction that is greater or equal to a half must have a value of 1 added to the integer portion and the fraction can then be discarded. The only problem with this rounding scheme is that the case where the fraction is exactly a half is always rounded up and causes a bias in the average rounding error. To prevent this bias from accumulating a 'round-to-the-nearest-even' scheme was developed. If the fraction is exactly a half and the integer that is to be rounded is even then no rounding is performed and the fraction is truncated. If the fraction is exactly a half and the integer is odd then 1 is added to the integer to round it up to be an even value. Providing the

probability of odd and even numbers is equal then the rounding scheme will be unbiased.

TRUNC (Round-towards-zero). This is another name for truncation, where any bits past the LSB are simply discarded. Truncation always reduces the magnitude of a number thus rounding it closer to zero. Truncation is the simplest form of rounding.

RTPI (Round-towards-plus-infinity). For negative values the bits past the LSB are truncated thus decreasing their magnitude and rounding them towards plus infinity. Positive values are rounded up if the bits below the LSB are greater than zero. If the bits below the LSB are zero then no rounding is required.

RTMI (Round-towards-minus-infinity). For positive values the bits past the LSB are truncated thus decreasing their magnitude and rounding them towards minus infinity. Negative values are rounded up if the bits below the LSB are greater than zero. If the bits below the LSB are zero then no rounding is required. RTPI and RTMI are commonly used in interval arithmetic schemes where the range of values that the computation can take is required. Rounding all the operations of the computation with RTPI gives the upper bound that the result can take and rounding all operations with RTMI gives the lower bound that the result can take. Thus the output range is produced.

3.4.4 IEEE std-754 exceptions

There are five types of exception: overflow, underflow, division-by-zero, invalid operation and inexact result. It is recommended by the IEEE binary floating-point standard that a flag is provided for each exception.

Overflow. The overflow exception flag is signalled whenever the exponent of an operation result exceeds the largest allowed normal exponent value. The output of an operation that overflows is determined by the sign of the intermediate result and the rounding mode as follows:
With round-to-nearest-even the output is set to infinity with the sign of the intermediate (overflowed) result.
For round-towards-zero the largest representable number is returned with the sign of the intermediate result.

For round-towards-plus-infinity if the intermediate result is negative then the largest representable number with a negative sign is returned. If the intermediate result is positive then the result is set to plus infinity.
For round-towards-minus-infinity if the intermediate result is positive then the largest representable number with a positive sign is returned. If the intermediate result is negative then the result is set to minus infinity.

Underflow. The two events of tininess and loss of accuracy must be detected before underflow is signalled. Tininess is detected if the result of a calculation is a nonzero result that lies between $\pm 2^{1-bias}$. Loss of accuracy is detected if the representation of a tiny value by a denormalized value is not exact.

Division-by-zero. The division-by-zero exception is signalled when the divisor is zero and the dividend is a finite nonzero number. The result is set to infinity and the sign is set to the XOR of the divisor and dividend signs.

Invalid operation. The invalid operation exception is signalled if any of the following invalid operations are attempted:

1. Any operation on a NaN.
2. Magnitude subtraction of infinities i.e. $\infty - \infty$
3. Multiplication of zero by infinity.
4. Division of zero by zero or infinity by infinity.
5. Square root of an operand less than zero.

The result of an invalid operation is a NaN.

Inexact result. The inexact result exception is signalled if the rounded result is not exact i.e. some accuracy was lost in the rounding process.

3.4.5 Arithmetic operators

For this work we are interested in implementing the four basic operations of addition/subtraction, multiplication, division and square root. The IEEE std-754 includes other arithmetic operations such as 'find the remainder', 'round to integer in floating-point format', 'converting binary <---> decimal' and 'compare' but we will not consider these.

3.4.6 IEEE std-754 single and double formats

There are two very popular formats called single precision and double precision that are described in the IEEE binary floating-point standard. Single precision is a 32-bit (4-bytes) long format where a single bit is used for the sign, 8-bits are used for the exponent and 23-bits are used for the mantissa. The double precision format is 64-bits (8-bytes) long where a single bit is used for the sign, 11-bits are used for the exponent and 52-bits are used for the mantissa. We will consider the implementation of formats up to double precision.

3.5 Floating-point addition/subtraction

Floating-point addition is the most complicated of the four basic arithmetic operations and there are more different implementation options for floating-point addition than the other operations. The crux of floating-point addition is that to add or subtract two floating-point values they have to have equal value exponents, which is achieved by shifting the smaller operand to the right and increasing the exponent to be the same as that of the larger value.

3.5.1 Vanilla algorithm

The following sequential steps are required when adding two floating-point numbers. The following algorithm is often called the 'vanilla algorithm' (think ice cream flavours), as it is the most basic floating-point addition method.

Assume we want to add two floating-point values X_1 and X_2 with fields (S_1, E_1, M_1) and (S_2, E_2, M_2) respectively and produce a result X_3 with fields (S_3, E_3, M_3).

Step 1.

Calculate the absolute difference of the two exponents d, where $d = |E_1 - E_2|$.

Select the smaller operand significand.

Step 2.

Shift the smaller operand significand d places to the right.

Step 3.

Add or subtract the aligned significands based on the two sign bits S_1 and S_2 and the originally required operation (addition or subtraction).

Set the result exponent to be the larger of the two input exponents.

Step 4.

Detect the number of leading zeros in the result of the significand addition/subtraction.

Step 5.

Normalize the significand by left shifting and adjust the result exponent.

Step 6.

Round the normalized significand then correct the exponent if required.

A basic block diagram of the above 'vanilla algorithm' is given in figure 3.4.

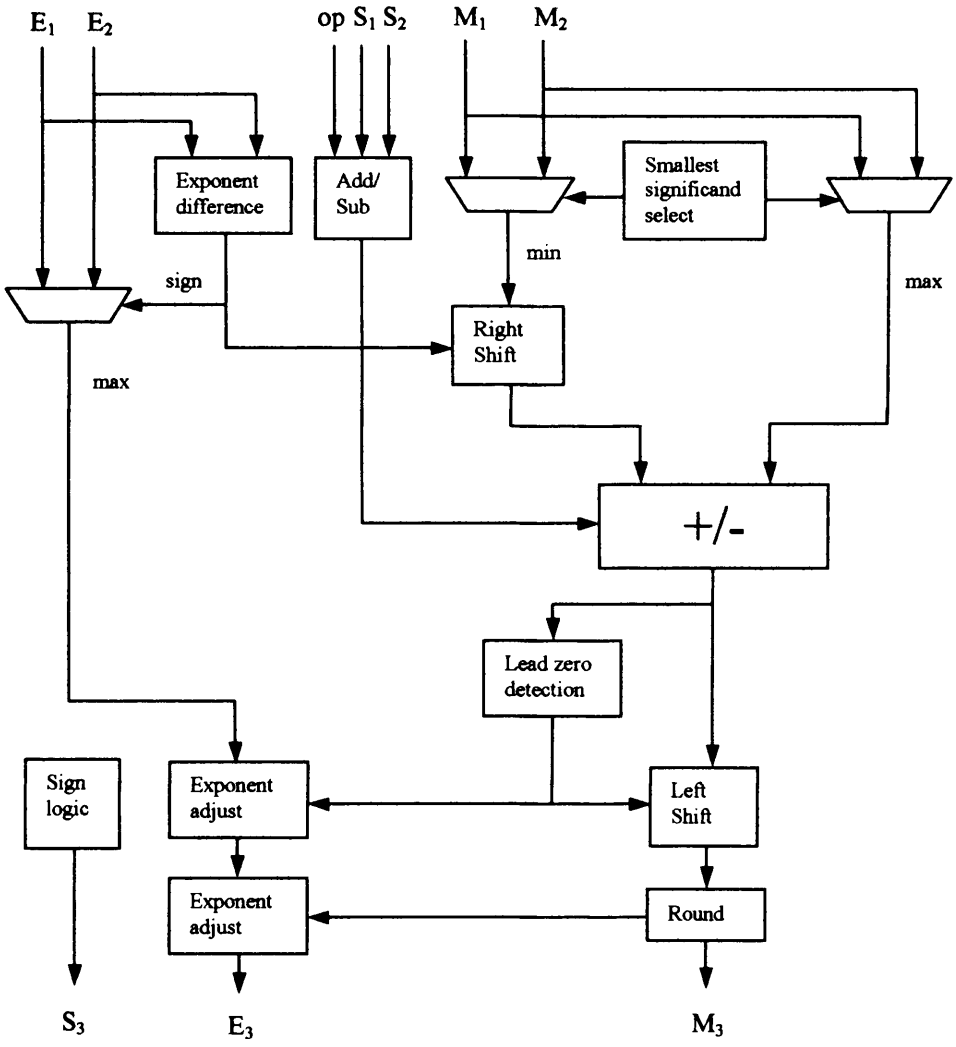


Figure 3.4. Block diagram of the 'Vanilla' floating-point addition algorithm

The block diagram of figure 3.4 does not have sufficient features to allow full IEEE floating-point addition. Extra features such as 'hidden bit' insertion, special value detection, rounding mode implementation, overflow detection and exception flags are

still required. Before these features are considered and before details of individual components are given we will discuss an improved floating-point addition method Farmwald [140] that introduces parallelism into the algorithm by analysing when full-length shifts are actually required.

3.5.2 Dual-path (near and far path) algorithm

The near and far path algorithm is commonly used in VLSI floating-point adder design to minimise the critical path of the adder. In the near and far path algorithm we have chosen, which we call the dual-path algorithm, one path (near) is developed for the scenario where a subtraction operation is performed and the absolute exponent difference is 1 or 0. The other path (far) is developed for all addition operations and also for subtraction if the absolute exponent difference is greater than 1.

3.5.2.1 Near path

The subtraction of one operand from another when their exponent difference is only 0 or 1 could cause catastrophic cancellation of the leading bits, thus many zeros would be in the leading bit positions of the resulting significand and multiple left shifts would be required to normalize the result. The detection of a 0 or 1 bit exponent difference only requires the LSB of both exponents to be inspected. An XOR of the two exponent's LSBs can be used with a result of '1' meaning align the smaller operand significand by right shifting it one place and a result of '0' meaning no shifting is necessary. A single 2 to 1 multiplexer component is used to right shift the smaller operand significand before the subtraction. The normalizing left shifting quantity depends on the incoming operands, but we can determine the worst-case situations so the shifter is designed to be the minimum length. If we assume that X_1 >= X_2 so that M_2 is subtracted from M_1 then the following cases of significant subtraction result are possible.

Extreme cases for near path subtraction

The mantissa width is 4-bits.

Maximum subtraction result value (not including zero operands)

```
 1.1111
-0.10000
 1.0111
```

Minimum subtraction result value (not including zero operands)

```
 1.0000
-0.11111
 0.00001
```

A result of zero is possible

```
 1.0101
-1.0101
 0.0000
```

Maximum width result

```
 1.1001
-0.10001
 1.00001
```

From the above worst-case examples it is clear that the resultant significand range is $[2^{-(m+1)}, 2)$ and therefore a normalization shifter is required that takes inputs that are $m+2$ bits wide and that must be able to shift left by up to $m+1$ places. It follows that an exponent adjustment of $-(m+1)$ to 0 maybe required. The near path has a reduced complexity exponent difference calculation, a reduced complexity alignment shifter and a fixed-point adder that only has to perform subtraction.

3.5.2.2 Far path

The far path requires that the smaller operand significand is right shifted by the absolute exponent difference. The following worst cases are apparent for floating-point addition for the far path algorithm.

Extreme cases for far path addition

The mantissa width is 4-bits

Maximum addition result value

```
  1.1111
+ 1.1111
 11.1110
```

Minimum addition result value (not including zero operands)

```
 1.0000
+0.00000...010000
 1.00000...010000
```

Minimum addition result value (one operand zero)

```
 1.0000
+0.0000
 1.0000
```

Minimum addition result value (both operands zero)

```
 0.0000
+0.0000
 0.0000
```

The minimum value (not including zero operands) and maximum value width depend on the right shift amount. The range of the possible results, not including the special case of two operands being zero, is [1, 4). Therefore a one bit right shift could be required to normalise the result.

For the far path subtraction we are only interested in the case where the absolute exponent difference is greater than 1. The following extreme cases for far path subtraction are apparent.

Extreme cases for far path subtraction

The mantissa width is 4-bits

Minimum subtraction result

```
 1.0000
-0.011111
 0.100001
```

Maximum subtraction result (not including zero operands)

```
 1.1111
-0.00000...010000
 1.11101...110000
```

Maximum subtraction result (with a single zero operand)

```
 1.1111
-0.0000
 1.1111
```

We do not care about the case where both operands are zero as the exponent difference is less than 2.

The maximum value (not including zero operands) and maximum value width depend on the right shift amount. The range of the possible results is (0.5, 2) therefore a one bit left shift could be required to normalise the result.

3.5.2.3 Far path right shifting

The right shift is a very important operation in the far path. At first glance it appears that the right shift would need to produce a shifted value that for the worst case could be as long as the absolute exponent difference. Thus a very long significand adder/subtracter would be required. However, by careful analysis it can be seen that the output of the shifter need only be three bits longer than the normalized significand length thus a significand adder/subtracter with a maximum length of m+4 bits is required. Furthermore the maximum shift length is constrained to be m+2. The three bits required are called the guard bit G, the round bit R and the sticky bit S. The guard and round bits are simply bits of the right shifted value that are kept unaltered. The sticky bit is the logical OR of itself and all the bits below it down to the LSB. The dot diagram for the right shifting operation is shown in figure 3.5.

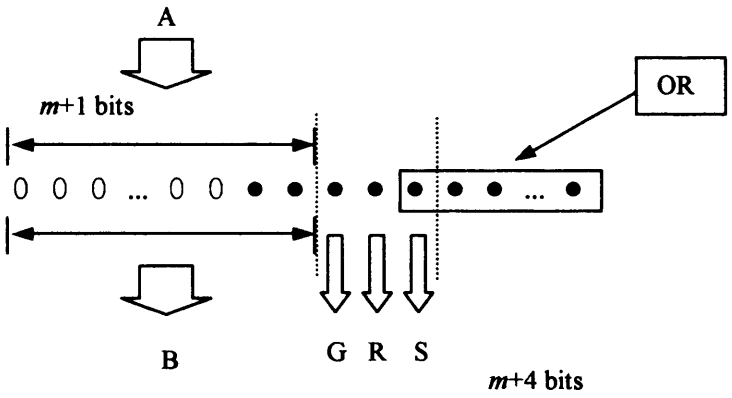


Figure 3.5. Dot diagram for right shifting

In figure 3.5 A is the input operand to shift, which is m+1 bits in length. The output value from a right shifter consists of the B operand, which is m+1 bits and the guard, round and sticky bits.

3.5.2.4 Guard, Round and Sticky bits

To correctly round a fixed point value with fraction and integer parts to the nearest integer a single bit is required called a round bit that determines whether the fraction is greater or equal to a half. The IEEE rounding mode RTNE (round-to-nearest-even)

also needs to know whether any of the bits below the round bit are set. Therefore an extra bit called the sticky bit, which is the logical OR of all the bits below the round bit, is used. These two bits are sufficient to perform the RTNE rounding mode. The round and sticky bits along with the sign bit are required to perform the other two rounding modes of round-to-plus-infinity and round-to-minus-infinity. The obvious question is "what is the point of the guard bit?" The purpose of the extra guard bit is to guard against a loss of precision if the result of the significant subtraction is in the range (0.5, 1). If the guard bit was not present and only the round and sticky bits were present then after the normalizing left shift operation only the sticky bit would be present and would not be sufficient to correctly round the result. The guard bit gives an extra bit of precision to enable correct rounding in this case. The use of the guard bit is illustrated in figure 3.6 using 4-bit mantissa values.

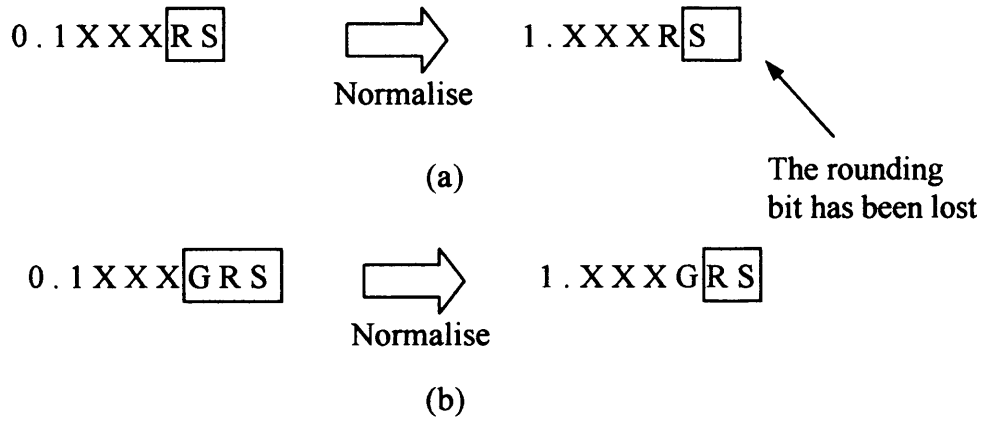


Figure 3.6. (a) Significand in the range (0.5, 1) without a guard bit. (b) Significand in the range (0.5, 1) with a guard bit.

Even if the guard bit is not needed it is still calculated and becomes the rounding bit and the original rounding bit is logically ORed with the sticky bit. This is illustrated in figure 3.7. A similar method can be used if the significand range is [2, 4).

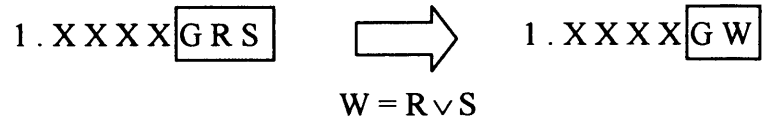


Figure 3.7. How to transform the G, R and S bits when the significand range is [1, 2)

A major asset of the guard, round and sticky bit calculation is that it can be used to shorten the significand subtraction length as well as the addition length.

A block diagram of the dual-path algorithm is shown in figure 3.8. The far and near paths are highlighted using dashed outlines.

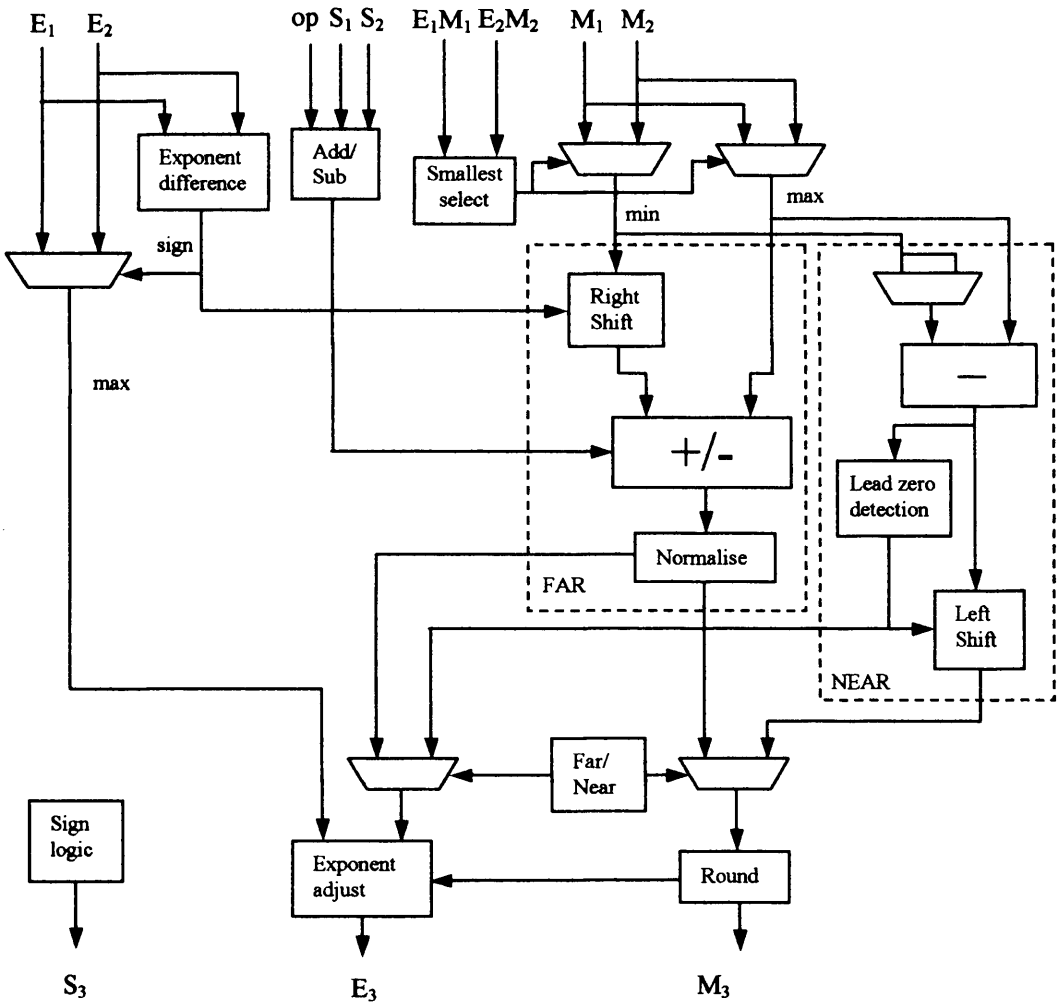


Figure 3.8. Block diagram of the dual-path floating-point addition algorithm

3.5.2.5 Dual-path summary

The near path requires an XOR operation to decide whether to shift the smaller operand significand. A multiplexer is used to perform the 1-bit shift. A subtracter is used to subtract the smaller significand from the larger one. The result needs to be normalised and could require up to m+1 left shifts. The normalised result has a range of [1, 2) and is at most m+2 bits in length so only has a single rounding bit. Often a compound adder that can calculate the *sum* and *sum*+1 *ulp* is used for the subtracter in the near path. This makes the rounding step a simple selection of the required output from the adder.

The far path uses a full absolute exponent difference to determine the right shift amount of the smaller operand's significand. The result of the right shift is m+4 bits in length due to the inclusion of the guard, round and sticky bits. The maximum right shift value is m+2. Any larger exponent difference than m+2 means only the sticky bit is calculated and due to the nature of the sticky bit calculation a shift greater than m+2 is not required. The significand addition/subtraction is m+4 bits in length. The result of the significand addition/subtraction is in the range (0.5, 4) and is up to m+5 bits in length and so there are three normalizing options: a single bit right shift, no shift and a single bit left shift.

3.5.3 Detailed component and algorithm discussion

Now the basic algorithm structure has been developed each component needs to be described in detail. We also elaborate on the implementation of some of the specific IEEE binary floating-point standard features, which impact on the elegance of the adder structure.

3.5.3.1 Special value detection

The five special IEEE std-754 values of zero, denormalized numbers, infinity, NaN and normal all need to be individually detected so there interactions can be handled separately and for the best part in parallel with the main addition algorithm. The special values, for arbitrary exponent and mantissa lengths, are encoded by the bit patterns shown in table 3.4.

Special value	Exponent	Mantissa
Zero	00...00	00...00
Denormalized	00...00	Not "00...00"
Infinity	11...11	00...00
NaN	11...11	Not "00...00"
Normal	Not "00...00" or "11...11"	XX...XX

Table 3.4. Bit patterns for the five special IEEE std-754 values

Basic wide logic gates that can be constructed from the carry-chain can be used to detect for the five special values. A possible special value detection implementation is shown in figure 3.9.

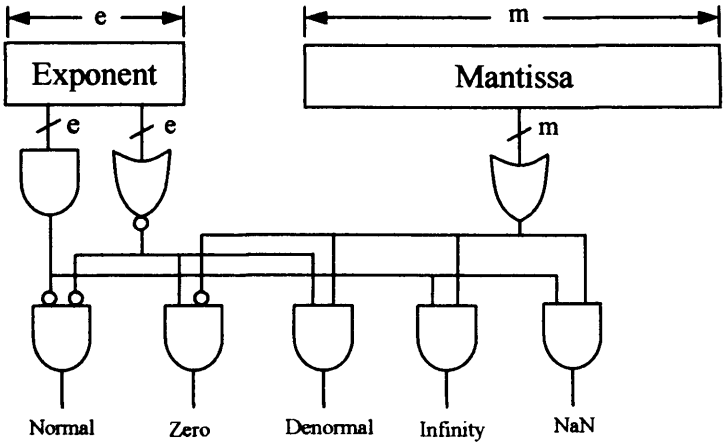


Figure 3.9. An IEEE std-754 special value detector implementation

Table 3.5 shows the output value that is generated due to the addition/subtraction of any combination of the 5 special values. From table 3.5 it can be seen that only when both inputs are normal or denormalized values does an addition need to take place to determine the output. The sign of NaN is not interpreted by the standard hence it only requires a single column. Table 3.5 is not an exhaustive list of the possible outcomes for floating-point addition. There are more options such as the exceptions generated when the result of a computation underflows or overflows, the varying outputs required for the different rounding modes and the output required when the difference of two values is zero.

A / B	+0	-0	+denorm	-denorm	+norm	-norm	+inf	-inf	NaN
+0	+0	+0	+A	-A	+A	-A	+inf	-inf	NaN
-0	+0	-0	+A	-A	+A	-A	+inf	-inf	NaN
+denorm	+B	+B	A+B	(-A)+B	A+B	(-A)+B	+inf	-inf	NaN
-denorm	-B	-B	A+(-B)	(-A)+(-B)	A+(-B)	(-A)+(-B)	+inf	-inf	NaN
+norm	+B	+B	A+B	(-A)+B	A+B	(-A)+B	+inf	-inf	NaN
-norm	-B	-B	A+(-B)	(-A)+(-B)	A+(-B)	(-A)+(-B)	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 3.5. Output values from combinations of special value inputs for floating-point addition

The implementation of denormalized arithmetic will not be considered for two reasons: firstly, previous research Digital Core Design [168], Jaenicke [167] and Fagin [156] has shown that the implementation causes the area of components to

increase dramatically and so the accuracy-area trade off is not deemed worthwhile. Secondly the floating-point system is to be compared with a logarithmic based number system. The concept of denormalized numbers will not be implemented for the logarithmic number system due to the complexity of such a system. However, denormalized inputs will be detected for and flushed to zero to allow computation to continue.

3.5.3.2 IEEE std-754 floating-point addition anomalies

When the sum of two values with opposite signs or the difference of two values with identical signs is exactly zero i.e. (-A)+B = 0, A+(-B)=0, (-A)-(-B)=0 or A-B=0 then the resulting zero shall be of positive sign in all rounding modes except round-towards-minus-infinity, in which the sign of zero shall be negative. This rule applies to zero operands as well. However, the sum of two like signed zero operands, or the difference of differing signed zero operands resolved to be the sum of like signed operands, has the sign of the input zero. This is shown in table 3.6.

A+/-B	Result
(+0) + (+0)	+0
(-0) + (-0)	-0
(-0) – (+0) = (-0) + (-0)	-0
(+0) – (-0) = (+0) + (+0)	+0

Table 3.6. The result of the sum of like signed and the difference of differing signed zero operands

The output can overflow or underflow and this can be detected by checking the value of the final exponent after the rounding stage. If overflow is detected then the result is selected depending on the rounding mode and the intermediate sign as described in the IEEE floating-point section. An exception flag is set if overflow is caused by a resulting exponent value that is too large. If underflow is detected due to the resulting exponent being smaller than the possible allowed size then the following rules are implemented. These rules differ from the IEEE std-754 because the denormalized arithmetic concept is not implemented. If the sign of the intermediate result is positive then the output is set to zero in all rounding modes except round-to-plus-infinity where the result is set to the smallest possible result. If the sign of the intermediate result is negative then the output is set to zero in all rounding modes

except round-to-minus-infinity where the result is set to the smallest possible result. If underflow is detected then an underflow exception flag is set.

3.5.3.3 Hidden bit insertion

This can be done by checking the exception detection logic for an all zero exponent, which will cause a '0' to be inserted otherwise a '1' is inserted.

3.5.3.4 Absolute exponent difference

This is a basic subtraction component. The output is two's complemented if it is negative by passing it through a separate subtracter.

3.5.3.5 Addition/subtraction operation decision

This can be calculated by using basic truth table logic. The function, which is based on the operator *op* and the two sign bits S_1 and S_2 is given in (3.3).

$$addsub = op \oplus S_1 \oplus S_2 \qquad \text{(3.3)}$$

3.5.3.6 Selection of the largest/smallest operand significand

A full-length subtraction of the two concatenated exponent and mantissa fields is performed to determine which mantissa is the smallest and the largest. The sign bit of the subtraction is used as the multiplexer selection inputs. The full-length subtraction ensures that the smaller mantissa is subtracted from the larger one when the exponent values of both operands are equal. Therefore the result of the significand subtraction is positive and would not need to be subsequently complemented. The result of the full-length subtraction can also be used to detect a result of zero when the subtraction of two identical values is performed.

3.5.3.7 Right shifter

The right shifter is constructed from levels of 4:1 and 2:1 multiplexers, which are described in Xilinx [35]. Now, depending on a select signal a multiplexer will output a particular input. If the inputs are shifted versions of some value then the select line can be seen as controlling the shift quantity. The multiplexers can be directly fed in parallel with the absolute exponent difference to control the "shift_amount". The 2:1 multiplexers are only required for an odd number of "shift_amount" bits. OR gates

are used at each level to generate the sticky bit, which constrains the size of the output from each shifter level and minimizes the hardware requirement for the next level. The structure of a 31-bit right shifter with sticky bit generation, which requires a 5-bit shift control input, is shown in figure 3.10.

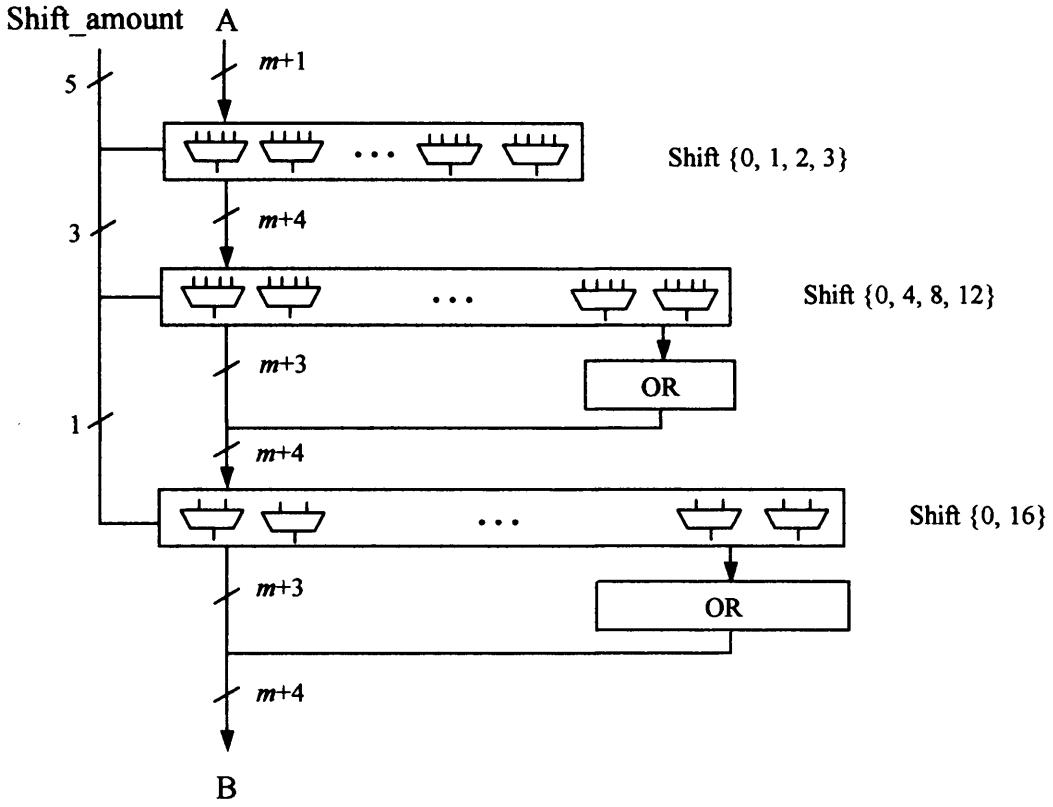


Figure 3.10. The structure of a 31-bit right shifter

3.5.3.8 Left shifter

The left shifter is a simpler, smaller and faster component compared to the right shifter. This is because the length of the operand at each level is naturally constrained to be m+2 bits in length and no sticky bit generation is required. The left shift operation causes zeros to be shifted out and these are discarded, therefore the length of each stage does not increase. The left shifter is built with layers of 4:1 and 2:1 multiplexers and requires a binary coded shift quantity input that directly controls the multiplexers.

3.5.3.9 Lead zero detector

A lead zero detector is a component that will detect the number of leading zeros in a binary string before the first '1' is encountered. The quantity of leading zero bits is

large converters as required for double precision are fast. In figure 3.12 an example of the conversion of a 15-bit one-hot value 'H' to a 4-bit binary value 'B' is given.

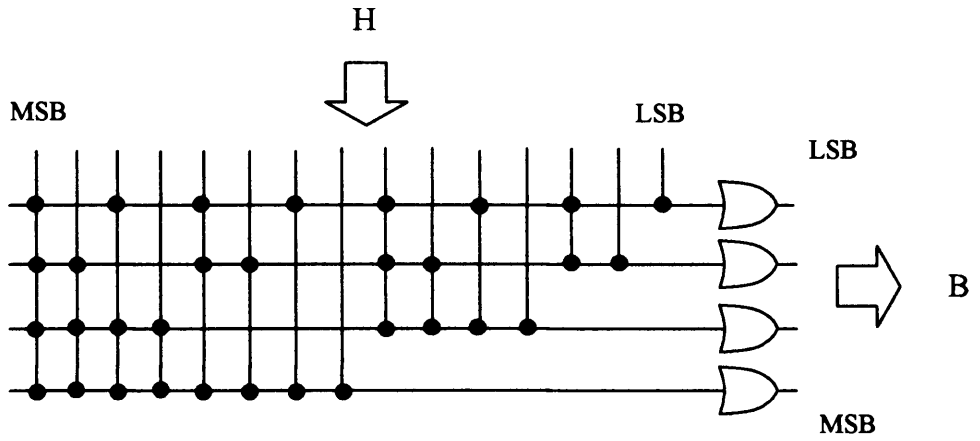
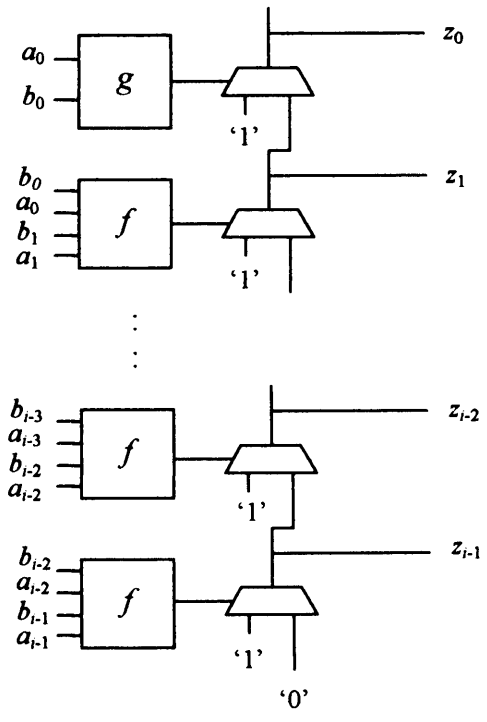


Figure 3.12. 15-bit one-hot to 4-bit binary conversion

Method 2

A commonly used circuit in VLSI floating-point adder design is the LZA (leading zero anticipator) Suzuki [144]. This circuit operates in parallel with the significand adder in order to reduce the delay of the sequential leading zero detection operation.



$$g = a_i \oplus b_i$$

$$f = a_i.b_i + \overline{a_i}.\overline{b_i} + \overline{a_{i-1}}.b_{i-1}$$

Figure 3.13. LZA circuit structure

The lead zero detector circuit operates MSB to LSB and so has to wait for the result of the full-length significand subtraction A-B. The LZA circuit operates in an MSB to LSB fashion and produces the one-hot value in parallel with the significand subtraction using the values A and B. However, there is a slight problem with the result of the LZA circuit in that it could be in error by one place and in such circumstances a correction is required. This correction can be done in parallel for each bit by analysing the significant subtraction and LZA results, which are available at the same time. The LZA circuit has the structure shown in figure 3.13. Assuming the LZA circuit result is 'Z' and the significand subtracter result is 'D' the parallel compare and correct function H is given in figure 3.14.

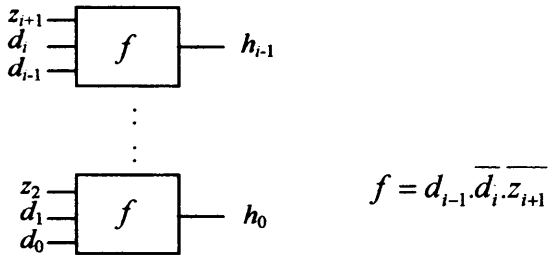


Figure 3.14. Parallel compare and correct function

A critical path comparison of methods 1 and 2 is given in figure 3.15. From figure 3.15 it can be seen that method 2 requires just as much logic as method 1 and has a shorter critical path if the large significand A and smaller pre-shifted significand B are provided simultaneously.

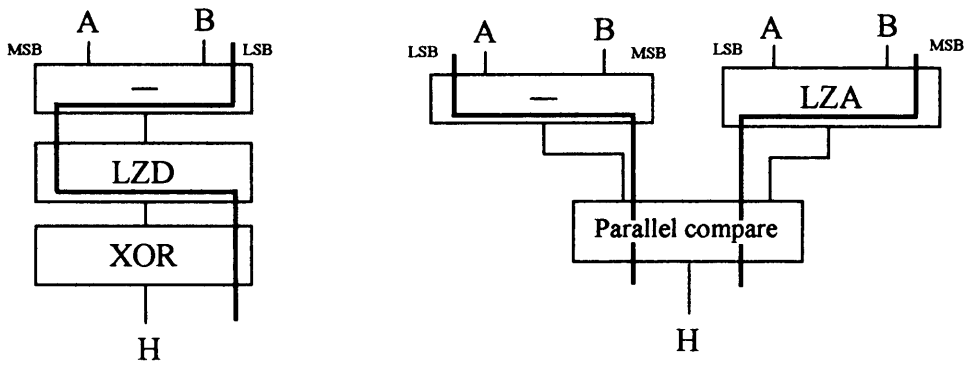


Figure 3.15. A critical path comparison of two leading zero detector methods

Despite the benefits of the LZA circuit other logic techniques can be taken advantage of that make method 1 more suitable. The near path requires a single normalising

right shift and for method 1 this can be blended into the significand subtraction A-B by using a subMux component described in the FPGA addition section. For method 2 the right shift cannot be blended into the LZA component and so a separate multiplexer (shifter) is required, which increases the logic by an extra LUT column component and also increases the critical path.

Method 3

This method is different from methods 1 and 2 in that it is sequential in nature and does not use a leading zero detector as such and also does not require a one-hot to binary converter.

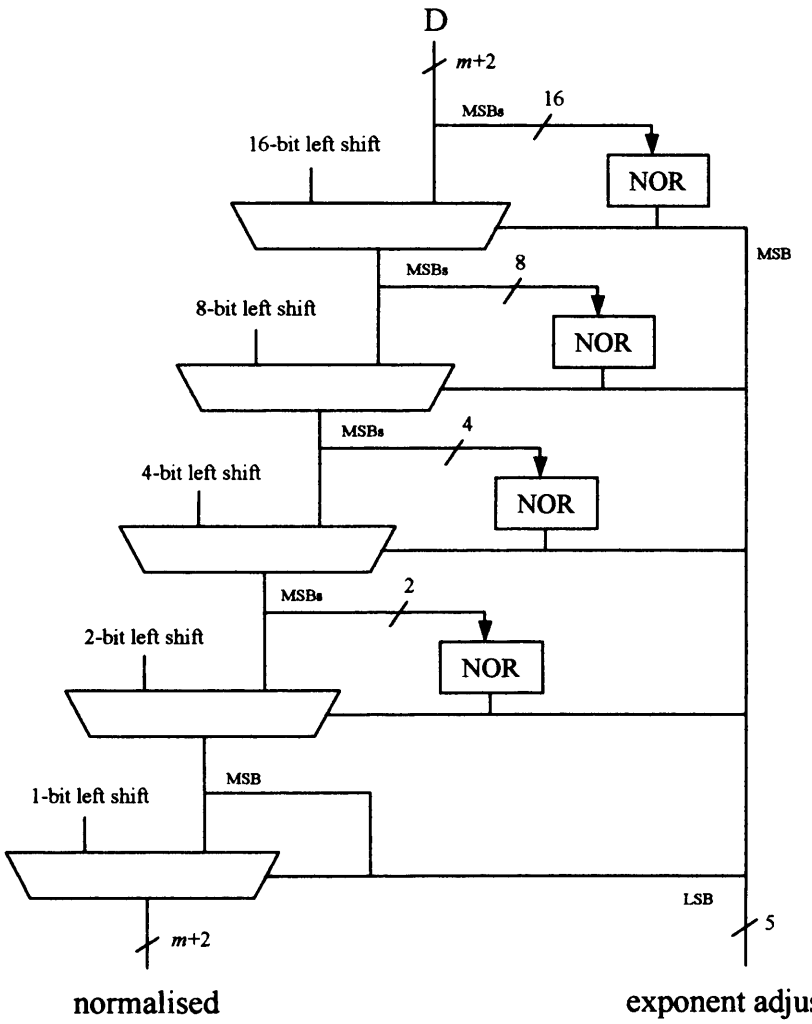


Figure 3.16. 31-bit left shifter incorporating the leading zero detection

A left shifter is used that is constructed solely of 2:1 multiplexers and is blended with the lead zero detection and operates in a very simple way: At the first 2:1 multiplexer

shift level the input operand is left shifted by N or 0 bits. An N-bit NOR gate is used to detect for N leading zero bits and the output of the NOR gate controls the shifting multiplexer. At the next level the input, which is the output of the previous level, is shifted by N/2 or 0 bits and an N/2 bit NOR gate is required to decide whether to shift or not. This is repeated for all levels. Typically the shifter is binary encoded so the initial N value is a power of 2 and ceil($\log_2$(N)) levels are required. The input select values that are passed to each multiplexer are concatenated and are used as the exponent adjustment value. A 31-bit left shifter based on method 3 is shown in figure 3.16.

3.5.3.9.1 Comparison

To determine the most suitable lead zero detector method methods 1 and 3 were implemented and their area and delay statistics for varying mantissa widths were calculated. Both methods take a significand difference as input and output the normalized value and the binary coded left shift quantity. Figure 3.17 compares the area of the two implementations and it is clear that method 3 is superior. Figure 3.18 compares the delay of the two methods. It can be seen that the methods have similar delays above 32-bit mantissa values but method 3 has a superior delay below this point. In conclusion method 3 is superior and will be used in the floating-point adder implementation.

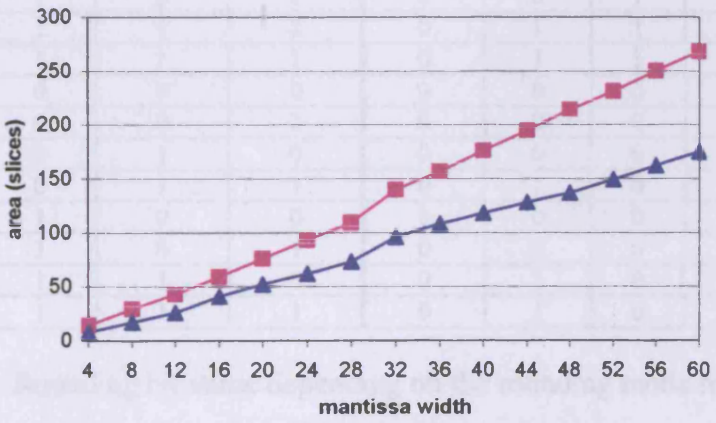


Figure 3.17. An area comparison of the lead zero detection methods 1 (■) and 3 (▲)

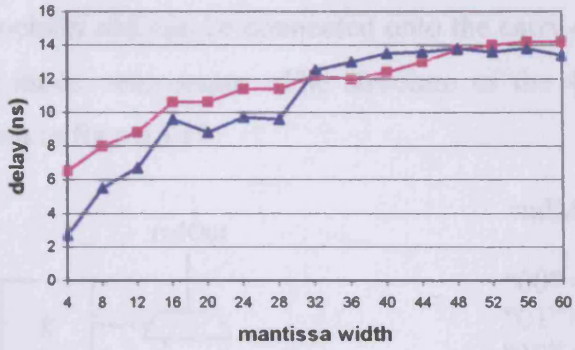


Figure 3.18. A delay comparison of the lead zero detection methods 1 (■) and 3 (▲)

3.5.3.10 Rounding and exponent correction

Five things need to be known before a value can be correctly rounded according to either of the four IEEE rounding modes: The intermediate sign, the least significant bit of the value to round, a round bit, a sticky bit and the mode of rounding required. Once these five values are known the bit to add to the value to correctly round it can be determined by using the logic in table 3.7.

sign	LSB, L	Round, R	Sticky, S	TRUNC	RTNE	RTPI	RTMI
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1
1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	1	0	1
1	1	1	1	0	1	0	1

Table 3.7. Rounding bit value depending on the rounding mode required

An adder is needed to round the significand value and the round bit can be injected in as a carry-in bit to the adder. To minimize the delay of the rounding operation a 4-IEEE rounding mode component that utilises the carry-chain so the output can be directly fed into the rounding adder carry-in input without requiring any routing logic

or delay has been developed. Furthermore the sticky bit generation is a wide OR gate that uses the carry-chain and can be connected onto the carry-chain as an input to the 4 IEEE rounding mode component. The structure of the 4-IEEE rounding mode component is shown in figure 3.19.

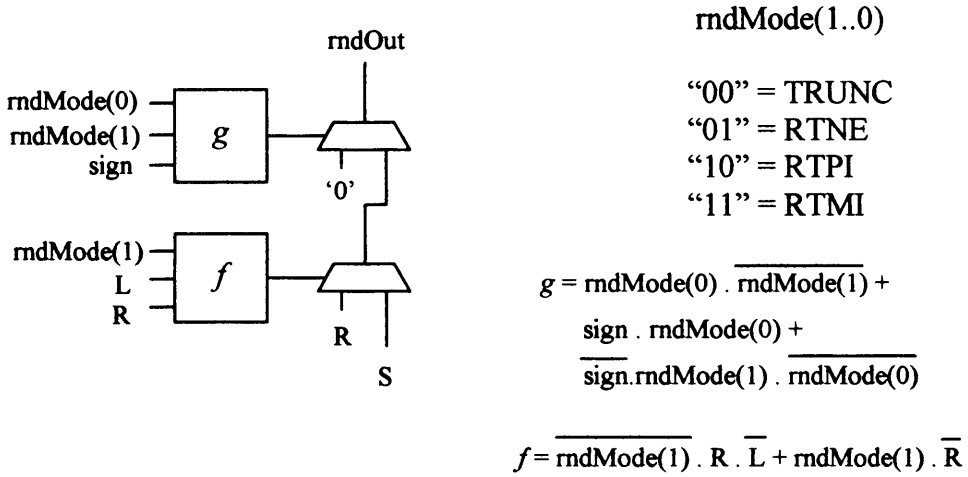


Figure 3.19. 4-IEEE rounding mode component

If the rounding operation overflows then the exponent must be increased and the significand must be right-shifted to be normalized. The carry-out signal of the rounding adder indicates whether the rounding operation has overflowed. This signal, which comes from the carry-chain, can be injected into the carry-in of an adder component that increases the exponent value. The result need not be right-shifted to correct the leading-bit as it is to be discarded. Figure 3.20 shows the general structure of the floating-point addition rounding, which utilises one carry-chain and so is fast.

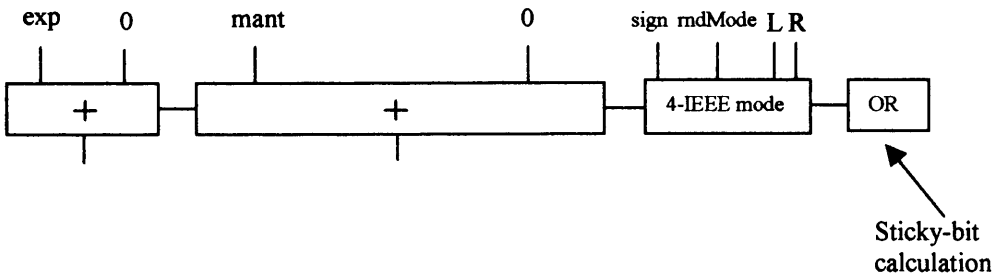


Figure 3.20. Floating-point addition rounding

3.5.3.11 Sign logic

Not considering special cases the basic sign logic is based on the signs of both operands, the operator (addition/subtraction) and the sign of the difference of the input

concatenated exponents and mantissas (i.e. the sign of the difference used to determine the largest and smallest operand significands). Assuming S_D is the sign of the difference (A-B), *op* is the operator, S_A is the sign of the A operand and S_B is the sign of the B operand then the equation for the basic output sign is given as (3.4).

$$sign = S_A.\overline{S_D} + op.S_D.\overline{S_B} + \overline{op}.S_B.S_D \quad \_\_\_(3.4)$$

3.5.3.12 Far/Near path detection

If the absolute exponent difference is less than 2 and the effective operation is subtract then the near path is selected otherwise the far path is selected. An OR gate can be used to check whether the absolute exponent difference is greater or equal to 2 and the effective operation is determined by (3.3).

3.5.3.13 Overflow/underflow detection

Overflow/underflow detection is done by checking the value of the exponent after rounding. To enable overflow and underflow detection an extra exponent bit is required to detect the sign of the exponent value. With certain custom exponent and mantissa formats is it possible to require more than a single sign bit to determine the underflow situation as the left shift normalising could be so great as to cause a single extra sign bit to be '0'. However, such design considerations are only applicable when the mantissa width is greater than the maximum negative value that can be created with an e+1 bit two's complement value. The maximum value as the result of the addition of two floating-point numbers has an all ones exponent and extra sign bit(s) of zero. This is the only pattern of exponent that causes overflow and can easily be detected with a wide AND gate with inverted MSB input. Underflow is detected by an all zeros exponent, which can be detected by a NOR gate, or a most significant extra sign bit of '1'.

3.5.3.14 Complete floating-point adder diagram

Using all the described points a diagram of the complete floating-point adder design for FPGA is shown in figure 3.21. The diagram has three unexplained functions named *f*, *g* and *h*. The *f* function checks if the output of the far path is in the range [0.5, 1), in which case the exponent needs to be decremented by 1. The *g* function checks the range of the far path result and whether a near or far path result is to be

selected and in turn controls the 4:1 normalising and path selecting multiplexer. The *h* function checks whether the far path is selected and whether the exponent needs to be incremented due to the significant addition being in the range [2, 4).

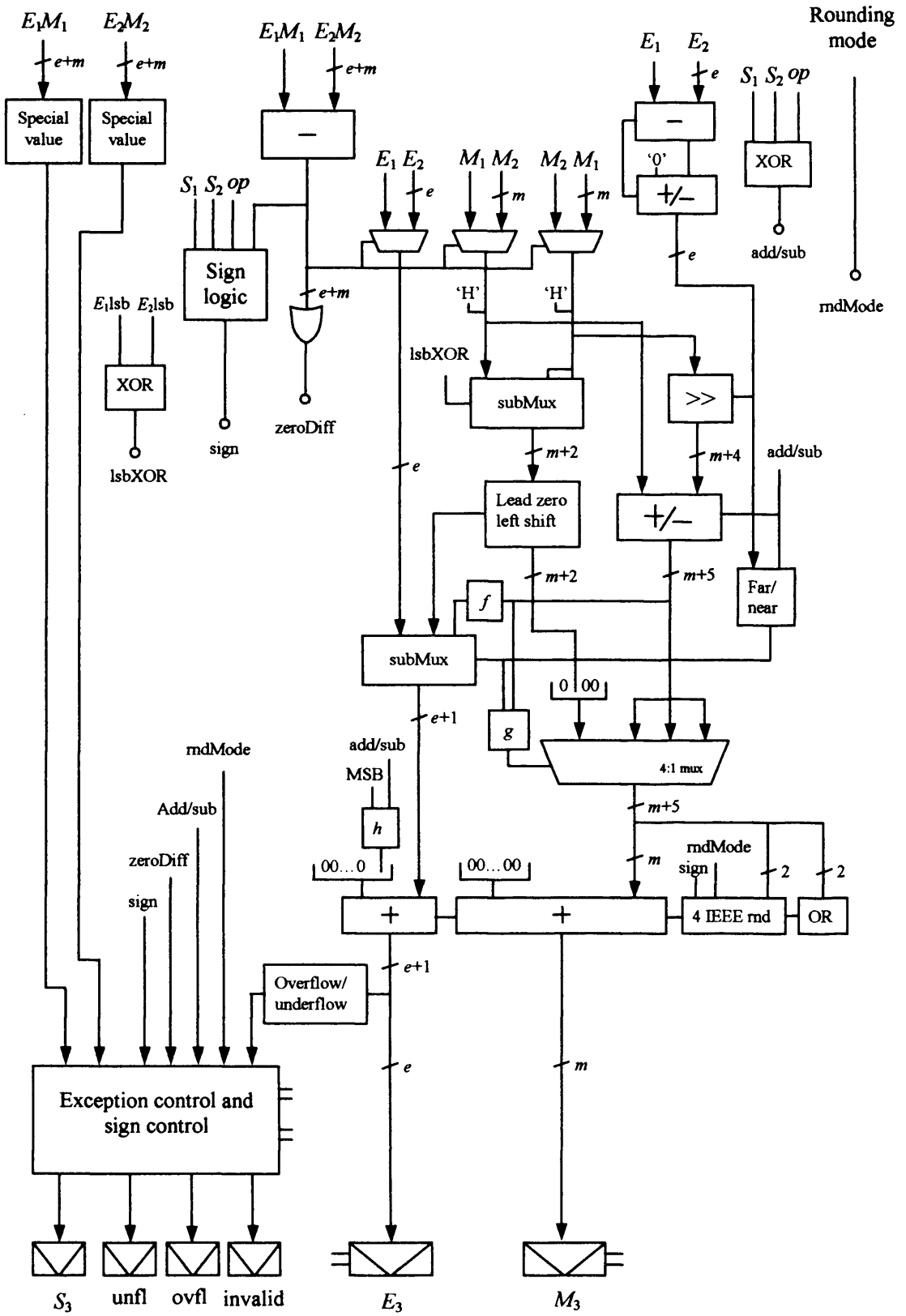


Figure 3.21. Complete floating-point adder diagram

3.6 Floating-point multiplication

Basic floating-point multiplication involves taking the XOR of the sign bits, adding the exponents and multiplying the significand values. For IEEE std-754 multiplication the above operations are complicated due to exception detection, exponent biasing, result normalisation, rounding and overflow/underflow detection. In this section we will discuss the implementation of a floating-point multiplication algorithm.

3.6.1 Special values

The five IEEE-754 special values can be detected using a similar component as was used for floating-point addition. Table 3.8 shows the output values deduced from different combinations of special input values.

B \ A	+0	-0	+denorm	-denorm	+norm	-norm	+inf	-inf	NaN
+0	0	0	0	0	0	0	NaN	NaN	NaN
-0	0	0	0	0	0	0	NaN	NaN	NaN
+denorm	0	0	A*B	A*B	A*B	A*B	inf	inf	NaN
-denorm	0	0	A*B	A*B	A*B	A*B	inf	inf	NaN
+norm	0	0	A*B	A*B	A*B	A*B	inf	inf	NaN
-norm	0	0	A*B	A*B	A*B	A*B	inf	inf	NaN
+inf	NaN	NaN	inf	inf	inf	inf	inf	inf	NaN
-inf	NaN	NaN	inf	inf	inf	inf	inf	inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 3.8. Output values from combinations of special value inputs for floating-point multiplication

From table 3.8 it can be seen that only the multiplication of denormalized and normal numbers need be performed as the results of the other special values can be determined from the table. We will not consider the implementation of denormalized numbers but will detect them and flush them to zero.

3.6.2 Exponent bias

The exponent is a biased value and adding two biased exponents will double the biasing value so one needs to be subtracted. The bias subtraction is done after the exponents are added and is blended into the rounding/normalising adder.

3.6.3 Significand multiplication

The significand multiplication is performed using a fixed-point multiplier. The optimised multiplier developed in section 2.7.7.3.4 is used in the floating-point design. The result of the developed fixed-point multiplier is double length and this needs to be rounded down to a single length result i.e. the significand length is m+1 and the result of the multiplication is of length 2*m+2. The round and sticky bit concept can be adopted and implemented to correctly round the significand multiplication. There is one slight complication in that the range of the significand multiplication is [1, 4) so a normalising right shift could be required.

3.6.4 Normalising and rounding

The value of the significand could need normalising by shifting it one bit to the right, which can be done with a multiplexer component. If the rounding adder is configured as an addMux component then the normalising shift can be blended into the rounding component thus saving a whole LUT column component. However, now two round bits (one for the shifted value and one for the non-shifted value) need to be created and fed to the rounding adder and so the carry-chain input as was used for the floating-point adder cannot be used. The rounding adder can also be used to subtract the bias from the exponent sum and to adjust the exponent if a post-rounding normalising right shift is required. For a value that needs to be normalised, the exponent is reduced by a value that is one less than the bias. Assuming that the result of the significand multiplication is a 2*m+2 bit value P, where P is given as (3.5), then the rounding adder structure with corresponding connections is shown in figure 3.22.

$$P = p_{2m+1}p_{2m}\cdots p_1p_0 \quad \text{(3.5)}$$

The LSB of the addMux result is discarded as it is used only to aid the rounding process, hence the result is e+m+1 bits wide whereas the inputs are e+m+2.

3.6.5 Overflow and underflow detection

The value of the exponent after rounding needs to be checked to determine whether overflow or underflow has occurred. If an extra bit of the exponent is kept so it is e+1 bits in length then underflow can be detected by an all zero exponent or if the two

most significant bits are '11'. Overflow is detected by an all ones exponent and a most significant bit of '0', or if the two most significant bits are '10'.

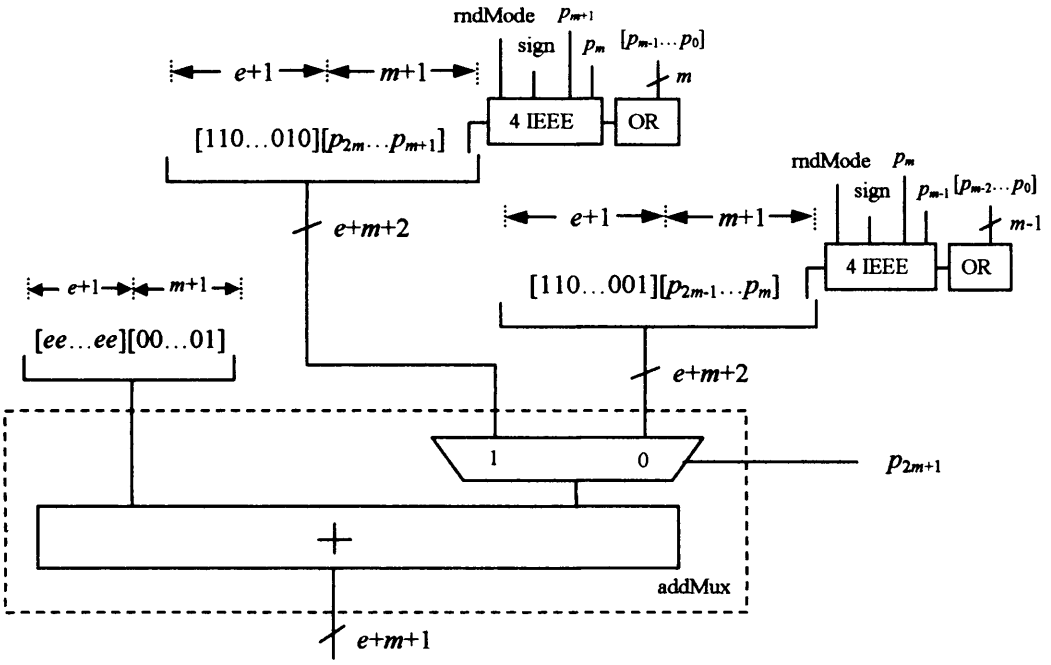


Figure 3.22. Rounding, normalisation and exponent correction for floating-point multiplication

3.6.6 Sign logic

The output sign is always the logical XOR of the two input signs even for infinity and zero.

3.6.7 Complete floating-point multiplier diagram

By combining all of the above points a complete floating-point multiplier design can be created and a diagram of the FPGA implementation is shown in figure 3.23.

3.7 Floating-point division

The basic floating-point division algorithm has three main steps: the exponent value is created by subtracting the divisor exponent from the dividend exponent; the significand is calculated by dividing the dividend significand by the divisor significand using a fixed-point divider; the sign is created by calculating the logical XOR of the two input operand sign bits. Extra consideration is needed for the five

IEEE special values and for the exceptions. In this section we will discuss the implementation of a floating-point division algorithm.

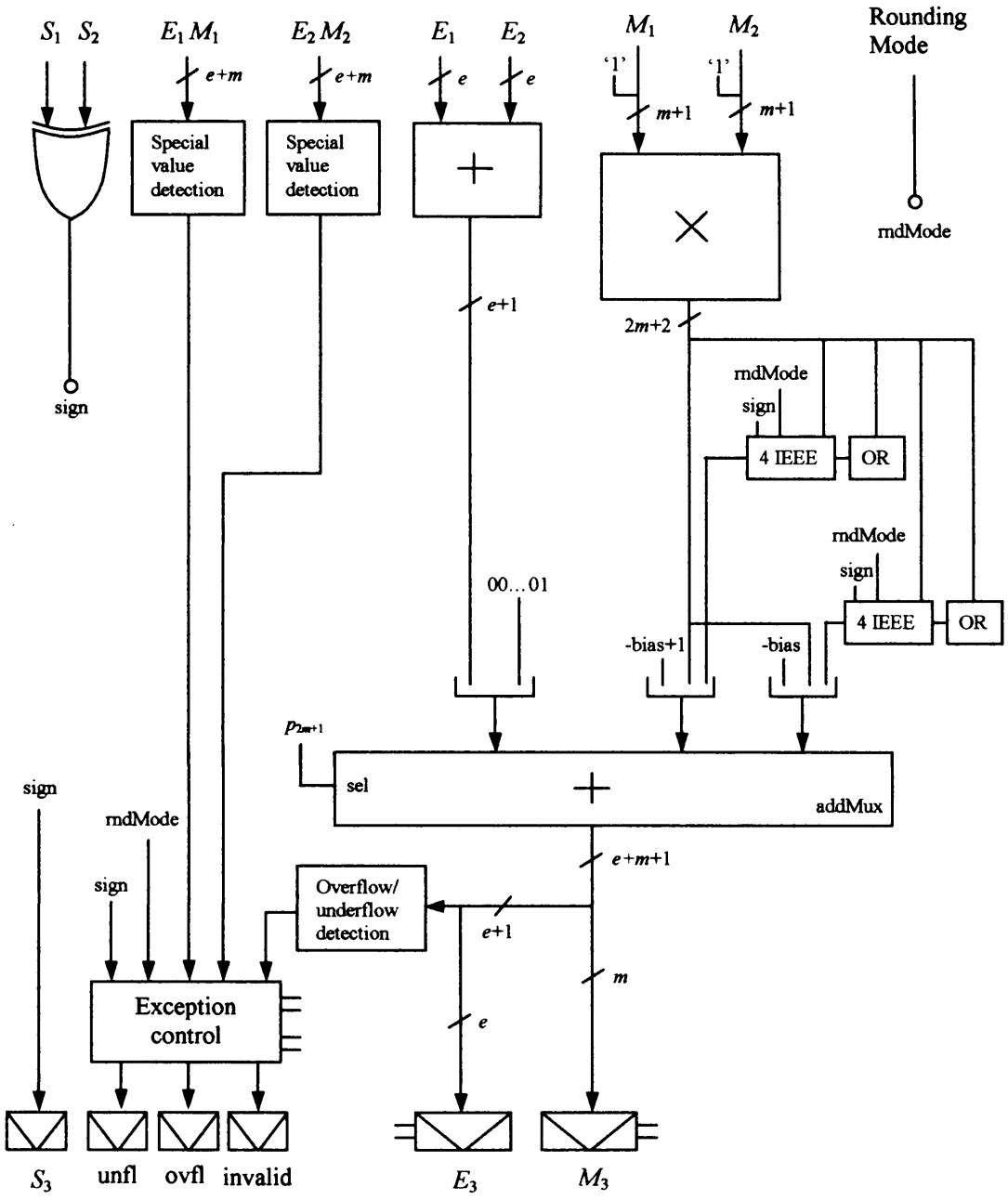


Figure 3.23. The complete floating-point multiplier design

3.7.1 Special values

The five IEEE std-754 special values can be detected using a similar component as was used for floating-point addition. Table 3.9 shows the output values deduced from different combinations of special values. A is the dividend and B is the divisor.

