Motivation

Suppose you are developing the assembly language BigInt_add() function. Further suppose that the function assembles and links cleanly, but executes incorrectly. How can you use GDB to debug the function?

The BigInt_add() function is somewhat difficult to debug because it uses the stack, structures, and arrays. This is an appropriate sequence...

Building for GDB

To prepare to use GDB, build your program as usual.

```
% gcc217 fib.c bigint.c bigintadd.s -o fib
```

Running GDB

Run GDB from within Emacs.

```
% emacs
<Esc key> x gdb <Enter key> fib <Enter key>
```

Setting Breakpoints

Set breakpoints at appropriate places. Breakpoints at the beginning of the main() and BigInt_add() functions would be appropriate.

```
(gdb) break main
(gdb) break BigInt_add
```
**Running Your Program**

Run the program, specifying some command-line argument.

```gdb
run 500000
```

Continue past the breakpoint at the beginning of the main() function.

```gdb
continue
```

Execution is paused after the two-instruction prolog of the first call of the BigInt_add() function. Issue the “continue” command nine more times. At this point the BigInt_add() function is being called to add the numbers 55 and 34.

**Examining Memory**

Use the print command to determine the contents of the EBP register:

```gdb
print/a $ebp
0xff9f2158
```

Thus you know the address of the base of the current stack frame. (That address might be different each time you run the program.) Now use the x command repeatedly to examine the function’s parameters as they exist in the stack and the heap.

Examine the function’s stack frame, interpreting each value as an address:

```gdb
x/a 0xff9f2158
0xff9f2158: 0xff9f21a8
(gdb) x/a 0xff9f215c
0xff9f215c: 0x80486b9 <main+357>
(gdb) x/a 0xff9f2160
0xff9f2160: 0xf7fb0008
(gdb) x/a 0xff9f2164
0xff9f2164: 0xf7fd1008
(gdb) x/a 0xff9f2168
0xff9f2168: 0xf7f84008
```
Examine the heap, interpreting each value as a decimal integer:

```
(gdb) x/d 0xf7fb0008
0xf7fb0008:     1
(gdb) x/d 0xf7fb000c
0xf7fb000c:     55
(gdb) x/d 0xf7fb0010
0xf7fb0010:     0
(gdb) x/d 0xf7fb0014
0xf7fb0014:     0
(gdb) x/d 0xf7fb0018
0xf7fb0018:     0

(gdb) x/d 0xf7fd1008
0xf7fd1008:     1
(gdb) x/d 0xf7fd100c
0xf7fd100c:     34
(gdb) x/d 0xf7fd1010
0xf7fd1010:     0
(gdb) x/d 0xf7fd1014
0xf7fd1014:     0
(gdb) x/d 0xf7ff1004
0xf7ff1004:     0

(gdb) x/d 0xf7f84008
0xf7f84008:     1
(gdb) x/d 0xf7f8400c
0xf7f8400c:     21
(gdb) x/d 0xf7f84010
0xf7f84010:     0
(gdb) x/d 0xf7f84014
0xf7f84014:     0
(gdb) x/d 0xf7fa4004
0xf7fa4004:     0
```

(That's 37 in hexadecimal)

(That's 22 in hexadecimal)

(That's 15 in hexadecimal)

As you traverse memory, draw a map of it as shown on the next page.
Suppose oAddend1 = 55, oAddend2 = 34, and oSum = 21
Using the Memory Map

Such a memory map can help with debugging. Moreover, such a memory map can help with writing assembly language code in the first place. Indeed if you did not have such a memory map, you probably would find it helpful/necessary to create one using pretend memory addresses before writing your assembly language code.

For example, suppose you must write assembly language code to access oAddend2->auiDigits[2]. Using the memory map, it is easy to see that either of these instruction sequences would work:

Using indirect addressing:

```
movl %ebp, %eax       # EAX contains ff9f2158
addl $12, %eax       # EAX contains ff9f2164, alias &oAddend2
movl (%eax), %eax    # EAX contains f7fd1008, alias oAddend2
addl $4, %eax        # EAX contains f7fd100c, alias oAddend2->auiDigits
movl $2, %ecx        # ECX contains 2, alias the index
sall $2, %ecx        # ECX contains 8, alias a byte offset
addl %ecx, %eax      # EAX contains f7fd1014, alias oAddend2->auiDigits + 2
movl (%eax), %eax    # EAX contains 00000000, alias *(oAddend2->auiDigits + 2), alias oAddend2->auiDigits[2]
```

Using base-pointer and indexed addressing:

```
movl 12(%ebp), %eax    # EAX contains f7fd1008, alias oAddend2
movl $2, %ecx          # ECX contains 2, alias the index
movl 4(%eax, %ecx, 4), %eax # EAX contains 00000000, alias oAddend2->auiDigits[2]
```