CSCI2467: Systems Programming Concepts

Slideset 3: Integer values and arithmetic (CS:APP 2.2, 2.3)

Course Instructors:

Matthew Toups
Caitlin Boyce

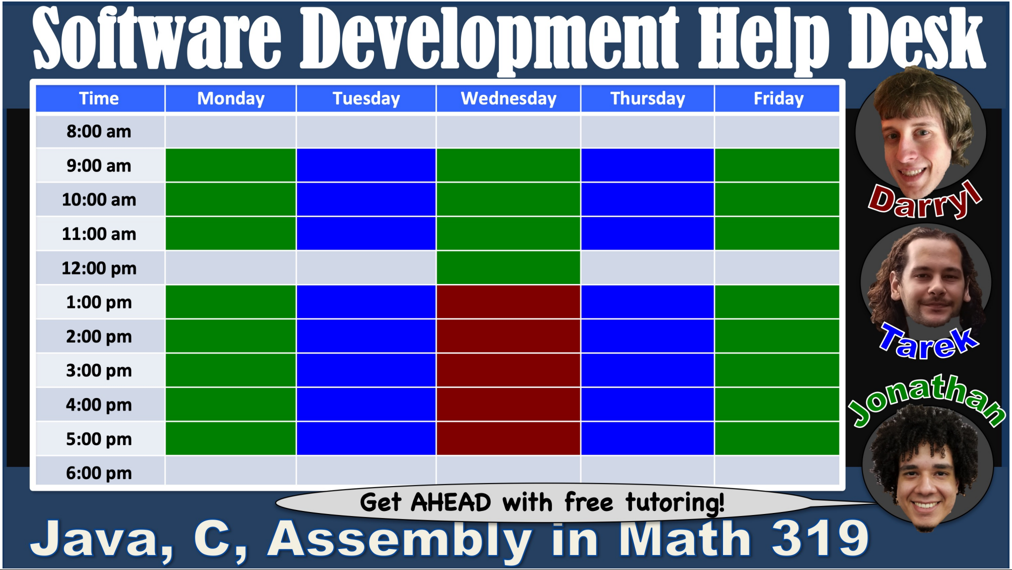
Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY of NEW ORLEANS

CSCI Help Desk



Datalab tips

- Check correctness with `./btest`
- Test for illegal operators with `./dlc bits.c`
- Get final score with `./driver.pl`

  (all of this is on page 3 of writeup; keep it close by)

Autolab

- Only submit `bits.c` to Autolab (not a tar file)
- You can use the Autolab website, or the `autolab` command-line interface, to submit
- Can re-submit, most recent submission is counted
- Scoreboard! points / ops

datalab scoreboard

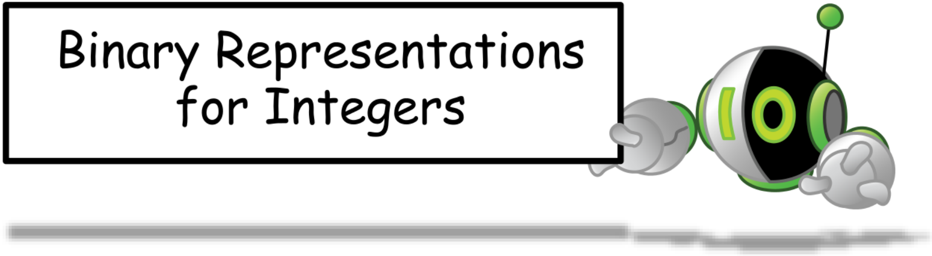
Systems Programming Concepts (2019-Spring) » Data Lab » **Scoreboard**

Datalab scoreboard!

RANK	NICKNAME	VERSION	TIME	TOTAL POINTS	TOTAL OPS	BITOR OPS	BITAND OPS	BITXOR OPS	ISNOTEQUAL OPS	COPYLSB OPS	SPECBITS OPS
1	m. toups	17	2019-02-02 12:32:09	10	32	7	4	7	3	2	3
2	Battousai	3	2019-02-01 21:50:26	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Overview

How we got here



In the early days of computing, designers made computers express numbers using **unsigned binary**.

