Real Numbers

range -- need to deal with larger range, not just

[not just \(-2^{(n-1)}\) to \(+2^{(n-1)-1}\)]

- large numbers
- small numbers (fractions)
von Neumann recommended using fixed point with the programmer doing the scaling. 

- fixed point = integer part <assumed binary point> fraction part
- use positional representation to convert a binary fraction to a decimal value

\[ d_{-1} d_{-2} d_{-3} \ldots = \text{sum over } i \left[ d_i \times 2^i \right] \]
use positional representation to convert a binary fraction to a decimal value

\[ .d_{-1} \, d_{-2} \, d_{-3} \, ... \, = \, \text{sum over } i \left[ \, d_i \, \times \, 2^i \, \right] \]

\[ .011 \, (\text{base } 2) = 0 \, \times \, 2^{(-1)} + 1 \, \times \, 2^{(-2)} + 1 \, \times \, 2^{(-3)} \]
\[ = 0 \times 1/2 \quad + \quad 1 \times 1/4 \quad + \quad 1 \times 1/8 \]
\[ = 0 \times 0.5 \quad + \quad 1 \times 0.25 \quad + \quad 1 \times 0.125 \]
\[ = 0 \quad + \quad 0.25 \quad + \quad 0.125 \]
\[ = 0.375 \]
Real Numbers

converting a decimal fraction to a binary fraction:

  repeatedly multiply decimal fraction by two, taking the integer part 0 or 1 as bits of the fraction from left to right, stop when fraction is zero

e.g., 0.375 (base 10)

\[
\begin{align*}
0.375 \times 2 &= 0.75 & \text{integer parts are } 0, 1, 1 \\
0.75 \times 2 &= 1.5 & \text{binary fraction is } .011 \\
0.5 \times 2 &= 1.0 & \text{stop}
\end{align*}
\]
Real Numbers

• note that a terminating decimal fraction may be non-terminating when

  e.g., \(0.1 \text{ base } 10 = 0.00011001100\ldots \text{ base } 2\)
Floating Point

- automatic scaling by hardware
- like scientific notation, automatic scaling by hardware requires an exponent field (i.e., the scale factor) and a mantissa (i.e., 1.fraction)
  - one sign bit
  - range is governed by the number of bits in the exponent
  - precision is governed by the number of bits in the mantissa
Floating Point

normal form - number represented as 1.fraction times an appropriate exponent

\[
\begin{array}{ccc}
\text{s} & \text{exp} & \text{fraction} \\
\end{array}
\]

\[
\text{value} = (-1)^s \times 1.\text{fraction} \times 2^{\text{exp}}
\]

exponent uses bias notation so that negative exponents have leading 0s

(historical reason is that integer compare operations could be used on this type of floating-point representation)
Floating Point

consider a 2-bit exponent using bias 2 (binary 10)

- zero exp means zero
  - 01 exp means (01 base 2 - bias 10 base 2) = $2^{-1} = 1/2$
  - 10 exp means (10 base 2 - bias 10 base 2) = $2^0 = 1$
  - 11 exp means (11 base 2 - bias 10 base 2) = $2^1 = 2$

and a 2-bit fraction w/ leading 1. hidden

for sign=0 (positive numbers):

- 0 00 00 = 0.0
- 0 01 00 = 1.00(base 2) x 2^(01 - 10 base 2) = 1.00 x $2^{-1} = 0.500$
- 0 01 01 = 1.01(base 2) x 2^(01 - 10 base 2) = 1.25 x $2^{-1} = 0.625$
- 0 01 10 = 1.10(base 2) x 2^(01 - 10 base 2) = 1.50 x $2^{-1} = 0.750$
- 0 01 11 = 1.11(base 2) x 2^(01 - 10 base 2) = 1.75 x $2^{-1} = 0.875
Floating Point

for sign=0 (positive numbers):

0 10 00 = 1.00(base 2) x 2^{(10-10 base 2)} = 1.00 x 2^0 = 1.00
0 10 01 = 1.01(base 2) x 2^{(10-10 base 2)} = 1.25 x 2^0 = 1.25
0 10 10 = 1.10(base 2) x 2^{(10-10 base 2)} = 1.50 x 2^0 = 1.50
0 10 11 = 1.11(base 2) x 2^{(10-10 base 2)} = 1.75 x 2^0 = 1.75

