Architecture 1001: x86-64 Assembly
About this class

Xeno Kovah – 2021
xeno@darkmentor.com

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/4.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Architecture 1001: x86-64 Assembly’ class, available at https://ost.fyi"

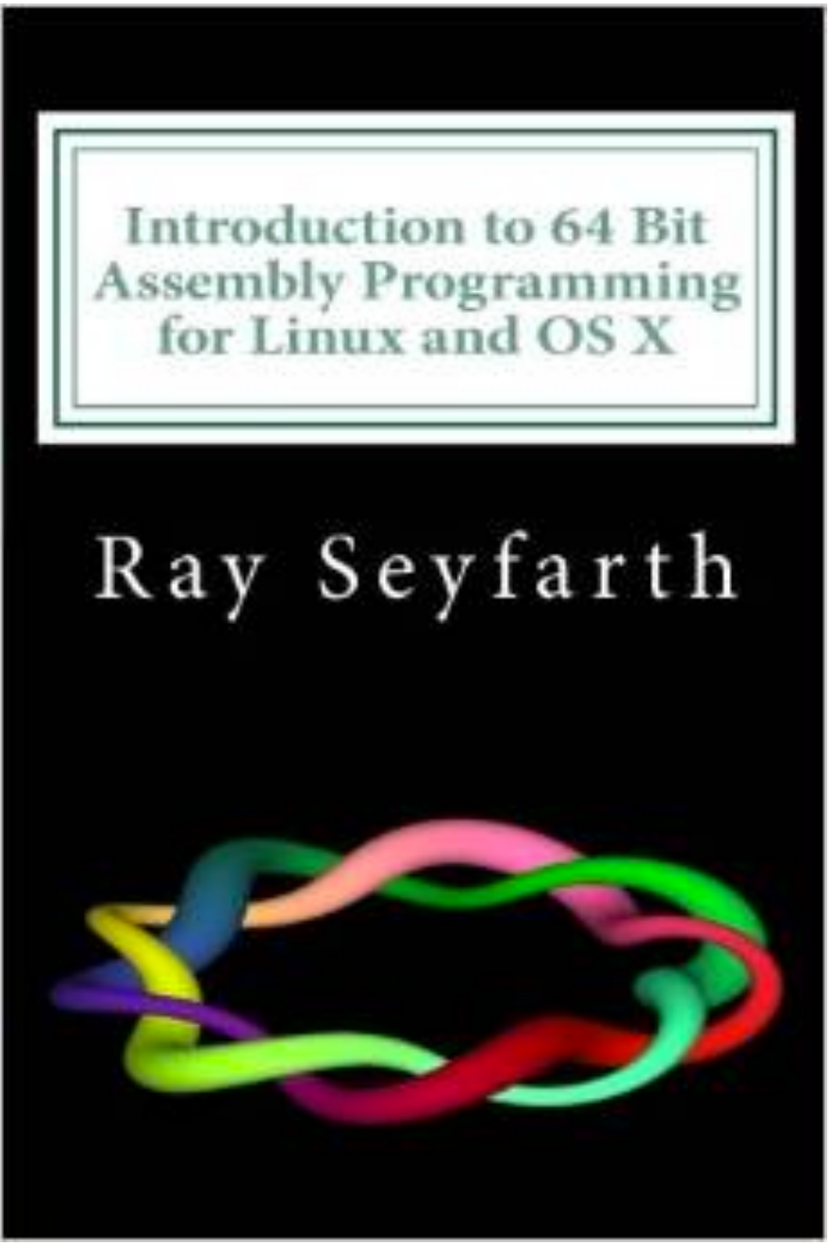
When you're "done" with this class… you're not done. You've just begun.

- I want peers who can do what I can do, and ultimately exceed me
- Mastery requires spending significant time outside of class honing your skills

About the Class

- The intent of this class is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.
  - 64 bit instructions/hardware
  - Implementation of a Stack
  - Common debugging/disassembling tools
- Many things will therefore be intentionally left out or deferred to later classes.
  - Floating point/vector instructions/hardware
  - 16 bit instructions/hardware
  - Complicated or rare instructions
  - Instruction pipeline, alternate modes of operation, hardware virtualization, etc (see other classes for those)

Optional Book (64 bit)



- "Introduction to 64 Bit Assembly Programming for Linux and OS X: Third Edition" by Ray Seyfarth
- Optional book for the class, to give you alternative explanations to my own
- When you see "*Book*" page references in the bottom of slides, it is referring to this book.

