Bits, Bytes, and Integers

CSE 238/2038/2138: Systems Programming

Instructor:

Fatma CORUT ERGİN

Slides adapted from Bryant & O'Hallaron's slides

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

Computer is a binary digital system

Binary (base 2) system:

- has two states: 0 and 1

- **Basic unit of information is the *binary digit*, or *bit*.**
- **Values with more than two states require multiple bits.**
  - A collection of two bits has four possible states: 00, 01, 10, 11
  - A collection of three bits has eight possible states: 000, 001, 010, 011, 100, 101, 110, 111
  - *A collection of n bits has 2^n possible states.*

What kinds of data do we need to represent?

- **Everything is bits**
  - Numbers – signed, unsigned, integers, floating point, complex, rational, irrational, …
  - Text – characters, strings, …
  - Images – pixels, colors, shapes, …
  - Sound
  - Logical – true, false
  - Instructions
  - …
- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Hexadecimal Notation

- **It is often convenient**

  to write binary (base-2) numbers
  as hexadecimal (base-16) numbers instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1's and 0's

- **C notation for hexadecimal numbers:**
  - `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

This is not a new machine representation, just a convenient way to write the number.

Converting from Binary to Hexadecimal

- **Every four bits is a hex digit.**
  - start grouping from right-hand side

011	1010	1000	1111	0100	1101	0111
3	**A**	**8**	**F**	**4**	**D**	**7**

Sizes of Data Types (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

1 byte = 8 bits

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

Logical Operations (Boolean Algebra)

- **Operations on logical TRUE or FALSE**
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	AND A & B	OR A | B	XOR A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

A	NOT ~ A
0	1
1	0

- **View *n*-bit number as a collection of *n* logical values**
  - operation applied to each bit independently

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 A={ 0, 3, 5, 6 }

      76543210

    - 01010101 A={ 0, 2, 4, 6 }

      76543210

- **Operations**
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - **`long, int, short, char, unsigned`**
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples**
  - `~ 0x41` → `0xBE`
    - `~` $0100\ 0001_2$ → $1011\ 1110_2$
  - `~ 0x00` → `0xFF`
    - `~` $0000\ 0000_2$ → $1111\ 1111_2$
  - `0x69 & 0x55` → `0x41`
    - $0110\ 1001_2$ & $0101\ 0101_2$ → $0100\ 0001_2$
  - `0x69 | 0x55` → `0x7D`
    - $0110\ 1001_2$ | $0101\ 0101_2$ → $0111\ 1101_2$

Logic Operations in C

- **Contrast to Bit-Level Logical Operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1

- **Examples**
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`

  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

Logic Operations in C

- **Contrast to Bit-Level Logical Operators**
  - `&&, ||, !`
    - View 0 as “Fa
    - Anything nonze ”
    - Alway
- **Example**
  - `!0x41`
  - `!0x00`
  - `!!0x41`
  - `0x69 &&`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

Watch out for difference
&& vs. & (and || vs. |)…
one of the more common errors in C programming

Shift Operations

- **Left Shift: `x << y`**
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- **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
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

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

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

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

Encoding Integers

1. Unsigned

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

- A *w-bit* unsigned binary number can represent 2^w unique values: from 0 to (2^w-1)
  - With 3 bits ($w=3$), we can represent the integers from 0 to 7 (2^3-1)

Binary number			Decimal Value
2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Encoding Integers

1. Unsigned

```
unsigned short int x =  12345;
unsigned short int y =  53191;
```

- C `unsigned short` data type (2 bytes)

	Decimal	Binary	Hex
`x`	12345	00110000 00111001	30 39
`y`	53191	11001111 11000111	CF C7

Unsigned Example

```
x = 12345: 00110000 00111001
y = 53191: 11001111 11000111
```

32768 (2^{15})	16384 (2^{14})	8192 (2^{13})	4096 (2^{12})	2048 (2^{11})	1024 (2^{10})	512 (2^{9})	256 (2^{8})	128 (2^{7})	64 (2^{6})	32 (2^{5})	16 (2^{4})	8 (2^{3})	4 (2^{2})	2 (2^{1})	1 (2^{0})	
0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1	
0	0	8192	4096	0	0	0	0	0	0	32	16	8	0	0	1	12345
1	1	0	0	1	1	1	1	1	1	0	0	0	1	1	1	
32768	16384	0	0	2048	1024	512	256	128	64	0	0	0	4	2	1	53191

Numeric Ranges

- **Unsigned Values**
  - $UMin \quad = \quad 0$

    000…0
  - $UMax \quad = \quad 2^w - 1$

    111…1

Encoding Integers

2. Two's Complement

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

Sign Bit

- A *w-bit* unsigned binary number can represent 2^w unique values: from (-2^{w-1}) to $(2^{w-1} - 1)$
  - With 3 bits (*w=3*), we can represent the integers from -4 (-2^2) to 3 (2^2-1)
- Sign Bit
  - For two's complement, most significant bit indicates sign
    - 0 for positive
    - 1 for negative

Binary number			Decimal Value
2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1

Encoding Integers

2. Two's Complement

```
short int x =  12345;
short int y = -12345;
```

- C `short` data type (2 bytes)

	Decimal	Binary	Hex
x	12345	00110000 00111001	30 39
y	-12345	11001111 11000111	CF C7

Two's-complement Encoding Example

```
x =  12345: 00110000 00111001
y = -12345: 11001111 11000111
```

-32768 (-2^{15})	16384 (2^{14})	8192 (2^{13})	4096 (2^{12})	2048 (2^{11})	1024 (2^{10})	512 (2^9)	256 (2^8)	128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)	
0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1	
0	0	8192	4096	0	0	0	0	0	0	32	16	8	0	0	1	12345
1	1	0	0	1	1	1	1	1	1	0	0	0	1	1	1	
-32768	16384	0	0	2048	1024	512	256	128	64	0	0	0	4	2	1	-12345