B \ A	+0	-0	+denorm	-denorm	+norm	-norm	+inf	-inf	NaN
+0	NaN	NaN	inf	inf	inf	inf	inf	inf	NaN
-0	NaN	NaN	inf	inf	inf	inf	inf	inf	NaN
+denorm	0	0	A/B	A/B	A/B	A/B	inf	inf	NaN
-denorm	0	0	A/B	A/B	A/B	A/B	inf	inf	NaN
+norm	0	0	A/B	A/B	A/B	A/B	inf	inf	NaN
-norm	0	0	A/B	A/B	A/B	A/B	inf	inf	NaN
+inf	0	0	0	0	0	0	NaN	NaN	NaN
-inf	0	0	0	0	0	0	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 3.9. Output values deduced from combinations of special value inputs for floating-point division

From table 3.9 it can be seen that only the division of denormalized and normal numbers need be performed, as the results of other special values can be determined from the table. We will not consider the implementation of denormalized values but will detect them and flush them to zero so computation can continue.

3.7.2 Exponent bias

The exponent values are biased so finding the difference of two exponent values will cause the bias to be cancelled. Therefore the exponent bias must be added onto the result of the exponent difference. The bias correction is done in conjunction with the exponent adjusting, which is required if the divisor significand is larger than the dividend significand.

3.7.3 Significand division

The significand divider requires a fixed-point component. The fractional dividers developed in the fixed-point division section can be used for the significand division. The floating-point significands have a range [1, 2) and so are not strictly fractional but they can be scaled by shifting them to the right so they then have a range of [0.5, 1). The scaling is a reinterpretation of the binary point and does not require any logic to implement. The range of the resultant quotient if both the divisor and dividend have a possible range of [0.5, 1) is [0.5, 2) and therefore the result could need scaling to ensure it is in the desired range of [1, 2). To overcome the scaling requirement the dividend is left shifted by one bit if the divisor is greater than the dividend, which

ensures the output range is [1, 2). If a scaling operation takes place then the exponent must be compensated by decreasing it by one. As mentioned this is blended into the exponent bias correction stage where depending on the scaling operation a value of *bias* or *bias*-1 is added to the exponent difference. The output of the significand divider, assuming a radix-4 SRT is used, is a quotient and a remainder that are not corrected i.e. the remainder could be negative and thus the remainder and quotient would need correcting (see Appendix A.1). The correction is not performed as it can be blended into the rounding stage using only a small amount of extra logic.

3.7.4 Rounding and quotient correction

The quotient is calculated with an extra bit of precision, which acts as a rounding decision bit. The sticky bit is calculated by taking the OR of the remainder value so the sticky bit is set if the remainder is not zero. The sticky bit, round bit and L bit, which is the bit juxtaposed to the round bit (figure 3.24), are passed to the 4 IEEE rounding mode component as used in the floating-point adder to determine the rounding bit to add to the quotient. However there is a problem in that the quotient has not been corrected and so the LSBs of the quotient may have the wrong value and rounding will generate the wrong answer. To overcome this problem the remainder sign and the two least significant bits of the quotient can be used to calculate the true value of the two least significant bits of the quotient as shown in table 3.10. These two bits can then be passed to the 4 IEEE rounding mode component and thus the correct rounding bit to add to the corrected quotient can be calculated.

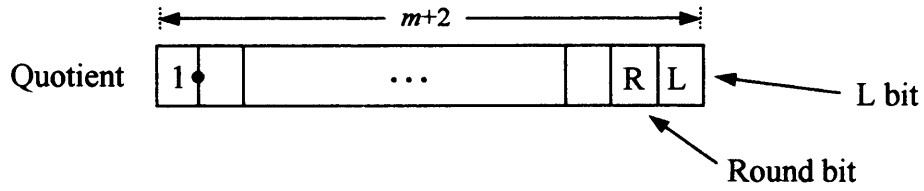


Figure 3.24. The round and L bit of the quotient value

The value of the round bit to add to the quotient is now known as well as whether to correct the quotient by subtracting one LSB. There is a conflict here because to generate the correctly rounded quotient there are 4 options: either add nothing to the quotient and truncate it to $m+1$ bits, or add a bit at the round bit position and truncate the quotient to $m+1$ bits, or subtract a bit at the L bit position and truncate the quotient

to m+1 bits, or simultaneously add a bit at the round bit position and subtract a bit at the L bit position (LSB) and then truncate the quotient to m+1 bits. The 4 options are summarised in table 3.11 together with the resolved correcting bits and whether they should be added or subtracted.

	Remainder sign (*RS*)	L bit (*L*)	Round bit (*R*)	New L bit (*NL*)	New round bit (*NR*)
No change to quotient bits.	0	0	0	0	0
	0	0	1	0	1
	0	1	0	1	0
	0	1	1	1	1
Subtract 1 from the LSB (R) position.	1	0	0	1	1
	1	0	1	0	0
	1	1	0	0	1
	1	1	1	1	0

$$NL = \overline{R}\,\overline{L}.RS + R.L + L.\overline{RS} \qquad NR = R.\overline{RS} + \overline{R}.RS$$

Table 3.10. The corrected values of the quotient's two LSBs

Rounding bit (*RB*)	Remainder sign (*RS*)	Correcting bits (*CB*)	Add/sub (*AS*)
0	0	00	-
+1	0	10	+
0	-1	01	-
+1	-1	01	+

$$CB(0) = RS \quad CB(1) = RB.RS \quad AS = RB$$

Table 3.11. The resolved correction bits to add or subtract from the quotient

The correcting bits can be concatenated with an all zero value that is as long as the quotient and then can be added or subtracted from the quotient value to create the correctly rounded quotient. An add/sub component is needed to round the quotient to create the result significand. The result of the rounding operation cannot overflow and hence a subsequent correction of the exponent is not required.

3.7.5 Overflow and underflow detection

The value of the exponent after the bias correction needs to be checked to determine whether overflow or underflow has occurred. If an extra bit of the exponent is kept so it is e+1 bits in length then underflow can be detected by an all zero exponent or if the two most significant bits are '11'. Overflow is detected by an all ones exponent and a most significant bit of '0', or if the two most significant bits are '10'.

3.7.6 Sign logic

The output sign is always the logical XOR of the two input signs.

3.7.7 Complete floating-point divider diagram

By combining all of the above points a complete floating-point divider design can be created and a diagram of the FPGA implementation is shown in figure 3.25.

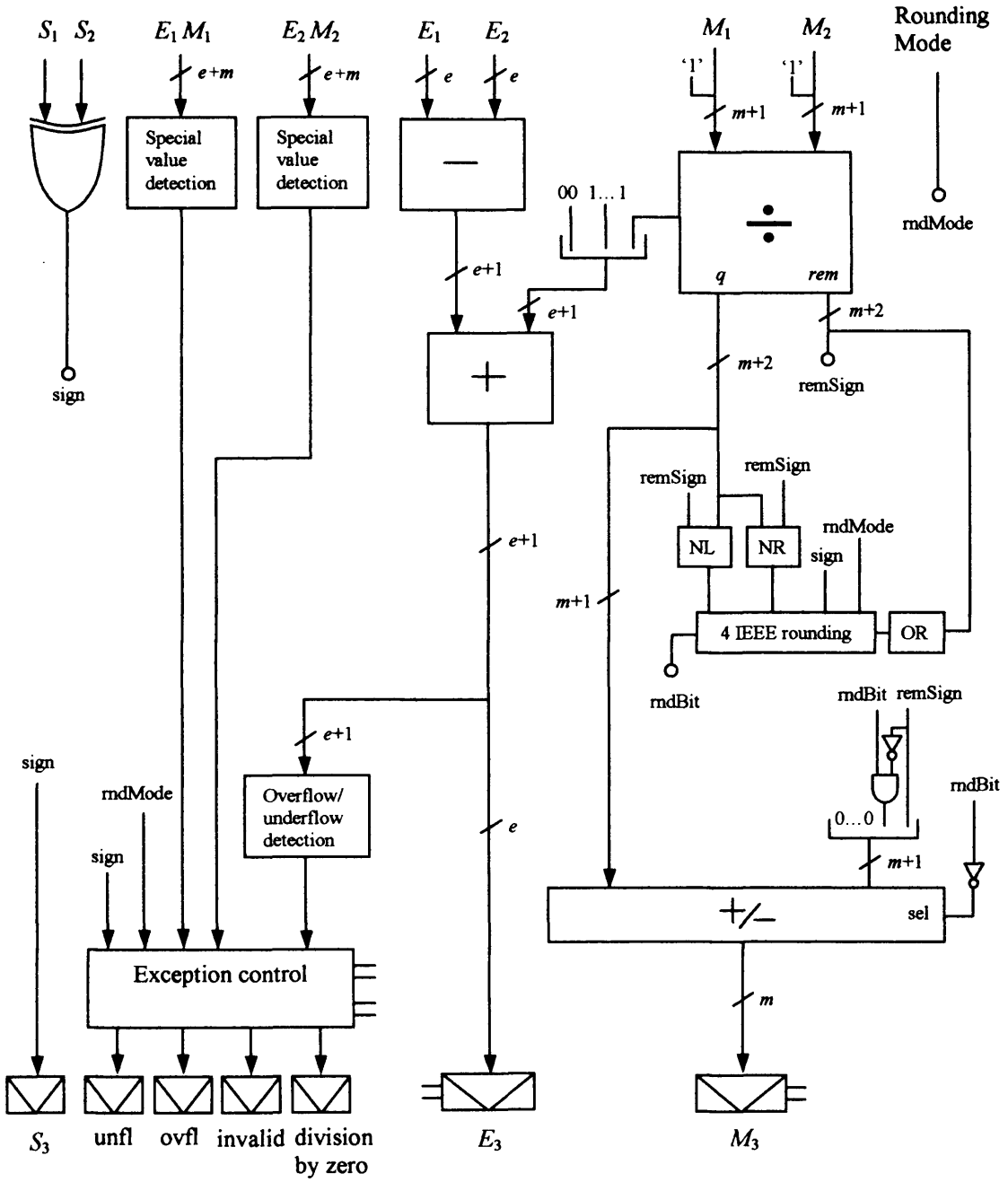


Figure 3.25. The complete floating-point divider diagram

3.8 Floating-point square root

Floating-point square root is the only floating-point monadic function we will consider in detail. The basic square root algorithm does not require anything being done to the sign bit, the exponent value is halved and the fixed-point square root of the significand needs to be found. Normalising, exponent biasing and rounding complicate the basic algorithm and the effects and implementation issues of these anomalies will be discussed in this section.

3.8.1 Special values

Square root is a monadic function so the table (table 3.12) that describes the output when a special value input is detected is simplified.

Input, A	+0	-0	+denorm	-denorm	+norm	-norm	+inf	-inf	NaN
Output	+0	-0	+sqrt(A)	NaN	+sqrt(A)	NaN	+inf	NaN	NaN

Table 3.12. The output value depending on the input value

From table 3.12 it can be seen that any negative input value apart from zero causes a NaN value to be returned. Only the square root of the positive denormalized and normal values need be calculated, as the outputs of the other special values can be determined from table 3.12. The implementation of denormalized numbers will not be considered but the values will be detected and flushed to zero to enable computation to continue.

3.8.2 Sign logic

From table 3.12 it can be seen that nothing needs to be done to the sign bit. The sign bit of NaN is never interpreted so if a negative value is passed to the square root unit the sign bit need not change.

3.8.3 Exponent handling

The floating-point square root algorithm requires the exponent to be divided by 2. However, the exponent must be even so that after the divide by 2 operation the resultant exponent is an integer. The exponent must be even; therefore, it is adjusted if it is odd by subtracting one from it and left shifting the significand by one bit. The

biasing complicates things slightly because it causes even exponents to have odd values and vice versa.

If the biased exponent is odd then the true exponent value is even and the incoming exponent is right shifted to divide it by 2. The right shifting operation causes the bias value to be halved and so half the bias value needs to be added back onto the exponent value.

If the biased exponent is even then the true exponent value is odd and it needs to be adjusted to make it even by subtracting 1 from the biased exponent value and subsequently adjusting the significand by left shifting. The adjusted exponent then needs to be right shifted and half the bias needs to be added.

The two scenarios of odd and even biased exponents are shown in figure 3.26.

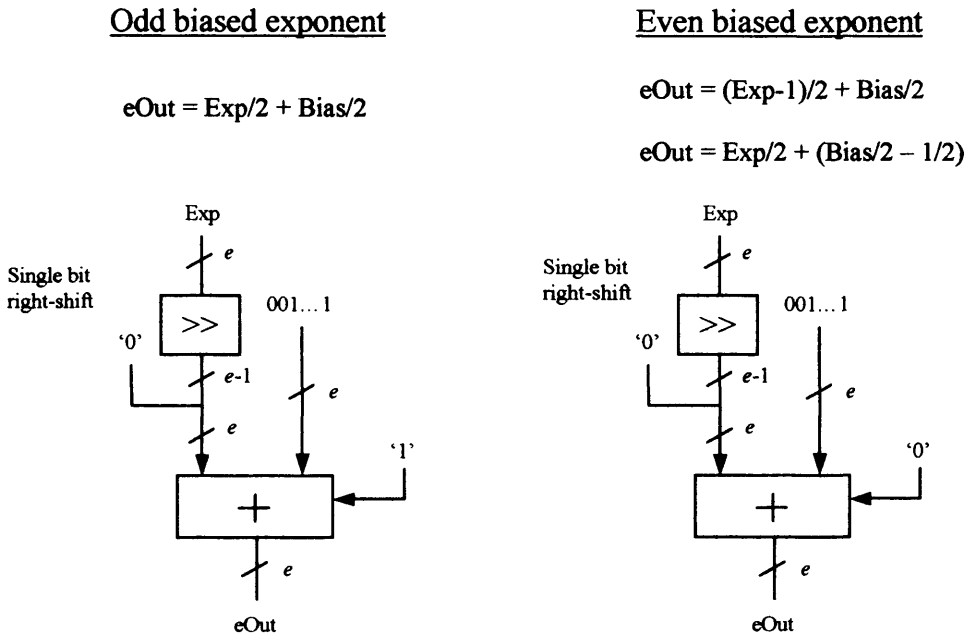


Figure 3.26. The hardware for handling odd and even exponents

The hardware required for the odd and even exponent scenarios is exactly the same with the only difference being the carry-in bit for the adder. The carry-in bit can be connected to the LSB of the incoming exponent.

3.8.4 Significand fixed-point square root

The implementation of fixed-point square root extractors was considered in the fixed-point arithmetic section and the efficient non-restoring design that was developed in section 2.11 can be used here. The range of the significand, including the scaled range due to an odd exponent, is [1, 4). The developed non-restoring square root component accepts fractional inputs. The significand range is clearly not fractional but can be scaled to be so, which is just a reinterpretation of the binary point. The reinterpretation of the binary point does not need any hardware to implement. The output of the non-restoring divider needs to be scaled so it is in the correct range of [1, 2) but again this is just a reinterpretation of the binary point. The square root result needs to be computed with an extra bit of precision, which acts as a rounding decision bit. The output of the significand square root extractor is the correct square root value and a remainder that is either negative or positive, but never zero.

3.8.5 Rounding, zero remainder detection and exponent adjusting

The non-restoring square root extractor outputs the correct square root value and an extra round bit. To implement the 4 IEEE rounding modes a sticky bit needs to be generated, which is done by checking the remainder. If the remainder is zero then the sticky bit is '0'; if the reminder is not zero then the sticky bit is '1'. The remainder produced by the square root extractor is incorrect if it is negative and in this case it needs to be corrected by adding on a left shifted quotient value with a single bit concatenated at the LSB end. After correction the remainder value could be positive or zero and a further check is needed to see if the reminder is zero, which can be accomplished using a wide OR gate. The remainder correction and subsequent zero detection can be implemented in a single component due to the following observation. The result of adding two operands will be zero if both operands LSBs are '1' and the logical XOR of every subsequent pair of digits is '1'. A component that will implement this algorithm and signal if the remainder is equal to zero or not is shown in figure 3.27. The component takes as input the remainder, the square root result and a single '1' bit. The *sticky bit* output in figure 3.27 is '0' if the remainder is zero and '1' if the remainder is not zero. The 4 IEEE rounding mode component can be used to round the square root and the output round bit can be fed into the rounding adder's carry-in line. The result of the rounding operation could require normalization and a

subsequent increment of the exponent. To handle this situation a similar method as was used for floating-point addition is adopted so the carry-out of the quotient rounding adder feeds the carry-in input of the exponent adjusting adder. Figure 3.28 illustrates the structure of the rounding stage that uses one carry-chain.

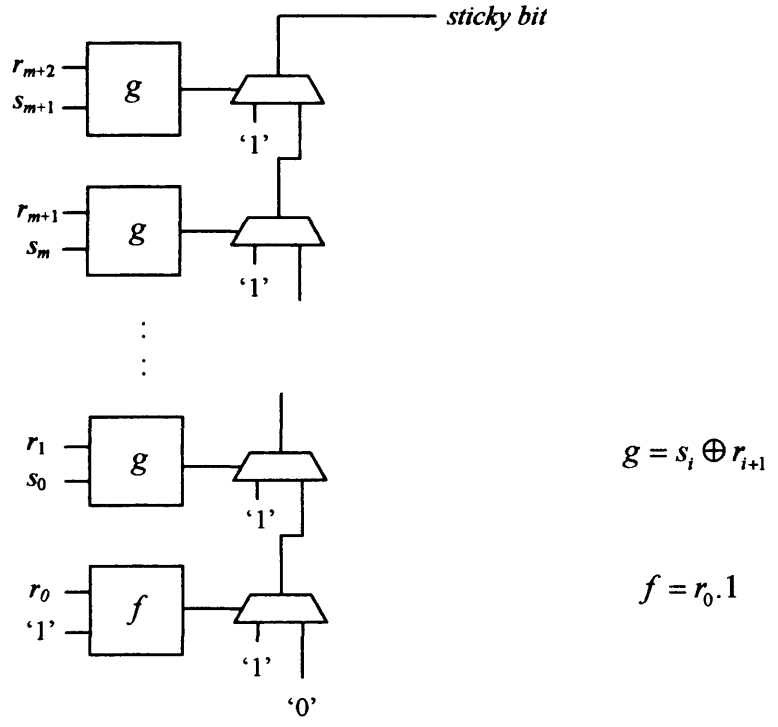


Figure 3.27. The zero remainder detection component

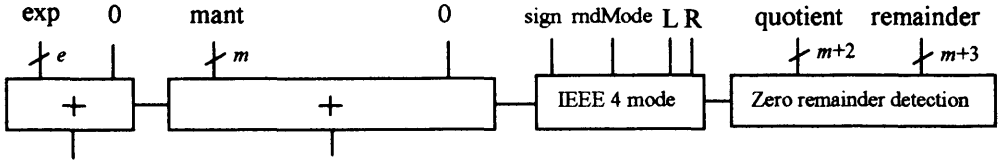


Figure 3.28. The floating-point square-root rounding logic

3.8.6 Overflow and underflow

The square root operation cannot overflow or underflow as the magnitude of the (unbiased) exponent is always halved (roughly).

3.8.7 Complete floating-point square root diagram

Combining the above points leads to the complete floating-point square root design, which is shown in figure 3.29.

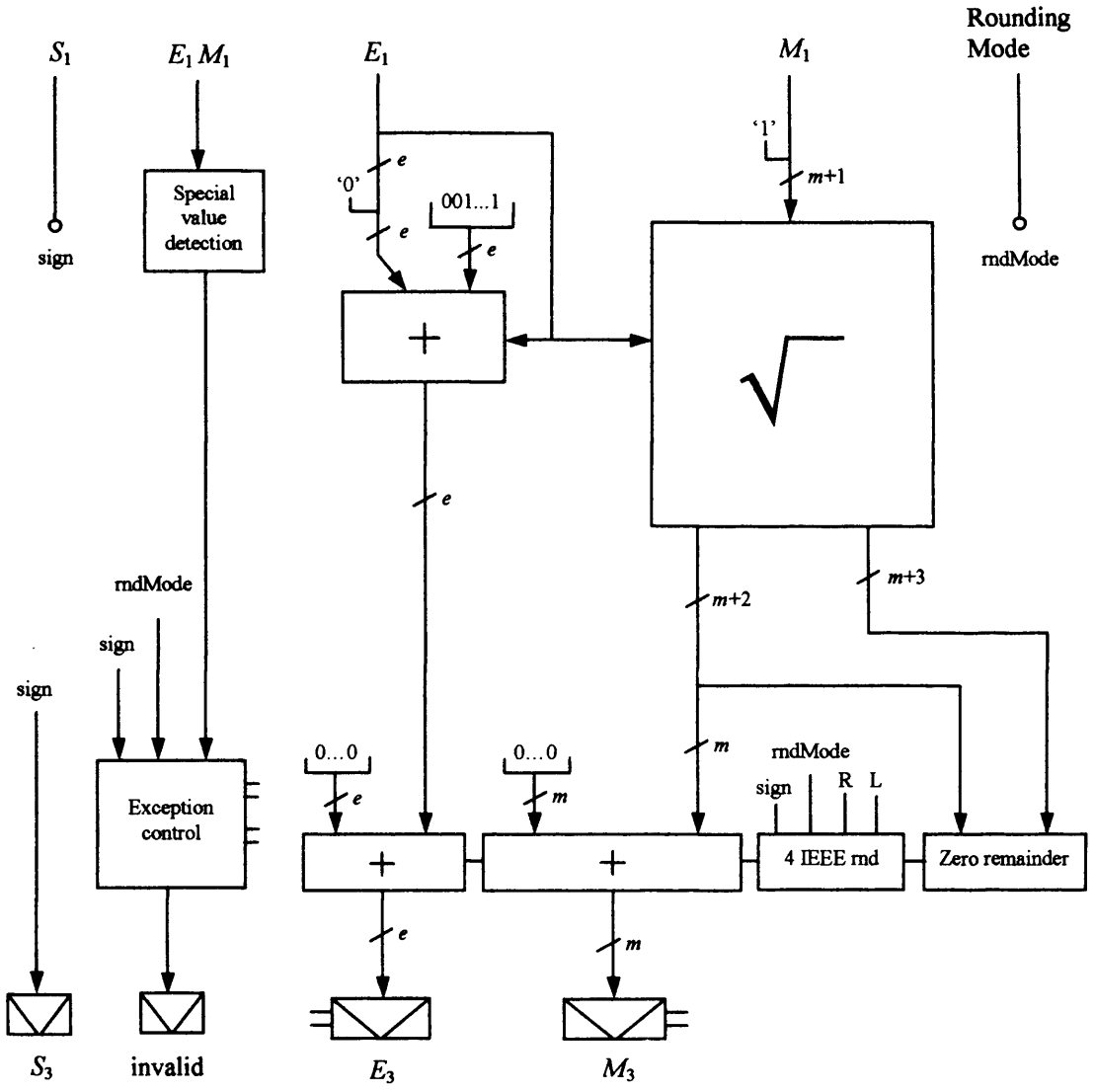


Figure 3.29. Complete floating-point square root diagram

3.9 Floating-point implementation results

All the floating-point operators have been designed to be parameterisable in terms of the exponent and mantissa width and these values are determined at design time. All other features are fixed and cannot be changed without changing the source code. The two metrics of area and delay are generated for various exponent and mantissa widths and these results are then compared with other FPGA designs available commercially and in the open literature. The designs have been pipelined so that with an exponent and mantissa width of 8 and 23-bits respectively the designs will operate at approximately 100 MHz. This is an arbitrary chosen speed but is realistic for modern

FPGA designs. Larger and smaller designs with a similar pipeline stage placement are expected to be slower and faster respectively.

To calculate the delay of a design the inputs and outputs must be registered as the place and route tools use the maximum register to register delay in order to calculate the maximum clock speed a design can be clocked at. To calculate the area of a design the input registers are removed and the design is run back through the place and route tools. The delay is measured in nanoseconds and the area is measured in slices and is also given in terms of LUT and flip-flop usage.

3.9.1 Fair comparison

Pipelining plays a very important role in ensuring the comparison of designs is fair. Heavily pipelined designs are larger due to the extra logic needed to store the registers and shift registers. Heavily pipelined designs also have a larger input to output delay if it is calculated using equation (3.6). The extra delay is due to unbalanced register placement.

$$total\ delay = pipeline\ stages * clock\ period \quad (3.6)$$

Automated register insertion schemes have been proposed but it is very time consuming to implement efficiently with a perfectly balanced delay between stages. Due to this problem the option of having a user configurable latency has not been investigated.

Lots of factors affect the speed and area of a design and make a fair comparison difficult. Such factors include: Technology production date, different vendors have different FPGA architectures, individual vendors have several different FPGA families with different architectures, the speed grade of a particular device affects the speed across a device family, the technology speed definition files, the place-and-route and synthesis tools, the number of pipelining stages of a design, the different floating-point options included in a design. Often some of the listed points are omitted when a design is presented and this can jeopardise the validity of comparisons. Where possible like for like comparisons will be made, however sometimes only an estimate of the equivalent metric can be made. In such circumstances a justification of the comparison will be made. The floating-point results available in the literature are split into groups with exponent lengths of 4, 6, 8

and 11 bits. The results of delay and area will be plotted for these exponent lengths showing how the results vary with mantissa length. The area will be given in slices where a single slice will be assumed to be equivalent to two LUTs. A design area quoted in LUTs will be converted into slices remembering that the number of slices of a design is not exactly equal to the LUT quantity divided by two (it is usually greater) so this is an optimistic estimate designed to ensure the proposed designs do not have an unfair advantage. The delay is calculated by multiplying the number of clock cycles that a design requires by the delay of one clock cycle. The delay of the proposed designs will also be calculated this way despite the fact that the non-pipelined delay has been calculated. Again this is done to try and ensure the proposed designs do not have an unfair advantage. As mentioned efficiently implementing a 'pipeline stage quantity' option is nontrivial and so the fairest and simplest way of calculating the delay for all designs is to multiply the number of pipeline stages by the minimum clock period. If a one off pipelined design is created the designer will try and place the registers in the most optimum position i.e. with equal delay between the registers. Thus for non-automated register placed designs the balance of delay between registers will be good and the proposed delay calculating metric seems a valid choice.

3.9.2 Floating-point addition results

Table 3.13 summarises the floating-point addition Xilinx Virtex-II-4 FPGA implementation results using version 5.1.03i of the Xilinx place and route tools and technology speed files created by Xilinx on 01/11/2002. The Xilinx tools are used for the complete design flow from design entry to FPGA programming bit stream.

There are many floating-point adder designs available to compare with the results of table 3.13. In tables 3.14, 3.15 , 3.16 and 3.17 the main results from the literature are presented. Table 3.14 groups the results in terms of a 4-bit exponent width, table 3.15 in terms of a 6-bit exponent width and table 3.16 in terms of an 8-bit exponent width. The tables contain 16 different columns, which are described as follows:

Author: This is the corresponding reference.

e : is the exponent width.

m : is the mantissa width.

Architecture : This describes the algorithm style where either the vanilla algorithm, dual-path (2-path) method or a combined structure have been used. The word format

is also described. The IEEE word format is most popular but there are some custom implementations.

Add/Sub : does the adder perform addition, subtraction or both?

Rounding : here the rounding modes that are supported are listed. RTNE is round-to-nearest-even, TRUNC is truncate and RTN is round-to-nearest. Certain implementations also support all four of the IEEE rounding modes.

Special values : what special values can the component handle? There is a choice of Denormalised numbers, Infinity and NaN.

Exception flags : what flag outputs are provided? There is a choice of ov (overflow), un (underflow), invalid, inexact and dbz (divide by zero) (for division only).

Param : Is the design generated from a parameterisable library?

Cycles : This states how many pipeline stages are used in the implementation.

Cycle Delay : is the minimum clock period in nanoseconds.

Total delay : is the total delay given by the author in nanoseconds. It is usually the non-pipelined delay of the design.

Cycles*delay : this gives the delay of the design in terms of the number of pipeline stages multiplied by the clock period of each stage.

Area : is the area of the design expressed in slices or the equivalent area of the given FPGA technology in terms of Virtex slices.

Chip maker : this is the device model, speed grade and the device vendor name.

Year : is the year of publication of the reference.

(e,m)	Area (slices)	Area ffs/LUTs	Delay (ns)	Pipeline stages	Area (slices)	Area ffs/LUTs	Delay (ns)	Stages* delay
4,5	99	13/178	22.5	4	119	91/196	8.1	32.4
4,7	127	15/232	26.1	4	146	106/242	8.2	32.8
4,9	144	17/263	26.2	4	164	120/274	8.3	33.2
4,11	160	19/295	26.8	4	182	134/306	8.7	34.8
6,9	153	19/285	27.5	4	178	130/301	8.8	35.2
6,11	170	21/316	27.5	4	195	144/332	8.9	35.6
6,13	187	23/350	28	4	213	158/366	8.9	35.6
6,15	217	25/406	30.4	4	252	173/437	9.7	38.8
6,17	235	27/442	31	4	272	187/475	9.9	39.6
6,19	252	29/475	31.5	4	292	201/511	9.9	39.6

Table 3.13. Floating-point adder implementation results for Xilinx Virtex-II-4

(e,m)	Area (slices)	Area ffs/LUTs	Delay (ns)	Pipeline stages	Area (slices)	Area ffs/LUTs	Delay (ns)	Stages* delay
8,11	177	23/330	27	4	203	154/346	9	36
8,13	194	25/363	28.2	4	221	168/379	9	36
8,15	225	27/420	31.5	4	260	183/451	10	40
8,17	242	29/455	32.1	4	280	197/488	10	40
8,19	259	31/489	32.7	4	300	211/525	10.2	40.8
8,21	279	33/525	32.9	4	321	225/562	10.2	40.8
8,23	296	35/560	33.5	4	340	239/597	10.2	40.8
8,25	316	37/597	33.8	4	361	253/635	10.5	42
10,29	367	43/691	35.6	4	418	291/730	11.2	44.8
11,52	642	67/1223	42.9	4	694	458/1240	13.1	52.4

Table 3.13. Floating-point adder implementation results for Xilinx Virtex-II-4

Author	e	m	Architecture	Add/sub	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles* delay	Area slices	Chip maker	Year
Belanovic [170]	4	3	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6	~20 ns		120	32	Xilinx Virtex	02
Belanovic [170]	4	7	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6	~20 ns		120	80	Xilinx Virtex	02
Detrey [180]	4	7	2 path, IEEE word	both	RTNE	Inf, NAN	invalid	yes	4	10 ns	24 ns	40	163	Xilinx VirtexII-4	03
Narasimhan [154]	4	9	Vanilla, IEEE word	add	none	none	none	no	8	10 ns		80	161	Xilinx XC4005	93
Roesler [134]	4	11	Vanilla, IEEE word	both	RTNE only	Inf, NaN	none	yes	20	4.5 ns		81	392	Xilinx VirtexII-6	02
Belanovic [170]	4	11	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6	~20 ns		120	121	Xilinx Virtex	02

Table 3.14. Floating-point adder implementation results for an exponent of 4-bits

Author	e	m	Architecture	Add/sub	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles* delay	Area slices	Chip maker	Year
Quixilica [176]	6	7	Vanilla, IEEE word	both	RTNE	Inf, NaN	(3) not exact	yes	10	6.5 ns		65	121	Xilinx Virtex-6	02
Belanovic [170]	6	9	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6			120	113	Xilinx Virtex	02
Shirazi [157]	6	9	Vanilla, IEEE word	both	none	none	none	no	3	108 ns		324	104	Xilinx XC4010	95
Quixilica [176]	6	11	Vanilla, IEEE word	both	RTNE	Inf, NaN	(3) not exact	yes	10	7 ns		70	158	Xilinx Virtex-6	02
Lee [51]	6	13	2 path, IEEE word	both	4 IEEE modes	Inf, NaN	(3) not inexact	yes	4	8.5 ns	28 ns	34	228	Xilinx VirtexII-4	02
Detrey [180]	6	13	2 path, IEEE word	both	RTNE	Inf, NAN	invalid	yes	4	10 ns	24 ns	40	257	Xilinx VirtexII-4	03
Quixilica [176]	6	15	Vanilla, IEEE word	both	RTNE	Inf, NaN	(3) not exact	yes	11	6.5 ns		71.5	208	Xilinx Virtex-6	02
Roesler [134]	6	16	Vanilla, IEEE word	both	RTNE only	Inf, NaN	none	yes	23	5 ns		115	584	Xilinx VirtexII-6	02

Table 3.15. Floating-point adder implementation results for an exponent of 6-bits

Author	e	m	Architecture	Add/sub	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles* delay	Area slices	Chip maker	Year
Ho [169]	8	7	IEEE word	both	none	none	none	yes	7	17 ns		119	120	Xilinx Virtex-6	02
Shirazi [157]	8	10	Vanilla, IEEE word	both	none	none	none	no	3	116 ns		348	112	Xilinx XC4010	95
Liang [178]	8	13	2 path, IEEE word	both	RTNE, TRUNC	none	none	yes	1		35 ns	35	220	Xilinx Virtex-6	03
Quixilica [176]	8	13	Vanilla, IEEE word	both	RTNE	Inf, NaN	(3) not exact	yes	11	7.3 ns		80.3	306	Xilinx Virtex-6	02
Detrey [180]	8	15	2 path, IEEE word	both	RTNE	Inf, NAN	invalid	yes	4	10 ns	28 ns	40	298	Xilinx VirtexII-4	03
Belanovic [170]	8	15	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6			120	216	Xilinx Virtex	02
Ho [169]	8	15	IEEE word	both	none	none	none	yes	7	22 ns		154	225	Xilinx Virtex-6	02
Liang [178]	8	23	2 path, IEEE word	both	RTNE, TRUNC	none	none	yes	1		40 ns	40	400	Xilinx Virtex-6	03
Detrey [180]	8	23	2 path, IEEE word	both	RTNE	Inf, NAN	invalid	yes	4	10 ns	31 ns	40	416	Xilinx VirtexII-4	03
Lee [51]	8	23	2 path, IEEE word	both	4 IEEE modes	Inf, NaN	(3) not inexact	yes	4	10.2 ns	32 ns	40.8	366	Xilinx VirtexII-4	02
Digital core design [168]	8	23	IEEE word	both	RTNE	Denorm, inf, NaN	(3) not inexact	no	4	13.7 ns		54.8	580	Xilinx VirtexII-5	01
Flores [179]	8	23	Vanilla, IEEE word	both	none	none	none	yes	8	9.5 ns		76	400	Xilinx VirtexII-6	03
Flores [179]	8	23	Vanilla, IEEE word	both	none	none	none	yes	2	40 ns		80	290	Xilinx VirtexII-6	03
Nallatech [175]	8	23	Custom word	both		Inf, NaN	(2) ov, invalid	no	14	6.6 ns		92.4	290	Xilinx VirtexII-4	02
Nichols [174]	8	23	IEEE word	both				no	2	50.5 ns	101 ns	101	696	Xilinx VirtexII-6	02
Roesler [134]	8	23	Vanilla, IEEE word	both	RTNE only	Inf, NaN	none	yes	23	4.7 ns		108.1	773	Xilinx VirtexII-6	02
Belanovic [170]	8	23	Vanilla, IEEE word	both	RTN, TRUNC	none	none	yes	6			120	291	Xilinx Virtex	02
Ho [169]	8	23	IEEE word	both	none	none	none	yes	7	24 ns		168	336	Xilinx Virtex-6	02
Jaenicke [167]	8	23	Vanilla, IEEE word	both	4 IEEE modes	Denorm, inf, NaN	(4) All IEEE	yes	8	35 ns		280		Xilinx Virtex	01
Ligon III [161]	8	23	Vanilla, IEEE word	both	RTNE only	none	none	no	15	25 ns		375	336	Xilinx XC4020	98
Louca [158]	8	23	Vanilla, IEEE word	both	none	none	none	no	3	143 ns	385 ns	429	240	Altera flex 81188	96
Fagin [156]	8	23	Combined add/mult	both	4 IEEE modes	Denorm, inf, NaN	(4) All IEEE	no	3	245 ns		735	2050	Actel A1280	94
Ho [169]	8	31	IEEE word	both	none	none	none	yes	7	25 ns		175	455	Xilinx Virtex-6	02
Liang [178]	8	32	2 path, IEEE word	both	RTNE, TRUNC	none	none	yes	1		46 ns	46	525	Xilinx Virtex-6	03
Roesler [134]	8	32	Vanilla, IEEE word	both	RTNE only	Inf, NaN	none	yes	24	5.8 ns		139.2	1010	Xilinx VirtexII-6	02

Table 3.16. Floating-point adder implementation results for an exponent of 8-bits

Author	e	m	Architecture	Add/sub	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles* delay	Area slices	Chip maker	year
Lee [51]	11	52	2 path, IEEE word	both	4 IEEE modes	Inf, NaN	(3) not inexact	yes	4	14.3 ns	43 ns	57.2	797	Xilinx VirtexII-4	02
Paschalakis [181]	11	52	Vanilla, IEEE word	both	RTNE	Inf, NaN	none	no	1		120 ns	120	675	Xilinx Virtex-6	03

Table 3.17. Floating-point adder implementation results for an exponent of 11-bits

3.9.2.1 Comparison of floating-point addition results

To compare each set of results for a particular exponent width two graphs of area and delay are used. The graphs compare the results in the literature (tables 3.14-3.16) with the results in table 3.13. The graphs compare the complete range of exponent and mantissa widths so that the advantages of different implementations can be seen. Figures 3.30, 3.32 and 3.34 show how the area of different 4, 6 and 8-bit exponent implementations vary with mantissa width. Figures 3.31, 3.33 and 3.35 show how the delay of 4, 6 and 8-bit exponent implementations vary with mantissa width.

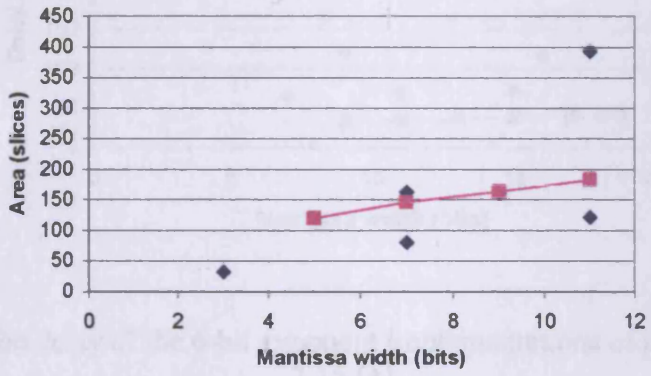


Figure 3.30. The area of the 4-bit exponent implementations of tables 3.13 [■] and 3.14 [♦]

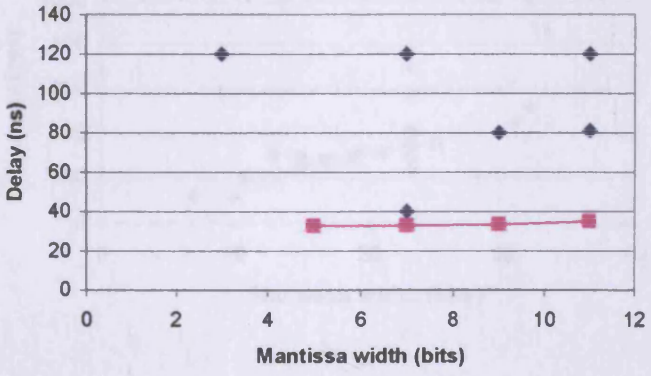


Figure 3.31. The delay of the 4-bit exponent implementations of tables 3.13 [■] and 3.14 [♦]

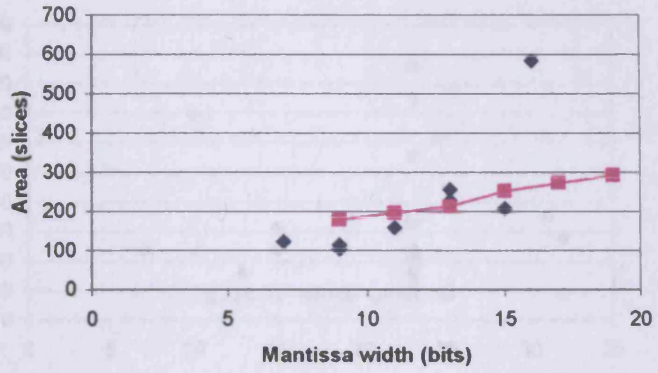


Figure 3.32. The area of the 6-bit exponent implementations of tables 3.13 [■] and 3.15 [♦]

Figure 3.33. The delay of the 6-bit exponent implementations of tables 3.13 [■] and 3.15 [♦]

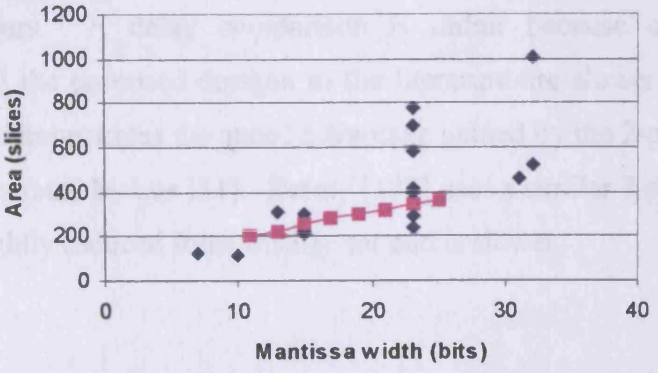


Figure 3.34. The area of the 8-bit exponent implementations of tables 3.13 [■] and 3.16 [♦]

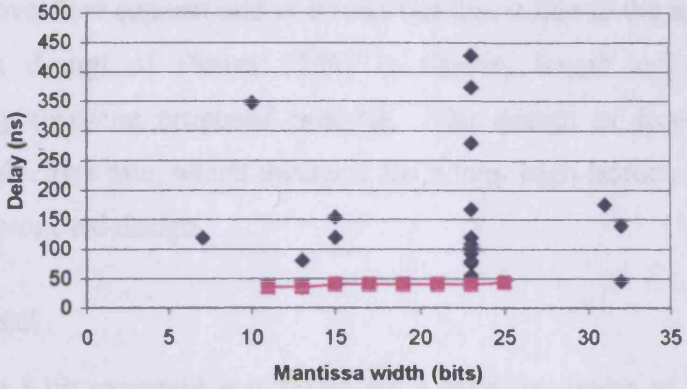


Figure 3.35. The delay of the 8-bit exponent implementations of tables 3.13 [■] and 3.16 [♦]

3.9.2.2 Result discussion

4-bit exponent

The designs of Belanovic [170], which are smaller than the proposed method, all have less functionality and are implemented using the vanilla algorithm. The designs of [170] do not offer the default IEEE rounding mode, any support of special values or any exception flags despite the author's claims that the modules are a superset of all previous FPGA implementations. Speed results for the designs of [170] are not given but from Belanivic [171] and personal communication at FPL'02 they can be deduced as being approximately 50 MHz and means they are slower than the proposed designs. The design of Narasimhan [154] despite being of similar area does not include rounding, overflow/underflow detection, special value handling and can only add unsigned numbers. A delay comparison is unfair because of the age of the technology. All the proposed designs in the literature are slower than the proposed design and this demonstrates the speed advantage gained by the 2-path method, which was first demonstrated by Lee [51]. Detrey [180] uses a similar 2-path method but the design has a slightly reduced functionality set and is slower.

6-bit exponent

All the designs that are smaller than the proposed method offer a reduced functionality and are based on the vanilla algorithm. The design of Quixilica [176] supports only the RTNE rounding mode and is slower than the proposed method. The

method of Shirazi [157] does not offer any rounding, special value or underflow/overflow support and is slower but this is due to the age of the technology. The 2-path design of Detrey [180] is slower, larger and offers slightly less functionality than the proposed method. The design of Roesler [134] is geared towards high clock rate, which means it has a very high latency and is over twice the size of the proposed design.

8-bit exponent

Most of the 8-bit exponent results are for a mantissa width of 23-bits, which is the length specified by the IEEE single precision format. The designs of Belanovic [170], Ho [169], Flores [179], Nallatech [175], Louca [158] for an 8-bit exponent that are smaller than the proposed design all have reduced functionality and all use the vanilla algorithm. Out of the methods that are larger only Digital Core Design [168] and Jaenicke [167] have better functionality as they both support denormalized arithmetic. Fagin [156] also supports denormalized arithmetic but combines floating-point addition and multiplication in a single component, which makes a fair comparison almost impossible. The proposed method is faster than all other methods apart from the 2-path method of Detrey [180] which is larger and has a slightly reduced functionality. All other designs are larger and have a reduced amount of functionality and so need not be considered.

11-bit exponent

There is only one other double precision floating-point addition design Paschalakis [181]. This design uses the vanilla algorithm, has reduced functionality and is also slightly smaller. However, the design is 9 times slower.

3.9.2.3 Conclusion

The original 2-path method implemented in Lee [51] is the fastest floating-point addition algorithm as confirmed recently by Liang [178] and Detrey [180] where similar 2-path algorithms are used. The method presented in this work is an extension of the original published implementation Lee [51] offering an area and delay reduction that improves with word length. The 2-path algorithm is larger than the basic vanilla algorithm because of the extra addition and normalisation logic required, which is evident because all the designs in the comparison graphs that are smaller

than the proposed design use the vanilla algorithm (they also have a reduced functionality set of some description). The design given is smaller than any other published 2-path algorithm and offers a greater level of functionality. When the design is scaled to very large word lengths it still offers very good speed performance as shown by the 9x speed up over a recently published 11-bit exponent 52-bit mantissa double precision implementation Paschalakis [181]. The support for denormalized arithmetic increases the size of the adder by about 50% as shown by Digital Core Design [168].

3.9.3 Floating-point multiplication results

Table 3.18 summarises the floating-point multiplication Xilinx Virtex-II-4 FPGA implementation results using version 5.1.03i of the Xilinx place and route tools and speed definition files created on 01/11/2002. The Xilinx tools are used for the complete design flow from entry to FPGA configuration bit stream generation.

(e,m)	Area (slices)	Area ffs/LUTs	Delay (ns)	18X18 mults	Pipeline stages	Area (slices)	Area ffs/LUTs	Delay (ns)	Stages* delay
(4,5)	33	13/59	16.5	1	3	57	50/64	8	24
(4,7)	34	15/62	16.8	1	3	59	54/67	8.2	24.6
(4,9)	38	17/66	17.8	1	3	64	58/71	8.3	24.9
(4,11)	39	19/70	18.7	1	3	67	62/75	8.4	25.2
(6,7)	38	17/71	16.9	1	3	65	60/76	8.3	24.9
(6,9)	41	19/75	18.4	1	3	70	64/80	8.3	24.9
(6,11)	43	21/79	18.8	1	3	73	68/84	8.4	25.2
(6,13)	44	23/84	19.4	1	3	75	72/89	8.6	25.8
(6,15)	46	25/87	19.9	1	3	77	76/92	9	27
(6,17)	67	27/127	23.1	1	4	148	172/147	9.1	36.4
(6,19)	109	29/206	25.1	1	4	173	223/226	9.2	36.8
(6,21)	155	31/297	25.7	1	4	201	201/316	9.9	39.6
(8,9)	43	21/78	18.7	1	3	74	70/83	8.4	25.2
(8,11)	45	23/82	18.8	1	3	77	74/87	8.6	25.8
(8,13)	46	25/87	19.1	1	3	79	78/92	8.8	26.4
(8,15)	48	27/90	20.7	1	3	81	82/95	9.1	27.3
(8,17)	69	29/130	23.3	1	4	152	176/152	9.1	36.4
(8,19)	111	31/209	25.2	1	4	177	327/231	9.4	37.6
(8,21)	157	33/300	26	1	4	205	205/321	9.8	39.2
(8,23)	207	35/396	26.6	1	4	254	258/420	10	40
(8,25)	262	37/502	29.3	1	4	333	312/527	10.1	40.1
(10,29)	390	43/746	31	1	4	465	395/772	11.7	46.8
(10,39)	353	53/673	33.3	4	4	456	418/696	14.1	56.4
(11,52)	321	67/605	35.6	9	5	513	737/640	13.1	65.3

Table 3.18. Floating-point multiplication implementation results for a Virtex-II-4

As for floating-point addition there are many floating-point multiplication designs available to compare with the results of table 3.18. In tables 3.19, 3.20, 3.21 and 3.22 the main results from the literature are presented and are grouped by exponent widths of 4, 6, 8 and 11-bits respectively. Tables 3.19-3.22 contain 16 columns many of which are the same as those of the equivalent floating-point addition tables. The 'Architecture' column differs as it describes the different multiplier architecture a design uses, which could be bit-serial, digit-serial and is assumed to be a parallel multiplier if unspecified. The word format is also listed here. The '18X18 mults' column specifies the number of 18X18-bit embedded multipliers used in the design. This is specific to Virtex-II FPGA designs.

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles * delay	Area slices	18X18 mults	Chip maker	Year
Belanovic [170]	4	3	IEEE word	RTN, TRUNC	none	none	yes	5	~20ns		100	36		Xilinx XCV1000	02
Detrey [180]	4	7	IEEE word	RTNE	Inf, NaN	invalid	yes	4	10 ns	17 ns	40	75		Xilinx VirtexII-4	03
Belanovic [170]	4	7	IEEE word	RTN, TRUNC	none	none	yes	5	~20 ns		100	140		Xilinx XCV1000	02
Roesler [134]	4	11	IEEE word	RTNE only	Inf, NaN	none	yes	6	4ns		24	81	1	Xilinx VirtexII-6	02
Belanovic [170]	4	11	IEEE word	RTN, TRUNC	none	none	yes	5	~20ns		100	208		Xilinx XCV1000	02
Jaenicke [167]	4	11	IEEE word	4 IEEE modes	Denorm, Inf, NaN	(4) All IEEE	yes	5				300		Xilinx XCV1000	01

Table 3.19. Floating-point multiplication implementation results for a 4-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles * delay	Area slices	18X18 mults	Chip maker	Year
Quixilica [176]	6	7	IEEE word	RTNE	Inf, NaN	(3) not inexact	yes	5	7.7 ns		38.5	67		Xilinx Virtex-6	02
Belanovic [170]	6	9	IEEE word	RTN, TRUNC	none	none	yes	5	~20ns		100	150		Xilinx XCV1000	02
Shirazi [157]	6	9	IEEE word	none	none	none	no	3	167 ns		501	150		Xilinx XC4010	95
Quixilica [176]	6	11	IEEE word	RTNE	Inf, NaN	(3) not inexact	yes	6	7.7 ns		46.2	119		Xilinx Virtex-6	02
Lee [51]	6	13	IEEE word	4 IEEE modes	Inf, NaN	(3) not inexact	yes	3	9.2 ns	20 ns	27.6	98	1	Xilinx VirtexII-4	02
Detrey [180]	6	13	IEEE word	RTNE	Inf, NaN	invalid	yes	4	10 ns	25 ns	40	155		Xilinx VirtexII-4	03
Quixilica [176]	6	15	IEEE word	RTNE	Inf, NaN	(3) not inexact	yes	6	8 ns		48	171		Xilinx Virtex-6	02

Table 3.20. Floating-point multiplication implementation results for a 6-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles * delay	Area slices	18X18 mults	Chip maker	Year
Roesler [134]	6	16	IEEE word	RTNE only	Inf, NaN	none	yes	6	5.7 ns		34.2	117	1	Xilinx VirtexII-6	02
Quixilica [176]	6	19	IEEE word	RTNE	Inf, NaN	(3) not inexact	yes	6	8.2 ns		49.2	229		Xilinx Virtex-6	02

Table 3.20. Floating-point multiplication implementation results for a 6-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles * delay	Area slices	18X18 mults	Chip maker	Year
Ho [169]	8	7	IEEE word		none	none	yes	8	9.7 ns		77.6	178		Xilinx XCV1000-6	02
Shirazi [157]	8	10	IEEE word	none	none	none	no	3	204 ns		612	180		Xilinx XC4010	95
Detrey [180]	8	15	IEEE word	RTNE	Inf, NaN	invalid	yes	5	10 ns	24 ns	50	195		Xilinx VirtexII-4	03
Ho [169]	8	15	IEEE word		none	none	yes	8	9.8 ns		78.4	375		Xilinx XCV1000-6	02
Belanovic [170]	8	15	IEEE word	RTN, TRUNC	none	none	yes	5	~20ns		100	431		Xilinx XCV1000	02
Lee [51]	8	23	IEEE word	4 IEEE modes	Inf, NaN	(3) not inexact	yes	4	10.5 ns	26 ns	42	230	1	Xilinx VirtexII-4	02
Roesler [134]	8	23	IEEE word	RTNE only	Inf, NaN	none	yes	8	5.9 ns		47.2	248	4	Xilinx VirtexII-6	02
Quixilica [176]	8	23	IEEE word	RTNE	Inf, NaN	(3) not inexact	yes	6	8.2 ns		49.2	326		Xilinx Virtex-6	02
Detrey [180]	8	23	IEEE word	RTNE	Inf, NaN	invalid	yes	5	10 ns	26 ns	50	388		Xilinx VirtexII-4	03
Nallatech [175]	8	23	Custom word		Inf, NaN	(2) ov, Invalid	no	6	8.8 ns		52.8	126	4	Xilinx VirtexII	02
Digital core design [168]	8	23	IEEE word	RTNE	Denorm, Inf, NaN	(3) not inexact	no	4	13.5 ns		54	677	4	Xilinx VirtexII-5	01
Flores [179]	8	23	IEEE word	none	none	none	yes	13	5.7 ns		74.1	973		Xilinx VirtexII-6	03
Ho [169]	8	23	IEEE word		none	none	yes	8	10 ns		80	598		Xilinx XCV1000-6	02
Belanovic [170]	8	23	IEEE word	RTN, TRUNC	none	none	yes	5	~20ns		100	674		Xilinx XCV1000	02
Nichols [174]	8	23	IEEE word				no	2	55 ns	110 ns	110	734		Xilinx XC2V8000	02
Jaenicke [167]	8	23	IEEE word	4 IEEE modes	Denorm, Inf, NaN	(4) All IEEE	yes	5	35 ns		175	880		Xilinx XCV1000	01
Sahin [165]	8	23	IEEE word			none	no	8	29 ns		232	834		Xilinx XC4044XL	00
Ligon III [161]	8	23	Digit-serial, IEEE word	RTNE only	none	none	no	15	30 ns		450	380		Xilinx XC4020E	98
Samet [159]	8	23	Bit-serial, IEEE word	none	none	none	no	26	27 ns		702	78		Xilinx XC4005	97
Fagin [156]	8	23	Combined add/mult	4 IEEE modes	Denorm, NaN, inf	(4) All IEEE	no	6	245 ns		1470	2050		Actel 1280	94
Louca [158]	8	23	Digit-serial, IEEE word	none	none	none	no	6	434 ns		2604	245		Altera flex 81188	96
Novak [155]	8	23	Bit serial, IEEE word	RTN	none	none	no			5ms	5000	620		Actel 1280	94
Ho [169]	8	31	IEEE word		none	none	yes	8	10 ns		80	694		Xilinx XCV1000-6	02
Roesler [134]	8	32	IEEE word	RTNE only	Inf, NaN	none	yes	8	5.8 ns		46.4	308	4	Xilinx VirtexII-6	02

Table 3.21. Floating-point multiplication implementation results for 8-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Total delay	Cycles * delay	Area slices	18X18 mults	Chip maker	Year
Jaenicke [167]	10	29	IEEE word	4 IEEE modes	Denorm, Inf, NaN	(4) All IEEE	yes	5				1330		Xilinx XCV1000	01
Lee [51]	11	52	IEEE word	4 IEEE modes	Inf, NaN	(3) not inexact	yes	5	15.1 ns	37 ns	75.5	321	9	Xilinx VirtexII-4	02
Paschalakis [181]	11	52	Digit-serial IEEE word	RTNE	Inf, NaN	none	no	1		270 ns	270	495		Xilinx Virtex-6	03

Table 3.22. Floating-point multiplication implementation results for 10 and 11-bit exponents

3.9.3.1 Comparison of floating-point multiplication results

To compare each set of results for a particular exponent width two graphs of area and delay are used. The graphs compare the results in the literature (tables 3.19-3.21) with the results in table 3.18. Figures 3.36, 3.38 and 3.40 show how the area of different 4, 6 and 8-bit exponent implementations vary with mantissa width. Figures 3.37, 3.39 and 3.41 show how the delay of different 4, 6 and 8-bit exponent implementations vary with mantissa width.

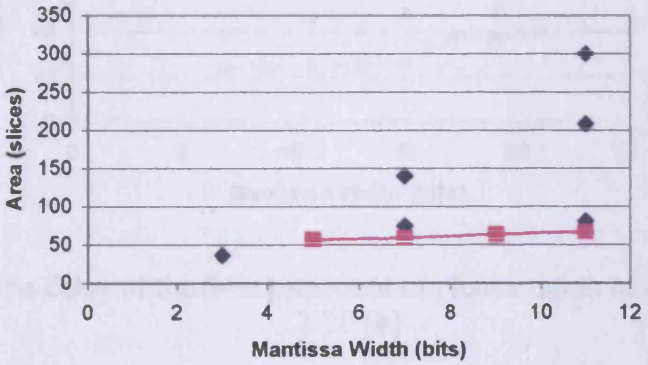


Figure 3.36. The area of the 4-bit exponent implementation of tables 3.18 [■] and 3.19 [♦]

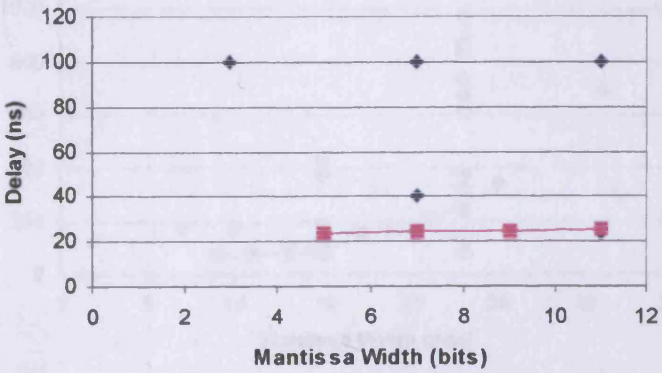


Figure 3.37. The delay of the 4-bit exponent implementation of tables 3.18 [■] and 3.19 [♦]

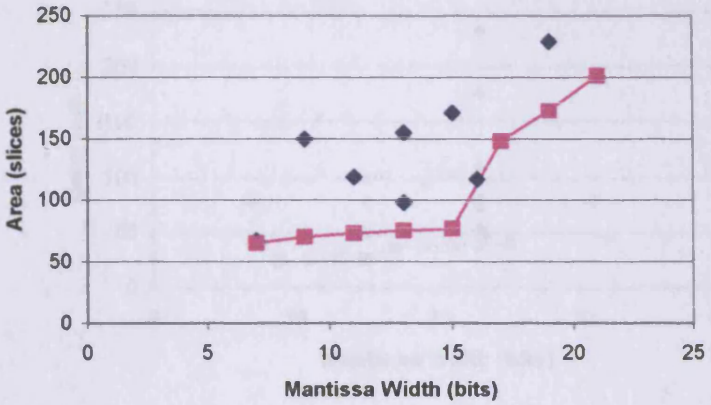


Figure 3.38. The area of the 6-bit exponent implementation of tables 3.18 [■] and 3.20 [♦]

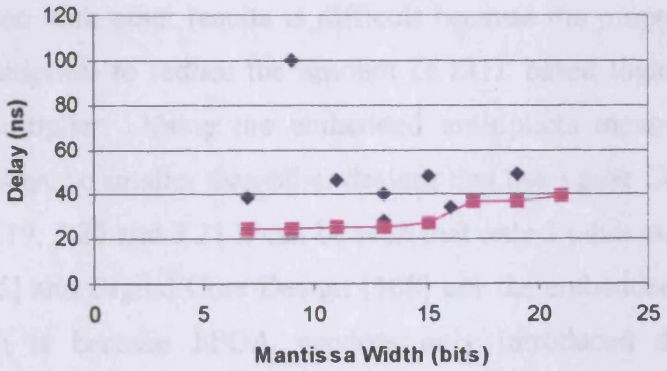


Figure 3.39. The delay of the 6-bit exponent implementation of tables 3.18 [■] and 3.20 [♦]

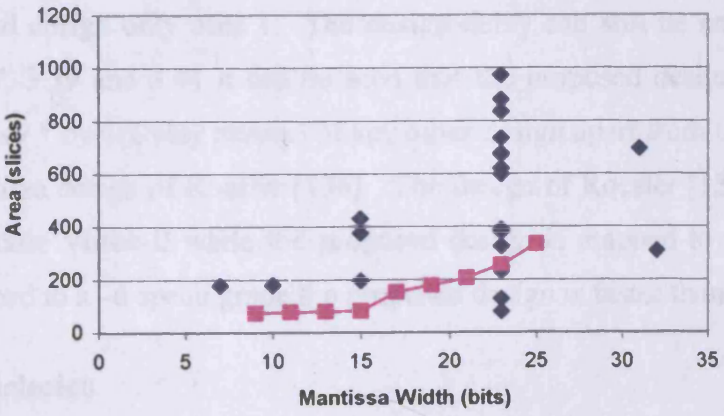


Figure 3.40. The area of the 8-bit exponent implementation of tables 3.18 [■] and 3.21 [♦]

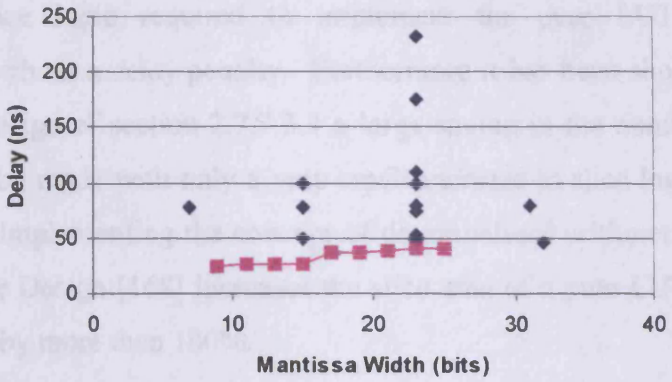


Figure 3.41. The delay of the 8-bit exponent implementation of tables 3.18 [■] and 3.21 [♦]

3.9.3.2 Result discussion

Fair comparison with other results is difficult because the proposed design uses the embedded multipliers to reduce the amount of LUT based logic that is used in the fixed-point multiplier. Using the embedded multipliers means that the proposed design will always be smaller than other designs that use a pure LUT based multiplier. From tables 3.19, 3.20 and 3.21 it can be seen that only 3 other authors Roesler [134], Nallatech [175] and Digital Core Design [168] use the embedded multipliers in their design, which is because FPGA vendors only introduced dedicated embedded multipliers in the past 4 years and many designs have not migrated to this technology. The designs that are smaller than the proposed design in terms of slice area use either digit-serial multiplication techniques Ligon III [161], Samet [159] and Louca [158] or use 4-embedded multipliers in their design Roesler [134] and Nallatech [175] when the proposed design only uses 1. The design delay can still be compared and from figures 3.37, 3.39 and 3.41 it can be seen that the proposed design has the smallest delay (latency * cycleDelay metric) of any other design apart from the 4-bit exponent, 11-bit mantissa design of Roesler [134]. The design of Roesler [134] is mapped to a –6 speed grade Virtex-II while the proposed design is mapped to a –4 speed grade. When mapped to a –6 speed grade the proposed design is faster than that of Roesler.

