Integers

Today

- Numeric Encodings
- Programming Implications
- Basic operations
- Programming Implications

Next time

- Floats

Integers in C

- C supports several integral data types
  - Note the `unsigned` modifier
  - Also note the asymmetric ranges

C data type (32b)	Size	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short int	2	-32,768	32,767
unsigned short int	2	0	65,535
int	4	-2,147,438,648	2,147,438,647
unsigned int	4	0	4,294,967,295
long int	4	-2,147,438,648	2,147,438,647
unsigned long int	4	0	4,294,967,295
long long int	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	8	0	18,446,744,073,709,551,615

Encoding integers

Unsigned

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

(Binary To Unsigned)

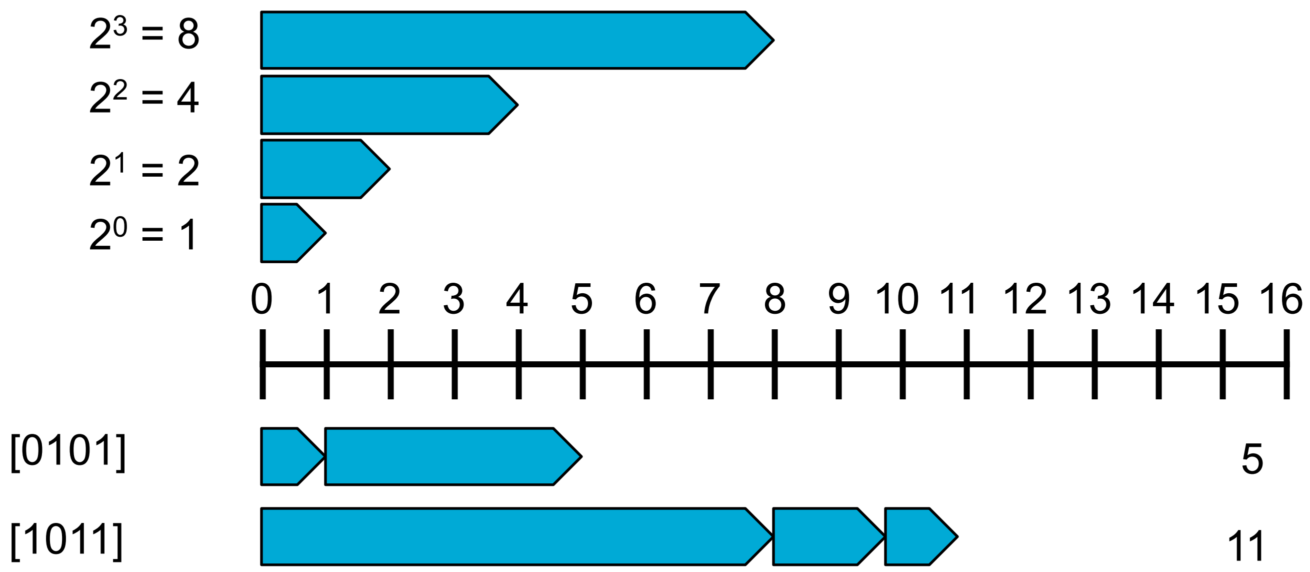
e.g. B2U ([1011]) = $1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$

– C `short` 2 bytes long

```
short int x = 15213;
```

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`

Some examples



Encoding integers

Two's Complement

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

Sign Bit

– e.g. B2T ([1011]) = $-1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -5$

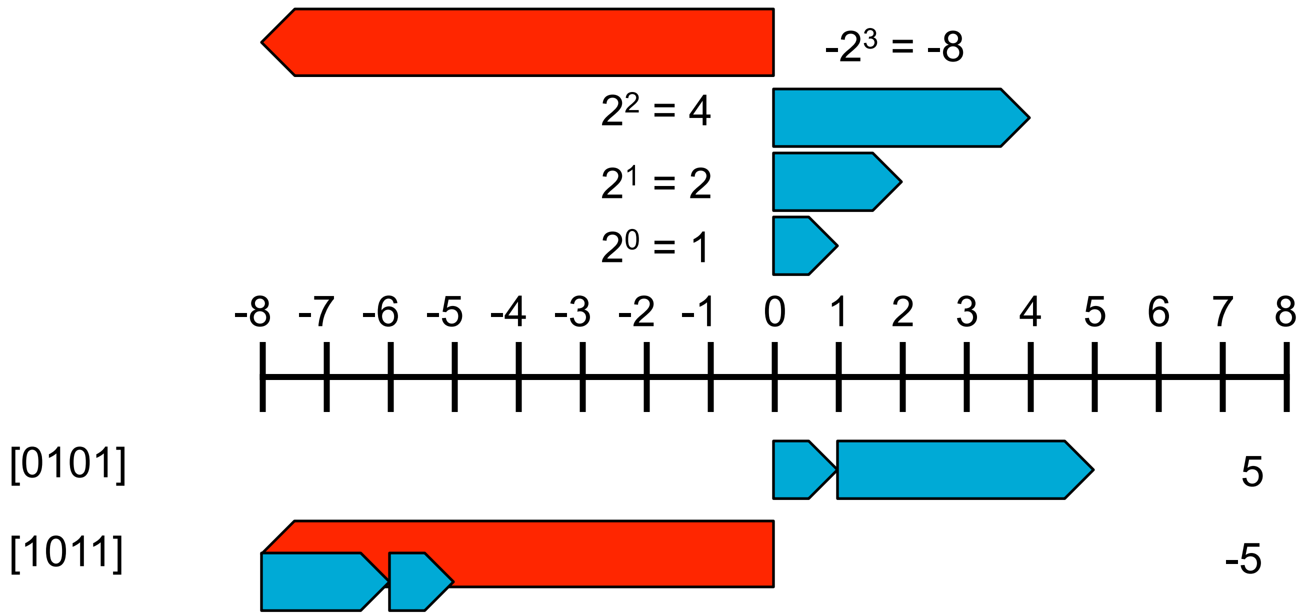
– C `short` 2 bytes long

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

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

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

Some examples



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 other 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
  - $|TMin| = |TMax| + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX, UINT_MAX`
    - `LONG_MAX, INT_MAX`
    - `LONG_MIN, INT_MIN`
  - Values platform-specific; for Java this is specified

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
  - Not based on a numeric perspective
  - No change in bit representation
  - Non-negative values unchanged
    - *ux* = 15213
  - Negative values change into (large) positive values
    - *uy* = 50323

