Praktikum Rechnerstrukturen

WiSe2024/2025

4

Assemblerebene
Stack

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen

Universität Hamburg – MIN – Fachbereich Informatik
Arbeitsbereich Technische Aspekte Multimodaler Systeme

https://tams.informatik.uni-hamburg.de

4 - Assemblerebene und Stackverwaltung

Inhalt dieses Bogens ist die Programmierung auf der Assemblerebene mit Stackunterstützung. Schon beim Erstellen der bisherigen kurzen Maschinenprogramme dürfte klargeworden sein, dass die Programmierung in der reinen Maschinensprache sowohl (zeit-) aufwändig als auch fehleranfällig ist. Ohne weitere Unterstützung lassen sich auf diese Weise kaum Programme mit mehr als einigen hundert Befehlen sinnvoll erstellen.

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

4 -9 Die Assemblerebene

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 der Assembler des GNU-Projektes *GAS* oder der *vasm*), 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 physikalischen Adressen
- Erstellen von Hilfsdateien, etwa eine Liste der verwendeten Namen, 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 aufwändiger als die Programmierung in Hochsprachen (wie Java oder C). Trotzdem gibt es Gründe, 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

4 -9.1 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 für den D·CORE, der `t3asm`, verwendet das folgende Format für die Eingabedateien:

```
; strtoint.asm
; Umwandeln eines nullterminierten Strings in eine Zahl:
; der String steht ab Adr. 0x8000 im Speicher und das Ergebnis im Reg. R12

Start:
          movi    r10, 8          ; R10 = 8
          lsli    r10, 12         ; R10 = 0x8000 Stringadresse
          movi    r0, 0           ; zum Vergleich
          movi    r12, 0          ; Zahl initialisieren
Schleife:
          ldw     r1, 0(r10)      ; Character laden
          cmpe    r1, r0          ; = 0?  (Ende des Strings)
          bt      ende
          andi    r1, 0xf         ; Character -> Zahl
          addu    r12, r12        ; 2*r12_alt
          mov     r2, r12         ; Sichern
          lsli    r12, 2          ; 4*r12= (8* r12_alt)
          addu    r12, r2         ; 10*R12_alt
          addu    r12, r1         ; + Zahl
          addi    r10, 2          ; Adresse erhöhen
          br      Schleife
ende:
          halt

         .org     0x8000          ; Adresszähler auf 0x8000
         .ascii   "1324"
         .defw  0                 ; Null-Wort als String-Ende
         .end                     ; Kann auch weggelassen werden
```

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.
- Anweisungen, die mit einem `Punkt` beginnen, stellen sog. Assemblerdirektiven dar. Direktiven sind keine Assemblerbefehle, die in Maschinenbefehle transformiert werden, sondern sind Anweisungen an den Assembler selbst. Im obigen Programm werden beispielsweise folgende Direktiven benutzt:
  - *.org-Direktive*: Sie sorgt dafür, dass die nachfolgenden Befehle oder Konstanten ab der angegebenen Adresse ⟨*addr*⟩ im ROM/RAM abgelegt werden.
  - *.defw-Direktive*: angegebenes Datenwort in die jeweilige Speicherstelle schreiben

- *.ascii-Direktive* erlaubt es, Zeichenketten im ROM/RAM abzulegen, mit jeweils einem ASCII-Zeichen pro Speicherwort.
- *.end* beendet das Assemblieren explizit; im Normalfall nicht erforderlich.

Aufgabe 4.1 Unterprogramme

Vergegenwärtigen Sie sich noch einmal die Arbeitsweise des Befehles `JSR` (*Jump to Subroutine*), und hier insbesondere wie und wohin aus dem Unterprogramm zurückgekehrt wird (siehe auch Abschnitt 7.3.6, Bogen 3).

