Technische Informatik I
Unterprogramme und Adressierung

Thorsten Thormählen

28. Januar 2020

Teil 10, Kapitel 2

Dies ist die Druck-Ansicht.

Aktiviere Präsentationsansicht

Steuerungstasten

→ nächste Folie (auch Enter oder Spacebar).
← vorherige Folie
d schaltet das Zeichnen auf Folien ein/aus
p wechselt zwischen Druck- und Präsentationsansicht
CTRL + vergrößert die Folien
CTRL - verkleinert die Folien
CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$\mathrm{f}, \mathrm{g}(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$, $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathtt{A}, \mathtt{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, B = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Unterprogramme in Maschinensprache

Stapelspeicher (Stack)

Speicheradressierung

Operationen auf Speicherblöcken

Unterprogramme in Maschinensprache

Um ein Unterprogrammen (auch "Subroutine", "Prozedur" oder "Funktion" genannt) in Assembler aufzurufen, wird der Befehl `call` verwendet

```
call ziel
```

Das Ziel bezeichnet dabei die Adresse (bzw. ein Label im Assembler-Code), an der sich das Unterprogramm im Speicher befindet

Der Befehl `ret` (engl. "return") steht am Ende des Unterprogramms und sorgt dafür, dass zum aufrufenden Programm zurückgekehrt und dieses weiter abgearbeitet wird

Um diese Sprünge auszuführen, modifizieren die Befehle `call` und `ret` den Befehlszähler `EIP`

Unterprogramme in Maschinensprache

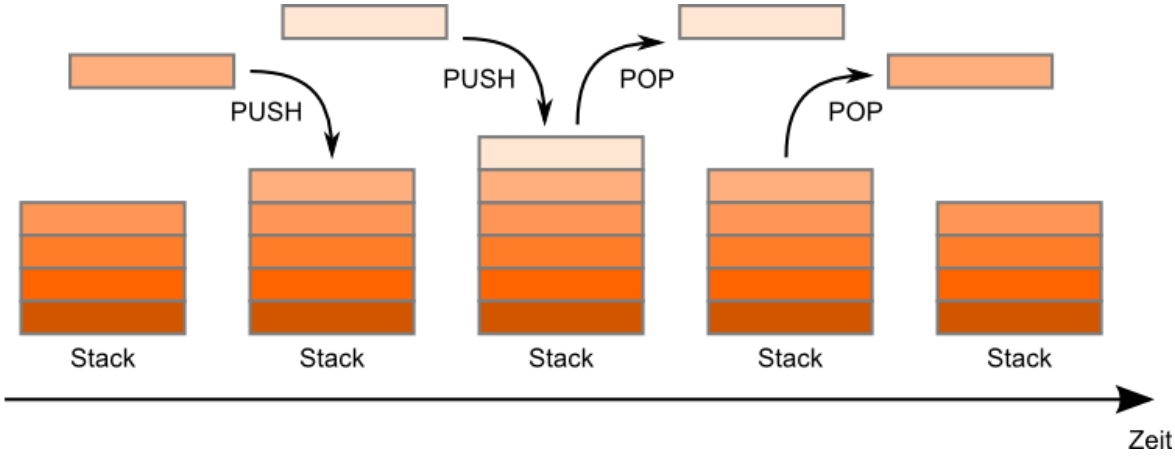
Beispiel:

```
int main() {
    int counter = 0;
    __asm {
        call Subroutine_IncrementCounter   ; // counter = 1
        call Subroutine_IncrementCounter   ; // counter = 2
        call Subroutine_IncrementCounter   ; // counter = 3
        jmp EndMainProgram

      Subroutine_IncrementCounter:
        mov eax, counter
        inc eax
        mov counter, eax
        ret

      EndMainProgram :
    }
}
```

Quelldatei: main.cpp

Um zu verstehen, was bei dem Aufruf eines Unterprogramms genau passiert, wird zunächst erläutert, was unter einem "Stack" verstanden wird

Stapelspeicher (Stack)



Auf einen Stapelspeicher (engl. "Stack") können Elemente abgelegt werden und später wieder entnommen werden

Elemente werden übereinander gestapelt und können nur in umgekehrter Reihenfolge wieder entnommen werden ("Last-In-First-Out")

Dabei gibt es nur zwei Operationen:

- PUSH legt ein Element auf den Stapel
- POP entnimmt das oberste Element vom Stapel

Die Assemblerbefehle PUSH und POP

Der Assembler-Befehl `push` legt den Inhalt von `quelle` auf den Stapel

```
push quelle
```

