# Cryptographic Engineering: Assignment 2 Prelab

In this assignment, you will implement efficient arithmetic operations which are necessary to implement some of the cryptographic protocols you will implement in the course.

## Revision

1. Given two 5-digit decimal numbers (00000-99999) $a$ and $b$, what is the maximum number of decimal digits can $a + b$ be? What about if they were 5-digit hex numbers (00000-FFFFF)?
2. Given one 5-digit binary number (00000-11111) $a$ and 3-digit binary number (000-111) $b$, what is the maximum number of binary digits can $a + b$ be?
3. In general, given any $x$-digit numbers $a$ and $b$ with digit size $s$, what is the maximum number of digits of size $s$ can $a + b$ be?
4. Given one 4-digit hex number $a$ and $b$, how many bits of storage space is needed to store $a$? $a+b$ ? $a - b$?
5. In general, given an $x$-bit number $a$ and a y-bit number $b$, how many bits of storage space is needed to store $a + b$? $a - b$ ?
6. When performing **addition** step-by-step using the schoolbook method, the carry you get at each step when adding two $x$-digit number of digit size $s$ cannot be more than?
7. When performing **subtraction** step-by-step using the schoolbook method, the borrow you get at each step when subtracting two $x$-digit number of digit size $s$ cannot be more than?
8. Given two 3-digit decimal numbers $a$ and $b$, what is the maximum number of decimal digits can $a \times b$ be? What about if they were 3-digit hex numbers ?
9. Given one 6-digit binary number $a$ and 4-digit binary number $b$, what is the maximum number of binary digits can $a \times b$ be?
10. In general, given any $x$-digit number $a$ and $y$-digit number $b$ with digit size $s$, what is the maximum number of digits of size $s$ can $a \times b$ be?
11. In general, given an $x$-bit numbers $a$ and a $y$-bit number $b$, how many bits of storage space is needed to store $a \times b$?
12. When performing multiplication step-by-step using the schoolbook method, the carry you get at each step when multiplying two $x$-digit number of digit size $s$ cannot be more than?

### Multi-precision Integers

In cryptography implementation, we are dealing with very large integers that cannot be allocated inside any default data types such as `int`, `unsigned long long` or `uint64_t`. Therefore, we need to allocate a big integer inside an array of C data-type. For instance, `uint64_t` data-type can allocate up to 64-bit length numbers. However, in order to allocate a 256-bit integer, we need to use an array of `uint64_t[4]`, i.e, $4 \times 64 = 256$. This is called **multi-precision integer**.

## Multi-precision arithmetic

When performing arithmetic operations, carry (for addition and multiplication) and borrow (for subtraction) might need to be propagated to the next array element. For example, in order to add two large multi-precision integers we perform:

res[0] = a[0] + b[0]

res[1] = a[1] + b[1] + carry overflow from res[0]

res[2] = a[2] + b[2] + carry overflow from res[1]

...

This is called **multi-precision arithmetic** and it is the most efficient way of handling big integer arithmetic.

## Radix-$2^n$ Representation

In cryptography implementation, we use different radix representations to allocate big integer inside memory. In the previous section, we allocate 256-bit integer inside four **64-bit** digits. That means each 64-bit element of array is occupied, and there is no room for carry or borrow propagation. This method is called **full radix** representation or radix-$2^{64}$ representation of numbers.

The main downside of full radix representation is that efficient arithmetic operations are **platform specific**. Some processors can handle full radix representation but this requires writing part of the code in assembly and it will only work on that processor's family. Some operating systems and compilers expose C libraries that will handle the full radix representation but it becomes dependent on the platform and might only work on a limited number of processors.

In order to get rid of all carry overflows, we can represent numbers in **smaller radix**. For instance, we can allocate each 256-bit integer inside 8×64-bit elements, and let each element be 32-bit occupied! In this case, we have an extra 32 bits memory left in each digit which ensure that no carry overflow will happen. Therefore, we consume more units of memory to eliminate all the carry propagation operations. This method give us radix-$2^{32}$ representation of numbers.

In the next exercise, we are going to work with relatively large integers (256 to 512 bits). The radix-$2^{32}$ representation can handle carry propagation of addition (and borrow propagation of subtraction) but not carry propagation of multiplication. Therefore, we store the integers inside the arrays of `uint64_t` data type (each element of array is 64-bits long) using **radix-$2^{16}$** representation. Only 16 bits of each array element is occupied and the extra 48 bits are reserved for carry overflows. Moreover, the result of each single-precision multiplication, e.g., a[0]×b[0] is only 32-bit (16-bit multiply by 16-bit gives 32-bit result).

