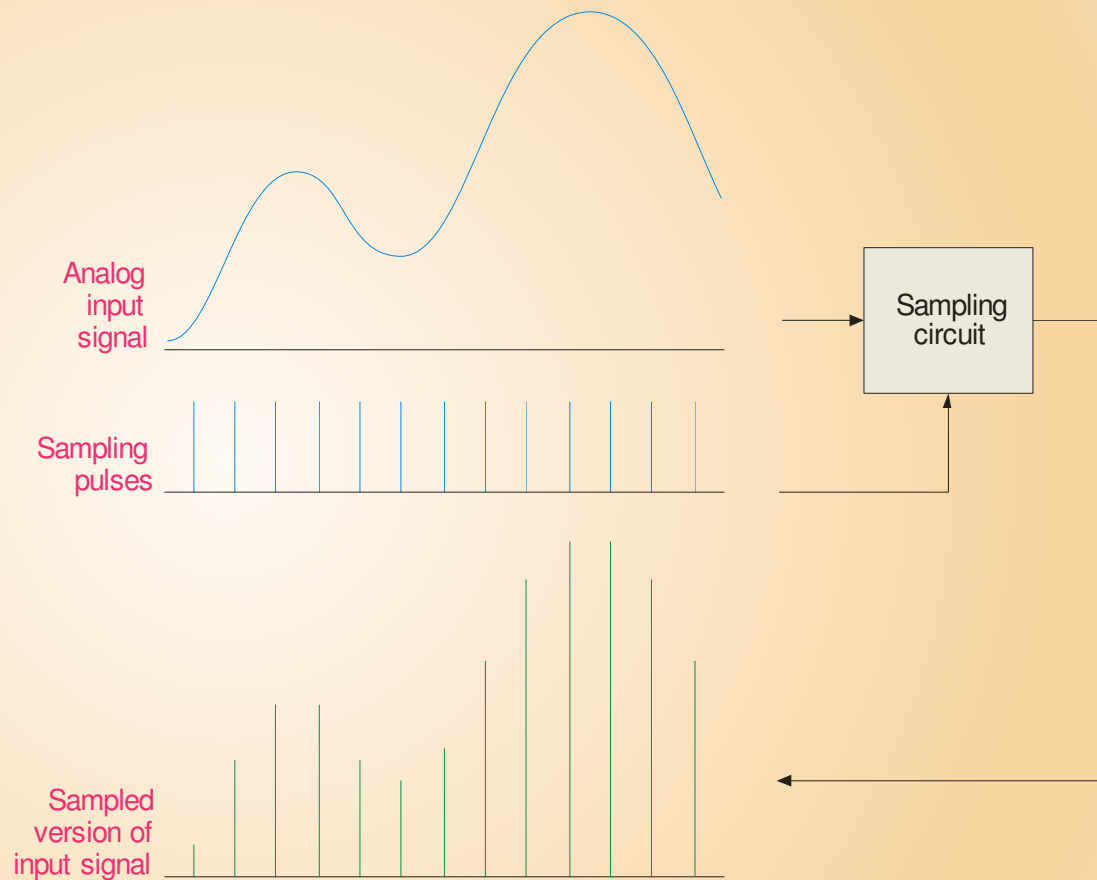


ADC – Sampling

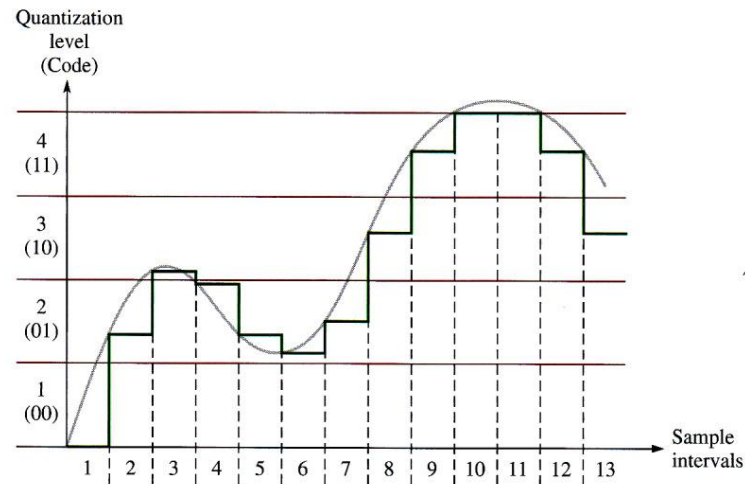
- Most input signals to an electronic system start out as analog signals. An analog quantity can be characterized as having a continuous set of values over a given range. For processing, the signal is normally converted to a digital signal by sampling the input.



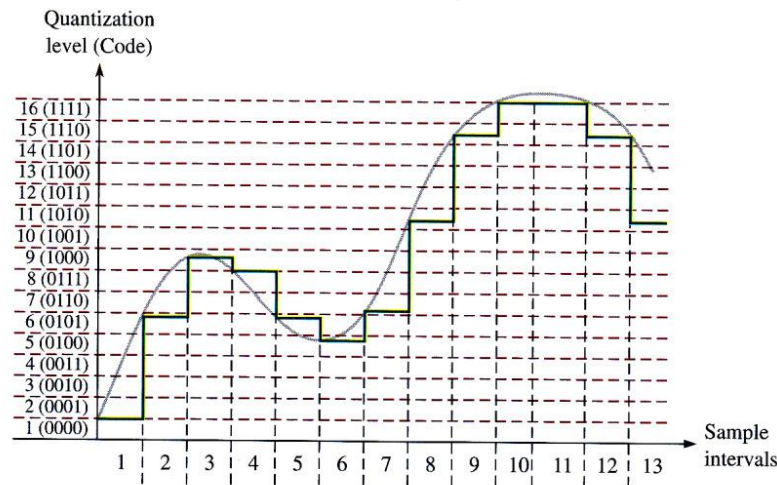
ADC – Quantization

- The number of bits used to represent the different levels of the signal

- Two Bits:

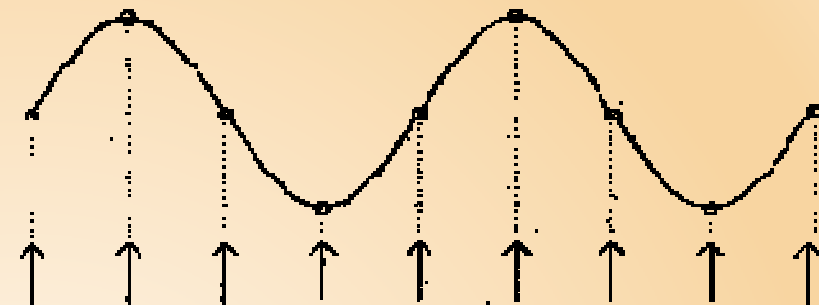


- Four Bits:

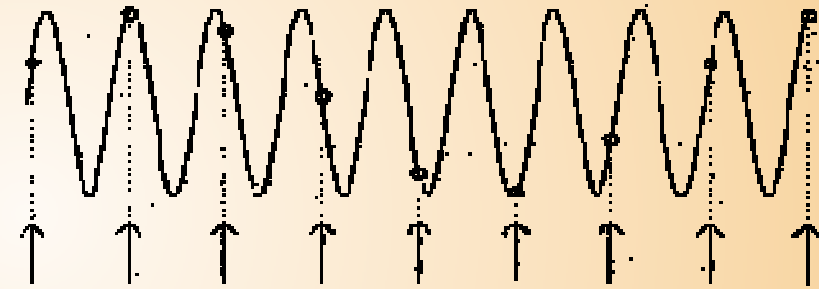


ADC – Anti Aliasing Filter

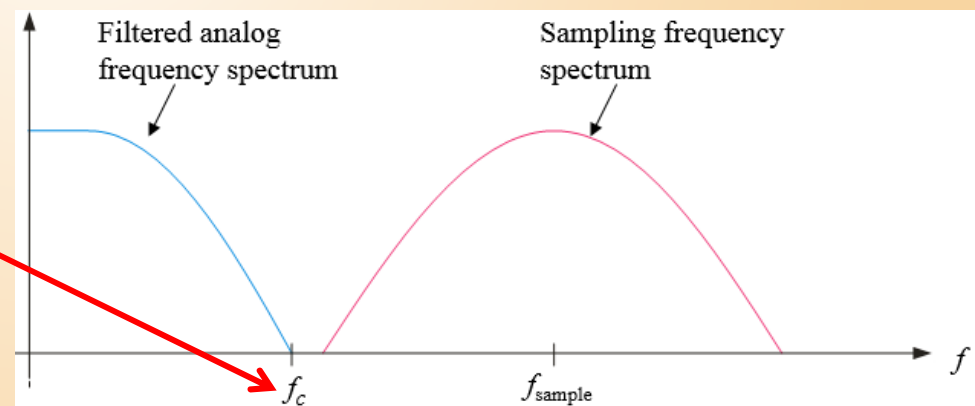
- Most signals have higher frequency harmonic and noise. For most ADCs, the sampling and filter cutoff frequencies are selected to be able to reconstruct the desired signal without including unnecessary harmonics and noise.
- The anti-aliasing filter is a low-pass analog filter that limits high frequencies in the input signal to only those that meet the requirements of the sampling theorem.
- The filter's cutoff frequency, f_c , is referred to as the **Nyquist frequency** and should be less than $\frac{1}{2}f_{\text{sample}}$.



(a) Point sampling within the Nyquist limit

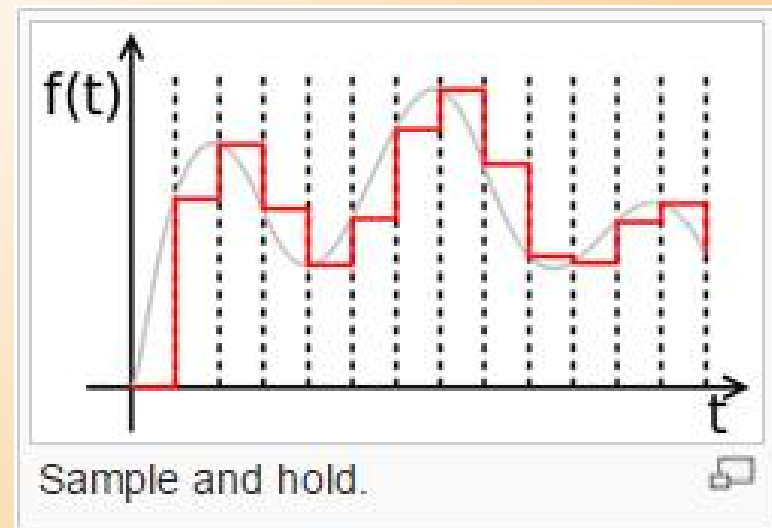
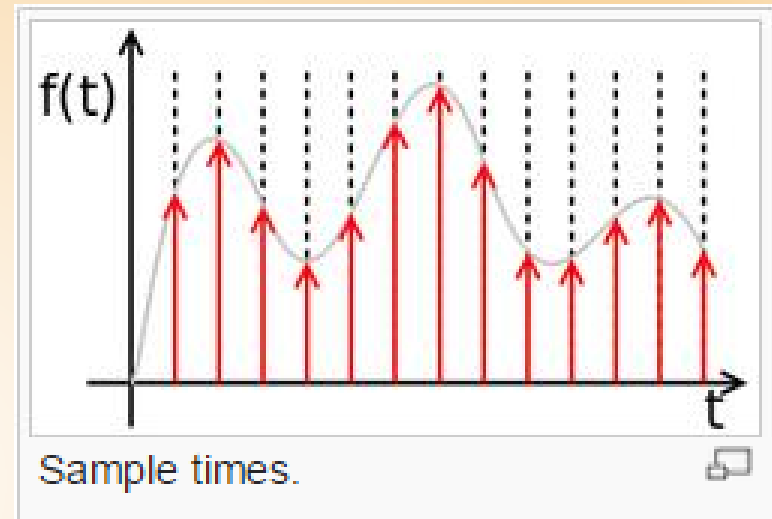
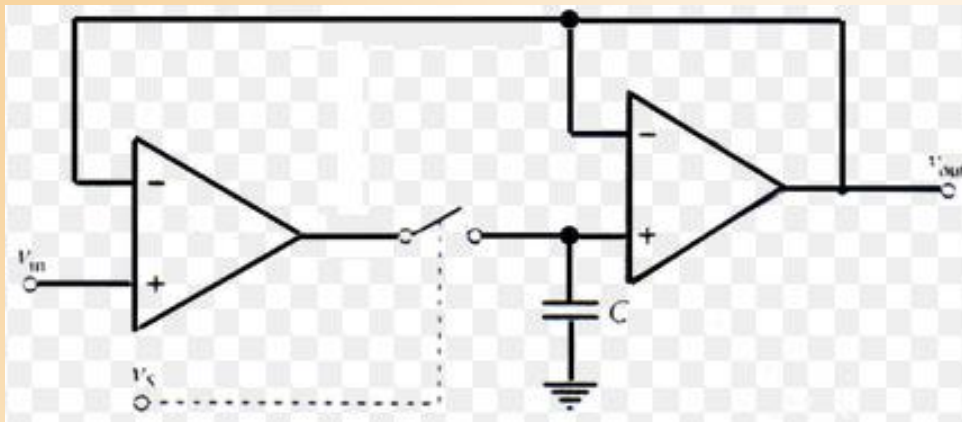


(b) Point sampling beyond the Nyquist limit

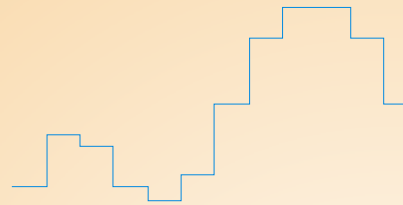


ADC – Sample and Hold (ref Fig 17.11 in text)

