Some Notes on Reverse Engineering and x86 Assembly

Advanced Computer Security

Jonathan Sönnerup

October 11, 2016

0x100 Introduction

> *"... there's way too much information to decode the Matrix. You get used to it, though. Your brain does the translating. I don't even see the code. All I see is blonde, brunette, redhead."*
>
> – Cypher, The Matrix

High-level languages such as Java and C are based on English keywords representing what the programmer wants to to rather than how the processor should execute it. However, a processor does not understand `if`, `while` or `Turtle t = new Turtle()`.

The processor is a rather stupid piece of hardware that can move data to and from registers, fetch data from memory, perform simple math, and jump around in the code. Thus, to talk directly to the processor, we need something "lower" than C and Java — the assembly language. The assembly language is the language of the processor, using keywords like `mov`, `add`, `cmp` and `jmp`. This is known as a processor's instruction set.

An assembler is a piece of software that converts assembly code into machine code (ones and zeros). Different assemblers may have different syntax for the same assembly language. The two major syntaxes are Intel and AT&T syntax. From this point forward, Intel syntax will be used. The assembler used in the examples is called NASM. The specific Intel syntax may differ between assemblers, but the overall structure is the same.

0x200 The x86 Processor

Here, the x86 processor is introduced along with the x86 assembly language.

0x300 Registers

The x86 processor have eight general purpose registers, each 32-bit wide. These registers are called (not case-sensitive)

```
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
```

Two of the registers are reserved for special purpose — the stack pointer (`ESP`) and the base pointer (`EBP`). This means that we effectively have six general purpose registers, shown in Figure 1. Even though the registers are 32-bit wide, we may access smaller parts. If we take the `EAX` as an example, we may access the lower 16-bit value by calling the register `AX` (`EAX` actually stands for Extended AX). We may further divide the `AX` register into the upper and lower 8-bit values called `AH` and `AL`. The same applies for all `E*X` registers. To keep track of the current instruction to execute, the processor uses the `EIP` register.

Some example usage of the registers are shown below (";" defines a comment).

```
mov eax, 0x12345678     ; moves 0x12345678 to eax
mov bx, 0xaabb          ; moves 0xaabb to the lower 16 bits of ebx
mov cl, 0x37            ;
mov ch, 0x13            ; moves 0x1337 to the lower 16 bits of ecx
mov al, cl              ; move 0x37 to al
                        ; eax = 0x12345637
```

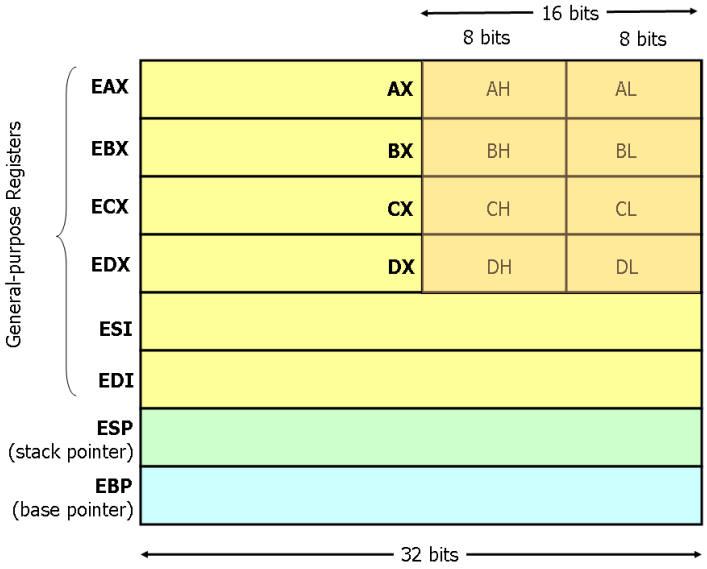

Figure 1: The x86 registers.

0x400 Memory and Addressing

0x410 Memory Sections

To run a program, we must first load it into the memory. Different parts of the program is placed in different *sections*, or *segments*, of the memory. A typical program can be divided into three sections:

text The text section contains the executable code. The section must begin with the declaration `global main`, which tells the operating system where to begin execute the code. This is equivalent to the `main` method in C and Java. The text section is declared as:

```
section .text
global main
main:
<code>
```

data The data section contains initialized static data, i.e. global variables and static local variables. The data section is read-write since the variables can be altered

at run time. This is in constrast to the read-only data section (`.rodata`), which contains static constants rather than variables. The data section is declared as:
`section .data`

bss The bss section is similar to the data section but contains uninitialized data instead. For example, the variable `static int i` ends up in the bss section. The bss section is declared as:
`section .bss`

There are two other data sections in the memory used to store local variables and dynamically allocated memory.

stack The stack section contains the stack of the program. It is implemented as a LIFO (Last In First Out) structure. In the x86, the stack begins at a high memory address and grows towards lower addresses. Data is pushed to the stack, making it grow downwards. A stack pointer register keeps track of the top of the stack, the `ESP` register.

heap The heap section is used for dynamic memory allocation. The heap grows up, towards the stack.

