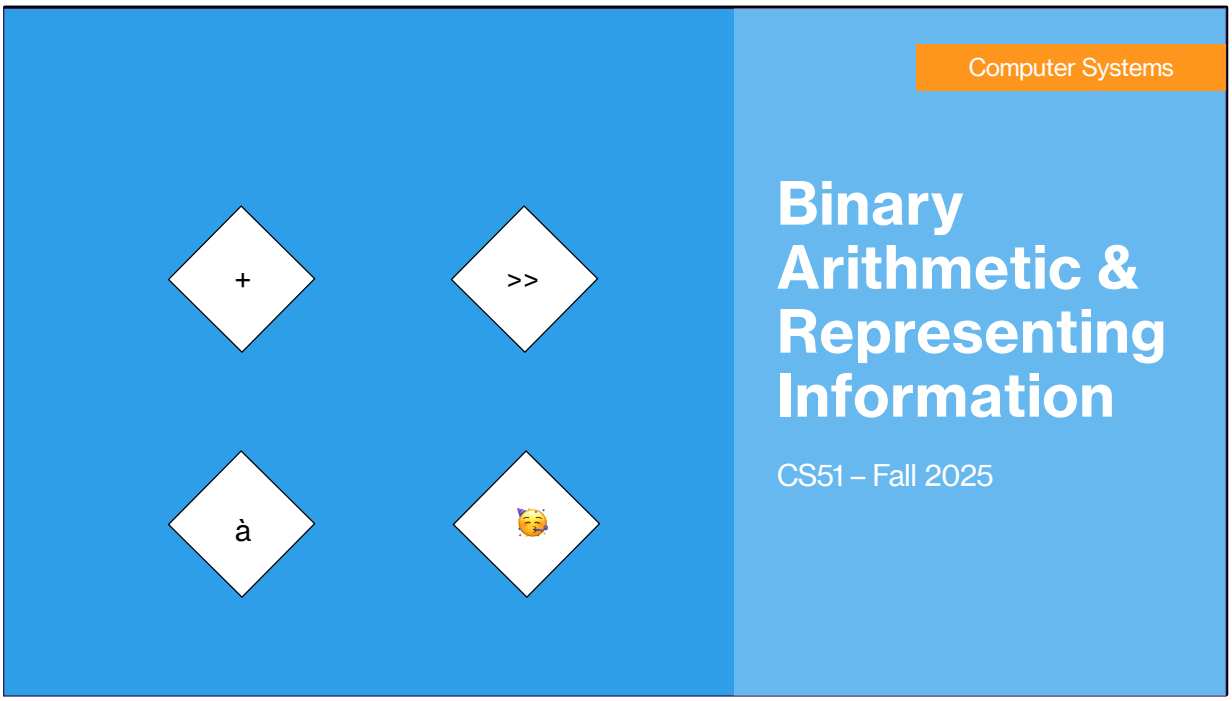


Computer Systems

Binary Arithmetic & Representing Information

CS51 – Fall 2025

Last week, we saw that there are different numeral systems. We examined binary, decimal, and hexadecimal and learned how to convert a number from one representation to another. Today, we will focus on binary numbers alone and learn i) how to perform basic arithmetic with them, and ii) how to use them to represent information that goes past positive integers (e.g., negative numbers, characters, emojis, even colors!)

Addition

2

How do we perform addition in decimal?

Decimal Addition

- Remember how you add decimal integers: start with adding the rightmost digits (the least significant digits). If the sum is greater or equal to 10, then carry a 1 and write down the sum%10. Repeat with the next digits to the left, but this time include the **carry** in the addition.
- For example, adding $4597_{10} + 334_{10}$:

```
 0 1 1 1 ← carry
 4 5 9 7
+  3 3 4
-------
 4 9 3 1
```

3

Let's start with a basic refresher of how to perform addition in the decimal system. We start with the right-most digits (remember, we call these the least-significant digits). If the sum of the two digits is greater or equal to 10, then we can a 1 and write down the sum modulo 10. We repeat with the next digits to the left, but this time include the "carry" in the addition

Binary and hexadecimal addition

- The same idea applies to any base. Just substitute 10 with the appropriate base.
  - If the base is b and the sum of two digits is greater or equal to b, then carry a 1 and write the digit sum%b.
  - In binary, if the sum of two numbers is greater or equal to 2, then carry a 1 and write the sum%2.
  - In hexadecimal, if the sum of two numbers is greater or equal to 16 (i.e. more than F), then carry a 1 and write the sum%16. Same example ($4597_{10} + 334_{10}$), translated in bases 2 and 16.

Binary		**Hexadecimal**
0 0 0 0 0 1 1 1 1 1 1 1 0 0	← carry →	0 0 1 1
0 1 0 0 0 1 1 1 1 1 0 1 0 1		1 1 F 5
+ 0 1 0 1 0 0 1 1 1 0		+ 1 4 E
0 1 0 0 1 1 0 1 0 0 0 0 1 1		1 3 4 3

4

Adding binary numbers and hex numbers follows the same idea. We just substitute the modulo 10 operation with the appropriate base, i.e. 2 for binary and 16 for hex. That means, in binary, if the sum of two numbers is greater or equal to 2, then carry a 1 and write the sum%2. In hexadecimal, if the sum of two numbers is greater or equal to 16 (i.e. more than F), then carry a 1 and write the sum%16. Same example, translated in base 2 and 16.

PRACTICE TIME - Addition

- Compute the following sums
- $0111_2 + 0101_2$
- $9B9_{16} + 38_{16}$

5

OK Let's see whether we can do addition in binary and hex.

ANSWER - Addition

- Compute the following sums
- $0111_2 + 0101_2$
- $9B9_{16} + 38_{16}$

$0111_2 + 0101_2$

```
 0 1 1 1
   0 1 1 1
 + 0 1 0 1
 ---------
   1 1 0 0
```

← carry →

$9B3_{16} + 38_{16}$

```
   0 0 1
     9 B 9
   +   3 8
   -------
     9 F 1
```

6

Dod you get these results?

