Praktikum Technische Informatik

T3-3

Assemblerebene

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

1 Die Assemblerebene

Inhalt dieses Versuchs ist die Programmierung auf der Assemblerebene. Schon beim Erstellen der winzigen Maschinenprogramme für das Aufgabenblatt `T3-2` dürfte klargeworden sein, dass die Programmierung in der reinen Maschinensprache sowohl extrem (zeit-) aufwendig als auch fehleranfällig ist. Ohne weitere Unterstützung lassen sich auf diese Weise keine Programme mit mehr als einigen hundert Befehlen erstellen.

Andererseits haben Sie beim Erstellen der Maschinenprogramme bereits alle Funktionen kennengelernt, die ein Assembler automatisiert — vom Zusammensetzen von Opcode und Registerangaben zu vollständigen Befehlsworten bis zur Berechnung von Sprungadressen.

Merkmal einer *Assemblersprache* ist die 1:1-Abbildung jedes Assemblerbefehls auf einen Maschinenbefehl. Wegen dieser direkten Zuordnung zu Maschinenbefehlen sind Assembler und ihre Eingabesprachen normalerweise auf eine bestimmte Architektur zugeschnitten. Es gibt aber auch universelle Assembler (wie den GNU-Assembler oder TASM), die für eine Reihe von verschiedenen Architekturen und Prozessoren benutzt werden können. Wichtige gemeinsame Merkmale aller Assemblersprachen sind die folgenden:

- Verwendung von einprägsamen Namen für die einzelnen Befehle (*Mnemonics*)
- Einfache und reguläre Syntax für Befehlsargumente wie Register oder Speicheradressen
- Definition von symbolischen Namen für Konstanten und Sprungmarken
- Unterstützung von Kommentaren und freie Formatierung
- Umrechnung der symbolischen Programmadressen in die wirklichen physikalischen Adressen
- Erstellen von Hilfsdateien, etwa eine Liste aller verwendeten Namen, aller Sprungmarken, usw.
- Voller Zugriff auf alle Befehle und Register des benutzten Prozessors
- Evtl. Unterstützung fortgeschrittener Techniken, etwa das Einbinden mehrerer Quelldateien mittels `include` oder Makrofähigkeit
- Häufig wird der eigentliche Assembler um weitere Tools wie Debugger und Disassembler ergänzt. Damit können Details völlig vom Benutzer ferngehalten werden (etwa die Umrechnung zwischen Byte- und Wortadressen)

Obwohl ein Assemblerprogramm weiterhin auf der Ebene einzelner Befehle geschrieben wird, ist der Produktivitätsgewinn gegenüber der Maschinensprache beträchtlich. Auf der anderen Seite ist die Assemblerprogrammierung natürlich immer noch sehr viel aufwendiger als die Programmierung in Hochsprachen (wie Java oder C usw.). Trotzdem gibt es eine Reihe von guten Gründen, in Assembler zu programmieren:

- es steht (noch) kein geeigneter Compiler für eine Hochsprache zur Verfügung
- kritische Programmanteile erfordern maximale Performance
- Zugriff auf Spezialregister und privilegierte Register, etwa für Gerätetreiber
- eingeschränkte Resourcen an Programm- und Datenspeicher — viele 8-bit Mikrocontroller enthalten weniger als 1KByte RAM

1.1 Format der Assemblersprache

Obwohl jede Assemblersprache auf die Struktur der Befehle der zugrundeliegenden Architektur zugeschnitten ist, ähneln sich die Assemblersprachen für verschiedene Prozessoren doch sehr stark. Fast immer werden die Programme mit genau einer Assemberanweisung pro Zeile geschrieben, und jede Zeile wiederum beginnt mit einer optionalen Marke, gefolgt vom Befehl (Opcode), den Operanden, und einem optionalen Kommentar. Der Assembler `winasm.exe` für den D·CORE verwendet das folgende Format für die Eingabedateien:

```
; write.asm
; einfacher Schreibzugriff auf das RAM ab Adresse 0x8000
;

Start:
          movi   r8, 8            ; R8 = 8
          lsli   r8, 12           ; R8 = 0x8000  (RAM base)
          movi   r0, 0            ; R0 loeschen
          stw    r0, (r8)         ; MEM[ R8 ] = 0
Label1:
          addi   r0, 1            ; R0 += 1
          addi   r8, 2            ; R8 += 2 (naechste Addresse)
          stw    r0, 2(r8)        ; MEM[ R8 + 2 ] = 1
          br     Label1           ; naechste Iteration der Schleife
          halt
Hello:
          .defs  2
          .defw  34               ; Wert 34
          .assho "Hello processor!"
          .defw  0                ; Null-Wort als String-Ende
          .end
```

