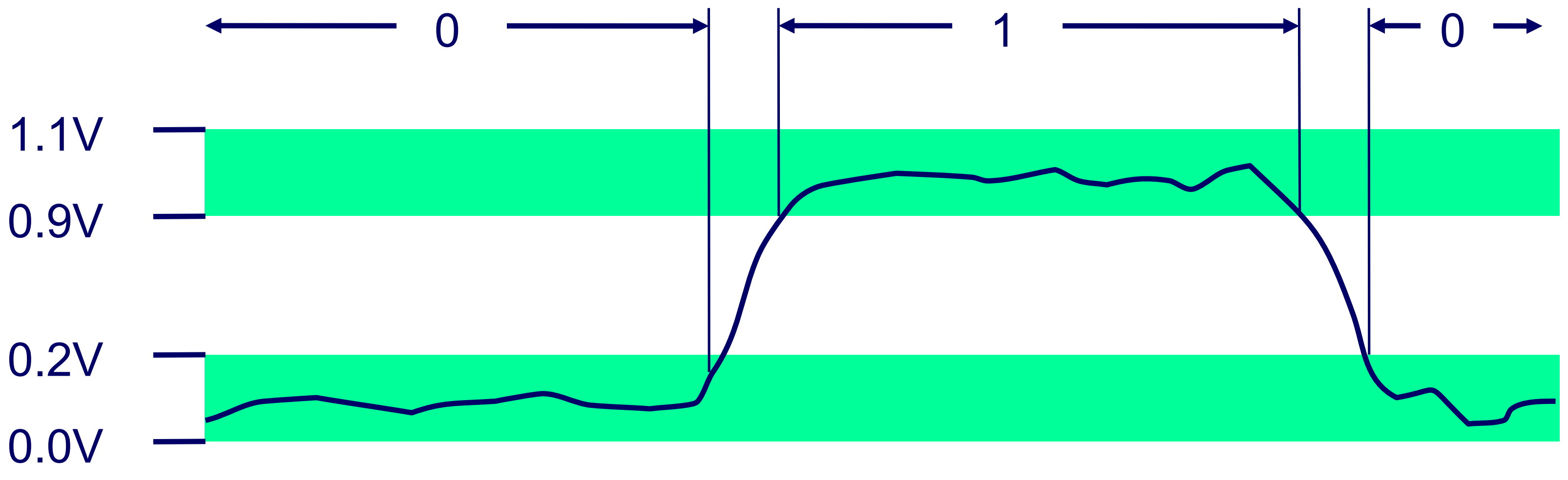
Introduction to Computer Systems
Lecture 3 – Bits, Bytes, and Integers

2022 Spring, CSE3030

Sogang University

Binary Representations

- Bit – a unit that represents two status, 0 and 1.
- Why not 10-base representation?
  - Easy to store with bistable elements
  - Compact implementation of arithmetic functions with logic gates.
  - Reliably transmitted on noisy and inaccurate wires
- Electronic implementation
  - Low voltage – 0, High voltage – 1



Representing Information

- Information = Bits + Context + Representation
  - Information is written in bits on the memory.
  - The context indicates the data type of a set of bits.
  - Representation will give meaning to bits.
- How many information can N bits represent?
  - 2^n things
- How to represent different types of information?
  - Each information type has its data representation.
  - Characters, numbers (integer and float), pixels, machine instructions

Binary	0101 0011	0100 1111	0100 0111	0100 0001	0100 1110	0100 0111	0100 0011	0101 0011
Character	'S'	'O'	'G'	'A'	'N'	'G'	'C'	'S'
Integer	1095192403				1396918094			
Double	1.256674 x 10^93							

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 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

Example Data Representations

	C Data Type	Typical 32-bit	Typical 64-bit	x86-64
Character	**`char`**	1	1	1
Integer	**`short`**	2	2	2
	`int`	4	4	4
	`long`	4	8	8
Real number	**`float`**	4	4	4
	`double`	8	8	8
	`long double`	–	–	10/16
	pointer	4	8	8

