Computertechnik

CT1

Basierend auf dem Buch

"Technische Informatik 1 Grundlagen

der Informatik und Assemblerprogrammierung"

Zusammengefasst durch Simon Flüeli

Autor	Simon Flüeli
E-Mail	fluelsim@students.zhaw.ch
Datum	07.11.2011
Fach	Computertechnik (CT1)

Inhaltsverzeichnis

1 Zahlensysteme

1.1 Additive Zahlensysteme

- Ziffern werden unabhängig von ihrer Position gleich gewichtet und einfach zusammengezählt
  - Striche beim jassen, Römische Zahlen

1.2 Stellenwertsysteme

- Ziffern haben aufgrund ihrer Position innerhalb der Zahl unterschiedliche Stellenwerte
  - Dezimalsystem im Alltag, Binärsystem in Rechnern

1.2.1 Eigenschaften

- Berechnung: Zahlenwert = $Z_n * b^n + \dots + Zi * b^i + \dots + Z_0 * b^0$
  - Basis = b
    a) Es werden b Ziffern benötigt Z_0, Z_1, … Z_{b-1}
    b) Stellennummer n bestimmt Stellenwerte. n beginnt links vom Punkt mit 0, nach links ansteigend 1, 2, 3 und nach rechts abnehmend -1, -2, -3
    c) Zwei benachbarte Stellenwerte unterscheiden sich um Faktor b
    d) Zahlenwert: Jede Ziffer mit ihrem Stellenwert multiplizieren und die Produkte addieren
    e) Um einen Stellenwert "zwischendrin" auszuschalten, braucht es eine Ziffer Null

1.2.2 Das Hexadezimalsystem

- 0 - 15

dez	**hex**	**binär**
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

1.3 Zahlenumwandlungen

1.3.1 Binär → Dezimal

Binär 101011 -> Dezimal 43

1.3.2 Dezimal → Binär

Dezimal 43 = 101011 Binär

43	:	2	=	21	Rest	1
21	:	2	=	10	Rest	1
10	:	2	=	5	Rest	0
5	:	2	=	2	Rest	1
2	:	2	=	1	Rest	0
1	:	2	=	0	Rest	1

Dezimal: 0.6875 = 0.1011 Binär

0.6875	*	2	=	1.375
0.375	*	2	=	0.75
0.75	*	2	=	1.5
0.5	*	2	=	1.0

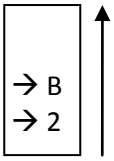
1.3.3 Hexadezimal → Dezimal

Hexadezimal 2B = 43 Dezimal
2B = (2 * 16 + 11) = 43
3FF = (3*16 + 15) * 16 + 15 = 1023

1.3.4 Dezimal → Hexadezimal

Dezimal 43 = 2B Hexadezimal

43	:	16	=	2	Rest	11	→ B
2	:	15	=	0	Rest	2	→ 2



1.4 Darstellung negativer Zahlen

1.4.1 Sign and Magnitude: SM

Im DualSystem: + = 0 - = 1

Beispiel mit 8-Bit-Wortlänge (inkl. Vorzeichen: 7-Bit-Betrag, 1-Bit-Vorzeichen)
+ 10 = 0000 1010 + 65 = 0100 0001
- 10 = 1000 1010 - 65 = 1100 0001

Darstellungsbereich: 1111 1111 (= -127) 0111 1111 (= +127)
Problem: Zwei verschiedene Darstellungen für Null
+ 0 = 0000 0000 - 0 = 1000 0000

1.4.2 Einerkomplement: EK

Das Vorzeichen einer binären Zahl wird gewechselt, indem sämtliche Bits invertiert werden
+ 10 = 0000 1010 + 65 = 0100 0001
- 10 = 1111 0101 - 65 = 1011 1110

Nachteil: auch hier zwei Nulldarstellungen + 0 = 0000 0000 - 0 = 1111 1111

1.4.3 Zweierkomplement: ZK

ZK = Einerkomplement + 1
- 10 = 1111 0110
- 65 = 1011 1111

Umwandlung von negativen Zahlen in positive erfolgt genau gleich, EK + 1
- Vorderstes Bit: Vorzeichen der Zahl, aber bei negativem Vorzeichen müssen die Nullen als Stellenwerte interpretiert werden (mit anschliessender ZK-Korrektur: "Betrag + 1+)

1.4.4 Exzessdarstellung EX

Siehe Buch Seite 22

1.5 Rechnen mit negativen Zahlen

1.5.1 Addition und Subtraktion negativer Zahlen

- Problem bei allen Rechenoperationen: Bereichsüberlauf

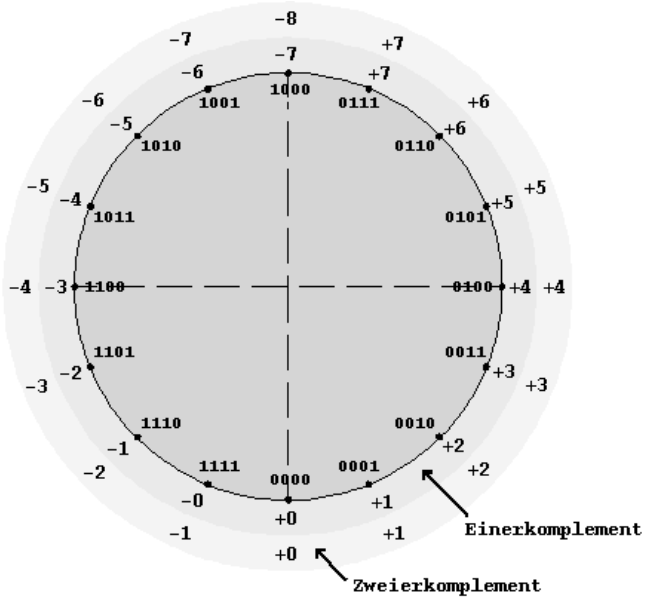


Abbildung 1: Zahlenring mit 4 Bit (besserer Seite 26)

Aus einem Carry im Resultat kann nicht auf einen Over- oder Underflow geschlossen werden. Einzig die Vorzeichen der beiden Operanden und des Resultates geben Auskunft über eine allfällige Bereichsüber- oder -unterschreitung:

- Haben beide Operanden verschiedene Vorzeichen, so kann bei der Addition kein Fehler entstehen, das Resultat ist immer richtig
- Haben beide Operanden dasselbe Vorzeichen, das Resultat jedoch das dazu entgegengesetzte Vorzeichen, so hat eine Bereichsüber- oder -unterschreitung stattgefunden
- Ein allfällig entstehendes Carry kann nicht zur Beurteilung des Resultates herangezogen werden. Es entsteht im Zusammenhang mit der Darstellung der negativen Zahlen beim Wechsel von den negativen zu den positiven Zahlen (bzw. umgekehrt als Borrow): Überschreitung des Nullpunktes beim Zahlenring

1.5.2 Multiplikation mit negativen Zahlen

Es muss eine Wortlängen-Erweiterung auf die doppelte Grösse vorgenommen werden.
Z.B. von 4-Bit (-8 bis +7)auf 8-Bit (-128 bis +127)
ACHTUNG: Bei 4-Bit-Operanden kann das Resultat nur Werte im Bereich von -56 bis +64 annehmen: Zwei mal 3 Bit für den Betrag plus Vorzeichen ergibt 6 Bit für den Betrag plus Vorzeichen: Im Prinzip würden 7 Bit genügen, ausser für den Fall -8 * -8 = +64

1. Fall

+5 * +7: 0101 * 0111
0101
01010
010100
00100011 → +35 Resultat stimmt

2. Fall

-5 * +7: 1011 * 0111
1011
10110
101100
01001101 → +77 Resultat falsch

→ Für richtiges Resultat muss der negative Operand -5 auf 8-Bit erweitert werden (Sign Extension)
Korrektur: 11111011 * 0111
11111011
111110110
1111101100
~~110~~11011101
→ 1101'1101 → -35 Resultat stimmt

Schlussfolgerung: Negative Faktoren müssen immer mit der Wortlänge des Resultates in die Rechnung genommen werden. Aus der Resultat-Wortlänge "herauslaufende" Bits dürfen nicht betrachtet werden!

2 Einführung in Mikrocomputersysteme

2.1 Was ist ein Mikrocomputer?

- Computer, deren Kernstück aus sehr wenigen spezialisierten Bauteilen besteht
- Maschine mit Steuer- und/oder Recheneigenschaften

2.2 Mikrocomputersystem

- Steuernde Teil eines technischen Prozesses
- Aufgabe: Veränderungen oder Zustände des zu steuernden Prozesses feststellen, Reaktionen ermitteln und durch Reaktionen den Prozess beeinflussen
- Besteht aus zwei Teilen: Hardware und Software

2.3 Mikrocomputer

- Steht für die Zusammenfassung eines Mikroprozessors (CPU (Central Processing Unit)), eines Speichers (Memory) und einer Ein-/ Ausgabe-Einheit (Input/Output, I/O), die über ein Bussystem miteinander verbunden sind (Siehe Bild Seite 105).
- Speicher: Aufbewahrung von Programmcode und Daten
- I/O: Dient dem Datenaustausch mit der Umgebung
- Bussystem: Interner Informationsfluss

2.4 Aufbau des Bussystems

- Adressbus, Datenbus und Steuerbus mit je einer Anzahl parallel geführten Signalleitungen
- Busbreite : Anzahl Signalleitungen
- Adressbus
  - ermöglicht Auswahl von adressierbaren Einheiten innerhalb der Speicher oder I/O-Einheit
  - Prozessor gibt die Werte auf diesem Bus vor
  - Breite des Busses definiert Grösse des direkt adressierbaren Speichers
    - Anzahl Adressen = $2^{\text{Anzahl Drähte}}$

- Datenbuss
  - überträge Daten auf den Leitungen
  - Daten können aus Prozessor in den Speicher oder I/O geschrieben werden, oder aus dem Speicher oder von I/O in den Prozessor eingelesen werden
- Steuerbus
  - Mittels Signalen werden allen angeschlossene Funktionseinheiten die Zugriffsart (lesen / schreiben), die Auswahl von entweder Speicher oder I/O sowie der zeitliche Ablauf des Zugriffs signalisiert

2.5 Speicher (Memory)

- Medien, die Informationen über gewisse Zeit aufbewahren können
- Zwei Arten von Speicher
  - periphere Speicher
    - sind über Ein-/ Ausgabe an Mikrocomputer angeschlossen
  - zentrale Speicher
    - sind an Bussystem des Mikrocomputers angeschlossen
    - Hauptspeicher
- Durch die Art und Dauerhaftigkeit der Datenspeicherung sowie durch die mögliche Geschwindigkeit des Zugriffes werden die zentralen Speicher auch als Arbeitsspeicher und die peripheren Speicher als Langzeitspeicher bezeichnet

2.5.1 Die zentralen Speicher

a) Magnetische Speicher
   Daten bleiben in einem Kernspeicher auch bei abgeschalteter Versorgungsspannung erhalten
b) Halbleiterspeicher
   Unterteilung in ROM und RAM

ROM (Read Only Memory):
- Daten können nur gelesen werden
- Gespeicherte Daten bleiben erhalten, wenn Versorgungsspannung abgeschaltet wird
- Wird für Programme und konstante Daten verwendet

RAM (Random Access Memory):
- Wahlfrei (random) Daten schreiben oder gelesen werden
- Daten gehen durch Abschaltung der Stromversorgungsspannung verloren
- Ablage von variablen Daten (z.B. Zwischenresultate)

2.5.2 Die peripheren Speicher

- Periphere Speicherelemente müssen für Schreib- oder Lesezugriff mechanisch bewegt werden
- Information ist je nach verwendetem Medium unterschiedlich gespeichert
  - mechanisch (Lochkarten, Lochstreifen)
  - magnetisch (Band, Disk, Floppy)
  - optisch (Compact Disc, Hologramm)
- häufig als serielle Speicher bezeichnet (es kann nicht zu jeder Zeit auf eine beliebige Information zugegriffen werden)

2.6 Speicherung von Daten in zentralen Speichern

- Siehe Buch Seite 109 Little Endian und Big Endian Speicherorganisation
- Little Endian
  - An der Basisadresse (niederste Adresse) steht das niederwertigste Byte des Operanden
- Big Endian
  - An der Basisadresse (niederste Adresse) steht das höchstwertigste Byte des Operanden

2.7 Ein- /Ausgabe (I/O-Ports)

- Am Eingang des Mikrocomputers: Sensoren für Informationsaufnahme benötigt
- Am Ausgang: Anzeigeelemente für die Informationsausgabe und Aktoren für die Einwirkung auf den zu steuernden Prozess verwendet
- Aktoren: Wandlung digitaler Informationen in analoge Signale, eventuell mit variablen Parametern; Galvanische Trennung

2.8 Grundsätzliche Funktion eines Computers

Funktion eines Computers anhand eines Modellcomputers Seite 110 ff.

2.9 Ablauf des Lesens und Schreibens über das Bussystem

Siehe Buch Seite 116
Aus dem Speicher lesen bedeutet: Der Prozessor gibt die Adresse, deren Inhalt gelesen werden soll, auf den Adressbus. Leitung wird mit AOK aktiviert, gefolgt von MEMR, das für einen Takt auf hohem Pegel gehalten wird. Der Wert wird bei der sinkenden Flanke von MEMR in den Prozessor eingelesen. Nach der Deaktivierung von MEMR werden die übrigen Signale wieder deaktiviert.

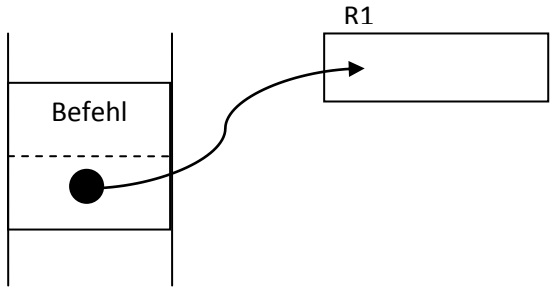
2.10 Adressierungsarten

2.10.1 Immediate-Adressierung

Bei der Immediate-Adressierung (Konstantenadressierung) ist der Operand `wert` Teil des Befehls und folgt unmittelbar auf den Befehlscode. Ein solcher Befehl würde beim Modellprozessor wie folgt aussehen:

```
SET_R1 wert
```

Der Wert des Operanden (`wert`) wird in das Register R1 eingetragen.

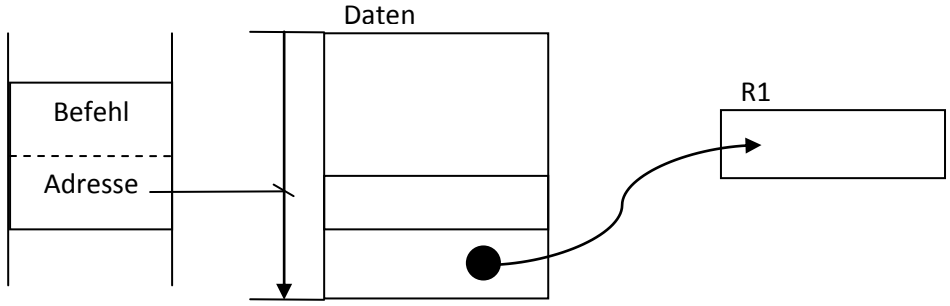


2.10.2 Direkte Adressierung

Der Operand liegt im Speicher. Die Adresse des Operanden ist im Befehl enthalten. Ein solcher Befehl würde beim Modellprozessor wie folgt aussehen:

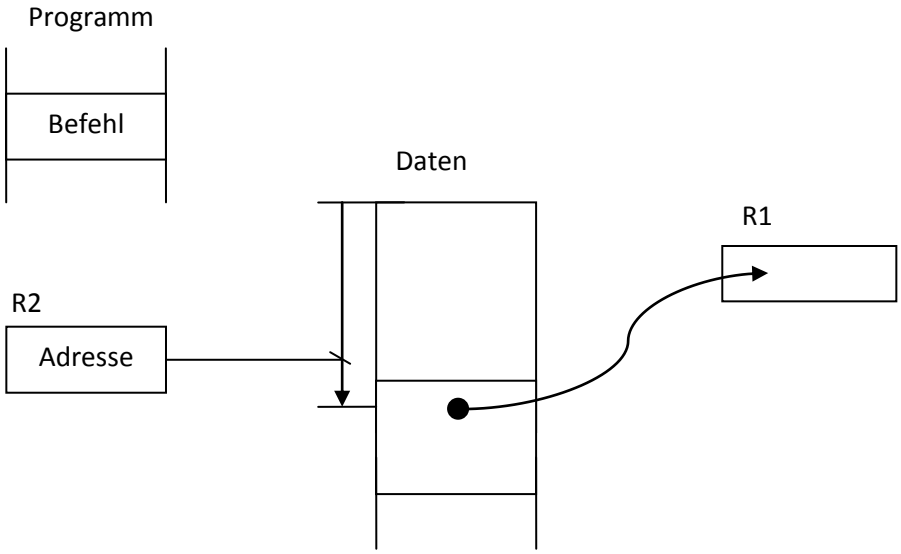
```
LOAD_R1 madr
```

Der Inhalt an der Speicherstelle `madr` wird in den Prozessor, in das Register R1, geladen.



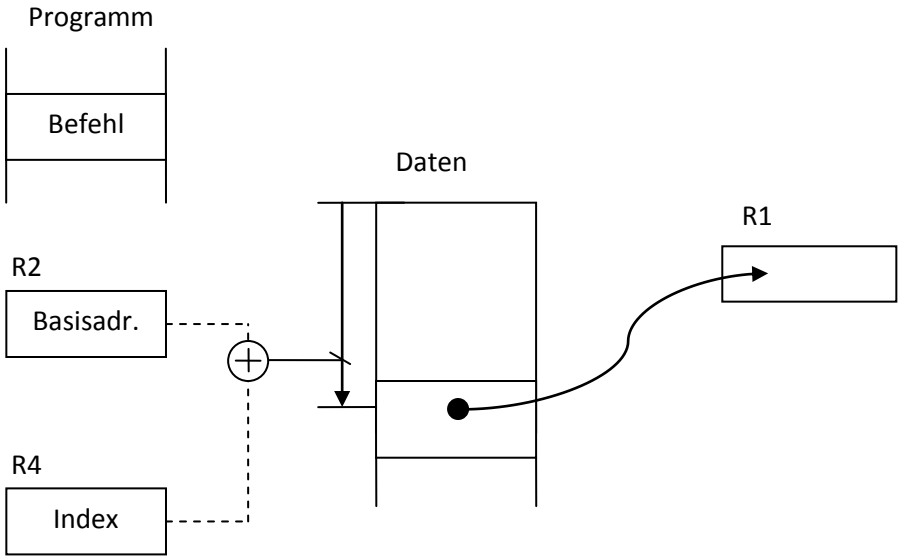
2.10.3 Indirekte Adressierung

Der Operand liegt im Speicher. Die Adresse des Operanden ist in einem Register enthalten. Ein solcher Befehl würde beim Modellprozessor wie folgt aussehen:
`LOAD_R1VR2`
Lade R1 via R2. R2 zeigt auf den Speicherplatz, dessen Inhalt in das Register R1 geladen werden soll.



