Chapter 5

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

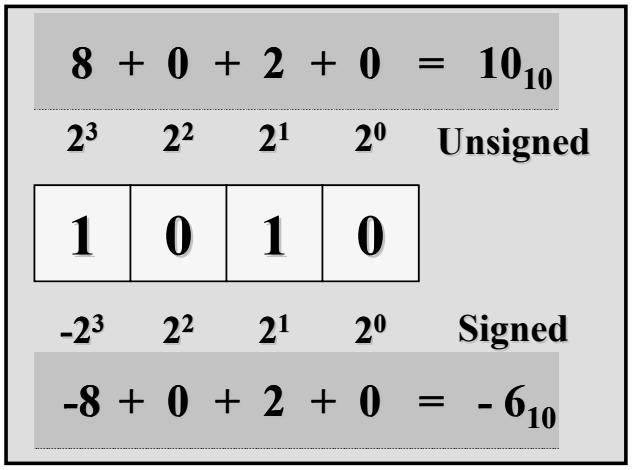
Type	Size	Representation
`char, signed char,` `unsigned char`	8 bits 8 bits	ASCII ASCII
`short` `unsigned short`	16 bits 16 bits	2's complement binary
`int, signed int` `unsigned int`	32 bits 32 bits	2's complement binary
`long, signed long` `unsigned long`	40 bits 40 bits	2's complement binary
`enum`	32 bits	2's complement
`float`	32 bits	IEEE 32-bit
`double`	64 bits	IEEE 64-bit
`long double`	64 bits	IEEE 64-bit
`pointers`	32 bits	binary

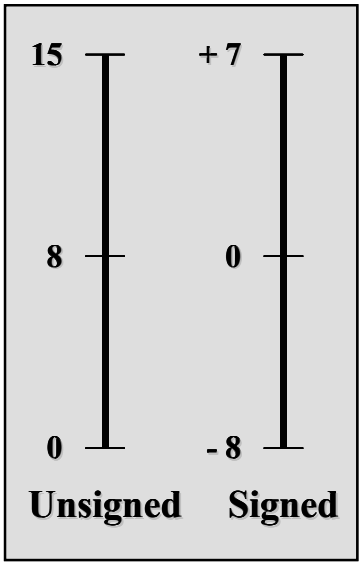
Q-Format Number Representation

- 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

$$D(B) = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_1 2^1 + b_0 2^0 \quad (5.1)$$

- For comparison purposes, consider for example 4-bit binary numbers in a signed and unsigned format



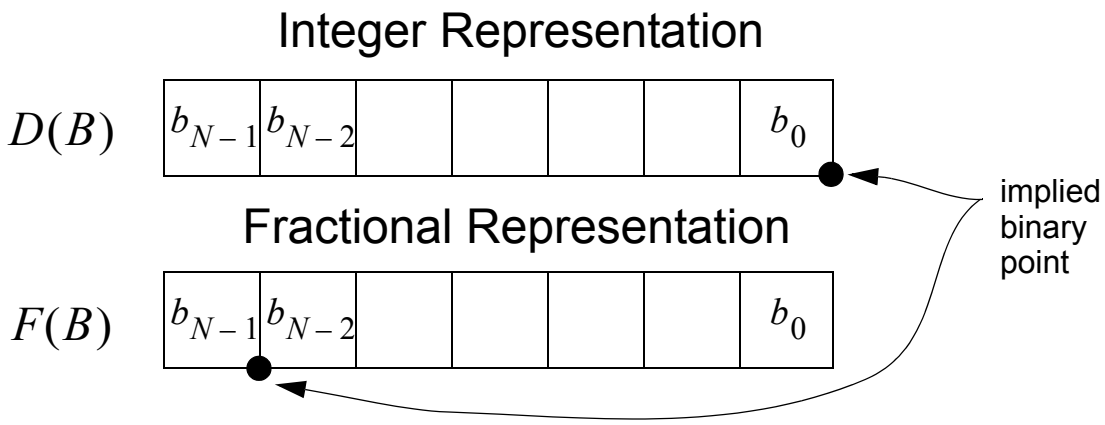


- **Signed (aka 2's complement) MSB is negative**
- **Signed vs. Unsigned Same range of precision Signed centered around zero**

