Integers

Chapter 2 of B&O

Some notes adopted from Bryant and O'Hallaron

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

x sub 0, x sub

width of the

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

– C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

significant bit is also called the sign bit

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

bit to unsigned

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

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

Numeric Ranges

- Unsigned Values
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1
- Two's Complement Values
  - *TMin* = -2^{w-1}

    100…0
  - *TMax* = $2^{w-1} - 1$

    011…1
- Other Values
  - **Minus 1**

    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

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

  $|\mathit{TMin}| = \mathit{TMax} + 1$

  - Asymmetric range

  $\mathit{UMax} = 2 * \mathit{TMax} + 1$

- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform-specific

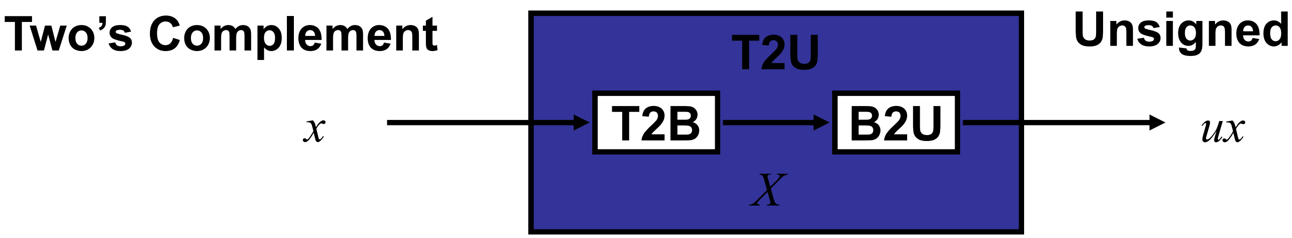
Casting Signed to Unsigned

- C Allows Conversions from Signed to Unsigned

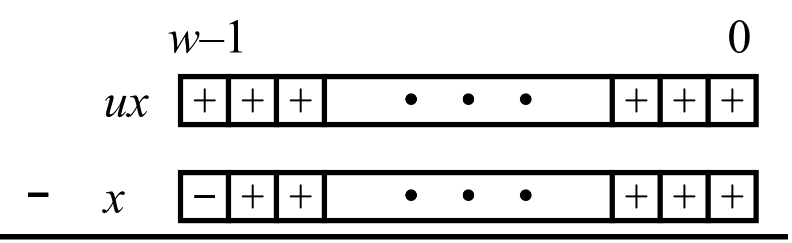
```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

- Resulting Value
  - No change in bit representation
  - Nonnegative values unchanged
    - *ux* = 15213
  - Negative values change into (large) positive values
    - *uy* = 50323

Relation between Signed & Unsigned



Maintain Same Bit Pattern

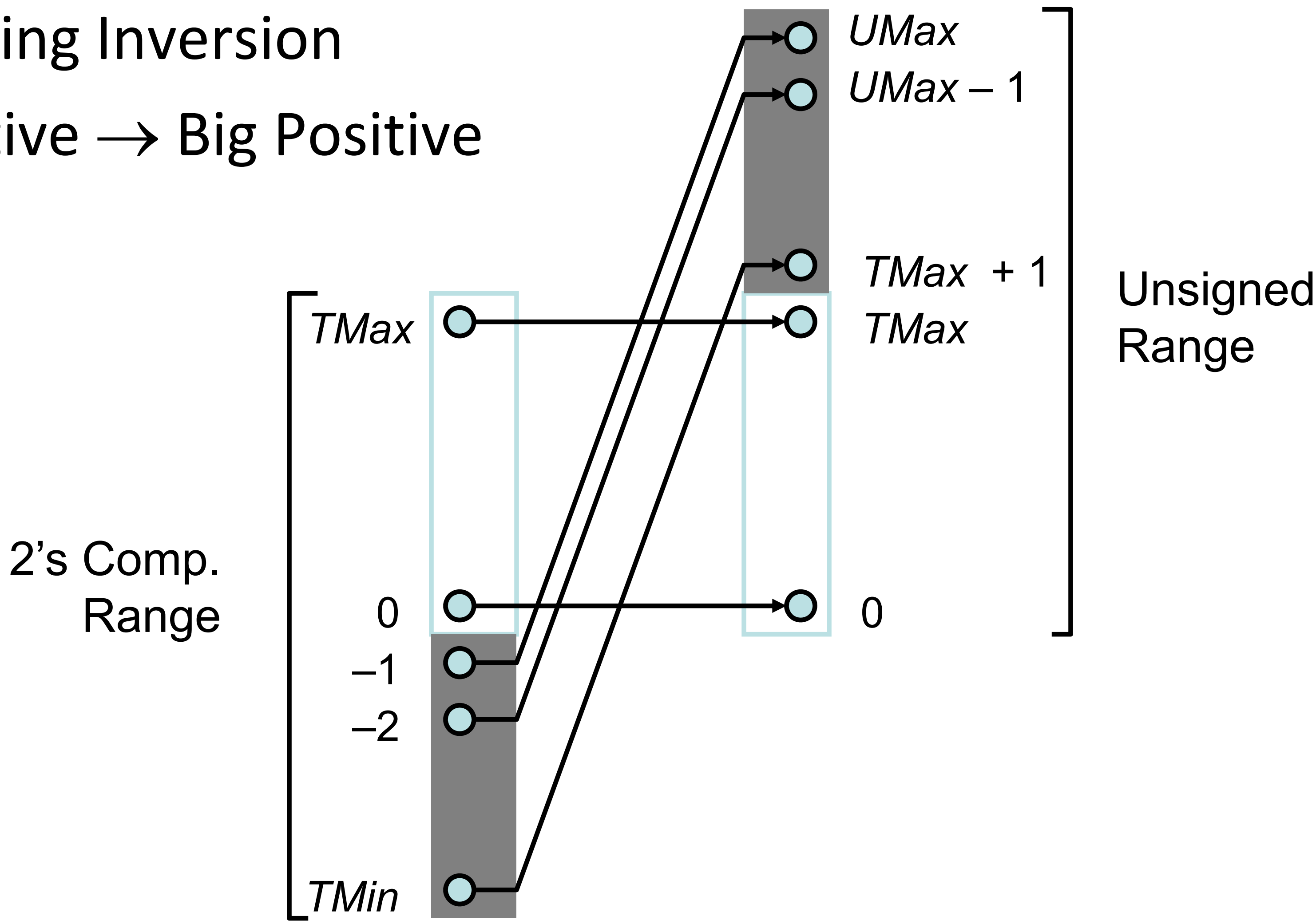


$$+2^{w-1} - -2^{w-1} = 2*2^{w-1} = 2^{w}$$

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

Illustration

- 2's Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



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**

$$uy \quad = \quad y + 2 * 32768 \quad = \quad y + 65536$$

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

