DSP on xCORE Multicore Microcontrollers for Embedded Developers

Document Number: X7402A

Publication Date: 2016/2/19

SYNOPSIS

This documents provides an introduction to the Data Signal Processing (DSP) operations available to embedded engineers who want to write application programs for xCORE multicore microcontrollers. xCORE devices are general purpose processors rather than a specialised DSP devices, but they includes a number of instructions that speed up common DSP operations. These can be run on the eight concurrent logical cores available in each xCORE Tile, along with specialized I/O and timing instructions.

Table of Contents

1 Data types

IN THIS CHAPTER

- Machine representation
- High level language representation

1.1 Machine representation

The xCORE-XS1 architecture supports a *word* as its basic data type. A word comprises 32 bits that can be held in a register, load/stored in a single operation into memory, or communicated between logical cores in a single operation over a channel end. Most operations are 32-bit wide, but there are a few 64-bit operations.

The four most important data types for numeric operations are:

- signed 32-bit integer
- unsigned 32-bit integer
- signed 64-bit integer
- unsigned 64-bit integer

A signed 32-bit integer is represented as a 2's complement word, a signed 64-bit integer is stored as two words: a signed 32-bit value (the high word), and an unsigned 32-bit value (the low word). Given the low and the high word, the 64-bit signed value is:

high * 0x100000000 + low

An unsigned 64-bit number is stored using two unsigned 32-bit integers, high and low. Fixed point types are represented using these four integer types, as will be shown in §2.4.

1.2 High level language representation

In C and XC, the standard `int` and `unsigned` types refer to 32-bit signed and unsigned values. 64-bit numbers can be declared as a single variable by declaring it as a `long long` or `unsigned long long`. In the compiled code, these values will occupy two registers or two memory words.

If desired, 64-bit values can be declared explicitly as two words, e.g.

```
int high; unsigned int low;
```

will declare the two halves of a 64-bit number. Functions can return a pair of values, e.g. `{int,unsigned}` or `{unsigned,unsigned}`.

Figure 1: Summary

Integer type	*XC, C*
signed 32-bit	int
unsigned 32-bit	unsigned
signed 64-bit	long long
unsigned 64-bit	unsigned long long

2 Operations

IN THIS CHAPTER

The full set of operations is beyond the scope of this document, and can be found in the XS1 Architecture Manual[1].

2.1 Word arithmetic

All standard operations are supported with *word* operands. In these operations only the least significant 32-bits of the result are stored, ignoring any overflow or carry:

- ADD: adds two 32-bit numbers, works on signed and unsigned values
- SUB: subtracts two 32-bit numbers, works on signed and unsigned values
- NEG: negates a 32-bit signed number
- MUL: multiplies two 32-bit numbers, works on signed and unsigned values
- DIVU: divides two 32-bit unsigned numbers
- DIVS: divides two 32-bit signed numbers
- REMU: remainder of division of two 32-bit unsigned numbers
- REMS: remainder of division of two 32-bit signed numbers

The first four operations complete in a single logical core cycle, the latter four take up to 32 logical core cycles to complete.

In C and XC these operations are denoted by +, -, *, / and %. Operations are signed, unless both operands are unsigned in which case an unsigned result is produced.

2.2 Multi word arithmetic

Multi-word arithmetic provides instructions that facilitate long arithmetic on 32, 64, 96, or longer data sizes. The instructions perform the following operations:

[1] http://www.xmos.com/published/xs1_en?version=latest

- LADD: add three 32-bit numbers (notionally two source operands and a carry-bit), resulting in a 64-bit number (notionally a 32-bit result and a carry-bit). This instruction has five register operands: three source operands and two destination operands.
- LSUB: subtract two 32-bit numbers from a third 32-bit number (notionally two source operands and a borrow-bit), resulting in a 64-bit number (notionally a 32-bit result and a borrow-bit). This instruction has five register operands: three source operands and two destination operands.
- LMUL: multiply two 32-bit unsigned numbers, and add two 32-bit values, resulting in a 64-bit number. Notionally, the inputs are two source operands to be multiplied, and two carries for implementing a long multiplication as an array of multiplications. This instruction has six register operands: four source operands and two destination operands.
- LDIVU: Divide a 64 bit number by a 32-bit number, resulting in a 32-bit divider and a 32-bit remainder. This instruction has five register operands: three source operands and two destination operands. This operation is not a single cycle operation, and takes up to 32 logical core cycles to complete.

These operations are used by the XC and C compilers when operating on the `long long` and `unsigned long long` data type.