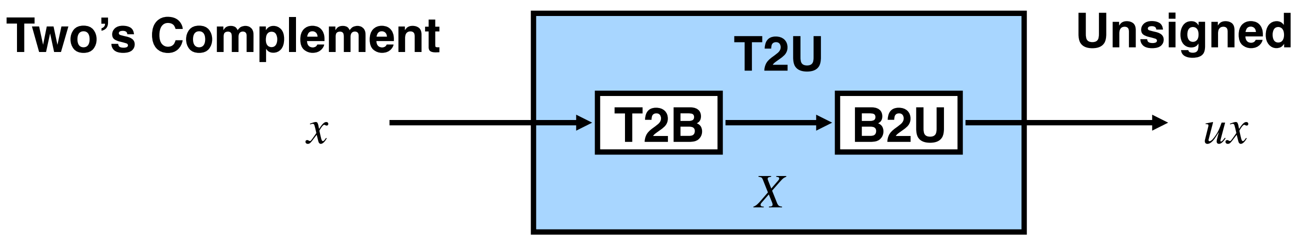
Unsigned & signed numeric values

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 (bijections)
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- Hence, well defined inverses
  - $U2B(\boldsymbol{x}) = B2U^{-1}(\boldsymbol{x})$
    - Bit pattern for unsigned integer
  - $T2B(\boldsymbol{x}) = B2T^{-1}(\boldsymbol{x})$
    - Bit pattern for two's comp integer

Relation between signed & unsigned

Casting from signed to unsigned



Maintain same bit pattern

Consider B2U and B2T equations

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \qquad B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

and a bit pattern X; compute $B2U(X) – B2T(X)$
weighted sum of for bits from 0 to w – 2 cancel each other

