

Visualizing Overflow

4 bit unsigned integers

Smallest value: 0 (0000)

largest value: 15 (1111)

total possible values: 16

$$15 + 1 = ?$$

$$\begin{array}{r} 1111 \\ + 0001 \\ \hline 10000 \end{array}$$

we only have 4 bits
this one is truncated

$$15 + 4 = ?$$

$$\begin{array}{r} 1 \\ 1111 \\ 0100 \\ \hline \end{array}$$

$$10011$$



truncate

$$0011 \rightarrow 3$$

$$15 + 4 = 19$$

$$19 \% 16 = 3$$

Binary Addition: Every Combination
possible when adding two
binary digits

when adding binary digits,
each column has three inputs

Three Inputs

digit A: bit from first num

digit B: bit from second num

carry in: a carry from the
column immediately to
the right

ex:
$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \\ + \quad 0 \quad 0 \quad 0 \quad 1 \\ \hline \end{array}$$

Annotations:

- 1 ← carry in
- digit A (points to the 1 in the 4th column)
- digit B (points to the 0 in the 3rd column)
- 0 (below the 4th column)

Two outputs

Sum: bit we write in
the current column

carry out: bit we carry
to the next column to the left

ex: 1 ← carry out
 1 1 1 1

+ 0 0 0 1

0 ← sum

Inputs

Outputs

A	B	CI	Sum	CO	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	★
1	1	1	1	1	★

Interesting Cases

A	B	CI	Sum	CO	
1	1	0	0	1	★
1	1	1	1	1	★

case 1:

$$\begin{array}{r}
 1 \\
 1 \\
 + 0 \\
 \hline
 0 \leftarrow \text{sum}
 \end{array}
 > 0 + \underset{\uparrow}{\text{CO}} : 1$$

case 2:

$$\begin{array}{r}
 1 \\
 1 \\
 + 1 \\
 \hline
 1 \leftarrow \text{sum}
 \end{array}
 > 0 + \underset{\uparrow}{\text{CO}} : 1$$

Ex: 8 + 3 as unsigned
4 bit numbers

$$\begin{array}{r} 1000 \\ + 0011 \\ \hline 1011 \quad (11) \end{array}$$

no overflow!

Ex: 3 + 3 as unsigned

4 bit numbers

$$\begin{array}{r} 11 \\ 0011 \\ + 0011 \\ \hline 0110 \quad (6) \end{array}$$

A B CI

1 1 1

1
0

$$0 + 1 = 1$$

CO: 1

Binary Subtraction

negate second operand
and then add.

Ex: $7 - 3$ as 4 bit
signed ints

$$7 - 3 = 7 + (-3)$$

3: 0011 $\rightarrow$ flip bits 1100 $\rightarrow$
add 1 to least sig bit $\rightarrow$

$$\begin{array}{r} -3 : 1101 \\ 7 : 0111 \\ + (-3) : 1101 \\ \hline 10100 \end{array}$$

10100
 $\uparrow$
carry out

Signed Extension

perform arith ops on numbers
with different sizes

unsigned: prepend 0

Signed: prepend the left most
bit.

Why? Most significant bit
is 1 for negative and 0
for positive (for signed values)

Signed Extension

Ex: 7, -7

Recall: $-7 \rightarrow 0111 \rightarrow 1000 \rightarrow 1001$

7: $0111 \rightarrow 0000\ 0111$

-7: $1001 \rightarrow 1111\ 1001$

Ex: $5 + (-5)$ as 8 bit numbers

5 : 0101

-5 : 1010 $\rightarrow$ 1011

		1	1	1	1		1	1	1
5 :	0	0	0	0		0	1	0	1

-5 :	1	1	1	1		1	0	1	1
------	---	---	---	---	--	---	---	---	---

1	0	0	0	0		0	0	0	0
---	---	---	---	---	--	---	---	---	---

↑
carry
out

Signed Extension: Example

0b 1011

$$-2^3 + 2^1 + 2^0 = -8 + 2 + 1 = -5$$

0b 1 1011

$$\begin{aligned} & -2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0 \\ & = -16 + 8 + 3 = -16 + 11 = -5 \end{aligned}$$

