Fixed-Point Considerations

Introduction

The C6x family can handle a variety of data types. The most popular type used in DSP work is the 16-bit signed integer or short. The C67xx can efficiently handle floating point, in particular type float. When we have a C67xx DSK available we will most often want to develop floating point algorithms, since the development time will be less. Still, being knowledgeable of fixed-point issues is important for coding other members of the C6x family and being able to port code to other DSP families, such as the C54x, etc.

We begin with an overview of the C6x data types and then discuss the Q-notation for representing integers as binary fractions. Multiplication is discussed first, followed by addition. Overflow is a serious problem that can be dealt with by proper scaling. The C6x floating point format is also discussed.
C6x Data Types

Table 5.1: C6x data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char,</td>
<td>8 bits</td>
<td>ASCII</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>ASCII</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>2’s complement</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>binary</td>
</tr>
<tr>
<td>int, signed int</td>
<td>32 bits</td>
<td>2’s complement</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32 bits</td>
<td>binary</td>
</tr>
<tr>
<td>long, signed long</td>
<td>40 bits</td>
<td>2’s complement</td>
</tr>
<tr>
<td>unsigned long</td>
<td>40 bits</td>
<td>binary</td>
</tr>
<tr>
<td>enum</td>
<td>32 bits</td>
<td>2’s complement</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>long double</td>
<td>64 bits</td>
<td>IEEE 64-bit</td>
</tr>
<tr>
<td>pointers</td>
<td>32 bits</td>
<td>binary</td>
</tr>
</tbody>
</table>

Q-Format Number Representation

- On the C6x signed arithmetic is handled using 2’s complement
- The decimal value of a 2’s-complement number having \( N \) bits \( B = b_{N-1}, b_{N-2}, \ldots b_1 b_0, b_i \in \{0, 1\} \) is

\[
D(B) = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_12^1 + b_02^0 \quad (5.1)
\]
• For comparison purposes, consider for example 4-bit binary numbers in a signed and unsigned format

\[
\begin{array}{cccc}
8 & + & 0 & + 2 & + 0 = 10_{10} \\
2^3 & 2^2 & 2^1 & 2^0 & \text{Unsigned} \\
1 & 0 & 1 & 0 \\
-2^3 & 2^2 & 2^1 & 2^0 & \text{Signed} \\
-8 & + & 0 & + 2 & + 0 = -6_{10}
\end{array}
\]

◆ Signed (aka 2’s complement) 
MSB is negative

◆ Signed vs. Unsigned 
Same range of precision 
Signed centered around zero

• In a 16-bit system the largest positive number is \(2^{15} - 1 = 32767\) and the largest negative number is \(-2^{15} = -32768\)

• A better way to view these numbers is as fractions, that is numbers normalized between -1 and 1

• The \textit{Q-format} representation does exactly this by expressing the equivalent fractional value as

\[
F(B) = - b_{N-1} 2^0 + b_{N-2} 2^{-1} + \ldots + b_1 2^{-(N-2)} + b_0 2^{-(N-1)} \tag{5.2}
\]

• Rather than having the implied binary point at the far right end of the word, it is moved to one binary digit from the left
These same issues exist when we consider obtaining inputs from I/O devices such as an A/D converter.

- We can view the hex representation as the ultimate form that the C6x always works with.
Multiplicative Overflow

- A 16-bit fraction can be coded as follows:

  ![Diagram showing fractions and their hex equivalents]

  - **Fractions**: \(~ 1\), \(\frac{1}{2}\), 0, \(-\frac{1}{2}\), \(-1\)
  - **Hex**: 7FFFh, 4000h, 0000, C000h, 8000h

  For convenience we are using fractions, but the processor still uses 2’s complement (hex)

  **Example:** Encode the fraction 0.14

  Value: `value .short 0x7fff * 14/100`
  Or: `value .short 0x11eb`

Multiplicative Overflow

- We have seen how binary fractions (Q-format numbers) are convenient
- The real motivation comes when we consider overflow in multiplication
- Consider a 4-bit system and compare integer and fractional
representations:

$$\begin{array}{c|c}
6 & .6 \\
\times & \times \\
\hline
1 & .2 \\
\hline
\end{array}$$

8-bit result

$$\begin{array}{c|c}
1 & 2 \\
\hline
\end{array}$$

8-bit result

$$\begin{array}{c|c}
\cdot & .1 \\
\times & \times \\
\hline
\cdot & 1 \\
\hline
\end{array}$$

truncate 4 LSB’s

- In the integer representation which digit should be stored?
- The fractional form solves this problem, we lose precision, but the returned value is again 4-bits

- Consider another example and how the C6x stores the result:

$$\begin{array}{c|c}
3/4 & 0110 \\
\times & 1110 \\
\hline
0000 \\
0110 \\
0110 \\
1010 \\
\hline
- 3/16 & 1110100 \\
\end{array}$$

Sign Extension Bit. Why?
• A sign extension bit is automatically generated to insure that problems are not created when storing to larger word length memory.

