ON Semiconductor®

User Guide
IP Programmer for LPDSP32–V3

Release 10R1
September 2013

SANYO Semiconductor Co., Ltd.
An ON Semiconductor Company

Confidential and Proprietary

SANYO Semiconductor Co., Ltd.
1-1-1, Sakata, Oizumi-machi, Ora-gun, Gunma 370-0596, Japan

Legal notes

Change log

Version	Date	Change
10R1	April, 2011	Initial version.
	May 4, 2011	Added `wide_offset_add()` pointer intrinsic
	Sep, 2011	Added optimization techniques
		Added porting examples
	Mar, 2012	Added exceptional cases
	Sep, 2013	Company logo is changed

Table of Contents

1. INTRODUCTION 5
2. C APPLICATION LAYER 6
2.1 Overview 6
2.2 C Integer Types 6
2.3 C Floating-Point Types 8
2.4 Accumulator Type 8
2.5 Intrinsic Functions 9
2.5.1 Type Conversion Functions 9
2.5.2 Arithmetic Functions 11
2.5.2.1 Multiplication 11
2.5.2.2 Divide Step 12
2.5.2.3 Normalization 13
2.5.2.4 Maximum, Minimum, Absolute Value 13
2.5.2.5 Bit Set, Bit Reset, Bit Invert, Bit Test 14
2.5.3 Special Addressing Modes 14
2.5.3.1 Cyclic Addressing 14
2.5.3.2 Bit Reversal Addressing 15
2.5.3.3 Wide Pointer Offset 15
2.5.4 Controlling the Processor Mode 16
2.6 Storage Qualifiers 16
2.7 Mapping Variables to a Fixed Address 17
2.8 Alignment of Data Types 17
2.9 I/O interface 18
2.10 Interrupts 18
2.11 Inline Functions 19
2.12 C Library Support 20
2.13 C-Application Design Flow 20
2.14 Optimization Techniques 21
2.14.1 Optimization of Loops 25
APPENDIX A. LINKER CONFIGURATION FILE 31
APPENDIX B. INITIALIZATION CODE 32
APPENDIX C. EXAMPLES 34
C.1 Dual MAC Operation 34
C.2 Normalization 35
C.3 Bit Reverse Addressing 36
C.4 FIR filter 37
C.5 Complex FIR Filter 39
C.6 IIR Filter 41
C.7 All-pole IIR Lattice Filter 42
C.8 Matrix Multiplication 44
C.9 Auto Correlation 46
APPENDIX D. EXCEPTIONAL CASES 48

D.1 Elongation failure (before version 10R1.12) **48**
D.2 Induction Variable Analysis failure **49**
BIBLIOGRAPHY **50**

1. Introduction

This is the end-user guide for the IP Programmer for LPDSP32 tool suite. The LPDSP32 processor is developed and owned by Sanyo.

The tool suite consists of following tools.

- C compiler, (dis)assembler, and linker, which are started from the Chess development environment CHESSDE.
- Two prebuilt instruction set simulators (`lpdsp32` and `lpdsp32_fast`) are provided to simulate, debug, or profile LPDSP32 programs.
- Also a debug client (`lpdsp32_client`) is provided, to connect a debugger to the hardware board containing the LPDSP32. This is done via the Amontec JTAG Key cable.

All these tools are integrated into CHESSDE. On Windows, CHESSDE can be started via the Start menu.

The overview manual [1] summarizes the different manuals included in the distribution. Most manuals are generic manuals belonging to the retargetable IP Designer tool suite. Processor-specific information is included in separate manuals like this one.

This LPDSP32-specific manual describes the C application layer (C data types and intrinsic functions). It also describes the flow generally followed when any application is ported to LPDSP32, various tips for optimization to make the best use of the processor and compiler resources and certain things the programmer should be aware of when porting applications.

At the end a few examples are provided to show the usage of LPDSP32 intrinsic functions and to give an idea as to how certain DSP functions can be ported to and optimized for LPDSP32.

2. C Application Layer

2.1 Overview

This chapter describes the C application layer that the CHESS compiler supports for LPDSP32. It consists of following information.

- The implementation of the C built-in types and operators on LPDSP32.
- The dedicated accumulator data type, called `accum_t`.
- The intrinsic C functions directly mapping to specific LPDSP32 instructions.
- I/O interface and interrupts.
- C application design flow.
- Optimization techniques for porting applications to LPDSP32.

Other information, like standard C compliance, C library support, CHESS-specific source code annotations, mixing C and assembly, can be found in ([2] §Chapters 3, 4)

2.2 C Integer Types

All built-in C integer types and operators are supported. As LPDSP32 is a 32-bit processor, the basic integer type `int` corresponds to a 32-bit word. The following table gives the width of all integer types.

Name	Number of Bits
`char`	8
`short`	16
`int`	32
`long`	32
`long long`	64

Every type also has an `unsigned` variant having the same width as its signed counterpart.

All C built-in operators and conversions are supported on the integer types. Specifically, following operators are supported on the types (`unsigned`) `int` and (`unsigned`) `long` `long`.

- The additive operators (+, -)
- The multiplicative operators (*, /, %)
- The shift operators (<<, >>)
- The relational operators (<, <=, >, >=, ==, !=)
- The bitwise logical operators (&, |, ^, ~)
- The logical operators (&&, ||)
- All derived operators like increment operator, unary minus, and assignment operators (+=, . . .)
- Type conversions between all integer types (only type conversions between different widths result in actual LPDSP32 instructions).

Some Remarks:

- As the LPDSP32 smallest addressable memory word is 8-bit, it holds that:

```
          sizeof(char) = 1
         sizeof(short) = 2
           sizeof(int) = 4
          sizeof(long) = 4
     sizeof(long long) = 8
```

- The C operators are either done on 32-bit (for the type (`unsigned`) `int`) or 64-bit precision (for

the type (unsigned) long long). Both precisions are efficiently supported on LPDSP32. The only difference is that only 4 data registers can hold 64-bit values, while 8 data registers can hold values up to 32-bit. Also the 64-bit multiplication and division are somewhat more expensive.

- In case of narrowing integer type conversions, the excessive bits are discarded (i.e., wrapping or modulo behavior).

- Left shifts (<<) and multiplication (*) always do wrapping (i.e., modulo behavior) in case the mathematical result exceeds the 32-bit or 64-bit range. A signed right shift (>> on int and long long) is injecting sign bits at the MSB side. The shift factor should be non-negative and smaller than 32 or 64 for 32 and 64-bit shifts respectively.

- Additive operators on the signed integer types (int and long long) have no well-defined behavior in case of overflow. Depending on the context, you can either obtain wrapping, saturation, or use of the extended precision of the 72-bit data-path. Use the unsigned type variant to force wrapping, or make use of the accum_t type to use the extended precision or force saturation (§2.4, §2.5). The accum_t type result is passed through a rounding and saturation unit to take care of the overflow.

  Examples:

```
int a, b, c;
c = (unsigned)b + c;                              //32-bit  modulo
c = rnd_saturate(to_accum(a) + to_accum(b));      //32-bit  saturated
long long aa, bb, cc;
cc = (unsigned long long)aa + bb;                 //64-bit  modulo
cc = saturate(llto_accum(aa) + llto_accum(bb));   //64-bit  saturated
```

- Division (/) and modulo (%) operations are supported for all data types (both signed and unsigned). An internal library uses the hardware division unit to compute the quotient and remainder of the divisor and the dividend (§2.5.2.2). If a/b and a%b are needed together, they are recognized as common sub-expressions when a and b are not changed in between.

- It is also possible to initialize int variables with fractional floating-point constants, using following reinterpretation function:

Function	Description
`int as_int(double d);`	d must be a constant in range [-1,1], interpreted as 32-bit number with 31 fractional bits.

  Example: `int a = as_int(0.5);  //same as a = 0x40000000`

- bool data type of C-language is not supported on LPDSP32.

- After including <stdint.h>, you have access to following exact-width C types:

```
typedef signed char             int8_t;
typedef signed short            int16_t;
typedef signed int              int32_t;
typedef signed long long        int64_t;
typedef unsigned char           uint8_t;
typedef unsigned short          uint16_t;
typedef unsigned int            uint32_t;
typedef unsigned long long      uint64_t;
```

2.3 C Floating-Point Types

Although LPDSP32 is a fixed-point processor, the CHESS compiler also supports the C built-in types `float` (single precision) and `double` (double precision) on LPDSP32, by emulating them in software. The type `long double` is not supported on LPDSP32.

All C built-in operators on the floating types are provided:

- arithmetic operators (+, -, *, /),
- the relational operators (<, <=, >, >=, ==, !=),
- conversion between `float/double` and between the integer types (`unsigned/signed int/long/long long`).
- Also some basic <`math.h`> functions are supported, listed in (§2.12).

The implementation is IEEE-754 conforming, and is based on the Softfloat package of John R. Hauser.

But floating point arithmetic is not recommended for final use since floating point code significantly increases the cycles required when compared to a fixed point implementation (like in any fixed point processor emulating floating point operations).

2.4 Accumulator Type

To model the 72-bit accumulator registers in LPDSP32, the 72-bit `accum_t` data type has been introduced. It consists of 8 overflow bits (extension word), 32-bit high word and 32-bit low word. This type is used to do DSP accumulations, where we accumulate the sum of products of fractional numbers (`Q1.31` x `Q1.31`) using the intrinsic function, `fract_mult()`. The final accumulated value can be rounded and saturated to obtain the output in `Q1.31` format or only saturated to obtain the output in `Q1.63` format.

Note: In this document to represent fractional numbers we use the Q format(`Qn.m`), where `n` is the number of bits before a notional binary point, and `m` is the number of bits that follow it. `n` specifies how many bits represent an integer value, and `m` specifies how many bits represent subdivisions within each integer value. We use the signed Q-format, so values are divided equally on either side of zero. The data range for a signed `n.m` bit fractional number is $(-2^{(n-1)})$ to $(1 - 2^{((n-1)-m)})$. Thus, the data range for a 32 bit(`Q1.31`) fractional number is -1 to +0.99999999953433, including '0.0', i.e. 0x80000000 to 0x7FFFFFFF, including 0x00000000.

On the accumulator type the same operators are provided as on the integer types (§2.2) with following exceptions.

- No multiplicative operators (*, /, %) are provided. Instead, intrinsic multiply functions are provided (§ 2.5.2).
- No type conversion operators with integer types are provided. Instead, intrinsic conversion functions are provided (§ 2.5.1).
- No increment/decrement operators (++,—).

Some Remarks:

- For variables of type `accum_t` the shift factor can be negative which will result in a shift to opposite direction. The compiler uses arithmetic shift instructions for right/left shift.

- When initializing an `accum_t` variable with a literal (32-bit variable), we must specify the same using the intrinsic function to_accum(val). The 32-bit value is stored in bits 63 to 32 of the accumulator, bits 31 to 0 are filled with zeros and the sign is extended. Similarly, for a 64-bit value, we use `llto_accum(val)`. The value is stored in bits 63 to 0 and the result is sign extended.

Examples:

```
accum_t val0 = 0;                        //Compiler error
accum_t val1 = to_accum(0);              //OK
accum_t val2 = to_accum(0x1276583F);     //OK: 72-bit literal
                                         //val2 = 0x00-1276583F-00000000
accum_t val3 =                           //OK: 72-bit literal
llto_accum(0x1276583F76EDF011);          //val3 = 0x00-1276583F-76EDF011
```

2.5 Intrinsic Functions

Intrinsic functions are dedicated functions, provided in the C compiler, to implement functionality that is not available through operators of ANSI C. These functions have an efficient hardware implementation on the LPDSP32. When the compiler encounters such a function, it recognizes it and uses the matching LPDSP32 functionality to implement the C code.

2.5.1 Type Conversion Functions

The following type conversion functions are provided to effectively convert from one data type to another.