2.10.4 Indexierte Adressierung

Der Operand liegt im Speicher. Die Adresse errechnet sich aus der in einem Register enthaltenen Basisadresse (z.B. einer Datenstruktur) und dem Index, der in einem zweiten Register enthalten ist. Ein solcher Befehlt würde beim Modellprozessor wie folgt aussehen:
`LOAD_R1VR2R4`
Lade R1 via R2+R4. Lade Register R1 mit dem Speicheroperanden, dessen Adresse aus der Summe der Basisadresse (in R2) und dem Index (in R4) errechnet wird.



3 Architektur des Prozessor 8086

3.1 Entwicklungsgeschichte der Intel-Prozessoren

Siehe Buch Seite 122
- 8086: 1978; Busbreite (Data: 16; Adr. 20); Vearb. intern: 16; Erster 16-Bit-Prozessor von Intel

3.2 Blockschaltbild des Prozessors 8086

- Besteht aus zwei parallel arbeitenden Einheiten
  - Bus Interface Unit (BIU, Bussteuereinheit)
    - liest Befehle auf Vorrat und reiht diese in Befehlswarteschlange (Instruction Queue) ein
  - Execution Unit (EU, Ausführungseinheit)
    - liest Befehle aus Instruction Queue und führt sie aus
- Paralleles Arbeiten erhöht Durchsatz

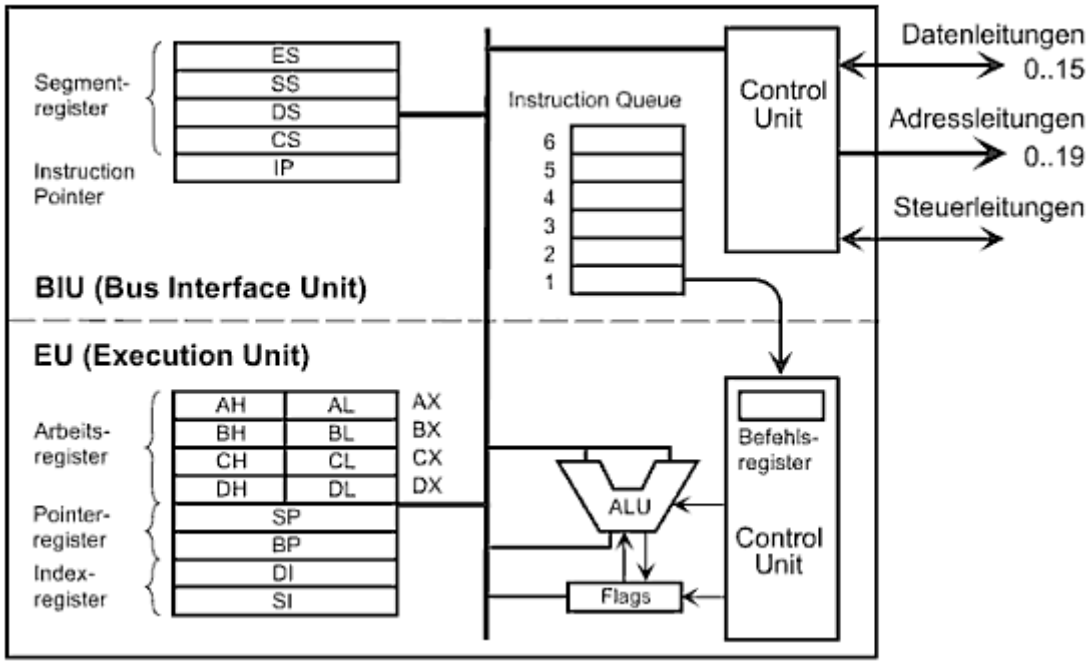


Abbildung 2: Blockschaltbild des Prozessors 8086

3.2.1 Die BIU (Bus Interface Unit)

- Segmentregister und Instruction Pointer sind hier angeordnet
- Segmentregister
  - für die Adressbildung beim Zugriff auf den Speicher
- Alle Register sind **16 Bit** breit!

3.2.2 Die EU (Execution Unit)

- Enthält die universellen 16-Bit-Arbeitsregister AX, BX, CX, DX
- Jedes Register setzt sich aus zwei 8-Bit-Registern zusammen
- Indexregister DI und SI sowie die Pointer-Register BP und SP sind nur als 16-Bit-Register ansprechbar
- Innerhalb der EU ist auch die Arithmetic and Logical Unit (ALU, arithmetische und logische Einheit)
  - Führt arithmetische und logische Instruktionen aus und hinterlegt zusätzlich zum ermittelten Resultat der Operation spezielle Hinweise über das Resultat und dessen Entstehung
  - Zusätzliche Informationen werden in Flag-Registern abgelegt

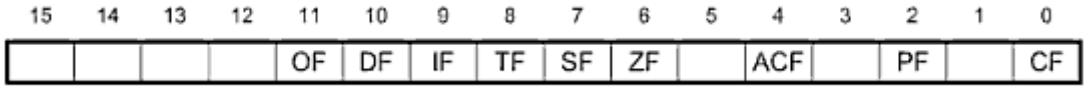


Abbildung 3: Flag-Register des Prozessors 8086

- Flag-Register: auch als Processor Status Word (PSW) bezeichnet

Arithmetische Flags	
Abkürzung	**Bezeichnung des Flag**
CF	Carry Flag
PF	Parity Flag
ACF	Auxiliary Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Steuerungs-Flags	
Abkürzung	**Bezeichnung des Flags**
TF	Trap Flag
IF	Interrupt Flag
DF	Direction Flag

3.2.3 Registersatz des Prozessors 8086

Index- und Pointer-Register

15 … 0	
DI	Destination Index[1]
SI	Source Index[1]
BP	Basepointer
SP	Stackpointer

kombinierte 8- und 16-Bit-Arbeitsregister

	15 … 0	7 … 0	
AX	AH	AL	Accumulator
BX	BH	BL	Base
CX	CH	CL	Count
DX	DH	DL	Data

Instruction Pointer und Segmentregister

15 … 0	
IP	Instruction Pointer
CS	Code Segment
DS	Data Segment
ES	Extra Segment
SS	Stack Segment

Zustand nach Reset:

IP: 0000h
CS: FFFFh
DS: 0000h
ES: 0000h
SS: 0000h

3.2.4 Die Adressbildung des 8086

Die Regel für die Erzeugung der physikalischen Speicheradresse lautet:

Segmentregister * 10h + Offset = physikalische Adresse

Siehe Buch Seite 128 für Beispiel.
- Inhalt eines Segmentregisters heisst Paragraphennummer
  - Die durch Paragraphennummer * 10h definierte physikalische Speicheradresse wird Segmentbasis genannt
- Vier Segmentregister sind vorhanden → vier Anfangsadressen können gespeichert werden →ohne weitere Veränderung des Inhalts der Segmentregister können maximal 4 verschiedene Speicherbereiche zu je 60 Kb adressiert werden (Beispiel Buch Seite 129)

3.3 Adressräume

- Der Prozessor 8086 kennt zwei Adressräume
  - Speicher-Adressraum
  - Ein-/ Ausgabe-Adressraum
    - wird mit speziellen Befehlen angesprochen
    - MEMR/MEMW und IOR/IOW wird den entsprechenden Komponenten angezeigt, ob gelesen oder geschrieben werden soll und in welchem Adressraum dies geschehen soll.
      Von diesen Signalen kann nur eines zur gleichen Zeit aktiv sein!

4 Datentransfer-Befehle

4.1 Assembler-Befehle

4.1.1 Assembler-Schreibweise und -Syntax

- Maschinenbefehle: durch CPU ausführbare Befehle
- Können nur in binärer Form (Opcode) interpretiert werden
- Assemblieren = Assembler-Code in Opcode übersetzen

Beispiel einer Schlaufe

Label	**Befehl**	**Operanden**	**Kommentar**
endless:	IN	AL, DX	; von Port DX in AL einlesen
	MOV	[BX], AL	; AL unter Adresse BX abspeichern
	INC	BX	; Adresse in BX um eins erhöhen
	JMP	endless	; springe nach "endless"

4.1.2 Symbolische Speicheradressen

- Variablen müssen in Assembler deklariert werden
  - DW = Define Word
  - DB = Define Byte

```
zaehler   DW   ?    (entspricht in C: int zaehler;)
alter     DB   ?    (entspricht in C: unsigned char alter)
```

4.2 Adressierung von Datenoperanden

4.2.1 Adressierungsarten

1. Immediate — Der Operand (Konstante) ist unmittelbar hinter dem Opcode im Codesegment abgelegt: MOV CL, **37**
2. Implizit — Der Befehl enthält implizit die fixe Operandenwahl (meist ein Register oder der Stack): Der CPU führt die Befehle PUSH und POP immer mit dem SP-Register aus (implizit von/nach Stack)
3. Register — Der Operand befindet sich in einem wählbaren CPU-Arbeitsregister: INC **CH**
4. Direkt — Die Adresse des Speicheroperanden befindet sich direkt hinter dem Opcode: MOV CX**, zaehler**
5. Register-indirekt — Der Speicheroperand wird indirekt über ein Register (oder der Kombination von zweien) adressiert: MOV **[BX+DI],** CH

Für die Speicheradressierung stehen also die **Immediate-** (unmittelbare), die **direkte** und die **indirekte** Adressierung zur Verfügung

4.2.2 Bildung der Offset-Adresse

Bei der indirekten Adressierung: verschiedene Kombinationsmöglichkeiten von bis zu drei Adressteilen vorhanden, damit auf effiziente Art und Weise strukturierte Daten (Tabellen, Records, …) adressiert werden können.
→ Operandenadresse innerhalb eines Segmentes (Offset) setzt sich aus maximal drei Teilen zusammen

$$\text{Offset} := \left\{ \begin{matrix} - \\ BX \\ BP \end{matrix} \right\} + \left\{ \begin{matrix} - \\ SI \\ DI \end{matrix} \right\} + \left\{ \begin{matrix} - \\ displ\_8 \\ displ\_16 \end{matrix} \right\}$$

Mögliche Adressanteile:

Basisregister (BX oder BP)	Das Basisregister enthält z.B. die Anfangsadresse einer Datenstruktur
Indexregister (SI oder DI)	Das Indexregister enthält z.B. einen Index (16-Bit-unsigned-Zahl: 0…65'535), der durch das Programm berechnet und verändert werden kann
Displacement (8 oder 16 Bit)	Der Programmierer kann eine konstante Zahl als vorzeichenbehaftete 8- oder 16-Bit-Konstante in die Adressberechnung einbeziehen

Beispiele

MOV	AX, zaehler	direkte Adressierung der Variablen zaehler
MOV	DX, [BX]	indirekte Adressierung via BX
MOV	AL, [BX+4]	indirekte Adressierung via BX+4
MOV	CX, [BX+SI]	indirekte Adressierung via BX+SI
MOV	ES, [BX+DI+2]	indirekte Adressierung via BX+DI+2

→ insgesamt 27 Mögliche Kombinationen ABER: nur 24 möglich!

NICHT möglich:

- Kein Adressteil vorhanden: Ohne Adressanteil ist keine Adressierung möglich
- Nur 8-Bit-Displacement: Damit könnte nur der Speicherbereit bis 255 adressiert werden
- Nur BP: Da mit dem BP normalerweise der Stack adressiert wird und dieser vor und nach der Rücksprungadresse (BP := SP) die interessierenden Werte enthält, hätte dieser Fall keine praktische Bedeutung

- 8-Bit-Displacement d8 wird vorzeichenbehaftet auf 16 Bit erweitert

4.3 Datentransfer-Befehle

- Verschieben von Daten zwischen Registern, dem Speicher oder der Peripherie
- Normalerweise: 8- oder 16-Bit-Werte (Byte / Word)

reg	Register, 8/16 Bit	mem	Speicherplatz, 8/16 Bit
reg16	Register, 16 Bit	mem16	Speicherplatz, 16 Bit
sreg	Segmentregister	const	Konstante, 8/16 Bit (aus Code-Segment)
accu	Akkumulator (Register AL / AX)	portadr	Portadresse, 8 Bit (direkte Adressierung)

Übersicht über die möglichen Datentransfers zwischen CPU, Speicher und I/O
→ Siehe Buch Seite 138

4.3.1 Move-Befehl: Kopieren von Datenwerten

- Hat Kopierfunktion (Inhalt eines Registers / Speicherplatzes wird an einen anderen Ort kopiert)
- Syntax: `MOV dst, src` move word or byte from source to destination

Wichtig:

- Bei allen Befehlen mit zwei Operanden (z.B. auch bei arithm. und logischen Befehlen) wird immer der linke Operand als Ziel-Operand und der rechte als Quelle verwendet
- Als Ziel-Operanden sind Speichervariablen oder Register wählbar (**ausser Register CS und IP**)
- Als Quellen-Operanden können Speichervariablen, Register oder Konstanten verwendet werden
- Aufgrund der Befehlscodierung kann grundsätzlich immer nur ein Speicheroperand benutzt werden: Der andere Operand muss dann ein Register oder eine Konstante sein

a) Register Adressierung

```
MOV BL, CL
MOV AX, DX
```

b) Immediate-Adressierung (Konstantenadressierung)

`MOV CL, 15`	CL ←15: Register ← Konstante (8 Bit)
`MOV BX, 1234h` BX	← 1234 hex: Register ← Konstante (16 Bit)
`MOV zaehler, 15`	Die Konstante 15 wird in der Variablen zaehler abgelegt, je nach Deklaration der Variablen als Byte oder als Word

ACHTUNG: Segmentregister DS, ES und SS können NICHT direkt mit dieser Immediate-Adressierung geladen werden - dies muss über ein anderes 16-Bit-Register erfolgen:

`MOV AX, 1234h`	Laden der Konstanten 1234h in ein Hilfsregister
`MOV DS, AX`	Nun kann DS via AX geladen werden

c) Direkte Speicheradressierung

`MOV BX, tab`	Register ← Speicher direkt
`MOV zaehler, CX`	Speicher direkt ← Register

d) Indirekte Speicheradressierung

`MOV BX, tab[SI]`	Register ← Speicher indirekt mit displ16(tab)
`MOV [BX], AL`	Speicher indirekt ← Register
`MOV CX, [BX+5]`	Register ← Speicher indirekt mit displ8 (=5)
`MOV [BX+SI], DI`	Speicher indirekt indexiert ← Register
`MOV AH, [DI]`	Register ← Speicher indirekt

Nicht möglich sind allgemeine Transfers Speicher → Speicher, da jeder Befehl nur eine universelle Speicheradresse haben kann

`MOV zaehler, [SI]`	Speicherinhalt an Adresse [SI] nach Speichervariable zaehler => FALSCH
`MOV [DI], [SI]`	Falscher Befehl, da zwei Speicheroperanden miteinander nicht möglich sind

4.3.2 Exchange-Befehl: Austauschen von Datenwerten

- Syntax: `XCHG op1, op2` exchange op1 with op2

Beispiele:

`XCHG AX, BX`	vertauschen Register <--> Register (16 Bit)
`XCHG CL, CH`	vertauschen Register <--> Register (8 Bit)
`XCHG [SI], AL`	vertauschen Speicher <--> Register (8 Bit)
`XCHG ES, DS`	vertauschen Register <--> Speicher (16 Bit)
`XCHG ES, DS`	nicht möglich → falscher Befehl!

4.3.3 Input- /Output-Befehle: Ein-/ Ausgabe von/zu Ports

- Befehle: IN und OUT
- Ein- /Ausgabe nur über den Akkumulator (Register AL bzw. AX) möglich!
- Byte-Transfer immer über AL
- Word-Transfer immer über AX
- 16-Bit-Portadresse des Peripheriegerätes muss vor IN- /OUT-Befehl in DX geladen werden
- Syntax

`IN`	`accu, DX`	8-/16-Bit-Ein-/Ausgabe via Akkumulator (AL / AX)
`OUT`	`DX, accu`	mit 16-Bit-Portadresse in DX
`IN`	`accu, port`	8-/16-Bit-Ein-/Ausgabe via Akkumulator (AL / AX)
`OUT`	`port, accu`	mit 8-Bit-Portadresse (direkte Port-Adressierung)

Beispiele für I/O Befehle Siehe Buch Seite 142

4.3.4 Ergänzungen zur Schreibweise von Speicheroperanden

- Speicheroperanden können Vorsilbe zur Bestimmung des Segmentregisters und Angaben zur Festlegung der Datengrösse (8- oder 16-Bit-Wert) haben
- Angabe der Operandengrösse mit Hilfe von `WORT PTR` und `BYTE PTR`

Beispiele

`MOV WORD PTR [BS], 12h`	Richtig: Es wird ein Word-Wert (0012h) gespeichert
`MOV DS:BYTE PTR 4711h, 0`	Richtig: Setzt das Byte im Datensegment an der Offsetadresse 4711h auf 0
`MOV [SI], 12h`	Falsch: Byte oder Word?
`MOV [SI], AL`	Richtig: AL definiert eine Byte-Operation

4.3.4.1 Die Segmentpräfixe

CS:, DS:, ES:, und SS: legen das zur Adressierung verwendete Segmentregister fest
- Normalerweise: Register DS (Datensegment), ausser wenn Adressierung das Basisregister BP enthält → dann wird standardmässig Register SS (Stacksegment) verwendet
- Kann weggelassen werden, wenn durch Verwendung eines symbolischen Variablennamens oder eines Adressregisters (BX, BP, SI oder DI) das zu verwendende Segmentregister bestimmt ist
- Bei direkten Adressierung: Segmentregister erforderlich!

