Bit operations

1 and 3 ➔ exclusive OR (^)
2 and 4 ➔ and (&)
5 ➔ or (|)

01100 carry*
0110 a
0111 b
01101 a+b

* Always start with a carry-in of 0

Did it work?
What is a?
What is b?
What is a+b?
What if 8 bits instead of 4?
Different encoding scheme than float

* Total number of values: $2^w$
  - where $w$ is the bit width of the data type

The left-most bit is the sign bit if using a signed data type (typically... B2T).

Unsigned $\rightarrow$ non-neg numbers ($\geq 0$)
  - Minimum value: 0
  - Maximum value: $2^{w-1}$

Signed $\rightarrow$ neg, zero, and pos numbers
  - Minimum value: $-2^{w-1}$
  - Maximum value: $2^{w-1}-1$

* Where $w$ is the bit width of the data type
Integer Decoding

- Binary to Decimal (mult of powers)
- Unsigned = simple binary = B2U
  - You already know how to do this :o)
    - 0101 = 5, 1111 = F, 1110 = E, 1001 = 9
- Signed = two’s complement = B2T*
  - 0 101 = positive number; same as B2U = 5
  - 1 111 = -1*2^3 + 7 = -8 + 7 = -1
  - 1 110 = -1*2^3 + 6 = -8 + 6 = -2
  - 1 001 = -1*2^3 + 1 = -8 + 1 = -7
  - Another way, if sign bit = 1, then it’s a negative number and to get the absolute value of that number, you:
    - invert bits and add 1
    - right to left, leave alone to first 1, then invert rest
    - 1010 = lookup says -6... verify both ways

* reminder: left most bit is sign bit
B2O & B2S

One’s complement = bit complement of B2U
Signed Magnitude = left most bit set to 1 with B2U for the remaining bits
Both include neg values
Min/max = \(-2^{w-1} - 1\) to \(2^{w-1} - 1\)
Pos and neg zero
Difficulties with arithmetic options

Note that +0 and −0 return TRUE when tested for zero, FALSE when tested for non-zero.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
<td>-7</td>
<td>-0</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
<td>-6</td>
<td>-1</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
<td>-2</td>
<td>-5</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
<td>-1</td>
<td>-6</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
<td>0</td>
<td>-7</td>
</tr>
</tbody>
</table>
Signed vs Unsigned

- Casting...
  - Signed to unsigned...
  - Unsigned to signed...

*** Changes the meaning of the value, but not the bit representation

- Notice, the difference of 16 i.e. left most bit →
  - Unsigned = $2^3 = 8$
  - Signed = $-2^3 = -8$

What is the largest possible value for short? 16-bits... see <limits.h>
When add 1 to -1 get 0; when add 1 to 4,294,967,295, you get zero. Why?
FFFF + 1 = 0000... overflow warning but okay i.e. -1 + 1 = 0... all is good
Signed vs Unsigned (cont)

When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative.

- Since bit representation does not change, it really doesn’t matter arithmetically
- However... relational operators have issues

<table>
<thead>
<tr>
<th>EXPRESSION</th>
<th>TYPE</th>
<th>EVALUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0u</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0u</td>
<td>unsigned</td>
<td>0*</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647-1</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>2147483647u &gt; -2147483647-1</td>
<td>unsigned</td>
<td>0*</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned) -1 &gt; -2</td>
<td>unsigned</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ 1 = \text{TRUE and } 0 = \text{FALSE} \]
### Sign Extend

- Already have an intuitive sense of this

\[ w = 8 \]

-27 B2T => B2U for +27 = 00011011

=> invert + 1 for -27 = 11100101

\[ w = 16, \text{ sign extend} \]

-27 B2T => B2U for +27 = 00000000 00011011

-27 = 11111111 11100101

- For unsigned
  - Fill to left with zero

- For signed
  - Repeat sign bit

### Truncation

- Drops the high order w-k bytes when truncating a w-bit number to a k-bit number

- 4-bit to 3-bit truncation

<table>
<thead>
<tr>
<th>HEX</th>
<th>UNSIGNED – B2U</th>
<th>TWO'S COMP – B2T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig truncate</td>
<td>orig truncate</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>-7</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>-5</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
# Integer Addition

Signed or unsigned... that is the question!

