PIPELINED LOGARITHMIC 32BIT ALU FOR CELOXICA DK1

A. Heřmánek, J. Kadlec, R. Matoušek, M. Líčko, Z. Pohl

UTIA Prague, Czech Republic

Abstract. This paper presents and compares two possible solution for floating point-like HW, based on a 32-bit logarithmic ALU. We describe the implementation, parameters and the basic use of a non-pipelined ALU and a 3stage-pipelined ALU. Both Virtex FPGA cores are encapsulated in function-like API interface compatible with the Handel C 2.1 and the new DK1 tool from Celoxica. DSP designers can create optimized VLIW program flow with 32bit FP data range and precision. Code can be source-code-debugged and subsequently compiled from this high-level to the target Virtex FPGA.

Introduction

The complexity of the IEEE floating point implementation negatively affects the use of advanced DSP and control algorithms in FPGA. We present two 32-bit logarithmic ALUs for Celoxica DK1 implementation path and Virtex FPGA. Cores have been implemented in the *non-pipelined* and in the *3-stage-pipelined* versions. Both cores implement all basic floating-point-like 32-bit logarithmic operations by representation of floating point numbers as 32-bit integer (fixed point) logarithms [1].

The logarithmic number system (LNS) is well suited to the FPGA environment. The core takes just 8% of the XILINX Virtex XCV2000E-6 device. Both cores operates at 53MHz and implement all the basic operations of logarithmic arithmetic (ADD, SUB, MUL, DIV and SQRT), with the covered data range and the precision equal to or better than the standard IEEE 32-bit floating point used in new DSPs. See [3], [4], [5] for details.

We consider these cores as possible candidates for upcoming advanced embedded DSP applications based often on orthogonal rotations (QR RLS, LATTICE, SVD).
See [2], [7] for details. This research is performed under the EU ESPRIT 33544 HSLA Long-term research project, coordinated by the University of Newcastle [1].

The cores are complemented with a Matlab library [5] emulating bit-exactly the properties of the final hardware. The library can be used in the following environments:

- Directly in Matlab M-scripts,
- C language via MEX functions,
- Simulink, via S-functions,
- and thereby in Real Time Workshop for rapid prototyping

Underlying Algorithms

In the presented Logarithmic Number System (LNS), a real number x is represented as a 31-bit signed two's compmenent fixed point value representing $\log_2(|x|)$ with an additional 32nd bit to indicate its sign. LNS multiplication, division and square-root can thus be implemented rapidly as integer addition, subtraction and right-shift (and round) respectively.

Addition and subtraction of values in the LNS present a challenge. Our solution is based on these formulae:

$$z = a + b \qquad \textbf{(1)}$$

$$\log(z) = \log(a) + \log(1 + 2^{\log(b) - \log(a)})$$

$$z = a - b \quad (2)$$
$$\log(z) = \log(a) + \log(1 - 2^{\log(b)-\log(a)})$$

The functions $\log(1 + 2^{\log(b)-\log(a)})$ and $\log(1 - 2^{\log(b)-\log(a)})$ are awkward transcendental functions which we must somehow evalute. The approximation of these functions is the principal problem in LNS research. We employ an innovative, patented solution developed by the HSLA project team under Dr. Coleman [1],[3], which yields a drastic reduction in the size of the look-up tables required compared to those needed for conventional linear interpolation of both functions. This is achieved by the parallel evaluation of a linear approximant and an error correction term.

