Lecture 03
Data Operations

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2023

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

Northwestern

Administrivia

- You should all have access to Piazza and Gradescope
  - Contact me via email immediately if you don't!!

- Office hours are now running
  - See Canvas homepage for office hours times
  - Mix of in-person (M/W/F) and online (T/R) hours
  - Today: Elder 30 2-4pm, Tech M128 5-8pm
    - Each staffed by 3+ PMs

Administrivia

- Homework 1 due by end-of-day Wednesday
  - Submit on Gradescope
- Pack Lab should be out later today!
  - Sometime this evening
- You'll do Pack Lab on one of the EECS servers
  - Usually we use Moore, but any should be fine for this lab
  - SSH + Command Line interface
  - See Piazza post for details on remote access
  - See Piazza post for tutorials on command line

Today's Goals

- Explore operations we can perform on integers and more generally on binary numbers
- Understand the edge cases of those 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 can make any choice it wants

Outline

- **Integer Operations**
  - **Addition**
  - Negation and Subtraction
  - Multiplication and Division

- Binary Operations
  - Boolean Algebra
  - Shifting
  - Bit Masks

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
```

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

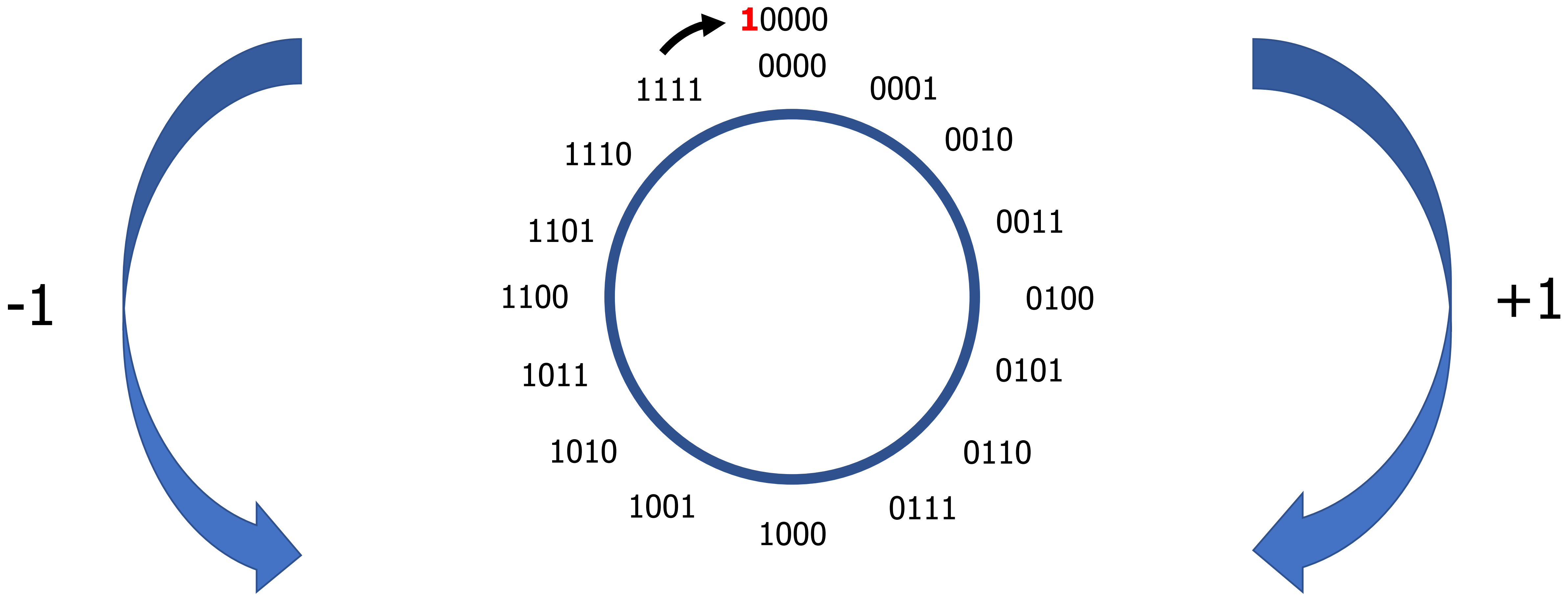
Unsigned Addition and Overflow

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

