AT&T Assembly Language Cheat Sheet

General Syntax

- OP src, dest

X86-64 Registers

Register	Description And Common Usage
RAX	Accumulator: Used in arithmetic operations
RBX	Base: Used as a pointer to data
RCX	Counter: Used in shift/rotate instructions
RDX	Data: Used in arithmetic operations
RDI	Destination Index: Used as a pointer to a destination
RSP	Stack Pointer: Pointer to the top of the stack
RBP	Base Pointer: Pointer to the base of the stack frame
RSI	Source Index: Used as a pointer to a source
R8-R15	General Purpose Registers

Referencing Lower Bits

- d suffix: Lower 32 bits of the register
- w suffix: Lower 16 bits of the register
- b suffix: Lower 8 bits of the register
- no suffix: Full register

Common Assembly Instructions Cheat Sheet (AT&T Syntax)

Instruction	Description	Syntax	Example
`mov`	Move data	`mov src, dest`	`mov %eax, %ebx`
`push`	Push to stack	`push src`	`push %eax`
`pop`	Pop from stack	`pop dest`	`pop %eax`
`add`	Add values	`add src, dest`	`add $5, %eax`
`sub`	Subtract values	`sub src, dest`	`sub $5, %eax`
`inc`	Increment by 1	`inc dest`	`inc %eax`
`dec`	Decrement by 1	`dec dest`	`dec %eax`
`cmp`	Compare values	`cmp src, dest`	`cmp $5, %eax`
`jmp`	Jump to label	`jmp label`	`jmp label`
`call`	Call subroutine	`call label`	`call function`
`ret`	Return from subroutine	`ret`	`ret`
`lea`	Load address	`lea src, dest`	`lea (%eax, %ebx, 4), %ecx`
`neg`	Negate value	`neg dest`	`neg %eax`
`imul`	Signed multiply	`imul src, dest`	`imul $5, %eax`
`idiv`	Signed divide	`idiv src`	`idiv %ebx`

Instruction Suffixes in AT&T Syntax

Suffix	Operand Size	Example
b	Byte (8 bits)	`movb $1, %al`
w	Word (16 bits)	`movw $1, %ax`
l	Long (32 bits)	`movl $1, %eax`
q	Quad (64 bits)	`movq $1, %rax`

Types of Operands

Type	Description	Syntax	Example
Immediate	Constant values directly within the instruction itself, used immediately by the CPU.	`$value`	`movl $10, %eax` # Moves 10 into %eax
Label Address	Represents the address of a label in the code	`$label`	`mov $message, %rdi` # Moves the address of the label `message` into %rdi
Register	Values stored in the CPU's registers, which are small, fast storage locations within the CPU.	`%register`	`movl %ebx, %eax` # Moves value in %ebx to %eax
Memory	Data stored in memory locations, specified by addresses in the operand.	`(address)` or `(base, index, scale, displacement)`	`movl 4(%ebp), %eax` # Moves value at (4 + %ebp) to %eax`movl (%eax,%ebx,4), %ecx` # Moves value at (%eax + 4 * %ebx) to %ecx
Effective Address	Addresses dynamically calculated using base, index, scale, and displacement.	`(base, index, scale, displacement)`	`leal (%eax,%ebx,4), %ecx` # Loads effective address into %ecx

Memory Addressing Modes

Most General Form:

`D(B, I, S)`

Where: - **D**: Constant displacement value (immediate value) or offset - **B**: Base register - **I**: Index register (except `%rsp`) - **S**: Scale factor (1, 2, 4, or 8)

This form translates to the effective address:

`Mem[Reg[B] + S * Reg[I] + D]`

Example:

`movq 8(%rbp, %rax, 4), %rdx`

Breaking this down: - **D (8)**: Displacement value of `8` - **B (%rbp)**: Base register `%rbp` - **I (%rax)**: Index register `%rax` - **S (4)**: Scale factor `4`

Now, using the most general form, we get:

`Effective Address = Reg[%rbp] + 4 * Reg[%rax] + 8`

Let's illustrate it step by step: 1. **Base Register (%rbp)**: - Assume `%rbp` contains the value `1000`. 2. **Index Register (%rax)**: - Assume `%rax` contains the value `2`. 3. **Scale Factor (4)**: - Multiply the value in `%rax` by the scale factor `4`: `4 * 2 = 8` 4. **Displacement (8)**: - Add the constant displacement `8`.

