CSC3501

- **Temporary website: http://www.haow.ca/csc3501/**
  - Online video + Slides + Assignments
- **Zoom link**
  - Meeting ID: 315 813 3353
  - Passcode: csc3501

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

Examples :

```
Input : arr = {1, 2, 3, 2, 3, 1, 3}
Output : 3

Input : arr = {5, 7, 2, 7, 5, 2, 5}
Output : 5
```

```c
// C program to find the element
// occurring odd number of times
#include <stdio.h>

// Function to find element occurring
// odd number of times

int getOddOccurrence(int ar[], int ar_size) {
    int res = 0;
    for (int i = 0; i < ar_size; i++)
        res = res ^ ar[i];

    return res;
}

/* Driver function to test above function */
int main() {
    int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
    int n = sizeof(ar) / sizeof(ar[0]);

    // Function calling
    printf("%d", getOddOccurrence(ar, n));
    return 0;
}
```

0010
XOR 0011 3
0001
XOR 0010
0011
0011
0000

Today: Bits, Bytes, and Integers

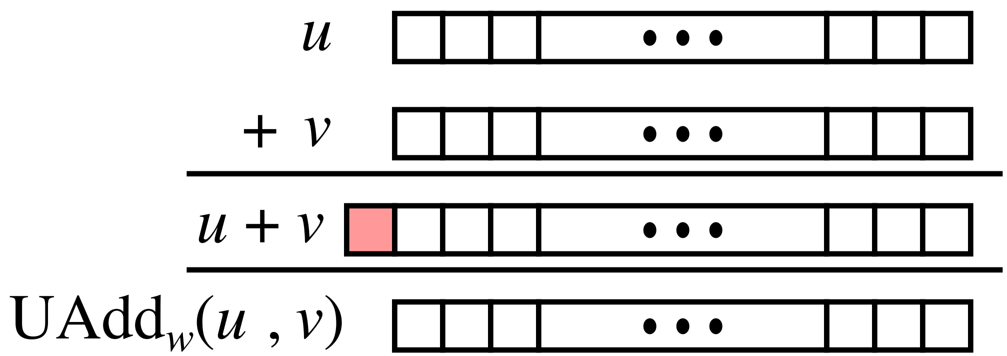
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

1101 13
1001 9
10110 22
0110 6

22 mod $2^4 = 16$
$= 6$

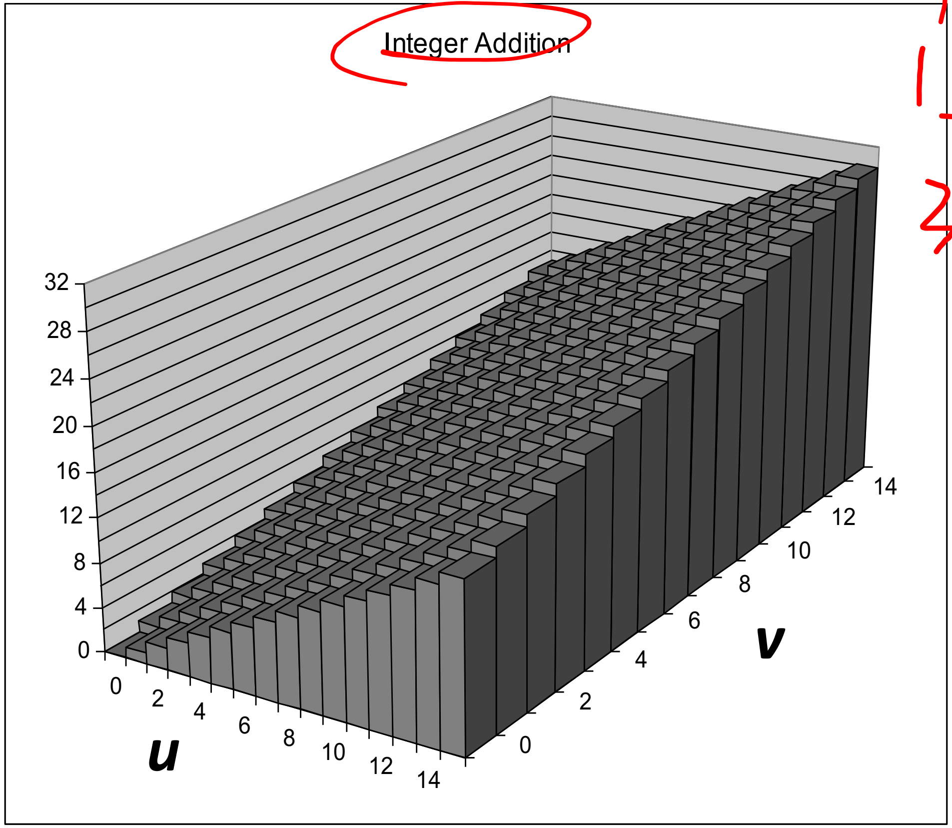
Visualizing (Mathematical) Integer Addition

