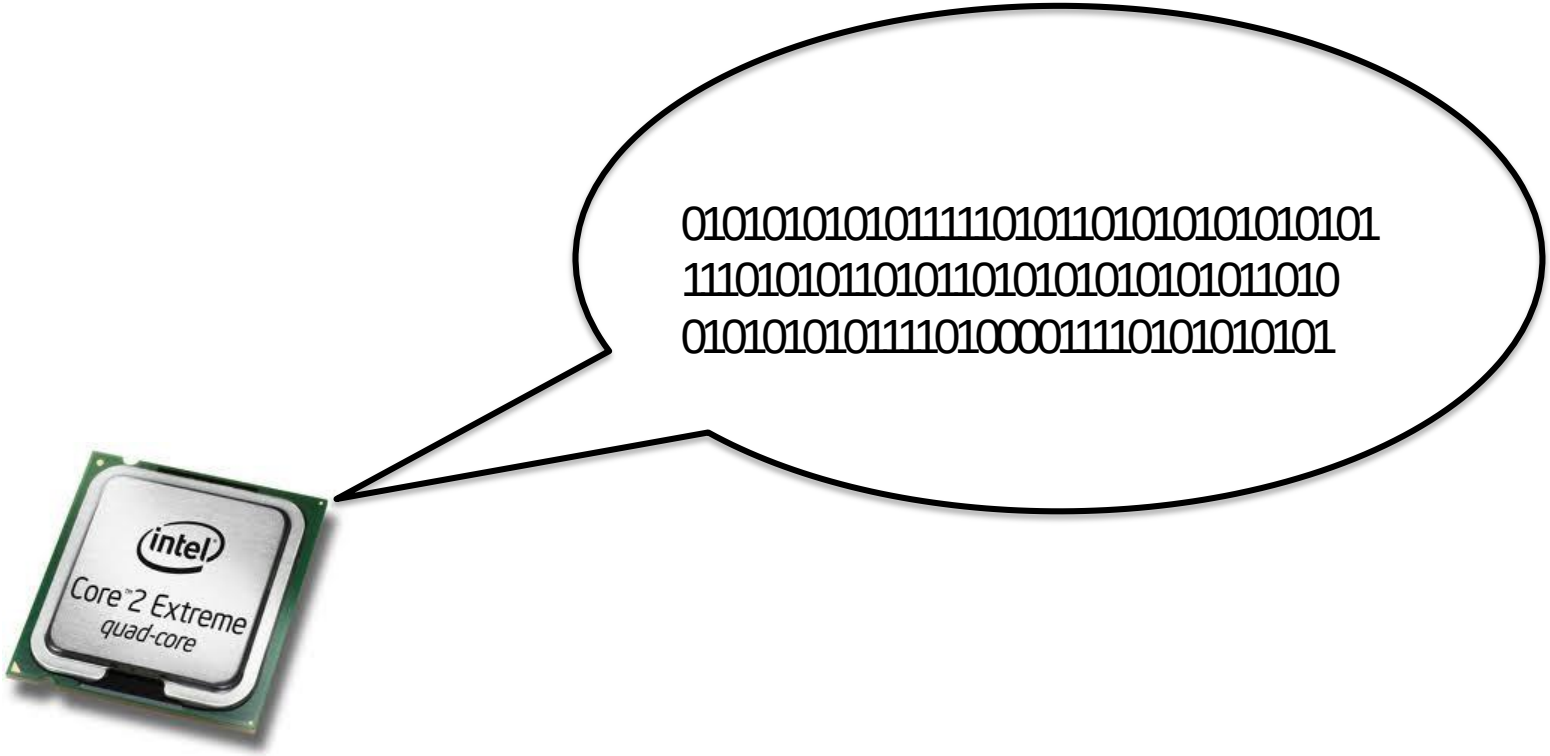
Assembly Language
Linux 86 Assembly part 1

1. What is Assembly Language?

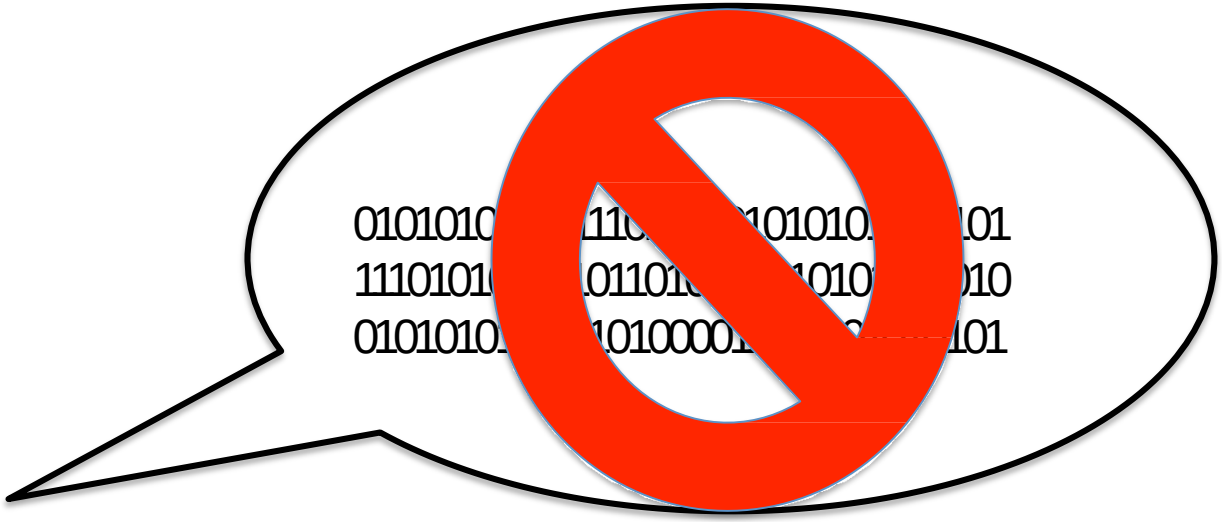
What is Assembly Language?

- Low--level programming language
- Communicate with microprocessor
- Specific to the processor family
- An almost one--one correspondence with machine code

I only speak binary!



Humans cannot speak binary



Assembly Language

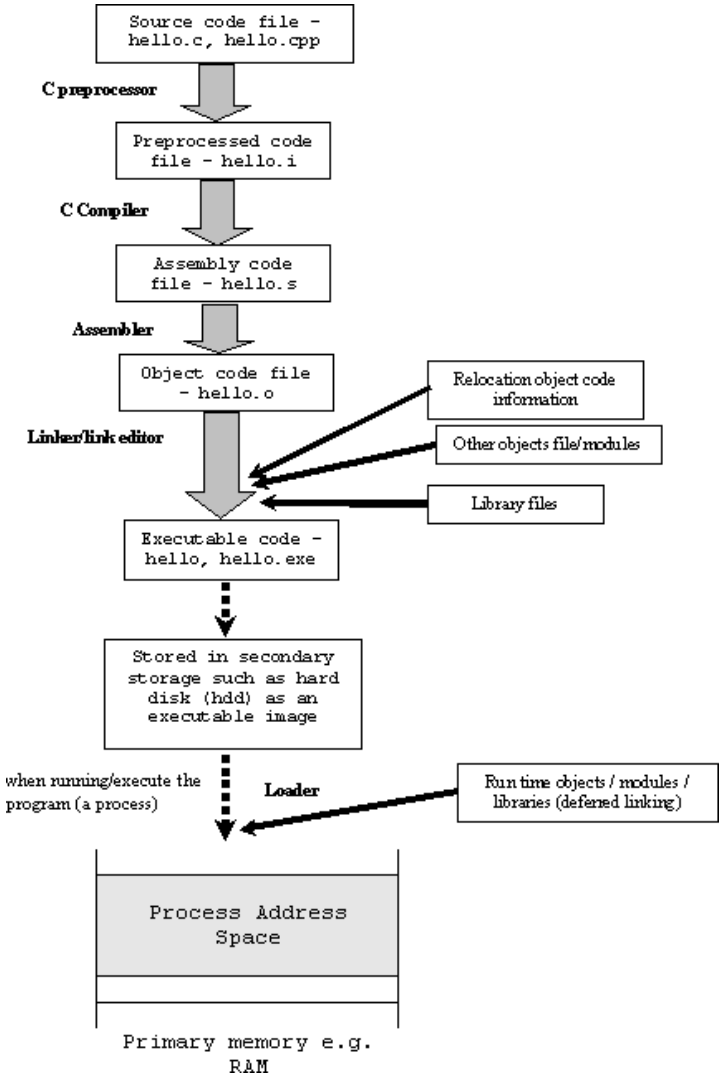
mov eax, ebx
xor eax, eax
add eax, 0xff

Assembler +Linker

Translator

010110100101
111010101010
101010101010

Correlation with HLLs



Different Processors – Different Assembly Language

- Intel
- ARM
- MIPS

Intel Architecture

- IA--32

- IA--64

SecurityTube Linux Assembly Expert 32

- IA--32 Assembly on the Linux OS

Why IA--32?

