

CS211: Computer Architecture

Instructor:

Jay Kania

Arithmetic and Bitwise Operations on Binary Data

Some Slides adapted from Bryant & O'Hallaron's slides

Arithmetic and Bitwise Operations

■ Operations

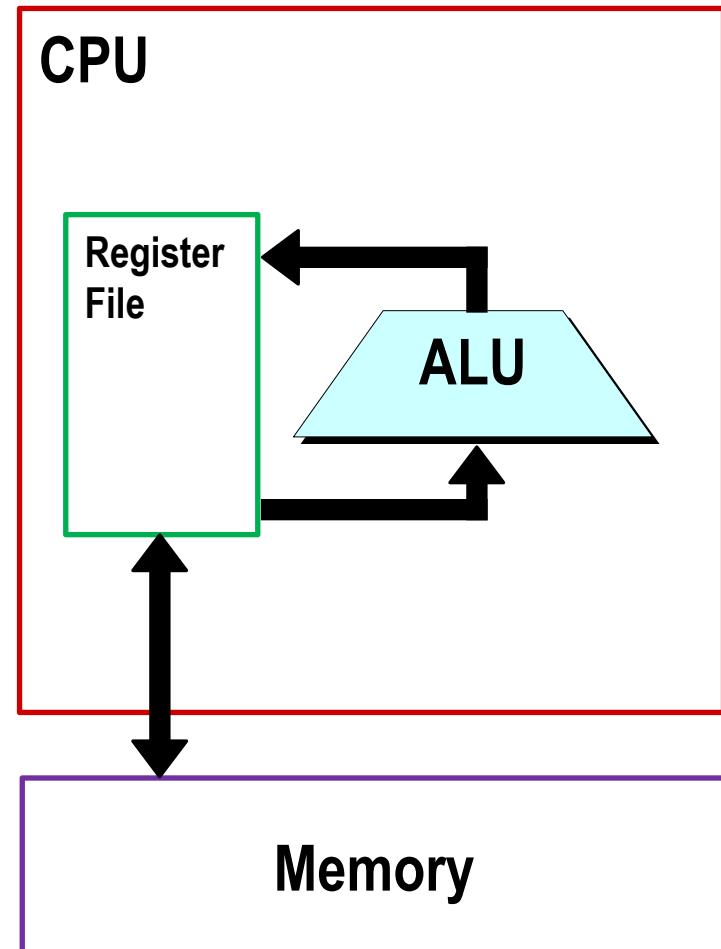
- Bitwise AND, OR, NOT, and XOR
- Logical AND, OR, NOT
- Shifts
- Complements

■ Arithmetic

- Unsigned addition
- Signed addition
- Unsigned/signed multiplication
- Unsigned/signed division

Basic Processor Organization

- **Register file (active data)**
 - We'll be a lot more specific later...
- **Arithmetic Logic Unit (ALU)**
 - Performs signed and unsigned arithmetic
 - Performs logic operations
 - Performs bitwise operations
- **Many other structures...**



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

- Operations applied bitwise
- Bitwise-AND operator: $\&$
- Bitwise- OR operator: $|$
- Bitwise-XOR operator: \wedge
- Bitwise-NOT operator: \sim

01101001	01101001	01101001	01101001
$\& \underline{01010101}$	$ \underline{01010101}$	$\wedge \underline{01010101}$	$\sim \underline{01010101}$
01000001	01111101	00111100	10101010

■ All of the Properties of Boolean Algebra Apply

Quick Check

■ Operate on Bit Vectors

- Operations applied bitwise
- Bitwise-AND operator: $\&$
- Bitwise- OR operator: $|$
- Bitwise-XOR operator: \wedge
- Bitwise-NOT operator: \sim

01100110	11110000	01101001	
$\& \underline{00101111}$	$ \underline{01010101}$	$\wedge \underline{00001111}$	$\sim \underline{00101111}$
00100110	11110101	01100110	11010000

■ All of the Properties of Boolean Algebra Apply

Bit-Level Operations in C

■ Operations $\&$, $|$, \sim , \wedge 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):

in hexadecimal

in binary

- | | | | |
|---|--|-------------------------------|--------------|
| ■ | $\sim 0x41 \rightarrow 0xBE$ | $\sim 01000001_2 \rightarrow$ | 10111110_2 |
| ■ | $\sim 0x00 \rightarrow 0xFF$ | $\sim 00000000_2 \rightarrow$ | 11111111_2 |
| ■ | $0x69 \ \& \ 0x55 \rightarrow 0x41$ | | |
| | $01101001_2 \ \& \ 01010101_2 \rightarrow$ | 01000001_2 | |
| ■ | $0x69 \ \ 0x55 \rightarrow 0x7D$ | | |
| | $01101001_2 \ \ 01010101_2 \rightarrow$ | 01111101_2 | |

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`
- `!!0x41 → 0x01`

- `0x69 && 0x55 → 0x01`
- `0x69 || 0x55 → 0x01`
- `p && *p` // avoids null pointer access

Bitwise Operations: Applications

■ Bit fields

- One byte can fit up to eight options in a single field

- Example: `char flags = 0x1 | 0x4 | 0x8`
 `= 000011012`

- Test for a flag:

```
if ( flags & 0x4 ){  
    //bit 3 is set  
} else {  
    //bit 3 was not set  
}
```

Shift Operations

■ Left Shift: $x \ll y$

- Shift bit-vector x left y places
 - 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 right