Was bewirkt das folgende Unterprogramm `LDR5`?

```
LDR5:   ldw     r5, 0(r15)
        addi    r15, 2
        JMP     r15
```

Aufgerufen wird es z.B. durch die Sequenz:

```
        ⋮
        jsr     LDR5
        .defw   0xAFFE   ; oder dezimal .defw 45054
        ⋮
```

Überlegen Sie sich bitte, welche Inhalte nach Ausführung der Assemblerbefehle der folgenden Tabelle in den angegebenen Registern gespeichert sein wird. Beachten Sie, dass bei `JSR` der angegebene Wert `0x0200` bereits die Adresse ist, zu der gesprungen werden soll und **nicht** der dafür erforderliche Offset. Schauen sie sich hierzu bitte die Kurz-Dokumentation der verwendeten Assembler- und Pseudobefehle (t3asm.pdf) an.
Dokumentieren Sie die Wirkung der Befehle (die Registerinhalte) in der folgenden Tabelle:

Adresse	Befehl/Daten	PC	R0	R5	R15
0x01FE	.....	0x0200	0xFFFF	0x1234	0x369C
0x0200	movi r0, 0				
0x0202	jsr 0x0300				
0x0204	.defw 0x8016				
0x0206	ldw r0, 0(r5)				
.....	.....				
0x0300	ldw r5, 0(r15)				
0x0302	addi r15, 2				
0x0304	jmp r15				
.....	.....				
0x8016	.defw 0xAFFA				

Was ist die Wirkung des unter den Adressen `0x0300--0x0304` abgelegen Unterprogrammes, bzw. des obigen Unterprogrammes `LDR5`?

Aufgabe bearbeitet	abzeichnen lassen

4 -9.2 Maschinenarithmetik

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

Hinweis: Für alle noch folgenden Aufgaben nutzen sie bitte den Assembler/Emulator `T3asm`. Geben Sie am besten Ihren Code mit dem integrierten Editor des Assemblers/Emulators `T3asm` ein, übersetzen den Code und führen ihn im Emulator aus.
Ggf. benötigte „Templates“ sind in `t3-hades/programs` hinterlegt. Dort sollten Sie auch eigene Programme abspeichern!

Aufgabe 4.2 Quadrate

In Bogen 2 , Aufgabe 2.3 und Bogen 3, Aufgabe 3.6 hatten wir eine einfache Möglichkeit betrachtet, das Quadrat einer Zahl $n \geq 0$ ohne Multiplikationen zu berechnen.

Schreiben Sie jetzt ein Hauptprogramm, das das Unterprogramm *nh2* aus der Datei `Quadrat.asm` aufruft und berechnen Sie mit seiner Hilfe das Quadrat der Zahlen 5, 55 und 101. *nh2* erwartet im Register `R5` die Zahl n, deren Quadrat berechnet werden soll, und liefert im Register `R7` das Ergebnis zurück. Weiterhin zerstört das Unterprogramm den Inhalt des Registers `R8`.

Label	Adresse	Assemblerbefehl	Kommentar
	0000		
	0002		
	0004		
	0006		
	0008		
	000a		
	000c		
	000e		
	0010		
	0012		
	0014		
	0016		
	0018		
	001a		
	001c		
	001e		

Wie groß darf der Eingabewert n für ein richtiges Ergebnis höchstens sein, wenn

(a) der Wert im Register `R7` für n^2 als Integer interpretiert wird:

(b) der Wert im Register `R7` für n^2 als Unsigned interpretiert wird:

Aufgabe 4.3 Quadrate II

Erweitern Sie das Unterprogramm *nh2*, so dass auch größere Zahlen als Eingabe für ***n*** verarbeitet werden können. Das möglicherweise 32-bit große Ergebnis passt dann natürlich nicht mehr in nur ein Register, sondern es werden zwei Register (z.B. `R6` für die hochwertigen 16-bit und `R7` für die niederwertigen 16-bit) benötigt.
Gehen Sie der Einfachheit halber von einer unsigned Arithmetik aus. Testen Sie Ihr Programm mit $n = 0$, $n = 256$ und $n = 1342$.

