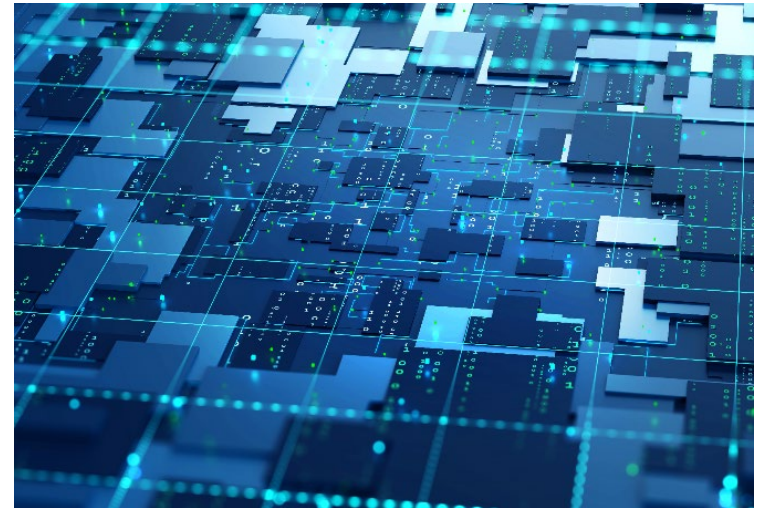


Compilerbau

Compilerbau Grundlagen



Letzte Aktualisierung: 16. September 2024

FH Zentralschweiz

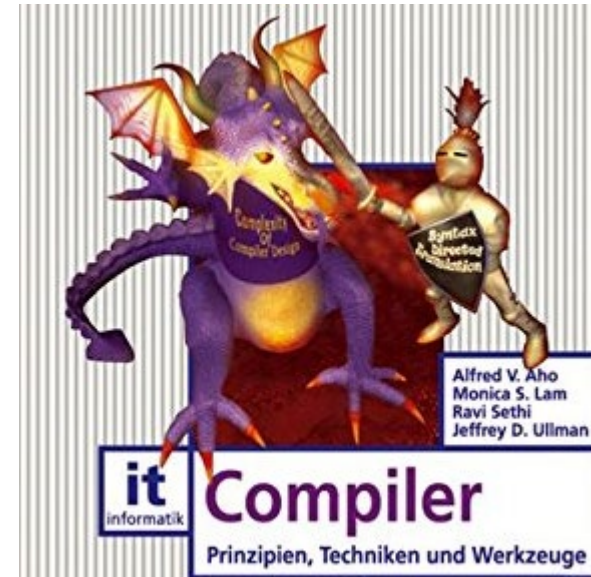
Autoren:

- Andreas Kurmann
- Martin Bättig

Inhalt

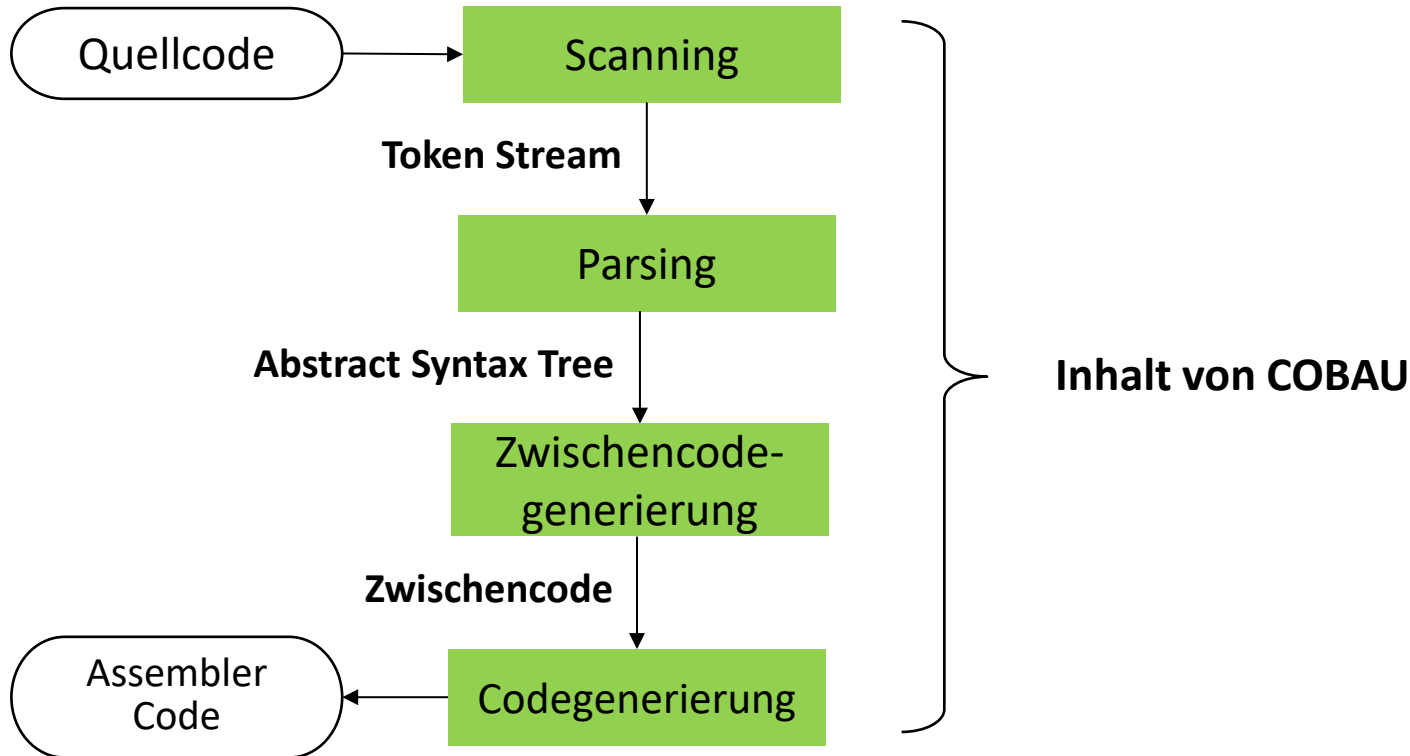
- Was ist ein Compiler?
- Übersicht «MiniJ»
- Zielplattform - x86Lite
- x86Lite - Codebeispiele und Meilenstein 1

Was ist ein Compiler?