Combining all these:

`Effective Address = 1000 + 8 + 8 = 1016`

So, the instruction `movq 8(%rbp, %rax, 4), %rdx` translates to: - Move the value at memory address `1016` into the register `%rdx`.

Endianness

- Memory is represented as a sequence of byte integers
- Example: 0x0A0B0C0Dh
  - Big Endian: High-order byte stored at the lowest memory address
    * 0A 0B 0C 0D
  - Little Endian: Low-order byte stored at the lowest memory address
    * 0D 0C 0B 0A

Load Effective Address (LEA)

`lea offset(base, index, scale), dest` where: - **offset**: Constant displacement value - **base**: Base address - **index**: Index register (optional) - **scale**: Scale factor (optional)

By effective address, we mean the address of the source operand, not the value stored at that address.

- Computing addresses without a memory dereference

Example

`leaq 8(%rbp, %rax, 4), %rdx`

Assuming in rbp we have 1000 and in rax we have 2, the effective address will be:

`1000 + 4 * 2 + 8 = 1016 which is stored in rdx`

Control Flow Instructions

Sure, let's focus on the `jmp` instruction and list all possible ways to use it, including relative and absolute jumps.

Jump

The `jmp` instruction sets the program counter (PC) and redirects the flow of control to another part of the program. Here are the different ways to use the `jmp` instruction:

1. Relative Jump by Label A relative jump changes the program counter to a new address that is a certain offset from the current instruction's address. This is also known as a direct jump.

- **Syntax**: `jmp label`
- **Description**: The `label` is a symbolic name for a memory location.
- **Example**:

```
.section .text
.global _start

_start:
    jmp loop_start       # Jump to the label 'loop_start'

loop_start:
    # Code for the loop or target of the jump
```

2. Relative Jump with Immediate Offset A relative jump can also be specified with an immediate offset, which is added to the address of the next instruction to compute the target address.

- **Syntax**: `jmp offset`
- **Description**: The `offset` is an 8-bit, 16-bit, or 32-bit immediate value.
- **Example**:

```
.section .text
.global _start

_start:
    jmp 0x6               # Jump 6 bytes forward from the next instruction

    # This code is skipped
    mov $0, %eax          # 2 bytes
    mov $0, %ebx          # 2 bytes
    mov $0, %ecx          # 2 bytes

    # Code after the jump
    mov $1, %edx
```

3. Absolute Jump by Address An absolute jump sets the program counter to a specific address, which can be stored in a register or a memory location. This is also known as an indirect jump and allows for dynamic jump targets.

- **Syntax**: `jmp *address`
- **Description**: The `address` is an absolute address in a register or memory location, indicated by a `*`.
- **Example**:

```
.section .text
.global _start

_start:
    mov $0x0A0B0C0D, %rax  # Load the absolute address 0x0A0B0C0D into RAX
    jmp *%rax              # Jump to the address stored in RAX
```

Summary The `jmp` instruction can be used in several ways to alter the flow of control in a program: 1. **Relative Jump by Label**: `jmp label` - Jumps to a symbolic label. 2. **Absolute Jump by Address**: `jmp *address` - Jumps to an address stored in a register or memory location. 3. **Relative Jump with Immediate Offset**: `jmp offset` - Jumps to an address computed by adding an immediate offset to the current instruction's address.

Conditional Jump

Conditional jumps in assembly language are used to alter the flow of control based on the result of a comparison or a test. These instructions typically follow a `cmp` (compare) or `test` instruction and jump to a specified label if a certain condition is met.

How `cmp` Works The `cmp` (compare) instruction performs a subtraction between two operands but does not store the result. Instead, it sets the appropriate flags in the EFLAGS register based on the result of the subtraction.

- **Syntax**: `cmp src, dest`
    - `src`: The source operand (immediate value, register, or memory).
    - `dest`: The destination operand (register or memory).

Flags Affected by `cmp`

- **ZF (Zero Flag)**: Set if the result of the subtraction is zero.
- **SF (Sign Flag)**: Set if the result of the subtraction is negative.
- **CF (Carry Flag)**: Set if there is a borrow from the subtraction (i.e., if `src` is greater than `dest`), if carrying out of the most significant bit. (unsigned overflow)
- **OF (Overflow Flag)**: Set if there is a signed overflow. if the sign of the result is different from the sign of the operands.