In the early days of computing, designers made

To include negative numbers, designers came up with **sign magnitude**.

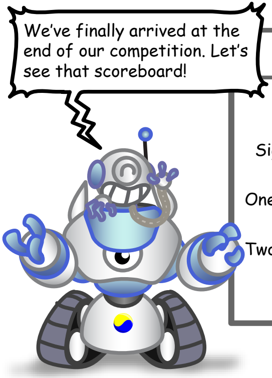
Then designers created **ones' complement**.

Finally, designers developed **two's complement**.

Finally, designers developed **two's complement**.

Comparing Integer Representations

The Thrilling Conclusion!



	Negation?	One Zero?	Zero = 0000 0000	Continuous?	Monotonically Increasing?
Unsigned		✓	✓	✓	✓
Sign Magnitude	✓		✓		
One's Complement	✓		✓		✓
Two's Complement	✓	✓	✓		✓
Bias	✓	✓		✓	✓

Well, well! It appears we have a three-way tie among Unsigned, Two's Complement, and Bias! We can certainly give each of our winners a prize, though!

CS Illustrated.berkeley.edu

Encoding Integer values

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Signed

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Change: Sign bit!

$$B2T(X) = {\color{red}-x_{w-1} \cdot 2^{w-1}} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

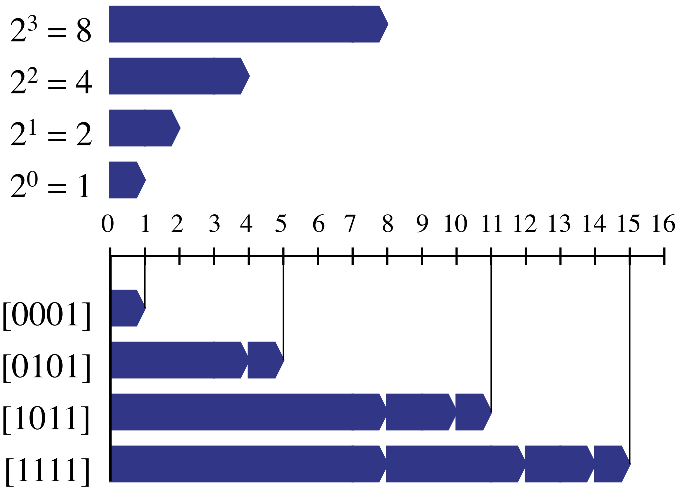
Change: Sign bit!

- Example using `short int` in C (2 bytes):

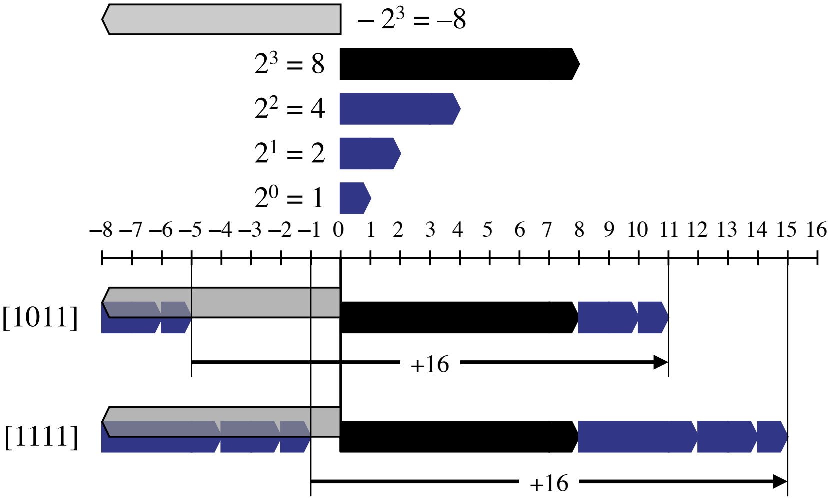
Decimal	Hex	Binary
2467	09A3	00001001 10100011
-2467	F65D	11110110 01011101

