Integer Arithmetic

CMPU 224 – Computer Organization
Jason Waterman

Native Data Representations in C

- `char`, `short`, `int`, and `long` are "integer" types
  - Signed by default
    - `int x;`
  - Can declare as unsigned
    - `unsigned int x;`
- `float` and `double` are "real" types
- A pointer is a data type that holds a memory address

C Data Type	Typical 32-bit	x86-64
`char`	1	1
`short`	2	2
`int`	4	4
`long`	**4**	**8**
`float`	4	4
`double`	8	8
pointer	**4**	**8**

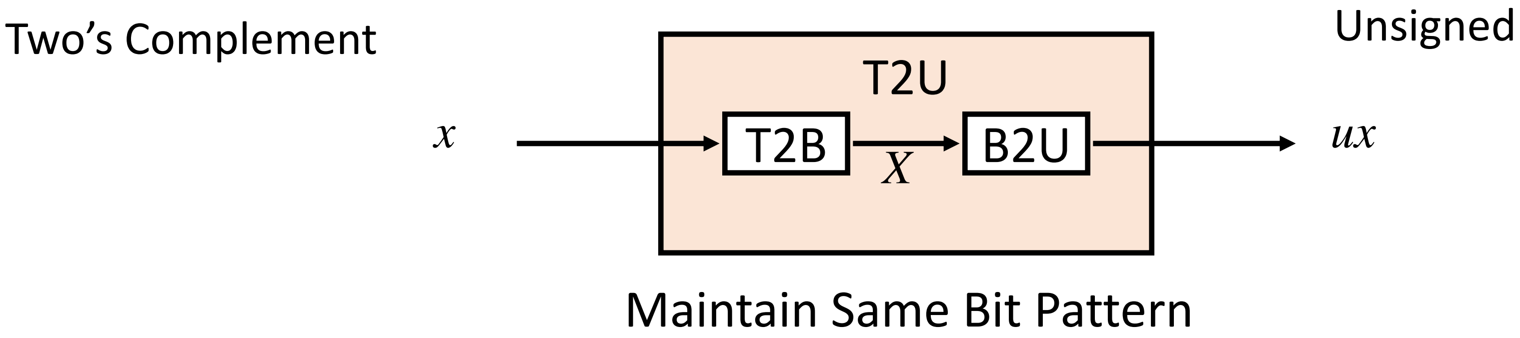
Values for Different Word Sizes

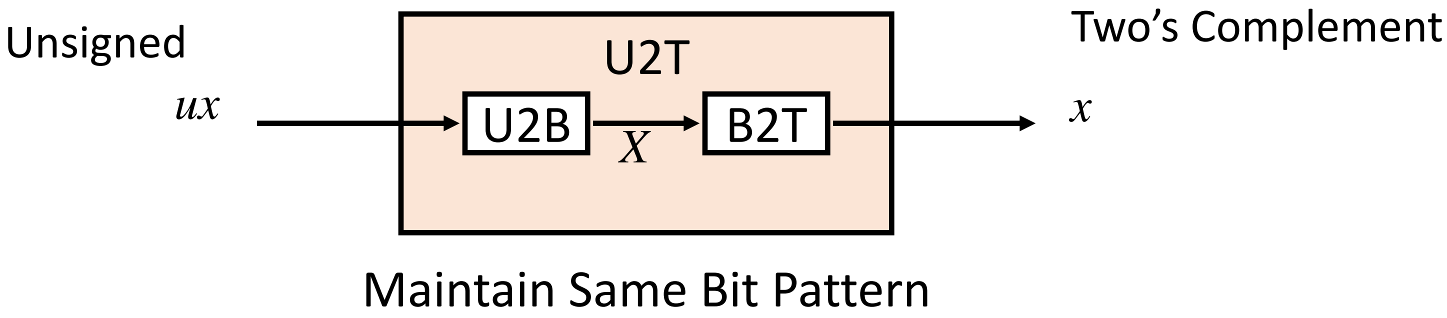
	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- 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 are platform specific