```
 1 1 1 1
   1101
+  0011
  -----
  10000
```

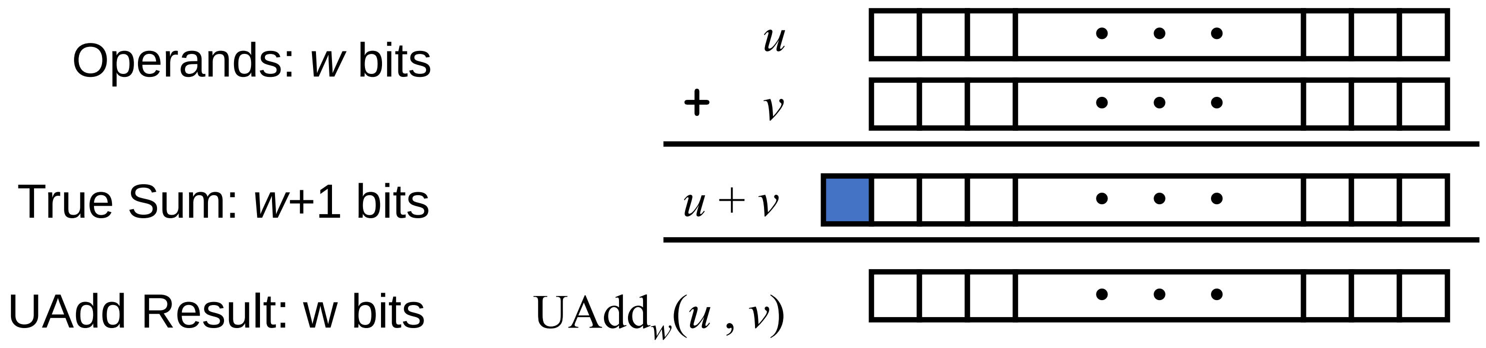
- **13_{10} + 3_{10} = 16_{10}**
  - Too large for 4 bits! Overflow
  - Result is the 4 least significant bits (all we can fit): so 0_{10}
  - Gives us modular (= modulo) behavior: 16 modulo 2^4 = 0

Modulo behavior in binary numbers



Unsigned addition is modular

- Implements modular arithmetic
  - $\text{UAdd}_w(u, v) \quad = \quad (u + v) \bmod 2^w$
- 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 silently 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

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**

```
  111
  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 negative overflow

- Overflow also 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)

Overflow: hardware vs C standard

- Hardware implementations for unsigned and signed addition are the same
  - Both implement modular arithmetic

- Unsigned overflow in C is defined as modular arithmetic

- Signed overflow in C is **UNDEFINED BEHAVIOR**
  - Compiler *probably* does modular result
  - But there are no promises about this and it can make *assumptions*
  - So don’t rely on it

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
- **How do speedrunners defeat the boss?**

Chrono Trigger signed overflow bug

- Solution: heal it
- Hit points go negative and it dies

Outline

- **Integer Operations**
  - Addition
  - **Negation and Subtraction**
  - Multiplication and Division
- Binary Operations
  - Boolean Algebra
  - Shifting
  - Bit Masks

Negating a number

- In C:
  - `x = -y;`

- Operation
  - Determine the negative, signed version of the number (two’s complement)
  - Hardware method: flip it and add one

Negating via Complement & Increment

- Claim: The following is true for 2’s complement
  - ~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: ~x + x == 1111…11$_2$ == -1

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

- Both unsigned and signed subtraction
  - Convert subtractor to its two's complement negative form
    - i.e., negate it
  - Then do addition
  - Treat result as an unsigned number

