Modern Assembly Language Programming with the ARM processor

Chapter 8: Non-Integral Mathematics

1. Base Conversion of Nonintegral Numbers
2. Fixed Point Calculations
3. Fixed Point Multiplication and Division
4. Fixed Point Sine and Cosine
5. Floating Point
6. Algorithms for Floating Point
7. Algorithms for Floating Point

Converting Fractions in Any Base to Decimal

The value 101.0101_2 can be converted to base ten by expanding it as follows:

$$\begin{aligned} & 1\times 2^2+0\times 2^1+1\times 2^0+0\times 2^{-1}+1\times 2^{-2}+0\times 2^{-3}+1\times 2^{-4} \\ = \quad & 4+0+1+0+\frac{1}{4}+0+\frac{1}{16} \\ = \quad & 5.3125_{10} \end{aligned}$$

Likewise, the hexadecimal fraction 4F2.9A0 can be converted to base ten by expanding it as follows:

$$\begin{aligned} & 4\times 16^2+15\times 16^1+2\times 16^0+9\times 16^{-1}+10\times 16^{-2}+0\times 16^{-3} \\ = \quad & 1024+240+2+\frac{9}{16}+\frac{10}{256}+\frac{0}{4096} \\ = \quad & 1266.6015625_{10} \end{aligned}$$

Converting Decimal Fractions to Binary

- The integer part is converted by repeated division (Chapter 1).
- The fractional part is converted by repeated multiplication.

Example: Convert the decimal value 5.625 to a binary representation. The integer part is 101_2.

1. Multiply the decimal fraction by two. The integer part of the result is the first binary digit to the right of the radix point.
   Because $.625 \times 2 = 1.25$, the first binary digit to the right of the point is a 1. So far, we have $.625_{10} = .1_2$
2. Disregard the integer portion and multiply by 2 once again.
   Because $.25 \times 2 = 0.50$, the second binary digit to the right of the point is a 0. So far, we have $.625_{10} = .10_2$
3. Disregard the integer portion and multiply by 2 once again.
   Because $.50 \times 2 = 1.00$, the third binary digit to the right of the point is a 1. So now we have $.625 = .101$
4. Disregard the integer portion. Since that leaves us with 0.0, we know that all remaining digits will be zero.

$$5.625_{10} = 101.101_2$$

Nonterminating (Repeating) Binary Fractions

In Radix Notation there are some fractions that repeat and never terminate.
Example: The binary representation of the decimal fraction 1/10 is a non-terminating repeating fraction.

$$
\begin{aligned}
.1 \times 2 &= 0.2 \\
.2 \times 2 &= 0.4 \\
.4 \times 2 &= 0.8 \\
.8 \times 2 &= 1.6 \\
.6 \times 2 &= 1.2 \\
.2 \times 2 &= 0.4
\end{aligned}
$$

The last step is exactly the same as the second step.

If we continue, we will repeat the sequence of steps 2-5 forever.

Hence, the final binary representation will be.

$$0.1_{10} = .0001100110011001\ldots_2$$

$$0.1_{10} = .\overline{0011}_2$$

Nonterminating Binary Fractions - How Many?

Question: Are there, in some sense of the word, more nonterminating decimals than binimals, more nonterminating binimals than decimals, or neither?

Answer: Obviously, since both are infinite sets, in one sense of the word, the answer is neither. But that is an oversimplification.

If we ask our question differently, we can discover some important information.

Question: Is the set of terminating decimals a subset of the set of terminating binimals, or vice versa, or neither.

Answer: The set of terminating binimals is a subset of the set of terminating decimals, but the set of terminating decimals is not a subset of the set of terminating binimals.

Nonterminating Binary Fractions - Lemma

Lemma

If x, $0 < x < 1$, terminates in some base B (a product of primes), then $x = \frac{N_x}{D_x}$, and $D_x = p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}$, where the p_i are the prime factors of B.

Proof.

Let $x = \frac{N_x}{D_x}$, and $D_x = p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}$, where the p_i are the prime factors of B. Then $D_x \mid N_x \times B^{k_{max}}$, where $k_{max} = max(k_1, k_2, \ldots k_n)$, so $x = \frac{N_x}{D_x x}$ terminates after k_{max} or fewer divisions.

Let $x = \frac{N_x}{D_x}$ terminate after k divisions. Then $D_x \mid N_x \times B^k$. Since D_x does not evenly divide N_x, D_x must be composed of some combination of the prime factors of B. Thus, D_x can be expressed as $p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}$. □

Nonterminating Binary Fractions - Theorem

Theorem

The set of terminating binimals is a subset of the set of terminating decimals, but the set of terminating decimals is not a subset of the set of terminating binimals.

Proof.

Let b be a terminating binimal. Then, by the Lemma on the previous slide, $b = \frac{N_b}{D_b}$, such that $D_b = 2^k$, for some $k \geq 0$. Therefore, $D_b = 2^k 5^m$, for some $k, m > 0$, and again by the Lemma, b, is also a terminating decimal. $\square$

Nonterminating Binary Fractions - Examples

Let $B = 30 = 2 * 3 * 5$. Then any number with denominator $2^{k_1}3^{k_2}5^{k_3}$ terminates in base B.

For example, $\frac{11}{15}$ will terminate in base 30 after one division since $15 = 3^1 5^1$.

To see what the number will look like, let's extend the hexadecimal system of using letters to represent digits beyond 9. So we get this chart for base 30:

10–A	12–C	14–E	16–G	18–I	20–K	22–M	24–O	26–Q	28–S
11–B	13–D	15–F	17–H	19–J	21–L	23–N	25–P	27–R	29–T

Since $\frac{11}{15} = \frac{22}{30}$, it is $0.\mathrm{M}_{30}$.

What is $\frac{13}{45}$ in base 30? Since $45 = 3^3 5^1$, this number will have two digits following the radix point. To compute the value, we will have to raise it to higher terms, using 30^2 as the denominator:

$$\frac{13}{45} = \frac{260}{900}.$$

$\frac{260}{30} = 8$ with remainder 20, so $\frac{13}{45} = 0.8\mathrm{K}_{30}$.

Nonterminating Binary Fractions - Bottom Line

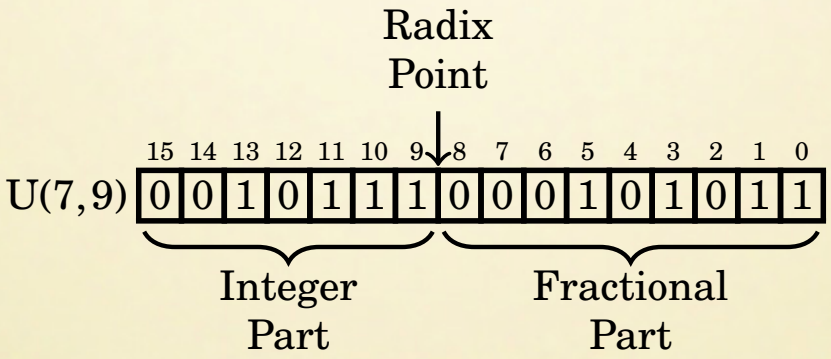
Since the set of terminating binimals is a subset
of the set of terminating decimals,
but not vice versa:

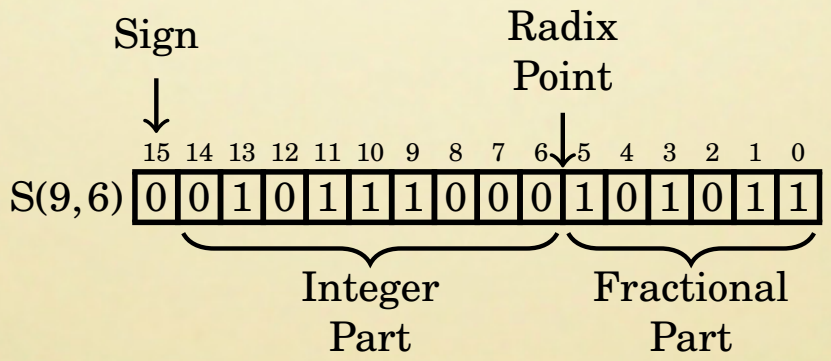
- there are terminating fractions in base 10 which do not terminate in base 2, but
- all fractions that terminate in base 2 will also terminate in base 10.

