# SYS1

## *Assembly*

20xx

---

**Version 1**

**Jules AUBERT** <jules1.aubert@epita.fr>

---

# Contents

# 1 Introduction

## 1.1 Goal

The purpose of this course is to help you work a bit with assembly. More precisely, Intel x64 assembly.

This is not a real assembly course, just pseudo training. I do not go into details because we do not need it. I just want to have you work on one of the assemblies.

## 1.2 Resources

### 1.2.1 Intel

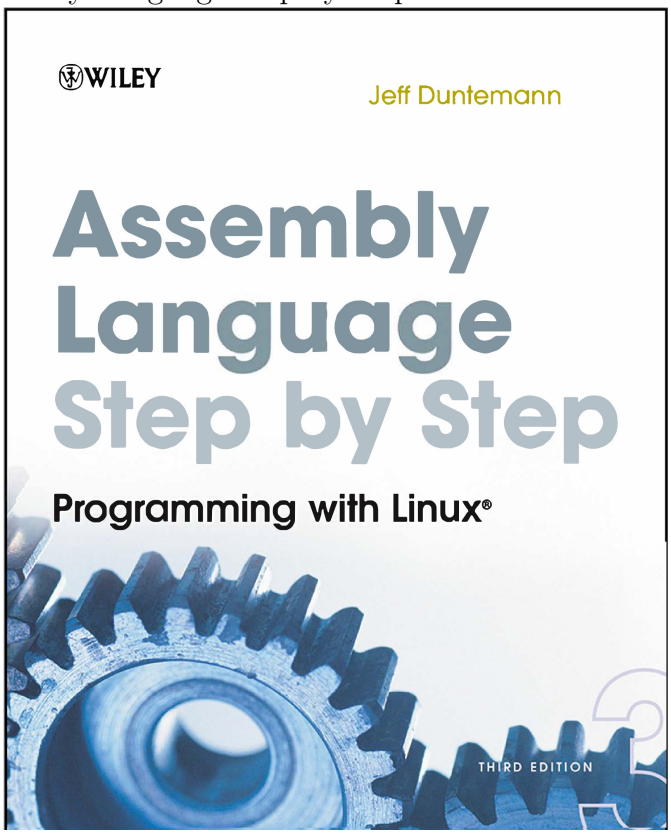
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
The second manual contains the instruction set.

### 1.2.2 Linux

Here is the almighty source about Linux syscalls when it comes to assembly. https://github.com/torvalds/linux
As you can see, `write (sys_write)` has syscall number 1. `exit (sys_exit)` has syscall number 60.

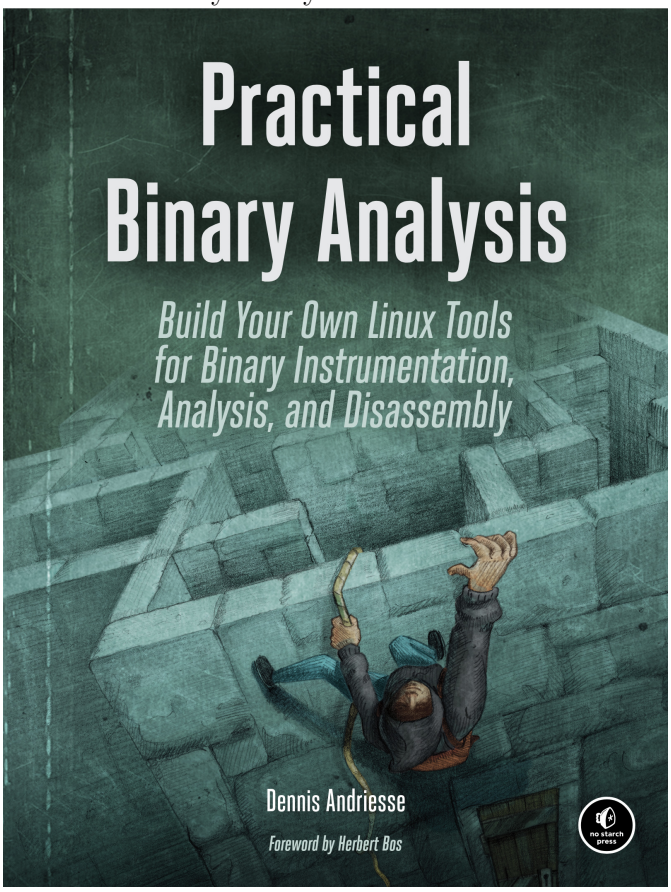
### 1.2.3 Book

Assembly Language Step by Step




ISBN: 0470497025

Practical Binary Analysis




ISBN: 978-1593279134

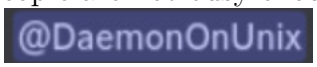
# 2 Assembly

## 2.1 Goal

The purpose of this chapter is to work with assembly, compile and execute some code, and so do some very tiny exercises.

## 2.2 What is assembly

Assembly is just *a human representation of machine code*. In the late 40s, Von Neumann was teaching students machine code. They had trouble to understand. 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 it into assembly: 


### 2.2.1 Using assembly nowadays

Assembly can be used for

- Reverse engineering
- Compiler development
- Embedded systems and architectures development
- Code optimization

## 2.3 Intel x86_64

We will use Intel x86_64 assembly. Depending of the architecture, you can have other assemblies (ARM, RISC-V, MIPS, SPARC, Web Assembly, ...).

The instructions are used for

- Arthmetic
- Logic
- Memory access
- IO access
- Flow control
- anything a programming language can do :)

### 2.3.1 Instructions order

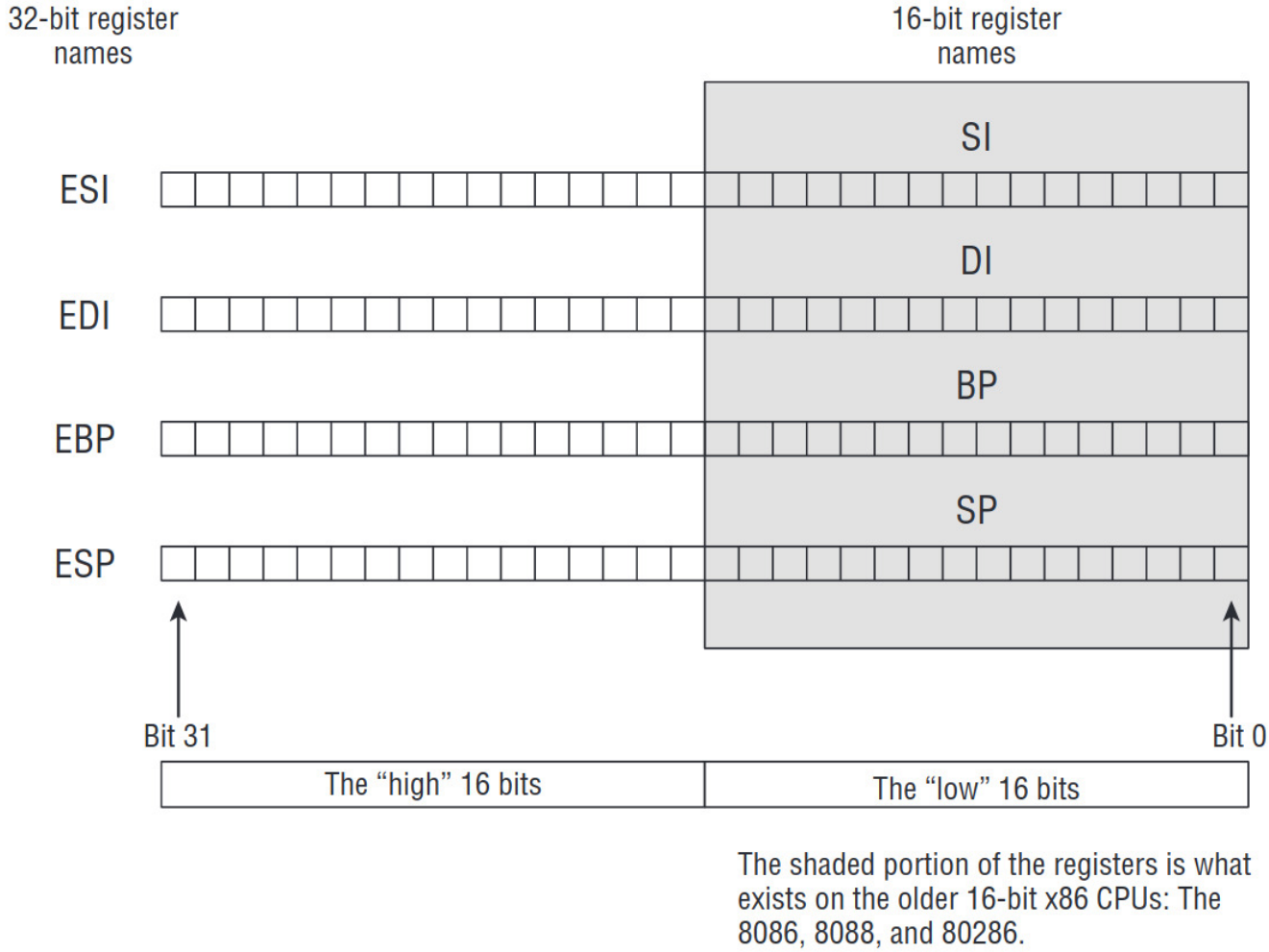
Instructions for Intel are used this way:

```
[prefix] opcode destinaton, source
```

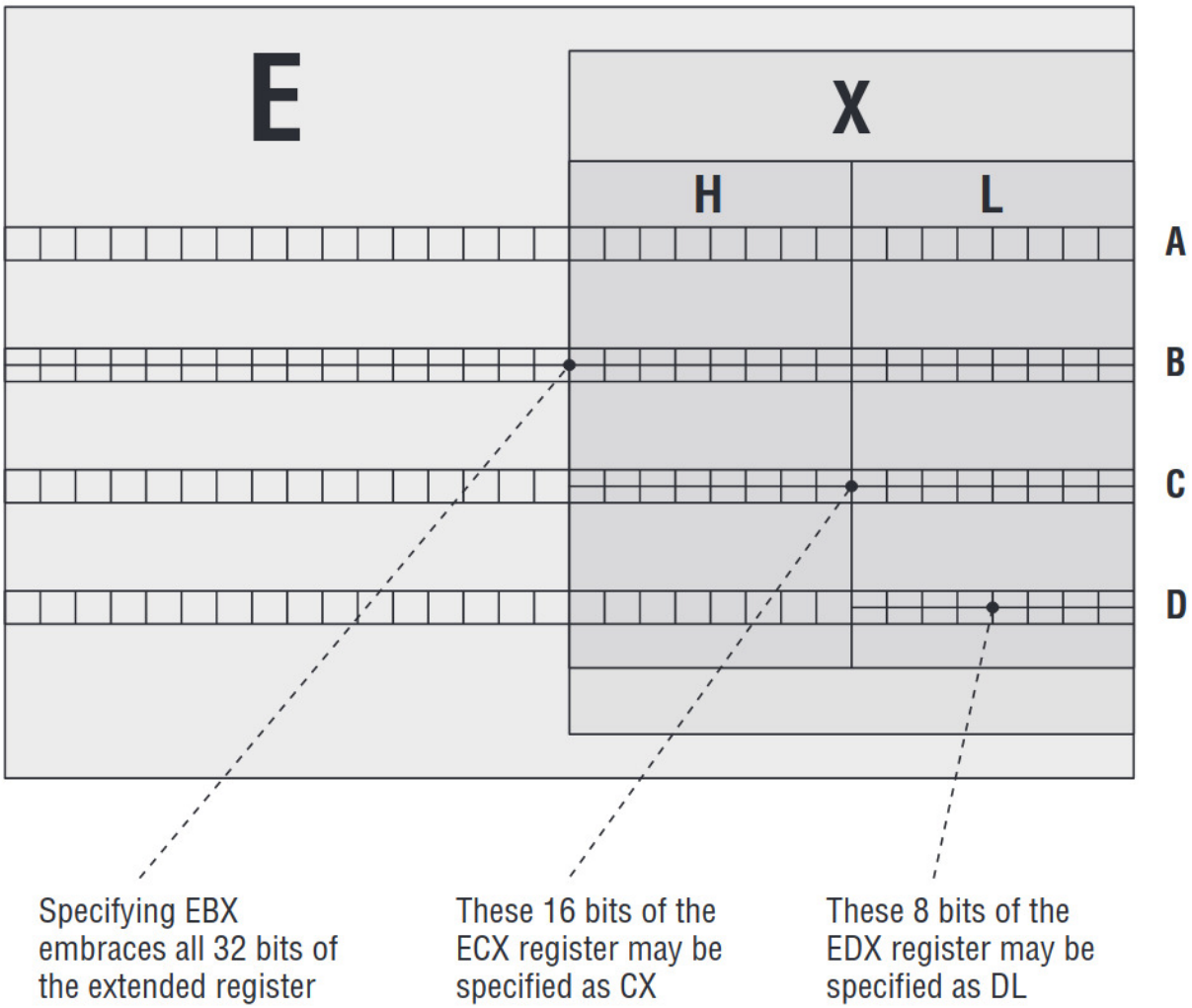
### 2.3.2 Registers

Registers are storage space **inside** the processor. They are up to 64 bits value and can be cut into sub-storage (32 bits, 16 bits, 8 bits).

Here are some 32 bits registers:

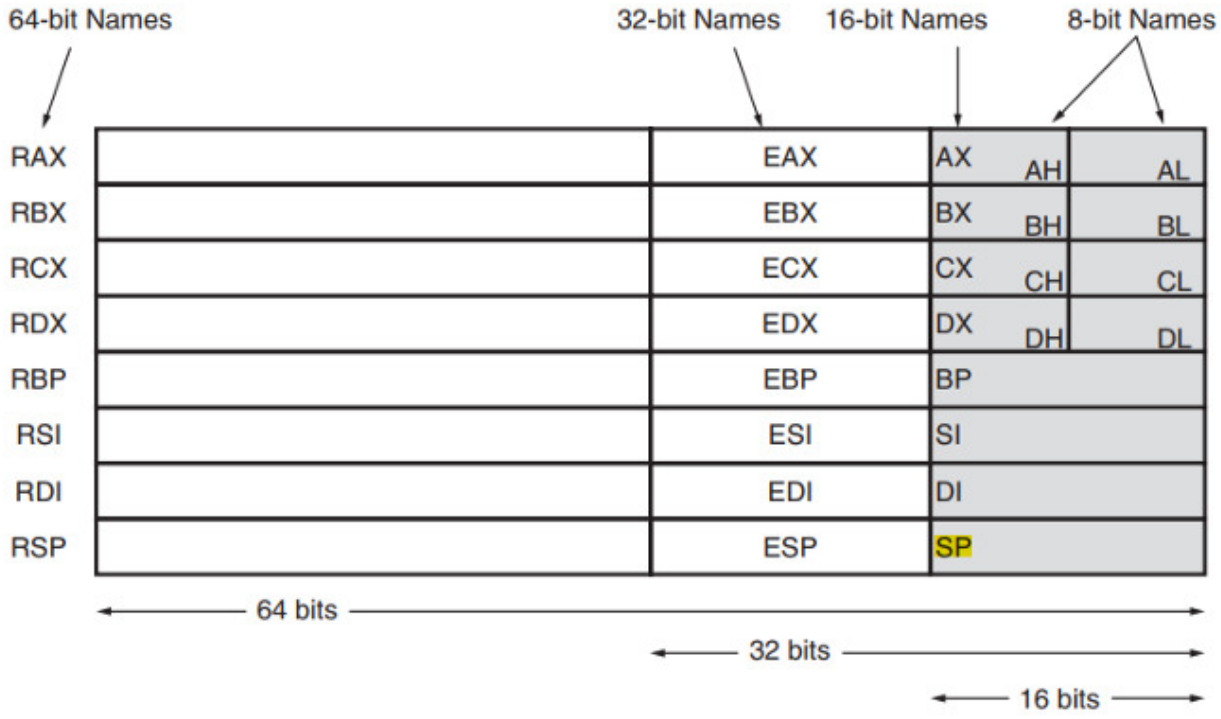



Among other 32 bits registers:




Cutted into 16 bits sub-registers also cutted into 8 bits sub-sub-registers.

And some 64 bits registers including the 32 bits registers:



Here are some useful registers to begin:

- **RAX**: Contains the return value from a function or the syscall number to call
- **RDI**: Contains the first parameter when calling a function
- **RSI**: Contains the second parameter
- **RDX**: Contains the third parameter
- **RCX**: Contains the fourth parameter
- **r8**: Contains the fifth parameter
- **r9**: Contains the sixth parameter

The other parameters are passed on the **stack**.
Here are some more specific registers:

- **RBP**: Base Pointer: Indicates the start of the function's stack frame
- **RIP**: Instruction Pointer: Indicates the next instruction to be executed
- **RSP**: Stack Pointer: Indicates the top of the stack

RIP is also more commonly known as Program Counter in the assembly world.

To get the 7th parameter, you can use `[RSP + 8]`.
To get the 8th parameter, you can use `[RSP + 16]`.
And so on...

Get the Intel Manual 2nd volume and open it. Then, for each instruction you see, search for it (they are list in lexical order).

The arrows are here to help you understand what are the effects from the instructions. **They are not assembly instructions!**

```
MOV RAX, 42     <=> RAX = 42
MOV RBX, RAX    <=> RBX = RAX
ADD RBX, RAX    <=> RBX = RBX + RAX (RBX += RAX)
ADD RBX, RBX    <=> RBX += RBX
XOR RAX, RAX    <=> RAX = RAX ^ RAX (RAX = 0)
MOV RAX, 0      <=> RAX = 0 ; but slower than the above instruction
```

You do not have to use uppercase for the opcodes and operands. You can write them in lowercase.

### 2.3.3 Operands

In assembly language, operands are the values or locations that an instruction operates on. An instruction typically consists of an operation (the opcode) and one or more operands. Operands can be:

- **Immediate values**: `42`, `0xFF`
- **Registers**: `rax`, `rbx`, `rsp`
- **Memory addresses**: `[rax]`
- **Labels**: which refer to addresses of code or data

Example: `mov rax, [rbp - 8]` Here, `rax` and `[rbp - 8]` are operands. The instruction moves the value from memory into the `rax` register.

## 2.4 Full minimalist code

**If you use Ubuntu, because that is what the school installed on your laptops (sigh), I may have to add assembly lines, because Ubuntu's toolchain makes a weird runstart.**

Here is a full minimalist and working code:

```
section .text           ; section with code
global _start           ; global because it needs to be access from outside the file

_start:                 ; _start is the needed symbol to link to an executable
    call main           ; pretty obvious
    call _exit          ; pretty obvious

_exit:                  ; _exit is the needed symbol for an elf file to exit
    mov rax, 60         ; 'sys_exit' is syscall number 60
    mov rdi, 42         ; '42' is the status exit code
    syscall             ; call the syscall number 60 (sys_exit)

main:                   ; our main file
    ret                 ; return from main (main returns int) the value in rax
```

To compile it, we will use the **Netwide Assembler**, and `ld(1)` to link it.

```
$ nasm -f elf64 asm.S
$ ld asm.o -o asm
$ ./asm
$ echo $?
42
$
```

Be aware that we need **__start** to be the entry point. Forget about **main**. **main** is called by **__start** when you compile with **gcc(1)**, which add **__start** itself during the compilation of a C program.

## 2.5 Hello APPING!

### 2.5.1 With the syscall write

```
section .text
global _start

_start:
    mov rax, 1                  ; syscall number for write
    mov rdi, 1                  ; stdout
    mov rsi, msg                ; pointer to message
    mov rdx, len                ; message length
    syscall                     ; invoke syscall (write)

    mov rax, 60                 ; syscall number for exit
    xor rdi, rdi                ; exit code 0
    syscall                     ; invoke syscall (exit)

section .data
msg: db "Hello, World!", 10, 0  ; message with newline and null terminator
len: equ $ - msg                ; compute length
;$ do not mind this line
```

```
$ nasm -f elf64 asm.S
$ ld asm.o -o asm
$ ./asm
Hello, APPING!
$
```

### 2.5.2 With puts and gcc

Here, we are going to use `puts(3)` to call our string.

```
extern puts                 ; gcc will link 'puts'

section .text
global main

main:                       ; our main file
    mov rdi, msg                ; 'msg' is the only parameter needed for puts
    call puts                   ; pretty obvious, isn't it?
    ret                         ; return from main (main returns int) the value in rax

section .data               ; section with variables, etc.
msg:
    db "Hello, APPING!", 0      ; msg is "Hello, APPING!" + '\0'
```

To compile, we need to use `-no-pie` with **gcc(1)**.

```
$ nasm -f elf64 asm.S
$ gcc -no-pie asm.o -o asm
$ ./asm
Hello, APPING!
$
```

Modern Linux distributions compile executables as PIE (Position Independent Executables) by default. This means the code is generated to support address randomization at runtime, enhancing security through ASLR (Address Space Layout Randomization). PIE executables use relative addressing (such as RIP-relative addressing on x86-64) and are loaded at random memory locations.

However, when writing raw assembly code, especially when defining your own `_start` symbol and accessing static data (like labels or strings) you typically assume a fixed layout with absolute addresses. If your code is linked as a PIE, these assumptions break, potentially leading to segmentation faults.

To prevent this, you should explicitly tell the compiler or linker to produce a classic, non-PIE executable using the `-no-pie` option. This ensures that the linker generates a fixed-address binary, compatible with your assembly.

About the prologue and epilogue: The function prologue (`push rbp; mov rbp, rsp`) and epilogue (`mov rsp, rbp; pop rbp`) are common in assembly functions to set up and tear down the stack frame. They are typically used in C generated code to enable reliable access to local variables and function arguments, and to support tools like debuggers and backtrace functions.

However, for simple functions that do not allocate stack space or use `rbp`, these steps are not strictly required. On some systems such as Arch Linux, you can omit the prologue and epilogue entirely if your function respects the calling convention (e.g., using `rdi` for the first argument and returning with `ret`). That said, including the standard prologue can improve compatibility and maintainability.

If you use Ubuntu, you need to add them. If you use EPITA's distro (NixOS) and EPITA's former distro (Arch Linux), you do not need to use it.

## 2.6 Mixing assembly and C

Because assembly and C are compiled into machine code using same tools from the toolchain, you can call assembly code from C.

Here is a simple example relying on previous code (calling puts). This is **test.c**:

```c
int asm_myprint(const char *);

int main(void)
{
    asm_myprint("Hello APPING!\n");
    return 0;
}
```

And this is **asm_myprint.S**:

```
global asm_myprint
extern puts

section .text
asm_myprint:
    call puts
    ret
```

And compile:

```
$ nasm -f elf64 asm_myprint.S
$ gcc -no-pie asm_myprint.o test.c -o test
$ ./test
Hello APPING!

$
# $ do not pay attention to this line
```

## 2.7 Addition

Almost 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);
}
```

```
// asm.S
global asm_add

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

Compile and execute:

```
$ nasm -f elf64 asm.S
$ gcc -no-pie asm.o test.c -o test
$ ./test
93
```

```
$
```

## 2.8 Exercises

Your turn to work now! Can you do these exercises? Remember, you have an Intel manual with you.

```c
// returns a - b
int asm_sub(int a, int b);

// returns a * b
int asm_mult(int a, int b);

// returns a / b
int asm_div(int a, int b);
```

## 2.9 Corrections

```
section .text
global asm_sub
global asm_mult

; int asm_sub(int a, int b)
asm_sub:
    mov rax, rdi      ; a
    sub rax, rsi      ; a - b
    ret

; int asm_mult(int a, int b)
asm_mult:
    mov rax, rdi      ; a
    imul rsi     ; a * b (signed multiplication)
    ret
```

### 2.9.1 div

We have a problem with the **div** exercise. Let's read the manual.

```
ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
        temp := RDX:RAX / SRC; (* Signed division *)
        IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
        (* If a positive result is greater than 7FFFFFFFFFFFFFFFH
        or a negative result is less than 8000000000000000H *)
            THEN
                #DE; (* Divide error *)
            ELSE
                RAX := temp;
                RDX := RDE:RAX SignedModulus SRC;
        FI;
    FI;
FI;
```

IDIV—Signed Divide Vol. 2A 3-506

We need rdx zeroed.

When using the `idiv` instruction with 64-bit operands, the CPU performs a signed division of a 128 bits dividend stored in the `RDX:RAX` register pair, divided by a 64 bits source operand (e.g., `rsi`, `rcx`, etc.).

Before committing the result, the CPU checks whether the quotient fits in a 64 bits signed register:

- If the result is greater than `0x7FFFFFFFFFFFFFFF` (INT64_MAX), or less than `0x8000000000000000` (INT64_MIN), the CPU raises a **division error exception** (#DE). This typically results in a segmentation fault or a SIGFPE in user space.
- If the result is within range, the quotient is stored in `RAX`, and the remainder is stored in `RDX`.

This is why it is crucial to prepare `RDX` correctly before executing `idiv`. Failing to do so may produce incorrect results or trigger an exception even when the division is mathematically valid.

```
global asm_div          ; Make the function symbol visible to the linker

section .text           ; Start of the code section

asm_div:
    cmp rsi, 0          ; Check if divisor (b) is zero
    je .divide_zero     ; Jump to handler if b == 0

    mov rax, rdi        ; Move dividend (a) into RAX
    mov rcx, rsi        ; Move divisor (b) into RCX

    xor rdx, rdx        ; Clear RDX (assumes dividend is non-negative)
                        ; No sign extension is performed

    idiv rcx            ; Perform signed division: RDX:RAX œ RCX
                        ; Result in RAX (quotient), remainder in RDX

    ret                 ; Return to caller with result in RAX

.divide_zero:
    mov rax, 0          ; Return 0 in case of division by zero
    ret                 ; Return to caller
```

## 2.10 Using assembly into C

From a C code, you can use, not call, but use assembly instructions.

```
#include <stdio.h>

int main(void)
{
    puts("before the surprise");
    asm volatile ("int3\n");
    puts("after the surprise");
    return 0;
}
```

I let you compile and execute it. :)

What just happened?

In the Intel manual, search for `INT3` or `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 do not want your code to be reverse engineered.

## 2.11 More exercises for the road

If you want to get better, try to exercises. They are bonus, you will have to find the solution yourself.

```
// 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
long 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);
```