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

C rules vs hardware rules

- Exact same overflow rules apply

- Unsigned subtraction can wrap below zero to make a large number
  - Modular arithmetic

- Signed subtraction is **UNDEFINED BEHAVIOR**
  - And therefore should not be trusted

Break + practice

- Adding two 8-bit numbers:
  - Also determine the decimal version of the result

```
  00010101
+ 10110001
```

Break + practice

- Adding two 8-bit numbers:
  - Also determine the decimal version of the result

```
    1 1   1
   00010101
 + 10110001
 ----------
   11000110
```

	Unsigned encoding		Signed encoding
00010101	16+4+1 = 21		16+4+1 = 21
+ 10110001	128+32+16+1 = 177	OR	-128+32+16+1 = -79
11000110	128+64+4+2 = 198		-128+64+4+2 = -58

Break + practice

- Adding two 8-bit numbers:
  - Also determine the decimal version of the result

```
   1 1    1
  00010101
+ 10110001
  --------
  11000110
```

	Unsigned encoding		Signed encoding
00010101	16+4+1 = 21		16+4+1 = 21
+ 10110001	128+32+16+1 = 177	OR	-128+32+16+1 = -79
11000110	128+64+4+2 = 198		-128+64+4+2 = -58

What about unsigned subtraction 21-79?

That would apply modulo arithmetic and result in the value 198

Outline

- **Integer Operations**
  - Addition
  - Negation and Subtraction
  - **Multiplication and Division**

- Binary Operations
  - Boolean Algebra
  - Shifting
  - Bit Masks

Multiplication

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

- But, exact results can be bigger than ***w*** bits
  - 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!)

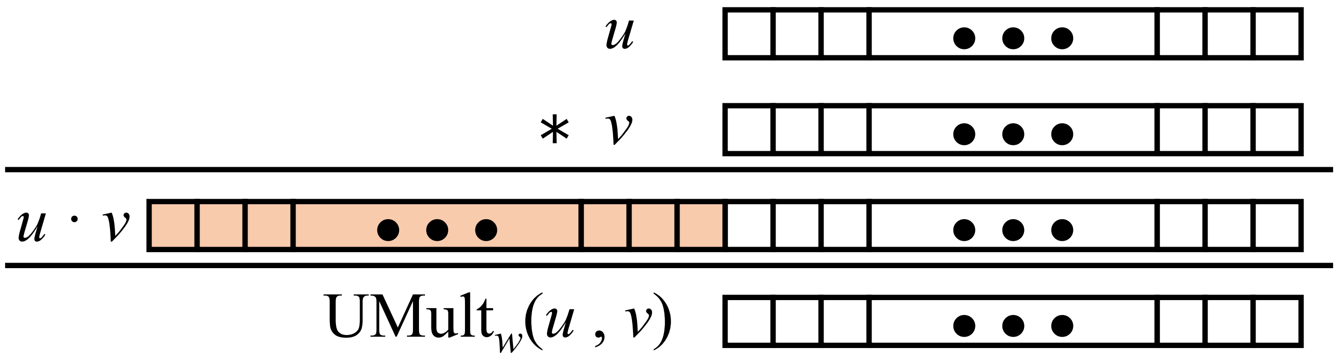
- As with addition, result is truncated to fit in ***w*** bits
  - Because computers are finite, results can't grow indefinitely

