Assembly 101

x86/x64

EPITA · APPING2 S7

Auteur(s)	Hugo Sibony
Date	24/12/2025

Table des matières

1 What is Assembly?

Assembly is the language closest to the processor. Each instruction corresponds directly to a CPU operation.

1.1 Simple Example

In C:

```
int x = 5;
x = x + 3;
```

In assembly, we directly manipulate CPU registers:

```
movl $5, %eax       # Put 5 in register EAX
addl $3, %eax       # Add 3 to EAX → EAX now contains 8
```

1.2 Structure of an Assembly File

```
.section .data          # Data section (global variables)
message: .asciz "Hello" # A string

.section .text          # Code section
.globl my_function      # Make the symbol visible to the linker

my_function:            # Label = start of function
    # your code here
    ret                 # Return to caller
```

⋆ **Important Directives**

Directive	Purpose
`.section .text`	Executable code
`.section .data`	Initialized data (modifiable)
`.section .rodata`	Read-only data (constants, strings)
`.globl name`	Export the symbol (required for C to find your function!)
`.ascii "..."`	String without trailing `\0`
`.asciz "..."`	String with trailing `\0`
`.long value`	32-bit value
`.quad value`	64-bit value

2 Registers: Your Fast Memory

Registers are « variables » directly inside the CPU. They are ultra-fast.

2.1 64-bit Registers (amd64)

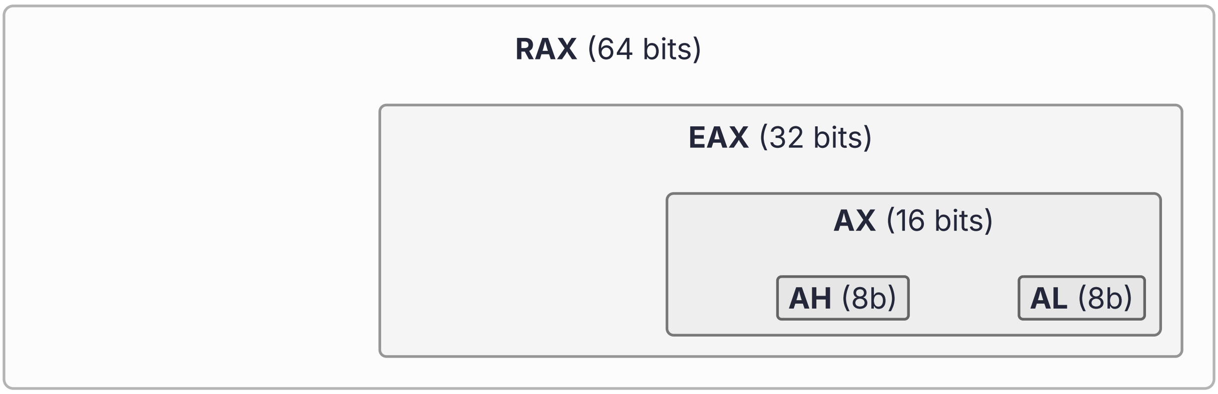
Register	Purpose
`RAX`	Function return value
`RBX`	General purpose (MUST SAVE!)
`RCX`	4th argument (historically: loop counter in i386)
`RDX`	3rd argument / used by divisions
`RSI`	2nd argument
`RDI`	1st argument
`RSP`	Stack pointer (NEVER LOSE IT!)
`RBP`	Base pointer (MUST SAVE!)
`R8`	5th argument
`R9`	6th argument
`R10-R11`	Temporaries
`R12-R15`	General purpose (MUST SAVE!)

2.2 32-bit Registers (i386)

Register	Purpose
`EAX`	Return value
`EBX`	General purpose (MUST SAVE!)
`ECX`	Counter (for loops)
`EDX`	Data / used by divisions
`ESI`	Source (MUST SAVE!)
`EDI`	Destination (MUST SAVE!)
`ESP`	Stack pointer
`EBP`	Base pointer (MUST SAVE!)

