

A VLIW Architecture for Logarithmic Arithmetic

Mark G. Arnold
Lehigh University
Bethlehem, PA 18015 USA
marnold@eecs.lehigh.edu

Abstract

The Logarithmic Number System (LNS) is an alternative to IEEE-754 standard floating-point arithmetic. LNS multiply, divide and square root are easier than IEEE-754 and naturally belong to the same class of one-cycle-latency instructions like integer addition, subtraction and shifting.

LNS addition is harder, requiring several cycles if the integer add determines the clock. Unlike prior LNS Instruction Set Architectures (ISAs), the proposed ISA uses a separate pipelined unit specialised for LNS addition that operates in parallel to a faster LNS-multiply-divide/integer-add unit. Their latencies are different: the former uses four to six cycles; the latter uses only one. The proposed Very Long Instruction Word (VLIW) architecture includes novel LNS increment-multiply and input-conversion instructions that improve performance at very low cost. The ISA overloads a novel comparison flag: LNS-less-than for divide and integer-less-than for subtract.

Features of other ISAs are omitted here due to hidden costs of feeding extra operands to the ALUs. The proposed ISA offers multiplication bandwidth equal to (not higher than) that of addition. Also, omitting a multiply-accumulate instruction does not degrade LNS performance at all.

Keywords: Logarithmic Number System, Very Long Instruction Word, pipeline, sum of products.

1. Introduction

One of the major motivations for parallel processing is to increase the speed of mathematical algorithms. These algorithms are easiest to describe when they are thought to manipulate the set of real numbers postulated by mathematical theory. Such numbers, like π and $100^{1000}10000$, can be described by formal mathematics, but do not actually exist inside ordinary computers. Rather, we accept two kinds of limitations. First, the maximum number representable is

within a finite dynamic range (say $\approx 2^{\pm 128}$) which eliminates the latter kind of astronomical value. Second, the precision is finite, which eliminates exact values like π , but allows finite approximations thereof.

There is a cost/benefit analysis involved: greater precision and greater dynamic range make it easier to program the processors and give better answers. Lower precision and lower dynamic range reduce the area of VLSI circuits, increase their speed and lower their power consumption. For general-purpose computers, we ignore the cost of excess dynamic range and excess precision for the sake of programming ease and compatibility with the IEEE-754 standard. For special-purpose computers, where we have greater knowledge of the application requirements, we seek the minimum dynamic range and minimum precision that produce acceptable results, and we may be willing to explore non-standard techniques in this quest for efficiency.

To do so, we need to distinguish between a mathematical value, denoted by a lower-case variable like x , and its machine representation, X , which is a finite word that can be processed inside a computer. We express a computation involving x with the ordinary operations of mathematics, such as multiplication, but the computer carries out our requests by manipulation of the bits that compose X .

There are two common number systems that allow real values to be represented: the fixed-point number system (FXNS) and the floating-point number system (FPNS). FXNS uses integer arithmetic with an implied scaling constant; thus it has the same cost and speed as integer operations. FPNS allows the hardware to determine the scaling through the use of exponents that are part of the representation, thus it has more complicated hardware. This paper explores a non-standard number system, called the Logarithmic Number System (LNS) [11], which does a better job of maintaining relative precision than FPNS. For low-to-moderate-precision applications in DSP and graphics, LNS consumes less power [8] and requires less area [5] than the equivalent FXNS or FPNS. LNS has been applied to diverse applications, including music synthesizers for Yamaha[4]

and aircraft controls for Boeing [1]. In 1999, an LNS-based special-purpose machine designed by Makino et al. [6] that solves the astrophysical N-body problem won the Gordon Bell Prize.

In LNS, the representation X is composed of two parts: the sign bit ($X_S = \text{sign}(x)$) and the logarithm ($X_L = Q(\log_b |x|)$). b is the base of the logarithm and $Q(\cdot)$ is a quantization function which limits the word size (and correspondingly the precision and dynamic range).

2. LNS Arithmetic

As illustrated by the slide rule, the logarithm of a product is the sum of the logarithms. Given that the values x and y are already represented by X and Y , to form the representation, P , of the product simply requires a fixed-point adder ($P_L = X_L + Y_L$), and trivial logic to manipulate the sign ($P_S = X_S \oplus Y_S$). Similar economical LNS operations include division, square and square root. It is usually convenient to deal with overflow by saturation. As long as results stay inside the computer in LNS format, there is no need to use the logarithm or antilogarithm functions, as these are required only on input and output, respectively. LNS is effective for computation, like those in DSP and graphics, where the ratio of arithmetic operations to input/output operations is high.

