Befehlsübersicht i8086

Prof. Dr.-Ing. Dietrich Ertelt
ertelt@et-inf.fho-emden.de

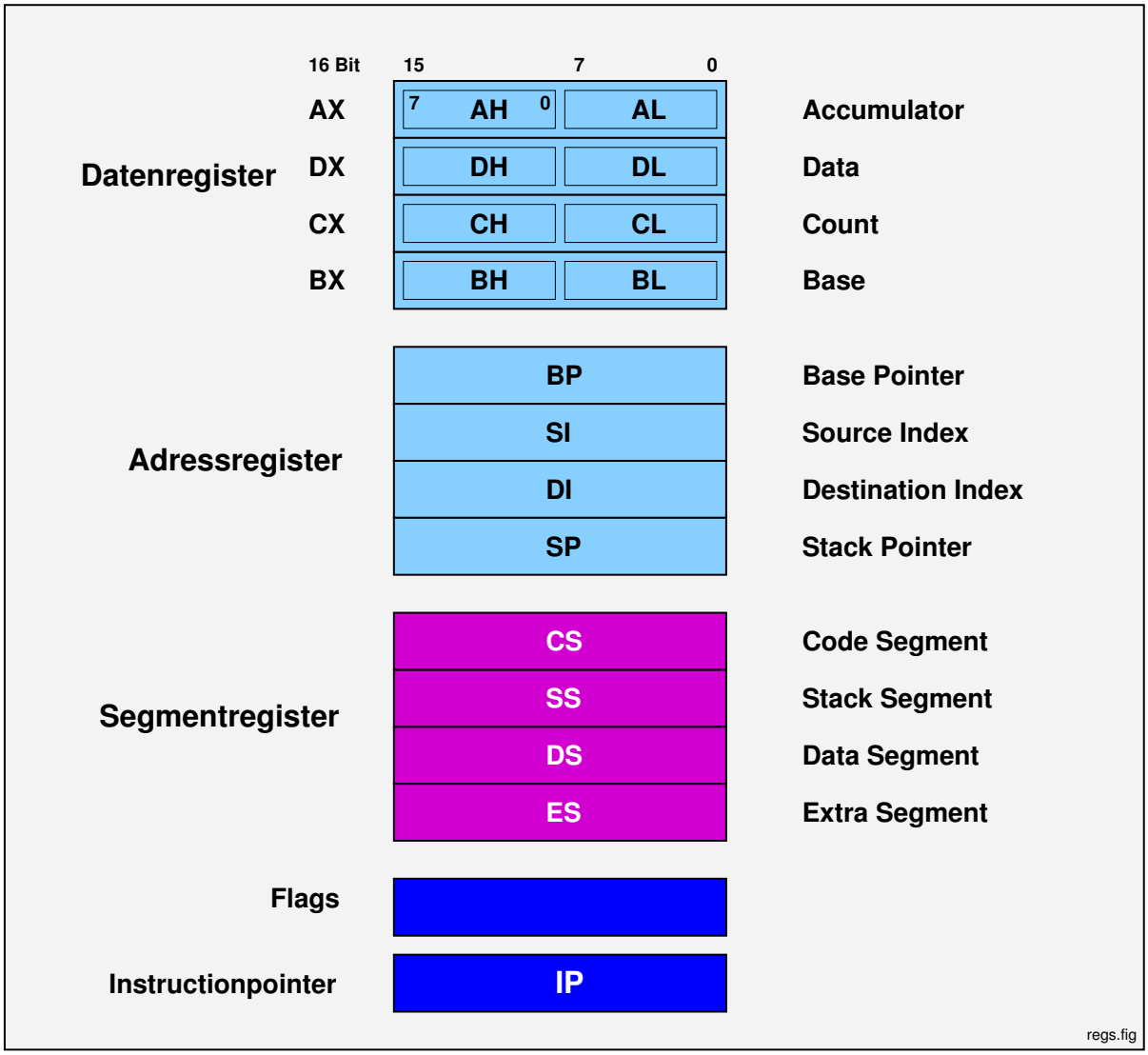
Fachhochschule
Oldenburg **Ostfriesland** Wilhelmshaven
Fachbereich Technik
Elektrotechnik und Informatik

FH Oldenburg **Ostfriesland** Wilhelmshaven

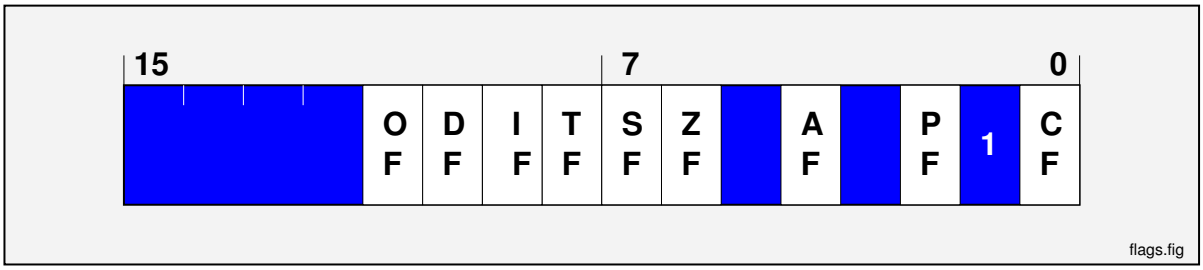
Fachbereich Technik
Elektrotechnik + Informatik

Gruppierung

Architektur



Registersatz 8086



Flagregister des 8086

Carry Statusflag **CF**
Unsigned Überlauf (Übertrag aus dem höchstwertigen Bit)

Zero Statusflag **ZF**
Ergebnis enthält nur 0-Bits

Sign Statusflag **SF**
Kopie des höchstwertigen Ergebnisbit

Overflow Statusflag **OF**
Integer Überlauf (Übertrag entweder **in** oder **aus** MSB)

Parity Statusflag **PF**
Low-Byte des Ergebnis hat gerade Anzahl von Einsen

Auxiliary Statusflag **AF**
Übertrag aus Bit 3 nach Bit 4

Direction Steuerflag **DF**
Adresszählung bei Stringbefehlen

Interrupt Steuerflag **IF**
Maskierung von Interrupts

Trap Steuerflag **TF**
Einzelschrittbetrieb

1. Datentransfer

1.1. CPU ⇔ Speicher MOV

MOV destination,source Move			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
memory,accu accu,memory	3	10	**mov [TOTAL], ax** **mov al, [TMP_R]**
register,register	2	2	**mov ax, cx**
register,memory	2-4	8+ea	**mov bp,[STACK_TOP]** **mov dh, [CHAR1]**
memory,register	2-4	9+ea	**mov [N24], cx** **mov [bp+2],ah**
register,immediate	2-3	4	**mov dx,1024** **mov al, 'Ä'**
memory,immediate	3-6	10+ea	**mov [MSK+bx+si],2ch**
seg-reg,reg16 reg16,seg-reg	2	2	**mov es,cx** **mov bp,ss**
seg-reg,mem16	2-4	8+ea	**mov ds, [SEG_BASE]**
mem16,seg-reg	2-4	9+ea	**mov [SEG_SAVE], cs**
Kopieren des Quelloperanden in den Zieloperand			

Erlaubte Kombination der Operanden	**Destination**	**Source**				
		A	R	S	M	K
	Accumul A	**X**	**X**	**X**	**X**	**X**
	Register R	**X**	**X**	**X**	**X**	**X**
	SegReg[1] S	**X**	**X**	–	**X**	–
	Memory M	**X**	**X**	**X**	–	**X**

[1] CS **nie** Zieloperand!

PUSH

PUSH source Push word onto stack			Flags **ODITSZAPC**
Operands	**Bytes**	**Clocks**	**Example**
register	1	11	**push si**
seg-reg (auch CS)	1	10	**push es**
mem16	2-4	16+ea	**push [PARAM_1]**
16-Bit-Wert auf den Stapelspeicher legen. Dazu wird **vor** dem Transfer SP um 2 verringert.			

POP

POP destination Pop word off stack			Flags **ODITSZAPC**
Operands	**Bytes**	**Clocks**	**Example**
register	1	8	**pop dx**
seg-reg (**nie** CS)	1	8	**pop ds**
mem16	2-4	17+ea	**pop [PARAM_1]**
16-Bit-Wert vom Stapelspeicher holen. **Nach** dem Transfer wird SP um 2 erhöht.			

XCHG

XCHG destination,source Exchange			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
accumulator,reg16	1	3	**xchg ax, bx**[2]
register,register	2	4	**xchg cl, ah**
memory,register register,memory	2-4	17+ea	**xchg [B_TB+bx+si],cl**
Vertauschen der beiden Operanden. Operandenkombinationen siehe Arithmetik (Ausnahme Immediate: beide Operanden werden geändert).			

XLAT

XLAT [source-table] Translate			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	11	**xlatb** (NASM) **xlat** (MASM)
[[segreg:]table]			**xlat [es:CHAR_TAB]**
AL-Inhalt wird als Index einer Tabelle interpretiert, deren Startadresse in BX steht. Er wird durch den zu diesem Index gehörenden Tabellenwert ersetzt (Umcodierung). Eine Operandenangabe dient speziell der Überschreibung des Segmentregisters (Default DS:).			

[2] xchg ax,ax dient als **NOP**-Befehl

1.2. CPU ⇔ Peripherie

IN

IN accumulator,port Input byte or word			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
accumulator,immed8	2	10	**in al, 60h** ;0..255 **in ax, 24** ;Port 24&25
accumulator,DX	1	8	**in al, dx** ;0..64k
Einlesen eines Byte oder Word in den Accu vom Port mit der direkt oder in DX spezifizierten Adresse. Direkt sind nur die ersten 256 Ports erreichbar. Wird AX benutzt, so wird von 2 aufeinanderfolgenden Portadressen gelesen.			

OUT

OUT port,accumulator Output byte or word			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
immed8,accumulator	2	10	**out 60h, al** ;0..255 **out 24, ax** ;Port 24&25
DX,accumulator	1	8	**out dx, al** ;0..64k
Ausgeben eines Byte oder Word aus dem Accu an das Port mit der direkt oder in DX spezifizierten Adresse. Direkt sind nur die ersten 256 Ports erreichbar. Wird AX benutzt, so wird auf 2 aufeinanderfolgende Portadressen geschrieben.			

1.3. Sonstige Transfers

LEA

LEA destination,source Load effective address			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
reg16,mem16	2-4	2+ea	**lea bx, [bp+di]** **lea si, VARX** ;(MASM) **lea si,[VARX]** ;(NASM)
Berechnet und speichert die durch mem16 definierte effektive Adresse in das Zielregister.			

LDS LES

LDS destination,source **LES destination,source** Load pointer using DS (ES)			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
reg16,mem16	2-4	16+ea	;DW_VAR: 1a 2b 3c 4d **lds bx, [DW_VAR]** ;BX=2b1ah,DS=4d3ch
reg16,mem16	2-4	16+ea	**les di, [si]**
Der durch mem16 adressierte 4-Byte-Wert wird als DS:reg16 bzw. ES:reg16 geladen.			

LAHF

LAHF (no operands) Load AH from flags			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	4	**lahf**
Niederwertiger Teil des Flagregisters wird nach AH kopiert (CF, PF, AF, ZF und SF).			

SAHF

SAHF (no operands) Store AH into flags			Flags **O D I T S Z A P C** * * * * *
Operands	**Bytes**	**Clocks**	**Example**
	1	4	**sahf**
Niederwertiger Teil des Flagregisters wird von AH kopiert[3](CF, PF, AF,ZF und SF).			

[3] 1. Ausnahmefall der Flagbeeinflussung durch Transfer

PUSHF

PUSHF (no operands) Push flags onto stack			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	10	**pushf**
Flagregister auf den Stack legen.			

POPF

POPF (no operands) Pop flags from stack			Flags **O D I T S Z A P C** * * * * * * * * *
Operands	**Bytes**	**Clocks**	**Example**
	1	8	**popf**
Flagregister vom Stack laden[4].			