```
int tx, ty;
unsigned 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 $W = 32$

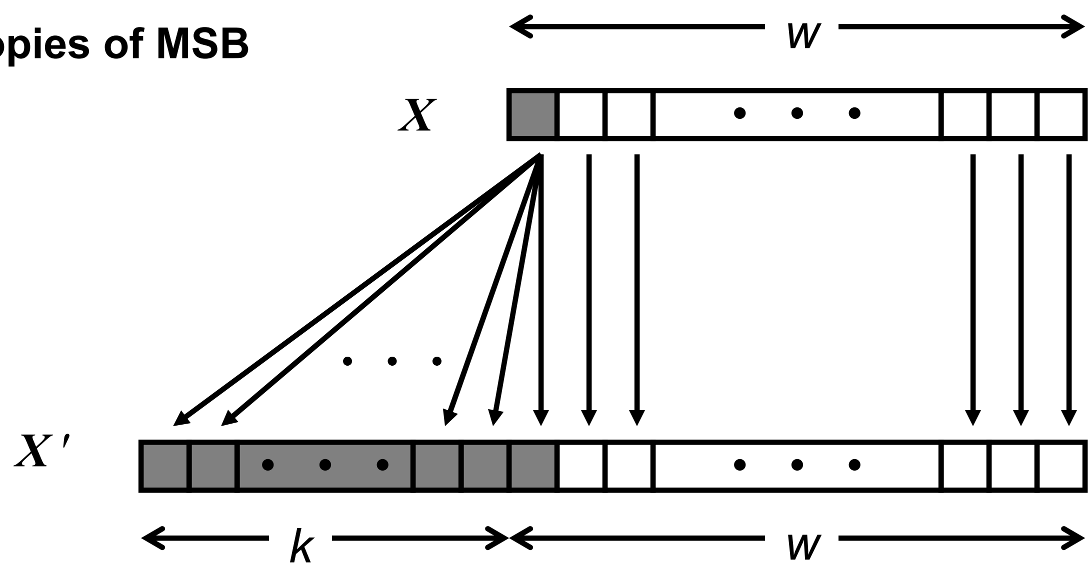
Add U while coding if you want it unsigned

convert 2s to unsigned

Constant$_1$	Constant$_2$	Relation	Evaluation
`0`	`0U`	`==`	unsigned
`-1`	`0` comparing 11111 (UMax) to 0000s	`<`	signed
`-1`	`0U`	`>`	unsigned
`2147483647`	`-2147483648`	`>`	signed
`2147483647U`	`-2147483648`	`<`	unsigned
`-1`	`-2` You don't compare the actual value, you compare the binary	`>`	signed
`(unsigned) -1`	`-2`	`>`	unsigned
`2147483647`	`2147483648U`	`<`	unsigned
`2147483647`	`(int) 2147483648U`	`>`	signed

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



Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

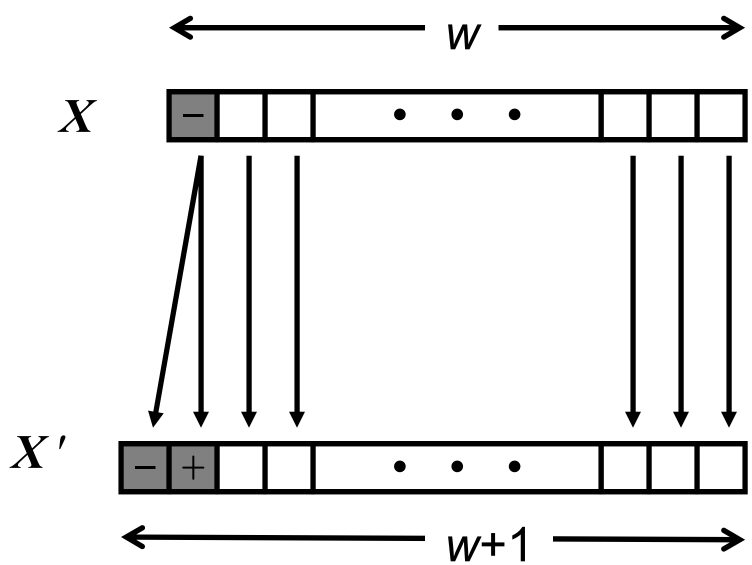
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

– 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:

$X \qquad -2^{w-1}\, x_{w-1}$

$X' \qquad -2^{w}\, x_{w-1} + 2^{w-1}\, x_{w-1} \quad = \quad -2^{w-1}\, x_{w-1}$