0 11 00 = 1.00(base 2) x 2^{(11 - 10 base 2)} = 1.00 x 2^1 = 2.0
0 11 01 = 1.01(base 2) x 2^{(11 - 10 base 2)} = 1.25 x 2^1 = 2.5
0 11 10 = 1.10(base 2) x 2^{(11 - 10 base 2)} = 1.50 x 2^1 = 3.0
0 11 11 = 1.11(base 2) x 2^{(11 - 10 base 2)} = 1.75 x 2^1 = 3.5
Floating Point

representing the preceding points on the number line:

\[
\begin{array}{ccc}
2^{-1} & 2^0 & 2^1 \\
\hline
0 & .5 & .625 \\
.75 & 1 & .875 \\
1.25 & 1.75 & 1.5 \\
1.75 & 2 & \\
2 & 2.5 & \\
2.5 & 3 & \\
3 & 3.5 & \\
3.5 & & \\
\end{array}
\]

notice the geometric increase in spacing between representable numbers (0.125, then 0.25, then 0.5)
Floating Point

precision - how many digits are used to represent a number

accuracy - how correct the number is; e.g., 4.56789 is a number with six-digit precision but rather inaccurate as an approximation to pi, while 3 is less precise but more accurate

rounding - choose nearest representable neighbor
while absolute error in representation increases as the numbers get larger, relative error remains the same, e.g., consider representing numbers one fourth of the distance between the "pickets" in the "picket fence" given on the previous slide

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-----</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.5</td>
<td>.75</td>
</tr>
<tr>
<td>1</td>
<td>1.25</td>
</tr>
<tr>
<td>1.75</td>
<td>2</td>
</tr>
<tr>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3.5</td>
</tr>
</tbody>
</table>

A = 1.0625 will be represented by 1.0, so absolute error = 0.0625, relative error = 0.0625/1.0625 = 0.0588

B = 2.125 will be represented by 2.0, so absolute error = 0.125, relative error = 0.125/2.125 = 0.0588
the maximum relative error in nonzero representations will occur for a number that is halfway between pickets, thus for a binary mantissa, the relative error will be bounded by

\[ 2^{- (\text{number of bits in fraction} + 1)} \]

for a number that gets truncated or rounded to zero, the maximum relative error can reach a factor of 1, e.g., if 0.1 is represented as 0

denormal numbers are used for gradual underflow rather than just truncating values less than \( 2^{(\text{minimum exponent})} \) to zero
in the previous example for 0.1, use 00 exp for smallest exp value \(2^{-1}\), but with no hidden bit; thus, assign 0001=0.125, 0010=0.250, and 0011=0.375

<table>
<thead>
<tr>
<th>0.5</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
<th>2.5</th>
<th>3</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.5</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>.375</td>
<td>.25</td>
</tr>
<tr>
<td>.125</td>
<td></td>
</tr>
</tbody>
</table>
Floating Point

a five-bit floating-point format with denormals yields 32 patterns

0 00 00 = zero 1 00 00 = -zero
0 00 01 = 0.125 1 00 01 = -0.125
0 00 10 = 0.25 1 00 10 = -0.25
0 00 11 = 0.375 1 00 11 = -0.375
0 01 00 = 0.5 0 1 01 00 = -0.5
01 01 = 0.625 1 01 01 = -0.625
0 01 10 = 0.75 1 01 10 = -0.75
0 01 11 = 0.875 1 01 11 = -0.875
0 10 00 = 1.0 1 10 00 = -1.0
0 10 01 = 1.25 1 10 01 = -1.25
0 10 10 = 1.5 1 10 10 = -1.5
0 10 11 = 1.75 1 10 11 = -1.75
0 11 00 = 2.0 1 11 00 = -2.0
0 11 01 = 2.5 1 11 01 = -2.5
0 11 10 = 3.0 1 11 10 = -3.0
0 11 11 = 3.5 1 11 11 = -3.5

relative error bound of $\frac{1}{2^{(\text{number of bits in fraction} + 1)}} = 0.125$ holds down to the smallest normalized number $2^{\text{min-exp}} = 0.5$
This table also shows the encoding of the 2-bit exponent and 2-bit fraction

<table>
<thead>
<tr>
<th>exponent</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0.0</td>
<td>0.125</td>
<td>0.25</td>
<td>0.375</td>
</tr>
<tr>
<td>$2^{-1}$</td>
<td>01</td>
<td>0.5</td>
<td>0.625</td>
<td>0.75</td>
</tr>
<tr>
<td>$2^0$</td>
<td>10</td>
<td>1.0</td>
<td>1.25</td>
<td>1.5</td>
</tr>
<tr>
<td>$2^1$</td>
<td>11</td>
<td>2.0</td>
<td>2.5</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Floating Point Operations

