Arithmetic and Bitwise Operations on Binary Data

CSCI 2400: Computer Architecture
ECE 3217: Computer Architecture and Organization

Instructor:

David Ferry

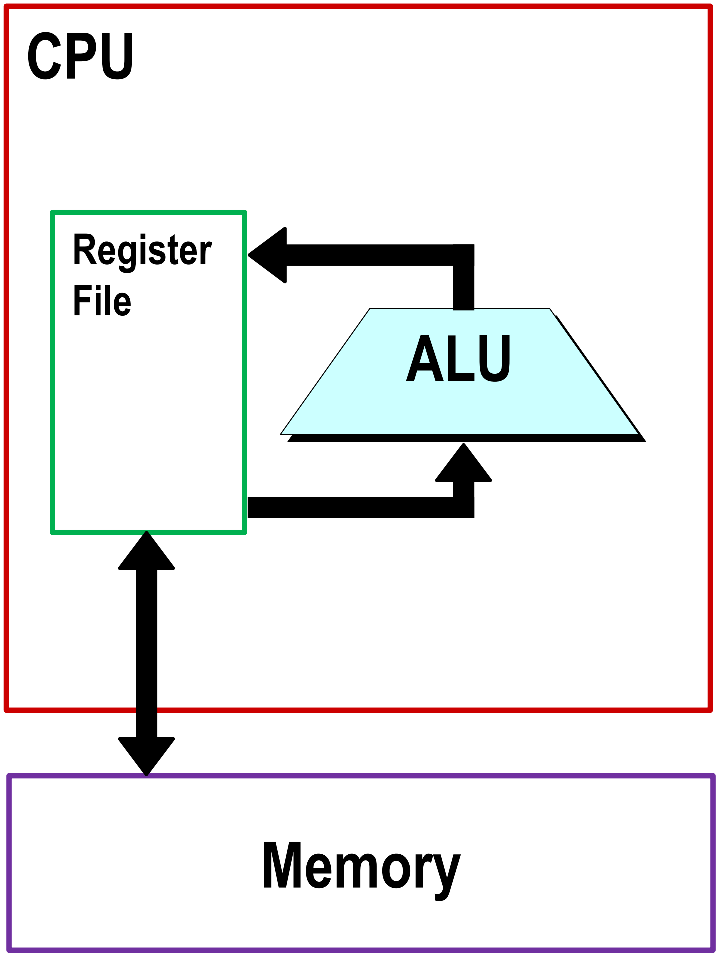
Slides adapted from Bryant & O'Hallaron's slides by Jason Fritts

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

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

~	
0	1
1	0

Exclusive-Or (Xor)

- **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
  - Bitwise-AND operator: &
  - Bitwise-NOR operator: |
  - Bitwise-XOR operator: ^
  - Bitwise-NOT operator: ~

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 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-NOR operator: |
  - Bitwise-XOR operator: ^
  - Bitwise-NOT operator: ~

```
  01100110    11110000    01101001
& 00101111  | 01010101  ^ 00001111  ~ 00101111
  00100110    11110101    01100110    11010000
```

- **All of the Properties of Boolean Algebra Apply**

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):**

in hexadecimal ***in binary***

- ~0x41 → 0xBE // ~01000001_2 → 10111110_2
- ~0x00 → 0xFF // ~00000000_2 → 11111111_2
- 0x69 & 0x55 → 0x41 // 01101001_2 & 01010101_2 → 01000001_2
- 0x69 | 0x55 → 0x7D // 01101001_2 | 01010101_2 → 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`

    $= 00001101_2$
  - Test for a flag:

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

Shift Operations

- **Left Shift:** `x << y`
  - Shift bit-vector x left y places
      - 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 right
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word 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`

Quick Check

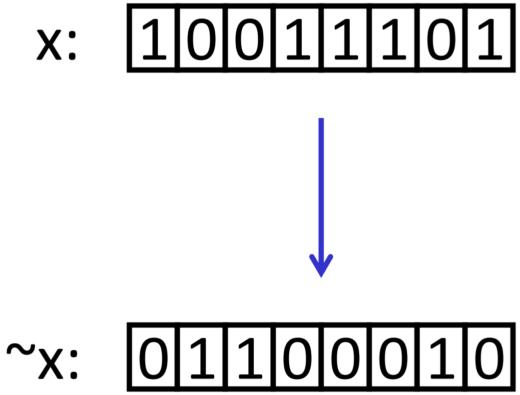
- **Left Shift:** `x << y`
  - Shift bit-vector x left y places
    - – Throw away extra bits on left
    - Fill with 0’s on right
- **Right Shift:** `x >> y`
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on right
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

Argument `x`	`00110011`
`<< 3`	
Log. `>> 4`	
Arith. `>> 3`	

Argument `x`	`1111111`
`<< 3`	
Log. `>> 4`	
Arith. `>> 3`	

Bitwise-NOT: One’s Complement

- **Bitwise-NOT operation: `~`**
  - Bitwise-NOT of **`x`** is **`~x`**
  - Flip all bits of **`x`** to compute **`~x`**
    - flip each 1 to 0
    - flip each 0 to 1
- **Complement**
  - Given **`x`** `==` `10011101`
  - Flip bits (one’s complement):



x: 1 0 0 1 1 1 0 1

~x: 0 1 1 0 0 0 1 0

Signed Integer Negation: Two's Complement

- **Negate a number by taking 2's Complement**
  - Flip bits (one's complement) and add 1

