Information Storage

Prof. Hyuk-Yoon Kwon
http://bigdata.seoultech.ac.kr

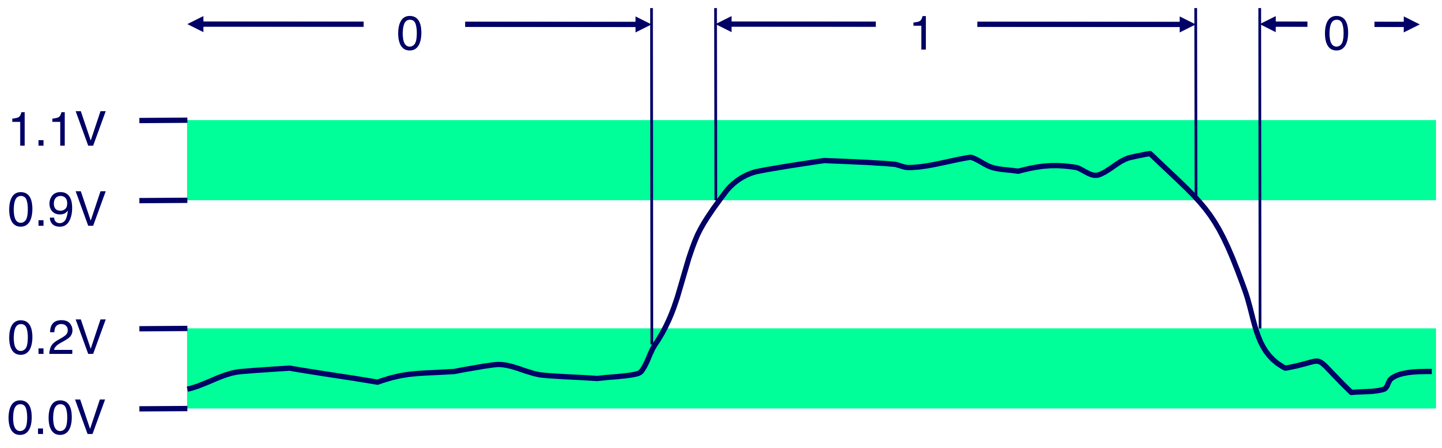
Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

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**

Everything is Bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



For example, can count in binary

Base 2 Number Representation

- Represent 15213_{10} as 11101101101101_2
- Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
- Represent $1.5213 \text{ X } 10^4$ as $1.1101101101101_2 \text{ X } 2^{13}$

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
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
`pointer`	4	8

(# of bytes)

char c = ‘A’;

short s = 10;

int i = 20;

long l = 40;

float f = 3.14;

double d = 42.195;

char *p = 0x12345678;

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

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

전부 T → T

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

&	0	1
0	0	0
1	0	1

Not

반전 F → T

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

~	
0	1
1	0

Or

하나라도 T → T

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

|	0	1
0	0	1
1	1	1

Exclusive-Or (Xor)

하나만 T → T

- **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**

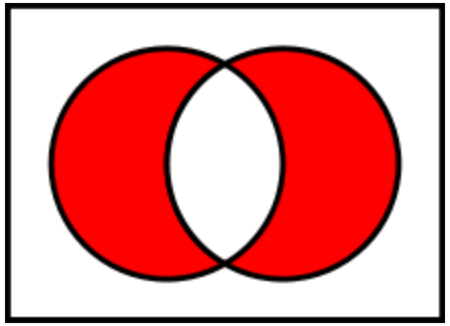
Example: Representing & Manipulating Sets

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
  - $\sim 01000001_2 \rightarrow 10111110_2$
- ~0x00 -> 0xFF
  - $\sim 00000000_2 \rightarrow 11111111_2$

Q1. 0x69 & 0x57 0110 1001 & 01010111 = 01000001 = 0x41

Q2. 0x69 | 0x57 01101001 | 01010111 = 0111 1111 = 0x7F

Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

Examples (char data type)

- !0x41 -> 0x00
- !0x00 -> 0x01

Q1. !!0x41 ?

Q2. 0x69 && 0x55

Q3. 0x69 || 0x55

- p && *p (avoids null pointer access)

check actual value existence stored in pointer.

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**
  - $\boxed{\text{Shift amount}}$ < 0 or ≥ word size

int → Left/Right Shift more than or equal to 32 → undefined.

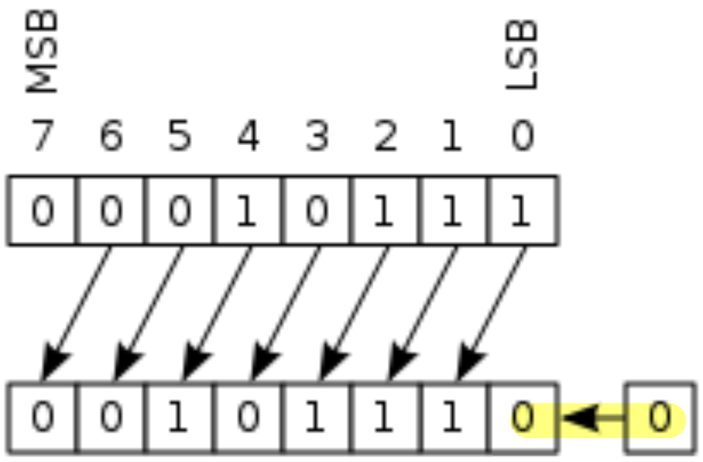
```
int i = 1;
i = i << 32
      << 33
       :
```
} undefined