■ 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

Quick Check

■ Left Shift: $x \ll y$

- Shift bit-vector x left y places
 - 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 right

■ Undefined Behavior

- Shift amount < 0 or \geq word size

Argument x	00110011
$\ll 3$	
Log. $\gg 4$	
Arith. $\gg 3$	

Argument x	1111111
$\ll 3$	
Log. $\gg 4$	
Arith. $\gg 3$	

Bitwise-NOT: One's Complement

■ Bitwise-NOT operation: \sim

- Bitwise-NOT of x is $\sim x$
- Flip all bits of x to compute $\sim x$
 - flip each 1 to 0
 - flip each 0 to 1

■ Complement

- Given $x == 10011101$

x :

1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---



$\sim x$:

0	1	1	0	0	0	1	0
---	---	---	---	---	---	---	---

- Flip bits (one's complement):

Signed Integer Negation: Two's Complement

■ Negate a number by taking 2's Complement

- Flip bits (one's complement) and add 1

$$\sim x + 1 == -x$$

■ Negation (Two's Complement):

- Given $\mathbf{x} == 10011101$

x:

1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---



- Flip bits (one's complement):

$\sim x$:

0	1	1	0	0	0	1	0
---	---	---	---	---	---	---	---

- Add 1:

$$+ 1$$

-x:

0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Arithmetic and Bitwise Operations

■ Operations

- Bitwise AND, OR, NOT, and XOR
- Logical AND, OR, NOT
- Shifts
- Complements

■ Arithmetic

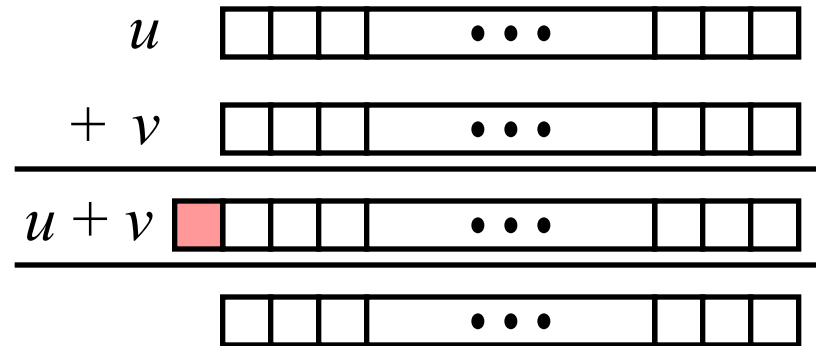
- Unsigned addition
- Signed addition
- Unsigned/signed multiplication
- Unsigned/signed division

Unsigned Addition

Operands: w bits

True Sum: $w+1$ bits

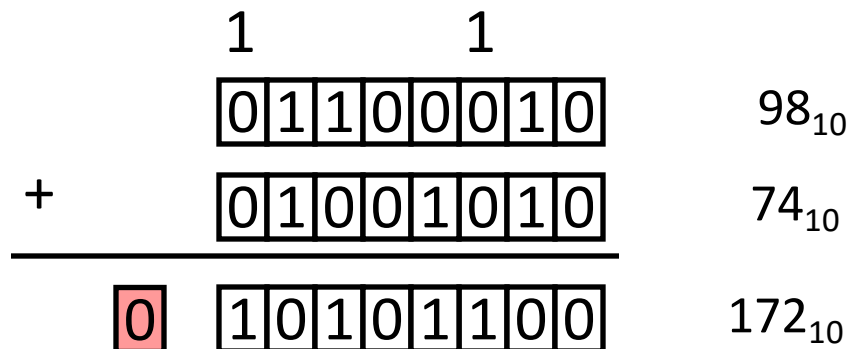
Discard Carry: w bits



■ Addition Operation

- Carry output dropped at end of addition
- Valid ONLY if true sum is within w -bit range

■ Example #1:



Valid in 8-bit
unsigned range

Unsigned Addition

■ Example #2:

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & 1 & & 1 & & 1 & 1 & 1 \\
 & & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
 + & & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 1 & & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0
 \end{array}
 \end{array}$$

110_{10}
 202_{10}
 ~~56_{10}~~

Not Valid in 8-bit
unsigned range
(312 is > 255)

■ Example #3:

$$\begin{array}{r}
 \begin{array}{cccccccccccccccc}
 1 & & 1 & 1 & & 1 & 1 & 1 & & 1 & & & & & 1 \\
 & & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
 + & & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 1 & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}
 \end{array}$$