Unsigned Multiplication

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



- **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

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

Unsigned multiplication

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

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

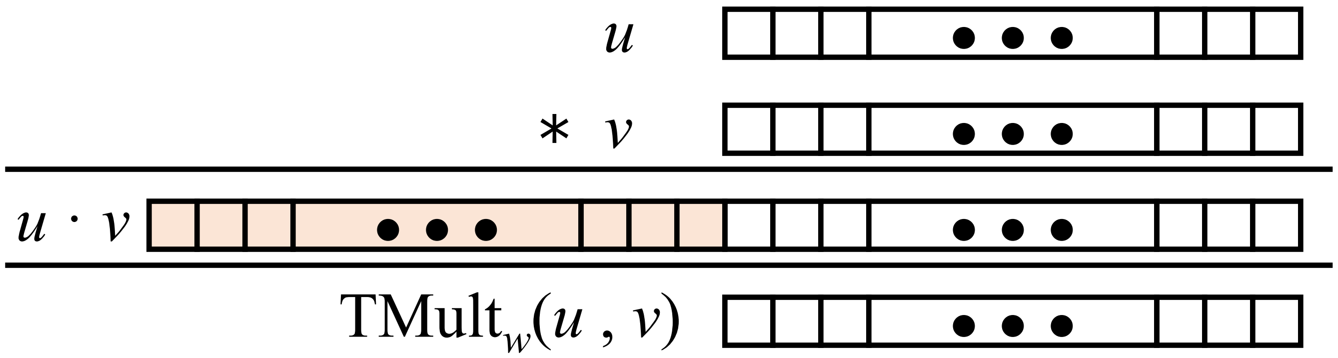
$\mathbf{2_{10} * 5_{10} = 10_{10}}$ ✓

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
---------
  1011010
```

What are these two 5-bit numbers?

What is the result of this addition?

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

What about divide?

- Annoying operation, not going to discuss in this class
  - Similar to long division process
  - Tedious and complicated to get right

- I’ve worked on computers that don’t have hardware support for division at all!!

- Important thing to remember is that integers don’t have fractional parts
  - In C: `1/2 == 0`
  - We’ll need a new encoding for fractional numbers: floating point

Outline

- Integer Operations
  - Addition
  - Negation and Subtraction
  - Multiplication and Division

- **Binary Operations**
  - **Boolean Algebra**
  - Shifting
  - Bit Masks

Boolean algebra

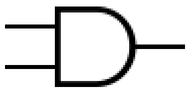
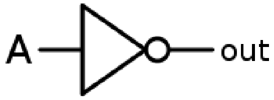
- You've programmed with **`and`** and **`or`** in earlier classes
  - Written **`&&`** and **`||`** in C and C++

- **Boolean algebra is a generalization of that**
  - A mathematical system to represent (propositional) logic
  - 2 truth values: true = **`1`**, false = **`0`**
  - 3 operations: and = **`&`**, or = **`|`**, not (or complement) = **`~`**

Performing Boolean algebra

- **Follow the rules for each operation to compute results**
  - Rules are the like those you know from programming

- OR: | AND: &   NOT: ~   1: True   0: False

(1 | 0) & 0 → 1 & 0 → 0

(1 & 1) & ~(0 | 0) → 1 & ~(0) → 1 & 1 → 1

Truth tables for Boolean algebra

- For each possible value of each input, what is the output
  - Column for each input
  - Column for the output operation

~A

A	~A
0	1
1	0

A | B

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

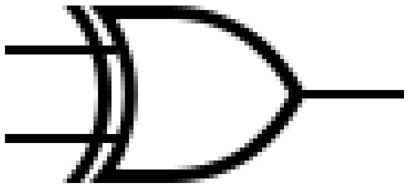
A & B

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Exclusive Or

A ^ B

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0



- Some operations aren’t available as C logical operators
  - Xor ^ - either A or B, but not both

- We can build Xor out of &, |, and ~

  - A^B = (~A & B) | (A & ~B)
    - (exactly one of A and B is true)

  - A^B = (A | B) & ~(A & B)
    - (either is true but not both are true)

  - The two definitions are equivalent
    - Produce the same Truth Table

Practice problem

(A & B) | B

A	B	(A&B)|B
0	0	
0	1	
1	0	
1	1	

Practice problem

(A & B) | B

A	B	(A&B)|B
0	0	0
0	1	1
1	0	0
1	1	1

Practice problem

(A & B) | B

A	B	(A&B)|B
0	0	0
0	1	1
1	0	0
1	1	1

This is equivalent to B
(A has no influence on the solution)

De Morgan's Law

- We can express Boolean operators in terms of the others

- De Morgan's laws: allow swapping & with |

  - A & B = ~(~A | ~B) → ~(A & B) = ~A | ~B
    - (neither A nor B is false)

  - A | B = ~(~A & ~B) → ~(A | B) = ~A & ~B
    - (A and B are not both false)

  - Useful for simplifying logical statements

Generalized Boolean algebra

- Boolean operations can be extended to work on vectors of bits (i.e., bytes)

- Operations are applied one bit at a time: ***bitwise***

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- All of the properties of Boolean algebra still apply
  - Relationships between operations, etc.

- Bitwise operations are usable in C: **&**, **|**, **~**, **^**
  - Can operate on any integer type (long, int, short, char, signed or unsigned)

Warning: bitwise operations are NOT logical operations

- Logical operations in C: **||**, **&&**, **!** (logical Or, And, and Not)
  - Only operate on a single bit
    - View 0 as "False"
    - View *anything nonzero* as "True"
    - Always return 0 or 1
  - Short-circuit evaluation: only checks the first operand if that is sufficient

- Examples
  - !0x41 -> 0x00   !0x00 -> 0x01   !!0x41 -> 0x01
  - 0x59 && 0x35 -> 0x01
  - (p != NULL) && *p (short circuit avoids null pointer access)

Useful for turning many bits into 1 bit

- Don't confuse the two!! It's a common C mistake

Break + Practice

1. 0x37 | 0xAA
2. 0x06 ^ 0xFF

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

A | B

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

A ^ B

A	B	A & B
0	0	0
0	1	1
1	0	1
1	1	0

Break + Practice

1. 0x37 | 0xAA

