Befehle des Intel 8086

Inhalt

0. Einleitung

Diese Zusammenstellung enthält die Befehle des Intel 8086 mit einer Beschreibung ihrer grundlegenden Funktionsweise. Diese Befehle können bei der Arbeit mit vielen Assemblerprogrammen (mit evt. kleineren Abweichungen) benutzt werden.
Besondere Hinweise:

- Verwendung von 16-bit Befehlen und 16-bit Registern
- Real-Mode-Adressierung
- keine Betrachtung von Ausnahme-Generierungen
- keine Befehle des mathematischen Koprozessors betrachtet

Die Befehle sind (willkürlich) in Gruppen eingeteilt, die Einteilung folgte dem Grundprinzip eines intuitiven Grundverständnisses in die Gruppen wie aus dem Inhaltsverzeichnis ersichtlich. Die Beschreibung je Befehl erfolgt in den Abschnitten:

- Operation
- Flags
- Mnemonik

Operation

Dieser Abschnitt enthält eine verbale Beschreibung der durch den Befehl veranlassten Operation(en). Weiterhin ist eine funktionelle Beschreibung des Befehls in einer Pseudocode-Notation gegeben. In dieser werden solche auch in anderen Programmiersprachen verwendete Sprachkonstrukte, wie `IF... THEN... ELSE... ELSIF... THEN... ENDIF`, `WHILE... LOOP... ENDLOOP` etc. benutzt. Der Operator `:=` ist eine Wertezuweisung, während `=` ein Vergleich bedeutet. Andere Zeichen sind selbsterklärend, wie `>, >=, <, <=, !=`. Spezielle Funktionen sind verbal in den Pseudocode gesetzt (z.B. `High byte of(...)`; deren Bezeichnungen sind auch selbsterklärend. `dummy` bezeichnet einen Wert, der nicht gespeichert wird.

Flags

Die Tabelle enthält Informationen über die von dem Befehl “angefassten” Statusflags. Deren Bitposition im wortbreiten Segmentregister *Flags* sowie deren Bedeutung ist in der folgenden Tabelle aufgezeigt. An höheren Bitpositionen befinden sich das Interrupt-Flag (s. INT-Befehle), das Direction-Flag (s. String-Befehle) und das Trap-Flag (für Debugger).

Bit	Name	Funktion
0	CF	Carry Flag: 1 bei Überlauf aus dem Top-Bit heraus bzw. Borgen in das Top-Bit hinein, sonst 0
2	PF	Parity Flag: ist 1 bei gerader Anzahl von Einsen im Low-Byte des Ergebnisses, sonst 0
4	AF	Adjust/Auxiliary Carry Flag: 1 bei Überlauf/Borgen des Low-Nibble (niederwertige 4 Bit) von AL, sonst 0; benutzt bei Dezimalarithmetik (BCD-Ziffern)
6	ZF	Zero Flag: 1 wenn Resultat 0, sonst 0
7	SF	Sign Flag: ist gleich dem Top-Bit des Ergebnisses (0 ≙positiv, 1 ≙negativ)
11	OF	Overflow Flag: 1 wenn das Ergebnis eine zu große (signed) positive Zahl oder eine zu kleine (signed) negative Zahl für den Zieloperanden ist, sonst 0

Die Symbole bedeuten:

T	Befehl testet Flag (und führt Operation abhängig davon aus)
M	Befehl modifiziert Flag (abhängig von Operanden)
0	Flag wird zu 0
1	Flag wird zu 1
?	Flagwert ist undefiniert
–	Flag wird nicht gelesen/geschrieben

Mnemonik

Zeigt (wichtige) Syntaxvarianten und deren Abarbeitungsdauer in 8086-CPU-Taktzyklen. Diese sind auf gegenwärtige PC's nicht übertragbar, vermitteln aber einen Eindruck von der Komplexität der Befehle und können als Vergleichswerte für die Abarbeitungsdauer der Befehle untereinander gesehen werden.
`EA` ist die Anzahl an Taktzyklen zur Berechnung der Effektiven Adresse bei Speicheroperanden (ergibt sich aus den Additionen der Einzelkomponenten des Adressoperanden, s.u. "Speicheradressierung").
Die Abkürzungen für die Operanden bedeuten:

- reg8: eines der acht byte(8-Bit)-breiten Arbeitsregister AL, CL, DL, BL, AH, CH, DH, BH
- reg16 eines der acht Wort(16-Bit)-breiten Arbeitsregister AX, CX, DX, BX, SP, BP, SI, DI
- reg: entweder ein 8-Bit- oder ein 16-Bit-Arbeitsregister, in der Größe gleich dem anderen Operanden des Befehls
- mem8: die Adresse zu einem Bytewert im Hauptspeicher (im angegebenen Segment)
- mem16: die Adresse zu einem Wortwert im Hauptspeicher (im angegebenen Segment)
- mem: die Adresse zu einem Byte- oder Wortwert im Hauptspeicher (im angegebenen Segment) – gleiche Größe wie anderer Operand des Befehls

- direkt8: bytegroßer Direktwert, Zahl steht unmittelbar im Befehlscode
- direkt16: wortgroßer Direktwert, Zahl steht unmittelbar im Befehlscode
- direkt: byte- oder wortgroßer Direktwert, gleiche Größe wie anderer Operand des Befehls
- rel8: Bytewert, wird als relative signed Adresse (-2^7 bis $+2^7-1$) interpretiert; die Zieladresse der Operation ist IP+rel8
- rel16: Wortwert, wird als relative signed Adresse (-2^{15} bis $+2^{15}-1$) interpretiert; die Zieladresse der Operation ist IP+rel16
- ptr32: 32-Bit Direktwert im Op-code; wird als vollständiger Adresspointer (FAR-Pointer) zu einer Speicherstelle interpretiert: das höherwertige Wort (= Segmentbasis) beschreibt das angegebene Segmentregister, das niederwertige Wort (= Offset) das Adressregister
- memptr32: die Adresse zu einem 32-Bit-Wert im Hauptspeicher; dieser 32-Bit Wert wird als vollständiger Adresspointer (FAR-Pointer) zu einer Speicherstelle interpretiert: das höherwertige Wort (= Segmentbasis) beschreibt das angegebene Segmentregister, das niederwertige Wort (= Offset) das Adressregister
- sreg: eines der (wortbreiten) Segmentregister CS, DS, SS, ES,
- memoffs: ein Direktwert, der die effektive Adresse einer Speicherstelle im aktuellen Segment angibt; Varianten bei MOV-Befehl Flags

Anmerkungen

Speicheradressierung

Adressoperanden – Befehlsoperanden, die eine Speicherstelle im Hauptspeicher referenzieren – können aus drei Teilen bestehen:

- Basis: eines der Register BX oder BP
- Index: eines der Register SI oder DI
- Displacement disp: ein 16-Bit Direktwert, der eine feste Verschiebung zur Adresse, die sich aus den anderen beteiligten Adressoperanden ergibt, angibt

Besteht ein Adressoperand nur aus einem Register, dann kann dies eines der 8 Wort-Arbeitsregister AX, CX, DX, BX, SP, BP, SI, DI sein; SP wird hier typischerweise nur für Zugriffe im Stacksegment SS benutzt. Folgende Formen sind beispielsweise möglich:

```
[AX]
[BX+disp]    = [BX]disp
[BP+DI]      = [BP][DI]
[BX+SI+disp] = [BX][SI]disp
...
```

Die sich aus der o.g. Berechnung ergebende Adresse ist die *effektive Adresse*. Der Speicherzugriff erfolgt dann mit der (*linearen*) Adresse = 16*Segmentregister + effektive Adresse. Segmentregister ist meist DS. Der Adressoperand `[SP]` referenziert das Stacksegment SS. Für die meisten Befehle (außer wenn anders gekennzeichnet) können Adressoperanden mittels *Segmentoverride* andere Segmente referenzieren, z.B. `ES:[DI]`.

Byte Ordering

Der Hauptspeicher Intel-basierter Rechnersysteme ist Byte-organisiert. Dadurch ergibt sich das Problem der Anordnung der einzelnen Byte von Multibyte-Datenstrukturen im Hauptspeicher. Intel folgt dem *Little Endian Byte ordering*. Das bedeutet: Daten werden mit dem niederwertigsten Byte an der angegebenen Adresse zuerst und dann nachfolgend mit steigender Byte-Wertigkeit an den nachfolgend höheren Byte-Adresspositionen gespeichert bis hin zum höchstwertigen Byte an der höchsten Adresse. Z.B.: `MOV [SI],AX` speichert das niederwertige Byte von AX an der (effektiven) Adresse SI und das höherwertige Byte von AX an der (effektiven) Adresse SI+1.

1. Datentransport, Speicher- und Stackbefehle

MOV dest,src – Move Data

Operation: Der Quelloperand wird zum Zieloperanden übertragen.

```
DEST := SRC;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`MOV akku,memoffs`	10
`MOV memoffs,akku`	10
`MOV reg,direkt`	4
`MOV reg,reg`	2
`MOV reg,mem`	8+EA
`MOV reg16,sreg`	2
`MOV mem,direkt`	10+EA
`MOV mem,reg`	9+EA
`MOV mem16,sreg`	9+EA
`MOV sreg,reg16`	2
`MOV sreg,mem16`	8+EA

PUSH src – Push Operand onto the Stack

Operation: PUSH legt den Operanden auf der Stackspitze ab. Laut Little Endian Byte Ordering (s. Einleitung) liegt nach dem Befehl das Low-Byte von SRC bei `[SS:SP]` und das High-Byte bei `[SS:SP+1]` gespeichert (jeweils neuer SP betrachtet).

```
SP := SP - 2;
[SS:SP] := SRC;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`PUSH reg16`	11
	`PUSH mem16`	16+EA
	`PUSH sreg`	10

PUSHF – Push the flags register

Operation: PUSHF legt das Flagregister auf dem Stack ab; das High-Byte des Flag-Registers liegt dann bei `[SS:SP+1]` und das Low-Byte bei `[SS:SP]` (neuer SP betrachtet).

```
SP := SP - 2;
[SS:SP] := Flags;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`PUSHF`	10

POP dest – Pop from stack

Operation: POP holt den Wert von der aktuellen Stackspitze und legt ihn im Zieloperanden ab. Laut Little Endian Byte Ordering (s. Einleitung) wird dabei das Low-Byte von DEST mit dem Byte bei `[SS:SP]` und das High-Byte mit `[SS:SP+1]` geladen (jeweils alter SP betrachtet).

```
DEST := [SS:SP];
SP := SP + 2;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`POP reg16`	8
	`POP mem16`	17+EA

`POP sreg`	8

POPF – Pop the flags register

Operation: POPF schreibt den Wert an der aktuellen Stackspitze in das Flagregister ein: das Byte bei `[SS:SP]` in das Low-Byte des Flag-Registers und das Byte bei `[SS:SP+1]` in das High-Byte (alter SP betrachtet; s.a. Byte-Ordering in der Einleitung).

```
Flags := [SS:SP];
SP := SP + 2;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`POPF`	8

XCHG dest,src – Exchange memory and register

Operation: XCHG vertauscht die angegebenen Operanden. Sendet LOCK-Signal aus, um einen Speicherzugriff durch andere Systemeinheiten (zwischenzeitliches Ändern der beteiligten Operanden) während der Befehlsabarbeitung zu verhindern.

```
temp := DEST;
DEST := SRC;
SRC := temp;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`XCHG AX,reg16`	3
`XCHG reg16,AX`	3
`XCHG reg,reg`	4
`XCHG reg,mem`	17+EA

`XCHG mem,reg`	17+EA

LAHF – Load Flags into AH

Operation: Lädt das niederwertige Byte des Flag-Registers nach AH.

```
AH := SF:ZF:xx:AF:xx:PF:xx:CF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`LAHF`	4

SAHF – Store AH into Flags register

Operation: Lädt den Inhalt von AH in das niederwertige Byte des Flag-Registers.

```
SF:ZF:xx:AF:xx:PF:xx:CF := AH;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`SAHF`	4

LEA dest,src – Load effective address

Operation: Schreibt die effektive Adresse des angegebenen Speicheroperanden (= Abstand *"Offset"* zur Datensegmentanfangsadresse *"Segmentbasis"*) in das angegebene Register.

```
DEST := Adresse(SRC);
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`LEA reg16,mem`	2+EA

LDS/LES dest,src – Load full pointer

Operation: Liest einen 4-Byte Wert (= vollständiger Adresspointer) von der Quelladresse und schreibt ihn in das angegebene Segmentregister:Zielregister-Paar: höherwertiges Wort in das Segmentregister, niederwertiges Wort in das angegebene Zielregister.

```
CASE instruction of
   LDS: DS := [SRC];
        DEST := [SRC+2];
   LES: ES := [SRC];
        DEST := [SRC+2];
ENDCASE;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`LDS reg16,memptr32`	16+EA
`LES reg16,memptr32`	16+EA

XLAT operand /XLATB – Table look-up translation

Operation: Liest den Wert an der angegebenen Stelle aus einer Tabelle. BX ist Adresse des ersten Tabellenelements und AL enthält den Index, multipliziert mit der Bytegröße pro Tabelleneintrag. Die Form mit Befehlsoperand erlaubt ein Segmentoverride, BX und AL werden aber immer benutzt.

```
AL := [BX + AL];
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`XLAT mem8`	11
`XLATB`	11

2. Arithmetische Operationen

ADD dest,src – Add operands

Operation: Addiert die zwei Operanden und schreibt Resultat in den Zieloperanden.

