

Bits, Bytes, and Integers

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

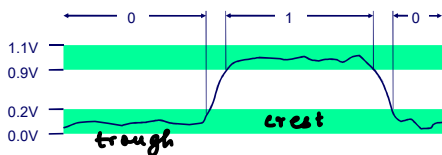
bits: high voltage or low voltage
(1) (0)
ternary elements do exist

unsigned ints (uints) - non-negative

Everything is bits

instructions are coded in bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
 - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
 - Easy to store with bistable elements
 - Reliably transmitted on noisy and inaccurate wires



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

For example, can count in binary

- Base 2 Number Representation
 - Represent 15213_{10} as 11101101101101_2
 - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
 - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

There are 16 pairs of 2 hex chars between 0000 and 0010.

Encoding Byte Values

- Byte = 8 bits
 - Binary 00000000_2 to 11111111_2
 - Decimal: 0_{10} to 255_{10}
 - Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write FA1D37B in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

	Hex	Decimal	Binary
0	0	0000	
1	1	0001	
2	2	0010	
3	3	0011	
4	4	0100	
5	5	0101	
6	6	0110	
7	7	0111	
8	8	1000	
9	9	1001	
A	10	1010	
B	11	1011	
C	12	1100	
D	13	1101	
E	14	1110	
F	15	1111	

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Example Data Representations

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

0000 represents everything from 0000 - 000f

1 hex char = 4 binary bits
1 byte = 2 hex chars

Understanding binary and hex

DECIMAL	1	2	3	4
base 10 →	10^3	10^2	10^1	10^0
	1000	100	10	1
<hr/>				
1024 = 1024				
<hr/>				

BINARY	1	0	1	0
base 2 →	2^3	2^2	2^1	2^0
	8	4	2	1
<hr/>				
1010 = 10				
<hr/>				

HEX	B	A	5	5
	11	10	5	5
base 2 →	16^3	16^2	16^1	16^0
	4096	256	16	1
<hr/>				
BA55 = 47701				
<hr/>				

$$\begin{array}{|c|c|c|} \hline \text{F} & \text{F} & \text{F} \\ \hline \text{R} & \text{G} & \text{B} \\ \hline \end{array} = (255, 255, 255)$$

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

7

Boolean Algebra

- Developed by George Boole in 19th Century
 - Algebraic representation of logic
 - Encode "True" as 1 and "False" as 0

And &

■ A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

ampers and

Or |

■ A|B = 1 when either A=1 or B=1

	0	1
0	0	1
1	1	1

pipe

Not ~

■ ~A = 1 when A=0

~	0	1
0	1	
1	0	

tilde

Exclusive-Or (Xor) ^

■ A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

caret

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

8

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise
- | | | |
|---------------|--------------|-------------------|
| 01101001 | 01101001 | 01101001 |
| $\&$ 01010101 | $ $ 01010101 | \wedge 01010101 |
| 01000001 | 01111101 | 00111100 |
| | | \sim 01010101 |
| | | 10101010 |
- All of the Properties of Boolean Algebra Apply
- bitwise calculation*

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

9

Example: Representing & Manipulating Sets

- Representation
 - Width w bit vector represents subsets of $\{0, \dots, w-1\}$
 - $a_j = 1$ if $j \in A$
- | | | | |
|----------|------------------|-----|-----|
| 01101001 | $\{0, 3, 5, 6\}$ | $-$ | A |
| 76543210 | | | |
| 01010101 | $\{0, 2, 4, 6\}$ | $-$ | B |
| 76543210 | | | |
- Operations (A, B)
 - $\&$ Intersection: 01000001 $\{0, 6\}$
 - $|$ Union: 01111101 $\{0, 2, 3, 4, 5, 6\}$
 - \wedge Symmetric difference: 00111100 $\{2, 3, 4, 5\}$
 - \sim Complement: 10101010 $\{1, 3, 5, 7\}$

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

10

Bit-Level Operations in C

- Operations $\&$, $|$, \sim , \wedge Available in C
 - Apply to any "integral" data type
 - long, int, short, char, unsigned
 - View arguments as bit vectors
 - Arguments applied bit-wise
 - Examples (Char data type)
 - $\sim 0x41 \rightarrow 0xBE$ *HEX*
 - $\sim 01000001_2 \rightarrow 10111110_2$ *BINARY*
 - $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \& 0x55 \rightarrow 0x41$
 - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
 - $0x69 | 0x55 \rightarrow 0x7D$
 - $01101001_2 | 01010101_2 \rightarrow 01111101_2$
- convert to binary, perform bitwise*

