Cody Cook
11/30/2018

Unbounded Modular Arithmetic Using Binary Operations

Code for Realizing Large Integers in Cryptographic Algorithms

ABSTRACT

Computers have afforded us the ability to create vast expansive programs without needed to understand the underlying hardware. Compilers and interpreters have allowed users to key in phrases and commands which will then, seemingly magically, be converted into the world of electrical voltages across transistors and then appear back in a format understood by people. Programmers can multiply, divide, even calculate roots and logarithms using a single line of code and think no more of it. However, the gap in the understanding of these operations and how they are executed on a processor can lead to serious decimas. What one may typically think of as a single operation may be dozens of operations in machine code. As well, though it may not always be evident, the size of numbers are beholden to the bit size of processors (32-bit, 64-bit, etc.). This becomes an issue when certain algorithms, many of those in cryptography, require hundred or thousand-digit numbers, which would need many more bits than most processors are capable of handling. While many cryptographic protocols have ways of getting around this issue that don't even require initializing a full 1024-bit number at once, it may still prove useful to have a way to generate and manipulate such numbers for testing and designing of cryptographic algorithms without added complications such as those. This paper describes a structure for holding integers unbounded by processor size and the binary algorithms which make it useful in standard programming applications.

THE CONTAINER

To contain an unbounded number of bits, one simply needs to have an array of some data type. All data types are inherently stored in binary, so simple storing all the necessary bits in some form and maintaining their order is all that is necessary. For this implementation however, *unsigned int* was used because then if only the bottom most block is filled, whatever the compiler interprets that number to be is the true value for our purpose. However, the interpreted values in each block should be largely ignored. If one has an array with {2, 5} where each number has four bits, the true value with all the bits aligned is 37, seen as follows:

8	**4**	**2**	**1**	**8**	**4**	**2**	**1**		
0	0	1	0					=	1
				0	1	0	1	=	2

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**		
0	0	1	0	0	1	0	1	=	37

It is therefore easy enough to store the values using arrays of unsigned integers but interpreting and operating on them is essence of the problem.

SIMPLE OPERATIONS

Processors are bounded by a subset of operations which can be easily executed using logical circuits. These operations though, are then gifted the ability to be executed extremely fast taking only one operation of the processor. Among the well-known are the addition, logical, and shift operations. The addition and shift operations will prove especially useful and can be implemented for the whole object with minor modifications.

Addition, unfortunately, cannot be computed by summing each respective block of two numbers. In normal addition, when the sum is greater than the value of the place, a carry is taken to the next place. Likewise, this is done in binary. But for each block's sum, if there is a carry on the last bit, it needs to be carried into the sum of the next block. This can be achieved by looping over each block as follows:

```
Unbounded Int A, B
for (i from 0 to #Blocks) {
        $A_i = A_i + B_i + Carry$
        Carry = $A_i < B_i$
}
If (carry)
        A.push_back(1)
```

Here B is added to A. For each iteration of the loop the value of A and B at that respective chunk is summed with the carry from the previous chunks sum. Then the new carry is calculated by taking advantage of integer overflows. If the new value in that block of A is less that B, then the sum caused an integer overflow which in this scenario should result in a carry and so the expression evaluates to 1 and places that value in the carry for the next blocks sum. If the final block's sum results in a carry, simply place that carry in a new block at the back of the array.

Subtraction is simple enough to implement using Two's Compliment. By flipping each individual bit of a number and then adding 1 the Two's Compliment of a number is found and the resulting sum will be as if the original number was subtracted. Two's Compliment can be found as follows:

```
Unbounded Int A
for (i from 0 to #Blocks) {
        $A_i = A_i$ ^ UINT_MAX
}
add(1)
```

UINT_MAX is the maximum number that can be held by the unsigned int data type and is comprised of 1's at every place. By taking every bit XOR 1, the original bit is flipped. Then using the previously described addition, 1 is added. After Two's Compliment, the original addition function simply needs to be modified to not push back the final carry if the number is negative. To do this, a sign flag is simply added to the overall object that is triggered in the negate function. This is useful because normally for signed integers if the leftmost bit is 1 the number is negative. Being unbounded, this structure has no final leftmost bit.

Shifting is a simple operation that shifts every bit in a string one place to the right or to the left but corresponds to multiplying and dividing. Knowing that each place of a digit, $\boldsymbol{n}$, holds a value of $\mathbf{2^n}$, it is evident that shifting every bit one place to the left changes its value to $\mathbf{2^{n+1}}$ and therefore the whole number $\boldsymbol{x}$, becomes $\mathbf{2x}$. Left shifting:

32	**16**	**8**	**4**	**2**	**1**		
0	0	1	1	0	1	=	13
0	1	1	0	1	0	=	26
1	1	0	1	0	0	=	52

Cody Cook
11/30/2018

To implement this shift, the built-in shift function can be used and then the carries need to be managed like with addition. This operation can be implemented by iterating over every block as follows:

```
Unbounded Int A
for (i from 0 to #Blocks) {
        temp = left most bit of A_i
        A_i = (A_i << 1) +  Carry
        Carry = temp
}
If (carry)
        A.push_back(1)
```

Before any operations are done on the block, the left most bit is saved to be used as the carry in the next block. The block is then shifted left once and the carry from the previous block is added. Again, if there is a final carry, it is added to a new block.