```
   00110111
|  10101010
-----------
   10111111
```

2. 0x06 ^ 0xFF

```
   00000110
^  11111111
-----------
   11111001
```

A | B

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

A ^ B

A	B	A & B
0	0	0
0	1	1
1	0	1
1	1	0

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Outline

- Integer Operations
  - Addition
  - Negation and Subtraction
  - Multiplication and Division
- **Binary Operations**
  - Boolean Algebra
  - **Shifting**
  - Bit Masks

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~~00010000

Argument **x**	10100010
<< 3	~~101~~00010000

- 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

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

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

- 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)`

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.

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

Shift to divide

- Division by powers of two could be shifts
  - 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);`

Outline

- Integer Operations
  - Addition
  - Negation and Subtraction
  - Multiplication and Division

- **Binary Operations**
  - Boolean Algebra
  - Shifting
  - **Bit Masks**

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

Set 6th bit to one:

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

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

Clear 6th bit to zero:

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

Example: swap nibbles in byte

- Nibble - 4 bits (one hex digit)
  - Input: 0x4F -> Output 0xF4
  - Method:
    - 1. Shift and select upper four bits
    - 2. Shift and select lower four bits
    - 3. Combine the two nibbles

What are the values of the new upper bits?

Unsigned -> Will be zero

```
uint8_t lower = input >> 4;
uint8_t upper = input << 4; // lower bits zeros
uint8_t output = upper | lower; // combines two halves
```

Shifting implicitly zero’d out irrelevant bits.
Otherwise we would have needed an & operation.

Example: selecting bits

- Select bits 2 and 3 from a number

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

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

In C:

```
result = (input & 0x0C) >> 2;
```

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