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

11

Contrast: Logic Operations in C

- Contrast to Logical Operators
 - $\&\&$, $||$, $!$
 - View 0 as "False"
 - Anything nonzero as "True"
 - Always return 0 or 1
 - Early termination
 - Examples (char data type)
 - $!0x41 \rightarrow 0x00$
 - $!0x00 \rightarrow 0x01$
 - $!!0x41 \rightarrow 0x01$
 - $0x69 \&\& 0x55 \rightarrow 0x01$
 - $0x69 || 0x55 \rightarrow 0x01$
 - $p \&\& *p$ (avoids null pointer access)
- Return T/F instead of the result of an operation*

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

12

Contrast: Logic Operations in C

Contrast to Logical Operators

- `&&`, `||`, `!`
 - View 0 as "False"
 - Anything nonzero
 - Always returns 0 or 1
 - Early exit
- Example
 - `!0x41`
 - `!0x00`
 - `!!0x41`
 - `0x69 & 0x55 → 0x01`
 - `0x69 || 0x55 → 0x01`
 - `p && *p` (avoids null pointer access)

Watch out for `&&` vs. `&` (and `||` vs. `|`)... one of the more common oopsies in C programming

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

13

Shift Operations

Left shift by y positions
Add 0.

Left Shift: `x << y`

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

Argument x	01100010
<code><< 3</code>	00010000
Log. <code>>> 2</code>	00011000
Arith. <code>>> 2</code>	00011000

Right Shift: `x >> y`

- Shift bit-vector x right y positions
 - Throw away extra bits on right
- Logical shift
 - Fill with 0's on left
- Arithmetic shift
 - Replicate most significant bit on left

Argument x	10100010
<code><< 3</code>	00010000
Log. <code>>> 2</code>	00101000
Arith. <code>>> 2</code>	11101000

Undefined Behavior

- Shift amount < 0 or \geq word size

MSB - leftmost
LSB - rightmost

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

14

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings
- Summary

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

15

Encoding Integers

Unsigned

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

Two's Complement

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

short int $x = 15213$;
short int $y = -15213$;

Sign Bit

C short 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

Sign Bit

- For 2's complement, most significant bit indicates sign
 - 0 for nonnegative
 - 1 for negative

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

16

Two-complement Encoding Example (Cont.)

$x = 15213$: 00111011 01101101
 $y = -15213$: 11000100 10010011

Weight	15213	-15213
1	1	1
2	0	1
4	1	0
8	1	0
16	0	1
32	1	0
64	1	0
128	0	1
256	1	0
512	1	0
1024	0	1
2048	1	0
4096	1	0
8192	1	0
16384	0	1
-32768	0	1
Sum	15213	-15213

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

17

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
- Other Values
 - Minus 1
111...1

Values for $W = 16$

	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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

18

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-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>
- Declares constants, e.g.,
 - ULONG_MAX
 - LONG_MAX
 - LONG_MIN
- Values platform specific

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

19

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 complement integer

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

20

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

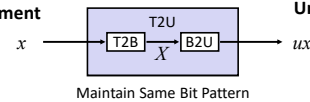
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

21

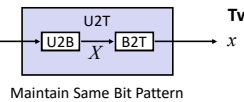
Mapping Between Signed & Unsigned

Two's Complement



Unsigned

Unsigned



Two's Complement

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

22

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

23

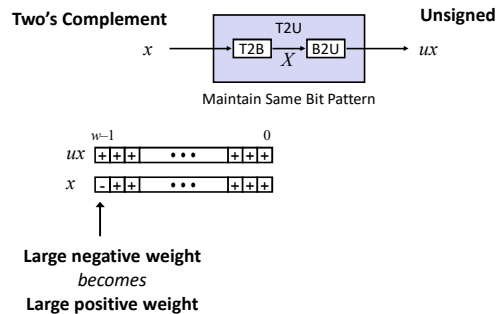
Mapping Signed ↔ Unsigned

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

24

Relation between Signed & Unsigned

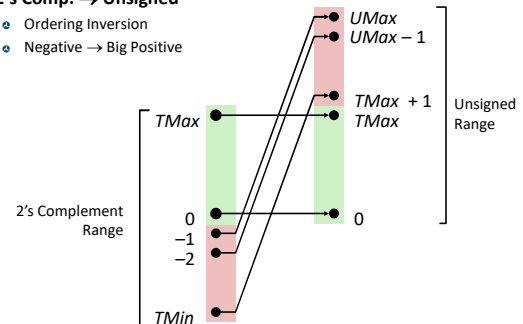


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

25

Conversion Visualized

- 2's Comp. \rightarrow Unsigned
 - Ordering Inversion
 - Negative \rightarrow Big Positive



