Today's Topics

- Representation of integers: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting

Encoding Integers

- C `short` 2 bytes long

Unsigned

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

Two's Complement

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

Sign Bit

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

short int x = 12345;
short int y = -12345;

	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
y	-12345	CF C7	11001111 11000111

-x → ~x + 1

Encoding Example (Cont.)

x = 12345: 00110000 00111001
y = -12345: 11001111 11000111

Weight	12345		-12345	
1	1	1	1	1
2	0	0	1	2
4	0	0	1	4
8	1	8	0	0
16	1	16	0	0
32	1	32	0	0
64	0	0	1	64
128	0	0	1	128
256	0	0	1	256
512	0	0	1	512
1024	0	0	1	1024
2048	0	0	1	2048
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**12345**		**-12345**

University of Washington

Shift Operations

- **Left shift: `x << y`**
  - Shift bit-vector `x` left by `y` positions
    - Throw away extra bits on left
    - Fill with 0s on right
  - Multiply by 2**y
- **Right shift: `x >> y`**
  - Shift bit-vector `x` right by `y` positions
    - Throw away extra bits on right
  - Logical shift (for unsigned)
    - Fill with 0s on left
  - Arithmetic shift (for signed)
    - Replicate most significant bit on right
    - Maintain sign of x
  - Divide by 2**y

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**

Undefined behavior when
y < 0 or y ≥ word_size

Using Shifts and Masks

- **Extract 2nd most significant byte of an integer**
  - First shift: x >> (2 * 8)
  - Then mask: (x >> 16) & 0xFF

x	**01100001 $\boxed{01100010}$ 01100011 01100100**
x >> 16	**00000000 00000000 01100001 $\boxed{01100010}$**
(x >> 16) & 0xFF	***00000000 00000000 00000000 11111111*** **00000000 00000000 00000000 01100010**

- **Extracting the sign bit**
  - (x >> 31) & 1 - need the "& 1" to clear out all other bits except LSB
- **Conditionals as Boolean expressions (x is 0 or 1)**
  - if (x) a=y else a=z; which is the same as a = x ? y: z;
  - Can be re-written as: a = ((x << 31) >> 31) & y + (!x << 31) >> 31) & z

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

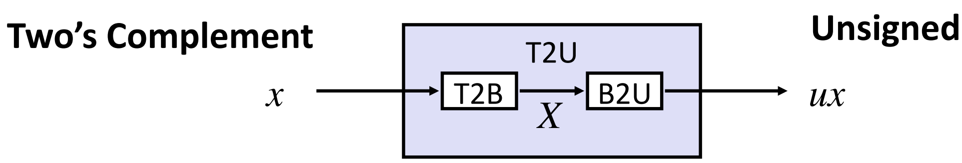
- **Observations**
  - \| *TMin* \| = *TMax* + 1
    - Asymmetric range
  - *UMax* = 2 * *TMax* + 1
- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

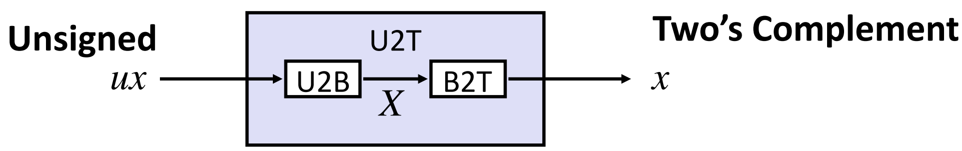
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **Can Invert Mappings**
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two’s comp integer