3.9.3.3 Conclusion

The smallest designs use bit-serial arithmetic for the implementation of the fixed-point multiplier but these designs are slow and cannot be pipelined to increase the

clock rate. Using the embedded multipliers in the fixed-point multiplication design reduces the slice logic required to implement the pure LUT based parallel multiplication without a delay penalty. Furthermore it has been shown that by using the multiplier design of section 2.7.7.3.4 a large saving in the number of embedded multipliers can be made with only a very small increase in slice logic and without a speed penalty. Implementing the concept of denormalized arithmetic Jaenicke [167] and Digital Core Design [168] increases the slice area of a pure LUT based floating-point multiplier by more than 100%.

3.9.4 Floating-point division results

Table 3.23 summarises the floating-point division Xilinx Virtex-II-4 FPGA implementation results using version 5.1.03i of the Xilinx place and route tools and speed definition files created on 01/11/2002. The Xilinx tools are used for the complete design flow from entry to FPGA configuration bit stream generation.

(e,m)	Area (slices)	Area ffs/LUTs	Delay (ns)	Pipeline stages	Area (slices)	Area ffs/LUTs	Delay (ns)	Stages* delay
(4,5)	75	14/137	31.2	6	115	102/163	7.8	46.8
(4,7)	98	16/183	38.3	7	151	143/208	7.8	54.6
(4,9)	127	18/238	46.8	8	193	192/266	7.9	63.2
(4,11)	158	20/300	58.7	9	240	249/331	9	81
(6,5)	79	16/146	31.2	6	123	106/172	8.2	49.2
(6,7)	103	18/194	38.5	7	157	147/221	8.6	60.2
(6,9)	132	20/249	47.3	8	199	196/279	8.8	70.4
(6,11)	162	22/309	59	9	246	253/344	9	81
(6,13)	198	24/380	66.5	10	298	318/418	9.1	91
(6,15)	237	26/458	74.6	11	356	391/499	9.6	105.6
(6,17)	282	28/545	83.7	12	422	472/589	9.7	116.4
(6,19)	330	30/641	92.2	13	493	561/686	9.8	127.4
(6,21)	382	32/744	103.5	14	569	658/792	9.8	137.2
(8,11)	164	24/313	60	9	250	257/350	9.2	82.8
(8,13)	200	26/384	67.6	10	302	322/424	9.2	92
(8,15)	239	28/462	74.9	11	360	395/505	9.6	105.6
(8,17)	284	30/549	84.6	12	426	476/595	9.7	116.4
(8,19)	332	32/645	93.4	13	497	565/692	9.7	126.1
(8,21)	384	34/748	103.9	14	573	662/798	9.8	137.2
(8,23)	438	36/856	114.4	15	655	767/911	9.8	147
(8,25)	499	38/975	125.4	16	745	880/1033	10	160
(10,29)	638	44/1245	147	18	950	1134/1312	10.6	190.8
(10,39)	1036	54/2037	206.9	23	1543	1899/2148	11	253
(11,52)	1708	68/3369	300	29	2512	3140/3516	12.3	356.7

Table 3.23. Floating-point division implementation results for Virtex-II-4

There are many floating-point division designs available to compare with the results of the proposed design in table 3.23. In tables 3.24, 3.25, 3.26 and 3.27 the main results from the literature are presented and are grouped in terms of 4-bit, 6-bit, 8-bit and 11-bit exponent respectively. The 'Architecture' column of tables 3.24-3.27 describes the word format used, which is typically IEEE format although a couple of designs use a custom word format. Several different fixed-point divider architectures have been used, which include: bit-serial, reciprocal method with off chip ROM, a shorter word length reciprocal method using on chip ROM, a Newton-Raphson method and the R4 SRT method both maximally redundant and minimally redundant forms.

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles * delay	Area slices	Embed Mults	Chip maker	Year
Dido [133]	4	6	Recip ROM, custm word	none	none	none	yes	5	7.1 ns	35.5	80		Xilinx Virtex-6	02
Detrey [180]	4	7	SRT R4, IEEE word	RTNE	Inf, NaN	invalid	yes	8	10 ns	80	195		Xilinx VirtexII-4	03
Roesler [134]	4	11	Newt-raph, IEEE word	none	Inf, NaN	none	yes	13	6.5 ns	84.5	185	4	Xilinx VirtexII-6	02
Wang [135]	4	11	SRT R4, IEEE word	RTNE	Inf, NaN	(5) all IEEE	yes	29	5.3 ns	153.7	1096		Xilinx VirtexII	03

Table 3.24. Floating-point division implementation results for a 4-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles * delay	Area slices	Embed Mults	Chip maker	Year
Quixilica [176]	6	7	IEEE word	RTNE	Inf, NaN	(4) not exact	yes	11	5 ns	55	124		Xilinx Virtex-6	02
Shirazi [1557]	6	9	Recip ROM IEEE word	none	none	none	no	5	169 ns	845	152		Xilinx XC4010	95
Quixilica [176]	6	11	IEEE word	RTNE	Inf, NaN	(4) not exact	yes	15	5.5 ns	82.5	220		Xilinx Virtex-6	02
Lee [51]	6	13	SRT R4, IEEE word	4 IEEE modes	Inf, NaN	(4) not exact	yes	10	8.7 ns	87	294		Xilinx VirtexII-4	02
Detrey [180]	6	13	SRT R4, IEEE word	RTNE	Inf, NaN	invalid	yes	11	10 ns	110	438		Xilinx VirtexII-4	03
Quixilica [176]	6	15	IEEE word	RTNE	Inf, NaN	(4) not exact	yes	19	6 ns	114	348		Xilinx Virtex-6	02
Roesler [134]	6	16	Newt-raph, IEEE word	none	Inf, NaN	none	yes	17	6.7 ns	113.9	369	10	Xilinx VirtexII-6	02
Wang [135]	6	17	SRT R4, IEEE word	RTNE	Inf, NaN	(5) all IEEE	yes	38	5.4 ns	205.2	2032		Xilinx VirtexII	03
Quixilica [176]	6	19	IEEE word	RTNE	Inf, NaN	(4) not exact	yes	23	6.5 ns	149.5	512		Xilinx Virtex-6	02

Table 3.25. Floating-point division implementation results for a 6-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles * delay	Area slices	Embed Mults	Chip maker	Year
Detrey [180]	8	15	SRT R4, IEEE word	RTNE	Inf, NaN	invalid	yes	12	10 ns	120	546		Xilinx VirtexII-4	03
Lee [51]	8	23	SRT R4, IEEE word	4 IEEE modes	Inf, NaN	(4) not exact	yes	15	9.6 ns	144	670		Xilinx VirtexII-4	02
Detrey [180]	8	23	SRT R4, IEEE word	RTNE	Inf, NaN	invalid	yes	16	10 ns	160	1109		Xilinx VirtexII-4	03
Nallatech [175]	8	23	Custom word		Inf, NaN	Inval, ovfl	no	26	6.5 ns	169	730		Xilinx VirtexII-4	02
Quixilica [176]	8	23	IEEE word	RTNE	Inf, NaN	(4) not exact	yes	27	7 ns	189	711		Xilinx Virtex-6	02
Roesler [134]	8	23	Newt-raph, IEEE word	none	Inf, NaN	none	yes	29	7.1 ns	205.9	958	24	Xilinx VirtexII-6	02
Wang [135]	8	23	SRT R4, IEEE word	RTNE	Inf, NaN	(5) all IEEE	yes	47	5.6 ns	263.2	3245		Xilinx VirtexII	03
Digital core design [168]	8	23	IEEE word	RTNE	Denorm, inf, NaN	ov, un, invalid	no	15	19 ns	285	1534		Xilinx VirtexII-5	01
Novak [1155]	8	23	Bit serial, IEEE word	RTN	none	none	no				620		Actel 1280	94
Roesler [134]	8	32	Newt-raph, IEEE word	none	Inf, NaN	none	yes	29	7.1 ns	205.9	1192	24	Xilinx VirtexII-6	02

Table 3.26. Floating-point division implementation results for an 8-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles * delay	Area slices	Embed Mults	Chip maker	Year
Lee [40]	11	52	SRT R4, IEEE word	4 IEEE modes	Inf, NaN	(4) not exact	yes	29	12.2 ns	353.8	2595		Xilinx VirtexII-4	02
Paschalakis [48]	11	52	Bit-serial, IEEE word	RTNE	Inf, NaN	none	no	1		1000	343		Xilinx Virtex-6	03

Table 3.27. Floating-point division implementation results for an 11-bit exponent

3.9.4.1 Comparison of floating-point division results

To compare each set of results for a particular exponent width two graphs of area and delay are used. The graphs compare the results in the literature (tables 3.24-3.26) with the results in table 3.23. Figures 3.42, 3.44 and 3.46 show how the area of different 4, 6 and 8-bit exponent implementations vary with mantissa width. Figures 3.43, 3.45 and 3.47 show how the delay of different 4, 6 and 8-bit exponent implementations vary with mantissa width.

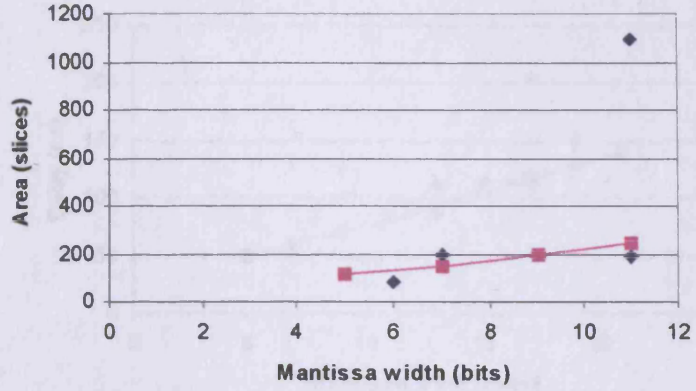


Figure 3.42. The area of the 4-bit exponent implementation of tables 3.23 [■] and 3.24 [♦]

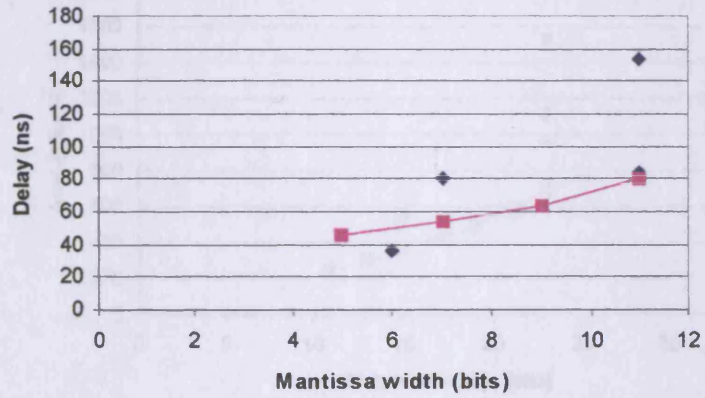


Figure 3.43. The delay of the 4-bit exponent implementation of tables 3.23 [■] and 3.24 [♦]

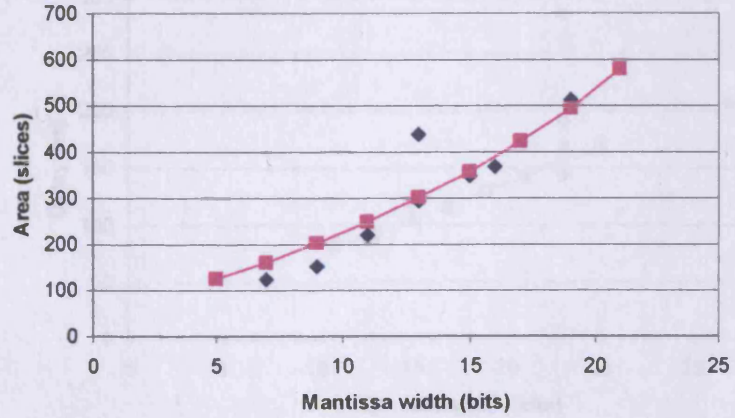


Figure 3.44. The area of the 6-bit exponent implementation of tables 3.23 [■] and 3.25 [♦]

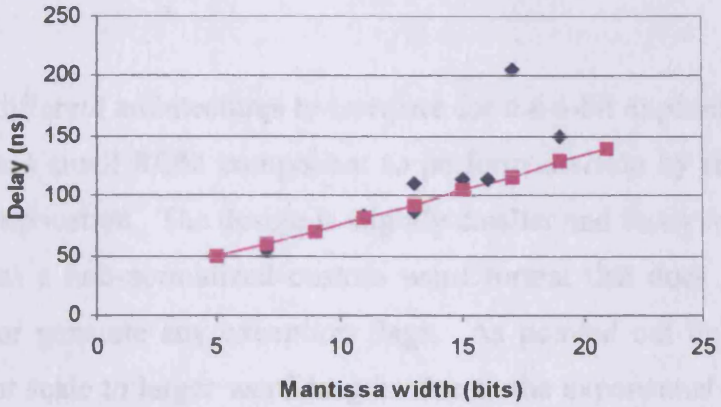


Figure 3.45. The delay of the 6-bit exponent implementation of tables 3.23 [■] and 3.25 [♦]

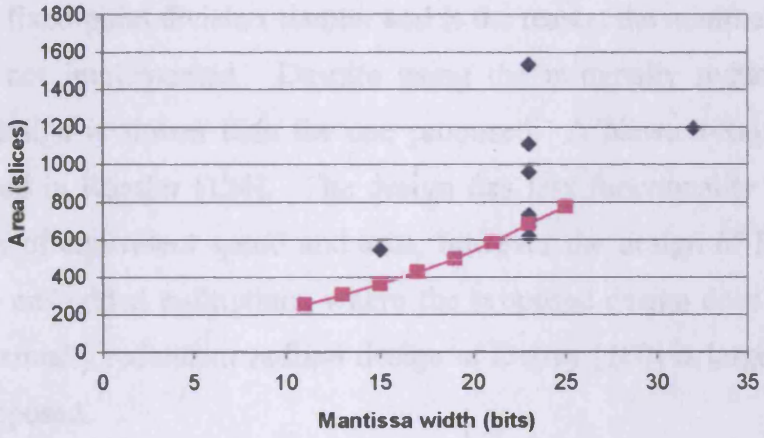


Figure 3.46. The area of the 8-bit exponent implementation of tables 3.23 [■] and 3.26 [♦]

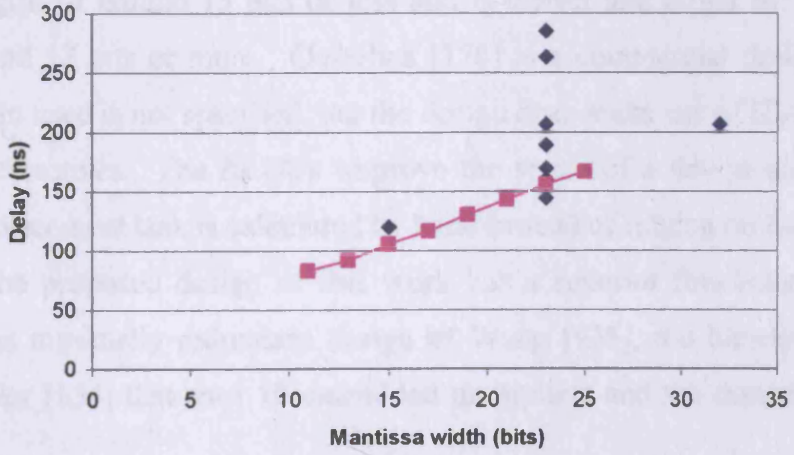


Figure 3.47. The delay of the 8-bit exponent implementation of tables 3.23 [■] and 3.26 [♦]

3.9.4.2 Result discussion

4-bit exponent

There are four different architectures to compare for the 4-bit exponent. The work of Dido [133] uses a small ROM component to perform division by reciprocation and subsequent multiplication. The design is slightly smaller and faster than the proposed design but it has a non-normalized custom word format that does not support any special values or generate any exception flags. As pointed out by the authors the method does not scale to larger word lengths due to the exponential area increase of the ROM. The design of Wang [135] uses a radix-4 minimally redundant SRT divider. The design is larger than the one proposed, which is probably due to the extra logic needed to perform the quotient digit selection function. This supports the theory from the fixed-point division chapter and is the reason the minimally redundant algorithm was not implemented. Despite using the minimally redundant radix-4 algorithm the design is slower than the one proposed. A Newton-Raphson divider algorithm is used in Roesler [134]. The design has less functionality than the one proposed and is of equivalent speed and area, however the design of Roesler [134] makes use of 4 embedded multipliers, where the proposed design does not use any. Finally, the maximally redundant radix-4 design of Detrey [180] is larger and slower than the one proposed.

6-bit exponent

The slightly reduced functionality design of Quixilica [176] is faster and smaller for short word lengths of around 13 bits or less and is slower and larger for long word lengths of around 17 bits or more. Quixilica [176] is a commercial design and the divider algorithm used is not specified, but the design does make use of RLOCs which are placement directives. The RLOCs improve the speed of a design and the slice packing as the placement task is calculated by hand instead of relying on the place and route tools. The proposed design of this work has a superior functionality and is smaller than the minimally redundant design of Wang [135], the Newton-Raphson design of Roesler [134] that uses 10 embedded multipliers and the design of Detrey [180].

8-bit exponent

Only the bit-serial design of Novak [155] is smaller than the proposed design for an 8-bit exponent. The proposed design is the fastest for any 8-bit exponent width. Only the design Digital Core Design [168] has more functionality than the proposed design as the concept of denormalized numbers is supported however this causes a significant area increase.

11-bit exponent

The reduced functionality design of Paschalakis [181] uses a bit-serial divider, which gives a very small design. However, due to the bit-serial divider implementation, the design is very slow and cannot be pipelined.

3.9.4.3 Conclusion

The smallest designs are produced using a bit-serial divider for the significand division. However these designs are also the slowest and cannot be pipelined. The Newton-Raphson division method has an equivalent slice area and speed to an SRT implementation but requires many embedded multipliers in its implementation when implemented as in Roesler [134]. The minimally redundant radix-4 SRT divider is much bigger than the maximally redundant version as the implementation of Wang [135] shows and as predicted in section 2.9.3. The proposed design is not the smallest or fastest for short word lengths because the commercial design of Quixilica [176] is superior, due to the use of placement directives. The proposed design is the fastest and the smallest for larger word length significands and has the most functionality of any other design apart from Digital Core Design [168], which includes support for denormalized numbers. Including support for denormalized numbers significantly increases the size of the design by around 100%. The reciprocal ROM method has good potential for very short significand word length operands and further study in its application to normalized division algorithms would be an interesting future work.

3.9.5 Floating-point square root results

Table 3.28 summarises the floating-point square root Xilinx Virtex-II-4 FPGA implementation results using version 5.1.03i of the Xilinx place and route tools and

speed definition files created on 01/11/2002. The Xilinx tools are used for the complete design flow from entry to FPGA configuration bit stream generation.

(e,m)	Area (slices)	Area ffs/LUTs	Delay (ns)	Pipeline stages	Area (slices)	Area ffs/LUTs	Delay (ns)	Stages* delay
(4,5)	43	12/74	21.1	5	65	64/92	7.1	35.5
(4,7)	56	14/100	26.6	6	82	89/117	7.2	43.2
(4,9)	73	16/132	33.3	7	105	119/148	7.4	51.8
(4,11)	90	18/167	39.2	8	130	154/182	7.6	60.8
(6,7)	59	16/105	26.7	6	88	93/125	7.3	43.8
(6,9)	76	18/137	33.4	7	110	123/155	7.4	51.8
(6,11)	93	20/172	39.7	8	135	158/189	7.6	60.8
(6,13)	116	22/213	47.6	9	165	198/229	7.9	71.1
(6,15)	135	24/254	57.5	10	197	243/271	8.1	81
(6,17)	160	26/302	67.4	11	233	293/318	8.3	91.3
(6,19)	187	28/355	77.5	12	273	348/368	8.6	103.2
(6,21)	216	30/411	87.8	13	316	408/423	8.9	115.7
(8,11)	95	22/175	39.9	8	139	162/194	7.6	60.8
(8,13)	118	24/216	47.7	9	169	202/234	7.9	71.1
(8,15)	137	26/257	58.9	10	201	247/276	8.2	82
(8,17)	162	28/305	67.4	11	237	297/323	8.4	92.4
(8,19)	189	30/358	77.5	12	277	352/373	8.6	103.2
(8,21)	218	32/414	88.3	13	320	412/428	8.9	115.7
(8,23)	248	34/473	95.6	14	366	477/486	9	126
(8,25)	281	36/537	105.8	15	417	547/549	9.3	139.5
(10,29)	355	42/681	126.6	17	533	706/693	9.9	168.3
(10,39)	566	52/1098	178.8	22	870	1181/1121	11.2	246.4
(11,52)	920	66/1792	253.3	28	1434	1959/1808	13	364

Table 3.28. Floating-point square root implementation results for Virtex-II-4

There are many floating-point square root designs available to compare with the results of the proposed design in table 3.28. In tables 3.29, 3.30, 3.31 and 3.32 the main results from the literature are presented and are grouped in terms of 4-bit, 6-bit, 8-bit and 11-bit exponent respectively. The architecture column of tables 3.29-3.32 describes the word format used, which is typically IEEE format although a couple of designs use a custom word format. Several different fixed-point square root architectures have been used, which include: bit-serial, radix-2 SRT, restoring and non-restoring.

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles* delay	Total delay	Area Slices	Chip maker	Year
Detrey [180]	4	7	SRT R2, IEEE word	RTNE	Inf, NaN	invalid	yes	7	10 ns	70	38 ns	84	Xilinx VirtexII-4	03
Wang [135]	4	11	Restore, IEEE word	RTNE	Inf, NaN	invalid, inexact	yes	16	4.9 ns	78.4		380	Xilinx VirtexII	03

Table 3.29. Floating-point square root implementation results for a 4-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles* delay	Total delay	Area Slices	Chip maker	Year
Lee [51]	6	13	Non-restor, IEEE word	4 IEEE modes	Inf, NaN	invalid	yes	9	8.4 ns	75.6	47 ns	153	Xilinx VirtexII-4	02
Detrey [180]	6	13	SRT R2, IEEE word	RTNE	Inf, NaN	invalid	yes	10	10 ns	100	62 ns	182	Xilinx VirtexII-4	03
Quixilica [176]	6	16	IEEE word	RTNE	Inf, NaN	Un, ov invalid	yes	19	5.8 ns	110.2		620	Xilinx Virtex-6	02
Wang [135]	6	17	Restore, IEEE word	RTNE	Inf, NaN	invalid, inexact	yes	22	5.4 ns	118.8		778	Xilinx VirtexII	03

Table 3.30. Floating-point square root implementation results for a 6-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles* delay	Total delay	Area Slices	Chip maker	Year
Detrey [180]	8	15	SRT R2, IEEE word	RTNE	Inf, NaN	invalid	yes	11	10 ns	110	70 ns	232	Xilinx VirtexII-4	03
Digital core design [168]	8	23	IEEE word		Denorm, Inf, NaN	(3) not inexact	no	9	13.2 ns	118.8		469	Xilinx VirtexII-5	
Lee [51]	8	23	Non-restor, IEEE word	4 IEEE modes	Inf, NaN	invalid	yes	14	9.6 ns	134.4	90 ns	357	Xilinx VirtexII-4	02
Detrey [180]	8	23	SRT R2, IEEE word	RTNE	Inf, NaN	invalid	yes	15	10 ns	150	106 ns	423	Xilinx VirtexII-4	03
Wang [135]	8	23	Restore, IEEE word	RTNE	Inf, NaN	invalid, inexact	yes	28	5.9 ns	165.2		1313	Xilinx VirtexII	03
Nallatech [175]	8	23	Custom word		Inf, NaN	Ovfl, invalid	no	29	6.6 ns	191.4		330	Xilinx VirtexII-4	02
Li [160]	8	23	Non-restor, IEEE word	none	none	none	no	15				408	Xilinx XC4000	97

Table 3.31. Floating-point square root implementation results for an 8-bit exponent

Author	e	m	Architecture	Rounding	Special values	Except flags	Param	Cycles	Cycle delay	Cycles* delay	Total delay	Area Slices	Chip maker	Year
Lee [51]	11	52	Non-restor, IEEE word	4 IEEE modes	Inf, NaN	invalid	yes	28	13.8 ns	386.4	239 ns	1433	Xilinx VirtexII-4	02
Paschalakis [181]	11	52	Bit-serial, IEEE word	RTNE	Inf, NaN	none	no	1		735	735 ns	347	Xilinx Virtex-6	03

Table 3.32. Floating-point square root implementation results for an 11-bit exponent

3.9.5.1 Comparison of floating-point square root results

To compare each set of results for a particular exponent width two graphs of area and delay are used. The graphs compare the results in the literature (tables 3.29-3.31) with the results in table 3.28. Figures 3.48, 3.50 and 3.52 show how the area of different 4, 6 and 8-bit exponent implementations vary with mantissa width. Figures 3.49, 3.51 and 3.53 show how the delay of different 4, 6 and 8-bit exponent implementations vary with mantissa width.

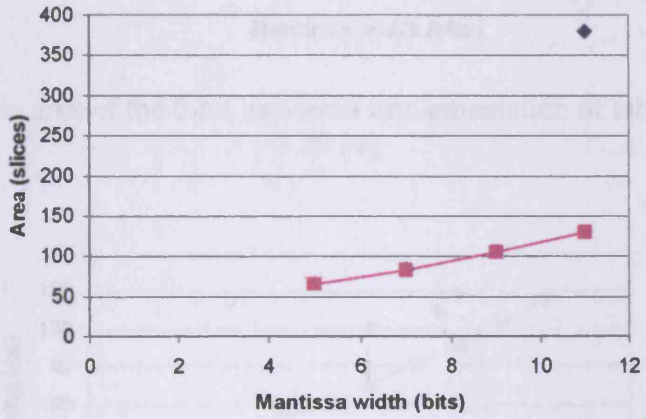


Figure 3.48. The area of the 4-bit exponent implementation of tables 3.28 [■] and 3.29 [♦]

Figure 3.49. The delay of the 4-bit exponent implementation of tables 3.28 [■] and 3.29 [♦]

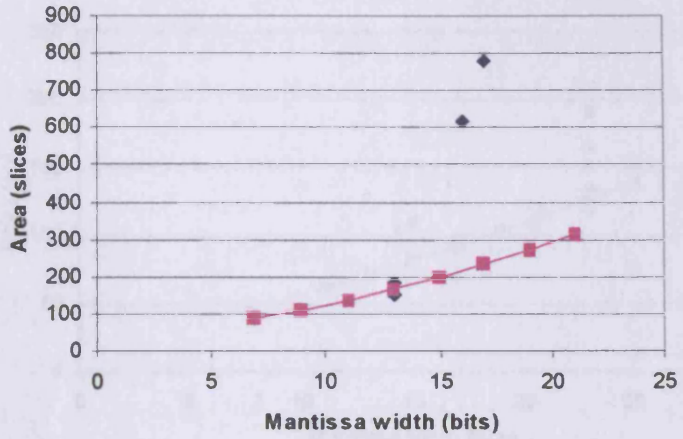


Figure 3.50. The area of the 6-bit exponent implementation of tables 3.28 [■] and 3.30 [♦]

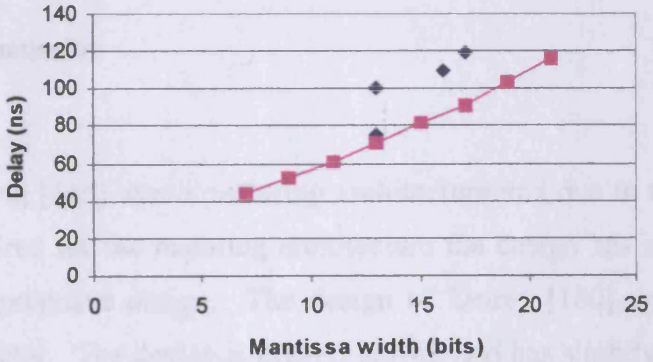


Figure 3.51. The delay of the 6-bit exponent implementation of tables 3.28 [■] and 3.30 [♦]

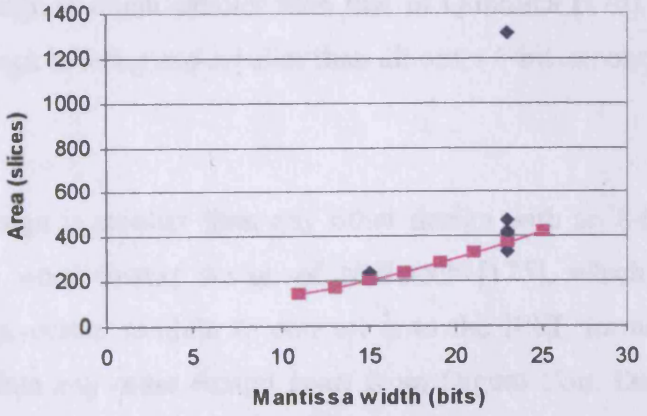


Figure 3.52. The area of the 8-bit exponent implementation of tables 3.28 [■] and 3.31 [♦]

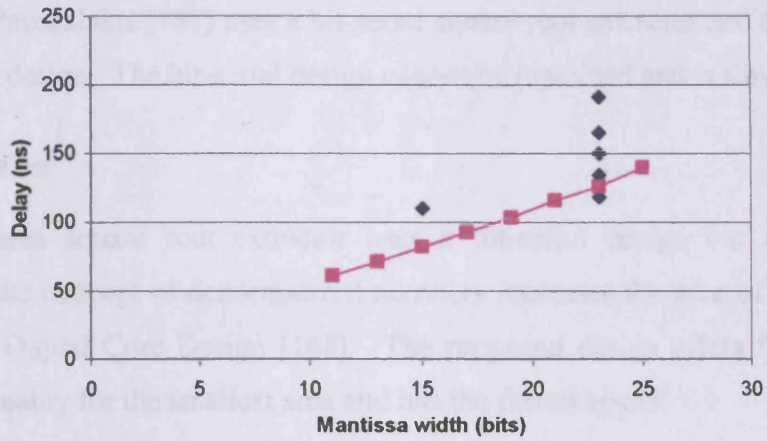


Figure 3.53. The delay of the 8-bit exponent implementation of tables 3.28 [■] and 3.31 [♦]

3.9.5.2 Result discussion

4-bit exponent

The work of Wang [135] uses a restoring architecture and due to the extra selection multiplexer required for the restoring architecture the design has a large area and is slower than the proposed design. The design of Detrey [180] uses a radix-2 SRT square root extractor. The design is bigger, slower and has slightly less functionality than the proposed design.

6-bit exponent

The proposed design is much smaller than that of Quixilica [176] and is also faster. The proposed design is faster and smaller than all other 6-bit exponent designs.

8-bit exponent

The proposed design is smaller than any other design with an 8-bit exponent apart from the custom word format design of Nallatech [175], which the authors state would need a conversion module to convert it to the IEEE format. The proposed design is faster than any other design apart from Digital Core Design [168], which uses a faster speed grade chip. The proposed design has more functionality than any other design apart from Digital Core Design [168], which implements the concept of denormalized numbers.

11-bit exponent

The design of Paschalakis [181] uses a bit-serial square root extractor and therefore is a very compact design. The bit-serial design cannot be pipelined and is slow.

3.9.5.3 Conclusion

The smallest area square root extractor uses a bit-serial design but it is slow. Implementing the concept of denormalized numbers increases the area of the design by about 50% Digital Core Design [168]. The proposed design offers the greatest level of functionality for the smallest area and has the fastest speed.

Chapter 4

Function evaluation

In the next chapter the logarithmic number system will be discussed and it will be seen that a highly non-linear complicated function needs to be approximated. In this section we will consider the implementation of FPGA function approximation units and compare them with existing methods given in the literature.

4.1 Common function approximation methods

4.1.1 Full table lookup

This is the most basic approximation method where for every value of x the value of the function, $f(x)$, at that point is stored in a table. This is in effect a memory that is addressed with the full function argument x. The size of the memory increases exponentially with address width so this scheme is only practical (for FPGA) for small word lengths of up to 7 or 8-bits. Two advantages to this method are: firstly it is generally fast because there aren't any arithmetic components and secondly the values in the memory can be correctly rounded to within ½ ulp.

4.1.2 Bipartite, SBTM, STAM and multipartite

The bipartite Sarma [183], SBTM Schulte [184], STAM Schulte [185] and multipartite Dinechin [186] methods are based on a first order Taylor series approximation and use only table lookups and additions. The bipartite method uses two tables each with an address width of approximately 2/3 * input_operand_width, which are addressed by parts of the input operand x. A single adder is used to add the values obtained from the two tables and this value is the approximation to the function $f(x)$. The size of the tables increase exponentially with argument width but due to the 2/3 scale factor the combined size of the two tables is smaller than the full table lookup. If the output can be used in carry-save form then no addition is required. The

bipartite method is really only suitable (for FPGA) for arguments of less than 12 bits. The STAM and multipartite methods increase the number of tables but reduce the maximum number of address bits and content bits a table requires. The maximum address width for the multipartite method is generally 1/2 * input_operand_width so the method is only suitable (for FPGA) for arguments of up to 16 bits. The method is slower than the bipartite and full table lookup because now a number of adders are required to add the lookup table outputs to generate the final function approximation $f(x)$. The bipartite and multipartite methods cannot return correctly rounded results without requiring very large tables and so the results are guaranteed to be faithfully rounded to 1 ulp. The major plus point of the bipartite, STAM and multipartite methods is that no multiplications are required. Other lookup-add methods have been proposed in Hassler [187], Wong [188], Lo [189] and Wan [190], but these methods require very large ROMs and so are prohibitive for FPGA implementation.

4.1.3 Polynomial approximation

The polynomial approximation is the broadest class of function approximation with many subsets. The major ones will be discussed in turn.

4.1.3.1 Taylor and Maclaurin series

The Taylor series, Jeffery [58] and Spiegel [191] generates an approximation to a specific function at a particular point and this will typically hold for a small region around that point and so the generated series can be used to approximate a function over a small range. If the series is generated at the origin then it is called the Maclaurin series. The Taylor series is a fixed coefficient series and does not need any ROM components to store function approximation values. However, many multiplications and additions are needed to calculate a very accurate approximation.

4.1.3.2 Approximation via a single polynomial

Typically a function needs to be evaluated for a given interval and as the Taylor series only approximates for a single point the error of such a series can be quite high and non-uniform. A better method is to fit a polynomial of degree N to the function so it coincides with the function at N+1 various points. The Lagrange approximation technique Jeffery [58], Noetzel [193] returns the equation for a curve that passes through a number of given points. The Lagrange technique fits a curve to points

rather than to a given function so in certain circumstances the error of the method can be quite high. The selection of the points can be crucial and Schulte [198] chooses Chebyshev nodes and interpolates them using the Lagrange technique. Fitting a polynomial so it has the criteria of having the minimal maximum error is called the minimax approximation Muller [195] and it is the most optimum polynomial approximation technique. The minimax approximation involves searching for optimum coefficients, which can be done via the Remez algorithm [193]. The Maple software package Maple [196] uses the Remez algorithm to calculate the optimal coefficients. Pineiro [194] proposes a faithful polynomial approximation technique using the minimax approximation with the coefficients developed by the Maple [196] package. The Chebyshev approximation Press [197] is very close to the optimal approximation technique and the coefficients can be quickly generated with mathematical formulae, whereas the minimax polynomial coefficients need to be searched for. A function that is highly non-linear will typically need a high degree of polynomial to approximate it accurately with a single polynomial. This implies that a large number of additions and multiplications are required.

4.1.3.3 Piecewise approximation

To reduce the required degree of polynomial to approximate a given interval the interval can be split up into several smaller intervals and each can be approximated with a single polynomial of reduced degree. The approximation of an interval by several polynomials is known as piecewise approximation. The piecewise approach trades arithmetic components such as multipliers and adders for coefficient ROM and striking the right balance is a very important design goal. The interval to approximate can be split into uniform sections and non-uniform sections. Optimal partitions are obtained using non-uniform splits but this causes difficulty in selecting the correct coefficients because they will typically be stored in a ROM with a uniform addressing scheme. A uniform split allows direct addressing of the coefficient ROM.

4.1.3.4 Rational approximation

To reduce the required degree of polynomial to approximate a given function a rational approximation can be used Muller [195] and Koren [199]. A rational approximation is like a polynomial fraction and has numerator and denominator parts. The degree of the numerator and denominator parts is usually less than that of a

standard polynomial that can approximate a given function interval with the same error. The major draw back with the rational approximation is that a division must take place and this is a very costly operation that we would like to avoid. The numerator and denominator part of a rational approximation are usually developed to have the same complexity (order and number of terms) so that both computation paths (numerator and denominator) arrive at the divider at the same time.

4.1.4 CORDIC

CORDIC, Volder [200] and Walther [201], is a linearly convergent shift-and-add algorithm that can be used to calculate many trigonometric functions as well as certain other functions. In general the algorithm requires two or three values, which are updated each iteration by their own addition/subtraction component. The updating addition/subtraction is controlled each iteration by the sign of one of the values. Another one of the values is updated each iteration by adding or subtracting a constant which is stored in a ROM and changes for each iteration. One or two of the values will converge to a desired result for example in the calculation of the sine and cosine functions there are three values. One value converges to sine, one to cosine and the other value is rotated to zero. For the basic implementation a single bit of the result is generated per iteration and the final result is wrong by a constant scale factor that needs to be removed by using a final multiplication.

4.1.5 Other linear convergence algorithms

Two classes of algorithm that are very similar to the CORDIC algorithms are the multiplicative and additive normalization algorithms Ercegovac [61]. In the multiplicative normalisation algorithm there are two sequences. An auxiliary sequence is used that determines a digit value. The digit value directly updates the auxiliary sequence on the next iteration. This digit value is also used to address a ROM that updates the primary sequence so that it converges to the required function. The additive normalization algorithm works in a similar, but opposite way. Here the auxiliary sequence is used to determine a digit value. The digit value addresses a ROM and updates the auxiliary sequence in the subsequent iteration. The digit value is also used to directly update the primary sequence so that it converges to the required function.

4.2 An overview of FPGA function approximation methods

CORDIC is by far the most popular function approximation method implemented on FPGA Andraka [211, 214], Dick [212, 217], Park [213], Mencer [215], Ligon [216], Kantabutra [218], Valls [220], Paplinski [221], Kharrat [223], Zhilu [225], Cardells-Tormo [226, 237], Lund [227], Vadlamani [229], Yang [232] and Ravichandran [233] are a few examples. Traditional and current FPGA architectures consist of registers and logic that can build efficient adders and small memories. These are exactly the logic elements that a CORDIC implementation requires. The CORDIC algorithms became popular for two reasons: firstly they can perform trigonometric functions, which are required in many complex-arithmetic DSP algorithms and secondly because they can be built in a very compact way using bit-serial or digit-serial arithmetic. The bit-serial algorithms are compact, they have local interconnects so can run at a high clock rate, and are very efficient if data are supplied in a serial fashion. Fully parallel CORDIC algorithms have also been proposed, due to the size increase of modern FPGA, in an attempt to increase the throughput of the CORDIC algorithm. The problem with the CORDIC algorithm is the dependency of the next iteration on the result of the previous iteration and so it is hard to introduce parallelism into the algorithm. Some attempts have been made to speed up the basic CORDIC algorithm by using signed-digits Valls [220] and high radix algorithms Kantabutra [218] and Vadlamani [229] but they turn out to require more hardware and are slower than the basic design. Boullis [219] presents a linear piecewise approximation of the natural logarithm function for the interval [1, 2). Boullis shows that the approximation requires fewer clock cycles than a CORDIC algorithm and is smaller than a fully unrolled CORDIC implementation for small operand lengths of up to 14-bits. For very short operand lengths of less than 8-bits the results show that full table lookup is the most efficient implementation. Mencer [222] compares four different function approximation methods of full table lookup, bipartite, linear piecewise approximation and CORDIC. The results are the same as presented in Boullis [219] except that bipartite results are included. The bipartite method does not offer an area advantage over the linear piecewise approximation but it does require fewer clock cycles. It must be noted that only the clock cycle number and not the clock frequency is given in Boullis [219] and Mencer [222], so a fair comparison is difficult. Pineiro [224] proposes a second order piecewise approximation of a powering function using the

minimax polynomial approximation. The polynomial is evaluated using a fused MAC tree structure, which calculates all the multiplier partial products in parallel and sums them with a single carry-save adder tree. The design is capable of being modified to calculate many different functions of restricted range and all faithfully rounded to 23-bit single precision. The design is much smaller than the 24-bit linear piecewise design of Mencer [222] and has a low latency. Sidahao [231] looks at the speed-area trade offs in implementing a single minimax polynomial approximation of a given function over a restricted range. The authors use a multiply accumulate function, which is fed with a ROM of coefficients, and an additional feedback loop to evaluate an N^{th}-degree polynomial. The number of multiply accumulate stages can be changed to vary the hardware requirement and also to vary the speed of the design. The design is compared to the bipartite design, which from Mencer [222] is known not to be the smallest FPGA function approximation algorithm but it does have a low delay. Due to the fact that a large polynomial is being evaluated in a recursive manner the proposed design is much slower than the bipartite method by between 3 to 8 times. Now, a function is being approximated by a single polynomial and when a greater accuracy is needed the coefficients of the polynomial increase in bit width and the degree of polynomial might also be required to increase. These factors that increase clearly do not cause the logic requirement to exponentially increase and so the area increase of the polynomial approximation with word size is much slower when compared to the bipartite method. The size of the bipartite method increases exponentially as the width of the function to approximate increases, which is due to the ROM size increase. Interestingly, for small word lengths of around 14-bits the bipartite method is much faster and of equivalent size to the design presented by Sidahao [231]. An FPGA implementation of the multipartite method is proposed by Dinechin [228] and subsequently improved by Detrey [230] using a table content optimising algorithm. Similar to the bipartite method an exponential memory increase is observed, but the table sizes required are smaller than the bipartite method. The results also show that the delay of the multipartite algorithm is very small and is smaller than any other published design considered in this overview. More recently a piecewise polynomial scheme has been proposed by Lee [236] that has a novel segmentation scheme. The novel scheme allows a non-uniform split of a function so that each segment can be approximated by a different polynomial and all the coefficients can be stored in a regular ROM addressed by the novel segmentation

scheme. The idea of the scheme is to allow highly non-linear functions to be approximated efficiently so that each segment produces an approximation to the function with roughly the same accuracy. The scheme is compared to a uniform segmented design and the multipartite methods. The design is reported to require a much smaller amount of memory than the uniform methods and multipartite methods. Speed comparisons are not made, as results given for the multipartite and uniform segmented methods are only theoretical. The design cannot improve on the classical uniform approach when a smooth function with restricted range is to be approximated e.g. the natural logarithm function on the interval [1, 2). Ho [235] uses the STAM lookup-add approximation method to approximate the function $x^{-3/2}$ with 16-bits of accuracy. The design requires 32 of the blockRAM components in a single Virtex FPGA and three adders. The design is constructed in such a way that is uses the same amount of logic as the bipartite method, which is due to the coarse granularity of the blockRAM components. For the STAM method to improve on the bipartite method the lookup tables must be implemented with custom size memories. The blockRAM resource is of fixed size and so it cannot take advantage of the STAM method. A possible solution is to use the LUT primitives to build the memories, as these primitives are much finer in granularity. The STAM method is less superior than the multipartite method proposed by Dinechin [228] and the work of Ho [235] illustrates this fact.

4.3 New piecewise Taylor series approximation design

As will be seen in the next section a scheme to approximate different functions is needed to evaluate various implementations of the logarithmic number system addition/subtraction function. The two 'best' FPGA function approximation methods in the literature are: the multipartite method of Dinechin [228], which offers a good trade off between area and speed, but does suffer from an exponential memory increase because it is predominantly a first order approximation; the polynomial approximation scheme of Sidahao [231] is another 'best' design because of its area efficiency for large word lengths. However, there is a speed penalty for the reduced area. These methods are fast and large or slow and small so are not really suitable for the application required. It was decided that an alternative method should be sought. In this section the design of a piecewise Taylor series function approximation unit is

described and is then compared to other designs available in the open literature. The Taylor series function is chosen because of its attrctive decomposition properties, which is why it is used in the multipartite, STAM and bipartite methods. A uniform partitioning scheme is also used to simplify the hardware addressing.

4.3.1 Taylor series

The Taylor series is used to approximate a function around a specific point. Consider the value x where x is split into two parts x_0 and x_1 as shown in figure 4.1.

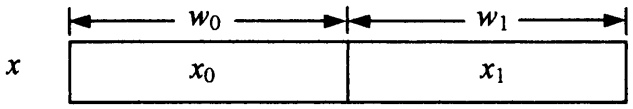


Figure 4.1. The two part split of the argument x

The Taylor series can be used to approximate arbitrary values of a function $f(x)$ by generating an approximation that is centred around x_0. Equation (4.1) shows the Taylor series approximation of a function $f(x)$ centred around a point x_0.

$$f(x) = f(x_0) + [x - x_0]f'(x_0) + \frac{[x - x_0]^2 f''(x_0)}{2!} + \ldots + \frac{[x - x_0]^{n-1} f^{n-1}(x_0)}{(n-1)!} + \varepsilon_n \quad \text{(4.1)}$$

In equation (4.1) the value of x_1 can be substituted for $[x\text{-}x_0]$ (4.2).

$$x_1 = [x - x_0] \quad \text{(4.2)}$$

The error committed in omitting high order terms, ε_n, is given by (4.3).

$$\varepsilon_n = \frac{MAX(f^n(\xi)[x - x_0]^n)}{n!} \quad \text{(4.3)}$$

From equation (4.3), for a fixed level of accuracy, it can be seen that a trade off between the split of the input operand and the order of approximation can be made. That is, if the order of approximation increases and the width of x_0 is held constant then the accuracy of the approximation will increase. However if the order of approximation is held constant and the width of x_0 increases then the accuracy of the approximation will also increase. So to create a more accurate design what should be increased: the order, which increases the arithmetic requirement, or the width of x_0, which increases the ROM requirement. Looking at the logic requirement to construct

arithmetic components such as multipliers and adders, and also looking at the logic required to construct ROMs should give some design guidance.

4.3.2 Arithmetic and ROM component logic requirements

FPGAs offer distributed memory to create arbitrary size lookup tables and provide dedicated logic to perform efficient multiplication and addition. The distributed memory on Virtex FPGAs is constructed from the LUT primitives as shown in Xilinx [35]. The graph in figure 4.2 shows a 2^N (exponential) increase in the LUT requirement as the address width for a 1-bit content memory grows (Xilinx Virtex FPGA). The graph in figure 4.3 illustrates the LUT increase for Virtex multiplier implementations that do not use embedded multiplier primitives and shows an N^2 (squared) increase. Also shown on figure 4.3 is the LUT usage for the basic adder component, which has an order N increase (linear).

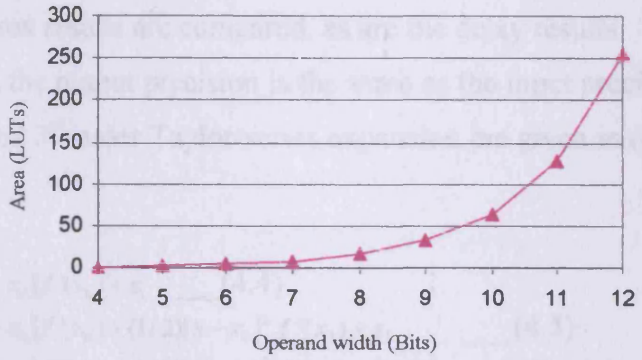


Figure 4.2. LUT requirement of a Virtex 1-bit content memory with and 'Operand width' address width

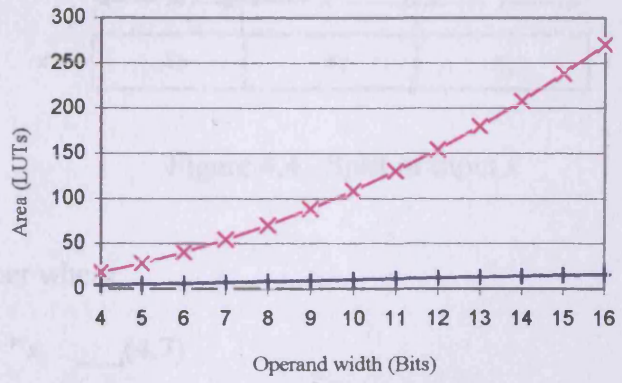


Figure 4.3. LUT requirement of an 'Operand width' by 'Operand width' multiplier [x] and adder [+]

The graph in figure 4.2 illustrates that for memories with large address and content widths the LUT quantity required to implement the ROM is too great to be practical even on current FPGA technology. Remember that the results of figure 4.2 are for a 1-bit content memory. A 12-bit address, 12-bit content memory would require 1500 slices, which is 30% of a Xilinx XC2V1000 1-million gate FPGA. In theory, calculating the Taylor series approximation by performing some of the required arithmetic and using a reduced amount of lookup should reduce the logic requirement while keeping the speed a lookup style approximation allows. The theory is naturally achieved by increasing the order of approximation.

4.3.3 1st, 2nd and 3rd order details

Three orders of approximation are considered from 1st to 3rd. Orders above the third are not considered due to the large delay that is required in evaluating the polynomial arithmetic. The three orders of approximation are implemented for varying precision inputs and the area results are compared, as are the delay results. The components are designed so that the output precision is the same as the input precision. The equations for the 1st, 2nd and 3rd order Taylor-series expansion are given in (4.4), (4.5), and (4.6) respectively.

$$f(x) = f(x_0) + [x - x_0]f'(x_0) + \varepsilon_1 \quad (4.4)$$

$$f(x) = f(x_0) + [x - x_0]f'(x_0) + (1/2)[x - x_0]^2 f''(x_0) + \varepsilon_2 \quad (4.5)$$

$$f(x) = f(x_0) + [x - x_0]f'(x_0) + (1/2)[x - x_0]^2 f''(x_0) + (1/6)[x - x_0]^3 f'''(x_0) + \varepsilon_3 \quad (4.6)$$

The input operand x is split as shown in figure 4.4.

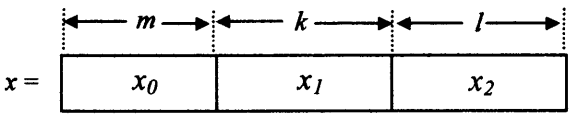


Figure 4.4. Split of input x

x is a d-bit number where

$$x = x_0 + 2^{-m}x_1 + 2^{-k-m}x_2 \quad (4.7)$$

For a first order approximation l is chosen as '0' and m is chosen so that $2m >= d$. Two ROMs are required

$$A(x_0) = f(x_0) \quad \text{(4.8)}$$
$$B(x_0) = f'(x_0)2^{-m} \quad \text{(4.9)}$$

The function $f(x)$ thus requires two lookups, one addition and one multiplication. It is given as

$$f(x) = A(x_0) + B(x_0)[x - x_0] + \varepsilon_1 \quad \text{(4.10)}$$

For the second and third order approximations m is chosen so that $3m >= d$ and $4m >= d$ respectively. k is chosen so that $2m+k >= d$ and $3m+k >= d$. The 4 ROMs required for the second order series are given in (4.11), (4.12), (4.13) and (4.14).

$$A(x_0) = f(x_0) \quad \text{(4.11)}$$
$$B(x_0) = f'(x_0)2^{-m} \quad \text{(4.12)}$$
$$C(x_0) = (1/2)f''(x_0)2^{-2m} \quad \text{(4.13)}$$
$$D(x_1) = x_1^2 \quad \text{(4.14)}$$

The function $f(x)$ requires four lookups, two additions and two multiplications. It is given as

$$f(x) = A(x_0) + [x - x_0]B(x_0) + D(x_1)C(x_0) + \varepsilon_2 \quad \text{(4.15)}$$

The following 5 ROMs are required for the third order approximation

$$A(x_0) = f(x_0) \quad \text{(4.16)}$$
$$B(x_0) = f'(x_0)2^{-m} \quad \text{(4.17)}$$
$$C(x_0) = (1/2)f''(x_0)2^{-2m} \quad \text{(4.18)}$$
$$D(x_0) = (1/6)f'''(x_0)2^{-3m} \quad \text{(4.19)}$$
$$E(x_1) = x_1^3 \quad \text{(4.20)}$$

The function $f(x)$ requires five lookups, three additions and three multiplications. It is given as

$$f(x) = A(x_0) + ([x - x_0]C(x_0) + B(x_0))[x - x_0] + E(x_1)D(x_0) + \varepsilon_3 \quad \text{(4.21)}$$

For all orders of Taylor series the relation between m, k and d is function dependant, however the values given are representative of the values used in practice.

4.3.4 Approximation error and rounding

All approximations are calculated with an error of less than 1 ulp to the true result. This is called faithful rounding and is the most practical rounding method for table-

based methods. To achieve faithful rounding, values are calculated with an internal precision that is greater than the target accuracy. Typically g guard-bits are used to guard against a loss of precision. The ROM that contains $f(x_0)$ is modified so the output rounding (i.e. reduction from the internal precision to the target precision) is truncation and also so the rounding of several internal multipliers can be performed by truncation. This keeps the width of the final adder-tree additions at a minimum.

4.3.5 Hardware structure

The hardware models of the 1st, 2nd and 3rd order approximations are shown in figures 4.5, 4.6 and 4.7 respectively.

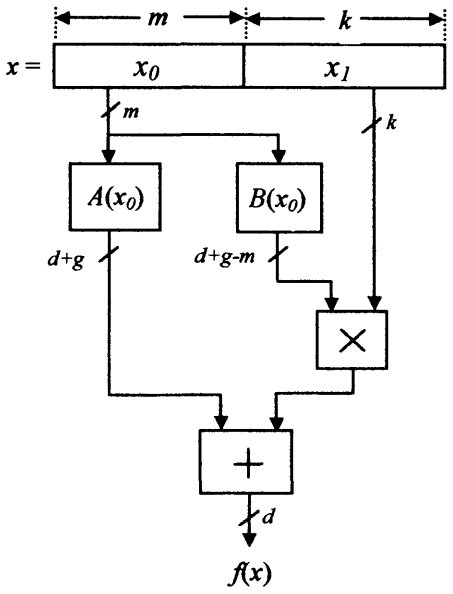


Figure 4.5. Hardware structure of the 1st order approximation

4.3.6 Model creation and testing

The software models of the 1st, 2nd, and 3rd order Taylor series have been developed using the MATLAB mathematical software package. The models for different word length operands are exhaustively tested for accuracy by enumeration to guarantee the faithful rounding criteria and to choose the guard bit quantity and operand split lengths. To implement the hardware method for FPGA a VHDL framework has been written. MATLAB is used to create the data to fill up the lookup tables in the framework and to generate bit-true results to compare against the hardware model. The width of the table contents depends on the values of the derivatives of the function being approximated and does vary from function to function although only

slightly for the common functions of $\log_2(x)$, $\sin(x)$, 2^x and e^x that all have well behaved derivatives.

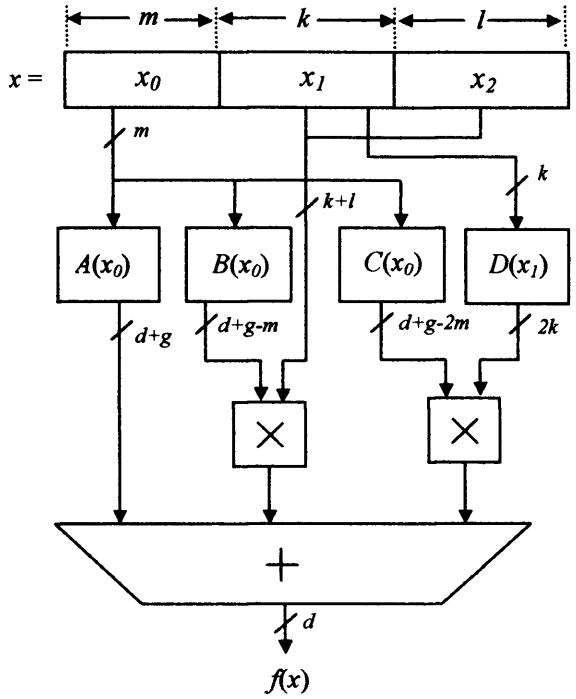


Figure 4.6. Hardware structure of the 2nd order approximation

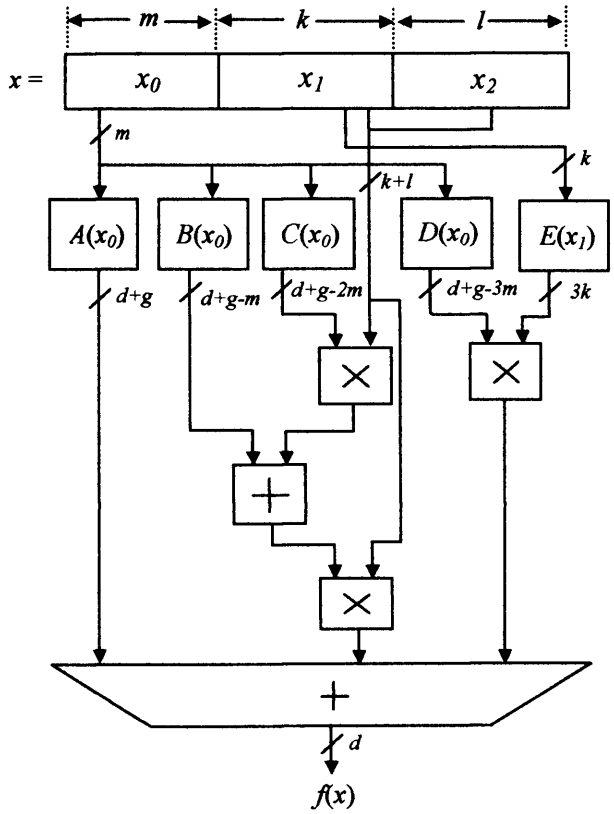


Figure 4.7. Hardware structure of the 3rd order approximation

4.3.7 Results

4.3.7.1 An implementation using only LUTs

To enable a comparison with other works and to demonstrate the adaptability of the design, results are presented for the sine(x) function for the domain $[0,\pi/4]$. The derivatives of the sine(x) function are either ±cosine(x) or ±sine(x), which gives a strict bound on the derivative of the function and is typical of the type of function suited to the Taylor series approximation. The graph in figure 4.8 illustrates the LUT requirement for the 1st, 2nd and 3rd order Taylor series implementations of different operand lengths on a Xilinx XCV1000-6 FPGA implemented using Xilinx ISE 5.1.03i.

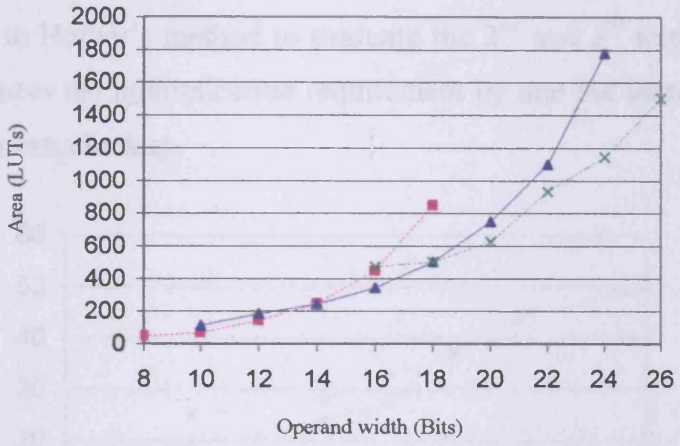


Figure 4.8. The LUT requirement of some 1st [■], 2nd [▲] and 3rd [x] order Taylor-series approximations of sine for varying input word length

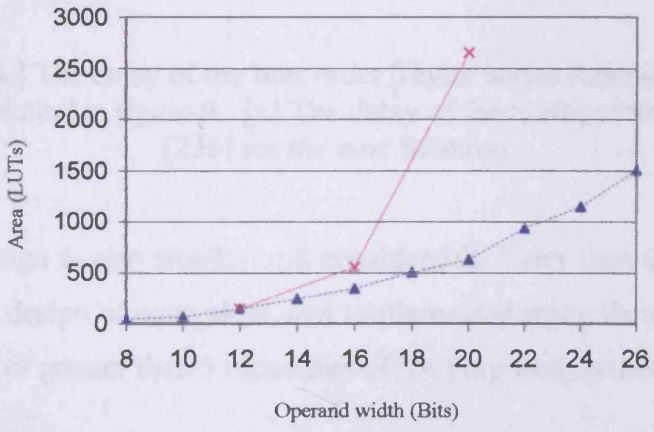


Figure 4.9. [x] The LUT requirement of the multipartite approximation method of Detrey [230] for sine; [▲] The LUT requirement of the proposed method for sine

4.3.7.2 Embedded multiplier and LUT implementation results

In this subsection the multiplication scheme developed in section 2.7.7.3.4 is used in the polynomial evaluation. The first order approximation uses one multiplier to reduce the logic content while the second and third order approximations use two. The graph in figure 4.11 illustrates the LUT requirements for the 1st, 2nd and 3rd order approximations that use the embedded multipliers. Figure 4.11 in effect illustrates the 'lookup-logic' required for each implementation. There is an exponentially increasing logic requirement for all orders but the point at which the logic requirement starts to dramatically increase occurs at a larger operand width for higher order approximations. As a percentage it takes just 6% of the LUT and multiplier resources of an XC2V1000 FPGA to implement a 23-bit sine(x) approximation for the range $[0, \pi/4]$.

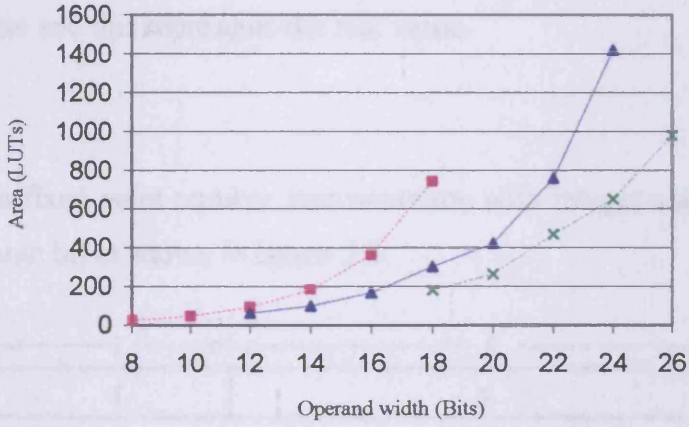


Figure 4.11. The LUT requirement of some 1st [■], 2nd [▲] and 3rd [x] order Taylor series approximations with varying word length.

4.3.8 Conclusion

In this chapter it has been demonstrated that the proposed piecewise polynomial approach is the most efficient method of approximating smooth functions of restricted range and domain on FPGA. FPGAs contain logic that can be configured to perform arithmetic functions and act as lookup tables. Varying the order of a piecewise polynomial approximation strikes a balance between the two logic configurations for a given level of accuracy that with straight forward analysis can be tuned to achieve an optimal design balance. The designs presented here are the fastest and smallest function approximation designs available in the open literature.

Chapter 5

Logarithmic number system

In the LNS (Logarithmic Number System) arithmetic is performed with logarithmic values. To convert values from the real domain into the LNS domain the log of the values is taken. The conversion logarithm function can be to any base but because computers natively use binary arithmetic we will only consider the base-2 case. To convert values from the LNS domain into the real domain they must be set as the power of the base and this represents the real value.

5.1 Format

The LNS uses a fixed-point number representation with integer and fraction sections and a separate sign bit as shown in figure 5.1.

