# SYS1 - Assembly

Avengers. . .

Jules Aubert

## About this document

This is not a real assembly cursus, just pseudo training.

You will have a real assembly class with another teacher, way more deep.

I won't go into details because we won't need it. I just want you to see one of the assembly languages.

To not fill your mind with too much information, just know that we will use Intel x86_64 language. You don't understand what it is exactly? Don't worry, you will see it in a near future with your assembly and reverse teacher (hurray!).

## Manual - Intel

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

The second manual contains the instructions set.

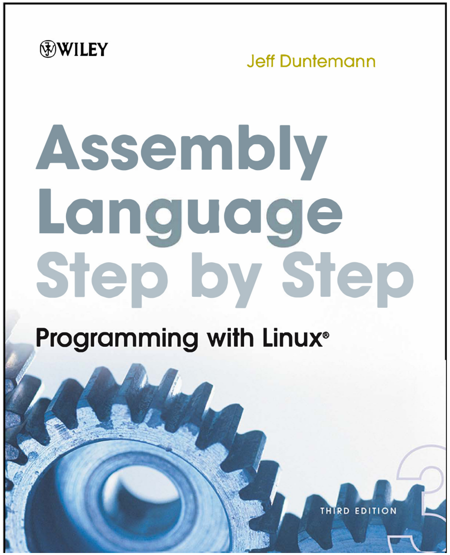
## Manual - Syscall

Since each Linux has its own architecture when it comes to the source, here the source of all sources about the Linux syscalls table.

https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

Example: write (sys_write) has syscall number 1. exit (sys_exit) has syscall number 60.

## Bibliography



WILEY

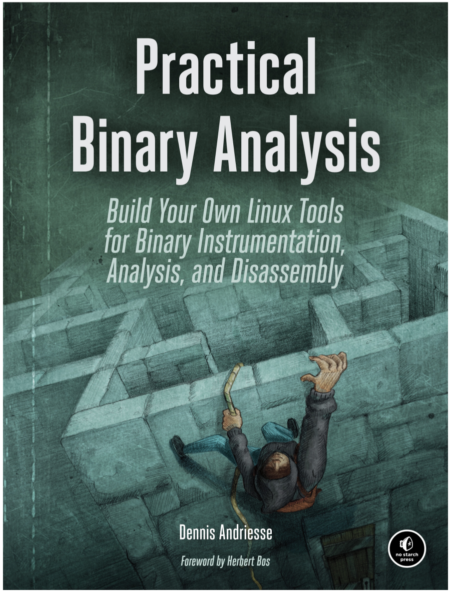
Jeff Duntemann

Assembly Language Step by Step

Programming with Linux®

THIRD EDITION

## Bibliography


Practical Binary Analysis
Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly
Dennis Andriesse
Foreword by Herbert Bos
no starch press

## What is assembly?

A human representation of machine code.

## The creation of assembly

Von Neumann (a genius among us, simple mortals) was teaching students machine code in the late 40s.

They weren't able to understand ~~(some people think they were EFREI students)~~.

Kathleen Booth decided to abstract the code proximity to the machine with mnemonic words.

## The creation of assembly

Von Neumann (a genius among us, simple mortals) was teaching students machine code in the late 40s.

They weren't able to understand ~~(some people think they were EFREI students)~~.

Kathleen Booth decided to abstract the code proximity to the machine with mnemonic words.

But some people are not busy enough and reverse directly machine code instead of translating into assembly

@DaemonOnUnix

Figure 1: Daniel

## Why using assembly nowadays?

Reverse kinda everything

Compiler Development (not in Tiger project)

Systems and architectures development

Code optimization

## Assembly x86

The most used architecture.

16 bits, 32 bits, 64 bits (x86_64)

## Instructions

*[prefix] opcode dst, src*

## Instructions

With the instructions, you can do whatever you want like in C programming language... with some more lines.

- Arithmetic
- Logic
- Memory access
- IO access
- Flow control
- ...

## Registers

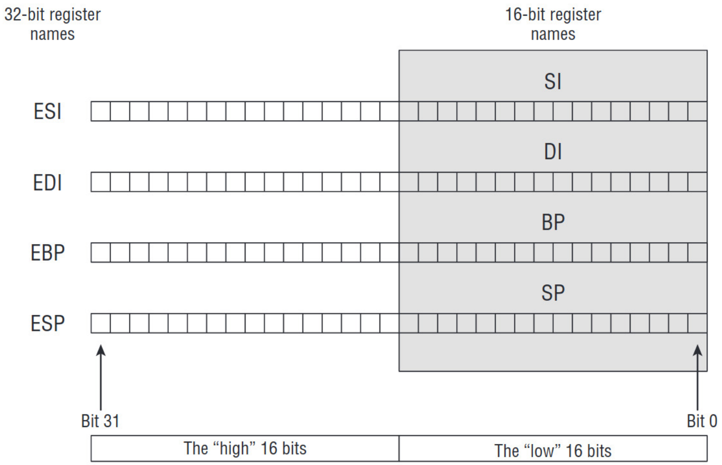
Storage spaces **inside** the processor.

