LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

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	***0x7f***	***127***	***127***
1111 1111	0xff	***-1***	***255***
0010 1010	***0x2a***	42	***42***
1000 0000	***0x80***	***-128***	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 ein 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[1].

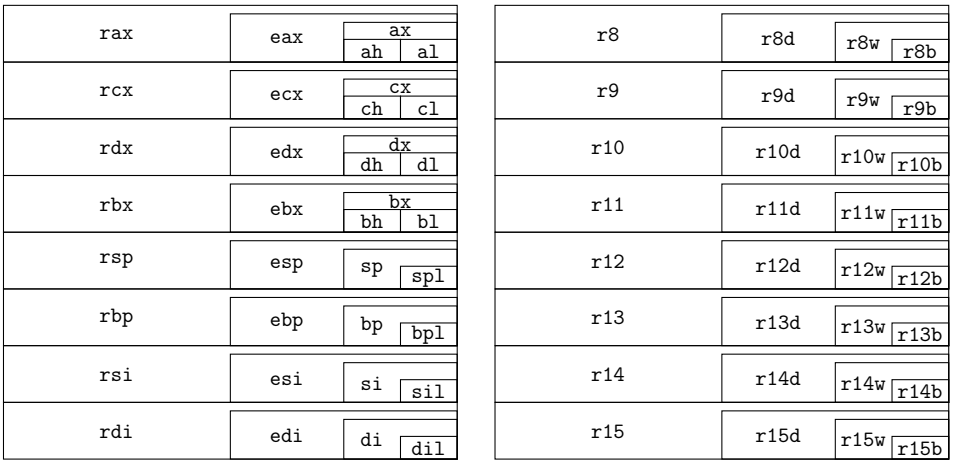
[1] `https://sourceware.org/binutils/docs/as/Symbol-Names.html`

- 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 eingführt, 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?
   - ***Wortbreite; bei 32-Bit ist diese auf 32-Bit beschränkt, weshalb Operationen auf größeren Zahlen nur ineffizient durchführbar sind.***
   - ***Adressraumgröße; bei 32-Bit können max.*** $2^{32}B = 4GiB$ ***adressiert werden, bei 64-Bit*** $2^{64}B = 16EiB$***. Praktisch ist der virtuelle Adressraum bei x86-64 auf 48-Bit (neuerdings auf 57-Bit) beschränkt.***
   - ***Speicherbedarf; ein 32-Bit System benötigt weniger Speicher, da Pointer und Größen von Speicherbereichen nur halb so groß sind.***
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`?



Beim Schreiben von `al`, `ah` und `ax` wird der Rest des Registers `rax` beibehalten. Beim Schreiben von `eax` werden die oberen 32-Bit von `rax` auf 0 gesetzt.

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?

   - ***x86-32: Immediate-Operand hat 32-Bit Wertebereich, also*** $\left[-2^{31}, 2^{31}-1\right]$***.***
   - ***x86-64: Immediate-Operand hat 32-Bit Wertebereich, also*** $\left[-2^{31}, 2^{31}-1\right]$***. Einzige Ausnahme ist die Instruktion `mov`, wo der Immediate-Operand 64 Bit groß sein kann und damit den Wertebereich*** $\left[-2^{63}, 2^{63}-1\right]$ ***hat.***

S1.4 Weitere x86-64 Befehle

Ziehen Sie das offizielle *Intel Software Development Manual*[2], Volume 2, heran und erläutern Sie, was in den folgenden Kurzprogrammen geschieht.

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[3] verwenden.

1. `add rbx,rdx; adc rax, rcx`

   Addiert zwei 128-Bit Zahlen in (`rax:rbx`) und (`rcx:rdx`) und speichert das Ergebnis in (rax:rbx). Der Befehl `adc` ("add carry") führt eine normale Addition durch und addiert zum Ergebnis den Wert des Carry-Flags vor Ausführung des Befehls.

2. `imul edx, esi, 10` (Ist diese Multiplikation vorzeichenbehaftet?)

   Setzt `edx = esi * 10`. Bei einer nicht-erweiternden Multiplikation gibt es keinen Unterschied zwischen einer vorzeichenlosen und einer vorzeichenbehafteten Multipliktion. Deshalb gibt es auch keine `mul`-Instruktion mit 2/3 Operanden.

