EECS 452 – Lecture 3

Today: Fixed point arithmetic

Announcements:
HW1 1 due today.
HW2 assigned today due next tue.
Pre-project ideas (PPI) due thurs. on Ctools
PPI comments and ranking due next wed. on Ctools
Project teaming meeting 7-9PM on Sept 18.
On Th Dr. Aswin Rao (TI Design Engineer) presents:
TI E2E `Innovation Challenge: North America`

References: Please see last slide.

Last one out should close the lab door!!!!

Please keep the lab clean and organized.

God does arithmetic. — Carl Friedrich Gauss

Pre-project idea (PPI) assignment (100 pts)

- Due before 11:55PM on thursday on Ctools (60 pts)
  - Assignment counts as one homework
  - Two PPI contributions: PPI 1 (30pts) and PPI 2 (30pts)
  - Each PPI must be one page and follow the specified format
- Between sat and wed eve do the following (40 pts)
  - comment on PPI's on Ctools Forums
    - Each PPI will have a comment thread
    - Comment on at least 2 PPI's other than your own
    - Respond to comments on your own PPI
    - Comments are open and should be constructive
  - Rank the top 4 PPIs that interest you
- Popular PPIs may be good prospects for teaming
  - You are encouraged to network during open comment period
  - Early efforts will give us head start in team formation on Sept 18

How you might approach the PPI exercise

- Take a look at the 3 examples listed in document linked to PPI assignment on Ctools
  - Background and overview
  - Proposed project effort
  - Who might use the device?
- Take a look at TI's repository of projects on Homework/Projects wiki
- Take a look at Harris's projects on Homework/Projects wiki
- Be imaginative but realistic in your PPI
  - An application that you and others may be interested in
  - A project idea that involves DSP in some way
  - A project idea that is feasible in context of a semester-long MDE
- In the comments part of exercise
  - Comment on at least two PPI's (not yours) of interest to you
  - Critique and propose extensions or improvements
  - Respond constructively to others comments on your PPIs

Number representation

- We have been spoiled by tools like MATLAB:
  - they use 64-bit floating point;
  - hide the details of computation from us.
- Most embedded processors use far fewer bits and do not natively support floating point. We need to know how to work with finite wordlength and get valid results.
- Today's lecture looks at number representations:
  - unsigned, signed, fractional;
  - properties of addition and multiplication.
- The two main concerns:
  - partial/end results might be too large for the word size used (overflow);
  - when discarding least significant bits (LSB) values need to be rounded properly.

Roundoff/overflow can have dire consequences

Human catastrophes (http://www.ima.umn.edu/ arnold/disasters/)

- The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.
- The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow

Societal impact
(http://www.ma.utexas.edu/users/arbogast/misc/disasters.html)

- Political: parliamentary elections in Schleswig-Holstein. Rounding error changes Parliament makeup.
- Financial: Vancouver Stock Exchange (1982). A stock index loses half of its value due to truncation instead of rounding.

Positional notation (1/2)

The number 124 is a value (implicitly) in decimal (base 10):
$124 = 1 \times 100 + 2 \times 10 + 4 \times 1$
$\quad = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$

What if I want to write it in binary form (base 2), using only 1s and 0s?
$124 = 64 + 32 + 16 + 8 + 4$
$\quad = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
$\quad = 1111100_2$ (or 1111100b) .

What if I want to write it in hexadecimal (hex) form (base 16)?
$124 = 7 \times 16^1 + 12 \times 16^0 = 7C_{16}$ (or 0x7C, or 7Ch)

To generalize, a value a can be written using any values of r (say N digit long).
$a = d_{N-1}r^{N-1} + d_{N-2}r^{N-2} + \cdots + d_1 r^1 + d_0 r^0$.

r is an integer called the *base* or *radix*.

We say that $d_{N-1} \ldots d_0$ is the **radix-r representation** of the number a.

Positional notation (2/2)

Similarly, knowing the base, we can convert any number to decimal

The number 1001 in binary (base 2) is evaluated by:
$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ $(= 9_{10})$

The number 156 in hexadecimal ("hex", or base 16) is evaluated by:
$1 \times 16^2 + 5 \times 16^1 + 6 \times 16^0 = 256 + 80 + 6 = 342.$

Or to generalize, given a N-digit number $d_{N-1}d_{N-2}\cdots d_1 d_0$ in base r, its value is
$d_{N-1}r^{N-1} + d_{N-2}r^{N-2} + \cdots + d_1 r^1 + d_0 r^0 = \sum_{n=0}^{N-1} d_n r^n$. where
$0 \leq d_n < r$.