Label	Adresse	Assemblerbefehl	Kommentar

Aufgabe bearbeitet	abzeichnen lassen

4 -9.3 Adressierungsarten

Die Befehle des Prozessors verwenden verschiedene *Adressierungsarten*. Die wichtigsten sind (angepasst auf unseren D·CORE):

(1) unmittelbare Adressierung
Im Befehlswort steht ein **Zahlwert** W und ein Register REG. Aus dem Inhalt von REG und der Zahl W wird ein neuer Wert berechnet und nach REG geschrieben. Es ist auch der Fall denkbar, dass REG nicht explizit angegeben wird, weil es sich aus der besonderen Art des Befehls ergibt.

(2) direkte Adressierung
Im Befehlswort steht die **absolute Adresse** ADR einer Speicherstelle in ROM/RAM und ein Register REG.

(3) Registeradressierung
Im Befehlswort stehen normalerweise **zwei Register**, deren Inhalt verknüpft und dann in eins der beiden Register geschrieben wird. Es ist aber auch der Fall denkbar, dass nur eines der Register explizit angegeben wird, und das andere sich aus der besonderen Art des Befehls ergibt. Im Extremfall kann sogar auch die Angabe dieses Registers fehlen.

(4) indirekte Registeradressierung
Im Befehlswort stehen **zwei Register** $REG1$ und $REG2$. Der Inhalt von $REG1$ wird als Adresse einer Speicherstelle in ROM/RAM interpretiert, in die der Inhalt von $REG2$ geschrieben wird, bzw. deren Inhalt nach $REG2$ gebracht wird.

(5) indizierte Adressierung
Wie die indirekte Adressierung, nur steht im Befehlswort zusätzlich noch ein Wert, der auf die ROM/RAM-Adresse addiert wird.

Aufgabe 4.4 Adressierungsarten

Welche der Adressierungsarten werden beim D·CORE verwendet? Beachten Sie, dass auch der PC ein Register ist, obwohl er nicht in der Registerbank liegt. Es sind Mehrfachnennungen möglich.

(a) für die arithmetischen Befehle (z.B MOV und MOVI):

(b) für Speicherzugriffe (LDW und STW):

(c) für den JMP-Befehl:

(d) für den Branch-Befehl BR:

(e) für den JSR-Befehl:

Aufgabe bearbeitet	abzeichnen lassen

4 -9.4 Zeichenketten

Eine besonders wichtige Anwendung von Arrays und indizierter Adressierung sind Zeichenketten (Strings). In C und verwandten 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 für die einzelnen Zeichen.

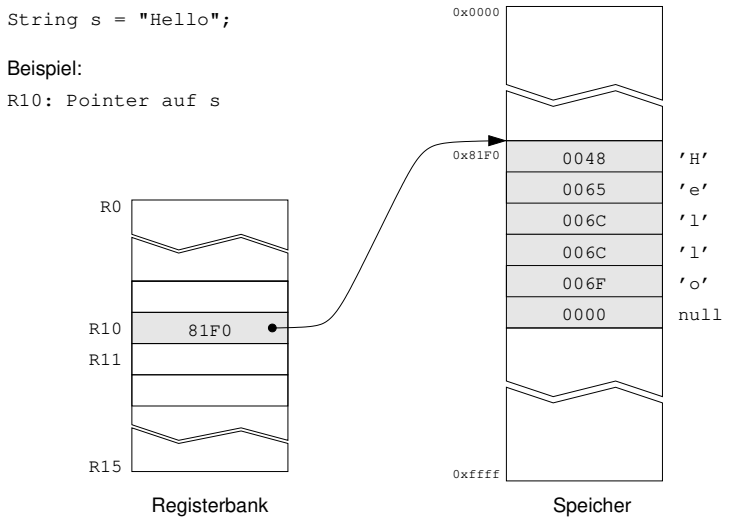


Abbildung 1: Null-terminierter String im Speicher, ein Zeichen pro Wort

Aufgabe 4.5 strlen()