Der Assembler-Befehl `pop` entnimmt das oberste Element vom Stapel und kopiert es in `ziel`

```
pop ziel
```

Beispiel: Wiederherstellen von Registerinhalten

```
Subroutine_IncrementCounter:
  push ebx      // push content of ebx to stack
  mov ebx, counter
  inc ebx
  mov counter, ebx
  pop ebx       // pop content from stack to ebx
  ret
```

Quelldatei: main.cpp

Stapelspeicher und die Register BP und SP

Jedes Programm hat einen Stapelspeicher, der im RAM abgelegt ist

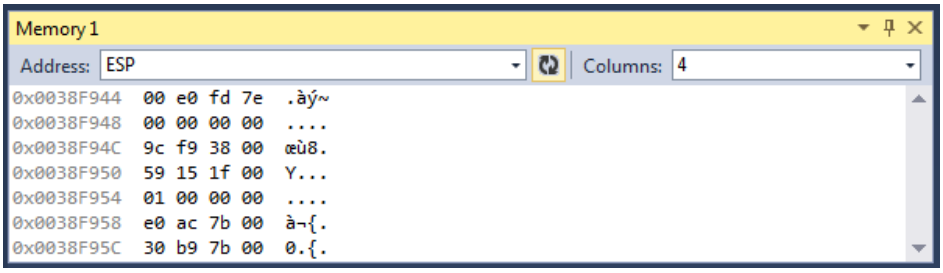
An welcher Speicheradresse sich der Stapelspeicher befindet, ist in den Registern BP und SP angegeben (bzw. in den entsprechenden 32-Bit Versionen: EBP und ESP):

BP: Anfangsadresse des Stapelspeichers

SP: Adresse des obersten Elements des Stapelspeichers

Der Stack wächst in Richtung kleinerer Speicheradressen

In MS Visual Studio kann der Stapelspeicher leicht angezeigt werden:
Menü→ Debuggen→ Fenster→ Arbeitsspeicher
und dort als Adresse "ESP" eintragen



Der Stapelspeicher bei Unterprogrammaufrufen

Bei der Ausführung von Unterprogrammen spielt der Stapelspeicher (engl. "Stack") eine wichtige Rolle

Der Stack wird verwendet, um sich die Rücksprungadresse zu merken, an die der Ret-Befehl des Unterprogramms zurückkehren soll

Dazu legt ein Call-Befehl per PUSH die Adresse des ihm nachfolgenden Befehls (aktueller Befehlszähler + 1 Befehl) auf den Stack und führt dann durch Setzen des Befehlszählers `EIP` den Sprung in das Unterprogramm aus

Um aus dem Unterprogramm zurückzukehren, entnimmt der nächste Ret-Befehl per POP die Rücksprungadresse und lädt diese in den Befehlszähler

Der Stapelspeicher bei Unterprogrammaufrufen

Um dies zu verdeutlichen, wird nochmal das vorherige Beispiel betrachtet (zu Beginn jeder Zeile steht die Speicheradresse des Befehls):

```
00881033h   call Subroutine_IncrementCounter;// push 00881038h; set EIP=088103Ah
00881038h   jmp EndMainProgram              ;// set EIP=00881044h
          Subroutine_IncrementCounter:
0088103Ah   push ebx       ;// push content of ebx to stack
0088103Bh   mov ebx, counter
0088103Eh   inc ebx
0088103Fh   mov counter, ebx
00881042h   pop ebx      ;// pop content of ebx from stack
00881043h   ret          ;// pop 00881038h; set EIP=00881038h
          EndMainProgram:
00881044h
```

Quelldatei: main.cpp

Was würde passieren, wenn der `pop` Befehl an Adresse `00881042h` auskommentiert würde

Antwort: der Befehlszähler `EIP` würde mit dem Inhalt von `EBX` geladen, d.h. das Programm würde an einer unsinnigen Stelle im Speicher weiter ausgeführt. Dies resultiert bei Windows typischerweise in einem Speicherzugriffsfehler (engl. "Access violation")