Die Details finden Sie in der ausführlichen, separaten Beschreibung `t3asm.pdf` für den Assembler. Zusammengefasst gelten die folgenden Regeln für das Eingabeformat:

- *Kommentare* beginnen mit `;` und reichen bis zum Zeilenende
- *Label-Definitionen* sind Strings, die in der ersten Spalte der Datei beginnen und mit einem Doppelpunkt abgeschlossen werden.
- *Hex-Konstanten* werden in der Schreibweise `0xCAFE` erwartet.
- Die *.org-Direktive* sorgt dafür, dass die nachfolgenden Befehle oder Konstanten ab der angegebenen Adresse `<addr>` im ROM abgelegt werden.
- Die *.defw-Direktive* dient dazu, ein bestimmtes Datenwort in die jeweilige Speicherstelle zu schreiben.
- Die *.defs-Direktive* reserviert die angegebene Anzahl von Speicherworten.
- Die *.assho-Direktive* erlaubt es, Zeichenketten im ROM abzulegen, mit jeweils einem ASCII-Zeichen pro Speicherwort.

2 Maschinenarithmetik

Die nächsten Aufgaben dienen dazu, noch einmal einige Aspekte der Zahldarstellung und Integerarithmetik aufzufrischen.

Achtung: Die folgenden Aufgaben bauen aufeinander auf. Denken Sie deshalb daran, alle Programme abzuspeichern und ausreichend zu kommentieren (z.B. die benutzten Register), damit Sie sie wiederverwenden können! Bitte schreiben Sie zusätzlich einen Kommentar-Header, der neben Programmnamen und -funktion auch ihre Namen und Matrikelnummern enthält.

Aufgabe 2.1: Absolutwert (Betrag) Schreiben Sie ein möglichst kurzes Unterprogramm, um den Absolutwert (Betrag) des Inhalts des Registers `R14` zu berechnen und in `R13` zurückzuliefern. Verwenden Sie möglichst wenig Zwischenregister. Wie behandeln Sie den Eingabewert `0x8000`?

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

Aufgabe 2.2: Betrag und Vorzeichen Verwenden Sie das Unterprogramm aus der letzten Aufgabe um den Wert aus Register `R14` von der Zweierkomplementdarstellung in die Betrag-und-Vorzeichen-Darstellung umzuwandeln, wobei das höchste Bit als Vorzeichen interpretiert wird. Die positiven Zahlen bleiben dabei natürlich unverändert. Zum Beispiel soll die negative Zahl -1 von der Zweierkomplementdarstellung `0xffff` in die Betrag-und-Vorzeichen-Darstellung `0x8001` umgewandelt werden, `0xfffe` in `0x8002` usw.

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

Aufgabe 2.3: 32-Bit Addition Schreiben Sie ein Unterprogramm zur Addition von 32-Bit Zahlen. Die Argumente werden in (`R8`,`R9`) und (`R10`,`R11`) übergeben, das Resultat soll in (`R12`,`R13`) sowie dem `C`-Flag abgelegt werden (jeweils MSW,LSW). Wie viele Maschinenbefehle werden benötigt?

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

Aufgabe 2.4: Multiplikation (optional) Viele Mikroprozessoren für eingebettete Systeme enthalten kein eigenes Rechenwerk für die Multiplikation oder Division, da diese Operationen nicht in allen Programmen benötigt werden und die notwendige Hardware sehr aufwendig ist. Auf solchen Rechnern werden Multiplikation und Division statt dessen als Unterprogramme mit fortgesetztem Addieren und Shiften realisiert. Schreiben Sie das notwendige Unterprogramm zur Multiplikation! Nehmen Sie an, dass die Ausgangswerte positiv sind und in den Registern `R10` und `R11` stehen.

