Mathematics and Problem Solving

Lecture 3

Numerical Systems

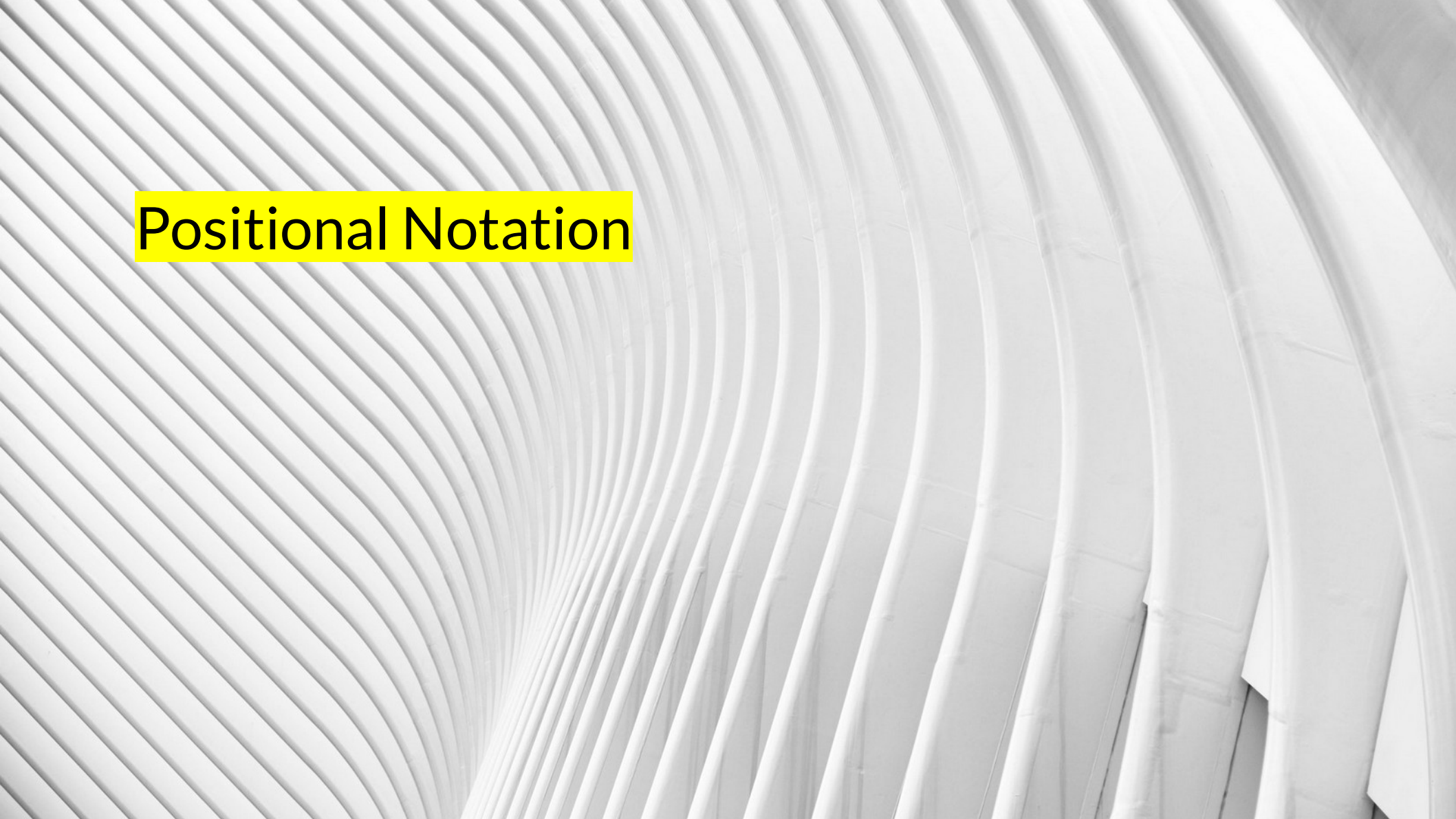
“He is the master who, after reaching the furthest shores and plumbing the inmost depths of the sea of ultimate knowledge of mathematics, kinematics and spherics, handed over the three sciences to the learned world.”

Bhaskara I, of Aryabhata

Overview

- Positional Notation
- Conversion Between Bases
- Binary Arithmetic
- Negative Numbers in Binary

Positional Notation



Positional Notation

- Positional notation is a way of writing numbers in some numerical systems
  - Decimal (base 10)
  - Binary (base 2)
  - Octal (base 8)
  - Hexadecimal (base 16)
- A digit’s value is dependant on its position
- Numbers are represented as a sequence of digits

Positional Notation

- Rightmost digit is smallest
  - Units = 1s
- Digits increase in significance to the left
  - Tens
  - Hundreds
  - Thousands
- 4 thousands = 4 x 1000 = 4 x 10^3

4	3	2	1
10^3	10^2	10^1	10^0
1000	100	10	1

Why use diferent bases?

- Decimal (base 10) is the most common human number system
  - A few exceptions:
    - Babylonian
    - Maya
    - Aztec
    - Kaktovik Inupiaq
- Data in a computer is stored and manipulated in binary (base 2)
- If encoding numbers as text, only using 10 symbols is inefficient. Higher bases can be used

Decimal

- Uses ten symbols
  - 0 1 2 3 4 5 6 7 8 9

7	0	6	1
10^3	10^2	10^1	10^0
1000	100	10	1

7061
$= 7x10^3 + 0x10^2 + 6x10^1 + 1x10^0$
= 7061

Binary

- Binary is a numerical system with 2 digits
  - 0 1

0	1	0	0	1	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

01001101_2
$= 1*64 + 1*8 + 1*4 + 1*1$
$= 77$

Binary Numbers

- $(00000000)_2 = 0$
- $(00000001)_2 = 1$
- $(00000010)_2 = 2$
- $(00000011)_2 = 3$
- $(00000100)_2 = 4$
- $(00000101)_2 = 5$
- ….
- $(11111011)_2 = 251$
- $(11111100)_2 = 252$
- $(11111101)_2 = 253$
- $(11111110)_2 = 254$
- $(11111111)_2 = 255$

You can have binary numbers with >8 digits, but 1 byte stores only 8 binary digits

Hexadecimal

Decimal	Hexadecimal	Decimal	Hexadecimal
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

- Binary uses only 2 different symbols, so it's very long to write out
- Hexadecimal uses the numbers 0-9 and then the letters A-F
- It is also called **base 16**

Hexadecimal

- We often represent data using hexadecimal to save space
  - 1 byte = 2 x 4 bits
  - 4 bits can store $2^4 = 16$ different symbols
  - 1 byte can be represented by a 2-digit hex number

4	E	0	2
16^3	16^2	16^1	16^0
4096	256	16	1

$4E00_{16}$
$= 4*4096 + 14*256 + 2*1$
$= 19970$

Example Hexadecimal Numbers

- $(0000)_{16} = 0$
- …
- $(0009)_{16} = 9$
- $(000A)_{16} = 10$
- …
- $(000F)_{16} = 15$
- $(0010)_{16} = 16$
- $(001F)_{16} = 31$
- $(0020)_{16} = 32$
- …
- $(00FF)_{16} = 255$
- $(0100)_{16} = 256$
- …
- $(FFFF)_{16} = 65535$