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

15
+
15
30

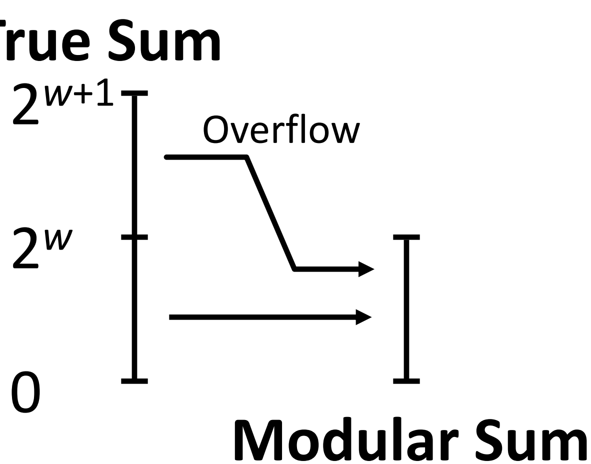
1111
1111

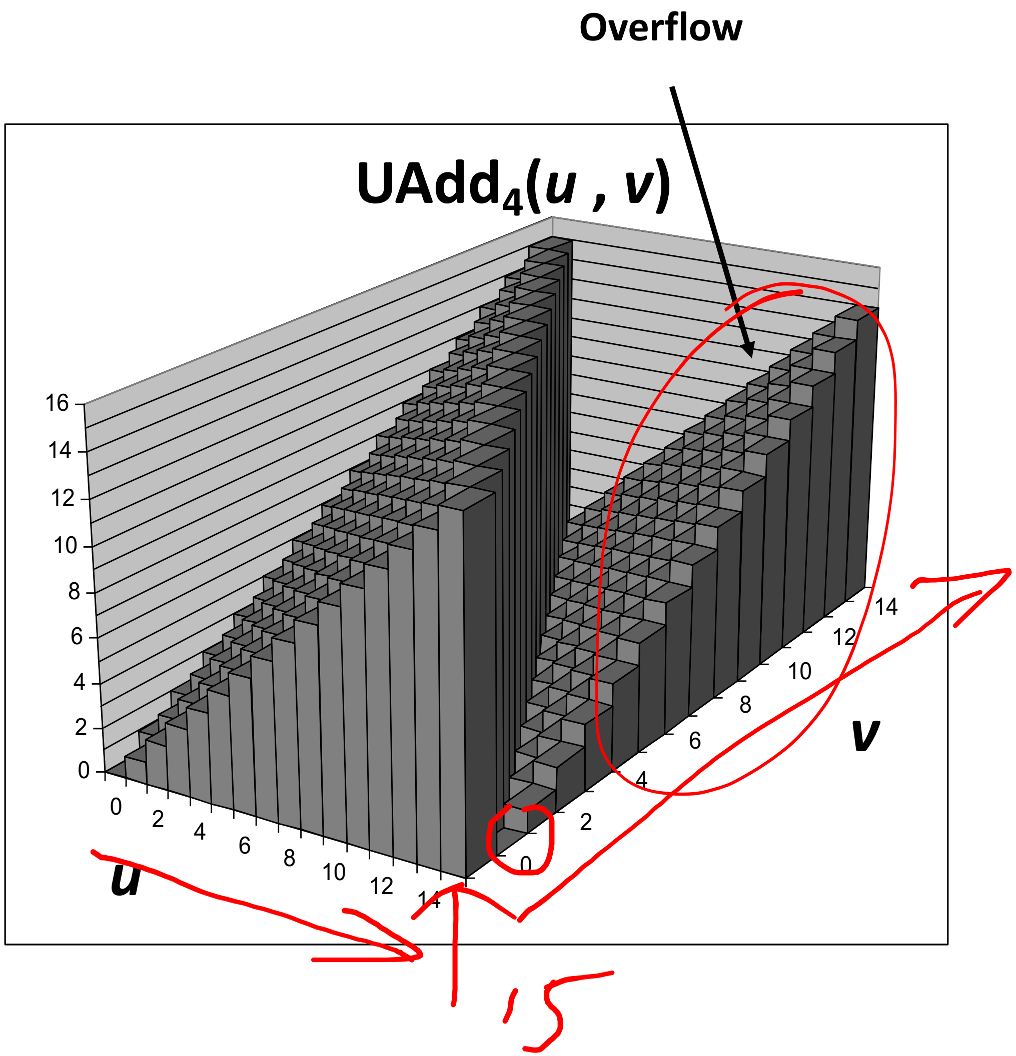
$Add_4(u, v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once



