Floating Point
Floating-Point data types

- Unsigned and two’s complement number systems
- Negating two’s complement numbers
- Binary addition and subtraction
- MIPS logical operations
- ALU design
  - Full adders, multiplexors, 1-bit ALUs
  - Addition/subtraction, logic operations, set less than
  - Overflow and zero detection.
- Carry Lookahead Addition
- Binary multiplication and booth’s algorithm
Floating-Point

- What can be represented in N bits?
- Unsigned: 0 to $2^N - 1$
- 2s Complement: $-2^{N-1}$ to $2^{N-1} - 1$
- But, what about?
  - very large numbers? 9,349,398,989,787,762,244,859,087,678
  - 1.23 x $10^{67}$
  - very small number? 0.000000000000000000000000000000000045691
  - 2.98 x $10^{-32}$
  - fractional values? 0.35
  - Mixed numbers? 10.28
  - Irrationals? $\pi$
Floating-Point

The location of the fractional point (radix) can be moved from one location to another according to the precision. ``that's where the name came from''. It takes the general format as:

Exp. 0. Fraction

“In mathematics, radix point refers to the symbol used in numerical representations to separate the integral part of the number (to the left of the radix) from its fractional part (to the right of the radix). The radix point is usually a small dot, either placed on the baseline or halfway between the baseline and the top of the numerals. In base 10, the radix point is more commonly called the decimal point.”

en.wikipedia.org/wiki/Radix_point
IEEE-754 Floating-Point Numbers

(a) Single format

(b) Double format
IEEE-754 Single Precision Floating-Point Numbers

Single precision (32 bits, float in C)

- Sign: S
- Exponent: E
- Mantissa or significand: M

Exponent:
- Excess 127 binary integer

Mantissa or significand:
- Sign + magnitude, normalized binary significand w/ hidden one bit: 1.M

Actual exponent is:
- \( e = E - 127 \) (bias)

Magnitude of numbers that can be represented is in the range:

- \( 2^{-126} (1.0) \) to \( 2^{127} (2 - 2^{23}) \)

which is approximately:

- \( 1.8 \times 10^{-38} \) to \( 3.40 \times 10^{38} \)
IEEE 754 Normalization

Normalization is the act of shifting the fractional part in order to make the left bit of the fractional point is one. During this shift the exponent is incremented.

**Normalized numbers** are the numbers that have their MSB 1 is in the most left bit of the fractional part.

**Denormalized numbers** are the opposite of the normalized numbers. (i.e. the MSB 1 is not in the most left bit of the fractional part).

**Operations:** Some operations requires that the exponent field is the same for all operands (like addition). In this case one of the operands should be denormalized.

**Importance:** Denormalized numbers have important use in some operations and numbers. For example, assume minimum exponent is -98, and number of digits is 3 and to perform the operation x-y where $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$. The result of this operation is $0.06 \times 10^{-97}$ if This number is normalized in decimal it will be $6.00 \times 10^{-99}$ which is too small to be represented as a normalized number in so it is normalized to zero. but if the result is not normalized we will get the correct result.

**Gradual underflow:** One of the advantages of the denormalized numbers is the gradual underflow. This came from the fact the normalized numbers can represent minimum numbers is $1.0 \times 2^{\text{min}}$ and all numbers smaller than that are rounded to zero (which means there are no numbers between $1.0 \times 2^{\text{min}}$ and 0. The denormalized numbers expands the range and gives gradual underflow through the division of the range between $1.0 \times 2^{\text{min}}$ to 0 with the same steps as the normalized numbers.
What is the single-precision hex number for the signed decimal number +3.421875?

1) convert to binary (S = +): 3 .421875
   0 0000011 .011011

2) Normalize (move radix to most significant 1. In this case 1 place left):
   0000001.1011011

3) calculate single precision bias:
   \((127 + 1) = 128 \Rightarrow 10000000\) binary

4) sign = + = 0

5) assemble all parts (remove normalized 1):
   0 100 0000 0 101 1011 0000 0000 0000 000

6) convert to hex: \(\Rightarrow 405B0000\)
Converting from Binary to Decimal Floating Point

- What is the decimal single-precision floating point number that corresponds to the hex number C2020000

- Use the equation $X = (-1)^S \times 2^{E-127} \times (1.M)$

1) convert to binary: 

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1100 0010 0000 0010 0000 0000 0000

2) remove sign bit:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 0010 0 000 0010 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3) Remove bias: 132 – 127 = 5

4) insert implied 1 and denormalize (shift radix right five places):