2.3 Multiply accumulate

The XS1 architecture has two instructions that are designed to efficiently implement multiply-accumulate operations on signed and unsigned numbers. These operations perform a full precision 32x32 into 64 bits multiplication, and accumulate into a 64-bit accumulator.

- MACCS: Multiply two signed 32-bit values to form a signed 64-bit answer, and accumulate this 64-bit value into a 64-bit accumulator. This instruction has 4 register-operands: two source 32-bit values to be multiplied, and 2 registers that are the accumulator.
- MACCU: Multiply two unsigned 32-bit values to form an unsigned 64-bit answer, and accumulate this 64-bit value into a 64-bit accumulator. This instruction has 4 register-operands: two source 32-bit values to be multiplied, and 2 registers that are the accumulator.

Any XC or C an expression `x + (long long) y*z` is mapped onto a multiply-accumulate operation, provided that `y` and `z` are `int`, and `x` is a `long long`. Simimlarly, `x + (unsigned long long) y*z` is mapped to an unsigned multiply accumulate provided that `y` and `z` are `unsigned int`, and `x` is a `unsigned long long`. Hence, the inner product over two vectors of length `N` can be written as:

```
#define N 10

long long innerProduct(int x[N], int y[N]) {
    long long hl = 0;
    for(int i = 0; i < N; i++) {
        hl += (long long) x[i]*y[i];
    }
    return hl;
}
```

2.4 Fixed point arithmetic

Fixed point arithmetic is implemented on top of signed arithmetic by the programmer defining the split between integral and fractional parts, and using long arithmetic to produce a full result fixed point answer.

For example: multiplying a 16.16 signed fixed-point number (a number between +32767.999985 and -32768) with a 8.24 signed fixed-point number (a number between 127.99999994 and -128) results in a 64-bit answer that contains a 24.40 fixed-point value (a number between 8388608 and -8388608). The multiplication yields an exact result, no rounding or truncation are performed during the operation.

If the input operands are known to have some headroom, then the result will also have headroom. For example, If the second operands are only in the range +1 to -1, then there are seven bits headroom, and a series of 128 multiply accumulate operations can be performed before the result is no longer guaranteed to fit in the 64-bit word.

The programmer must decide:

- When to perform rounding, and how to round numbers. Rounding is implemented by adding 1 or triangular noise below the lowest bit.
- When to reduce the number of bits, for example, reduce a 64-bit answer into a 32-bit number, or reduce 96 bits to 64 bits. At this stage, any bits occupying the headroom typically indicate an overflow.
- When to check that there is no headroom available (overflow), and what to do. For example, you may want to implement saturating arithmetic.

Rounding can be implemented in a variety of ways - depending on the type of rounding required, and depending on the sequence of operations. Typically, if there is a sequence of multiply-accumulate operations, the accumulator will be initialised to 0.5 in the lowest bit, or some other value that averages out to 0.5. Alternatively, 0.5 can be added after all operations are completed.

Reducing the number of bits involves shifting the data right. In C and XC we can simply shift the 64-bit value using >>, and then cast it to an `int`. At machine level, this involves shifting both words involved, and ORing the new value. This is implemented using three standard bit-operations:

- SHL: Shift a word of bits to the left, inserting zero bits on the right. When used for reducing a fixed point number, then this operation is performed on the more-significant word.
- SHR: Shift a word of bits to the right, inserting zero bits on the left. When used for reducing a fixed point number, this operation is performed on the less-significant word.
- OR: or two bit patterns. When used for a reducing a fixed point number, this operation is performed on the result of the two shift operations.

Before shifting the value down, a test on overflow can be implemented by simply comparing the value against the maximum allowed value, e.g. `if (accumulator > 0x007FFFFFFFFFFFFFL)` will test if the accumulator has strayed positively into the top 8 bits, and `if (accumulator > 0xFF80000000000000L)` will test if the accumulator has strayed negatively into the top 8 bits. If both tests fail, then the top 8 bits of the number can be chopped off safely.

signed word add, sub, mul, div, rem	+, -, *, /, %
unsigned word add, sub, mul, div, rem	+, -, *, /, %
multiply double word accumulate	+=
test on overflow	>, <
64 to 32-bit conversion	>>

Figure 2: Summary

3 Fixed point algorithms

IN THIS CHAPTER

This chapter shows how to implement two common DSP algorithms on an xCORE multicore microcontroller: a FIR and a Biquad (complete programs that perform cascaded biquads and FIRs can be found on http://github.com/xcore/sc_dsp_filters/).

