An Abstract Of The Thesis Of

Kevin Djang for the degree of Master of Science in Computer Science presented on May 11, 2000. Title: An Alternative Architecture For Performing Basic Computer Arithmetical Operations.

Redacted for privacy

Abstract approved: ____________________

Paul Cull

The arithmetic portions of almost all modern processor architectures are of very similar design. We use the term "traditional" to describe this design, the primary characteristics of which are native support for integer and floating-point number types and special disjoint instructions and hardware for each supported type. Decades of refinement have endowed this traditional arithmetic architecture with high performance, but also certain inherent limitations.

The highly-specific instruction sets and circuitry that provide optimized performance for supported number types, also make it difficult to synthesize unsupported number types and manipulate them in an efficient manner. This trait also applies when using supported number types for arbitrary ranges greater than those directly implemented by the processor.

In this thesis we present an alternative to the traditional computer arithmetic architecture, designed to address the limitations of the traditional approach while preserving most of its benefits.

Instead of the specific number representation support provided by the instructions, hardware and native data types in a traditional ALU/FPU pair, we define a single data type, the *XLU digit* that forms a base from which other number types may be easily derived, along with a set of instruction primitives from which basic arithmetic operations may be efficiently realized.

Our data type has a signed-digit representation, which allows algorithms for addition, subtraction and multiplication to achieve a high degree of parallelism at the primitive instruction level. The instruction primitives and algorithms are designed to hide or eliminate as much branching as possible, further increasing instruction-level independence.

We provide details of the data type, an overview of the set of instruction primitives, and a discussion of how to use those instruction primitives to perform basic arithmetic algorithms for addition, subtraction and multiplication. We also give examples for three derived number represenations; integer, fixed-point and floating-point numbers.

We believe that our approach of building from a unified base provides flexibity and scalability beyond that of the traditional arithmetic architecture.

Our data type, the XLU digit, and the primitive operations to manipulate it may be implemented with modest amounts of circuitry, and this, together with the highly parallel nature of the entire design means that many XLU circuit blocks can be realized in the same silicon area as one traditional ALU/FPU pair. An ALU or FPU may only work when it has the correct type to work on, whereas we believe any and all XLUs available to the processor can be kept busy almost all of the time, achieving greater utilization of the available silicon.

©Copyright by Kevin Djang
May 11, 2000
All Rights Reserved

An Alternative Architecture For Performing Basic Computer Arithmetical Operations

by

Kevin Djang

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented May 11, 2000
Commencement June 2001

Master of Science thesis of Kevin Djang presented on May 11, 2000.

APPROVED:

Redacted for privacy

Major Professor; representing Computer Science

Redacted for privacy

Chair of Department of Computer Science

Redacted for privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of thesis to any reader upon request.

Redacted for privacy

Kevin Djang, Author

Acknowledgements

Without Dr. Lawrence Crowl there would have been no beginning to this thesis and without Dr. Paul Cull there would have been no ending. Dr. Crowl's ideas form the basis for this thesis and it is his guidance that turned those ideas into a cohesive body of work. Dr. Cull's patience and persistence carried me and the work forward to completion. I offer my thanks to them both.

The company I work for, Rogue Wave Software, has supported me throughout the difficult process of balancing a full-time job with writing a thesis.

I would also like to thank my parents for their love and support, and my wife Rebecca, who figures deeply in all the aspects of my life.

Table of Contents

1 Introduction 1

1.1 Background 1

1.2 Motivation 2

1.3 Outline 4

2 Historical Contexts 7

2.1 Number Representation 7

2.2 Instruction Set Designs 10

2.3 A Note Concerning Floating-Point Support 14

2.4 Parallelism 14

3 Key Points 16

3.1 Representation 16

3.2 Details of the XLU Representation 19

3.3 Algorithms for Basic Arithmetic Operations 21

4 Composite Representations 37

4.1 Fixed-Point Numbers 37

4.2 Floating-Point Numbers 44

Table of Contents (Continued)

5 Hardware Considerations 60

5.1 Small Implementation Footprint 60

5.2 High Speed and Multiple-Instruction Issue 60

5.3 Take Advantage of the Short Carry 61

6 Summary and Evaluation 62

6.1 A Summary of Things Done and Not Done 62

6.2 Questions Raised by the Design 64

6.3 Summary of the XLU Architecture 66

6.4 How to Evaluate the Design 68

7 Conclusion 75

References 77

Appendices 86

Appendix A - Small Glossary of Terms 87

Appendix B - The XLU Simulator 90

List of Figures

Figure		Page
1	Addition Primitives – Sum of two digits	23
2	Multi-digit Addition Example	25
3	A multi-digit Squash “Cascade” Example	27
4	Subtraction-by-negation Example	29
5	Subtraction Example	30
6	Multiplication Primitives – Product of two digits	32
7	3x2 Multiplication Example “By Pencil”	33
8	3x2 Multiplication Example “By Pencil” Symbols	35
9	Partial Product Accumulation	36
10	An example XLU digit Implementation	41
11	The scuh and scul operations	41
12	The scdh and scdl operations	42
13	Multi-digit scale-down example	43
14	Multi-digit fnbc example	50
15	Single-digit fnbc example	51
16	Pre-normalization example	53
17	Testing if $f = 0$ for a two-digit value f.	57
18	Testing if $f = n$ for a two-digit value f where $n \neq 0$.	58

List of Tables

Figure		Page
1	XLU Digit Bit Patterns	20
2	XLU Addition Operation Primitives	23
3	The XLU Squash Operation Primitive	26
4	XLU Subtraction Operation Primitives	27
5	XLU Multiplication Operation Primitives	31
6	XLU Scaling Operation Primitives	40
7	XLU Normalization and Rounding Primitives	49
8	An example XLU floating-point number	52

An Alternative Architecture For Performing Basic Computer Arithmetical Operations

1 Introduction

1.1 Background

The manipulation of numbers lies at the center of every computer processor. The predecessors of electronic, digital computers such as Pascal and Leibniz's mechanical adding machines and Babbage's difference engine were designed specifically to help people perform basic arithmetic operations.

The potential of computing machines as "universal simulators" was recognized early (Babbage, Lovelace, Turing, et al. [Ran82]), and nowadays people regard them as much more than sophisticated adding machines. Nevertheless, the heart of any computer is a device that "knows" how to add, subtract, multiply and divide. (And often one or more of these four things are understood in terms of one or two of the others.)

Since basic arithmetic lies at the center of any computer, how computers perform basic arithmetic operations and the number representations they use when performing them has been an area of continual study and developement for as long as there has been computing machinery.

Over the decades, as computer hardware and software have evolved, the methods through which computers perform arithmetic have converged from many, diverse methods to a few, similar methods. New improvements and innovations occur on a regular basis but the focus of attention has shifted from broad inquiry to narrow refinement.

Despite this narrowing of focus, there are many methods of representing numbers, and many ways of implementing basic arithmetic.

In this thesis, we describe and explore the details of one such alternate implementation.

Our work uses no "new" discoveries or principles. At its center is a number representation first proposed over thirty years ago. What is new is the resulting whole; our overall design is a significant departure from the currently established norms for how a processor "should" do basic arithmetic.

In our alternative design we consider a computer arithemetic architecture based upon a number representation and arithmetic first described in the 1960s [Avi61], coupled with concepts taken from the Reduced Instruction Set movement of the 1980s, [CM90], [PH90] and ideas from the multiple, parallel-instruction execution units that have been incorporated into some computer designs from early mainframes (e.g. the CDC 6600 [BH92]) to present day high-performance microsprocessors (e.g. HP's PA-RISC [AAD+93], Intel's Pentium [AA93] and the Motorola/IBM PowerPC [BAM+93] architectures).

We believe that our ideas, which we collectively refer as "the XLU[1] architecture" provide an interesting and viable alternate to the established norm.

1.2 Motivation

The work we present here was motivated and shaped by the following observations, the questions those observations raised, and the subsequent conclusions we drew from considering those questions.

A survey of the arithmetical portions of current microprocessor architectures DEC Alpha, Motorola/IBM PowerPC, Intel Pentium, Sun SPARC [PH90] reveals that all of them have these traits in common:

1. Hardware to directly support two number representations; integer and floating-point.

[1]Following Knuth's naming convention for TEX, We pronounce "XLU" as *clue*, giving X the χ ("chi") sound.

2. CISC-type instructions embedded in special-pupose hardware to perform the floating-point arithmetic.
3. A set of (usually) simpler instructions, disjoint from the floating-point set, to perform the same operations for integers.
4. Although fixed-point arithmetic is the preferred representation for some situations, it is not supported as a native type (like integer and floating-point). Using fixed-point arithmetic means resorting to software. (The same observation holds true for complex arithmetic, etc.)
5. Double-precision arithmetic is often available, but there is no apparent consideration at the hardware level for handling arbitrary-precision numbers (e.g. if you need double-double-precision, you must resort to software).

These observations raised the following questions:

1. In light of observations 1, 4 and 5, can we design the arithmetic portion of a computer to use a single number representation (early machines got by with only one) yet efficiently support integer, fixed-point, floating-point and multiprecision representations and operations? If so, what characteristics would that number representation need to have?
2. Observation 2 applies even to machines that are otherwise completely RISC-based (e.g. the MIPS processor architecture [Cho89]). Research [Dal89] shows that there are advantages to exposing the sub-operations of floating-point arithmetic to compiler optimization but none of the commercially available architectures we know of do this.
3. Taken together, observations 2 and 3 imply that portions of the computers arithmetic hardware go unused some of the time. Special-purpose floating-point hardware is unused during integer-only calculations, and vice-versa. Some machines can make use of at least some of their

floating-point hardware for integer calculations, but the question raised remains interesting; can we design a machine that uses a maximal amount of its available arithmetical hardware for any/every calculation?

Exploring possible answers to these questions led us to design the arithmetical portion of a hypothetical microprocessor. This design encompasses the following distinct – yet related – attributes.

1. A single number representation that can serve as a base for a wide range of simple or complicated numerical types.
2. A set of algorithms to perform basic arithmetic (addition, subtraction, and multiplication – division is shown to be possible but we do not address it) using this number representation.
3. The arithmetic op-codes for our hypothetical microprocessor. These op-codes, or *primitive operations* may be used to realize the algorithms mentioned above.
4. A software simulation of the op-code set, which was used to test the correctness of the algorithms and provide a base from which a larger simulator could be created.
5. Overall guidelines for how the processor might be organized to best take advantage of the number representation, algorithms and its op-codes.

1.3 Outline

The following sections of this work describe the historical context from and around which our ideas evolved, the key points of the XLU architecture, how composite number representations may be derived from it, a discussion

of hardware considerations for implementing it, a summary and evaluation, concluding remarks, a glossary, and a brief description of the simulator software we wrote to help form and test our ideas as we developed the work.

The history section gives an overview of how the task of doing basic arithmetic has evolved through the preceding decades. It focuses on number representations, instruction set architecture, some aspects of floating-point numbers (the peculiarity of CISC FP instructions, on otherwise purely RISC instruction sets is noted), and some aspects of parallelism within computer systems.

The evolution of these particular topics; number representation, instruction sets and parallelism informed and guided our own design efforts, which are presented in the key points section. Of central importance to our design is the reliance upon a single unifying number representation. We list the requirements that the representation must possess, note the work during the 1960s by Algirdas Avizienis that provided us with a suitable candidate, and describe the details of our *XLU digit* along with the primitive operations used to add, subtract, and multiply it.

Since the XLU architecture provides only one number representation, others deemed necessary must be derived from it. We discuss how this may be done in the composite representations section. Two representations are discussed as examples; fixed-point and floating-point. In doing so, the scalability and high degree of instruction-level independence of the XLU digit and its primitive operations is illustrated.

The hardware considerations section contains remarks and observations about the general requirements for actually implementing our design onto hardware. The XLU architecture specifies only a portion of an overall processor design and is meant to allow as much flexibility as possible to a processor architect, so the comments in this section are mostly guidelines, not specific directives.

The summary and evaluation section summarizes what we present in this work and addresses some of the things that we do not include, and why. The XLU architecture is evaluated in terms of feasibility, expected performance, and positive versus negative characteristics. Concluding remarks, the glossary and the description of the simulator round out the work.

2 Historical Contexts

Early digital computers managed to perform various types of arithmetic with one number representation and modest hardware. Since this is what we hoped to achieve[2], a review of how computer arithmetic has evolved is worthwhile.

The subject of computer arithmetic is very large, so we confine our remarks to a brief overview of the topics especially pertinent to our own lines of inquiry. Our purpose is to set our own work in the context of what has been tried before, rather than present a detailed historical account. With that goal in mind we briefly sketch past practice and thought regarding the evolution of number representations, arithmetic algorithms and the hardware and software that implement them.

2.1 Number Representation

From the earliest mechanical devices that performed arithmetic through the first "real" computers of the 1930s and 1940s, the number representations used by computing machinery seem to have been kept fairly close to the number representation the user would work with (i.e. radix-10). Apart from magnitude, early representations carried little "auxillary" information per number. It was often left up to the user, for instance, to assign and track the radix point or sign of a particular quantity.

The flurry of electro-mechanical and electronic computers designed and/or built during the 1940s prompted investigation of how numbers could "best" be represented inside the machines.

Eckert and Mauchly's ENIAC [GG96] represented numbers as radix-10 integers (or fixed point numbers, if the user tracked the radix-point) and ex-

[2] Albeit, with greater flexibility and performance than those early machines.

pressed negative numbers through a form of 10's-complement. The machine also appears to have had a "double-precision" capabilty:

> The equipment normally handles signed 10-digit numbers expressed in the decimal system. It is, however, so constructed that operations with as many as 20 digits are possible.

ENIAC notwithstanding, the advantages of designing a computer's number representation around radix-2 were understood early on. Von Neumann's "First Draft of a Report on the EDVAC" [vN45] succintly articulates the primary tradeoffs inherent in the choice of a radix:

> It takes more binary digits (hence more operations) to perform an arithmetic operation like addition or subtraction, etc., but using decimal arithmetic results in complex circuitry.

Von Neumann understood the tradeoff between computing run-time and component complexity. The inherent speed advantage the early electronic computers enjoyed over their electro-mechanical contemporaries, and the relatively high price and complexity of their individual components (vacuum tubes) led designers like Von Neuman to seek hardware simplicity first. Binary numbers looked like a good way to achieve this:

> A consistent use of the binary system is likely to simplify the operations of multiplication and division considerably ... Binary arithmetics [*sic*] has a simpler and more one-piece logical structure than any other, particulary than the decimal one.

Around ten years earlier, in 1936, Konrad Zuse must have come to similar conclusions. His Z1 through Z4 series of computers were all based upon binary number representations. Several other early machines were also designed to represent binary number (Atanasoff's machine, the Harvard Mark-I, and

the Manchester Mark-I [Roj97, tables 2 and 3]) and by the 1950s, binary numbers for internal representation were the norm[3] .

Although the radix of choice (binary) was settled early on and with fairly rapid unanimity, general acceptance of the desirability for hardware support of floating-point numbers in addition to integers took more time and argument. This is unsurprising. Early electronic computing machinery had strictly limited hardware resources.

Floating-point arithmetic is convenient for users (like radix-10 numbers) and provides a large dynamic range for a given number of digits, but (again, like radix-10 numbers) it requires more information per number, hence more complex circuitry to implement. Floating-point arithmetic algorithms are more complex than their integer or fixed-point counterparts, which once again requires larger amounts of both time and space resources.

Among the early machines, only Zuse's Z1-Z4 machines supported floating-point as their native number type. Early computer designers only dreamed of having the hardware resources to support two or more native types on the same machine. The available machinery was just too precious.

Floating-point number representations were viewed as important and desirable, but too costly – "luxury" items, given secondary consideration, or second-tier support, if at all. The CDC 6600, for example, provided the instructions to produce either floating-point or integer results, but internally supported only one number representation – a 60-bit format that could be interpreted as either floating-point or integer, depending upon the instruction context [BH92].

As time passed the amount of circuitry that could be crammed into a given amount of surface area increased with a rapidity matched only by the drop in cost for that same amount of circuitry. As space and complexity became less expensive and more available, the constraints that argued against supporting

[3]Von Neumann's report mentions the necessity (for human convenience) of incoporating decimal-binary and binary-decimal conversion hardware into input/output devices. The memory and processor internals, however "use strictly binary procedures." [vN45]

floating-point relaxed, and the argument shifted from whether to include it to how to include it efficiently, or what format to use. In the past decade this last question has been resolved in a majority of the available CPU designs through adoption of the IEEE-754 and IEEE-854 floating-point standards.

2.2 Instruction Set Designs

Arithmetic instructions formed a larger percentage of the instruction set for early computers than for their more powerful descendants. Computers were originally conceived as high-speed calculating engines and getting the machine to perform arithmetic correctly and quickly was the paramount aim. In fact prior to Wilkes et al.'s EDSAC (operational in 1949) and the notion of the "stored program computer", there wasn't much of a concept of instruction set architecture at all [PH90].

