

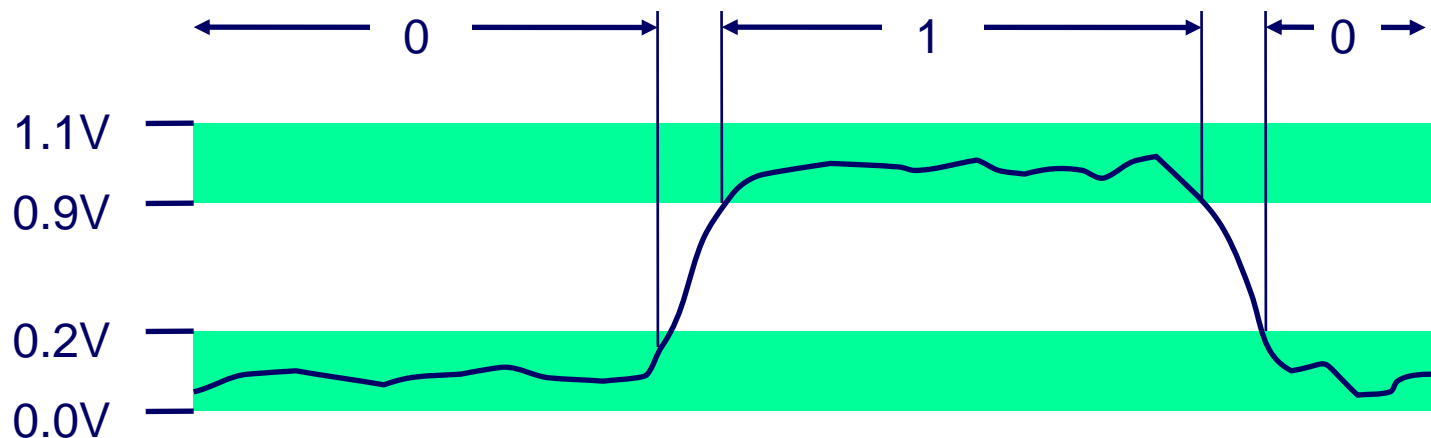
Computer Systems Organization

Topic 2

Based on chapter 2 from Computer Systems
by Randal E. Bryant and David R. O'Hallaron

Everything is bits

- Each bit is 0 or 1 (binary digits)
- Form basis of digital revolution
- Why bits? Electronic Implementation
 - Storing/performing computations is simple/reliable
 - Easy to store with bistable elements
 - Reliably transmitted on noisy and inaccurate wires



Everything is bits

- Single bit may not be useful but bit patterns do (groups of bits)
- 3 important representations of numbers
 - Unsigned encodings based on binary notation
 - Two's complement encoding to represent signed integers
 - Floating point encoding are base-2 version for representing real numbers
- Limited number of bits to encode numbers can have surprising effects e.g., $200 * 300 * 400 * 500$ yields -884901888 (32 bit representation)

Binary representation

- Base 2 Number Representation
 - Represent 15213_{10} as 11101101101101_2
 - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
 - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

How to convert

- $11 = (1011)_2 = 2^3 * 1 + 2^2 * 0 + 2^1 * 1 + 2^0 * 1$
- $11/2 = 5 \text{ (1)}$
- $5/2 = 2 \text{ (1)}$
- $2/2 = 1 \text{ (0)}$
- $1/2 = 0 \text{ (1)}$

- $12 = (1100)_2 = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0$
- $12/2 = 6(0)$
- $6/2 = 3(0)$
- $3/2 = 1(1)$
- $1/2 = 0(1)$

How to convert (fraction)

- Convert 0.8125
- $.8125 * 2 = 1.6250$ (1)
- $.6250 * 2 = 1.250$ (1)
- $.250 * 2 = 0.5$ (0)
- $.5 * 2 = 1.0$ (1)
- $0 * 2 = 0$ (0)
- Soln: 0.11010
- Converting back: $1*2^{-1} + 1*2^{-2} + 0 + 1*2^{-3} + 0 = .5 + .25 + 0 + .0625 + 0 = 0.8125$

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

Hex Notation

- $314156 = 19634.16 + 12$
- $19634 = 1227.16 + 2$
- $1227 = 76.16 + 11$
- $76 = 4.16 + 12$
- $4 = 0.16 + 4$
- Hence 0x4CB2C
- What is binary and hex notation for 158 ?
- What is $0x605c + 0x5$?
- Please practice...

Data sizes

- Each computer has word size indicating nominal size of pointer data
- Virtual address is encoded by such a word, hence word size determines size of virtual address space
- For machine with w -bit word size, virtual address can range from 0 to $(2^w)-1$, giving program access to at most 2^w bytes
- 32 bit word size limits virtual address space to 4GB [4×10^9 bytes] while 64 bit leads to 16 exabytes i.e., 1.84×10^{19} bytes
- Most 64-bit machines can run programs compiled to use for 32-bit machines i.e., backward compatible
- 32 bit vs. 64 bit programs [rather than machine distinction lies in how the program is compiled]

Typical Data Representations in C

C Data Type	Typical 32-bit	Typical 64-bit
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	4	8
<code>float</code>	4	4
<code>double</code>	8	8
<code>long double</code>	–	–
<code>pointer</code>	4	8