Der Stapelspeicher bei Unterprogrammaufrufen

Durch Verwendung des Stacks sind auch verschachtelte oder (wie hier gezeigt) rekursive Unterprogrammaufrufe möglich (Quelldatei: main.cpp):

```
int main() {
    int counterForward = 0;
    int counterBackward = 0;
    __asm {
        call Subroutine_IncrementCounter   ;
        jmp EndMainProgram
      Subroutine_IncrementCounter:
        mov eax, counterForward
        inc eax
        mov counterForward, eax
        cmp eax, 5
        jz Subroutine_IncrementCounterPart2
        call Subroutine_IncrementCounter  ; // recursive call of subroutine
      Subroutine_IncrementCounterPart2:
        mov eax, counterBackward
        inc eax
        mov counterBackward, eax
        ret
    EndMainProgram :
    }
}
```

Sichern von Registerinhalten auf dem Stapelspeicher

Bei den Allzweck-Registern gibt es zwei Gruppen:

Caller-Saved-Register: `EAX`, `ECX` und `EDX`

Callee-Saved-Register: `EBX`, `ESI` und `EDI` (sowie `EBP` und `ESP`)

Bei Unterprogrammaufrufen ist es eine Konvention, dass die Caller-Saved-Register in einem Unterprogramm verändert werden können, während die Inhalte von Callee-Saved-Registern erhalten bleiben

Möchte ein Unterprogramm (engl. "Callee") trotzdem Callee-Saved-Register verwenden, muss es die Inhalten vor Gebrauch mit PUSH auf den Stack sichern und nach Gebrauch mit POP wiederherstellen

Möchte ein aufrufendes Programm (engl. "Caller") den Inhalt von Caller-Saved-Registern nach dem Aufruf eines Unterprogramms weiterverwenden, muss es die Inhalten vor dem Call-Befehl mit PUSH auf den Stack sichern und nach Gebrauch mit POP wiederherstellen

Parameterrückgabe mittels Registern

Subroutinen in Hochsprachen haben in der Regel einen Rückgabewert, z.B. in C/C++:

```
int add(int varA, int varB ) {
  int retVal = varA + varB;
  return retVal;
}
```

Bei der Umsetzung in Maschinensprache gilt:

- Ein 32-bit Rückgabewert wird im EAX-Register an das aufrufende Programm übergeben
- Ein 64-bit Rückgabewert wird in den Registern EDX:EAX an das aufrufende Programm übergeben

Parameterübergabe mittels Stapelspeicher

Äquivalent zur Parameterrückgabe könnte auch bei der Parameterübergabe an Unterprogramme bestimmte Register verwendet werden

Dies hat jedoch den Nachteil, dass nur eine relativ begrenzte Anzahl an Parametern übergeben werden können

Stattdessen werden die Parameter von dem aufrufenden Programm auf den Stack gelegt und von dem Unterprogramm dort ausgelesen

Bei der konkreten Umsetzung in Maschinensprache gibt es viele Möglichkeiten dies zu tun. Im Folgenden wird die so genannte "C Declaration"-Konvention (kurz "cdecl") näher beschrieben

Es gibt aber auch andere Konventionen, z.B. "stdcall" oder "fastcall".

Welche Aufrufkonvention verwendet wird, kann bei Compilern typischerweise eingestellt werden. In MS Visual Studio unter:
Menü→ Projekt→ Eigenschaften→ Konfigurationseigenschaften → C/C++→ Erweitert → Aufrufkonventionen

Parameterübergabe mittels Stapelspeicher

Bei der Aufrufkonvention "cdecl" werden die Parameter von rechts nach links auf den Stack gelegt

Beispiel:

```
int add(int varA, int varB ) {
  return varA + varB;
}

int main() {
  int result = add(1, 2);
  return 0;
}
```

Quelldatei: main.cpp

Das heisst in diesem Beispiel würde in `main()` zunächst ein `push 2` und dann ein `push 1` ausgeführt

Wir wollen nun ausprobieren, ob sich der MS Visual Studio C/C++-Compiler an diese Konvention hält

Dazu ist auf der nächsten Folie der disassemblierte Maschinencode dieses Beispiels gezeigt

Parameterübergabe mittels Stapelspeicher

```
--- main.cpp ---------
     1: int add(int varA, int varB ) {
009C1020 55               push        ebp  // push stack base pointer
009C1021 8B EC            mov         ebp,esp // setup new stack frame
     2:   return varA + varB;
009C1023 8B 45 08         mov         eax,[ebp+08h] // access parameters
009C1026 03 45 0C         add         eax,[ebp+0Ch]
     3: }
009C1029 5D               pop         ebp  // pop stack base pointer
009C102A C3               ret
--- keine quelldatei ---
009C102B CC               int         3
009C102C CC               int         3
009C102D CC               int         3
009C102E CC               int         3
009C102F CC               int         3
--- main.cpp ---------
     4:
     5: int main() {
009C1030 55               push        ebp
009C1031 8B EC            mov         ebp,esp
009C1033 51               push        ecx
     6:   int result = add(1, 2);
```