## Subtraction

In C, whenever you decrement below 0, you loop back to the digit with all bits equal to 1. For example, in a 16-bit datatype, performing $0 - 1$ will give `0xFFFF`. The reverse is true. When you increment the largest number in a data-type, the output will be 0. For example, Adding 1 to the value `0xFFFF` will give 0 in a 16-bit data type. Subtraction (minuend - subtrahend = difference) is sometimes tricky to perform when the values are less than 0. There are two methods to compute the output of the subtraction by hand. For example, let's perform `0x1234-0x4321=0xFFFFCF13` in a 32-bit processor.

- Use the schoolbook subtraction and add `0x100000000` to the minuend if the minuend is smaller than the subtrahend. Below b indicates borrow while n indicates no borrow.

```
  b b b b b b n n
  1 0 0 0 0 1 2 3 4
-   0 0 0 0 4 3 2 1
----------
  0 F F F F C F 1 3
```

- Compute minuend + (-subtrahend). To compute -subtrahend, you perform 2's complement (invert all bits and add one). In hex, you would subtract each digit from F and then add 1. `-0x4321=0xFFFFBCDE+1=0xFFFFBCDF`. Below c indicates carry while n indicates no carry.

```
  n n n n c n c
  0 0 0 0 1 2 3 4
+ F F F F B C D F
---------
  F F F F C F 1 3
```

## Schoolbook and Comba Multiplications

Schoolbook and Comba multiplications are two methods to compute a multiplication. The main difference between the two methods is the order at which you get the output as shown in Figure 1. In schoolbook method, we work on one digit of the input at a time from smallest to largest ($a_0 \rightarrow a_1 \rightarrow a_2$). In Comba method, we work on one digit of the output at a time from smallest to largest ($r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow r_4 \rightarrow r_5$).

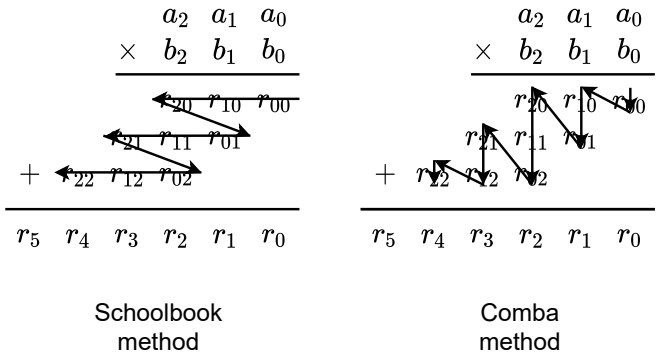


Figure 1: Schoolbook vs Comba method

## Karatsuba Multiplication

The one level Karatsuba multiplication allows you to compute the product of two large integers $a$ and $b$ using three multiplications of smaller numbers, each with about half as many digits as $a$ and $b$, plus some additions and digit shifts. Let's assume $a$ and $b$ are $n$ binary digits. Then $a$ can be written as $a_h 2^{n/2} + a_l$ where $a_l$ are the $n/2$ low bits of $a$ and $a_h$ are the $n/2$ high bits of $a$. Similarly, $b$ can be written as $b_h 2^{n/2} + b_l$. We then have

$$a \times b = (a_h 2^{n/2} + a_l)(b_h 2^{n/2} + b_l) = z_2 2^n + z_1 2^{n/2} + z_0$$

where

$$z_2 = a_h b_h$$

$$z_1 = a_h b_l + a_l b_h,$$

$$z_0 = a_l b_l$$

Karatsuba observed that one can compute $z_1$ using $z_2$ and $z_0$ as

$$z_1 = (a_h + a_l)(b_h + b_l) - z_2 - z_0$$

which leads to better performance since one large multiplication is substituted with additions and subtractions. After computing, $z_0$, $z_1$, and $z_2$, a simple addition can be performed as shown in Figure 2