Function	Description
`accum_t to_accum(int i);`	puts `i` in high word, with sign extension in extension word, and zero in low word
`int rnd_saturate(accum_t a);`	extracts high word of `a` after rounding and saturation (dependent on round & saturate mode bits)
`int extract_high(accum_t a);`	extracts high word of `a` (not affected by mode bits)
`int extract_low (accum_t a);`	extracts low word of `a`
`int extract_ext (accum_t a);`	extracts extension word of `a` (8-bits sign-extended to 32-bits)
`accum_t update_low(accum_t a, int i);`	overwrites low word of `a` with `i`
`accum_t update_ext(accum_t a, int i);`	overwrites extension word of `a` with 8 LSBs of `i`
`accum_t llto_accum(long long l);`	puts 64-bit `l` in high::low word, with sign extension in extension word
`long long saturate(accum_t a);`	extracts 64-bit high::low word after saturation (dependent on S mode bit)
`long long extract_long(accum_t a);`	extract 64-bit high::low word (not affected by mode bits)
`int extract_high(long long l);`	extracts 32-bit high word of `l`
`int extract_low(long long l);`	extracts 32-bit low word of `l`
`long long deposit_high(int i);`	puts `i` in high word, zero in low word
`long long update_low(` `long long l, int i);`	overwrites low word of `l` with `i`
`unsigned long long llcompose(` `int a, int b);`	concatenates two 32-bit words to a 64-bit word (`a` at LSB side, `b` at MSB side).
`void lldecompose(` `unsigned long long l, int& a, int& b);`	splits 64-bit `l` in two numbers `a` (32 LSBs) and `b` (32 MSBs).

The most natural type conversion functions on LPDSP32 are `to_accum`(), `rnd_saturate`() for conversions between `int` and `accum_t`, and `llto_accum`(), `saturate`() for conversions between `long long` and `accum_t`. The other functions are also mapped efficiently on LPDSP32.

The (de)compose functions are useful for complex arithmetic, where a 64-bit (`unsigned long long`) variable contains a real and imaginary part. The (de)composition is possible only on 64-bit aligned memory accesses, so it only makes sense to apply these functions on 64-bit arrays. If the memory is not aligned, the simulator breaks with an error but the hardware JTAG debugger currently has no means to check the address for alignment, so the behavior is undefined.

Below are some example usages of the above functions.

Example 1: Initialization

```
accum_t  acc = to_accum(0);          //acc = 0x00-00000000-00000000
int i = 0x1347845D;
acc = to_accum(i);                   //acc = 0x00-1347845D-00000000

int i = 0x8378239D;
accum_t  acc = to_accum(i);          //acc = 0xFF-8378239D-00000000
long long j = 0x7459024A-09567FEA;
accum_t  acc = llto_accum(j);        //acc = 0x00-7459024A-09567FEA
```

Example 2: Fractional multiplication example with various ways of using the above functions.

```
int coef = 0xDE7F3456;
int img = 0x125690EF;
accum_t acc = fract_mult(coef, img);  //acc = coef×img×2
                                      //acc = 0xFF-FB333AE3-CE2C7894
```

Case 1:
```
int out1 = rnd_saturate(acc);  //out1 = 0xFB333AE4
                               //If round/saturate bits are set
```

Case 2:
```
int out2 = extract_high(acc);  //out2 = 0xFB333AE3
                               //Extracts higher 32-bit of acc
```

Case 3:
```
int out3 = extract_low(acc);   //out3 = 0xCE2C7894
                               //Extracts lower 32-bit of acc
```

Case 4:
```
int out4 = extract_ext(acc);   //out4 = 0xFFFFFFFF
                               //Extracts the overflow bits
```

Case 5:
```
accum_t acc = update_low(acc, img);
//acc = 0xFF-FB333AE3-125690EF
//Note that the lower bits are replaced by img
```

Case 6:
```
accum_t acc = update_ext(acc, coef);
//acc = 0x56-FB333AE3-CE2C7894
//Note that the overflow bits are replaced by the coef
```

Case 7: Assume after 'N' MAC operations, we get, `acc = 0xf2-93456212-A4B23612`

Saturation enabled:
```
long long p = saturate(acc);          //p = 0x80000000-00000000
```

Saturation disabled:
```
long long p = saturate(acc);          //p = 0x93456212- A4B23612
```

Mode Independent:
```
long long p = extract_long(acc);      //p = 0x93456212- A4B23612
```

Example 3:

```
long long Coef[100] = { .... };
//sine and cosine coefficients packed into 64-bit buffer
long long Data[100] = { .... };
//real and imaginary data packed into 64-bit buffer

int Csin, Ccos;
int Dr, Di;

//load the sine and cosine  coefficients in a single cycle
lldecompose(Coef[i], Csin, Ccos);

//load the real and imaginary data in a single cycle
lldecompose(Data [i], Dr, Di);

// real part of the product
accum_t Ar = fract_mult(Csin, Dr) + fract_mult(Ccos, Di);

//imaginary part of the product
accum_t Ai = fract_mult(Ccos, Dr) - fract_mult(Csin, Di);

//store the product in a single cycle
llcompose(rnd_saturate(Ar>>1), rnd_saturate(Ai >>1));
```

Note:

- In the last line of the above example, storing the scaled data is directly mapped to the "scaled" operation in the instruction. Thus the data is rounded, saturated, scaled and stored to the memory in a single cycle.
- After arithmetic operations, before using `llcompose` always use `rnd_saturate`.

2.5.2 Arithmetic Functions

2.5.2.1 Multiplication

For regular C types int and unsigned int, the regular * operator is supported. The result is again an `int` after wrapping and truncation. However to take advantage of the double precision output and fractional multiplication support on the LPDSP32, the intrinsic functions listed below are provided.

The difference between integer and fractional multiplication is that in the latter one the result is shifted up by one bit so that the result of `Q1.31` * `Q1.31` is aligned as `Q1.63`. The output (typically of type `accum_t`) always has full precision.

Function	Description
`accum_t fract_mult(int, int);`	fractional signed × signed multiplication
`accum_t fract_mult_su(int, unsigned);`	fractional signed × unsigned multiplication
`accum_t fract_mult_uu(unsigned, unsigned);`	fractional unsigned × unsigned multiplication
`long long long_mult(int, int);`	integer signed x signed multiplication
`long long long_mult_su(int, unsigned);`	integer signed × unsigned multiplication
`accum_t along_mult_uu(unsigned, unsigned);`	integer unsigned x unsigned multiplication with 72-bit result, with zero in extension word

The fractional functions above are of importance. The integer multiplications are selected automatically by the CHESS compiler based on rewrite rules.

Example 1: Rewrite rule

```
int a, b;
long long c = (long long)a * (long long)b;
                                    //mapped to long_mult(a, b);
```

Example 2:

```
int coef = 0xDE7F3456;
int img = 0x125690EF;
long long p = long_mult(coef, img);    //p = coef×img
                                       //p = 0xFD999D71-E7163C4A
accum_t acc = fract_mult(coef, img);   //acc = coef×img×2,
                                       //acc = 0xFF-FB333AE3-CE2C7894
```

2.5.2.2 Divide Step

The LPDSP32 has iterative divide step hardware for division support. There is a library function that uses this hardware for performing division, which the compiler maps when a division is called i.e., a/b maps to this function. Below is the intrinsic function for the divide step.

Function	Description
`accum_t div_step(accum_t x, accum_t y, uint3_t& sr);`	Non-restoring division step, i.e., `2x+y` or `2x-y` dependent on previous sign, and complement of new sign is added to LSB.

An example to divide two numbers with a zero `Q` format is available in the install folder of LPDSP32 processor (`$ToolInstallFolder\designs\lpdsp32\lib\`), in the files `lpdsp32_div.h` and `lpdsp32_div.c`.

Example 1: 32-bit division of positive numbers

```
unsigned div32_pos(accum_t x, accum_t y, unsigned& r)
//assuming that 32-bit dividend is in low word of 'x' and 32-bit
//divisor is in high word of 'y' both with zero extension
{
    uint3_t sr = 0;                 // clear N-bit
    for (int i = 0; i < 32; i++)
        x = div_step(x,y,sr);
    unsigned q = extract_low(x);    // quotient is in low word
    if (neg(sr)) x = x + y;         // restore remainder
    r = extract_high(x);            // remainder is in high word
    return q;
}
```

Example 2: To divide two fractional numbers with a `Qn.m` format

```
int div32_nr_by_dr_for_q (int nr, int dr, short qf)
{
    if(dr == 0)                 //validating the denominator
        return(nr);

    int sign = 0;               //set sign to  zero
    accum_t  num, den;          //numerator and denominator declarations

    //find the number of steps for "step divide"
    int steps = 31 - norm(to_accum(nr));

    sign = 1|(nr>>31);              //find the sign of numerator
    sign = sign*(1|(dr>>31));       //now decide the sign of the quotient
    num = to_accum(abs(nr));        //load the numerator(unsigned)
```

```
    num = num >> steps;
    den  = to_accum(abs(dr));      //load the denominator(unsigned)

    //initialize the status reg 'VNZ' flags
    //because the div_step will use these flags
    uint3_t sr = 0;

    //Run the step division for required Qn.m format
    //qf = n in Qn.m
    for(int i = 0; i < (steps + qf); i++)
        num  = div_step (num ,den ,sr);

    //Quotient will be accumulated in the low word
    //Return the quotient with the sign
    return (extract_low(num)*sign);
}
```

2.5.2.3 Normalization

A normalization function is provided that computes the number of sign bits (minus 9 to account for the 8 overflow bits) of an `accum_t` variable. Sign bits mean, the number of ones before the first zero in case of a negative number and number of zeros before the first one in case of a positive number. When the input is zero, zero is returned, when the input is minus one (all ones), 63 is returned.

Function	Description
`int norm(accum_t a);`	computes shift factor (to the left) to normalize the input `a`

Example:
After some operations, the contents of ax0 and the corresponding normalization results are shown below:

ax0	nrm(ax0)	Remark
0x00000000000000000	0	//zero
0xFFFFFFFFFFFFFFFFF	63	//-1
0x000000000004F55007	36	//pos. no.
0xFFFF80EF0600050100	8	//neg. no.
0x08F08C000000000700	-5	//pos. no.
0xC93367000000004500	-7	//neg. no.

2.5.2.4 Maximum, Minimum, Absolute Value

The following functions are provided for minimum, maximum, and absolute value, working on signed integers:

Function	Description
`int max (int, int);`	returns the maximum of two signed integers
`int min (int, int);`	returns the minimum of two signed integers
`int abs (int);`	returns the absolute value of an integer
`long long llmax(long long, long long);`	returns the maximum of two signed `long long` variables
`long long llmin(long long, long long);`	returns the minimum of two signed `long long` variables
`long long llabs(long long);`	returns the absolute value of a `long long` variable
`accum_t max (accum_t, accum_t);`	returns the maximum of two `accum_t` variables
`accum_t min (accum_t, accum_t);`	returns the minimum of two `accum_t` variables

`accum_t abs (accum_t);`	returns the absolute value of an `accum_t` variable

Since these functions have very efficient, single cycle execution support in the hardware, it is recommended to use them to improve efficiency.

2.5.2.5 Bit Set, Bit Reset, Bit Invert, Bit Test

The following functions can be used to set (`bitset`), reset (`bitrst`), invert (`bitinv`), or test (`bittst`) a bit in an integer variable. These functions can as well be applied to unsigned integer types (without any extra cost as the implicit signed/unsigned conversions have zero cost).

Function	Description

`int bitset(int, int i);`
`int bitrst(int, int i);`
`int bitinv(int, int i);`
`bool bittst(int, int i);`
	bit number `i` must be in range [0,31]

`long long llbitset(long long, int i);`
`long long llbitrst(long long, int i);`
`long long llbitinv(long long, int i);`
`bool llbittst(long long, int i);`
	bit number `i` must be in range [0,63]

`accum_t bitset(accum_t, int i);`
`accum_t bitrst(accum_t, int i);`
`accum_t bitinv(accum_t, int i);`
`bool bittst(accum_t, int i);`
	bit number `i` must be in range [0,63]

The CHESS compiler will automatically select these functions in case of bit-wise logical operations with manifest operands that are a power of two.