Mapping Between Signed & Unsigned





- Mappings between unsigned and two's complement numbers:
  Keep the same 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

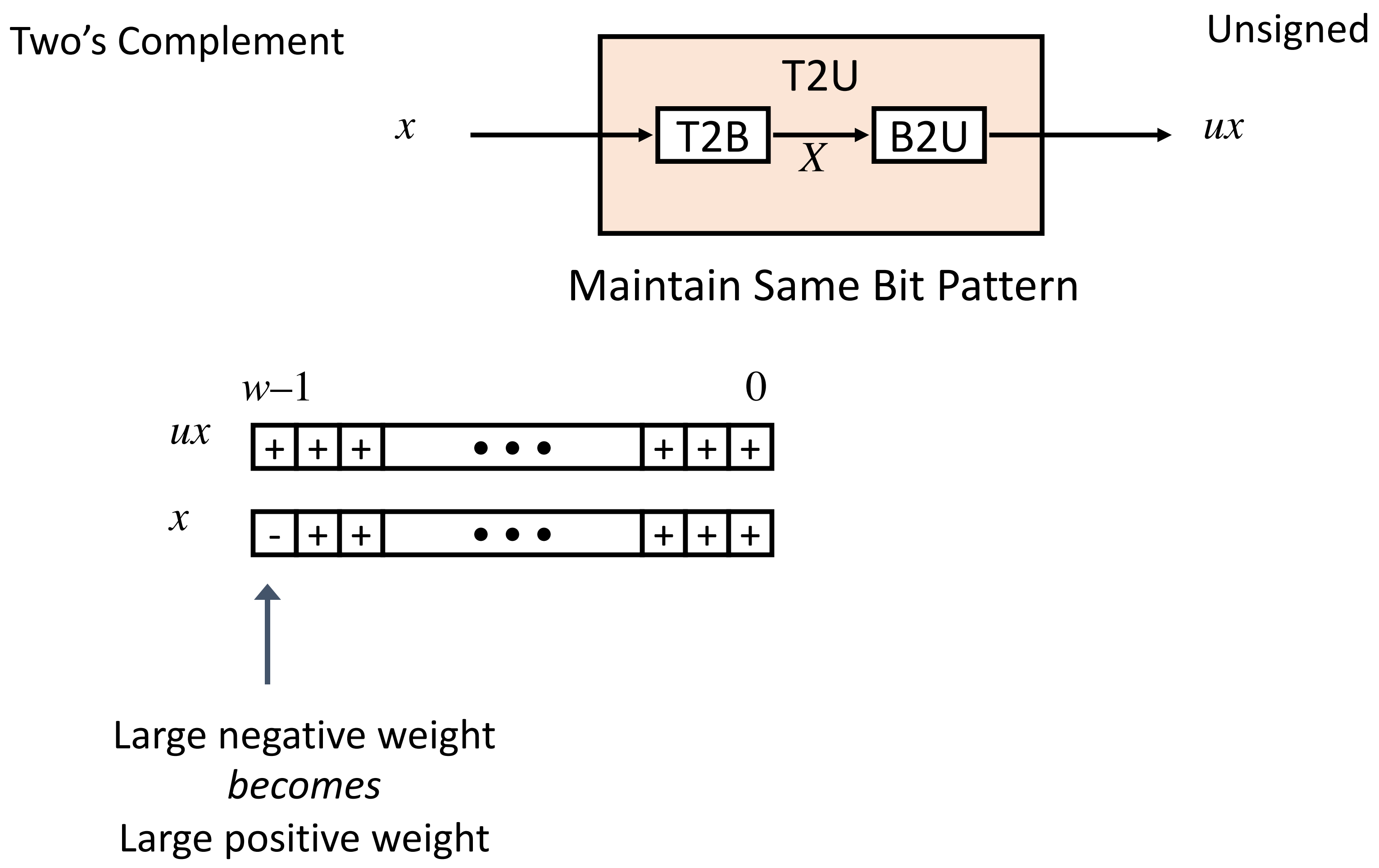
→ T2U →

← U2T ←

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

Relation between Signed & Unsigned



Signed vs. Unsigned in C

- Constants
  - Are by default considered to be **signed** integers
  - Unsigned if have "U" as suffix

```
0U, 4294967259U
```

