# Lecture 03
# Integer Operations

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2022

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O’Hallaron (CMU), Garcia, Weaver (UC Berkeley)

# Administrivia

- You should all have access to Campuswire and Gradescope
  - Contact me via email immediately if you don't!!
- Office hours are now running
  - See Canvas homepage for office hours times
  - Be sure to sign up on the queue that's also on Canvas
    - That's how we track what order to help people in

# Administrivia

- Homework 1 due by end-of-day Thursday
  - Submit on Gradescope

- Data Lab due next week Thursday
  - Start working on the Integer Puzzles now
  - Floating Point puzzles can wait until after lecture on Thursday
  - Can be tricky. Don’t spend forever on any one, jump around

# Today's Goals

- Explore operations we can perform on binary numbers
- Understand the edge cases of those operations
- Discuss performance of various operations

# C versus the hardware

- Operations you can perform on binary numbers have edge conditions
  - Usually going above or below the bit width

- If we say what happens in that scenario, it'll be what "the hardware" (i.e., a computer) does
  - In today's examples, pretty much every computer does the same thing

- That is not the same as what C does
  - Unclear choices are left as: **UNDEFINED BEHAVIOR** 😱
  - Which is to say, the compiler make any choice it wants

# Outline

- **Addition**
- Negation and Subtraction
- Multiplication
- Shifting
- Bit Masks
- Optimizations

# Unsigned Addition

- Like grade-school addition, but in base 2, and ignores final carry
  - If you want, can do addition in base 10 and convert to base 2. Same result!
- **Example: Adding two 4-bit numbers**

```
   1 1 1
   0101
 + 0011
 ------
   1000
```

  - $\mathbf{5_{10} + 3_{10} = 8_{10}}$ ✔

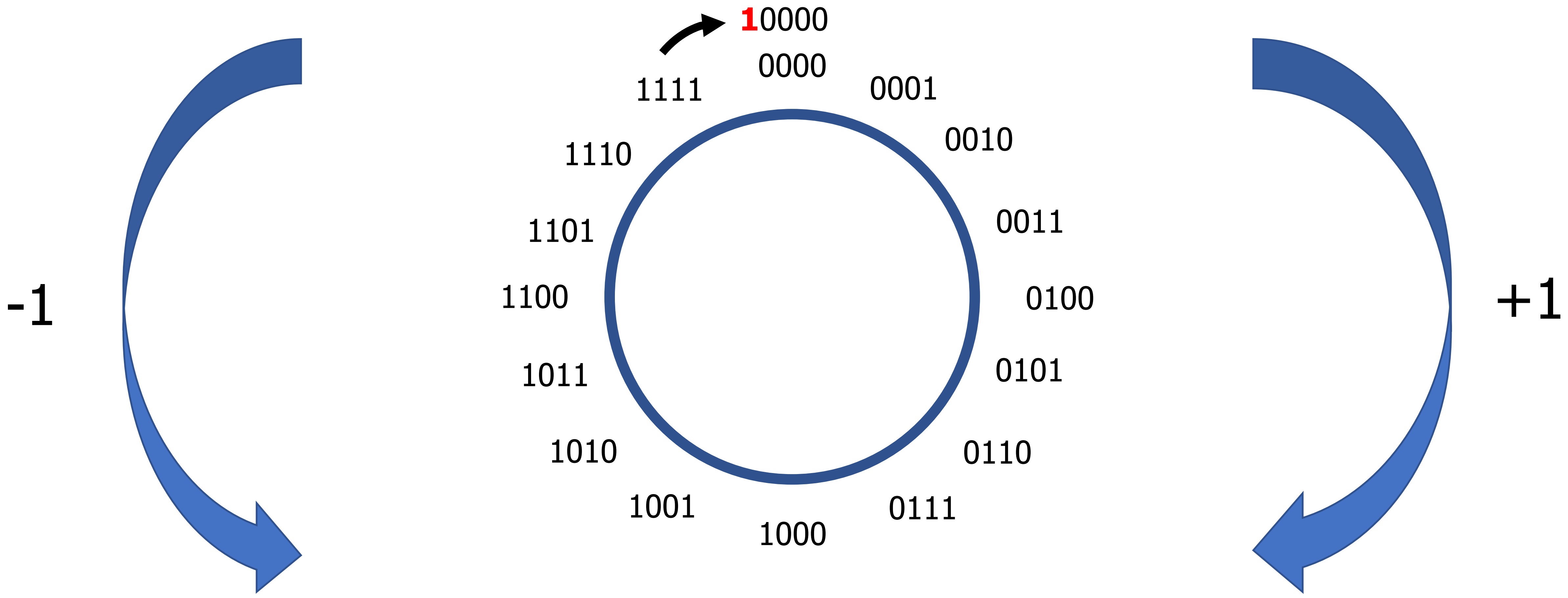
# Unsigned Addition and Overflow

- What happens if the numbers get too big?
- **Example: Adding two 4-bit numbers**

```
 1111
  1101
+ 0011
 ------
 10000
```

- **$13_{10} + 3_{10} = 16_{10}$**
  - Too large for 4 bits! Overflow
  - Result is the 4 least significant bits (all we can fit): so $0_{10}$
  - Gives us modular (= modulo) behavior: 16 modulo $2^4 = 0$