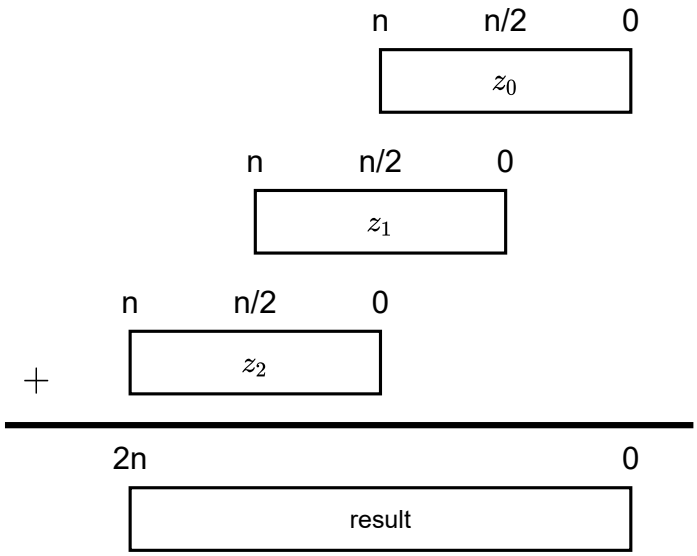


Figure 2: Karatsuba final step

## Constant-time Implementation

In cryptography, it is important to have a constant-time implementation of your algorithm. Otherwise, side channel attack (timing attacks, power-analysis attack, ...) can be performed. Usage of if statements without an else statement that performs a similar arithmetic operation may result in your code not performing at a constant time. Usage of a loop without having a constant number of iterations (usually used with while loop) can also result in your code not running at a constant time. In addition, usage of if statements can be targeted by fault injection attack (flipping your condition from false to true or vice-verse to perform the other statement). It is therefore discouraged to use if statements. It is recommended to stick to using a simple for loop aka `for(i=0;i<N;i++)`.

## SageMath Verification

It is very important in cryptography to be able to verify that your implementation produces a correct output. SageMath has the ability to write large integers directly. However, in C, you will often get your result in Radix-$2^n$ form. SageMath has the ability to convert the numbers into any Radix form.

If you have a very large integer $a$ and you would like to write it into radix $2^d$ and there are $s$ array elements, use the following code. The least significant digit is on the LEFT (reverse of how you write the digits).

```
a.digits(base=2^d,padto=s)
```

The above code will show the output in decimal. If you would like to show the output in hexadecimal with each digit showing $n$ hexadecimal digits, use the following. Note the additional code here is pure Python.

```
["0x{0:0{1}x}".format(x,n) for x in a.digits(base=2^d,padto=s)]
```

Assuming any of the above outputs is stored in `v`, to convert it back to one large integer, use the following. Note that if one of the digits is larger that the radix $2^d$, then SageMath will automatically propagate the excess value (carry) to the next digit.

```
Integer(v,base=2^d)
```

Additional SageMath examples: https://nbviewer.jupyter.org/urls/pastebin.com/raw/GJMCtHfA

# Prelab

For questions 1, 2, and 3, you are going to write a piece of code with very basic instruction set. The syntax is exactly like in C but

- Conditional statements (`if`), loops (`while`, `for`, ...), are NOT available
- Arrays are available but their syntax is $a_i$ instead of $a[i]$. Whenever $a$ $(a_3a_2a_1a_0)$ is written, it means $a$ is an array of four elements and the elements are $a_0, a_1, a_2, a_3$ with $a_0$ being the first element.
- Radix-$2^8$ representation is used to represent multi-precision integer with each element having 32 bit data type. For example, a 32-bit number $a$ $(a_3a_2a_1a_0)$ has 4 digits $a_0$, $a_1$, $a_2$, $a_3$ with each of these digits having 32-bits of storage and only the least significant 8-bit are used to represent the large integer (the remaining 24-bits are 0s by default).

1. **Addition**: In this exercise, you are going to add two 32-bit numbers $a$ $(a_3a_2a_1a_0)$ and $b$ $(b_3b_2b_1b_0)$ into the 32-bit number $r$ $(r_3r_2r_1r_0)$. Even though the output is supposed to be 33 bits, we are going to ignore the 33rd bit and assume it is an overflow (the first 32 bit are still outputted).
   (a) **Carry propagation**: When performing schoolbook addition, carry propagation from one digit to the next digit is required as shown in Figure 3. This can be seen as an add with carry function that takes as inputs the carry in and two addends and as outputs the sum and the carry out as shown in Figure 4. Write a code that implements the function `ADDC(carryIn, addend1, addend2, sum, carryOut)`. Remember that `addend1` and `addend2` are 32-bit data-type with only the lowest 8-bit filled; therefore, the `sum` must similarly be 32-bit datatype with the lowest 8-bit filled and the remaining 24-bit 0s at the end of the function. The `carryIn` and `carryOut` are 1 bit only.

