Report of the 1st project of CSC3060

(1) Why bitwise operations work for arithmetic:

Firstly, the decimal system is an arithmetic method that is easy for humans to understand, but machines can only understand the binary system, which consists of 0s and 1s.

For addition functions, the most difficult part is handling the carry operation. Given numbers a and b, we can use a^b (XOR) to perform addition while ignoring carries (1+1 = 0, 0+0=0, 1+0=1, 0+1=1). We can also use a&b to find the bits that require a carry (a carry is needed only when the corresponding bits of a and b are both 1). In decimal, a carry means adding one to the next higher digit, while in binary, this operation is equivalent to a left shift by 1 bit (i.e., (a&b) << 1). The carry operation is repeated until this value becomes 0, meaning there are no more bits equal to 1, and the addition process is complete.

For subtraction, understanding addition makes it much simpler, because a-b is equivalent to a+(-b). Here, we only need to find -b (which is the two's complement of b), and then call the previously implemented addition function add(a, -b). As shown in the lecture slides (CA3060-2-DataRepresentation(1)), -b can be obtained by taking the bitwise complement and adding one (~b+1).

For the multiplication step, similar to addition, it's important to understand the reason and process of carrying over. Each digit of a binary number has a fixed weight (i.e., 2**n). For example, in the process of A × B, B can be broken down into b_n*2**n + b_(n-1)*2**(n-1) + ... + b_1*2**1 + b_0*2**0 (where b_n to b_0 can only be 0 or 1). Therefore, to perform binary multiplication of A × B, it is essentially equivalent to (A*b_n*2**n) + (A*b_(n-1)*2**(n-1)) + ... + (A*b_0*2**0). Multiplying A by 2**n is essentially a left shift of A by n bits, while the process involving b_0 to b_n is essentially a logical right shift. Since we are dealing with 32-bit numbers, we can complete the multiplication by repeating this operation 32 times.

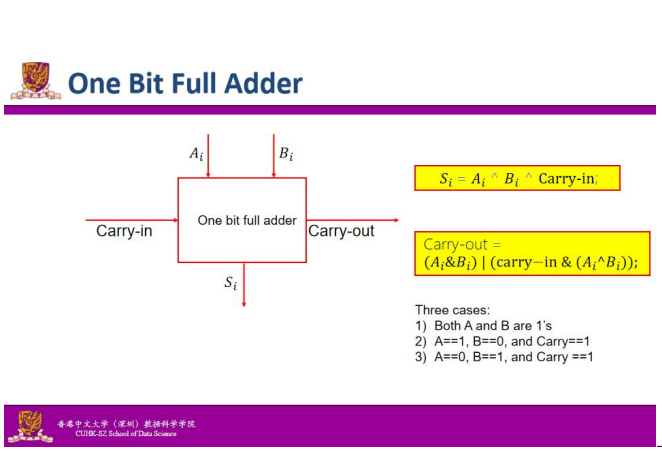
For devision step,the Bitwise division is essentially a hardware-level simulation of binary long division. Its core logic utilizes a "sliding window" approach, determining the quotient bit by bit through 32 iterations. The implementation first uses an arithmetic right shift of 31 bits to generate a mask, converting the operands to their absolute values to eliminate the effects of two's complement representation.In each iteration, the most significant bit of the dividend is "shifted" into the remainder window (i.e., the remainder is left-shifted and a new bit is added), and then the divisor is subtracted from the remainder. This cleverly uses the sign bit of the subtraction result (diff >> 31) to replace traditional if-else comparisons: if the sign bit is 0, it

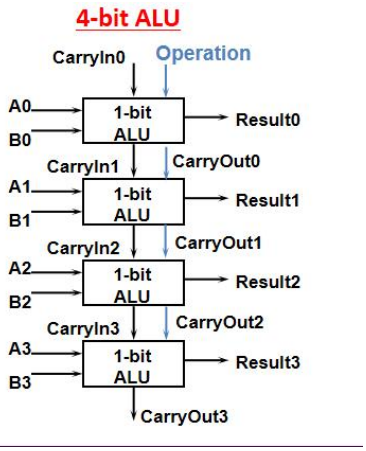
means "sufficient to subtract," the current bit of the quotient is set to 1, and the remainder is updated; if it is -1, it means "insufficient to subtract," the quotient bit is set to 0, and the remainder remains unchanged.This "shift-trial subtraction-quotient assignment" process is repeated 32 times, ensuring that every bit of the dividend is processed. Finally, the sign mask is used again to restore the correct sign to the result.

The approach for the modulo function is the same as for the division function, with the only difference being that the remainder must have the same sign as the dividend. This will be explained in detail in Part 2.