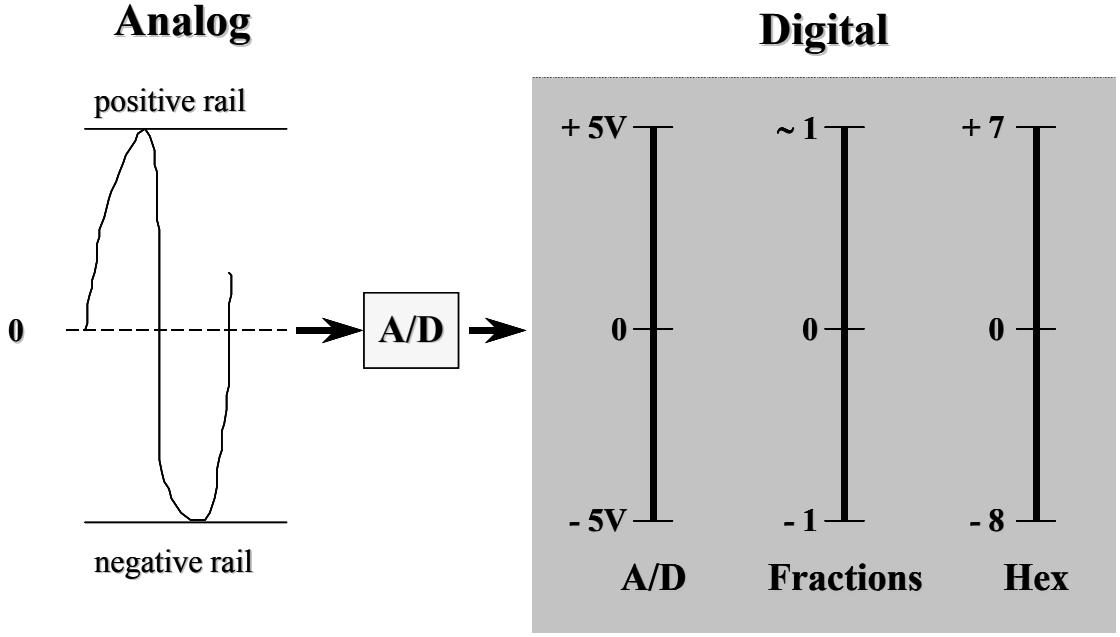
- In a 16-bit system the largest positive number is $2^{15} - 1 = 32767$ and the largest negative number is $-2^{15} = -32768$
- A better way to view these numbers is as fractions, that is numbers normalized between -1 and 1
- The *Q-format* representation does exactly this by expressing the equivalent 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

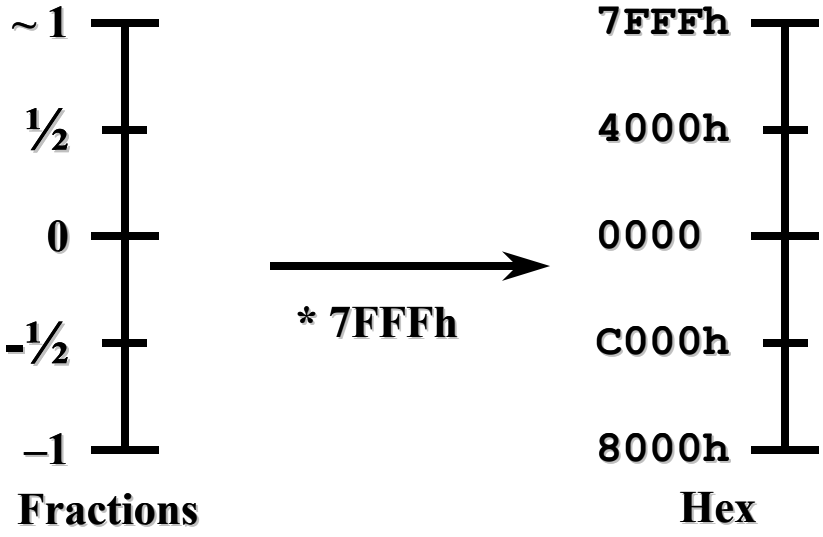


- These same issues exist when we consider obtaining inputs from I/O devices such as an A/D converter



