# 15-213

*"The course that gives CMU its Zip!"*

# Integer Arithmetic Operations
# Jan. 25, 2001

## Topics

- **Basic operations**
  - Addition, negation, multiplication
- **Programming Implications**
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide

# C Puzzles

- **Taken from Exam #2, CS 347, Spring '97**
- **Assume machine with 32 bit word size, two's complement integers**
- **For each of the following C expressions, either:**
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

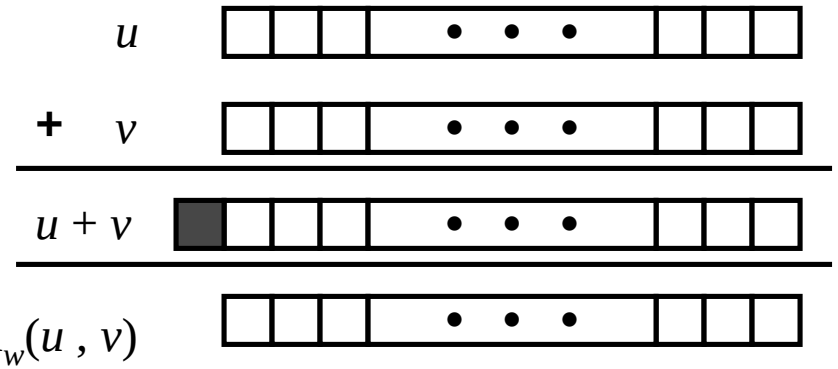
- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

# Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits




## Standard Addition Function

- **Ignores carry output**

## Implements Modular Arithmetic

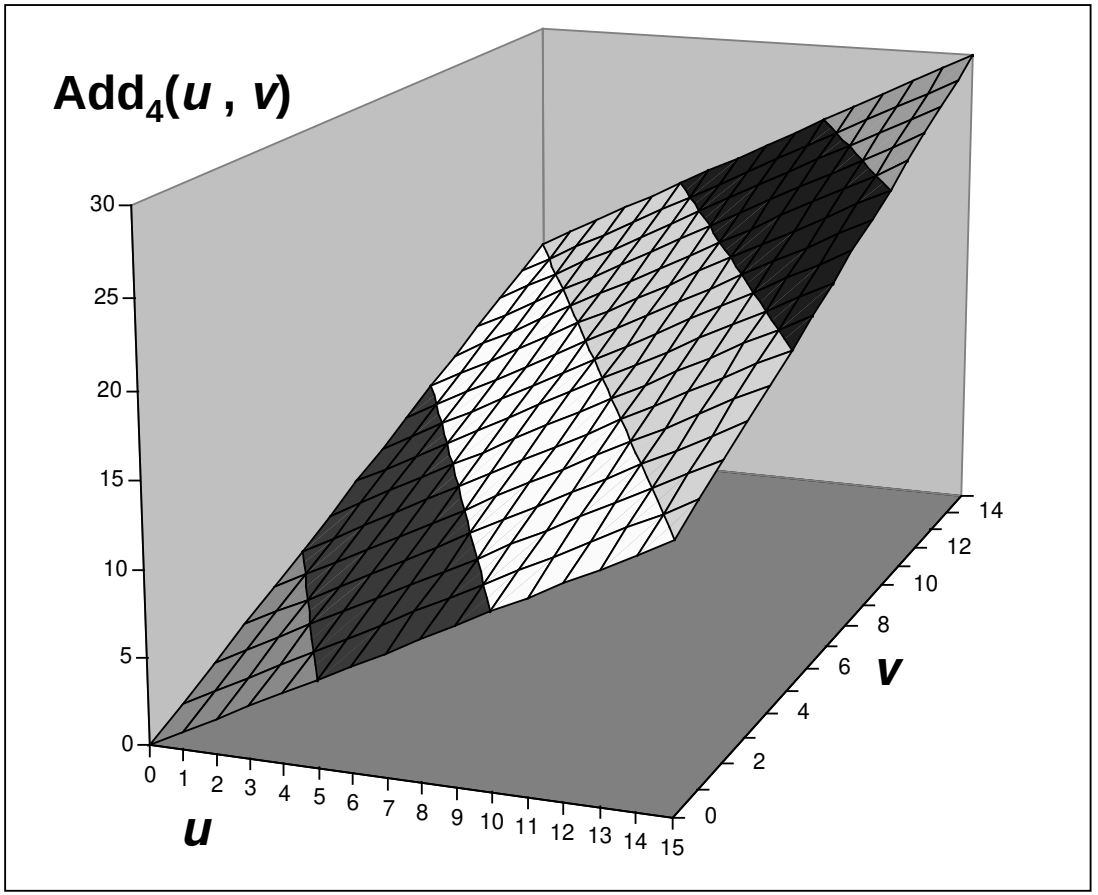
$s = \mathrm{UAdd}_w(u , v) = u + v \mod 2^w$

$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

# Visualizing Integer Addition

## Integer Addition

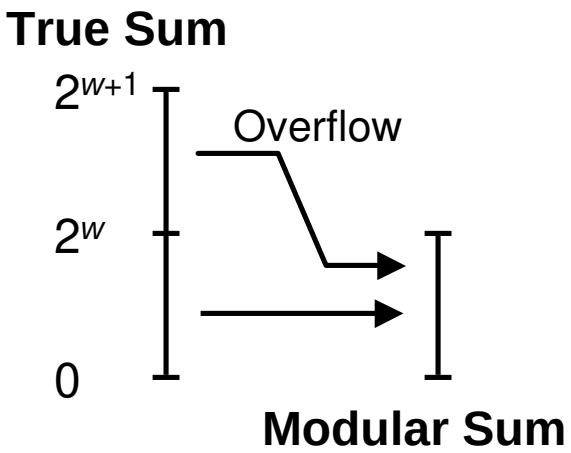
- **4-bit integers *u* and *v***
- **Compute true sum $\text{Add}_4(u, v)$**
- **Values increase linearly with *u* and *v***
- **Forms planar surface**

