Integers

Topics

- Representations of Integers
  - Basic properties and operations
  - Implications for C

Slides with GREY titles will not be covered in class

Having been through the computer engineering courses, it is expected you are familiar enough with twos-compliment. You can read through the immediately following slides as reference and review. Except note this new information for C:

C Programming

- #include <limits.h>
  - K&R App. B11
- Declares constants defining range of builtin variable types, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values are platform-specific, using them increases code portability and readability

Encoding Integers

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

```
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

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
```

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**

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>`
  - K&R App. B11
- 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

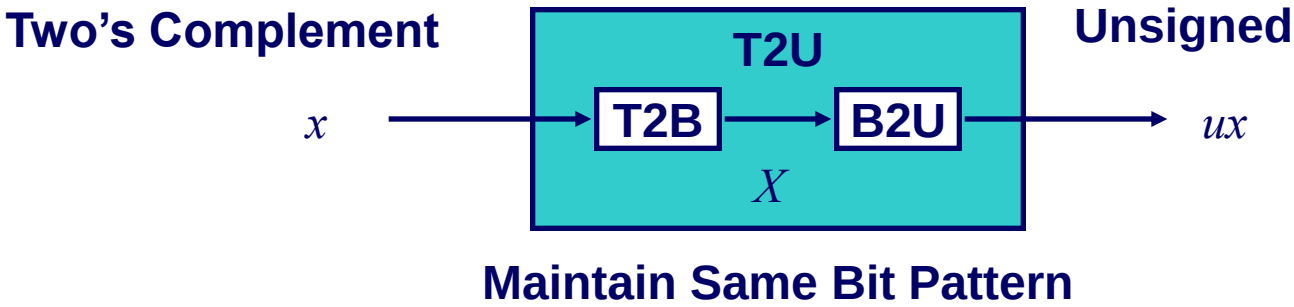
- **Every bit pattern represents unique integer value**
- **Each representable integer has unique bit encoding**

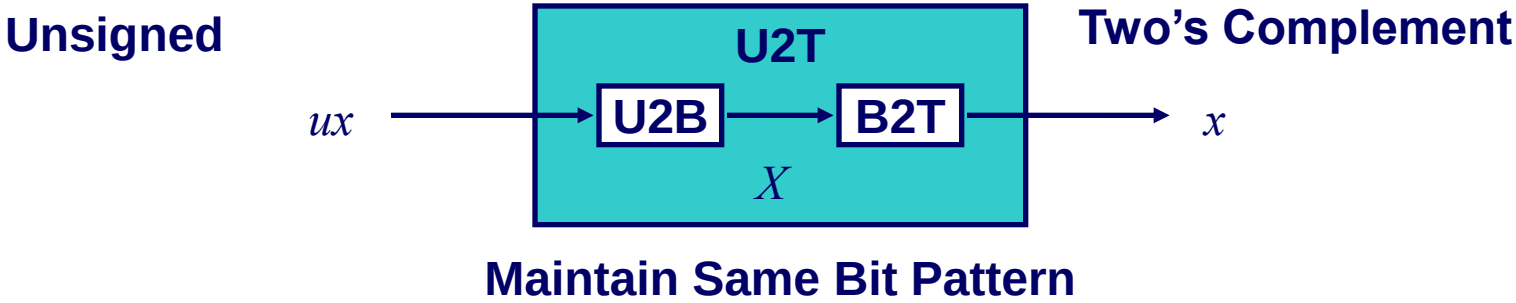
⇒ Can Invert Mappings

- **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

conversions are defined by reinterpretation of the same bits.





- **Define mappings been unsigned and two's complement numbers based on their bit-level representations**

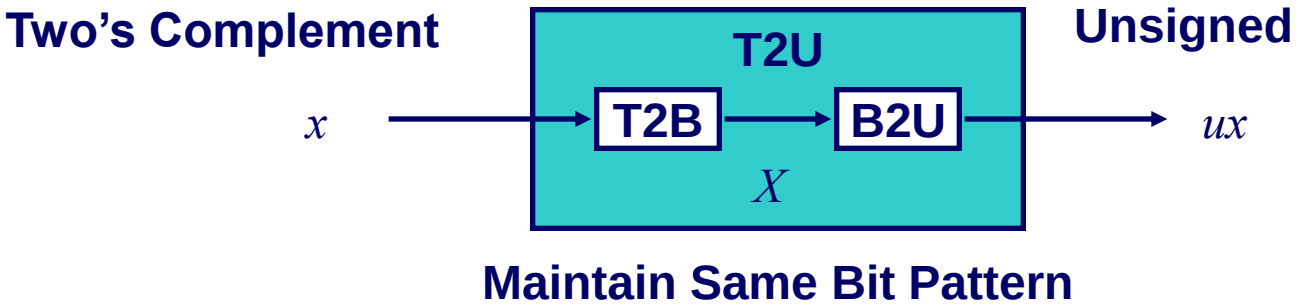
Mapping Signed ↔ Unsigned

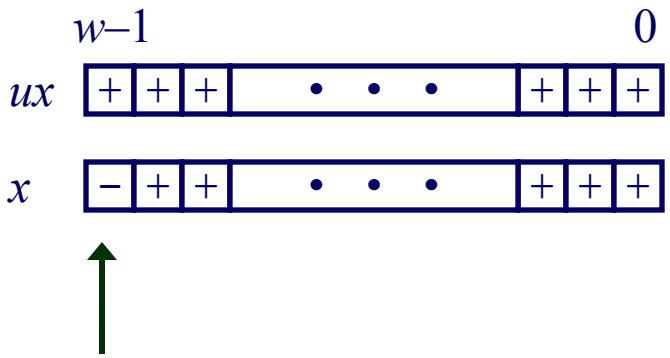
Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7	→ T2U →	7
1000	-8		8
1001	-7	← U2T ←	9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

