INTRODUCTION TO ASSEMBLY

$whoami

Kostas Mpenos

Computer Science Undergraduate

Linux Enthusiast

Programmer

Security Engineer (working on it)

ASSEMBLY LANGUAGE

Some basic definitions

ASSEMBLY BASICS

- Low-level programming language
- Human readable representation of machine language
- The interconnection between hardware and software programming
- Defined by the manufacturer of the cpu architecture

PROGRAMMING AND ASSEMBLY

- Assembly is a bit painful to program in
- Requires good understanding of both software and hardware functionality in order to build efficient programs
- High-level languages have been built in order to decrease the learning curve required and make programming a faster process

PROGRAMMING AND ASSEMBLY
-
THE PROCESS

- ➢ A program is created in a High-level language
- ➢ The program’s code is somehow translated (via a compiler/interpreter etc) into assembly code
- ➢ Assembly code is translated into machine code
- ➢ Hardware(Electronic Circuitry) takes over

THE PROCESS

HLL: High Level programming Languages
ASM: Assembly
ML: Machine Language

“This sounds good and all but i don’t need Assembly. I’m programming in HLL and i don’t see a reason to learn a language i’ll never use…”


If I agreed with you,
we would both be wrong.

So why do we need Assembly?

Any Ideas?

ASSEMBLY USAGE

Security

- Reverse Engineering
- Vulnerability Research
- Exploitation

Programming

- Debugging
- Hardware awareness

Hardware

- Every computer architecture has its own assembly
- Provides a fundamental functionality to circuits

Knowledge

- Better understanding of computers in overall
- It’s always interesting to learn new things :)

HARDWARE CONCEPTS

Before seeing and analysing some assembly code we need to address some hardware concepts of high importance.

HARDWARE TOPICS

CPU and Memory information we need in order to understand assembly

MEMORY

Computer Memory and Segmentation Concepts

MEMORY

Computer memory is a **flip-flop circuit structure** that's used as **temporary storage** in program execution.

The concept we will focus on, for the purposes of this presentation, is the **memory segmentation**.

MEMORY SEGMENTATION

A compiled program's memory is divided into **5** segments.

These are:

- *Code*
- *Data*
- *BSS*
- *Heap*
- *Stack*

MEMORY SEGMENTATION

Code Segment: This is where the assembled machine language instructions *(opcodes)* of the program are located.

Data Segment: The initialized global and static variables of the running program are stored here.

BSS: The uninitialized global and static variables of the program are stored in this segment.

MEMORY SEGMENTATION

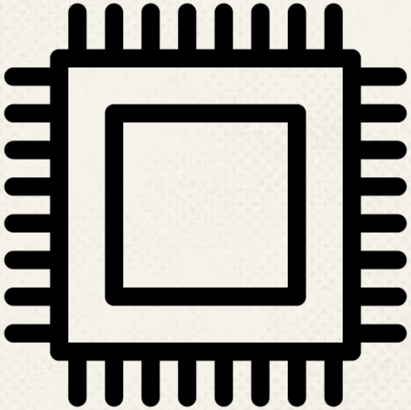
Heap: This memory segment is fully-controlled by the programmer. Memory can be allocated/unallocated as needed.

Stack: The stack memory segment is a storage for local function variables and context during function calls.

MEMORY SEGMENTATION

The **Code**, **Data** and **BSS** segments are of fixed size, while **Heap** and **Stack** are of variable size.

The **Heap** begins in lower memory addresses and *grows towards higher* ones, while the **Stack** starts in higher memory addresses and *grows towards lower* ones.



CPU

Central Processing Unit and its Registers

CPU PARTS

CPU is the part of a computer that translates instructions and carries out operations.

The very basic parts of a CPU are:

- ➢ *Control Unit*
- ➢ *Registers*
- ➢ *ALU (Arithmetic and Logic Unit)*

CPU PARTS

Control Unit: Fetches instructions from memory and executes them, in a very specific way.

ALU: Performs all the arithmetic and logic operations.

Registers: Special “variables” that store the result of ALU operations and /or supply ALU with operands for its operations.

CPU PARTS

While every part of the cpu is very interesting to analyse and strongly increases our understanding of computers, we will focus on explaining the **registers** for this presentation.

CPU REGISTERS

We will refer to the **Intel x86 registers** since they are the most common (very similar to the Intel x64, we will address the differences later).

CPU REGISTERS

There are mainly 4 kinds of registers:

- *General-purpose registers*
- *Segment registers*
- *The EFLAGS register*
- *The Instruction Pointer*

We will examine each of them separately and in depth.

GENERAL PURPOSE REGISTERS

There are **8** general-purpose registers in an Intel x86 architecture cpu.

These are:

EAX, ECX, EDX, EBX

ESP, EBP, ESI, EDI

Let’s walk through each one of them…

GENERAL PURPOSE REGISTERS

- **EAX (Accumulator)**: This register is used for I/O port access, arithmetic temp storage, interrupt calls, etc.
- **ECX (Counter)**: Mainly used as a counter for loops.
- **EBX (Base)**: This register is mainly used as a base pointer for memory access.
- **EDX (Data)**: Its use is similar to the one of Accumulator.

GENERAL PURPOSE REGISTERS

The registers mentioned above can be considered a distinct category.

You can simply think of them as temporary variables for the CPU while it is executing instructions, in order to efficiently bring out operations.

GENERAL PURPOSE REGISTERS

ESP (Stack Pointer): This register holds the memory address of the top of the stack.

EBP (Base Pointer): Usually referred to as stack base pointer or stack frame pointer, this register points to the current stack frame.

ESI (Source Index): Register used for pointing at the source of data for read/write operations.

EDI (Destination Index): Similar to the source index register but points to the destination.

GENERAL PURPOSE REGISTERS

The above registers can also be considered a distinct category.

You can simply think of them as pointers that provide an “image” of the memory operations in every frame of a program.

SEGMENT REGISTERS

The segment registers of the Intel x86 architecture are **6.**

These are:

CS, SS, ES, FS, DS, GS

SEGMENT REGISTERS

CS (Code Segment): This register holds the code segment in which the currently executed program runs.

DS (Data Segment): The Data segment register holds the data segment the currently executed program has access to.

SS (Stack Segment): This register holds the stack segment that the running program program uses.

SEGMENT REGISTERS

ES, **FS**, **GS**: These are some extra registers, used for far memory addressing, like video memory etc.

REMAINING REGISTERS

EIP (Instruction Pointer): Register that always points to the next instruction to be executed, for the currently running program.

EFLAGS: The eflags register consists of several bit flags that are used for comparisons and memory segmentations. It is also considered a representation of the state of the processor, in each frame of execution.

X86 VS X64

There are 3 basic differences between x86 and x64 Intel Architecture:

- ➢ In x64 architecture there are 8 additional general purpose registers (**r8**-**r15**)
- ➢ The registers that were **32-bit** in x86 were substituted by **64-bit** ones, in x64
- ➢ The “e” has been substituted by “r” in register names (ex. eax=rax, eip=rip etc.)

INTEL ENDIANNESS

Another fundamental concept is the **endianness** of the processor.

Endianness refers to the way a value/number/address is being stored in memory.

Intel x86 architecture uses **little endian representation** which means the least significant byte is stored first.

ASSEMBLY CODE

Program disassembling and Assembly Instructions

General syntax rule for Assembly instructions*:

<operation>   <destination>, **<source>**

For example:

mov   ebp, esp

sub   esp, eax

*Using Intel syntax and not AT&T

Common operations:

mov eax, 0x0: Move the value 0x0 in register eax.

add edi, esi: Add the content of esi register to the content of edi register and store it in edi.

sub edx, 0x12: Subtract the content of edx register by 0x12 and store the results in edx.

ASSEMBLY CODE

push ebp: Push the contents of ebp register onto the stack.

pop ebp: Pop the contents of ebp register out of the stack.

leave: Release the stack memory storage of the current stack frame.

ret: Return from procedure

ASSEMBLY CODE

cmp eax, 0x23: Compare the contents of eax register with the value 0x23 and update some flags.

jmp 0x4008a5: Unconditional jump to address 0x4008a5.

jle 0x500f3b: Jump to address 0x500f3b if the le (less or equal) flag is “invoked”. This refers to the result of the previous cmp operation.

(General rule: <*j[condition]*> <*address*>)

call 0x400ffa <func>: Call the subroutine located at address 0x400ffa.

and/or/xor edi, eax: Bitwise and/or/xor operations, storing the result in edi register.

not edx: Bitwise not operation, storing the result in edx register.

inc/dec ecx: Increment/Decrement the contents of ecx register.

ASSEMBLY CODE

imul eax, edx: Integer multiplication of the contents of eax and edx registers, storing the result in eax register.

idiv edi: Integer div of the value stored in EDX:EAX with the contents of the edi register. Stores the result in edi.

lea edi, [ebp + 0x1c] (Load effective address): Store the address of ebp + 0x1c into the edi register.

Now that we have the basic hardware knowledge and know some fundamental commands we can start examining some actual code.

Let’s start by disassembling a simple program, written in C.

ASSEMBLY CODE

```c
#include <stdio.h>

int main()
{
    printf("This is a simple program that prints to the screen and exits.");
    return 0;
}
```

ASSEMBLY CODE

```
gcc -g example.c -o example
gdb -q example
```

I compile the example.c using the -g option of gcc so i can then use gdb (GNU Debugger) in order to disassemble it.

ASSEMBLY CODE

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000000000400506 <+0>:     push   rbp
   0x0000000000400507 <+1>:     mov    rbp,rsp
   0x000000000040050a <+4>:     mov    edi,0x4005a8
   0x000000000040050f <+9>:     mov    eax,0x0
   0x0000000000400514 <+14>:    call   0x4003e0 <printf@plt>
   0x0000000000400519 <+19>:    mov    eax,0x0
   0x000000000040051e <+24>:    pop    rbp
   0x000000000040051f <+25>:    ret
End of assembler dump.
```

I passed gdb the command *disass main* (short for disassemble main) and i got the output as shown above.

The first 2 lines of output are called the **function prologue**.

Their sole responsibility is to prepare the stack and the registers for use in the function.

The last 2 lines of output are called the **function epilogue**.

Their responsibility is to return the stack and the registers to the state they were before the function was called.

The remaining 4 lines of code can be translated to the following actions respectively:

1. Move the value ***0x4005a8*** to the ***edi*** register
2. Move ***0x0*** to the ***eax*** register
3. Call the ***printf*** function
4. Move ***0x0*** to the ***eax*** register

Let's examine what are the contents of 0x4005a8 memory address

ASSEMBLY CODE

```
(gdb) break 5
Breakpoint 1 at 0x40050a: file example.c, line 5.
(gdb) run
Starting program: /home/mpkostas/Desktop/test/example

Breakpoint 1, main () at example.c:5
5               printf("This is a simple program that prints to the screen and exits.");
(gdb) x/64b 0x4005a8
0x4005a8:       84      104     105     115     32      105     115     32
0x4005b0:       97      32      115     105     109     112     108     101
0x4005b8:       32      112     114     111     103     114     97      109
0x4005c0:       32      116     104     97      116     32      112     114
0x4005c8:       105     110     116     115     32      116     111     32
0x4005d0:       116     104     101     32      115     99      114     101
0x4005d8:       101     110     32      97      110     100     32      101
0x4005e0:       120     105     116     115     46      0       0       0
```

I set a breakpoint at line 5 so the execution will stop right before the call to printf. Then i run the program and when it hits the breakpoint i examine the contents of memory address 0x4005a8. It doesn't really make sense at first but if you look closely the range of these numbers follows a pattern.

ASSEMBLY CODE

```
(gdb) x/64bc 0x4005a8
0x4005a8:       84 'T'  104 'h' 105 'i' 115 's' 32 ' '  105 'i' 115 's' 32 ' '
0x4005b0:       97 'a'  32 ' '  115 's' 105 'i' 109 'm' 112 'p' 108 'l' 101 'e'
0x4005b8:       32 ' '  112 'p' 114 'r' 111 'o' 103 'g' 114 'r' 97 'a'  109 'm'
0x4005c0:       32 ' '  116 't' 104 'h' 97 'a'  116 't' 32 ' '  112 'p' 114 'r'
0x4005c8:       105 'i' 110 'n' 116 't' 115 's' 32 ' '  116 't' 111 'o' 32 ' '
0x4005d0:       116 't' 104 'h' 101 'e' 32 ' '  115 's' 99 'c'  114 'r' 101 'e'
0x4005d8:       101 'e' 110 'n' 32 ' '  97 'a'  110 'n' 100 'd' 32 ' '  101 'e'
0x4005e0:       120 'x' 105 'i' 116 't' 115 's' 46 '.'  0 '\000'        0 '\000'
'\000'
```

So examining again using ascii interpretation for the results we get the output as shown above…

Finally we understand that the address of the string we want to print is loaded to the edi register which will be later used by the printf function.

In the end the value 0x0 is stored inside the eax register because in x86 Assembly the return value of functions is stored into the eax register.

ASSEMBLY CODE

We should keep in mind that the assembly code produced by a program is compiler dependent.

We should also keep in mind that the code is created by the compiler as a result of automation.

(Rarely, some operations might be redundant)

Let us, now, examine a more complex program, with more logic into it.

ASSEMBLY CODE

```c
#include <stdio.h>

int func(int a, int b)
{
    int sum;
    sum = a + b;
    return sum;
}

int main()
{
    int x, y, *z;
    x = 32;
    y = 19;
    int sum = func(x,y);
    int i;
    for(i=0; i<35; i++)
    {
        if(func(i,y)>50)
        {
            z = &x;
        }
    }
    return 0;
}
```

After we compile it and run gdb in the same way as before, we get the following output from the disassembling.

ASSEMBLY CODE

```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004004d0 <+0>:     push   rbp
   0x00000000004004d1 <+1>:     mov    rbp,rsp
   0x00000000004004d4 <+4>:     sub    rsp,0x20
   0x00000000004004d8 <+8>:     mov    DWORD PTR [rbp-0x1c],0x20
   0x00000000004004df <+15>:    mov    DWORD PTR [rbp-0x8],0x13
   0x00000000004004e6 <+22>:    mov    eax,DWORD PTR [rbp-0x1c]
   0x00000000004004e9 <+25>:    mov    edx,DWORD PTR [rbp-0x8]
   0x00000000004004ec <+28>:    mov    esi,edx
   0x00000000004004ee <+30>:    mov    edi,eax
   0x00000000004004f0 <+32>:    call   0x4004b6 <func>
   0x00000000004004f5 <+37>:    mov    DWORD PTR [rbp-0xc],eax
   0x00000000004004f8 <+40>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004004ff <+47>:    jmp    0x400521 <main+81>
   0x0000000000400501 <+49>:    mov    edx,DWORD PTR [rbp-0x8]
   0x0000000000400504 <+52>:    mov    eax,DWORD PTR [rbp-0x4]
   0x0000000000400507 <+55>:    mov    esi,edx
   0x0000000000400509 <+57>:    mov    edi,eax
   0x000000000040050b <+59>:    call   0x4004b6 <func>
   0x0000000000400510 <+64>:    cmp    eax,0x32
   0x0000000000400513 <+67>:    jle    0x40051d <main+77>
   0x0000000000400515 <+69>:    lea    rax,[rbp-0x1c]
   0x0000000000400519 <+73>:    mov    QWORD PTR [rbp-0x18],rax
   0x000000000040051d <+77>:    add    DWORD PTR [rbp-0x4],0x1
   0x0000000000400521 <+81>:    cmp    DWORD PTR [rbp-0x4],0x22
   0x0000000000400525 <+85>:    jle    0x400501 <main+49>
   0x0000000000400527 <+87>:    mov    eax,0x0
   0x000000000040052c <+92>:    leave
   0x000000000040052d <+93>:    ret
End of assembler dump.
```

ASSEMBLY CODE

```
(gdb) disass func
Dump of assembler code for function func:
   0x00000000004004b6 <+0>:     push   rbp
   0x00000000004004b7 <+1>:     mov    rbp,rsp
   0x00000000004004ba <+4>:     mov    DWORD PTR [rbp-0x14],edi
   0x00000000004004bd <+7>:     mov    DWORD PTR [rbp-0x18],esi
   0x00000000004004c0 <+10>:    mov    edx,DWORD PTR [rbp-0x14]
   0x00000000004004c3 <+13>:    mov    eax,DWORD PTR [rbp-0x18]
   0x00000000004004c6 <+16>:    add    eax,edx
   0x00000000004004c8 <+18>:    mov    DWORD PTR [rbp-0x4],eax
   0x00000000004004cb <+21>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004004ce <+24>:    pop    rbp
   0x00000000004004cf <+25>:    ret
End of assembler dump.
```

We can also disassemble the func function and get the output as shown above.

Let’s try to translate it as an exercise.

ASSEMBLY CODE

Ideas for research:

- ➢ Programming in Assembly
- ➢ Code Auditing in Assembly
- ➢ Creation of a minimal microprocessor and implementation of its own Assembly
- ➢ Research on a different Assembly architecture

THANKS!

Any questions?

You can find me at:

mpenoskp@csd.auth.gr
mpenos.ks@gmail.com

CREDITS

Links:

- ➢ Hacking - The Art of Exploitation (Book by Jon Erickson)
- ➢ http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- ➢ http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

Presentation Design:

- ➢ Presentation template by SlidesCarnival
- ➢ Backgrounds by SubtlePatterns