No number system is immune from this problem. There are fractions in base 30, for example, which do not terminate.
Changing the base of the computer's number system is not the answer.

Fixed Point Numbers

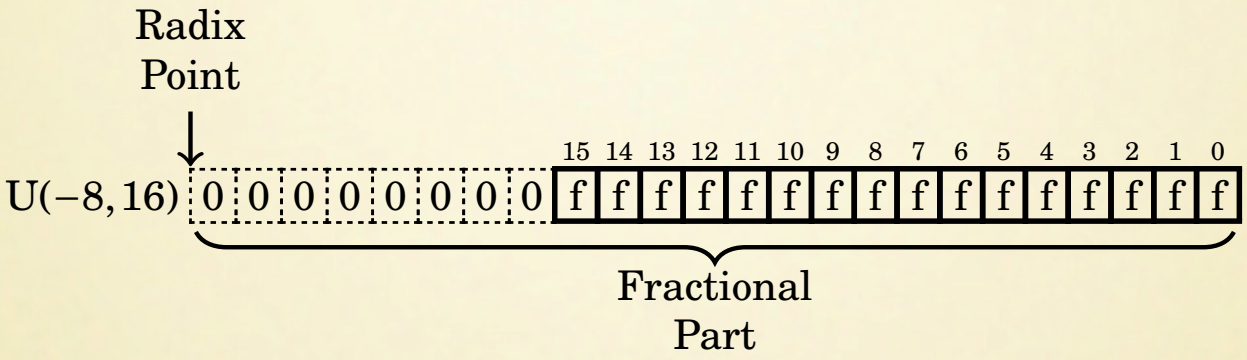
- Given a finite number of bits, a computer can only *approximately* represent nonintegral numbers.
- Some computers have no hardware support for nonintegral calculations.
- Real numbers can be represented as *fixed point* numbers.

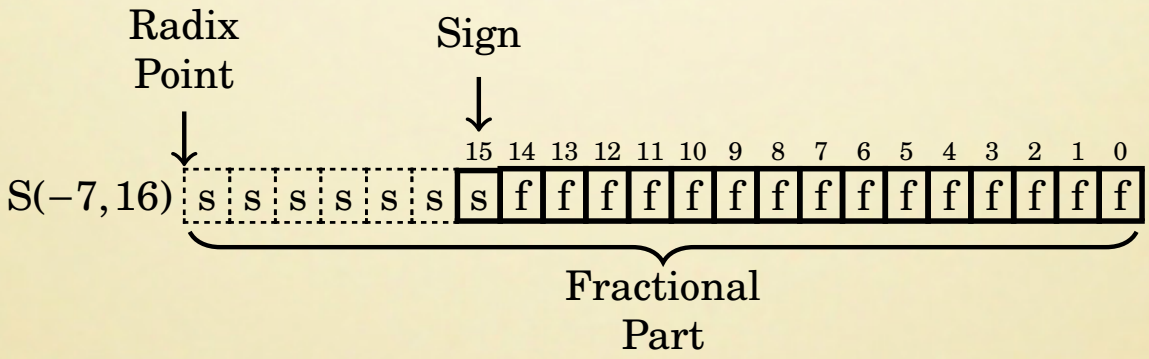




Fixed Point Numbers Continued

Sometimes it is useful to represent a negative number of bits in the integer part.





Size and Radix Point

The combination of size and radix will affect several properties.

Precision: the maximum number of non-zero bits representable,

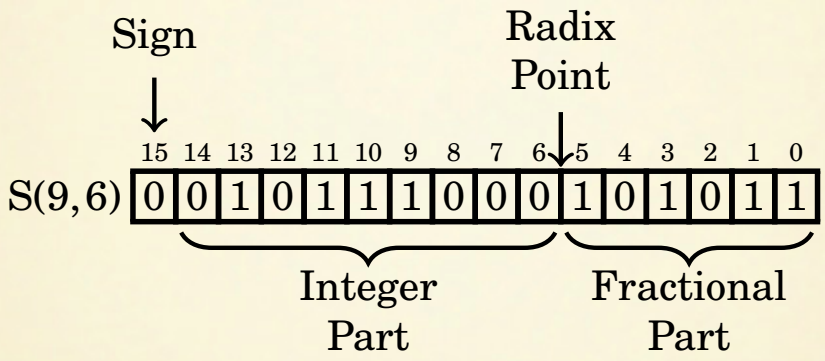
Resolution: the smallest non-zero magnitude representable,

Accuracy: the magnitude of the maximum difference between a true real value and it's approximate representation,

Range: the diffence between the greatest and smallest number that can be represented,

Dynamic range: the ratio of the maximum absolute value, and and the minimum positive absolute value representable.

Size and Radix Point – Example



Precision: 16 bits

Resolution: $A = 2^{-6} = 0.015625$

Accuracy: $A = \frac{R}{2} = 0.0078125$

Range: Minimum value is $1111111111.111111 = -512$
Maximum value is $0111111111.111111 = 511.9921875$
Range is $511.9921875 + 512 = 1023.9921875$

Dynamic range: For a signed fixed-point rational representation, $S(a,b)$, the dynamic range is

$$2 \times \frac{2^a}{2^{-b}} = 2^{a+b+1} = 2^N \qquad 2^{16} = 65536$$

Adding and Subtracting

Fixed point addition and subtraction work exactly like their integer counterparts.

2.25		00010.010
+ 1.50	=	+ 00001.100
3.75		00011.110

11.125		01011.001
− 5.625	=	+ 11010.011
5.500		00101.100

−12.375		10011.101
+ 5.250	=	+ 00101.010
− 7.125		11000.111

In fact, integer math is just fixed point with no bits in the fractional part.

Multiplication

The result of multiplying an n bit number by an m bit number is an $n+m$ bit number

$$
\begin{array}{r}
3.75 \\
\times\ 2.50 \\
\hline
.0000 \\
1.875 \\
7.50 \\
\hline
9.3750
\end{array}
\quad = \quad
\begin{array}{r}
00011.110 \\
\times\ 00010.100 \\
\hline
0001.1110 \\
000111.10 \\
\hline
0000001001.011000
\end{array}
$$

Note that the radix point in the result has shifted.

The result of multiplying two numbers $S(a_1, b_1)$ and $S(a_2, b_2)$ is an $S(a_1 + a_2 + 1, b_1 + b_2)$ number.

Fixed point multiplies may be followed by an appropriate shift.

Fixed Point Multiplication Result Formats

Unsigned Multiplication

The result of multiplying two unsigned numbers $\mathrm{U}(i_1,f_1)$ and $\mathrm{U}(i_2,f_2)$ is a $\mathrm{U}(i_1+i_2,f_1+f_2)$ number.

Mixed Multiplication

The result of multiplying a signed number $\mathrm{S}(i_1,f_1)$ and an unsigned number $\mathrm{U}(i_2,f_2)$ is an $\mathrm{S}(i_1+i_2,f_1+f_2)$ number.

Signed Multiplication

The result of multiplying two signed numbers $\mathrm{S}(i_1,f_1)$ and $\mathrm{S}(i_2,f_2)$ is an $\mathrm{S}(i_1+i_2+1,f_1+f_2)$ number.

Fixed Point Multiplication on ARM

```
1     @@ Multiply two S(10,5) numbers
2     mul   r0,r1,r2       @ x = a * b -> S(21,10)
3
4     @@ Multiply two U(12,4) numbers and produce a U(12,4)
5     mul   r3,r4,r5       @ x = a * b -> U(24,8)
6     lsr   r3,r3,#4       @ shift back to U(12,4)
7
8     @@ Multiply two S(16,15) numbers a produce an S(16,15)
9     smull r0,r1,r2,r3    @ x = a * b -> S(33,30)
10    lsr   r0,r0,#17      @ get 15 bits from r0
11    orr   r0,r1,lsl #15 @ combine with 17 bits from r1
12
13    @@ Multiply two U(10,22) numbers
14    umull r0,r1,r2,r3    @ x = a * b -> U(20,44)
```

Division

Given a dividend, N, with format U(i_1,f_1) and a divisor, D, with format U(i_2,f_2).