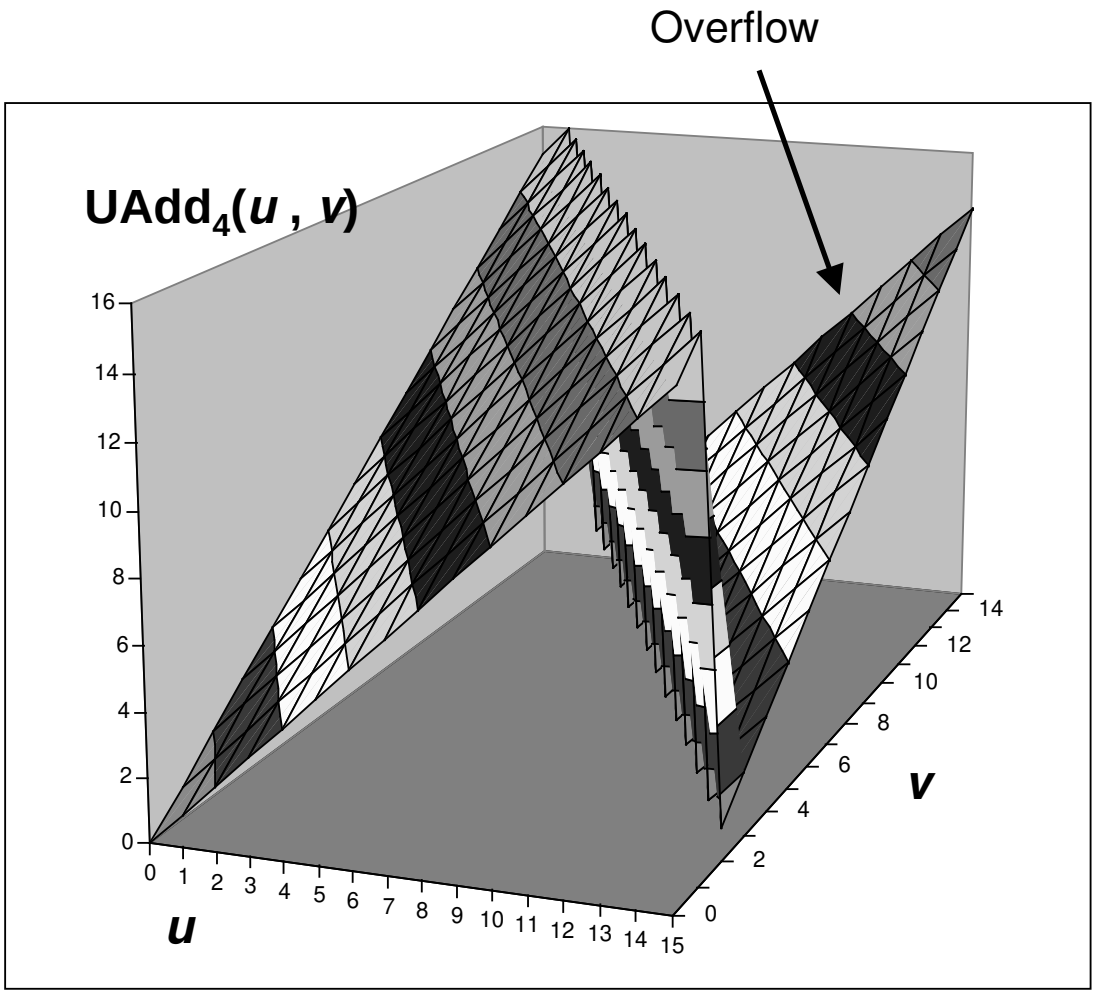


# Visualizing Unsigned Addition

## Wraps Around

- **If true sum ≥ $2^w$**
- **At most once**



# 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 $\text{UComp}_w(u) = 2^w - u$

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

# Detecting Unsigned Overflow

## Task

- **Given** $s = \text{UAdd}_w(u, v)$
- **Determine if** $s = u + v$

## Application

```
unsigned s, u, v;
s = u + v;
```

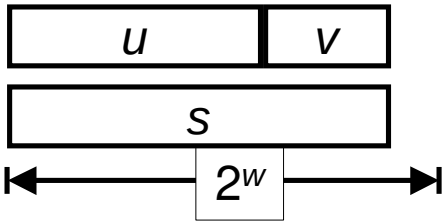
- **Did addition overflow?**

## Claim

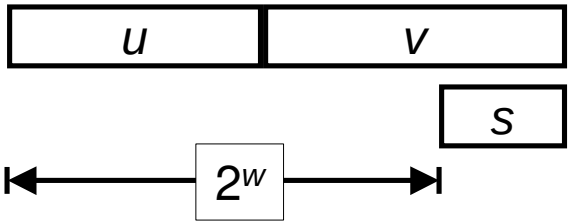
- **Overflow iff** $s < u$

  `ovf = (s < u)`
- **Or symmetrically iff** $s < v$

**No Overflow**



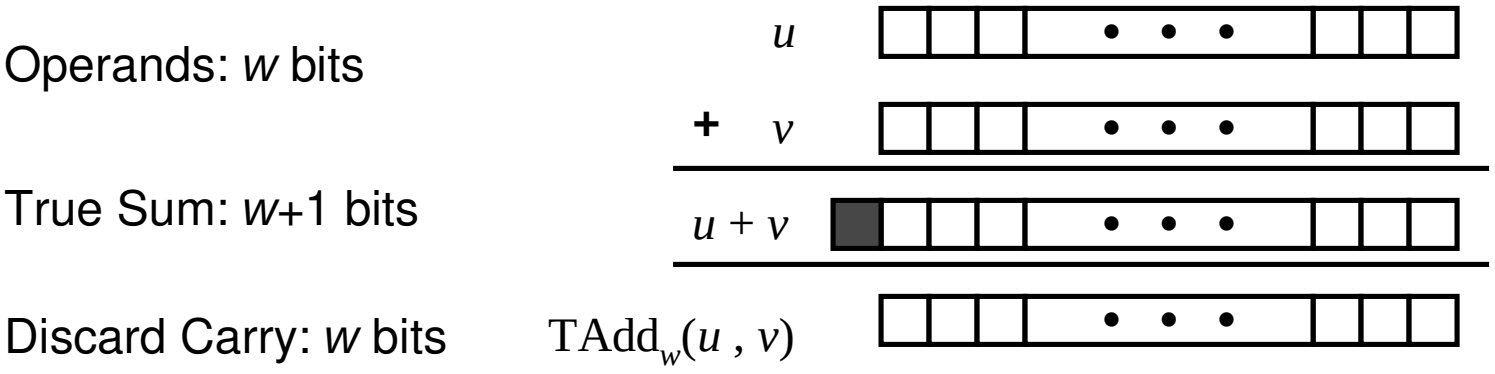

**Overflow**




## Proof

- **Know that** $0 \le v < 2^w$
- **No overflow** $\Rightarrow \quad s = u + v \quad \ge \quad u + 0 = u$
- **Overflow** $\Rightarrow \quad s = u + v - 2^w \quad < \quad u + 0 = u$