Mapping Between Signed & Unsigned



Maintain Same Bit Pattern

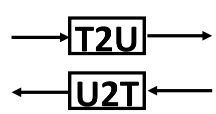


Maintain Same Bit Pattern

- **Keep bit representations and reinterpret**

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15



Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

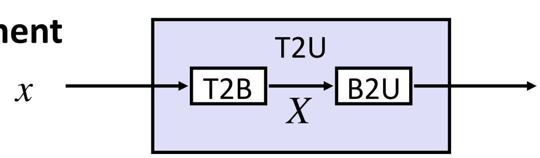
=

+16

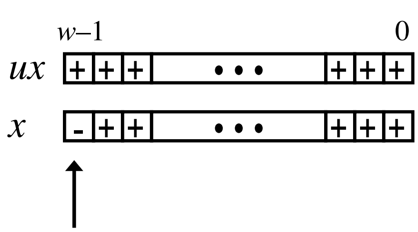
Relation between Signed & Unsigned

Two's Complement

Unsigned



Maintain Same Bit Pattern

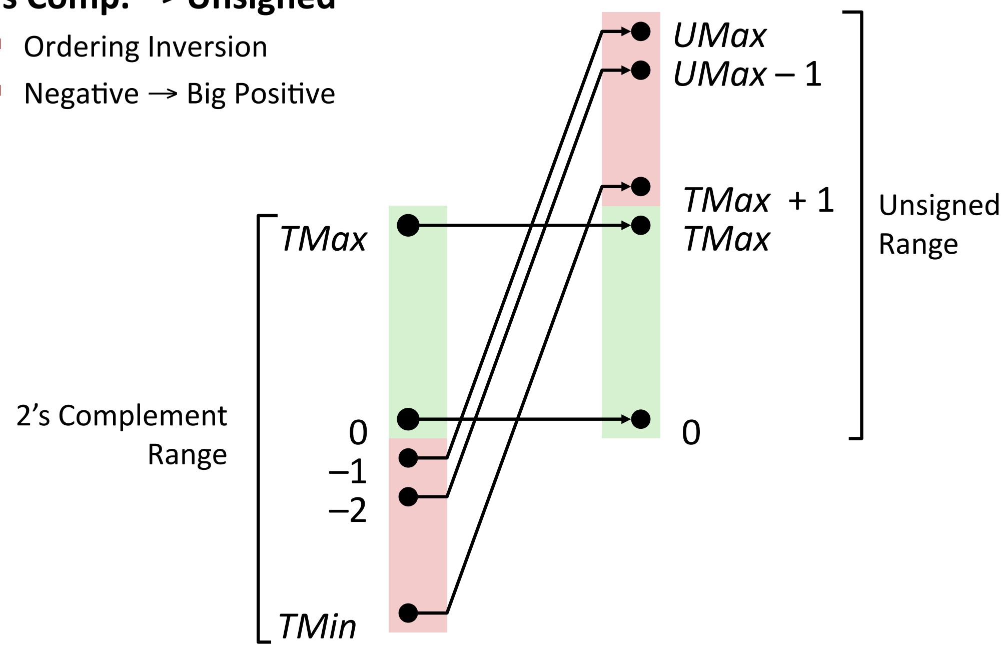


Large negative weight
becomes
Large positive weight

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

Conversion Visualized

- **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" as suffix

```
0U, 4294967259U
```

- **Casting**

```
int tx, ty;
unsigned ux, uy;
```

  - Explicit casting between signed & unsigned same as U2T and T2U

```
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: **TMIN = -2,147,483,648** **TMAX = 2,147,483,647**

Constant₁	Constant₂	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Code Security Example (revisited)

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

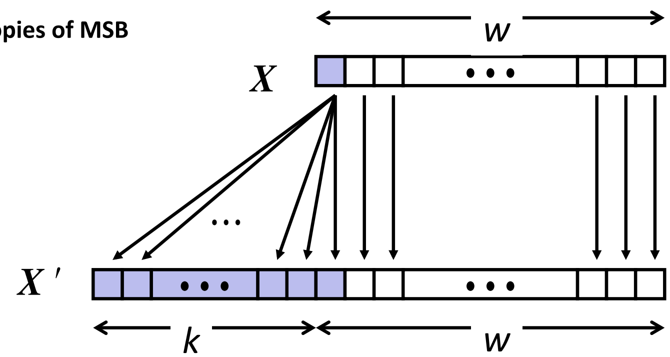

Summary
Casting Signed ↔ Unsigned: Basic Rules

- **Bit pattern is maintained**
- **But reinterpreted**
- **Can have unexpected effects: adding or subtracting 2^w**

- **Expression containing signed and unsigned int**
  - int is cast to unsigned!!

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



Sign Extension Example

```
short int x =  12345;
int      ix = (int) x;
short int y = -12345;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 01101101
ix	12345	00 00 30 39	00000000 00000000 00110000 01101101
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

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

Summary:
Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
    - for small numbers only – why?

Negation: Complement & Increment

- **Claim: Following Holds for 2's Complement**

  `~x + 1 == -x`

- **Complement**
  - Observation: `~x + x == 1111…111 == -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

- **Complete Proof?**

Complement & Increment Examples

x = 12345

	Decimal	Hex	Binary
`x`	12345	30 39	00110000 00111001
`~x`	-12346	CF C6	11001111 11000110
`~x+1`	-12345	CF C7	11001111 11000111
`y`	-12345	CF C7	11001111 11000111

x = 0

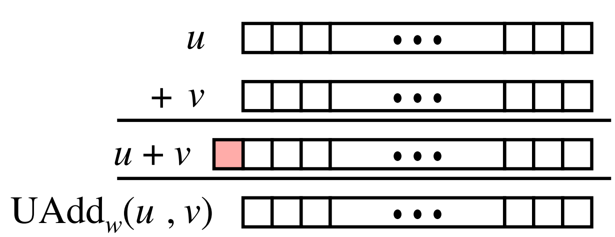
	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

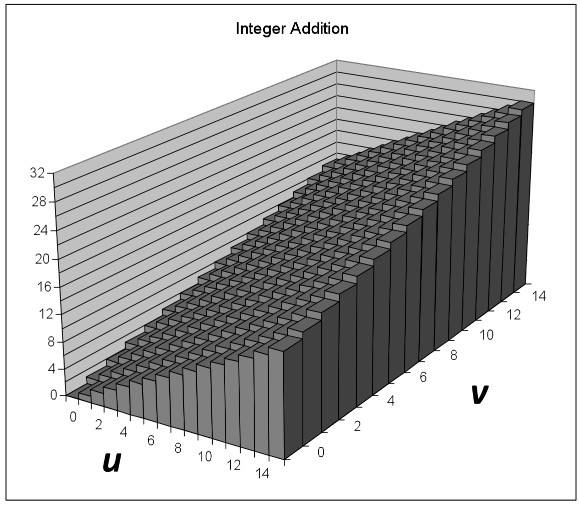
  $s \;=\; \mathrm{UAdd}_w(u\,,v) \;=\; u + v \mod 2^w$

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

Visualizing (Mathematical) Integer Addition

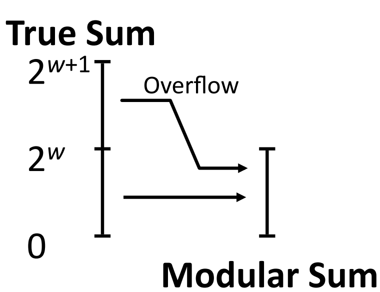
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