3.1 FIR example

In its most general form, a FIR is performed by multiplying two vectors of signed numbers. One vector contains the sample data, and one vector contains the FIR coefficients. In the example code below, we have assumed that we have 80 coefficients that are represented in a 16.16 format, and that the sample data is represented in a 8.24 format. Coefficients are assumed to have values in the full range [-32768..32767] whereas we know that samples are only values in the range [-0.5..+0.5]. We want the result of the FIR as a sample value, represented in an 8.24 format. The code to perform this is:

```
#define N 80

int FIR(int samples[N], int coefficients[N]) {
    long long acc;
    acc = 0x8000L;                   // Rounding bit.
    for(int i = 0; i < N; i++) {
        acc += samples[i]*coefficients[i];
    }
    if (acc > 0x00007FFFFFFFFFFFL) { // Check pos overflow
        return 0x7FFFFFFF;           // Saturate
    }
    if (acc < 0xFFFF800000000000L) { // Check neg overflow
        return 0x80000000;           // Saturate
    }
    return (int)(acc >> 16);         // Make result 8.24
}
```

Since all numbers in X only occupy 24 bits (the bottom 23 bits and a sign bit), there are 8 bits headroom, and up to 256 products can be added together without overflow. So we know we can safely multiply the two vectors.

After the multiplication, the resulting number is represented as a 24.40 number. This needs to be made into a 8.24 number. The simplest operation that performs this coercion is to shift the number to the right by 16 places, and cast it to an `int`. That will truncate the bottom 16 bits (during the shift) and the top 16 bits (during the cast to `int`). If we wanted to round the bottom bits to the nearest half, then we should add 0x8000 prior to the shift.

If we wanted to implement saturating arithmetic, then rather than simply casting the value to an `int`, we should first check whether the value is greater than `0x0000 7FFF FFFF FFFF` or less than `0xFFFF 8000 0000 0000`.

The function can be generalised to work on coefficients other than 16.16, and sample data other than 8.24. In the code below, the constant `FRAC` defines the number of bits in the fraction of each coefficient. The function returns the same representation as it gets on the input samples, but assumes that there is sufficient headroom to not overflow during the 80 additions, i.e. the total headroom in coefficients and samples is at least 7 bits.

```
#define N 80
#define FRAC 16

int FIR(int samples[N], int coefficients[N]) {
    long long acc;
    acc = 1L << (FRAC-1);            // Rounding bit.
    for(int i = 0; i < N; i++) {
        acc += samples[i]*coefficients[i];
    }
    if (acc > (0x7FFFFFFFFFFFFFFFL >> (32-FRAC))) {
        return 0x7FFFFFFF;           // Saturate
    }
    if (acc < (0x8000000000000000L >> (32-FRAC))) {
        return 0x80000000;           // Saturate
    }
    return (int)(acc >> FRAC); // shift result to be same
}                              // representation as samples[]
```

If performance is paramount, then the test on overflow can be optimized. A test of whether an overflow has occurred can be performed by simply testing whether the top bits are all zero (in the case of unsigned numbers) or whether the top bits are all identical (in the case of signed numbers). On an xCORE device this operation can be performed by zero-extending or sign-extending the result and testing whether this is the same as the result prior to zero/sign extension:

- ZEXT: zero extends a 32-bit value, from a specific bit upwards. When used as an overflow test on an unsigned number, ZEXT should be a no-operation. ZEXT has two operands, the input value and the bit-position to ZEXT from.
- SEXT: sign extends a 32-bit value, from a specific bit upwards. When used as an overflow test on a signed number, SEXT should be a no-operation. SEXT has two operands, the input value and the bit-position to SEXT from.

If this test indicates an overflow, then a saturated value can be returned. Otherwise the number can be truncated safely:

```
#include <xs1.h>

#define N 80

int FIR(int samples[N], int coefficients[N]) {
    long long acc;
    int ah;
    acc = 0x8000;             // Rounding bit.
    for(int i = 0; i < N; i++) {
        acc += samples[i]*coefficients[i];
    }
    ah = acc >> 32;
    if (sext(ah, 16) != ah) { // Quickly test on overflow
        if (ah < 0) {         // Negative overflow
            return 0x80000000;// Saturate MAXNEG
        } else {
            return 0x7FFFFFFF;// Saturate MAXPOS
        }
    } else {
        return acc >> 16;
    }
}
```

3.2 Biquad example

The biquad is implemented assuming 8.24 input values, and 8.24 coefficients. Coefficients and values do not need to use the same representation (in the case of a biquad, coefficients could be represented as 4.28 values), and it is up to the designer to chose precisions that suit the domain.

