18-600 Foundations of Computer Systems

Lecture 3:
"Bits, Bytes, and Integers"

September 6, 2017

- 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

Socrative Experiment

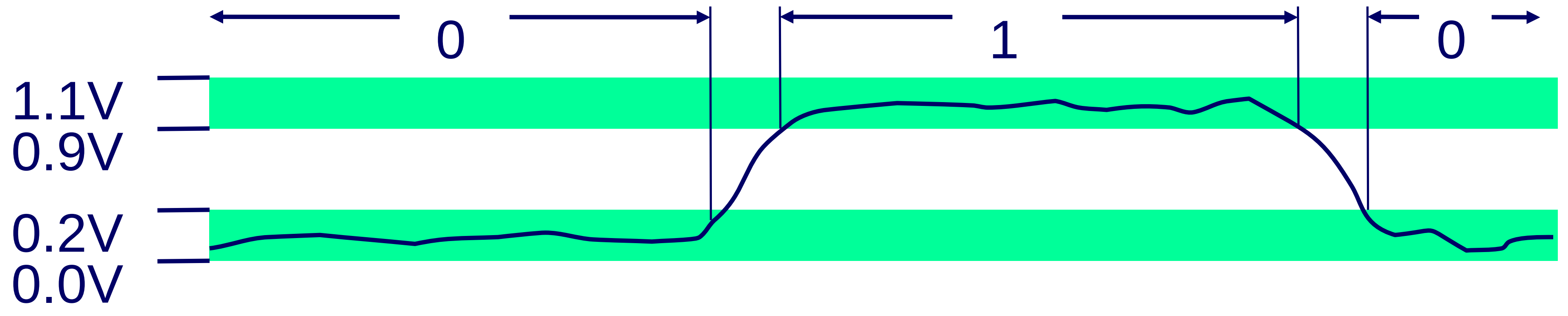
- Pittsburgh Students (18600PGH): https://api.socrative.com/rc/icJVVC
- Silicon Valley Students (18600SV): https://api.socrative.com/rc/iez85z
- Microphone/Speak out/Raise Hand: Still G-R-E-A-T!
- Socrative:
  - Let's me open floor for electronic questions, putting questions into a visual queue so I don't miss any
  - Let's me do flash polls, etc.
  - Prevents cross-talk and organic discussions in more generalized forums from pulling coteries out of class discussion into parallel question space.
    - Keeps focus and reduces distraction while adding another vehicle for classroom interactivity.
  - Won't allow more than 150 students per "room"
    - So, I created one room per campus
    - May later try random assignment to a room, etc.

Today: Bits, Bytes, and Integers

- Representing information as bits
- 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 (*bi*nary dig*its*)**
- The binary (two) digits are 0 and 1, represented by low or high voltages
- Why bits (digital) vs continuous (analog)?
  - Easier to tell "on" vs "off" than 18.3% vs 22.5%, etc.
  - Especially true once wires act as antennas and pick up extraneous signals and also act as resistors and lose data signal. Precise levels become noisy. Signal-noise ratio (SNR) can go from high (good) to low (bad), but in the real world always needs to be a "tolerance" for noise.



Power-of-two bases Group Binary Nicely

- Base-2 (Binary) groups 1 bit (0-1) into 2 digits (0,1)
- Base-4 groups 2 bits (00-11) into 4 digits (0, 1, 2, 3)
- Base-8 (Octal) groups 3 bits (000-111) into 8 digits (0, 1, 2, 3, 4, 5, 6, 7)
- **Base-16 (Hexadecimal) groups 4 bits (0000-1111) into 16 digits**
  (0, 1,2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
  - Letters conventionally used past 0-9. They're familiar and we know the order.

Power-of-two Bases, Example Grouping

- Consider 0111111101011010 (Base-2)

  - 01 11 11 11 01 01 10 10
    1 3 3 3 1 1 2 2 (Base-4)

  - *00*0 111 111 101 011 010 (Base-2)
    0 7 7 5 3 2 (Base-8, Octal)
    - Note leading 0s don't change value. They just fill out grouping.
    - **Important to group from the right.**

  - ***0*011 1111 1010 1101 (Base-2)**
    3 F A D (Base-16, Hexadecimal)
    - **Note leading 0s don't change value. They just fill out grouping.**
    - **Important to group from the right.**

Octal and (Mostly) Hexadecimal Best Choices

- They have "approximately" as many digits as decimal
  - Convenient for humans.
- Fewer digits means longer numbers, which are harder for humans
- More digits means shorter numbers, but it is hard for humans to keep track of more digits to interpret the numbers and numbers that group too many bits are harder to keep track of and break down to manipulate.
- **"Hex" is most common because, in practice, it is most convenient balance of complexity and length.**

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
  - Addition, multiplication, shifting
- Representations in memory, pointers, strings
- Summary

Representing Positive and Non-Positive Numbers

- Non-negative values are straight-forward to represent.
  - Read bit values directly as powers of 2 and add together
  - But, how to represent a negative number?
- Can reserve left-most bit to represent minus sign: 0 (non-negative), 1 (Negative)
  - 1010 represents -2
  - Maximum range is -7 to +7, +/- 0 values (1000, 0000)
  - Bit pattern is discontinuous, which special cases arithmetic, e.g. -0+1=0 and (7+1=0),etc.
- **Use "2s complement" to represent negative numbers**
  - Represent negative numbers as complement of number plus 1
    - E.g. -5 = (~0101 + 1) = (1010 + 1) = (1011)
  - Addition with negative and positive numbers works, allowing subtraction by addition.
    - 1011 + 0101 = *1* 0000
  - Number line stays clean

Let's play with binary arithmetic

- We're building up to why this "weird 2s complement thing" works

- Assumption:
  - Computers have finite memory. Numbers have finite sizes, e.g. a fixed number of bits.
  - For this example, we assume 4-bit integers (Real systems typically have 8-64 bit integers)

- 0000 + 0001 = 0001 (Make sense? Sure it does)
- 0001 + 0001 = 0010 (We still good?)
- …
- **1111 + 0001 = *1* 0000 (Wrap-Around!)**
  - But, we lose the 1 since we only have 4 digits. It is "Carry out", which processors typically store separately in a flag.

New Math! Let's Keep Playing!

- 1111 + 1 = *1* 0000 (Wrap-Around!)
  - But, we lose the 1 since we only have 4 digits. It is "Carry out"
- 1111 + 1 = 0 (Wow! New math!)
  - Remember: We lost the "carry out" since it couldn't fit in 4 digits
- (1111 + 1) - 1 = 0 – 1 (Let's do some algebra)
- **1111 = -1** (Huh. That is curious. Let's roll with it)

That's strange! Can 1111 really represent -1?

- 1111 + 0101 = 0100 $(-1+5 = 4)_{10}$
  - Look right. It worked!
- 1111 + 0100 = 0011 $(-1+4 = 3)_{10}$
  - Still consist as $-1 + 4_{10} = -1 + (5_{10} -1)$ and 0011 = (0100 -1)
- 1111 + 0001 = 0 (We expect -1+1 = 0. Note carry-out)
  - Look right. It worked!
- 0 + –1111 = 0 – 1111 = -1111 (Okay, still consistent, 0 – 1 = -1)
- **Yes, yes, 1111 can really represent -1!**

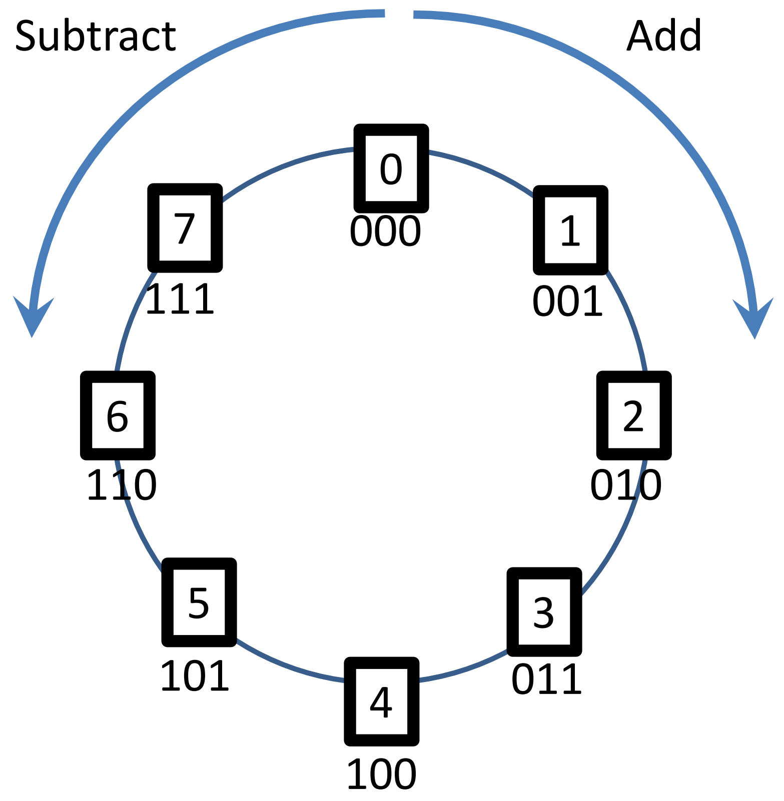
```
   1 1 1 1
   1111 (-1) +
   0101 (5)
   -------
1  0100 (4)
```