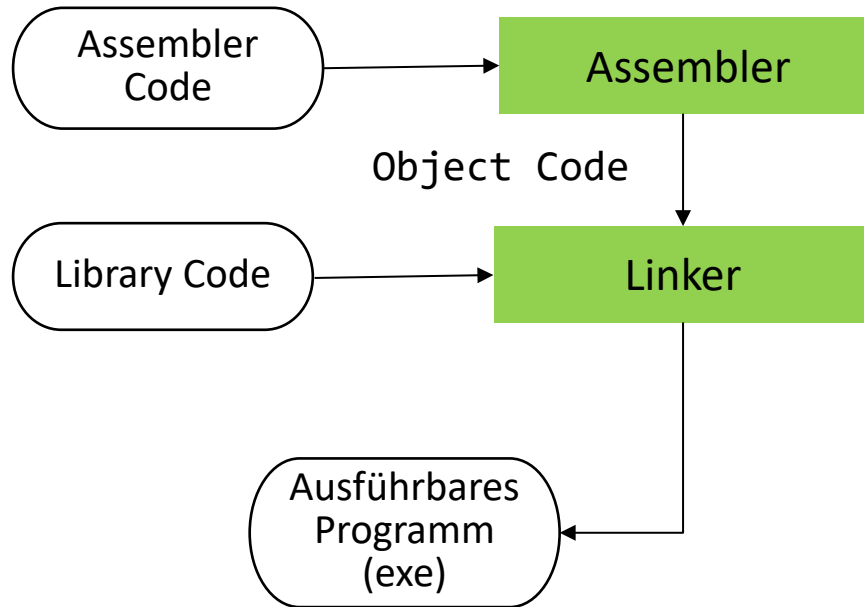
Was ist ein Compiler...

Ein Compiler übersetzt einen Quelltext einer höheren Programmiersprache, z.B. Java oder C# in Maschinencode (oder Assembler-Code).



Erzeugen von ausführbarem Code

Um aus Assemblercode (Maschinensprache) ein ausführbares Programm zu erzeugen wird noch ein Assembler und ein Linker benötigt.

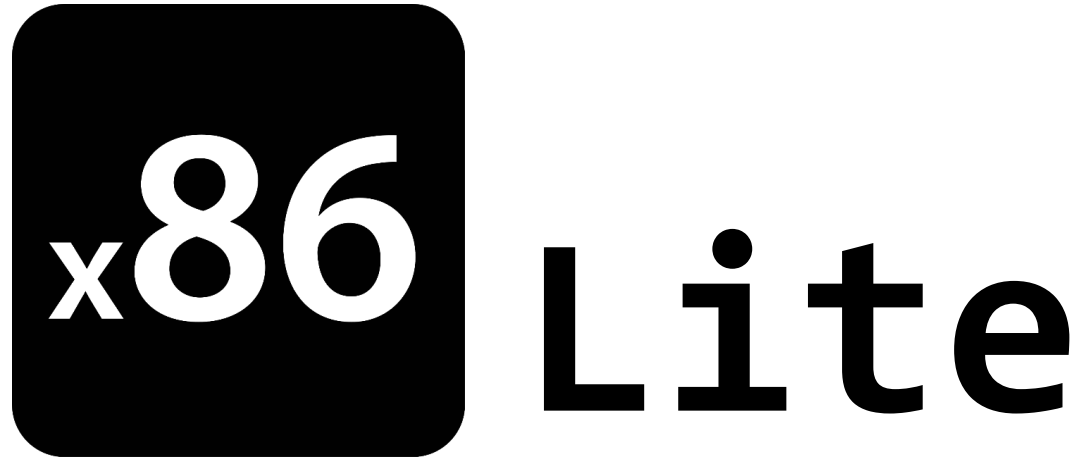


Compiler History in a nutshell

- Die ganz frühen Computer wurden zuerst mit der Eingabe von Binär-Code (Byte für Byte) programmiert.
- Später, bis in die 1950er Jahre gab es Assembler, welche den Befehlssatz von Prozessoren beherrschten.
- 1952 entwickelte Grace Hopper das «A-0 System» für den UNIVAC I (Unisys). Später hatte sie einen grossen Anteil am Design von Cobol
- 1957 hat IBM die Sprache Fortran entwickelt
- 1960, 1970er: Blüte der prozeduralen Sprache. Es entstanden C, PL/1 aber auch Algol60, Pascal und Modula.
- 1980/1990er: Zwei Jahrzehnte der Konsolidierung, Einführung der Objektorientierung, Java usw.
- Heute: Gibt es Hunderte von Programmiersprachen!