Welche Algorithmen kennen Sie, um auch negative Operanden korrekt behandeln zu können? Wie könnte der Befehlssatz des D·CORE verbessert werden, um die Multiplikation zu erleichtern?

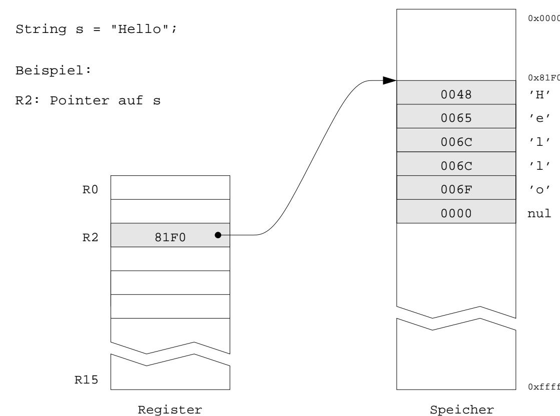


Abbildung 1: Null-terminierter String im Speicher, ein Zeichen pro Wort

3 Indizierte Adressierung und Zeichenketten

Machen Sie sich aus Ihren Vorlesungsunterlagen noch einmal dem Begriff der *Adressierungsarten* vertraut. Beschreiben Sie die Funktion der folgenden Adressierungsarten (aus Tanenbaum, *Computerarchitektur*, S.395ff): unmittelbare Adressierung, direkte Adressierung, Registeradressierung, indirekte Registeradressierung, indizierte Adressierung, basisindizierte Adressierung, Stapeladressierung.

Aufgabe 3.1: Adressierungsarten Welche der Adressierungsarten werden beim D·CORE verwendet:
für die arithmetischen Befehle:
für Speicherzugriffe:
für den jmp-Befehl:
für die Branch-Befehle:

Aufgabe 3.2: Zeichenketten Eine besonders wichtige Anwendung von Arrays und indizierter Adressierung sind Zeichenketten (Strings). In C und verwandte Sprachen wird eine Zeichenkette nur durch ihre Speicheradresse spezifiziert. Alle nachfolgenden Bytes bis zum ersten Null-Byte `0x00` (einschließlich) stellen die Zeichenkette dar. Einige andere Sprachen benutzen statt dessen eine zusammengesetzte Datenstruktur mit einem Integer für die Anzahl der Zeichen und einem separaten Array von Zeichen.

Aufgabe 3.3: strlen() Erstellen Sie ein Assemblerunterprogramm zusammen mit einem aufrufenden Hauptprogramm für die Funktion `strlen()`, die die Länge einer Zeichenkette zurückliefert. Die Startadresse des Strings stehe in `R10`, das Resultat soll in `R11` zurückgeliefert werden. Verwenden Sie die C-Konvention mit null-terminierten Strings und 16-Bit pro Zeichen, wie in Abbildung 1 illustriert.

Einen String bekommen Sie dabei mit der Befehlsfolge

```
.org   0x8000          ; Adresse
.ascii "Ein String"    ; der String
.defw  0               ; terminierende Null
```

in der Speicher. Setzen Sie Sie diese Anweisungen bitte immer ganz an das Ende ihres Programms!

Aufgabe 3.4: strcpy() Schreiben Sie jetzt das analoge Assemblerunterprogramm für die Funktion `strcpy()`, um eine Zeichenkette zu kopieren. Die Startadresse für Original und Kopie stehe in `R10` und `R11`.

Testen Sie Ihre Funktion, indem Sie einen nicht zu langen String, der ab der Adresse 0x8000 im Speicher steht, zuerst in einen Bereich ab der Adresse 0x80A0 und dann direkt hinter sich selbst kopieren (verwenden Sie für den zweiten Teil die Funtion strlen(), um die Länge des Strings zu ermitteln.