Common Conditional Jump Instructions (jcc) A conditional jump instruction is executed based on the state of the flags set by the `cmp` instruction. Here are some common conditional jump instructions and their descriptions:

Instruction	Description	Condition	Example Usage
`jmp`	Unconditional jump	Always	`jmp target_label`
`je` / `jz`	Jump if equal / zero	ZF = 1	`je equal_label`
`jne` / `jnz`	Jump if not equal / not zero	ZF = 0	`jne not_equal_label`
`jg` / `jnle`	Jump if greater / not less or equal	ZF = 0 and SF = OF	`jg greater_label`
`jge` / `jnl`	Jump if greater or equal / not less	SF = OF	`jge greater_equal_label`
`jl` / `jnge`	Jump if less / not greater or equal	SF OF	`jl less_label`
`jle` / `jng`	Jump if less or equal / not greater	ZF = 1 or SF OF	`jle less_equal_label`
`ja` / `jnbe`	Jump if above / not below or equal (unsigned)	CF = 0 and ZF = 0	`ja above_label`
`jae` / `jnb`	Jump if above or equal / not below (unsigned)	CF = 0	`jae above_equal_label`

Instruction	Description	Condition	Example Usage
`jb` / `jnae`	Jump if below / not above or equal (unsigned)	CF = 1	`jb below_label`
`jbe` / `jna`	Jump if below or equal / not above (unsigned)	CF = 1 or ZF = 1	`jbe below_equal_label`

Example 1: Jump if Equal

```
.section .text
.global _start

_start:
    mov $5, %eax          # Load 5 into EAX
    cmp $5, %eax          # Compare EAX with 5
    je equal_label        # Jump to 'equal_label' if EAX == 5

    # Code if not equal
    mov $0, %ebx          # Set EBX to 0

equal_label:
    # Code if equal
    mov $1, %ebx          # Set EBX to 1
```

Example 2: Jump if Greater

```
.section .text
.global _start

_start:
    mov $10, %eax         # Load 10 into EAX
    cmp $5, %eax          # Compare EAX with 5
    jg greater_label      # Jump to 'greater_label' if EAX > 5

    # Code if not greater
    mov $0, %ebx          # Set EBX to 0

greater_label:
    # Code if greater
    mov $1, %ebx          # Set EBX to 1
```

Summary Conditional jumps are essential for implementing control flow in assembly language. They allow the program to make decisions based on comparisons and tests, enabling the creation of loops, conditional branches, and other control structures. The `cmp` instruction is used to set the appropriate flags in the EFLAGS register, which are then checked by the conditional jump instructions to determine whether to jump to a specified label.

Yes, in assembly language, you typically need to use the `cmp` instruction before each conditional jump to set the appropriate flags in the EFLAGS register. Each `cmp` instruction performs a comparison and sets the flags based on the result, which the subsequent conditional jump instruction uses to determine whether to jump.

Example: If-Else Ladder with Multiple `cmp` Instructions

```
.section .text
.global _start

_start:
    mov $2, %eax          # Load a value into EAX for testing

    cmp $1, %eax          # Compare EAX with 1
    je case_1             # Jump to case_1 if EAX == 1
    cmp $2, %eax           # Compare EAX with 2
    je case_2              # Jump to case_2 if EAX == 2

    cmp $3, %eax           # Compare EAX with 3
    je case_3              # Jump to case_3 if EAX == 3

    jmp default_case       # Jump to default_case if none of the above conditions are met

case_1:
    # Code for case 1
    mov $1, %ebx           # Set EBX to 1
    jmp end                # Jump to end

case_2:
    # Code for case 2
    mov $2, %ebx           # Set EBX to 2
    jmp end                # Jump to end

case_3:
    # Code for case 3
    mov $3, %ebx           # Set EBX to 3
    jmp end                # Jump to end

default_case:
    # Default case
    mov $0, %ebx           # Set EBX to 0

end:
    # End of the program
```

1. **Comparison**: Each `cmp` instruction compares the value in `eax` with a constant.
2. **Conditional Jump**: Based on the result of the comparison, a conditional jump instruction (e.g., `je`) is used to jump to the corresponding code block.
3. **Default Case**: If none of the conditions are met, the program jumps to the default case.
4. **End**: Each code block ends with an unconditional jump to the end of the program to avoid falling through to the next case.