- This system of numbering is called *Two's complement*
- Sign bit indicates sign
  - 0 for non-negative
  - 1 for negative

Unsigned Integers



Signed Integers



Back to the 2's complement encoding example

```
short int x=   2467:  00001001 10100011
short int y=  -2467:  11110110 01011101
```

Weight	2467		-2467	
1	1	1	1	1
2	1	2	0	0
4	0	0	1	4
8	0	0	1	8
16	0	0	1	16
32	1	32	0	0
64	0	0	1	64
128	1	128	0	0
256	1	256	0	0
512	0	0	1	512
1024	0	0	1	1024
2048	1	2048	0	0
4096	0	0	1	4096
8192	0	0	1	8192
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum:		2467		-2467

Numeric Ranges

- Unsigned values
  - UMin $= 0$

    000 … 0
  - UMax $= 2^w - 1$

    111 … 1
- Two's complement values
  - TMin $= -2^{w-1}$

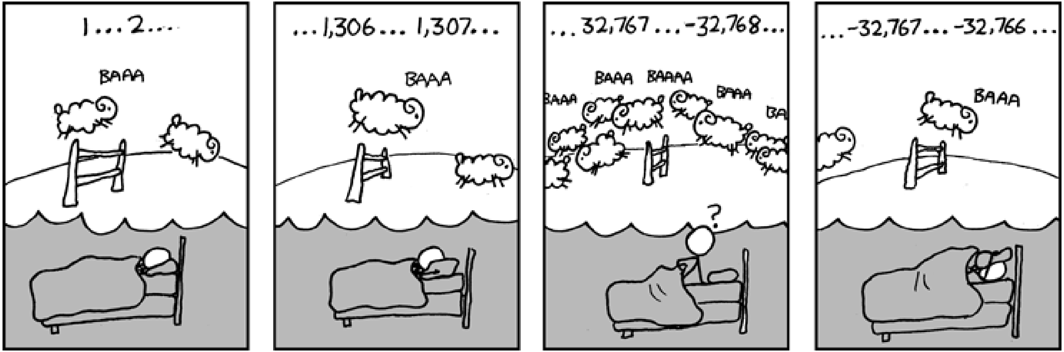
    100…0
  - TMax $= 2^{w-1} - 1$

    011…1

Values for $w = 16$ (`short int`)

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

16-bit sheep counter



Source: xkcd.com

Values for Different Word Sizes

	W (bits)			
	8	16	32	64
UMax	255	65535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations:

  $|TMin| = TMax + 1$

  (Asymmetric range)
- $UMax = 2 * TMax + 1$

- C Programming

  #include <limits.h>

  Defines constants:

  INT_MAX

  INT_MIN

  LONG_MAX

  ULONG_MAX

Data Types in C

	Size in Bytes		
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	-	-	10/16
pointer	4	8	8

Unsigned & Signed Numeric Values

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **⇒ Can Invert Mappings**
  - U2B(*x*) = $B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - T2B(*x*) = $B2T^{-1}(x)$
    - Bit pattern for two's comp integer

Overview

Real-world problems!

Why does any of this matter? Rocket science
(Fatal bug in Patriot missile, Ariane-5 explosion)

