

Fixed Point Numbers

- The binary integer arithmetic you are used to is known by the more general term of **Fixed Point** arithmetic.
 - *Fixed Point* means that we view the decimal point being in the same place for all numbers involved in the calculation.
 - For integer interpretation, the decimal point is all the way to the right

$$\begin{array}{r} \$C0 \\ + \$25 \\ \hline \\ \$E5 \end{array}$$

$$\begin{array}{r} 192. \\ + 37. \\ \hline 229. \end{array}$$

Unsigned integers, decimal point to the right.

A common notation for fixed point is ‘X.Y’, where X is the number of digits to the left of the decimal point, Y is the number of digits to the right of the decimal point.

Fixed Point (cont).

- The decimal point can actually be located anywhere in the number -- to the right, somewhere in the middle, to the right

Addition of two 8 bit numbers; different interpretations of results based on location of decimal point

$$\begin{array}{r} \$11 \\ + \$1F \\ \hline \$30 \end{array}$$

$$\begin{array}{r} 17 \\ + 31 \\ \hline 48 \end{array}$$

$$\begin{array}{r} 4.25 \\ + 7.75 \\ \hline 12.00 \end{array}$$

$$\begin{array}{r} 0.07 \\ + 0.12 \\ \hline 0.19 \end{array}$$

xxxxxxxx.0
decimal point to right.
This is 8.0 notation.

xxxxxx.yy
two binary fractional
digits. This is 6.2
notation.

0.yyyyyyyyy
decimal point to left (all
fractional digits). This is
0.8 notation.

Algorithm for converting fractional decimal to Binary

An algorithm for converting any fractional decimal number to its binary representation is successive multiplication by two (results in shifting left). Determines bits from MSB to LSB.

- a. Multiply fraction by 2.
- b. If number ≥ 1.0 , then current bit = 1, else current bit = 0.
- c. Take fractional part of number and go to 'a'. Continue until fractional number is 0 or desired precision is reached.

Example: Convert .5625 to binary

$$\begin{aligned} .5625 \times 2 &= 1.125 \quad (\geq 1.0, \text{ so MSB bit} = '1') \\ .125 \times 2 &= .25 \quad (< 1.0 \text{ so bit} = '0') \\ .25 \times 2 &= .5 \quad (< 1.0 \text{ so bit} = '0') \\ .5 \times 2 &= 1.0 \quad (\geq 1.0 \text{ bit} = 1), \text{ finished.} \\ .5625 &= .1001\text{b} \end{aligned}$$

Unsigned Overflow

- Recall that a carry out of the Most Significant Digit is an unsigned overflow. This indicates an error - the result is NOT correct!

Addition of two 8 bit numbers; different interpretations of results based on location of decimal point

$$\begin{array}{r}
 \$FF \\
 + \$01 \\
 \hline
 \$00
 \end{array}$$

XXXXXXXX.0
decimal point to right

$$\begin{array}{r}
 255 \\
 + 1 \\
 \hline
 0
 \end{array}$$

$$\begin{array}{r}
 63.75 \\
 + 0.25 \\
 \hline
 0
 \end{array}$$

XXXXXX.yy
two binary fractional
digits (6.2 notation)

$$\begin{array}{r}
 0.99600 \\
 + 0.00391 \\
 \hline
 0
 \end{array}$$

0.yyyyyyyy
decimal point to left (all
fractional digits). This
0.8 notation

Saturating Arithmetic

- Saturating arithmetic means that if an overflow occurs, the number is clamped to the maximum possible value.
 - Gives a result that is closer to the correct value
 - Used in DSP, Graphic applications.
 - Requires extra hardware to be added to binary adder.
 - Pentium MMX instructions have option for saturating arithmetic.

\$FF	255	63.75	0.99600
+ \$01	+ 1	+ 0.25	+ 0.00391
-----	-----	-----	-----
\$FF	255	63.75	0.99600

XXXXXXXXX.0
decimal point to right

XXXXXX.yy
two binary fractional
digits.