```
0b00000001
```

Outline

- Integer Operations
  - Addition
  - Negation and Subtraction
  - Multiplication and Division

- Binary Operations
  - Boolean Algebra
  - Shifting
  - Bit Masks

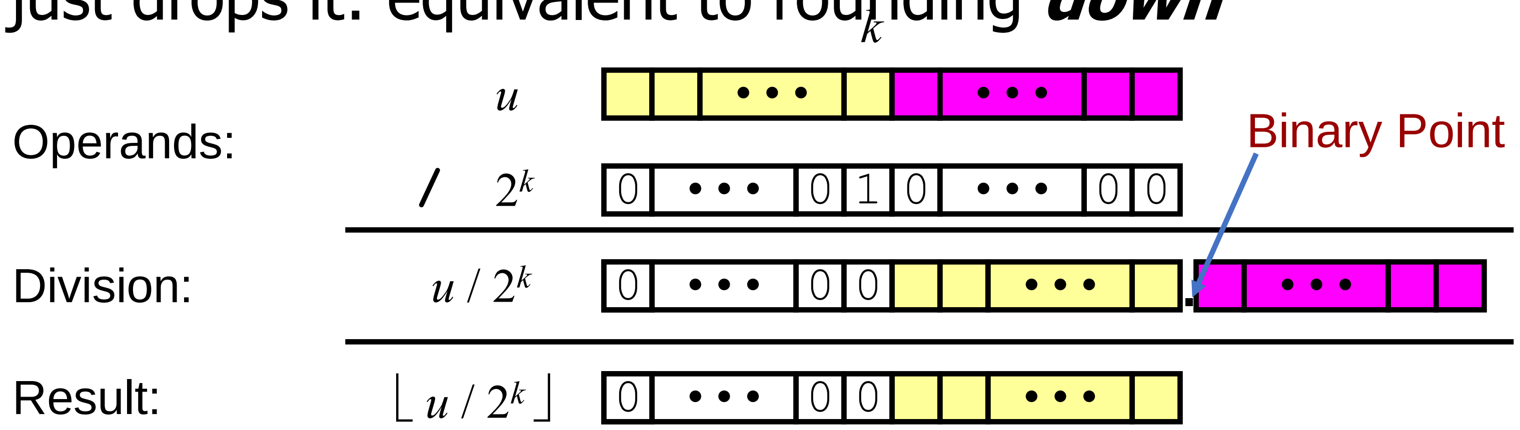
Outline

- Dividing with bit shift
- Bonus material isn't required and won't be on an exam
  - Unless it becomes main lecture material in a different lecture
- Usually the material is just for students who want more depth
  - As is the case here

Unsigned Power-of-2 Divide with Right Shift