```
DEST := DEST + SRC;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`ADD akku,direkt`	4
`ADD reg,direkt`	4
`ADD reg,reg`	3
`ADD reg,mem`	9+EA
`ADD mem,reg`	16+EA
`ADD mem,direkt`	17+EA
`ADD reg16,direkt8`	4
`ADD mem16,direkt8`	17+EA

ADC dest,src – Add operands with carry

Operation: Addiert die zwei Operanden und berücksichtigt dabei einen Übertrag aus einer vorangegangenen Addition. Das Resultat wird in den Zieloperanden geschrieben.

```
DEST := DEST + SRC + CF;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	TM

Mnemonik:

Syntax	CPU-Takte 8086
`ADC akku,direkt`	4
`ADC reg,direkt`	4

`ADC reg,reg`	3
`ADC reg,mem`	9+EA
`ADC mem,reg`	16+EA
`ADC mem,direkt`	17+EA
`ADC reg16,direkt8`	4
`ADC mem16,direkt8`	17+EA

SUB dest,src – Subtract operands

Operation: Subtrahiert den zweiten Operanden (SRC) vom ersten Operanden (DEST) und schreibt Resultat in den Zieloperanden (DEST).

```
DEST := DEST - SRC;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`SUB akku,direkt`	4
`SUB reg,direkt`	4
`SUB reg,reg`	3
`SUB reg,mem`	9+EA
`SUB mem,reg`	16+EA
`SUB mem,direkt`	17+EA
`SUB reg16,direkt8`	4
`SUB mem16,direkt8`	17+EA

SBB dest,src – Subtract operands with borrow

Operation: Subtrahiert den zweiten Operanden (SRC) vom ersten Operanden (DEST) und berücksichtigt dabei eine geborgte Stelle aus einer vorangegangenen Subtraktion. Das Resultat wird in den Zieloperanden (DEST) geschrieben.

```
DEST := DEST - (SRC + CF);
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	TM

Mnemonik:

Syntax	CPU-Takte 8086
`SBB akku,direkt`	4
`SBB reg,direkt`	4
`SBB reg,reg`	3
`SBB reg,mem`	9+EA
`SBB mem,reg`	16+EA
`SBB mem,direkt`	17+EA
`SBB reg16,direkt8`	4
`SBB mem16,direkt8`	17+EA

INC operand – Increment by one

Operation: Erhöht den Operanden um eins.

```
Operand := Operand + 1;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	–

Mnemonik:

Syntax	CPU-Takte 8086
`INC reg`	3
`INC mem`	15+EA

DEC operand – Decrement by one

Operation: Erniedrigt den Operanden um eins.

```
Operand := Operand - 1;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	–

Mnemonik:

Syntax	CPU-Takte 8086
`DEC reg`	3
`DEC mem`	15+EA

MUL src – Multiply

Operation: Multipliziert die Operanden. Es werden implizit die Register AL, AX und DX benutzt abhängig von der Datenbreite von SRC. `DX:AX` stellt einen 32-Bit-Wert dar, dessen high-word in DX und dessen low-word in AX steht.

```
IF byte-size operation
   THEN AX := AL * src;
ELSE (* word operation *)
   DX:AX := AX * src;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	?	?	?	?	M

Mnemonik:

Syntax	CPU-Takte 8086
`MUL reg8`	70−77
`MUL mem8`	76−83+EA
`MUL reg16`	118−113
`MUL mem16`	124−139+EA

IMUL src – Signed multiply

Operation: Multipliziert die Operanden vorzeichenbehaftet. Es werden implizit die Register AL, AX und DX benutzt abhängig von der Datenbreite von SRC. Pseudocode-Notation siehe oben.

Flags:

O	S	Z	A	P	C
M	?	?	?	?	M

Mnemonik:

Syntax	CPU-Takte 8086

`IMUL reg8`	80−89
`IMUL mem8`	86−104+EA
`IMUL reg16`	128−154
`IMUL mem16`	134−160+EA

DIV src – Divide

Operation: Dividiert die Operanden. Abhängig von der Größe von SRC werden die Register AL, AX und DX implizit benutzt. `DX:AX` stellt einen 32-Bit-Wert dar, dessen high-word in DX und dessen low-word in AX steht.
Passt das Ergebnis nicht in das Zielregister, wird ein INT 0 ausgelöst – wenn SRC = 0 oder Bitbreite von Dividend minus Bitbreite von Divisor (SRC) ist größer als Bitbreite von Ergebnisregister.

```
IF byte-size operation THEN
   temp := AX / src;
   IF temp does not fit in AL
      THEN Interrupt 0;
   ELSE
      AL := temp;
      AH := AX MOD src;
   ENDIF;
ELSE (* word operation *)
   temp := DX:AX / src;
   IF temp does not fit in AX
      THEN Interrupt 0;
   ELSE
      AX := temp;
      DX := DX:AX MOD src;
   ENDIF;
ENDIF;
```

Flags:

O	S	Z	A	P	C
?	?	?	?	?	?

Mnemonik:

Syntax	CPU-Takte 8086
`DIV reg8`	80
`DIV mem8`	86+EA
`DIV reg16`	144
`DIV mem16`	154+EA

IDIV src – Signed Divide

Operation: Dividiert die Operanden vorzeichenbehaftet. Abhängig von der Größe von SRC werden die Register AL, AX und DX implizit benutzt.
Passt das Ergebnis nicht in das Zielregister, wird ein INT 0 ausgelöst – wenn SRC = 0 oder Bitbreite von Dividend − Bitbreite von Divisor (SRC) ist größer als Bitbreite von Ergebnisregister.
Pseudocode-Notation siehe oben.

Flags:

O	S	Z	A	P	C
?	?	?	?	?	?

Mnemonik:

Syntax	CPU-Takte 8086
`IDIV reg8`	101−112
`IDIV mem8`	107−118+EA
`IDIV reg16`	165−184
`IDIV mem16`	171−190+EA

NEG operand – Negate

Operation: Negiert den Operanden im Zweierkomplement, d.h. subtrahiert ihn von 0.

```
Operand := -Operand;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`NEG reg`	3
`NEG mem`	16+EA

3. BCD-Korrekturbefehle

AAA – ASCII Adjust after Addition

Operation: Korrigiert das Resultat der Addition zweier BCD-Ziffern, welches in AL steht (in Binärform), zu einer BCD-Ziffer. Ein nachfolgendes `OR AL,30H` wandelt die Dezimalziffer in AL in den entsprechenden ASCII-Code um. Vergleiche auch AAS, DAA.