Aufgabe 3.5: itoa() Die Umwandlung von numerischen Werten in Zeichenketten ist eine häufig benötigte Operation — zum Beispiel benutzen die Speicherkomponenten in Hades eine solche Funktion, um die Speicherinhalte lesbar darzustellen.

Realisieren Sie die Funktion `itoa()` (Integer to ASCII), die einen Integerwert in einen String zur Basis 16 umwandelt. Zum Beispiel soll die Funktion zu den Eingabewerten 1, 32 und 65533 die Zeichenketten `"0x0001"`, `"0x0020"`, `"0xFFFD"` erzeugen.

Da bisher noch keine dynamische Speicherverwaltung zur Verfügung steht, müssen sowohl die Eingabezahl in `R10` als auch ein Zeiger auf einen ausreichend großen Speicherbereich (7 Worte) in `R11` als Argumente übergeben werden. Natürlich können Sie dabei auf die Funktion aus `T3-2-2.10` zurückgreifen. Der ASCII-Wert für das Zeichen „x“ ist 0x78.

4 Ein- und Ausgabe

Gerade bei der Programmierung von Treibern zur Ansteuerung von I/O-Geräten wird Assemblerprogrammierung oft eingesetzt. Beim Konzept von *memory mapped I/O* werden die Geräte an bestimmten Adressen in den Adressraum des Prozessors eingeblendet und vom Prozessor direkt mit Lese- und Schreibbefehlen angesprochen. Zur Demonstration dieses Verfahrens enthält das Hades-Design `processor-io.hds` neben RAM und ROM zusätzlich zwei I/O-Komponenten: ein einfaches Register zur Ansteuerung einiger LEDs und ein ebenfalls parallel angesteuertes Terminal. Der Adressdekoder in diesem Entwurf ist so eingestellt, dass das ROM im Adressbereich von `0x0000..0x6ffe` aktiviert wird, das Terminal bei Adresse `0x7000`, das Display bei Adresse `0x7002` und das RAM im Bereich von `0x8000..0xfffe`.

Sie können aber diese Aufgaben aber auch mit dem Assembler lösen, der im Emulatorfenster ebenfalls eine Siebensegmentanzeige und ein Terminal hat.

Aufgabe 4.1: Ansteuerung der LEDs Schreiben Sie eine Funktion, um den Inhalt von `R10` auf den LEDs auszugeben. Demonstrieren Sie die Funktion, indem Sie in einer Endlosschleife einen Zähler inkrementieren und den aktuellen Wert dieses Zählers jeweils auf den LEDs ausgeben.

Aufgabe 4.2: Ansteuerung von Speicher vs. I/O Machen Sie sich an dieser Stelle klar, dass trotz der identischen Ansteuerung ein fundamentaler Unterschied zwischen Speicher und I/O-Komponenten wie dem Terminal oder einem Drucker besteht: mehrfaches Schreiben einer Speicherstelle mit demselben Wert bewirkt keine Änderung, während mehrfaches Schreiben eines Wertes in ein I/O-Geräts durchaus mehrfache Wirkung haben kann. Entsprechendes gilt für das Lesen; insbesondere dürfen Speicherbereiche mit I/O-Geräten normalerweise nicht vom Cache abgedeckt werden!

Dies gilt auch für die hier verwendete Ansteuerung des Terminals. Dieses verfügt neben den acht Datenleitungen, die direkt über die Datenbits `7..0` angesprochen werden, zusätzlich über zwei Steuerleitungen `nreset` und `strobe` (clk), die an die Datenbits `9` und `8` angeschlossen sind, siehe Abbildung 2. Für jedes Zeichen muss ein `clk`-Impuls erzeugt werden, indem das Datenbit `8` zuerst auf `0` und dann auf `1` gesetzt wird. Für den normalen Betrieb muss das `nreset` Bit auf `1` gesetzt werden (ansonsten wird der gesamte Bildschirm gelöscht).

Zum Beispiel müssen zur Ausgabe des Zeichens „A“ (Ascii-Code 65 bzw. 0x41) nacheinander die Werte `0x0241`, `0x0341`, `0x0241` an die Adresse `0x7000` geschrieben werden.