Numeric Ranges

- **Two's Complement Values**
  - *TMin* = -2^{w-1}

    100…0
  - *TMax* = $2^{w-1} - 1$

    011…1
  - -1

    111…1

Values for W = 16

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

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
  - $UMax = 2 * TMax + 1$
    - *or* $(2 * |TMin| - 1)$
- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific

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

Converting Decimal to Binary (2's C)

First Method: *Division*

1. **Find magnitude of decimal number. (Always positive.)**
2. **Divide by two – remainder is least significant bit.**
3. **Keep dividing by two until answer is zero, writing remainders from right to left.**
4. **Append a zero as the MS bit; if original number was negative, take complement and add 1.**

$X = 104_{10}$

104/2	=	52 r0	*bit 0*
52/2	=	26 r0	*bit 1*
26/2	=	13 r0	*bit 2*
13/2	=	6 r1	*bit 3*
6/2	=	3 r0	*bit 4*
3/2	=	1 r1	*bit 5*
1/2	=	0 r1	*bit 6*

$X = 01101000_2$

Converting Decimal to Binary (2's C)

First Method: *Division*

1. **Find magnitude of decimal number. (Always positive.)**
2. **Divide by two – remainder is least significant bit.**
3. **Keep dividing by two until answer is zero, writing remainders from right to left.**
4. **Append a zero as the MS bit; if original number was negative, take take complement and add 1.**

$X = -104_{10}$

104/2	=	52 r0	*bit 0*
52/2	=	26 r0	*bit 1*
26/2	=	13 r0	*bit 2*
13/2	=	6 r1	*bit 3*
6/2	=	3 r0	*bit 4*
3/2	=	1 r1	*bit 5*
1/2	=	0 r1	*bit 6*

Two's complement of 1101000 = 0010111+1 = 0011000

$X = 10011000_2$

Converting Decimal to Binary (2's C)

Second Method: *Subtract Powers of Two*

1. Find magnitude of decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit;
   if original was negative, MS bit is 1 and take take complement and add 1.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$X = 104_{10}$

$104 - 64\ (2^6)$	$=$	40	*bit 6*
$40 - 32\ (2^5)$	$=$	8	*bit 5*
$8 - 8\ (2^3)$	$=$	0	*bit 3*

$X = 01101000_2$

Converting Decimal to Binary (2's C)

Second Method: *Subtract Powers of Two*

1. **Find magnitude of decimal number.**
2. **Subtract largest power of two less than or equal to number.**
3. **Put a one in the corresponding bit position.**
4. **Keep subtracting until result is zero.**
5. **Append a zero as MS bit;**
   if original was negative,
   MS bit is 1 and take complement and add 1.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$X = -104_{10}$

$104 - 64\ (2^6)$	=	40	*bit 6*
$40 - 32\ (2^5)$	=	8	*bit 5*
$8 - 8\ (2^3)$	=	0	*bit 3*

Two's complement of 1101000 = 0010111+1 = 0011000

$X=10011000_2$

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

Conversions Between Signed & Unsigned

```
short int x = -12345;
unsigned short ux = (unsigned short) x;
printf("x = %d, ux = %u\n", x,ux);
```

Output:

```
x = -12345, ux = 53191
```

```
unsigned u = 4294967295u;    /* Umax */
int tu = (int) u;
printf("u = %u, tu = %d\n", u,tu);
```