Addition and subtraction pose a dilemma, however. LNS addition and subtraction effectively require the computation of the logarithm of the sum or difference of the antilogarithms of the two LNS operands. It is well known [11] that this dyadic LNS addition function can be reduced to a monadic function, $F(z) = \log_b |1 \pm b^z|$, where z is the logarithm of the ratio of the values. The justification for this can be viewed from the perspective of the values x and y : $x \pm y = y \cdot (x/y \pm 1)$. Thus, the representation of the total is:

$$T_L = Y_L + F(X_L - Y_L) = \log_b |x \pm y| \quad (1)$$

The sign of the result, T_S , and whether F is calculated as $\log_b (1 + b^z)$, known as the addition logarithm, or $\log_b |1 - b^z|$, known as the subtraction logarithm, depends on the signs of the operands, X_S and Y_S . The addition logarithm is used when the signs are the same; the subtraction logarithm is used when they are different. For low-precision systems (word size of less than, say, 12 bits), F can be implemented by direct table lookup using a ROM or PLA [11]. For larger word sizes, table reduction [12, 10] and/or interpolation [5] are feasible. The addition logarithm is much easier to interpolate than the subtraction logarithm because the latter has a singularity at $z = 0$ [5]. To calculate the subtraction logarithm in the same time as the addition logarithm requires a much larger ROM. Alternatively, one can trade area for speed. The time for adding numbers of the

same sign can be shorter than for numbers of different signs [2, 1]. The architecture described below capitalises on this variable latency.

Figure 1 shows a simple linear interpolator using a multiplier. With the addition logarithm the partitioning can be as simple as splitting z into two fixed-size buses, z_H and z_L , where $0 \leq z_L < \Delta$, where Δ is constant. The low-order bits, z_L , are input to the multiplier. The high-order bits, z_H , address the function ROM ($F(z_H)$) and the slope ROM ($C(z_H)$). Figure 1 shows two pipeline stages (ROM and multiplier/adder), although depending on the technology, it might be sensible to allocate additional stage(s).

3. Prior LNS ISAs

Despite the wealth of information on LNS arithmetic algorithms, only a handful of instruction-set architectures (ISAs) have been reported for programmable processors that use LNS [2, 9, 12, 7, 3]. This section will review these.

Most prior ISAs have Harvard instruction sets, with separate memories for data and code. For example, Taylor [12] reports a simple 20-bit-LNS ALU with a 6-deep FIFO stack for data storage and a simple microcode-like instruction set. An exception to this trend towards streamlined ISAs is the ‘‘European Logarithmic Microprocessor’’ (ELM) [3], which has a memory-register CISC ISA reminiscent of the IBM-S/360.

3.1 2-Multiplier

The designers [12, 9, 2, 3] of most prior LNS ISAs wish to exploit the perceived advantages of LNS (easy multiplication and division) whilst mitigating its disadvantages (difficult addition and subtraction) by offering unequal multiplication and addition bandwidths. Several architects [12, 9, 2] have sought to achieve this goal with a datapath like Figure 2. The dotted vertical lines separate pipeline stages, and the dashed-line box implements the LNS addition algorithm. The number of pipeline stages needed for this adder depends on the implementation of the interpolator. For a multiplier-based interpolator like Figure 2, at least two stages are required: ROM access (obtain $f(z_H)$ and $c(z_H)$) and fixed-point calculation ($f(z_H) + c(z_H) \cdot z_L$). Thus, the LNS adder shown in the dashed lines has a latency of four pipeline stages, and the complete architecture has a latency of five stages. For consistency, the rest of this paper assumes a similar interpolator.

Data in Figure 2 come from a register file with at least four read ports. Compared to conventional floating-point instruction sets, the novel feature these LNS architectures [9, 12, 2] share is two parallel LNS multipliers that feed the LNS adder. Chester [2] has identified a niche application

(2 x 2 matrix operations) where this 2-multiply architecture performs well, but in general most applications do not.

3.2 ELM

The ELM [3] is a 32-bit LNS vector processor, considerably more sophisticated than the 2-multiplier architectures. The ELM has the same add/multiply bandwidth mismatch found in the 2-multiplier architectures. For LNS multiply and divide, the ELM can operate on four independent elements of a vector in a single clock cycle. This is justified the same way as in [9, 12, 2], by the supposed low cost of the fixed-point adder used for LNS multiply/divide. In the case of addition and subtraction, only two elements of the vector can be processed concurrently, and the processor stalls for three cycles (away from the singularity) or four cycles (near the singularity). The ELM has two identical interpolators, but these are not pipelined. The ELM is reported [3] as achieving 0.29 FLOPS/cycle.

3.3 2-ALU

A more mainstream approach is to make an orthogonal architecture, in which different fields of an instruction are not as interdependent on each other as they would be in a CISC. For example, Paliouras and Stouraitis [7] propose a VLIW design, which has, in the simplest case, two LNS units as shown in Figure 3. Each unit is identical and capable of LNS addition or multiplication, and this orthogonality makes the processor flexible and powerful since the same instruction set is used for each unit. Similar orthogonality is found in floating-point VLIW DSPs from vendors such as TI.

4. Novel 32-bit LNS Processor

This section describes an experimental ISA for a 32-bit Princeton VLIW LNS microprocessor. Like the earlier attempts at LNS ISAs in the literature, we are willing to depart from conventional wisdom about ISA to exploit the advantages of LNS. We are also willing to sacrifice ease of compilation or similarity to commercial processors for the sake of reduced hardware complexity. The goal is that this ISA be able to achieve at least one FLOP per cycle.