Beware When Using Unsigned

- *Don't* Use Just Because Number Nonzero
  - Easy to make mistakes

```
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

if cnt was -1 then it would result in a huge looping number than expected

Negating with Complement & Increment

- Claim: Following Holds for 2's Complement

  `~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
  - $\sim x + \cancel{x} + (-\cancel{x} + 1) == -\cancel{1} + (-x + \cancel{1})$
  - `~x + 1 == -x`

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 1001001`**`1`**
`y`	-15213	`C4 93`	`11000100 10010011`

0

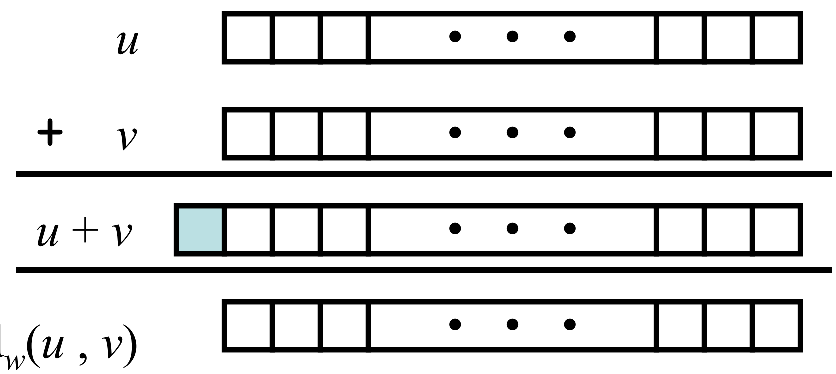
	Decimal	Hex	Binary
`0`	0	`00 00`	`00000000 00000000`
`~0`	-1	`FF FF`	`11111111 11111111`
`~0+1`	0	`00 00`	`00000000 00000000`

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

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

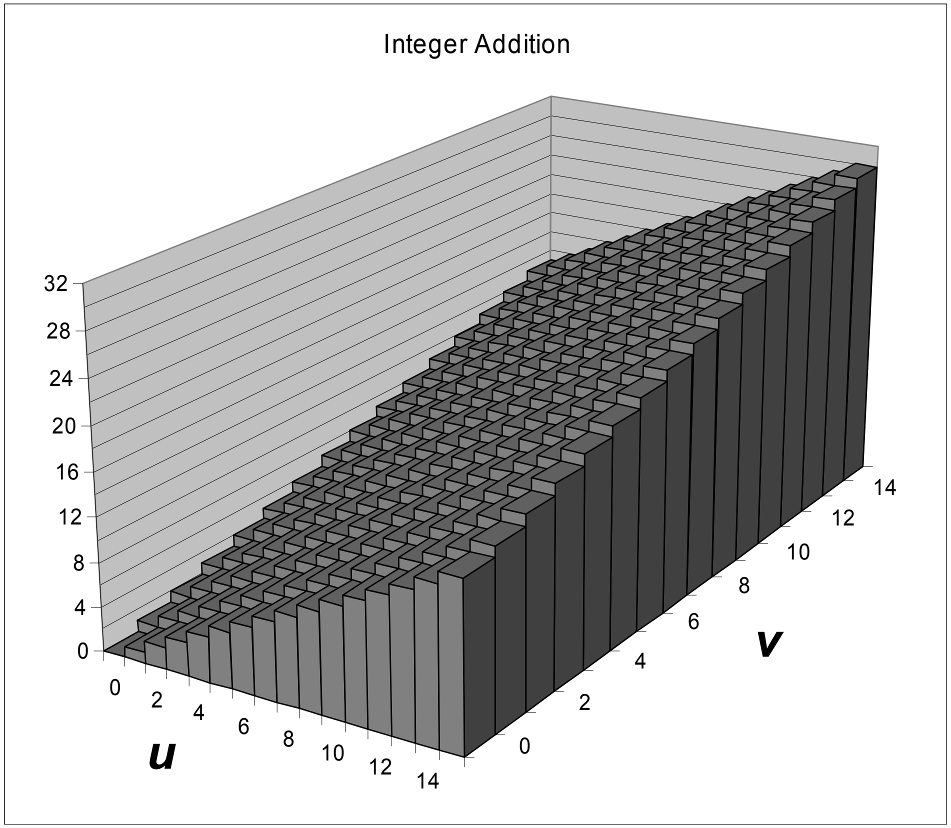
$$UAdd_w(u,v) \;=\; \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \ge 2^w \end{cases}$$

For integer overflow, unsigned to truncate

Visualizing Integer Addition

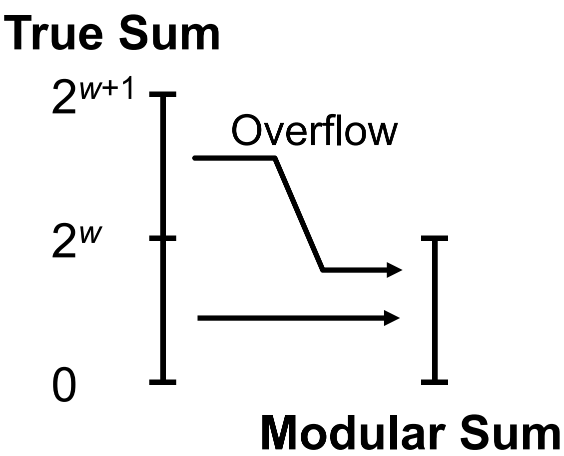
- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u , v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

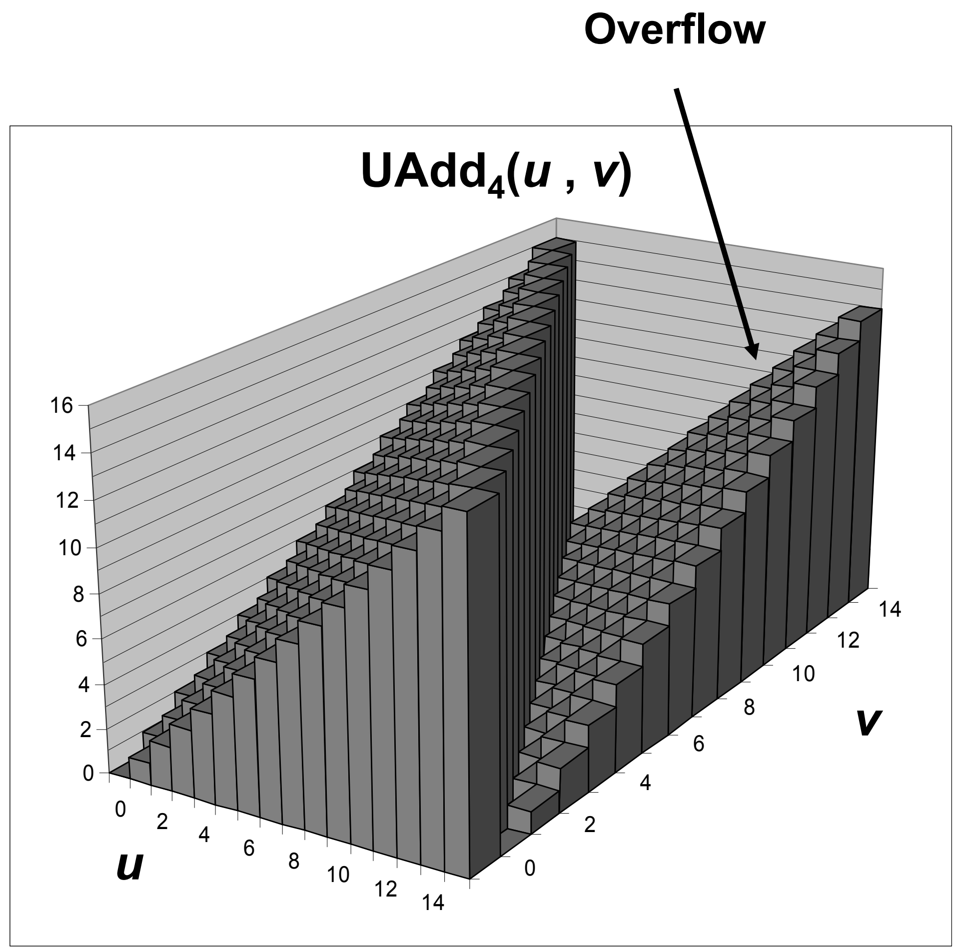
$\text{Add}_4(u , v)$



Visualizing Unsigned Addition

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





Mathematical Properties

- Modular Addition
  - Closed under addition

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

  - Commutative

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

  - Associative

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

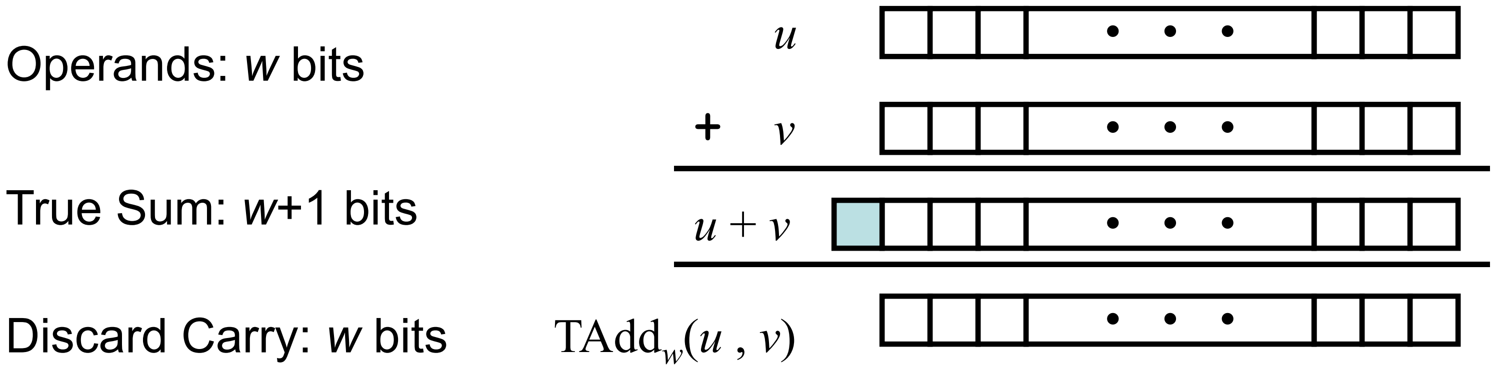
  - 0 is additive identity

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

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

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

Two's Complement Addition



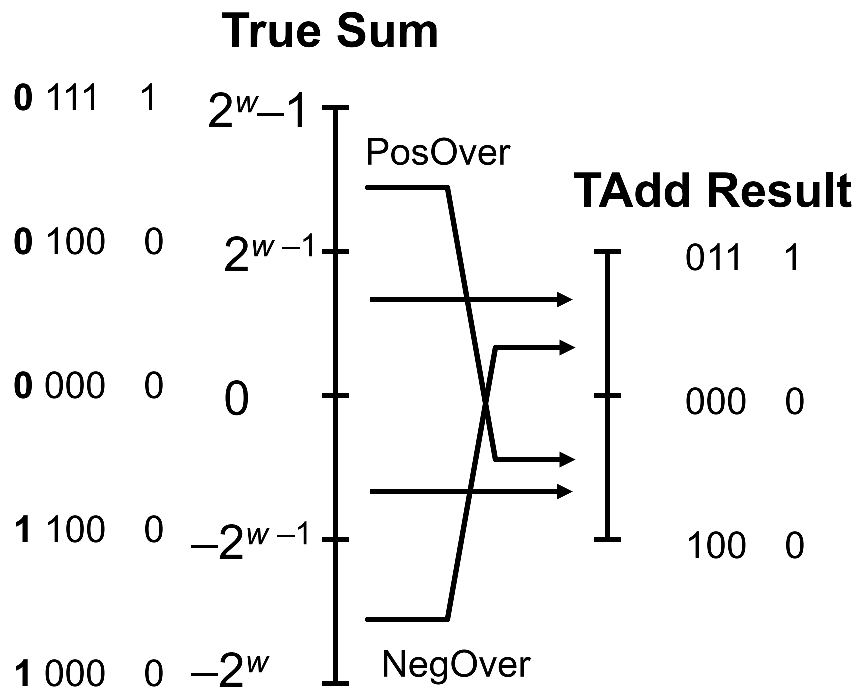
- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:

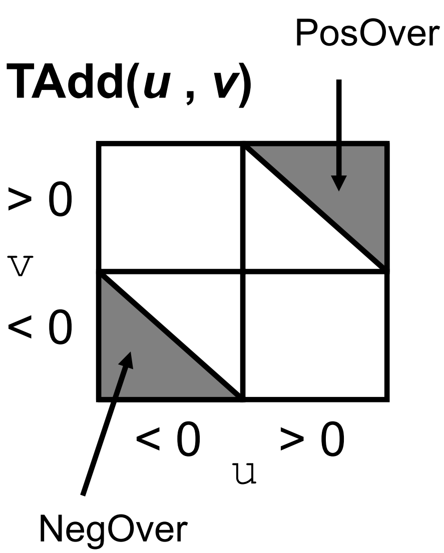
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

Characterizing TAdd

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

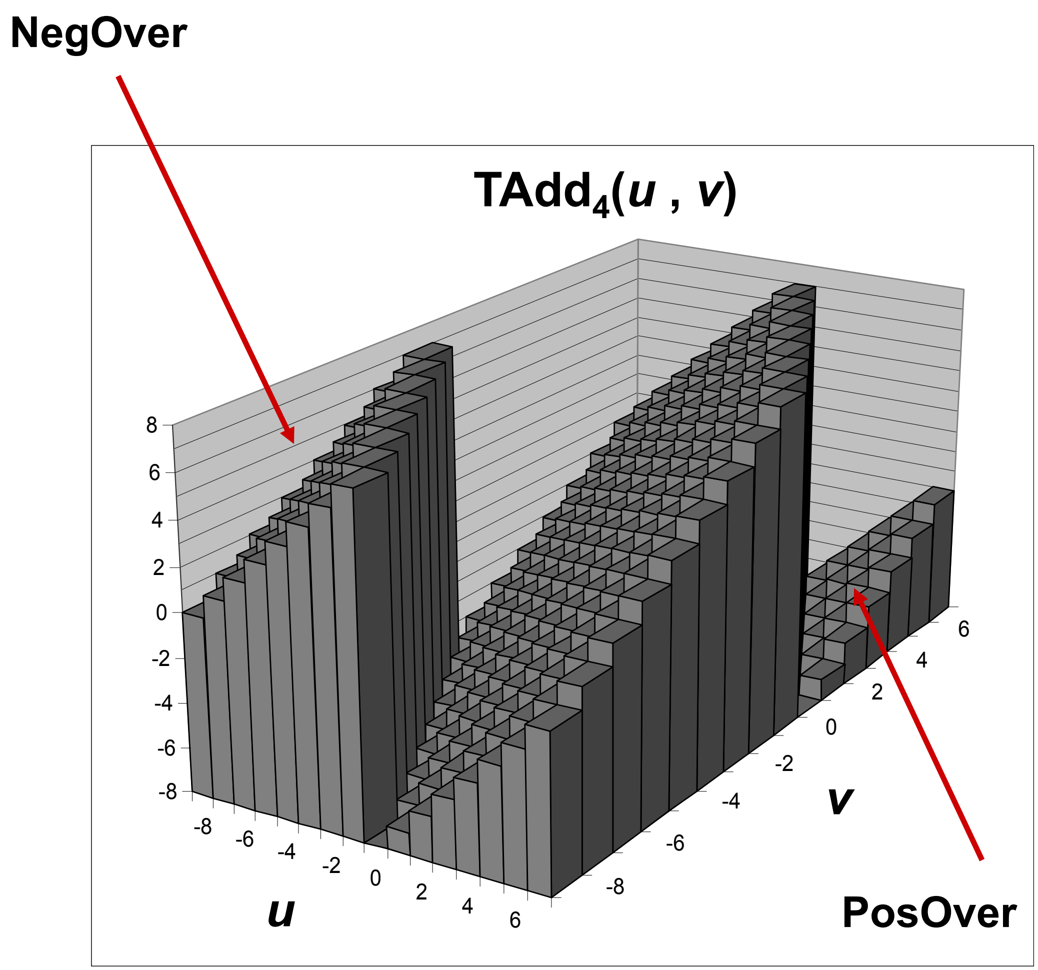




$$TAdd_w(u,v) \quad = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

Visualizing 2's Comp. Addition

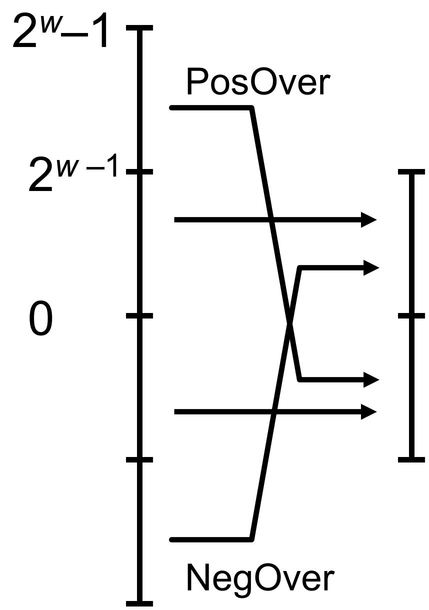
- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum ≥ 2w–1
    - Becomes negative
    - At most once
  - If sum < –2w–1
    - Becomes positive
    - At most once



Detecting 2's Comp. Overflow

- Task
  - Given **s = TAdd$_w$(u , v)**
  - Determine if s **= Add$_w$(u , v)**
  - Example

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



- Claim
  - Overflow iff either:

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

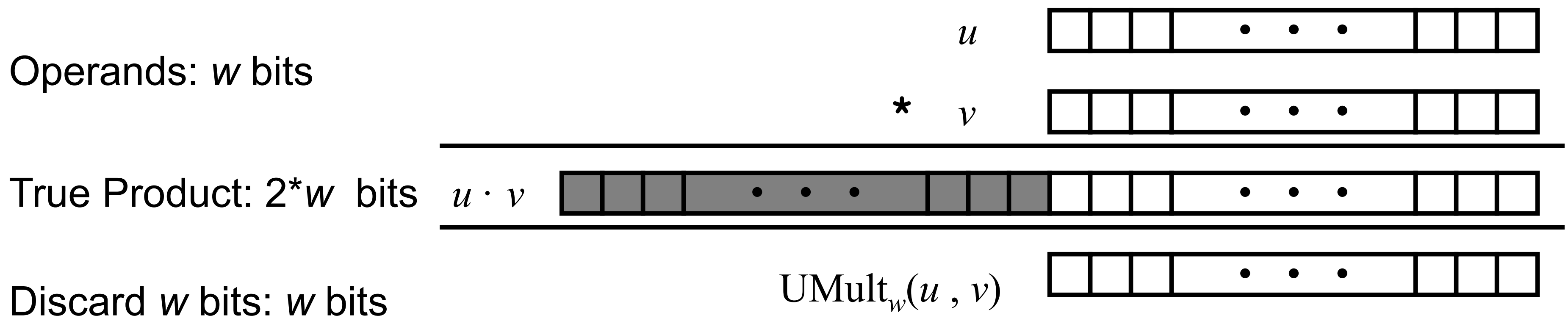
$u, v \geq 0, s < 0$ (PosOver)

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

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 (GMP, BigNum, etc.)

Unsigned Multiplication in C



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

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

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*** = **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*** = **TMult$_w$(*x, y*)**

Unsigned vs. Signed Multiplication

- Unsigned Multiplication

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

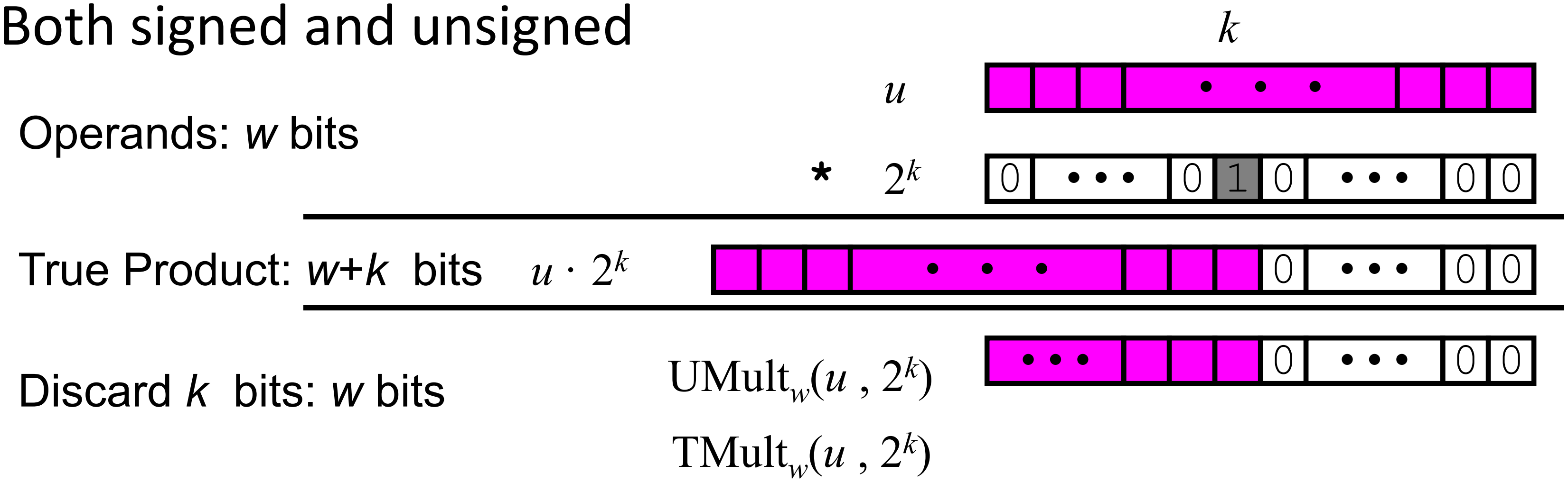
- Two's Complement Multiplication

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

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

Power-of-2 Multiply with Shift

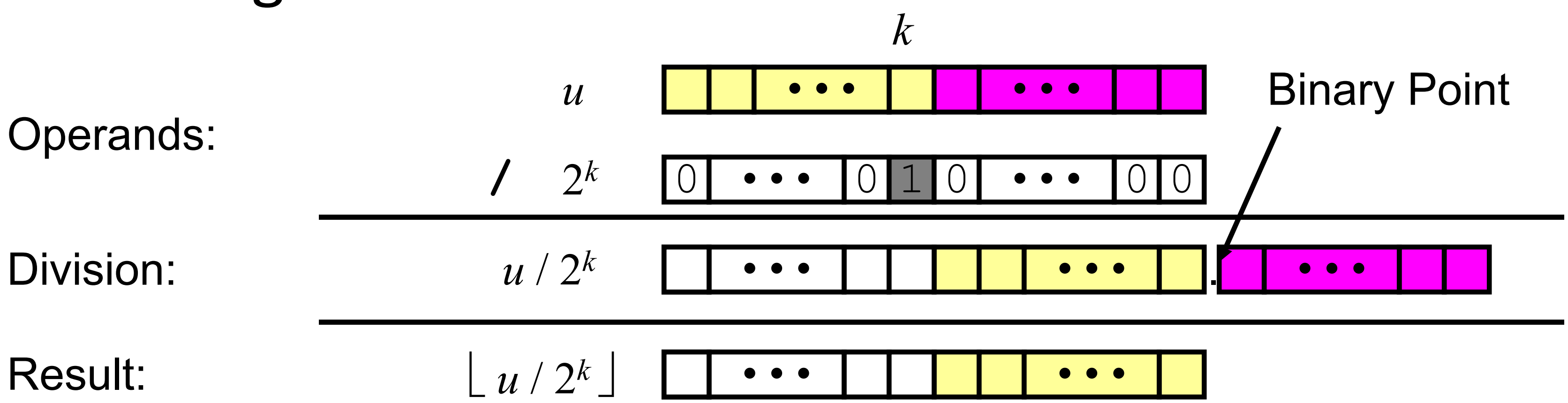
- Operation
  - `u << k` gives `u * 2`^{k}
  - Both signed and unsigned



- 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

Unsigned Power-of-2 Divide with Shift

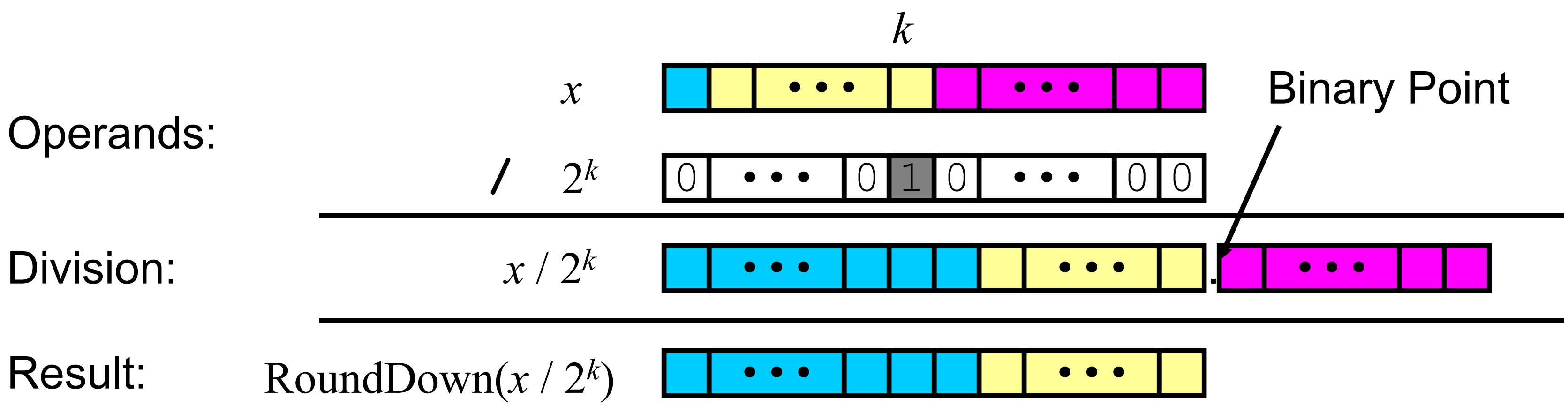
- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - Uses logical shift



	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`

Signed Power-of-2 Divide with Shift

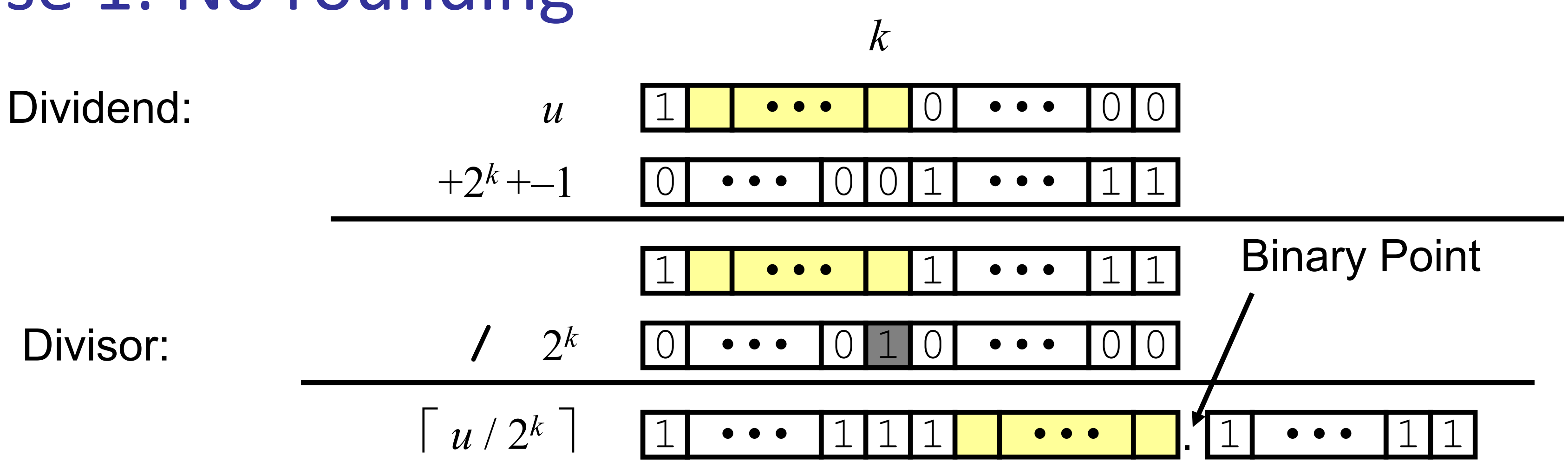
- Quotient of Signed by Power of 2
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0`



	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

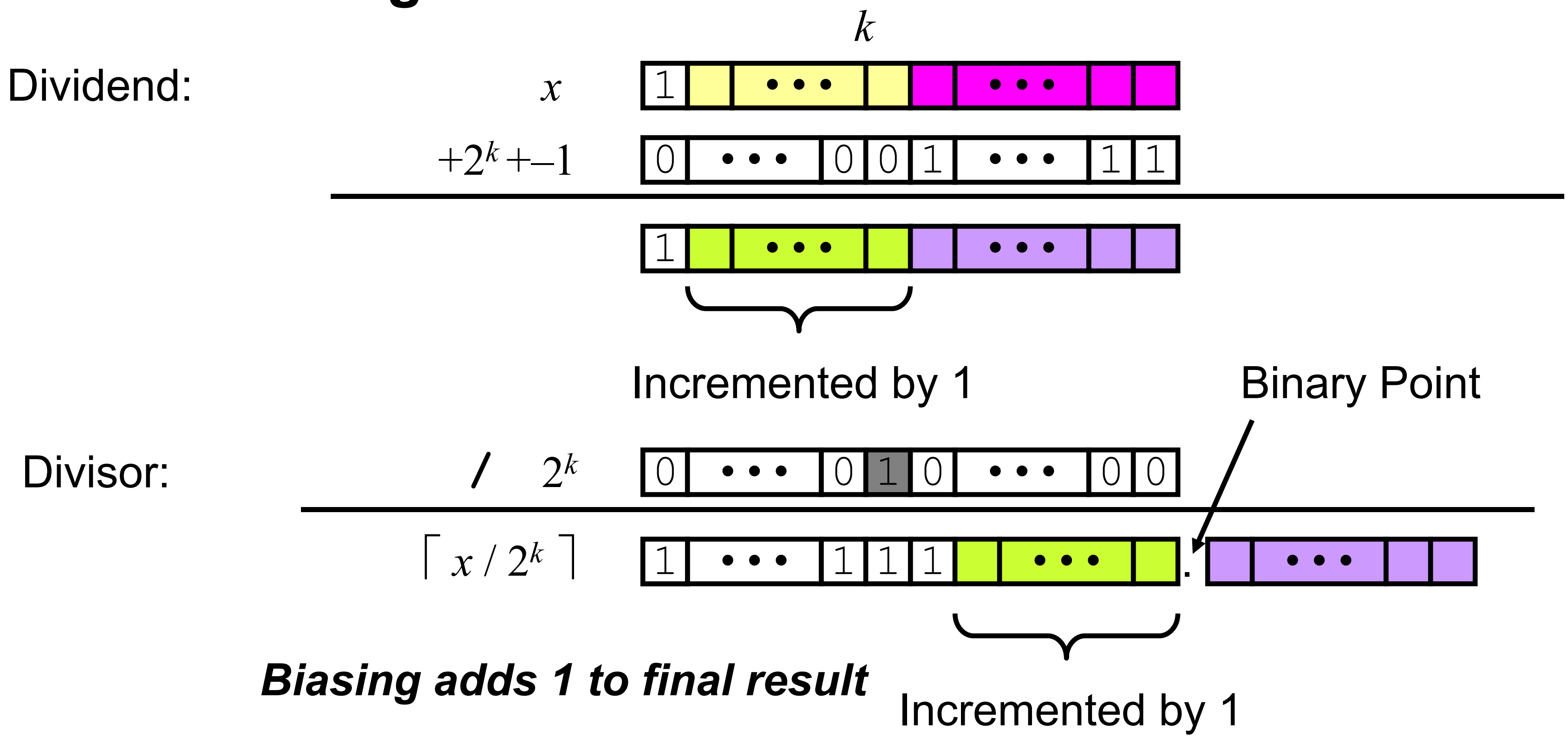
- Quotient of Negative Number by Power of 2
  - Want $\lceil$ `x / 2`$^k \rceil$ (Round Toward 0)
  - Compute as $\lfloor$ `(x+2`^{k}`-1) / 2`$^k \rfloor$
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0
- Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

0101110011011011
Want to count all the 1's and getting the sum essentially

You could iterate through everything & it with 1 and then adding it to the counter
or

Break it up into chunks or 4, sum up each one in the quads, and create quad masks of 0001
0001 0000 0001 0001 = Sum of the first 4 on the right
-Move to the next quad
Shift it and compare 2, 6, 10 to the mask and you get the sum and you should get 4 independent sums you can add together

C Puzzles

- Assume machine with 32 bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Give example where not true