Output:

```
u = 4294967295, tu = -1
```

Conversions Between Signed & Unsigned

Keep bit representations and reinterpret

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

$\rightarrow$ T2U $\rightarrow$

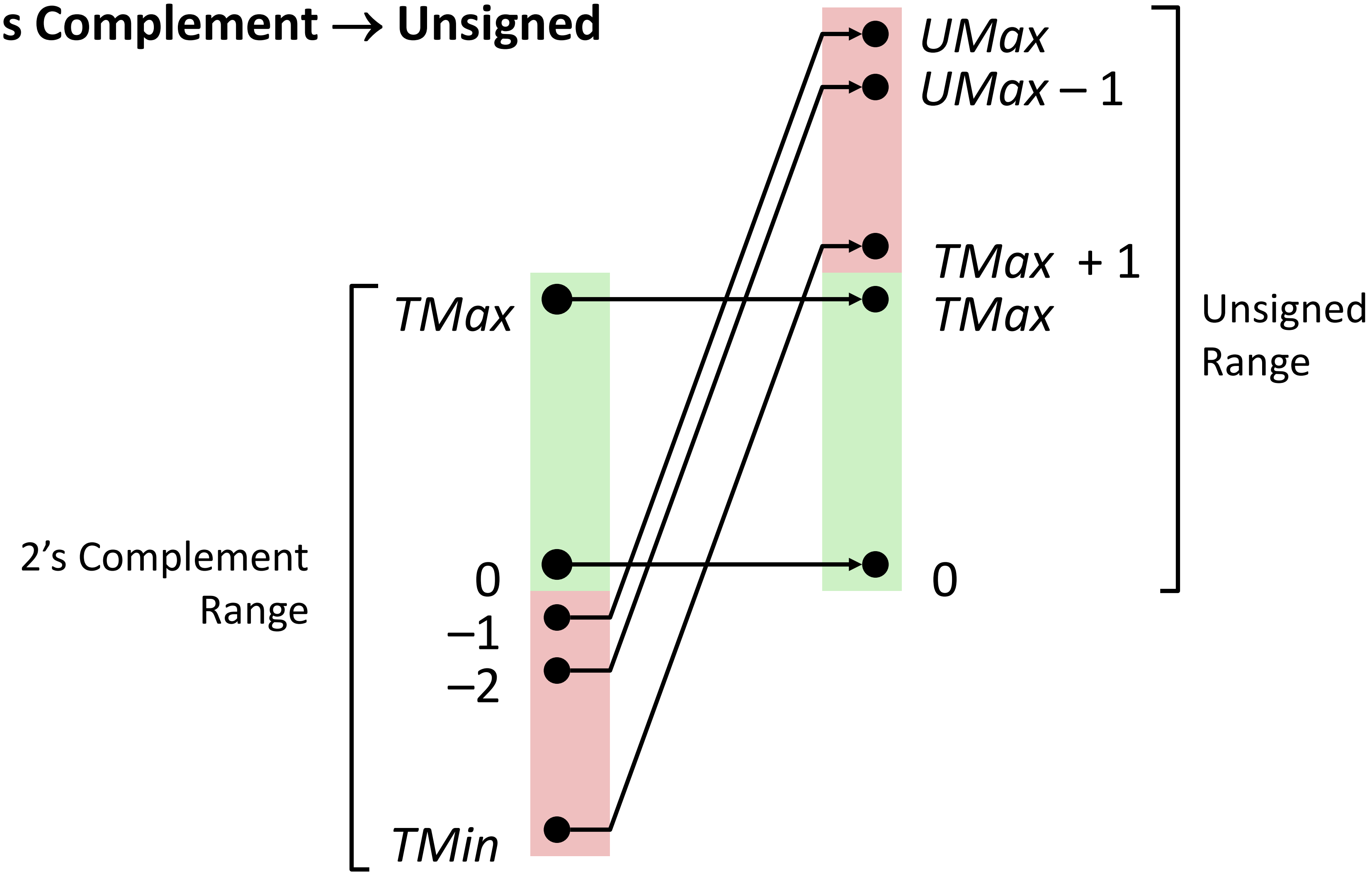
$\leftarrow$ U2T $\leftarrow$

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	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Conversion Visualized

- **2's Complement → Unsigned**



Signed vs. Unsigned in C

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

    ```
    0U, 4294967259u
    ```

- **Casting**

  ```
  int tx, ty;
  unsigned ux, uy;
  ```

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

    ```
    tx = (int) ux;       /* cast to signed */
    uy = (unsigned) ty; /* cast to unsigned */
    ```

  - Implicit casting also occurs via assignments and procedure calls

    ```
    tx = ux;    /* cast to signed */
    uy = ty;    /* cast to unsigned */
    ```

Casting Examples in C

- If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 4$: **TMin = -8 ,** **TMax = 7**

Constant 1	Expression	Constant 2	Type	Evaluation
0	==	0U	unsigned	1
-1	<	0	signed	1
-1	<	0U	unsigned	0
7	>	-7-1	signed	1
7U	>	-7-1	unsigned	0
-1	>	-2	signed	1
(unsigned)-1	>	-2	unsigned	1
7	>	(int) 8U	signed	1
7	<	8U	unsigned	1

Unsigned vs. Signed: Easy to Make Mistakes

```
unsigned i;
for (i=cnt-2; i>=0; i--)     /* loop never ends */
    a[i] += a[i+1];    /* invalid array subscript */
```

- **C Standard guarantees that unsigned addition will behave like modular arithmetic**
  - 0 – 1 = UMax
- **Proper way to use unsigned as loop index**

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

- **Even better**

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

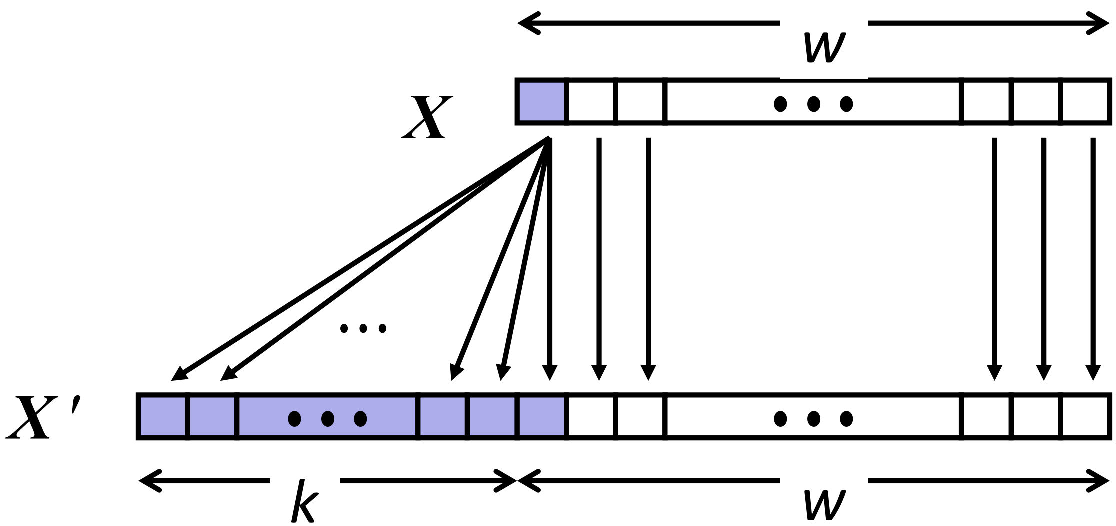
  - Data type **`size_t`** defined as unsigned value with length = word size
  - Code will work even if **`cnt`** = *UMax*
  - What if **`cnt`** is signed and < 0?