The instruction set has a variable-length-instruction format for compression purposes, but decompresses this into a longer fixed-length format for execution. Different parts of the decompressed instruction control four independent units of the datapath: the LNS unit, the ALU, the load/store/table lookup memory unit and the branch unit. Figure 4 shows how the register file of the novel processor connects to the ALU and LNS units. For brevity the memory and branch units are not shown in Figures 2, 3 and 4. The memory

Table 1. Implementation Cost

	Inter-polators	Fixed-pt. Adders
[2]	1	4
[7]	2	4
proposed	1	3

unit will require an additional four ports (two read and two write).

In contrast to the orthogonal architecture of [7] (Figure 3), where an identical style of instruction format controls both LNS ALU1 and LNS ALU2, the proposed architecture (Figure 4) is asymmetrical. An entirely different instruction set is issued to the ALU from that which is issued to the LNS unit. The proposed ALU in Figure 4 provides integer/fixed-point arithmetic/logical operations similar to those on conventional processors (integer addition and subtraction as well as bit-wise operations such as AND). The ALU also provides the LNS operations for multiply, divide, square, square root. The criterion for including an instruction in this set is whether it can be performed reasonably in a single cycle, as demanded by the single pipeline stage given for this unit in Figure 4. In contrast, the LNS unit does LNS addition and subtraction. These require several cycles to complete, as illustrated by the multiple stages in Figure 4. Although this asymmetry is a departure from accepted wisdom in RISC design, the advantage is that it cuts implementation cost, as illustrated in Table 1.

5. Instruction Set

The compressed instruction is composed of one to four *chunks*. Each chunk of an instruction is 16-bits wide. If present, branch addresses and immediate data occupy two chunks. In the decompressed instruction there are three chunks, plus the 32-bit immediate data or absolute address.

There are fifteen general-purpose registers. Register number 0 is special: as a destination it causes a result to be discarded and as a source it uses immediate data. (As described in the next paragraph, the decompression of the immediate data is specified elsewhere in the instruction.) Fifteen registers are barely enough registers by modern compiler standards, but this is as large as will fit into the dual-port single-CLB-per-bit memory of the Virtex FPGA. Implementation as an ASIC could alleviate this limitation.

The left four bits ($ir[63:60]$) of the decompressed instruction determine how many chunks are present in the current instruction. The left chunk (call it $chunk3=ir[63:48]$) is always present and is where these four bits reside. Also, $chunk3$ is where the ALU

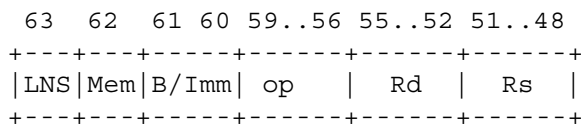
Table 2. One-cycle ALU Instruction Results

inst.	result
LMUL	(l _{ns}) Rd*(l _{ns}) Rs
ADD	(int) Rd+(int) Rs
LDIV	(l _{ns}) Rd/(l _{ns}) Rs
SUB	(int) Rd-(int) Rs
AND	(int) Rd&(int) Rs
OR	(int) Rd (int) Rs
XOR	(int) Rd^(int) Rs
SBB	(int) Rd-(int) Rs-flag
ADC	(int) Rd+(int) Rs+flag
ROR	Rd ROR Rs
MOVPC	PC
MOV	Rs

operation is specified. If $ir[63]$ is one, there will be $chunk2=ir[47:32]$ in the compressed instruction to describe the instruction for the LNS unit. If $ir[63]$ is zero, there is no operation in the LNS unit, and $chunk2$ will not be present in the compressed instruction. If $ir[62]$ is one, there will be $chunk1=ir[31:16]$ for the memory reference unit. With some restrictions if $ir[61:60]$ is not zero, there will be thirty-two bits ($ir[31:0]$) for either the branch unit or as an immediate data for the other unit(s) that reference register number 0. Since there are some prohibited combinations in $ir[63:60]$, there are some unused patterns that can be used for special instructions (enable interrupts, reset, etc.) Included among these in the simulator are instructions, not intended for hardware implementation, that print LNS values in decimal for debugging purposes.

5.1 ALU operations

Given these assumptions, the layout of $chunk3$ is



Thus, $chunk3$ specifies the length of the compressed instruction using four bits, and also specifies a two-operand single-cycle ALU instruction using twelve bits. Rd changes at the beginning of the next clock cycle as described in Table 2. If Rs references register 0, the immediate value later in the instruction will be used as the operand instead. If the immediate value is not permitted, the value used is not defined. (There is no register 0.) The above list shows casts of either (int) or (l_{ns}), depending on whether the instruction interprets the word as an integer or an LNS value. To

Table 3. Flag Usage

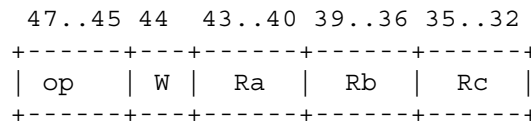
inst.	flag	inst.	flag
AND	0	XOR	Rd==Rs
OR	0	SUB	(int) Rd<(int) Rs
MOV	flag	LDIV	(l _{ns}) Rd<(l _{ns}) Rs
LMUL	flag	SBB	(unsigned) Rd<(unsigned) Rs
ADD	carry	ADC	carry

simplify the hardware, there is only a single bit of status, known as the flag. As shown above, this is used in the Subtract with Borrow (SBB) and Add with Carry (ADC) instructions, analogous to the role of the carry flag in the Pentium. In addition, unary operations include LNS square root, LNS reciprocal and LNS absolute value. The only unit which changes the flag is the ALU. Having a single flag simplifies the design of the branch unit, but this flag is overloaded with several different meanings, as shown in Table 3.