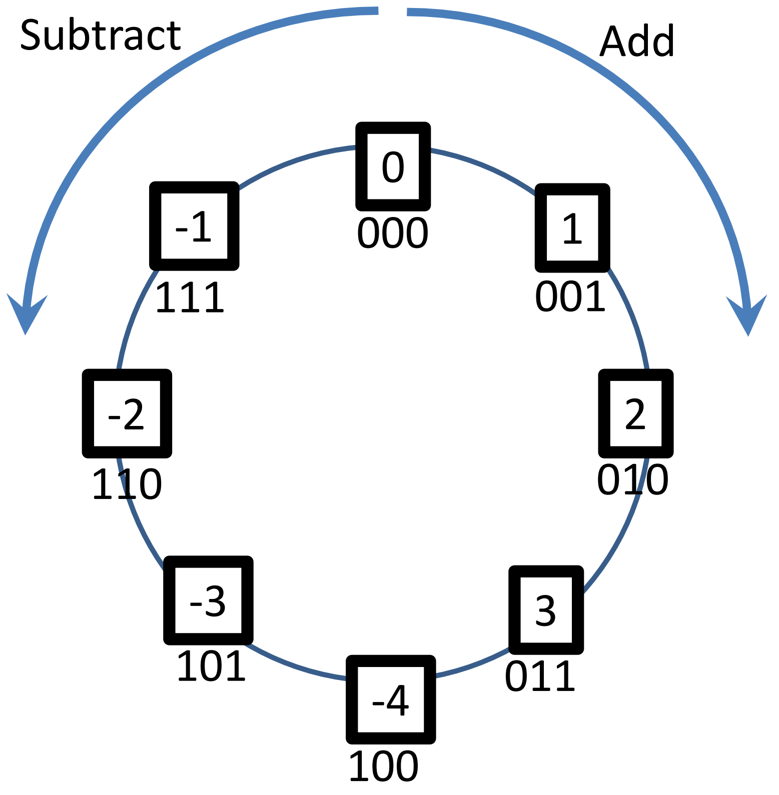
```
   1 1 1 1
   1111 (-1) +
   0001 (1)
   -------
1  0000 (0)
```

If 1111 represents -1, what does 1000 represent?

- 1111 = 1000 (x_{10}) + 0111 (7_{10}) (Just addition)
- -1 = x + 7_{10} ➔ -1 -7_{10} = x ➔ x = -8_{10}
- 1000 (-8_{10}) + 0100 (4_{10}) = 1100 (4_{10})
- 1000 (-8_{10}) + 0010 (2_{10}) = 1010 (-6_{10})
- 1000 (-8_{10}) + 0001 (1_{10}) = 1001 (-7_{10})
- 1000 (-8_{10}) + 0100 (4_{10}) + 0010 (2_{10}) + 0001 (1_{10}) = 1111 (-1_{10})
- **Upshot: For 4-bit 2s complement, 1000 is -8_{10}**
  - We'll show that this generalizes w.r.t. powers-of-two and the left-most bit position, ie. -2^{w-1}, where *w* is the number of bits used to represent a number.
  - E.g. For 16-bit numbers the most negative value is -2^{15}

2s Complement As A Ring/Modular Arithmetic





Signed Number Line: Bit Patterns and Values

- Zero is always represented with a bit pattern of all 0s
  - E.g. 0000 ($0_{10)}$
- The most negative number always has the bit pattern 1000…000
  - E.g. 1000 (-8_{10})
- The most positive number always has the bit pattern 0111…1111
  - E.g 0111 (7_{10})
- The most negative number always has a value of -2^{w-1}
  - `Where *w* is the width of the number in bits, e.g. 1101 has a width of 4
  - This is because the left-most digit represents Base^{w-1}
    - e.g. the third digit from the left in decimal represents 10^2
    - and the 3rd digit in binary represents 2^2
- The most positive number always has a value of $2^{w-1}+1$

Signed Number Line: Overarching Properties

- Non-negative binary numbers start at 0 and add from there
  - 0101 = 0 + 4 + 2
- Negative (2s complement numbers) start with -2^{w-1} and add from there
  - 1101 = -8 + 4 + 1 = -3
- The number line is off-balance, e.g. -8 to 7
  - The high-order bit is negative
  - The sum of the low-order bits is less than the high-order bit, e.g. 1000 = 0111 + 1
- There is only one zero, the bit pattern with all 0s
  - It is not represented in 2s complement
  - Thus we say that we use twos complement for "negative numbers"
    - Not "non-positive numbers"
  - Thus, 0 comes out of the otherwise-positive side of the number line (another way to remember the off-balanced-ness)

Summary: Signed Numbers and Arithmetic

- Negative numbers are represented via "2s complement"
  - Complement all bits and add 1
- Subtraction is accomplished by adding to a 2s complement (negative) number.
  - The carry-out and the added bit work together to make this work
  - This means that computers only need an adder, not a subtractor
- A number can be made negative by complementing it and adding 1
- A negative number can be made positive by subtracting one and complementing it
- The most negative number has no peer on the positive side of the number line
  - Subtracting one and complementing it gives itself, because it "wraps around"