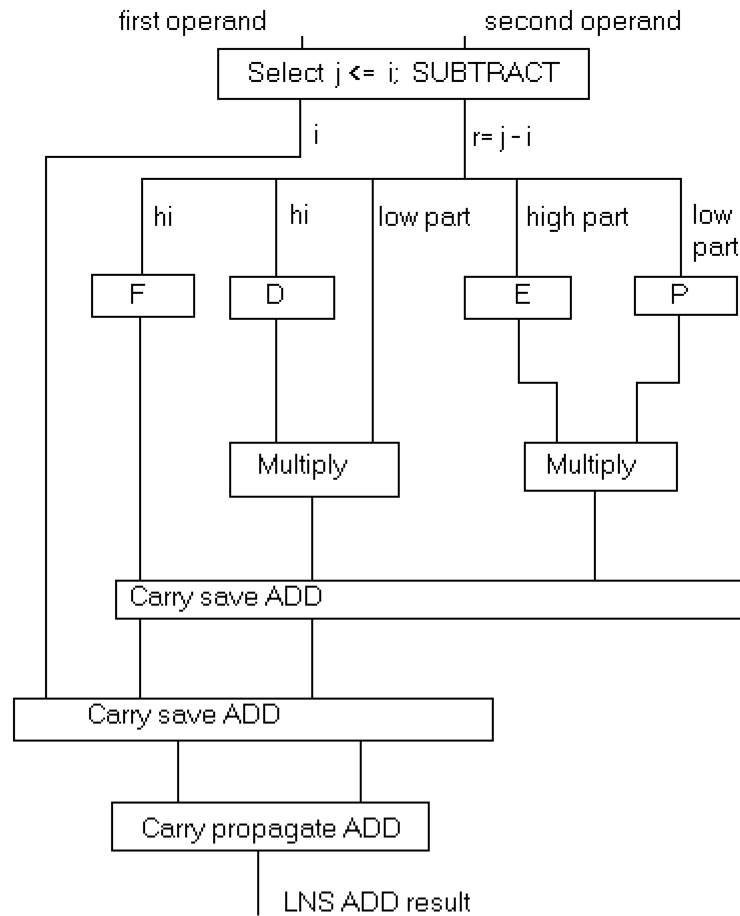

Fig. 1*: Coleman's approximation method based on parallel access to 2 sets of look-up tables. The tables F and D serve for the linear interpolation. The idea behind Coleman's method is the special (patented) selection of the approximation intervals. It results in an identical correction term for each of the approximation sub-intervals. This enables parallel computation of the correction term from a single set of E and P tables and drastic reduction of the look-up table size for 32-bit precision.*

Coleman's approach [1], [3] leads to a solution which is suitable for the FPGA implementation because it avoids the need for a barrel shifter. Implementation of the hardware equivalent of barrel shifts is area-costly and ineffective in FPGA; that is one reason why FPGA design does not use floating point arithmetic a lot. The LNS ALU provides one of the first hardware solutions to this problem (see Fig.1).

The idea behind Coleman's method is the special (patented) selection of the approximation intervals. It results in an identical correction term for each of the approximation sub-intervals. This enables parallel computation of the correction term from a single set of E and P tables and drastic reduction of the look-up table size for 32-bit precision.

IEEE single precision:

31 30 23 22 0

32b LNS:

31 30 23 22 0

Elementary LNS operations:		
x + y	ADD	Lz=Lx+log(1+2^(Ly-Lx)), Sz depends on sizes of x, and y
x - y	SUB	Lz=Lx+log(1- 2^(Ly-Lx)), Sz depends on sizes of x and y
x * y	MUL	Lz=Lx+Ly, Sz=Sx OR Sy
x / y	DIV	Lz =Lx-Ly, Sz=Sx OR Sy
x^0.5	SQRT	Lx >> 1, Sz=Sx

Fig. 2: *IEEE 32-bit floating-point format, LNS 32-bit format and the internal implementation of the elementary LNS operations applied to the 32-bit integer representation of the logarithm.*

The 32-bit LNS addition can be performed in time comparable to floating-point 32-bit addition without loss of precision. It needs approximately 32k bytes of look-up table space with 32-bit word-length. Internally the LNS ALU operates on an extended 35-bit wide data representation, enabling the final results to be rounded to within the error bounds of 32-bit floating point. Therefore, the LNS ALU is a valid candidate for the development of the IP Cores for the FPGA based designs, which need to operate with the range and precision of 32-bit floating point numbers.