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

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



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

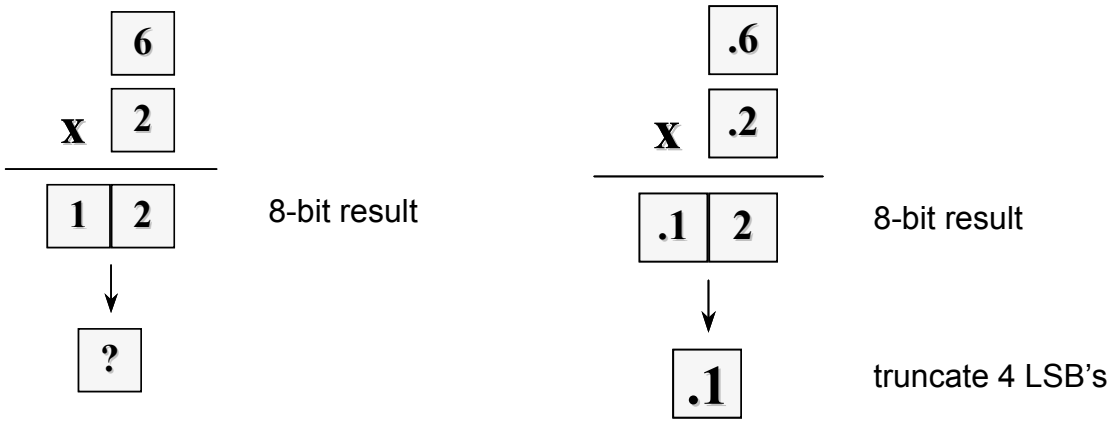
Example: Encode the fraction 0.14

```
    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:



– 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

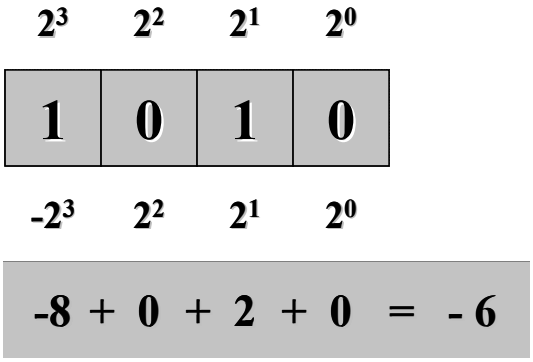
- Consider another example and how the C6x stores the result:

```
   3/4      0110
x -1/4      1110
-------------------
            0000
           0110
          0110
         1010
-------------------
- 3/16   1110100
```

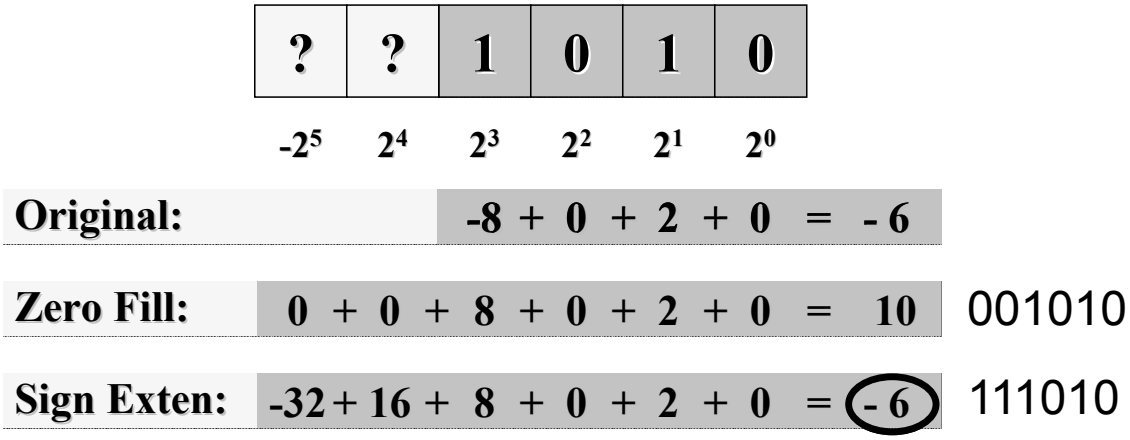
reg 11110100

Sign Extension Bit. Why?

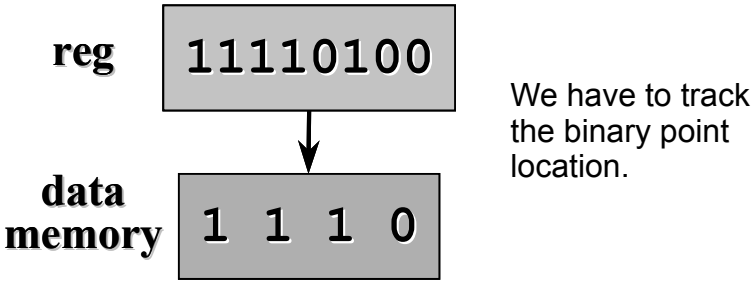
- A sign extension bit is automatically generated to insure that problems are not created when storing to larger word length memory
- Consider storing `1010b` = -6



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

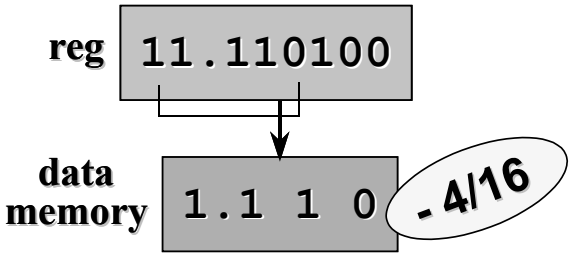


- In the 3/4 x -1/4 example what is stored in memory is



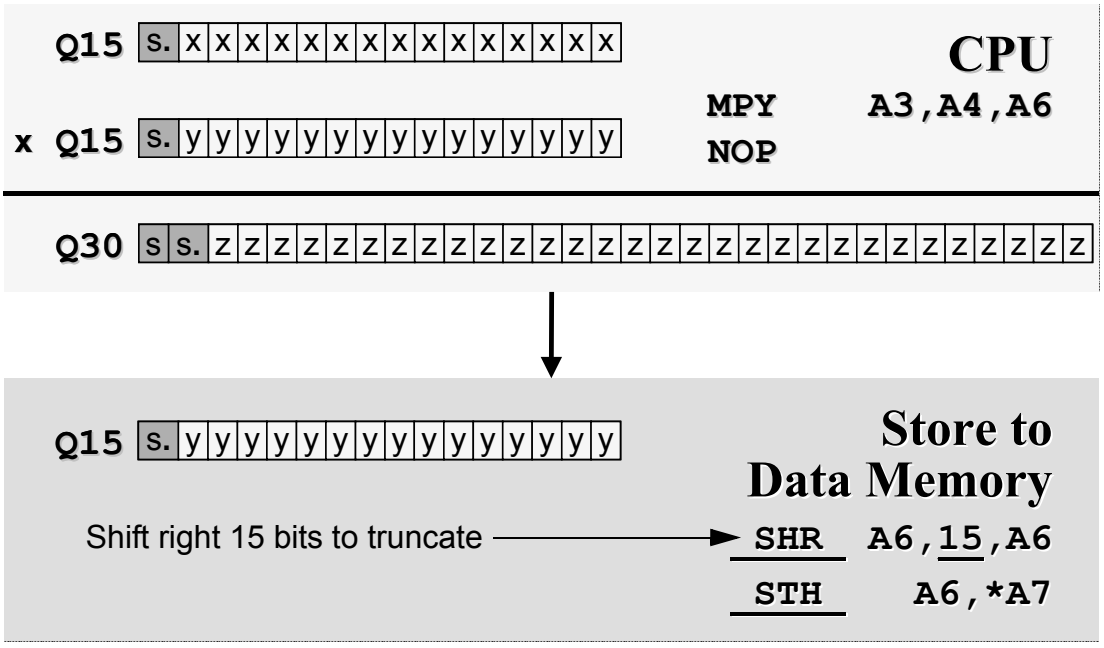
- Rework the previous fractional multiply example, except now use Q-notation to keep track of the binary point:

```
   3/4     0.110
x -1/4     1.110
-------------------
             0000
            0110
           0110
          1010
-------------------
- 3/16   1110100
```

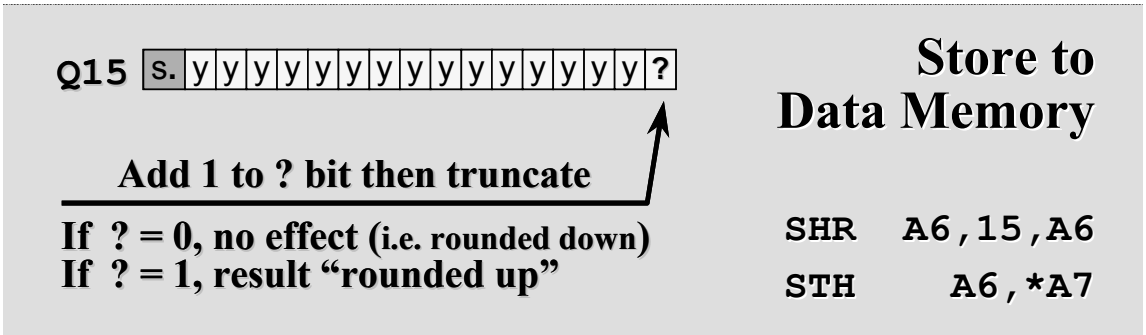


$$-1+\frac{1}{2}+\frac{1}{4}=\frac{12-16}{16}=-\frac{4}{16}$$

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



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



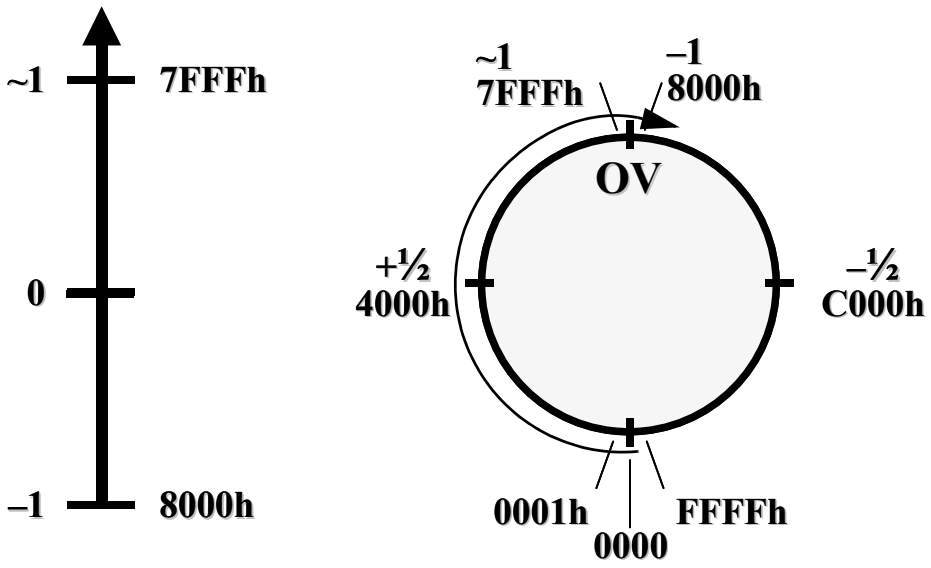
- A special multiplication case is representing -1×-1
  - There is no exact representation for 1, as no hex value exists
  - In assembly this is handled with the saturate multiply instructions `SMPY` and `SMPYH`

Results	MPY(H)	SMPY(H)
Positive Result	00.xxxxxx	0.xxxxxx
Negative Result	11.xxxxxx	1.xxxxxx
-1 x -1 Result	01.xxxxxx	0.111111

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 * f < f, but what about f + f ?



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

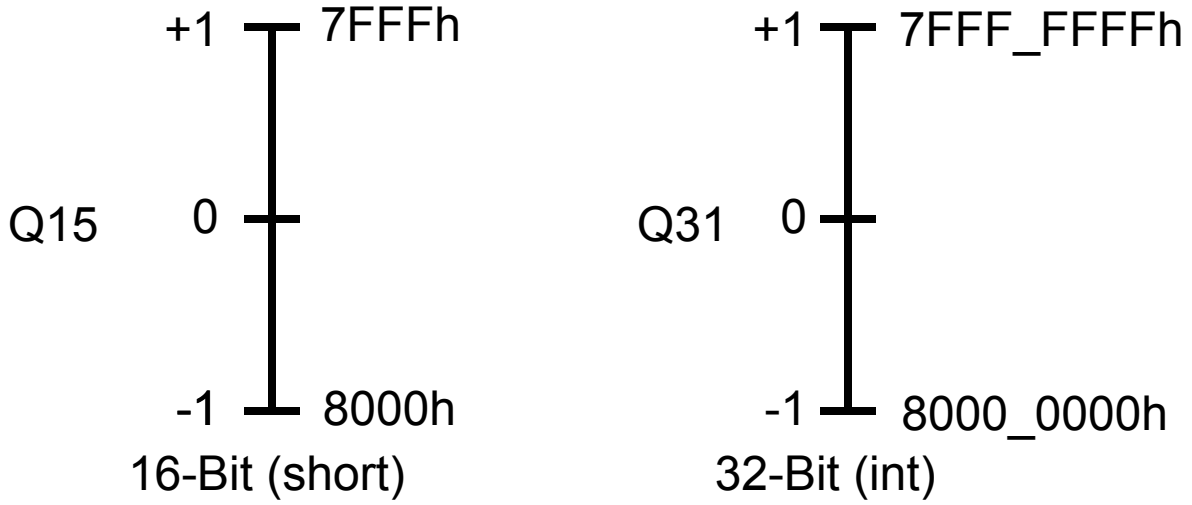
$$y = \sum_{n} a[n]x[n] \tag{5.3}$$

- 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

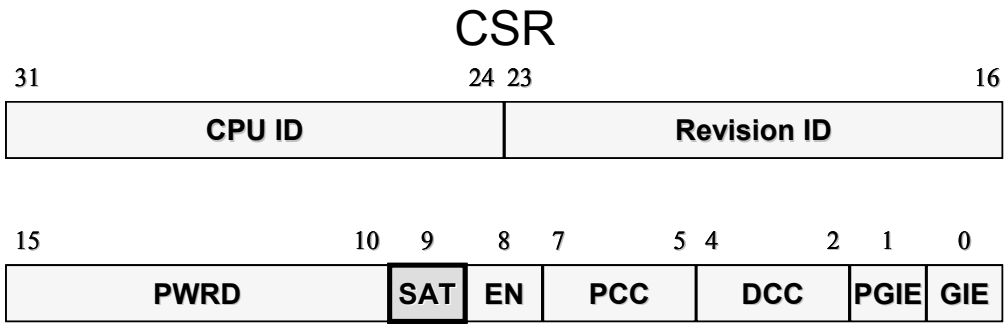
Saturating the Result

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