# Modulo behavior in binary numbers

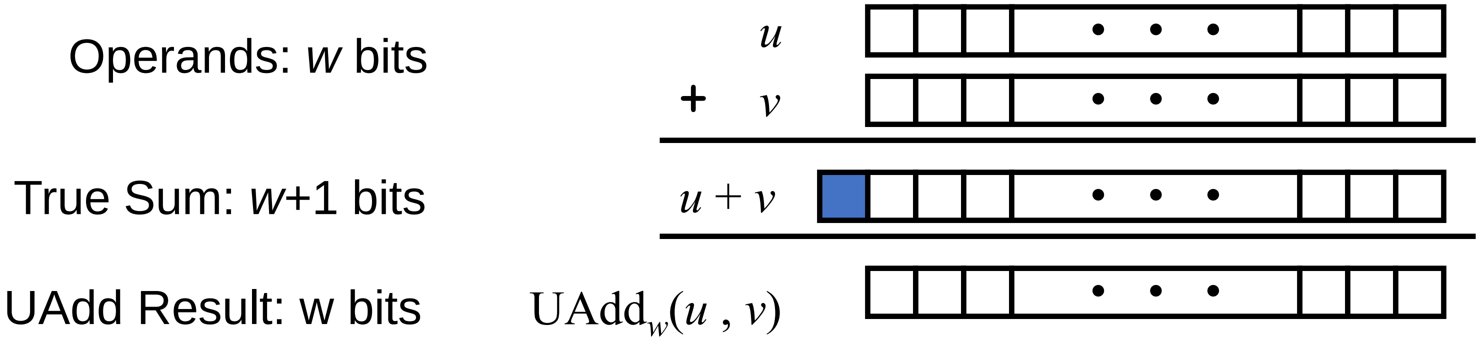


# Basis for unsigned addition

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

- Implements modular arithmetic
  - $UAdd_w(u, v) = (u + v) \mod 2^w$

- Need to drop carry bit, otherwise results will keep getting bigger
  - Example in base 10: $80_{10} + 40_{10} = 120_{10}$ (2-digit inputs become a 3-digit output!)




- Warning: C does not tell you that the result had an overflow!
  - Unsigned addition in C behaves like modular arithmetic

# Signed (2's Complement) Addition

- Works exactly the same as unsigned addition!
  - Just add the numbers in binary, and the result will work out

- Signed and unsigned sum have the exact same bit-level representation
  - Computers use the same machine instruction and the same hardware!
  - That's a big reason 2's complement is so nice! Shares operations with unsigned

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

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

  - Result: `s == t` (in all cases)

# Signed addition example

- Same addition method as unsigned
- **Example: Adding two 4-bit signed numbers**

```
      1 1
     1011    (-8 + 3 = -5)
  +  0011    (         +3)
     ----
     1110    (-8 + 6 = -2)
```

- **$-5_{10} + 3_{10} = -2_{10}$** ✔

# Combining negative and positive numbers

- Overflow sometimes makes signed addition work!
- **Example: Adding two 4-bit signed numbers**

```
 1 1 1 1
  1101   (-8 + 5 = -3)
+ 0011   (        +3)
 ------
 10000
```

- **$-3_{10} + 3_{10} = 0_{10}$**
  - Too large for 4 bits! Drop the carry bit
  - Result is what we expect as long as we truncate

# Signed addition and overflow

- Overflow can still happen in signed addition though
- **Example: Adding two 4-bit signed numbers**

```
  1 1 1
   0101
 + 0011
 ------
   1000
```

- $\mathbf{5_{10} + 3_{10} = -8_{10}}$ (+8 is too big to fit)
- Remember, this was also unsigned $\mathbf{5_{10} + 3_{10} = 8_{10}}$

# Signed addition and underflow

- Underflow happens in the negative direction
- **Example: Adding two 4-bit signed numbers**

```
1  1 1
  1011
+ 1011
------
 10110
```

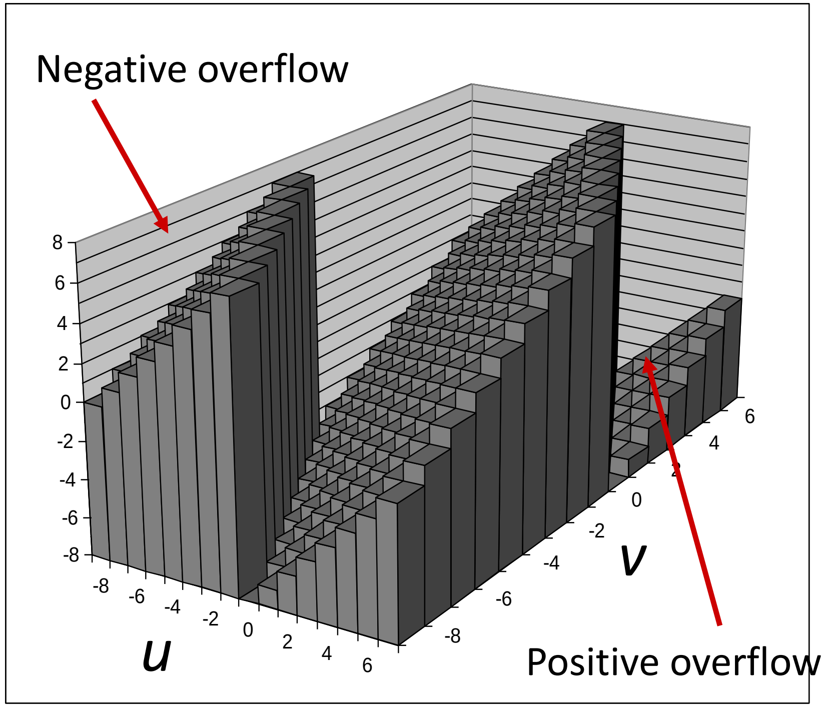
- **$-5_{10}$ + $-5_{10}$ = $+6_{10}$** (-10 was too small to fit)