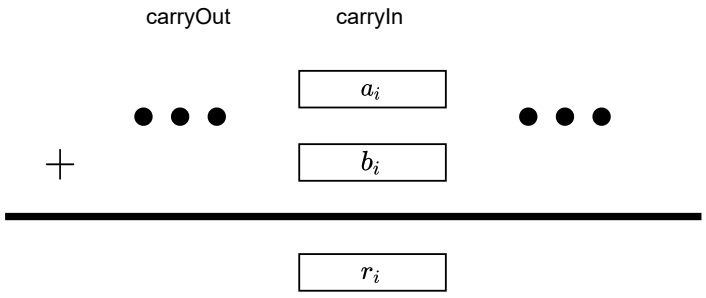


Figure 3: Carry propagation

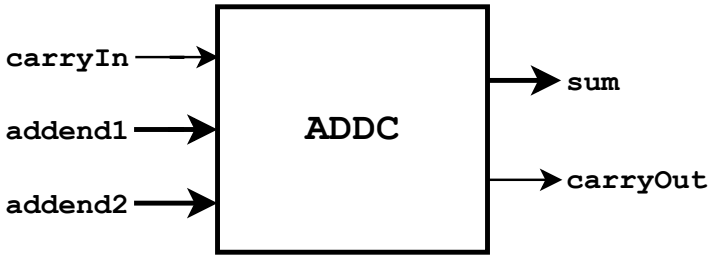


Figure 4: `ADDC` function

(b) Using the `ADDC` function created in the previous part, write the function `add32(r, a, b)` which performs $r = a + b$. You have access to one additional variable called `carry` which is 1 bit.

(c) We are going to test `a=0x5b02deb9` and `b=0x41fdec03`. Write the initial values of each array element of `a` and `b`. Then, in a table, write the output of the `carry` and each array element of `r` after each line in function `add64`. Use line 0 to show the output before the start of the first line and `-` if the variable is uninitialized. Write all values in hexadecimal.

| Line | $r_0$ | $r_1$ | $r_2$ | $r_3$ | `carry` |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| ... | | | | | |
| | | | | | |

(d) Verify in SageMath that your final output is correct. Show the output in hexadecimal in this form ($r_0$, $r_1$, $r_2$, $r_3$) with each digit showing 2 hex digits. Hint: use the code from the SageMath Verification section.

2. **Subtraction**: In this exercise, you are going to subtract two 32-bit numbers $a$ ($a_3a_2a_1a_0$) and $b$ ($b_3b_2b_1b_0$) into the 32-bit number $r$ ($r_3r_2r_1r_0$). Even though the output is supposed to be 33 bits to accommodate all values, we are going to ignore the 33rd bit and assume it is an overflow (the first 32 bit are still outputted). Read the subtraction section before continuing.

(a) **Borrow propagation**: When performing schoolbook subtraction, borrow propagation from one digit to the next digit is required as shown in Figure 5. This can be seen as a subtract with borrow function that takes as inputs the borrow in, minuend and subtrahend, and as outputs the difference and the borrow out as shown in Figure 6. The subtraction function is a bit tricky. When the minuend is smaller than the subtrahend, you will get the output similar to the subtraction section. However, when borrowing from the next digit, you are actually borrowing $2^8$ =`0x100` (aka the radix). Therefore, you will add `0x100` when the `minuend` is smaller than the `subtrahend+borrowIn`. There is no access to if statements which means the `difference` and `borrowOut` must be extracted using bit manipulation. To figure out how to extract the values, compute `0x12-0x21` which requires borrowing. Then add `0x100` to figure out what the difference should be. Compare the lowest 8 bits of both values. Assume radix-$2^8$ and all data types are 32 bits.

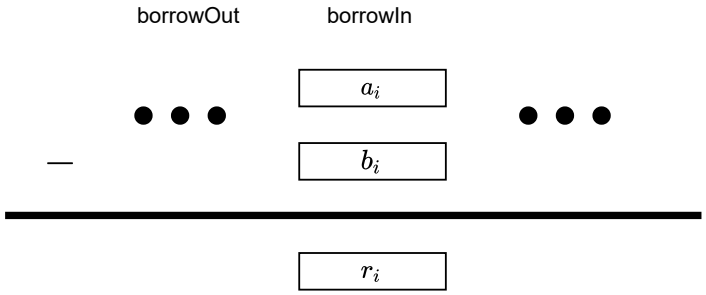


Figure 5: Borrow propagation

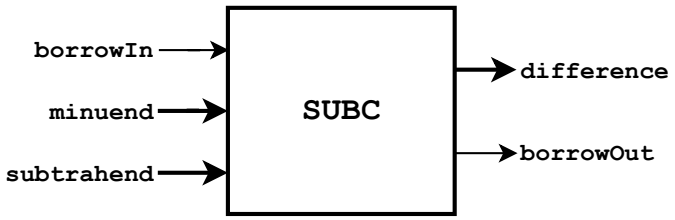


Figure 6: `SUBC` function

(b) Write a code that implements the function `SUBC(borrowIn, minuend, subtrahend, difference, borrowOut)`. Remember that `minuend` and `subtrahend` are 32-bit data-type with only the lowest 8-bit filled; therefore, the `difference` must similarly be 32-bit datatype with the lowest 8-bit filled and the remaining 24-bit 0s at the end of the function. The `borrowIn` and `borrowOut` are 1 bit only.