- **Unsigned** 4-bit BTU
  - 0 to 15 are valid values
  - Only checking carry-out

- **Signed** 4-bit B2T
  - -8 to 7 are valid values
  - Checking carry-in AND carry-out

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x+y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>13</td>
<td>13 ok</td>
</tr>
<tr>
<td>00 101</td>
<td>1101</td>
<td>15</td>
<td>15 ok</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>17</td>
<td>1 OF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x+y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>-5</td>
<td>-13</td>
<td>3 negOF</td>
</tr>
<tr>
<td>1000</td>
<td>1011</td>
<td>10 011</td>
<td>0 negOF</td>
</tr>
<tr>
<td>-8</td>
<td>-8</td>
<td>-16</td>
<td>-3 ok</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1 0000</td>
<td>1101 ok</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>7 ok</td>
</tr>
<tr>
<td>00 10</td>
<td>0101</td>
<td>10 010</td>
<td>-6 posOF</td>
</tr>
</tbody>
</table>

Negative overflow when \(x+y < -2^{w-1}\)
Positive overflow when \(x+y > 2^{w-1}-1\)
### B2T integer negation

- **How determine a negative value in B2T?**
  - Reminder: B2U=B2T for positive values
  - B2U → invert the bits and add one

- **Two’s complement negation**
  - \(-2^{w-1}\) is its own additive inverse
  - Other values are negated by integer negation
  - Bit patterns generated by two’s complement are the same as for unsigned negation

<table>
<thead>
<tr>
<th>GIVEN</th>
<th>NEGATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEX</td>
<td>binary</td>
</tr>
<tr>
<td>0x00</td>
<td>0b00000000</td>
</tr>
<tr>
<td>0x40</td>
<td>0b01000000</td>
</tr>
<tr>
<td>0x80</td>
<td>0b10000000</td>
</tr>
<tr>
<td>0x83</td>
<td>0b10000011</td>
</tr>
<tr>
<td>0xFD</td>
<td>0b11111101</td>
</tr>
<tr>
<td>0xFF</td>
<td>0b11111111</td>
</tr>
</tbody>
</table>

*binary = invert the bits and add 1

See next page: signvsuns.c
#include <stdio.h>
#include <limits.h>

void main() {  
    int n = 0;
    printf("neg of %d is %d ",n,-n);
    n = 64;
    printf("\nneg of %d is %d ",n,-n);
    n = -64;
    printf("\nneg of %d is %d ",n,-n);
    n = INT_MIN;
    printf("\nneg of %d is %d ",n,-n);

    unsigned int a = 0;
    printf("\n\n0 - 1 unsigned is %u ",a-1);
    printf("\nunsigned max is %u ", UINT_MAX);
    a = 5;
    printf("\nneg of unsigned %d is %u",a, -a);  
  }
## Rounding