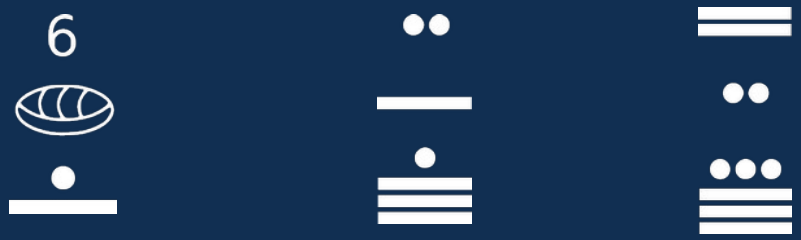
ASCII

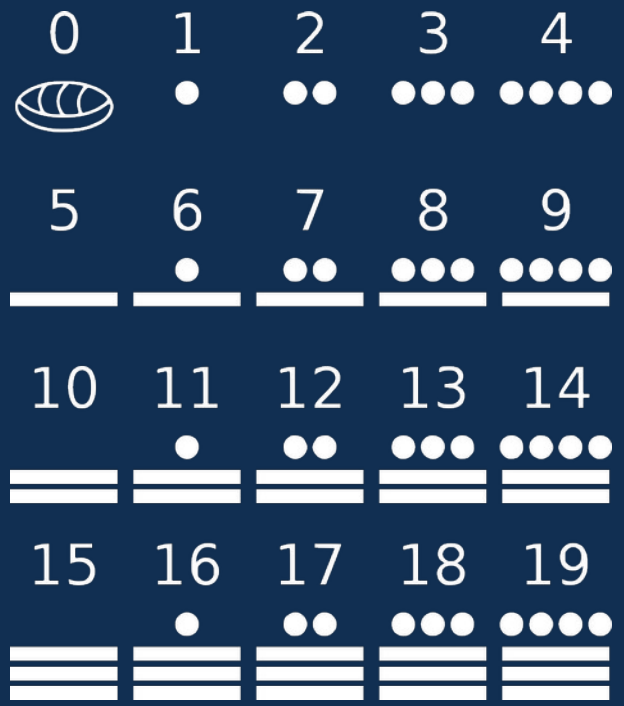
- What if we represented a byte with one character?
- 1 byte = 8 bits = 2^8 = 256 different symbols
  - Base 256
- ASCII
  - 0-31 Control characters
  - 32-126 Printable characters
  - 127-255 Extension to original 7-bit ASCII standard

Mayan Numbers

Exercise 1:

Translate the following numbers.
The one to the left is done for you (=6)





Base *n*

- For any base *b*, the weight (positional value) of the *n*th digit is

$$b^{(n-1)}$$

- An *n* digit number can store unique values equal to

$$b^{n}$$

- The highest value stored by an n digit number is

$$b^{n} - 1$$

digit	8	7	6	5	4	3	2	1
weight	b^7	b^6	b^5	b^4	b^3	b^2	b^1	b^0
unique values	b^8	b^7	b^6	b^5	b^4	b^3	b^2	b^1
highest value	b^8-1	b^7-1	b^6-1	b^5-1	b^4-1	b^3-1	b^2-1	b^1-1

Exercise 2:

What is the ‘weight’ of the highest digit in the following numbers?

1. 3 digit decimal number
2. 2 digit binary number
3. 3 digit octal number
4. 2 digit hexadecimal number

- Hint:
  - For any base *b*, the weight of the *n*th digit is

$$b^{(n-1)}$$

Exercise 3:

How many unique values can the following represent?

1. 2 binary digits
2. 3 binary digits
3. 2 octal digits
4. 3 hexadecimal digits

- Hint:
  - An *n* digit number can store unique values equal to

$$b^n$$

Introducing Logs

- Log is the inverse of exponentiation
  - If

    $b^a = k$

  - then

    $a = \log_b k$

- If the base is 10 and you have a 3 digit number, how many values can it take?
  - $10^3 = 1000$ (0-999)
- If the base is 10 and you want to represent 1000 values, how many digits do you need?
  - $\log_{10} 1000 = 3$

Logs

- If an *n* digit number in base *b* can store b^n unique values, the number of base *b* digits required to store *k* unique values is
  - $n = \log_b k$
  - Round up non-integer values
- E.g. to store 18 unqiue values in base 2
  - $\log_2 18 = 4.1699\ldots$
  - You'd need 5 binary digits

Exercise 4:

How many digits does it take to store the following unqiue values in the given bases?

1. 9 → base 3
2. 123 → base 7
3. 2422 → base 11

- Hint:
  - If $b^a = k$ then $\log_b k = a$
  - An *n* digit number can store unique values equal to

$$b^n$$

Exercise 5:

The weight of a digit is given below, what is its position?

1. 64 → base 2
2. 256 → base 16
3. 2401 → base 7

- Hint:
  - If $b^a = k$ then $\log_b k = a$
  - For any base *b*, the weight of the *n*th digit is

$$b^{(n-1)}$$

Exercise 6:

How many digits are required to store these values in the given bases?

1. 14 → base 8
2. 18 → base 7
3. 242 → base 17

- Hint:
  - If $b^a = k$ then $\log_b k = a$
  - The highest value stored by an n digit number is