0.yyyyyyyyyy
decimal point to left (all
fractional digits)

Saturating Arithmetic

The Intel Xx86 MMX instructions perform SIMD operations between MMX registers on packed bytes, words, or dwords.

The arithmetic operations can be made to operate in Saturation mode.

What saturation mode does is clip numbers to Maximum positive or maximum negative values during arithmetic.

In normal mode: $FFh + 01h = 00h$ (unsigned overflow)

In saturated, unsigned mode: $FFh + 01 = FFh$ (saturated to maximum value, closer to actual arithmetic value)

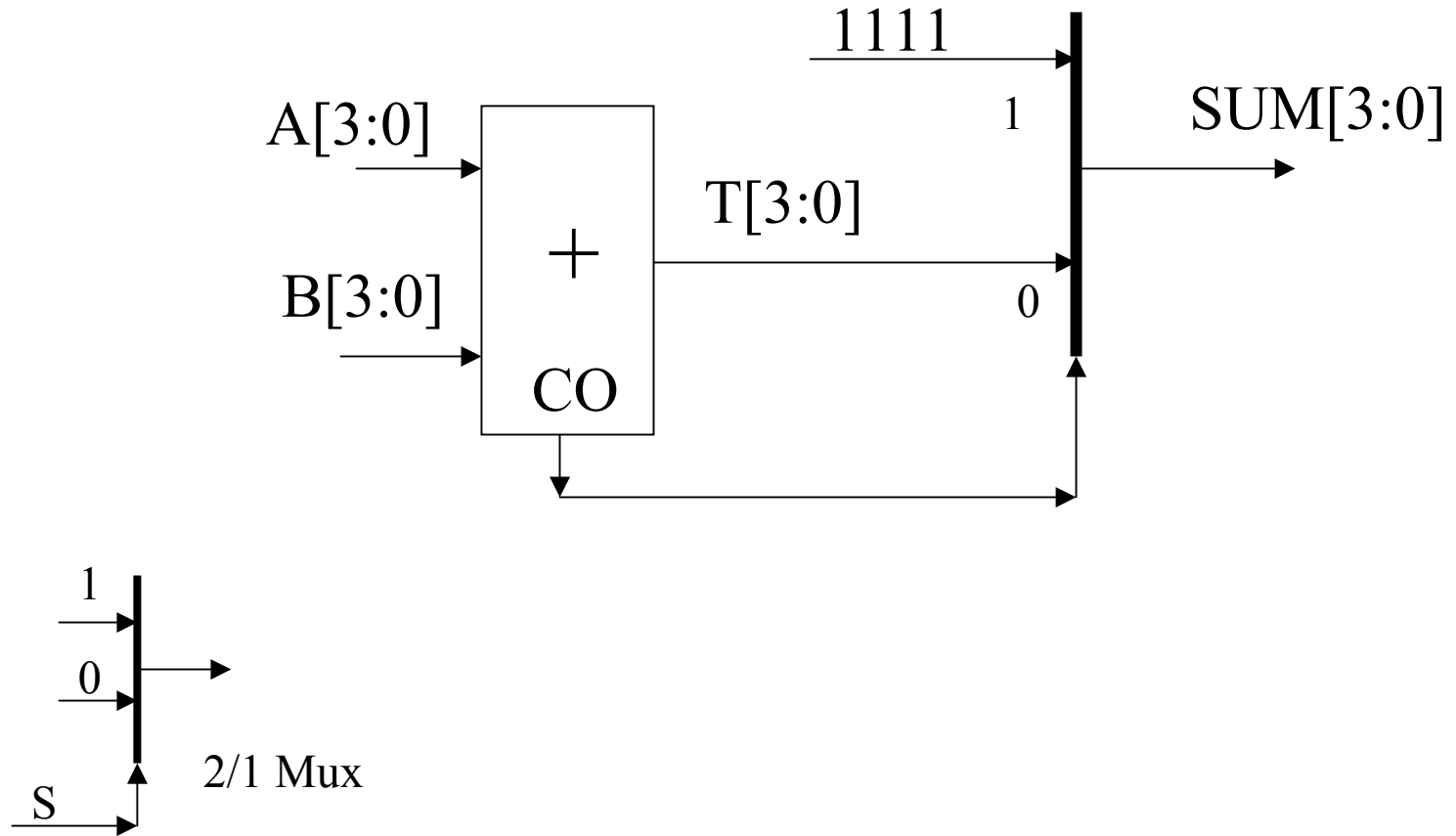
In normal mode: $7fh + 01h = 80h$ (signed overflow)

