CSE-4323:
Lab 1 - Amdahl's Law & ALU Simulation

GROUP 4 MEMBERS: Sam Trinh and Charles A-Darfah

PART A - Amdahl's Law

For our Amdahl’s Law Lab, we have decided to do bubble sort as our unoptimized algorithm and quick sort as our optimized algorithm.

Bubble sort is an easy to understand and quick to code sorting method, with it being one of the slowest methods to sort as it is an O(n^2) sorting method on average. Bubble sort simply checks if the first value of the array is larger than the next value of the array. If it is, then swap the two values in the array and continue incrementing. If it is not, then stop and go to the second value of the array. This continues until the entire array is sorted.

Quick sort is also relatively easy to understand and quick to code as well. It is majorly faster than Bubble sort as it is an O(n log n) sorting method on average. Quick sort works by selecting a “pivot” element from the array and then partitioning the other elements into two subarrays: those less than the pivot and those greater than the pivot. The algorithm then recursively sorts the subarrays in the same way until the entire array is sorted.

The main structure of the code is not touched. Therefore, 50000 random numbers are generated within 2 arrays and sorted using their respective algorithms.

For one of the tests, the numbers given are as follows:

Unoptimized Code		Optimized Code	
T_unoptimized	5.572000 seconds	T_optimized	0.008000 seconds
T_processing_unoptimized	5.571000 seconds	T_processing_optimized	0.006000 seconds

P = total_processing_unoptimized / total_unoptimized;
=> 5.5710 seconds / 5.5720 seconds = 0.999821

S = total_processing_unoptimized / total_processing_optimized;
=> 5.5710 seconds / 0.0060 seconds = 928.5

Measured_Speedup = total_unoptimized / total_optimized;
=> 5.5720 seconds / 0.008 seconds = 696.5

Theoretical Speedup = 1 / ((1 - P) + (P / S))
=> 1 / ((1 - 0.999821) + (0.999821 / 928.5)) = 796

Proportion of Optimizable Code (P):	0.999821
Speedup of Optimized Part (S):	928.500000
Measured Overall Speedup:	696.500000
Theoretical Overall Speedup (Amdahl's Law):	796.000000

In our experiment, the measured speedup of Quick sort compared to Bubble sort was 696.5×, while the theoretical speedup predicted by Amdahl's Law was 796×. The values are similar, which shows that Amdahl's Law gives a good estimate of the overall performance improvement. The small difference between the measured and theoretical speedup is due to real-world factors such as extra overhead from recursive calls and others that are not accounted for in the theoretical calculation. This experiment demonstrates the core principle of Amdahl's Law: even if almost all of a program is optimized, the small portion of the program that cannot be improved limits the total speedup.

Part B - ALU Simulation

The ALU is the computational unit of the CPU. The ALU handles the logic and arithmetic operations. This includes adding, subtracting, and'ing, or'ing, xor'ing, and more.

To implement two's complement for the sub operation we first subtract our second binary number from our first binary number (bin1Int - bin2Int). Then if the resulting sum is negative, that is when we can proceed with our two's complement. We then do the operation sum = 128 - (sum*(-1)); followed by sum += 128;. This adjusts the integer to the two's complemented version. For example, say I were to subtract 127 and 128. The difference would be -1. Then I subtract 1 from 128 which is 127. Then I add 128 again which would be 255. 255 in binary is 0b11111111. This aligns with what -1 is in two's complement.

Test Cases:

ADD	1st number	2nd number	Result
	10101010	01010101	11111111
	11111111	00000001	00000000
	00110011	00110011	01100110
	01010101	11111111	01010100

SUB	1st number	2nd number	Result
	10101010	01010101	01010101
	11111111	00000001	11111110
	00110011	00110011	00000000
	01010101	11111111	01010110

AND	1st number	2nd number	Result
	10101010	01010101	00000000
	11111111	00000001	00000001
	00110011	00110011	00110011
	01010101	11111111	01010101

OR	1st number	2nd number	Result
	10101010	01010101	11111111
	11111111	00000001	11111111
	00110011	00110011	00110011
	01010101	11111111	11111111

NOT		1st number	Result
		01010101	10101010
		00000001	11111110
		00110011	11001100
		11111111	00000000

XOR	1st number	2nd number	Result
	10101010	01010101	11111111
	11111111	00000001	11111110
	00110011	00110011	00000000
	01010101	11111111	10101010

EQUAL	1st number	2nd number	Result
	10101010	01010101	00000000
	11111111	00000001	00000000
	00110011	00110011	00000001
	01010101	11111111	00000000

For our ALU and addition, overflow would produce a negative number thus a negative flag and a carry flag would be raised. In most cases the MSB would be carried out as well. For subtraction overflow can only happen if a negative number is subtracted from a positive number produces a positive result ((-) - (+) = (+)) or if a positive number is subtracted from a negative number and you get a negative result ((+) - (-) = (-)). Overflow sets off certain flags within our ALU and lets us know it is happening and even what we can do with it.

PART C: Group Reflection

Sam Trinh - Report Setup, Part A, Part A Report and Part B Test Cases.
Charles A-Darfah - Part B, Part B Report.

Sam Trinh Learnings: I have learned and tested Amdahl's Law empirically. I have also learned how to properly calculate the optimization time and improvement factor as well. I learned the importance of testing ALU operations with edge cases (like subtracting a larger number from a smaller one) to ensure correctness in all scenarios.

Charles A-Darfah Learnings: I learned how two’s complement allows a fixed number of bits to represent both positive and negative integers, and how subtraction can be implemented as addition of a negative number. I practiced breaking a complex operation (subtraction using two’s complement) into steps: convert inputs, compute, handle negatives, and convert back. This strengthened my skills in bitwise thinking and careful indexing in C.