$$B2U(X) - B2T(X) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$$

$$B2U(X) = x_{w-1}2^w + B2T(X)$$

If we let $B2T(X) = x$

$$B2U(T2B(x)) = T2U(x) = x_{w-1}2^w + x$$

Sign bit

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

Relation between signed & unsigned

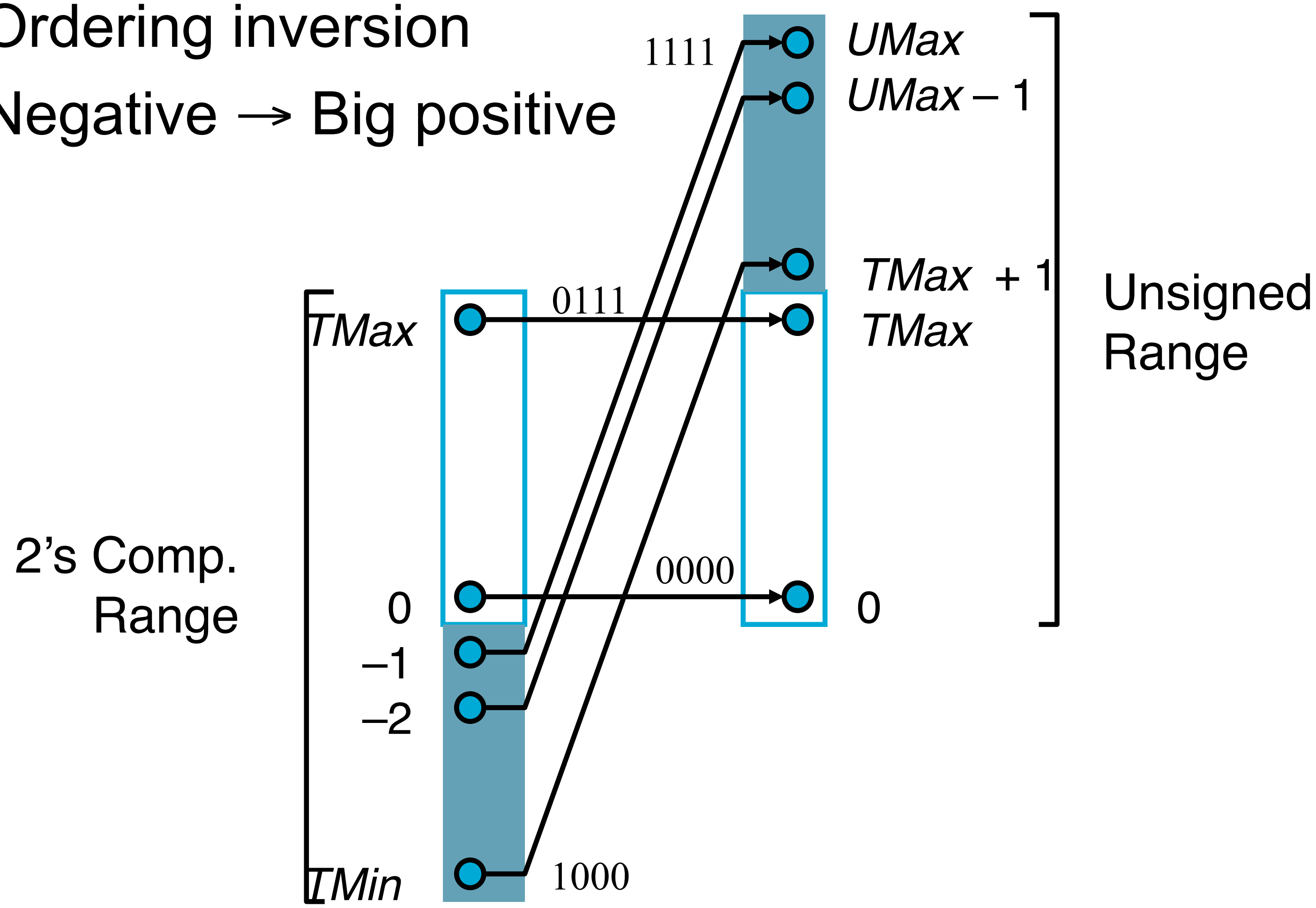
$$T2U(x) = x_{w-1} 2^w + x$$

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

***ux* = *x* + 2^{16} = *-15213* + 65536**

Conversion - graphically

- 2's Comp. → Unsigned
  – Ordering inversion
  – Negative → Big positive



Signed vs. unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U/u” as suffix

    ```
    0U, 4294967259U
    ```

- Casting
  - Explicit casting bet/ signed & unsigned same as U2T and T2U

    ```
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```

  - Implicit casting also occurs via assignments & procedure calls

    ```
    tx = ux;
    uy = ty;
    ```

Casting surprises

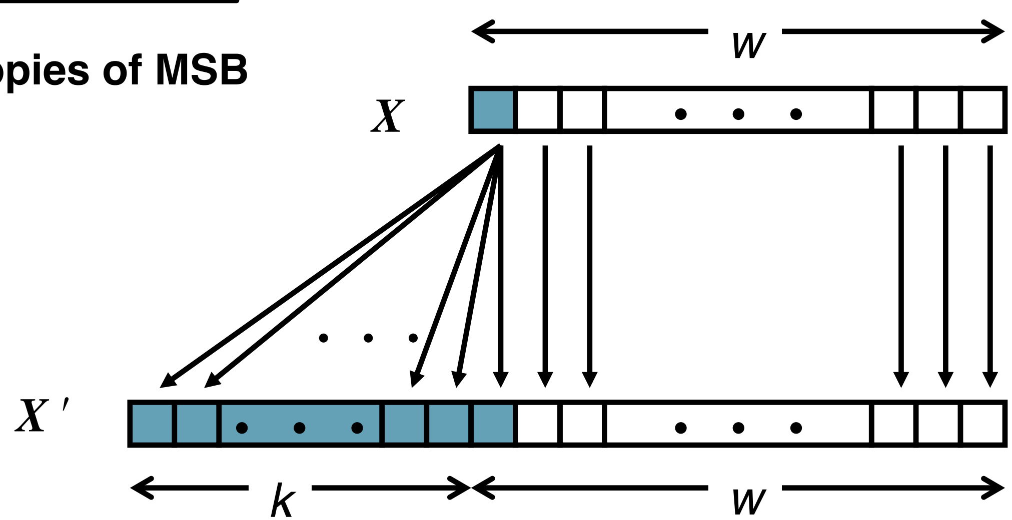
- Expression evaluation
  - If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
  - Including comparison operations <, >, ==, <=, >=
  - Examples for $W = 32$

Expression	Type	Eval
`0 == 0U`	unsigned	1
`-1 < 0`	signed	1
`-1 < 0U`	*unsigned*	*0*
`2147483647 > -2147483647-1`	signed	1
`2147483647U > -2147483647-1`	*unsigned*	*0*
`2147483647 > (int) 2147483648U`	*signed*	*1*
`-1 > -2`	signed	1
`(unsigned) -1 > -2`	unsigned	1