The value of the least significant bit of N is 2^{-f_i} and the value of the least significant bit of D is 2^{-f_2}.

In order to perform the division using integer operations, it is necessary to multiply N by 2^{f_i} and multiply D by 2^{f_2} so that both numbers are integers. Therefore the division operation can be written as:

$$Q = \frac{N \times 2^{f_1}}{D \times 2^{f_2}} = \frac{N}{D} \times 2^{f_1 - f_2}.$$

Example: Given two U(5,3) numbers:

$$Q = \frac{N \times 2^{3}}{D \times 2^{3}} = \frac{N}{D} \times 2^{0}.$$

If the programmer wants to have f_r fractional bits in the result, then the dividend must be shifted left an additional f_r bits before the division.

Division Example

For example, suppose the programmer wants to divide 01001.011 stored as a U(28,3) by 00011.110 which is also stored as a U(28,3), and wishes to have six fractional bits in the result. The programmer would first shift 01001.011 to the left by six bits, then perform the division and compute the position of the radix in the result as shown:

$$01001.011 \div 00011.110 = (0000001001011000000 \div 00011110) \times 2^{-6-3+3}$$

$$\begin{array}{r} 10100000 \\ 11110\overline{)\,1001011000000} \\ \underline{111100000000} \\ 1111000000 \\ \underline{1111000000} \\ 0 \end{array} \times 2^{-6} = 10.100000$$

Division Result Formats

Consider:
The largest possible value of the dividend is $N_{max} = 2^{i_1} - 2^{-f_1}$, and the smallest positive value for the divisor is $D_{min} = 2^{-f_2}$. Therefore, the maximum quotient is given by:

$$Q_{max} = \frac{2^{i_1} - 2^{-f_1}}{2^{-f_2}} = 2^{i_1+f_2} - 2^{f_1-f_2}.$$

Taking the limit of the previous equation,

$$\lim_{f_1-f_2 \to -\infty} Q_{max} = 2^{i_1+f_2},$$

provides the following bound on how many bits are required in the integer part of the quotient:

$$Q_{max} < 2^{i_1+f_2}.$$

Therefore, in the worst case, the quotient will require $i_1 + f_2$ integer bits.

Division Result Formats

Unsigned Division

The result of dividing an unsigned fixed point number $\mathrm{U}(i_1,f_1)$ by an unsigned number $\mathrm{U}(i_2,f_2)$ is a $\mathrm{U}(i_1+f_2,f_1-f_2)$ number.

Mixed Division

The result of dividing two fixed point numbers where one of them is signed and the other is unsigned is an is an $\mathrm{S}(i_1+f_2,f_1-f_2)$ number.

Signed Division

The result of dividing two signed fixed point numbers is an $\mathrm{S}(i_1+f_2+1,f_1-f_2)$ number.

Division Without Losing Precision

The smallest positive value for the dividend is $N_{min} = 2^{-f_1}$, and the largest possible value of the divisor is $D_{max} = 2^{i_2} - 2^{-f_2}$.
In the worst case, the least significant bit of the quotient will be $2^{-(i_2+f_1)}$.

Shifting the dividend left by $i_2 + f_2$ bits will convert it into a U($i_1, i_2 + f_1 + f_2$).

When it is divided by a U(i_2, f_2), the result is a U($i_1 + f_2, i_2 + f_1$). This is the minimum size which is guaranteed to preserve all bits of precision.

The general method for performing fixed point division while maintaining maximum precision is as follows:

1. shift the dividend left by $i_2 + f_2$ then
2. perform the division, and
3. use the rules from the previous slide to determine the result format.

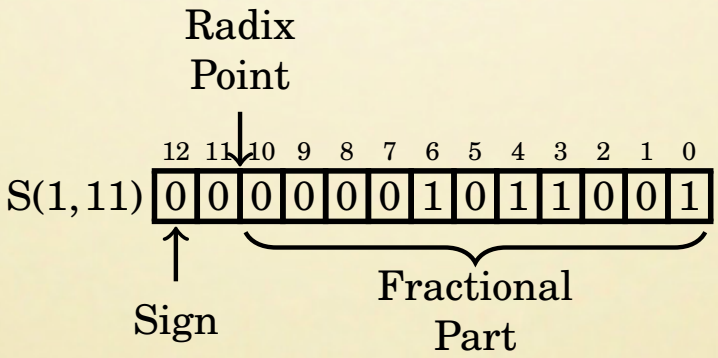
Division By a Constant Revisited

Calculate $x \div 23$ using only 8-bit signed integer multiplication.

The reciprocal of 23 is

$$R = \frac{1}{23} = 0.0000101100100001011\ldots_2 .$$

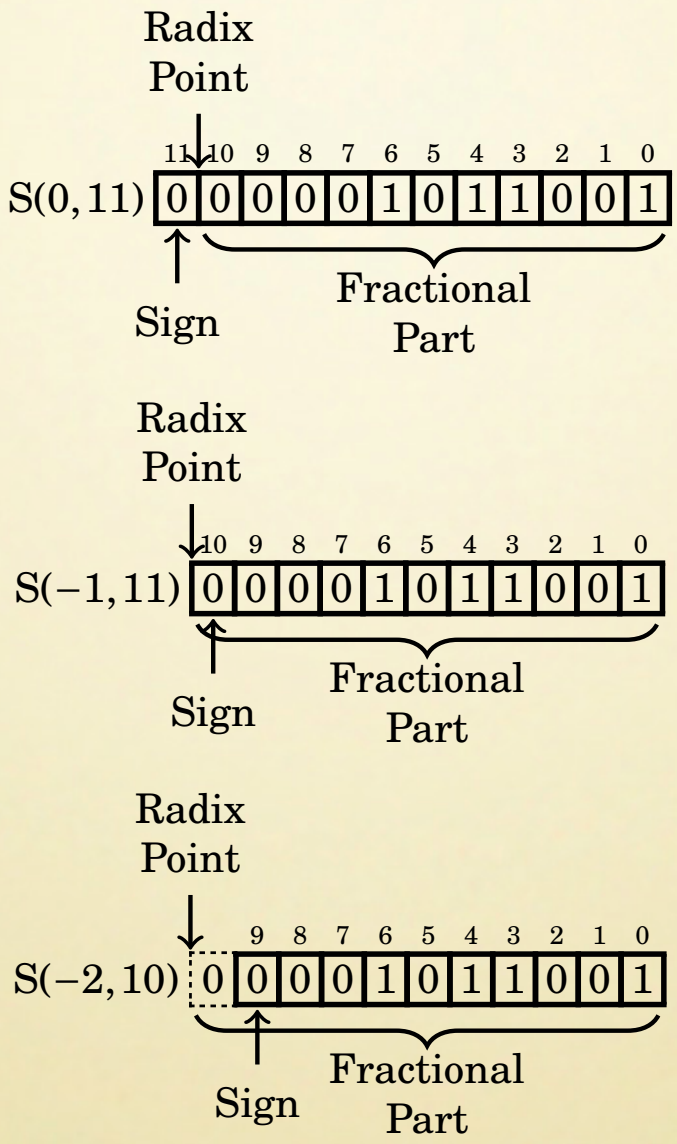
If we store R as an S(1,11), it would look like this:



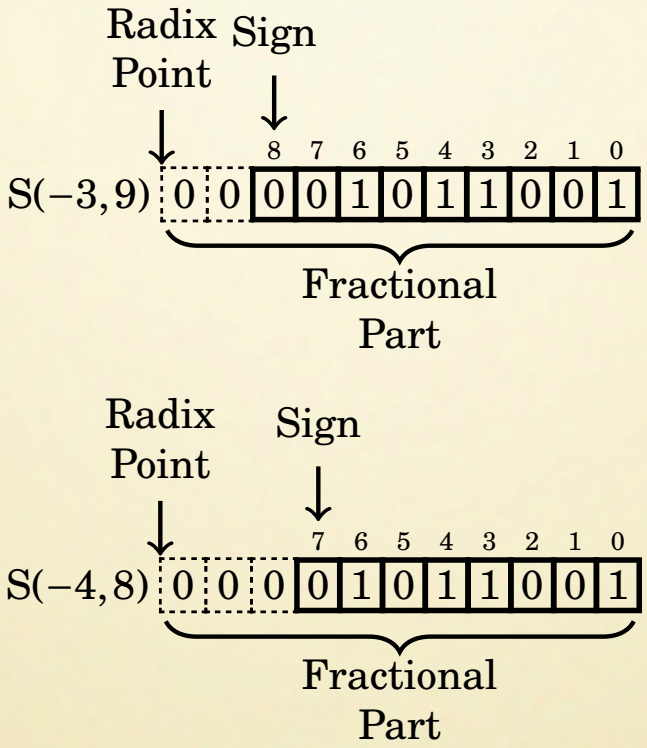
Note that in this format, the reciprocal of 23 has five leading zeros.

