COS 217: Introduction to Programming Systems

Assignment 5:
Assembly Language Programming and Testing

PRINCETON UNIVERSITY

Assignment 5 Goals

Apply your knowledge of ARMv8 assembly language!

1. Emulate the compiler: translate C to assembly language
   - Also: practice testing!
2. Beat the compiler: re-implement one critical function to run as quickly as possible

PART 1

The wc command

Consider a file named proverb containing the following text:

```
Learning is a
treasure which
accompanies its
owner everywhere.
-- Chinese proverb
```

@danieltuttle

Then running `wc < proverb` prints the number of lines, words, and characters:

```
5       12      82
```

Our implementation: `mywc.c`

```
while ((iChar = getchar()) != EOF) {
   lCharCount++;
   if (isspace(iChar)) {
      if (iInWord) {
         lWordCount++;
         iInWord = FALSE;
      }
   } else {
      if (! iInWord)
         iInWord = TRUE;
   }
   if (iChar == '\n')
      lLineCount++;
}
if (iInWord)
   lWordCount++;
printf("%7ld %7ld %7ld\n", lLineCount, lWordCount, lCharCount);
```

Part 1a Task

Translate `mywc.c` into `mywc.s`

- Generate flattened C code (using conventions in Lecture 18) and include as comments!
- Use exactly the same algorithm/logic – don't simplify or optimize
- Use the same 5 `static` variables
- Still call `getchar`, `isspace`, and `printf`
- Don't use the output from `gcc217`
  (it's convoluted, and it's against the rules)

Part 1b Task

Compose data files (called `mywc*.txt`) that perform the following (see lecture 9):

- boundary tests ("corner cases")
- statement tests (exercise every line of code)
- stress tests (but don't get too wild – not too big, and only ASCII)

Some hints:

- Pretend you're us: design test cases to expose what's wrong
- Write a program that uses `rand()` to generate random characters
- Programatically generated data can also help with boundary tests (which might be hard to generate with an editor)

PART 2

Motivation

Secure communication is enabled by *cryptography*, which is based on the conjectured difficulty of solving certain problems involving big numbers.

Example: discrete logarithm

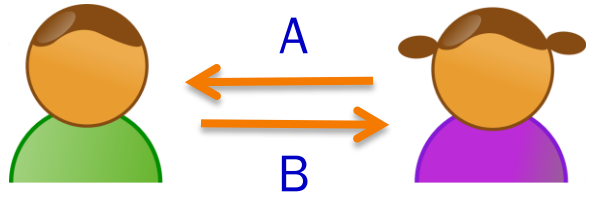
Let $A = g^a \bmod p$

It is believed to be hard to find a given A, g, and p.
(This might change with quantum computers…)

Diffie-Hellman Key Exchange

Suppose that Alice creates a secret a and sends $A = g^a \bmod p$ to Bob.

Then Bob creates a secret b and sends $B = g^b \bmod p$ to Alice.



Alice computes $B^a \bmod p = g^{ba} \bmod p$, and Bob computes $A^b \bmod p = g^{ab} \bmod p$

- Alice and Bob now share the same secret number! (To be used e.g. as an encryption key.)
- Any eavesdropper knowing A, B, g, and p can't efficiently compute the secret.

But, to make trial-and-error attacks hard, these computations need numbers much bigger than 32 bits (`int`) or 64 bits (`long`).

Multiple Precision Arithmetic or "Bignum" Libraries

Emulate arithmetic on quantities bigger than a machine word

Do operations "by hand", except operating on bigger chunks than single digits

- In fact, each "digit" is a machine word – 64 bits in our case
- When adding two "digits", they both range not from 0 to 9, but from 0 to 18 quintillion (-ish)

Example: the GMP library (`gmplib.org`)