- Large number of machines out there still running IA--32
- Logical progression to IA--64
- Shellcoding, Encoders, Decoders, Packers etc. implementation difference

Exercise 1.2: Understanding your CPU

- Find CPU details on the Ubuntu System
- How do you know if you are on a 32/64 bit CPU?
- How do you know your CPUs additional capabilities such as FPU, MMX, SSE, SSE2 etc.

Module 1: 32--Bit ASM on Linux

Exercise 1.1
Topic: What is Assembly Language?

Exercise 1.1: Lab Setup

- Ubuntu 12.04 LTS 32--bit Edition

- Installed in Virtualbox/Vmware

Ubuntu

- 32--bit Ubuntu Desktop Edition 12.10 used for the course

http://www.ubuntu.com/download/desktop

•Install Virtualbox

https://www.virtualbox.org

/

Installation

- Nasm

Module 1: 32--Bit ASM on Linux

Exercise 1.2
Topic: What is Assembly Language?

Exercise 1.2: Understanding your CPU

- Find CPU details on the Ubuntu System
- How do you know if you are on a 32/64 bit CPU?
- How do you know your CPUs additional capabilities such as FPU, MMX, SSE, SSE2 etc.

Find CPU Details

```
securitytube@securitytube-VirtualBox:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 37
Stepping:              5
CPU MHz:               2534.821
BogoMIPS:              5069.64
L1d cache:             32K
L1d cache:             32K
L2d cache:             6144K
securitytube@securitytube-VirtualBox:~$
```

/proc/cpuinfo

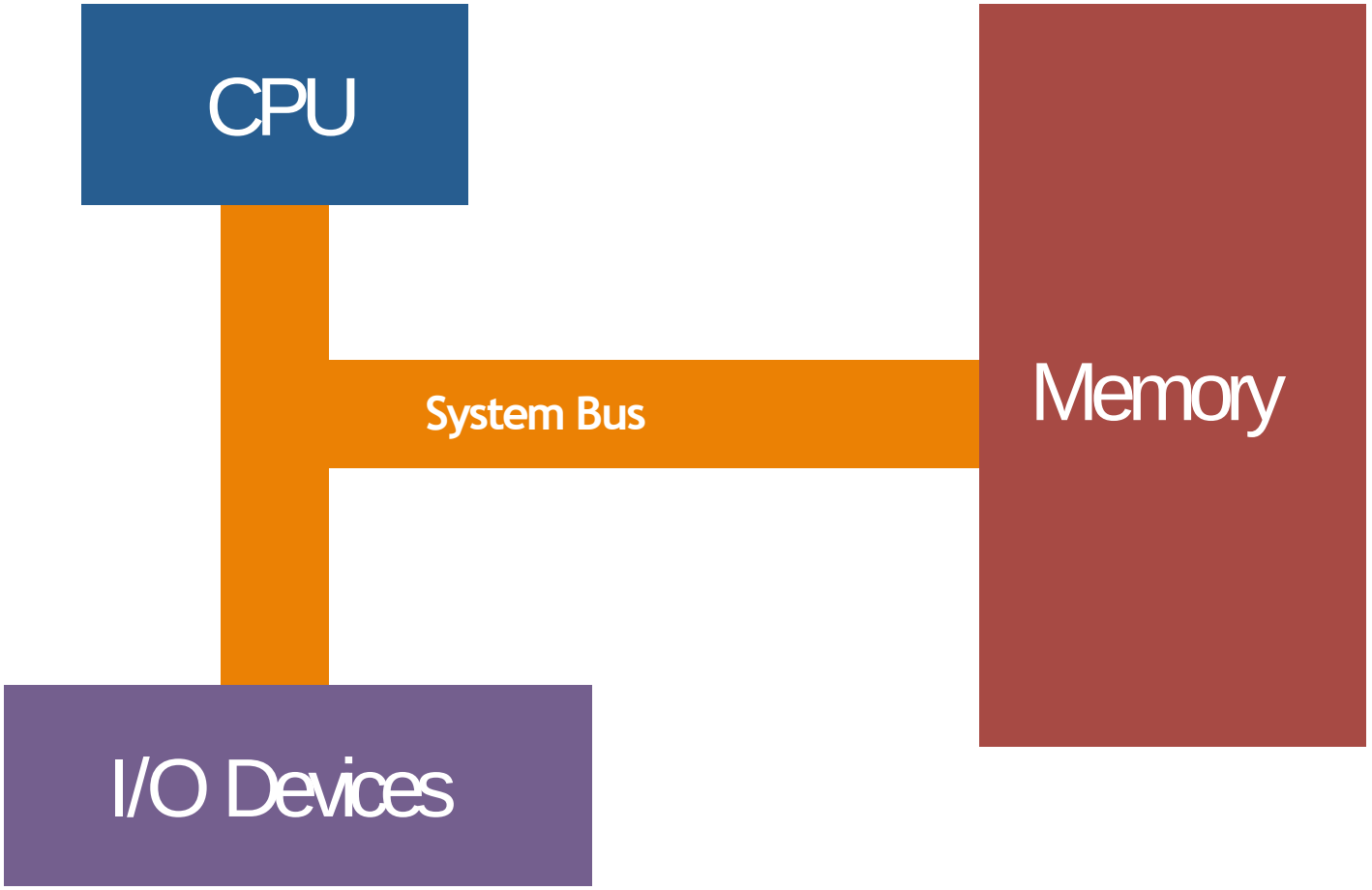
```
securitytube@securitytube-VirtualBox:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i5 CPU       M 540  @ 2.53GHz
stepping        : 5
cpu MHz         : 2534.821
cache size      : 6144 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 syscall nx rdtscp lm constant_tsc up pni monitor ssse3 lahf_lm
bogomips        : 5069.64
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

securitytube@securitytube-VirtualBox:~$
```

