Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems
3rd Lecture, May 25, 2017

Instructors:

Brian Railing

Autolab/TPZ/Gitlab accounts

- You should have all your accounts by now
- You must be enrolled to get an account
- If you are on waitlist, just keep on hanging in there

- Redshelf and digital content

First Assignment: Data Lab

- Due: Thursday, June 1, 11:59:00 pm
- Last Possible Time to Turn in: Fri, June 2, 11:59PM
- Read the instructions carefully
- You should have started
- Seek help (office hours have started)
- Based on Lecture 2, 3 , and 4
- After today's lecture you know everything for the integer problems, float problems covered on Tuesday

Summary From Last Lecture

- 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

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 (Char data type)**
  - ~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
    - $0110\ 1001_2$ | $0101\ 0101_2$ → $0111\ 1101_2$

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

Logic Operations in C

- **Logic Operations: &&, ||, !**
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination
- **Examples (char data type)**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41→ 0x01

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

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

Two's Complement Examples (w = 5)

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

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

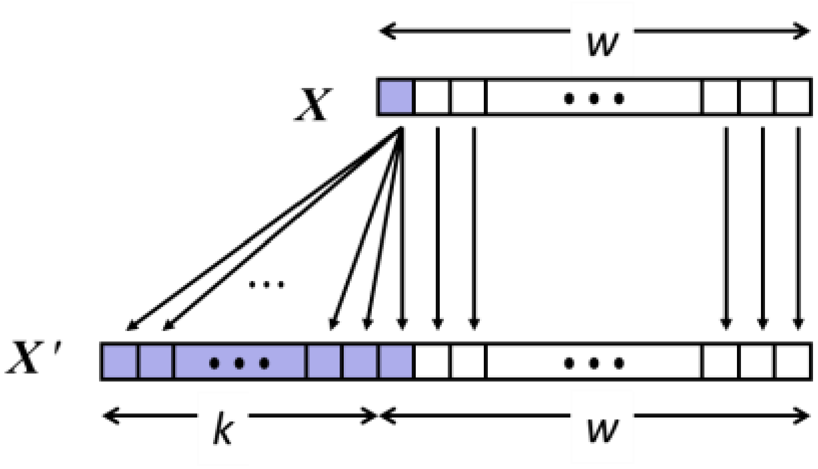
Unsigned & Signed Numeric Values

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

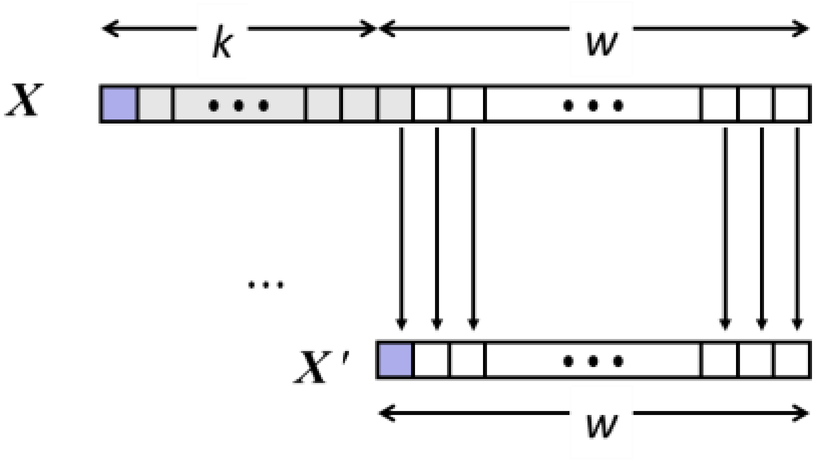
- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **Expression containing signed and unsigned int:**
  `int` is cast to `unsigned`!!

Sign Extension and Truncation

- **Sign Extension**



- **Truncation**



Today: Bits, Bytes, and Integers

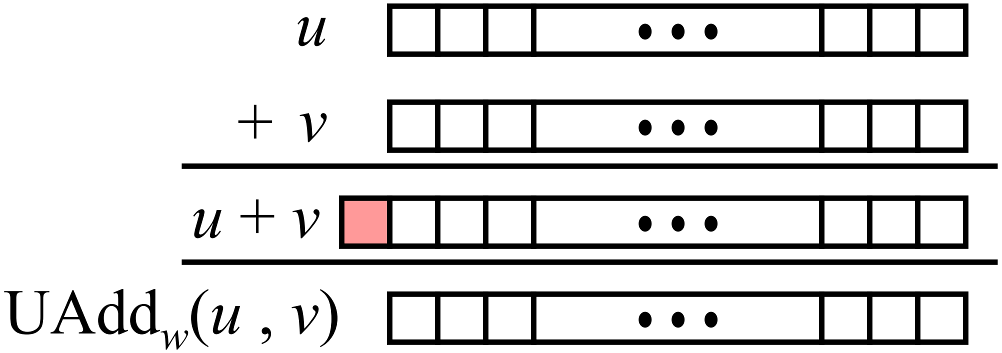
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

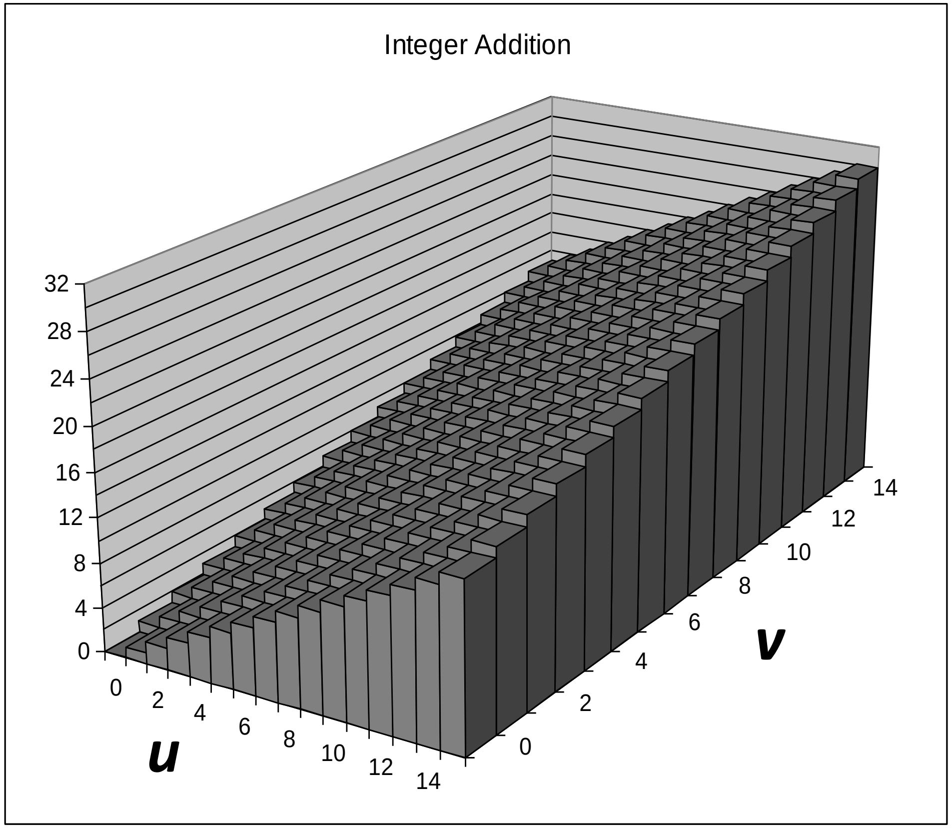
```
unsigned char       1110 1001       E9       223
                +   1101 0101     + D5     + 213
                  1 1011 1110      1BE       446
                    1011 1110       BE       190
```

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

Visualizing (Mathematical) Integer Addition

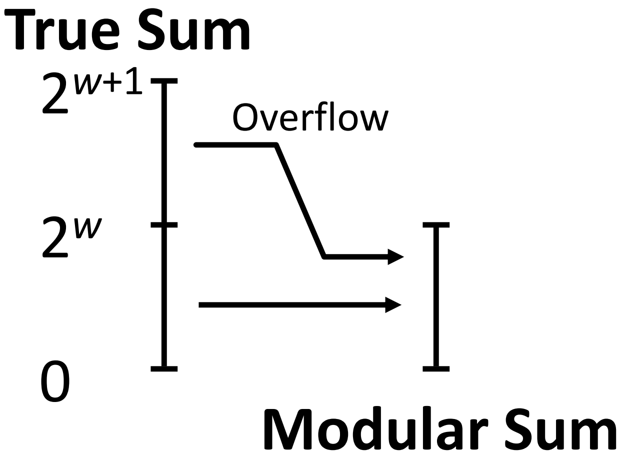
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

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



Visualizing Unsigned Addition

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



True Sum

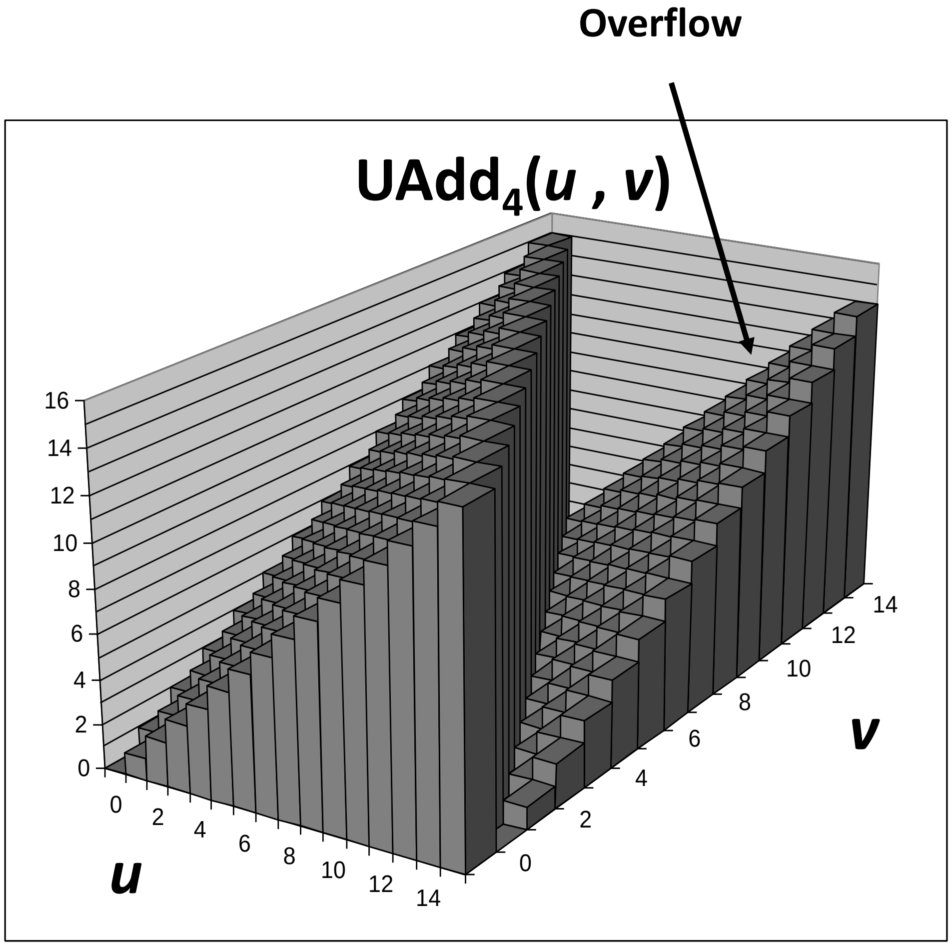
2^{w+1}

Overflow

2^w

0

Modular Sum

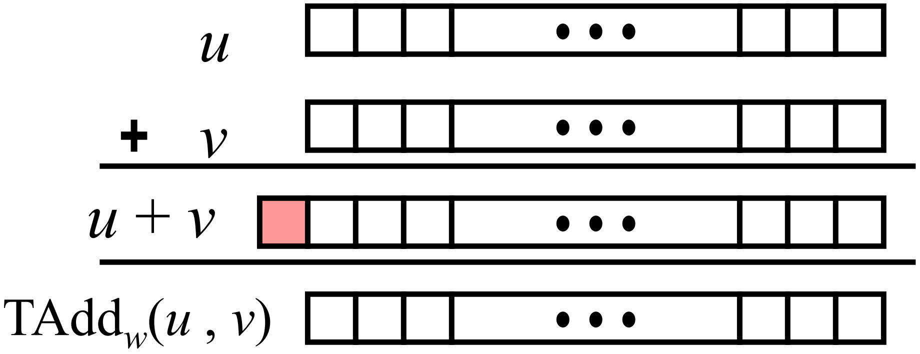


Two's Complement Addition

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:

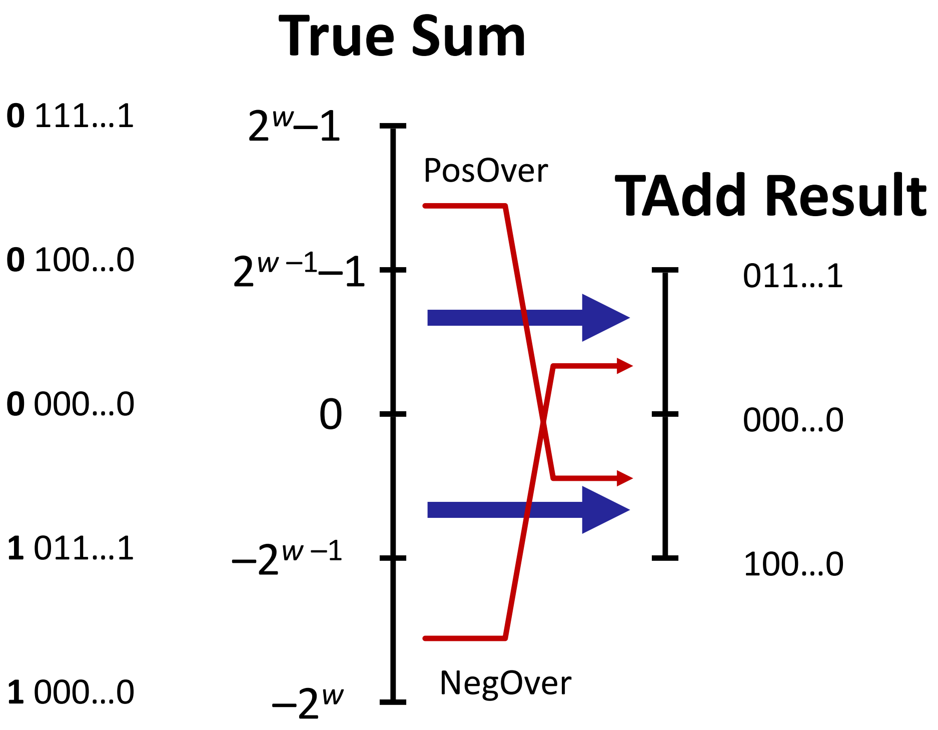
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	446
1011 1110	BE	-66

