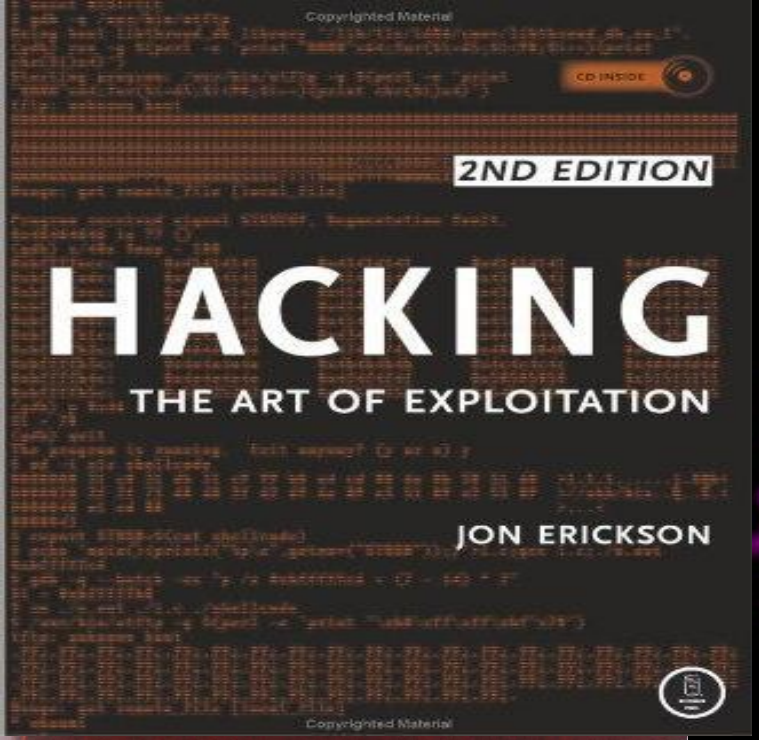
Offensive Security & Reverse Engineering (OSRE)

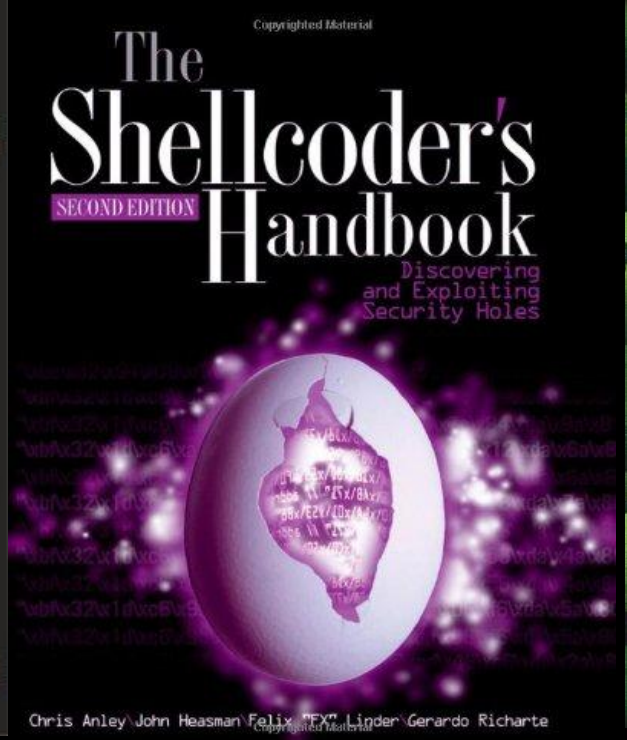
Ali Hadi

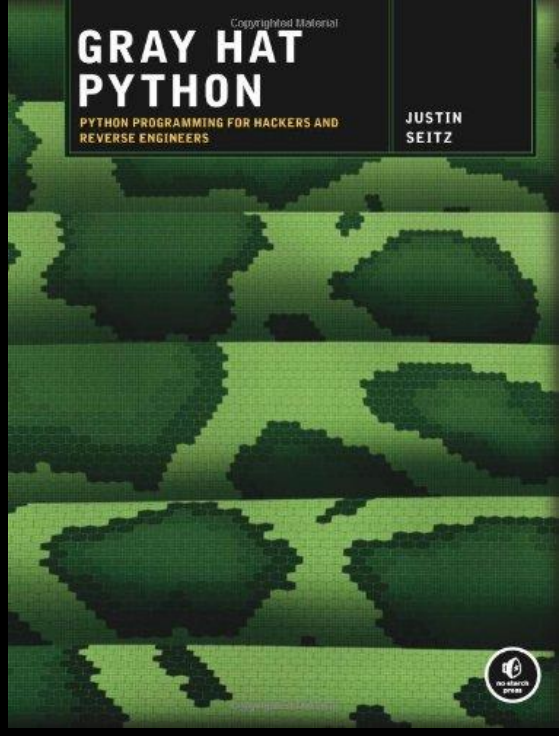
Intro. to x86 Assembly
"Crash Course"

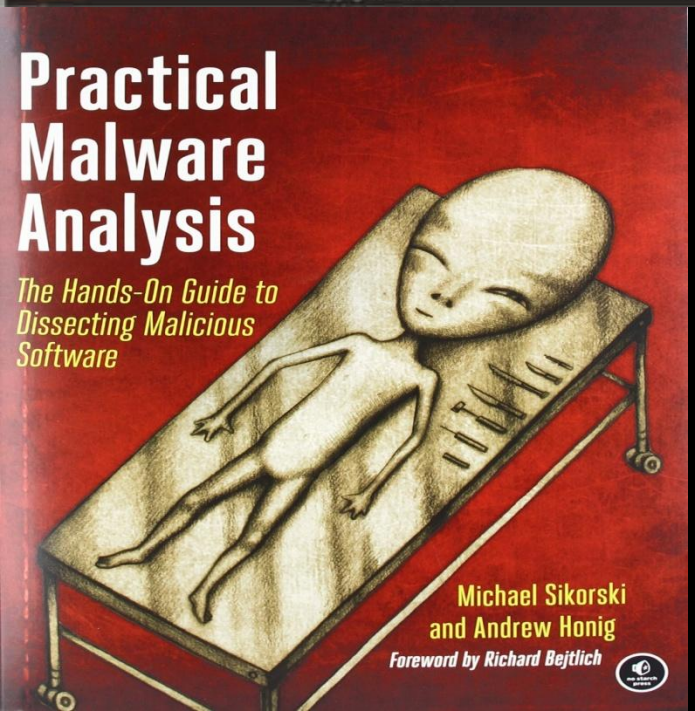
today's lecture has been re-formatted from Xeno Kovah's "Intro. to X86" course found at Open Security Training …

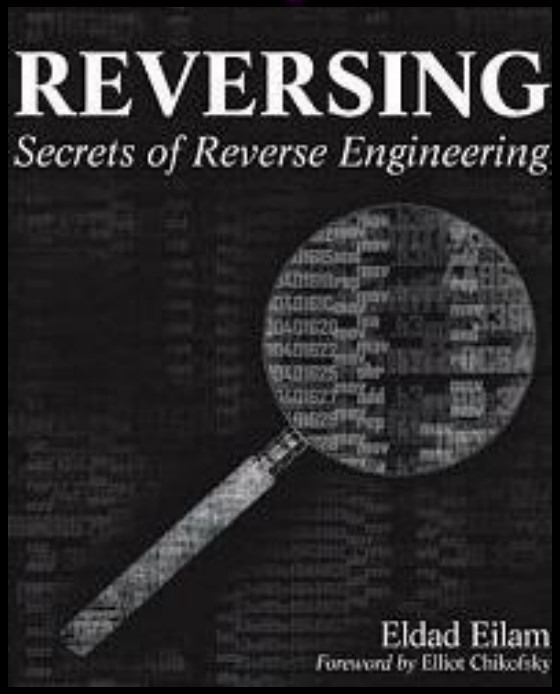
Module #2

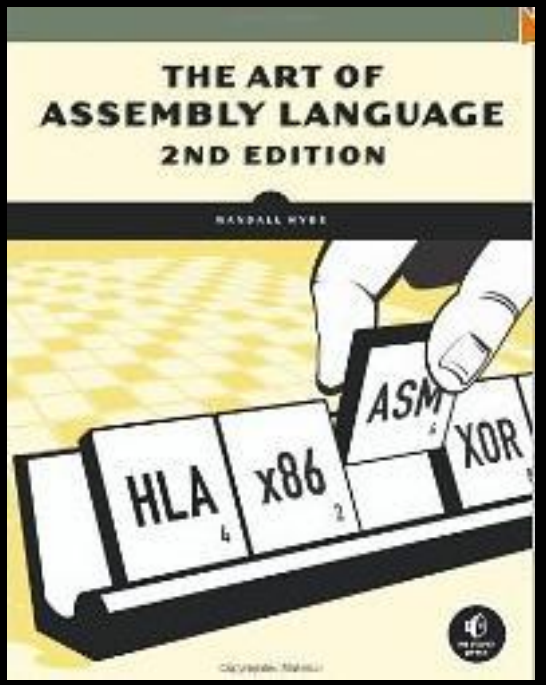

Copyrighted Material
CD INSIDE
2ND EDITION
HACKING
THE ART OF EXPLOITATION
JON ERICKSON
Copyrighted Material


Copyrighted Material
The Shellcoder's Handbook
SECOND EDITION
Discovering and Exploiting Security Holes
Chris Anley John Heasman Felix "FX" Linder Gerardo Richarte


Copyrighted Material
GRAY HAT PYTHON
PYTHON PROGRAMMING FOR HACKERS AND REVERSE ENGINEERS
JUSTIN SEITZ


Practical Malware Analysis
The Hands-On Guide to Dissecting Malicious Software
Michael Sikorski and Andrew Honig
Foreword by Richard Bejtlich


REVERSING
Secrets of Reverse Engineering
Eldad Eilam
Foreword by Elliot Chikofsky


THE ART OF ASSEMBLY LANGUAGE
2ND EDITION
RANDALL HYDE
HLA
x86
ASM
XOR

About this Lecture

The intent of this lecture is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.

