Chapter 2: Casting and Multiplication

Topics

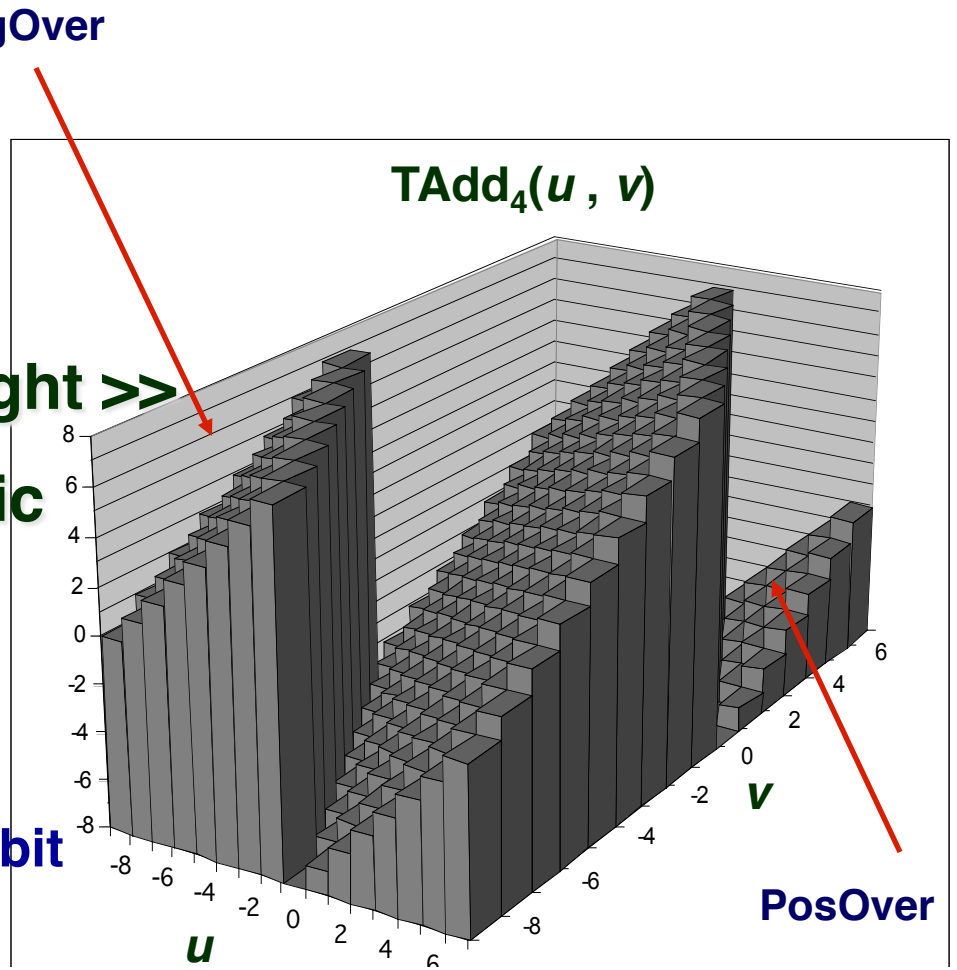
- Casting between unsigned and signed
- Sign extension
- Multiplication
- Division

Announcements

- Data Lab is due Friday Sept 12 by 11:55 pm
- Shifting Wednesday office hours for the remainder of the semester to 3-4 pm
  - Tuesday office hours remain the same 4-5 pm
- Prof. Han traveling to Washington DC next Tuesday
  - Guest lecture by Yogesh Virkar TA
- Essential that you read the textbook in detail & do the practice problems
  - Finish Chapter 2 (2.1-2.3)
  - Chapter 3 (3.1-3.11 and 3.13-3.14)

Recap…

- Logical operators &&, ||, !
- Bit shifting – left << and right >>
- Unsigned integer arithmetic
  - Overflow
- Signed integer arithmetic
  - Two's complement:
    - MSbit = -2^{w-1} if '1' = sign bit
  - Overflow
    - Two large positive #'s can sum to a negative # (PosOver)
    - Two large negative #'s can sum to a positive # (NegOver)



- Then, sum of u and v = TrueSum(u + v) - 2^w, given w bits
- Then, sum of u and v = TrueSum(u + v) + 2^w, given w bits

Checking for Signed Overflow in C

Approach #1

- Have the compiler insert checks every time a signed integer arithmetic operation is performed
- Use gcc compiler with –ftrapv flag