TAdd Overflow

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



True Sum

0 111…1 — $2^w–1$

0 100…0 — $2^{w-1}–1$

0 000…0 — 0

1 011…1 — $–2^{w-1}$

1 000…0 — $–2^w$

PosOver

NegOver

TAdd Result

011…1

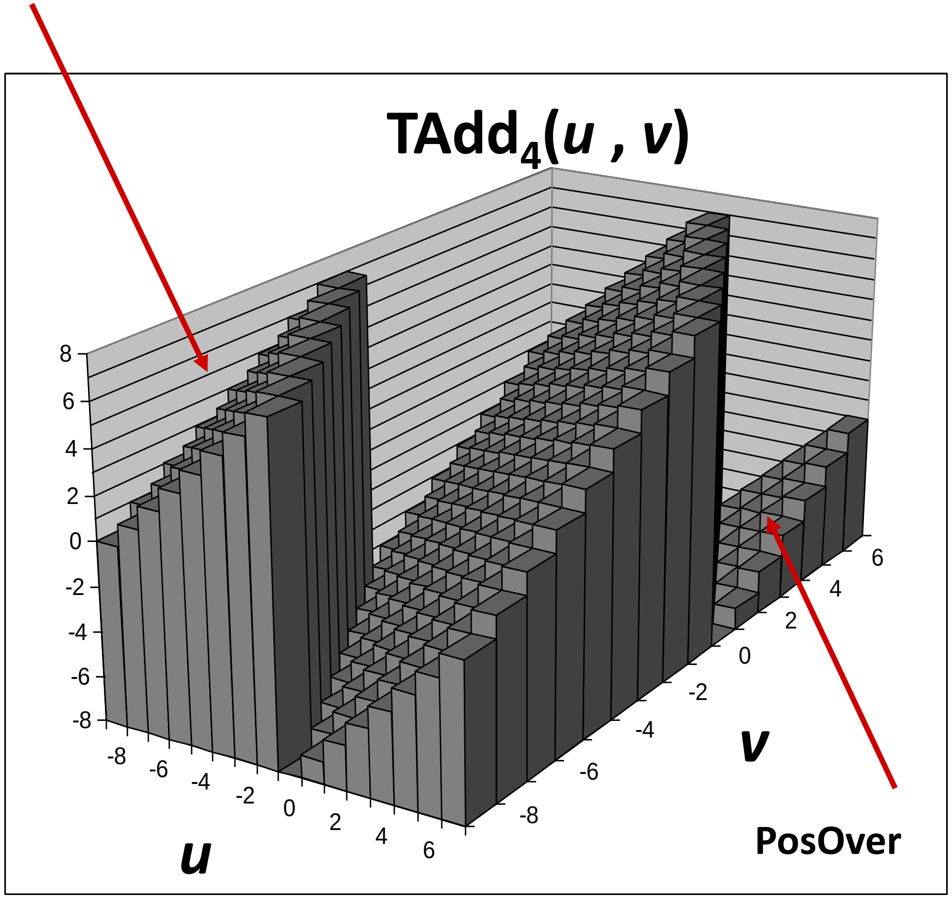
000…0

100…0

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

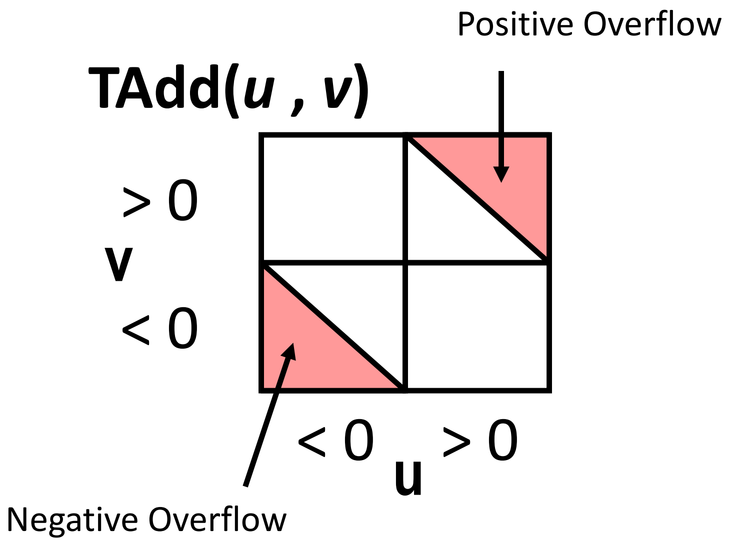
NegOver



PosOver

Characterizing TAdd

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



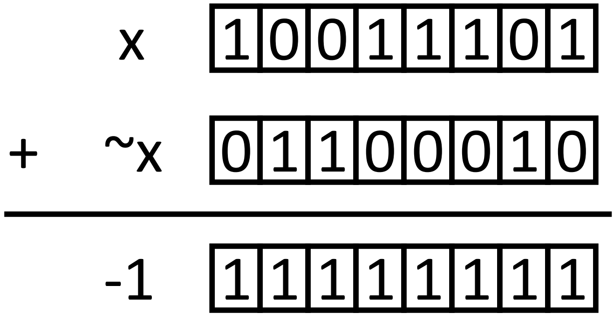
$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

Negation: Complement & Increment

- **Negate through complement and increase**

  `~x + 1 == -x`

- **Example**
  - Observation: `~x + x == 1111…111 == -1`



x = 15213

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`~x`	-15214	`C4 92`	`11000100 10010010`
`~x+1`	-15213	`C4 93`	`11000100 10010011`
`y`	-15213	`C4 93`	`11000100 10010011`

Complement & Increment Examples

x = 0

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

x = TMin

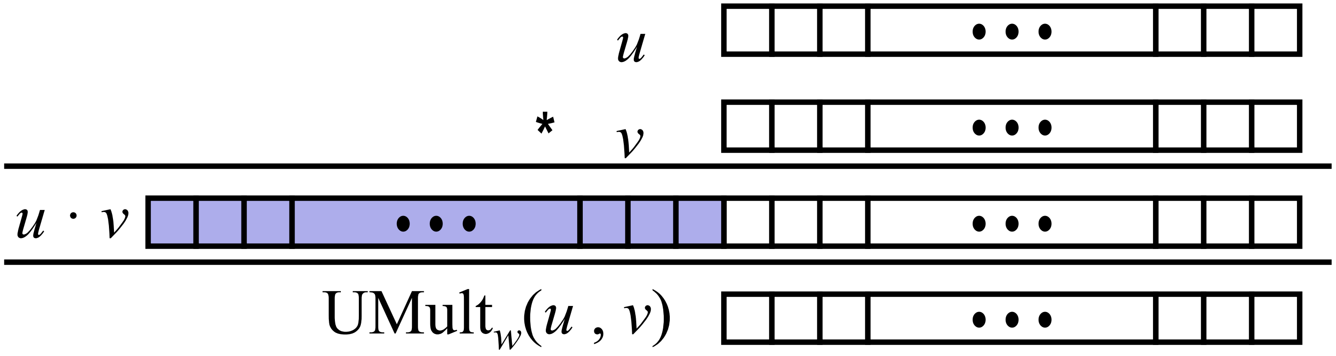
	Decimal	Hex	Binary
`x`	**-32768**	`80 00`	`10000000 00000000`
`~x`	**32767**	`7F FF`	`01111111 11111111`
`~x``+1`	**-32768**	`80 00`	`10000000 00000000`

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**
  - 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

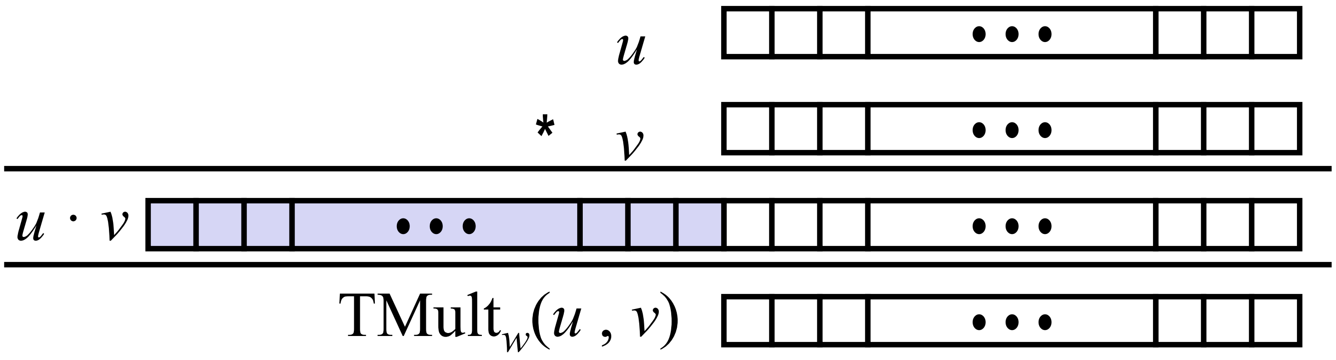
Discard *w* bits: *w* bits

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

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

1110 1001	E9	223
* 1101 0101	* D5	* 213
1100 0001 1101 0010	C1DD	47499
1101 1101	DD	221

Signed Multiplication in C



Operands: *w* bits u $* \; v$

True Product: 2**w* bits $u \cdot v$

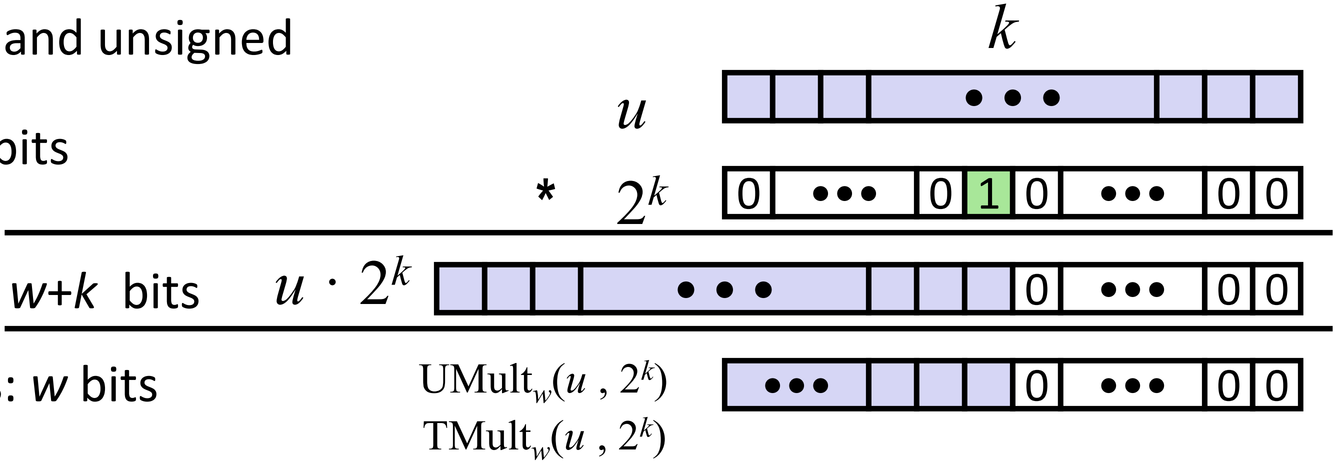
Discard *w* bits: *w* bits $\mathrm{TMult}_w(u\, ,\, v)$

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

```
                1110 1001        E9        -23
*               1101 0101    *   D5   *    -43
      1100 0001 1101 0010      C1DD      16896
                1101 1101        DD        -35
```

Power-of-2 Multiply with Shift

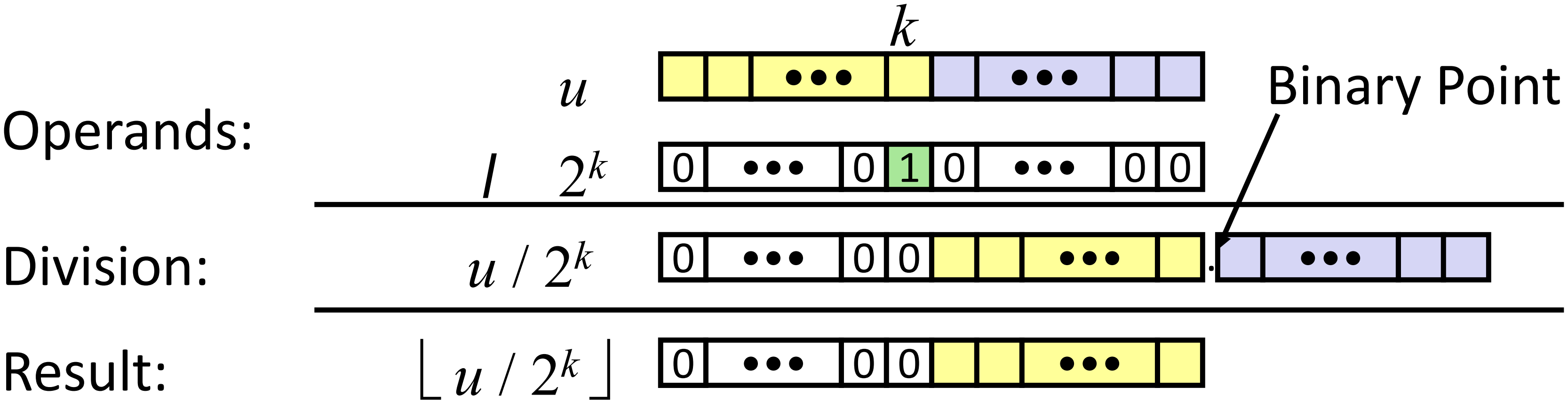
- **Operation**
  - `u << k` gives `u * 2`^{k}
  - Both signed and unsigned



- **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

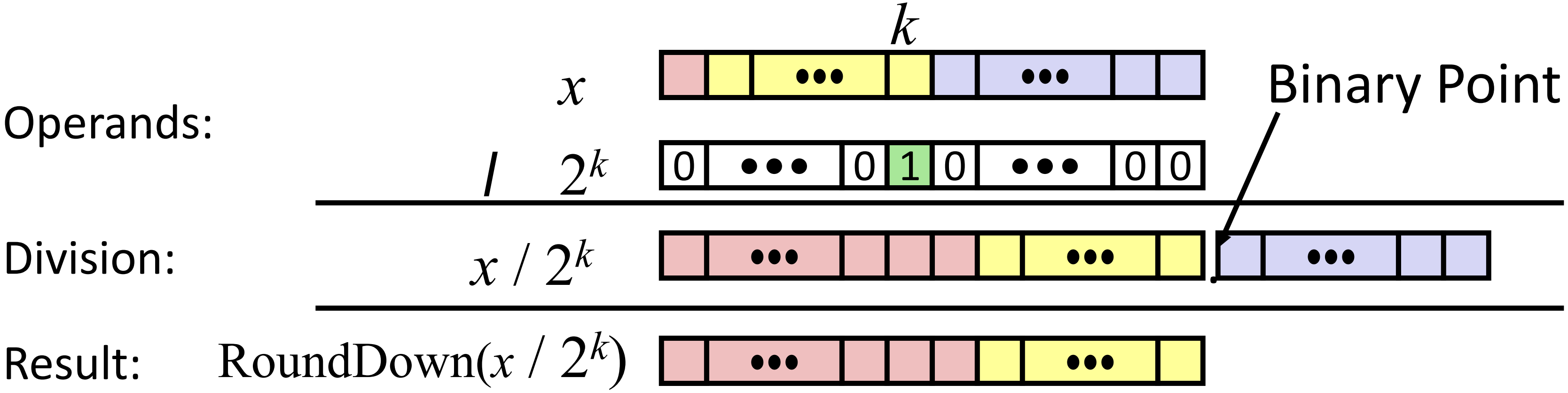
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses logical shift