    `~x + 1 == -x`
- **Negation (Two's Complement):**
  - Given `x == 10011101`

x: 1 0 0 1 1 1 0 1

  - Flip bits (one's complement):

~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

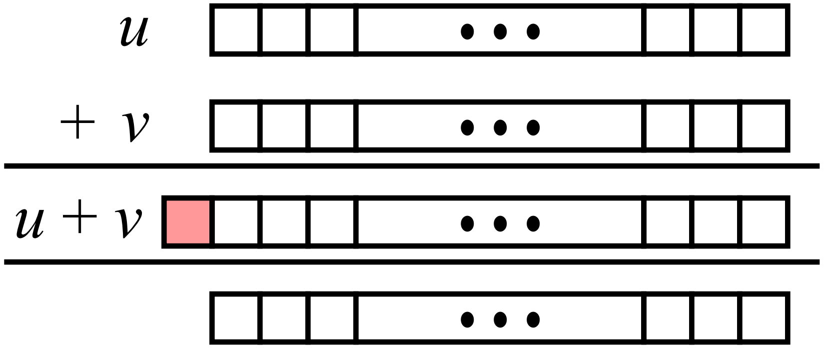
- **Operations**
  - Bitwise AND, OR, NOT, and XOR
  - Logical AND, OR, NOT
  - Shifts
  - Complements
- **Arithmetic**
  - Unsigned addition
  - Signed addition
  - Unsigned/signed multiplication
  - Unsigned/signed division

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:**

		1					1			
		0	1	1	0	0	0	1	0	98_{10}
+		0	1	0	0	1	0	1	0	74_{10}
	0	1	0	1	0	1	1	0	0	172_{10}

***Valid** in 8-bit unsigned range*

Unsigned Addition

- **Example #2:**

	1	1			1	1	1			
		0	1	1	0	1	1	1	0	110_{10}
+		1	1	0	0	1	0	1	0	202_{10}
	1	0	0	1	1	1	0	0	0	$\cancel{56_{10}}$

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

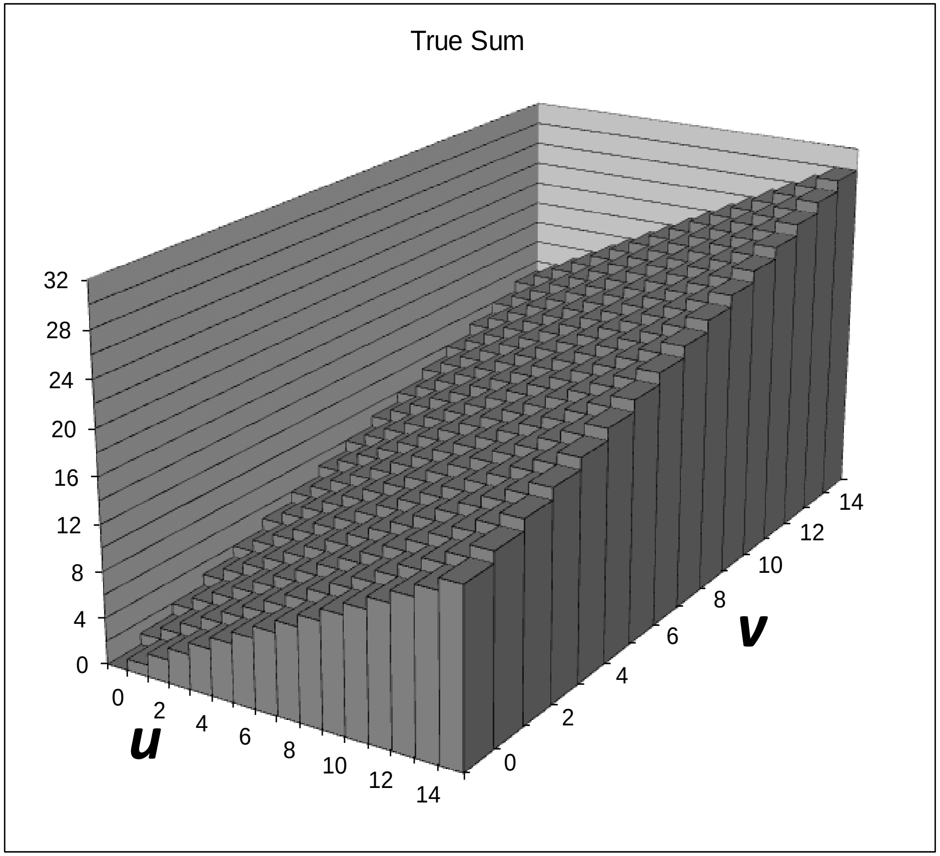
- **Example #3:**

	1	1	1		1	1	1			1					1			
		0	0	1	0	0	1	1	1	0	1	1	0	0	0	1	0	10082_{10}
+		1	1	1	0	1	0	1	0	0	1	0	0	1	0	1	0	59978_{10}
	1	0	0	0	1	0	0	0	1	1	0	1	0	1	1	0	0	$\cancel{4524_{10}}$