Stapelsegmente

Das aufrufende Programm legt demnach die Übergabeparameter von rechts nacht links per `push` auf den Stapel

Es ist interessant zu sehen, dass das Unterprogramm die Parameter aber nicht per `pop` vom Stapel holt

Das wäre auch nicht so einfach, da bekanntlich der Call-Befehl die Rücksprungadresse auf den Stack legt und diese somit den direkten Zugriff auf die Parameter per `pop` blockiert

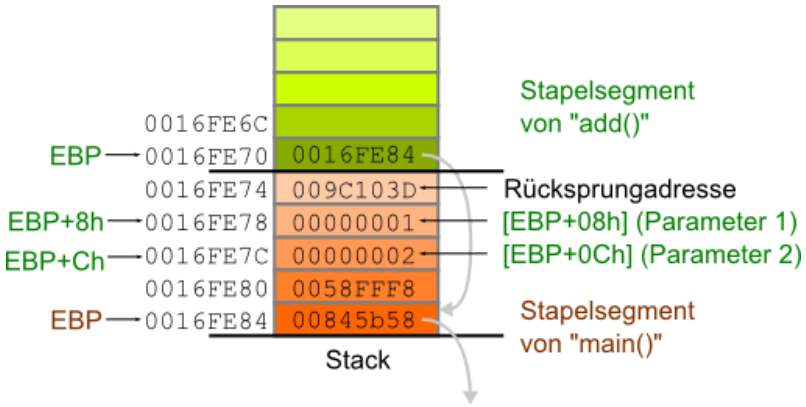
Stattdessen spielt scheinbar das `EBP`-Register ein Rolle:

- Der Stapelspeicher wird typischerweise in Segmente unterteilt. Jedes Unterprogramm hat sein eigenes Segment
- Das `EBP`-Register zeigt immer auf die Anfangsadresse des aktuellen Stapelsegments
- Das Unterprogramm verwendet einen feste relative Adresse zum seinem `EBP` um auf die Übergabeparameter zuzugreifen

Stapelsegmente

Im Unterprogramm wird daher wie folgt vorgegangen:

```
     1: int add(int varA, int varB ) {
009C1020 push ebp     // legt die alte Segmentanfangsadresse auf den Stack
009C1021 mov  ebp,esp // die aktuelle Stackadresse wird neue Segmentanfangsadresse
     2:   return varA + varB;
009C1023 mov eax,[ebp+08h] // greift auf den ersten Parameter zu
009C1026 add eax,[ebp+0Ch] // greift auf den zweiten Parameter zu
     3: }
009C1029 pop ebp // stellt die alte Segmentanfangsadresse wieder her
009C102A ret     // entfernt die Rücksprungadresse vom Stack
```



Parameterübergabe mittels Stapelspeicher

Bei der Aufrufkonvention "cdecl" ist es die Aufgabe des aufrufenden Programms den Stack nach eine Parameterübergabe wieder aufzuräumen (bei "stdcall" Aufgabe des Unterprogramms)

Dies könnte in unserem Beispiel durch zwei `pop` Befehle passieren

Eine Alternative, die scheinbar der C/C++ Kompiler von Visual Studio im gezeigten Beispiel bevorzugt, ist die Addition von `ESP` mit 8 (8 Byte, weil 4 Byte pro 32-Bit Übergabeparameter):

```
add  esp,8
```

Mit der Verständnis der Aufrufkonvention ist es damit möglich, das erzeugte Maschinenspracheprogramm des Kompilers von Visual Studio für das gezeigt Beispiel komplett nachzuvollziehen

Es wird dadurch auch offensichtlich, dass nur solche Programmteile sich gegenseitig aufrufen können, wenn sie die gleichen Aufrufkonvention verwenden

Direkte Speicheradressierung

Früher, z.B. unter DOS, war es kein Problem, direkt auf eine bestimmte Speicherstelle im Hauptspeicher lesend oder schreibend zuzugreifen

Eine in eckigen Klammern stehende Zahl wird als Adresse interpretiert

```
mov eax, [110h]
```

Dabei war 110h nur eine Offset-Adresse. Die komplette Adresse wurde bei Datenelementen aus dem Inhalt des `DS` (Daten-Segment) Segmentregister plus dieser Offset-Adresse gebildet, d.h. der oben angegebene Befehl war gleichbedeutend mit

```
mov eax, ds:[110h]
```