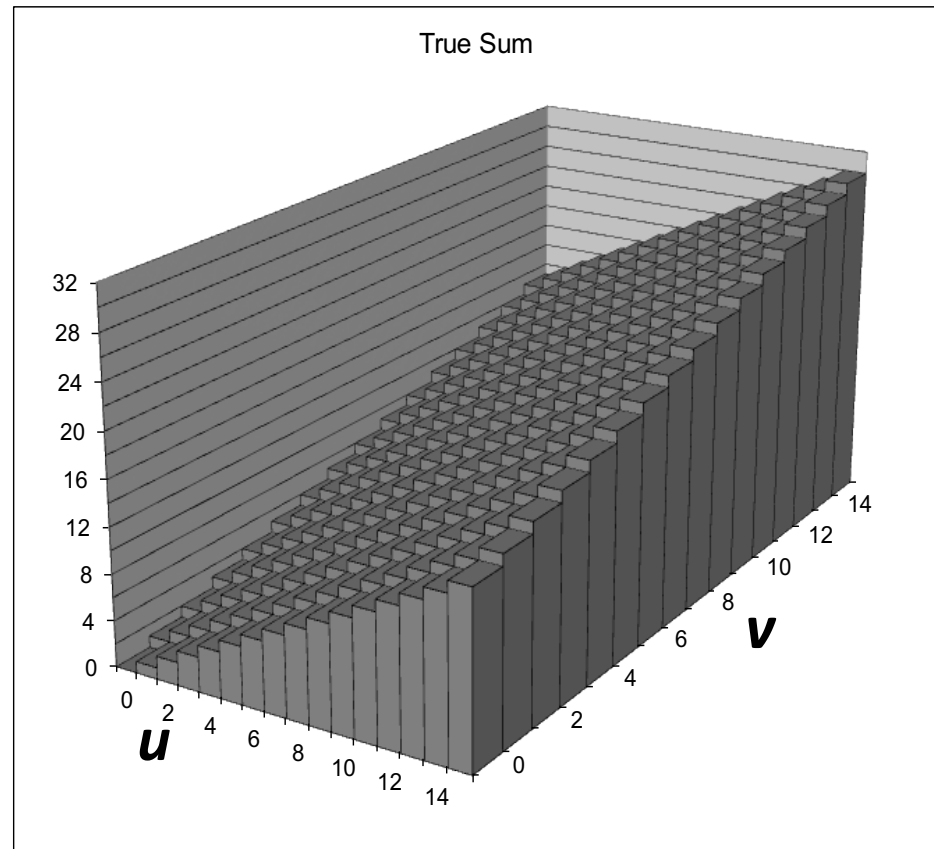
10082_{10}
 59978_{10}
 ~~4524_{10}~~

Not Valid in 16-bit
unsigned range
(70060 is > 65535)

Visualizing True Sum (Mathematical) Addition

■ Integer Addition

- 4-bit integers u, v
- Compute true sum
- Values increase linearly with u and v
- Forms planar surface

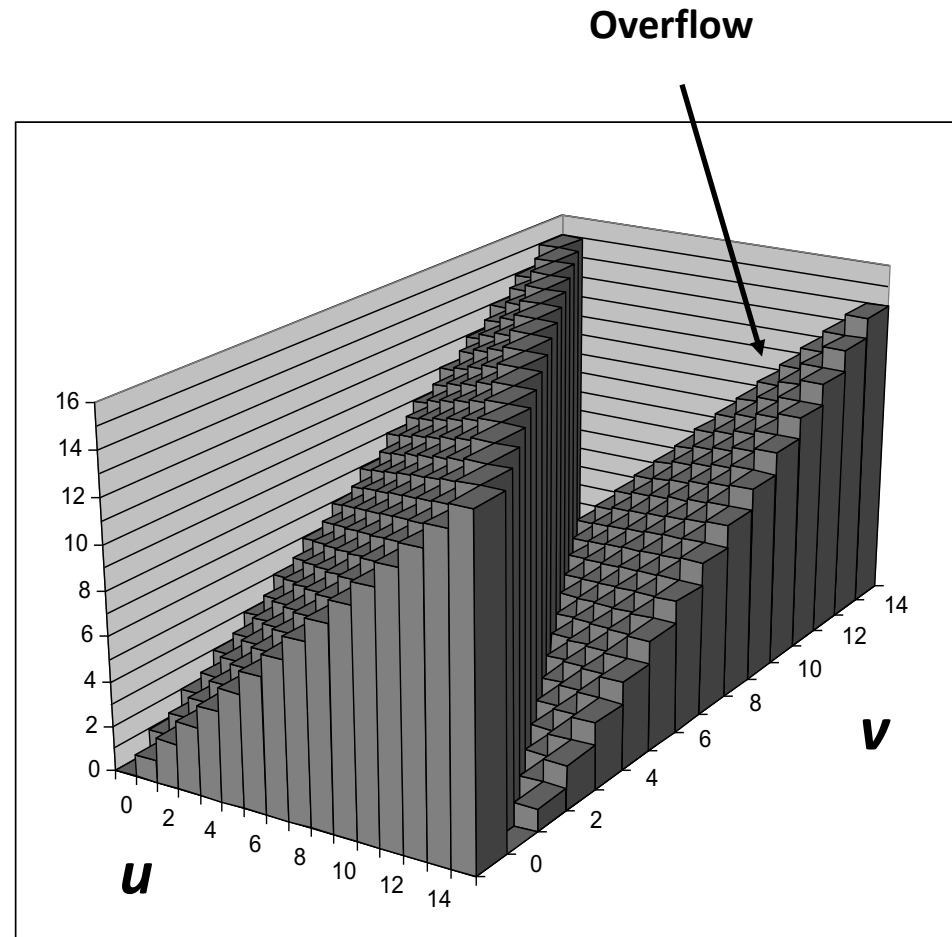
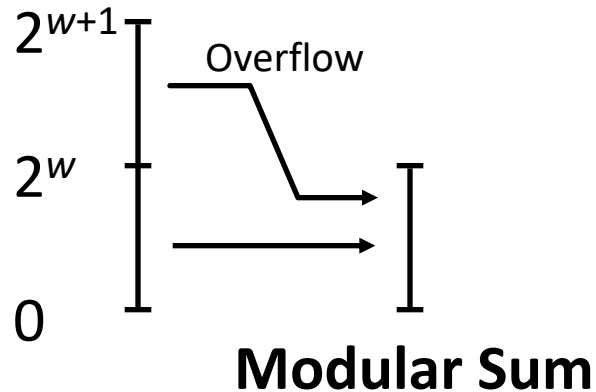