Overflow

Overflow

- If we operate on a fixed number of digits, addition can **overflow** if the result is too big to fit in the available digits.
- For example, a 4-bit number can represent the values 0, 1, ..., 15. If the addition exceeds 15, the fifth bit would need to be discarded and we would end up receiving an incorrect result.
- We can detect overflow by checking whether we have a carry bit out of the leftmost/most significant column. Here is an example:

$1101_2 + 0101_2$

```
11 01          ← carry
 1 1 0 1
+0 1 0 1
--------
1 0 0 1 0
```

8

A potential issue when we add two numbers is that if we operate on a fixed number of digits, adding them can result in overflow if the result exceeds the available digits. For example, if we work with 4 bits, we can only represent the 16 first numbers: 0, 1, …, 15. If the addition exceeds 15th, the fifth bit (most significant bit) would need to be discarded and we would end up receiving an incorrect result. We can detect overflow by checking whether we have a carry bit for the MSB column s in this example.

Signed numbers

Signed integers

- So far, we have only considered **unsigned** numbers that represent positive quantities.
- What if we want to represent negative quantities, too?
- Two popular schemes to support this:
  - Sign/magnitude
  - Two's complement

So far, we have only considered unsigned numbers that represent positive quantities. Any ideas of what do we do if we have negative quantities? There are two possible approaches to support this. We will start the more simple one, sign or magnitude numbers.

Sign/magnitude numbers

- Intuitive representation that matches the negative sign (–) in front of negative numbers.
- If we have n bits, we store in the first bit (MSB) the sign and the remaining n-1 bits represent the number as usual.
  - Positive is indicated with 0 and negative with 1.
- For example, if we have 8 bits, 47_{10} and -47_{10} can be written as sign/magnitude numbers:
- 47_{10} = 00101111
- -47_{10} = 10101111
- Notice that both numbers have the same **magnitude**, 0101111, it's only the first bit that changes.

11

The first scheme is the more intuitive one and it matches the negative sign in front of negative numbers. If we are working with n bits, the MSB is reserved for the sign and the rest of n-1 are available for representing the number as usual. We will indicate positive with 0 and negative with 1. For example, if we are working with the number 47 and we have 8 available bits, as a positive it would be 00101111 and as negative it flip the MSB to 1, so it would be 10101111, Notice that both numbers have the same magnitude, 0101111, it's only the first bit that changes.

Issues with sign/magnitude numbers

- If we have n bits, we can represent the sign/magnitude numbers $[-2^{n-1}+1\ ,\ 2^{n-1}+1]$.
- Notice that we have two different representations for 0: +0 and -0.
- Ordinary binary addition does not work. For example, adding 47_{10} and -47_{10} results:

00101111

+10101111

011011110 = 222_{10}

Which is nonsensical!

12

If we have n bits available, we can represent the range of $[-2^{n-1}+1\ ,\ 2^{n-1}+1]$. Unfortunately, problems can quickly arise. First, notice that we have two representations for 0. A positive and negative one! Also ordinary binary addition that we just saw does not work. For example if we try to add the binary representations of the decimal numbers 47 and -47, we wouldn't get 0 but the nonsensical 222!

Two's complement numbers

- Assume we had the following binary representation of an *unsigned* integer*:*
  - 1001_2
  - That would be equal to $1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = 9_{10}$
- In two's complement representation, in an n-bit number, the most significant bit (leftmost bit), represents -2^{n-1}. That means that:
  - 1001_2
  - Now becomes $-1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = -7$.
- This representation has only one 0 and leads to easy calculations with common arithmetic operations.

13

So what happens in practice? The second approach is called two's complements. Let's start by assuming the following binary representation of an unsigned number: 1001. That would correspond to the decimal number 9. In two's complement representation, in an n-bit number, the most significant bit (leftmost bit), represents -2^{n-1}. That means that in this representation, 1001 is the number -7! The advantage of two's complement is that it has only one 0 and it supports common arithmetic operations without any issues.

PRACTICE TIME - Two’s complement numbers

- What *unsigned* and *signed* integer does the 1111_2 represent?
- Unsigned:
- Signed (in two's complement):

14

OK let's give it a try with 1111.

ANSWER- Two's complement numbers

- What *unsigned* and *signed* integer does the 1111_2 represent?
- Unsigned: $1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 15_{10}$
- Signed (in two's complement): $-1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = -1_{10}$

15

Is this what you got?

PRACTICE TIME - Two’s complement numbers

- What *unsigned* and *signed* integer does the 1100_2 represent?
- Unsigned:
- Signed (in two's complement):

16

One more!

ANSWER- Two's complement numbers

- What *unsigned* and *signed* integer does the 1100_2 represent?
- Unsigned: $1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = 12_{10}$
- Signed (in two's complement): $-1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 = -4_{10}$

17

Range of two's complement numbers

- Regardless of whether we work with signed or unsigned numbers, with n bits, we can represent 2^n numbers. What changes is the range:
  - Unsigned: $0, 1, 2, 3, \ldots\ 2^n - 1$
  - Signed: $-2^{n-1}, \ldots 0, \ldots\ 2^{n-1} - 1$
- For example, with 4 bits, we can represent in:
  - Unsigned: $0, 1, 2, 3, \ldots 15$
  - Signed: $-8, -7, \ldots 0, \ldots 6, 7$
- Notice that the most negative number ($10 \ldots 00_2$) does not have a counterpart positive number. This is why sometimes it is called *the weird number*!

18

Regardless of whether we work with signed or unsigned numbers, with n bits, we can represent 2^n numbers. What changes is the range. With unsigned, we start at 0 and go all the way to 2^n-1. With signed, we have -2^(n-1), …, 2^(n-1) -1 (and a single 0 in between). For example, with 4 bits we can represent the numbers 0, …, 15 while with signed -8, .., 0, … 7. Notice that the most negative number ($10 \ldots 00_2$) does not have a counterpart positive number. This is why sometimes it is called *the weird number*!