[4] 2. Ausnahmefall der Flagbeeinflussung durch Transfer

2. Programmflusssteuerung

2.1. Unbedingte Programmsteuerung

JMP

JMP target Jump (unconditioned)			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
short-label	2	15	**jmp KURZ** ;-128..127
near-label	3	15	**jmp IM_SEGM**
far-label	5	15	**jmp NEUSEGM**
memptr16	2-4	18+ea	**jmp [W_TAB+bx]**
regptr16	2	11	**jmp ax**
memptr32	2-4	24	**jmp dword [bx];(NASM)** **jmp [WEIT]** ;DD-Obj.
Unbedingter Sprung zur angegebenen Programmadresse. Dies geschieht durch Laden von IP (und CS bei far jmp) mit den Operandwerten des Befehls			

CALL

CALL target Call a procedure			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
near-proc	3	19	**call IN_SEG_UP**
far-proc	5	28	**call OUT_SEG_UP**
memptr16	2-4	21+ea	;UP_ADR=offset UP1 **call [UP_ADR]**
regptr16	2	16	mov ax, UP1 ;(NASM) **call ax**
memptr32	2-4	24	extrn UP2:far UP_TB dd ?,?,?,? ;UP_TB[8]=offset UP2 ;UP_TB[10]=seg UP2 **call [UP_TB+2*4]**
Aufruf eines Unterprogramms. Dazu wird die Adresse des folgenden Befehls auf den Stack gerettet und dann IP (und CS bei far proc) mit den Operandwerten des Befehls geladen			

RET

RET [pop-value] Return from procedure			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
(near-proc)	1	16	**ret**
immed16 (near-proc)	3	20	**ret 4**
(far-proc)	1	26	**ret**
immed16 (far-proc)	3	25	**ret 2**

Rücksprung aus einem Unterprogramm in das aufrufende Programm. Dazu wird IP (und CS bei far proc) vom Stack geladen. Die optionale Angabe einer Konstanten dient dem Abgleich des Stack, wenn er für die Argumentübergabe vom Hauptprogramm an das Unterprogramm benutzt wurde.

IRET

IRET (no operands) Interrupt return			Flags **O D I T S Z A P C** * * * * * * * * *
Operands	**Bytes**	**Clocks**	**Example**
	1	32	**iret**

Rückkehr von einer Interrupt-Service-Routine in das unterbrochene Programm. Dazu werden IP, CS und Flags vom Stack geladen.

INT

INT interrupt-number Call interrupt procedure			Flags **O D I T S Z A P C** **0 0**
Operands	**Bytes**	**Clocks**	**Example**
3	1	52	**int 3**
immed8	2	52	**int 21h**

Software Interrupt
Realisiert einen indirekten Far-Call für häufig gebrauchte Systemfunktionen mit Hilfe der Interrupt-Pointer-Tabelle (im Hauptspeicher 0...3ffh). Flags, CS und IP werden auf den Stack gelegt, bevor CS und IP umgeladen werden.

INTO

INTO (no operands) Interrupt if overflow			Flags **O D I T S Z A P C** **0 0**
Operands	**Bytes**	**Clocks**	**Example**
	1	53	**into**

Software Interrupt Nr. 4, wenn OF=1 (Vorzeichenbit zerstört, arithmetischer Überlauf).

2.2. Bedingter Sprung

Jcond

Jcond short-label Jump if condition true			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
short-label	2	16(4)	**jnz SMARK**

Bedingter Sprung zu einer Marke, die innerhalb -128..+127 Byte liegt. Der Assembler ermittelt die Distanz zu dieser Marke als 1-Byte-Wert.

Mnemonic	bin	Flag-Condition	Type	jump if
JB/JNA **JC**	72h	CF=1	unsigned	below/not above or equal carry
JAE/NB **JNC**	73h	CF=0	unsigned	above or equal/not below not carry
JBE/NA	76h	(CF or ZF)=1	unsigned	below or equal/not above
JA/NBE	77h	(CF or ZF)=0	unsigned	above/not below or equal
JE/JZ	74h	ZF=1		equal/zero
JNE/JNZ	75h	ZF=0		not equal/not zero
JL/JNGE	7ch	(SF xor OF)=1	signed	less/not greater or equal
JGE/JNL	7dh	(SF xor OF)=0	signed	greater or equal/not less
JLE/JNG	7eh	(ZF or(SF xor OF))=1	signed	less or equal/not greater
JG/JNLE	7fh	(ZF or(SF xor OF))=0	signed	greater/not less or equal
JS	78h	SF=1		sign
JNS	79h	SF=0		not sign
JO	70h	OF=1		overflow
JNO	71h	OF=0		not overflow
JP/JPE	7ah	PF=1		parity/parity even
JNP/JPO	7bh	PF=0		no parity/parity odd

2.3. Schleifensteuerung

LOOP

LOOP short-label Loop			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
short-label	2	17(5)	**loop NOCH_MAL**
CX wird dekrementiert (ohne Flagbeeinflussung). Ist es dann $\neq 0$, wird ein Sprung zur angegebenen Adresse ausgeführt. Bei CX=0 wird der nächste Befehl im Text bearbeitet. (Realisierung einer Zählschleife).			

LOOPc

LOOPc short-label Loop conditionally			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
short-label	2	18(6)	**loope AGAIN**
short-label	2	19(5)	**loopnz WDH**
Wie LOOP; zusätzlich wird das Zeroflag getestet und die Schleife vorzeitig verlassen, wenn die Bedingung nicht (mehr) erfüllt ist. c = Z/E (ZF=1) oder c = NZ/NE (ZF=0)			

JCXZ

JCXZ short-label Jump if CX is zero			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
short-label	2	16(4)	**jcxz FERTIG**
Bedingter Sprung zu einer Marke innerhalb -128...+127 Byte, wenn CX=0 ist.			

