Praktikum Rechnerstrukturen (WS 14/15)

Bogen 4

Assembler-Ebene

Stack

Name: .....

Bogen erfolgreich bearbeitet: .....

Department Informatik, AB TAMS
MIN Fakultät, 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 dürfte klargeworden sein, dass die Programmierung in der reinen Maschinensprache sowohl extrem (zeit-) aufwändig 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 kennen gelernt, 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 aufwändiger 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 `winT3asm.exe` für den D·CORE verwendet das folgende Format für die Eingabedateien:

```
; strtoint.asm
; Unwandeln eines nullterminierten Strings in eine Zahl
; der String steht ab Adresse 0x8000 im Speicher
; Das Ergebnis im Register 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 erhoehen
          br     Schleife
ende:
          halt

         .org    0x8000          ; Adresszaehler 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.

- Die *.org-Direktive* sorgt dafür, dass die nachfolgenden Befehle oder Konstanten ab der angegebenen Adresse `<addr>` im ROM/RAM 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 *.ascii-Direktive* erlaubt es, Zeichenketten im ROM/RAM abzulegen, mit jeweils einem ASCII-Zeichen pro Speicherwort.

2 Adressierungsarten und Zeichenketten

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

- **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.

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

- **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 eins 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.

- **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.

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

Aufgabe 2.1: 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.

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

für Speicherzugriffe (LDW und STW):

für den JMP-Befehl:

für die Branch-Befehle (BR, BT und BF):

für den JSR-Befehl:

Aufgabe 2.1: Ein Unterprogramm
a) Was tut 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
```

Notieren Sie dazu die Registerinhalte nach Ausführung der folgenden Befehle (beim *JSR* ist $0x200$ die Adresse, zu der gesprungen werden soll, **nicht** der Offset):

Adresse	Befehl/Daten	PC	R0	R5	R15
0x009F	.....	0x0100	0xFFFF	0x1234	0x369C
0x0100	movi R0, 0				
0x0102	jsr 0x0200				
0x0104	.defw 0x8010				
0x0106	ldw r0, 0(r5)				
.....	.....				
0x0200	ldw r5, 0(r15)				
0x0202	addi r15, 2				
0x0204	jmp r15				
.....	.....				
0x8010	.defw 0xAFFE				

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.

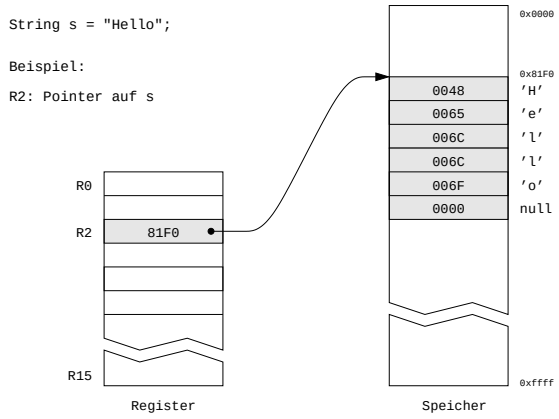


Abbildung 1: Null-terminierter String im Speicher, ein Zeichen pro Wort

Aufgabe 2.2: 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 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 der Speicher. Setzen Sie Sie diese Anweisungen bitte immer ganz an das Ende ihres Programms!

Einen Rumpf finden Sie in der Datei *m_strlen.asm*, die Sie sich in den Assembler *winT3asm.exe* laden können. Auf dem Desktop sollte es ein entsprechendes Icon geben.

Aufgabe 2.3: strcpy() Erstellen Sie ein Assemblerunterprogramm zusammen mit einem aufrufenden Hauptprogramm für die Funktion `strcpy(char* stri1, char* stri2)`, das eine Zeichenkette `stri2` (inklusive des terminierenden Nullbytes) in einen vorgebenen Speicherbereich `stri1` kopiert. Die Startadresse des Strings `stri2` stehe dabei in `R10` und die Adresse von `stri1`, in den kopiert werden soll, in `R11`.

Testen Sie Ihr Programm, indem Sie von Ihrem Hauptprogramm durch zwei Aufrufe des Unterprogramms einen String erst auf die Adresse *0x8100* und dann an die Adresse *0x8200* kopieren.

Aufgabe 2.4: strcat() Erstellen Sie ein Assemblerunterprogramm zusammen mit einem aufrufenden Hauptprogramm für die Funktion `strcat(char* stri1, char* stri2)`, das eine Zeichenkette `stri2` hinten an den String `stri1` anfügt. Die Startadresse des Strings `stri1` stehe in `R10` und die Adresse des Strings `stri1`, an den `stri2` angefügt werden soll, in `R11`.

