Assignment 1

$a \times 2^n$ with bit manipulation in C with a case study

Pedram Pasandide

Due Date: 2025, 11 July

1 Introduction

Before reading this section, take a moment and think about writing a code in any language to compute the result of $a \times 2^n$. What algorithm would you pick?

In the realm of integer arithmetic, bit manipulation can be a powerful tool to perform various operations efficiently. The Appendix describes a breakdown of common arithmetic and logical operations that can be performed using bit manipulation.

2 $a \times 2^n$ in C with bit manipulation

2.1 Implementation in C (1 point)

Create `a2n.c` file. Implement `my_a2n(a, n)` where `a` and `n` are constant value. Inside this function you must implement $a \times 2^n$ using bit manipulation. `main()` function will call `my_a2n(a, n)` and **print** the returned results. After compiling, I should be able to run your program with:

`./a2n a n`.

for example:

`./a2n 3 5`,

which computes 3×2^5.

2.2 Test Cases (1 point)

Using `a2n.c`, complete the following table when `n=29`, `n=30`, `n=31`, and `n=32`. Include the table in your `report.pdf`. You can validate your outputs with `pow(2, n)` from `math.h`.

n in 2^n (`1 << n`)	binary	number	compile status
1	...010	2	Completed
2	...0100	4	Completed
3	...01000	8	Completed
28	...01...(with 28 zeros)	268435456	Completed
29	...	...	...
30	...	...	...
31	...	...	...
32	...	...	...
50	...01...(with 50 zeros)	262144	...

Table 1: Computing 2^n.

Answer the following questions:

- What is the command line that you used to compile the program?
- Beyond what `n` the results are not correct?
- Why the results are incorrect from a certain `n`. Please be specific with numbers supporting your answer.
- Is there any compiler flag we can use to give us a warning avoiding such problems?

2.3 Case study for CPU time (2 points)

Your task in this section is to measure the efficiency of you function by comparing the CPU time with other alternative methods, including:

1. Using `pow(2, n)` from `math.h`,
2. Using `my_a2n(a, n)` where $a = 1$ from the previous section,
3. Using a `for` loop to compute 2^n.

Write the codes for this section in `main.c`. Create case studies to measure the CPU time in a `main()` calling above methods. Conduct multiple tests and complete the following tables in your report. I must be able to run your program with:

```
./main n m
```

where `n` is n in 2^n, and it repeats each method `m` times, measuring CPU time for each. For example, with `./a.out 28 100000000`, I get the following results:

```
Bit shift result: 268435456, CPU time: 0.211149 sec
Loop multiplication result: 268435456, CPU time: 4.075165 sec
pow() result: 268435456, CPU time: 1.829394 sec
```

m	**my_a2n(1,n)**	**pow(2, n)**	2^n with a loop
10			
100			
10000			
100000000			
1000000000			

Table 2: CPU time when `n=2` in seconds.

Times	**my_a2n(1,n)**	**pow(2, n)**	2^n with a loop
10			
100			
10000			
100000000			
1000000000			

Table 3: CPU time when `n=28` in seconds.

Answer the following questions:

- Why does the ratio of CPU times between different methods vary when we change the number of times (`m`) the computation is repeated? In another word, which one is more reliable 10^1 or 10^8?
- Why the CPU time ratio differs when the input `n` is changed? Which `n` is giving us a better case study results?

3 Report (1 point)

Create a PDF file named `report.pdf` that includes the documentation for your program. The report should cover essential information, such as how to compile and run the program as well as the tables above. **Create a section called Appendix in your report and include all the source code files as text (not screenshots).**

4 Submission On Avenue to Learn

Submit your assignment as:

- `a2n.c`
- `main.c`
- `report.pdf`

Please do NOT submit any zip files.

5 Appendix

Bit Manipulation Operations for Integer Arithmetic

The only one you need to study for this assignment is the first one, called **Multiplication and Division by Powers of Two**.