Internal and External Data Representation

Log data are represented in a 32-bit format. The MSB indicates the sign of the corresponding real number. MSB=0: positive real number. MSB=1: negative real number. The remaining 31 bits hold the base-two logarithm of the real value being represented, in two's complement format (Fig. 2). In Matlab, the conversion into this format could be implemented as: z=round(8388608*log2(abs(u)));

- The maximum value of a real number which can be represented is $u=3.4\times10^{38}$
- The minimum value of a real number which can be represented is $u=(1/3.4)\times10^{-38}$
- LNS zero is represented by special code `0x40000000`.
- The result of division by zero is indicated by a `NaN 0xC0000000`.

Algorithms for conversion to and from the log. domain are provided. Real-time applications need these conversion in hardware and this is done by the int2log() and log2int() Handel C and DK1 modules. These modules support conversion of real-domain fixed-point data with up to 22 bit precision in the range (-1,1). The conversion algorithms are designed to support up to 22-bit precision A/D and D/A devices.

Description of implemented functions

The following 32bit precision LNS operations are supported:

Matlab	*ANSI C*	*Hancel C*	*Description*
lm2()	logmul()	lmul()	Saturating multiply with full status handling (overflow, etc).
-	-	lm()	Fast regular multiply - can execute in parallel with additions and subtractions.
ld2()	logdiv()	ldiv()	Saturating divide with full status handling (overflow, etc).
-	-	ld()	Fast regular divide - can execute in parallel with additions and subtractions.
lsq2()	logsqrt()	lsqrt()	Saturating square-root with full status handling (overflow, etc).
-	-	lsq()	Fast regular divide - can execute in parallel with additions and subtractions.

Addition and subtraction supported by non-pipelined ALU:

Matlab	*ANSI C*	*Hancel C*	*Description*
la2()	logadd()	ladd()	Saturating add with full status handling (overflow, etc).
ls2()	logsub()	lsub()	Saturating subtract with full status handling (overflow, etc).

Add and subtract require a total of 64 kilobytes of LUT storage, organised as 4×4Kword (32-bit) SRAMs.

Addition and subtraction supported by 3-stage pipelined ALU:

Matlab	*ANSI C*	*Hancel C*	*Description*
la2p()	-	ladd()	Start of saturating add with full status handling (overflow, etc).
ls2p()	-	lsub()	Saturating subtract with full status handling (overflow, etc).
result()	-	result()	Read result of 3-stage pipelined saturating add, subtract.

Add and subtract require a total of 64 kilobytes of LUT storage, organised as 4×4Kword (32-bit) SRAMs.

Conversion routines:

Matlab	*ANSI C*	*Hancel C*	*Description*
d2log	-	-	Conversion from decadic to logarithmic domain – double precission format
log2d	-	-	Conversion from logarithmic to decadic domain – double pricision format
int2log2	-	int2log	22-bit precision conversion from integer to logaritmic domain (approx.)
log2int2	-	log2int	22-bit precision conversion from logarithmic to integer domain (approx.)

The parallel execution capability of the first group of operations (lm,ld,lsq) allows them to proceed concurrently with logarithmic addition or subtraction. Since these are hardware macros, the only limit on the number of such operations that can be executed at once is FPGA capacity.

How to use LNS routines:

In this section we will show how to use our LNS routines. Declarations and source codes for diferent enviroments (Matlab, ANSII C, Handel C) will be presented.

EXAMPLE CODE in Matlab, using HSLA emulation library:

```
for k=1:8,            % la2(),lsq2(),lm2() are MEX
      z(k) = la2(a,b);        % the Matlab HSLA library in C
      y(j+1)= lsq2(x(j));     % bit-exact emulation of the
      r(k) = lm2(a,b);        % In Matlab: arrays start with index 1
      j= j + 1;               % In C and Handel C: arrays start from 0
      x(i) = lm2(r(k),y(j));
    end;
```

All variables for LNS calculations (a,b.. etc.) are standard double precision Matlab variables and they had to be converted to logartihmic domain using d2log(), log2d() routines befor. For arrays use d2logM(), log2dM() functions. The M at the end of function name represents matrix function. In Matlab matrix mersions of additon, subtraction, multiplication and division also exist.