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

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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension: Simple Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Sign Extension Example (in C)

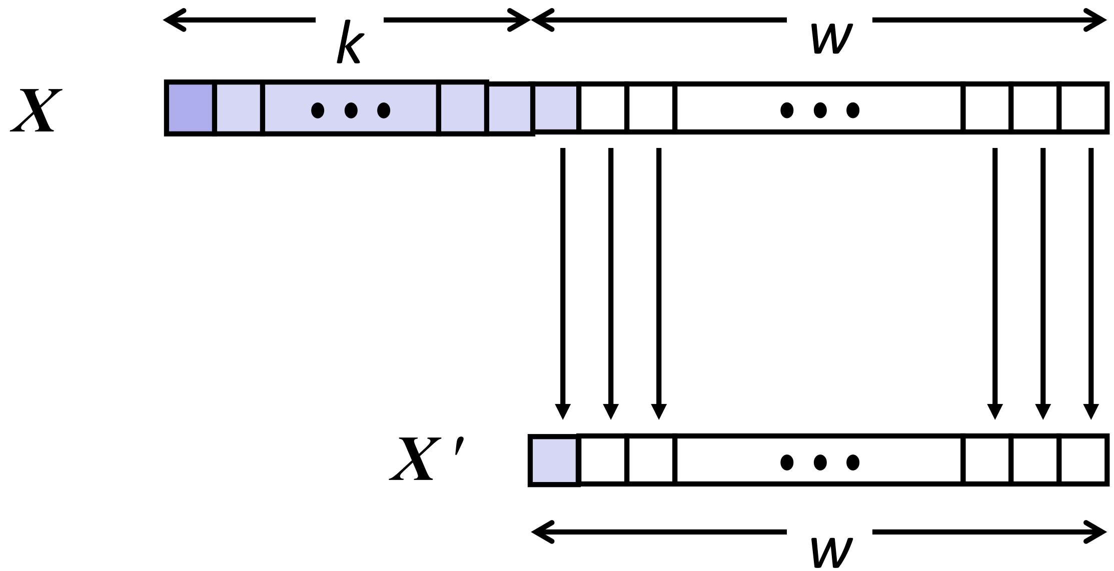
```
short sx =  -12345;
unsigned ux = sx;                /* ux=4294954951 */
unsigned uy = (unsigned)(unsigned short) sx;   /* uy=53191 */
```

	Decimal	Binary	Hex
sx	-12345	11001111 11000111	CF C7
ux	4294954951	11111111 11111111 11001111 11000111	FF FF CF C7
uy	53191	00000000 00000000 11001111 11000111	00 00 CF C7

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

Truncation

- **Task:**
  - Given (k+w)-bit signed or unsigned integer X
  - Convert it to w-bit integer X' with same value for "small enough" X
- **Rule:**
  - Drop top k bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation: Simple Example

No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

2 mod 16 = 2

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

-6 mod 16 = 26U mod 16 = 10U = -6

Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

10 mod 16 = 10U mod 16 = 10U = -6

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1
6 =	0	1	1	0

-10 mod 16 = 22U mod 16 = 6U = 6

Truncation Example (in C)

```
int    x = 53191;
short sx = (short) x;  /* sx = -12345 */
int    y = sx;         /*  y = -12345 */
```

	Decimal	Binary	Hex
x	53191	00000000 00000000 11001111 11000111	00 00 CF C7
sx	-12345	11001111 11000111	CF C7
y	-12345	11111111 11111111 11001111 11000111	FF FF CF C7

Today: Bits, Bytes, and Integers

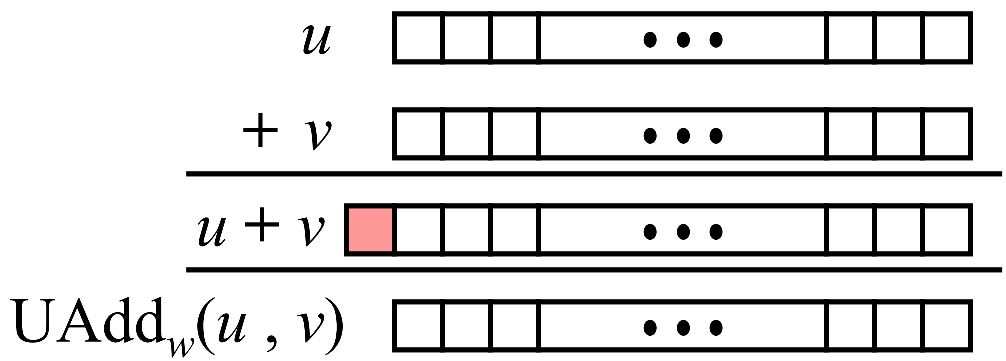
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

$$s \quad = \quad \mathrm{UAdd}_w(u\ ,\ v) \quad = \quad (u + v) \bmod 2^w$$

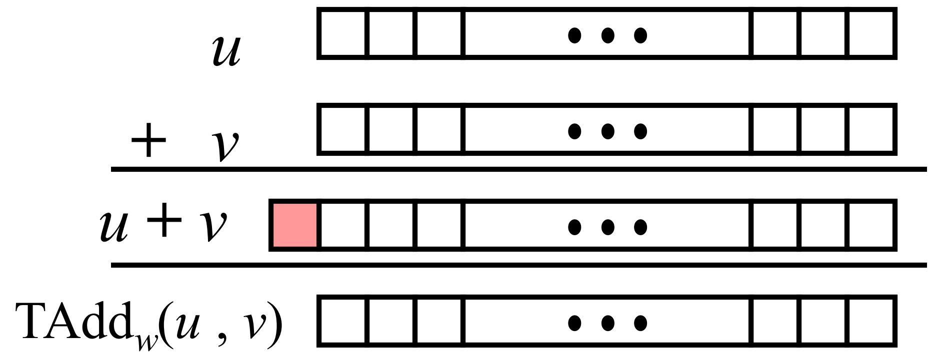
```
  233          E9        1110 1001
+ 213        + D5      + 1101 0101
  446         1BE      1 1011 1110
  190          BE        1011 1110   -- discard carry
```