Addition -- shift smaller number to right until exponents are equal, then add

\[1.5 = 01010 = 1.10 \times 2^0 = 1.10 \times 2^0\]
\[+0.875 = +00111 = +1.11 \times 2^{-1} = +0.11 \times 2^0\]
\[2.375 = 10.01 \times 2^0 = 1.00 \times 2^1 = 2.0 = 01100\]

Note loss of bits when shifting to match exponents and when truncating to fit into 5-bit format (absolute error of 0.375, relative error of 15.8%)
Floating Point Operations

error magnification in case of effective subtraction of two approximately equal numbers [that is, \(a-b\) or \(a+(-b)\), where \(a\) and \(b\) are approx. equal];

this is called **catastrophic cancellation**; the leading (most significant) bits cancel out and all that is left are the trailing (least significant) bits, which are most affected by representation errors.

\[
\begin{align*}
2.00 &= 0\ 11\ 00 = 1.00\times2^1 = 1.00\times2^1 \\
-1.75 &= -0\ 10\ 11 = -1.11\times2^0 = -0.11\times2^1 \\
0.25 &= 0.01\times2^1 = 1.00\times2^{-1} = 0.5 = 0\ 01\ 00
\end{align*}
\]

(absolute error of 0.25, relative error of 100%)
Floating Point Operations

to maintain accuracy in effective subtraction, most FP hardware adds three bits: guard bit, round bit, and sticky bit

e.g., error in the previous subtraction eliminated with addition of guard bit

\[\begin{align*}
2.00 &= 01100 = 1.00 \times 2^1 = 1.000 \times 2^1 \\
-1.75 &= -01011 = -1.11 \times 2^0 = -0.111 \times 2^1 \\
\hline
0.25 &= 0001 \times 2^{-1} = 0.10 \times 2^{-1} = 0.25 = 00010
\end{align*}\]

(these extra bits would also help in the first addition above to produce a result of 2.5, which would then have less absolute error, 0.125, and less relative error, 5.3%)
Floating Point Operations

the extra bits also help to implement round to even

fraction bits | g | r | s |
-------------|---|---|---|
0 0 1 \       |   |   |   | round down (truncate)
0 1 1 |       |   |   |
1 0 0 /       |   |   |   | (0100 round to even -> truncate)
1 0 0 \       |   |   |   | (1100 round to even -> add one to last
1 0 1 |       |   |   | bit in fraction)
... |   |   |   | round up
1 1 1 /
Floating Point Operations

In summary, to round to the nearest even using G, R and S:

0xx - round down = do nothing (x means any bit value)

100 - this is a tie: round up if the mantissa's lsb is 1, else round down=do nothing

101 - round up

110 - round up

111 - round up
Floating Point Operations

multiplication -- multiply 1.fraction parts, add exponents

\[
1.5 = 0\ 10\ 10 = 1.10\times2^0
\]

\[
\times1.5 = \times\ 0\ 10\ 10 = \times1.10\times2^0
\]

\[
2.25 = 10.01\times2^0 = 1.00\times2^1 = 2.0 = 0\ 11\ 00
\]

or rounding up \[
10.01\times2^0 = 1.001\times2^1 = 1.01\times2^1 = 2.5 = 0\ 11\ 01
\]

using extra bit
IEEE Standards

first floating point support was on IBM 704 in 1950s; each manufacturer had its own FP format until standardization by IEEE in 1980s

IEEE formats

single precision - 32-bit format = 1-bit sign, 8-bit exp, 23-bit fraction

double precision - 64-bit format = 1-bit sign, 11-bit exp, 52-bit fraction

also

extended precision - 80-bit format
quad precision - 128-bit format
IEEE Standards

special codes for

NaN - Not a Number (propagates itself through any operation)

infinity - (also propagates in most cases)

denormal numbers
IEEE single-precision floating-point encoding (exponent is bias-127)

23-bit fraction

<table>
<thead>
<tr>
<th>8-bit</th>
<th>000...00</th>
<th>000...01</th>
<th>...</th>
<th>111...11</th>
</tr>
</thead>
</table>

exponent +-----------------------------

<table>
<thead>
<tr>
<th>8-bit</th>
<th>000...00</th>
<th>000...01</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2^(-126)</td>
<td>...</td>
<td>almost 2^(-125)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>011...10</td>
<td>0.5</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>011...11</td>
<td>1.0</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>100...00</td>
<td>2.0</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>111...10</td>
<td>2^(+127)</td>
<td>...</td>
<td>almost 2^(+128)</td>
<td></td>
</tr>
<tr>
<td>111.1111</td>
<td>inf</td>
<td>NaN</td>
<td>...</td>
<td>NaN</td>
</tr>
</tbody>
</table>