Corresponding DECLARATIONS in the standard ANSI C used in floating point DSP:

```
float      a, b, x[8], y[8], z[8], r[8];
int        i, j, k;
```

Corresponding CODE of the standard ANSI C used in floating point DSP:

```
for(k=0;k<8;k=k+1){
  z[k] = a + b;           // z[k], j[j+1] and r{k] can be
  y[j+1]= sqrt(x[j]);     // exec. Indep.(can be in par)
  r[k] = a * b;           // see Handel C version
  j= j + 1;               // integer operation
  x[i] = r[k] * y[j]);    // x[i] needs j incremented
}
```

Corresponding DECLARATIONS in Handel C and the Celoxica DK1 tool:

```
Int 32     a,b;                          //(int 32...LNS)
ram int    32 x[8],y[8],z[8],r[8];       //(size=2^3)
unsigned   3  i,j,k;                     //relates to size 2^3
unsigned   3  zsl;                       //status of LNS op.
```

Corresponding parallel CODE for non-pipelined ALU in Handel C and DK1:

```
for(k=0;k<8;k=k+1){
par{                         // 3 parallel threads
      ladd(a,b,z[k],zsl);
      y[j+1] = lsq(x[j]);
      {                      r[k] = lm(a,b);  // sequential exec.
         j = j + 1;          // integer HW
         x[i] = lm(r[k],y[j]);
      }
    }
}
```

TIMING of parallel computations non-pipelined ALU:

Clock	t	t+1	t+2	t+3	...	t+8	t+9	t+10	t+11
ladd	■	■	■	■	■	■	■	■	■
lsq	■						■		
lm	■		■				■		■
j++		■						■	
k =	0	0	0	0	0	0*	1**	1	1

* - z[0] done, ** - next loop

FPGA Implementation Summary

Both LNS IP-cores has been tested at the clock rates up-to 53 MHz XCV2000E-6. Non-pipelined core consumes **8%** and the pipelined core **11%** of XCV2000E-6 slices. None of the Virtex internal block-RAM are used in both implementations.

The look-up tables are located in the four external banks of SRAM. Tables take 4Kwords in each of four 32bit-wide SRAMs on the RC1000 board. The tables are DMA pre-booted via the PCI interface of the board. Each SRAM has 512Kwords of SRAM. 508Kwords remain free for the user.

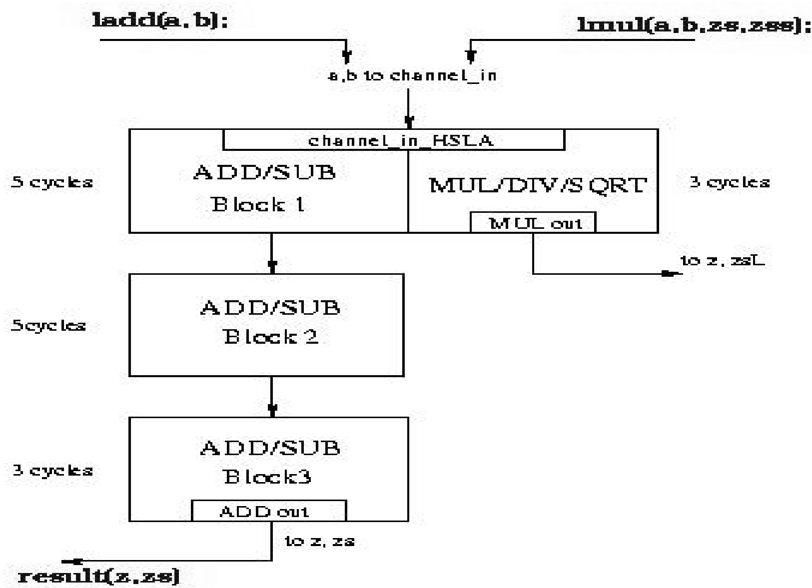

Fig. 3. *3-stage pipelined ALU. The use of MUL/DIV/SQRT can be used for execution of operations in parallel to the ADD/SUB in stage 2 or 3 of the pipe. The example of log-to-int conversion is using the dedicated 1-cycle macros at the cost of the additional distributed hardware.*