Initialization

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

is

- `x < 0` ⇒ `((x*2) < 0)`
- yes — `ux >= 0`
- yes — `x & 7 == 7` ⇒ `(x<<30) < 0`
- no, negative -1 will be converted to unsigned — `ux > -1`
- no if x is 1000 — `x > y` ⇒ `-x < -y`
- no, overflow — `x * x >= 0`
- no, overflow — `x > 0 && y > 0` ⇒ `x + y > 0`
- true — `x >= 0` ⇒ `-x <= 0`
- false — `x <= 0` ⇒ `-x >= 0`

C Puzzle Answers

– Assume machine with 32 bit word size, two's comp. integers

– TMin makes a good counterexample in many cases

❑ `x < 0`	$\Rightarrow$ `((x*2) < 0)`	**False:**	*TMin*
❑ `ux >= 0`		**True:**	0 = *UMin*
❑ `x & 7 == 7`	$\Rightarrow$ `(x<<30) < 0`	**True:**	$x_1 = 1$
❑ `ux > -1`		**False:**	0
❑ `x > y`	$\Rightarrow$ `-x < -y`	**False:**	`-1`, *TMin*
❑ `x * x >= 0`		**False:**	`30426`
❑ `x > 0 && y > 0`	$\Rightarrow$ `x + y > 0`	**False:**	*TMax*, *TMax*
❑ `x >= 0`	$\Rightarrow$ `-x <= 0`	**True:**	–*TMax* < 0
❑ `x <= 0`	$\Rightarrow$ `-x >= 0`	**False:**	*TMin*