<table>
<thead>
<tr>
<th>$y$</th>
<th>round down (towards $-\infty$)</th>
<th>round up (towards $+\infty$)</th>
<th>round towards zero</th>
<th>round away from zero</th>
<th>round to nearest</th>
</tr>
</thead>
<tbody>
<tr>
<td>+23.67</td>
<td>+23</td>
<td>+24</td>
<td>+23</td>
<td>+24</td>
<td>+24</td>
</tr>
<tr>
<td>+23.50</td>
<td>+23</td>
<td>+24</td>
<td>+23</td>
<td>+24</td>
<td>+24</td>
</tr>
<tr>
<td>+23.35</td>
<td>+23</td>
<td>+24</td>
<td>+23</td>
<td>+24</td>
<td>+23</td>
</tr>
<tr>
<td>+23.00</td>
<td>+23</td>
<td>+23</td>
<td>+23</td>
<td>+23</td>
<td>+23</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>−23.00</td>
<td>−23</td>
<td>−23</td>
<td>−23</td>
<td>−23</td>
<td>−23</td>
</tr>
<tr>
<td>−23.35</td>
<td>−24</td>
<td>−23</td>
<td>−23</td>
<td>−24</td>
<td>−23</td>
</tr>
<tr>
<td>−23.50</td>
<td>−24</td>
<td>−23</td>
<td>−23</td>
<td>−24</td>
<td>−24</td>
</tr>
<tr>
<td>−23.67</td>
<td>−24</td>
<td>−23</td>
<td>−23</td>
<td>−24</td>
<td>−24</td>
</tr>
</tbody>
</table>
Rounding – our system

```c
#include <stdio.h>
void main()
{
    float y[9] = {23.67, 23.50, 23.35, 23.00, 0, -23, -23.35, -23.5, -23.67};
    int i;
    for (i=0;i<9;i++) {
        printf("y = %.4f %.2f %.1f %.0f\n", y[i],y[i],y[i],y[i]);
    }
}
```

//OUTPUT

```
 y = 23.6700  23.67  23.7  24
 y = 23.5000  23.50  23.5  24
 y = 23.3500  23.35  23.4  23
 y = 23.0000  23.00  23.0  23
 y = 0.0000  0.00  0.0  0
 y = -23.0000 -23.00 -23.0 -23
 y = -23.3500 -23.35 -23.4 -23
 y = -23.5000 -23.50 -23.5 -24
 y = -23.6700 -23.67 -23.7 -24
```
Integer multiplication

- 1 * 1 = 1
- 1 * 0 = 0 * 1 = 0 * 0 = 0

<table>
<thead>
<tr>
<th>8-bit multiplication</th>
<th>8-bit multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4 -&gt;</td>
<td>1 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>9 -&gt;</td>
<td>0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 0 0</td>
<td>1 1 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1 1 0 0</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 0 0</td>
<td>0 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 0 0</td>
<td>0 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
<td>0 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>&lt;- carry</td>
<td>&lt;- carry</td>
</tr>
<tr>
<td>1 0 0 0 1 1 0 1 1 1 0 0 = -36?</td>
<td>0 1 1 0 1 1 1 0 0 = 108?</td>
</tr>
</tbody>
</table>

B2T 8-bit range → -128 to 127

B2U 8-bit range → 0 to 255

B2U = 252*9 = 2268 (too big)

B2T = same

Typically, if doing 8-bit multiplication, you want 16-bit product location i.e. 2w bits for the product
## Integer multiplication (cont)

- **Unsigned i.e. simple binary**
  - For x and y, each with the same width (w)
  - x*y yields a w-bit value given by the low-order w bits of the 2w-bit integer product
    - Equivalent to computing the product (x*y) modulo 2^w
    - Result interpreted as an unsigned value

- **Signed = similar, but result interpreted signed value**

<table>
<thead>
<tr>
<th>binary</th>
<th>unsigned</th>
<th>two's comp</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111*0011</td>
<td>7*3=21=00010101</td>
<td>same</td>
<td>0101</td>
</tr>
<tr>
<td>1001*0100</td>
<td>9*4=36=00100100</td>
<td>-7*4=-28=11100100</td>
<td>0100</td>
</tr>
<tr>
<td>1100*0101</td>
<td>12*5=60=00111100</td>
<td>-4*5=-20=11101100</td>
<td>1100</td>
</tr>
<tr>
<td>1101*1110</td>
<td>13*14=182=101110110</td>
<td>-3*-2=6=00000110</td>
<td>0110</td>
</tr>
<tr>
<td>1111*0001</td>
<td>15*1=15=00001111</td>
<td>-1*1=-1=11111111</td>
<td>1111</td>
</tr>
</tbody>
</table>
Multiply by constants