The MOV and the logarithmic multiply (LMUL) instructions leave the flag alone. The exclusive OR (XOR) instruction performs equal-to comparisons for all data types. SUB performs two's-complement integer-less-than comparison. SBB performs unsigned-integer-less-than comparison. The logarithmic division LDIV instruction performs LNS less-than comparison. These comparisons are the by-product of the associated ALU operations. A compiler, by judicious selection of the instruction(s) before a branch, can cast the data as required. Thus, if (a<b) will generate SUB, SBB or LDIV prior to a branch depending on the type of the high-level language variables. The way in which MOVPC manipulates the flag is special. It clears the flag, but it also causes the current branch instruction, if any, to act as though the flag is currently zero. Thus MOVPC executing in parallel to a branch instruction provides a way to save a return address whilst unconditionally branching to a subprogram.

5.2 LNS operations

The layout of $chunk2$, which controls LNS operations, is:



The operations in this category have latency of more than one cycle. The LNS unit is fully pipelined, so several independent LNS operations may proceed in parallel. Thus, one can keep several partial sums in different stages of the pipeline, and only combine them at the end of a summation. Table 4 shows the variations of the LADD instructions,

Table 4. Addition Instructions

	Latency		Delay
	Inst.	Clock	Clock
LADD	6	6	1
LADDW	6	6	6
LADDQ	4	4/6	1
LADDQW	4	4/6	4/6

their latency (in instructions and cycles), and their delay (in cycles).

L_{SUB} is nearly identical. It simply complements Y_S first, thus we can focus on the LNS unit from the perspective of LADD. As described earlier, addition of numbers of different signs but of nearly equal magnitude ($z \approx 0$) is a slower operation. Say, for example, it takes six cycles, compared to four cycles otherwise. The LNS unit must provide enough pipeline stages (six) to cope with this most difficult case, even though in most cases two of these stages simply pass the result through unmodified.

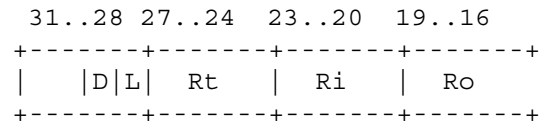
The W (wait) bit makes programming easier (and code more compact) in the case where a dependency demands the result of the operation be available before any other computation may proceed. The W bit stalls all units, including the integer ALU and the LNS unit, until the LNS unit completes the instruction that issued the W bit. In other words, the W bit makes the delay equal to the latency.

From a real-time perspective, the Q (quick) suffix makes latency (measured in clock cycles) of the LADDQ (and the delay if the W bit is also used) variable. When the specific operands require it, the latency will be six cycles, otherwise it will be four cycles. From the programmer's or compiler's perspective, the latency (measured in instructions) of the LADDQ is always four. Thus, the compiler can generate code under this simplifying assumption. The actual speed of a LADDQ program depends on the statistics of the data it processes. The more often the singularity region is encountered, the slower the effective speed of the LADDQ will be, but this is transparent to the software since the hardware will stall all units for two cycles when needed.

All of the instructions discussed up to this point are similar to those found in most conventional processors. Two other potential LNS instructions were the subject of experimentation: Logarithmic Multiply Accumulate (LMAC) and Logarithmic Increment and Multiply (LIM). The former is a logarithmic version of an instruction type commonly found on DSP processors considered useful for sum-of-products applications, such as FIR filters. The latter is a unique LNS instruction which has the potential to reduce the latency of short inner-loop computations. A discussion of these instructions is given in the Section 6.

5.3 Memory operations

The layout of chunk1, which controls the memory unit, is:



The only addressing mode supported is post-increment, thus the effective address is given by R_i which is later changed to $R_i \leftarrow R_i + R_o$. Two sizes of data transfer are supported: normal (32-bit, $D=0$) and double (64-bit, $D=1$). The latter is useful in vector and sum-of-product code, and involves a consecutive pair of registers. The L bit indicates a LOAD operation. The memory interface is assumed to be 64-bits wide, thus this instruction has the potential to execute in a single cycle. In the event the required data is not in the data cache, the entire processor stalls until the data is available. The unused bits $ir[31:30]$ could be used for a more sophisticated memory interface, such as pre-fetching into the cache, but this was not considered in the present processor.

The format for absolute branch and immediate values has $ir[31:0]$ as a simple 32-bit value. The microcode-like (absolute, single-flag) branches incur no penalty. Using the memory unit precludes the use of the branch unit and the meaningful use of immediate operands for the ALU or LNS units. These restrictions exist so no compressed instruction exceeds 64 bits.

6. Example Code

This section considers some candidate instructions for inclusion in the VLIW instructions set. We consider the cost of these instructions, and their usefulness in example code.

