x86 Crash Course

With a focus on Linux and a glance to x86_64

Many contributors from the NECST Laboratory, DEIB

Politecnico di Milano

October 9, 2017

Contributors as of October 9, 2017

- Main contributor: Andrea Mambretti (wrote the first version)
- Mario Polino
- Stefano Zanero
- Federico Maggi

Instruction Set Architecture (ISA)

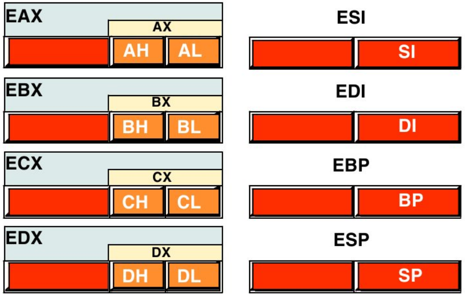
Specification of Computer Architecture related to programming.
Instructions, registers, memory architecture, interrupt, and etc.

- **x86**, x86_64, IA-32: PCs, Servers, High-Performance, ...
- **ARM**: Mobile Devices
- **MIPS**: Routers
- **PowerPC**: Embedded and high-performance processors
- **m68k**: Very old
- **and more**

(1) How does the IA-32 look like?

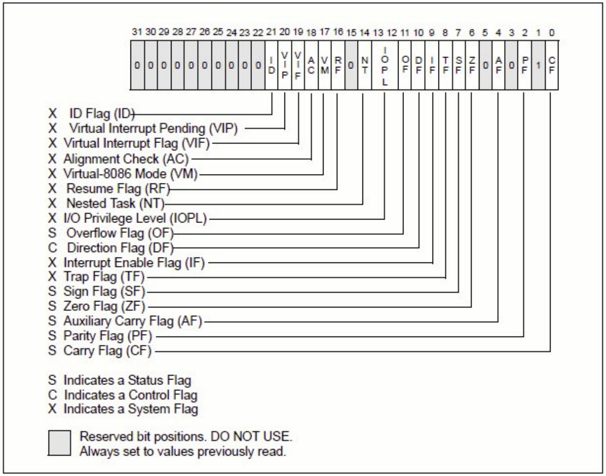
- The processor has 32-bits internal registers to manage and execute operations on data
- **EAX**, **EBX**, **ECX**, **EDX** are for general purposes
- **EBP** (BP = base pointer) and **ESP** (SP = stack pointer) are the stack bounds
- **EDI** and **ESI** are extra registers used as source and dest for special instructions (e.g strcpy)
- **EIP** (IP = instruction pointer) is the register that contains the address of the next instruction

(2) How does AI look like?



- The register naming system is from IA-16 where the registers were called AX, BX, CX, DX.
- the “E” indicates extended (32 bits). Without it, we consider the corresponding 16-bits register
- **Memorandum:** although we are in IA-32
  - byte (8 bits)
  - word (2 bytes)
  - dword (double word, 4 bytes)
  - qword (quadruple word, 8 bytes)

(1) What about EFLAGS?



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

(2) What about EFLAGS?

- It's another 32-bits register
- Only 8 bits out of 32 are of interest for us. The others are either for the kernel mode function or are of little interest for programmers
- These 8 bits are called flags. We consider them singularly. They are boolean (true/false)
- They represent overflow, direction, interrupt disable, sign, zero, auxiliary carry, parity and carry flags
- Since they represent information about the instruction last executed, they change at every execution step. They are VERY important for the control flow of the program

Assembly

- Low-Level Language
- Directly mapped with binary code
- Custom per Architecture (**x86**, PowerPC, ARM ...)

Syntax

- In the assembly world we can find two main syntaxes: AT&T and Intel
- AT&T syntax is used by all UNIX programs (e.g., `gdb` by default)
- Intel syntax is used by Microsoft programs (IDApro and others)

(1) Differences in the notation

- Consider the following operation:

  "move the value 0 to EAX"
- AT&T:

  ```
  mov $0x0,%eax
  ```
- Intel:

  ```
  mov eax, 0h
  ```
- Main differences:
  - In AT&T syntax the destination is the second operand, and vice-versa in Intel syntax
  - In AT&T syntax registers are denoted with % and immediates (i.e., costants) with $. In the Intel syntax these tokens are not used.