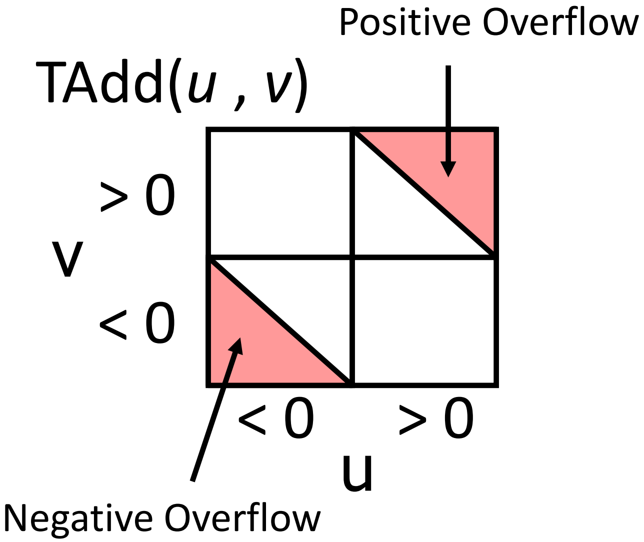
# TAdd Overflow

- Can overflow two ways!
  - By going too far into the positives
  - *OR* too far into the negatives!
  - Modular behavior either way

- *BUT*, beware signed overflow in C
  - **UNDEFINED BEHAVIOR**
  - Compiler *probably* does modular result

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



# Special boss in Chrono Trigger


- Dream Devourer
  - Special boss in the Nintendo DS edition
- Wanted to make it even more challenging
  - 32000 hit points
  - Takes *forever* to defeat
- Hit points stored as a 16-bit signed integer
  - Range: -32768 to +32767

# Chrono Trigger signed overflow bug

- Solution: heal it

- Hit points go negative and it dies

# Outline

- Addition
- **Negation and Subtraction**
- Multiplication
- Shifting
- Bit Masks
- Optimizations

# Negating with Complement & Increment

- Claim: The following is true for 2's complement
  - $\sim x + 1 == -x$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **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 |

- Complement
  - Observation: $\sim x + x == 1111\ldots11_2 == -1$

- Increment
  - $\sim x + 1 == \sim x \boldsymbol{+ x} \boldsymbol{- x} + 1 == \cancel{-1} - x + \cancel{1} == -x$
  - $\sim x + 1 == -x$

- Example, 4 bits: $6_{10} = 0110_2$
  - Complement: $1001_2 \rightarrow$ Increment $= 1010_2 = -8 + 2 = -6_{10}$

# Subtraction in two's complement

- Subtraction becomes addition of the negative number
  - 5 – 3 = 5 + -3 = 2

- Unsigned subtraction
  - Convert subtractor to its two's complement negative form
  - Do addition
  - Treat result as an unsigned number

```
 1 1  1
   0101    (+5)
 + 1101    ( -3)
  ------
  10010
```

# Question + Break

- In 8-bit two's complement binary:
  - **What is $120_{10} – 20_{10}$?**
    - Solve as decimal. Then translate

# Question + Break

- In 8-bit two's complement binary:
  - **What is $120_{10} - 20_{10}$?**
    - Solve as decimal. Then translate
    - $100_{10} = 01100100_2$

# Question + Break

- In 8-bit two's complement binary:

  - **What is $120_{10} - 20_{10}$?**
    - Solve as decimal. Then translate
    - $100_{10} = 01100100_2$

  - **What is 0x84 - 0x20?**
    - Solve as hexadecimal. Then translate

# Question + Break

- In 8-bit two's complement binary:
  - **What is $120_{10} – 20_{10}$?**
    - Solve as decimal. Then translate
    - $100_{10} = 01100100_2$
  - **What is 0x84 - 0x20?**
    - Solve as hexadecimal. Then translate
    - 0x64 = 0b01100100

# Outline

- Addition
- Negation and Subtraction
- **Multiplication**
- Shifting
- Bit Masks
- Optimizations

# Multiplication

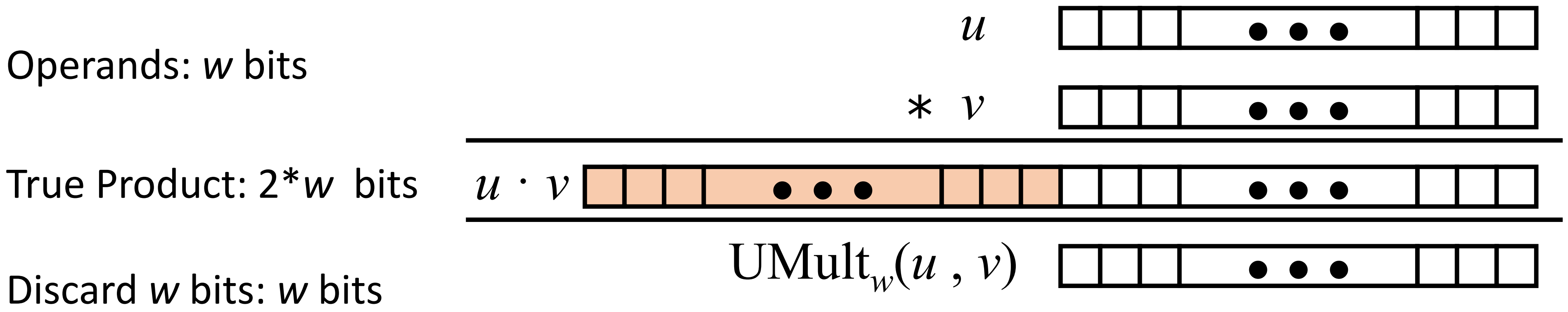
- Goal: Compute the Product of ***w***-bit numbers *x*, *y*
  - Either signed or unsigned

- But, exact results can be bigger than ***w*** bits
  - Around double the size (2*w*), in fact!