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

26

Signed vs. Unsigned in C

- Constants**
 - By default are considered to be signed integers
 - Unsigned if have "U" as suffix
`0U, 4294967295U`
- 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;
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

27

Casting Surprises

- Expression Evaluation**
 - If there is a mix of unsigned and signed in single expression, **signed values implicitly cast to unsigned**
 - Including comparison operations `<`, `>`, `==`, `<=`, `>=`
 - Examples for $W = 32$: **$TMin = -2,147,483,648$, $TMax = 2,147,483,647$**
- | Constant ₁ | Constant ₂ | Relation | Evaluation |
|-----------------------|-----------------------|----------|------------|
| 0 | 0U | == | unsigned |
| -1 | 0 | < | signed |
| -1 | 0U | > | unsigned |
| 2147483647 | -2147483647-1 | > | signed |
| 2147483647U | -2147483647-1 | < | unsigned |
| -1 | -2 | > | signed |
| (unsigned)-1 | -2 | > | unsigned |
| 2147483647 | 2147483648U | < | unsigned |
| 2147483647 | (int) 2147483648U | > | signed |

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

28

Summary

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!!

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

29

Today: Bits, Bytes, and Integers

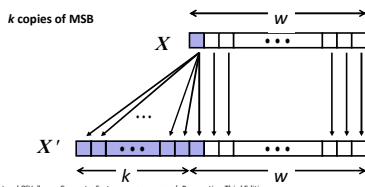
- Representing information as bits
- Bit-level manipulations
- Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

30

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:
 - $X' = X_{w-1}, \dots, X_{w-1}, X_{w-1}, X_{w-2}, \dots, X_0$



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

31

Sign Extension Example

```
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

32

Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
 - Unsigned: zeros added
 - Signed: sign extension
 - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
 - Unsigned/signed: bits are truncated
 - Result reinterpreted
 - Unsigned: mod operation
 - Signed: similar to mod
 - For small numbers yields expected behavior

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

33

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

34

Unsigned Addition

Operands: w bits

$$\begin{array}{r} u \\ + v \\ \hline \end{array}$$

True Sum: $w+1$ bits

$$u + v$$

Discard Carry: w bits

$$UAdd_w(u, v)$$

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

$$s = UAdd_w(u, v) = u + v \bmod 2^w$$

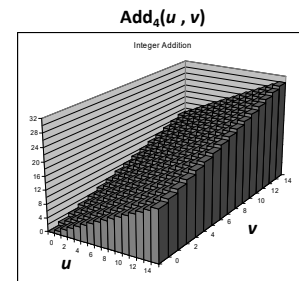
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

35

Visualizing (Mathematical) Integer Addition

Integer Addition

- 4-bit integers u, v
- Compute true sum $Add_4(u, v)$
- Values increase linearly with u and v
- Forms planar surface



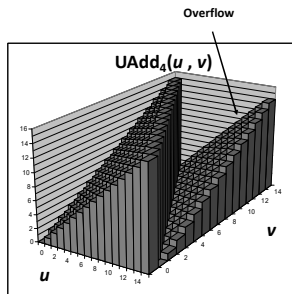
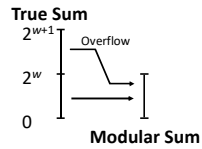
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

36

Visualizing Unsigned Addition

Wraps Around

- If true sum $\geq 2^w$
- At most once

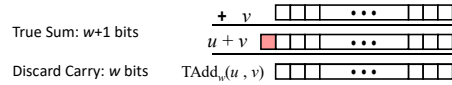


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

37

Two's Complement Addition

Operands: 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
```

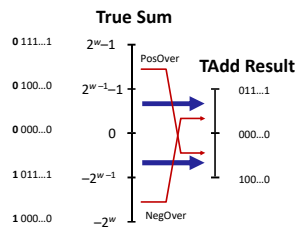
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

38

TAdd Overflow

Functionality

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



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

39

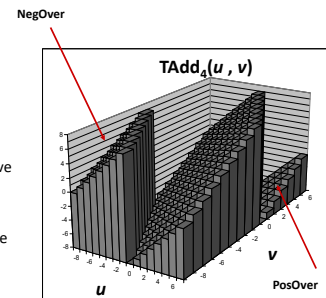
Visualizing 2's Complement Addition

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



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

40

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

So, maintaining exact results...