Covers

- The intent of this lecture is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.
  - 32 bit instructions/hardware
  - Implementation of a Stack

Doesn't Cover

- Floating point instructions/hardware
- 16/64 bit instructions/hardware
- Complicated or rare 32 bit instructions
- Instruction pipeline, caching hierarchy, alternate modes of operation, HW virtualization, etc

What you're going to learn

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

Is the same as…

```
.text:00401730 main
   .text:00401730                push    ebp
   .text:00401731                mov     ebp, esp
   .text:00401733                push    offset aHelloWorld ; "Hello
   world\n"
   .text:00401738                call    ds:__imp__printf
   .text:0040173E                add     esp, 4
   .text:00401741                mov     eax, 1234h
   .text:00401746                pop     ebp
   .text:00401747                retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off

Is the same as…

```
08048374 <main>:
 8048374:       8d 4c 24 04                   lea    0x4(%esp),%ecx
 8048378:       83 e4 f0                      and    $0xfffffff0,%esp
 804837b:       ff 71 fc                      pushl  -0x4(%ecx)
 804837e:       55                            push   %ebp
 804837f:       89 e5                         mov    %esp,%ebp
 8048381:       51                            push   %ecx
 8048382:       83 ec 04                   sub    $0x4,%esp
 8048385:       c7 04 24 60 84 04 08       movl   $0x8048460,(%esp)
 804838c:       e8 43 ff ff ff                call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00             mov    $0x1234,%eax
 8048396:       83 c4 04                   add    $0x4,%esp
 8048399:       59                            pop    %ecx
 804839a:       5d                            pop    %ebp
 804839b:       8d 61 fc                      lea    -0x4(%ecx),%esp
 804839e:       c3                            ret
 804839f:       90                            nop
```

Ubuntu 8.04, GCC 4.2.4
Disassembled with “objdump -d”

Is the same as…

```
_main:
00001fca    pushl   %ebp
00001fcb    movl    %esp,%ebp
00001fcd    pushl   %ebx
00001fce    subl    $0x14,%esp
00001fd1    calll   0x00001fd6
00001fd6    popl    %ebx
00001fd7    leal    0x0000001a(%ebx),%eax
00001fdd    movl    %eax,(%esp)
00001fe0    calll   0x00003005  ; symbol stub for: _puts
00001fe5    movl    $0x00001234,%eax
00001fea    addl    $0x14,%esp
00001fed    popl    %ebx
00001fee    leave
00001fef    ret
```

Mac OS 10.5.6, GCC 4.0.1
Disassembled from command line with “otool -tV”

But it all boils down to…

```
.text:00401000 main
.text:00401000         push    offset aHelloWorld ;
  "Hello world\n"
.text:00401005         call    ds:__imp__printf
.text:0040100B         pop     ecx
.text:0040100C         mov     eax, 1234h
.text:00401011         retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 4.9 Free Version

Instructions Needed

- 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
- Knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 11 instructions, just in the hello world variations!

Refresher(s)

let's remember some basics …

Data Types

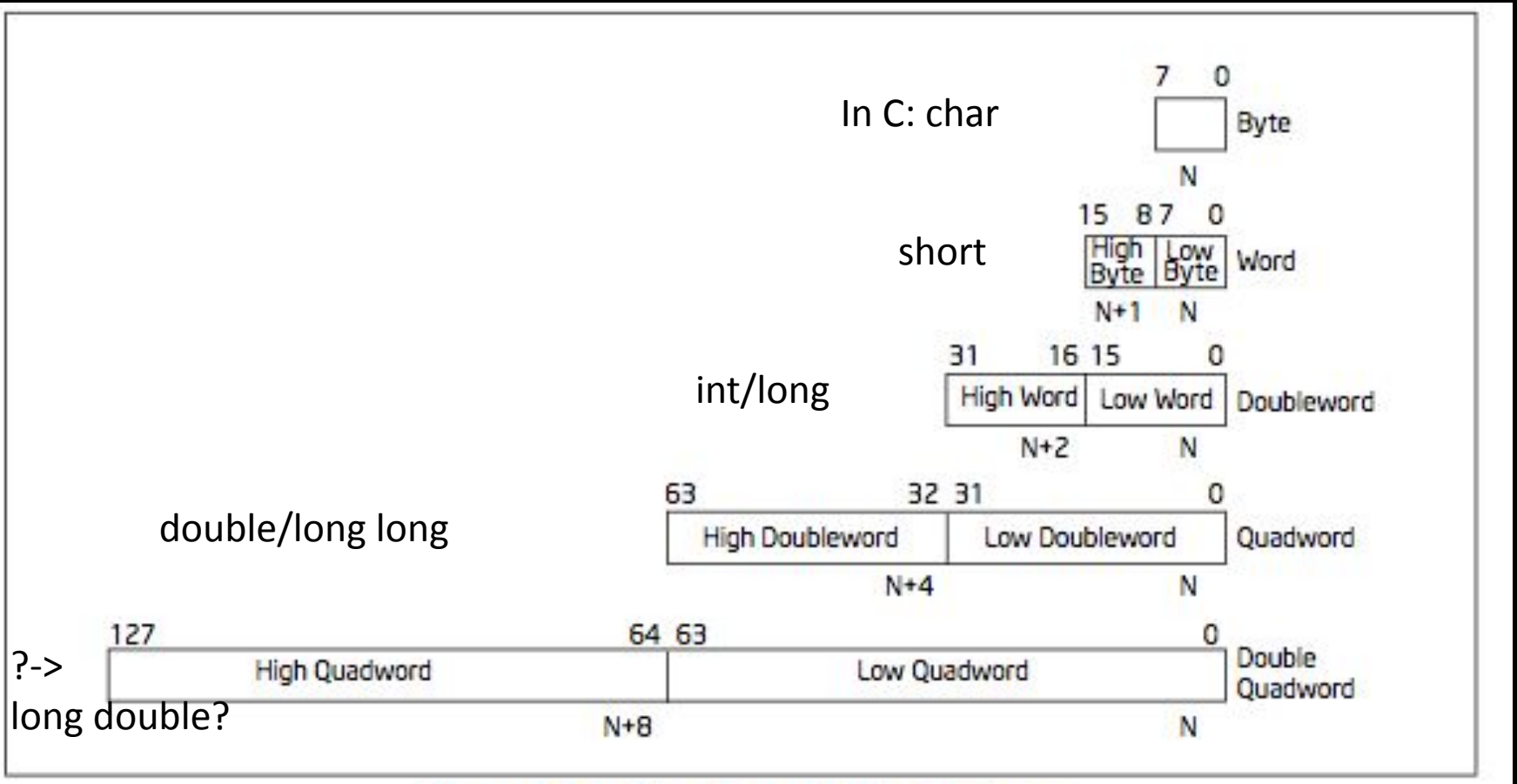


Figure 4-1. Fundamental Data Types

Decimal, Binary, Hexidecimal

Decimal (base 10)	Binary (base 2)	Hex (base 16)
00	0000b	0x00
01	0001b	0x01
02	0010b	0x02
03	0011b	0x03
04	0100b	0x04
05	0101b	0x05
06	0110b	0x06
07	0111b	0x07
08	1000b	0x08
09	1001b	0x09
10	1010b	0x0A
11	1011b	0x0B
12	1100b	0x0C
13	1101b	0x0D
14	1110b	0x0E
15	1111b	0x0F

Negative Numbers

- “one's complement” = flip all bits. 0->1, 1->0
- “two's complement” = one's complement + 1
- Negative numbers are defined as the “two's complement” of the positive number

Number	One's Comp.	Two's Comp. (negative)
00000001b : 0x01	11111110b : 0xFE	11111111b : 0xFF : -1
00000100b : 0x04	11111011b : 0xFB	11111100b : 0xFC : -4
00011010b : 0x1A	11100101b : 0xE5	11100110b : 0xE6 : -26
?	?	10110000b : 0xB0 : -?

- 0x01 to 0x7F positive byte, 0x80 to 0xFF negative byte
- 0x00000001 to 0x7FFFFFFF positive dword
- 0x80000000 to 0xFFFFFFFF negative dword

Architecture(s)

the machines world …

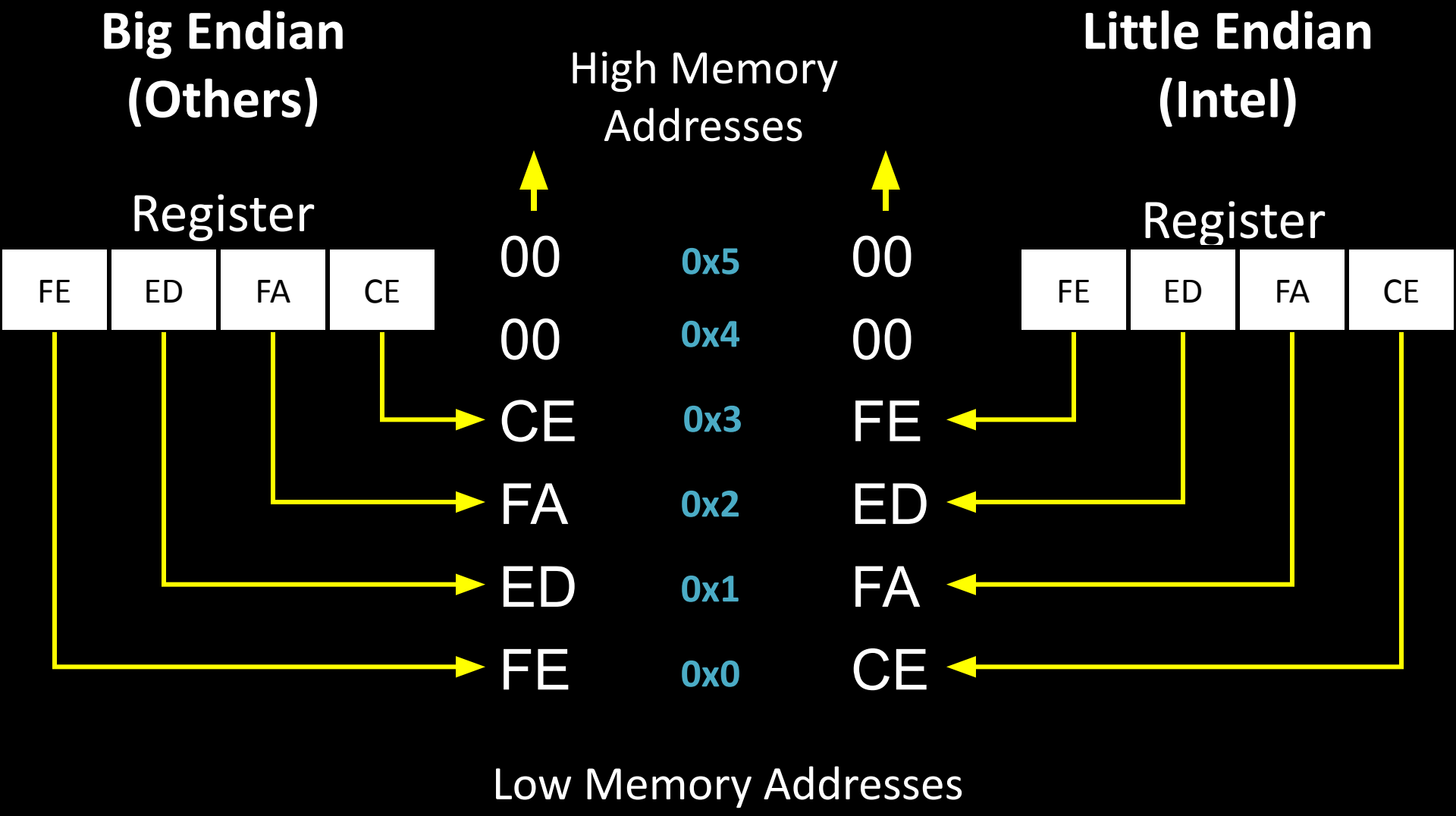
CISC vs. RISC

- Intel is CISC - Complex Instruction Set Computer
  - Many very special purpose instructions that you will never see, and a given compiler may never use
    - just need to know how to use the manual
  - Variable-length instructions, between 1 and 16(?) bytes long.
    - 16 is max len in theory, not sure in practice
- Other major architectures are typically RISC - Reduced Instruction Set Computer
  - Typically more registers, less and fixed-size instructions
  - Examples: PowerPC, ARM, SPARC, MIPS