  - Example in base 10: $50_{10} * 20_{10} = 1000_{10}$
    - (2-digit inputs become a 4-digit output!)

- As with addition, result is truncated to fit in ***w*** bits
  - Because computers are finite, results can't grow indefinitely

# Unsigned Multiplication




- **Standard Multiplication Function**
  - Equivalent to grade-school multiplication
  - But ignores most significant *w* bits of the result
  - As a person, we can do base 10 multiplication, convert to base 2, then truncate

- Implements modular arithmetic like addition does

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

# Unsigned multiplication

- **Example: Multiplying two 4-bit numbers**

```
     0010
   x 0101
   ------
     0010
    00000
   001000
+ 0000000
---------
  0001010
```

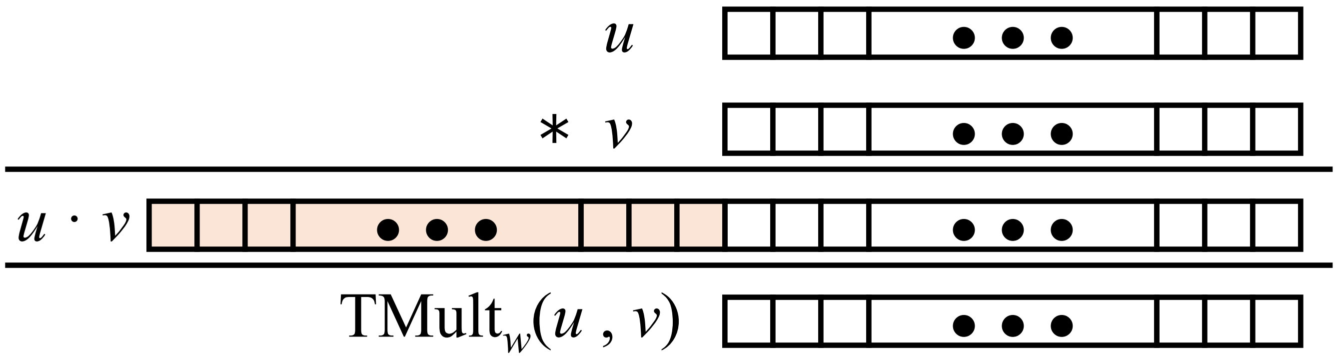
$$\mathbf{2_{10} * 5_{10} = 10_{10}}$$ ✔

# Signed (2’s Complement) Multiplication

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits




- **Standard Multiplication Function**
  - Ignores most significant *w* bits
  - Lower bits still give the correct result
    - So we can use same machine instruction for both!
    - Again, that’s one reason why 2’s complement is so nice

- **In C, signed overflow is undefined**
  - ...but probably you’ll see the two’s complement behavior

# Signed multiplication

- **Example: Multiplying two's complement 5-bit numbers**

```
    11110    -2
 x  00011     3
   ------
    11110
 + 111100
 --------
 ~~10~~11010
```

What are these two 5-bit numbers?

What is the result of this addition?

$$\mathbf{-2_{10} * 3_{10} = -6_{10}}$$ ✔

# Outline

- Addition
- Negation and Subtraction
- Multiplication
- **Shifting**
- Bit Masks
- Optimizations

# Left Shift: `x << y`

- Shift bit-vector x left by y positions
  - Throw away extra bits on left
  - Fill empty bits with 0
    - Same behavior for signed or unsigned

| Argument `x` | 00000010 |
|---|---|
| `<< 3` | ~~000~~00010<u>*000*</u> |

| Argument `x` | 10100010 |
|---|---|
| `<< 3` | ~~101~~00010<u>*000*</u> |

- Equivalent to multiplying by $2^y$
  - And then taking modulo (i.e. truncating overflow bits)

- Undefined behavior in C when:
  - `y < 0`, or `y ≥ bit_width(x)`
  - Also when some non-0 bits get shifted off (*probably* they get truncated)

# Right Shift: `x >> y`

- Shift bit-vector x right y positions
  - Throw away extra bits on right
- But how to fill the new bits that open up?
  - Will depend on signed vs unsigned
- Unsigned: **Logical shift**
  - Always fill with 0's on left
- Signed: **Arithmetic shift**
  - Replicate most significant bit on left
  - Necessary for two's complement integer representation (sign extension!)
- Undefined behavior in C when:
  - `y < 0`, or `y ≥ bit_width(x)`

| Argument `x` | 01100010 |
|---|---|
| Logi. >> 2 | 00011000 |
| Arith. >> 2 | 00011000 |

| Argument `x` | 10100010 |
|---|---|
| Logi. >> 2 | 00101000 |
| Arith. >> 2 | 11101000 |

# Practice shifting in C

`unsigned char x = 0b10100010;`

`x << 3 = ?` `0b00010000`

Steps:
0b10100010**000**
0b~~101~~00010**000**

`unsigned char x = 0b10100010;`

`x >> 2 = ?` `0b00101000`

Steps:
0b**00**10100010
0b**00**101000~~10~~

`signed   char x = 0b10100010;`

`x >> 2 = ?` `0b11101000`

Steps:
0b**11**10100010
0b**11**101000~~10~~

**Note:**
GCC supports the prefix **`0b`** for binary literals (like **`0x`**… for hex) directly in C.
This is not part of the C standard! It may not work on other compilers.

# Outline

- Addition
- Negation and Subtraction
- Multiplication
- Shifting
- **Bit Masks**
- Optimizations

# Bit Masking

- How do you manipulate certain bits within a number?