PRACTICE TIME

- Fill in the rest of the table

Binary Representation	Unsigned Decimal	Two's Complement
0000	0	
0001	1	
0010		
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

19

Let's do a quick exercise by filling out the unsigned decimal and two's complement representation of all possible numbers with 4 bits.

ANSWER

- Fill in the rest of the table

Binary Representation	Unsigned Decimal	Two's Complement
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

20

Is this what we have?

ANSWER

- Fill in the rest of the table
- Notice that the MSB indicates whether its two's-complement representation is a positive (0) or negative (1) number!
- 0 is all 0000 (and thus is positive).
- Odd numbers end in 1 and even numbers end in 0.

Binary Representation	Unsigned Decimal	Two's Complement
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

21

Notice that the MSB indicates whether its two's-complement representation is a positive (0) or negative (1) number! 0 is considered positive and odd numbers end in 1 and even numbers end in 0.

Taking the two's complement

- To obtain the two's complement representation of a negative number in binary use the following trick.
  - Step 1: Take the absolute value and convert it in binary. Prefix with 0s to show it's positive.
  - Step 2: Invert all the bits, i.e. change all the 0s to 1s and 1s to 0s.
  - Step 3: Add 1 to the entire inverted number.
- For example, to calculate the binary representation of the two's complement of -47_{10} using 8 bits:
  - Step 1: Binary of absolute value 101111. Prefixing it with 0s to match 8 bits: 00101111.
  - Step 2: Invert all bits: 11010000.
  - Step 3: Add 1 to entire inverted number: 11010001.

22

How do we go from a negative decimal to its two's complement representation?

Step 1: Take the absolute value and convert it in binary. Prefix with 0s to show it's positive.

Step 2: Invert all the bits, i.e. change all the 0s to 1s and 1s to 0s.

Step 3: Add 1 to the entire inverted number.

For example, to calculate the binary representation of the two's complement of -47_{10} using 8 bits:

Step 1: Binary of absolute value 101111. Prefixing it with 0s to match 8 bits: 00101111.

Step 2: Invert all bits: 11010000.

Step 3: Add 1 to entire inverted number: 11010001.

PRACTICE TIME - Taking the two's complement

- Find the two's complement representation of -18_{10} using 8 bits.

23

Your turn to find the two's complement of -18 assuming you have 8 bits.

ANSWER - Taking the two's complement

- Find the two's complement representation of -18_{10} using 8 bits.
- Step 1: Absolute value (18) converted in binary including prefix of 0s: 00010010.
- Step 2: Invert all bits: 11101101
- Step 3: Add 1 to entire inverted number: 11101110

24

Did you get this?

From binary in two's complement to decimal

- We will use a similar trick to reverse the process and convert the two's complement representation of a negative from binary to decimal:
  - Step 1: Invert all of the bits.
  - Step 2: Add 1 to the inverted number.
  - The resulting number is the magnitude of the original number.
- For example, if you are given the two's complement in binary: 1101 (you know it's negative because it starts with a 1).
  - Step 1: Invert bits to 0010
  - Step 2: Add 1 to the inverted number: 0011
  - Convert 0011_2 to decimal -> 3. That means the original number was -3.

25

We can also see the inverse process when converting from binary in two's complement to binary.

Step 1: Invert all of the bits.
Step 2: Add 1 to the inverted number.
The resulting number is the magnitude of the original number.

For example, if you are given the two's complement in binary: 1101 (you know it's negative because it starts with a 1).

Step 1: Invert bits to 0010
Step 2: Add 1 to the inverted number: 0011
Convert 0011_2 to decimal -> 3. That means the original number was -3.

You could of course also have done the trick we saw before -

1x2^3+2x2^2+0x2^2+1x2^0 = -8+4+1=-3.

Adding/subtracting two's complement numbers

- The advantage of using two's complement numbers is that they provide an easy way to add (and subtract) both positive and negative numbers.
- We will add all the bits from right to left as usual but will disregard the last carry bit from the MSB.
- For example, add $-2_{10} + 1_{10}$ using two's complements assuming 4 bits:
  - -2_{10} is 1110_2
  - 1_{10} is 0001_2
  - Adding them results to 1111_2 which is the negative number -1_{10}
- Subtraction is performed by taking the two's complement of the second number, then adding the two. For example, $-2_{10} - 1_{10}$ would be adding $1110_2 + 1111_2 = 1101_2$ (we drop the left carry), which is equal to -3_{10}!

26

The advantage of using two's complement numbers is that they provide an easy way to add (and subtract) both positive and negative numbers. We will add all the bits from right to left as usual but will disregard the last carry bit from the MSB. Let's say we have -2+1. -2 in two's complementary would be transformed to 1110, assuming 4 bits (First write it as 0010, then flip it to 1101, then add 1, 1110). 1 is just 0001 since it's positive. Adding them together will result to 1111 which is -1 (using either process we saw before).
Subtraction is performed by taking the two's complement of the second number, then adding the two. For example, $-2_{10} - 1_{10}$ would be adding $1110_2 + 1111_2 = 1101_2$ (we drop the left carry), which is equal to -3_{10}!

PRACTICE TIME – Subtracting two's complements

- Calculate $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

27

Let's practice by calculating 3-5 in two's complement assuming 4 bits,

ANSWER – Subtracting two's complements

- Calculate $3_{10} - 5_{10}$ using 4-bit two's complement numbers.
- 3_{10} is 0011_2
- Take the two's complement of -$5_{10} = 1011_2$:
  - $5_{10} = 0101_2$
  - Invert 1010_2
  - Add 1: 1011_2
- Add them: $0011_2 + 1011_2 = 1110_2 = -2_{10}$
  - Invert and then 1: $0001 + 1 = 0010 = 2$
  - So $1110_2 = -2_{10}$

28

Did you get this?