Endian

- Endianness comes from Jonathan Swift's *Gulliver's Travels*. It doesn't matter which way you eat your eggs :)
- Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most of the others you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

Endianess Pictures



Registers

- Registers are small memory storage areas built into the processor (still volatile memory)
- 8 “general purpose” registers + the instruction pointer which points at the next instruction to execute
  - But two of the 8 are not that general
- On x86-32, registers are 32 bits long
- On x86-64, they're 64 bits

Register Conventions

- These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why.
- **EAX** - Stores function return values
- **EBX** - Base pointer to the data section
- **ECX** - Counter for string and loop operations
- **EDX** - I/O pointer

Intel Arch v1 Section 3.4.1 - General-Purpose Registers

Registers Conventions – Cont.

- **ESI** - Source pointer for string operations
- **EDI** - Destination pointer for string operations
- **ESP** - Stack pointer
- **EBP** - Stack frame base pointer
- **EIP** - Pointer to next instruction to execute (“instruction pointer”)

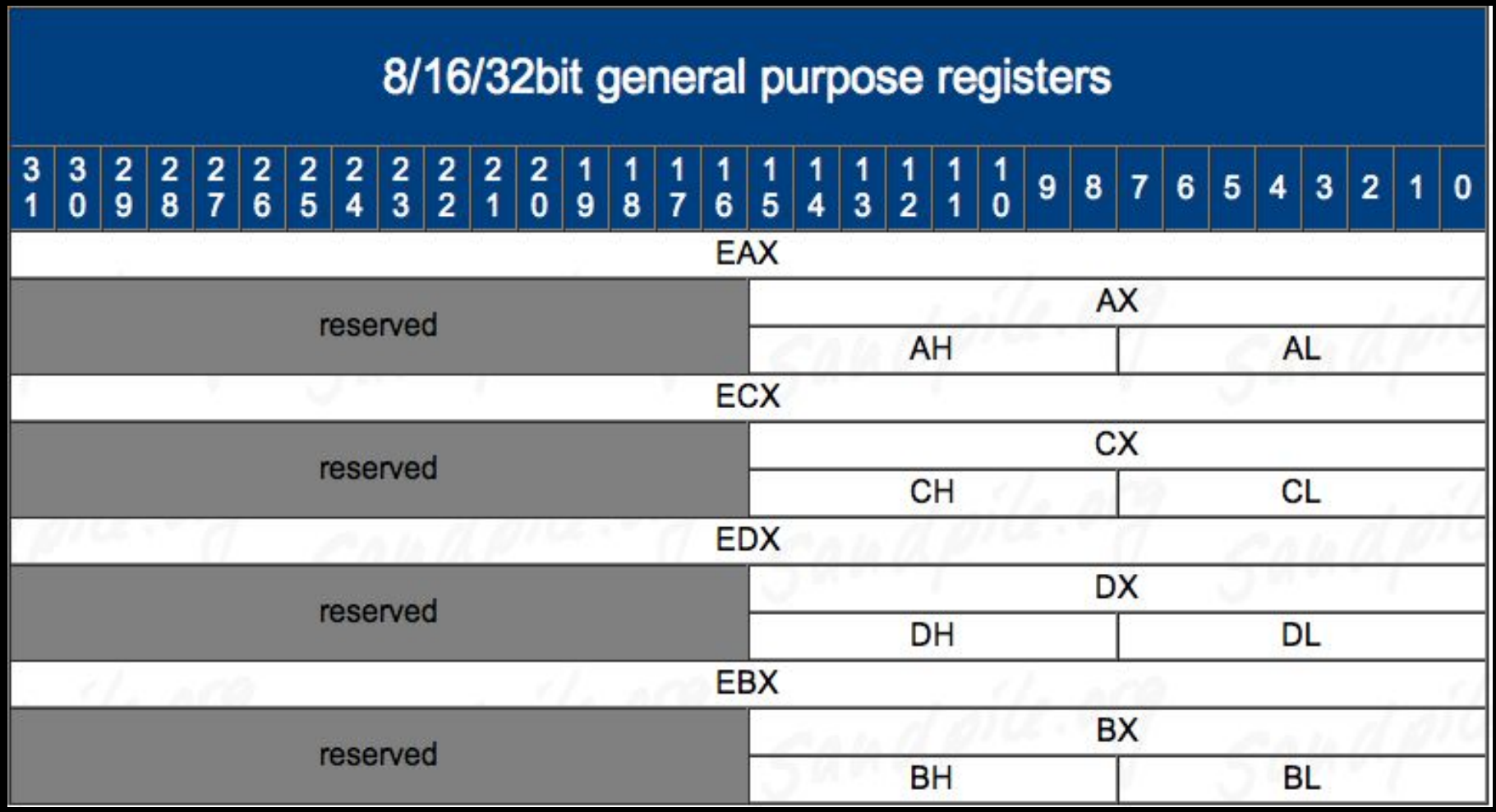
Registers Conventions – Cont.

- Caller-save registers - EAX, EDX, ECX
  - If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  - Put another way - the callee can (and is highly likely to) modify values in caller-save registers

Registers Conventions – Cont.

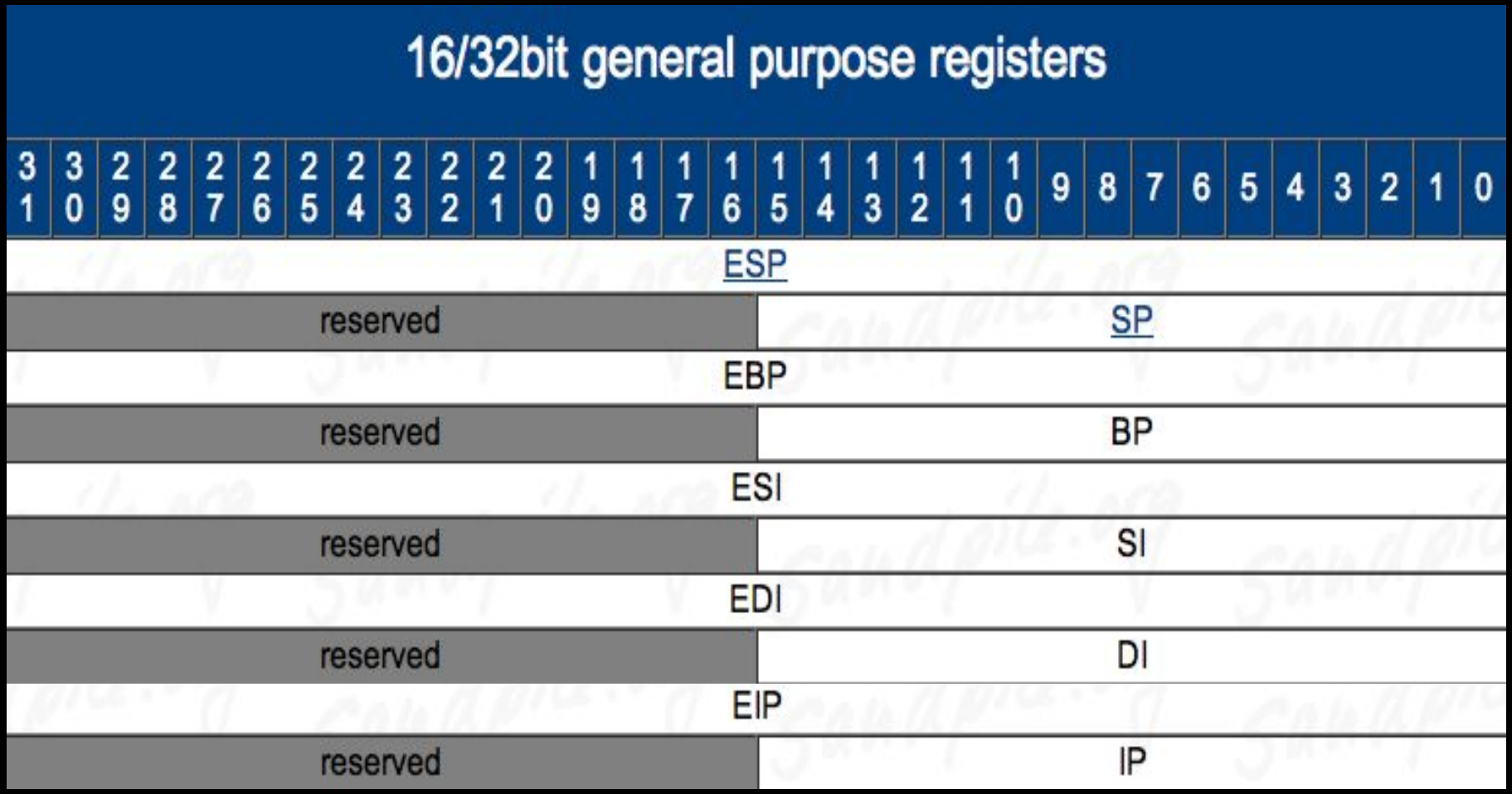
- Callee-save registers - EBP, EBX, ESI, EDI
  - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  - Put another way - the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values

Registers - 8/16/32 bit Addressing



http://www.sandpile.org/ia32/reg.htm

Registers - 8/16/32 bit Addressing – Cont.



http://www.sandpile.org/ia32/reg.htm

EFLAGS

- EFLAGS register holds many single bit flags.
- Remember the following for now:
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

Intel Vol 1 Sec 3.4.3 - page 3-20

Your first x86 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable
  - We'll get to this later ☺

Extra! Extra!
Late-breaking NOP news!