- Combines some of the ideas we've already learned
    - ~, &, |, <<, >>

- Steps
    1. Create a "bit mask" which is a pattern to choose certain bits
    2. Use & or | to combine it with your number
    3. Optional: Use >> to move the bits to the least significant position

# Bit mask values

- Selecting bits, use the AND operation
  - 1 means to select that bit
  - 0 means to not select that bit

Select bottom four bits:

```
num & 0x0F
```

- Writing bits
  - Writing a one, use the OR operation
    - 1 means to write a one to that position
    - 0 is unchanged
  - Writing a zero, use the AND operation
    - 0 means to write a zero to that position
    - 1 is unchanged

Set $6^{th}$ bit to one:

```
num | (1 << 6)
num | (0b01000000)
```

Clear $6^{th}$ bit to zero:

```
num & (~(1 << 6))
num & (~(0b01000000))
num & (0b10111111)
```

# Example: selecting bits

- Select bits 2 and 3 from a number

```
Input: 0b01100100
 Mask: 0b00001100
```

```
  0b01100100
& 0b00001100
  ----------
  0b00000100
```

Finally, shift right by two to get the values in the least significant position:

```
0b00000001
```

# Outline

- Addition
- Negation and Subtraction
- Multiplication
- Shifting
- Bit Masks
- **Optimizations**

# What about division?

- Similar to long division process
  - Tedious and complicated to get right

- Even more complicated than multiply to make work in hardware
  - I’ve worked on a computer that didn’t even have divide

# Concept: Not all operations are equally expensive!

- Some operations are pretty simple to perform in hardware
  - E.g., addition, shifting, bitwise operations
  - Also true of doing the same by hand on paper

- Others are much more involved
  - E.g., multiplication, or even more so division
  - Consider long multiplication / long division; quite tedious!
  - Hardware is not doing the exact same thing, but similar principle

- ***Trick*: try to replace expensive operations with simple ones!**
  - Doesn’t work in all cases, but often does when mult/div by constants

# Multiplication as shift operations

- Multiply 2 x 5:

$$
\begin{array}{r}
0010 \\
\times\ \ 0101 \\
\hline
0010 \\
00000 \\
001000 \\
+\ \ 0000000 \\
\hline
\cancel{000}1010
\end{array}
$$

- This is actually just bit shifts and additions

- 2 x 5 = (2 << 0) + (2 << 2)

  = 2 + 8

  = 10

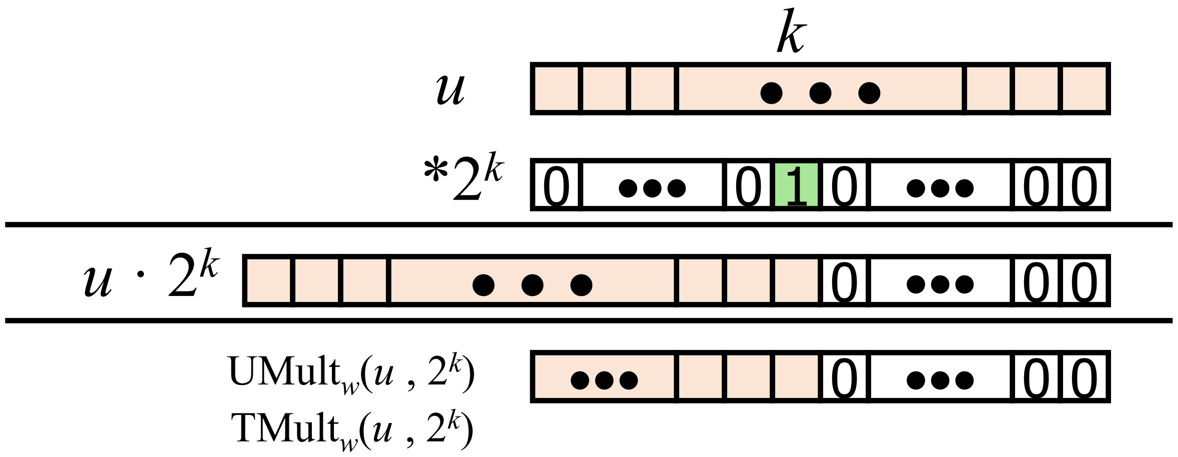
# Power-of-2 Multiply with Left Shift

- **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 * 32 - u * 8 = u * 24`
    - Can combine multiple shifts with addition to get multiplications by non-powers-of-2

# Shift to divide

- Division works too
  - unsigned int x = y / 2; unsigned int x = y >> 1;
- Even more important because division is a complicated operation
  - Multiply is implemented in (relatively) simple hardware on most systems
  - Compiler might actually translate your divide by powers of two into shift operations though!
- Warning: rounding needs to be handled correctly for signed numbers and division
  - See bonus slides

# Compilers automatically chose the best operations

- Should you use shifts instead of multiply/divide in your C code?
  - **NO**

- Just write out the math
  - Math is more readable if that's what you meant
  - Compiler automatically converts code for you for best performance