First case: Multiplying by a power of 2
- Power of 2 represented by \( k \)
- So \( k \) zeroes added in to the right side of \( x \)
  - Shift left by \( k \): \( x<<k \)
- Overflow issues the same as \( x*y \)

General case
- Every binary value is an addition of powers of 2
- Has to be a run of one’s to work
  - Where \( n \) = position of leftmost 1 bit in the run and \( m \) = the rightmost
- “multiplication of powers” where \( K = 7 = 0111 = 2^2 + 2^1 + 2^0 \)
  - \((x<<n)+(x<<n-1) + \ldots + (x<<m)\)
- Also equal to \( 2^3 - 2^0 = 8 - 1 = 7 \)
  - \((x<<n+1) - (x<<m)\) ... subtracting

Why looking at it this way?
- Shifting, adds and subtracts are quicker calculations than multiplication (2.41)
- Optimization for C compiler

What is \( x*4 \) where \( x = 5 \)?
- \( x = 5 = 00000101 \)
- \( 4 = 2^k \), so \( k = 2 \)
- \( x<<(k) = 00010100 = 20 \)
- What if \( x = -5 \)?

\( x = 5, \ n = 2 \ and \ m = 0 \)
- \( x*7 = 35? \)
- \( x<<(2) + x<<(1) + x<<(0) \)
- \( 00010100 \)
- \( 00001010 \)
- \( 00000101 \)
- \( 00100011 = 35? \ YES \)
- OR
- \( 00101000 \)
- \( 11111011 \)
- \( 00100011 \)
Multiply by constants (cont)

What if the bit position \( n \) is the most significant bit? assuming 4 bits for this part (easier!)

Since \((x<<n+1) - (x<<m)\) and shifting \( n+1 \) times gives zero, then formula is \(-(x<<m)\)

What if \( K \) is negative? i.e. \( K = \text{constant} = -6\)

back to 8-bits

\[ 0000 \ 0110 = +6 \]

Using \((x<<n+1) - (x<<m)\) where \( n=2 \) and \( m=1 \)

If want \(-6\) then negate formula which gives:

\[ (x<<m) - (x<<n+1) \]
\[ 2^1 - 2^3 \]
which equals \( 2 - 8 = -6 \) (good) so...

\[ (x<<1) - (x<<3) = 0000 \ 1010 - 0010 \ 1000 \]

To add, negate the 2nd value \( \rightarrow \ 1101 \ 1000 \)

And \( 0000 \ 1010 + 1101 \ 1000 = 1110 \ 0010 = -30? \) YES!
Dividing by powers of 2

- Even slower than integer multiplication
- Dividing by powers of 2 $\rightarrow$ right shifting
  - Logical - unsigned
  - Arithmetic – two’s complement
- Integer division always rounds toward zero (i.e. truncates)
  - C float-to-integer casts round towards zero.
  - These rounding errors generally accumulate

<table>
<thead>
<tr>
<th>8÷2</th>
<th>100</th>
<th>9÷2</th>
<th>100</th>
<th>12÷4</th>
<th>100</th>
<th>15÷4</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1 0 0 0</td>
<td>10</td>
<td>1 0 0 1</td>
<td>100</td>
<td>1 1 0 0</td>
<td>100</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

What if the binary numbers are B2T?
Unsigned Integer Division by $2^k$

- Logical right shift by $k$ ($x >>= k$)
  - $x$ divided by $2^k$ and then rounding toward zero
  - Pull in zeroes

Example $x = 12$

- $1100 >> 0110 >> 0011 >> 0001 >> 0000$
- $12/2 = 6/2 = 3/2 = 1/2 = 0$
- $12/2^1 = 12/2^2 = 12/2^3 = 12/2^4$

Example $x = 15$

- $1111 >> 0111 >> 0011 >> 0001 >> 0000$
- $15/2 = 7/2 = 3/2 = 1/2 = 0$
- $15/2^1 = 15/2^2 = 15/2^3 = 15/2^4$