Erstellen Sie ein Assemblerunterprogramm zusammen mit einem aufrufenden Hauptprogramm für die Funktion `strlen()`, das die Länge einer Zeichenkette (ohne das terminierende Nullbyte!) zurückliefert. Die Startadresse des Strings stehe in `R10`, das Resultat (die Länge des Strings) 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
```

ab der Adresse `0x8000` in den Speicher. Setzen Sie Sie diese Anweisungen bitte immer ganz an das Ende ihres Programms! Weshalb ist es wichtig, bzw. was wäre die Konsequenz, wenn nicht?

Einen Rumpf finden Sie in der Datei `m_strlen.asm`, die Sie sich in den Assembler laden können.

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 4.6 strcpy()

Erstellen Sie das Unterprogramm `strcpy(char* str1, char* str2)`, das eine Zeichenkette `str2` in den vorgebenen Speicherbereich `str1` kopiert. Die Startadresse des Strings `str2` erwartet Ihr Unterprogramm in Register `R10` und die Adresse von `str1`, in den kopiert werden soll, in Register `R11`. Vergessen Sie beim Kopieren des Strings nicht das terminierenden Nullbyte.

Testen Sie Ihr Unterprogramm, indem Sie ein dazugehöriges Hauptprogramm entwickeln, dass einen auf Adresse *0x8100* beginnenden String durch zwei Aufrufe Ihres Unterprogramms in einen String auf die Adresse *0x8000* und dann in einen zweiten String an der Adresse *0x8200* kopiert. Ein Template finden sie wieder unter `m_strcp.asm`

Aufgabe bearbeitet	abzeichnen lassen

4 -10 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 2 gezeigt. Dabei werden die folgenden Speicherbereiche (*Segmente*) unterschieden:

- 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 am 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 dem oberen Ende des zur Verfügung stehenden Speichers angelegt und wächst mit jedem Aufruf nach unten.

Der Befehl `JSR` des D·CORE speichert die Rücksprungadresse in Register `R15`. Das bedeutet, dass ohne weitere Maßnahmen höchstens ein Unterprogramm aufgerufen werden kann, da sonst ein zweiter Aufruf die Rücksprungadresse des ersten Aufrufs überschreibt. Für geschachtelte Aufrufe muss daher ein *Stack* 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 – soweit möglich – alle Parameter in Registern übergibt und den Stack nur verwendet, wenn sich dies nicht vermeiden lässt.