Representations for Integers

- Unsigned integer representation
  - Unsigned int/short/long

$$\mathbf{B} = [b_{w-1}, b_{w-2}, \ldots, b_0] = \sum_{i=0}^{w-1} b_i 2^i = b_0 1 + b_1 2 + \ldots + b_{w-1} 2^{w-1}$$

$$10011011 = 2^7 + 2^4 + 2^3 + 2^1 + 2^0 = 155$$

- Signed integer representation
  - Using **two's complement encoding** to represent negative numbers
  - The left-most bit is a sign bit.

$$\mathbf{B} = [b_{w-1}, b_{w-2}, \ldots, b_0] = -b_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i = b_0 1 + b_1 2 + \ldots + b_{w-2} 2^{w-2} - b_{w-1} 2^{w-1}$$

$$10011011 = -2^7 + 2^4 + 2^3 + 2^1 + 2^0 = -101$$

- W-bit integer representation can represent 2^W numbers.
  - Represent a finite set of integer numbers.

Two's Complement Encoding

- Its complement with respect to 2^w.
  - The sum of a number and its two's complement is 2^w.

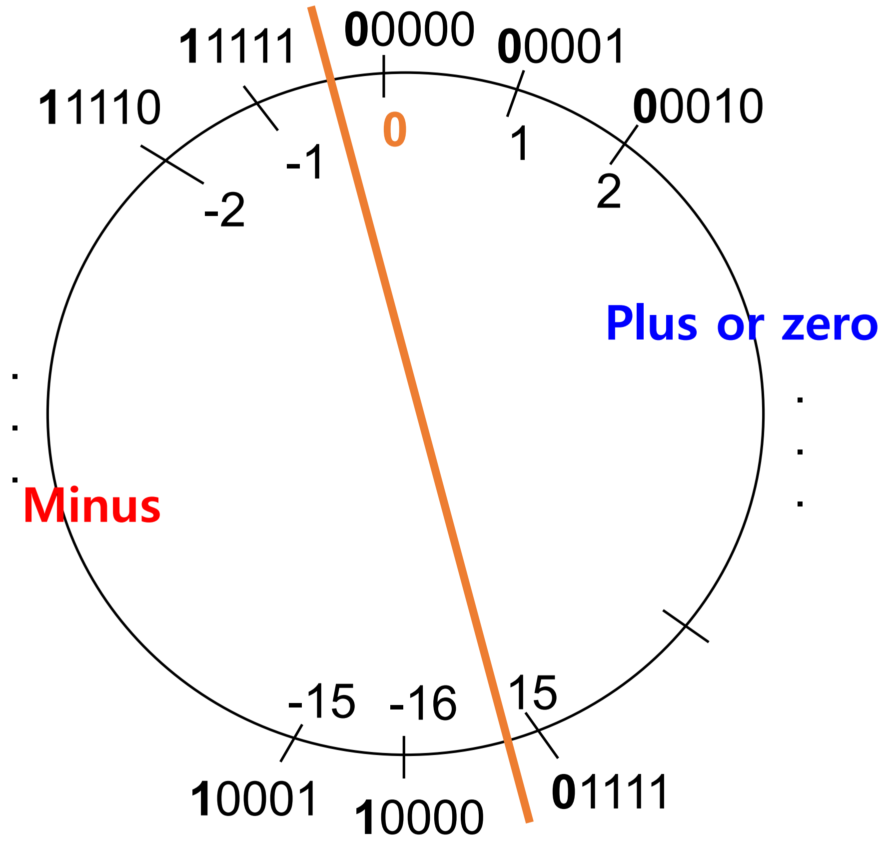
$$
\begin{aligned}
100000000_{(2)} &= \mathrm{complement}\left(01110011_{(2)}\right) + 01110011_{(2)} \\
&= 10001101_{(2)} + 01110011_{(2)}
\end{aligned}
$$

