Seminar 4

Rechnerarchitektur

Sprungbefehle

Unbedingter Sprungbefehl – JMP

- Syntax: *JMP Sprungziel*

- Das Sprungziel ist in der Regel eine Marke (ein Label) die irgendwo im Programm erklärt ist.
- Beispiel:

```
Lese:
        .
        .
        .
        jmp Lese
```

Bedingte Sprungbefehle

- Ist die Bedingung wahr, wird der Sprung ausgeführt, ist sie falsch, wird er nicht ausgeführt.
- Die Bedingung ist im Namen des Befehls angedeutet und bezieht sich entweder direkt auf Flags oder auf einen vorausgegangenen Compare-Befehl.
- Die Namen der bedingten Sprungbefehle (JXXX) sind nach folgendem Schema zusammengesetzt:
  - J – immer erster Buchstabe, “JUMP“
  - N – evtl. zweiter Buchstabe, “NOT“, steht für die Negierung der Bedingung
  - Z,C,S,O,P – wenn Zero-/Carry-/Sign-/Overflow-/Parityflag gesetzt
  - E – Equal, gleich
  - A,B – **Above/Below**, größer/kleiner bei **vorzeichenloser** Arithmetik
  - G,L – **Greater/Less**, größer/kleiner bei Arithmetik **mit Vorzeichen**

Befehl	Jump if...	Bedingung
JB **JNAE** **JC**	below not above or equal carry	CF=1
JAE **JNB** **JNC**	above or equal not below no carry	CF=0
JBE **JNA**	below or equal not above	CF=1 oder ZF=1
JA **JNBE**	above not below or equal	CF=0 und ZF=0
JE **JZ**	is equal Is zero	ZF=1
JNE **JNZ**	not equal not zero	ZF=0
JL **JNGE**	less than not greater or equal	SF≠OF

Befehl	Jump if...	Bedingung
JGE **JNL**	greater or equal not less than	SF=OF
JLE **JNG**	less than not greater than	ZF=1 oder SF≠OF
JG **JNLE**	greater than not less than	ZF=0 und SF=OF
JP **JPE**	parity even parity	PF=1
JNP **JPO**	not parity odd parity	PF=0
JS	negative sign	SF=1
JNS	no negative sign	SF=0
JO	overflow	OF=1
JNO	no overflow	OF=0

Befehle welche die Flags setzen

- Syntax: ***CMP ziel, quelle***
  - Ist eigentlich eine Sonderform von SUB. Er arbeitet intern genau wie SUB, schreibt aber das Ergebnis nicht in den ersten Operanden (fiktive Subtraktion). Der Sinn von CMP liegt im Setzen von Flags.

- Syntax: ***TEST ziel, quelle***
  - TEST ist eine fiktive AND-Operation.

Schleifen

- Bedingte Sprungbefehle werden meistens benutzt um Verzweigungen und Schleifen zu realisieren.

```
cmp Operand1, Operand2
jxxx Wahr-Zweig    ; Bedingter Sprungbefehl
.
.     ;Falsch-Zweig, wird ausgeführt, wenn Bedingung falsch .
jmp Verzweigungsende
Wahr-Zweig:
.
.     ;Wahr-Zweig, wird ausgeführt, wenn Bedingung wahr .
Verzweigungsende:
```

Bedingte Sprungbefehle

- **Einschränkung:** Alle bedingten Sprünge können nur Ziele im Bereich von -128 Byte bis +127 Byte erreichen. Liegt ein Sprungziel weiter entfernt, wird die Assemblierung mit einer Fehlermeldung abgebrochen.
- Beispiel:

  ```
  jz ende                ;Sprungziel ende zu weit entfernt!!! Fehlermeldung beim Assemblieren
  ```

- In diesem Fall muss man eine Hilfskonstruktion mit einem unbedingten Sprungbefehl benutzen; dieser kann beliebige Entfernungen überbrücken.

Loop-Befehle

LOOP

- erniedrigt ECX und springt anschließend zu einem als Operanden angegebenen Sprungziel, falls ECX nicht 0 ist.
- Zählschleife deren Zählindex in ECX geführt wird und abwärts bis auf 0 läuft.

LOOPE / LOOPZ

- LOOPE oder LOOPZ macht den Sprung abhängig von zwei Bedingungen:
  - ECX ungleich 0
  - ZF = 1
- Nur wenn beide Bedingungen erfüllt sind, wird der Sprung ausgeführt.
- Anders gesagt wird die Schleife abgebrochen, wenn ECX = 0 oder ZF = 0 ist.