3. Arithmetik

3.1. Addition

ADD

ADD destination,source Addition			Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**add cl,ah** **add ax,si**
register,memory	2-4	9+ea	**add cx, [W_VAR]** **add dl,[B_TAB+bx+si]**
memory,register	2-4	16+ea	**add [W_VAR],dx** **add [B_TAB+5],bl**
register,immediate	3-4	4	**add al, 3** **add cx, 369ch**
memory,immediate	3-6	17+ea	**add [VAR], 8** **add [W_TAB+di],444**
accu,immediate	2-3	4	**add al,7** **add ax,0a000h**

Addition des Source-Operanden zum Destination-Operand.

Erlaubte Kombination der Operanden	**Destination**	**Source**				
		A	R	S	M	K
	Accumul A	**X**	**X**	–	**X**	**X**
	Register R	**X**	**X**	–	**X**	**X**
	SegReg S	–	–	–	–	–
	Memory M	**X**	**X**	–	–	**X**

ADC

ADC destination,source Add with Carry			Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**adc cl,ah**
register,memory	2-4	9+ea	**adc cx, [W_VAR]**
memory,register	2-4	16+ea	**adc [W_VAR],dx**
register,immediate	3-4	4	**adc dx,0**
memory,immediate	3-6	17+ea	**adc [W_TAB+di],444**
accu,immediate	2-3	4	**adc al,7**
Addition von Source-Operand **und** Carrybit zum Destination-Operand.			

INC

INC destination Increment by 1			Flags **O D I T S Z A P C** ↕ ↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register16	1	3	**inc bx**
register8	2	3	**inc ah**
memory	2-4	15+ea	**inc [W_TAB+si+bx]**
Zieloperanden um 1 erhöhen. (CF **nicht** beeinflusst!)			

AAA

AAA (no operands) ASCII adjust for Addition			Flags **O D I T S Z A P C** **?** **? ?** ↕ **?** ↕
Operands	**Bytes**	**Clocks**	**Example**
	1	8	;AH=0, AL=4 add al,7 ;AH=0,AL=11 **aaa** ;AH=1,AL=1
Korrektur von AL **nach** einer Addition **ungepackter** BCD- bzw. ASCII-Ziffern. Falls erforderlich wird zu AL 6 addiert und AH um 1 erhöht.			

DAA

DAA (no operands) Decimal adjust for Addition			Flags **O D I T S Z A P C** **?** ↕ ↕ ↕ ↕ ↕
Operands	**Bytes**	**Clocks**	**Example**

	1	4	mov al,43h
			add al,18h ;AL=5bh
			daa ;AL=61h

Korrektur von AL **nach** einer Addition **gepackter** BCD-Ziffern. Wenn das 1er-Nibbel den Wert 9 übersteigt, wird zu diesem 6 addiert und das 10er-Nibbel um 1 erhöht.			

3.2. Subtraktion

SUB

SUB destination,source Subtract			Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**sub ax,si**
register,memory	2-4	9+ea	**sub dl,[B_TAB+bx+si]**
memory,register	2-4	16+ea	**sub [W_VAR], dx**
register,immediate	3-4	4	**sub bl, 3**
memory,immediate	3-6	17+ea	**sub [W_TAB+di],444**
accu,immediate	2-3	4	**sub al, 7**
Subtraktion des Source-Operand vom Destination-Operand. (Operandenkombination wie bei ADD)			

SBB

SBB destination,source Subtract with borrow			Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**sbb bx,si**
register,memory	2-4	9+ea	**sbb cl,[B_TAB+bx+si]**
memory,register	2-4	16+ea	**sbb [W_VAR], ax**
register,immediate	3-4	4	**sbb dx, 21h**
memory,immediate	3-6	17+ea	**sbb [W_TAB+si],444**
accu,immediate	2-3	4	**sbb al, 0**
Subtraktion des Source-Operand **und** des Carrybit vom Destination-Operand.			

DEC

DEC destination Decrement by 1			Flags **O D I T S Z A P C** ↕ ↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register16	1	3	**dec ax**
register8	2	3	**dec cl**
memory	2-4	15+ea	**dec byte [bx]** ;(NASM)
Zieloperanden um 1 vermindern. (CF **nicht** beeinflusst!)			

CMP

CMP destination,source Compare destination to source			Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**cmp bx, cx**
register,memory	2-4	9+ea	**cmp dh, [ALPHA]**
memory,register	2-4	9+ea	**cmp [bp+2], si**
register,immediate	3-4	4	**cmp bl, –1**
memory,immediate	3-6	10+ea	**cmp [W_TAB+si],444**
accu,immediate	2-3	4	**cmp al, 01000011b**
Vergleich der beiden Operanden (temporäre Subtraktion, beide Operanden bleiben ungeändert, nur die Flags werden entsprechend dem Ergebnis gesetzt).			

NEG

NEG destination Negate			Flags **O D I T S Z A P C** ↕ ↕↕↕↕*
Operands	**Bytes**	**Clocks**	**Example**
register	2	3	**neg ax**
memory	2-4	16+ea	**neg byte [bx];(NASM)**
Arithmetische Negation des Zieloperanden durch Ausführung des 2er-Komplements (aus 00000101b=5 wird 11111011b=–5) *) CF=1; Ergebnis=0 ⇒ CF=0			

AAS

AAS (no operands) ASCII adjust for subtraction			Flags **O D I T S Z A P C** **?** **? ?** ↕ **?** ↕
Operands	**Bytes**	**Clocks**	**Example**
	1	8	;AH=2, AL=4 sub al,8 ;AL=-4=0fch **aas** ;AH=1,AL=6
Korrektur von AL **nach** einer Subtraktion **ungepackter** BCD- bzw. ASCII-Ziffern. Falls erforderlich wird von AH 1 abgezogen und die 1er-Stelle korrigiert.			

DAS

DAS (no operands) Decimal adjust for subtraction			Flags **O D I T S Z A P C** **?** ↕ ↕ ↕ ↕ ↕
Operands	**Bytes**	**Clocks**	**Example**

	1	4	mov al,53h
			sub al,17h ;AL=3ch
			das ;AL=36h

Korrektur von AL **nach** einer Subtraktion **gepackter** BCD-Ziffern. Wenn das 1er-Nibbel den Wert 9 übersteigt, wird nochmals 6 davon subtrahiert.			