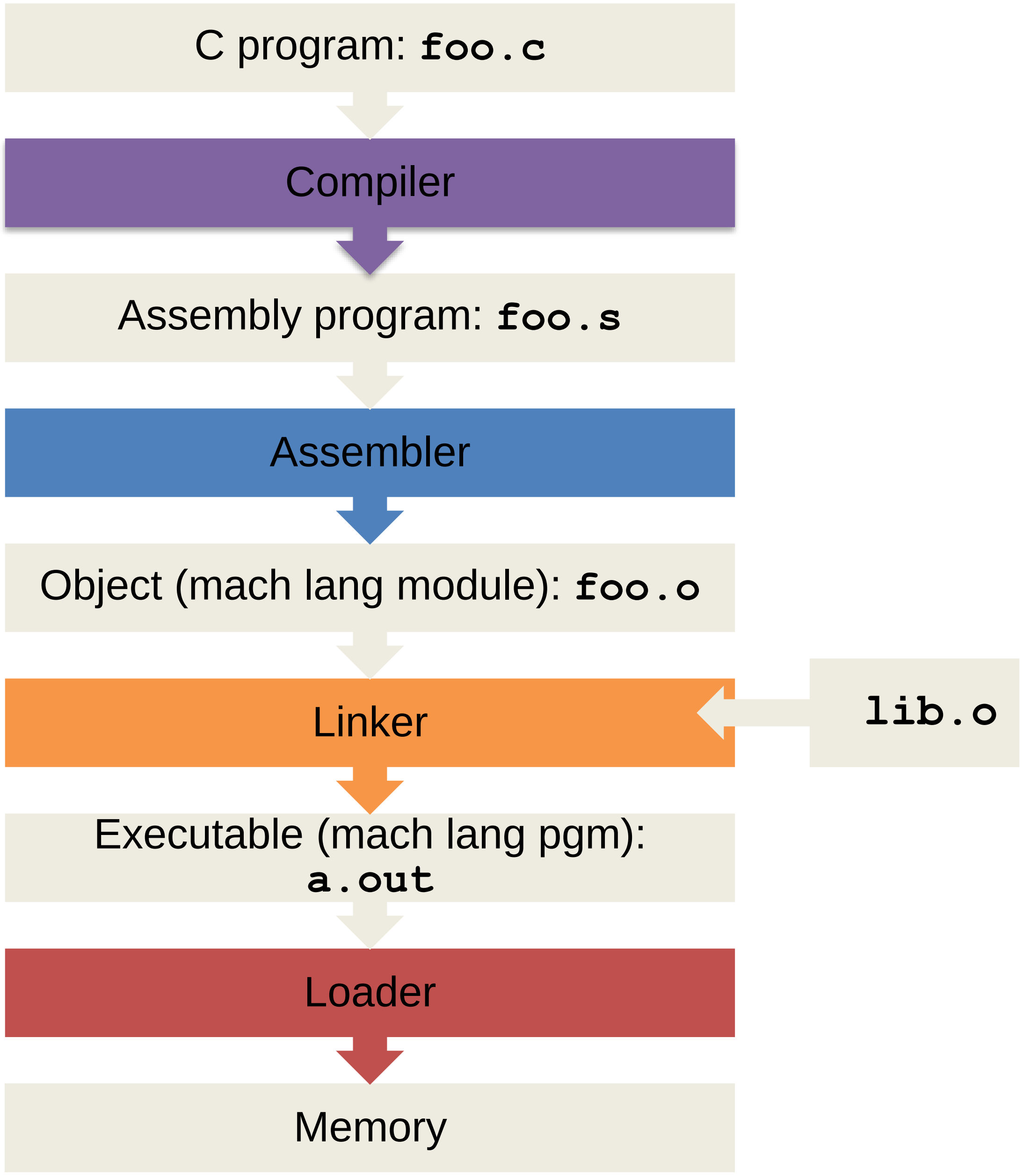
- These two mean the same thing, but one is way more understandable
  - `int x = y * 32;`
  - `int x = (y << 5);`

# C code translation

- Steps for C

**CALL**

1. **C**ompiler
2. **A**ssembler
3. **L**inker
4. **L**oader



# Compiler

- Input: higher-level language code (C, C++, Java, etc.)
- Output: assembly language code (for a particular computer)

- Process
  - Handle pre-processor (defines and includes)
  - Preform optimizations on code
    - Make it faster (such as divide-into-shift)
    - Make it use less memory (eliminate unused variables)

- Entire course worth of material here: CS322

# Outline

- Addition
- Negation and Subtraction
- Multiplication
- Shifting
- Bit Masks
- Optimizations