• Consider storing \(1010_2 = -6\)

\[
\begin{array}{cccc}
2^3 & 2^2 & 2^1 & 2^0 \\
1 & 0 & 1 & 0 \\
\hline
-2^3 & 2^2 & 2^1 & 2^0 \\
-8 + 0 + 2 + 0 = -6
\end{array}
\]

• If this number is loaded into a larger register the upper bits must be filled

\[
\begin{array}{cccccc}
? & ? & 1 & 0 & 1 & 0 \\
-2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\hline
\text{Original:} & 0 + 0 + 8 + 0 + 2 + 0 = 10 & 001010 \\
\text{Zero Fill:} & 0 + 0 + 8 + 0 + 2 + 0 = 10 & 111010 \\
\text{Sign Exten:} & -32 + 16 + 8 + 0 + 2 + 0 = -6 & 111010
\end{array}
\]

• In the \(3/4 \times -1/4\) example what is stored in memory is

\[
\begin{array}{c}
\text{reg} \\
11110100 \\
\downarrow \\
\text{data} \\
1 1 1 0 \\
\hline
\text{memory}
\end{array}
\]

We have to track the binary point location.
• Rework the previous fractional multiply example, except now use Q-notation to keep track of the binary point:

\[
\begin{array}{c|c}
\frac{3}{4} & 0.110 \\
\times -\frac{1}{4} & 1.110 \\
\hline
& 0000 \\
& 0110 \\
& 0110 \\
& 1010 \\
\hline
\end{array}
\]

\[
- \frac{3}{16} \quad 1110100
\]

-1 + \frac{1}{2} + \frac{1}{4} = \frac{12}{16} - \frac{16}{16} = -\frac{4}{16}

• Now, let’s go through a 16-bit example; two Q15 numbers gives us a Q30 result:

<table>
<thead>
<tr>
<th>Q15</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{S.xxxxxxxxxxxxxxxxxxx}</td>
<td>MPY \ A3, A4, A6</td>
</tr>
<tr>
<td>x Q15</td>
<td>NOP</td>
</tr>
<tr>
<td>\text{S.yyyyyyyyyyyyyyyyy}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q30</th>
<th>Store to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{S.zzzzzzzzzzzzzzz}</td>
<td>SHR \ A6, 15, A6</td>
</tr>
<tr>
<td>SHH \ A6, *A7</td>
<td></td>
</tr>
</tbody>
</table>

Shift right 15 bits to truncate
To implement rounding, as opposed to truncation, we can add 1 to the LSB

A special multiplication case is representing $-1 \times -1$

- There is no exact representation for 1, as no hex value exists
- In assembly this is handled with the saturate multiply instructions SMPY and SMPYH

<table>
<thead>
<tr>
<th>Results</th>
<th>MPY (H)</th>
<th>SMPY (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Result</td>
<td>00.xxxxxxx</td>
<td>0.xxxxxxxx</td>
</tr>
<tr>
<td>Negative Result</td>
<td>11.xxxxxxxx</td>
<td>1.xxxxxxxx</td>
</tr>
<tr>
<td>-1 x -1 Result</td>
<td>01.xxxxxxxx</td>
<td>0.1111111</td>
</tr>
</tbody>
</table>

• To implement rounding, as opposed to truncation, we can add 1 to the LSB

- There is no exact representation for 1, as no hex value exists
- In assembly this is handled with the saturate multiply instructions SMPY and SMPYH

<table>
<thead>
<tr>
<th>Results</th>
<th>MPY (H)</th>
<th>SMPY (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Result</td>
<td>00.xxxxxxx</td>
<td>0.xxxxxxxx</td>
</tr>
<tr>
<td>Negative Result</td>
<td>11.xxxxxxxx</td>
<td>1.xxxxxxxx</td>
</tr>
<tr>
<td>-1 x -1 Result</td>
<td>01.xxxxxxxx</td>
<td>0.1111111</td>
</tr>
</tbody>
</table>