0b 11 1011

$$\begin{aligned} & -2^5 + 2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0 \\ & = -32 + 16 + 8 + 2 + 1 \\ & = -32 + 16 + 11 = -16 + 11 = -5 \end{aligned}$$

Floating Point Numbers

way to encode numbers in a format that can handle very large and very small values.

Based on Scientific notation where numbers are represented as a fraction and an exponent

Floating Point

Interpretation:

$$(-1)^n \times 2^{(e-127)} \times 1.5$$

↑

Controls sign

$$n = 0$$

↳ pos b/c

$$(-1)^0 = 1$$

$$n = 1$$

↳ neg b/c

$$(-1)^1 = -1$$

7

controls
magnitude

$$e - 127 < 0$$

↳ small number

$$e^{-127} > 0$$

↳ big number

7

controls
value

Floating Point

First bit : signed bit

Next 8 bits : exponent

The rest : significand/fraction

Ex:

sign bit exponent
1100 0001 1011 0100
0000 0000 0000 0000

(this is all one num. It's hard to fit on one line)

$$n = 1 \rightarrow (-1)^1 = -1 \text{ (negative)}$$

$$e = 2^7 + 2^1 + 2^0 = 128 + 2 + 1 = 131$$

$$e - 127 = 131 - 127 = 4 \rightarrow 2^4 = 16$$

$$s = 0 \cdot 2^{-1} + 2^{-2} + 2^{-3} + 0 \cdot 2^{-4}$$

$$+ 2^{-5} = \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^5} = .40625$$

$$(-1)^n \times 2^{(e-127)} \times 1.5 = (-1)(16)(1.40625) \\ = -22.5$$

Bitwise Operations

True = 1
False = 0

Ex: $4 \& 8 = 0$

$$\begin{array}{r} 0100 \\ 1000 \\ \hline 0000 \end{array}$$

Ex: $4 | 8 = 12$

$$\begin{array}{r} 0100 \\ 1000 \\ \hline 1100 \end{array}$$

$$2^2 + 2^3 = 4 + 8 = 12$$

Ex: ~ 4

$$0100 \rightarrow 1011 = 11$$

Recall: Right Shift

unsigned int right = 0x AAOO;

unsigned int rightShift = right >> 8;

* Each hexadecimal digit represents 4 bits.

* unsigned int = 4 bytes

* 1 byte = 8 bits

* $4 \times 8 = 32$ bits

* A = 10 (decimal)

unsigned int right = 0x AA00;

unsigned int rightShift = right >> 8;

A = 10

10 = 1010

32 bit number

0000 0000 0000 0000 1010 1010 0000 0000



shift right by
8 bits

0000 0000 0000 0000 0000 0000 1010 1010

So: 0x AA00 >> 8

= 0x 00AA

AND

Ex: Compute a & b , give
answer as hexadecimal

$$a = 7$$

$$b = 4$$

0111

0100

$$\hline 0100 \rightarrow 4 \rightarrow 0x4$$

Ex: $a \mid b$

OR

$$a = 7$$

$$b = 4$$

$$\begin{array}{r} 0111 \\ 0100 \\ \hline 0111 \end{array} \rightarrow 7 \rightarrow 0x7$$

Ex: compute $a \wedge b$

$$a = 7$$

XOR

$$b = 4$$

0 1 1 1

0 1 0 0

$$0011 \rightarrow 3 \rightarrow 0x3$$

Ex: compute $\sim a$

$$a = 7$$

$$0111 \rightarrow 1000 \rightarrow 8 \rightarrow 0x8$$

Bitops Code Example

Each char is 8 bit values

↳ 1 byte is the smallest data type in C

Note :