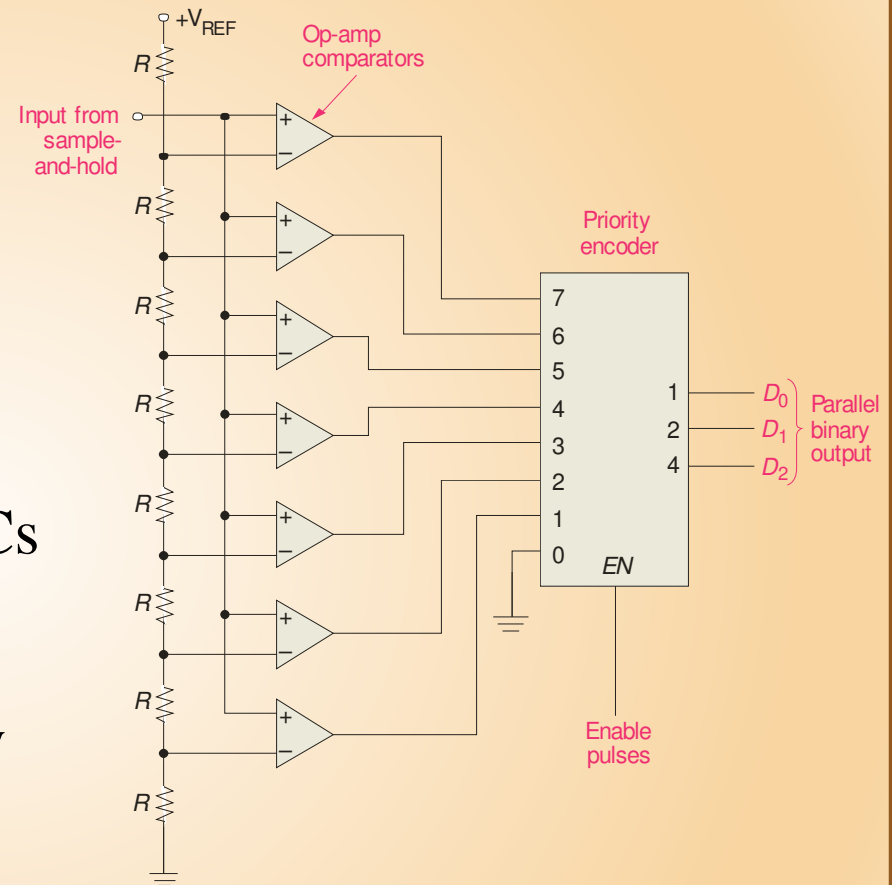
- In electronics, a sample and hold (S/H, also "follow-and-hold") circuit is an analog device that samples (captures, grabs) the voltage of a continuously varying analog signal. It then holds (locks, freezes) its value at a constant level for a specified minimum period of time. They are typically used in analog-to-digital converters to eliminate variations in input signal that can corrupt the conversion process. (ref: Wikipedia)



ADC – Converters: Flash Converter:



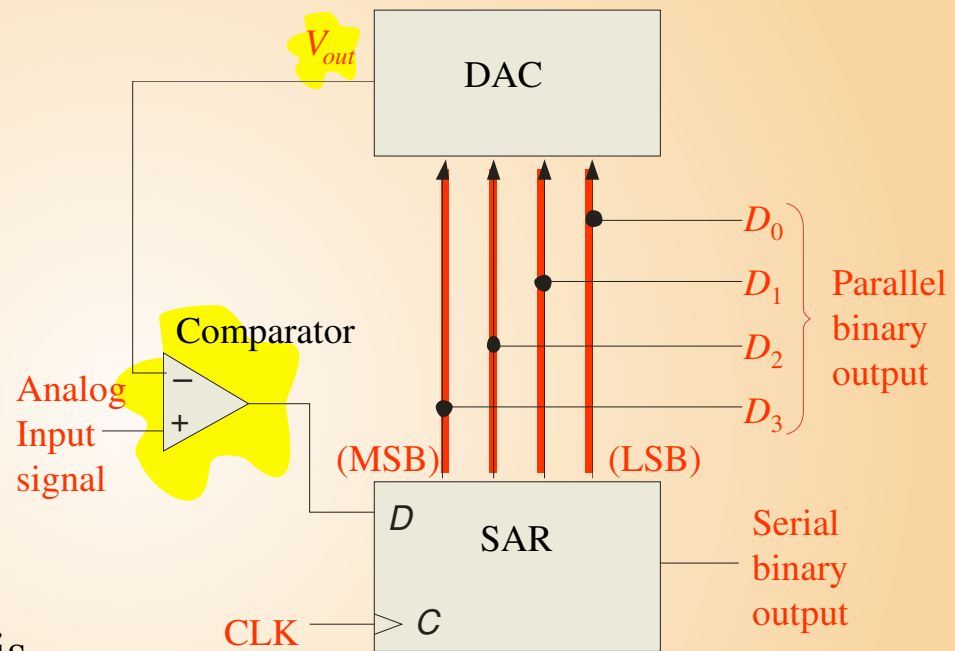
- The flash ADC uses a series high-speed comparators that compare the input with reference voltages. Flash ADCs are fast but require $2^n - 1$ comparators to convert an analog input to an n -bit binary number.
- So, for 10 bits of resolution, how many comparators would be needed?



ADC – Converters: Successive Approximation:

- Section 17.4.2 in your textbook for another reference:

1. Starting with the MSB, each bit in the successive approximation register (SAR) is activated and tested by the digital-to-analog converter (DAC).
2. After each test, the DAC produces an output voltage that represents the bit.
3. The comparator compares this voltage with the input signal. If the input is larger, the bit is retained; otherwise it is reset (0).



The method is fast and has a fixed conversion time for all inputs.

Integers Overview

- Topics:
 - Representations of Integers
 - Basic properties and operations
 - Implications for C
- Slides with – this slide set is originally from the Fall, 2009 CMSC313 class. Because of this, not all the slides presented will be covered in class – but they are still presented for reference. Any slides with Blue titles are for reference, but will not be covered in class tonight.
- 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. Please 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