What's 77.1 x 850? Don't ask Excel 2007

65,535 = the Number of the Beast

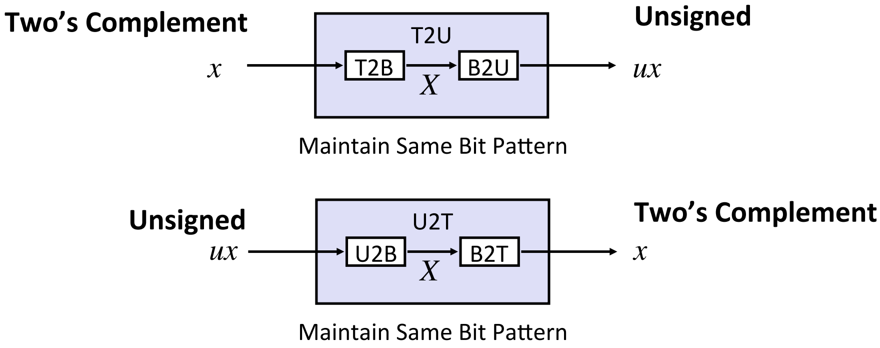
26 Sep 2007 at 17:45, Dan Goodin

A Microsoft manager has confirmed the existence of a serious bug that could give prog
number crunchers a failing grade when relying on the latest version of Excel to do basi

The flaw presents itself when multiplying two numbers whose product equals 65,535. F
favorite calculator and multiply 850 by 77.1. Through the magic of zeros and ones, you
answer of 65,535. Those using the Excel 2007, however, will be told the total is 100,00
similarly fails when multiplying 11 other sets of numbers, including 5.1*12850, 10.2*642
20.4*3212.5, according to this blog post from Microsoft manager David Gainer.

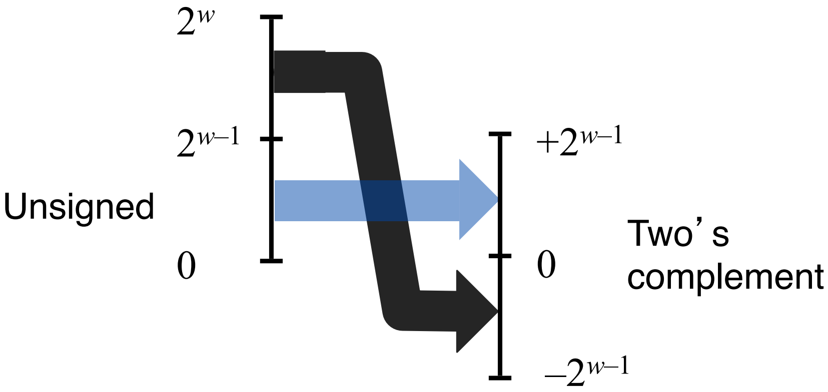
He stressed that the bug, which was introduced when Microsoft made changes to the E

Mapping Between Signed & Unsigned

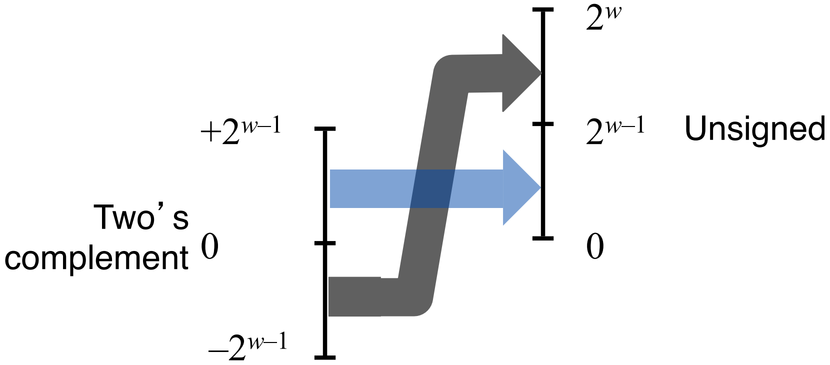


- **Mappings between unsigned and two's complement numbers:**
  Keep bit representations and reinterpret

Unsigned → Signed (U2T)



Signed → unsigned (T2U)



Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

→ T2U →

← U2T ←

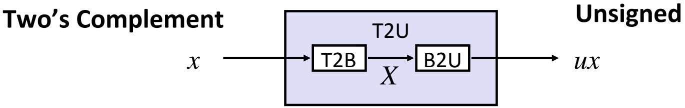
Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