Most data types encode signed values unless prefixed by unsigned

Exception for char – need to declare signed char

Addressing and Byte Ordering

- In almost all machines, a multi-byte object stored as a contiguous sequence of bytes
 - E.g., variable `x` of type `int` has address `0x100` i.e., address expression `&x` is `0x100`. Assuming `int` is 4 bytes, `x` would be stored in location `0x100`, `0x101`, `0x102` and `0x103`.
- Two notations – big endian vs. little endian
- Number `0x01234567` stored as `{01}{23}{45}{67}` in big endian notation while stored as `{67}{45}{23}{01}` in little endian notation for the addresses `0x100`, `0x101`, `0x102`, `0x103`.
- Most intel compatible machines are little endian while most IBM/Oracle machines are big endian

Representing code

- Consider the following C function:

```
int sum(int x, int y) {  
    Return x + y;  
}
```

- Following machine code generated when compiled
- Linux 32: 55 89 e5 8b 45 0c 03 45 08 c9 c3
- Windows: 55 89 e5 8b 45 0c 03 45 08 5d c3
- Instruction codings can be different – binary code is seldom portable across combinations of machines + OS
- From machine perspective – program is simply a sequence of bytes and has no/minimal information of original source program

Boolean Algebra

- 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

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors (strings of 0's and 1's of fixed length w) - operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
01000001	01111101	00111100	10101010

- All of the Properties of Boolean Algebra Apply
- Let a and b denote bit vectors $[a_{w-1}, a_{w-2}, \dots, a_0]$ and $[b_{w-1}, b_{w-2}, \dots, b_0]$.
- $a \& b$ would be a bit vector of length w , where the i th element would be $a_i \& b_i$

Representing & Manipulating Sets

- Representation

- Can encode any subset $\{0, \dots, w-1\}$ with a bit vector $[a_{w-1}, a_{w-2}, \dots, a_0]$
- represents subsets of Width w
- $a_j = 1$ 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 & (AND), | (OR), ~ (NOT), ^ (Exclusive-OR) Available in C
 - Apply to any “integral” data type
 - long, int, short, char, unsigned
 - View arguments as bit vectors
- Examples (Char data type)
 - ~0x41 is 0xBE
 - $\sim 01000001_2$ is 10111110_2
 - ~0x00 is 0xFF
 - $\sim 00000000_2$ is 11111111_2
 - 0x69 & 0x55 is 0x41
 - $01101001_2 \& 01010101_2$ is 01000001_2
 - 0x69 | 0x55 is 0x7D
 - $01101001_2 | 01010101_2$ is 01111101_2

Contrast: Logic Operations in C

- Contrast to Logical Operators

- && (AND), || (OR), ! (NOT)

- View 0 as “False”

- Anything nonzero as “

- Always return

- **Early termina**

- Examples (ch

- !0x41 is 0x00

- !0x00 is 0x01

- !!0x41 is 0x01

- 0x69 && 0x55 is

- 0x69 || 0x55 is 0x01

- p && *p - avoids null pointer access since logical operators do not evaluate second argument if result can be determined with first

- Similarly a && 5/a will never cause a division by 0

Watch out for && vs. & (and || vs. |)... logical vs. bit level operators
one of the more common oopsies in C programming

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Integer Representations: Encoding Integers

