COS 217: Introduction to Programming Systems

Assignment 5: Assembly Language Programming, Testing, and Debugging

PRINCETON UNIVERSITY

Assignment 5 Goals

Apply your knowledge of AARCH64 assembly language!

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

Also, practice testing and debugging!

PART 1

The wc command

Consider a file named proverb containing the following text:

```
Learning is an
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 seen in lecture)
- Use the flattened C as comments in `mywc.s`
- Use exactly the same algorithm/logic/memory interaction: 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)
- Make the code readable, with liberal use of `.equ`

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 a subset of ASCII)

Explain how your tests match up with your code

Some hints:

- Pretend you're us: design test cases to expose what's wrong
- Write a program that uses `rand()` to generate random characters
- Programmatically generate boundary tests (which might be hard with an editor)
- Complete Part 1 in the first week of the assignment period (i.e., by W 11/20)

PART 2

Bignum Motivation

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

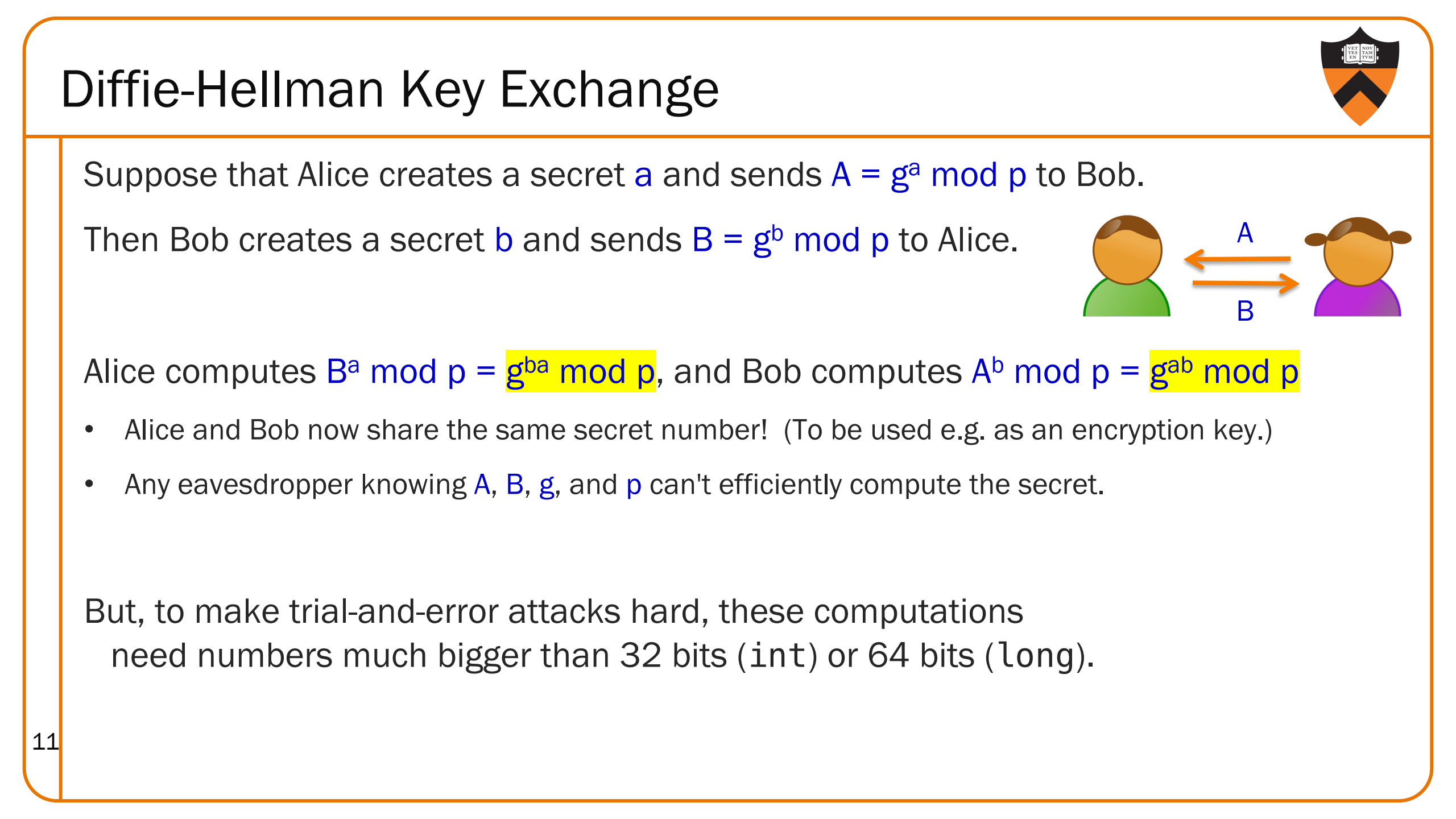
Example: discrete logarithm

Let $A = g^a \bmod p$

It is believed to be Hard to find a given A, g, and p.

(This might or might not 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 18.4 quintillion (-ish)

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

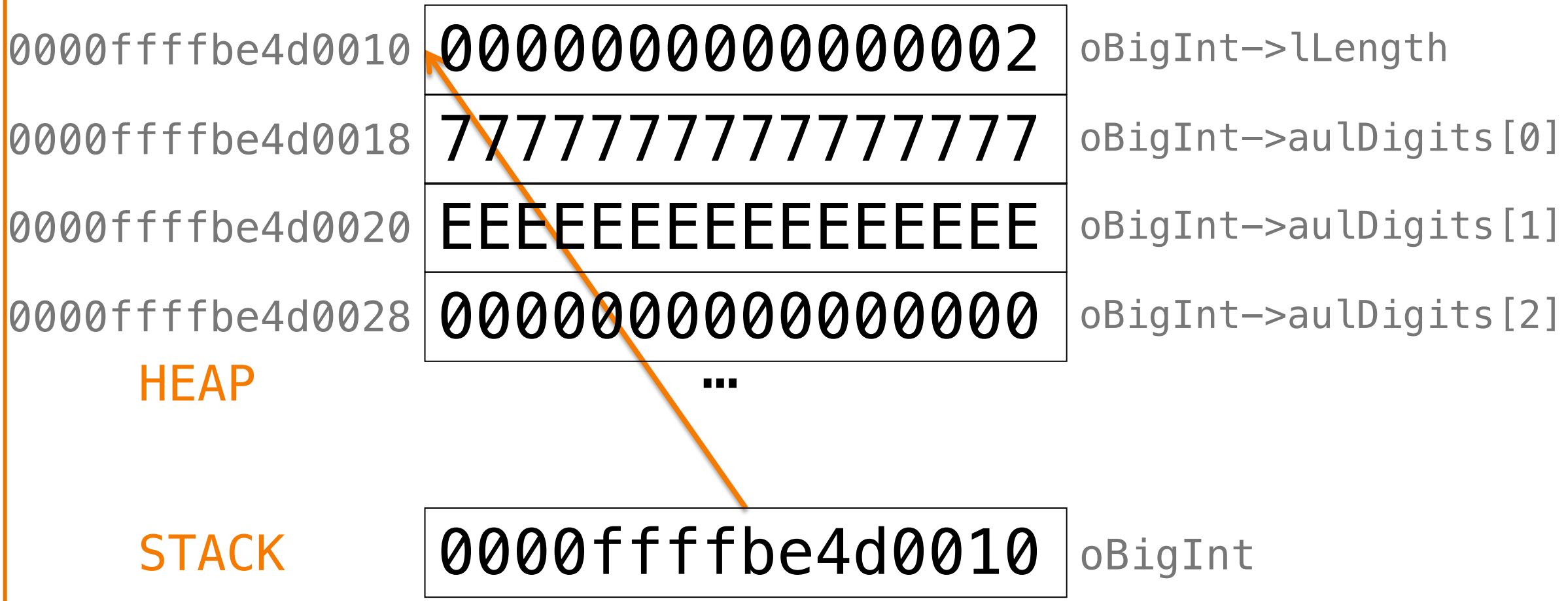
Our simplified version: `BigInt`

- "Limited" to 32768 64-bit words
- No negative numbers
- Only implemented operation: +