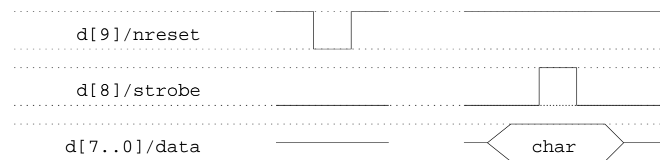


Abbildung 2: Ansteuerung des parallelen Terminals: alle Zeichen löschen (links), Ausgabe eines Zeichens (rechts)

Aufgabe 4.3: putc() Schreiben Sie eine Funktion, um den Inhalt der unteren 8 Bits von `R10` auf das parallele Terminal (an Adresse `0x7000`) auszugeben. Demonstrieren Sie die Funktion, indem Sie in einer Endlosschleife einen Zähler inkrementieren und den aktuellen Wert dieses Zählers jeweils auf des Terminal ausgeben.

Falls beim Testen Ihres Programms Probleme auftreten, weil das Terminal in einen undefinierten Zustand übergeht, kontrollieren Sie den Microcode für den `stw`-Befehl: Der Datenbus darf sich nicht gleichzeitig mit der `nWE` oder `nOE` Leitung ändern.

Aufgabe 4.4: puts() Schreiben Sie jetzt eine Funktion `puts()`, um eine Zeichenkette auf einen Drucker (bzw. ein Terminal) auszugeben. Das Argument mit dem Zeiger auf die auszugebende Zeichenkette wird wiederum in `R10` übergeben.

5 Speicherbereiche und Stack

Da beim von-Neumann-Rechner sowohl die Programme als auch alle Daten im Hauptspeicher liegen, ist die Organisation des Speichers von zentraler Bedeutung. Die in Unix übliche Konvention zur Einteilung der Speicherbereiche ist in Abbildung 3 gezeigt. Dabei werden die folgenden Speicherbereiche (*Segmente*) unterschieden:

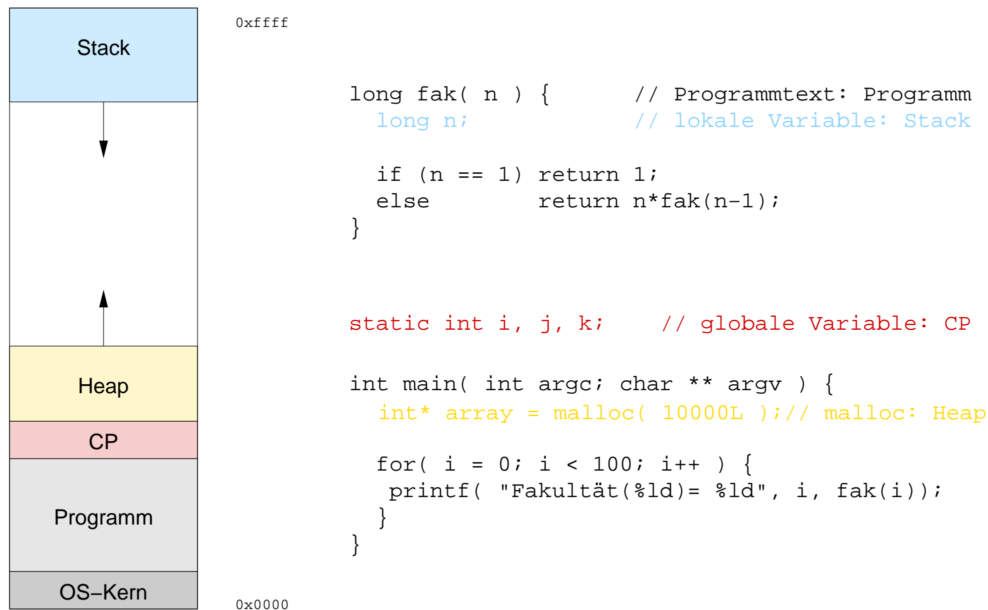


Abbildung 3: Speicherbereiche im Hauptspeicher: Textsegment, Konstantenpool, Heap, Stack