(c) Using the `SUBC` function created in the previous part, write the function `sub32(r, a, b)` which performs $r = a - b$. You have access to one additional variable called `borrow` which is 1 bit.

(d) We are going to test `a=0x5b02deb9` and `b=0x41fdec03`. In a table, write the output of the `borrow` and each array element of `r` after each line in function `sub64`. Use line 0 to show the output before the start of the first line and `-` if the variable is uninitialized. Write all values in hexadecimal.

| Line | $r_0$ | $r_1$ | $r_2$ | $r_3$ | `borrow` |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| ... | | | | | |
| | | | | | |

(e) Verify in SageMath that your final output is correct. Show the output in hexadecimal in this form ($r_0$, $r_1$, $r_2$, $r_3$) with each digit showing 2 hex digits. Hint: use the code from the SageMath Verification section.

3. **Multiplication**: In this exercise, you are going to multiply two 32-bit numbers $a$ ($a_3a_2a_1a_0$) and $b$ ($b_3b_2b_1b_0$) into the 64-bit number $r$ ($r_7r_6r_5r_4r_3r_2r_1r_0$) using 3 different methods. We are NOT going to propagate the carry at each intermediate step of the multiplication because we can perform it once at the very end. However, we are not going to do this either because we can do it when computing the reduction in assignment 3. So, we are going to hold the accumulated carry in the 24-bits of free memory in each digit for this assignment (radix-$2^8$ in 32-bit datatype). Read the Schoolbook, Comba, and Karatsuba sections before continuing.

(a) **Schoolbook:** Write a code that implements the function `schoolbook_mul64(r, a, b)` which performs $r = a \times b$ using the schoolbook method. No additional variables are given. Do not propagate any accumulated carry bits.

(b) **Comba Method:** Write a code that implements the function `comba_mul64(r, a, b)` which performs $r = a \times b$ using the Comba method. No additional variables are given. Do not propagate any accumulated carry bits.

(c) **Karatsuba Method:** Write a code that implements the function `karatsuba_mul64(r, a, b)` which performs $r = a \times b$ using the Karatsuba method. You have access to the

following two 32-bit element array $ma$ and $mb$ and four 32-bit element array $z0$, $z1$, $z2$. A code template is given to you below. The smaller multiplications can be computed using schoolbook method or Comba method. Do not propagate any accumulated carry bits including the addition (and borrow bits in subtraction).

```
//Compute ma = a_h + a_l
...
//Compute mb = b_h + b_l
...
// Compute z0 = a_l * b_l
...
// Compute z1 = ma * mb
...
// Compute z2 = a_h * b_h
...
// Compute z1 = z1 - z0 - z2
...
// Compute r
...
```

(d) We are going to test `a=0x5b02deb9` and `b=0x41fdec03`. Write a table for each multiplication function (previous 3 questions) showing the output of each variable after each line (For Karatsuba create one table for each section and only show the variables that update). Use line 0 to show the output before the start of the first line and `-` if the variable is uninitialized. Write all values in hexadecimal. Hint: All three methods should give the same answer.

(e) Verify in SageMath that your final output is correct. Show the output in hexadecimal in this form ($r_0$, $r_1$, $r_2$, $r_3$, $r_4$, $r_5$, $r_6$, $r_7$) with each digit showing 2 hex digits. For this question, you will need to propagate the carry in the previous section to get the same result. SageMath will automatically propagate it for you. Check the SageMath verification section.

4. **Questions.** Assume the cost of 32-bit addition/subtraction is A and cost of 32-bit multiplication of M. Cost of assignment and initializing variables is assumed 0 (negligible).

   (a) Count the number of additions and multiplications for Schoolbook method, Comba method, and Karatsuba method.

   (b) If M = A, then which method is the fastest. Show the cost for each method in terms of A.

   (c) If M = 4×A, then which method is the fastest. Show the cost for each method in terms of A.

   (d) Assuming carry propagation is required (because of using full-radix for example), name two advantages of Comba method over Schoolbook method.