2.3 Register Sub-parts

A 64-bit register can be accessed in parts:



! Important

WARNING: Writing to EAX automatically zeroes the upper 32 bits of RAX!

```
movq $0xFFFFFFFFFFFFFFFF, %rax   # RAX = 0xFFFFFFFFFFFFFFFF
movl $1, %eax                     # RAX = 0x0000000000000001 (not 0xFFFFFFFF00000001!)
```

3 The Two Syntaxes: AT&T vs Intel

You need to know how to write in both syntaxes. Here are the differences:

3.1 Comparison Table

Aspect	AT&T	Intel
Order	source, destination	destination, source
Registers	`%rax`	`rax`
Immediates	`$42`	`42`
Memory	`8(%rsp)`	`[rsp + 8]`
Size	Suffix: `movl` , `movq`	Keyword: `DWORD` , `QWORD`

3.2 Side-by-Side Examples

Put a value in a register:

AT&T

```
movl $42, %eax
movq $100, %rax
```

Intel

```
mov eax, 42
mov rax, 100
```

Copy one register to another:

AT&T

```
movq %rdi, %rax
```

Intel

```
mov rax, rdi
```

Read from memory (stack):

AT&T

```
movl 4(%esp), %eax
```

Intel

```
mov eax, [esp + 4]
```

Write to memory:

AT&T

```
movq %rax, (%rdi)
movq %rax, 8(%rdi)
```

Intel

```
mov [rdi], rax
mov [rdi + 8], rax
```

3.3 Size Suffixes (AT&T)

Suffix	Size	Example Register
`b`	1 byte	`AL`, `BL`, `CL`, `R8B`
`w`	2 bytes (word)	`AX`, `BX`, `CX`, `R8W`
`l`	4 bytes (long/dword)	`EAX`, `EBX`, `ECX`, `R8D`
`q`	8 bytes (quad)	`RAX`, `RBX`, `RCX`, `R8`

3.4 Intel Template with GAS

To use Intel syntax with the GNU Assembler (GAS):

```
.intel_syntax noprefix    # Enable Intel, disable % and $ prefixes

.text
.globl my_function
my_function:
    # your code in Intel syntax here
    ret
```

4 Loops and Conditions

4.1 CPU Flags

The CPU has « flags » updated by certain instructions:

- **ZF** (Zero Flag): set to 1 if the result is zero
- **SF** (Sign Flag): set to 1 if the result is negative

Jump instructions (`jz` , `jnz` , etc.) read these flags to decide whether to jump.

4.2 Setting a Register to Zero

```
xorq %rax, %rax         # RAX = RAX XOR RAX = 0
```

⋆ **Astuce**

Why `xor` and not `movq $0, %rax` ?

- **Shorter:** `xorl %eax, %eax` is 2 bytes vs 5 bytes for `movl $0, %eax`
- **Faster:** CPUs recognize this pattern and optimize it (zero-idiom)
- **Bonus:** Writing to EAX automatically zeros the upper 32 bits of RAX!

4.3 Comparing Two Values: `cmp`

```
cmpq %rsi, %rdi         # Computes RDI - RSI, sets flags, discards result
```

After `cmpq %rsi, %rdi` :

- If RDI == RSI → ZF = 1
- If RDI < RSI → SF = 1 (for signed numbers)

4.4 Testing if a Register is Zero: `test`

```
testq %rdi, %rdi        # Computes RDI AND RDI, sets flags, discards result
```

`RDI AND RDI` = RDI. Therefore:

- If RDI == 0 → result = 0 → **ZF = 1**
- If RDI != 0 → result != 0 → **ZF = 0**

This is the standard way to test if a register is zero.