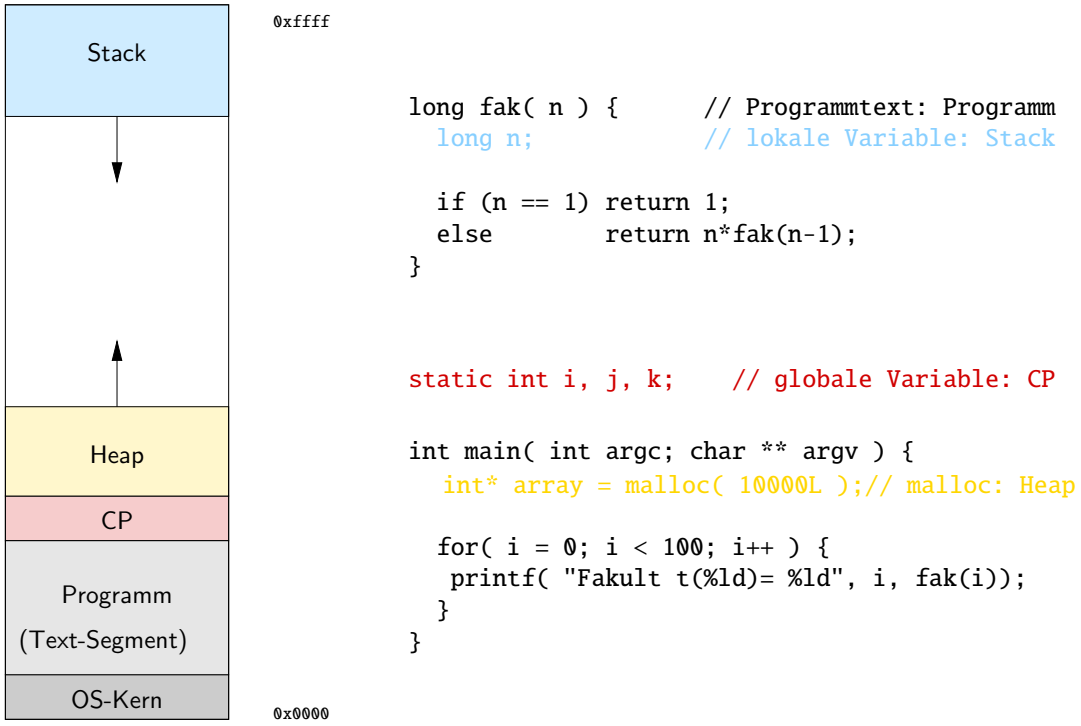


Abbildung 2: Speicherbereiche im Hauptspeicher: Textsegment, Konstantenpool, Heap, Stack

Aufgabe 4.7 Stack

Machen Sie sich aus den Vorlesungsunterlagen die Funktion eines Stacks klar.

(a) Was könnten in diesem Zusammenhang die beiden folgenden Begriffe bedeuten, wenn es um Informationen (z.B. Registerinhalte) geht, die noch benötigt, bzw. überschrieben werden:

- *„caller save“*
- *„callee save“*

(b) Mit welchen Befehlen kann der D·CORE-Stackpointer auf den in Abbildung 3 verwendeten Wert von `0xfffe` initialisiert werden?

(c) Worin liegt unter Berücksichtigung von Abbildung 2 bzw. 3 der konzeptionelle Unterschied, wenn der Stackpointer auf `0xfffe` oder aber auf `0x0` initialisiert wird, bzw. worauf zeigt der Stackpointer in den beiden Fällen (letztes belegtes / erstes freies Element)?
`0xfffe`:
`0x0000`:

Aufgabe bearbeitet	abzeichnen lassen

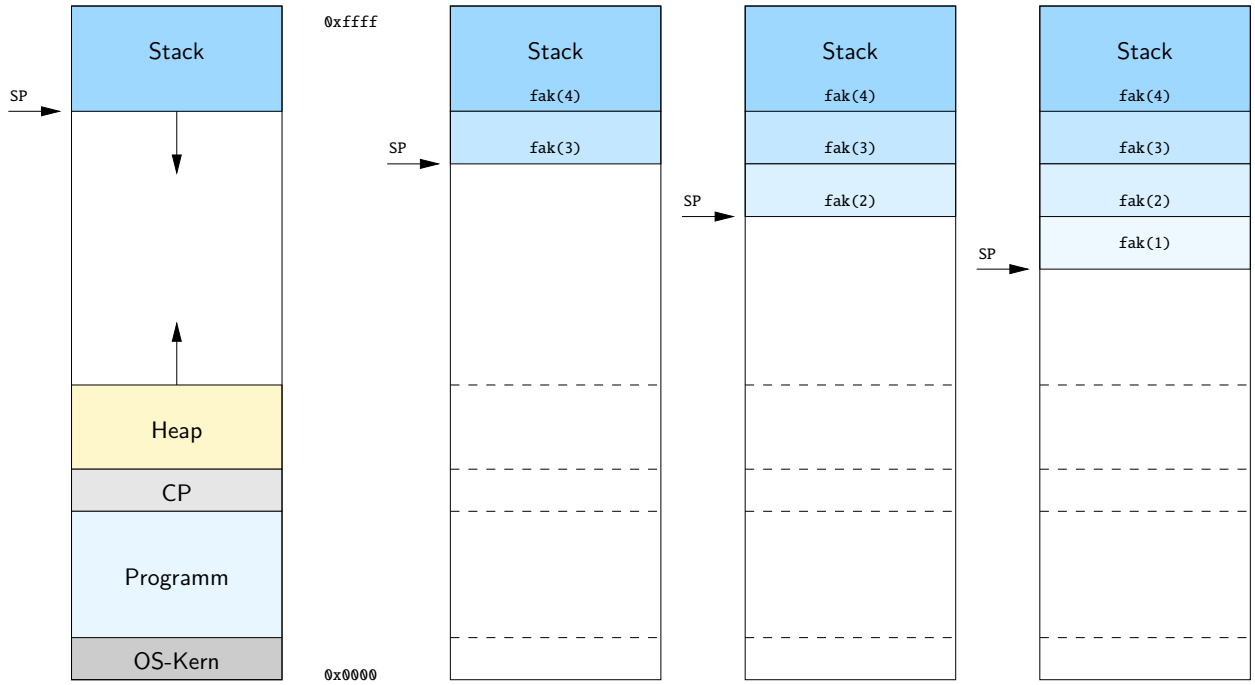


Abbildung 3: Rekursiver Aufruf der Fakultätsfunktion.

Aufgabe 4.8 push(), pop()

Sichern auf dem Stack (push()):
In den meisten Situationen müssen nicht alle, sondern nur einige Register auf den Stack gesichert werden. Die Macros `.push` und `.pop` (siehe Abschnitt 4-10.1) stehen Ihnen hier noch nicht zur Verfügung. Notieren Sie die **Assemblerbefehle**, um den Inhalt der Register `R4`, `R5` und `R10` auf den Stack zu sichern.
Per Konvention soll Register `R0` als Stackpointer verwendet werden und dieser ist, wie in Aufgabe 4.7 (c) vorgestellt, bereits mit dem Wert `0x0000` initialisiert.

Assemblerbefehl	Kommentar	R0 (nach Ausführung)
. . . . .	. . . . .	0xfffe

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

Assemblerbefehl	Kommentar	R0 (nach Ausführung)
. . . . .	. . . . .	

Aufgabe bearbeitet	abzeichnen lassen

4 -10.1 Stackunterstützung des D·CORE-Assemblers