Unsigned

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

Two's Complement

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

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

Sign
Bit

- C short 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

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

Two's-complement Encoding Example

$x =$ 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 $w = 16$ bits

	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

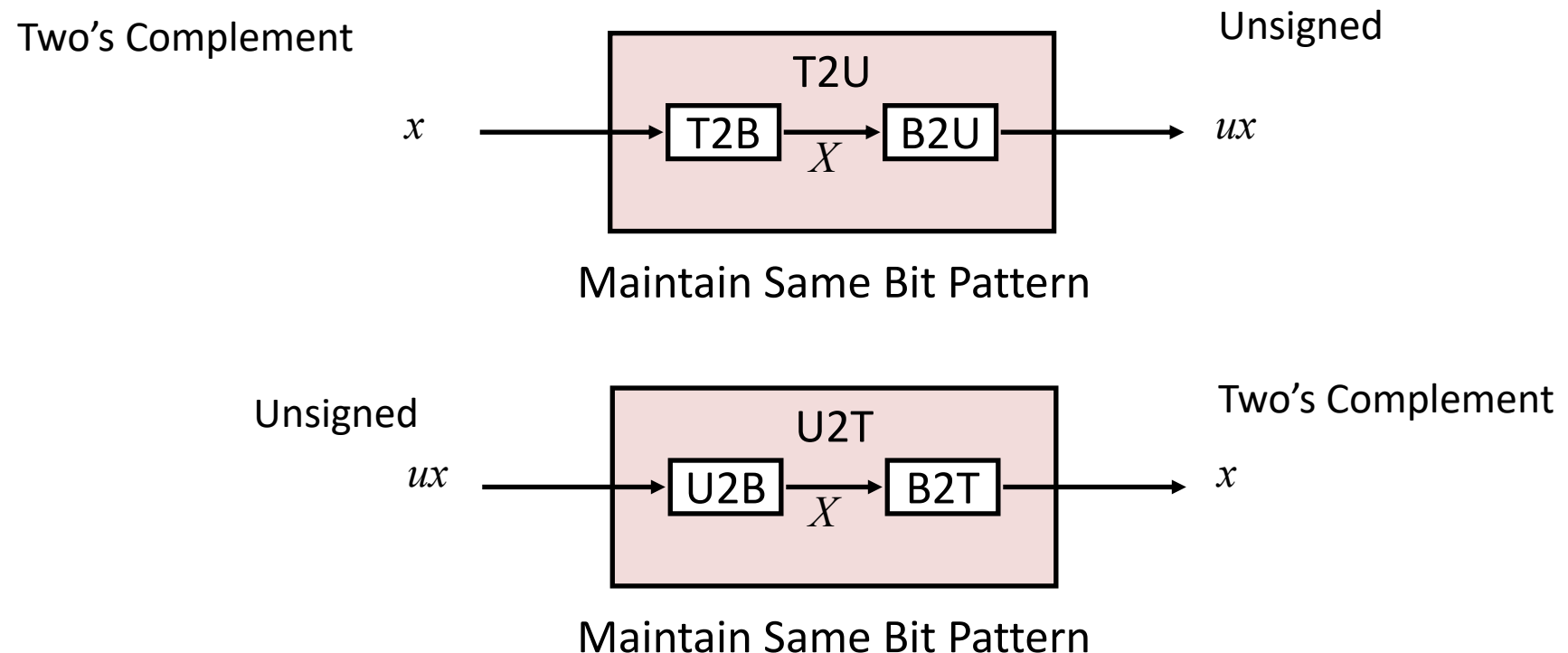
- #include <limits.h>
- Declares constants, e.g.,
 - ULONG_MAX
 - LONG_MAX
 - LONG_MIN
- Values platform specific

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

Mapping Between Signed & Unsigned



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

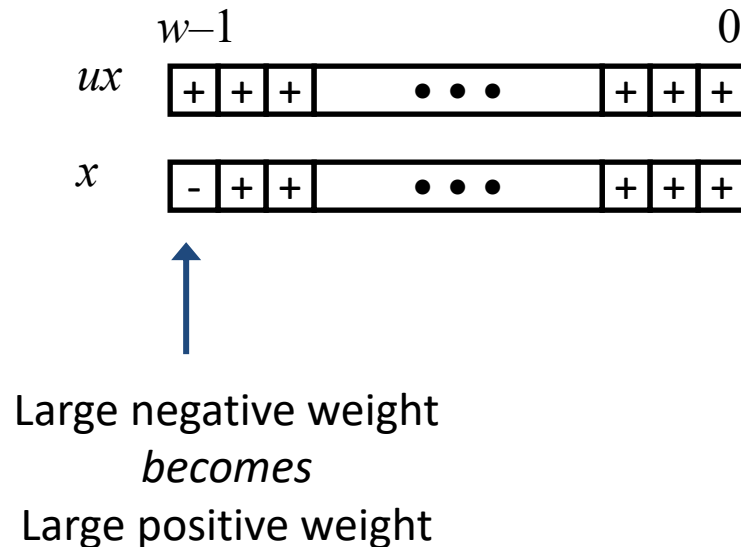
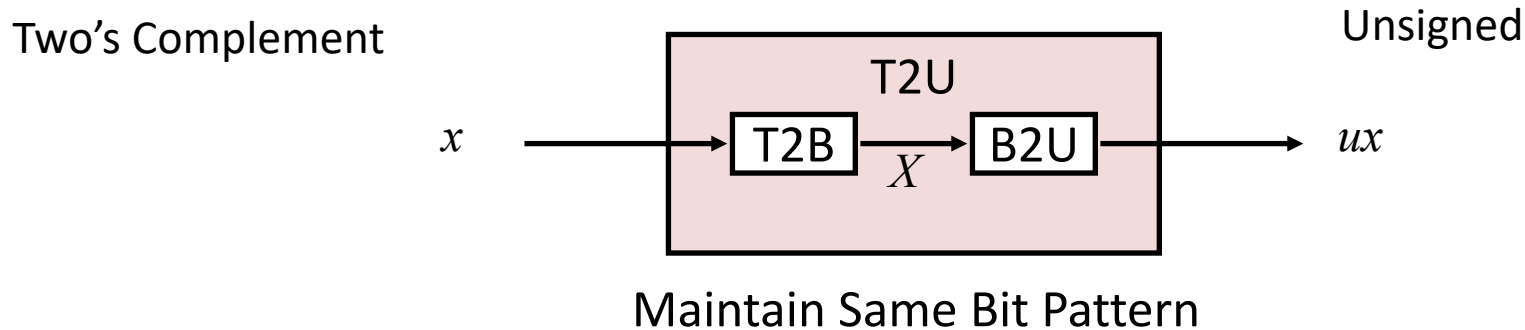
Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	→ T2U →	5
0110	6		6
0111	7	← U2T ←	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 \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	\longleftrightarrow =	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\longleftrightarrow +/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