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

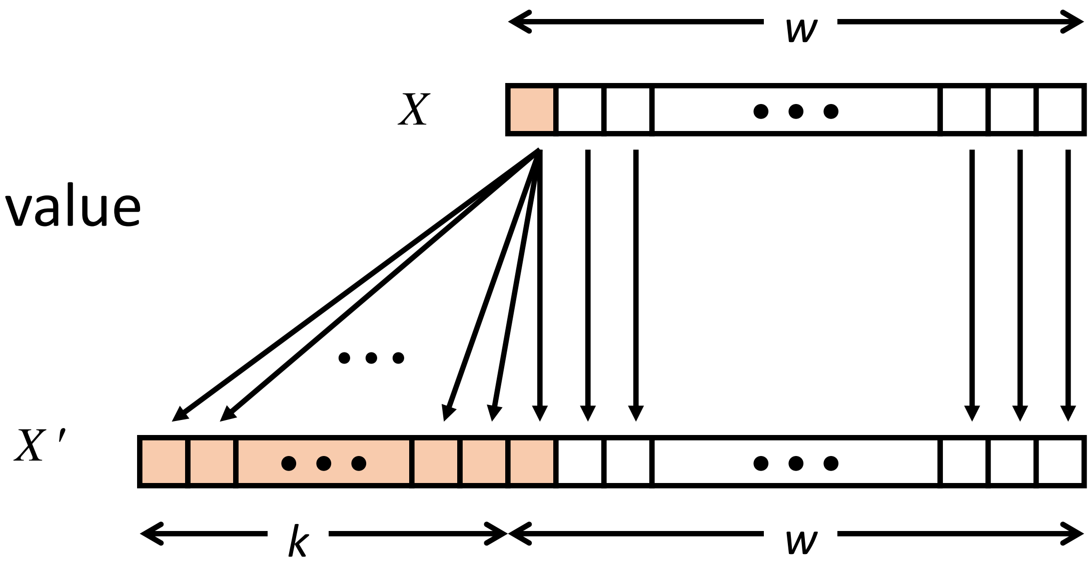
```
int tx, ty;
unsigned int ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

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

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



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

Expanding and Truncating Rules

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

- Truncating (e.g., `int` to `short`)
  - Unsigned/signed: high order bits are truncated (drop)
  - Result reinterpreted
  - For small numbers this yields expected behavior

    `11111010` → -6 8-bit two's complement

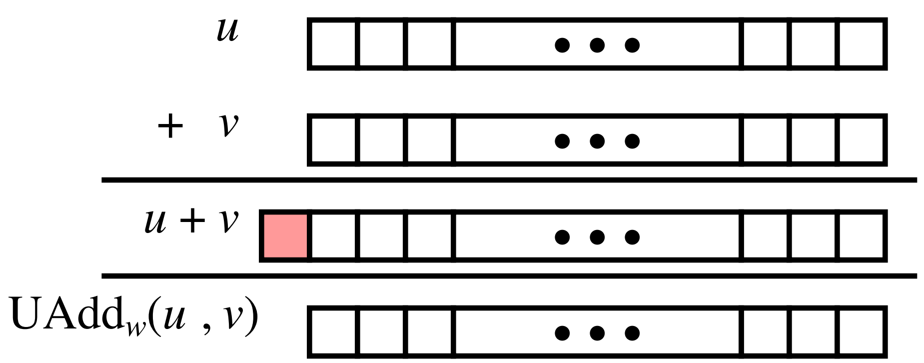
    `1010` → -6 4-bit two's complement
  - Overflow can result in a sign change

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