4.5 Conditional Jumps

Instruction	Condition (after `cmp b, a` or `test`)
`je` / `jz`	a == b (or result == 0)
`jne` / `jnz`	a != b (or result != 0)
`jl` / `jg`	a < b / a > b (signed)
`jle` / `jge`	a <= b / a >= b (signed)
`jb` / `ja`	a < b / a > b (unsigned)
`jbe` / `jae`	a <= b / a >= b (unsigned)
`js`	result is negative
`jmp`	always

★ **Astuce**

Signed vs unsigned: `jl` / `jg` for int, `jb` / `ja` for unsigned

4.6 Example: Simple Condition

In C:

```
if (x == 0) return -1;
return x;
```

In assembly:

```
    testq %rdi, %rdi        # RDI AND RDI → ZF=1 if RDI==0
    jnz not_zero            # jump if ZF=0 (so if RDI ≠ 0)
    movq $-1, %rax
    ret
not_zero:
    movq %rdi, %rax
    ret
```

4.7 Example: While Loop

In C:

```c
int sum = 0;
while (n > 0) { sum += n; n--; }
return sum;
```

In assembly:

```
    xorq %rax, %rax         # sum = 0 (xor with itself = 0)
loop:
    testq %rdi, %rdi        # n == 0?
    jz done                 # if ZF=1 (n==0), finish
    addq %rdi, %rax         # sum += n
    decq %rdi               # n--
    jmp loop
done:
    ret
```

4.8 Example: Traversing a String

In C:

```c
int len = 0;
while (*s != '\0') { len++; s++; }
return len;
```

In assembly:

```
    xorq %rax, %rax         # len = 0
loop:
    movzbl (%rdi), %ecx     # load 1 byte at address RDI into ECX
    testb %cl, %cl          # byte == 0?
    jz done
    incq %rax               # len++
    incq %rdi               # s++ (move to next character)
    jmp loop
done:
    ret
```

i Information

`movzbl (%rdi), %ecx` = « Move Zero-extend Byte to Long »

- Reads 1 byte at address RDI
- Zero-extends it to 32 bits
- Puts it in ECX

Intel equivalent: `movzx ecx, BYTE PTR [rdi]`

4.9 Example: For Loop with Array

In C:

```c
int sum = 0;
for (int i = 0; i < len; i++) { sum += arr[i]; }
return sum;
```

In assembly:

```
    xorl %eax, %eax         # sum = 0
    xorl %ecx, %ecx         # i = 0
loop:
    cmpl %esi, %ecx         # compare i with len
    jge done                # if i ≥ len, finish
    movl (%rdi,%rcx,4), %edx  # edx = arr[i]
    addl %edx, %eax         # sum += arr[i]
    incl %ecx               # i++
    jmp loop
done:
    ret
```

⋆ Indexed Addressing

`(%rdi,%rcx,4)` = address `RDI + RCX * 4`

- RDI = base address of array
- RCX = index
- 4 = element size (4 bytes for `int`)

4.10 Tip: lea for Arithmetic

`lea` (Load Effective Address) computes an address without accessing memory. It's a powerful tool for arithmetic!

★ Why use lea instead of add/imul?

- **Doesn't modify flags:** You can keep flags from a previous `cmp` / `test`
- **Multiple operations in one:** `lea (%rdi,%rsi,4), %rax` does `rax = rdi + rsi*4` in one instruction
- **Result in different register:** `lea 1(%rdi), %rax` computes `rdi + 1` into `rax` without modifying `rdi`

AT&T

```
leaq (%rdi,%rsi), %rax     # rax = rdi + rsi
leaq (%rdi,%rdi,4), %rax   # rax = rdi * 5
leaq 1(%rdi), %rax         # rax = rdi + 1
```

Intel

```
lea rax, [rdi + rsi]
lea rax, [rdi + rdi*4]
lea rax, [rdi + 1]
```

★ Multiplication by 10 (useful for atoi)

To compute `result = result * 10 + digit` (where digit is in RDX):

```
leaq (%rax,%rax,4), %rax        # rax = rax + rax*4 = rax * 5
leaq (%rdx,%rax,2), %rax        # rax = rdx + rax*2 = digit + (old_rax*5)*2
                                #                   = digit + old_rax*10
```

Step by step: If RAX=42 and RDX=7:

- After 1st lea: RAX = 42 + 42×4 = 210 (which is 42×5)
- After 2nd lea: RAX = 7 + 210×2 = 427 (which is 7 + 42×10)

5 Division

Division in assembly is a bit special because it uses a pair of registers.