Visualizing Unsigned Addition

■ Wraps Around

- If true sum $\geq 2^w$
- At most once

True Sum



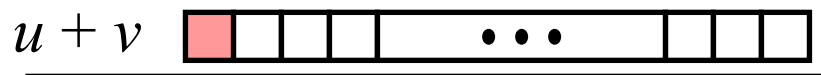
$$\text{Sum} = \text{UAdd}_w(u,v) = \text{true}(u+v) \% 2^w$$

Two's Complement Addition

Operands: w bits



True Sum: $w+1$ bits



Discard Carry: w bits



■ Signed/Unsigned adds 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`

■ The only difference is our interpretation of sign bit value

Signed Addition

*Note: Same bytes as for Ex #1 and Ex #2
in unsigned integer addition, but
now interpreted as 8-bit signed integers*

■ Example #1:

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & 1 & & & 1 & & \\
 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
 + & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}
 \end{array}$$

98_{10}
 74_{10}
 ~~-84_{10}~~

Not Valid in 8-bit
signed range
($172 > 127$)

■ Example #2:

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & 1 & & 1 & & 1 & 1 & 1 \\
 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
 + & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0
 \end{array}
 \end{array}$$

110_{10}
 -54_{10}
 56_{10}

Valid in 8-bit
signed range
($-128 < 56 < 127$)

Signed Addition

Note: Same bytes as for Ex #1 and Ex #2 in unsigned integer addition, but now interpreted as 8-bit signed integers

■ Example #3:

$$\begin{array}{r}
 \begin{array}{c} 1 \quad 1 \\ 11100010 \end{array} \\
 + \begin{array}{c} 11011000 \end{array} \\
 \hline
 \begin{array}{c} \boxed{1} \quad 10111010 \end{array}
 \end{array}
 \quad
 \begin{array}{l}
 -30_{10} \\
 -40_{10} \\
 -70_{10}
 \end{array}$$

Valid in 8-bit signed range
 $(-128 < -74)$

■ Example #2:

$$\begin{array}{r}
 \begin{array}{c} 1 \quad 1 \quad 1 \quad 1 \\ 10011100 \end{array} \\
 + \begin{array}{c} 11001110 \end{array} \\
 \hline
 \begin{array}{c} \boxed{1} \quad 01101010 \end{array}
 \end{array}
 \quad
 \begin{array}{l}
 -100_{10} \\
 -50_{10} \\
 \cancel{106_{10}}
 \end{array}$$