# Two's Complement Addition




## TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

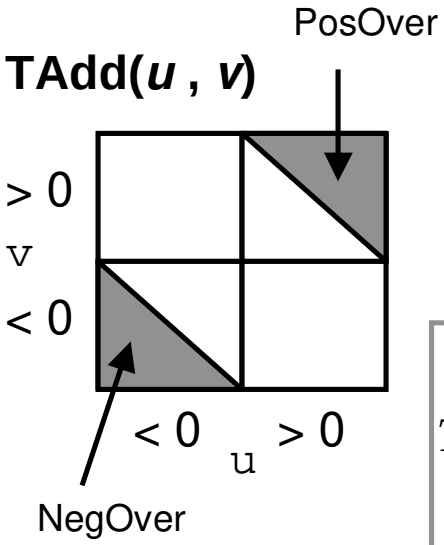
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
 t = u + v
```

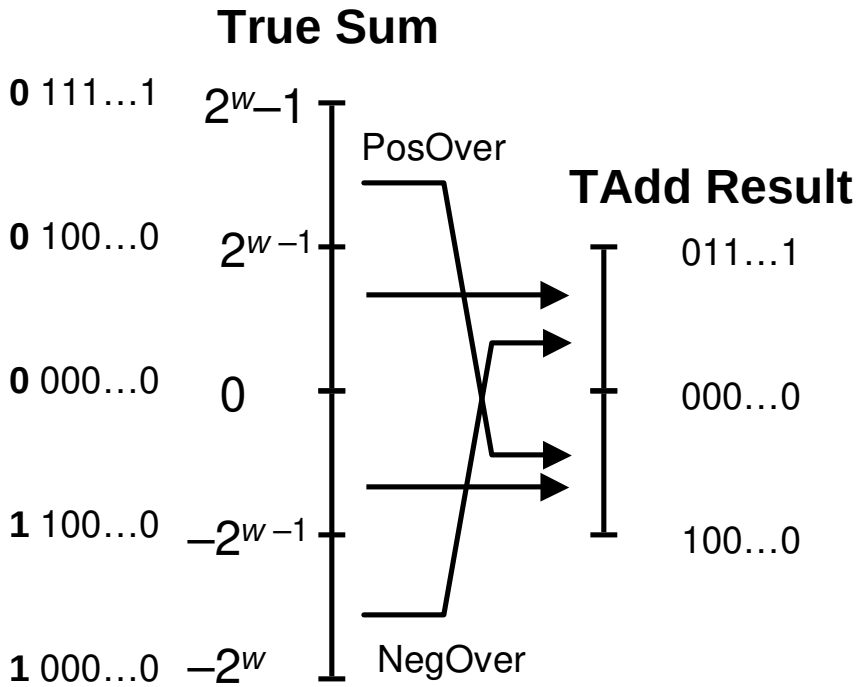
- **Will give `s == t`**

# Characterizing TAdd

## Functionality

- **True sum requires *w*+1 bits**
- **Drop off MSB**
- **Treat remaining bits as 2's comp. integer**




$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

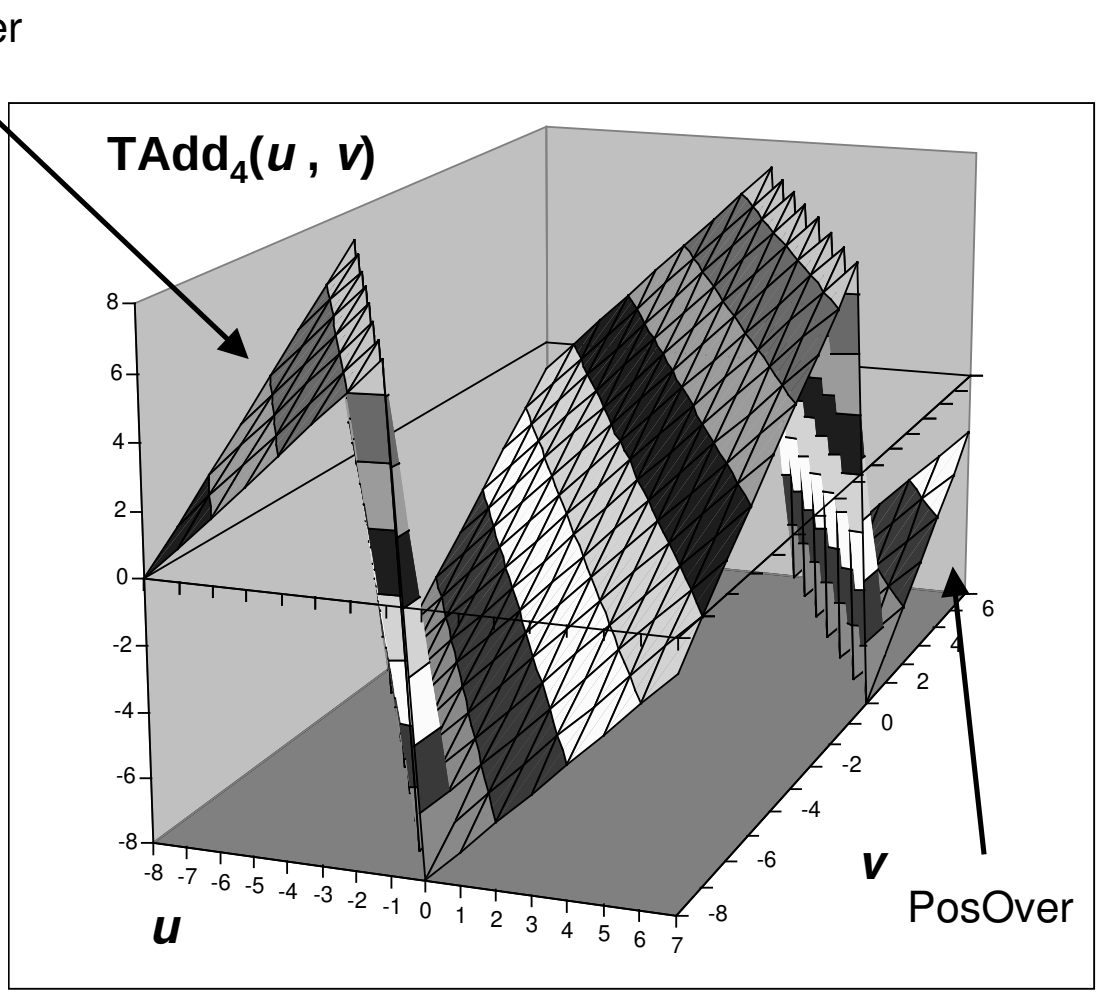
# Visualizing 2's Comp. Addition

## Values

- **4-bit two's comp.**
- **Range from -8 to +7**

## Wraps Around

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

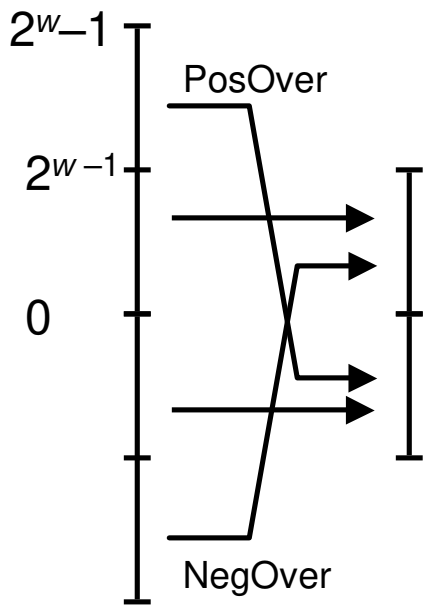


# Detecting 2's Comp. Overflow

## Task

- **Given** $s = \text{TAdd}_w(u , v)$
- **Determine if** $s = \text{Add}_w(u , v)$
- **Example**