3.3. Multiplikation

MUL

MUL source Multiplication (unsigned)			Flags **O D I T S Z A P C** ↕ **? ? ? ?** ↕
Operands	**Bytes**	**Clocks**	**Example**

register8	2	..77	mov al,200 ;0...255
			mov cl,111
			mul cl ;AX=22200

register16	2	..133	mov ax,3000;0...65535
			mov bx,1100
			mul bx
			;DX:AX=3300000

memory8	2-4	..83+ea	**mul [BYTE_VAR]**
memory16	2-4	..139+ea	**mul [W_TAB+bp+si]**
Vorzeichenlose (Ganzzahl-)Multiplikation. 8-Bit-Faktoren ⇒ 16-Bit-Produkt in AX, 16-Bit-Faktoren ⇒ 32-Bit-Produkt in DX:AX, (Zieloperand implizit)			

IMUL

IMUL source Integer multiplication			Flags **O D I T S Z A P C** ↕ **? ? ? ?** ↕
Operands	**Bytes**	**Clocks**	**Example**

register8	2	..98	mov al,35 ;-128..+127
			mov cl,10
			imul cl ;AX=350

register16	2	..154	mov ax,–3000
			mov bx,1100
			imul bx
			;DX:AX=–3300000

memory8	2-4	..104+ea	**imul [BYTE_VAR]**
memory16	2-4	..160+ea	**imul [W_TAB+bp+si]**
Vorzeichenbehaftete Ganzzahl-Multiplikation. 8-Bit-Faktoren ⇒ 16-Bit-Produkt in AX, 16-Bit-Faktoren ⇒ 32-Bit-Produkt in DX:AX, (Zieloperand implizit)			

AAM

AAM (no operands) ASCII adjust for multiply			Flags **O D I T S Z A P C** **? ? ? ↕ ? ↕**
Operands	**Bytes**	**Clocks**	**Example**
	2	83	;AH=?, AL=5 mov cl,7 mul cl ;AX=35=23h **aam** ;AH=3,AL=5
Korrektur von AL **nach** einer Multiplikation **ungepackter** BCD-Ziffern. In AH wird die 10er-Stelle und in AL die 1er-Stelle erzeugt.			

AAD

AAD (no operands) ASCII adjust for division			Flags **O D I T S Z A P C** **? ? ? ↕ ? ↕**
Operands	**Bytes**	**Clocks**	**Example**
	2	60	;AH=4, AL=7 **aad** ;AX=2fh mov cl,5 div cl ;AH=2,AL=9
Korrektur von AL **vor** einer Division **ungepackter** BCD-Ziffern			

3.4. Division

DIV

DIV source Division (unsigned)			Flags **O D I T S Z A P C** **? ? ? ? ? ? ?**
Operands	**Bytes**	**Clocks**	**Example**

register8	2	..77	mov ax,35 ;AX=23h
			mov cl,10
			div cl ;AL=3,AH=5

register16	2	..162	mov dx,2 ;Hi=131072
			mov ax,5
			;DX:AX=131077
			mov bx,1000
			div bx ;AX=131
			;DX=77

memory8	2-4	..96+ea	**div [BYTE_DIV]**
memory16	2-4	..168+ea	**div [W_TAB+bp+si]**

Vorzeichenlose (Ganzzahl-)Division. 16-Bit-Dividend in AX durch 8-Bit-Divisor ⇒ Quotient in AL, Rest in AH. 32-Bit Dividend in DX:AX durch 16-Bit-Divisor ⇒ Quotient in AX, Rest in DX (Zieloperand implizit). Bei Überlauf des Quotientenregisters wird automatisch ISR #0 aufgerufen!

IDIV

IDIV source Integer division			Flags **O D I T S Z A P C** **? ? ? ? ? ?**
Operands	**Bytes**	**Clocks**	**Example**

register8	2	..112	mov ax,35 ;AX=23h
			mov cl,10
			idiv cl ;AL=3,AH=5
			mov ax,–35
			;AX=0ffddh
			idiv cl
			;AL=–3,AH=5

register16	2	..184	mov ax,–30000
			cwd ;DX:AX=–30000
			mov bx,1100
			idiv bx ;AX=–27
			;DX=–300

memory8	2-4	..96+ea	**idiv [BYTE_DIV]**
memory16	2-4	..168+ea	**idiv [W_TAB+bp+si]**

Vorzeichenbehaftete Ganzzahl-Division. 16-Bit-Dividend in AX durch 8-Bit-Divisor ⇒ Quotient in AL, Rest in AH. 32-Bit Dividend in DX:AX durch 16-Bit-Divisor ⇒ Quotient in AX, Rest in DX. Vorzeichen des Rest = Vorzeichen des Dividenden (Zieloperand implizit). Bei Überlauf des Quotientenregisters wird automatisch ISR #0 aufgerufen!

CBW

CBW (no operands) Convert byte to word			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**

	1	2	mov al,5 ;AH=?, AL=5
			cbw ;AH=0,AL=5
			mov al,133 ;AX=0085h
			cbw ;AX=0ff85h

Vorzeichengerechte Erweiterung des Wertes in AL nach AX.			

CWD

CWD (no operands) Convert word to doubleword			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	5	mov ax,–13 ;DX=? **cwd** ;DX=0ffffh
Vorzeichengerechte Erweiterung des Wertes in AX nach DX:AX.			

4. Logische Operationen

4.1. Logische Verknüpfung

NOT

NOT destination Logical not (One's complement)			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
register	2	3	**not ax**
memory	2-4	16+ea	**not [BITMSK]**
Bitweise Negation des Zieloperanden.			

