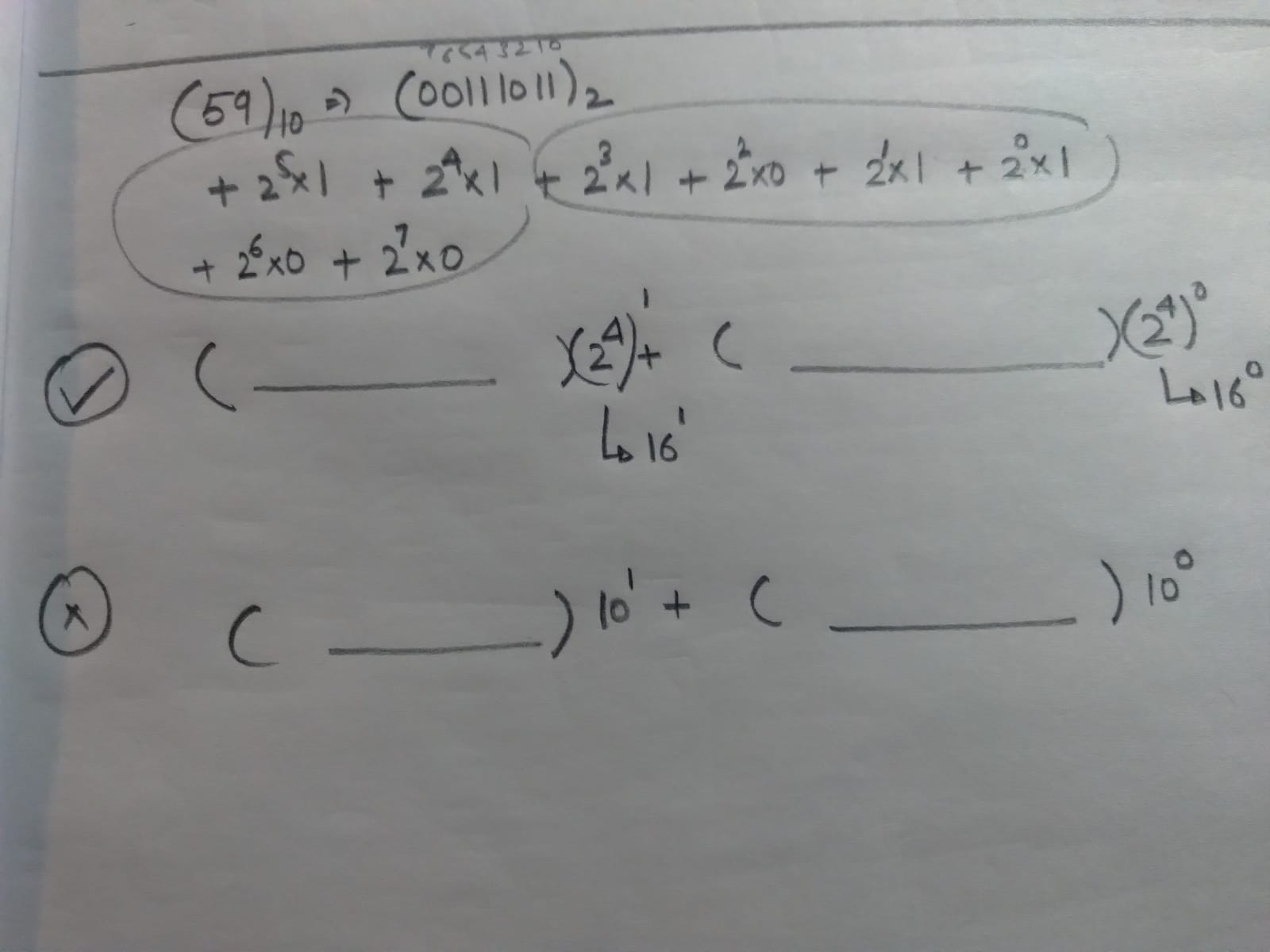
Low-level memory operations:

1. Binary number system
2. Derive the conversion formula of Decimal to Binary - from the first principles
3. Why is the Hexadecimal system used instead of the Decimal? Because it can provide information about the number as well as the memory space used to store it. Since memory is all about bits and the numbers in base 10 do not hold any correspondence with the memory space used, we can not represent them in memory explicitly. While writing the number in the decimal number system, we will also have to specify the memory space used to save the number. However, the numbers represented in binary and hexadecimal convey the information about the number as well the memory size used to represent it. So, numbers in binary and hexadecimal number systems can be represented in the memory. Hexadecimal is used instead of binary for better readability. <https://medium.com/@savas/why-do-we-use-hexadecimal-d6d80b56f026>

If a binary sequence needs to be written down in decimal notation, then the entire sequence has to be converted into a single decimal number contrary to the hexadecimal system where the numbers can be grouped in sets of 4.



In other words, the decimal number and the binary digits do not line up in the memory. For example, The number of bits required to represent a 3 digit decimal number varies depending upon the value of the number. Test the idea by representing random numbers from 100 to 999. On the other hand, a hexadecimal number of 2 digits will always take 8 bits or 1 byte of memory.

1. Arithmetic operations (+ - \* /) with binary numbers - convert it all to addition

Multiplication and division operations can be represented with consecutive addition and consecutive subtraction operations respectively. However, representing the subtraction as an addition operation is tricky.

1. *[In Nutshell]*

* **Method of 1s/2s complements for subtraction in the decimal number system.**
* **A binary Number system with a fixed number of digits with MSB reserved as a sign bit.**
* **1s Complement / 2s Complement of a positive number represents its negative number and vice versa.**

**[Fix the number of binary digits with MSB reserved as the sign bit**

**and**

**then obtain one’s/two’s complement of the abs(NegativeDecimalNumber)]**

* **Do not use the method (2s complement) to subtract 0 from a number or subtract a number from 0→ No need.**
* **Count from 0000 to 1111 both as signed and unsigned representation.**

*[Details]*

**Representation of negative numbers in binary** is closely tied to the way any two numbers are subtracted as addition by using the ‘Method of Complements’.

Understand the *‘Method of complements’*  by using the decimal number system and then extend it to the binary number system with a difference of using a fixed count of total bits for the representation in binary throughout the computations. The MSB represents the sign bit in the binary notation with 0 representing a positive number while 1 represents a negative number.

[Method of complements - Wikipedia](https://en.wikipedia.org/wiki/Method_of_complements)

[1s complement method](https://en.wikipedia.org/wiki/Ones%27_complement): 59 - 45 = 59 - (99 - 54) = 59 + 54 - 99

[2s complement method](https://en.wikipedia.org/wiki/Two%27s_complement): 59 - 45 = 59 - (100 - 55) = 59 + 55 -100

***[i] Represent the negative number as 1s complement or 2s complement and***

***[ii] then follow the corresponding approach.***

**[i] Representation of negative numbers:**

[Ensure the use of a fixed count of the total bits for representation throughout the computations]

Reserve the MSB for storing the sign of the number, that is if the total of 8 bits is available then use 7 bits for storing the magnitude and use MSB as the sign bit. 0 is for positive numbers while 1 is for the negative numbers.

* 1. 1s Complement: Write the positive counterpart of the negative number with the sign bit equals 0. After taking the 1s complement of the positive number, we get the negative number.
  2. 2s Complement: Write the positive counterpart of the negative number with the sign bit equals 0. After taking the 2s complement of the positive number, we get the negative number.

Note that in 1s complement, 0 has two values -0 and +0 but in 2s complement, 0 has a single representation.