Encoding Integers: Closed Form Expressions

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

- 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: Summary

- 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

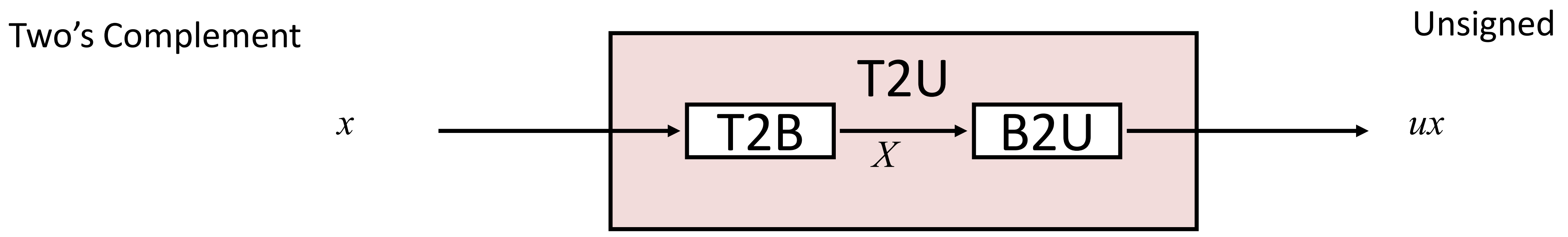
Example Data Representations in Byte

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

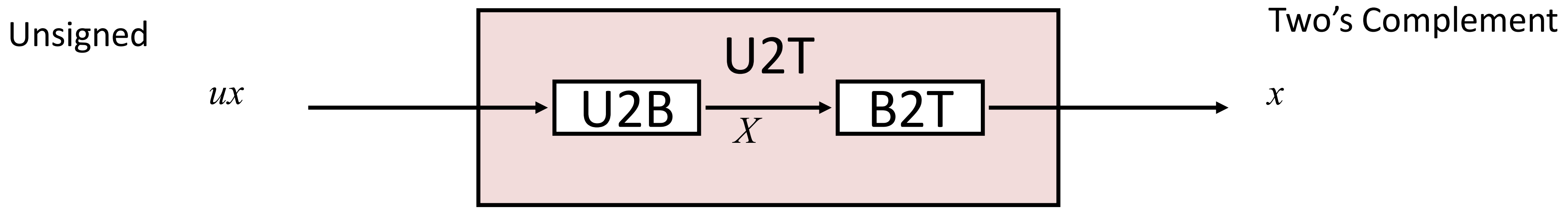
Today: Bits, Bytes, and Integers

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

Mapping Between Signed & Unsigned



Maintain Same Bit Pattern

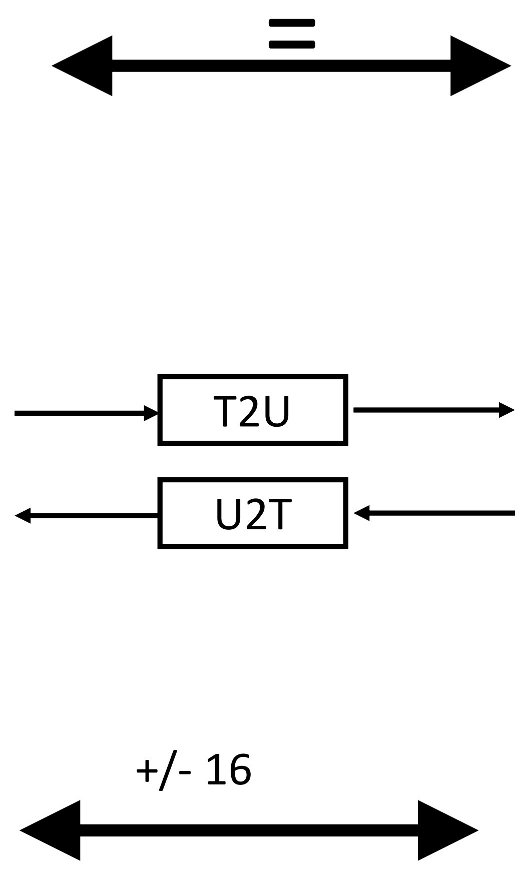


Maintain Same Bit Pattern

- 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



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;
```

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as "False"