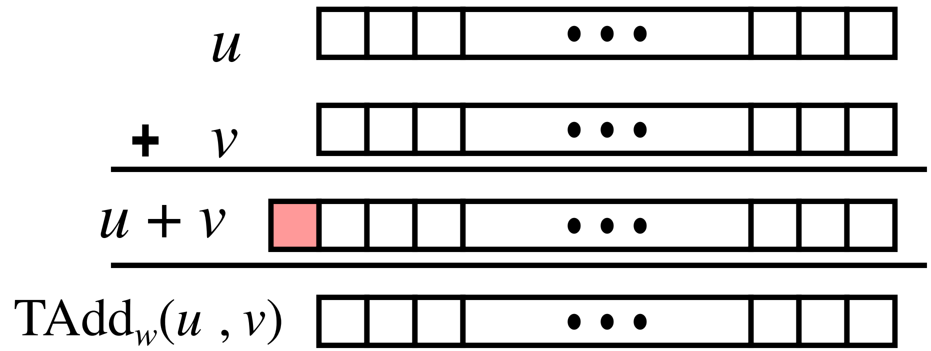


Two’s Complement Addition

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
```

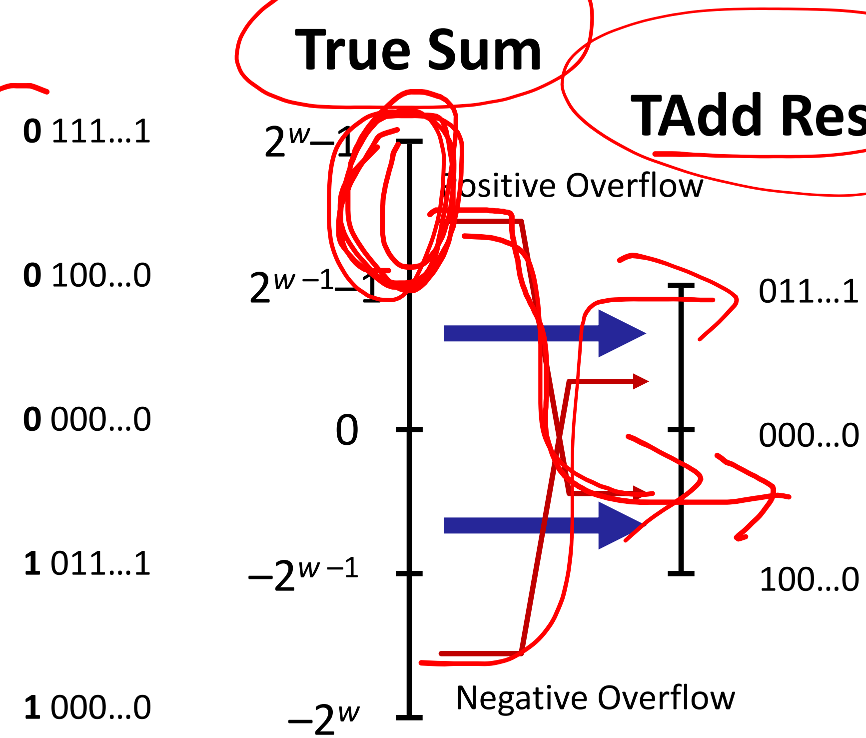
- Will give `s == t`

1101 -3
0101 5
10010 2

1101 (-3)
1010 (-6)
10111 (+7)

TAdd Overflow

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



0111 +7
0101 5
1100 -4

- **Two's Comp. Addition Range:**

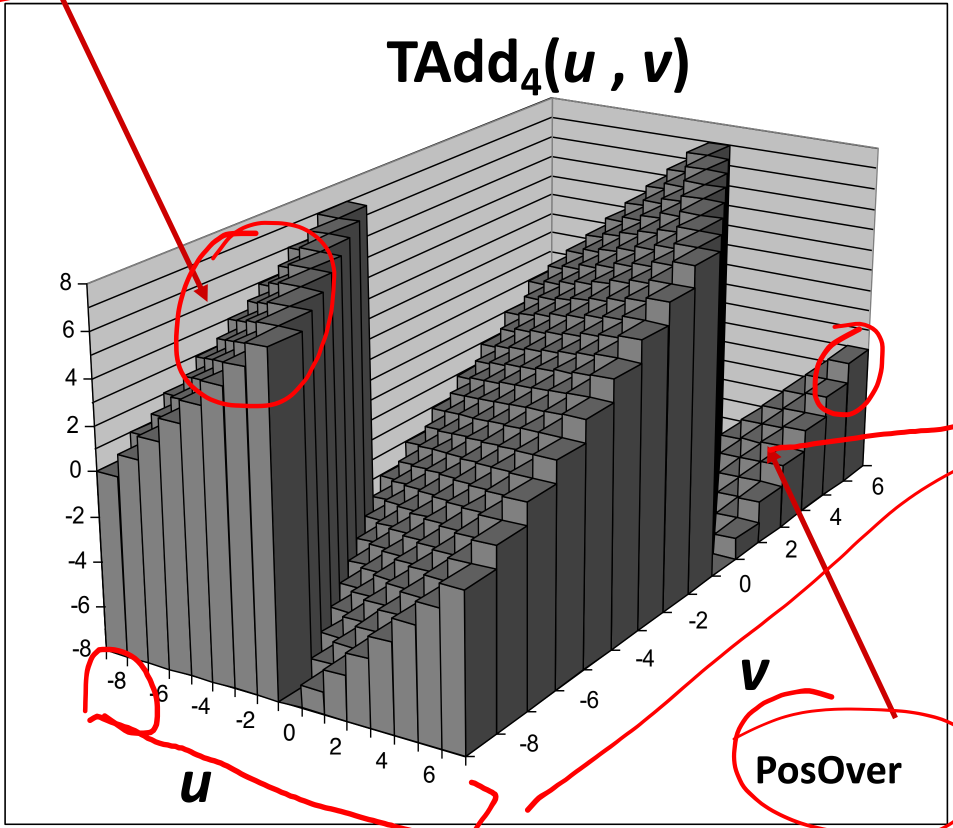
$[-2^{w-1} \times 2$ (-8)

$(2^{w-1} - 1) \times 2]$ 7

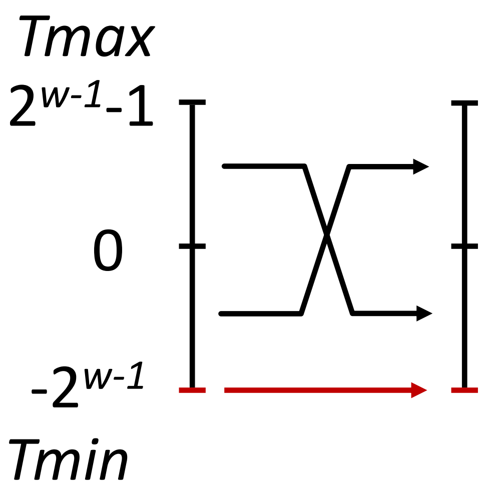
Visualizing 2's Complement Addition

- **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

NegOver



Two's-Complement Negation



- **For w-bit two's-complement addition**
  - *TMin* is its own additive
- **inverse, while any other value x has −x as its additive inverse.**

$$-^{t}_{w}\, x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases}$$