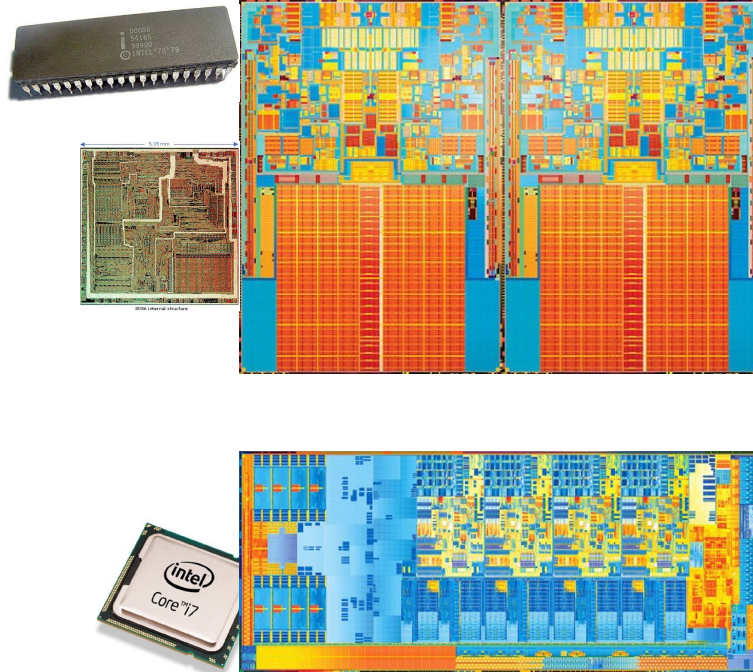


Zielplattform – x86Lite



Intel's x86 Architektur

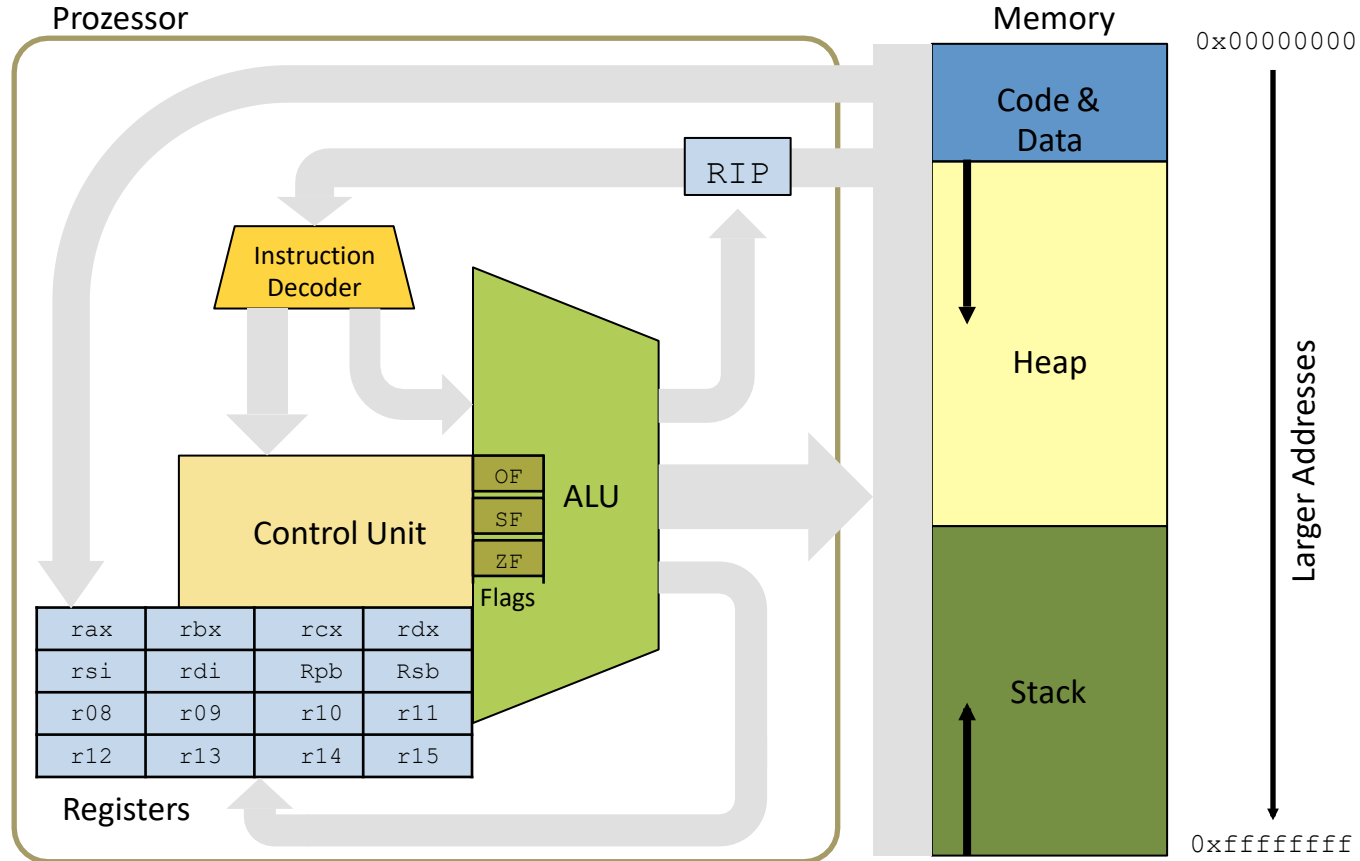
- 1978: Intel introduces 8086
- 1982: 80186, 80286
- 1985: 80386
- 1989: 80486 (100MHz, 1 μ m)
- 1993: Pentium
- 1995: Pentium Pro
- 1997: Pentium II/III
- 2000: Pentium 4
- 2003: Pentium M, IntelCore
- 2006: Intel Core 2
- 2008: Intel Core i3/i5/i7
- 2011: SandyBridge / IvyBridge
- 2013: Haswell
- 2014: Broadwell
- 2015: Skylake (4.2GHz, 14nm)
- AMD has a parallel line of processors



x86 versus x86Lite

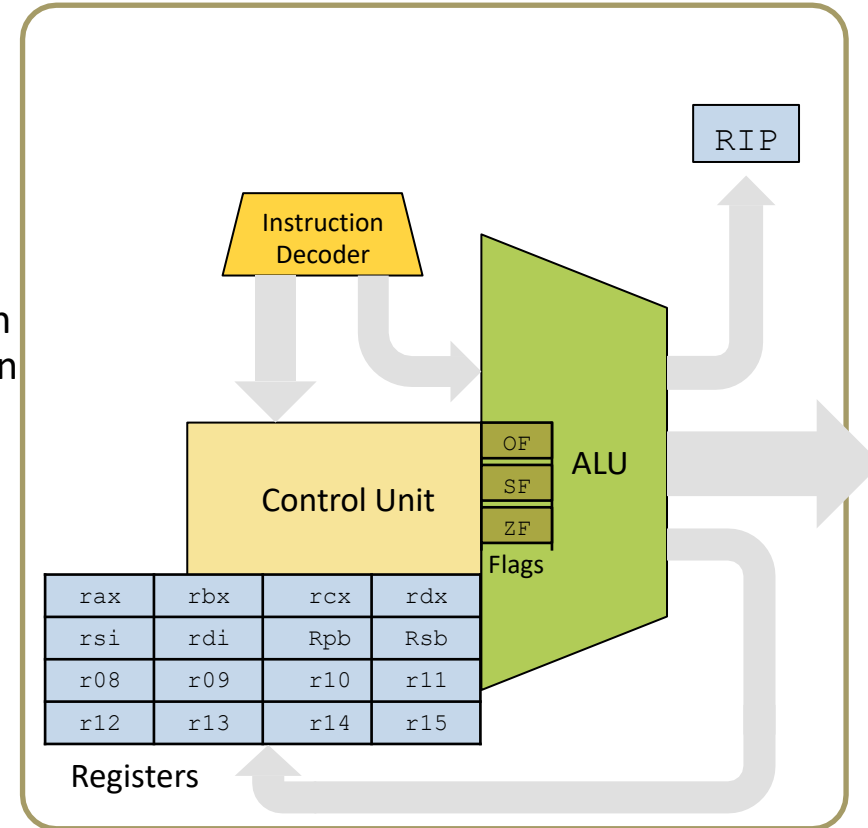
- x86 Assemblersprache ist ziemlich kompliziert!
 - 8-, 16-, 32-, 64-bit Werte und + Floating Point, etc.
 - Intel 64 Architekturen haben eine grosse Anzahl von Funktionen
 - Maschinen Code: Anweisungsgrößen von 1 bis 17 Byte
 - Hohe Komplexität wegen Rückwärts-Kompatibilität
 - Schwer verständlich!
- x86Lite ist ein Subset von x86 und sehr einfach:
 - Nur 64-bit signed Integer Zahlen (Keine floating point, kein 16-bit, usw.)
 - Nur etwa 20 Instruktionen
 - Ausreichend als Zielsprache für allgemeine Computer

x86 – schematischer Aufbau der CPU



Elemente des Prozessors – CPU (Central Processing Unit)