You should know how to convert a number between different bases...

Representable value range

Assume an N digit representation (word size) using radix r. Each digit value lies within the range 0 through $r-1$.

The smallest representable value is 0. The largest representational value is $r^N - 1$.

Clearly the largest representable value occurs when all the digits have value $r-1$. Then

$$\text{value} = \sum_{n=0}^{N-1} (r-1) r^n = (r-1) \frac{r^N - 1}{r-1} = r^N - 1.$$

For example, with $N = 4$, $r = 2$: 1111_2 has a decimal value of $2^4 - 1 = 15$.

Common radix values

Why radix other than 10?

- Computers work in base 2: everything is represented by 1s and 0s.
  - Numbers, letters, graphics, programs.
  - Data is organized into fixed word sizes, typically 8, 16, 32, and 64 bits per word.
- Computing is most often done using binary numbers.
- Radix/Base 2 (or binary) is common, but it can also be cumbersome.
  - For example, to represent decimal 8,000,000 we need 23 binary digits.
- Convenient to express the content of a word (size being multiples of four) in hex notation, radix/base 16: every four binary digits converts to a single hex digit.

Note that radix 16 uses digit values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Converting between binary and hex

- Converting between binary and hexadecimal is easy.
  - Binary to hexadecimal: group the binary digits (bits) into groups of four bits; replace each group with a hexadecimal digit.
  - Example: 0001 1111 0101 becomes 1F5.
  - Hexadecimal to binary: replace each hexadecimal digit with it's associated binary bit pattern.
  - Example: C42 becomes 1100 0100 0010.
- Converting between other radices or other representations generally takes a little work.
  - Convert 43_{16} to decimal
  - Convert 50_{10} to hex
  - Convert 56_{10} to binary

Fractions

When we (in the US) write a value like 123.4 this indicates a fractional part. The . separates the integer part and the fractional part of a number.
For this example we have

$$123.4 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1}$$

The . separates the integer part and the fractional part of a number.

For the above example the . is called the decimal point. In more general terms it is referred to as the radix point. For binary numbers it is called the binary point.

What is 100.101_2 in decimal?

The position of the binary point is implicit

Given an eight bit word size containing the (fixed point) bit pattern

$$10101011$$

Where is the binary point?
If it was to right of 5th MSB then it would be the binary number

$$10101.011$$

which has *value*

$$21.375.$$

Because a bit pattern can represent values with their binary point anywhere in the word, and *even somewhere not in the word*, the position of the binary point must be made explicit by the programmer/system designer.

This is unlike floating point representations.

Q notation

Using a N-bit word for $N = 8$, consider $b_7b_6b_5b_4b_3b_2b_1b_0$ where

$b_7b_6b_5b_4b_3b_2b_1b_0.$ — an integer value,

$b_7b_6b_5b_4b_3b_2b_1b_0xxx.$

$b_7b_6b_5b_4.b_3b_2b_1b_0$

$b_7.b_6b_5b_4b_3b_2b_1b_0$ — ranges from 0 to $2 - 2^{-7}$.

A convention has developed to help keep track of the location of the binary point. Assigned to each value is a number Qn where n is the index of the digit which the radix point immediately follows.

Often, for clarity, Qn is written $Q(n)$

Qn examples

For the 8 bit binary pattern $b_7b_6b_5b_4b_3b_2b_1b_0$ we have

$b_7b_6b_5b_4b_3b_2b_1b_0.$ — Q0,

$b_7b_6b_5b_4b_3b_2b_1b_0xxx.$ — Q-3,

$b_7b_6b_5b_4.b_3b_2b_1b_0$ — Q4

$b_7.b_6b_5b_4b_3b_2b_1b_0$ — Q7

$.b_7b_6b_5b_4b_3b_2b_1b_0$ — Q8

$x.xxb_7b_6b_5b_4b_3b_2b_1b_0$ — Q10

Keep in mind that n does not have to correspond to a bit position within the word itself.