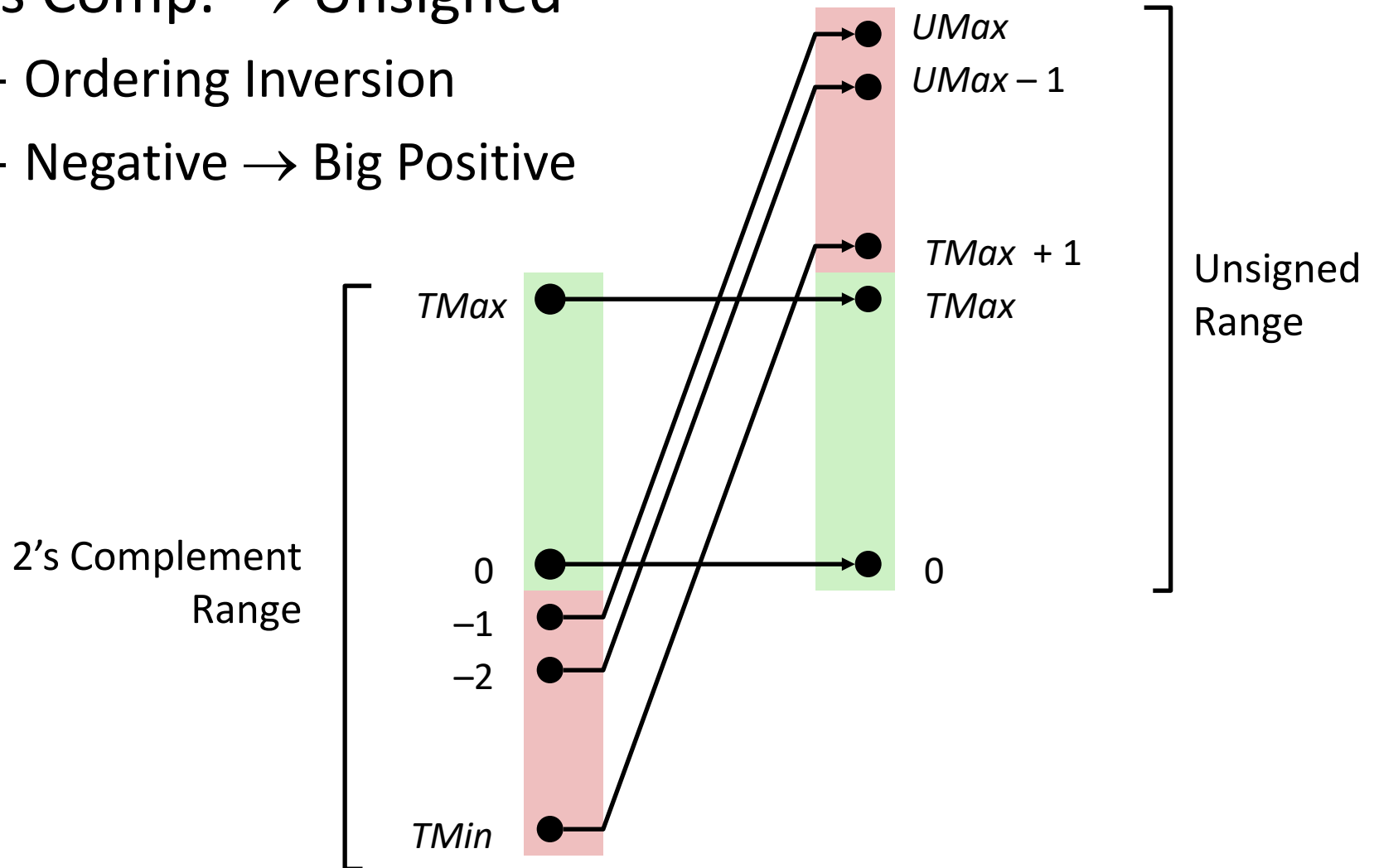
Relation between Signed & Unsigned



Add 2^w if $x < 0$,
otherwise remains same

Conversion Visualized

- 2's Comp. \rightarrow Unsigned
 - Ordering Inversion
 - Negative \rightarrow 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; /* cast to signed */
uy = ty; /* cast to unsigned */
```

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

Casting Surprises

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

Summary: Casting Signed \leftrightarrow 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`!!

Sign Extension

- Task:

- Given w -bit signed integer x

for sign extension just add msb
that many times

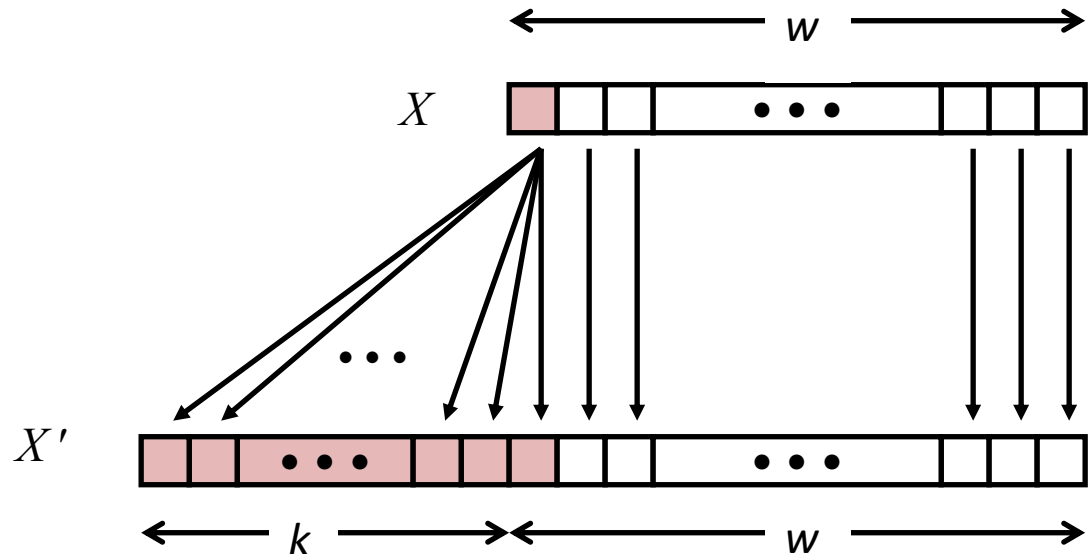
- Convert it to $w+k$ -bit integer with same value

- Rule:

- Make k copies of sign bit:

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

k copies of MSB



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

Truncation of number

- When truncating a w bit number to a k -bit number, we drop the high order $w-k$ bits
- Result: $x' = x \bmod 2^k$
- While similar property holds for twos-complement, it converts the most significant bit into a sign bit