Approach #2

- Manually insert the checks yourself
- Can use macros, e.g. replace each add '+' with a macro ADD(A,B)
  - These macros essentially test the operands against the integer limits in the header file <limits.h> before doing the operation
- See CERT Web page for examples
  - https://www.securecoding.cert.org/confluence/display/seccode/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow?showComments=false
- Some compiler optimization may remove these code checks

Approach #3 : ignore and hope = risky strategy

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

⇒ 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

Casting Signed to Unsigned

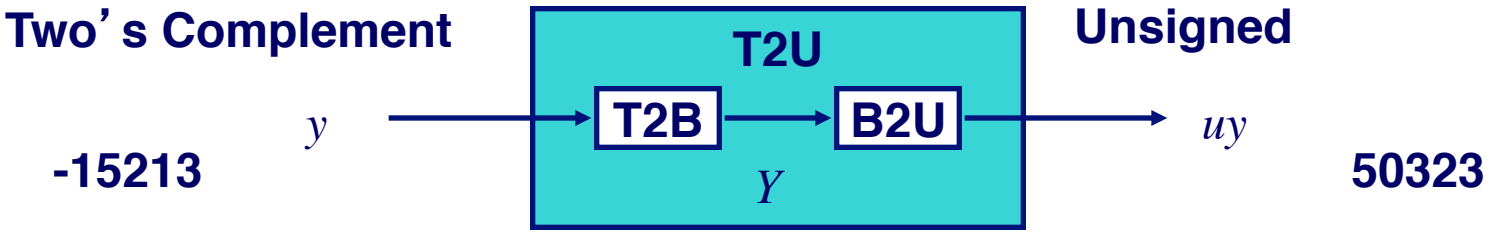
C Allows Conversions from Signed to Unsigned

```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

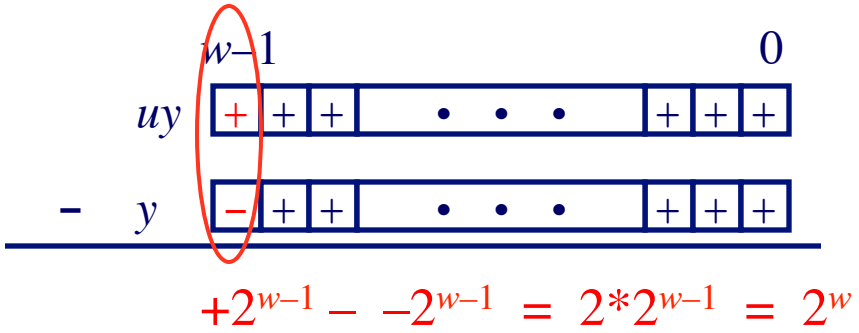
Resulting Value

- No change in bit representation
- Nonnegative values unchanged
  - *ux* = 15213
- Negative values change into (large) positive values
  - *uy* = 50323