- Can't quite do Diffie-Hellman key exchange, but our client computes reallyreallyreally 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

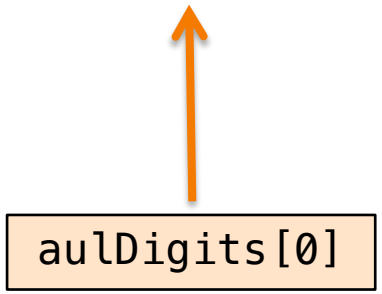


BigInt_add

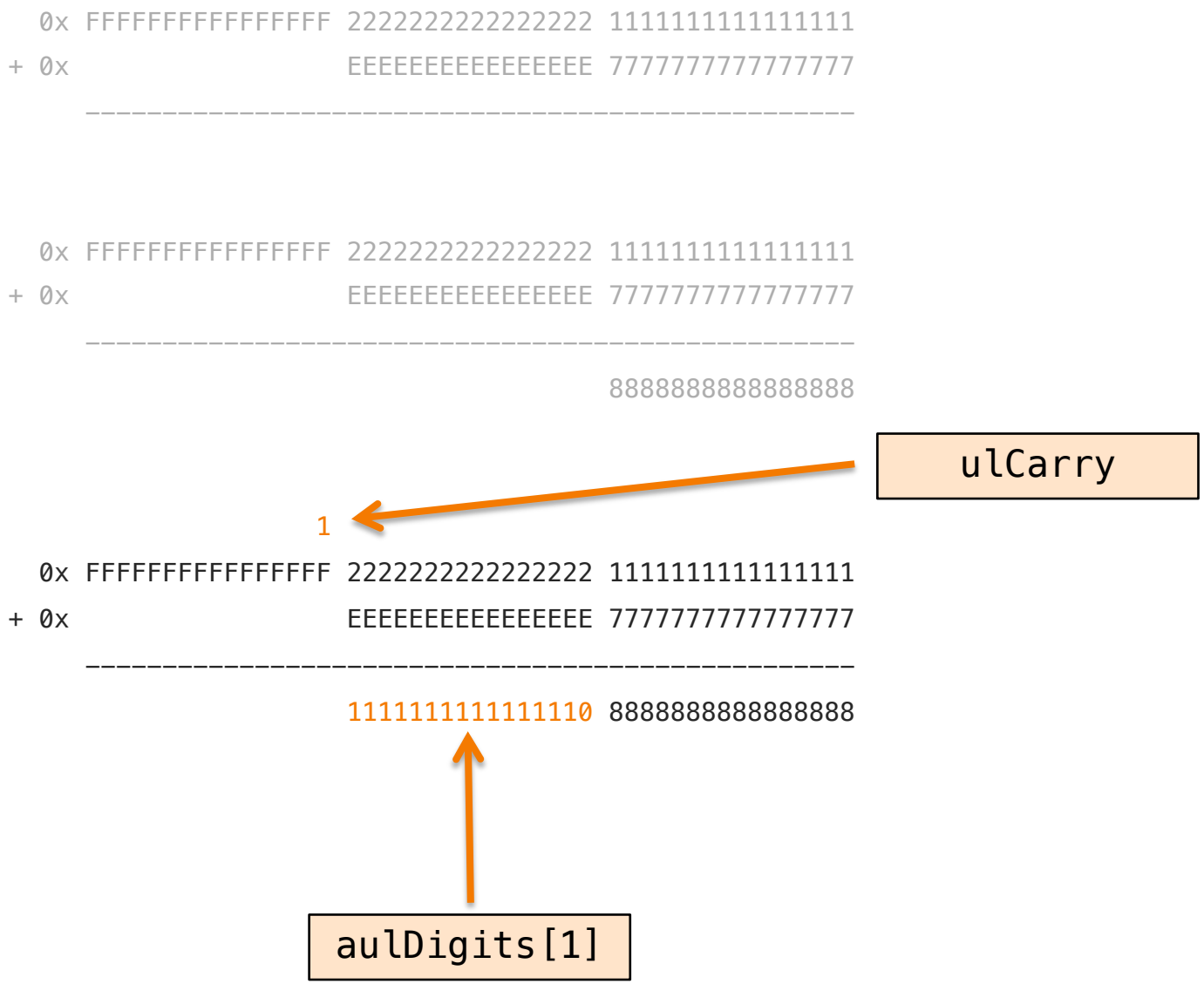
```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
     ------------------------------------------------
```

BigInt_add

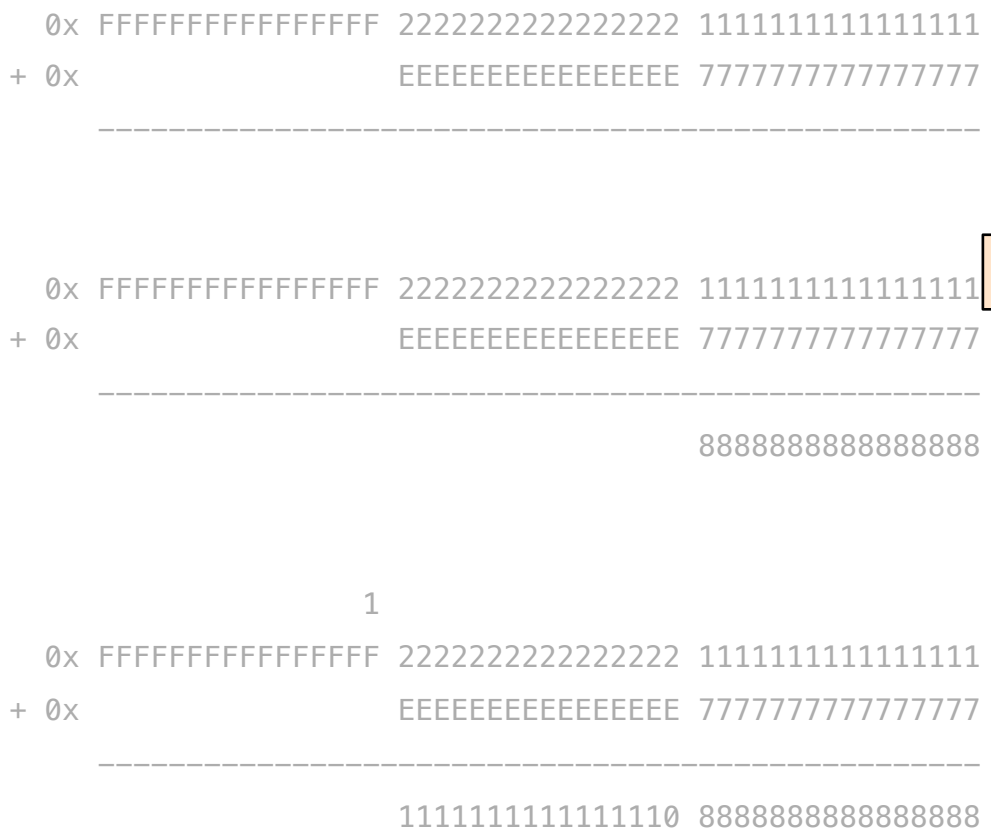
```
  0x FFFFFFFFFFFFFFFF 2222222222222222 1111111111111111
+ 0x                  EEEEEEEEEEEEEEEE 7777777777777777
     ------------------------------------------------