Argument **x**	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

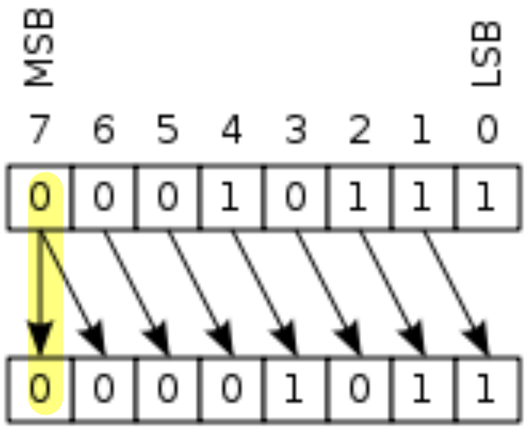
Argument **x**	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Arithmetic Shift

- **Arithmetic left shift** bigger num



- **Arithmetic right shift** smaller num



Arithmetic Shift in Programming (from Microsoft)

C++

```
#include <iostream>
#include <bitset>
using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1{short2};
    cout << bitset1 << endl;  // 0100000000000000

    short short3 = short1 << 1;
    bitset<16> bitset3{short3};  // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl;  // 100000000000000

    short short4 = short1 << 14;
    bitset<16> bitset4{short4};  // 4 left-shifted by 14 = 0
    cout << bitset4 << endl;  // 0000000000000000
}
```

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
- Summary

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

```
short int x =  15213;
short int y = -15213;
```

C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 1001001$\boxed{1}$

complement all bits & add 1

Sign Bit

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

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

Values for Different Word Sizes

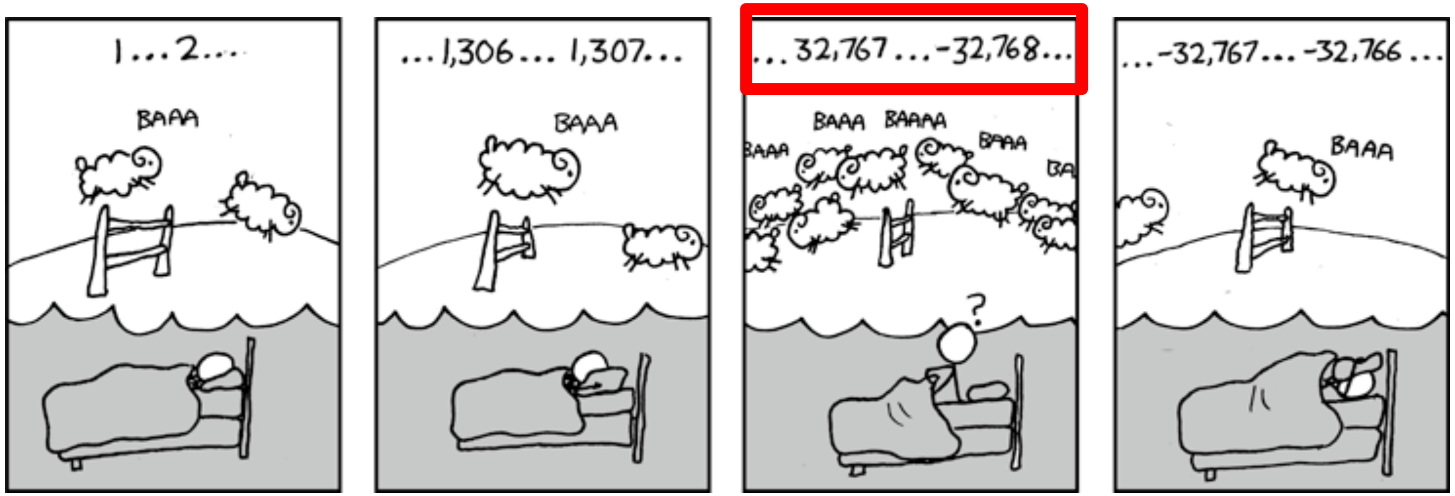
	W			
	char	short	int	long
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **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

Great Reality #1: Ints are not Integers?

- **Example : Is $x^2 \geq 0$?**



Source: xkcd.com/571

```
int a = 40000 * 40000; // 1600000000
int b = 50000 * 50000; // 2500000000 ??
```