The memory sections are summarized in Figure 2.

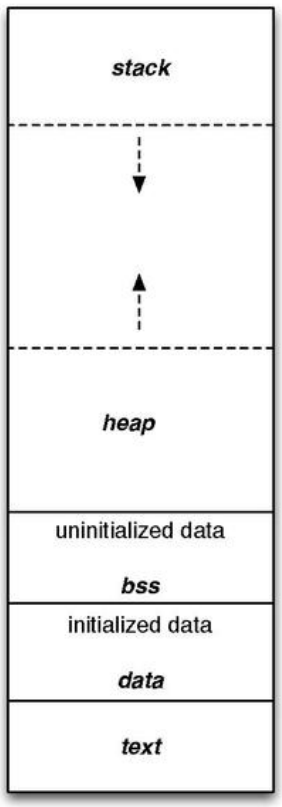

Figure 2: Memory sections of a program.

0x420 Addressing Memory

Just like C and Java, we can declare variables of different sizes, as in the example below.

```
section .data
    myvar1 db 0x65          ; declare a byte, called myvar (= 0x65)
    myvar2 dw 0x1337        ; declare a 2-byte variable
    myvar3 dd 0xdeadbeef    ; declare a 4-byte variable
    myarr1 db 1, 2, 3       ; declare a byte array with 3 values
    mystr db "hello",0      ; declare a NULL-terminated string
```

Notice that there is no such thing as signed or unsigned types. It is up to the programmer to handle the variables in a suitable way.

The variable names `myvar1`, `myarr1` etc. are replaces by their memory address by the assembler. The x86 is a 32-bit architecture, thus the addresses are 32-bit quantities. To use the variables in the assembly code, we must enclose the variable name with brackets. This tells the assembler that we want the value at the specific address given (variable name = address). Some examples of how to use addresses are shown below.

```
mov al, [myvar1]    ; move the 1-byte value at myvar1 to al
mov [myvar2], bx    ; move the 2-byte value in BX to myvar2
mov edx, [edx+ecx]  ; move the 4-byte value at edx+ecx
mov edx, [esi+4*ebx]; move the 4-byte value at esi+4*ebx
```

In the code above, the assembler knows implicitly how much data to move, due to the size of the variables. Sometimes, the size is ambiguous and we have to tell the assembler how much data to operate on. The code below shows an example of an ambiguous data size.

```
mov [eax], 0x42
```

Should we move the byte 0x42 to the address at eax, or the 2-byte 0x0042, or the 4-byte 0x00000042? To solve the problem, a size directive is added. The size directives `BYTE PTR`, `WORD PTR` and `DWORD PTR` indicates sizes of 1, 2 and 4 bytes. Note that in NASM syntax, the `PTR` keyword is discarded. An example:

```
mov BYTE [eax], 0x42    ; moves the byte 0x42 to the address in eax
mov WORD [eax], 0x42    ; moves the 2-byte 0x0042 to ...
mov DWORD [eax], 0x42   ; moves the 4-byte 0x00000042 to ...
```

0x430 Stack Manipulation

We have a very limited number of registers to store temporary data in, and it would be inconvenient to declare a bunch of variables. Since we can not remove the variables during run time, they would take up uneccessary space. The stack is our saviour — we can allocate and deallocate memory as we go. To add a value on the stack, we must

first allocate space in the memory. Remember that the stack grows downwards, so in order to allocate 4-bytes on the stack we subtract the stack pointer, `ESP` by 4 then move the value to that address, as shown below.

```
sub esp, 4
mov DWORD [esp], ebx    ; moves the value in ebx onto the stack
```

To retreive the value on the stack and deallocating the memory, we move the value and simply add 4 to the `ESP`.

```
mov ecx, DWORD [esp]
add esp, 4
```

Note, that this does not clear the value but it will be overwritten sooner or later, thus it should not be used! There is a simpler way to perform the actions above, by using the `push` and `pop` instructions, as shown below.

```
push ebx    ; allocate 4 bytes on the stack and move ebx there
pop ecx     ; move the value on the stack to ecx and deallocate 4 bytes
```

While the `ESP` register keeps track of the top of the stack, the `EBP` register keeps track of the bottom of the stack. The data area enclosed by `ESP` and `EBP` is known as a program's *stack frame*, depicted in gray in Figure 3.