```
IF (LowNibble(AL) > 9) OR (AF = 1) THEN
   AL := LowNibble(AL + 6);
   AH := AH + 1;
   AF := 1;
   CF := 1;
ELSE
   CF := 0;
   AF := 0;
ENDIF;
```

Flags:

O	S	Z	A	P	C
?	?	?	TM	?	M

Mnemonik:

Syntax	CPU-Takte 8086
`AAA`	8

AAD – ASCII Adjust AX before Division

Operation: Wandelt eine 2-stellige ungepackte BCD-Zahl in AX (AH und AL halten jeweils eine Ziffer in Binärform) in ihr Binäräquivalent um, was nach AL geschrieben wird (AH wird zu 0). Wichtig für eine nachfolgende Operation, die nur mit Binärwerten möglich ist, z.B. für eine nachfolgende Division. Vergleiche auch AAM.

```
AL := 10*AH + AL;
AH := 0;
```

Flags:

O	S	Z	A	P	C
?	M	M	?	M	?

Mnemonik:

Syntax	CPU-Takte 8086

`AAD`	60

AAM – ASCII Adjust AX after Multiply

Operation: Korrigiert das Byteresultat der Multiplikation zweier ungepackter BCD-Ziffern, welches in AX steht (in Binärform, AH ist 0), zu einer 2-stelligen ungepackten BCD-Zahl, deren Ziffern in AH und in AL stehen.
Vergleiche auch AAD.

```
AH := AL / 10;
AL := AL mod 10;
```

Flags:

O	S	Z	A	P	C
?	M	M	?	M	?

Mnemonik:

Syntax	CPU-Takte 8086
`AAM`	83

AAS – ASCII Adjust AL after Subtraction

Operation: Korrigiert das Resultat der Subtraktion zweier BCD-Ziffern, welches in AL steht (in Binärform), zu einer BCD-Ziffer. Ein nachfolgendes `OR AL,30H` wandelt die Dezimalziffer in AL in den entsprechenden ASCII-Code um.
Vergleiche auch AAA.

```
IF (LowNibble(AL) > 9) OR (AF = 1) THEN
   AL := LowNibble(AL - 6);
   AH := AH - 1;
   AF := 1;
   CF := 1;
ELSE
   CF := 0;
   AF := 0;
ENDIF;
```

Flags:

O	S	Z	A	P	C
?	?	?	TM	?	M

Mnemonik:	Syntax	CPU-Takte 8086
	`AAS`	8

DAA – Decimal Adjust AL after Addition

Operation: Korrigiert das Resultat der Addition zweier 2-stelliger gepackter BCD-Zahlen, welches in AL steht (in Binärform), zu einer 2-stelligen gepackten BCD-Zahl, deren Ziffern in AH und in AL stehen.
Vergleiche auch DAS, AAA.

```
IF (LowNibble(AL) > 9) OR (AF = 1) THEN
   AL := AL + 6;
   AF := 1;
ELSE
   AF := 0;
ELSIF (HighNibble(AL) > 9) OR (CR = 1) THEN
   Al := AL + 60H;
   CF := 1;
ELSE
   CF := 0;
ENDIF;
```

Flags:

O	S	Z	A	P	C
?	M	M	TM	M	TM

Mnemonik:	Syntax	CPU-Takte 8086
	`DAA`	4

DAS – Decimal Adjust AL after Subtraction

Operation: Korrigiert das Resultat der Subtraktion zweier 2-stelliger gepackter BCD-Zahlen, welches in AL steht (in Binärform), zu einer 2-stelligen gepackten BCD-Zahl deren Ziffern in AH und in AL stehen.
Vergleiche auch DAA, AAS.

```
IF (LowNibble(AL) > 9) OR (AF = 1) THEN
   AL := AL - 6;
```

```
   AF := 1;
ELSE
   AF := 0;
ELSIF (HighNibble(AL) > 9) OR (CR = 1) THEN
   Al := AL - 60H;
   CF := 1;
ELSE
   CF := 0;
ENDIF;
```

Flags:

O	S	Z	A	P	C
?	M	M	TM	M	TM

Mnemonik:

Syntax	CPU-Takte 8086
`DAS`	4

4. Logische/Vergleichsoperationen

CMP dest,src – Compare

Operation: Vergleicht die Operanden und setzt die Flags entsprechend. Das Ergebnis wird verworfen.

```
dummy := DEST - SRC; (* skip result, just set the flags *)
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`CMP akku,direkt`	4
`CMP reg,direkt`	4
`CMP reg,reg`	3
`CMP reg,mem`	9+EA
`CMP mem,direkt`	10+EA
`CMP mem,reg`	9+EA
`CMP reg16,direkt8`	4
`CMP mem16,direkt8`	10+EA

AND dest,src – Logical And

Operation: Verknüpft die Operanden logisch-Und.

```
DEST := SRC AND DEST;
CF := 0;
OF := 0;
```

Flags:

O	S	Z	A	P	C
0	M	M	?	M	0

Mnemonik:

Syntax	CPU-Takte 8086
`AND akku,direkt`	4

`AND reg,reg`	3
`AND reg,mem`	9+EA
`AND reg,direkt`	4
`AND mem,direkt`	17+EA
`AND mem,reg`	16+EA
`AND reg16,direkt8`	4
`AND mem16,direkt8`	17+EA

OR dest,src – Logical inclusive Or

Operation: Verknüpft die Operanden inklusiv-Oder.

```
DEST := SRC OR DEST;
CF := 0;
OF := 0;
```

Flags:

O	S	Z	A	P	C
0	M	M	?	M	0

Mnemonik:

Syntax	CPU-Takte 8086
`OR akku,direkt`	4
`OR reg,reg`	3
`OR reg,mem`	9+EA
`OR reg,direkt`	4
`OR mem,reg`	16+EA
`OR mem,direkt`	17+EA

NOT operand – One's complement negation

Operation: Berechnet das Einerkomplement (Bit-weises negieren) des Operanden.

```
operand := NOT operand;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`NOT reg`	3
	`NOT mem`	16+EA

XOR dest,src – Exclusive Or

Operation: Verknüpft die Operanden exklusiv-Oder.

```
DEST := SRC XOR DEST;
CF := 0;
OF := 0;
```

Flags:

O	S	Z	A	P	C
0	M	M	?	M	0

Mnemonik:	Syntax	CPU-Takte 8086
	`XOR akku,direkt`	4
	`XOR reg,reg`	3
	`XOR reg,mem`	9+EA
	`XOR reg,direkt`	4
	`XOR mem,reg`	16+EA
	`XOR mem,direkt`	17+EA

TEST dest,src – AND Compare

Operation: Verknüpft die Operanden logisch-Und und setzt die Flags entsprechend. Das Ergebnis wird verworfen.