$2^{32-1}-1 = 2147483647$

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



Sign extension example

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

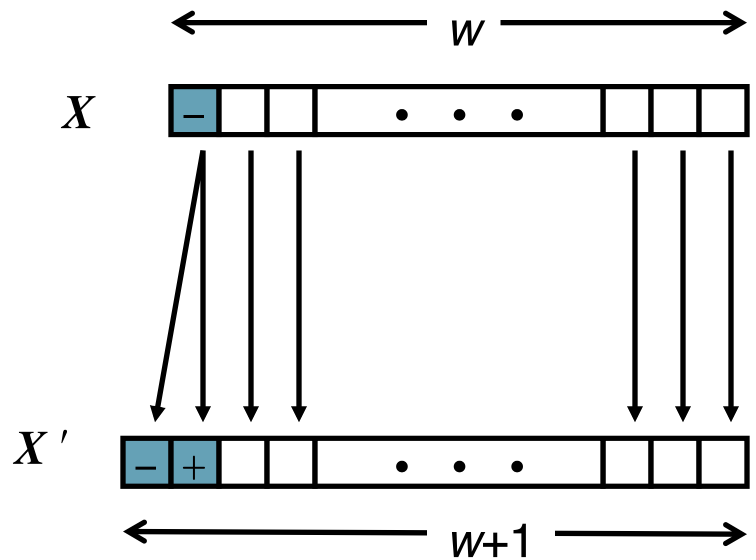
```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

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

Justification for sign extension

- Prove correctness by induction on k
  - Induction Step: extending by single bit maintains value

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



  - Key observation: $2^w - 2^{w-1} = 2^{w-1}$
  - Look at weight of upper bits:
    - X $\quad -2^{w-1}\, x_{w-1}$
    - X' $\quad -2^{w}\, x_{w-1} + 2^{w-1}\, x_{w-1} = -2^{w-1}\, x_{w-1}$

Why should I use unsigned?

- Don’t use just because number nonzero
  – C compilers on some machines generate less efficient code
  – Easy to make mistakes (e.g., casting)
  – Few languages other than C supports unsigned integers
- Do use when need extra bit’s worth of range
  – Working right up to limit of word size

Negating with complement & increment

- Claim: Following holds for 2's complement
  - $\sim x + 1 == -x$
- Complement
  - Observation: $\sim x + x == 1111\ldots11_2 == -1$

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

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`

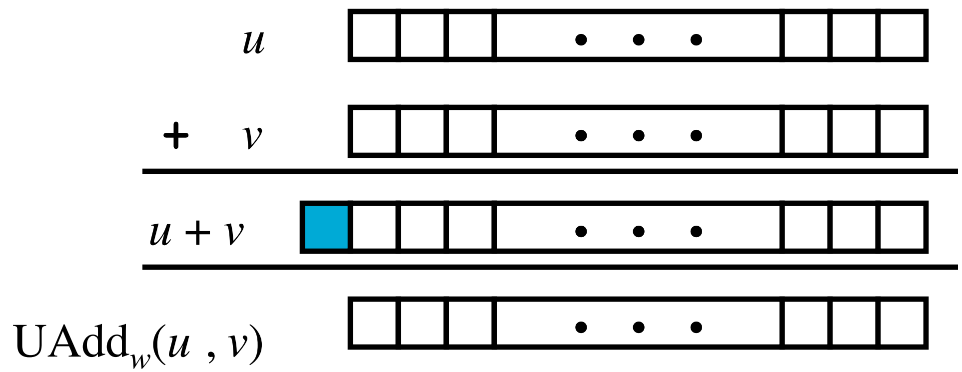
Unsigned addition

- Standard addition function
  - Ignores carry output