- Because conversion between signed and unsigned is so confusing, languages like Java only permit signed integers

Relation between Signed & Unsigned



Maintain Same Bit Pattern



$$uy = \begin{cases} y & y \geq 0 \\ y + 2^w & y < 0 \end{cases}$$

Relation Between Signed & Unsigned

Weight	-15213		50323	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
32768	1	*-32768*	1	*32768*
Sum		**-15213**		**50323**

- $uy = y + 2 * 2^{15} = y + 2 * 32768 = y + 65536$ (y<0)

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 int 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

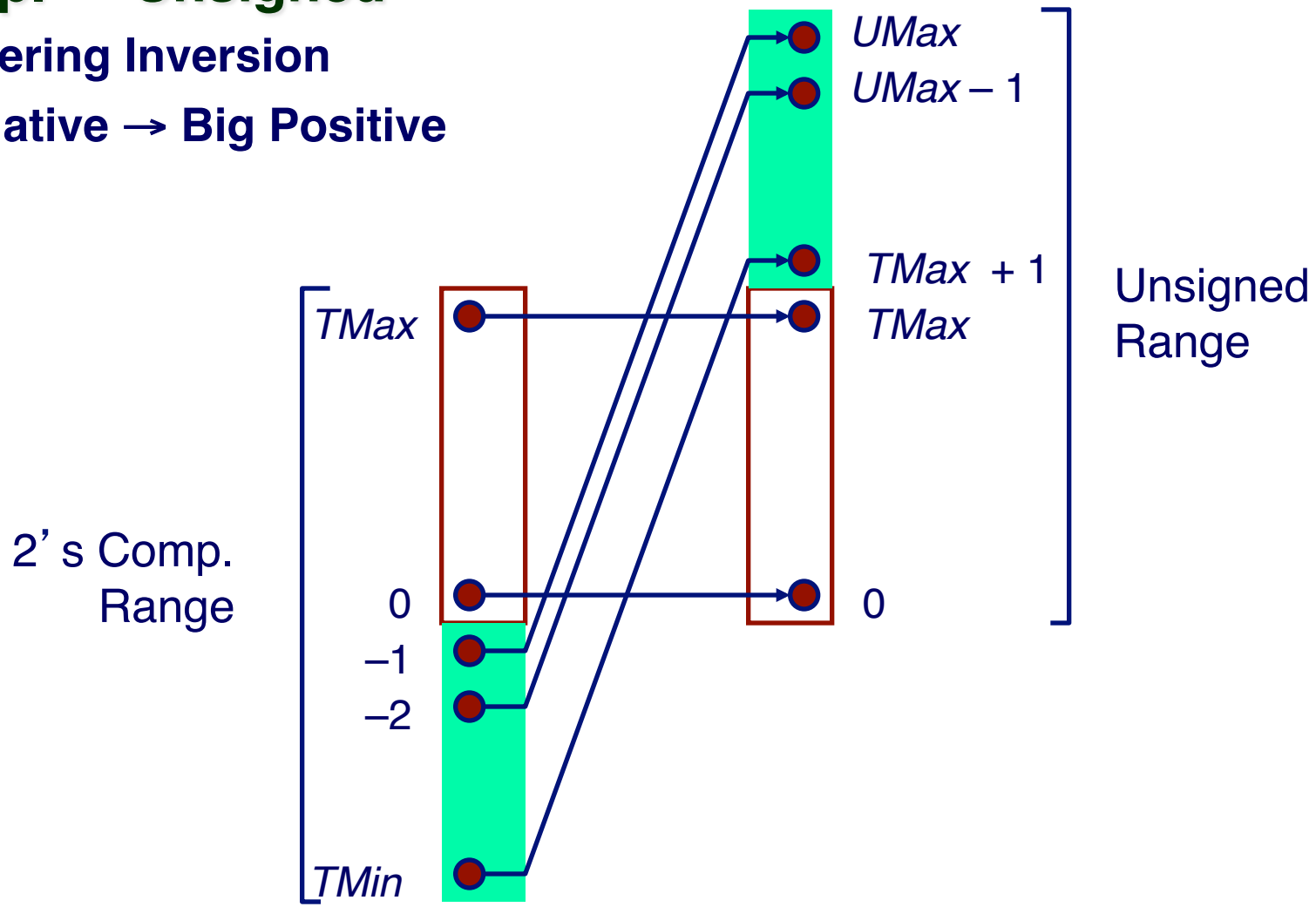
- If mix unsigned and signed in single expression, *signed values implicitly cast to unsigned*
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned) -1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Explanation of Casting Surprises

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



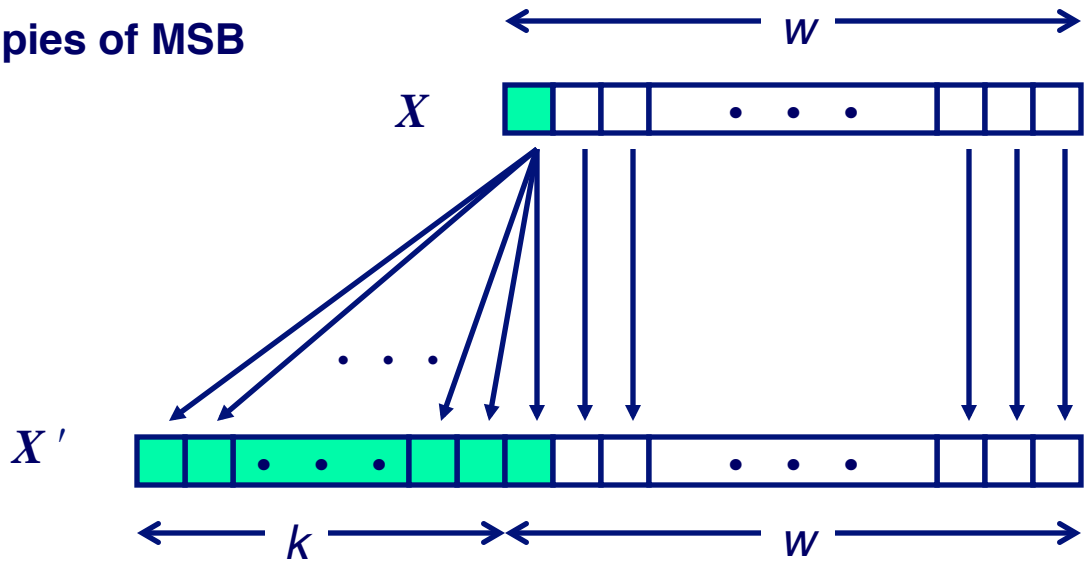
Sign Extension

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;
```