Direkte Speicheradressierung

Bei moderneren 32-bit Betriebsystemen, wie Windows oder Linux, sind solche direkten Speicherzugriffe nicht erlaubt, da sie andere Programme oder das Betriebssystem stören könnten

Jedes Benutzerprogramm erhält seinen eigenen Adressraum, in dem es Daten schreiben und lesen kann ("protected mode")

Wie dieser Adressraum auf den physikalisch vorhandenen Speicher abgebildet wird, ist Aufgabe des Betriebssystems

Innerhalb des eigene Adressraums gibt es ein flaches Speichermodell. Ein Programmierer muss sich nicht um Segmentregister kümmern, wie `DS` (Daten-Segment), `SS` (Stapel-Segment) oder `CS` (Code-Segment)

Im Folgenden wird daher nicht mehr zwischen Offset-Adresse und Adresse unterschieden, gemeint ist immer die "virtuelle" Adresse im Adressraum des Programms

Direkte Speicheradressierung

Beim Inline-Assembler werden die Variablen typischerweise in der Hochsprache definiert und mit dem Variablennamen im Assemblercode auf die Speicheradresse zugegriffen

Dabei ist es egal, ob der Variablennamen in eckigen Klammern gesetzt ist oder nicht. Gemeint ist immer das Lesen bzw. Schreiben des Werts an der Speicheradresse, die der Variablen zugeordente ist

Wird vor den Variablenname der Bezeichner `offset` gesetzt ist die Speicheradresse selbst gemeint und nicht der gespeicherte Wert

Beispiel:

```
int myVar = 42;
int main() {
  __asm {
    mov  eax, myVar        // move the content of myVar to eax
    mov  eax, [myVar]      // move the content of myVar to eax
    mov  eax, offset myVar // move the address of myVar to eax
  }
}
```

Quelldatei: main.cpp

Speicheradressierung von Feldern

Beim der direkten Adressierung von Elementen in Feldern (engl. "Arrays") kann der Versatz (in Bytes) zur Startadresse des Felds angegeben werden

Beispiel:

```
#include <stdio.h>
int main() {
  int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
  for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output

  __asm {
    mov  eax, [studentIds+4]  // move 12341 to eax
    mov  ecx, [studentIds+12] // move 13343 to ecx
    mov  [studentIds+4], ecx  // move 13343 to studentIds[1]
    mov  [studentIds+12], eax  // move 12341 to studentIds[3]
  }

  for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output
  return 0;
}
```

Quelldatei: main.cpp

Speicheradressierung von Feldern

Dabei gibt es folgende äquivalente Notationen:

```
#include <stdio.h>
int main() {
  int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
  for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output

  __asm {
    mov  eax, studentIds+4    // move 12341 to eax
    mov  ecx, studentIds[12]  // move 13343 to ecx
    mov  studentIds[4], ecx   // move 13343 to studentIds[1]
    mov  studentIds+12, eax   // move 12341 to studentIds[3]
  }

  for(int i=0; i<5; i++) printf("studentIds[%d]: %d\n", i, studentIds[i]); // output
  return 0;
}
```

Indirekte Speicheradressierung

Bei der indirekte Speicheradressierung wird die Adresse durch ein Register übergeben und steht somit erst zur Laufzeit fest

Dabei wird der Inhalt des Registers als Adresse interpretiert

Beispiel:

```
mov eax, [ebx]
```

Steht z.B. im EBX-Register der Wert 110h, dann würde der Wert an der Speicheradresse 110h in das EAX-Register kopiert

Der Assemblerbefehl LEA

Der Assemblerbefehl `lea` (engl. "load effective address") dient dazu die Adresse einer Variablen zu ermitteln

```
lea ziel quelle
```

Die Adresse der in `quelle` angegebene Variablen, wird in `ziel` geladen

Beispiel:

```
int main()
{
  int num = 8;
   __asm {
      mov eax, 4;
      lea ecx, num     ;// ecx = address of variable "num";
      add eax, [ecx]   ;// add 4 + 8
   }
}
```