3. `xor eax, eax`

   Exklusives Oder mit sich selbst – setzt alle Bits eines Registers auf 0. Wird aufgrund des kürzeren Maschinencodes manchmal anstelle von `mov eax, 0` verwendet.

4. `xchg rax, rax`

   Macht nichts - no operation. Der Befehl `nop` ist ein Alias dafür.

5. `lea rax, [rbx+rdx*8-18]`

   Speichert das Ergebnis der Rechnung `rbx+rdx*8-18` in das Register `rax`. Äquivalent wäre:

[2] `https://intel.com/sdm`
[3] z.B. `https://www.felixcloutier.com/x86/`

```
1 mov rax, rdx
2 sal rax, 3
3 add rax, rbx
4 sub rax, 18
```

`lea` lädt im Gegensatz zu `mov` die Adresse seines Operanden, nicht das Ergebnis. Da es keinen Unterschied zwischen Zahlen und Adressen gibt, nutzt der Compiler dies als Trick um Rechnungen zu verkürzen, die als indirekte Referenzierung geschrieben werden können.

6. `cmovl edi, r9d`

   Wenn die Statusflags `l` (less-than) anzeigen (z.B. nach einem `cmp`), dann wird der Wert von `r9d` nach `edi` geschrieben. Achtung: Die oberen 32 Bit von `edi` werden hier immer auf 0 gesetzt. Äquivalent wäre:

```
1   mov edi, edi
2   jge 1f
3   mov edi, r9d
4 1:// continue
```

   Die Verwendung von `cmov` ist vorteilhaft, da hier ein Sprung vermieden werden kann.

7. `popcnt rax, rdi`

   Zählt die Anzahl der gesetzten Bits in `rdi` und schreibt das Ergebnis in `rax`.

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 Aufrufe 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?

     Mit der Instruktion `syscall`.

   - Wie wird spezifiziert, welcher Systemaufruf ausgeführt werden soll?

     Die Nummer des Systemaufrufs wird in Register `rax` geschrieben.

- In welchem Register steht das Ergebnis des Systemaufrufs? Wie werden Fehler angezeigt?

  Das Ergebnis steht in `rax`. Fehler werden dadurch angezeigt, dass in dem Rückgaberegister ein negativer Wert im Bereich von -1–-4095 steht. Im Gegensatz zu den Funktionen der C-Standardbibliothek wird die globale Variable `errno` nicht gesetzt. Siehe auch `man 3 errno`.

- In welchen Registern werden Argumente übergeben?

  Das Argumente stehen in `rdi`, `rsi`, `rdx`, `r10` (!), `r8`, `r9`. Ein Systemaufruf nimmt nicht mehr als sechs Argumente.

- Welche Register bleiben nach dem Systemaufruf unverändert?

  Alle bis auf `rax` (Ergebnis), `rcx` (`rip`) und `r11` (`rflags`).

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 -hlW` heraus.

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?

   - ***Der Systemaufruf schreibt lediglich einen Puffer in einen* File Descriptor *(z.B. `stdout`), führt aber keine Formatierung durch.***
   - ***Der Systemaufruf garantiert nicht, dass alles geschrieben wurde — es kann pro Systemaufruf auch nur jeweils 1 Byte geschrieben werden. Die Standardbibliothek abstrahiert dies und ruft den Systemaufruf so lange auf, bis alles geschrieben wurde.***

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.

Referenzlösung:

```
gauss:
  lea rax, [rdi + 1]
  mul rdi
  rcr rax, 1
  ret
```

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 Paramter wird in `rdi` übergeben; der Rückgabewert wird in `rdx:rax` erwartet.

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

Referenzlösung:

```
factorial:
    cmp rdi, 34
    ja 3f
    xor edx, edx  // Result stored in rdx:rax
    mov eax, 1
    cmp edi, 1
    jbe 2f
1:  // Compute rdx:rax = rdx:rax * rdi
    //   New rdx: rdx*rdi + (rax*rdi >> 64)
    //   New rax: rax*rdi
    mov rcx, rdi
    imul rcx, rdx // First part of new rdx
    mul rdi       // New rax + second part of new rdx
    sub edi, 1
    add rdx, rcx  // New rdx
    cmp edi, 1
    ja 1b         // Break loop if edi <= 1
2:  ret
3:  mov rax, -1
    mov rdx, -1
    ret
```

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