x		$-x$	
[1100]	−4	[0100]	4
[1000]	−8	[1000]	−8
[0101]	5	[1011]	−5
[0111]	7	[1001]	−7

Multiplication

$$
\begin{array}{cccccccccl}
 & & & & 1 & 1 & 0 & 1 & (13)_{10} & \text{Multiplicand M} \\
 & & & \times & 1 & 0 & 1 & 1 & (11)_{10} & \text{Multiplier Q} \\
\hline
 & & & & 1 & 1 & 0 & 1 & & \\
 & & & 1 & 1 & 0 & 1 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \text{Partial products} \\
 & 1 & 1 & 0 & 1 & & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & (143)_{10} & \text{Product P}
\end{array}
$$

Multiplication

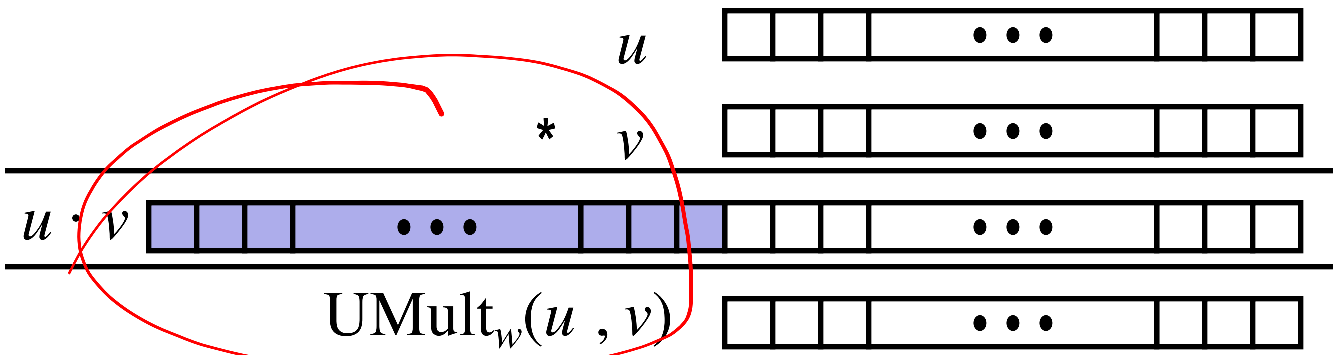
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to $2w$ bits
    - Result range: $0 \leq x * y \leq (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - 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

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



5 * 5 = 25