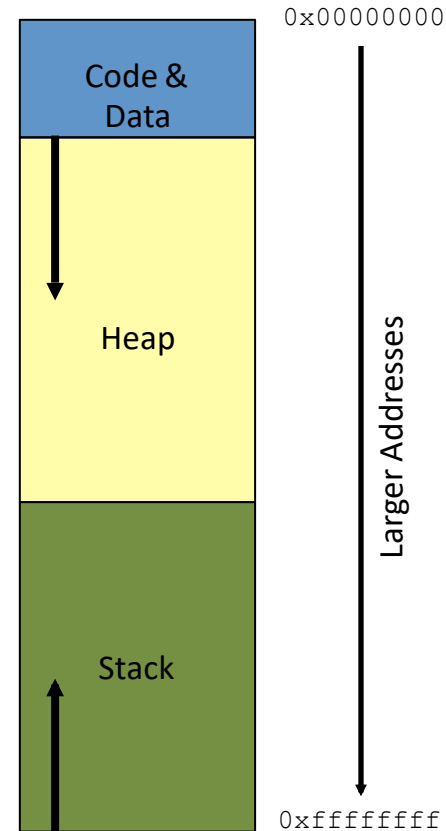
- **ALU:** Arithmetic Logic Unit ist das elektronische Rechenwerk. Beherrscht arithmetische und logische Funktionen(z.B. OR-Verknüpfung)
- **Control Unit:** Steuert den Ablauf der Befehlsverarbeitung. Holt Befehle aus dem Programmcode und führt sie aus.
- **Instruction Decoder:** Übersetzt komplexe Befehle in sogenannte Micro-Instruktionen und zerlegt diese in einzelne Arbeitsschritte.
- **Register:** Register sind Speicherzellen, welche direkt mit den Recheneinheiten des Prozessors verbunden sind.
- **Flags:** Zeigen bestimmte Stati nach Ausführung von Befehlen oder arithmetischen/logischen Operationen an. Zum Beispiel: ZF (zero flag) wenn das Resultat 0 ist.
- **RIP:** Instruction Pointer, zeigt auf die aktuelle Instruktion.



Speicheraufbau eines Programms resp. Prozesses

Es gibt drei wesentliche Speichersegmente für Programme/Prozesse:

- **Code & Data:** Das kompilierte Programm und statische Variablen.
- **Stack:** Parameter und lokale Variablen von Prozeduren und Funktionen.
- **Heap:** Dynamisch allozierte Variablen resp. Objekte.



64-Bit Register des x86 Prozessors

Ein x86-Prozessor stellt, 16, 32 und 64-Bit Register bereit, mit unterschiedlichen Namen. In x86lite betrachten wir nur die 16 64-Bit Register

- `rax` Akkumulator, Operand für ALU und meist Resultat
- `rbx` Base Register, ähnlich Akkumulator: Operand für ALU
- `rcx` Counter, speziell für Zählerstände gedacht (Schleifen)
- `rdx` Data Register, ähnlich `rax/rbx` aber speziell für Daten (anstatt Adressen) gedacht.
- `rsp` Stack Pointer, enthält die aktuelle Adresse des zuletzt auf dem Stack abgelegten Elements (also oberstes Element)
- `rbp` Auch Adresse auf Stack; meist zum Festhalten der Grenze zwischen zwei Stack Frames oder der Grenze zwischen lokalen Variablen und Prozedurparameter verwendet.
- `rsi` Quelle (eng: source) für Stringoperationen
- `rdi` Ziel (eng: destination) für Stringoperationen
- `r08-r15` Allgemein verwendbar Register, z.B. für Gleitkommazahlen
- `rip` Der Instruction Pointer zeigt jeweils auf den aktuellen Befehl

Einführung Befehlssatz von x86Lite

Hier ein einfaches Beispiel eines x86-Befehles:

`mov DEST, SRC` → kopiert SRC nach DEST

- DEST und SRC sind Operanden (*operand*)
- DEST ist ein Speicherort (*location*), ein Register oder eine Memoryadresse
- SRC ist ein Wert (*value*)
 - Er ist Inhalt eines Registers oder einer Memoryadresse
 - Ein Wert kann auch eine Konstante (*immediate*) oder ein Label sein.

Beispiele

- `mov rax, 4` // kopiert den 64-Bit immediate Wert 4 nach rax
- `mov rax, rbx` // kopiert den Inhalt von rbx nach rax

Befehle - Syntaxvarianten

- Es gibt zwei verschiedene x86-Assembler-Dialekte:

1. AT&T-Syntax

Transfer- bzw. Leserichtung



```
movq %rsp, %rbp  
instr source, dest
```

2. Intel-Syntax

Transfer- bzw. Leserichtung



```
mov rbp, rsp  
instr dest, source
```

In x86Lite verwenden wir die Intel-Syntax und nur 64-Bit Instruktionen / Register

Befehlsoperanden

- Sind entweder Daten oder Adressen, auf die ein Befehl angewendet wird.
- x86-Befehle haben null bis maximal drei Operanden.
- Ein Operand kann entweder *immediate*, *register*, oder *memory* sein.
 - Immediate: eine Konstante bzw. Literal (auch *inline value* genannt).
 - Register: Wert in einem Register.
 - Memory: Wert, der an einer Speicheradresse gespeichert ist.

Beispiele:

```
mov rbp, rsp
mov ebx, 0ffh
mov eax, dword prt [ebx]
```

immediate, *register* automatisch detektiert

[] = memory

Size based on register

identifier or specifier:

byte byte = 1 byte

word word = 2 byte

dword long = 4 byte

qword quad = 8 byte

Befehlsarten

Es existieren drei wesentliche Befehlsarten:

1. Befehle zum Datentransfer.
2. Arithmetische und logische Verknüpfungen/Operationen.
3. Kontrollflusssteuerung bzw. (bedingte) Sprünge.

Nachfolgend ein Überblick der wichtigsten Befehle – repräsentativ aber nicht umfassend.

x86Lite Memorymodell

- Das x86Lite Memorymodell besteht auf 2^{64} Bytes nummeriert von `0x0000000000000000` bis `0xffffffffffffffff`.
- x86Lite betrachtet das Memory bestehend aus 64-Bit (8-Byte) Datenblöcken (Quadwords).
- Gültige x86Lite Memoryadressen bestehen aus 64-Bit quadword-aligned Pointers.
- Der Stack wächst immer von hohen nach niedrigen Adressen (siehe auch Folie 15)
- Das Register `rsp` zeigt jeweils auf die Spitze (top) des Stacks
 - Bei `push` wird `rsp` immer um 8 reduziert
 - Bei `pop` wird `rsp` immer um 8 erhöht

Datentransfer

mov – Move

Kopiert Daten von Operand zu Operand.

Achtung! Transfer memory-to-memory mit **movq** nicht direkt möglich; nur mittels „Umweg“ über Register.

; Syntax	; Kopiere Wert in rbx nach rax.
mov <reg>, <reg>	mov rax, rbx,
mov <reg>, <mem>	
mov <mem>, <reg>	; Speichere Wert 5 der
mov <reg>, <const>	; Speicherstelle, die in rax ist.
mov <mem>, <const>	mov byte ptr [rax], 5

Datentransfer

push – Push stack

Legt Operand oben auf Stack ab, nachdem zuerst der Stack Pointer (SP) dekrementiert wurde; z.B. um 4 bei Wortbreite von 32 Bit; bzw. 8 bei 64 Bit Wortbreite. Arbeitet immer in der Wortbreite der CPU.

