HSLA Package ver. 3.0.0

Matlab HSLA Toolbox
32- and 19-bit TWIN LNS ALU

Jiří Kadlec[1]
Milan Tichý
Zdeněk Pohl
Antonín Heřmánek
Miroslav Líčko
Rudolf Matoušek
Jan Schier

May 29, 2002

[1]Department of Signal Processing, Institute of Information Theory and Automation, Pod vodárenskou věží 4, 182 08 Prague, Czech Republic, e-mail: kadlec@utia.cas.cz, tel.: +420-2-66052216

© 2001 Department of Signal Processing
All rights reserved. Published 2002

Department of Signal Processing
Institute of Information Theory and Automation
Pod vodarenskou vezi 4
P.O. Box 18
182 08 Praha 8
Czech Republic

Contents

List of Figures

List of Tables

Preface

The complexity of the standard IEEE floating point implementation negatively affects the use of advanced DSP and control algorithms in FPGA applications. A perspective solution is the *Logarithmic Number System* (LNS) which is well suited for the FPGA environment. The *High Speed Logarithmic Arithmetic* (HSLA) represents an attempt to implement the LNS in the FPGA technologies. All the basic operations of logarithmic arithmetic in the HSLA are implemented both with the covered data range and the precision equal to or better than the standard IEEE 32-bit floating point used in new DSP's.

This report describes the *Matlab HSLA Toolbox* and the *Logarithmic Arithmetic Unit* (LNS ALU) operating in the 32- or 19-bit precision environment. The toolbox consists of a set of *C* libraries and a set of Matlab *mex* functions. The *C* libraries enable users to develop their own applications. The *mex* functions provide basic arithmetic operations, i.e. addition, subtraction, multiplication, division, and square-root. All Matlab functions work with scalars, some of them can also operate on matrices. Two versions (32- and 19-bit precision) of LNS arithmetics are provided. All functions perform operations bit-exact to hardware level unless otherwise specified.

The Structure of the Document

This document is intended for a user who plans to use the HSLA libraries to write his own *C* applications or a user whose intention is to use the HSLA toolbox in the Matlab environment. The first part of document provides description of the *HSLA Toolbox*. The second one discusses basic aspects of the *LNS ALU* hardware implementation.

First, in the Chapter 1, the installation is briefly described and the notes on building the executable binary DLL's for Matlab and environment requirements are given.

In the Chapter 2, binary representation of logarithmic numbers and 19-bit and 32-bit format specifications are described. The HSLA toolbox supports standard and extended precision LNS arithmetics. These features are discussed in this chapter too.

Next, in the Chapter 3, the description of the functions contained in the *C* library is provided. The chapter is divided into two sections, the first one describing the functions operating on the standard precision LNS numbers (Section 3.1) the latter giving the description of functions operating on the extended precision LNS numbers (Section 3.2) mapped in the range $\langle -1; 1 \rangle$. The functions are sorted alphabetically in both sections.

The Matlab environment is very popular in the community of scientists. The *mex* functions are used to interface the *C* functions contained in the HSLA libraries in order to provide a comfortable way of using the LNS functions in Matlab. These functions are described in the Chapter 4 and also alphabetically sorted.

The Chapter 5 provides the notes on the LNS ALU installation. It describes the contents of the package and the environment settings required for LNS ALU examples.

The next Chapter 6 discusses provided LNS ALU operations, its use, and the *Celoxica DK1* targets that can be used for compiling the design using ALU.

In the Chapter 7 the implementation details of the LNS ALU modules are discussed. The module interfaces of the LNS ALU are provided in the header files described here.

In the Chapter 8 illustrative examples of using the LNS ALU are given. The description of *DK1 Handel-C* examples and other subsidiary files is provided.

In the last Chapter 9 the main features that should be included in the future versions of the HSLA toolbox are presented.

Part I

Matlab HSLA Toolbox

1 The Installation of the HSLA Toolbox

1.1 The Contents of the HSLA Toolbox

This chapter gives you a brief information on the contents of the HSLA toolbox directory. After unpacking the *lnsalu* package you will find the following directory structure in the `libhsla/` directory:

`include/`	*ANSI C* header files for 32- and 19-bit versions
`lib/`	*C* static libraries for 32- and 19-bit versions
`mex/`	Matlab *mex* source files
`runlib/`	runtime Matlab *mex* libraries (DLL) for 32- and 19-bit versions

The HSLA library has been designed to provide support for developing standalone *C* applications using the LNS arithmetics. The *C* header file `hsla_LNS.h` provides programming interface to the applications. It can be found in the `include/` directory. This directory contains also header files `defs32_LNS.h` and `defs19_LNS.h`. They should not be used separately because they are included in the basic header file `hsla_LNS.h` in connection with the 32- or 19-bit precision LNS arithmetics. To select the 32- or 19-bit precision arithmetics the macro (`#define`) `LOGPREC` is used. It must be either set to the value 32 or 19 respectively in the source code before the point where the `hsla_LNS.h` header file is included or this macro definition has to be provided as the *C* compiler parameter.

The *C* static libraries `hsla_32.lib` for the 32-bit version and `hsla_19.lib` for the 19-bit version of the HSLA toolbox can be found in the `lib/` directory. The appropriate library has to be linked into your application to access the LNS functions.

The directory `mex/` contains the source code files of the Matlab *mex* libraries. Users can customize these source files but it is highly recommended to save the backup copies of them to preserve the correct set of Matlab *mex* libraries.

The directory `runlib` contains two subdirectories `32bit` and `19bit` where the appropriate Matlab DLL's are stored after they have been built.

1.2 Building Matlab *MEX* libraries

The Matlab *mex* libraries has been pre-built and tested using the *Microsoft Visual C/C++ Version 6.0* (MSVC) and the *Matlab Version 6.1 Release 12.1*. The user can use the *MSVC* and the Matlab *mex* script to rebuild libraries. The *DOS* batch file `make.bat` should be used for this purpose.

It is supposed that the *DOS* environment is correctly set. Ensure that you have set enough space for environment variables, correct PATH and other variables required by the *C* compiler, and that you have set PATH variable to access *mex* (Matlab) executables.

To access HSLA functions directly from the Matlab command line or from your Matlab scripts, set the correct PATH variable in your Matlab environment using the following command from the Matlab command line:

```
addpath <HSLA_PACKAGE>/runlib/32bit;
```

or

```
addpath <HSLA_PACKAGE>/runlib/19bit;
```

where `<HSLA_PACKAGE>` specifies the location of the `libhsla/` directory in your directory tree.

2 Data Representation for the LNS Arithmetics

2.1 Data Format, Range, and Precision

The LNS data representation consists of a MSB[1] sign bit (denoted by s in Fig. 1) and the two's complement fixed point value equal to $\log_2|x|$, where x is the value to be represented and $|\cdot|$ is the absolute value operator. The signed two's complement fixed point number is divided to an integer part, which is always 8-bit long, and to a fraction part (its size depends on the data precision). Binary data representations of the 32- and 19-bit precision LNS number format are depicted in Fig. 1.