6.1 LMAC

One common computation is the sum of products. Suppose R1 and R2 contain the addresses of the length-eight vectors x and y respectively, R3 contains 2 (the number of elements that LOADD can obtain in one cycle) and that R10 has the LNS representation of 0.0. Then we can use two load-double instructions to get the first two elements of each vector into registers:

```

1*  NOP
    LOADD R6, R2, R3    R6=y[0]; R7=y[1]
2*  NOP
    LOADD R4, R1, R3    R4=x[0]; R5=x[1]

```

An asterisk indicates the chunk3 required for every VLIW instruction. Code between the asterisks corresponds to the operations that issue in parallel during one cycle. Since each instruction above does not use the ALU, chunk3 must be NOP (which is coded as MOV R0, R0). The clock-cycle number preceding the asterisk assumes that the pipeline does not need to stall for cache misses or singularity computations, but does consider deterministic stalls for the W bit (equivalent to NOPs if the W bit were not used).

In one cycle, a LOADD gets the next two elements from y; in the next cycle, another LOADD gets the corresponding elements from x. This pattern will repeat in later instructions until all elements of the vector have been accessed. Continuing on, let's consider how to produce the sum of products using LMACQ instructions:

```

3*  NOP
    LMACQ R10, R4, R6   R10a=0.0+y[0]*x[0]
    LOADD R8, R2, R3   R8=y[2]; R9=y[3]
4*  NOP
    LMACQ R10, R5, R7   R10b=0.0+y[1]*x[1]
    LOADD R4, R1, R3   R4=x[2]; R5=x[3]
5*  NOP
    LMACQ R10, R4, R8   R10c=0.0+y[2]*x[2]
    LOADD R6, R2, R3   R6=y[4]; R7=y[5]
6*  NOP
    LMACQ R10, R5, R9   R10d=0.0+y[3]*x[3]
    LOADD R4, R1, R3   R4=x[4]; R5=x[5]

```

Assume the latency of this hypothetical LMACQ is five instructions (one more than LADDQ to allow for the extra fixed-point adder). Thus, there can be up to five distinct values in the pipeline destined to be loaded into the R10 register, but for optimum efficiency, we only use four. In the comments, we will refer to these four values as R10a, R10b, R10c and R10d, but in the assembly language these are all called simply R10. These are defined by LMACQ instructions that add 0.0 to a product. Using LMACQ may seem an inefficient way to accomplish this, since a simple LMUL could compute these products, but LMACQ is necessary to set up the proper pipeline schedule in cycles 7-11. R10 obtains a new value four cycles later than the corresponding LMACQ. Thus, R10 will have R10a in cycle 7, R10b in cycle 8, etc. The following code, which continues the summation, must use these values at just the right moment:

```

7*  NOP
    LMACQ R10, R4, R6   R10a+=x[4]*y[4]
    LOADD R8, R2, R3   R8=y[6]; R9=y[7]
8*  NOP
    LMACQ R10, R5, R7   R10b+=x[5]*y[5]
    LOADD R4, R1, R3   R4=x[6]; R5=x[7]
9*  NOP
    LMACQ R10, R4, R8   R10c+=x[6]*y[6]

```

```

10* NOP
    LMACQ R10, R5, R9   R10d+=x[7]*y[7]
11* NOP
    latency of five

```

If the vectors were longer, cycles 9 and 10 would have had similar LOADD instructions, and the pattern would continue in cycles 11-14. Since having vectors of length eight means cycle 8 is the last cycle for LOADD, cycles 12 onward simply have to combine the partial sums (R10a, R10b, R10c and R10d) into the final answer. This can be done most easily with LADDQ and LADDQW:

```

12* MOV R11, R10       R11=R10a
13* NOP
    LADDQ R11, R10, R11  R11a=R10b+R10a
14* MOV R11, R10       R11=R10c
15* NOP
    LADDQW R10, R10, R11 R10=R10d+R10c
19* NOP
    LADDQW R10, R10, R11
                                R10=R11a+R10c+R10d
                                =R10a+R10b+R10c+R10d

```

R11 does not get R11a until cycle 17, so the value of R11 in cycle 15 is R10c. The processor is waiting in cycles 16-18 and 20-22, thus the final sum becomes available in cycle 23.

LMACQ would be the most complex instruction in the machine. LMACQ would require an extra fixed-point adder, but that is very small in comparison to the existing LNS pipeline. Previous LNS architects have used the insignificance of this cost to justify similar structures [12, 9]; however, we need to look at the larger context and the hidden costs. The register file would require an extra read port since Ra would not otherwise be used as an input operand to the LNS unit. Also, as mentioned above, the latencies of LMAC and LMACQ would be a cycle longer than that of LADD and LADDQ respectively, thus we need a pipeline depth of seven instead of six.

6.2 LMUL/LADDQ

To show that LMACQ is not worth its cost, consider how the above program could be modified to replace all the LMACQs with a combination of LMULs and LADDQs:

```

3*  LMUL R6, R4         R6=y[0]*x[0]
    LOADD R8, R2, R3   R8=y[2]; R9=y[3]
4*  LMUL R7, R5         R7=y[1]*x[1]
    LADDQ R10, R10, R6 R10a=0.0+y[0]*x[0]
    LOADD R4, R1, R3   R4=x[2]; R5=x[3]
5*  LMUL R8, R4         R8=y[2]*x[2]
    LADDQ R10, R10, R7 R10b=0.0+y[1]*x[1]
    LOADD R6, R2, R3   R6=y[4]; R7=y[5]
6*  LMUL R9, R5         R9=y[3]*x[3]
    LADDQ R10, R10, R8 R10c=0.0+y[2]*x[2]