Beispiele

`MOV ES:WORD PTR[BX], 1234h`	Richtig: anstelle des Segmentregisters DS wird ES verwendet
`MOV AL, DS: [BP+SI]`	Richtig: anstelle des Segmentregisters SS wird DS verwendet
`MOV BYTE PTR 11h, 0`	Falsch: direkte Adressierung erfordert Angabe des Segmentregisters
`MOV ES:BYTE PTR 11h, 0`	Richtig: direkte Adressierung mittels Segmentregister ES
`MOV BYTE PTR [SI], 0`	Richtig: SI bestimmt Segmentregister DS

5 Maschinencode der 8086-Prozessoren

5.1 Aufbau des 8086-Opcodes

- Zur Codierung der Maschinenbefehle: jeweils ein bis sechs Byte nötig

5.1.1 Prinzipieller Aufbau

1. Byte	2. Byte	3. Byte	4. Byte	5. Byte	6. Byte
opcode	addr-mode	displacement / address		immediate-data	
xxxxxxdw	mod reg r/m	addr-low	addr-high	data-low	data-high

- Erstes Byte codiert Befehl
- Im zweiten Byte werden Register oder Speicheroperanden codiert
- 3. und 4. Byte bilden den konstanten Adressanteil (Byte- oder Word-Displacement), wobei dieser auch fehlen kann
- 5. und 6. Bytes sind Immediate-Data-Anteile (Byte- oder Word-Konstante), wobei diese auch fehlen können

Vor dem Befehlscode können Befehls-Präfixe stehen:

- Segment-Override-Präfix zur Auswahl des zur Adressierung verwendeten Segmentregisters
- Repeat-Präfix zur Hardware-mässigen Wiederholung von String-Befehlen
- Lock-Präfix zur Hardware-mässigen Sperrung des Busses in Mehrrechnersystemen

5.1.2 Bedeutung der Bitgruppen

5.1.2.1 Erstes Byte "opcode"

D7	D6	D5	D4	D3	D2	D1	D0
x	x	x	x	x	x	d	w

Bedeutung

xxxxxx	(opcode)	6 Bit → 64 Befehlscodecs
d	(destination)	0 → Register reg ist Source (r/m ← reg)
		1 → Register reg ist Destination (reg ← r/m)
w	(word)	0 → Byte: 8-Bit-Operation
		1 → Word: 16-Bit-Operation

5.1.2.2 Zweites Byte "addr-mode" (address-mode)

D7	D6	D5	D4	D3	D2	D1	D0
mod			reg			r/m	

Bedeutung

reg	Die mittleren drei Bit (D5, D4, D3) codieren den Registeroperanden
mod-r/m	Die fünf Bits (D7, D6 - D2 D1 D0) codieren den Speicheroperanden (24 verschiedene Adressierungsarten gemäss den nachfolgenden Tabellen) oder den zweiten Registeroperanden (8 Möglichkeiten) bei den Register <---> Register-Befehlen (total 32 Kombinationen)

reg: Registerauswahl (Auswahl des Source- oder Destination-Registers)

reg	000	001	010	011	100	101	110	111
w = 0	AL	CL	DL	BL	AH	CH	DH	BH
w = 1	AX	CX	DX	BX	SP	BP	SI	DI

mod (adressing mode) und r/m (Register / Memory):
Bitgruppe zur Auswahl der Speicher- oder Registeradressierungsart und zur Auswahl der Register- oder Speicheradressierung

mod = 00 01 10 → r/m = Speicheroperand (direkte oder indirekte Adressierung)
mod = 11 → r/m = Registeroperand gemäss Tabelle
r/m → Register/Speicher-Adressierungsart 000 bis 111

Aufschlüsselung der 5-Bits "mod-r/m"

r/m	mod = 00	mod = 01	mod = 10	mod = 11	
				w = 0	w = 1
000	[BX+SI]	[BX+SI+displ8]	[BX+SI+displ16]	AL	AX
001	[BX+DI]	[BX+DI+displ8]	[BX+DI+ displ16]	CL	CX
010	[BP+SI]	[BP+SI+ displ8]	[BP+SI+ displ16]	DL	DX
011	[BP+DI]	[BP+DI+ displ8]	[BP+DI+ displ16]	BL	BX
100	[SI]	[SI+ displ8]	[SI+ displ16]	AH	SP
101	[DI]	[DI+ displ8]	[DI+ displ16]	CH	BP
110	direkt	[BP+ displ8]	[BP+ displ16]	DH	SI
111	[BX]	[BX+ displ8]	[BX+ displ16]	BH	DI

5.1.2.3 Default-Segmentregister und Segment-Override-Präfix

- Falls bei Operandenadressierung kein Segmentregister angegeben, verwendet CPU für Datenzugriffe mit einer Ausnahme immer das DS-Segmentregister:
-Falls in Adressierung das BP-Register vorkommt, wird das Stack-Segment via Segmentregister SS adressiert. Programmierer kann nun diese Standardzuordnung mit Hilfe von Segment-Override-Präfixen überschreiben. Es existieren vier Segment-Override-Präfixe CS:, DS:, ES: und SS:, die dem Opcode vorangestellt werden

5.2 Opcode-Aufbau der Move-Befehle

Allgemeiner Move-Befehl: Register <---> Register/Speicher
Erlaubt Transfers zwischen einem der acht gewöhnlichen Register (reg) und einem Operanden mit allgemeiner Adressierungsart (mod-r/m). Ein allgemein adressierter Operand kann ein Speicheroperand oder nochmals ein Register sein, aber **keine** Konstante

Immediate-Move-Befehl: Register/Speicher <---> Konstante (adressiert mit CS:IP)
Das Immediate-Move-Befehlsformat wird verwendet, um einen Operanden mit allgemeiner Adressierungsart (Speicheroperand oder Register, mod-r/m) mit einer Konstanten (Immediate-Adressierung) zu laden. Für die Segmentregister gibt es keinen Immediate-Move-Befehl.

Move-Befehl mit Akkumulator: AX oder AL <---> Speicher (direkte Adressierung)
- Optimiertes Befehlsformat zum Laden/Speichern des Akkumulators mit direkt adressierten Speicheroperanden
- Da Akkumulator implizit adressiert ist und Speicheroperand immer direkt adressiert ist, werden nur ein Opcode-Byte sowie das Displacement (direkte Adresse) benötigt
→ Befehl ist ein Byte kürzer als der allgemeine Move-Befehl

Move-Befehl mit Segmentregistern: DS, ES, SS oder CS <---> Register/Speicher
- Erlaubt Transfers zwischen einem Segmentregister (sreg) und einem Operanden mit allgemeiner Adressierungsart (mod-r/m), d.h. einem Speicheroperanden oder einem der acht gewöhnlichen Registern
- Nur ein Operand kann ein Segmentregister sein!

- Segmentregister CS kann nicht mit MOV-Befehl geladen werden

5.2.1 Move-Befehl mit allgemeiner Adressierung

Siehe Buch Seite 152

5.2.2 Move-Befehl mit Immediate-Adressierung

Siehe Buch Seite 152

5.2.3 Akkumulator-Move-Befehle mit direkter Adressierung

Siehe Buch Seite 153

5.2.4 Move-Befehle für Segmentregister

Siehe Buch Seite 154

5.3 Opcode-Aufbau des Exchange-Befehls

- Exchange-Befehl vertauscht die Dateninhalte von zwei Registern oder von einem Register und einem Speicherplatz (Byte oder Word)

5.3.1 Exchange-Befehl mit allgemeiner Adressierung

- Es kann immer nur einen Operanden mit allgemeiner Adressierung geben

`XCHG reg, reg`	Register <---> Register: Byte- oder Word-Exchange
`XCHG reg, mem`	Register <---> Speicher: Byte- oder Word-Exchange
`XCHG mem, reg`	Speicher <---> Register: Byte- oder Word-Exchange

Beispiele siehe Buch Seite 156

5.3.2 Exchange-Befehl für Akkumulator und 16-Bit-Register

- Für Vertauschungen der Dateninhalte des Akkumulators AX und eines 16-Bit-Registers existiert ein optimierter Exchange-Befehl

`XCHG AX, reg16`	AX <---> 16-Bit-Register: nur Word-Exchange
`XCHG reg16, AX`	AX <---> 16-Bit-Register: nur Word-Exchange

- Adressierung von AX wird als implizit bezeichnet, da für AX keine Registeradresse (000) vorhanden ist.
- Zieloperand ist implizit im Opcode 10010 enthalten

Beispiele siehe Buch Seite 157

5.4 Opcode-Aufbau der IN-/OUT-Befehle

- Zwei Arten von IN-/OUT-Befehlen
  - Üblicherweise: Portadresse als 16-Bit-Adresse im DX-Register angegeben
    → maximal 64K Ports werden indirekt adressiert
  - 8-Bit-Adresse kann auch direkt angegeben werden (aus Kompatibilitätsgründen)
- Alle IN-/OUT-Befehle können eine Datenwortbreite von 8 oder 16 Bit haben
- Datenwert ist immer im Akkumulator (von/nach AL oder AX)

5.4.1 Indirekte Portadressierung

Die Ein- und Ausgabebefehle IN und OUT können je als Byte- oder Word-Befehle mit AL bzw. AX verwendet werden.
- 16-Bit-Portadresse muss immer im Register DX sein

```
IN accu, DX
     OUT DX, accu
1   1   1   0   1   1   0   w
            Opcode
```

```
1   1   1   0   1   1   1   w
                Opcode
```

5.4.2 Direkte Portadressierung

- 8-Bit-Portadresse (port8) befindet sich direkt hinter dem Opcode

```
IN    AL, port8          8-Bit-Eingabe von 8-Bit-Port (port8)
IN    AX, port8          16-Bit-Eingabe von 8-Bit-Port (port8)
OUT   port8, AL          8-Bit-Ausgabe an 8-Bit-Port (port8)
OUT   port8, AX          16-Bit-Ausgabe an 8-Bit-Port (port8)
```

```
IN
1   1   1   0   0   1   0   w
                                  portadress
            Opcode                direkte Adresse
```

```
OUT
1   1   1   0   0   1   1   w
                                  portadress
            Opcode                direkte Adresse
```

5.4.3 Befehlsgruppen: Immed, Shift, Grp1 und Grp2

Die folgenden 14 Opcodes werden jeweils für eine ganze Gruppe von acht Befehlen verwendet, indem die drei Bits "reg" (D5…D3) des zweiten Bytes als Opcode-Erweiterung verwendet werden (maximal acht verschiedene Befehle). Die anderen fünf Bits werden normal zur Operanden-adressierung eingesetzt.

Opcode (1. Byte)	D5 D4 D3)	000	001	010	011	100	101	110	111
80 81 82 83	immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
C0 C1 D0 D1 D2 D3	shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	-	SAR
F6 F7	grp1	TEST	-	NOT	NEG	MUL	IMUL	DIV	IDIV
FE FF	grp2	INC	DEC	CALL nr, id	CALL fr, id	JMP nr, id	JMP fr, id	PUSH i	-

Beispiel: DEC AL Opcode: FEh + 11 001 000 b = FE C8h

5.4.4 Opcode der 80186-Befehle

- Sieben weitere Befehle gegenüber 8086

Opcodes	
6xh	PUSHA, POPA, BOUND, PUSH I, IMUL I, INS, OUTS
C0h, C1h	shift immediate
C8h, C9h	ENTER, LEAVE

6 Arithmetische Operationen

7 Einführung

- Zwei Typen von arithmetischen Operationen
  - Binäre Operationen
    - besitzen zwei Operanden (z.B. Subtraktion A-B)
  - Unäre Operationen
    - besitzen einen Operanden (z.B. Negation -A)
- Prozessoren liefern neben Resultat noch zusätzliche Informationen (Flags)
- Flags erlauben Auswertung des Resultats

7.1.1 Übersicht der arithmetischen Befehle

7.1.1.1 Binäre arithmetische Operationen

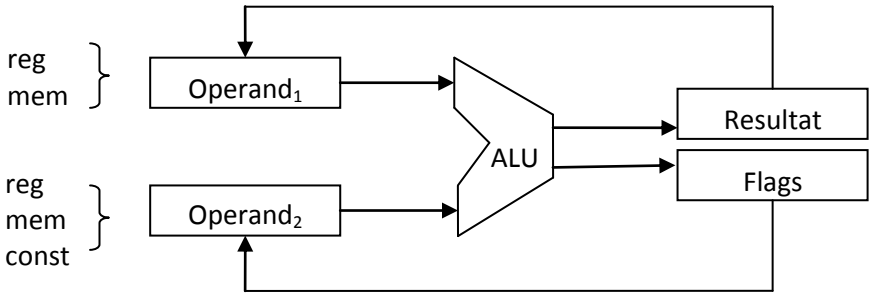
Mnemonic	Funktion
ADD	Addition zweier Operanden
ADC	Addition zweier Operanden mit Übertrag
SUB	Subtraktion zweier Operanden
SBB	Subtraktion zweier Operanden mit Borrow
MUL	Multiplikation zweier vorzeichenloser Operanden
IMUL	Multiplikation zweier vorzeichenbehafteter Operanden
DIV	Division eines vorzeichenlosen Operanden durch einen Operanden
IDIV	Division eines vorzeichenbehafteten Operanden durch einen O peranden

7.1.1.2 Unäre arithmetische Operationen

Mnemonic	Funktion
INC	Erhöhen eines Operanden um 1
DEC	Erniedrigen eines Operanden um 1
NEG	Mathematischer Vorzeichenwechsel eines Operanden
CBW	Konversion eines vorzeichenbehafteten Byte-Operanden zu einem Word-Operanden
CWD	Konversion eines vorzeichenbehafteten Word-Operanden zu einem Double-Word-Operanden

7.1.2 Datenfluss im Prozessor bei arithmetischen Befehlen

- ALU (Arithmetic and Logical Unit)
  - Verantwortlich für Ausführung der arithmetischen Operationen



Der erste Operand kann ein normales Register oder ein Speicheroperand sein. Der zweite Operand kann ein Register, ein Speicheroperand oder eine Konstante sein. Zusätzlich kann der im Carry-Flag gespeicherte Übertrag bei der Berechnung berücksichtigt werden.

Bei der Ausführung der arithmetischen Befehle liest die ALU die Operanden (sowie evtl. das Carry-Flag) und schreibt das Resultat in den ersten Operanden zurück.
→ Ursprünglicher Inhalt des ersten Operanden geht verloren
ALU beschreibt aufgrund des Resultates auch die Flags, damit im weiteren Verlauf des Programmes das Resultat ausgewertet werden kann.

7.1.2.1 Einschränkungen

- Beide Operanden müssen gleich gross sein (Byte oder word)!
- Nur einer der beiden Operanden darf ein Speicheroperand sein!

7.2 Die "arithmetischen" Flags des Prozessors 8086

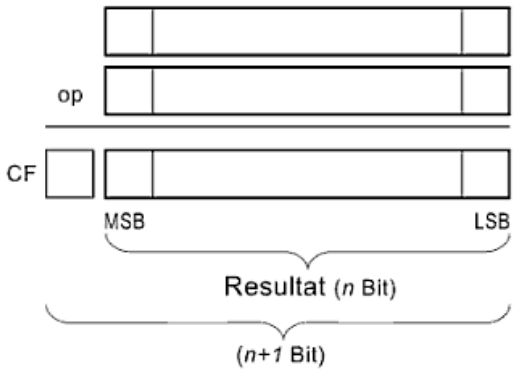
- Flags = Bits im Prozessor-Status-Wort (PSW)
  - erlauben Auswertung eines Resultates
  - werden durch arithmetische und logische Operationen und durch Schiebe- und Rotationsbefehle verändert
  - enthalten nur Aussage über Resultat der letzten Operation, welche die Flags beeinflusst hat
  - durch Transferbefehl nicht verändert!

Flag	Bedeutung
Carry (CF)	Über- oder Unterlauf bei vorzeichenlosen Operanden
Overflow (OF)	Über- oder Unterlauf bei vorzeichenbehafteten Operanden
Sign (SF)	Vorzeichen (höchstwertigstes Bit des Resultates)
Zero (ZF)	Resultat gleich null
Parity (PF)	Parität des Resultates (nur für 8-Bit-Resultate)
Auxiliary Carry (ACF)	Zur Normalisierung des Resultats im Register AL (Befehl DAA) nach einer Addition oder Subtraktion von Packed-BCD-Operanden.

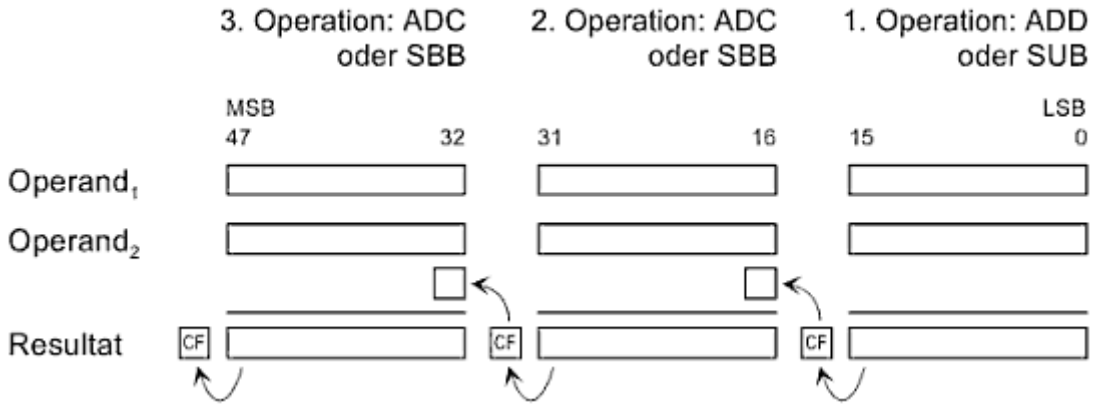
7.2.1 Das Carry-Flag

- zeigt bei vorzeichenloser Operation eine Bereichsverletzung (Über- oder Unterlauf) an
- bei vorzeichenbehafteten Operationen muss das Overflow-Flag verwendet werden

Bildungsregel: Bei einem n Bit breiten Resultat entspricht das Carry-Flag einem fiktiven weiteren Bit des Resultats
- dient auch dazu, grosse vorzeichenlose und vorzeichenbehaftete Operanden "scheibenweise" zu verarbeiten



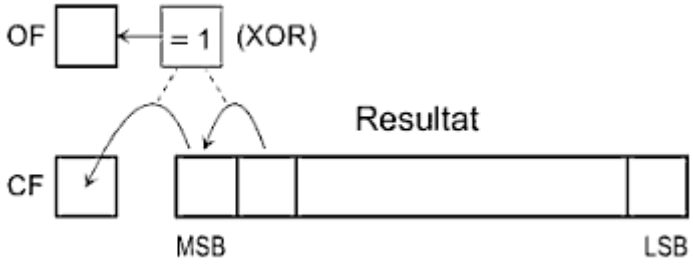
Beispiel, wie 48-Bit-Operanden in drei Schritten (mit je einer 16-Bit-Operation) mit Hilfe des Carry verarbeitet werden:
Der bei der ersten Operation allenfalls entstehende Übertrag wird im Carry-Flag gespeichert und bei der nachfolgenden Opeartion mit verrechnet (zu- oder abgezählt).