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`

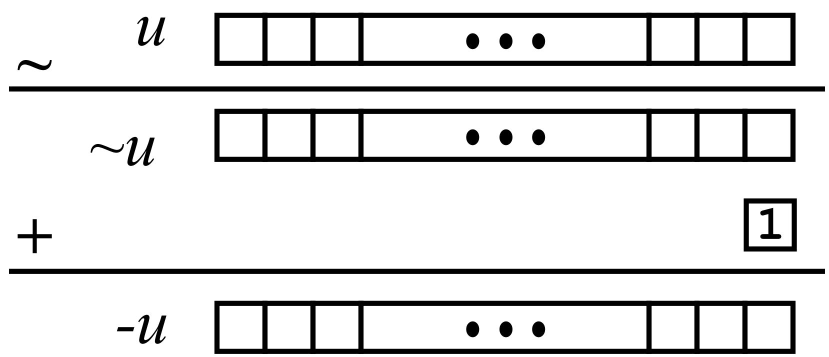
-23	E9	1110 1001
+ -107	+ 95	+ 1001 0101
-130	17E	1 0111 1110
126	7E	0111 1110

Negation: Complement and Increment

Operand

Take complement

Add 1



Negation: Complement and Increment

Observation: `~x + 1 == -x` → `~x + x == -1`

```
  01011100   x
+ 10100011  ~x
  11111111  -1
```

	Decimal	Binary	Hex
u	7	0111	7
~u	-8	1000	8
~u+1	-7	1001	9

	Decimal	Binary	Hex
u	-7	1001	9
~u	6	0110	6
~u+1	7	0111	7

Complement & Increment Examples

x=0

	Decimal	Binary	Hex
0	0	00000000 00000000	00 00
~0	-1	11111111 11111111	FF FF
~0+1	0	00000000 00000000	00 00

x=TMin

	Decimal	Binary	Hex
x	-32768	10000000 00000000	80 00
~x	32767	01111111 11111111	7F FF
~x+1	-32768	10000000 00000000	80 00

Canonical counter example

Multiplication

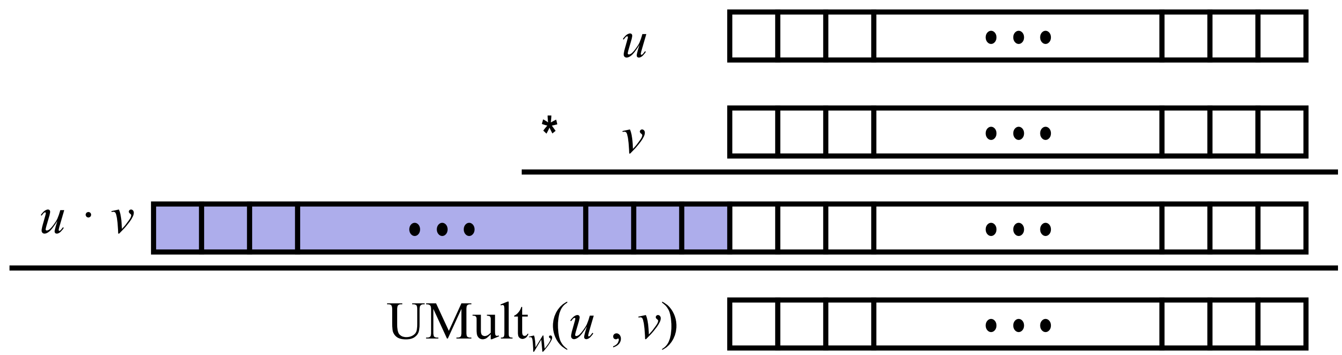
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Up to 2*w* bits

Unsigned Multiplication

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



Result range: $0 \leq x*y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

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

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

3	3	0011
* 5	* 5	* 0101
15	F	1111
15	F	1111

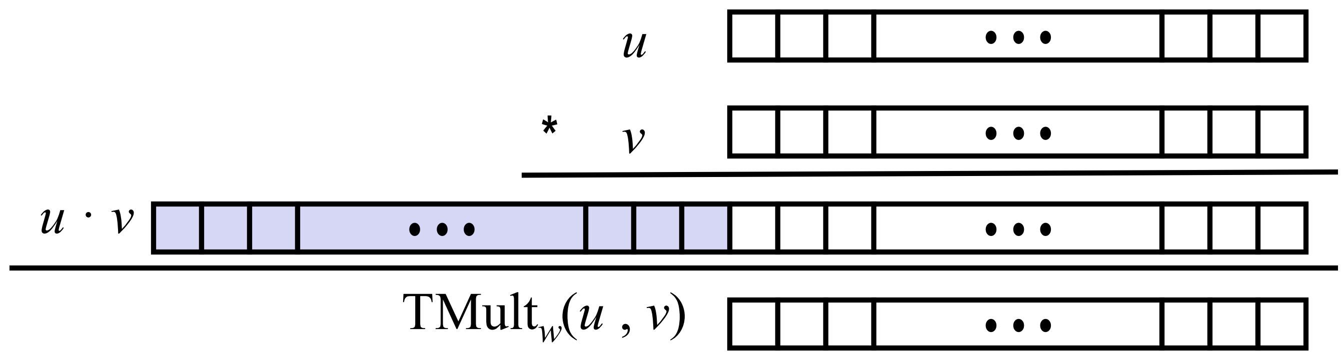
10	A	1010
* 5	* 5	* 0101
50	32	11 0010
2	2	0010

Signed Multiplication

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



$$(-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \le x*y \le (-2^{w-1})^2 = 2^{2w-2}$$