```

```

    LOADD R4,R1,R3    R4=x[4]; R5=x[5]
7* LMUL  R6,R4       R6=y[4]*x[4]
    LADDQ R10,R10,R9 R10d=0.0+y[3]*x[3]
    LOADD R8,R2,R3    R8=y[6]; R9=y[7]
8* LMUL  R7,R5       R7=y[5]*x[5]
    LADDQ R10,R10,R6 R10a+=x[4]*y[4]
    LOADD R4,R1,R3    R4=x[6]; R5=x[7]
9* LMUL  R8,R4       R8=y[6]*x[6]
    LADDQ R10,R10,R7 R10b+=x[5]*y[5]
10*LMUL  R9,R5       R9=y[7]*x[7]
    LADDQ R10,R10,R8 R10c+=x[6]*y[6]
11*NOP
    LADDQ R10,R10,R9 R10d+=x[7]*y[7]

```

Only cycles 3-11 are different here. The sum appears in cycle 23. This is 0.83 FLOPs/cycle (which improves as vector lengths increase). The LADDQ version requires no more cycles than the LMACQ version. The many NOPs in the LMACQ version were replaced with LMULs. Thus, the same computation can be achieved in the same time without the LMACQs, thereby lowering register-file and pipeline costs.

6.3 LIM

In tight inner-loop computations the latency of the LNS unit may determine the number of cycles per iteration. With LADDQ, this latency is four. To improve on this, consider the Logarithmic Increment and Multiply Quick (LIMQ) instruction, where Ra becomes $(1.0 + (\lns)Rb) * (\lns)Rc$. By looking at the dashed-line LNS adder in Figure 2, we see LIM and LIMQ can be implemented at no appreciable cost (just multiplexors) by skipping the first stage (“LNS divide/integer sub”) of the pipeline; thus its latency is three, rather than four. No previous architecture, either LNS or conventional, has offered such an instruction. We propose that this uniquely LNS instruction is a better way to capitalise on the low cost of logarithmic multiplication/division than previous approaches (like Figure 2 or LMAC).

First, we note that this instruction is sufficient to implement all LNS addition/subtraction by substituting $Rb = X_L - Y_L$ and $Rc = Y_L$ in (1). Then we expand on the observation of Stouraitis [10] that $x_0 + x_1 + x_2 = (x_0/x_1 + 1) \cdot x_1 + x_2 = ((x_0/x_1 + 1) \cdot x_1/x_2 + 1) \cdot x_2$ and note these computations are in the form suitable for this instruction. Assume R6 has the vector length minus 3, R4 has the integer 2, R3 has the integer 1 and R2 has the address of x. The following code uses LIMQ to compute the sum of the elements of x:

```

1* NOP
    LOADD R10,R2,R4    R10=x[0]; R11=x[1]
2* LDIV  R10,R11      R10=x[0]/x[1]

```

```

    LOAD  R8,R2,R3    R8=x[2]
3* LDIV  R11,R8      R11=x[1]/x[2]
4* L2:
    ADD   R6,-1      i++;R11=x[i-1]/x[i]
5* MOV   R11,R8      R11=R8=x[i];
    LIMQ  R10,R10,R11 R10=(1+R10)*R11
    LOAD  R8,R2,R3    R8=x[i+1]
6* LDIV  R11,R8      R11=x[i]/x[i+1]
    BF    L2          branch if R6!=0
7* NOP
    MOV   R11,R8      R11=R8=x[n-1];
    LIMQW R10,R10,R11 R10=(1+R10)*R11
11*NOP
    LIMQW R10,R10,R11 R10=x[0]+...+x[n-1]

```

The cycle numbers shown above are for a vector of length 4. This simple, easy-to-compile loop will take only 3 additional cycles for each additional element summed. This is clearly more efficient than a similar single-LADDQ-per-iteration loop. In comparison to a fully-pipelined and unrolled (branchless) loop (like the LADDQ-style summation of the previous example), the overhead here is 10 cycles (cycles 1-3 and 7-13) outside the loop, compared to 11 in the last section. Thus, the LIMQ loop is faster than the fully unrolled LADDQ computation when $3 \cdot (n-3) + 10 \leq n + 11$, or $n \leq 5$. Therefore, we conclude that our novel LIMQ instruction, which has no appreciable cost, offers benefit for a common programming situation: short inner-loop summation.

6.4 ROMLOG

One objection raised against LNS processors is that their non-standard data format requires costly input conversion. To overcome this, we propose a ROMLOG instruction, which in a single cycle can convert the 11-bit input from an Analog-to-Digital converter to LNS.