```
int s, u, v;
s = u + v;
```




## Claim

- **Overflow iff either:**

  $u, v < 0, s \geq 0$ (NegOver)

  $u, v \geq 0, s < 0$ (PosOver)

```
ovf = (u<0 == v<0) && (u<0 != s<0);
```

## Proof

- **Easy to see that if** $u \geq 0$ **and** $v < 0$**, then** $TMin_w \leq u + v \leq TMax_w$
- **Symmetrically if** $u < 0$ **and** $v \geq 0$
- **Other cases from analysis of** TAdd

# Mathematical Properties of TAdd

## Isomorphic Algebra to UAdd

- $TAdd_w(u , v) = U2T(UAdd_w(T2U(u ), T2U(v)))$
  - Since both have identical bit patterns

## Two's Complement Under TAdd Forms a Group

- **Closed, Commutative, Associative, 0 is additive identity**
- **Every element has additive inverse**

  Let $TComp_w(u) = U2T(UComp_w(T2U(u ))$

  $TAdd_w(u , TComp_w(u )) = 0$

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

# Two's Complement Negation

**Mostly like Integer Negation**

- TComp($u$) = $-u$

***TMin* is Special Case**

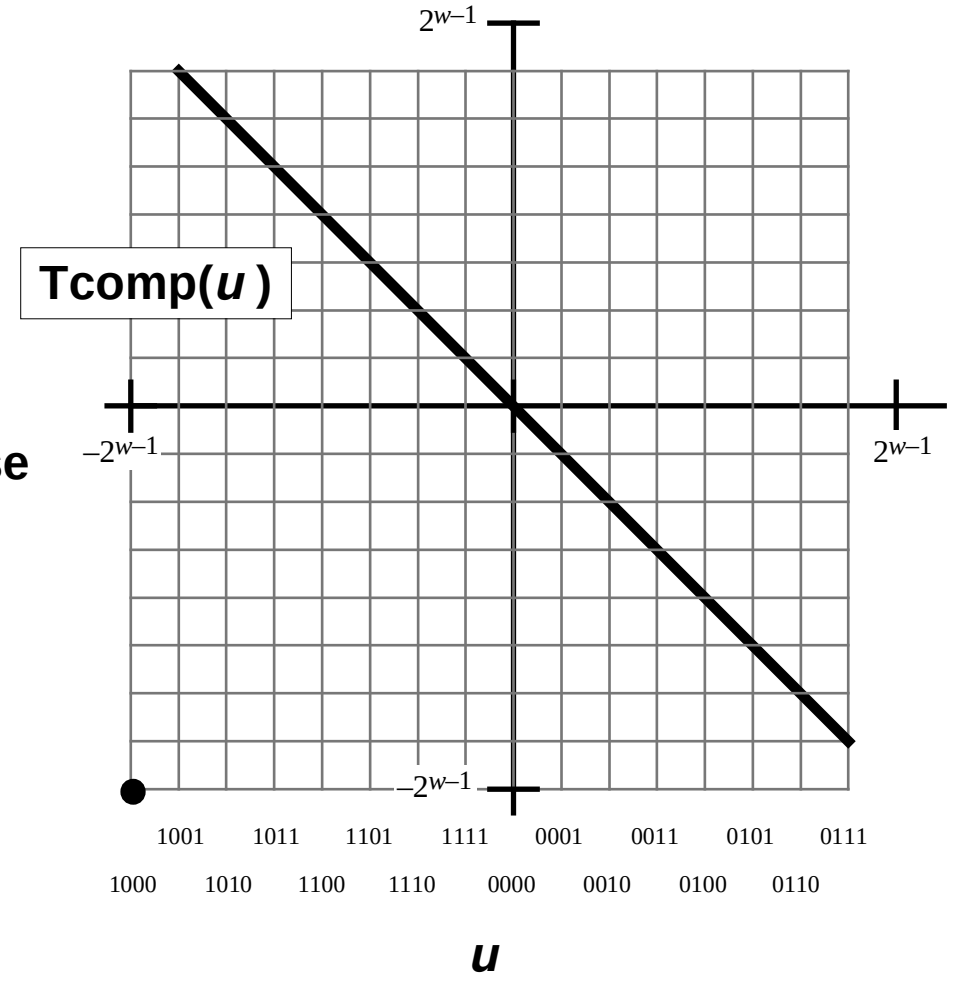
- TComp(*TMin*) = *TMin*

**Negation in C is Actually** TComp