`test` Instruction

The `test` instruction performs a bitwise AND operation between two operands and sets the appropriate flags in the EFLAGS register based on the result. It does not store the result of the AND operation; it only affects the flags.

Syntax

```
test src, dest
```

- `src`: The source operand (immediate value, register, or memory).
- `dest`: The destination operand (register or memory).

Flags Affected by `test`

- **ZF (Zero Flag)**: Set if the result of the AND operation is zero.
- **SF (Sign Flag)**: Set if the result of the AND operation is negative.
- **CF (Carry Flag)**: Always cleared to 0.
- **OF (Overflow Flag)**: Always cleared to 0.

Example 1: Testing a Register

```
.section .text
.global _start

_start:
    mov $5, %eax          # Load 5 into EAX
    test %eax, %eax       # Perform bitwise AND between EAX and EAX
    je zero_label         # Jump to 'zero_label' if the result is zero (ZF is set)

    # Code if not zero
    mov $1, %ebx          # Set EBX to 1
    jmp end               # Jump to end

zero_label:
    # Code if zero
    mov $0, %ebx          # Set EBX to 0

end:
    # End of the program
```

Example 2: Testing Specific Bits

```
.section .text
.global _start

_start:
    mov $0b1010, %eax     # Load binary 1010 into EAX
    test $0b1000, %eax    # Test if the third bit is set
    je bit_not_set        # Jump to 'bit_not_set' if the third bit is not set

    # Code if the third bit is set
    mov $1, %ebx          # Set EBX to 1
    jmp end               # Jump to end

bit_not_set:
    # Code if the third bit is not set
    mov $0, %ebx          # Set EBX to 0

end:
    # End of the program
```

Summary

- **Operation**: The `test` instruction performs a bitwise AND operation between two operands.
- **Flags**: It sets the ZF and SF flags based on the result of the AND operation, and always clears the CF and OF flags.
- **Usage**: Commonly used to check if specific bits are set or to test if a value is zero.

The `test` instruction is useful for bitwise operations and checking conditions without modifying the operands.

Call

Calls a subroutine or function. - Relative (fixed offset): `call label` (direct call) - Absolute (register or memory location): `call *address` (indirect call) - Also pushes the return address onto the stack (address of the next instruction after the call)

Calling Conventions Defines how to pass arguments, return values, and manage the stack. This is important when using 3rd party libraries or calling functions written in other languages. Calling conventions are enforced by the compiler and runtime environment.

Conventions include additional information such as handling of floating point regs

- **Caller-Saved Registers**: Registers that the caller must save before calling a function if it wants to preserve their values.

- This is known as register spill, which involves saving the register values to stack memory.

- **Callee-Saved Registers**: Registers that the callee must save and restore if it uses them.

- **Return Address**: The address to return to after a function call.

- **Stack Frame**: The region of the stack that stores local variables and function call information.

cdecl (C Declaration) Common calling convention for C functions on x86 (32-bit).

- **arguments**: Pushed onto the stack in reverse order (right to left).
- **return value**: Stored in `%eax`.
- **caller-saved registers**: `%eax`, `%ecx`, and `%edx`.
- Callee functions can modify these registers without saving them.
- **callee-saved registers**: `%ebx`, `%esi`, `%edi`, `%ebp`.
- Callee functions must save and restore these registers if they use them.

System V AMD64 ABI (Application Binary Interface) Calling convention for x86-64 systems.

- **arguments**: Passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`.
- Additional arguments are passed on the stack.
- **return value**: Stored in `%rax`.
- **caller-saved registers**: `%rax`, `%rcx`, `%rdx`, `%r8`, `%r9`, `%r10`, `%r11`.
- **callee-saved registers**: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`.
- Callee functions must save and restore these registers if they use them.

`call` **Instruction** Pushes next_ins_addr on stack and transfers control to address described by operand

is equivalent to:

```
push next_ins_addr
jmp function
```

32-bit x86 (cdecl)

```
# Arguments: int a, int b, int c
pushl %ecx # Push third argument
pushl %ebx # Push second argument
pushl %eax # Push first argument
call function
```

64-bit x86-64 (System V AMD64 ABI)

```
# Arguments: int a, int b, int c
movl %edi, %edi # First argument in %edi
movl %esi, %esi # Second argument in %esi
movl %edx, %edx # Third argument in %edx
call function
```

`call` **Instruction in 64-bit** In x86-64 bit AT&T assembly, the `call` instruction performs the following steps: 1. **Push the Return Address**: The address of the next instruction (after the call) is pushed onto the stack. 2. **Jump to the Function**: Control is transferred to the target function.