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

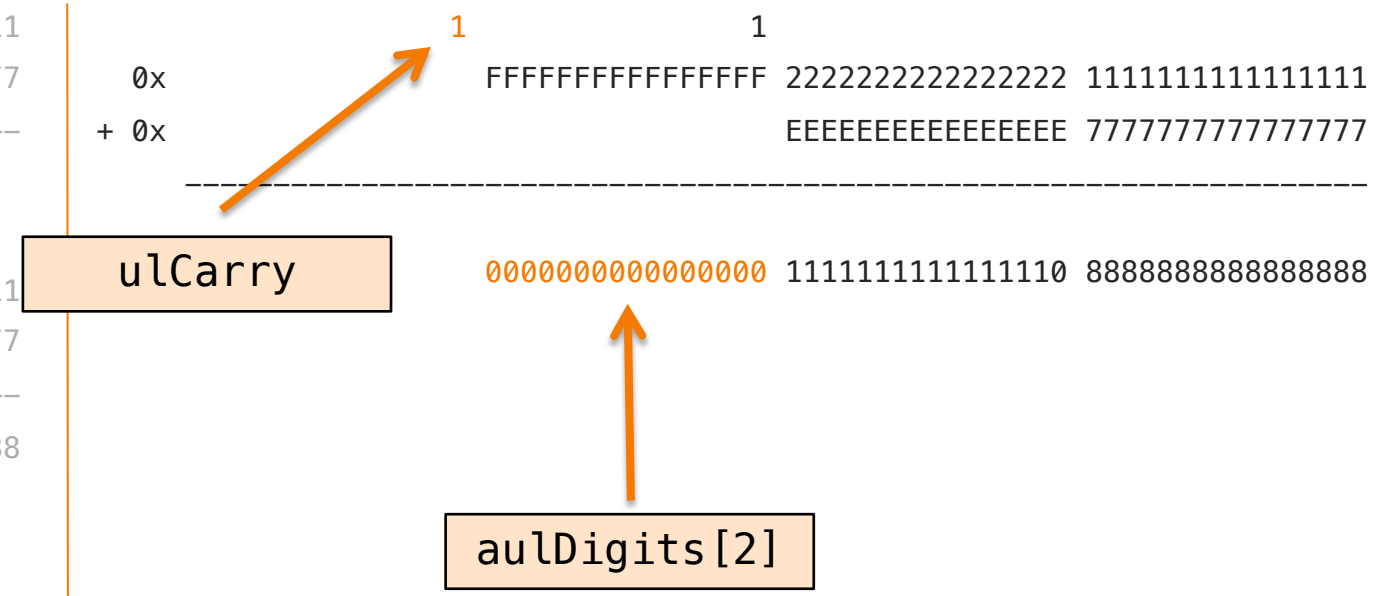


BigInt_add


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 in an upcoming 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

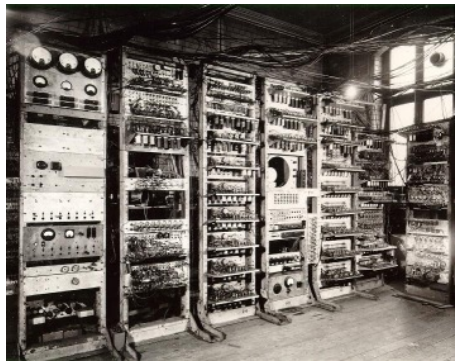
- Translate both the `BigInt_larger` and `BigInt_add` functions
- Use exactly the same algorithm/logic – don't simplify or optimize
- Use the same local variables, stored in memory (on the stack)
- Make the code readable, with liberal use of `.equ`
- Test by comparing output against `bigintadd.c` using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

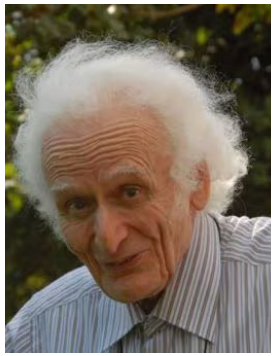
Part 2e: Optimize to use registers, not the stack

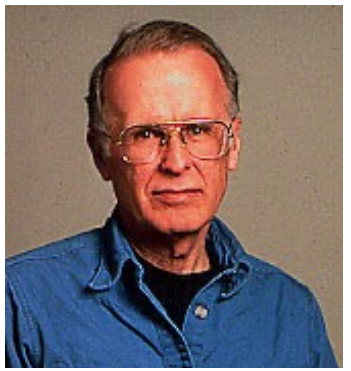
Straightforward translation won't beat the compiler. :-(

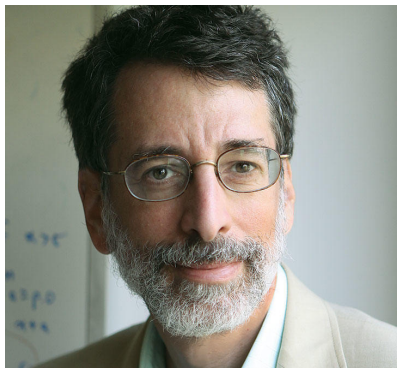
So, modify your assembly language code to use
callee-saved registers instead of memory
for all parameters and local variables (see slides from a previous lecture).

This should get you *close* … but probably *still* won't beat the compiler.
(Darn you 70+ years of compilers research!)