    - Anything
    - Alway
    - Early
- Examples
  - !0x41
  - !0x00
  - !!0x41
  - 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

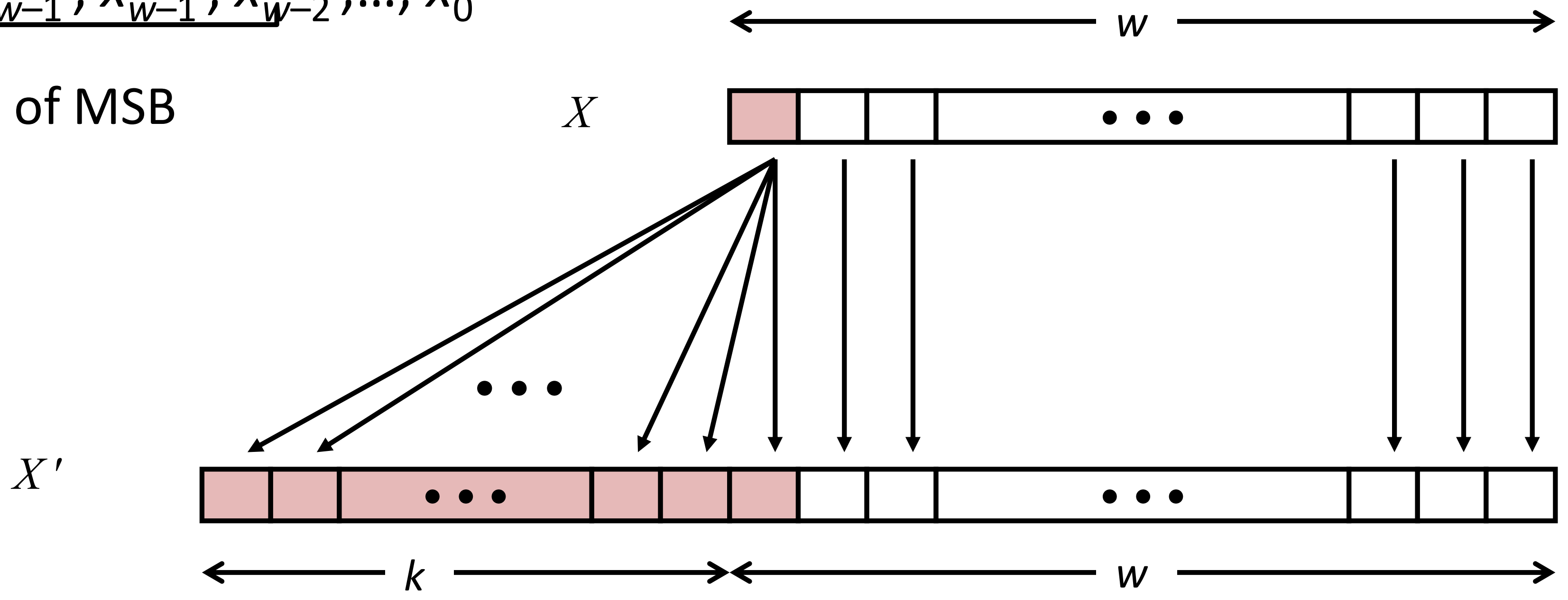
Casting Surprises

- 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

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



Today: Bits, Bytes, and Integers

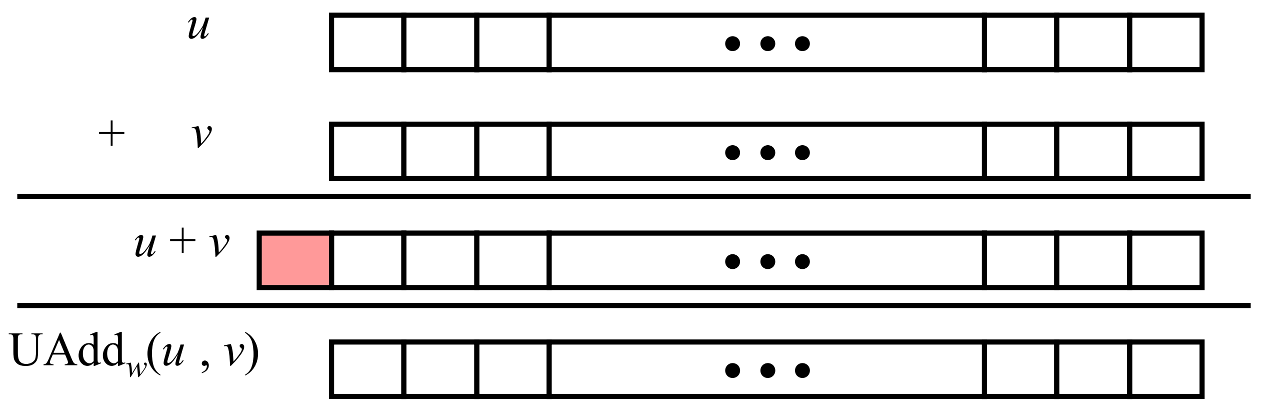
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion (casting), expanding
  - **Addition, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



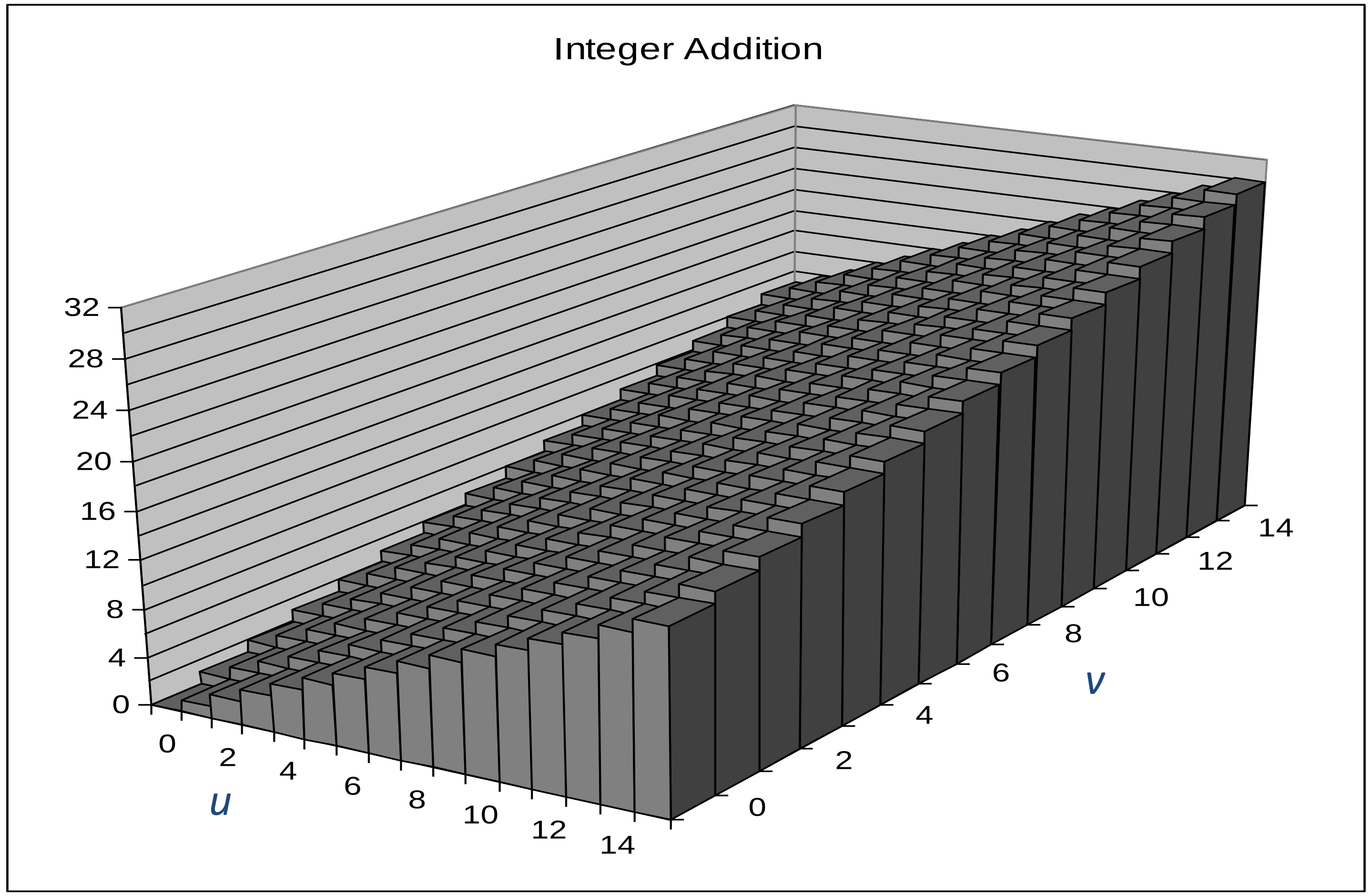
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

Visualizing (Mathematical) Integer Addition

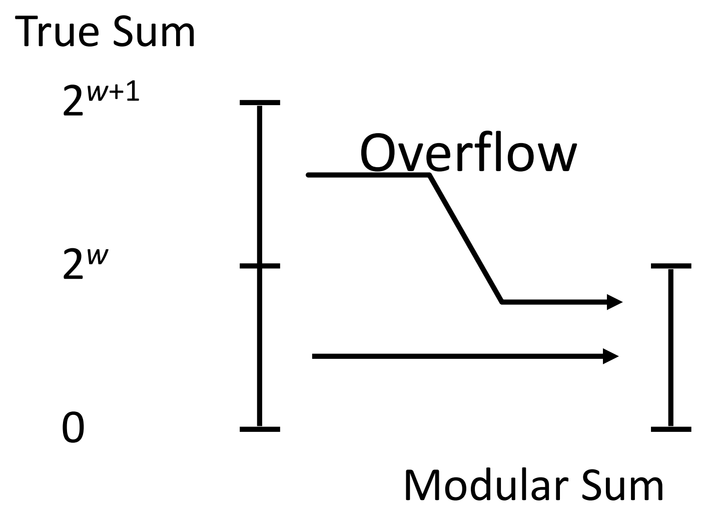
- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $\mathrm{Add}_4(u , v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