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

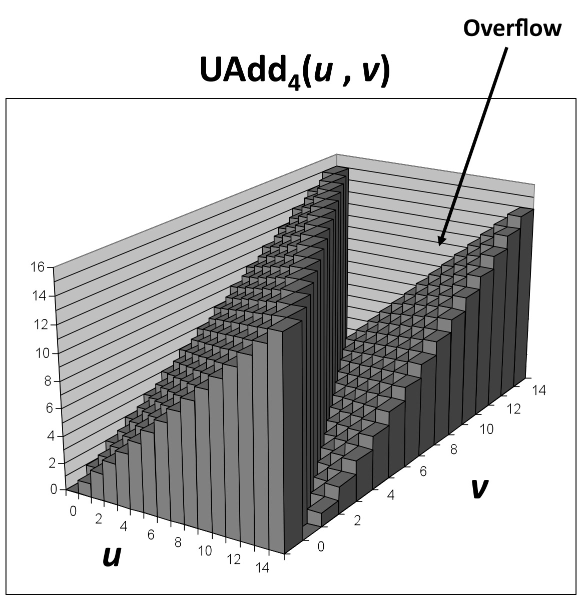


Visualizing Unsigned Addition

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

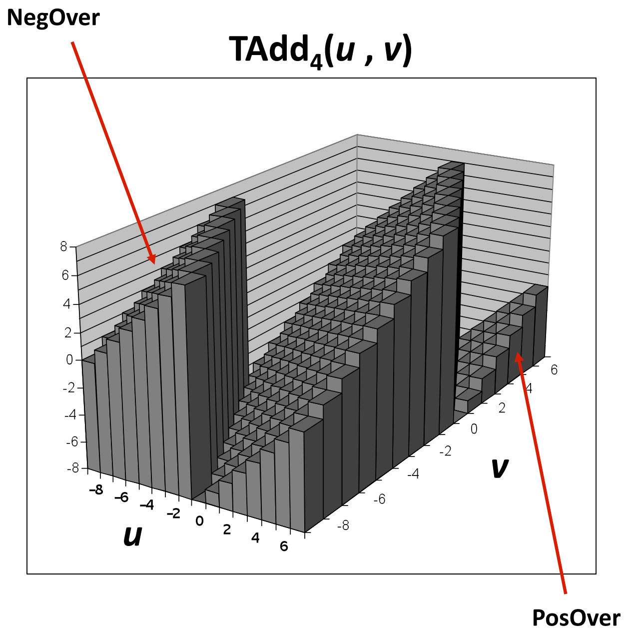


$\text{UAdd}_4(u, v)$



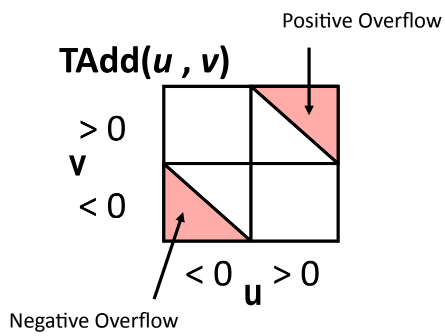
Visualizing 2's Complement Addition

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



Characterizing TAdd

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



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

Multiplication

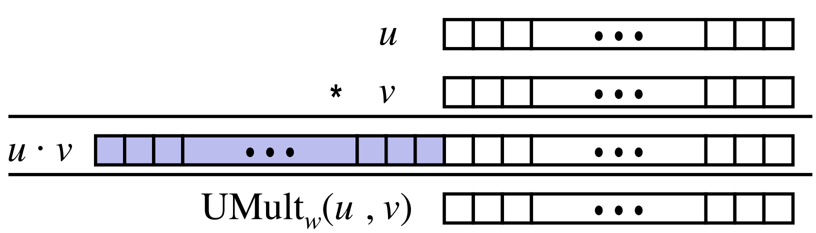
- **Computing Exact Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **Ranges**
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to 2*w* bits
  - Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to 2*w*–1 bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to 2*w* 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

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**

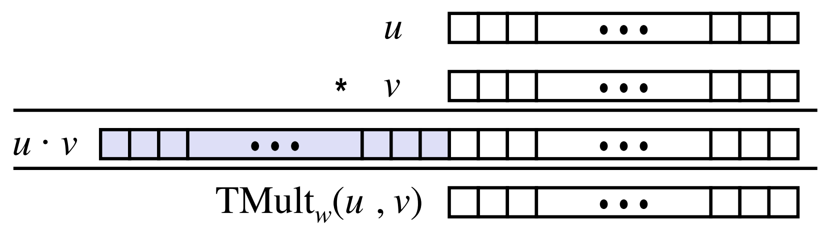
  $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