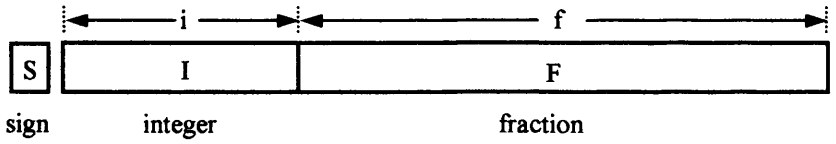


Figure 5.1. The three fields of a logarithmic number system operand

The integer and fraction section make up a signed two's complement number, which is the magnitude of the logarithmic number. This number needs to be signed to represent numbers less than 1. The separate sign bit represents the true sign of the value. The three field system is very similar to the floating-point format discussed in section 3.4.1. Zero cannot be represented in the LNS as there is no value a base can be raised by to obtain a result of zero, therefore zero is treated as a special case.

5.2 An LNS literature overview

Kingsbury [238] proposes the use of logarithmic arithmetic to perform digital filtering because simulation showed that a 16-bit implementation of a filter using logarithmic arithmetic could have a much better dynamic range than a 16-bit fixed-point implementation. Three methods of calculating the LNS addition/subtraction function are proposed: the first method uses a direct table lookup, the second uses interpolation to reduce the memory requirement and the third method, which is known as the *direct method* involves sequentially calculating the two functions which make up the composite addition and subtraction functions. No error loss is mentioned in calculating the subtraction function via the direct method although such an error does occur. Swartzlander [239] gives a more detailed overview of the construction of a sign/logarithm unit. In contrast to many current LNS implementations a biasing technique is used to ensure that the magnitude part of a logarithmic value is always a positive value. The magnitude of a fixed-point value is multiplied by a bias before it is converted into the number system and this prevents the log of values less than 1 from being taken. Frey [246] uses three different techniques to reduce the table size of the LNS addition and subtraction functions. The first technique as reported by Edgar [240] exploits the fact that for addition/subtraction function arguments above a certain size (magnitude) the value of the function quantizes to zero. This feature, which is termed *essential zeros* means that a table look up is not required for arguments above a certain size as the result is always zero. A second exploited property is that: for every integer value increase in the argument of the function the width of the value of the function to store decreases by one bit. This is due to the reduction in the number of significant bits in the function so the MSBs are zero invariant and need not be stored. A final logic reduction technique used is to split the addition and subtraction tables into two ROMs that can be optimised individually. Henkel [252] uses a linear interpolation technique to approximate the LNS addition function. An optimal number of non-uniform segments are calculated to achieve a given error bound. Considerable savings over previous table sizes are reported but at the expense of a linear interpolation and special segment selection logic (due to the non-uniform segmentation). Arnold [253] introduces the concept of the dual redundant logarithm number system where a system number consists of two unsigned fixed-point LNS values. One value is considered to have a negative weighting and the

other a positive weighting. This way addition involves updating the positive value and subtraction involves updating the negative value, where both operations are done via addition, thus avoiding the problematic subtraction function. However, one foreseeable problem is if a number close to zero ends up being represented by the difference of two very large values. In this circumstance the subsequent overflow of one value, which could easily occur, would return an undefined result when the result could be (well) in the range of the number system. Multiplication is slightly more complex in the proposed number system and division and square root even more so. Arnold [253] also proposes that the accuracy in the singularity region of the subtraction function can be relaxed. Lewis [254] uses a modified first order Taylor series approximation to approximate the addition and subtraction functions in a 31-bit (1-bit sign, 8-bit integer, 22-bit fraction) system. Logarithmic arithmetic is used in the function evaluation to avoid the multiplication and convert it into an addition. Using logarithmic arithmetic for the multiplication means the log must be taken of both inputs and the output must be converted by passing it through an exponential table function. The derivative table is removed by using a mathematical identity as first suggested by Arnold [242] and is in effect swapped for logarithm and exponential tables of smaller size. Huang [259] implements the first 32-bit design (1-bit sign, 8-bit integer, 23-bit fraction equivalent to IEEE std-754 single precision) by using the direct method to evaluate the LNS addition and subtraction functions. No mention is made of the loss of accuracy for the subtraction function in the singularity region due to using this method. A set of guidelines is proposed by Arnold [257] to standardise the LNS word format, as was done for floating-point by the IEEE std-754. The encodings for special values are proposed so the LNS has an equivalent dynamic range to floating-point. Arnold gives a description of a multilayer structure where successive layers have increased functionality and the designer chooses a layer depending on their requirements. A proposal for the implementation of the denormalized number concept in the LNS is also given. Lewis [258] presents a function interpolator that uses a second order approximation data path. An error bound is developed for the approximation of the addition and subtraction functions that allows a relative error that is *better than floating-point.* Lewis's approximation technique adheres to this error bound for all regions except close to the singularity where a relaxed error bound as proposed by Arnold [254] is used due to the difficulty in approximation the subtraction function. A cotransform is introduced by Coleman

[260, 263], which simplifies the task of calculating the logarithmic subtraction function in the singularity region. The cotransform shifts a value in the singularity region up to a more non-linear region where it is simple to approximate. In parallel the transform calculates a correction term due to the range shifting. Paliouras [264] introduces an identity that decomposes the logarithmic subtraction function into two much more easily approximated functions of restricted range. The transform enables the subtraction function in the singularity region to be calculated with high accuracy but with a low ROM requirement. A down side to the identity is that two functions need to be calculated in parallel, thus doubling the approximation arithmetic. Sacha [266] remarks that Givens rotations used in QR decomposition are a good candidate for LNS implementation due to the need for multiplication, division and square root operations in their calculations. The QR decomposition is often mapped to a systolic array for high performance processing and Papadourakis [250] proposes a processing element (PE) for a general systolic array based on logarithmic arithmetic, where each element can perform the four basic functions of addition/subtraction, multiplication, division and square root. Because a single PE has so much functionality, many different algorithms could be mapped to the array. Coleman [268] gives details of the design and accuracy of a 32-bit LNS ALU. Coleman's design uses the cotransform of Coleman [260] and a first order Taylor series approximation with concurrent error correction. The accuracy of the design is better than floating-point. A method of implementing long word length (up to 64-bits) LNS addition/subtraction is presented by Chen [269], which makes use of the direct approximation method. Additive and online multiplicative normalization methods are used for the exponential and log function approximations respectively, which constrains the amount of lookup memory required. A special algorithm is used to predict the number of leading zeros in the exponential function, which ensures that the value is calculated with sufficient accuracy to prevent the accuracy loss in the singularity region of the subtraction function that would otherwise occur. The ROM usage is low but the sequential nature of the function approximations makes the design slow. Arnold [271] discusses the implications of relaxing the better than floating-point accuracy measure as used by Lewis [259] and Coleman [268]. The relaxation of the error from better-than-floating-point to *faithful* can dramatically reduce the table size needed to calculate the LNS addition/subtraction function. Arnold illustrates the area savings by comparing an FPGA implementation of a reduced precision LNS adder with estimated

implementations of the adder proposed by Lewis [259] and Coleman [268]. Arnold [274] shows how the third term in certain quadratic interpolators (Lagrange and Taylor) can be performed with logarithmic arithmetic to reduce the multiplication requirement to one. An interesting paper, Paliouras [280], studies the optimal LNS base so as to minimize the number of words required for the addition/subtraction function compared to the commonly used base-2 implementation. Arnold [293] implements a compact iterative subtraction function approximation based solely on the addition function. The method relies on inverse interpolation of the LNS addition function, which can be exploited because the LNS subtraction function is the same as the inverse of the LNS addition function for positive arguments. Kurokawa [241], Swartzlander [243], Sicuranza [244], Shenoy [245], Chandra [247, 265], Frey [246], Coleman [268, 277], Youssef [279], Wang [281], Arnold [283-286, 289], Albu [287], Vainio [288] and Ruan [290, 291] show the accuracy or performance benefits of the logarithmic number system when compared to floating-point and fixed-point systems. The comparison of various algorithms is considered including: The Fast Fourier Transform, adaptive filters, recursive filters, low pass filters, FIR filters, QR decomposition Recursive least squares using Givens rotations, QR decomposition least-squares lattice, N-body problem and the Inverse Discrete Cosine Transform used in MPEG decoding.

5.3 An FPGA LNS literature overview

The first publication to describe the development of LNS operators on FPGA was Hermanek [294]. Hermanek's design used a 32-bit word length equivalent to IEEE std-754 single precision and was based on the design of Coleman [268]. The ROMs needed for the addition/subtraction function were stored off chip and this was a major bottleneck for the design and restricted the number of LNS addition/subtraction operations that could be placed on a single FPGA. Albu [296] presents details of the implementation of lattice algorithms to solve the least-squares problem recursively. The FPGA implementation uses logarithmic arithmetic and is based on the logarithmic ALU design of Hermanek [294]. Results showed that the FPGA implementation was capable of outperforming a commercial DSP solution. Arnold [271] implements an LNS adder with faithful rounding to restrict the area of the required lookup table. Arnold estimates the area of the designs of Lewis [258] and

Coleman [268] to demonstrate the savings of relaxing the addition function approximation error. Details of the LNS subtraction function are not given by Arnold [271]. Matousek [297] and Matousek [299] give a 32-bit and 20-bit word length FPGA implementation of the LNS addition/subtraction and other functions. The lookup tables for the addition/subtraction functions are stored in the on-chip BlockRAM memories. Three different algorithms of an information filter of square root RLS form, square root free Givens, and square root Givens, which are all candidates for systolic array implementation are implemented and compared. The algorithms are shown to have a higher throughput and be of smaller area when implemented in logarithmic arithmetic. The use of the BlockRAMs in dual port mode to calculate two LNS addition/subtraction operations in parallel is highlighted, as is the very high consumption of BlockRAM components in the 32-bit design. Albu [298] presents details of the implementation of the a priori error-feed-back least-squares lattice algorithm. The LNS operator designs of Matousek [297] are used by Albu to implement the algorithm in 20 and 32-bit arithmetic. The speed of the implementation is shown to be equivalent to a commercial DSP device, but the number of cycles required to execute the design is significantly less. UTIA [301] have developed a number of commercial LNS cores for FPGA including the conversion cores to convert to and from the log domain. Finally and most recently Detrey [303] applies the multipartite table based function approximation method to the implementation of the LNS addition/subtraction function. Detery presents a library of the four basic arithmetic operations implemented for the Virtex FPGA family. The addition/subtraction operator is fast because a lookup-add approximation method is used, however the size of the operators increases very quickly with word length due to the high ROM requirement of the multipartite method. Detrey uses a reduced precision approximation for the subtraction in the singularity region as proposed by Arnold [254] to help constrain the high memory requirements. The design compares favourably with other low precision implementations.

5.4 LNS word format

The LNS format is modelled on the format given in Arnold [257], which applies features of the IEEE 754 standard to LNS in an attempt to provide a unified LNS format. The work of Arnold [257] provides guidelines for the support of the special

values of NaN, infinity, normalised, denormalized and zero as provided by the IEEE 754 standard. Adopting the features of Arnold [257], which uses special integer values to represent the special values, gives equivalent LNS and floating-point number systems the same dynamic range and special value support, which allows a fair comparison of the two systems. As for the developed floating-point operations the aim of this work is to provide parameterisable LNS components where the length of the integer and fraction parts can be specified. Support will not be provided for different rounding modes or for denormalized values.

5.5 Relative error

A benefit of the LNS is the superior relative error of the system over floating-point. Firstly consider the relative error for a floating-point system with a t-bit mantissa m. We will assume that m is normalised to the range [1, 2). The relative error for an arbitrary base b system is given in (5.1), where m' is an approximation to m.

$$\frac{(m'-m)}{m} \leq \frac{b^{-t-1}}{m} \qquad (5.1)$$

The relative error can be seen to have a range given in (5.2).

$$\frac{b^{-t-1}}{1} \leq \frac{b^{-t-1}}{m} < \frac{b^{-t-1}}{2} \qquad (5.2)$$

The maximum relative error for a base-2 floating-point system is given in (5.3).

$$2^{-t-1} \qquad (5.3)$$

The absolute error A_e for a logarithmic number system with a fraction of length t-bits is given in (5.4).

$$A_e = k'-k = 2^{-t-1} \qquad (5.4)$$

The equivalent absolute linear error is given in (5.5).

$$A_{elln} = 2^{k'} - 2^{k} \qquad (5.5)$$

The relative error of (5.5) is given by (5.6).

$$R_e = \frac{2^{k'} - 2^{k}}{2^{k}} = 2^{k'-k} - 1 = 2^{2^{-t-1}} - 1 \qquad (5.6)$$

We can write (5.6) in a similar style to (5.3) to allow a direct comparison for different fraction lengths by performing the mathematical steps of (5.7-5.9).

$$2^{2^{-t-1}} - 1 = 2^{-t-x} \quad \text{(5.7)}$$

$$\frac{2^{2^{-t-1}} - 1}{2^{-t}} = 2^{-x} \quad \text{(5.8)}$$

$$-\log_2\left(\frac{2^{2^{-t-1}} - 1}{2^{-t}}\right) = x \quad \text{(5.9)}$$

t is the length of the number of fraction bits. By varying t a value for the relative error for each value of t can be found by solving (5.9) for x and placing x into the RHS of (5.7). The RHS of (5.7), which is the LNS relative error, can be directly compared with (5.3), which is the floating-point relative error. Figure 5.2 shows a graph of how the x term in (5.9) varies with the fraction bit length t.

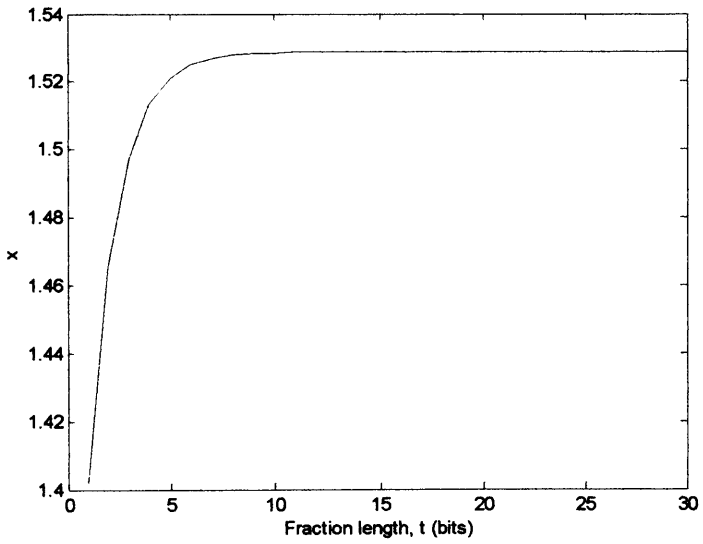


Figure 5.2. The variation of the x term in (5.9) with the t term

Figure 5.2 shows that the relative error is not constant for every fraction length, but does converge to a constant value of approximately 1.529 rounded to 3 decimal places. Furthermore from figure 5.2 it can be deduced that the LNS (5.11) has a ½ ulp relative error improvement over floating-point (5.10) for an equivalent word format above 4 fraction/mantissa bits.

$$R_{flp} = 2^{-t-1} \quad (5.10)$$

$$R_{LNS} = 2^{-t-1.529} \quad (5.11)$$

5.6 Dynamic range

Consider an *e*-bit exponent, *m*-bit mantissa floating-point number system without any exponents reserved for special value encoding. The maximum value the number system can represent is given in (5.12).

$$Mx_{flp} = \pm 2^{(2^{e-1}-1)} * (2 - 2^{-m}) \approx 2^{2^{e-1}} \quad (5.12)$$

The minimum value the number system can represent is given in (5.13).

$$Mn_{flp} = \pm 2^{-2^{e-1}} \quad (5.13)$$

The same can be done for an *i*-bit integer *f*-bit fraction logarithmic number system where the maximum value that can be represented is shown in (5.14).

$$Mx_{LNS} = \pm 2^{i-1} - 2^{-f} \approx 2^{i-1} \quad (5.14)$$

To convert (5.14) to the real domain it needs to be set as the power that raises the base, which is shown in (5.15).

$$Mx_{LNS}real = 2^{2^{i-1}} \quad (5.15)$$

The minimum LNS value and its corresponding real value are shown in (5.16) and (5.17) respectively.

$$Mn_{LNS} = -2^{i-1} \quad (5.16)$$

$$Mn_{LNS}real = \pm 2^{-2^{i-1}} \quad (5.17)$$

Comparing (5.15) with (5.12) and (5.17) with (5.13) shows that when the integer width of a logarithmic number system is the same as the exponent width of a floating-point system the two systems will have the same dynamic range.

5.7 Special value encoding

Table 5.1 shows the LNS encodings used for the special values of zero, denormalized, normalized, infinity and NaN.

Special value	Integer bit pattern	Fraction bit pattern	Integer value in two's complement	Fraction value
Zero	10...01	00...00	$-(2^{i-1})$	0
Denormalized	10...01	Not (00...00)	$-(2^{i-1})$	$\neq 0$
Infinity	10...00	00...00	$-(2^{i-1}-1)$	0
NaN	10...00	Not (00...00)	$-(2^{i-1}-1)$	$\neq 0$
Normalised	Not above	Any	$[(2^{i-1}-1), -(2^{i-1}-2)]$	$[0,1-2^{-f}]$

Table 5.1. Encodings of the special values used in the LNS

5.8 Logarithmic number system addition/subtraction

The standard arithmetic operators as used in the real domain do not perform the equivalent operation in the logarithmic domain so alternative operators need to be devised. Addition/subtraction in the logarithmic domain is the most complex operation and it is one of the reasons why the LNS is considered difficult to implement. Consider the two LNS values K and M given by equations (5.18) and (5.19) respectively.

$$K = \log_2(X) \quad \text{(5.18)}$$

$$M = \log_2(Y) \quad \text{(5.19)}$$

To add or subtract two logarithmic values they need to be converted to the real domain added and converted back to the log domain as shown by equation (5.20).

$$K \pm_{LNS} M = \log_2(2^K \pm 2^M) \quad \text{(5.20)}$$

The RHS of (5.20) can be rewritten by using the steps of (5.21).

$$\log_2(2^K \pm 2^M) = \log_2\left(\frac{2^K(2^K \pm 2^M)}{2^K}\right) = K + \log_2(1 \pm 2^{M-K}) \quad \text{(5.21)}$$

A logarithmic number Z consists of two parts a sign bit z_s and a magnitude part z_v. In (5.21) the values of M and K are just the magnitudes of the logarithmic numbers as the signs are used along with the operator to decide whether to perform and addition/subtraction. So the RHS of equation (5.21) should read as (5.22).

$$k_v + \log_2(1 \pm 2^{m_v - k_v}) \quad \text{(5.22)}$$

If we let $R = m_v - k_v$ then the addition function is given by (5.23). For $k_v \geq m_v$ the subtraction is given by (5.24). However for $k_v < m_v$ the log of a negative number is attempted so for $k_v < m_v$ the subtraction function is given by (5.25).

$$k_v + \log_2(1 + 2^R) \quad \text{(5.23)}$$

$$k_v + \log_2(1 - 2^R) \text{ where } R \leq 0 \quad \text{(5.24)}$$

$$k_v + \log_2\left|1 - 2^R\right| \text{ where } R \geq 0 \quad \text{(5.25)}$$

k_v and m_v can be swapped around without loosing generality so that $k_v \geq m_v$ and therefore only equations (5.23) and (5.24) need be approximated. This is the most commonly used approach to the calculation of the addition and subtraction functions and is the one which shall be used.

5.8.1 The basic algorithm

The basic logarithmic addition/subtraction algorithm is summarised in the following 6 steps.

Step 1.

Compare the two input operand magnitudes to determine the smallest and largest.

Step 2.

Subtract the largest magnitude from the smallest to determine the difference to feed the addition/subtraction function.

Step 3.

Use the input operand sign bits and the operator, which has the convention of being '1' for subtract and '0' for add, to determine whether the function addressed by R should be the addition or the subtraction.

Step 4.

Calculate the addition/subtraction function.

Step 5.

Add the result of the addition/subtraction function onto the largest magnitude operand.

Step 6.

Check the result for overflow/underflow.

5.8.2 The addition/subtraction function

From the basic algorithm the most complicated part is the calculation of the addition/subtraction function. In figure 5.3 a graph of the addition and subtraction functions of (5.23) and (5.24) is given.

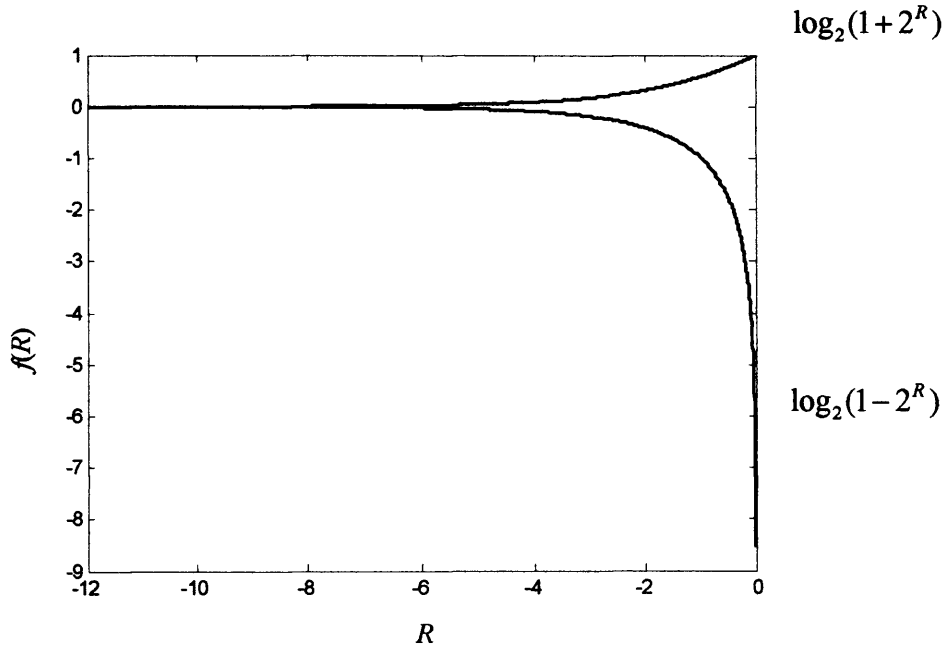


Figure 5.3. The addition/subtraction function

The addition function is a smooth function with restricted range. The subtraction on the other hand is highly non-linear due to the singularity, which causes the range to tend to negative infinity as the domain tends to zero. The addition/subtraction functions are evaluated at a point R, which has a domain $[a, b]$. The lower bound 'a' has a minimum value of $\min(x_v)$-$\max(x_v)$ where x_v is the magnitude of an LNS number. In the system used in this work the lower bound has a value (5.26).

$$a = -(2^{i-1} - 2) - (2^{i-1} - 2^{-f})$$
$$a = -2^i + 2 + 2^{-f} \quad \text{(5.26)}$$

The upper bound for R is 0. The domain of R is given in (5.27).

$$R \in [-2^i + 2 + 2^{-f}, 0] \quad \text{(5.27)}$$

5.8.3 The addition function

Now the domain of R is known the range of the addition function can be calculated. From (5.27) it can be deduced that the smallest value of $\log_2(1+2^R)$ depends on the number of integer bits. If the chosen format does not contain enough fraction bits then a portion of the smallest values of the function $\log_2(1+2^R)$ cannot be represented in the precision available. This means that these values need not be stored or calculated because after rounding they would quantize to zero. This is analogous to the situation in the floating-point number system where the exponent of one operand is so small that the significand is shifted out of range of the other and so an addition of zero has the same effect. A numerical example will illustrate the point: suppose we have a format with 8 integer and 23 fractional bits then the smallest logarithmic value that can be represented is $2^{-f} = 2^{-23}$. If a value of R is –40, which is easily possible with an 8-bit integer, then $\log_2(1+2^R)$ is approximately $2^{-39.5}$, which is much smaller than the smallest value that can be represented.

Now there is a converse problem here, which is, what is the largest value (in magnitude) of R that is permissible without the add function quantising to zero? This is governed by the fractional precision f and is also affected by rounding. The value of the add function must be greater than ½ ulp otherwise it will quantise to zero. This constraint creates equation (5.28).

$$\log_2(1+2^R) > 2^{-f-1} \quad \text{(5.28)}$$

Rearranging equation (5.28) gives an integer limit on the value of R and these steps are shown in (5.29).

$$\begin{aligned} &\log_2(1+2^R) > 2^{-f-1} \\ &2^R > 2^{2^{-f-1}} - 1 \\ &R > \left\lceil \log_2(2^{2^{-f-1}} - 1) \right\rceil \\ &R > -(f+2) \end{aligned} \quad \text{(5.29)}$$

Therefore R must be greater than $-(f+2)$ otherwise the function will quantise to 0. The addition function need only be evaluated for the domain given in (5.30).

$$R \in (-(f+2), 0] \quad \text{(5.30)}$$

The range of the function is given in (5.31).

$$\log_2(1+2^R) \in [0,1] \quad \text{___(5.31)}$$

The range in (5.31) implies that the maximum output length of a rounded logarithmic addition function is f+1 bits.

5.8.4 The subtraction function

The value of R has similar domain interval restrictions for the subtraction function. The arithmetic steps to deduce the maximum (in magnitude) value of R for subtraction are shown in (5.32).

$$\begin{aligned} &\log_2(1-2^R) > -2^{-f-1} \\ &-2^R > 2^{-2^{-f-1}} - 1 \\ &2^R > 1-2^{-2^{-f-1}} \quad \text{___(5.32)} \\ &R > \left\lceil \log_2(1-2^{-2^{-f-1}}) \right\rceil \\ &R > -(f+2) \end{aligned}$$

Equation (5.32) gives the lower integer bound on the domain of R. The singularity at zero causes the subtraction function to be highly non-linear. Despite the function evaluating to negative infinity at zero this is not a problem since when R is zero and a subtraction operation is required the operation is analogous to the difference of two values that are equal and so the output is zero. Zero is a special case value in the LNS and if R is zero and a subtraction operation is detected the output is forced to the special value zero. Equation (5.33) shows the domain of R that the subtraction function should be evaluated on.

$$R \in (-(f+2), -2^{-f}] \quad \text{___(5.33)}$$

The range of the function is given in equation (5.34).

$$\log_2(1-2^R) \in [0, -(f+1)) \quad \text{___(5.34)}$$

The range in (5.34) implies that the output of the logarithmic subtraction function is $\lceil \log_2(f+1) \rceil + f$ bits in length (not including sign bits).

As shown in (5.30) and (5.33) the function approximations are heavily dependent on the length of the fractional part of the LNS operand. Both approximations have an input argument length given by (5.35) and tabulated in table 5.2 for different fraction lengths.

$$\lceil \log_2(f+2) \rceil + f \qquad (5.35)$$

f	f+2	$\lceil \log_2(f+2) \rceil$	Input width
3	5	3	6
4	6	3	7
5	7	3	8
6	8	3	9
7	9	4	11
8	10	4	12
9	11	4	13
10	12	4	14
11	13	4	15
12	14	4	16
13	15	4	17
14	16	4	18
15	17	5	20
16	18	5	21
17	19	5	22
18	20	5	23
19	21	5	24
20	22	5	25
21	23	5	26
22	24	5	27
23	25	5	28
24	26	5	29
25	27	5	30

Table 5.2. The input width of the function approximation for a given fraction length f

For small fraction widths f of up to 5-bits full table lookup can be used to approximate the addition and subtraction functions. This is because the function to approximate is up to a maximum of 8-bits in width as shown in table 5.2. Using full table lookup means the results can be correctly rounded. Figure 5.4 shows the structure of the addition/subtraction function created with only table lookups. The addition and subtraction functions can be calculated in parallel as shown in figure 5.2. The output is selected using a multiplexer depending on whether an addition or subtraction operation is to take place. For fraction widths larger than 5-bits full table lookup is not feasible due to the large memory requirements. A more appropriate method of calculating the addition subtraction function is to use a piecewise polynomial function approximation scheme. However by using a polynomial approximation the addition/subtraction functions cannot be correctly rounded without creating very

accurate approximations and then rounding Schulte [198] and Lefevre [304]. The reason that results cannot be correctly rounded is due to the table-makers dilemma Lefevre [304], which is described next.

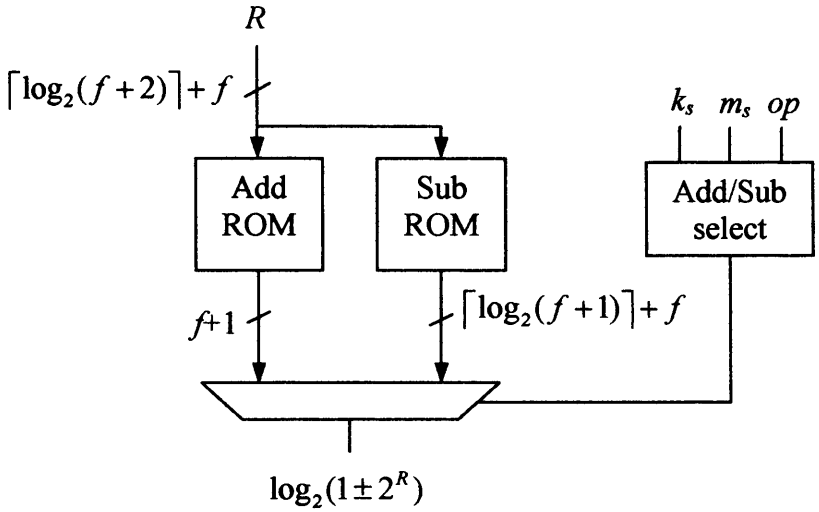


Figure 5.4. Addition/subtraction function approximation using full table lookup

5.8.5 Table-makers dilemma

Consider a function where at a point x the function takes the value (5.36).

$$f(x) = 0.10111000000101 \quad \text{(5.36)}$$

The function of (5.36) has 14 fraction bits. If (5.36) is rounded to 4 fraction bits using a round-to-nearest scheme then the result is given by (5.37).

$$f(x)_{md} = 0.1100 \quad \text{(5.37)}$$

An approximation to (5.36) is given in (5.38).

$$\tilde{f}(x) = 0.10110111111011 \quad \text{(5.38)}$$

Equation (5.38) is a very good approximation to (5.36) as the difference is less than 2^{-10}. The difference is shown in (5.39).

$$f(x) - \tilde{f}(x) = 0.00000000001010 \quad \text{(5.39)}$$

Equation (5.40) shows the value of (5.38) rounded to 4 fractional bits.

$$\tilde{f}(x)_{md} = 0.1011 \quad \text{(5.40)}$$

Despite (5.38) being a very good approximation to (5.36) when both values are rounded to 4 fractional bits the result is different. So even if a value is calculated with many more bits of accuracy than is needed the value might still not be correctly rounded. This was the main motivation for the development of the faithful rounding criterion which rounds with a maximum error of less than 1 ulp instead of the correctly rounded ½ ulp maximum error. By analysing (5.36) and (5.38), which might be interchanged because one might be the approximation and the other the true value and vice-versa, it can be seen that the table-makers dilemma is caused when the exact value of a function has one of two rounding bit patterns (for round to nearest rounding) shown in figure 5.5. Producing very accurate approximations that can be correctly rounded requires a lot of hardware in terms of both lookup ROM and arithmetic components. The extra hardware requirement is too great to make the LNS addition/subtraction scheme practical. Having correctly rounded results makes the accuracy of the LNS better than the equivalent floating-point system. However, the accuracy can be reduced to be equivalent to, but still slightly better than, floating-point and this enables the addition/subtraction function to be calculated sufficiently accurately using a polynomial approximation.

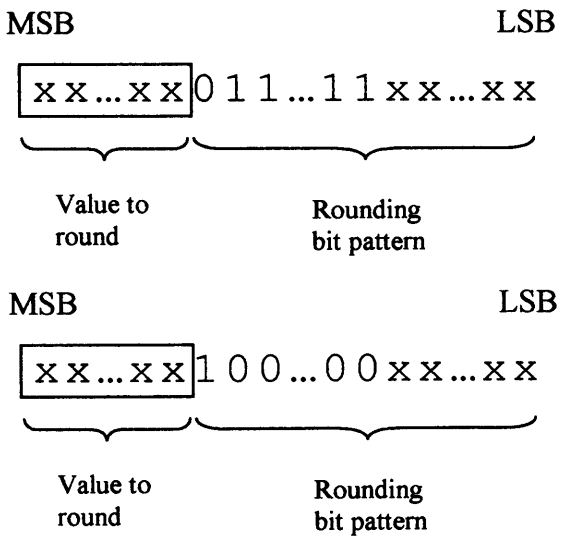


Figure 5.5. The rounding bit patterns that cause the table-makers dilemma

5.8.6 Better than floating-point accuracy (BTFP)

The LNS addition/subtraction function will incur an error as a result of rounding, which is 2^{-f-1}. The addition/subtraction approximation will also have an error ε_{Φ},

which gives a total error of $2^{-f-1}+\varepsilon_{dp}$. The relative error for a realistic LNS implementation, i.e. one with approximation errors is given by (5.41).

$$\varepsilon_{LNS}=2^{(\varepsilon_{dp}+2^{-f-1})}-1 \quad \text{(5.41)}$$

We would like the LNS to have an error that is equivalent or less than that of floating-point. The equation to calculate the permissible data path error that results in an equivalent accuracy of floating-point is

$$2^{(\varepsilon_{dp}+2^{-f-1})}-1\leq 2^{-f-1} \quad \text{(5.42)}$$

Solving (5.42) for ε_{dp} gives (5.43).

$$\varepsilon_{dp}\leq \log_2(2^{-f-1}+1)-2^{-f-1} \quad \text{(5.43)}$$

Equation (5.43) gives the maximum permissible approximation error for a given fraction length f. Equation (5.43) can be arranged and expressed in terms of the number of fractional bits of accuracy required and is shown in the following steps. Letting,

$$\varepsilon_{dp}=2^{-f+x} \quad \text{(5.44)}$$

$$x=\log_2(\varepsilon_{dp})+f \quad \text{(5.45)}$$

Substituting (5.43) into (5.45) gives,

$$x=\log_2\left(\log_2(2^{-f-1}+1)-2^{-f-1}\right)+f \quad \text{(5.46)}$$

A graph of x for various fraction lengths f is shown in figure 5.6. The graph of figure 5.6 implies that the addition/subtraction function must be calculated with an error of less than $2^{-f-2.2}$ to enable the result to have better than floating-point accuracy (BTFP) for faction lengths of 6-bits or more. This is the maximum permissible error between the function and the true result before rounding to the target fraction length of f-bits.

5.9 Addition/subtraction function approximation methods

Now the input domain and the output range with permissible error is known different function approximation methods can be implemented. In this section two FPGA

approximation methods are considered, one that is based on the dual-path floating-point adder algorithm and the other uses a parallel-lookup approximation to constrain the delay of the addition/subtraction function to be a single polynomial function.

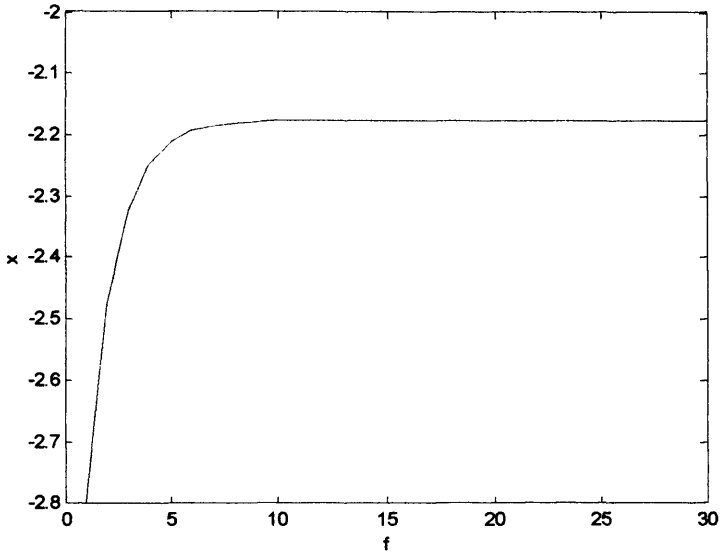


Figure 5.6. A graph of x versus f as expressed in equation (5.46)

5.9.1 Dual-path LNS addition/subtraction approximation

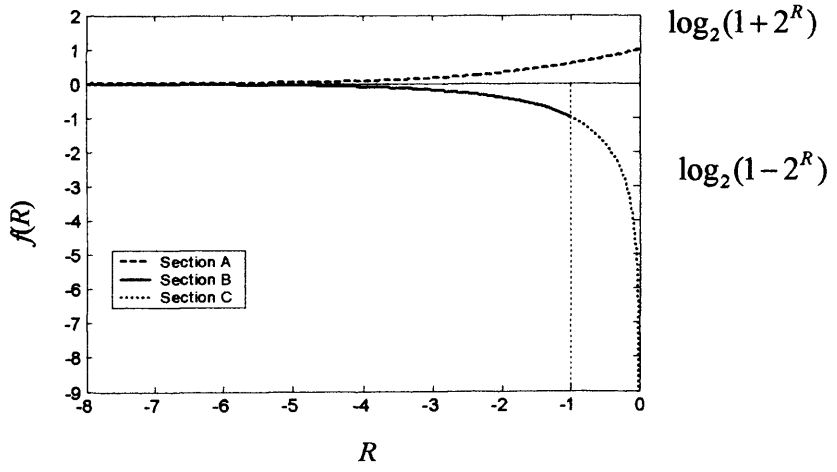


Figure 5.7. The three section split of the addition/subtraction function

The dual-path floating-point addition scheme has two paths. One path is chosen if a subtraction operation is performed and the exponent difference is 1 or 0. The other path is chosen if an addition function is chosen or if a subtraction is chosen and the

exponent difference is greater than 1. Using a similar method the LNS addition function is calculated for the whole domain of (-(*f*+2), 0].

The subtraction function domain is split into two intervals. One interval is (-(*f*+2), -1] and the other is (-1, 0), which are analogous to the near and far paths of the dual-path floating-point addition algorithm. The addition function (section A) and the split of the subtraction function (sections B and C) are shown in figure 5.7.

5.9.1.1 Far path

The hardware required to calculate the addition function for the domain (-(*f*+2), 0) is shown in figure 5.8 (a). The hardware required to calculate the subtraction function for the domain (-(*f*+2), -1) is shown in figure 5.8 (b). The range of the values at each stage is shown in figure 5.8. The ranges are small and the functions to approximate are smooth and suit the piecewise polynomial approximation approach. The approximations of figure 5.8 (a) and (b) share the same hardware and can easily be combined into one data path as shown in figure 5.9.

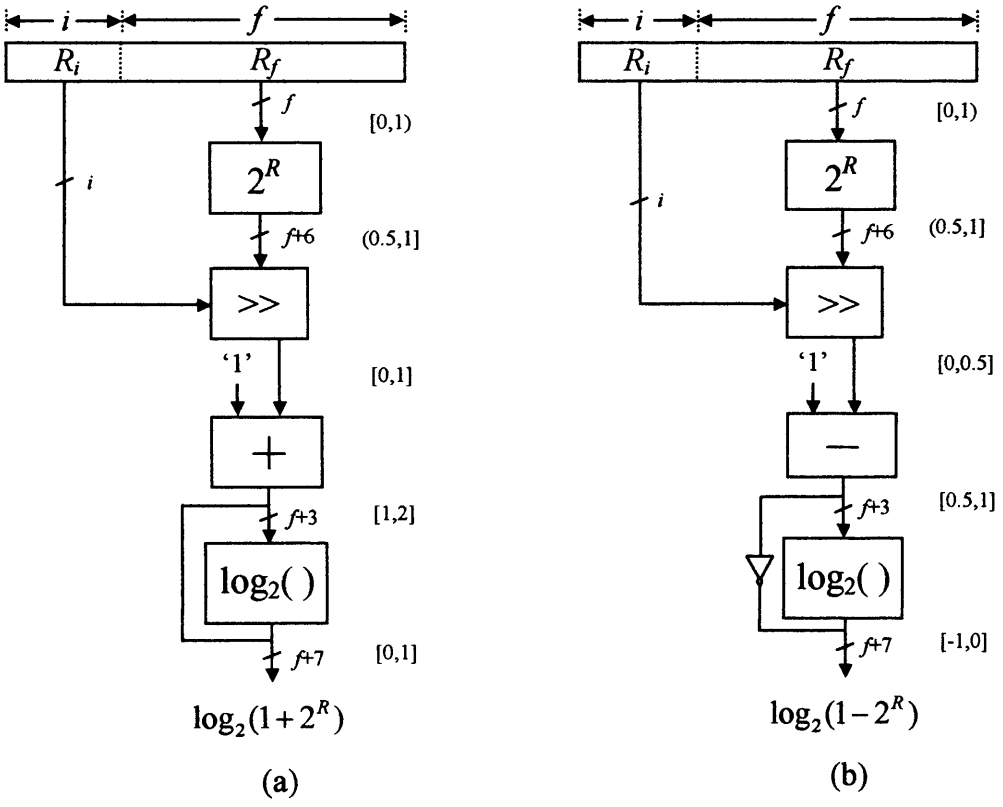


Figure 5.8. The hardware needed to approximate sections A and B of figure 5.7

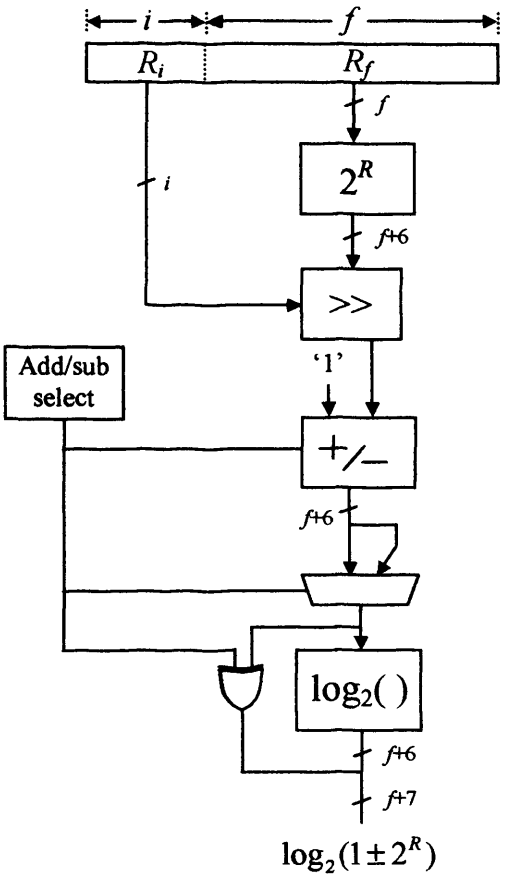


Figure 5.9. The combined hardware unit to approximate the far path sections A and B

The only difference between figures 5.8 and 5.9 is that a multiplexer is needed to shift the input value to the log approximation when a subtraction operation is chosen. The log function approximation needs to be calculated for the interval [1, 2]. Typically, to save and simplify the hardware, functions are only approximated on a single power of 2 interval. Now the interval [1, 2] is just larger than a power of 2 interval, which would be [1, 2), i.e. not including 2. Therefore for the log approximation the value 2 is treated as a special case and is corrected by adding 1 onto the output of the approximation of $\log_2(1)$. No extra hardware is needed to do this as the '1' bit can be concatenated onto the MSB end of the result of the log approximation. For the subtraction path the value that feeds the log approximation must be left shifted by 1-bit from the interval [0.5, 1] to the interval [1, 2]. This shift is compensated by subtracting the value 1 from the output of the function approximation, which because of the result range and the properties of two's complement need only be concatenated onto the MSB end. When the shifted value is 2 for subtraction 1 is added and 1 is subtracted resulting in an addition of 0. The value of the bit to concatenate is

calculated by XORing the add/sub select bit and the MSB of the value that feeds the log approximation function.

5.9.1.2 Near path

The design of figure 5.9 cannot be used to approximate the subtraction function for the range (-1, 0) because of a loss of accuracy. The loss of accuracy is primarily caused due to the 2^R approximation being close to 1. The subsequent subtraction from 1 has a very small result with many leading zeros and few significant bits. The very small quantity of significant bits means the log function approximation is inaccurate. The only way to solve the problem would be to calculate the 2^R function with double the accuracy but that would require a great deal of extra hardware. To solve the problem the identity shown in (5.47) is used.

$$\log_2(1-2^R) = \log_2\left(\frac{1-2^R}{-R}\right) + \log_2(-R) \qquad \text{(5.47)}$$

The identity of (5.47) was first proposed in Paliouras [264] and consists of two parts. The first part (5.48) is the correction part and the second part is the logarithm function.

$$\log_2\left(\frac{1-2^R}{-R}\right) \qquad \text{(5.48)}$$

A graph of the correction function is shown in figure 5.10 for the domain (-2, 0).

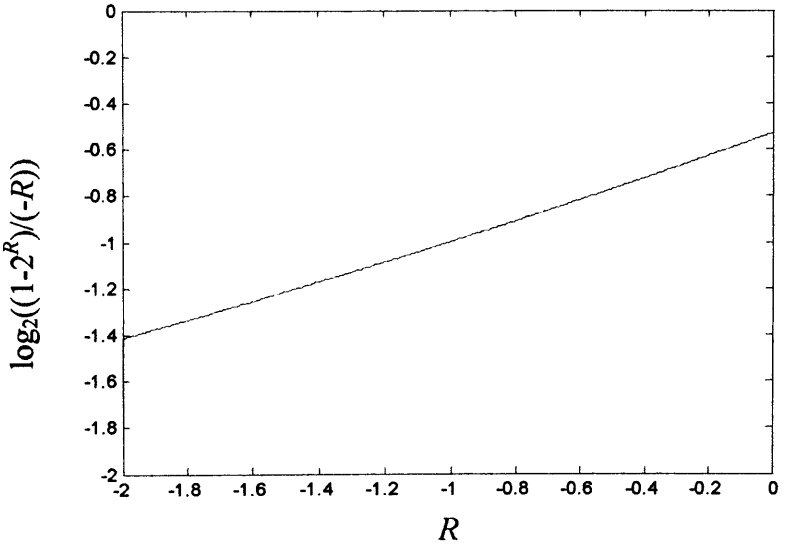


Figure 5.10. The correction function

Figure 5.10 shows that (5.48) is a smooth function of restricted range and can easily and accurately be approximated with a piecewise polynomial scheme. The log function of (5.47) is calculated for an interval of [1, 2). To calculate the log of a function with a domain of (0, 1) the argument must first be left shifted into the interval [1, 2), which does not cause any accuracy loss. The log of the shifted value is calculated and is then added onto the negated shift amount giving the final result. Adding the log value and the correction value gives an accurate approximation to the logarithmic subtraction function. The hardware needed to implement the described process is shown in figure 5.11.

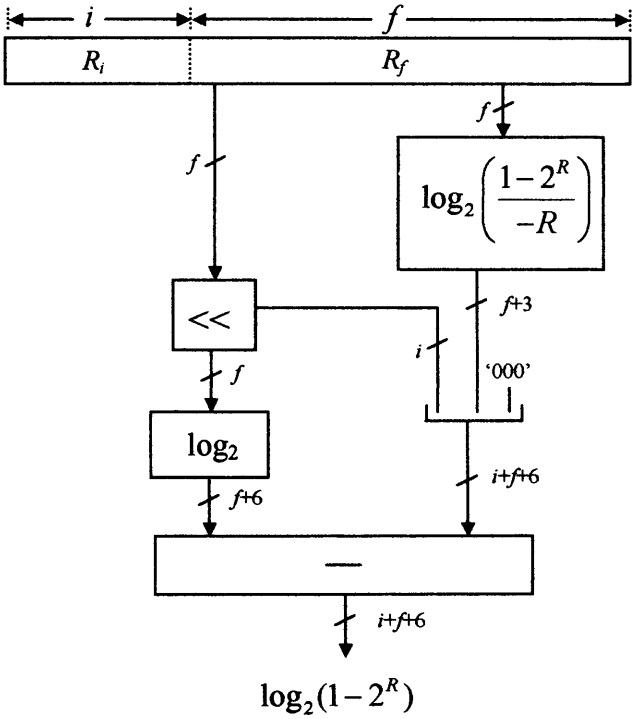


Figure 5.11. The hardware unit to approximate the near-path section C

Figure 5.9 shows the far-path and figure 5.11 shows the near-path. The complete addition/subtraction function approximation requires both the far and near path to be combined into one unit. We can take advantage of common components in the two paths and share the hardware so reducing the size of the final design. Comparing the two paths it can be seen that the log approximation for the domain [1, 2) is common and can be shared between the two paths. The 2^R and 'correction' functions can share the same approximation arithmetic i.e. multipliers and adders. Figure 5.12 shows the complete function approximation design based on the dual-path (near and far path) approach.

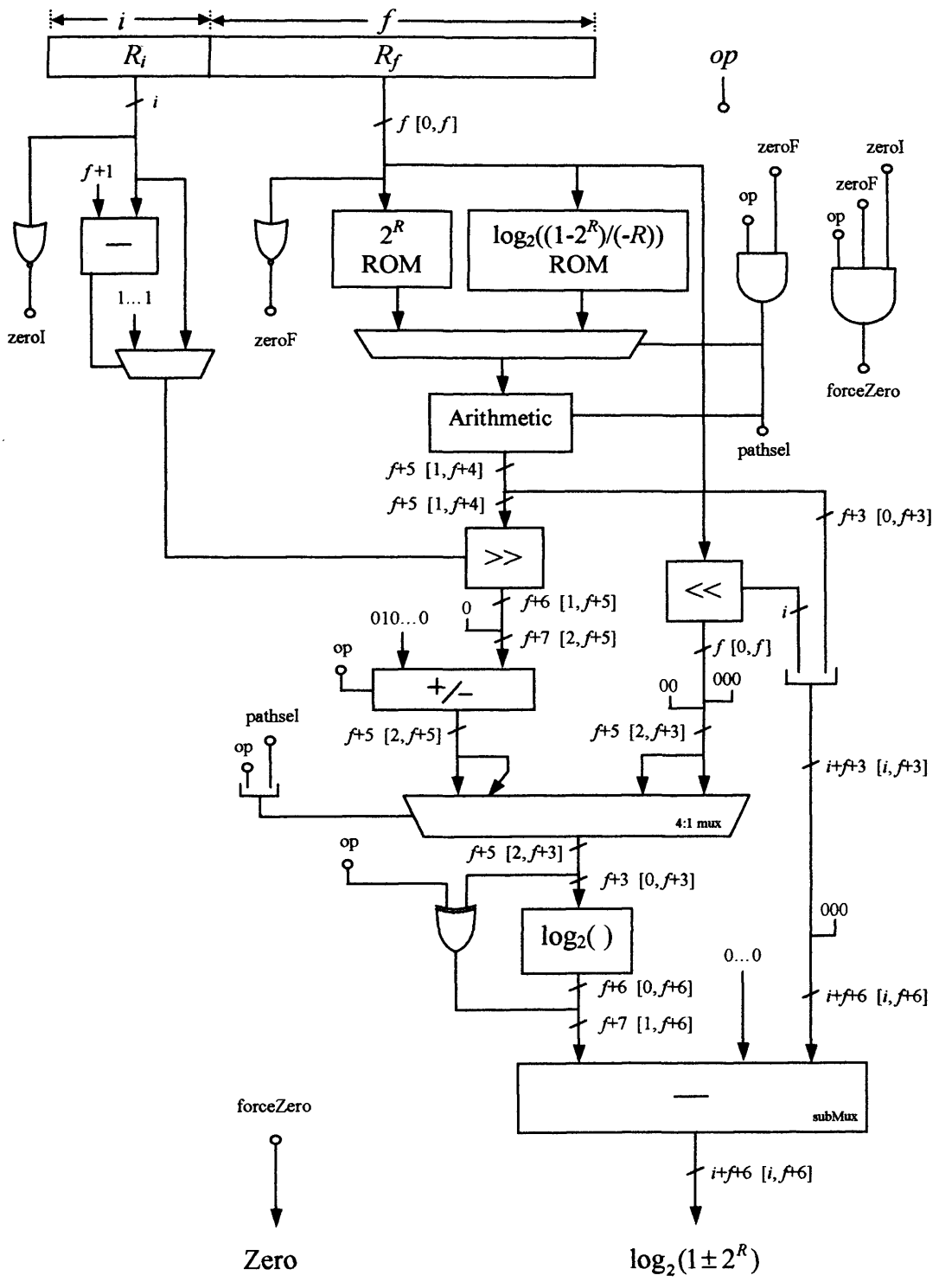


Figure 5.12. The complete dual-path LNS addition/subtraction function

The '*op*' input indicates whether an addition or subtraction approximation needs to take place. The path selection depends on the desired operation and whether the input integer is zero or not. An integer of zero can be detected by a wide NOR gate. The

'Zero' output signals whether the input is zero and a subtraction operation is required. In such circumstances the output of the LNS subtraction should be zero because the subtraction of two equal values is performed. The 4 to 1 path selecting multiplexer only has 3 inputs to pass to the logarithm approximation and because of this the input for the near-path is duplicated to simplify the selection logic. The left shifter uses the same scheme as was used in the dual-path floating-point adder. The output of the approximation is not rounded but has an error that after rounding will produce a better-than-floating-point (BTFP) level of accuracy. The three function approximations have been constrained to be f and f+3 bits, with restricted domains and ranges. These lookup sizes should be compared with the direct approximations of table 5.2 where for example a 23-bit fraction length requires two 28-bit (f+5) approximations and one of a highly non-linear function. The shifting is the key to the approximation width savings as they act as range reducers. The main problem with the dual-path approximation is that it requires two sequential function approximations. As a single 23-bit approximation can take 45 nanoseconds (see section 4.3.7.1) the sequential style of the algorithm adds at least this amount of time compared to a single polynomial solution. Furthermore the benefit of the solution decreases for smaller fraction lengths as the extra function approximation width varies with the log of the fractional section (see table 5.2).

5.9.1.3 Function approximation

For the base-2 logarithm, 2^R and correction functions a piecewise polynomial approximation scheme is used. The domain of each approximation is partitioned into 2^x intervals and each interval is approximated using the Chebyshev approximation technique. The Chebyshev technique was chosen over the Taylor series scheme described in section 4.3 because the table structure is simpler and the error of the approximation is less. The domain of the approximation is split into a power of 2 intervals so that the coefficients for the intervals can be grouped into a single ROM that can be addressed by the MSBs of the function argument. This makes the coefficient selection for a particular argument very simple.

5.9.1.4 Chebyshev function approximation

The Chebyshev polynomial function approximation technique is very close to the optimal polynomial function approximation technique the 'minimax approximation'.

The Chebyshev approximation is much simpler to calculate, as it does not require the task of searching for ideal polynomial coefficients. Implemented in fixed-point arithmetic with correct rounding the Chebyshev approximation can simply and easily be used to approximate a function with almost equal maximum negative and positive errors. The Chebyshev approximation of degree n is denoted $T_n(x)$ and is generated recursively by (5.49).

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad \text{(5.49)}$$

Where,

$T_0(x) = 1$, $T_1(x) = x$ and $n \geq 1$.

The heart of the approximation involves calculating the formula for $f(x)$ (5.50).

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2}c_0 \quad \text{(5.50)}$$

In (5.50) $T_k(x)$ is known from (5.49), but the coefficients are the crucial part of (5.50) as they need to be produced exclusively for each function and for each interval that the approximation is applied to. Equation (5.51) shows the formula to calculate the coefficients to evaluate a function $f(x)$ for the domain [-1, 1].

$$c_j = \frac{2}{N}\sum_{k=0}^{N-1} f(x_k)T_j(x_k)$$

$$c_j = \frac{2}{N}\sum_{k=0}^{N-1} f\left[\cos\left(\frac{\pi(k+0.5)}{N}\right)\right]\cos\left(\frac{\pi j(k+0.5)}{N}\right) \quad \text{(5.51)}$$

Typically we want to evaluate a function for a domain other than [-1, 1], for example when performing a piecewise approximation. Here a change of variable formula (5.52) is used to allow the coefficients to be calculated for the domain [a, b].

$$y \equiv \frac{x - 0.5(b+a)}{0.5(b-a)} \quad \text{(5.52)}$$

Equation (5.52) is applied to the argument of x_k of $f(x_k)$ in (5.51) and also to the argument of $T_k(x)$ in (5.50). Equipped with equations (5.49-5.52) it is possible to generate a Chebyshev approximation of any order and of any function $f(x)$ for the

domain [a, b]. The first and second order Chebyshev approximations of a function $f(x)$ are given in (5.53) and (5.54) respectively.

$$f(x) \approx C1 * x + C0 \quad \text{___(5.53)}$$

$$f(x) \approx C2 * x^2 + C1 * x + C0 \quad \text{___(5.54)}$$

For a piecewise approximation the CX coefficients depend on the MSBs of the function argument. Let the argument x be split into two sections the MSBs x_h and the LSBs x_l (5.55).

$$x = x_h + x_l \quad \text{___(5.55)}$$

Equations (5.53) and (5.54) can be written as (5.56) and (5.57) respectively, where the CX coefficients are functions of x_h.

$$f(x) \approx C1(x_h) * x + C0(x_h) \quad \text{___(5.56)}$$

$$f(x) \approx C2(x_h) * x^2 + C1(x_h) * x + C0(x_h) \quad \text{___(5.57)}$$

In equations (5.56) and (5.57) arithmetic is performed with full-length x terms. The full-length arithmetic operations can be avoided by rearranging (5.56) and (5.57) and modifying the ROM contents as shown in the following steps for the second order case:

$$f(x_h + x_l) \approx C2(x_h) * x^2 + C1(x_h) * x + C0(x_h) \quad \text{___(5.58)}$$

$$\begin{aligned} f(x_h + x_l) &\approx C2(x_h) * (x_h + x_l)^2 + C1(x_h) * (x_h + x_l) + C0(x_h) \\ f(x_h + x_l) &\approx C2(x_h) * (x_h^2 + 2.x_h.x_l + x_l^2) + C1(x_h) * (x_h + x_l) + C0(x_h) \\ f(x_h + x_l) &\approx [C2(x_h)] * x_l^2 + [2.x_h.C2(x_h) + C1(x_h)] * x_l + [C1(x_h).x_h + C0(x_h)] \end{aligned} \quad \text{___(5.59)}$$

Letting,

$$\begin{aligned} S2(x_h) &= C2(x_h) \\ S1(x_h) &= 2.x_h.C2(x_h) + C1(x_h) \\ S0(x_h) &= x_h.C1(x_h) + C0(x_h) \end{aligned} \quad \text{___(5.60)}$$

gives:

$$f(x_h + x_l) \approx S2(x_h) * x_l^2 + S1(x_h) * x_l + S0(x_h) \quad \text{___(5.61)}$$

Equation (5.61) shows that arithmetic operations only need to be done on the least significant bits of the x argument, while the SX ROMs only need to be addressed by the most significant bits of the x argument.

Horner's method is used to evaluate the second order approximation as it leads to more accurate approximations. Horner's method also reduces the number of multiplications by 1, but increases the critical path to include an extra table lookup and addition. Equation (5.62) shows the rearrangement of (5.61) according to Horner's method.

$$f(x_h + x_l) \approx (S2(x_h) * x_l + S1(x_h)) * x_l + S0(x_h) \qquad (5.62)$$

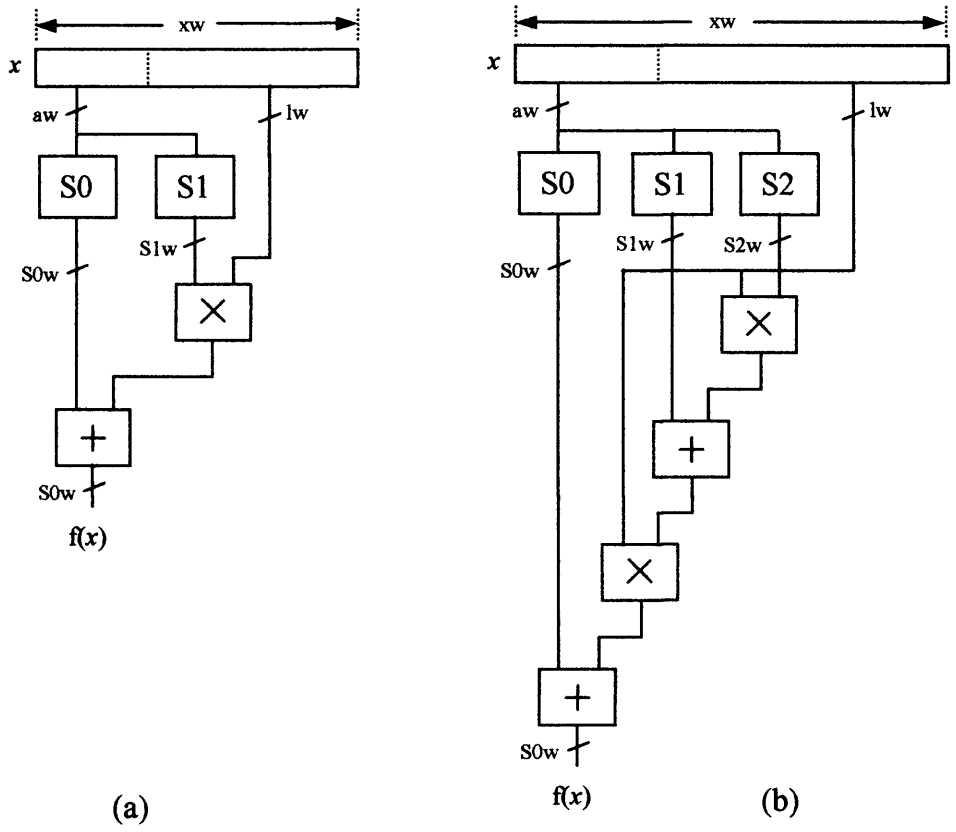


Figure 5.13. First and second order approximation hardware structures

The approximations are all performed with fixed-point arithmetic and the data path through the approximations has an equal number of fractional bits as the input argument plus a number of guard bits. The guard bits guard against a loss of precision in the data path so that the final result has the desired accuracy. Increasing the number of guard bits reduces the error committed by the initial coefficient rounding

and by the rounding of the output of each of the multipliers. However increasing the number of guard bits increases the amount of hardware required, so it is desirable to minimise the number of guard bits used. The first and second order approximation structures are shown in figure 5.13 (a) and (b) respectively.

5.9.1.5 Log_2, 2^R and correction approximation ROM address, LSB and ROM content widths

Table 5.3 illustrates the ROM address, LSB and ROM content widths for the 2^R and correction function approximations used in the dual-path LNS add/sub approximation and table 5.4 for the base-2 logarithm. The table also shows the order of approximation that is used for each fraction width. The orders are the same for the 2^R and correction function approximations because they share the same arithmetic. A zero approximation order is a pure table lookup. The number of guard bits g is 5, 3 and 2 for the 2^R, correction and base-2 logarithm functions respectively.

	2^R						correction					
f	aw	S0w	S1w	S2w	lw	order	aw	S0w	S1w	S2w	lw	order
3	3	f+g+1	X	X	0	zero	3	f+g	X	X	0	zero
4	4	f+g+1	X	X	0	zero	4	f+g	X	X	0	zero
5	5	f+g+1	X	X	0	zero	5	f+g	X	X	0	zero
6	6	f+g+1	X	X	0	zero	6	f+g	X	X	0	zero
7	7	f+g+1	X	X	0	zero	7	f+g	X	X	0	zero
8	4	f+g+1	f+g	X	4	fst	3	f+g	f+g+1	X	5	fst
9	5	f+g+1	f+g	X	4	fst	3	f+g	f+g+1	X	6	fst
10	5	f+g+1	f+g	X	5	fst	3	f+g	f+g+1	X	7	fst
11	6	f+g+1	f+g	X	5	fst	4	f+g	f+g+1	X	7	fst
12	7	f+g+1	f+g	X	5	fst	4	f+g	f+g+1	X	8	fst
13	7	f+g+1	f+g	X	6	fst	5	f+g	f+g+1	X	8	fst
14	5	f+g+1	f+g	f+g-2	9	sec	3	f+g	f+g	f+g-5	11	sec
15	5	f+g+1	f+g	f+g-2	10	sec	3	f+g	f+g	f+g-5	12	sec
16	5	f+g+1	f+g	f+g-2	11	sec	3	f+g	f+g	f+g-5	13	sec
17	5	f+g+1	f+g	f+g-2	12	sec	3	f+g	f+g	f+g-5	14	sec
18	5	f+g+1	f+g	f+g-2	13	sec	3	f+g	f+g	f+g-5	15	sec
19	6	f+g+1	f+g	f+g-2	13	sec	3	f+g	f+g	f+g-5	16	sec
20	6	f+g+1	f+g	f+g-2	14	sec	3	f+g	f+g	f+g-5	17	sec
21	6	f+g+1	f+g	f+g-2	15	sec	3	f+g	f+g	f+g-5	18	sec
22	6	f+g+1	f+g	f+g-2	16	sec	4	f+g	f+g	f+g-5	18	sec
23	7	f+g+1	f+g	f+g-2	16	sec	4	f+g	f+g	f+g-5	19	sec

Table 5.3. Address (aw), content (SXw) and LSB (lw) width of the correction and 2^R function approximations

		$\log_2$					
f	f+3	aw	S0w	S1w	S2w	lw	order
3	6	6	f+g+3	X	X	0	zero
4	7	7	f+g+3	X	X	0	zero
5	8	4	f+g+3	f+g+4	X	4	fst
6	9	4	f+g+3	f+g+4	X	5	fst
7	10	4	f+g+3	f+g+4	X	6	fst
8	11	4	f+g+3	f+g+4	X	7	fst
9	12	5	f+g+3	f+g+4	X	7	fst
10	13	5	f+g+3	f+g+4	X	8	fst
11	14	6	f+g+3	f+g+4	X	8	fst
12	15	6	f+g+3	f+g+4	X	9	fst
13	16	7	f+g+3	f+g+4	X	9	fst
14	17	7	f+g+3	f+g+4	X	10	fst
15	18	5	f+g+3	f+g+4	f+g+3	13	sec
16	19	5	f+g+3	f+g+4	f+g+3	14	sec
17	20	5	f+g+3	f+g+4	f+g+3	15	sec
18	21	6	f+g+3	f+g+4	f+g+3	15	sec
19	22	6	f+g+3	f+g+4	f+g+3	16	sec
20	23	6	f+g+3	f+g+4	f+g+3	17	sec
21	24	7	f+g+3	f+g+4	f+g+3	17	sec
22	25	7	f+g+3	f+g+4	f+g+3	18	sec
23	26	7	f+g+3	f+g+4	f+g+3	19	sec

Table 5.4. Address (aw), content (SXw) and LSB (lw) width of the base-2 log function

5.9.2 Parallel-lookup function approximation

The proposed parallel-lookup function approximation scheme constrains the delay of the addition/subtraction function to include only a single polynomial approximation. A similar, but slightly modified, function decomposition as was used for the dual-path approximation is adopted and the output of the scheme is designed to have better-than-floating-point accuracy. The function domains are large for the parallel-lookup scheme and so for fraction widths above 7-bits they are partitioned up into subsections which complicates the design but reduces the area of the coefficient ROMs.

5.9.2.1 Function decomposition

The addition function is approximated for the whole domain of $R \in (-(f+2), 0]$ using a single polynomial approximation. For fraction widths greater than 4-bits the subtraction function domain is split into two intervals. For fraction widths of 5 to 7-bits the two intervals are $R \in (-(f+2), -1]$ and (-1, 0). The first interval is approximated using a polynomial approximation and the second interval is approximated using a single ROM with an f-bit address. For fraction widths above 7-bits the first interval $R \in (-(f+2), -2]$ is approximated using a single polynomial

approximation. The second interval (-2, 0) is approximated using the identity given in (5.47), which requires two function approximations: one for the base-2 log function for the domain [1, 2) and the other for the 'correction' function for the domain (-2, 0).

5.9.2.2 Addition function approximation

f	domain	Input width	Domain split
3	(-5, 0]	6	(-(f+2), 0]
4	(-6, 0]	7	
5	(-7, 0]	8	
6	(-8, 0]	9	
7	(-9, 0]	11	
8	(-10, 0]	12	(-(f+2), -8] (-8, -4] (-4, -2] (-2, 0]
9	(-11, 0]	13	
10	(-12, 0]	14	
11	(-13, 0]	15	
12	(-14, 0]	16	
13	(-15, 0]	17	
14	(-16, 0]	18	
15	(-17, 0]	20	(-(f+2), -16] (-16, -8] (-8, -4] (-4, 0]
16	(-18, 0]	21	
17	(-19, 0]	22	
18	(-20, 0]	23	
19	(-21, 0]	24	
20	(-22, 0]	25	
21	(-23, 0]	26	
22	(-24, 0]	27	
23	(-25, 0]	28	

Table 5.5. The domain, input widths and the domain splits of the addition function for various fraction widths

For a fraction width of 3 to 4 bits the input domain is not split and a full table lookup is used to approximate the LNS addition function, as the input is has a maximum of 7 input bits. For a fraction width of 5 to 7 bits the LNS addition function is approximated using a single piecewise polynomial and the domain is not split. For every fraction length from 8 to 23 bits the addition function domain is split into four intervals. The four interval boundaries for each fraction length from 8 to 23 bits are shown in table 5.5. The interval boundaries are all placed at power of 2 values, which is done for two reasons: firstly this placement of the boundaries enables the more non-linear parts of the function to be approximated over narrower intervals. The narrower intervals make the ROM address widths roughly even for each of the four boundaries, which is desirable for the ROM construction. Secondly, the chosen intervals allow leading zeros to be omitted from the ROMs thus reducing the ROM's size. For each

of the four intervals a separate piecewise polynomial approximation is used. For a fraction width f of 8 to 10 bits a first order piecewise polynomial approximation is used for each of the four intervals and for a width of 11 to 23 bits a second order approximation is used.

First order approximation (5 to 7 bits)

For a fraction width f of 5 to 7 bits a single piecewise polynomial approximation is used to approximate the LNS addition function. Two coefficient ROMs, a single multiplier and a single adder are used to approximate the function. The ROMs are addressed directly with the input argument R and the outputs of the ROMs directly feed the multipliers and adders in the polynomial evaluation arithmetic. The approximation looks very similar to the structure in figure 5.14 except the addressing multiplexers and interval selection are not required.

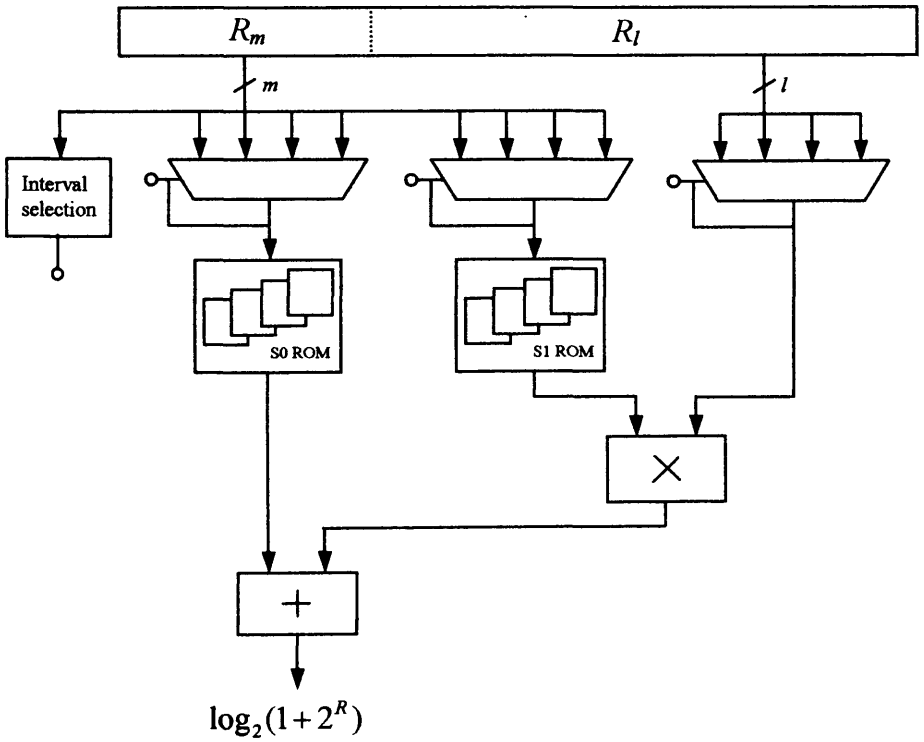


Figure 5.14. The first order approximation of the LNS addition function

First order approximation (8 to 10 bits)

Two ROMs (S0 and S1) are needed for a first order approximation, which are both addressed by the MSBs of the input argument to the function. Four separate first order approximations are needed and they can all share the same arithmetic hardware.

As the arithmetic is shared the ROMs need to be multiplexed to the arithmetic logic. For the first order approximation all the four S0 ROMs are grouped into one ROM and the input addressing is controlled by a multiplexer, which selects the input address depending on the interval the input argument is in. A similar scheme is used for the S1 ROM. The LSBs of the input argument that are passed to the approximation arithmetic also differs for each of the four intervals therefore a multiplexer is used to select the correct LSB slice. Hardware sharing is not a problem providing careful attention is paid to the weighting of all bits and the width of all data paths so that the function output is correct. Where a signal is not long enough for a particular data path it must be padded with zeros in the correct place. The first order approximation hardware is shown in figure 5.14.

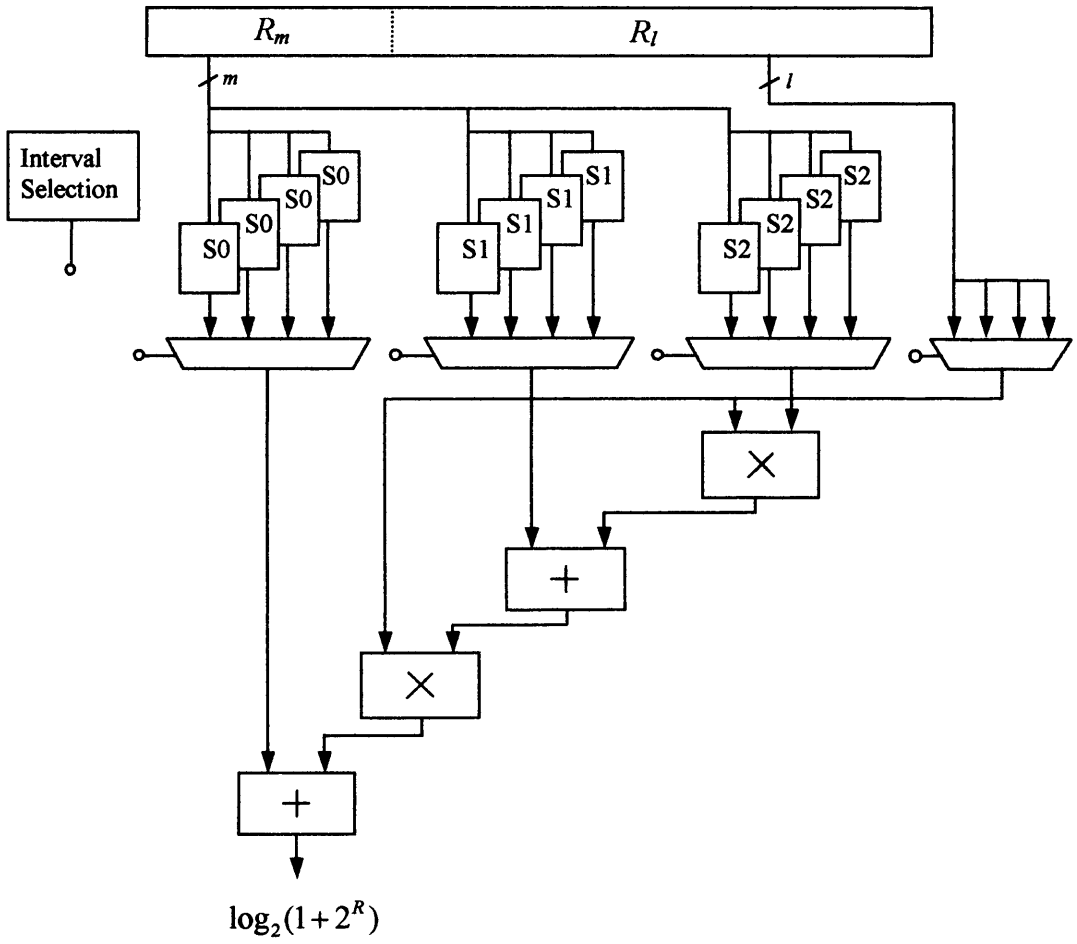


Figure 5.15. Second order approximation of the LNS addition function

Second order approximation (11 to 23 bits)

The second order approximation needs three ROMs (S0, S1 and S2). For 11 to 17 bits the ROM structure and addressing is similar to that used for the first order approximation for 8 to 10 bits. However, above 17 bits an alternative addressing and ROM structure is used that is more area efficient. As there are four intervals there are four different S0, S1 and S2 ROMs. If the ROMs that are grouped each require different address widths to achieve the required accuracy then grouping the ROMs into one is inefficient since the address width of some ROMs must increase beyond the minimum required width. An alternative scheme to the ROM grouping is to keep each ROM separate and multiplex the output to select the required interval coefficients. The input address value is hardwired for each coefficient ROM. Figure 5.15 shows the hardware structure of the second order approximation.

5.9.2.3 Subtraction function approximation

f	domain	Input width	Domain split
3	(-5, 0]	6	(-(*f*+2), 0]
4	(-6, 0]	7	
5	(-7, 0]	8	(-(*f*+2), -1] (-1, 0]
6	(-8, 0]	9	
7	(-9, 0]	11	
8	(-10, 0]	12	(-(*f*+2), -8] (-8, -6] (-6, -4] (-4, -2] (-2, 0]
9	(-11, 0]	13	
10	(-12, 0]	14	
11	(-13, 0]	15	
12	(-14, 0]	16	
13	(-15, 0]	17	
14	(-16, 0]	18	
15	(-17, 0]	20	(-(*f*+2), -16] (-16, -8] (-8, -4] (-4, -2] (-2, 0]
16	(-18, 0]	21	
17	(-19, 0]	22	
18	(-20, 0]	23	
19	(-21, 0]	24	
20	(-22, 0]	25	
21	(-23, 0]	26	
22	(-24, 0]	27	
23	(-25, 0]	28	

Table 5.6. The domain, input widths and the domain splits of the subtraction function for various fraction widths

As stated the subtraction function domain for fraction widths greater than 4-bits is split into two parts. Following the method for addition full table lookup is used for fraction widths of 3 and 4-bits. A single piecewise polynomial approximation is used for fraction widths of 5 to 7-bits on the domain $R \in (-(f+2), -1]$. From 8 to 10-bits

and 11 to 23-bits first and second order approximations respectively are used, which have a four-interval domain split for the domain $R \in (-(f+2),-2]$. The four intervals the domain $R \in (-(f+2),-2]$ is split into for each fraction width is shown in table 5.6. For fraction widths of 5 to 7 bits the domain interval (-1, 0] is approximated using a single ROM that is addressed by the f fraction bits of the input argument. For fraction widths of 8 to 23 bits the domain interval (-2 ,0) is approximated using the identity given in equation (5.47), which requires a base-2 log and a correction function to be calculated in parallel. The correction function and LNS subtraction function never need to be calculated simultaneously so the arithmetic logic to calculate the LNS subtraction approximation can be shared between both functions. A multiplexer is used to select the ROMs that feed the polynomial evaluation hardware. The base-2 log function is calculated for the domain [1, 2) in parallel with the correction function using separate arithmetic and ROMs. If the argument is not in this interval it is left shifted until it is and the left shift quantity is subtracted from the approximation result.

5.9.2.4 A new idea

During the writing of this thesis it became apparent that the correction and base-2 log functions can share the same arithmetic due to the following observation. The correction function and base-2 logarithm functions are approximated by polynomials shown in (5.63) and (5.64) respectively.

$$corr \approx a_{corr}x^2 + b_{corr}x + c_{corr} \quad \text{(5.63)}$$

$$\log_2 \approx a_l x^2 + b_l x + c_l \quad \text{(5.64)}$$

The sum of the two equations that is required to evaluate the subtraction function for the domain (-2, 0) is given in (5.65).

$$corr + \log_2 \approx (a_{corr} + a_l)x^2 + (b_{corr} + b_l)x + (c_{corr} + c_l) \quad \text{(5.65)}$$

Providing the x terms are the same width and weighting the base-2 log and correction functions can be approximated using the same arithmetic just by adding the coefficients as shown by (5.65). There are trade offs in reducing the arithmetic logic, which include: Extra adders are required to add the coefficients, which will add to the

area; The ROM flexibility is lost because the address widths of both coefficient ROMs must be equal; The scheme adds another layer of complication to the design. However, the optimisation constrains the overall arithmetic logic of the logarithmic addition/subtraction function approximation to be that of a second/first order polynomial. The scheme can easily be pipelined keeping the single-issue cycle property. The scheme has not been included in the final design due to time restrictions and has been left as a future work.

5.9.2.5 Combining the addition/subtraction functions with hardware sharing

The addition/subtraction functions use the same order of approximation and the same number of function splits. The functions never need to be calculated simultaneously thus they can be combined and the arithmetic hardware can be shared. The complete addition/subtraction approximation including the approximation for the correction function is shown in figure 5.16 for the second order case.

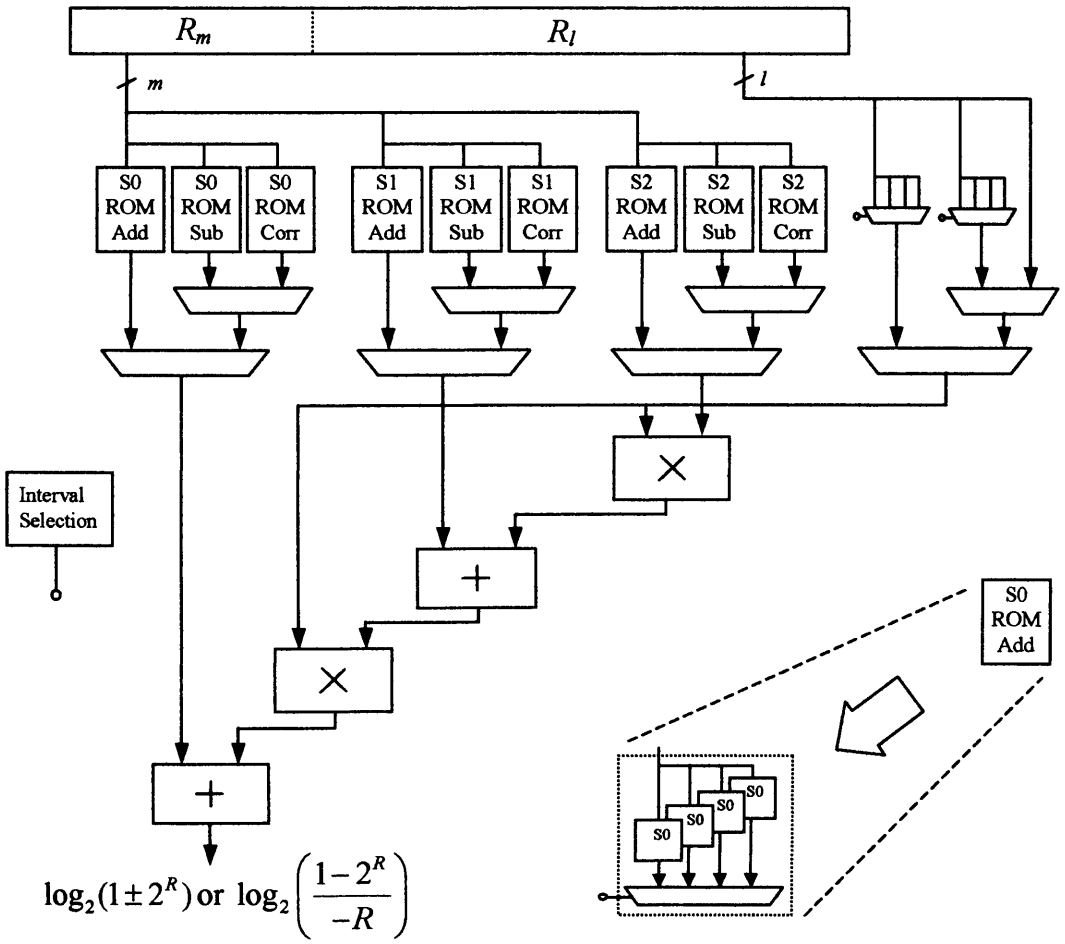


Figure 5.16. The addition/subtraction and correction function approximation

5.9.2.6 Utilising the Virtex-II FPGA on chip memory resource

The Virtex-II FPGA contains on chip memory blocks Xilinx [35], which can be configured in a 9-bit address, 36-bit content format. The 23-bit fraction LNS format requires the domain of the LNS addition function to be split into 4 sections, where each section is approximated with a 2^{nd} degree polynomial. The 2^{nd} order polynomial requires each section to have three coefficient ROMs S0, S1 and S2, where each has a maximum address width of 7-bits. The four S0 ROMs (one for each section) can be combined into one blockRAM component when it is configured in the 9-bit address 36-bit content format. The same can be done with the S1 and S2 ROMs. The same can also be done for the subtraction function, which can also accommodate the correction ROMs. This means that the entire add/sub/corr memory can fit into 6 blockRAMs for a 23-bit fractional implementation. The blockRAMs can be used for smaller fraction lengths but it results in inefficient implementations due to the address width not being used completely and therefore further use of the blockRAMs is not considered in this work.

5.9.2.7 The complete parallel-lookup addition/subtraction function

The complete parallel-lookup addition/subtraction design, which includes the approximation of figure 5.16, is shown in figure 5.17. The design shown in figure 5.17 consists of several components: There is a subtraction component that detects for integers larger than $f+1$ because when this occurs the output is to be set to zero. There is a NOR gate that detects for an input of zero because when this occurs the subtraction of two equal values is being performed and this case is signalled so the output can be forced to the special value of zero. The add/sub/corr component is the approximation shown in figure 5.16. The output of the add/sub/corr component can be negated or set to zero by the addSubZero component. The left shifter, $\log_2$ and subtracter components are used to calculate the base-2 logarithm function, which is required when the input is in the range (-2,0) and a subtraction operation is to be performed. Finally the addMux component adds either zero or the base-2 logarithm value to the add/sub/corr value.

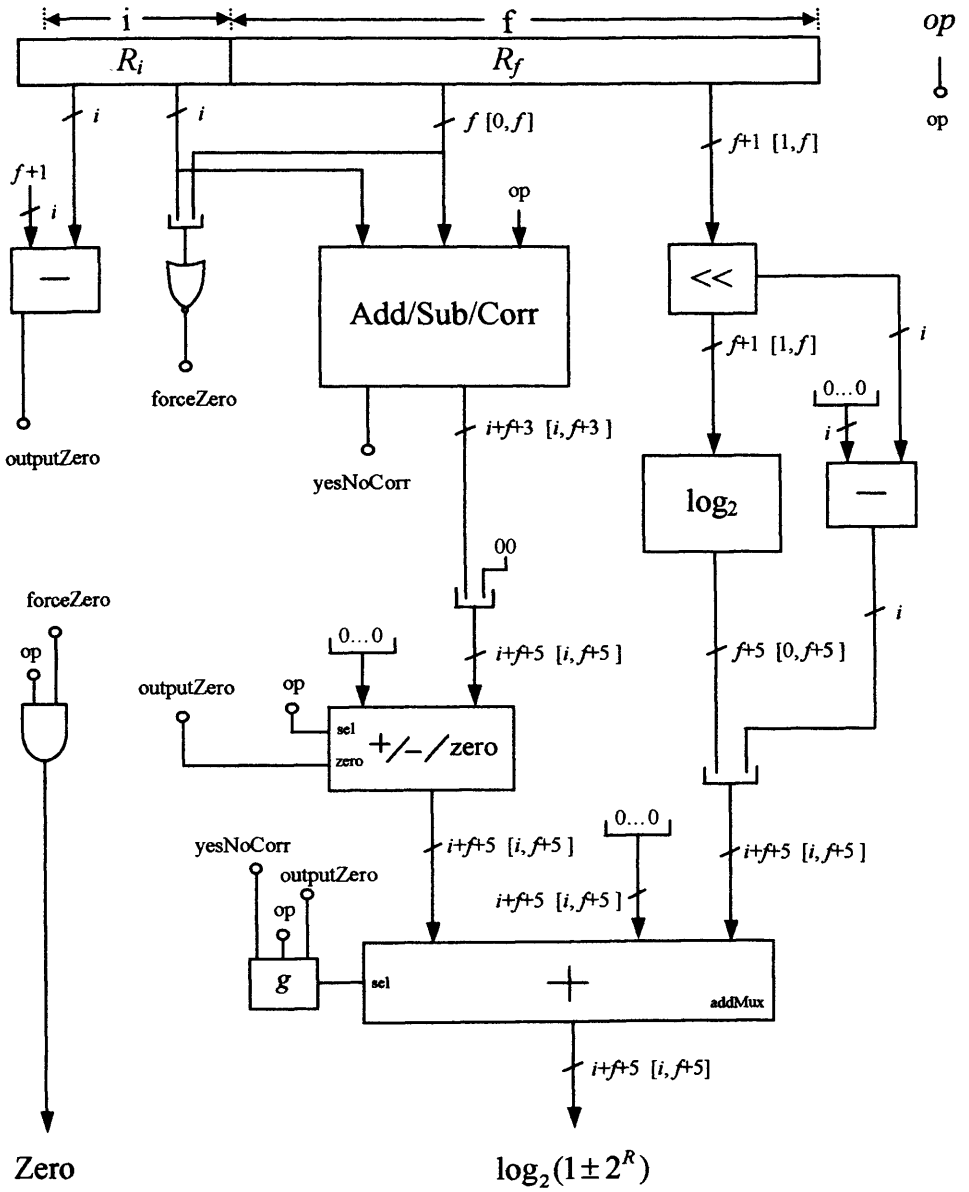


Figure 5.17. The complete parallel-lookup addition/subtraction function

5.9.2.8 Lookup table content and address widths for the parallel-lookup method

Base-2 logarithm

The base-2 log function is approximated for the domain [1, 2) and has a range of [0, 1). Table 5.7 shows the address width, content width, order of approximation, LSB width and the guard bit quantity for the base-2 logarithm function.

f	aw	S0w	S1w	S2w	lw	order	g
8	4	f+g	f+g+1	X	4	fst	5
9	5	f+g	f+g+1	X	4	fst	5
10	5	f+g	f+g+1	X	5	fst	5
11	6	f+g	f+g+1	X	5	fst	5
12	6	f+g	f+g+1	X	6	fst	5
13	7	f+g	f+g+1	X	6	fst	5
14	7	f+g	f+g+1	X	7	fst	5
15	4	f+g	f+g+1	f+g	11	sec	5
16	5	f+g	f+g+1	f+g	11	sec	5
17	5	f+g	f+g+1	f+g	12	sec	5
18	5	f+g	f+g+1	f+g	13	sec	5
19	6	f+g	f+g+1	f+g	13	sec	5
20	6	f+g	f+g+1	f+g	14	sec	5
21	7	f+g	f+g+1	f+g	14	sec	5
22	7	f+g	f+g+1	f+g	15	sec	5
23	7	f+g	f+g+1	f+g	16	sec	5

Table 5.7. Base-2 logarithm implementation details

Correction function

The correction function has a domain of (-2, 0) and a range of (-1.5, 0). Table 5.8 shows the address width, content width, order of approximation, LSB width and the guard bit quantity for the correction function.

f	aw	S0w	S1w	S2w	lw	order	g
8	4	f+g+1	f+g-1	X	5	fst	3
9	4	f+g+1	f+g-1	X	6	fst	3
10	5	f+g+1	f+g-1	X	6	fst	3
11	4	f+g+1	f+g	f+g-5	8	sec	3
12	4	f+g+1	f+g	f+g-5	9	sec	3
13	4	f+g+1	f+g	f+g-5	10	sec	3
14	4	f+g+1	f+g	f+g-5	11	sec	3
15	4	f+g+1	f+g	f+g-5	12	sec	3
16	4	f+g+1	f+g	f+g-5	13	sec	3
17	4	f+g+1	f+g	f+g-5	14	sec	3
18	4	f+g+1	f+g	f+g-5	15	sec	3
19	5	f+g+1	f+g	f+g-5	15	sec	3
20	5	f+g+1	f+g	f+g-5	16	sec	3
21	5	f+g+1	f+g	f+g-5	17	sec	3
22	5	f+g+1	f+g	f+g-5	18	sec	3
23	6	f+g+1	f+g	f+g-5	18	sec	3

Table 5.8. Correction function implementation details

LNS addition function

The LNS addition function has a domain of $(-(f+2), 0]$ and a range of $[0, 1]$. Table 5.9 shows the address width, content width, order of approximation, LSB width and the

guard bit quantity for the LNS addition function. For a fraction width of 8 to 17-bits the input domain is split, however the ROMS are grouped together and table 5.9 reflects this fact.

f	Domain	Input width	Integer width	aw	S0w	S1w	S2w	lw	order	g
3	(-5, 0]	6	3	6	f+1	X	X	0	zero	0
4	(-6, 0]	7	3	7	f+1	X	X	0	zero	0
5	(-7, 0]	8	3	4	f+g+1	f+g-1	X	4	fst	3
6	(-8, 0]	9	3	5	f+g+1	f+g-1	X	4	fst	3
7	(-9, 0]	11	4	6	f+g+1	f+g-1	X	5	fst	3
8	(-10, 0]	12	4	6	f+g+1	f+g-1	X	6	fst	3
9	(-11, 0]	13	4	6	f+g+1	f+g-1	X	7	fst	3
10	(-12, 0]	14	4	7	f+g+1	f+g-1	X	7	fst	3
11	(-13, 0]	15	4	5	f+g+1	f+g	f+g-3	10	sec	3
12	(-14, 0]	16	4	5	f+g+1	f+g	f+g-3	11	sec	3
13	(-15, 0]	17	4	5	f+g+1	f+g	f+g-3	12	sec	3
14	(-16, 0]	18	4	6	f+g+1	f+g	f+g-3	12	sec	3
15	(-17, 0]	20	5	7	f+g+1	f+g	f+g-3	13	sec	3
16	(-18, 0]	21	5	7	f+g+1	f+g	f+g-3	14	sec	3
17	(-19, 0]	22	5	7	f+g+1	f+g	f+g-3	15	sec	3

Table 5.9. LNS addition function implementation details

For fraction widths of 18 to 23-bits the ROMs are not combined and separate address and content widths are needed for each interval the domain is split into. Tables 5.10 and 5.11 show the details of the ROMs used when the fraction width is 18 to 23-bits.

f	Domain	Input width	Integer width	Domain split (-4, 0]				Domain split (-8, -4]				g
				aw	S0w	S1w	S2w	aw	S0w	S1w	S2w	
18	(-20, 0]	23	5	6	f+g+1	f+g	f+g-3	5	f+g-3	f+g-4	f+g-5	3
19	(-21, 0]	24	5	6	f+g+1	f+g	f+g-3	6	f+g-3	f+g-4	f+g-5	3
20	(-22, 0]	25	5	6	f+g+1	f+g	f+g-3	6	f+g-3	f+g-4	f+g-5	3
21	(-23, 0]	26	5	7	f+g+1	f+g	f+g-3	6	f+g-3	f+g-4	f+g-5	3
22	(-24, 0]	27	5	7	f+g+1	f+g	f+g-3	7	f+g-3	f+g-4	f+g-5	3
23	(-25, 0]	28	5	7	f+g+1	f+g	f+g-3	7	f+g-3	f+g-4	f+g-5	3

Table 5.10. LNS addition function implementation details

f	Domain	Input width	Integer width	Domain split (-16, -8]				Domain split (-(f+2), -16]				g
				aw	S0w	S1w	S2w	aw	S0w	S1w	S2w	
18	(-20, 0]	23	5	5	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3
19	(-21, 0]	24	5	5	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3
20	(-22, 0]	25	5	6	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3
21	(-23, 0]	26	5	6	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3
22	(-24, 0]	27	5	6	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3
23	(-25, 0]	28	5	7	f+g-7	f+g-8	f+g-9	4	f+g-15	f+g-15	f+g-16	3

Table 5.11. LNS addition function implementation details

LNS subtraction function

The LNS subtraction function has a domain of (-(f+2), 0) and a range of (-(f+1), 0]. Table 5.12 shows the address width, content width, order of approximation, LSB width and the guard bit quantity for the LNS subtraction function. The various domains for different fraction widths are shown in the table. For a fraction width of 8 to 17-bits the input domain is split, however the ROMs are grouped together and table 5.12 reflects this fact.

f	Domain	Input width	Integer width	aw	S0w	S1w	S2w	lw	order	g
3	(-5, 0]	6	3	6	5	X	X	0	zero	0
4	(-6, 0]	7	3	7	7	X	X	0	zero	0
5	(-7, -1]	8	3	5	f+g+1	f+g-1	X	3	fst	3
6	(-8, -1]	9	3	6	f+g+1	f+g-1	X	3	fst	3
7	(-9, -1]	11	4	7	f+g+1	f+g-1	X	4	fst	3
8	(-10, -2]	12	4	6	f+g-1	f+g-1	X	6	fst	3
9	(-11, -2]	13	4	6	f+g-1	f+g-1	X	7	fst	3
10	(-12, -2]	14	4	7	f+g-1	f+g-1	X	7	fst	3
11	(-13, -2]	15	4	5	f+g-1	f+g-1	f+g-2	10	sec	3
12	(-14, -2]	16	4	5	f+g-1	f+g-1	f+g-2	11	sec	3
13	(-15, -2]	17	4	6	f+g-1	f+g-1	f+g-2	11	sec	3
14	(-16, -2]	18	4	6	f+g-1	f+g-1	f+g-2	12	sec	3
15	(-17, -2]	20	5	6	f+g-1	f+g-1	f+g-2	14	sec	3
16	(-18, -2]	21	5	7	f+g-1	f+g-1	f+g-2	14	sec	3
17	(-19, -2]	22	5	7	f+g-1	f+g-1	f+g-2	15	sec	3

Table 5.12. LNS subtraction function implementation details

For fraction widths of 18 to 23-bits the ROMs are not combined and separate address and content widths are needed for each interval the domain is split into. Tables 5.13 and 5.14 show the details of the ROMs used when the fraction width is 18 to 23-bits.

f	Domain	Input width	Integer width	Domain split (-4, -2]				Domain split (-8, -4]				g
				aw	S0w	S1w	S2w	aw	S0w	S1w	S2w	
18	(-20, 0]	23	5	5	f+g-1	f+g-1	f+g-2	5	f+g-3	f+g-3	f+g-5	3
19	(-21, 0]	24	5	6	f+g-1	f+g-1	f+g-2	6	f+g-3	f+g-3	f+g-5	3
20	(-22, 0]	25	5	6	f+g-1	f+g-1	f+g-2	6	f+g-3	f+g-3	f+g-5	3
21	(-23, 0]	26	5	6	f+g-1	f+g-1	f+g-2	6	f+g-3	f+g-3	f+g-5	3
22	(-24, 0]	27	5	6	f+g-1	f+g-1	f+g-2	7	f+g-3	f+g-3	f+g-5	3
23	(-25, 0]	28	5	7	f+g-1	f+g-1	f+g-2	7	f+g-3	f+g-3	f+g-5	3

Table 5.13. LNS subtraction function implementation details

f	Domain	Input width	Integer width	Domain split (-16, -8]				Domain split (-(f+2), -16]				g
				aw	S0w	S1w	S2w	aw	S0w	S1w	S2w	
18	(-20, 0]	23	5	5	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3
19	(-21, 0]	24	5	5	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3
20	(-22, 0]	25	5	6	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3
21	(-23, 0]	26	5	6	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3
22	(-24, 0]	27	5	6	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3
23	(-25, 0]	28	5	7	f+g-7	f+g-7	f+g-9	4	f+g-15	f+g-15	f+g-17	3

Table 5.14. LNS subtraction function implementation details

Now a method to calculate the LNS addition/subtraction function with the required accuracy has been determined attention can be focused on the other parts of the LNS addition/subtraction component.

5.9.3 An addition or subtraction operation?

The XOR of the operand's sign bits and the input operator, which has the convention of being '1' for subtract and '0' for add, determines whether an addition or subtraction operation is required.

5.9.4 Calculating the largest magnitude and the difference

The sign bit of the subtraction of the two input operand magnitudes can be used to determine the largest and the smallest magnitudes. By connecting the sign bit to two multiplexers the largest and smallest values can be chosen. The difference can be found by using a subtraction component.

5.9.5 Sign logic

The output sign is determined by the operator, the signs of the two inputs and by which operand has the largest magnitude.

5.9.6 Special value detection

The five special values are encoded using reserved integers and can be detected for by using wide logic gates and single LUTs configured as 4 input functions. Figure 5.18 shows the logic to detect for the five special values of NaN, Inf, Normal, Denormal and Zero.

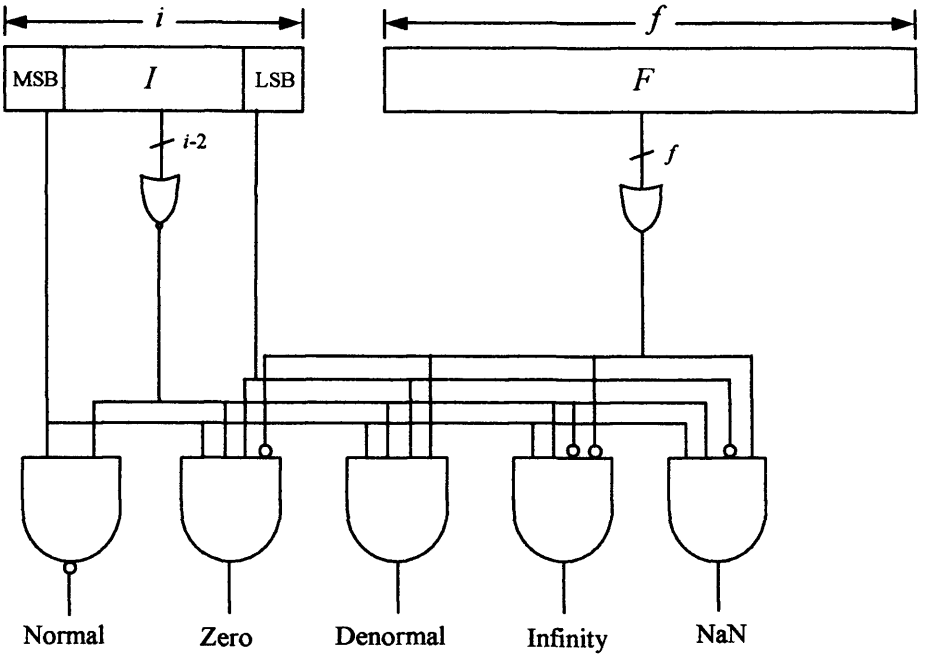


Figure 5.18. LNS five special value detection component

5.9.7 Special value input and output combinations

The table of the output values depending on the different combinations of special input values is the same as that used for floating-point addition and the reader is referred to the table 3.5 of section 3.5.6.1.

5.9.8 Overflow/underflow detection

Overflow and underflow is determined by checking the final result.

5.9.8.1 Overflow

The largest magnitude operand (remember the magnitude is a two's complement number) is "01...1.1...1" and any value larger than this results in overflow. Adding two of the largest magnitude operands results in the largest result of an addition which is "(0)10...0.1...1". The bracketed bit is an extra bit added to determine the sign of the overflow. Overflow is therefore detected by checking for an integer pattern of "(0)10...0". When overflow is detected the output is forced to infinity.

5.9.8.2 Underflow

The smallest permissible magnitude operand is "10...010.0...0" and any value smaller than this results in underflow. The smallest value that can be the result of a

subtraction depends on the number of fraction bits. Underflow is detected by the following integer bit patterns:

```
(1)10...01
(1)10...00
(1)0X...XX
```

The output is forced to zero if underflow is detected.

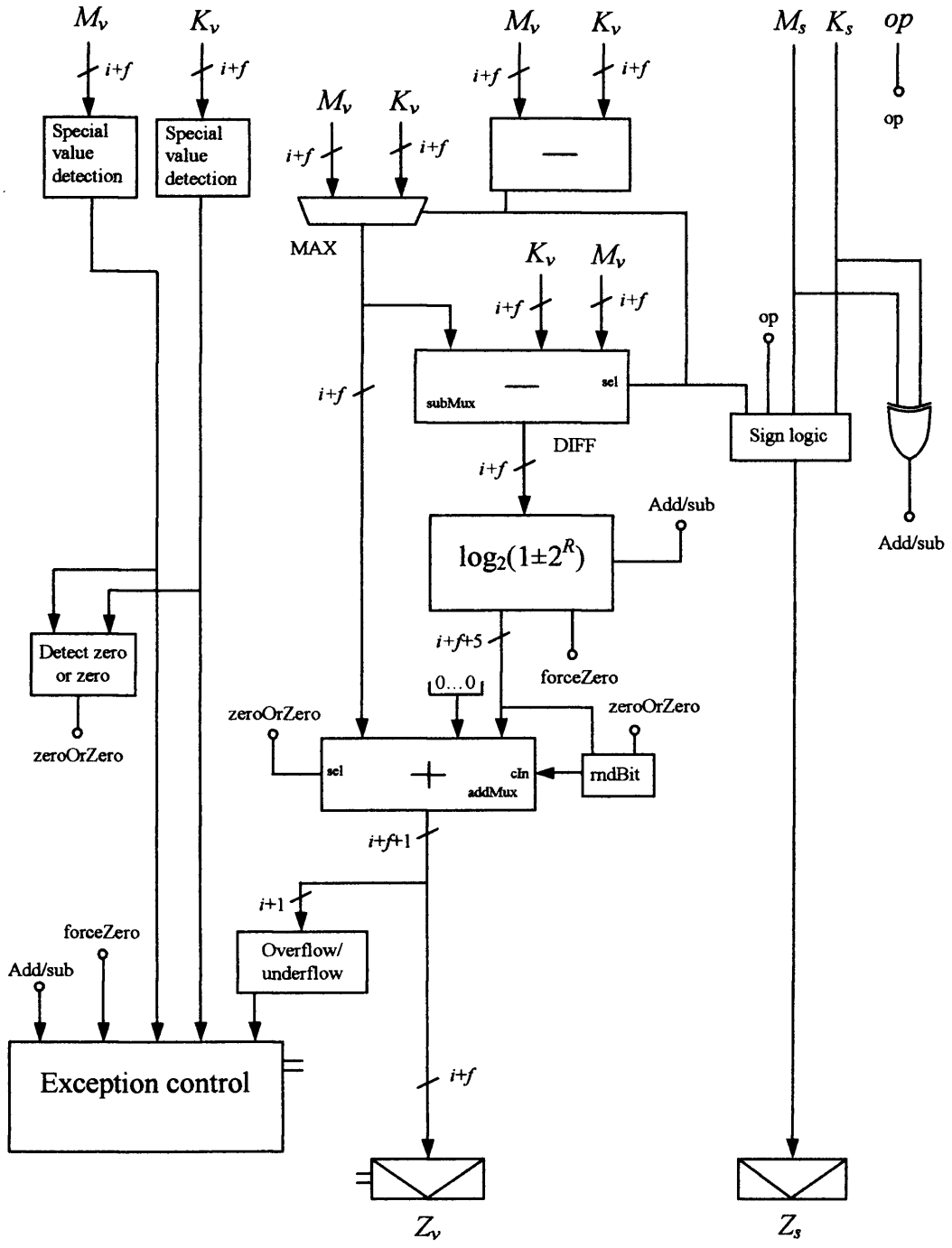


Figure 5.19. Complete logarithmic number system adder diagram

5.9.9 Complete LNS addition design

A block diagram of the complete LNS adder is shown in figure 5.19. The 'Detect zero or zero' component detects for either input being zero because if this occurs the output is set to the input that is not zero (providing an exception case is not detected). The 'rndBit' component generates the rounding bit that feeds the addMux component to round the output using the round-to-nearest scheme. If one input operand is zero then the output of the addMux does not need to be rounded hence the 'rndBit' component depends on the 'zeroOrZero' value.

5.10 Logarithmic number system multiplication

Multiplication is a very simple operation in the LNS as it is simply a fixed-point addition operation. Consider two LNS values K and M given by equations (5.66) and (5.67) respectively.

$$K = \log_2(X) \quad \text{(5.66)}$$

$$M = \log_2(Y) \quad \text{(5.67)}$$

Equations (5.68) and (5.69) show the linear domain multiplication operation and corresponding logarithmic domain multiplication operation respectively.

$$X * Y = 2^K * 2^Y \quad \text{(5.68)}$$

$$\log_2(X * Y) = \log_2(2^K * 2^M) = K + M \quad \text{(5.69)}$$

The LNS word format consists of two fields. To perform the full multiplication operation the sign bits are XORed and the magnitude bits are added using a fixed-point adder.

5.10.1 LNS multiplication error

As two values are added using fixed-point arithmetic there are no bits generated below the input LSB position and therefore the result does not need to be rounded. Furthermore the LNS multiplication is an exact operation and does not add any error to the computation except in the cases of underflow and overflow.

5.10.2 Special value detection and handling

Special values are detected as for LNS addition. The special value handling is the same as for floating-point multiplication and the reader is referred to table 3.8 in section 3.61.

5.10.3 Overflow/Underflow detection and handling

Overflow and underflow are determined by checking the result of the fixed-point addition. The adder is extended by one extra bit so the extended output integer part can be used to check for overflow and underflow. In the following the bracketed term is the extended bit.

5.10.3.1 Overflow

An output integer pattern range of (5.70) signals that overflow has occurred.

[(0)11...11, (0)10...00] ___(5.70)

Therefore overflow is determined by checking the top two bits, where "(0)1" signals that overflow has occurred. If overflow is detected the output is set to infinity.

5.10.3.2 Underflow

An integer bit pattern in the range of (5.71) signals that underflow has occurred.

[(1)10...0001, (1)00...0100] ___(5.71)

Therefore underflow is signalled by the following integer bit patterns:

(1)10...0X
(1)0X...XX

The underflow bit patterns can be detected by checking the top two bits and taking the logical OR of the middle i-2 bits to detect for a group of i-2 zeros. If underflow is detected the output is set to zero.

5.10.4 Complete LNS multiplier diagram

The complete logarithmic number system multiplier design is shown in figure 5.20.

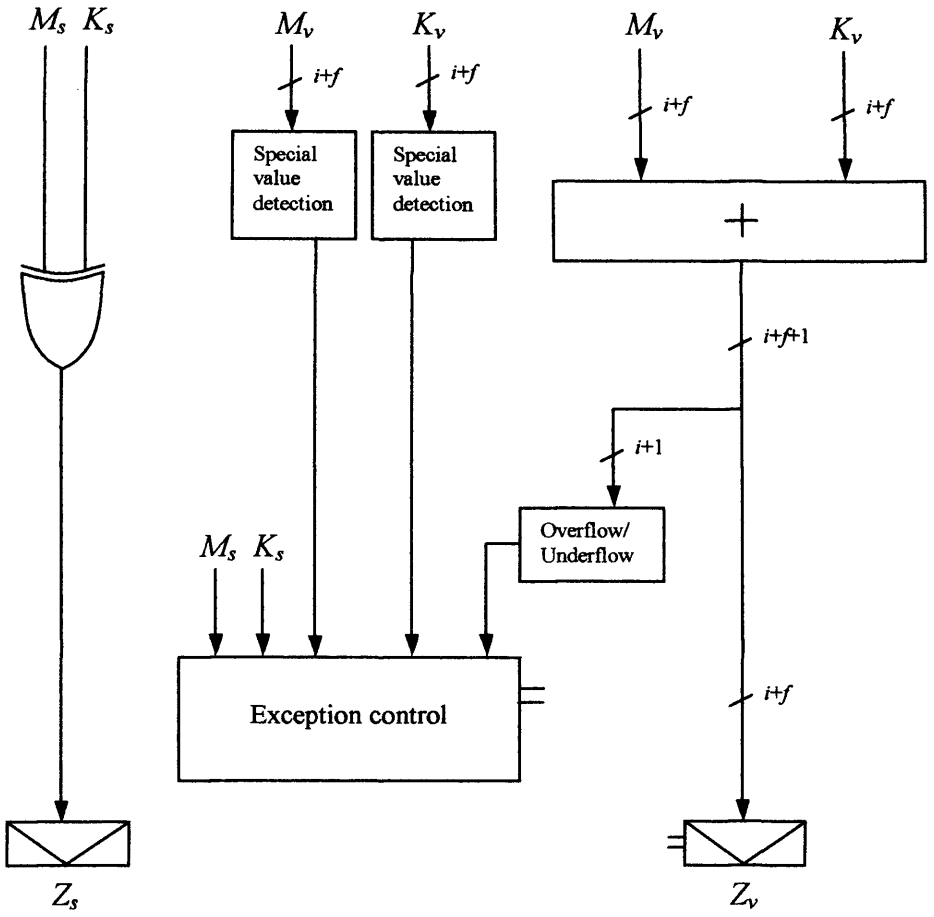


Figure 5.20. Complete logarithmic number system multiplier diagram

5.11 Logarithmic number system division

Division is a very simple operation in the logarithmic number system as it is simply a fixed-point subtraction operation. Consider the two LNS values given by equations (5.72) and (5.73) respectively.

$$K = \log_2(X) \quad \text{(5.72)}$$

$$M = \log_2(Y) \quad \text{(5.73)}$$

Equations (5.74) and (5.75) show the linear domain division operation and corresponding logarithmic domain operation.

$$\frac{X}{Y} = \frac{2^K}{2^M} \quad \text{(5.74)}$$

$$\log_2\left(\frac{X}{Y}\right)=\log_2\left(\frac{2^K}{2^M}\right)=K-M \qquad (5.75)$$

The LNS word format consists of two fields. To perform the full division operation the sign bits are XORed and the magnitude parts are subtracted using a fixed-point subtracter.

5.11.1 LNS division error

As two magnitudes are subtracted using a fixed-point subtracter no bits below the LSB position are generated and therefore the result does not need to be rounded. The LNS division operation is exact except in the cases of overflow and underflow.

5.11.2 Special value detection and handling

Special values are detected as for LNS addition. The special value handling is the same as for floating-point division and the reader is referred to the table 3.9 in section 3.7.1.

5.11.3 Overflow/underflow detection and handling

The subtraction component is extended by one bit so that the sign of the resultant value can be determined if overflow or underflow occur. The extended bit is shown in brackets in the following.

5.11.3.1 Overflow

An integer bit pattern range of (5.76) signals that overflow has occurred.

$$[(0)10...000, (0)11...101] \qquad (5.76)$$

Therefore overflow is determined by checking the top two bits where "(0)1" indicates that overflow has occurred. If overflow is detected the output is set to infinity.

5.11.3.2 Underflow

An integer bit pattern range of (5.77) signals that underflow has occurred.

$$[(1)10...001, (1)00...010] \qquad (5.77)$$

Therefore underflow is determined by checking for the following bit patterns:

(1)10...0X
(1)0X...XX

The underflow bit patterns can be detected by checking the top two bits and taking the logical OR of the middle i-2 bits to detect for a group of i-2 zeros. If underflow is detected the output is set to zero.

5.11.4 Complete LNS divider diagram

The complete logarithmic number system divider design is shown in figure 5.21.

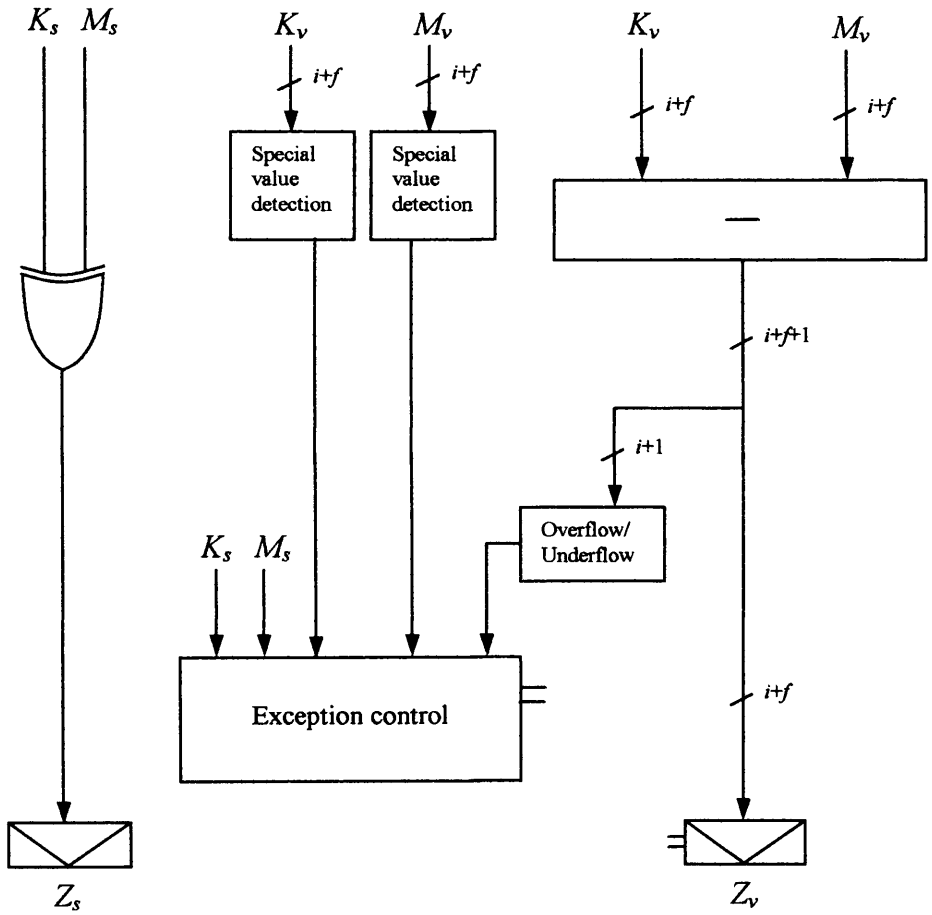


Figure 5.21. Complete logarithmic number system divider diagram

5.12 Logarithmic number system powering

Powering is a simple fixed-point multiplication operation in the LNS. Consider the LNS value K given by (5.78).

$$K = \log_2(X) \quad \text{(5.78)}$$

The linear and corresponding logarithmic operation of powering is given by equations (5.79) and (5.80).

$$X^P = \left(2^K\right)^P \quad \text{(5.79)}$$

$$\log_2\left(X^P\right) = \log_2\left(\left(2^K\right)^P\right) = P * \log_2\left(2^K\right) = P * K \quad \text{(5.80)}$$

5.12.1 Powering error

The powering operation is exact for integer powers and can be correctly rounded for powers with fractional sections. The exact rounding commits at worst the worst-case relative rounding error that was shown in equation (5.11) in section 5.5.

5.13 Logarithmic number system square root

P equal to 0.5 is the special case powering operation of square root. From (5.80) it can be seen that in the logarithmic domain the square root operation is a multiplication by 0.5, which is a single bit right shift.

5.13.1 LNS square root rounding

The single bit right shift means a single bit below the LSB is created that must be rounded. The rounding can be done by truncation because at most a ½ ulp error will be committed in the truncation as only a single bit is being discarded. Furthermore a rounding adder is not required.

5.13.2 Special value detection and handling

Special values are detected in the same way as for LNS addition. The special values are handled in the same way as for floating-point square root and the reader is referred to the table 3.12 of section 3.8.1.

5.13.3 Overflow/underflow

The result of a square root operation cannot overflow or underflow.

5.13.4 The complete LNS square root diagram

The complete logarithmic number system square root operator design is shown in figure 5.21. The right shift operation is hardwired and the LSB is discarded. The

result of the right shift is sign extended so the output has the correct number of bits. The sign bit remains unchanged.

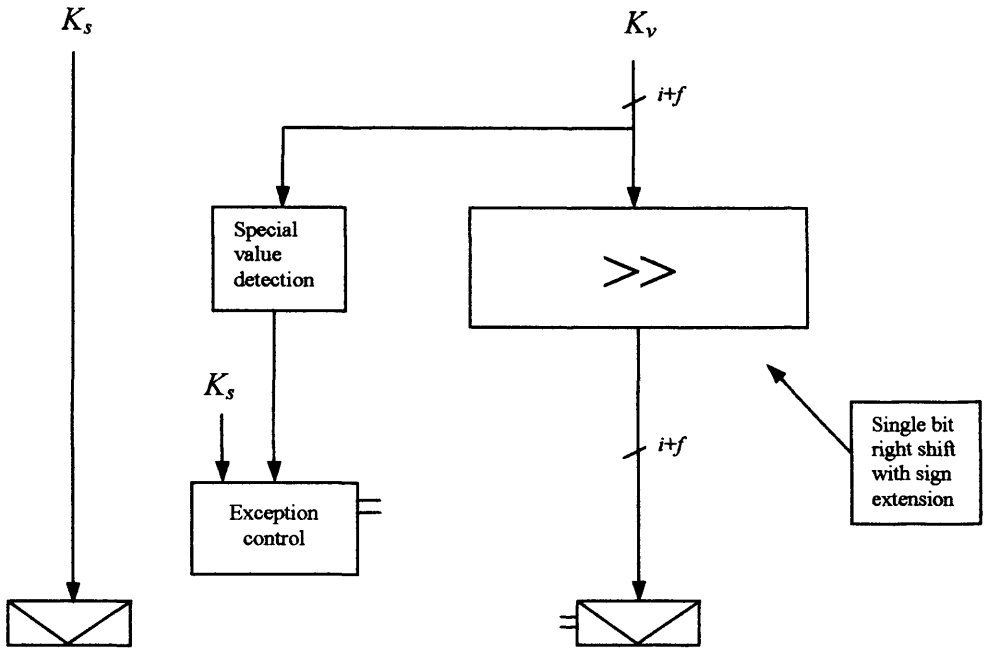


Figure 5.22. Complete logarithmic number system square root diagram

5.14 LNS implementation results

All the logarithmic number system operators have been designed to be parameterisable in terms of the integer and fraction widths and these values are set at design time. In this section the implementation results for the LNS addition, multiplication, division and square root operators are given and are compared with other published designs. LNS designs are not as common as floating-point as only seven other authors give design statistics for LNS components. Similar area and delay calculating techniques as used for floating-point are used here. Two sets of design results are presented for LNS addition to compare the dual-path and parallel lookup methods.

5.14.1 LNS addition results

Table 5.15 summarises the dual-path LNS addition Xilinx XC2V1000-4 Virtex-II FPGA implementation results using version 5.1.03i of the Xilinx place and route tools

and technology speed files created by Xilinx on 01/11/2002. Table 5.16 summarises the parallel-lookup LNS addition implementation results.

(i, f)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Embed Mults	Delay (ns)
4, 5	141	10/271	41.4	122	10/233	1	41.7
4, 7	249	12/440	46	214	12/367	1	43.6
4, 9	288	14/556	57.9	202	14/382	2	55.6
4, 11	426	16/827	60.5	310	16/592	2	58.8
6, 9	296	16/572	59	208	16/398	2	57.6
6, 11	436	18/844	61	320	18/609	2	58.1
6, 13	682	20/1250	65.2	558	20/980	2	67.3
6, 15	914	22/1727	100.5	486	22/911	4	100.2
6, 17	1103	24/2099	101	617	24/1155	4	102.2
6, 19	1404	26/2688	105.9	871	26/1646	4	108.1
8, 11	442	20/858	61.7	326	20/623	2	62
8, 13	688	22/1264	66.8	564	22/994	2	67.9
8, 15	921	24/1741	101.3	492	24/925	4	102.5
8, 17	1110	26/2113	102.6	623	26/1169	4	103.2
8, 19	1409	28/2702	106.3	877	28/1660	4	107.9
8, 21	1776	30/3373	118.4	1213	30/2280	4	118.8
8, 23	2170	32/4097	123.6	1608	32/3016	4	125.5

Table 5.15. Dual-path LNS addition implementation results

(i, f)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Embed Mults	Delay (ns)
4, 5	133	10/249	26.5	87	10/169	1	27.6
4, 7	195	12/340	29.1	171	12/298	1	28
4, 9	307	14/586	38.3	259	14/480	2	38.2
4, 11	504	16/962	55.5	318	16/613	3	47.6
6, 9	321	16/605	39	271	16/500	2	39.4
6, 11	518	18/987	56.5	337	18/644	3	49
6, 13	741	20/1385	58	495	20/917	3	50.8
6, 15	1053	22/1840	61.5	635	22/1033	4	58
6, 17	1368	24/2317	64.9	909	24/1433	4	62.7
6, 19	1839	26/3526	69.9	1358	26/2585	4	69.4
8, 11	528	20/1005	57.5	348	20/660	3	49.5
8, 13	753	22/1404	59.2	504	22/934	3	51.6
8, 15	1063	24/1857	62	644	24/1049	4	59.4
8, 17	1380	26/2335	66.3	921	26/1450	4	64
8, 19	1848	28/3543	72.3	1367	28/2601	4	70.1
8, 21	2498	30/4712	73.9	1985	30/3710	4	75.3
8, 23	3607	32/6567	74.9	3062	32/5511	4	76

Table 5.16. Parallel-lookup LNS addition implementation results

Due to time restrictions efficient pipelining of the designs has not been undertaken and so results are only given for single cycle designs. The pipelining study has been left as a future work. Two sets of design results are shown in tables 5.15 and 5.16. One set where the design uses only slices and another where slices and embedded multipliers are used. From tables 5.15 and 5.16 it can be seen that the area and delay is mostly dependent on the fraction length and due to this the comparison of the two methods will only be made for any one fractional width i.e. only (8, 11) (say) not (8, 11) and (6, 11). Figure 5.23 compares the area of the two LNS addition methods. Figure 5.23 compares only the implementations that do not use the embedded multipliers. The area results that use the embedded multipliers are shown in figure 5.24. The delay results are shown in figures 5.25 and 5.26 for the implementations that do not and do use the embedded multipliers respectively.