LOOPNE / LOOPNZ

- LOOPNE oder LOOPNZ macht den Sprung abhängig von zwei Bedingungen:
  - ECX ungleich 0
  - ZF = 0

Loop-Befehle

- **Bemerkung. ECX wird erstmal dekrementiert und dann mit Null verglichen!**
- **Problem:** wenn der Startwert ECX = 0 ist, dann wird ECX dekrementiert, d.h. ECX = 0FFFFFFFFh. Also die Schleife wird 2^{32} Male ausgeführt.
- **Lösung:** Man benutzt **JECXZ** um zu überprüfen ob ursprünglich ECX = 0 ist (am Anfang der Schleife).

Zeichenfolgen (strings of bytes)

- Eine Zeichenfolge ist durch folgende Eigenschaften charakterisiert:
  - **Datentyp** der Elemente
  - Die **Adresse des ersten Elementes** aus der Folge
  - Die **Länge** der Folge (die Anzahl der Elemente)
  - Die **Richtung** in welcher die Folge durchlaufen wird:
    - Von links nach rechts, d.h. kleine Speicheradresse zu großer Speicheradresse
    - Von rechts nach links, d.h. große Speicheradresse zu kleiner Speicheradresse

Zeichenfolgen (strings of bytes)

- Felder/Folgen/Sequenzen definieren:
  - **TIMES** und ein Zahlenwert mit einer Deklaration danach –die Deklaration wird genauso viele Male wie angegeben ausgeführt
  - **RESB/RESW**/… und die Anzahl der gewünschten Elemente
  - durch Aufzählungen bei der Vorbesetzung, wobei die Anzahl der aufgezählten Elemente gleichzeitig die Feldgröße festlegt (bei Texten nützlich)
- Beispiel:

```
a times 80 db 0      ;Feld aus 80 Bytes, Vorbelegung mit 0
b resb 80            ;Feld aus 80 Bytes, keine Vorbelegung
meldung DB ’Divisionsfehler!’    ;Vorbesetzung mit einer Zeichenkette
                                 ; das Feld erhält 16 Byte Speicherplatz
```

Offset

- Bei Feldern repräsentiert der Name des Feldes die Adresse des ersten Speicherplatzes.
- Der Name einer Variablen enthält eigentlich der **Offset** der Variable, d.h. **Abstand vom Segmentanfang oder relative Adresse im Datensegment.**

$ - Positionszähler (location counter)

- Der Positionszähler ist der aktuelle Offset (Abstand in Bytes vom Segmentanfang) im Datensegment bis zu der Codezeile, wo er sich befindet
- Wird benutzt um die Länge einer Folge zu bestimmen.
- Beispiel:

```
meldung DB ’Divisionsfehler!’
l EQU $ - meldung                ; l = 16 (Konstante)
```

- Bemerkung! Für Konstanten wird kein Speicherplatz allokiert.

Aufgabe

1. Gegeben sei eine Bytefolge, die kleine Buchstaben enthält. Erstellen Sie eine neue Bytefolge, die die entsprechenden Großbuchstaben enthält.

Hausaufgabe

1. Gegeben sei eine Bytefolge von vorzeichenlosen Zahlen. Bestimmen Sie das Minimum aus der Zahlenfolge.
2. Gegeben sei eine Bytefolge von vorzeichenbehafteten Zahlen. Bestimmen Sie das Maximum der Zahlen, die sich auf gerade Positionen (Indexe) in der Zahlenfolge befinden.

   Beispiel:

   s db -2, 3, 4, -5, 7, 0, -4

   Wenn wie annehmen, dass die erste Position 0 ist, dann sind die Zahlen von den geraden Positionen: -2, 4, 7, -4 , d.h. das Maximum ist 7.

Stringbefehle

- Stringbefehle zur Datenübertragung: LODS_ , STOS_ , MOVS_
- Stringbefehle zur Datenzugriff und Vergleich: SCAS_ , CMPS_

- Load String = LODS
- Store String = STOS
- Move String = MOVS
- Scan String = SCAS
- Compare String = CMPS

Stringbefehle

- Alle Stringbefehle (MOVS_, LODS_, STOS_, CMPS_, SCAS_) haben folgende Gemeinsamkeiten:
  - Die Adressierung erfolgt immer über die Registerpaare DS:ESI und|oder ES:EDI
  - Die beteiligten Indexregister EDI und/oder ESI werden nach der Ausführung des Befehls automatisch verändert. Diese Veränderung hängt von der Bitbreite des Befehls und der Stellung von DF ab:

	DF = 0	**DF = 1**
Byteoperation	+1	-1
Wortoperation	+2	-2
Doppelwortoperation	+4	-4