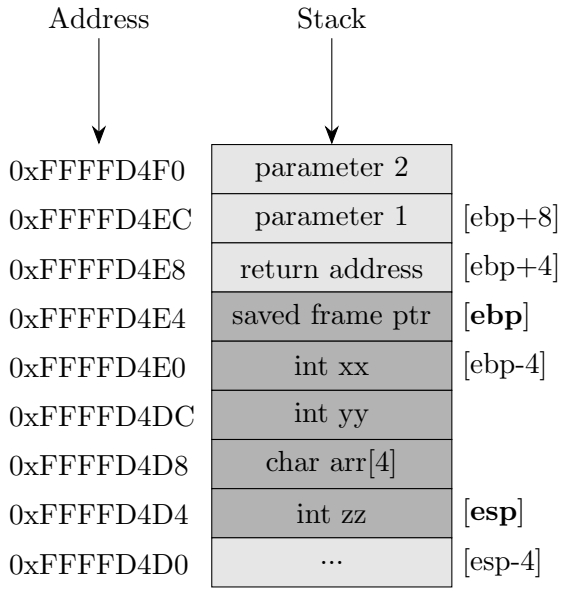

Figure 3: The stack with a highlighted stack frame.

0x500 Calling Conventions

When calling a subroutine, arguments are passed to the subroutine and the return value is passed back out. There must also be an agreement on how to manage the stack and its stack frame. How this is implemented is known as the *calling convention*. In the C language, there are three major calling conventions: *stdcall*, *cdecl* and *fastcall*. The function calling a subroutine is known as the *caller*, while the subroutine called is known as the *callee*.

0x510 cdecl

This is the default convention used in the C language. If the compiler used does not use this convention, it may be specified by prepending the keyword `_cdecl` to the function declaration.

The arguments are pushed on the stack in right-to-left order, i.e. the last argument first, and the return value is placed in the `EAX` register. The *caller* is responsible for cleaning the stack.

Let's look at the following C code:

```
_cdecl int myadd(int x, int y)
{
    return x + y;
}
```

This would produce the following assemly code:

```
myadd:
    push ebp            ;
    mov ebp, esp        ; set up the stack frame
    mov eax, [ebp+8]    ; fetch "x"
    mov ebx, [ebp+12]   ; fetch "y"
    add eax, ebx        ; eax = x + y
    mov esp, ebp        ;
    pop ebp             ; restore old stack frame
    ret                 ; return
```

The calling function (*caller*) would look like:

```
push 5        ; y = 5
push 7        ; x = 7
call myadd    ; myadd(7, 5)
add esp, 8    ; clean up the stack
```

0x520 stdcall

This is the standard convention for the Win32 API, so `stdcall` is also known as `winapi`. To specify the usage of this convention, `_stdcall` may be prepended to a function declaration.

Just like `cdecl`, the arguments are passed right-to-left and the return value is placed in the `EAX` register. In the Microsoft documentation, they claim that arguments are passed left-to-right, however this is not the case. Unlike `cdecl`, the `callee` is responsible for cleaning the stack.

The following C code:

```
_stdcall int myadd2(int x, int y)
{
    return x + y;
}
```

produces the assembly code:

```
myadd2:
    push ebp            ;
    mov ebp, esp        ; set up stack frame
    mov eax, [ebp+8]    ; fetch "x"
    mov ebx, [ebp+12]   ; fetch "y"
    add eax, ebx        ; eax = x + y
    mov esp, ebp        ;
    pop ebp             ; restore old stack frame
    ret 8               ; return and clean up stack
```

Note the `ret 8` statement. The `ret` instruction takes an optional argument that tells the program how many byte to pop off the stack. This is equivalent to `add esp, 8`.

The calling function would look like:

```
push 5      ; y = 5
push 7      ; x = 7
call myadd2 ; myadd2(7, 5)
```

0x530 fastcall

Coming soon

0x540 Tips & Tricks

Setting up the stack frame is so common that there are macros defined in NASM to do this for you, namely `enter a,b` and `leave`. In the `enter` macro, the `a` parameter specifies how much memory to allocate for on the stack for local variables, and the `b` parameter specifies the nesting level of the subroutine. The latter will not be further discussed, and may be set to 0.

```
enter 8,0
```

is equivalent to

```
push ebp
mov ebp, esp
sub esp, 8
```

The `leave` instruction is equivalent to

```
mov esp, ebp
pop ebp
```

0x600 x86 Disassembly

Coming soon