(2) How do I come up with such a solution:

The idea for addition comes from the PowerPoint slides in the second section. Here, a single-bit full adder and a 4-bit ALU are shown. We can then imagine a 32-bit ALU. The carry-in and carry-out signals clearly explain the hardware implementation of carries. Returning to C++, when calculating a + b, we first use XOR0 = a ^ b to get the sum without considering carries, and then use Carry0 = (a & b) << 1 to get the carry result. Now we need to add these two results, but adding these two results may still generate a carry, leading to a recursive process. We repeat this process. Because the width of a 32-bit register is only 32 bits, a carry signal is like a baton, shifting at least one bit to the left in each cycle. Even in the most extreme case (e.g., 0xFFFFFFFF + 1), the carry will propagate from bit 0 to bit 31, and finally disappear (become 0) after the 32nd shift. In reality, 32 carries might not be necessary, but for safety, considering the most extreme case, we repeat this carry operation 32 times.

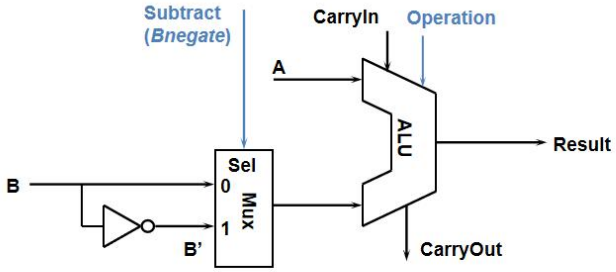




Subtraction is only one step more complicated than addition: it involves changing b to -b, which is a simple process of bitwise negation.

Subtraction Operation

- 2's complement: Take inverse of every bit and add 1 (at c_{in} of first stage)
  - A + B' + 1 = A + (B' + 1) = A + (-B) = A - B
  - Bit-wise inverse of B is B'



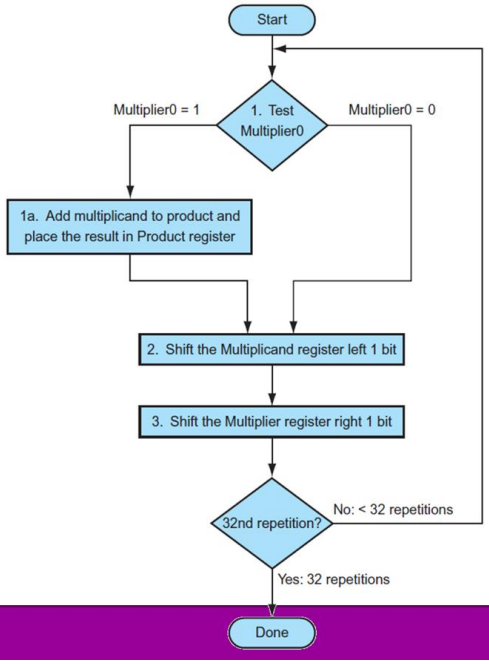
The multiplication operation was inspired by this diagram, which gave me the idea to multiply the multiplicand by each digit of the multiplier, starting from the least significant digit to the most significant digit, and then add the results together. (Each time we move from a lower digit to a higher digit of the multiplier, the multiplication result needs to be shifted one position to the left before being added to the result.)Subsequently, the next slide presented a more detailed approach. Translating the ideas on this slide into C++ code was quite easy. The only difficulty for me was representing the multiplication of the lower bits of the multiplier with the multiplicand, which will be discussed in the next section. Similarly, repeating the approach I mentioned earlier 32 times yields the final multiplication result.Additionally, I considered the multiplication of negative and positive numbers, and the presentation slides provided two solutions. The first method involves representing the sign first and then the absolute value. However, I used the solution provided by the LLM: in the two's complement system, since overflow is automatically ignored, the lower 32 bits of the 32-bit result do not require any additional consideration of the sign.(https://gemini.google.com/share/5840264ef158)

Multiplication

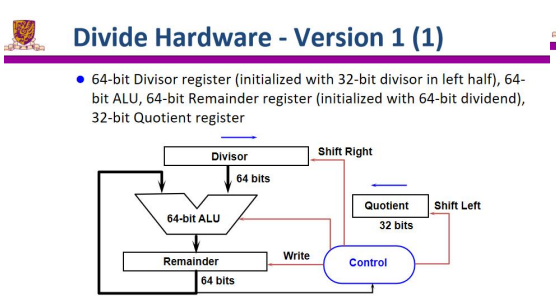
- More complicated than addition
  - Can be accomplished via shifting and adding

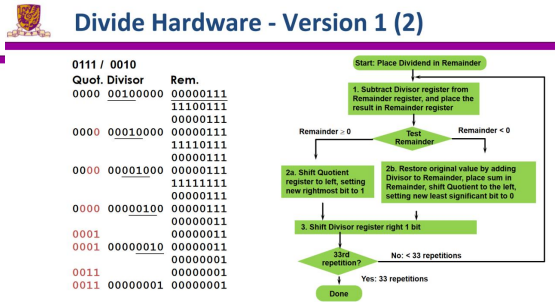
```
    0010      (multiplicand)
×   1011      (multiplier)
    0010
   0010
  0000        (partial product array)
 0010
00010110      (product)
```