_All_* storages are up to 64 bits value**, but we have few storages.

* A storage can be cut into sub-storage.

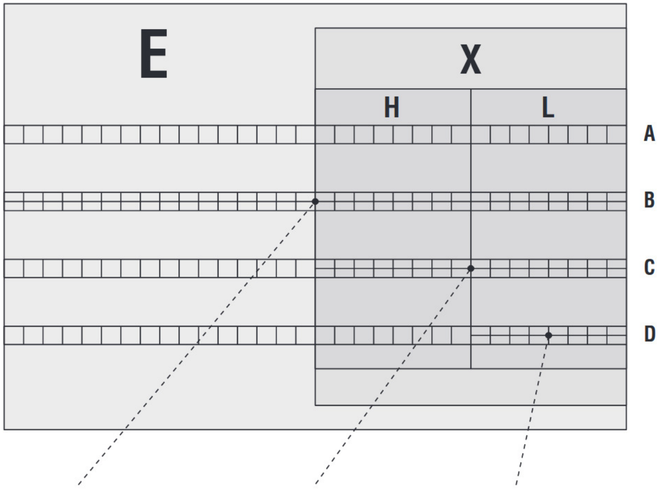
** because we're coding in x86_64 Intel assembly.

## Registers



The shaded portion of the registers is what exists on the older 16-bit x86 CPUs: The 8086, 8088, and 80286.

## Registers

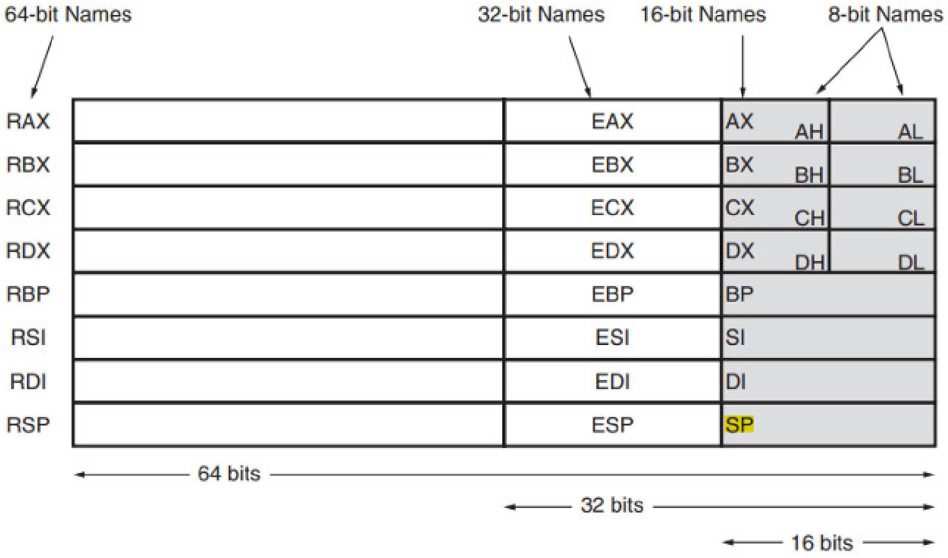


Specifying EBX embraces all 32 bits of the extended register

These 16 bits of the ECX register may be specified as CX

These 8 bits of the EDX register may be specified as DL

## Registers


64-bit Names
32-bit Names
16-bit Names
8-bit Names
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
EAX
EBX
ECX
EDX
EBP
ESI
EDI
ESP
AX
AH
AL
BX
BH
BL
CX
CH
CL
DX
DH
DL
BP
SI
DI
SP
64 bits
32 bits
16 bits

## Registers - Useful ones

- RAX: Contains the return value from a function
- RDI: Contains the first parameter when calling a function
- RSI: Contains the second parameter when calling a function
- RDX: Contains the third parameter when calling a function
- RCX: Contains the fourth parameter when calling a function
- %r8: Contains the fifth parameter when calling a function
- %r9: Contains the sixth parameter when calling a function

The other parameters are passed on the **stack**

## Registers - More specific

- EBP/RBP : Base Pointer ; Indicates the start of the function's stack frame
- EIP/RIP : Program counter ; Indicates the next instruction to be executed
- ESP/RSP : Stack Pointer ; Indicates the top of the stack

To get the 7th parameter, you can use **[rsp + 8]**

To get the 8th parameter, you can use **[rsp + 16]**

And so on. . .