```
int x = 53191
short sx = (short) x  /* -12345 */
int y = sx;           /* -12345 */
```

Truncation of number

- Summary:
- $B2U_k([x_{k-1}, x_{k-1}, \dots, x_0]) = B2U_k([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k$
- $B2T_k([x_{k-1}, x_{k-1}, \dots, x_0]) = U2T_k(B2U([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k)$

Summary: 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

Unsigned Addition

Operands: w bits



True Sum: $w+1$ bits



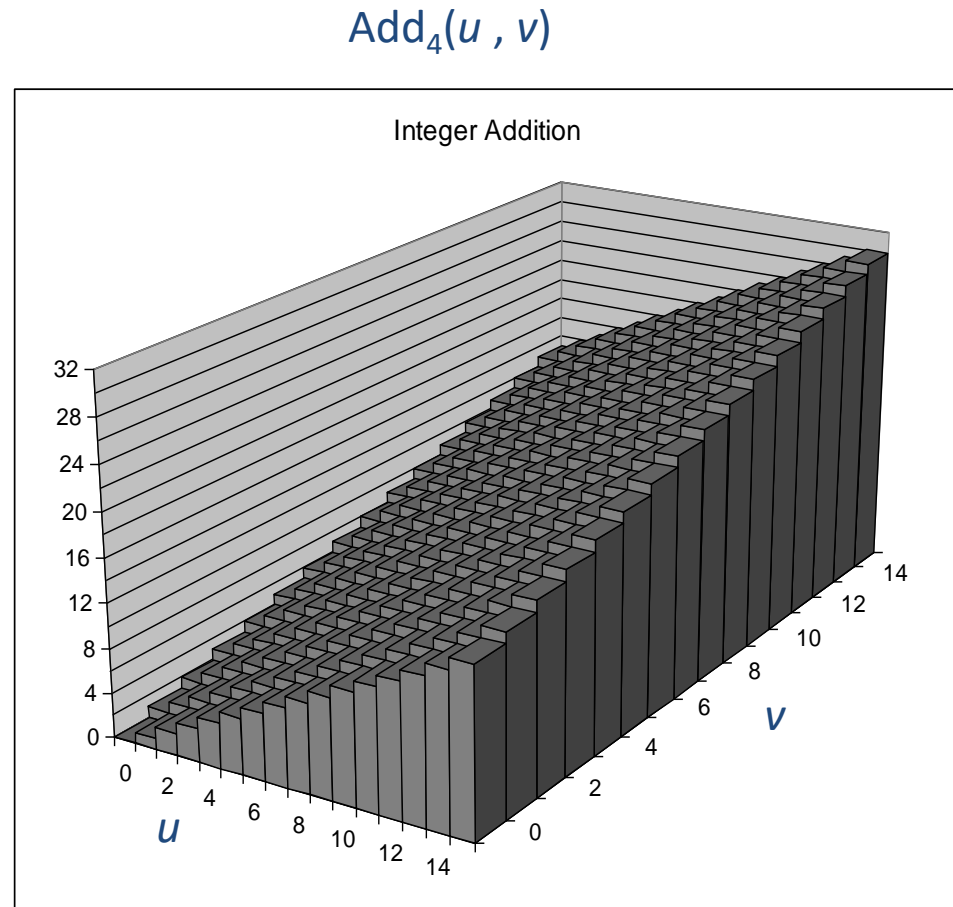
Discard Carry: w bits



- Standard Addition Function