In saturated, signed mode: $7fh + 01 = 7fh$ (saturated to max value)

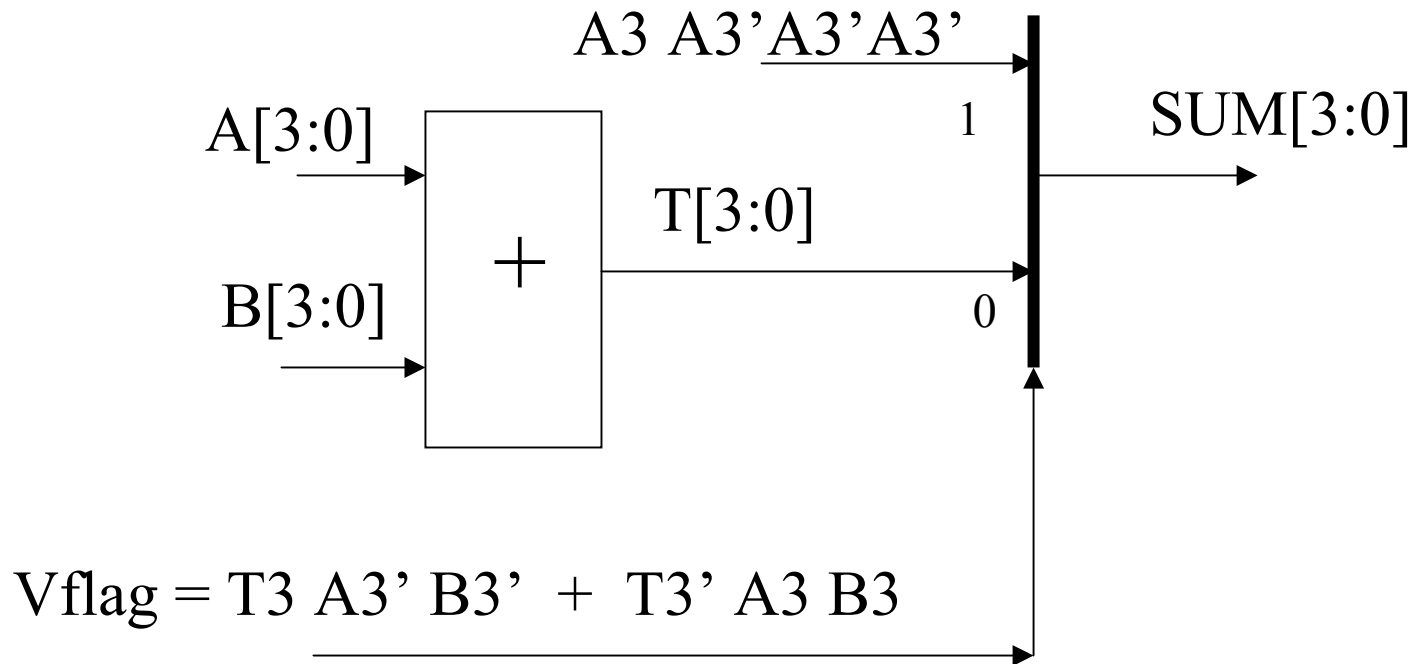
Saturating Adder: Unsigned and 2' Complement

- For an unsigned saturating adder, 8 bit:
 - Perform binary addition
 - If Carryout of MSB = 1, then result should be a \$FF.
 - If Carryout of MSB = 0, then result is binary addition result.
- For a 2's complement saturating adder, 8 bit:
 - Perform binary addition
 - If Overflow = 1, then:
 - If one of the operands is negative, then result is \$80
 - If one of the operands is positive, then result is \$7f
 - If Overflow = 0, then result is binary addition result.