```
dummy := SRC AND DEST; (* skip result, just set the flags *)
CF := 0;
OF := 0;
```

Flags:

O	S	Z	A	P	C
0	M	M	?	M	0

Mnemonik:	Syntax	CPU-Takte 8086

`TEST akku,direkt`	4
`TEST reg,reg`	3
`TEST reg,direkt`	5
`TEST mem,reg`	9+EA
`TEST mem,direkt`	11+EA

5. Ein-/Ausgabebefehle

IN dest,src – Input from port

Operation: IN liest einen Wert von der angegebenen Portadresse ein.

```
DEST := [SRC];
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`IN AL,imm8`	10
`IN AL,DX`	8
`IN AX,imm8`	10
`IN AX,DX`	8

OUT dest,src – Output to port

Operation: OUT gibt den Akkumulator auf die angegebene Portadresse aus.

```
[DEST] := SRC;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`OUT imm8,AL`	10
`OUT DX,AL`	8
`OUT imm8,AX`	10
`OUT DX,AX`	8

6. String-/Blockbefehle

CMPS operand /CMPSB/CMPSW – Compare String (Byte/Word type)

Operation: Vergleicht Element der Quellzeichenkette in DS:[SI] mit Element der Zielzeichenkette in ES:[DI]. SI und DI werden automatisch erhöht/erniedrigt für eine evt. Befehlswiederholung. Die Flags werden entsprechend gesetzt. Die Form mit Befehlsoperanden erlaubt ein Segmentoverride für den Quelloperanden, SI und DI werden aber immer benutzt. Der Befehlsoperand legt die Datenbreite fest.

```
IF byte type of instruction THEN
   dummy:= [SI] - [DI]; (* byte comparison *)
   IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1; ENDIF;
ELSE (* word type of instruction *)
   dummy:= [SI] - [DI]; (* word comparison *)
   IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2; ENDIF;
ENDIF;
SI := SI + IncDec;
DI := DI + IncDec;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`CMPS mem,mem`	22
`CMPSB`	22
`CMPSW`	22

MOVS operand /MOVSB/MOVSW – Move String (Byte/Word type)

Operation: Schreibt Element der Quellzeichenkette in DS:[SI] an die Adresse der Zielzeichenkette in ES:[DI]. SI und DI werden automatisch erhöht/erniedrigt für eine evt. Befehlswiederholung. Die Flags werden entsprechend gesetzt. Die Form mit Befehlsoperanden erlaubt ein Segmentoverride für den Quelloperanden, SI und DI werden aber immer benutzt. Der Befehlsoperand legt die Datenbreite fest.

```
IF byte type of instruction THEN
   [DI] := [SI]; (* byte assignment *)
   IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1; ENDIF;
ELSE (* word type of instruction *)
   [DI] := [SI]; (* word assignment *)
   IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2; ENDIF;
ENDIF;
SI := SI + IncDec;
DI := DI + IncDec;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`MOVS mem,mem`	18
`MOVSB`	18
`MOVSW`	18

SCAS operand /SCASB/SCASW – Compare String Data (SCAn String [Byte/Word type])

Operation: Vergleicht Akkumulator mit Element der Zielzeichenkette in ES:[DI]. DI wird automatisch erhöht/erniedrigt für eine evt. Befehlswiederholung. Die Flags werden entsprechend gesetzt. Bei der Variante mit Befehlsoperanden legt jener die Datenbreite fest.

```
IF byte type of instruction THEN
   dummy := AL - [DI]; (* Compare byte in AL and dest *)
   IF DF = 0 THEN IndDec := 1 ELSE IncDec := -1; ENDIF;
ELSE (* word type of instruction *)
   dummy := AX - [DI]; (* compare word in AL and dest *)
   IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2; ENDIF;
ENDIF;
DI := DI + IncDec;
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`SCAS mem`	15
`SCASB`	15

`SCASW`	15

LODS operand /LODSB/LODSW – Load String (Byte/Word type)

Operation: Lädt Element der Quellzeichenkette in DS:[SI] in das Akkumulatorregister. SI wird automatisch erhöht/erniedrigt für eine evt. Befehlswiederholung. Die Flags werden entsprechend gesetzt. Die Form mit Befehlsoperand erlaubt ein Segmentoverride für den Quelloperanden, SI wird aber immer benutzt. Der Befehlsoperand legt die Datenbreite fest.

```
IF byte type of instruction THEN
   AL := [SI]; (* byte load *)
   IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1; ENDIF;
ELSE (* word type of instruction *)
   AX := [SI]; (* word load *)
   IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2; ENDIF;
ENDIF;
SI := SI + IncDec;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`LODS mem`	12
`LODSB`	12
`LODSW`	12

STOS operand /STOSB/STOSW – Store String (Byte/Word type)

Operation: Lädt das Akkumulatorregister in das Element der Zielzeichenkette in ES:[DI]. DI wird automatisch erhöht/erniedrigt für eine evt. Befehlswiederholung. Die Flags werden entsprechend gesetzt. Bei der Variante mit Befehlsoperanden legt jener die Datenbreite fest.

```
IF byte type of instruction THEN
   [DI] := AL; (* byte load *)
   IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1; ENDIF;
ELSE (* word type of instruction *)
```

```
   [DI] := AX; (* word load *)
IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2; ENDIF;
ENDIF;
DI := DI + IncDec;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`STOS mem`	11
`STOSB`	11
`STOSW`	11

REP/REPE/REPZ/REPNE/REPNZ (Prefix Byte) – Repeat string operation while Equal/Zero or Not Equal/Not Zero

Operation: Wiederholt die angegebene Stringoperation CX-mal bzw. bei CMPS- und SCAS-Befehlen solange das Zero-Flag 1 (bei REPE; REPZ ist Synonym für REPE) bzw. 0 ist (bei REPNE; REPNZ ist Synonym für REPNE).
Wichtig ist die bei jedem REP-Befehl folgende Reihenfolge der Aktionen:

1. Test CX auf 0, wenn ja → exit REP-Befehl
2. Stringoperation ausführen
3. CX dekrementieren und zurück zu 1.

Vgl. LOOP: gleiche Reihenfolge, aber Beginn bei 2.!

```
WHILE CX != 0 LOOP
   INT-Behandlung;
   Stringoperation einmal;
   CX := CX - 1;
   IF string operation is CMPS or SCAS THEN
      IF (instruction is REPE/REPZ) AND (ZF=0)
        THEN exit WHILE loop;
      ELSIF (instruction is REPNZ or REPNE) AND (ZF=1)
        THEN exit WHILE loop;
      ENDIF;
   ENDIF;
ENDLOOP;
```

Flags:

O	S	Z	A	P	C
–	–	T	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086

REP	2
REPE/REPZ	2
REPNE/REPNZ	2

7. Rotate&Shift

ROL operand1,operand2 – Rotate Left without Carry

Operation: Rotiert die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach links. ROL kopiert das Bit, welches vom höchstwertigen zum niederwertigsten Bit verschoben wird in das CF.