- Amaze those who know x86 by citing this interesting bit of trivia:
- “The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.”
- XCHG instruction is not officially in this class. But if I hadn't just told you what it does, I bet you would have guessed right anyway.

The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  - Different OS start it at different addresses by convention
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses.
- Adding something to the stack means the top of the stack is now at a lower memory address.

The Stack – Cont.

- As already mentioned, ESP points to the top of the stack, the lowest address which is being used
  - While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
- A firm understanding of what is happening on the stack is ***essential*** to understanding a program's operation.

PUSH

Push Word, Dword, Qword onto the Stack

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer ESP by 4.

Registers Before

eax 0x00000003

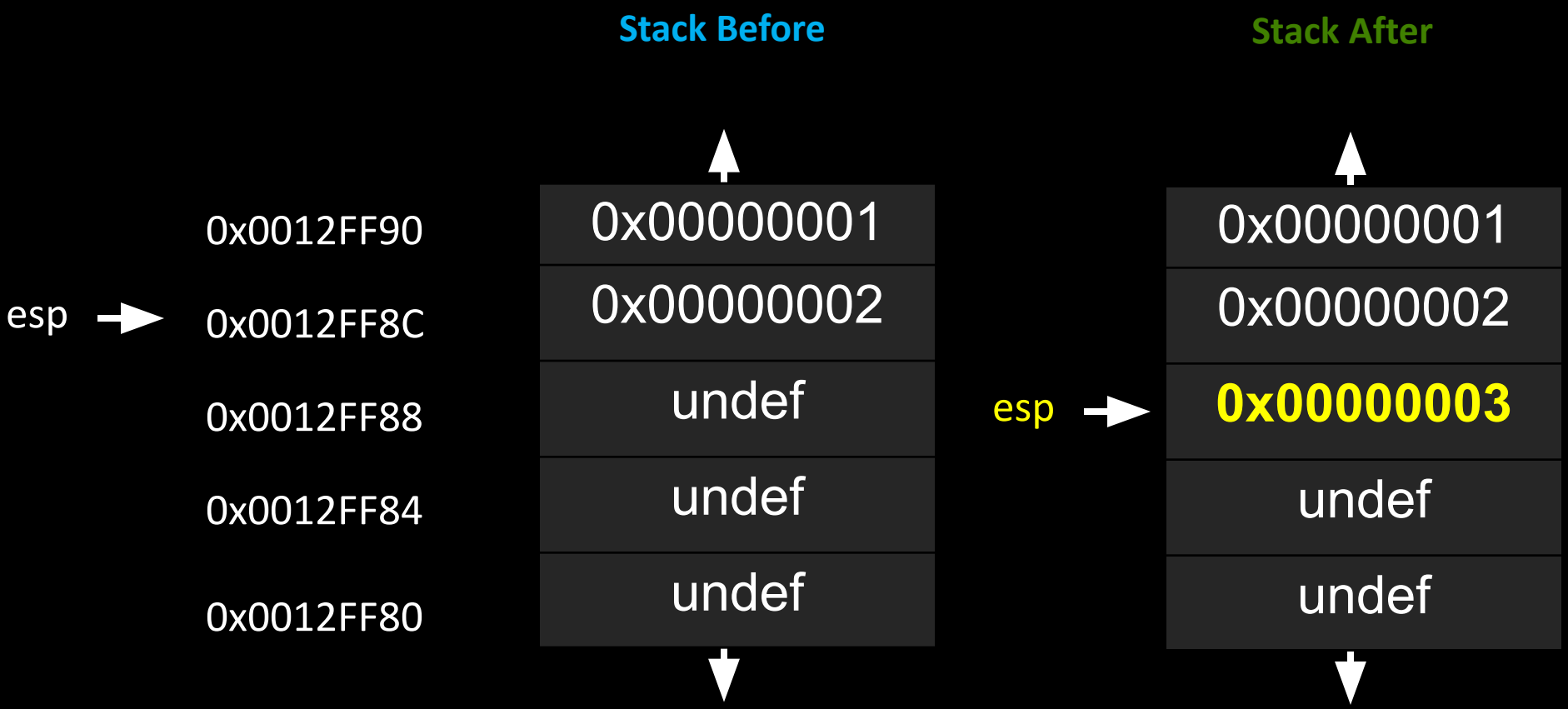
esp 0x0012FF8C

push eax

Registers After

eax 0x00000003

esp 0x0012FF88



POP
Pop a Value from the Stack

- Take a DWORD off the stack, put it in a register, and increment ESP by 4

Registers Before

eax 0xFFFFFFFF

esp 0x0012FF88

pop eax

Registers After

eax 0x00000003

esp 0x0012FF8C

Calling Conventions

- How code calls a subroutine is compiler-dependent and configurable. *But there are a few conventions*.
- We will only deal with the "cdecl" and "stdcall" conventions.
- More info at
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - http://www.programmersheaven.com/2/Calling-conventions

Calling Conventions - cdecl

- "C declaration" - most common calling convention
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- EAX or EAX:EDX returns the result for primitive data types
- Caller is responsible for cleaning up the stack

Calling Conventions - stdcall

- Typically used by Microsoft C++ code (ex: Win32 API)
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- EAX or EDX:EAX returns the result for primitive data types
- Callee responsible for cleaning up any stack parameters it takes

CALL
Call Procedure

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes EIP to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction)

RET
Return from Procedure

Two forms

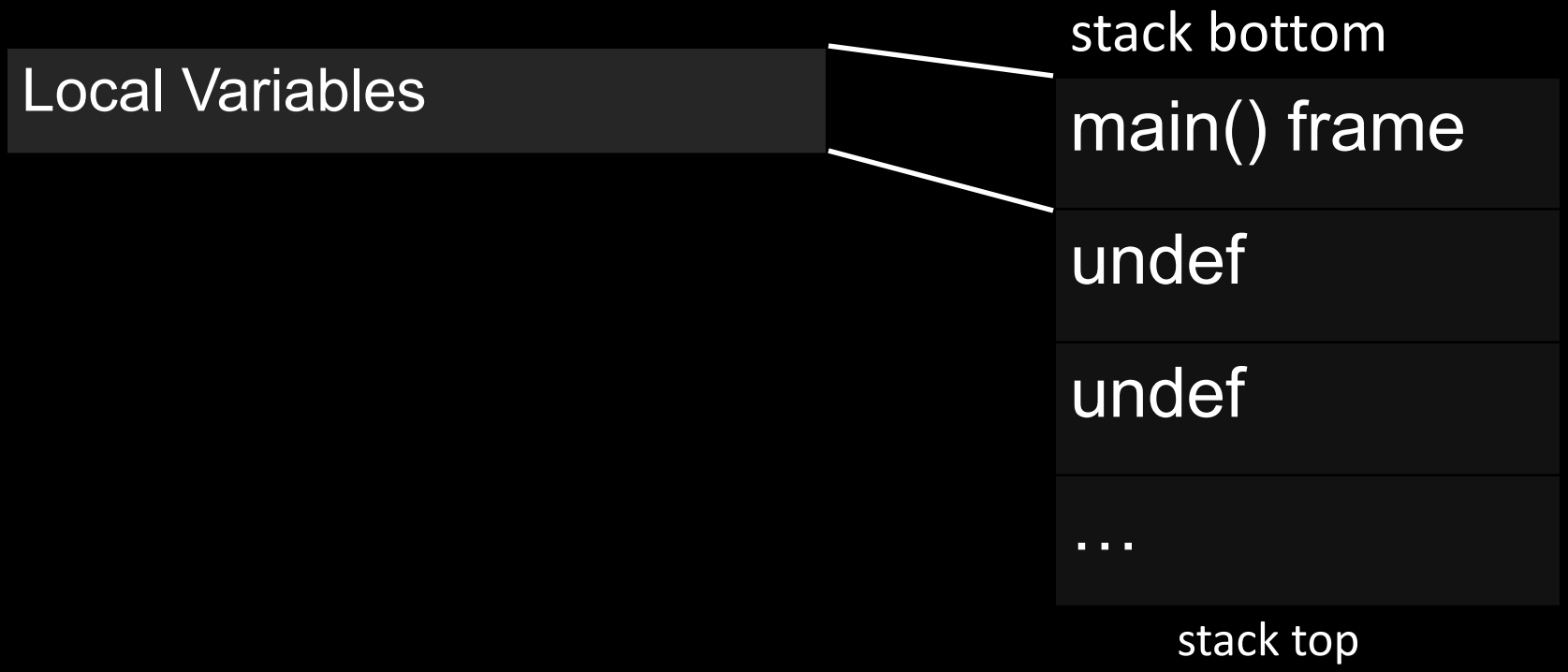
- Pop the top of the stack into EIP (remember pop increments stack pointer)
  - In this form, the instruction is just written as "ret"
  - Typically used by cdecl functions
- Pop the top of the stack into EIP and add a constant number of bytes to ESP
  - In this form, the instruction is written as "ret 0x8", or "ret 0x20", etc
  - Typically used by stdcall functions

MOV
Move

- Can move:
  - register to register
  - memory to register, register to memory
  - immediate to register, immediate to memory
- **Never** memory to memory!
- Memory addresses are given in r/m32 form (coming later)

General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).