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

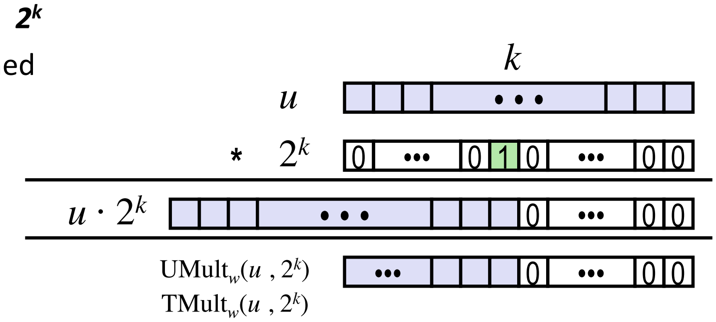
Power-of-2 Multiply with Shift

- **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

Compiled Multiplication Code

- **C compiler automatically generates shift/add code when multiplying by constant**

C Function

```
int mul12(int x)
{
  return x*12;
}
```

Compiled Arithmetic Operations

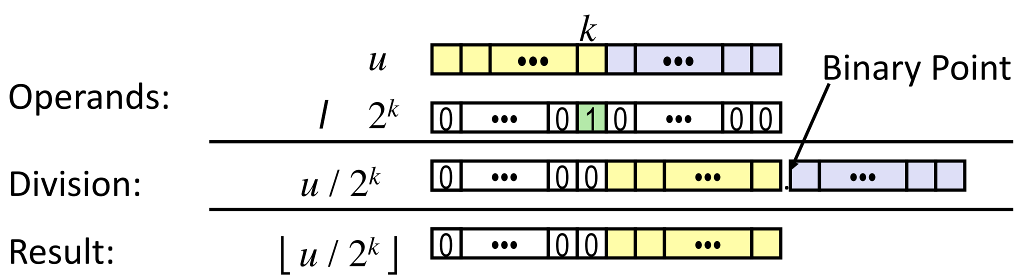
```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

```
t <- x+x*2
return t << 2;
```


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`	12345	12345	30 39	00110000 00111001
`x >> 1`	6172.5	6172	18 1C	00011000 00011100
`x >> 4`	771.5625	771	03 03	00000011 00000011
`x >> 8`	48.2226163	48	00 30	00000000 00110000

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

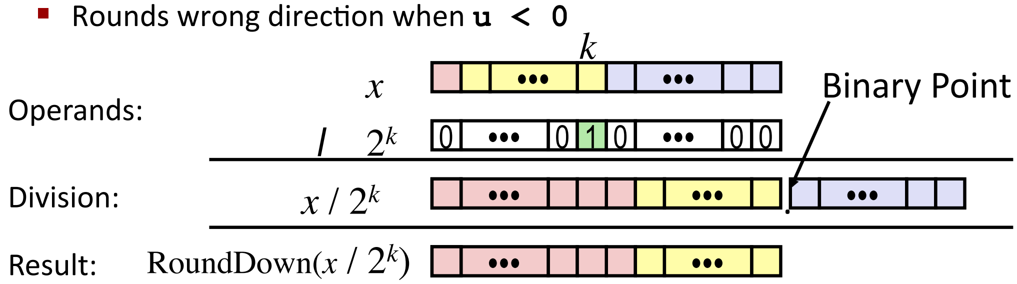
Explanation