AND

AND destination,source Logical AND			Flags **O D I T S Z A P C** **0** ↕ ↕ **?** ↕ **0**
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**and cl,ah**
register, memory	2-4	9+ea	**and cx, [W_VAR]**
memory,register	2-4	16+ea	**and [B_TAB+bx+di],dl**
register,immediate	3-4	4	**and dl,111b**
memory,immediate	3-6	17+ea	**and [W_TAB+bx],24h**
accu,immediate	2-3	4	**and al, 7**
Logische bitweise UND-Verknüpfung der beiden Operanden. (1-Bit entsteht nur, wenn **beide** entsprechenden Bits der Operanden = 1 sind.)			

TEST

TEST destination,source Logical Compare			Flags **O D I T S Z A P C** **0** ↕ ↕ **?** ↕ **0**
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**test ax, si**
register, memory memory,register[5]	2-4	9+ea	**test bl, [si+2]**
register,immediate	3-4	5	**test bl, 0fh**
memory,immediate	3-6	11+ea	**test [W_TAB+bx], 2**
accu,immediate	2-3	4	**test ax, 400h**

Temporäre logische bitweise UND-Verknüpfung der beiden Operanden. (Zieloperand bleibt ungeändert, nur die Flags werden dem Ergebnis entsprechend gesetzt.)

[5] memory,register wird vertauscht!

OR

OR destination,source Logical inclusive OR			Flags **O D I T S Z A P C** **0** ↕ ↕ **?** ↕ **0**
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**or ax, si**
register, memory	2-4	9+ea	**or dl, [B_TABsi]**
memory,register	2-4	16+ea	**or [W_VAR], ax**
register,immediate	3-4	4	**or cl, 0f0h**
memory,immediate	3-6	17+ea	**or [W_TAB+di],1011b**
accu,immediate	2-3	4	**or al, 7**

Logische bitweise ODER-Verknüpfung der beiden Operanden. (1-Bit entsteht dann, wenn mindestens **eines** der entsprechenden Bits der beiden Operanden = 1 ist.)

XOR

XOR destination,source Logical exclusive OR			Flags **O D I T S Z A P C** **0** ↕ ↕ **?** ↕ **0**
Operands	**Bytes**	**Clocks**	**Example**
register,register	2	3	**xor cl, ah**
register, memory	2-4	9+ea	**xor cx, [W_VAR]**
memory,register	2-4	16+ea	**xor [B_TAB+bx], al**
register,immediate	3-4	4	**xor cx, 36ch**
memory,immediate	3-6	17+ea	**xor [VAR], 8**
accu,immediate	2-3	4	**xor ax, 0a000h**

Logische bitweise Exklusiv-ODER-Verknüpfung der beiden Operanden. (1-Bit entsteht nur, wenn die entsprechenden Bits der beiden Operanden **ungleich** sind.)

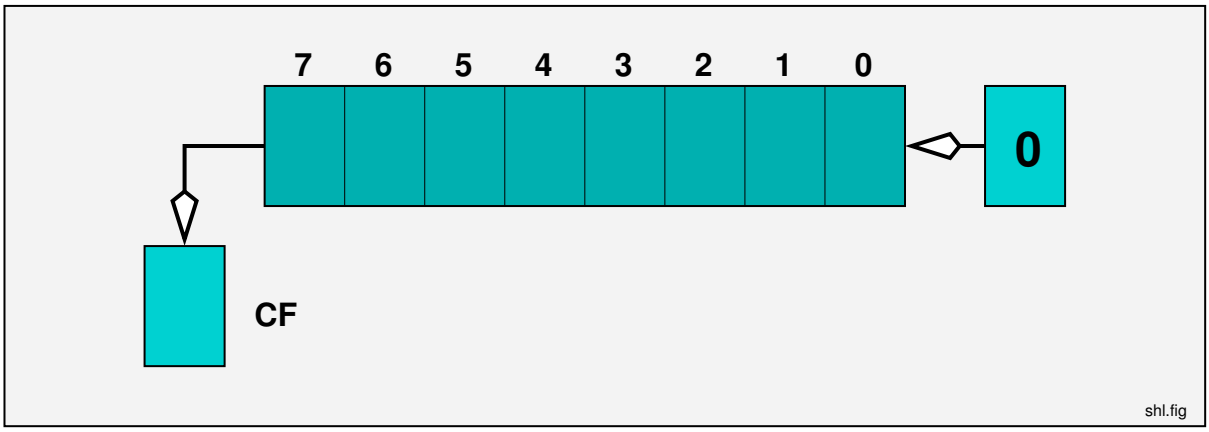
4.2. Bitmuster verschieben

SAL SHL

SAL destination,count **SHL destination,count** Shift arithmetic/logical left	Flags **O D I T S Z A P C** ↕ ↕↕ **?** ↕↕

Operands	Bytes	Clocks	Example
register,1	2	2	**sal cx, 1** ;CX:=CX*2
register,CL	2	8+4*c	mov cl,3 **shl ax, cl** ;AX:=AX*8
memory,1	2-4	15+ea	**shl [VAR], 1**
memory,CL	2-4	20++	**sal [BETA], cl**

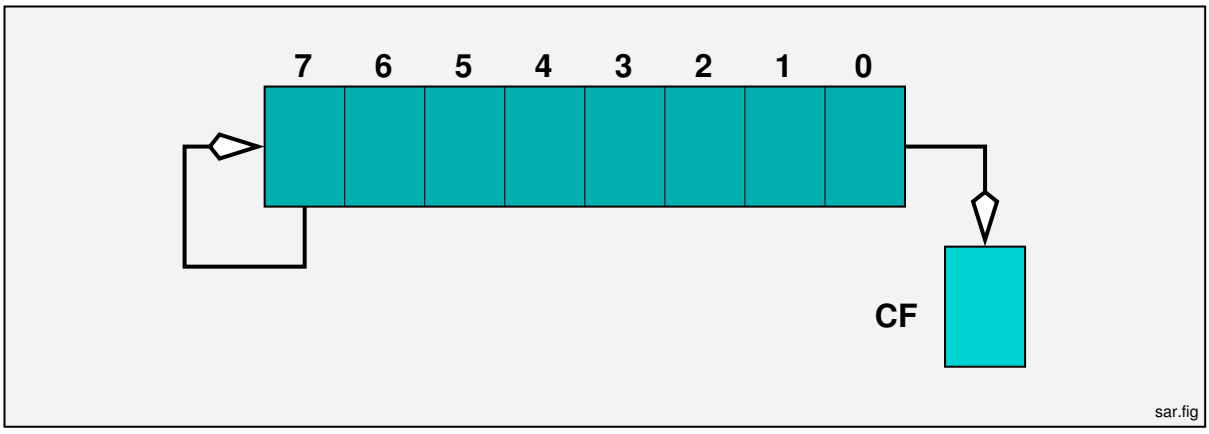
Das Bitmuster des Operanden wird um die durch *count* spezifizierte Zahl von Stellen nach links verschoben. Von rechts rückt 0 nach, CF enthält das zuletzt herausgeschobene Bit. OF signalisiert die Änderung des MSB.