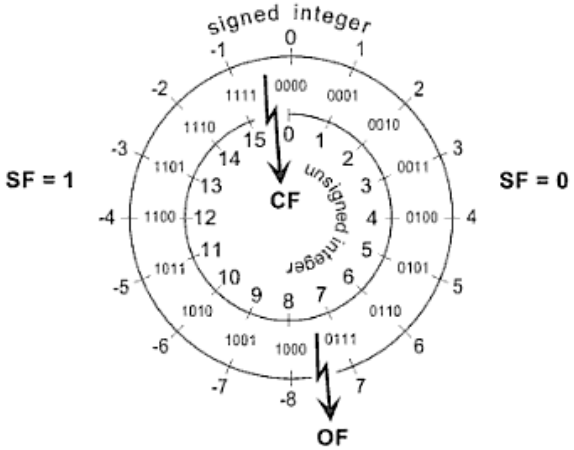
7.2.2 Das Overflow-Flag

- zeigt nach einer vorzeichenbehafteten arithmetischen Operation einen Über- oder Unterlauf an
- Voraussetzung: Zahlen liegen in der Komplement-2-Darstellung vor

Bildungsregel: Das Overflow-Flag entspricht der Exklusiv-Oder-Verknüpfung (XOR) des Übertrags zum höchstwertigen Bit und dem carry-Flag



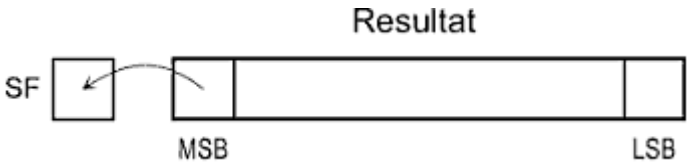
- Aufgrund Komplement-2-Darstellung: Für Ausführung der Additions- und Subtraktionsoperationen gleichgültig, ob mit vorzeichenlose (unsigned integer) oder vorzeichenbehaftete (signed integer) Operanden gerechnet wird
- Prozessor setzt immer beide Flags (Carry und Overflow)



7.2.3 Das Sign-Flag

- Vorzeichenbit
- Zeigt nach arithmetischen Operation das Vorzeichen des Resultates an
- Auswertung macht nur bei vorzeichenbehafteten Operationen Sinn

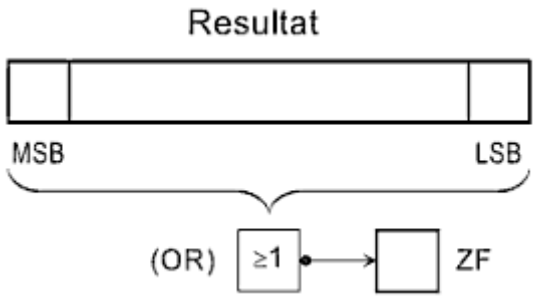
Bildungsregel: Das Sign-Flag entspricht nach den Regeln der Komplement-2-Darstellung dem höchstwertigen Bit (MSB) des Resultates



7.2.4 Das Zero-Flag

- Zeigt an, ob Resultat nach einer Operation null ist
- Null: gleiche Darstellung bei Zahlen mit und ohne Vorzeichen → keine Unterscheidung nötig

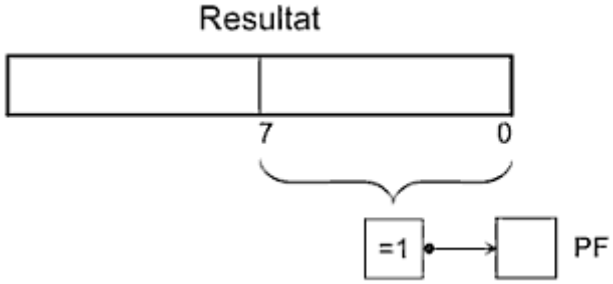
Bildungsregel: Das Zero-Flag entspricht der invertierten Oder-Verknüpfung (NOR) aller Bits des Resultats



7.2.5 Das Parity-Flag

- Zeigt eine gerade Parität des tieferwertigen Bytes des Resultats an
- Gerade Parität: gerade Anzahl auf Eins gesetzter Bits bewirkt, dass das Parity-Flag gesetzt wird, d.h. inkl. Parity-Flag wird Odd Parity erzeugt
- Stellenwert der einzelnen Bits spielt keine Rolle
- Dient der Fehlererkennung und -korrektur

Bildungsregel: Das Parity-Flag wird gebildet durch die XOR-Verknüpfung der Bits 0…7 des Resultats mit anschliessender Inversion (XNOR-Verknüpfung)



7.3 Die arithmetischen Befehle im Detail

7.3.1 Addition

- Addiert zwei Operanden und schreibt Resultat in ersten Operanden
- Ursprünglicher Inhalt geht verloren

ADD	reg, reg
	reg, mem
	reg, const
	mem, reg
	mem, const

Beeinflusste Flags

Carry	Übertrag/Überlauf bei vorzeichenlosen Operanden
Overflow	Überlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

7.3.2 Addition mit Carry

- Addiert zwei Operanden sowie den Wert des Carry-Flag (Übertrags) und schreibt Resultat in ersten Operanden
- Ursprünglicher Inhalt geht verloren

ADC	reg, reg
	reg, mem
	reg, const
	mem, reg
	mem, const

Beeinflusste Flags

Carry	Übertrag/Überlauf bei vorzeichenlosen Operanden
Overflow	Überlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

7.3.3 Erhöhen um eins (Increment)

- Addiert eins zum Operanden

INC	reg
	mem

Beeinflusste Flags

Overflow	Überlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

Hinweis: Carry-Flag wird nicht verändert!
Überlauf kann am Zero-Bit erkannt werden, da das Resultat bei Überlauf null sein muss - der (implizite) Operand ist genau eins
- Anwendung z.B. bei Kontrollvariable einer Schleife ohne dass das Carry-Flag verändert wird

7.3.4 Subtraktion

- Subtrahiert zweiten Operanden vom ersten und schreibt Resultat in ersten Operanden
- Ursprünglicher Inhalt geht verloren

	reg, reg
	reg, mem
SUB	reg, const
	mem, reg
	mem, const

Beeinflusste Flags

Carry	Übertrag/Unterlauf bei vorzeichenlosen Operanden
Overflow	Unterlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

7.3.5 Subtraktion mit Borrow

- Subtrahiert zweiten Operanden vom ersten sowie den Wert des Carry-Flags (Borrow) und schreibt Resultat in ersten Operanden
- Ursprünglicher Inhalt geht verloren

	reg, reg
	reg, mem
SBB	reg, const
	mem, reg
	mem, const

Beeinflusste Flags

Carry	Übertrag/Unterlauf bei vorzeichenlosen Operanden
Overflow	Unterlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

Hinweis: Wenn Subtraktion grosser vorzeichenbehafteter Operanden in mehrere Teilschritte zerlegt werden muss, wird der Übertrag mit Hilfe des Carry-Flags von den niederwertigeren zu den höherwertigen Operanden weitergegeben

7.3.6 Verminderung um eins (Decrement)

- Subtrahiert eins vom Operanden

DEC	reg mem

Beeinflusste Flags

Overflow	Überlauf bei vorzeichenbehafteten Operanden
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

Hinweis: Carry-Flag wird nicht verändert. Überlauf ist nur am Resultat FFh erkennbar

- Mit DEC kann zum Beispiel die Kontrollvariable einer Schleife erniedrigt werden, ohne das Carry-Flag zu verändern

7.3.7 Negieren einer vorzeichenbehafteten Zahl

- Aus einer positiven wird eine negative Zahl und umgekehrt

NEG	reg mem

Beeinflusste Flags

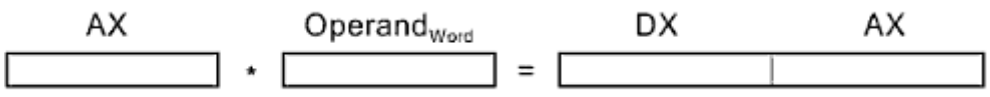
Carry	gesetzt, wenn der Operand nicht null war (Abfrage aber nicht sinnvoll)
Overflow	gesetzt, wenn es keine positive Darstellung der Zahl gibt (grösste negative Zahl)
Zero	gesetzt, wenn Resultat null ist
Sign	gesetzt, wenn vorzeichenbehaftetes Resultat negativ ist
Parity	gesetzt, wenn Parität gerade ist

7.3.8 Multiplikation

- MUL für vorzeichenlose Operanden
- IMUL für vorzeichenbehaftete Operanden

MUL / IMUL	reg mem

- Der explizite Operand legt Grösse (Byte / Word) fest. Der andere Operand (implizit) ist entsprechen das Register AL oder AX.
- Resultat ist immer doppelt so gross wie die Operanden und wird in den Akkumulator (Register AX) oder den Extended-Akkumulator (Register DX/AX) geschrieben



Beeinflusste Flags

Carry	gesetzt, wenn zur Darstellung des Resultats die höherwertige Hälfte des (Extended-)Akkumulators benötigt wird
Zero	verändert (undefiniert)
Sign	verändert (undefiniert)
Parity	verändert (undefiniert)

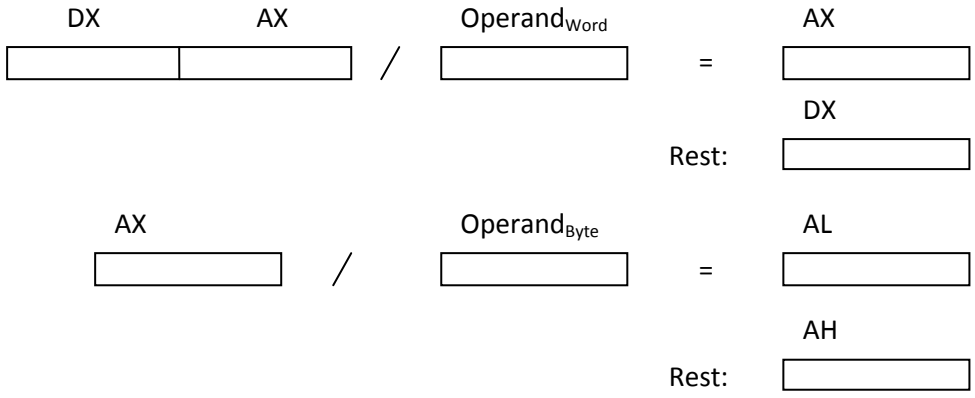
Hinweis: Beim Prozessor 8086 kann nicht mit Konstanten (Immediate-Adressierung) multipliziert werden!

7.3.9 Division

- DIV für vorzeichenlose Operanden
- IDIV für vorzeichenbehaftete Operanden

DIV / reg
IDIV mem

-Der explizite Operand legt die Grösse (Byte / Word) fest. Der erste (implizite) Operand ist entsprechen der Akkumulator (Register AX) oder der Extended-Akkumulator (AX/DX). Bei der Division wird dieser durch den angegebenen Operanden geteilt.
- Das Resultat der ganzzahligen Division wird in die tieferwertige Hälfte des ersten Operanden geschrieben (Register AL oder AX)
- Divisionsrest wird in der höherwertigen Hälfte des ersten Operanden gespeichert (Register AH oder DX)

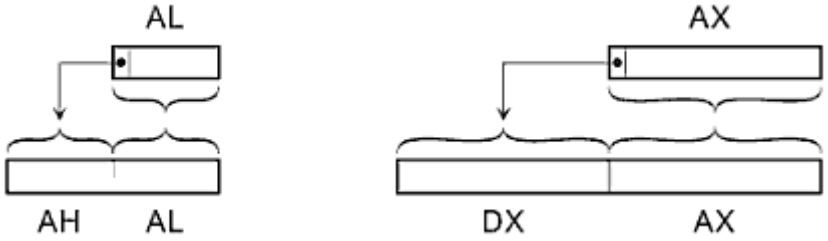


Hinweis: Sollte das Resultat der Division im Register AL resp. AX keinen Platz finden (z.B. 7ED8H / 3F6CH), wird ein Interrupt ausgelöst (Division Error). Wird dieser nicht behandelt, so ist mit einem Abbruch oder Absturz des Programms zu rechnen (je nach Umgebung).

7.3.10 Konvertierung der Operandengrösse

→ wandeln vorzeichenbehaftete Zahl in dieselbe von doppelter Länge
- CBW (Convert Byte to Word)
  - wandelt Byte-Wert im Register AL in ein Word im Register AX
- CWD (Convert Word to Doubleword)
  - wandelt Word-Wert im Register AX in ein Double-Word im Extended-Akkumulator (Register DX/AX)

- Darstellbarer Zahlenbereich soll vergrössert werden, ohne mathematischen Wert der Zahl zu verändern
- Höherwertige Hälfte des Resultats wird mit dem Vorzeichen der Eingangsgrösse (MSB) aufgefüllt
→ Sign Extension (Vorzeichenerweiterung)



8 Logische Befehle und Sift-/Rotate-Befehle

8.1 Das Prinzip der logischen Operationen

- Logische Operationen: AND (und), OR (oder), XOR (exklusiv-oder), NOT (nicht)

8.1.1 Anwendung eines Bitmusters

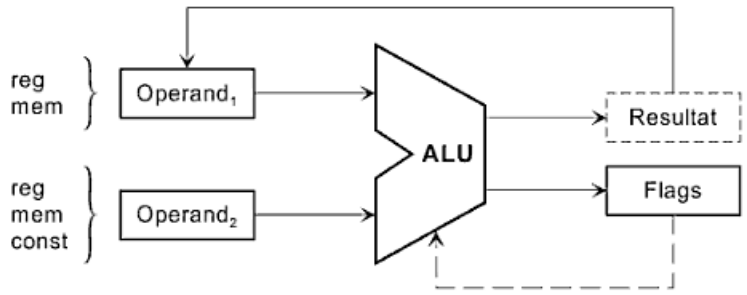
- Anwendung eines Bitmusters zum löschen, setzen oder invertieren von Bits

Operation	Eingang (a)	Konstante (b)	Ausgang (F)	Zweck / Funktion
AND	x	0	0	Bit löschen
	x	1	X	
OR	x	0	x	Bit setzen
	x	1	1	
XOR	x	0	x	Bit invertieren
	x	1	NOT x	

8.2 Logische Befehle

8.2.1 Die logischen Befehle AND, OR und XOR

- Logische Operationen werden in der ALU durchgeführt
- AND, OR, XOR brauchen zwei Operanden
- Beide Operanden brauchen gleiche Grösse (Byte/Word)
- Im ersten Operanden wird das Resultat gespeichert



	reg, reg
AND	reg, mem
OR	reg, const
XOR	mem, reg
	mem, const

Beeinflusste Flags
CF und OF werden immer zurückgesetzt (0)
ZF und SF werden gemäss Resultat gesetzt

8.2.2 Der logische Befehl NOT

- Benötigt nur einen Operanden
- Operand wird Bit-weise invertiert

NOT	reg
	mem

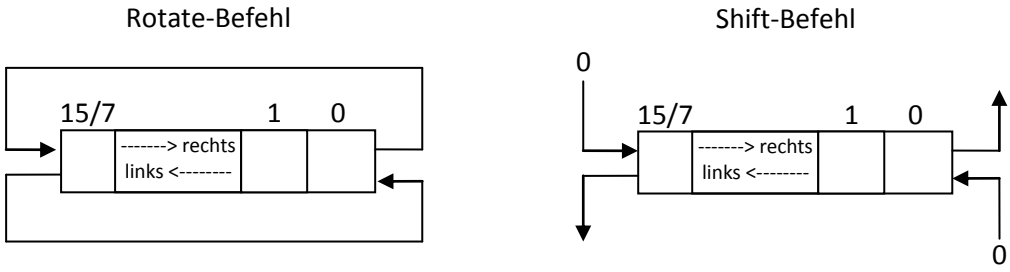
Beeinflusste Flags
Es werden keine Flags verändert

Beispiele
```
AND   AH, 00001111b
OR    BX, 000Fh
OR    BYTE PTR [SI], 10001000b
XOR   AX, BX
NOT   CX
AND   variable_a, AL
```

8.3 Shift- und Rotate-Befehle

8.3.1 Das Prinzip der Shift- und Rotate-Befehle

- Werden für Bit-Manipulationen verwendet
- Bitmuster wird um eine bestimmte Anzahl Bits nach links oder rechts bewegt
- Bei Rotate-Befehlen ist der Kreislauf geschlossen: was auf der einen Seite herausgeschoben wird, kommt auf der anderen Seite wieder rein
- Bei Shift-Befehlen wird eine Null nachgeschoben, was herausfällt ist verloren
- Rotate- und Shift-Befehle wirken auf Register oder Speicherstelle
- Soll mehr als eine Binärstelle rotiert oder geschoben werden, so muss die Anzahl der Binärstellen im Register CL stehen, und es wird im Befehl das CL-Register angegeben

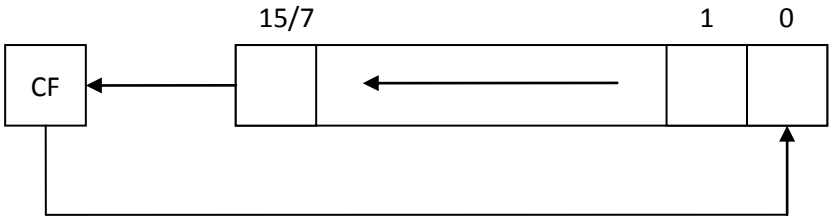


8.3.2 Rotate-Befehle

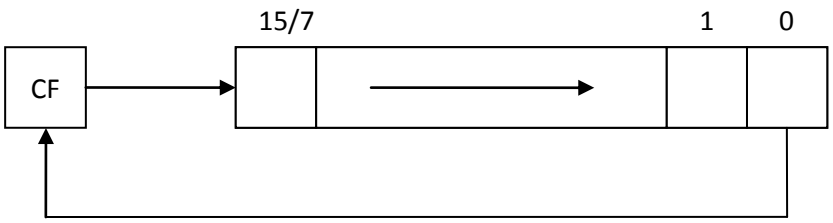
- Rotate ohne Carry-Flat rotieren nur Operanden
- Rotate durch das Carry-Flag vergrössern Operanden um ein Bit

8.3.2.1 Rotate durch Carry-Flag

RCL	reg	1
	mem	CL



RCR	reg	1
	mem	CL

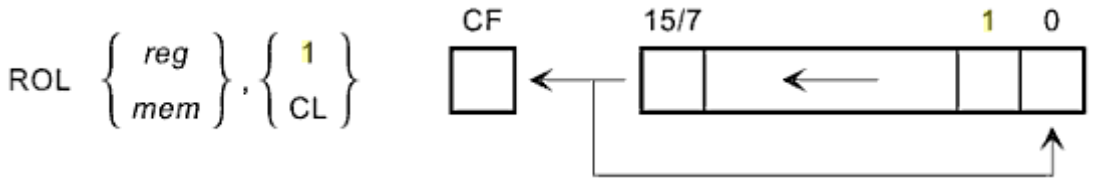


Beeinflusste Flags
CF, (OF)

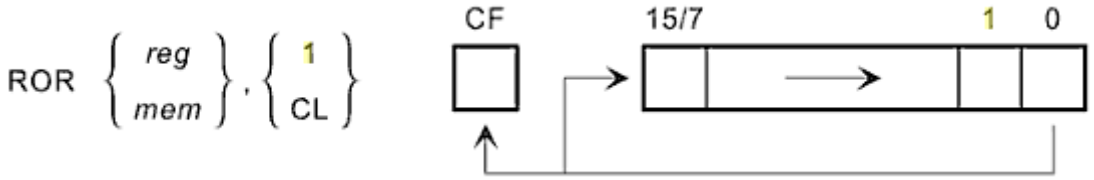
Bei Einzelbit-Rotationen ist das Overflow-Flag gesetzt, falls das MSB durch die Rotation verändert wurde. Wird das MSB nicht verändert, so ist das Overflow-Flag null. Nach Mehrbit-Rotationen ist der Zustand des Overflow-Flag undefiniert.