Binary (unsigned) N-bit Qn: range and resolution of representation ($N = 4$)

N	Qn	range (max value)	resolution (step size)
4	$Q0$(xxxx.)	$2^4 - 1 = 15$	1
4	$Q1$(xxx.x)	$4 + 2 + 1 + \frac{1}{2}$	1/2
		$= 2^3 - \frac{1}{2} = 7.5 = 15/2$	
4	$Q2$(xx.xx)	$2 + 1 + \frac{1}{2} + \frac{1}{2^2}$	1/4
		$= 2^2 - \frac{1}{2^2} = 3.75 = 15/4$	
4	$Q3$(x.xxx)	$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3}$	1/8
		$= 1.875 = 3.75/2 = 15/8$	
N	$Q0$	$2^N - 1$	1
N	Qn	$2^{N-n} - \frac{1}{2^n} = \frac{2^N - 1}{2^n}$	$\frac{1}{2^n}$
N	$Q(N-1)$	$2 - \frac{1}{2^{N-1}}$	$\frac{1}{2^{N-1}}$

Negative values

We are used to representing a negative decimal value simply preceding it's representation by a dash, $-$.

For example the negative of 124_{10} is written as -124_{10}.

This representation has the name: *signed magnitude*.

In general, when two numbers are added to each other with the result equalling 0 one number must be the negative of the other.

Binary signed representation Option 1: signed magnitude

- Treat the most significant bit (left-most) as a sign
  - 1 is negative; 0 is positive.
- 1001 would be -1; 1111 would be -7; 0111 would be 7.
- Problems with this method:
  - Two zeros (0000, 1000)
  - A number and its negative don't add up to zero (1111+0111=0110 $\neq$ 0 !!)
  - What add up to zero are not negatives of each other (0011+1101 (3+(-5)) = 0000 !!)

Option 2: two's complement (1/3)

Make the most significant bit represent a negative value rather than a negative sign:

- $1011_2 = 1 \times (-2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -5$.
- More generally, an N-bit two's compliment bit pattern $b_{N-1}b_{N-2}\cdots b_1 b_0$ has the value

$$v = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_1 2^1 + b_0 2^0.$$

Some observations on two's complement numbers:

- Positive values have a 0 in its MSB; negative values have a 1.
- There is a single zero: 0000
- Max value of an N-bit two's complement number: $2^{N-1} - 1$
- Min value of an N-bit two's complement number: -2^{N-1}.

However, the important question is whether additions will work out?

Converting a number to its negative

Negation of two's complement number: inverting all bits and adding one.

- 4-bit example: $0101_2 = 5 \rightarrow 1010 + 1 = 1011_2 = -8 + 3 = -5$.
- Does not work on the most negative number: out of range. $1000 \rightarrow 0111 + 1 = 1000$.
- Why does it work otherwise?
  - An N-bit two's complement number $b_{N-1}b_{N-2}\cdots b_1 b_0$.
  - It has a value $v = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_1 2^1 + b_0 2^0$.
  - The conversion yields bit pattern $(1-b_{N-1})(1-b_{N-2})\cdots(1-b_1)(1-b_0)$.
  - The new number has value

$$\begin{aligned} v' &= -(1-b_{N-1})2^{N-1} + (1-b_{N-2})2^{N-2} + \cdots + (1-b_0)2^0 + 1 \\ &= -2^{N-1} + 2^{N-2} + \cdots + 2^1 + 2^0 + 1 \\ &\quad - (-b_{N-1}2^{N-1} + b_{N-2}2^{N-1} + \cdots + b_0 2^0) \\ &= -v. \end{aligned}$$

