18-600 Foundations of Computer Systems

Lecture 3: "Bits, Bytes, and Integers"

John Shen & Zhiyi Yu

September 7, 2016

- Required Reading Assignment:
  - **Chapter 2 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron**
- Assignments for This Week:
  - Lab 1

Electrical & Computer ENGINEERING

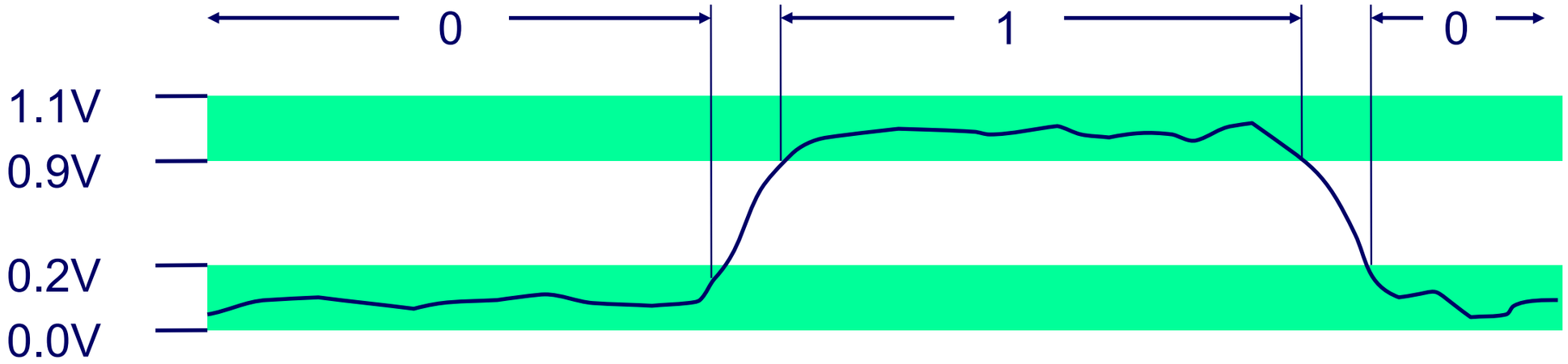
Carnegie Mellon University

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion (casting), expanding
  - Addition, multiplication, shifting
- Representations in memory, pointers, strings

Everything is bits

- Everything in computers including instructions and data are bits
- Each bit is 0 or 1, represented by low or high voltages
- Why bits? Reliability of Electronic Implementation
  - Easy and reliable to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Encoding Byte Values

- Hex is often used to represent data
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Start as 0x in C Language. Example. $FA1D_{16}$ as 0xfa1d
- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}

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

Example Data Representations in Byte

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

Today: Bits, Bytes, and Integers

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

Boolean Algebra, well matched to digital circuits

- Developed by George Boole in 19th Century

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

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

^	0	1
0	0	1
1	1	0

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`
    - $\sim 01000001_2$ ☞ 10111110_2
  - `~0x00` ☞ `0xFF`
    - $\sim 00000000_2$ ☞ 11111111_2
  - `0x69 & 0x55` ☞ `0x41`
    - 01101001_2 & 01010101_2 ☞ 01000001_2
  - `0x69 | 0x55` ☞ `0x7D`
    - 01101001_2 | 01010101_2 ☞ 01111101_2

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&, ||, !`
    - View 0 as "False"
    - Anything
    - Alway
    - Early
- Examples
  - `!0x41`
  - `!0x00`
  - `!!0x4`
  - `0x69 && 0x55` ☞ `0x01`
  - `0x69 || 0x55` ☞ `0x01`

Watch out for && vs. & (and || vs. |)… one of the more common oopsies 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	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Today: Bits, Bytes, and Integers

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

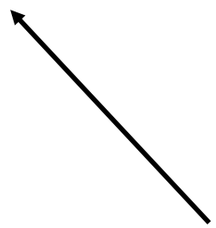
Encoding Integers

Unsigned

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

Signed: Two's Complement

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



- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative
- Equation: x + (-x) = 0
- C `short` (2 bytes)

```
short int x =  100;
short int y = -100;
```

	Decimal	Hex	Binary
x	100	00 64	00000000 01100100
y	-100	FF 9C	11111111 10011100

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- Observations
  - $|TMin|$ = $TMax + 1$
  - *UMax* = $2 * TMax + 1$

- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1

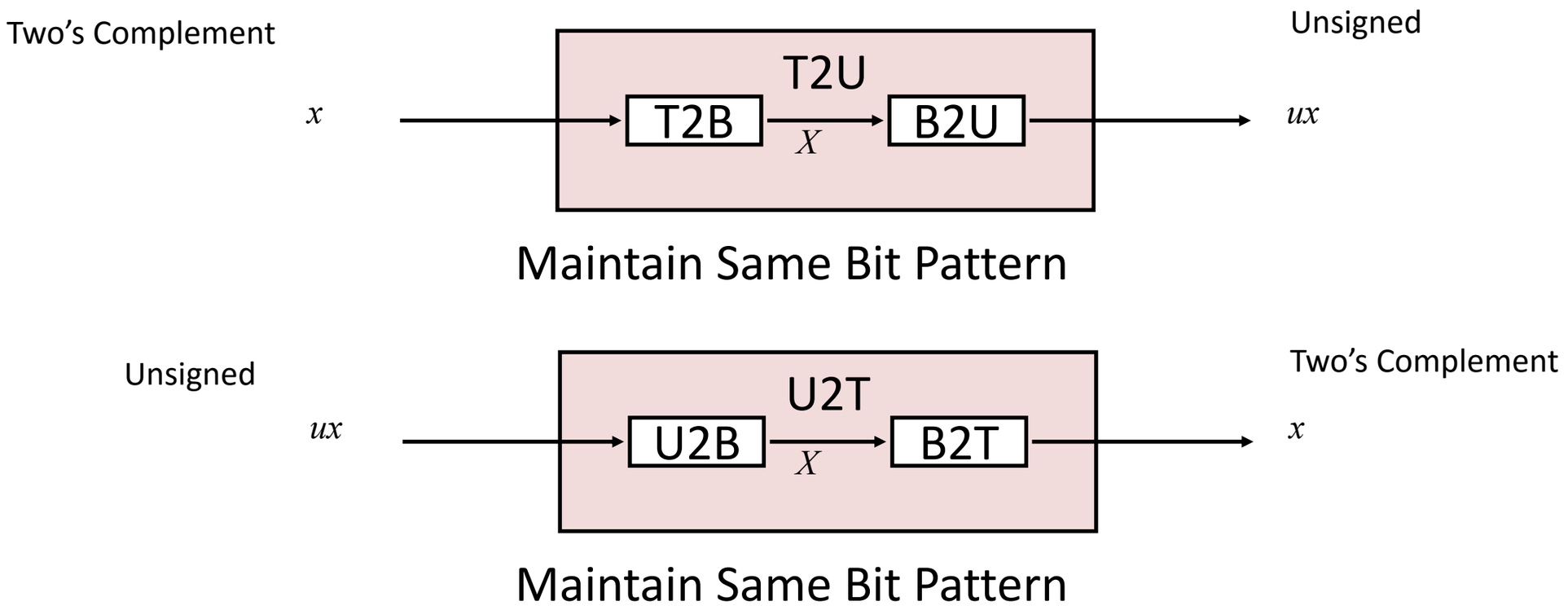
Values for $W = 16$

	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

Today: Bits, Bytes, and Integers

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

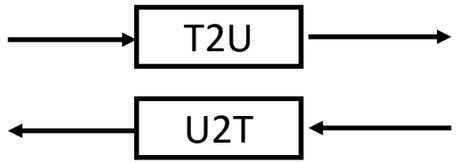
Mapping Between Signed & Unsigned



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

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



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

Signed vs. Unsigned in C

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

    `0U, 4294967259U`
- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

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

  - Implicit casting also occurs via assignments and procedure calls

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

Casting Surprises

- 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

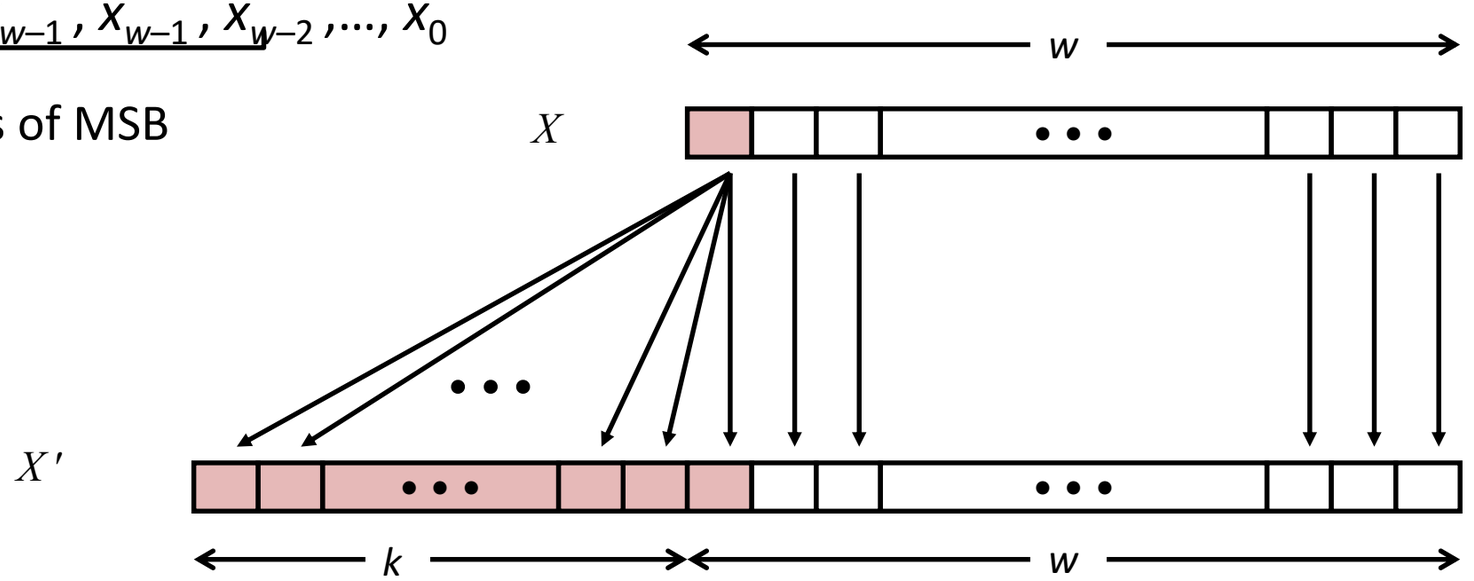
Another Casting Surprise

```
unsigned i;
for (i=n-1; i>=0; i--)
  for (a[i]);



signed i;
for (i=n-1; i - sizeof(char)>=0; i--)
  for (a[i]);
```

Sign Extension

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

k copies of MSB



Today: Bits, Bytes, and Integers

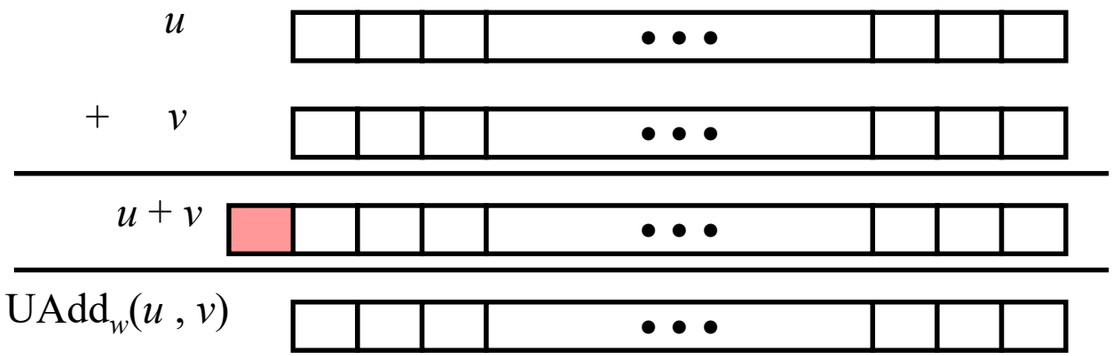
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion (casting), expanding
  - **Addition, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