8.3.2.2 Rotate ohne Carry-Flag

ROL	reg	1
	mem	CL



ROR	reg	1
	mem	CL



Beeinflusste Flags
CF, (OF)

Bei Einzelbit-Rotationen ist das Overflow-Flag gesetzt, falls das MSB durch die Rotation verändert wurde. Wird das MSB nicht verändert, so ist das Overflow-Flag null. Nach Mehrbit-Rotationen ist der Zustand des Overflow-Flag undefineirt.

Beispiele

```
RCL AX, 1
ROR XY, CL
RCR BYTE PTR [SI], 1
```

8.3.3 Befehle zur Veränderung des Carry-Flag

CLC	Clear Carry Flag	Zurücksetzen des Carry-Flag (CF = 0)
STC	Set Carry Flag	Setzen des Carry-Flag (CF = 1)
CMC	Complement Carry Flag	Invertieren des Carry-Flag (CF = NOT CF)

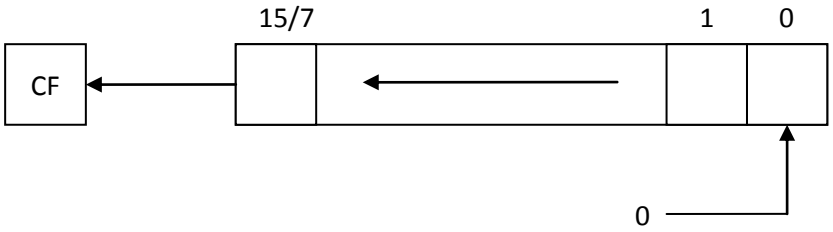
8.3.4 Shift-Befehle

- Zum Multiplizieren und Dividieren eines Operanden mit einer Zweierpotenz (Faktoren 2, 4, 8, ...)
- Unterscheidung zwischen logischen und arithmetischen Shift-Befehlen
- Für Multiplikation und Division von vorzeichenbehafteten Operanden → arithmetische Shift-Befehle
- Für vorzeichenlose Operanden → logische Shift-Befehle

8.3.4.1 Shift Left (SHL) und Shift Arithmetic Left (SAL)

- Sind funktional identisch
- Multiplizieren Wert mit zwei oder mit 2^{CL}

SHL	reg	1
SAL	mem	CL



Beeinflusste Flags
CF, ZF, SF, OF

Bei Einzelbit-Shifts zeigt ein gesetztes Carry-Flag einen Overflow eines vorzeichenlosen Operanden an. Bei vorzeichenbehafteten Operanden bleibt das Vorzeichen im Normalfall erhalten. Falls Operationen das Vorzeichen ändert, bedeutet dies Overflow und das Overflow-Flag ist gesetzt.

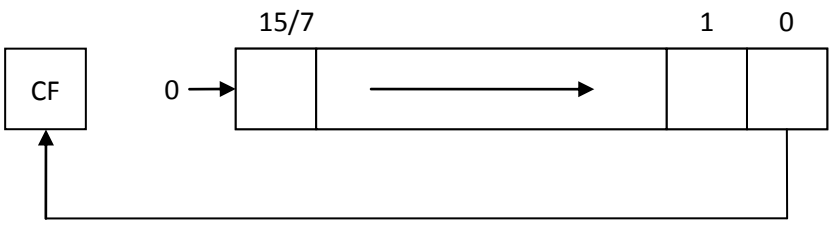
Beispiele

```
SHL AH, 1
MOV CL, 4
SAR XY, CL
SAR BYTE PTR [DI], 1
```

8.3.4.2 Shift Right (SHR)

- Dividiert einen vorzeichenlosen Wert durch zwei oder 2^{CL}

SHR	reg	1
	mem	CL



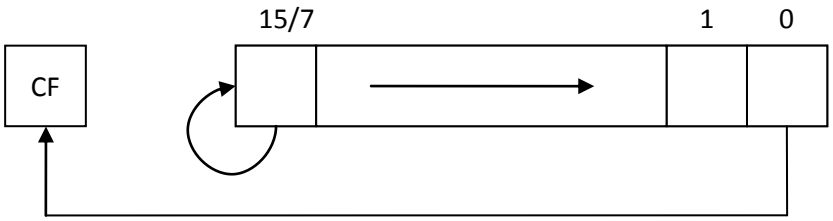
Beeinflusste Flags
CF, ZF, SF, (OF)

Bei Einzelbit-Shifts beinhaltet das Carry-Flag den Divisionsrest

8.3.4.3 Shift Arithmetic Right (SAR)

- Dividiert einen Wert durch zwei oder durch 2^{CL}
- Bei negativen Werten ist allerdings zu beachten, dass anders gerundet wird als bei der Instruktion IDIV, so dass z.B. das Resultat -3 (1111'1101b) entsteht, falls der Wert -5 (1111'1011b) um ein Bit arithmetisch nach rechts geschoben wird

SAR	reg	1
	mem	CL



Beeinflusste Flags
CF, ZF, SF, (OF)

Bei Einzelbit-Shifts beinhaltet das Carry-Flag den Divisionsrest

9 Vergleichs- und Sprungbefehle

- Sprungbefehle: Beeinflussung der Reihenfolge auszuführender Programmteile zur Laufzeit

9.1 Einteilung der Sprungbefehle

9.1.1 Begriffe

9.1.1.1 bedingte <---> unbedingte Sprungbefehle

- Bei Ausführung von unbedingten Sprungbefehlen: Programm wird **immer** an angegebener Adresse fortgesetzt
- Bei Ausführung von bedingten Sprungbefehlen: zuerst prüfen, ob Bedingung (z.B. Zustand von Flags) erfüllt ist oder nicht

9.1.1.2 Short <---> Near <---> Far-Sprungbefehle

- Distanzbereiche
  - short (8 Bit vorzeichenbehaftete Distanz)
  - near(16 Bit vorzeichenlose Distanz)
  - far (32 Bit bestehend aus Segment und Offset)

9.1.1.3 direkte <---> indirekte Sprungbefehle

- Unterscheidung der Art der Angabe des Sprungziels
- Entweder wird im Befehl neue Adresse angegeben (direkte Adressierung), oder es wird der Ort beschrieben, wo die neue Adresse zu lesen steht (indirekte Adressierung)

9.1.1.4 absolute <---> relative Sprungbefehle

- Unterscheidung der Behandlung des im Parameter Sprungziel angegebenen Wertes
- Entweder stellt Angabe vollständig die neue Adresse dar (absoluter Sprung), oder Angabe stellt Distanz zwischen dem Sprungbefehl und der neuen Adresse dar (relativer Sprung)

Mögliche Kombinationen
Siehe Buch Seite 212

9.1.2 Intra- und Intersegment-Sprungbefehle

- Alle Sprungbefehle ändern das Register IP (Instruction Pointer)
- Far-Sprungbefehle ändern zusätzlich Register CS (Codesegmentregister)

9.1.2.1 Intrasegment-Sprung

- Bei Short- und Near-Sprüngen: Nur Register IP verändert, weil Sprungziel liegt innerhalb des aktuellen (durch CS adressierten Segmentes)

Near Sprung

- Direkte Near-Sprünge sind immer relativ zu IP'. Nehmen wir an, die Distanz von IP' zur höchstwertigen Adresse im aktuellen Codesegment betrage n, wird die Adresse 0 des aktuellen Codesegmentes durch die Addition IP' + n + 1 erreicht. Dies bedeutet, dass bei den Adressrechnungen ein entstehender Überlauf ignoriert wird
- Bei indirekten Near-Sprüngen steht im durch den Befehl referenzierten Register oder in der im Befehl referenzierten Speicherzelle immer die absolute Adresse innerhalb des aktuellen Codesegmentes

Siehe Bild Seite 213

Short-Sprung

- Short-Sprünge sind bedingte oder unbedingte Sprünge, relativ zu IP'. Als Basis wird die Anfangsadresse des auf den Sprungbefehl folgenden Befehles verwendet

Siehe Bild Seite 214

9.1.2.2 Intersegment-Sprung

- Segmentregister CS wird verändert (Far-Sprungbefehl)
- Sprung führt über Segmentgrenze hinweg
- Im Parameter steht die absolute Adresse des Sprungziels, das aus einer neuen Segmentbasis und einem neuen Offset besteht

Siehe Bild Seite 215

9.2 Unbedingte Sprungbefehle

JMP $displ_8$	Das 8-Bit-Displacement wird als Signed-Zahl zum IP addiert
JMP $displ_{16}$	Das 16-Bit-Displacement wird als Unsigned-Zahl zum IP addiert
JMP $const_{32}$	Hinter dem Opcode steht direkt die 32-Bit FAR-Adresse
JMP reg_{16}	Der IP wird mit einem 16-Bit-Register geladen
JMP mem_{16}	Der IP wird mit einem 16-Bit-Speicheroperanden geladen
JMP mem_{32}	Das CS und der IP werden mit einem 32-Bit-Speicheroperanden geladen

- Sprungziele wo immer möglich mit Label (symbolische Adressen) angeben
- Assembler bestimmt anhand der Distanz selber den Typ (short, near, far) kann aber auch angegeben werden
- Assembler berechnet entsprechendes Displacement automatisch

Beispiel Short-Sprung zu Label

```
kurz:
      <= 128 Bytes zwischen
      Label x: und kurz:
      JMP kurz
x:

      JMP short kurz1
x1:   <= 127 Bytes zwischen
      Label x1: und kurz1:
kurz1:
```

- Bei indirekten Near-Sprüngen wird anstelle des Labels ein 16-Bit-Register (ohne eckige Klammern!) oder ein Speicheroperand (Typ Word) angegeben

```
      MOV BX, OFFSET nsprung
      JMP BX
      Hier können zw. 0 und < 64 KByte
      Code stehen
nsprung:
```

Der für einen speicherindirekten Sprung verwendete Speicheroperand kann mit einer beliebigen Adressierungsart angegeben werden. Damit können Sprungtabellen elegant verwendet werden.

Beispiel Sprungtabelle Buch Seite 217
Beispiel indirekte Far-Sprünge Buch Seite 218

9.3 Bedingte Sprungbefehle

- Werten Zustand von einzelnen oder mehreren Flags aus und springen an eine Adresse, wenn Bedingung erfüllt ist
- Vor bedingten Sprung müssen Flags durch arithmetische oder logische Operationen gesetzt werden
→ z.B. mit CMP und TEST

Arithmetische Sprünge: Es wird gesprungen, falls die beiden Operanden die im Sprungbefehl angegebene Bedingung des wertmässigen Grössenverhältnisses zueinander erfüllen. Um die Flags entsprechend dem wirklichen Grössenverhältnis zu setzen, werden die zwei Operanden vorgängig voneinander subtrahiert. Im Allgemeinen müssen mehrere Flags ausgewertet werden, um das Grössenverhältnis der zwei Zahlen zueinander darstellen zu können und damit auf die Erfüllung der Bedingung schliessen zu können

Flag-orientierte Sprünge: Es wird gesprungen, falls *ein* bestimmtes Flag (ZF, SF, usw.) gesetzt oder gelöscht ist

9.3.1 Arithmetische Sprungbefehle

- Sprung erfolgt, wenn Grössenverhältnis der zwei Operanden zueinander der im Befehl ausgedrückten Bedingung entspricht

Relation	unsigned	signed
=	equal	equal
<	below	less
>	above	greater

9.3.1.1 Arithmetisch "unsigned" (below or above)

Relation	Befehl		Flags / Bedingung (Sprung, falls erfüllt)	Bezeichnung
=	JE	$displ_8$	ZF = 1	Jump if equal
!=	JNE	$displ_8$	ZF = 0	Jump if not equal
<	JB	$displ_8$	CF = 1	Jump if below
not >=	JNAE	$displ_8$	CF = 1	Jump if not above or equal
>=	JAE	$displ_8$	CF = 0	Jump if above or equal
not <	JNB	$displ_8$	CF = 0	Jump if not below
<=	JBE	$displ_8$	CF = 1 or ZF = 1	Jump if below or equal
not >	JNA	$displ_8$	CF = 1 or ZF = 1	Jump if not above
>	JA	$displ_8$	CF = 0 and ZF = 0	Jump if above
not <=	JNBE	$displ_8$	CF = 0 and ZF = 0	Jump if not below or equal

9.3.1.2 Arithmetisch "signed" (greater or less)

Relation	Befehl		Flags / Bedingung (Sprung, falls erfüllt)	Bezeichnung
=	JE	$displ_8$	ZF = 1	Jump if equal
!=	JNE	$displ_8$	ZF = 0	Jump if not equal
<	JL	$displ_8$	OF != SF	Jump if less
not >=	JNGE	$displ_8$	OF != SF	Jump if not greater or equal
>=	JGE	$displ_8$	OF = SF	Jump if greater or equal
not <	JNL	$displ_8$	OF = SF	Jump if not less
<=	JLE	$displ_8$	OF != SF or ZF = 1	Jump if less or equal
not >	JNG	$displ_8$	OF != SF or ZF = 1	Jump if not greater
>	JG	$displ_8$	OF = SF or ZF = 0	Jump if greater
not <=	JNLE	$displ_8$	OF = SF or ZF = 0	Jump if not less or equal

9.3.1.3 Flag-orientierte Sprungbefehle

- Sprung erfolgt, wenn die angegebene Bedingung erfüllt ist

Befehl		Flags / Bedingung (Sprung, falls erfüllt)	Bezeichnung
JZ	$displ_8$	ZF = 1	Jump if zero
JNZ	$displ_8$	ZF = 0	Jump if not zero
JC	$displ_8$	CF = 1	Jump if carry
JNC	$displ_8$	CF = 0	Jump if no carry
JS	$displ_8$	SF = 1	Jump if sign
JNS	$displ_8$	SF = 0	Jump if no sign
JO	$displ_8$	OF = 1	Jump if overflow
JNO	$displ_8$	OF = 0	Jump if no overflow
JP	$displ_8$	PF = 1	Jump if parity
JNP	$displ_8$	PF = 0	Jump if no parity
JPE	$displ_8$	PF = 1	Jump if parity even
JPO	$displ_8$	PF = 0	Jump if parity odd

Beispiel

Sprungbefehl verzweigt zum Label Ungerade, falls das LSB gesetzt ist, d.h. falls Wert eine ungerade Zahl enthält

```
      MOV   AX, Wert   ; LSB ins Carry-Flag
      SHR   AX, 1
      JC    ungerade
x1:
```

<= 127 Bytes zwischen x1: und ungerade:

```
ungerade:
```

Beispiel
Befehlssequenz springt nach Wert_Null, falls das Register DX gleich 0 ist

```
Wert_Null:
```

<= 128 Bytes zwischen x2: und Wert_Null:

```
      OR    DX, DX
      JZ Wert_Null
x2:
```

9.3.2 Vergleichsbefehle

- Um Verhältnis von ganzen Werten zueinander oder Zustand einzelner Bits eines Wertes zu ermitteln, könnten die Operationen SUB und AND verwendet werden
- SUB vergleicht einen ganzen Wert mit einem ganzen zweiten Wert
- AND überprüft einzelne Bits innerhalt eines Wertes auf ihren Status
→ Bei SUB und AND wird der erste Operand überschrieben → unnötig für Vergleich

CMP	Vergleicht zwei 8-Bit- oder 16-Bit-Operanden durch Subtraktion. Die Flags werden beeinflusst, aber das Resultat wird **nicht** zurückgeschrieben
TEST	AND-Verknüpfung zweier 8-Bit- oder 16-Bit-Operanden. Die Flags werden beeinflusst, aber das Resultat wird nicht zurückgeschrieben

Befehlsformat
Der linke Operand wird mit dem rechten Operanden durch Subtraktion (CMP) oder durch logische UND-Verknüpfung (TEST) verglichen. Die Flags werden entsprechend dem Resultat gesetzt.

