

**Grundlagenpraktikum: Rechnerarchitektur****Arbeitsblatt 1**

25.04.2022 - 01.05.2022

**Aufgabentypen:**

- Typ **T**: Aufgaben zur Bearbeitung während des Tutoriums (beginnend ab Woche 2).
- Typ **S**: Aufgaben zum Selbststudium, welche meist durch Videos begleitet werden.
- Typ **P**: Programmieraufgaben, die mithilfe des Aufgabentesters oder (auf späteren Blättern) mit bereitgestellten Materialien bearbeitbar sind. Zwecks Notenbonus bewertete Programmieraufgaben werden *nur über den Aufgabentester* angenommen und bewertet!
- Typ **Q**: Quizze auf der Praktikumswebsite.
- Typ **X**: Zusatzaufgaben, die das Verständnis erweitern, aber nicht prüfungsrelevant sind.

**S1.1 Binärsystem und Zweierkomplement**

Vervollständigen Sie folgende Tabelle, wobei die Binär-/Hexadezimalrepräsentation auf 8 Bit beschränkt sein und für negative Zahlen das Zweierkomplement verwendet werden soll.

Binär	Hexadezimal	Dezimal (mit Vorzeichen)	Dezimal (kein Vorzeichen)
0111 1111	0xff	42	128

**S1.2 Unterschiede NASM-Syntax und GNU/Intel-Syntax**

Zu der in der ERA-Vorlesung verwendeten NASM-Syntax ist die in diesem Praktikum verwendete GNU-Intel-Syntax in einigen kleinen Punkten unterschiedlich:

- Wenn bei Speicherzugriffen die Operandengröße explizit angegeben werden muss, muss zusätzlich der Zusatz `ptr` angegeben werden:

```
add qword ptr [rdi], 123
```

Ohne den Zusatz `ptr` würde `qword` als zusätzliches Offset interpretiert werden:

```
add qword [rdi], rax    ist identisch zu    add [rdi + 8], rax
```

- Beim GNU Assembler beginnen lokale Label innerhalb einer Funktion per Konvention mit `.L` (Punkt, großes L). Alternativ können auch numerische Label verwendet werden, wie diese im Manual beschrieben werden<sup>1</sup>.

<sup>1</sup><https://sourceware.org/binutils/docs/as/Symbol-Names.html>

- Mehrere Befehle können durch ; getrennt in eine Zeile geschrieben werden.
- Zeilenkommentare können mit // oder # eingeleitet werden, Blockkommentare mit /\* \*/.

### S1.3 Der 64-Bit Modus von x86

In der Vorlesung *Einführung in die Rechnerarchitektur* haben Sie x86 als 32-Bit-Architektur kennengelernt. Diese haben jedoch Limitierungen, die für heutige Anforderungen nicht mehr genügen. Deshalb wurde von AMD der 64-Bit Modus für x86 eingeführt, welcher später von Intel übernommen wurde. Im Rahmen des Praktikums werden wir den 64-Bit Modus von x86 behandeln, welcher auch unter dem Namen *x86-64* bekannt ist.

1. Welche Vor- und Nachteile hat eine 32-Bit Architektur im Vergleich zu einer 64-Bit Architektur?
2. Betrachten Sie folgende Ihnen bereits bekannte Grafik. Wie verändert sich der Inhalt des Registers *rax* beim Schreiben der Teilregister *al*, *ah*, *ax*, und *eax*?

rax	eax	ax	
		ah	al
rcx	ecx	cx	
		ch	cl
rdx	edx	dx	
		dh	dl
rbx	ebx	bx	
		bh	bl
rsp	esp	sp	spl
rbp	ebp	bp	bpl
rsi	esi	si	sil
rdi	edi	di	dil

r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

3. Einige Instruktionen wie *mov* und *add* erlauben, dass eine Konstante als *immediate*-Operand angegeben werden kann. Welchen Wertebereich haben diese bei x86-32 und wie verändert sich dieser bei x86-64?

### S1.4 Weitere x86-64 Befehle

Ziehen Sie das offizielle *Intel Software Development Manual*<sup>2</sup>, Volume 2, heran und erläutern Sie, was in den folgenden Kurzprogrammen geschieht.