100000.10000000000000000000000

5) apply sign bit (1 = negative) and convert to decimal: -32.5
IEEE-754 Double Precision Floating-Point Numbers

double precision
(64 bits, double in C)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
</tbody>
</table>

- **Exponent:** excess 1023 binary integer
- **Sign:** sign + magnitude, normalized binary significand with hidden one bit: 1.M

**Actual exponent:**

\[ e = E - 1023 \text{ (bias)} \]

**Number representation:**

\[ X = (-1)^S 2^{E-1023} (1.M) \]

**Magnitude of numbers that can be represented:**

\[ 2^{-1022} (1.0) \text{ to } 2^{1025} (2 - 2^{52}) \]

which is approximately:

\[ 2.2 \times 10^{-308} \text{ to } 1.8 \times 10^{308} \]

The IEEE 754 standard also supports extended single-precision (more than 32 bits) and extended double-precision (more than 64 bits).

Special values for the exponent and mantissa are used to indicate other values, like zero and infinity.
Converting from Binary to Decimal Floating Point

What is the double-precision hex number that corresponds to the signed decimal number +3.4219?

1) convert to binary:

   00000011.011011

2) Normalize (move radix to most significant 1. In this case 1 place left):

   0000001.1011011

3) calculate double precision bias:

   \[(1023 + 1) = 1024 \Rightarrow 00000100000000000000000000000000000000000000000000\

4) sign = + = 0

5) assemble all parts (remove normalized 1):

   0 10000000000 110110000000000000000000000000000000000000000000000

6) convert to hex: \(\Rightarrow 400B6000000000\)
Converting from Binary to Decimal Floating Point

What is the decimal double-precision floating point number that corresponds to the hex number 4031400000000000

Use the equation \[ X = (-1)^S \times 2^{E-1023} \times (1.M) \]

1) convert to binary:

0100 0000 0011 0001 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

2) remove sign bit:

0 100 0000 0011 0001 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3) remove bias: 1027 – 1023 = 4

4) insert implied 1 and denormalize (shift radix to right 4 places):

1 0001. 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

5) apply sign bit and put into decimal: \( \Rightarrow +17.25 \)
Special cases

denormalized number

A nonzero floating-point number that has the following properties:
- Its exponent has a reserved value, usually the format's minimum.
- Its explicit or implicit leading significand bit is zero.

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point.
Special cases – Infinity and NAN’s

result of operation *overflows*, i.e., is larger than the largest number that can be represented

overflow is not the same as divide by zero (raises a different exception)

+/- infinity

It may make sense to do further computations with infinity
e.g., \( X/0 > Y \) may be a valid comparison

Not a number, but not infinity (e.g. \( \sqrt{-4} \))
invalid operation exception (unless operation is = or =)  \( / \)

\( NaN \)

\( NaN \)s propagate: \( f(NaN) = NaN \)

HW decides what goes here
IEEE 754 NaN

• Some computations generate undefined results like $0/0$ and $\sqrt{(-1)}$. These operations should be handled or we will get strange results and behavior. NaN is defined to be generated upon these operations and so the operations are defined for it to let the computations continue.

• Whenever a NaN participates in any operation the result is NaN.

• There is a family of NaN according to the above table and so the Implementations are free to put any information in the fraction part.

• All comparison operators ($\neq, <, \leq, >, \geq$) (except $\neq$) should return false when NaN is one of its operands.

Sources of NaN

<table>
<thead>
<tr>
<th>Operation</th>
<th>Produced by</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$\infty+(-\infty)$</td>
</tr>
<tr>
<td>$\times$</td>
<td>$0\times\infty$</td>
</tr>
<tr>
<td>/</td>
<td>$0/0, \infty/\infty$</td>
</tr>
<tr>
<td>REM</td>
<td>$x \operatorname{REM} 0, \infty\operatorname{REM} y$</td>
</tr>
<tr>
<td>$\sqrt{}$</td>
<td>$\sqrt{x}$ (when $x &lt; 0$)</td>
</tr>
</tbody>
</table>
IEEE 754 Infinity

• The infinity is like the NaN, it is a way to continue the computation when some operations are occurred.

• **Generation** Infinity is generated upon operations like x/0 where x ≠ 0

• **Results:** The results of operations that get ∞ as parameter is defined as: "Replace the ∞ by the limit lim \(_{x \to \infty}\) \(3/x\) = 0 and \(\sqrt{\infty}\) = ∞ and 4−∞ = −∞

• The infinity is used instead of the saturation on maximum representable number and the computation should continue.