CMP TEST	reg, reg reg, mem reg, const mem, reg mem, const

Beispiel
Folgende Befehlssequenz springt nach `kleiner_gleich`, falls der Wert in AL kleiner oder gleich dem Wert in AH ist (vorzeichenlos!)

```
      CMP   AL, AH
      JBE   kleiner_gleich  ; AL <= AH
groesser:
```

Programmteil für den Fall AL > AH
(vorzeichenlos)

```
      JMP   fertig
kleiner_gleich:
```

Programmteil für den Fall AL <= AH
(vorzeichenlos)

```
fertig:
```

Beispiel
Die folgende Befehlssequenz springt nach gesetzt, falls das Bit 3 des Wertes in AL gesetzt ist

```
      TEST  AL, 00001000b
      JNZ   gesetzt          ; Resultat bei AND Verknüpfung <> 0?
geloescht:
```

Programmteil für den Fall "Bit gelöscht"

```
      JMP   fertig
gesetzt:
```

Programmteil für den Fall "Bit gesetzt"

```
fertig:
```

9.3.3 Befehle zur Konstruktion von Zählschleifen

LOOP $displ_8$	LOOP dekrementiert CX um 1 und führt, falls CX ungleich 0 ist, den Sprung an die angegebene Adresse aus. Andernfalls wird das Programm beim auf LOOP folgenden Befehl fortgesetzt
LOOPE $displ_8$ LOOPZ $displ_8$	LOOPE (Loop while equal) und LOOPZ (Loop while zero) sind unterschiedliche Mnemonics für denselben Befehl. CX wird um 1 dekrementiert und der Sprung ausgeführt, falls ein zuvor gesetztes Zero-Flag gleich 1 und CX nicht 0 ist. Andernfalls wird das Programm an der auf LOOPx folgenden Adresse fortgesetzt
LOOPNE $displ_8$ LOOPNZ $displ_8$	LOOPNE (Loop while not equal) und LOOPNZ (Loop while not zero) sind unterschiedliche Mnemonics für denselben Befehl. CX wird um 1dekrementiert und der Sprung ausgeführt, falls ein zuvor gesetztes Zero-Flag gleich 0 und CX nicht 0 ist. Andernfalls wird das Programm an der auf LOOPNx folgenden Adresse fortgesetzt
JCXZ $displ_8$	JCXZ (Jump if CX zero) führt den Sprung an die angegebene Adresse aus, falls CX den Wert 0 enthält. Andernfalls wird das Programm an der auf JCXZ folgenden Adresse fortgesetzt

- Durch diese Befehle werden **keine** Flags beeinflusst

Beispiele für Loops siehe Buch Seite 225

10 Strukturierte Codierung in Assembler

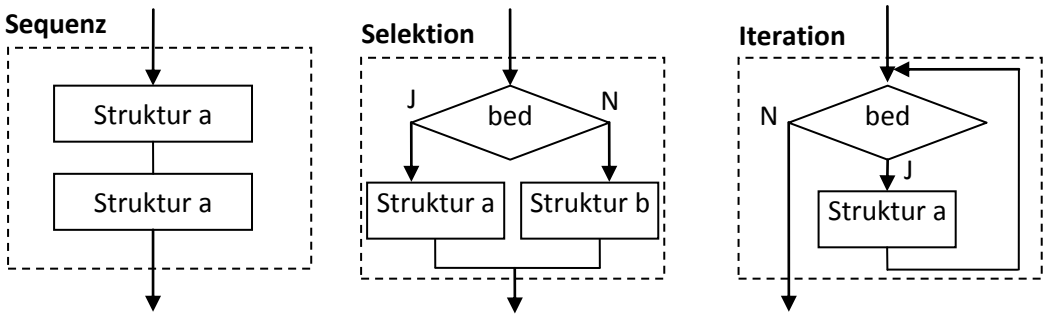
10.1 Strukturierte Codierung im Entwicklungsprozess

- Methode, die in der Realisierungsphase einer Software-Entwicklung angewendet wird
- Regeln für den Aufbau von Programmen

10.2 Das Prinzip der strukturierten Codierung

- Definition von Ablaufstrukturen, die beliebig miteinander kombiniert werden können
- Ablaufstrukturen: Immer einen eingang und einen Ausgang
- Beim Zusammenfügen: Ausgang der vorgehenden Struktur wird zum Eingang des nachfolgenden Elementes

Drei Grundstrukturen



- Mit diesen drei Grundablaufstrukturen könnten beliebig komplexe Programme klar strukturiert aufgebaut werden
- Meistens sinnvoll, um einige Elemente zu erweitern

→ do/until, for, endless und case

10.3 Realisierung der Strukturelemente in Assembler

10.3.1.1 If/then/else

```
if_n: CMP   op1, op2
      Jxx   then_n
      JMP   else_n
then_n:
      ┌─────┐
      │  A  │
      └─────┘
      JMP   eif_n
else_n:
      ┌─────┐
      │  B  │
      └─────┘
eif_n:
```

10.3.1.2 while

```
while_n:
      CMP   op1, op2
      Jxx   do_n
      JMP   ewhile_n
do_n:
      ┌─────┐
      │  A  │
      └─────┘
      JMP   while_n
ewhile_n:
```

10.3.1.3 repeat/until

```
until_n:
      ┌─────┐
      │  A  │
      └─────┘
      CMP   op1, op2
      Jxx   euntil_n
      JMP   until_n
euntil_n:
```

- Für den bedingten Sprungbefehl Jxx ist je nach Datentyp und Bedingung der entsprechende Befehl zu setzen

Bedingung (op1 bed op2)	Datentyp	
	Unsigned	Signed
gleich (=)	JE	JE
ungleich (!=)	JNE	JNE
kleiner (<)	JB	JL
grösser/gleich (>=)	JAE	JGE
kleiner/gleich (<=)	JBE	JLE
grösser (>)	JA	JG

Vereinfachte Version für if/then/else (für bedingte Sprünge im Bereich -128… +127 Byte) siehe Buch Seite 234

- Jxx# bedeutet: springe, falls die Bedingung nicht erfüllt ist

10.3.1.4 endless (Endlosschleife)

```
endless:
      ┌─────┐
      │  A  │
      └─────┘
      JMP   endless
```

10.3.1.5 for (universelle Zählschleife)

```
      op := n

for_n:     CMP   op, m
      JBE   body_n
      JMP   efor_n
body_n:
      ┌──────┐
      │  A   │
      └──────┘
      INC   op
      JMP   for_n
efor_n:
```

10.3.1.6 do (Zählschleife unter Verwendung des LOOP-Befehls)

```
      MOV   CX, op

until_n:
      ┌──────┐
      │  A   │
      └──────┘
      LOOP   until_n
euntil_n:
```

10.3.1.7 switch-case

```
case_n:          CMP   op, a
            JE    do_a_n
            JMP   case_b_n
do_a_n:     ┌──────┐
            │  A   │
            └──────┘
            JMP   ecase_n

case_b_n:   CMP   op, b
            JE    do_b_n
            JMP   case_c_n
do_b_n:     ┌──────┐
            │  B   │
            └──────┘
            JMP   ecase_n
case_c_n:   CMP   op, c
            JE    do_c_n
            JMP   case_d_n
do_c_n:     ┌──────┐
            │  C   │
            └──────┘
            JMP   ecase_n
case_d_n:
do_d_n:     ┌──────┐
            │  D   │
            └──────┘

ecase_n:
```

10.3.2 Strukturelemente mit mehreren Bedingungen

- Durch UND oder ODER-Funktionen verknüpfte Strukturbedingungen

10.3.2.1 Zwei Bedingungen mit UND verknüpft

```
if_n: CMP   op1, op2
      Jxx   bed_2_n
      JMP   else_n
bed_2_n:   CMP   op3, op4
           Jxx   then_n
           JMP   else_n
then_n:
      ┌──────┐
      │  A   │
      └──────┘
      JMP   eif_n
else_n:
      ┌──────┐
      │  B   │
      └──────┘
eif_n:
```

10.3.2.2 Zwei Bedingungen mit ODER verknüpft

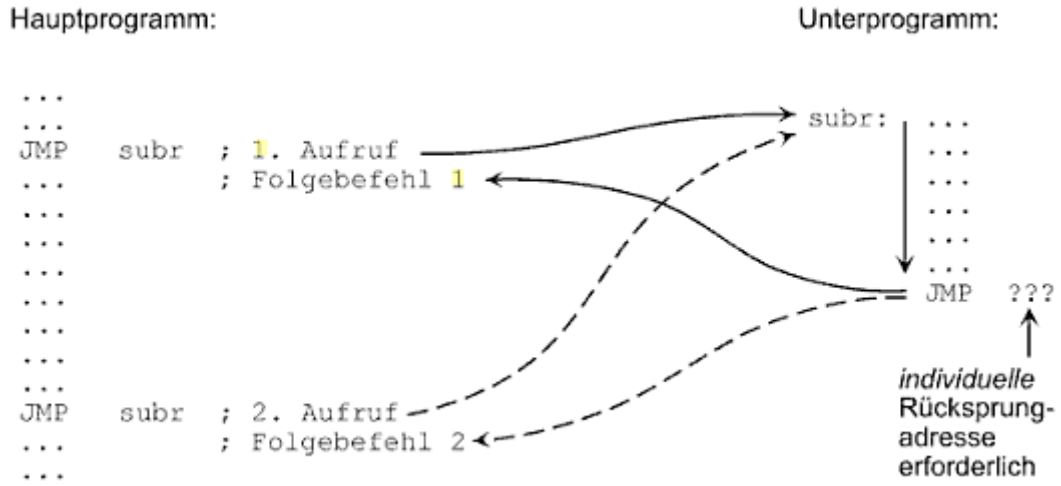
```
if_n:      CMP   op1, op2
           Jxx   then_n
           (JMP  bed_2_n)
bed_2_n:   CMP   op3, op4
           Jxx   then_n
           JMP   else_n
then_n:
      ┌──────┐
      │  A   │
      └──────┘
      JMP   eif_n
else_n:
      ┌──────┐
      │  B   │
      └──────┘
eif_n:
```

- Sind die Bedingungen durch eine UND-Funktion verknüpft, so müssen beide Bedingungen erfüllt sein, damit der then-Zweig abgearbeitet wird.
- Sind die Bedingungen durch eine ODER-Funktion verknüpft, so muss nur eine der beiden Bedingungen erfüllt sein, damit der then-Zweig durchlaufen wird

11 Unterprogramme und Stack

11.1 Einführung

- Unterprogramme (Subroutinen, Prozeduren, Funktionen)
  - wichtige Konzepte für klare und überschaubare Programme
  - Speicher sparen (Befehlsfolge erscheint nur einmal im Code)



- Beim Sprung zum Unterprogramm (JMP subr) soll bestimmte Befehlsfolge ablaufen
- Anschliessend soll der Prozessor beim nachfolgenden Befehl des Hauptprogramms fortfahren (Folgebefehl)
- Aufruf von subr kann von verschiedenen Orten mit JMP subr erfolgen, der Rücksprung aus dem Unterprogramm an unterschiedliche Folgeadressen ist aber mit JMP ??? nicht so einfach!

→ Bei JMP ??? ist individuelle Rücksprungadresse notwendig

→ Grundsätzliches Problem: wie erfolgt der Rückweg vom Schluss der Subroutine zum Befehl "Folgebefehl"

- Rücksprung muss je nach Ort des Aufrufes an verschiedene Stellen erfolgen

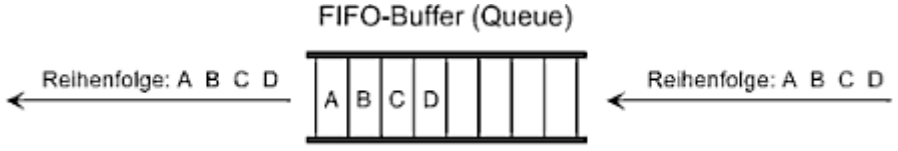
  → *Sicherung der Rücksprungadresse zum Zeitpunkt des Aufrufes!*

11.2 Das Stack-Prinzip

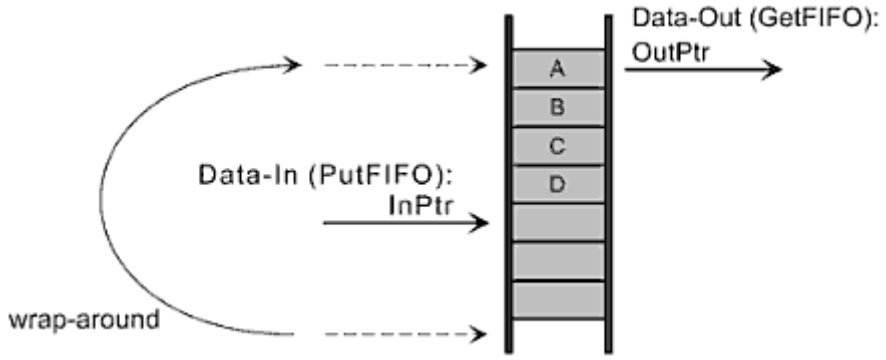
- Für die kurzfristige Ablage und Pufferung einer variablen Anzahl von Elementen (Daten, Adressen) existieren zwei Speichertechniken Stack und Queue

11.2.1 Queue

- FIFI = First In First Out
- Elemente werden in der gleichen Reihenfolge ausgelesen, wie sie vorher eingeschrieben wurden
- Warteschlange



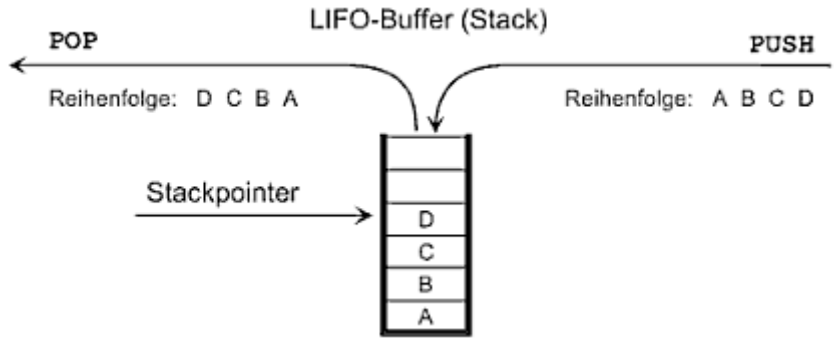
- Wird dann verwendet, wenn Produzent und Konsument von Daten nicht starr synchron laufen, die Verarbeitung aber in der ursprünglichen Reihenfolge geschehen soll (Tastaturpuffer / Message-Boxen bei parallelen Prozessen)
- Werden als Puffer bei nicht synchron laufendem Produzent/Konsument-Verhalten eingesetzt



- Ringpuffer
- Einschreibezeiger InPtr zeigt auf nächsten freien Platz
- Auslesezeiger OutPtr liest ältestes Element aus
- Bei jeder Schreib- (PutFIFO) bzw. Leseoperation (GetFIFO) müssen der FIFO-Überlauf und das Erreichen des Zustandes Full bzw. Empty überprüft werden

11.2.2 Stack

- LIFO = Last In First Out
- Elemente werden in umgekehrter Reihenfolge ausgelesen
- Bei geschachtelten Subroutinenaufrufen mit Rücksprungadressen notwendig, da jeweils die erste notwendige Rücksprungadresse auf dem Stack gerade vom letzten Subroutinenaufruf stammt



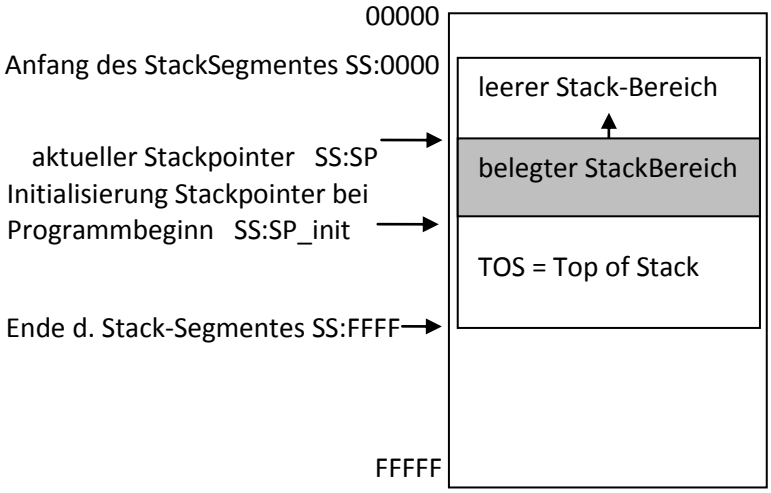
- Stack hat nur ein Zugriffsort: Ein neu eingeschriebenes Element versteckt den Zugriff auf die bisher gespeicherten Elemente
→ es kann immer nur das zuletzt eingeschriebene Element ausgelesen werden
→ nur ein Zeiger notwendig (Stackpointer)