SAR

SAR destination,count Shift arithmetic right			Flags **O D I T S Z A P C** ↕ ↕ **?** ↕ ↕
Operands	**Bytes**	**Clocks**	**Example**
register,1	2	2	**sar cx, 1** ;CX:=CX/2
register,CL	2	8+4*c	mov cl,4 **sar ax,cl** ;AX:=AX/16
memory,1	2-4	15+ea	**sar [VAR], 1**
memory,CL	2-4	20++	**sar [BETA], cl**

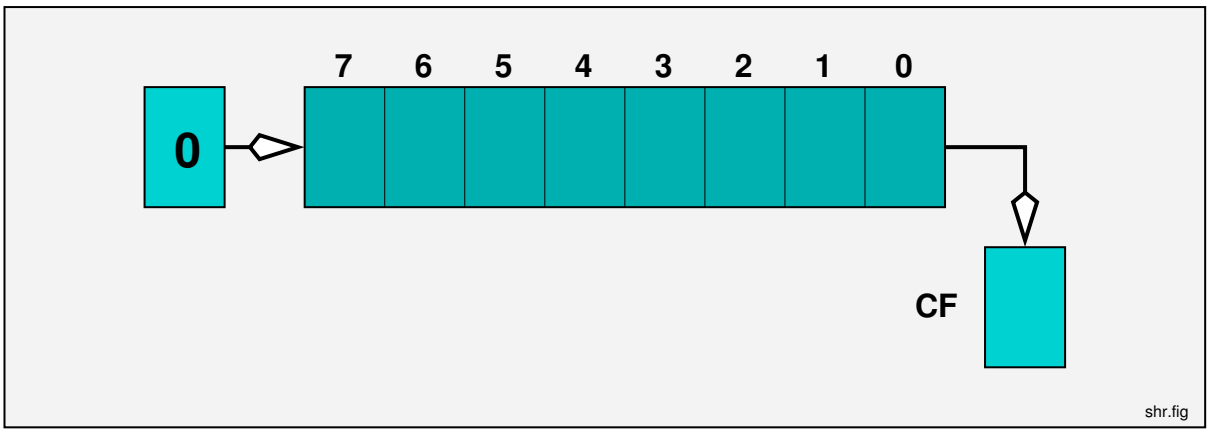
Das Bitmuster des Operanden wird um die durch *count* spezifizierte Zahl von Stellen nach rechts verschoben. Das MSB wird beibehalten, CF enthält das zuletzt herausgeschobene Bit. OF wird bei count=1 gelöscht (sonst undefiniert).



SHR

SHR destination,count Shift logical right			Flags **O D I T S Z A P C** ↕ ↕ **?** ↕ ↕
Operands	**Bytes**	**Clocks**	**Example**
register,1	2	2	**shr cx, 1** ;CX:=CX/2
register,CL	2	8+4*c	mov cl,8 **shr ax,cl**;AX:=AX/256
memory,1	2-4	15+ea	**shr [VAR], 1**
memory,CL	2-4	20++	**shr [BETA], cl**

Das Bitmuster des Operanden wird um die durch *count* spezifizierte Zahl von Stellen nach rechts verschoben. Ins MSB rückt 0 nach, CF enthält das zuletzt herausgeschobene Bit. OF widerspiegelt die Änderung des MSB bei count=1 (sonst undefiniert).