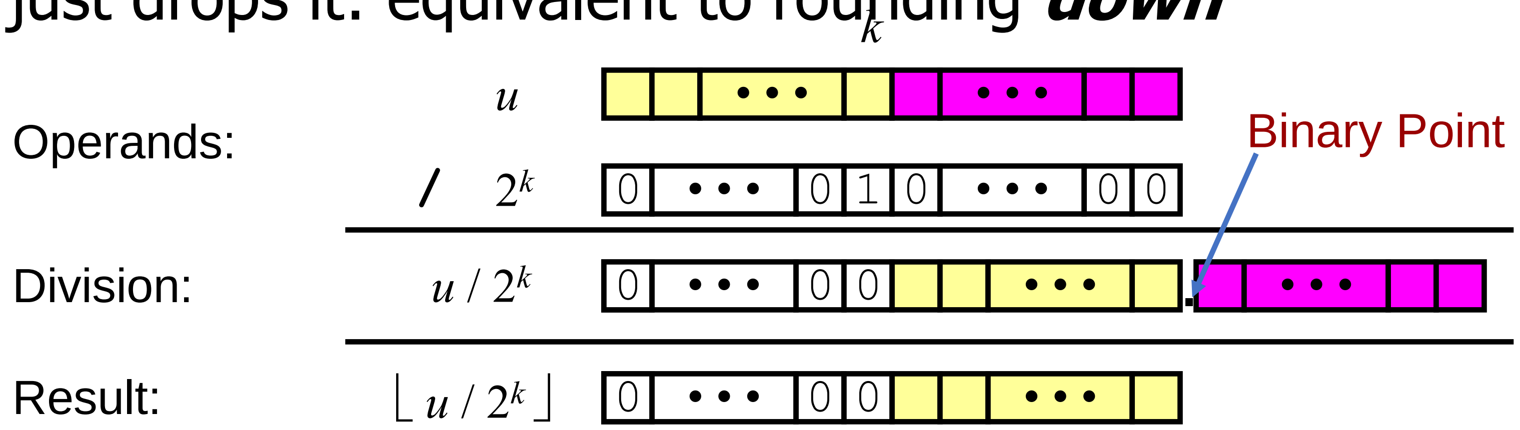
# Outline

- Dividing with bit shift

# Unsigned Power-of-2 Divide with Right Shift

$\lfloor x \rfloor$ : round x down
$\lceil x \rceil$ : round x up

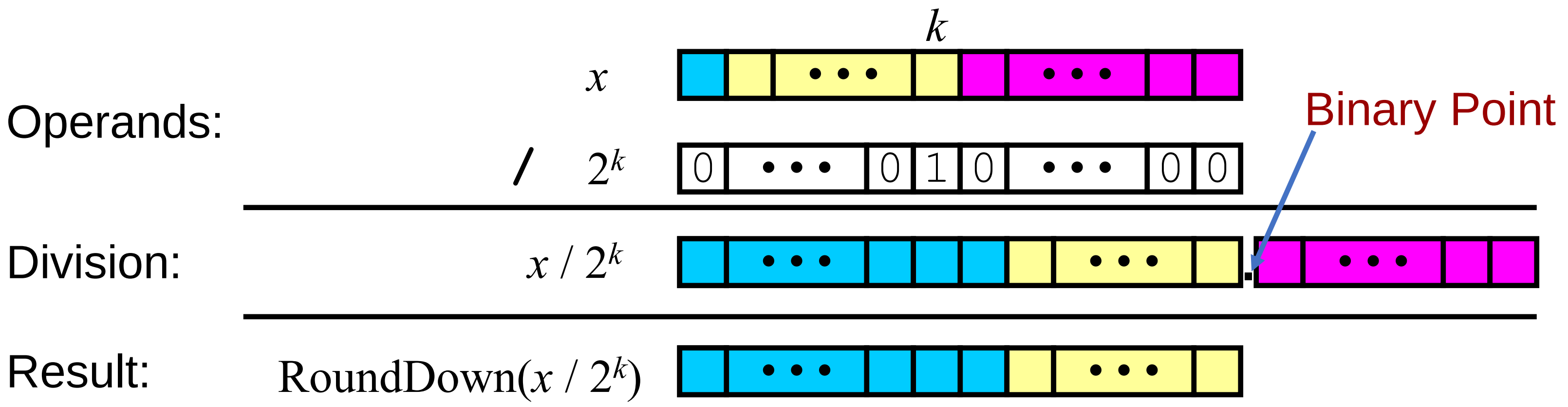
- **Quotient of unsigned by power of 2**
  - **`u >> k`** gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift
  - Pink part would be remainder / fractional part (right of the point)
    - Shift just drops it: equivalent to rounding ***down***




| | 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 (Almost)

- **Quotient of signed by power of 2**
  - `x >> k` gives $\lfloor \mathbf{x} / \mathbf{2^k} \rfloor$
  - Uses arithmetic shift
  - Also rounds down, again by dropping bits
    - But signed division should round ***towards 0!*** (that's its math definition)
    - That means rounding ***up*** for negative numbers!

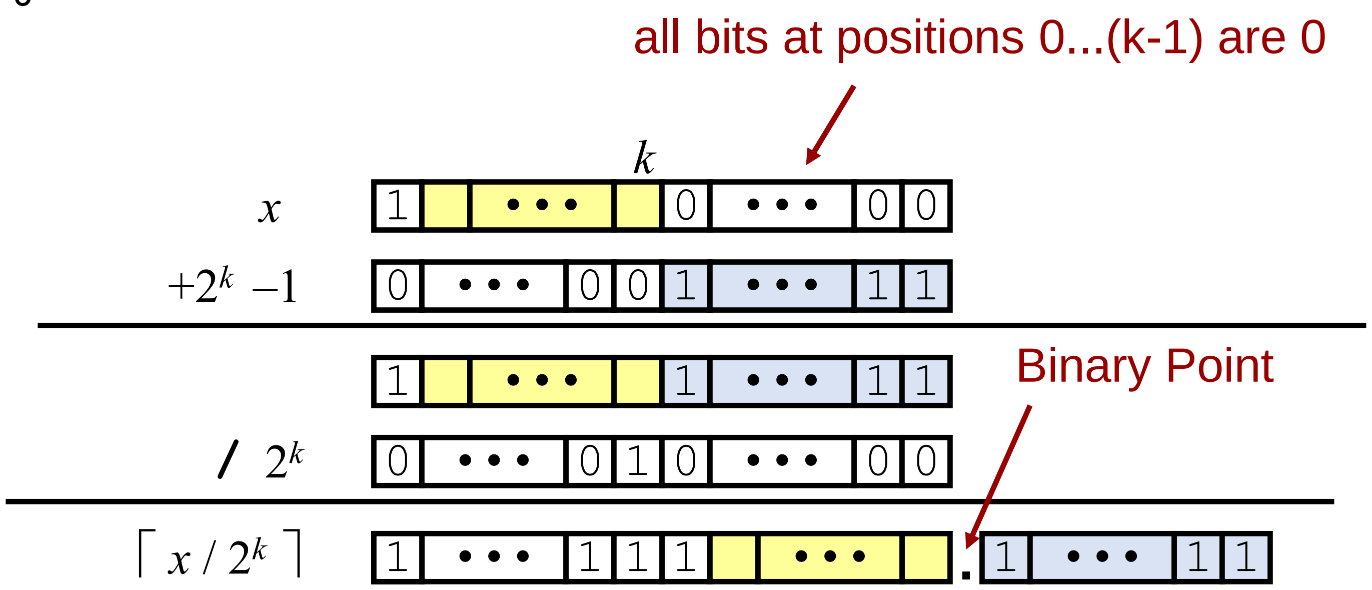



- **Example, 4 bits: -6 / 4 = -1.5  (should round towards 0, to -1)**
  - $1010_2$ >> 2 = $\textcolor{red}{\mathit{11}}10_2 = -2_{10}$
  - Rounds the wrong way!