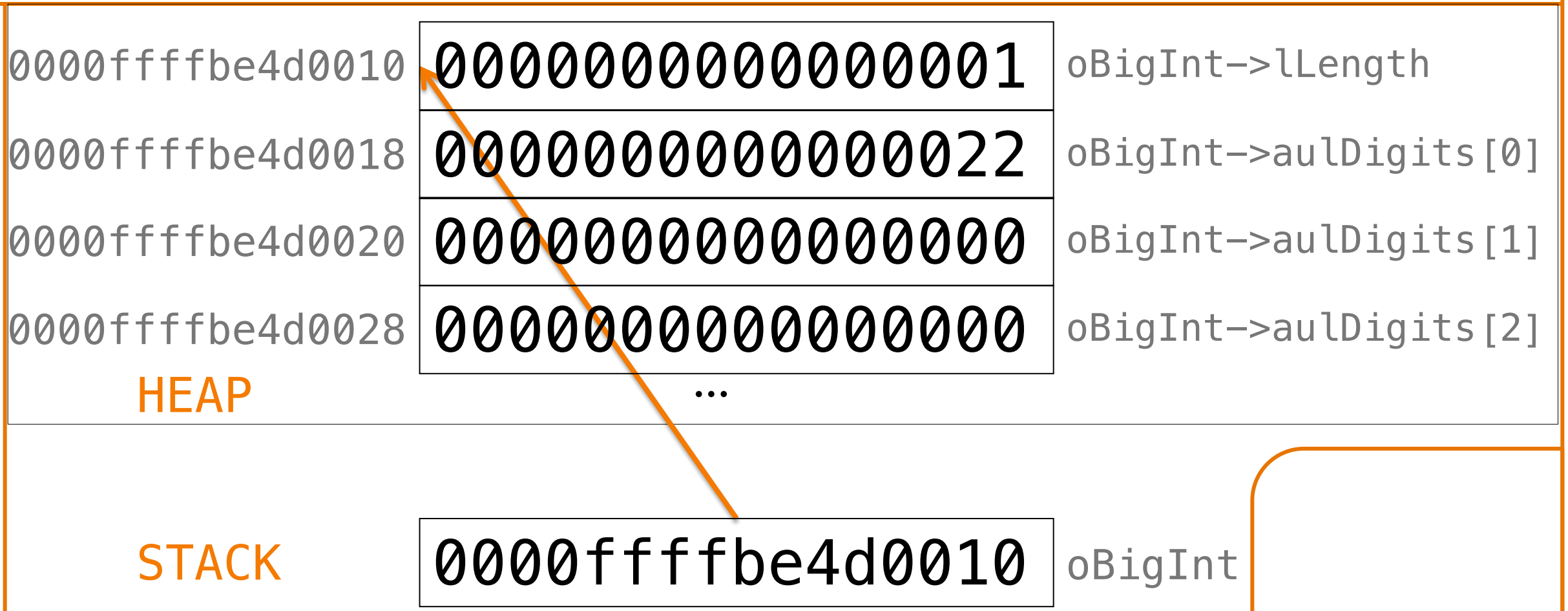
Our simplified version: `BigInt`

- Limited to 32768 64-bit words
- No negative numbers
- Only implemented operation: +
- Can't quite do Diffie-Hellman, but our client computes reallyreally large Fibonacci numbers (which grow exponentially)

BigInt Objects

```
enum {MAX_DIGITS = 32768};

struct BigInt
{
   /* The number of used digits in the BigInt object. The integer 0
      has length 0. This field could be of type int, but then the
      compiler would place padding between this field and the next. */
   long lLength;

   /* The digits comprising the BigInt object. aulDigits[0] stores the
      least significant digit.  The unused digits are set to 0. */
   unsigned long aulDigits[MAX_DIGITS];
};

typedef struct BigInt *BigInt_T;
```