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

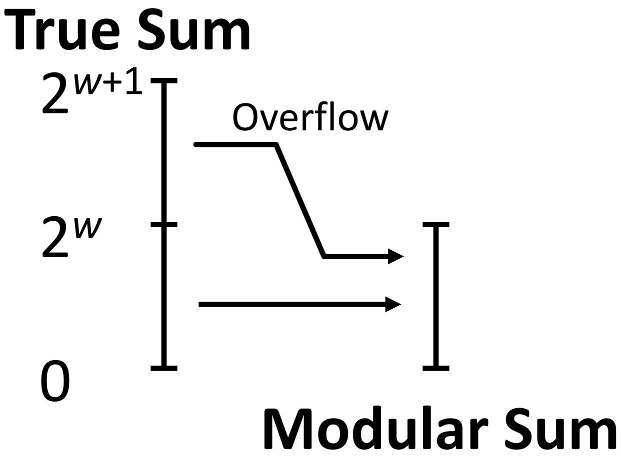
Visualizing True Sum (Mathematical) Addition

- **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 ≥ 2^w
  - At most once



True Sum

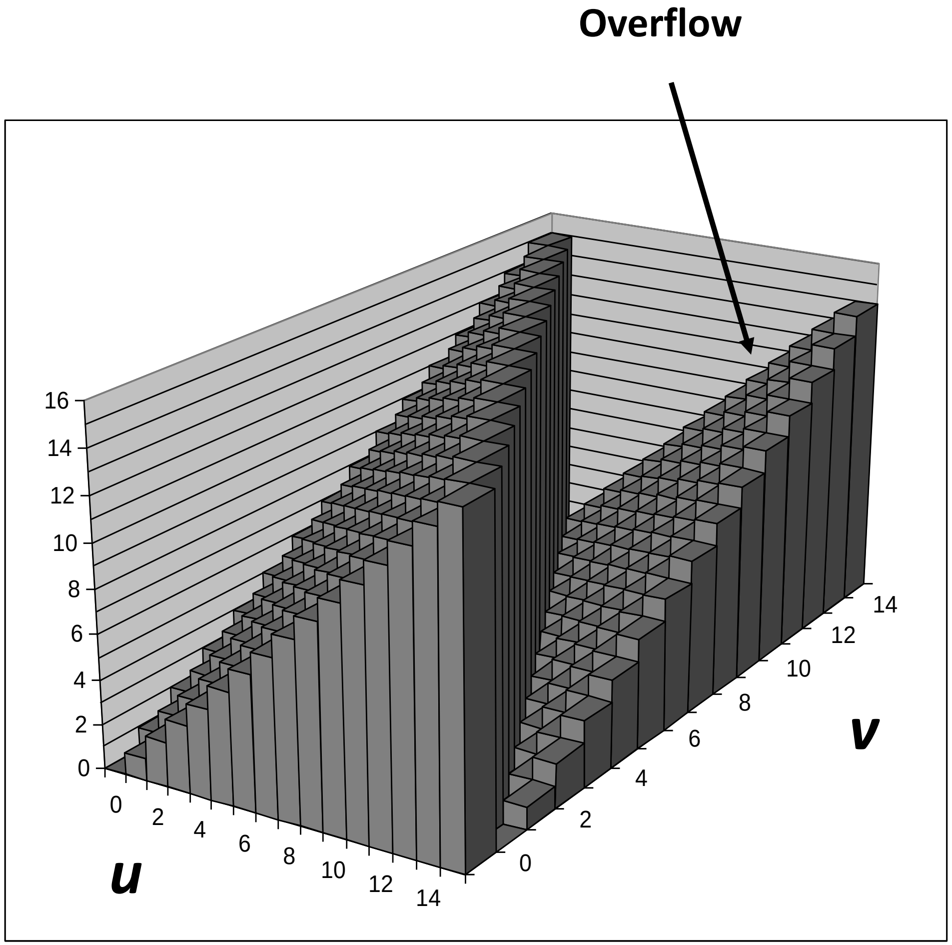
2^{w+1}

Overflow

2^w

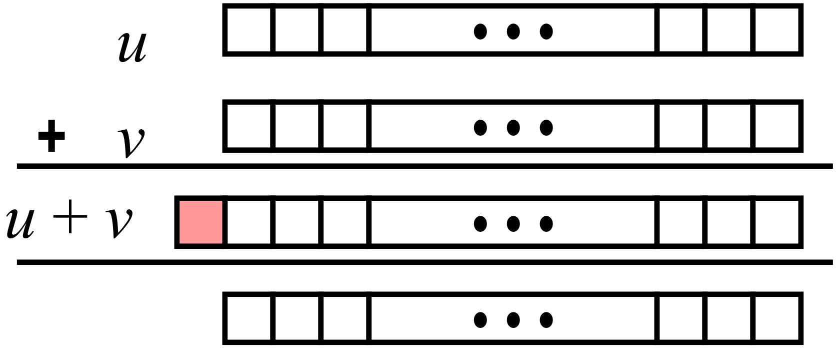
0

Modular Sum



Overflow

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`

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:**

$$\begin{array}{cc|cccccccc|c}
 & & 1 & & & & & 1 & & & \\
 & & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 98_{10} \\
+ & & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 74_{10} \\
\hline
 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & \cancel{-84_{10}}
\end{array}$$

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

- **Example #2:**

$$\begin{array}{cc|cccccccc|c}
 & 1 & 1 & & & 1 & 1 & 1 & & & \\
 & & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 110_{10} \\
+ & & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & -54_{10} \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 56_{10}
\end{array}$$