- Double precision product is produced
- More time and more area are required



Division is essentially the inverse operation of multiplication. Inspired by the lesson materials, we can still solve it using the concept of carrying over. However, we need to compare the remainder (initially the entire dividend) with the divisor.Version 1 (1) of the lecture slides provides a solution approach. Both the divisor and remainder registers are 64-bit. The divisor is placed in the high 32 bits, and the dividend is placed in the low 32 bits (the dividend is placed in the remainder register), while the quotient is set to 32 bits. Because the number of bits in the divisor is initially much higher than the dividend, the first approximately 30 comparisons will show the remainder is less than the divisor. Only after that will the quotient be incremented. The specific implementation process involves first shifting the remainder left by one bit to free up the least significant bit, then taking the most significant bit of the current dividend and using the OR operation. The new remainder is equal to the old remainder × 2 + the current most significant bit of the dividend. Then the dividend is shifted left by one bit, preparing for the next bit's operation. When determining whether the current remainder is greater than the dividend, the difference is calculated, and then the difference is right-shifted by 31 bits to obtain the sign bit. If it is 0, the sign is positive; if it is -1, the sign is negative (11...11, this is an arithmetic right shift). This is then used in the if statement. If it is positive, it means subtraction is possible, and the quotient is shifted left and incremented. If subtraction is not possible, only the quotient is shifted left. This operation is performed 32 times to obtain the answer.





For the modulo function, the basic approach is the same as division; the only thing to note is Since the assignment requires the remainder to have the same sign as the dividend, the effect of positive and negative signs needs to be considered. My approach is as follows: take the absolute values of a and b and convert them to unsigned integers (uint32_t). After the loop, the remainder is a non-negative absolute value. Then, use (remainder ^ sign of the dividend) - sign of the dividend to obtain the signed remainder. This ensures that the remainder has the same sign as the dividend.

(3) What difficulties I encounter, and how do I solve them

Fristly,My initial version of the addition function didn't involve casting to uint32_t and then back to int32_t. However, I found that this method didn't handle overflow situations well, such as with 0x7FFFFFFF + 1. So I consulted an LLM (Large Language Model), and it suggested converting to an unsigned integer first, and then back to a signed integer, which handles overflow situations more effectively.(https://gemini.google.com/share/f947be194c7d)

Secondly,In a multiplication function, how to represent the multiplication of the least

significant bit with the multiplicand,I was initially puzzled by this problem for over ten minutes, but then I figured out that I could convert the least significant bit of the multiplier into its two's complement representation, and then perform a bitwise AND operation with the multiplicand to obtain the partial result. The principle behind this is that the two's complement of 0 is still 0, so performing a bitwise AND with the multiplicand still results in 0. However, the two's complement of 1 is -1, which is represented as 111...11 (all ones). Therefore, during the bitwise AND operation, the result only depends on the multiplicand, thus successfully solving this difficulty.

Lastly,in the division function, I realized that I needed to take the absolute value of both the dividend and the divisor, because bitwise operations only have the expected semantics when the operands are non-negative; directly using negative numbers would cause the sign bit to participate in the shifting, leading to distorted comparison results. However, I was quite confused about how to take the absolute value.In the division function, I realized that I needed to take the absolute value of both the dividend and the divisor because bitwise operations only have the expected semantics when the operands are non-negative; directly using negative numbers would cause the sign bit to participate in the shift operation, distorting the comparison result. However, I was quite confused about how to take the absolute value. My initial logic was simple: negate negative numbers and add one, and leave positive numbers unchanged. But this involved checking the sign bit. I wanted to use a unified formula that would work for any number with an unknown sign. I thought of using sign representation, which is right-shifting by 31 bits; positive numbers correspond to 0, and negative numbers correspond to 1. Then, how to use -1 to negate and 0 to keep the number unchanged? I thought of using XOR, that is, XORing the operand with the operand right-shifted by 31 bits. Then, how to represent adding one? This is where I directly subtracted the operand right-shifted by 31 bits, conveniently using the previously defined subtract function. This solved the problem.

(4) Time complexity for your implementation

Add: o(1)
Subtract:o(1)
Multiply:o(1)
Divide:o(1)
Modulo:o(1)

(5) Thoughts about unit testing

I noticed that the test files contain many tests involving 0 and -1, such as TEST(AddTest, WithZero) and TEST(MultiplyTest, WithZero). When writing code, I often overlook the case of 0. For example, when performing division, if the dividend is 0, the code logic might fail if not handled carefully. However, TEST(DivideTest, ZeroDividend) helped me verify that my code returns the correct result even when the input is 0, which made me realize the importance of handling special values.

My subtract function actually reuses the add function. The principle is that a - b is equal to a + (~b + 1). The test file not only tests the result of subtract but also the result of add. Since the subtract tests (such as TEST(SubtractTest, PositiveNumbers)) pass, it indirectly proves that my add

function has no problems handling negative number conversions.

In the test file, each function ends with two RandomNumber tests, such as TEST(AddTest, RandomNumber1). If these random number tests pass, it indicates that my bitwise operation logic is generalizable to any 32-bit integer.

(6) LLM usage statistics

The specific links have been included at the end of the corresponding paragraphs.