BigInt Objects

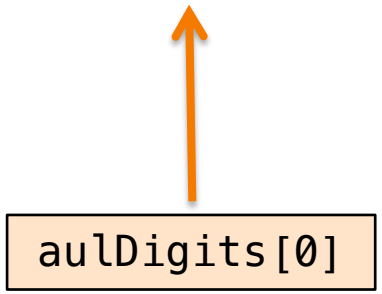


BigInt_add

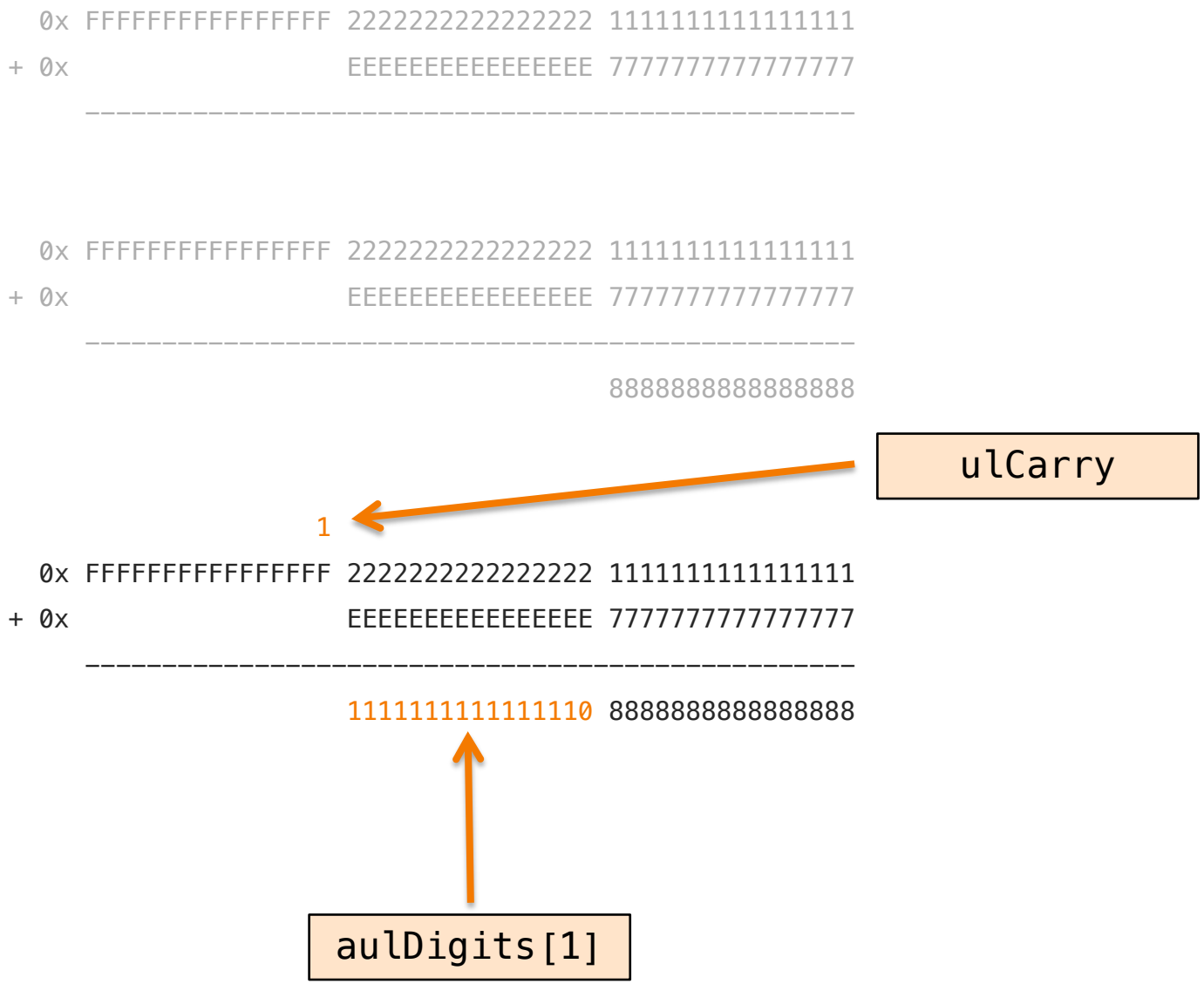
```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
     ------------------------------------------------
```