***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:**

	1	1								
		1	1	1	0	0	0	1	0	-30_{10}
+		1	1	0	1	1	0	0	0	-40_{10}
	1	1	0	1	1	1	0	1	0	-70_{10}

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

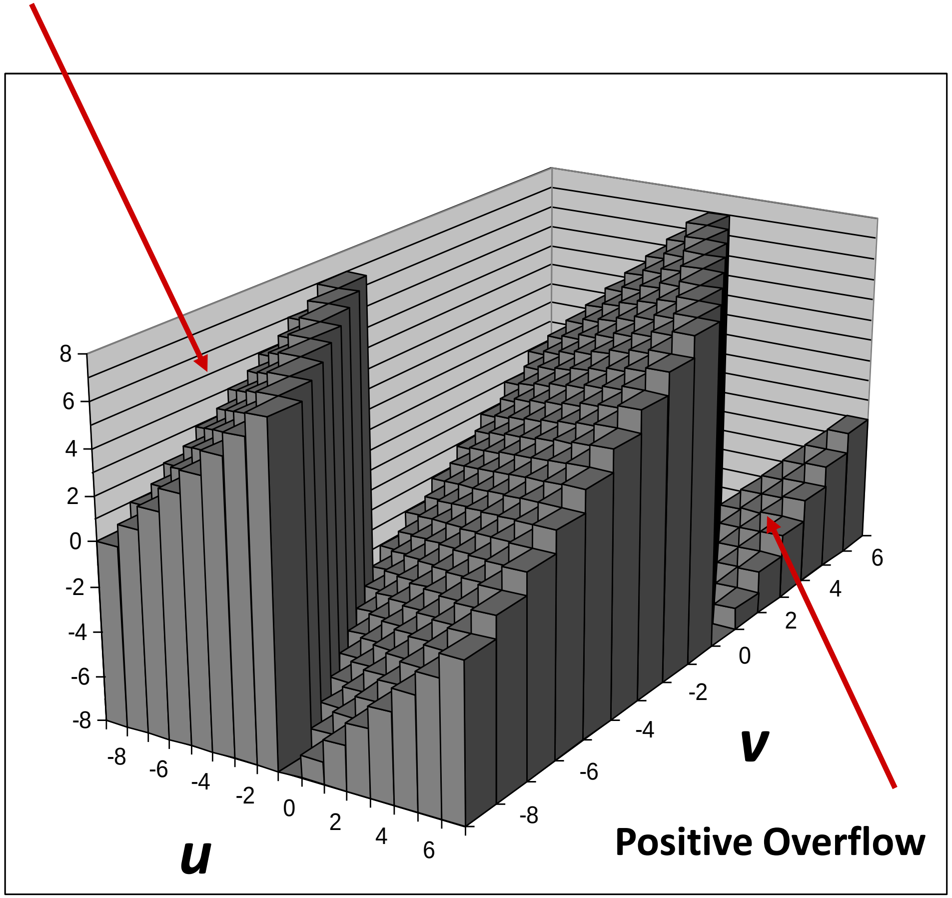
- **Example #2:**

	1			1	1	1				
		1	0	0	1	1	1	0	0	-100_{10}
+		1	1	0	0	1	1	1	0	-50_{10}
	1	0	1	1	0	1	0	1	0	~~106_{10}~~

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



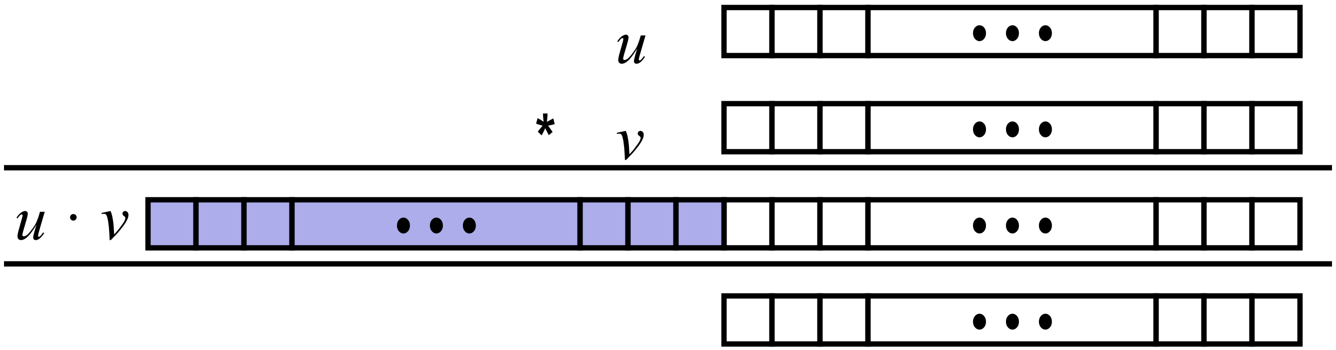
Negative Overflow

Positive 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 2*w* 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 2*w*-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 2*w* 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