  - By switching all bits of *x* and adding one, you can get the complement of x.

$$
\mathrm{complement}\left(01110011_{(2)}\right) = 10001100_{(2)} + 1 = 10001101_{(2)}
$$

Signed Integer Representation

- Two's Complement Representation
  - Unique zero
  - A signed bit represents "minus".
    - 0 – greater than or equal to 0
    - 1 – less than 0



$$\mathbf{B} = [b_{w-1}, b_{w-2}, \ldots, b_0] = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

Principle of using Complements as Negative Numbers

- By adding and subtracting a base to the power *n* at different positions, subtraction can be considered as addition of a complement.
- A>B
  - $999_{(10)} - 100_{(10)} = \mathbf{0}999_{(10)} + (1\mathbf{0}000 - \mathbf{0}100)_{(10)} - 1\mathbf{0}000_{(10)}$
    $= \mathbf{0}999_{(10)} + \mathbf{9}900_{(10)} - 1\mathbf{0}000_{(10)}$ (get the complement of 0100, i.e. 9900)
    $= 1\mathbf{0}899_{(10)} - 1\mathbf{0}000_{(10)} = \cancel{1}\mathbf{0}899_{(10)}$ (delete a digit that is out of the range)
    $= \mathbf{0}899_{(10)}$
- A<B
  - $0099_{(10)} - 0100_{(10)} = \mathbf{0}099_{(10)} + (1\mathbf{0}000 - \mathbf{0}100)_{(10)} - 1\mathbf{0}000_{(10)}$
    $= \mathbf{0}099_{(10)} + \mathbf{9}900_{(10)} - 1\mathbf{0}000_{(10)}$ (get the complement of 0100, i.e. 9900)
    $= \mathbf{9}999_{(10)} - 1\mathbf{0}000_{(10)} \Rightarrow \mathbf{9}999_{(10)} = -1_{(10)}$ (abbreviate -10000)
- Bold numbers represents signs (0 – positive or zero, 9 – minus)

Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- Other Values
  - Minus 1
    111…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

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
  - $UMax = 2 * TMax + 1$

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

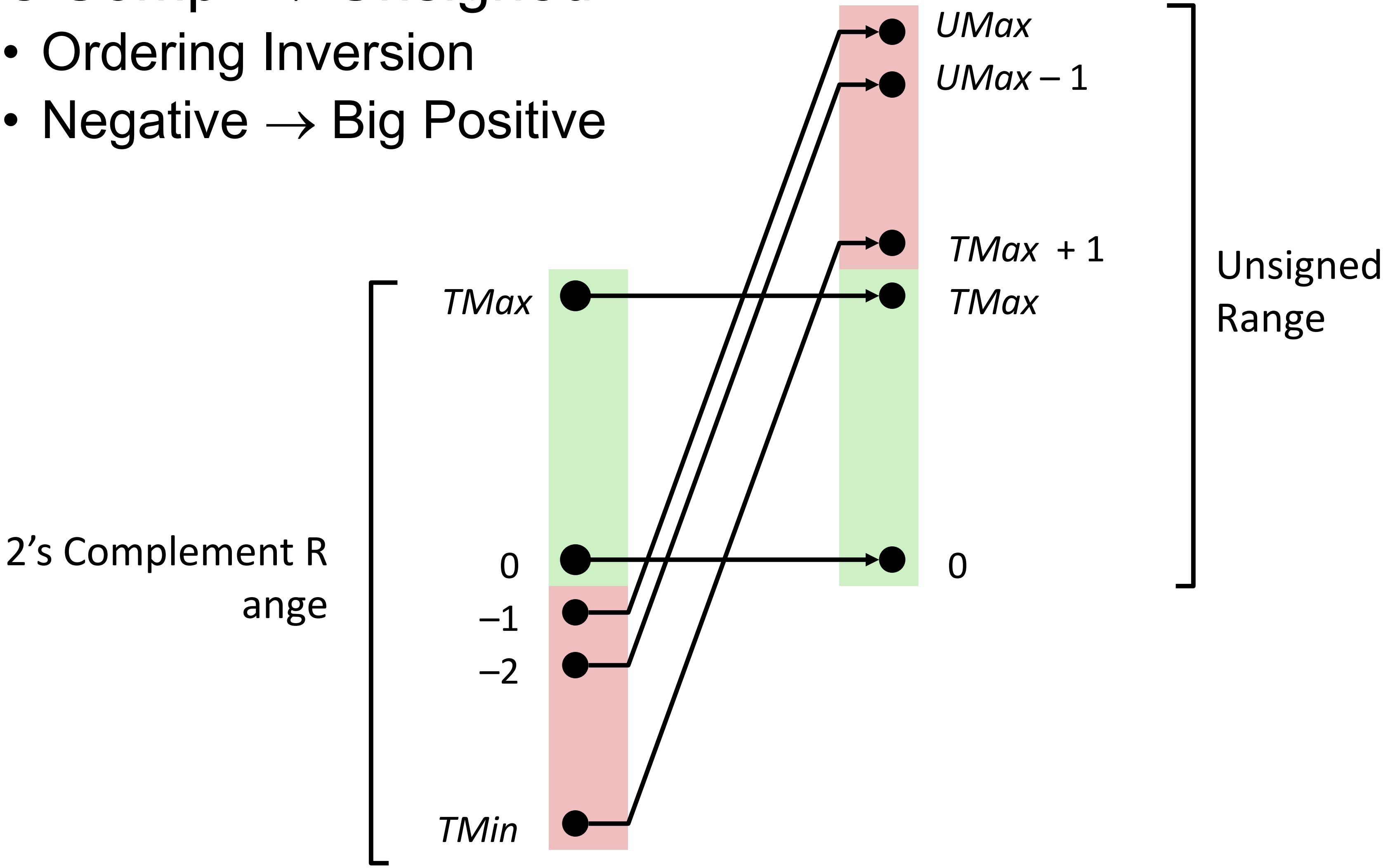
Unsigned & Signed Numeric Values

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
- $\Rightarrow$ Can Invert Mappings
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer

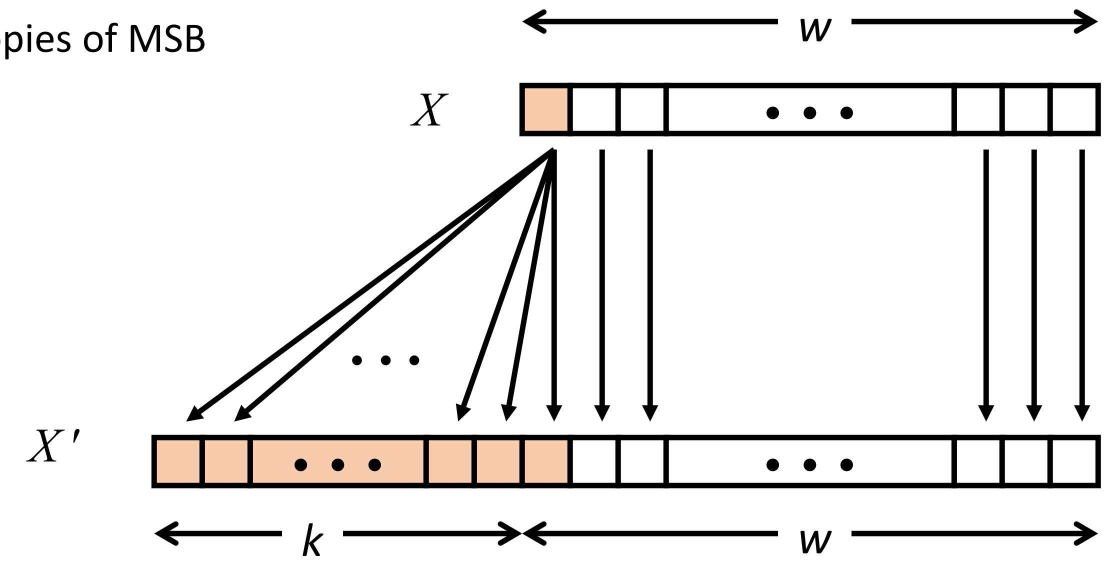
Conversion Visualized

- 2's Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



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 Example

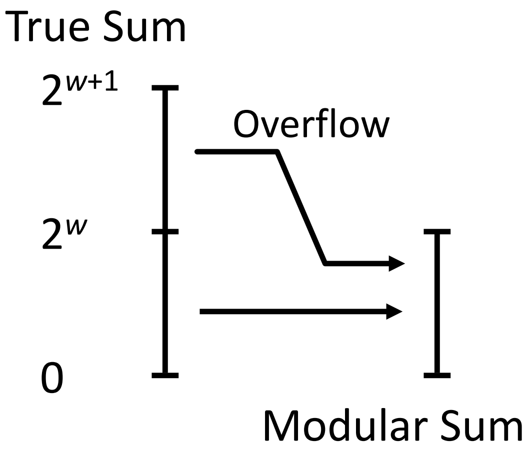
```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

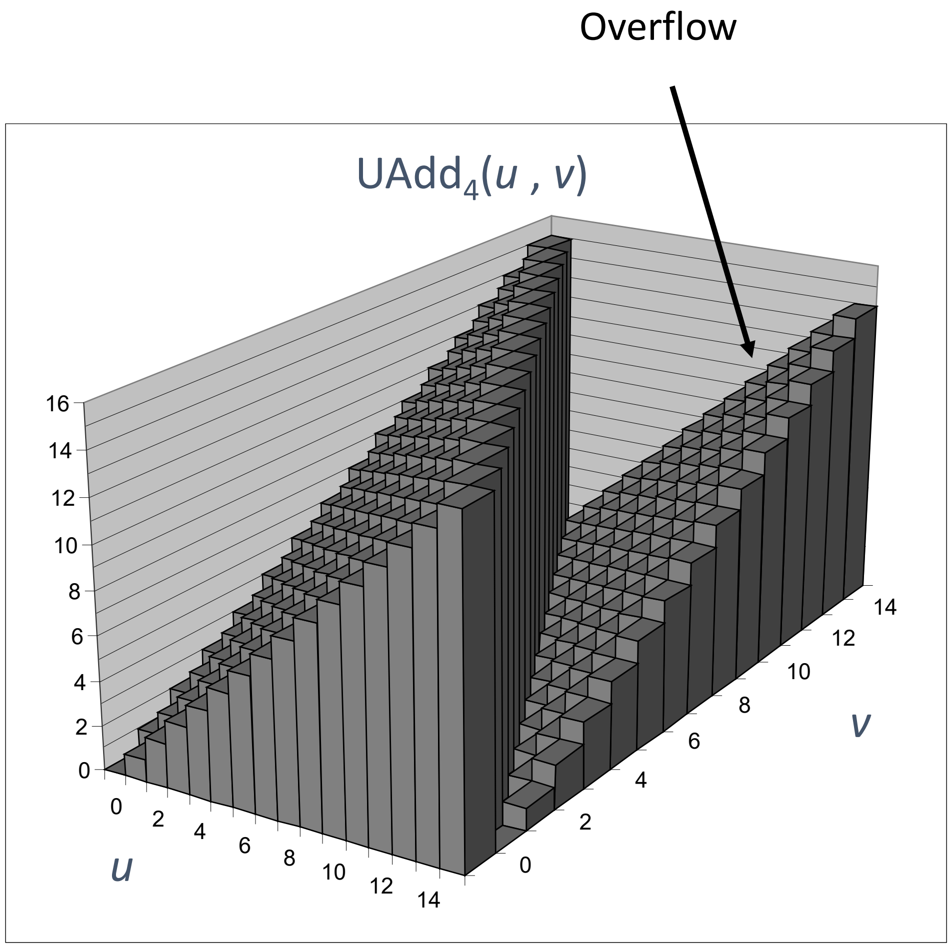
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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

Visualizing Unsigned Addition

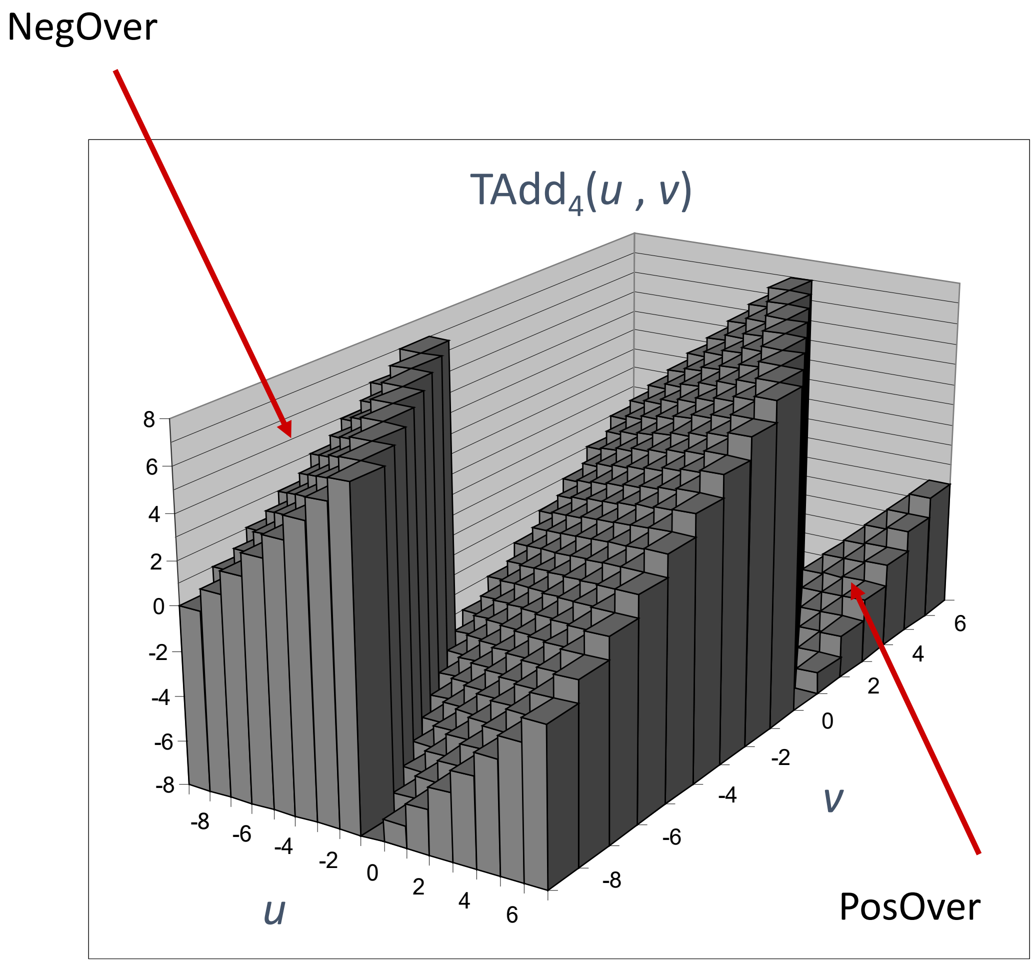
- Wraps Around
  - If true sum ≥ 2^w
  - At most once





