CSC 6580 Spring 2020

Instructor: Stacy Prowell

Homework: Several Square Roots

Modify the square root program to compute and print the square root of all arguments to the program. There may be zero arguments. You can use `argc`, or you can watch for the `NULL` pointer in `argv` to iterate through the list, but watch out for registers being clobbered by the functions you call! Call your program `sqrt_list`.

```
$ sqrt_list
$ sqrt_list 16 9 65536 2
sqrt(16.000000) = 4.000000
sqrt(9.000000) = 3.000000
sqrt(65536.000000) = 256.000000
sqrt(2.000000) = 1.414214
```

Solutions

- Advance the array pointer and watch for NULL
- Use a counter and check for the end of the array

Remember that some registers can be modified by called routines; it is up to the *caller* (that's you) to preserve them. Others should not be modified by called routines; it is up to the *callee* to preserve them.

- Preserved: `RBP`, `RBX`, `R12`, `R13`, `R14`, and `R15`

For instance, if you use `RBX` to hold a pointer initialized to `argv`, then you don't have to save it. If you use `EBX` to hold the index into `argv`, then you can just check to see if it is equal to `argc`.

Solutions

```
        mov rbx, QWORD [rbp-12]
loop:
        mov rdi, QWORD [rbx]
        test rdi, rdi
        je done
                    ; … [rdi]
        add rbx, 8
        jmp loop
done:
```

```
; remove adjusting argv
          mov rbx, 0
loop:
          inc ebx
          cmp ebx, DWORD [rbp-4]
          jge good
            ; … [rdi+rbx*8]
          jmp loop
good:
```

Trivia!

If you write the first line of your program correctly, you should be able to compile it with the following bash function:

```
function ac() { eval $( head -1 $1 | cut -c3- ) ; }
```

That assumes that the *first line* of your program tells how to compile it into an executable with the correct name.

```
ac sqrt_list.asm
```

Two's Complement Arithmetic

Representations

There are *many* ways to represent numbers in binary.

- Packed Binary Coded Decimal (BCD): Use a nybble for each digit.

  `87 = 0x87 = 0b 1000 0111`

  The adjust flag (AF) is used explicitly for this representation.

Representations

- Ones complement.

```
87 = 0x57 = 0b 0101 0111
```

- Negation (complement) is performed by subtracting from all ones.

```
   255 = 0xff = 0b 1111 1111 = -0
-   87 = 0x57 = 0b 0101 0111
   168 = 0xA8 = 0b 1010 1000 = -87
```

- Values for *n* bits run from -2^{n-1} to 2^{n-1}. Every positive number and negative number has the expected complement. There is also a -0 represented by all ones.

Representations

- With ones complement some math doesn't work the way you'd expect.

```
(0-1) = 0b 0000 0000 - 0b 0000 0001
            = 0b 1111 1111 = -0

3 + (-2) = 0b 0000 0011 + (0b 1111 1111 - 0b 0000 0010)
              = 0b 0000 0011 +  0b 1111 1101
              = 0b 0000 0000 = 0
```

Representations

- Two's complement: Negation in an *n*-bit field is performed by subtracting from 2^n.

```
  256 = 0x100 = 0b 1 0000 0000
-  87 =  0x57 = 0b   0101 0111
  169 =  0xa9 = 0b   1010 1001 = -87
```

- Now adding, subtracting, and multiplication work as you'd expect, with both positive and negative numbers.

```
3 + (-2) = 0b 0000 0011 + (0b 1 0000 0000 - 0b 0000 0010)
         = 0b 0000 0011 + 0b 1111 1110
         = 0b 0000 0001 = 1
```

Representations

Basic negation in two's complement: flip all bits, and then add one.

```
-2 = - 0b 0000 0010 = 0b 1111 1101 + 1 = 0b 1111 1110
```

Representations

Still a trade-off. Negation of one particular value doesn't work as expected.

```
-128 = - 0b 1000 0000 = 0b 0111 1111 + 1 = 0b 1000 0000 = -128
```

The value 128 cannot be represented in 8 bit two's complement. The values range from -2^{n} up to 2^{n-1}.

Representations

This is important to understand. This means that *it does not matter* whether we consider numbers to be signed or unsigned in assembly, for the most part.

```
add eax, ebx
```

This adds EAX and EBX and stores the value in EAX, and it works whether we consider EAX and EBX to be signed or not.

```
 -5 + 6 = 0b 1111 1011 + 0b 0000 0110 = 0b 0000 0001 = 1
251 + 6 = 0b 1111 1011 + 0b 0000 0110 = 0b 0000 0001 = 1 (257 mod 2^8)
```

Inline Assembly

Inline assembly

- Inline assembly on GCC (others are different)
- `asm` vs `__asm__`

  The keywords `asm`, `typeof` and `inline` are not available in programs compiled with `-ansi` or `-std` (although `inline` can be used in a program compiled with `-std=c99` or a later standard).

  The way to solve these problems is to put '__' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, and `__inline__` instead of inline.