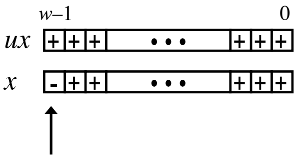
=

+/- 16

Relation between Signed & Unsigned



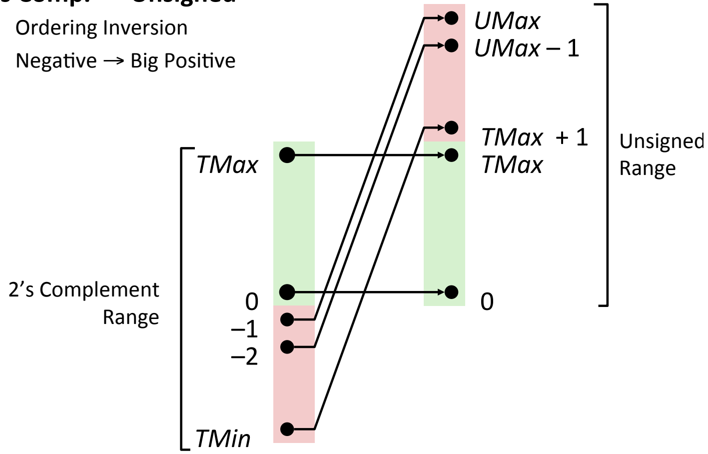
Maintain Same Bit Pattern



Large negative weight
becomes
Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Casting surprises!

Expression Evaluation:

- If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
- includes comparison operations > < == <= >=
- When W=32: TMIN= -2,147,483,648 TMAX= 2,147,483,647

Constant1	Relation	Constant2	Evaluation
0		0U	
0	==	0U	unsigned
-1		0	
-1	<	0	signed
-1		0U	
-1	>	0U	unsigned
2147483647		-2147483647-1	
2147483647	>	-2147483647-1	signed
2147483647U		-2147483647-1	
2147483647U	<	-2147483647-1	unsigned

Casting Signed $\leftrightarrow$ Unsigned: Basic rules

- Bit pattern is maintained
- ... but reinterpreted
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing signed and unsigned int:
  `int` is cast to `unsigned` ! (implicitly)

Pitfalls of unsigned

Don't use `unsigned` without understanding implications:
It is easy to make mistakes!

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```

Pitfalls of unsigned

Don't use `unsigned` without understanding implications:
It is easy to make mistakes!

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
      a[i] += a[i+1];
```

Can be very subtle:

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
      . . .
```

Counting down with unsigned

A better way to use loop index:

```
unsigned i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

C Standard guarantees that unsigned addition will behave like modular arithmetic: $0 - 1 \rightarrow UMax$

Modular arithmetic is useful in many situations.

Why even use `unsigned` then?

- *Do* use when performing modular arithmetic (multiprecision arithmetic)
- *Do* use when using bits to represent sets

  Logical right shift $\rightarrow$ no sign extension

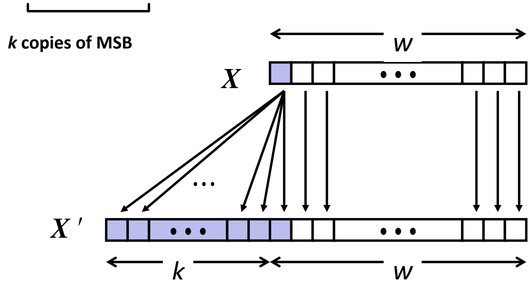
Java?

no `unsigned`! Everything is signed.

introduces $>>>$ for logical shift ($>>$ is arithmetic shift)

Sign extension

- **Task:**
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - Make *k* copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign extension example