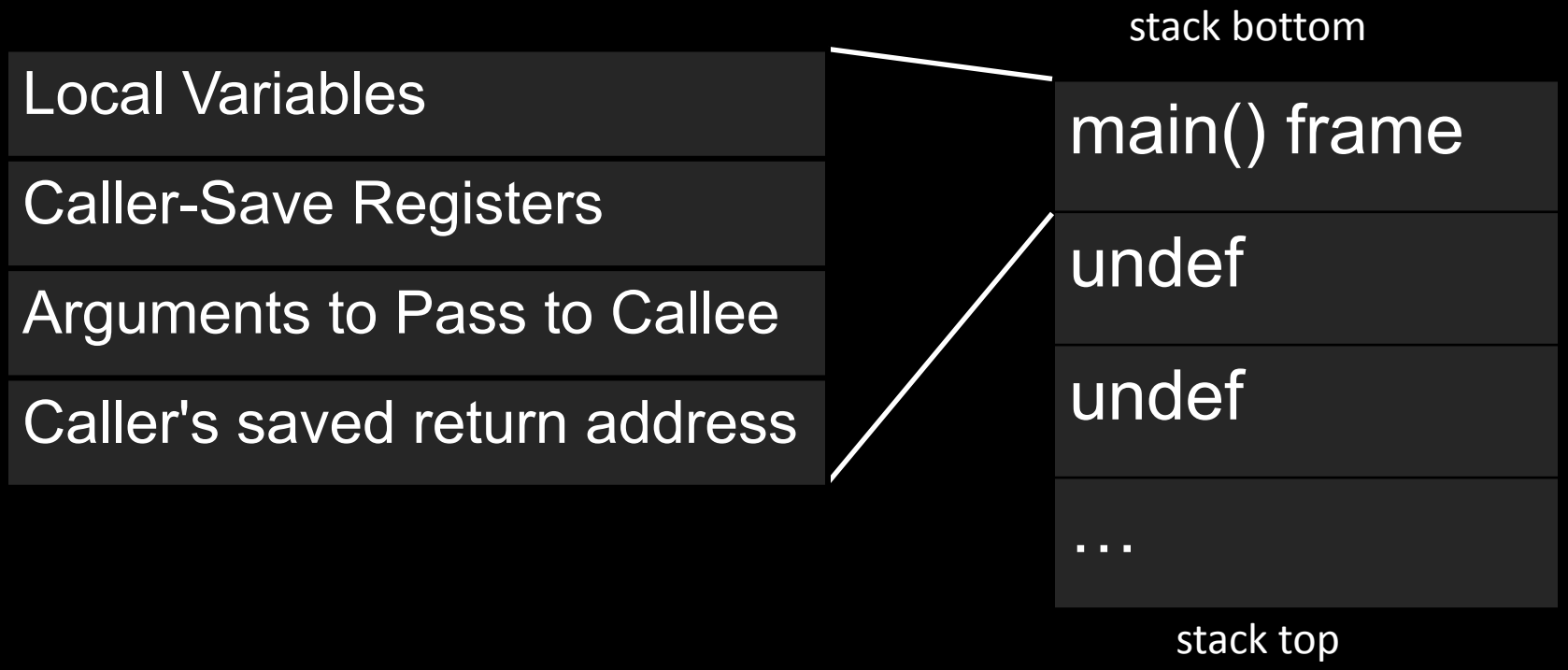


Stack Frame Operation – Cont.

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.

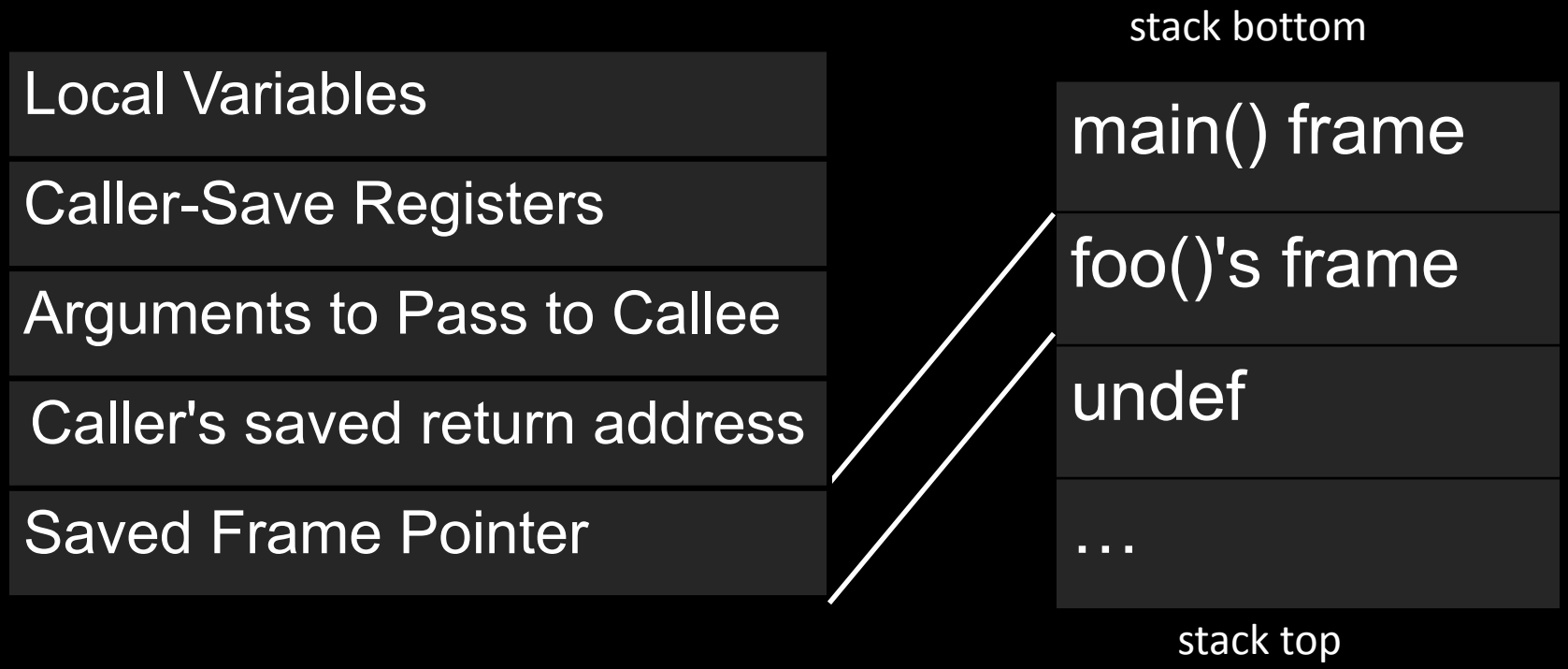
Stack Frame Operation – Cont.

When main() actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



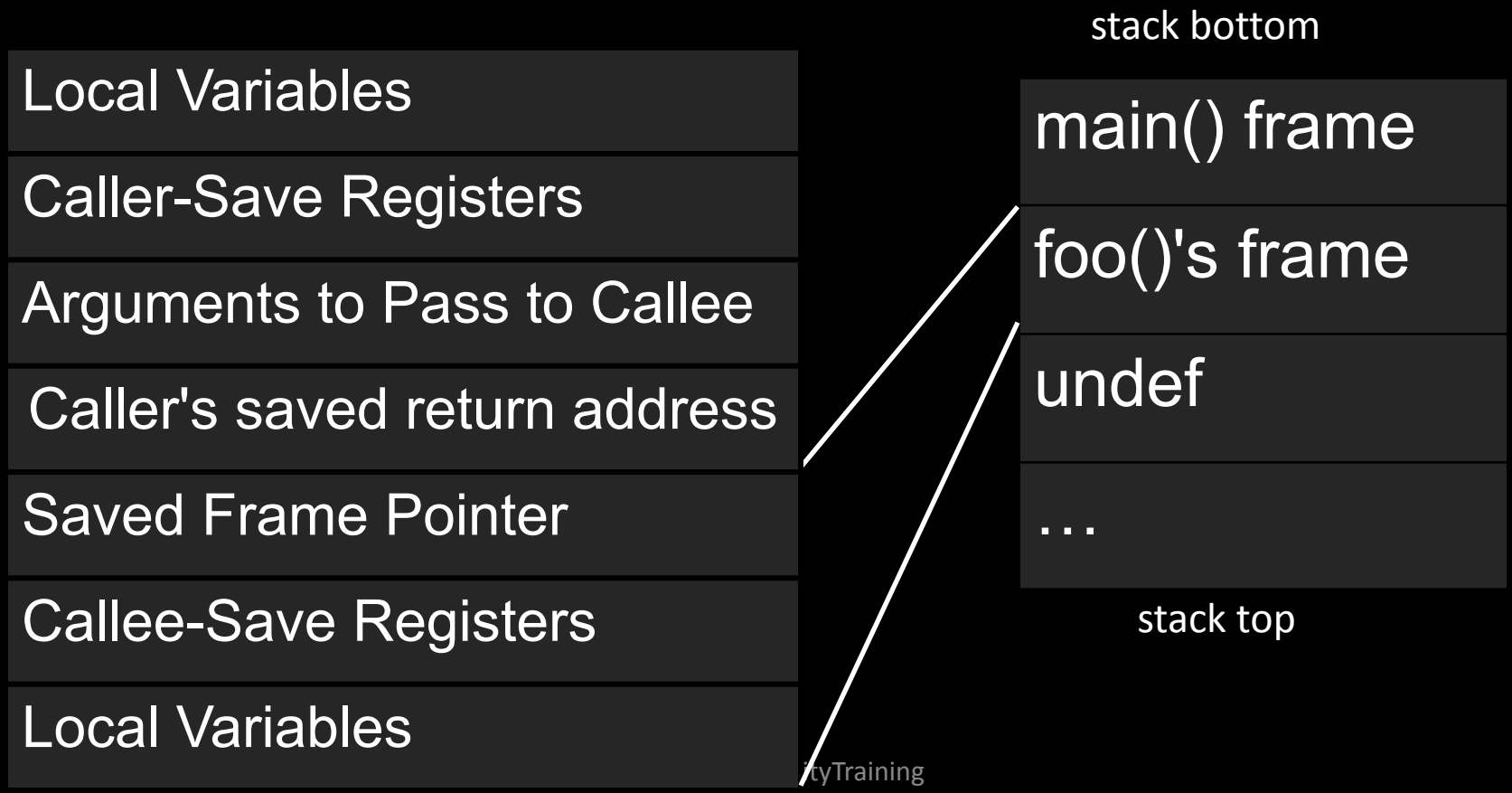
Stack Frame Operation – Cont.

When foo() starts, the frame pointer (EBP) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.