"almost" means that instead of exactly two, the significand = 2 - 2^(-23)
Examples

consider this C program

```c
int main(void){
    float a=2.5, b=0.1, c;
    int *pa=(int *)&a,*pb=(int *)&b,*pc=(int *)&c;
    c = a + b;
    printf("a = %f (0x%x)\n",a,*pa);
    printf("b = %f (0x%x)\n",b,*pb);
    printf("c = %f (0x%x)\n",c,*pc);
}
```

it prints the float and hex values (hex requires dereferencing an int pointer)

- a = 2.500000 (0x40200000)
- b = 0.100000 (0x3dcccccd)
- c = 2.600000 (0x40266666)
Examples

next, consider this C program

```c
int main(void){
    int i;
    float sum;
    sum = 0.0;
    for(i=1; i<=10000000; i++){
        sum = sum + 1.0/((float)i);
    }
    printf("decreasing order: %f\n",sum);
    sum = 0.0;
    for(i=10000000; i>0; i--){
        sum = sum + 1.0/((float)i);
    }
    printf("increasing order: %f\n",sum);
}
```

when run, it prints

decreasing order: 15.403683
increasing order: 16.686031

can you explain why? what is the correct answer?
Examples

next, this program

```c
int main(void)
{
    double hd, xd, yd;
    float hs, xs, ys;
    int i;
    hs = 0.1;
    xs = 0.0;
    for(i=0;i<10;i++) { xs += hs; }
    ys = 1.0 - xs;
    hd = 0.1;
    xd = 0.0;
    for(i=0;i<10;i++) { xd += hd; }
    yd = 1.0 - xd;
    printf("left-overs: %g %g\n", ys, yd);
    return 0;
}
```

prints "left-overs: -1.19209e-07 1.11022e-16" -- note the difference in the signs. Can you explain why?

for the single precision case, ten adds of 0.1 is slightly larger than 1
for the single precision case, ten adds of 0.1 is slightly larger than 1

32b SP rep 28b addition (carry, hidden, fraction, grs)

\[
1.0 = 0x3f800000 \\
- xs = 0x3f800001 \\
\text{-----} \quad \text{absolute difference} = 0x00000008 \quad (\text{grs bits} = 0x0) \\
\text{ys} = 0xb4000000 \quad (\text{normalize by 23 left shifts})
\]

\[
ys = -1.0 \times 2^{-23} = -1/8,388,608 = -1.19209e-07
\]
Examples

double sub(int a)
{
    register double x;
    x = a;
    return(x);
}

which, if you compile (only) with "armc -O2 -S", gives (edited) in the .s file:

.global __aeabi_i2d
.text
.align 2
.global sub
.type sub, %function
sub:
    stmfd sp!, {r3, lr}
    bl __aeabi_i2d
    ldmfd sp!, {r3, pc}

where, double __aeabi_i2d(int) is a standard ARM function to convert integer to double)
Finally, to extract and print the fields and values of a float representation.

/* display fields of normalized, non-zero, single-precision float */
/* -- does not range test to verify input value assumptions */

void show_flt( unsigned int a )
{
    union{ unsigned int i; float f; } v;
    unsigned int sign, exp, fraction, significand;
    int int_exp;

    sign = ( a >> 31 ) & 1; /* sign is leftmost bit */
    exp = ( a >> 23 ) & 0xff; /* exp is next 8 bits */
    int_exp = ( (int) exp ) - 127; /* exp is encoded bias-127 */
    fraction = a & 0x007ffffff; /* fraction is last 23 bits */
    significand = fraction | 0x00800000; /* assume a normalized value */
    /* so insert hidden bit */

    v.i = a;
    printf( "%4.1f, 0x%08x, ", v.f, a );
    printf( "sign %x, hex exp %x, int exp %+d, hex significand %x\n",
            sign, exp, int_exp, significand );
}
int main()
{
    union{ float f; unsigned int i; } a;

    a.f = 10.0; show_flt( a.i );
    a.f = 2.0; show_flt( a.i );
    a.f = 1.0; show_flt( a.i );
    a.f = 0.5; show_flt( a.i );
    a.f = 0.1; show_flt( a.i );

    return 0;
}

Note that C will promote (coerce) floats to doubles when passing to printf() and other library routines, so you need to use a union (or int pointer deref).