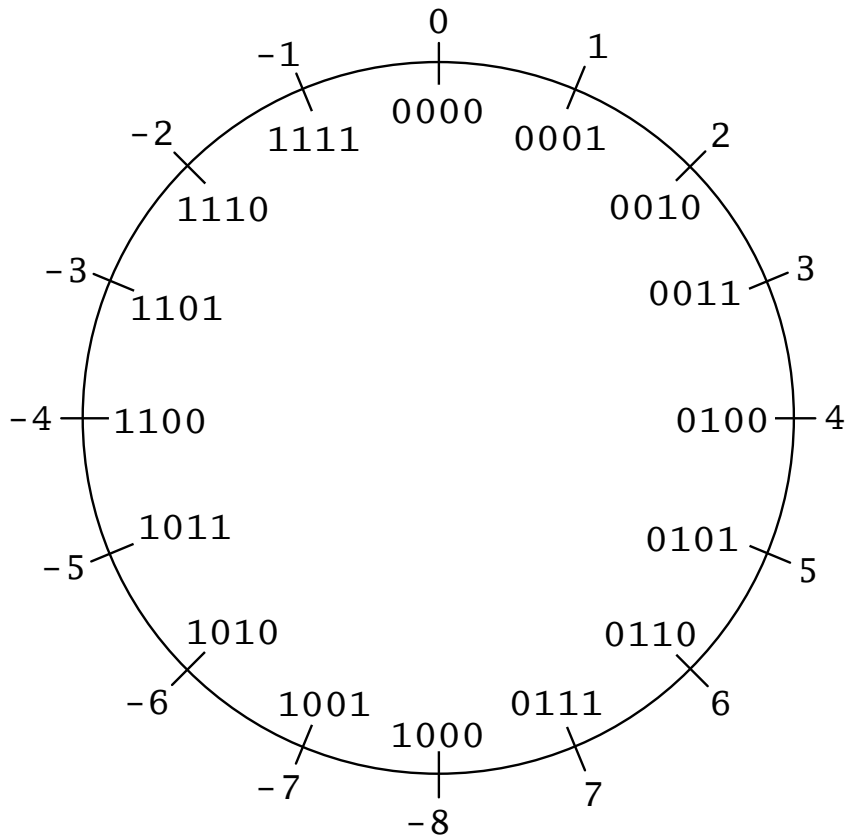
Two's complement circular representation

Note that

- numbers are arranged around the circle,
- they are symmetric about the point 0
- sum of two numbers is equal to sum of angles

This means that regular addition will work:
If two numbers sum to zero, they must be negatives of each other, and vice versa.

$0001 + 1111 = 0$
$0010 + 1110 = 0$
$0011 + 1101 = 0$
$0100 + 1100 = 0$
$0101 + 1011 = 0$
$0110 + 1010 = 0$
$0111 + 1001 = 0$



Two's complement: discussion

The two's complement representation is the most commonly encountered scheme defining negative numbers.

- There is one more negative value than there are positive values.
- It works because bits are lost on overflow: modulo arithmetic.
- The value 0 and the most negative (-8 in the 4-bit example) are self complements.

Sign extension

How to convert an N-bit two's complement number to $N + P$ bits? Duplicate the sign bit (MSB) P times in the leading position.

- Example: $1101_2 = -3$, $11101_2 = -3$, $111101_2 = -3$, and so on.

Why?

- Consider an N-bit two's complement number $b_{N-1} \cdots b_0$.
- It has value $-b_{N-1}2^{N-1} + v$, where v is the value of the lower bits.
- Duplicate the MSB once we get $b_{N-1}b_{N-1} \cdots b_0$, and it has value $-b_{N-1}2^N + b_{N-1}2^{N-1} + v = -b_{N-1}2^{N-1} + v$, same as before!
- Now you can repeat this arbitrary number of times and won't change its value.

The effect of negativity on Qn value range

N	Qn	range	step size
4	$Q0$(xxxx.)	$[-2^3, 2^3 - 1]$	1
4	$Q1$(xxx.x)	$[-2^2, 2^2 - \frac{1}{2}]$	1/2
4	$Q2$(xx.xx)	$[-2^1, 2^1 - \frac{1}{2^2}]$	1/4
4	$Q3$(x.xxx)	$[-1, 1 - \frac{1}{2^3}]$	1/8
N	$Q0$	$[-2^{N-1}, 2^{N-1} - 1]$	1
N	Qn	$[-2^{N-n-1}, 2^{N-n-1} - \frac{1}{2^n}]$	$\frac{1}{2^n}$
N	$Q(N-1)$	$[-1, 1 - \frac{1}{2^{N-1}}]$	$\frac{1}{2^{N-1}}$

Examples of 2's complement Q15 fractions

`0.100 0000 0000 0000` has the value 0.5.

`0.000 0000 0000 0001` has the value 2^{-15}.

`0.111 1111 1111 1111` has the value $1 - 2^{-15}$.

`1.000 0000 0000 0000` has the value -1.

`1.100 0000 0000 0000` has the value -0.5.

`1.111 1111 1111 1111` has the value -2^{-15}.

To make a negative value positive invert all of the bits and add a one in the rightmost bit position.

Note that negating minus one gives minus one (ouch!), complement of itself.