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	+16 →	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Relation between Signed & Unsigned





Large negative weight
→
Large positive weight

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

Integer and Float Literal Constants

- By default integer numerical literals are considered to be signed integers
- Unsigned if have “U” as suffix
  - `0U, 4294967259U`
- Long and unsigned long use suffix L and UL
  - 123L, 123UL
- Floating point numbers can use suffix F or L for long doubles along with a decimal. or an E
  - 12.3F, 123E-1F

Signed vs. Unsigned in C

Casting

- **Explicit casting between signed & unsigned same as U2T and T2U** conversions are defined by dumb ***reinterpretation*** of the same bits.

```
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;
```

(Implicit) Casting Surprises

Expression Evaluation

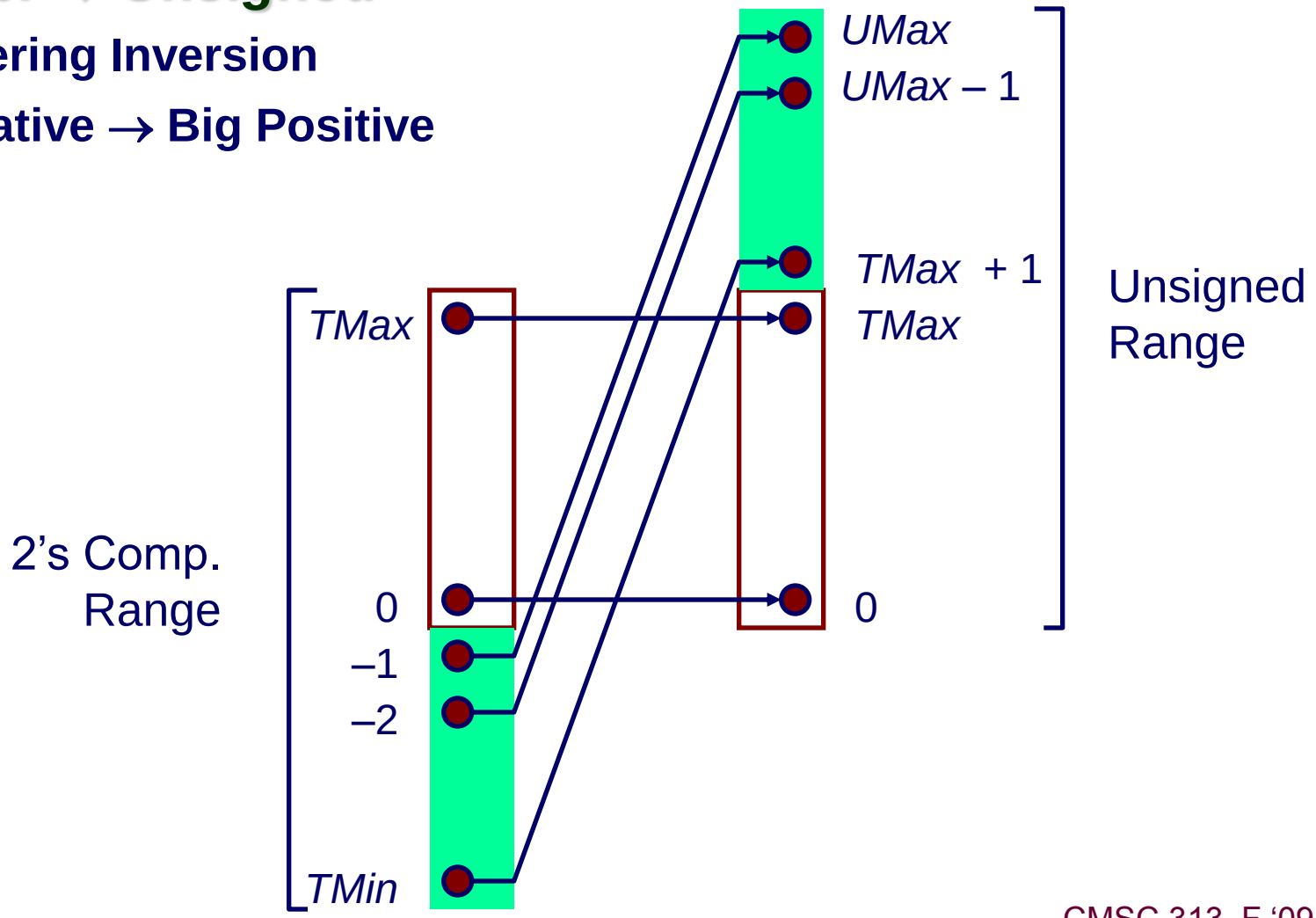
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for *W* = 32