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

Start with the following optimizations:

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

Start with the following optimizations:

- Use the *guarded loop* pattern (Pyeatt/Ughetta Ch. 5, Sec. 3.2)

Original C

```
while (expr) {
   statement1;
   ...
   statementN;
}
```

Basic Flattened C

```
loop1:
   if (! expr) goto endloop1;
   statement1;
   ...
   statementN;
   goto loop1;
endloop1:
```

Guarded Loop Pattern

```
if (! expr) goto endloop1;
loop1:
   statement1;
   ...
   statementN;
   if (expr) goto loop1;
endloop1:
```

Pro: 1 fewer instruction per iteration of the loop

Con: Harder to maintain duplicated code (to compute and test !expr and expr)

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

Start with the following optimizations:

- Use the *guarded loop* pattern (Pyeatt/Ughetta Ch. 5, Sec. 3.2)
- *Inline* the call of the `BigInt_larger` function

Effectively: replace function calls with the function body of the callee

Pro: Fewer instructions executed: no bl, no prologue, no epilogue, no ret
Con: Harder to read/maintain less modular code

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

Start with the following optimizations:

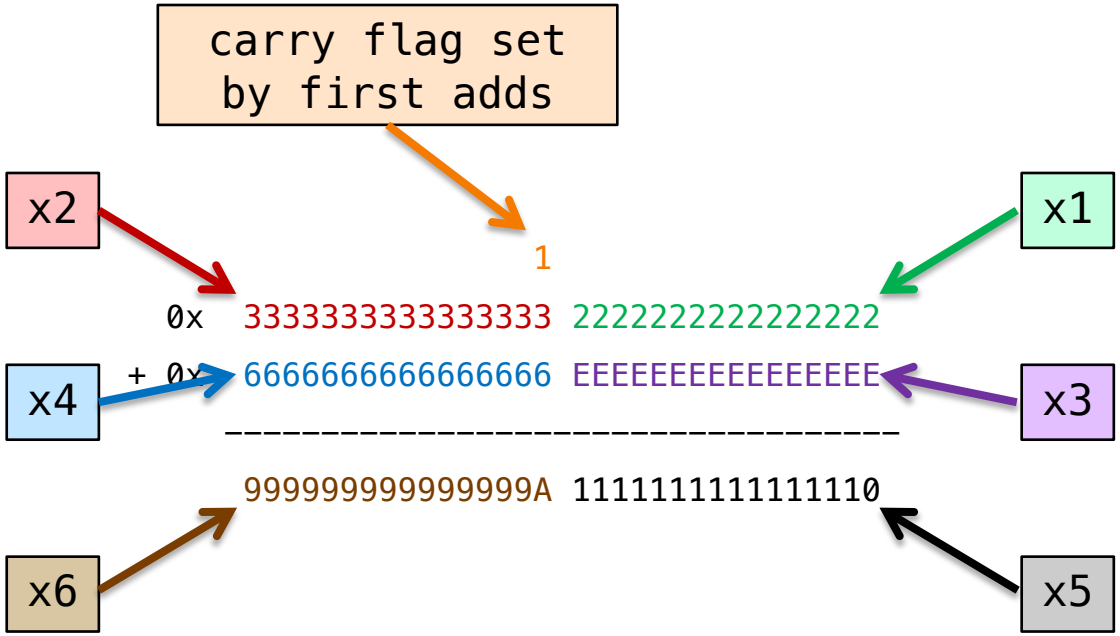
- Use the *guarded loop* pattern (Pyeatt/Ughetta Ch. 5, Sec. 3.2)
- *Inline* the call of the `BigInt_larger` function
- Use the `adcs` ("add with carry and set condition flags") instruction

```
                          1
    0x  3333333333333333 2222222222222222
+   0x  6666666666666666 EEEEEEEEEEEEEEEE
    ------------------------------------
    0x  999999999999999A 1111111111111110
```

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

Start with the following optimizations:

- 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




```
adds x5, x1, x3
adcs x6, x2, x4
```

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

Start with the following optimizations:

- 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

Then feel free to implement any additional optimizations!

Equaling/beating the compiler is totally realistic!

But this part is challenging. Don't let it consume your life. Don't fail your other classes.

We will not think unkindly of you if you decide not to push too hard on it.

Reminder: this is a *partnered* assignment. Please make the effort to find a partner!

IN A4 I FINALLY GOT GOOD AT DEBUGGING …
DO I HAVE TO RE-LEARN GDB FOR ASSEMBLY?

Debugging Assembly Language with GDB

Most of the `gdb` commands you already know can be used with assembly language!