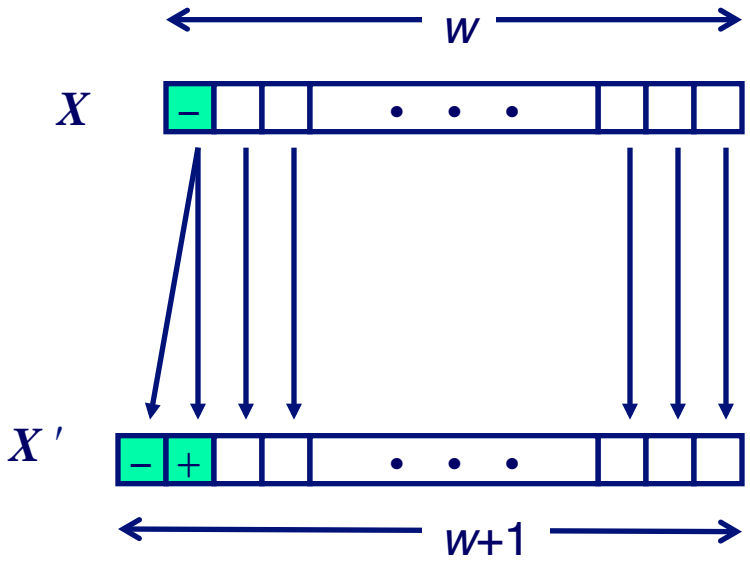
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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

Justification For Sign Extension

Prove Correctness by Induction on *k*

- **Induction Step: extending by single bit maintains value**



- **Key observation:** $-2^{w-1} = -2^{w} + 2^{w-1}$
- **Look at weight of upper bits:**

x $\quad -2^{w-1} x_{w-1}$

x' $\quad -2^{w} x_{w-1} + 2^{w-1} x_{w-1} \quad = \quad -2^{w-1} x_{w-1}$

Why Should I Use Unsigned?

Don't Use Just Because Number Nonzero

- Easy to make mistakes

```
unsigned int i;
for (i = cnt-2; i > -1; i--)
  a[i] += a[i+1];
```

Loop not executed when cnt is a large (+) #

- C compilers on some machines generate less efficient code

```
unsigned int i;
for (i = 1; i < cnt; i++)
  a[i] += a[i-1];
```

Loop executed when cnt<0

Do Use in Limited Cases…

- When representing a set of Boolean flags with bits
- When representing addresses (IP or memory, always >=0)
- When performing modular & multiprecision arithmetic, etc.

Do Use When Need Extra Bit's Worth of Range

- Working right up to limit of word size

Multiplication

Computing Exact Product of *w*-bit numbers *x*, *y*

- Either signed or unsigned

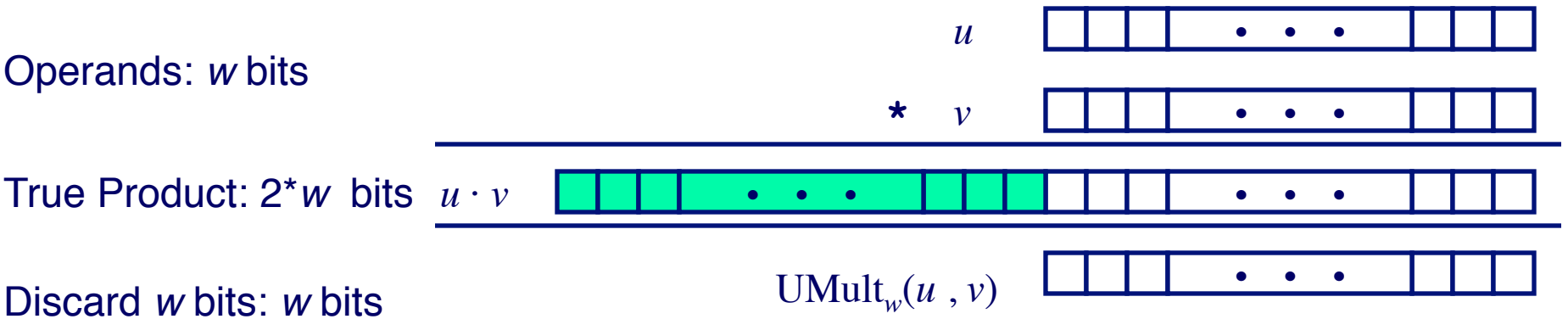
Ranges

- **Unsigned:** $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- **Two's complement min:** $x * y \ge (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- **Two's complement max:** $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2w$ bits, but only for $(TMin_w)^2$

Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C



Standard Multiplication Function

- **Ignores high order *w* bits**

Implements Modular Arithmetic

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

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

- **Truncates product to *w*-bit number** $up = \text{UMult}_w(ux, uy)$
- **Modular arithmetic:** $up = (ux \cdot uy) \mod 2^w$

Two's Complement Multiplication

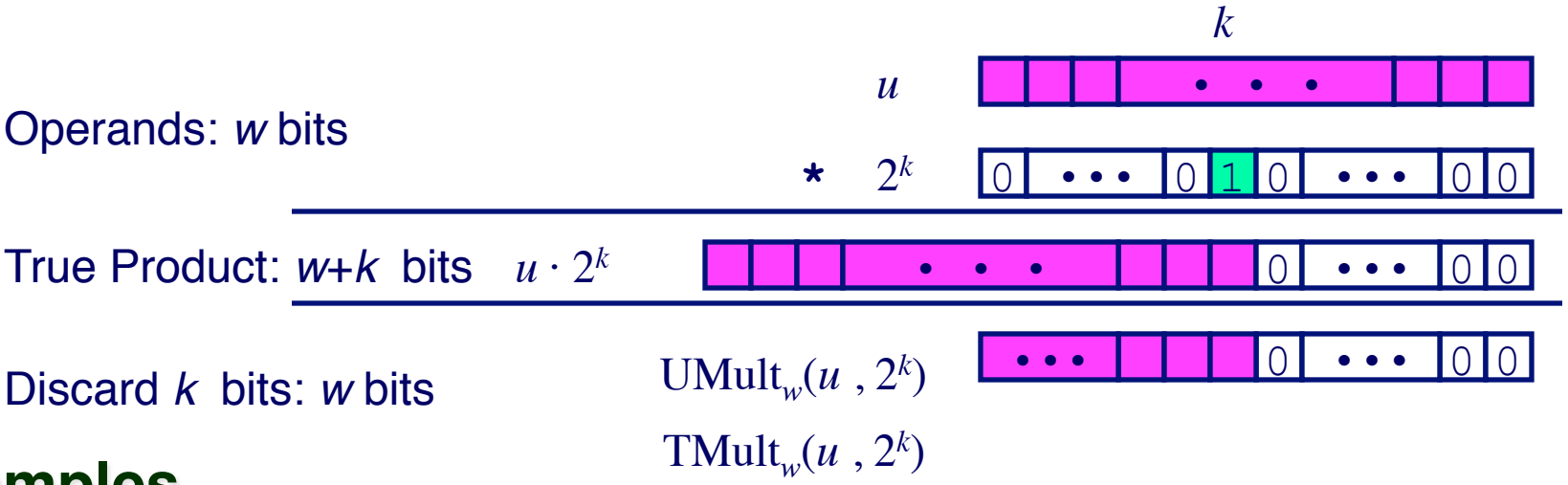
```
int x, y;
int p = x * y;
```

- **Compute exact product of two *w*-bit numbers *x*, *y***
- **Truncate result to *w*-bit number** $p = \text{TMult}_w(x, y)$
- **Signed multiplication gives same bit-level result as unsigned**
- `up == (unsigned) p`

Power-of-2 Multiply with Shift

Operation

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



Examples

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

General Multiplication via Shifts and Adds

- Any integer can be expressed as a sum of powers of 2
  - Example: $37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$
  - Example: $117 = 64 + 32 + 16 + 4 + 1 = 2^6 + 2^5 + 2^4 + 2^2 + 2^0$
- A product of two integers can therefore be expressed as an integer times a sum of powers of 2
  - Example: $6*37 = 6*(2^5 + 2^2 + 2^0)$