Example:

```
int a;
if ((a & 0x8) != 0) ...;          //mapped to bittst(a,3)
```

2.5.3 Special Addressing Modes

2.5.3.1 Cyclic Addressing

To do cyclic addressing on an array, following pointer update function can be used.

Function	Description
`T* cyclic_add(T* p, int i, T* start, int len);`	Compute `p+i` considering wrapping when exceeding the start address (when `i` is negative) or the end address i.e., `start + len` (when `i` is positive)

This function is available for any type `T`, both built-in C types and user defined types (C structs). The pointer offset (`i`) and length (`len`) arguments are scaled automatically based on `sizeof(T)`. The pointer offset may not exceed the array length. There are no alignment constraints on the allocation of the array in memory. So, cyclic addressing can be applied to any array, e.g., local arrays on the software stack or array struct members.

Example:

```
char chess_storage(DMA) buff_in[256];
char chess_storage(DMA) buff_out[256];
char *data_ptr;
char data_adr;
int adr_inc;
adr_inc = 3;
```

```
data_ptr=buff_in;
   for (int i = 0; i < 256; i++){
     data_ptr=cyclic_add(data_ptr, adr_inc, buff_in, 256);
     buff_out[i] = *data_ptr;
   }
```

2.5.3.2 Bit Reversal Addressing

Bit-reversal addressing is supported by the LPDSP32 hardware, which can be used with the following function:

Function	Description
`T* reverse_add(T* p, int Len/2, T* start);`	compute `p + Len/2` while propagating the carry from right to left instead of left to right

Again, this function is available for any type `T`, and the pointer offset `Len/2` is scaled automatically based on `sizeof(T)`.

Given a pointer `p`, the function returns the value in `p` incremented in bit-reversed order for an array of size `Len`, which must be a power of 2.

For bit-reversal addressing, the start (or base) address of the buffer must be aligned to a multiple of a power of two, being equal or larger than the size of the buffer. Suppose that the length of the bit-reversal buffer is `N`, then the start address must be a multiple of 2^n, with $N \leq 2^n$. This address alignment is done through the `chess_storage()` specifier.

Example: Address alignment

```
char chess_storage (DMA %1024) BitRevBuf[1024]
```

Example: To perform bit-reversed addressing on an array of length 128

```
int chess_storage(DMA % 128*sizeof(int)) A[128];
//Alignment  constraint  of  128*4

void copy_reversed(int B[])
{
   int* p = A;
   for (int i = 0; i < 128; i++)
   {
      *p = B[i];
      p = reverse_add(p,128/2,A);
   }
}
```

The compiler tends to automatically put circular buffers and bit-reversal buffers at the end of the memory map. It is advised to force the linker to put them at the beginning of the memory space.

2.5.3.3 Wide Pointer Offset

The pointer offset registers in the address generation units (AALU) of LPDSP32 are restricted to 18-bit signed values. This is sufficient for most applications. As long as the array sizes remain below 131072 Kbytes, no problem can occur. Also bigger arrays are typically handled correctly. Here further explanation is provided on how *wide* pointer offsets, i.e., pointer offsets exceeding the 18-bit signed range, after scaling, are handled by the compiler and simulator.

- When the pointer offset is a constant expression, known at compile time, the compiler will always compute the correct result (on the AALU when possible and on the wider ALU when needed).

Example:

```
struct X
{
   int A[32768]; // sizeof(A) == 131072
   int a;
   int b;
}
```

The compiler will correctly compute the address of `a` and `b` struct members (based on the start address of the struct), even when these offsets are *wide*.

- When the pointer offset is data-dependent (which is only allowed inside arrays in C), the compiler assumes that the offset fits in the 18-bit signed range, and will do the pointer addition on the AALU. However, the simulator will generate a run-time warning when moving a value exceeding the 18-bit signed range to an LPDSP32 18-bit AALU offset register.

  To force a pointer addition on the wider ALU, to do a pointer addition with a wide data-dependent offset, you can use following intrinsic function.

Function	Description
`T* wide_offset_add(T* p, int i);`	compute `p + i`, where `i*sizeof(T)` may exceed the 18-bit signed range

Again, this function is available for any type `T`, and the pointer offset `i` is scaled automatically based on `sizeof(T)`.

2.5.4 Controlling the Processor Mode

The following functions are provided.

Function	Description
`void set_round_bit(int b);`	sets round bit R (`b` = 0 or 1)
`void set_saturate_bit(int b);`	sets saturate bit S (`b` = 0 or 1)
`void disable_interrupts();`	sets IE bit to zero
`void enable_interrupts();`	sets IE bit to one
`void set_interrupt_mask(int m);`	sets interrupt mask register (`m` = 0 to 32767). A zero bit will discard any interrupt request
`int get_interrupt_mask();`	returns the value of the interrupt mask register
`int get_irq_stat();`	returns the value of the interrupt request status register
`void software_interrupt(int x);`	call interrupt routine number (`x` = 1..15), independent from mask register
`void core_halt();`	put core in power-down mode

2.6 Storage Qualifiers

By default global variables are allocated to memory `DMA`. Using `chess_storage()` annotations [2], global or static variables can also be allocated to `DMB` or `DMIO`. LPDSP32 supports parallel memory access using `DMA` and `DMB`.

Example:

```
int A[10];                          //same as int chess_storage(DMA) A[10];
int chess_storage(DMB) B[10];       //This array will be mapped in DMB
volatile int chess_storage(DMIO) C;   //This array will be mapped in DMIO
```

Note that 64-bit `long long` variables can only be allocated to the default memory `DMA` which supports 64-bit single cycle access.

Explicit pointer definition using the `chess_` directive is required, when using a pointer to data mapped in `DMB` memory.

Example:
```
int chess_storage(DMB) * pointer_B;
```

When passing a variable or pointer mapped to non-`DMA` memory space through a function argument, the function must be defined with the argument in the same memory space.

2.7 Mapping Variables to a Fixed Address

Variables can be assigned to a fixed address in the following two ways:

a) Using the `chess_storage`() specifier in the C code
From within the C code, assign absolute addresses to variables in a fixed memory space, using the `chess_storage`() specifier. The address being an unsigned integer value represented in decimal, octal or hexadecimal.

Example:
```
int chess_storage (DMA:155) xyz;
int chess_storage (DMIO: 0xC20000) check_var;
```

Especially when assigning variable names to I/O registers, this is very useful.

b) Mapping the symbol in the linker configuration file
The linker configuration file (<file_name>.bcf) can be customized as per the individual project needs [3].

Example:
```
C code:  int some_var
bcf file: _symbol some_var 84
```

Functionally, the methods are identical. It is advised to map the symbol in a linker configuration file, so that the source code need not be recompiled if any change is there in the address mapping.

2.8 Alignment of Data Types

Unlike some processors where unaligned memory access is supported, the LPDSP32 does not support unaligned access to memory.

```
char        ->   always aligned to an address of 1
short       ->   always aligned to an address of 2
int         ->   always aligned to an address of 4
long long   ->   always aligned to an address of 8
```

When declaring variables or arrays, the C compiler automatically takes care of the alignment constraints involved in placing the data in the memory. Memory allocation in the stack for local variables is also done automatically by the compiler. Since the stack space is defined in the linker configuration file (.bcf file), it is necessary to initialize the stack space which is aligned to the maximum word length supported by the processor i.e., 8 words for LPDSP32.

Example:
```
_stack DMA 0x4000 8192
```

The above statement tells the processor that the stack space starts from address `0x4000` (i.e., address `16384` in `DMA`) and is of length 8192 bytes.

2.9 I/O interface

Data exchange with external devices or peripherals is implemented through I/O memory mapped variables. Such I/O variables need to be declared as `volatile` and their scope has to be global. This informs the compiler that their values can be changed by some external device, not under control of this code.

Example:

```
volatile int chess_storage(DMIO:0xC50100) in_data_perix;
volatile int chess_storage(DMIO:0xC50104) out_data_perix;
int rx_data, tx_data;
rx_data= in_data_perix;
out_data_perix = tx_data;
```

2.10 Interrupts

The interrupt functions or interrupt service routines can also be described in C. The chess directive `property(isr)` must be used with the function heading to inform the compiler that the function is an Interrupt Service Routine. This directive also ensures that a return from interrupt is executed at the end of the function. The compiler automatically saves the status register and all the other registers (except hardware loop registers) when entering the routine and restores them back when exiting the routine.

When there are function calls from inside an interrupt service routine and if the functions have `for/while` loops, the compiler might use hardware loops. In order to inform the compiler to use the software loop, `for` statement inside the functions should be followed by a `chess_no_hw_loop` property Note that the compiler will warn when calling an unannotated function in an ISR. Further explanation about interrupts and hardware loops is given in Section (§ 2.14.1).

Example 1:

An example for loop used in a function called from within the ISR.

```
for (int i = 0;i< 16;i++)  chess_no_hw_loop
{
   fifo[i] = *ptrOut++;
}
```

Example 2:

The function below is `ISR_process_samples`() and returns a void argument. The symbol to be used in the `lpdsp32_init.s` file is <`isr_function_name`>, in this case, `ISR_process_samples`.

```
extern "C" void ISR_process_samples() property(isr)
{
    count++;
    left_out = *pcm_buf;
    pcm_buf = cyclic_add(pcm_buf,1,PCMOutputBuffer,BUFFER_SIZE);
    right_out = *pcm_buf;
    pcm_buf = cyclic_add(pcm_buf,1,PCMOutputBuffer,BUFFER_SIZE);
}
```

The corresponding assembly initialization file `lpdsp32_init.s` then becomes:

```
// $Id: lpdsp32_init.s
//initialization before entering the main function
.text global 0 _main_init
   r = 1                      //enable rounding
   s = 1                      //enable saturation
   sp = _sp_start_value_DMA   //init SP (adjusted to stack in lpdsp.bcf)
   ie = 1 ; nop               //enable interrupts

//area to load main() arguments
.bss global 0 _main_argv_area DMA 256

//define the ISR vector to corresponding ISR
.undef global text ISR_process_samples

//the interrupt vector table with 15 interrupts
.text global 0 _ivt
   jp _main_init                  //0  - reset
   reti ; nop                     //2  - interrupt 1
   jp ISR_process_samples         //4  - interrupt 2
   reti ; nop                     //6  - interrupt 3
   reti ; nop                     //8  - interrupt 4
   reti ; nop                     //10 - interrupt 5
   reti ; nop                     //12 - interrupt 6
   reti ; nop                     //14 - interrupt 7
   reti ; nop                     //16 - interrupt 8
   reti ; nop                     //18 - interrupt 9
   reti ; nop                     //20 - interrupt 10
   reti ; nop                     //22 - interrupt 11
   reti ; nop                     //24 - interrupt 12
   reti ; nop                     //26 - interrupt 13
   reti ; nop                     //28 - interrupt 14
   reti ; nop                     //30 - interrupt 15
```

2.11 Inline Functions

When a function is inlined, processor cycles used for pushing variables onto the stack can be saved. Hence it is recommended to use inlining when the function size is small and it is not used often.

When the inline function is called many times in a code, the program memory size will increase since the inline code will be replicated every time it is called.

To inline a function, use the keyword `inline` before the function definition. If the function is called in one source file only, write the definition in this source file. If this function is going to be called in many source files, then write the function definition in a header file and include this header file in the corresponding source files.

Note: For Microsoft Visual C++, the `__inline` keyword is available in both C and C++, but the `inline` keyword is available only in C++. During native simulation (compiling and executing the application source code on your host machine with a regular C/C++ compiler), since the projects are compiled with the `/TP` switch, either of the syntaxes can be used. To maintain uniformity we suggest the use of only the `inline` keyword.

It is also possible to inline assembly functions within the C program, similar to inlining C code functions.