```
SADD/SSUB.L    src1, src2, dst
```



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



  - The `SAT` bit actually stays set (latched) until cleared
  - You clear the `SAT` bit via system reset, or more practically by writing 0 to $\texttt{CSR}_{\texttt{SAT}}$
- 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`):

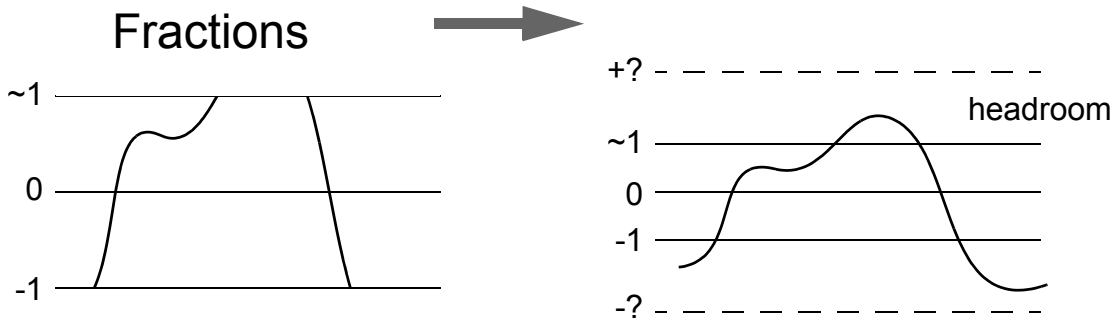
```
#include <c6x.h>