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 negativ
      e
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once

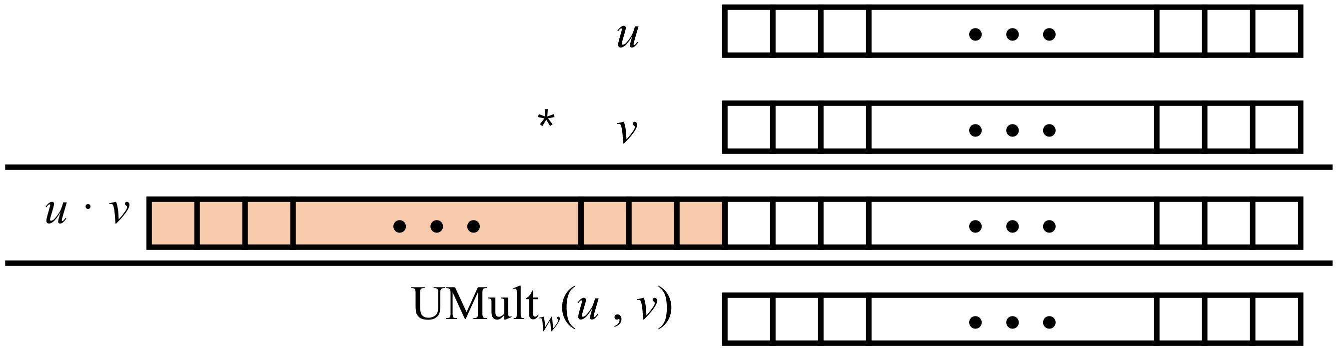


Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

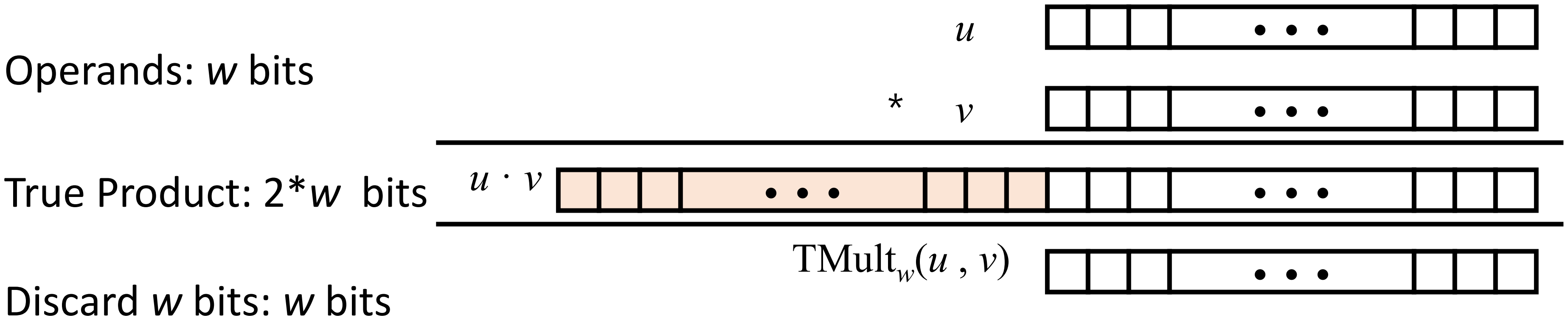
Discard *w* bits: *w* bits



- 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



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

Boolean Algebra

- How computers manipulate bits?
- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

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