```
   0101
   0101
-------
   0101
  0000
 0101
0000
-------
0011001 = 9
```

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

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

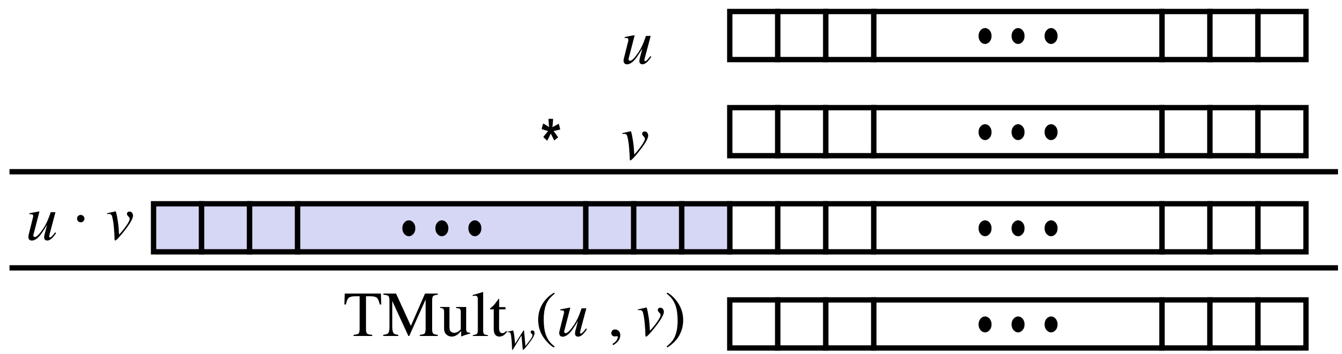
25 mod 16 = 9

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



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

1101 (-3)
1110 (-2)
0000
1101
1101
1101
... 10110
6
182
-6

Example

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's complement	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's complement	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's complement	3	[011]	3	[011]	9	[001001]	1	[001]

Figure 2.27 Three-bit unsigned and two's-complement multiplication examples. Although the bit-level representations of the full products may differ, those of the truncated products are identical.

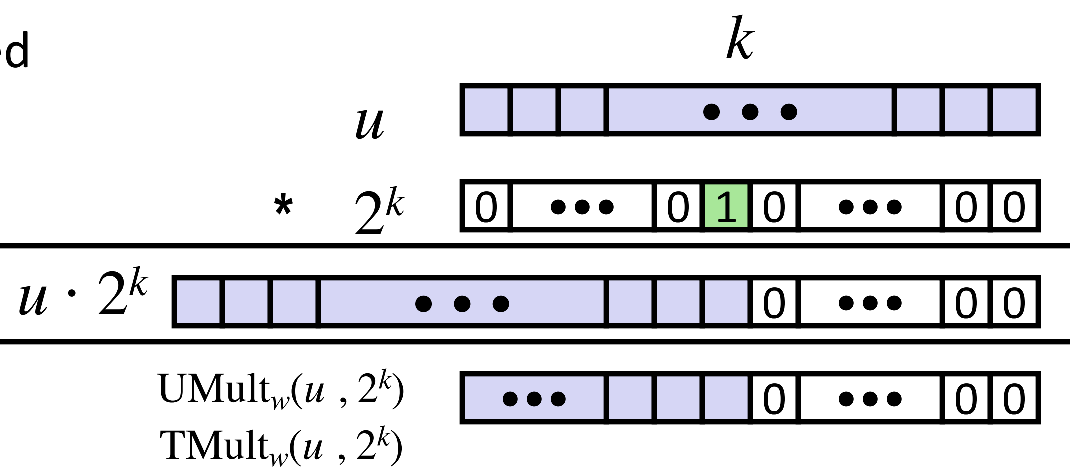
Power-of-2 Multiply with Shift

- **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

0110 (6)

1 <<

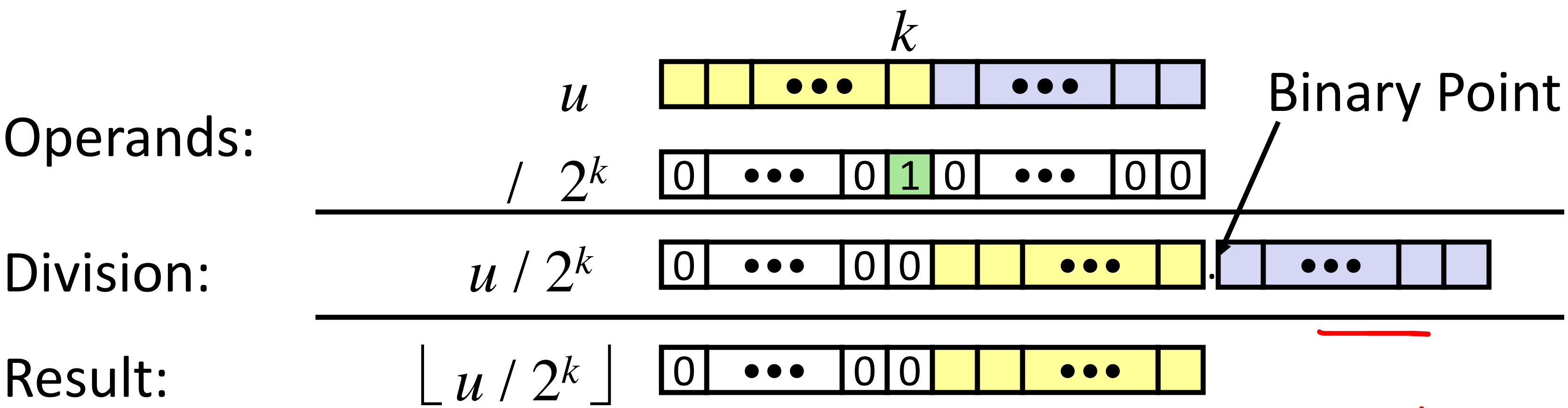
1100 (12)

6 * 2 = 12

Unsigned Power-of-2 Divide with Shift

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

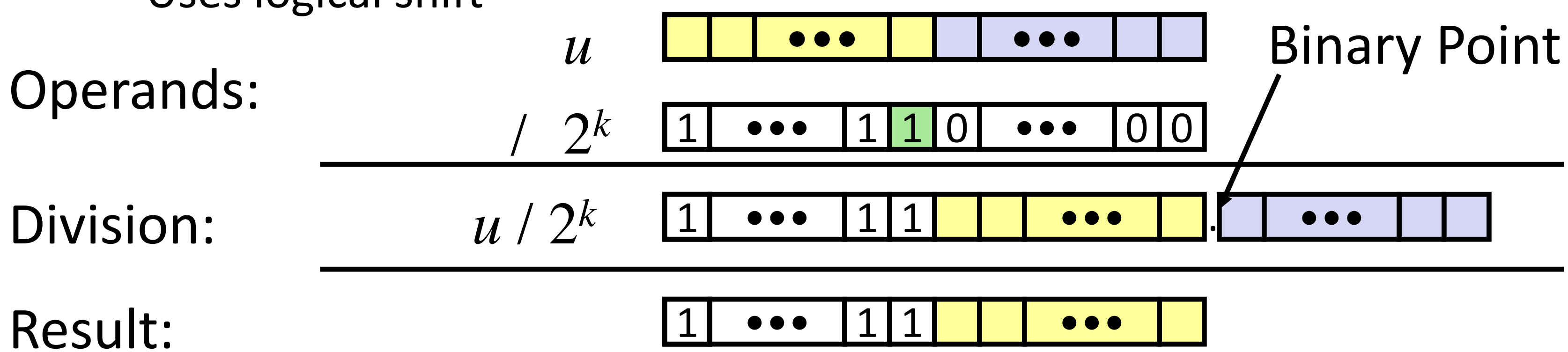
floor



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

Two's-Complement Division with Shift

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



k	>> k (binary)	Decimal	$-12{,}340/2^k$
0	**1100111111001100**	–12,340	–12,340.0
1	*1***110011111100110**	–6,170	–6,170.0
4	*1111***110011111100**	–772	–771.25
8	*11111111***11001111**	–49	–48.203125

Two's-Complement Division with Shift

- **Correction**
  - Adding a bias to fix
  - (u+ (1 << k) -1) >> k gives $\lceil u/2^k \rceil$.

ceil sym

k	Bias	–12,340 + bias (binary)	**>> k** (binary)	Decimal	–12,340/$2^{\mathbf{k}}$
0	0	**1100111111001100**	**1100111111001100**	–12,340	–12,340.0
1	1	**110011111100110***1*	*1***110011111100110**	–6,170	–6,170.0
4	15	**110011111101***1011*	*1111***110011111101**	–771	–771.25
8	255	**11010000***11001011*	*11111111***11010000**	–48	–48.203125

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

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)

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];
```