Unsigned & Signed Numeric Values

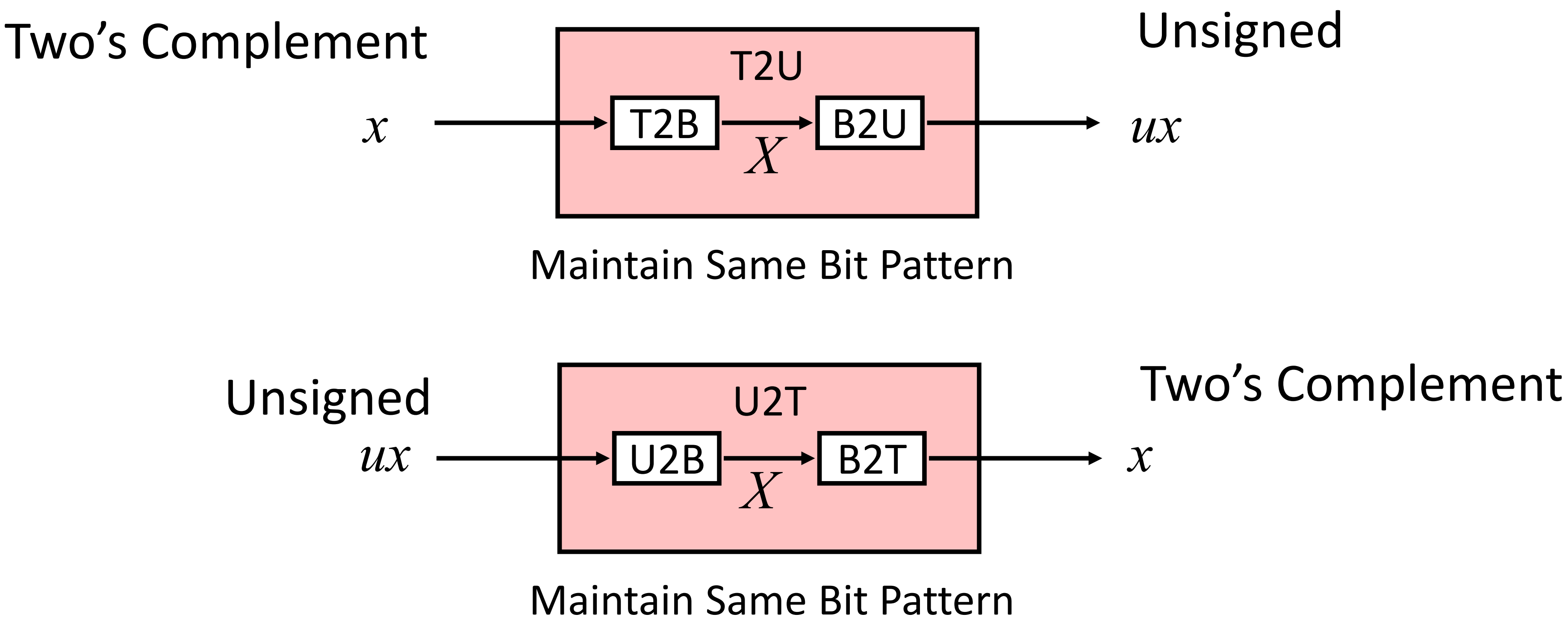
X	B2U(X)	B2T(X)
	unsigned	twos complements
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- $\Rightarrow$ **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

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

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

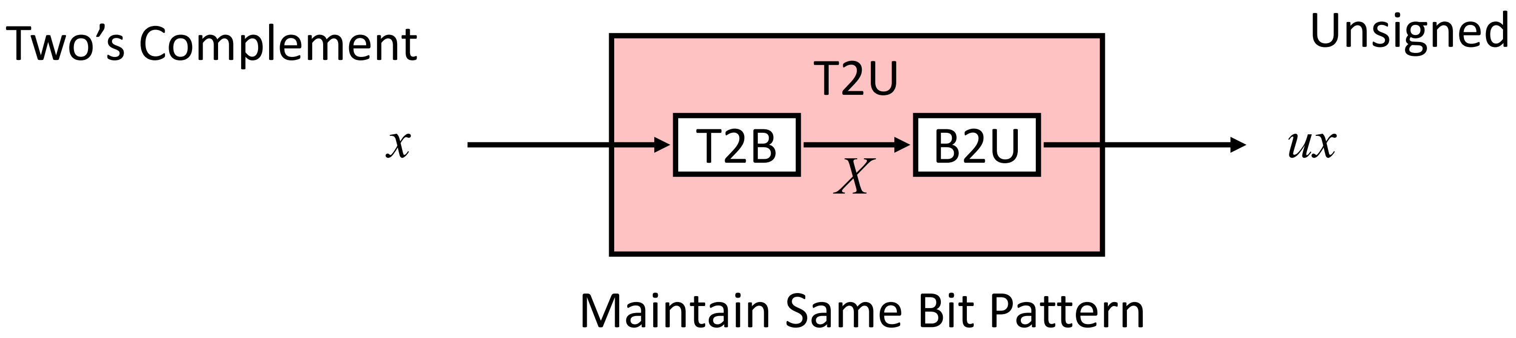
$\longrightarrow$ T2U $\longrightarrow$

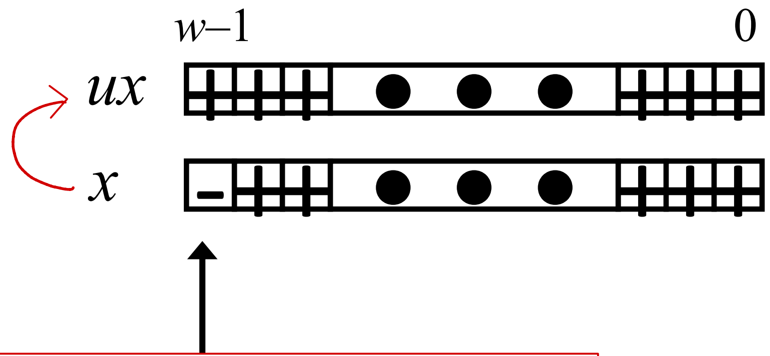
$\longleftarrow$ U2T $\longleftarrow$

Mapping Signed ↔ Unsigned

4 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 $= 2^4$	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Relation between Signed & Unsigned