; Syntax	; Lege Wert in rax auf Stack.
push <reg>	push rax
push <mem>	
push <const>	; Lege Wert der Speicherstelle, die
	; in rax ist auf Stack.
	push [rax]

Datentransfer

pop – Pop stack

Entfernt oberstes Element vom Stack (4 oder 8 Byte je nach Wortbreite) und kopiert Wert in Operand. Danach wird Stack Pointer (SP) entsprechend inkrementiert.

; Syntax

push

<reg>

push

<mem>

; Entferne Wert von Stack nach rax.

pop rax

; Entferne Werte vom Stack und lege ihn
; in die Speicherstelle, die in rax ist.

pop [rax]

Integer mit Vorzeichen

Integer OHNE Vorzeichen :

- 64bit; von 0 .. $2^{64}-1$

Integer MIT Vorzeichen:

- Das «höchste Bit» (MBS) wird als Vorzeichen verwenden 1=negative, 0=positiv
- Also bleiben nur 63 Bit für die Zahl: $-2^{63} - 2^{63}-1$
- Bei arithmetischen Operationen mit signed (vorzeichenbehaftet) Integer müssen also Operation verwendet werden, welche das Vorzeichenbit nicht verändern.
Zum Beispiel sar anstelle von shr oder idiv anstelle von div.

Arithmetische Operationen

add – Integer Addition; Ergebnis im ersten Operand.

sub – Integer Subtraktion; Ergebnis im ersten Operand.

```
add rax, rbx      // rax ← rax+ rbx
sub rsp, 4         // rsp ← rsp- 4
```

inc, dec – Integer Inkrement, Dekrement des Operanden.

imul – Integer signed Multiplikation; Ergebnis in ersten Operand.

mul – Integer unsigned Multiplikation; Ergebnis in ersten Operand.

```
imul <reg>,<reg>[,<const>] ; 2. * 3. Operand = 1. Operand
imul <reg>,<mem>[,<const>]
```

idiv – Integer signed Division mit 128Bit-Zahl in rdx:rax

div – Integer unsigned Division mit 128Bit-Zahl in rdx:rax

```
idiv <reg> ; rdx:rax / <reg>; rax=Resultat, rdx=Rest
idiv <mem> ; rdx:rax / <mem>; rax=Resultat, rdx=Rest
div  <reg> ; rdx:rax / <reg>; rax=Resultat, rdx=Rest
```

Beispiel mit idiv (signed division) & div (signed division)

- Idiv arbeitet immer mit 128Bit-Zahlen als Dividend, welcher in den Registern rdx:rax abgelegt sein muss. rax enthält die ersten 64 Bit, rdx die zweiten 64 Bit der 128Bit-Zahl.
- Der Divisor ist dann eine 64Bit-Zahl welche in irgendeinem Register abgelegt sein kann.
- Also rdx:rax / [reg]. Arbeiten wir nur mit 64Bit-Zahlen, dann enthält rax den Dividenten und rdx muss 0 sein.
- Da idiv mit Integer (ganzen Zahlen) rechnet, gibt idiv das Resultat in rax zurück und einen allfälligen Rest in rdx.

Beispiel: $3456 : 19 = 181 \text{ Rest } 17$

```
mov rdx, 0
mov rax, 3457
mov rcx, 19
```

```
idiv rcx /* man hätte hier auch div rcx schreiben können */
```

rax enthält danach das Resultat der Division: 181

rdx enthält den Rest der Division: 17

Logische Operationen

and – Bitweise Konjunktion; Ergebnis im ersten Operand.

or, xor – Bitweise Disjunktion, Kontravalenz; Ergebnis im ersten Operand.

not – Bitweise Negation.

neg – Zweierkomplement (zur Darstellung vorzeichenbehafteter Integer).

shl, shr – Bitweise links resp. rechts schieben

sal, sar – Bitweise links resp. rechts schieben mit signed Integerzahlen

; Beispiele

and rax, 120 ; and-Operation von rax und 120, Resultat in rax.

xor rax, r08 ; XOR der Inhalt der Register von rax und r08,
Resultat in rax

shr rax, 1; Dividiere Wert (ohne Vorzeichen) in rax mit 2

shl rbx, 3; Multipliziere Wert in rbx mit 2^3 also 8

sar rax, 1; Dividiere Wert mit Vorzeichen in rax mit 2

Codeblöcke, Labels & Sprungoperationen

- x86 Assemblycode ist aus *labeled blocks* aufgebaut

```
label01:  
    <instruction>  
    <instruction>  
    ...  
    <instruction>  
  
label02:  
    <instruction>  
    <instruction>  
    ...  
    <instruction>
```

- Labels zeigen Code-Positionen an, die Sprungziele sein können (entweder durch bedingte Verzweigungsbefehle oder Funktionsaufrufe).
- Labels werden vom Linker und Loader wegübersetzt - Anweisungen werden live im Heap im "Code-Segment" gespeichert.
- Ein x86-Programm beginnt seine Ausführung an einem bestimmten Code-Label (normalerweise "main").

Sprungoperationen

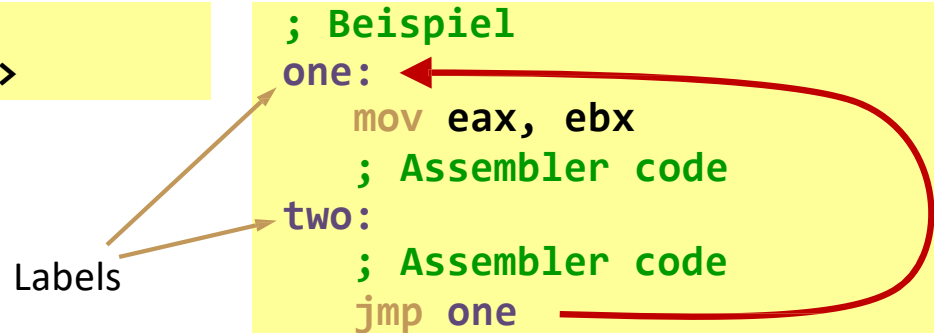
jmp – Jump

Lade Instruction Pointer mit Adresse des Befehls, der unter genanntem Label zu finden ist. Dadurch Sprung zum Befehl mit Label.

; Syntax
jmp <label>

Labels

; Beispiel
one:
 mov eax, ebx
 ; Assembler code
two:
 ; Assembler code
 jmp one



Sprungoperationen

jcondition – Conditional Jump

Basierend auf Wert im Spezialregister *Machine Status Word* (MSW), wenn Test true liefert, dann Lade Instruction Pointer mit Adresse des Befehls, der unter genanntem Label zu finden ist. Dadurch Sprung zum Befehl mit Label. Andernfalls, setze mit nächstem Befehl fort.