i = 0

u

-1

a + b

s

↓

u

  - Can be very subtle

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

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 → *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?

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

What if `cnt` is signed and < 0?

- **If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned***

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

{1, 3, 6, 5}

unsign

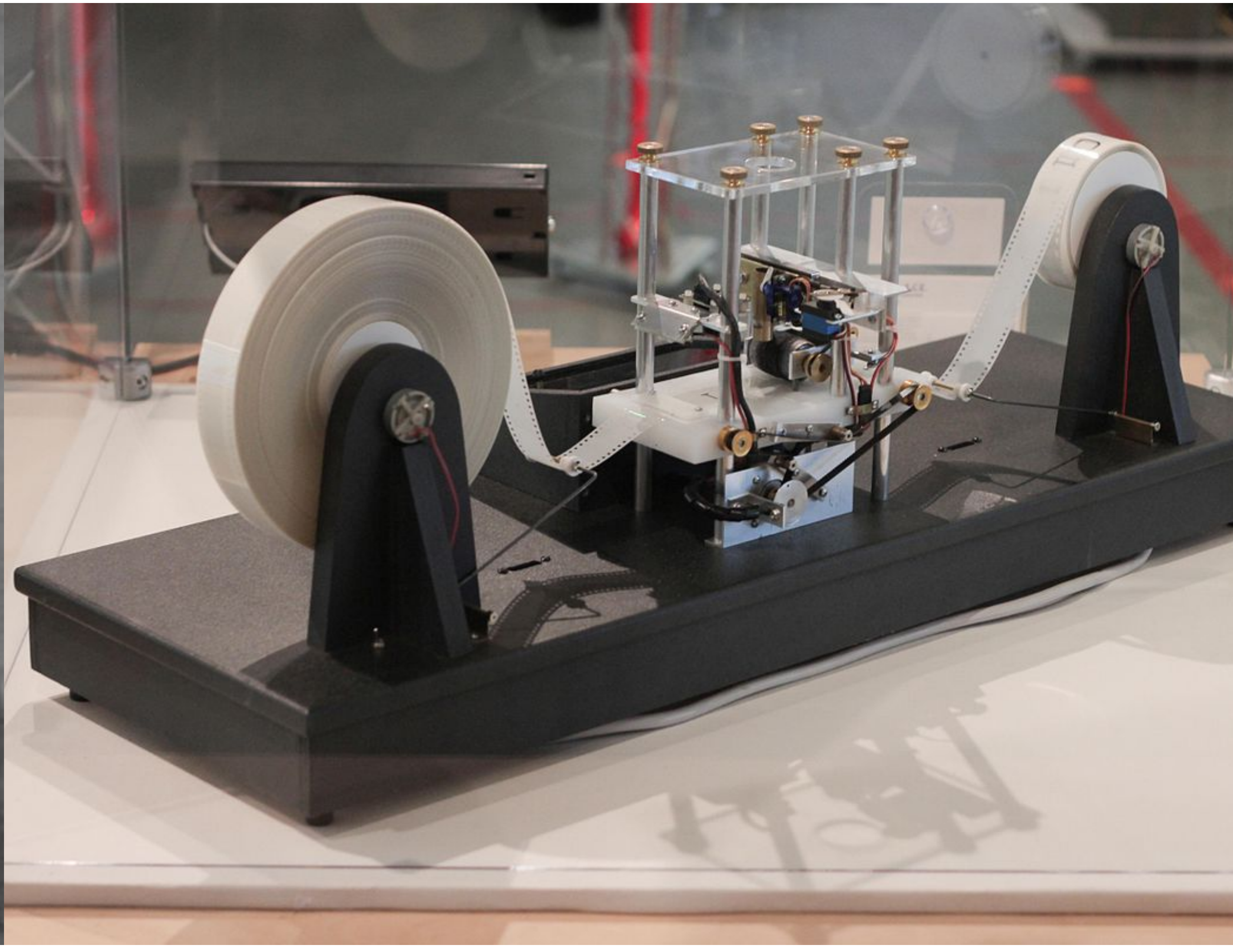
1 1 0 1 0 1 0

6 5 4 3 2 1 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**

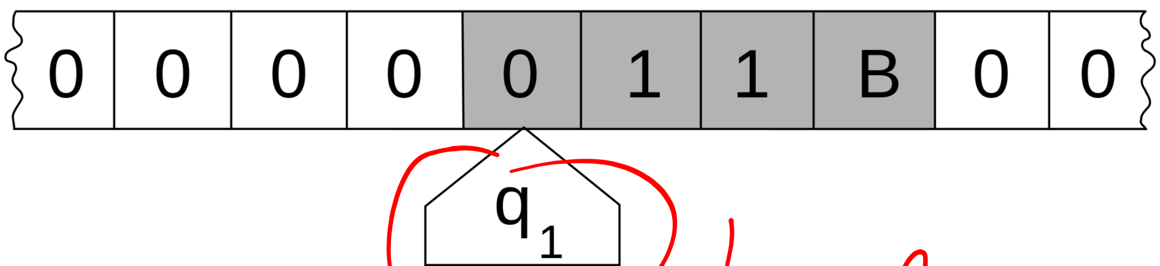
Turing Machine



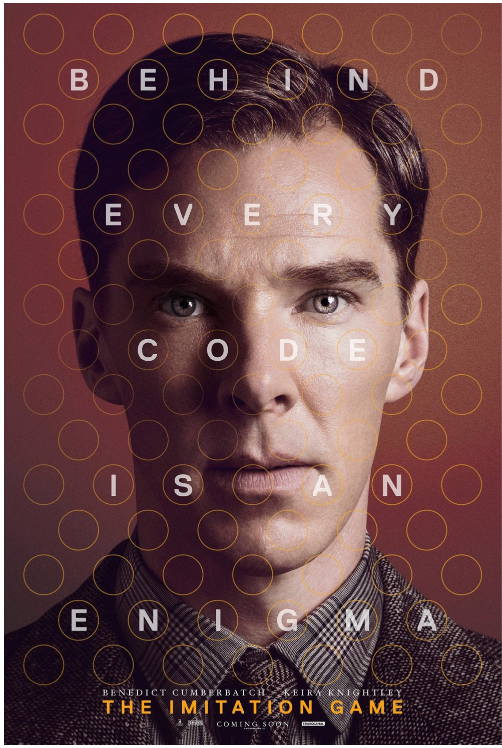
Turing Machine

- **Proposed by Alan Turing in 1936**

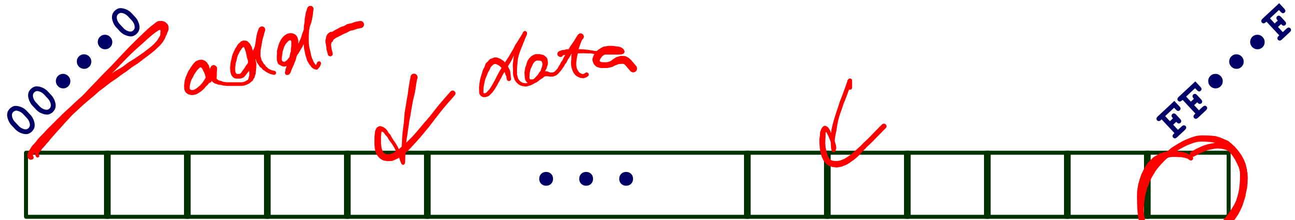
inf i



header



Byte-Oriented Memory Organization



byte.