The co-transformations that help solve the subtraction singularity problem [1, 2, 3] need a ROM of $\log_b(1 - b^z)$ for z near zero. Although it has been shown previously [1] that in this region $\log_b(1 - b^z) \approx \log_b |z| - \log_b(\log_b(e)) + z/2 + z^2 \cdot \log_e(b)/24 + \dots$, no earlier researcher has noted that this relationship can be turned around and used to assist with input conversion. When z is limited to the low-order 11 bits of the 32-bit format considered here, the quadratic term is insignificant and $\log_b |z| \approx \log_b(1 - b^z) + \log_b(\log_b(e)) - z/2$. It is easy to adapt the co-transformation to work with a ROM containing $R(z) = \log_b(1 - b^z) + \log_b(\log_b(e))$ instead of $\log_b(1 - b^z)$; thus by using this same ROM, the ROMLOG instruction can be implemented as $Ra <- R(Rb) - Rb/2$, which fits in a single cycle. If we need more accuracy than 11 bits, we can apply (1) and use ROMLOG instructions to convert a 32-bit unsigned integer in R1 to its LNS equivalent in R6:

```

1* ROR    R1,11    R1=mid 11 bits
   ROMLOG R2,R1    R2=log(low 11 bits)
2* LDIV   R2,2048.0 scale R2
   ROMLOG R3,R1    R3=log(mid 11 bits)
3* ROR    R1,11    R1=high 10 bits
   LADDQ  R4,R3,R2 R4=(lns)R3+(lns)R2
4* AND    R1,0x3ff mask
5* NOP
   ROMLOG R5,R1    R5=log(high 10 bits)
6* NOP
7* LDIV   R4,2048.0 scale R4
8* NOP
   LADDQW R6,R5,R4 R6=(lns)R5+(lns)R4
12* LMUL  R6,2048.0^2 scale result

```

Since this novel rearrangement of the co-transformation ROM costs essentially nothing, we conclude this useful ROMLOG instruction is worth implementing.

7. Conclusions

This paper describes a new VLIW instruction set suitable for a 32-bit LNS microprocessor. Unlike earlier LNS architectures [2, 3, 9, 12], the proposed ISA does not provide more LNS multiplication unit(s) (admittedly cheap in LNS) than LNS addition unit(s). This paper shows the hidden costs to the total architecture of such an approach. Also, the proposed ISA departs from an earlier VLIW proposal [7] of orthogonal LNS ALUs (each capable of LNS addition and multiplication) in order to reduce implementation cost.

Novel features of the proposed architecture include a parallel unit for single-cycle operations (like LNS multiply and integer add), another unit for multi-cycle operations (like LNS addition), and a compressed instruction format which is expanded to a VLIW format for execution.

This paper shows it is worthwhile to include a novel logarithmic increment-and-multiply instruction and a novel logarithmic conversion instruction in the instruction set to capitalise on the advantages of LNS arithmetic. Both of these improvements can be implemented at very low cost. Also this paper shows, somewhat surprisingly, that there is no benefit that justifies the cost of implementing a logarithmic multiply-accumulate instruction.

References

- [1] M. G. Arnold, T. A. Bailey, J. R. Cowles and M. D. Winkel, "Arithmetic Co-transformations in the Real and Complex Logarithmic Number Systems," *IEEE Trans. Comput.*, vol. 47, no. 7, pp. 777-786, July 1998.
- [2] E. I. Chester and J. N. Coleman, "Matrix Engine for Signal Processing Applications Using the Logarithmic Number System," Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, San Jose, CA, pp. 315-324, 17-19 July 2002.
- [3] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl and A. Hermanek, "The European Logarithmic Microprocessor - a QR RLS Application," *35th IEEE Asilomar Conference on Signals, Systems and Computers*, p. 155-159, Pacific Grove, California, October 2001.
- [4] M. Kahrs and K. Branderburg, eds., *Applications of Digital Signal Processing to Audio and Acoustics*, Kluwer Academic Publishers, Norwell, MA, p. 224, 1998.
- [5] D. M. Lewis, "114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications," *Intl. Solid-State Circuits Conf.*, San Francisco, pp. 86-87, Feb. 1995.
- [6] Junichiro Makino and Makoto Taiji, *Scientific Simulations with Special-Purpose Computers—the GRAPE Systems*, John Wiley and Sons, Chichester, 1998.
- [7] V. Paliouras, et al., "A Very-Long Instruction Word Digital Signal Processor Based on the Logarithmic Number System," *Proc. of the 5th Intl. Conf. On Electronics, Circuits and Systems*, vol. 3, pp. 59-62, 1998.
- [8] V. Paliouras and T. Stouraitis, "Logarithmic Number System for Low-Power Arithmetic," *PATMOS 2000: Intl. Workshop on Power and Timing Modeling, Optimization and Simulation*, Gottingen, Germany, pp. 285-294, 13-15 September, 2000.
- [9] J. Skog, O. Vainio and J. Nurmi, "Processor Architecture for Logarithmic Signal Processing," *Proc. TUT Symp. on Signal Processing '94, Tampere, Finland*, 20 May 1994.
- [10] T. Stouraitis, *Number System Theory, Analysis, and Design*, PhD Dissertation, University of Florida, Gainesville, 1986.
- [11] E. E. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Trans. Comput.*, vol. C-24, pp. 1238-1242, Dec 1975.
- [12] F. J. Taylor, R. Gill, J. Joseph and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. Comput.*, vol. C-37, pp. 190-199, 1988.

Figure 1. LNS Interpolator used in Figures 2-4.