[I want to find 2's complement of 0000 - Stack Overflow](https://stackoverflow.com/questions/33751662/i-want-to-find-2s-complement-of-0000)

[What is “2's Complement”? - Stack Overflow](https://stackoverflow.com/questions/1049722/what-is-2s-complement)

[Advantage of 2's complement over 1's complement? - Stack Overflow](https://stackoverflow.com/questions/11054213/advantage-of-2s-complement-over-1s-complement)

All modern computers use 2s complement for the representation of the negative numbers. It provides an extra number on the negative side.

**[No need to remember anything, derive all the concepts from the decimal number system]**

Note that: [Derive it all from the basics using the decimal number]

1s complement of ‘positive’ ←→ ‘negative’ representation

1s complement of ‘negative’ ←→ ‘positive’ representation

2s complement of ‘positive’ ←→ ‘negative’ representation

2s complement of ‘negative’ ←→ ‘positive’ representation

**[ii] Approach to obtain the subtraction using the complements**

**[No need to remember anything, derive all the concepts from the decimal number system]**

Note that the subtraction of 0 from a number using 2s complement will give the wrong answer if the rules for subtraction using 2s complement are followed exactly. If the number to be subtracted is 0, there is no need to do anything. Similarly, when subtracting a number from 0, just obtain 2s complement of the number to obtain the result.

Steps for subtracting two numbers with 1s(or 2s complement): (a-b)

1. Fix the number of bits with the MSB as the sign bit
2. Obtain b’s complement (1/2s)
3. Perform the steps
4. If the result turns out be negative then add - sign in the front
5. The number is stored in the binary notation by obtaining it’s complement
6. The complement at the last for the negative numbers will not be required if we dont obtain the complement at the last step if negative number comes up.
7. **Bit Manipulation using Bitwise operators** [Refer to C++ videos by Cherno]

[Compiler Explorer](https://godbolt.org/)

int num = 0xAF // Hexadecimal representation

int num = ob10101 //Binary representation (C++14 onwards)

std::bitset<>() [bitset - C++ Reference](https://www.cplusplus.com/reference/bitset/bitset/bitset/)

**[Used in combination]**

| *<<* | *(Unary) Bitwise Left* | *Shift bits to the left - Implications on the decimal representation - Limitations* |
| --- | --- | --- |
| *>>* | *(Unary) Bitwise Right* | *Shift bits to the right - Implications on the decimal representation - Limitations* |
| *&* | *(Binary) Bitwise AND* | *Masking using a bitmask - Removing certain bits - Applications* |
| *|* | *(Binary) Bitwise OR* | *Combine - Attaching certain bits to a bitset - can not set a particular bit to 0* |
| *^* | *(Binary) Bitwise Exclusive OR (XOR)* | *Hashing - Identify places with different bits between two numbers* |
| *~* | *(Unary) Bitwise NOT* | *Invert - Combination with &* |

Bitwise Left Shift << and Bitwise Right Shift >> UNSIGNED INTS

1. As the name says, they shift the bits by the specified digits and the empty bit is filled with 0.
2. a>>1; a=a>>1; a>>=1; a<<1; a=a<<1; a<<=1
3. Implications of shifting operation:

Bitshift Left: a>>n results in a number a\*2^n

Bitshift Right: a<<n results in a number a /2^n

The above points are based on two assumptions:

* The numbers on which they are applied are unsigned ints.
* There are an infinite number of bits available to represent the number.

Explanation:

* If the number is a signed int then it will have a sign bit at the MSB. Sifting it to the right or left will cause the undefined/compiler-specific behavior, which renders the resulting decimal number unpredictable. That is, in some compilers, the sign bit is not shifted at all.
* There will come a point when shifting the unsigned number to the left will cause a loss of bit from the number. For example, bit shifting 11111111 on an 8-bit memory register will result in 11111110, which is smaller than the original number.