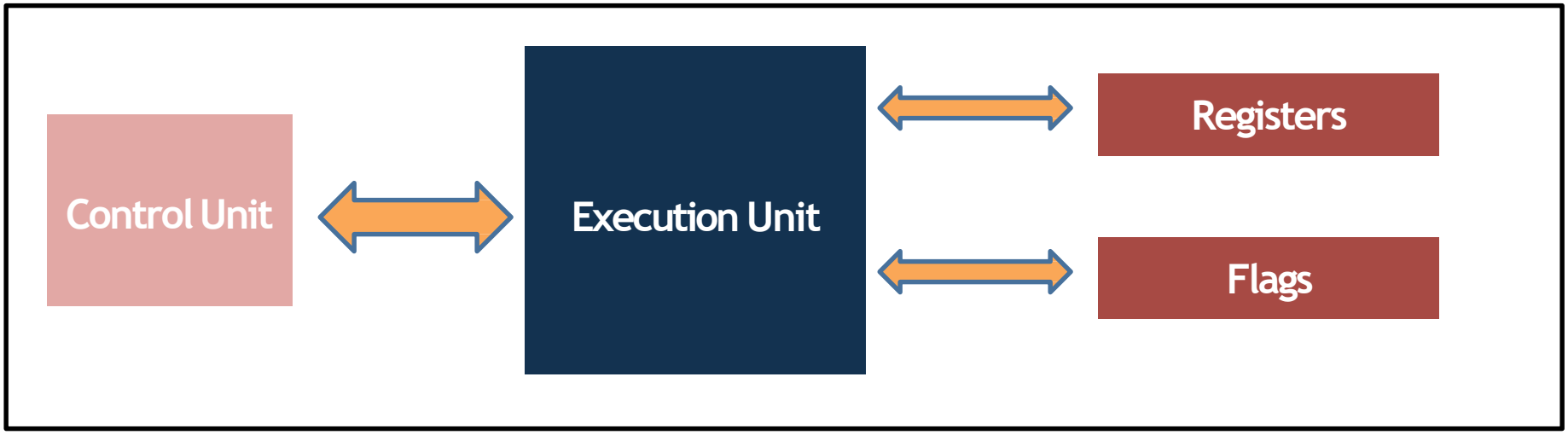
Module 1: 32--Bit ASM on Linux

2. IA--32 Architecture

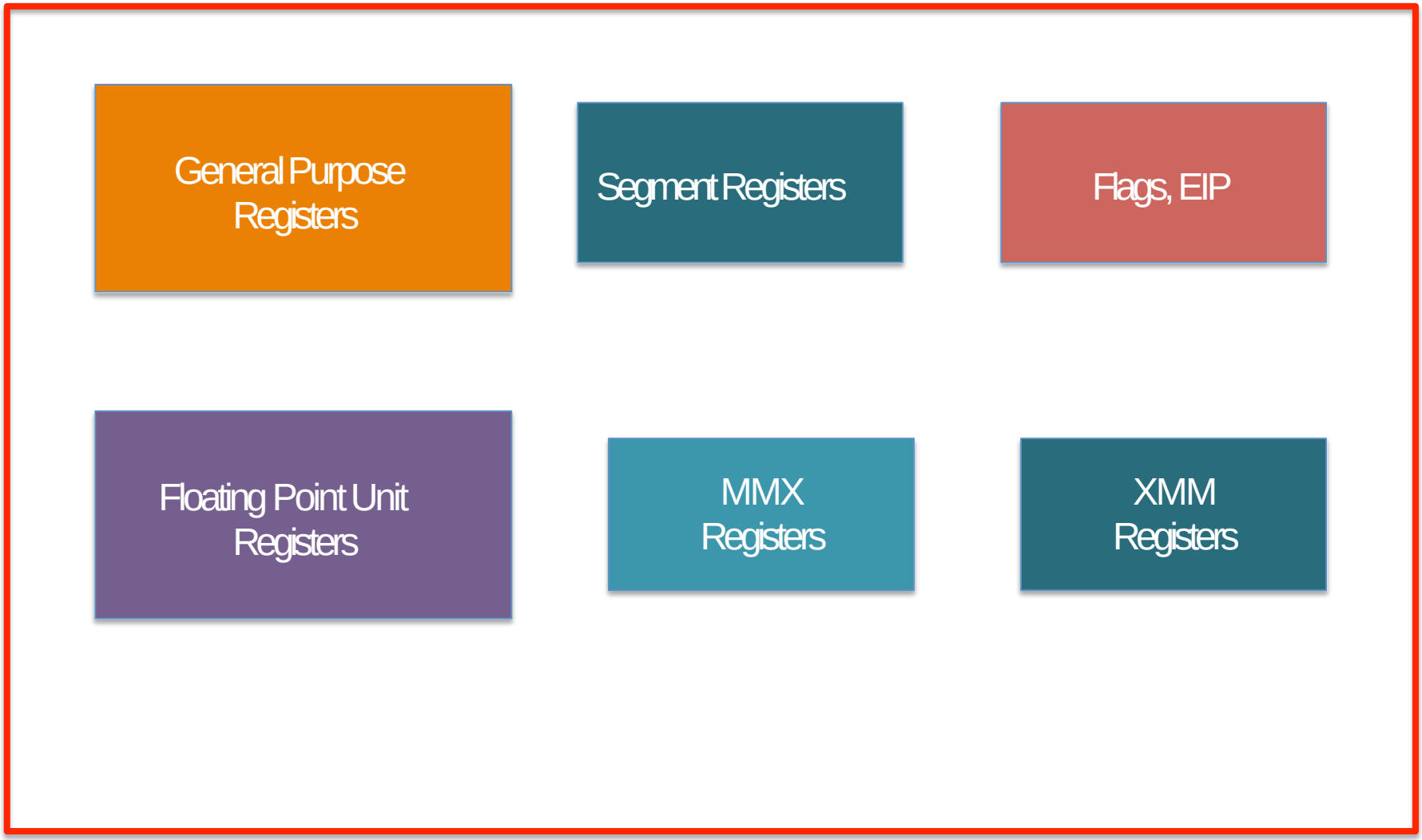