Constant$_1$	Constant$_2$	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

Explanation of Casting Surprises

2's Comp. → Unsigned

- **Ordering Inversion**
- **Negative → Big Positive**



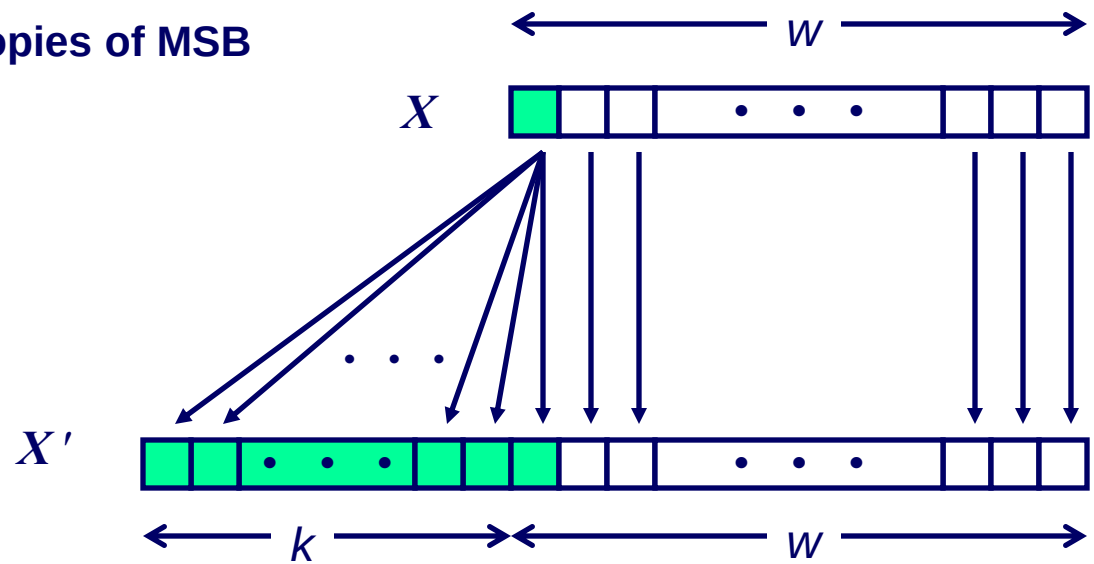
Sign Extension (CE Review)

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 =  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**

Why Should I Use Unsigned?

For larger max number, otherwise avoid use to avoid pitfalls

Be aware of pitfalls

- **Easy to make mistakes**

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

- **Can be very subtle**

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

***Do* Use When Performing Modular Arithmetic**

- **Multiprecision (arbitrary resolution) arithmetic**

***Do* Use When Using Bits to Represent Sets/Lists of 1's and 0's**

- **Logical right shift, no sign extension**

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

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

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

- **OK here**

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`

0

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

Long/short unsigned/signed Conversion Rules

```
unsigned long = unsigned long
unsigned long =   signed long
  signed long = unsigned long
  signed long =   signed long
```

```
unsigned short = unsigned long
unsigned short =   signed long
  signed short = unsigned long
  signed short =   signed long
```

```
unsigned long = unsigned short
unsigned long =   signed short
  signed long = unsigned short
  signed long =   signed short
```

```
unsigned short = unsigned short
unsigned short =   signed short
  signed short = unsigned short
  signed short =   signed short
```

The rules are "simple":

<u>Step 1</u>: extend or truncate

Going from longer to shorter, upper bits are truncated

Going from shorter to longer, zero or sign extension is done depending on source type being unsigned or signed respectively

<u>Step 2</u>: bit copy

To/From signed/unsigned is just bit copying, no other smart manipulations or conversions are done

Summary of points

Two compliment addition of two N-bit numbers can require up to N+1 bits to store a full result

Two's compliment addition can only overflow if signs of operands are the same (likewise for subtraction the signs must be different)

Result of N-bit addition with overflow is dropping of MSBits's: A+B = (A+B) mod (2^N)

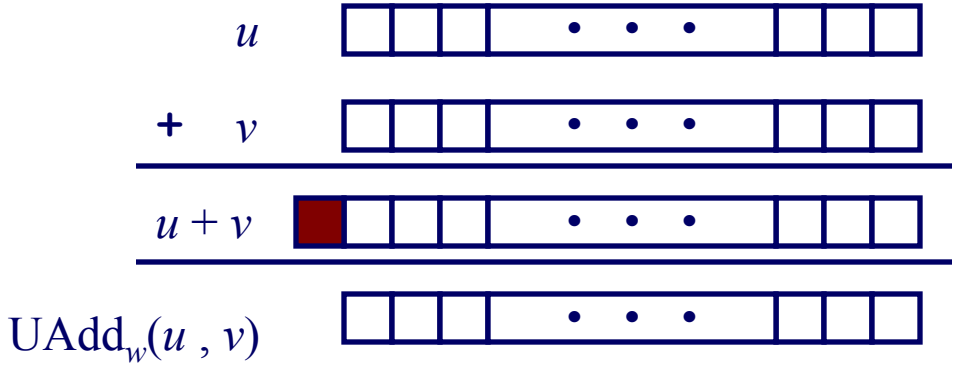
For multiplication, multiplying two N-bit numbers requires up to 2N bits to store the operand. Multiplying a N-bit with a M-bit requires up to N+M bits.