$$b^n - 1$$

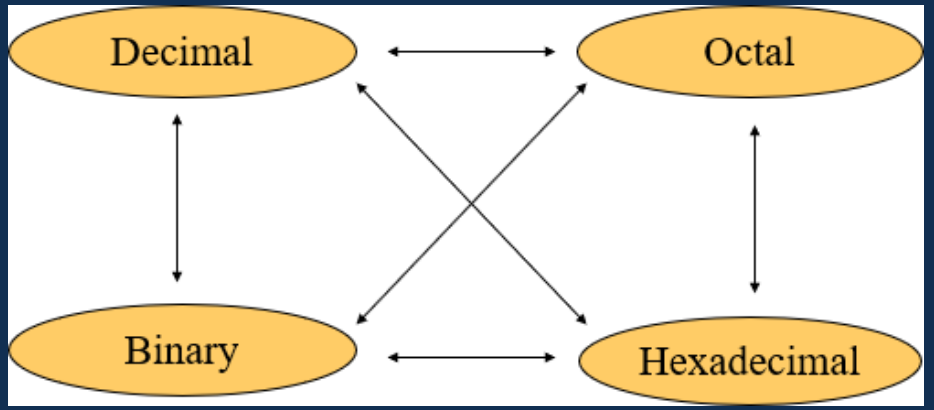
Summary

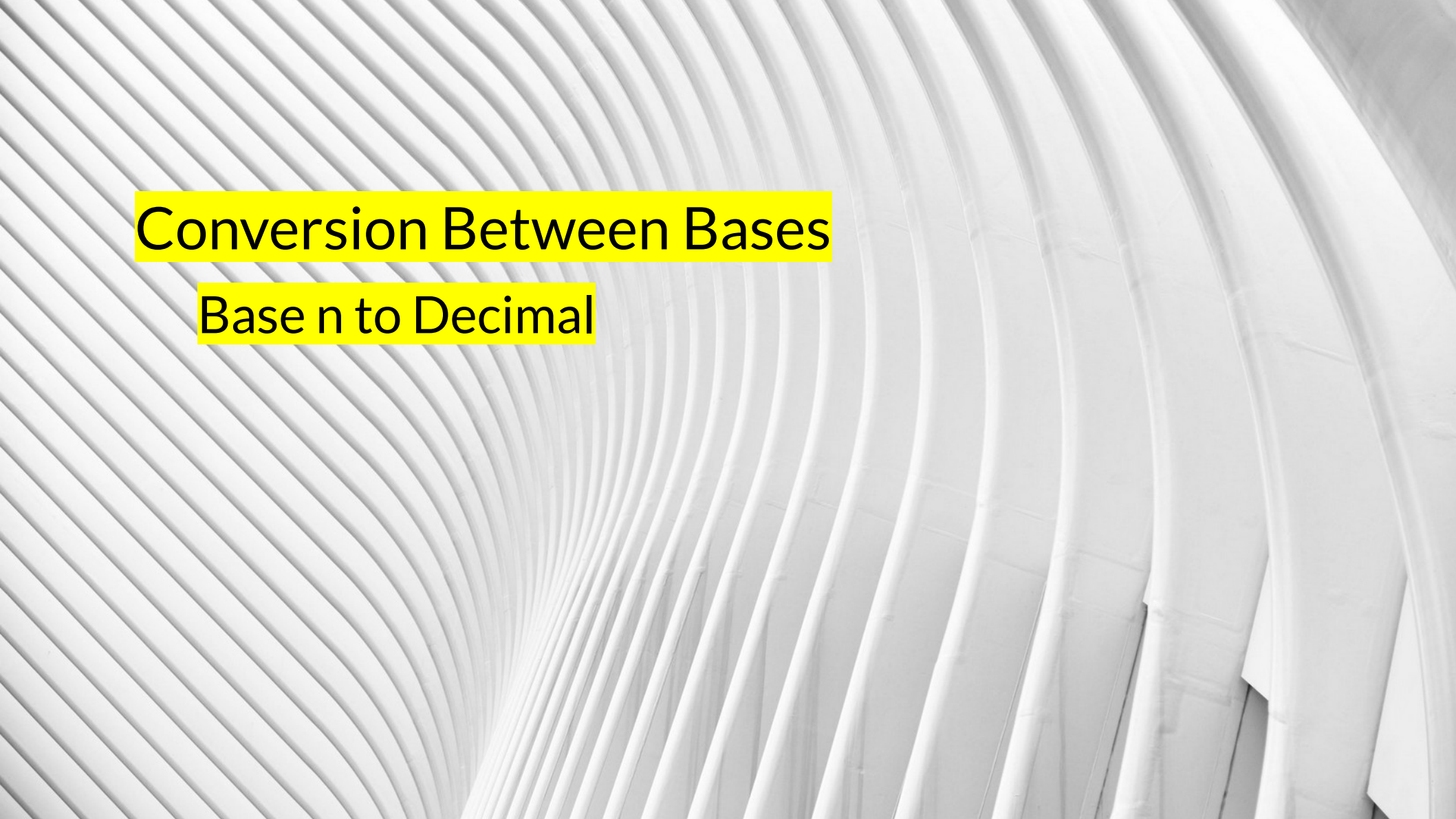
- Data is stored in binary
  - 1 byte = 8 bits → 2^8 = 256 possible values
- Binary is a number system (base 2)
  - A way of representing numbers just like decimal (base 10)
  - Hexadecimal (base 16) is commonly used as well
  - While bytes only store numbers 0-255, number systems can represent any number

Conversion Between Bases

Conversion between bases

- To use different bases, we need to be able to convert between them
- Easiest conversions are:
  - Those involving decimal
  - Between bases that are powers of two





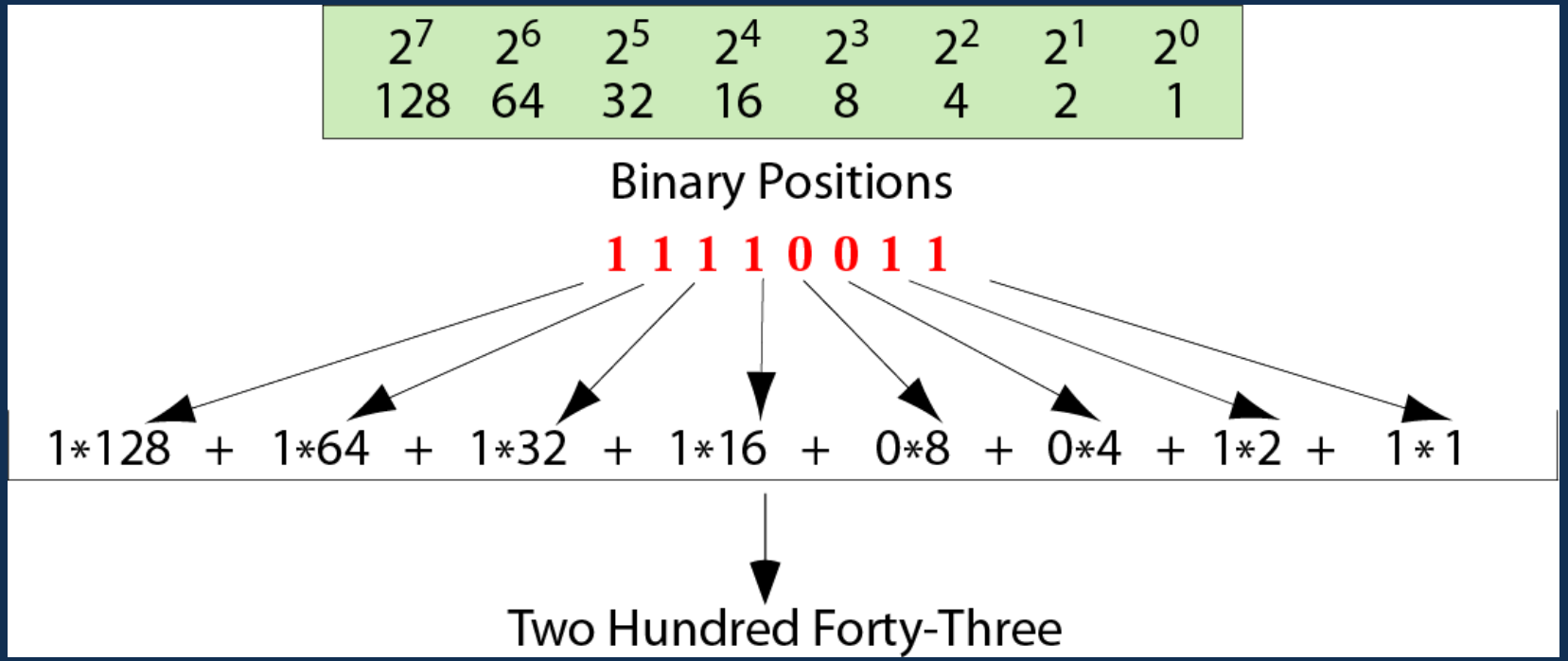
Conversion Between Bases

Base n to Decimal

Binary to Decimal Conversion

- Technique
  - Multiply each bit by 2^n, where n is the position of the bit starting from 0 on the right
  - Sum the results