Tanenbaum [Tan90] writes:

> The earliest digital computers were extremely simple. They had to be. It was hard enough to get them to work at all. From the ENIAC through the IBM 7094, and on to the CDC 6600, computers had relatively few instructions ...

Hardware rapidly developed greater potential capability per unit of space. The entire capabilities of the machine were no longer required to simply do arithmetic. As a result, designers had the freedom to incorporate more types of "basic" operations into their machines. Examples include floating-point and BCD arithmetic, character handling operations, multi-tasking and/or memory-protection primitives, etc.

Naturally, as instruction sets became increasingly large, multifaceted, and complex, the complexity of their hardware implementation increased. Implementing instruction sets by wiring the instructions directly into the hardware

became more and more difficult, costly, and error prone. All these factors led computer designers to establish a level of abstraction between the actual hardware capabilities of the machine, and the instruction-set interface as seen by the programmer. This abstraction was microcode and microprogramming.

With the advent of the IBM 360 architecture (in 1964) microprogramming took off in a big way. In a microprogrammed architecture, each machine instruction is actually a sequence of smaller, simpler instructions. Individual instructions may perform rather involved, complex tasks, the details of which are spelled out by the simpler, individual microperations of the instructions microprogram.

Two factors encouraged actually creating the sort of complex instructions made possible by microprogramming. The instruction-execution speed of a typical 1960s-1970s processor was quite high relative to memory access time. This situation encouraged minimizing fetches to and from memory. Since each instruction costs a significant amount of time to fetch, designers strove to supply instructions that individually performed a lot of work.

Also, computers were still frequently programed directly through the processor's instruction-set, rather than indirectly through a high-level language and translator. This situation led designers to favor "complex" or "powerful" instructions; i.e. instructions that offer many modes of usage or perform fairly complicated operations (e.g. add the contents of a register with the contents of a memory location pointed at by another register, storing the result to yet another memory location). Such instruction sets seek to provide abstraction away from the complexities of the processor internals while still directly presenting the processor's capabilities to the programmer.

Nearly all the prominent processor architectures of the 60s and 70s were built around relatively "high-level", complex instruction sets of the type that are now characterized as Complex Insruction Set Computers (CISC).

During the late 1970s and early 1980s, semiconducter technology brought ever-increasing gains in both time and space and designers began to question

the premises upon which CISC designs were based. The cost to fetch an instruction from memory dropped (primarily due to caches). This implied that the tradeoff of fewer, slower instructions, each performing a complicated task versus many, faster instructions each performing part of the same complicated task should be re-evaluated.

As software projects expanded in size and scope, high-level languages gained increasing popularity and the quality of machine code they produced approached ever-closer the quality of hand-tooled assembly language.

Also, the complexity of microprogrammed assembly-language instruction sets had become increasingly significant. Microprogramming had been employed as an abstraction between the complexity of the instruction set and its hardwired manifestation in the machine. Microcode itself had grown so complex that proposals appeared for adding a "nanocode" abstraction level between the microcode and the bare hardware.

Several groups of computer designers idenpendently arrived at the same alternate solution. Observing that the complex instructions of the then-popular computer architectures were in fact often under-utilized, they proposed returning to a computer architecture based upon a few, simple instructions, each of which would be heavily utilized. Then, instead of adding extra levels of complexity to the machine's instruction set, they proposed adding extra complexity to the optimization portions of high-level language translators, to optimize sequences of the few, simple instructions. In short, simplify the hardware and deal with complexity in the more malleable medium of software.

The ever-increasing speed and sophistication of both memory and processors insured that although these new machines would have to execute many instructions to mimic the operations performed by one CISC instruction, the time to perform the overall task would be comparable, if not better.

Three designs from the early 1980s; the IBM 801, the Berkeley RISC-1 and the Stanford MIPS all differed from the status quo and form a basis for

the so-called Reduced Instruction Set Computer (RISC) design movement that followed.

RISC designs can vary significantly from one another, but all of them share common hallmarks. Among these are:

1. Simple instructions, designed to execute rapidly.

2. No microcode level – the instruction set runs directly on the hardware.

3. Complex operations are short sequences of the provided simple instructions. The burden of efficiently executing these sequences is placed upon the compiler's optimizer.

4. Simple memory access model – operands are LOADed from memory to registers, processed via an instruction, and the results are STOREd back into memory from registers. The LOAD/STORE instructions are decoupled from other instructions, which in turn only "know" about the registers.

The benefits this sort of design, as implied in the preceeding paragraphs are that the burden of complexity is shifted to the very malleable medium of software (the language translator). This simplifies the hardware, and if the instruction set is carefully designed, allows that hardware to be as versatile as a good language translator will allow.

This is, of course, a tradeoff. Overall complexity is not dimished. It *is* shifted from a medium that is very static (the hardware) to one that is easier to change (the software).

In some ways, RISC computers are a return to earlier designs. Whereas in the early days, simplicity was sought because there was no alternative (due to the physical devices of the day), in recent years, simplicity was sought because it provided the best performance and the most versatility.

In both cases – early computers or latter-day RISC designs – the end result is a small, straightforward, simple architecture relative to the CISC machines developed in the intervening years.

2.3 A Note Concerning Floating-Point Support

It is interesting to note that while RISC design principles have become widely accepted and put into practice during the last decade, not all parts of modern processors are RISC-based.

Some recent processor designs (such as Intel's Pentium or Motorola's Coldfire architecture) are termed "hybrid" architectures, because they combine some RISC practices with some CISC. The motivation in hybrid designs is often to achieve both high performance and high code density, but even processor designs that are RISC throughout the rest of their instruction set and layout look like CISC machines in their support of floating-point.

2.4 Parallelism

The term "parallelism" has progressively encompassed several meanings as computer hardware has evolved.

In most early computers, all operations were performed one bit at time. Parallelism at the level of bitwise operations was an aspiration, but often not a practical reality.

The response times of the earliest electronic computers were extremely rapid, relative to their electromechanical forebears, while at the same time, their circuit complexity was limited by the fragility and size of their components relative to their silicon descendents.

Von Neuman addressed this matter in sections 5.5 and 5.6 of the EDVAC draft report [vN45]. He refers to performing arithmetic on all the bits representing a quantity at once as "telescoping operations", and while asserting the obvious attraction and value of such, offers cogent (for the time) reasons why "accelerating these arithmetical operations does therefore not seem necessary."

As circuit integeration and reliability increased, doing more at once became the order of the day at increasingly higher levels of the hardware design.

Processors began to work on chunks of bits at a time, instead of considering each bit individually. "Increasing the level of parallelism" during this era most often meant widening the number of bits the processor could handle at once – i.e., creating machines with ever larger data-words. The level of sequentalism rose to word-level. The bits of the word were processed in parallel, the words were processed in serial.

However, quite early on some pioneering designers attempted the next logical step of processing several words in parallel. Notable among these is the CDC 6600, capable of performing several instructions in parallel.

At this level of abstraction (machine word-level) the task of providing parallelism may be shared by both hardware and software, and there are combinations that use more or less of both. New specialized fields of study such as instruction scheduling arose.

The logical progression from one processor with multiple function units, exemplified in the early 1960s by the CDC 6000, and currently by nearly all high-end CPU's (e.g. the Intel Pentium, Motorola PowerPC and HP PA-RISC architectures) is to provide several processors connected via efficient communcations channels and all working in concert. Venerable examples of this "true-multiprocessor" design are the ILLIAC IV, the Cray-XMP, the Connection Machine of Thinking Machines Incorporated, and more recently, the Sequent Balance/Symmetry and the Intel Paragon architectures [BH92].

3 Key Points

In this section we present the central design ideas of the XLU architecture. We introduce the number representation that forms a base for the design, and arithmetic algorithms that can efficiently manipulate the number representation.

3.1 Representation

3.1.1 Representation Requirements

The XLU's number representation is of central importance. It must meet the following criteria:

1. It must carry enough information to allow it to serve as a basis upon which various other number representations (integer, floating-point, fixed-point, etc.) may be efficiently realized by the software or firmware utilizing the XLU primitive operations.

2. Efficiently realizing other number representations with multiple instructions implies rapid execution of those instructions. Therefore, our base number representation must lend itself to manipulation in a manner promoting as much instruction-level parallelism as possible. Since we must execute more instructions to perform, say, a floating-point addition than a traditional FPU would, we must perform those instructions faster – i.e., in parallel.

3. The number representation must scale well. That is, the performance of multiples of the XLU machine word size must perform well relative

> to the performance of operations involving only the XLU machine word size.

We believe that a signed-digit number, modified to carry additional information for signalling overflow or special conditions such as $+/-\infty$ or IEEE NaN (Not-a-Number) fulfills all the requirements listed above.

A complete treatment of computer arithmetic using a signed-digit number representation first appeared in Avizienis 1961 IRE paper [Avi61]. In chapter 3 of Computer Arithmetic, Swartzlander [Jr.90a] lists the characteristics which make this representation attractive for our own work:

> By reducing carry propagation to one digit position, signed-digit arithmetic forms the basis for cascadable on-line arithmetic algorithms.

and

> Elimination of carry propagation allows online operations to be overlapped ... arithmetic operations can be overlapped by starting operations as soon as digits become available from previous operations (i.e., it is not necessary to wait until the previous operations have been completed).

3.1.2 Signed-Digit Number Representations

Chapter 2 of Koren's *Computer Arithmetic Algorithms* contains a concise summary of signed-digit number systems. We quote liberally from that source and Avizienis' original paper in this section.

A signed-digit system allows the following digit set:

$$x_i \in \{\overline{(r-1)}, \overline{(r-2)}, \ldots, \bar{1}, 0, 1, \ldots, (r-1)\} \qquad (1)$$

where $\bar{i}$ equals $-i$, and r is the radix of the particular number system. This is a relaxation of the requirement in conventional number systems that members of the digit set all be greater-than or equal-to zero. It provides signed-digit systems with the characteristic of redundant number representations.

As an example: If $r = 10$ and we restrict ourselves to two-digit numbers, there are 19 possibilities ($[\bar{9}, \ldots, 9]$) for each digit of each two-digit number. This means for instance, that $(01) = (1\bar{9}) = 1$, and $(0\bar{2}) = (\bar{1}8) = -2$. Not every number representable in this example has redundant representations – the representation of zero is unique. Koren writes:

> ...adding some redundancy in a number system can be very beneficial. On the other hand, a high level of redundancy might be too costly, since a larger digit set requires a larger number of bits to represent each digit.

The redundancy of the number system as a whole may be reduced by restricting the digit set to a subset of the complete set available. The XLU representation is *maximally redundant*, that is, the complete set of available digits is used, so we will not discuss reduced digit sets here.

As an example, here are two signed-digit representations along with their "normal" decimal equivalent values. Both examples assume the following maximally redundant signed-digit (SD) number system:

$$\begin{array}{rc} \text{radix:} & r = 10 \\ \text{digit set:} & [\bar{9}, \bar{8}, \ldots, \bar{2}, \bar{1}, 0, 1, 2, \ldots, 8, 9] \end{array} \qquad (2)$$

$$\begin{array}{lcl} \text{SD value } \ 1\bar{8}2 & = & (1)10^2 + (-8)10^1 + 2 \\ & = & 100 - 80 + 2 \\ & = & 22 \end{array}$$

$$\begin{array}{lcl} \text{SD value } \ \bar{1}53 & = & (-1)10^2 + (5)10^1 + 3 \\ & = & -100 + 50 + 3 \\ & = & -47 \end{array}$$

3.2 Details of the XLU Representation

XLU's number representation is thus based upon the signed-digit representation first discussed in Avizienis' [Avi61] paper. Henceforth, when we refer to an *XLU digit* we mean an entity having sign, overflow and magnitude bits. A particular XLU digit is defined as having the same number of bits as a particular XLU processor implementation's word size. For instance, if the XLU architecture is implemented on a 32-bit processor, with 32 bits per word, the XLU digit for this machine will have 32 bits.

However many bits are available, an XLU digit must encode the following three pieces of information; a value representing a magnitude, an indication of the sign of that magnitude (positive or negative) and an overflow indicator. The overflow indicator requires one bit. The remainder of the bits in the digit represent the magnitude of the digits value in two's-complement form which provides both sign and magnitude. For a word size of n total bits, the digit will have $n - 1$ bits devoted to the digit's sign and magnitude, and the one remaining 1 bit to the overflow indicator.

Our choice of two's-complement form for the magnitude is somewhat arbitrary. The signed-digit representation properties that we wish to take advantage of are oblivious to the magnitude's encoding, and we could have used whatever we wished. We felt using an integral power of two would simplify our work (for example, in writing a software simulator) and among the common binary representations, two's-complement has "nice" properties and is the default encoding for integers on most modern machines.

The actual position of the overflow bit relative to the magnitude portion is not specified by the XLU design. Magnitude and overflow may be arranged within the digit/word in whatever way the implementor deems most suitable. (For example: The XLU simulator we created during the course of our work places the overflow bit in the low-order position, because this happened to make it easy to manipulate.)

Although the magnitude portion of the digit represents a numerical value in two's complement form, an XLU digit with m bits of magnitude has a range that is one less than a standard, m-bit, two's-complement integer. The range differs because the "extra-negative" pattern, characteristic of two's-complement representation is reserved as a *special pattern* by the XLU definition. For a value I, represented by m bits, the two's-complement range would be:

$$-2^{m-1} \leq I \leq 2^{m-1} - 1 \tag{3}$$

but the XLU digit range is:

$$-2^{m-1} - 1 \leq I \leq 2^{m-1} - 1 \tag{4}$$

The combination of the leading bit of the magnitude (the sign bit), the overflow bit, and whether or not all magnitude bits are equal provides twelve distinct patterns. What these patterns repesent is given in table 1.

Table 1: XLU Digit Bit Patterns

s	magnitude	o	condition of the XLU digit
0	all zeros	0	zero
0	nonzero	0	positive numeric value
0	all ones	0	maximum numeric value
0	all zeros	1	positive overflow
0	nonzero	1	positive overflow
0	all ones	1	positive overflow
1	all zeros	0	NaN (for $n/0$)
1	nonzero	0	negative numeric value
1	all ones	0	minimum numeric value
1	all zeros	1	NaN (for $n +$ overflow)
1	non zero	1	negative overflow
1	all ones	1	negative overflow

In table 1, the s column shows the value of the sign bit, the magnitude column describes the state of the magnitude bits excluding the sign, and the o column shows the value of the overflow bit. The column entitle "condition of the XLU digit" gives the meaning of a particular combination of sign, magnitude and overflow bits. Apart from the "NaN" entries, the conditions should be self-explanatory.

The "NaN" (Not-a-Number) patterns allow the XLU digit to be used as a base for IEEE floating-point arithmetic. Note that there are two distinct patterns and two distinct NaNs. One signals the result of an illegal arithmetic operation (such as division-by-zero) and the other signals the result of a binary operation where at least one of the two operands was in an overflow state. Both types of NaNs are "sticky" – i.e. should the result of an operation produce a NaN, any future operation using that result will also produce NaN as *its* result.

3.3 Algorithms for Basic Arithmetic Operations

3.3.1 Addition

According to Koren, "the original motivation for introducing *SD* numbers was to eliminate carry propagation chains in addition and subtraction" and this property shapes the way the XLU performs addition as well. We reproduce the general algorithm here as Koren [Kor93] [4] describes it.

We wish to perform the operation $X + Y = S$, for a signed-digit number system of radix r, where X, Y, S consist of signed-digits x_i, y_i, s_i:

$$X = (x_{n-1}, \ldots, x_0) \quad Y = (y_{n-1}, \ldots, y_0) \quad S = (s_{n-1}, \ldots, s_0)$$

[4] As noted, the algorithm is first described in Avizieni's 1961 paper. We follow (and quote) from Koren's book here because its notation is more up-to-date.

The addition operation consists of two steps:

1. For each x_i and y_i, compute an interim sum u_i and a carry digit c_i:

$$u_i = x_i + y_i - rc_i \tag{5}$$

where:

$$c_i = \begin{cases} 1 & \text{if } (x_i + y_i) \geq a \\ 0 & \text{if } |x_i + y_i| < a \\ \bar{1} & \text{if } (x_i + y_i) \leq \bar{a} \end{cases} \tag{6}$$

with (for a maximally redundant system):

$$\overline{r-1} = \bar{a} \quad \text{and} \quad a = r - 1 \tag{7}$$

2. For each interim sum u_i and carry digit c_i compute a final sum s_i:

$$s_i = u_i + c_{i-1} \tag{8}$$

By removing the carry propagation chain, this algorithm effectively renders the addition algorithm *totally-parallel*[5] at the digit level. Avizienis writes:

> ...each sum ...digit is the function only of the digits in two adjacent digital positions of the operands. The addition time for signed-digit numbers of any length is equal to the addition time for two digits. [Avi61]

The XLU primitive operations for addition directly implement the various steps of the signed-digit addition algorithm. Table 2 lists the addition primitives and the parts of the signed-digit addition algorithm they perform.

The flow diagram in figure 1 shows how the XLU addition primitives may be employed to compute the sum of two XLU digits. As the flow diagram shows, for the case of two single digit numbers, the sum is immediately

[5]I believe Avizienis originated the use of this term in this context.