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Standard Addition Function

- **Ignores carry output for result in C (though it is stored in carry bit in machine)**

Implements Modular Arithmetic

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

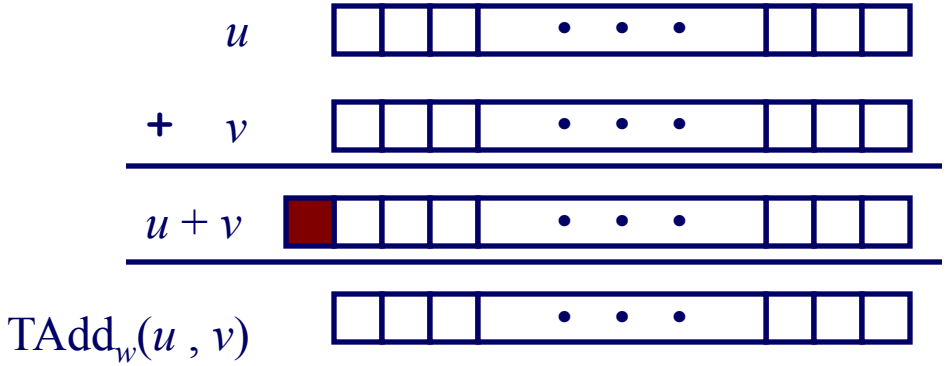
$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical Bit-Level Behavior

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

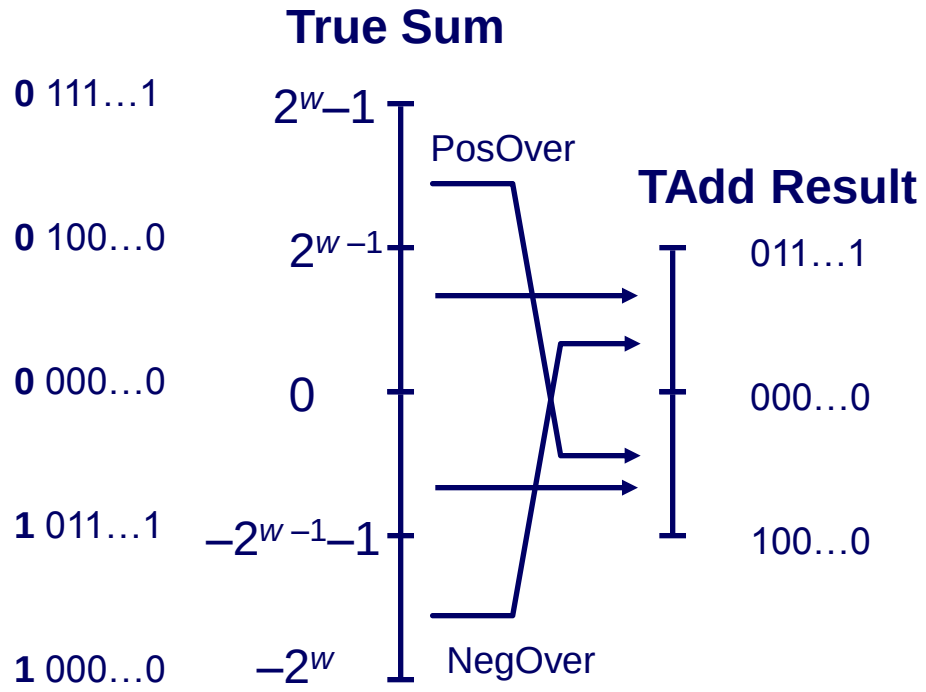
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

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

TAdd Overflow

Functionality

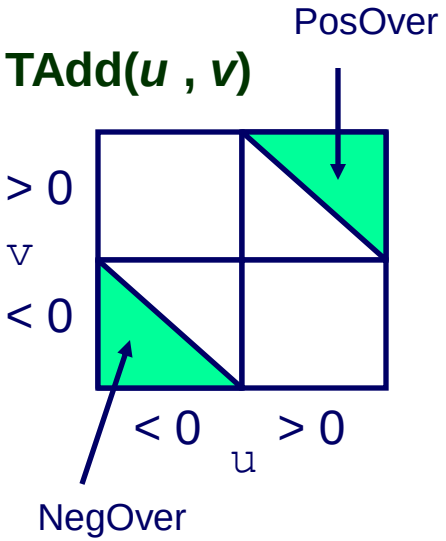
- **True sum requires w+1 bits**
- **Drop off MSB**
- **Treat remaining bits as 2's comp. integer**



Characterizing TAdd

Functionality

- **True sum requires *w*+1 bits**
- **Drop off MSB**
- **Treat remaining bits as 2's comp. integer**
- **Can only overflow if signs of operands are the same**



