# Übung 01

## Aufgabe 1

**Was heißt "general-purpose Computer"?**

Ein “general-purpose Computer” ist nicht für die Lösung eines spezifischen Problems gedacht, sondern allgemein für die Lösung von Problemen. Welches Problem der Computer letztendlich lösen kann ist von den Programmen auf dem Rechner bzw. von der Programmierung abhängig.

**Was heißt "Programme sind auch nur Daten"?**

Das von-Neumann-Rechnermodell kennt nur einen Speicher. Das Programm, das aus den Instruktionen/Befehlen besteht, wird im selben Speicher abgelegt wie die Daten, mit denen gerechnet wird. Somit können Code und Daten nicht unterschieden werden. Was am Ende ein Befehl ist und was Daten, mit denen man rechnet, hängt von der Interpretation ab. Dies kann allerdings zu Problemen führen, da die Daten als Befehle interpretiert werden können und anders herum.

Probleme damit:

- Daten können als Befehle interpretiert werden.

- Programme können sich selbst verändern (nützlich für Viren)

**Was sind die 4 Hauptbestandteile des von-Neumann-Rechnermodells?**

CPU (Enthält Control Unit, Arithmetic Logical Unit bzw. Operating Unit), Memory, Peripherie (Input/Output Unit), Verbindung (Bus?)

- CPU führt Befehle aus.

- Memory speichert die Daten und Programm-Befehle als eine Sequenz von bits.

- I/O gibt ein Interface für Input (via. Maus, Tastatur, Mikrofon) und Output (Monitor, Lautsprecher)

- Bus verbindet alle Komponenten (So kann die CPU etwa auf den Speicher und die I/O zugreifen)

**Was machen der Datenprozessor und Befehlsprozessor in der CPU?**

// TODO

**Warum sind Daten- und Adressbus getrennt?**

Damit man zur selben Zeit an unterschiedlichen Positionen auf sie zugreifen kann. Wären die beiden Busse kombiniert, dann wären die Zugriffe sehr viel langsamer. Siehe auch "von-Neumann Flaschenhals".

**Was bedeutet SISD? Was für andere Kategorien gibt es nach Flynn?**

- SISD steht für "Single Instruction Single Data". Die CPU kann zur selben Zeit nur einen Befehl ausführen und jeder Befehl kann nur einen Operanden bearbeiten.

- SIMD ("Single Instruction Multiple Data"): Mehrere CPUs können Instruktionen sequenziell oder parallel ausführen und mehrere Datenströme bearbeitet.

- MISD ("Multiple Instruction streams, single data stream"): Mehrere PUs führen gleichzeitig mehrere Instruktionen auf dem selben Datenstrom aus.

- MIMD ("Multiple instruction streams, multiple data streams"): Mehrere PUs führen asynchron Befehle auf mehreren Datenströmen aus. Die Daten können von mehreren Prozessen gleichzeitig bearbeitet werden oder auch nicht. Daten können auch auf unterschiedlichen Computern liegen, die bspw. über einen zentralen Server gesteuert werden. So können sehr viele komplexe Berechnungen unabhängig durchgeführt und die Ergebnisse dann verbunden werden.

**Was unterscheidet Maschinencode, Assemblersprachen sowie Ein- und Mehr-Adress-Befehle?**

// TODO

**Was ist das Zwei-Phasen-Konzept der Befehlsverarbeitung und welches Problem wird damit auf welche Weise gelöst?**

- Es gibt 2 Phasen, die pro Befehl während der Ausführung eines Programms durchlaufen werden: Interpretationsphase und Exekutions-Phase.

- Interpretation: Der sogenannte Program-Counter (Auch PC oder Instruction Pointer genannt) zeigt, welcher Befehl gerade ausgeführt werden soll und wie groß die Operanden sind. Je nachdem wird der Program-Counter nach der Exekution soweit inkrementiert, dass er auf den nächsten Befehl und nicht auf irgendwelche Daten zeigt.

- Exekution: Die Daten, die für die Ausführung des Befehls benötigt werden, werden ausgelesen und bearbeitet. Danach werden sie an die Position geschrieben, die der Befehl angibt.

- Der Program-Counter SOLLTE immer auf einen Befehl zeigen. Damit SOLLTE auch immer ein Befehl ausgeführt werden anstatt dass arbiträre Daten als Befehl interpretiert werden. "SOLLTE" weil theoretisch auch der PC verschoben und so das Programm durcheinander bringen könnte. ("jmp 0xWHATEVER" oder "jmp eax")

**Was ist der "Flaschenhals" des von-Neumann-Rechnermodells? Wie versuchte man es später zu lösen?**

- Wenn der Zugriff auf den Speicher und/oder die Busse langsamer ist als die CPU, dann kann die CPU nicht ihre volle Geschwindigkeit nutzen, da Befehle nur sequenziell abgearbeitet werden können.

## Aufgabe 2:

**Schreiben Sie eine NASM-Funktion, welche …**

// Siehe gauss.asm

**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.

**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.

// TODO: Double-check answer...