3. You still need to manage the stack frame within the function (prologue and epilogue).

Here is the complete assembly equivalent to a `call` instruction, including all the necessary steps:

```
# Arguments: int a, int b, int c, int d, int e, int f, int g, int h
movq $1, %rdi # First argument in %rdi
movq $2, %rsi # Second argument in %rsi
movq $3, %rdx # Third argument in %rdx
movq $4, %rcx # Fourth argument in %rcx
movq $5, %r8 # Fifth argument in %r8
movq $6, %r9 # Sixth argument in %r9
```

```
# Push the additional arguments onto the stack (in reverse order)
movq $8, %rax # Eighth argument
pushq %rax
movq $7, %rax # Seventh argument
pushq %rax

# Push the return address (next instruction address)
leaq next_ins_addr(%rip), %rax # Load the address of the next instruction into %rax
pushq %rax # Push the return address onto the stack

# Jump to the function
jmp function

# next_ins_addr:
# (The next instruction after the call)
nop # Placeholder for the next instruction

# Function prologue (inside the callee function)
function:
# This is equivalent to the enter instruction
pushq %rbp # Save the caller's frame pointer
movq %rsp, %rbp # Set up the callee's frame pointer

# Function body
# ...

# Accessing the seventh and eighth arguments (on the stack)
movq 16(%rbp), %rax # Move the seventh argument from the stack to %rax
movq %rax, -0x48(%rbp) # Move the seventh argument to a local variable
movq 24(%rbp), %rax # Move the eighth argument from the stack to %rax
movq %rax, -0x50(%rbp) # Move the eighth argument to a local variable

# Function epilogue (inside the callee function)
# This is equivalent to the leave instruction
movq %rbp, %rsp # Restore the stack pointer
popq %rbp # Restore the caller's frame pointer
ret # Return to the caller (pops the return address and jumps to it)
```

Return

Returns from a subroutine. - Pops the return address from the stack - Sets PC to absolute address popped from the stack

`ret` Instruction Pops return address from stack and transfers control to that address

The `ret` instruction performs the following steps: 1. Pops the return address from the stack. 2. Jumps to the return address.

Here is the complete assembly equivalent to a `ret` instruction:

```
# Function epilogue (inside the callee function)
movq %rbp, %rsp # Restore the stack pointer
popq %rbp # Restore the caller's frame pointer

# Equivalent to `ret`
popq %rax # Pop the return address into %rax
jmp *%rax # Jump to the return address
```

In this example: - The `movq %rbp, %rsp` instruction restores the stack pointer to the value it had before the function was called. - The `popq %rbp` instruction restores the caller's frame pointer. - The `popq %rax` instruction pops the return address from the stack into the `%rax` register. - The `jmp *%rax` instruction jumps to the return address stored in `%rax`.

This sequence of instructions is equivalent to the `ret` instruction, which combines these steps into a single instruction.

In x86 assembly, there are several ways to load an address into a register. The method you choose depends on whether you are working in 32-bit or 64-bit mode. Here are some common techniques for both 32-bit and 64-bit modes:

Loading an Address in x86 32-bit

Using a Constant

You can directly load an address into a register using the `mov` instruction with an immediate value.

```
.section .data
my_data:
    .long 42              # Define a 32-bit integer

.section .text
.global _start

_start:
    mov $my_data, %eax    # Load the address of 'my_data' into EAX
```

Using `call` and `pop` (GETPC)

This technique is useful for position-independent code (PIC). It involves using a `call` instruction to push the return address onto the stack and then popping it into a register.

```
.section .text
.global _start

_start:
    call get_pc           # Call the 'get_pc' label
get_pc:
    pop %eax              # Pop the return address into EAX
    add $offset, %eax     # Adjust the address if necessary

    # Now EAX contains the address of the 'get_pc' label
    # You can use this address to access data relative to it

    # Example usage
    mov $my_data - get_pc + offset, %ebx  # Load the address of 'my_data' into EBX

    # End of the program
    mov $1, %eax          # Exit system call number
    xor %ebx, %ebx        # Exit status 0
    int $0x80             # Invoke system call

.section .data
my_data:
    .long 42              # Define a 32-bit integer
```