Binary to Decimal Conversion



Octal to Decimal conversion

- Technique
  - Multiply each bit by 8^n, where n is the position of the bit starting from 0 on the right
  - Sum the results

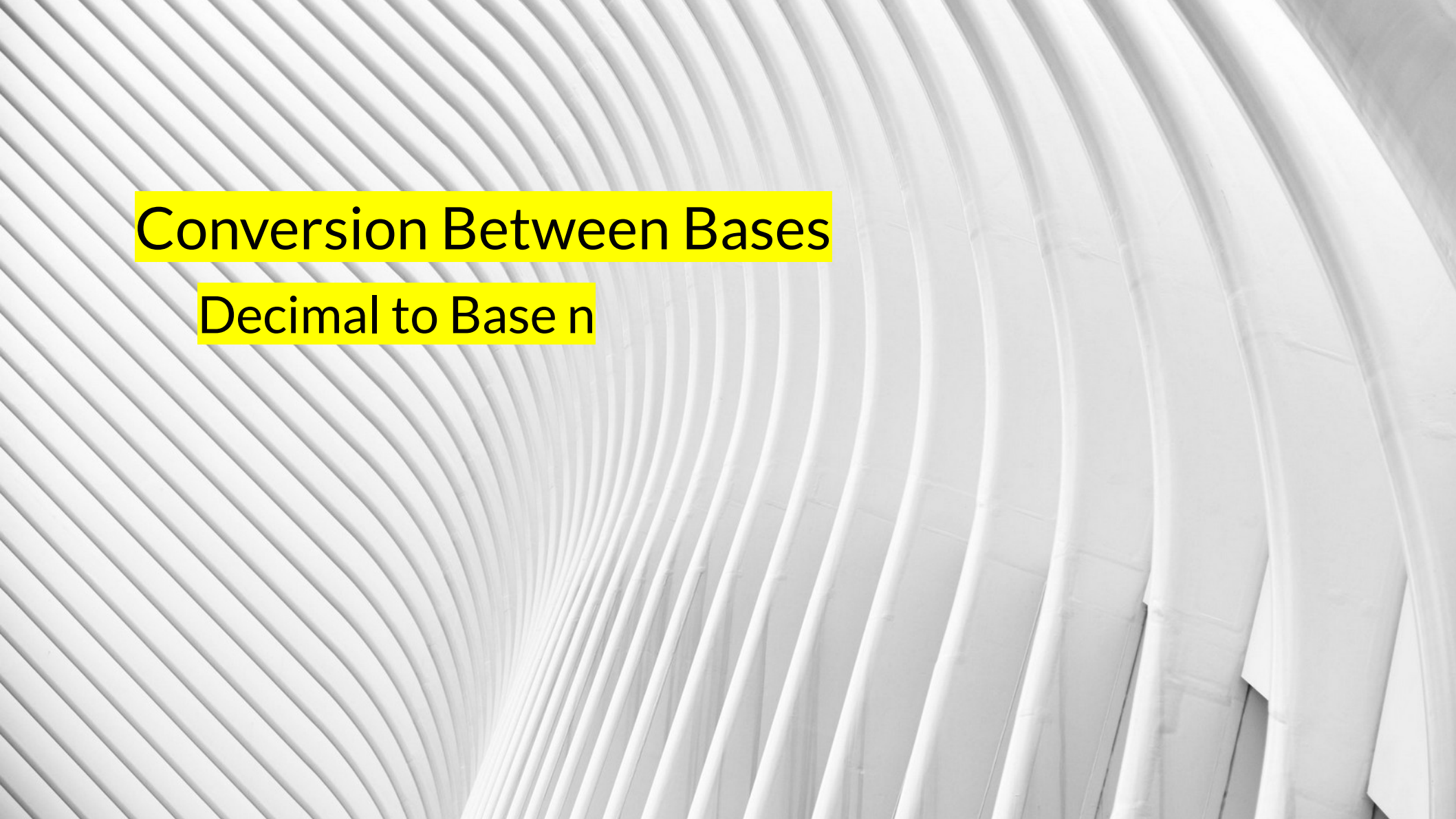
Base n to Decimal

Exercise 7:

Convert the following numbers from their bases into decimal

1. $(1010)_2$
2. $(351)_8$
3. $(4E)_{16}$

- Hint:
  - Multiply each bit by b^n,
    - b is the base
    - n is the position of the bit, starting from 0 on the right
  - Add the results



Conversion Between Bases

Decimal to Base n

Decimal to Binary

- To convert from decimal to binary, we repeatedly divide by 2.
  - Each time, the remainder becomes the next least significant binary digit.
- We divide by 2 because we converting into base 2 (binary).
- In the example, we convert 140 from decimal into binary, getting the answer 10001100.

140 / 2 = 70 r 0

70 / 2 = 35 r 0

35 / 2 = 17 r 1

17 / 2 = 8 r 1

8 / 2 = 4 r 0

4 / 2 = 2 r 0

2 / 2 = 1 r 0

1 / 2 = 0 r 1

10001100

Decimal to Octal

- To convert from decimal to octal, we repeatedly divide by 8.
  - Each time, the remainder becomes the next least significant octal digit.
- We divide by 8 because we converting into base 8 (octal).

140 / 8 = 17 r 4

17 / 8 = 2 r 1

2 / 8 = 0 r 2

214

Decimal to Hexadecimal

- To convert from decimal to hexadecimal, we repeatedly divide by 16.
  - Each time, the remainder becomes the next least significant binary digit.
- We divide by 16 because we converting into base 16 (hexadecimal).

140 / 16 = 8 r 12 (r = C)