Example: Inlined assembly code for `powerdown`
This example has limited functionality, no list of registers to save in the `clobbers` [2].

```
inline assembly void core_halt()
clobbers() property(volatile functional loop_free)
{
    asm_begin
    powerdown; nop
    asm_end
}
```

2.12 C Library Support

Chess provides support for most standard C headers, as discussed in,([2]§3.3). In addition, on LPDSP32, following functions from <`math.h`> are supported.

Single Precision	Double Precision
`float ceilf(float);`	`double ceil(double);`
`float floorf(float);`	`double floor(double);`
`float truncf(float);`	`double trunc(double);`
`float roundf(float);`	`double round(double);`
`float fabsf(float);`	`double fabs(double);`
`float ldexpf(float, int);`	`double ldexp(double, int);`
`float frexpf(float, int*);`	`double frexp(double, int*);`
`float copysignf(float, float);`	`double copysign(double, double);`
`float cosf(float);`	`double cos(double);`
`float sinf(float);`	`double sin(double);`
`float expf(float);`	`double exp(double);`
`float logf(float);`	`double log(double);`
`float log10f(float);`	`double log10(double);`
`float sqrtf(float);`	`double sqrt(double);`
`float powf(float, float);`	`double pow(double, double);`

2.13 C-Application Design Flow

Develop the fixed point C code for the application. Compile it with the C compiler and run it on the native platform to make sure it is working as expected ([2], §3.2 Native compilation of target-specific C/C++ source code).

- Wherever possible use the appropriate types, operations, etc. as outlined in this chapter
- Replace all dynamic memory allocations with static allocations.
- To increase performance, replace all the appropriate code with intrinsic functions.
- Apply source code annotations to provide the compiler with more information to increase performance.
- Use storage qualifiers (memory specifiers) to explicitly assign some arrays or scalars to `DMB` memory.
- Check the results of this modified version with that of the fixed point version using a set of test cases. Define an acceptance criteria for the differences in the results and proceed to native compilation.

- Native compilation is performed using the standard C++ compiler by including the library which supports the LPDSP32 data types, operations and intrinsic functions. Since the simulation time using the CHESS compiler is much longer as compared to native compilation, the latter can be used to quickly simulate and debug the functionality of the C code.

- For native compilation, follow these steps:
  - Add `lpdsp32_native.c,.\isg\lpdsp32_chess_opns.c` and `.\isg\lpdsp32_chess_types.c` to the C++ application project. These files are located in: `$ToolInstallFolder\designs\lpdsp32\lib\`
  - Include the CHESS compiler processor header file `lpdsp32_chess.h`, using an include statement at the beginning of some central header file or use the include file option of the C++ compiler (eg. /FI option i.e Force include file option in Microsoft Visual C++). By including this file the functions to emulate the LPDSP32 are automatically called. This file will include the other required header files located in the following directories: `$ToolInstallFolder\chessdir\`, `$ToolInstallFolder\designs\lpdsp32\lib\` and `$ToolInstallFolder\designs\lpdsp32\lib\isg`.
  - Use the /TP switch to compile C code as C++.
  - If exact-width C types such as `int32_t`, `int64_t` are used in the application source files, include the `stdint.h` file to the project.

- During native compilation, there is no need to remove chess-specific directives like `chess_storage`() from the code. They are defined as nothing in the `chess.h` file except for the `restrict` keyword [2].

- At this point check the results of the test cases according to the criteria set before; proceed to compilation and simulation on the LPDSP32 compiler and simulator.

- Open the CHESSDE and create a new project, and include all the source code files into it. Run CHESS compiler to compile the application and use CHECKERS simulator to simulate it. This is a bit-true and cycle-true simulation. Compare the results of the test cases – they should be identical to the results of the native simulation. Generate the profiling data for further analysis.

- Use DSP optimisation techniques to reduce the cycle count and the memory usage (§2.14). Also analyze using alternative algorithms for sub-modules

- After trying out these changes, the application can still be simulated and compiled with the C compiler on the native platform, to debug the function, and then same can be run on the LPDSP32 simulator to check for the cycles/memory reductions.

- C compiler directives allow code optimizations, while remaining completely within the unified C language environment, hence they should be used wherever possible.

- As far as possible write the code for the LPDSP32 in ANSI C. Use assembly language programming only when absolutely necessary.

2.14 Optimization Techniques

In this section different optimization techniques to generate efficient code are described. By applying these techniques the required number of cycles and the program memory can be reduced.

Before optimizing, it is important that the programmer understands the processor architecture including its advantages/ limitations, the instructions supported by the processor, ANSI-C code etc.

1. Some of the limitations of the processor are given below :
   - There is a single ALU unit (shift operators, logical operators etc).
   - There is a dual adder for the accumulator (only add/sub and multiply from the second ALU).
   - The dual load/store will work only with 32-bit data types.
   - The 64-bit load/store works only on the `DMA` address space.
   - Although there are two multipliers, only one of them supports unsigned multiplications.
   - Output of the multiplier is always placed in an accumulator. (i.e., `a*b*c` may result in inefficient code if not written properly).

   Note: By 'limitations', we mean the properties of the processor. The programmer should be aware of the above so that he can design the C code with the above architecture in mind.

2. Some of the ISA points to be understood are listed below :
   - Load/Store instruction execution with the pipeline.
   - Relative jumps and absolute jumps.
   - Delayed branch and calls.
   - Arrangement of instructions in the 40-bit program memory.
   - Types of instructions available in short and long instructions.

3. Some general comments for generating good assembly code are listed below. The best way to accomplish it would be to get to know the minor C details that affect the optimizations.
   - If variables/pointers remain constant, declare them using the keyword `const`. The compiler will be free to store these anywhere.
   - Declare the scope of function and variables properly. Do not make the scope global un-necessarily, as that will hurt the optimization.
   - Use `restrict` keyword when necessary, this will help the compiler to generate better assembly code for independent pointers [2].
   - Use `chess_xxxxx` qualifiers when necessary. This will help the programmer to tune the code for the requirements [2].
   - Take a look at the compiler specific options. This will help in advanced programming situations, e.g. at times for a specific file, one would like the compiler not to generate a specific optimization/instruction class. So by setting the properties for the compiler for the specific file/project one can generate code, fine tuned.
   - In all situations, one MUST avoid writing code that is not defined by ANSI-C,
     `e.g., int x = y >> (-5)`, this is a negative shift and is not defined in ANSI-C.
     The compilers may not flag any errors for this code, but may result in different outputs for different platforms /compilers.
   - Another example of the above non-ANSI C code would be as below :
     `*ptr_inp = (*ptr_inp++) + *ptr_inp;`
     On LPDSP -> `ptr_inp[0] = ptr_inp[0]  + ptr_inp[1];`
     On VC -> `ptr_inp[0] = ptr_inp[0]  + ptr_inp[0];`
     Depending on its precedence of evaluation, each compiler can generate its own version of the code as the ANSI C has not defined the rule for above case.

With the above points as a start, the programmer may get a brief idea about how to write good code and where to be cautious. The above points are not comprehensive and they are not a substitute to the architecture manual, ISA manual, Chess C compiler manual and the ANSI-C standard.

Some points below give more specific examples of generating good quality code.

- If constants which need more than 24 bits are used in the code, for example in a comparison operation, then they will be stored in `DMA`, occupying one location. This is a limitation of the instruction set. This will be an unintended usage of `DMA` during their mapping in memory. Also the load for this data uses direct addressing mode (long instructions) and can cost extra cycles especially inside a loop. A solution is, declare such data as constants using the keyword `const` and access them using pointers (indirect access).

Example:

```
/* original code */
#define VAR1 = 0x12345678;
#define VAR2 = 0x76843219;
.............
{
    int x;
    ...........
    if(x==VAR1)
      x++;
    .............
    if(x==VAR2)
      x--;
   ............
  }

/* modified code */
/* constant declaration & indirect access using pointers*/
const int constArray [] ={ 0x12345678, 0x76843219 };
.............
{
  int x;
  int *dataptr;
  dataptr = (int *) &constArray[0];
  .............
    if(x==*dataptr++)
      x++;
    .............
    if(x==*dataptr++)
      x--;
   ............
  }
```

- Try to get the linker to put bit-reversal buffers at the beginning of the memory space of your application. This way it will be simpler to place the other address independent data anywhere in the application's space.

- The size and start address of the stack are defined in the .bcf file that should be included in every project. An example of the linker configuration file is given in Appendix A.
  It contains the line,

```
_stack DMA 0xe000 8184
//stack region - 8184 bytes stored at addresses 0xe000 to FFF8
```

  The above definition sets the stack size to 8184 bytes, which may be way too large for most applications. The maximum stack size of the application can be found out by generating the *.`calltree` file (using the dump call tree option of the linker), where the maximum number of calls and stack depth information can be seen. Also in the ISS, the stack size used in the current simulation can be monitored in the "Statistics" tab. When the stack size is set smaller than the needs of the application in the current simulation, the simulator generates an error as and when an overrun occurs, but the hardware JTAG debugger currently has no means to check the stack overrun. Note that the stack start address MUST be aligned to 64-bit memory. This is because the compiler allocates the variables on the stack assuming the above while entering any function.

- Normally `accum_t` variables are used only in a small scope. If the scope of the `accum_t`

variable is very large, then it may be stored in the stack or memory. Avoid such instances (storing of `accum_t` variables in memory or on stack) as they will occupy ~ 9 bytes of stack excluding the alignment for the next memories and will need at least 2 cycles for load/store. If possible convert them to `int` or `long long` so that a save can be avoided or be cheaper and faster.

- Small function code can be inlined for more efficiency. This may increase the program memory, but this will save ~4 to 6 instructions before and after the call for the register/context restoring excluding the data storage in the register for the function call convention.

- It is better not to use more pointers than the available number of address generation registers – 8 for `DMA` memory and 4 for `DMB` memory. Otherwise, the complete address (base, step and modulo registers) have to be saved to the stack and again restored back later which will consume a lot of processor cycles.

- Try to reduce memory loads and stores as much as possible by rewriting the C code in such a way that data such as coefficients or other elements are re-used efficiently after being loaded into registers.

- If possible rewrite the code in loops such that a store instruction is not followed by a load instruction. This constraint is to eliminate the store -> load hazard of the compiler. (Refer the instruction pipeline of load/store instruction for more details). If a store instruction is followed by a load instruction, a `nop` is introduced (if no other useful instruction was found) by the compiler. By designing a software pipeline in the code, it may be possible to avoid the hazard.

  Example:

```
/* original code */
.............
for (int i=0;i<N;i+=2)
{
   acc_0 += fract_mult(coef[i], data[i]);
   acc_1 += fract_mult(coef[i+1], data[i+1]);
   out[i] = rnd_saturate(acc_0 + acc_1);
   out[i+1] = rnd_saturate(acc_0 - acc_1);
}
.............
.............

/* modified code for better pipelining between store/load */
.............
int iCoef = coef[0];  // do a pre-load before entering loop (DMB)
int iData = data[0];  // do a pre-load before entering loop (DMA)
for (int i= 0;i<N;i+=2)
{
   acc_0 += fract_mult(iCoef, iData);
   iCoef = coef[i+1];  int iData = data[[i+1];
   acc_1 += fract_mult(iCoef, iData);
   iCoef = coef[i+2];  int iData = data[[i+2];
   out[i] = rnd_saturate(acc_0 + acc_1);
   out[i+1] = rnd_saturate(acc_0 - acc_1);
}
.............
.............
```

  Sometimes it may be possible that the compiler itself will be able to do the above optimization without any re-writing of the code. Note that the above code will result in one extra memory load.

- The usage of compiler directives is very important to generate optimized assembly code. A number of features are supported by the compiler by means of directives [2].

- Write the code in a way that helps the compiler to do register allocation better and access arrays at the appropriate scope. For example if an element `buff[m]` is going to be used in a loop and `m` changes only outside the loop, then explicitly preload `buff[m]` to a scalar variable and then use the variable inside the loop. The compiler will not miss the possibility of preloading, but the programmer doing it explicitly seems to help it choose a better register usage strategy.

  Example:

```
.............
.............
for(int i=0;i<N;i++)
{
   int iCoef_tmp = coef[i];
   for(int j = 0;j<M;j++)
   {
      //acc += long_mult(coef[i],inp[j]);
      acc += long_mult(iCoef_tmp,inp[j]); //may result in better code
   }
}
.............
.............
```

- To place some constant data in the memory the keyword `const` is used, for instance with tables that need to go into ROM. When using pointers to access data from such a table, the keyword `const` has to be repeated so that the compiler can place the load instruction optimally.

  Example:

```
const int chess_storage(DMB) SineCoef[32] = {...};
//int *ptrCoef;                          //regular code
const int chess_storage(DMB) *ptrCoef;  //recommended code
ptrCoef = &(SineCoef[index-1]);
```

2.14.1 Optimization of Loops

LPDSP32 supports four nested levels for hardware do loops. Hence, the compiler can translate up to four nested levels of `for/while` loops in the C code into efficient hardware do loops. Since there are none reserved for interrupts, the ISR should be written such that the compiler does not use the loop registers, instead uses the non-hardware (i.e. software) loop for code looping, else the programmer should take certain precautions as explained the remarks below.

The compiler assumes that the upper bound of the loop is not larger than 65535(that it fits into the 16-bit loop count register). It is the programmer's responsibility to make sure that the upper bound is not too large. If the programmer is not sure that the upper bound will not exceed 65535, the compiler can be informed not to use a hardware loop by using the `chess_loop_range(N,M)` property (`M` is the upper bound, `N` is the lower bound) to specify a value higher than 65535 for the upper bound of the loop([2], § 4.1.6.1 Loop Count Annotation). The actual value used for `M` does not actually matter, as long as it is higher than 65535. For manifest values larger than 65535, the compiler automatically uses a software loop, and issues a warning.

Note that during execution the simulator stops with an error for an invalid loop value (exceeding the valid range [1,65536]). Such a check is not present in the hardware JTAG debugger.

The source code annotation `property(loop_levels_N)` can be used with functions to inform the compiler as to how many hardware loops are being used by the function. It also informs to compiler

the number of hardware loops it should restrict itself to inside the function. If this annotation is not present for a function and it is called inside a loop, then the compiler will assume the worst case, that all the hardware loops are being used inside the function and will implement the current outer loop with a non-hardware(software) loop.

`N` has to be replaced by the actual value. If `N = 0`, it can be specified using `property(loop_free)` i.e., `property(loop_levels_0) = property(loop_free)`

Specifying the `chess_loop_range(N,M)` property wherever possible will help the compiler to decide whether to generate the initial test to check the loop count (it skips the loop if loop count is zero) or not.

The compiler tries to generate code as efficient as possible by using software pipelining. ([2], § 4.1.6.2 Prepare for Software Pipelining).

Some remarks:

1. The compiler will not use hardware loop `lp` instruction in the following cases:
   - If the for/while variable is not an `int` variable

     Example:
     ```
     short i;
     for(i = 0; i < 5; i++)  // lp is not used
     ```

   - If the `for/while` loop condition involves a volatile variable

     Example:
     ```
     volatile int loopMaxCnt;                /* Global volatile variable */
     void funcA(void)
     {
        volatile int loopMaxCntX;            /* Local volatile variable */
        ....
        ....
        for(int i = 0; i < loopMaxCnt; i++) // lp is not used

        /* or */

        loopMaxCntX = 25 * 10 + loopMaxCnt;  /* could be anything */
        for(int i = 0; i < loopMaxCntX; i++) // lp is not used
     }
     ```

   - As discussed above, if the `for/while` loop calls a function and the function declaration doesn't have loop levels property (`property(loop_levels_N)` or `property(loop_free)`)

     Example:

     ```
     void SomeOtherFunction(void);
     void funcA(void)
     {
        for(int i = 0; i < 10; i++)  // lp isn't used
        {
            SomeOtherFunction();
        }
     }
     ```

2. In the case of nested function calls or nested for/while loops, the compiler will use the hardware loops starting from the inner most for/while loop. Once all the four hardware loops get used, the

compiler implements the remaining loops with non-hardware(software) loops.

- Nested function calls

  Example:
  Consider an example with nested function calls, where each function has one loop and one function call.

```
/*original code*/
void func1(void)property(loop_levels_1);
void func2(void)property(loop_levels_1);
void func3(void)property(loop_levels_1);
void func4(void)property(loop_levels_1);
void func5(void);
int counter;

void func5(void){
   for(int i=0; i<10; i++) counter++;
}

void func4(void){
   for(int i=0; i<10; i++)   func5();
}

void func3(void){
   for(int i=0; i<10; i++)   func4();
}

void func2(void){
   for(int i=0; i<10; i++)   func3();
}

void func1(void){
   for(int i=0; i<10; i++)   func2();
}

void main(){
     func1();
}
```

  Here even though programmer's intention of using `property(loop_levels_1)` for every function is to use one hardware loop for each of the outer four functions, compiler will detect the nested function calls and hardware loops are not used.
  So, the programmer needs to know the depth of the loop nesting and use `property(loop_levels_x)` with an appropriate value for each function, (`x`=0 to 4).

  Modified code to use hardware loops for the four outer functions:

```
/*modified code*/
void func1(void)property(loop_levels_4);
void func2(void)property(loop_levels_3);
void func3(void)property(loop_levels_2);
void func4(void)property(loop_levels_1);
void func5(void)property(loop_levels_0);
```

3. Compiler can decide to choose a hardware loop based on the `chess_property default_loop_range_maximum`, redefined either in local C file or in a central header file of

the project. This property indicates the maximum value the loop count can reach, whenever the loop count is not known at compile-time. It is used by the compiler to decide whether it is safe to use a hardware loop or not. By default the `default_loop_range_maximum` is set to 65535 in the `lpdsp32_chess.h` file.

When the following setting is done:

```
chess_properties { default_loop_range_maximum : 65537; }
//any number higher than 65535(2^16-1)
```

If the loop count is known at compile-time: the compiler will select hardware loop, when loop count is less than 2^{16}. If the loop count is not known (data-dependent), then it considers the `default_loop_range_maximum` parameter and since the maximum possible value exceeds the unsigned integer 16 bit range (0 to 65535), hardware loop will be not be used.

The above line can be added directly in a C file or in a header file which will be included in the required C file. If entire project needs to use the header file then we can add the header file to `Project_settings → C-front-end → Always include files`

4. If the function is an Interrupt Service Routine(ISR) i.e., the function has `property(isr)`, the compiler does not use hardware loops for any loop. However, if the property is changed to `property(isr loop_levels_N)`, then the compiler uses hardware loops for such loops which do not fall into any of the above mentioned cases.

   Note: `property(isr)` source code annotation indicates that the function is an ISR and the `property(isr loop_levels_N)` informs the compiler the number of loops that are being used in the ISR, similar to the source code annotation `property(loop_levels_N)` for functions.

5. The default zero-overhead loop levels used for normal functions and interrupt routines can be overwritten by the following statements (with `N` a non-negative integer) :

   ```
   chess_properties {
      loop_levels : N ;
      isr_loop_levels : N ;
   }
   ```

   These statements have to come at the start of every compilation module, but after including the file `lpdsp32_chess.h`.
   When changing the calling convention in this way, you have to take care that all source files and libraries are compiled with the same settings. The best way to obtain this is by using the `Always include files` option of the CHESS front end(`Project_settings → C-front-end → Always include files`), where you include the relevant headers in the correct order. Note that inconsistencies resulting from different default settings cannot be detected at compile-time, which makes this approach unsafe (as opposed to overriding the defaults locally for functions).

6. A possible case where a hardware loop in ISR may corrupt the program is shown below.

   Example:

   - Let all the 4 hardware loops be used. Normally when running various codes, this may be the case.

   ```
   for(int i = 0; i < 10; i++) {
     for(int j = 0; j < 10; j++) {
        for(int k = 0; k < 10; k++) {
           unsigned int loopCntLocal1 = loopCnt1;
           unsigned int loopCntLocal2 = loopCnt2;

            for(int l = 0;l < 10; l++) {
            /*At this point all 4 HW loops used*/
   ```

```
                loopCntLocal2 += loopCntLocal1;  /*Check point 1*/
              }
              loopCnt1 = loopCntLocal1;
              loopCnt2 = loopCntLocal2;
           }
        }
     }
```

- Consider a case where an interrupt occurs when the core is executing the code at `/*Check point 1*/` and inside the interrupt sub-routine, a separate function that has an hardware loop instruction is being called, as shown below:

```
void func1(void);

  extern "C" void isr1(void)  property(isr)
  {
      func1();
  }

  void func1(void)
  {
      unsigned int loopCntLocal1 = loopCnt1;
      unsigned int loopCntLocal2 = loopCnt2;
       for(int i = 0; i < 10; i++) {      /*Check point 2*/
          loopCntLocal2 += loopCntLocal1;
       }
       loopCnt1 = loopCntLocal1;
       loopCnt2 = loopCntLocal2;
       loopCnt2--;
  }
```

- When the code reaches `/*Check point 2*/`, core tries to use hardware loop instruction, but since all the hardware loops are already in use, the loop count pointer register, `LCP` will be 0. This will lead to a wrong execution.
  So, it is recommended that the programmer should make sure that any functions that are called in the interrupt context do not use a hardware loop instruction.

7. To overcome the problem mentioned in the above point, one possible solution is to write inline assembly code for saving the Hardware loop registers onto stack inside the ISR and restoring them while returning back from the ISR as shown below.

```
inline assembly void chess_isr_envelope_open()
   clobbers() property(volatile functional loop_free){
   asm_begin
      sp += -56
      sp[0] = lcp
      lcp = 0
      sp[4] = lc
      sp[8] = lstk
      sp[12] = lpa

      lcp = 1
      sp[16] = lc
      sp[20] = lstk
      sp[24] = lpa

      lcp = 2
      sp[28] = lc
```

```
      sp[32] = lstk
      sp[36] = lpa

      lcp = 3
      sp[40] = lc
      sp[44] = lstk
      sp[48] = lpa

      lcp = 4
   asm_end
}

inline assembly void chess_isr_envelope_close()
   clobbers() property(volatile functional loop_free){
   asm_begin
      lcp = 3
      lpa = sp[48]
      lstk = sp[44]
      lc = sp[40]

      lcp = 2
      lpa = sp[36]
      lstk = sp[32]
      lc = sp[28]

      lcp = 1
      lpa = sp[24]
      lstk = sp[20]
      lc = sp[16]

      lcp = 0
      lpa = sp[12]
      lstk = sp[8]
      lc = sp[4]

      lcp = sp[0]

      sp += 56
   asm_end
}
```

These envelope functions are called automatically at beginning/end of every ISR.

For other directives and more details, please refer to the Chess Compiler User Manual [2].