5.1 64-bit Division

```
xorq %rdx, %rdx          # REQUIRED! Set RDX to 0
divq %rbx                # Divide RDX:RAX by RBX
                         # RAX = quotient
                         # RDX = remainder
```

! Important

`xorq %rdx, %rdx` is **mandatory** before `divq`. The instruction divides the 128-bit number formed by RDX:RAX. If RDX contains random values, the result will be wrong.

5.2 Signed Division

For signed numbers, use `idivq` and `cqto` (sign-extend RAX into RDX:RAX):

```
cqto                     # Sign-extend RAX into RDX
idivq %rbx               # Signed division
```

6 The Stack: How It Works

The stack is a memory area that **grows downward** (decreasing addresses).

6.1 Basic Operations

```
# PUSH: Push a value onto the stack
pushq %rax
# Equivalent to:
#   1. RSP = RSP - 8     (stack grows downward)
#   2. Write RAX to address RSP

# POP: Pop a value from the stack
popq %rax
# Equivalent to:
#   1. Read value at address RSP into RAX
#   2. RSP = RSP + 8     (stack shrinks)
```

6.2 Stack Visualization After a CALL

When C calls your function `add(3, 4)` in 32-bit:

```
High addresses
     |
     v
+-----------+
|     4     |  ← ESP + 8  (2nd argument)
+-----------+
|     3     |  ← ESP + 4  (1st argument)
+-----------+
| RetAddr   |  ← ESP      (return address, pushed by CALL)
+-----------+
     ^
     |
Low addresses
```

! Important

This is why the 1st argument is at ESP+4 and not ESP+0! The CALL instruction automatically pushes the return address.

7 Stack Frame and Alignment

7.1 Why 16 Bytes?

The ABI requires: **RSP % 16 == 0 before each `call`**.

7.1.1 What is SSE?

SSE (Streaming SIMD Extensions) = instructions that process **multiple data in parallel**.

The CPU has special 128-bit (16-byte) registers: `XMM0` to `XMM15`.

```
# Classic registers: 1 operation at a time
addq %rax, %rbx         # 1 64-bit addition

# SSE registers: 4 operations in parallel
addps %xmm0, %xmm1      # 4 32-bit additions at once
```

The libc uses SSE to optimize memcpy, strlen, printf, etc. It's faster.

7.1.2 Why Alignment?

SSE instructions that access memory (`movaps` , `movdqa`) require an address divisible by 16. This is a hardware constraint: the memory bus loads 16 aligned bytes more efficiently.

```
movaps (%rsp), %xmm0    # Load 16 bytes from RSP into XMM0
                        # If RSP % 16 ≠ 0 → immediate SIGSEGV
```

If your stack is misaligned when you call printf → printf uses `movaps` → crash.

7.2 The Mechanism

Let's follow RSP step by step with real values:

```
1. Someone calls your function
   RSP = 0x7fff0010 (multiple of 16 ✓)

2. The CALL instruction executes
   - Pushes return address (8 bytes)
   - RSP = 0x7fff0010 - 8 = 0x7fff0008

3. You enter your function
   RSP = 0x7fff0008 → 0x7fff0008 % 16 = 8 × MISALIGNED
```

Problem: At the entry of your function, RSP is ALWAYS misaligned by 8.

7.3 How to Realign

You must subtract 8 bytes (or an odd number of times 8) to get back to a multiple of 16.