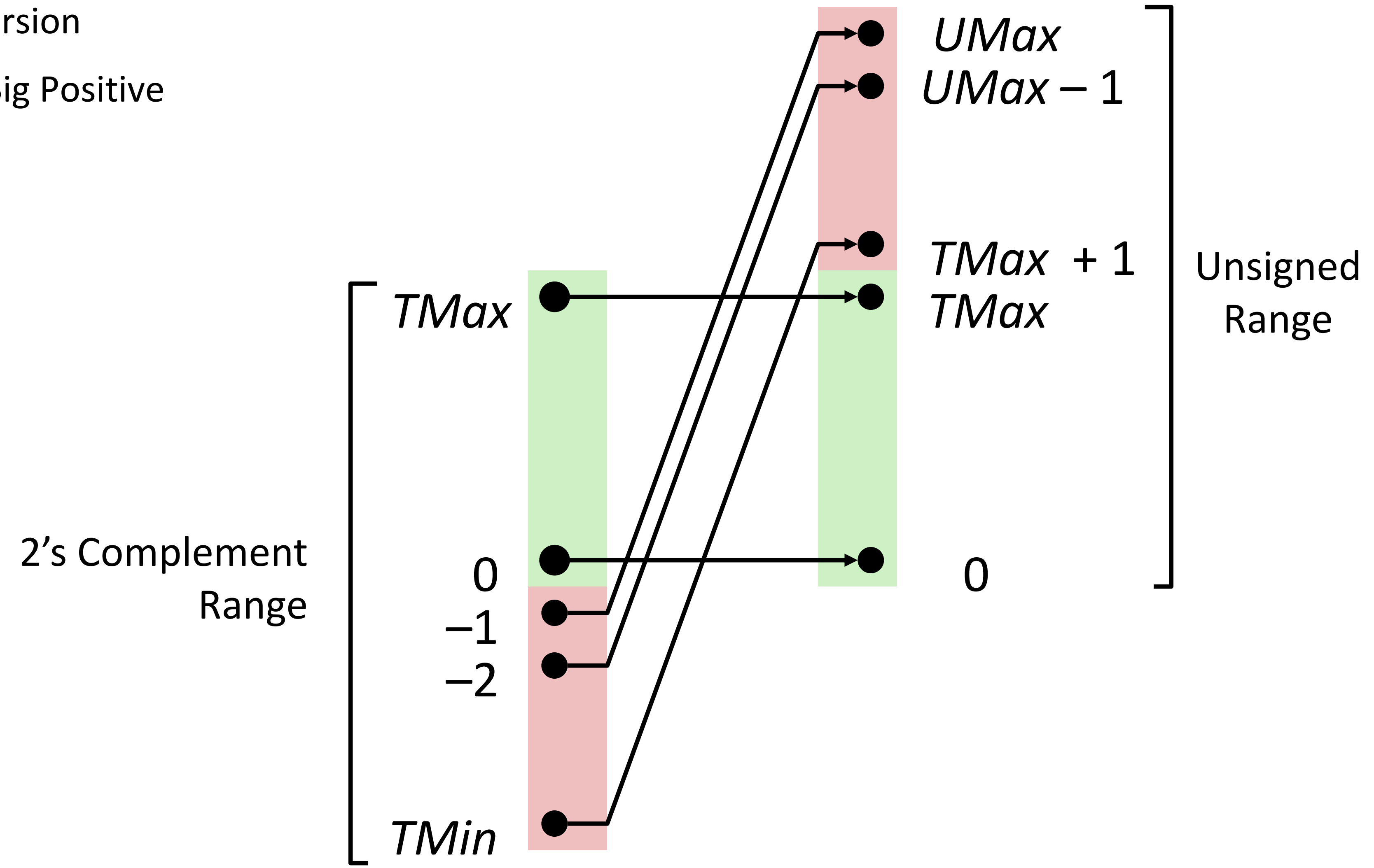




Large negative weight
becomes
Large positive weight

Conversion Visualized

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



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

Expression Evaluation

- 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, UMIN = 0, UMAX = 4,294,967,295**

  int

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1 2147483648U	<	unsigned
-1	-2	>	signed
(unsigned)-1 1111···1111	-2 1111···1110	>	unsigned
2147483647 0111···1111	2147483648U 1000···0000	<	unsigned
2147483647	(int) 2147483648U −2147483648	>	signed

Casting Signed ↔ Unsigned: Basic Rules

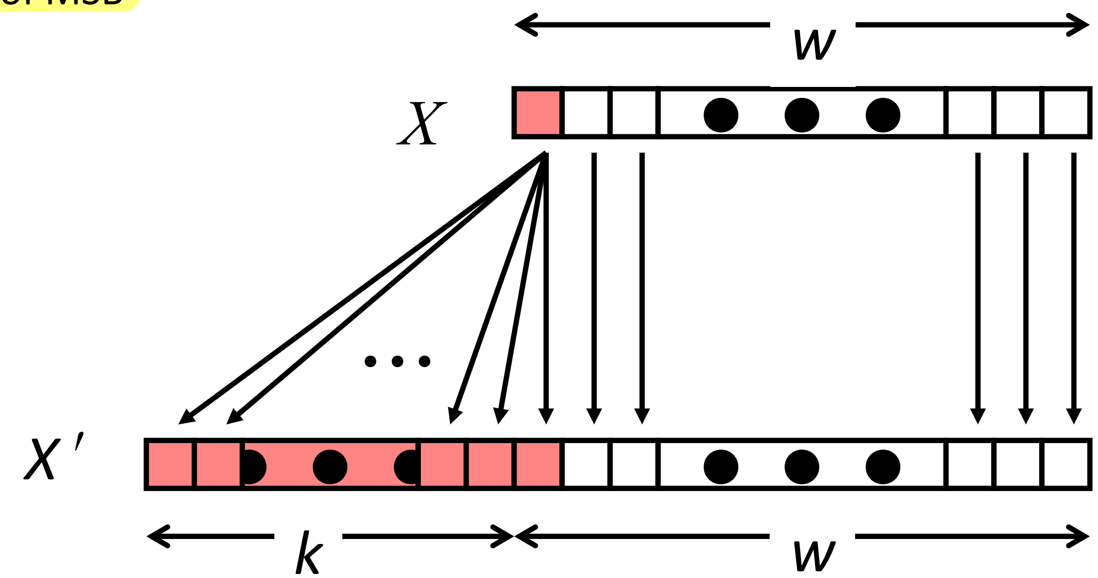
- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w
- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

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**

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



Decimal: -2
Binary (length: 4) 1110
Binary (length: 8) 1111 1110

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

Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Practice

Print the result of ‘a’ and ‘b’

```
int a = 40000 * 40000; // 1600000000
int b = 50000 * 50000; // 2500000000 ??
```

2147483647

- Hint: how to print integers

```
printf ("a: %d\n", a);
printf ("b: %d\n", b);
```

Compare the following constants.