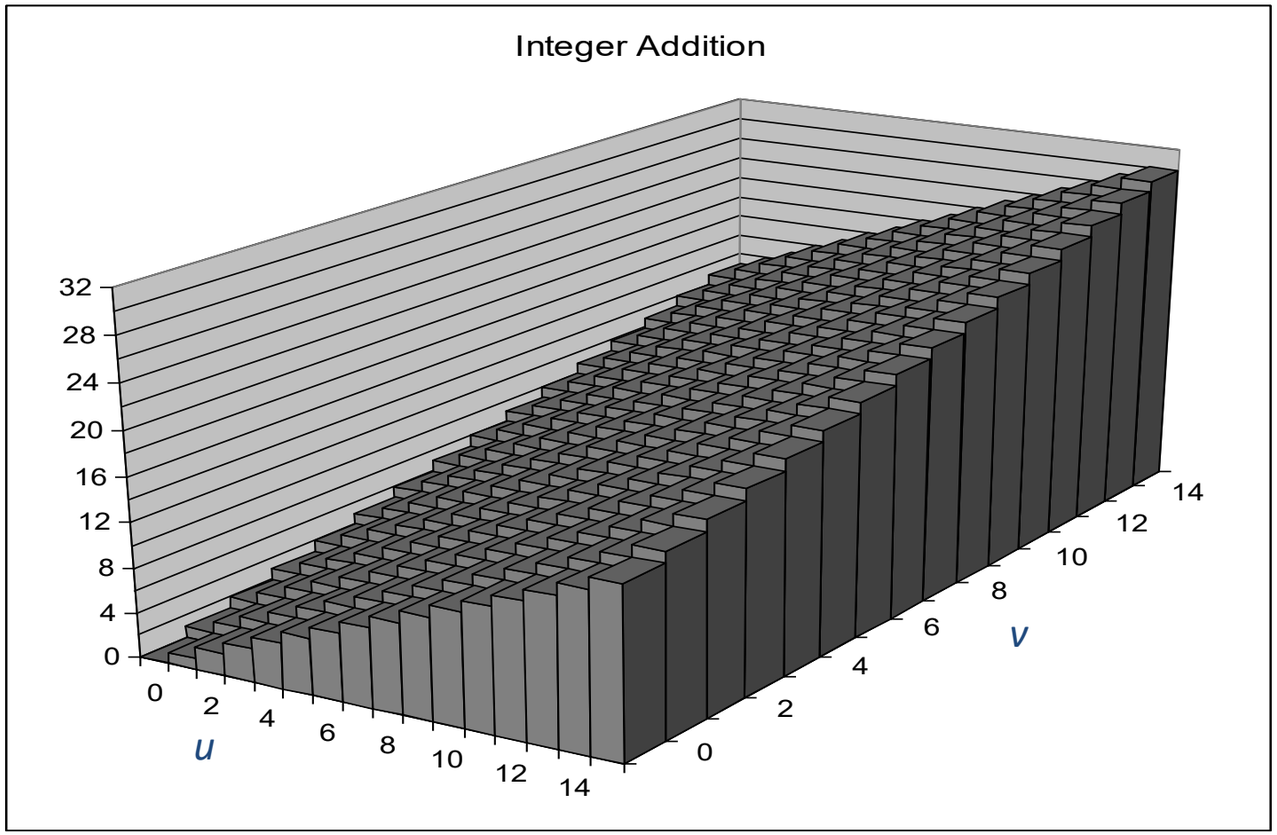
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

Visualizing (Mathematical) Integer Addition

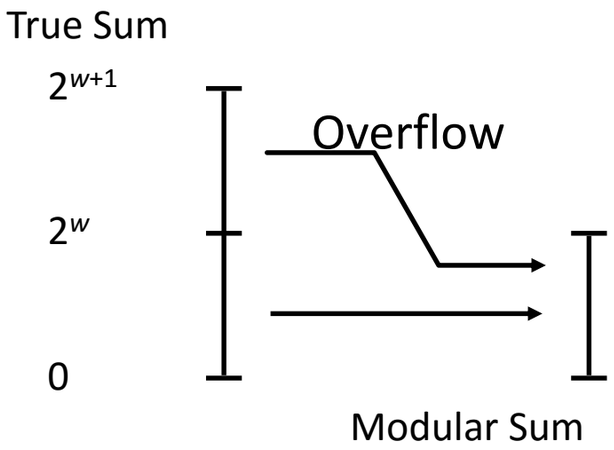
- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u , v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