7.3.1 Solution 1: Single Push

```
func:
    pushq %rbx          # RSP -= 8
    # RSP was 0x7fff0008, now 0x7fff0000 ✓

    call printf@PLT     # OK, RSP aligned

    popq %rbx
    ret
```

Calculation: `0x7fff0008 - 8 = 0x7fff0000` → `0x7fff0000 % 16 = 0` ✓

7.3.2 Solution 2: Direct Sub

```
func:
    subq $8, %rsp       # Same effect as a push

    call printf@PLT

    addq $8, %rsp
    ret
```

7.3.3 Solution 3: Frame Pointer

```
func:
    pushq %rbp          # RSP -= 8 → aligned
    movq %rsp, %rbp
    subq $32, %rsp      # 32 bytes of local variables (multiple of 16)

    # Variables: -8(%rbp), -16(%rbp), -24(%rbp), -32(%rbp)

    leave               # RSP = RBP, pop RBP
    ret
```

Why `sub $32` and not `sub $24` if I have 3 variables of 8 bytes?

```
After push %rbp:  RSP = 0x7fff0000 (aligned)
After sub $24:    RSP = 0x7fff0000 - 24 = 0x7ffefff8
                  0x7ffefff8 % 16 = 8 ×

After sub $32:    RSP = 0x7fff0000 - 32 = 0x7ffefff0
                  0x7ffefff0 % 16 = 0 ✓
```

Rule: after `push %rbp`, always allocate a multiple of 16.

7.4 Visual Summary

```
                        RSP % 16
                        --------
Before call             0        ✓ aligned
After call (entry)      8        × misaligned
After 1 push            0        ✓ aligned
After 2 push            8        × misaligned
After 3 push            0        ✓ aligned
```

Pattern: odd number of pushes → aligned. Even number → misaligned.

7.5 Complete Stack Frame

```
          +------------------+
+24       |  argument 7      |  (if > 6 args)
          +------------------+
+16       |  argument 8      |
          +------------------+
+8        |  return address  |  ← pushed by CALL
          +------------------+
0         |  old RBP         |  ← RBP points here
          +------------------+
-8        |  variable 1      |
          +------------------+
-16       |  variable 2      |  ← RSP after sub $16
          +------------------+
```

Access: `movq -8(%rbp), %rax` to read variable 1.

7.6 Common Mistake

```
# WRONG - will segfault
func:
    call printf@PLT     # RSP = 8 mod 16 → CRASH
    ret

# CORRECT
func:
    pushq %rbx
    call printf@PLT     # RSP = 0 mod 16 → OK
    popq %rbx
    ret
```

! Important

Debug: In GDB, before a suspicious call: `p $rsp % 16`

If it shows 8 → your stack is misaligned.

8 The ABI: The Contract Between Functions

The ABI (Application Binary Interface) defines the rules for functions to call each other correctly.

8.1 i386 ABI (32-bit)

Category	Value
Arguments	On stack: `[ESP+4]` , `[ESP+8]` , `[ESP+12]` ...
Return	EAX
Caller-saved	EAX, ECX, EDX
Callee-saved	EBX, ESI, EDI, EBP

8.2 amd64 ABI (64-bit)

Category	Value
Arguments	RDI, RSI, RDX, RCX, R8, R9 (in this order)
Return	RAX
Caller-saved	RAX, RCX, RDX, RSI, RDI, R8-R11
Callee-saved	RBX, RBP, R12-R15
Stack	RSP multiple of 16 before `call`

8.3 Saving Registers: When and How?

If your function uses a « callee-saved » register (like RBX), you MUST:

1. Save it at the beginning (with push)
2. Restore it at the end (with pop)

```
my_function:
    pushq %rbx              # Save RBX

    # ... you can use RBX here ...

    popq %rbx               # Restore RBX
    ret
```

If you only use « caller-saved » registers (RAX, RDI, RSI...), no need to save them.