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

3	3	0011
* 5	* 5	* 0101
15	F	1111
-1	F	1111

-6	A	1010
* 5	* 5	* 0101
-30	32	11 0010
2	2	0010

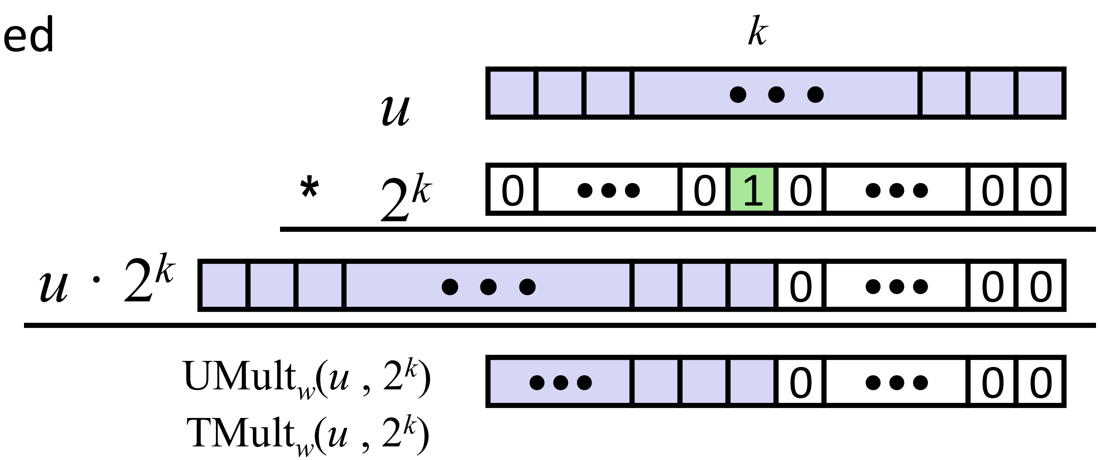
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w*+*k* bits

Discard *k* bits: *w* bits



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

Compiled Multiplication Code

C Function

```
long mul_12(long x){
  return x*12;
}
```

Compiled Arithmetic Operations

```
  leaq (%rax,%rax,2), %rax
  salq $2, %rax
```

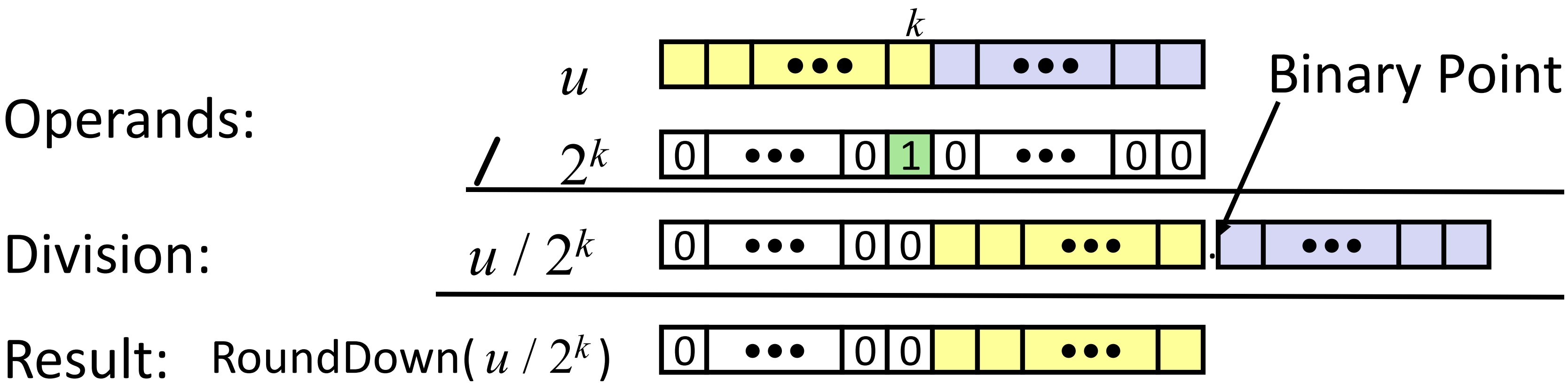
Explanation

```
  x = x + x*2
  return x << 2;
```

C compiler automatically generates shift/add code when multiplying by constant

Unsigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses logical shift



	Division	Computed	Binary
`x`	15	15	1111
`x >> 1`	7.5	7	0111
`x >> 2`	3.75	3	0011
`x >> 3`	1.875	1	0001

Compiled Unsigned Division Code

C Function

```
unsigned long udiv_8 (unsigned long x){
  return x/8;
}
```

Compiled Arithmetic Operations

```
  shrq $3, %rax
```

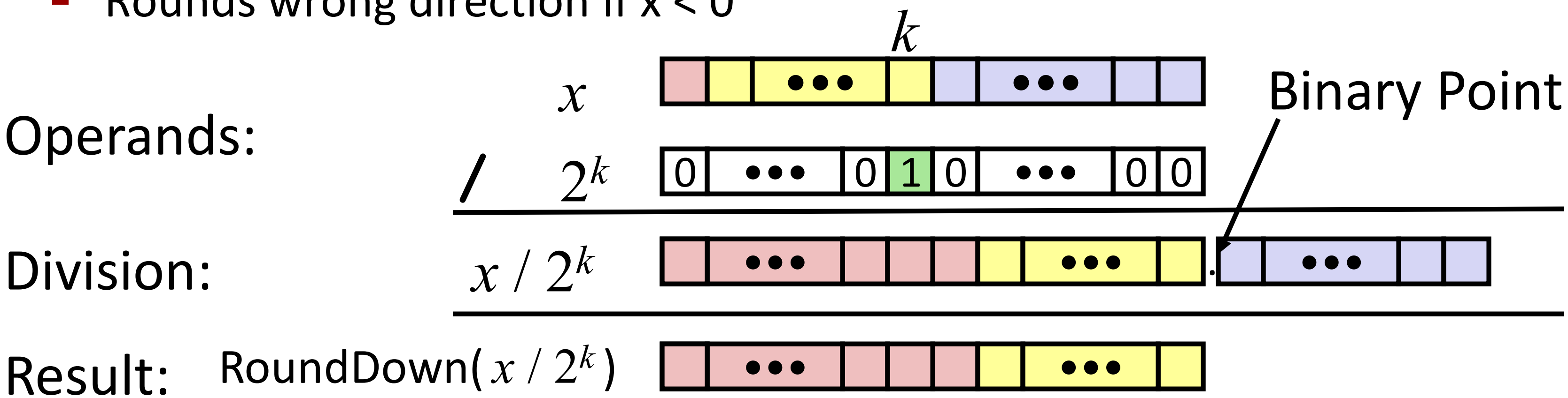
Explanation

```
  # Logical shift
  return x >> 3;
```