Appendix A. Linker Configuration file

A default linker configuration file (`lpdsp32.bcf`) is included with the compiler installation in the `$ToolInstallFolder\designs\lpdsp32\lib\` folder. This file must be copied and customized according to the needs of the project. The modified .bcf file location should be mentioned on CHESSDE ->`Project settings->Linking->Linker configuration file.` If no settings are done, then the default configuration file will be used. In the default file the entry point for the program is set (which is present in the initialization file), a stack is allocated and the use of `argv, argc` arguments is enabled.

```
// file lpdsp.bcf
// central link file with default link constraints
_symbol _ivt 0          // interrupt vector table at PM 0
_entry_point _ivt
_symbol _main_init _after _ivt
_symbol _main _after _main_init

_stack DMA 0xe000 8184  //stack region in DMA (start_address size).
                        //SP is initialised to (start_address + size),
                        //which must be multiple of 8
                        //(to have aligned long access).

// include to use main(argc,argv) arguments
_always_include _main_argv_area
```

For a more detailed explanation please refer the Bridge Linker User Manual [3]

Appendix B. Initialization Code

A default initialization code written in assembly, `lpdsp32_init.s`, is included with the compiler installation in the `$ToolInstallFolder\designs\lpdsp32\lib\` folder. This file must be copied to your project directory, adapted and added to your compilation project. If not the default initialization file will be used. This initialization code is then first executed before the control is transferred to the main C function. Typically, the rounding and saturation control bits are set, interrupts are enabled, the stack is initialized according to the stack definition in the .bcf file, followed by the Interrupt Vector Table(IVT). This functionality can be modified according to the project requirements.

```
// $Id: lpdsp32_init.s
//initialization before entering the main function
.text global 0 _main_init
   r = 1                      //enable rounding
   s = 1                      //enable saturation
   sp = _sp_start_value_DMA   //init SP (adjusted to stack in lpdsp.bcf
   ie = 1 ; nop               //enable interrupts

//area to load main() arguments
.bss global 0 _main_argv_area DMA 256

//the interrupt vector table with 15 interrupts
.text global 0 _ivt
   jp _main_init              //0  - reset
   reti ; nop                 //2  - interrupt 1
   reti ; nop                 //4  - interrupt 2
   reti ; nop                 //6  - interrupt 3
   reti ; nop                 //8  - interrupt 4
   reti ; nop                 //10 - interrupt 5
   reti ; nop                 //12 - interrupt 6
   reti ; nop                 //14 - interrupt 7
   reti ; nop                 //16 - interrupt 8
   reti ; nop                 //18 - interrupt 9
   reti ; nop                 //20 - interrupt 10
   reti ; nop                 //22 - interrupt 11
   reti ; nop                 //24 - interrupt 12
   reti ; nop                 //26 - interrupt 13
   reti ; nop                 //28 - interrupt 14
   reti ; nop                 //30 - interrupt 15
```

In the default initialization code since no interrupt is defined, any hardware/software interrupt results in a return from the ISR vector.
To define an interrupt:

- use `jp <label>` in the interrupt vector table and define this function to be `extern` as this function may not reside in the same file.
- use this label for the function describing this interrupt.

Example:
In the initialization file:

```
.undef global text _isr_pcm
jp _isr_pcm             // 4  - interrupt 2
```

In the C file having the ISR routine:

```
extern "C" void _isr_pcm(void) property(isr)
```

The native simulation will not contain the above code since this is specific to LPDSP32, thus it is built such that on start up the round and saturation bits are enabled. To maintain the same behavior in native compilation it is recommended to enable them in the C code rather than the initialization code.

Appendix C. Examples

C.1 Dual MAC Operation

Below is an example to use the dual MAC operation of LPDSP32 and the corresponding generated assembly code.

C-code:

```
int chess_storage(DMIO:0xD01000) output_port;
static int x_1[6];
static int chess_storage(DMB) y_2[6];

void dual_mac(void)
{
    int i, out_D;
    accum_t ansA_72 = to_accum(0);
    accum_t ansB_72 = to_accum(0);
    accum_t ansC_72;

    for(i=0;i<5;i+=2)
    {
       ansA_72 += fract_mult(x_1[i],y_2[i]);
       ansB_72 += fract_mult(x_1[i+1],y_2[i]);
       ansA_72 += fract_mult(x_1[i],y_2[i+1]);
       ansB_72 += fract_mult(x_1[i+1],y_2[i+1]);
    }
    ansC_72 = ansA_72 + ansB_72;
    out_D = rnd_saturate(ansC_72);
    output_port = out_D;
}
```

Assembly code:

```
.text global 2 void_dual_mac
c0 = 4; axs0 = zero
a0 = __test1_x_1
a4 = __test1_y_2
lp 2 4
bx0 = ax0 + 0; ra1 = [a0+c0]; rb0 = [a4+c0]
bx0 = bx0+ra1*rb0; ra0 = [a0+c0]; rb1 = [a4+c0]
ax0+= ra0 * rb0; bx0+= rb1 * ra1; ra1 = [a0+c0]; rb0 = [a4+c0]
ax0+= ra0 * rb1; bx0+= rb0 * ra1; ra0 = [a0+c0]; rb1 = [a4+c0]
ax0+= ra0 * rb0; bx0+= rb1 * ra1
ax0 = ax0+ra0*rb1
ax0 = bx0 + ax0; retdb
 [13635584] = axs0
.label void_dual_mac__end last
nop; nop
```

C.2 Normalization

This example shows the use of normalization function.

Without using built in normalization function:

```
#define EXP30 0x400000001
long norm32bit(long long acc, int *nrm)
{
    short exp;
    long answer_w;

    answer_w = (long)acc;
    exp = 0;
    if (answer_w == 0l)
    {
      exp = 0;
    }
    else if (answer_w >= 0)
    {
       while (answer_w < EXP30)
      {
        answer_w = answer_w << 1;
        exp++;
      }
    }
    else
    {
      while (answer_w >= -EXP30)
      {
        answer_w = answer_w << 1;
        exp++;
      }
    }
    *nrm = exp;
    return answer_w;
}

void main(void)
{
    int nrm;
    long long Data = 0xF8l;
    int Val = (int )norm32bit(Data,&nrm);
}
```

Using built in normalization function:

```
int Val;
void main(void)
{
    long Data = 0xF8l;
    int nrm = norm(to_accum(Data));
    Val = extract_high(to_accum(Data) << nrm);
}
```

C.3 Bit Reverse Addressing

This example shows the use of bit reverse addressing mode of LPDSP32.

Without using bit reverse addressing:

```
int Real[8]={0};
int Imag[8]={0};
int *RevDataPtr;

int main(void)
{
    int NumBits=0,i,rev,index;
    int NumSamples = 8;
    int sample;
    while(!(NumSamples & (1 << NumBits)))
    NumBits++;
    for (sample = 0; sample < NumSamples; sample++)
    {
      index = sample;
      for (i = rev = 0; i < NumBits; i++)
      {
        rev = (rev << 1) | (index & 1);
        index >>= 1;
      }
      RevDataPtr = &Real[rev];
    }
    return 0;
}
```

Optimized using bit reverse addressing:

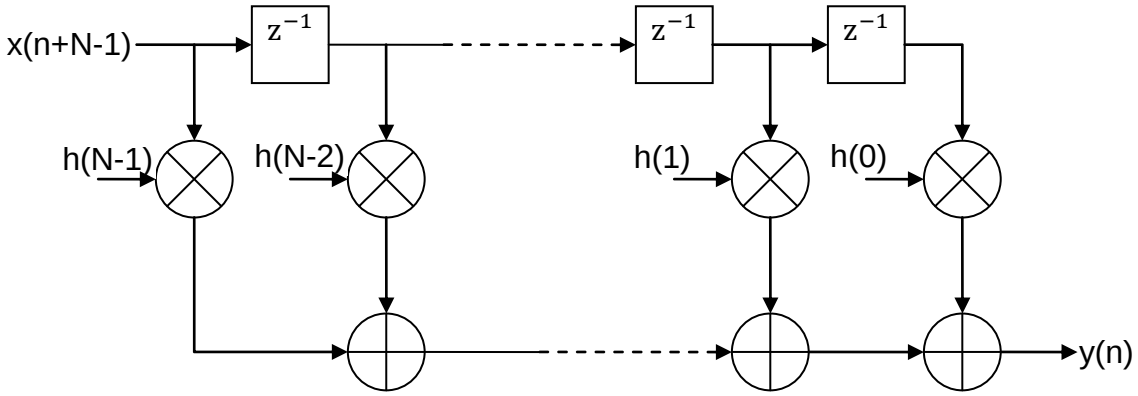
```
int chess_storage(DMA % 8*sizeof(int)) Real[8]={0}; // memory alignment
int chess_storage(DMA % 8*sizeof(int)) Imag[8]={0}; // memory alignment
int RevDataPtr[8];

int main(void)
{
    int N = 8;
    int *fftptr = Real;
    for ( int i =0; i< N; i++)
    {
      RevDataPtr[i] = *fftptr;
      fftptr = reverse_add( (int *) fftptr, N>>1, Real );
    }
    return 0;
}
```

C.4 FIR filter

The FIR filter of order $N-1$ is given by,

$$y[n] = \sum_{k=0}^{k=N-1} h[k] * x[n+k] \text{ for } n = \{0, 1, \cdots, NS-1\}$$

where NS is the number of input and output samples.



FIR Filter of Order (N-1)

The C-code for this FIR filter when the data and the coefficients are integers, is as shown below.

```
#define NS 256     //No. of samples
#define N 64       //No. of filter coefficients or No. of tap weights

int y[NS];         //Output Signal
int x[NS+N-1];     //Input Signal
int h[N];          //Filter coefficients or tap weights

void fir(int *y, int *x, int *h)
{
    for(int n=0; n<NS; n++)
    {
       long long sum = 0;
       for(int k=0; k<N; k++)
       {
          sum += x[n+k] * h[k];
       }
       y[n] = sum;
    }
}
```

If the data and the coefficients are in the fixed-point `Q1.31` format, then the multiplier output will be in the `Q1.62` format, which must be converted back to `Q1.31` format. Using the 72-bit accumulator, `fract_mult()` and the `rnd_saturate()` functions of the LPDSP32, the fixed-point real FIR filter can be coded as shown below.

Fixed-point initial C-code for porting on LPDSP32:

```
#define NS 256  //No. of samples
#define N 64    //No. of filter coefficients or No. of tap weights
```

```
int y[NS];      //Output Signal
int x[NS+N-1];  //Input Signal
int h[N];       //Filter coefficients or tap weights

void fir(int *y, int *x, int *h)
{
    for(int n=0; n<NS; n++)
    {
      accum_t sum = to_accum(0);
      for(int k=0; k<N; k+=2)
      {
        sum += fract_mult(x[n+k] , h[k]);
        sum += fract_mult(x[n+k+1] , h[k+1]);
        sum = to_accum(rnd_saturate(sum));
      }
      y[n] = extract_high(sum);
    }
}
```

The above code can be further optimized using the `chess` directives, cyclic addressing, dual load/store features of LPDSP32 as shown below.

Fixed-point optimized C-code for porting on LPDSP32:

```
#define NS 256    //No. of samples
#define N 64      //No. of filter coefficients or No. of tap weights

int chess_storage(DMB) y[NS];                          //Output Signal
int chess_storage(DMA %(sizeof(long long))) x[NS+N-1]; //Input Signal
//Filter coefficients or tap weights
int chess_storage(DMA %(sizeof(long long))) h[N];

void fir(int *y, int *x, int *h)
{
    int chess_storage(DMB) *p_y = y;
    int chess_storage(DMA) *p_x = x;
    int chess_storage(DMA) *p_h = h;
    int coef1, coef2;
    int dat1, dat2;

    for(unsigned int n=0; n<NS; n+=2) chess_loop_range(1,)
    {
      p_x = x + n;
      lldecompose(*((long long *)p_h), coef1, coef2);
      p_h = cyclic_add(p_h,2,h,N);
      lldecompose(*((long long *)p_x), dat1, dat2); p_x += 2;

      accum_t sum1 = fract_mult(dat1, coef1);
      accum_t sum2 = fract_mult(dat2, coef1);

      sum1 += fract_mult(dat2 , coef2);
      sum1 = to_accum(rnd_saturate(sum1));

      for(unsigned int k=2; k<N; k+=2) chess_loop_range(1,)
      {
        lldecompose(*((long long *)p_x), dat1, dat2); p_x += 2;
        sum2 += fract_mult(dat1, coef2);
```

```
        sum2 = to_accum(rnd_saturate(sum2));

        lldecompose(*((long long *)p_h), coef1, coef2);
        p_h = cyclic_add(p_h,2,h,N);

        sum1 += fract_mult(dat1, coef1);
        sum2 += fract_mult(dat2, coef1);

        sum1 += fract_mult(dat2, coef2);
        sum1 = to_accum(rnd_saturate(sum1));
      }
      lldecompose(*((long long *)p_x), dat1, dat2);
      sum2 += fract_mult(dat1, coef2);
      sum2 = to_accum(rnd_saturate(sum2));

      *p_y++ = extract_high(sum1);
      *p_y++ = extract_high(sum2);
    }
}
```

C.5 Complex FIR Filter

This example shows the FIR filter for complex data and complex filter coefficients. The real and imaginary parts of the complex data are stored in consecutive memory locations. The C-code for the complex FIR filter when the data and the coefficients are complex integers, is as shown below.

```
#define NS (256 * 2)  //No. of samples
#define N (64 * 2)    //No. of filter coefficients

int y[NS];            //Complex Output Signal
int x[NS+N-2];        //Complex Input Signal
int h[N];             //Complex Filter coefficients

void fir_cmplx(int *y, int *x, int *h)
{
    for(int n=0; n<NS; n+=2)
    {
      long long sum_re = 0;
      long long sum_im = 0;
      for(int k=0; k<N; k+=2)
      {
        sum_re += x[n+k] * h[k];
        sum_re -= x[n+k+1] * h[k+1];
        sum_im += x[n+k] * h[k+1];
        sum_im += x[n+k+1] * h[k];
      }
      y[n] = sum_re;
      y[n+1] = sum_im;
    }
}
```

If the real and imaginary parts of the data and the coefficients are in the fixed-point `Q1.31` format, then the multiplier output is in the `Q1.62` format, which must be converted back to `Q1.31` format. Using the 72-bit accumulator, `fract_mult()` and the `rnd_saturate()` functions of the LPDSP32, the fixed-point complex FIR filter can be coded as shown below.