Operands: *w* bits

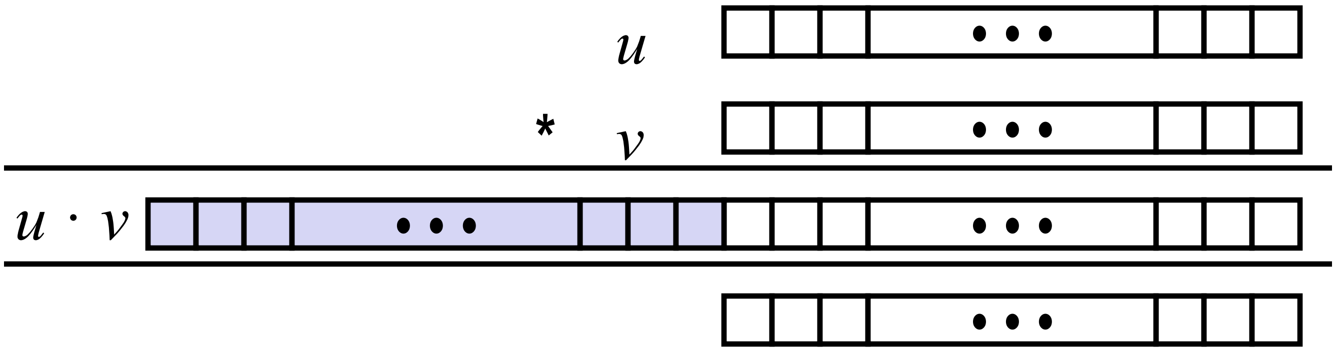
True Product: 2**w* bits

Discard *w* bits: *w* bits

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

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

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

True Binary Multiplication

- **Multiply <u>positive</u> integers using the same place-value algorithm you learned in grade school**

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

True Binary Multiplication

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

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

	0	1	1	1	1	0	1	1	123_{10}
X	1	1	1	0	1	0	1	0	234_{10}

True Binary Multiplication

- **Multiply <u>positive</u> integers using the same place-value algorithm you learned in grade school**

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

$$
\begin{array}{rccccccccl}
 & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & 123_{10} \\
\text{X} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} & 234_{10} \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &
\end{array}
$$

True Binary Multiplication

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

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

$$\begin{array}{rl} \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1} & \quad 123_{10} \\ \text{X}\ \ \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0} & \quad 234_{10} \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ & \end{array}$$

True Binary Multiplication

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

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

$$\begin{array}{ccccccccccccc}
 & & & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & & 123_{10} \\
\text{X} & & & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} & & 234_{10} \\
\hline
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 & & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & & & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & &
\end{array}$$

True Binary Multiplication

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

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

$$\begin{array}{rl} \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1} & \quad 123_{10} \\ \times\ \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0} & \quad 234_{10} \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1 & \\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1 & \end{array}$$

True Binary Multiplication

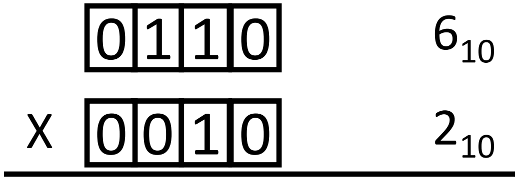
- **Multiply <u>positive</u> integers using the same place-value algorithm you learned in grade school**

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

$$\begin{array}{cccccccccccccccccc}
 & & & & & & & & & 0&1&1&1&1&0&1&1 & 123_{10} \\
\text{X} & & & & & & & & & 1&1&1&0&1&0&1&0 & 234_{10} \\ \hline
 & & & & & & & & & 0&0&0&0&0&0&0&0 & \\
 & & & & & & & & 0&1&1&1&1&0&1&1 & & \\
 & & & & & & & 0&0&0&0&0&0&0&0 & & & \\
 & & & & & & 0&1&1&1&1&0&1&1 & & & & \\
 & & & & & 0&0&0&0&0&0&0&0 & & & & & \\
 & & & & 0&1&1&1&1&0&1&1 & & & & & & \\
 & & & 0&1&1&1&1&0&1&1 & & & & & & & \\
+ & & 0&1&1&1&1&0&1&1 & & & & & & & & \\ \hline
 & 0&1&1&1&0&0&0&0&0&1&1&0&1&1&1&0 & 28782_{10}
\end{array}$$

Power-of-2 Multiply with Shift

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



0110 6_{10}

X 0010 2_{10}

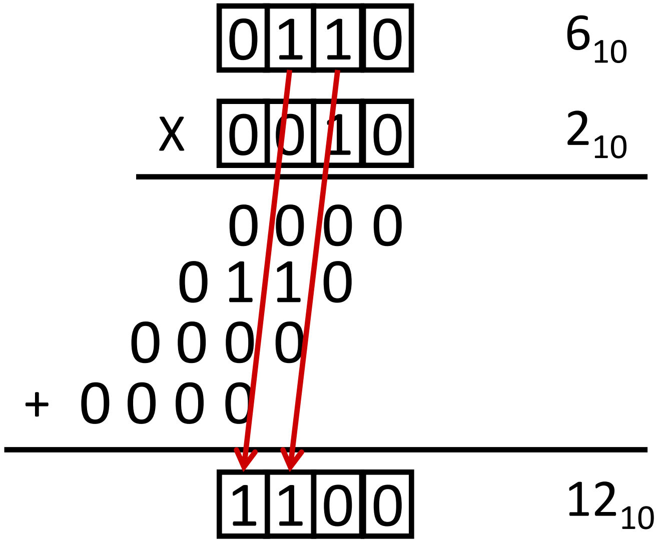
Power-of-2 Multiply with Shift

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

$$\begin{array}{cccccccccc}
 & & & & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{0} & & 6_{10} \\
 & & & \mathrm{X} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & & 2_{10} \\
\hline
 & & & & 0 & 0 & 0 & 0 & & \\
 & & & 0 & 1 & 1 & 0 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
+ & 0 & 0 & 0 & 0 & & & & & \\
\hline
 & & & & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{0} & & 12_{10}
\end{array}$$