Miss Alaineous

- It's called **x**86 because of the progression of Intel chips from **80**86, **801**86, **802**86, etc.
- Originally 16-bit architecture. Later evolved to 32 and 64-bit, but kept the backwards compatibility. The hardware actually starts up in 16 bit before software transitions it to 32 or 64 bit operation.

Miss Alaineous 2

- Intel originally wanted to break from x86 when moving to 64 bit. This was IA64 (Intel Architecture 64 bit) aka Itanium. However, AMD decided to extend x86 to 64 bits itself, leading to the AMD64 architecture. When Itanium had very slow adoption, Intel decided to bite the bullet and license the 64 bit extensions from AMD.
- In the Intel manuals you will see the 64 bit extensions referred to as IA32e or EMT64 or Intel 64 (but never IA64. Again, that's Itanium, a completely different architecture).

Miss Alaineous 3

- You might sometimes see it called amd64 or x64 by MS or some linux distributions
- In this class we're going to go with x86-64

Where is x86-64 used?

- More powerful (but thus power-hungry) systems such as PCs, servers, and even super-computers
  - Minimal adoption on phones or embedded systems. Intel does have the entire Atom line of lower-power chips targeted towards embedded systems though (and they're starting to focus more on performance per Watt, which is where ARM has always been better)

What you’re going to learn:

```
#include <stdio.h>
int main(){
   printf(“Hello World!\n”);
   return 0x1234;
}
```

Is the same as…

```
0000000140001000  sub         rsp,28h
0000000140001004  lea         rcx,[__NULL_IMPORT_DESCRIPTOR+15C0h (0140006000h)]
000000014000100B  call        printf (0140001080h)
0000000140001010  mov         eax,1234h
0000000140001015  add         rsp,28h
0000000140001019  ret
```

Windows Visual Studio 2019 Community
/GS (buffer overflow protection) option turned off
Disassembled with Visual C++

which could be viewed as…

```
main                                                    invoke_main:1400013f4(c)
140001000 48 83 ec 28     SUB         RSP,0x28
140001004 48 8d 0d        LEA         _Argc,[.data] = "Hello World!\n"
          f5 4f 00 00
14000100b e8 70 00        CALL        printf      int printf(char * _Format, ...)
          00 00
140001010 b8 34 12        MOV         EAX,0x1234
          00 00
140001015 48 83 c4 28     ADD         RSP,0x28
140001019 c3              RET
```

Windows Visual Studio 2019 Community
/GS (buffer overflow protection) option turned off
Disassembled with Ghidra 9.2

which is equivalent to…

```
0000000000001149 <main>:
    1149: f3 0f 1e fa                   endbr64
    114d: 55                            push   %rbp
    114e: 48 89 e5                      mov    %rsp,%rbp
    1151: 48 8d 3d ac 0e 00 00          lea    0xeac(%rip),%rdi
    1158: e8 f3 fe ff ff                callq  1050 <puts@plt>
    115d: b8 34 12 00 00                mov    $0x1234,%eax
    1162: 5d                            pop    %rbp
    1163: c3                            retq
    1164: 66 2e 0f 1f 84 00 00          nopw   %cs:0x0(%rax,%rax,1)
    116b: 00 00 00
    116e: 66 90                         xchg   %ax,%ax
```

Ubuntu 20.04, GCC 9.3.0
Disassembled with “objdump -d”

which is equivalent to…

```
_main:
0000000100003f50 pushq    %rbp
0000000100003f51 movq     %rsp, %rbp
0000000100003f54 subq     $0x10, %rsp
0000000100003f58 movl     $0x0, -0x4(%rbp)
0000000100003f5f leaq     0x38(%rip), %rdi          ## literal pool for: "Hello World!
   \n"
0000000100003f66 movb     $0x0, %al
0000000100003f68 callq    0x100003f7e ## symbol stub for: _printf
0000000100003f6d movl     $0x1234, %ecx               ## imm = 0x1234
0000000100003f72 movl     %eax, -0x8(%rbp)
0000000100003f75 movl     %ecx, %eax
0000000100003f77 addq     $0x10, %rsp
0000000100003f7b popq     %rbp
0000000100003f7c retq
```

macOS 11.2, Apple clang version 12.0.0 (clang-1200.0.32.29)
Disassembled from command line with “otool -tV”