```
short int x  = 2467;
int      ix  = (int) x;
short int y  = -2467;
int      iy  = (int) y;
```

	Decimal	Hex	Binary
x	2467	09 A3	00001001 10100011
ix	2467	00 00 09 A3	00000000 00000000 00001001 10100011
y	-2467	F6 5D	11110110 01011101
iy	-2467	FF FF F6 5D	11111111 11111111 11110110 01011101

Converting from smaller to larger integer data type:

C automatically performs sign extension

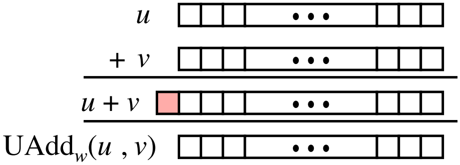
Overview

Unsigned addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

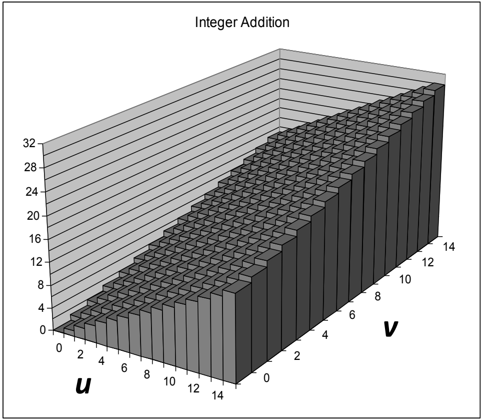
$$s \;=\; \text{UAdd}_w(u, v) \;=\; u + v \mod 2^w$$

Visualizing (Mathematical) Integer addition

Integer Addition

- 4-bit integers *u*, *v*
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with *u* and *v*
- Forms planar surface

$\text{Add}_4(u, v)$

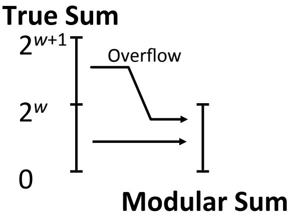


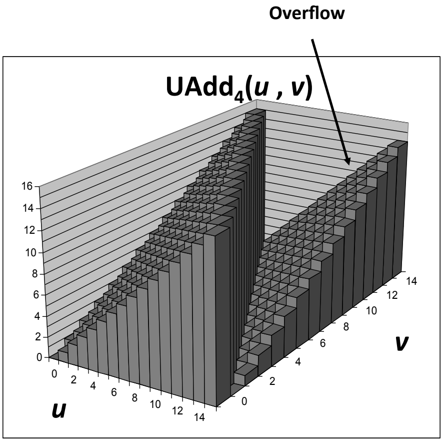
CS:APP3e **Figure 2.21**: With a 4-bit word size, the sum could require 5 bits.

Visualizing Unsigned addition

Wraps Around

- If true sum ≥ 2^w
- At most once





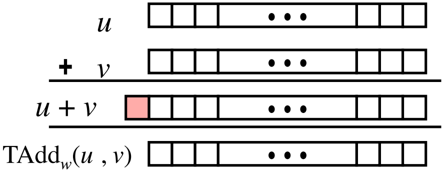
CS:APP3e **Figure 2.23**: With a 4-bit word size, addition is performed modulo 16.

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

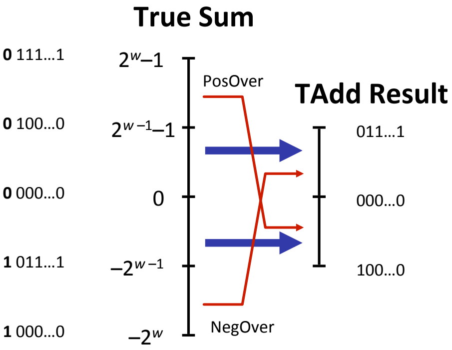
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

TAdd Overflow

Functionality

- True sum requires w+1 bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

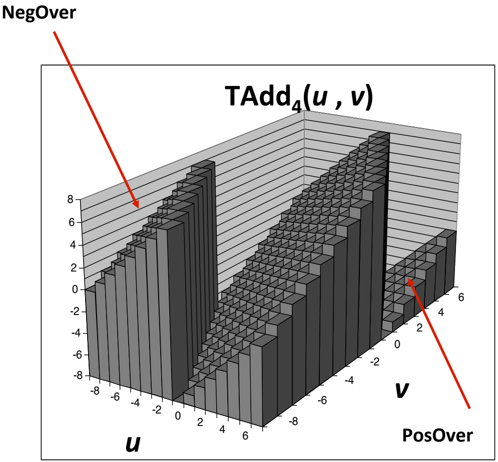


Values

- 4-bit two's comp.
- Range from -8 to +7

Wraps Around

- If sum $\geq 2^{w-1}$
  - Becomes negative
  - At most once
- If sum $< -2^{w-1}$
  - Becomes positive
  - At most once



Multiplication

Goal: Computing Product of w-bit numbers x, y

- Either signed or unsigned

But, exact results can be bigger than w bits

- Unsigned: up to $2w$ bits
  - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
- Two's complement min (negative): Up to $2w$-1 bits
  - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
- Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
  - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

So, maintaining exact results…

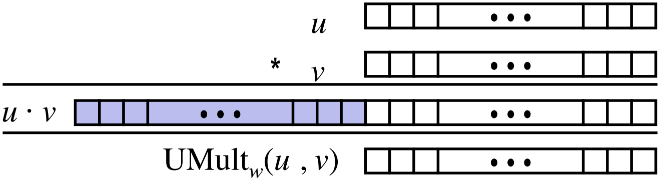
- would need to keep expanding word size with each product computed
- is done in software, if needed
  - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

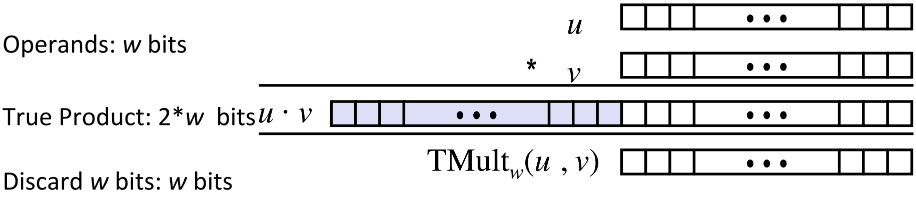
Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
- **Implements Modular Arithmetic**

  $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$

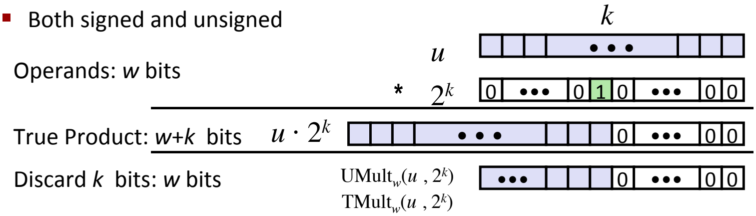
Signed Multiplication in C



- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Power-of-2 Multiply with Shift

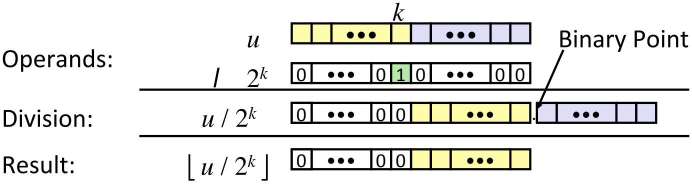
- **Operation**
  - `u << k` gives `u * 2`^{k}
  - Both signed and unsigned



- **Examples**
  - `u << 3 == u * 8`
  - `(u << 5) – (u << 3)== u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives ⌊ **u / 2^k** ⌋
  - Uses logical shift



	Division	Computed	Hex	Binary
x	2467	2467	09 A3	00001001 10100011
x >> 1	1233.5	1233	04 D1	00000100 11010001
x >> 4	154.1875	154	00 9A	00000000 10011010
x >> 8	9.63671875	9	00 09	00000000 00001001

Summary of Arithmetic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

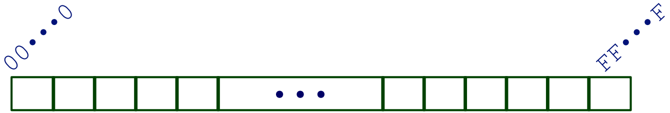
C `int` Puzzles!

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

If...	true for all values, or false?
$x < 0$	$\Rightarrow (x * 2) < 0$
	$ux \geq 0$
$x \& 7 == 7$	$\Rightarrow (x << 30) < 0$
	$ux > -1$
$x > y$	$\Rightarrow -x < -y$
	$x * x \geq 0$
$x > 0 \ \&\& \ y > 0$	$\Rightarrow x + y > 0$
$x \geq 0$	$\Rightarrow -x \leq 0$
$x \leq 0$	$\Rightarrow -x \geq 0$
	$(x \mid -x) >> 31 == -1$
	$ux >> 3 == ux/8$
	$x >> 3 == x/8$
	$x \ \& \ (x-1) \ != \ 0$

Overview

Byte-oriented memory organization

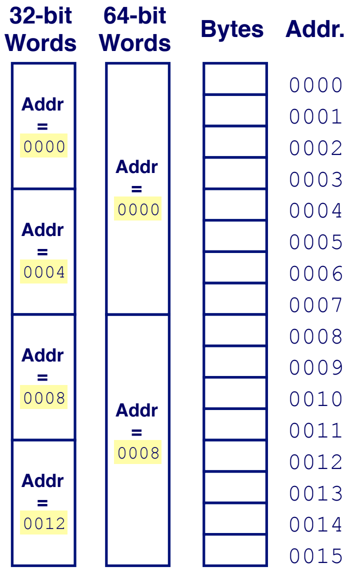


- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- **Note: system provides private address spaces to each “process”**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Word-oriented memory organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



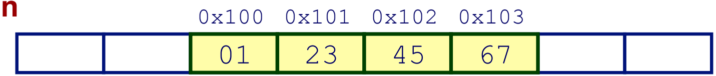
Byte ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

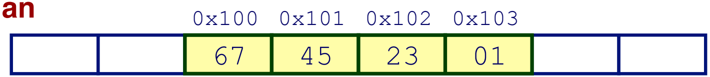
Byte ordering example

- **Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

Big Endian



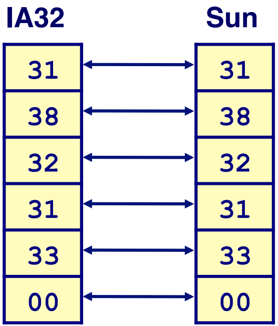
Little Endian



Representing strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue



Code security example

Similar to FreeBSD's implementation of getpeername()[1]

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/*Copy at most maxlen bytes from kernel region to user buffer*/
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528
void getstuff() {
     char mybuf[MSIZE];
     copy_from_kernel(mybuf, MSIZE);
     printf("%s\n", mybuf);
}
```

[1] See CVE-2002-0973 for more info on this real-world security vulnerability.

Malicious usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

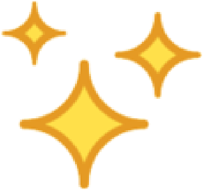
```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/*Copy at most maxlen bytes from kernel region to user buffer*/
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528
void getstuff() {
     char mybuf[MSIZE];
     copy_from_kernel(mybuf, -MSIZE);
     printf("%s\n", mybuf);
}
```

Overview

XOR is magic



`XOR (^)` has magic powers!

```
temp = a;
a = b;
b = temp;
```

XOR is magic

Swap a and b without a temporary variable!

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

XOR is magic

How the magic works: (see page 54 in CS:APP text)

```
x = x ^ y ; // x == A, y == B
y = x ^ y ; // x == A ^ B, y == B
 // y == (A ^ B) ^ B == A ^ (B ^ B)
 //    == A ^ 0
x = x ^ y ; // x == A ^ B, y == A
 // x == ( A ^ B ) ^ A
 //    == ( A ^ A ) ^ B
 //    == 0 ^ B
 //    == B
```

XOR magic is crucial

If we have 3 "data" bits and 1 "parity" bit...