• **Example[1]:** compute \(\sqrt{(x^2+y^2)}\) when max exponent is 98 and only three decimal digits are supported. If x = \(3 \times 10^{70}\) and y = \(4 \times 10^{70}\) and saturation is used \(x^2 = 9.99 \times 10^{98}\) and so \(y^2\). and so the final result is \((9.99 \times 10^{98})^{1/2} = 3.16 \times 10^{49}\) which is different than the correct result \((5 \times 10^{70})\). Instead when Infinity is used, \(x^2 = \infty\) and \(y^2 = \infty\) so the final result is \(\infty\) which is much better than giving incorrect result.
/* Double precision example */

main(argc, argv)

int argc;
char *argv[];
{
    double x, atof();                /* double precision storage */
    int t, y;
    unsigned char *px;              /* address of double */

    if (argc != 2){
        printf("usage: %s real-number \n",argv[0]);
        exit(1);
    }

    x = atof(argv[1]);
    px = (char *)&x;                /* get variable address */

    printf("Value of x = %4.4f\n\n",x);
    printf("It is stored in memory as: \n\n");

    for (t = 0; t < sizeof(x); t++){
        printf("Byte = %d, Contents = %0x\n",t,*px);
        px++;
    }
}

Test run of double precision storage for special case = 0:

<table>
<thead>
<tr>
<th>Value of x = 0.0000</th>
<th>Value of x = 1.0000</th>
<th>Value of x = 42.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is stored in memory as:</td>
<td>It is stored in memory as:</td>
<td>It is stored in memory as:</td>
</tr>
<tr>
<td>Byte = 0, Contents = 0</td>
<td>Byte = 0, Contents = 0</td>
<td>Byte = 0, Contents = 0</td>
</tr>
<tr>
<td>Byte = 1, Contents = 0</td>
<td>Byte = 1, Contents = 0</td>
<td>Byte = 1, Contents = 0</td>
</tr>
<tr>
<td>Byte = 2, Contents = 0</td>
<td>Byte = 2, Contents = 0</td>
<td>Byte = 2, Contents = 0</td>
</tr>
<tr>
<td>Byte = 3, Contents = 0</td>
<td>Byte = 3, Contents = 0</td>
<td>Byte = 3, Contents = 0</td>
</tr>
<tr>
<td>Byte = 4, Contents = 0</td>
<td>Byte = 4, Contents = 0</td>
<td>Byte = 4, Contents = 0</td>
</tr>
<tr>
<td>Byte = 5, Contents = 0</td>
<td>Byte = 5, Contents = 0</td>
<td>Byte = 5, Contents = 0</td>
</tr>
<tr>
<td>Byte = 6, Contents = 0</td>
<td>Byte = 6, Contents = f0</td>
<td>Byte = 6, Contents = 45</td>
</tr>
<tr>
<td>Byte = 7, Contents = 0</td>
<td>Byte = 7, Contents = 3f</td>
<td>Byte = 7, Contents = 40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value of x = -0.0000</th>
<th>Value of x = -1.0000</th>
<th>Value of x = -3.3828</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is stored in memory as:</td>
<td>It is stored in memory as:</td>
<td>It is stored in memory as:</td>
</tr>
<tr>
<td>Byte = 0, Contents = 0</td>
<td>Byte = 0, Contents = 0</td>
<td>Byte = 0, Contents = 0</td>
</tr>
<tr>
<td>Byte = 1, Contents = 0</td>
<td>Byte = 1, Contents = 0</td>
<td>Byte = 1, Contents = 0</td>
</tr>
<tr>
<td>Byte = 2, Contents = 0</td>
<td>Byte = 2, Contents = 0</td>
<td>Byte = 2, Contents = 0</td>
</tr>
<tr>
<td>Byte = 3, Contents = 0</td>
<td>Byte = 3, Contents = 0</td>
<td>Byte = 3, Contents = 0</td>
</tr>
<tr>
<td>Byte = 4, Contents = 0</td>
<td>Byte = 4, Contents = 0</td>
<td>Byte = 4, Contents = 0</td>
</tr>
<tr>
<td>Byte = 5, Contents = 0</td>
<td>Byte = 5, Contents = 0</td>
<td>Byte = 5, Contents = 0</td>
</tr>
<tr>
<td>Byte = 6, Contents = 0</td>
<td>Byte = 6, Contents = f0</td>
<td>Byte = 6, Contents = b</td>
</tr>
<tr>
<td>Byte = 7, Contents = 0</td>
<td>Byte = 7, Contents = bf</td>
<td>Byte = 7, Contents = c0</td>
</tr>
</tbody>
</table>
/* Single precision example - the only change is to declare variable x and atof function as single precision floats */

main(argc, argv)

int argc;
char *argv[];
{

float x, atof(); /* single precision storage */
int t, y;
unsigned char *px; /* address of double */

if (argc != 2){
    printf("usage: %s real-number \n",argv[0]);
    exit(1);
}

x = atof(argv[1]);
px = (char *)&x; /* get variable address */

printf("Value of x = %4.4f\n\n",x);
printf("It is stored in memory as: \n\n");

for (t = 0; t < sizeof(x); t++){
    printf("Byte = %d, Contents = %0x\n",t,*px);
    px++;
}
}
Single precision storage