Implemented API for Celoxica DK1 API

All implemented function for Celoxica DK1 API and their clock requirements are presented in the next table:

Function	**Description**	**Cycles**	
		Non-pipelined ALU	**Pipelined ALU**
`ladd(al,bl,zl,zsl)`	`zl=al+bl;`	9-12	
`lsub(al,bl,zl,zsl)`	`zl=al-bl;`	9-12	
`ladd(al,bl)`	`pipe al+bl;`	-	5(*)
`lsub(al,bl)`	`pipe al-bl;`	-	5(*)
`result(zl,zs)`	`result from`	-	1
`zl=ld(al,bl)`	`zl=al/bl;`	1	1
`zl=lsq(al)`	`zl=sqrt(al);`	1	1
`int2log(al,zl)`	`int to log`	45-60	27(**)
`log2int(al,zl)`	`log to int`	45-60	27(**)

(*) result can be collected after 5+5+3 cycles
(**) 27 cycles in the case of multiple conversions in parallel. See Example 1.

FPGA implementation of the 22bit precision LOG to INT conversion macro

Standard polynomial approximation is used for the conversion with the implementation outlined in the equation (3).

$$d_{int} = k_0 + k_1 d_{log} + k_2 d_{log}^2 + k_3 d_{log}^3 + k_4 d_{log}^4 + k_5 d_{log}^5 \quad (3)$$
$$d_{int} = k_0 + (k_1 + (k_2 + (k_3 + (k_4 + k_5 d_{log}) d_{log}) d_{log}) d_{log}) d_{log}$$

See Example 1. and Exapmle 2. for the implementation examples. Both examples provide conversion of 3 numbers (block of length 3). This is needed for the pipelined version of convrersion in Example 2. to get the maximal performance. Input of the conversion (dLog[i]) is the LNS 32 bit value, representing the data to be converted in the range (0,1). Output of the algorithm is the 32 bit variable dInt[i], which in lower 24bits holds the integer equivalent. Bits [23:2] can be used to drive up to 22-bit precision D/A device.

```
for (i=0;i<3;i++){
  ladd(k4,lm(k5,dLog[i]) ,zl1,zs);
  ladd(k3,lm(dLog,zl1),zl2,zs);
  ladd(k2,lm(dLog,zl2),zl3,zs);        //3 numbers converted in:
  ladd(k1,lm(dLog,zl3),zl4,zs);        //best case : 3*5*9=135 cycles
  ladd(k0,lm(dLog,zl4),dInt[i],zs);    //worst case: 3*5*12=180 cycles
}
```

Example 1. Implementation of log to integer conversion of 3 length vector by the non-pipelined log ALU.

```
ladd(k[4],lm(k[5],dLog[i]));             // 3*5+1=16 cycles
ladd(k[4],lm(k[5],dLog[i+1]));
ladd(k[4],lm(k[5],dLog[i+2]));
result(zl1,  zs1);
for (j=3;j>=0;j--){                     // 4*3*5=60 cycles
  par{ladd(k[j],lm(dLog[i]   ,zl1));result(zl2,zs2);}
  par{ladd(k[j],lm(dLog[i+1],zl2));result(zl3,zs3);}
  par{ladd(k[j],lm(dLog[i+2],zl3));result(zl1,zs1);}
}
par{dInt[i]=zl1; zs[i]=zs1;}            // 3 cycles
result(dInt[i+1],zs[i+1]);              //3 numbers converted in:
result(dInt[i+2],zs[i+2]);              // total 79 cycles
```

Example 2. Implementation of log to integer conversion of 3 length vector by the 3-stage pipelined log ALU.