Example

- $12/2^2 = 3$
- $10/2^2 = 2$
- $12/2^3 = 1$
Signed Integer Division by $2^k$

- **Two’s complement**
- **Sign extend for arithmetic shift**
- **Examples**
- **Rounds down (rather than towards zero)**
  - $-7/2$ should yield $-3$ rather than $-4$

<table>
<thead>
<tr>
<th>$k$</th>
<th>Binary</th>
<th>Decimal</th>
<th>$-82/2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 1 0 1 1 1 0</td>
<td>-82</td>
<td>-82.000000</td>
</tr>
<tr>
<td>1</td>
<td>1 1 0 1 0 1 1 1</td>
<td>-41</td>
<td>-41.000000</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1 0 1 0 1 1</td>
<td>-21</td>
<td>-20.500000</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1 1 0 1 0 1</td>
<td>-11</td>
<td>-10.250000</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 1 1 0 1 0</td>
<td>-6</td>
<td>-5.125000</td>
</tr>
<tr>
<td>5</td>
<td>1 1 1 1 1 1 0 1</td>
<td>-3</td>
<td>-2.562500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$y$</th>
<th>same binary</th>
<th>Dec (rd)</th>
<th>$(x+y-1)/y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-82</td>
<td>-82.000000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-41</td>
<td>-40.500000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-20</td>
<td>-19.750000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-10</td>
<td>-9.375000</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>-5</td>
<td>-4.187500</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>-2</td>
<td>-1.593750</td>
<td></td>
</tr>
</tbody>
</table>

For now...

Corrected...

$x = -82$
$y = 2^k$
Signed Integer Division by $2^k$ (cont)

- By adding a “bias” of y-1 before performing the arithmetic right shift causes the result to be correctly rounded “upward” i.e. towards zero

- So using the given property,

  $$\left\lfloor \frac{x}{y} \right\rfloor = \left\lfloor \frac{x + y - 1}{y} \right\rfloor$$

- For integers x and y such that y > 0
  - Example: x=-30, y=4
    - x+y-1=-27 and -30/4 (ru) = -7 = -27/4 (rd)
  - Example x=-32 and y=4
    - x+y-1=-29 and -32/4 (ru) = -8=-29/4 (rd)

- Equivalent to $x/2^k$
  - $y = 1 << k$
    - $4=2^k$, so if $y = 4$ then $k = 2$, so shift 1 left by 2 = 100 is 4
  - In C ➜ $(x<0 ? x+(1<<k)-1 : x) >> k$
Bit-level operations in C

- Can be applied to any “integral” data type
  - One declared as type char or int
    - with or without qualifiers (ex. short, long, unsigned)
- How use?
  - Expand hex arguments to their binary representations
  - Perform binary operation
  - Convert back to hex
- NOTE: the expression ~0 will yield a mask of all ones, regardless of the word size of the machine; same as 0xFFFFFFFF for a 32-bit machine, but such code is not portable.

<table>
<thead>
<tr>
<th>value of x</th>
<th>machine rep</th>
<th>mask</th>
<th>type of x and mask</th>
<th>c expr</th>
<th>result</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>153 (base 10)</td>
<td>0b10011001 == 0x99</td>
<td>0b10000000 == 0x80</td>
<td>char</td>
<td>x &amp; mask</td>
<td>0b10000000 == 0x80</td>
<td>2^7 = 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mask &gt;&gt; 1</td>
<td>0b01000000 == 0x40</td>
<td>2^6 = 64 (etc)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b01000000 == 0x40</td>
<td>x &amp; mask</td>
<td>0b00000000 == 0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>153 (base 10)</td>
<td>0b10011001 == 0x99</td>
<td>0b10000000 == 0x80</td>
<td>int</td>
<td>x &amp; mask</td>
<td>0b10000000 == 0x80</td>
<td>same</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x &lt;&lt; 1</td>
<td></td>
<td>????</td>
</tr>
</tbody>
</table>
Shift operations

- **Shifting bit patterns**
  - What if shift > sizeof variable type? “undefined”
    - “warning: right shift count >= width of type”
    - k (shift) mod w (width)
  - What if shift using a negative number? “undefined”
    - “warning: right shift count is negative”
    - w (width) – k (neg shift)

- **Left shift**
  - Always brings in zero, but
    - What if left shift a signed value? Oh well...