 - Ignores carry output
- Implements Modular Arithmetic
$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

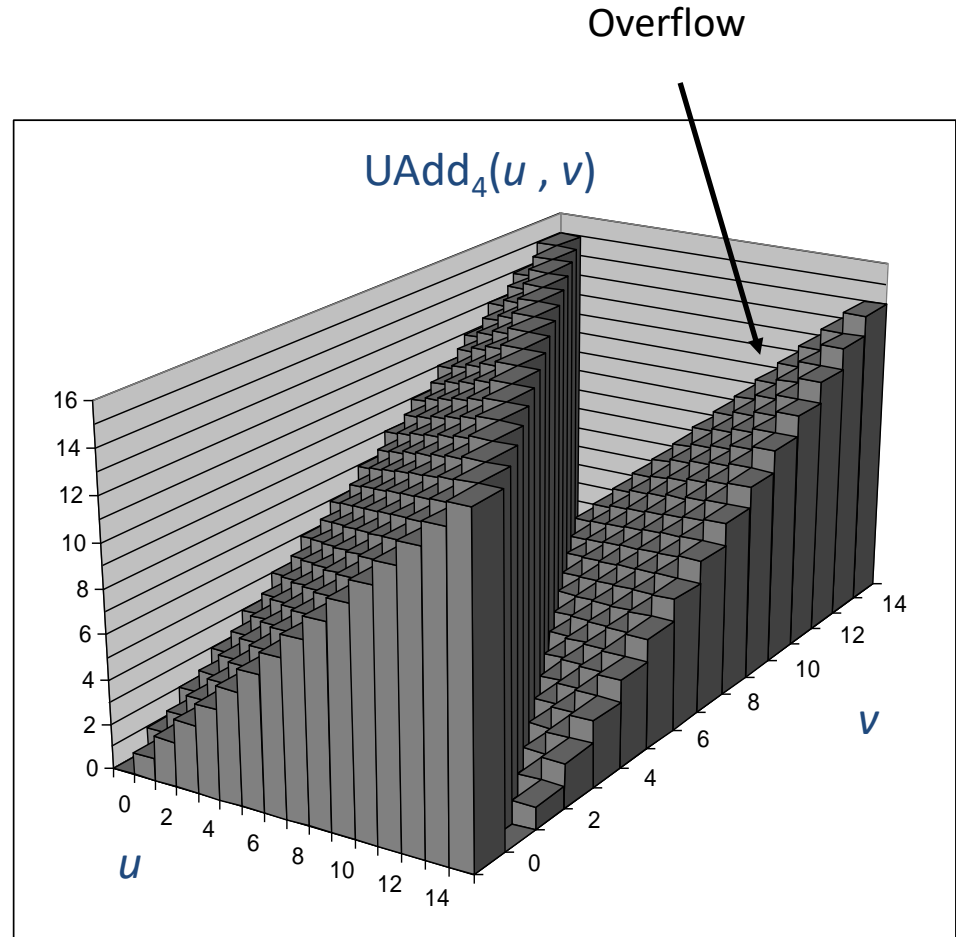
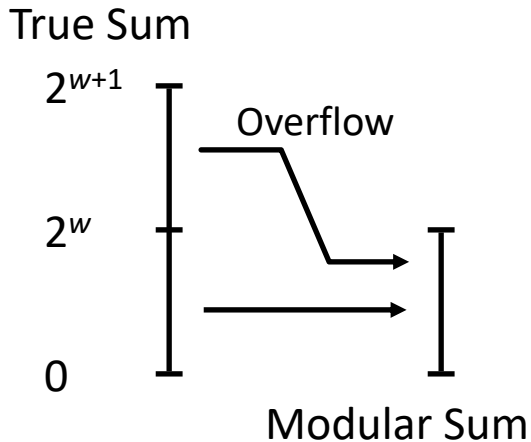
Visualizing (Mathematical) Integer Addition

- 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



Visualizing Unsigned Addition

- Wraps Around
 - If true sum $\geq 2^w$
 - At most once
 - Decrements by 2^w

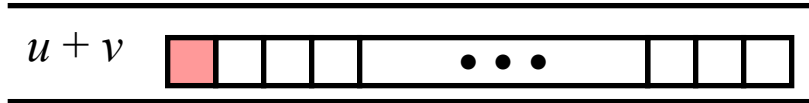


Two's Complement Addition

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;
```