(2) Differences in pointer de-referencing

- Consider this new operation:

  "move the value 0 to the address contained in EBX+4"
- AT&T:

```
mov $0x0,0x4(%ebx)
```
- Intel:

```
mov [ebx+4h],0h
```
- Main differences:
  - Parentheses: round vs. square
  - Offsets: out and before vs. inside with a '+' sign

(1) Basic instructions overview

- Every processor has a large instruction set (see Intel manual[1])
- A subset of the whole instruction set is usually processor dependent
- We will focus on the subset of instructions that is common among the processors
- We will use the Intel syntax as it is the same syntax used in IDApro by default
- Very good reference: `http://ref.x86asm.net/geek32.html`

[1] `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

(2) Basic Instruction MOV

- MOV **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: Every combination except memloc to memloc: It is invalid in every instruction!
- Move value from source to destination. There are tons of different versions, depending on the operands size (byte, word, dword), immediate to reg, immediate to memory, etc.
- Examples

MOV eax, ebx	MOV eax, FFFFFFFFh	MOV ax, bx
MOV [eax],ecx	MOV [eax],[ecx] NO!!!	MOV al, FFh

(3) Basic Instruction ADD

- ADD **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination has to be at least as large as the source.
- Semantic: destination = destination + source
- Examples

ADD esp, 44h	ADD eax, ebx	ADD al, dh
ADD edx, cx	ADD [eax],[ecx] NO!!!	ADD [eax],1h

(4) Basic Instruction SUB

- SUB **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination has to be at least as large as the source.
- Semantic: destination = destination - source
- Examples

SUB esp, 33h	SUB eax, ebx	SUB al, dh
SUB edx, cx	SUB [eax],[ecx] NO!!!	SUB [eax],1h

(5) Basic Instruction MUL

- MUL **Operand**
  Operand can be an immediate, a register, a memory location
- Multiplies **Operand** by the value of AL, AX, EAX register, depending on sizeof(Operand)

sizeof(Operand):	1 byte	2 bytes	4 bytes
Other operand	AL	AX	EAX
Most sign. part of result in:	AH	DX	EDX
Least sign. part of result in:	AL	AX	EAX

- Examples

sizeof(Operand):	1 byte	2 bytes	4 bytes
Immediate	MUL 44h	MUL 4455h	MUL 44556677h
Register	MUL cl	MUL dx	MUL ebx

(6) Basic Instruction DIV

- DIV **Operand**
  Operand can be an immediate, a register, a memory location
- Divides the value in the dividend register(s) by Operand

sizeof(Operand):	1 byte	2 bytes	4 bytes
Dividend	AX	DX:AX	EDX:EAX
Remainder	AH	DX	EDX
Quotient	AL	AX	EAX

- Examples

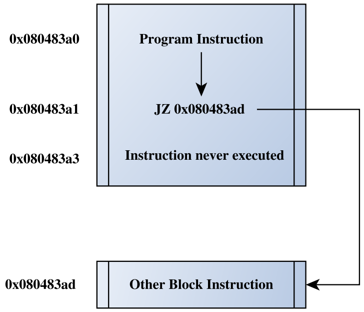
sizeof(Operand):	1 byte	2 bytes	4 bytes
Register	DIV bl	DIV bx	DIV ebx
Immediate	DIV 66h	DIV 6677h	DIV 66778899h

(7) Control-flow Instruction CMP

- CMP **Operand_1**, **Operand_2**
- Operand_1 minus Operand_2 subtraction and sets the flags (ZF,CF,OF, etc.), it doesn't store the result
- Examples

CMP eax, ebx	CMP eax, 44BBCCDDh	CMP al, dh
CMP al, 44h	CMP ax,FFFFh	CMP [eax],4h

(8) Control-flow Instruction JMP

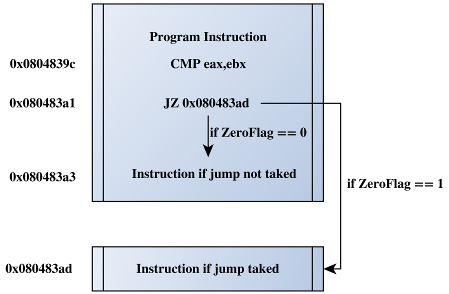


- JMP **address** or **offset**
- Called "unconditional jump": sets the EIP to the address. We say that the execution "jumps to **address**" and it's unconditional because it always happens.
- Using an **offset** the current EIP is incremented (or decreased) by the constant expressed by the **offset** resulting in a relative jump.