System Organization Basics



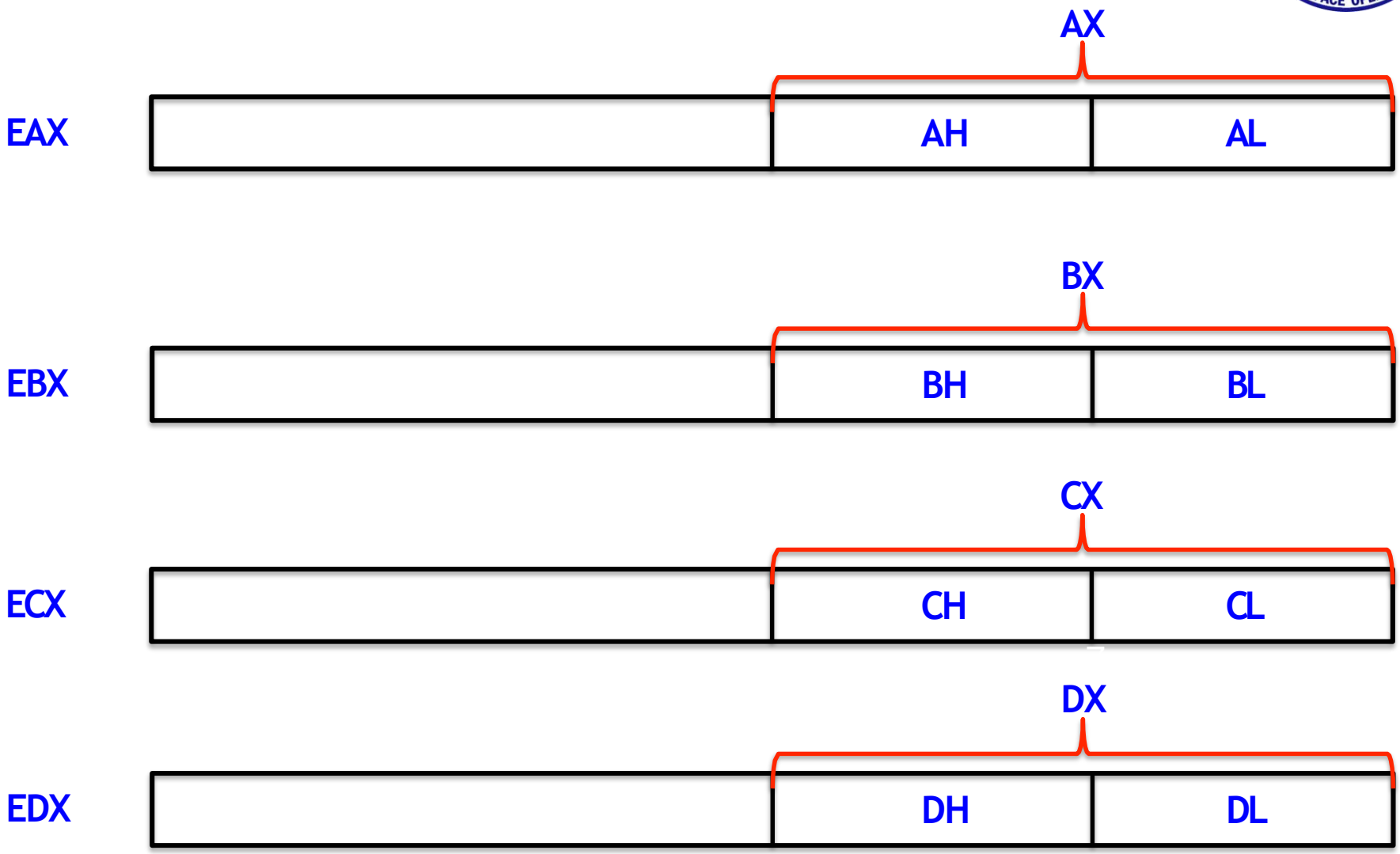
CPU



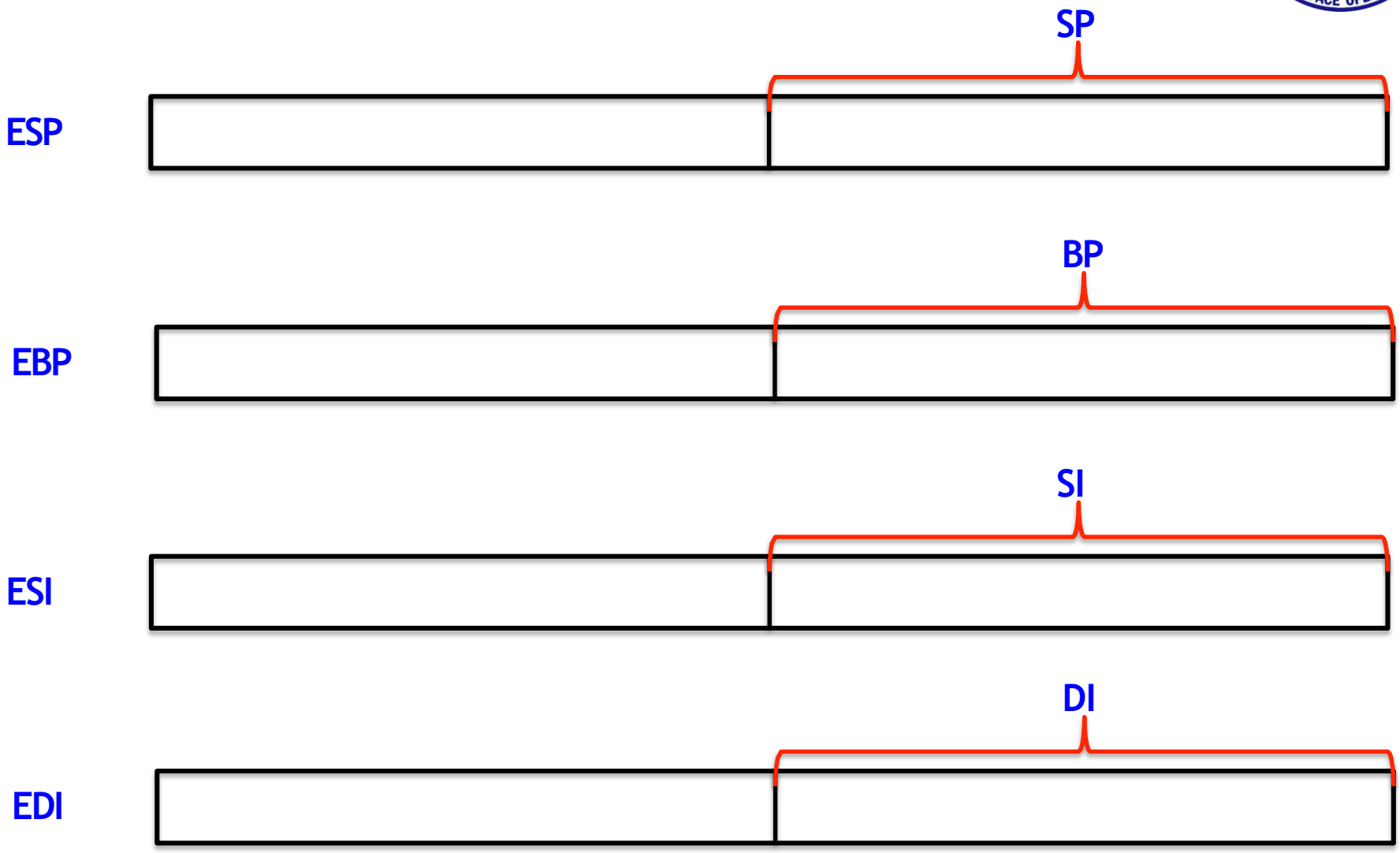
IA--32 Registers (Logical Diagram)



General Purpose Registers



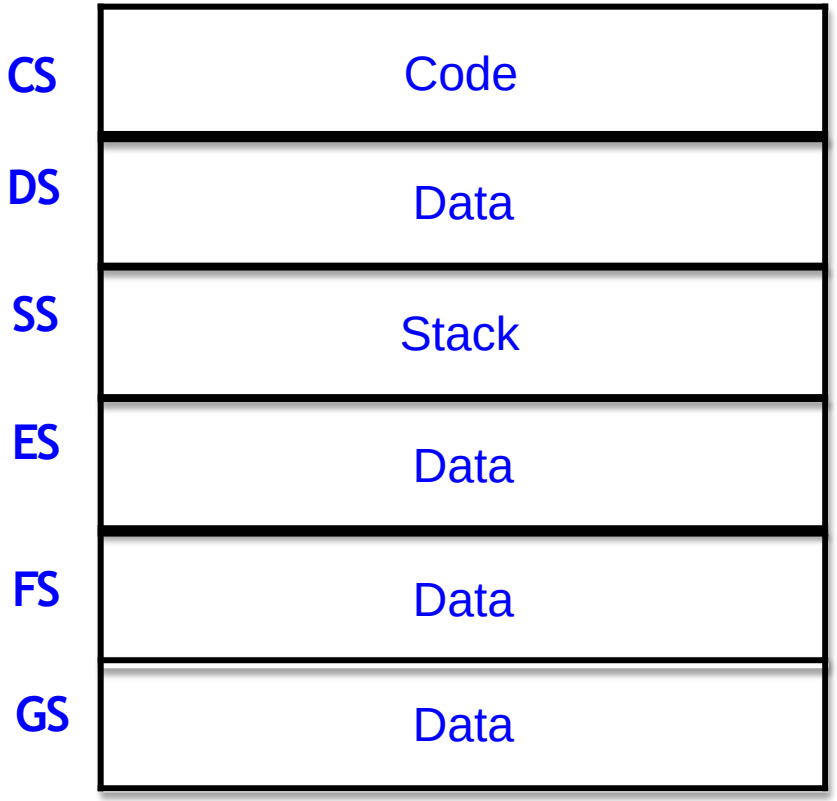
General Purpose Registers



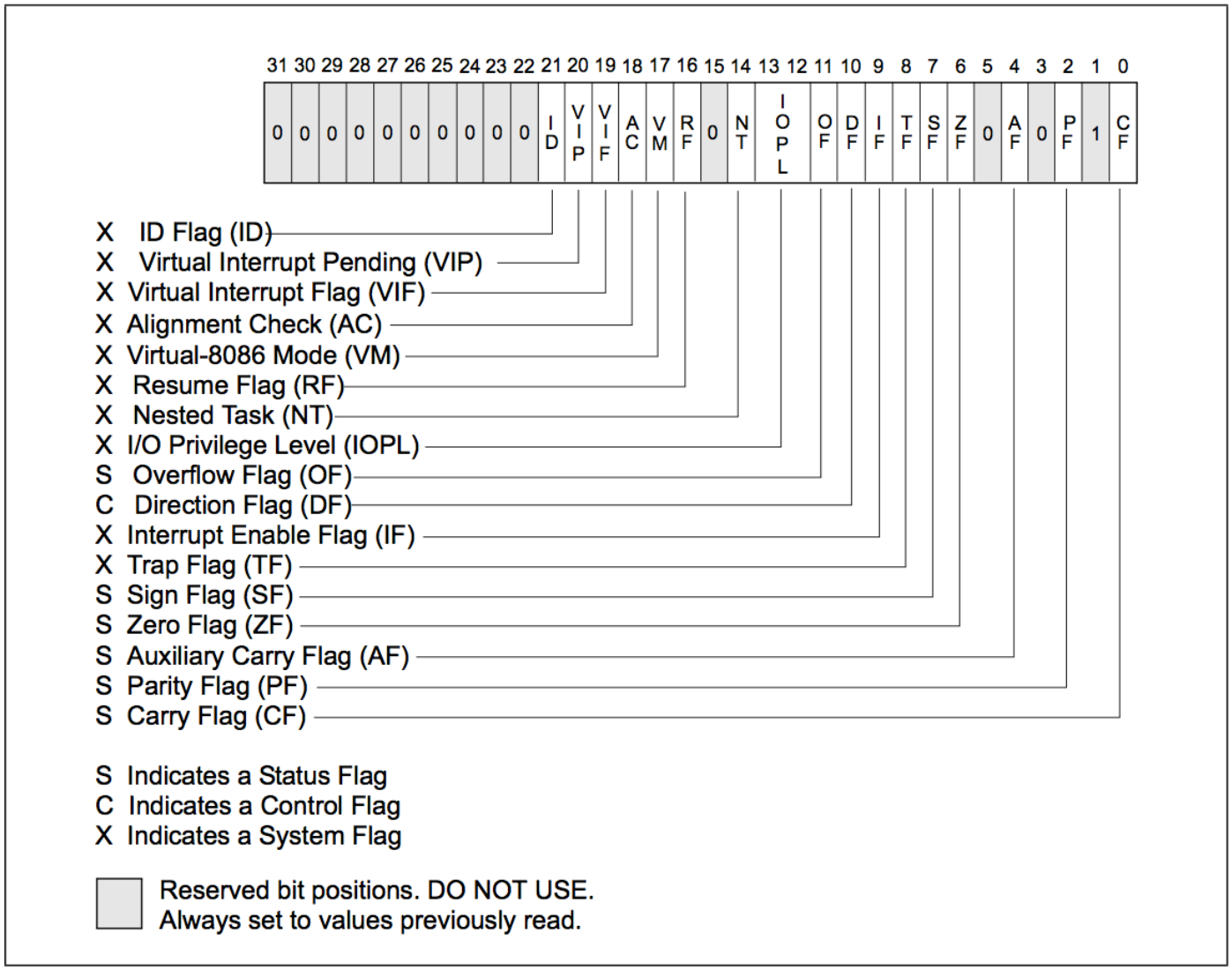
GPR Common Functionality

EAX	Accumulator Register – used for storing operands and result data
EBX	Base Register – Pointer to Data
ECX	Counter Register – Loop operations
EDX	Data Register – I/O Pointer
ESI	EDI Data Pointer Registers for memory
ESP	operations Stack Pointer Register
EBP	Stack Data Pointer Register

Segment Registers



EFLAGS Register

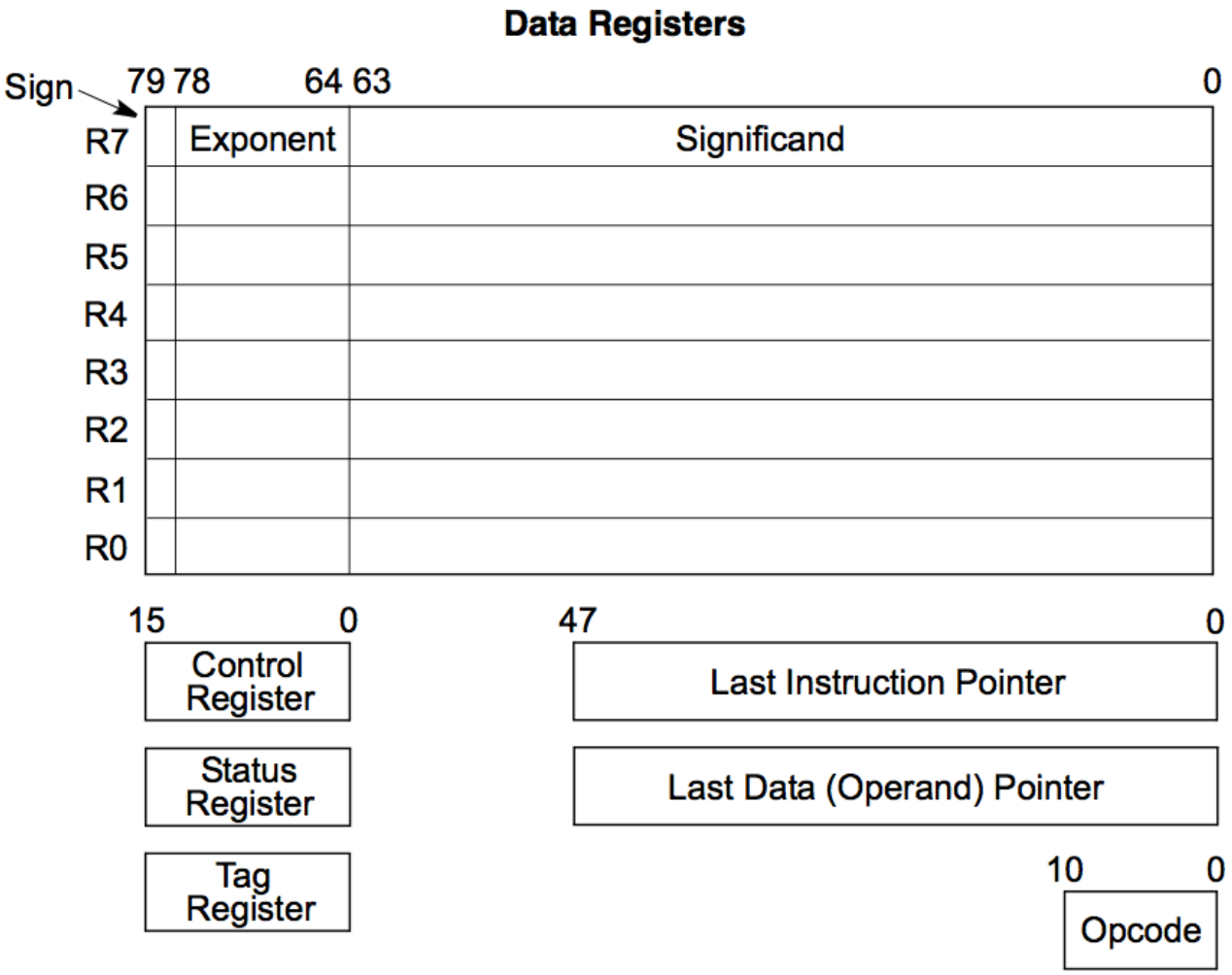


EIP

- Instruction Pointer
- Holy grail for Shellcoding, Exploit Research etc.

Floating Point Unit (FPU) or x87

ST(0) to ST(7)



Reference: Intel Manual

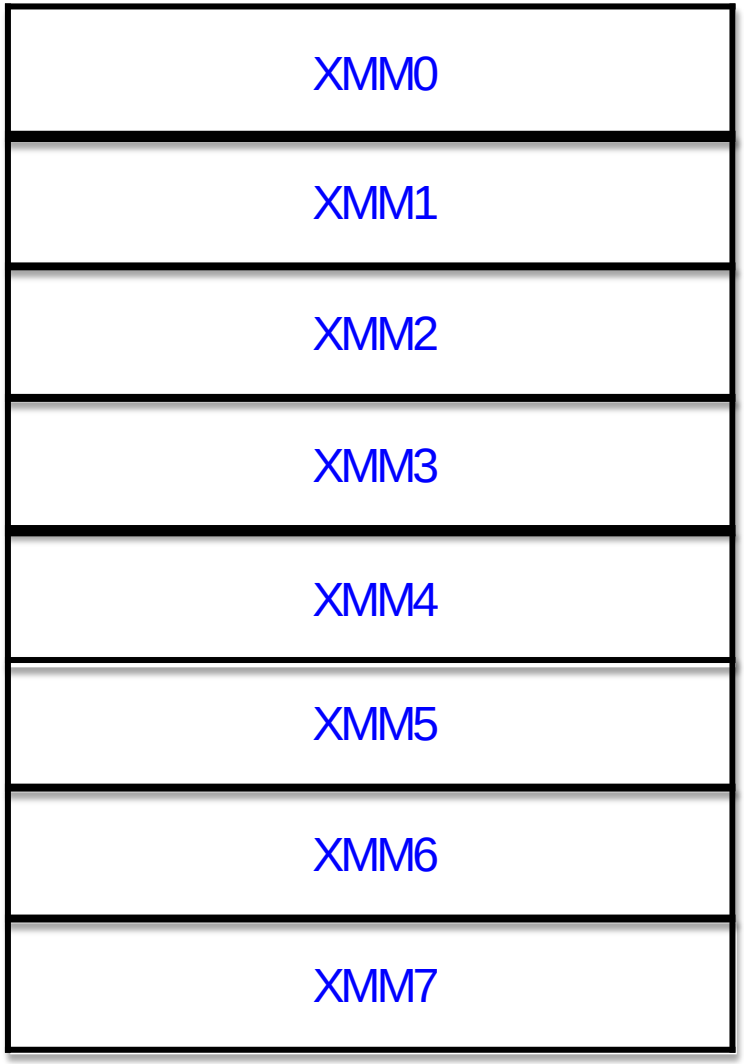
SIMD

- Single Instruction Multiple Data
- Extensions
  - MMX
  - SSE
  - SSE2
  - SSE3
- Uses MMX and XMM Registers

MMX

ST(0)		MM0
ST(1)		MM1
ST(2)		MM2
ST(3)		MM3
ST(4)		MM4
ST(5)		MM5
ST(6)		MM6
ST(7)		MM7


SHADOW WARRIORS
1
90TH CYBERSPACE OPERATIONS SQ


XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7

Exercise 1.2.1: Lab Setup

- Inspect the General Purpose, Segment, EFLAGS, FPU, MMX, XMM etc. registers on your Ubuntu system

Exercise 1.2.1: Lab Setup

- Inspect the General Purpose, Segment, EFLAGS, FPU, MMX, XMM etc. registers on your Ubuntu system

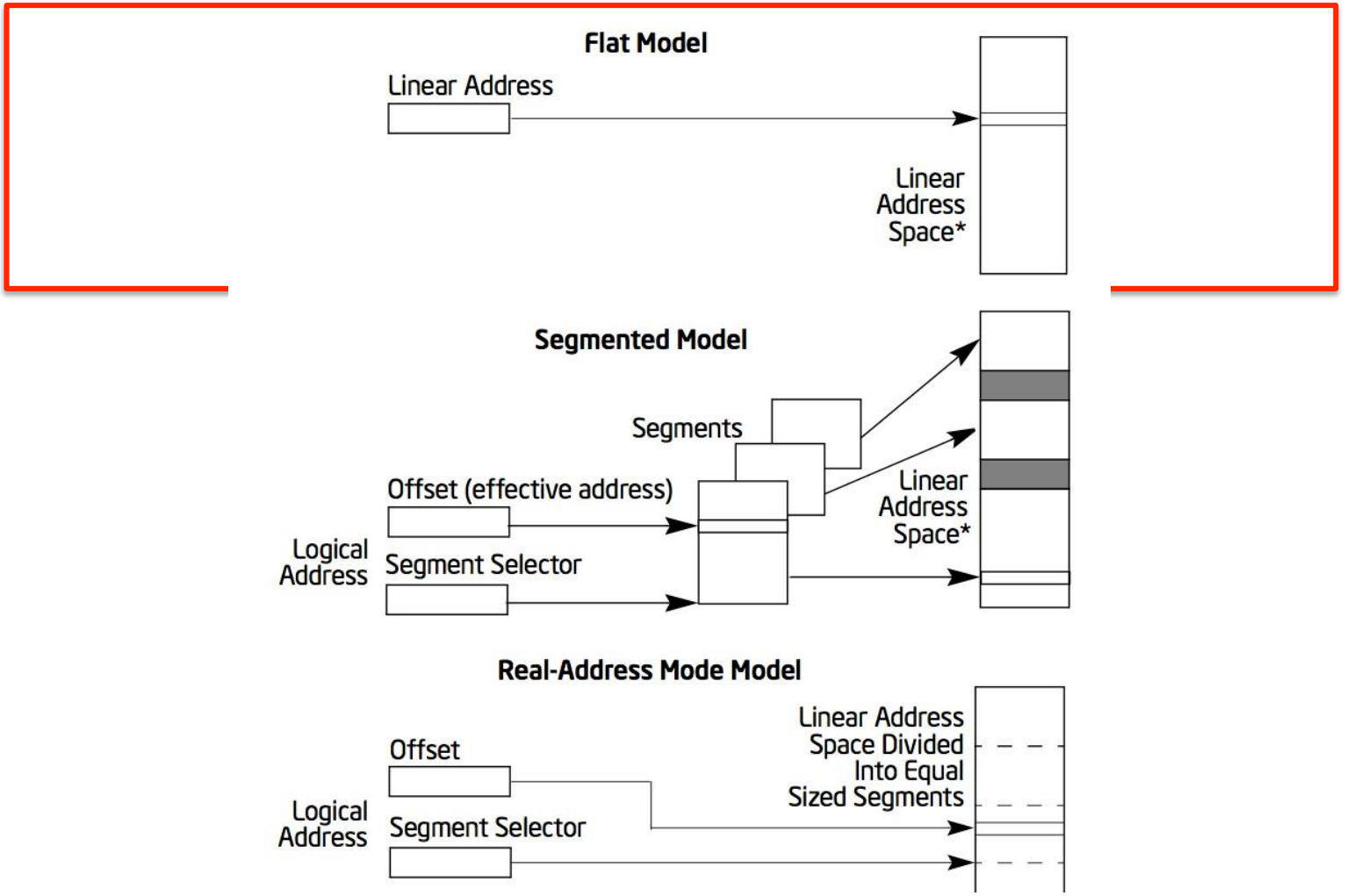
Module 1: 32--Bit ASM on Linux

3. CPU Modes and Memory Management

CPU Modes for IA--32

- Real Mode
  - At power up or reset
  - Can only access 1 MB memory
  - No memory protection
  - Privilege Levels (Kernel vs User space) not possible
- Protected Mode
  - Up to 4GB memory
  - memory protection / privilege level / multi--tasking
  - Supports Virtual--8086 mode
- System Management Mode
  - Used for power management tasks

Memory Models

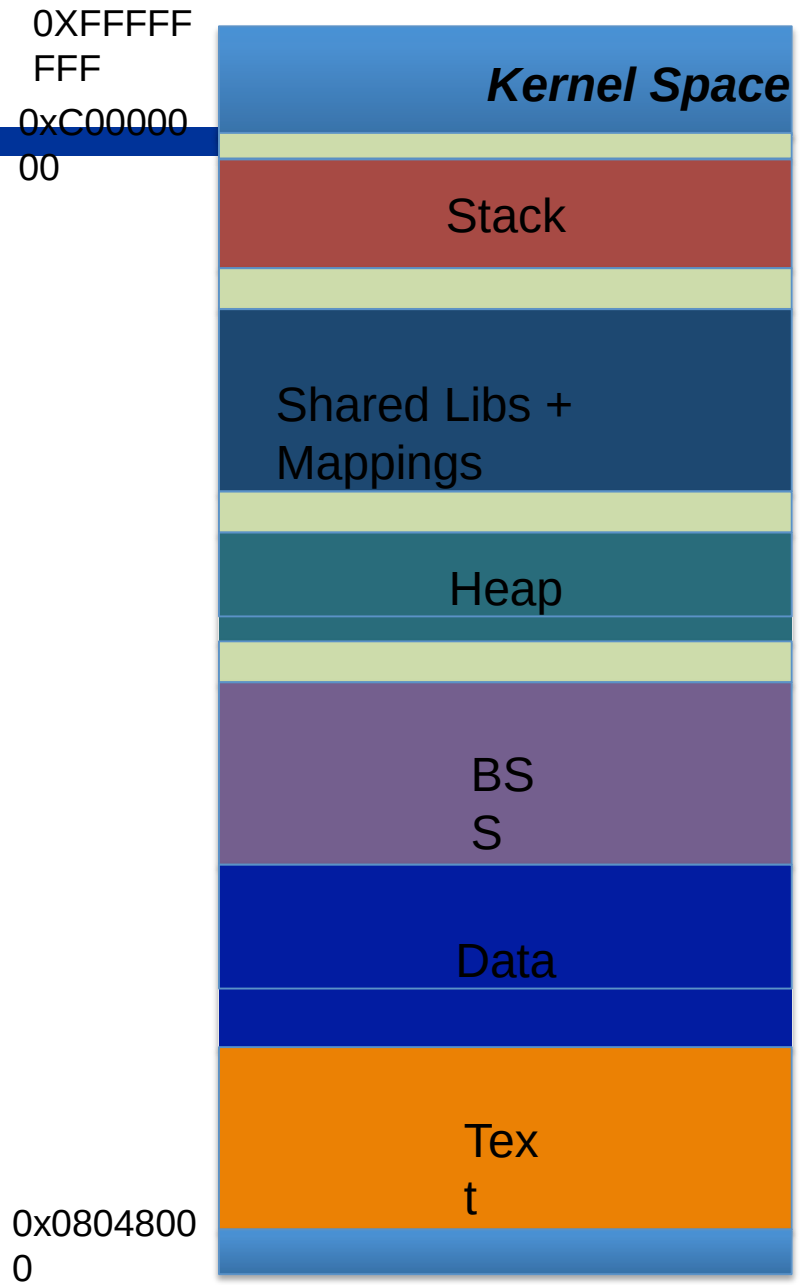


Source: Intel manuals

Linux: CPU Mode and Memory Model

- 32--Bit Linux uses:
  - Protected Mode
  - Flat Memory model
  - 4GB Addressable space => 2^{32}
  - Memory Protection
  - Privilege Levels of Code
  - Segment registers point to segment descriptors
    - GDT / LDT / IDT (Global / Local / Interrupt)

Virtual Memory Model


0XFFFFFFFF
0xC0000000
Kernel Space
Stack
Function Args + Local Vars
Shared Libs + Mappings
Heap
Dynamic Memory
BSS
Uninitialized Data
Data
Initialized Data
Text
Program Code
0x08048000
0x00000000

View Process Organization

- /Proc
  - /proc/pid/maps
- pmap
- Attach and view using GDB

cat /proc/pid/maps


```
securitytube@securitytube-VirtualBox:~$ cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 262170     /bin/cat
08053000-08054000 r--p 0000a000 08:01 262170     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 262170     /bin/cat
0917e000-0919f000 rw-p 00000000 00:00 0          [heap]
b73cd000-b75cd000 r--p 00000000 08:01 140017     /usr/lib/locale/locale-archive
b75cd000-b75ce000 rw-p 00000000 00:00 0
b75ce000-b776d000 r-xp 00000000 08:01 265843     /lib/i386-linux-gnu/libc-2.15.so
b776d000-b776f000 r--p 0019f000 08:01 265843     /lib/i386-linux-gnu/libc-2.15.so
b776f000-b7770000 rw-p 001a1000 08:01 265843     /lib/i386-linux-gnu/libc-2.15.so
b7770000-b7773000 rw-p 00000000 00:00 0
b7782000-b7783000 r--p 005db000 08:01 140017     /usr/lib/locale/locale-archive
b7783000-b7785000 rw-p 00000000 00:00 0
b7785000-b7786000 r-xp 00000000 00:00 0          [vdso]
b7786000-b77a6000 r-xp 00000000 08:01 265823     /lib/i386-linux-gnu/ld-2.15.so
b77a6000-b77a7000 r--p 0001f000 08:01 265823     /lib/i386-linux-gnu/ld-2.15.so
b77a7000-b77a8000 rw-p 00020000 08:01 265823     /lib/i386-linux-gnu/ld-2.15.so
bfdfc000-bfe1d000 rw-p 00000000 00:00 0          [stack]
securitytube@securitytube-VirtualBox:~$
```

What does all this mean?

```
08048000-08053000 r-xp 00000000 08:01 262170     /bin/cat
```

Start and End Address of the section

Offset in file for memory mapped files. 0 otherwise.

Inode number

File Path

Major – Minor device number of device from where the file was loaded

Permissions on the section:

- r = readable
- w = writable
- x = executable
- p = private not shared
- s = shared

Process Map within GDB

```
(gdb) info proc mappings
process 2837
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8124000    0xdc000        0x0 /bin/bash
         0x8124000  0x8125000     0x1000    0xdb000 /bin/bash
         0x8125000  0x812a000     0x5000    0xdc000 /bin/bash
         0x812a000  0x812f000     0x5000        0x0 [heap]
        0xb7e00000 0xb7e01000     0x1000        0x0
        0xb7e01000 0xb7fa0000   0x19f000        0x0 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fa0000 0xb7fa2000     0x2000   0x19f000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fa2000 0xb7fa3000     0x1000   0x1a1000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fa3000 0xb7fa7000     0x4000        0x0
        0xb7fa7000 0xb7faa000     0x3000        0x0 /lib/i386-linux-gnu/libdl-2.15.so
        0xb7faa000 0xb7fab000     0x1000     0x2000 /lib/i386-linux-gnu/libdl-2.15.so
        0xb7fab000 0xb7fac000     0x1000     0x3000 /lib/i386-linux-gnu/libdl-2.15.so
        0xb7fac000 0xb7fc8000    0x1c000        0x0 /lib/i386-linux-gnu/libtinfo.so.5.9
        0xb7fc8000 0xb7fca000     0x2000    0x1b000 /lib/i386-linux-gnu/libtinfo.so.5.9
        0xb7fca000 0xb7fcb000     0x1000    0x1d000 /lib/i386-linux-gnu/libtinfo.so.5.9
        0xb7fdb000 0xb7fdd000     0x2000        0x0
        0xb7fdd000 0xb7fde000     0x1000        0x0 [vdso]
        0xb7fde000 0xb7ffe000    0x20000        0x0 /lib/i386-linux-gnu/ld-2.15.so
        0xb7ffe000 0xb7fff000     0x1000    0x1f000 /lib/i386-linux-gnu/ld-2.15.so
        0xb7fff000 0xb8000000     0x1000    0x20000 /lib/i386-linux-gnu/ld-2.15.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
(gdb)
```

Module 1: 32--Bit ASM on Linux

4. Hello World

IA--32 Instruction Set

- General Purpose Instructions
- x87 FPU Instructions
- MMX / SSE / SSE2/ SSE3 / SSE4 Instructions
- Other Instruction Set extensions