Power-of-2 Multiply with Shift

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



0 1 1 0 6_{10}

X 0 0 1 0 2_{10}

0 0 0 0

0 1 1 0

0 0 0 0

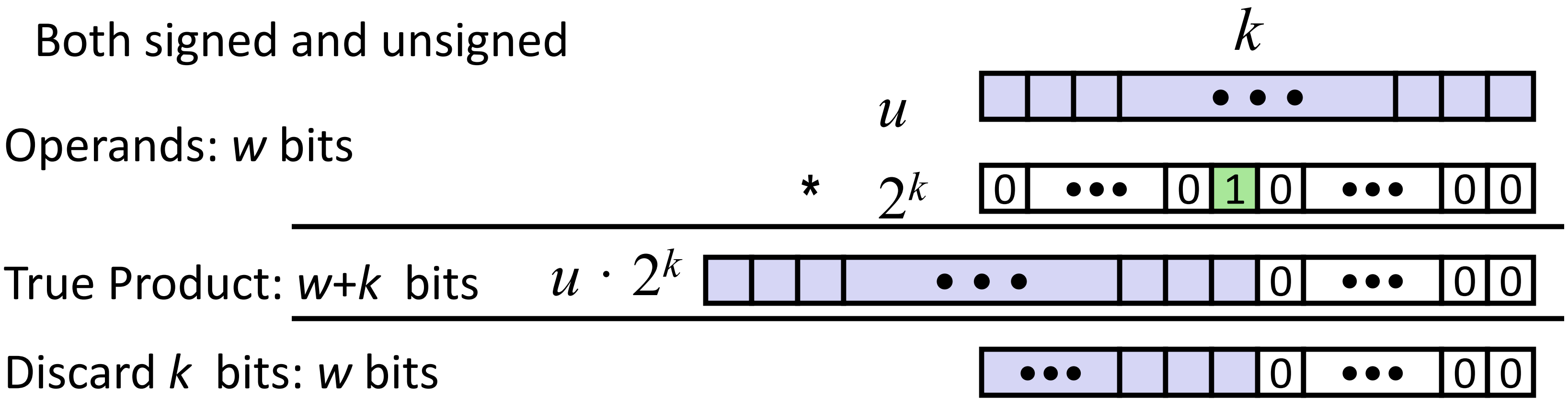
\+ 0 0 0 0

1 1 0 0 12_{10}

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

Power-of-2 Multiply with Shift

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



Operands: w bits

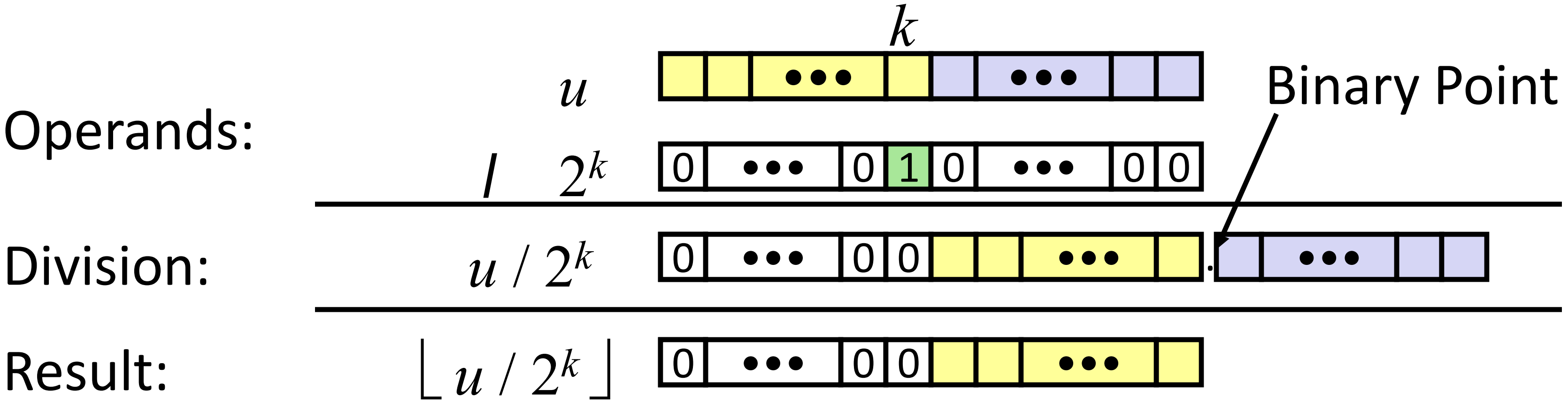
True Product: $w+k$ bits

Discard k bits: w bits

- **Examples**
  - `u << 3 == u * 8`
  - `(u << 5) – (u << 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 >> k` gives $\lfloor \mathbf{u} / 2^k \rfloor$
  - 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`