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned

- Hint: how to define unsigned variables

```
int c1 = 0;
unsigned int c2 = 0U;
```

Today: Bits, Bytes, and Integers

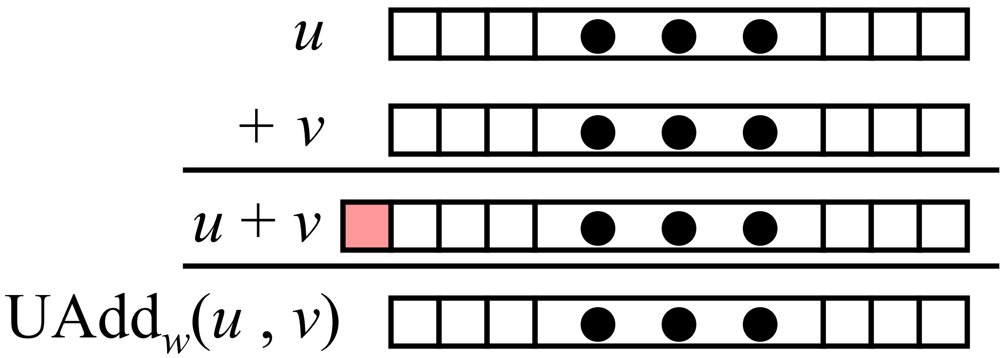
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

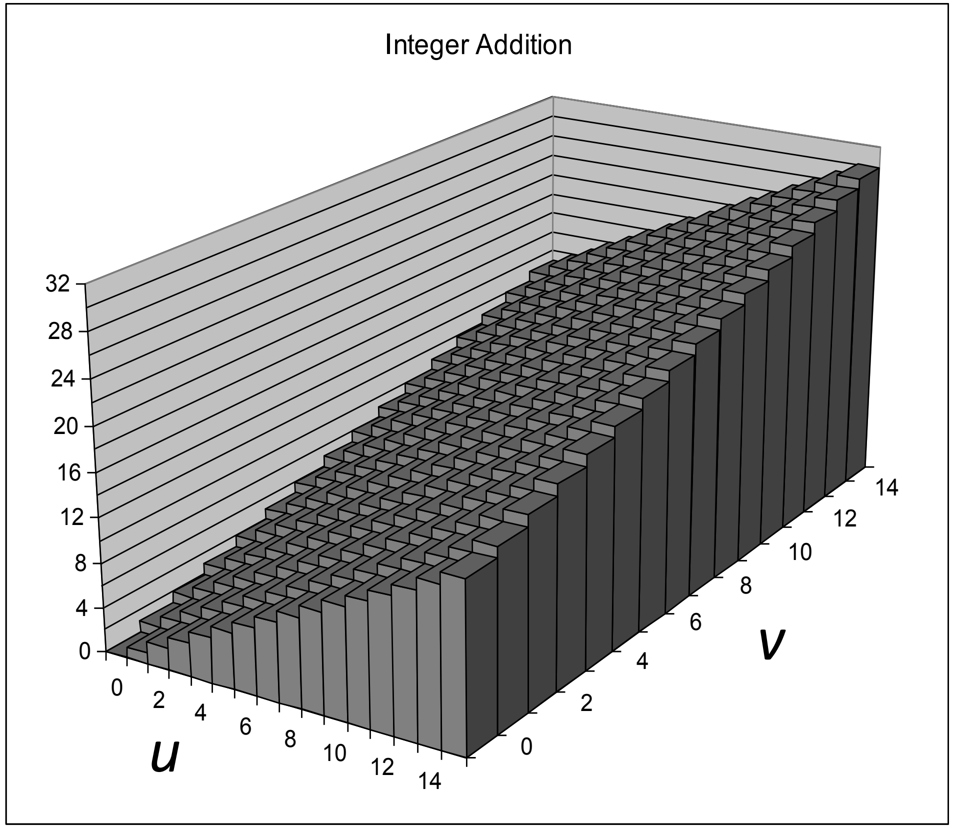
Visualizing (Mathematical) Integer Addition