8 / 16 = 0 r 8

8C

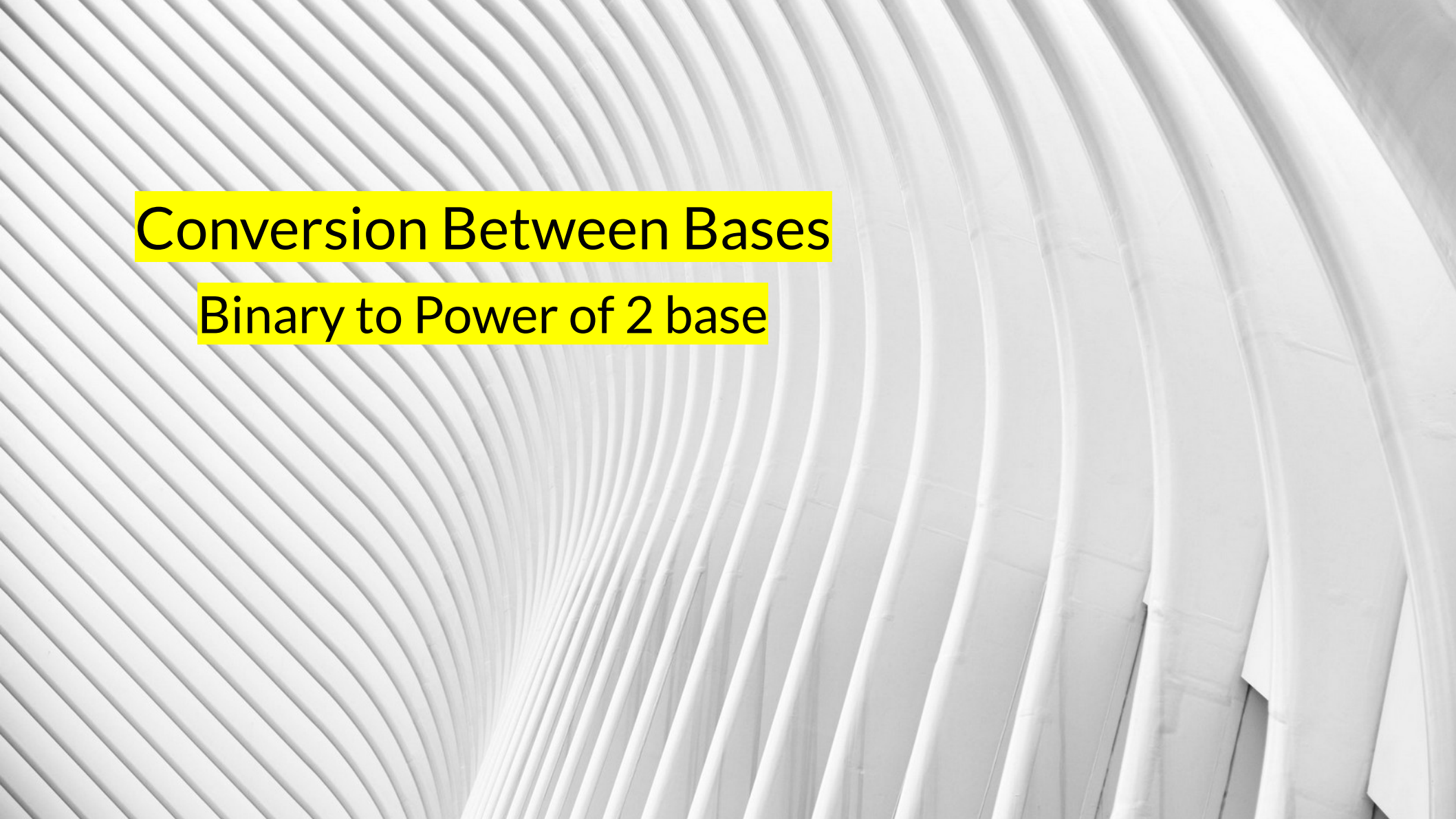
Decimal to base n

Exercise 8:

Convert the following decimal numbers into the given base

1. 14 → base 2
2. 19 → base 6
3. 43 → base 12

- Hint:
  - To convert from decimal to base n, we repeatedly divide by n
  - Each time, add the remainder to the left of the result


Conversion Between Bases
Binary to Power of 2 base

Binary to Octal

- Because each octal digit can be 8 different values, we need 3 binary bits to store one octal digit
  - $2^3 = 8$ (and $\log_2 8 = 3$).
- So to convert binary to octal, we
  - group our bits into triplets (threes), starting on the right
  - convert each triplet to its value in hexadecimal.

- Binary: 10 001 100
- Decimal: 2 1 4
- Octal: 214

Binary to Hexadecimal

- Because each hexadecimal digit can be 16 different values, we need 4 binary bits to store it
  - $2^4 = 16$ (and $\log_2 16 = 4$).
- So to convert binary to hexadecimal
  - we group our bits into quartets (fours)
  - convert each quartet to its value in hexadecimal.

- Binary: 1000 1100
- Decimal: 8 12
- Hexadecimal: 8 C

Convertion to other power of 2 bases

- To convert binary to another power of 2 base, e.g. 8 or 32, we can group the bits differently, e.g. into 3s or 5s.
- We can reverse the steps to convert from a power of 2 base back to binary

Binary and Power of 2 Bases

Exercise 9:

Convert the following numbers between the given bases

1. $(0111)_2 \rightarrow$ base 8
2. $(1001)_2 \rightarrow$ base 16
3. $(722)_8 \rightarrow$ base 2
4. $(EF0)_{16} \rightarrow$ base 2

- Hint:
  - Work out how many bits, *n*, store a single digit of base *b*
  - $\log_2 b = n$
  - $2^n = b$

Conversion Between Bases

Other conversions

Other conversions

Exercise 10:

Convert the following

1. $(E2)_{16}$ → base 8
2. $(72)_{8}$ → base 16

- Hint:
  - Use binary or decimal as an intermediary

Binary Addition

Adding 1 digit binary numbers

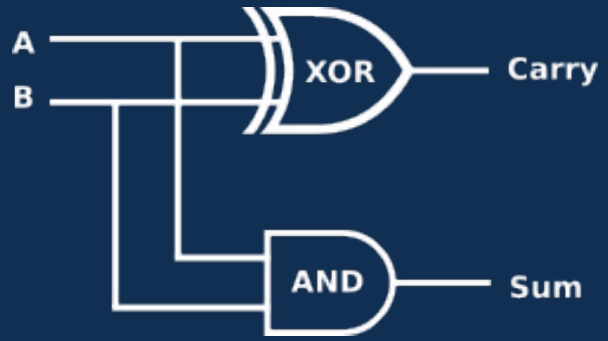
- Adding 1 digit binary numbers together is straight forward
  - As easy as 1+1 = 10
- Digital circuits work with 0 and 1
  - How do they store the output of 1 + 1?

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	10

Half Adder

- Store output of 1-bit addition as two bits
  - Sum bit
  - Carry bit
- Allows us to do addition for one position at a time
  - 1 + 1 = 0 (carry 1)
- This is the same thing that a half adder does
  - A half adder is a circuit used to add two bits of data together

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



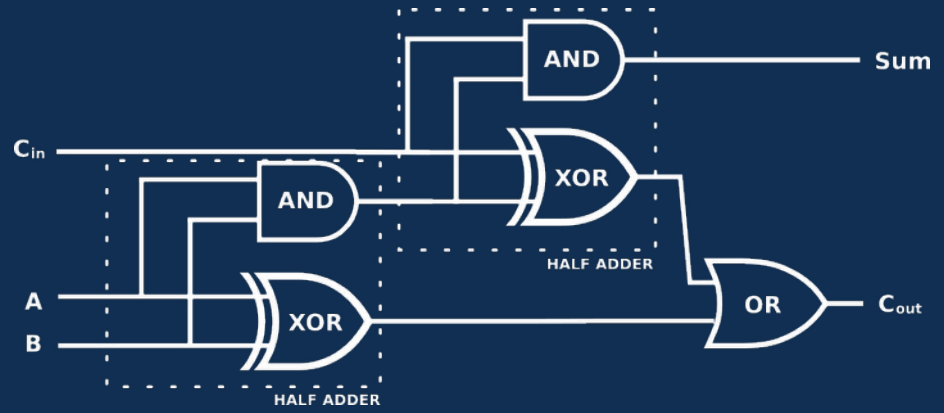
Full Adder

- So really we need to be add three bits together
  - *a*, *b* and a carry bit (c_{in})
- A full adder is a circuit used to add two bits of data together
  - Takes in two bits and a carry bit
  - Returns the sum and a carry bit

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

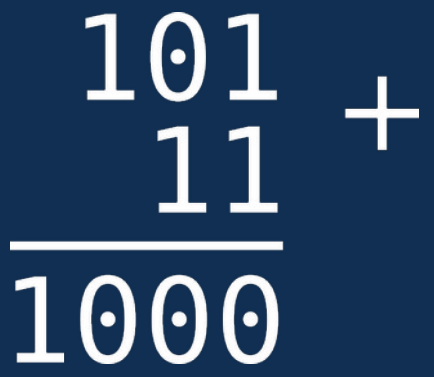
Full Adder

- To add two 8 bit numbers we need
  - 1 half adder (first bit)
  - 7 full adders (remaining bits)
- Or, if doing it manually
  - Add lowest pair of bits
  - Determine sum and carry
  - From then on, add each pair of bits with previous carry
  - Final carry is overflow



Manual Binary Addition

- Write the two numbers above each other.
- Working from right to left, add the columns of digits, carrying numbers to the next column as required.