Not Valid in 8-bit signed range
 $(-150 < -128)$

Visualizing Signed Addition

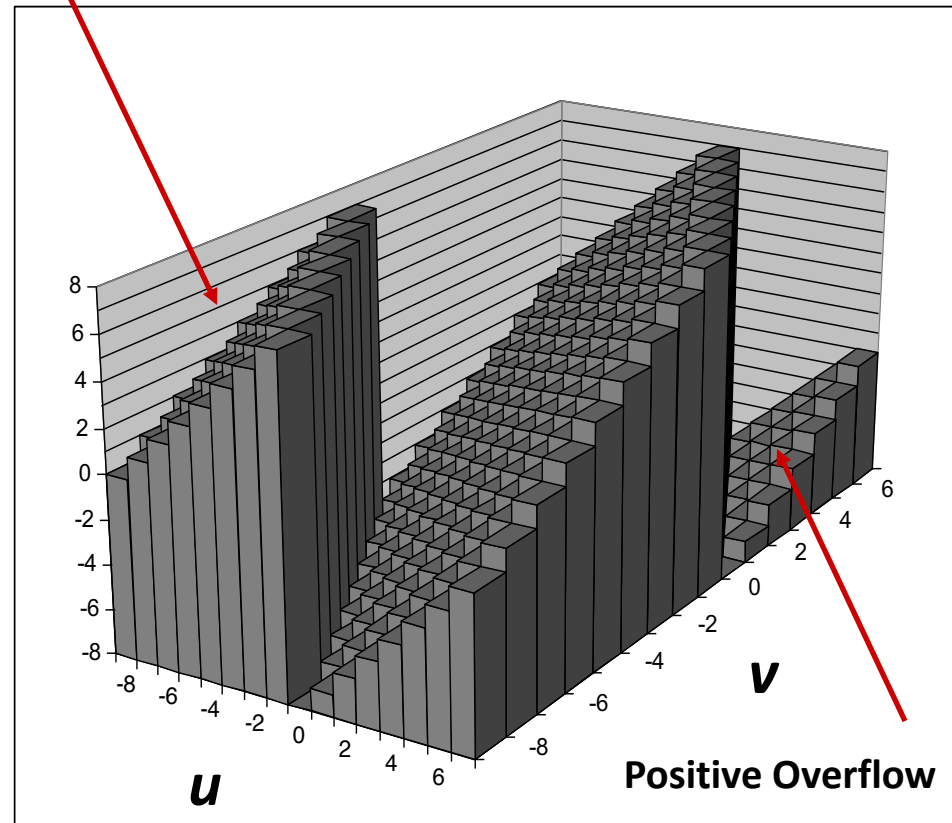
■ Values

- 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

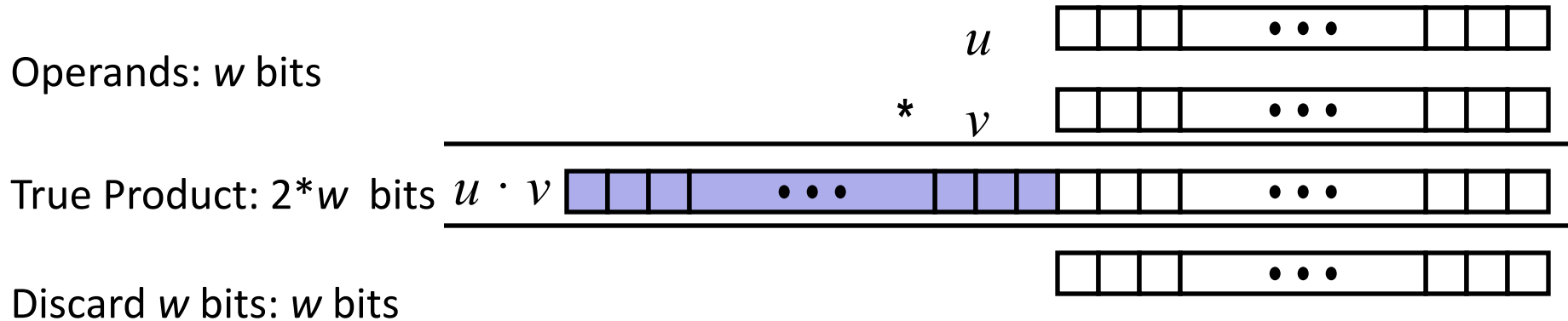
Negative Overflow



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 $(SMin_w)^2$
 - 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



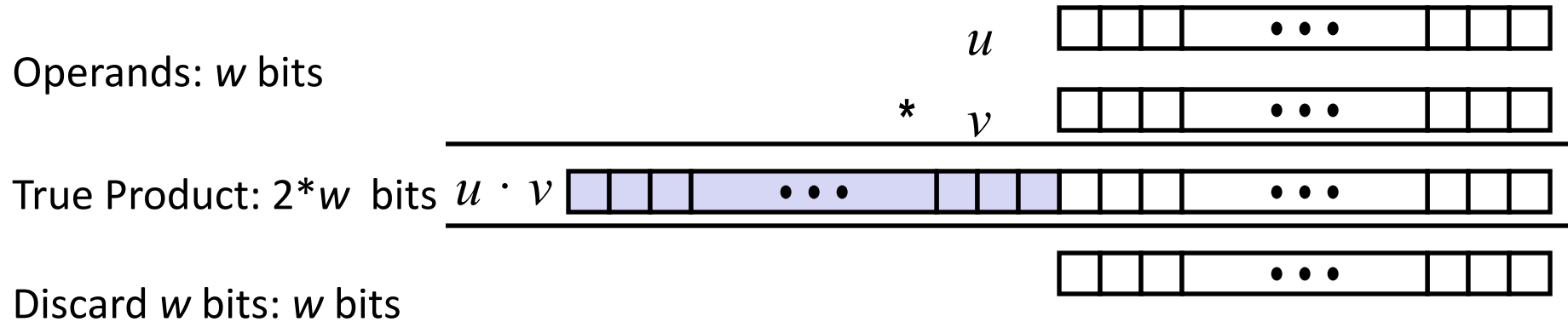
■ Standard Multiplication Function

- Ignores high order w bits

■ Implements Modular Arithmetic

$$UMult_w(u \cdot v) = true(u \cdot v) \bmod 2^w$$

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

■ The only difference is our interpretation of sign bit value

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r} 123_{10} \\ \times 234_{10} \\ \hline 492 \\ 3690 \\ + 24600 \\ \hline 28782_{10} \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r}
 123_{10} \\
 \times 234_{10} \\
 \hline
 492 \\
 3690 \\
 + 24600 \\
 \hline
 28782_{10}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} & 123_{10} \\
 \times \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} & 234_{10} \\
 \hline
 \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r}
 123_{10} \\
 \times 234_{10} \\
 \hline
 492 \\
 3690 \\
 + 24600 \\
 \hline
 28782_{10}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} & 123_{10} \\
 \times \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} & 234_{10} \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r}
 123_{10} \\
 \times 234_{10} \\
 \hline
 492 \\
 3690 \\
 + 24600 \\
 \hline
 28782_{10}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} & 123_{10} \\
 \times \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} & 234_{10} \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1
 \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r}
 123_{10} \\
 \times 234_{10} \\
 \hline
 492 \\
 3690 \\
 + 24600 \\
 \hline
 28782_{10}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} & 123_{10} \\
 \times \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} & 234_{10} \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$$\begin{array}{r}
 123_{10} \\
 \times 234_{10} \\
 \hline
 492 \\
 3690 \\
 + 24600 \\
 \hline
 28782_{10}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline \end{array} & 123_{10} \\
 \times \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array} & 234_{10} \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 &
 \end{array}$$

True Binary Multiplication

- Multiply positive integers using the same place-value algorithm you learned in grade school

$ \begin{array}{r} 123_{10} \\ \times 234_{10} \\ \hline 492 \\ 3690 \\ + 24600 \\ \hline 28782_{10} \end{array} $	$ \begin{array}{r} \boxed{01111011} \quad 123_{10} \\ \times \boxed{11101010} \quad 234_{10} \\ \hline \begin{array}{r} 00000000 \\ 01111011 \\ 00000000 \\ 01111011 \\ 00000000 \\ 01111011 \\ 01111011 \\ + 01111011 \\ \hline \boxed{0111000001101110} \quad 28782_{10} \end{array} \end{array} $
---	--