---

ECE 5655/4655 Real-Time DSP 5–9
Accumulative Overflow

- When we multiply binary fractions the result is a binary fraction that is also bounded on \([-1,1)\)
- When we add binary fractions we may overflow

\[
f \times f < f, \text{ but what about } f + f?\]

- In the familiar dot product example we need to be able to deal with accumulative overflow

\[
y = \sum_{n} a[n]x[n] \tag{5.3}
\]

- Several options exist:
  - Saturate the result and test for saturation
  - Use guard bits
  - Implement a non-gain algorithm, i.e., digital gain less than or equal to unity
Saturating the Result

• For 32-bit (int) numbers we can saturate the result of an addition or subtraction using

\[
\text{SADD/SSUB.L src1, src2, dst}
\]

+1 \(7\text{FFFFh}\) | +1 \(7\text{FFFF_FFFFh}\)

Q15 0 | Q31 0

-1 \(8000\text{h}\) | -1 \(8000_0000\text{h}\)

16-Bit (short) | 32-Bit (int)

• If saturation occurs the result is saturated or clipped and the SAT bit in the CSR is set to 1

\[
\text{CSR}
\]

\begin{array}{cccc}
31 & 24 & 23 & \text{Revision ID} \\
\hline
\text{CPU ID} & & & \\
\end{array}

- The SAT bit actually stays set (latched) until cleared
- You clear the SAT bit via system reset, or more practically by writing 0 to CSRSAT

• If saturation is detected an error or scaling routine may then be invoked
Kehtarnavaz (p. 145) provides a C algorithm to test 16-bit integer addition (modified here to use c6x.h instead of regs.h):
#include <c6x.h>

short safe_add(short A, short B, int *status)
{
    int X, Y, result, SAT_BIT;
    X = A << 16; //left shift to high 16 bits
    Y = B << 16; //left shift to high 16 bits
    result = _sadd(X,Y); //get to ASM via C intrinsic
    SAT_BIT = (CSR >>9) & 0x1; //get bit 9
    if(SAT_BIT==1)
    {
        //Overflow Occurred
        CSR = _clr(CSR,9,9);  //Reset bit 9
        *status = 1;
    }
    else
    {
        *status = 0;
    }
    return (result >> 16) //shift to lower 16 bits
}

– This function returns a saturated short, but also indicates if saturation occurred, and finally resets the SAT bit

• More efficient ways of doing the above in pure assembly are also possible
Use Guard Bits

- The idea with guard bits is to temporarily obtain more *headroom*

![Fractions diagram](image)

- We can use the full 40-bits available from the `ADD` instruction since the ALU has this capability on the C62x

```
ADD.L1 A5:A4, A3, A1:A0
```

- **Note:** The C67x has the hardware to perform full 64-bit operations

- Eventually the 40-bit result has to be converted back to 32-bits
  - The multiplier cannot use the 40-bit value
  - Additional memory is required to store these values

- A 40-bit value can easily be restored back to 32-bits using
  ```
  SAT.L src (40-bit), dst (32-bit)
  ```
Overflow Allowed by Design

- A third approach is to allow overflow to occur by design
- The best insurance to have a *non-gain* system

\[ x[n] \rightarrow h[n] \rightarrow y[n] \]

- If the system is bounded and linear, then given \(|x[n]| < 1\),
  we are assured by proper design that \(|y[n]| < 1\)
- If the final result is known not to have an overflow problem,
  then intermediate overflows will eventually come back below
  the saturation point due to the way 2’s complement arithmetic
  works

\[
\begin{align*}
7\text{FF}0\text{h} & \quad + \quad 10\text{h} \quad = \quad 8\text{0F}0\text{h} \\
7\text{F0}0\text{h} & \quad + \quad 10\text{h} \quad = \quad 8\text{100h} \\
-200\text{h} & \quad = \quad 7\text{F0}0\text{h}
\end{align*}
\]
Floating Point

- The problems associated with fixed-point number calculations go away when we switch to floating point numbers.
- The C67x has hardware dedicated to floating-point operations, while on the C62x floating-point can be emulated in software.
- In this section the format float and double precision, SP and DP respectively, formats will be discussed.