; Syntax

```
je <label>;    jump    when    equal
jne <label>;    jump    when    not equal
jz  <label>;    jump    when    result zero
jg  <label>;    jump    when    greather than
jge <label>;    jump    when    gr or equal
jl  <label>;    jump    when    less than
jle <label>;    jump    when    less or equal
```

; Beispiel: Springe

```
; zu one wenn
; r08 <> rax
one:
    mov rax, r08
two:
    cmp rax, r08
    jne one
```

Sprungoperationen

call, ret – Subroutine call, return

call (eine Prozedur aufrufen): Legt die Adresse des nächsten Befehls auf den Stack und springt dann zur Adresse des Operand (indem EIP entsprechend gesetzt wird).

ret (von einer Prozedur zurückspringen): Lädt zuerst die Sprungadresse vom Stack (pop) in Instruction Pointer, löscht optional Anzahl Bytes des Operanden vom Stack (durch Inkrementieren von ESP) und fährt dann mit nächsten Befehl fort (im EIP).

```
; Syntax  
call <label>  
ret    [<reg>]; Operand optional  
ret    [<const>]; Operand optional
```

Weitere Befehle zur Kontrollflusssteuerung

nop Keine Operation. Tut nichts, ausser auf einen Befehlszyklus zu warten.

Value vs Address

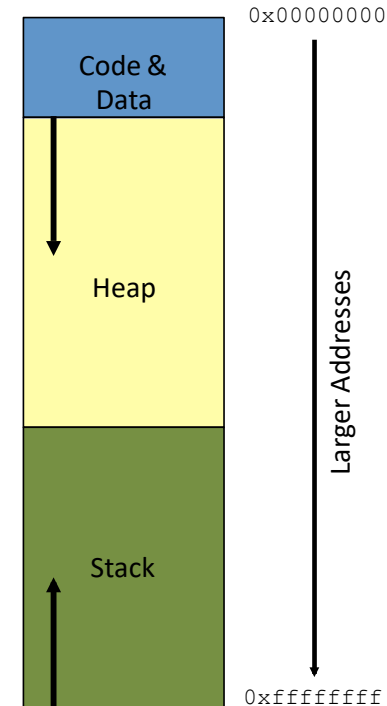
Um adäquat mit einem Assembler umgehen zu können, muss man den Unterschied zwischen einem WERT (value) und einer Memory-ADRESSE (pointer) verstehen. Was ist der Unterschied zwischen einem Pointer und einem Wert.

VALUE - Wert

- Ein Wert ist ein definierter Inhalt – Wert - einer Variable oder eines Registers, z.B. INTEGER, REAL oder CHAR. Beispiel: `mov rax, 'a'`. Hier wird der Code (97) des Zeichens 'a' ins Register rax kopiert.

POINTER – Memoryadresse

- Hier ist der Inhalt eines Registers keine Zahl oder ein Character sondern eine Memoryadresse. Der Inhalt ist zwar auch eine Zahl, diese Zahl entspricht der Nummer des adressierten Memoryblockes.
- Pro Memoria: Bei x86Lite besteht das Memory aus 64-Bit grossen Memoryblöcken (8 Byte). Beispiele:
 - `mov rbx, 0ffffh` ; Kopiere Memoryadresse 0ffffh nach rbx
 - `mov rax, [rbx]` ; Kopiere 8 Byte (Länge Integer) vom Memory, mit Start bei 0ffffh nach rax
 - `add rbx, 8` ; verschiebe den Pointer um 8 Byte



Pointer in Hochsprachen

- Hochsprachen wie C# Java oder C++ gehen unterschiedlich mit Pointern um. In C++ etwa sind **Pointer** Speicheradressen (wie in Assembler) und sie sind auch veränderbar.
- Java kennt keine klassischen Pointer sondern nur **References**. References können vereinfacht als typisierte Pointer bezeichnet werden, sind aber nicht manipulierbar. Sonst würde die Garbage Collection nicht mehr funktionieren.
- Einfache Datentypen wie `int`, `float` oder `char` werden immer als Values behandelt. Bei komplexen Datentypen wie Objects werden hingegen immer nur die Reference übergeben, nie die Inhalte des Objects.
- Beispiel:
 `clAuto` sei eine Javaklasse mit den Attributen `Jahr`, `Modell`, `Motor` und der Methode `setModell`
 und `auto01` eine Objektvariable vom Typ `clAuto`

 `auto01 = new clAuto() /* erzeugt ein Objekt auto01 */`
- `auto01` ist eine Reference auf das Objekt. Die Variable enthält nicht das ganze Objekt sondern nur die Memoryadresse im Heap, wo Speicher für das Objekt vom Java Runtime-System reserviert wurde.

Java – Funktionen, Parameter und Objekte

- In Java muss man wegen der impliziten Behandlung von Objekten als Pointer bei Parameterübergaben bei Funktionsaufrufen vorsichtig sein.

`void MyFunction (int MyInt, clAuto MyCar)` sei eine definierte Funktion in Java

`int preis` und `clAuto auto01 = new clAuto()` eine Variable resp. Objekt

Beim Aufruf `MyFunction(preis, auto01)` wird Preis als Value (by value) und Auto01 als Pointer (by reference) übergeben.

- Dieser Effekt ist unter dem Begriff Aliasing bekannt. Veränderungen bei Attributen des Objekts wirken sich immer auf das ganze Programm aus. Will man das vermeiden, muss das Objekt lokal alloziert und dann der Inhalt des referenzierten Objekts in das lokale Objekt kopiert werden.
- In vielen anderen Hochsprachen müssen die Pointer explizit ausgewiesen werden und die Parameterübergabe erfolgt standardmässig immer by value so auch in MiniJ, siehe auch Kapitel 2.

x86Lite – Codebeispiele und Meilenstein 1



NASM – Assembler und Referenzen

NASM – Netwide Assembler

- In COBAU verwenden wir den Assembler NASM (Netwide Assembler, siehe <https://www.nasm.us/doc/>).

x64dbg & gdb – Debugger für Windows & Linux

- Zum Debuggen, können einen Debugger x64dbg (Windows <https://x64dbg.com/#start>) / gdb (Linux <https://www.gnu.org/software/gdb/>) verwenden. Wir empfehlen aber eher, die `_write` Routine oder den `exit` Befehl zu verwenden.

x86-Referenz

- Hier finden Sie eine gute Anleitung und eine Referenz für die wichtigsten x86-Befehle: [https://de.wikibooks.org/wiki/Assembler-Programmierung_f%C3%BCr_x86-Prozessoren/ Druckversion](https://de.wikibooks.org/wiki/Assembler-Programmierung_f%C3%BCr_x86-Prozessoren/Druckversion).
- Bitte beachten Sie, dass die meisten Beispiele für 32-Bit Rester geschrieben sind. Im Anhang A finden Sie eine Übersicht der Bezeichnungen der x86 Standardregister für 64-, 32- und 16-Bit.