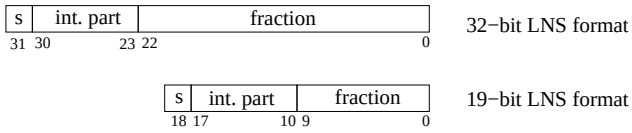

Figure 1: The 32- and 19-bit LNS data format

The standard IEEE single precision floating point (FLP) representation uses a sign bit, 8-bit biased exponent, and 23-bit mantissa. This format holds signed values in the range 1.2×10^{-38} to 3.4×10^{38}.

In the equivalent 32-bit precision LNS representation, the integer and fractional parts are kept as coherent two's complement fixed-point value in the range -128 to ≈ 128. The real numbers represented are signed and in the range $\approx 2.9 \times 10^{-39}$ to 3.4×10^{38}. One special value is used to represent the real number zero.

The 19-bit precision LNS format maintains the same range as 32-bit but has precision reduced to 11 fractional bits. It is comparable to the 16-bit FLP formats used on commercial DSP devices.

2.2 The LNS Operations

In the LNS, a value x is represented as the fixed point quantity $i = log|x|$, with an extra bit to indicate the sign of x and a special arrangement to accommodate zero and other exceptional values. Base-2 logarithms are used, though in principle any base could be used. For two LNS values $i = log_2|x|$ and $j = log_2|y|$, the LNS arithmetics involves the following computations:

$$log_2(x+y) = i + log_2(1+2^{j-i}), \quad (1)$$

$$log_2(x-y) = i + log_2(1-2^{j-i}), \quad (2)$$

$$log_2(x*y) = i + j, \quad (3)$$

$$log_2(\frac{x}{y}) = i - j, \quad (4)$$

$$log_2(\sqrt{x}) = \frac{i}{2}, \quad (5)$$

where in (1) and (2), without loss of generality, we choose $j \leq i$. In all these cases the sign bits are handled separately by a simple logic.

[1] Most Significant Bit

The equations (3), (4), and (5) can be implemented as simple as fixed point addition, subtraction, and shift operations. Unfortunately the equations (1) and (2) require evaluation of a non-linear function

$$\mathcal{F}(r = j - i) = log_2(1 \pm 2^r), \tag{6}$$

which can be seen of the Figure 2. The solution of this problem is discussed in [2] and [1].

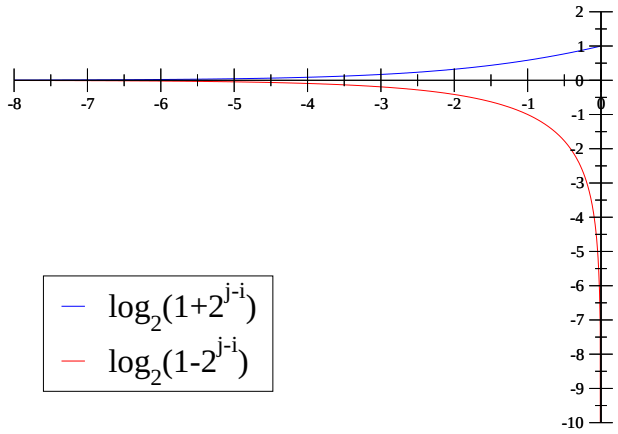

Figure 2: LNS sum and difference functions

Each operation in the HSLA returns result and a status flag. Status flag corresponds to the calculation result. The different statuses and the reserved LNS values are presented in the Table 1.

Description	status flag	32-bit value	19-bit value	output value
positive value	0			result
negative value	1			result
LNS zero	2	`0x40000000`	`0x00020000`	zero
LNS NAN	3	`0xC0000000`	`0x00060000`	NaN
+ overflow	4	`0x3FFFFFFF`	`0x0001FFFF`	largest positive
− overflow	5	`0xBFFFFFFF`	`0xFFFDFFFF`	largest negative
+ underflow	6	`0x40000000`	`0x00020000`	zero
− underflow	7	`0x40000000`	`0x00020000`	zero

Table 1: Reserved values and statuses for the 32- and 19-bit precision LNS

2.3 Standard and Extended Precision Arithmetics

The HSLA library includes special versions of the LNS operations working in an extended range of values. If the values in a section of an algorithm are guaranteed to fall in the range $\langle -1; 1 \rangle$, the special form of LNS operations provides extended data range in respect to underflow.

The classical LNS would only recognize values in the range of $\langle \pm 10^{-38}; \pm 10^{38} \rangle$ (the smallest step size is 10^{-38}). The extended data representation is able to distinguish the values in the range of $\langle \pm 10^{-76}; \pm 10^{76} \rangle$.

The extended precision is achieved by mapping the range $\langle -1; 1 \rangle$ to the range $\langle -Max; Max \rangle$. Arithmetic operations use identical hardware with only a few additional shift-type operations. This is the LNS analogy of the fixed-comma operations in the fixed point arithmetic world, or the floating point operations with re-defined exponent range.

All operands of the extended precision operations must be scaled from the range $\langle -1; 1 \rangle$ to the range $\langle -Max; Max \rangle$ before the extended precision functions are used. Let C be a scaling constant $C = Max$, where Max is the maximum LNS number. For normalization (scaling from $\langle -1; 1 \rangle$ to $\langle -Max; Max \rangle$) the following expression can be used:

```
logmul (a, C, &na, &flag);
```

where `a` is a LNS variable in the range $\langle -1; 1 \rangle$ to be normalized, `na` is the LNS variable after normalization in the range $\langle -Max; Max \rangle$.

For the de-normalization (scaling from $\langle -Max; Max \rangle$ to $\langle -1; 1 \rangle$) the following expression can be used:

```
logdnore (na, &a, &flag);
```

or

```
logdiv (na, C, &a, &flag);
```

where `na` is a normalized variable in the range $\langle -Max; Max \rangle$, `a` is the de-normalized variable in the range $\langle -1; 1 \rangle$. See logarithmic de-normalization (see Sec. 3.2.2), multiplication (see Sec. 3.1.5), and division (see Sec. 3.1.4).

3 The *C* Level Functions

The HSLA libraries hold the basic low level functions required for any C application using LNS arithmetics. The following libraries are available:

`hsla_19.lib`	19-bit LNS arithmetic operations
`hsla_32.lib`	32-bit LNS arithmetic operations

Both libraries holds the same two sets of functions. One set operates on numbers in the full data range $(\pm 3.4 \times 10^{38}; \pm 2.9 \times 10^{-39})$, the other covers the special version of LNS functions operating on the extended precision data range, intended for the algorithms calculating in the range $\langle -1; 1 \rangle$.