But it all boils down to…

```
.text:0000000140001000 sub     rsp, 28h
.text:0000000140001004 lea     rcx, Format     ; "Hello World!\n"
.text:000000014000100B call    printf
.text:0000000140001010 mov     eax, 1234h
.text:0000000140001015 add     rsp, 28h
.text:0000000140001019 retn
```

Windows Visual Studio 2019 Community, /GS option turned off
Disassembled with IDA Freeware 7 (with some omissions for fitting on screen)

Take Heart!

- By one measure, only 14 assembly instructions account for 90% of code!
  - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf
- You've already seen 10 common instructions, just in the hello world variations! (And 2 special security instructions we'll talk about later.)
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently

Take Heart!

- Check out this newer reference!
  - https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.5071&rep=rep1&type=pdf

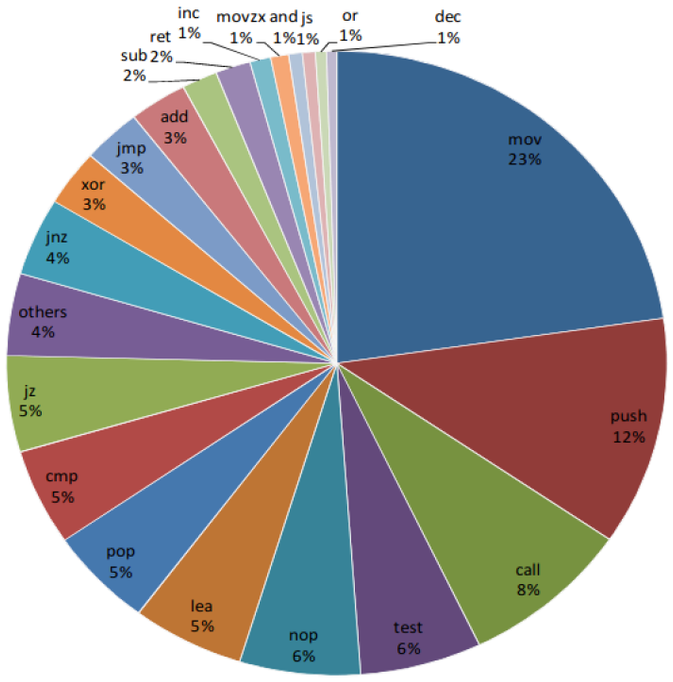


Fig. 2 Web browsers' instruction frequencies

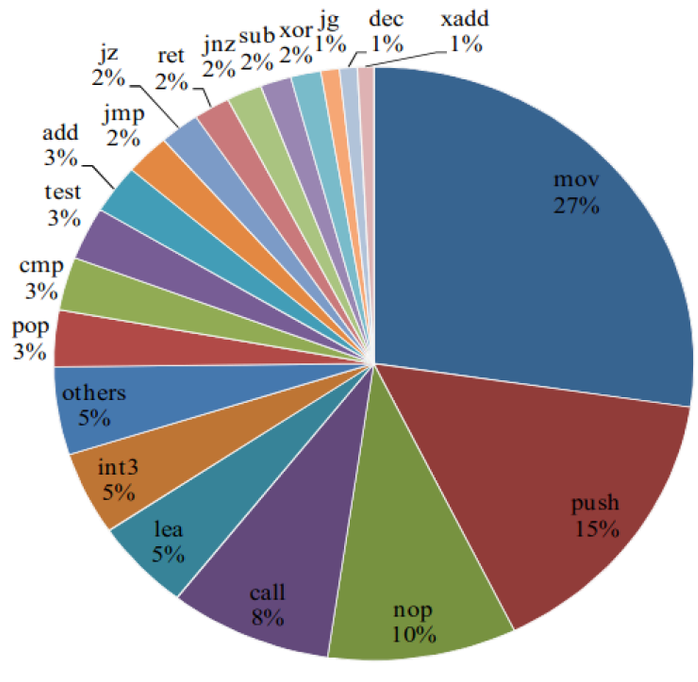


Fig. 14 General Purpose Applications' Instruction Frequencies

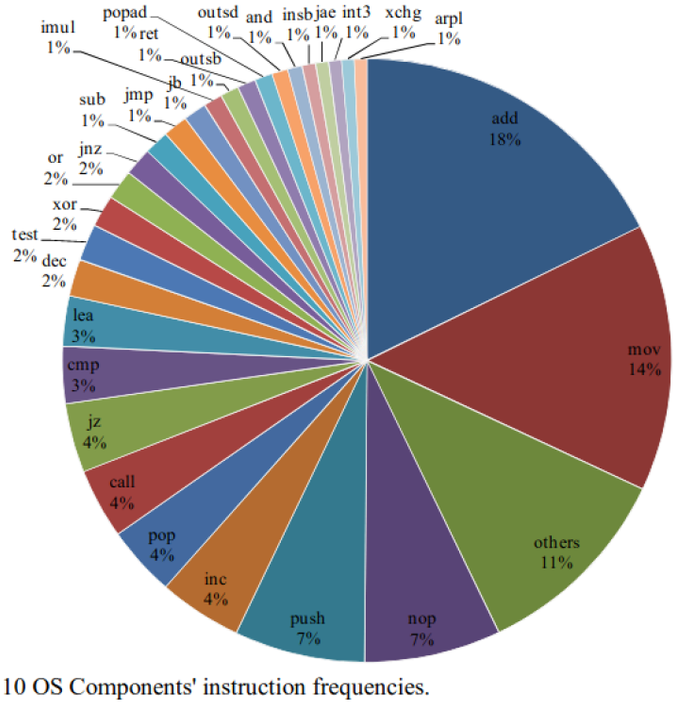


Fig. 10 OS Components' instruction frequencies.

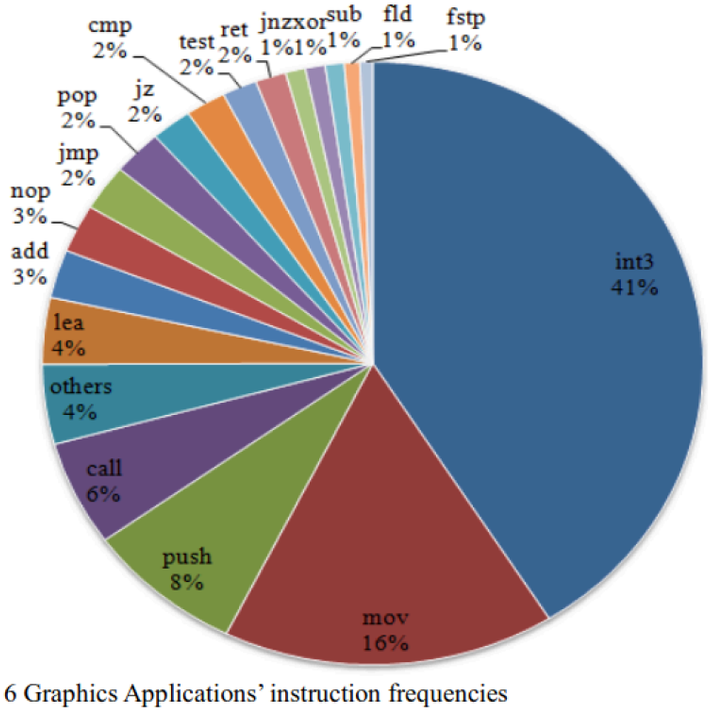


Fig. 6 Graphics Applications' instruction frequencies

Take Heart!