A benefit of this data structure is also that entire blocks can shifted by any number of blocks in what amounts to one operation (programmatically not for the processor). This can be done by shifting each block by its index plus the shift number. The following is an implementation of left shift by block for some number of blocks, *shift*:

```
Unbounded Int A
Int shift
for (i from (A.size() - 1)  to  shift) {
        A[i] = A[i - shift];
}
```

Multiplication can be achieved by expanding on the shift operation. From the previous example, one can see that any multiple of a binary string can be achieved by shifting and adding a certain number of times. For example, to find 13 x 5, one would take the binary expansion of 5 with 13 x (2^2 + 2^0) = 13 x 2^2 + 13 x 2^0. By summing 13 shifted zero times with 13 shifted twice, the result of 13 x 5 = 65 is found:

64	32	16	8	4	2	1		
0	0	0	1	1	0	1	=	13
0	1	1	0	1	0	0	=	52
1	0	0	0	0	0	1	=	65

This operation can be implemented by shifting the base number A left for every bit in the multiplicand B. If the bit at that point in B is 1, add the shifted A to a final sum. This can be achieved as follows:

```
Unbounded Int A, B, shift
Shift = A
A = 0
for (i from 0 to #Bits in B) {
        shift << 1
        If (bit i in B = 1)
                A.add(shift)
}
```

BINARY INVERSE

After fully realizing binary multiplication the next, obvious step is division. However, taking the previous scheme for multiplication and shifting right, which is known as division by two, would not achieve the correct results. With that method, trying to find 13 / 5, the binary expansion results in a fraction 13 / (2^2 + 2^0) where you cannot factor out the denominator to have 13 / 2^2 and 13 / 2^0. This method does not work but relating addition and subtraction to multiplication and division one can recall two's compliment. By transforming one of the operands in a certain way, the number acts as if it were a negative in the normal addition operation. Such a process can be achieved for division of numbers by finding a transform corresponding to an inverse. To do this it is necessary to expand the view of binary numbers into fractions. Taking the bit to the right of the 0th bit would give 2^{-1} or $\frac{1}{2}$. Going further you would get $2^{-2} = \frac{1}{4}$, $2^{-3} = \frac{1}{8}$, etc. With fractional values an inverse can be found by using what amounts to Euclidean Division, as follows:

```
Unbounded Int A, Num, Den, Digit
Den = A
A = 0
While (A.size() <= baseSize) {
        digit = 1;
        while (x < Den) {
                Num << 1
                digit << 1
        }
        A.add(digit)
        Num = Num - Den
        Den = Den * Digit
        }
        A >> 1
```

The main loop runs until the inverted number becomes the size of the base as described previously. A digit is shifted left until it is greater that the denominator. When that digit is found it is added to the inverted number. The new numerator is found by subtracting the denominator. The new denominator is found by multiplying itself by the digit. Finally, after the loop has run, the number is shifted to the right by one to remove the ones place corresponding to 2^0. The resulting number is in the following form:

2^{-8} 2^{-7} 2^{-6} 2^{-5} 2^{-4} 2^{-3} 2^{-2} 2^{-1}

With an inverted number, a division function can be implemented just like multiplication, but shifting to the right like as follows:

```
Unbounded Int A, B, shift
B.invert()
Shift = A
A = 0
for (i from 0 to #Bits in B) {
        shift >> 1
        If (bit i in B = 1)
                A.add(shift)
}
```

However, in the case where A is a multiple of the divisor B, the result will be one less that expected and the remainder will be 0.999 repeating. If the inverted number were long enough the limit of the remainder would approach one, correcting the number. For example, with 6 /3:

64	**32**	**16**	**8**	**4**	**2**	**1**		
0	0	0	0	1	1	0	=	6

$\frac{1}{128}$	**$\frac{1}{64}$**	**$\frac{1}{32}$**	**$\frac{1}{16}$**	**$\frac{1}{8}$**	**$\frac{1}{4}$**	**$\frac{1}{2}$**		
0	1	0	1	0	1	0	=	~0.333

$\frac{1}{128}$	$\frac{1}{64}$	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$		
0	0	0	0	0	0	1		
0	0	0	0	1	1	0		
0	0	1	1	0	0	0		
0	0	1	1	1	1	1	=	~0.97

If the number of repeating ones in the remainder is the same size or greater than that of the numerator, the remainder can be set to zero and one added the result of the division.

It should be noted that the binary inverse function takes significant time to calculate. Each time a digit is found greater than the current denominator, a multiplication of unbounded numbers must occur which requires potentially *N* shifts and additions for a number sized *N*. As well these multiplications cause the denominator's size to expand massively beyond the original size of the number. However, once the inverse of a number *p* is calculated, calculating any number divided by *p* proves to be as quick as a normal multiplication.

FAST MODULO

With the implementation of division, it becomes possible to implement the modulo operation crucial for modular arithmetic and cryptographic algorithms. The operation can be realized as follows:

```
Unbounded Int A, B, Base

Base = A

Base = Base / B

Base = Base * B

A = A - Base
```

By performing integer division on the operand A by the operand B, the remainder is removed. Then by multiplying back the operand B and subtracting this from the original A, the non-remainder part is removed and what is left is the remainder.

As before with division, if the inverse of the modulus, like the quotient, can be found prior to this operation, calculating the modulo will prove to be quick. This fact will prove useful for cryptographic algorithm implementations with exceedingly large primes. The inverse of these primes can be pre-calculated and stored throughout the life of the prime.

CONCLUSION

Though this class is not ready for implementation in any real use cases, it is useful for demonstrating and testing large numbers with cryptographic algorithms. Using precalculated inverses, algorithms can use massive primes, those in the thousands of bits, with striking speed.

First iterations of this code can be found on the following repository: https://github.com/codycook96/Unbounded-Int/

Developed by Cody Cook, 2018