Table 2: XLU Addition Operation Primitives

Name	opCode	Function
Addition Intermediate Sum	addi x_i, y_i, z_i	$z_i \leftarrow u_i$, by equation 5
Addition Carry Digit	addc x_i, y_i, z_i	$z_i \leftarrow c_i$, by equation 6
Addition Final Sum	adds u_i, c_{i-1}, z_i	$z_i \leftarrow s_i$, by equation 8

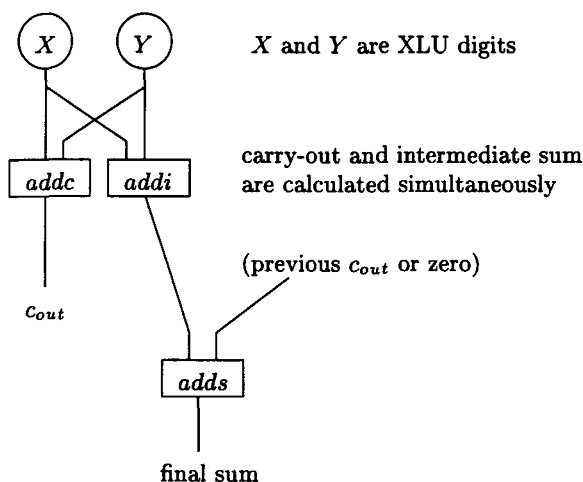


Figure 1: Addition Primitives – Sum of two digits

available after the addi operation. In fact, since the addi operation will set the overflow bit of I_{sum} for cases where an addc would return 1, the addc operation is also only required for multi-digit work. For a single-digit-by-single-digit (i.e. word-by-word) addition, with a single-digit result, the XLU can deliver the result via a single addi operation. "Integer addition" as it is commonly available on conventional processors is therefore also available

under the XLU as a natural consequence of the design, and as a specific subset of the general scheme.

The completely parallel nature of addition at the digit-level provided by the signed-digit representation and choice of addition primitives is illustrated in the following 3-digit number by 3-digit number addition example. In these examples we will use the same digit set and radix used for signed-digit representation examples[6] on page 18.

The sum S of the 3-digit numbers:

$$X = 396 \quad \text{and} \quad Y = \bar{7}87$$

is computed by the following sequence of XLU addition primitive operations. The sequence makes use of temporary storage digits i_2, i_1, i_0 and c_{out}, c_1, c_0. The final result is stored in s_3, s_2, s_1, s_0:

$$\begin{array}{llll}
\texttt{addi} & 6, & 7 & \rightarrow i_0 \\
\texttt{addc} & 6, & 7 & \rightarrow c_0 \\
\texttt{addi} & 9, & 8 & \rightarrow i_1 \\
\texttt{addc} & 9, & 8 & \rightarrow c_1 \\
\texttt{addi} & 3, & \bar{7} & \rightarrow i_2 \\
\texttt{addc} & 3, & \bar{7} & \rightarrow c_{out} \\
\texttt{adds} & i_0, & 0 & \rightarrow s_0 \\
\texttt{adds} & i_1, & c_0 & \rightarrow s_1 \\
\texttt{adds} & i_2, & c_1 & \rightarrow s_2 \\
 & & c_0 & \rightarrow s_3 \quad \text{(leftmost carry-out)}
\end{array}$$

The sequence may be more easily visualized by the equivalent flow diagram shown in figure 2. Notice that regardless of the number of digits involved, all `addi` and `addc` operations may be performed in parallel (assuming adequate processor resources). All `adds` operations may be performed in parallel, pending completion of the `addi` and `addc` operations.

In general, the addition of two N-digit numbers can produce an N+1-digit result. For number schemes that allow dynamic scaling, overflow is handled

[6]The radix-10 example from page 18 is used here rather than a radix-2 SD representation (as the XLU design specifies) because it makes the arithmetic examples easier to read. We will present examples that use "real" XLU digits later.

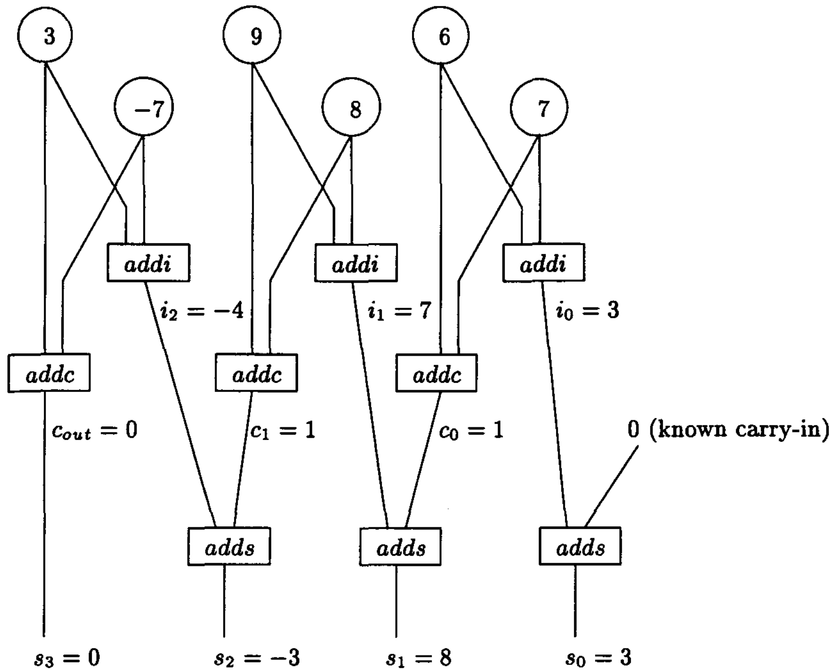


Figure 2: Multi-digit Addition Example

naturally (at least until the resources available on the system give out). For number schemes that require results to fit into a fixed number of digits (if our example's result had been constrained to three, for instance) some decision must be made about which portion of the result will be "lost".

Notice also, that our result of $\bar{3}83$ is in a redundant form. It has the correct magnitude (−217), but in a signed-digit format. In general, signed-digit numbers may require extra processing to convert them to a non-redundant form. In general this may or may not be of concern, and the extra steps may be applied as seen fit.

A slightly more important, and related case, however, is transforming a signed-digit number from a redundant form to a redundant form that requires fewer digits. For example; consider the three digit, radix-10, SD number $\bar{1}98$,

which may also be represented by the single-digit number −2. The XLU **squash** primitive operation (see table 3) provides the ability to perform this conversion on multi-digit XLU values.

Table 3: The XLU Squash Operation Primitive

Name	opCode	Function
Convert multi-digit value	**squash** x_i, y_i, z_i	$z_i \leftarrow (x_i \times radix) + y_i$

The squash primitive takes two XLU digits, assumed to be the neighboring digits of a multi-digit number. It returns the combination of the two as a single-digit value. The result may have its overflow bit set, if the two-digit input is not "squashable". The flow diagram in figure 3 shows how multiple squash primitives may be used in a "cascade" to attempt to compress an n-digit representation down to an $(n - m)$-digit representation. Unlike the highly parallel addition operations previously described, the squash instruction cascade is completely sequential. Each squash primitive must wait for its predecessor's result.

3.3.2 Subtraction

The XLU instruction set performs subtraction through negative addition. The addend takes the part of the subtrahend and the negative of the augend takes the part of the minuend. The XLU primitive operations provided for support of subtraction are listed in table 4.

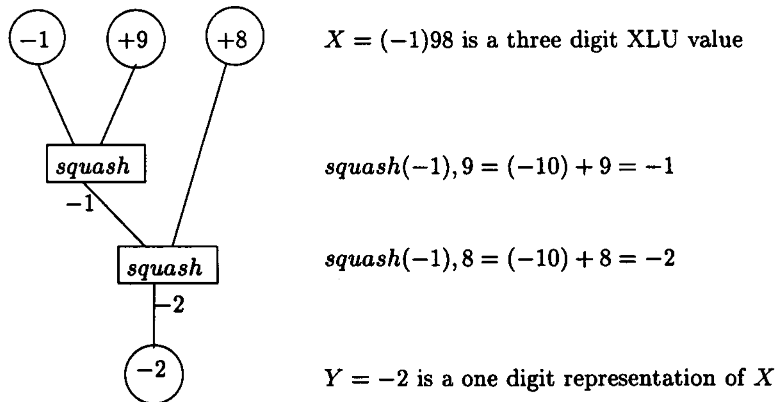


Figure 3: A multi-digit Squash "Cascade" Example

Table 4: XLU Subtraction Operation Primitives

Name	opCode	Function
Negation	neg x_i, y_i	neg $y_i \leftarrow \bar{x}_i$
Subtraction Intermediate Sum	subi x_i, y_i, z_i	$z_i \leftarrow$ addi $x_i, \bar{y}_i, z_i$
Subtraction Carry Digit	subc x_i, y_i, z_i	$z_i \leftarrow$ addc $x_i, \bar{y}_i, z_i$

In early versions of the XLU design, the **neg** operation was provided to allow subtraction. It is a unary operation, taking an XLU digit as input, and returning the negative of that value. For example:

$$\textbf{neg}\ \bar{7} \rightarrow 7$$

Inserting neg operations into the code sequences in the addition section converts them into subtraction algorithms.

Here is the XLU code sequence from the three-digit addition example, with **neg** operations inserted to convert it to subtraction. (Note the inclusion of three additional temporary storage values; n_2, n_1 and n_3):

$$
\begin{array}{llll}
\texttt{neg} & 7 & & \rightarrow n_0 \\
\texttt{neg} & 8 & & \rightarrow n_1 \\
\texttt{neg} & \bar{7} & & \rightarrow n_0 \\
\texttt{addi} & 6, & n_0 & \rightarrow i_0 \\
\texttt{addc} & 6, & n_0 & \rightarrow c_0 \\
\texttt{addi} & 9, & n_1 & \rightarrow i_1 \\
\texttt{addc} & 9, & n_1 & \rightarrow c_1 \\
\texttt{addi} & 3, & n_2 & \rightarrow i_2 \\
\texttt{addc} & 3, & n_2 & \rightarrow c_{out} \\
\texttt{adds} & i_0, & 0 & \rightarrow s_0 \\
\texttt{adds} & i_1, & c_0 & \rightarrow s_1 \\
\texttt{adds} & i_2, & c_1 & \rightarrow s_2 \\
 & & c_0 & \rightarrow s_3 \quad \text{(leftmost carry-out)}
\end{array}
$$

The flow diagram for our modified example is shown in figure 4. Notice that the result, $101\bar{1}$, of this example requires the full four-digit sum a 3x3 digit addition (subtraction) can incur. In this instance, if a maximum of less than three digits was allowable, a decision would have to be made as to what digit to throw away.

The neg operation alone is enough to provide support for subtraction, but the XLU defines two more operations for subtraction (see table 4). They are exact analogues of the addition intermediate sum and addition carry-digit operations, except that they perform the negation of the second argument prior to calculation. In effect, they encapsulate the **neg** portion of the subtraction example given above. No subtraction analogue for the addition final sum operation is needed or provided, since at that point in the computation sequence, the operation is identical for both addition and subtraction.

Figure 5 illustrates our subtraction example again, this time implemented with `subi` and `subc` operations instead of `neg`, `addi`, and `addc`. Encapsulating the negation operation removes a discrete step in the calculation sequence,

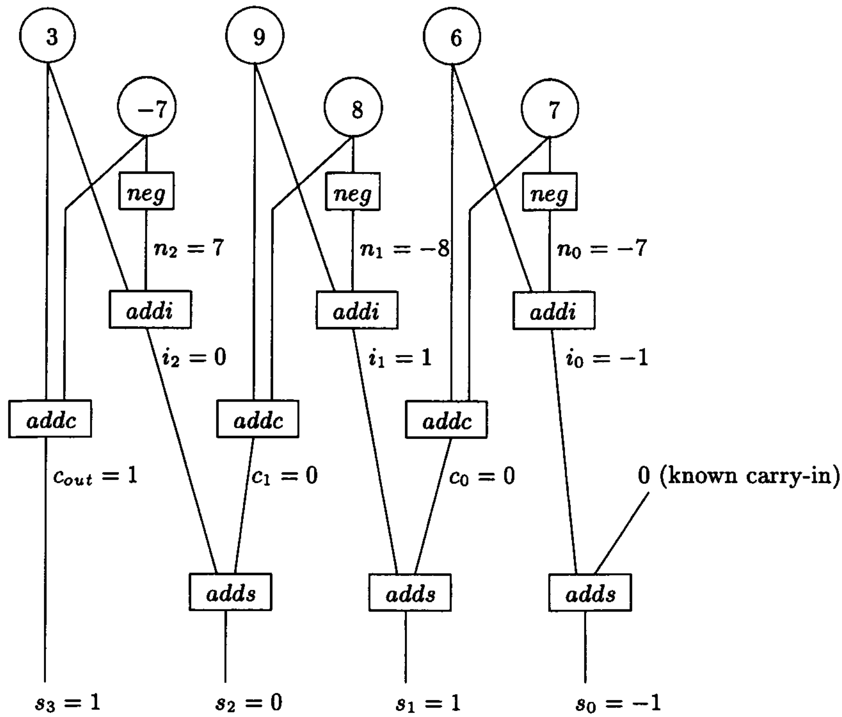


Figure 4: Subtraction-by-negation Example

reducing the required time, instruction count and intermediate storage, all at a minimal complexity cost for the underlying `subi` and `subc` implementation. Nevertheless, in order to provide the compiler with the largest possible set of options for code generation, the `subi`, `subc` and `neg` operations are all provided in the primitive operations set.

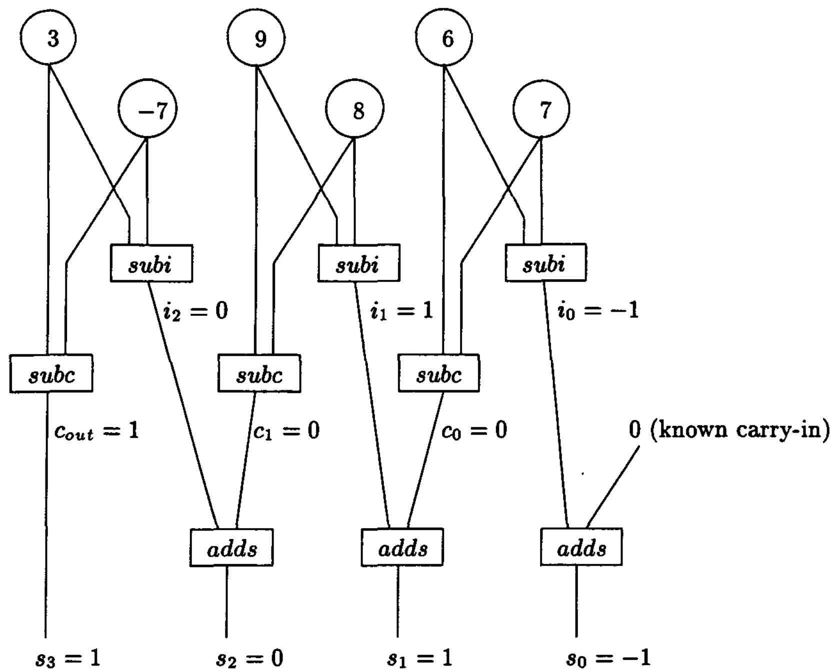


Figure 5: Subtraction Example

3.3.3 Multiplication

Multiplication is the repeated addition of partial products. For the addition portion of this task the parallelization potential of the signed-digit addition algorithm provides a good foundation to build upon. There remains the task of computing the partial products for the digits of the multiplication.

The result of multiplying an XLU digit by another XLU digit may be one or two digits. In order to keep the primitive operations regular (and therefor easier to process efficiently) we perform two "multiply" operations per XLU digit of the multiplicand and multiplier. One operation returns the value of the high-order digit of the product, the other returns the value of the low-order digit of the product. These operations are listed in Table 5.

Table 5: XLU Multiplication Operation Primitives

Name	opCode	Function
multiply (high-digit)	`mulh` x_i, y_i, z_i	$z_i \leftarrow p_1$ of $p_1 p_0 \leftarrow x_i \times y_i$
multiply (low-digit)	`mull` x_i, y_i, z_i	$z_i \leftarrow p_0$ of $p_1 p_0 \leftarrow x_i \times y_i$

Here is an example of how these operations may be used to compute the high and low order results of the product of an digit-by-digit multiplication. For the simplest complete case, a 1-digit-by-1-digit multiplication, this flow diagram shows the sequence of operations: The 2-digit case in figure 6 forms the basic "multiplcation cell" of multiple-digit XLU multiplication arithmetic.

For an n-digit multiplicand and an m-digit multiplier, $n \times m$ multiplication cells are generated; one for each combination of multiplier digit and multiplicand digit. Each multiplication cell is complete independent of any other, and (assuming enough resources) all may be generated in parallel. The results of the multiplication cells are the partial products of the multiplication, and are then added together (using the addition operations described in section 3.3.1) to for the final product.

As an example, consider the product P of a 3-digit multiplicand X and a 2-digit multiplier Y where

$$X = 4\bar{7}8 \quad \text{and} \quad Y = 29$$

Figure 7 shows the partial product generation, intermediate sum accumulation and final product as they might be generated if the intermediate values were generated "by pencil". Two sequences of XLU primitive operations

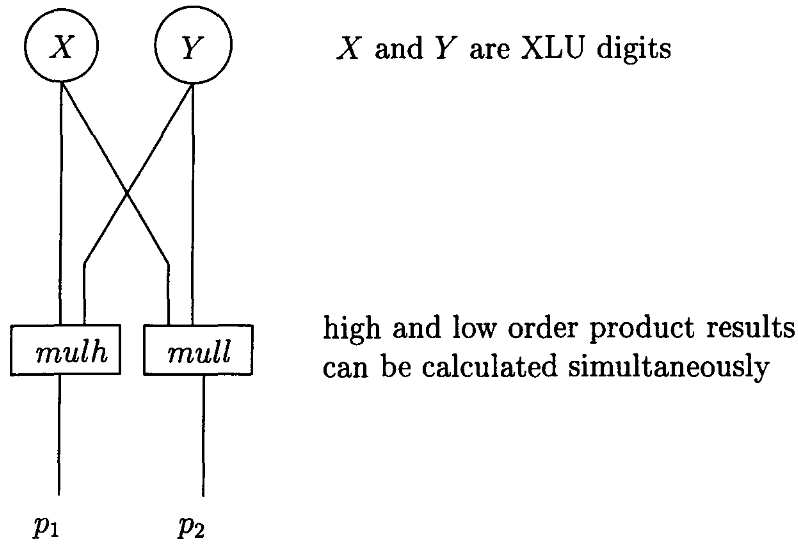


Figure 6: Multiplication Primitives – Product of two digits

comprise this 3x2 multiplication. First, the following sequence generates the necessary partial products.

$$
\begin{array}{llll}
\texttt{mull} & 8, & 9 & \rightarrow \; l_{0,0} \\
\texttt{mulh} & 8, & 9 & \rightarrow \; h_{0,0} \\
\texttt{mull} & \bar{7}, & 9 & \rightarrow \; l_{1,0} \\
\texttt{mulh} & \bar{7}, & 9 & \rightarrow \; h_{1,0} \\
\texttt{mull} & 4, & 9 & \rightarrow \; l_{2,0} \\
\texttt{mulh} & 4, & 9 & \rightarrow \; h_{2,0} \\
\texttt{mull} & 8, & 2 & \rightarrow \; l_{0,1} \\
\texttt{mulh} & 8, & 2 & \rightarrow \; h_{0,1} \\
\texttt{mull} & \bar{7}, & 2 & \rightarrow \; l_{1,1}
\end{array}
$$

$$
\begin{array}{cccccccl}
 & & & 4 & \bar{7} & 8 & (X) \\
 & & \times & & 2 & 9 & (Y) \\
\hline
 & & & & 7 & 2 & \text{(partial products)} \\
 & & & \bar{6} & \bar{3} & & \\
 & & 3 & 6 & & & \\
 & & & 1 & 6 & & \\
 & & \bar{1} & \bar{4} & & & \\
 & 0 & 8 & & & & \\
\hline
 & 0 & 3 & \bar{6} & 7 & 2 & \text{re-arranging terms} \\
 & & \bar{1} & 6 & \bar{3} & & \\
 & & 8 & 1 & 6 & & \\
 & & & \bar{4} & & & \\
\hline
 & 0 & 3 & \bar{6} & 7 & 2 & \text{accumulate} \\
+ & & \bar{1} & 6 & \bar{3} & & \\
\hline
 & 0 & 2 & 0 & 4 & 2 & \\
+ & & 8 & 1 & 6 & & \\
\hline
 & 1 & 0 & 2 & 0 & 2 & \\
+ & & & \bar{4} & & & \\
\hline
 & 1 & 0 & \bar{2} & 0 & 2 &
\end{array}
$$

Figure 7: 3x2 Multiplication Example “By Pencil”

$$
\begin{array}{llcll}
\texttt{mulh} & \bar{7}, & 2 & \rightarrow & h_{1,1} \\
\texttt{mull} & 4, & 2 & \rightarrow & l_{2,1} \\
\texttt{mulh} & 4, & 2 & \rightarrow & h_{2,1}
\end{array}
$$

Next, the following sequence accumulates final product from partial products through repeated application of the totally parallel addition algorithm.

$$
\begin{array}{lllcl}
\texttt{addi} & h_{0,0} & l_{1,0} & \rightarrow & t_0 \\
\texttt{addc} & h_{0,0} & l_{1,0} & \rightarrow & t_1
\end{array}
$$

addi	$h_{1,0}$	$l_{2,0}$	$\rightarrow$	t_2
addc	$h_{1,0}$	$l_{2,0}$	$\rightarrow$	t_3
addi	$h_{2,0}$	$h_{1,1}$	$\rightarrow$	t_4
addc	$h_{2,0}$	$h_{1,1}$	$\rightarrow$	t_5
adds	t_1	t_2	$\rightarrow$	t_6
adds	t_3	t_4	$\rightarrow$	t_7
adds	$h_{2,1}$	t_5	$\rightarrow$	t_8
addi	t_0	$l_{0,1}$	$\rightarrow$	p_1
addc	t_0	$l_{0,1}$	$\rightarrow$	t_{10}
addi	t_6	$h_{0,1}$	$\rightarrow$	t_{11}
addc	t_6	$h_{0,1}$	$\rightarrow$	t_{12}
addi	t_7	$l_{2,1}$	$\rightarrow$	t_{13}
addc	t_7	$l_{2,1}$	$\rightarrow$	t_{14}
adds	t_{10}	t_{11}	$\rightarrow$	t_{15}
adds	t_{13}	t_{12}	$\rightarrow$	t_{16}
addi	t_8	t_{14}	$\rightarrow$	t_{17}
addc	t_8	t_{14}	$\rightarrow$	p_5
addi	t_{15}	$l_{1,1}$	$\rightarrow$	p_2
addc	t_{15}	$l_{1,1}$	$\rightarrow$	t_{18}
addi	t_{16}	t_{18}	$\rightarrow$	p_3
addc	t_{16}	t_{18}	$\rightarrow$	t_{19}
adds	t_{17}	t_{19}	$\rightarrow$	p_4

Regarding the notation used in this sequence: The original digits of X and Y are shown in the primitive operations arguments as their values. Partial products are shown as either an $h_{x,y}$ for the high-order value (generated by mulh) of digits X_x and Y_y, or an $l_{x,y}$ for the low-order value (generated by mull) of digits X_x and Y_y. Intermediate sums and carry digits are notated by temporaries (t_n).

To help clarify the sequence of XLU primitive operations, figure 8 shows a variation of figure 7. The partitial products and intermediate sums are represented by their temporary value labels from the sequence of XLU primitive operations.

Figure 9 shows a flow diagram corresponding to the sequence of XLU primitive operations that accumulate the partial products of the multiplication example into a final sum.

$$
\begin{array}{ccccccc}
 & & & & x_2 & x_1 & x_0 \\
\times & & & & & y_1 & y_0 \\
\hline
 & & & & & h_{0,0} & l_{0,0} \\
 & & & & h_{1,0} & l_{1,0} & \\
 & & & h_{2,0} & l_{2,0} & & \\
 & & & & h_{0,1} & l_{0,1} & \\
 & & & h_{1,1} & l_{1,1} & & \\
 & & h_{2,1} & l_{2,1} & & & \\
 & & & & & & \\
\hline
 & & h_{2,1} & h_{2,0} & h_{1,0} & h_{0,0} & l_{0,0} \\
 & & & h_{1,1} & l_{2,0} & l_{1,0} & \\
 & & & l_{2,1} & h_{0,1} & l_{0,1} & \\
 & & & & l_{1,1} & & \\
\hline
 & & h_{2,1} & h_{2,0} & h_{1,0} & h_{0,0} & p_0 \\
+ & & & h_{1,1} & l_{2,0} & l_{1,0} & \\
\hline
 & & t_8 & t_7 & t_6 & t_0 & p_0 \\
+ & & t_{14} & l_{2,1} & h_{0,1} & l_{0,1} & \\
\hline
 & (p_5) & t_{17} & t_{16} & t_{15} & p_1 & p_0 \\
+ & & & & l_{1,1} & & \\
\hline
 & (p_5) & t_{17} & t_{16} & p_2 & p_1 & p_0 \\
+ & & & t_{18} & & & \\
\hline
 & (p_5) & t_{17} & p_3 & p_2 & p_1 & p_0 \\
+ & & t_{19} & & & & \\
\hline
 & (p_5) & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Figure 8: 3x2 Multiplication Example “By Pencil” Symbols

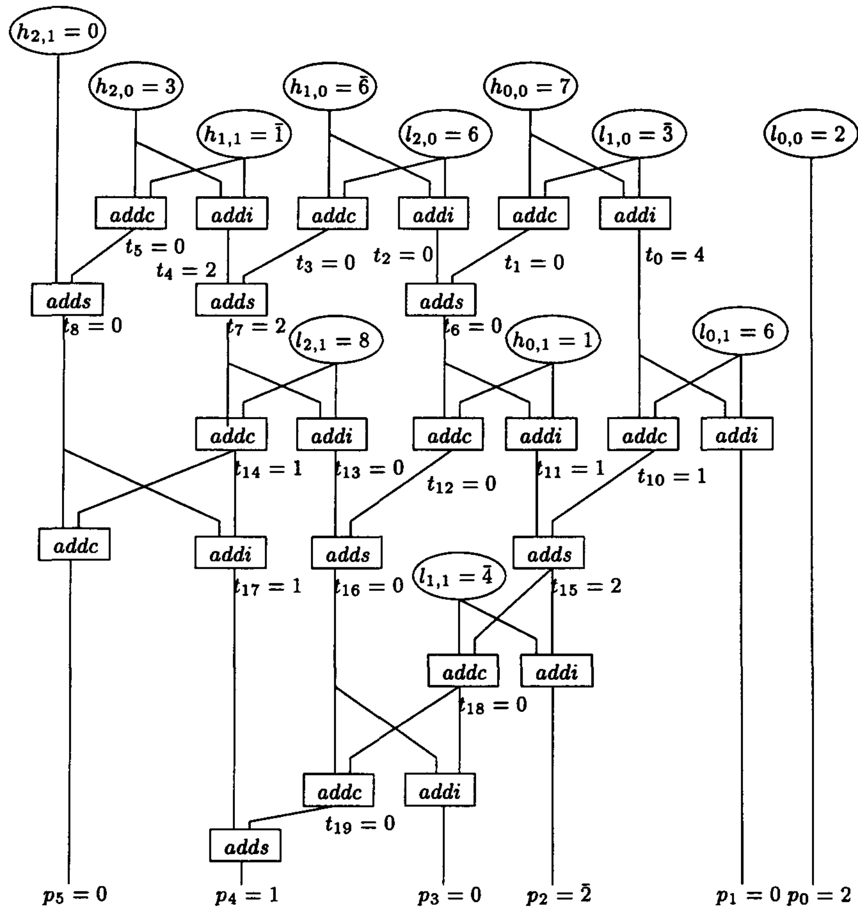


Figure 9: Partial Product Accumulation

4 Composite Representations

This section demonstrates how the XLU number representation and primitive operations may be used as a base from which other number representations may be created. We refer to such representations as *synthesized* representations, and in this sections we describe two examples: fixed-point and floating-point. We also introduce the XLU primitive operations provided specifically to support the needs of synthetic representations such as these.

4.1 Fixed-Point Numbers

We use the definition of fixed-point numbers given in Koren [Kor93]: A fixed-point number X is a sequence of n radix r digits $x_{n-1}x_{n-2}\ldots x_1x_0$ that is partitioned into a fractional part of m digits as well as an integral (integer) part of k digits, with $k+m=n$. Quoting Koren:

> The value of an n-tuple with a radix point between the k most significant digits and the m least significant digits
>
> $$(\underbrace{x_{k-1}x_{k-2}\cdots x_1x_0}_{integral\quad part} \cdot \underbrace{x_{-1}x_{-2}\cdots x_{-m}}_{fractional\quad part})r \tag{9}$$
>
> is
>
> $$\begin{aligned} X &= x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1r + x_0 + x_{-1}r^{-1} + \\ &\quad \cdots + x_{-m}r^{-m} \\ &= \textstyle\sum_{i=-m}^{k-1} x_ir^i \end{aligned} \tag{10}$$
>
> The radix point is not stored in the register but is understood to be in a fixed position between the k most significant digits and the m least significant digits. Therefore, we call such representations

fixed-point representations. The programmer of the digital computer is not necessarily restricted to the use of numbers having the predetermined position of the radix point but can properly scale the operands. As long as the same scaling factor is used for all operands, the add and subtract operations yield the correct results, since $aX \pm aY = a(X \pm Y)$, where a is the scaling factor. However, corrections are required when performing multiplication and division, since $aX \cdot aY = a^2XY$ and $aX/aY = X/Y$.

It should be clear that integers are the subset of fixed-point numbers that have no fractional part. Koren writes "Commonly used positions for the radix are at the rightmost side of the number (i.e., pure integers, $m = 0$) and at the leftmost side of the number (i.e. pure factions, $k = 0$)."

4.1.1 XLU Fixed-Point Number Representation

Substituting "XLU digit" for "digit" in the discussion above shows how naturally Fixed-Point numbers may be represented by the XLU architecture. A fixed-point number may be represented by one or more XLU digits, with the additional constraint that the radix point lies somewhere to the right of the leftmost bit of the leftmost digit, and somewhere to the left of the rightmost bit of the rightmost digit. (Otherwise, the value of digit or digits would be either a pure fraction or a pure integer.)

We leave tracking the position of the radix point to the higher-level (compiler, interpreter, other code-generation) abstraction layers, so the representation itself requires no additional complexity.

The information carried by the XLU digit, and the particular bit-pattern it shows at a given point in time does not change, although the interpretation of what it represents at that point in time may differ, depending upon where the higher-level abstraction chooses to "place" the radix point.

As an example, consider a hypothetical 16-bit implementation of an XLU digit X. Seen as a pure integer, 16 bits gives us a magnitude in the range of $\pm 2^{14} - 1$ and a radix $r = 2^{14}$ with the digit's radix point understood to be adjacent to the rightmost (low-order) magnitude bit.

But we can equally view the 14 bits of the magnitude as having the radix point positioned in the middle (i.e. between the 7th and 8th magnitude bits). The XLU digit now represents both integer and fractional parts, each with representable values according to equation (10) with $k = m = 7$. The radix point could equally be placed between any of the other bits of the XLU digit (with k and m changed appropriately).

4.1.2 Fixed-Point Arithmetic

The arithmetic operations given in section 3.3 apply without modification to an XLU digit sequence whether it represents a pure-integer, pure-fraction, or fixed-point number. However fixed-point arithmetic differs from pure-integer and pure-fraction arithmetic because of the possibility of scaling the operands.

When the scaling involves changing the radix point by an integral number of XLU digits, the change is "transparent" to the digits themselves. Whatever higher-level abstraction is keeping track of the radix position is expected to remember the fact that the radix is now between two different digits.

However, scaling by an integral number of digits is likely to be the exception. For cases where scaling involves changing the radix point's position by less than one digit's amount of bits, XLU primtives are provided. They accept an XLU digit x, and a value s to scale it by.

The scaling primitives work by shifting the magnitude bits of x s positions in the appropriate direction. The sign and overflow bits may play a part in a given scaling primitive's operation, but they are never shifted themselves.

Since both x and s are XLU digits, the range of s (i.e. the number of bit positions to shift) will far exceed the number of magnitude bits in x. For example, an XLU digit implemented in a 16-bit binary word has a total of fourteen magnitude bits, but can specify a maximum shift value of 16383 positions. The scaling primitives handle this discrepency by "maxing-out" at the number of magnitude bits for the particular implementation. Using the same example XLU digit, when an s value exceeds 14, all the magnitude bits are shifted, but "no more" shifting is done, i.e., s effectively equals 14 for values of $[15, 16 \ldots 16383]$. Negative values in s equal a shift value of 0.

Table 6: XLU Scaling Operation Primitives

Name	opCode	Function
Scale up (high digit)	`scuh` x_i, s_i, y_i	$y_i \leftarrow (x_i \times 2^{s_i}) - \text{maximum}(x_i)$
Scale up (low digit)	`scul` x_i, s_i, y_i	$y_i \leftarrow x_i \times 2^{s_i}$
Scale down (high digit)	`scdh` x_i, s_i, y_i	$y_i \leftarrow x_i \div 2^{s_i}$
Scale down (low digit)	`scdl` x_i, s_i, y_i	$y_i \leftarrow \text{maximum}(x_i) - (x_i \times 2^{s_i})$
Scale down intermediate	`scdi` x_i, s_i, y_i	"signed-digit-aware" `scdl`
Scale down carry	`scdc` x_i, s_i, y_i	"signed-digit-aware" `scdh`

The individual scaling primitives are summarized in table 6. Like the multiply instructions, they are designed to be used primarily in pairs, with a given pair returning the high-word and low-word results of a given operation.

In the following paragraphs we explain each of these operations and provide simple examples of their use. For the examples, we use the 16-bit XLU digit implementation mentioned in the discussion above and summarized in figure 10.

In the examples that follow, we ignore the sign (s) and overflow (o) bits as the scaling primitives do[7] and show only the 14 magnitude bits in the diagrams.

[7]This is actually only *mostly* true. The `scdc` and `scdi` pair are sign-aware.

Bit layout within each digit:

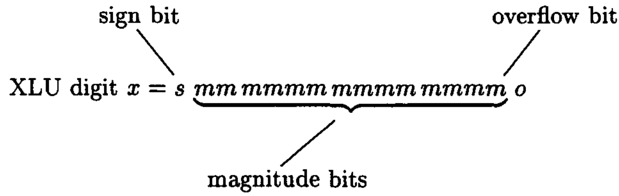


Figure 10: An example XLU digit Implementation

The `scul` primitive operation scales the XLU digit x up by a factor of 2^s (i.e. multiplies x by 2^s) by shifting the magnitude bits of x to the left by s positions. The `scuh` primitive operation returns "the bits shifted off the left end." Figure 11 illustrates both operations.

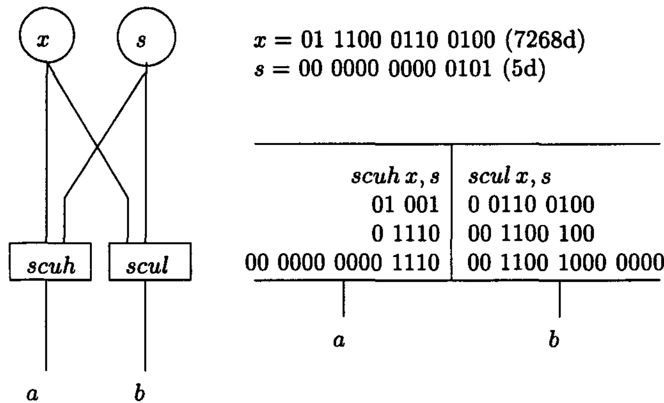


Figure 11: The scuh and scul operations

The `scdh` primitive operation scales the XLU digit x down by a factor of 2^s (i.e. divides x by 2^s) by shifting the magnitude bits of x to the right

by s positions. The `scul` primitive operation returns "the bits shifted off the right end." (Note that the meanings of "high" and "low" for scale-down operations are inverted relative to scale-up operations.) Figure 12 illustrates the `scdh` and `scdl` operations.

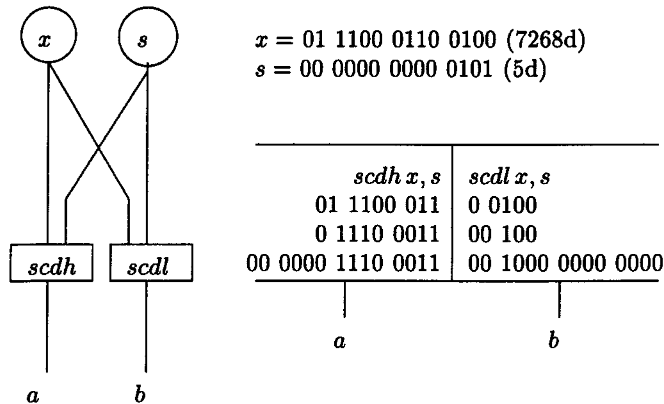


Figure 12: The scdh and scdl operations

The `scdc` and `scdi` primitive operations scale the XLU digit x down by a factor of 2^s bits in the same manner as the `scdh` and `scdl` operations pair. Unlike `scdh/scdl`, the `scdc/scdi` primitives adjust as well as scale x. This additional feature is required for situations where we might wish to add one (for example, a carry-in) after a the scale-down operation(s).

The results from an `scdc/scdi` combination may be added together using the parallel addition algorithm discussed earlier, because the `scdc/scdi` pair assures that their respective results will not overflow. They may be thought of as the scaling equivalents of the `addc/addi` primitive operations.

Figure 13 illustrates a four-bit (i.e. divide-by-16) scale-down of a two-digit XLU value, and also illustrates how the `scdc/scdi` operations pair differs in behavior from the `scdh/scdl` operations pair. In figure 13 the `scdc/scdi`

XLU digits are: x = + 00 0000 0011 1111
and: y = + 11 1111 1111 1111
scale digit: s = + 00 0000 0000 0100

The primitives operation sequence is:

`scdc` $x, s \rightarrow t_3$ = + 00 0000 0000 0100
(`scdh` would give + 00 0000 0000 0011)
`scdi` $x, s \rightarrow t_2$ = − 11 1100 0000 0000
(`scdl` would give + 11 1100 0000 0000)
`scdc` $y, s \rightarrow t_1$ = + 00 0100 0000 0000
(`scdh` would give + 00 0011 1111 1111)
`scdi` $y, s \rightarrow t_0$

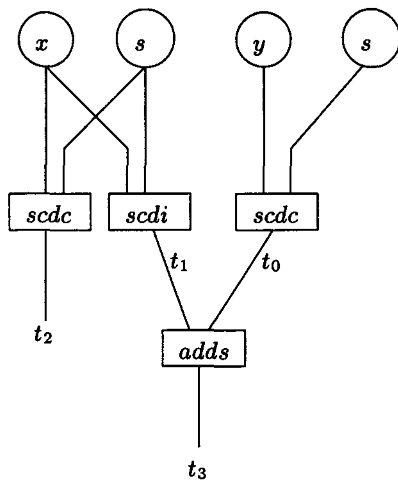


Figure 13: A multi-digit scale-down example (sign bit shown as +/-)

combination leading to the adds result t_3 will tolerate the subsequent addition of a one, while the equivalent `scdh/scdl` combination will not.

4.2 Floating-Point Numbers

Floating-point numbers, like fixed-point numbers, provide values with an integer and fractional portion. However, floating-point numbers accept increased complexity of representation for greater convenience. A floating-point number explicitly tracks the position of the radix, and that position may vary as the particular number is manipulated. The burden of shifting the position of the radix point is taken on by the arithmetic algorthms for floating-point numbers.

Floating-point numbers are commonly represented in the literature by a pair of values (e.g. Knuth [Knu81]). We use Knuth's notation with some slight differences.

Given the base b, excess q, a floating-point number of n digits may be represented by the pair of values:

$$(e, f) = f \times b^{e-q} \tag{11}$$

Where the exponent, e is an integer of a specified range, and the fraction, f is either a signed, pure fraction or a fixed-point number. In this latter case, the fixed-point number is usually a pure-fraction, often with what is known as an "implied one" to the left of the leftmost significant digit (e.g. IEEE-754 floating-point).

4.2.1 Floating-Point Arithmetic

Shifting the burden of explicitly tracking the radix position onto the representation and its operations increases the complexity of both. For the

arithmetic operations we discuss, the additional complexity is almost entirely due to explicitly accounting for the dynamic behavior of the radix-point into the arithmetic operations themselves.

Before discussing methods for implementing floating-point under the XLU architecture, we present general descriptions of the steps required to add, subtract and multiply floating-point numbers. Our descriptions are taken primarily from Knuth [Knu81], whom we will quote liberally here. (Goldberg [Gol91] and Koren [Kor93] also provide good coverage of this material.)

It's important to note at this point that the XLU digit and primitive operations are a sufficent base upon which many different floating-point number systems/semantics may be realized. That is, what we describe here is not the only way to do floating-point arithmetic, nor is it the only sort that the XLU architecture can support. We use Knuth's algorithms here because they are simple, clear, and "classic" among the various descriptions available in the literature.

Also note that Knuth's algorithms devote steps at the beginning and end to "unpacking" and "packing" the floating-point numbers, i.e. separating and recombining the exponent and fractional portions. We leave those steps implicit in our descriptions here.

4.2.2 Floating-Point Addition and Subtraction

Given two n-digit, floating-point numbers $x = (e_x, f_x)$ and $y = (e_y, f_y)$, the sum $s = x + y$ is calculated through the following steps. (These same steps are also used to perform floating-point subtraction by substituting $-y$ for y.)

1. Pre-normalize x and y. Knuth writes:

> A floating-point number (e, f) is *normalized* if the most significant digit of the representation of f is nonzero, so that

$$1/b \leq |f| < 1 \tag{12}$$

or if $f = 0$ and e has its smallest possible value.

That is, we want to scale x and y up by shifting leading 0's off the left-hand, most-significant end of f, while decrementing e, subject to the bounds Knuth notes.

2. Compare e_x and e_y, and if $e_x < e_y$, interchange x and y (i.e., after this step, x is the number with the larger exponent).

3. Set the exponent of the result: $e_s \leftarrow e_x$

4. Compute the difference between the exponents; $d = e_x - e_y$. If $d \geq n+2$, the difference between the exponents is "too large", and the procedure may be terminated in whatever manner is deemed most suitable.

5. Scale right: Shift y's fraction, f_y right by d places; i.e., divide it by $b^{e_x - e_y}$.

6. Add the fractional parts of the numbers: $f_s \rightarrow e_x + e_y$

7. Post-Normalize the result. Post-Normalization is involved enough to merit its own algorithm. See section 4.2.9 for details.

4.2.3 Floating-Point Multiplication

Given two n-digit numbers $x = (e_x, f_x)$ and $y = (e_y, f_y)$, the product $p = x \times y$ is calculated through the following steps.

1. Compute the product's exponent by computing the sum of the multiplicand and multiplier exponents: $e_p \leftarrow e_x + e_y$

2. Compute the product's faction by computing the product of the multiplicand and multiplier fractions: $f_p \leftarrow f_x \times f_y$

3. Post-Normalize the result. (See section 4.2.9 for post-normalization details.)

4.2.4 An XLU Floating-Point Number Representation

As Knuth's definition of a floating-point (equation 11) shows, the heart of an FP representation is two separate pieces of information, exponent and fraction, that together represent one numerical value. From this observation it's easy to see that we may implement each of the two pieces of an FP number with one or more XLU digits, the exponent as a pure integer and the fraction as either a fixed-point value or a pure fraction.

The special patterns (see table 1) defined for the XLU digit allow floating-point representations built from XLU digits to communicate overflow, underflow and other special conditions associated with floating-point arithmetic. In particular, the XLU special patterns provide direct support for the NaN, $+\infty$ and $-\infty$ required by the IEEE-754 and IEEE-854 standards for floating-point arithmetic. (The IEEE floating-point standard's requirements for denormalized arithmetic and specific rounding modes/behavior must be generated by sequences of XLU primitives.)

Obviously, many different implementations of floating-point numbers are possible using XLU digit as a base for exponent and fraction. We describe one here – a simple implementation – for use in the arithmetic examples that follow.

Our example floating-point representation is synthesised from XLU digits as follows:

- An XLU digit composed of a 16-bit binary word, with 14 magnitude bits, and the sign and overflow bits in the leftmost and rightmost bit

positions respectively (i.e. the same example XLU digit implementation used for the scaling primitive examples).

- radix $r = 2^{14}$
- An exponent, e, composed of a single XLU digit.
- A fraction, f, composed of two XLU digits playing the role of a pure-fraction fixed-point number.
- An implied radix point, the left of the leftmost fraction-digit, with no implied one (unlike, for example, IEEE-754) on the other side. The radix point is completely implicit, and as stated in the sections dealing with fixed-point representations.

The radix and exponent range of our example representation are quite large compared to the total number of fraction digits. This gives the representation a relatively large range, but poor precision on that range. This same trait is mitigated in real FP representations by carefully balancing radix, exponent and fraction. We keep the fraction small here to enhance the illustrative capacity of the representation.

4.2.5 Scaling, Normalization and Rounding

Pre-Normalizing the fractional portion of a floating-point number involves scaling it up or down. The various XLU scaling primitive operations are used to actually scale the number, and the XLU primitive operation `fnbc` provides the amount needed to actually scale. Table 7 contains a summary of `fnbc`'s behavior.

The `fnbc` operation takes two arguments; x_{i+1} and x_i. Conceptually the two arguments represent the left-hand and right-hand XLU digits of a two-or-

Table 7: XLU Normalization and Rounding Primitives

Name	opCode	Function
Find Normalization Bit Count	fnbc x_{i+1}, x_i, z	$z \leftarrow \begin{cases} \texttt{fnbc}\ N, x_i & \text{if } x_{i+1} = N \\ 0 & \text{if } x_{i+1} < N \\ 0 & \text{if } x_{i+1} = 0 \end{cases}$

more-digit value. Fnbc returns the number of zeros to the left of the leftmost 1-bit in x_i's magnitude, subject to x_{i+1}'s value.

If $0 \leq x_{i+1} < N$, fnbc returns 0 without evaluating x_i. This signals that there are one or more "1" bits to the left of x_i. If x_{i+1} exactly equals the number of magnitude bits in the XLU digit implementation (N), x_i is evaluated, because $x_{i+1} = N$ implies every magnitude bit to the left of x_i is a zero.

This behavior allows a sequence of fnbc and adds (or addo) operations to "chain" the number of zeros for a multi-digit value. Figure 14 shows a flow diagram of a four-digit fnbc chain. As with multi-digit squash operation sequences, each sucessive fnbc primitve relies on its predecessor's result. Unlike the squash chain, the fnbc chain includes addition operations and as the length of the fnbc chain increases, some parallelism in performing the additions can be realized.

For single-digit values, x is set to N. Figure 15 shows the flow diagram for fnbc with a single digit, along with examples of its use. (The examples use the XLU digit implementation described in figure 10.)

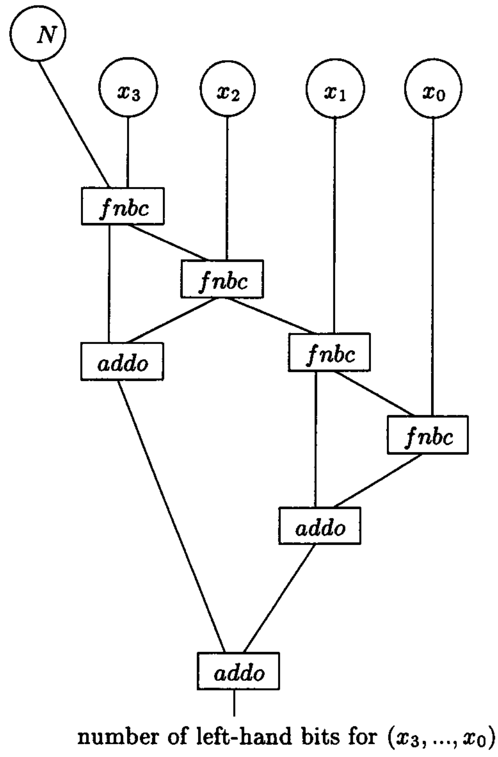
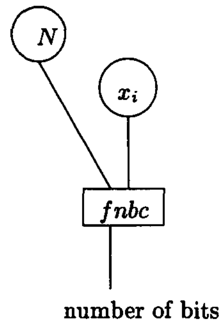


Figure 14: Multi-digit fnbc example



Examples:
fnbc 00 0000 0000 1110 (14d), 11 1111 1111 1111 (16383d)
→ 00 0000 0000 0000 (0d)
fnbc 00 0000 0000 1110 (14d), 00 0000 0010 1001 (41d)
→ 00 0000 0000 1000 (8d)

Figure 15: Single-digit fnbc example

4.2.6 An XLU Floating-Point Example Number

Table 8 illustrates the components and values for an example floating-point number created using our example floating-point representation definition. Notice that the fractional portion consists of two XLU digits; f_{x_1} and f_{x_0}.

4.2.7 XLU Floating-Point Addition and Subtraction

Addition or subtraction of two floating-point XLU numbers is broken down into suboperations. The suboperations implement, through sequences of XLU primitves, the steps in Knuth's floating-point addition/subtraction algorithm (detailed previously).

Table 8: An example XLU floating-point number

$$X = (e_x, f_{(x_1,x_0)})$$

where:

$$e_x = 0\ 00\ 0000\ 0000\ 0011\ 0$$

and:

$$f_{x_1} = 0\ 00\ 0110\ 1010\ 1100\ 0$$

and:

$$f_{x_0} = 0\ 00\ 0110\ 1100\ 1001\ 0$$

Addition of exponent and fractional portions of the addend and augend are implemented using the basic XLU addition sequences detailed earlier for pure-integer, pure-fraction, and fixed-point XLU numbers.

Pre-normalization of the addend and augend is computed via a combination of `fnbc`, scaling and addition primitives. Figure 16 shows a flow-diagram for our example XLU number. In this figure, notice that no `scuh` operation is required for f_{x1} because the behavior of the `fnbc` operation guarantees that no ones will be shifted into the result of `scuh`.

The comparision of the two exponents (step 2, in Knuth's algorithm) requires some explanation. No explicit XLU primitive exists for the comparison of two XLU values.

We assume comparison operations for word-sized operands will exist as part of the "other" portion of whatever instruction-set within which an XLU architecture is realized. The XLU architecture requires only that two operations exist; one to distinguish and branch on whether a value is > 0, the other to do the same if a value is $= 0$.

As the previous paragraph implies, comparing two XLU values depends on first computing the difference between the two XLU values, then using the "other" comparison operation to branch. The comparison is one of the few places within an XLU sequence where branching is not avoided.

Given $X = (e_x, f_x)$ with

$$e_x = \begin{array}{llllll} \texttt{s} & \texttt{mm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{v} \\ \texttt{0} & \texttt{00} & \texttt{0000} & \texttt{0000} & \texttt{0011} & \texttt{0} \end{array}$$

$$f_{(x1,x0)} = \begin{array}{llllllllllll} \texttt{s} & \texttt{mm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{v} \mid & \texttt{s} & \texttt{mm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{mmmm} & \texttt{v} \\ \texttt{0} & \texttt{00} & \texttt{0110} & \texttt{1010} & \texttt{1100} & \texttt{0} \mid & \texttt{0} & \texttt{00} & \texttt{0110} & \texttt{1100} & \texttt{1001} & \texttt{0} \end{array}$$

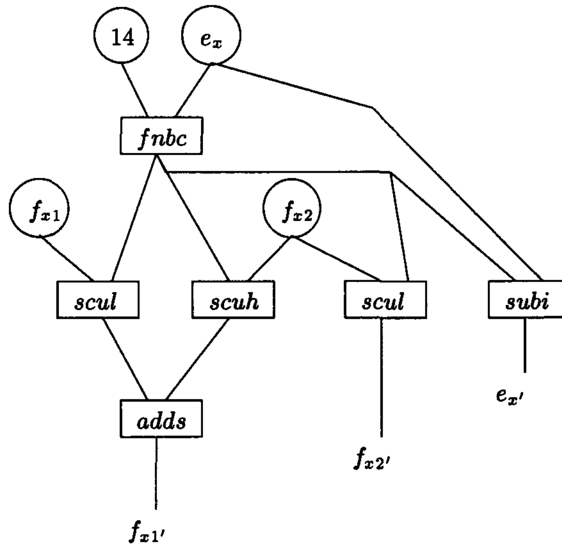


Figure 16: Pre-normalization example

This doesn't address how the "other" comparison operations (which are not required to understand multi-digit XLU numbers) deal with difference values greater than one digit. For the general case, this is future work. For the specific case of finding which exponent is larger, once we compute the difference, we only need to know the leading digit's (leading word, for the "other" comparison operation) relation to zero. We may still need to post-

normalize and round the results. These topics are covered in sections 4.2.9 and 4.2.10.

4.2.8 XLU Floating-Point Multiplication

Compared with floating-point addition, floating-point multiplication is straightforward. Implementing the first two steps of Knuth's previously-detailed algorithm through XLU primitives involves generating an integer addition sequence for the exponents e_x and e_y and a fixed-point multiplication sequence for the fractions $f_{(x_1,x_0)}$ and $f_{(y_1,y_0)}$.

The multiplication sequence may be quite involved, but in addition to the instruction-level parallelism inherent in XLU primitive operations sequences, note that the addition of the exponents and the multiplication of the fractions may also proceed independently of each other.

As with floating-point addition, the end-result of these operations need to undergo post-normalization and rounding. See sections 4.2.9 and 4.2.10 for details.

4.2.9 XLU Post-Normalization

At the completion of either a floating-point addition, subtraction or multiplication sequence, the computed result may not be normalized, according to the definition given in equation 11. This may be acceptable in some instances, but more often the semantics of a particular floating-point type implementation require that the result must be post-normalized.

The goal of post-normalization is the same as pre-normalization; to adjust the exponent and fractional parts of a floating-point value so that together they fit within the definition given in equation 11.

In the following steps we paraphrase Knuth's normalization algorithm [Knu81]. (The algorithm assumes that $|f| < b$, where, as we recall, b is the number base.)

1. Test the result's fraction f: $|f| \geq 1$ indicates "fraction overflow", and a need to scale the result down (i.e. to the right). $|f| = 0$ indicates that the exponent should be set to its lowest possible value, after which, no further work is necessary; the normalization process is complete.

2. Test for normaliztion: For any other value, f may or may not be normalized. Testing whether $|f| \geq 1/b$ determines this. If the number *is* normalized, proceed to rounding, otherwise, scale the result up (i.e. to the left) and try again.

3. Scaling Up (left): Shift f to the left by one position and decrease e by 1, then test for normalization again.

4. Scaling Down (right): Shift f to the right by one position and increase e by 1, then proceed to rounding.

5. Rounding:

   For a fraction f of at most n places, Knuth writes "We take this to mean that f is changed to the nearest multiple of b^{-n}." There are various methods for rounding a value of f greater than n places back down to n places. Some of these may result in $|f| = 1$, which is not allowable in Knuth's algorthim. If this happens, return to the scaling down step and proceed from there.

6. Check e: Either the original result, or prior operations of the normalization process may have resulted in an exponent underflow or overflow condition (e is either smaller or larger than its allowed range). Such cases indicate that the result computed can not be expressed within the system, and appropriate actions must be taken on a per-implementation

basis. If e is safely within tolerance, then the normalization process is successfully complete.

As Knuth's normalization algorithm shows, implementing normalization for floating-point numbers based upon XLU digits is largely a matter of providing the primitive operation sequences to scale the fraction up or down, increment or decrement the exponent, and test the value of the fraction at the appropriate point.

We have already shown how the scaling primitives may be used to construct scaling operations of arbitrary scope, and the exponent increment and decrement operations are simple addition or subtractions upon a signed digit (i.e. an unadorned XLU digit). However, the test to determine if f is normalized, requires further discussion.

As in the case for the comparison operations required in XLU implementations of floating-point addition and subtraction, methods for testing the value of the fraction f are not specified within the XLU primitive operation set. We assume comparison operations for word-sized operands exist as part of the "rest" (i.e. non-XLU portion) of the processor's instruction set, and as previously mentioned, those comparison operations are not assumed to deal with or "understand" multi-digit XLU numbers.

Providing general-case methods for comparisons of multi-digit XLU numbers currently stands as future work, however normalization demands that we supply some specific schemes for comparison of the fraction, f of a floating-point number, which may be multi-digit in its composition.

For specific cases where f is compared relative to a number (i.e. 1 or b) in a strictly greater-than or less than manner, it is enough to test against the most-significant (leftmost) digit of f, since the value of subsequent, lesser-significant digits of f will not change the comparison.

For specific cases where the value of f must be compared equal to a value (e.g. $f = 0$), and we may not assume comparsion operations beyond those provided by the "rest" of the ISA, we resort to a "comparison chain", similar in concept to the scaling chain seqeuences of XLU primitive.

The particular structure required to test if $f = 0$ for our example XLU floating-point number (table 8) representation is given in figure 17. This structure only works for comparisons to 0. It will not work for, say $f = 1$.

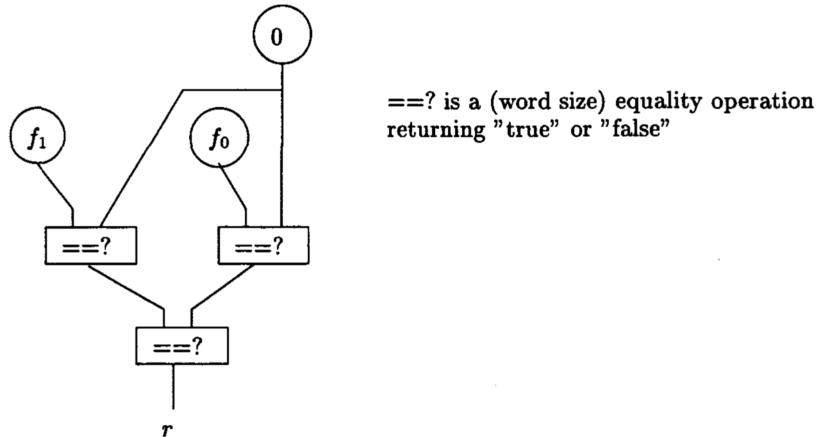


Figure 17: Testing if $f = 0$ for a two-digit value f.

A similar, derivative structure to test if $f = n$, where $n \neq 0$ is shown in figure 18. Obviously, this structure could be used to test if $f = 0$ as well, and for the simple number representation shown, there's no difference.

This structure is very specific to our example floating-point number (table 8) representation. For example, it "knows" the position of the radix point, and relies upon that information. This brings up the fundamental difference between figures 17 and 18. The former is specific to 0, but makes no assumption about radix point. The latter is not specific to zero, but makes assumptions about radix point position.

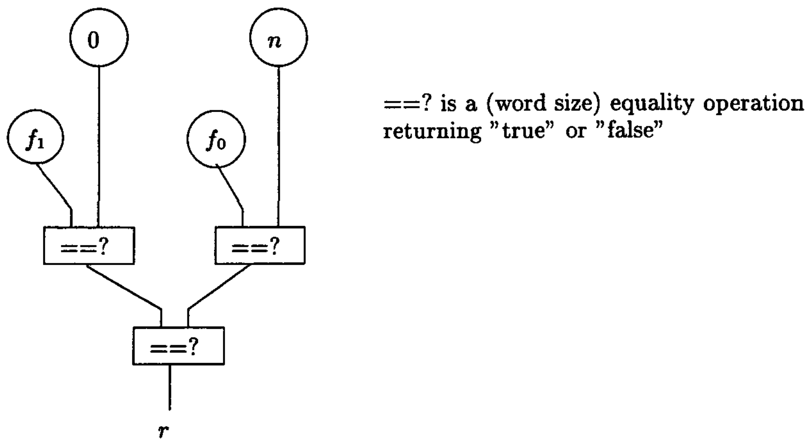


Figure 18: Testing if $f = n$ for a two-digit value f where $n \neq 0$.

Both structures are adequate but aesthetically unsatisfactory, and computationally expensive. While both structures scale, both will form an inverted binary tree, the time cost of which (i.e. depth) is $\log_n$, where n is the number of digits in the XLU representation of f. More over, the zero-comparision structure is general for the XLU number, but compares only to zero, while the n-comparison structure is specific to the XLU number representation. And both representations assume that one of the two items being compared is a single word (i.e. one XLU digit) in size, whether 0 or n in value. Clearly, good strutures for comparison/test operations are a primary candidate for any future work.

4.2.10 Rounding Considerations

The primitive operations for rounding an XLU digit or string of XLU digits representing a number have not been finalized. Specifics of this topic

thus lie outside the scope of this particular thesis. Nevertheless, some general observations may be made.

Whatever form of rounding is required and specified, for example round-towards-infinity or round-to-nearest, the approach we have experimented with is to provide XLU primitive operations (one per desired rounding mode) that will perform the operation for a single XLU digit, but may also serve as "links" in a sequence of such operations so that the same function may be performed on a string of XLU digits. We've demonstrated this approach in the squash and fnbc primitive ops (tables 3 and 7).

Our hypothetical rounding operations could be used to form operation sequences in the same way as we use squash and fnbc. We believe this will work, and it has the benefit of being simple to scale up – that is, the structure produced by using it for multi-digit XLU representations is something we know and have worked with. However, like the squash and fnbc primtive ops, the structure is not highly parallel, and subject to linear time complexity. Thus, future work includes determining if better performance is possible within the XLU framework, and finding a way to provide it if it does. (This might also have implications for improved methods for handling the squash and fnbc operations.)

5 Hardware Considerations

The XLU architecture's hardware is left largely unspecified. Should the XLU ever be realized as an actual device, designers are left with a large amount of lattitude to realize the processor. Nevertheless, the XLU number representation and arithmetic algorithms presuppose certain characteristics and traits about the hardware implementation.

Obviously, the instruction set must implement the base operations required to support the XLU arithmetic algorithms. Beyond that, the following characteristics – perhaps *design principles* is a better term – are expected to be embodied by the hardware.

5.1 Small Implementation Footprint

Whatever the complexity of the overall processor, the XLU portion is expected to be small and compact. We believe that the single number representation and small, simple instruction set may be implemented in a relatively modest number of transistors and on a small amount of surface area.

5.2 High Speed and Multiple-Instruction Issue

In the use of a single number representation, and small, simple instructions we are making the tradeoff pioneered by the RISC processors of the late 1980s: The single number representation and simple instruction set give us flexibility, scalability and uniformity, but we require several instructions to perform arithmetical tasks that on conventional processors require one, or a fraction of one instruction. Thus, to provide the same level of performance as conventional processors, we must execute more instructions per unit time.

The same factors that require this throughput (single number representation, and simple instructions) also provide us with the means of acheiving the goal.

We assume the clock rate of the processor will be "fast enough", and that the cache and register set will be able to keep the XLU portion of the processor supplied with instructions.

We also assume that there will be more than one XLU processing unit per overall processor. If the compactness rule above is followed, there should be room for several XLUs in the space that would be spent on one conventional ALU and FPU pair.

5.3 Take Advantage of the Short Carry

The number representation and algorithms are designed to maximize independence of the primitive operations in a given algorithm's instruction stream, so there should be sufficient fine-grain parallelism to keep any and all available XLUs busy.

The use of Avizienis' algorithms for addition and in the sub-steps of multiplication ensures that carry propagation – a major factor in creating interdependence between suboperations of these algorithms – is minimized. The primitive operations were chosen to limit or hide branching as much as possible. This mitigates a second major factor in inhibiting instruction-level parallelism.

In summary, a processor implementing XLU must provide sufficient speed, and sufficient numbers of XLUs to keep the instruction throughput high. If this is the case, we believe the arithmetical performance of the processor will not suffer relative to conventional designs, and the processor will additionally enjoy the advantages of good scaling, extensibility and flexibility inherent in the XLU design.

6 Summary and Evaluation

6.1 A Summary of Things Done and Not Done

We began this thesis by reviewing how the portions of computer architecture dealing with basic arithmetic have evolved to one dominant approach; that of direct support for integer and floating-point data types, with distinct, specialized instructions and hardware for each type. This approach is presently employed almost universally in new processor design.

We note that while this is true of the arithmetic portion of the architecture, it is not generally true for other portions of processor. The shift from CISC to RISC designs brings this dichotomy into a sharp light; while RISC-based processor Instruction sets generally stand in strong contrast to CISC-based ISAs, the integer arithmetic and (more particularly) floating-point arithmetic portions of such designs differ little from those on CISC processors.

Processor and Instruction-set design in general has a history of inovation, but in the particular sphere of arithmetic, it has focused on refinement rather than diversity.

The premise for the work following these observations is that alternate data types and instruction sets for computer architecture are possible, interesting, and realizable. In support of that premise, we've developed and presented here a data type (the XLU digit) and set of primitive arithmetic operations (the XLU primitive ops) that we believe provide a suitable basis for exploring one alternative to the current "traditional" computer arithmetic architecture.

Our work does not infer a particular implementation, and we do not here address many obvious (and interesting) questions raised by the basic XLU design.

For the work done in this thesis, we remained removed from the physical realities of implementing a processor, and we have not done much analysis to estimate the arithmetical performance (in time and space) of a processor implementing the XLU digit and primitive operations.

We implemented a simple simulator[8] in the C programming language. The simulator works with a 16-bit XLU digit as its data type, and allowed us to check the feasibility of XLU primitive operations and explore various primitive operations sequences. The simulator is not advanced enough to provide metrics for evaluation of the XLU design. Extending it to do so, and to accept alternative XLU primitive operations sets for comparison lies in the realm of future work.

Our goal was to provide a framework with a single data type and a very flexible basic set of instructions upon which further exploratory work might be done. Performance and implementation are future work, and as such are treated in only a speculative fashion here.

The topic of division has not been addressed. It is a basic arithmetic operation and as such obviously deserves inclusion in our work.

In his original article on signed-digit arithmetic Avizienis stated that "Signed-digit division is performed as a sequence of additions or subtractions and left shifts" and recommends "the Robertson division method" [Rob58] as "most readily applicable to signed-digit representations. This is because the Robertson method requires an estimate of quotient magnitude which the signed-digit can provide with "limited uncertainty" [Avi61].

Knuth gives the "classical" algorithm for digit-by-digit division in the his 2nd volume [Knu81] and this algorithm is further explained in Brinch Hansen's 1994 paper [Han94]. Studying the work done by Avizienis, Knuth, Hansen, et al., and implementing the basic algorithm given by Knuth for our XLU digit representation convinced us of the feasibility of implementing division, but also of its complexity.

[8]See Appendix B for a brief description

Dividing two numbers is conceptually not much more difficult than multiplication, but it involves a multitude of details which complicate it. Doerfler [Doe93] in his book on the art of calculating writes that "Division reveals difficulties in simplification". These inherent "difficulties" along with the fact that we have no new insights to share beyond those expounded by the authors noted above placed division (along with extending the simulator, performance metrics, etc.) into the future work category.

6.2 Questions Raised by the Design

As the previous section notes, generating metrics for the actual time and space requirements of XLU impelementation schemes lies beyond the scope of the present work. Nevertheless, by drawing upon inferences from prior work and current industry examples we may make inferences regarding such matters.

6.2.1 Is the Design Realizable?

We believe that the XLU type and primitive operations are easily realized with present technology, but have not explored this in detail. Our belief is based on the similarity of the XLU primitive ops, in both design and computation complexity, to the arithmetic operations of existing processor implementations, e.g. the MIPS R3000 [PH90].

The XLU digit lies somewhere between an integer and a floating-point value, in terms of time/space complexity for a single instance. We assume this would translated into a complexity somewhere between an integer and a floating-point instruction for each of the XLU primitive operations. For example, no single XLU primitive approaches the algorithmic complexity of,

say, the FADD floating-point addition instruction of the Intel i387 Architecture, but the XLU addi operation is algorithmically more complex than the integer ADD instruction of the companion i386 Architecture [IP86] [Cof83]. Using this "mid-range" complexity level as a rule-of thumb, we believe that the entire set of XLU primitives will fit into less silicon than is required to implement a traditional 32-bit ALU/FPU pair of current complexity/capability.

6.2.2 What is the Expected Performance?

The XLU design trades away high instruction density to gain high instruction throughput (through the regularity of its primitive operations and their operands) and high instruction parallelism.

Increasing the availability of fine-grain i.e. instruction-level, parallelism is a primary goal of the XLU design. Two of the design's primary traits – minimization of the interdependence of individual primitive operations upon each other's results, and avoidance of branching instructions – promote this goal. Both of these traits are a natural outcome of the signed-digit number representation underlying the XLU digit specification.

Given this, we expect processors impelementing an XLU digit type and the primitive operations that manipulate it to gain performance in proportion to the number of XLU primitive operations the processor can perform in parallel. We believe the ability to implement a number of complete "XLU processor function-subunits" within the floorplan of a single processor is not unreasonable, based upon the superscalar processor designs produced during the 1990s (e.g. the Intel Pentium, HP PA-RISC 8K, 9K and 10K and the Motorola/IBM PowerPC architectures).

As mentioned in previous sections, performance also depends upon the processor's compiler. In the same way that the first MIPS processors relied upon the compiler to generate pipeline interlocks in the instruction stream,

rather than spend hardware resources for that purpose [Tan90], the XLU primitive operations depend upon all the capabilities a good optimizing compiler can bring to bear on the source code.

The success of RISC computer systems of the 1990s illustrate how well compilers can reorder code. Examples of optimization techniques that give good results for floating-point arithmetic [Dal89] and for situations where a particular idiom (say, the XLU addition or subtraction code sequence) may be optimized "offline", then generated "from memory" upon identification of the idiom's use [Mas87] increase our confidence in the ability of compiler optimizations to provide performance gains for an XLU implementation.

6.3 Summary of the XLU Architecture

What follows is a final summary of the main points of the XLU Architecture divided into three separate lists; one defining the important characteristics, a second stating the positive claims we feel we can make about the design, and a third listing the potential negative aspects offsetting the perceived gains.

6.3.1 Imporant Characteristics

- There is one base-level number representation (the XLU digit) provides a basis for the synthesis of any desired number type.
- The number representation is a signed-digit, redundant form.
- The number representation has very high radix.
- The primitive operations are kept regular and simple (in the tradition of the original RISC MIPS ideas).

- Primitive operations were chosen and designed to allow implementation of arithmetic algorithms with minimum branching.

6.3.2 Positive Implications of these Characteristics

- Since there is only one base-level number representation, all available hardware may manipulate that representation. There is no integer ALU/FPU dichotomy.
- The limited carry-digit increases the potential for parallel addition and subtraction algorithms, and portions of multiplication algorithms. There is a high level of instruction-level parallelism and operation independence available to the compiler.
- The potential to scale up is "built into" primitive operations set. The high radix and parallelism inherent in the algorithms means that multi-digit numbers may be handled without too high an overhead.

6.3.3 Negative Implications of these Charactersitics

- Small numbers i.e. magnitudes of less than one radix, may not be processed as efficiently as traditional ALU/FPU combinations. Although multi-digit numbers scale well, there is some overhead, and for "small" i.e. single-digit or double-digit values, the overhead may be more than for traditional ALU/FPU designs.
- There is an added burden for the high-level language translators. There is also an additional burden for writing code "manually" at the "assembler" level. (This is reflected to some extent in all RISC-based instruction sets.)

- The code density is poor, relative to traditional ALU/FPU designs. Again, this reflects the choice of a RISC-style approach. Instruction density has been traded for the ability to handle more instructions efficiently at a time. The code density of an XLU-base processor would be even less than a RISC design featuring a traditional ALU/FPU section.

6.4 How to Evaluate the Design

The following section outlines plans for evaluating the XLU architecture. Beyond a few a priori observations, making claims about XLU performance in time or space requires implementing an XLU-based processor or series of processors, running a series of test programs (benchmarks) on the implementation(s) and gathering metrics based on the program runs.

Between iterations of testing, we can use the data gathered to modify the XLU primitive operations set, or find better sequences of them to express our goals, or find better optimizations to apply to those sequences.

6.4.1 A Priori Observations

The XLU architecture trys to provide as much flexibility and instruction-level parallelism as possible to the processor in which it is implemented. The XLU digit and primitive operations provided are a generalized system from which many specific numeric types may be manipulated many ways.

The signed-digit arithmetic upon which the XLU design is based gives an implementing processor the ability to perform addition or subtraction on XLU digits (processor words) without regard for carry propagation. As we have seen, the addition algorithm requires three distinct steps, realized in

three primitive operations. This means that for small values the XLU does more work that the equivalent integer or floating-point hardware.

As the size of the values required increases to multiple processor words, sufficient numbers of XLU processors working in parallel provide a constant time of three operations per result, since all of the primitive operations for each of the three steps are independent and carry propagation is strictly limited. Integer or floating-point hardware requires linear processing of results as the number of processor words required to express the result increases due to carry propagation between processor words.

A single XLU cannot expect to do better than a standard processor design of even modest power but as the number of XLUs implemented into a processor increases, its ability to outperform standard processors does also. To discover how much requires more than a priori observations. For that, we require a working XLU-based processor.

6.4.2 Implementing an XLU-based Processor

Evaluating XLU-based implementations by simulation is more attractive than actually realizing different XLU-based implementations in hardware. A simulator is easier to reconfigure and instrument than actual hardware.

The existing XLU simulator (see Appendix B) is not adequate for the task of evaluating the design. Considerable effort would be required to bring it up to the level of a full simulation of a microprocessor including one or more XLUs.

A possible alternative would be to modify the existing simulator for Hennessy and Patterson's DLX [PH90] microprocessor. Several XLUs would replace the ALU/FPU portion of the DLX, and the XLU primitive operations would be added to the ISA. Apart from the obvious advantage of not having to create a complete simulator from scratch, another benefit of modifying

the DLX is that the unmodified version can then be used for side-by-side comparisons with the DLX+XLU version.

6.4.3 Gathering Data

Once we have a simulator that implements an XLU-based processor, we can use test programs (benchmarks) run on it to search the design space of the XLU. Arbitrarily, we propose breaking the testing into three steps or "levels"; basic arithmetic, simple programs and finally, large programs.

Each level of testing should provide us with data about XLU-base processor performance for time and space, and relative to standard processors. We can then use the data collected to modify the XLU design, improving it based upon our findings.

6.4.4 Level One Tests

Our first test programs should be simply additions of numbers of various digit lengths. From our a priori observations, we can predict that for integers, in terms of instruction count and running time, the XLU won't do better than a standard ALU until the number of digits (equivalent to processor words here) increases beyond about three.

Number types with nontrivial representations such as floating-point present more interesting fare for our relatively "uninteresting" addition tests. Efficient representations for floating-point numbers in XLU digits and optimizations that may be applied to the floating-point addition algorithm are things we need to discover based on simulator results.

We believe that for types like floating-point, the advantage of carry propagation may not make up for the fact that scaling may take a significant number of XLU primitive operations to perform. We should still see the XLU-based version performing better than the standard version, but any crossover in advantage to the XLU will be at later point than for integers.

After addition tests, multiplying numbers of various digit lengths should be tried. The highly independent nature of the XLU primitive operations should provide interesting opportunities for optimization of the instruction sequences.

As with addition we predict that the XLU will not do better than a standard design until the number of digits (processor words) involved in the calculation increases beyond those which a standard processor is capable of handling directly (word or double-word) and that floating-point or similar complicated number representations will present less performance to the XLU than integers.

However, for addition, subtraction or multiplication operations, if the number representation tested is not native to the standard processor (e.g. a fixed-point number type) then we believe the XLU-based design should provide competitive performance even for small sizes (number of processor words) because the basic arithmetic operations must be synthesized from instruction sequences on both the XLU and standard processors.

The overall goal for this initial level of testing is to explore how the XLU architecture performs basic arithmetic for a variety number representations, for a range of available XLUs per processor. The information gathered would us to modify the design and composition of the XLU primitive operation set, if required. It may also provide us with information about how many XLUs per processor is optimal for a given number representation.

As a postscript to this level of testing, we should extend the XLU design to include provisions for performing division operations (and perhaps remainder and square root) before continuing to level two. Assuming we did extend the

architecture in this way, those operations would require the same testing described for addition, etc.

6.4.5 Level Two Tests

Once we have data for basic arithmetic, and have applied adjustments to the XLU design based on that data. We need to explore how XLU-based processors perform for nontrivial yet still fairly simple programs. Again, we need to exercise XLU-based processors with varying numbers of XLUs available for various number representations.

As initial test programs for integer, fixed-point and floating-point number representations, we propose the RSA encryption algorithm, Mandelbrot set calculations, and the computation of π and/or e.

The RSA encryption algorithm provides a good way to test the multiplication of large integers, and an opportunity to explore integer representations of different radices, since the computations are modulo a specific value.

Mandelbrot calculations are usually performed using a processor's floating-point number type. However the arithmetic involved is well-suited to a fixed-point number representation, or even a representation of complex numbers with fixed-point components.

We propose the computation of π or perhaps e for floating-point representations not so much because the calculation possesses special traits but because computation of trancendentals is a traditional exercise for new floating-point processors. Hence a large body of algorithms and optimization work already exist for such computation.

For all of these tests we need to study both fixed-range number representations and dynamic-range number representations For this latter type, the range of representable values is allowed to grow as required by adding XLU digits a number as required.

For dynamic-range number representations, we expect XLU-based processors to perform better, for two main reasons. First such representations must be synthesized by sequences of instructions on standard processors as well as the XLU-based processors (the standard processor gains no speed advantage from hardwired instructions in this case) and presumably the standard processor's ISA was not specifically designed with this sort of thing in mind (as was the XLU). Second, adding digits may actually improve XLU-based computations, since it may also reduce the need for some types of scaling and normalization.

Independent of any particular number representation, a primary focus of all these level two tests is to gather and study the sequences of code generated for the algorithms. Optimization opportunities for sequences of basic arithmetic should be found, and generalized into rules for a compiler to apply.

We believe that for some case the level of optimization will be quite high, but not for all cases. Algorithms that require large amounts of inter-digit (inter-word) scaling operations may not fare as well for optimization, because the amount of parallelism available at the primitive operation level is smaller than (for example) the totally parallel addition algorithm.

6.4.6 Level Three Tests

Passing beyond the level two tests implies we've iterated the XLU design at least once. We should have a feel for what the "right" number of XLUs per processor is for a given requirement. Assuming that, in addition, we've used the data gathered to improve the XLU primitive operations set and the compiler's optimization ability, we should, at level three discover how XLU-based processors perform on full benchmark test suites and on real programs. An example of a likely benchmark is the SPEC suite. Real programs could

include anything from rendering complex graphics with Renderman or a similar program, Analog circuit simulation using SPICE, or formatting a sizable document with TeX.

To reach this level of tests would require a *significant* amount of work. Both our simulator and the compiler servicing it would need to be complete enough to handle "real" code. Getting an experimental compiler to the level of efficiently handling production-level C source code (or any other language) is a research project in and of itself.

At this level of testing, we are at last evaluating the XLU architecture "globally". Distribution of the code across the XLUs available on the processor is of interest and a study of possible global-level code optimizations should be done. It is also at this level that we will finally be able to say with some authority whether the XLU design is viable or not relative to standard processor designs.

7 Conclusion

In this thesis, we first examined the evolution of architectures for performing computer arithmetic, and noted how different approaches rapidly converged to a single approach.

This "standard" architecture's characteristics include direct hardware support for a binary, signed-digit integer (the "ALU") and separate, disjoint circuitry providing direct hardware support for a floating-point format (the "FPU"). Regardless of the overall processor design, the ALU instruction set is most often RISC-like, while the FPU instruction set is very CISC-like.

We do not argue with the success of this "traditional" ALU/FPU design. Instead we propose a new architecture, "XLU", as an alternative. Our approach provides unification of the traditional ALU/FPU circuitry and instruction sets by defining one basic number representation; the XLU digit.

The XLU digit contains enough information to support many different number formats. We provide examples of integer, fixed-point and floating-point within this work. Moreover, the XLU digit allows us to used signed-digit arithmetic operations which, as we have shown, provide significant amounts of instruction-level parallelism during the basic operations of addition, subtraction and multiplication.

The XLU instruction set is RISC-based, and designed to minimize branching. These characteristics provide language translation tools (compilers, etc.) with wide possibilties for optimization, and at the same time make efficient use of available circuitry.

Our design involves a tradeoff between complexity in circuitry and in code. The XLU digit and the average XLU instruction are more complex to realize in circuitry than the average traditional integer value or ALU instruction, but are significantly less complex to realize than the average floating-point value or FPU instruction.

This tradeoff means that the XLU architecture as a whole allows a large amount of flexibility between time/space tradeoffs in implementation while still keeping overall performance as a reachable end-goal. When more circuitry is available, some poritions of the basic arithmetic algorithms may be hardwired and time performance is enhanced. When less circuitry is available, the algorithms are generated as instruction sequences, but may be agressively optimized, so that time performance does not greatly suffer.

We believe that this gives a processor designer using the XLU approach more freedom relative to the traditional ALU/FPU design. He may choose to devote more of the available circuitry to arithmetic performance, and gain a relative speed advantage over traditional ALU/FPU designs, or he may choose to devote less, and reduce the overall circuitry required for arithmetic at a smaller cost to performance than for a traditional ALU/FPU.

This thesis does not provide details of implementation, experimental metrics for size and time, or any exploration of tradeoffs using the XLU architecture. Initial results from a basic simulator provided us with evidence that the algorithms are sound and the design is realizable in reasonable amounts of code or circuitry, but more detailed simulations are required to gather accurate time and space metrics. Such simulations (and an advanced simulator) lie beyond the coverage of this work.

Within its limits, we believe that the work presented here reaffirms the fact that there are many ways to design the arithmetic portion of a processor, and forms a viable basis to support future work.

References

[AA93] Donald Alpert and Dror Avnon. Architecture of the pentium processor. *IEEE Micro*, pages 11–21, June 1993.

[AAD+93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance features of the pa7100 microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Architectural Support for Programming Languages and Operating Systems*, pages 108–120. ACM and IEEE, April 1991.

[AT67] Algirdas Avizienis and Chin Tung. Design of combinational arithmetic nets. In *Digest of the 1rst Annual IEEE Computer Conference*, pages 25–28, September 1967.

[AT70] Algirdas Avizienis and Chin Tung. A universal arithmetic building element (abe) and design methods for arithmetic processors. *IEEE Transactions on Computers*, C-19(8):733–745, August 1970.

[AT93] Makoto Awaga and Hiromasa Takahashi. The μvp 64-bit vector coprocessor. *IEEE Micro*, pages 24–36, October 1993.

[AtWC84] Loyce M. Adams and thomas W. Crockett. Modeling algorithm execution time on processor arrays. *IEEE Computer*, 17(7):38–43, July 1984.

[Avi61] Algirdas Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10:289–400, 1961.

[Avi66] Algirdas Avizienis. Arithmetic microsystems for the synthesis of function generators. *Proceedings of the IEEE*, 54(12):1910–1919, December 1966.

[BAM+93] Michael C. Becker, Michael S. Allen, Charles R. Moore, John S. Muhich, and David P. Tuttle. The powerpc 601 microprocessor. *IEEE Micro*, pages 54–67, October 1993.

[BD84] Pradip Bose and Edward S. Davidson. Design of instruction set architectures for support of high-level languages. In *The 11th Annual International Symposium on Computer Architecture*, pages 198–206. ACM and IEEE Computer Society, June 1984.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrated register allocation and instruction scheduling for riscs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131. ACM and IEEE, April 1991.

[BGvN61] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. *John von Neumann Collected Works*, volume 5, chapter Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, pages 41–65. Pergamon Press, Oxford, England, 1961.

[BH92] Robert J. Baron and Lee Higbie. *Computer Architecture: case studies.* Addison-Wesley Publishing Co., Inc., Reading, Massachusetts, 1992.

[BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):144–147, March 1982.

[BSBF89] David H. Bailey, Horst D. Simon, John T. Barton, and Martin J. Fouts. Floating point arithmetic in future supercomputers. *Supercomputer Applications(?)*, 3(3):86–90, Fall 1989.

[BSK92] Jr. Burton S. Kaliski. Multiple-precision arithmetic in c. *Dr. Dobb's Journal*, pages 40–48,116,117,118,119, August 1992.