Saturating Adder: Unsigned, 4 Bit example



Saturating Adder: Signed, 4 Bit example



Vflag is true if sign of both operands are the same (both negative, both positive) and different from Sum (overflow if add two positive numbers, get a negative or add two negative numbers and get a positive number. Can't get overflow if add a positive and a negative).

Saturated value has same sign as one of the operands, with other bits equal to NOT (sign) : 0111 (positive saturation), 1000 (negative saturation).

Saturating Arithmetic

The MMX instructions perform SIMD operations between MMX registers on packed bytes, words, or dwords.

The arithmetic operations can be made to operate in Saturation mode.

What saturation mode does is clip numbers to Maximum positive or maximum negative values during arithmetic.

In normal mode: $\text{FFh} + 01\text{h} = 00\text{h}$ (unsigned overflow)

In saturated, unsigned mode: $\text{FFh} + 01 = \text{FFh}$ (saturated to maximum value, closer to actual arithmetic value)

In normal mode: $7\text{fh} + 01\text{h} = 80\text{h}$ (signed overflow)

In saturated, signed mode: $7\text{fh} + 01 = 7\text{fh}$ (saturated to max value)

Why Saturating Arithmetic?

- In case of integer overflow (either signed or unsigned), many applications are satisfied with just getting an answer that is close to the right answer or saturated to maximum result
- Many DSP (Digital Signal Processing) algorithms depend on this feature
 - Many DSP algorithms for audio data (8 to 16 bit data) and Video data (8-bit R,G,B values) are integer based, and need saturating arithmetic.
- This is easy to implement in hardware, but slow to emulate in software. A nice feature to have.

Floating Point Representations

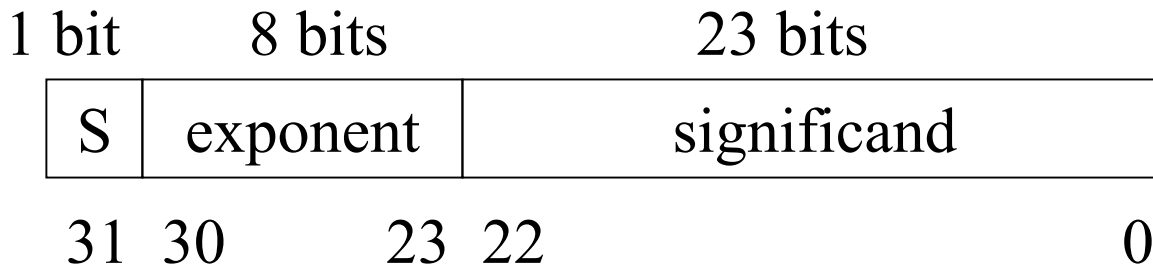
- The goal of floating point representation is represent a large range of numbers
- Floating point in decimal representation looks like:
 $+3.0 \times 10^3$, 4.5647×10^{-20} , etc
- In binary, sample numbers look like:
 -1.0011×2^4 , 1.10110×2^{-3} , etc
- Our binary floating point numbers will always be of the general form:
 $(sign) 1.mmmmmm \times 2^{exponent}$
- The sign is positive or negative, the bits to the right of decimal point is the mantissa or *significand*, exponent can be either positive or negative. The numeral to the left of the decimal point is ALWAYS 1 (normalized notation).