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

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

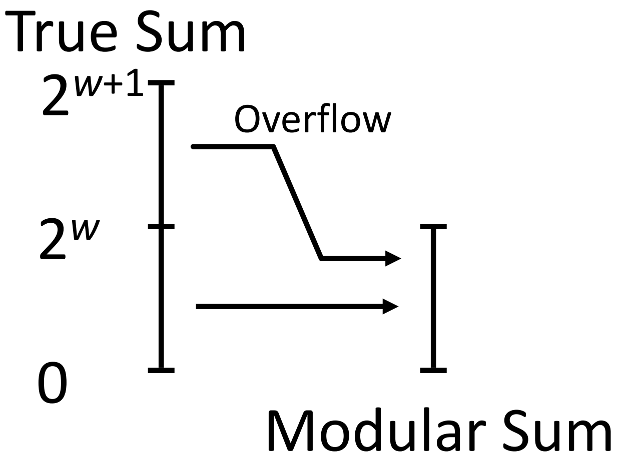
True sum

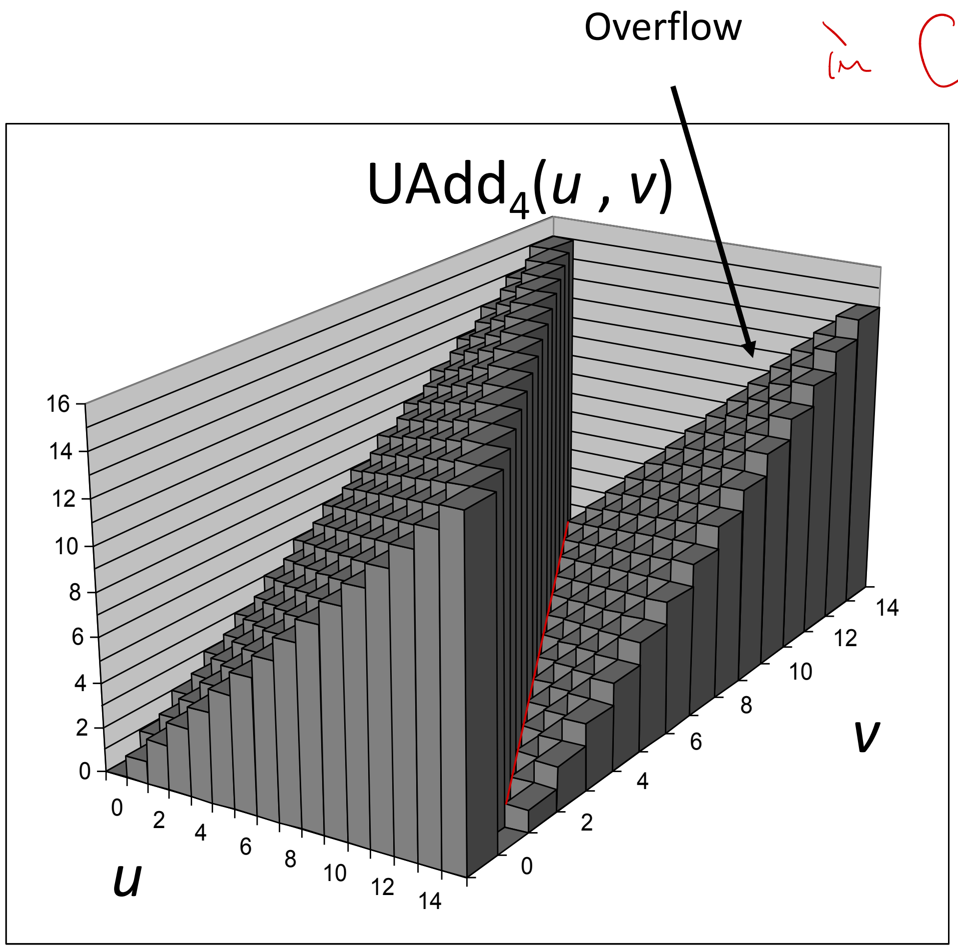


Visualizing Unsigned Addition

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

in C





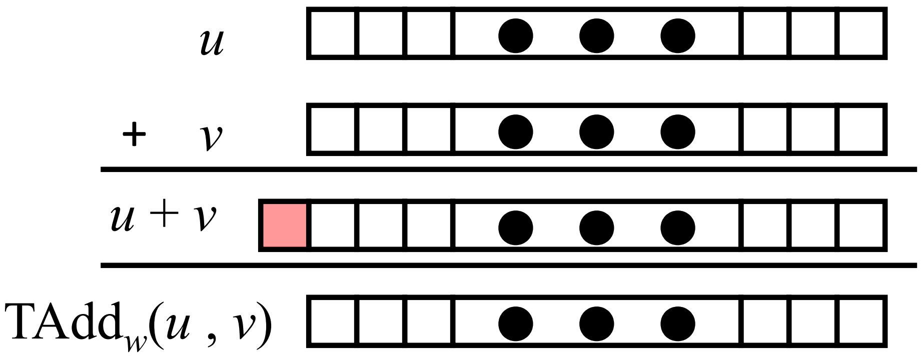
Two's Complement Addition

Signed

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:

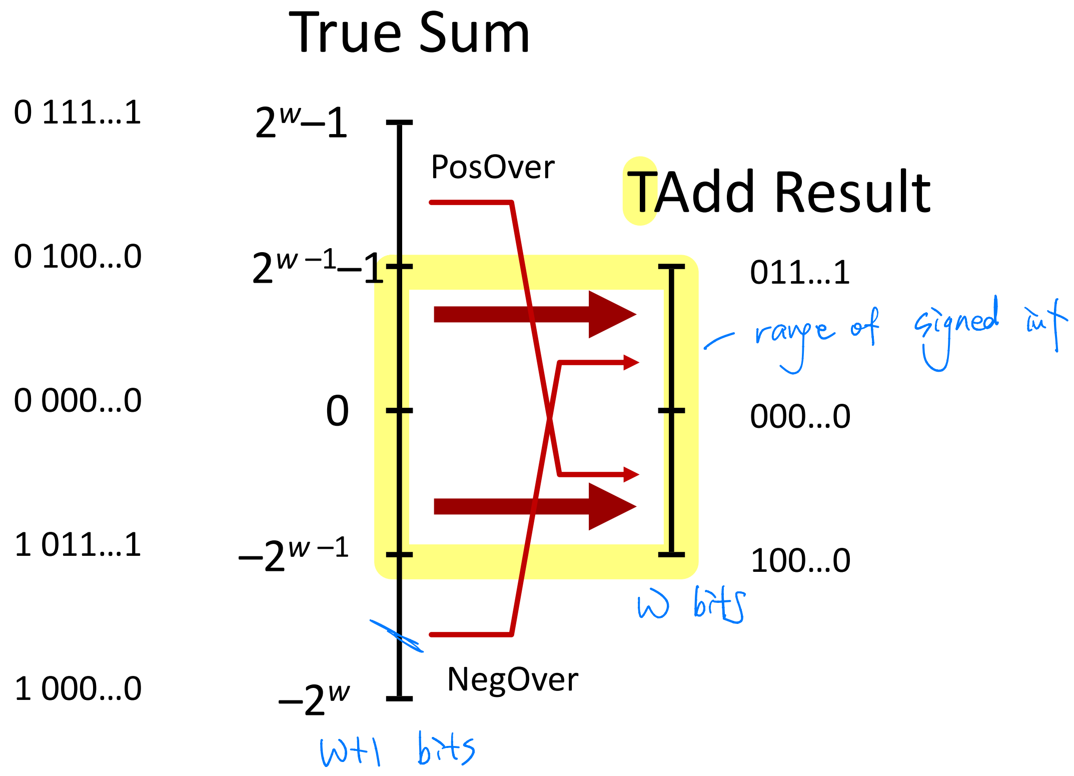
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