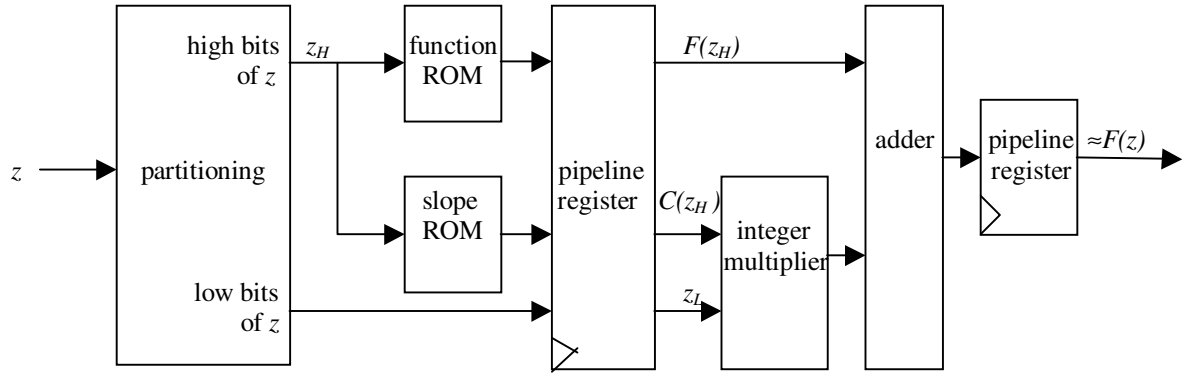


Figure 2. Prior 2-multiplier LNS architecture [3,12,16].

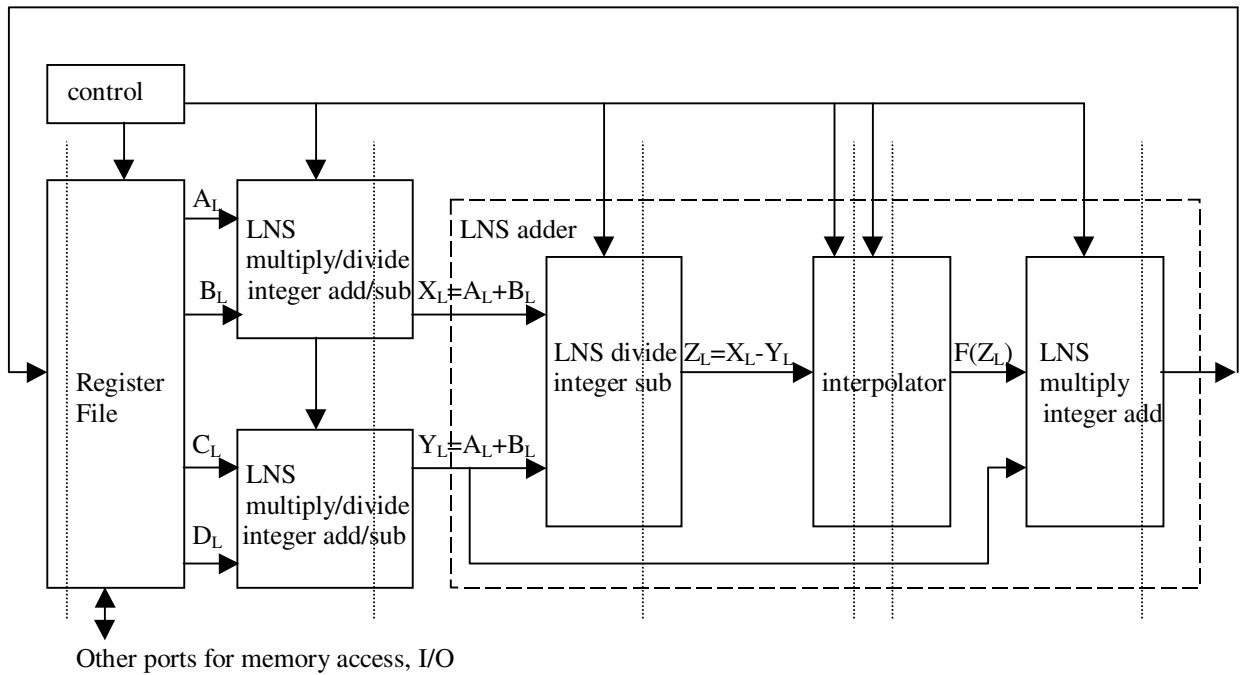


Figure 3. Prior 2-ALU LNS Architecture[8].

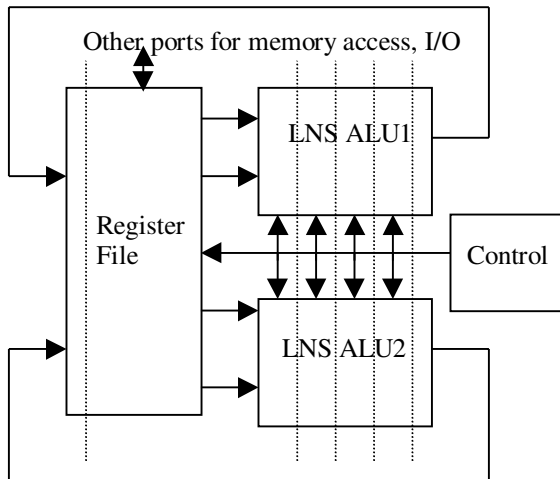


Figure 4. Proposed Asymmetrical LNS Architecture