A general biquad operation performs:

```
y[n] = (b0*x[n] + b1*x[n-1] + b2*x[n-2] + a1*y[n-1] + a2*y[n-2])/a0
```

Where b0, b1, b2, a0, a1, and a2 are the biquad coefficients. Assuming that the coefficients are constant, all constants are normally divided by a0, meaning that the computation reduces to:

```
y[n] = B0*x[n] + B1*x[n-1] + B2*x[n-2] + A1*y[n-1] + A2*y[n-2]
```

You can implement this by using multiply-accumulate operations, with a 64 bit accumulator that holds a 16.48 value, and then rounding and clamping the accumulator into y[n]. Assuming that the samples span the full 8.24 range (i.e. from +128..-128), and assuming that the coefficients are all in the range [-2..2] then:

- After the first multiply-accumulate the accumulator is in the range +256..-256, requiring 57 bits
- After the second multiply-accumulate the accumulator is in the range +512..-512, requiring 58 bits

- After the third multiply-accumulate the accumulator is in the range +768..-768, requiring 58 bits
- After the fourth multiply-accumulate the accumulator is in the range +1024..-1024, requiring 59 bits
- After the fifth multiply-accumulate the accumulator is in the range +1280..-1280, requiring 59 bits

Hence, no overflow can happen during the computation. If you wish, you could extend the precision of the coefficients by 4 bits without having to worry about overflow, i.e. representing them as 4.28 enabling higher precision results.

At the end of the sequence, there may be an overflow, which is normal - a biquad on a signal in the range +128..-128 is not guaranteed to result in a signal in the range +128..-128.

Alternatively it can be written in XC:

```
#include <xs1.h>

int xn, xn1, xn2, yn1, yn2;
int B0, B1, B2, A1, A2;

int biquad(int xn) {
    int ah, yn;
    long long accumulator = 0x800000;
    accumulator += (long long) xn * B0;
    accumulator += (long long) xn1 * B1;
    accumulator += (long long) xn2 * B2;
    accumulator += (long long) yn1 * A1;
    accumulator += (long long) yn2 * A2;
    ah = accumulator >> 32;
    if (sext(ah, 24) != ah) {
        if (ah < 0) {
            yn = 0x80000000;
        } else {
            yn = 0x7FFFFFFF;
        }
    } else {
        yn = accumulator >> 24;
    }
    xn2 = xn1;
    xn1 = xn;
    yn2 = yn1;
    yn1 = yn;
    return yn;
}
```

Which translates to the following assembly:

```
#define xn 0
#define xn1 1
#define xn2 2
#define yn1 3
#define yn2 4

biquad:
    ldc   r0, 1
    shl   r0, r0, 24
    ldc   r1, 0
    ldw   r2, dp[B0]
    ldw   r3, sp[xn]
    maccs r0, r1, r2, r3
    ldw   r2, dp[B1]
    ldw   r3, sp[xn1]
    maccs r0, r1, r2, r3
    ldw   r2, dp[B3]
    ldw   r3, sp[xn2]
    maccs r0, r1, r2, r3
    ldw   r2, dp[A1]
    ldw   r3, sp[yn1]
    maccs r0, r1, r2, r3
    ldw   r2, dp[A2]
    ldw   r3, sp[yn2]
    maccs r0, r1, r2, r3
    or    r11, r0, r0
    sext  r11, 24
    eq    r10, r11, r0
    bt    r10, selectBits
    shr   r10, r10, 24
    bt    r10, useMinInt
    ldw   r0, cp[MAXINT]
    bu    done
useMinInt:
    ldw   r0, cp[MININT]
    bu    done
selectBits:
    shl   r0, r0, 8
    shr   r1, r1, 24
    or    r0, r0, r1
done:
```

4 Summary

Numeric instructions in the xCORE XS1 architecture work on values stored in one or two 32-bit words representing signed or unsigned, 32- or 64-bit numbers. These values can be interpreted as fixed point values if you require.

- Single cycle operations perform (long) multiplications and multiply-accumulate.
- The xCORE architecture performs full precision multiply-accumulate operations.
- The programmer inserts rounding and saturation where required.
- Operations are provided to construct multi-word arithmetic, e.g. 96-bit operations for fixed-point, or 1024 bit multiplications for cryptographic applications.