8 bits.

- **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

addr

a[ ]

&p = addr

- **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

*p =

p+1

Machine Words

225 xxx xxxx

0-9 10

10-12

32

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses

$2^{32} = 4GB$ addr space

  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)

10^7

2^{32}

- Increasingly, machines have 64-bit word size
  - Potentially, could have 18 EB (exabytes) of addressable memory
  - That's 18.4 X 10^{18}

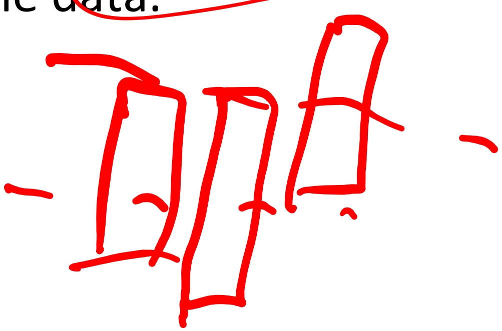
4bit.

8bit

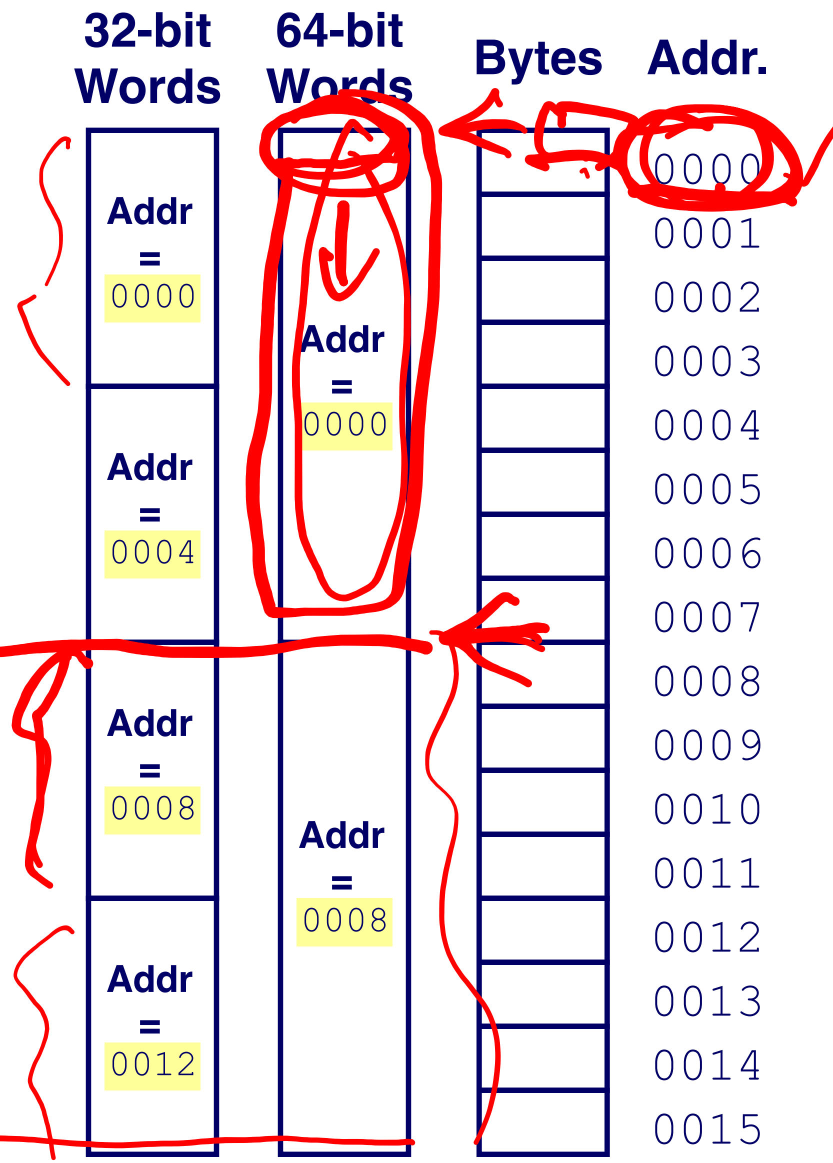
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

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)
  - Addresses of multi-byte data items are typically *aligned* according to the size of the data.



>> word-size.



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

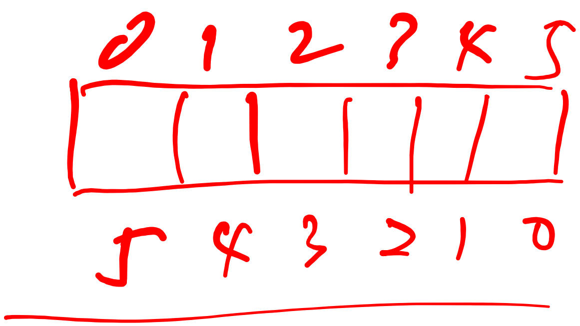
"a"