All LNS functions work with scalars and have the following syntax:

```
void fn_name (int_T a, int_T b, int_T *res, int_T *fl);
```

or

```
void fn_name (int_T a, int_T *res, int_T *fl);
```

where the type `int_T` is either *integer* or it is defined in the the Matlab header file `mex.h` if Matlab *mex* compiler is used.

3.1 Standard Precision Functions

The basic LNS functions contained in the `hsla_XX.lib` library will be described in this section.

3.1.1 The Function *int2log ()*

Purpose: Integer to logarithm domain conversion.

Synopsis:

```
void int2log (int_T a, int_T *res, int_T *fl);
```

Description: The function converts integer domain numbers into the LNS representation. The input numbers are supposed to be in the 32- or 19-bit precision LNS format (see below). The result and return status of conversion are returned in the variables `res` and `fl`.

Input format: Input numbers are supposed to be two's complement fraction part of the fixed-point value in the range $\langle -1; 1 \rangle$. There is difference in the input format if used 32- or 19-bit version of `int2log()`. 32-bit version uses 24-bit fraction part of the fixed-point input number. 19-bit version uses 16-bit fraction part of the fixed-point input number. Negative numbers are stored in the two's complement. Zero, 1, and -1 are represented as depicted in the Table 2.

Output format: Output is a 32- or 19-bit LNS format number.

DEC value	HEX for 32-bit	HEX for 19-bit
0	`0x00000000`	`0x00000000`
1	`0x01000000`	`0x00010000`
-1	`0xFF000000`	`0xFFFF0000`

Table 2: Integer domain number representation for 32- and 19-bit HSLA

Example: Suppose we want to convert number -0.423 in 32- or 19-bit LNS environment. We will use the following:

```
#define LOGPREC        32
#include "hsla_LNS.h"

#define INT_SHIFT_32   24
#define INT_SHIFT_19   16

#if LOGPREC == 32
#define CONV_SHIFT   (1 << INT_SHIFT_32)
#else
#define CONV_SHIFT   (1 << INT_SHIFT_19)
#endif  /* LOGPREC */

int main (void) {
  int_T a;             /* integer domain number */
  int_T result, flag;

  /*
     .
     .
     .
  */

  a = (int_T) (-0.423 * CONV_SHIFT);
  int2log (a, &result, &flag);

  /*
     .
     .
     .
  */

  exit (0);
}
```

See also: `log2int()`

3.1.2 The Function *log2int ()*

Purpose: Logarithm to integer domain conversion.

Synopsis:

```
void log2int (int_T a, int_T *res, int_T *fl);
```

Description: The function converts LNS numbers into the integer domain. The result and return status of conversion are returned in the variables `res` and `fl`.

Input format: Input is a 32- or 19-bit LNS format number in the range $\langle -1; 1 \rangle$.

Output format: The output integer number is held in the same format as the input number of the `int2log()` function (see Sec. 3.1.1).

Example: Suppose we want to get the floating-point number from the conversion in 32- or 19-bit LNS environment. We will use the following:

```
#define LOGPREC        32
#include "hsla_LNS.h"

#define INT_SHIFT_32   24
#define INT_SHIFT_19   16

#if LOGPREC == 32
#define CONV_SHIFT   (1 << INT_SHIFT_32)
#else
#define CONV_SHIFT   (1 << INT_SHIFT_19)
#endif  /* LOGPREC */

int main (void) {
  int_T a;             /* logarithm domain number */
  int_T result, flag;
  float f;

  /*
     .
     .
     .
  */

  log2int (a, &result, &flag);
  f = (float) (result / CONV_SHIFT);

  /*
     .
     .
     .
  */

  exit (0);
}
```

See also: `int2log()`

3.1.3 The Function *logadd ()*

Purpose: Logarithmic addition in the meaning of (1).

Synopsis:

```
void logadd (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs the logarithmic addition. The result and return status of the operation are returned in the variables `res` and `fl`.

Note: The logarithmic addition can be used also for the extended precision arithmetic without any change (see 2.3).

See also: `logsub()`

3.1.4 The Function *logdiv ()*

Purpose: Logarithmic division in the meaning of (4).

Synopsis:

```
void logdiv (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs the logarithmic division. The result and return status of the operation are returned in the variables `res` and `fl`.

See also: `logdive()`

3.1.5 The Function *logmul ()*

Purpose: Logarithmic multiplication in the meaning of (3).

Synopsis:

```
void logmul (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs the logarithmic multiplication. The result and return status of the operation are returned in the variables `res` and `fl`.

See also: `logmule()`

3.1.6 The Function *logsqrt ()*

Purpose: Logarithmic square root in the meaning of (5).

Synopsis:

```
void logsqrt (int_T a, int_T *res, int_T *fl);
```

Description: The function takes one argument and performs the logarithmic square root. The result and return status of the operation are returned in the variables `res` and `fl`.

See also: `logsqrte()`

3.1.7 The Function *logsub ()*

Purpose: Logarithmic subtraction in the meaning of (2).

Synopsis:

```
void logsub (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs the logarithmic subtraction. The result and return status of the operation are returned in the variables `res` and `fl`.

Note: The logarithmic subtraction can be used also for the extended precision arithmetic without any change (see 2.3).

See also: `logadd()`

3.1.8 Conversion Functions *double2log ()* and *log2double ()*

Purpose: Double to logarithm and logarithm to double conversions.

Synopsis:

```
void double2log (double *a, int nelems, double *res,
                 double *fl);
```

and

```
void log2double (double *a, int nelems, double *res);
```

Description: The functions convert an array of double precision numbers to the LNS format and vice versa. They convert numbers in the full LNS format range. To compute the results the definition of base-2 logarithm, $log_2(x) = \frac{log(x)}{log(2)}$, is used. The functions work with an array of data. Input and output arrays are held as pointers to double. Number of elements of array is specified by the parameter `nelems`. The array of return statuses is held in the variable `fl`.

Note: These functions serve for conversions with maximum precision and they do not have bit-exact equivalents at the hardware level.

Example:

```
#include <stdlib.h>

#define LOGPREC      32
#include "hsla_LNS.h"

int main (void) {
  double *a;           /* double precision numbers */
  double *results;     /* LNS numbers */
  double *flags;       /* return statuses */
  int n = 10;          /* number of elements */

  a = malloc (n, sizeof (double));
  results = malloc (n, sizeof (double));
  flags = malloc (n, sizeof (double));

  /*
     .
     .
     .
  */

  double2log (a, n, results, flags);

  /*
     .
     .
     .
  */

  exit (0);
}
```

3.2 Extended Precision Functions

3.2.1 The Function *logdive ()*

Purpose: Logarithmic division in the meaning of (4) for extended precision arithmetics.