Viele gute Assembler stellen entsprechende Macros zum Sichern und Rücksichern von Registerinhalten auf dem Stack zur Verfügung, wobei die betroffenen Register als Argumente übergeben werden. Das gilt auch für den von uns verwendeten Assembler. Hier heißen die betreffenden Macros `.push` und `.pop` (siehe `t3asm.pdf`). Der Stackpointer wird per Default im Register `R0` gehalten, welches dann entsprechend der Implementierung der Assemblermacros `.push` und `.pop` auf die zuletzt beschriebene Speicherstelle im Stack zeigt. Falls Sie ein anderes Register als Stackpointer verwenden möchten (z.B. das `R14`), können sie dies dem Assembler mit `.stack R14` mitteilen. Vergessen Sie bitte nicht, den Stackpointer auf einen definierten Wert (z.B. 0x0) zu initialisieren.
Sollten sie bei den folgenden Aufgaben die Stackunterstützung benötigen, verwenden Sie die Macros bei der Implementierung.

Aufgabe 4.9 Analyse der Stackunterstützung

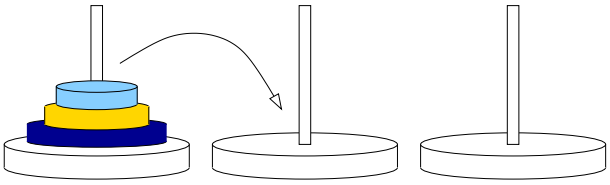
In der Datei `strcat_stack.asm` finden Sie ein Assemblerprogramm, dass die Stackunterstützung des D·CORE benötigt.
Die eigentliche Funktion des Programms ist für die Bearbeitung der Aufgabe nicht relevant. Nur der Vollständigkeit halber sei erwähnt, dass das Programm `strcat_stack.asm` zwei Strings verkettet.
Analysieren Sie den Assemblercode in Bezug auf die Stackoperationen und beantworten Sie folgenden Fragen. Begründen Sie Ihre Antwort. Beachten Sie, dass die Stackoperationen nicht optimiert wurden, um die Struktur klarer hervortreten zu lassen.

- Weshalb benötigt dieses Programm einen Stack?
- Welche Strategie zur Sicherung von Registerinhalten bei Funktionsaufrufen (vgl. Aufgabe 4.7 (a)) wurde hier umgesetzt?
- Sind alle Operationen auf dem Stack durch die Sicherungsstrategie (caller/callee-save) bedingt, oder gibt es da noch weitere?
- Welches Register hält den Stackpointer und worauf zeigt der Stackpointer? Zeigt er auf die letzte belegte oder auf die erste freie Adresse im Stack (vgl. Aufgabe 4.7 (c))

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 4.10 Rekursive Unterprogramme – Türme von Hanoi (freiwillige Zusatzaufgabe)

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, wobei 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:

$$1 \longrightarrow 3, \quad 1 \longrightarrow 2, \quad 3 \longrightarrow 2, \quad 1 \longrightarrow 3, \quad 2 \longrightarrow 1, \quad 2 \longrightarrow 3, \quad 1 \longrightarrow 3$$

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