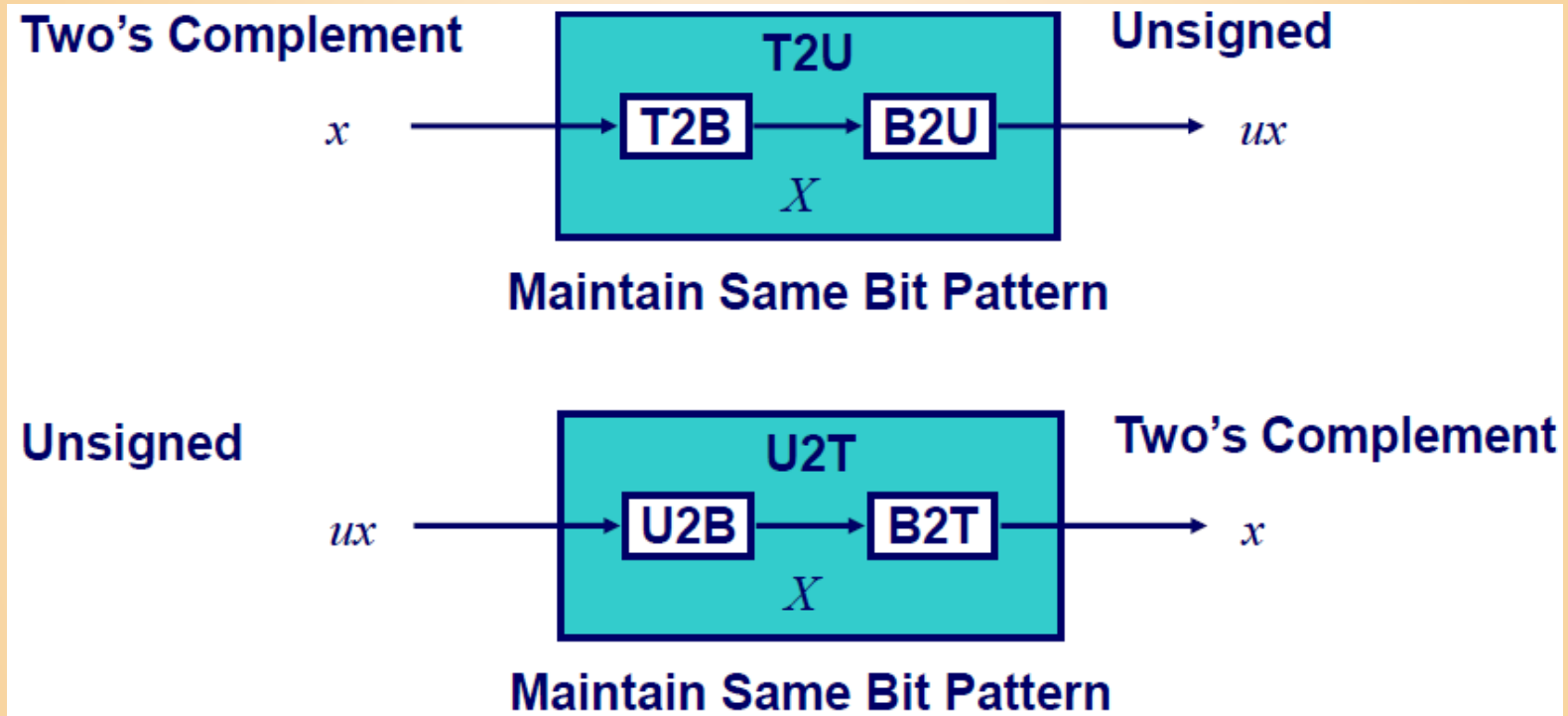
Unsigned & Signed Numeric Values

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 between unsigned and two's complement numbers based on their bit-level representations

Mapping Signed \leftrightarrow Unsigned

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

Mapping Signed \leftrightarrow 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

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;
```

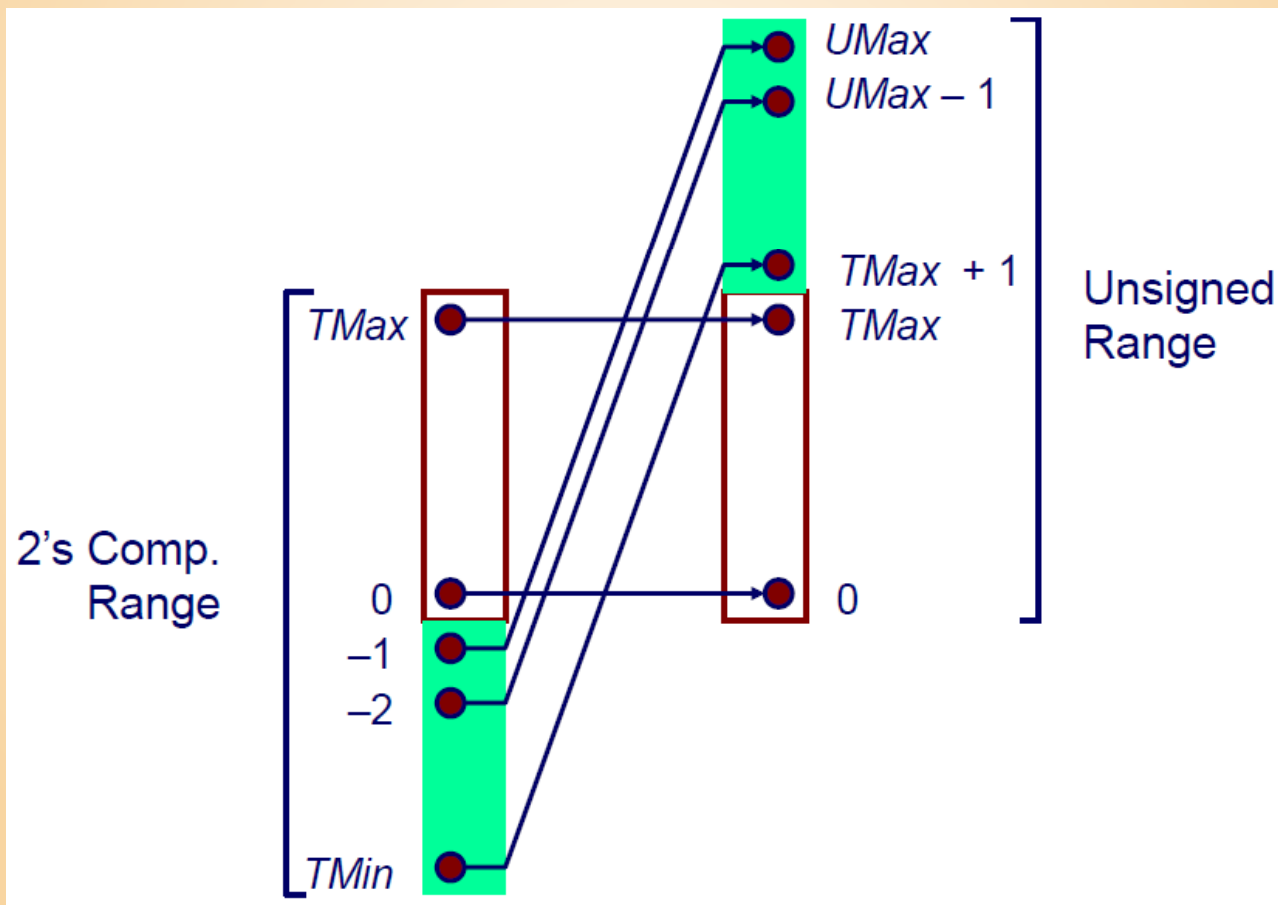

(Implicit) Casting Surprises

- 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	Relation	Constant ₂	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

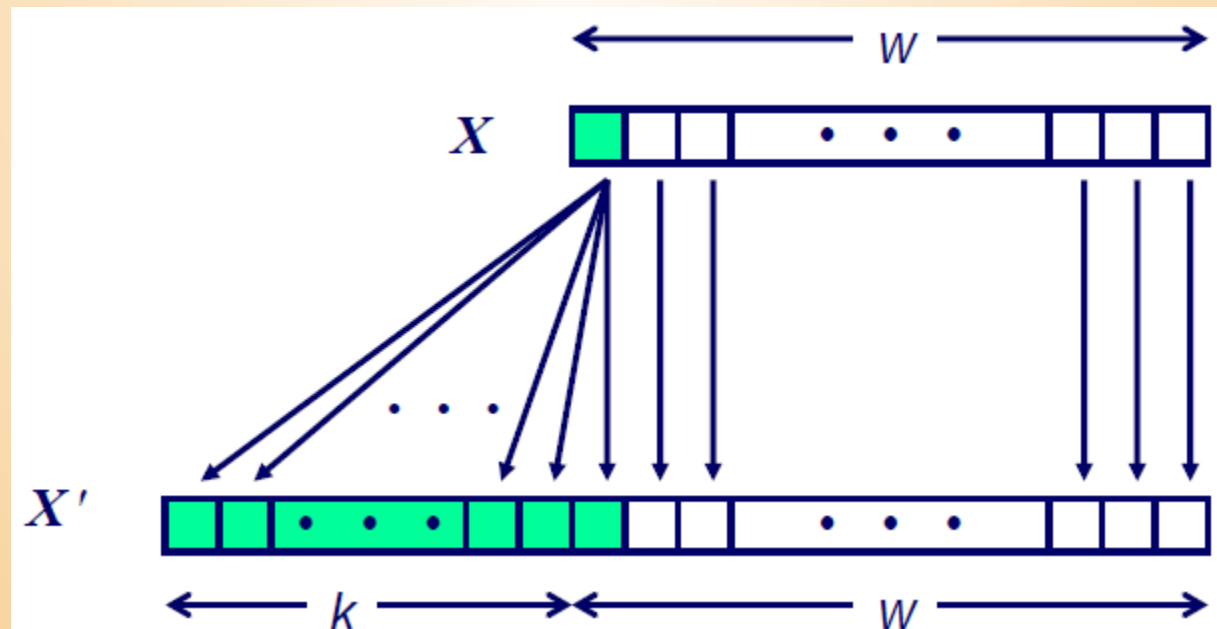
Explanation of Casting Surprises

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



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

- $\sim x + 1 == -x$

- Complement

- Observation: $\sim x + x == 1111\dots 11_2 == -1$

x	1	0	0	1	1	1	0	1
+ ~x	0	1	1	0	0	0	1	0
<hr/>								
-1	1	1	1	1	1	1	1	1

- Increment

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

- Warning: Be cautious treating int's as integers

- OK here

Complement & Increment 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

Long/short unsigned/signed Conversion Rules

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

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

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

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

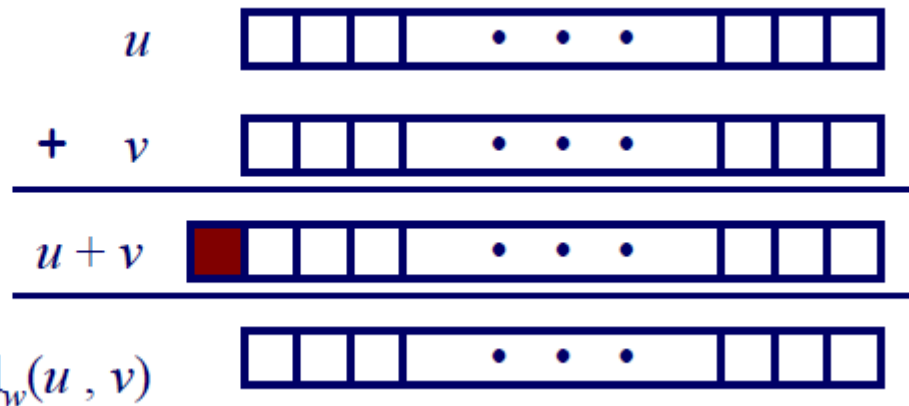
- The rules are “simple”:
 - Step 1: extend or truncate
 - Going from longer to shorter, upper bits are truncated
 - Going from shorter to longer, zero or sign extension is done depending on source type being unsigned or signed respectively
 - Step 2: bit copy
 - To/From signed/unsigned is just bit copying, 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:
 $A+B = (A+B) \bmod (2^N)$
- 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

$UAdd_w(u, v)$

- Standard Addition Function
 - Ignores carry output for result in C (though it is stored in carry bit in machine)
- Implements Modular Arithmetic
 - $s = UAdd_w(u, v) = u + v \bmod 2^w$

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

Two's Complement Addition

Operands: w bits

u



+ v



True Sum: $w+1$ bits

$u + v$



Discard Carry: w bits

$\text{TAdd}_w(u, v)$



- TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