Overflow in two's complements

- If we have n bits available, adding two positive or two negative numbers can cause overflow if the result is outside the range of $-2^{n-1}, \ldots 0, \ldots\ 2^{n-1} - 1$
- Adding a positive to a negative number never causes overflow.
- Overflow is detected if two numbers have the same sign and the result has the opposite sign. For example,
- If we calculate $4_{10} + 5_{10}$ using 4-bit two's complement numbers we get $0100_2 + 0101_2 = 1001_2 = -7_{10}$ which indicates overflow.
- We could avoid it if we used at least 5 bits! For example, with 5 bits we would have - $00100_2 + 00101_2 = 01001_2 = 9_{10}$ which would be correct.

29

With n bits available, we can get overflow if we add two positive or two negative (but not one positive one negative) and they go out of the range $-2^{n-1}, \ldots 0, \ldots\ 2^{n-1} - 1$

How we detect overflow in two's complements is that when we have two numbers with the same sign resulting in a number with the opposite sign.

Sign extension in two's complements

- **Sign extension**: when extending a two's complement number in more bits, we need to copy the sign bit.
- For example, if we have 4 bits the numbers -3_{10} and 3_{10} are written in binary as 1101_2 and 0011_2.
- If we want to sign extend them to 8 bits, we would need to copy the sign bit 4 times to 11111101_2 and 00000011_2, respectively.

30

When we want to extend a two's complement number to more bits, we are going to copy the sign bit (0 for positive, 1 for negative). E.g., if we have 4 bits and the binary representations for -3 and 3 (1101 and 0011, respectively), we would pad them with 1s and 0s, respectively.

Why number representation errors matter

- 1. Ariane 5 Rocket Explosion (1996)
  - Flight software converted a 64-bit floating point value into a 16-bit signed integer.
  - The value was too large which caused an overflow which led to system failure.
  - The rocket self-destructed 40 seconds after launch leading to a loss of $370 million.
- 2. Classic Video Games
  - Many classic video games stored scores/lives in 8-bit signed integers (−128, … ,127)
  - If score exceeded 127 it wrapped to negative values.
  - Players suddenly saw “negative lives” or game crashes.

31

Why do representations of numbers matter? Here are two examples when things can go wrong. Ariane 5 rocket launched in 1996 and the flight software converted a 64-bit floating point number into a 16-bit integer. The value was too large, causing an overflow and system failure. The rocket self-destructed less than a minute pass its launch, taking with it more than $370 million. A less catastrophic example would be to look at classic video games, e.g., some used 8 bits to store scores or lives. If a score exceeded the max number, ie. 127, it wrapped to negative values and played suddenly saw negative lives or game crashes.

Shifting

Shifting

- If we work with n bits, **shifting** shifts the binary representation of a number right or left by a specified number of positions k.

33

Sometimes, we will need to shift the binary representation by k bits either to the left or right.

Right shift

- When shifting right, we have two options:
  - **Arithmetic right shift** which shifts a binary number x *right* by k bits and results in the number being the *division* of x by 2^k (rounding toward minus infinity).
    - We denote this as x >> k, where x is the number to shift to the right by k bits.
    - The left k discarded bits are filled using sign extension with the MSB of the original number so that the number maintains n bits.
  - **Logical right shift** which shifts a binary number *right* by k bits but does not offer any mathematical guarantees for negative numbers.
    - We denote this as x >>> k, where x is the number to shift to the right by k bits.
    - The right k discarded bits are filled with 0s so that the number maintains n bits.

34

When shifting to the right, we have two options: arithmetic shift right which shifts a binary number x right by k bits and results in the number being the division of x by 2^n. We would denote this as x>>k The left k discarded bits are filled using sign extension with the MSB of the original number so that the number maintains n bits.
Or, logical right shift which shifts them right by k and fills them out using 0s, thus not offering any mathematical guarantees for negative numbers.

Arithmetic right shift - Example

- Let's assume we work with 4 bits and we want to shift the binary representation of the number 6_{10} by 2 bits to the right, that is 6>>2.
- We start by converting the decimal number to its binary representation:
  - $6_{10} = 0110_2$
- Remember, since we have arithmetic right shift, we shift two positions to the right and copy 2 times the MSB (0) to the left. This results in:
  - $00011\cancel{0}\cancel{0}$, that is 0001_2.
- If we convert back to decimal, we get $0001_2 = 1_{10}$.
- We said that arithmetic right shift by k divides the original number by 2^k rounding down to minus infinity.
  - Indeed $\frac{6}{4} = 1.5$ which is rounded down to 1.

35

Let's see an example of both. Let's assume we have 4 bits and we want to shift 6 to the right by 2 using arithmetic right shift. First, let's write the binary representation of 6, i.e. 0110. Since we are performing arithmetic right shift, we would shift two to the right, discarding the last two digits and padding the left with the MSB, i.e., 0, thus resulting to 0001 which is equal to 1 in decimal. We said that arithmetic right shift by k divides the original number by 2^k rounding down to minus infinity.

$$\text{Indeed } \frac{6}{4} = 1.5 \text{ which is rounded down to 1.}$$

PRACTICE TIME - Arithmetic right shift

- Let's assume we work with 4 bits and we want to shift the binary representation of the number -6_{10} by 2 bits to the right, that is -6>>2.
- Using two's complement representation show what the effect of the shift will be on the binary representation of -6_{10}.

36

Your turn to apply arithmetic right shift by 2 bits to the binary -6 using two's complement and assuming 4 bits.

ANSWER - Arithmetic right shift

- We start by converting the decimal number to its binary representation in two's complement:
  - $-6_{10} = 1010_2$
- Remember, since we have arithmetic right shift, we shift two positions to the right and copy 2 times the MSB (1) to the left. This results in:
  - $1110\cancel{10}$, that is 1110_2.
- If we convert back to decimal, we get $1110_2 = -2_{10}$.
- We said that arithmetic right shift by k divides the original number by 2^k rounding down to minus infinity.
  - Indeed $-6/4 = -1.5$ which is rounded down to -2.