Aufbau eines Sourcefiles in NASM

Ein Codefile in Assembler (NASM) besteht aus mehreren Abschnitten (section), Macros und Pseudo Befehlen und x86-Code. Folgende Struktur für NASM in Cobau ist vorgegeben:

```
DEFAULT REL                ; Gibt an, dass relative Adressen verwendet werden.

extern _read                ; Hier werden die externen Funktionen für das Lesen und schreiben von Strings von der
extern _write               ; Konsole importiert. Sind in Cobau für fast alle Beispiele notwendig.
extern _exit                ; Externe Funktion für die Terminierung des Programms. Die Funktionen sind in
                            ; iosyscalls.asm implementiert.

global <label>              ; Macht das Label public
< Const definition >        ; Hier werden mittels NASM Pseudobefehlen, z.B. EQU Konstanten definiert.

Section .data               ; Hier werden Variablen (Integer oder Strings) mit vorgegeben Werten definiert
< Var definition >

section .bss                ; In der Section .bss (Block Started by Symbol) werden die nicht initialisierten
< Var definition >          ; Variablen definiert.

section .text               ; Definiert den Start der Code-Sektion
< x86 Code >
```

Beispiel 1 – Hello World*

Gibt den Text «hello world» auf der Konsole aus.

```
DEFAULT REL           ; defines relative addresses are used; is a NASM command
; import the required external symbols / system calls, you find these function in iosyscalls.asm
extern _write
extern _exit

global _start          ; exports public label _start; NASM command

Section .data
    TEXTHW: db 'hello world', 13, 10    ; defines the string constant TEXTHW with 'hello world'&cr/lf)
                                         ; db is a NASM pseudo command, reserves n bytes
    TXTLEN: equ $-TEXTHW                ; Copies the length of TEXTHW into TXTLEN

section .text          ; defines start of code section
_start:                ; start label
    mov rdi, TEXTHW     ; copy address of the variable TEXTHW into register rdi
    mov rsi, TXTLEN     ; writes the length of TEXTHW into register rsi
    call _write          ; now calls function _write to write to console (stdout)

    mov rdi, 0           ; terminate program with exit 0
    call _exit
```

*NASM ergänzt das q für 64-Bit Befehle automatisch, aus mov wird movq

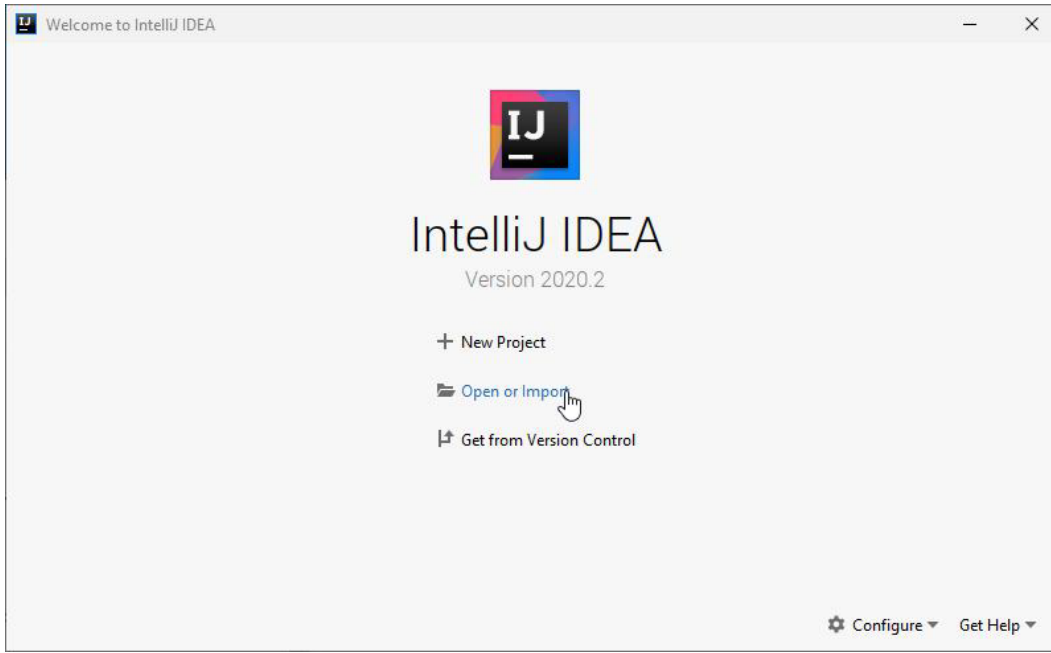
x86Lite: IntelliJ & NASM

COBAU Framework

- Um mit dem NASM arbeiten zu können, stellen wir für COBAU ein Framework bereit, in welchem NASM bereits integriert ist. Das Framework können Sie als ZIP-File COBAU-MS1.zip in ILIAS aus dem Ordner «Aufträge zum Compilerprojekt» herunterladen.
 - **Laden Sie das ZIP-File herunter und entzippen Sie es im gewünschten Verzeichnis**

IntelliJ IDEA

- Um mit dem COBAU-Framework arbeiten zu können, benötigen Sie IntelliJ IDEA.
 - **Laden Sie IntelliJ von <https://www.jetbrains.com> herunter und installieren Sie die Software.**