- Check out this newer reference!
  - https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.5071&rep=rep1&type=pdf

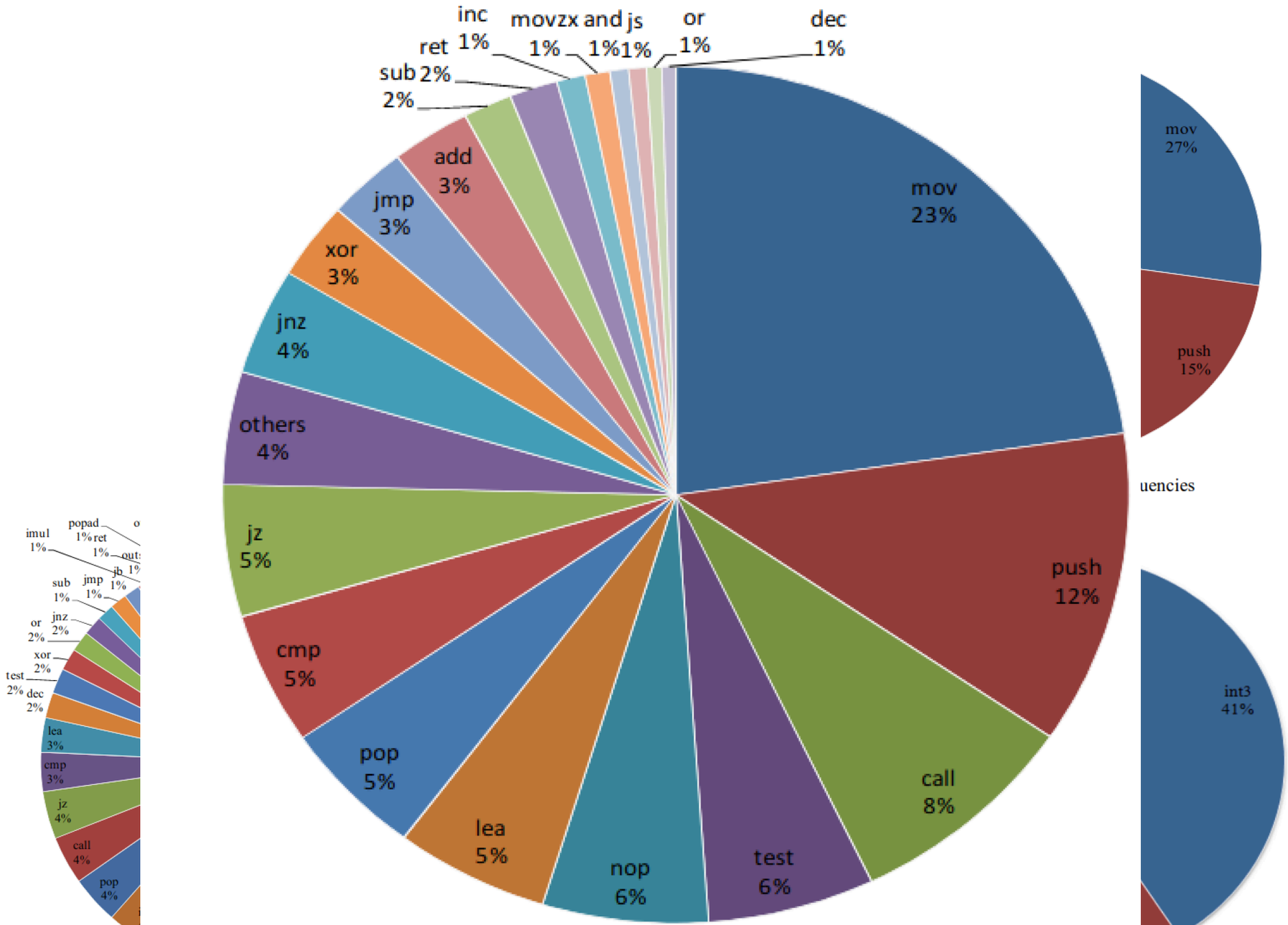


Fig. 2 Web browsers' instruction frequencies

Fig. 10 OS Components' instruction frequencies.

Fig. 6 Graphics Applications' instruction frequencies

Outcomes

- By the end of this class you should…
  - Know the x86-64 general purpose registers, and their 32 and 16 bit sub-register names
  - Understand how data like local variables or return addresses are stored on the stack
  - Understand function calling conventions
  - Be comfortable writing C code in an IDE (like Visual Studio) and reading and stepping through the disassembly (so you can find new instructions)
  - Be able to read assembly well enough to determine the expected inputs to influence the control flow for an opaque binary (the infamous “Binary Bomb lab”)