(9) Other Control-flow Instructions: JZ, JNZ and so on

- General: J**X** **<u>address</u>** or **<u>offset</u>**
  X $\in \{O, NO, S, NS, E, Z, NE \ldots\}$
- These are called "conditional jumps"
- The execution will go to **address** if and only if the specific flag of the condition is verified.
- For example: JZ jumps if and only if ZF (zero flag) is 1
- **Offset** is used for relative position (as in JMP)

[2]



[2]http://www.unixwiz.net/techtips/x86-jumps.html

(10) Basic Instruction NOP

- NOP
- The meaning of NOP is **No Operation**. Just move to next instruction.
- The opcode is pretty famous and is **0x90**
- Really useful in exploitation (we will see).

(11) Basic Instruction INT

- INT **VALUE**
- **VALUE** is the software interrupt number to generate (0-255)
- Famous values are 21h for call service under Windows and 80 for Linux
- Every OS has its set of interrupt numbers

(12) Syscall/SYSENTER

Nowadays Syscalls are done with specific instructions.

- syscall is used for Linux 64-bit
- SYSENTER is used by Microsoft Windows

How much is x86_64 different from x86?

- The prefix of the registers is R instead of E so we have (**RIP**, **RAX** etc.)
- There are 8 new registers (**R8** to **R15**)
- each of them can be considered at 8, 16, 32, 64 bits (qword) with $X \in \{8..15\}$ we have

bits	8	16	32	64
reg	rXb	rXw	rXd	rX

- for better syntax information look at `http://www.x86-64.org/documentation/assembly.html`

Binary File Formats

- **PE (Portable Executable):** used by Microsoft binary executable.
- **ELF:** common binary format for Unix, Linux, FreeBSD and others
- In both cases, we are interested in how each executable is mapped into memory, rather than how it is organized on disk.

How an executable is mapped to memory in Linux (ELF)

Executable	Description
.bss	This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.comment	This section holds version control information.
.data/.data1	These sections hold initialized data that contribute to the program's memory image
.debug	This section holds information symbolic debugging.
.text	This section holds the "text," or executable instructions, of a program.
.init	This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).
.got	This section holds the global offset table.

How an executable is mapped to memory in Windows (PE)

Executable	**Description**
.text	Contains the executable code
.rdata	Holds read-only data that is globally accessible within the program
.data	Stores global data accessed throughout the program
.idata	stores the import function information;
.edata	stores the export function information;
.pdata	Present only in 64-bit executables and stores execption-handling information
.rsrc	Stores resources needed by the executable
.reloc	Contains information for relocation of library files

A more realistic view of an elf in memory

Low addresses (0x80000000)

`Shared libraries`
`.text`
`.bss`
`heap` (grows $\downarrow$)
`...`
`stack` (grows $\uparrow$)
`env`
`argc`

High addresses (0xbfffffff)

The stack

- Last In First Out (LIFO) data structure: the most recent data placed, or pushed, onto the stack is the first item to be removed, or popped, from it
- A LIFO is ideal for storing transitory data, or data that does not need to be stored for long.
- The stack stores local variables, information related to function calls or used to clean up the stack after a function or procedure returns.
- Remember: the stack grows downward the address space, toward lower addresses.

Stack Management Instruction: PUSH

- Pushes a word onto the stack
- Example: PUSH **immediate** or PUSH **register**
- Pushes the immediate or register value at the top of the stack and decrements the ESP of sizeof(immediate or register)

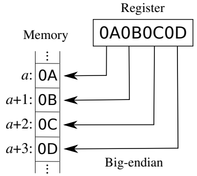
Stack Management Instruction: POP

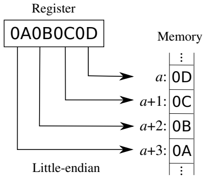
- Pops a word from the stack
- Example: POP **destination**
- Pops the word off of the top of the stack and moves it at destination, then it increases the ESP of sizeof(data popped).

Endianness

The **endianness** is a convention that specifies in which order the bytes of a data word are lined up sequentially in memory.

Big-endian (left) systems are systems in which the *most significant byte* of the word is stored in the *smallest address* given.

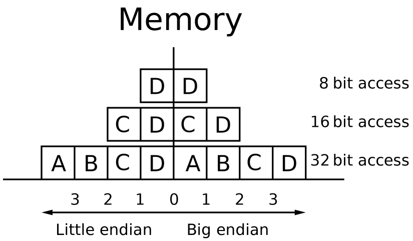


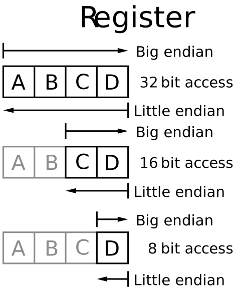


Little-endian (right) systems are those in which the *least significant* byte is stored in the *smallest address*. A-32 is "little endian".