## Instructions

MOV RAX, 42 $\leftrightarrow$ RAX $=$ 42

MOV RBX, RAX $\leftrightarrow$ RBX $=$ RAX

ADD RBX, RAX $\leftrightarrow$ RBX $=$ RBX $+$ RAX (RBX $+=$ RAX)

ADD RBX, RBX $\leftrightarrow$ RBX $+=$ RBX

? ?, ? $\leftrightarrow$ RAX $=$ 0

## Operands

- Immediate: 0x42
- Register: RAX
- Memory: [RAX]
- Memory shifting: [RAX + 8]

## Codes

Codes are long to write and may not be well displayed on the slides. Open the assembly tutorial and get into the full minimalist code chapter.

## Full minimalist code

Check the tutorial

## Compilation

```
nasm -f elf64 asm.S
ld asm.o -o asm
```

## Another minimal code (the compilation is different)

Check the tutorial

## Compilation, again

You can also use gcc after calling nasm and ask for not having Position Independent Executable. This will deactivate ASLR (Address Space Layout Randomization) and other securities for your binary, but it's not a big deal and it eases the compilation to learn a bit of assembly.

```
nasm -f elf64 asm.S
gcc -no-pie asm.o -o asm
```

## Difference?

The difference between the two “minimalist” codes, is that you define the start and exit functions in the first one. They are usually automatically created in the compilation process, but here you can define them aswell.

## Hello, World! with a syscall

Check the tutorial

## Hello, World! with puts

Check the tutorial

## Mixing assembly and C

Check the tutorial

## Mixing assembly and C

Check the tutorial

## Compile the two

```
nasm -f elf64 asm_myprint.s
gcc -no-pie asm_myprint.o test.c -o test
```

## One for the road: Addition

Your turn to work! From this C code, create an assembly code **add**ing up two parameters and returning the sum.

```c
// test.c
#include <stdio.h>

int asm_add(int a, int b);

int main(void)
{
    int c = asm_add(42, 51);
    return printf("%d\n", c);
}
```

## Solution

```
global asm_add

section .text
asm_add:
    mov rax, rdi
    add rax, rsi
    ret
```

## Compile it

```
nasm -f elf64 asm_add.s
gcc -no-pie asm_add.o test.c -o test
```

## Let’s do the math

Can you do the same for those function signatures?

```
int asm_sub(int a, int b);
int asm_mult(int a, int b);
int adm_div(int a, int b);
```

## The first problem with div

Let's read the manual about idiv

(more info in the tutorial)

## The first problem with div

Let's read the manual (it should be page 611)

We need rdx zeroed

(more info in the tutorial)

## The second problem with div

What if b equals 0?

```
a / 0 = ERROR
```

## The second problem with div

```
cmp rsi, 0
je .divide_zero
```

## asm div

```
global asm_div
.section text
asm_div:
    cmp rsi, 0
    je .divide_zero
    mov rax, rdi
    mov rcx, rsi
    xor rdx, rdx
    idiv rcx
    ret

.divide_zero:
    mov rax, 0
    ret
```

## Using assembly into C

```c
#include <stdio.h>

int main(void)
{
    puts("before the surprise");

    asm volatile ("int3\n");

    puts("after the surprise");

    return 0;
}
```

## What just happened?

Go to Intel vol.2, chapter 3. Search for “INT n/INTO/INT3/INT1-Call to Interrupt Procedure”.

int3 is a procedure to invoke a breakpoint exception.

Pretty neat when you want to write a debugger. . . or when you don’t want your code to be reverse engineered.

## Some more exercises

```c
// returns (a+b+c+d) / 4
int my_average(unsigned int a, unsigned int b, unsigned int c, unsigned int d);
// returns 1 if c belongs to [a-zA-Z], 0 otherwise
int my_isalpha(char c);
// returns the sum
int my_big_sum(int a, int b, int c, int d, int e, int f, int g);
// returns the biggest value
long int my_max(long int a, long int b, long int c);
// man 3 strlen
unsigned int my_strlen(const char *s);
// swaps the values pointed by 'a' and 'b'
void my_swap(int *a, int *b);
```