We can store more of the *significant* bits of R by shifting it left to remove some of the leading zeros.

Division By a Constant Revisited



Division By a Constant Revisited



We cannot shift any more without changing the sign bit.

Division By a Constant Revisited

An S(7,0) number x multiplied by an S(−4,8) number R will yield an S(4,8) number y. The value y will be $2^3 \times \frac{x}{23}$ because we have three "hidden" bits to the right of the radix point.

To calculate $y = 101_{10} \div 23_{10}$, we can multiply and perform a shift as follows:

$$
\begin{array}{r}
.\,01100101 \\
\times\ 01011010 \\
\hline
0.\,11001010 \\
011.\,00101 \\
0110.\,0101 \\
011001.\,11 \\
\hline
00100100.\,00000010
\end{array}
$$

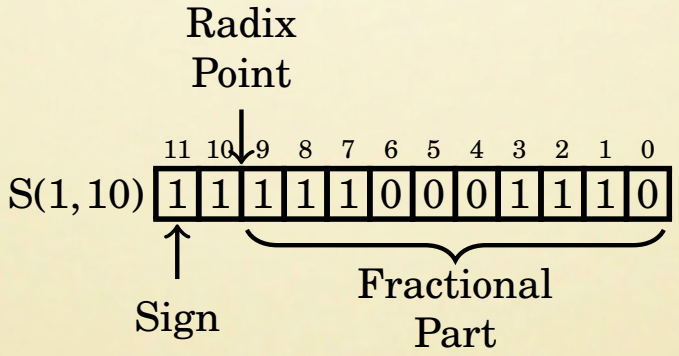
The integer portion, 100100_2, shifted right three bits, is $100_2 = 4_{10}$.

Division by a Negative Constant

Calculate $x \div -9$ using 8-bit signed integer multiplication.

$$
\begin{aligned}
-\frac{1}{9} &= -0.000111000111000111000111\ldots \\
&= 111\ldots11111.111000111000111000111000\ldots
\end{aligned}
$$

We can represent $\frac{1}{-9} \times 2^2$ as the following S(7,8) fixed point number:

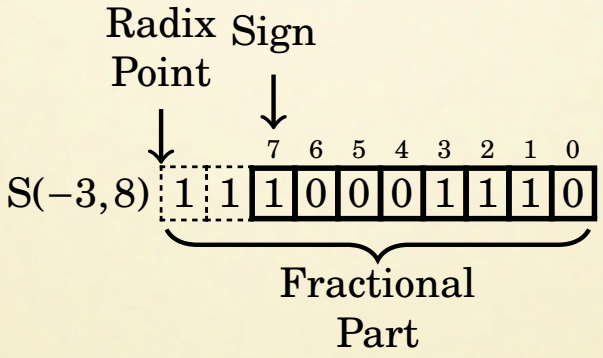


Note that the upper 5 bits are all one.

Division by a Negative Constant – Part 2

Since the upper 4 bits are all copies of the sign, we don't need to keep them.

$\frac{1}{-9}$ in 8 bits, can be represented as an S(−3,8) number R:



Given an S(7,0) number X, $R \times X$ will yield an S(7,8) number, such that

$$Y \times 2^2 = R \times X = \left(\frac{1}{-9} \times 2^2 \right) \times X$$

$$Y = R \times X \times 2^{-2}$$

Division by a Negative Constant - Example

To calculate $Y = 101_{10} \div -9_{10}$, use a signed multiply:

$$
\begin{array}{r}
. \; 1\,0\,0\,0\,1\,1\,1\,0 \\
\times \quad 0\,1\,0\,1\,1\,0\,0\,1 \\
\hline
1\,1\,1\,1\,1\,1\,1\,1 \,.\; 1\,0\,0\,0\,1\,1\,1\,0 \\
1\,1\,1\,1\,1\,1\,0\,0 \,.\; 0\,1\,1\,1\,0 \\
1\,1\,1\,1\,1\,0\,0\,0 \,.\; 1\,1\,1\,0 \\
1\,1\,1\,0\,0\,0\,1\,1 \,.\; 1\,0 \\
\hline
1\,1\,0\,1\,0\,1\,0\,0 \,.\; 0\,1\,0\,1\,1\,1\,1\,0
\end{array}
$$

We can immediately throw away the fractional part of the result, keeping only the upper 8 bits.

11010100_2 shifted right 2 bits is : 11110101_2
$-(00001010 + 1)_2 = -(8 + 2 + 1)_{10} = -11_{10}$.

If the modulus is required, it can be calculated as: $101 - (-11 \times -9) = 2$

Implementing Sine and Cosine

- $\cos x = \sin \frac{\pi}{2-x}$
- $\sin x$ is cyclical, so $\ldots \sin -2\pi = \sin 0 = \sin 2\pi \ldots$. This means that we can limit the domain of our function to the range $[-\pi, \pi]$
- $\sin x$ is symmetric, so that $\sin -x = -\sin x$. This means that we can further restrict the domain to $[0, \pi]$.
- After we restrict the domain to $[0, \pi]$, we notice another symmetry, $\sin x = \sin(\pi - x)$, $\frac{\pi}{2} \leq x \leq \pi$ and we can further restrict the domain to $[0, \frac{\pi}{2}]$.
- the range of both functions, $\sin x$ and $\cos x$, is in the range $[-1, 1]$.

Write a single shared function, `sinq`, to be used by both sine and cosine.

- `sinq` will accept x as an S(1,30), and
- `sinq` will return an S(1,30)

Taylor Series

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

- Only need a few terms to get 30 fractional bits of precision.
- All of the factorial terms are constants.
- Combine the -1 term with the factorials.

$$\sin x = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)!} x^{2n+1}$$

$$= \frac{1}{1!}x + \frac{-1}{3!}x^3 + \frac{1}{5!}x^5 + \frac{-1}{7!}x^7 + \ldots$$

Pre-compute the constants and store them in a table.

$$= c_0 x + c_1 x^3 + c_2 x^5 + c_3 x^7 + \ldots$$

Fixed Point Formats

Each successive power-of-x term must be reduced to 32-bits, or the format would become unmanageable:

Term	Format	32-bit
x	S(1,30)	S(1,30)
x^3	S(3,90)	S(3,28)
x^5	S(5,150)	S(5,26)
x^7	S(7,210)	S(7,24)
x^9	S(9,270)	S(9,22)
x^{11}	S(11,330)	S(11,20)
x^{13}	S(13,390)	S(13,18)
x^{15}	S(15,450)	S(15,16)
x^{17}	S(17,510)	S(17,14)

- Compute x^2 at beginning of function.
- For each successive term:
  - Multiply the previous term by x^2 (64-bit result)
  - Discard the lower 32 bits.