- **Uses logical shift for unsigned**
- **For Java Users**
  - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - **`x >> k`** gives $\lfloor$ **`x / 2`**^{k} $\rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction if x < 0



	Division	Computed	Binary
`x`	-7	-7	1001
`x >> 1`	-3.5	-4	1100
`x >> 2`	-1.75	-2	1110
`x >> 3`	-0.875	-1	1111

Compiled Signed Division Code

C Function

```
long idiv_4(long x){
  return x/4;
}
```

Compiled Arithmetic Operations

```
  testq %rax, %rax
  js    L4
L3:
  sarq $2, %rax
  ret
L4:
  addq $4, %rax
  jmp   L3
```

Explanation

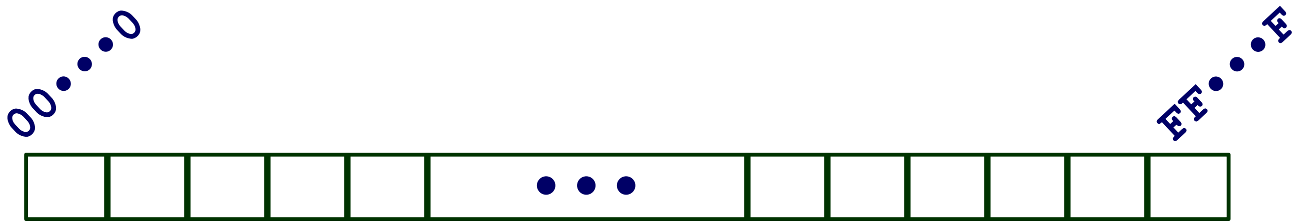
```
  if x < 0
    x += 3;
  # Arithmetic shift
  return x >> 2;
```

- Uses arithmetic shift for int
- **For Java Users**
  - Arith. shift written as >>

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**

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, imagine 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

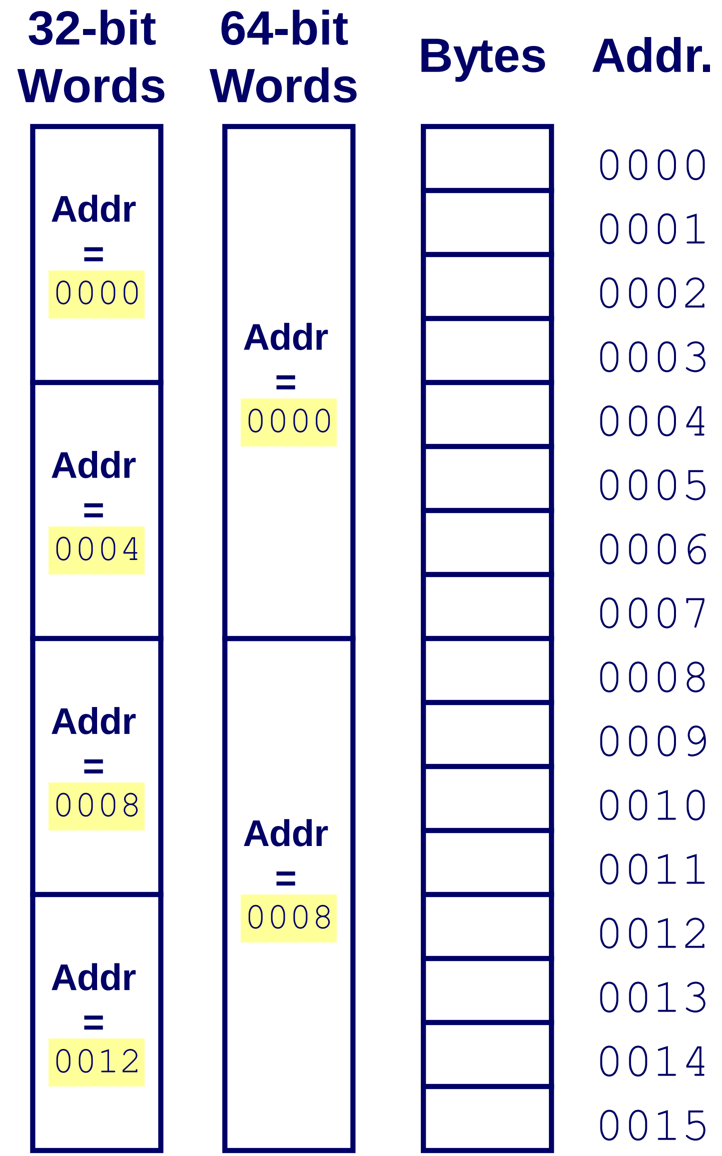
- **Note: system provides private address spaces to each “process”**
  - Think of a process as a program being executed
  - So, a program can overwrite its own data, but not that of others

Machine Words

- **Any given computer has a "Word Size"**
  - Size of integer-valued data
    - and size 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 X 10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

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)



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

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

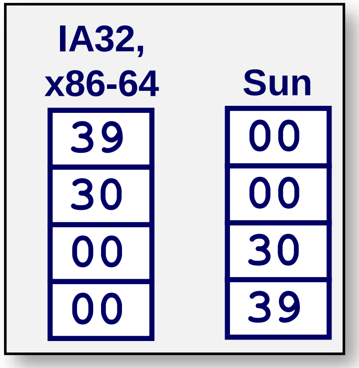
		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