Power-of-2 Multiply with Shift

■ Consider: $6_{10} * 2_{10} = 12_{10}$

$$\begin{array}{r} \boxed{0}\boxed{1}\boxed{1}\boxed{0} \quad 6_{10} \\ \times \boxed{0}\boxed{0}\boxed{1}\boxed{0} \quad 2_{10} \\ \hline \end{array}$$

Power-of-2 Multiply with Shift

■ Consider: $6_{10} * 2_{10} = 12_{10}$

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline \end{array} & 6_{10} \\
 \times & \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 0 \\ \hline \end{array} & 2_{10} \\
 \hline
 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 0 \\
 & 0 & 0 & 0 & 0 \\
 + & 0 & 0 & 0 & 0 \\
 \hline
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \end{array} & 12_{10}
 \end{array}$$

Power-of-2 Multiply with Shift

- Consider: $6_{10} * 2_{10} = 12_{10}$

A handwritten multiplication of 6 (0110) by 2 (0010) in binary. The first number is 0110 (6₁₀) and the second is 0010 (2₁₀). The multiplication is shown with a horizontal line. Below the line, the partial products are listed: 0000, 0110, 0000, and + 0000. Red arrows point from the first two bits of the first number (01) to the first two bits of the result (11), and from the last two bits of the first number (10) to the last two bits of the result (00). The final result is 1100 (12₁₀).

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline \end{array} & 6_{10} \\
 \times & \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 0 \\ \hline \end{array} & 2_{10} \\
 \hline
 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 0 \\
 & 0 & 0 & 0 & 0 \\
 + & 0 & 0 & 0 & 0 \\
 \hline
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \end{array} & 12_{10}
 \end{array}$$

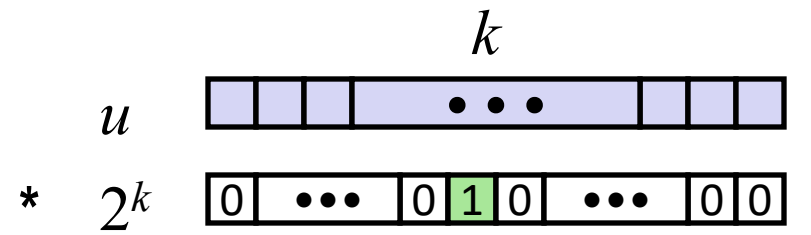
- Multiplying by two always shifts the input bit pattern by one to the left. That is: $(6_{10} * 2_{10}) == (0110_2 \ll 1)$
- More generally- multiplying by 2^k always shifts the input by k to the left: $(x_{10} * 2^k) == (x_2 \ll k)$

Power-of-2 Multiply with Shift

■ Operation

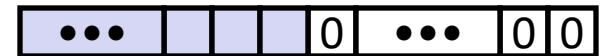
- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned

Operands: w bits



True Product: $w+k$ bits $u \cdot 2^k$

Discard k bits: w bits



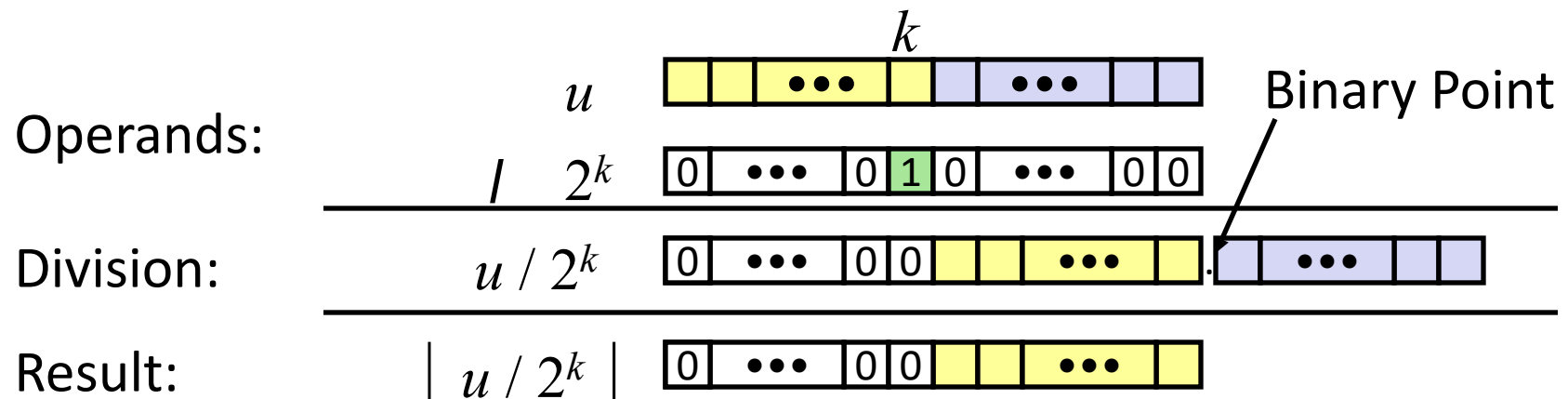
■ Examples

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

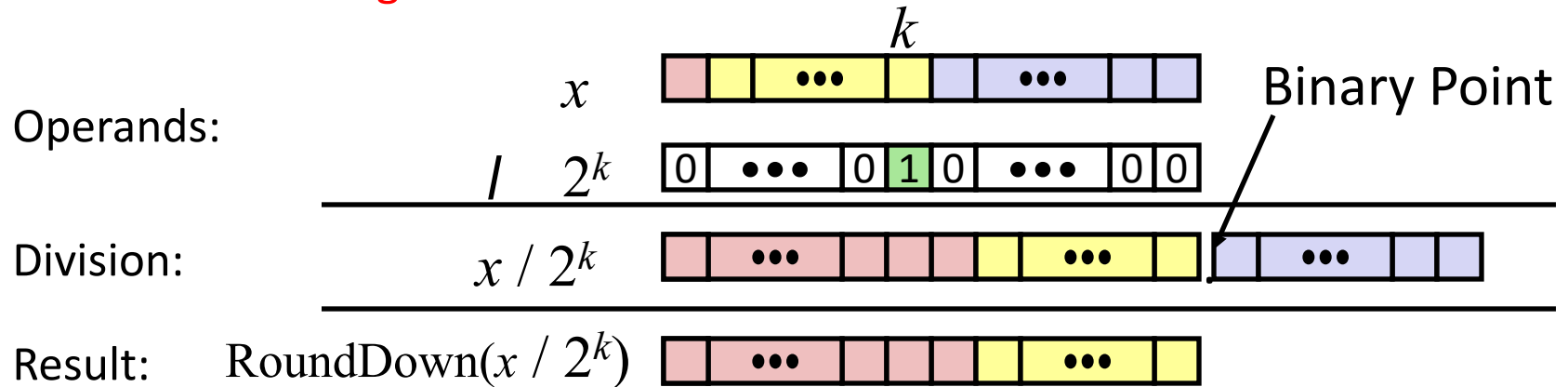


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Signed Power-of-2 Divide with Shift

■ Quotient of Signed by Power of 2

- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift (why not logical?)
- Rounds wrong direction when $u < 0$



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	11100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	11111100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	11111111 11000100

Incorrect Power-of-2 Divide

- Consider: $-25 / 2$
- We expect that $-25 / 2 = -12$, however:

1. $-25_{10} = 11100111_2$
2. $(-25 / 2)$ becomes $(11100111_2 \ggg 1)$
3. $(11100111_2 \ggg 1) = 11110011_2$
4. $11110011_2 = -13$

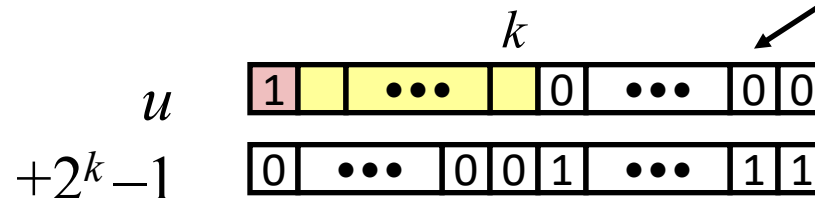
Correct Power-of-2 Divide with *Biasing*

■ Quotient of Negative Number by Power of 2

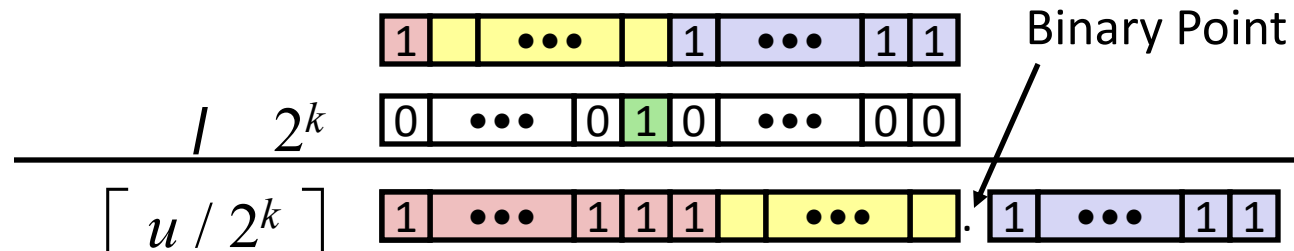
- Want $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lfloor (x + 2^k - 1) / 2^k \rfloor$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

Case 1: No rounding

Dividend:



Divisor:



Biasing has no effect

Biasing without changing result

■ Consider: $-20 / 4$ (answer should be -5)

Without bias:

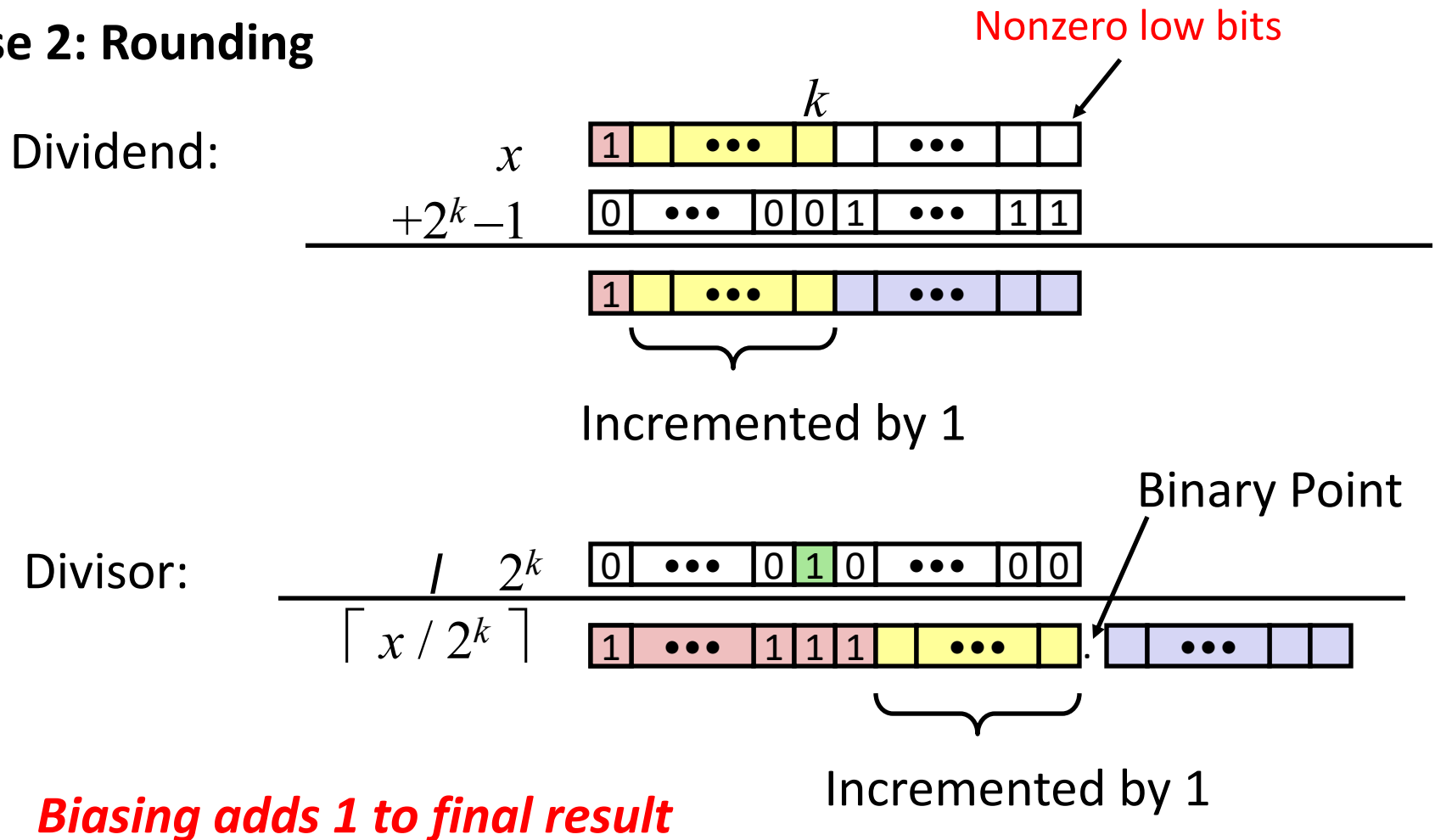
1. $-20_{10} = 11101100_2$
2. $(-20 / 4)$ becomes $(11101100_2 \gg 2)$
3. $(11101100_2 \gg 2) = 11111011_2$
4. $11111011_2 = -5$

With bias: $2^k = 4$, bias: $4 - 1 = 3$

1. $-20_{10} + 3_{10} = 111011\textcolor{red}{11}_2$
2. $(-23 / 4)$ becomes $(111011\textcolor{red}{11}_2 \gg 2)$
3. $(111011\textcolor{red}{11}_2 \gg 2) = 11111011_2$
4. $11111011_2 = -5$

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing that does change the result

- Consider: $-21 / 4$ (answer should be -5)

Without bias:

1. $-21_{10} = 11101011_2$
2. $(-21 / 4)$ becomes $(11101011_2 \gg 2)$
3. $(11101011_2 \gg 2) = 11111010_2$
4. $11111010_2 = -6$ (incorrect!)

With bias: $2^k = 4$, bias: $4 - 1 = 3$

1. $-21_{10} + 3_{10} = 11101110_2$
2. $(-18 / 4)$ becomes $(11101110_2 \gg 2)$
3. $(11101110_2 \gg 2) = 11111011_2$
4. $11111011_2 = -5$

Biasing that does change the result

- Consider: $-21 / 4$ (answer should be -5)

Without bias:

1. $-21_{10} = 11101011_2$
2. $(-21 / 4)$ becomes $(11101011_2 \gg 2)$
3. $(11101011_2 \gg 2) = 11111010_2$
4. $11111010_2 = -6$ (incorrect!)

Recall- lowest
order bit has
value 1!

With bias: $2^k = 4$, bias: $4 - 1 = 3$

1. $-21_{10} + 3_{10} = 11101110_2$
2. $(-18 / 4)$ becomes $(11101110_2 \gg 2)$
3. $(11101110_2 \gg 2) = 11111011_2$
4. $11111011_2 = -5$



Arithmetic: Basic Rules

- **Unsigned ints, 2's complement ints are isomorphic rings:
isomorphism = casting**

- **Left shift**
 - Unsigned/signed: multiplication by 2^k
 - Always logical shift

- **Right shift**
 - Unsigned: logical shift, div (division + round to zero) by 2^k
 - Signed: arithmetic shift
 - Positive numbers: div (division + round to zero) by 2^k
 - Negative numbers: div (division + round away from zero) by 2^k
Use biasing to fix