Constant Terms

$$-\frac{1}{3!} = -\frac{1}{6}$$

Convert to binary:

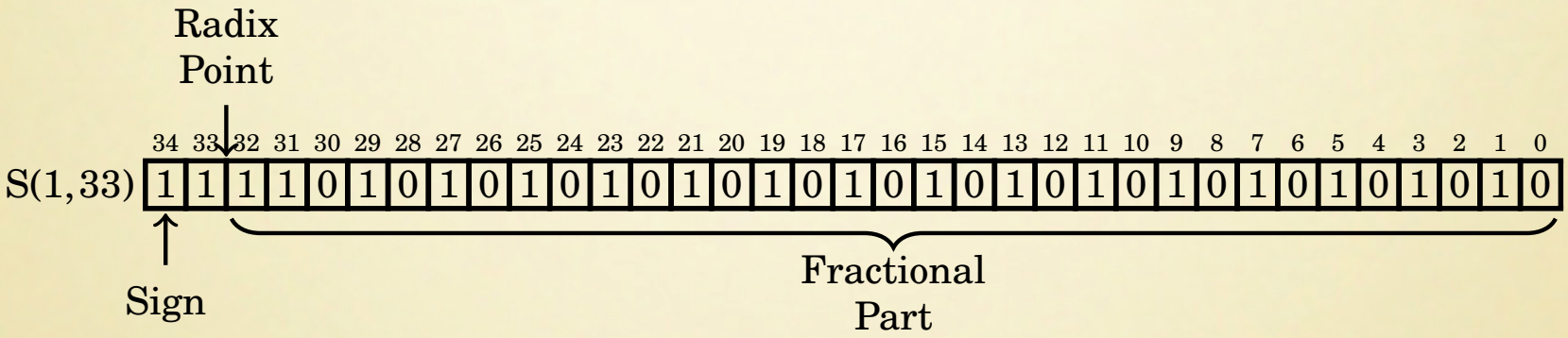
Multiplication	Result: Integer	Result: Fraction
$\frac{1}{6} \times 2 = \frac{2}{6}$	0	$\frac{2}{6}$
$\frac{2}{6} \times 2 = \frac{4}{6}$	0	$\frac{4}{6}$
$\frac{4}{6} \times 2 = \frac{8}{6}$	1	$\frac{2}{6}$
$\frac{2}{6} \times 2 = \frac{4}{6}$	0	$\frac{4}{6}$
$\frac{4}{6} \times 2 = \frac{8}{6}$	1	$\frac{2}{6}$

$$\begin{aligned} -\frac{1}{3!} &= -0.0\overline{01}_2 \\ &= \ldots 111.1\overline{10}_2 \end{aligned}$$

Constant Terms (Part 2)

$$-\frac{1}{3!} = -0.0\overline{01}_2$$

$$= \ldots 111.1\overline{10}_2$$



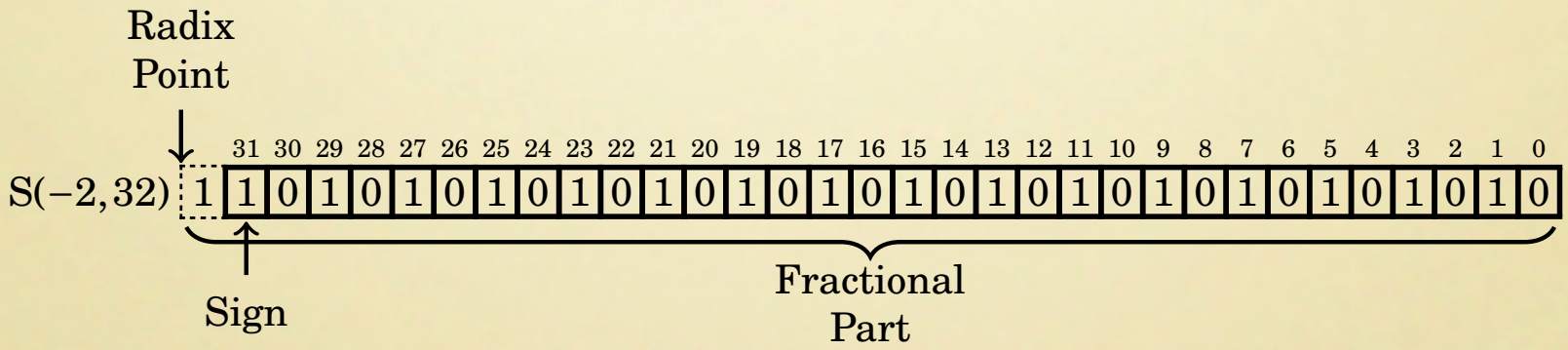


Table of Constant Terms

Term	Reciprocal Format	Reciprocal Value (Hex)
$-\frac{1}{3!}$	S(−2,32)	AAAAAAAA
$\frac{1}{5!}$	S(−6,32)	44444445
$-\frac{1}{7!}$	S(−12,32)	97F97F97
$\frac{1}{9!}$	S(−18,32)	5C778E96
$-\frac{1}{11!}$	S(−25,32)	9466EA60
$\frac{1}{13!}$	S(−32,32)	5849184F
$-\frac{1}{15!}$	S(−40,32)	94603063
$\frac{1}{17!}$	S(−48,32)	654B1DC1

Computing Each Term

	Numerator		Reciprocal			Result
Term	Value	Format	Value	Format	Hex	Format
1	x	S(1,30)	Extend to 64 bits and shift right			S(2,61)
2	x^3	S(3,28)	$-\frac{1}{3!}$	S(−2,32)	AAAAAAAA	S(2,61)
3	x^5	S(5,26)	$\frac{1}{5!}$	S(−6,32)	44444444	S(0,63)
4	x^7	S(7,24)	$-\frac{1}{7!}$	S(−12,32)	97F97F97	S(−4,64)
5	x^9	S(9,22)	$\frac{1}{9!}$	S(−18,32)	5C778E96	S(−8,64)
6	x^{11}	S(11,20)	$-\frac{1}{11!}$	S(−25,32)	9466EA60	S(−13,64)
7	x^{13}	S(13,18)	$\frac{1}{13!}$	S(−32,32)	5849184F	S(−18,64)
8	x^{15}	S(15,16)	$-\frac{1}{15!}$	S(−40,32)	94603063	S(−24,64)
9	x^{17}	S(17,14)	$\frac{1}{17!}$	S(−48,32)	654B1DC1	S(−30,64)

Correcting Shifts for Each Term

Term Number	Original Format	Shift Amount	Resulting Format
1	S(1,30)	1	S(2,61)
2	S(2,61)	0	S(2,61)
3	S(0,63)	2	S(2,61)
4	S(−4,64)	6	S(2,61)
5	S(−8,64)	10	S(2,61)
6	S(−13,64)	15	S(2,61)
7	S(−18,64)	20	S(2,61)
8	S(−24,64)	26	S(2,61)
9	S(−30,64)	32	S(2,61)

C Implementation

```
#define pi    0x3243F6A8 /* pi as an S(3,28)   */
#define pi_2  0x1921FB54 /* pi/2 as an S(3,28) */
#define pi_x2 0x6487ED51 /* 2*pi as an S(3,28) */

#define TABSIZE 6  /* use first 7 terms of Taylor series */
struct tabentry{int coeff; int shift;};
const static struct tabentry sintab[TABSIZE]={
  {0xAAAAAAAA,  0}, {0x44444445,  2}, {0x97F97F97,  6},
  {0x5C778E96, 10}, {0x9466EA60, 15}, {0x5849184F, 20}};

/* sinq does all of the real work. x is S(1,30) */
static int sinq(int x)
{ long long sum = (long long)x << 31; /* initialize running total S(2,61) */
  long long xsq = ((long long)x*(long long)x)>>31;/*calculate x^2 S(2,28) */
  long long curpower = x;               /* curpower holds x^(2n+1) S(1,30) */
  long long tmp;          /* tmp holds each term as it is computed S(2,61) */
  int i=0;
  do {
    curpower = ((curpower * xsq) >> 31); /* calculate x^(2n+1) S(1,30) */
    tmp = curpower * sintab[i].coeff;    /* calculate term */
    if(tmp < 0)                          /* adjust for round-off */
      tmp++;                             /* if term is negative  */
    tmp >>= sintab[i].shift;             /* shift to align radix S(2,61) */
    sum += tmp;                          /* add to running total S(2,61) */
  } while(++i<TABSIZE);                  /* repeat 5 more times  */
  return (sum >> 33);                    /* shift and truncate to S(3,28) */
}
```

