Seminar 3

Rechnerarchitektur

Bitverarbeitung

- Bitweise logische Befehle:
  - AND
  - TEST
  - OR
  - XOR
  - NOT

AND – das logische UND

AND	0	1
0	**0**	**0**
1	**0**	**1**

- Syntax: *AND op1, op2*
- verknüpft zwei Operanden bitweise entsprechend dem logischen UND
- Die Operanden können 8, 16 oder 32 Bit haben und Register-, Speicher- oder Direktoperanden sein. Da das Ergebnis im ersten Operanden abgelegt wird kann dieser kein Direktoperand sein.
- Z.B. MOV AL, 0C3h ; AL = 1100 0011b
  AND AL, 66h ; AL AND 0110 0110b
  ; Ergebnis AL = 0100 0010b = 42h

TEST

- Arbeitet genau wie AND mit dem Unterschied, dass er das Ergebnis nicht in den ersten Operanden zurückschreibt.
- Setzt die Flags wie bei AND

OR – das logische ODER

OR	0	1
0	**0**	**1**
1	**1**	**1**

- Ein bitweise logisches ODER: Die Ergebnisbits sind nur dann gleich Null, wenn beide Operandenbits gleich Null sind, sonst Eins.
- Für die Operanden gilt das gleiche wie bei AND

XOR – das exclusive ODER

XOR	0	1
0	**0**	**1**
1	**1**	**0**

- Ein bitweise logisches exclusives ODER: Ein Ergebnisbit ist gleich Eins, wenn die Operandenbits ungleich sind, sonst gleich Null.
- Beispiel:

```
MOV AL, 0C3h      ;          AL = 11000011b
XOR AL, 033h      ;          XOR 00110011b
                  ; Ergebnis AL = 11110000b
```

NOT – bitweise Invertierung

NOT	
0	**1**
1	**0**

- Der NOT-Befehl invertiert alle Bits eines Operanden und daher braucht er nur einen Operanden
- Beispiel:

```
MOV AL, 0E5h      ;                AL = 11100101b
NOT AL            ; Ergebnis: AL = 00011010b = 1Ah
```

Wofür sind die bitweisen logischen Befehle nützlich?

- Der **AND**-Befehl ist nützlich um **ausgewählte Bits** eines Operanden **zu löschen** (auf Null zu setzen) und **bestimmte Bits zu isolieren**
  - z.B. AND AX, 1111 1111 1111 0111b - welches Bit wird hier gelöscht?
  - z.B. Setze Bits 0, 2 und 3 auf Null: AND a, 11110010b
- Man benutzt TEST anstatt AND wenn man den Operanden unverändert behalten will.
  - z.B. TEST AL, 01h - man kann überprüfen ob AL eine gerade oder ungerade Zahl ist (ohne den Wert aus AL zu überschreiben)
- **OR** ist geeignet, um **ausgewählte Bits** eines Operanden **gleich eins zu setzen**.
  - z.B.

```
MOV AL, 0CCh        ;   AL = 1100 1100b
OR AL, 2h           ; AL OR 0000 0010b
                    ; Ergebnis AL = 1100 1110b = CEh
```

Wofür sind die bitweisen logischen Befehle nützlich?

- Der **XOR-Befehl** kann benutzt werden um gezielt **einzelne Bits zu invertieren**.
  - z.B. XOR AX,02h invertiert Bit 1 und lässt alle anderen Bits unverändert
- **Schnelles Null-setzen**:
  - XOR AX, AX