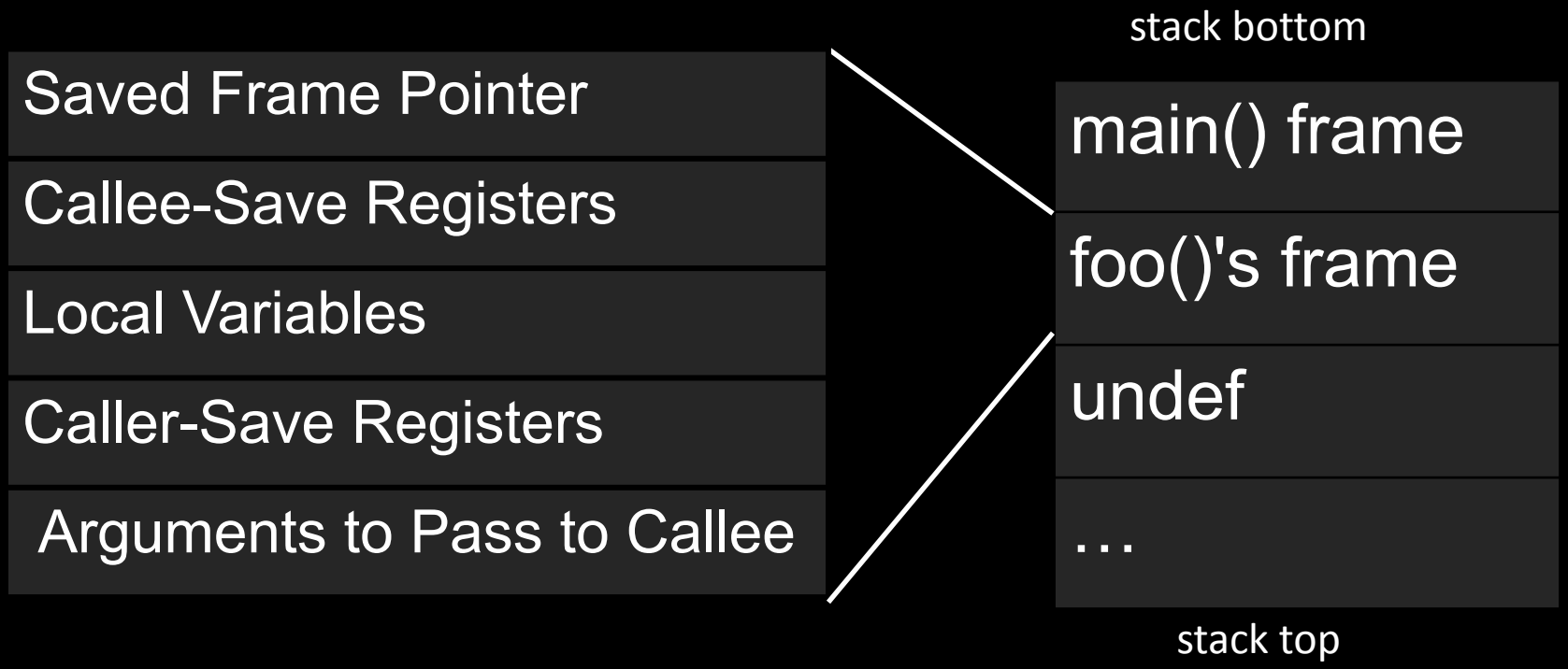
Stack Frame Operation – Cont.

Next, we'll assume the the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.



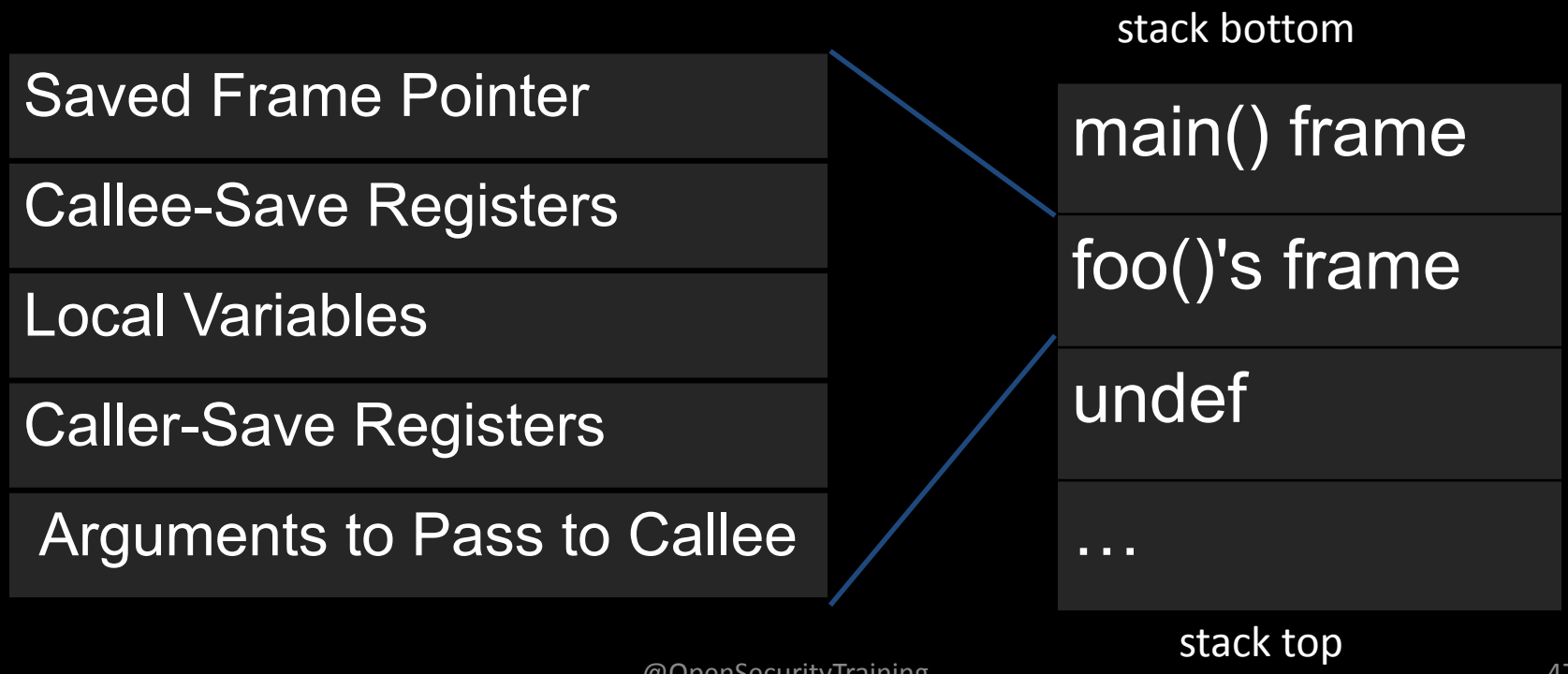
Stack Frame Operation – Cont.

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.



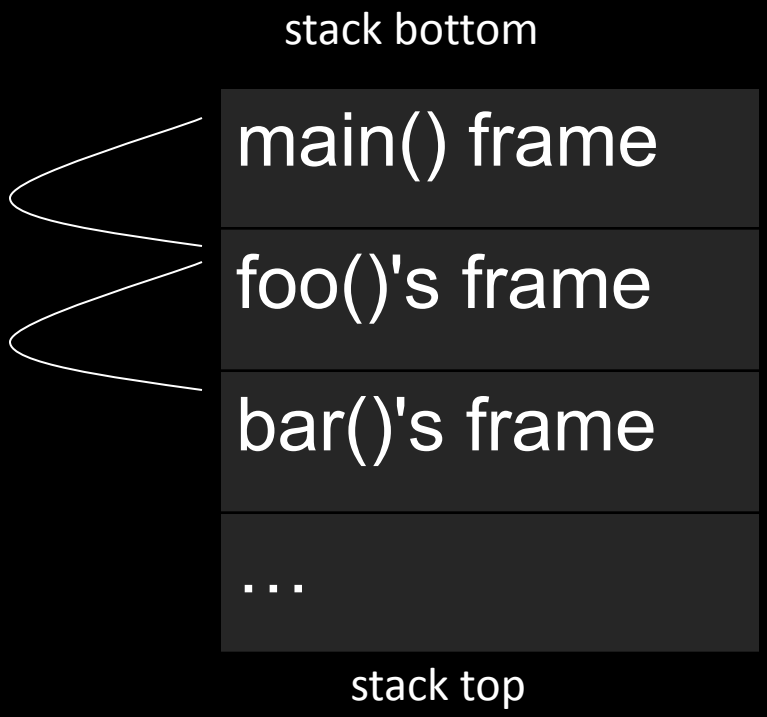
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.)
But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Stack Frames are a Linked List!

The EBP in the current frame points at the saved EBP of the previous frame.



Example1.c

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
        return 0xbeef;
}
int main(){
        sub();
        return 0xf00d;
}
```

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

The stack frames in this example will be very simple.
Only saved frame pointer (EBP) and saved return addresses (EIP).

Example1.c 1:

EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	undef
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Belongs to the frame *before* main() is called

Example1.c – Cont.

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF68 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp  ☒
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8 ♍
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 3

eax	0x003435C0 ⌘
ebp	0x0012FF68 ♍
esp	0x0012FF68

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp ☒
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 4

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF64 ♍

Key:
☒ executed instruction
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h) ☒
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018 ♍
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 5

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF60 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp ☒
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68 ♍
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 6

eax	0x003435C0 ⌘
ebp	0x0012FF60 ♍
esp	0x0012FF60

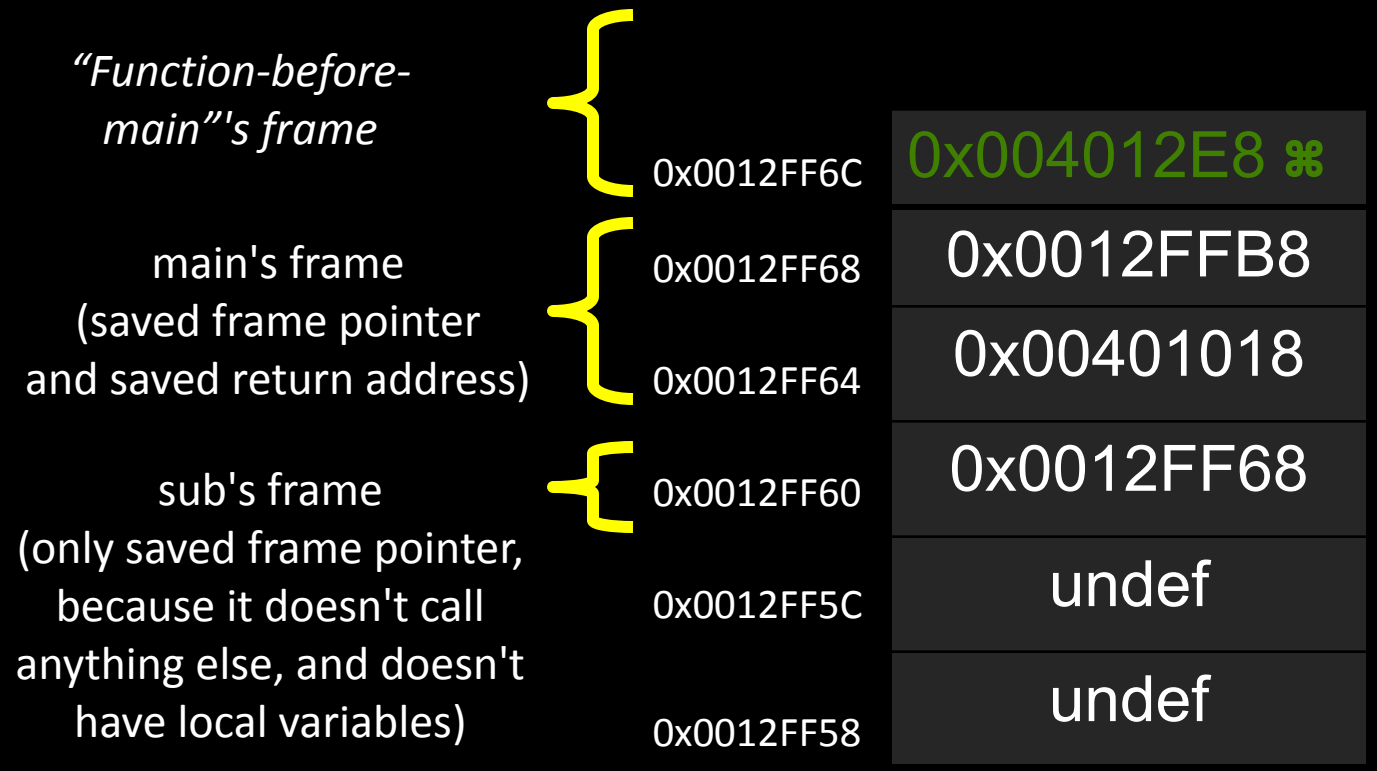
Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp ☒
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 6
STACK FRAME TIME OUT