TAdd Overflow

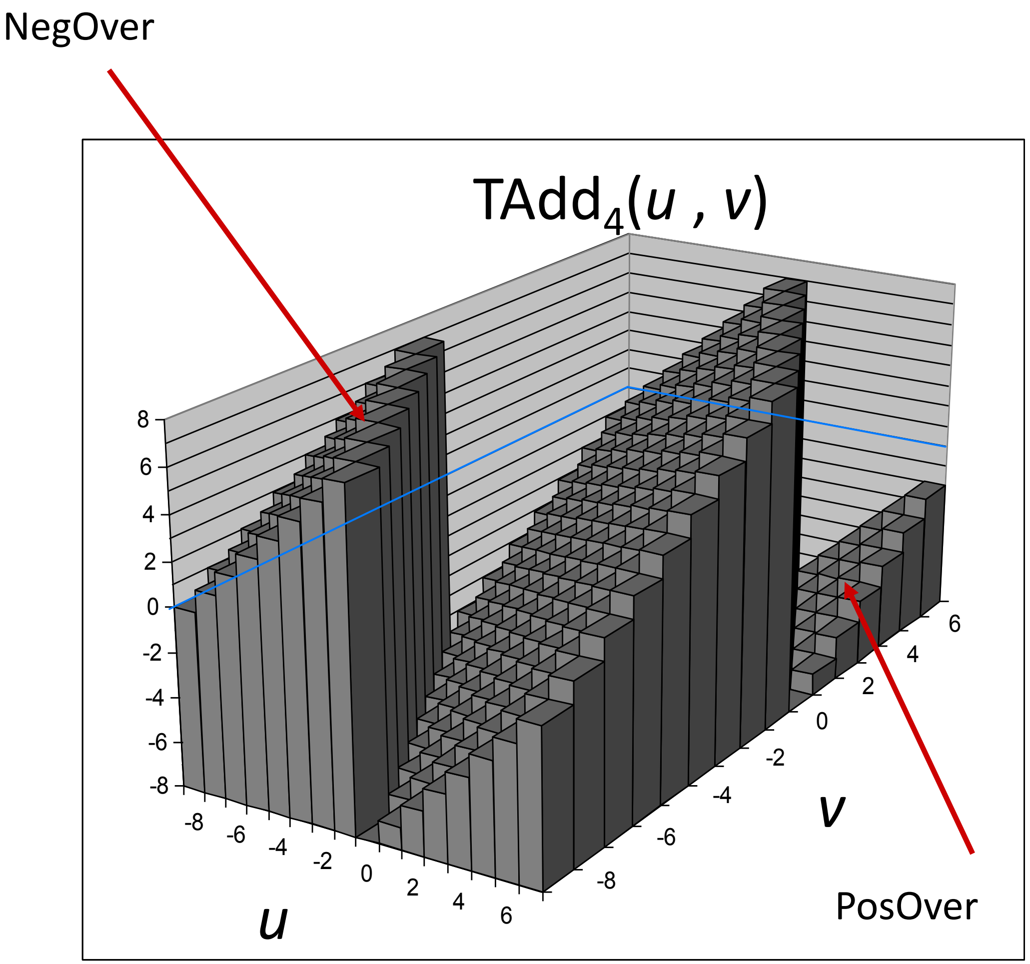
Functionality

- 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 $\geq 2^{w-1}$ PosOver
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$ NegOver
    - Becomes positive
    - At most once



Multiplication

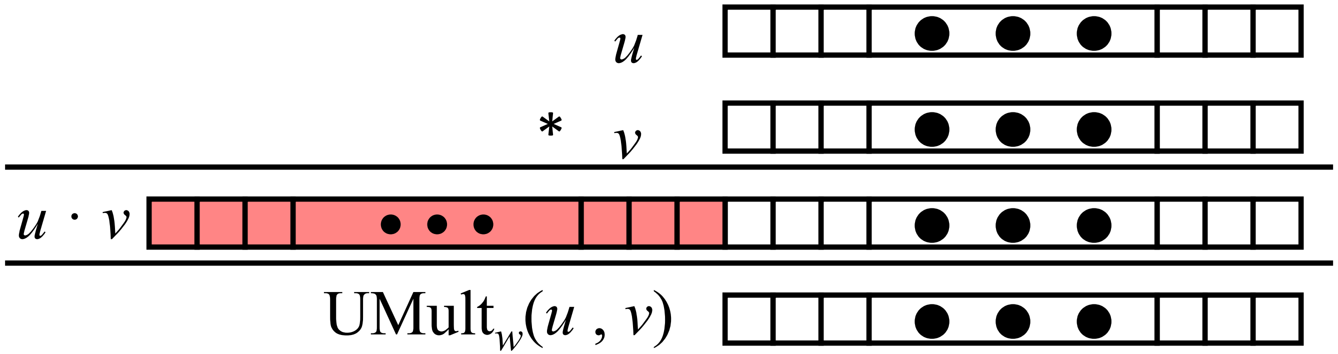
- **Goal: Computing Product of w-bit numbers x, y**
  - Either signed or unsigned
- **But, exact results can be bigger than w bits**
  - Unsigned: up to $2w$ bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - Result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