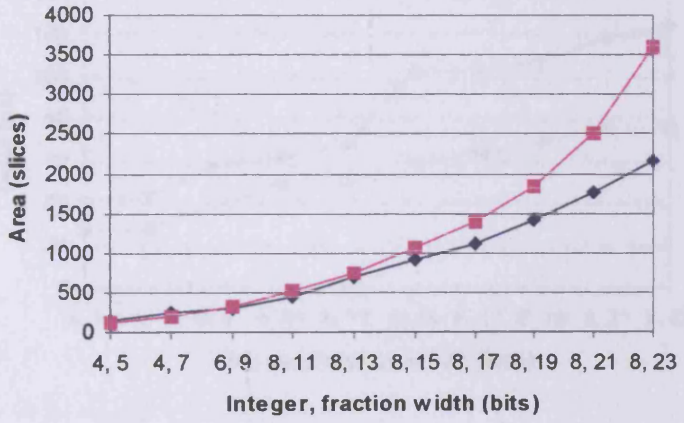


Figure 5.23. The area results of the slice only dual-path [♦] and parallel-lookup [■] LNS addition methods

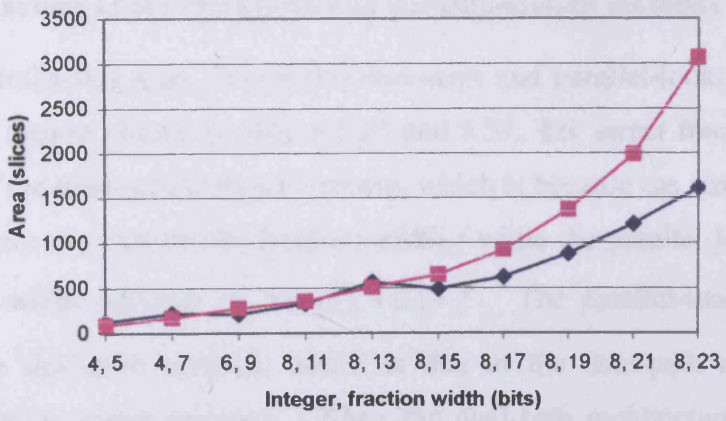


Figure 5.24. The area results of the slice and embedded multiplier dual-path [♦] and parallel-lookup [■] LNS addition methods

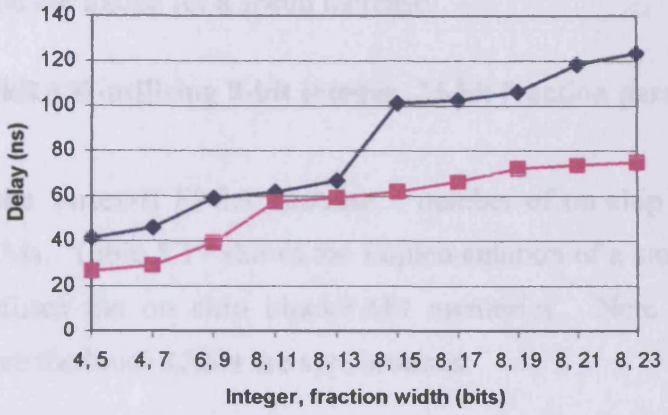


Figure 5.25. The delay results of the slice only dual-path [♦] and parallel-lookup [■] LNS addition methods

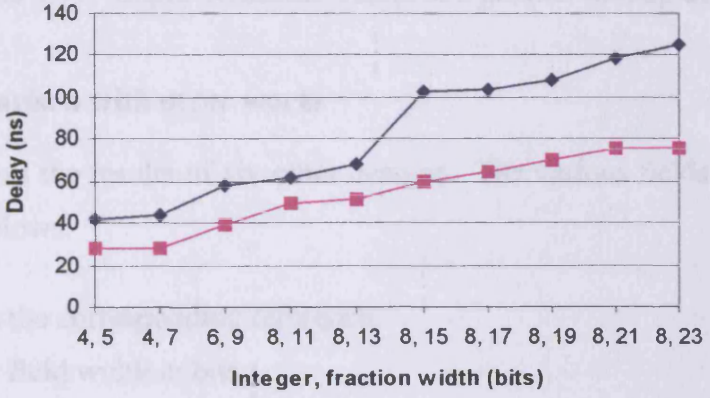


Figure 5.26. The delay results of the slice and embedded multiplier dual-path [♦] and parallel-lookup [■] LNS addition methods

5.14.1.1 Comparison of the dual-path and parallel-lookup methods

For fraction widths less than 15-bits the dual-path and parallel-lookup methods are very similar in area as shown in figures 5.23 and 5.24. For larger fraction widths the area savings of the dual-path method increase, which is because the size of function to approximate only depends on the fraction width f while the parallel-lookup function approximation width depends on $\log_2\lceil f+2\rceil+f$. The parallel-lookup method is faster than the dual-path method, which is due to the dual-path method having sequential function approximations. When the dual-path architecture contains two sequential 2^{nd} order function approximations at a fraction length of 15-bits the

difference in speed of the two methods is substantial. Clearly the area savings of the dual-path method are traded for a speed increase.

5.14.1.2 A blockRAM-utilising 8-bit integer, 23-bit fraction parallel-lookup design

As mentioned the Virtex-II FPGA contains a number of on chip memory resources called blockRAMs. Table 5.17 shows the implementation of a single parallel-lookup method that utilises the on chip blockRAM memories. Note that the design is pipelined because the blockRAMs are synchronous.

(i, f)	Area (slices)	bRAMs	Embed Mults	Cycles	Delay (ns)	Cycles* Delay
8, 23	1210	6	4	5	25	125

Table 5.17. Implementation results of a parallel-lookup design

5.14.1.3 Comparison with other works

Table 5.18 shows the results of six other designs. The various fields of the table are described as follows:

Author : this is the corresponding reference.
i : is the integer field width in bits.
f : is the fraction field width in bits.
Add/Sub : indicates whether the design can do addition or subtraction only or both.
Error : this describes the error allowed for the addition/subtraction function. There are two choices either BTFP (better-than-floating-point) or faithful.
Slices : this is the equivalent number of Virtex FPGA slices used.
bRAMs : is the number of blockRAM resources used.
Embed Mults : is the number of on chip multipliers used in the design.
Delay : this is the delay of a single clock cycle.
Cycles : this is the number of cycles it takes for data to pass through a pipelined design.
Delay*Cycles : this is the basic delay metric used.
Chip maker : this is the chip maker and corresponding part number the design is targeted for.
Year : is the year of publication of the reference.

Author	i	f	Add/ Sub	Error	Slices	bRAMs	Embed Mults	Delay (ns)	Cycles	Delay* Cycles	Chip maker	Year
UTIA[1] [301]	8	10	both	BTFP	1229	6	X	12.5	8	100	Xilinx XC2V1000-4	03
Matousek [299]	8	11	both	BTFP	720	4	X	20	8	160	Xilinx XCV2000E-6	02
Kadlec[2] [295]	8	23	both	BTFP	1728	X	X	18.9	12	226.8	Xilinx XCV2000E-6	01
Arnold [272]	8	23	add	Faithful	768	X	X	X	X	X	Xilinx XCV300	01
Lewis[3] [258]	8	23	add	BTFP	2243	X	X	X	X	X	Xilinx XCV300	01
Coleman[3] [268]	8	23	add	BTFP	4209	X	X	X	X	X	Xilinx XCV300	01
Matousek [299]	8	23	both	BTFP	1300	96	X	20	8	160	Xilinx XCV2000E-6	02
UTIA[1] [301]	8	23	both	BTFP	1843	28	X	14.3	8	114.4	Xilinx XC2V1000-4	03

[1] Can calculate two operations simultaneously
[2] Function lookup tables are stored in external RAM
[3] Estimated in Arnold [272]

Table 5.18. Some results on the FPGA implementation of the LNS addition operator

Most of the designs in table 5.18 are for an integer width of 8-bits and a fraction width of 23-bits, which is the LNS format equivalent to IEEE single precision. Each implementation will be discussed in turn. The 8-bit integer 10-bit fraction design of UTIA [301] can calculate 2 operations simultaneously but uses 1229 slices, which is more than double the amount required by the proposed parallel-lookup design. Furthermore the proposed design does not use any blockRAM memories. The 720 slice, 4 blockRAM design of Matousek [299] is larger than the proposed parallel-lookup design and uses a number of blockRAMs. Kadlec [295] uses off chip RAM to store the lookup tables for the LNS add/sub function, which causes a delay overhead and means that the number of addition operations that can be placed on a chip is limited to 1. Arnold [272] only gives details of the implementation of the addition function so a fair comparison is not possible. Arnold [272] also estimates the area for the addition functions of Lewis [258] and Coleman [268], but does not mention the subtraction function so again a fair comparison is not possible. The area for a 2^{nd} degree interpolator is given by Arnold [272] to be around 276 slices and we can use this figure to estimate the area of the addition function proposed in this work to compare with Arnold [272], Lewis [258] and Coleman [265]. The 2^{nd} order LNS addition function uses three 512-word memories (a 9-bit address memory), each of which are about 26-bits in length allowing for guard bits. 16-slices are needed to construct a 9-bit address 1-bit content memory so a total of 3*26*16 + 276 = 1524 slices are needed for the BTFP addition function. This is better that Lewis [258] and

Coleman [268] but worst than Arnold [272], which is due to the relaxed error criteria used by Arnold. The 8-bit integer, 23-bit fraction design of Matousek [299] is mapped to a Virtex-E and utilises 96 blockRAMs. The proposed design, given in table 5.17, which uses 6 blockRAMs is mapped to a Virtex-II FPGA. The blockRAM sizes of the two technologies is different and if mapped to a Virtex-E FPGA the design of table 5.17 would need approximately 19 blockRAMs, which is considerably less than the design of Matousek [299]. Finally the 32-bit design of UTIA [301] is mapped to a Virtex-II FPGA so a direct comparison with the design of table 5.17 is possible. UTIA [301] uses 28 blockRAMs, which is 70% of any resource of a 1M gate XC2V1000 FPGA and can calculate two operations simultaneously. The proposed 32-bit design in table 5.17 uses a maximum of 23% of the resources of a 1M gate XC2V1000 FPGA and so 4 addition/subtraction operations can be placed on a single chip. In general a greater than 2X increase in the number of operations that can be placed on a single FPGA is possible when using the proposed design of table 5.17 compared to that of UTIA [301].

5.14.1.4 Comparison with another parameterisable design

Table 5.19 shows the results of a parameterisable design given in Detrey [303], which is based on the multipartite function approximation technique. Detrey uses a BTFP accuracy bound for the addition and subtraction functions except for the subtraction function in the singularity region between -1 and 0. The relaxation of the error bound reduces the amount of hardware needed to do the approximation of the subtraction function and also increases the error to be worse than floating-point.

i	f	Add/Sub		Add/Sub pipelined			Chip
		Slices	delay	slices	delay	Cycles	
3	6	114	22	138	10	3	XC2V1000-4
4	7	192	24	219	10	3	XC2V1000-4
5	10	627	25	721	10	4	XC2V1000-4
8	11	1019	28	1133	10	4	XC2V1000-4
6	13	2697	35	2940	10	4	XC2V1000-4
8	15	X	X	X	X	X	XC2V1000-4
8	23	X	X	X	X	X	XC2V1000-4

Table 5.19. The LNS addition implementation results of Detrey [303]

Detrey [303] gives pipelined and non-pipelined results, but only the non-pipelined results will be considered for comparison purposes. Figure 5.27 compares the area of

the non-pipelined results of table 5.19 with the slice only results (i.e. not including the embedded multipliers) for the parallel lookup approximation shown in table 5.16. Figure 5.28 compares the delay of the two mentioned implementations. The comparisons are based solely on the fraction width, as the effect of varying integer width is relatively small. The largest integer width for a given fraction width is used in all cases to enable a fair comparison.

5.14.1.5 Discussion

For small fraction widths of less than 8-bits the area of Detrey's method is very similar to the proposed method and is slightly smaller for fraction widths less than 7-bits. Above 8-bits the area of Detrey's method rapidly increases, which is due to the fact that the multipartite method is a first order approximation constructed solely of ROMs and adders. Due to the exponential memory increase results above a fraction width of 13-bits are not given. Detrey's method cannot make use of the embedded multipliers while the proposed method can and if the embedded multipliers are used in the proposed design (table 5.16) it is always smaller. The major advantage of the multipartite method is the speed of the design. As figure 5.28 shows the design of Detrey is always faster than the proposed design. For small word lengths of less than 7-bits the delay of Detrey's design is about 17% less than the proposed design, however above 7-bits the delay saving increases and it peaks at about 50% for 11-bits. Above 11-bits the delay curves of Detrey's method and the parallel-lookup method start to converge.

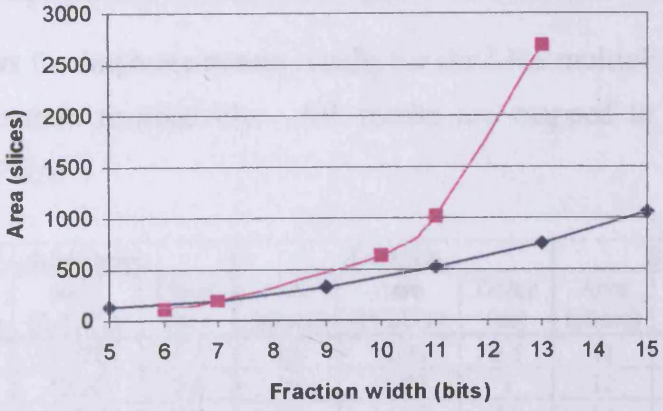


Figure 5.27. The variation of the area of the parallel-lookup method [♦] and the method of Detrey [■] with fraction width

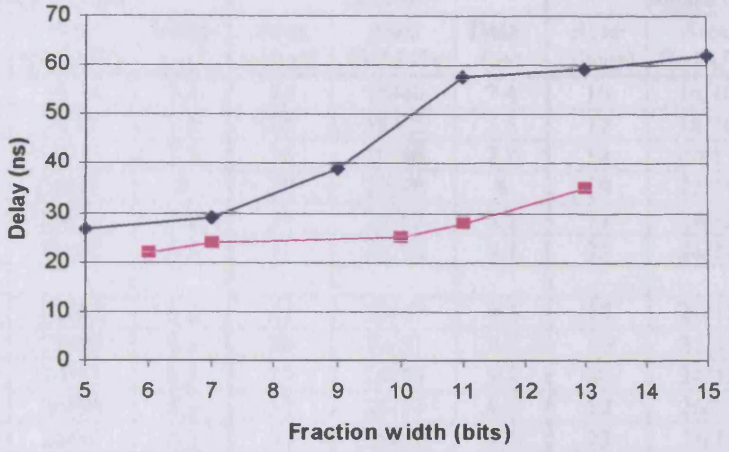


Figure 5.28. The variation of the delay of the parallel-lookup method [♦] and the method of Detrey [■] with fraction width

5.14.1.6 Conclusion

The proposed parallel-lookup method is superior in terms of area and comparable in terms of speed to all other published LNS addition/subtraction designs of equivalent accuracy shown in table 5.18. The fastest design is that of Detrey [303] and below 9 fraction bits the design of Detrey offers a superior area and delay combination although the accuracy of Detery's design is compensated when the subtraction function is evaluated in the singularity region. Above 7-bits the 'slice only' proposed parallel-lookup design is superior to the design of Detrey and furthermore can offer even greater savings by making use make use of the on chip embedded multipliers.

5.14.2 LNS multiplication, division and square root results

Table 5.20 shows the implementation results for the LNS multiplication, division and square root operators respectively. All results are mapped to a Xilinx Virtex-II XC2V1000-4 FPGA.

(i, f)	Multiplication			Division			Square root		
	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)
4, 5	16	10/28	7.1	18	10/31	6.8	11	10/8	4
4, 7	17	12/30	7.3	19	12/33	7	12	12/8	4.1
4, 9	20	14/34	7.4	22	14/37	7.2	14	14/9	4.6
4, 11	21	16/36	7.5	23	16/39	7.4	15	16/9	4.8

Table 5.20. LNS multiplication, division and square root implementation results

(i, f)	Multiplication			Division			Square root		
	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)
6, 9	23	16/39	7.4	24	16/40	7.4	16	16/10	4.7
6, 11	24	18/41	7.5	25	18/42	7.5	17	18/10	4.8
6, 13	25	20/45	7.8	26	20/46	7.8	18	20/11	4.8
6, 15	26	22/47	8	27	22/48	8	19	22/11	5
6, 17	29	24/51	8.2	30	24/52	8.2	21	24/12	5.2
6, 19	30	26/53	8.3	31	26/54	8.2	22	26/12	5.2
8, 11	26	20/45	8.2	27	20/47	8.3	18	20/11	4.9
8, 13	27	22/49	8.3	28	22/51	8.3	19	22/12	4.9
8, 15	28	24/51	8.5	29	24/53	8.3	20	24/12	5
8, 17	31	26/55	8.5	32	26/57	8.6	22	26/13	5
8, 19	32	28/57	8.5	33	28/59	8.6	23	28/13	5.1
8, 21	33	30/61	8.6	34	30/63	8.6	24	30/14	5.2
8, 23	34	32/63	8.7	35	32/65	8.8	25	32/14	5.2

Table 5.20. LNS multiplication, division and square root implementation results

5.14.2.1 Comparison with other works

Only UTIA [301] and Detrey [303] offer statistics for the multiplication, division and square root operators. Table 5.21 shows the implementation results of UTIA [301] for a Xilinx Virtex-II XC2V1000-4 FPGA and table 5.22 shows the implementation results of Detrey [303] for a similar FPGA.

(i, f)	Multiplication			Division			Square root		
	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)
8, 10	<1%	X	10	<1%	X	10	<1%	X	10
8, 23	<1%	X	10	<1%	X	10	<1%	X	10

Table 5.21. The LNS multiplication, division and square root implementation results of UTIA [301]

(i, f)	Multiplication			Division			Square root		
	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)	Area (slices)	Area ffs/LUTs	Delay (ns)
3, 6	9	X	6	9	X	6	1	X	2
4, 7	10	X	6	10	X	6	1	X	2
5, 10	12	X	7	12	X	8	1	X	2
8, 11	14	X	7	14	X	8	1	X	2
6, 13	14	X	7	14	X	8	1	X	2
8, 15	16	X	8	16	X	8	1	X	2
8, 23	20	X	8	20	X	8	1	X	2

Table 5.22. The LNS multiplication, division and square root implementation results of Detrey [303]

The multiplication, division and square root results shown in tables 5.20 and 5.22 are plotted on two graphs shown in figures 5.29 and 5.30. Figure 5.29 plots the variation of component area, in slices, with fraction width and figure 5.30 plots the variation of component delay, in nanoseconds, with fraction width.

5.14.2.2 Discussion

The proposed multiplication, division and square root designs are very similar in size to the other designs of UTIA [301] and Detrey [303], which is due to the simplicity of the implementation of these operations. The proposed designs are larger than those of Detrey, as figure 5.29 illustrates, which is probably caused by the way special values are handled and the fact that the results in this work are for designs with registers on their outputs. Detrey uses special bits to signal the special values of NaN, infinity and zero and so does not use any logic to detect for them or to force the output to a particular value if a particular pair of input values are detected. The area of the square root design of Detrey is just one slice, which means that the design results are not given for components with registers on their outputs which reduces the area of the designs.

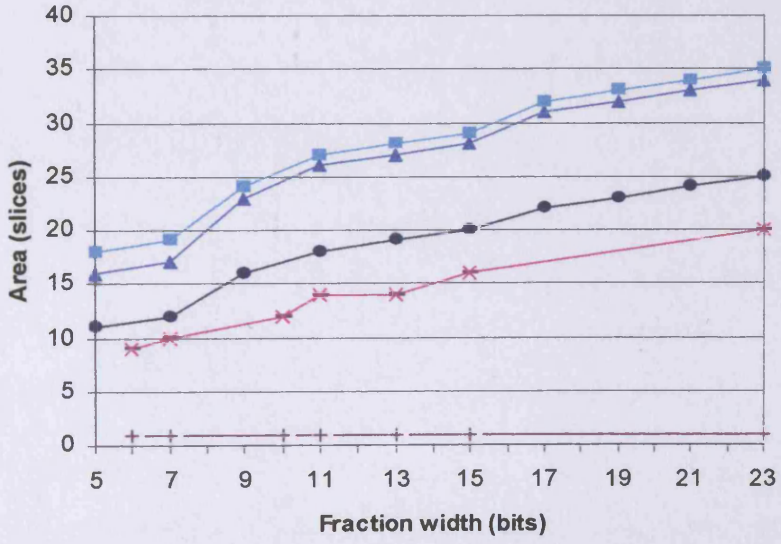


Figure 5.29. The variation of the area, in slices, of the proposed LNS division [■], LNS multiplication [▲] and LNS square root [●] operators with fraction width. The variation of the area of the LNS division [-], LNS multiplication [x] and LNS square root [+] operators of Detrey with fraction width.

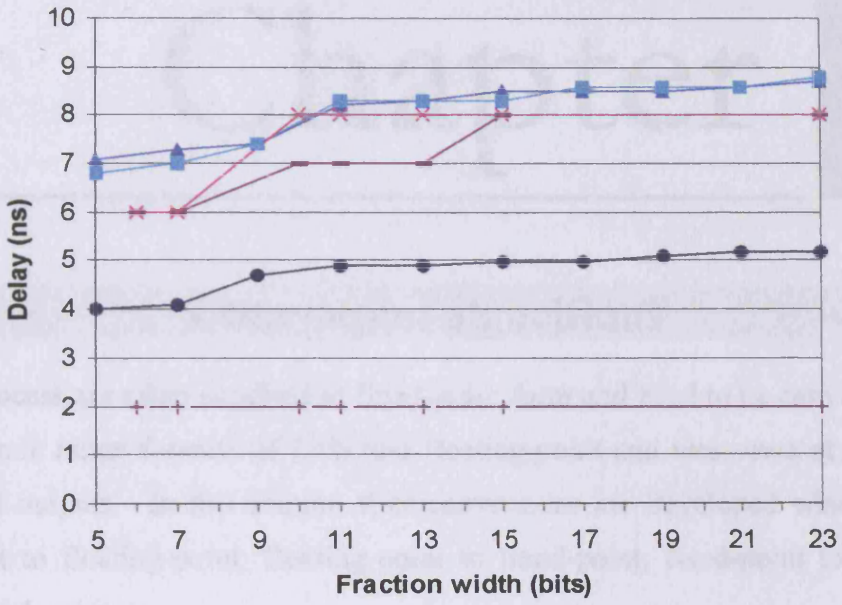


Figure 5.30. The variation of the delay, in nanoseconds, of the proposed LNS division [■], LNS multiplication [▲] and LNS square root [●] operators with fraction width. The variation of the delay of the LNS division [-], LNS multiplication [x] and LNS square root [+] operators of Detrey with fraction width.

Chapter 6

Conversion algorithms

Data to process are often supplied in fixed-point form and need to be converted to the high dynamic range formats of LNS and floating-point and vice-versa at the system inputs and outputs. In this section four conversions are developed which include: fixed-point to floating-point, floating-point to fixed-point, fixed-point to LNS, and LNS to fixed-point.

6.1 Fixed-point to floating-point conversion and vice-versa

The following conversion algorithms assume that the fixed-point data is in two's complement form with i-bit integer and f-bit fraction sections. The floating-point data is the same three-field system used in section 3.4.1 that has a single sign bit, an e-bit exponent and an m-bit mantissa. All conversion components are designed to be parameterisable in terms of the integer, fraction, exponent and mantissa widths.

6.1.1 Fixed-point to floating-point conversion

Fixed-point to floating-point conversion consists of a variable length shift to normalize the incoming fixed-point value to the range [1, 2) and a subsequent exponent creation based on the shift quantity. Fixed-point to floating-point conversion can be described by the following 7 steps:

1. Complement the fixed-point input if it is negative to ensure a positive magnitude. Detect for an input of zero using a wide NOR gate as this is a special case in floating-point. The input sign bit determines the output sign.
2. Left shift only the positive input value so it is normalized with a '1' in the MSB position.

3. Subtract the left shift amount from a constant value of 'i-1', where i is the length of the input integer section. This generates an unbiased exponent value. The subtraction is needed because the input value is only left shifted.
4. Round the shifted input to the output mantissa length. Correct the unbiased exponent value if necessary.
5. Compare the unbiased exponent value with the maximum and minimum exponent values that are allowed in the target floating-point system. These values are given as $2^{(e-1)}$-1 and -($2^{(e-1)}$-2) respectively, where e is the length of the output exponent. These comparisons determine whether underflow or overflow should be signalled.
6. Add a bias to the unbiased exponent value.
7. Check the overflow and underflow signals and also whether the input was zero and set the output registers accordingly.

6.1.1.1 Error

The only source of error in the conversion is in the rounding of the normalized input, where the maximum relative error is less than $2^{-(m+1)}$, or if underflow or overflow occur. If m, the output mantissa length, is larger than or equal to (i+f-2) then rounding error cannot occur and the converted value is exact. In such cases no rounding component is needed in the design.

6.1.1.2 Word lengths that determine the overflow possibility

Overflow cannot occur if $i \leq 2^{(e-1)}$ and $f \leq m$ (note. i, e, f and m are the lengths of the integer, exponent, fraction, mantissa respectively). In such circumstances no component is needed in the conversion design to check for overflow. If $f > m$ then overflow cannot occur if $i < 2^{(e-1)}$.

6.1.1.3 Word lengths that determine the underflow possibility

Underflow cannot occur if $-f \geq -(2^{(e-1)}-2)$ and again in such circumstances no underflow detection logic is required.

A diagram of the complete fixed-point to floating-point conversion is shown in figure 6.1.

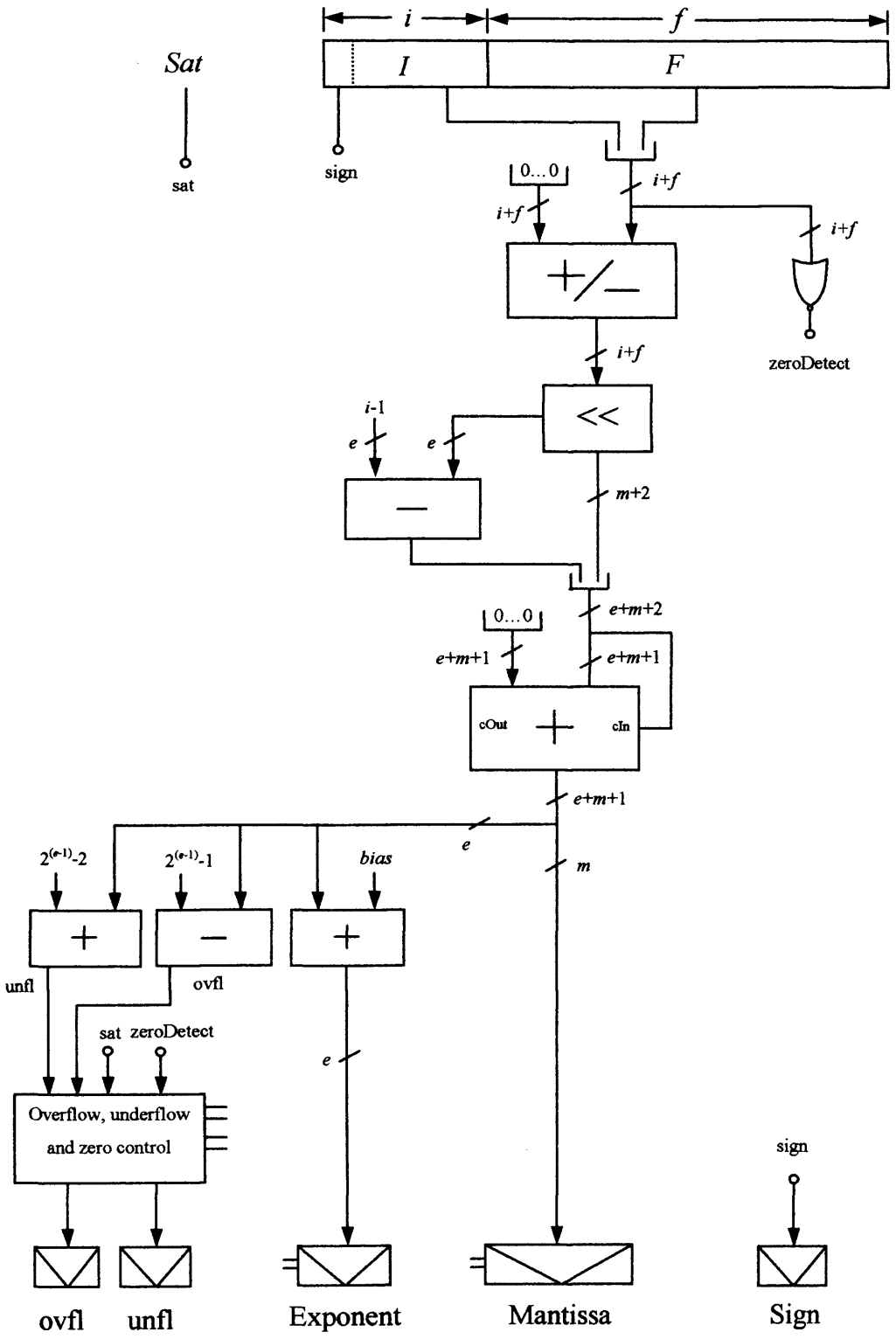


Figure 6.1. Complete fixed-point to floating-point conversion component

6.1.2 Floating-point to fixed-point conversion

Floating-point to fixed-point conversion consists of a variable length shift to adjust the mantissa according to the exponent value to produce a fixed-point value. The floating-point to fixed-point conversion is described in the following 7 steps:

1. Subtract the bias from the input exponent to produce the true exponent value. Detect for the input being a special floating-point value.
2. Compare the true exponent value with the maximum and minimum allowed values to determine if underflow or overflow has occurred.

Underflow

If the exponent value is less than $-(f+1)$ then the output will be less than the smallest fixed-point value after rounding and underflow should be signalled.

Overflow

Due to the non-symmetrical nature of the twos complement system overflow detection is more complicated then underflow detection. It is assumed that the fixed-point bit pattern of "10...0.00...00" that causes the non-symmetry is a negative value (the maximum negative value). If the exponent value is greater than $(i-2)$ then overflow should be signalled, except if the exponent value is $(i-1)$ the mantissa is zero and the sign is negative then overflow should not be signalled (this is the maximum negative value situation).

3. Subtract the exponent with the bias removed from $(i-1)$ to determine the right shift quantity. This is done so that the shifting is only done in one direction. The value to shift is initially assumed to be shifted $(i-1)$ places to the left.
4. Perform the right shift. The result of the right shit is $i+f+1$ bits in length.
5. Round the shifted value to $i+f$ bits. If the exponent is equal to $(i-2)$, $(m+1) > (i+f-1)$ and the input sign is positive then overflow can occur due to rounding and a check is needed. However if the exponent is equal to $(i-2)$, $(m+1) > (i+f-1)$ and the input sign is negative then overflow cannot occur due to rounding.
6. Negate the result according to the input sign.
7. Perform the handling of special values and the detection and handling of underflow/overflow.

6.1.2.1 Error

The only source of error in the conversion is in the rounding and if underflow or overflow occur. If $(m+1) \leq (i+f-1)$ then rounding is not required in the component design and the output will be exact.

6.1.2.2 Word lengths that determine the overflow possibility

Overflow cannot occur if $2^{(e-1)}-1 < (i-2)$ and in such circumstances no component is needed in the conversion design to check for overflow.

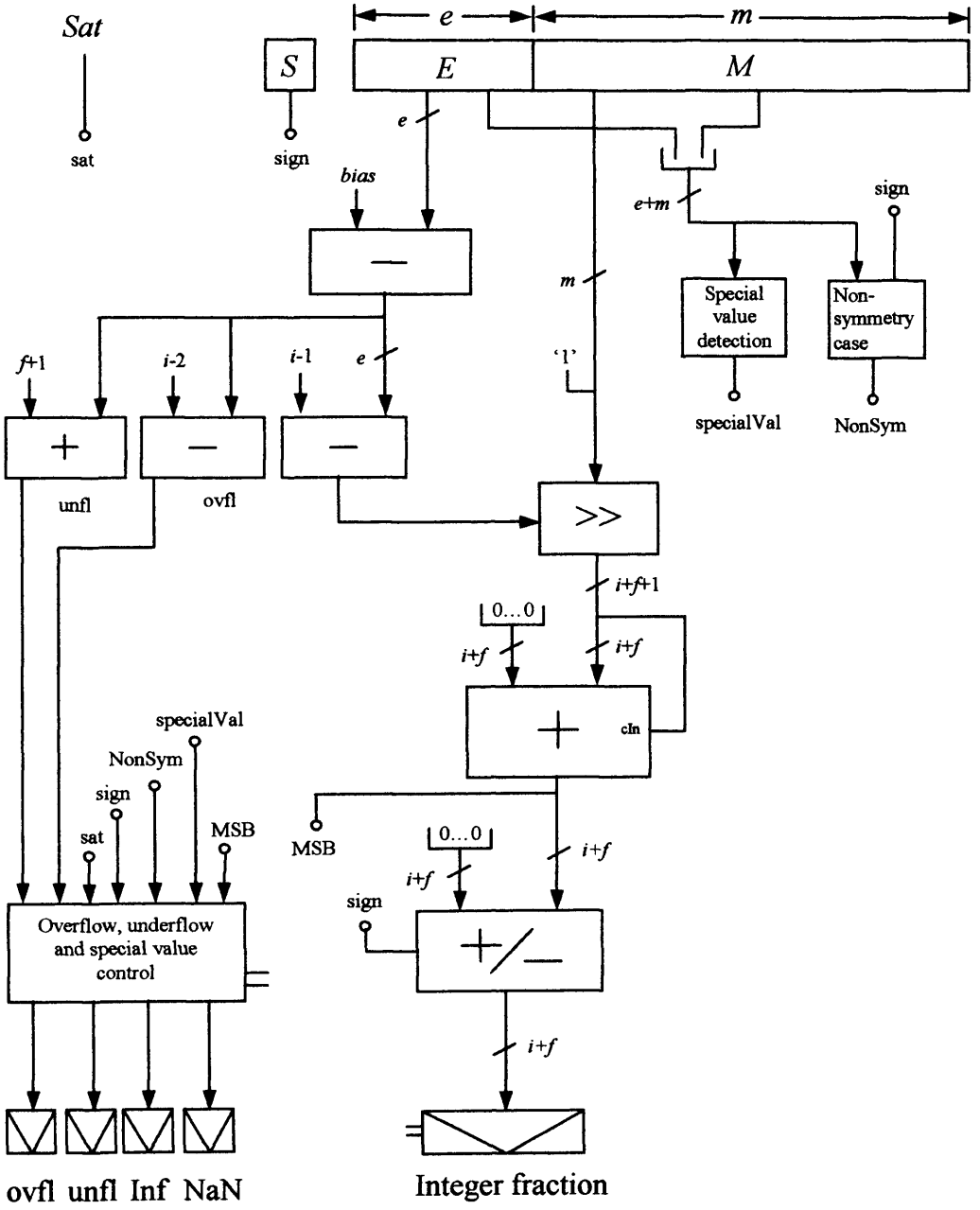


Figure 6.2. Complete floating-point to fixed-point conversion component

6.1.2.3 Word lengths that determine the underflow possibility

Underflow cannot occur if $-(2^{(e-1)}-2) \geq -(f+1)$ and in such circumstances no underflow detection logic is required.

The diagram of the complete floating-point to fixed-point conversion is shown in figure 6.2.

6.1.3 Implementation results

The results of area and delay are generated for each conversion component, which are all mapped to a Xilinx XC2V1000-4 Virtex-II FPGA.

6.1.3.1 Fixed-point to floating-point conversion

Table 6.1 shows the implementation results for different word length fixed-point to floating-point conversion functions. All of the implementations map a fixed-point word to a higher dynamic range floating-point value.

Fixed-point (i, f)	Floating-point (e, m)	Area (slices)	Area (ffs/LUTs)	Delay (ns)
4, 4	4, 3	28	8/49	14.3
6, 6	6, 5	45	12/79	16.9
8, 8	6, 9	60	16/112	18.2
10, 10	8, 11	84	20/156	20.2
12, 12	8, 23	106	32/199	15.6

Table 6.1. Implementation results for various fixed-point to floating-point conversion components

6.1.3.2 Discussion

The left shifter is the dominant component in the larger word length designs and contributes the most to the area of the component. The split of the input fixed-point word into different length fraction and integer parts has very little affect on the area of the component (other implementation results not shown here have shown this). The most important thing is the total length of the fixed-point word. As the fixed-point values are mapped to a higher dynamic range format the underflow and overflow detection adders are not used in the chosen word length implementations. The delay reduction of the final entry in the table is caused because the input does not need to be rounded and therefore the rounding component is not implemented.

6.1.3.3 Floating-point to fixed-point conversion

Table 6.2 shows the implementation results for different word length floating-point to fixed-point conversion functions. The word lengths have been chosen to complement the ones selected in table 6.1.

6.1.3.4 Discussion

Similarly to the fixed-point to floating-point converters the area of large word length components is dominated by the shifter. Interestingly the components occupy almost the same area as the ones they complement shown in table 6.1. All the components include a rounding adder and overflow and underflow detecting adders.

Floating-point (e, m)	Fixed-point (i, f)	Area (slices)	Area (ffs/LUTs)	Delay (ns)
4, 3	4, 4	41	12/65	12.7
6, 5	6, 6	51	16/87	12.4
6, 9	8, 8	67	20/118	12.6
8, 11	10, 10	83	24/149	13.9
8, 23	12, 12	104	28/190	13.9

Table 6.2. Implementation results for various floating-point to fixed-point conversion components

6.2 Fixed-point to LNS conversion and vice-versa

The following conversion algorithms assume that the fixed-point data is in two's complement form with i_{fix}-bit integer and f_{fix}-bit fractional sections. The LNS data uses the same two field format described in section 5.1 that has a single sign bit and an i_{LNS}-bit integer and f_{LNS}-bit fractional two's complement magnitude. All conversion components are designed to be parameterisable in terms of the integer and fraction widths.

6.2.1 Fixed-point to LNS conversion

The fixed-point to LNS conversion consists of a normalizing shifter, which normalizes the input fixed-point value to the range [1, 2) so the base-2 logarithm can be taken. The shift quantity acts as the output integer and the base-2 logarithm value acts as the output fraction. The full fixed-point to LNS conversion is described in the following 6 steps:

1. Complement the fixed-point input if it is negative to ensure a positive magnitude. Detect for an input of zero using a wide NOR gate as zero is a special value in the LNS. The input sign determines the output sign.
2. Left shift the positive magnitude until the most significant bit is '1'. The result of the shift will be i_{fix}-bits in length and is assumed to be in the range [1, 2).
3. Subtract the shift quantity from i_{fix}-1 to determine the value to add to the base-2 logarithm. This subtraction is required because the shift is only done in one direction.
4. Calculate the base-2 logarithm of the normalised value, which is an $i_{fix}+f_{fix}+2$ bit fractional length approximation. The result is in the range [0, 1) and is $f_{LNS}+g$ bits wide, where g is the number of guard bits.
5. Concatenate the shift value and the base-2 logarithm value and round to f_{LNS} fraction bits.
6. Check the rounded result for overflow and underflow by comparing the integer section with maximum and minimum values allowed. If the integer value is less than $-(2^{(iLNS-1)}-2)$ then underflow has occurred and should be signalled. If the integer is greater than $(2^{(iLNS-1)}-1)$ then overflow has occurred and should be signalled.
Set the output depending on overflow and underflow detection and an input of zero.

6.2.1.1 Error

Error is generated by the base-2 logarithm approximation and its subsequent rounding. This error is always present in the conversion component regardless of the input and output word lengths (as opposed to the fixed-point to floating-point conversion which with *certain* word lengths is exact). However, the BTFP criterion is used so the maximum relative error is less than the maximum relative error of the equivalent fixed-point to floating-point conversion if word lengths *are* chosen that cause errors to be generated. Error is also generated in overflow and underflow situations.

6.2.1.2 Word lengths that determine the overflow possibility

Overflow cannot occur if $(i_{fix}-1) \leq (2(i_{LNS}-1)-1)$ in such circumstances a check for overflow is not required.

6.2.1.3 Word lengths that determine the underflow possibility

Underflow cannot occur if $f_{fix} \leq (2(i_{LNS}-1)-2)$ and in such circumstances a check for underflow is not required.

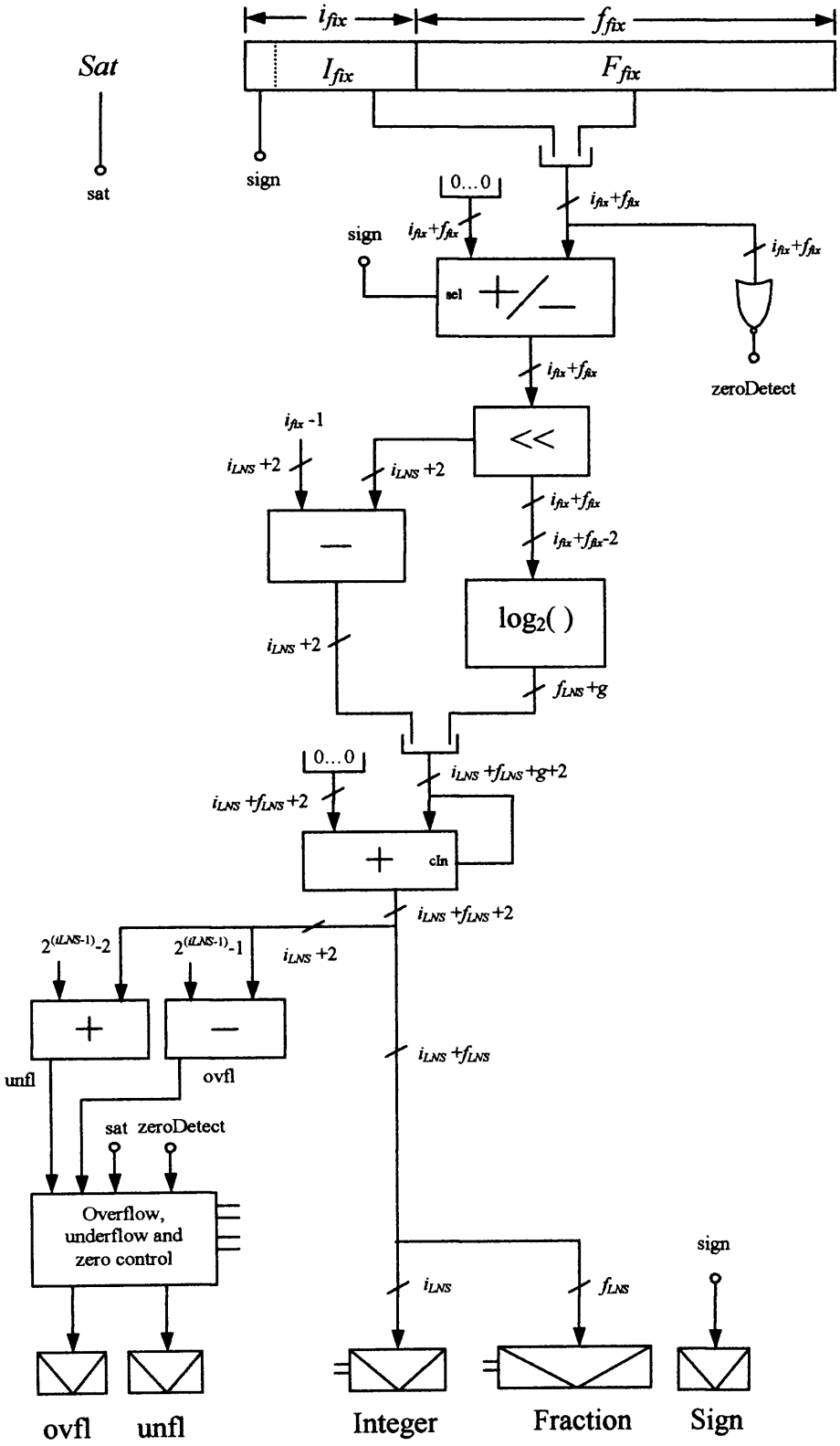


Figure 6.3. Complete fixed-point to LNS conversion diagram

6.2.2 LNS to fixed-point conversion

Conversion from LNS to fixed-point involves setting the input LNS value to be the power of the base of the LNS and solving. To restrict the domain of the 2^x approximation (for base-2 LNS) a variable length shifter is used to shift the output of the approximation for LNS values that are not in the range [0, 1) i.e. the identity of (6.1) is used.

$$2^{i+f} = 2^i * 2^f \quad \text{(6.1)}$$

i is an integer value so 2^i is a simple shift operation. The full LNS to fixed-point conversion is described by the following 8 steps:

1. Split the input LNS magnitude into integer and fractional parts. Calculate the 2^f approximation of the fraction section. The input domain of the function approximation is [0, 1) and the output range is [1, 2). The input width to the function approximation is f_{LNS} and the output is $i_{fix}+f_{fix}+g$.
2. Check the LNS input for special values.
3. Compare the LNS integer part with the maximum and minimum allowed values.

Underflow

If the LNS integer value is less than $-(f_{fix}+1)$ then underflow should be signalled.

Overflow

Due to the non-symmetrical nature of the twos complement system overflow detection is more complicated than underflow. This is the same situation that occurred in the floating-point to fixed-point conversion component. If the LNS integer value is greater than $(i_{fix}-2)$ then overflow should be signalled, except if the exponent value $(i_{fix}-1)$ the LNS fraction value is zero and the LNS sign is negative.

4. Subtract the LNS integer value from $(i_{fix}-1)$ to create the value that controls the shifter. The subtraction is done because the shifter only shifts in one direction.
5. Right shift the 2^f function approximation result. The output of the shifter is $i_{fix}+f_{fix}+1$ bits in length.
6. Round the shifted value to $i_{fix}+f_{fix}$ bits. Overflow can occur after rounding and a check is needed.
7. Negate the rounded result according to the input LNS sign.
8. Perform the handling of special values and the detection and handling of overflow and underflow.

6.2.2.1 Error

The sources of error in the conversion are in the function approximation and its subsequent rounding. The error of the function approximation is chosen so that the relative error is less than that of the equivalent floating-point system (BTFP accuracy). Underflow and overflow are also sources of error.

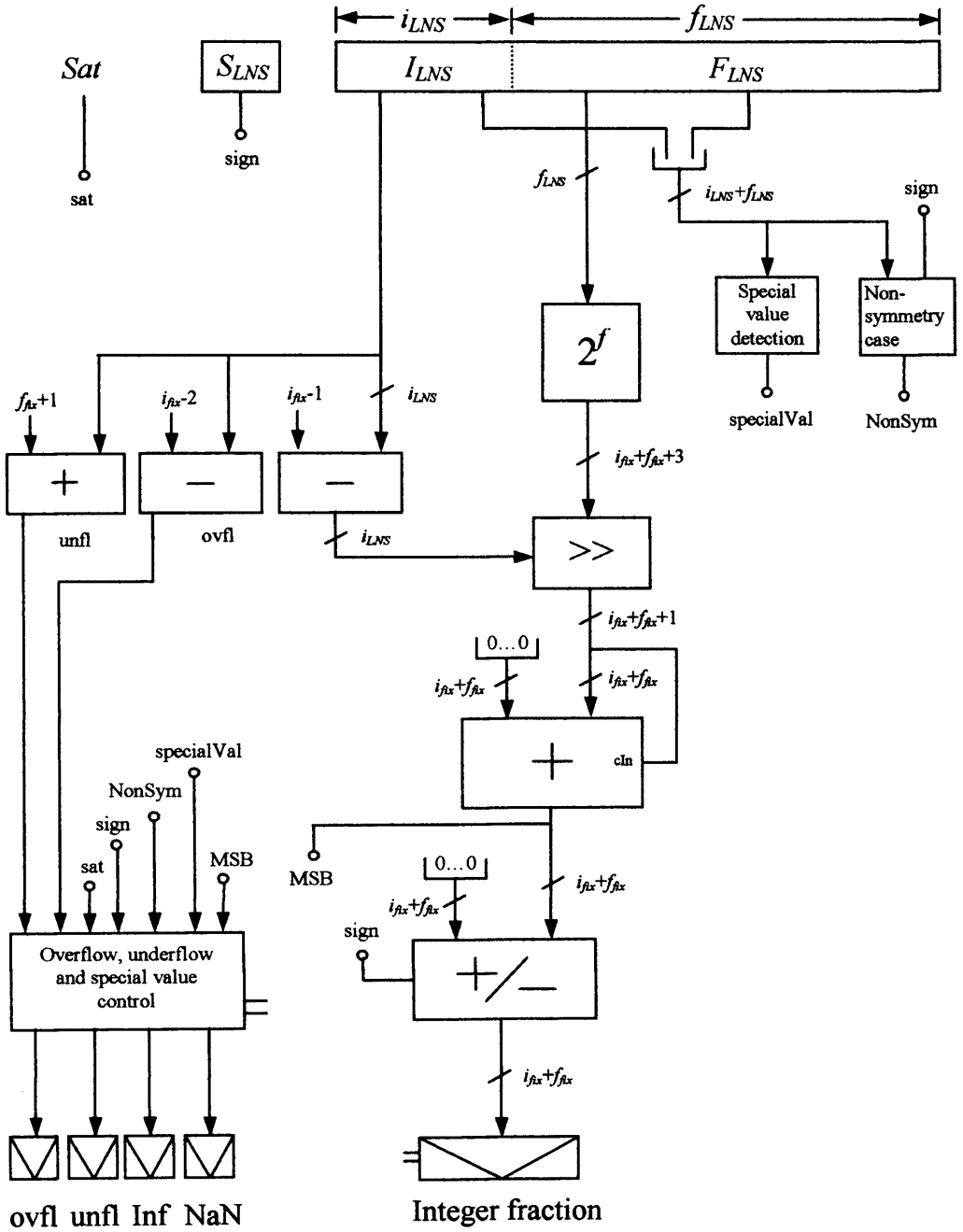


Figure 6.4. Complete LNS to fixed-point conversion diagram

6.2.2.2 Word lengths that determine the overflow possibility

Overflow cannot occur if $2(i_{LNS}-1)-1 < (i_{fix}-2)$ and in such circumstances no component is needed in the conversion design to check for overflow.

6.2.2.3 Word lengths that determine the underflow possibility

Underflow cannot occur if $-(2(i_{LNS}-1)-2) \geq -(f_{fix}+1)$ and in such circumstances no underflow detection logic is required.

The diagram of the complete LNS to fixed-point conversion component is shown in figure 6.4. The component has special outputs to indicate whether the input LNS value was infinity or NaN and if overflow or underflow were detected.

6.2.3 Implementation results

The results of area and delay are generated for each conversion component, which are all mapped to a Xilinx XC2V1000-4 Virtex-II FPGA. The embedded multipliers are not used in the function approximation components as the results are to be compared with the results from the fixed-point to floating-point and floating-point to fixed-point conversion section.

6.2.3.1 Fixed-point to LNS conversion

Table 6.3 shows the implementation results for different word length fixed-point to floating-point conversion functions. All of the implementations map a fixed-point word to a higher dynamic range LNS value.

Fixed-point (i, f)	LNS (i, f)	Area (slices)	Area (ffs/LUTs)	Delay (ns)
4, 4	4, 3	31	8/56	11.3
6, 6	6, 5	77	12/139	25.8
8, 8	6, 9	199	16/342	29.1
10, 10	8, 11	330	20/617	49.3
12, 12	8, 23	842	32/1573	52

Table 6.3. Implementation results for various fixed-point to LNS conversion components

6.2.3.2 Discussion

The function approximation is the dominant area part of the larger word length implementations. The area of the conversion components increases exponentially with word length due to the function approximation. The function approximation also contributes significantly to the delay of the implementations.

6.2.3.3 LNS to fixed-point conversion

Table 6.4 shows the implementation results for different word length LNS to fixed-point conversion functions. The word lengths have been chosen to complement the ones selected in table 6.3.

6.2.3.4 Discussion

Similarly to the fixed-point to LNS converters the area of larger components is dominated by the function approximation. The components occupy almost the same area as the ones they complement shown in table 6.3.

LNS (i, f)	Fixed-point (i, f)	Area (slices)	Area (ffs/LUTs)	Delay (ns)
4, 3	4, 4	42	12/70	11
6, 5	6, 6	67	16/118	11.4
6, 9	8, 8	195	20/371	34.8
8, 11	10, 10	334	24/637	37.5
8, 23	12, 12	861	28/1643	47.7

Table 6.4. Implementation results for various LNS to fixed-point conversion components

Chapter 7

MATLAB libraries

All the components designed have equivalent bit-true MATLAB models to enable their functionality to be tested and to enable the convenient development of algorithms in the friendly MATLAB environment before hardware design is undertaken. MATLAB also provides support for serial communication, which allows the functionality of components to be tested after they have been placed on the FPGA using a simple serial communication protocol. However the speed of designs cannot be tested as the speed of the serial link is slow, for such data we have to rely solely on the vendor place and route tool estimates.

7.1 Floating-point library

The MATLAB environment does not support arbitrary precision floating-point formats so a library of functions was written that act as bit-true models to test the VHDL floating-point components against. The MATLAB models have been written to provide exact functionality including support for special values (NaN, Infinity, etc). The interfaces of the four operations of addition, multiplication, division and square root are shown below:

7.1.1 Addition

```
[sX,eX,mX] = flpAdd(sA,sB,eA,eB,mA,mB,op,rndMode,e,m)

% sX is the returned sign
% eX is the returned exponent
% mX is the returned mantissa

% sA is the input sign of operand A
% sB is the input sign of operand B
% eA is the input exponent of operand A
% eB is the input exponent of operand B
% mA is the input mantissa of operand A
% mB is the input mantissa of operand B
% op is whether an addition or subtraction operation is required
```

```
% rndMode specifies which one of the four IEEE rounding modes to use
% e is the length of the input exponent in bits
% m is the length of the input mantissa in bits
```

7.1.2 Muliplication

```
[sX,eX,mX] = flpMult(sA,sB,eA,eB,mA,mB,rndMode,e,m)

% see above (floating-point addition) for argument description
```

7.1.3 Division

```
[sX,eX,mX] = flpDiv(sA,sB,eA,eB,mA,mB,rndMode,e,m)

% see above (floating-point addition) for argument description
```

7.1.4 Square root

```
[sX,eX,mX] = flpSqrt(sA,eA,mA,rndMode,e,m)

% see above (floating-point addition) for argument description
```

The fixed-point division and square root routines used in the floating-point operations are based on simple restoring implementations, which produce correct quotients and remainders. These are also used to test the fixed-point SRT (all radices) and non-restoring based division and square root algorithms for correctness.

7.2 LNS library

Similarly to the floating-point library a bit-true parameterisable LNS library has also been developed. The interfaces to the library components are shown below.

7.2.1 Addition

```
[sX,sum] = lnsAddSub(op,sA,sB,A,B,i,f)

% sX is the output sign.
% sum is the output magnitude.  It is a signed value.

% op is whether an addition or subtraction operation is required
% sA is the sign of input A
% sB is the sign of input B
% A is the magnitude of operand A, which is a signed value
% B is the magnitude of operand B, which is a signed value
% i is the length of the input integer section
% f is the length of the input fraction section
```

7.2.2 Multiplication

```
[sX,prod] = lnsMult(sA,sB,A,B,i,f)

% prod is the output product, it is a signed value
```

7.2.3 Division

```
[sX,quot] = lnsDiv(sA,sB,A,B,i,f)

% quot is the output quotient, it is a signed value
```

7.2.4 Square root

```
[sX,root] = LNSsqrt(sA,A,i,f)

% root is the output square root, it is a signed value
```

The LNS addition/subtraction function is written as a separate function so can be exchanged in the main LNS adder routine as required. Parameterisable bit-true models of the dual-path and parallel-lookup addition/subtraction functions have been written, which have been exhaustively tested for accuracy.

7.3 Conversion library

Fixed-point data needs to be converted to and from the floating-point and LNS formats. A library of parameterisable components to allow such conversions has been written and the interfaces of these functions are described below.

7.3.1 Fixed-point to floating-point

```
[sFlp,eFlp,mFlp,ovfl,unfl] = fix2float(sFix,Fix,i,f,e,m,sat)

% sFlp is the output sign of the floating-point value
% eFlp is the output exponent of the floating-point value
% mFlp is the output mantissa of the floating-point value
% ovfl is an overflow flag
% unfl is an underflow flag

% sFix is the input fixed-point sign
% Fix is the input fixed-point magnitude, an unsigned value
% i is the length of the fixed-point integer
% f is the length of the fixed-point fraction
% e is the length of the floating-point exponent
% m is the length of the floating-point mantissa
% sat controls whether saturation values are used for overflow
```

7.3.2 Floating-point to fixed-point

```
[sFix,Fix,ovfl,unfl,inf,NaN] = float2fix(sFlp,eFlp,mFlp,i,f,e,m,sat)

% inf is an infinity flag
% NaN is a Not-a-Number flag
% (see above for other argument descriptions)
```

7.3.3 Fixed-point to LNS

```
[sLNS,LNS,ovfl,unfl] = fix2LNS(iFix,fFix,fix,iLNS,fLNS,sat)

% sLNS  is the sign of the output LNS value
% LNS is the magnitude of the output LNS value, it is a signed value
% ovfl is an overflow flag
% unfl is an underflow flag

% iFix is the length of the fixed-point integer
% fFix is the length of the fixed-point fraction
% fix is the input fixed-point value, it is a signed value
% iLNS is the length of the LNS integer
% fLNS is the length of the LNS fraction
% sat controls whether saturation values are used for overflow
```

7.3.4 LNS to fixed-point

```
[sFix,Fix,ovfl,unfl,inf,NaN] = LNS2fix(iLNS,fLNS,sLNS,LNS,iFix,fFix,sat)

% sFix is the output fixed-point sign
% Fix is the output fixed-point magnitude, an unsigned value
% inf is an infinity flag
% NaN is a Not-a-Number flag
% (see fix2LNS for other argument descriptions)
```

7.4 Serial communication system

The FPGA development board used to test the functionality of the developed modules contains an RS232 serial communication port. The transmit and receive connections of the serial port are connected via an external UART (Universal Asynchronous Receive Transmit) chip directly to two pins of the FPGA, which allows direct communication with the FPGA. Only three wires are needed in the communication link: a wire to transmit data from the host PC to the FPGA, a wire so the FPGA can transmit data back to the host PC and a common ground line.

7.4.1 Communication procedure

The RS232 serial port operates in an asynchronous mode. Each frame of data sent consists of 8 data bits, a start bit and a stop bit as shown in figure 7.1.

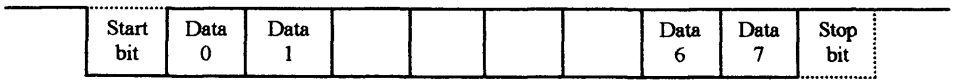


Figure 7.1. The serial data frame

When no data is being sent the communication line is held high. The receiver thus waits for a single start bit, which is a low bit and it then knows that the next 8-bits are the data bits. Once the 8-bits have been sent the transmitter sends a stop bit, which is a high bit and then holds the line high or starts transmitting a new frame with a new start bit. Once the receiver receives a stop bit it waits for the next start bit to read in the next 8-bits of data and so on.

7.4.2 FPGA communication system

MATLAB is used to send and receive data from the FPGA via the RS232 serial port. A diagram of the communication system is shown in figure 7.2.

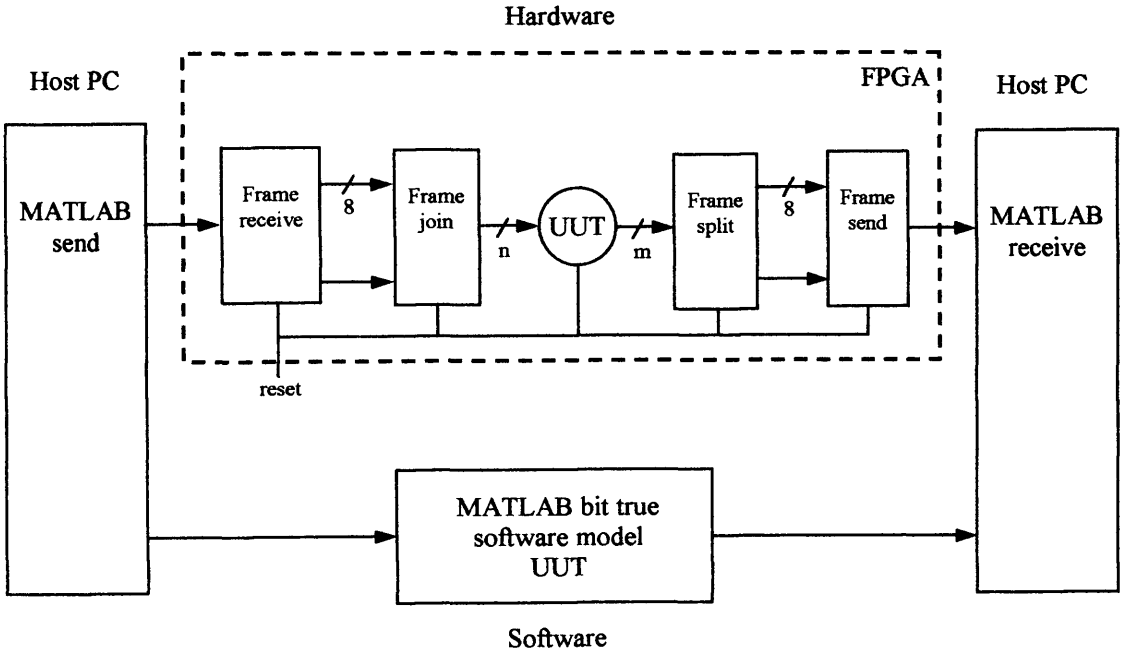


Figure 7.2. MATLAB to FPGA communication system

The diagram in figure 7.2 consists of the following components:

1. **MATLAB send**. This is the MATLAB software running on the host PC. The data to send is split into 8-bit packets and is then sent serially to the FPGA.

2. **FPGA**. The FPGA contains five modules, which are all tied to a common reset line to ensure the registers and control logic start in a known state and to reset the system. The five modules are described as follows:

a. Frame receive. The frame receiver waits for a start bit then clocks in 8-bits of data. The single input data line is fed into 8 registers and these registers are enabled in turn to latch in the 8-bits of data. The enable line of each register is controlled by a counter, which divides down the clock and allows the register to latch in the data at the correct time. The frame receiver signals when a new frame of 8-bits is available. Figure 7.3 shows the basic structure of the frame receiver.

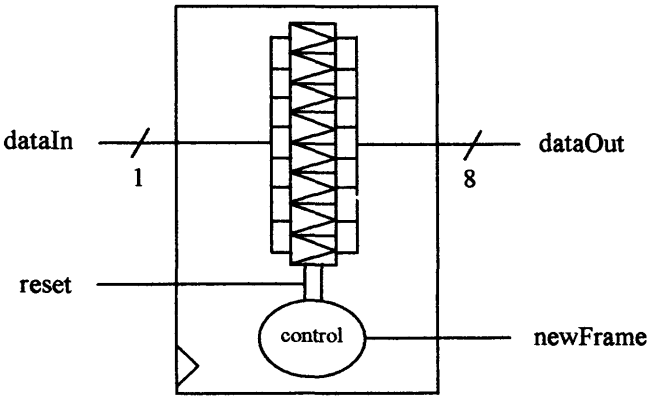


Figure 7.3. Frame receiver structure

b. Frame join. The frame join component takes 8-bit frames and joins them into a larger frame. It operates in a very similar way to the frame receiver and signals when a new x*8-bit frame is ready. Figure 7.4 shows an example of a frame joiner that joins four 8-bit frames to produce a 32-bit word.

c. UUT. UUT stands for unit under test and is the component that the input data is being fed through to generate the output data to compare against the software model.

d. Frame split. The frame split component splits up large words of data into smaller 8-bit frames, which can be sent over the communication link. The frame split component signals each time a new frame is ready. Figure 7.5 shows the structure of a frame split component that splits 16-bit data into 8-bit frames.

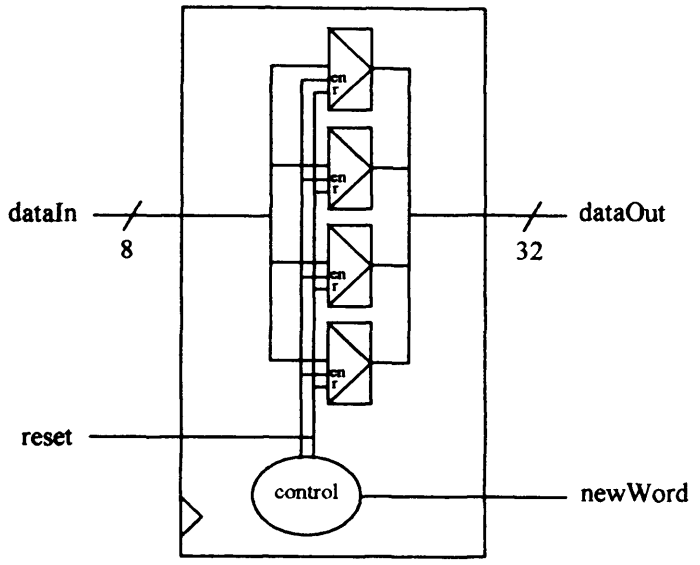


Figure 7.4. Frame join structure

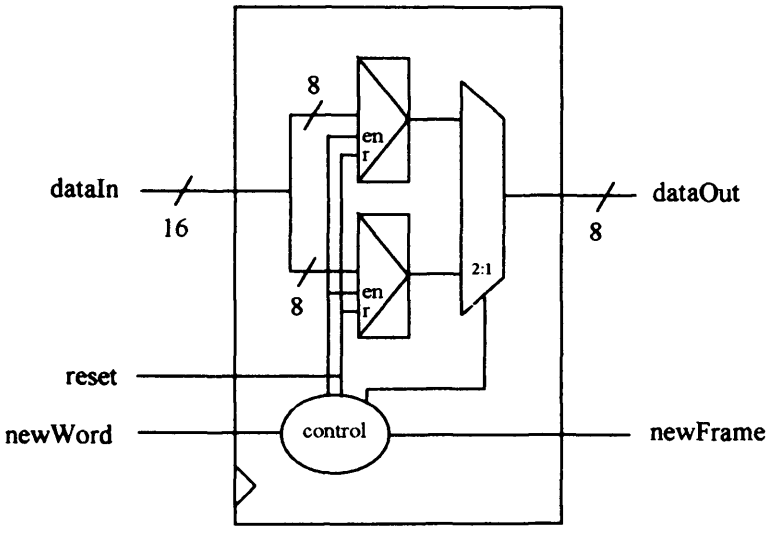


Figure 7.5. Frame split structure

e. Frame send. The frame send component holds the line high and waits for a frame to send. When a new 8-bit frame arrives it sends it along with an initial start bit and a final stop bit. After sending a frame the frame transmitter holds the line high. Figure 7.6 shows the structure of a frame send component.

