Preparing the environment

Controlling the presence of GNU C Compiler

```
gbiondo@LinuxMint:~/Development$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.4.0-1ubuntu1~18.04.1'
--with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c+
+,go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-
suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id
--libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/
lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --
enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object
--disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-
system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --
disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --
enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-
driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --
target=x86_64-linux-gnu
Thread model: posix
gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)
gbiondo@LinuxMint:~/Development$
```

Controlling the presence of Python

```
gbiondo@LinuxMint:~/Development$ python -V
Python 2.7.17
gbiondo@LinuxMint:~/Development$
```

Controlling the presence of nasm

```
gbiondo@LinuxMint:~/Development$ nasm -v
NASM version 2.13.02
gbiondo@LinuxMint:~/Development$
```

Controlling the presence of ld

```
gbiondo@LinuxMint:~/Development$ ld -v
GNU ld (GNU Binutils for Ubuntu) 2.30
gbiondo@LinuxMint:~/Development$
```

Registers

A CPU register, or just register, is a temporary storage or working location built into the CPU itself, separate from memory. Computations are typically performed by the CPU using registers. It is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage, although some registers have specific hardware functions, and may be read-only or write-only.

X64 assembly code uses sixteen 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers. The register names are as follows:

Bytes 0-7 (64 bits)	**Bytes 0-3 (32 bits)**	**Bytes 0-1 (16 bits)**	**Byte 0 (8 bits)**	**Usage**
rax	eax	ax	al	Used in arithmetic operations
rcx	ecx	cx	cl	Used in shift/rotate instructions and loops
rdx	edx	dx	dl	Used in arithmetic operations and I/O operations
rbx	ebx	bx	bl	Used as a pointer to data (located in segment register DS, when in segmented mode)
rsi	esi	si	sil	Used as a pointer to a source in stream operations
rdi	edi	di	dil	Used as a pointer to a destination in stream operations
rsp	esp	sp	spl	Pointer to the top of the stack, and should not be used for data or other uses.
rbp	ebp	bp	bpl	Used to point to the base of the stack. It is used as a base pointer during function calls. This register should not be used for data or other uses.
r8	r8d	r8w	r8b	
r9	r9d	r9w	r9b	
r10	r10d	r10w	r10b	
r11	r11d	r11w	r11b	
r12	r12d	r12w	r12b	
r13	r13d	r13w	r13b	
r14	r14d	r14w	r14b	
r15	r15d	r15w	r15b	

Instruction Pointer Register (`RIP`)

In addition to the GPRs, there is a special register, `RIP`, which is used by the CPU to point to the next instruction to be executed. Specifically, since the `RIP` points to the next instruction, that means the instruction being pointed to by `RIP`, and shown in the debugger, has not yet been executed.

Flag Register (`rFlags`)

The flag register, `rFlags`, is used for status and CPU control information. The `rFlags` register is updated by the CPU after each instruction and not directly accessible by programs. This register stores status information about the instruction that was just executed. Of the 64-bits in the rFlag register, many are reserved for future use. We will come back to this topic later in greater detail.

Access modes

All registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, the register is identified by its two-letter abbreviation from the list above. In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (extended). For example, 'EAX' is the accumulator register as a 32-bit value. Similarly, in the 64-bit version, the 'E' is replaced with an 'R' (register), so the 64-bit version of 'EAX' is called ‘RAX'.

It is also possible to address the first four registers (AX, CX, DX and BX) in their size of 16-bit as two 8-bit halves. The least significant byte (LSB), or low half, is identified by replacing the 'X' with an 'L'. The most significant byte (MSB), or high half, uses an 'H' instead. For example, CL is the LSB of the counter register, whereas CH is its MSB.
In total, this gives us five ways to access the accumulator, counter, data and base registers: 64-bit, 32-bit, 16-bit, 8-bit LSB, and 8-bit MSB. The other four are accessed in only four ways: 64-bit, 32-bit, 16-bit, and 8-bit.

	ACCUMULATOR	COUNTER	DATA	BASE	STACK PTR	STACK BASE	SOURCE	DESTINATION
64 BITS	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32 BITS	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16 BITS	AX	CX	DX	BX	SP	BP	SI	DI
8 BITS	AH AL	CH CL	DH DL	BH BL	SPL	BPL	SIL	DIL

Data Types

The following table explains the data types in assembly based on length:

Name	Directive	Bytes
Byte	db	1
Word	dw	2
Doubleword	dd	4
Quadword	dq	8

First steps in assembly programming

The format of a program

We need first the system call (syscall) `exit`. To find it, we can look at the file

`/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

as follows

```
gbiondo@LinuxMint:~/Development$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h |
grep -i exit
#define __NR_exit 60
#define __NR_exit_group 231
gbiondo@LinuxMint:~/Development$
```

Once this is known (the `syscall` is the number `60`), we need to investigate how it works and the parameters it requires:

```
gbiondo@LinuxMint:~/Development$ man 2 exit
NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);

       #include <stdlib.h>

       void _Exit(int status);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
...
gbiondo@LinuxMint:~/Development$
```

Chiefly, this instruction only requires one parameter, `status`, and returns nothing (it is a `void` function).

Traditionally, the first program one writes is the well known 'hello world'. Let's stick to this tradition - so we only need to know how to write on the screen. We know how to do this; first we look fo the `write` syscall:

```
gbiondo@LinuxMint:~/Development$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h |
grep -i write
#define __NR_write 1
#define __NR_pwrite64 18
#define __NR_writev 20
#define __NR_pwritev 296
#define __NR_process_vm_writev 311
#define __NR_pwrite2 328
gbiondo@LinuxMint:~/Development$
```

which has the number `1` associated; then we look at its man page:

```
gbiondo@LinuxMint:~/Development$ man 2 write
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the file
referred to by the file descriptor fd.
…
gbiondo@LinuxMint:~/Development$
```

The system call has then three arguments: a file descriptor, the location of memory containing the value to print, and the number of bytes to print.

File descriptors are usual Unix ones:

Value	Name	Alias in `stdio.h`
0	Standard Input	stdin
1	Standard Output	stdout
2	Standard Error	stderr

In order to invoke a syscall, the programmer should load the RAX register with the syscall number (in this case, 1), and the arguments must be load as follows:

Argument	Register
1	`RDI`
2	`RSI`
3	`RDX`
4	`R10`
5	`R8`
6	`R9`

In this case, we will proceed by loading the registers as follows:

Register	VALUE
RAX	`1 (syscall number)`
RDI	`1 (stdout)`
RSI	`Address of “Hello world”`

Register	VALUE
RDX	`Length of “Hello world”`

A variable will be defined, `hello_world`, containing the string `'hello world’`, terminated by a line feed (`\n`), whose ASCII encoding is `10`, or `0xa`. This is a string of bytes, so the data type will be `db`.

Another variable to be defined is the length of the `hello_world` string. This is obtained by the instruction `$–`, which evaluates the current line.

The two variables will be declared in the .data section of our program. The .text section, containing the real program, will just need to load the registers as previously described and to invoke the system call.

The code is as follows:

```
global _start

section .text

_start:

       mov rax, 1
       mov rdi, 1
       mov rsi, hello_world
       mov rdx, length
       syscall

section .data

       hello_world: db 'hello world',0xa
       length: equ $-hello_world
```

We compile, link and run the executable as follows:

```
gbiondo@LinuxMint:~/Development/HelloWorld$ nasm -felf64 hello.nasm -o hello.o
gbiondo@LinuxMint:~/Development/HelloWorld$ ld hello.o -o helloWorld
gbiondo@LinuxMint:~/Development/HelloWorld$ ./helloWorld
hello world
Segmentation fault (core dumped)
gbiondo@LinuxMint:~/Development/HelloWorld$ echo $?
139
gbiondo@LinuxMint:~/Development/HelloWorld$
```

The core dump: Unix systems return error n. 128 + signal when a signal is received. 128 + 11 = 139 . Signal 11 is SIGSEV (i.e. segmentation violation). In fact, there has been a segmentation violation, in the sense that the program invokes no ‘`exit`’ syscall. In fact, changing the program as follows:

```
global _start

section .text

_start:

	mov rax, 1
	mov rdi, 1
	mov rsi, hello_world
	mov rdx, length
	syscall

	mov rax, 0x3C
	syscall

section .data

	hello_world: db 'hello world',0xa
	length: equ $-hello_world
```

solves the problem.

Fun with Arithmetics

The very first program does not do so much:

```
global _start

section .text

_start:

    mov rax,0x1
    add rax,0x2

    mov rax, 60
    mov rdi, 0
    syscall
```

An analysis with GDB can help further. The program has been compiled as follows:

```
gbiondo@LinuxMint:~/Development/sumDiffs $ nasm -felf64 main1.nasm -g -F dwarf -o
main1a1.o
gbiondo@LinuxMint:~/Development/sumDiffs $ ld main1a1.o -o main1a1
```

And we invoke GDB as follows:

```
gbiondo@LinuxMint:~/Development/sumDiffs $ gdb main1a1
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main1a1...done.
```

First, we want to see the source code, to understand where putting breakpoints

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:	mov    $0x1,%eax
   0x0000000000400085 <+5>:	add    $0x2,%rax
   0x0000000000400089 <+9>:	mov    $0x3c,%eax
   0x000000000040008e <+14>:	mov    $0x0,%edi
   0x0000000000400093 <+19>:	syscall
End of assembler dump.
(gdb)
```

It comes handy to put a breakpoint on the first line, to see the initial status; after the first mov instruction, and after the add instruction:

```
(gdb) break _start
Breakpoint 1 at 0x400080: file main1.nasm, line 7.
(gdb) break 8
Breakpoint 2 at 0x400085: file main1.nasm, line 8.
(gdb) break 9
Breakpoint 3 at 0x400089: file main1.nasm, line 9.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400080 main1.nasm:7
2       breakpoint     keep y   0x0000000000400085 main1.nasm:8
3       breakpoint     keep y   0x0000000000400089 main1.nasm:9
```

Starting the program and checking the EAX register, which should contain the default value (which is zero):

```
(gdb) run
Starting program: /home/gbiondo/Development/sumsDiffs/main1a1

Breakpoint 1, _start () at main1.nasm:7
7	        mov rax,0x1
(gdb) info registers eax
eax            0x0	0
(gdb) info registers rax
rax            0x0	0
```

Taking a step, the first instruction gets executed, so the register is loaded with the value 1:

```
(gdb) s

Breakpoint 2, _start () at main1.nasm:8
8          add rax,0x2
(gdb) info registers eax
eax            0x1  1
(gdb) info registers rax
rax            0x1  1
```

We then run the second instruction, adding 2 to the contents of the RAX registry, we obtain:

```
(gdb) info registers eax
eax            0x3  3
(gdb) info registers rax
rax            0x3  3
(gdb)
```

This shows also how the EAX and RAX are interchangeable, in this context.

Let us add something more to the code to see some further functionalities. Consider the following:

```
global _start

section .text

_start:

    mov rax,0x1
    add rax,0x2

    mov rbx, rax
    add rbx, rbx
    add bl, byte [addr1]

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    addr1: db 0x23
```

(keep in mind that 0x23 = 35 in decimal)

The analysis with GDB is quite interesting. As usual we disassemble the code and add some breakpoints:

```
gbiondo@LinuxMint:~/Development/sumDiffs $ gdb main2
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main2...done.
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:      mov    eax,0x1
   0x00000000004000b5 <+5>:      add    rax,0x2
   0x00000000004000b9 <+9>:      mov    rbx,rax
   0x00000000004000bc <+12>:     add    rbx,rbx
   0x00000000004000bf <+15>:     add    bl,BYTE PTR ds:0x6000d4
   0x00000000004000c6 <+22>:     mov    eax,0x3c
   0x00000000004000cb <+27>:     mov    edi,0x0
   0x00000000004000d0 <+32>:     syscall
End of assembler dump.

(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004000b9 main2.nasm:9
2       breakpoint     keep y   0x00000000004000bc main2.nasm:11
3       breakpoint     keep y   0x00000000004000c6 main2.nasm:13
4       breakpoint     keep y   0x00000000004000bf main2.nasm:12
(gdb) run
```

We first move the value of RAX into RBX, hence RBX should have a value of 3; then we add to RBX the value of RBX, which should give us a result of 6; finally we add to RBX the contents of the memory address `0x6000d4`, which is the address of `addr1`. The result should be 41 (6 + 35).

We have:

```
Breakpoint 1, _start () at main2.nasm:10
10          mov rbx, rax
(gdb) i r rbx
rbx            0x0   0
(gdb) s

Breakpoint 2, _start () at main2.nasm:11
11          add rbx, rbx
(gdb) i r rbx
rbx            0x3   3
(gdb) s

Breakpoint 4, _start () at main2.nasm:12
12          add bl, byte [addr1]
(gdb) i r rbx
rbx            0x6   6
(gdb) s

Breakpoint 3, _start () at main2.nasm:14
14          mov rax, 60
(gdb) i r rbx
rbx            0x29 41
```

And results, here, are in line with the expectations.

Consider the following code, instead:

```
global _start

section .text

_start:

    mov cl, byte [addr2]
    sub cl, byte [addr1]

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    addr1: db 0x23
    addr2: db 0x17
```

GDB'ing it:

```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:     mov    cl,BYTE PTR ds:0x6000cd
   0x00000000004000b7 <+7>:     sub    cl,BYTE PTR ds:0x6000cc
   0x00000000004000be <+14>:    mov    eax,0x3c
   0x00000000004000c3 <+19>:    mov    edi,0x0
   0x00000000004000c8 <+24>:    syscall
End of assembler dump.
(gdb) break 8
Breakpoint 1 at 0x4000b7: file main3.nasm, line 8.
(gdb) break 9
Breakpoint 2 at 0x4000be: file main3.nasm, line 9.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004000b7 main3.nasm:8
2       breakpoint     keep y   0x00000000004000be main3.nasm:9
(gdb) run
Starting program: /home/gbiondo/Development/sumsDiffs/main3

Breakpoint 1, _start () at main3.nasm:8
8           sub cl, byte [addr1]
(gdb) i r cl
cl             0x17 23
(gdb) s

Breakpoint 2, _start () at main3.nasm:10
10          mov rax, 60
(gdb) i r cl
cl             0xf4 -12
(gdb)
```

Results are obviously in line with expectations. Interestingly, we worked only with the 'small registers'.

Flags

The FLAGS register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16 bits wide. Its successors, the EFLAGS and RFLAGS registers, are 32 bits and 64 bits wide, respectively. The wider registers retain compatibility with their smaller predecessors.
The fixed bits at bit positions 1, 3 and 5, and carry, parity, adjust, zero and sign flags.

The flags register is as follows:

Bit #	Mask	Abbreviation	Description	Notes
0	`0x0001`	`CF`	Carry flag	Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
1	`0x0002`		Reserved	
2	`0x0004`	`PF`	Parity flag	Set if the number of set bits in the least significant byte is a multiple of 2.
3	`0x0008`		Reserved	
4	`0x0010`	`AF`	Adjust flag	Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
5	`0x0020`		Reserved	
6	`0x0040`	`ZF`	Zero flag	Set if the result of an operation is Zero (0).
7	`0x0080`	`SF`	Sign flag	Set if the result of an operation is negative.
8	`0x0100`	`TF`	Trap flag	Set if step by step debugging.
9	`0x0200`	`IF`	Interrupt enable flag	Set if interrupts are enabled.
10	`0x0400`	`DF`	Direction flag	If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.
11	`0x0800`	`OF`	Overflow flag	Set if signed arithmetic operations result in a value too large for the register to contain.
12, 13	`0x3000`	`IOPL`	I/O privilege lvl.	I/O Privilege Level of the current process.
14	`0x4000`	`NT`	Nested task flag	Controls chaining of interrupts. Set if the current process is linked to the next process.
15	`0x8000`		Reserved	

In 32 and 64-bits architectures, the following flags are available as well:

Bit #	Abbreviation	Description	Notes
16	RF	Resume Flag	Response to debug exceptions.

Bit #	Abbreviation	Description	Notes
17	VM	Virtual-8086 Mode	Set if in 8086 compatibility mode.
18	AC	Alignment Check	Set if alignment checking of memory references is done.
19	VIF	Virtual Interrupt Flag	Virtual image of IF.
20	VIP	Virtual Interrupt Pending flag.	Set if an interrupt is pending.
21	ID	Identification Flag	Support for CPUID instruction if can be set.

Consider the following piece of code:

```
global _start

section .text

_start:

    mov rax, 0x0
    stc
    adc rax, 0x0
    stc
    adc rax, 0x0
    stc
    stc

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

We prepare for the execution as follows:

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    eax,0x0
   0x0000000000400085 <+5>:      stc
   0x0000000000400086 <+6>:      adc    rax,0x0
   0x000000000040008a <+10>:     stc
   0x000000000040008b <+11>:     adc    rax,0x0
   0x000000000040008f <+15>:     stc
   0x0000000000400090 <+16>:     stc
   0x0000000000400091 <+17>:     mov    eax,0x3c
   0x0000000000400096 <+22>:     mov    edi,0x0
   0x000000000040009b <+27>:     syscall
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400080 jumpingJack.nasm:7
2       breakpoint     keep y   0x0000000000400085 jumpingJack.nasm:8
3       breakpoint     keep y   0x0000000000400086 jumpingJack.nasm:9
4       breakpoint     keep y   0x000000000040008a jumpingJack.nasm:10
5       breakpoint     keep y   0x000000000040008b jumpingJack.nasm:11
6       breakpoint     keep y   0x000000000040008f jumpingJack.nasm:12
7       breakpoint     keep y   0x0000000000400090 jumpingJack.nasm:13
8       breakpoint     keep y   0x0000000000400091 jumpingJack.nasm:14
```

The following values have been obtained by invoking at every single breakpoints the following commands:

```
(gdb) i r eax
(gdb) p $eflags
(gdb) p/t $eflags
```

Finally, the instruction STC sets the Carry Flag, taking no operand; whereas in its most basic form, the instruction ADC takes a Register R and an Operand O, and performs the Add With Carry operation as follows: `R = R + O + CF`.

we have:

Breakpoint	i r eax			p $eflags	p/t $eflags
1	eax	0x0	0	[IF]	1000000010
2	eax	0x0	0	[IF]	1000000010
3	eax	0x0	0	[CF IF]	1000000011
4	eax	0x1	1	[IF]	1000000010
5	eax	0x1	1	[CF IF]	1000000011
6	eax	0x2	2	[IF]	1000000010
7	eax	0x2	2	[CF IF]	1000000011
8	eax	0x2	2	[CF IF]	1000000011

Observe the following:
1. After an ADC operation, the CF is reset to 0
2. Two subsequent STC operation don't reset CF values.

More fun with Arithmetics

Consider the following code:

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    eax,0x17
   0x0000000000400085 <+5>:      stc
   0x0000000000400086 <+6>:      sbb    rax,0x22
   0x000000000040008a <+10>:     mov    eax,0x3c
   0x000000000040008f <+15>:     mov    edi,0x0
   0x0000000000400094 <+20>:     syscall
```

The instruction SBB is the Integer Subtraction with Borrow. In its most basic syntax, it takes a register (the destination, DEST) and subtracts from it the Operand (SRC) and the Carry Flag (CF) summed together: `DEST = DEST – (SRC + CF)`. In this case the expected result is 0x17 (which is 23) minus the sum of 0x22 (which is 34) and the carry flag (1). Shortly, we expect RAX to contain 23-(34+1) = -12.

Let's debug. The initial values are

```
(gdb) i r eax
eax            0x0  0
(gdb) p $eflags
$1 = [ IF ]
(gdb) p/t $eflags
$2 = 1000000010
```

After performing the initialisation of RAX, and after having set the carry flag, we have

```
(gdb) i r eax
eax            0x17 23
(gdb) p $eflags
$6 = [ CF IF ]
(gdb) p/t $eflags
$7 = 1000000011
```

Performing the subtraction gives the following results:

```
(gdb) i r eax
eax            0xfffffff4  -12
(gdb) p $eflags
$8 = [ CF SF IF ]
(gdb) p/t $eflags
$9 = 1010000011
```

which is interesting, as it shows:

a) the way negative numbers are represented. We'll come back on this later.
b) the carry flag is not affected by `SBB`
c) the operation sets the Sign Flag (`SF`).

Memory representation of negative numbers

In the last example, we saw first EAX loaded with `0x00000017`, and then with `0xFFFFFFF4` to represent -12. What happened?

Value	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
17	0	0	0	0	0	0	1	7
12	0	0	0	0	0	0	1	2
-12	F	F	F	F	F	F	F	4

The negative numbers are represented with the so called '2 complement'.

In binary, 12 is 00001100. First we calculate the 1-complement, which is obtained by swapping all 0s with 1s and vice versa. The result is quite straightforward.

Increment and decrement values

Consider the following piece of code:

```
 0x0000000000400080 <+0>:  mov    eax,0x3b
   0x0000000000400085 <+5>:      inc    rax
   0x0000000000400088 <+8>:      inc    rax
   0x000000000040008b <+11>:     inc    rax
   0x000000000040008e <+14>:     dec    rax
   0x0000000000400091 <+17>:     dec    rax
   0x0000000000400094 <+20>:     mov    edi,0x0
   0x0000000000400099 <+25>:     syscall
```

When debugging it:

```
(gdb) i breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400085 incDec.asm:8
2       breakpoint     keep y   0x0000000000400088 incDec.asm:9
3       breakpoint     keep y   0x000000000040008b incDec.asm:10
4       breakpoint     keep y   0x000000000040008e incDec.asm:11
5       breakpoint     keep y   0x0000000000400091 incDec.asm:12
6       breakpoint     keep y   0x0000000000400094 incDec.asm:13
...
(gdb) r
Starting program: /home/gbiondo/Development/incsDecs/incDec

Breakpoint 1, _start () at incDec.asm:8
8          inc rax
(gdb) i r eax
eax            0x3b 59
(gdb) s

Breakpoint 2, _start () at incDec.asm:9
9          inc rax
(gdb) i r eax
eax            0x3c 60
(gdb) s

Breakpoint 3, _start () at incDec.asm:10
10         inc rax
(gdb) i r eax
eax            0x3d 61
(gdb) s

Breakpoint 4, _start () at incDec.asm:11
11         dec rax
(gdb) i r eax
eax            0x3e 62
(gdb) s

Breakpoint 5, _start () at incDec.asm:12
12         dec rax
(gdb) i r eax
eax            0x3d 61
(gdb) s

Breakpoint 6, _start () at incDec.asm:14
14         mov rdi, 0
(gdb) i r eax
eax            0x3c 60
(gdb) s
15         syscall
(gdb) s
[Inferior 1 (process 13422) exited normally]
(gdb)
```

Here EAX, which also contains the SysCall ID, is already set to 60, so we can invoke directly the API.

We have shown the INC and DEC instructions that take as operand a register, and respectively add and subtract 1 from it. The state of CF is always preserved.

Integer multiplication

Assembly implements two kind of multiplications, signed and unsigned. Let us begin by the unsigned multiplication, which in its simplest form has the syntax:

```
mul <src> <fct>
```

in which case, an 'A' register (AL, AX, EAX, RAX) must be used for the operands.

The results follow a general rule: if the operands are n bits, the result will be $2n$ bits. It follows that the result may need two registers to be stored, in fact in the single operand case the results are stored in the A and the D registers, as follows:

First operator	Second operator	First part of the result	Second part of the result
AL	8 bits operator	AH	AL
AX	16 bits operator	DX	AX
EAX	32 bits operator	EDX	EAX
RAX	64 bits operator	RDX	RAX

Consider the following piece of code:

```
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    al,0x33
   0x0000000000400082 <+2>:      mov    bl,0x32
   0x0000000000400084 <+4>:      mul    bl
   0x0000000000400086 <+6>:      mov    eax,0x3c
   0x000000000040008b <+11>:     mov    edi,0x0
   0x0000000000400090 <+16>:     syscall
```

We are multiplying two 8-bits values so the result is potentially a 16-bits value.

```
Breakpoint 1, _start () at mul.asm:6
6             mov    al,'3'
(gdb) s
8             mov    bl, '2'
(gdb) s
9             mul    bl
(gdb) i r al
al              0x33 51
(gdb) i r bl
bl              0x32 50
(gdb) s
11            mov rax, 0x3C
```

Something is already strange. The registers have been loaded with `0x33` and `0x32`, respectively. These are the ASCII codes of 3 and 2, respectively, so something did not work out properly - we will come back on this later, anyway. Since both 51 and 50 are numbers that can be represented with a byte, the multiplication should work as previously described. In fact, EAX contains the correct result of the operation:

```
(gdb) i r bl
bl              0x32 50
(gdb) i r al
al              0xf6 -10
(gdb) i r eax
eax             0x9f6       2550
```

The problem of the 'wrong' values was that we assigned the values of the chars '2' and '3' (note the hyphens).

How can we convert these chars into integers? The next example gives an idea:

```
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    al,0x33 ;'3'
   0x0000000000400082 <+2>:      sub    al,0x30 ;'0'...
   0x0000000000400084 <+4>:      mov    bl,0x32
   0x0000000000400086 <+6>:      sub    bl,0x30
   0x0000000000400089 <+9>:      mul    bl
   0x000000000040008b <+11>:     mov    eax,0x3c
   0x0000000000400090 <+16>:     mov    edi,0x0
   0x0000000000400095 <+21>:     syscall
```

Integer division (`div`)

If $b \neq 0$, the division between two integers a (the dividend) and b (the divisor) is defined as:

$a/b = q + r$

where q is the quotient and r is the reminder. These values may also be null.

Dividend and Divisor size (bytes)	Functioning
2 and 1	It is assumed that the dividend is in AX. Quotient is stored in AL Remainder is stored in AH

Dividend and Divisor size (bytes)	Functioning
4 and 2	It is assumed that the dividend is in DX and in AX. Quotient is stored in AX Remainder is stored in DX
8 and 4	It is assumed that the dividend is in EDX and in EAX. Quotient is stored in EAX Remainder is stored in EDX

The example is not so different from the one regarding multiplication:

```
section      .text
global _start

_start:

       mov ax, 23

       mov bl, 8
       div bl

       mov rax, 0x3C
       mov rdi, 0
       syscall

section .data
```

so is the debugging:

```
6             mov ax, 23
(gdb) s
8             mov bl, 8
(gdb) s
9             div bl
(gdb) i r ax
ax              0x17 23
(gdb) i r al
al              0x17 23
(gdb) i r bx
bx              0x8  8
(gdb) p $eflags
$1 = [ IF ]
(gdb) s
11            mov rax, 0x3C
(gdb) s
11            mov rax, 0x3C
(gdb) i r al
al              0x2  2
(gdb) i r ah
ah              0x7  7
(gdb) p $eflags
$2 = [ IF ]
```

The result is coherent: the result (2, in AL) times the divisor (8, in BL) equals 16, and 16 plus the remainder (7, in AH) equals 23 (the dividend, in AX).

With this knowledge under our belt, we can start talking about Loops.

Controlling the program flow
Loops

The `for` loop

Loops, in assembly, are implemented by using the instruction '`loop`', unsurprisingly. The instruction provides a simple way to repeat a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats.

The syntax is quite easy:

```
loop <label>
```

being the argument a text label in the line of code. The execution of the statement involves two steps:

1. Subtracts 1 from ECX
2. Compares ECX to 0:
   1. If the comparison fails, the program jumps back to `<label>`
   2. If the comparison succeeds, the program flows passes to the instruction following the `loop` instruction.

An example helps understanding better:

```
global _start
section .text
_start:
       mov rcx, 10
       mov rax, 1

lbl:
       inc rax
       loop lbl

       mov rax, 60
       mov rdi, 0
       syscall

section .data
```

Disassembling may seem tricky

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    ecx,0xa
   0x0000000000400085 <+5>:      mov    eax,0x1
End of assembler dump.
```

This is because we declared a label, thus:

```
(gdb) disassemble lbl
Dump of assembler code for function lbl:
   0x000000000040008a <+0>:      inc    rax
   0x000000000040008d <+3>:      loop   0x40008a <lbl>
   0x000000000040008f <+5>:      mov    eax,0x3c
   0x0000000000400094 <+10>:     mov    edi,0x0
   0x0000000000400099 <+15>:     syscall
End of assembler dump.
```

Setting some breakpoints:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
7       breakpoint     keep y   0x0000000000400080 loop.asm:4
8       breakpoint     keep y   0x0000000000400085 loop.asm:5
9       breakpoint     keep y   0x000000000040008a loop.asm:8
10      breakpoint     keep y   0x000000000040008d loop.asm:9
11      breakpoint     keep y   0x000000000040008f loop.asm:10
```

As usual, we debug the application. The registers are initialised with zeros, as usual:

```
(gdb) i r ecx
ecx            0x0  0
(gdb) i r eax
eax            0x0  0
```

but when entering the loop, these are properly initialised:

```
(gdb) i r ecx
ecx            0xa  10
(gdb) i r eax
eax            0x1  1
```

The body of the loop is constituted by the inc and the loop statements, so we will just report the steps

End of step #	`i r eax`	`i r ecx`
1	`eax 0x2 2`	`ecx 0x9 9`
2	`eax 0x3 3`	`ecx 0x8 8`
3	`eax 0x4 4`	`ecx 0x7 7`
4	`eax 0x5 5`	`ecx 0x6 6`
…	…	…
9	`eax 0xa 10`	`ecx 0x1 1`
10	`eax 0xa 11`	`ecx 0x1 0`

After this last iteration, the program flows returns and the `exit` syscall is invoked.

Infinite loops

Never modify the value of the RCX value during a loop, it could have unpredictable effects. In the next example, we show some code that leads to a never-ending loop ("infinite loop").

```
_start:

      mov rcx, [threshold]

infiniteLoop:
      mov r10, 1
      inc rcx
      loop infiniteLoop

      mov rax, [sys_call]
      mov rdi, [exit_code]
      syscall

section .data
      threshold dq 3
      exit_code dq 0
      sys_call dq 60
```

The reason being the fact that the counter register RCX gets incremented in every cycle, thus it. will never reach 0.

In fact:

```
Breakpoint 1, infiniteLoop () at buggy.asm:8
8             mov r10, 1
(gdb) i r rcx
rcx             0x3  3
(gdb) s
9             inc rcx
(gdb) i r rcx
rcx             0x3  3
(gdb) s
10            loop infiniteLoop
(gdb) i r rcx
rcx             0x4  4
(gdb) s

Breakpoint 1, infiniteLoop () at buggy.asm:8
8             mov r10, 1
(gdb) i r rcx
rcx             0x3  3
(gdb) ...
```

Beware that also syscalls may change the value of the counter register. In fact, let's get back to our old Hello World:

```
global _start

section .text

_start:
       mov rax, 1
       mov rdi, 1
       mov rsi, hello_world
       mov rdx, length
       syscall
       mov rax, 0x3C
       syscall


section .data
       hello_world: db 'hello world',0xa
       length: equ $-hello_world
```

Let's set some breakpoints and see registers' contents:

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:      mov    eax,0x1
   0x00000000004000b5 <+5>:      mov    edi,0x1
   0x00000000004000ba <+10>:     movabs rsi,0x6000d4
   0x00000000004000c4 <+20>:     mov    edx,0xc
   0x00000000004000c9 <+25>:     syscall
   0x00000000004000cb <+27>:     mov    eax,0x3c
   0x00000000004000d0 <+32>:     syscall
End of assembler dump.
(gdb) break 10
Breakpoint 5 at 0x4000c9: file hello2.asm, line 10.
(gdb) break 11
Breakpoint 6 at 0x4000cb: file hello2.asm, line 11.
...
Breakpoint 5, _start () at hello2.asm:10
10             syscall
(gdb) i r
rax            0x1  1
rbx            0x0  0
rcx            0x0  0
rdx            0xc  12
rsi            0x6000d4    6291668
rdi            0x1  1
rbp            0x0  0x0
rsp            0x7fffffffe450    0x7fffffffe450
... Breakpoint 6, _start () at hello2.asm:11
11             mov rax, 0x3C
(gdb) i r
rax            0xc  12
rbx            0x0  0
rcx            0x4000b4    4194484
rdx            0xc  12
rsi            0x6000d4    6291668
rdi            0x1  1
rbp            0x0  0x0
rsp            0x7fffffffe450    0x7fffffffe450
...
```

The debug clearly shows how the API has changed RCX register's contents.

Putting it all together 1: Fibonacci sequence

The Fibonacci sequence is recursively defined as follows:

$$a_n := \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ a_{n-1} + a_{n-2} & n > 2 \end{cases}$$

Its elements are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on. We wanted to write an assembly program that calculates the 8th term of this sequence (which is 55).

We have written the following code:

```
global _start
section .text
_start:

       mov rax, 1
       mov rbx, 1
       mov rcx, 8
       mov r10, 0

fiboloop:
       mov r10, rbx
       add rbx, rax
       mov rax, r10
       loop fiboloop

       mov rax, 60
       mov rdi, 0
       syscall

section .data
```

The algorithm is quite straightforward: the start section initialises to 1 RBX and RAX (which, in our case, will always be respectively a_{n-1} and a_{n-2}, and a buffer variable (`R10`) that will be used to save the value of RBX when swapping variables.

We then start a loop in which:
1. We save the value of RBX in `R10`
2. We increase the value of RBX with the contents of RAX
3. We update the value of RAX, putting into it the old RBX value (now loaded in `R10`)

It is immediate noticing that this algorithm converges to the intended result.

Debugging this is quite funny. Here we give only the first and the last iterations:

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    eax,0x1
   0x0000000000400085 <+5>:      mov    ebx,0x1
   0x000000000040008a <+10>:     mov    ecx,0x8
   0x000000000040008f <+15>:     mov    r10d,0x0
End of assembler dump.
(gdb) disassemble fiboloop
Dump of assembler code for function fiboloop:
   0x0000000000400095 <+0>:      mov    r10,rbx
   0x0000000000400098 <+3>:      add    rbx,rax
   0x000000000040009b <+6>:      mov    rax,r10
   0x000000000040009e <+9>:      loop   0x400095 <fiboloop>
   0x00000000004000a0 <+11>:     mov    eax,0x3c
   0x00000000004000a5 <+16>:     mov    edi,0x0
   0x00000000004000aa <+21>:     syscall
End of assembler dump.

(gdb) i breakpoints
Num     Type           Disp Enb Address            What
5       breakpoint     keep y   0x0000000000400095 fibonacci.asm:11
6       breakpoint     keep y   0x000000000040009e fibonacci.asm:14
7       breakpoint     keep y   0x00000000004000a0 fibonacci.asm:15

(gdb) i r
rax            0x1  1
rbx            0x1  1
rcx            0x8  8
rdx            0x0  0
rsi            0x0  0
rdi            0x0  0
rbp            0x0  0x0
rsp            0x7fffffffe450    0x7fffffffe450
r8             0x0  0
r9             0x0  0
r10            0x0  0
…
rip            0x400095   0x400095 <fiboloop>
eflags         0x202      [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
...
(gdb) i r
rax            0x15 21
rbx            0x22 34
rcx            0x1  1
rdx            0x0  0
rsi            0x0  0
rdi            0x0  0
rbp            0x0  0x0
rsp            0x7fffffffe450    0x7fffffffe450
r8             0x0  0
r9             0x0  0
r10            0x15 21
…
rip            0x400095   0x400095 <fiboloop>
eflags         0x216      [ PF AF IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

This code is not perfect, at least in terms of flexibility (we could have added a threshold constant, for instant, to make it more maintainable), but shows how easy could be solving a problem in assembly.

This can be solved rewriting the code as follows (we also had fun with the syscalls). Up to the reader debugging it.

```
global _start
section .text
_start:

       mov rax, 1
       mov rbx, 1
       mov rcx, [threshold]
       mov r10, 0

fiboloop:
       mov r10, rbx
       add rbx, rax
       mov rax, r10
       loop fiboloop

       mov rax, [sys_call]
       mov rdi, [exit_code]
       syscall

section .data
       threshold dq 8
       exit_code dq 0
       sys_call dq 60
```

Jumping around the code

A jump is basically a way to skip some parts of the code, or going back in the execution.

Unconditional jump

The simplest form of jump is called unconditional jump, it executes no matter what are the side conditions. The syntax is

```
jmp <label>
```

the effect is executing the first statement after the textual `label`.

A simple example follows:

```
global _start

section .text

_start:
        mov rax, 0x3C           ;we already prepare the exit syscall

        jmp exitLabel

        mov rdi, 123            ;this instruction won't ever be executed

exitLabel:
        mov rdi, 0
        syscall


section .data
```

You may have notice the presence of some plain English text after the semicolumn. These are comments, and never get considered by the compiler.

The flow of the program is as follows:

```
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:      mov    eax,0x3c
   0x0000000000400085 <+5>:      jmp    0x40008c <exitLabel>
   0x0000000000400087 <+7>:      mov    edi,0x7b
End of assembler dump.
(gdb) disassemble exitLabel
Dump of assembler code for function exitLabel:
   0x000000000040008c <+0>:      mov    edi,0x0
   0x0000000000400091 <+5>:      syscall
End of assembler dump.
(gdb) break _start
Breakpoint 1 at 0x400080: file jump1.asm, line 6.
(gdb) r
Starting program: /home/gbiondo/Development/LX_012_jumps/jump1

Breakpoint 1, _start () at jump1.asm:6
6             mov rax, 0x3C       ;we already prepare the exit syscall
(gdb) s
8             jmp exitLabel
(gdb) i r edi
edi            0x0  0
(gdb) s
exitLabel () at jump1.asm:13
13            mov rdi, 0
(gdb) ...
```

It is evident how the instruction located at <+7> does not get executed. We can cross check that the exit value is, obviously, 0:

```
gbiondo@LinuxMint:~/Development/LX_012_jumps$ ./jump1
gbiondo@LinuxMint:~/DevelopmentLX_012_jumps$ echo $?
0
```

Jump if below (`jb`)

This is the first form of conditional jumps. It makes the control flow jumping if `CF = 1`.

Used after a CMP or SUB instruction, JB transfers control to the label if the first operand was less than the second. (Both operands are treated as unsigned numbers).

Consider the following example:

```
global _start

section .text

_start:
       mov rax, 0x3C       ;we already prepare the exit syscall
       mov rdi, 0          ;initializing the exit code
       jb exitLabel        ;this won't be executed...
       mov rdi, 123        ;changing the exit code
       stc                 ;we set the carry flag
       jb exitLabel

exitLabel:
       syscall


section .data
```

Once again, disassembling shows how the program flow works:

```
Breakpoint 1, _start () at jump2.asm:6
6             mov rax, 0x3C       ;we already prepare the exit syscall
(gdb) s
7             mov rdi, 0          ;initializing the exit code
(gdb) s
8             jb exitLabel        ;this won't be executed...
(gdb) s
9             mov rdi, 123        ;changing the exit code
(gdb) p $eflags
$1 = [ IF ]
(gdb) s
10            stc                 ;we set the carry flag
(gdb) p $eflags
$2 = [ IF ]
(gdb) s
11            jb exitLabel
(gdb) p $eflags
$3 = [ CF IF ]
(gdb) s
exitLabel () at jump2.asm:14
14            syscall
(gdb) p $eflags
$4 = [ CF IF ]
(gdb) s
[Inferior 1 (process 8253) exited with code 0173]
(gdb) q
gbiondo@LinuxMint:~/Development/LX_012_jumps$ ./jump2
gbiondo@LinuxMint:~/DevelopmentLX_012_jumps$ echo $?
123
```

Jump if not sign (jns)

This statement performs the jump to the label if and only if the sign flag (**SF**) is null. The example clarifies.

```
global _start

section .text

_start:

	mov al, 0x2
	sub al, 0x1

land2:
	jns land1
	mov rdi, 0x123

land1:
	sub al, 0x2
	jns land2

	mov rax, 60
	mov rdi, 10
	syscall

section .data
```

Below, the program flow obtained with the debugger:

```
Breakpoint 1, _start () at jump3.asm:7
7            mov al, 0x2
(gdb) s
8            sub al, 0x1
(gdb) s
land2 () at jump3.asm:10
10           jns land1
(gdb) i r al
al             0x1  1
(gdb) p $eflags
$1 = [ IF ]
(gdb) s
land1 () at jump3.asm:14
14           sub al, 0x2
(gdb)
15           jns land2
(gdb) p $eflags
$2 = [ CF PF AF SF IF ]
(gdb) i r al
al             0xff -1
(gdb) s
17           mov rax, 60
(gdb)
18           mov rdi, 10
(gdb)
19           syscall
```

As expected, the first jump is executed (2>1, and hence the subtraction does not set the Sign Flag), whereas the second jump is not.

Other conditional jumps

There are several other conditional jumps to take into consideration. We start with Jump if Equal (`je`). Consider the following code:

```
global _start

section .text

_start:

	mov rcx, 0x23
	mov rdx, 0x173
	mov rdi, 100

	cmp rcx, rdx
	je first_label

	cmp rdx, rcx
	je second_label

	cmp rcx, 0x23
	je third_label

first_label:
	mov rdi, 1
	jmp vaivia

second_label:
	mov rdi, 2
	jmp vaivia

third_label:
	mov rdi, 3
	jmp vaivia

vaivia:
	mov rax, 60
	syscall

section .data
```

The CMP statement compares the value of a register and another or a register and a constant and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction.

We will follow first the flow of the code, to determine generic rules for all conditional jumps. The first block of operations is assignments and comparing two different numbers. We do not expect that je will be executed.

```
Breakpoint 1, _start () at je.asm:7
7             mov rcx, 0x23
(gdb) s
8             mov rdx, 0x173
(gdb) s
9             mov rdi, 100
(gdb) s
11            cmp rcx, rdx
(gdb) s
12            je first_label
(gdb) i r rcx rdx rdi rip
rcx            0x23 35
rdx            0x173      371
rdi            0x64 100
rip            0x400092   0x400092 <_start+18>
(gdb) p $eflags
$1 = [ CF SF IF ]
```

Observe that `CMP` leverages `SUB`, hence the Carry and the Sign flags are set (the first operand is smaller than the second, hence a negative number is returned). Similarly, also the conditional jump in the following block of code won't be executed:

```
(gdb) s
14          cmp rdx, rcx
(gdb) s
15          je second_label
(gdb) i r rcx rdx rdi rip
rcx            0x23 35
rdx            0x173      371
rdi            0x64 100
rip            0x400097   0x400097 <_start+23>
(gdb) p $eflags
$2 = [ PF IF ]
```

(the parity flag is set because 371-35 gives an even result). There is the last part of code:

```
(gdb) s
17          cmp rcx, 0x23
(gdb) s
18          je third_label
(gdb) s
third_label () at je.asm:29
29          mov rdi, 3
(gdb) i r rcx rdx rdi rip
rcx            0x23 35
rdx            0x173      371
rdi            0x64 100
rip            0x4000ad   0x4000ad <third_label>
(gdb) p $eflags
$3 = [ PF ZF IF ]
(gdb)
```

In this case, the jump is executed (see `RIP`'s value), as the difference between the contents of RCX and `0x23` is `0` (differently said, they're the same value).

There are several other conditions that can be used. These are listed in the tables below. The reader is suggested to test these conditional jumps using this methodology.

Arithmetic operations

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/ Equal	`OF, SF, ZF`
JGE/JNL	Jump Greater/Equal or Jump Not Less	`OF, SF`
JL/JNGE	Jump Less or Jump Not Greater/ Equal	`OF, SF`
JLE/JNG	Jump Less/Equal or Jump Not Greater	`OF, SF, ZF`

Logical operations

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/ Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/ Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

Flag-related operations

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

The Stack

The stack, in Computer Science, is nothing more than an Abstract Data Type (ADT) that collects coherent elements, and supports (only) two operations:

- push operations: an element is added to the collection
- pop operations: the most recently added element is removed from the collection

From the definition, it emerges that the Stack works in the 'LIFO' mode (Last In, First Out).

In the computer implementation, each thread has a reserved region of memory which is referred as its stack, for when a function executes, it may add some data to this ADT, removing when the function exits. At a minimum, a thread's stack is used to store the location of function calls in order to allow return statements to return to the correct location.

Another relevant aspect is that the stack is an area of memory with a fixed origin and a variable size. Initially the size of the stack is zero. The stack pointer (hardware register), points to the most recently referenced location on the stack. When the stack null size, the pointer obviously points to the origin.

The stack space is located just under the OS kernel space (more on this later), generally opposite the heap area and grows downwards to lower addresses. (it may grow the opposite direction on some other architectures).

Stack area is devoted to storing all the data needed by a function call in a program. Calling a function is the same as pushing the called function execution onto the top of the stack, and once that function completes, the results are returned popping the function off from the stack.
The dataset pushed for function call is named a stack frame, and it contains the following data:

- the arguments (parameter values) passed to the routine
- the return address back to the routine's caller
- space for the local variables of the routine

push: data is placed at the location pointed to by the stack pointer, and the address in the stack pointer is adjusted by the size of the data item
pop (or pull): data at the current location pointed to by the stack pointer is removed (and usually put in a special register), the stack pointer is adjusted accordingly

The stack is the memory set aside as scratch space for a thread of execution.

When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When the function returns, the block becomes unused and can be used the next time a function is called. Mainly, the stack stores local data and return addresses used for parameter passing.

Let us see a practical implementation. Consider the below code:

```
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:     mov    eax,0x12345678
   0x0000000000400085 <+5>:     push   rax
   0x0000000000400086 <+6>:     push   0x9
   0x0000000000400088 <+8>:     push   0x1234
   0x000000000040008d <+13>:    pop    rax
   0x000000000040008e <+14>:    pop    rcx
   0x000000000040008f <+15>:    pop    rbx
   0x0000000000400090 <+16>:    mov    eax,0x3c
   0x0000000000400095 <+21>:    mov    edi,0x0
   0x000000000040009a <+26>:    syscall
End of assembler dump.
```

We are using the instruction

```
x/4xw $sp
```

Whose meaning is as follows:

`x` - examine
`4xv` - four (`4`, but it may be more) words (`w`) of memory above the stack pointer (here, `$sp`) in hexadecimal (`x`).

At the beginning, the situation is as follows

Registers	Stack				
rax 0	0x7fffffffe4c0:	0x00000001	0x00000000	0xffffe70b	0x00007fff
rax 305419896	0x7fffffffe4b8:	0x12345678	0x00000000	0x00000001	0x00000000
…	0x7fffffffe4b0:	0x00000009	0x00000000	0x12345678	0x00000000
…	0x7fffffffe4a8: 0x7fffffffe4b8:	0x00001234 0x12345678	0x00000000 0x00000000	0x00000009 0x00000001	0x00000000 0x00000000
rax 4660	0x7fffffffe4b0: 0x7fffffffe4c0:	0x00000009 0x00000001	0x00000000 0x00000000	0x12345678 0xffffe70b	0x00000000 0x00007fff
rax 4660 rbx 305419896	0x7fffffffe4b8: 0x7fffffffe4c8:	0x12345678 0xffffe70b	0x00000000 0x00007fff	0x00000001 0x00000000	0x00000000 0x00000000
rax 4660 rbx 305419896 rcx 9	0x7fffffffe4c0: 0x7fffffffe4d0:	0x00000001 0x00000000	0x00000000 0x00000000	0xffffe70b 0xffffe72b	0x00007fff 0x00007fff
…	…				

Note how the stack shrunk and enlarged during the program run.

Logical operations

We are now examining the bitwise logical operations. Assembly on Linux implements the following logical operators:

- `AND`
- `OR`
- `NOT`
- `TEST`
- `XOR`

We are going to see how they work shortly. It is beneficial to remember the definitions of these operands:

A	B	NOT A	A AND B	A OR B	A XOR B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Bitwise operation means that each operand is analysed at a bit level, for instance:

A	0	1	0	1	0	0	1	0
B	1	1	1	0	0	1	1	1
A | B	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**
A&B	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**

Let's dive down to the code.

In this exercise we will use the values in the table below; we are also computing the results manually to cross check.

RAX	1	0	1	1	1	0	1	1
RBX	1	1	0	1	0	1	1	0
RAX AND RBX	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**
RAX OR RBX	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**
RAX XOR RBX	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**1**
NOT RAX	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**

Consider then this code:

```
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:	mov    eax,0x10111011
   0x0000000000400085 <+5>:	mov    ebx,0x11010110
   0x000000000040008a <+10>:	and    rax,rbx
   0x000000000040008d <+13>:	mov    eax,0x10111011
   0x0000000000400092 <+18>:	mov    ebx,0x11010110
   0x0000000000400097 <+23>:	or     rax,rbx
   0x000000000040009a <+26>:	mov    eax,0x10111011
   0x000000000040009f <+31>:	mov    ebx,0x11010110
   0x00000000004000a4 <+36>:	xor    rax,rbx
   0x00000000004000a7 <+39>:	mov    eax,0x10111011
   0x00000000004000ac <+44>:	not    rax
   0x00000000004000af <+47>:	mov    eax,0x10111011
   0x00000000004000b4 <+52>:	mov    ebx,0x11010110
   0x00000000004000b9 <+57>:	test   rax,rbx
   0x00000000004000bc <+60>:	mov    eax,0x3c
   0x00000000004000c1 <+65>:	mov    edi,0xa
   0x00000000004000c6 <+70>:	syscall
```

The result of the AND operation is stored again in RAX, as shown below:

```
10          and rax, rbx
(gdb) i r rax rbx rip eflags
rax            0x10111011  269553681
rbx            0x11010110  285278480
rip            0x40008a    0x40008a <_start+10>
eflags         0x202       [ IF ]
(gdb) s
12          mov rax,0x10111011
(gdb) i r rax rbx rip eflags
rax            0x10010010  268501008
rbx            0x11010110  285278480
rip            0x40008d    0x40008d <_start+13>
eflags         0x202       [ IF ]
```

The second operation behaves in the same way.

```
15          or rax, rbx
(gdb) i r rax rbx rip eflags
rax            0x10111011  269553681
rbx            0x11010110  285278480
rip            0x400097    0x400097 <_start+23>
eflags         0x202       [ IF ]
(gdb) s
17          mov rax,0x10111011
(gdb) i r rax rbx rip eflags
rax            0x11111111  286331153
rbx            0x11010110  285278480
rip            0x40009a    0x40009a <_start+26>
eflags         0x206       [ PF IF ]
```

As for the XOR, once again the result is stored in RAX:

```
(gdb) i r rax rbx rip eflags
rax            0x10111011  269553681
rbx            0x11010110  285278480
rip            0x4000a4    0x4000a4 <_start+36>
eflags         0x206       [ PF IF ]
(gdb) s
22          mov rax,0x10111011
(gdb) i r rax rbx rip eflags
rax            0x1101101   17830145
rbx            0x11010110  285278480
rip            0x4000a7    0x4000a7 <_start+39>
eflags         0x202       [ IF ]
```

The NOT operator behaves in the same manner:

```
23           not rax
(gdb) i r rax rbx rip eflags
rax            0x10111011  269553681
rbx            0x11010110  285278480
rip            0x4000ac    0x4000ac <_start+44>
eflags         0x202       [ IF ]
(gdb) s
25           mov rax,0x10111011
(gdb) i r rax rbx rip eflags
rax            0xffffffffefeeefee-269553682
rbx            0x11010110  285278480
rip            0x4000af    0x4000af <_start+47>
eflags         0x202       [ IF ]
```

The TEST operator, finally, behaves like AND and it is not discussed here.

Subroutines

Subroutine definition follows this syntax:

```
subroutine_name:
      <statements>
      ret
```

Subroutines must be defined just after the `_start` section, they begin with a symbolic name, and end with the `ret` statement.

Consider the following piece of code:

```
global _start

section .text

addition:

      mov eax, ecx
      add eax, edx
      ret

_start:

    mov ecx, 0x14
    mov edx, 0x15
    call addition

    mov r8,0x4
    mov r9, 0x2
    call addition

    mov rax, 60
    mov rdi, 1
    syscall

section .data
```

Into GDB we have:

```
Dump of assembler code for function _start:
   0x0000000000400085 <+0>:      mov    ecx,0x14
   0x000000000040008a <+5>:      mov    edx,0x15
   0x000000000040008f <+10>:     call   0x400080 <addition>
   0x0000000000400094 <+15>:     mov    r8d,0x4
   0x000000000040009a <+21>:     mov    r9d,0x2
   0x00000000004000a0 <+27>:     call   0x400080 <addition>
   0x00000000004000a5 <+32>:     mov    eax,0x3c
   0x00000000004000aa <+37>:     mov    edi,0x1
   0x00000000004000af <+42>:     syscall
End of assembler dump.
(gdb) disassemble addition
Dump of assembler code for function addition:
   0x0000000000400080 <+0>:      mov    eax,ecx
   0x0000000000400082 <+2>:      add    eax,edx
   0x0000000000400084 <+4>:      ret
End of assembler dump.
```

It is important that the reader now puts attention also on statements' addresses, since we will take into consideration also the RSP register, which points to the address of the next instruction to be executed.

```
Breakpoint 1, _start () at subs.asm:13
13          mov ecx, 0x14
(gdb) i r
...
rcx            0x14 20
...
rsp            0x7fffffffe480    0x7fffffffe480
...rip            0x400085 0x400085 <_start>
eflags         0x202       [ IF ]
...
```

This is correct, since the first instruction has not been run yet. Observe that first instruction to be ran is the first after the RET.

```
14          mov edx, 0x15
(gdb) i r
...
rcx            0x14 20
...
rsp            0x7fffffffe480    0x7fffffffe480
...
rip            0x40008a    0x40008a <_start+5>
eflags         0x202       [ IF ]
...
(gdb) s
15          call addition
(gdb) info registers
...
rcx            0x14 20
rdx            0x15 21
...
rsp            0x7fffffffe480    0x7fffffffe480
...
rip            0x40008f    0x40008f <_start+10>
eflags         0x202       [ IF ]
...
(gdb) s
addition () at subs.asm:7
7            mov eax, ecx
(gdb) info registers
...
rcx            0x14 20
rdx            0x15 21
...
rsp            0x7fffffffe478    0x7fffffffe478
...
rip            0x400080    0x400080 <addition>
eflags         0x202       [ IF ]
...
(gdb) s
8            add eax, edx
(gdb) i r
rax            0x14 20
rbx            0x0  0
rcx            0x14 20
rdx            0x15 21
...
rsp            0x7fffffffe478    0x7fffffffe478
...
rip            0x400082    0x400082 <addition+2>
eflags         0x202       [ IF ]
...
```

```
(gdb) s
addition () at subs.asm:9
9            ret
(gdb) i registers
rax            0x29 41
rbx            0x0  0
rcx            0x14 20
rdx            0x15 21
…
rsp            0x7fffffffe478    0x7fffffffe478
…
rip            0x400084    0x400084 <addition+4>
eflags         0x202       [ IF ]
…
17          mov r8,0x4
(gdb) i r
rax            0x29 41
rbx            0x0  0
rcx            0x14 20
rdx            0x15 21
…
rsp            0x7fffffffe480    0x7fffffffe480
…
rip            0x400094    0x400094 <_start+15>
eflags         0x202       [ IF ]
…
```

Observe the following:

1. The next instruction pointed is `<_start+15>`, which is the first instruction after the subroutine invocation
2. The result is stored in RAX (more properly, in EAX).

Now the remaining part of the program is straightforward, but what happens in the subroutine the second time? We just ran it, and we skip debugging, obtaining:

```
(gdb) i r
rax            0x29 41
rbx            0x0  0
rcx            0x14 20
rdx            0x15 21
rsi            0x0  0
rdi            0x0  0
rbp            0x0  0x0
rsp            0x7fffffffe480    0x7fffffffe480
r8             0x4  4
r9             0x2  2
r10            0x0  0
r11            0x0  0
r12            0x0  0
r13            0x0  0
r14            0x0  0
r15            0x0  0
rip            0x4000a5    0x4000a5 <_start+32>
eflags         0x202       [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

The computation still took place on the first given registers, which does not add flexibility to the subroutine mechanism. We will now see another, more flexible version.