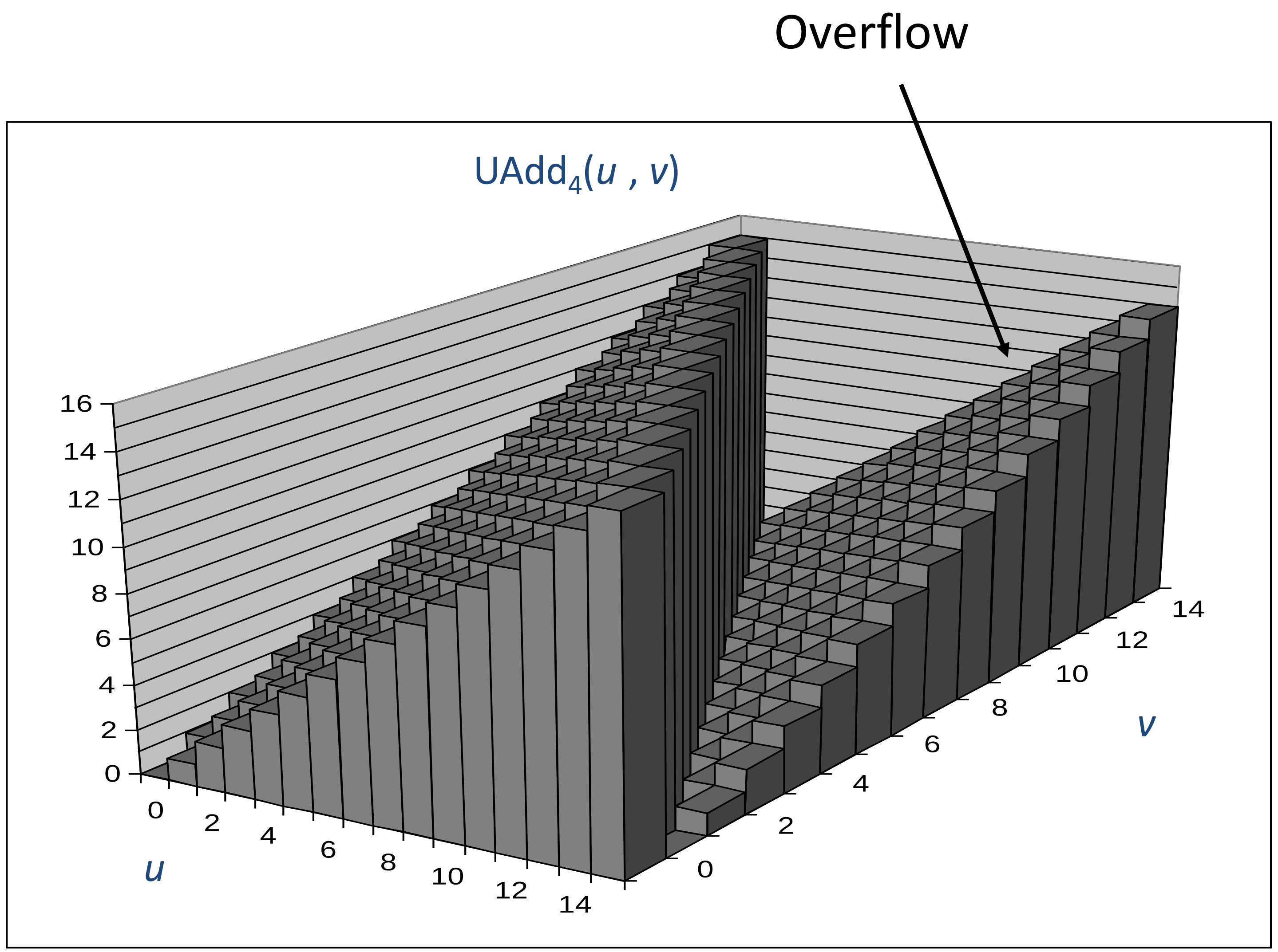
$\mathrm{Add}_4(u , v)$



Visualizing Unsigned Addition

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



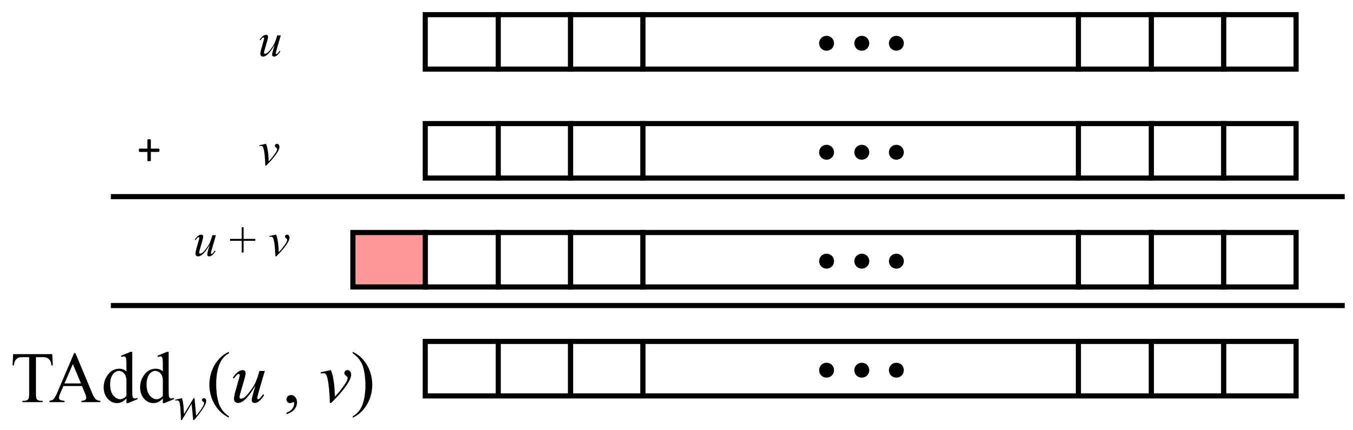


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



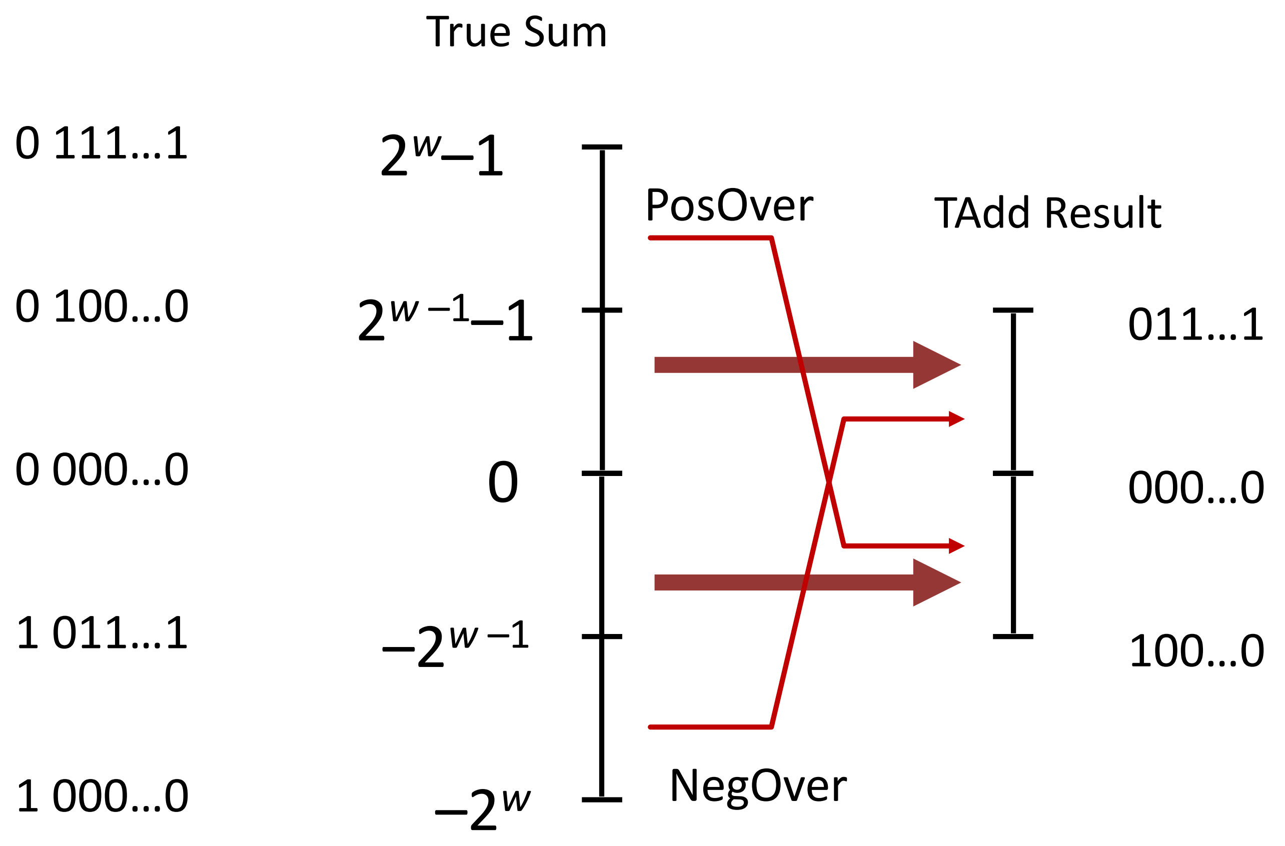
- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:

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

  - Will give `s == t`

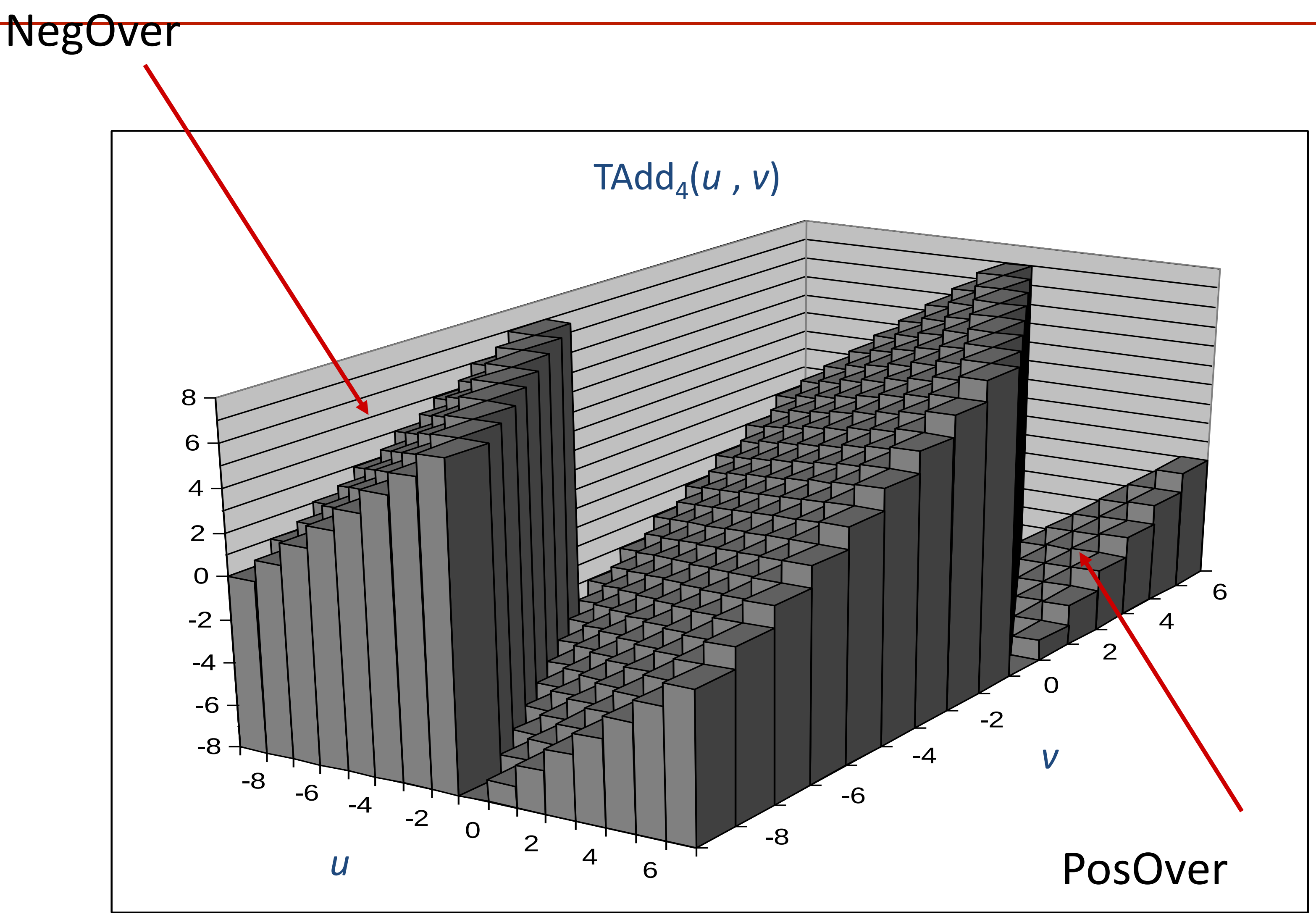
Twos Compliment Add Overflow (TAdd)

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



Visualizing 2's Complement Addition

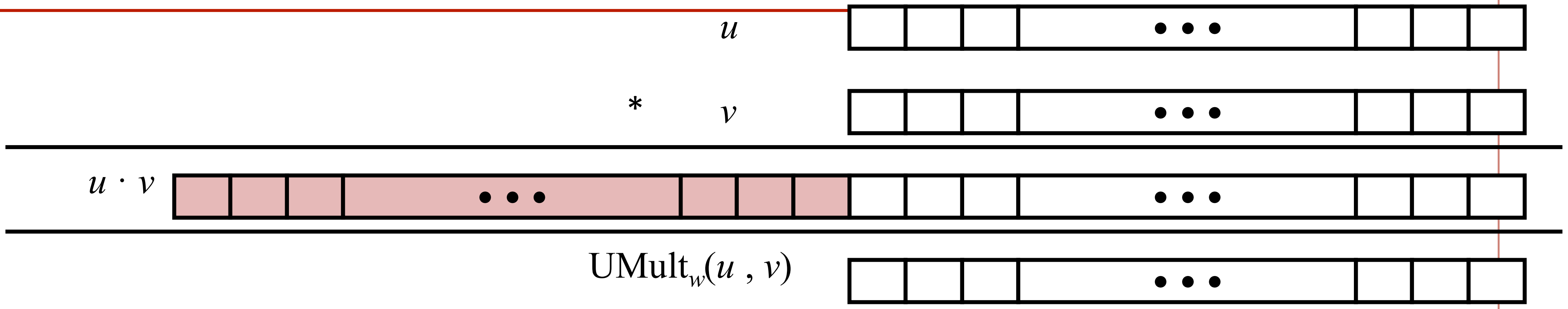
- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



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