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
   tmphigh := high-order bit of (operand1);
   operand1 := operand1 * 2 + (tmphigh);
   CF := tmphigh;
   temp := temp - 1;
ENDLOOP;
IF COUNT = 1 THEN
   OF := (high-order bit of operand1) XOR CF;
ELSE OF := undefined;
   ENDIF;
```

Flags:

O	S	Z	A	P	C
M	–	–	–	–	M

Mnemonik:

Syntax	CPU-Takte 8086
`ROL reg,1`	2
`ROL mem,1`	15+EA
`ROL reg,CL`	8+4 per Bit
`ROL mem,CL`	(20+4 per Bit)+EA

ROR operand1,operand2 – Rotate Right without Carry

Operation: Rotiert die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach rechts. ROR kopiert das Bit, welches vom niederwertigsten Bit zum höchstwertigen Bit verschoben wird in das CF.

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
```

```
   tmplow := low-order bit of (operand1);
   operand1 := operand1 /2 + (tmplow * 2^(width(operand1)-1));
   CF := tmplow;
   temp := temp - 1;
ENDLOOP;
IF COUNT = 1 THEN
   OF := (high-order bit of operand1) XOR
        (bit next to high-order bit of operand1);
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	–	–	–	–	M

Mnemonik:

Syntax	CPU-Takte 8086
`ROR reg,1`	2
`ROR mem,1`	15+EA
`ROR reg,CL`	8+4 per Bit
`ROR mem,CL`	(20+4 per Bit)+EA

RCL operand1,operand2 – Rotate Left with Carry

Operation: Rotiert die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach links und setzt zwischen das höchstwertige und niederwertigste Bit das Carry-Flag.

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
     tmphigh := high-order bit of (operand1);
       operand1 := operand1 * 2 + (CF);
     CF := tmphigh;
   temp := temp - 1;
ENDLOOP;
IF COUNT = 1 THEN
     OF := (high-order bit of operand1) XOR CF;
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	–	–	–	–	M

Mnemonik:

Syntax	CPU-Takte 8086

`RCL reg,1`	2
`RCL mem,1`	15+EA
`RCL reg,CL`	8+4 per Bit
`RCL mem,CL`	(20+4 per Bit)+EA

RCR operand1,operand2 – Rotate Right with Carry

Operation: Rotiert die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach rechts und setzt zwischen das höchstwertige und niederwertigste Bit das Carry-Flag.

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
   tmplow := low-order bit of (operand1);
   operand1 := operand1 /2 + (CF * 2^(width(operand1)-1));
   CF := tmplow;
   temp := temp - 1;
ENDLOOP;
IF COUNT = 1 THEN
   OF := (high-order bit of operand1) XOR
         (bit next to high-order bit of operand1);
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	–	–	–	–	M

Mnemonik:

Syntax	CPU-Takte 8086
`RCR reg,1`	2
`RCR mem,1`	15+EA
`RCR reg,CL`	8+4 per Bit
`RCR mem,CL`	(20+4 per Bit)+EA

SAL/SHL operand1,operand2 – Shift Left (Arithmetical)

Operation: Verschiebt die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach links. Das dabei herausgeschobene Bit wird in das CF kopiert. SAL und SHL füllen das niederwertigste Bit mit 0 auf.

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
   CF := high-order bit of operand1;
   operand1 := operand1 * 2;
   temp := temp - 1;
ENDLOOP;
   (* Determine overflow for the various instructions *)
IF COUNT = 1 THEN
   OF := (high-order bit of operand1) XOR (CF);
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	M	M	?	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`SAL reg,1`	2
`SAL mem,1`	15+EA
`SHL reg,1`	2
`SHL mem,1`	15+EA
`SAL reg,CL`	8+4 per Bit
`SAL mem,CL`	(20+4 per Bit)+EA
`SHL reg,CL`	8+4 per Bit
`SHL mem,CL`	(20+4 per Bit)+EA

SAR operand1,operand2 – Shift Right Arithmetical

Operation: Verschiebt die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach rechts. Das dabei herausgeschobene Bit wird in das CF kopiert. SAR belässt die höchstwertige Bitstelle auf dem alten Wert (= signed Division, aber mit Runden nach $-\infty$).

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
   CF := low-order bit of operand1;
   tmpsign := sign bit of operand1;
      (* is high-order bit *);
   operand1 := operand1 /2 + (tmpsign * 2^(width(operand1)-1));
```

```
      (* Signed divide, rounding toward negative infinity *)
   temp := temp - 1;
ENDLOOP;
(* Determine overflow for the various instructions *)
IF COUNT = 1 THEN
   OF := 0;
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	M	M	?	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`SAR reg,1`	2
`SAR mem,1`	15+EA
`SAR reg,CL`	8+4 per Bit
`SAR mem,CL`	(20+4 per Bit)+EA

SHR operand1,operand2 – Shift Right

Operation: Verschiebt die einzelnen Bitstellen von Operand1 um 1 oder CL Stellen nach rechts. Das dabei herausgeschobene Bit wird in das CF kopiert. SHR setzt die höchstwertige Bitstelle auf 0 (unsigned Division).

```
(* COUNT is operand2 *)
temp := COUNT;
WHILE (temp != 0) LOOP
   CF := low-order bit of operand1;
   operand1 := operand1 /2; (* Unsigned divide *)
   temp := temp - 1;
ENDLOOP;
(* Determine overflow for the various instructions *)
IF COUNT = 1 THEN
   OF := high-order bit of operand1;
ELSE OF := undefined;
ENDIF;
```

Flags:

O	S	Z	A	P	C
M	M	M	?	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`SHR reg,1`	2

`SHR mem,1`	15+EA
`SHR reg,CL`	8+4 per Bit
`SHR mem,CL`	(20+4 per Bit)+EA

8. Programmverzweigungen

JMP operand – Jump

Operation: Programm springt zur angegebenen Stelle. Adresse kann relativ, indirekt oder für Far-Jmp direkt (4 Byte Pointer) angegeben sein.