```
1  #include <stdio.h>
2
3  /*
4    Übertrage n Scheiben von Stab i auf Stab j. (1 <= i, j <= 3).
5  */
6
7    void towers(int n, int i, int j) {
8      if (n == 1) {
9        printf("Übertrage Scheibe von %d nach %d\n", i, j );
```

```
10      }
11      else {
12        int k = 6 - i - j;
13        towers(n-1, i, k);
14        towers(  1, i, j);
15        towers(n-1, k, j);
16      }
17    }
18
19  void main() {
20    towers(3, 1, 3);
21  }
```

Realisieren Sie das Programm in D·CORE Assembler und testen Sie es zuerst mit dem angegeben Aufruf `towers(3, 1, 3)`, der zu insgesamt sieben Ausgaben (s.o.) auf dem Terminal führen sollte.

Hinweise zur praktischen Realisierung

Der Assembler kennt drei Pseudobefehle, um etwas auf dem Display des Emulators ausgeben zu können und zwar:

`.prdez` ⟨***reg***⟩	gibt den Inhalt des Registers ⟨*reg*⟩ als Dezimalzahl aus
`.prnewline`	gibt einen Zeilenumbruch aus
`.prstr` ⟨***reg***⟩	gibt den String aus, dessen Startadresse im Register ⟨*reg*⟩ steht

Es sei betont, dass es sich hier um eine besondere Form der Pseudoanweisungen handelt, die anders als z.B. .push und .pop durch den Assembler **nicht** auf gültige Befehle bzw. Befehlsfolgen des D·CORE-Prozessors abgebildet werden.
Diese Form der Pseudoanweisungen erzeugen normalerweise keinen Maschinencode, sondern dienen der Steuerung des Assemblers bzw. Linkers. Die o. g. Pseudobefehle allerdings dienen der Steuerung des Emulators und werden deshalb vom Assembler in Maschinenbefehle mit Opcode `0x0` umgesetzt (`0x0xxx`), die ausschließlich Anweisungen an den Emulatior darstellen, der dann z. B. die gewünschten Ausgaben im Emulatorfenster erzeugt.
Bei Ausführung durch den D·CORE-Prozessor würde der Befehl mit dem Opcode `0x0` mit Standardmikroprogramm aus der Decode-Phase direkt in die Fetch-Phase des nächsten Befehls verzweigen; würde also ein `NOOP` (no operation) bewirken.

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. Das Hauptprogramm sollte dann diese Zahl zusammen mit einem String (z.B. *Zahl der Aufrufe*=) auf dem Display ausgeben.

(a) Wieviele Aufrufe ergeben sich für `towers(8,1,3)`?

(b) Was folgt daraus für die Komplexität des Algorithmus?