```
mx = -x
```

- *mx* = TComp(*x*)
- **Computes additive inverse for** TAdd

```
x + -x == 0
```



# Negating with Complement & Increment

## In C

```
~x + 1 == -x
```

## Complement

- **Observation: `~x + x == 1111…11`$_2$ `== -1`**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | x | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| + | ~x | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| | −1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## Increment

- `~x + x + (-x + 1)  ==  -1 + (-x + 1)`
- `~x + 1  ==  -x`

## Warning: Be cautious treating `int`'s as integers

- **OK here: We are using group properties of** TAdd **and** TComp

# Comp. & Incr. 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 |
| y | -15213 | C4 93 | 11000100 10010011 |

TMin

| | Decimal | Hex | Binary |
|---|---|---|---|
| TMin | -32768 | 80 00 | 10000000 00000000 |
| ~TMin | 32767 | 7F FF | 01111111 11111111 |
| ~TMin+1 | -32768 | 80 00 | 10000000 00000000 |

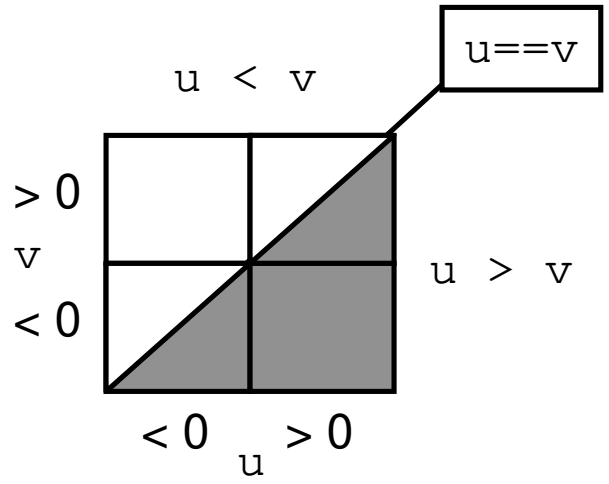
0

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

# Comparing Two's Complement Numbers

## Task

- **Given** signed numbers *u*, *v*
- Determine whether or not *u* > *v*
  - Return 1 for numbers in shaded region below



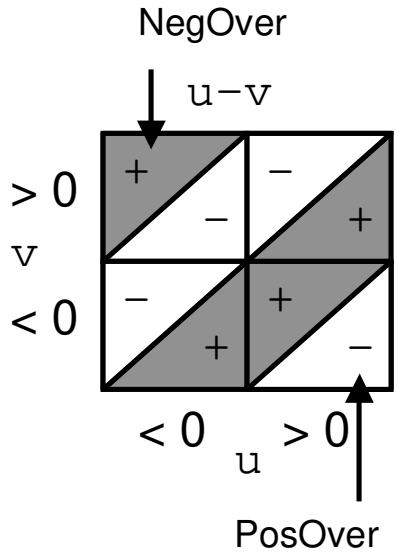

## Bad Approach

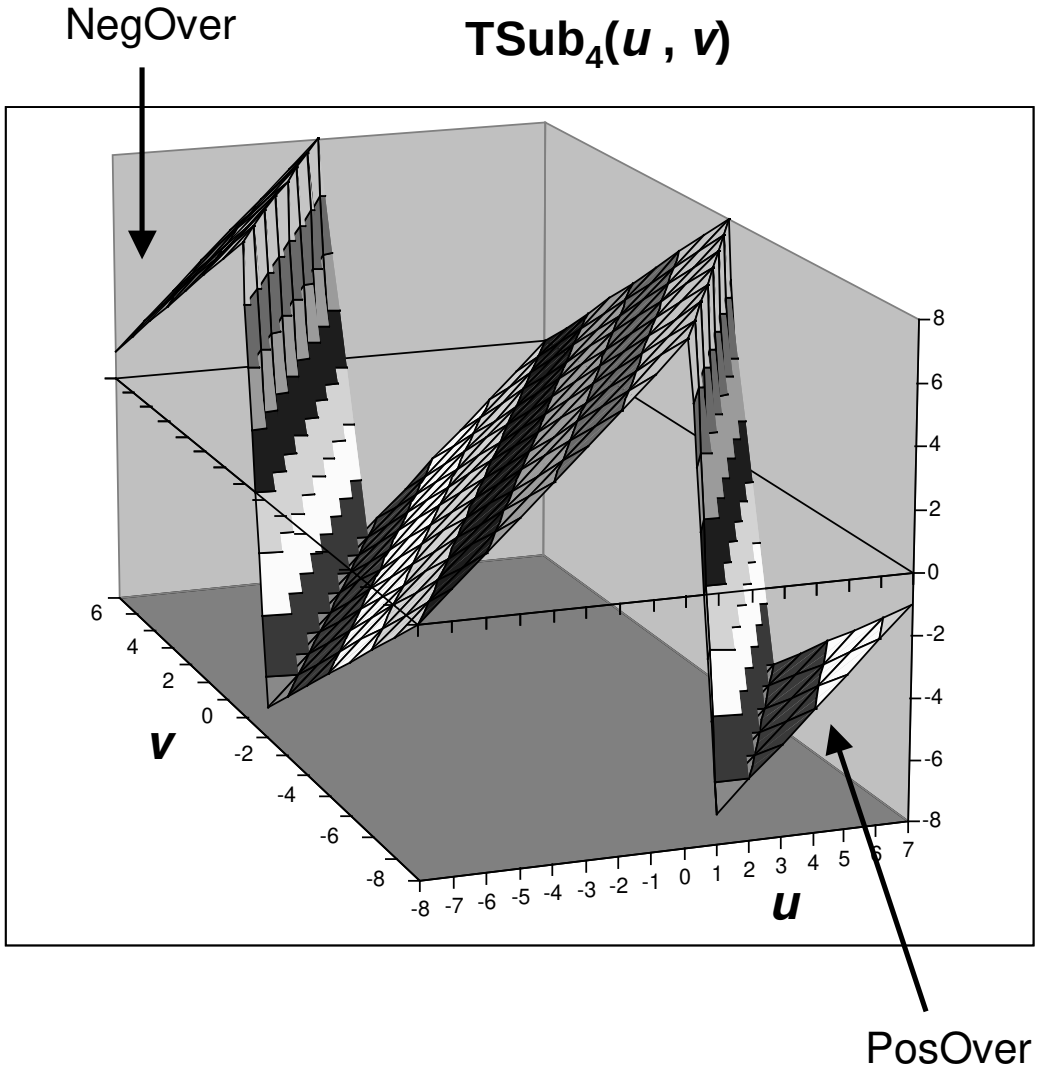
- Test `(u-v) > 0`
  - TSub(*u*,*v*) = TAdd(*u*, TComp(*v*))
- Problem: Thrown off by either Negative or Positive Overflow