More examples; practice on your own

`0100  0.100 0000  0000` has the value 8.5 (Q11).

`0000  0000  00.00 0001` has the value 2^{-6} (Q6).

`0111  1111  111.1 1111` has the value 1023 31/32 (Q5).

`1111  0000  0.010 0000` has the value -31 3/4 (Q7).

`1111  1111  1111  111.1` has the value -0.5 (Q1).

`1000  00.00 0000  0000` has the value -32 (Q10).

One can interpret a Qn bit pattern as an integer value and then multiply that value by 2^{-n} to get the mixed fractional value.

Overflow

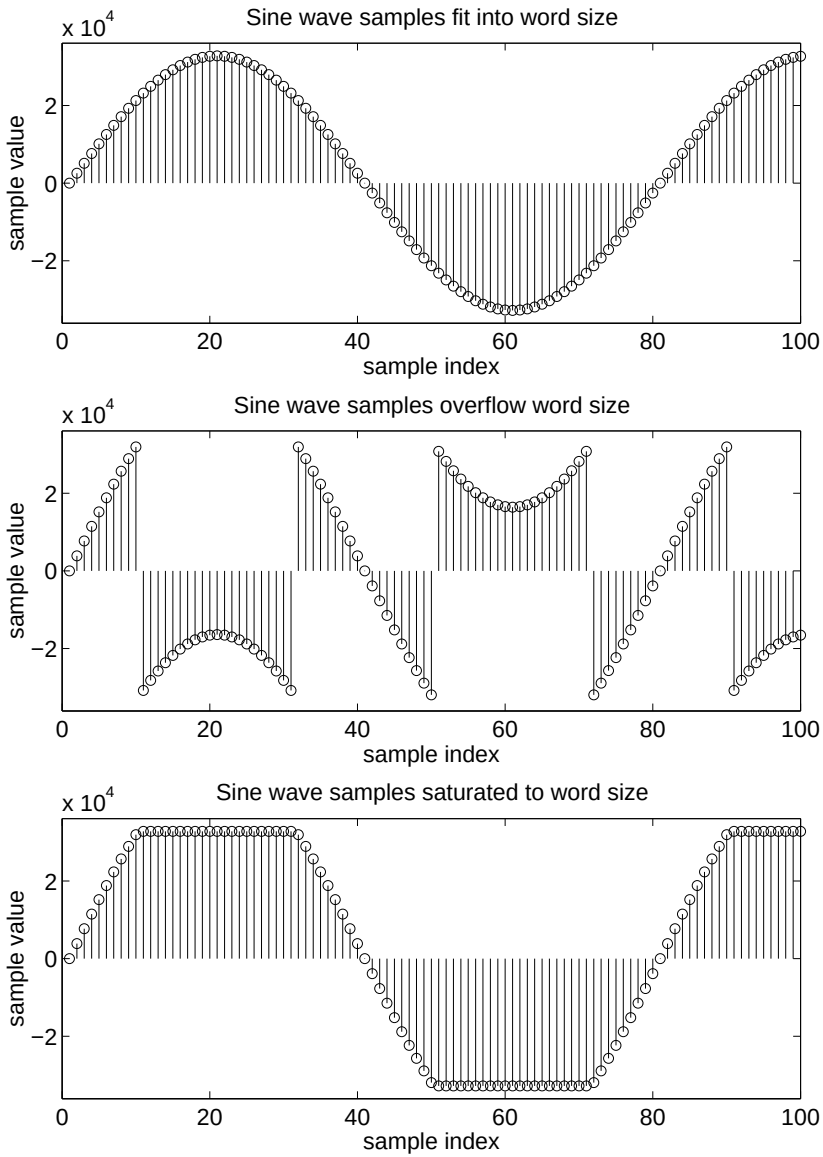
What happens if we add the fractions 0.75 and 0.5? The sum is 1.25 but if we use 6-bit Q5 notation, then

```
  011000      0.75
+ 010000      0.50
--------
  101000     -0.75 !!
```

How do you deal with this problem?

- We could use the Q4 representation ($001100+001000=010100_2 = 1.25$).
- We could use more bits to defer the problem.
- We could design the adder to *saturate* the result to 011111 (0.96875).
- We could try to detect the overflow and do something later...

Which is better: allow overflow or saturate?



The top plot is a sampled sinewave.