C Implementation

```
1
2  /* fixed_sin_C(S(3,28) x) calculates sine of x*/
3  int fixed_sin_C(int x)
4  {
5    while(x<0) x += pi_x2;
6    while(x>pi_x2) x -= pi_x2;
7    if(x<=pi_2) return sinq(x<<2);
8    if(x<=pi) return sinq((pi-x)<<2);
9    if(x<=(pi+pi_2)) return -sinq((x-pi)<<2);
10   return  -sinq((pi_x2 -x)<<2);
11 }
12
13 /* fixed_cos_C(S(3,28) x) calculates cosine of x */
14 int fixed_cos_C(int x)
15 {
16   while(x<0) x += pi_x2;
17   x = pi_2 - x;
18   return fixed_sin(x);
19 }
```

Assembly Implementation

```
1         @@*****************************************************
2         @@ Name: sincos.S
3         @@ Author: Larry Pyeatt
4         @@ Date: 2/22/2014
5         @@*****************************************************
6         @@   This is a version of the sin/cos functions that uses
7         @@ symmetry to enhance precision.  The actual sin and cos
8         @@ routines convert the input to lie in the range 0 to pi/2,
9         @@ then pass it to the worker routine that computes the
10        @@ result.  The result is then converted back to correspond
11        @@ with the original input.
12        @@   We calculate sin(x) using the first seven terms of the
13        @@ Taylor Series: sin(x) = x - x^3/3! + x^5/5! - x^7/7! +
14        @@ x^9/9! - ...  and we calculate cos(x) using the
15        @@ relationship: cos(x) = sin(pi/2-x)
16        @@   We start by defining a helper function, which we call sinq.
17        @@ The sinq function calculates sin(x) for 0<=x<=pi/2.  The
18        @@ input, x, must be an S(1,30) number. The factors of x that
19        @@ sinq will use are: x, x^3, x^5, x^7, x^9, x^11, and x^13.
20        @@   Dividing by (2n+1)! is changed to a multiply by a
21        @@ coefficient as we compute each term, we will add it to the
22        @@ sum, stored as an S(2,61). Therefore, we want the product
23        @@ of each power of x and its coefficient to be converted to
24        @@ an S(2,61) for the add.  It turns out that this just
25        @@ requires a small shift.
26        @@   We build a table to decide how much to shift each product
27        @@ before adding it to the total.  x^2 will be stored as an
```

Assembly Implementation

```
1         @@ S(2,29), and x is given as an S(1,30).  After multiplying
2         @@ x by x^2, we will shift left one bit, so the procedure is:
3         @@ x   will be an  S(1,30) - multiply by x^2 and shift left
4         @@ x^3 will be an  S(3,28) - multiply by x^2 and shift left
5         @@ x^5 will be an  S(5,26) - multiply by x^2 and shift left
6         @@ x^7 will be an  S(7,24) - multiply by x^2 and shift left
7         @@ x^9 will be an  S(9,22) - multiply by x^2 and shift left
8         @@ x^11 will be an S(11,20)- multiply by x^2 and shift left
9         @@ x^13 will be an S(13,18)- multiply by x^2 and shift left
10        @@
11        @@   The following table shows the constant coefficients
12        @@ needed for calculating each term.
13        @@ -1/3! =  AAAAAAAA as an S(-2,32)
14        @@  1/5! =  44444445 as an S(-6,32)
15        @@ -1/7! =  97F97F97 as an S(-12,32)
16        @@  1/9! =  5C778E96 as an S(-18,32)
17        @@ -1/11! = 9466EA60 as an S(-25,32)
18        @@  1/13! = 5849184F as an S(-32,32)
19        @@
20        @@   Combining the two tables of power and coefficient formats,
21        @@ we can now determine how much shift we need after each
22        @@ step in order to do all sums in S(2,61) format:
23        @@ power powerfmt    coef  coeffmt     resultfmt right shift
24        @@  x    S(1,30)   *  1 (skip the multiply)        1 -> S(2,61)
25        @@  x^3  S(3,28)   * -1/3!  S(-2,32)  = S(2,61)    0 -> S(2,61)
26        @@  x^5  S(5,26)   *  1/5!  S(-6,32)  = S(0,63)    2 -> S(2,61)
27        @@  x^7  S(7,24)   * -1/7!  S(-12,32) = S(-4,64)   6 -> S(2,61)
```

Assembly Implementation

```
1          @@  x^9  S(9,22)  *  1/9!  S(-18,32) = S(-8,64)  10-> S(2,61)
2          @@  x^11 S(11,20) * -1/11! S(-25,32) = S(-13,64) 15-> S(2,61)
3          @@  x^13 S(13,18) *  1/13! S(-32,32) = S(-18,64) 20-> S(2,61)
4
5          .data
6          .align  2
7          @@ We will define a few constants that may be useful
8          .global pi
9  pi:     .word   0x3243F6A8      @ pi as an S(3,28)
10         .global pi_2
11 pi_2:   .word   0x1921FB54      @ pi/2 as an S(3,28)
12         .global pi_x2
13 pi_x2:  .word   0x6487ED51      @ 2*pi as an S(3,28)
14
15 sintab: @@ This is the table of coefficients and shifts
16         .word 0xAAAAAAAA,  0    @ -1/3!  as an S(-2,32)
17         .word 0x44444445,  2    @  1/5!  as an S(-6,32)
18         .word 0x97F97F97,  6    @ -1/7!  as an S(-12,32)
19         .word 0x5C778E96,  10   @  1/9!  as an S(-18,32)
20         .word 0x9466EA60,  15   @ -1/11! as an S(-25,32)
21         .word 0x5849184F,  20   @  1/13! as an S(-32,32)
22         .equ tablen,(.-sintab)  @ set tablen to the size of table.
23         @@ The '.' refers to the current address counter value.
24         @@ Subtracting the address of sintab from the current
25         @@ address gives the size of the table.
```

Assembly Implementation

```
1          .text
2          @@-------------------------------------------------------------
3          @@ sinq(x)
4          @@ input: x -> S(1,30) s.t. 0 <= x <= pi/2
5          @@ returns sin(x) -> S(1,30)
6  sinq:   stmfd   sp!,{r4-r11,lr}
7          smull   r2,r4,r0,r0     @ r4 will hold x^2.
8          @@ The first term in the Taylor series is simply x, so
9          @@ convert x to an S(2,61) by doing an asr in 64 bits,
10         @@ and use it to initialize the sum.
11         mov     r10, r0,lsl #31 @ low 32 bits of sum
12         mov     r11, r0,asr #1  @ high 32 bits of sum
13         @@ r11:r10 now contains the sum (currently x) as an S(2,61)
14         @@ We are going to convert x^2 to an S(2,28), and round it
15         adds    r2,r2,#0x40000000 @ Round x^2 up by adding 1 to
16                                   @ the first bit that will be lost.
17         adccs   r4,r4,#0          @ Propagate the carry.
18         lsl     r4,r4,#1          @ Make room for one bit in LSB
19         orr     r4,r4,r2,lsr#31 @ Copy least significant bit of x^2
20         @@ r4 now contains x^2 as an S(2,28)
21         mov     r5,r0             @ r5 will keep x^(2n-1).
22         @@ r5 now contains x as an S(1,30)
23         @@ The multiply will take time, and on some processors,
24         @@ there is an extra clock cycle penalty if the next
25         @@ instruction requires the result, so do the multiply now.
26         smull   r0,r5,r4,r5     @ r5:r0 <- x^3 as an S(4,59)
27         ldr     r6, =sintab     @ get pointer to beginning of table
```

Assembly Implementation

```
1          add     r7, r6, #tablen @ get pointer to end of table
2          @@ We know that we will always execute the loop 6 times,
3          @@ so we use a post-test loop.
4  sloop:  ldmia   r6!,{r8,r9}     @ Load two values from the table
5          @@ r8 now has (-1^n)/(2n+1)!
6          @@ r9 contains the correcting shift
7          @@ the previous smull r0,r5,r4,r5 should be complete soon
8          lsl     r5,r5,#1        @ Shift and copy the m.s. bit of the
9          orr     r5,r5,r0,lsr#31 @ LSW to l.s. bit of MSW -> S(3,60)
10         @@ r5 now contains x^(2n+1) as an S(3,60)
11         @@ Start next multiply now
12         smull   r0,r1,r5,r8     @ multiply by reciprocal that we
13                                 @ loaded earlier (5 cycles)
14         @@ Apply correcting right shift to make an S(2,61).
15         @@ Note: r9 was loaded from the table earlier.
16         rsb     r2,r9,#32       @ calculate inverse shift amount
17         lsr     r0,r0,r9        @ Make room in low word for bits
18         orr     r0,r0,r1,lsl r2 @ paste bits into low word
19         asr     r1,r1,r9        @ shift upper word right
20         @@ accumulate result in r10:r11
21         adds    r10,r10,r0
22         adc     r11,r11,r1
23         @@ check to see if there is another term to compute
24         cmp     r6, r7
25         @@ Start next multiply now
26         smulllt r0,r5,r4,r5     @ r5:r0 <- x^(2n+1) as an S(4,59)
27         @@  The multiply will take three cycles, so start it now
```