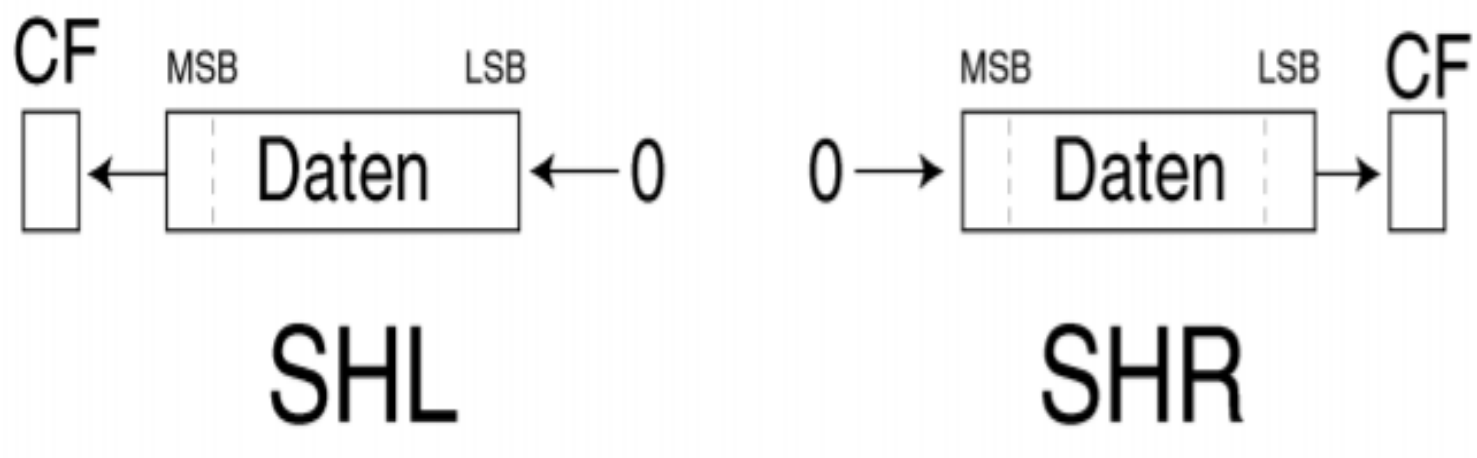
Schiebe- und Rotationsbefehle

- Erlauben es, ein komplettes Bitmuster nach links oder rechts zu schieben
- Wenn das Bit, dass an einem Ende herausfällt, am anderen Ende der Datenstruktur wieder eingesetzt wird spricht man von Rotation, sonst von Schieben (Shift)
- Das **letzte herausgefallene Bit** wird ins **Carryflag** geschrieben.
- Das bearbeitete Bitmuster kann in einem Register oder im Hauptspeicher liegen und 8, 16 oder 32 Bit umfassen.

Schiebe- und Rotationsbefehle

- Syntax: *Schiebe-/Rotationsbefehl Reg/Mem, Konstante/CL*
- Umfasst immer zwei Operanden:
  - das zu bearbeitende Bitmuster und
  - die Anzahl Bits die geschoben oder rotiert werden soll
- Die Bitzahl kann eine Konstante sein oder in CL stehen.
- Für folgende Beispiele seien die Anfangswerte:
  - **AL = abcdefgh**, wobei a-h Bits sind, und
  - **CF=k**

SHL (Shift Left), SHR (Shift Right)



- Einfaches Schieben nach links oder rechts, die frei werdenden Bitstellen werden mit einer Null aufgefüllt.

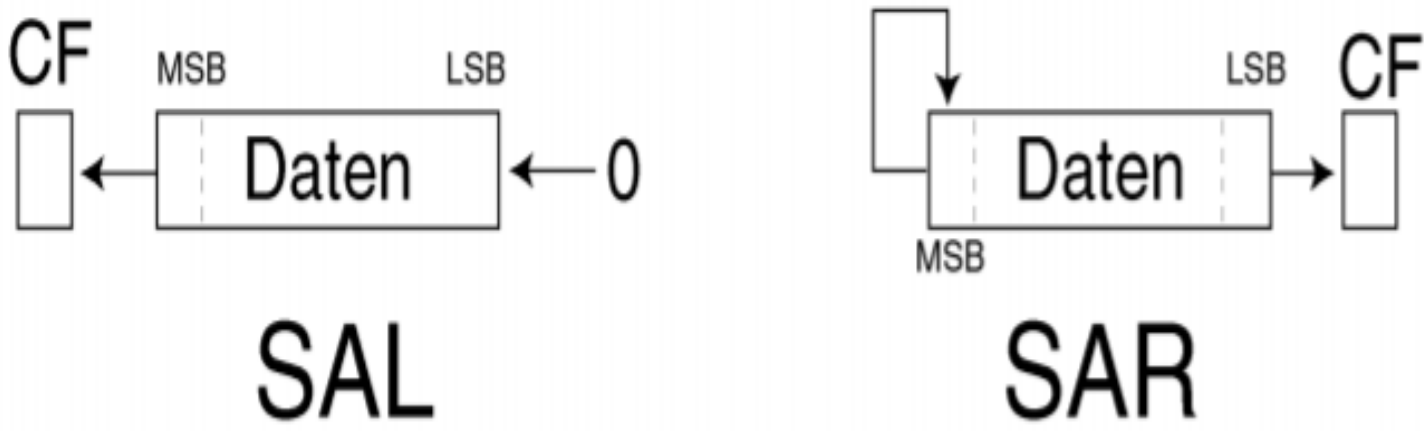
SHL (Shift Left), SHR (Shift Right)