$$= 6*2^5 + 6*2^2 + 6*2^0$$

$$= 6 << 5 + 6 << 2 + 6$$

$$= 00000110_2 << 5 + 00000110_2 << 2 + 00000110_2$$

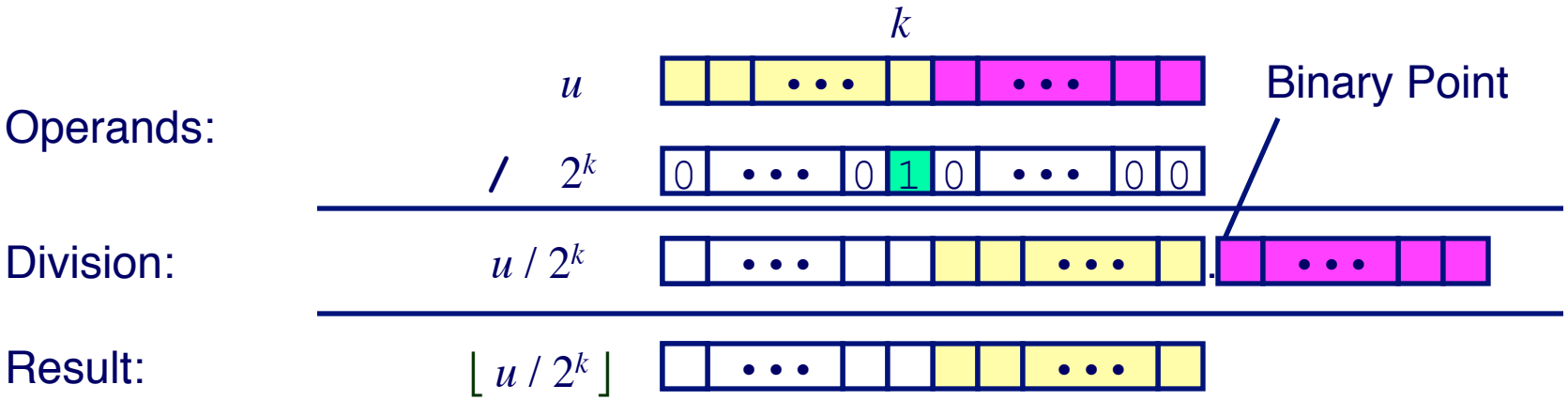
$$= 11000000_2 + 00011000_2 + 00000110_2$$

$$= 11011110_2 = 222_{10}$$

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- `u >> k` gives $\lfloor u / 2^k \rfloor$ ← "floor" function or RoundDown
- Uses logical shift

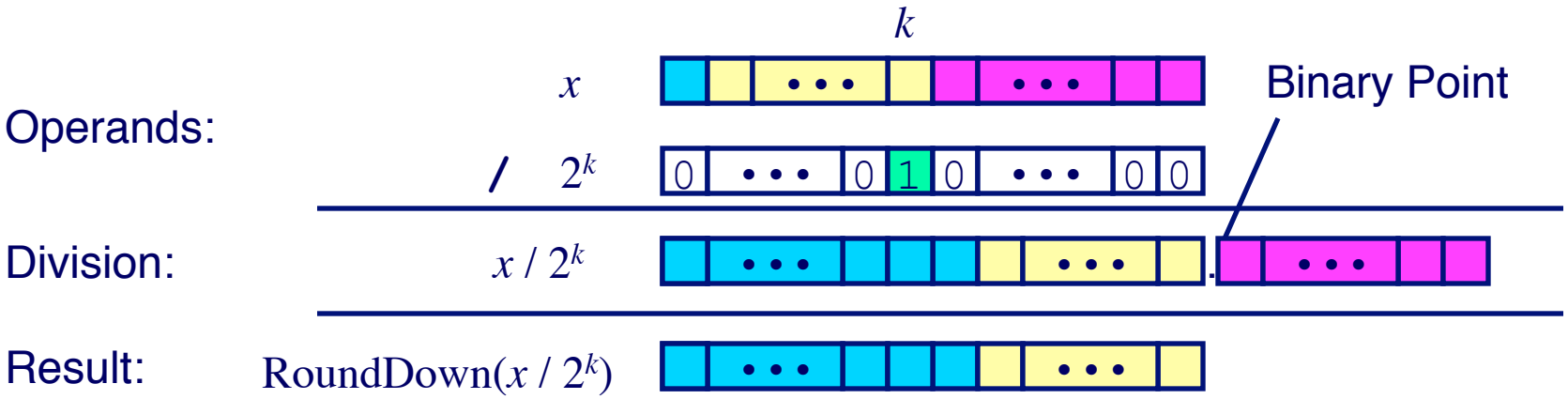


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	**0**0011101 10110110
x >> 4	950.8125	950	03 B6	**0000**0011 10110110
x >> 8	59.4257813	59	00 3B	**00000000** 00111011

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- **`x >> k` gives $\lfloor$ `x / 2`^{k} $\rfloor$**
- **Uses arithmetic shift**
- **Rounds wrong direction when `x < 0`**