Programming in Assembly

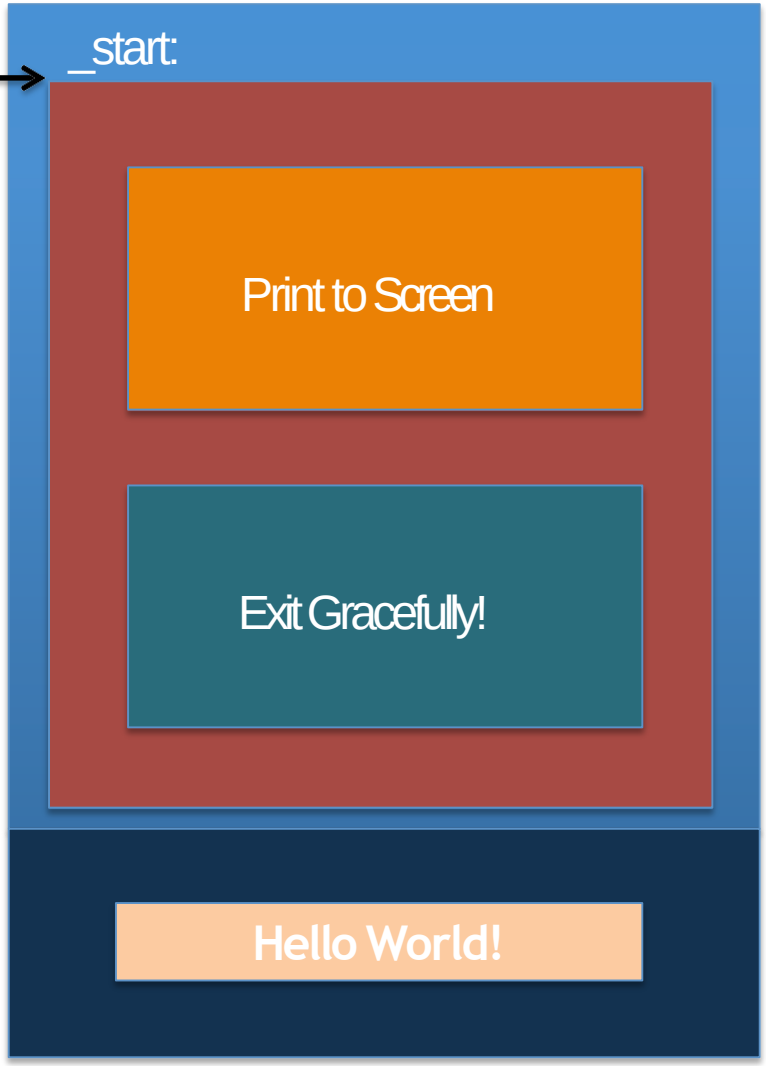
- NASM + LD for assembling and linking
- Executable in ELF format

NASM Documentation:

http://nasm.us/

Hello World!

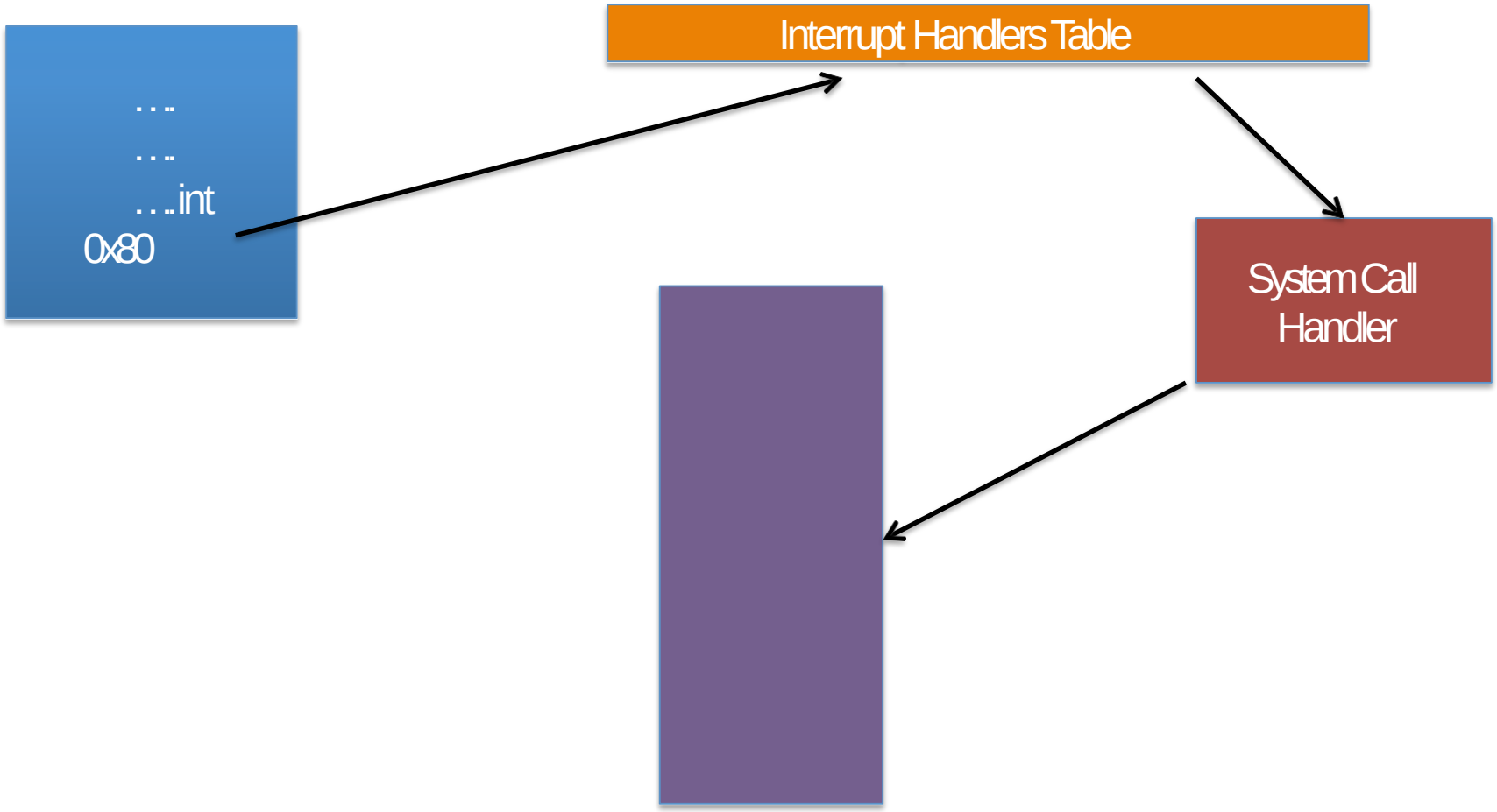
Entry Point of Program?



Why System Calls?

- Leverage OS for tasks
- Imagine if you had to write code from scratch to:
  - write to disk
  - print on screen
  - …
- System Calls provide a simple interface for user space programs to the Kernel

How do System calls work?



IA--32 Mechanism to invoke System Call

- int 0x80

- SYSENTER

Modern implementations using VDSO [Virtual Dynamic Shared Object]

http://articles.manugarg.com/systemcallinlinux2_6.html

Where are these system calls defined?

```
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13
#define __NR_mknod               14
#define __NR_chmod               15
#define __NR_lchown              16
#define __NR_break               17
#define __NR_oldstat             18
#define __NR_lseek               19
#define __NR_getpid              20
#define __NR_mount               21
#define __NR_umount              22
#define __NR_setuid              23
#define __NR_getuid              24
#define __NR_stime               25
"/usr/include/i386-linux-gnu/asm/unistd_32.h"
```

write()

```
WRITE(2)                                                Linux Programmer's Manual

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer pointed buf to th
```

exit

```
_EXIT(2)                                                    Linux Pro

NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);
```

Invoking System Call with 0x80

EAX	System Call Number	Return Value in EAX
EBX	1st Argument	
ECX	2nd Argument	
EDX	3rd Argument	
ESI	4th Argument	
EDI	5th Argument	

Calling Write

```
WRITE(2)                                          Linux Programmer's Manual

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer pointed buf to th
```

EAX=system call number

EBX=STDOUT

ECX=Pointer to "HelloWorld"

EDX=Length of "HelloWorld"

Calling Exit

```
_EXIT(2)                                                    Linux Pro

NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);
```

EAX=system call number

EBX=Status Code

Exercise 1.4.1: GDB

- Use GDB to step through the Hello World program and observe:
  - CPU Registers
  - Memory Location
  - …

Exercise 1.4.1: GDB

- Use GDB to step through the Hello World program and observe:
  - CPU Registers
  - Memory Location
  - …