Assembly Implementation

```
1         blt     sloop           @ Repeat for every table entry
2         @@ shift result left 1 bit and move MSW to r0
3         lsl     r11,r11,#1
4         orr     r0,r11,r10,lsr #31
5         @@ return the result
6         ldmfd   sp!,{r4-r11,pc}
7
8
9
10        @@-----------------------------------------------------------
11        @@ cos(x)  NOTE: The cos(x) function does not return.
12        @@               It is an alternate entry point to sin(x).
13        @@ input: x -> S(3,28)
14        @@ returns cos(x) -> S(3,28)
15        .global fixed_cos
16fixed_cos:
17        ldr     r1,=pi_x2       @ load pointer to 2*pi
18        ldr     r1,[r1]         @ load 2*pi
19        cmp     r0,#0           @ Add 2*pi to x if needed, to make
20        addle   r0,r0,r1        @ sure x does not become too small
21cosgood:ldr     r1,=pi_2        @ load pointer to pi/2
22        ldr     r1,[r1]         @ load pi/2
23        sub     r0,r1,r0        @ cos(x) = sin(pi/2-x)
24        @@ now we just fall through into the sin function
```

Assembly Implementation

```
1         @@-------------------------------------------------------------
2         @@ sin(x)
3         @@ input: x -> S(3,28)
4         @@ returns sin(x) -> S(3,28)
5         .global fixed_sin
6 fixed_sin:
7         stmfd   sp!,{lr}
8         ldr     r1,=pi_2          @ r1 has pointer to pi/2
9         ldr     r2,=pi            @ r2 has pointer to pi
10        ldr     r3,=pi_x2         @ r3 has pointer to pi*2
11        ldr     r1,[r1]           @ r1 has pi/2
12        ldr     r2,[r2]           @ r2 has pi
13        ldr     r3,[r3]           @ r3 has pi*2
14        @@ step 1: make sure x>=0.0 and x<=2pi
15 negl:  cmp     r0,#0             @ while(x < 0)
16        addlt   r0,r0,r3          @   x = x + 2 * pi
17        blt     negl              @ end while
18 nonneg: cmp    r0,r3             @ while(x > pi/2)
19        subgt   r0,r0,r3          @   x = x - 2 * pi
20        bgt     nonneg            @ end while
21        @@ step 2: find the quadrant and call sinq appropriately
22 inrange:cmp    r0,r1
23        bgt     chkq2
24        @@ it is in the first quadrant... just shift and call sinq
25        lsl     r0,r0,#2
26        bl      sinq
27        b       sin_done
```

Assembly Implementation

```
1  chkq2:  cmp     r0,r2
2          bgt     chkq3
3          @@ it is in the second quadrant... mirror, shift, and call
4          @@ sinq
5          sub     r0,r2,r0
6          lsl     r0,r0,#2
7          bl      sinq
8          b       sin_done
9  chkq3:  add     r1,r1,r2        @ we will not need pi/2 again
10         cmp     r0,r1           @ so use r1 to calculate 3pi/2
11         bgt     chkq4
12         @@ it is in the third quadrant... rotate, shift, call sinq,
13         @@ then complement the result
14         sub     r0,r0,r2
15         lsl     r0,r0,#2
16         bl      sinq
17         rsb     r0,r0,#0
18         b       sin_done
19         @@ it is in the fourth quadrant... rotate, mirror, shift,
20         @@ call sinq, then complement the result
21 chkq4:  sub     r0,r0,r2
22         sub     r0,r2,r0
23         lsl     r0,r0,#2
24         bl      sinq
25         rsb     r0,r0,#0
```

Assembly Implementation

```
1 sin_done:
2         @@ shift result right 2 bits
3         asr     r0,r0,#2
4         @@ return the result
5         ldmfd   sp!,{pc}
6         @@----------------------------------------------------------
```

Performance

Optimization	Implementation	CPU seconds
None	32-bit Fixed Point Assembly	3.85
	32-bit Fixed Point C	18.99
	Single Precision Software Float C	56.69
	Double Precision Software Float C	55.95
	Single Precision VFP C	11.60
	Double Precision VFP C	11.48
Full	32-bit Fixed Point Assembly	3.22
	32-bit Fixed Point C	5.02
	Single Precision Software Float C	20.53
	Double Precision Software Float C	54.51
	Single Precision VFP C	3.70
	Double Precision VFP C	11.08

Patriot Missile Failure

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile.
- The Scud struck an American Army barracks, killing 28 soldiers and injuring around 98 other people.
- The cause was an inaccurate calculation of the time since boot. arithmetic errors
- The time in tenths of second as measured by the system's internal clock was multiplied by $\frac{1}{10}$ to produce the time in seconds.
- This calculation was performed using a U(0,24) fixed point representation.
- The small error, when multiplied by the large number giving the time in tenths of a second, led to a significant error.

Patriot Missile Failure - Continued

- The Patriot battery had been up for around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds.
- The binary expansion of $\frac{1}{10}$ is $0.0\overline{0011}$
- The 24 bit register in the Patriot stored 0.00011001100110011001100 introducing an error of $0.000000000000000000000\overline{1100}_2$ or about 0.000000095_{10}
- Multiplying by the number of tenths of a second in 100 hours gives $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$.
- A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in 0.34 seconds.
- This was far enough that the incoming Scud was outside the “range gate” that the Patriot tracked.

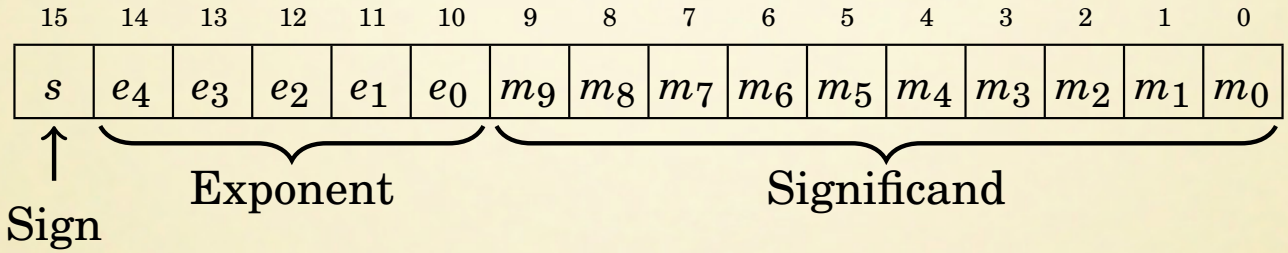
PEOPLE DIED

Floating Point – Half Precision

Sometimes we need more range than we can get from fixed precision.

IEEE has set standards for various floating point formats.

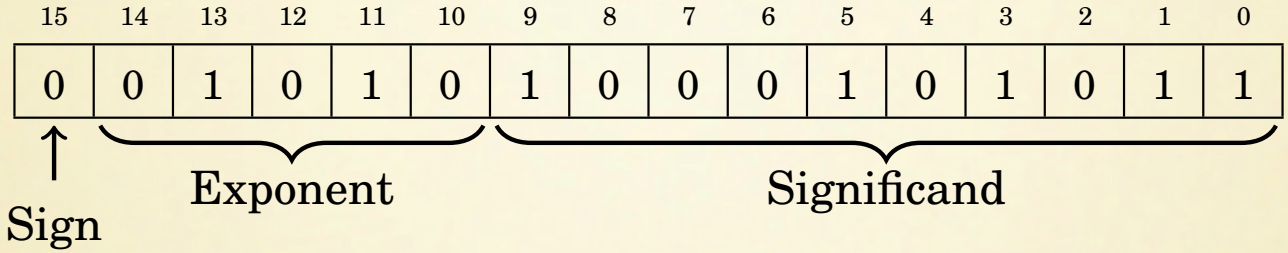
IEEE 754 Half-Precision



- The Significand (a.k.a. “Mantissa” or “Fractional Part”) is in sign-magnitude coding, with bit 15 being the sign bit.
- There are 10 bits of significand, but there are 11 bits of significand precision. There is a “hidden” bit, m_{10}, between m_9 and e_0.
- The exponent is an excess-15 number. i.e. The number stored is 15 greater than the actual exponent.