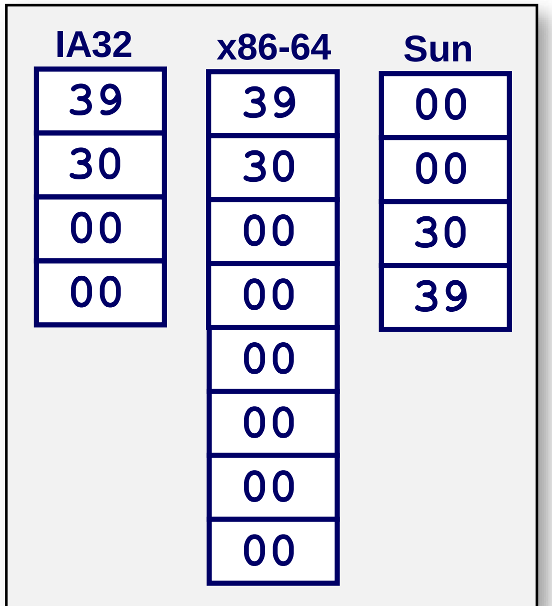
		0x100	0x101	0x102	0x103		
		67	45	23	01		

Representing Integers

`int A = 12345;`



`long int C = 12345;`

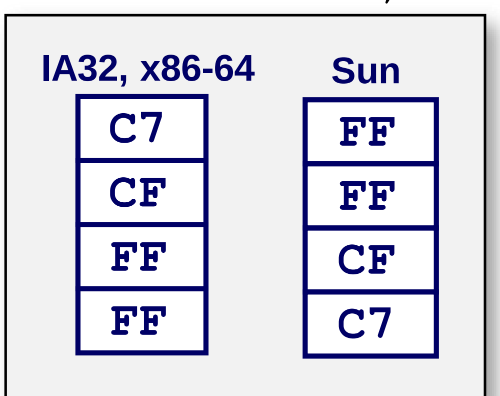


Decimal: 12345

Binary: 0011 0000 0011 1001

Hex: 3 0 3 9

`int B = -12345;`



Two's complement representation

Decimal: -12345

Binary: 1100 1111 1100 0111

Hex: C F C 7

Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting `pointer` to `unsigned char *` allows treatment as a byte array

```
typedef unsigned char *ucpointer;

void show_bytes(ucpointer 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

`show_bytes` Execution Example

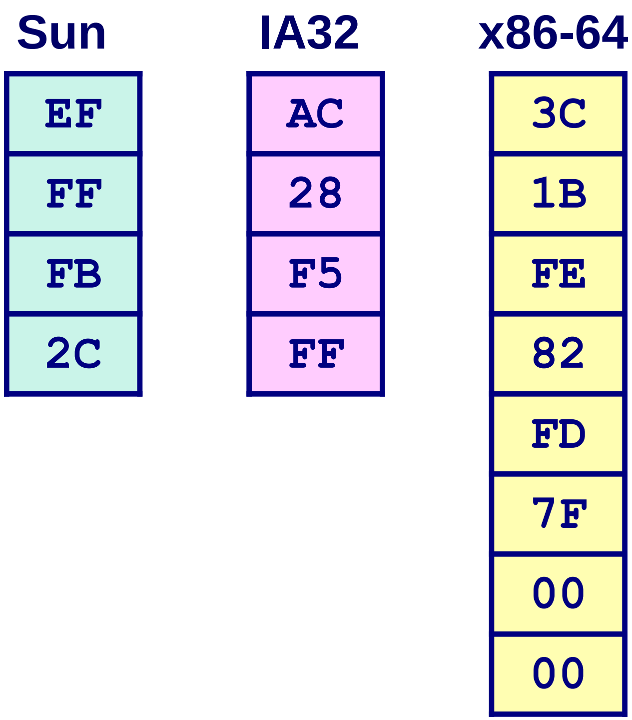
```
int a = 12345;
printf("int a = 12345;\n");
show_bytes((ucpointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 12345;
0x7fffb7f71dbc    39
0x7fffb7f71dbd    30
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

```
int B = -12345;
int *P = &B;
```



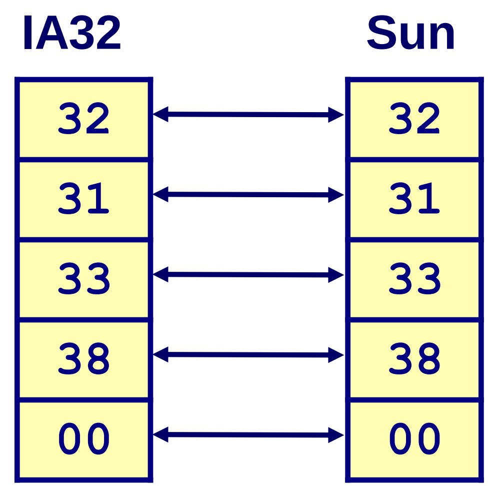
Different compilers & machines assign different locations to objects

Even get different results each run of the program

Representing Strings

- **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

```
char S[6] = "2138";
```



Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example Fragment (little endian machine)**

```
Address     Instruction Code          Assembly Code
8048365:    5b                        pop    %ebx
8048366:    81 c3 ab 12 00 00         add    $0x12ab,%ebx
804836c:    83 bb 28 00 00 00 00      cmpl   $0x0,0x28(%ebx)
```

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 4 bytes: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

Integer C Puzzles

Initialization

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

`(x < 0)` → `((x*2) < 0)`	False
`ux >= 0`	True
`x & 7 == 7` → `(x<<30) < 0`	True
`ux > -1`	False
`x > y` → `-x < -y`	False
`(x*x) >= 0`	False
`(x>0) && y > 0` → `x+y > 0`	False
`x >= 0` → `-x <= 0`	True
`x <= 0` → `-x >= 0`	False
`(x|-x)>>31 == -1`	False
`ux >> 3 == ux/8`	True
`x >> 3 == x/8`	False
`x & (x-1) != 0`	False