- Implements modular arithmetic
  - $s = \mathrm{UAdd}_w(u , v) = u + v \mod 2^w$

Operands: *w* bits

True Sum: *w*+1 bits

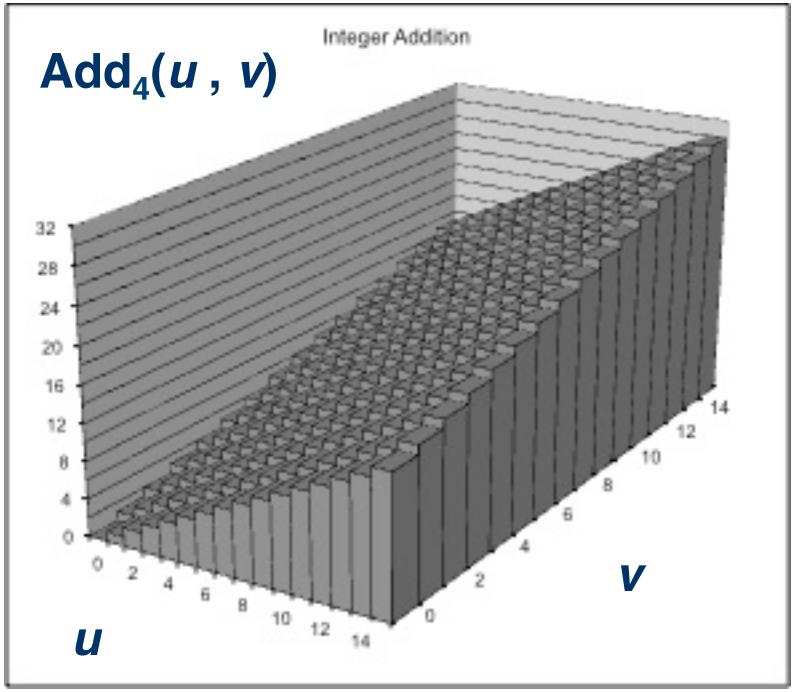
Discard Carry: *w* bits



$$UAdd_w(u,v) = \begin{cases} u+v, u+v < 2^w \\ u+w-2^w, 2^w \le x+y < 2^{w+1} \end{cases}$$

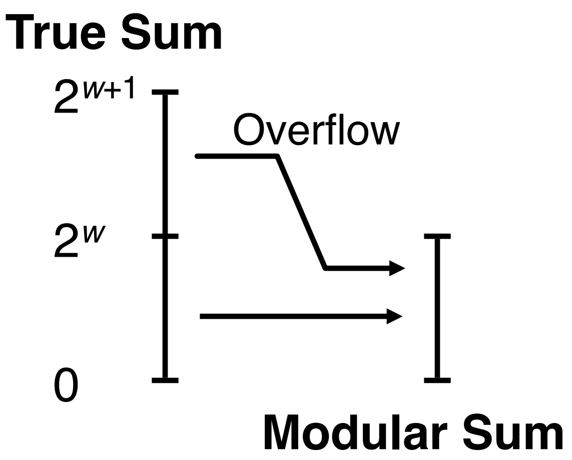
Visualizing integer addition

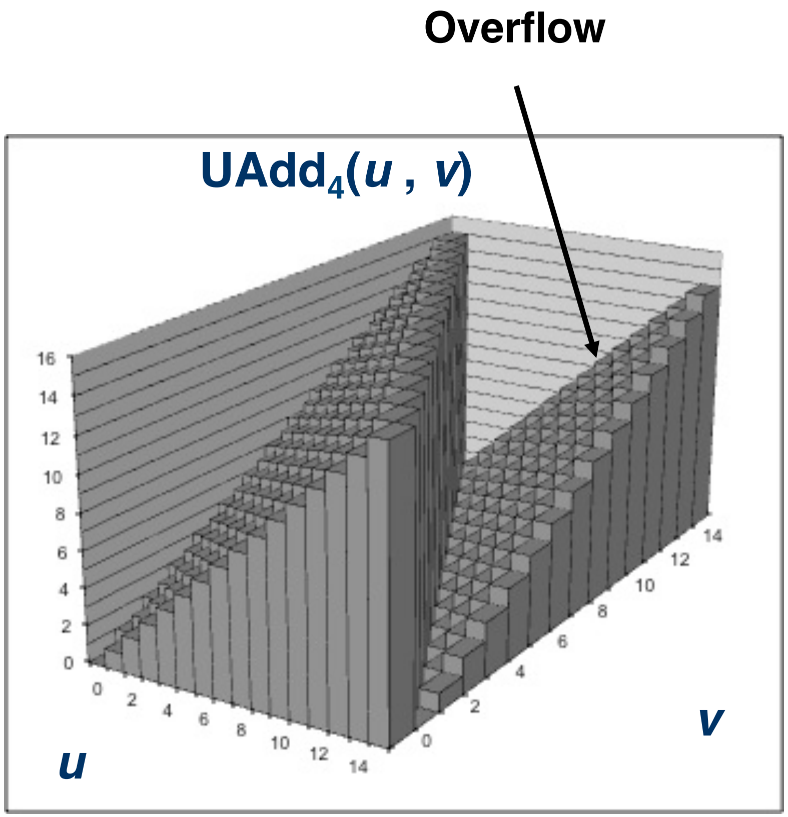
- Integer addition
  - 4-bit integers u, v
  - Compute true sum Add4(u , v)
  - Values increase linearly with u and v
  - Forms planar surface



Visualizing unsigned addition

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





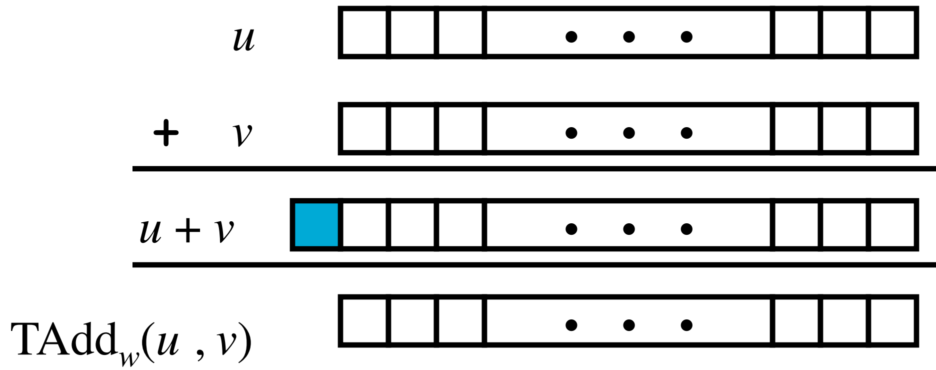
Two’s complement addition

- 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

Operands: *w* bits

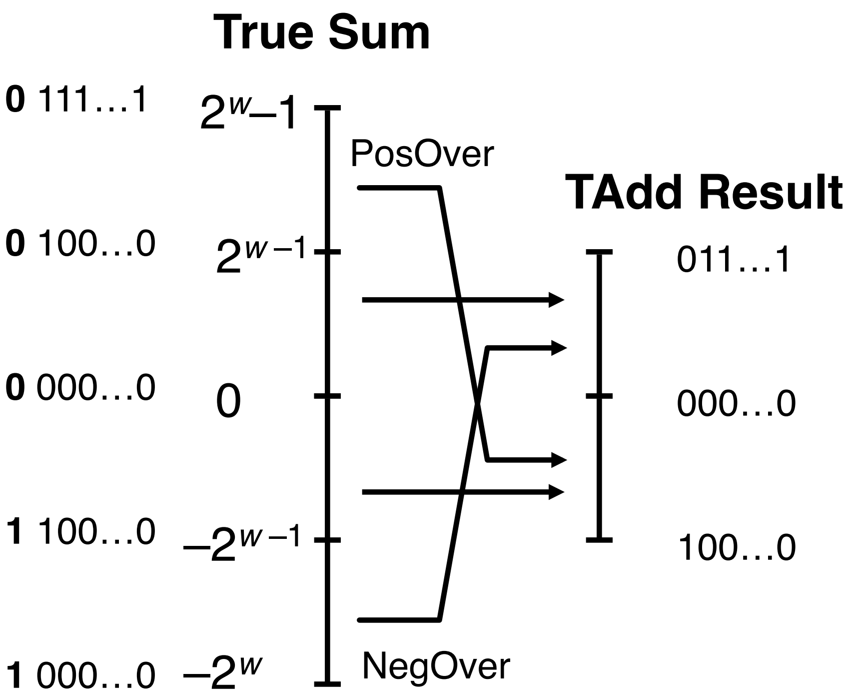
True Sum: *w*+1 bits

Discard Carry: *w* bits



Characterizing TAdd

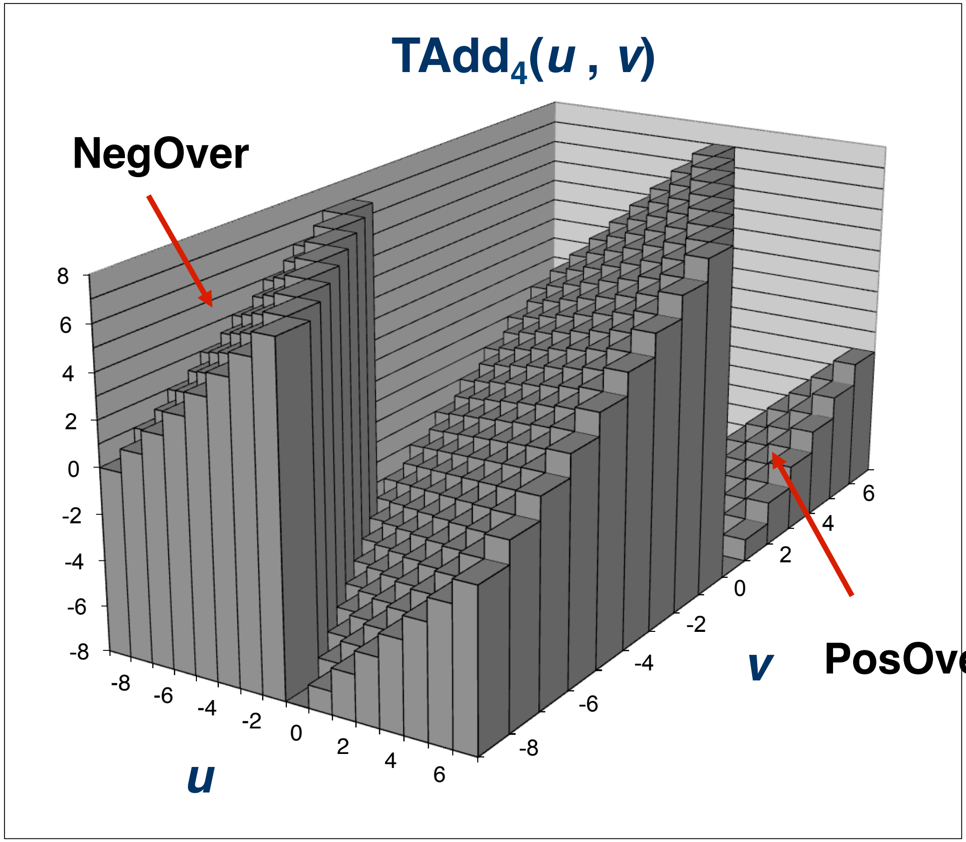
- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



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

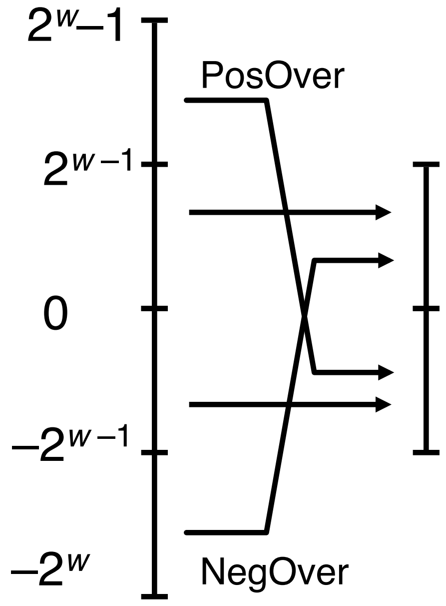
Visualizing 2's comp. addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Detecting 2's comp. overflow

- Task
  - Given s = TAddw(u , v)
  - Determine if s = Addw(u , v)
  - Example
  - int s, u, v;
  - s = u + v;



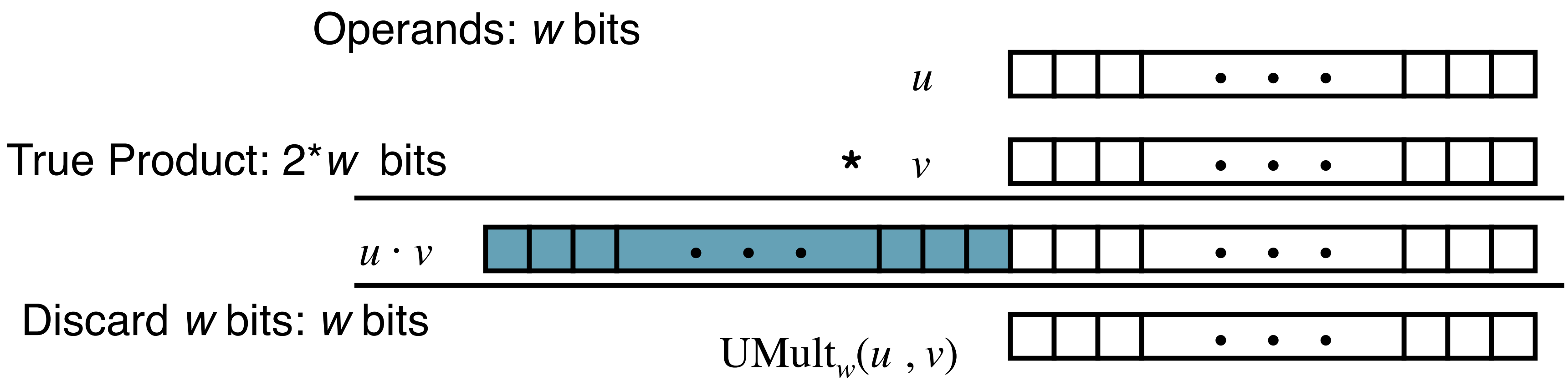
- Claim
  - Overflow iff either:
    - u, v < 0, s ≥ 0 (NegOver)
    - u, v ≥ 0, s < 0 (PosOver)