Exercise 11:

Add the following binary numbers

1. 0010 + 101
2. 101 + 111
3. 010101 + 01101
4. 1111 + 1

- Hint:
  - Write the two numbers above each other.
  - Working from right to left, add the columns of digits, carrying numbers to the next column as required.


Negative Numbers

Representing Negative Numbers

- 8 bits can represent 256 positive numbers
  - 0 – 255
- How do we encode negative numbers?
  - There are no “negative bits” so we cannot store negative values
  - We *can* treat some range of values *as if* they are negative values
  - Three common representations in microprocessors
    - Sign and Magnitude
    - One’s Compliment
    - Two’s Complement ← most modern microprocessors

Representing Negative Numbers

- We map some range of positive numbers onto the negative numbers
  - i.e. we need a transformation that
    - takes a positive number e.g. 1, and
    - returns the number that represents its **additive inverse**, i.e. -1
  - For example 1 might map to 255 (which represents -1)

Representing Negative Numbers

- When we map some range of positive numbers onto the negative numbers
- We want it to
  - Represent as many numbers as possible with the bits available (be efficient)
  - Be able to use the same algorithm/hardware for addition for positive and negative numbers

Sign and Magnitude

- Use the largest bit to represent the sign (+/-), store number with remaining bits
  - if working with *n* bits, the additive inverse of a number *k* is
    - $(k + 2^{n-1}) \bmod 2^n$
- Equivalent to mapping second half of range to negative numbers
  - 0-127 → 0 - 127
  - 128-256 → -0 - -127

0010 = 2

0001 = 1

0000 = 0

1000 = -0

1001 = -1

1010 = -2

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Red values map to

-0, -1, -2, -3, -4, -5, -6, -7

Sign and Magnitude

- With 8 bits
  - 1 bit sign, 7 bit number
  - -127 → 127 - only 255 numbers!
- Inefficient
  - Represents both +0 and -0
- Requires different addition and subtraction algorithm than for unsigned binary

Exercise 12:

What is the value of the following 4-bit signed numbers, stored using sign-and-magnitude?

1. 1001
2. 0011
3. 1111
4. 1000

- Hint:
  - Leftmost bit stores sign
  - 0 = positive
  - 1 = negative

One’s Complement

- Here’s another idea, instead of inverting a sign bit, invert *every* bit
- One’s complement of
  - 0 is 1
  - 1 is 0
- To get the 1s complement of a binary number, invert every bit
  - 0000 → 1111

One's Complement

- Stores negative numbers as the inverse of the corresponding positive number
  - First digit still gives sign
- Still two 0s, only 255 unique numbers
- But notice that values are stored symmetrically

- 0010 = 2
- 0001 = 1
- 0000 = 0
- 1111 = -0
- 1110 = -1
- 1101 = -2

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Red values map to

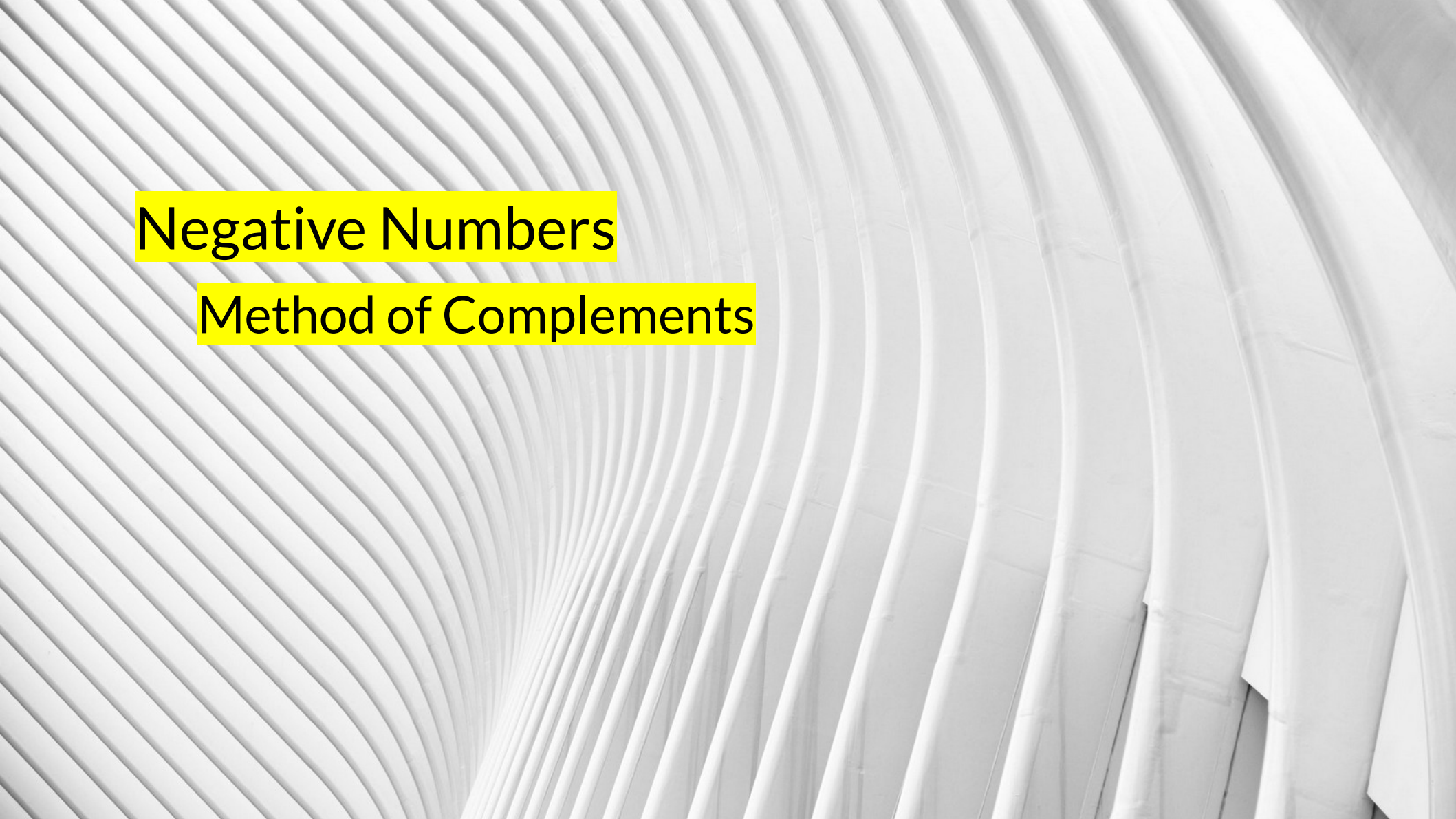
-7, -6, -5, -4, -3, -2, -1, -0

Exercise 13:

What is the value of the following 4-bit signed numbers stored using one's complement?