9 Calling C Functions from Assembly

9.1 Recipe

1. Put arguments in RDI, RSI, RDX, RCX, R8, R9
2. Align the stack (RSP must be multiple of 16 before `call`)
3. `call function@PLT` → result in RAX

9.2 Stack Alignment

At entry of your function, RSP = 8 mod 16 (return address was pushed).

Simple solution: `pushq %rbx` before `call` (adds 8 bytes → RSP multiple of 16).

Solution with frame pointer (recommended for complex functions):

AT&T:

```
my_function:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    # ... your code ...
    # Local variables: -8(%rbp), -16(%rbp)

    leave
    ret
```

Intel:

```
my_function:
    push rbp
    mov rbp, rsp
    sub rsp, 16

    ; ... your code ...
    ; Local variables: [rbp-8], [rbp-16]

    leave
    ret
```

⋆ Astuce

`leave` = restore RSP from RBP then pop RBP. Handy for cleaning up the stack frame.

9.3 @PLT and @GOTPCREL

When calling libc functions or accessing external global variables:

```
# Call an external function (printf, puts, getline, etc.)
call printf@PLT           # @PLT = Procedure Linkage Table

# Access an external global variable (stdin, stdout, stderr)
movq stdin@GOTPCREL(%rip), %rdx    # Load ADDRESS of stdin from GOT
movq (%rdx), %rdx                   # Load VALUE of stdin (the FILE*)
```

i Information

Why @PLT? The dynamic linker resolves the function address at runtime.

Why @GOTPCREL? External global variables are in the Global Offset Table.

9.4 lea vs mov

```
leaq msg(%rip), %rdi      # RDI = ADDRESS of msg (to pass a string)
movq var(%rip), %rdi      # RDI = CONTENTS of var (to read a variable)
```

9.5 Example: Calling puts

```
.section .rodata
msg: .asciz "Hello"

.section .text
.globl example
example:
    pushq %rbx                # Align stack
    leaq msg(%rip), %rdi      # arg1 = address of msg
    call puts@PLT
    popq %rbx
    ret
```

9.6 printf with Multiple Arguments

```
    leaq format(%rip), %rdi   # arg1: format string
    movq %rcx, %rsi           # arg2: value to display
    xorl %eax, %eax           # RAX = 0 (no floating-point arguments)
    call printf@PLT
```

! Important

`xorl %eax, %eax` is mandatory before printf/scanf (variadic functions).

9.7 Libc File Functions

Functions for file manipulation:

Function	Prototype	Description
`fopen`	`FILE *fopen(char *path, char *mode)`	Open a file, returns NULL on error
`fclose`	`int fclose(FILE *stream)`	Close a file
`fgetc`	`int fgetc(FILE *stream)`	Read a character, returns −1 (EOF) at end
`fputc`	`int fputc(int c, FILE *stream)`	Write a character
`putchar`	`int putchar(int c)`	Write a character to stdout

⋆ Astuce

fopen modes: `"r"` = read, `"w"` = write (creates/truncates)

10 Syscalls: Talking to the Linux Kernel

10.1 Difference from C Functions

> **! Important**
>
> **WARNING:** For syscalls, the 4th argument is in **R10**, not RCX! (The `syscall` instruction uses RCX internally to save RIP)

	C Functions	Syscalls
4th arg	RCX	**R10**
Call	`call func`	`syscall`
Error	Variable	Negative RAX

10.2 Syscall Structure

```
movq $NUMBER, %rax      # Syscall number
movq arg1, %rdi         # 1st argument
movq arg2, %rsi         # 2nd argument
movq arg3, %rdx         # 3rd argument
movq arg4, %r10         # 4th argument (WARNING: R10, not RCX!)
movq arg5, %r8          # 5th argument
movq arg6, %r9          # 6th argument
syscall                 # Execute syscall
# Result in RAX (negative = error)
```