[BSK93] Jr. Burton S. Kaliski. the z80180 and big-number arithmetic. *Dr. Dobb's Journal*, pages 50–58, September 1993.

[CCG+84] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984.

[CGLT89] Robert Cohn, Thomas Gross, Monica Lam, and P.S. Tseng. Architecture and compiler tradeoffs for a long instruction word microprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Boston, Massachusetts, April 1989. ACM and IEEE.

[CH90] Rulph Chassaing and Darrell W. Horning, editors. *Digital Signal Processing with the TMS320C25.* John Wiley and Sons, Inc., New York, NY, 1990.

[Cho89] Paul Chow, editor. *The MIPS-X RISC Microprocessor.* Kluwer Academic Publishers, Norwell, MA, 1989.

[Cir95] Joe Circello. Cold fire: A hot architecture. *Byte Magazine*, pages 173–174, May 1995.

[CM90] John Cocke and V. Markstein. The evolution of RISC technology at IBM. *IBM Journal of Research and Development*, 34(1):4–11, January 1990.

[Cof83] James W. Coffron. *Programming the 8086/8088.* Sybex, Berkeley, CA, 1983.

[Coo80] Jerome T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *IEEE Computer*, 13(1):68–79, January 1980.

[Cra80] Harvey G. Cragon. The elements of single-chip microcomputer architecture. *IEEE Computer*, 13(10):27–41, October 1980.

[Dal89] William J. Dally. Micro-optimization of floating-point operations. In *Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, Massachusetts, April 1989. ACM and IEEE.

[Dew88] A. K. Dewdney. *The Armchair Universe.* W. H. Freeman and Co., New York, New York, 1988.

[DHK93] Jean Duprat, Yvan Herreros, and Sylvanus Kla. New redundant representations of complex numbers and vectors. *IEEE Transactions on Computers*, 42(7):817–824, July 1993.

[DM82] David R. Ditzel and H.R. MacLellan. Register allocation for free: The c machine stack cache. In *Proceeding of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Palo Alto, California, March 1982. ACM and IEEE.

[Doe93] Ronald W. Doerfler. *Dead Reckoning, Calculating Without Instruments.* Gulf Publishing Company, Houston, Tx, 1993.

[DV87] Jack W. Davidson and Richard A. Vaughan. The effect of instruction set complexity on program size and memory performance. In *Architectural Support for Programming Languages and Operating Systems*, pages 60–64, Palo Alto, California, October 1987. ACM and IEEE.

[DW90] Jack W. Davidson and D.B. Whalley. Reducing the cost of branches by using registers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 182–191, Los Alamitos, CA, May 1990.

[EL92] Miloš D. Ercegovac and Thomas Lang. On-the-fly rounding. *IEEE Transactions on Computers*, 41(12):1497–1503, December 1992.

[ES94] Mohamed El-Sharkawy, editor. *Signal Processing, Image Processing and Graphics Applications with Motorola's DSP96002 Processor.* Prentice-Hall, Inc., Engelwood Cliffs, NJ, 1994.

[Fat82] Richard J. Fateman. Hight-level language implications of the proposed ieee floating-point standard. *ACM Transactions on Programming Languages and Systems*, 4(2):239–257, April 1982.

[Fly80] Michael J. Flynn. Directions and issues in architecture and language. *IEEE Computer*, 13(10):5–22, October 1980.

[Gar76] Harvey L. Garner. A survey of some recent contributions to computer arithmetic. *IEEE Transactions on Computers*, C–25(12):1277–1282, December 1976.

[GG96] H.H. Goldstine and Adele Goldstine. The electronic numerical integrator and computer (ENIAC). *IEEE Annals of Computing History*, 18(1):10–16, 1996. This reprint of the original paper appeared in the Annals of Computing History by permission of the American Mathematical Society and National Academy of Sciences.

[GLS93] Susan L. Graham, Steven Lucco, and Oliver Sharp. Orchestrating interactions among parallel computations. *ACM SIGPLAN Notices*, 28(6):100–111, June 1993.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[Han94] Per Brinch Hansen. Multiple-length division revisited: a tour of the minefield. *Software–Practice and Experience*, 24(6):579–601, June 1994.

[HJB+82] John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–10, Palo Alto, California, March 1982. ACM and IEEE.

[Hol97] W. Neville Holmes. Composite arithmetic: Proposal for a new standard. *IEEE Computer*, 30(3):65–73, March 1997.

[Hwa93] Kai Hwang. *Advanced Computer Architecture with Parallel Programming*, pages 286–298. McGraw-Hill, Inc, New York, preliminary edition, 1993.

[IP86] William H. Murray III. and Chris H. Pappas. *80386/80286 Assembly Language Programming*. Osborne McGraw-Hill, Berkeley, CA, 1986.

[Jep99] Brian Jepson. Old workstations never die. *Performance Computing*, page 14, March 1999.

[Jou89] Norman P. Jouppi. Architectural and organizational tradeoffs in the design of the multititan cpu. Wrl research report 89/9, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301 USA, July 1989.

[Jr.90a] Earl E. Swartzlander Jr., editor. *Computer Arithmetic, Volume II*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Jr.90b] H. S. Warren Jr. Instruction scheduling for the ibm risc system/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.

[KAJW93] Sanjaya Kumar, James H. Aylor, Barry W. Johnson, and Wm. A. Wulf. A framework for hardware/software codesign. *IEEE Computer*, 26(12):39–45, December 1993.

[KC93] Kishore Kota and Joseph R. Cavallaro. Numerical accuracy and hardware tradeoffs for cordic arithmetic for special-purpose processors. *IEEE Transactions on Computers*, 42(7):769–779, July 1993.

[KD91] Robert F. Krick and Apostolos Dollas. The evolution of instruction sequencing. *IEEE Computer*, 24(4):5–15, April 1991.

[KDSPS77] David J. Kuck, Jr. Douglass S. Parker, and Ahmed H. Sameh. Analysis of rounding methods in floating-point arithmetic. *IEEE Transactions on Computers*, C-26(7):643–650, July 1977.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice-Hall, Engelwood Cliffs, New Jersey, 1992.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2; Seminumerical Algorithms.* Addison-Wesley Publishing Co., Inc., Reading, Massachusetts, 1981.

[Kor93] Israel Koren. *Computer Arithmetic Algorithms.* Prentice Hall, Engelwood Cliffs, New Jersey, 1993.

[KS77] Peter Kornerup and Bruce D. Shriver. A unified numeric representation arithmetic unit and its language support. *IEEE Transactions on Computers*, C-26(7):651–659, July 1977.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[Lil94] David J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, 27(2):13–26, February 1994.

[LKB91] Roland L. Lee, Alex Y. Kwok, and Fayé A. Briggs. The floating point performance of a superscalar sparc processor. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 28–36. ACM and IEEE, April 1991.

[MA96] Mitchell Marcus and Atsushi Akera. Exploring the architecture of an early machine: The historical relevance of the eniac machine architecture. *IEEE Annals of the History of Computing*, 18(1):17–24, 1996.

[Mas87] Henry Massalin. Superoptimizer – a look at the smallest program. In *Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, California, October 1987. ACM and IEEE.

[MHR90] R. K. Montoye, E. Hokenek, and S. L. Runyon. Design of the IBM RISC system/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, January 1990.

[Mon85] Peter L. Montgomery. Modular multipliation without trial division. *Mathematics Of Compuation*, 44(170):519–521, April 1985.

[Mot93] Frederick C. Motteler. Arbitrary precision floating-point arithmetic. *Dr. Dobb's Journal*, pages 28–34,84,86,87, September 1993.

[MRMP80] Stephen P. Morse, Bruce W. Ravenel, Stanley Mazor, and William B. Pohlman. Intel microprocessors – 8008 to 8086. *IEEE Computer*, 13(10):42–60, October 1980.

[mWHC92] Wen mei W. Hwu and Pohua P. Chang. Efficient instruction sequencing with inline target insertion. *IEEE Transcations on Computers*, 41(12):1537–1551, December 1992.

[Och91] Tom Ochs. Numeric types, representations, and other fictions. *Computer Language*, 8(8):?, August 1991.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hamond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Architectural Support for Programming Languages and Operating Sytems VII*, pages 2–11. ACM and IEEE, October 1996.

[Par90] Behrooz Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, January 1990.

[Pat94] Peter C. Patton. Multiprocessors: Architecture and applications. *IEEE Computer*, 18(6):29–40, June 1994.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Inc., San Mateo, CA, 1990.

[PV94] James Phillips and Stamatis Vassiliadis. High-performance 3-1 interlock collapsing alus's. *IEEE Transactions on Computers*, 43(3):257–268, March 1994.

[PWW97] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, January 1997.

[Ran82] Brian Randell, editor. *The Origins of Digital Computers.* Springer-Verlag, Berlin, 1982.

[Rob58] James E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, 7:218–222, September 1958.

[Roj97] Raúl Rojas. Konrad zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16, April–June 1997.

[Rym82] James W. Rymarczyk. Coding guidelines for pipelined processors. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 12–19, Palo Alto, California, March 1982. ACM and IEEE.

[SC92] André Seznec and Karl Courtel. Opac: A floating-point coprocessor dedicated to compute-bound kernals. In *The 19 Annual International Symposium on Computer Architecture*, page 427. ACM SIGARCH and IEEE Technical Committee on Computer Architecture, May 1992.

[Sch94] Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, Inc., New York, New York, 1994.

[Sit93] Richard L. Sites. Alpha axp architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[SKG77] William J. Stenzel, William J. Kubitz, and Giles H. Garcia. A compact high-speed parallel multiplication scheme. *IEEE Transactions on Computers*, C-26:948–957, 1977.

[Sto92] Harold S. Stone. Copyrights and author responsibilities. *IEEE Computer*, 25(12):46–51, December 1992.

[Tan90] Andrew S. Tanenbaum. *Structured Computer Organization - 3rd ed.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

[UAN+93] Kunio Uchiyama, Fumio Arakawa, Susumu Narita, Hirokazu Aoki, Ikuya Kawasaki, Shigezumi Matsui, Mitsuyoshi Yamamoto, Norio Nakagawa, and Ikuo Kudo. The Gmicro/500 superscalar microprocessor with branch buffers. *IEEE Micro*, pages 12–22, October 1993.

[USS97] Augustus K. Uht, Vijay Sindagi, and Sajee Somanathan. Branch effect reduction techniques. *IEEE Computer*, 30(5):71–81, May 1997.

[vN45] John von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Enginnering, University of Pennsylvania, 1945. Exact copy (but with typographical errors corrected) reprinted in IEEE Annals of the History of Computing, Vol.15, No.4, 1993.

[VPB93] Stamatis Vassiliadis, James Phillips, and Bart Blaner. Interlock collapsing alus's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[Wal90] David W. Wall. Limits of instruction-level parallelism. Wrl technical note tn-15, Digital Equipment Corporation Western Research Laboratory, January 1990.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188. ACM and IEEE, April 1991.

[WF82] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Holt, Rinehart and Winston, Orlando, Florida, 1982.

[Wil93] Al Williams. 32-bit floating-point math. *Dr. Dobbs Journal*, pages 70–74,76,78,80, June 1993.

[WS91] Andrew Wolfe and John P. Shen. Variable instruction stream extension to the vliw architecture. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14. ACM and IEEE, April 1991.

[Yoh73] J. Michael Yohe. Roundings in floating-point arithmetic. *IEEE Transactions On Computers*, C-22(6):577–585, June 1973.

Appendices

Appendix A - Small Glossary of Terms

Architecture An abstract specification or list of attributes, either of hardware or software.

In his article describing the Alpha AXP Architecture Sites [Sit93] repeats a definition of Amdahl et al., defining

> COMPUTER ARCHITECTURE ... the attributes and behavior of a computer as seen by a machine language programer. This definition includes the instruction set, instruction formats, operation codes, addressing modes, and all registers and memory locations that may be directly manipulated by a machine language programmer.

We use the term here in this same sense, although our architecture is only a partial description of a single portion of a CPU (i.e., the arithmetical portions of the arithmetic-logic unit).

In a Unix Review article, Brian Jepson [Jep99] gives another, definition that, while less formal, is still useful and slightly amusing:

> Architecture is a term used in computing that refers to the topology of anything that can be looked at as a whole made of separate parts, such as a network, operating system, or computer. When architecture is brought up in polite conversation, it's usually used to discuss the contstraints put on you by whoever designed the thing you are talking about.

Implementation A particular physical example of an architecture or algorithm. One instance of the possibilities presented by an architecture or algorithm.

Instruction In the context of this work, the implementation of an algorithm (arithmetical or otherwise) through a sequence of one or more XLU

primitives. An instruction is a sequence of one or more XLU primitive operations.

ISA Instruction Set Architecture – the specification of the instruction set for a processor, or, in the case of the XLU primitive instructions, for a subset of the instruction set for a processor.

Multiplication Cell The simplest complete XLU multiplication sequence involving only two digits; one as multiplicand, the other as multiplier.

Pure Fraction A number with its radix position to the right of its rightmost digit. See [Kor93].

Pure Integer A number with its radix position to the left of its leftmost digit. See [Kor93].

Radix The base of a number system (e.g. radix-10 = base-10 or decimal, and radix-2 = base-2 or binary).

Radix point Separator between the integer and fractional portions of a number.

Signed-digit Arithmetic The algorithms for performing the four basic arithmetic functions (addition, subtraction, multiplication and division) using Signed-digit numbers. See [Avi61] and [Kor93].

Signed-digit Number A member of a fixed-radix number system in which the digit set may contain both positive and negative values. See [Avi61] and [Kor93].

XLU The "eXperimental aLU" (pronounced "clue"). The overall name for the architecture we propose in this work. The "X" is given the χ ("chi") sound, following Knuth's pronounciation of TeX.

XLU Digit (Also, just "digit") The fundamental data type of the XLU architecture. It is designed to form a base containing just enough information to support derivation of whatever number types are required.

XLU Instruction (Also, just "instruction") Informally, this may be used as a synonym for "XLU Primitive", but formally it is a synonym for "Instruction".

XLU Primitive (Also, "XLU operation") A base-level ability or function, such as "Add two digits and return the carry out value", expressible in a single XLU mnemonic.

Appendix B - The XLU Simulator

Early on in the course of our work, we wrote a simple software simulator to help us test ideas, and to encourage "tinkering" that might lead to new insights. What follows is a brief description of the basic simulator's design, functionality and usage.

The simulator was never "complete"; new features were added as research or whim required. The plan was to extend the abilities of the simulator as the basic XLU architecture (presented in this work) was analyzed and extended in further study. This Appenix covers the basic functionality in place in the simulator during the formulation of the ideas presented in this work.

Design

The XLU architecture represents only a small portion of a complete processor, and the XLU simulator simulates that portion. Originally, It had no provision for branching and assumed a simple memory model consisting of a large number of registers, and flat memory. The memory addressing scheme is a pure load-store model. Values are loaded from memory to registers and stored from registers to memory. Operations are only register-to-register.

Later additions included statement labels (for planned branching features) and several instructions to report on or analyze various aspects of XLU operations sequences loaded into the simulator. These advanced features were not used extensively during the development of the ideas presented in this work.

Implementation

The simulator is implemented in several thousand lines of C code. Flex and bison lexor and parser generator tools were used to create the foundation of the code; individual functions implement individual XML primitive operations, or, in some cases a short sequence of XML primitive operations (for exploring hybrid RISC/CISC variations of the XML operations set).

A simulated 16-bit machine word yields an XML digit with 14 bits of magnitude. For simplicity XML digits are input or output to and from the simulator as multi-digit, positive or negative radix-10 integers. So, for instance the XML digit with the magnitude -1024 is keyed into the simulator as exactly same string; -1024. (The simplicity is in relation to the implementation, not the user, although radix-10 should not be a great inconvenience for the user.)

Usage

In operation, the simulator functions as a sequential interpreter. Programs, consisting of sequences of "instructions" and "statements" are either typed directly into the simulator at its ready prompt, or presented to it in a file via a `include` instruction. Instructions are commands that directly tell the simulator to perform some meta-action (e.g. `clear_reg` tells the simulator to clear all its registors). Statements consist of an XLU primitive operation, followed by its arguments, the keyword `to` and the location (register) to store the result.

Here is a simple example of using xmc to directly interpret a single XLU primitive operation[9] statement:

[9]Notice that XMC recongnizes `add_i` for the XLU primitive operation `addi`. We had to code the primitive ops names into the simulator with underbar-separators, but we frequently dropped them when writing notes or referring to them in email, and ended up

```
$ xmc
>interpret
>add_i 500 1400 to %0
>print %0
 1900
>quit
xmc done
$
```

First of all, the `interpret` instruction places the simulator in direct interpreter mode – each carriage-return is assumed to signal the end of either an XMC instrution or an XLU statement. Next, the intermediate sum of the two XLU digits "500" and "1400" is computed, and the result stored in register 0. Then the contents of register 0 are echoed using the XMC `print` instruction, and finallly, the `quit` instruction exits the XMC simulator.

---

specifying them without underbars, while never changing the names in the simulator.