Zeichenfolgen (strings of bytes)

- Eine Zeichenfolge ist durch folgende Eigenschaften charakterisiert:
  - **Datentyp** der Elemente – gegeben von der letzten Buchstabe des Befehl Namens (B=Byte, W=Word, D=Doubleword)
  - Die **Adresse des ersten Elementes** aus der Folge:
    - In **DS:ESI** für den Quellestring
    - In **ES:EDI** für den Zielstring
  - Die **Anzahl der Elemente** – angegeben durch **ECX**
  - Die **Richtung** in welcher die Folge durchlaufen wird – gegeben durch **DF**:
    - DF = 0 – kleine Speicheradresse zu großer Speicheradresse
    - DF = 1 – große Speicheradresse zu kleiner Speicheradresse

LODSB, LODSW, LODSD

- **LODS_**
  - Kopiert ein Byte/Wort/Doppelwort von der Adresse <DS:ESI> nach AL/AX/EAX
  - Falls **DF=0** dann wird **ESI mit Anzahl von gelesenen Bytes inkrementiert**
  - Falls **DF=1** dann wird **ESI mit Anzahl von gelesenen Bytes dekrementiert**

- Beispiel:

```
titel DB 'Beispiele zum Assemblerskript'
...
MOV ESI, titel  ;  Source index = Quellzeiger
;  ''titel'' liegt im Datensegment, DS:ESI zeigt jetzt auf ''titel''
CLD    ; DF = 0, -> Inkrement
LODSB  ;  load string byte ; -> AL='B'
```

STOSB, STOSW, STOSD

- **STOS_**
  - Speichert das Byte/Wort/Doppelwort von AL/AX/EAX in dem Byte/Wort/Doppelwort von der Adresse <ES:EDI>
  - Falls **DF=0** dann wird **EDI mit Anzahl von geschriebenen Bytes inkrementiert**
  - Falls **DF=1** dann wird **EDI mit Anzahl von geschriebenen Bytes dekrementiert**

MOVSB, MOVSW, MOVSD

- **MOVS_**
  - Kopiert ein Byte/Wort/Doppelwort von der Adresse <DS:ESI> nach <ES:EDI>
  - Falls **DF=0** dann werden **ESI und EDI mit Anzahl von gelesenen Bytes inkrementiert**
  - Falls **DF=1** dann werden **ESI und EDI mit Anzahl von gelesenen Bytes dekrementiert**

Beispiel

```
mov ECX, string_dim      ;Anzahl Elemente in dem String
mov ESI, source_string   ;speichert den Offset des Strings in ESI
mov EDI, dest_string
CLD
Again:
     LODSB
     STOSB
LOOP Again
```

Beispiel

```
Again:
      MOVSB
LOOP Again
```

- Da LODS + STOS = MOVS, ist es äquivalent mit:

```
Again:
      LODSB
      STOSB
LOOP Again
```

Wiederholung

- ***CMP ziel, quelle***
  - Vergleicht die zwei Operanden und setzt die Flags OF, SF, ZF, AF, PF und CF
  - Ist eigentlich eine Sonderform von SUB. Er arbeitet intern genau wie SUB, schreibt aber das Ergebnis nicht in den ersten Operanden (fiktive Subtraktion)
- **PF** (Parity Flag) wird gesetzt, wenn bei der letzten Operation ein Bitmuster entstanden ist, das in den niederwertigen acht Bit aus einer geraden Anzahl von Einsen besteht (wird relativ selten benutzt, u.a. weil es nur acht Bit auswertet)
- **AF** (Auxiliary Flag) wird gesetzt, wenn bei der letzten Operation ein Übertrag von Bit 3 auf Bit 4, also ein Übertrag vom der unteren auf die obere Tetrade, entstanden ist
- **ZF** (Zero Flag) wird gesetzt, wenn das Ergebnis der letzten arithmetischen oder bitweise logischen Operation Null war
- **SF** (Sign Flag) wird gesetzt, wenn das Ergebnis der letzten Operation negativ war.
- **OF** (Overflow Flag) wird gesetzt, wenn bei der letzten Operation der vorzeichenbehaftete Wertebereich überschritten wird

Wiederholung

- Bedingte Sprungbefehle
  - **Vorzeichenlose** Arithmetik: **below** und **above**
  - **Vorzeichenbehaftete** Arithmetik: **less** und **greater**