```
int s, t, u, v;
```

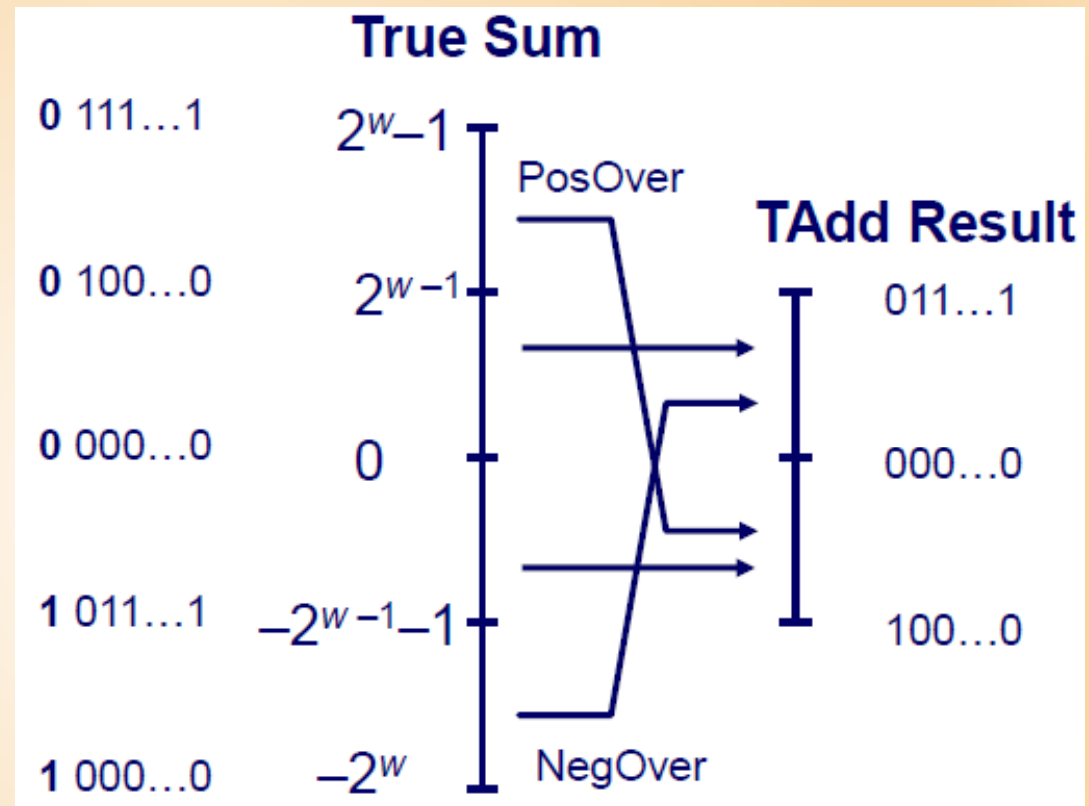
```
s = (int) ((unsigned) u + (unsigned) v);
```