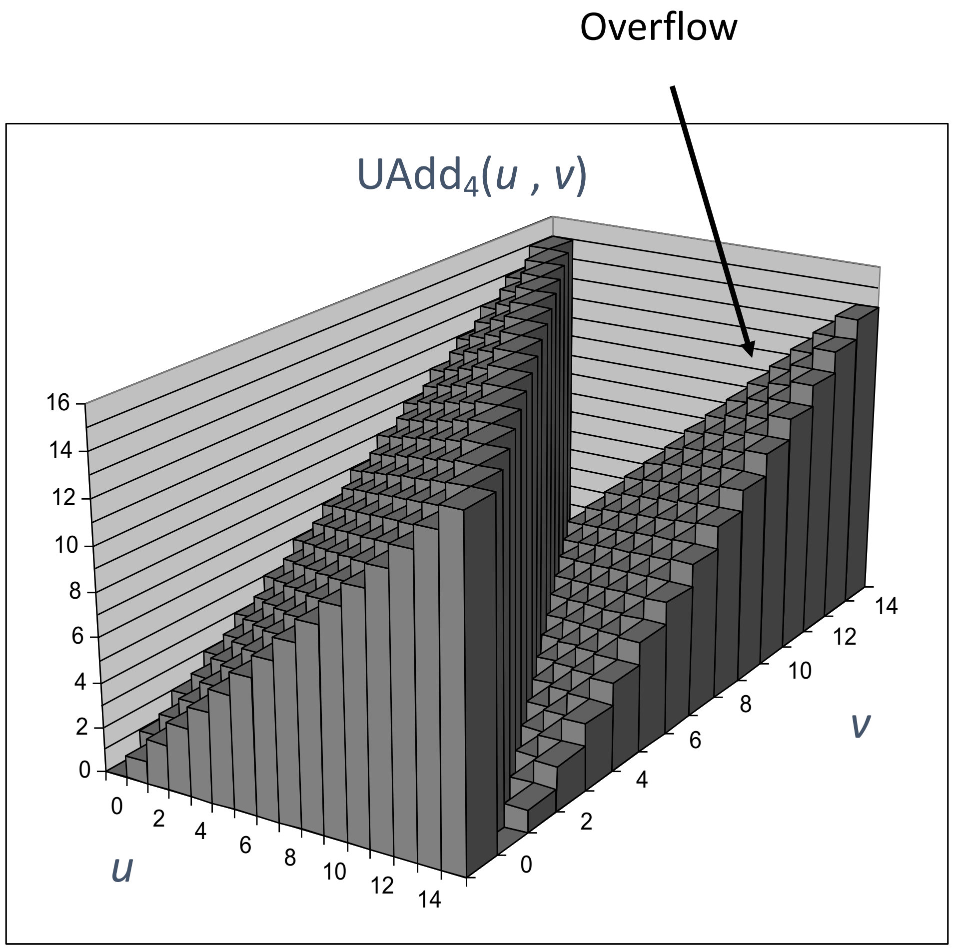


- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

Visualizing Unsigned Addition

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

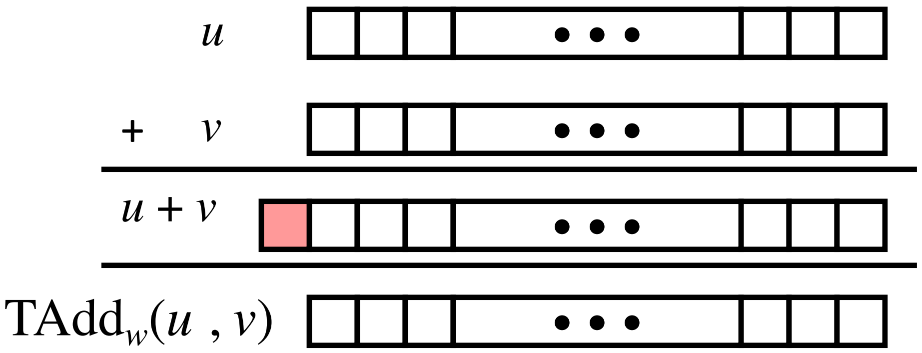


Two’s Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

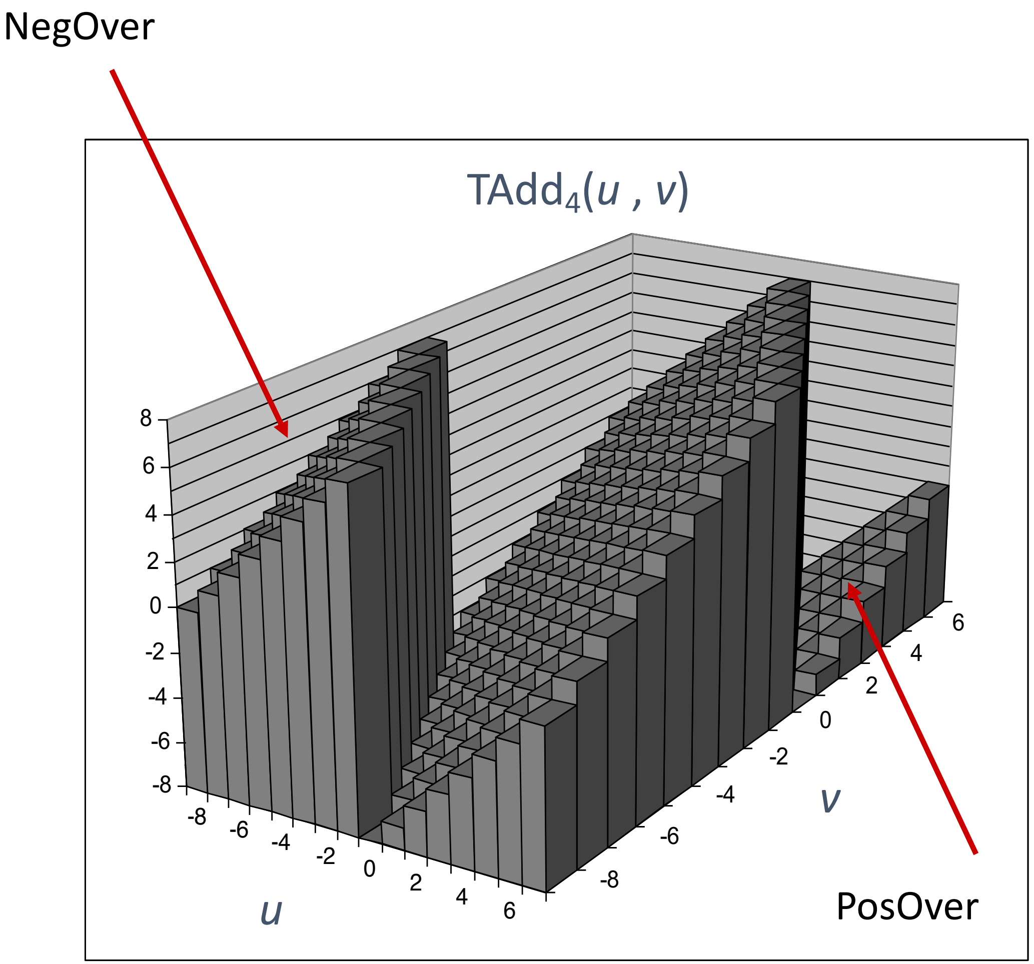
Discard Carry: *w* bits



- TAdd and UAdd have Identical Bit-Level Behavior

Visualizing 2's Complement Addition

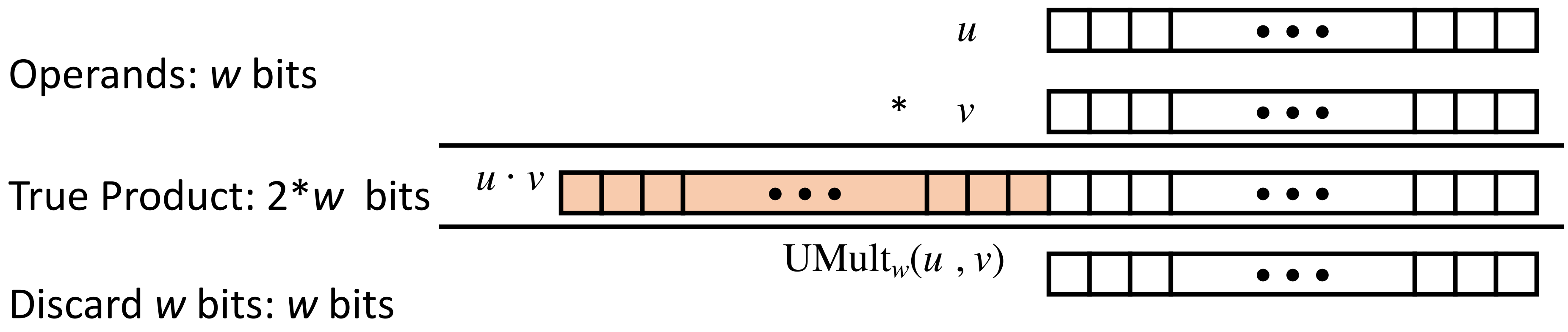
- Example Values
  - 4-bit two's comp
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$: **Positive Overflow**
    - Adding two positive numbers
      - Answer should be positive
      - Becomes negative
  - If sum $< -2^{w-1}$: **Negative Overflow**
    - Adding two negative numbers