- `run`, `break`, `backtrace`, `frame`, `step`, `next`, `continue`, `list`, `print`, `display`, `x`, `watch`, etc.
- Major difference: we'll primarily care about *contents of registers* and *memory pointed to by registers*
- Let's compare…

GDB: C vs. Assembly Language – Preparation

C

- Build with the `-g` flag:

```
gcc217 -g -c myfile.c -o myfile.o
```

ARM Assembly

- Add `.size` directive to the end of every function:

```
    .global myfunc
myfunc:
    ...
    ret
    .size myfunc, (. - myfunc)
```

- Then build with `-g` flag:

```
gcc217 -g -c myfile.s -o myfile.o
```

GDB: C vs. Assembly Language – Running

C

- From emacs:

```
Meta-x gdb / Esc-x gdb
```

- Or from command line:

```
$ gdb myprog
```

- And then start the program:

```
(gdb) run [arguments]
```

ARM Assembly

- Exactly the same

GDB: C vs. Assembly Language – Where Am I?

C

- From command-line:

```
(gdb) where (or backtrace or bt)

(gdb) list (or l)
```

- In emacs (or TUI mode): code and current location displayed in split-screen

ARM Assembly

- Exactly the same

GDB: C vs. Assembly Language – Printing Variables

C

- Print contents of variable i:

```
(gdb) print i (or p)
```

- Prints using format appropriate to type of i. Can override format to hex, decimal, character, etc.:

```
(gdb) p/x i
(gdb) p/d i
(gdb) p/c i
```

ARM Assembly

- Print contents of register x1:

```
(gdb) print $x1
```

- Can override format:

```
(gdb) p/x $sp
(gdb) p/d $x1
(gdb) p/c $w2
```

- Print contents of all registers:

```
(gdb) info registers (or i r)
```

GDB: C vs. Assembly Language – Pointers

C

- Dereference pi and print value:

```
(gdb) p *pi
(gdb) x pi
```

ARM Assembly

- Dereference sp+8 and print value:

```
(gdb) p *(int *)($sp+8)
(gdb) x $sp+8
```

- Override data size and format:

```
(gdb) x/bx $sp (byte in hex)
(gdb) x/h $x29 (16-bit halfword)
(gdb) x/wd $x1 (32-bit word in dec)
(gdb) x/g $x10 (64-bit giantword)
(gdb) x/i $pc  (instruction)
```

GDB: C vs. Assembly Language – Breakpoints

C

- Set breakpoint:

```
(gdb) break foo.c:37 (or b)
(gdb) b 42 (current file)
(gdb) b 59 if j > 17
(gdb) watch i (break if i changes)
```

- Step to next line of code:

```
(gdb) step (or s)
(gdb) next (or n – step over
            function calls)
```

- Resume execution:

```
(gdb) continue (or c)
(gdb) c 7 (skip next 7 breakpoints)
```

ARM Assembly

- Set breakpoint:

```
(gdb) break foo.s:37
(gdb) b 59 if $w2 > 17
(gdb) watch $x1
```

- Step to next instruction:

```
(gdb) stepi (or si)
(gdb) nexti (or ni)
```

- Resume execution:

```
(gdb) continue
(gdb) c 7
```

GDB: C vs. Assembly Language – Auto-Display

C

- Print contents of variable i every time gdb resumes control:

```
(gdb) display i (or disp)
```

- Prints using format appropriate to type of i. Can override format to hex, decimal, character, etc.:

```
(gdb) disp/x i
(gdb) disp/d i
(gdb) disp/c i
```

ARM Assembly

- Auto-display contents of register x1:

```
(gdb) display $x1
```

- Must use cast/dereference syntax to auto-display memory contents:

```
(gdb) disp *(unsigned *)($sp+8)
(gdb) disp/x *(long *)($x1+8*$x2)
```

Debugging Assembly Language with GDB

Simple tutorial provided this week as a precept handout: take the time to work it!

Probably worth your time to learn to use advanced features.
Especially conditional breakpoints, watchpoints, and displays!

For a full assembly debugging session, watch Lecture 20B from Fall 2020
(posted on course schedule page)

... as a bonus, it also gives a live walkthrough of iterative optimization similar to what you'll be moving through in bigintadd.s → bigintaddopt.s → bigintaddoptopt.s