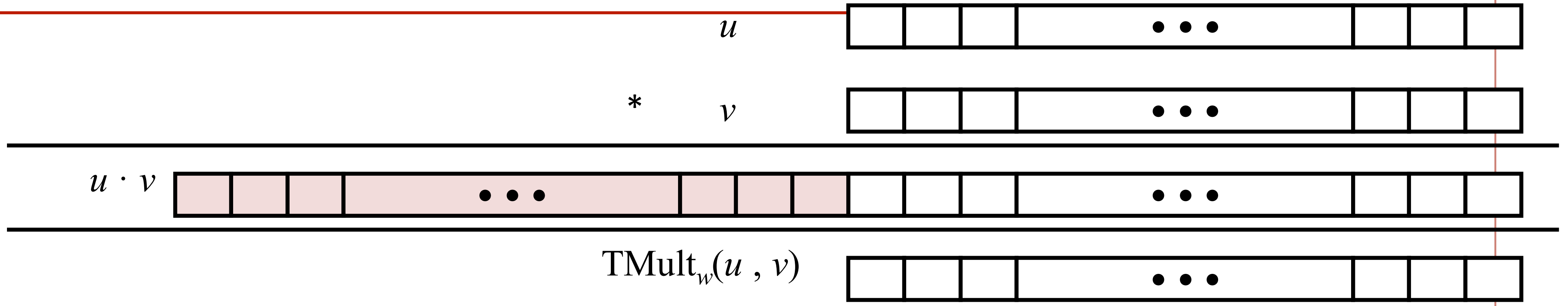
  $\text{UMult}_w(u, v) \quad = \quad u \cdot v \mod 2^w$

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

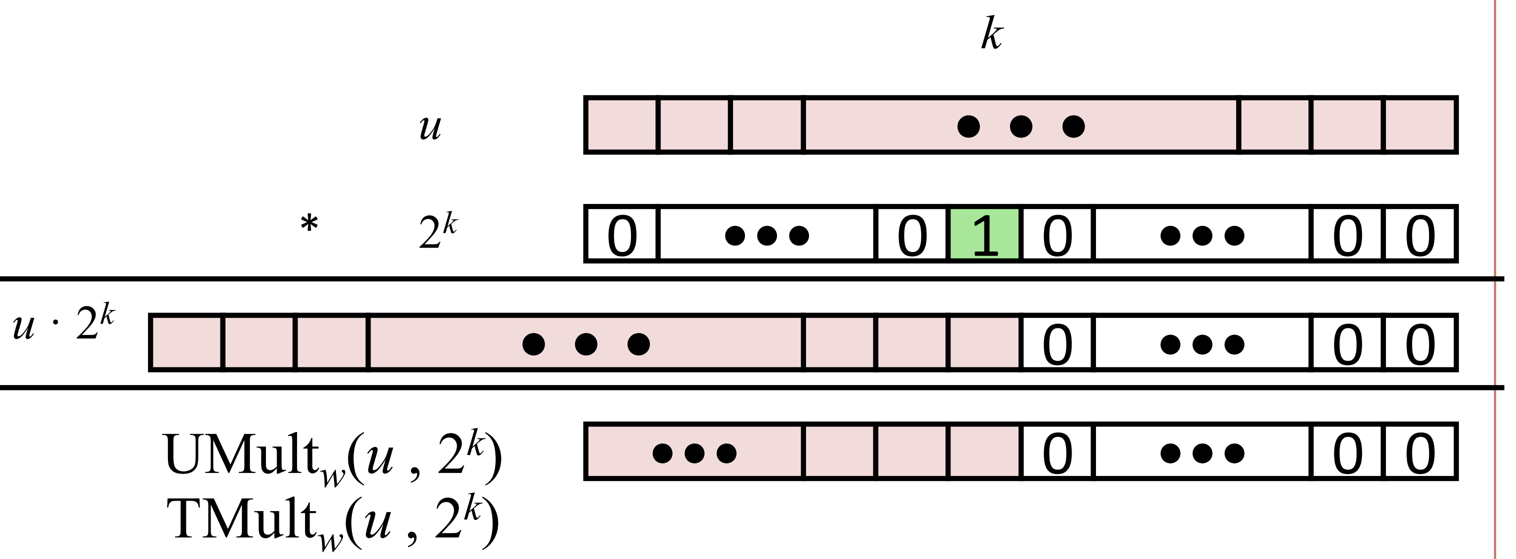
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w*+*k* bits

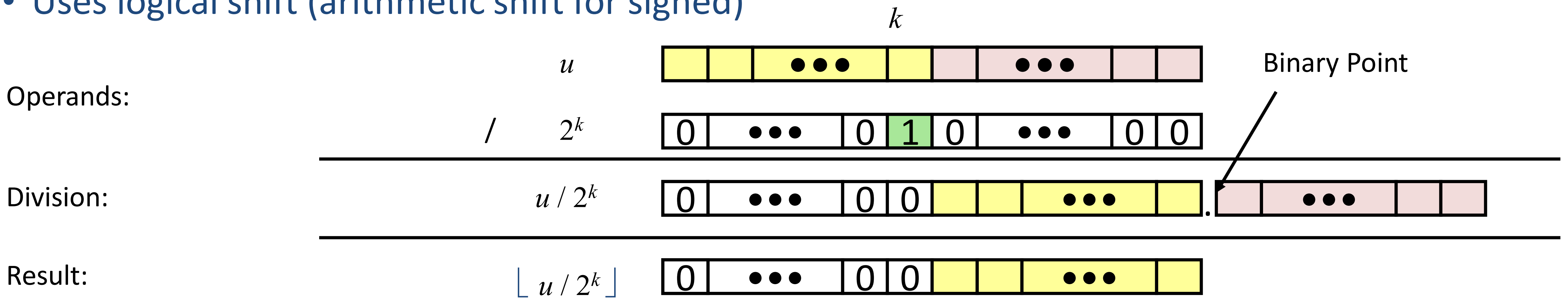
Discard *k* bits: *w* bits



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

Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - **`u >> k`** gives $\lfloor$ **`u / `** $2^k \rfloor$
  - Uses logical shift (arithmetic shift for signed)

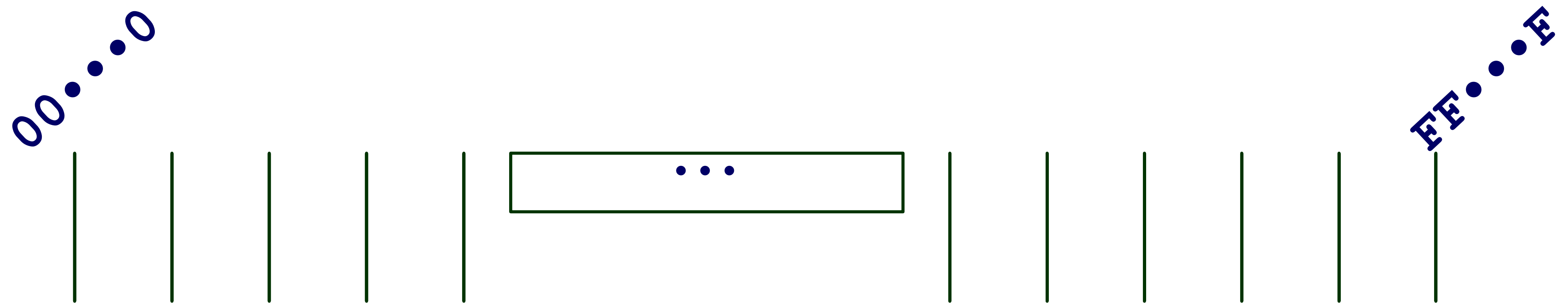


	Division	Computed	Hex	Binary
`x`	15213	15213	3B 6D	00111011 01101101
`x >> 1`	7606.5	7606	1D B6	00011101 10110110
`x >> 4`	950.8125	950	03 B6	00000011 10110110
`x >> 8`	59.4257813	59	00 3B	00000000 00111011