3. **MATLAB receive**. This is the MATLAB software running on the host PC. Data packets are received and are stored in a buffer until they are read. The data packets are read and are used to reconstruct the output from the hardware UUT. The output of

the hardware UUT is compared with the bit-true software model of the UUT to ensure the two match exactly.

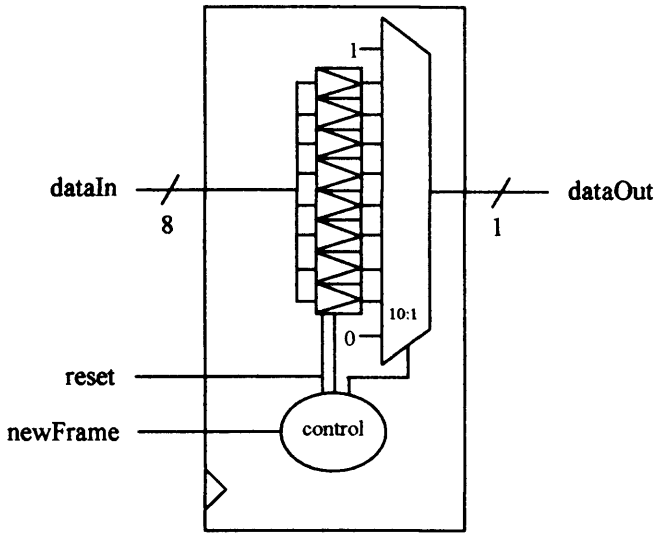


Figure 7.6. Frame send structure

Chapter 8

Comparison

In this chapter the LNS and floating-point formats are compared in terms of speed and area. The four basic operators of addition, multiplication, division and square root are compared as are the conversion components. In order to provide a range of results different equations are compared as well as the straight operator comparison. Only non-pipelined results are considered. As shown in previous implementation results sections the fractional part of an LNS number or the mantissa length of a floating-point value have the greatest effect on the size of components. Therefore only different fraction/mantissa lengths will be considered in the results and not different integer/exponent lengths.

8.1 A comparison of the four basic operators

Table 8.1 compares the area and delay of the two basic operators of addition and multiplication and table 8.2 compares the division and square root operators. LNS addition results are given for the parallel-lookup (Par) and dual-path (Dual) methods.

(i, f) / (e, m)	Addition								Multiplication				
	Area			Delay			Mults 18x18	Mults 18x18	Area		Delay		Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	LNS	Flp	LNS	Flp	Flp
4, 5	87	122	99	27.6	41.7	22.5	1	1	16	33	7.1	16.5	1
4, 7	171	214	127	28	43.6	26.1	1	1	17	34	7.3	16.8	1
6, 9	271	208	153	39.4	57.6	27.5	2	2	23	41	7.4	18.4	1
8, 11	348	326	177	49.5	62	27	3	2	26	45	8.2	18.8	1
8, 13	504	564	194	51.6	67.9	28.2	3	2	27	46	8.3	19.1	1
8, 15	644	492	225	59.4	102.5	31.5	4	4	28	48	8.5	20.7	1
8, 17	921	623	242	64	103.2	32.1	4	4	31	69	8.5	23.3	1
8, 19	1367	877	259	70.1	107.9	32.7	4	4	32	111	8.5	25.2	1
8, 21	1985	1213	279	75.3	118.8	32.9	4	4	33	157	8.6	26	1
8, 23	3062	1608	296	76	125.5	33.5	4	4	34	207	8.7	26.6	1

Table 8.1. Area and delay of selected LNS and floating-point operators

(i, f) / (e, m)	Division				Square root			
	Area		Delay		Area		Delay	
	LNS	Flp	LNS	Flp	LNS	Flp	LNS	Flp
4, 5	18	75	6.8	31.2	11	43	4	21.1
4, 7	19	98	7	38.3	12	56	4.1	26.6
6, 9	24	132	7.4	47.3	16	76	4.7	33.4
8, 11	27	164	8.3	60	18	95	4.9	39.9
8, 13	28	200	8.3	67.6	19	118	4.9	47.7
8, 15	29	239	8.3	74.9	20	137	5	58.9
8, 17	32	284	8.6	84.6	22	162	5	67.4
8, 19	33	332	8.6	93.4	23	189	5.1	77.5
8, 21	34	384	8.6	103.9	24	218	5.2	88.3
8, 23	35	438	8.8	114.4	25	248	5.2	95.6

Table 8.2. Area and delay of selected LNS and floating-point division and square root operators

To show the trends in the results more clearly the results are plotted in figures 8.1-8.8. Figures 8.1-8.8 show the area and delay of the addition, multiplication, division and square root operators respectively.

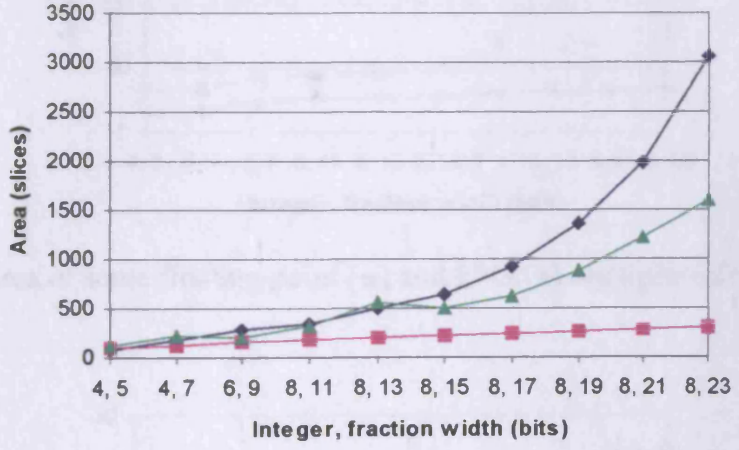


Figure 8.1. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] adder implementations

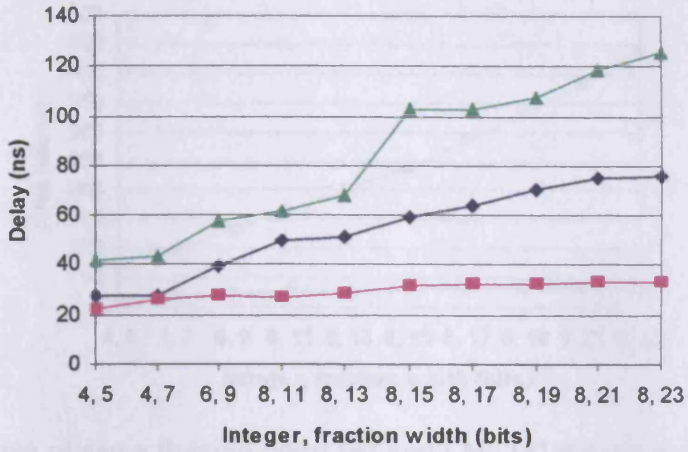


Figure 8.2. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] implementations

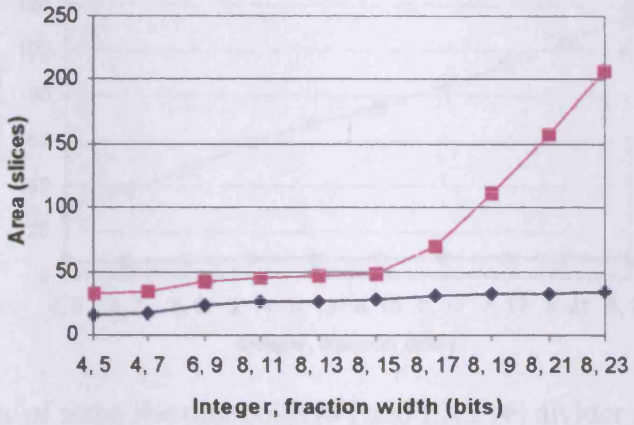


Figure 8.3. Area of some floating-point [■] and LNS [♦] multiplier implementations

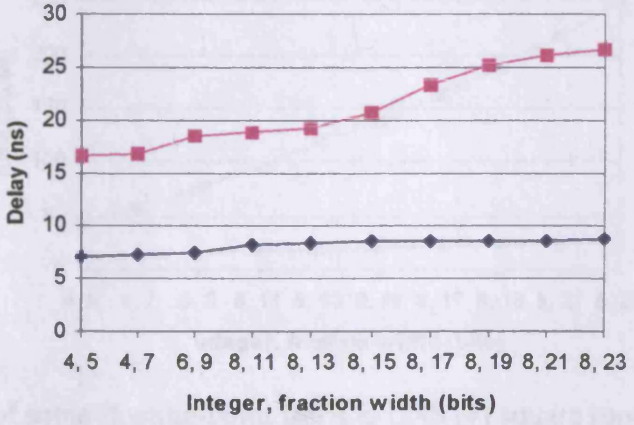


Figure 8.4. Delay of some floating-point [■] and LNS [♦] multiplier implementations

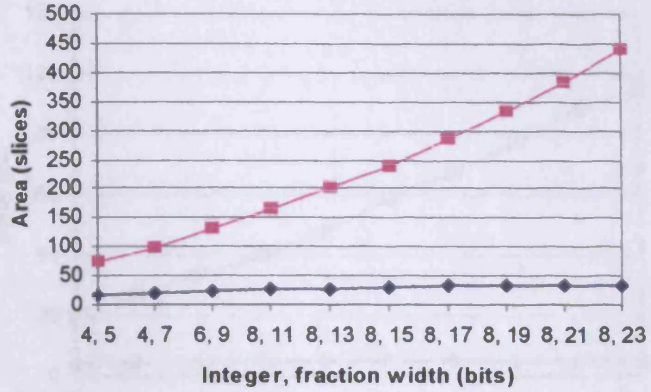


Figure 8.5. Area of some floating-point [■] and LNS [♦] divider implementations

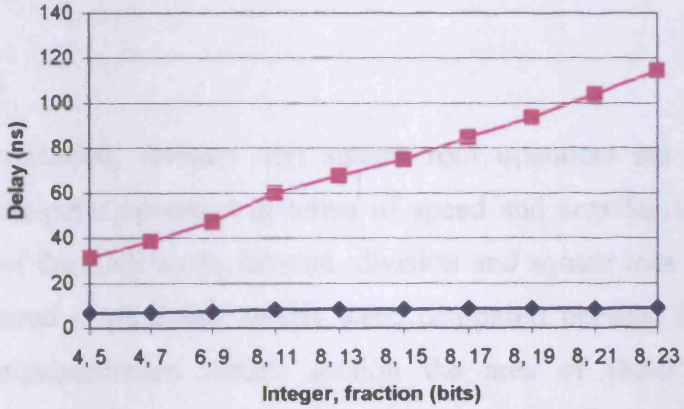


Figure 8.6. Delay of some floating-point [■] and LNS [♦] divider implementations

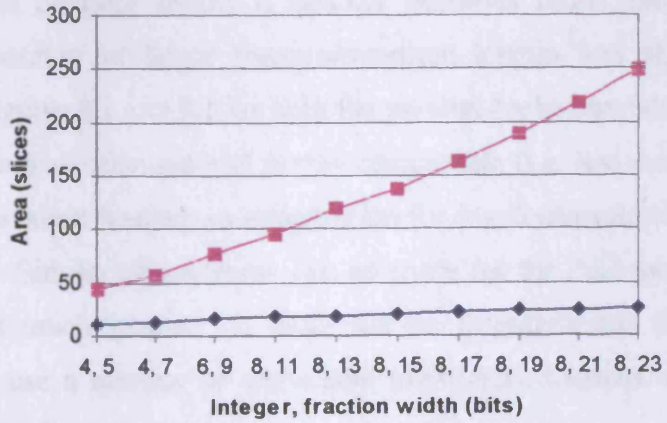


Figure 8.7. Area of some floating-point [■] and LNS [♦] square root implementations

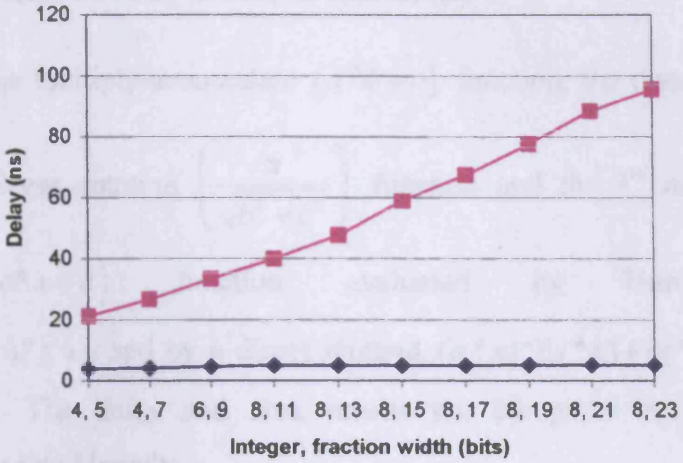


Figure 8.8. Delay of some floating-point [■] and LNS [♦] square root implementations

8.1.1 Discussion

The LNS multiplication, division and square root operators are superior to the equivalent floating-point operators in terms of speed and area for all word lengths. The superiority of the LNS multiplication, division and square root operators would be further enhanced if pipelined results were compared because as shown in the floating-point implementation results section the area of these three operators increases significantly when the components are pipelined. The LNS parallel-lookup implementation of the smallest (5-bit) fraction/mantissa adder is faster and smaller than the equivalent floating-point operator. However the exponential size increase of the LNS addition operator means it quickly becomes larger than the equivalent floating-point operator for larger fraction/mantissa lengths and also slower. This trend is seen in figures 8.1 and 8.2 for both the parallel-lookup and dual-path methods. The parallel-lookup addition method is still comparable (i.e. less than twice the area) in area to the equivalent floating-point operation for fraction/mantissa word lengths up to about 11-bits. Similar observations can be made for the dual-path method except that the delay is much greater. It must not be forgotten that the LNS addition implementations use a number of embedded multipliers whereas the floating-point addition operations do not use any.

8.2 A comparison of four selected functions

In this section the multiply-accumulate $(a*b+c)$ function, the distance $\left(\sqrt{a^2+b^2}\right)$ function, the Givens rotation $\left(\frac{a}{\sqrt{b^2+c^2}}\right)$ function and the 3rd order polynomial $\left(a*x^3+b*x^2+c*x+d\right)$ function evaluated by Horner's method $((a*x+b)*x+c)*x+d$ and by a direct method $(a*x)*(x*x)+(x*x)*b+c*x+d$ are compared. The delay and area results are compared for a selection of fraction/mantissa word lengths.

8.2.1 Multiply-accumulate

Table 8.3 illustrates the results of the multiply-accumulate function implemented with floating-point (Flp) and logarithmic (LNS) arithmetic. Both the parallel-lookup (Par) and dual-path (Dual) LNS addition implementations are considered. To illustrate the result trends more clearly the area and delay results of table 8.3 are plotted in figures 8.9 and 8.10 respectively.

(i, f) / (e, m)	Multiply-accumulate								
	Area			Delay			Mults 18x18	Mults 18x18	Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp
4, 5	103	138	132	34.7	48.8	39	1	1	1
4, 7	188	231	161	35.3	50.9	42.9	1	1	1
6, 9	294	231	194	46.8	65	45.9	2	2	1
8, 11	374	352	222	57.7	70.2	45.8	3	2	1
8, 13	531	591	240	59.9	76.2	47.3	3	2	1
8, 15	672	520	272	67.9	111	52.2	4	4	1
8, 17	952	654	311	72.5	111.7	55.4	4	4	1
8, 19	1399	909	370	78.6	116.4	57.9	4	4	1
8, 21	2018	1246	436	83.9	127.4	58.9	4	4	1
8, 23	3096	1642	503	84.7	134.2	60.1	4	4	1

Table 8.3. LNS and floating-point multiply-accumulate implementation results

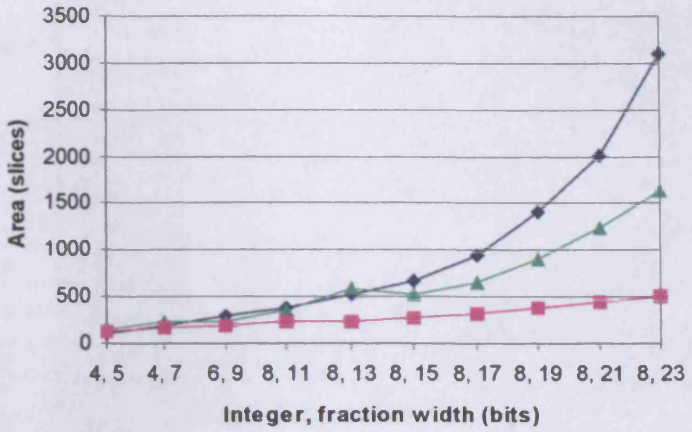


Figure 8.9. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] multiply-accumulate (MAC) implementations

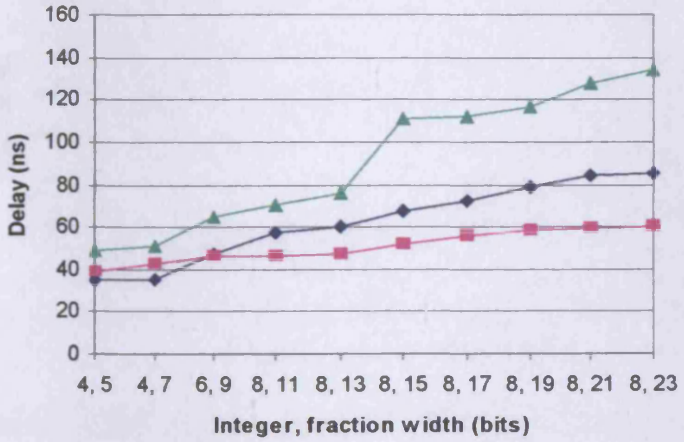


Figure 8.10. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] multiply-accumulate (MAC) implementations

8.2.2 Distance

Table 8.4 illustrates the results of the distance function implemented with floating-point (Flp) and logarithmic (LNS) arithmetic. Both the parallel-lookup (Par) and dual-path (Dual) LNS addition implementations are considered. To illustrate the result trends more clearly the area and delay results of table 8.4 are plotted in figures 8.11 and 8.12 respectively.

(i, f) / (e, m)	Distance								
	Area			Delay			Mults 18x18	Mults 18x18	Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp
4, 5	130	165	208	38.7	52.8	60.1	1	1	2
4, 7	217	260	251	39.4	55	69.5	1	1	2
6, 9	333	270	311	51.5	69.7	79.3	2	2	2
8, 11	418	396	362	62.6	75.1	85.7	3	2	2
8, 13	577	637	404	64.8	81.1	95	3	2	2
8, 15	720	568	458	72.9	116	111.1	4	4	2
8, 17	1005	707	542	77.5	116.7	122.8	4	4	2
8, 19	1454	964	670	83.7	121.5	135.4	4	4	2
8, 21	2075	1303	811	89.1	132.6	147.2	4	4	2
8, 23	3155	1701	958	89.9	139.4	155.7	4	4	2

Table 8.4. LNS and floating-point distance implementation results

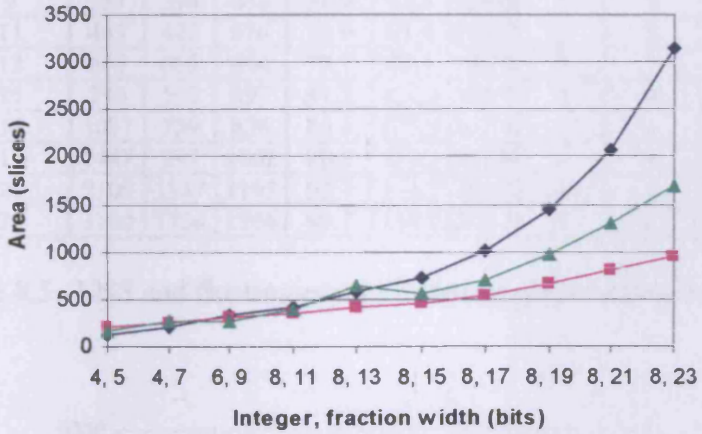


Figure 8.11. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] distance implementations

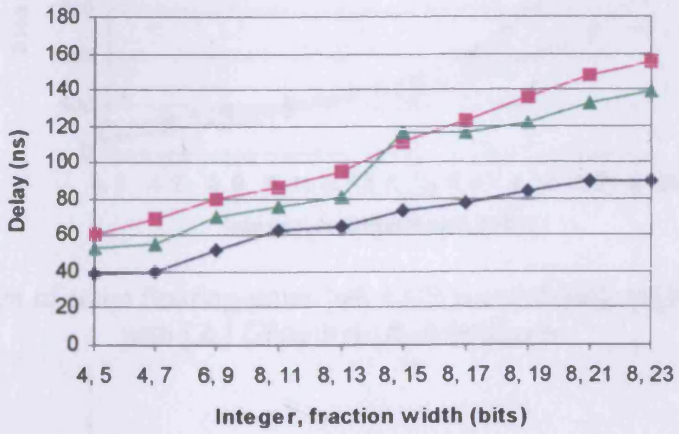


Figure 8.12. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] distance implementations

8.2.3 Givens

Table 8.5 illustrates the results of the Givens function implemented with floating-point (Flp) and logarithmic (LNS) arithmetic. Both the parallel-lookup (Par) and dual-path (Dual) LNS addition implementations are considered. To illustrate the result trends more clearly the area and delay results of table 8.5 are plotted in figures 8.13 and 8.14 respectively.

(i, f) / (e, m)	Givens								
	Area			Delay			Mults 18x18	Mults 18x18	Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp
4, 5	148	183	283	45.5	59.6	91.3	1	1	2
4, 7	236	279	349	46.4	62	107.8	1	1	2
6, 9	357	294	443	58.9	77.1	126.6	2	2	2
8, 11	445	423	526	70.9	83.4	145.7	3	2	2
8, 13	605	665	604	73.1	89.4	162.6	3	2	2
8, 15	749	597	697	81.2	124.3	185.9	4	4	2
8, 17	1037	739	826	86.1	125.3	207.4	4	4	2
8, 19	1487	997	1002	90.3	130.1	228.8	4	4	2
8, 21	2109	1337	1195	97.7	141.2	251.1	4	4	2
8, 23	3190	1736	1396	98.7	148.2	270.1	4	4	2

Table 8.5. LNS and floating-point Givens implementation results

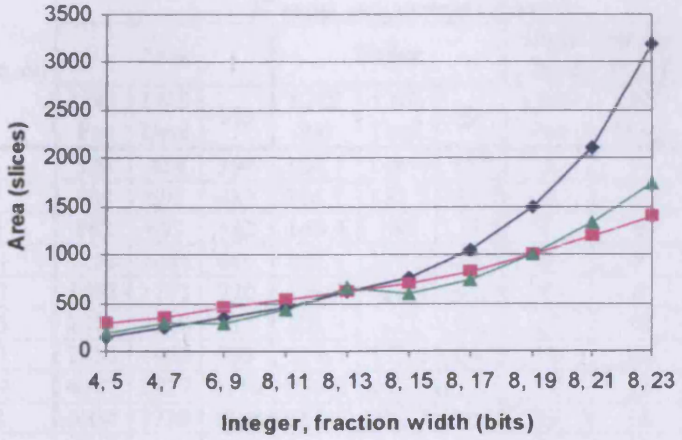


Figure 8.13. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] Givens implementations

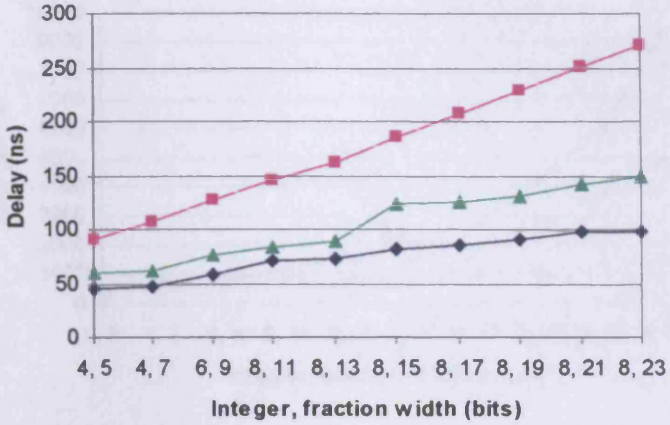


Figure 8.14. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] Givens implementations

8.2.4 3rd order polynomial (Horner)

Table 8.6 illustrates the results of a 3rd order polynomial function implemented with floating-point (Flp) and logarithmic (LNS) arithmetic. Both the parallel-lookup (Par) and dual-path (Dual) LNS addition implementations are considered. To illustrate the result trends more clearly the area and delay results of table 8.6 are plotted in figures 8.15 and 8.16 respectively.

(i, f) / (e, m)	3rd order polynomial (Horner)								
	Area			Delay			Mults 18x18	Mults 18x18	Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp
4, 5	309	414	396	104.1	146.4	117	3	3	3
4, 7	564	693	483	105.9	152.7	128.7	3	3	3
6, 9	882	693	582	140.4	195	137.7	6	6	3
8, 11	1122	1056	666	173.1	210.6	137.4	9	6	3
8, 13	1593	1773	720	179.7	228.6	141.9	9	6	3
8, 15	2016	1560	816	203.7	333	156.6	12	12	3
8, 17	2856	1962	933	217.5	335.1	166.2	12	12	3
8, 19	4197	2727	1110	235.8	349.2	173.7	12	12	3
8, 21	6054	3738	1308	251.7	382.2	176.7	12	12	3
8, 23	9288	4926	1509	254.1	402.6	180.3	12	12	3

Table 8.6. LNS and floating-point 3rd order polynomial implementation results

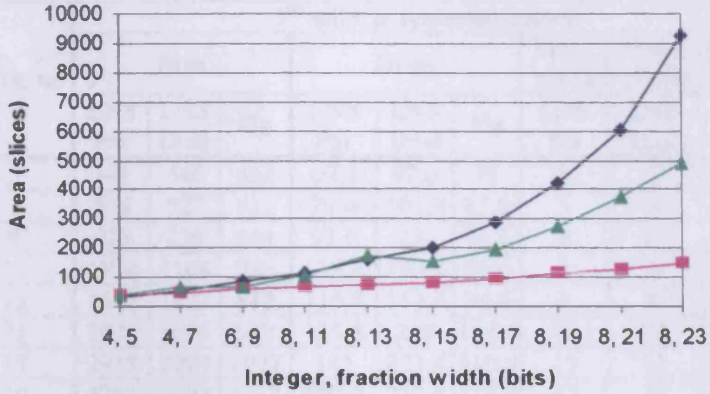


Figure 8.15. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] 3rd order polynomial implementations

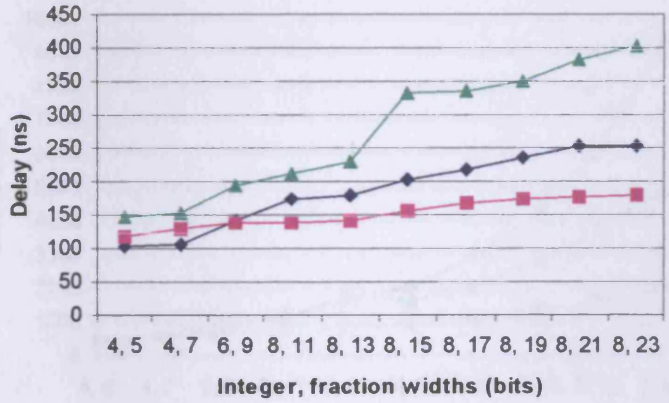


Figure 8.16. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] 3rd order polynomial implementations

8.2.5 3rd order polynomial (direct)

Table 8.7 illustrates the results of a 3rd order polynomial function implemented with floating-point (Flp) and logarithmic (LNS) arithmetic. Both the parallel-lookup (Par) and dual-path (Dual) LNS addition implementations are considered. To illustrate the result trends more clearly the area and delay results of table 8.7 are plotted in figures 8.17 and 8.18 respectively.

(i, f) / (e, m)	3rd order polynomial (direct)								
	Area			Delay			Mults 18x18	Mults 18x18	Mults 18x18
	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp	LNS Par	LNS Dual	Flp
4, 5	341	446	462	69.4	97.6	78	3	3	5
4, 7	598	727	551	70.6	101.8	85.8	3	3	5
6, 9	928	739	664	93.6	130	91.8	6	6	5
8, 11	1174	1108	756	115.4	140.4	91.6	9	6	5
8, 13	1647	1827	812	119.8	152.4	94.6	9	6	5
8, 15	2072	1616	912	135.8	222	104.4	12	12	5
8, 17	2918	2024	1071	145	223.4	110.8	12	12	5
8, 19	4261	2791	1332	157.2	232.8	115.8	12	12	5
8, 21	6120	3804	1622	167.8	254.8	117.8	12	12	5
8, 23	9356	4994	1923	169.4	268.4	120.2	12	12	5

Table 8.7. LNS and floating-point 3rd order polynomial implementation results

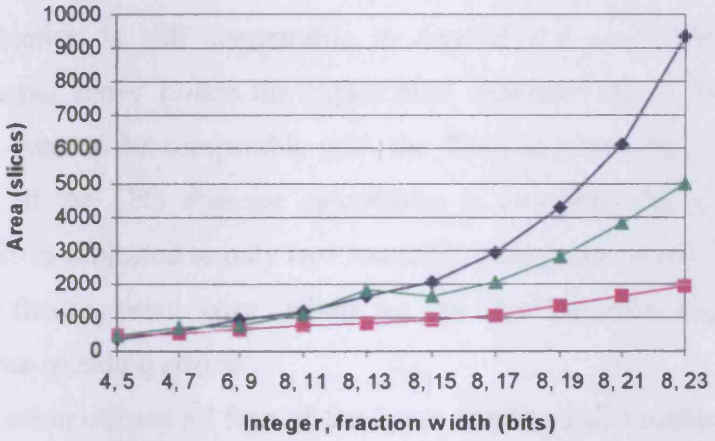


Figure 8.17. Area of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] 3rd order polynomial implementations

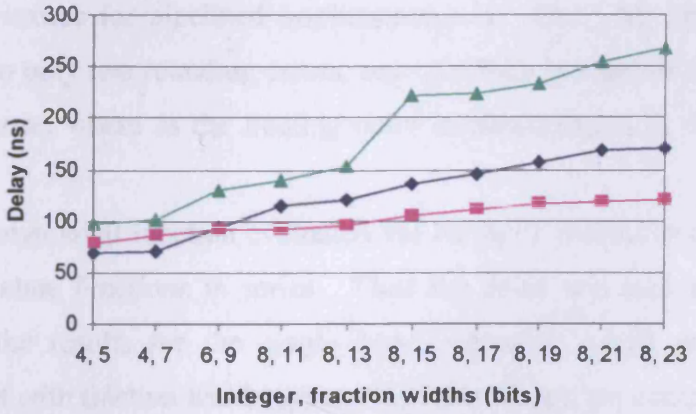


Figure 8.18. Delay of some floating-point [■], LNS parallel-lookup [♦] and LNS dual-path [▲] 3rd order polynomial implementations

8.2.6 Discussion

The multiply-accumulate (MAC) function is crucial to many algorithms. The LNS implementation for the shortest length fraction/mantissa is the only LNS implementation that is superior in terms of both speed and area to the equivalent floating-point implementation. The LNS implementation with the 2nd shortest word length fraction/mantissa is superior in terms of delay only. Up to 11 fraction/mantissa bits the area of the LNS implementation is comparable to the floating-point version but above 11-bits the exponential area increase of the LNS addition function means the floating-point implementation is far superior.

The distance function is more suited to LNS implementation, which is evident from the superior delay of the LNS for all word lengths. The LNS is superior in area for fraction/mantissa lengths of up to 9-bits. Above 9-bits and up to about 17-bits the LNS implementation is still comparable to that of the equivalent floating-point implementation but above 17-bits the exponential area increase of the LNS addition function is too great to be comparable with the floating-point implementation. The rounding error of the LNS distance calculation is less than that of floating-point because the LNS is subjected to only two rounding errors, one of which is superior to the equivalent floating-point error, where as the floating-point implementation is subjected to three rounding errors.

The Givens function utilizes all four of the basic operators and makes the function a very good candidate for LNS implementation as the results show. The delay of the LNS implementation is far superior to floating-point for all word lengths. The area of the LNS implementation is also superior up to a fraction/mantissa width of 19-bits, which would increase for pipelined implementations. The LNS implementation is also subjected to only two rounding errors, one of which is superior to the equivalent floating-point error, where as the floating-point implementation is subjected to four rounding errors.

The 3rd order polynomial function evaluated via Horner's method is essentially three multiply-accumulate functions in series. Thus the delay and area are simply a 3x increase over the results for the single MAC operator, which means that LNS implementations with fraction widths greater than 11-bits are not comparable.

The direct implementation method utilises two extra multiplications compared to Horner's method and is thus more suited to LNS implementation. Furthermore the

initial multiplier tree as shown in figure 8.19 can be implemented without any rounding errors.

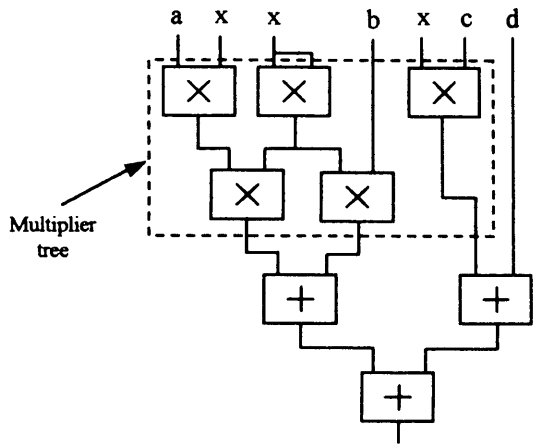


Figure 8.19. Direct implementation of a 3rd order polynomial

Despite having a 5:3 multiplication to addition ratio the LNS implementation of the direct polynomial function is only superior for fraction/mantissa widths of 5 and 7-bits and is comparable up to 11-bits. This is due to the efficiency of implementing floating-point multiplication and the exponential area increase of the LNS addition operator.

8.3 Conversion component comparison

8.3.1 Fixed-point to LNS/floating-point

A direct comparison of the fixed-point to LNS/floating-point (Flp) conversion algorithms is shown in table 8.8.

Fixed-point (i,f)	Flp/LNS (e,m)/(i,f)	LNS area (slices)	Flp area (slices)	LNS delay (ns)	Flp delay (ns)
4,4	4,3	31	28	11.3	14.3
6,6	6,5	77	45	25.8	16.9
8,8	6,9	199	60	29.1	18.2
10,10	8,11	330	84	49.3	20.2
12,12	8,23	842	106	52	15.6

Table 8.8. Fixed-point to LNS/floating-point conversion component comparison

The area and delay of the LNS conversions are clearly greater than those of the equivalent floating-point operations. Looking at the diagrams of the conversion components in sections 6.1 and 6.2 the delay and area increase are caused by the

logarithm function approximation. The area of the approximation increases exponentially with word length and this greatly impacts on the size of the conversion component for all but the smallest word length conversions where the area of the LNS and floating-point conversions are equivalent. The logarithm approximation also greatly impacts on the delay of the conversion, however for the shortest word lengths the delay of the LNS conversion is less than that of the equivalent floating-point conversion. This is caused by a combination of the 'bias' adder and the P&R tools.

8.3.2 LNS/floating-point to fixed-point

A direct comparison of the LNS/floating-point to fixed-point conversions algorithms is shown in table 8.9.

Flp/LNS (e,m)/(i,f)	Fixed-point (i,f)	LNS area (slices)	Flp area (slices)	LNS delay (ns)	Flp delay (ns)
4,3	4,4	42	41	11	12.7
6,5	6,6	67	51	11.4	12.4
6,9	8,8	195	67	34.8	12.6
8,11	10,10	334	83	37.5	13.9
8,23	12,12	861	104	47.7	13.9

Table 8.9. LNS/floating-point to fixed-point conversion component comparision

The delay and area increase of the LNS conversion is caused by the 2^x function approximation. The area of the 2^x function approximation increases exponentially with word length and this greatly impacts on the size of the conversion component for all but the smallest word length designs. The delay of the two shortest word length conversions for floating-point is greater than the LNS conversion due to a combination of the bias subtractor and P&R tools.

Chapter 9

Conclusion

This work has investigated the efficient implementation of fixed-point operators on the Virtex-II FPGA and then applied these operators to a function approximation design implementation. A variation of the function approximation design using a more accurate approximation technique was then used in LNS addition/subtraction components to facilitate a comparison of logarithmic and floating-point arithmetic. A low level design strategy has been adopted which makes use of the fine grain primitives of the FPGA technology to achieve maximum control over the logic used and to try and extract the maximum functionality out of the available resources. In this chapter the main conclusions of the different chapters are given, the question of the suitability of LNS and floating-point arithmetic for different FPGA algorithms is answered and future studies are suggested.

9.1 Fixed-point

The fixed-point section demonstrated that traditional carry-free addition techniques have 'carry-chain' pipelining issues that prevent them from offering any benefit to the basic RCA design for short word length implementation. A new 4:2 CSA mapping was proposed which is suitable for long word length carry-free applications such as cryptography algorithms, but this mapping still suffers from the pipelining issue. Many ripple-carry adder configurations, some not previously published in the open literature, were described as methods to reduce the logic requirement and delay of certain functions.

The classic Booth encoding technique was investigated for implementation on the Virtex-II FPGA as it had not been previously considered in the open literature. The technique was dismissed due to the (already present) dedicated radix-4 partial product generator logic, which is superior to the Booth-2 partial product generator

implementation. The Booth-3 algorithm was considered next and was dismissed due to the double-subtraction problem, which prevents the simultaneous negation and addition of two operands. Higher-radix algorithms were not considered due to the difficulty of the multiple generation and the overhead of the recoding logic. A new way to expand the embedded multipliers was described and its superiority in area and delay to other methods was shown. The technique was also applied to squaring and a substantial area saving was observed. Different radices of digit recurrence division up to radix-16 were discussed. The radices above 16 were dismissed due to the excessively large quotient digit selection ROMs that are required. The algorithms were compared in terms of the LUT columns used, the critical path and the size of the quotient digit selection ROM. The results showed that the basic maximally redundant radix-4 algorithm was the most suitable as the quotient digit selection function only required a 5-bit ROM. The digit recurrence analysis highlighted that the minimally redundant radix-8 algorithm offered certain efficiency benefits but the size of the quotient digit selection function was too large to make the implementation practical. Therefore, the technique of pre-scaling was investigated for use with the minimally redundant radix-8 algorithm. The results showed that the pre-scaled minimally redundant algorithm offered a significant speed up over the maximally redundant radix-4 algorithm, but the extra area of the initial pre-scaling multiplications was too great to justify the delay reduction. Pipelined results were not given but the minimally redundant radix-8 algorithm is expected to be more competitive in terms of area to the radix-4 algorithm if both of the algorithms are pipelined.

The implementation of SRT digit recurrence square root was discussed and no benefit of the algorithm over the basic non-restoring algorithm was evident. This is due to the fact that each iteration of the square root algorithm depends on adding/subtracting multiples of the partial result. The digits of the partial result are selected from a redundant digit set for the SRT algorithm and so a full-length operation is required to update the partial result each iteration to convert it to a non-redundant form. The full-length addition or subtraction to update the partial result substantially increases the area of the square root component compared to the basic non-restoring algorithm and so the SRT algorithm was dismissed.

9.2 Floating-point

The first published dual-path floating-point adder implementation for FPGA was proposed as a result of this study into floating-point, as was the first group of double-precision floating-point modules. The results showed the proposed addition algorithm to be the fastest in the open literature, which is due to the new and efficient 4-IEEE rounding mode component mapping, the new rounding and exponent correction structure, a new lead-zero detection algorithm and a new shifter mapping. The results showed that all the algorithms smaller than the one proposed were based on the 'vanilla' algorithm, however, the smaller algorithms also had a reduced functionality set which further reduced their size. The 'vanilla' algorithm is expected to be smaller since extra logic such as adders and normalising components that are required to facilitate the dual-path structure are not required for the 'vanilla' algorithm. Other dual-path adders have been proposed since the original published work but they do not improve on the results presented. The algorithm scales well to larger operand widths as shown by its 9x speed improvement and comparative size to a recently published double-precision design.

Few other floating-point multiplication designs use the embedded multipliers so a fair comparison was difficult. However when compared to the designs that do use embedded multipliers the proposed design was more efficient as it uses the multiplier developed in the fixed-point section. This structure gives the proposed design a delay and area advantage over other designs. The only designs smaller than the proposed design either use more embedded multipliers in their multiplier structure or are based on a digit-serial structure, which causes a significant delay penalty. The proposed design is faster than all other designs apart from designs mapped to a faster speed grade Virtex-II FPGA.

The smallest floating-point division designs use a bit-serial divider for the significand division. However, these designs are also the slowest and cannot be pipelined. The proposed design is not the smallest or fastest for short word length mantissas as a commercial design has slightly better results, which is due to the use of RLOC placement directives. The proposed design is the smallest and fastest for larger word length mantissas and has the highest level of functionality of all other non-fully IEEE std-754 compliant designs.

The proposed square root design uses a non-restoring square root extractor and is smaller than other designs that use restoring designs and radix-2 SRT designs as predicted in the fixed-point square root section. The proposed design offers the greatest level of functionality for the smallest area and has the fastest speed. The smallest square root design in the literature uses a bit-serial design but it is slow and cannot be pipelined.

9.3 Function evaluation

The main function evaluation methods were discussed and an overview of the current methods implemented on FPGA was given. None of the methods in the open literature met the requirements of the study and so a new function approximation mapping was investigated. A new piecewise Taylor series approximation technique was proposed and the trade off of arithmetic and ROM logic that is achieved by changing the order of approximation was discussed. The final design, as the results showed, proved to be the most efficient way of approximating smooth functions of restricted range and domain on FPGA and furthermore could take advantage of the embedded multiplier primitives.

9.4 Logarithmic number system

The aim of the LNS chapter was to provide a set of modules with the same functionality level as the equivalent floating-point operators and therefore to allow a fair comparison between the two number systems. The accuracy, dynamic range and special value usage were shown to be equivalent or better than the floating-point system. Two methods were developed for the LNS addition/subtraction function approximation. The aim of the two methods was to provide area efficient and delay efficient implementations of the function. The parallel-lookup method offered a low delay LNS addition/subtraction implementation and the dual-path method, which was based on the dual-path floating-point method, offered a low area solution for large fraction word lengths. The function approximations used in both methods were based on the piecewise method developed in the function evaluation chapter. However, instead of using a Taylor approximation a more accurate Chebyshev approximation was used. The multiplication, division and square root designs are very simple and there are very few different design options so all the mappings of these operators are

very similar. The addition algorithm was shown to be considerably smaller than all other 8-bit integer, 23-bit fraction designs (i.e. the format comparable to IEEE std 754 single precision) and of similar speed. The proposed design is not the smallest in terms of slice only logic for short (5-6 bits) fraction lengths, however it is the smallest if the design makes use of the embedded multipliers. Furthermore the design that is smaller has a reduced accuracy subtraction function in the singularity region due to the difficulty of accurately approximating the function in this region. The design is not the fastest for short (5-13 bits) fraction lengths, but the design that is faster is larger above 7 fraction bits and has a reduced accuracy (i.e. not as good as floating-point) subtraction function.

9.5 Comparison

9.5.1 Operator comparison

The comparison of the four basic operators gave conclusive results showing the area and delay superiority of the LNS for implementing the three basic operations of multiplication/division/square root. However the exponential area increase of the LNS addition/subtraction function meant the equivalent floating-point operation was superior in area for characteristics (fraction/mantissa) greater than 5-bits. Furthermore the delay of the function approximation used in the LNS addition/subtraction operator meant it was slower than the equivalent floating-point operator for all characteristic lengths. The superiority of floating-point addition and the efficiency of implementing floating-point multiplication, due to the Virtex-II FPGA embedded multipliers, means that the MAC operation and other useful functions and algorithms solely involving additions and multiplications are more efficiently implemented with floating-point arithmetic. This is true for all characteristics above 5-bits as shown by the functions in tables 8.3, 8.6 and 8.7.

The LNS is more competitive for specialist applications that use the operations of division and square root and few additions as shown in tables 8.4 and 8.5. Here the LNS exhibited a 50-100% speed-up over the equivalent floating-point implementation and was comparable and at times superior in area for characteristic lengths of up to 15-bits. The area savings of the LNS implementations are expected to increase for these specialist applications if pipelined results are compared as the area of the floating-point multiplication/division/square root operators will increase as shown in

tables 3.18, 3.23 and 3.28, but the area of the LNS operators will remain the same. A pipelined comparison cannot be made as pipelined LNS addition designs have not been produced due to time restrictions.

Certain algorithms will be more accurate when implemented with LNS arithmetic because the LNS multiplication and division operators are exact and powering operators can be correctly rounded with a maximum error that is less than the equivalent floating-point error. Furthermore the LNS addition/subtraction operator has been designed to have an accuracy that is at least as good as the equivalent floating-point operator.

9.5.2 Conversion comparison

The conversions in to and out of the LNS/floating-point domains are a one off operation as all 'internal' operations needed for an algorithm can be carried out in the domain's native arithmetic. A direct comparison of the LNS and floating-point conversions highlights the area and delay advantage of the floating-point conversions for large fixed-point and LNS/floating-point word lengths. The accuracy of the LNS and floating-point conversions is equivalent but in some cases the floating-point conversions can be exact where as the LNS conversions always incorporate a conversion error. In certain systems with many operations the delay and area overhead of the conversions from fixed-point to LNS/floating-point and vice-versa are relatively small and so the size increase of the LNS is not significant. However the difference does exist and does enter into the argument against using LNS arithmetic.

9.6 LNS or floating-point: which is the most suitable for FPGA implementation?

Now an initial study and comparison of floating-point and logarithmic high dynamic range number systems has been completed an answer to the question posed in the introduction can be constructed. There are clearly trade offs in the two number systems so a single straightforward answer to the question cannot be given, so instead a set of guide lines to help the designer select the appropriate system is given:

All algorithms that can tolerate a small characteristic of 5-bits or less should be implemented with logarithmic arithmetic.

Algorithms that are based solely on additions, or multiplications and additions, and that require a larger characteristic than 5-bits should be implemented with floating-point arithmetic.

Algorithms that require very few additions and consist of the two operations of division and square root are candidates for LNS implementation for characteristics of up to 19-bits. However, the actual choice is algorithm dependant and the implementation results of tables 8.1 and 8.2 can be used to make a pre-implementation decision.

For 'internal' accuracy purposes algorithms with a high proportion of multiplications, divisions and square root operations compared to the number of addition operations are good candidates for implementation using LNS arithmetic. The MATLAB libraries developed can be used to test the accuracy of a system before it is implemented in hardware to ensure it meets the desired accuracy.

For 'conversion' accuracy purposes the floating-point system is the best choice as the target word length can be chosen to prevent rounding errors. However when conversion rounding does occur the accuracy of both LNS and floating-point conversions are equivalent.

Algorithms that contain many sequential multiplication, division and square root operations are fast and accurate for LNS implementation.

The only candidate for algorithms that require very accurate high dynamic range computations is floating-point arithmetic as it is feasible to implement double precision floating-point operations on FPGA.

Algorithms that involve powering operations and few additions should be implemented with LNS arithmetic as powering is a simple fixed-point multiplication operation that can easily be rounded.

9.7 Future work

There are many more studies that can be performed to further validate the findings for the comparison of the two number systems. A few of the studies are listed below:

Pipelined results have not been generated for the LNS addition operator so a natural next step would be to investigate the pipelining of the LNS addition operator to enable a comparison of the pipelined results. Furthermore an automated pipeline option could be developed for all components so the number of pipeline stages (latency) could be set as a design parameter.

The pipelining study for the LNS addition operator would go hand-in-hand with a study into using the dedicated synchronous memory blocks that reside throughout the Virtex-II FPGA. So far the blocks have only been studied for use in an 8-bit integer, 23-bit fraction design and this study, which gave promising results, could be expanded to other characteristic lengths.

Reducing the accuracy of the LNS addition/subtraction function has been shown to dramatically reduce the memory requirement and thus the area of the addition/subtraction function approximation. This would be an interesting study to see the impact on the area of the FPGA implementation, however the precision of the floating-point addition operators would need to be reduced to keep the fair comparison criteria.

The LNS implementation of the powering function can be implemented with a fixed-point multiplier and can be correctly rounded. The floating-point implementation on the other hand requires a function approximation unit, which increases exponentially in area with argument word length. Furthermore the table-makers dilemma means that a great deal of extra logic is required to correctly round the result. The LNS powering operation has much in common with the constant coefficient multiplication (KCM) operator and the development and comparison of general floating-point and LNS powering functions would be an interesting study.

An investigation into the other mentioned fixed-point square root and division implementations would be a useful study. This work has only considered one class of algorithm for both operations but others do exist, which might have area or speed advantages.

A commercial design has shown the area and delay benefits of using placement directives (RLOCs) and these could be added to the developed components to improve the layout, delay and area consumption.

To make the floating-point modules fully IEEE std-754 compliant the concept of denormalised numbers could be implemented. This will involve the use of extra shifters to correctly align the inputs or normalise the outputs as required.

The parallel-lookup LNS addition algorithm could be redesigned to include the new idea mentioned in section 5.9.2.4. This would significantly reduce the hardware needed to implement the function approximation arithmetic.

The current study has focused on using the Chebyshev function approximation scheme for the functions involved with the LNS addition/subtraction operator. This is not the most accurate polynomial approximation method and so a useful study would be to see the area and accuracy gain (if any) achieved by using the optimum minimax function approximation method.

The power consumption of devices is a major topic area due to the increase in mobile device technology. A beneficial study would be to compare the power consumption of FPGA algorithms implemented with LNS and floating-point arithmetic.

This study has focused on Xilinx Virtex/Virtex-II FPGAs and as specific primitives of the Virtex/Virtex-II FPGA have been used a straightforward port of the developed designs to other technology is not possible. Similar fixed-point studies would be needed to determine the most efficient implementations for other FPGA technology. However, the high level function evaluation, floating-point, LNS and conversion algorithms can be used to guide the design process for other FPGA technologies and in certain cases for ASIC implementations. Other FPGAs have different low-level structures and different macro blocks (i.e. different adder structures, LUT sizes, memory block size and DSP blocks) and an interesting study would be to see the impact of these alternative features on the implementation of logarithmic and floating-point arithmetic.

Appendix A

SRT division

A.1 Radix-2 SRT division with non-redundant residual

x/d is the division operation where x is the dividend and d is the divisor. The process of division calculates a quotient q and a remainder rem such that equation (A.1) is true.

$x = q^*d + rem$, where $rem<d$ ___(A.1)

We will assume $x \in [0,1)$, $d \in [0.5,1)$ and follow the constraint that $x<d$, thus $q \in [0,1)$. The general recurrence for SRT algorithms is given in (A.2).

$w[j+1] = r.w[j] - d.q_{j+1}$ ___(A.2)

The digit set for SRT algorithms is typically selected to be as in (A.3).

$q_j \in \{-a,-a+1,\ldots,-1,0,1,\ldots,a-1,a\}$ ___(A.3)

The redundant quotient digit set {-1,0,1} is chosen for the radix-2 algorithm. For this digit set a=1 and the radix r=2. The redundancy factor ρ is given by (A.4).

$$\rho = \frac{a}{r-1} = \frac{1}{1} = 1 \quad \text{___(A.4)}$$

Generally a digit set with $0.5 < \rho < 1$ is redundant. A digit set with ρ=1 is maximally redundant and with ρ>1 over redundant. When a=r/2 the digit set is minimally redundant. From (A.4) the digit set is redundant. For a particular quotient digit q we can use (A.5) and (A.6) to determine the upper and lower selection bounds as illustrated in table A.1.

$$U_k = (k + \rho).d \quad \text{(A.5)}$$

$$L_k = (k - \rho).d \quad \text{(A.6)}$$

$q_{j+1}=k$	U_k	L_k
1	2d	0
0	d	-d
-1	0	-2d

Table A.1. Selection boundaries for radix-2 SRT division quotient digit set {-1,0,1}

The bounds given in table A.1 can be plotted on a pd-diagram, which can then be used to determine the selection values. Figure A.1 shows the q_{j+1}=1 selection region in red the q_{j+1}=0 selection region in yellow and the q_{j+1}=-1 selection region in blue. The orange and green regions are where the selection regions overlap. In these regions it is possible to choose either value for the next quotient digit and the algorithm will still converge to the correct result. It is this property that ultimately allows reduced precision comparisons for the selection function. On figure A.1 horizontal lines have been drawn at 0.5 and –0.5. These are the selection constants that are used to determine the next quotient digit and are independent of the divisor (x-axis).

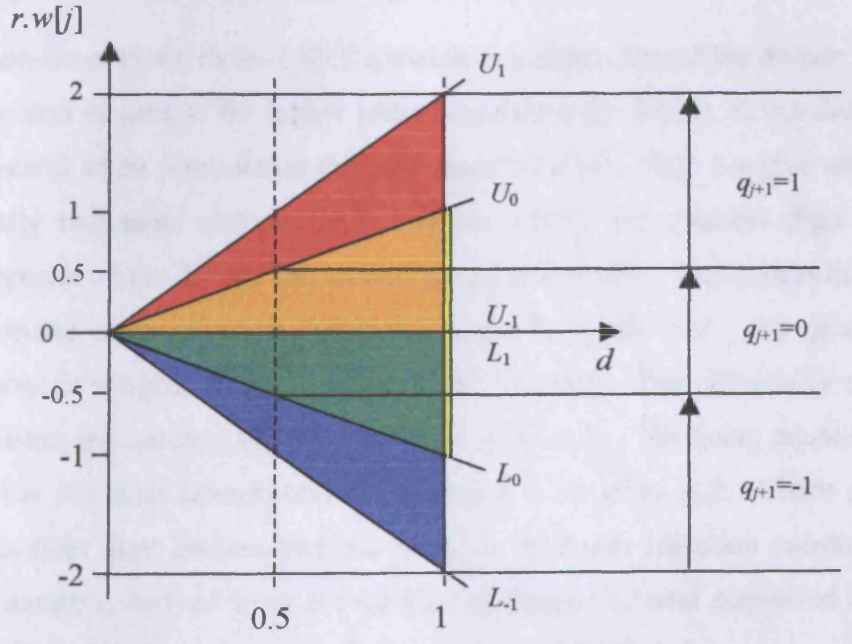


Figure A.1. A pd-diagram to illustrate the selection constants for a radix-2 SRT division algorithm with quotient digit set {-1,0,1}.

The selection function can be written as in (A.7).

$$q_{j+1} = \begin{cases} 1 \text{ if } 0.5 \le 2w[j] \\ 0 \text{ if } -0.5 \le 2w[j] < 0.5 \\ -1 \text{ if } 2w[j] < -0.5 \end{cases} \quad \text{(A.7)}$$

Equation (A.7) is used to determine the next quotient digit value. This quotient digit value is used to form multiples of the divisor to up date the shifted partial remainder as shown in equation (A.2). The digit is also the next digit of the result quotient. From (A.7) it appears that a full length comparison of the shifted partial remainder $r.w[j]$ with the values of ±0.5 are needed to determine the next quotient digit. However checking whether a number with a range (-2, 2) is greater than or equal to ±0.5 simply involves checking the three most significant bits. So determining the next quotient digit involves looking at the three MSBs instead of a full-length comparison. A correction step is needed at the end of SRT division to correct the remainder if it is negative. This involves adding on the value of the divisor to the negative remainder before it is correctly scaled. A correction of the final quotient is also required, which involves a subtraction of a single bit from the least significant position.

A.2 Radix-4 SRT division with non-redundant residual and maximally redundant quotient digit set

The selection function for radix-2 SRT division is independent of the divisor but it is a special case and in general for higher radix algorithms the MSBs of the divisor must also be checked when determining the next quotient digit. Here we give an example of maximally redundant radix-4 SRT division where the quotient digit selection function depends on the divisor and shifted partial remainder. This results in a pd-plot with a staircase style selection function shown in figure A.2. To generate the selection bounds we proceed as in radix-2 SRT division. For maximally redundant radix-4 division the quotient digit set is {-3,-2,-1,0,1,2,3}. By using equations (A.5) and (A.6) the selection bounds can be tabulated as in table A.2. There are seven different quotient digit choices and therefore six different selection constants. The selection constants, derived from the pd-plot of figure A.2 and numbered $M_{k,k+1}$ for the different quotient digits they separate, are shown in table A.3.

$q_{j+1}=k$	U_k	L_k
3	4d	2d
2	3d	d
1	2d	0
0	d	-d
-1	0	-2d
-2	-d	-3d
-3	-2d	-4d

Table A.2. Selection boundaries for radix-4 SRT division quotient digit set {-3,-2,-1,0,1,2,3}

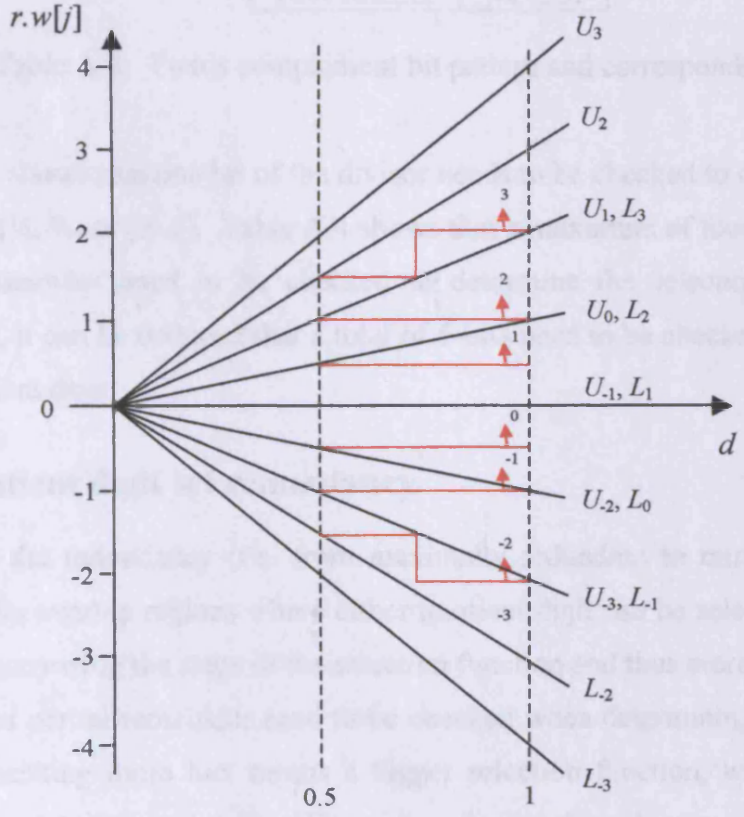


Figure A.2. A pd-plot for radix-4 SRT division

d \ $M_{k,k+1}$	$M_{-3,-2}$	$M_{-2,-1}$	$M_{-1,0}$	$M_{0,1}$	$M_{1,2}$	$M_{2,3}$
$\left[\frac{1}{2},\frac{3}{4}\right)$	-1.5	-1	-0.5	0.5	1	1.5
$\left[\frac{3}{4},1\right)$	-2	-1	-0.5	0.5	1	2

Table A.3. Selection constants for radix-4 maximally redundant SRT division

Bit pattern	Range
011.1XX...XX	[3.5, 4)
011.0XX...XX	[3, 3.5)
010.1XX...XX	[2.5, 3)
010.0XX...XX	[2, 2.5)
001.1XX...XX	[1.5, 2)
001.0XX...XX	[1, 1.5)
000.1XX...XX	[0.5, 1)
000.0XX...XX	[0, 0.5)
111.1XX...XX	[-0.5, 0)
111.0XX...XX	[-1, -0.5)
110.1XX...XX	[-1.5, -1)
110.0XX...XX	[-2, -1.5)
101.1XX...XX	[-2.5, -2)
101.0XX...XX	[-3, -2.5)
100.1XX...XX	[-3.5, -3)
100.0XX...XX	[-4, -3.5)

Table A.4. Two's complement bit pattern and corresponding range

Table A.3 shows that one bit of the divisor needs to be checked to determine if it is in the range [½, ¾) or [¾,1). Table A.4 shows that a maximum of four bits of the shifted partial remainder need to be checked to determine the selection region it is in. Therefore, it can be deduced that a total of 5-bits need to be checked to determine the next quotient digit.

A.3 Quotient digit set redundancy

Reducing the redundancy (i.e. from maximally redundant to minimally redundant) narrows the overlap regions where either quotient digit can be selected. This has the effect of narrowing the steps of the selection function and thus more bits of the divisor and shifted partial remainder need to be checked when determining the next quotient digit. Checking more bits means a bigger selection function, which increases the implementation area and delay. However reducing the redundancy means that fewer multiples of the divisor need to be created to update the shifted partial result.

Appendix B

Square root derivation

B.1 Sequential square root

We wish to compute $S = \sqrt{x}$, where x is the radicand, such that $x = S^2 + rem$.

$$S[n] = 0.s_1 s_2 \ldots s_n = S[0] + \sum_{i=1}^{n} 2^{-i}.s_i \quad \text{(B.1)}$$

$$x \in [0.25,1) \quad \therefore \quad S \in [0.5,1) \quad \text{(B.2)}$$

Each bit of S can be calculated by comparing x with various squares as follows:

$$s_1 = \begin{array}{l} 1 \; if \; x \geq (0.1)^2 \\ 0 \; otherwise \end{array} \quad \text{(B.3)}$$

$$s_2 = \begin{array}{l} 1 \; if \; x \geq (0.s_1 1)^2 \\ 0 \; otherwise \end{array} \quad \text{(B.4)}$$

$$s_3 = \begin{array}{l} 1 \; if \; x \geq (0.s_1 s_2 1)^2 \\ 0 \; otherwise \end{array} \quad \text{(B.5)}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$s_n = \begin{array}{l} 1 \; if \; x \geq (0.s_1 s_2 \ldots s_{n-1} 1)^2 \\ 0 \; otherwise \end{array} \quad \text{(B.6)}$$

If performed in a step-by-step manner then at step k the bits $s_1 s_2 \ldots s_{k-1}$ are known, thus by using a squarer and a comparator the next bit of S, s_k can be deduced. A variable width squarer is a costly component and so an alternative computation method is required.