      - Answer should be negative
      - Becomes positive



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
  - Two's complement min (negative): Up to $2w$-1 bits
  - Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
- So, maintaining exact results…
  - Would need to keep expanding word size with each product computed
  - Can be done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C



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

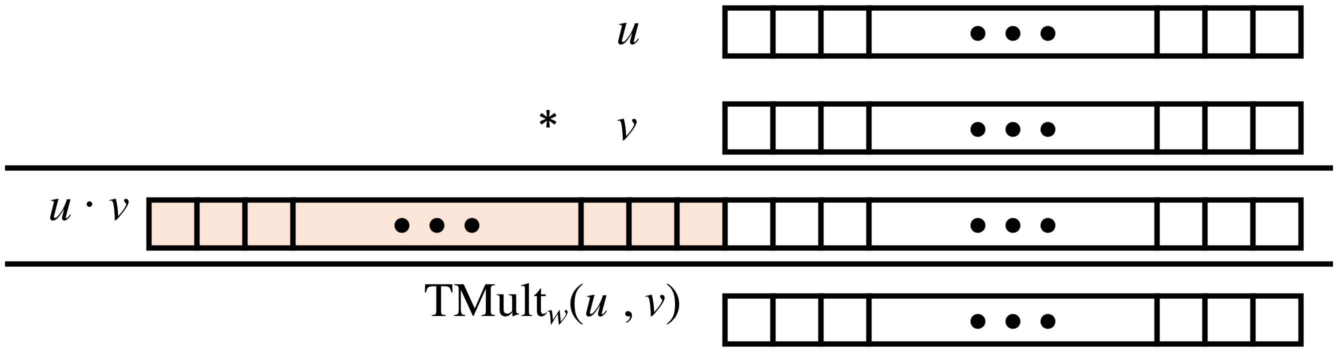
  $\mathrm{UMult}_w(u , v) = u \cdot v \mod 2^w$

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

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 ≥ data 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

Power-of-2 Multiply with Shift

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

- Examples
  - `u * 8 == u << 3`
  - `u * 24 == (u << 5) - (u << 3)`
  - 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 right shift
  - Rounds towards zero (truncates decimal part)

	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
  - **`u >> k`** gives $\lfloor$ **`u / 2`**^{k} $\rfloor$ (greatest integer less than)
  - Uses arithmetic right shift
  - Rounds down

	Division	Computed	Binary
x	-12,340.0	-12,340	11001111 11001100
x >> 1	-6,170.0	-6,170	*1*1100111 11100110
x >> 4	-771.25	-772	*1111*1100 11111100
x >> 8	-48.203125	-49	*11111111* 11001111
X >> 14	-0.75317382	-1	*11111111* *11111111*

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)

- Shifting:
  - Multiplying/Dividing by powers of 2
  - Logical right shift: shift in 0
  - Arithmetic right shift: shift in the sign bit