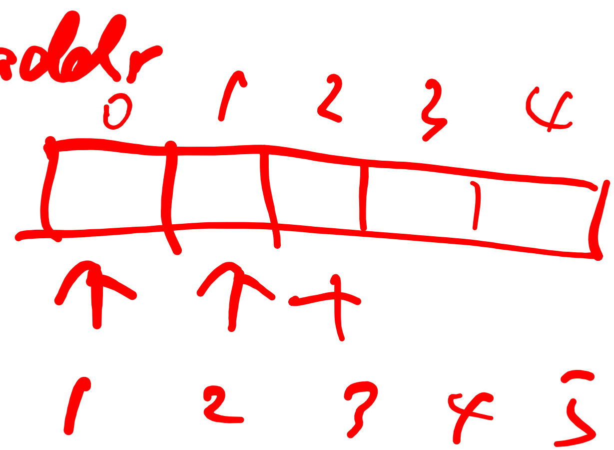
signed.

addr

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		

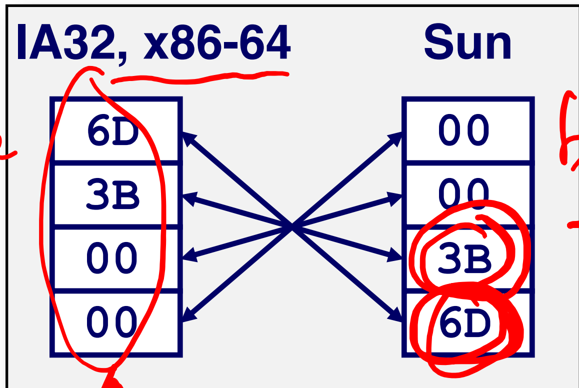
Representing Integers

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

0x3B6D

`int A = 15213;`

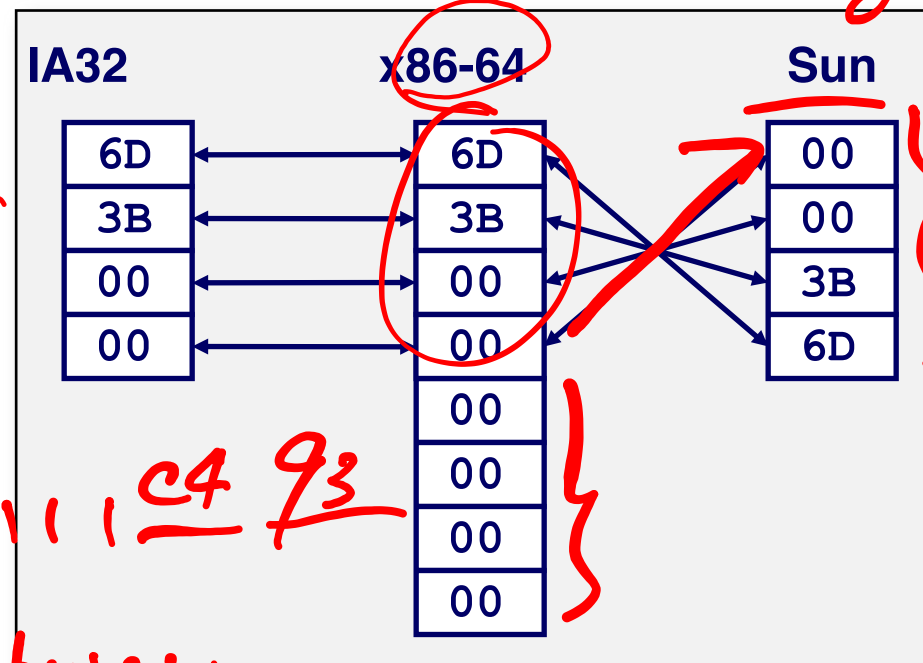
little



big

`long int C = 15213;`

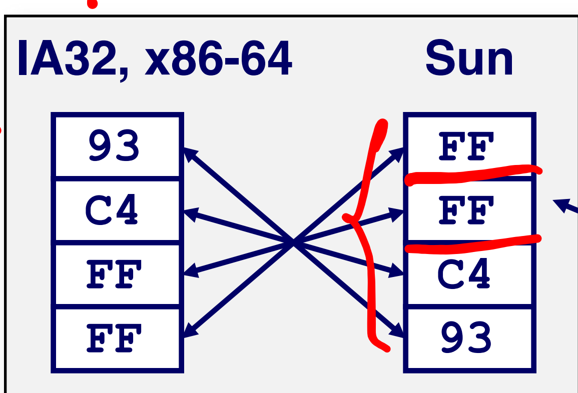
big



1111 c4 93

4 bytes.

`int B = -15213;`



Two's complement representation

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");
}
```

list.

pointer + offset

Printf directives:

%p: Print pointer addr.

%x: Print Hexadecimal

show_bytes Execution Example

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

pointer

6D
3B
00
00

list

Result (Linux x86-64):

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

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

Representing Strings

end of line

```
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

"0" "1"

0x30 0x31

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

end of line!

0x30

"5" → 0x35

7-bit character set encoding

Basic Character Set[2]

	0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
0x00	@	Δ	SP	0	¡	P	¿	p
0x01	£	_	!	1	A	Q	a	q
0x02	$	Φ	"	2	B	R	b	r
0x03	¥	Γ	#	3	C	S	c	s
0x04	è	Λ	¤	4	D	T	d	t
0x05	é	Ω	%	5	E	U	e	u
0x06	ù	Π	&	6	F	V	f	v
0x07	ì	Ψ	'	7	G	W	g	w
0x08	ò	Σ	(	8	H	X	h	x
0x09	Ç	Θ	)	9	I	Y	i	y
0x0A	LF	Ξ	*	:	J	Z	j	z
0x0B	Ø	ESC	+	;	K	Ä	k	ä
0x0C	ø	Æ	,	<	L	Ö	l	ö
0x0D	CR	æ	-	=	M	Ñ	m	ñ
0x0E	Å	ß	.	>	N	Ü	n	ü
0x0F	å	É	/	?	O	§	o	à