```
 sub
push    ebp
mov     ebp, esp
mov     eax, 0BEEFh
pop     ebp
retn
main
push    ebp
mov     ebp, esp
call    _sub
mov     eax, 0F00Dh
pop     ebp
retn
```



Example1.c 7

eax	0x0000BEEF
ebp	0x0012FF60
esp	0x0012FF60

☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh ☒
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 8

eax	0x0000BEEF
ebp	0x0012FF68 ♍
esp	0x0012FF64 ♍

☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp ☒
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	undef ♍
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 9

eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret ☒
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef ♍
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 9

eax	0x0000F00D ♍
ebp	0x0012FF68
esp	0x0012FF68

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh ☒
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 10

eax	0x0000F00D
ebp	0x0012FFB8 ♍
esp	0x0012FF6C ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp ☒
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	undef ♍
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 11

eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 ♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret ☒
```

0x0012FF6C	undef ♍
0x0012FF68	undef
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Execution would continue at the value ret removed from the stack: 0x004012E8

Example1 Notes

- sub() is deadcode - its return value is not used for anything, and main always returns 0xF00D.
- If optimizations are turned on in the compiler, it would remove sub()
- Also, because there are no input parameters to sub(), there is no difference whether we compile as cdecl vs stdcall calling conventions

"r/m32" Addressing Forms

- Anywhere you see an r/m32 it means it could be taking a value either from a register or a memory address
- I'm just calling these “r/m32 forms” because anywhere you see “r/m32” in the manual, the instruction can be a variation of the forms in the next slide

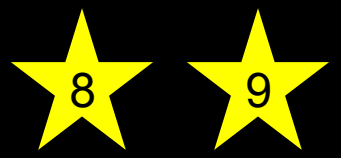
"r/m32" Addressing – Cont.

- In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  - mov eax, ebx
  - mov eax, [ebx]
  - mov eax, [ebx+ecx*X] (X=1, 2, 4, 8)
  - mov eax, [ebx+ecx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: `[base + index*scale + disp]`

LEA
Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/m32 form but **is the exception to the rule** that the square brackets [ ] syntax means dereference (“value at”)

- Example: suppose ebx = 0x2, edx = 0x1000
  - lea eax, [edx+ebx*2]
  - eax = 0x1004, not the value at 0x1004



ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate
- No source ***and*** destination as r/m32s, because that could allow for memory to memory transfer, which isn't allowed on x86
- Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags
- add esp, 8
- sub eax, [ebx*2]

Example2.c - 1

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp ☒
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0xcafe ⌘
ecx	0xbabe ⌘
edx	0xfeed ⌘
ebp	0x0012FF50 ⌘
esp	0x0012FF24 ♍

0x0012FF30	0x12FFB0 (char ** argv)⌘
0x0012FF2C	0x2 (int argc) ⌘
0x0012FF28	Addr after “call _main” ⌘
0x0012FF24	0x0012FF50(saved ebp)♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

executed instruction ☒ modified value ♍ arbitrary example start value ⌘

Example2.c - 2

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24 ♍
esp	0x0012FF24

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 3

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
                          mov     eax, [ebp+0Ch]
                          mov     ecx, [eax+4]
                          push    ecx
                          call    dword ptr ds:__imp__atoi
                          add     esp, 4
                          mov     [ebp-4], eax
                          mov     edx, [ebp-4]
                          push    edx
                          mov     eax, [ebp+8]
                          push    eax
                          call    _sub
                          add     esp, 8
                          mov     esp, ebp
                          pop     ebp
                          retn
```

Caller-save, or space for local var? This time it turns out to be space for local var since there is no corresponding pop, and the address is used later to refer to the value we know is stored in a.

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a) ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 4

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
                          mov     ecx, [eax+4]
                          sh      ecx
                          l       dword ptr ds:__imp__atoi
                          d       esp, 4
                          v       [ebp-4], eax
                          v       edx, [ebp-4]
                          sh      edx
                          v       eax, [ebp+8]
                          sh      eax
                          call    _sub
.text:00000034            add     esp, 8
.text:00000037            mov     esp, ebp
.text:00000039            pop     ebp
.text:0000003A            retn
```

Getting the base of the argv char * array (aka argv[0])

eax	0x12FFB0 ♍
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 5

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
.text:00000017            mov     ecx, [eax+4] ⌧
.text:0000001A            push    ecx
                          ll      dword ptr ds:__imp__atoi
                          d       esp, 4
                          ov      [ebp-4], eax
                          ov      edx, [ebp-4]
                          sh      edx
                          ov      eax, [ebp+8]
                          sh      eax
                          l       _sub
                          d       esp, 8
                          ov      esp, ebp
                          p       ebp
                          :n
```

Getting the char * at argv[1]
(I chose 0x12FFD4 arbitrarily since it's out of the stack scope we're currently looking at)

eax	0x12FFB0
ecx	0x12FFD4♍(arbitrary⌘)
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 6

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
                          op     ebp
                          etn
                          ush    ebp
                          ov     ebp, esp
                          sh     ecx
                          v      eax, [ebp+0Ch]
                          v      ecx, [eax+4]
                          h      ecx ☒
                          ll     dword ptr ds:__imp__atoi ☒
                          dd     esp, 4 ☒
                          ov     [ebp-4], eax
                          ov     edx, [ebp-4]
                          ush    edx
                          ov     eax, [ebp+8]
                          ush    eax
                          ll     _sub
                          dd     esp, 8
                          ov     esp, ebp
                          op     ebp
                          etn
```

Saving some slides… This will push the address of the string at argv[1] (0x12FFD4). atoi() will read the string and turn in into an int, put that int in eax, and return. Then the adding 4 to esp will negate the having pushed the input parameter and make 0x12FF1C undefined again (this is indicative of cdecl)

eax	0x100♍ (arbitrary⌘)
ecx	0x12FFD4
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef ♍
0x0012FF18	undef ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 7

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
                          mov    eax, [ebp+0Ch]
                          ov     ecx, [eax+4]
                          ush    ecx
                          ll     dword ptr ds:__imp__atoi
                          d      esp, 4
                          ov     [ebp-4], eax
                          v      edx, [ebp-4]
                          ush    edx
                          ov     eax, [ebp+8]
                          ush    eax
                          ll     _sub
                          dd     esp, 8
                          ov     esp, ebp
                          op     ebp
                          retn
```

First setting “a” equal to the return value. Then pushing “a” as the second parameter in sub(). We can see an obvious optimization would have been to replace the last two instructions with “push eax”.

eax	0x100
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF1C

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 8

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
                          mov    eax, [ebp+8]
                          push   eax
                          call   _sub
                          add    esp, 8
                          mov    esp, ebp
                          pop    ebp
                          retn
```

Pushing argc as the first parameter (int x) to sub()

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 9

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF14 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034 ♍
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 10

```
.text:00000000 _sub:      push    ebp ⌧
.text:00000001            mov     ebp, esp ⌧
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
.text:00000017            mov     ecx, [eax+4]
.text:0000001A            push    ecx
.text:0000001B            call    dword ptr ds:__imp__atoi
.text:00000021            add     esp, 4
.text:00000024            mov     [ebp-4], eax
.text:00000027            mov     edx, [ebp-4]
.text:0000002A            push    edx
.text:0000002B            mov     eax, [ebp+8]
.text:0000002E            push    eax
.text:0000002F            call    _sub
.text:00000034            add     esp, 8
.text:00000037            mov     esp, ebp
.text:00000039            pop     ebp
.text:0000003A            retn
```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF10 ♍
esp	0x0012FF10 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24(saved ebp)♍
0x0012FF0C	undef

Example2 - 11

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
                          ov     eax, [ebp+8]
                          v      ecx, [ebp+0Ch]
                          a      eax, [ecx+eax*2]
                          op     ebp
                          etn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

Move “x” into eax, and “y” into ecx.

eax	0x2 ♍ (no value change)
ecx	0x100 ♍
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 12

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:0000000E            push   ebp
.text:0000000F            mov    ebp, esp
.text:00000011            push   ecx
.text:00000012            mov    eax, [ebp+0Ch]
.text:00000015            mov    ecx, [eax+4]
.text:00000018            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

Set the return value (eax) to 2*x + y.
Note: neither pointer arith, nor an "address" which was loaded. Just an afficient way to do a calculation.

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 13

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp ⊠
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24 ♍
esp	0x0012FF14 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	undef ♍
0x0012FF0C	undef

Example2 - 14

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn ☒
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
.text:00000017            mov     ecx, [eax+4]
.text:0000001A            push    ecx
.text:0000001B            call    dword ptr ds:__imp__atoi
.text:00000021            add     esp, 4
.text:00000024            mov     [ebp-4], eax
.text:00000027            mov     edx, [ebp-4]
.text:0000002A            push    edx
.text:0000002B            mov     eax, [ebp+8]
.text:0000002E            push    eax
.text:0000002F            call    _sub
.text:00000034            add     esp, 8
.text:00000037            mov     esp, ebp
.text:00000039            pop     ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	undef ♍
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 15

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
.text:00000017            mov     ecx, [eax+4]
.text:0000001A            push    ecx
.text:0000001B            call    dword ptr ds:__imp__atoi
.text:00000021            add     esp, 4
.text:00000024            mov     [ebp-4], eax
.text:00000027            mov     edx, [ebp-4]
.text:0000002A            push    edx
.text:0000002B            mov     eax, [ebp+8]
.text:0000002E            push    eax
.text:0000002F            call    _sub
.text:00000034            add     esp, 8 ☒
.text:00000037            mov     esp, ebp
.text:00000039            pop     ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF20 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	undef ♍
0x0012FF18	undef ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 16

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF24 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 17

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp ☒
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF50 ♍
esp	0x0012FF28 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	undef ♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Control Flow

Two forms of control flow

- Conditional - go somewhere if a condition is met. Think “if”s, switches, loops
- Unconditional - go somewhere no matter what. Procedure calls, goto, exceptions, interrupts.
- We’ve already seen procedure calls manifest themselves as push/call/ret, let’s see how goto manifests itself in assmembly.