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

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

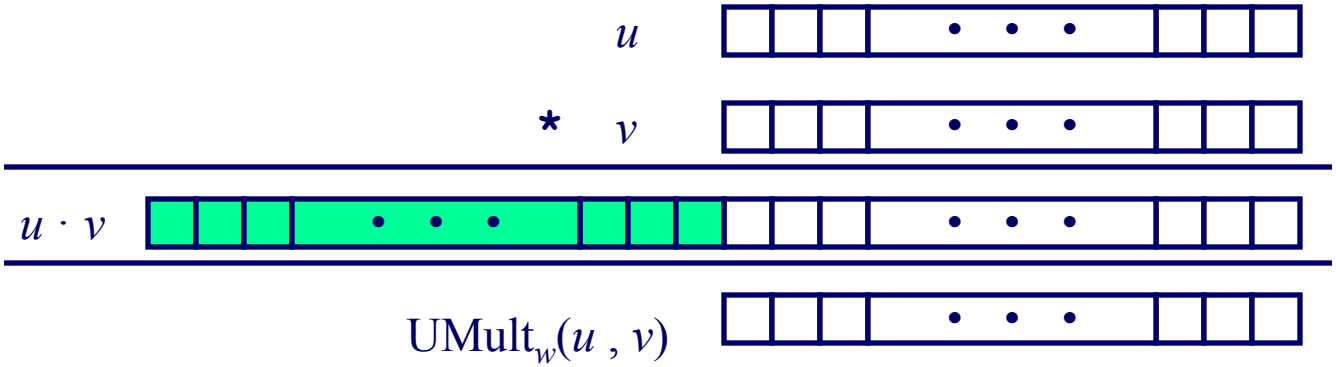
- **Would need to keep expanding word size (approximately add the number of bits of each operand) with each product computed**
- **Done in software by "arbitrary precision" arithmetic packages**

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

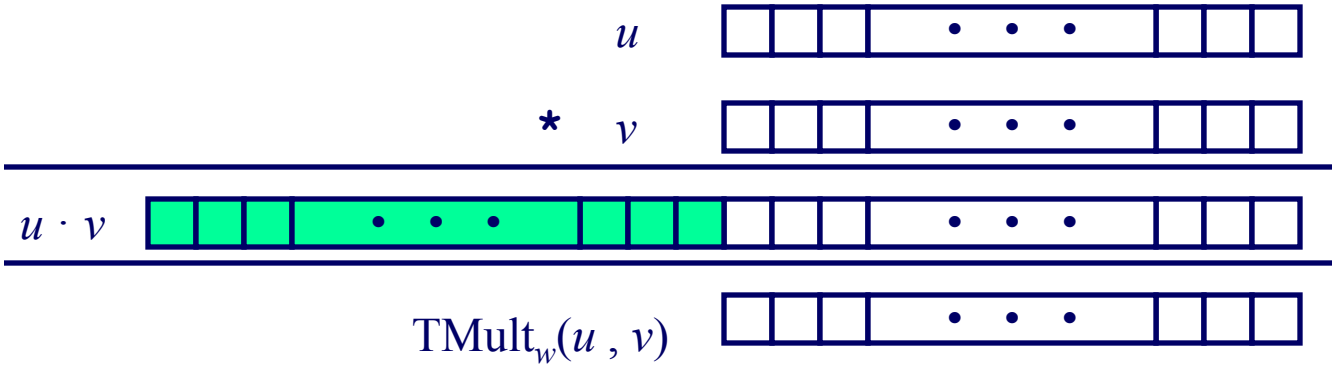
$$\text{UMult}_w(u\ ,\ v) \quad = \quad u \cdot v \mod 2^w$$