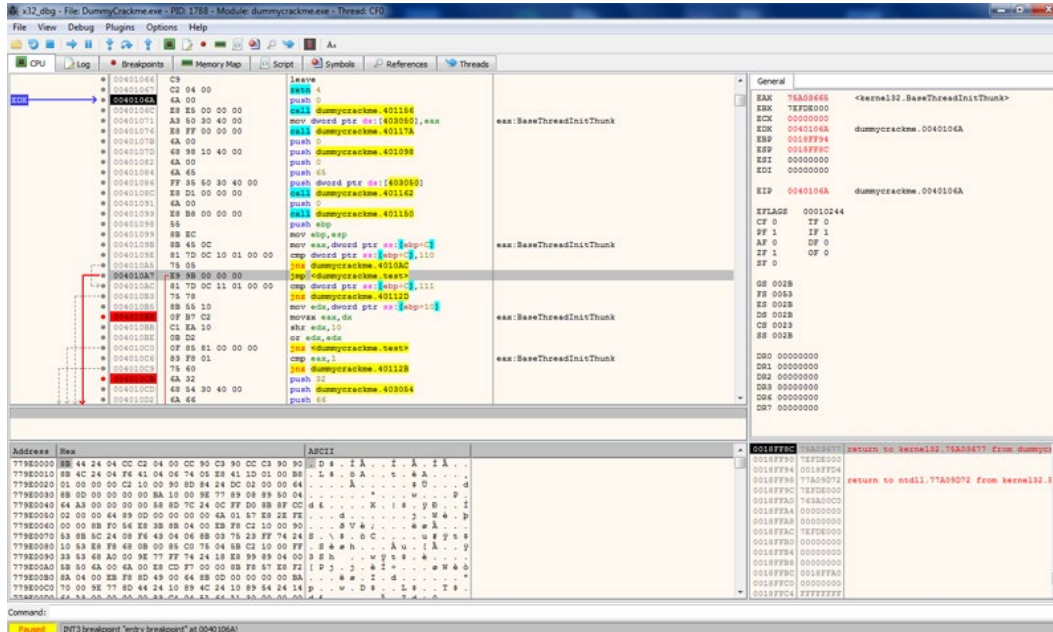


- Nach der Installation öffnen Sie mit „Open or Import“ den Ordner **CompilerFw** .
- Einen Build können Sie auf der rechten Seite über das Menü „Gradle“ ausführen. Pfad: „Tasks.build.build“, einfach Doppelklicken.
- Das Executable wird im Ordner „build“ abgelegt.

- Das Beispiel Hello World, wie auch die übrigen Beispiele finden Sie in IntelliJ im Ordner „src“. Eine vollständige Beschreibung des Cobau-Frameworks finden Sie im File COBAU Framework.pdf im Ordner „Aufträge zum Compilerprojekt “.

Machen Sie keine Veränderung am Build-System. Ändern Sie nur Dateien im Verzeichnis «src». → Kurzdemo IntelliJ

Kurzdemo x64dbg



Wichtigste Befehle

- Executable mit „Datei.Öffnen“ im Verzeichnis build öffnen
- Dann „Debug.Ausführen“ bis Benutzercode anwählen.
- Dann mit F7 oder „Debug.Einzelschritt“ den Code durchlaufen.
- Die Register können im Fenster rechts betrachtet werden.

Alternative

- „exit“-Befehl oder
- „_write“ Befehl verwenden
- Beispiel siehe helloworld.asm

Beispiel 2 – Echo

Liest und schreibt einen String von der Konsole und gibt ihn wieder aus

```
DEFAULT REL           ; defines relative addresses are used; is a NASM command
; import the required external symbols / system calls, you find these function in iosyscalls.asm
extern _read
extern _write
extern _exit

global _start          ; exports public label _start; NASM command
    LENGTH EQU 128     ; definition constant LENGTH (buffer length);
                        ; EQU is a pseudo command of NASM to define constants
section .bss           ; defines start of section of uninitialized data (Block Started by Symbol)
    alignb 8           ; align to 8 bytes (for 64-bit machine), alignb is a NASM Macro (pseudo command)
    BUFFER resb LENGTH ; reserves 128 (length) byte for the variable buffer; resb is a NASM pseudo command

section .text          ; defines start of code section
_start:               ; start label
    mov rdi, BUFFER    ; copy address of variable BUFFER into register rdi
    mov rsi, LENGTH    ; copy value of LENGHT into register rsi
    call _read         ; now calls function _read to read from console (stdin)

    mov rdi, BUFFER    ; copy address of variable BUFFER into register rdi
    mov rsi, rax        ; register rax contains the number of typed char, copy value of rax into register rsi
    call _write        ; now calls function _write to write to console (stdout)

    mov rdi, 0         ; terminate program with exit 0
    call _exit
```

Beispiel 3 – Loop (1)

Schreibt die Ziffern 0 – 9 auf die Konsole

```
; loop.asm - Example of a loop implementation in assembler  
; This program outputs the numbers 0 to 9 in a loop.
```

```
DEFAULT REL                ; defines relative addresses (required for MacOS)
```

```
; import the required external symbols of the system calls abstractions  
extern _read  
extern _write  
extern _exit
```

```
; export entry point  
global _start                ; exports public label _start
```

```
section .data                ; defines start of section of initialized data  
    alignb 8                 ; align to 8 bytes (for 64-bit machine)  
    CRLF      db  10, 13     ; carriage return (CR) / line feed (LF)  
    CRLF_LEN   equ $-CRLF    ; current position ($) minus address of CRLF => 2 bytes
```

```
section .bss                 ; defines start of section of uninitialized data  
    alignb 8                 ; align to 8 bytes (for 64-bit machine)  
    BUFFER resb 2            ; output buffer (2 bytes)
```

Beispiel 3 – Loop (2)

Schreibt die Ziffern 0 – 9 in die Konsole

```
section .text                ; defines start of code section
_start:                     ; program entry point
; loop initialization
    mov r12, 0;                ; initialize loop register (r12)
    mov byte [BUFFER + 1], ' ' ; set second byte of buffer to space character
    jmp cond                 ; check condition before first iteration

; loop iteration: compute ascii character match the value of the loop counter
loop:  mov r13, r12            ; copy r12 to r13
      add r13, '0'            ; add the ascii value (48) of '0' to r13
      mov [BUFFER], r13b      ; copy lowest byte of r13 to first byte in buffer
      mov rdi, BUFFER         ; copy address of variable BUFFER into register rdi
      mov rsi, 2              ; length of output: 2 (digit + ' ')
      call _write             ; now call function _write to write to console (stdin)
      add r12, 1              ; increment loop counter

; loop condition check
cond:  cmp r12, 10             ; compare loop register (r12) to 10
      jl  loop                ; perform loop iteration if r12 <= 10

; output a newline
      mov rdi, CRLF           ; copy address of variable BUFFER into register rdi
      mov rsi, CRLF_LEN       ; length of output: 3 (number + CRLF)
      call _write             ; now calls function _write to write to console (stdin)
; terminate program with exit code 0
exit:  mov rdi, 0              ; terminate program with exit code 0
      call _exit
```

Meilenstein 1 – Erstellung eines Assemblerprogramms

- In der Aufgabe zum Meilenstein 1 erstellen Sie ein einfaches Assemblerprogramm mit den bekannten Befehlen für x86Lite. Das Programm soll eine durch den Benutzer eingegebene Zahl mit fünf multiplizieren und auf der Konsole wieder ausgeben.
- Um mit der Implementierung zu starten, können Sie das im Framework abgelegte Assemblerfile «multiply.asm» benutzen.
- Anzahl Punkte: 20

Lesen Sie die vollständige Aufgabe im Dokument COBAU-MS1.pdf

Quellenangaben, Literatur

- [1] Aho, Lam, Sethi, Ullmann; Compiler – Prinzipien, Techniken und Werkzeuge, Pearson Studium, 2008
- [2] Zhendong, Grosser; Compiler Design, HS 2019 Vorlesung, ETH Zürich
- [3] Möller Th.; Assembler –x86-64 ISA, Uni Basel, 2019

Anhang A – x86 Standardregister