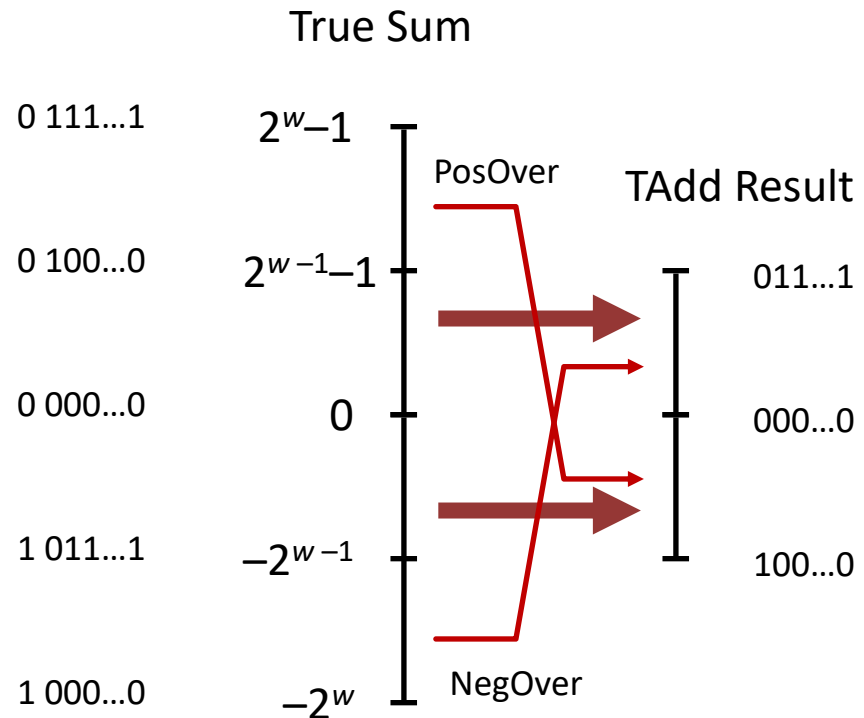
```
s = (int) ((unsigned) u + (unsigned) v);
```

```
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



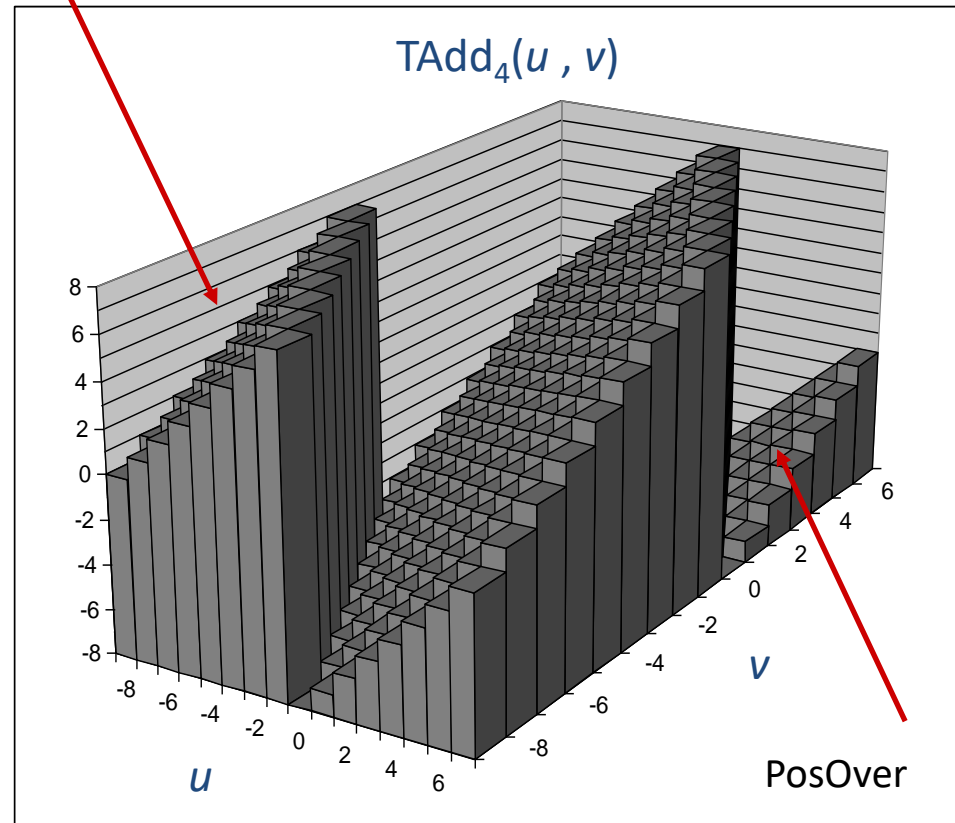
Two's Complement Addition

- In summary, subtract 2^w if positive overflow
- Add 2^w if negative overflow
- No changes if $2^{(w-1)} \leq \text{sum} < 2^w$
- For $w = 4$ bits,
 - $-8 [1000] + -5 [1011] = -13 [10011] = 3 [0011]$
 - $5 [0101] + 5 [0101] = 10 [01010] = -6 [1010]$

Visualizing 2's Complement Addition

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

NegOver



Two's Complement Negation

- Complement the bits, increment the result by 1
- 0101 [5] \rightarrow 1010 [-6] \rightarrow 1011 [-5]
- 1000 [-8] \rightarrow 0111 [7] \rightarrow 1000 [-8]
- ...

Multiplication

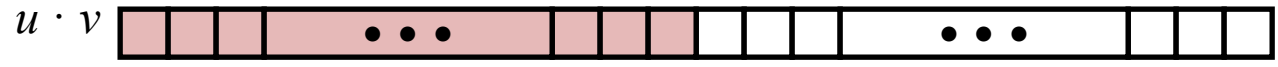
- Goal: Computing Product of w -bit numbers x, y
 - Either signed or unsigned
- But, exact results can be bigger than w bits
 - Unsigned: up to $2w$ bits
 - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
 - Two's complement min (negative): Up to $2w-1$ bits
 - Result range: $x *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



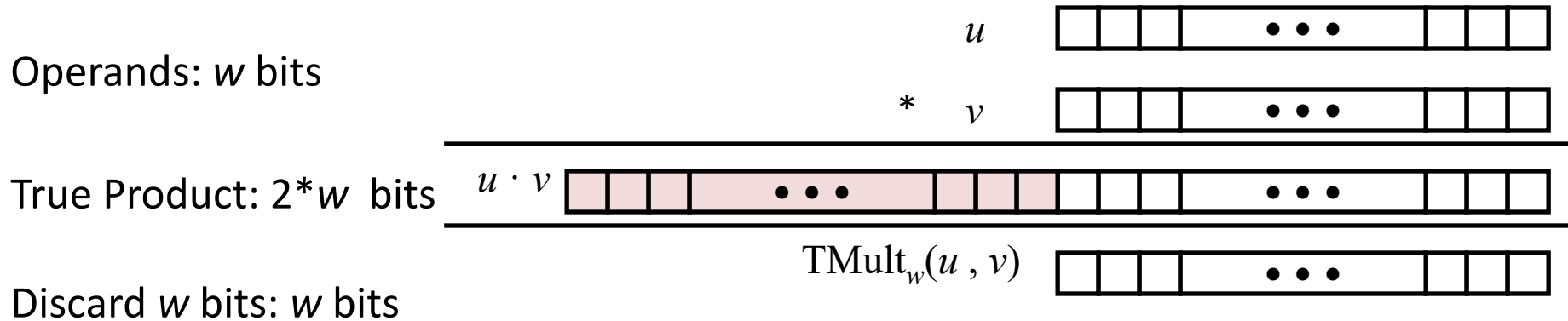
Discard w bits: w bits