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

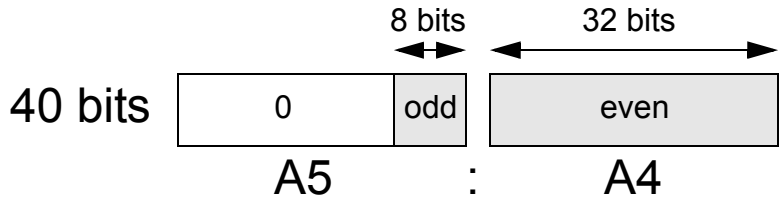
- More efficient ways of doing the above in pure assembly are also possible

Use Guard Bits

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



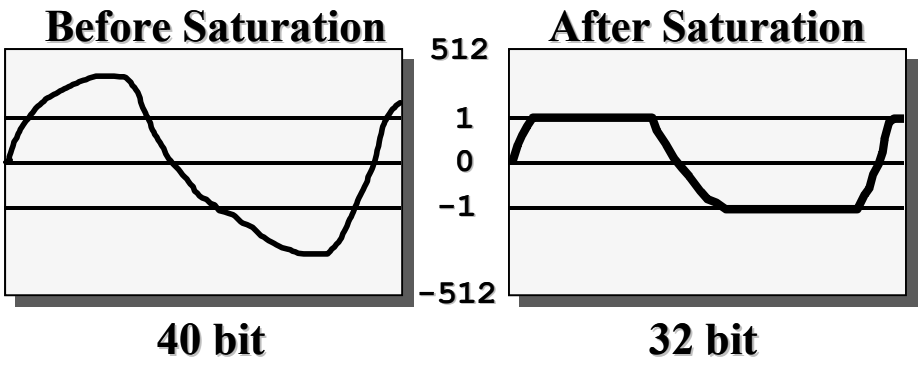
- We can use the full 40-bits available from the `ADD` instruction since the ALU has this capability on the C62x




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

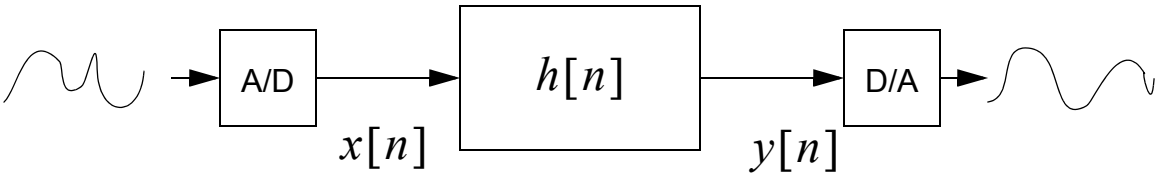
  - Note: The C67x has the hardware to perform full 64-bit operations
- Eventually the 40-bit result has to be converted back to 32-bits
  - The multiplier cannot use the 40-bit value
  - Additional memory is required to store these values
- A 40-bit value can easily be restored back to 32-bits using

```
SAT.L  src (40-bit), dst (32-bit)
```