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	**1**1100010 01001001
y >> 4	-950.8125	-951	FC 49	**1111**1100 01001001
y >> 8	-59.4257813	-60	FF C4	**11111111** 11000100

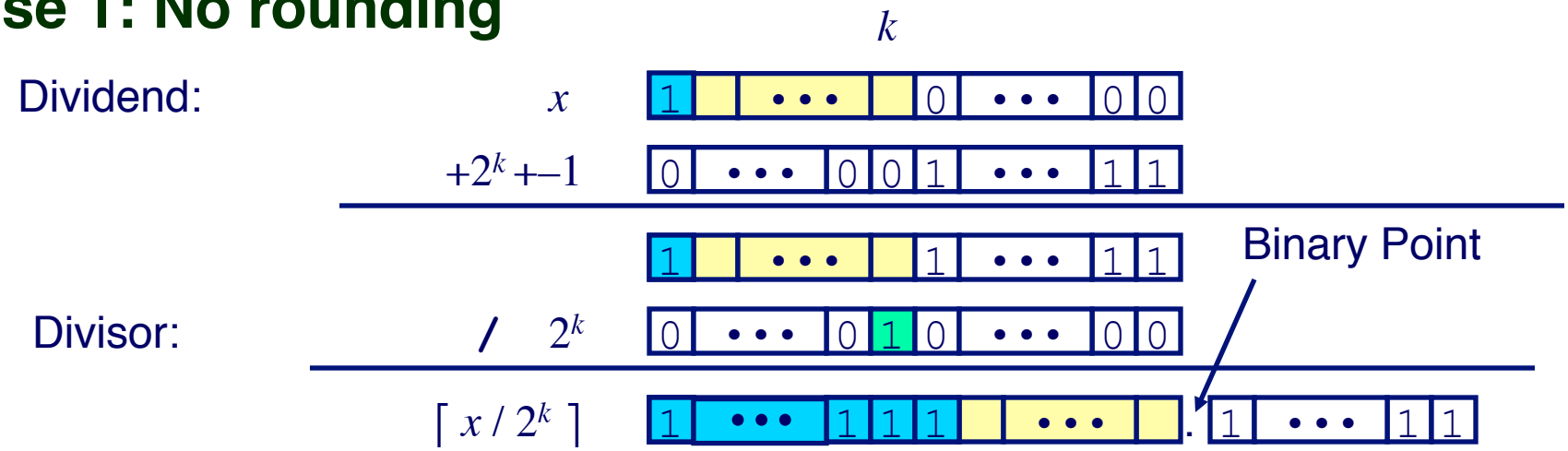
Need to add one!

Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want $\lceil x / 2^k \rceil$ (Round Toward 0) ← "ceiling" function
- Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
  - Adding a pre-bias $2^k - 1$
  - In C: `(x + (1<<k)-1) >> k`
  - Biases dividend toward 0

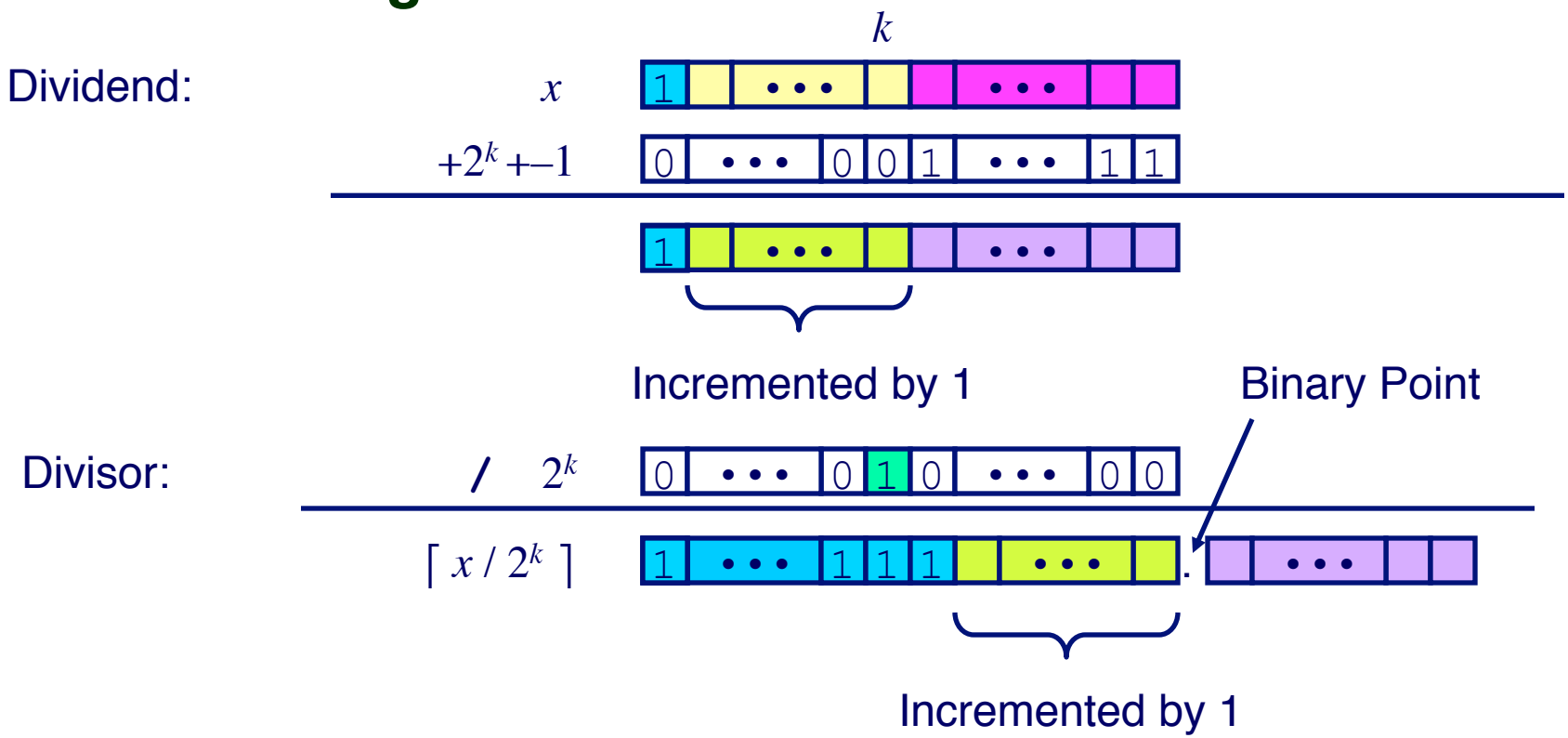
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds the desired 1 to final result

In C, Signed 2^k Division via >>

The default for >> is RoundDown() or the Floor function

- **Either positive or negative**
- **Imprecision for RoundDown for negative #'s**

To get RoundToZero() rounding in signed 2^k Division, you have to implement the following C code:

- `(x<0 ? x+(1<<k)-1 : x) >> k`
- **This is equivalent to:**

```
If (x<0) {
    quotient = (x+2^k – 1) >> k      /* pre-bias by 2^k – 1 before
                                        right-shifting*/
} else {   /* x >= 0 */
    quotient = x >> k
}
```

No Shortcuts to General Integer Division

Example: 117/37

$= 117/(2^5 + 2^2 + 2^0) \neq 117/2^5 + 117/2^2 + 117/2^0$

- Cannot distribute the division, unlike the multiplication
- So we cannot easily use a sum of right shifts as a shortcut for general integer division

Special hardware circuits are designed to implement an integer division

- Access these integer division circuits by invoking the appropriate assembly language instruction, e.g. idivl

Supplementary Slides

Mathematical Properties

Modular Addition Forms an *Abelian Group*

- Closed under addition

  $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$

- Commutative

  $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$

- Associative

  $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$

- 0 is additive identity

  $\text{UAdd}_w(u, 0) = u$

- Every element has additive inverse
  - Let $\text{UComp}_w(u) = 2^w - u$

  $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms Commutative Ring

- Addition is commutative group
- Closed under multiplication

  $0 \leq \text{UMult}_w(u, v) \leq 2^w - 1$
- Multiplication Commutative

  $\text{UMult}_w(u, v) = \text{UMult}_w(v, u)$
- Multiplication is Associative

  $\text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v)$
- 1 is multiplicative identity

  $\text{UMult}_w(u, 1) = u$
- Multiplication distributes over addtion

  $\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$

Properties of Two's Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to *w* bits
- Two's complement multiplication and addition
  - Truncating to *w* bits

Both Form Rings

- Isomorphic to ring of integers mod 2^w

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,

  $u > 0 \quad \Rightarrow \quad u + v > v$

  $u > 0,\ v > 0 \quad \Rightarrow \quad u \cdot v > 0$

- These properties are not obeyed by two's comp. arithmetic

  TMax `+ 1` `==` ***TMin***

  `15213 * 30426 == -10030` (16-bit words)