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



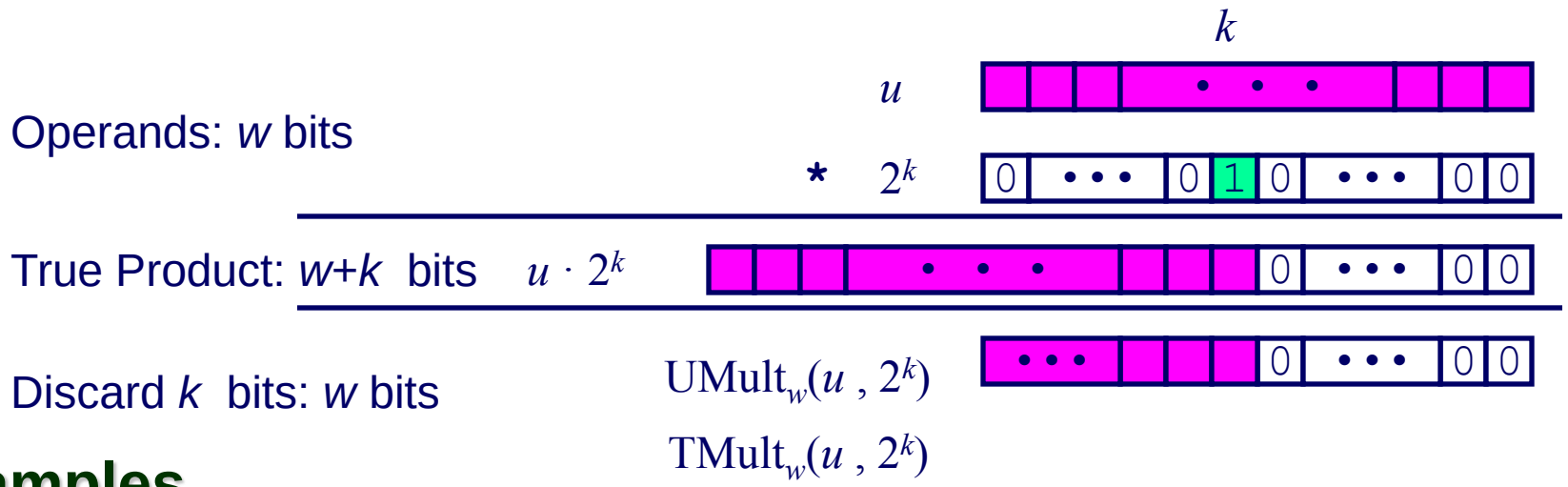
Standard Multiplication Function

- **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**



Examples

- `u << 3 == u * 8`
- **Important "trick" for a CMPE to know:**
  - **To implement `u * 24` can use `u << 4 + u << 3`**
- **Most machines shift and add faster than multiply**
  - **Compiler generates this code automatically**

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

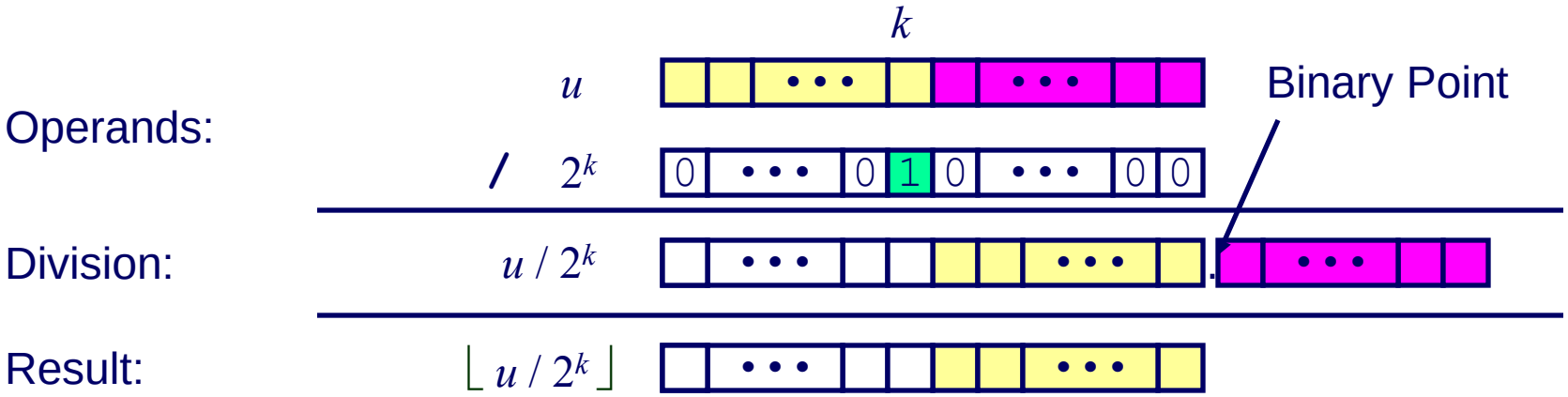
```
  t <- x + x * 2
  return t << 2;
```

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

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`

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

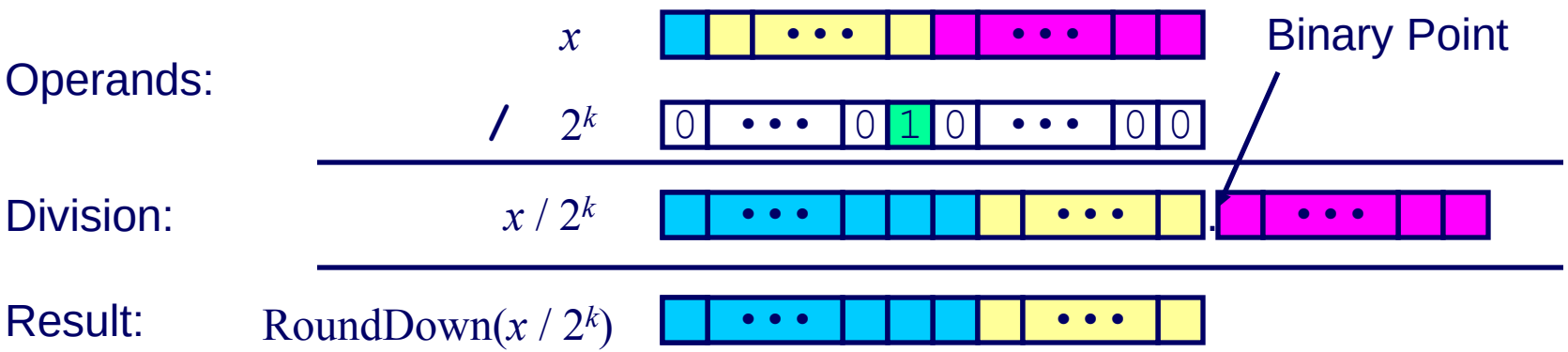
```
  # Logical shift
  return x >> 3;
```