37

We will start by converting the decimal number to its binary representation in two's complement, i.e. 1010. We shift two to the right, i.e. 10 and pad it with two MSB signs, i.e. 1110. That would be the number -2. Let's check that the division by 4 holds. Yes, $-6/4 = -1.5$ which is rounded down to -2.

Logical right shift - Example

- Let's assume we work with 4 bits and we want to shift the binary representation of the number -6_{10} by 2 bits to the right using logical shift, that is -6>>>2.
- We start by converting the decimal number to its binary representation using two's complement:
  - $-6_{10} = 1010_2$
- Remember, since we have logical right shift, we shift two positions to the right and add two 0s to the left. This results in:
  - $0010\cancel{10}$, that is 0010_2.
- If we convert back to decimal, we get $0010_2 = 2_{10}$.
- Note that since we worked with a negative integer, we cannot offer any guarantees about the effect of a logical shift to the right!

38

OK, now what about logical right shift. Remember we shift k bits to the right but then pad with 0s! So for -6 that would be 0010 or 2 in decimal. So no guarantees about its effects on negative numbers.

Left shift

- Shifting a binary number *left* by k bits results in the number being *multiplied* by 2^k. There is no distinction between logical and arithmetic left shift.
  - We denote this as x << k, where x is the number to shift to the left by k bits.
  - The right k discarded bits are filled with 0s so that the number maintains n bits.
- Note that the resulting number has to be within the range $[-2^{n-1}, 2^{n-1} - 1]$ otherwise we will have an overflow and can have unpredictable results.

39

We can also perform left shift which results in a number being multiple by 2^k. Note that there is no logical vs arithmetic left shift, just left shift. We denote it with <<. What happens to the right k discarded bits is that they get replaced with k 0s. Note that the resulting number has to be within the range $[-2^{n-1}, 2^{n-1} - 1]$ otherwise we will have an overflow and can have unpredictable results.

Left shift - Example

- Let's assume we work with 6 bits and we want to shift the binary representation of the number 3_{10} by 2 bits to the left, that is 3<<2.
- We start by converting the decimal number to its binary representation:
  - $3_{10} = 000011_2$
- Remember, since we have left shift, we shift two positions to the left and add 2 0s to the right. This results in:
  - ~~00~~001100, that is 001100_2.
- If we convert back to decimal, we get $001100_2 = 12_{10}$.
- We said that left shift by k multiplies the original number by 2^k.
  - Indeed $3 \times 2^2 = 12$.

40

Let's see an example of left shift together. If we have 4. bits and the decimal 3 which we want to shift to the left by 2 bits (aka 3<<2), we start by converting it to its binary representation, i.e. 0011. Then we discard the first two bits and pad 11 with two 0s, i.e. we get 1100 which is the number 12 in decimal. So we just got that shifting 3<<2=12=3x2^2!

PRACTICE TIME – Left shift

- What happens when we shift -10<<3 assuming we have 8 bits?

41

Your turn, assume you have 8 bits, and you try to shift -10 by 3 bits to the left.

ANSWER – Left shift

- What happens when we shift -10<<3 assuming we have 8 bits?
- We start by converting the decimal number to its binary representation using two's complement and sign extension:
  - $-10_{10} = 11110110_2$
- Remember, since we have left shift, we shift three positions to the left and add three 0s to the right. This results in:
  - ~~111~~10110000, that is 10110000_2.
- If we convert back to decimal, we get $10110000_2 = -80_{10}$.
  - Invert to 01001111 and then adding 1 results to $01010000 = 80$ so overall $10110000_2 = -80_{10}$
- We said that left shift by k multiplies the original number by 2^k.
  - Indeed $-10 \times 2^3 = -80$.

42

Start by converting to the two's complement representation in binary I,e, 10110. Pad it with 1s to reach 8 bits. 11110110.
Shift three to the left and add three 0s to the right. 10110000 which is equal to -80 and it checks out in terms of multiplying -10 by 2^3!

Shifting in Python

- Python supports left and arithmetic right shifting. It denotes them as << and >>, respectively.
- Other programming languages, like Java, support both arithmetic and logical right shift and use the >> and >>> operators to distinguish them.

43

Python supports left shifting and right arithmetic shifting using the << and >> operators. Other languages, like Java, also support logical right shifting making a distinction by using >>>.

Bitwise operations

Bitwise operations

- If we are given two numbers, we can apply the operators NOT, AND, OR, and XOR **bitwise**, that is bit by bit.
- Bitwise NOT takes one operand and flips each of its bits, that is it changes the 0s to 1s and 1s to 0s.
- Bitwise AND takes two operands of the same number of bits and compares each bit in the corresponding position. If they are both 1, it returns 1, otherwise it returns 0.
- Bitwise OR takes two operands of the same number of bits and compares each bit in the corresponding position. If they are both 0, it returns 0, otherwise it returns 1.
- Bitwise XOR takes two operands of the same number of bits and compares each bit in the corresponding position. If exactly one of them is 1, it returns 1, otherwise it returns 0.

45

What other things can we do with binary numbers? We can apply the logical operators NOT, AND, OR, XOR bitwise, that is bit by bit.
Bitwise NOT takes one operand and flips each of its bits, that is it changes the 0s to 1s and 1s to 0s.
Bitwise AND takes two operands of the same number of bits and compares each bit in the corresponding position. If they are both 1, it returns 1, otherwise it returns 0.
Bitwise OR takes two operands of the same number of bits and compares each bit in the corresponding position. If they are both 0, it returns 0, otherwise it returns 1.
Bitwise XOR takes two operands of the same number of bits and compares each bit in the corresponding position. If exactly one of them is 1, it returns 1, otherwise it returns 0.