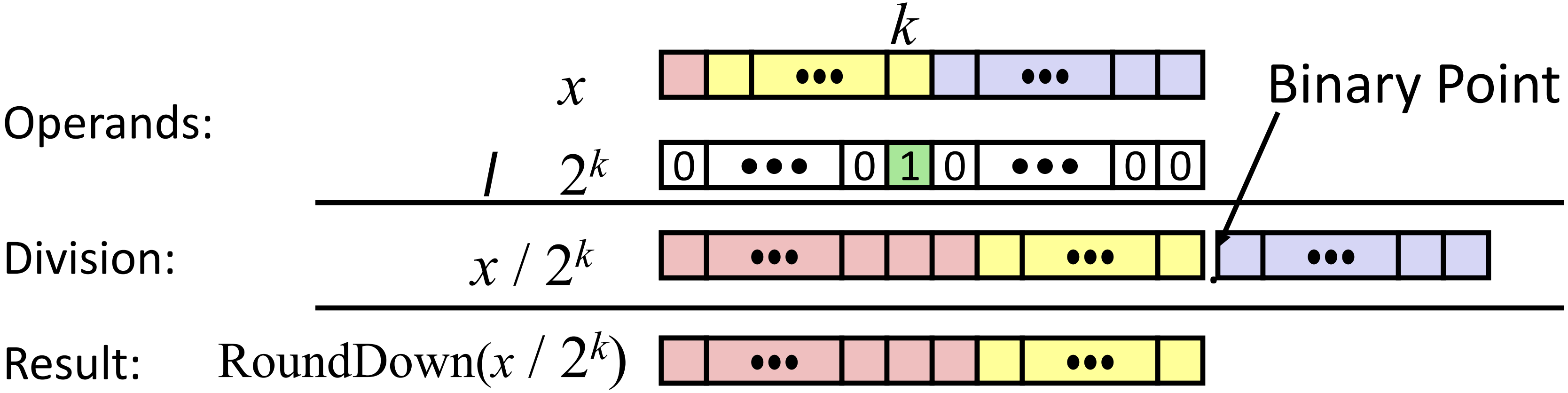
Incorrect Power-of-2 Divide

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

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

Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - `x >> k` gives $\lfloor \mathtt{x} / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0`

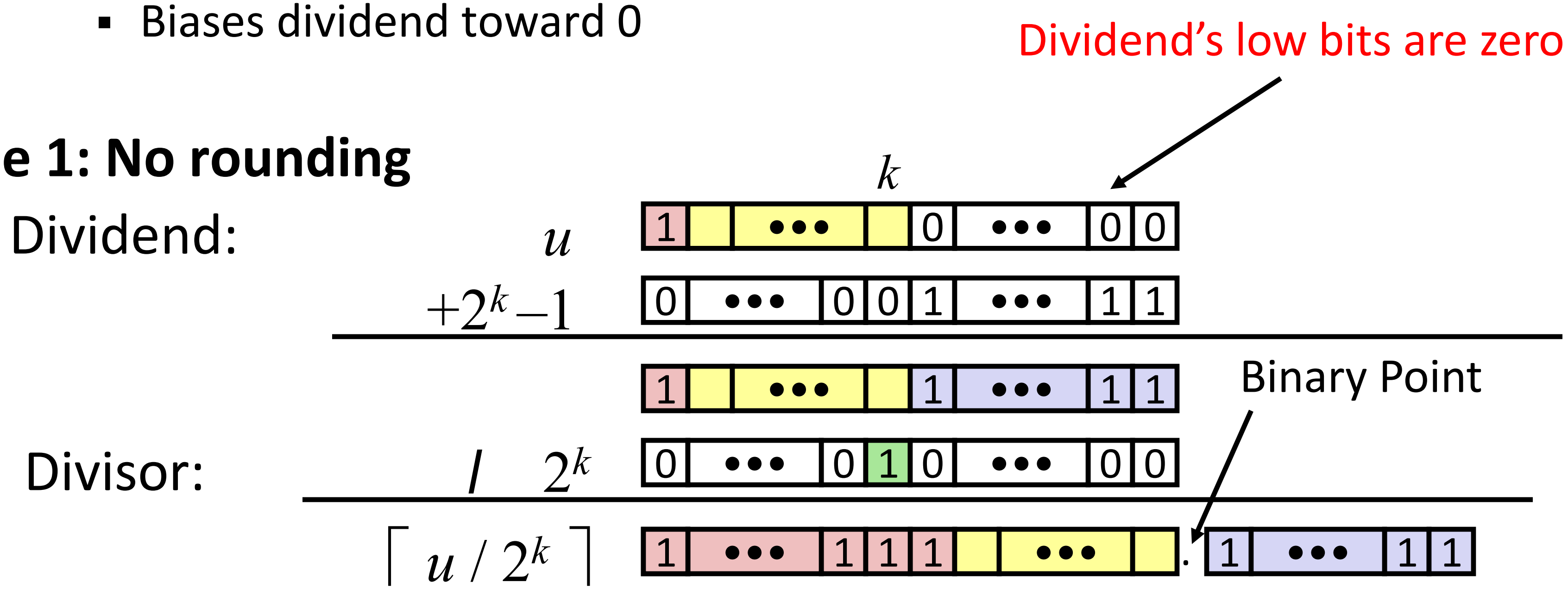


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide with *Biasing*

- **Quotient of Negative Number by Power of 2**
  - Want $\lceil \mathtt{x} \;/\; 2^k \rceil$ (Round Toward 0)
  - Compute as $\lfloor (\mathtt{x}+2^k-1) \,/\; 2^k \rfloor$
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0