Testen Sie Ihr Programm mit geeigneten Strings. Könnte man mit Ihrer Funktion auch einen String an sich selbst anfügen, also z.B. aus der Zeichenkette 123 den neuen String 123123 machen?

Aufgabe gelöst: .....

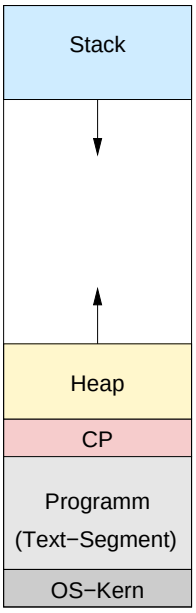
3 Speicherbereiche und Stack

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 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.



0xffff

```
long fak( n ) {      // Programmtext: Programm
  long n;            // lokale Variable: Stack

  if (n == 1) return 1;
  else        return n*fak(n-1);
}


static int i, j, k;    // globale Variable: CP

int main( int argc; char ** argv ) {
  int* array = malloc( 10000L );// malloc: Heap

  for( i = 0; i < 100; i++ ) {
   printf( "Fakultät(%ld)= %ld", i, fak(i));
  }
}
```

0x0000

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

Aufgabe 3.1: Stack Machen Sie sich aus den Vorlesungsskripten die Funktion eines Stacks klar. Was könnten in diesem Zusammenhang folgende Begriffe bedeuten, wenn es um Informationen (z.B. Registerinhalte) geht, die noch benötigt, bzw. überschrieben werden:
caller save:
callee save:

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

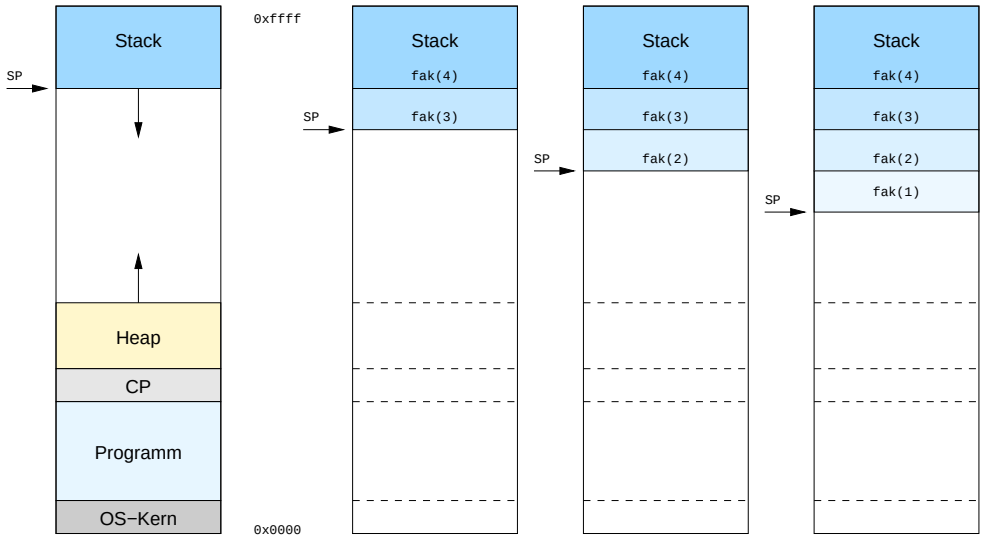


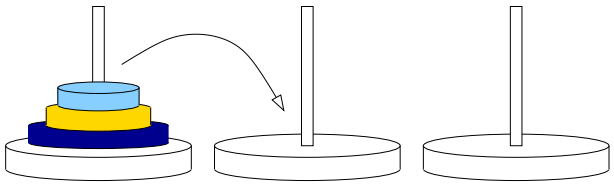
Abbildung 3: Rekursiver Aufruf der Faktultätsfunktion.

Aufgabe 3.2: 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?

Aufgabe 3.3: pop() Notieren Sie die notwendigen Assemblerbefehle, um den Inhalt der Register `R4`, `R5`, `R10` vom Stack wiederherzustellen:

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. Vergessen Sie bitte nicht, ihren Stackpointer auf einen definierten Wert (z.B. 0) zu initialisieren.

Aufgabe 3.4: Rekursive Unterprogramme – 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 (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 dies wirklich nur Pseudoanweisungen sind, die anders als z.B. .push und .pop **nicht** auf reale Befehle des DCOR-Prozessors abgebildet werden, auch wenn sie einen Opcode in ROM oder RAM schreiben. Dieser dient nur als Anweisung an den Emulator, eine entsprechende Ausgabe durchzuführen.

Aufgabe 3.5: 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.
Wieviele Aufrufe ergeben sich für `towers(8,1,3)`? Was folgt daraus für die Komplexität des Algorithmus?

Aufgabe gelöst: .....