11.3 Stack-Realisation beim 8086

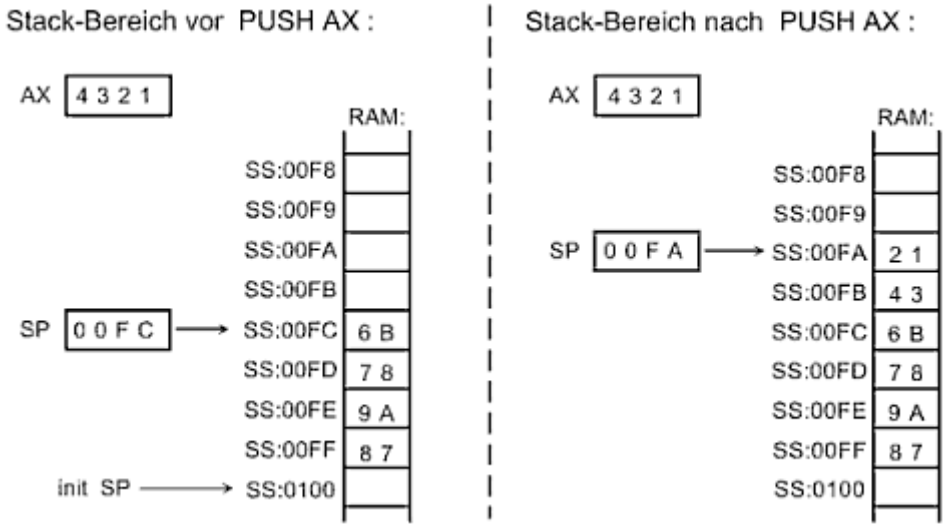
11.3.1 Stack-Segment und Stackpointer

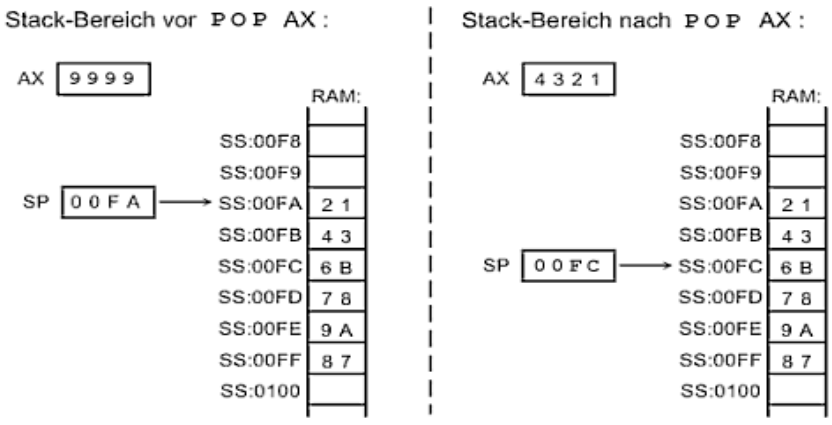
- Stack befindet sich im RAM-Bereich des Speichers und wird immer über das Stack-Segmentregister SS angesprochen
- Mit dem Stackpointer wird der Stack-Bereich verwaltet, der von hohen nach tiefen Adressen gefüllt wird
- Stackoperationen: immer wortweise (16-Bit)
- Beim Schreiben auf Stack (Operation PUSH) wird der Stackpointer fortlaufend dekrementiert (-2)
- Beim Lesen vom Stack (Operation POP) wird der Stackpointer wieder inkrementiert (+2)
- Stackpointer zeigt auf das zuletzt in den Stack geschriebenes Wort → letzten ***belegten*** Platz

- Beim Beschreiben des Stacks muss der Stackpointer zuerst um 2 erniedrigt werden (pre-decrement), und anschliessend wird der Datenwert via SP in den Speicher geschrieben
- Beim Lesen wird zuerst der Datenwert via SP geholt und anschliessend der Stackpointer um 2 erhöht (post-increment)



11.3.2 Funktionen der Stack-Operationen PUSH und POP





11.4 Stack-Befehle des 8086

- Daten sichern / zurückholen: PUSH POP
- Unterprogramme aufrufen CALL RET

11.4.1 Daten sichern / zurückholen

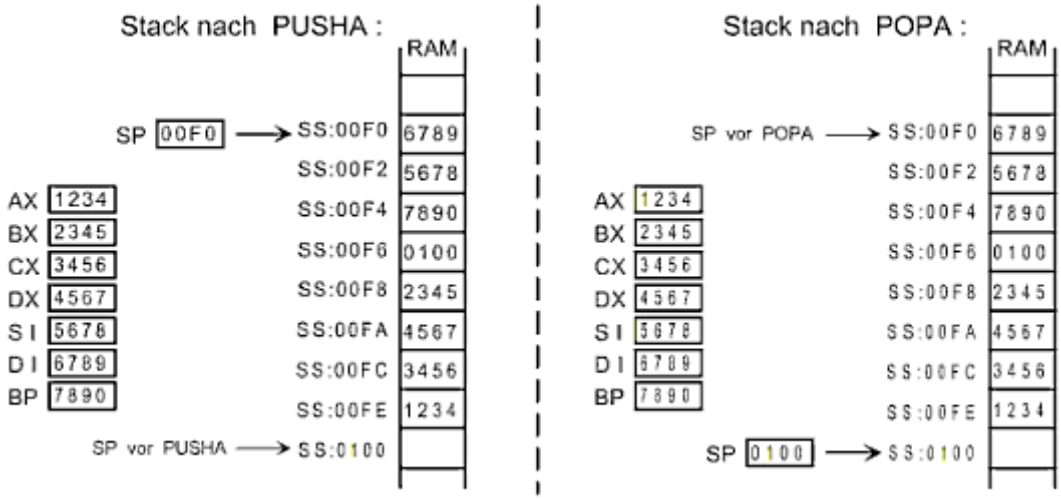
Syntax:

`PUSH reg16`	Wortregister auf den Stack schreiben
`PUSH mem16`	Speicherwort auf den Stack schreiben
`PUSHF`	Flag-Register auf den Stack schreiben
`POPF`	Flag-Register vom Stack zurückladen
`POP mem16`	Speicherwort vom Stack zurückladen
`POP reg16`	Speicherregister vom Stack zurückladen

- POP CS ist verboten!

Beispiele:

```
PUSH  resultat          ; Variable resultat
PUSH  [BX+7]            ; Speicherwort an Adresse [BX+7]
PUSH  tab[SI]           ; Speicherwort an Adresse tab[SI]

PUSHA             ; Alle 8 Arbeitsregister auf den Stack schreiben
POPA              ; Alle 8 Arbeitsregister vom Stack lesen
PUSH  const16           ; 16-Bit-Konstante auf den Stack schreiben
```



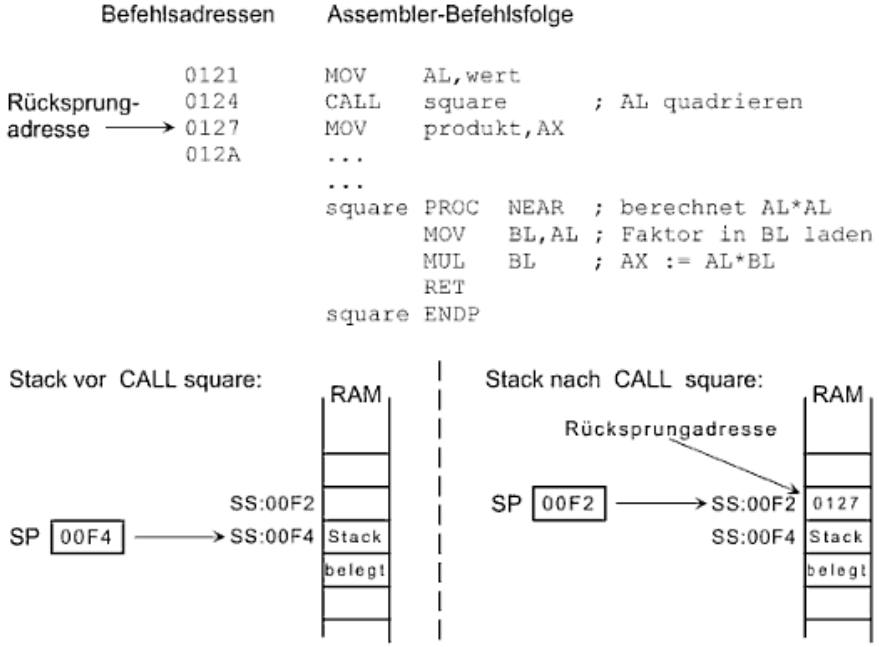
Bei PUSHA werden alle 8 Arbeitsregister in folgender Reihenfolge auf den Stack geschrieben bzw. in umgekehrter Reihenfolge vom Stack gelesen:
AX CX DX BX SP+ BP SI DI

11.4.2 Unterprogramme aufrufen / beenden

- Near-Calls (Intrasegment, innerhalb 64K) bei denen nur der Instruction-Pointer (IP) verändert wird
- Bei Far-Calls wird zusätzlich noch das Codesegmentregister (CS) gesichert/geladen

Syntax: `CALL subadr` Unterprogramm-Aufruf an Adresse subadr

- Rücksprungadresse (Offset-Adresse des Befehls nach dem CALL-Befehl) wird im Stack gesichert




```
PUSH  IP          ; "hypothetischer" Befehl "IP sichern" (existiert
                  eigentlich nicht)
JMP   subadr      ; Sprung zum Unterprogramm
```

- Jedes Unterprogramm muss mit folgendem Rücksprungbefehl beendet werden

Syntax: `RET` Beenden des Unterprogramms

- RET-Befehl am Ende des Unterpgrogramms lädt die Rücksprungadresse vom Stack zurück in den IP (entspricht "hypothetischen" Stack-Befehl POP IP)

11.5 Deklaration von Unterprogrammen in Assembler

11.5.1 Minimaldeklaration: PROC, ENDP und RET

```
square      PROC        ; Beginn der Prozedur-Klammer
            MOV   BL, AL      ; Procedure-Body = Prozedur-Befehle
            MUL   BL
            RET         ; Zwingender Abschluss jeder Prozedur
square      ENDP        ; Ende der Prozedur Klammer
```

11.5.2 Far- und Near-Prozedur-Deklaration

- Im 8086: zwei Arten von Adressen (near = 16Bit und far = 32Bit)
- Bei Near-Prozeduren wird 16-Bit-Offsetadresse auf den Stack gelegt
- Bei Far-Prozeduren wird die 32-Bit-Adresse CS:IP des auf den Call-Befehl folgenden Befehls auf den Stack gelegt

Syntax:

```
name  PROC  FAR   Deklaration einer Far-Prozedur
name  PROC  NEAR  Deklaration einer Near-Prozedur (default)
```

11.5.3 Prozedurdeklaration mit Register-Save und -Restore

- Um Registerinhalte, die im aufrufenden Programm verwendet werden, nicht zu zerstören, werden oft diejenigen Registerinhalte, die innerhalb der Prozedur verändert werden, auf den Stack gesichert

```
test  PROC  NEAR
      PUSH  AX    ; save registers
      PUSH  BX
      PUSH  CS
      PUSH  SI
      …
; Hier steht der Kern des Unterprogramms
      …
; Die gesicherten Register müssen in der umgekehrten Reihenfolge
zurückgespeichert werden
      POP   SI    ; restore registers
      POP   CS
      POP   BX
      POP   AX
      RET
test  ENDP
```

Hinweis: In Interrupt Service Routinen müssen immer alle veränderten Register gesichert werden

11.5.4 Aufruf von Unterprogrammen

- Eine Prozedur wird mit CALL aufgerufen

```
CALL test  ; Aufruf der Prozedur test
```

Damit wird die folgende Befehlsfolge der Prozedur test abgearbeitet

```
      PUSH  AX    ; save registers
      PUSH  BX
      PUSH  CS
      PUSH  SI
      …     ; Hier steht der Kern des Unterprogramms
      …
      POP   SI    ; restore registers
      POP   CS
      POP   BX
      POP   AX
      RET
```

11.6 Parameterübergabe an Unterprogramme

11.6.1 Arten an Parameterübergabe

- Drei Arten von Parameterübergaben

Globale Variablen: Der Datenaustausch über gemeinsame Speicherbereiche ist in kleinen Programmen praktisch, aber wegen der Fehleranfälligkeit und der schlechten Wartbarkeit möglichst zu vermeiden

Wertparameter: Für Grössen, die nur in die Prozedur hineingehen (Input-Parameter). In C sind dies Konstanten oder Variablen in Parameterlisten

Adressparameter: Für Grössen, die als Resultate aus der Prozedur an das aufrufende Programm zurückgegeben werden (Output-Parameter). In C wird dies mit Pointern realisiert

11.6.2 Realisierung in der Assembler-Programmierung

Globale Variablen: Es wird im Hauptprogramm und im Unterprogramm mit denselben Variablennamen auf dieselben Adressen zugegriffen

Beispiel: Vor der Prozedur shiftwl muss das aufrufende Programm die Werte in den globalen Variablen glv_data und glv_n richtig initialisieren, nach dem Aufbau steht das Resultat in der globalen Variablen glv_data zur Verfügung

```
; im Datensegment:
data_seg   SEGMENT
glv_data   DW    ?     ; Globale Variable date und
glv_n      DW    ?     ; n für Prozedur shiftwl
data_set   ENDS

; im Hauptprogramm im Codesegment:
code_seg   SEGMENT
     MOV   DX, data   ; Parameter laden
     MOV   glv_data, DX
     MOV   AL, n
     MOV   glv_n, AL
     CALL  shiftwl    ; Prozedur-Aufruf
     MOV   DX, glv_data    ; Resultat weiterverwenden
     ...
     ...
; Prozedur-Deklaration von shiftwl
shiftwl    PROC  NEAR
     MOV   AX, glv_data
     MOV   CL, glv_n
     SHL   AX, CL
     MOV   glv_data, AX
     RET
shiftwl    ENDP

; Konstantendefinitionen
data  DW    1234 ; "aktueller" Wert
n     DW    3    ; Anzahl shifts

code_seg   ENDS
```

Wertparameter: Vor Aufruf der Prozedur werden die aktuellen Werte in geeignete Register geladen (8-/16-Bit-Grössen). Das Unterprogramm verwendet diese Registerinhalte

Beispiel: In der folgenden Lösung wird die Anzahl Shifts als Wertparameter im Register CL und das zu schiebende Bitmuster als Adressparameter (Input und Output) im Register BX übergeben

```
; im Datensegment:
data_seg   SEGMENT
data  DW   ?     ; "aktueller" Wert
n     DB   ?     ; Anzahl shifts
data_seg   ENDS

; im Hauptprogramm im Codesegment
code_seg   SEGMENT
      MOV   data, 1234 ; Werte initialisieren
      MOV   n, 5
      MOV   BX, OFFSET data ; Adresse von data laden
      MOV   CL, n      ; shift-count laden
      CALL  shiftwl    ; Prozedur-Aufruf
      MOV   DX, data   ; Resultat verwenden
      ...
      ...

; Prozedur-Deklaration von shiftwl
shiftwl    PROC  NEAR
      MOV   AX, [BX]   ; data nach AX laden
      SHL   AX, CL
      MOV   [BX], AX
      RET
shiftwl    ENDP
code_seg   ENDS
```

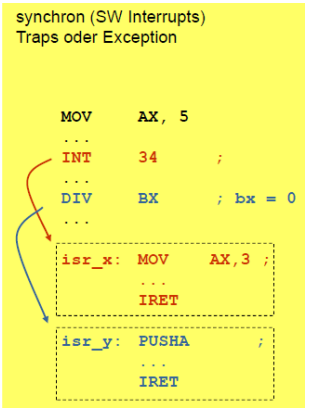
- In C würde der Prozedurkopf mit den Input- und Output-Parametern wie folgt deklariert:
void shiftwl(int* data, char n)

Adressparameter: Vor dem Aufruf der Prozedur wird die Adresse des Parameters in ein geeignetes Register geladen (meist BX, SI oder DI), und innerhalb der Prozedur wird mit der indirekten Adressierung auf den Speicherplatz der Variablen zugegriffen (read oder write = Input-/ Output-Parameter)

12 Interrupt

- Interrupts sind abrupte Unterbrechungen des Programmablaufs auf Grund eines Ereignisses

12.1 Interrupt Typen

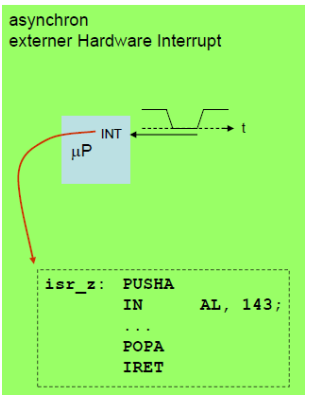


12.1.1 Synchron zur Programmausführung (SW Interrupts)

- Trap (Instruktion INT) für System Calls
  - von Software erzeugter Interrupt
  - durch expliziten Aufruf z.B. INT 34

- Exception
  - unerwartetes Ereignis während des Programmablaufes
  - z.B. divide by zero, overflow, single step, etc

12.1.2 Asynchron zur Programmausführung



- Durch Hardware ausgelöst
  - zwischen zwei Instruktionen
  - z.B. Timer, "data ready", "data transmitted", "page fault", etc.
  - externe Quelle ausserhalb des Prozessors

12.1.3 Traps → Instruktion INT

- Ausführen des Befehls INT n löst Interrupt aus
- Wie ein Call-Befehl, aber statt Prozeduradresse wird eine 8-Bit Interrupt Nummer (0-255) angegeben
  - 0-31 sind reserviert, 32-255 frei verfügbar
- Ablauf
  - Flags und Rücksprungadresse auf Stack schreiben
  - Mit Interrupt-Nummer wird Sprungadresse aus Interrupt-Vektor-Tabelle gelesen
  - Sprung an die gelesene Adresse → ISR
  - Ausführung der ISR
  - Rücksprung mit Befehl RET
    - Flags und Rücksprungadresse werden vom Stack gelesen

12.1.4 Traps (INT n) vs. Prozeduraufrufe

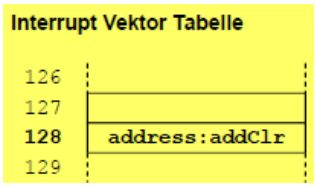
Prozeduraufruf

```
...
MOV     AX, 5
MOV     BX, 3
CALL    addClr      ← Adresse
...

addClr: ADD     AX, BX
        XOR     BX, BX
        RET
```

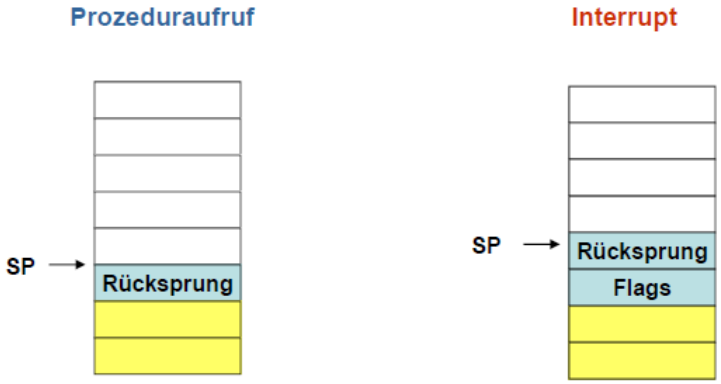
Traps (INT n)

```
...
MOV     AX, 5
MOV     BX, 3
INT     128         ← Referenz
...

addClr: ADD     AX, BX
        XOR     BX, BX
        IRET
```