Case 1: No rounding



Biasing has no effect

Biasing without changing result

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

Without bias:

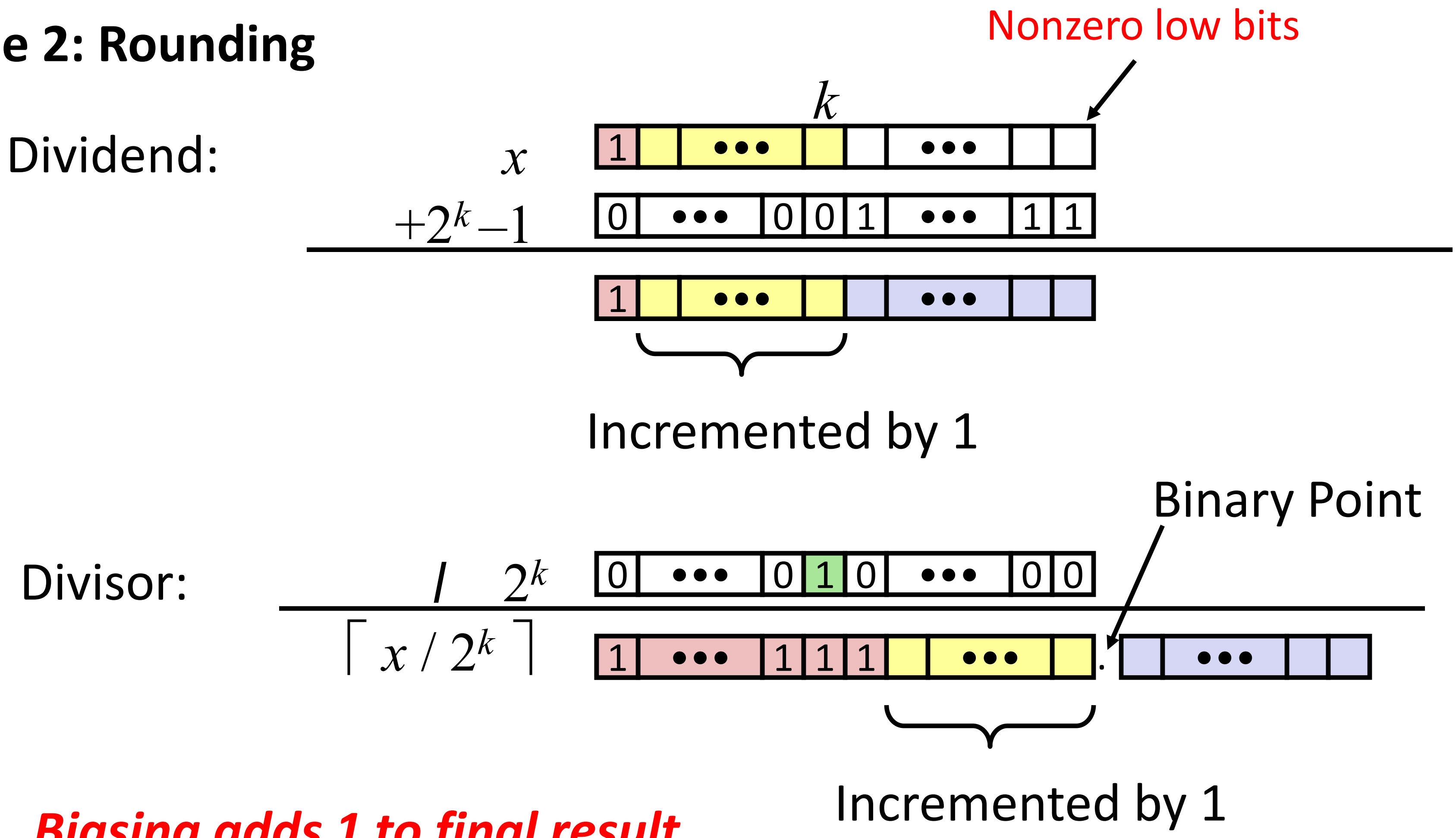
1. $-20_{10} = 11101100_2$
2. $(-20 \ / \ 4\ )$ becomes $(11101100_2 >> 2)$
3. $(11101100_2 >> 2) = 11111011_2$
4. $11111011_2 = -5$

With bias:

1. $-20_{10} + 3_{10} = 11101111_2$
2. $(-23 \ / \ 4\ )$ becomes $(11101111_2 >> 2)$
3. $(11101111_2 >> 2) = 11111011_2$
4. $11111011_2 = -5$

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Nonzero low bits

Dividend: x

$+2^k - 1$

Incremented by 1

Binary Point

Divisor: $/ \quad 2^k$

$\lceil x / 2^k \rceil$

Incremented by 1

Biasing adds 1 to final result

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 >> 2)
3. (11101011_2 >> 2) = 11111010_2
4. $11111010_2 = -6$ (incorrect!)

With bias:

1. $-21_{10} + 3_{10} = 11101110_2$
2. (-18 / 4) becomes (11101110_2 >> 2)
3. (11101110_2 >> 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 $(11101011_2 >> 2)$
3. $(11101011_2 >> 2) = 11111010_2$
4. $11111010_2 = -6$ **(incorrect!)**

With bias:

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

Recall- lowest order bit has value 1!

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