Bitwise operations - Example

- In Python, we denote bitwise NOT, AND, OR, and XOR as ~, &, |, and ∧ respectively.
  - Note that ^ is not for conjunction here!
  - In languages like Java, bitwise and logical operations are distinguished by using a single or double operator, e.g., & for bitwise AND, but && for logical AND
- Let's assume we have $x = 7_{10} = 0111_2$ and $y = -4_{10} = 1100_2$ here is the effect of each bitwise operation.
  - $\sim x = 1000_2 = -8_{10}$ and $\sim y = 0011_2 = 3_{10}$
  - $x\&y = 0100_2 = 4_{10}$
  - $x|y = 1111_2 = -1_{10}$
  - $x \wedge y = 1011_2 = -5_{10}$

46

Let's see bitwise operations in Python. we denote bitwise NOT, AND, OR, and XOR as ~, &, |, and ∧ respectively. Note that ^ is not for conjunction here! In languages like Java, bitwise and logical operations are distinguished by using a single or double operator, e.g., & for bitwise AND, but && for logical AND

Let's assume we have $x = 7_{10} = 0111_2$ and $y = -4_{10} = 1100_2$ here is the effect of each bitwise operation.

$$\sim x = 1000_2 = 8_{10} \text{ and } \sim y = 0011_2 = 3_{10}$$
$$x\&y = 0100_2 = 4_{10}$$
$$x|y = 1111_2 = -1_{10}$$
$$x \wedge y = 1011_2 = -5_{10}$$

PRACTICE TIME – Bitwise operations

- Let's assume we have $x = 5_{10}$ and $y = -9_{10}$
  - 1) Start by converting them to binary, assuming five bits
  - 2) What will the effect be for:
  - $\sim x$
  - $\sim y$
  - $x \& y$
  - $x | y$
  - $x \wedge y$

47

Let's check our understanding with these two numbers.

ANSWER – Bitwise operations

- Let's assume we have $x = 5_{10}$ and $y = -9_{10}$
  - 1) Start by converting them to binary: $x = 00101_2$ and $y = 10111_2$
  - 2) What will the effect be for:
  - $\sim x = 11010_2 = -6_{10}$
  - $\sim y = 01000_2 = 8_{10}$
  - $x \& y = 00101_2 = 5_{10}$
  - $x | y = 10111_2 = -9_{10}$
  - $x \wedge y = 10010_2 = -14_{10}$

48

Did you get these results?

Masking

- We can use bitwise operations to **mask** a certain number of bits in a number. By masking, we can reset, set, or invert certain bits in a number.
- Bitwise AND will reset a subset of the bits to 0
- Bitwise OR will set a subset of the bits to 1
- Bitwise XOR will invert a subset of the bits

49

Bitwise operations often go hand in hand with masking. When we apply a mask on an input, we isolate certain bits and reset them to 0 (logical AND), set them to 1 (logical OR) or invert them (XOR). We will see each one with examples and an opportunity to practice.

Bitwise AND – Resetting bits to 0

- Let's say we work with 8 bits and we have the number $47_{10} = 00101111_2$ and we want to reset the three least significant bits to 0.
- We would mask these bits by applying a bitwise AND on the mask $11111000_2 = -8_{10}$.

Bit	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
Data	0	0	1	0	1	1	1	1
Mask	1	1	1	1	1	0	0	0
Result	0	0	1	0	1	**0**	**0**	**0**

- This would result to $00101000_2 = 40_{10}$

50

Let's say we work with 8 bits and we have the number 47 in decimal which translates to 00101111 in binary. If we want to reset the three least significant bits (i.e. the last three bits) to 0, then we would mask these bits by applying the bitwise AND on the mask 11111000 which is equal to -8n decimal. Notice how the 5 most significant bits stay unaffected and the last three are reset, resulting to the number 00101000 in binary which is equal to 40.

PRACTICE TIME - Bitwise AND

- What would the result be of applying the mask 42_{10} on the input 85_{10} using bitwise AND? You can assume you have 8 bits.

51

Your turn. Start by converting 42 abd 85 to binary before you apply bitwise AND. What was its effects on the bits? What will the result be in decimal?

ANSWER - Bitwise AND

- What would the result be of applying the mask 42_{10} on the input 85_{10} using bitwise AND? You can assume you have 8 bits.
- First, we convert from decimal to binary, working with 8 bits:
  - $42_{10} = 00101010_2$
  - $85_{10} = 01010101_2$
- We then apply the bitwise AND operation which will reset the the 7th (no real effect), 6th, 4th, 2nd, and 0th bit of the input, resulting to 0_{10}:

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	0	1	0	1	0	1
Mask	0	0	1	0	1	0	1	0
Result	**0**	**0**	0	**0**	0	**0**	0	**0**

52

Bitwise OR – Setting bits to 1

- Let's say we work with 8 bits and we have the number $47_{10} = 00101111$ and we want to set the first two bits to 1.
- We would mask these bits by applying a bitwise OR on the mask $-64_{10} = 11000000_2$.

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0	0	1	0	1	1	1	1
Mask	1	1	0	0	0	0	0	0
Result	**1**	**1**	1	0	1	1	1	1

- This would result in -17_{10}.

53

When we apply a mask on an input using a bitwise OR, we set the bits where 1 appears in the mask, to 1. For example, if we use the mask -64 which has two 1s in the 7th and 6th bit, we alter these to 1.

PRACTICE TIME - Bitwise OR

- What would the result be of applying the mask 17_{10} on the input -27_{10} using bitwise OR? You can assume you have 8 bits.

54

Your turn! As before, convert 17 and -27 to binary (using signed extension for two's complement) and assume you have 8 bits.