Synopsis:

```
void logdive (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs logarithmic division with an extended data range (see Sec. 2.3). All input parameters must be scaled from the range $\langle -1; 1\rangle$ to the range $\langle -Max; Max\rangle$ before the use of this function. The result and return status of the operation are returned in the variables `res` and `fl`. For normalization and de-normalization use functions `logmul()` (see Sec. 3.1.5) and `logdnore()` (see Sec. 3.2.2).

See also: `logmul()`, `logdnore()`

3.2.2 The Function *logdnore ()*

Purpose: De-normalization routine.

Synopsis:

```
void logdnore (int_T a, int_T *res, int_T *fl);
```

Description: The function takes one argument and performs its de-normalization in the context of extended precision arithmetics (see Sec. 2.3). Input variable is scaled from the range $\langle -Max; Max \rangle$ to the range $\langle -1; 1 \rangle$. The result and return status of the operation are returned in the variables `res` and `fl`.

3.2.3 The Function *logmule ()*

Purpose: Logarithmic multiplication in the meaning of (3) for extended precision arithmetics.

Synopsis:

```
void logmule (int_T a, int_T b, int_T *res, int_T *fl);
```

Description: The function takes two arguments and performs logarithmic multiplication with an extended data range (see Sec. 2.3). All input parameters must be scaled from the range $\langle -1; 1 \rangle$ to the range $\langle -Max; Max \rangle$ before the use of this function. The result and return status of the operation are returned in the variables `res` and `fl`. For normalization and de-normalization use functions `logmul()` (see Sec. 3.1.5) and `logdnore()` (see Sec. 3.2.2).

See also: `logmul()`, `logdnore()`

3.2.4 The Function *logsqrte ()*

Purpose: Logarithmic square root in the meaning of (5) for extended precision arithmetics.

Synopsis:

```
void logsqrte (int_T a, int_T *res, int_T *fl);
```

Description: The function takes one argument and performs logarithmic square root with an extended data range (see Sec. 2.3). The input parameter must be scaled from the range $\langle -1; 1 \rangle$ to the range $\langle -Max; Max \rangle$ before the use of this function. The result and return status of the operation are returned in the variables `res` and `fl`. For normalization and de-normalization use functions `logmul()` (see Sec. 3.1.5) and `logdnore()` (see Sec. 3.2.2).

See also: `logmul()`, `logdnore()`

3.3 Interfaces for Double Precision Variables

To make the application programming easier a set of functions referred to as *double precision interfaces* has been designed. They do not perform any calculations but they are used for "double precision" access to the functions described in the Sections 3.1 and 3.2. The list of all available functions can be found in the Table 3. All listed functions have their bit-exact equivalents at the hardware level.

interface	LNS function	description
la ()	logadd ()	logarithmic addition
ld ()	logdiv ()	logarithmic division
lm ()	logmul ()	logarithmic multiply
ls ()	logsub ()	logarithmic subtraction
lsq ()	logsqrt ()	logarithmic square root
lde ()	logdive ()	extended precision logarithmic division
lme ()	logmule ()	extended precision logarithmic multiply
lsqe ()	logsqrte ()	extended precision logarithmic square root
ldnore ()	logdnore ()	logarithmic de-normalization
i2l ()	int2log ()	integer to logarithm domain conversion
l2i ()	log2int ()	logarithm to integer domain conversion

Table 3: Double interfaces and the corresponding LNS functions

Interface functions have the following syntax:

```
double interface_name (double a, double b);
```

or

```
double interface_name (double a);
```

Interface functions return the result as double. Status flag is written to the global variable `zstatus` declared in the header file `hsla_LNS.h`.

The use of an interface functions is shown in the following example:

```
#include <stdio.h>

#define LOGPREC        32
#include "hsla_LNS.h"

#define INT_SHIFT_32   24
#define INT_SHIFT_19   16

#if LOGPREC == 32
#define CONV_SHIFT   (1 << INT_SHIFT_32)
#else
#define CONV_SHIFT   (1 << INT_SHIFT_19)
#endif  /* LOGPREC */

int main (void) {
  double a, b;         /* floating point operands */
```

```
  double la, lb;       /* logarithmic operands */
  double z;            /* floating point result */
  double lz;           /* logarithmic result */

  a = 0.435;
  b = - 0.298;

  /* convert numbers to logarithm */
  la = i2l (a * CONV_SHIFT);
  lb = i2l (b * CONV_SHIFT);

  /* perform logarithmic addition */
  lz = la (la, lb);

  /* status flag is written to zstatus */
  if (zstatus == 0)
    printf ("The result is positive:   ");

  /* convert result to floating point */
  z = l2i (lz) / CONV_SHIFT;

  /* print result */
  printf ("%f\n", z);

  exit (0);
}
```

4 Matlab HSLA Toolbox *MEX* Functions

Matlab HSLA toolbox consists of a set of basic functions for LNS arithmetics. These functions can be called from the Matlab command line and used in the Matlab *m-scripts*. The source code of the *mex* functions can be found in the directory `mex/`.

4.1 Standard Precision Functions

4.1.1 The Function *i2log ()*

Purpose: Integer to logarithm domain conversion for Matlab.

Synopsis:

```
z = i2log (a);
```

or

```
[z, zfl] = i2log (a);
```

Description: Function works with scalars or matrices of double precision numbers holding inputs in the integer format and returns scalar or matrices holding numbers in the LNS format. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

Input format: Input numbers are supposed to be double precision numbers having the same data format and range as the input numbers used in the `int2log()` function (see Sec. 3.1.1).

Output format: Output is a double precision variable holding 32- or 19-bit LNS format number.

Example: Suppose we want to convert random data vector to its logarithmic equivalent in the 32-bit LNS environment. We will use the following:

```
INT_SHIFT_32 = 24;
a = fix (rand (5, 1) * (2^INT_SHIFT_32));
la = i2log (a);
```

In the case of the 19-bit LNS a small modification is required:

```
INT_SHIFT_19 = 16;
a = fix (rand (5, 1) * (2^INT_SHIFT_19));
la = i2log (a);
```

Variable `a` is a column vector of doubles holding the input integer data in the range $\langle -1; 1\rangle$. Variable `la` is a column vector of doubles holding the values in the LNS format.

See also: `log2i()`

4.1.2 The Function *log2i ()*

Purpose: Logarithm to integer domain conversion for Matlab.

Synopsis:

```
z = log2i (a);
```

or

```
[z, zfl] = log2i (a);
```

Description: Function works with scalars or matrices of double precision variables holding inputs in the range $\langle -1; 1\rangle$ in the LNS format and returns scalars or matrices of integer data in the double precision representation. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

Input format: Inputs are supposed to be double precision values holding 32- or 19-bit LNS format value in the range $\langle -1; 1\rangle$.

Output format: Output numbers are stored in the double precision having the same integer format as the input numbers of the `int2log()` function described in the Section 3.1.1.

Example: Suppose we want to convert random vector holding values in the range $\langle -1; 1\rangle$ stored in the LNS format to floating-point representation in the 32-bit LNS environment. We will use the following:

```
INT_SHIFT_32 = 24;
la = d2log (rand (5, 1));
a = log2i (a) / (2^INT_SHIFT_19);
```

In the case of the 19-bit LNS a small modification is required:

```
INT_SHIFT_19 = 16;
la = d2log (rand (5, 1));
a = log2i (a) / (2^INT_SHIFT_19);
```

Variable `la` is a column vector of double precision numbers in the range $\langle -1; 1\rangle$ stored in the LNS format. Variable `a` is the result of conversion divided to get the final real floating-point vector in the range $\langle -1; 1\rangle$, corresponding to the original real number vector generated by $rand\,(5, 1)$.

See also: `i2log()`, `d2log()`

4.1.3 The Function *ladd ()*

Purpose: Logarithmic addition for Matlab.

Synopsis:

```
z = ladd (a, b);
```

or

```
[z, zfl] = ladd (a, b);
```

Description: Function works with scalars and returns sum of two LNS numbers. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

Note: The logarithmic addition can be used also for the extended precision arithmetic without any change (see Sec. 2.3).

See also: `lsub()`

4.1.4 The Function *ldiv ()*

Purpose: Logarithmic division for Matlab.

Synopsis:

```
z = ldiv (a, b);
```

or

```
[z, zfl] = ldiv (a, b);
```

Description: Function works with scalars and returns the result of division of two LNS numbers. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

See also: `ldive()`

4.1.5 The Function *lmul ()*

Purpose: Logarithmic multiplication for Matlab.

Synopsis:

```
z = lmul (a, b);
```

or

```
[z, zfl] = lmul (a, b);
```

Description: Function works with scalars and returns the result of multiplication of two LNS numbers. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

See also: `lmule()`

4.1.6 The Function *lsub ()*

Purpose: Logarithmic subtraction for Matlab.

Synopsis:

```
z = lsub (a, b);
```

or

```
[z, zfl] = lsub (a, b);
```

Description: Function works with scalars and returns the result of subtraction of two LNS numbers. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

Note: The logarithmic subtraction can be used also for the extended precision arithmetic without any change (see Sec. 2.3).

See also: `ladd()`

4.1.7 The Function *lsqrt ()*

Purpose: Logarithmic square root for Matlab.

Synopsis:

```
z = lsqrt (a);
```

or

```
[z, zfl] = lsqrt (a);
```

Description: Function works with scalar and returns the result of the square root operation of LNS number. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

See also: `lsqrte()`

4.1.8 Conversion Functions *d2log ()* and *log2d ()*

Purpose: Conversions between double precision and the LNS format.

Synopsis:

```
z = d2log (a);
```

or

```
[z, zfl] = d2log (a);
```

and

```
z = log2d (a);
```

Description: The functions convert the double precision numbers to the LNS format numbers and vice versa. They use the *C* conversion functions `double2log()` and `log2double()` (see Sec. 3.1.8) and perform conversions both on scalars and on matrices.

4.2 Extended Precision Functions

4.2.1 The Function *ldive ()*

Purpose: Logarithmic extended precision division for Matlab.

Synopsis:

```
z = ldive (a, b);
```

or

```
[z, zfl] = ldive (a, b);
```

Description: Function works with scalars and returns the result of extended precision division of two LNS numbers. Input and output operands are in the extended data range (see Sec. 2.3). For normalization or de-normalization use `lmul()` (see Sec. 4.1.5) or `ldnore()` (see Sec. 4.2.2) respectively. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

Example:

```
la = d2log (rand (1));          % la and lb in <-1;1>
lb = d2log (rand (1));

if (la > lb)                    % must be: |la| < |lb|
  tmp = lb;                     % swap la and lb
  lb = la;
  la = tmp;
end;

lzero = d2log (zeros (1));      % LNS zero
lmax = lzero - 1;               % maximal LNS number

na = lmul (la, lmax);           % normalization
nb = lmul (lb, lmax);

nz = ldive (na, nb);            % ext. prec. division

     .
     .
     .

lz = ldnore (nz);               % denormalization
res = log2d (lz);
```

See also: `lmul()`, `ldnore()`, `d2log()`, `log2d()`

4.2.2 The Function *ldnore ()*

Purpose: De-normalization routine for Matlab.

Synopsis:

```
z = ldnore (a);
```

or

```
[z, zfl] = ldnore (a);
```

Description: The function takes one argument and performs de-normalization in the context of extended precision arithmetic (see Sec. 2.3). Input variable is scaled from the range $\langle -Max; Max \rangle$ to the range $\langle -1; 1 \rangle$. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

4.2.3 The Function *lmule ()*

Purpose: Logarithmic extended precision multiplication for Matlab.

Synopsis:

```
z = lmule (a, b);
```

or

```
[z, zfl] = lmule (a, b);
```

Description: Function works with scalars and returns the result of the extended precision multiplication of two LNS numbers. Input and output operands are in the extended data range (see Sec. 2.3). For normalization or de-normalization use `lmul()` (see Sec. 4.1.5) or `ldnore()` (see Sec. 4.2.2) respectively. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

See also: `lmul()`, `ldnore()`

4.2.4 The Function *lsqrte ()*

Purpose: Logarithmic extended precision square root for Matlab.

Synopsis:

```
z = lsqrte (a);
```

or

```
[z, zfl] = lsqrte (a);
```

Description: Function works with scalar and returns the result of the extended precision square root operation of LNS number. Input and output operands are in the extended data range (see Sec. 2.3). For normalization or de-normalization use `lmul()` (see Sec. 4.1.5) or `ldnore()` (see Sec. 4.2.2) respectively. At least one output parameter has to be specified. Status flag is optionally returned as the second output variable.

See also: `lmul()`, `ldnore()`

Part II

32- and 19-bit TWIN LNS ALU

5 The Installation of the LNS ALU

5.1 The Contents of the LNS ALU

This chapter gives you a brief information on what you can find in the LNS ALU directory. After unpacking the *lnsalu* package you will find the following directory structure in the `lalu/` directory:

`edif/`	*EDIF* files for 32- and 19-bit versions
`examples/`	*DK1 Handel-C* examples for 32- and 19-bit versions
`include/`	*DK1 Handel-C* header files for 32- and 19-bit versions
`plugins/`	*DK1 Handel-C* plugins for 32- and 19-bit versions

The LNS ALU cores are provided as *EDIF* files that can be found in the `edif/` directory. It contains two subdirectories, the directory `virtex/` holding "edifs" for *Virtex* targets and the directory `virtex2` holding "edifs" for *Virtex II* targets. Each of these subdirectories keeps all required *EDIF* files both for 32- and 19-bit precision LNS ALU. Optional cores of the fast *XILINX 32-bit dual-port block RAM's* are also contained in these directories.

Celoxica DK1 Handel-C examples for 32- and 19-bit LNS ALU together with the appropriate Matlab script examples are located in the directory `examples/` (see Sec. 8).

In the directory `include/` can be found *DK1 Handel-C* header files. The header file `laluTWIN.h` provides necessary interfaces to the individual LNS ALU modules. Other header files in this directory are optional and will be described in the Section 7.

It is possible to use *DK1 SIMULATION* option to test and debug user application in the DK1 simulator. Plugins provide bit-exact and cycle-exact model of modules that are not available as *Handel-C* source code and are located in the `plugins/` directory.

5.2 LNS ALU Examples Settings

All *DK1* examples have been prepared and tested with the *Celoxica RC1000* PCI card equipped with the *XILINX Virtex XCV2000E-6* FPGA device.

All *DK1* projects have been prepared for compiling without any additional user changes in the default project settings. The `README` files can be found in each `dk1` directory. The project settings changes prepared for the appropriate example are given in `README` files.

It is assumed that the support for *Celoxica RC1000* has been installed on the computer and that the following settings have been added into *DK1* environment. Choose **Tools** → **Options** → **Directories** → **Include files** and add:

```
<CELOXICA_ROOT>\DK1\Source
<CELOXICA_ROOT>\RC1000-PP\fpga\Virtex
```

where `<CELOXICA_ROOT>` is the directory where *Celoxica* products are installed (e.g. `C:\Program Files\Celoxica`). The `DK1\Source` directory has to be set to allow access to the file `stdlib.c`. The `RC1000-PP\fpga\Virtex` directory is required to provide access to the *Handel-C* header file for the *RC1000* card.

The *Celoxica RC1000* card support is also needed. In the *MS Visual Studio* choose **Tools** → **Options** → **Directories** → **Include files** and add:

```
<CELOXICA_ROOT>\RC1000-PP\INCLUDE
```

and in the **Tools** → **Options** → **Directories** → **Library files** add:

```
<CELOXICA_ROOT>\RC1000-PP\LIB
```

where `<CELOXICA_ROOT>` is the same directory as discussed in the previous paragraph.

The *MSVC* example source files facilitate communication with the *RC1000* card, load FPGA design, and to send or receive data to or from the card. All *MSVC* projects have been prepared to allow compilation without any additional user changes in the default project settings. The `README` files in each `msvc` directory can be found. The project settings changes prepared for the appropriate example are given in these files.

Matlab *m-scripts* does not require any additional user settings. For customization and other *HSLA Toolbox* settings see Section 1.

6 The LNS ALU and Celoxica DK1 Targets

6.1 LNS ALU Operations

Logarithmic addition, subtraction, multiplication, division, and square root are provided. Multiplication, division, and square root are implemented to accomplish these operations in one clock cycle. Addition and subtraction are implemented as 8-stage pipelined operations, i.e. appropriate operands can be provided to ALU in every clock cycle and the corresponding results are ready after eight clock cycles.

The *EDIF* and *DK1 Handel-C* files are available for the purpose of *DK1* simulations and of hardware implementation. The main LNS ALU modules `lalu32.edf` for 32-bit precision ALU and `lalu19.edf` for 19-bit precision are both implemented as dual-port 8-stage pipelined ALU that handles logarithmic addition and subtraction operations. It is possible to process two sets of operands in a parallel way in every clock cycle.

Division, multiplication, and square root operations are available as an array of functions (source codes are provided) in the ALU header file named `laluTWIN.h` used in the *DK1* environment. It is located in the `include/` directory. It also includes multiplication, division, and square root functions working in the extended precision data range for operands in the range $\langle -1; 1\rangle$. This header file also provides necessary interface to the addition-subtraction ALU module mentioned in the previous paragraph.

6.2 LNS ALU Functions

Function or macro	Operation
$z = lm\,[N]\,(a, b)$	logarithmic multiplication
$z = ld\,[N]\,(a, b)$	logarithmic division
$z = lsq\,[N]\,(a)$	logarithmic square root
$z = lme\,[N]\,(a, b)$	logarithmic extended precision multiplication
$z = lde\,[N]\,(a, b)$	logarithmic extended precision division
$z = lsqe\,[N]\,(a)$	logarithmic extended precision square root
$z = ldnor\,[N]\,(a)$	converting from extended precision
$z = ladd1\,(a, b)$	logarithmic addition, the first port function
$z = lsub1\,(a, b)$	logarithmic subtraction, the first port function
$z = ladd2\,(a, b)$	logarithmic addition, the second port function
$z = lsub2\,(a, b)$	logarithmic subtraction, the second port function
$la1\,(a, b)$	logarithmic addition, the first port macro
$ls1\,(a, b)$	logarithmic subtraction, the first port macro
$la2\,(a, b)$	logarithmic addition, the second port macro
$ls2\,(a, b)$	logarithmic subtraction, the second port macro
$z = int2log\,(a)$	integer to logarithm domain conversion
$z = log2int\,(a)$	logarithm to integer domain conversion

Table 4: TWIN LNS ALU functions

The functions and macros provided by 32- and 19-bit precision LNS ALU are shown in the Table 4. All operands have 32- or 19-bit width. The constant N is used as reference to the appropriate element of function array. A number of available parallel

function instances in the array is specified using an appropriate definition (see Sec. 7) during the compilation.

The $ladd1$, $lsub1$, $ladd2$, and $lsub2$ are functions. They return results in the same way as the other LNS functions. These functions wait for the ALU hardware and return result in the 10-th clock cycle. However, they block effective use of the ALU hardware in a pipeline.

The ALU can be interfaced in Handel-C using low-level macros $la1$, $ls1$, $la2$, and $ls2$. They do not return any results. The macros set appropriate values (connect registers) to the LNS ALU inputs. The results have to be acquired after 8 clock cycles (in the 9-th clock cycle) using the LNS ALU interface, i.e. by reading `lalu.z1` and `lalu.z2` (see Sections 7.1 and 8).

6.3 Celoxica DK1 Targets

It is possible to use the *DK1 SIMULATION* option to test and debug user application in the *DK1* simulator. The LNS ALU plugins `lalu32.dll` and `lalu19.dll` are available for this purpose. They are located in the `plugins/` directory. These plugins provide bit-exact and cycle-exact model of the pipelined dual-port LNS ALU.

Besides the SIMULATION mode, the *DK1* user can choose one of two compilation ways. *DK1* source codes can be compiled to the *VHDL* or *EDIF*. The files `lalu32.edf` and `lalu19.edf` are required by *XILINX P&R* tools. These files are located in the `edif/` directory. Additional necessary *EDIF* files can be found in appropriate subdirectories. All included `edf` and `edn` files have been compiled for *XILINX Virtex XCV2000E-6* and *XILINX Virtex II XC2V6000-4* FPGA devices.

7 LNS ALU Header Files

The following header files are provided:

`connect.h`	communication between PC and *Celoxica RC1000* card
`conv.h`	integer to logarithm domain and vice versa conversions
`lalu19.h`	19-bit LNS ALU parameters
`lalu32.h`	32-bit LNS ALU parameters
`laluTWIN.h`	basic LNS ALU definitions and interfaces
`ram.h`	dual-port block RAM interfaces
`ram32x256.h`	32-bit 256 record long block RAM parameters
`ram32x512.h`	32-bit 512 record long block RAM parameters
`ram32x1024.h`	32-bit 1024 record long block RAM parameters
`ram32x2048.h`	32-bit 2048 record long block RAM parameters
`ram32x4096.h`	32-bit 4096 record long block RAM parameters
`U_ram.h`	example block RAM interface referred to as *U*
`W_ram.h`	example block RAM interface referred to as *W*
`Y_ram.h`	example block RAM interface referred to as *Y*
`Z_ram.h`	example block RAM interface referred to as *Z*

The following header files can be directly used and the “including” order should be respected. A small fragment of using in the source code is given:

```
/* standard DK1 files */
#include <stdlib.h>
#include "stdlib.c"


/* dual-port LNS ALU header                      */
/* must be defined before including 'laluTWIN.h' */
#define LOGPREC     32
#include "laluTWIN.h"

/* communication between design and "PC" */
#include "connect.h"

/* int2log and log2int functions */
#include "conv.h"

/* defines length of 32-bit block RAM       */
/* must be defined before including 'ram.h' */
#define RAMLEN      1024
#include "ram.h"
```

The next sections provide description of individual header files in more detail.

7.1 The Header File *laluTWIN.h*

The LNS ALU has been designed to develop *DK1 Handel-C* applications using the LNS arithmetics. The *Handel-C* header file `laluTWIN.h` provides interface to the LNS ALU modules. In the same directory can also be found header files `lalu32.h` and `lalu19.h`. They should not be used separately because they are included in the

basic header file `laluTWIN.h` in connection with the 32- or 19-bit precision LNS arithmetics. To select the 32- or 19-bit precision arithmetics the macro (`#define`) `LOGPREC` is used. It must be set to the value 32 or 19 in the source code before including the `laluTWIN.h` header file or this macro definition has to be provided as the *Handel-C* compiler parameter.

Three types of compiler targets can be chosen. The default is compiling to *VHDL*. If a user decides to compile to *EDIF* the macro `TARGET_EDIF` has to be defined. Optionally the macro `SIMULATE` can be defined if compiling for the *DK1 SIMULATION* target. However in the most *DK1* configurations the macro `SIMULATE` is pre-defined by the *DK1 GUI* if the target *Debug* is chosen.

The LNS ALU parts designated to the logarithmic addition and subtraction are implemented as dual-port 8-stage pipelined hardware modules. By default the set of registers is connected to the input ports of ALU. This configuration is recommended due to timing restraints. In spite of this a user can disable this behavior by defining the macro `NOT_REG_LALU_INPUTS`.

The LNS ALU modules for multiplication, division, and square root both for standard and extended precision data range arithmetic (see Sec. 2.3) are implemented as arrays of functions. The number of functions in the appropriate array can be set by defining the following macros to the required value:

```
#define NUMBER_OF_LM_MODULES      1
#define NUMBER_OF_LD_MODULES      1
#define NUMBER_OF_LSQ_MODULES     1
#define NUMBER_OF_LME_MODULES     1
#define NUMBER_OF_LDE_MODULES     1
#define NUMBER_OF_LSQE_MODULES    1
#define NUMBER_OF_LDNOR_MODULES   1
```

The default number of functions in arrays is 1 as shown above.

7.2 The Header File *connect.h*

The functions `pc2alg()` and `alg2pc()` are provided and defined in the header file `connect.h` also located in the directory `include/`. Their purpose is communication between PC and a "hardware design". If *DK1 SIMULATION* mode has been chosen these functions use data files for communication. Otherwise hardware dependent macros are used and the support for *Celoxica RC1000* card has to be installed.

Communication data files in the *SIMULATION* mode can be defined using macros `SIM_INPUT_FNAME` and `SIM_OUTPUT_FNAME`. The following macros are defined by default:

```
#define SIM_INPUT_FNAME    "..\\input.dat"
#define SIM_OUTPUT_FNAME   "..\\outfpga.dat"
```

7.3 The Header File *conv.h*

The header file `conv.h` holds conversion functions `int2log()` and `log2int()`. The use of functions and data formats are intimately discussed in the Sections 3.1.1 and 3.1.2.

The functions use conversion tables that are implemented as the *DK1 dual-port block RAM's* or as the fast *XILINX dual-port block RAM's* generated using *XILINX*

CORE Generator System. The first implementation is easier to read and analyze the source code but less efficient and it is used only for demonstration purposes. The second one does the same job but it is capable to operate with faster clocks. It is highly recommended to use the latter implementation and it is the default behavior. The `int2log()` and `log2int()` functions return results in 20 and 44 clock cycles respectively in the 32-bit LNS ALU implementation. In the case of 19-bit LNS ALU functions return results in 11 and 26 clock cycles.

The default behavior can be disabled using the following macro definition before the including the header file `conv.h`. Then *DK1 dual-port block RAM's* are used.

```
#define NOT_I2L_COREGEN_RAM
```

7.4 The Header File *ram.h*

In the header file `ram.h` two hardware macros, `wd()` and `rd()`, prepared for accessing the fast 32-bit *XILINX dual-port block RAM's*. The macro `wd()` can be used to write data into the memory and the macro `rd()` to read data from the memory. The format of the macros is the following:

```
macro proc wd (bram, port, addr, din);
```

and

```
macro proc rd (bram, port, addr);
```

where the `bram` parameter represents the symbolic name of the appropriate memory (e.g. `U_RAM`), the `port` parameter assumes the value `portA` or `portB`, the `addr` is the memory address, and the parameter `din` represents the value to be written to the memory in the function `wd()`. The function `rd()` does not read the contents of the memory immediately but it has to be collected in the next clock cycle using the memory interface (e.g. `result = u.douta`).

As mentioned above the provided memories are of 32-bit width. The number of memory records is determined by the macro `RAMLEN`. The macro has to be defined before the including the header file `ram.h`. The RAM parameters are defined in the individual header files `ram32x<RAMLEN>.h`, where `<RAMLEN>` is assumed to be set to 256, 512, 1024, 2048, or 4096.

The header files `U_ram.h`, `W_ram.h`, `Y_ram.h`, and `Z_ram.h` are example interfaces for memories referred to as `U_RAM`, `W_RAM`, `Y_RAM`, and `Z_RAM`.

The array of signals connected directly to RAM interface is used to provide unified connection to individual memories using `wd()` and `rd()` functions. The coding conventions should be traceable from the `ram.h` header file and have to be changed to customize or add other memory modules.

All RAM modules are supported by *DK1* plugins, enabling bit- and cycle-exact simulation of the designs under the *DK1 SIMULATOR*.

8 The LNS ALU Examples

Examples of using LNS ALU can be found in the `examples/` directory. The *DK1 Design Suite Version 1.0 Service Pack 1* has been used. Examples have been tested in DK1 simulation and on the *Celoxica RC1000* card equipped with the *XILINX Virtex XCV2000E-6* FPGA device.

The 32- or 19-bit LNS ALU examples can be found in the subdirectories `32bit/` or `19bit` respectively. The *DK1 Handel-C* source codes are almost the same for both LNS ALU precisions except the `LOGPREC` macro definition.

The individual example directories has the same structure and the contents of subdirectories is very similar in principle:

`dk1/` *Celoxica DK1* project and source code files that can be compiled to EDIF, VHDL, or for DK1 simulation/debugger. The plugins (DLL files) required for *SIMULATION* are contained.

`msvc/` Microsoft Visual C source code of program that enables to upload hardware design *(bit-stream)* file to the *Celoxica RC1000* card and run final design in FPGA using the test vectors read from file.

<**`ex_name`**>**`sim.m`** Matlab test script for the *DK1 SIMULATION*. It creates the data file for *DK1* debugger, reads the results from the data file prepared by *DK1* simulator, and tests for identical bit-exactness.

<**`ex_name`**>**`hw.m`** Matlab test script for the design testing in FPGA. It writes data to the file, calls <`ex_name`>`.exe`, reads results from the file, and verifies for the bit-exactness.

<**`ex_name`**>**`.exe`** Win32 console application to be used to upload hardware design to FPGA and provide data to hardware.

<**`ex_name`**>**`.bit`** Hardware design. It is downloaded by <`ex_name`>`.exe` to the *Celoxica RC1000* card.

<**`ex_name`**>**`-edif.bit`** Hardware design built using EDIF path.

<**`ex_name`**>**`-vhdl.bit`** Hardware design built using VHDL path.

The <`ex_name`> stands for the appropriate name of example.

Running the *DK1 SIMULATION* the appropriate plugins (DLL) have to be copied to the `dk1/` directory. Follow these steps to simulate:

1. Start Matlab and execute the *m-script* <`ex_name`>`sim`. The script creates input data file for the simulator. The following message appears in the Matlab window:

   ```
   Run DK1 simulation.
   --- press any key ---
   ```

2. Start (use) *DK1* simulator to run the simulation. You can run the *Handel-C* example by pressing the key `F5` or trace and debug the code, use breakpoints etc. in *DK1*.

3. When the simulation has been finished the output file is created. Return back to the Matlab and press any key. The following or similar message should appear:

```
<ex_name>: Results are OK.
```

This message indicates that the simulator generated bit-exactly identical results with the Matlab libraries.

To run and test examples in the *Virtex* FPGA the appropriate <ex_name>.bit file has to exist in the current directory where the Matlab script is located. Follow these steps to test example:

1. Start Matlab and execute the *m-script* <ex_name>hw. The script creates input data file for the *Win32* console application <ex_name>.exe which is executed by the script. The program <ex_name>.exeloads the design into the FPGA, provides input data, and stores the results to the output file.

2. After the output file has been created the finishes the computation and the following or similar message should appear:

   ```
   <ex_name>: Results are OK.
   ```

   This message indicates that the "FPGA design" generated bit-exactly identical results with the Matlab libraries.

8.1 The Example *alutest*

The example *alutest* tests most of the LNS ALU operations described in the Section 6.2. It is the most simple example.

8.2 The Example *vadd01*

The example *vadd01* demonstrates addition of two vectors. It is implemented to use the pipelined ALU effectively. It uses just one port of ALU to reduce the complexity of demo code.

Fast *XILINX 32-bit dual-port block RAM's* are used to store input and output vectors. They have been generated using *XILINX CORE Generator System*. The appropriate EDIF files are located in the `edif/virtex/ram/` or `edif/virtex2/ram/` directories. The *DK1* plugins needed for simulation are located in the `plugins/ram` directory. RAM interfaces are discussed in the Section 7.4.

8.3 The Example *vadd02*

The example *vadd02* also demonstrates pipelined addition of two vectors except it uses both ports of ALU to compute result more effectively.

In FPGA hardware, if 50 MHz clock rate is used, the LNS ALU provides 100 MFlops performance.

8.4 The Example *i2ll2i*

The example *i2ll2i* shows how to use the conversion, `int2log()` and `log2int()`, functions that can be found in the header file `conv.h` (see Sec. 7.3) located in the `include/` directory. To store input and output vectors the "CoreGen" RAM's are

used (see Sec. 7.4). The functions use the different ports of the LNS ALU if the "CoreGen" RAM implementation of conversion tables is used. Thus they can be used in a parallel way.

The appropriate EDIF files for hardware implementation are located in the directories `edif/virtex/` or `edif/virtex2/` both for 32- and 19-bit precision implementations. The *DK1* plugins (DLL files) needed for the simulation are located in the `plugins/32bit/` or `plugins/19bit/` directories.

8.5 The Example *vaddsp*

The example *vaddsp* implements the same vector addition as the example *vadd02* (see Sections 8.2 and 8.3) except it includes `int2log()` and `log2int()` conversion functions for input and output vectors respectively. The input and output numbers are stored in the same format as described in the Sections 3.1.1 and 3.1.2.

9 Future Work

This chapter briefly presents what changes and new features you can expect in the next versions of the *Matlab HSLA Toolbox* and the *TWIN LNS ALU*:

- the most of *mex* functions in the *HSLA Toolbox* support only scalar operations in this version; both scalar and matrix operations will be supported
- HSLA libraries are guaranteed to work under *MS Visual C/C++* environment at present; the next versions should also support the *GNU gcc* compiler
- *C* and *mex* libraries will be platform independent and available also for *UNIX* platforms
- both *HSLA Toolbox* and *LNS ALU* will also provide the conversion routines (`float2log` and `log2float`) converting numbers from the standard IEEE 32-bit single precision floating point to the LNS format and vice versa
- the next version of the *HSLA Toolbox* will provide the equivalent IEEE 32-bit floating point functions available for the test and benchmark purposes
- communication functions provided in the header file `connect.h` will be available for the *XESS 800* board
- the same communication functions will also be provided for the *Alpha Data ADM-XRC* card equipped with the *XILINX Virtex XCV1000* FPGA device

References

[1] J. N. Coleman and E. I. Chester. *A 32-bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-point*. Research report, Department of Electrical and Electronic Engineering, The University of Newcastle upon Tyne, 2000.

[2] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec. Arithmetic on the European Logarithmic Microprocessor. In *IEEE Transactions on Computers*, number 7 in 49, pages 702–715. IEEE Computer Society, July 2000.