- `__volatile__` (and `volatile`)
- AT&T vs Intel format

AT&T and Intel

There are two major dialects of assembly language for X86: Intel and AT&T

- **Intel syntax** was originally used in the documentation of the Intel processor and is the dialect primarily used in Windows
- **AT&T syntax** was created by Bell Labs (who created Unix) and is only common in the Unix / Linux world

AT&T versus Intel: Operand Prefixes

AT&T:

```
addq $0x21, %rax
```

Immediate

Register

Intel:

```
add rax, 0x21
```

AT&T versus Intel: Mnemonic Suffixes

AT&T:

```
addq $0x21, %rax
```

q=quad, l=long, w=word, b=byte

Intel:

```
add rax, 0x21
```

AT&T versus Intel: Operand Order

AT&T: `movl %esp, %ebp`

Intel: `mov ebp, esp`

AT&T versus Intel: Memory References

Memory references consist of:

- (Optional) Segment **register**
- Base **register**
- (Optional) Index **register**
- (Optional) A scale multiplier **constant** (1, 2, 4, 8 - default is 1)
- (Optional) A displacement **constant**

The actual address is [*base*] + [*index*]**scale* + *displacement*

(Read about prefix and SIB bytes here: https://wiki.osdev.org/X86-64_Instruction_Encoding)

AT&T versus Intel: Memory References

AT&T: `lea %fs : -0x4a ( %rcx , %rax , 2 ) , %eax`

Intel: `lea eax, fs : [ rcx + rax * 2 -0x4a ]`

Segment

AT&T versus Intel: Memory References

AT&T: `lea %fs : -0x4a ( %rcx , %rax , 2 ) , %eax`

Intel: `lea eax, fs : [ rcx + rax * 2 -0x4a ]`

Base Register

AT&T versus Intel: Memory References

AT&T: `lea %fs : -0x4a ( %rcx , %rax , 2 ) , %eax`

Intel: `lea eax, fs : [ rcx + rax * 2 -0x4a ]`

Index Register

AT&T versus Intel: Memory References

AT&T: `lea %fs : -0x4a ( %rcx , %rax , 2 ) , %eax`

Intel: `lea eax, fs : [ rcx + rax * 2 -0x4a ]`

Scale Factor (1,2,4,8)

AT&T versus Intel: Memory References

AT&T: `lea %fs : -0x4a ( %rcx , %rax , 2 ) , %eax`

Intel: `lea eax, fs : [ rcx + rax * 2 -0x4a ]`

Displacement Value

Default for GCC is AT&T, but Intel is supported

```
$ gcc -masm=intel -o hello hello.c
```

Select the Intel assembly dialect.

If you use gdb, you can execute
`set disassembly-flavor intel`
or put this command in your `~/.gdbinit` to make it the default.

Conversely, Capstone can be made to produce AT&T formatted instructions by
`md.syntax = CS_OPT_SYNTAX_ATT`.

Inline Assembly

```
__asm__ __volatile__ (
        assembly
      : outputs
      : inputs
      : clobbers )
```

- The *assembly* is a string, with instructions separated by newlines. Note that in C, juxtaposed strings are concatenated.
- This can be the only part present, if that is all that is needed.

```
__asm__ ("mov edx, eax")
```

Inline Assembly

```
__asm__ __volatile__ (
            assembly
          : outputs
          : inputs
          : clobbers )
```

- The *outputs* and *inputs* connect values from the assembly to variables in the enclosing C program, if necessary.
- The syntax can be cryptic; you have to read about *constraints*. https://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html

```
__asm__("mov %0, eax" : : "r"(0x21));

__asm__("mov rdi, %[form]"
        : : [form] "m"(fmt));

__asm__("mov %0, eax" : "=m"(result));
```

Inline Assembly

```
__asm__ __volatile__ (
            assembly
          : outputs
          : inputs
          : clobbers )
```

- The *clobbers* communicate to the compiler that the assembly changes registers (`rax`, `rbx`, …), the condition flags (`cc`), or memory (`memory`).
- The compiler *does not understand* the assembly, so you (often) need to tell it what is happening… though usually the compiler will figure it out.
- The compiler sometimes keeps memory values cached in registers, and (1) we might overwrite that, and (2) we might change memory directly and invalidate these cached values.

```
__asm__ (“mov rax,17” : : : “rax”);
```

```c
#include <stdio.h>

int main( void ) {
    int value = 17;
    int incr = 12;
    printf("value=%d and incr=%d\n", value, incr);
    __asm__ (
        "mov eax, %[value]\n"
        "add eax, %[incr]\n"
        "mov %[value], eax\n"
        : [value] "=r"(value)
        : "0"(value), [incr] "r"(incr)
        : "eax", "cc"
    );
    printf("value=%d and incr=%d\n", value, incr);
}
```

Next time:
More assembly, and tools

Homework
Due: Tuesday, 4 February

Homework: Binary Math

Modify the binary program to compute the sum and difference of two numbers given on the command line. Call your program `addsub.asm`.

```
$ ./addsub
Expected exactly two integer arguments.
```

Homework: Binary Math

```
$ ./addsub 65535 1
Adding:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000
Subtracting:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1110
```

Homework: Binary Math

```
$ ./addsub 65535 -1
Adding:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1110
Subtracting:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000
```