- **Uses logical shift for unsigned**

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- `x >> k` gives $\lfloor x / 2^k \rfloor$ instead of the round-towards-zero that standard integer division provides -3/2=-1
- Uses arithmetic shift
- Rounds wrong direction when $u_k < 0$



	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`

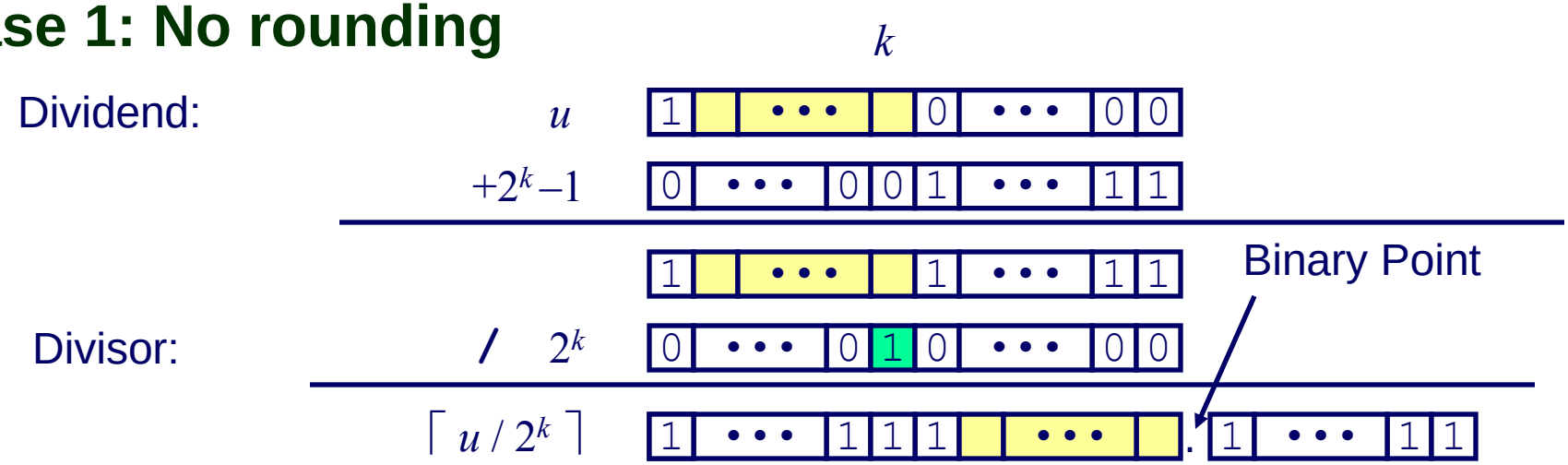
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

For negative number,

- 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

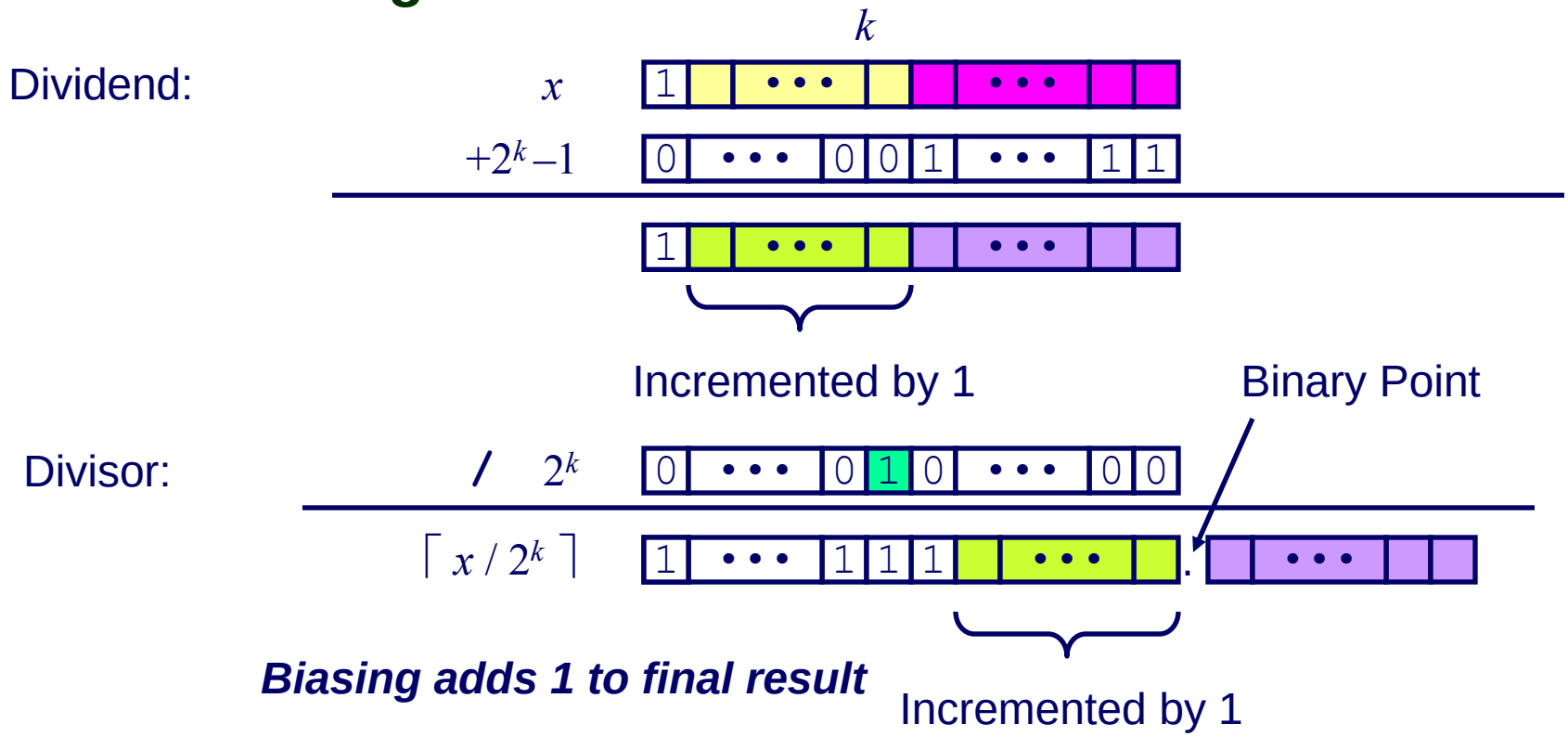
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