ANSWER - Bitwise OR

- What would the result be of applying the mask 17_{10} on the input -27_{10} using bitwise OR? You can assume you have 8 bits.
- First, we convert from decimal to binary, working with 8 bits:
  - $17_{10} = 00010001_2$
  - $-27_{10} = 11100101_2$
- We then apply the bitwise OR operation which will set the the 4th and 0th bit of the input to 1, resulting to -11_{10}:

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	1	1	1	0	0	1	0	1
Mask	0	0	0	1	0	0	0	1
Result	1	1	1	**1**	0	1	0	**1**

55

This will result in the 4th and 0th bit being set to 1.

Bitwise XOR – Inverting bits

- Let's say we work with 8 bits and we have the number $117_{10} = 01110101$ and we want to invert the first and last two digits and leave the four in between unchanged.
- We would mask these bits by applying a bitwise XOR on the mask $-61_{10} = 11000011_2$.

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	1	1	0	1	0	1
Mask	1	1	0	0	0	0	1	1
Result	**1**	**0**	1	1	0	1	**1**	**0**

- This would result in -74_{10}.

56

bitwise XOR inverts the digits the digits that have a 1 in the mask.

Digital logic mysteries

- Bitwise operations can be quite useful in digital logic and can make code both more concise and faster.
- This week's assignment will give you more practice with bitwise operations.
- In the meantime, let’s work on some mystery functions.

57

Bitwise operations can be quite useful in digital logic and can make code both more concise and faster.
This week's assignment will give you more practice with bitwise operations.
In the meantime, let’s work on some mystery functions. You want to try different numbers to come up with theories of what they do.

PRACTICE TIME - Mystery 1

- What does the following Python function do?

```
def mystery1(a, b):
      return (a ^ b) == 0
```

58

Think, what type will this function return? If you assume a and b are integers, what are good inputs to test? When would the result of applying a bitwise xor result to the decimal number 0?

ANSWER - Mystery 1

- What does the following Python function do?

```
def mystery1(a, b):
      return (a ^ b) == 0
```

- The function takes two numbers (a,b), applies the bitwise XOR (a^b), turns it into a decimal and compares the result to 0. If it is equal to 0, it returns True, otherwise it returns False.
- If both numbers have n bits, we have $a_{n-1}a_{n-2} \dots a_0$
- Tests if a and b are equal.
- $5_{10} \wedge 5_{10}$ is $0101_2 \wedge 0101_2 = 0000_2 = 0_{10}$
- $5_{10} \wedge 6_{10}$ is $0101_2 \wedge 0110_2 = 0100_2 = 4_{10}$

59

The function returns a bool. It seems like it's converting the result of the XOR bitwise operation on a and b to a decimal number and comparing it against 0. 0 in decimal is 0000…0 in binary. For the result of a bitwise XOR operation to always be 0, we would need every single bit we compare between the two numbers to be equal.

PRACTICE TIME - Mystery 2

- What does the following Python function do? You can assume we are working with 4 bits

```
def mystery2(a):
      return (a & (1<<3)) != 0
```

60

Again, what is the return type? Try a few different inputs for a.

ANSWER - Mystery 2

- What does the following Python function do? You can assume we are working with 32 bits

```
def mystery2(a):
      return (a & (1<<3)) != 0
```

- Checks whether a is negative by isolating the MSB.

61

Don't forget that if you have an n-bit number, the MSB would be at the n-1 bit (starting 0 to the right and advancing to the left)

PRACTICE TIME - Mystery 3

- What does the following Python function do?

```
def mystery3(a,b):
      return (a & (1<<b)) != 0
```

62

This should feel very familiar. The only thing we changed is 3 to b.

ANSWER - Mystery 3

- What does the following Python function do?

```
def mystery3(a,b):
      return (a & (1<<b)) != 0
```

- Tests if the b-th bit in a is 1.

63

So now we test the b-th bit, not necessarily the MSB. Ideally, we should have checked that b is within the n bits that a has but that's a bit complicated in languages like Python.

PRACTICE TIME - Mystery 4

- What does the following Python function do?

```
def mystery4(a):
      return a | 1
```

64

Last head scratcher. What does this super concise function do? Let's start by realizing that it does not return a bool but a decimal number

ANSWER - Mystery 4

- What does the following Python function do?

```
def mystery4(a):
      return a | 1
```

- If a is even, it returns a+1, if a is odd, it leaves it unchanged.

65

Cool, huh?

Beyond numbers

66

Representing information

- So far, we have seen that computers store information in bits and we have interpreted these bits as integers.
- But what about non-integer numbers, such as floating-point numbers, or any other type of information, such as letters, emojis, colors, sound etc?
- There are different international standards that determine how binary is **encoded** into other representations.
- For example, the most used standard to encode real numbers is the IEEE 754 standard.
- But under the hood, it's all bits!

67

So far, we have seen that computers store information in bits and we have interpreted these bits as integers.
But what about non-integer numbers, such as floating-point numbers, or any other type of information, such as letters, emojis, colors, sound etc?
There are different international standards that determine how binary is **encoded** into other representations.
For example, the most used standard to encode real numbers is the IEEE 754 standard.
But under the hood, it's all bits!

Encoding text - ASCII

- **ASCII** (American Standard Code for Information Interchange) which was established in the '60s and used 7 bits to represent 128 characters: These included the capital and lowercase letters of the English alphabet, digits 0-9, punctuation and special symbols like @.
  - For example, 01000011 01010011 00110101 00110001 00100001 represents the text "CS51!"
- ASCII was adopted by different computer manufacturers ensuring **interoperability**.
- Soon an 8th bit was used to expand the available set to 255 characters.
- In the US, this was used for accented characters and mathematical symbols and many other languages used these extra characters to encode their own alphabets.
- But that meant that text written in different languages encoded to gibberish when read in a different context.

68

ASCII (American Standard Code for Information Interchange) which was established in the '60s and used 7 bits to represent 128 characters: These included the capital and lowercase letters of the English alphabet, digits 0-9, punctuation and special symbols like @.

For example, 01000011 01010011 00110101 00110001 00100001 represents the text "CS51!"