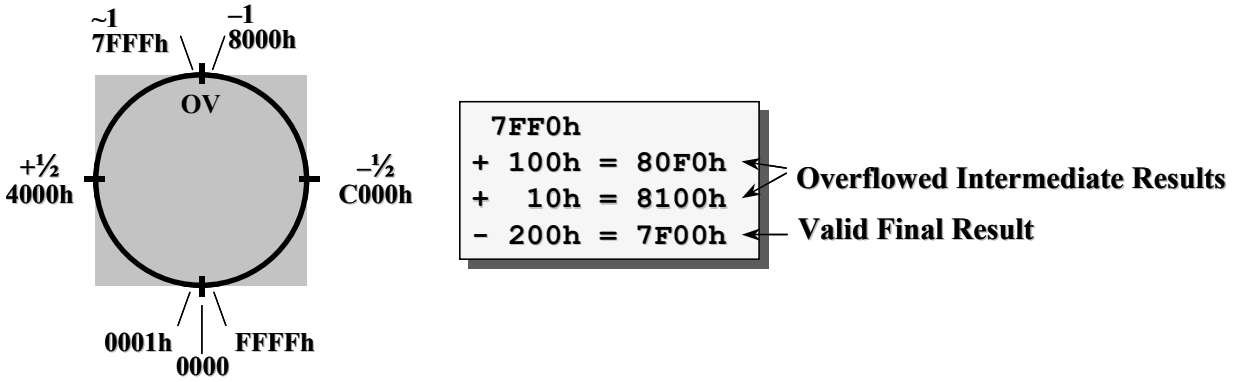


Overflow Allowed by Design

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



  - 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

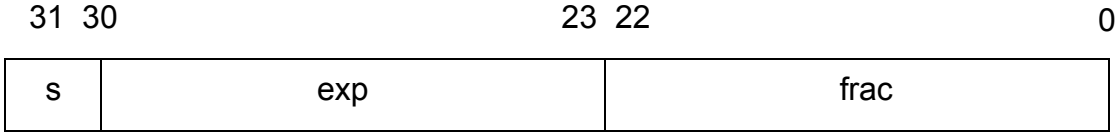


Floating Point

- 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



- Mathematically we have,

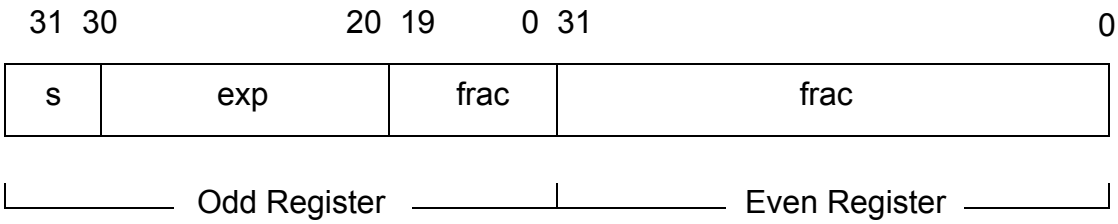
$$-1^{s} \cdot 2^{exp-127} \cdot 1.frac \qquad (5.4)$$

where s is the sign bit (bit 31), *exp* represents the exponent (bits 23–30), and *frac* represents the fractional or mantissa (bits 0–22)

- The dynamic range for SP runs from 1.175×10^{-38} to 3.4×10^{38}

Double Precision

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



- Mathematically we have,

$$-1^{s} \cdot 2^{exp-1023} \cdot 1.frac \tag{5.5}$$

where s is the sign bit (bit 31), *exp* represents the exponent (bits 20–30), and *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 2.2×10^{-308} to 1.7×10^{308}

Addition

- When adding two floating point numbers

$$a = a_{frac} \times 2^{a_{exp}} \tag{5.6}$$

$$b = b_{frac} \times 2^{b_{exp}} \tag{5.7}$$

we get

$$
\begin{aligned}
c &= a + b \\
&= (a_{frac} + (b_{frac} \times 2^{-(a_{exp} - b_{exp})})) \times 2^{a_{exp}}, a_{exp} \geq b_{exp} \\
&= ((a_{frac} \times 2^{-(b_{exp} - a_{exp})}) + b_{frac}) \times 2^{b_{exp}}, a_{exp} < b_{exp}
\end{aligned}
\tag{5.8}
$$

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

Fixed-Point Processor	Floating-Point Processor
16- or 24- bits popular	32-bits most common
Limited dynamic range	Large dynamic range
Overflow and quantization errors **must** be resolved	Easier to program since no scaling is required (?)
Poorer C-compiler efficiency; assembly is more common (?)	Better C-compiler efficiency; can be developed in C
Long product development time	Quick time to market
Faster clock rate	Slower clock rate
Less silicon area required; functional units are simpler	More silicon are is required; functional units more complex

Fixed-Point Processor	Floating-Point Processor
Cheaper	More expensive
Lower power consumption	Higher power consumption
Disk drives and motor control	Image processing in radar, sonar, and seismic applications
Consumer audio applications such as MP3 players, multimedia gaming, and digital cameras	High-end audio applications such as ambient acoustic simulators, professional audio encoding/decoding, and audio mixing
Speech coding/decoding and channel coding	Sound synthesis in professional audio and video coding/decoding
Communications devices such as modems and vcellular phones	Prototyping

1. S. Kuo and W. Gan, *Digital Signal Processors, Architectures, Implementations, and Applications*, Prentice Hall, New Jersey, 2005.