Loading an Address in x86-64 (64-bit)

Using RIP-Relative Addressing In 64-bit mode, you can use RIP-relative addressing to load an address into a register. This is useful for position-independent code.

```
.section .data
my_data:
    .quad 42              # Define a 64-bit integer

.section .text
.global _start

_start:
    lea my_data(%rip), %rax  # Load the address of 'my_data' into RAX
```

Using a Constant You can also directly load an address into a register using the `mov` instruction with an immediate value.

```
.section .data
my_data:
    .quad 42              # Define a 64-bit integer

.section .text
.global _start

_start:
    mov $my_data, %rax    # Load the address of 'my_data' into RAX
```

Summary

- **x86 32-bit**:
  - **Using a Constant**: Directly load the address using `mov`.
  - **Using `call` and `pop` (GETPC)**: Use `call` to push the return address onto the stack and `pop` to load it into a register.
- **x86-64 (64-bit)**:
  - **Using RIP-Relative Addressing**: Use `lea` with RIP-relative addressing to load the address.
  - **Using a Constant**: Directly load the address using `mov`.

Disassembly

The process of recovering the assembly code of a binary executable from its machine code, this can be hard, especially for CISC variable length instruction sets like x86.

Some tools that can help with disassembly: - **objdump**: A command-line tool that displays information about object files. - Example: `objdump -d executable` - gdb: A debugger that can disassemble code. - Example: `gdb -batch -ex "file executable" -ex "disassemble /m main"`

When using `gdb` (GNU Debugger) to debug a program, you can disassemble code to view the assembly instructions. This can be particularly useful for low-level debugging and understanding how your code is being executed at the machine level.

Using the `disas` Command in `gdb`

The `disas` (disassemble) command in `gdb` is used to display the assembly instructions for a specified function or memory range.

Syntax

```
disas [start[, end]]
```

- `start`: The starting address or function name to disassemble.
- `end`: The ending address (optional).

Example Usage

1. **Disassemble a Function**:

   ```
   (gdb) disas main
   ```

   This command disassembles the `main` function, showing the assembly instructions for that function.

2. **Disassemble a Memory Range**:

   ```
   (gdb) disas 0x400080, 0x4000A0
   ```

   This command disassembles the instructions in the memory range from `0x400080` to `0x4000A0`.

Using the Assembly Layout in `gdb`

`gdb` also provides a layout mode that allows you to view the source code and assembly instructions side by side. This can be very helpful for understanding how high-level code maps to assembly instructions.

Enabling the Assembly Layout

1. **Start `gdb`**:

   ```
   gdb ./executable
   ```

2. **Run the Program**:

   ```
   (gdb) start
   ```

3. **Enable the Assembly Layout**:

   ```
   (gdb) layout asm
   ```

   This command switches to the assembly layout, displaying the assembly instructions as the program executes.

4. **Switch to Source and Assembly Layout**:

   ```
   (gdb) layout split
   ```

   This command displays both the source code and the corresponding assembly instructions.

Summary

- **`disas` Command**: Use the `disas` command to disassemble functions or memory ranges in `gdb`.
  - Example: `(gdb) disas main`
- **Assembly Layout**: Use the `layout asm` command to view assembly instructions in a layout mode.
  - Example: `(gdb) layout asm`
- **Source and Assembly Layout**: Use the `layout split` command to view both source code and assembly instructions side by side.
  - Example: `(gdb) layout split`

Shellcode

Shellcode is a small piece of code used as the payload in a software exploit. It is typically written in assembly language and injected into a vulnerable program to gain control over its execution.

System Calls in 32-bit Linux

- In 32-bit Linux systems, system calls are invoked using the `int 0x80` instruction which generates a software interrupt.
- The system call number is passed in the `eax` register.
- Arguments are passed in `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp` registers.
  - All registers are preserved across the system call except `eax`.
- The return value is stored in `eax`.

System Calls in 64-bit Linux

- Use the `syscall` instruction.
- System call number is passed in `rax`.
- Arguments are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` registers.
  - The kernel destroys `rcx` and `r11`.
  - System calls are limited to 6 arguments, you can't passed more on the stack.
- The return value is stored in `rax`, which is a value between -4095 and -1 (-errno) on error, and a positive number or zero on success.
- Note that 64-bit can also execute 32-bit code, so it can use the `int 0x80` method as well.