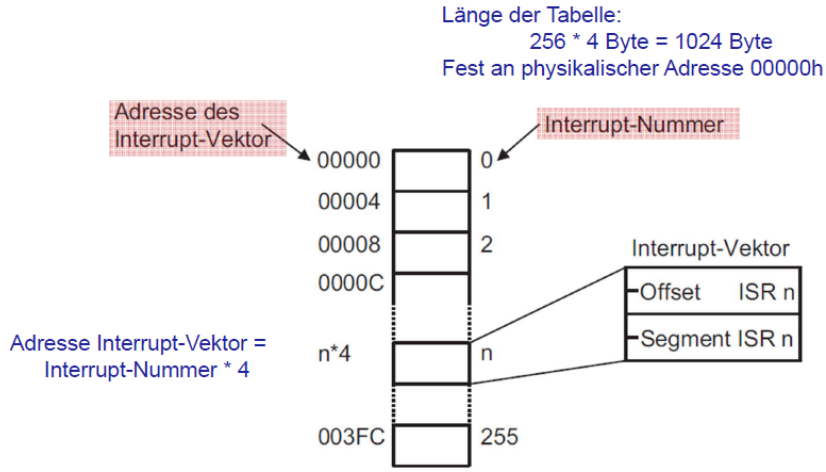
- Vorteil:
  - Adresse der ISR muss nicht bekannt sein
  - Interrupt Nr. (Referenz) genügt



Unterschiede auf dem Stack



12.2 Interrupt-Vektor-Tabelle



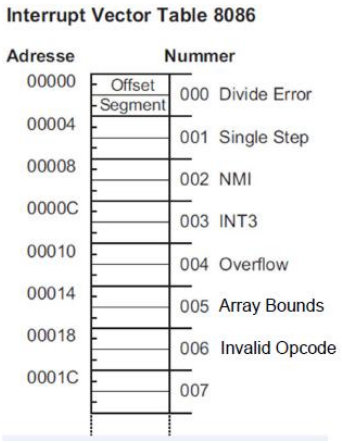
12.2.1 Initialisierung Interrupt-Vektor-Tabelle

Beispiel ISR für Interrupt Nummer = 62 registrieren

```
MOV   AX, 0
MOV   EX, AX
MOV   SI, 4*62
MOV   WORD PTR ES:[SI], OFFSET isr
MOV   WORD PTR ES:[SI+2], SET isr
```

12.3 Exceptions

- Nicht sperrbare Software Interrupts
- Treten bei bestimmten Befehlen oder Fehlern auf
  - Divide Error (Division durch Null bei DIV/IDIV)
  - -Overflow (Falls OF=1 beim Befehl INTO)
  - Array Bounds (186) (Befehl BOUND)
  - Invalid Opcode (186)
  - Breakpoint "INT type 3" Befehl
- Fest zugewiesene Interrupt Nummern



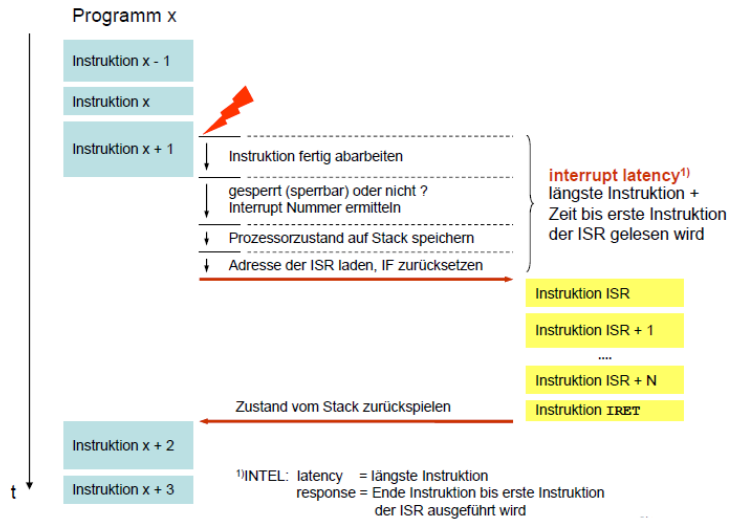
12.4 Hardware Interrupts

- Aufruf der ISR: Gleicher Mechanismus für den Aufruf wie bei Software Interrupts
  - Sichern der Flags und der Rücksprungadresse auf Stack
  - Sprungadresse aus Interrupt-Vektor-Tabelle

- Unterschiedlicher Auslöser
  - Synchroner (SW) Interrupt → Zeitpunkt ist bekannt
  - Asynchroner (HW) Interrupt: z.B. Flanke an einem Pin
    - kann jederzeit auftreten → Zeitpunkt nicht bekannt

- Interrupt-Nummer
  - Jede Interrupt-Quelle hat eine unterschiedliche Interrupt-Nummer
  - Interrupt Controller meldet Interrupt-Nummer an Prozessor
  - Interrupt-Nummer → Sprungadresse

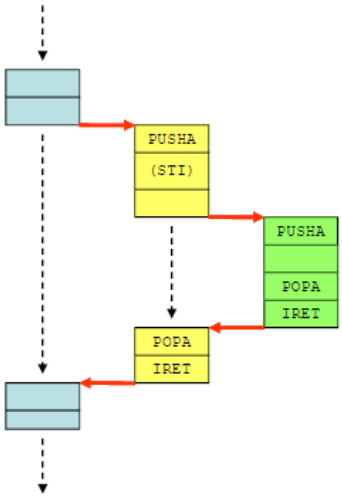
12.4.1 Ablauf externer (HW) Interrupt



- Sperrbare Interrupts
  - Über Interrupt-Enable Flag (IF) sperrbar
  - Reset: IF = 0 → Interrupt gesperrt
  - STI: setzen des IF → Interrupt freigegeben
  - CLI: löschen des IF → Interrupt gesperrt

- Nicht sperrbare Interrupts
- Non-maskable Interrupt (NMI)
- Power-fail, Not-Aus, Watchdog, …

12.5 Nested Interrupts (extern), Register sichern



- Eintritt in die ISR: HW Interrupts
  - das Interrupt Flag wird automatisch zurückgesetzt
    - weitere Interrupts sind damit gesperrt
    - mit STI (set interrupt flag) können Interrupts wieder zugelassen werden
  - alle Register sichern, die verändert werden

Softwarestruktur einer ISR (x86)

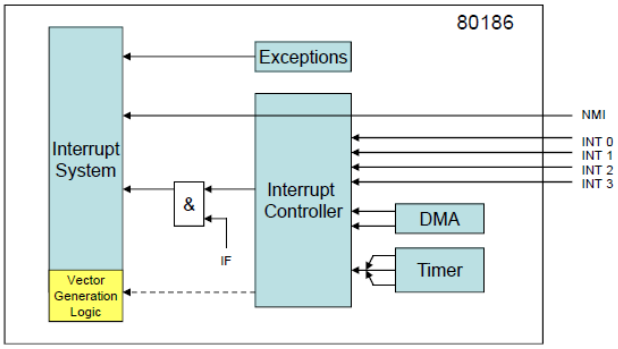
```
ISR_HW: PUSHA    ; save all registers
        (STI     ; enable nested interrupts)
        ...      ; code to be executed
                 ; by ISR
        POPA     ; restore all registers
        IRET     ; return from interrupt
```

12.6 Stack-Aufruf und Rücksprung

- Beim ISR Aufruf werden Flags und Rücksprungadresse auf den Stack gesichert
- Nach ISR Abschluss erfolgt damit die Rückkehr ins unterbrochene Programm
- Wieso Flags?
  - Unterbrechung kann jederzeit erfolgen
  - z.B. zwischen CMP und JZ Instruktionen
- Rücksprung von ISR
  - spezieller Return Befehl: IRET
  - holt Rücksprung vom Stack
  - Flags werden vom Stack zurückgeladen
    → Interrupt-Enable-Flag ist dadurch wieder gesetzt

12.7 Interrupt-System des 80186

- 801086 mit integriertem Interrupt Controller
  - die ICU: Interrupt Control Unit

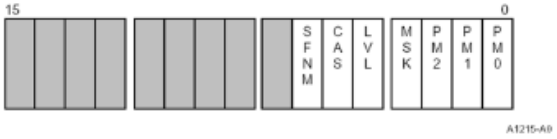


- Für jeden Interrupt gibt es ein Control Register

Register Mnemonic: I0CON, I1CON

Register Function: Control register for the cascadable external interrupt pins

1 0 0 0 0 0 0



Bit Mnemonic	Bit Name	Reset State	Function
SFNM	Special Fully Nested Mode	0	Set to enable special fully nested mode.
CAS	Cascade Mode	0	Set to enable cascade mode.
LVL	Level-trigger	0	Selects the interrupt triggering mode: 0 = edge triggering 1 = level triggering. The LVL bit **must be set** when external 8259As are cascaded into the Interrupt Control Unit.
MSK	Interrupt Mask	1	Clear to enable interrupts from this source.
PM2:0	Priority Level	111	Defines the priority level for this source.

NOTE: Reserved register bits are shown with gray shading. Reserved bits must be written to a logic zero to ensure compatibility with future Intel products.

Alle Control Register werden über den Peripheral Control Block angesprochen

Die Control Register für INT 0 und INT 1 liegen an den Adressen 0FF38h resp. 0FF3Ah

INT 0 wird z.B. folgendermassen initialisiert

```
MOV   DX, 0FF38h
MOV   AX, 0040h
OUT   [DX], AX
```

12.8 Initialisieren von externen Interrupts (HW)

```
INT0NUM     EQU  12

init:  MOV  AX, 0
       MOV  ES, AX
       MOV  SI, 4 * INT0NUM
       MOV  WORD PTR ES:[SI],OFFSET myISR
       MOV  WORD PTR ES:[SI+2],SEG myISR

       MOV  DX, 0FF38h
       MOV  AX, 0040h
       OUT  [DX], AX

       STI

prog:  ...
       JMP    prog

myISR: PUSHA
       ...
       POPA
       IRET
```

Registrieren der ISR *myISR* in der Interrupt Vektor Tabelle

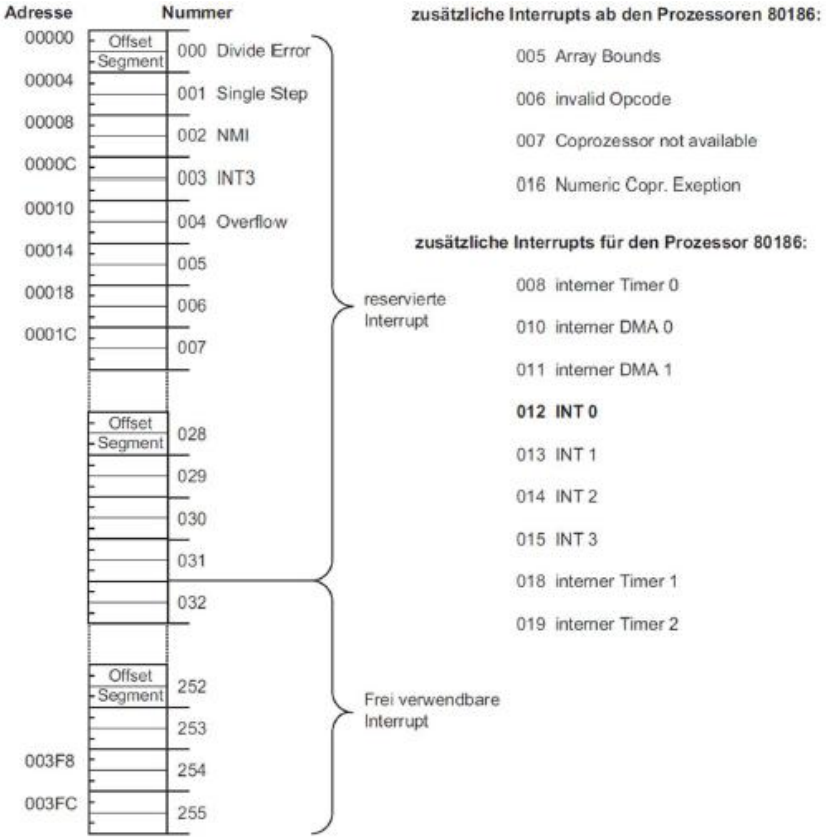
Interrupt Control Register initialisieren

Enable Interrupt

Hauptprogramm

Service Interrupt in ISR

12.9 Interrupt Vektor-Tabelle



12.10 Interrupt Prioritäten

- Mit Interrupt Nesting: Prioritätssteuerung notwendig
  - welcher Input wird zuerst bedient?

- Interrupt Controller Hardware
  - Hardware priorisiert die Anfragen
  - Prioritäten oft programmierbar
  - Default Prioritäten beim 80186

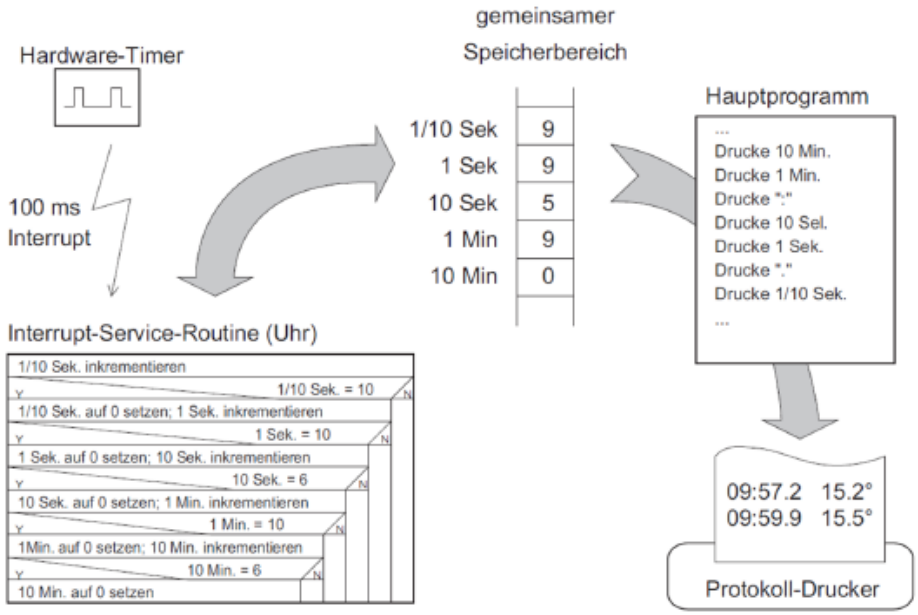
- Wenn kein Interrupt Controller & single interrupt line
  - I/O-Geräte abfragen, wer Interrupt ausgelöst hat
  - Priorität durch Abfragereihenfolge gegeben

80186 Default Priorities

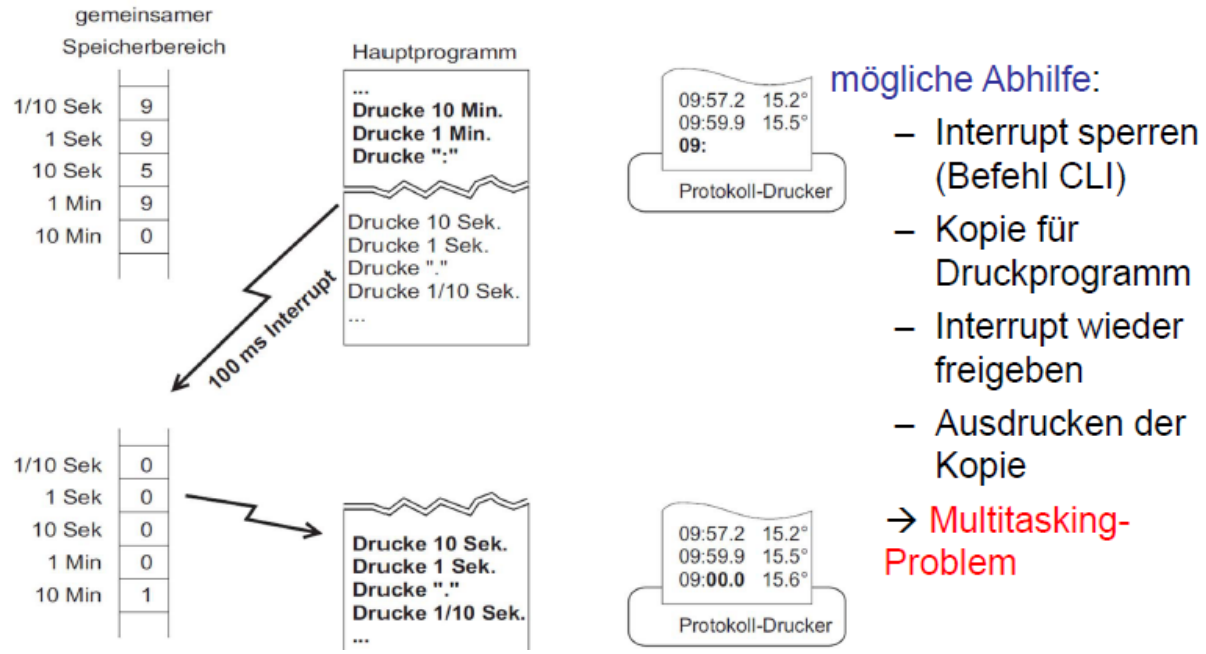
Interrupt Name	Relative Priority
Timer 0	0 (a)
Timer 1	0 (b)
Timer 2	0 (c)
DMA0	1
DMA1	2
INT0	3
INT1	4
INT2	5
INT3	6

12.11 Datenkonsistenz Problematik bei Interrupt: Beispiel

- Timer-Interrupt alle 100ms → +0.1 s
- Zeitstempel als Datenstruktur aus mehreren Teilen zusammengesetzt
- Zeit-Update wird in ISR sequentiell bearbeitet → Überläufe auf die 5 Stellen können inkonsistente Zwischenwerte ergeben



Möglichkeit von inkonsistenten Resultaten
→ aus 09:59:59 + 1 sec entsteht 09:00:00



mögliche Abhilfe:

- Interrupt sperren (Befehl CLI)
- Kopie für Druckprogramm
- Interrupt wieder freigeben
- Ausdrucken der Kopie

→ Multitasking-Problem

12.12 Zusammenfassung Interrupt

- Abrupte Unterbrechung des Programmablaufs durch Ereignis
  - synchron → SW Interrupt: Traps und Exceptions
  - asynchron → externer HW Interrupt
- 8-bit Interrupt-Nummer als Index für Interrupt-Vektor-Tabelle
- Ablauf Interrupt
  - Beenden der momentanen Instruktion (nur bei HW Interrupt)
  - Flags und Rücksprungadresse auf Stack
  - Sprung an die Adresse aus der Interrupt-Vektor-Tabelle
  - Ausführung der ISR
  - Rücksprung mit Befehl IRET
- Initialisieren des Interrupt Systems
  - Interrupt Vektor setzen
  - Interrupt Control Register initialisieren
  - Interrupt freigeben: STI