<sup>2</sup><https://intel.com/sdm>

**Hinweis:** Wir empfehlen für die Instruktionsreferenz einen PDF-Viewer welcher die in das PDF integrierte Gliederung anzeigt. Dies erleichtert es Ihnen deutlich, bestimmte Instruktionen zu finden. Alternativ können Sie auch eine inoffizielle (und ggf. falsche) Online-Referenz<sup>3</sup> verwenden.

1. `add rbx,rdx; adc rax, rcx`
2. `imul edx, esi, 10` (Ist diese Multiplikation vorzeichenbehaftet?)
3. `xor eax, eax`
4. `xchg rax, rax`
5. `lea rax, [rbx+rdx*8-18]`
6. `cmovl edi, r9d`
7. `popcnt rax, rdi`

## S1.5 Betriebssystemschnittstelle

In dieser Aufgabe werden wir die Schnittstelle zwischen dem ausgeführten Programm (sog. *User-Space*) und dem Betriebssystem (*Kernel*) genauer betrachten. Dies geschieht über *System Calls*, wo das ausgeführte Programm einen Aufruf in das Betriebssystem tätigt. Während diese Schnittstelle üblicherweise von der C-Standardbibliothek abstrahiert wird und damit direkte Systemaufrufe meist nicht notwendig sind, werden wir in dieser Aufgabe die direkt vom Linux-Kernel bereit gestellte Schnittstelle betrachten.

1. Verwenden Sie man 2 `syscall` und machen Sie sich mit der *Calling Convention* für Systemaufrufe in der x86-64-Architektur unter Linux vertraut. Beantworten Sie folgende Fragen, und ziehen Sie hierzu auch die Architektur-Dokumentation zu der verwendeten Instruktion heran:
  - Mit welcher Instruktion wird ein Systemaufruf ausgeführt?
  - Wie wird spezifiziert, welcher Systemaufruf ausgeführt werden soll?
  - In welchem Register steht das Ergebnis des Systemaufrufs? Wie werden Fehler angezeigt?
  - In welchen Registern werden Argumente übergeben?
  - Welche Register bleiben nach dem Systemaufruf unverändert?
2. Beim Start eines Programms wird die Programmbinary vom Betriebssystem in den Speicher geladen. Hierzu sind insbesondere Informationen über die zu ladenden Segmente und deren Stelle in der Binary sowie den Einsprungpunkt notwendig. Finden Sie diese Informationen für das Programm `/bin/ls` mittels des Befehls `readelf -hW` heraus.

---

<sup>3</sup>z.B. <https://www.felixcloutier.com/x86/>

---

3. Mithilfe des Programms `strace` lassen sich alle Systemaufrufe eines Programms auf der Kommandozeile anzeigen. Führen Sie ein Programm Ihrer Wahl wie folgt aus:

```
strace /bin/ls
```

Können Sie erkennen, wo die C-Standardbibliothek vom Laufzeitlinker geladen wird? Verwenden Sie auch `man 2 <syscall>`, um Informationen über Ihnen nicht bekannte Systemcalls zu erhalten.

4. Lesen Sie sich die Dokumentation zum Syscall `write` (`man 2 write`) durch. Welche Abstraktionen führt die Funktion `printf` durch?

### P1.1 Gauß [2 Pkt.]

Schreiben Sie in x86-64 Assembly eine Funktion `gauss`, welche die Summe der Zahlen von 1 bis `rdi` berechnet und das Ergebnis in `rax` schreibt.

**Aufgabentester:** Nutzen Sie den Aufgabentester, um diese Funktion zu entwickeln.

### P1.2 Fakultät [4 Pkt.]

Implementieren Sie in Assembler eine Funktion mit folgender Signatur, welche  $n!$  berechnet. Die Fakultät von 0 ist 1. Falls das Ergebnis nicht mehr im Datentyp des Rückgabewertes darstellbar sein sollte, soll die maximal darstellbare Zahl zurückgegeben werden. Der Parameter wird in `rdi` übergeben; der Rückgabewert wird in `rdx:rax` erwartet.

```
unsigned __int128 factorial(uint64_t n);
```

### Q1.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)

---