1. 1101
2. 0010
3. 1111
4. 0101

- Hint:
  - One's Complement stores negative numbers as the corresponding positive number with all bits inverted
  - First digit still gives sign


Negative Numbers
Method of Complements

Method of Complements

- A Method of Complements is a way to encode a symmetric range of numbers
  - Make it easy to perform arithmetic
- Examples of Complements
  - Ones Complement
  - Tens Complement
  - Twos Complement
- A number and its **additive inverse** make a compliment
  - e.g. Ten's Complement is the number you add to a digit to make 10

Ten's Complement

- Imagine we used ten's complement to encode a symmetric range of positive and negative numbers
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - 0, 1, 2, 3, 4, 5, -4, -3, -2, -1
- Addition works, observe:
  - 3 + 6 = 9
  - 3 + -4 = -1

- Ten's compliment of
  - 0 is 0 (*overflow)
  - 1 is 9
  - 2 is 8
  - 3 is 7
  - 4 is 6
  - 5 is 5

Ten's Complement

- The ten's complement of *y* with *n* decimal digits is
  - $10^n - y$
- Again observe, with 2 decimal digits
  - $72 + 33 \equiv 105 \equiv 5 \pmod{10^2}$
  - $72 \equiv -(10^2 - 72) \equiv -28 \pmod{10^2}$
  - $-28 + 33 = 5$

- For example,
  - with 1 decimal digits, the tens complement of 8 is
    - $10 - 8 = 2$
  - with 2 decimal digits, the tens complement of 64 is
    - $100 - 64 = 36$
  - with 3 decimal digits, the tens compliment of 268 is
    - $1000 - 268 = 732$

Exercise 14:

What is the ten's complement of the following (using number of digits shown?

1. 2
2. 57
3. 9999

Which of the above represent negative numbers?

- Hint:
  - Tens complement of an *n* digit number *y* is

$$10^n - y$$

Radix Complement

- In general, the **radix compliment*** of an *n* digit number *y*, in radix *b*, is

$$b^n - y$$

- So to calculate the radix compliment of the base 8 number 274 using 4 digits (i.e. the Eight's Compliment)

$8^4 - 274$

$= 4096 - 274$

$= 3822$

*radix means the same thing as base

Twos Compliment

- Binary has a radix of 2, so it's the Two's Compliment
- For *n* bits, the Two's Compliment of a number *y* is

$$2^n - y$$

or in other words the number you add to get 2^n

- Using 1 bit (2^1)
  - 0 is 0 (*overflow)
  - 1 is 1
- Using 2 bits (2^2)
  - 0 is 0 (*overflow)
  - 1 is 3
  - 2 is 2
  - 3 is 1

Twos Complement Example

- We're working with 8 binary bits (radix 2). We can store numbers 0-255
- Our formula for radix compliment was

$$b^n - y$$

- To store -25, we find

$$2^8 - 25$$
$$= 256 - 25$$
$$= 231$$

- So we can encode the additive inverse of 25 as 231

Twos Complement in Binary

- With 8 bits, the Twos Compliment of 25 is 231
- With the same numbers in binary, the Twos Compliment of
  - $(00011001)_{10}$ is
  - $(11100111)_{10}$
- Note how similar it looks to the One's Compliment
  - Flip every bit (= One's Compliment)
  - Add 1

Diminished Radix Complement

- One's compliment is a **diminished radix complement**, which is the same as the radix compliment minus 1

$$(b^n - 1) - y$$

- Observe that $b^n - 1$ is the largest digit repeated n times, e.g.
  - $10^3 - 1 = (999)_{10}$
  - $2^8 - 1 = (11111111)_2$
- One's compliment in binary is like 9's complement in decimal
  - Find the complement of each digit with b - 1

Finding the Twos Complement

- So, an easy way to find the Twos Complement of a binary number is to
  - Replace each bit with its one's complement
    - i.e. its complement with b-1, where b=2
    - i.e. the diminished radix complement, $(b^n - 1) - y$
    - i.e. flip each bit
  - Add 1
    - To get the radix complement $(b^n - y)$

Finding the Twos Complement

Exercise 15:

Find the twos complement of the following 4-bit numbers.

1. 0101
2. 1101
3. 0001

Which of the above represent negative numbers?

- Hint:
  - Replace each bit with its one's complement (i.e. flip each bit)
    - = diminished radix complement $(b^n - 1) - y$
  - Add 1
    - = radix complement $b^n - y$

Negative Numbers, Summary

- By representing negative numbers symmetrically using the Method of Complements, we get nice arithmetic properties
- Using Twos complement we can:
  - Store 256 different values (with 8 bit numbers)
  - Don’t have two 0s
  - Add/Subtract signed and unsigned values using the same algorithm/hardware

Binary Subtraction

Binary Subtraction

- We've already covered the complexity of binary subtraction
- By encoding negative values using Two's Complement, we can perform sutraction with the same method as addition
  - Recognise that instead of *subtracting* a positive number, we can *add* its **additive inverse**

$$A - B = A + (-B)$$

  - So a subtraction (A - B) is a two stage process
    - Find the twos complement of B
    - Add the two's complement of B to A

Manual Binary Subtraction

- 0101 - 1101 (storing numbers as 4 bits),
  - find the twos compliment of 1101
    - = 0010 + 1 = 0011
  - Add 0101 to this value
    - 0101 + 0011 = 10000
  - As we're only storing 4 bits, we dispose of the overflow, leaving
    - 0000 = 0

Exercise 16:

Solve the following binary subtractions

1. 11 - 01
2. 101 - 011
3. 0101 – 1100

- Hint:
  - A - B = A + (-B)
  - Find the twos complement of the second number (flip each bit and add 1)
  - Add the first and second numbers together

Binary Multiplication

Binary Multiplication

- To explain multiplication, we need to know 3 things
  - Multiplication of 1 bit values
    - Super simple!
  - Multiplication by 2^n
    - Like multiplying by powers of 10 in decimal
  - That multiplication distributes over addition
    - $a(b + c) = ab + ac$
    - i.e. we can break a number into parts, multiply each part, then add it together at the end