Single Precision

- The 32-bit SP format used on the C6x is the IEEE standard.

\[
\begin{array}{c|c|c|c}
31 & 30 & 23 & 22 & 0 \\
\hline
s & \text{exp} & \text{frac} & \\
\end{array}
\]

- Mathematically we have,

\[
-1^s \cdot 2^{\text{exp} - 127} \cdot 1.\text{frac} \tag{5.4}
\]

where \( s \) is the sign bit (bit 31), \( \text{exp} \) represents the exponent (bits 23–30), and \( \text{frac} \) represents the fractional or mantissa (bits 0–22).
- The dynamic range for SP runs from \( 1.175 \times 10^{-38} \) to \( 3.4 \times 10^{38} \).
Double Precision

- The 64-bit DP format used on the C6x is also the IEEE standard

\[ \begin{array}{cccccc}
31 & 30 & 20 & 19 & 0 & 31 \\
\hline
s & \text{exp} & \text{frac} & \text{frac} & & \\
\end{array} \]

<table>
<thead>
<tr>
<th>Odd Register</th>
<th>Even Register</th>
</tr>
</thead>
</table>

- Mathematically we have,

\[-1^s \cdot 2^{\text{exp}-1023} \cdot 1.\text{frac} \quad (5.5)\]

where \( s \) is the sign bit (bit 31), \( \text{exp} \) represents the exponent (bits 20–30), and \( \text{frac} \) represents the fractional or mantissa (all 32 bits of the even register and bits 0–19 of the odd register)

- The dynamic range for SP runs from \( 2.2 \times 10^{-308} \) to \( 1.7 \times 10^{308} \)

Addition

- When adding two floating point numbers

\[ a = a_{\text{frac}} \times 2^{a_{\text{exp}}} \quad (5.6) \]

\[ b = b_{\text{frac}} \times 2^{b_{\text{exp}}} \quad (5.7) \]
we get
\[ c = a + b \]
\[ = (a_{frac} + (b_{frac} \times 2^{-(a_{exp} - b_{exp})})) \times 2^{a_{exp}}, \quad a_{exp} \geq b_{exp} \quad (5.8) \]
\[ = (((a_{frac} \times 2^{-(b_{exp} - a_{exp})}) + b_{frac}) \times 2^{b_{exp}}, \quad a_{exp} < b_{exp} \]

**VC5505 Considerations**

- When working with a fixed-point only processor such as the VC5505, we need to consider what the overall gains and losses are, from a programming standpoint
- In the Kuo\(^1\) text the following tables appears

<table>
<thead>
<tr>
<th>Fixed-Point Processor</th>
<th>Floating-Point Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>16- or 24- bits popular</td>
<td>32-bits most common</td>
</tr>
<tr>
<td>Limited dynamic range</td>
<td>Large dynamic range</td>
</tr>
<tr>
<td>Overflow and quantization</td>
<td>Easier to program since no scaling is required (?)</td>
</tr>
<tr>
<td>errors <strong>must</strong> be resolved</td>
<td>Poorer C-compiler efficiency; assembly is more common (?)</td>
</tr>
<tr>
<td></td>
<td>Better C-compiler efficiency; can be developed in C</td>
</tr>
<tr>
<td>Long product development</td>
<td>Quick time to market</td>
</tr>
<tr>
<td>time</td>
<td>Faster clock rate</td>
</tr>
<tr>
<td></td>
<td>Slower clock rate</td>
</tr>
<tr>
<td>Faster clock rate</td>
<td>Slower clock rate</td>
</tr>
<tr>
<td>Less silicon area required;</td>
<td>More silicon area is required; functional units more complex</td>
</tr>
<tr>
<td>functional units are simpler</td>
<td></td>
</tr>
<tr>
<td>Fixed-Point Processor</td>
<td>Floating-Point Processor</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Cheaper</td>
<td>More expensive</td>
</tr>
<tr>
<td>Lower power consumption</td>
<td>Higher power consumption</td>
</tr>
<tr>
<td>Disk drives and motor control</td>
<td>Image processing in radar, sonar, and seismic applications</td>
</tr>
<tr>
<td>Consumer audio applications such as MP3 players, multimedia gaming, and digital cameras</td>
<td>High-end audio applications such as ambient acoustic simulators, professional audio encoding/decoding, and audio mixing</td>
</tr>
<tr>
<td>Speech coding/decoding and channel coding</td>
<td>Sound synthesis in professional audio and video coding/decoding</td>
</tr>
<tr>
<td>Communications devices such as modems and vcellular phones</td>
<td>Prototyping</td>
</tr>
</tbody>
</table>