The real HW macro `log2int()` which is part of the final API for the Celoxica DK1 core is in addition handling the conversion of the data in the range (-1,1) at no additional precision or time cost. The corresponding logic has been removed for the clarity. Example 2. demonstrates the use of Handel C par{} constructs, and utilization of the pipelined ALU.

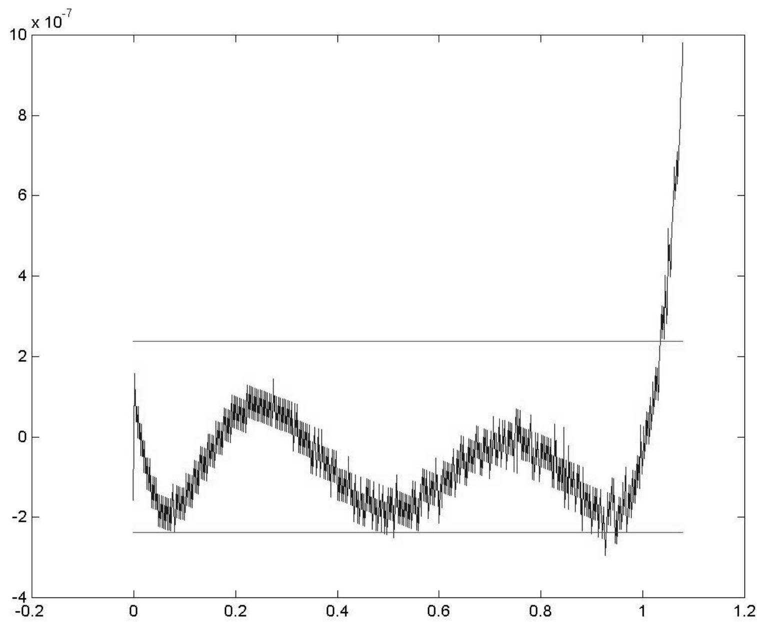

Fig. 4. *Absolute precision of the log. to real domain (fixed-point) conversion algorithm. Horizontal lines indicate the precision margins required for the support of a 22bit D/A output convertor.*

Performance comparison for the log to int example

If both ALUs operate at 50MHz, data can be converted at these sampling rates:

- At least to **833kHz** (worst case, corresponding to **8.3Mflop** performance) for non-pipelined ALU and

- At **1.85MHz** for the 3-stage pipelined ALU. This is corresponding to **18.5Mflop** performance. The pipelined ALU provides this performance only if the data can be processed in batches of at least 3 measurements.

In general, both ALUs implement the DSP or control algorithms with the 32-bit log. data type described above. The conversion to and from the corresponding real-domain fixed-point representation is performed only for I/O.

FPGA Implementation Path

Both LNS ALU cores have been implemented in FPGA by Handel-C 2.1 and Celoxica DK1. The verrified performance (53MHz clockfor XCV2000E-6 on RC1000 board) has been achieved by the use of the DK1 and this path:

1. Celoxica DK1 (using the Handel C2.1 compatible code) with export to VHDL.
2. Synplify 5.3 from Synplicity to create EDIF.
3. XILINX Alliance 3.3i tools to place and route from the EDIF netlist for the FPGAs.
4. Download to FPGA in the target platform. The Celoxica RC1000 board [11] has been used as the target platform in our case. The board is booted and interfaced to the PC by a straightforward C or C++ application compiled by MS VC 6.0.

Fast simulation support at the high algorithmic level in Matlab or DK1

Debugging of the complex designs (simulation time) becomes huge pronlem in the case of multi-million gate FPGAs. It becomes specially importentant in the case of the pipelined ALU. The presented API for Matlab and Celoxica DK1 might be one of possible solutions.