# Comparing with TSub

## Will Get Wrong Results

- **NegOver:** `u < 0, v > 0`
  - but `u-v > 0`
- **PosOver:** `u > 0, v < 0`
  - but `u-v < 0`

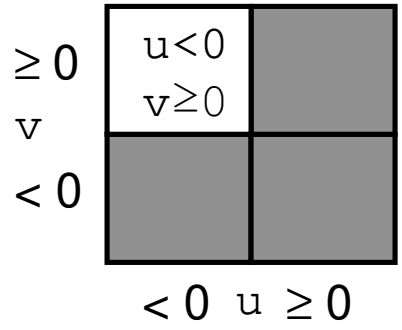


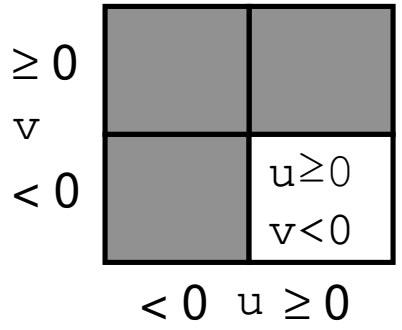
# Working Around Overflow Problems

## Partition into Three Regions

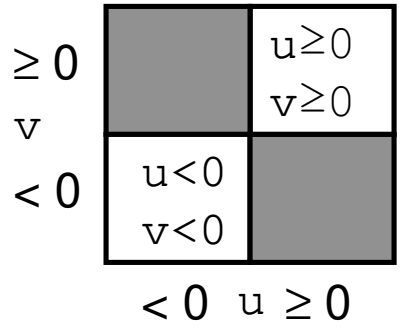
- $u < 0, v \geq 0 \quad \Rightarrow \quad u < v$

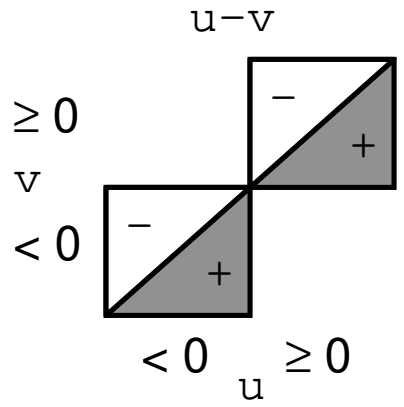



- $u \geq 0, v < 0 \quad \Rightarrow \quad u > v$




- *u*, *v* same sign ⇒ `u-v` does not overflow
  - **Can safely use test** `(u-v) > 0`



# Multiplication

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

- **Either signed or unsigned**

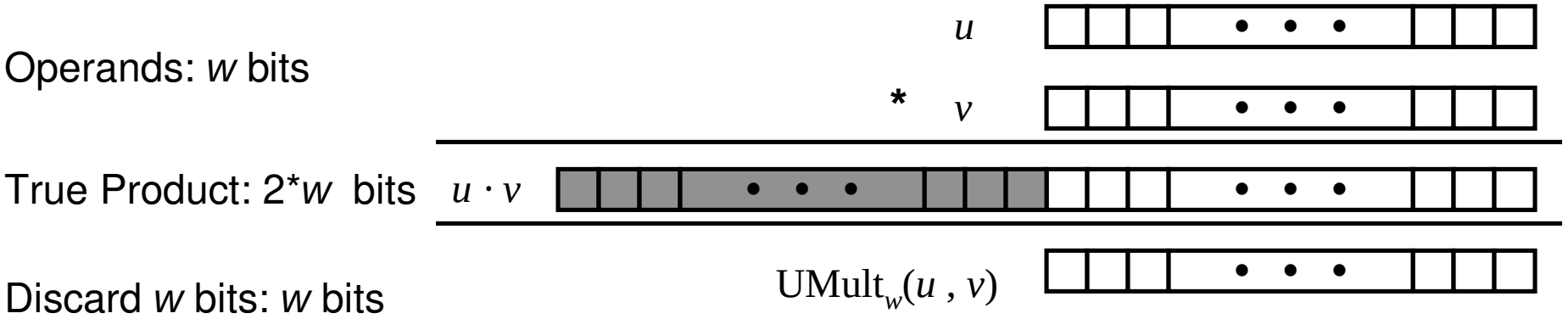
## Ranges

- **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**
- **Also implemented in Lisp, ML, and other "advanced" languages**

# Unsigned Multiplication in C




## Standard Multiplication Function

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

## Implements Modular Arithmetic

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

# Unsigned vs. Signed Multiplication

## Unsigned Multiplication

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

- **Truncates product to *w*-bit number** $up = \text{UMult}_w(ux, uy)$
- **Simply 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** $p = \text{TMult}_w(x, y)$

## Relation

- **Signed multiplication gives same bit-level result as unsigned**
- **`up == (unsigned) p`**

# Multiplication Examples