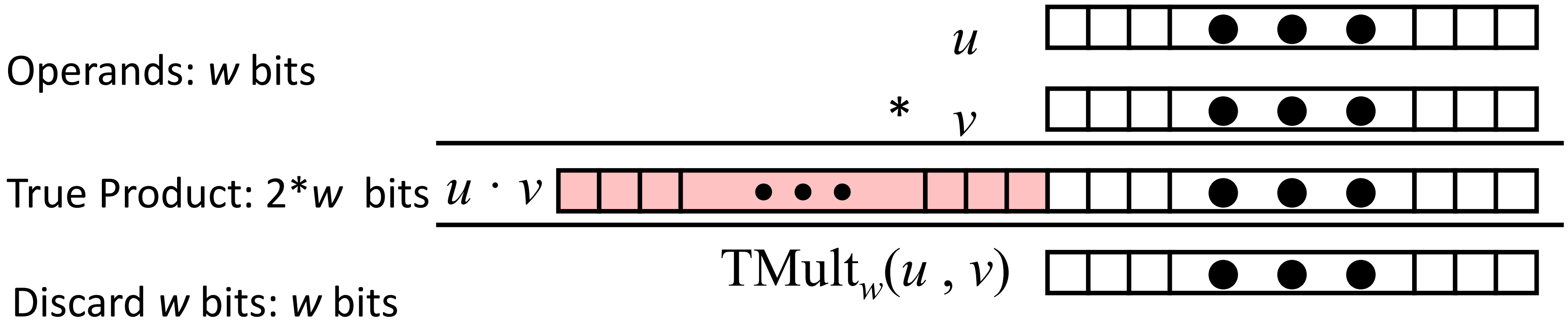
Discard *w* bits: *w* bits



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

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

Signed Multiplication in C

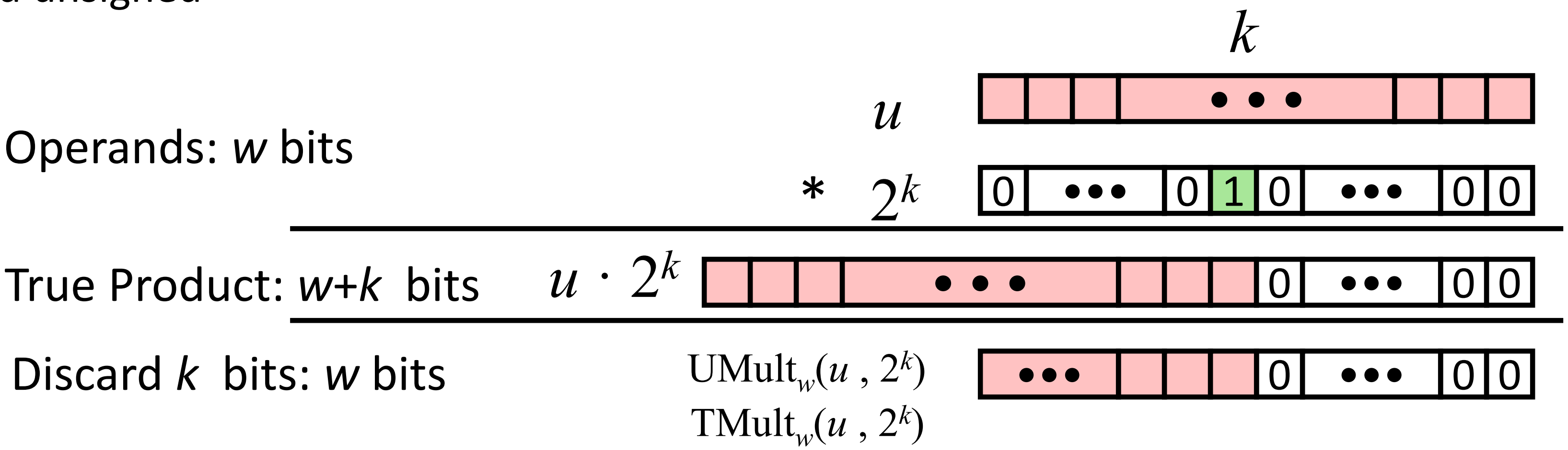


- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication

Power-of-2 Multiply with Shift

Operation

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



Examples

- `u << 3`   `==`   `u * 8`

$((u \ll 1) + u) \ll 3$

- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Q. How can we convert `u * 24` into shift operations?

$u * (8+(2+1))$

Compiled Multiplication Code

C Function

```
long mul12(long x)
{
  return x*12;
}
```

$2^2 \times (2+1)$

Compiled Arithmetic Operations

```
leaq (%rax,%rax,2), %rax
salq $2, %rax
```

Explanation

```
t <- x+x*2
return t << 2;
```

faster than (*4)!

- **C compiler automatically generates shift/add code when multiplying by constant**