JMP
Jump

- Change EIP to the given address
- Main forms of the address
  - Short relative (1 byte displacement from end of the instruction)
    - "jmp 00401023" doesn't have the number 00401023 anywhere in it, it's really "jmp 0x0E bytes forward"
    - Some disassemblers will indicate this with a mnemonic by writing it as "jmp short"
  - Near relative (4byte displacement from current EIP)
  - Absolute (hardcoded address in instruction)
  - Absolute Indirect (address calculated with r/m32)

Example3.c

(Remain calm)

```
int main(){
        int a=1, b=2;
        if(a == b){
                return 1;
        }
        if(a > b){
                return 2;
        }
        if(a < b){
                return 3;
        }
        return 0xdefea7;
}
```

Jcc

```
main:
00401010  push    ebp
00401011  mov     ebp,esp
00401013  sub     esp,8
00401016  mov     dword ptr [ebp-4],1
0040101D  mov     dword ptr [ebp-8],2
00401024  mov     eax,dword ptr [ebp-4]
00401027  cmp     eax,dword ptr [ebp-8]
0040102A  jne     00401033
0040102C  mov     eax,1
00401031  jmp     00401056
00401033  mov     ecx,dword ptr [ebp-4]
00401036  cmp     ecx,dword ptr [ebp-8]
00401039  jle     00401042
0040103B  mov     eax,2
00401040  jmp     00401056
00401042  mov     edx,dword ptr [ebp-4]
00401045  cmp     edx,dword ptr [ebp-8]
00401048  jge     00401051
0040104A  mov     eax,3
0040104F  jmp     00401056
00401051  mov     eax,0DEFEA7h
00401056  mov     esp,ebp
00401058  pop     ebp
00401059  ret
```

Jcc
Jump If Condition Is Met

- There are more than 4 pages of conditional jump types!
- Luckily a bunch of them are synonyms for each other.
- JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

Notable Jcc Instructions

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG : if ZF == 1 or SF != OF
- JGE/JNL : if SF == OF
- JBE: if CF == 1 OR ZF == 1
- JB: if CF == 1

- Note: Don't get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at EFLAGS and/or watch whether it takes a jump.

Flag Setting

- Before you can do a conditional jump, you need something to set the condition flags for you.
- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects

CMP
Compare Two Operands

- "The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction."
- What's the difference from just doing SUB? Difference is that with SUB the result has to be stored somewhere. With CMP the result is computed, the flags are set, but the result is discarded. Thus this only sets flags and doesn't mess up any of your registers.
- Modifies CF, OF, SF, ZF, AF, and PF
- (implies that SUB modifies all those too)

TEST
Logical Compare

- "Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result."
- Like CMP - sets flags, and throws away the result

Example4.c

```
#define MASK 0x100

int main(){
        int a=0x1301;
        if(a & MASK){
                return 1;
        }
        else{
                return 2;
        }
}
```

jcc

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  push      ecx
00401014  mov       dword ptr [ebp-4],1301h
0040101B  mov       eax,dword ptr [ebp-4]
0040101E  and       eax,100h
00401023  je        0040102E
00401025  mov       eax,1
0040102A  jmp       00401033
0040102C  jmp       00401033
0040102E  mov       eax,2
00401033  mov       esp,ebp
00401035  pop       ebp
00401036  ret
```

I actually expected a TEST, because the result isn't stored

Eventually found out why there are 2 jmps!
(no optimization, so simple compiler rules)

Refresher: Boolean (”bitwise”) logic

AND “&”

0	0	0
0	1	0
1	0	0
1	1	1

Operands **Result**

OR “|”

0	0	0
0	1	1
1	0	1
1	1	1

XOR “^”

0	0	0
0	1	1
1	0	1
1	1	0

NOT “~”

0	1
1	0

AND
Logical AND

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

and al, bl

	00110011b (al - 0x33)
AND	01010101b (bl - 0x55)
result	00010001b (al - 0x11)

and al, 0x42

	00110011b (al - 0x33)
AND	01000010b (imm - 0x42)
result	00000010b (al - 0x02)

OR
Logical Inclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

or al, bl

	00110011b (al - 0x33)
OR	01010101b (bl - 0x55)
result	01110111b (al - 0x77)

or al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110011b (al - 0x73)

XOR
Logical Exclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

xor al, al

	00110011b (al - 0x33)
XOR	00110011b (al - 0x33)
result	00000000b (al - 0x00)

xor al, 0x42

	00110011b (al - 0x33)
OR	01000010b (imm - 0x42)
result	01110001b (al - 0x71)

XOR is commonly used to zero a register, by XORing it with itself, because it's faster than a MOV

NOT
One's Complement Negation

- Single source/destination operand can be r/m32

not al

NOT	00110011b (al - 0x33)
result	11001100b (al - 0xCC)

Xeno trying to be clever on a boring example, and failing…

not [al+bl]

al	0x10000000
bl	0x00001234
al+bl	0x10001234
[al+bl]	0 (assumed memory at 0x10001234)
NOT	00000000b
result	11111111b

Example5.c - simple for loop

```
#include <stdio.h>

int main(){
   int i;
   for(i = 0; i < 10; i++){
      printf("i = %d\n“, i);
   }
}
```

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  push      ecx
00401014  mov       dword ptr [ebp-4],0
0040101B  jmp       00401026
0040101D  mov       eax,dword ptr [ebp-4]
00401020  add       eax,1
00401023  mov       dword ptr [ebp-4],eax
00401026  cmp       dword ptr [ebp-4],0Ah
0040102A  jge       00401040
0040102C  mov       ecx,dword ptr [ebp-4]
0040102F  push      ecx
00401030  push      405000h
00401035  call      dword ptr ds:[00406230h]
0040103B  add       esp,8
0040103E  jmp       0040101D
00401040  xor       eax,eax
00401042  mov       esp,ebp
00401044  pop       ebp
00401045  ret
```

What does this add say about the calling convention of printf()?

Interesting note: Defaults to returning 0

SHL
Shift Logical Left

- Can be explicitly used with the C "<<" operator
- First operand (source and destination) operand is an r/m32
- Second operand is either CL (lowest byte of ECX), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **multiplies** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the left hand side are "shifted into" (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shl cl, 2

	00110011b (cl - 0x33)
result	11001100b (cl - 0xCC) CF = 0

shl cl, 3

	00110011b (cl - 0x33)
result	10011000b (cl - 0x98) CF = 1

SHR
Shift Logical Right

- Can be explicitly used with the C “>>” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **divides** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the right hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shr cl, 2

	00110011b (cl - 0x33)
result	00001100b (cl - 0x0C) CF = 1

shr cl, 3

	00110011b (cl - 0x33)
result	00000110b (cl - 0x06) CF = 0

Example6.c

```
//Multiply and divide transformations
//New instructions:
//shl - Shift Left, shr - Shift Right

int main(){
	unsigned int a, b, c;
	a = 0x40;
	b = a * 8;
	c = b / 16;
	return c;
}
```

```
main:
  push    ebp
  mov     ebp,esp
  sub     esp,0Ch
  mov     dword ptr [ebp-4],40h
  mov     eax,dword ptr [ebp-4]
★ shl     eax,3
  mov     dword ptr [ebp-8],eax
  mov     ecx,dword ptr [ebp-8]
★ shr     ecx,4
  mov     dword ptr [ebp-0Ch],ecx
  mov     eax,dword ptr [ebp-0Ch]
  mov     esp,ebp
  pop     ebp
  ret
```

LEAVE
High Level Procedure Exit

```
1026EE94  mov     eax,dword ptr [ebp+8]
1026EE97  pop     esi
1026EE98  pop     edi
1026EE99  leave
1026EE9A  ret
```

- “Set ESP to EBP, then pop EBP”
- That’s all :)
- Then why haven’t we seen it elsewhere already?
- Depends on compiler and options

Back to Hello World

```
.text:00401730 main
.text:00401730          push    ebp
.text:00401731          mov     ebp, esp
.text:00401733          push    offset aHelloWorld ; "Hello world\n“
.text:00401738          call    ds:__imp__printf
.text:0040173E          add     esp, 4
.text:00401741          mov     eax, 1234h
.text:00401746          pop     ebp
.text:00401747          retn
```

Are we all comfortable with this now?

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version

Instructions we now know(20)

- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT
- SHR/SHL
- LEAVE

Intel vs. AT&T Syntax

- Intel: Destination <- Source(s)
  - Windows. Think algebra or C: y = 2x + 1;
  - mov ebp, esp
  - add esp, 0x14 ; (esp = esp + 0x14)
- AT&T: Source(s) -> Destination
  - *nix/GNU. Think elementary school: 1 + 2 = 3
  - mov %esp, %ebp
  - add $0x14,%esp
  - So registers get a % prefix and immediates get a $
- Important to know both, so you can read documents in either format
  - *We will use Intel syntax*

Intel vs AT&T Syntax – Cont.

- IMO the hardest-to-read difference is for r/m32 values
- For intel it's expressed as

  `[base + index*scale + disp]`
- For AT&T it's expressed as

  `disp(base, index, scale)`
- Examples:
  - call DWORD PTR [ebx+esi*4-0xe8]
  - call *-0xe8(%ebx,%esi,4)

  - mov eax, DWORD PTR [ebp+0x8]
  - mov 0x8(%ebp), %eax

  - lea eax, [ebx-0xe8]
  - lea -0xe8(%ebx), %eax

Intel vs AT&T Syntax – Cont.

- For instructions which can operate on different sizes, the mnemonic will have an indicator of the size.
  - movb - operates on bytes
  - mov/movw - operates on word (2 bytes)
  - movl - operates on “long” (dword) (4 bytes)
- Intel does indicate size with things like “mov dword ptr [eax], but it’s just not in the actual mnemonic of the instruction

SUMMARY

- Learned about the basic hardware registers and how they're used
- Learned about how the stack is used
- Saw how C code translates to assembly
- Learned basic usage of compilers, disassemblers, and debuggers so that assembly can easily be explored
- Learned about Intel vs AT&T asm syntax

References

- Open Security Training, Introductory Intel x86: (Architecture, Assembly, Applications, & Alliteration) by Xeno Kovah, http://www.opensecuritytraining.info/IntroX86.html
- Professional Assembly Language by Blum