```
short int x  =  15213;
int     txx  =  ((int) x) * x;
int      xx  =  (int) (x * x);
int     xx2  =  (int) (2 * x * x);
```

```
x   =      15213:                      00111011 01101101
txx =  231435369: 00001101 11001011 01101100 01101001
xx  =      27753: 00000000 00000000 01101100 01101001
xx2 =     -10030: 11111111 11111111 11011000 11010010
```

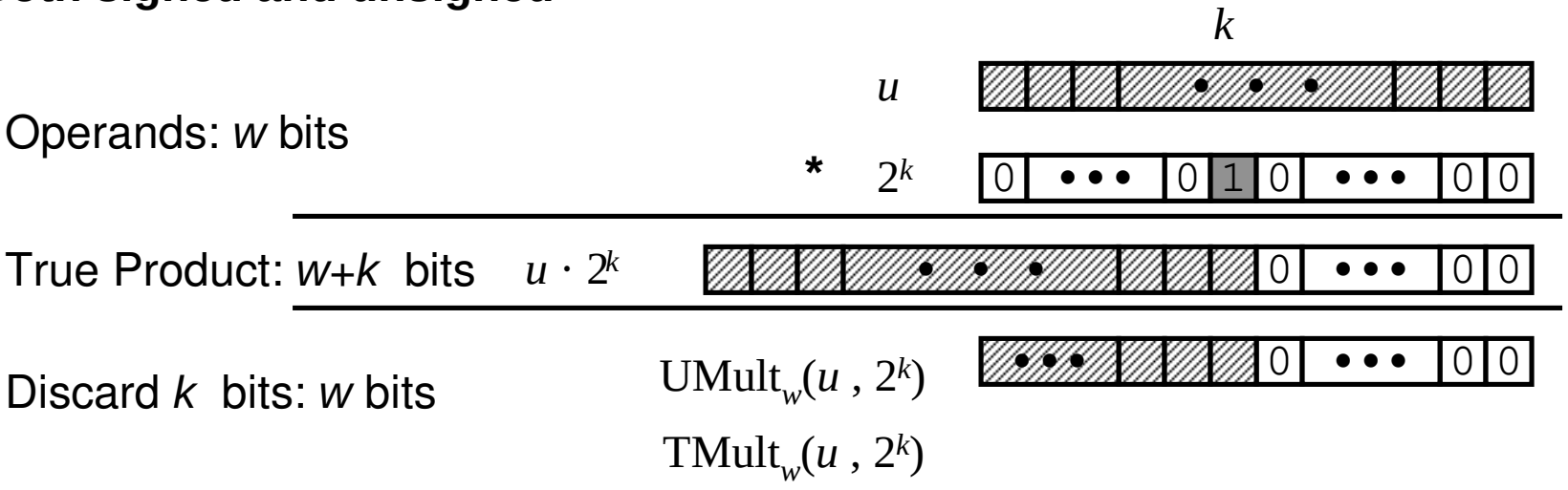
## Observations

- **Casting order important**
  - If either operand **`int`**, will perform **`int`** multiplication
  - If both operands **`short int`**, will perform **`short int`** multiplication
- **Really is modular arithmetic**
  - Computes for `xx`: $15213^2$ mod `65536 = 27753`
  - Computes for `xx2`: `(int) 55506U = -10030`
- **Note that `xx2 == (xx << 1)`**

# Power-of-2 Multiply with Shift

## Operation

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



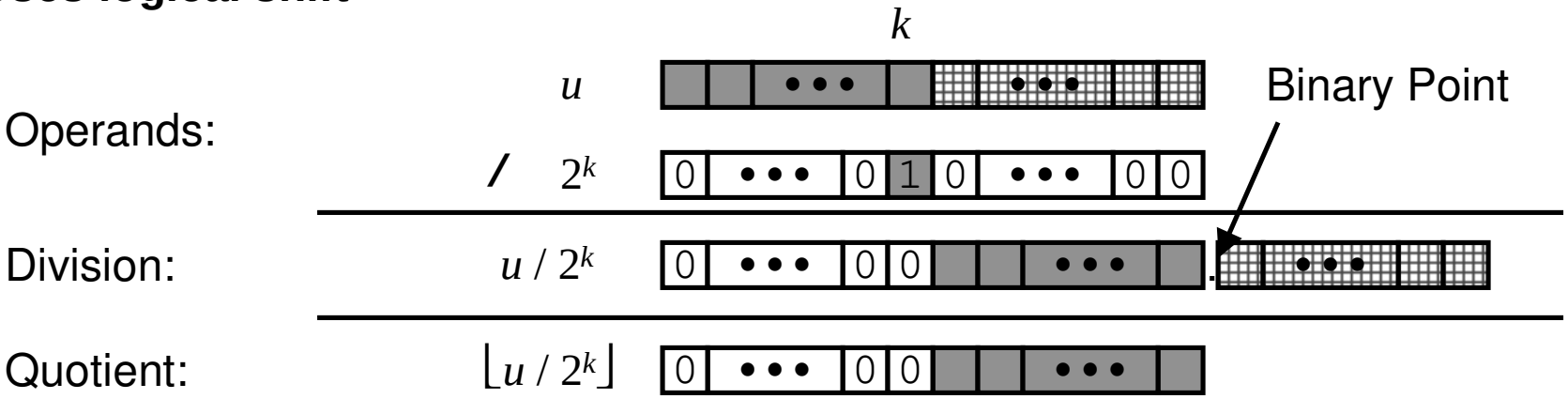

## Examples

- **u << 3 == u * 8**
- **u << 5 - u << 3 == u * 24**
- **Most machines shift and add much faster than multiply**
  - Compiler will generate 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 shift**

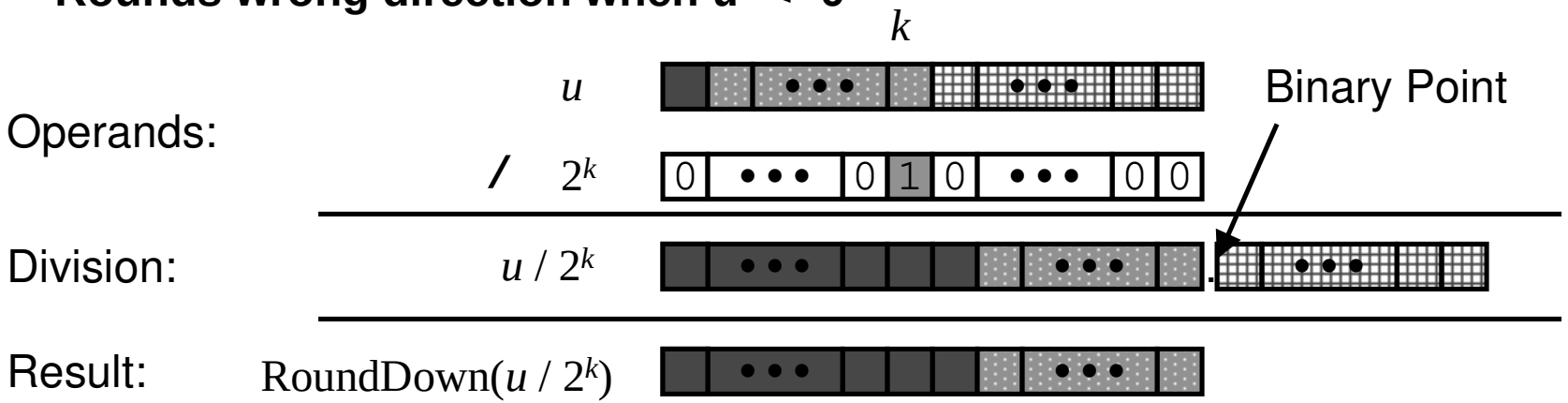



| | Division | Computed | Hex | Binary |
|---|---|---|---|---|
| x | 15213 | 15213 | 3B 6D | 00111011 01101101 |
| x >> 1 | 7606.5 | 7606 | 1D B6 | **0**0011101 10110110 |
| x >> 4 | 950.8125 | 950 | 03 B6 | **0000**0011 10110110 |
| x >> 8 | 59.4257813 | 59 | 00 3B | **00000000** 00111011 |

# 2's Comp Power-of-2 Divide with Shift

## Quotient of Signed by Power of 2

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



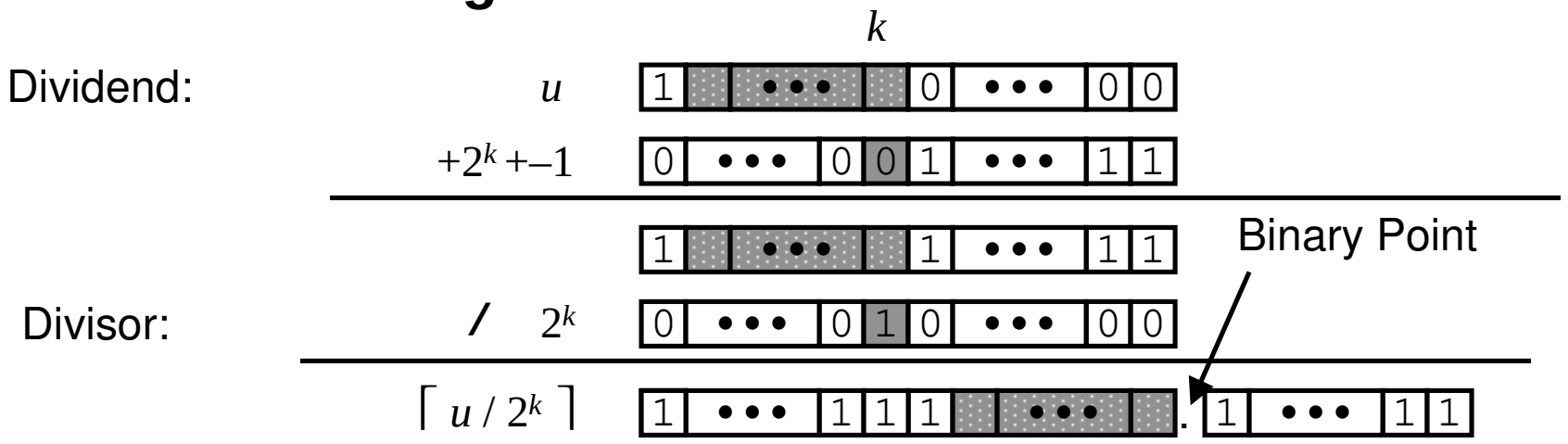

| | Division | Computed | Hex | Binary |
|---|---|---|---|---|
| y | -15213 | -15213 | C4 93 | 11000100 10010011 |
| y >> 1 | -7606.5 | -7607 | E2 49 | **1**1100010 01001001 |
| y >> 4 | -950.8125 | -951 | FC 49 | **1111**1100 01001001 |