Endianness cont.

The **endianness** is a convention that specifies in which order the bytes of a data word are lined up sequentially in memory.





Functions

- The concept of *"function"* at the machine level is the same as in high-level programming languages
- **Function**: piece of code that receives data from the caller, execute code and returns some value
- **Calling convention**: how parameters are passed and value is retured = which instructions are executed to do this
- There exist several calling conventions, depending on architecture, OS and binary format

CALL and RET Instruction

- CALL **func** changes the execution flow to the **func** location. It stores return address of the instruction after the call onto the stack and moves into EIP the address of the first instruction of **func** (PUSH EIP; JMP func;)
- RET **value** returns to the previous function (i.e., caller). It restores return address saved by the CALL. (POP EIP;)

Calling Convention

- How a function receives parameters and returns the value and how it administers the stack
- There are several calling conventions that dictate where a caller should place any parameters that a function requires
- The high-level language, the compiler, the OS, and the target architecture all together “implement” and “agree upon” a certain calling convention.
- Let’s look at some examples

The C Calling Convention

- It's the default calling convention used by most C compilers for the x86 architecture
- When a compiler doesn't use this convention we can force it using the modifier _cdecl
- It specifies that the **caller places** parameters on the stack in the right to left order and that the **caller removes** the parameters from the stack after the called function completes

The C Calling Convention example

```c
void demo_cdecl(int a, int b, int c, int z);

//...

demo_cdecl(1, 2, 3, 4); //calling
```

```asm
; ...
push 4 ; push last parameter value
push 3 ; push third parameter value
push 2 ; ...
push 1
call demo_cdecl ; call the subroutine
```

The Standard Calling Convention

- This is Miscrosoft's calling convention standard
- When a compiler doesn't use this convention we can force it using the modifier _stdcall
- The parameters are passed using the stack as in _cdecl, but in _stdcall the called function is **responsible for clearing the function parameters from the stack before returning**. To do this, the function needs to know the right number of parameter passed. Can be used only with functions having a fixed number of parameters (e.g., no `printf`).

```
void _stdcall demo_stdcall(int a, int b, int c);
demo_stdcall(1, 2, 3);
```

```
ret 12 ; return and clear 12 bytes from the stack
```

fastcall Convention for x86

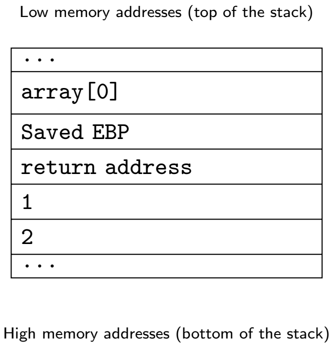
- Variation of the standard calling convention
- passes up to 2 parameters via registers (faster than stack).
- The Microsoft Visual C/ C++ and GNU GCC/g++ compilers recognize the fastcall modifier in function declaration.
- Example: the 1st and 2nd parameters are passed via ECX and EDX, the remainder parameters pushed on the stack in right to left order. The called function must cleanup the stack when returning.

```
; demo_fastcall(1, 2, 3, 4);
push 4 ; push 4th parameter
push 3 ; push 3rd parameter
mov edx, 2 ; move 2nd parameter in EDX
mov ecx, 1 ; move 1st parameter in ECX
call demo_fastcall
```

A Closer Look at the cdecl Calling Convention

```c
void function (int a, int b) {
  int array [5];
}

int main (int argc, char** argv) {
  function(1, 2);
  printf("This is where the ret
       address points");
}
```

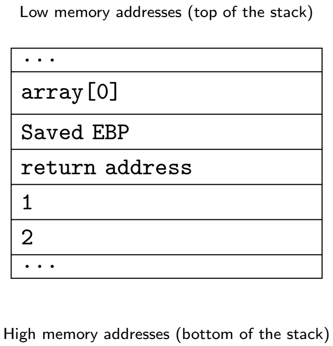


Low memory addresses (top of the stack)
...
array[0]
Saved EBP
return address
1
2
...
High memory addresses (bottom of the stack)

Assembler-level View of the Same Example

```
public main
main proc near
push ebp
mov ebp, esp ; create stack
and esp,0FFFFFFF0h
sub esp,10h ; make room for vars
mov dword ptr[esp+4],2 ; push 2
mov dwrod ptr[esp],1 ; push 1
call function
mov dword ptr [esp], offset format; "This is..."
call _printf
leave
ret
main endp

public function
function proc near
push ebp
mov ebp,esp ; create stack
sub esp,20h ; make room for array
leave
retn 8 ; remove 2 dwords
function endp
```



Low memory addresses (top of the stack)
...
array[0]
Saved EBP
return address
1
2
...
High memory addresses (bottom of the stack)

Objdump

- **man objdump**
  objdump **displays information** about one or more **object files**.
- -**x** all-headers
- -**d** disassemble
- -**M intel** intel syntax (default is att)

Our friend gdb

- **What is GDB?**
  GDB is GNU Project's Debugger: allows to follow, step by step, at assembler-level granularity, a running program, or what a program was doing right before it crashed.[3]

[3] http://www.gnu.org/software/gdb/

Start, break and navigate the execution with gdb

- Suppose you have an executable binary and want run it
  - **gdb /path/to/executable** loads the binary in gdb
- Now you decide to start the program with two parameters
  - **run 1 "abc"** passes 1 via `argv[1]` and `"abc"` as `argv[2]`
  - **run ‘printf "AAAAAAAAAAAA"‘** (with the back ticks) we're passing the output of the print (very useful when you need to pass non printable characters such as raw bytes)
- Suppose you want to stop the execution at the address of a certain instruction
  - **break *0xDEADBEAF** places a break point at that address
  - **break *main+1** with debugging symbols this can be less painful
  - **catch syscall** block the execution when a syscall happens

Start, break and navigate the execution with gdb

- Now the execution stops at our break point. Here we can do several things
- Examples:
  - **ni** allows to procede instruction per instruction
  - **next 4** moves 4 lines ahead (if you have the line-numbers information in the binary)
  - **si** step into function
  - **finish** run until the end of current function
  - **continue** runs until the next break point (if any)
- To see info about the execution state:
  - **info registers** to inspect the content of the registers
  - **info frame** to see the values of the stack frame related to the function where we are in
  - **info file** print the information about the sections of the binary

Navigate the stack

- Suppose we're stopped somewhere in the code and want to inspect the stack
- Some useful view of the stack is achievable with:
  - **x/100wx $esp** prints 100 words of memory from the address found in the ESP to ESP+100 (x = hexadecimal formatting)
  - **x/10wo $ebp-100** prints 10 words of memory from EBP-100 to EBP-100+10 (o = octal formatting)
  - **x/s $eax** prints the elements pointed by EAX (s = string formatting)
- Do you have debug symbols? (i.e., `gcc -ggdb`)
  - **print args** prints info about the main parameters
  - **print a** prints the content of variable 'a'
  - **print *b** prints the value pointed by 'b'

Our friend gdb

- **The '∼/.gdbinit' file**
  Gdb is a command line tool and it supports the configuration script as almost all the *nix software.
  Some options that you may want to tune are:
  - **set history save on**
    To have the lastest commands always available also when we re-open gdb
  - **set follow-fork-mode child**
    Allows you, if the process spawns children, to follow them and not only wait their end.
  - **set disassembly-flavor [intel | att]**
    This option sets in which predefined syntax your disassembled will be showed up. The default one is at&t
- Highly recommended to put this `https://github.com/gdbinit/Gdbinit` file in `~/.gdbinit`

Layout in gdb

- You're not a big fun of the gdb command line?
- Give a simple text-based interface to it
  - **layout asm** turn the interface to the assembly view always visible during debugging
  - **layout src** if your binary has the dubugging symbols you will have your c source view visible
  - **layout reg** add to the interface the register status view. It could be used in combination with one of the view described above
  - **gdb -tui ./mybin** runs gdb directly in this Text User Interface
- Use **CTRL+X A** to go back to standard interface.

man strace

In the simplest case **strace** runs the specified command until it exits. It intercepts and records the **system calls** which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed on standard error

ltrace

ltrace is a program that simply runs the specified command until it exits. It intercepts and records the **dynamic library calls** which are called by the executed process and the signals which are received by that process. It can also intercept and print the system calls executed by the program.

Bibliography

- The Ida Pro Book 2 Edition
- The Shellcoder Handbook
- Reverse Engineering Code with IDA Pro
- Secrets of Reverse Engineering
- Reverse Engineering for Beginners

End

That's all folks!
Time for (more) questions!