%02X → zero padded to minimum width of 2 digits (hexadecimal formatting)

Recall: Compute $\sim a$

$$a = 7$$

with 4 bits

$$0111 \rightarrow 1000 \rightarrow 8 \rightarrow 0x8$$

with 8 bits

$$\begin{array}{rcl} 7 & \rightarrow & 0000 \quad 0111 \\ \sim 7 & \rightarrow & 1111 \quad 1000 \\ & & \underbrace{\hspace{1cm}} \quad \underbrace{\hspace{1cm}} \\ & & F \quad \quad 8 \end{array}$$

$$\hookrightarrow 0xF8$$

Ex: Use bit ops to test
even / odd

Example: extract rightmost bit,
use 0x1 as mask

mask	→	0x1	→	0000	0001
& a	→	0x7	→	0000	0111
					0000 0001
					↑

when we AND with a mask, we can test whether the bits in question are 0 or 1

when we OR with a mask, we can set desired bits to 1

Want to test if two
signed numbers have opposite
signs.

What value should we use
as a mask?

highest order bit (determines
sign)

→ 1000 0000
→ 0x8 } our
mask

Exercise

value 1 $\rightarrow$ 4 bytes $\rightarrow$ 32 bits

value 2 $\rightarrow$ 4 bytes $\rightarrow$ 32 bits

mask $\rightarrow$ 0x80000000
4 bytes

result $\rightarrow$ either 0x80000000
or 0x0 depending
on sign of value

	A	A	B	B
&	1010	1010	1011	1011
	1111	1111	0000	0000

upper Byte	1010	1010	0000	0000
---------------	------	------	------	------

	A	A	B	B
&	1010	1010	1011	1011
	0000	0000	1111	1111

lower Byte	0000	0000	1011	1011
---------------	------	------	------	------

Upper
Byte
>> 8 1010 1010 0000 0000

shifted
Right 0000 0000 1010 1010

lower
Byte
<< 8 0000 0000 1011 1011

shifted
left 1011 1011 0000 0000

shifted
Right 0000 0000 1010 1010

OR
shifted
left 1011 1011 0000 0000

1011 1011 1010 1010

↳ 0x BBAA

Ex: Swap Nibbles

$$a = 0xAB$$

$$\text{upper NM} = 0xF0$$

$$\text{lower NM} = 0x0F$$

a	1010	1011
& upper NM	1111	0000
upper Nib	1010	0000
a	1010	1011
& lower NM	0000	1111
lower Nib	0000	1011

upper
Nib 1 0 1 0 0 0 0 0
 >> 4

shift+R 0 0 0 0 1 0 1 0

lower
Nib 0 0 0 0 1 0 1 1
 << 4

shift+L 1 0 1 1 0 0 0 0

shift+R 0 0 0 0 1 0 1 0

shift+L 1 0 1 1 0 0 0 0

1 0 1 1 1 0 1 0

↪ 0x BA

OR

Bit Shift : Revisited

$\ll$ (shift left): works the same for signed and unsigned

$\gg$ (shift right): works differently

↳ unsigned: can just prepend with zero

↳ signed: want to preserve the sign

Sign Extension

- * if val is negative,
pre-pend one to preserve
the sign
- * if val is negative, pre-pend
zero for the same reason

Bit Shift Example

unsigned:

0x FF000000 >> 12



0x 000FFF00

signed:

0x FFFFFFFF >> 12

0x FFFFFFFF

(prepend 1 because it's signed)