| y >> 8 | -59.4257813 | -60 | FF C4 | **11111111** 11000100 |

# Correct Power-of-2 Divide

## Quotient of Negative Number by Power of 2

- **Want** ⌈ **u / $2^k$** ⌉ **(**Round Toward 0)
- **Compute as** ⌊ **(u+$2^k$-1)/ $2^k$** ⌋
  - In C: **(u + (1<<k)-1) >> k**
  - Biases dividend toward 0

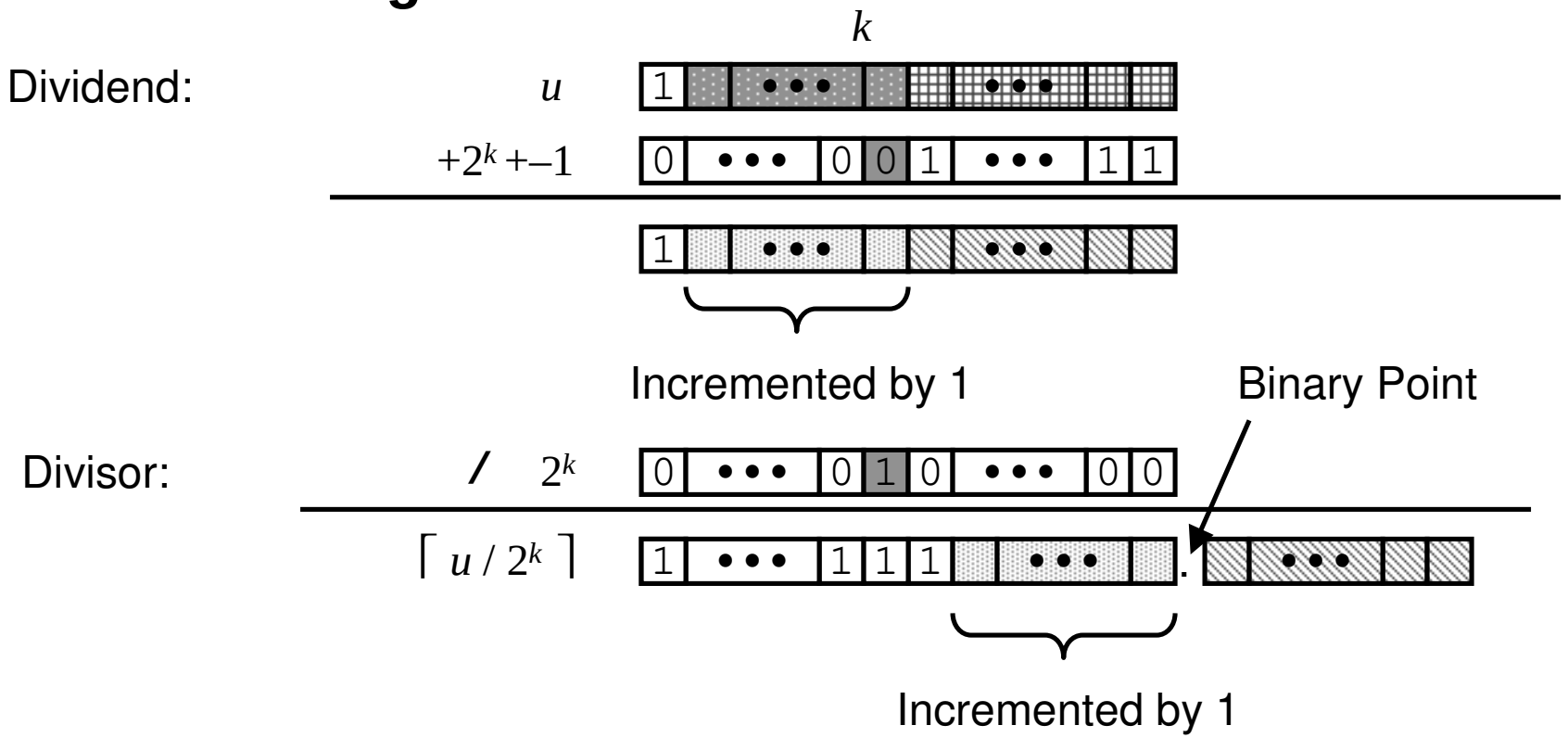
## Case 1: No rounding




Biasing has no effect

# Correct Power-of-2 Divide (Cont.)

## Case 2: Rounding




Biasing adds 1 to final result

# Correct Power-of-2 Divide Examples

| | $y/2^k$ | Computed | Hex | Binary |
|---|---|---|---|---|
| y | -15213 | -15213 | C4 93 | 11000100 10010011 |
| y+1 | | -15212 | C4 94 | 11000100 10010***100*** |
| (y+1)>>1 | -7606.5 | -7606 | E2 4A | **1**1100010 01001***10*** |
| y | -15213 | -15213 | C4 93 | 11000100 10010011 |
| y+15 | | -15197 | C4 A2 | 11000100 10***100010*** |
| (y+15)>>4 | -950.8125 | -950 | FC 4A | **1111**1100 01001***10*** |
| y | -15213 | -15213 | C4 93 | 11000100 10010011 |
| y+255 | | -14958 | C5 92 | 11000101 ***10010010*** |
| (y+255)>>8 | -59.4257813 | -59 | FF C5 | **11111111** 11000101 |

# 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 complement arithmetic**

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

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

# C Puzzle Answers

- **Assume machine with 32 bit word size, two's complement integers**
- *TMin* **makes a good counterexample in many cases**

| | | | |
|---|---|---|---|
| • `x < 0` | ⇒ | `((x*2) < 0)` | **False:** *TMin* |
| • `ux >= 0` | | | **True:** 0 = *UMin* |
| • `x & 7 == 7` | ⇒ | `(x<<30) < 0` | **True:** $x_1 = 1$ |
| • `ux > -1` | | | **False:** 0 |
| • `x > y` | ⇒ | `-x < -y` | **False:** −1, *TMin* |
| • `x * x >= 0` | | | **False:** 65535 (x*x = −131071) |
| • `x > 0 && y > 0` | ⇒ | `x + y > 0` | **False:** *TMax*, *TMax* |
| • `x >= 0` | ⇒ | `-x <= 0` | **True:** –*TMax* < 0 |
| • `x <= 0` | ⇒ | `-x >= 0` | **False:** *TMin* |