- Das *Textsegment* enthält den eigentlichen Programmtext mit allen Befehlen. Sofern keine selbstmodifizierende Programme zum Einsatz kommen, bleibt das Textsegment während des Programmablaufs unverändert. Es wird häufig ab unteren Ende des Speichers abgelegt.
- der *constant pool* (Konstantenbereich) nimmt alle Konstanten und statischen Variablen des Programms auf. Typ und Anzahl dieser Variablen ergeben sich unmittelbar aus dem Programm. Der Speicherplatz für diese Variablen wird normalerweise direkt oberhalb des *Textsegments* angelegt.
- der *Heap* (Halde) nimmt alle dynamisch zur Laufzeit des Programms erzeugten Variablen bzw. Objekte auf. Der Heap wird oberhalb des Konstantenpools angelegt und wächst nach oben. Für den Heap werden (Betriebssystem-) Funktionen benötigt, um freie Speicherbereiche für neu anzulegende Variablen zu finden und diese auch wieder freigeben zu können.
- der *Stack* (Stapel) wird für die Parameterübergabe zwischen Funktionen und für die Speicherung der lokalen Variablen der einzelnen Funktionen benutzt. Der Stack wird häufig ab oberen Ende des zur Verfügung stehenden Speichers angelegt und wächst mit jedem Aufruf nach unten.

Der im Befehlssatz des D·CORE definierte Befehl `JSR` speichert die Rücksprungadresse immer in Register `R15`. Das bedeutet, dass ohne weitere Maßnahmen immer nur höchstens ein Unterprogramm aufgerufen werden kann, da sonst der zweite Aufruf die Rücksprungadresse des ersten Aufrufs überschreibt. Für geschachtelte Aufrufe muss daher ein *Stapel* bereitgestellt und vom Anwenderprogramm aus verwaltet werden.

Wie bei fast allen RISC-Prozessoren (ausser SPARC), gibt es im Befehlssatz keine weitere Unterstützung für die Stack-Verwaltung. Die Motivation ist, dass der Compiler oft in der Lage ist, soweit möglich alle Parameter über Register zu übergeben und den Stack nur verwendet, wenn sich dies nicht vermeiden lässt.

Aufgabe 5.5: Stack Machen Sie sich aus den Vorlesungsskripten die Funktion eines Stacks klar. Was bedeuten die Begriffe:
caller save:
callee save:

Mit welchen Befehlen kann der D·CORE-Stackpointer auf den in Abbildung 4 verwendeten Wert von `0xfffe` initialisiert werden?

Aufgabe 5.6: push() In den meisten Situationen müssen nicht alle sondern nur einige Register auf den Stack gesichert werden. Notieren Sie als Beispiel die Assemblerbefehle, um den Inhalt der Register `R4`, `R5`, `R10` auf den Stack zu sichern. Per Konvention soll Register `R0` als Stackpointer verwendet werden. Wie behandeln Sie den Stackpointer?

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			

Aufgabe 5.7: pop() Notieren Sie die notwendigen Assemblerbefehle, um den Inhalt der Register `R4`, `R5`, `R10` vom Stack wiederherzustellen:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			

Viele gute Assembler stellen entsprechende Macros zur Verfügung, wobei die betroffenen Register als Argumente übergeben werden. Das gilt für das von uns verwendete Programm. Hier heißen sie `.push` und `.pop`. Als Stackpointer wird als Default das Register R0 angenommen, das auf die zuletzt beschriebene Speicherstelle zeigt. Falls Sie ein anderes Register als Stackpointer verwenden möchten (z.B. das R14), können sie dies dem Assembler mit `.stack R14` mitteilen. Veregessen Sie bitte nicht, ihren Stackpointer auf einen definierten Wert (z.B. 0) zu initialisieren.