Quelldatei: main.cpp

Der Assemblerbefehl LEA

Mit Hilfe des `lea`-Befehls und bei bekannten Aufrufkonventionen der C-Funktionen, kann die `printf` C-Funktion nun auch aus dem Inline-Assembler aufgerufen werden (Quelldatei: main.cpp) :

```
#include <stdio.h>
char format[] = "%s %s %d\n";
char hello[] = "Hello";
char world[] = "world: num=";
int num = 8;
int main() {
   __asm {
      push num         ;// pushes parameter to stack
      lea  eax, world  ;// load address of array "world" to eax
      push eax         ;// pushes parameter to stack
      lea  eax, hello  ;// load address of array "world" to eax
      push eax         ;// pushes parameter to stack
      lea  eax, format ;// load address of array "format" to eax
      push eax         ;// pushes parameter to stack
      mov  eax, printf ;// get address of C-printf function
      call eax         ;// calls C-printf function
      add esp, 16      ;// clean up the stack
   }
}
```

Indirekte Speicheradressierung

Das Register `bx` bzw. `ebx` ("Base") ist dafür vorgesehen, um in Felder zu indizieren

Neben einfachen Indexangaben, wie `[studentIds+ebx]` sind auch einfache konstante Ausdrücke zugelassen wie z.B. `[studentIds+4*ebx-4]`

Als Faktoren sind nur die typischen Größen von Datentypen zulässig: 1, 2, 4 und 8

Beispiel: Alle Matrikelnummern um eins erhöhen:

```
int main() {
  int studentIds[] = {8012, 12341, 12412, 13343, 13423}; // create C-Array
  __asm {
      mov ebx, 5   ;// set loop counter
    LoopOverIds:
      mov eax, [studentIds+4*ebx-4]     ;// studentIds[4], studentIds[3], ...
      inc eax
      mov [studentIds+4*ebx-4], eax
      dec ebx;
      jnz LoopOverIds
  }
  return 0;
}
```

Quelldatei: main.cpp

Operationen auf Speicherblöcken

Zur Operation auf Zeichenketten oder Feldern bzw. ganz allgemein auf Speicherblöcken gibt es die `movs` ("Move Data from String to String") und `cmps` ("Compare String") Befehle

Diese gibt es für verschiedene Größen der Datenelemente (siehe Tabelle)

Zur Adressierung von Quelle und Ziel der Datenbewegung müssen die Register `esi` ("Extended Source Index") und `edi` ("Extended Destination Index") verwendet werden

So kopiert beispielsweise der Befehl `movsb` ein Byte von der Adresse in `esi` zur Adresse in `edi`

Befehl	Elementgröße
`movsb`	Byte (1 Byte)
`movsw`	Word (2 Bytes)
`movsd`	Double Word (4 Bytes)
`movsq`	Quad Word (8 Bytes)
`cmpsb`	Byte (1 Byte)
`cmpvsw`	Word (2 Bytes)
`cmpsd`	Double Word (4 Bytes)
`cmpsq`	Quad Word (8 Bytes)

Danach werden die Register `esi` und `edi` automatisch erhöht oder erniedigt, je nachdem ob das Direction-Flag UP gesetzt ist oder nicht

Zur Manipulation des Direction-Flags gibt es die Befehle:

- `cld` ("clear direction") für aufsteigende Abarbeitung
- `std` ("set direction") für absteigende Abarbeitung

Operationen auf Speicherblöcken

Beispiel (Quelldatei: main.cpp):

```
#include <stdio.h>
int main() {
   char myText[] = "John is the fastest kid in town";
   char myReplace[] = "Bill";

   printf("%s\n", myText);

   __asm {
      cld               ;// clear direction flag
      mov ecx, 4        ;// set loop counter
      lea eax, myReplace
      mov esi, eax      ;// set esi to address of myReplace
      lea eax, myText
      mov edi, eax;     ;// set edi to address of myText
    LoopOver:
      movsb;
      loop LoopOver
   }

  printf("%s\n", myText);
  return 0;
}
```

Operationen auf Speicherblöcken

Ein Wiederholungspräfix `rep` vor einem `movs` Befehl kann den `loop` Befehl ersetzen. Auch bei `rep` wird das `ecx` Register als Zähler verwendet (Quelldatei: main.cpp) :

```
#include <stdio.h>
int main() {

   char myText[] = "John is the fastest kid in town";
   char myReplace[] = "pet";

   printf("%s\n", myText);

   __asm {
      cld                ;// clear direction flag
      mov ecx, 3         ;// set loop counter
      lea eax, myReplace
      mov esi, eax       ;// set esi to address of myReplace
      lea eax, myText+20
      mov edi, eax;      ;// set edi to address of myText
      rep movsb;
   }

  printf("%s\n", myText);
  return 0;
}
```

Gibt es Fragen?

Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: Kontakt

Weitere Vorlesungsfolien