The middle plot shows the result if add values of of a smaller amplitude sinewave of same frequency without overflow protection.

The bottom plot shows the effects of saturation. Which, the middle or the bottom waveform, would you rather listen to?

When can an overflow occur in 2's complement?

Let #1 and #2 be two N-bit 2's complement numbers

Overflow occurs when adding two negative numbers and getting a positive result.

$$\text{sign}(\#1) = 1, \;\; \text{sign}(\#2) = 1, \;\; \text{sign}(\text{sum}) = 0.$$

Overflow occurs when adding two positive numbers and getting a negative result.

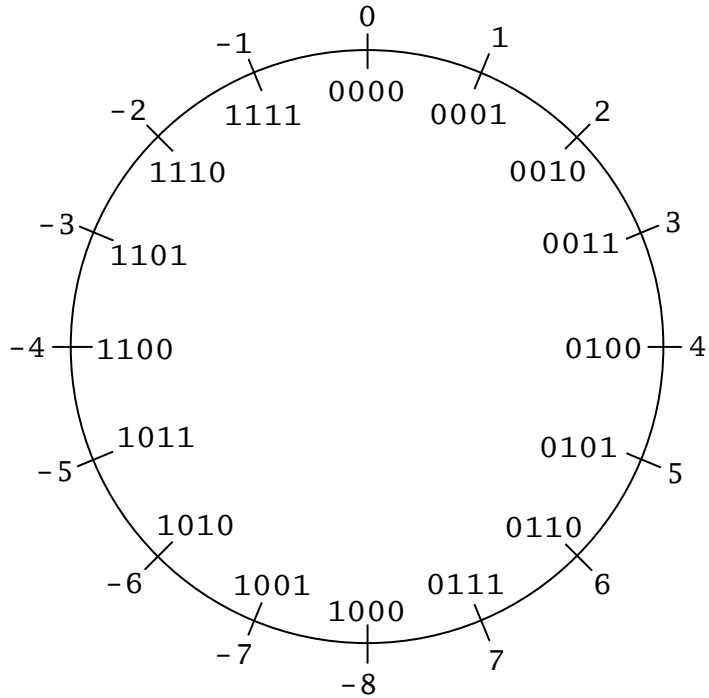
$$\text{sign}(\#1) = 0, \;\; \text{sign}(\#2) = 0, \;\; \text{sign}(\text{sum}) = 1.$$

Overflow cannot occur when adding a negative number and a positive number .

Two's complement overflow property

Given a set of numbers that sum to a value representable using a given word size it does not matter how many times an overflow occurs in forming the sum. The result will be correct.

- This assumes that one does *not* saturate automatically when an overflow occurs.
- A consequence of the cyclic nature of the two's complement representation. One can think of the overflow process as having gone around the number circle as many times in the clockwise direction as in the counter-clockwise direction.
- Depending on the hardware resources present it is generally not wise to saturate early in a series of calculations.



Comments on overflow and saturation (1/2)

A filter design could exploit the two's complement overflow resiliency.

For example: consider an FIR filter having an equal number of positive and negative coefficient values. The filter may not be susceptible to overflow if the terms are ordered in positive/negative pairs. However if all of the positive terms and all the negative terms are summed separately there may be an overflow prior to adding the two sub sums together. Saturating at the time of overflow would be the wrong thing to do in this case.

This is analogous to non-absolutely convergent series

$$\sum_{i=0}^{\infty}(-1)^i \neq \sum_{i=even}(-1)^i + \sum_{i=odd}(-i)^i$$

Using the range $[-1, 1)$ is often convenient

Consider implementing an FIR filter

$$y[n] = \sum_{k=0}^{P} b[k]x[n-k] \, .$$

We pre-scale (pre-normalize) the input samples: $-1 < x[n] < 1$.

We scale the coefficients so that maximum filter gain is 1, i.e. $\max_f |H(f)| = 1$, resulting in coefficient values having magnitude less than 1. This greatly reduces the chance of filter overflow.

- each of the individual products $b[k]x[n-k]$ lie in the range $[-1, 1)$ - no per-term overflow.
- for (most) normalized inputs the filter outputs will lie in the range of $[-1, 1)$.
- We can represent sample values $x[n]$ and coefficient values $b[n]$ using $Q(N-1)$.