1. The compiler automatically optimizes your code related to multiplication and division to be written as bitwise << or >> *where ever possible* in the assembly. For example

* 25\*16 → 5<<4 because 16 can be written as 2^4
* 25/16 → 5>>4 because 16 can be written as 2^4
* 25\*15 → 25\*15 because 15 can not be written as 2^n
* 25/15 →25/15 because 15 can not be written as 2^n

Bitwise AND(&), OR(|), XOR(^) and NOT(~) operators

1. **Bitwise AND(&)**: Masking using a bitmask - Removing certain bits - create a suitable Bitmask and perform AND
2. **Bitwise OR(|):** Combine - Attaching certain bits to a bitset - can not set a particular bit to 0
3. **Bitwise XOR(^):** Hashing **-** Identify places with different bits between two numbers - AND gives 1 when both the bits are 1, OR gives one when one of the bit is 1 and XOR gives 1 when both the bits are different
4. **Bitwise NOT(~):** Invert - Combination with other operators

Use cases -

1. **Identify a certain bit in a number ‘a’ as 0 or 1**.

* Create a bitmask that is a number ‘b’ that has all the bits as zero except the bit to be tested.
* Perform a&b → result
* Perform: if(result)
* result = 0 if the bit was 0. (as all the bits in result will be 0)
* result = anyNumber if the bit was 1. (all other bits except this bit will be 0 which provides a non-zero value to result)

1. **Isolate a group of bits from a number**

* Create a bit mask with the digits to be retained set as 1 and all others as 0.
* Perform a&b

1. **Toggle a certain bit of a number ‘a’ to 0 : Combination of & and ~**

* Create a bitmask number ‘b’ with the particular bit set to 0.
  + Use the NOT operator to create the bitmask. All zeros but the particular bit set as 1. Invert by using NOT. Result is the desired bitmask.
* Perform a&b.

1. **Attach certain bits to the left or right of a number ‘a’**

* Create a new bitset/number ‘b’ using << or >> operator
* Perform a|b

1. Multicore processors
2. Cycle speed or Clock speed per core [100 MHz or 1GHz]
3. IPC - Instructions per cycle
4. Cycle speed \* IPC = 10^8 Instructions per second
5. CPU performance governing factors - Need benchmarking for different systems; measured in FLOPS(Floating point operations per second) for HPC systems.
6. 32/64 bit CPUs - Memory Register size
7. Computer Memory - [Cost per amount of memory vs Speed]
8. [Primary] CPU Registers, Cache (L1, L2), RAM (Physical, Virtual)
9. [Secondary] Storage Devices, Input Sources
10. Static RAM, Dynamic RAM - Classification is based on the technology used to hold the electric charge. Both are volatile in nature. Static RAM uses flip flop transistor circuits and needs a continuous power supply contrary to the Dynamic RAM that uses transistors and capacitors. Dynamic memory needs to be refreshed periodically. Static memory is expensive and is a bit larger physically, so it is used for small memory Registers and Cache. Dynamic memory is relatively cheaper than static memory and it occupies relatively less physical space, so it is used for larger memory RAM.
11. Memory Management or Memory representation of a process -

(Higher Memory Address)

* Kernel
* Stack (Higher Memory Address → Lower memory Address)
* Heap
* Data (Initialized data, read-only; Initialized data, read-write; Uninitialized data, read-write)
* Text

(Lower Memory Address)

\* String literals are stored in the read-only data segment.

<https://www.geeksforgeeks.org/memory-layout-of-c-program/>

1. Buffer Overflow Exploits: <https://www.youtube.com/watch?v=1S0aBV-Waeo&t=117s>
2. Logical Address
3. Virtual Memory Address translation and Virtual Memory
4. Stack frames
5. Segmentation fault or core dump
6. Stack overflow - overshooting the stack bound
7. Fragmentation error
8. Size of a pointer
9. Byte Ordering - Endianness
10. Memory padding

<https://fylux.github.io/2017/07/11/Memory_Alignment/>

<https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/>

1. Bitwise operators
2. Serialization: <https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization>