Assignment 1

Programming $a \times 2^n$ with bit manipulation in C and adding the function as a shared library to Python with a case study

Pedram Pasandide

Due Date: 2024, 08 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.

Let's delve deeper into how bit manipulation can be used for multiplication and division by powers of two, with detailed examples and binary representations.

Multiplication and Division by Powers of Two

When you multiply a number by a power of two, you're effectively shifting its binary representation to the left by a certain number of places. This is because each left shift corresponds to multiplying the number by 2.

Example 1: Multiplying by 2^1 (i.e., multiplying by 2)

Consider the number $x = 5$:

- Binary representation of 5: 0101
- Operation: compute 5×2^1 **or** shift the binary left by 1 bit with `5<<1`
- Binary after shift: 1010 (shift left by 1 place)
- Equivalent decimal: 10

When you divide a number by a power of two, you're effectively shifting its binary representation to the right by a certain number of places. Each right shift corresponds to dividing the number by 2 and discarding any fractional part (i.e., integer division).

Example 2: Dividing by 2^3 (i.e., dividing by 8)

Consider the number $x = 25$:

- Binary representation of 25: 111001
- Operation: compute $25/2^3$ **or** shift the binary right by 3 bit with `25>>3`
- Binary after shift: 00011 (shift right by 3 place)
- Equivalent decimal: 3

The following table shows a few more examples:

Operation	Decimal (x)	Binary Before	Binary After	Decimal Result
5×2^1	5	0101	1010	10
3×2^2	3	0011	1100	12
7×2^3	7	0111	0111000	56
$12 \div 2^1$	12	1100	0110	6
$20 \div 2^2$	20	10100	0101	5
$25 \div 2^3$	25	11001	00011	3

Table 1: Multiplication and Division by Powers of Two using Bit Manipulation.

You can double check the binary representation of integers on online converter, or you can write the code for yourself which here it is not necessary. The following C code shows **Example 2** computing $25/2^3$ using bit manipulation.

```c
#include <stdio.h>

void main() {
 int a = 25;
 int result = a >> 3;
 printf("left Shift: %d\n", result); // output: 3
}
```

2 Bit manipulation (1 point)

The binary representation of integer 1 is equal to ...01. When `a = 1`, then `1 << n` simply computes $1 * 2^n$ (or just 2^n) by shifting ...01 with n bits. Having this method, we don't need to carry out the multiplication n times. Take a look at the following table.

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

Table 2: Computing 2^n with `1<<n`.

Obviously the answer when `n=50` (or 2^{50}) is not correct, and during compiling we should get the following Warning:

```
In function ‘main’:
warning: left shift count >= width of type [-Wshift-count-overflow]
int result = a << 50;
```

Complete the above table and include it in your `report.pdf`. Fill up the table when `n=29`, `n=30`, `n=31`, and `n=32`. At what `n` the compiler gives a warning? Why?

3 $a \times 2^n$ in C with bit manipulation

3.1 Implementation (1 point)

Create a new `.c` file and name it `a2n.c`. Copy past the following format into the file. `my_a2n()` is the name of a function we chose to receive two constant integers `a` and `n`. Inside this function you must implement $a \times 2^n$. `main()` is the main function calling a user defined function using `my_a2n(a, n)`, and printing the results after the computed value is returned from `my_a2n()`.

```c
// a2n.c
#include <stdio.h>

int my_a2n(const int a, const int n) {
 // Code: here you implement a*2^n computation using bit
    manipulation
}
```

```c
int main() {
 const int a = 3; // you can try for different a
 const int n = 4; // you can try for different n
 printf("Result: %d\n", my_a2n(a, n));
 return 0;
}
```

Compile and run the code to make sure it works and the results are consistent with previous section. Here you might not get the warnings let's say when `n=50`, but the results must be the same as previous section.

3.2 Creating the shared library (1 point)

We need to create an object file that can be shared with Python. We name this file as `liba2n` using:

1. For **Linux/Mac**:

`gcc -shared -o liba2n.so -fPIC a2n.c`

2. For **Windows**:

`gcc -shared -o liba2n.dll -fPIC a2n.c`

The `gcc` command compiles the `a2n.c` file into a shared library (`liba2n.so` on Linux/Mac or `liba2n.dll` on Windows). The `-shared` flag tells the compiler to produce a shared object (dynamic library). The `-fPIC` flag is required for position-independent code, which is necessary for shared libraries.

4 Using the library on Python with case study

4.1 Adding the shared library to Python

I already did this part and you don't need to make any changes here. The Python code is given in `use_a2n_example.py`. You will not submit the `use_a2n_example.py` file. If you did make any changes within this Python code, make sure the names are still the same.

The purpose of `use_a2n_example.py` is to call the function we created in C named `my_a2n()`. Now, the function implementation is inside a shared memory that we called it `liba2n.so`. Here

is the explanation what we did in this Python code.

First we load the shared library in Python. In `use_a2n_example.py`, `ctypes.CDLL` is imported to load the shared library (`liba2n.so`) and save it into `my_lib`.

```
import ctypes
my_lib = ctypes.CDLL('./liba2n.so')
```

The argument types (argtypes) and return type (restype) for the compute function to match the C function signature. You can try the code without it and see the different when the inputs given to function in Python doesn't match with the original format in C.

```
my_lib.my_a2n.argtypes = (ctypes.c_int, ctypes.c_int)
my_lib.my_a2n.restype = ctypes.c_int
```

A user defined function named `power_of_two(n)` is implemented in Python to compute 2^n without using built-in `pow()` function in Python.

```
def power_of_two(n):
  result = 1
  for _ in range(n):
     result *= 2

  return result
```

We can now call the compute function with the desired parameters directly from Python.

```
// Constants in a*2^n
a = 1
n = 30

// my_a2n() is our C code package
print(f"my_a2n({a}, {n}): {my_lib.my_a2n(a,n)}")

// pow() is a built-in function on python
print(f"{a} * pow(2,{n}): {a*pow(2, n)}")

// power_of_two() is developed from scratch in Python
print(f"{a} * 2^{n}     : {a*power_of_two(n)}")
```

4.2 Case study $a \times 2^n$ with three different methods (2 points)

Now I can run the Python code, e.g. with `python3 use_a2n_example.py` , computing $a \times 2^n$ with the following methods:

- `my_a2n(a,n)` where the implementation is located in a user defined shared library called `liba2n.so` with a C source code.
- `a*pow(2, n)` where `pow()` is a built-in Python package.
- `a*power_of_two(n)` where it uses a for loop in Python to carry out n times multiplication.

When `a=1` and `n=500` we have:

```
my_a2n(1, 500): 1048576
1 * pow(2,500): 327339060789614187001318969682759915221664204604306478948329136809613379640467455488327009232590415715088668412756007100921725654588539305332852758937 6
1 * 2^500     : 327339060789614187001318969682759915221664204604306478948329136809613379640467455488327009232590415715088668412756007100921725654588539305332852758937 6
```

Conduct multiple tests and complete the following table in your report.

n	my_a2n(a,n)	a*pow(2, n)	a*power_of_two(n)
1	2	2	2
2	4	4	4
3	8	8	8
30	...	...	...
31	...	...	...
32	...	...	...
50	262144	...	...

Table 3: Case study on Python.

Although the answer given by `my_a2n(a,n)` when `n=50` is wrong, we don't see any warning or errors here. **Which method is faster and why?** Answering this question you need to conduct multiple tests and measuring CPU time in Python for each method above.

5 Submission On Avenue to Learn

Only `report.pdf` file and `a2n.c`. Do **NOT** submit `use_a2n_example.py` as I will use the same version I sent you.

6 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.