10.3 Numbers and Arguments

#	Name	Arguments
0	read	fd, buffer, count → bytes read
1	write	fd, buffer, count → bytes written
2	open	path, flags, mode → fd
3	close	fd
9	mmap	addr, len, prot, flags, fd, offset → ptr

10.4 Standard File Descriptors

Name	Value	Description
stdin	0	standard input
stdout	1	standard output
stderr	2	standard error

10.5 Flags for open()

Constant	Value	Description
`O_RDONLY`	`0`	Read only
`O_WRONLY`	`1`	Write only
`O_RDWR`	`2`	Read and write
`O_CREAT`	`0x40`	Create if doesn't exist
`O_TRUNC`	`0x200`	Truncate if exists

⋆ Astuce

To open for writing with creation: `O_WRONLY | O_CREAT | O_TRUNC = 0x241`

10.6 Flags for mmap()

Constant	Value
`PROT_READ`	1
`PROT_WRITE`	2
`MAP_PRIVATE`	`0x02`
`MAP_ANONYMOUS`	`0x20`

⋆ Astuce

To allocate memory:

- `prot = 3` (PROT_READ | PROT_WRITE = 1 | 2)
- `flags = 0x22` (MAP_PRIVATE | MAP_ANONYMOUS = 0x02 | 0x20)
- `fd = -1` (no file, anonymous mapping)
- `offset = 0`

10.7 mmap() to Allocate Memory

```
movq $9, %rax            # syscall mmap
xorq %rdi, %rdi          # addr = NULL
movq $SIZE, %rsi         # len = desired size
movq $3, %rdx            # prot = PROT_READ | PROT_WRITE
movq $0x22, %r10         # flags = MAP_PRIVATE | MAP_ANONYMOUS
movq $-1, %r8            # fd = -1
xorq %r9, %r9            # offset = 0
syscall
# RAX = pointer to allocated memory (or negative if error)
```

10.8 Error Handling

Syscalls return a negative value on error:

```
syscall
testq %rax, %rax     # SF=1 if RAX negative
js error             # Jump if Sign (if negative)
```

11 Debugging with GDB

11.1 Launching GDB

```
gdb ./my_program
```

Command	Description
`break add`	Breakpoint on function add
`run`	Start
`stepi`	1 instruction (steps into calls)
`nexti`	1 instruction (steps over calls)
`continue`	Continue
`info registers`	View registers
`p/x $rax`	Print RAX in hex
`x/s $rdi`	Print string pointed to by RDI
`x/10x $rsp`	10 hex values from RSP

11.2 Example Session

```
$ gdb ./my_program
(gdb) break add              ← breakpoint on your function
(gdb) run                    ← start the program
Breakpoint 1, add ()
(gdb) info registers rdi rsi ← check arguments
rdi            0x3
rsi            0x4
(gdb) stepi                  ← execute 1 instruction
(gdb) info registers rax     ← check result
(gdb) continue               ← continue execution
```

12 Quick Reference

12.1 Calling Conventions

Type	Arguments	Return
C functions (amd64)	RDI, RSI, RDX, RCX, R8, R9	RAX
Syscalls (amd64)	RDI, RSI, RDX, **R10**, R8, R9 (number in RAX)	RAX
i386	[ESP+4], [ESP+8], ...	EAX

12.2 Callee-saved (must save)

Arch	Registers
amd64	RBX, RBP, R12-R15
i386	EBX, ESI, EDI, EBP

12.3 Syscalls

#	Name	Args
0	read	fd, buf, count
1	write	fd, buf, count
2	open	path, flags, mode
3	close	fd
9	mmap	addr, len, prot, flags, fd, off

12.4 Templates

amd64 AT&T

```
.text
.globl func
func:
    ret
```

amd64 Intel

```
.intel_syntax noprefix
.text
.globl func
func:
    ret
```

i386

```
.text
.globl func
func:
    ret
```