Today: Bits, Bytes, and Integers

- Representing information as bits
- 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

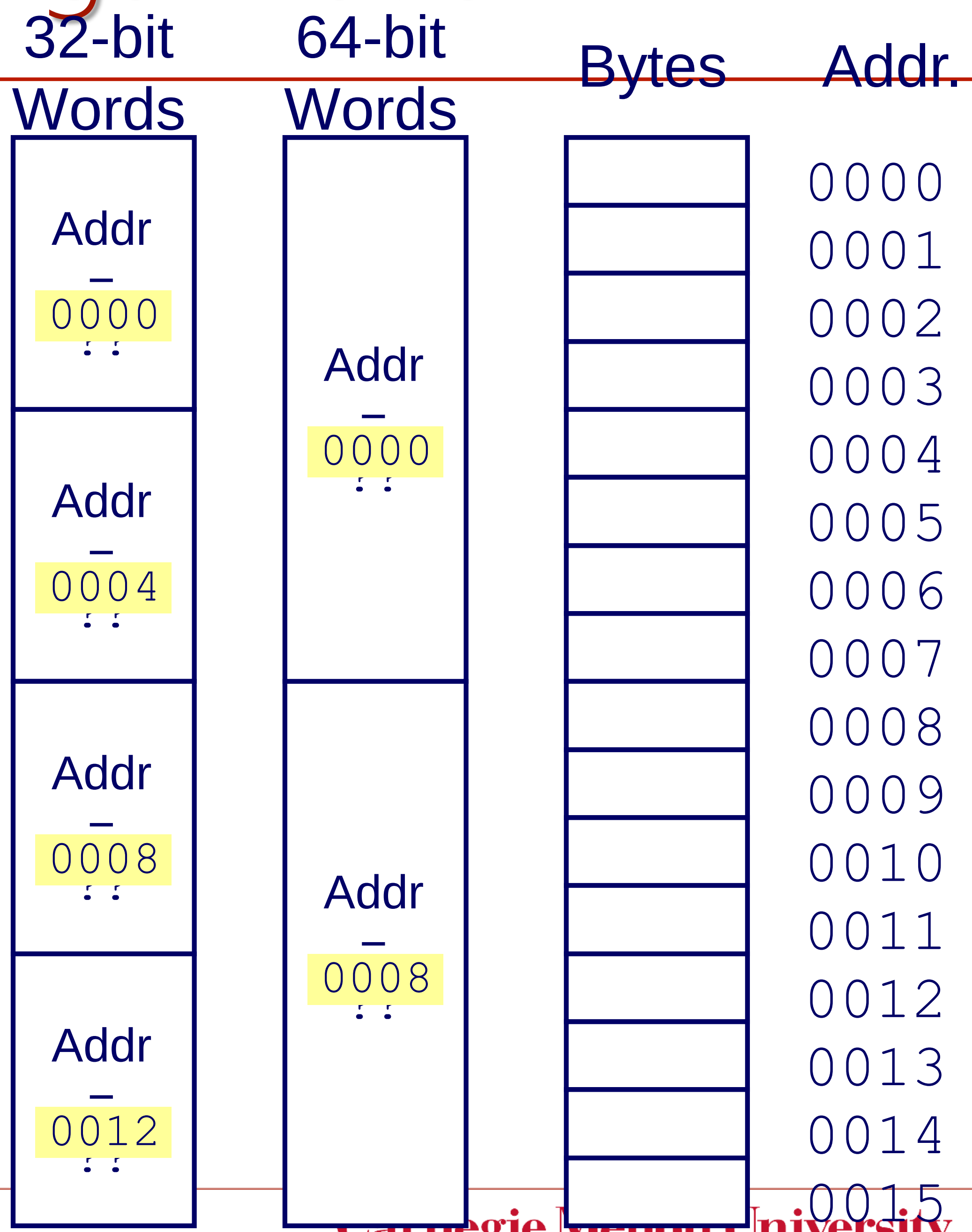
- Note: system provides private address spaces to each "process"
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

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)

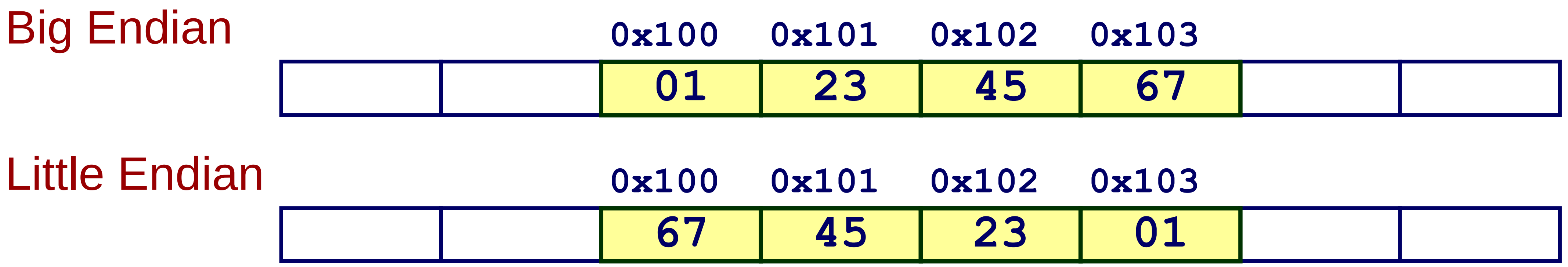


Byte Ordering

- 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

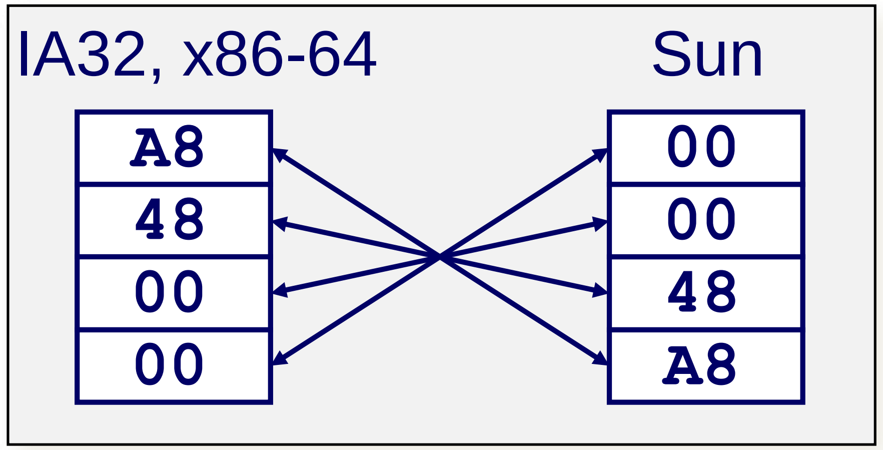
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



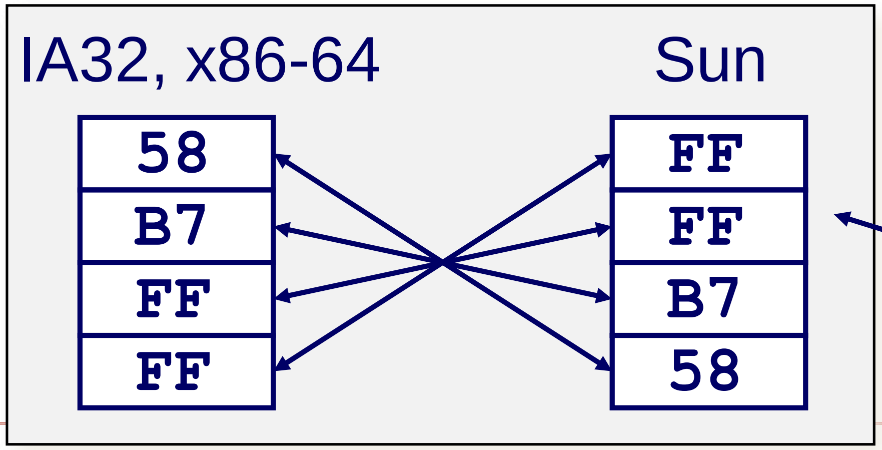
Representing Integers

Decimal:	**18600**
Binary:	**0100 1000 1010 1000**
Hex:	**4 8 A 8**

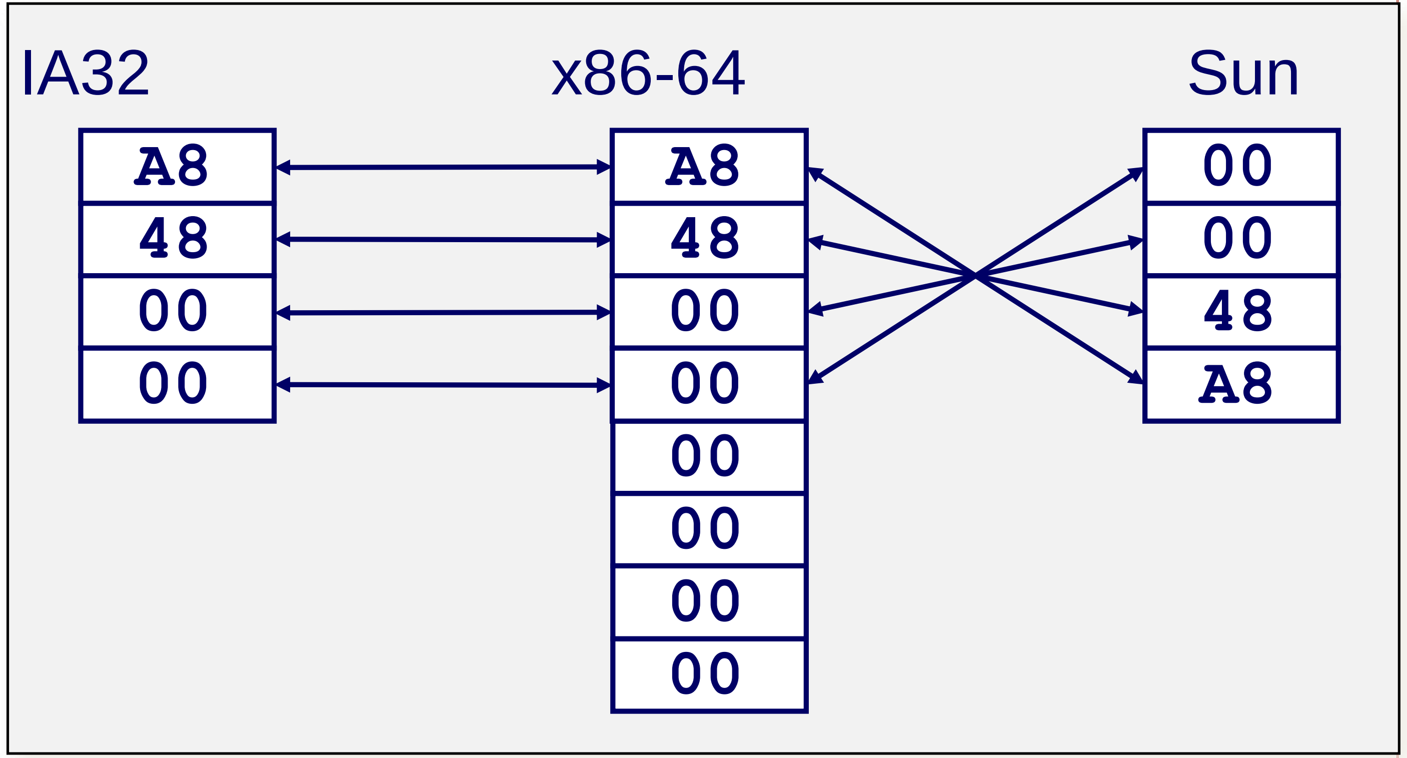
```
int A = 18600;
```



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