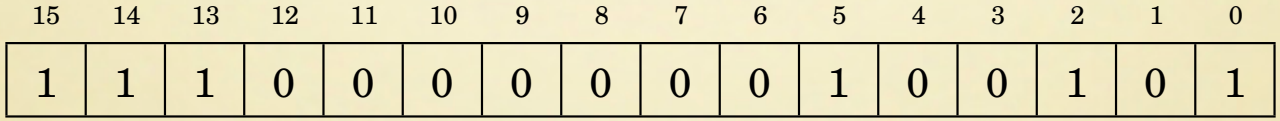
Floating Point – Half Precision Examples

IEEE 754 Half-Precision



$$+1.1000101011 \times 2^{01010-01111} = 1.1000101011 \times 2^{-5} = .000011000101011$$

$$\approx 0.04819_{10}$$



$$-1.0000100101 \times 2^{11000-01111} = -1.0000100101 \times 2^{9} = -1000010010.1$$

$$= -530.5_{10}$$

Floating Point – Half Precision Special Values

The exponents 00000_2 and 11111_2 have special meaning.

Exponent	Significand = 0	Significand ≠ 0	Equation
00000	±0	subnormal	$-1^{sign} \times 2^{-14} \times 0.significand$
00001, . . . , 11110	normalized value		$-1^{sign} \times 2^{exponent-15} \times 1.significand$
11111	±∞	NaN	

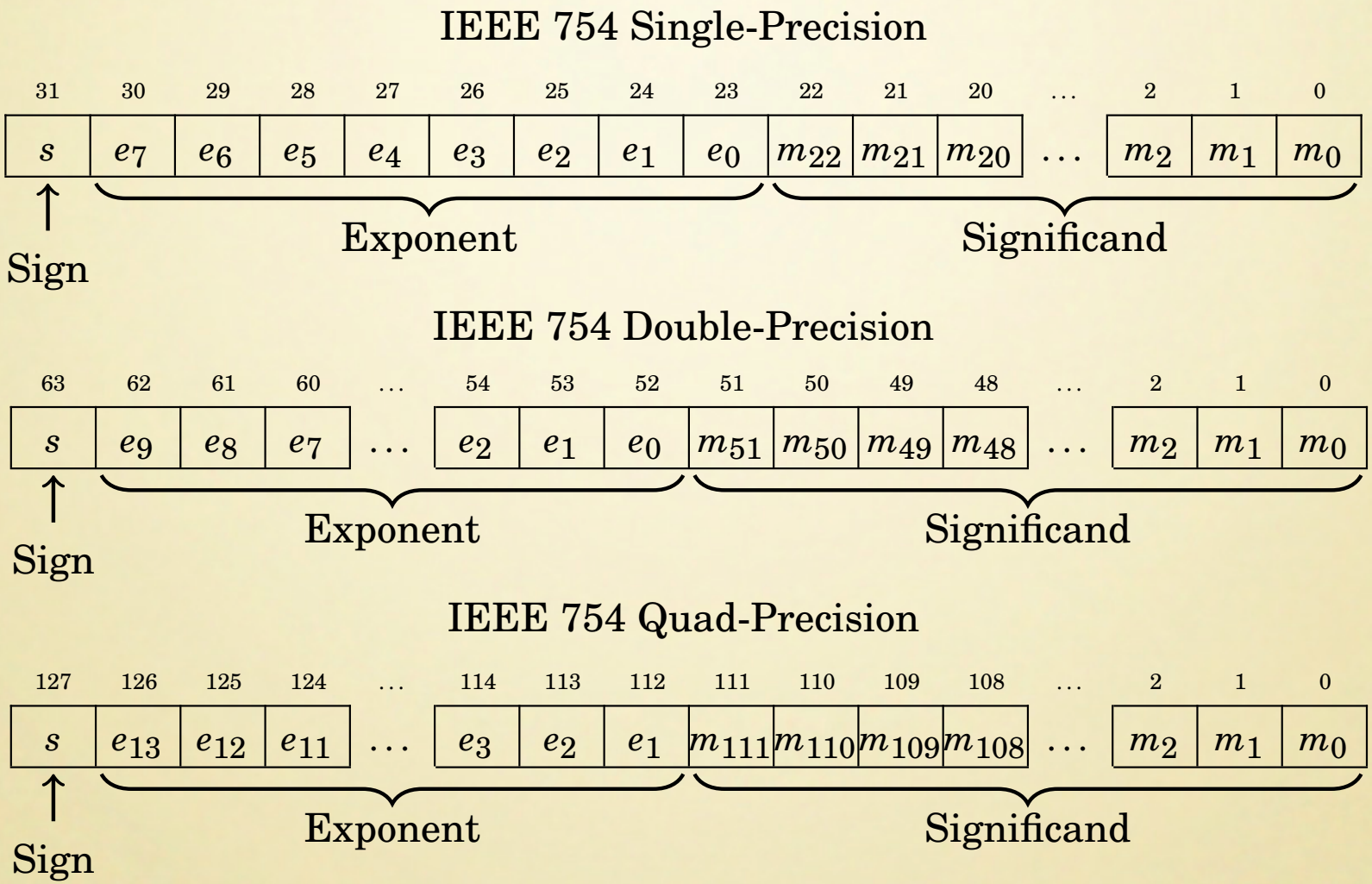
Subnormal means that the value is too close to zero to be completely normalized.

The minimum strictly positive (subnormal) value is $2^{-24} \approx 5.96 \times 10^{-8}$.

The minimum positive normal value is $2^{-14} \approx 6.10 \times 10^{-5}$.

The maximum *exactly* representable value is $(2 - 2^{-10}) \times 2^{15} = 65504$.

Floating Point – More Precision



Many processors do not have hardware support for floating point.

Floating Point Addition and Subtraction

The basic algorithm is the same for floats of all sizes.

1. Extract the exponents E_a and E_b.
2. Extract the significands M_a and M_b. and convert them into 2's complement numbers, using the signs S_a and S_b.
3. Shift the significand with the smaller exponent right by $|E_a - E_b|$.
4. Perform addition (or subtraction) on the significands to get the significand of the result, M_r. Remember that the result may require one more significant bit to avoid overflow.
5. If M_r is negative, then take the 2's complement and set S_r to 1. Otherwise set S_r to 0.
6. Shift M_r until the leftmost 1 is in the "hidden" bit position, and add the shift amount to the smaller of the two exponents to form the new exponent E_r.
7. Combine the sign S_r, the exponent E_r, and significand M_r to form the result.

Floating Point Multiplication and Division

The basic algorithm is the same for floats of all sizes.

1. Calculate the sign of the result S_r.
2. Extract the exponents E_a and E_b.
3. Extract the significands M_a and M_b.
4. Multiply (or divide) the significands to form M_r.
5. Add (or subtract) the exponents (in excess-N) to get E_r.
6. Shift M_r until the leftmost 1 is in the "hidden" bit position, and add the shift amount to E_r.
7. Combine the sign S, the exponent E_r, and significand M_r to form the result.

Summary

- The two common ways to represent non-integral numbers in a computer are
  - fixed point and
  - floating point.
- Fixed point is a way to perform calculations on non-integral numbers using only integer operations.
- Floating point allows the radix point to be tracked automatically, but requires much more complex software and/or hardware.
- Fixed point will usually provide better performance than floating point, but requires more programming skill.
- Fractions which terminate in base two will also terminate in base ten, but the converse is not true.
- Programmers should avoid counting using fractions which do not terminate in base two, because it leads to the accumulation of round-off errors.