- **Right shift**
  - Logical – is unsigned; always brings in zeros
  - Arithmetic – is signed; repeats left most bit (sign bit)
    - sign specification default in C is “signed”
    - Almost all compilers/machines do repeat sign (most)

- **Examples** (shiftex.c)
  - Note: difference between 0xFOF0 and 0xFOF0F0F0F0?
Boolean Algebra

Boolean algebra has many of the same properties as arithmetic over integers

- **+ and &&**
  - Multiplication distributes over addition
    - \( a \cdot (b+c) = (a \cdot b) + (a \cdot c) \)
  - Boolean operation & distributes over ||
    - \( a \& (b | c) = (a \& b) | (a \& c) \)
  - Boolean operation || distributes over &
    - \( a | (b & c) = (a | b) & (a | c) \)
  - CANNOT distribute addition over multiplication
    - \( a + (b \cdot c) \neq (a + b) \cdot (a + c) \ldots \) for all integers

Boolean ring – commonalities with integers

- Every value has an additive inverse \(-x\), such that \( x + -x = 0 \)
- \( a^a = 0 \) each element is its own additive inverse
  - \((a^b)^a = b \) above holds even in different ordering
  - Consider (swap):
    - \(*y = *x \land *y;\)
    - \(*x = *x \land *y;\)
    - \(*y = *x \land *y;\)
      - Don’t worry about dereferencing issues... just substitute
      - If \(*y = *x^y\), then the next line is equal to
        - \(*x = *x \land (x \land y) \) so \(*x = *y\)
      - And \(*y = (*x^x^y) \land x^y = *x\)
IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

- Limited range and precision (finite space)
- Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers.
- Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.
Floating Point

“real numbers” having a decimal portion != 0

Example: 123.14 base 10

Meaning:

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

Digit format: $d_md_{m-1}...d_1d_0 . d_{-1}d_{-2}...d_{-n}$

$\text{dnum} \rightarrow \text{summation of} (i = -n \text{ to } m) \ d_i \times 10^i$

Example: 110.11 base 2

Meaning:

- $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$

Digit format: $b_mb_{m-1}...b_1b_0 . b_{-1}b_{-2}...b_{-n}$

$\text{bnum} \rightarrow \text{summation of} (i = -n \text{ to } m) \ b_i \times 2^i$

“.” now a “binary point”

In both cases, digits on the left of the “point” are weighted by positive power and those on the right are weighted by negative powers.
Floating Point

- Shifting the binary point one position left
  - Divides the number by 2
  - Compare 101.11 base 2 with 10.111 base 2

- Shifting the binary point one position right
  - Multiplies the number by 2
  - Compare 101.11 base 2 with 1011.1 base 2

- Numbers 0.111...11 base 2 represent numbers just below 1
  \[ 0.111111 \text{ base 2} = \frac{63}{64} \]

- Only finite-length encodings
  - 1/3 and 5/7 cannot be represented exactly

- Fractional binary notation can only represent numbers that can be written \( x \times 2^y \) i.e. \( \frac{63}{64} = 63 \times 2^{-6} \)
  - Otherwise, approximated
  - Increasing accuracy = lengthening the binary representation but still have finite space
Fractions value of the following binary values:

- .01 =
- .010 =
- 1.00110 =
- 11.001101 =

123.45 base 10

Binary value =

FYI also equals:

- 1.2345 x 10^2 is normalized form
- 12345 x 10^-2 uses significand/mantissa/coefficient and exponent
Floating point example

Put the decimal number 64.2 into the IEEE standard single precision floating point representation...