- Beispiel: mov cl, 2
  shl al, cl ; AL=cdefgh00, CF=b

- Beispiel: mov cl, 1
  shr al, cl ; AL=0abcdefg, CF=h

- Bem. Für **vorzeichenlose** Binärzahlen entspricht das einfache Schieben um ein Bit **nach links einer Multiplikation mit zwei**, und **nach rechts** einer **Division durch zwei** (funktioniert aber nur bei vorzeichenlosen Zahlen!).

SAL (Shift Arithmetic Left), SAR (Shift Arithmetic Right)



- SAL arbeitet exakt wie SHL.
- SAR funktioniert folgendermaßen: beim Schieben nach rechts wird auf das frei werdende Bit das Most Significant Bit reproduziert.

SAL (Shift Arithmetic Left), SAR (Shift Arithmetic Right)

- Beispiel: mov cl, 2

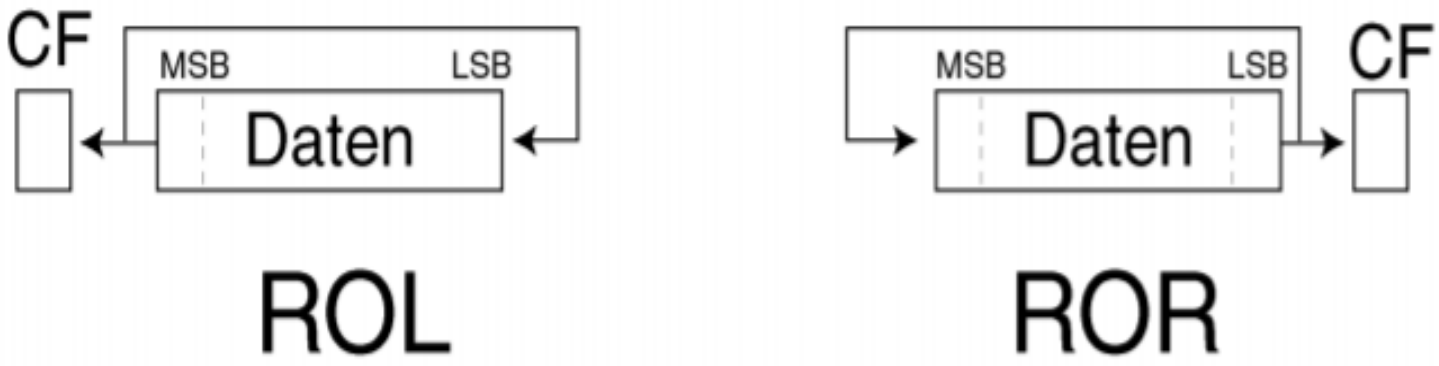
  sar al, cl ; AL=aaabcdef, CF=g

- Bem. Leisten Division durch zwei oder Multiplikation mit zwei auch bei **vorzeichenbehafteten Zahlen**!
- Beispiele:

```
MOV AL, -1   ; AL = 1111 1111b = -1
SAL AL, 1    ; AL = 1111 1110b = -2
SAL AL, 1    ; AL = 1111 1100b = -4
```

```
MOV AL, -16   ; AL = 1111 0000b = -16
SAR AL, 1     ; AL = 1111 1000b = -8
SAR AL, 1     ; AL = 1111 1100b = -4
```

ROL (Rotate Left), ROR (Rotate Right)



- Einfache Rotationen nach links oder rechts, d.h. das herausgefallene Bit kommt auf die freiwerdende Bitstelle und ins Carryflag.