Value of $x = 315.0000$

It is stored in memory as:

Byte = 0, Contents = 0
Byte = 1, Contents = 80
Byte = 2, Contents = 9d
Byte = 3, Contents = 43

Value of $x = -92.0000$

It is stored in memory as:

Byte = 0, Contents = 0
Byte = 1, Contents = 0
Byte = 2, Contents = b8
Byte = 3, Contents = c2
DEFINITION - Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory. Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first. For example, in a big-endian computer, the two bytes required for the hexadecimal number 4F52 would be stored as 4F52 in storage (if 4F is stored at storage address 1000, for example, 52 will be at address 1001). In a little-endian system, it would be stored as 524F (52 at address 1000, 4F at 1001).

IBM's 370 mainframes, most RISC-based computers, and Motorola microprocessors use the big-endian approach. TCP/IP also uses the big-endian approach (and thus big-endian is sometimes called network order).

On the other hand, Intel processors (CPUs) and DEC Alphas and at least some programs that run on them are little-endian.

Big-endian and little-endian derive from Jonathan Swift's Gulliver's Travels in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

Source: http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211659,00.html
IEEE 754 Precision

• **Precisions** The IEEE-754 defines set of precisions which depends on the number of bits used. There are two main precisions, the single and the double ``the quad-precision is not used often''.

• **Extended** The standard also define an extended precision for both standard precisions. The number of used bits is enlarged. The standard defines the minimum number of bits of the extended format. and its up to the implementer to increase it. The IEEE standard requires that the implementation should support the corresponding extended format.

• **Reason** The main reason for extended format came from calculators which displays 10 digits but use 13 digits internally, which makes the user feel as if the calculator computes to 10 digits accuracy[1]. This feature is important to make all calculations on all IEEE-754 platforms give the same results after rounding. It is also needed to distinguish between exact and inexact results.

• **Note:** The standard requires that all calculations are made in extended format and then rounded to the precision. It is important to round the extended result of each operation alone to the corresponding precision. Because if it is not done the final result will depend on the extra bits and produce and unexpected results.
IEEE 754 Zero

The zero is represented as a signed zero (-0 and +0) it is represented as min-1 in the exponent and zero in the fraction. The signed zero is important for operations that preserves the sign like multiplication and division. It is also important to generate $+\infty$ or $-\infty$

It is also used in the signum function that return the sign of a number.

Event though the standard defines the comparison $-0 = +0$ as true.
IEEE 754 Exceptions

• Exceptions are important factors in the standard to signal the system about some operations and results.
• when an exception occurs, a status flag is set.

• The implementation should provide the users with a way to read and write the status flags.

• The Flags are "sticky" which means once a flag is set it remains until its explicitly cleared.

• The implementation should give the ability to install trap handlers that can be called upon exceptions.

• **Overflow, underflow and division by zero** are obvious from the table below. The distinction between Overflow and division by zero is to give the ability to distinguish between the source of the infinity in the result.

• **Invalid** This exception is generated upon operations that generates NaN results. But this is not a reversible relation (i.e. if the output is NaN because one of the inputs is NaN this exception will not raise).

• **Inexact** It is raised when the result is not exact because the result can not be represented in the used precision and rounding cannot give the exact result.

• **Software flags** The inexact result exception is raised so often. So some implementations suggests that the hardware generates interrupts upon exceptions, while the software keeps the sticky status flags. In this case once an exception occurs, an interrupt is signaled to the software and the flag is set and that interrupt is masked. Once the flag is unset the interrupt is unmasked again.

Exceptions in IEEE 754 standard

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cased by</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>Operation produce large number</td>
<td>±∞</td>
</tr>
<tr>
<td>Underflow</td>
<td>Operation produce small number</td>
<td>0</td>
</tr>
<tr>
<td>Divide by Zero</td>
<td>x/0</td>
<td>±∞</td>
</tr>
<tr>
<td>Invalid</td>
<td>Undefined Operations</td>
<td></td>
</tr>
<tr>
<td>Inexact</td>
<td>Not exact results</td>
<td>Round(x)</td>
</tr>
</tbody>
</table>