Floating Point Encoding

- The number of bits allocated for exponent will determine the maximum, minimum floating point numbers (range)
 1.0×2^{-max} (small number) to
 $1.0 \times 2^{+max}$ (large number)
- The number of bits allocated for the significand will determine the precision of the floating point number
- The sign bit only needs one bit (negative:1, positive: 0)

Single Precision, IEEE 754

Single precision floating point numbers using the IEEE 754 standard require 32 bits:



Exponent encoding is *bias 127*. To get the encoding, take the exponent and add 127 to it.

If exponent is -1 , then exponent field = $-1 + 127 = 126 = 7Eh$

If exponent is 10, then exponent field = $10 + 127 = 137 = 89h$

Smallest allowed exponent is -126 , largest allowed exponent is $+127$. This leaves the encodings 00H, FFH unused for normal numbers.

Convert Floating Point Binary Format to Decimal

1 10000001 010000.....0

S	exponent	significand
---	----------	-------------

What is this number?

Sign bit = 1, so negative.

Exponent field = 81h = 129.

Actual exponent = Exponent field - 127 = 129 - 127 = 2.

Number is:

$$-1 \cdot (01000...000) \times 2^2$$

$$-1 \cdot (0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} \dots + 0) \times 4$$

$$-1 \cdot (0 + 0.25 + 0 + \dots 0) \times 4$$

$$-1.25 \times 4 = -5.0.$$

Convert Decimal FP to binary encoding

What is the number -28.75 in Single Precision Floating Point?

1. Ignore the sign, convert integer and fractional part to binary representation first:

a. $28 = 1Ch = 0001\ 1100$

b. $.75 = .5 + .25 = 2^{-1} + 2^{-2} = .11$

-28.75 in binary is $- 00011100.11$ (ignore leading zeros)

2. Now NORMALIZE the number to the format $1.mmmm \times 2^{\text{exp}}$
Normalize by shifting. Each shift right add one to exponent, each shift left subtract one from exponent:

$$\begin{aligned} - 11100.11 \times 2^0 &= - 1110.011 \times 2^1 \\ &= - 111.0011 \times 2^2 \\ &= - 1.110011 \times 2^4 \end{aligned}$$

Convert Decimal FP to binary encoding (cont)

Normalized number is: **- 1.110011 x 2⁴**

Sign bit = 1

Significand field = 110011000...000

Exponent field = 4 + 127 = 131 = 83h = 1000 0011

Complete 32-bit number is:

1 10000011 110011000....000

S	exponent	significand
---	----------	-------------

Overflow/Underflow, Double Precision

- Overflow in floating point means producing a number that is too big or too small (underflow)
 - Depends on Exponent size
 - Min/Max exponents are 2^{-126} to 2^{+127}
is 10^{-38} to 10^{+38} .
- To increase the range, need to increase number of bits in exponent field.
- Double precision numbers are 64 bits - 1 bit sign bit, 11 bits exponent, 52 bits for significand
- Extra bits in significand gives more precision, not extended range.

Special Numbers

Min/Max exponents are 2^{-126} to 2^{+127} .

This corresponds to exponent field values of 1 to 254.

The exponent field values 0 and 255 are reserved for *special numbers*. *Special Numbers* are zero, +/- infinity, and NaN (not a number)

Zero is represented by ALL FIELDS = 0.

+/- Infinity is Exponent field = 255 = FFh, significand = 0. +/- Infinity is produced by anything divided by 0.

NaN (Not A Number) is Exponent field = 255 = FFh, significand = nonzero. NaN is produced by invalid operations like zero divided by zero, or infinity – infinity.

Comments on IEEE Format

- Sign bit is placed in MSB for a reason – a quick test can be used to sort floating point numbers by sign, just test MSB
- If sign bits are the same, then extracting and comparing the exponent fields can be used to sort Floating point numbers. A larger exponent field means a larger number since the ‘bias’ encoding is used.
- All microprocessors that support Floating point use the IEEE 754 standard. Only a few supercomputers still use different formats.