Quick summary (1/2)

- Digital devices use binary values organized into words.
  - Fixed word size has finite range and finite resolution and is susceptible to overflow and roundoff error.
  - Bit pattern : $b_{N-1}b_{N-2}\ldots b_2b_1b_0$; decimal value: $v = \sum_{n=0}^{N-1} b_n 2^n$.
- Qn notation: binary point separates integer and fraction parts.
  - Binary point does not have physical presence, must keep track.
  - Need to align Qn and Qm values prior to adding.

Quick summary (2/2)

- Signed values typically use two's complement form.
  - Two's complement value: $-b_{N-1}2^{N-1} + \sum_{n=0}^{N-2} b_n 2^n$.
  - To get the negative: invert bits and add 1.
  - Unsigned and two's complement addition use same hardware. Not necessarily so for multiplication!
- When adding values, sum might not fit (overflow).
  - Solutions: do nothing, add bits, saturate.
  - Two's complement is robust to intermediate overflows!
  - Generally saturate only when storing sum!

Multiplication in binary arithmetic

$$\text{multiplicand} \times \text{multiplier} = \text{product}$$

multiplicand is what is being multiplied

multiplier what is doing the multiplication

product is the result

Unsigned binary multiplication

How large is the product of the two largest unsigned N-bit integers?

$$(2^N - 1) \times (2^N - 1) = 2^{2N} - 2^{N+1} + 1$$

Assume $N = 8$.

The value of $255 \times 255 = 65,025 = 1111\ 1110\ 0000\ 0001_2$.

Generalizing, the product of two N-bit unsigned binary numbers can be up to $2N$ bits in length.

An example

Let's try 7×12 or 0111×1100:

```
          0111
        x 1100
        ----------
          0000
         0000
        0111
       0111
  --------------
     01010100
```

Which is the same as 1010100 after discarding the MSB 0 bit:
$2^6 + 2^4 + 2^2 = 84$. Correct answer!

Two's complement multiplication

Four cases:

Positive multiplier times positive multiplicand.

Positive multiplier times negative multiplicand.

Negative multiplier time positive multiplicand.

Negative multiplier times negative multiplicand.

Brute force would be to negate any negative values, multiply using unsigned multiplication hardware and, if necessary, negate the result. Sometimes brute force is a reasonable way to go!

Positive times positive

No problem.

Because b_{N-1} is zero in both cases, unsigned and two's complement values will never overflow as long as 2^{N-1} register used to store result.

Maximum positive value is: $2^{N-1} - 1$.

Product of two maximum values is: $2^{2N-2} - 2^N + 1$.

Illustrating using 8 bits:

$$127 \times 127 = 16,129 = 0011\ 1111\ 0000\ 0001_2\,.$$

Only need $2N - 1$ bits to hold the result (including one bit for the sign, which in this case is 0).

Product will normally be placed into two N-bit words.

Negative times negative

Most negative value is -2^{N-1}

Expected product is 2^{2N-2}. Fits into $2N-1$ bits.

Illustrate using 8-bit values and a 16 bit accumulator

Expected: $(-128) \times (-128) = 16,384 = 0100\ 0000\ 0000\ 0000_2$

No overflow if we use a $2N$-bit accumulator

Positive times negative

Largest value of positive multiplicand is: $2^{N-1} - 1$, or $0111\ldots1111$

Largest value of negative multiplier is -2^{N-1}, or $1000\ldots0000$

Their product should be: $-2^{2N-2} + 2^{N-1}$ since

$$(2^{N-1} - 1) \times 2^{N-1} = 2^{2N-2} - 2^{N-1}$$

Unsigned multiplier gets it wrong!

Illustrate using 4-bit input and 8 bit accumulator:

Expected: $7 \times (-8) = -56 = 1100\ 1000_2$.

Got: $0011\ 1000_2$.

Unsigned and signed multiplication differ!

Need a multiplier designed for use with two's complement values.

One solution

The same hardware for both unsigned and signed multiplication:

- add one extra bit sign extension for 2's complement,
- or, add an extra 0 in the MSB for unsigned numbers,
- followed by signed multiplication.