```
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: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to 2w bits to represent
  - 2's complement min: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to 2^{w-1} bits
  - 2's complement max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to 2w bits
- 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

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

Security vulnerability in XDR

```
/*
 * Illustration of code vulnerability similar to
 * that found in Sun's XDR library
 */
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
  /*
   * Allocate buffer for ele_cnt objects, each
   * of ele_size bytes and copy from ele_src
   */
   void *result = malloc(ele_cnt * ele_size);
   if (result == NULL) return NULL; /* malloc failed */
   void *next = result;
   int i;
   for (i = 0; i < ele_cnt; i++) {
      memcpy(next, ele_src[i], ele_size); /* Copy object i to dest */
      next += else_size; /* Move pointer to next */
   }
   return result;
}
```

Call it with ele_cnt = 2^{20}+1 and ele_size = 2^{12}

Then this overflows, allocating only 4096B

… and this for loop will write over the allocated buffer, corrupting other data structures!

US-CERT Vulnerability note
http://www.kb.cert.org/vuls/id/192995

Two’s complement multiplication

- 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 = TMultw(x, y)
- Relation
  - Signed multiplication gives same bit-level result as unsigned
  - up == (unsigned) p

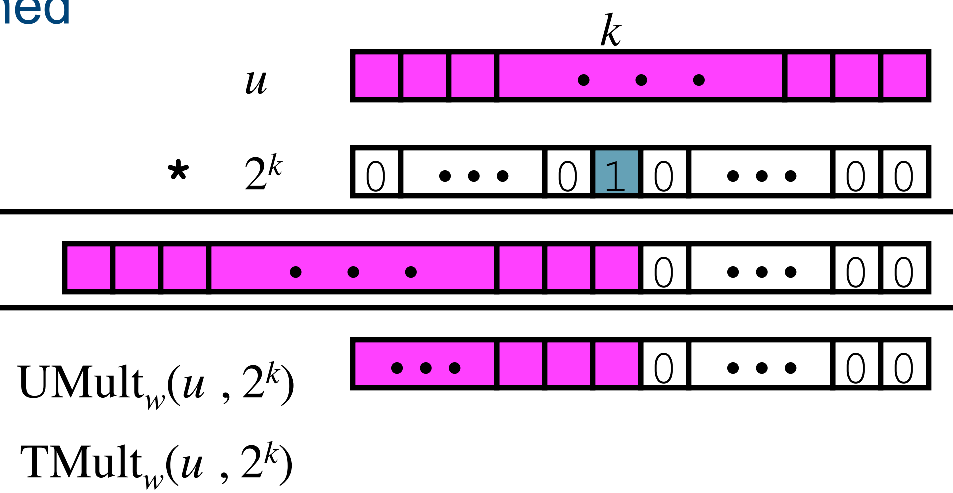
Power-of-2 multiply with shift

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

Operands: *w* bits

True Product: *w+k* bits

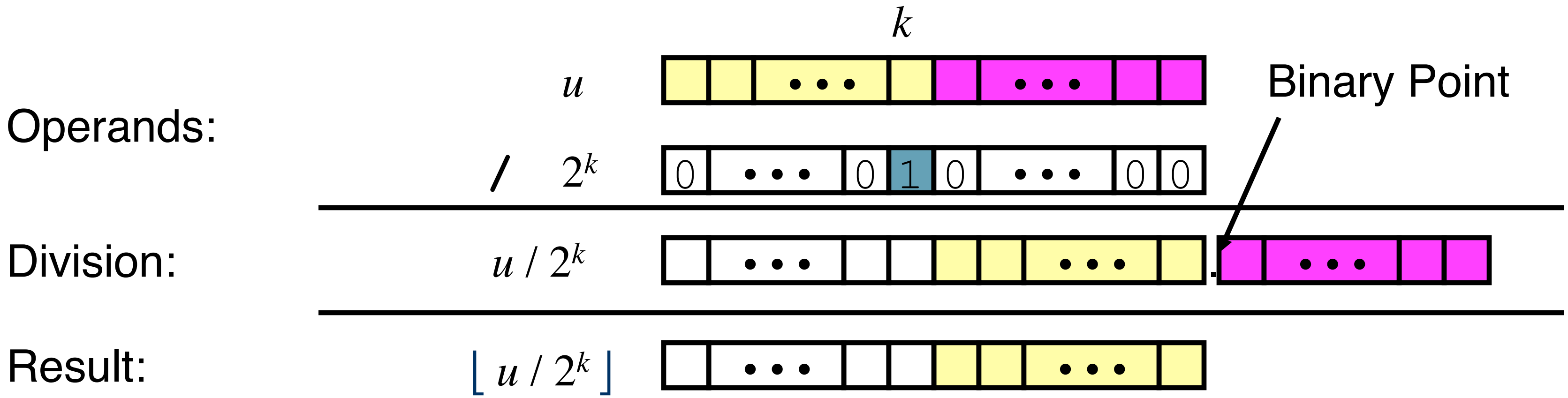
Discard *k* bits: *w* bits



- Examples
  - `u << 3 == u* 8`
  - `u << 5 - u << 3 = u * 24`
  - Most machines >> and + much faster than * (1 to 12 cycles)
    - Compiler generates this code automatically

Unsigned power-of-2 divide with shift

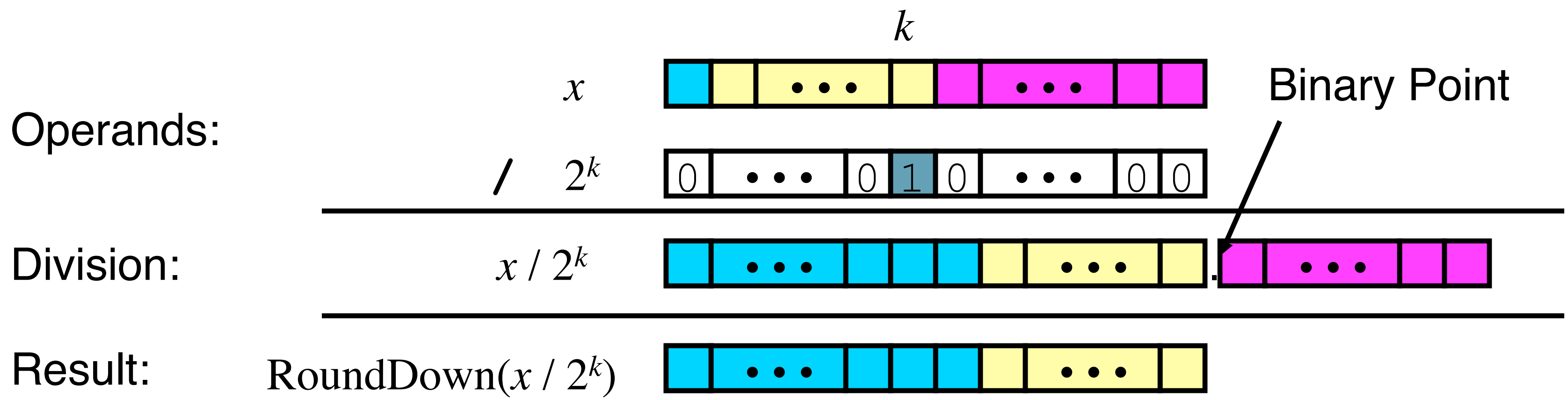
- Quotient of unsigned by power of 2
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`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 >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `x < 0`