	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`

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 when `u < 0`



k

Operands: x ••• •••

$/\ 2^k$ 0 ••• 0 1 0 ••• 0 0

Binary Point

Division: $x / 2^k$ ••• ••• •••

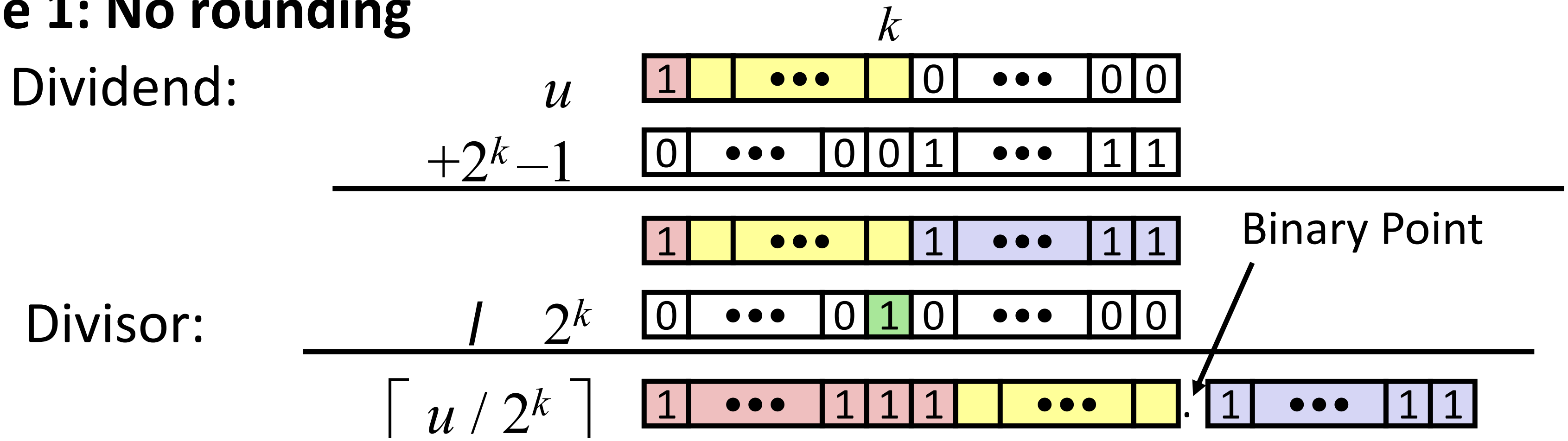
Result: RoundDown($x / 2^k$) ••• •••

	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want $\lceil \mathtt{x} \;/\; 2^k \rceil$ (Round Toward 0)
  - Compute as $\lfloor (\mathtt{x}+2^k-1) / 2^k \rfloor$
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0

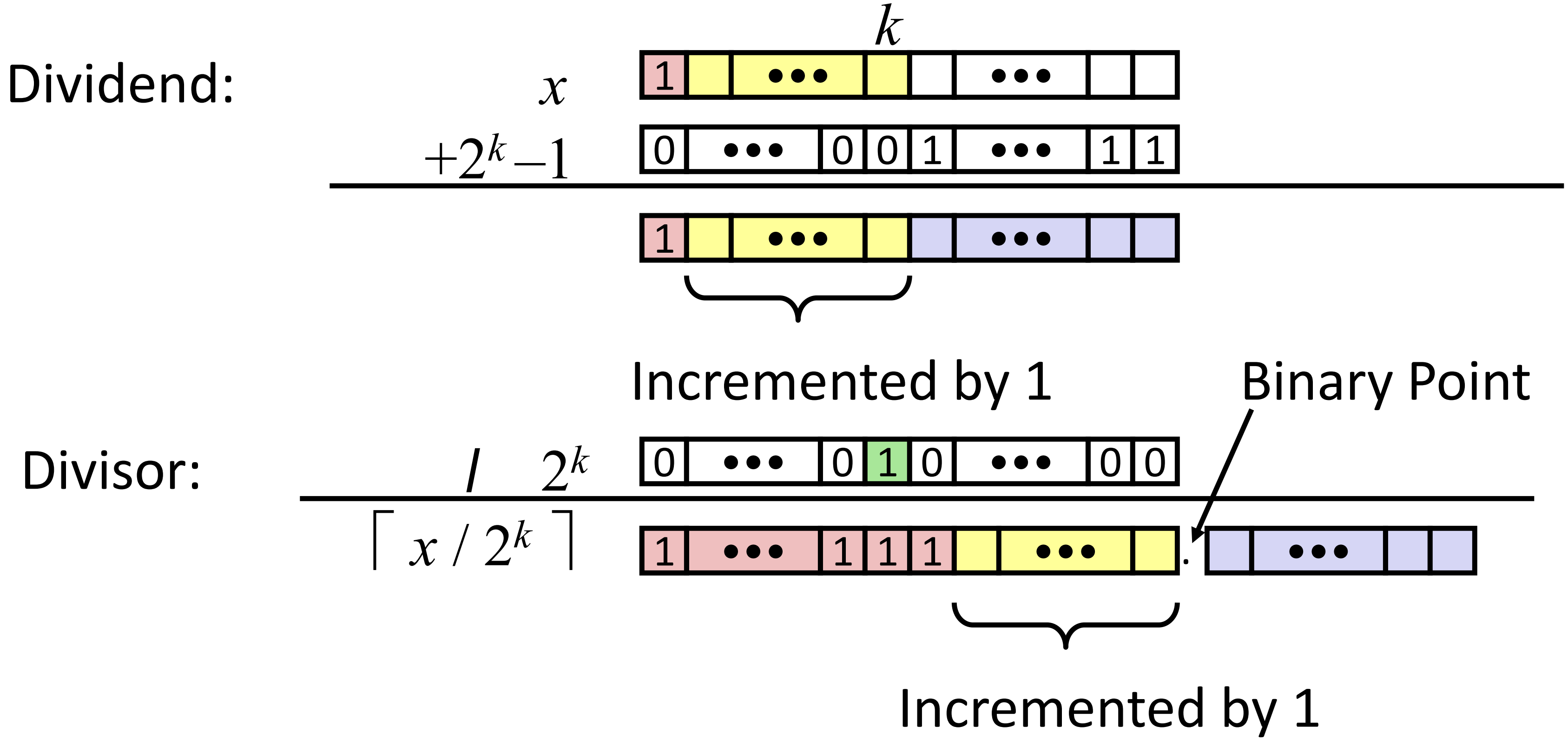
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