Byte layouts : Data Type Padding

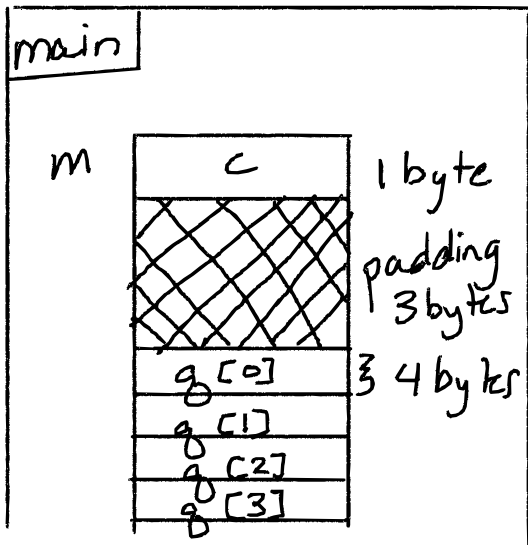
Data is always on a 1/4/8
byte boundary (depending on
type size)

Byte layouts : Data Type

Padding

Ex: struct mystery {
 char c; $\rightarrow$ 1 byte
 float q[4]; $\rightarrow 4 \times 4 =$
 16 bytes
}

Size of (mystery) = 20 bytes



main() {
 struct mystery m;
}

Byte Order

Big endian : highest order bit is stored at lowest memory addr

* common use case: networking protocols (human readable)

Little endian : lowest order bit is stored at lowest memory addr

* common use case: consumer CPUs
allows reinterpreting a pointer to a larger integer as a smaller integer without needing to offset memory

Byte Order

Ex:

x	AA
x+1	BB
x+2	CC
x+3	DD

Big
Endian

DD	x
CC	x+1
BB	x+2
AA	x+3

Little
Endian

prints:

0xAABBCCDD

prints:

0xDDCCBBAA

Ex: Dinner Options

water	0x01	0000 0001
wine	0x02	0000 0010
dinner roll	0x04	0000 0100
Salad	0x08	0000 1000
Soup	0x10	0001 0000
main	0x20	0010 0000
dessert	0x40	0100 0000
Coffee	0x80	1000 0000

Ex: Dinner Options

Options	bit field
water, wine, salad, main, coffee	0b10101011
water, salad, soup, main, dessert	0b01111001
wine, main, dessert, coffee	0b11100010
water, salad, dessert	0b01001001

water | wine | salad
 0x1 | 0x2 | 0x08

OR
 0000 0001
 0000 0010

 0000 0011
 OR
 0000 1000

 0000 1011

0000 1011
 OR
 0010 0000

 0010 1011
 OR
 1000 0000

 1010 1011

Ex: Dinner Options

Bit field	Options
0xAE = 0b10101110	coffee, main, salad, roll, wine
0x31 = 0b00110001	main, soup, water
0x9B = 0b10011011	coffee, soup, salad, wine, water
0x74 = 0b01110100	dessert, main, soup, roll

$$A = 10$$

$$E = 14$$

$$\begin{aligned} 2^2 + 2^1 + 2^0 \\ = 4 + 2 + 1 \end{aligned}$$

Ex: Dinner Options

1) set preferences to water,
wine, dessert

char options = water | wine |
dessert

2) code that tests whether

Someone chose the main option

```
if (options & main == main) {  
    // do stuff
```

```
}
```

Ex: Dinner Options

3) code that tests whether
someone chose wine AND
dessert options

```
wine-dessert = wine & dessert;
```

```
if (options & wine-dessert ==  
    wine-dessert) {
```

```
    // do stuff
```

```
}
```

Ex: File Permissions Use

Bit Fields

* 3 options for file/dir

permissions: read, write, execute

- r w -	r - -	r - -
<u> </u>	<u> </u>	<u> </u>
user	group	world
↓	↓	↓
can	can	can
read/ write	read	read

Note: use chmod to set
file permissions

Ex: File Permissions Use

Bit Fields

Note: each octal digit directly maps to a 3-bit binary value

Ex: -rw-r--r-- in octal

110 100 100

6 4 4 → 0644

↓
denotes
octal number

Ex: -rwxr-x--x

111 101 001

7 5 1 → 0751