Now, the comparison values can be written as:

$$(0.1)^2 = (0.1)^2 \quad \text{(B.7)}$$

$$(0.s_1 1)^2 = (0.s_1)^2 + 0.0s_1 01 \quad \text{(B.8)}$$

$$(0.s_1s_21)^2 = (0.s_1s_2)^2 + 0.00s_1s_201 \quad \text{(B.9)}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$(0.s_1s_2\ldots s_{n-1}1)^2 = (0.s_1s_2\ldots s_{n-1})^2 + 0.00\ldots0s_1s_2\ldots s_{n-1}01 \quad \text{(B.10)}$$

Replacing the comparison values into equations (B.3-B.6) gives:

$$s_1 = \begin{matrix} 1\ if\ x \geq (0.1)^2 \\ 0\ otherwise \end{matrix} \quad \text{(B.11)}$$

$$s_2 = \begin{matrix} 1\ if\ x - (0.s_1)^2 \geq 0.0s_101 \\ 0\ otherwise \end{matrix} \quad \text{(B.12)}$$

$$s_3 = \begin{matrix} 1\ if\ x - (0.s_1s_2)^2 \geq 0.00s_1s_201 \\ 0\ otherwise \end{matrix} \quad \text{(B.13)}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$s_n = \begin{matrix} 1\ if\ x - (0.s_1s_2\ldots s_{n-1})^2 \geq 0.00\ldots0s_1s_2\ldots s_{n-1}01 \\ 0\ otherwise \end{matrix} \quad \text{(B.14)}$$

Now if we let $w[k\text{-}1]$ and $t[k\text{-}1]$ denote the left and right hand sides respectively of the inequality at the start of iteration k then the following algorithm behaviour is evident:

Initially set $w[0] = x$ and $t[0] = (0.1)^2 = 0.01$

Iteration 1.

Determine s_1 by checking the sign of the comparison $w[0]$-$t[0]$.
Depending on s_1 $w[1]$ will either be $w[0]$-$(0.0)^2$ or $w[0]$-$(0.1)^2$
i.e. $w[0]$ or $w[0]$-$t[0]$ see equation (B.7). $t[1]$ is set to $0.0s_101$.

Iteration 2.

Determine s_2 by checking the sign of the comparison $w[1]$-$t[1]$.
Depending on s_2 $w[2]$ will either be $w[0]$-$(0.s_10)^2$ or $w[0]$-$(0.s_11)^2$
i.e. $w[1]$ or $w[1]$-$t[1]$ see equation (B.8). $t[2]$ is set to $0.00s_1s_201$.

Iteration j.

Determine s_j by checking the sign of the comparison $w[j\text{-}1]$-$t[j\text{-}1]$.
Depending on s_j $w[j]$ will either be $w[0]$-$(0.s_1s_2 \ldots s_{j-1}\ 0)^2$ or $w[0]$-$(0.s_1s_2 \ldots s_{j-1}1)^2$
i.e. $w[j\text{-}1]$ or $w[j\text{-}1]$- $t[j\text{-}1]$. $t[j]$ is set to $0.00\ldots0s_1s_2\ldots s_j01$.

B.2 The Restoring algorithm

Now we can write the above algorithm as a recurrence.

$$w[j] = w[j\text{-}1] - s_j(t[j\text{-}1]) \quad \text{(B.15)}$$

In equation (B.15) s_j must be determined by firstly calculating the tentative remainder $w[j-1] - t[j-1]$ and then checking the sign of the remainder, if negative, s_j is 0, if positive, s_j is 1.

We can define $S[j]$ to be the value of the square root result after j iterations (B.1). $t[j]$ can be written in terms of $S[j]$.

$$t[j-1] = 2^{-(j-1)}S[j-1]+2^{-2j} \quad \text{(B.16)}$$

Substituting (B.16) into (B.15) gives a recurrence of (B.17).

$$w[j] = w[j-1] - s_j(2^{-(j-1)}S[j-1]+2^{-2j}) \quad \text{(B.17)}$$

Where $w[j]$ is the remainder after j iterations such that $X = S^2 + w[j]$.

By scaling the partial remainder $w[j]$ for each iteration step the range of $w[j]$ can be constrained. Equation (B.18) shows the scaled recurrence.

$$w[j] = 2.w[j-1] - s_j(2.S[j-1]+2^{-j}) \quad \text{(B.18)}$$

Restoring algorithm

```
S[0] = 0;          w[0] = x;

For n = 1 to N

      w[n] = 2*w[n-1] - (2*S[n-1] + 2^-n);

      If (w[n] >= 0)                  % positive no need to restore
            S[n] = S[n-1] + 2^-n;
      Else                            % negative so restore
            S[n] = S[n-1];
            w[n] = 2*w[n-1];
      End if
End for
```

B.3 The non-restoring algorithm

As for division a non-restoring form of the recurrence can be developed. Following a similar method such that if the result of a subtraction is negative then an addition is performed for the next iteration and vice-versa. We now briefly describe the non-restoring operation. In cycle k a speculative subtraction of $(2^{-(j-1)}S[j-1]+2^{-2j})$ is performed (substitute k for j). If the resulting partial remainder is negative then a value needs to be added on that forces a speculative subtraction of $(2^{-(j-1)}S[j-1]+2^{-2j})$

for cycle k+1 (substitute k+1 for j). This is achieved in cycle k+1 by adding on $(2^{-(j-1)}.S[j-1] + 3.2^{-2j})$. Therefore the non-restoring algorithm can be written as:

Non-restoring algorithm

```
S[0] = 0;          w[0] = x;

For n = 1 to N
      If (w[n-1] >= 0)
            w[n] = w[n-1] - (2^-(n-1)*S[n-1] + 2^-2n);
      Else
            w[n] = w[n-1] + (2^-(n-1)*S[n-1] + 3*2^-2n);
      End if
      If (w[n] >= 0)
            S[n] = S[n-1] + 2^-n;
      Else
            S[n] = S[n-1];
      End if
End for
```

Simulation shows this algorithm produces the same square root as the restoring algorithm. The algorithm can also be scaled as for the restoring algorithm by changing the if-else clause lines in the above algorithm with the following lines:

```
      If (w[n-1] >= 0)
            w[n] = 2*w[n-1] - (2*S[n-1] + 2^-n);
      Else
            w[n] = 2*w[n-1] + (2*S[n-1] + 3*2^-n);
      End if
```

Simulation shows that the above non-restoring square root algorithm produces the correct result but the remainder needs to be corrected if it is negative. The remainder correction is done before the final remainder scaling by adding on (B.19).

$2*S[n]+2^{-n}$ ___(B.19)

References

FPGA datasheets

[1] Actel Corporation, "*ACT 1 Series FPGAs*," Device Datasheet, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. April 1996. www.actel.com.

[2] Actel Corporation, "*ACT 2 Family FPGAs*," Device Datasheet v4.0.1, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. December 2000. www.actel.com.

[3] Actel Corporation, "*Integerator Series FPGAs: 1200XL and 3200DX Families*," Device Datasheet v3.0, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. February 2001. www.actel.com.

[4] Actel Corporation, "*Accelerator Series FPGAs-ACT 3 Family*," Device Datasheet, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. September 1997. www.actel.com.

[5] Actel Corporation, "*Accelerator Series FPGAs: ACT 3 PCI-Compliant Family*," Device Datasheet, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. March 1997. www.actel.com.

[6] Actel Corporation, "*40MX and 42MX FPGA Families*," Device Datasheet v5.0, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. February 2001. www.actel.com.

[7] Actel Corporation, "*eX Family FPGAs*," Device Datasheet v3.0, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. December 2001. www.actel.com.

[8] Actel Corporation, "*SX-A Family FPGAs*," Device Datasheet v4.0, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. April 2003. www.actel.com.

[9] Actel Corporation, "*Axcelerator Family FPGAs*," Device Datasheet v1.5, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. March 2003. www.actel.com.

[10] Actel Corporation, "*ProASIC 500K Family*," Device Datasheet v3.0, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. February 2002. www.actel.com.

[11] Actel Corporation, "*ProASIC^{PLUS} Flash Family FPGAs*," Device Datasheet v3.5, Actel Corporation, 2061 Stierlin Court, Mountain View, CA, USA. April 2004. www.actel.com.

[12] Atmel Corporation, "*Coprocessor Field Programmable Gate Arrays. AT6000(LV) Series*," Device Datasheet, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA, USA. October 1999. www.atmel.com.

[13] Atmel Corporation, "*5K-50K Gates Coprocessor FPGA with FreeRAM. AT40K(LV),*" Device Datasheet, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA, USA. April 2002. www.atmel.com.

[14] Quicklogic Corporation, "*pASIC 1 Family,*" Device Datasheet, Quicklogic Corporation, Sunnyvale, CA, USA. www.quicklogic.com.

[15] Quicklogic Corporation, "*pASIC 2 FPGA Family,*" Device Datasheet Rev. D, Quicklogic Corporation, Sunnyvale, CA, USA. www.quicklogic.com.

[16] Quicklogic Corporation, "*pASIC 3 FPGA Family,*" Device Datasheet Rev. B, Quicklogic Corporation, Sunnyvale, CA, USA. www.quicklogic.com.

[17] Quicklogic Corporation, "*QuickRAM ESP Family,*" Device Datasheet Rev. H, Quicklogic Corporation, Sunnyvale, CA, USA. www.quicklogic.com.

[18] Quicklogic Corporation, "*Eclipse Family Data Sheet,*" Device Datasheet Rev. C, Quicklogic Corporation, Sunnyvale, CA, USA. 2003. www.quicklogic.com.

[19] Quicklogic Corporation, "*EclipsePlus Data Sheet,*" Device Datasheet, Quicklogic Corporation, Sunnyvale, CA, USA. 2003. www.quicklogic.com.

[20] Lattice Semiconductor Corporation, "*ORCA Series 2 Field-Programmable Gate Arrays,*" Device Datasheet, October 2003. www.latticesemi.com.

[21] Lattice Semiconductor Corporation, "*ORCA Series 3C and 3T Field-Programmable Gate Arrays,*" Device Datasheet, November 2003. www.latticesemi.com.

[22] Lattice Semiconductor Corporation, "*ORCA OR3LxxxB Series Field-Programmable Gate Arrays,*" Device Datasheet, March 2002. www.latticesemi.com.

[23] Lattice Semiconductor Corporation, "*ORCA Series 4 FPGAs,*" Device Datasheet, November 2003. www.latticesemi.com.

[24] Lattice Semiconductor Corporation, "*ispXPGA Family,*" Device Datasheet, June 2004. www.latticesemi.com.

[25] Lattice Semiconductor Corporation, "*LatticeECP/EC Family Data Sheet,*" Device Datasheet, June 2004. www.latticesemi.com.

[26] Xilinx, Inc., "*XC3000 Series Field Programmable Gate Arrays (XC3000A/L, XC3100A/L),*" Device Datasheet v3.1, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. November 9th 1998. www.xilinx.com.

[27] Xilinx, Inc., "*XC4000E and XC4000X Series Field Programmable Gate Arrays,*" Device Datasheet v1.6, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. May 14th 1999. www.xilinx.com.

[28] Xilinx, Inc., "*XC4000XLA/XV Field Programmable Gate Arrays*," Device Datasheet v1.3, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. October 18th 1999. www.xilinx.com.

[29] Xilinx, Inc., "*XC5200 Series Field Programmable Gate Arrays*," Device Datasheet v5.2, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. November 5th 1998. www.xilinx.com.

[30] Xilinx, Inc., "*Spartan and Spartan-XL Families Field Programmable Gate Arrays*," Device Datasheet v1.7, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. June 27th 2002. www.xilinx.com.

[31] Xilinx, Inc., "*Spartan-II 2.5V FPGA Family: Complete Data Sheet*," Device Datasheet, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. September 3rd 2003. www.xilinx.com.

[31] Xilinx, Inc., "*Spartan-IIE 1.8V FPGA Family: Complete Data Sheet*," Device Datasheet, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. July 9th 2003. www.xilinx.com.

[32] Xilinx, Inc., "*Spartan-3 FPGA Family: Complete Data Sheet*," Device Datasheet, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. March 4th 2004. www.xilinx.com.

[33] Xilinx, Inc., "*Virtex 2.5V Field Programmable Gate Arrays*," Device Datasheet v2.5, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. April 2nd 2001. www.xilinx.com.

[34] Xilinx, Inc., "*Virtex-E 1.8V Field Programmable Gate Arrays*," Device Datasheet v2.3, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. July 17th 2002. www.xilinx.com.

[35] Xilinx, Inc., "*Virtex-II Platform FPGAs: Complete Data Sheet*," Device Datasheet v3.3, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. June 24th 2004. www.xilinx.com.

[36] Xilinx, Inc., "*Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*," Device Datasheet v4.0, Xilinx, Inc., 2100 Logic Drive, San Jose, CA, USA. June 30th 2004. www.xilinx.com.

[37] Altera Corporation, "*FLEX 6000 Programmable Logic Device Family*," Device Datasheet v4.1, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. March 2001. www.altera.com.

[38] Altera Corporation, "*FLEX 8000 Programmable Logic Device Family*," Device Datasheet v11.1, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. January 2003. www.altera.com.

[39] Altera Corporation, "*FLEX 10K Embedded Programmable Logic Device Family*," Device Datasheet v4.2, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. January 2003. www.altera.com.

[40] Altera Corporation, "*FLEX 10KE Embedded Programmable Logic Device Family*," Device Datasheet v2.5, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. January 2003. www.altera.com.

[41] Altera Corporation, "*ACEX 1K Programmable Logic Device Family*," Device Datasheet v3.4, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. May 2003. www.altera.com.

[42] Altera Corporation, "*APEX 20K Programmable Logic Device Family*," Device Datasheet v4.3, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. February 2002. www.altera.com.

[43] Altera Corporation, "*APEX II The Complete I/O Solution*," Device Product Brochure, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. July 2002. www.altera.com.

[44] Altera Corporation, "*Mercury Programmable Logic Device Family*," Device Datasheet v2.2, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. January 2003. www.altera.com.

[45] Altera Corporation, "*Cyclone Device Handbook, Volume 1*," Device Datasheet, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. 2004. www.altera.com.

[46] Altera Corporation, "*Cyclone II Device Handbook, Volume 1*," Device Datasheet, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. 2004. www.altera.com.

[47] Altera Corporation, "*Stratix Device Handbook, Volume 1*," Device Datasheet, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. 2004. www.altera.com.

[48] Altera Corporation, "*Stratix II Device Handbook, Volume 1*," Device Datasheet, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. 2004. www.altera.com.

[49] Altera Corporation, "*Stratix GX FPGA Family*," Device Datasheet v2.1, Altera Corporation, 101 Innovation Drive, San Jose, CA, USA. February 2004. www.altera.com.

Author's publications

[50] B. R. Lee, N. Burgess, "Improved small multiplier based multiplication, squaring and division," *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, Napa, California, 2003. pp. 91-97.

[51] B. R. Lee, N. Burgess, "Parameterisable floating-operations on FPGA," *Proceedings of the 36th IEEE Asilomar Conference on Signals, Systems and Computers*, 2002. pp. 1064-1068.

[52] B. R. Lee, N. Burgess, "Some results on Taylor-series function approximation on FPGA," *Proceedings of the 37th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. 2, November 9th-12th 2003. pp. 2198-2202.

[53] B. R. Lee, N. Burgess, "A dual-path logarithmic number system addition/subtraction scheme for FPGA," *Proceedings 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Portugal, 1st-3rd September 2003. pp. 808-817.

[54] B. R. Lee, N. Burgess, "A parallel look-up logarithmic number system addition/subtraction scheme for FPGA," *Proceedings of the 2003 International Conference on Field Programmable Technology (FPT)*, Tokyo, Japan, 15th-17th December 2003. pp.76-83.

[55] B. R. Lee, K. Lever, "Logarithmic number system and floating-point implementations of a well-conditioned RLS estimation algorithm on FPGA," *Proceedings of the 37th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. I, November 9th-12th 2003. pp. 109-113.

ASIC fixed-point

[56] A. R. Omondi, "*Computer arithmetic systems: algorithms, architecture and implementations*," Prentice Hall International Ltd, Hemel Hempstead, UK. 1994. ISBN 0-13-334301-4.

[57] M. D. Ercegovac, T. Lang, "*Division and square root: digit-recurrence algorithms and implementations*," Kluwer Academic Publishers, 1994.

[58] A. Jeffery, "*Mathematics for engineers and scientists - Fifth edition*," Chapman and Hall, London, U.K.1996. ISBN 0-412-62150-9.

[59] B. Parharmi, "*Computer arithmetic: algorithms and hardware designs*," Oxford University Press. 2000. ISBN 0195125835.

[60] I. Koren, "*Computer arithmetic algorithms – 2nd Edition*," A. K. Peters Ltd, Natick, MA. 2002. ISBN 1-56881-160-8.

[61] M. D. Ercegovac, T. Lang, "*Digital arithmetic*," Morgan Kaufmann Publishers, San Francisco, USA. 2004. ISBN 1-55860-798-6.

[62] D. E. Knuth, "*The art of computer programming - 2nd Edition*," Vol. II, Addison-Wesley, Reading, Massachusetts, USA. 1981. ISBN 0-201-03822-6 (v.2).

[63] A. Avizienis, "Signed digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, Vol. EC-10, No. 9, 1961. pp. 389-400.

[64] A. C. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 4, No. 2, 1951. pp. 236-240.

[65] O. L. MacSorley, "High-speed arithmetic in binary computers," *IRE Proceedings*, Vol. 49, 1961. pp. 67-91.

[66] A. Karatsuba, Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Physics-Doklaty*, Vol. 7, No. 7, 1962. pp. 595-596.

[67] C. S. Wallace, "A suggestion for a fast multiplier," *IRE Transactions on Electronic Computers*, Vol. EC-13, No. 2, 1964. pp.14-17.

[68] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, Vol. 34, 1965. pp. 349-356.

[69] H. H. Guild, "Fully iterative fast arrays for binary multiplication and addition," *Electronic Letters*, Vol. 5, 1969. pp. 263.

[70] J. C. Majithia, R. Kitai, "An iterative array for multiplication of signed binary number," *IEEE Transactions on Computers*, Vol. C-20, No. 2, 1971. pp. 214-216.

[71] S. D. Pezaris, "A 40-ns 17-bit by 17-bit array multiplier," *IEEE Transactions on Computers*, Vol. C-20, No. 4, 1971. pp. 442-447.

[72] S. Bandhopadhyay, B. Basu, A. K. Choudhary, "An iterative array for multiplication of signed binary number," *IEEE Transactions on Computers*, Vol. C-21, No. 8, 1972. pp. 921-922.

[73] C. R. Baugh, B. A. Wooley, "A twos complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, Vol. C-22, No. 12, 1973. pp. 1045-1047.

[74] R. E. Goldschmidt, "*Applications of division by convergence*," Master's thesis, Massachusetts Institute of Technology. 1964.

[75] D. L. Fowler, J. E. Smith, "An accurate, high speed implementation of division by reciprocal approximation," *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, September 1989. pp. 60-67.

[76] D. C. Wong, M. J. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, 1991. pp. 191-201.

[77] M. D. Ercegovac, T. Lang, P. Montuschi, "Very-high radix division with prescaling and selection by rounding," *IEEE Transactions on Computers*, Vol. 43, No. 8, August 1994. pp. 909-918.

[78] S. F. Oberman, M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, Vol. 46, No. 8, August 1997. pp. 833-854.

FPGA fixed-point

[79] J. Stohmann, E. Barke, "An universal CLA adder generator for SRAM-based FPGAs," *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'96)*, 1996. pp. 44-45.

[80] S. Xing, W. W. H. Yu, "FPGA adders: performance evaluation and optimal design," *IEEE design and test of computers*, January-March 1998. pp. 24-29.

[81] S. Perri, "Speed-efficient wide adders for Virtex FPGAs," *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS'02)*, Croatia, September 2002. pp. 599-602.

[82] L. M. Smith, B. Nowrouzian, "Fixed-point bit-serial implementations of LDI Jaumann digital filters," *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, May 1993. pp. 112-115.

[83] R. J. Andraka, "FIR filters in an FPGA using a bit-serial approach," *Proceedings of the 3rd Annual PLD Conference*, March 1993.

[84] J. H. Satyanarayana, B. Nowrouzian, "A comprehensive approach to the design of digit-serial modified Booth multipliers," *Proceedings of the 26th IEEE Southeastern Symposium on System Theory*, Athens, OH, March 1994. pp. 229-233.

[85] S. He, M. Torkelson, "FPGA implementation of FIR filters using pipelined bit-serial canonical signed digit multipliers," *Proceedings IEEE 1994 Custom Integrated Circuits Conference*, 1994. pp. 5.3.1-5.3.4.

[86] L. E. Turner, P. J. W. Graumann, S. G. Gibb, "Bit-serial FOR filters with CSD coefficients for FPGAs, "*Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, 1995. pp. 311-320.

[87] J. Li, Y. Du, J. Wang, "Design a pocket multi-bit multiplier in FPGA," 1996. pp. 275-279.

[88] V. Rao, B. Nowrouzian, "A novel approach to the design and implementation of very high-speed digit-serial modified-Booth multipliers," *Proceedings of the 39th IEEE Midwest Symposium on Circuits and Systems*, August 1997. pp. 61-64.

[89] H. Lee, G. Sobelman, "FPGA-based FIR filters using digit-serial arithmetic," *Proceedings of the Tenth Annual IEEE International ASIC Conferenne and Exhibit*, 1997. pp. 225-228.

[90] A. Khalil, M. A. Ashour, A. E. Salama, H. I. Saleh, "FPGA implemented fast two's complement serial-parallel multiplier with PCI interface," *Proceedings of the 10th International Conference on Microelectronics*, December 1998. pp. 21-24.

[91] H. Lee, G. E. Sobelman, "Digit-serial DSP library for optimized FPGA configuration," *Proceedings of the 6th IEEE Symposium on Field-programmable Custom Computing Machines (FCCM'98)*, Napa, CA, April 1998.

[92] J. Valls, M. M. Peiro, T. Sansaloni, E. Boemo, "A study about FPGA-based digital filters," *IEEE Workshop on VLSI Signal Processing: Design and implementation (SiPS'98)*, 1998, Boston, MA. pp. 192-201.

[93] J. Valls, T. Sansaloni, M. M. Peiro, E. Boemo, "Fast FPGA-based pipelined digit-serial/parallel multipliers," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'99)*, Vol. 1, Orlando, Florida, 1999. pp.482-485.

[94] H. Lee, J. Chung, G. E. Sobelman, "FPGA-based digit-serial CSD for filter for image signal format conversion," *International Conference on Signal Processing Applications and Technology (ICSPAT'98)*, September 1998.

[95] K. D. Adaos, G. P. Alexiou, N. Kanopoulos, "Efficient implementation of a serial/parallel multiplier for IP based development and rapid prototyping in VLSI digital signal processing," *Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems (ICECS'99)*, 1999. pp. 33-36.

[96] T. Sansaloni, J. Valls, K. K. Parhi, "FPGA-based digit-serial complex number multiplier-accumulator," *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 28th – 31st 2000, Geneva, Switzerland. Vol. IV. pp. 585-588.

[97] C. Coulston, "Sequential truncated multiplication," *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, August 2001. pp. 356-358.

[98] O. Nibouche, A. Bouridane, M. Nibouche, "New architectures for serial-serial multiplication," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'01)*, Sydney, Australia, May 2001. Vol. II. pp. 705-708.

[99] T. Sansaloni, J. Valls, K. K. Parhi, "Digit-serial complex-number multipliers on FPGAs," *Journal of VLSI Signal Processing 33*, Kluwer Academic Publishers, 2003. pp. 105-115.

[100] M. Daumas, J-M. Muller, J. Vuillemin, "Implementing on-line arithmetic with PAM," *Proceeding of the 4th International Workshop on Field-Programmable Logic and Applications*, 1994. pp. 196-207.

[101] B. Girau, A. Tisserand, "On-line arithmetic-based reprogrammable hardware implementation of multilayer perceptron back-propagation," *Proceedings of MicroNeuro'96*, IEEE, 1996. pp. 168-175.

[102] A. Tisserand, M. Dimmler, "FPGA implementation of real-time digital controllers using online arithmetic," *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL'97)*, LNCS 1304, September 1997. pp. 472-481.

[103] R. McIlhenny, M. D. Ercegovac, "Online algorithms for complex number arithmetic," *Proceedings of the 32nd IEEE Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, October 1998. pp. 172-176.

[104] A. Tenca, M. D. Ercegovac, "A variable long-precision arithmetic unit design for reconfigurable coprocessor architectures," *Proceedings of the 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, 1999.

[105] D. Lau, A. Schneider, M. D. Ercegovac, "FPGA structures for on-line FFT and DCT," *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, April 1999. pp. 310-311.

[106] D. Lau, A. Schnieder, M. D. Ercegovac, "A FPGA-based library for on-line signal processing," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, Vol. 28, No. 1-2, May-June 2000. pp. 129-134.

[107] J. Valls, M. Kuhlmann, K. K. Parhi, "Efficient mapping of CORDIC algorithms on FPGA," *IEEE Workshop on Signal Processing Systems (SiPS'00)*, October 2000. pp. 336-345.

[108] A. Schneider, R. McIlhenny, M. D. Ercegovac, "BigSky an online arithmetic design tool for FPGAs," *Proceedings of the 8th IEEE Symposium on Field-programmable Custom Computing Machines (FCCM'00)*, Napa, CA, 17th-19th April 2000. pp. 303-304.

[109] A. P-Pascual, T. Sansaloni, J. Valls, "FPGA based on-line complex-number multipliers," *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS'01)*, 2001. pp. 1481-1484.

[110] A. F. Tenca, S. U. Hussaini, "A design of radix-2 online division using LSA organisation," *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, 11th-13th June 2001. pp. 266-273.

[111] W. G. Natter, B. Nowrouzian, "A novel multiplier recoding technique and its application to the development of a high-speed parallel online multiply-accumulate architecture," *IEEE Symposium on Circuits and Systems*, 2001. Vol. II. pp. 713-716.

[112] A. Saha, R. Krishnamurthy, "Rapid prototyping and performance evaluation of recoded multipliers using FPGAs," *Proceedings of the IEEE Southeast con'94*, 1994. pp. 236-240.

[113] A. Saha, R. Krishnamurthy, "Implementation and performance evaluation of cellular array multipliers using FPGAs," *Proceedings of the 26th IEEE Southeastern Symposium on System Theory*, 20th-22nd March 1994. pp. 224-228.

[114] M. E. Louie, M. D. Ercegovac, "A variable precision multiplier generator for field programmable gate arrays," *Proceedings of the ACM Second International Workshop on FPGAs*, Berkeley, California, February 1994.

[115] R. H. Strandberg, J-C. L. Duc, L. G. Bustamante, V. G. Oklobdzija, M. A. Soderstrand, "Implementation of adaptive sample rate Kwan-Martin notch filter using efficient realizations of reciprocal and squaring circuit," *Proceedings of the 28th IEEE Asilomar Conference on Signals, Systems and Computers*, 1995. pp. 324-328.

[116] S. Kumar, K. Forward, M. Palaniswami, "A fast-multiplier generator for FPGAs," *Proceedings of the 8th International Conference on VLSI Design*, January 1995. pp. 53-56.

[117] R. W. Canik, E. E. Swartzlander Jr., "Implementing array multipliers in Xilinx FPGAs," *Proceedings of the 28th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. II, 31st October-2nd November 1995. pp. 1378-1382.

[118] J. P. Singh, A. Kumar, S. Kumar, "A multiplier generator for Xilinx FPGAs," *Proceedings of the 9th International Conference on VLSI design*, January 1996. pp. 322-323.

[119] P. L. Ruiz, T. Riesgo, J. Uceda, "Design and prototyping of DSP custom circuits based on a library of arithmetic circuits," *IEEE Industrial Electronics Conference (IECON)*, Neuva Orleans, USA, November 1997. pp. 191-196.

[120] J. Stohamm, E. Barke, "A universal Pezaris array multiplier generator for SRAM-based FPGAs," *IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, 12th-15th October 1997. pp. 489-495.

[121] H. Ho, V. Szwarc, T. A. Kwasniewski, "Pipelined digital design in SRAM FPGAs," *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'97)*, 1997. pp. 23-26.

[122] S. Tagzout, L. Sahli, "Compact parallel multipliers using the sign-generate method in FPGA," *Proceedings of the 21st International Conference on Microelectronics (MIEL'97)*, Vol.2, Yugoslavia, 14th – 17th September 1997. pp. 815-818.

[123] B. Laurent, G. Bosco, G. Saucier, "Fast arithmetic on Xilinx 5200 FPGA," *Proceedings of the 11th IEEE International Conference on VLSI Design*, 4th-7th June 1998. pp. 322-325.

[124] M. A. Thornton, J. D. Gaiche, J. V. Lemieux, "Tradeoff analysis of integer multiplier circuits implemented in FPGAs," *IEEE Pacific Rim Conference on Communications, Computer and Signal Processing (PACRIM)*, August 22nd-24th 1999. pp. 301-304.

[125] S. Shah, A. J. Al-Khalili, D. Al-Khalili, "Comparison of 32-bit multipliers for various performance measures," *Proceedings of the 12th International Conference on Microclectronics*, Tehran, Iran, October 31st – November 2nd 2000. pp. 75-80.

[126] T. Courtney, R. Turner, R. Woods, "Multiplexer based reconfiguration for Virtex multipliers," *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*, (LNCS 1896), August 27th-30th 2000. pp. 749-758.

[127] K. Chapman, "Expanding Virtex-II™," Xilinx, UK, June, 2001. www.xilinx.com.

[128] J-L. Beuchat, A. Tisserand, "Small multiplier-based multiplication and division operators for Virtex-II devices," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France, September 2002. LNCS 2438, pp. 513-522.

[129] A. J. Al-Khalili, A. Hu, "Design of a 32-bit squarer – exploiting addition redundancy," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'03)*, 2003. Vol. V. pp. 325-328.

[130] M. E. Louie, M. D. Ercegovac, "Mapping division algorithms to field programmable gate arrays," *Proceedings of the 26th IEEE Asilomar Conference on Signals, Systems and Computers*, October 26th-28th 1992. pp. 371-375.

[131] M. E. Louie, M. D. Ercegovac, "On digit-recurrence division implementations for field programmable gate arrays," *Proceedings of the 11th IEEE Symposium on Computer arithmetic*, Windsor, Canada, June 1993. pp 202-209.

[132] Xilinx, "*LogiCore pipelined divider v2.0*," Product specification, 30th June 2000. www.xilinx.com

[133] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, D. Poirier, "A flexible floating-point format for optimising data-paths and operators in FPGA based DSPs," *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'02)*, Monterey, California, USA, February 24th-26th 2002. pp. 50-55.

[134] E. Roesler, B. Nelson, "Novel optimisations for hardware floating-point units in modern FPGA architecture," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France, September 2002. LNCS 2483. pp. 637-646.

[135] X. Wang, B. Nelson, "Tradeoffs of designing floating-point division and square root on Virtex FPGAs," *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, April 2003. pp. 195

[136] M. E. Louie, M. D. Ercegovac, "A digit-recurrence square root implementation for field programmable gate arrays," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, 5th-7th April 1993. pp. 178-183.

[137] I. V. Klotchkov, S. Pedersen, "A codesign case study: implementing arithmetic functions in FPGA's ," *IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96)*, March 11th-15th 1996. pp. 389-394.

[138] V. Tchoumatchenko, T. Vassileva, P. Gurov, "A FPGA based square-root coprocessor," *Proceedings of IEEE Euromicro-22*, 1996. pp. 520-525.

[139] Z. Luo, M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," *IEEE Transactions on Computers*, Vol. 49, No. 3, March 2000. pp. 208-218.

ASIC floating-point

[140] M. P. Farmwald, "*On the design of high performance digital arithmetic units*," PhD thesis, Stanford University, USA,1981.

[141] ANSI/IEEE, "*IEEE standard for binary floating-point arithmetic*," ANSI/IEEE standard, std 754, New York, USA, 12th August 1985.

[142] D .Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, March 1991.

[143] V. G. Oklodzija, "Algorithmic design of a hierarchical an modular leading zero detector circuit," *Electronics Letters*, Vol. 29, No. 3, 4th February 1993. pp. 283-284.

[144] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, T. Sumi, "Leading-zero anticipatory logic for high-speed floating-point addition," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 8, August 1996. pp. 1157-1164.

[145] W. Kahan, "*Lecture notes on the status of 'IEEE Standard 754 for Binary Floating-Point Arithmetic'*," University of California, Berkley, CA. 31st May 1996.

[146] A. M. Nielsen, D. W. Matula, C. N. Lyu, G. Even, "Pipelined packet-forwarding floating-point: II. An Adder," *Proceedings 13th IEEE International Symposium on Computer Arithmetic*, March 6th-9th 1997. pp. 148-155.

[147] J. F. Blinn, "Floating-point tricks," *IEEE Journal of Computer Graphics and Applications*, July/August 1997. pp. 80-84.

[148] G. Even, P-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," *Proceedings of the 14th IEEE International Symposium on Computer Arithmetic*, Adelaide, Australia, April 1999. pp. 225-232.

[149] J. D. Bruguera, T. Lang, "Leading-one prediction with concurrent position correction," *IEEE Transactions on Computers*, Vol. 48, No. 10, October 1999. pp. 1083-1097.

[150] A. Beaumont-Smith, N. Burgess, S. Lefrere, C. C. Lim, "Reduced latency IEEE floating-point standard adder architectures," *Proceedings of the 14th IEEE International Symposium on Computer Arithmetic (ARITH14)*, 1999. pp. 35-44.

[151] N. Burgess, S. Knowles, "Efficient implementation of rounding units," *Proceedings of the 33rd IEEE Asilomar Conference on Signals, Systems, and Computers*, Vol. 2. 1999. pp. 1489-1493.

[152] P-M. Seidel, G. Even, "On the design of fast IEEE floating-point adders," *Proceedings of the 15th IEEE International Symposium on Computer Arithmetic (ARITH15)*, 2001. pp.184-194.

[153] E. M. Schwarz, M. Schmookler, S. D. Trong, "Hardware implementations of denormalized numbers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, 15th-18th June 2003. pp.70-78.

FPGA floating-point

[154] D. Narasimhan, D. Fernandes, V. K. Raj, J. Dorenbosch, M. Bowden, V. S. Kapoor, "A 100 MHz FPGA based floating-point adder," *Proceedings 1993 IEEE Custom Integrated Circuits Conference*, 1993. pp. 3.1.1-3.1.4.

[155] J. H. Novak, E. Brunvand, "Using FPGAs prototype a self-timed floating-point co-processor," *Proceedings 1994 IEEE Custom Integrated Circuits Conference*, 1994. pp. 5.4.1-5.4.4.

[156] B. Fagin, C. Renard, "Field programmable gate arrays and floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 3, September 1994. pp. 365-367.

[157] N. Shirazi, A. Walters, P. Athanas, "Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM95)*, April1995. pp. 155-162.

[158] L. Louca, T. A. Cook, W. H, Johnson, "Implementation of IEEE single precision floating-point addition and multiplication on FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM96)*, 1996. pp. 107-116.

[159] L. Samet, N. Masmoudi, M. W. Kharrat, L. Kamoun, "Un multiplier 32 bits a virgule flottante sur FPGA," *Proceedings Conference International JTEA '97*. 1997.

[160] Y. Li, W, Chu, "Implementation of single precision floating-point square root on FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM97)*, 1997. pp. 226-232.

[161] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998. pp.206-215.

[162] C. Souani, M. Abid, R. Tourki, "An FPGA implementation of the floating-point addition," *Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society (IECON'98)*, Vol. 3, 31st August- 4th September 1998. pp. 1644-1648.

[163] A. Walters, P. Athanas, "A Scaleable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998. pp. 333-334.

[164] I. Stamoulis, M. White, P. F. Lister, "Pipelined floating-point arithmetic optimised for FPGA architectures," *Lecture Notes in Computer Science*, 1999, LNCS 1673, pp. 365-370.

[165] I. Sahin, C. S. Gloster, C. Doss, "Feasibility of floating-point arithmetic in reconfigurable computing systems," *Military and Aerospace Applications of Programmable Devices and Technology (MAPLD) on Adaptive Computing*, Vol. 3, September 2000.

[166] R. V. K. Pillai, S. Y. A. Shah, A. J. Al-Khalili, D. Al-Khalili, "Low power floating-point MAFs – a comparative study," *Proceedings of the IEEE International Symposium on Signal Processing and its Applications (ISSPA)*, Kuala Lumpur, Malaysia, 13th-16th August 2001. pp. 284-287.

[167] A. Jaenicke, W. Luk, "Parameterised floating-point arithmetic on FPGAs," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2001. pp. 897-900.

[168] Digital Core Design, "*Alliance Core Datasheets for floating-point arithmetic*," 2001. www.xilinx.com

[169] C. H. Ho, M. P. Leong, P. H. W. Leong, J. Becker, M. Glesner, "Rapid prototyping of FPGA based floating-point DSP systems," *Proceedings of the 13th IEEE International Workshop on Rapid System Prototyping (RSP'02)*, Darmstadt, 2002. pp.19-24.

[170] P. Belanovic, M. Leeser, "A library of parameterized floating-point modules and their use," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France, September 2002. LNCS 2483. pp. 657-666.

[171] P. Belanovic, "*Library of parameterized hardware modules for floating-point arithmetic with an example application*," MSc Masters Thesis, Northeastern University, Boston, Massachusetts, May 2002.

[172] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, J. Hwang, "Automating customisation of floating-point designs," *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France, September 2002. LNCS 2483. pp. 523-533.

[173] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," *2002 IEEE International Conference on Field-programmable (FPT)*, Hong-Kong, December 2002. pp. 158-165.

[174] K. R. Nichols, M. A. Moussa, S. M. Areibi, "Feasibility of floating-point arithmetic in FPGA based artificial neural networks," *CAINE*, San Diego, California, November 2002. pp.8-13.

[175] Nallatech Limited, "*Floating-point cores*," Commercial Floating-point Core Library Datasheet, 2002. www.nallatech.com.

[176] QinetiQ Limited, "*Quixilica floating-point FPGA cores*," Commercial Floating-point Core Library Datasheet, 2002. www.quixilica.com

[177] M. Bera, G. Danese, I. De Lotto, F. Leporati, A. Spelgatti, "A development and simulation environment for floating-point operations FPGA based accelerator," *Proceedings of the Euromicro Symposium on Digital System Design (DSD'03)*, 2003. pp. 173-179.

[178] J. Liang, R. Tessier, O. Mencer, "Floating-point unit generation and evaluation for FPGAs," *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, April 2003.

[179] I. O. Flores, M. Jimenez, "Scalable pipeline insertion in floating-point units for FPGA synthesis," *IASTED Conference on Circuits and Systems IASTED/CSS-2003*, Cancun, Mexico, May 2003.

[180] J. Detrey, "Biblotheque d'operateurs parametrables pour l'arithmetic "reelle" sur FPGA," *Research Report No. 2003-07*, ENS, Lyon, France. 2003.

[181] S. Paschalakis, P. Lee, "Double precision floating-point arithmetic on FPGAs," *2003 IEEE International Conference on Field-programmable (FPT), Tokyo*, Japan, December 15th-17th 2003. pp. 352-358.

[182] Celoxica Limited, "*Celoxica pipelined floating-point 32-bit macros DK2 SP1*," 27th August 2003. www.Celoxica.com

ASIC Function evaluation

[183] D. D. Sarma, D. W. Matula, "Faithful bipartite ROM reciprocal table," *Proceedings of the IEEE 12th Symposium on Computer Arithmetic*, Bath, UK. 1995. pp. 17-28.

[184] M. J. Schulte, J. E. Stine, "Symmetric bipartite tables for accurate function approximation," *Proceedings of the 13th IEEE Symposium on Computer Arithmetic (ARITH'13)*, 1997. pp. 175-183.

[185] M. J. Schulte, J. E. Stine, "Accurate function approximation by symmetric table lookup and addition," *Proceedings of the 11th International Conference on Application-Specific Systems, Architectures, and Processors*, Zurich, Switzerland, July, 1997. pp. 144-153.

[186] F. de Dinechin, A. Tisserand, "Some improvements on multipartite table methods," *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, June 2001. pp. 128-135.

[187] H. Hassler, N. Tagaki, "Function evaluation by table lookup and addition," *Proceedings of the 12th IEEE International Symposium on Computer Arithmetic*, 1995. pp. 10-16.

[188] W. F. Wong, E. Goto, "Fast evaluation of the elementary functions in single precision," *IEEE Transactions on Computers*, Vol. 44, No. 3, March 1995. pp. 453-457.

[189] H-Y. Lo, H-F. Lin, Y-Y. Ho, "Logarithmic conversion by four partitioned hybrid-ROMs," *IEEE International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'96)*, June 12th-14th 1996. pp. 550-552.

[190] Y. Wan, C. L. Wey, "Efficient algorithms for binary logarithm conversion and addition," *IEE Proceedings, Comp. Digit. Tech.*, Vol. 146, No. 3, May 1999. pp. 168-172.

[191] M. R. Spiegel, J. Liu, "*Mathematical handbook of formulas and tables – Second edition*," McGraw-Hill, New York, USA. 1999. ISBN 0-07-038203-4.

[192] A. S. Noetzel, "An interpolating memory unit for function evaluation: analysis and design," *IEEE Transactions on Computers*, Vol.38, No.3, March 1989. pp. 377-384.

[193] E. W. Cheney, "*Introduction to approximation theory*," Chelsea, New York, Second Edition. 1986. ISBN 0-821-81374-9.

[194] J. A. Pineiro, J. D. Bruguera, J-M. Muller, "Faithful powering computation using table lookup and a fused accumulation tree," *Proceedings of the 15th IEEE International Symposium on Computer Arithmetic (ARITH'15)*, 2001. pp. 40-47.

[195] J-M. Muller, "*Elementary functions, algorithms and implementations*," Boston, Birkhauser.1997. ISBN 081763990X.

[196] Waterloo Maple Inc, "*Maple V Programming Guide*," 1998. ASIN 0387945377.

[197] W. H. Press, et al., "*Numerical recipes in C: the art of scientific computing*," Cambridge University Press, 1997. ISBN 0-521-43108-5.

[198] M. J. Schulte, E. E. Swartzlander Jr., "Exact rounding of certain elementary functions," *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, 1993. pp. 138-145.

[199] I. Koren, O. Zinaty, "Evaluating elementary functions in a numerical coprocessor based on rational approximations," *IEEE Transactions on Computers*, Vol. 39, No. 8, August 1990. pp. 1030-1037.

[200] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computers*, Vol. EC-9, September 1960. pp. 227-231.

[201] J. S. Walther, "A unified algorithm for elementary functions," *Proceedings AFIPS 1971 Spring Joint Computer Conference*. 1971. pp. 379-385.

[202] W. E. Fergusson Jr, T. Brightman, "Accurate and monotone approximations of some transcendental functions," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, 1991. pp. 237-244.

[203] J. Cao, B. W. Y. Wei, "High-performance hardware for function generation," *Proceedings of the 13th IEEE Symposium on Computer Arithmetic (ARITH'97)*, Asilomar, CA, March 6th-9th 1997. pp. 184-188.

[204] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, 1991. pp. 232-236.

[205] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software*, Vol. 17, No. 3, September 1991. pp. 410-423.

[206] G. Knittel, "A fast logarithm converter," *Proceedings of the 7th IEEE ASIC Conference*, 1994. pp. 450-453.

[207] W. F. Wong, E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, Vol. 43, No.3, March 1994. pp. 278-294.

[208] M. D. Ercegovac, T. Lang, J-M. Muller, A. Tisserand, "Reciprocal, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Transactions on computers*, Vol. 49, No. 7, July 2000. pp. 628-637.

[209] S. K. Lam, T. Srikanthan, "A linear approximation based hybrid approach for binary logarithmic conversion," *Microprocessors and Microsystems, Elsevier Science B. V.*, No. 26, 2002. pp. 353-361.

[210] D. Defour, F. de Dinechin, J-M. Muller, "A new scheme for table-based evaluation of functions," *Research Report No. 2002-45*, ENS, Lyon, France. November 2002.

FPGA Function evaluation

[211] R. J. Andraka, "Building a high performance bit serial processor in an FPGA," *Proceedings On-chip System Design Conference (design Supercon '96)*, January 1996. pp. 5.1-5.21.

[212] C. H. Dick, "FPGA based systolic array architectures for computing the discrete Fourier transform," *IEEE International Symposium on Circuits and Systems (ISCAS'96)*, 1996. pp. 465-468.

[213] M. Park, S. Choi, S. Kim, J-A. Lee, "A digital sinusoid synthesis based on the postscaled CORDIC," *APCC/OECC'99*, Vol.2, October 1999. pp. 984-951.

[214] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," *Sixth ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Montrey, CA, 22nd-24th February 1998. pp. 191-200.

[215] O. Mencer, M. Morf, "Parallel pipelined CORDICS for reconfigurable computing," 1998. www.doc.ic.ac.uk/~oskar/pubs/cordic.pdf

[216] W. B Ligon III. G. Monn, D. Stanzione, F. Stivers, K. D. Underwood, "Implementation and analysis of numerical components for reconfigurable computing," *Aerospace*, 1999.

[217] C. Dick, F. J. Harris, "Configurable logic for digital communications: some signal processing perspectives," *IEEE Communications Magazine*, August 1999. pp. 107-111.

[218] V. Kantabutra, "High-radix CORDIC for vector rotation with pipelined FPGA implementation," *Proceedings IEEE International Conference on Electronics, Circuits and Systems (ICECS '99)*, 1999. pp. 1131-1134.

[219] N. Boullis, "Designing arithmetic units for adaptive computing with PAM-Blox," *Second Year MIM Internship Report*, ENS-Lyon, France, September 2000.

[220] J. Valls, M. Kuhlmann, K. K. Parhi, "Efficient mapping of CORDIC algorithms on FPGA," *IEEE Workshop on Signal Processing Systems (SiPS 2000)*, Louisiana, USA, October 2000. pp. 336-345.

[221] A. P. Paplinski, N. Bhattacharjee, C. Greif, "Rotating ultrasonic singal vectors with a word-parallel CORDIC processor," *Proceedings of the IEEE EUROMICRO Symposium on Digital System Design: Architectures, Methods and Tools*, Warsaw, Poland, September 2001. pp. 254-261.

[222] O. Mencer, N. Boullis, W. Luk, H. Styles, "Parameterised function evaluation for FPGAs," *Proceedings Field-Programmable Logic and Applications (FPL'01)*, LNCS 2147, Belfast, Northen Ireland, August 2001. pp. 544-554.

[223] M. W. Kharrat, M. Loulou, N. Masmoudi, L. Kamoun, "A new method to implement CORDIC algorithm," *8th International Conference on Electronics, Circuits and Systems (ICECS 2001)*, Malta, 2nd-5th September 2001. pp. 715-718.

[224] J. Pineiro, J. D. Bruguera, J. M. Muller, "FPGA implementation of a faithful polynomial approximation for powering function computation," *Proceedings of the EuroMicro Symposium on Digital Systems Design*, 2001. pp. 262-269.

[225] W. Zhilu, R. Guanghui, Z. Yaqin, "A study on implementing wavelet transform and FFT with FPGA," *Proceedings of the 4th IEEE International Conference on ASIC*, 2001. pp. 486-489.

[226] F. Cardells-Tormo, J. Valls-Coquillat, "Quadrature direct digital frequency synthesisers area-optimised design map for LUT-based FPGAs," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'03)*, Vol. II, 2003. pp. 260-263.

[227] T. Lund, M. Aguirre, A. Torralbu, "Making use of CORDICs and distributed arithmetic to produce a field-programmable fuzzy logic controller in an FPGA," *Proceedings of the 28th Annual IEEE Conference of the Industrial Electronics Society (IECON'02)*, 2002. pp. 3205-3208.

[228] F. de Dinechin, A. Tisserand, "Some improvements of multipartite table methods," *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, 2001. pp. 128-135.

[229] S. Vadlamani, W. Mahmoud, "Comparison of CORDIC algorithm implementations on FPGA families," *Proceedings of the 34th IEEE Southeastern Symposium on System Theory*, March 18th-19th 2002. pp. 192-196.

[230] J. Detrey, F. de Dinechin, "Multipartite tables in Jbits for the evaluation of functions on FPGAs," *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS' 02)*, 2002. p. 154.

[231] N. Sidahao, G. A. Constantinides, D. Y. K. Cheung, "Architectures for function evaluation on FPGAs," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Vol. II, Thailand, May 25th-28th 2002. pp. 804-807.

[232] Y. Yang, C. Wang, M. Ahmad, M. N. S. Swamy, "An FPGA implementation of an on-line radix-4 CORDIC 2-D IDCT core," IEEE, 2002. Vol. IV. pp. 763-766.

[233] S. Ravichandran, V. Asari, "Pre-computation of rotation bits in unidirectional CORDIC for trigonometric and hyperbolic computations," *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)*, 2003.

[234] D-U. Lee, W. Luk, J. Villasenor, P. Y. K. Cheung, "Non-uniform segmentation for hardware function evaluation," *Proceedings Field Programmable Logic and Applications (FPL'03)*, LNCS 2778, Lisbon, Portugal, September 2003. pp. 796-807

[235] C. H. Ho, K. H. Tsoi, H. C. Yeung, Y. M. Lam, K. H. Lee, P. H. W. Leong, R. Ludewig, P. Zipf, A. G. Ortiz, M. Glesner, "Arbitrary function approximations in HDLs with application to the N-body problem," *Proceedings of the 2003 International Conference on Field-Programmable Technology (FPT)*, The University of Tokyo, 15th-17th December, 2003. pp. 84-91.

[236] D-U. Lee, J. Villusenor, P. Y. K. Cheung, "Hierarchical segmentation schemes for function evaluation," *Proceedings of the 2003 International Conference on Field-Programmable Technology (FPT)*, The University of Tokyo, 15th-17th December, 2003. pp. 92-99.

[237] F. Cardells-Tormo, J. Valls-Coquillat, "Area-optimized implementation of quadrature direct digital frequency synthesisers on LUT-based FPGAs," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 50, No. 3, March 2003. pp. 153-138.

Logarithmic Number System ASIC

[238] N. G. Kingsbury, P. J. W. Rayner, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, Vol. 7, No. 2, 1971. pp. 56-58.

[239] E. E. Swartzlander Jr., A. G. Alexopoulos, "The sign/logarithm number system," *IEEE Transactions on Computers*, Vol. C-24, December 1975. pp. 1238-1242.

[240] A. D. Edgar, S. C. Lee, "FOCUS microcomputer number system," *Communications of the ACM*, Vol. 22, No.3, March 1979. pp. 166-177.

[241] T. Kurokawa, J. A. Payne, S. C. Lee, "Error analysis of recursive digital filters implemented with logarithmic number systems," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-28, No.6, December 1980. pp. 706-715.

[242] M. G. Arnold, "*Extending the precision of the sign logarithmic number system*," M. S. Thesis, University of Wyoming, Laramie, WY, July 1982.

[243] E. E. Swartzlander Jr., D. V. S. Chandra, H. T. Nagle Jr., S. A. Starks, "Sign/Logarithm arithmetic for FFT implementation," *IEEE Transactions on Computers*, Vol. C-32, June 1983. pp. 526-534.

[244] G. L. Sicuranza, "On efficient implementations of 2-D digital filters using logarithmic number systems," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-31, No. 4, August 1983. pp. 877-885.

[245] V. P. Shenoy, "Error analysis of LMS adaptive digital filter implemented with logarithmic number system," *Proceedings ICASSP 84*, San Diego, CA. 1984. pp. 30.10.1-30.10.4.

[246] M. L. Frey, F. J. Taylor, "A table reduction technique for logarithmically architected digital filters," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-33, No. 3, June 1985. pp. 718-719.

[247] D. V. S. Chandra, "Accumulation of coefficient roundoff error in fast Fourier transforms implemented with logarithmic number system," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-35, No. 11, November 1987. pp. 1633-1636.

[248] F. J. Taylor, R. Gill, J. Joseph, J. Radke, "A 20 bit logarithmic number system processor," *IEEE Transactions on Computers*, Vol. 37, No. 2, February 1988. pp. 190-199.

[249] T. Stouraitis, F. J. Taylor, "Analysis of logarithmic number system processors," *IEEE Transactions on Circuits and Systems*, Vol. 35, No. 5, May 1988. pp. 519-527.

[250] G. M. Papadourakis, J. Condorodis, "A VLSI design of processing element for reconfigurable systolic architectures based on LNS," *Proceedings of the IEEE International Conference on ASSP*, New York, April1988. pp. 2080-2083.

[251] M. G. Arnold, J. Cowles, T. Bailey, "Improved accuracy in DSP applications," *Proceedings of the 1988 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-88)*, Vol. 3, April 11th-14th 1988. pp. 1714-1717.

[252] H. Henkel, "Improved addition for the logarithmic number system," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, No. 2, February 1989. pp. 301-303.

[253] M. G. Arnold, T. A. Bailey, J. R. Cowles, J. J. Cupal, "Redundant logarithmic arithmetic," *IEEE Transactions on Computers*, Vol. 39, No. 8, August 1990. pp. 1077-1085.

[254] D. M. Lewis, "An architecture for addition and subtraction of long word length numbers in the logarithmic number system," *IEEE Transactions on Computers*, Vol. 39, No. 11, November 1990. pp. 1325-1336.

[255] L. K. Yu, D. M. Lewis, "A 30-b integrated logarithmic number system processor," *IEEE Journal of Solid-state Circuits*, Vol. 26, No. 10, October, 1991. pp. 1433-1440.

[256] M. G. Arnold, T. A. Bailey, J. R. Cowles, "Comments on 'An architecture for addition and subtraction of long word length numbers in the logarithmic number system'," *IEEE Transactions on Computers*, Vol. 41, No. 6, June 1992. pp. 786-788.

[257] M. G. Arnold, T. A. Bailey, J. R. Cowles, M. D. Winkel, "Applying features of IEEE 754 to Sign/logarithmic arithmetic," *IEEE Transactions on Computers*, Vol. 41, No. 8, August 1992. pp. 1040-1050.

[258] D. M. Lewis, "An accurate LNS arithmetic unit using interleaved memory function interpolator," *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, 29th June- 2nd July 1993. pp. 2-9.

[259] S-C. Huang, L-G. Chen, T-H. Chen, "The chip design of a 32-b logarithmic number system," *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1994. pp. 167-170.

[260] J. N. Coleman, "Simplification of table structure in logarithmic arithmetic," *Electronics Letters*, Vol. 31, No.22, 26th October 1995. pp. 1905-1906.

[261] I. Orginos, V. Paliouras, T. Stouraitis, "A novel algorithm for multi-operand logarithmic number system addition and subtraction using polynomial approximation," *IEEE International Symposium on Circuits and Systems*, Seattle, Washington, 30th April-5th May 1995. pp. 1992-1995.

[262] D. M. Lewis, "114 MFLOPS logarithmic number system unit for DSP applications," *IEEE Journal of Solid-state Circuits*, Vol. 30, No. 12, December 1995. pp. 1547-1553.

[263] "Errata in 'Simplification of table structure in logarithmic arithmetic'," *Electronics Letters*, Vol. 32, No.22, 24th October 1996. pp. 2103-2103.

[264] V. Paliouras, T. Stouraitis, "A novel algorithm for accurate logarithmic number system subtraction," *IEEE International Symposium on Circuits and Systems*, Atlanta, Georgia, 12th-15th May1996. pp. 268-271.

[265] D. V. Chandra, "Error analysis of FIR filters implemented with logarithmic arithmetic," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, Vol. 45, No. 6, June 1998. pp. 744-747.

[266] J. R. Sacha, M. J. Irwine, "The logarithmic number system for strength reduction in adaptive filtering," *Proceedings 1998 International Symposium on Low Power Electronics and Design*, 10th-12th August 1998. pp. 256-261.

[267] E. I. Chester, J. N. Coleman, "Development of a high speed 32b real arithmetic core for DSP and graphics applications using logarithmic number system techniques," 1999. http://napier.ncl.ac.uk/HSLA/docs/prep99paper.pdf

[268] J. N. Coleman, E. I. Chester, C. I. Softley, J. Kadlec, "Arithmetic on the European logarithmic microprocessor," *IEEE Transactions on Computers*, Vol. 49, No. 7, July 2000. pp. 702-715.

[269] C. Chen, R-L. Chen, C-H. Yang, "Pipelined computation of very large word-length LNS addition/subtraction with polynomial hardware cost," *IEEE Transactions on Computers*, Vol. 49, No. 7, July 2000. pp. 716-726.

[270] M. G. Arnold, "A pipelined LNS ALU," *Workshop on VLSI*, Orlando, FL, 19th – 20th April 2001. pp. 155-160.

[271] M. G. Arnold, C. Walter, "Unrestricted faithful rounding is good enough for some LNS applications," *Proceedings 15th IEEE International Symposium on Computer Arithmetic*, Vail, Colorado, 11th – 13th June 2001. pp. 237-245.

[272] M. G. Arnold, "Design of a faithful LNS interpolator", *Proceedings of the IEEE EUROMICRO Symposium on Digital Systems Design*, September 2001. pp. 336-345.

[273] T. Stouraitis, V. Paliouras, "Considering the alternatives in low power design," *IEEE Transactions on Circuits and Devices*, July 2001. pp. 23-29.

[274] M. G. Arnold, M. D. Winkel, "A single-multiplier quadratic interpolator for LNS arithmetic," *Proceedings 2001 IEEE International Conference on Computer Design, (ICCD 2001)*, Austin, TX, September 2001. pp. 178-183.

[275] V. Paliouras, T. Stouraitis, "Signal activity and power consumption reduction using the logarithmic number system," *Proceedings of the 2001 IEEE Symposium on Circuits and Systems (ISCAS'01)*, May 6th-9th 2001. Vol. II. pp. 653-656.

[276] V. Paliouras, T. Stouraitis, "Low power properties of the logarithmic number system," *Proceedings 15th IEEE International Symposium on Computer Arithmetic*, Vail, Colorado, 11th – 13th June 2001. pp. 229-236.

[277] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl, A. Hermanek, "The European microprocessor – a QR RLS application," *Proceedings of the 35th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. 1, November 4th-7th 2001. pp. 155-159.

[278] J. N. Coleman, J. Kadlec, "Extended precision logarithmic arithmetic," *Proceedings of the 34th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. 1, October 29th – November 1st 2000. pp. 124-129.

[279] M. I. Youssef, "A new stable second-order section for recursive digital filters realized with logarithmic arithmetic," *Proceedings IEEE MELECON*, Cairo, Egypt, May 7th-9th 2002. pp. 308-314.

[280] V. Paliouras, "Optimization of LNS operations for embedded signal processing applications," *Proceedings IEEE International Symposium on Circuits and Systems*, Scottsdale, AZ, Vol. III, May 2002. pp. 744-747.

[281] Y. Wang, H. M. Lam, C-Y. Tsui, R. S. Cheng, W. H. Mow, "Low complexity OFDM receiver using log-FFT for coded OFDM system," *Proceedings IEEE Symposium on Circuits and Systems (ISCAS'02)*, Vol. III, May 2002. pp. 445-448.

[282] M. G. Arnold, "Improved cotransformation for LNS subtraction," *Proceedings IEEE Symposium on Circuits and Systems (ISCAS'02)*, Vol. II, May 2002. pp. 752-755.

[283] M. G. Arnold, "Asymmetric and compressed logarithmic number systems for a multimedia coprocessor," *Proceedings of the 37th IEEE Asilomar Conference on Signals, Systems and Computers*, November 9th-12th 2003.

[284] M. G. Arnold, "Avoiding oddification to simplify MPEG-1 decoding with LNS," *Proceedings of the 2002 IEEE International Workshop on Multimedia Signal Processing*, December 9th-11th 2002.

[285] M. G. Arnold, "LNS for low-power MPEG decoding," *SPIE Advanced Signal Processing Algorithms Architectures and Implementations XII*, Seattle, July 7th-11th 2002.

[286] E. I. Chester, J. N. Coleman, "Matrix engine for signal processing applications using the logarithmic number system," *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02)*, San Jose, California, July 17th-19th 2002. pp. 716-726.

[287] F. Albu, C. Paleologu, S. Ciochina, "Analysis of LNS implementation of the QRD-LSL algorithms," *CSNDSP*, Stafford, UK. July 2002. pp.364-267.

[288] O. Vainio, "Logarithmic arithmetic for QRD-RLS systolic arrays," 2002. www.rsf.rtu.lv/Latvieshu%20lapa/jaunumi/jaunums_1/pdf_workbox/c02_vainio.pdf

[289] M. G. Arnold, J. Garcia, M. J. Schulte, "The interval logarithmic number system," *Proceedings 16th IEEE Symposium on Computer Arithmetic (ARITH'03)*, June 15th-18th 2003. pp. 253-261.

[290] J. Ruan, M. G. Arnold, "LNS arithmetic for MPEG encoding using a fast DCT," *29th Euromicro Conference, Work-in-progress Session*, September 2003. pp. 47-48.

[291] J. Ruan, M. G. Arnold, "Combined LNS adder/subtractors for DCT hardware," *1st Workshop on Embedded Systems for Real-time Media*, October 3rd-4th 2003. pp. 118-123.

[292] M. G. Arnold, "Geometric-mean interpolation for logarithmic number systems," *2004 IEEE International Conference on Circuits and Systems*, May 2004.

[293] M. G. Arnold, "Iterative methods for logarithmic subtraction," *Proceedings IEEE International Conference on Application Specific Systems, Architectures and Processors (ASAP'03)*, The Hague, June 24th-26th 2003. pp. 315-325.

Logarithmic Number System FPGA

[294] A. Hermanek, R. Matousek, M. Licko, J. Kadlec, "FPGA implementation of logarithmic unit," 2000. http://dce.felk.cvut.cz/cak/Publikace/2000/00Hermanek_FPGAlogUnit.pdf

[295] J. Kadlec, A. Hermanek, C. Coftley, R. Matousek, M. Licko, "32-bit logarithmic ALU for Handel C 2.1 and Celoxica DK1," *presented at Celoxica user Conference*, Stratford, UK. 2nd-3rd April 2001.

[296] F. Albu, J. Kadlec, C. Softley, R. Matousek, A. Hermanek, N. Coleman, A. Fagan, "Implementation of (normalised) RLS lattice on Virtex," *Proceedings 11th International Conference on Field Programmable Logic and Applications (FPL 2001)*, Belfast, NI, UK, August 2001. LNCS 2147. pp. 91-100.

[297] R. Matuosek, M. Licko, A. Hermanek, C. Softley, "Floating-point like arithmetic for FPGA," 2001. (*poster for SIGDA/ACM FPGA 2002 Conference*, p. 253) http://dce.felk.cvut.cz/cak/Publikace/2001/Matousek01_float4FPGA_2039.pdf

[298] F. Albu, J. Kadlec, N. Coleman, A. Fagan, "Pipelined implementations of the a priori error-feedback LSL algorithm using logarithmic arithmetic," *Proceedings ICASSP*, Orlando, Florida, 2002. Vol. III. pp. 2681-2684.

[299] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, N. Coleman, "Logarithmic number system and floating-point arithmetic on FPGA," *Proceedings 12th International Conference on Field Programmable Logic and Applications (FPL 2002)*, Montpellier, France, September 2002. LNCS 2438. pp. 627-636.

[300] Z. Pohl, M. Licko, "Utilization of the HSLA toolbox for the FPGA prototyping," 2002. www.celoxica.com/techlib/files/CEL-W0307171J99-28.pdf

[301] UTIA, M. Tichy, "*The LNS ALU Parameters Datasheet*," www.utia.cas.cz/ZS/, July 26th 2003.

[302] J. Schier, J. Kadlec, "Using logarithmic arithmetic for FPGA implementation of the Givens rotations," *Proceedings Baiona Workshop*, September 2003. www.baionaworkshop.org/proceedings/schier.pdf

[303] J. Detrey, F. de Dinechin, "A VHDL library of LNS operators," *Proceedings of the 37th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. II, November 9th-12th 2003. pp. 2227-2231.

Table makers Dilemma

[304] V. Lefevre, J-M. Muller, "Correctly rounded functions for better arithmetic," *Proceedings of the 34th IEEE Asilomar Conference on Signals, Systems and Computers*, Vol. II, October 29th-November 1st 2000. pp. 875-878.

Prescaling

[305] N. Burgess, "Prescaled maximally-redundant radix-4 SRT divider," *Electronics letters*, Vol. 30, No. 23, 10th November 1994. pp. 1926-1928.

SRT Division

[306] J. Cocke, D. W. Sweeney, "High speed arithmetic in a parallel device," *Technical report*, IBM, February 1957.

[307] J. E. Robertson, "A new class of digital division methods," *IRE Transactions on Electronic Computers*, Vol. EC-7, No. 3, September 1958. pp. 88-92.

[308] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 11, No. 3, 1958. pp. 364-384.