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

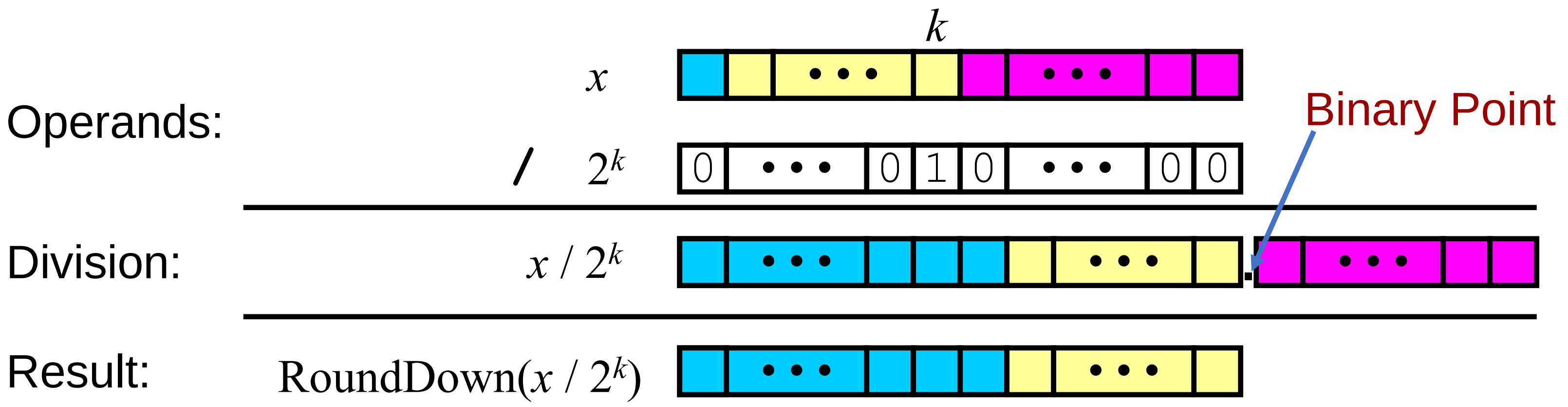
- **Quotient of unsigned by power of 2**
  - **`u >> k`** gives $\lfloor$ **`u / 2^k`** $\rfloor$
  - Uses logical shift
  - Pink part would be remainder / fractional part (right of the point)
    - Shift just drops it: equivalent to rounding ***down***



	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 $\lfloor$ **`x / 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!

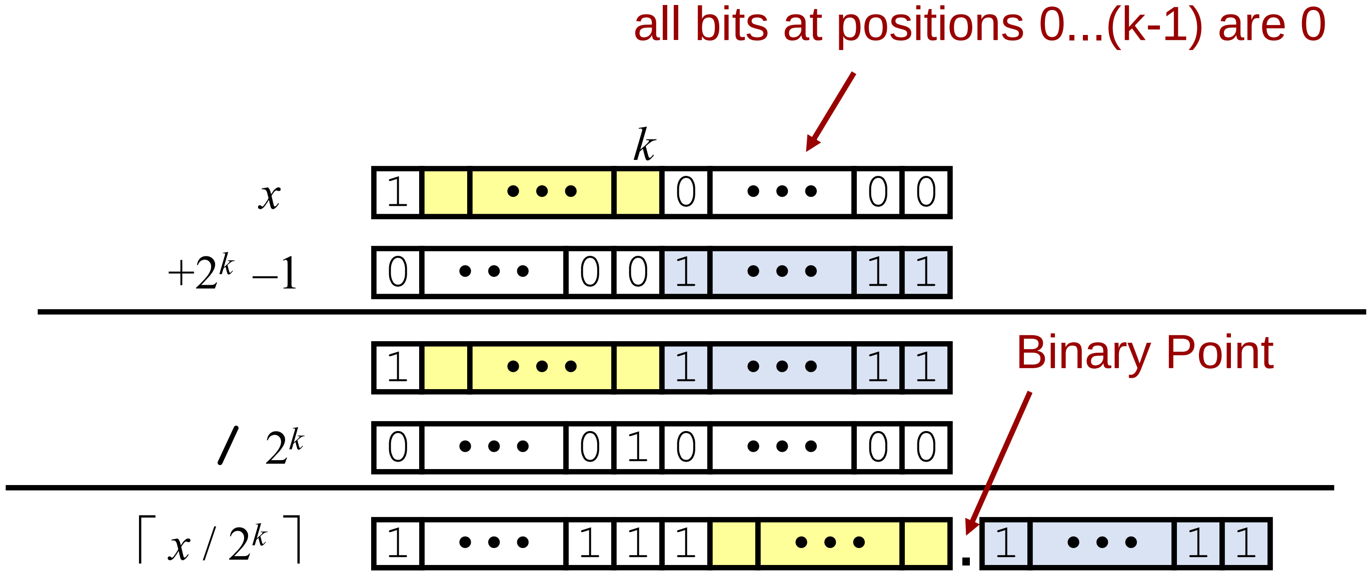


- **Example, 4 bits: -6 / 4 = -1.5 (should round towards 0, to -1)**