```
# Logical shift
return x >> 3;
```

- **Uses logical shift for unsigned**
- **For Java Users**
  - Logical shift written as >>>

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`

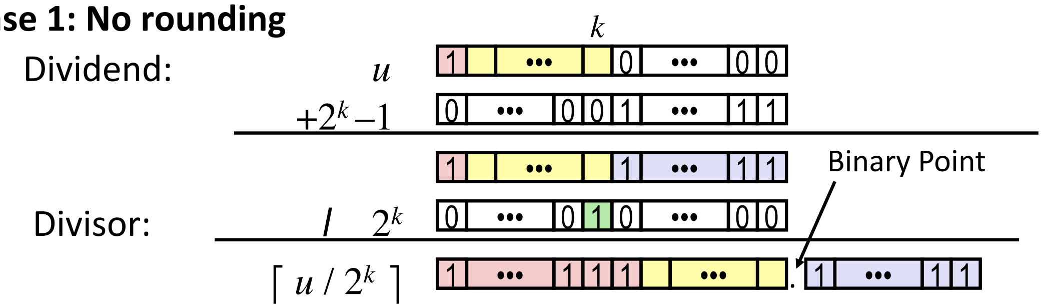


	Division	Computed	Hex	Binary
`y`	-12345	-12345	`CF C7`	`11001111 11000111`
`y >> 1`	-6172.5	-6173	`E7 E3`	`11100111 11100011`
`y >> 4`	-771.5625	-772	`FC FC`	`11111100 11111100`
`y >> 8`	-48.2226563	-49	`FF CF`	`11111111 11001111`

Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want ⌈ `x / 2`^{k} ⌉ (Round Toward 0)
  - Compute as ⌊ `(x+2`^{k}`-1)/ 2`^{k} ⌋
    - In C: `(x + (1<<k)-1) >> k`
    - Biases dividend toward 0

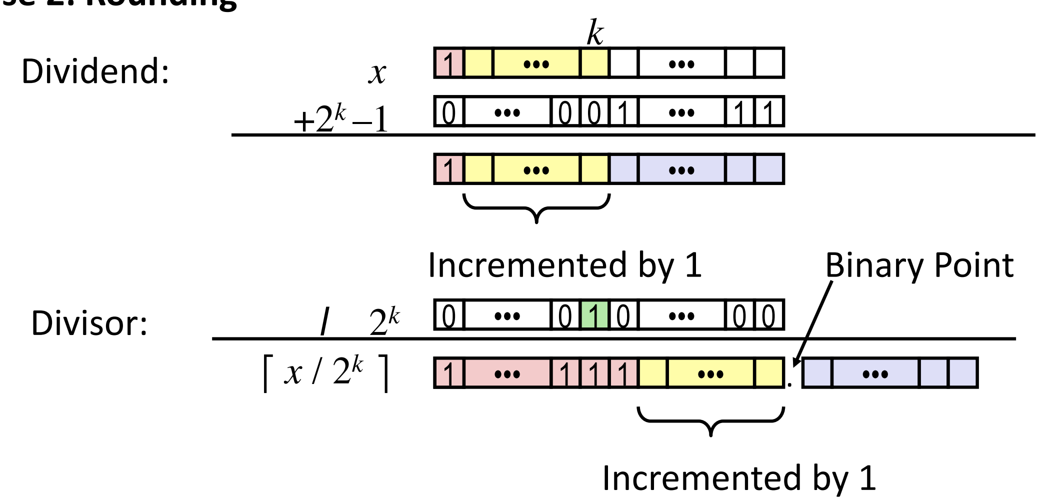
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
  return x/8;
}
```

Compiled Arithmetic Operations

```
  testl %eax, %eax
  js    L4
L3:
  sarl $3, %eax
  ret
L4:
  addl $7, %eax
  jmp   L3
```

Explanation

```
  if x < 0
    x += 7;
  # Arithmetic shift
  return x >> 3;
```

- **Uses arithmetic shift for int**
- **For Java Users**
  - Arith. shift written as >>

Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2w

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Arithmetic: Basic Rules

- **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