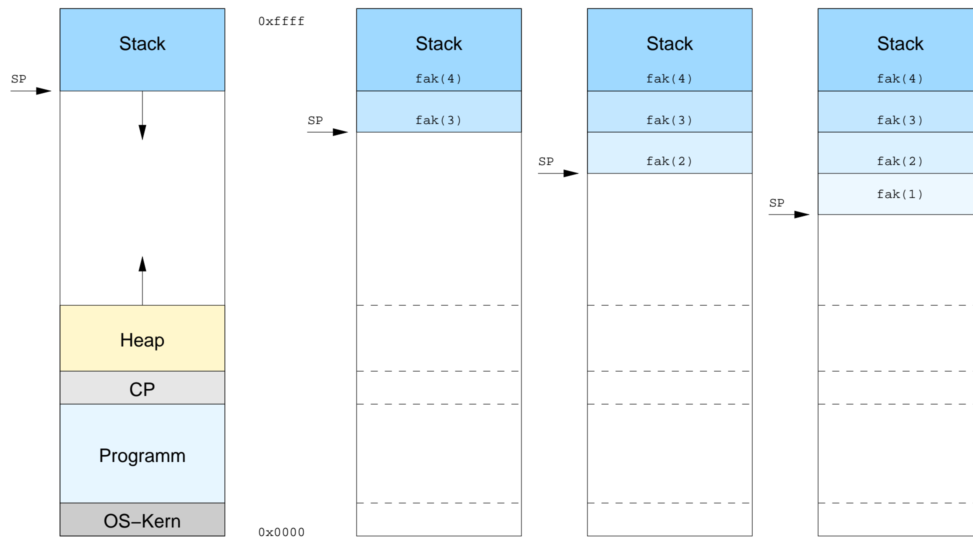
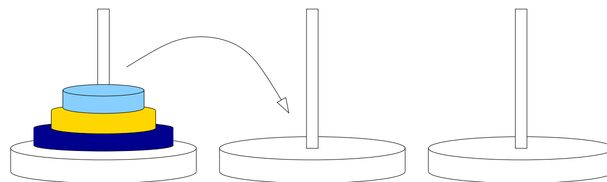


Abbildung 4: Rekursiver Aufruf der Faktultätsfunktion.

Aufgabe 5.8: Türme von Hanoi Das Problem der Türme von Hanoi ist eine der bekanntesten Aufgaben mit einer einfachen rekursiven Lösung.



Es geht dabei darum, die Scheiben von Stab 1 auf Stab 3 zu übertragen, dass jede Scheibe immer auf einem der drei Stäbe liegt und immer eine kleinere Scheibe auf einer größeren liegt. Für drei Scheiben hat man z.B. die sieben Verschiebungen:

$$1 \longrightarrow 3, 1 \longrightarrow 2, 3 \longrightarrow 2, 1 \longrightarrow 3, 2 \longrightarrow 1, 2 \longrightarrow 3, 1 \longrightarrow 3.$$

Das folgende Programm zur Lösung des Problems stammt aus [Tanenbaum]:

```
#include <stdio.h>

/*
   Uebertrage n Scheiben von Stab i auf Stab j. (1 <= i,j <= 3).
*/

  void towers( int n, int i, int j ) {
    if (n == 1) {
      printf( "Uebertrage Scheibe von %d nach %d\n" , i, j );
    }
    else {
      int k = 6 - i - j;
      towers( n-1, i, k );
      towers(   1, i, j );
      towers( n-1, k, j );
    }
  }

void main() {
    towers( 3, 1, 3 );
    }
```

Realisieren Sie das Programm in Assembler und testen Sie es zuerst mit dem angegeben Aufruf `towers(3,1,3)`, der zu insgesamt sieben Ausgaben auf dem Terminal führt.

Aufgabe 5.9: Stacklayout Welche Parameter liegen beim Aufruf von `towers()` wo auf dem Stack? Dokumentieren Sie das von ihnen gewählte Stacklayout:

Aufgabe 5.10: Laufzeit Für größere Parameter wachsen die Laufzeit und der auf dem Stack benötigte Platz schnell an. Erweitern Sie ihre Funktion so, dass die Anzahl der Aufrufe mitgezählt wird. Wieviele Aufrufe ergeben sich für `towers(8,1,3)`? Was folgt daraus für die Komplexität des Algorithmus?

Literaturempfehlungen

- Tanenbaum, Goodman: *Computerarchitektur*, 4. Auflage, Prentice-Hall 1999
- Tanenbaum: *Modern Operating Systems*, Second Ed., Prentice-Hall 2001
- Plauger: *The Standard C Library*, Prentice-Hall 1992