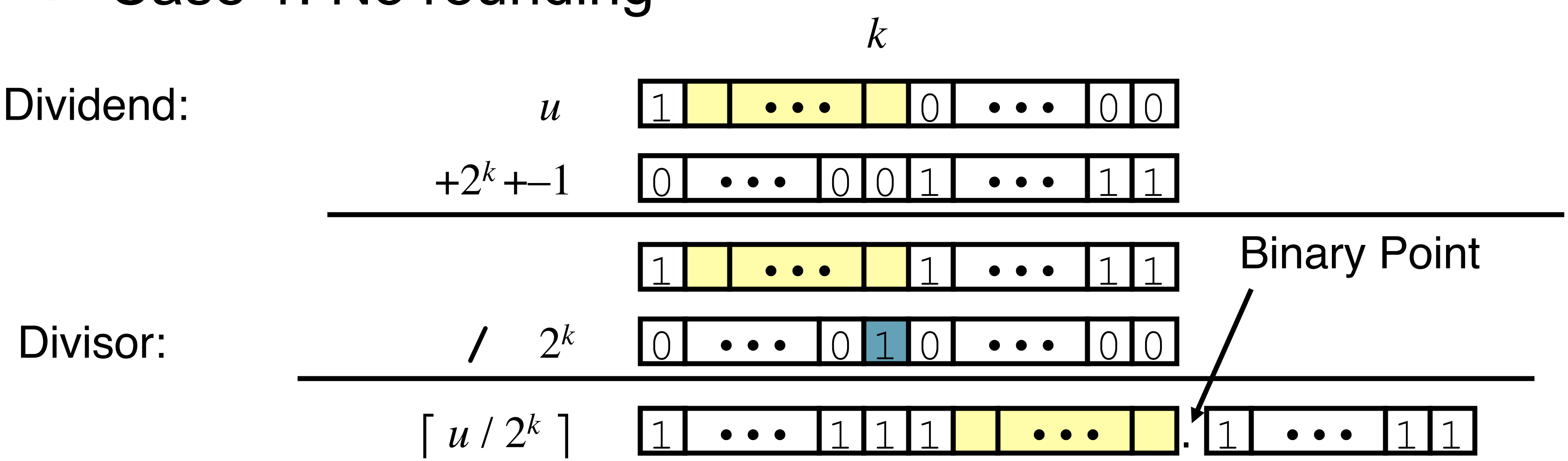
	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	**1**`1100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	**1111**`1100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	**11111111** `11000100`

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<0 ? (x + (1<<k)-1) : x) >> k`
    - Biases dividend toward 0

$\lceil$`x/y`$\rceil$ `=`$\lfloor$`(x+y-1)/y`$\rfloor$

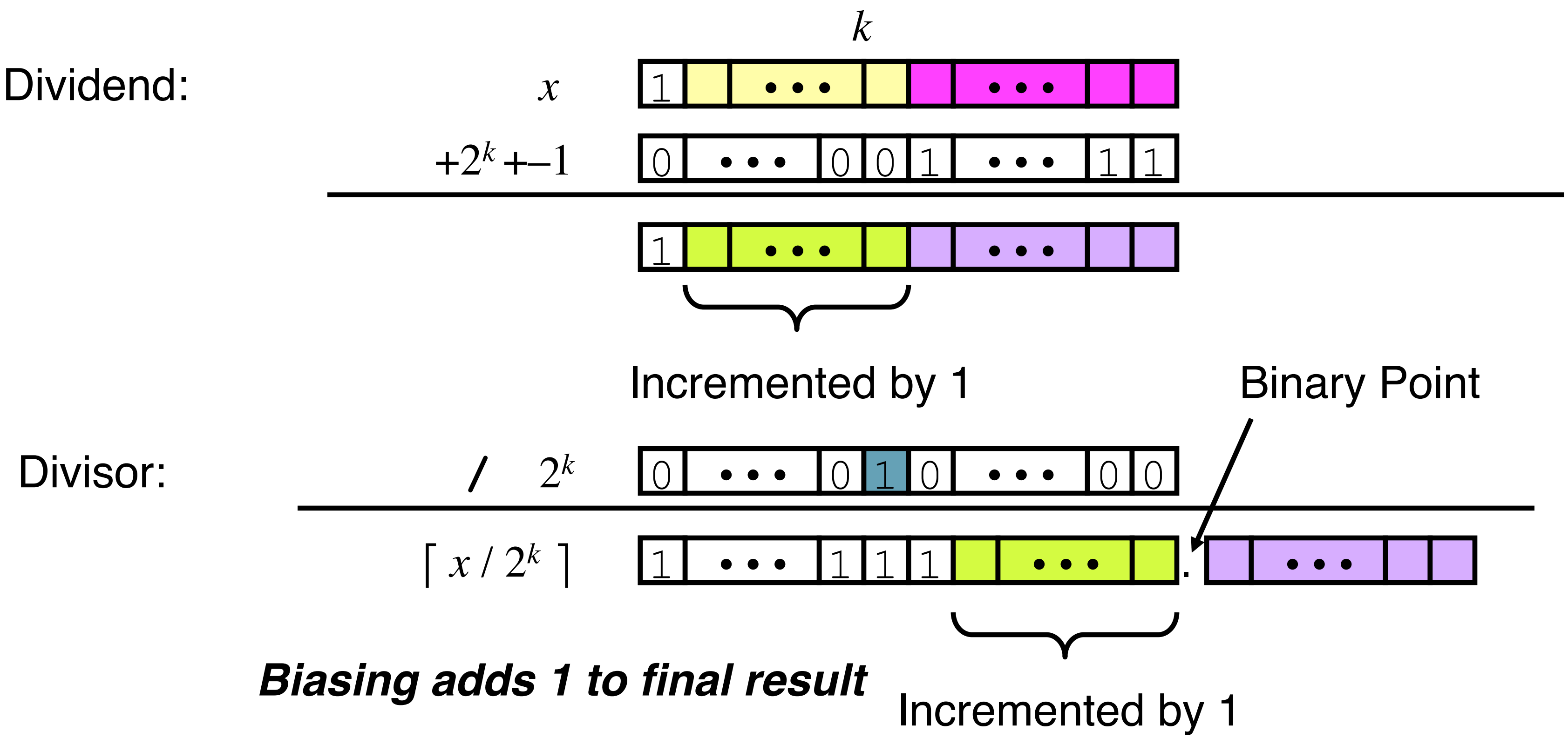
- Case 1: No rounding



Biasing has no effect

Correct power-of-2 divide (Cont.)