Consider the two 4-bit numbers $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$. The above operation is done as follows:

$$
\begin{array}{cccccccccc}
 & b_0 & \times & & & & a_3 & a_3 & a_2 & a_1 & a_0 \\
+ & b_1 & \times & & & a_3 & a_3 & a_2 & a_1 & a_0 & \\
+ & b_2 & \times & & a_3 & a_3 & a_2 & a_1 & a_0 & & \\
+ & b_3 & \times & a_3 & a_3 & a_2 & a_1 & a_0 & & & \\
- & b_3 & \times & a_3 & a_3 & a_2 & a_1 & a_0 & & & \\
\hline
 & & & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

- The b_i multiplications are by 0 or 1. Rows are added using unsigned addition.
- The last subtraction is done using addition: complement all bits and add 1.

Let's try the previous example

We will get to talk about why this works in the next lecture. For now we will only try to verify this with some examples.

Try again $7 \times (-8)$ or 0111×1000:

```
          00111
        x 11000
       ----------
          00000
         00000
        00000
       00111
 (-)00111          (row corresponding to the MSB of b)
    11001          (flip all bits and add 1)
---------------
    111001000
```

Which is the same as 11001000 after discarding the MSB (a repeated sign bit) in 2's complement: $-2^7 + 2^6 + 2^3 = -56$. Correct answer!

An unsigned example

Let's try 7×12 or 0111×1100:

```
          00111
        x 01100
        ----------
          00000
         00000
        00111
       00111
 (-)00000          (row corresponding to the MSB of b)
    00000          (recall that 0 is self-complement)
---------------
    001010100
```

Which is the same as 01010100 after discarding the MSB 0 bit:
$2^6 + 2^4 + 2^2 = 84$. Correct answer!

Note: the C55x has a 17-bit multiplier

The C5515 has a 16-bit word size and possess a 17-bit multiplier.

The hardware is significantly different for multiplying two 16-bit signed values from that for multiplying two 16-bit unsigned values. (You will get to see this when we talk about FPGA)

Note: a N+1 bit signed multiplier can be used to correctly multiply two unsigned N bit values.

The C5515 when doing signed multiplication extends the signs of the values into the 17th bits and does a signed multiplication.

The C5515 when doing unsigned multiplication places zeros into the 17th bits and does a signed multiplication.

Multiplication of Q values

Consider the following decimal multiplication problem:

```
   1.23
x 2.013
-------
    369
   123
    0
 246
-------
 247599
```

Where does the decimal point go? Why?

If we multiply a Qm value by a Qn value what is the Q number of the product?

Multiplying (mixed) fraction values

Drawing from our training on multiplying decimal fractions we realize that if we multiply a Qs number by a Qn number the number of fractional bits must be

$$s + n.$$

Assuming a N-bit word size, if we use a double word to hold the product then the number of bits available to represent the integer part of a mixed fraction must be

$$2N - 1 - s - n.$$

Illustrative binary multiplication example:

```
    11.101     Q3
  x   1.1      Q1
----------
    11101
   11101
----------
  1010111  ---> 101.0111   Q4
```

The two primary fixed point concerns

Other than getting the algorithm correct the two key concerns when using fixed point are felt to be:

- Guarding against overflow.
  - Using extra guarding bits.
  - Rounding (reducing the number of bits used to address overflow).
  - Saturation.
- Minimizing round off errors when values have their word sizes reduced.

Floating point

Floating point is a different representation where the radix point is not in fixed location.

- Using part of the bits (in a fixed size word) to specify where the binary point is (a *floating* Q number).
- A floating number has two parts:
  - the *mantissa*, or the value of the digits;
  - the *characteristic*, or power/exponent of the base.
- In a standard 32-bit floating number, we use 24 bits for the signed mantissa (s) in Q23 format, and 8 bits for the exponent p, generating a value of $s \times 2^p$.

The C55x C/C++ compiler supports floating point calculations.

The C55x does not possess floating point hardware. All floating point calculations are simulated . . . done in software.

For the C55x both floats and doubles use 32-bits.

Summary of what we covered today

- Fixed point arithmetic
  - Number representation; radix; conversion between different bases/radix.
  - Q-notation; fractions; range of representation.
  - Negative values: two's complement numbers.
  - Overflow and saturation.
  - Signed multiplication.
- Next: Gate level arithmetic

Some references

Real time digital signal processing, Kuo and Lee.

Two's complement, `http://en.wikipedia.org/wiki/Two's_complement`

Computer Arithmetic Algorithms and Hardware Designs, B. Parhami.

Computer Arithmetic Algorithms, 2nd ed, I. Koren.

Digital-Serial Computation, R. Harley and K. Parhi.