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

```
#add bias for negatives
if x < 0
  x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int

Arbitrary Precision Arithmetic for Longer numbers

For 16 bit calculations, an 8-bit architecture may support double-register arithmetic

For even longer numbers results can be calculated a piece at a time and overflow bits (add/sum) or overflow registers (multiply) can be used to compute larger results. The built-in C variable types are usually automatically handled by the compiler. If even longer types are needed, find an arbitrary precision arithmetic software library.

Floating Point Math and Fixed-Point Math

If no floating point unit (FPU) is available, you can find a floating point software library. This will likely be slow.

Another option is fixed-point math. You can write or use a library or just do it as needed inline….

Fixed-point arithmetic

```
Want to add 0011.1110 and 0001.1000

Store 0011.1110 as 00111110 and
Store 0001.1000 as 00011000

Add 00011000 + 00111110 = 01010110

Interpret 01010110 as 0101.0110
```

Fixed-point arithmetic

Up to you to determine number of bits to use for whole and fraction parts depending on range and precision needed

Then, you get a scalar factor S that converts is to a whole number

```
1101.0000 * 16 = 11010000   S=16
01.011000 * 64 = 01011000   S=64
```

Fixed-point arithmetic

Addition:

A+B computed as A*S+B*S=C*S Divide result by S to obtain answer C (powers of two are efficient choices for S)

Subtraction:

A-B computed as A*S-B*S=C*S Divide result by S to obtain answer C

Fixed-point arithmetic

Multiplication:

A*B computed as (A*S)*(B*S)=C*S^2

Divide result by S^2
to obtain answer C
Divide result by S
to obtain scaled answer C*S
which you can use futher

Unfortunately, the intermediate result C * S^2 required more storage than the scaled result C*2

Fixed-point arithmetic

Division:

A/B could be computed as (A*S)/(B*S)=C

Scales cancel. Which is fine if you only wanted an integer answer

Would need to multi by S to obtain scaled result C*S for further math …but this is less accurate since the lower bits have already been lost

Better to prescale one of the operands ((A*S)*S)/(B*S)=C*S

Unfortunately, the intermediate term ((A*S)*S) required more storage

Rounding Errors float to int

```
float to int always truncates fractional
digits, effectively rounding towards zero

5.7 -> 5
-5.7 -> -5

Need Nearest Integer Rounding?
Add +/- .5 before truncation depending on
sign

5.7+.5 = 6.2 -> 6
5.4+.5 = 5.9 -> 5

-5.7+.5 = -5.2 -> -5 doesn't work the same
For negative numbers, need to subtract
-5.7-.5 = -6.2 -> 6
-5.4-.5 = -5.9 -> 5
```

Rounding errors

A/B gives floor(A/B) rather than round(A/B)

So (A+B/2)/B is how we get rounding with integer-only arithmetic

If B is odd, we need to choose to round B/2 up or down depending on application

Example:

Want to set serial baud rate using clock divider
i.e. BAUD_RATE=CLK_RATE/CLK_DIV

Option 1:

```
#define CLK_DIV (CLK_RATE/BAUD_RATE)
```

Option 2: BETTER CHOICE IS

```
#define CLK_DIV (CLK_RATE+ (BAUD_RATE/2)) /BAUD_RATE
```

Choices for order of operations

Need to consider loss of MSB or LSB digits with intermediate terms and make choices base on application or expected ranges for input values

Example 1:

```
int i = 660, x = 54, int y = 23;
```

want i/x*y : true answer is 255.555…

```
i/x*y          gives 253 good
i*y/x          gives 255 better
(i*y+x/2)/x    gives 256 best
```

Example 2:

```
unsigned int c = 7000, x=10,y=2;
```

want c*x/y which is truly 35000

```
c*=x;     overflows c since (c*x)>65535 resulting in x = 4465
c/=y;     get 2232 here

c/=y;
c*=x;     gives 35000 !!!!!
```

In general, if intermediate result will fit in the allowed integer word size, apply integer multiplications before integer divisions to avoid loss of precision