```
IF instruction = relative JMP (* operand is rel8/16 *)
  THEN IP := IP + operand;
ELSIF instruction = near indirect JMP (* operand reg/mem16 *)
  THEN IP := [operand];
ELSIF instruction = far JMP THEN (* operand is ptr32 *)
  CS := high word of operand;
  IP := low word of operand;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`JMP rel8`	15
`JMP rel16`	15
`JMP reg16`	11
`JMP mem16`	18+EA
`JMP ptr32`	15

Jcc operand – Jump Conditional

Operation: Programm springt zur angegebenen Stelle (Marke), wenn Bedingung `cc` erfüllt ist. Im Anhang befindet sich eine Tabelle der verfügbaren Sprungbedingungen.
Bedingte Sprünge können nur in das gleiche Segment erfolgen und die Sprungmarke muss von IP−128 bis IP+127 liegen – = SHORT-Jump (IP ist hier der IP für den nachfolgenden Befehl). Bei Sprungzielen in andere Segmente muss die inverse Bedingung abgetestet werden und dann mittels FAR-Jump der Sprung ausgeführt werden.

Zum Beispiel nicht so:

```
        JZ FARLABEL;
```

sondern so schreiben:

```
        JNZ weiter;
        JMP FARLABEL;
     weiter:
```

```
IF cc = 1 THEN
   IP := IP + operand;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`Jcc rel8`	16,4*

* 1.Zahl: Sprung erfolgt (branch taken), 2.Zahl: Sprung erfolgt nicht (branch not taken).

JCXZ operand – Jump if CX equals Zero

Operation: Programm springt zur angegebenen Stelle, wenn CX gleich 0 ist; das ist nützlich bei LOOP-Schleifen, damit diese bei anfangs CX=0 nicht versehentlich 65536-mal abgearbeitet werden (siehe auch LOOP). Sprünge können nur in das gleiche Segment erfolgen (s. Jcc).

```
IF CX = 0 THEN
   IP := IP + operand;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`JCXZ rel8`	18,6*

* 1.Zahl: Sprung erfolgt (branch taken), 2.Zahl: Sprung erfolgt nicht (branch not taken).

LOOP/LOOPE/LOOPZ/LOOPNE/LOOPNZ operand – Loop while Zero or Not Zero

Operation: Schleifenkonstrukt in der Art `DO...WHILE`; die Befehle zwischen der Beginnmarke und dem LOOP-Befehl werden mind. 1-mal ausgeführt. Die Schleife wird durchlaufen, solange CX (nach dem Dekrementieren) ungleich 0 ist und solange das Zero-flag 1 (LOOPE bzw. das Synonym LOOPZ) oder 0 (LOOPNE bzw. das Synonym LOOPNZ) ist.
Beachte: Ist CX anfangs der LOOP-Schleife 0, dann wird diese 65536-mal ausgeführt! Zum Abtesten den Befehl JCXZ verwenden.
Wichtig ist die bei jedem LOOP-Befehl folgende Reihenfolge der Aktionen:

1. Befehle ab Beginn-Marke bis LOOP-Befehl ausführen
2. CX dekrementieren
3. Test CX auf 0, wenn ja → exit LOOP-Schleife

Vgl. REP: gleiche Reihenfolge, aber Beginn bei 3.!

```
Befehle-bis-LOOP-Befehl-ausführen;
CX := CX - 1;
IF instruction != LOOP THEN
   IF (instruction = LOOPE) OR (instruction = LOOPZ) THEN
      BranchCond := (ZF = 1) AND (CX != 0);
   ELSIF (instruction = LOOPNE) OR (instruction = LOOPNZ) THEN
      BranchCond := (ZF = 0) AND (CX != 0);
   ENDIF;
ENDIF;
IF BranchCond THEN
   IP := IP + operand;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`LOOP rel8`	17,5*
`LOOPE/LOOPZ rel8`	18,6*
`LOOPNE/LOOPNZ rel8`	19,5*

* 1.Zahl: Sprung erfolgt (branch taken), 2.Zahl: Sprung erfolgt nicht (branch not taken).

CALL operand – Call procedure

Operation: Ruft die Prozedur an der Stelle, die durch operand gegeben ist auf. Nach Ende der Prozedur durch einen RET-Befehl wird die Programmabarbeitung mit dem Befehl, der dem CALL-Befehl folgt fortgesetzt.
`CALL rel/reg/mem16` sind sog. NEAR-Calls, die in das gleiche Segment erfolgen. Diese überschreiben nur den IP. `CALL ptr/memptr32` (FAR-Calls) rufen Prozeduren in einem anderen Segment auf: Das höherwertige Wort des 4-Byte-Operanden wird in das CS-Register geschrieben (Code-Segment, in dem die Prozedur steht) und das niederwertige Wort in das IP-Register (Stelle, an dem die Prozedur im anderen Segment steht).
CALL sichert den IP des Nachfolgebefehls (und bei FAR-Calls das aktuelle CS-Register) vor dem Überschreiben auf dem Stack.

```
IF rel16 type of call THEN (* near relative call *)
  Push(IP);
  IP := IP + operand;
ELSIF reg/mem16 type of call THEN (* near absolute call *)
  Push(IP);
  IP := operand;
ELSIF memptr32 or ptr32 type of call THEN (* far call *)
  Push(CS);
  Push(IP); (* address of next instruction *)
  IF operand type is memptr32 THEN (* indirect far call *)
    CS := word at [operand +2];
    IP := word at [operand];
  ELSIF operand type is ptr32 THEN (* direct far call *)
    CS := High word of operand;
    IP := Low word of operand;
  ENDIF;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`CALL rel16`	19
`CALL reg16`	16
`CALL mem16`	21+EA
`CALL ptr32`	28
`CALL memptr32`	37+EA

RET operand – Return from procedure

Operation: Rückkehr von Prozedur zum normalen Programmablauf. IP des Befehls nach dem CALL-Befehl zur Prozedur (und bei FAR-Return CS) werden vom Stack geholt. Operand gibt die Zahl an Bytes (sollte gerade sein) an, um die der Stackpointer nach dem Einlesen von IP (und CS) erhöht wird; das ist üblicherweise der Platz auf dem Stack, der von Übergabeparametern an die Prozedur vor deren Aufruf durch `CALL` belegt wurde.

```
IF instruction = near RET THEN
IP := Pop();
ELSE (* instruction = far RET *)
IP := Pop();
CS := Pop();
ENDIF;
IF RET with operand THEN
SP := SP + operand;
ENDIF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`RET`	16*
`RET`	26*
`RET direkt`	20*
`RET direkt`	25*

* 1.&3. Variante für NEAR-Return, 2.&4. Variante für FAR-Return

INT operand / INTO – Call interrupt procedure

Operation: INT ruft die Interruptprozedur mit der Nr. operand. Diese Nummer gibt die Stelle in der 4-Byte-breiten IDT (Interrupt Descriptor Table) an, von der die Ansprungadresse (`selector`=Segmentadresse, `offset`=Adresse der INT-Routine im Segment) gelesen wird. INT ist ein FAR-Call mit zusätzlichem Sichern des Flags-Registers auf dem Stack. INTO ist ein Interrrupt der Nr. 4, der bei OF = 1 erfolgt.