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];
```

  - 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?

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
- ***Do*** **Use In System Programming**
  - Bit masks, device commands,…

Integer Arithmetic Example

```
unsigned char

    1111 0011      F3       243
+   0101 0010    + 52     +  82
-------------    ----     -----
  1 0100 0101     145       325
-------------    ----     -----
    0101 0101      45        69
```

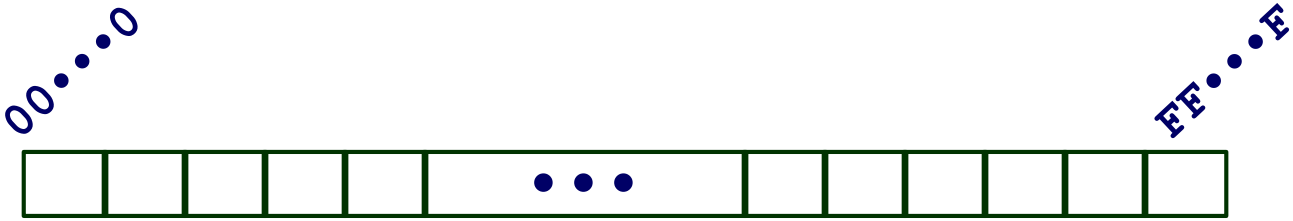
```
unsigned char