BigInt_add

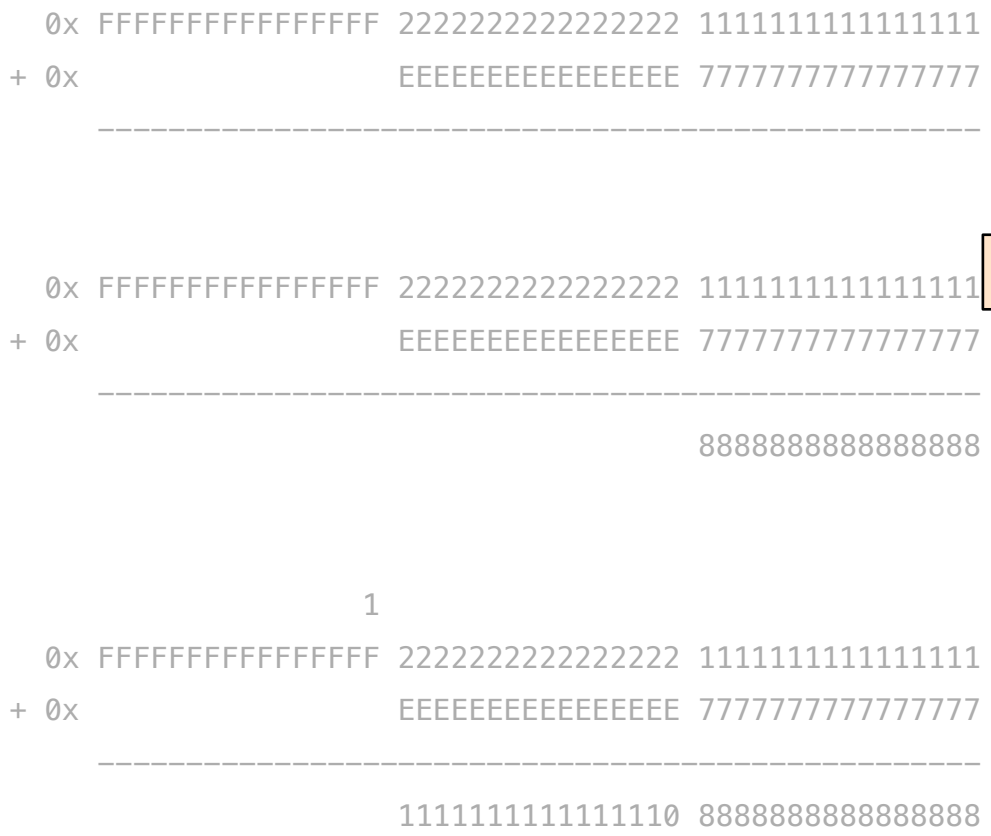
```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
     ------------------------------------------------

  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
     ------------------------------------------------
                                       8888888888888888
```