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 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 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 (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

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

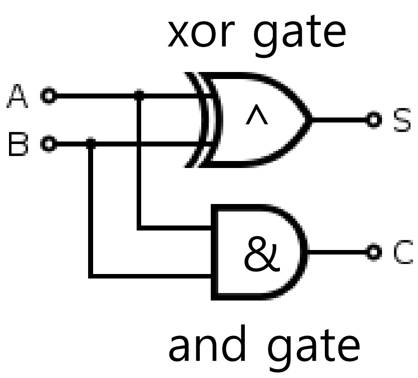
Adder

- Using bit operations, n-bit addition can be computed.

Carries	0	0	1	0	1	0	1	1	
A		0	0	1	0	1	0	1	1
B	+	0	0	0	0	1	0	0	1
		(0) 0	(0) 1	(0) 1	(0) 1	(1) 0	(0) 1	(1) 0	(1) 0

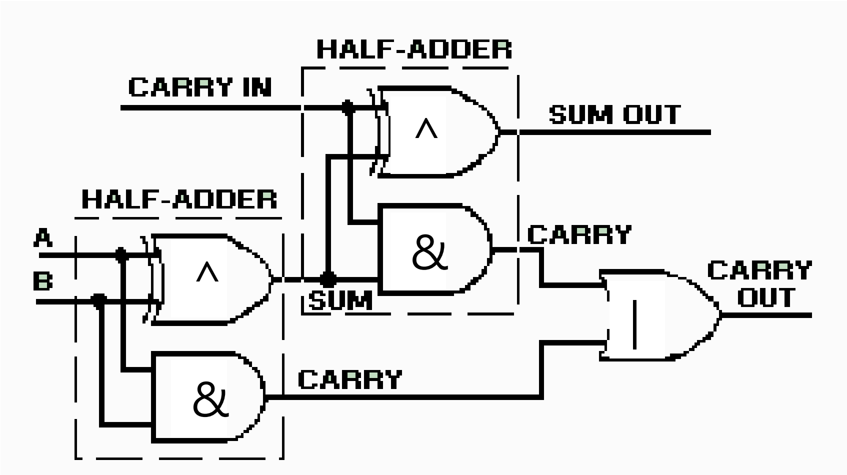
Implementation of Adder

- By consecutively concatenating full adders n-times, we can get a n-bit adder.