```
t = u + v
```

- Will give `s == t`

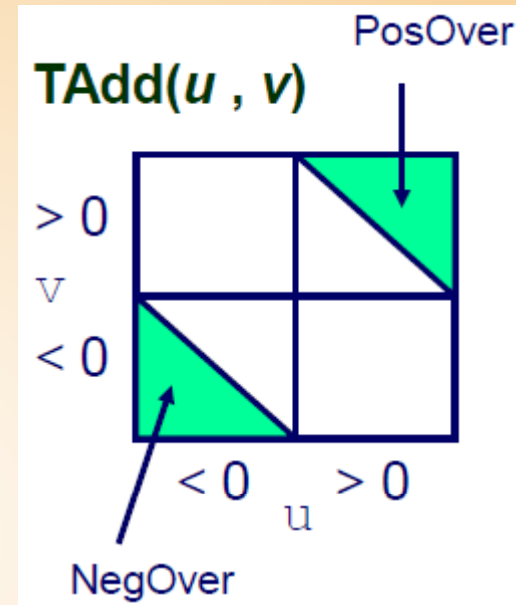
TAdd Overflow

- 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**

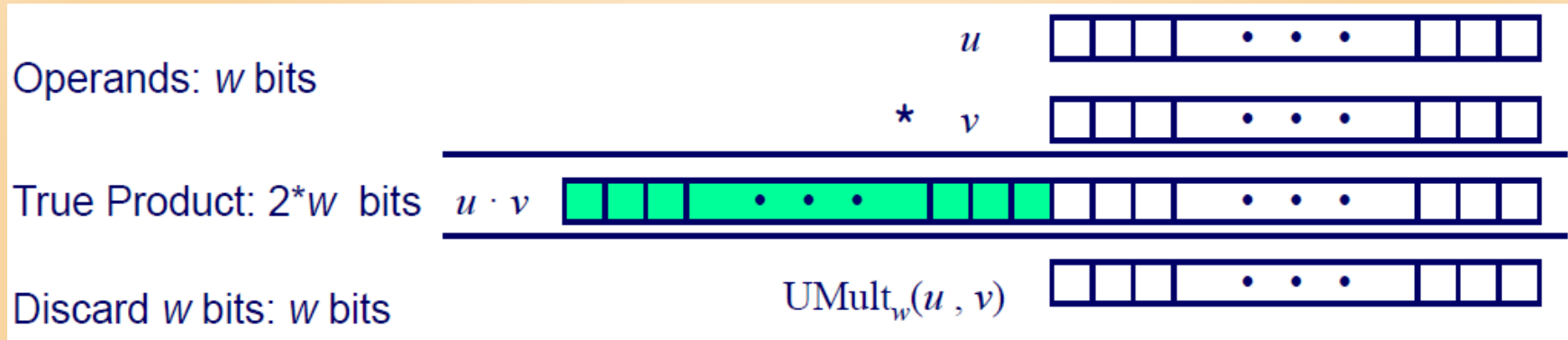


$$TAdd_w(u, v) = \begin{cases} u + v + 2^{w-1} & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^{w-1} & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

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
 - Two's complement min: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
 - Up to 2w-1 bits
 - Two's complement max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
 - Up to 2w bits, but only for $(TMin_w)^2$
- Maintaining Exact Results
 - Would need to keep expanding word size (**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

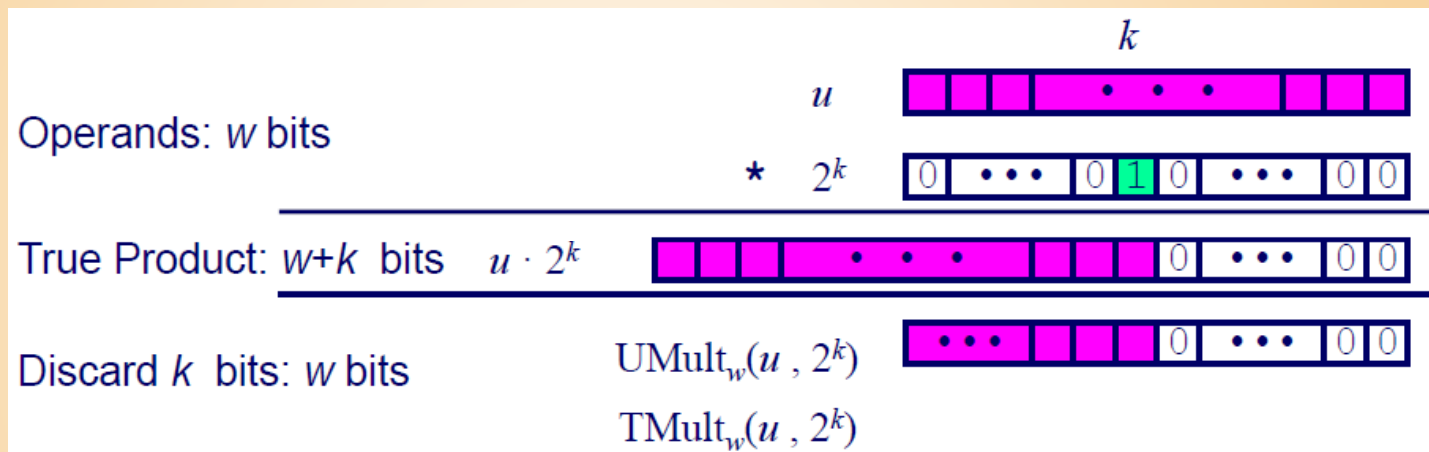


- Standard Multiplication Function
 - Ignores high order w bits
- Implements Modular Arithmetic
 - $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Power-of-2 Multiply with Shift

- Operation

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



- Examples

- $u \ll 3 == u * 8$
- Important “trick” for a CMPE to know:
 - To implement $u * 24$ can use $u \ll 4 + u \ll 3$
- **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
```

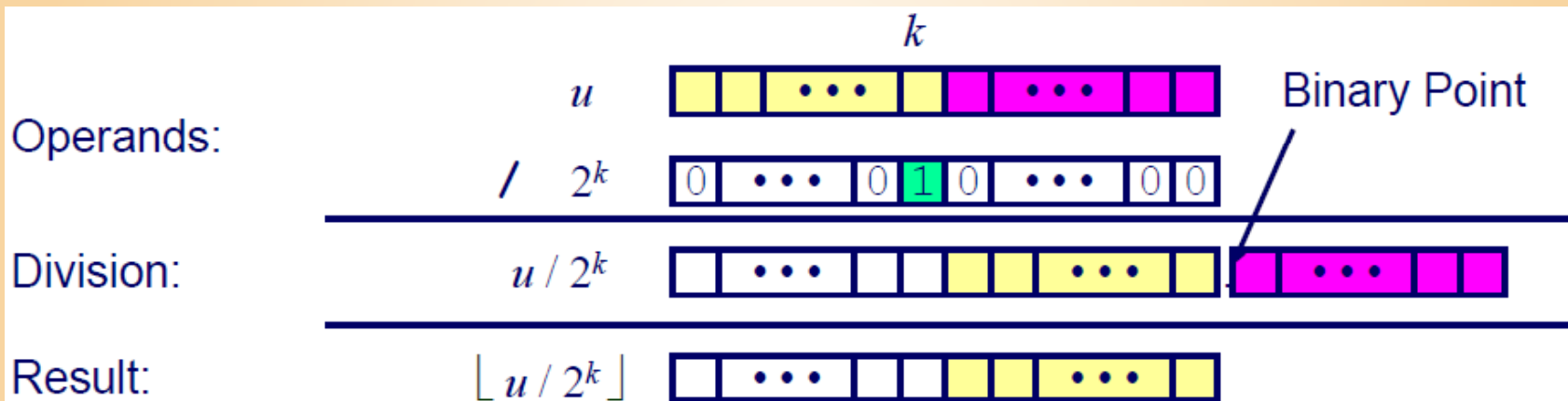
Explanation

```
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
 - $u \gg k$ gives $u / 2^k$
 - Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
$x \gg 1$	7606.5	7606	1D B6	00011101 10110110
$x \gg 4$	950.8125	950	03 B6	00000011 10110110
$x \gg 8$	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
```

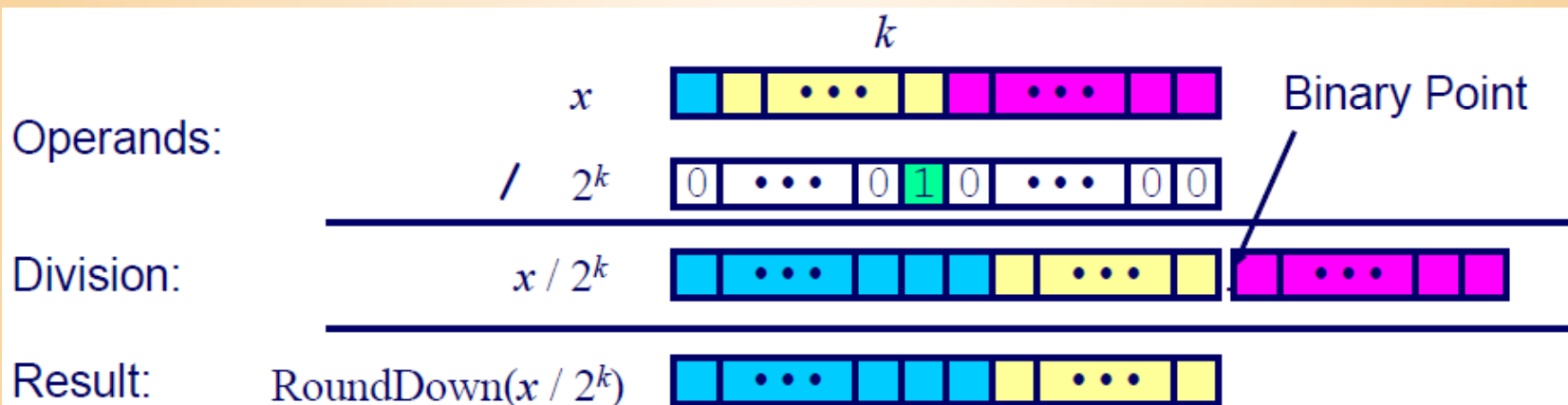
Explanation

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

- Uses logical shift for unsigned

Signed Power-of-2 Divide with Shift

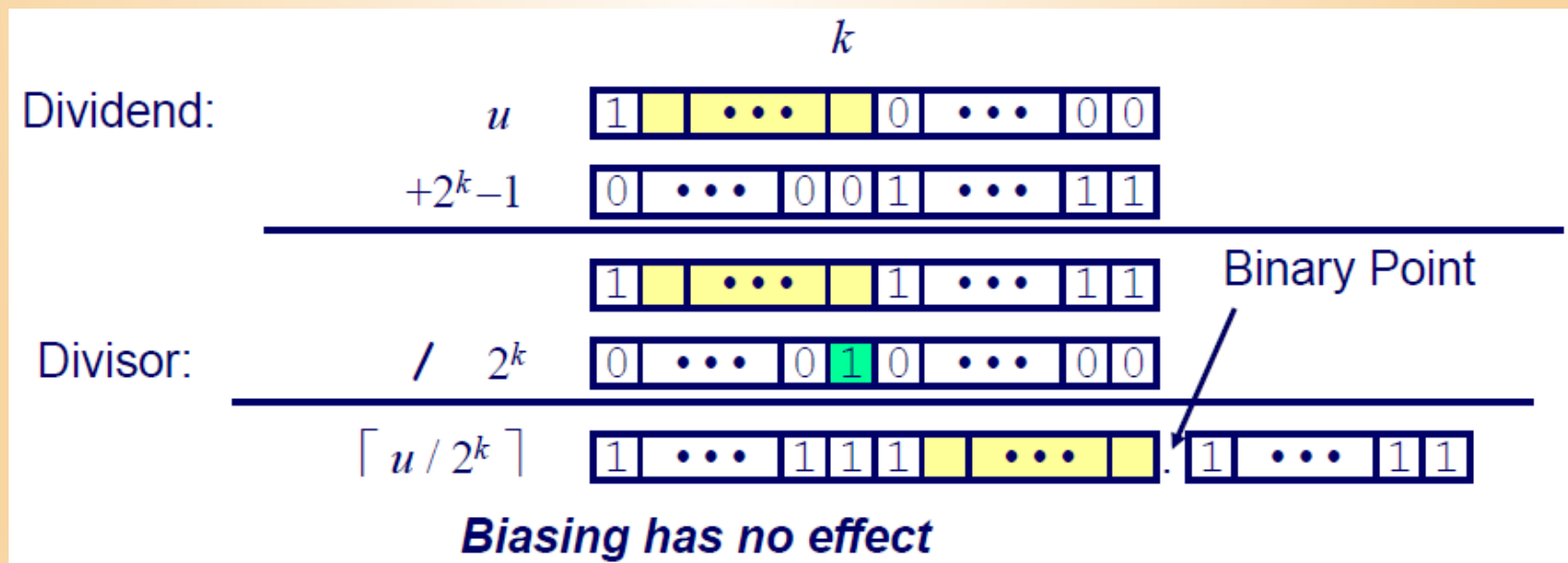
- Quotient of Signed by Power of 2
 - $x \gg k$ gives $x / 2^k$ instead of the round-towards-zero that standard integer division provides $-3/2=-1$
 - Uses arithmetic shift
 - Rounds wrong direction when $u < 0$



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	11100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	11111100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	11111111 11000100

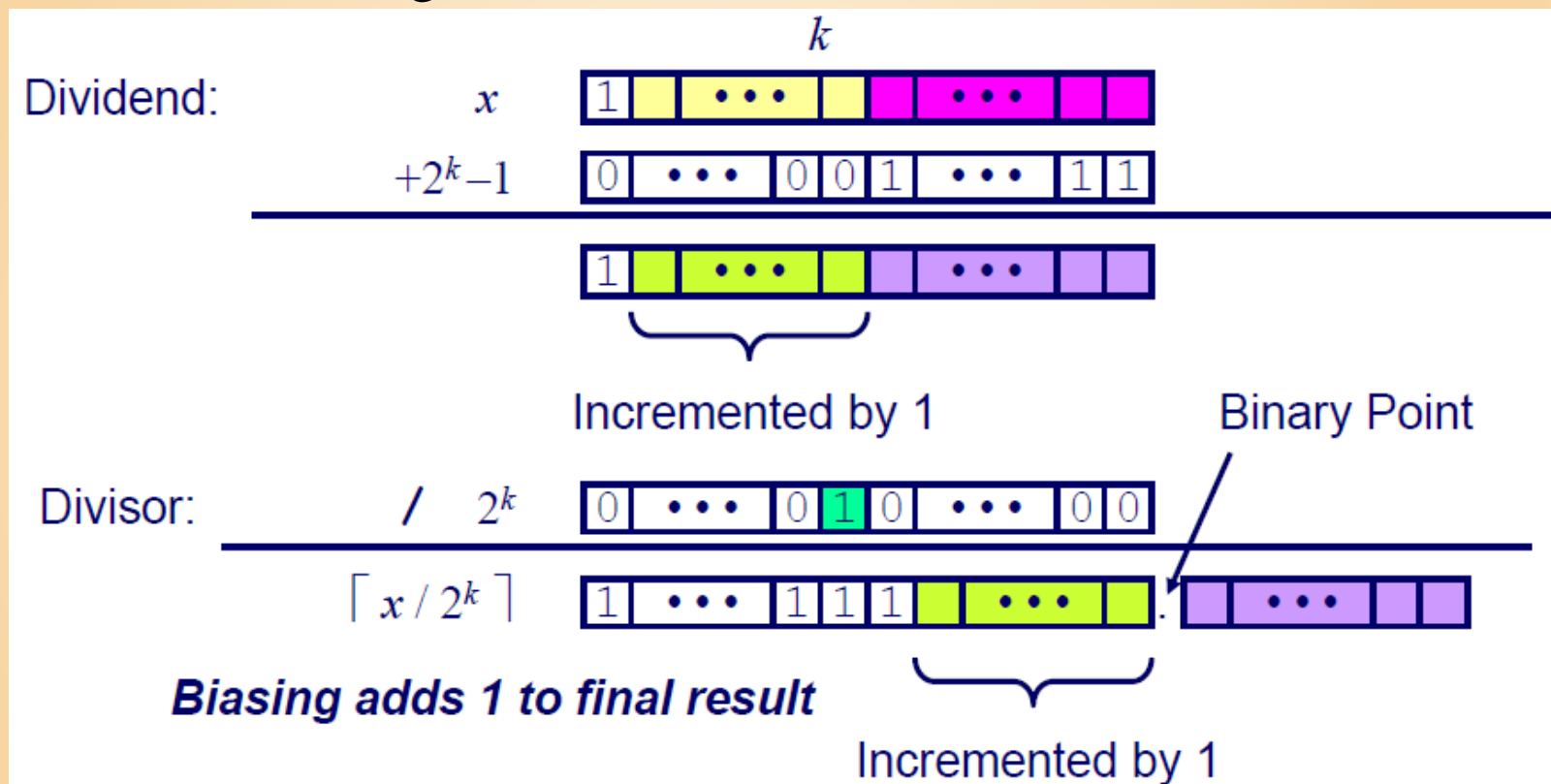
Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2 For negative number,
 - Want $x / 2^k$ (Round Toward 0)
 - Compute as $(x+2^k-1) / 2^k$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0
- Case 1: No rounding



Correct Power-of-2 Divide (cont)

- Case 2: Rounding



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
```

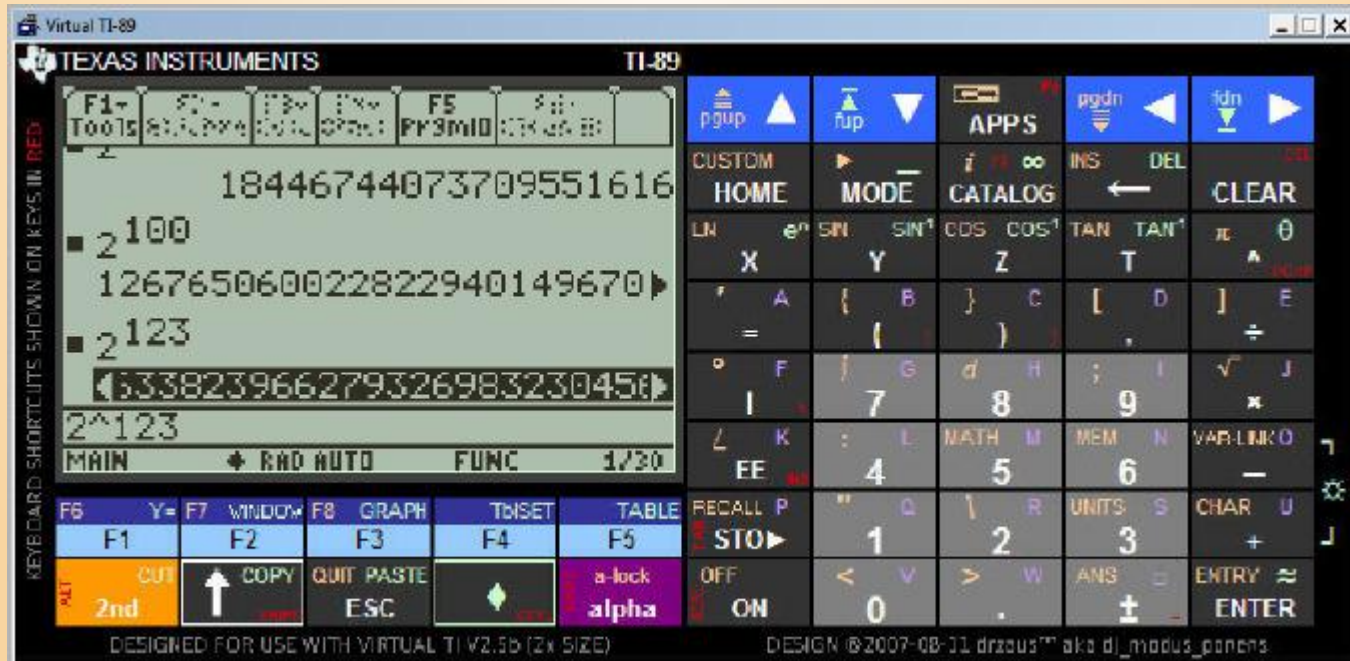
Explanation

```
#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
- 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
- $1101.0000 * 16 = 11010000 \quad S=16$
- $01.011000 * 64 = 01011000 \quad S=64$

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

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

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
 $5.7 \rightarrow 5$
 $-5.7 \rightarrow -5$
- Need Nearest Integer Rounding?
- Add +/- .5 before truncation depending on sign
 $5.7 + .5 = 6.2 \rightarrow 6$
 $5.4 + .5 = 5.9 \rightarrow 5$

 $-5.7 + .5 = -5.2 \rightarrow -5$
- The above equation doesn't work the same for negative numbers, need to subtract
 $-5.7 - .5 = -6.2 \rightarrow -6$
 $-5.4 - .5 = -5.9 \rightarrow -5$
- A/B gives $\text{floor}(A/B)$ rather than $\text{round}(A/B)$
- 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. $\text{BAUD_RATE} = \text{CLK_RATE} / \text{CLK_DIV}$
 - Option 1:
 - $\#define \text{CLK_DIV} (\text{CLK_RATE} / \text{BAUD_RATE})$
 - Option 2: BETTER CHOICE IS
 - $\#define \text{CLK_DIV} (\text{CLK_RATE} + (\text{BAUD_RATE} / 2)) / \text{BAUD_RATE}$

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 !!!!!
```