1. Multiplication and Division by Powers of Two

- **Multiplication by** 2^n:
  - Achieved using left shift: $x \times 2^n = x \ll n$
  - Example: $x \times 8$ is $x \ll 3$
- **Division by** 2^n:
  - Achieved using right shift: $x \div 2^n = x \gg n$
  - Example: $x \div 8$ is $x \gg 3$

2. Checking if a Number is Odd or Even

- **Odd Check**:
  - A number x is odd if $x\&1$ is 1.
  - Example: `if (x & 1) { /* x is odd */ }`

- **Even Check**:
  - A number x is even if $x\&1$ is 0.
  - Example: `if (!(x & 1)) { /* x is even */ }`

3. Setting, Clearing, and Toggling Bits

- **Set a Specific Bit**:
  - To set the n-th bit: $x \mid= (1 \ll n)$
  - Example: To set the 3rd bit, use `x |= (1 << 3)`
- **Clear a Specific Bit**:
  - To clear the n-th bit: $x \,\&= \sim (1 \ll n)$
  - Example: To clear the 3rd bit, use `x &= ~(1 << 3)`
- **Toggle a Specific Bit**:
  - To toggle the n-th bit: $x \oplus = (1 \ll n)$
  - Example: To toggle the 3rd bit, use `x |= (1 << 3)`

4. Checking if a Specific Bit is Set

- **Check if n-th Bit is Set**:
  - To check if the n-th bit is set: $x\&(1 \ll n)$
  - Example: To check if the 3rd bit is set, use `if (x & (1 << 3)) { /* bit is set */ }`

5. Counting the Number of Set Bits

- **Count Set Bits (Hamming Weight)**:
  - Various algorithms exist to count the number of 1s in the binary representation of a number.
  - Example: `__builtin_popcount(x)` in GCC or `popcount` intrinsic in other compilers.

6. Swapping Two Numbers Without a Temporary Variable

- **Swap Using XOR**:

$$x = x \oplus y$$
$$y = x \oplus y$$
$$x = x \oplus y$$

- Example: `x ^= y; y ^= x; x ^= y;`

7. Finding the Absolute Value

- **Absolute Value Using Bit Manipulation**:

$$\text{abs}(x) = (x \oplus (x \gg 31)) - (x \gg 31)$$

- Example: Given an integer `x`, `x >> 31` will be -1 for negative numbers and 0 for non-negative numbers.

8. Minimum or Maximum of Two Numbers

- **Find Minimum Using Bit Manipulation**:

$$\min(a, b) = b \oplus ((a \oplus b) \& -(a < b))$$

- Example: Given integers `a` and `b`, find the minimum without using comparison operators.
- **Find Maximum Using Bit Manipulation**:

$$\max(a, b) = a \oplus ((a \oplus b) \& -(a < b))$$

- Example: Given integers `a` and `b`, find the maximum without using comparison operators.

9. Checking if a Number is a Power of Two

- **Power of Two Check**:

$$(x \& (x - 1)) = 0$$

- Example: `if (x & (x - 1)) == 0 { /* x is a power of two */ }`

10. Negating a Number

- **Negation Using Bitwise NOT and Addition**:

$$-x = \sim x + 1$$

- Example: `-x` is equivalent to `~x + 1`

11. Rounding Down to the Nearest Power of Two

- **Rounding Down Using Bit Manipulation**:

$$\begin{aligned}
&x = x \,|\, (x \gg 1) \\
&x = x \,|\, (x \gg 2) \\
&x = x \,|\, (x \gg 4) \\
&x = x \,|\, (x \gg 8) \\
&x = x \,|\, (x \gg 16) \\
&x = x \,|\, (x \gg 32) \quad \text{// For 64-bit integers} \\
&x = x - (x \gg 1)
\end{aligned}$$

12. Finding the Most Significant Bit (MSB)

- **Finding MSB Using Bit Manipulation**:
  - `int msb = 0; while (x >>= 1) { msb++; }`
  - Alternatively, use built-in functions like `__builtin_clz` in GCC to find the number of leading zeros.

Summary

Bit manipulation can greatly optimize integer operations by reducing the need for multiple instructions. These techniques are particularly useful in performance-critical applications, embedded systems, and algorithms where efficient computation is essential.

However, it's important to balance the use of bit manipulation with readability and maintainability of code. While bitwise operations can be faster, they can also make the code harder to understand and debug.