# Correct Signed Power-of-2 Divide

- Want $\lceil$ **x / 2$^k$** $\rceil$ (round towards 0)
  - Math identity: $\lceil$**x / y**$\rceil$ **=** $\lfloor$ **(x + y - 1) / y** $\rfloor$
  - Compute negative case as $\lfloor$ **(x+2$^k$–1) / 2$^k$** $\rfloor$ → gets us correct rounding!
  - Computing both cases in C: `(x<0 ? (x + (1<<k)-1) : x) >> k`
    - Biases dividend toward 0

- **Case 1: No rounding**

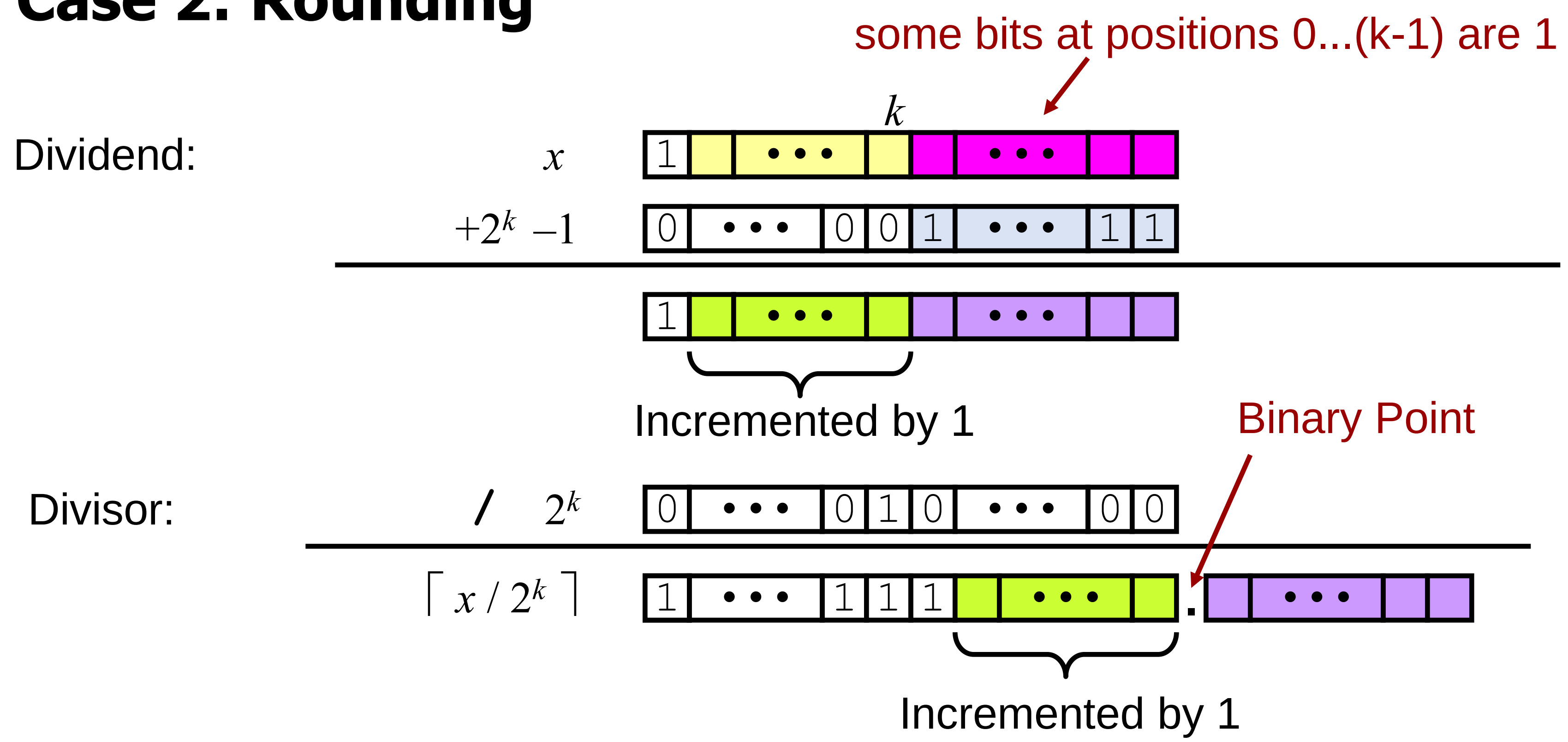



***Biasing has no effect; all affected bits are dropped***

- **Example, 4 bits: -8 / 2$^2$ = -2** **bias = (1<<2)-1 = 3**
  - (1000 + 0011) >> 2 = 1011 >> 2 = *11*10 = $-2_{10}$ (correct, no rounding)

# Correct Signed Power-of-2 Divide (Cont.)

## Case 2: Rounding




***Biasing adds 1 to final result; just what we wanted***

- **Example, 4 bits: -6 / $2^2$ = -1 bias = (1<<2)-1 = 3**
  - (1010 + 0011) >> 2 = 1101 >> 2 = *11*11 = $-1_{10}$ (correct, rounds towards 0)
- **Compiler does that for you (but you need to be able to read it!)**