SEE HANDOUT
IEEE standard floating point representation

- The bit representation is divided into 3 fields
  - The single sign bit $s$ directly encodes the sign $s$
  - The $k$-bit exponent field encodes the exponent
    $\text{exp} = e_{k-1}...e_1 e_0$
  - The $n$-bit fraction field encodes the significand $M$ (but the value encoded also depends on whether or not the exponent field equals 0... later)
    $\text{frac} = f_{n-1}...f_1 f_0$

- Two most common formats
  - Single precision (float)
  - Double-Precision (double)

<table>
<thead>
<tr>
<th>Format</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
<th>Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision (4 bytes)</td>
<td>1 [31]</td>
<td>8 [30-23]</td>
<td>23 [22-00]</td>
<td>127</td>
</tr>
<tr>
<td>Double Precision (8 bytes)</td>
<td>1 [63]</td>
<td>11 [62-52]</td>
<td>52 [51-00]</td>
<td>1023</td>
</tr>
</tbody>
</table>
The sign bit is as simple as it gets.

0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The exponent field needs to represent both positive and negative exponents.

A *bias* is added to the actual exponent in order to get the stored exponent.

For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.
In IEEE 754 floating point numbers, the exponent is biased in the engineering sense of the word – the value stored is offset from the actual value by the exponent bias.

**Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**

To solve this problem the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison.

By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.

When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.

For a single-precision number, an exponent in the range −126 .. +127 is biased by adding 127 to get a value in the range 1 .. 254 (0 and 255 have special meanings).

For a double-precision number, an exponent in the range −1022 .. +1023 is biased by adding 1023 to get a value in the range 1 .. 2046 (0 and 2047 have special meanings).
The fraction

- Typically called the “significand”
- Represents the precision bits of the number.
- It is composed of an implicit (i.e. hidden) leading bit and the fraction bits.
- In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form.

FYI: A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa/significand has effectively 24 bits of resolution, by way of 23 fraction bits.
Putting it all together

So, to sum up:

- The sign bit is 0 for positive, 1 for negative.
- The exponent's base is two.
- The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
- The first bit of the mantissa/significand is typically assumed to be $1.f$, where $f$ is the field of fraction bits.
Another Example

- $\pi$, rounded to 24 bits of precision, has:
  - sign = 0;
  - $e = 1$;
  - $f = 11001001000011111011011$ (including the hidden bit)
- The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

  0 10000000 1001001000011111011011
  (excluding the hidden bit) = 0x40490FDB

- In binary single-precision floating-point, this is represented as $s = 1.10010010000111110110111$ with $e = 1$. This has a decimal value of
  - 3.1415927410125732421875, whereas a more accurate approximation of the true value of $\pi$ is
  - 3.14159265358979323846264338327950...
- The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of $\pi$ in the first 7 digits. The difference is the discretization error and is limited by the machine epsilon.
Why are we doing this?

- Can’t use integers for everything
- Trying to cover a much broader range of real values; but something has to give, and it’s the precision
- Pi a good example:
  - Whether or not a rational number has a terminating expansion depends on the base.
    - For example, in base-10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...).
    - In base-2 only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion.
Special values

The hardware that does arithmetic on floating point numbers must be constantly checking to see if it needs to use a hidden bit of a 1 or a hidden bit of a 0 (for 0.0)

Zero could be 0x00000000 or 0x80000000

What number(s) cannot be represented because of this?