Fixed-point initial C-code for porting on LPDSP32:

```
#define NS (256 * 2)  //No. of samples
#define N (64 * 2)    //No. of filter coefficients

int y[NS];            //Complex Output Signal
int x[NS+N-2];        //Complex Input Signal
int h[N];             //Complex Filter coefficients

void fir_cmplx(int *y, int *x, int *h)
{
    for(int n=0; n<NS; n+=2)
    {
        accum_t sum_re = to_accum(0);
        accum_t sum_im = to_accum(0);
        for(int k=0; k<N; k+=2)
        {
            sum_re += fract_mult(x[n+k], h[k]);
            sum_re -= fract_mult(x[n+k+1], h[k+1]);
            sum_im += fract_mult(x[n+k], h[k+1]);
            sum_im += fract_mult(x[n+k+1], h[k]);
            sum_re = to_accum(rnd_saturate(sum_re));
            sum_im = to_accum(rnd_saturate(sum_im));
        }
        y[n] = extract_high(sum_re);
        y[n+1] = extract_high(sum_im);
    }
}
```

The above code can be further optimized using the `chess` directives, cyclic addressing, dual load/store features of LPDSP32 as shown below.

Fixed-point optimized C-code for porting on LPDSP32:

```
#define NS (256 * 2)  //No. of samples
#define N (64 * 2)    //No. of filter coefficients

//Complex Output Signal
int chess_storage(DMA %(sizeof(long long))) y[NS];
//Complex Input Signal
int chess_storage(DMA %(sizeof(long long))) x[NS+N-2];
//Complex Filter coefficients
int chess_storage(DMA %(sizeof(long long))) h[N];

void fir_cmplx(int *y, int *x, int *h)
{
    int chess_storage(DMA) *p_y = y;
    int chess_storage(DMA) *p_x = x;
    int chess_storage(DMA) *p_h = h;
    int h_re, h_im;
    int dat_re, dat_im;
    for(int n=0; n<NS; n+=2) chess_loop_range(1,)
    {
        p_x = x + n;
        accum_t sum_re = to_accum(0);
        accum_t sum_im = to_accum(0);
        for(int k=0; k<N; k+=2) chess_loop_range(1,)
```

```
        {
            lldecompose(*((long long *)p_h), h_re, h_im);
            p_h = cyclic_add(p_h,2,h,N);
            lldecompose(*((long long *)p_x), dat_re, dat_im); p_x += 2;

            sum_re += fract_mult(dat_re, h_re);
            sum_re -= fract_mult(dat_im, h_im);
            sum_im += fract_mult(dat_re, h_im);
            sum_im += fract_mult(dat_im, h_re);

            sum_re = to_accum(rnd_saturate(sum_re));
            sum_im = to_accum(rnd_saturate(sum_im));
        }
        *((long long *)p_y) =
          llcompose(extract_high(sum_re), extract_high(sum_im));
        p_y += 2;
    }
}
```

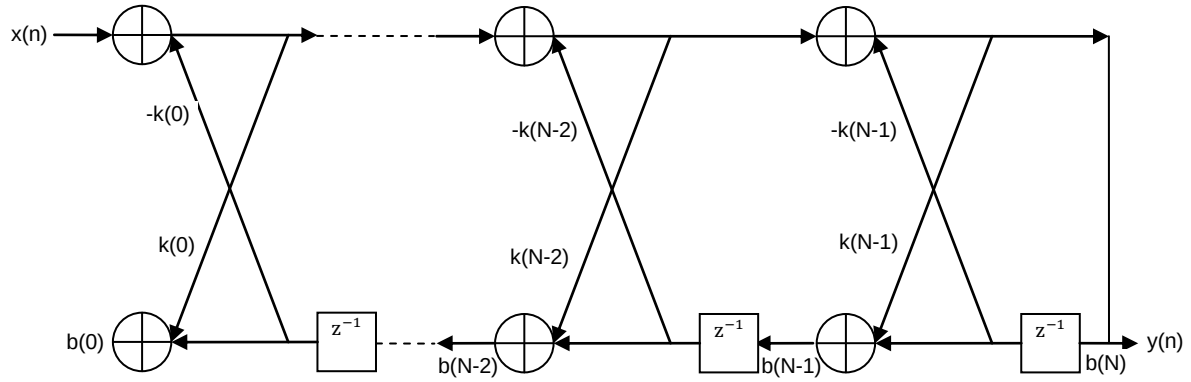
C.6 IIR Filter

Below, the tutorial example in [[2], §Chapter 1] is rewritten in terms of the LPDSP32 intrinsics.

```
// file : iirdirect.c
// Low pass filter:
// Sample frequency (Hz) : 44000
// Cut off frequency (Hz) : 5000
// Damping factor : 1.5
   const double a = 0.0409501; // m = 2, s = 1
   const double b = 0.170625;
   const double g = 0.506825;
   const int C[5] = {
         as_int(a), as_int(2*a), as_int(a), as_int(g), as_int(-b)
    };

   int xd[2];
   int yd[2];

   int low_pass(int x)
   {
       accum_t sum = fract_mult(x, C[0])
      + fract_mult(xd[0],C[1]) + fract_mult(xd[1],C[2])
      + fract_mult(yd[0],C[3]) + fract_mult(yd[1],C[4]);

       int y = rnd_saturate(sum << 1);

       xd[1] = xd[0];
       xd[0] = x;
       yd[1] = yd[0];
       yd[0] = y;
       return y;
    }
    volatile int chess_storage(DMIO:0xC00004) input_port;
    volatile int chess_storage(DMIO:0xC00008) output_port;

    void main()
    {
        while(1)
           output_port = low_pass(input_port);
     }
```

C.7 All-pole IIR Lattice Filter

This example shows a real all-pole IIR filter in lattice structure with and without optimization for LPDSP32. This filter consists of N lattice stages (N must be an even number and N >=4). Each stage requires one reflection coefficient k and one delay element b. The delay elements are initialized to zero. The order of the coefficients is such that $k[0]$ corresponds to the first lattice stage after the input and $k[N-1]$ corresponds to the last stage.



All-pole IIR Filter with N Lattice Stages

The c-code for this real all-pole IIR filter in lattice structure when the data and the coefficients are integers, is as given below.

```
#define NS 256  //No. of samples
#define N 64    //Lattice structure with N stages for IIR filter of order N
int y[NS];      //Output Signal
int x[NS];      //Input Signal
int k[N];       //Reflection coefficients
int b[N+1];     //Delay elements initialized to zero

void iir_lattice(int *y, int *x, int *k, int *b)
{
    int rt;
    for(int n=0; n<NS; n++)
    {
        rt = x[n];
        for(int i=0; i < N; i++)
        {
            rt -= b[i+1] * k[i];
            b[i] = b[i+1] + rt * k[i];
        }
        b[N] = rt;
        y[n] = rt;
    }
}
```

If the data and the coefficients are in the fixed-point `Q1.31` format, then the multiplier output is in the `Q1.62` format, which must be converted back to `Q1.31` format. Using the 72-bit accumulator, `fract_mult()` and the `rnd_saturate()` functions of the LPDSP32, the fixed-point real all-pole IIR filter in lattice structure can be coded as shown below.

Fixed-point initial C-code for porting on LPDSP32:

```
#define NS 256 //No. of samples
#define N 64   //Lattice structure with N stages for IIR filter of order N

int y[NS];     //Output Signal
int x[NS];     //Input Signal
int k[N];      //Reflection coefficients
int b[N+1];    //Delay elements initialized to zero
void iir_lattice(int *y, int *x, int *k, int *b)
{
    int rt;
    for(int n=0; n<NS; n++)
    {
        accum_t acc1 = to_accum(x[n]);
        for(int i=0; i < N; i++)
        {
            acc1 -= fract_mult(b[i+1], k[i]);
            rt = rnd_saturate(acc1);
            acc1 = to_accum(b[i+1]) + fract_mult(rt, k[i]);
            b[i] = rnd_saturate(acc1);
            acc1 = to_accum(rt);
        }
        b[N] = rt;
        y[n] = rt;
    }
}
```

The above code can be further optimized using the `chess` directives as shown below.

Fixed-point optimized C-code for porting on LPDSP32:

```
int chess_storage(DMA) y[NS];  //Output Signal
int chess_storage(DMA) x[NS];  //Input Signal
int chess_storage(DMA) k[N];   //Reflection coefficients
int chess_storage(DMB) b[N+1]; //Delay elements initialized to zero
void iir_lattice(int *y, int *x, int *k, int chess_storage(DMB) *b)
{
    int rt;
    for(int n=0; n<NS; n++) chess_loop_range(1,)
    {
        accum_t acc1 = to_accum(x[n]);
        for(int i=0; i < N; i++) chess_loop_range(1,)
        {
            acc1 -= fract_mult(b[i+1], k[i]);
            rt = rnd_saturate(acc1);
            acc1 = to_accum(b[i+1]) + fract_mult(rt, k[i]);
            b[i] = rnd_saturate(acc1);
            acc1 = to_accum(rt);
        }
        b[N] = rt;
        y[n] =rt;
    }
}
```

C.8 Matrix Multiplication

For the matrix multiplication of two matrices A and B, the number of columns of A must be equal to the number of rows of B The resulting matrix Y has the same number of rows as A and the same number of columns as B. Consider the matrix multiplication, $\mathrm{Y(3x3)} = \mathrm{A(3x2)} * \mathrm{B(2x3)}$.

$$\begin{bmatrix} y(0,0) & y(0,1) & y(0,2) \\ y(1,0) & y(1,1) & y(1,2) \\ y(2,0) & y(1,0) & y(2,2) \end{bmatrix} = \begin{bmatrix} a(0,0) & a(0,1) \\ a(1,0) & a(1,1) \\ a(2,0) & a(2,1) \end{bmatrix} * \begin{bmatrix} b(0,0) & b(0,1) & b(0,2) \\ b(1,0) & b(1,1) & b(1,2) \end{bmatrix}$$

$$= \begin{bmatrix} a(0,0)*b(0,0)+a(0,1)*b(1,0) & a(0,0)*b(0,1)+a(0,1)*b(1,1) & a(0,0)*b(0,2)+a(0,1)*b(1,2) \\ a(1,0)*b(0,0)+a(1,1)*b(1,0) & a(1,0)*b(0,1)+a(1,1)*b(1,1) & a(1,0)*b(0,2)+a(1,1)*b(1,2) \\ a(2,0)*b(0,0)+a(2,1)*b(1,0) & a(2,0)*b(0,1)+a(2,1)*b(1,1) & a(2,0)*b(0,2)+a(2,1)*b(1,2) \end{bmatrix}$$

The matrix elements of $\mathrm{A(R1 \times C1)}$are stored as,

$$\mathrm{A(ar, ac) = A[ac + (ar * C1)]} \text{ where } \mathrm{ar} = \{0 \cdots \mathrm{R1}\} \text{ and } \mathrm{ac} = \{0 \cdots \mathrm{C1}\}$$

The matrix elements of $\mathrm{B(C1 \times C2)}$are stored as,

$$\mathrm{B(ac, bc) = B[ac + (bc * C1)]} \text{ where } \mathrm{ac} = \{0 \cdots \mathrm{C1}\} \text{ and } \mathrm{bc} = \{0 \cdots \mathrm{C2}\}$$

The matrix elements of $\mathrm{Y(R1 \times C2)}$are stored as,

$$\mathrm{Y(ar, bc) = Y[bc + (ar * C2)]} \text{ where } \mathrm{ar} = \{0 \cdots \mathrm{R1}\} \text{ and } \mathrm{bc} = \{0 \cdots \mathrm{C2}\}$$

Note that the elements of matrix B are stored in a different order when compared with the matrices A and Y.