- **Matlab:** We provide bit-exact Matlab version (C coded DLL MEX functions) of pipelined `ladd(),lsub()` and `result()` specially for the emulation of the pipelined ALU. User can write the floating-point like part of the algorithm in Matlab as sequential code. `ladd()` and `result()` takes track and care about posible deadlocks related to the ALU pipe. Example: If the user would fill the pipe with a sequence of 4 subsequent `ladd()` or `sub()` operations without the `result()`after the 3-rd call, deadlock would happen. This is directly reported at the Matlab level simulation. Complete add/sub operation takes just 800 nanoseconds to execute on a 300MHz Pentium 2 PC. ***User is debugging at the high level with ZERO compile time, bit-exact, and at the speed of 10-20 Million clock cycles per second on a moderate PC.***

- **Celoxica DK1:** Similar set of DLL function plugs has been designed for of the pipelined `ladd(),lsub()` and `result()` for the accelerated high level emulation of the pipelined ALU under the Celoxica DK1 simulator. Again the deadlocks are detected and reported to the user. ***User is debugging complete parallel code with all threads and parallel executions with a substantially reduced simulation-related HandelC-to-Gates. Simulation remains bit-exact, and the speed of the ALU part is again 10-20 Million clock cycles per second before the long final P&R step.***

Both outlined methods are relatively intuitive are straightforward to use. Simulation at the Matlab level is typically used in the first stages of the port. Major concern is effective use of the pipelined ALU without creation of deadlocks. DK1 based simulation covers in addition the complete parallel environment within the FPGA coded in Handel C.

Conclusion

Both presented ALU provide efficient FPGA implementation of elementary operations (add, subtract, multiply, divide and square root). Presented Handel C2.1 and DK1 API cover the basic foundation for creation of advanced algorithms in hardware.

The realistic sustained (and easy to program) performance of the non-pipelined ALU is **10-15 Mflop** for XCV2000E-6 in the area of algorithms like the RLS QR as used in e.g. radar and control applications, QR-Lattice, and Normalized Lattice. Programming of the 3-stage pipelined ALU is not as straightforward. Depending on the algorithm one can count with the performance from **15** to **30 Mflop.**

The DK1 design path was found to be a friendly, robust and high-productivity tool.
In combination with the libraries presented here, the DSP algorithm designer creates, in effect, his own optimized VLIW-like (Very Large Instruction Word) and floating-point-like program flow, which can be source-code-debugged and subsequently straightforwardly compiled from this high-level to the target FPGA. Using the DK1 toolset and the HSLA libraries, rapid FPGA implementation of high performance DSP hardware can be achieved.

References

[1] J.N. Coleman, E.I.Chester, 'A 32-bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point', *14th Symposium on Computer Arithmetic',* Adelaide, April 1999

[2] Kadlec J., Matousek R., Vialatte C., Coleman N.: Port of Pascal FPGA-logarithmic-unit simulator to Simulink/RTW. In: Sbornik prispevku 7. rocniku
konference MATLAB '99. VSCHT, Praha 1999, pp. 84-90.

[3] J.N.Coleman, E.Chester, C.Softley and J.Kadlec "Arithmetic on the European Logarithmic Microprocessor", IEEE Trans. Comput. Special Edition on Computer Arithmetic, July 2000. Vol. 49, No. 7, p702-715.

[4] Coleman J. N., Kadlec J.: Extended Precision Logarithmic Arithmetics. In Proceedings of the 34-th IEEE Asilomar Conference on Signals, Systems and Computers, Monterey USA. November 2000.

[5] J. Kadlec, A. Hermanek, Ch.Softley, R. Matousek, M. Licko "32-bit Logarithmic ALU for Handel C 2.1 and Celoxica DK1 (53 MHz for XCV2000E-6 based RC1000 board)",Celoxica user conference, Stratford, UK, April, 2001. Download from: http://www.celoxica.com/programs/university/academic_papers.htm

[6] RC1000-PP Hardware Reference Manual, Celoxica UK.

[7] F.Albu, J. Kadlec, Ch. Softley, R. Matousek, A. Hermanek " Implementation of (Normalised) RLS Lattice on Virtex", submitted to FPL 2001, Belfast, Northern Ireland, August 2001.