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>e</th>
<th>f</th>
<th>hidden bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0 or 1</td>
<td>all zero</td>
<td>all zero</td>
<td>0</td>
</tr>
<tr>
<td>subnormal</td>
<td>0 or 1</td>
<td>all zero</td>
<td>not all zero</td>
<td>0</td>
</tr>
<tr>
<td>normalized</td>
<td>0 or 1</td>
<td>&gt;0</td>
<td>any bit pattern</td>
<td>1</td>
</tr>
<tr>
<td>+infinity</td>
<td>0</td>
<td>11111111</td>
<td>00000... (0x7f80 0000)</td>
<td></td>
</tr>
<tr>
<td>-infinity</td>
<td>1</td>
<td>11111111</td>
<td>00000... (0xff80 0000)</td>
<td></td>
</tr>
<tr>
<td>NaN*</td>
<td>0 or 1</td>
<td>0xff</td>
<td>anything but all zeros</td>
<td></td>
</tr>
</tbody>
</table>

*Not a Number
### 5-bit floating point representation with one sign bit, two exponent bits \((k=2)\) and two fraction bits \((n=2\text{ so } 2^{n}=4)\); the exponent bias is \(2^{2-1}-1 = 1\)

<table>
<thead>
<tr>
<th>bits</th>
<th>e</th>
<th>E</th>
<th>(2^{E})</th>
<th>f</th>
<th>M</th>
<th>(2^{E}xM)</th>
<th>V</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0/4</td>
<td>0/4</td>
<td>0/4</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>00001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>0.25</td>
</tr>
<tr>
<td>00010</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2/4</td>
<td>2/4</td>
<td>2/4</td>
<td>1/2</td>
<td>0.50</td>
</tr>
<tr>
<td>00011</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3/4</td>
<td>3/4</td>
<td>3/4</td>
<td>3/4</td>
<td>0.75</td>
</tr>
<tr>
<td>00100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0/4</td>
<td>4/4</td>
<td>4/4</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>00101</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1/4</td>
<td>5/4</td>
<td>5/4</td>
<td>5/4</td>
<td>1.25</td>
</tr>
<tr>
<td>00110</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2/4</td>
<td>6/4</td>
<td>6/4</td>
<td>3/2</td>
<td>1.50</td>
</tr>
<tr>
<td>00111</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3/4</td>
<td>7/4</td>
<td>7/4</td>
<td>7/4</td>
<td>1.75</td>
</tr>
<tr>
<td>01000</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0/4</td>
<td>0/4</td>
<td>8/4</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>01001</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1/4</td>
<td>1/4</td>
<td>10/4</td>
<td>5/2</td>
<td>2.50</td>
</tr>
<tr>
<td>01010</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2/4</td>
<td>2/4</td>
<td>12/4</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td>01011</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3/4</td>
<td>3/4</td>
<td>14/4</td>
<td>7/2</td>
<td>3.50</td>
</tr>
<tr>
<td>01100</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>inf</td>
<td>--</td>
</tr>
<tr>
<td>01101</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>01110</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>01111</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>NaN</td>
<td>--</td>
</tr>
</tbody>
</table>

Note the transition between denormalized and normalized
Have to always check for the hidden bit

- **e**: the value represented by considering the exponent field to be an unsigned integer
- **E**: the value of the exponent after biasing = e - bias ... 1-bias when e = 0
- **2^E**: numeric weight of the exponent
- **f**: the value of the fraction
- **M**: the value of the significand =1+f ==1.f
- **V**: the reduced fractional value of the number
- **2^E*M**: the (unreduced) fractional value of the number
- **Decimal**: the decimal representation of the number
Denormalized values

- Also called denormal or subnormal numbers
- Values that are very close to zero
- Fill the “underflow” gap around zero
- Any number with magnitude smaller than the smallest normal number (i.e. values == x*2^y)
- When the exponent field is all zeros
  - E = 1-bias
- Significand M = f without implied leading 1
  - h = 0 (hidden bit)
- Two purposes
  - Provide a way to represent numeric value 0
    - -0.0 and +0.0 are considered different in some ways and the same in others
  - Represents numbers that are very close to 0.0
    - Gradual underflow = possible numeric values are spaced evenly near 0.0
In a normal floating point value there are no leading zeros in the significand, instead leading zeros are moved to the exponent.

For example:

0.0123 would be written as 1.23 * 10^{-2}.

Denormal numbers are numbers where this representation would result in an exponent that is too small (the exponent usually having a limited range). Such numbers are represented using leading zeros in the significand.