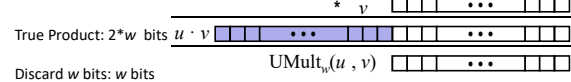
- would need to keep expanding word size with each product computed
- is done in software, if needed
 - e.g., by "arbitrary precision" arithmetic packages

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

41

Unsigned Multiplication in C

Operands: w bits



Standard Multiplication Function

- Ignores high order w bits

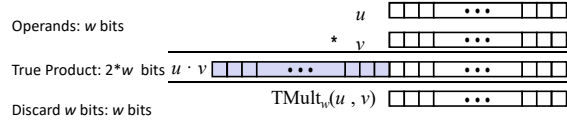
Implements Modular Arithmetic

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

42

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

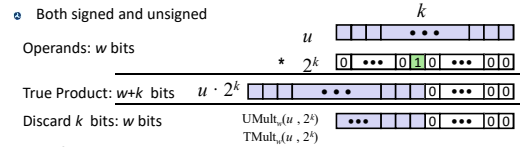
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

43

Power-of-2 Multiply with Shift

Operation

- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned



Examples

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- Most machines shift and add faster than multiply
 - Compiler generates this code automatically

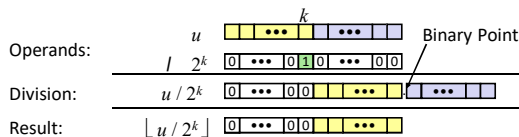
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

44

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
$x \gg 1$	7606.5	7606	1D B6	00011101 10110110
$x \gg 4$	950.8125	950	03 B6	00000011 10110110
$x \gg 8$	59.4257813	59	00 3B	00000000 00111011

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

45

Today: Bits, Bytes, and Integers

Representing information as bits

Bit-level manipulations

Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

46

Arithmetic: Basic 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)

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

47

Why Should I Use Unsigned?

Don't use without understanding implications

- 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)
    ...
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

48

Counting Down with Unsigned

- Proper way to use unsigned as loop index


```
unsigned i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```
- See Robert Seacord, *Secure Coding in C and C++*
 - C Standard guarantees that unsigned addition will behave like modular arithmetic
 - $0 - 1 \rightarrow UMax$
- Even better


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

 - Data type `size_t` defined as unsigned value with length = word size
 - Code will work even if `cnt = UMax`
 - What if `cnt` is signed and `< 0`?

Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

49

Why Should I Use Unsigned? (cont.)

- Do Use When Performing Modular Arithmetic
 - Multiprecision arithmetic
- Do Use When Using Bits to Represent Sets
 - Logical right shift, no sign extension

Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

50

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

51

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

Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

52

Machine Words

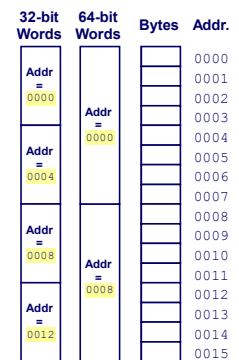
- Any given computer has a "Word Size"
 - Nominal size of integer-valued data
 - and of addresses
 - Until recently, most machines used 32 bits (4 bytes) as word size
 - Limits addresses to 4GB (2^{32} bytes)
 - Increasingly, machines have 64-bit word size
 - Potentially, could have 18 EB (exabytes) of addressable memory
 - That's 18.4×10^{18}
 - Machines still support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

53

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)



Notes adapted from Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

54

Example Data Representations

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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

55

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

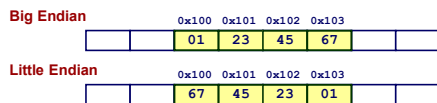
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

56

Byte Ordering Example

Example

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



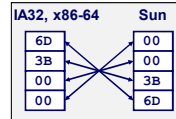
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

57

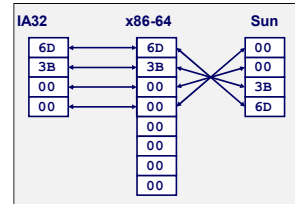
Representing Integers

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

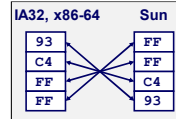
int A = 15213;



long int C = 15213;



int B = -15213;



Two's complement representation

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

58

Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

59

show_bytes Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

60

Representing Pointers

```
int B = -15213;
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

Different compilers & machines assign different locations to objects
Even get different results each time run program

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

61

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

IA32	Sun
31	31
38	38
32	32
31	31
33	33
00	00

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

62

Integer C Puzzles

Initialization

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

- $x < 0 \rightarrow ((x*2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \rightarrow (x \ll 30) < 0$
- $ux > -1$
- $x > y \rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&\& y > 0 \rightarrow x + y > 0$
- $x \geq 0 \rightarrow -x \leq 0$
- $x \leq 0 \rightarrow -x \geq 0$
- $(x|-x) \gg 31 == -1$
- $ux \gg 3 == ux/8$
- $x \gg 3 == x/8$
- $x \& (x-1) != 0$

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

63