```
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

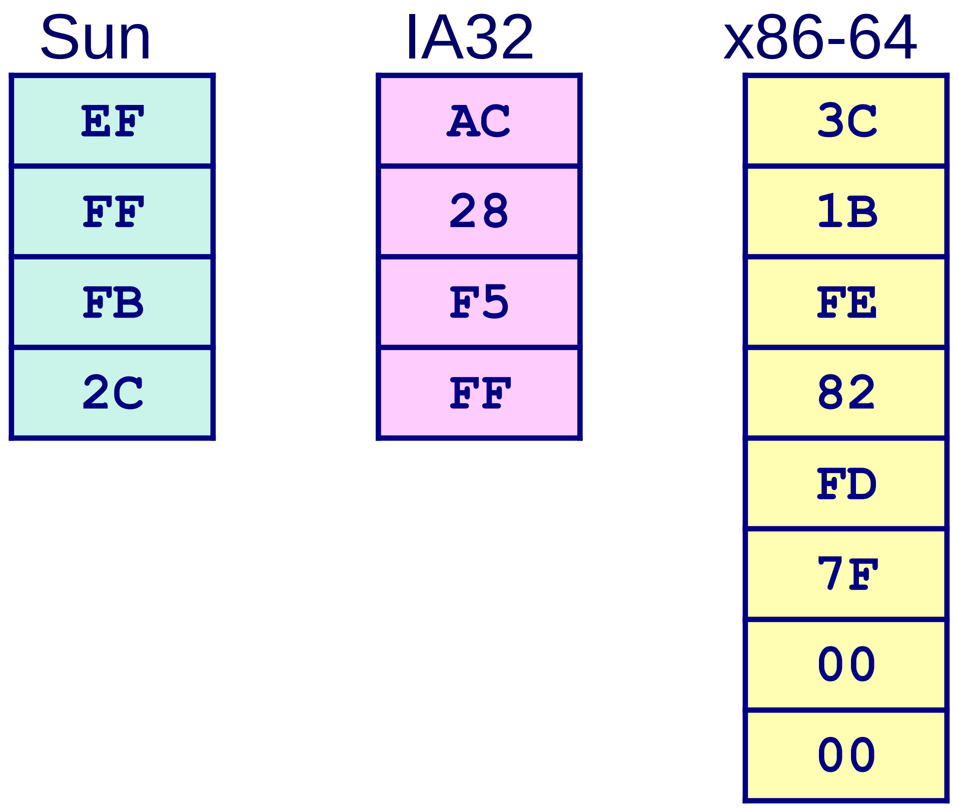
```
int a = 18600;
printf("int a = 18600;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

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

Representing Pointers

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



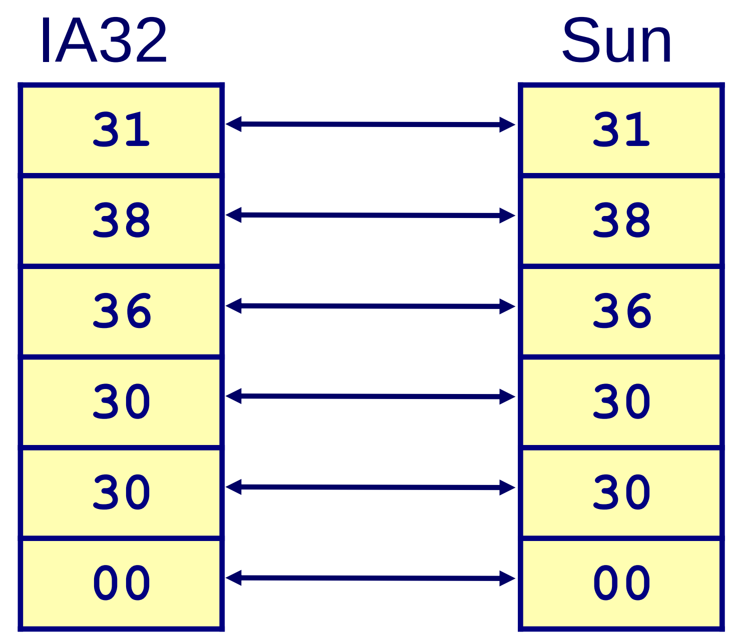
Different compilers & machines assign different locations to objects

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

- `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;
```

18-600 Foundations of Computer Systems

Lecture 4:
"Floating Point"

September 11, 2017

Next Time …

Electrical & Computer ENGINEERING

Carnegie Mellon University