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

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

Signed Multiplication in C



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

Example

- Unsigned: $5 [101] * 3 [011] = 15 [01111] \rightarrow 7$
[111] Truncated

- 101

- 011

- 101

- 101

- 000

- 01111

Example

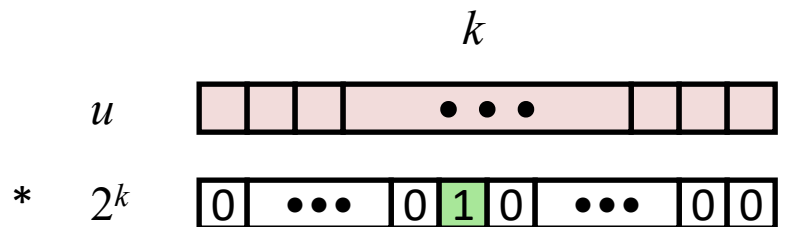
- Two's C: $-3 [101] * 3 [011] = -9 [110111] \rightarrow -1 [111]$
Truncated
- Need to sign extend and then multiply
- 111101
- 000011
- 111101
- 111101
- 000000
- 000000
- 000000
- 000000
- -----
- 000101**110111**

Power-of-2 Multiply with Shift

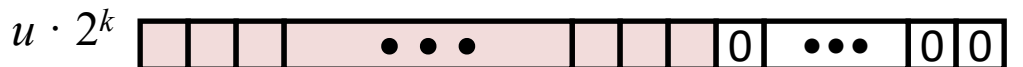
- Operation

- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned

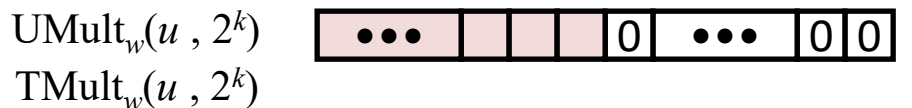
Operands: w bits



True Product: $w+k$ bits



Discard k bits: w bits

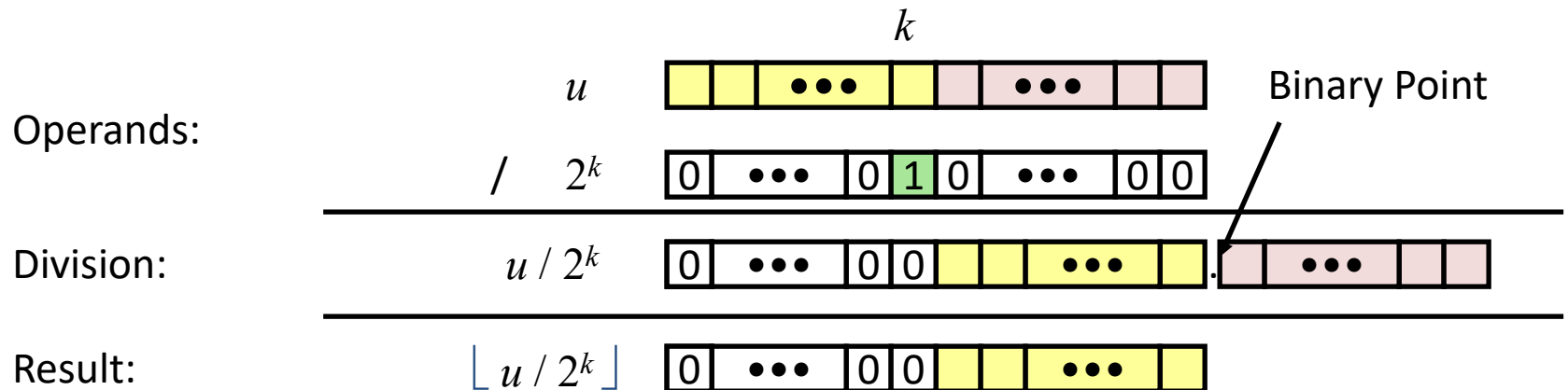


- Examples

- $u \ll 3 \quad == \quad u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- 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
 - $u \gg k$ 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

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)

Using 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)  
    . . .
```