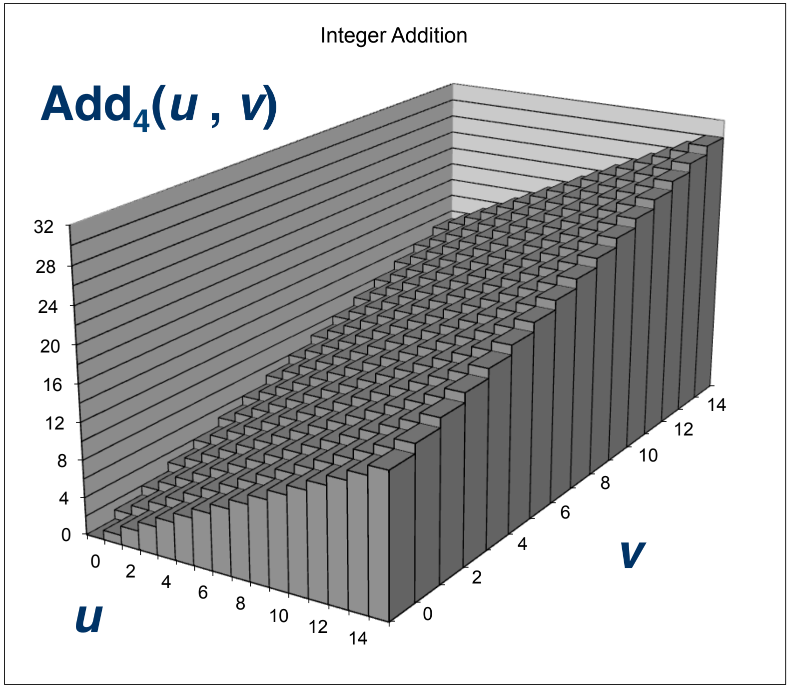
Case 2: Rounding



Biasing adds 1 to final result

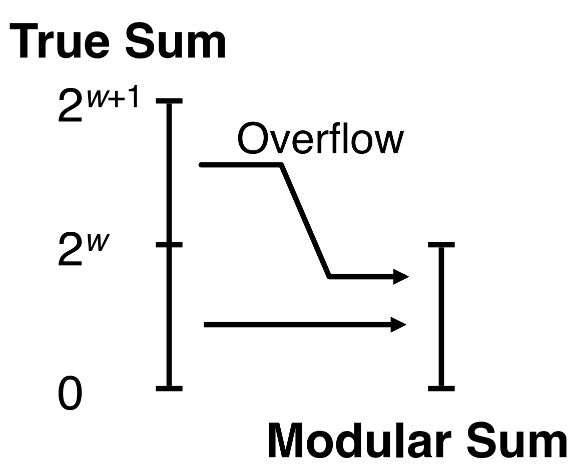
Visualizing integer addition

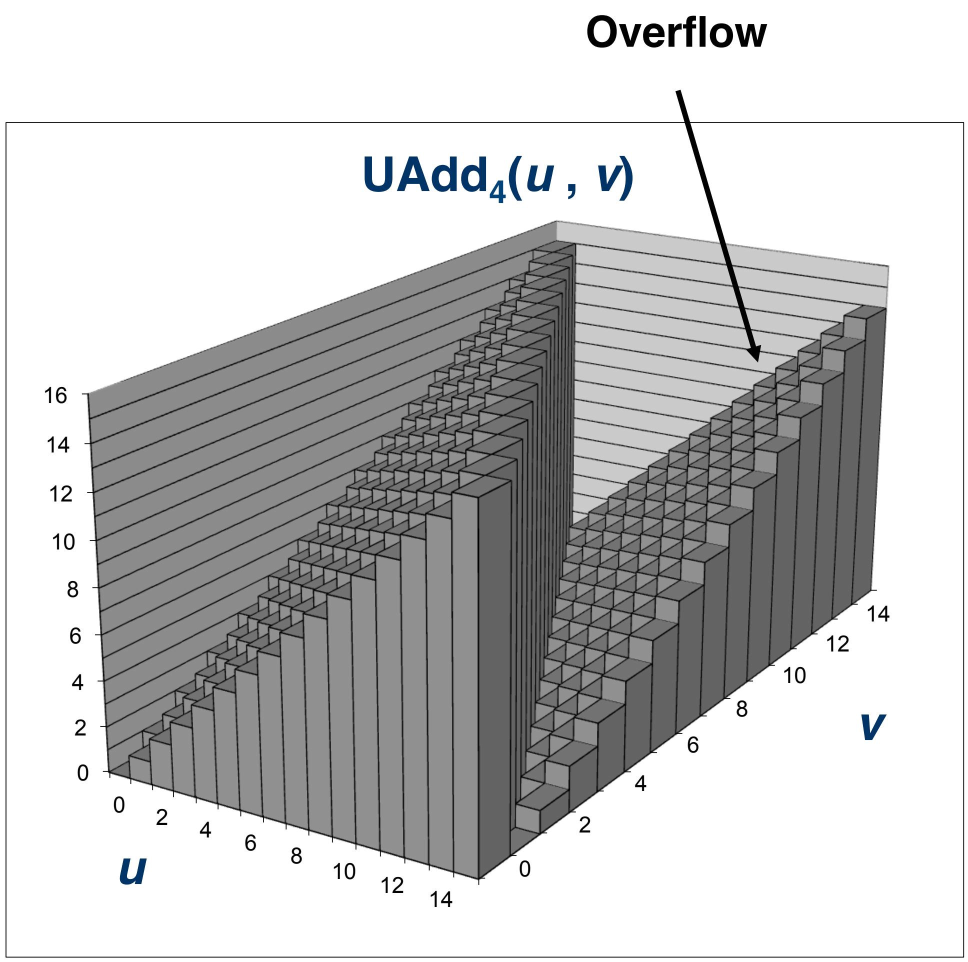
- Integer addition
  - 4-bit integers u, v
  - Compute true sum Add4(u , v)
  - Values increase linearly with u and v
  - Forms planar surface



Visualizing unsigned addition

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





Encoding example

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

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

C Puzzles

- Taken from old exams
- 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`

C Puzzle answers

- Assume machine with 32 bit word size, two's comp. 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:** `30426`
`x > 0 && y > 0`	⇒	`x + y > 0`	**False:** *TMax*, *TMax*
`x >= 0`	⇒	`-x <= 0`	**True:** $-TMax < 0$
`x <= 0`	⇒	`-x >= 0`	**False:** *TMin*