```
Push (FLAGS);
IF := 0; (* Clear interrupt flag *)
TF := 0; (* Clear trap flag *)
Push(CS);
Push(IP);
CS := IDT[Interrupt number * 4].selector;
IP := IDT[Interrupt number * 4].offset;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`INT 3`	52
`INT direkt8`	51
`INTO`	4/53*

* 1.Zahl: kein INT, 2.Zahl: INT-Sprung

IRET/IRETW – Return from INT

Operation: Rückkehr von INT-Routine.

```
IP := Pop();
CS := Pop();
Flags := Pop();
```

Flags:

O	S	Z	A	P	C
M	M	M	M	M	M

Mnemonik:

Syntax	CPU-Takte 8086
`IRET/IRETW`	32

9. Flagbefehle/CPU-Steuerbefehle

CLD – Clear Direction flag

Operation: Setzt das Direction-Flag auf 0. Nachfolgende Stringbefehle werden SI und DI inkrementieren; siehe auch dort.

```
DF := 0;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`CLD`	2

STD – Set Direction flag

Operation: Setzt das Direction-Flag auf 1. Nachfolgende Stringbefehle werden SI und DI dekrementieren; siehe auch dort.

```
DF := 1;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`STD`	2

CLC – Clear Carry flag

Operation: Setzt das Carry-Flag auf 0; kein Übertrag vorhanden.

```
CF := 0;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	0

Mnemonik:

Syntax	CPU-Takte 8086
`CLC`	2

STC – Set Carry flag

Operation: Setzt das Carry-Flag auf 1; Übertrag vorhanden.

```
CF := 1;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	1

Mnemonik:

Syntax	CPU-Takte 8086
`STC`	2

CMC – Complement Carry flag

Operation: Invertiert das Carry-Flag.

```
CF := NOT CF;
```

Flags:

O	S	Z	A	P	C
–	–	–	–	–	TM

Mnemonik:

Syntax	CPU-Takte 8086
`CMC`	2

CLI – Clear Interrupt flag

Operation: Setzt das Interrupt-Flag auf 0. Der Prozessor bearbeitet ab sofort keine Interruptanforderungen.

```
IF := 0;
```

Flags:	O	S	Z	A	P	C
	–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`CLI`	2

STI – Set Interrupt flag

Operation: Setzt das Interrupt-Flag auf 1. Der Prozessor reagiert erst mit Ende des nächsten Befehls auf Interruptanforderungen, wenn jener das Interrupt-Flag nicht wieder auf 0 setzt.

```
IF := 1;
```

Flags:	O	S	Z	A	P	C
	–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`STI`	2

LOCK (prefix byte) – Assert LOCK# signal

Operation: Während des folgenden Befehls sendet die aktive CPU ein Lock-Signal aus. In einer Multiprozessorumgebung kann die betreffende CPU für die Dauer des einen Befehls alleinigen Buszugriff haben.
XCHG sendet immer ein LOCK-Signal aus.

Flags:	O	S	Z	A	P	C
	–	–	–	–	–	–

Mnemonik:	Syntax	CPU-Takte 8086
	`LOCK`	2

NOP – No Operation

Operation: Keine Operation. Nur der IP wird inkrementiert und CPU-Zeit wird "verbraucht". NOP ist ein Alias für `XCHG AX,AX`.

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`NOP`	3

HLT – Halt CPU

Operation: HLT hält die Befehlsabarbeitung an, versetzt die CPU in den HALT-Zustand. Ein INT, NMI oder ein Reset wird die Befehlsabarbeitung wieder fortgesetzt; bei INT oder NMI mit dem nachfolgenden Befehl (dessen IP wurde vor dem Halt in das IP-Register geschrieben).

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`HLT`	2

WAIT – Wait until BUSY# pin is inactive (high)

Operation: Setzt die Befehlsabarbeitung der CPU aus bis das BUSY-Signal auf inaktiv ("high") steht. Dieses Signal wird vom numerischen Koprozessor gesendet. Dieser Befehl ist nur für die Abarbeitung von Koprozessorbefehlen wichtig.

Flags:

O	S	Z	A	P	C
–	–	–	–	–	–

Mnemonik:

Syntax	CPU-Takte 8086
`WAIT`	4+5n*

* n: Zahl der Wiederholungen des WAIT-Befehls.

Anhang Bedingungscodes

Anmerkung:

Die Tabelle enthält mögliche Kombinationen von Flags "Bedingungscodes", die durch einige Befehle getestet werden (Jcc, REPNE, LOOPZ etc.).

Die Begriffe "above" und "below" beziehen sich auf Werte, die als vorzeichenlose Zahlen (unsigned) interpretiert werden – Werte mit höchstwertigem Bit gleich 1 sind größer als Werte mit höchstwertigem Bit gleich 0. Nur CF und ZF und weder SF noch OF werden getestet.
Die Begriffe "greater" und "less" dagegen beziehen sich auf Werte, die als vorzeichenbehaftete Zahlen ("signed") interpretierte werden – Werte mit höchstwertigem Bit gleich 1 sind kleiner (negativ) als Werte mit höchstwertigem Bit gleich 0 (positiv). Das CF wird nicht getestet, sondern neben dem ZF nur das SF und das OF.

Mnemonik	Bedeutung	Getestete Bedingung
E/Z	Equal / Zero	ZF = 1
NE/NZ	Not equal / Not zero	ZF = 0
B/NAE	Below / Neither above nor equal	CF = 1
NB/AE	Not below / Above or equal	CF = 0
BE/NA	Below or equal / Not above	(CF or ZF) = 1
NBE/A	Neither below nor equal / Above	(CF or ZF) = 0
L/NGE	Less / Neither greater nor equal	(SF xor OF) = 1
NL/GE	Not less / Greater or equal	(SF xor OF) = 0
LE/NG	Less or equal / Not greater	((SF xor OF) or ZF) = 1
NLE/G	Neither less nor equal / Greater	((SF xor OF) or ZF) = 0
O	Overflow	OF = 1
NO	No overflow	OF = 0
S	Sign	SF = 1
NS	No sign	SF = 0
P/PE	Parity / Parity even	PF = 1
NP/PO	No parity / Parity odd	PF = 0