Multiplication of 1 Bit Values

A	B	A x B
0	0	0
0	1	0
1	0	0
1	1	1

- Multiplying 1-bit binary numbers is easy, as shown in the table

Multiplication by 2^n

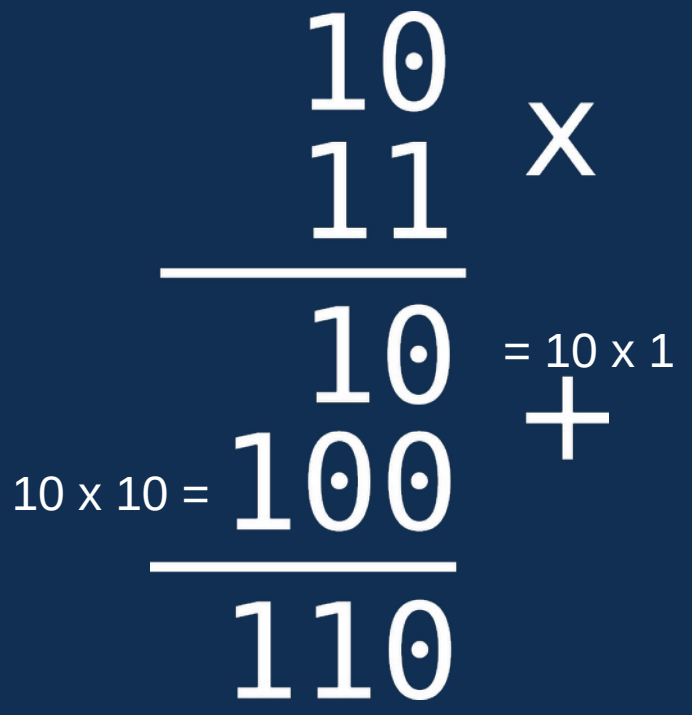
- Multiplying any number by 1 is straightforward
  - $1k = k$
- Multiplying any number by 2^n is the same as above, with a left shift of n
  - $2^n k = k << n$

```
  10          101
 x 1         x 1
----        ----
  10          101
```

```
  11          11
 x 10        x 100
----        ----
 110         1100
```

Multiplication distributes over addition

- A number *k* can be expressed as a sum of powers of 2, e.g. (in binary)
  - 11 = 10 + 1
  - 1011 = 1000 + 10 + 1
  - *k* = *a* + *b* + ...
  - Express second operand like this
    - 11 = 10 + 1
- Multiplication distributes over addition
  - *n* x *k* = *n*(*a*+*b* + ...) = *na* + *nb* + ...
  - 10 x 11 = 10 (10+1) = 10 x 10 + 10 x 1 = 110



Multiplication in Computers

- Multiplication by 2^n can be implemented as a bit shift
- Multiplication by all other numbers can be implemented as a series of shifts followed by addition

```
    1110
  x 1011
  ------
    1110
   11100
  000000
 1110000
 -------

10011010
```

Exercise 17:

Following the example of 1110 x 1011 (right), solve the following binary multiplications

1. 10 x 111
2. 101 x 100
3. 10101 x 10110

```
    1110
  x 1011
  ------
    1110
   11100
  000000
 1110000
 -------
 10011010
```

Binary Division

Division Terminology

Dividend ÷ Divisor = Quotient r. Remainder

- For example,
  - 7 ÷ 3 = 2 r. 1
    - Dividend = 7
    - Divisor = 3
    - Quotient = 2
    - Remainder = 1

Long Division in Decimal

- Arrange numbers, e.g. for 144 ÷ 12
- Work from left of dividend
  - Do 12s go into 1? No.
  - Do 12s go into 14? Yes, once, remainder 2 (=20)
- What's left? We've still got that 4, so bring it down. Continue working across 24
  - Do 12s go into 24? Yes, twice
- No remainder (or remainder < 12), so stop.

a.
```
    0
12 ⟌144
```

b.
```
    01
12 ⟌144
     2
```

c.
```
    01
12 ⟌144
    24
```

d.
```
    012
12 ⟌144
    24
```

Long Division in Decimal

- Align the divisor, D, with the most significant end of the dividend (= left)
- Let the “aligned” part of the dividend be X
- If X > D
  - The quotient digit becomes X ÷ D
  - Subtract D from X
- Else quotient digit is 0
- Repeat with D shifted one digit to the right

a.
```
    0
12 ⟌144
```

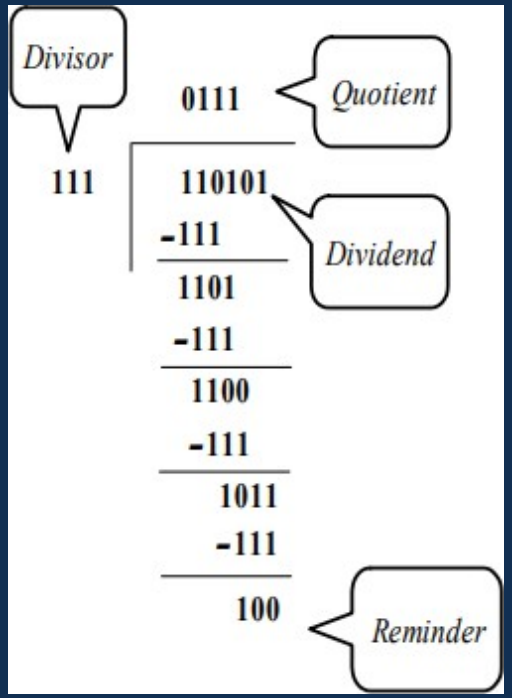
b.
```
    01
12 ⟌144
     2
```

c.
```
    01
12 ⟌144
    24
```

d.
```
    012
12 ⟌144
    24
```

Division in Binary

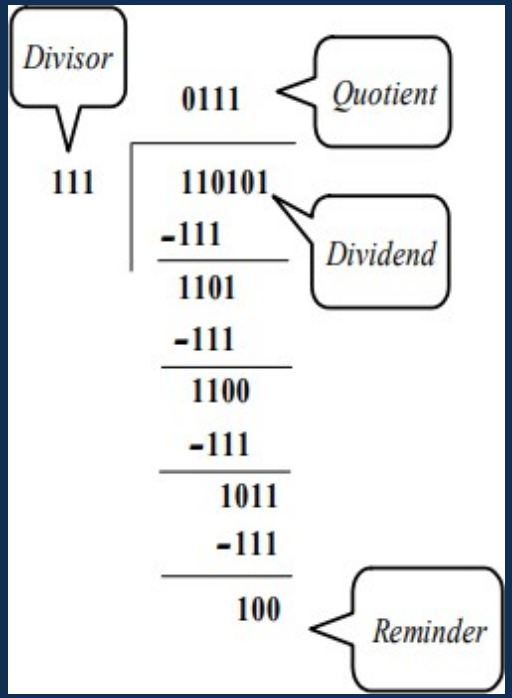
- Align the divisor with the most significant end of the dividend (=left)
- Let the portion of the dividend from its MSB to its bit aligned with the LSB of the divisor be denoted X
- Compare X and Y.
  - a) If X >= Y, the quotient bit is 1 and perform the subtraction X-Y.
  - b) If X < Y, the quotient bit is 0 and do not perform any subtractions.
- Shift Y one bit to the right and go to step 2.



Exercise 18:

Following the example of 110101 ÷ 111 (right), solve the following binary divisions:

1. 110 ÷ 11
2. 10101 ÷ 10
3. 110101 ÷ 101




Summary

Summary

- Number Systems, using Positional Notation
  - Decimal (base 10)
  - Binary (base 2)
  - Octal (base 8)
  - Hexadecimal (base 16)
- Method of Complements to symmetrically encode negative numbers
- Binary Arithmetic
  - Addition (+ Subtraction with Twos Complement)
  - Multiplication
  - Division