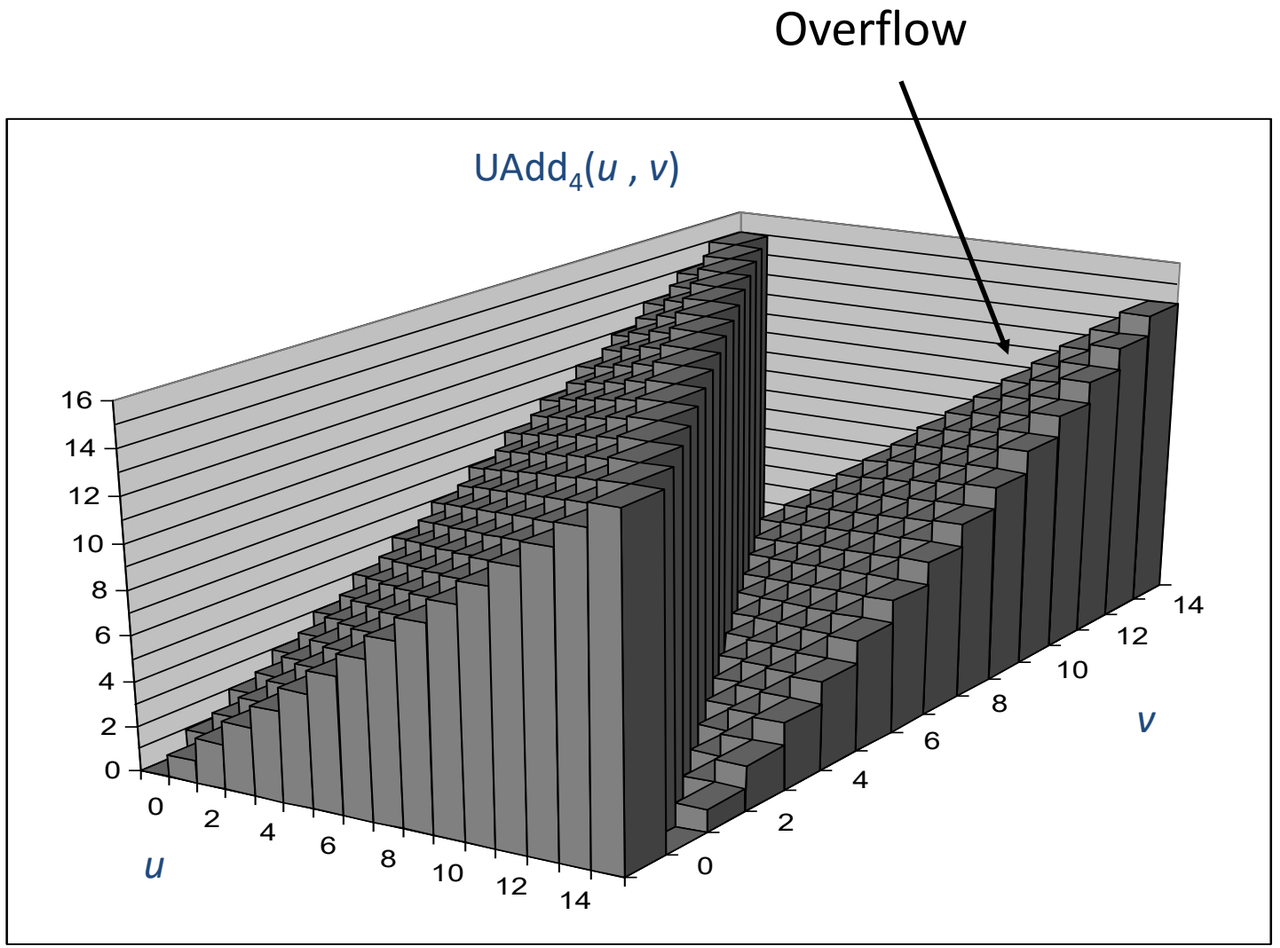
$\text{Add}_4(u , v)$



Visualizing Unsigned Addition

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



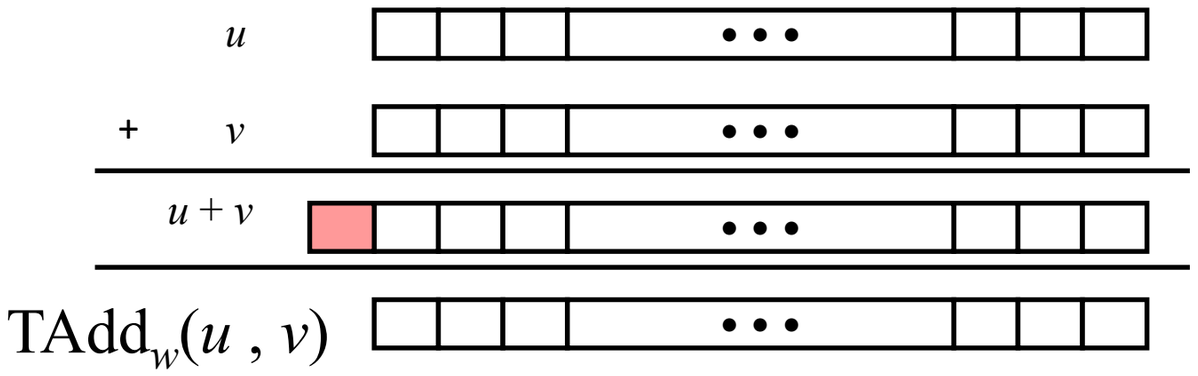


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



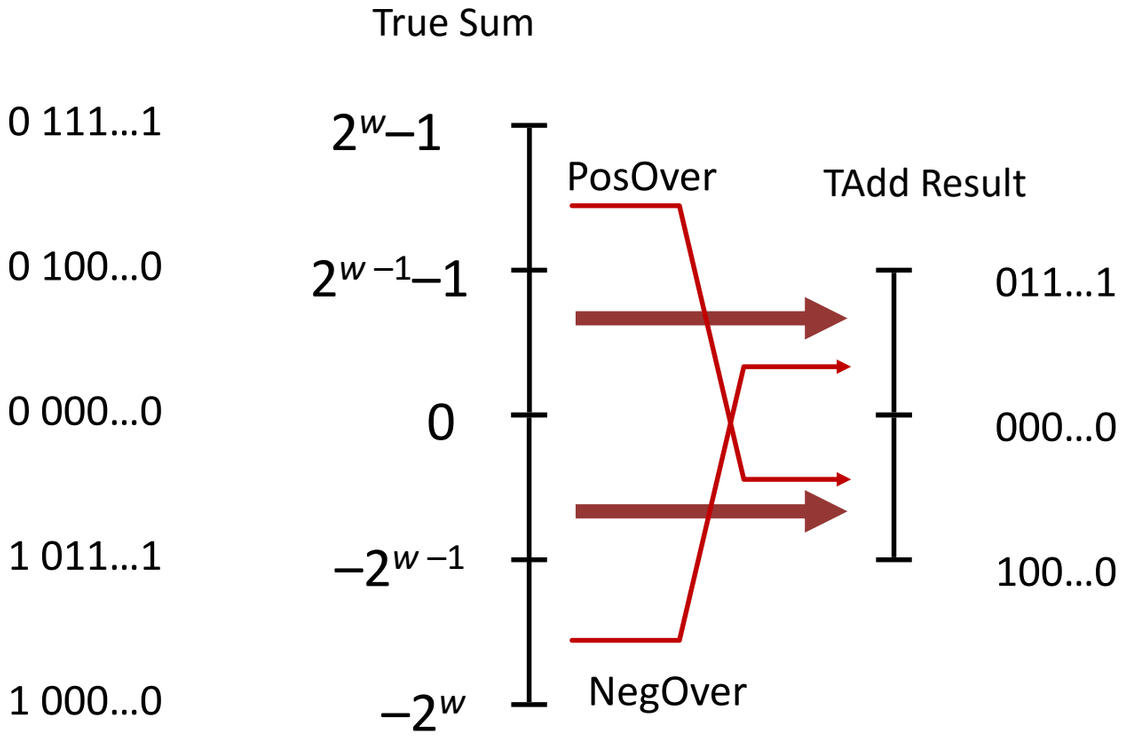
- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:

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

  - Will give `s == t`

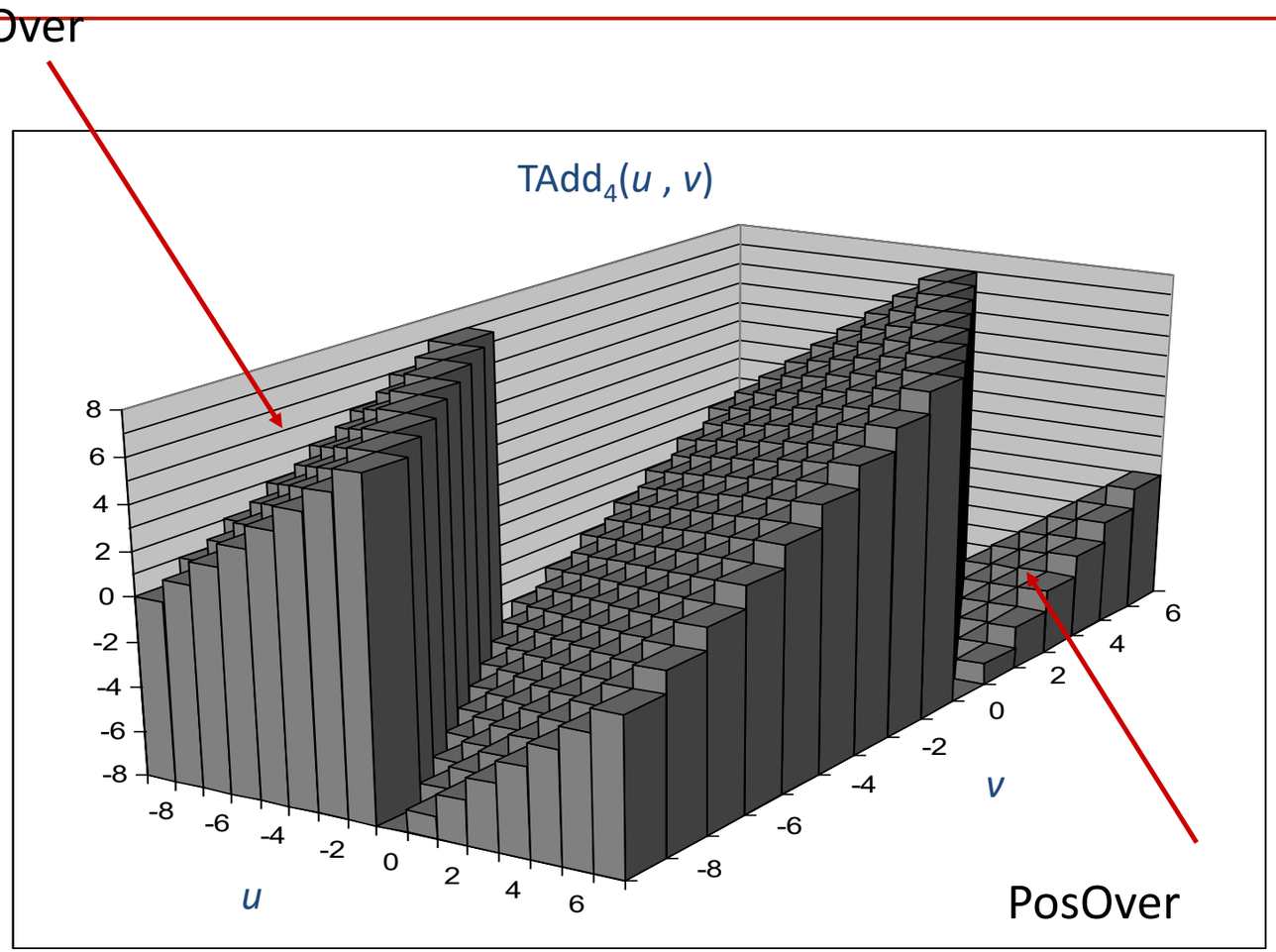
TAdd Overflow

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



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



Handling overflow (saturation)

- Some processors (especially DSPs) have special instructions to handle overflow (saturation) situations
- Example (w=4):
  - Normal instruction 0111 + 1 = 1000 -> 7 + 1 = -8
  - Instructions handling overflow: 0111 + 1 = 0111 -> 7+1=7

Multiplication

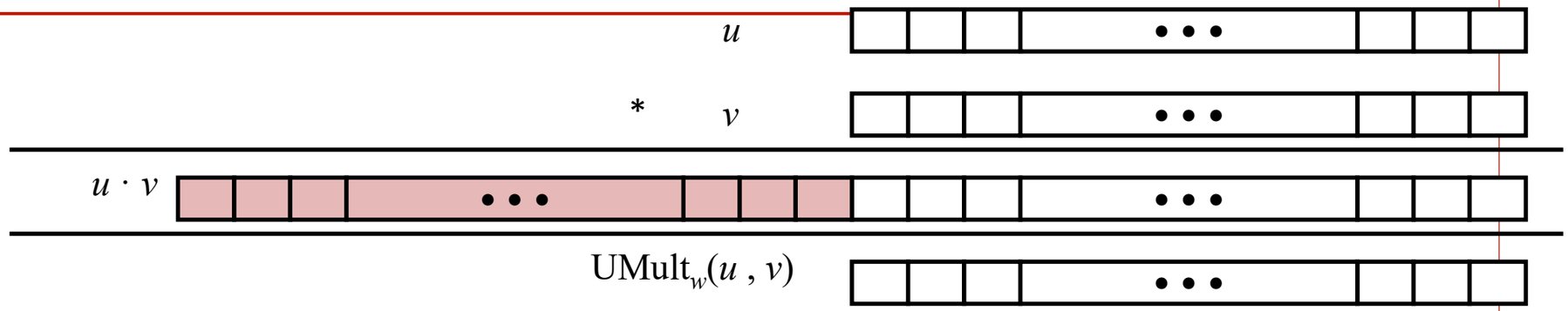
- Goal: Computing Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result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

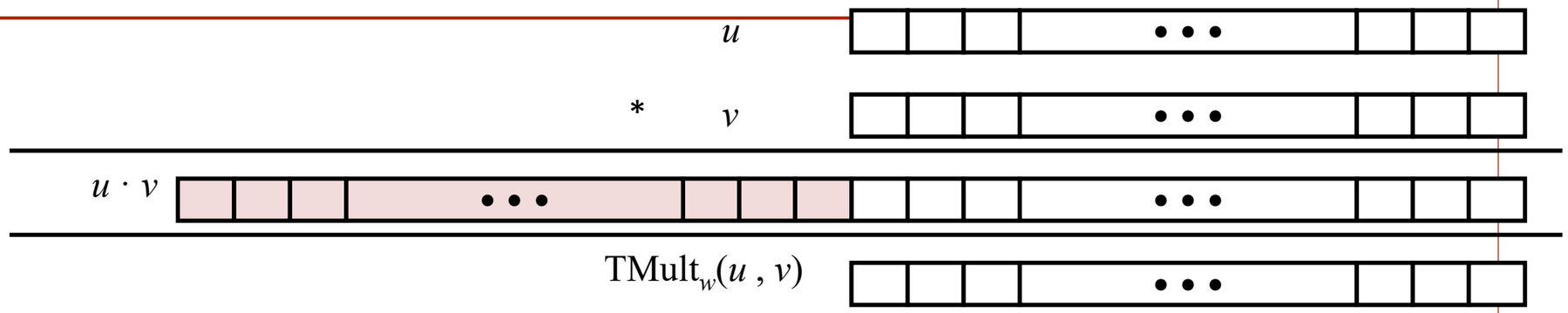
  $\text{UMult}_w(u , v) = u \cdot v \bmod 2^w$

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

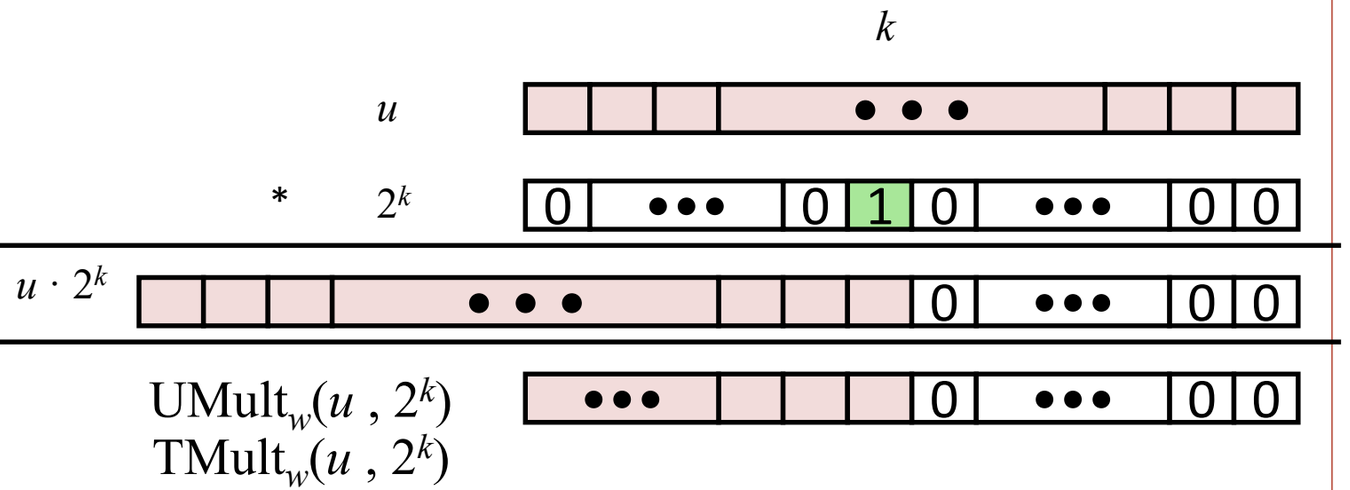
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w*+*k* bits

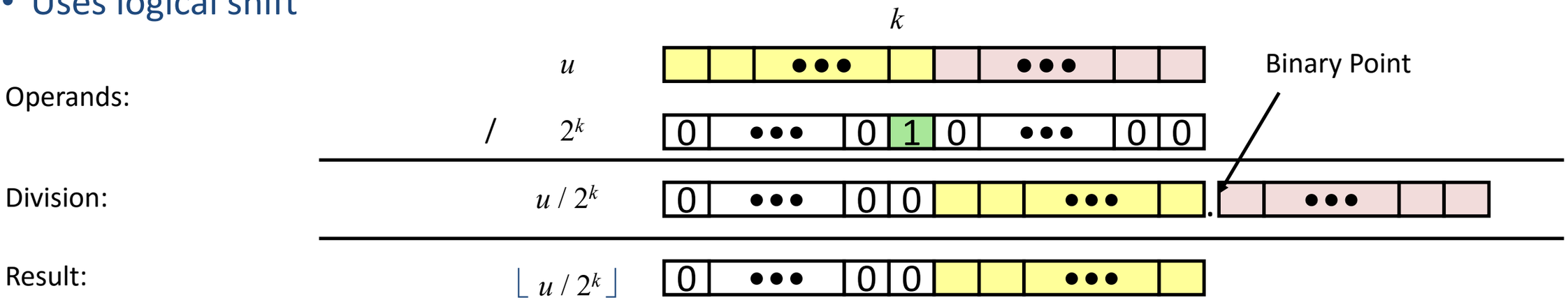
Discard *k* bits: *w* bits



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

Unsigned Power-of-2 Divide with Shift

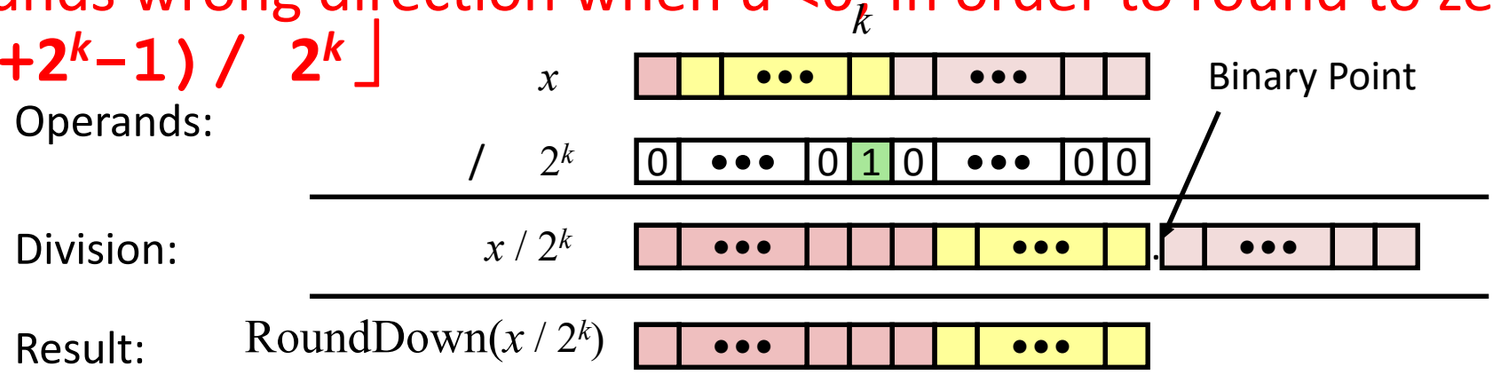
- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor$ `u / 2^k` $\rfloor$
  - Uses logical shift



	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; In order to round to zero, Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$

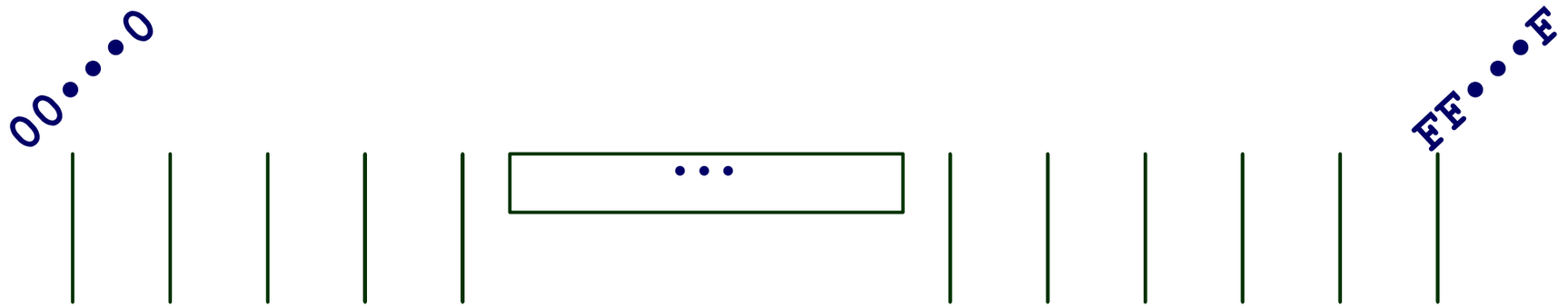


	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion (casting), expanding
  - Addition, multiplication, shifting
- 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

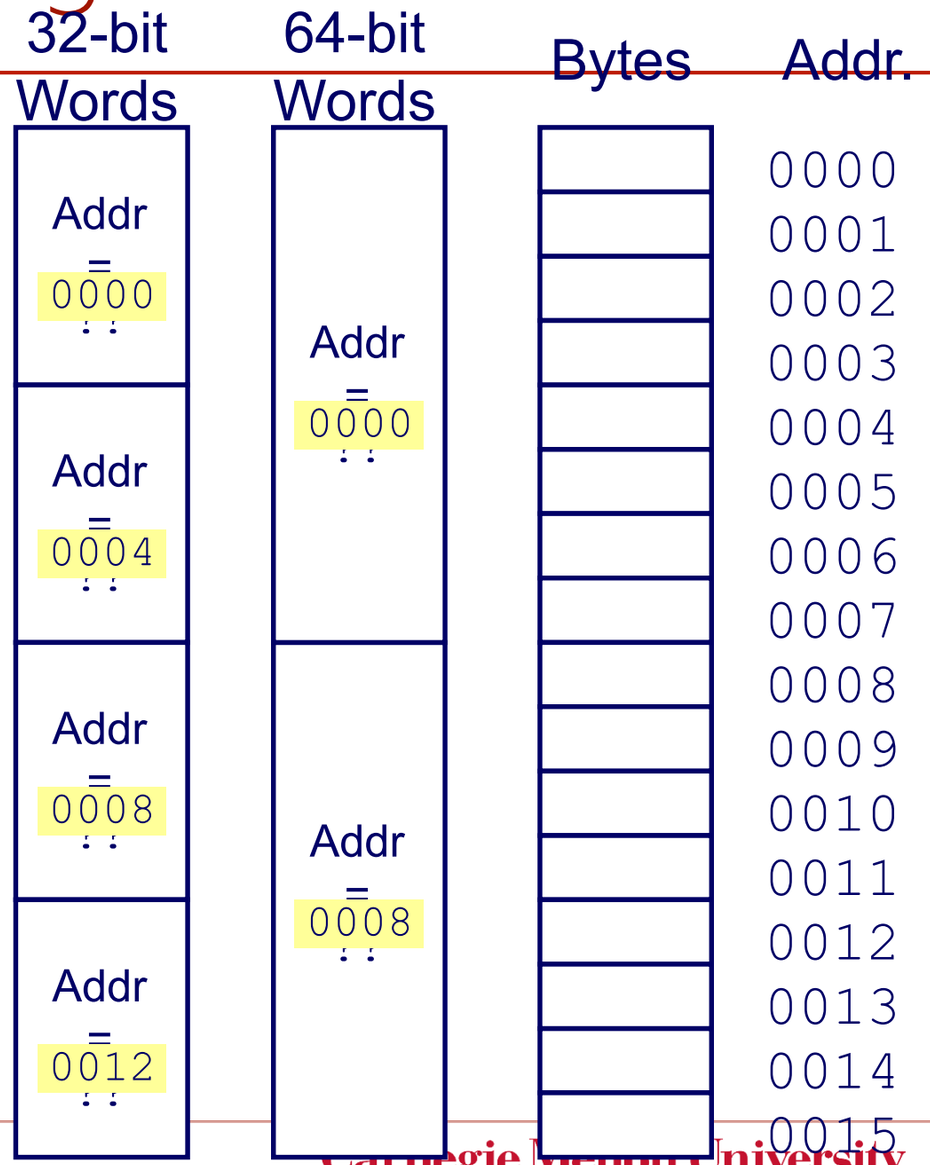
- Note: system provides private address spaces to each "process"
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

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)



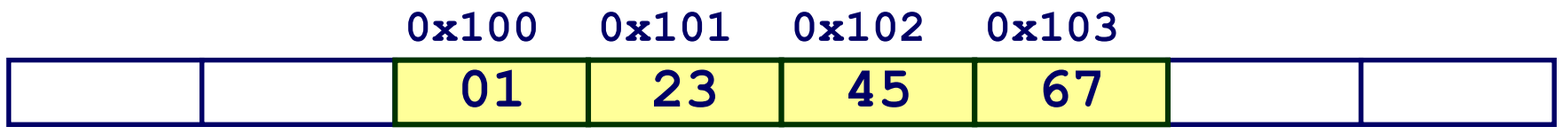
Byte Ordering

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

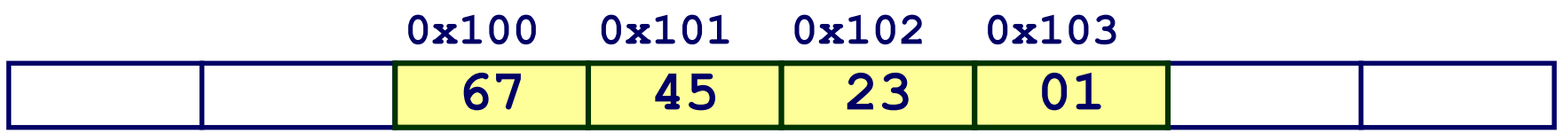
Byte Ordering Example

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

Big Endian

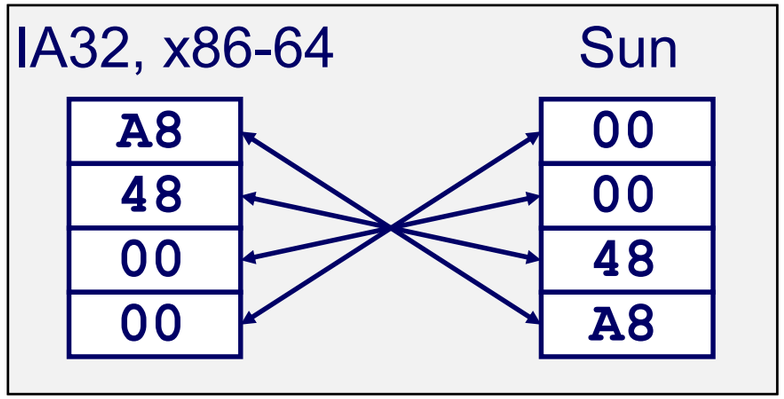


Little Endian

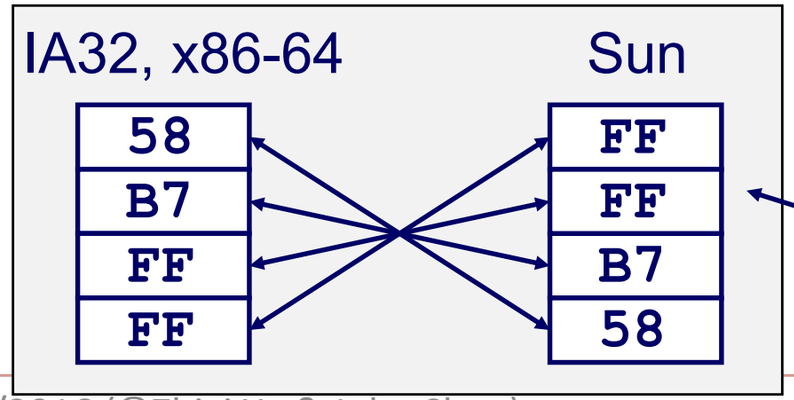


Representing Integers

```
int A = 18600;
```




```
int B = -18600;
```

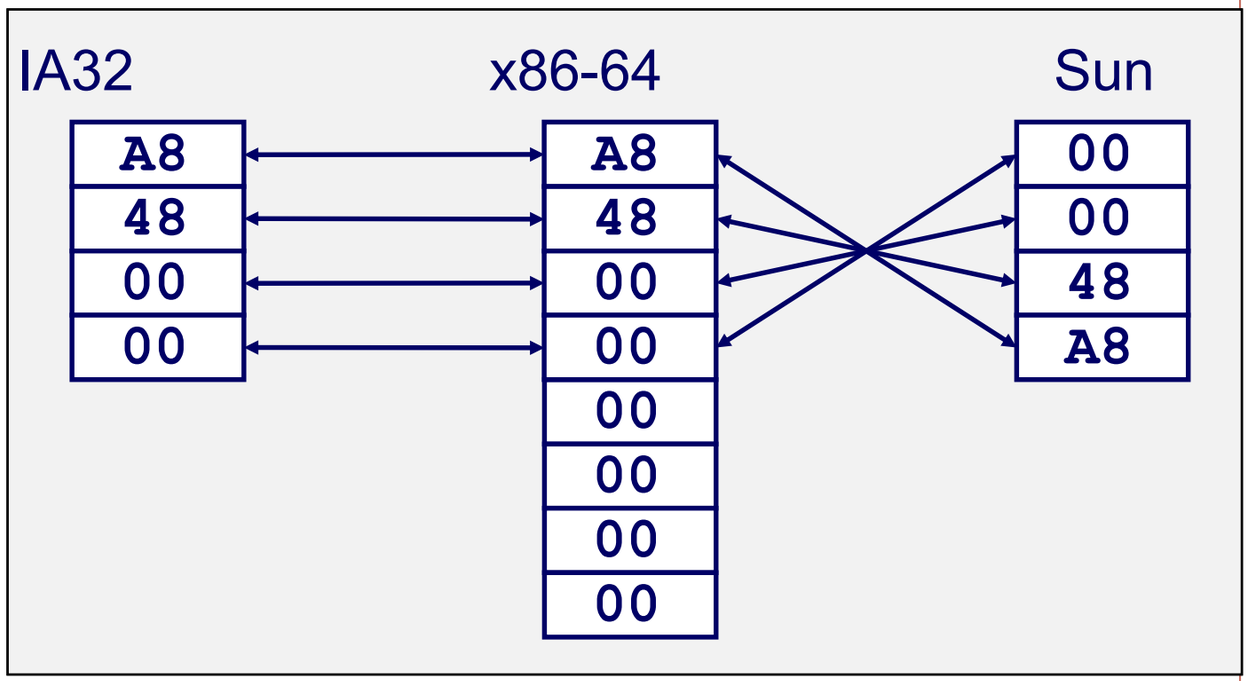


Decimal: **18600**

Binary: **0100 1000 1010 1000**

Hex: **4 8 A 8**

```
long int C = 18600;
```



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

`%t`: Tab space

`%x`: Print Hexadecimal

`show_bytes` Execution Example

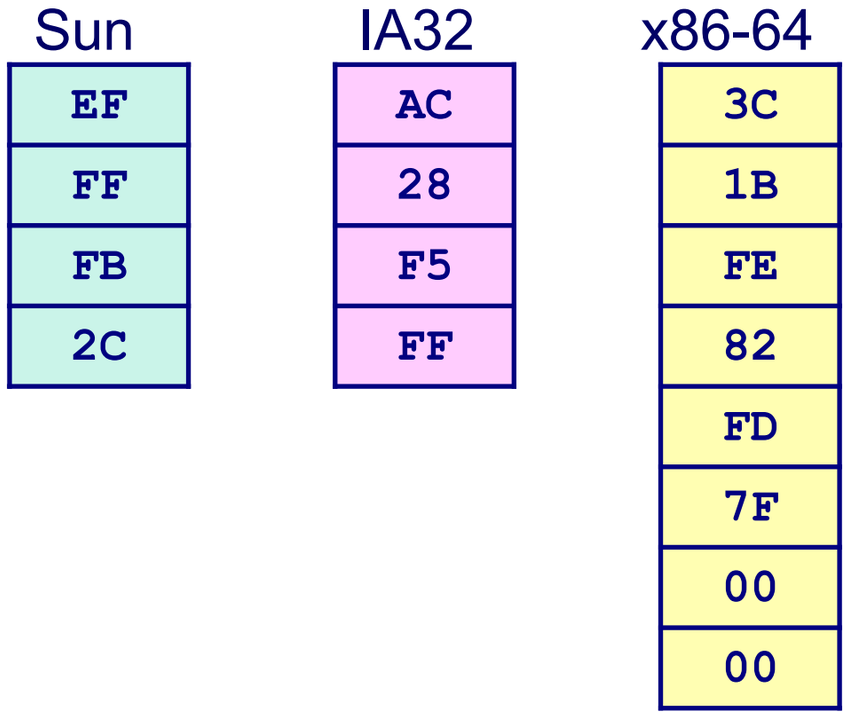
```
int a = 18600;
printf("int a = 18600;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 18600;
0x7fffb7f71dbc    A8
0x7fffb7f71dbd    48
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

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



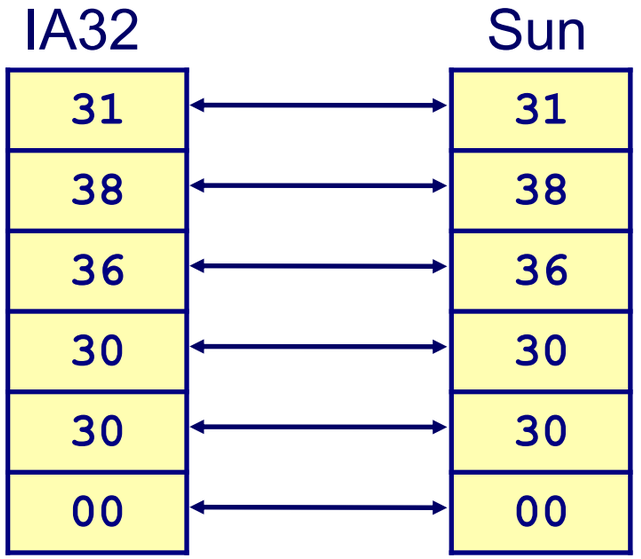
Different compilers & machines assign different locations to objects

Even get different results each time run 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] = "18600";
```



Integer C Puzzles

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

`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`			✗

18-600 Foundations of Computer Systems

Lecture 4: "Floating Point"

September 12, 2016 (September 13 in GZ)

Next Time ...

Electrical & Computer ENGINEERING