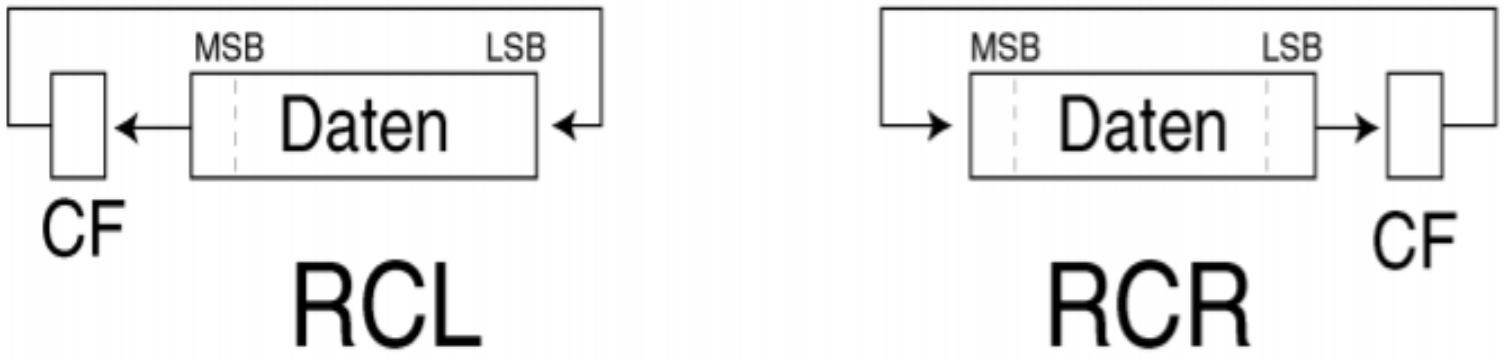
ROL (Rotate Left), ROR (Rotate Right)

- Beispiele:

```
mov cl, 2
rol al, cl   ;  AL=cdefghab, CF=b

mov cl, 2
ror al, cl   ;  AL=ghabcdef, CF=g
```

RCL (Rotate Carry Left), RCR (Rotate Carry Right)



- Ähnlich ROL und ROR, mit dem Unterschied, dass hier **das Carryflag als Bit auf die freiwerdende Stelle gelangt** und das herausgefallene Bit ins Carryflag kommt.

RCL (Rotate Carry Left), RCR (Rotate Carry Right)

- Beispiele:

```
mov cl, 2
rcl al, cl   ;  AL=cdefghka, CF=b

mov cl, 2
rcr al, cl   ;  AL=hkabcdef, CF=g
```

STC (Set Carryflag), CLC (Clear Carryflag)

- Um den Wert des Carryflags zu bestimmen benutz man die zwei Befehle: STC und CLC.
- STC setzt den Wert CF = 1
- CLC setzt den Wert CF = 0.

Aufgabe

Gegeben seien die Wörter A und B. Berechne das Wort C, wobei:

- **Die Bits 0-4 aus C sind gleich mit den Bits 5-9 aus A**
- **Die Bits 5-8 aus C sind gleich mit den Bits 2-5 aus B**
- **Die Bits 9-10 aus C haben den Wert 1**
- **Die Bits 11-12 aus C haben den Wert 0**
- **Die Bits 13-15 aus C sind komplementär zu den Bits 1-3 aus A**

Hausaufgabe

- **Gegeben sei das Wort A. Berechne die ganze Zahl *n* dargestellt auf den Bits 11-14 aus A. Dann erhalte das Wort B indem man A mit *n* Positionen nach rechts rotiert.**

Hinweise

- Man muss **Sequenzen von Bits isolieren** (unverändert behalten, wobei alle anderen Bits Null sind). Dafür benutzt man **AND mit einer Maske** von 1 auf den gewünschten Positionen.
- Die isolierten Bits verschiebt man auf den richtigen Positionen mit Rotationsoperationen.
- Man bildet das Endergebnis mit einer **OR-Operation zwischen den Zwischenergebnisse**.

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 ja 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:
  - CX ungleich 0
  - ZF = 1
- Nur wenn beide Bedingungen erfüllt sind, wird der Sprung ausgeführt.
- Anders gesagt wird die Schleife abgebrochen, wenn CX = 0 oder ZF = 0 ist.

LOOPNE / LOOPNZ

- LOOPNE oder LOOPNZ macht den Sprung abhängig von zwei Bedingungen:
  - CX 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.

Hinweise:

- Wir implementieren eine Schleife mithilfe z.B. von ESI als Index. Bei jedem Schritt wird s1[ESI] bearbeitet und in s2[ESI] gespeichert.

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.