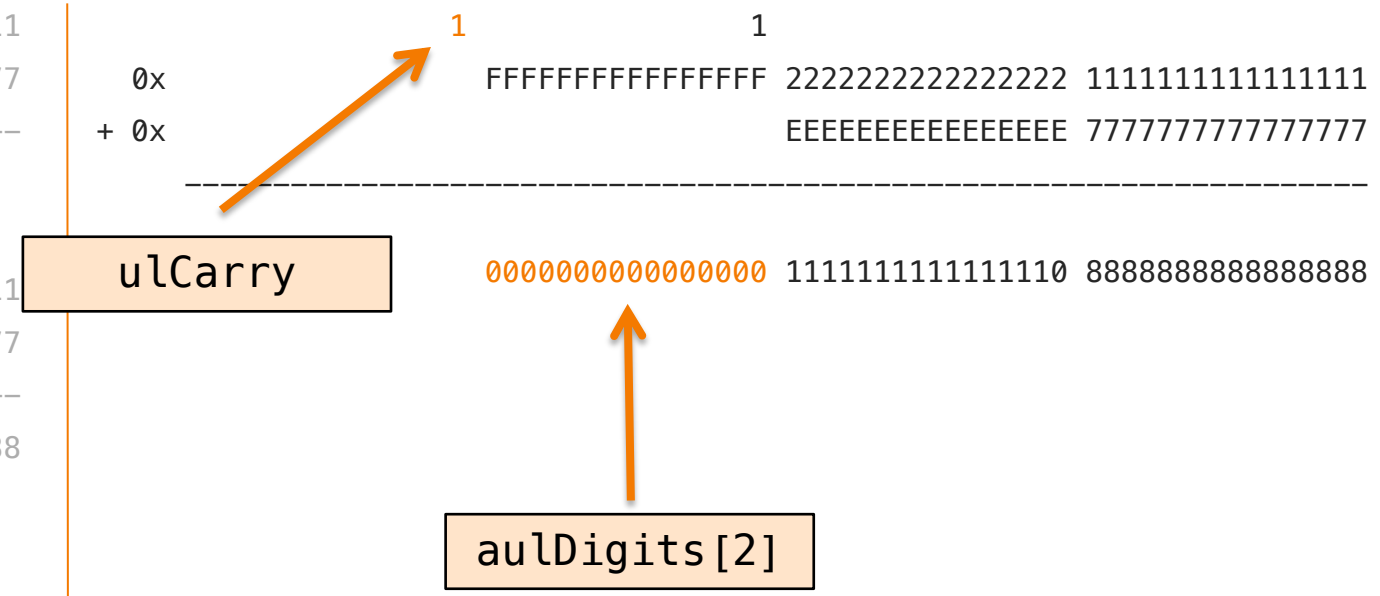


BigInt_add


0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
8888888888888888
ulCarry
1
0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
1111111111111110 8888888888888888
aulDigits[1]

BigInt_add


0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
8888888888888888
1
0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
1111111111111110 8888888888888888


1 1
0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x EEEEEEEEEEEEEEEE 7777777777777777
ulCarry
0000000000000000 1111111111111110 8888888888888888
aulDigits[2]

BigInt_add

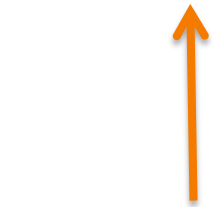
```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
   ------------------------------------------------------
```

```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
   ------------------------------------------------------
                                       8888888888888888
```

```
                    1
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
   ------------------------------------------------------
                     1111111111111110 8888888888888888
```

```
                  1                  1
  0x                FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                                 EEEEEEEEEEEEEEEE 7777777777777777
   ------------------------------------------------------------------------
                     0000000000000000 1111111111111110 8888888888888888
```

```
                  1                  1
  0x                FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                                 EEEEEEEEEEEEEEEE 7777777777777777
   ------------------------------------------------------------------------
   0000000000000001 0000000000000000 1111111111111110 8888888888888888
```



aulDigits[3]

lLength = 4;

Part 2a: Unoptimized C `BigInt_add` Implementation

Study the given code.

Then build a `fib` program consisting of the files `fib.c`, `bigint.c`, and `bigintadd.c`, *without* the `-D NDEBUG` or `-O` options.

Run the program to compute `fib(250000)`.
In your readme file note the amount of CPU time consumed.

Part 2b/c: Optimized C `BigInt_add` Implementation

Then build a `fib` program consisting of the files `fib.c`, `bigint.c`, and `bigintadd.c`, *with* the `-D NDEBUG` and `-O` options.

Run the program to compute `fib(250000)`.
In your readme file note the amount of CPU time consumed.

Profile the code with `gprof`. (More on this next lecture.)

Part 2d/e/f: Implement in Assembly Language

Suppose, not surprisingly, your `gprof` analysis shows that most CPU time is spent executing the `BigInt_add` function. In an attempt to gain speed, you decide to code the `BigInt_add` function manually in assembly language…

- Callable from C code!
- Most realistic way of using assembly: you usually won't write entire programs…
- Common to see highly-optimized "kernel" libraries for cryptography, image/video processing, compression, scientific computing, etc.

- Your task: write correct, optimized code, and eventually beat the compiler!

Part 2d: Translate to Assembly Language

Straightforward translation, as in part 1

- Translate both the `BigInt_larger` and `BigInt_add` functions
- Use exactly the same algorithm/logic – don't simplify or optimize
- Use the same local variables, stored in memory (on the stack)
- Test by comparing against `bigintadd.c` using `diff`

Part 2e: Optimize to use registers, not the stack

Straightforward translation won't beat the compiler. :-(

So, modify your assembly language code to use
callee-saved registers instead of memory
for all parameters and local variables.

Part 2f (Challenge Portion): Optimize All You Want

- Use the *guarded loop* pattern (Pyeatt/Ughetta Ch. 5, Sec. 3.2)
- *Inline* the call of the `BigInt_larger` function
- Use the `adcs` ("add with carry and set condition flags") instruction
- Feel free to implement any additional optimizations!

Beating the compiler's best version by 2X is realistic!

But, this part is challenging. Don't let it consume all your time.
We will not think unkindly of you if you decide not to do it.

Reminder: this is a *partnered* assignment.