**Was machen ADD, SUB, MUL, DIV, IDIV, IMUL und NEG in Nasm und warum gibt es je 2 Befehle für Multiplikation und Division?**

* ADD Addition von zwei ganzen Zahlen. Das Ergebnis wird in das erste angegebene Register geschrieben. Z.B. “add rdx, rcx” addiert die Werte in rdx und rcx und schreibt das Ergebnis in rdx.
* SUB Subtraktion von zwei ganzen Zahlen. Das Ergebnis wird in das erste angegebene Register geschrieben. Z.B. “sub rdx, rcx” subtrahiert den Wert in rcx vom Wert in rdx und schreibt das Ergebnis in rdx.
* IMUL Multiplikation von ganzen Zahlen mit Vorzeichen. Bei der Multiplikation von 32-Bit Integern ist das Ergebnis eine 64-Bit Zahl. Bei der Multiplikation von 64-Bit Integern ist das Ergebnis eine 128-Bit. Da es viele Verschiedene Formen für IMUL gibt (Anzahl und Größe der Argumente), zählen wir hier nicht alle auf.
* MUL Multiplikation von ganzen Zahlen ohne Vorzeichen. Bei der Multiplikation von 32-Bit Zahlen ist das Ergebnis eine 64-Bit Zahl. Bei der Multiplikation von 64-Bit Integern wird RDX:RAX beschrieben (128-Bit). Der Befehl nimmt ein Argument entgegen, das ein Register sein muss. Der Wert wird mit rax multipliziert und das Ergebnis in RAX bzw RDX:RAX geschrieben.
* IDIV Division von ganzen Zahlen mit Vorzeichen.
* DIV Division von ganzen Zahlen ohne Vorzeichen.

Das Ergebnis der Division wird in RAX geschrieben und der Rest in RDX.

Alle Antworten von Wikipedia und alten Kenntnissen aus Talks und Stackoverflow.

**Wo stehen die übergebenen Parameter in einer Funktion?**

Bei der von uns genutzten Architektur (x86-64) werden die Parameter der Reihe nach (von links nach rechts) in die Register rdi, rsi, rdx, rcx, r8 und r9. Weitere Werte werden auf den Stack "gelegt". Das kann mit folgenden Befehlen erreicht werden:

mov DWORD PTR [esp], <value>

add esp, <byte-zahl>

Das "DWORD PTR [esp]" ist notwendig, damit die Position im Speicher beschrieben wird, auf die esp zeigt und nicht das Register esp selbst. Danach muss mit add das Register esp erneuert werden, um auf das neue Offset zu zeigen.

Alternativ kann auch folgender Befehl verwendet werden:

push <value>

Andere Calling Conventions verlangen, dass alle Werte auf dem Stack abgelegt werden und nicht in Registern.

C Calling Conventionen verlangen, dass die Werte von links nach rechts auf dem Stack abgelegt werden und von rechts nach Links gelesen werden (Das nennt man auch "RTL" oder "RTL(C)").

Viele Zwischensprachen unterstützen 2 Calling Conventions: C und Pascal. Pascal legt die Argumente in umgekehrter Reihenfolge auf dem Stack ab.

**Wo muss der Rückgabewert hingeschrieben werden?**

In eax wenn eine 32-bit Zahl zurückgegeben werden soll, rax wenn eine 64-bit Zahl zurückgegeben werden soll oder rdx:rax wenn eine 128-bit Zahl returned werden soll.

Wenn der Rückgabewert größer ist, kann der Wert auf dem Stack abgelegt werden (aus Konvention wird dann ein Pointer zu dieser Position in eax oder rax geschrieben.

Ein geeigneter C-Wrapper zum Ausführen der Funktion wird Ihnen im KVV gestellt.Warum ist ein solcher Wrapper nötig?

Durch den C-Wrapper ist es für uns einfacher, den NASM-Code ausführbar zu machen. Der Compiler generiert den ganzen "Boilerplate"-Code für main, den wir sonst selbst schreiben müssten.