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

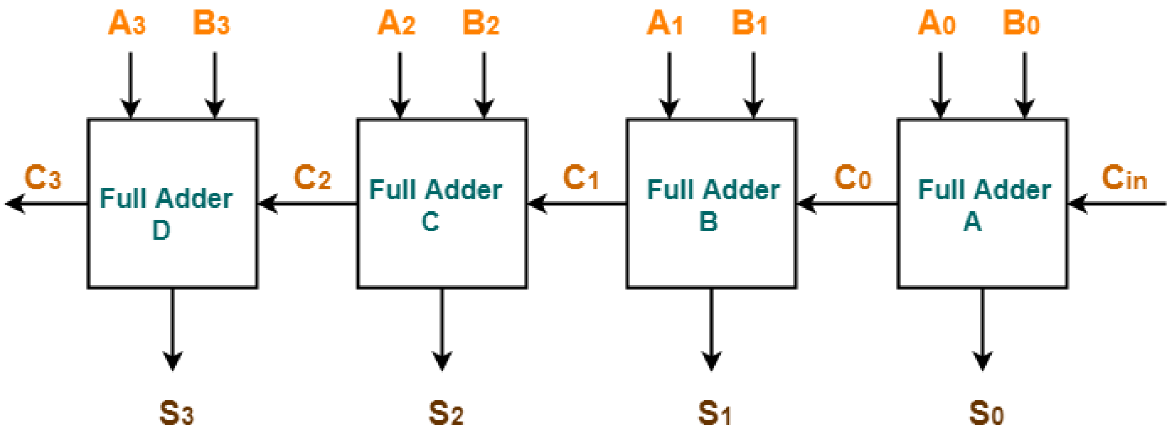
Half adder



A	B	C_{in}	S	C
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Full adder

4-bit integer adder



Carries	0	0	1	0	0
A		0	0	1	0
B	+	0	0	0	0
		(0) 0	(0) 1	(0) 1	(0) 0

https://www.gatevidyalay.com/delay-in-ripple-carry-adder/

Summary

- A computer encodes, stores, and manipulates information in bits.
- Representing negative numbers as 2's complements
- Use the same logic hardware for unsigned and signed integers.
  - If the true result is out of scope, the result is not valid.