ASCII was adopted by different computer manufacturers ensuring **interoperability**.

Soon an 8th bit was used to expand the available set to 255 characters.

In the US, this was used for accented characters and mathematical symbols and many other languages used these extra characters to encode their own alphabets.

But that meant that text written in different languages encoded to gibberish when read in a different context.

Encoding text - Unicode

- In 1990s, **Unicode** expanded encoding to thousands of characters to account for all different schemes individual languages used and even includes emojis!
- The most common implementations of Unicode are **UTF-8** ,which was designed for backward compatibility with ASCII, and **UTF-16**.



Smileys & Emotion

face-smiling

№	Code	Browser	Sample	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600				–	–	–	grinning face
2	U+1F603							grinning face with big eyes
3	U+1F604					–	–	grinning face with smiling eyes
4	U+1F601							beaming face with smiling eyes
5	U+1F606				–		–	grinning squinting face
6	U+1F605				–		–	grinning face with sweat
7	U+1F923			–	–	–	–	rolling on the floor laughing
8	U+1F602					–		face with tears of joy
9	U+1F642				–	–	–	slightly smiling face
10	U+1F643			–	–	–	–	upside-down face

In 1990s, **Unicode** expanded encoding to thousands of characters to account for all different schemes individual languages used and even includes emojis!
The most common implementations of Unicode are **UTF-8** ,which was designed for backward compatibility with ASCII, and **UTF-16**.

Encoding images

- Images consist of pixels, the smallest image unit.
- Each pixel stores a **color** using three **color channels**: Red, Green, Blue (RGB).
- Each color channel can be encoded using binary numbers
  - Typical: 8 bits per channel, that is 24 bits per pixel. With 8 bits, we can represent 0,..., 255
  - For example, 00000000 represents no red and 11111111 represents full red. Same for green, blue.
- Putting it together, we have triples of numbers to represent pixels. E.g.,
  - Pure red = 11111111 00000000 00000000→ (255,0,0) – only red
  - Pure white = 11111111 11111111 11111111→ (255,255,255) – all three colors
  - Pure black =00000000 00000000 00000000→ (0, 0, 0) – absence of all three colors
- An entire image would be represented as a matrix of pixels (width × height), each pixel's RGB values encoded in binary

70

Images consist of pixels, the smallest image unit.
Each pixel stores a **color** using three **color channels**: **Red, Green, Blue (RGB)**.
Each color channel can be encoded using binary numbers

Typical: 8 bits per channel, that is 24 bits per pixel. With 8 bits, we can represent 0,..., 255
For example, 00000000 represents no red and 11111111 represents full red. Same for green, blue.

Putting it together, we have triples of numbers to represent pixels. E.g.,

Pure red = 11111111 00000000 00000000→ (255,0,0) – only red
Pure white = 11111111 11111111 11111111→ (255,255,255) – all three colors
Pure black =00000000 00000000 00000000→ (0, 0, 0) – absence of all

three colors

An entire image would be represented as a matrix of pixels (width × height), each pixel's RGB values encoded in binary

PRACTICE TIME - Encoding images

- If we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?

71

Let's think for a moment, if we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?

ANSWER - Encoding images

- If we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?
- Each channel supports $2^8 = 256$ possible values.
- Thus, the total number of supported colors is $256 \times 256 \times 256 = 16{,}777{,}216$
- That means that a 24-bit RGB system, can represent over 16 million colors.

72

Each channel supports $2^8 = 256$ possible values.
Thus, the total number of supported colors is
$256 \times 256 \times 256 = 16{,}777{,}216$
That means that a 24-bit RGB system can represent over 16 million colors.

RGBA format

- Sometimes, the RGB format is supplemented with one more channel called the **alpha channel**.
- The alpha channel indicates how opaque a pixel is.
- By convention RGBA colors are stored in hex. For example, for alpha, 00 would be fully transparent and FF fully opaque. For colors, 00 would be lack of color and FF would be pure color.
- For example, the RGBA color #FF00FF80 is a semi-transparent purple:
  - FF_{16} stands for a full red in the red channel
  - 00_{16} stands for no green in the green channel
  - FF_{16} stands for a full blue in the blue channel
  - and $80_{16} = 128_{10}$ represents 50% opacity.

73

Beyond rbg, we often encode one more piece of information about pixels, their alpha value which stands for how opaque a pixel is. We typically work with hex numbers, that is 00 would be fully transparent for alpha or lack of color for rgb, and ff would be fully opaque for alpha and full on color for rgb. You can now see why #FF00FF80 is a semi transparent purple.

Practice Problems – Problem 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10}$
  - How many bits do you need to not have overflow?
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2$
  - Convert these three numbers to decimal to double check your work.
- Add the hex numbers 0xA7+0x5C
  - Convert these three numbers to decimal to double check your work.

Practice Problems – Answer 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10} = 101101_2 + 011011_2 = 1001000$
  - How many bits do you need to not have overflow? 7
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2 = 000011_2$. No overflow
  - Convert these three numbers to decimal to double check your work. -18+21=-3
- Add the hex numbers $A7_{16} + 5C_{16}$
  - 103_{16}
  - Convert these three numbers to decimal to double check your work. $167_{10} + 92_{10} = 259_{10}$

75

Practice Problems – Problem 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2$

Practice Problems – Answer 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2 = 11100100_2 \gg 2 = 11111001_2 = -7_{10}$

Practice Problems – Problem 3

- Compute the following expressions in Python:
- $10110101 2_2 \& 11110000_2$
- $01011011_2 | 00101101_2$
- $11001010_2 \wedge 10101100_2$

78

Practice Problems – Answer 3

- Compute the following expressions in Python:
- $101101012_2 \& 11110000_2 = 10110000_2$
- $01011011_2 | 00101101_2 = 01111111_2$
- $11001010_2 \wedge 10101100_2 = 01100110_2$

79