The C-code for matrix multiplication function when all the elements of all the matrices are integers, is as given below.

```
#define R1 128  //No. of rows in matrix A and Y
#define C1 64   //No. of columns in matrix A and No. of rows in matrix B
#define C2 256  //No. of columns in matrix B and Y

int A[R1 * C1]; //Input matrix of size R1 x C1
int B[C1 * C2]; //Input matrix of size C1 x C2
int Y[R1 * C2]; //Output matrix of size R1 x C2

void matrix_mul(int *Y, int *A, int *B)
{
    for(int ar=0; ar<R1; ar++)           //A rows and Y rows
    {
        for(int bc=0; bc<C2; bc++)       //B columns and Y columns
        {
            long long sum = 0;
            for(int ac=0; ac<C1; ac++)   //A columns and B rows
            {
                sum += A[ac + (ar*C1)] * B[ac + (bc*C1)] ;
            }
            Y[bc + (ar*C2)] = sum;
        }
    }
}
```

If all the elements of all the matrices are in the fixed-point `Q1.31` format, then the multiplier output is in the `Q1.62` format, which must be converted back to `Q1.31` format. Using the 72-bit accumulator, `fract_mult()` and the `rnd_saturate()` functions of the LPDSP32, the fixed-point matrix multiplication function can be coded as shown below.

Fixed-point initial C-code for porting on LPDSP32:

```
#define R1 128  //No. of rows in matrix A and Y
#define C1 64   //No. of columns in matrix A and No. of rows in matrix B
#define C2 256  //No. of columns in matrix B and Y

int A[R1 * C1]; //Input matrix of size R1 x C1
int B[C1 * C2]; //Input matrix of size C1 x C2
int Y[R1 * C2]; //Output matrix of size R1 x C2

void matrix_mul(int *Y, int *A, int *B)
{
    for(int ar=0; ar<R1; ar++)             //A rows and Y rows
    {
        for(int bc=0; bc<C2; bc++)         //B columns and Y columns
        {
            accum_t sum = to_accum(0);
            for(int ac=0; ac<C1; ac++)     //A columns and B rows
            {
                sum += fract_mult(A[ac + (ar*C1)] , B[ac + (bc*C1)] );
                sum = to_accum(rnd_saturate(sum));
            }
            Y[bc + (ar*C2)] = extract_high (sum);
        }
    }
}
```

The above code can be further optimized using the `chess` directives, cyclic addressing feature of LPDSP32 as shown below.

Fixed-point optimized C-code for porting on LPDSP32:

```
int chess_storage(DMA) A[R1 * C1];  //Input matrix of size R1 x C1
int chess_storage(DMB) B[C1 * C2];  //Input matrix of size C1 x C2
int chess_storage(DMA) Y[R1 * C2];  //Output matrix of size R1 x C2

void matrix_mul(int *Y, int *A, int *B)
{
    int chess_storage(DMA) *p_Y = Y;
    int chess_storage(DMA) *p_A_st = A;
    int chess_storage(DMA) *p_A = p_A_st;
    int chess_storage(DMB) *p_B = B;

    //A rows and Y rows
    for(unsigned int ar=0; ar<R1; ar++) chess_loop_range(1,)
    {
        //B columns and Y columns
        for(unsigned int bc=0; bc<C2; bc++) chess_loop_range(1,)
        {
            accum_t sum = to_accum(0);

            //A columns and B rows
            for(unsigned int ac=0; ac<C1; ac++) chess_loop_range(1,)
            {
                sum += fract_mult(*p_A, *p_B);
                p_A = cyclic_add(p_A,1,p_A_st,C1);
                p_B = cyclic_add(p_B,1,B,C1*C2);
```

```
                sum = to_accum(rnd_saturate(sum));
            }
            *p_Y++ = extract_high (sum);
        }
        p_A_st += C1;
        p_A = p_A_st;
    }
}
```

C.9 Auto Correlation

The auto correlation function of length L is given by,

$$y[n] = \sum_{k=N-L}^{k=N-1} x[k] * x[k-n] \text{ for } n = \{0, 1, \cdots, N-L-1\}$$

where N is the Number of input samples.

The code below shows the autocorrelation function with and without optimization for LPDSP32.

Initial C-code for porting on LPDSP32:

```
#define L 16      //Length of Autocorrelation
#define N 64+L    //No. of Input Samples

int x[N];   //Input Signal
int r[N-L]; //Output Signal

void auto_corr(int *r, int *x)
{
    int i, k,sum;
    for(int i=0; i<(N-L); i++){
        sum = 0;
        for(int k=(N-L); k<N; k++){
            sum += x[k] * x[k-i];
        }
        r[i] = sum;
    }
}
```

Optimized C-code for porting on LPDSP32:

```
#define L 16    //Length of Autocorrelation
#define N 64+L  //No. of Input Samples

int chess_storage(DMA %(sizeof(long long))) x[N]; //Input Signal
int chess_storage(DMB) r[N-L];                    //Output Signal

void auto_corr(int *p_r, int *p_x)
{
    int chess_storage(DMA) *p_c_st, *p_c;
    long long sum0, sum1;
    int dat1, dat2, dat3, dat4;

    p_c_st = p_x + N-L;
    p_c =  p_x + N - 2;
    lldecompose(*((long long *)p_c), dat2, dat1);
    p_c = cyclic_add(p_c,-2,p_c_st,L);
```

```
    sum0 = long_mult(dat1 , dat1);
    sum0 += long_mult(dat2 , dat2);
    sum1 = long_mult(dat1 , dat2);
    for(unsigned int i=0; i<3; i++) chess_loop_range(1,)
    {
        lldecompose(*((long long *)p_c), dat4, dat3);
        p_c = cyclic_add(p_c,-2,p_c_st,L);
        sum0 += long_mult(dat3, dat3);
        sum0 += long_mult(dat4, dat4);
        sum1 += long_mult(dat2, dat3);
        sum1 += long_mult(dat3, dat4);
        lldecompose(*((long long *)p_c), dat2, dat1);
        p_c = cyclic_add(p_c,-2,p_c_st,L);
        sum0 += long_mult(dat1, dat1);
        sum0 += long_mult(dat2, dat2);
        sum1 += long_mult(dat4, dat1);
        sum1 += long_mult(dat1, dat2);
    }
    lldecompose(*((long long *)p_c), dat4, dat3);
    p_c = cyclic_add(p_c,-2,p_c_st,L);
    sum0 += long_mult(dat3, dat3);
    sum0 += long_mult(dat4, dat4);
    sum1 += long_mult(dat2, dat3);
    sum1 += long_mult(dat3, dat4);
    int chess_storage(DMA) *p_v;
    p_v = p_x + N-L -1; //63
    sum1 += long_mult(dat4 , *p_v);
    *p_r++ = extract_low(sum0);
    *p_r++ = extract_low(sum1);
    for(unsigned int i=0; i<(N-L-2); i+=2) chess_loop_range(1,)
    {
        p_v = p_x + N - 4 - i;        //76, 74,...,16
        int cdat1, cdat2;
        lldecompose(*((long long *)p_c), cdat2, cdat1);
        p_c = cyclic_add(p_c,-2,p_c_st,L);
        lldecompose(*((long long *)p_v), dat2, dat1); p_v -= 2;
        sum0 = long_mult(cdat1, dat1);
        sum0 += long_mult(cdat2, dat2);
        sum1 = long_mult(cdat1 , dat2);
        for(unsigned int j=0; j<7; j++) chess_loop_range(1,)
        {
            int dat1, dat2;
            lldecompose(*((long long *)p_v), dat2, dat1); p_v -= 2;
            sum1 += long_mult(cdat2 , dat1);
            lldecompose(*((long long *)p_c), cdat2, cdat1);
            p_c = cyclic_add(p_c,-2,p_c_st,L);
            sum0 += long_mult(cdat1, dat1);
            sum0 += long_mult(cdat2, dat2);
            sum1 += long_mult(cdat1, dat2);
        }
        lldecompose(*((long long *)p_v), dat2, dat1);
        sum1 += long_mult(cdat2, dat1);
        *p_r++ = extract_low(sum0);
        *p_r++ = extract_low(sum1);
    }
}
```

Appendix D. Exceptional cases

D.1 Elongation failure (before version 10R1.12)

In certain exceptional cases, when having inline assembly code, sometimes elongation breaks, because CHESSDE's scheduler does not want to elongate any inline assembly instructions. In a later scheduler version, this has been improved (so as to elongate instructions present in inline assembly code). But, the scheduler has not been updated into the IP Programmer, since the assembly code generated might differ from the assembly code generated by the previous version of the scheduler.

One such example which can trigger the error and the workaround are given below:

```
int data[4];
void test();

void test()
{
    // nothing
}

void main(void)
{
    data[0] =  0x000000;
    data[1] =  0x187DE2;
    data[2] =  0x2D413C;
    data[3] =  0x3B20D7;
      test();
      enable_interrupts();
      while(1){
            test();
      }
}
```

The inline assembly code of the user function `enable_interrupts()` is as follows:

```
inline assembly void enable_interrupts()
    clobbers() property(volatile functional loop_free)
{
    asm_begin
        ie = 1
        nop
    asm_end
}
```

The above C code generates the following error message

```
Error: found no preceding instruction that could be elongated in order to
meet instruction alignment constraint encountered in:
in "$working_directory/main.c", line 9
```

The workaround for this is, the programmer has to introduce a nop() between the `enable_interrupts()` and the `while` loop as shown below:

```
int data[4];
void test();

void test()
{
```

```
    // nothing
}

void main(void)
{
    data[0] =  0x000000;
    data[1] =  0x187DE2;
    data[2] =  0x2D413C;
    data[3] =  0x3B20D7;
      test();
      enable_interrupts();
      nop();              //added to remove elongation error
      while(1){
            test();
      }
}
```

D.2 Induction Variable Analysis failure

The CHESSDE front-end tool can fail to do Induction Variable Analysis in a few rare cases.

One such case is caused by the simultaneous presence in one function of both:

```
    void* operator+/- (void*, int)
    int operator- (void*, void*)
```

on the same pointer variable.

Example:

```
    *(p_stmp0--) = ...;

     .........


     non = (int)(p_stmp0 - p_c);
```

One easy workaround is by isolating the pointer difference use, replacing the second line by:

```
   non = (int)(chess_copy(p_stmp0) - p_c);
```

The front-end should detect this automatically and act appropriately, however there is no easy (short term) tool fix possible. Since the issue is very uncommon, we propose to use the workaround.

Bibliography

[1] Overview of manuals. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, May 2010. Release 10R1.

[2] Chess Compiler User manual. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, May 2010. Release 10R1.

[3] Bridge Linker User manual. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, May 2010. Release 10R1.