    0001 1001      19       25
*   0000 0010    * 02     *  2
-------------    ----     ----
  0 0011 0010     032       50
-------------    ----     ----
    0011 0010      32       50
```

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

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, 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

Machine Words

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

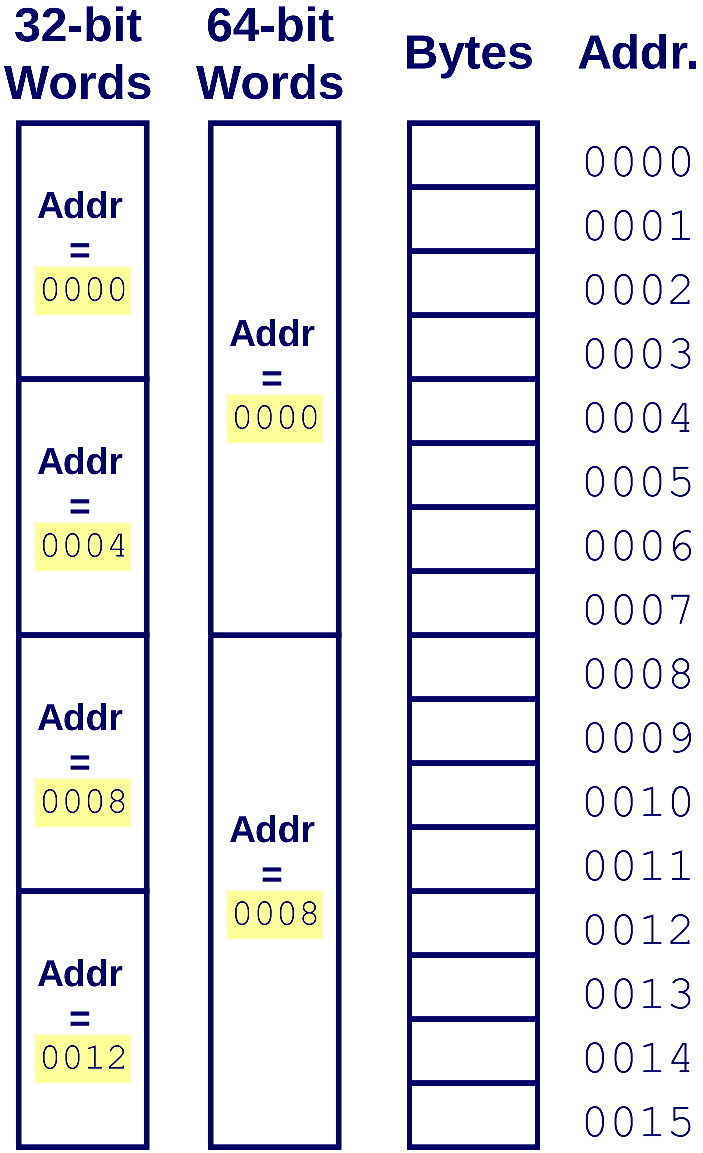
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)

  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That’s 18.4 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)



32-bit Words

64-bit Words

Bytes

Addr.

Addr = 0000

Addr = 0004

Addr = 0008

Addr = 0012

Addr = 0000

Addr = 0008

0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015

Example Data Representations

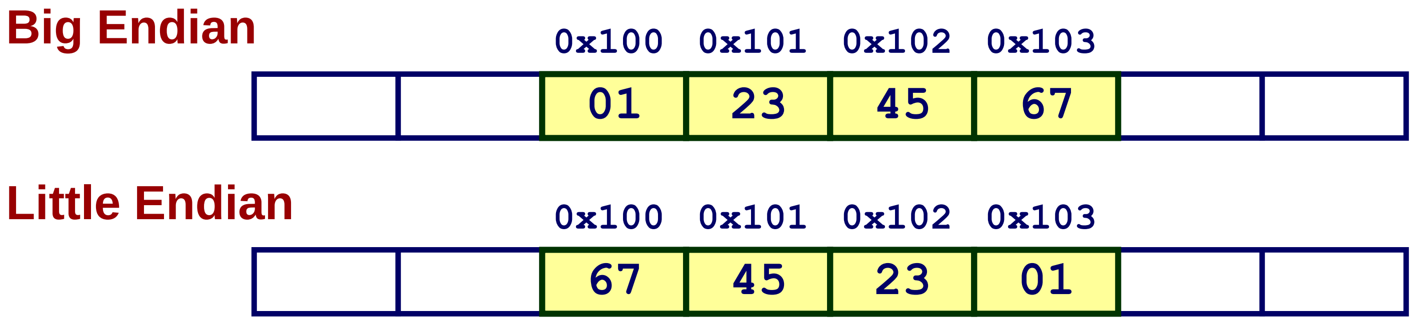
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

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		

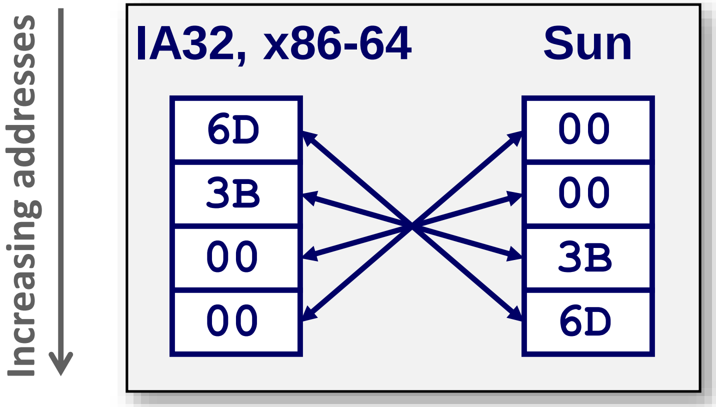
Representing Integers

Decimal: 15213

Binary: 0011 1011 0110 1101

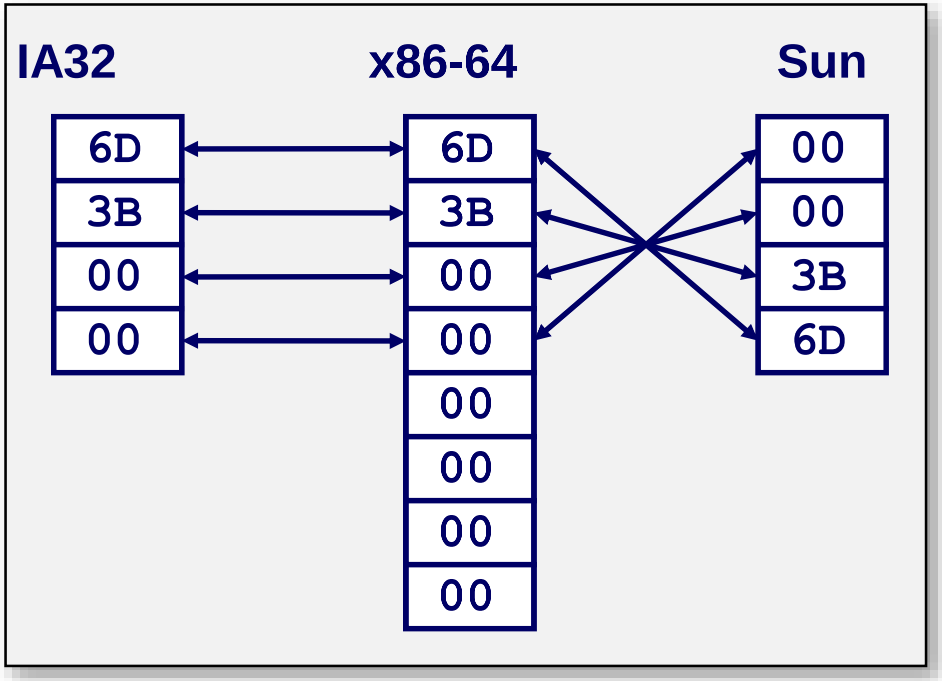
Hex: 3 B 6 D

`int A = 15213;`



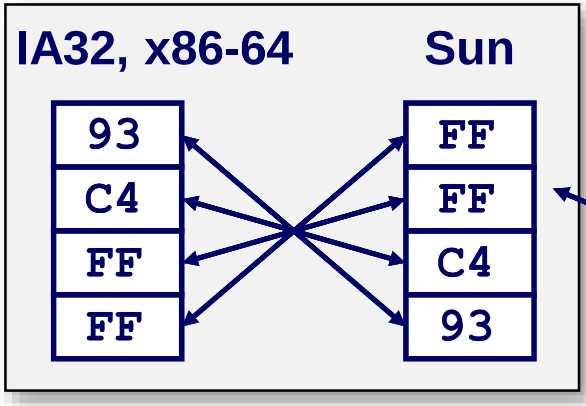
IA32, x86-64	Sun
6D	00
3B	00
00	3B
00	6D

`long int C = 15213;`



IA32	x86-64	Sun
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

`int B = -15213;`



IA32, x86-64	Sun
93	FF
C4	FF
FF	C4
FF	93

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");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

`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
```

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

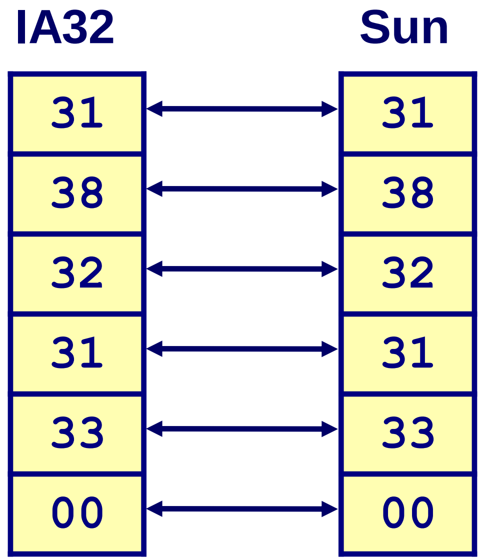
Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

```
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

Reading Byte-Reversed Listings

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

```
Address      Instruction Code          Assembly Rendition
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 32 bits: `0x000012ab`
  - Split into bytes: `00 00 12 ab`
  - Reverse: `ab 12 00 00`

Integer C Puzzles

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

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

Summary

- 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