  - 1010_2 >> 2 = $\textcolor{red}{\mathit{11}}10_2$ = -2_{10}
  - Rounds the wrong way!

Correct Signed Power-of-2 Divide

- Want $\lceil$ **`x / 2`**$^k\rceil$ (round towards 0)
  - Math identity: $\lceil \mathbf{x / y} \rceil = \lfloor \mathbf{(x + y - 1) / y} \rfloor$
  - 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**

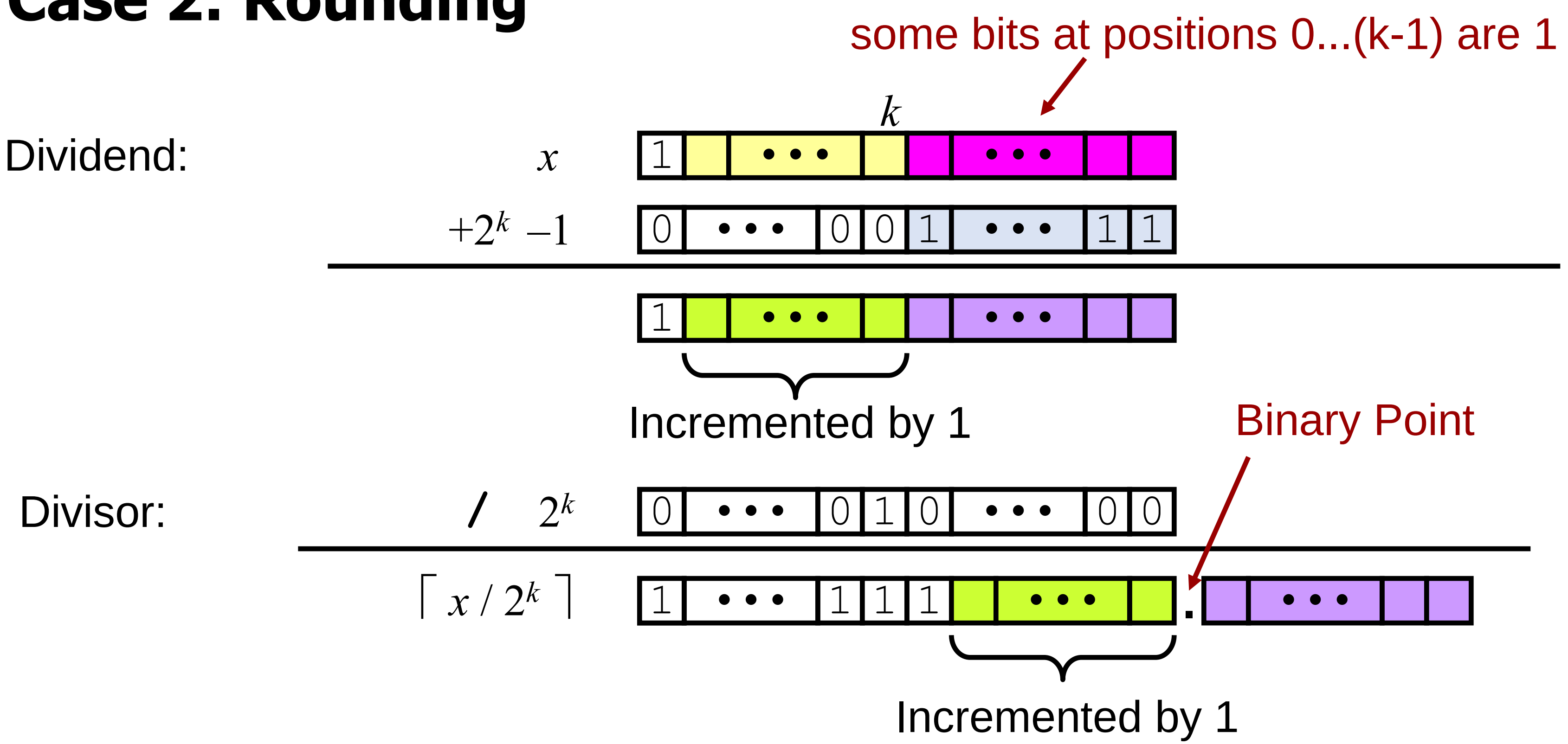


Biasing has no effect; all affected bits are dropped

- **Example, 4 bits: -8 / 2² = -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!)**