SCASB, SCASW, SCASD

- **SCAS_**
  - CMP AL, <ES:EDI> oder CMP AX, <ES:EDI> oder CMP EAX, <ES:EDI>
  - Vergleicht das Byte/Wort/Doppelwort in AL/AX/EAX mit dem an <ES:EDI> und setzt die Flags wie bei CMP
  - Falls **DF=0** dann wird **EDI mit Anzahl von gelesenen Bytes inkrementiert**
  - Falls **DF=1** dann wird **EDI mit Anzahl von gelesenen Bytes dekrementiert**

CMPSB, CMPSW, CMPSD

- **CMPS_**
  - CMP <DS:ESI>, <ES:EDI> (Byte/Wort/Doppelwort)
  - Vergleicht das Byte/Wort/Doppelwort an <DS:ESI> mit dem an <ES:EDI> und setzt die Flags wie bei CMP
  - Falls **DF=0** dann werden **ESI und EDI mit Anzahl von gelesenen Bytes inkrementiert**
  - Falls **DF=1** dann werden **ESI und EDI mit Anzahl von gelesenen Bytes dekrementiert**

Bemerkung

- LODS, STOS, MOVS setzen keine Flags
- SCAS und CMPS setzen die Flags, da diese ein CMP ausführen

Beispiel

```
MOV EDI, string + l - 1
MOV AL, 'e'
MOV ECX, l
STD
Cont_search:
      SCASB
      JE Found
LOOP Cont_search
;...
Found:
      INC EDI   ;gehe zurück zu dem Charakter bevor man EDI geändert hat
```

- Was is der Effekt?
  - Man findet die letzte Position im String *string* des Charakters `'e'`

Wiederholungspräfixe für Stringbefehle

```
wiederholungsPräfix Stringbefehl
```

wobei `wiederholungsPräfix` folgende Werte haben kann:

- **REP** bewirkt, dass nach jeder Ausführung des nachstehenden Stringbefehls ECX dekrementiert und, falls ECX≠0, der Stringbefehl erneut ausgeführt wird.
- **REPE – Repeat While Equal |REPZ – Repeat While Zero**
  - bewirkt, dass nach jeder Ausführung des nachstehenden Stringbefehls ECX dekrementiert und, falls ECX≠0, der Stringbefehl erneut ausgeführt wird.
  - Hier existiert ein zweites Abbruchkriterium, nämlich die Ungleichheit der Operanden, d.h. Schleifen werden bei Ungleichheit der Operanden abgebrochen (wenn ZF=0)
- **REPNE (Repeat While Not Equal) | REPNZ (Repeat While Not Zero)**
  - bewirkt, dass nach jeder Ausführung des nachstehenden Stringbefehls ECX dekrementiert und, falls ECX≠0, der Stringbefehl erneut ausgeführt wird.
  - Hier existiert ein zweites Abbruchkriterium, nämlich die Gleichheit der Operanden, d.h. Schleifen werden bei Gleichheit der Operanden abgebrochen (wenn ZF=1)

Bemerkung

- Man kann **REPE, REPZ,REPN,REPNZ** nur mit den Stringbefehlen **CMPS und SCAS** benutzen**.**
- Man kann zum Beispiel REP STOS_ benutzen um einen großen Speicherblock zu initialisieren (es ist schnell).

Beispiel

```
Again:
     MOVSW
LOOP Again
```

ist äquivalent mit:

```
rep MOVSW
```

Aufgaben

Löse folgende Aufgaben mit Stringbefehle:

1. Gegeben sei eine Bytefolge, die kleine Buchstaben enthält. Erstellen Sie eine neue Bytefolge, die die entsprechenden Großbuchstaben enthält.

Aufgaben

2. Gegeben sei ein Doppelwort-String. Erstelle ein String, der in umgekehrter Reihenfolge folgende Bytes aus dem gegebenen String enthält:
   - die high Bytes aus der low Words, die zusätzlich Vielfache von 10 sind (in der vorzeichelose Arithmetik)

Beispiel:

- Gegeben: s dd 12345678h, 1A2B3C4Dh, 0FE98DC76h
- Man sollte den String d mit folgenden Werten erstellen: DCh, 3Ch
- Bem. 56h ist kein Vielfache von 10 und gehört deshalb nicht zu dem Ergebnis

Aufgaben

3. Gegeben sei ein String mit Charakter. Bestimme die Anzahl der Vokale aus dem String.

Hausaufgabe

1. Gegeben seien zwei String s1 und s2 mit derselben Länge und Datentyp. Erstelle ein String d durch Verschachteln von s1 und s2.

Bespiel: s1: 1,2,3,4,5

s2: 11,12,13,14,15

Dann ist d: 1, 11, 2, 12, 3, 13, 4, 14, 5, 15

2. Gegeben sei ein String von Quadwords. Berechne die Summe der strikt negativen low Bytes aus den high Doublewords.