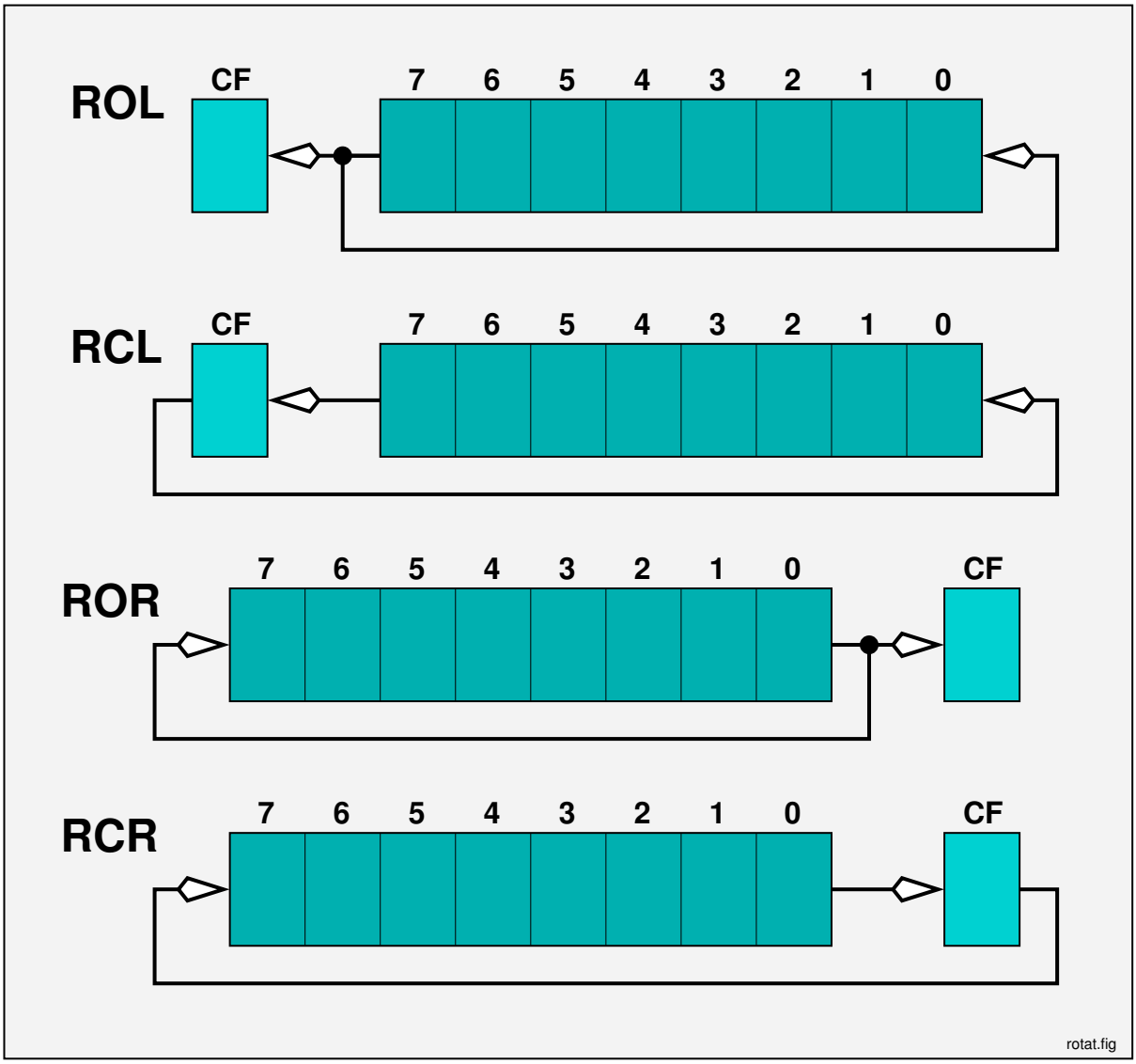
4.3. Bitmuster rotieren

ROL RCL ROR RCR

ROL destination,count **RCL destination,count** **ROR destination,count** **RCR destination,count** Rotate	Flags **O D I T S Z A P C** ↕ ↕

Operands	**Bytes**	**Clocks**	**Example**
register,1	2	2	**rol cx, 1**
register,CL	2	8+4*c	mov cl,3 **rcl ax,cl**
memory,1	2-4	15+ea	**ror [VAR], 1**
memory,CL	2-4	20++	**rcr [BETA], cl**

Das Bitmuster des Operanden wird um die durch *count* spezifizierte Zahl von Stellen nach links bzw. rechts verschoben. Herausgeschobene Bits rücken auf der Gegenseite nach: entweder direkt oder nachdem sie CF passiert haben. CF enthält das zuletzt herausgeschobene Bit. OF signalisiert die Änderung des MSB.



Rotation von Bitmustern

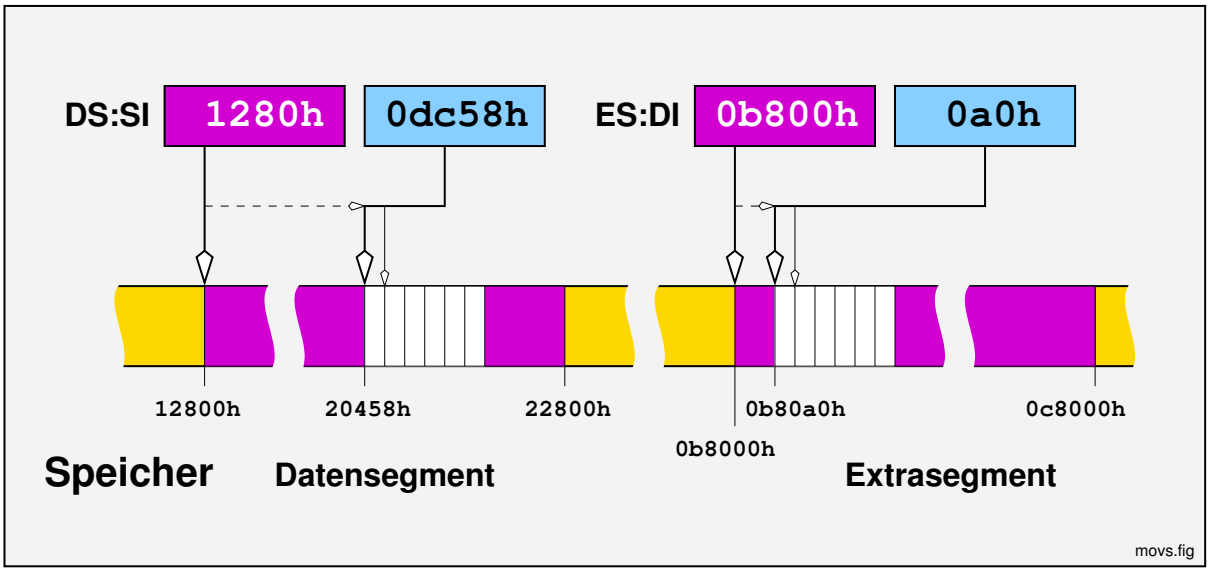
5. Zeichenkettenbefehle

MOVSB MOVSW

MOVSB **MOVSW** Move string element	Flags **O D I T S Z A P C**

Operands	Bytes	Clocks	Example
	1+1	9+17*r	lea di,STR1 lea si,STR2 mov cx,24 **rep movsw** ;24 Words kopiert
	1	18	**movsb**
	1	18	**movsw**

Objekt des Originalstring wird in den Zielstring kopiert. Die Adressierung findet **ausschließlich** über die Registerkombinationen **DS:SI** und **ES:DI** statt. SI und DI werden entsprechend dem Directionflag nach dem Transfer byte- bzw. wortweise erhöht oder erniedrigt. Durch Verwendung des REP-Präfix-Befehls wird erreicht, dass die zuvor in CX spezifizierte Elementeanzahl übertragen wird (sog. HW-Schleife für Block-Transfer).



LODSB LODSW

LODSB
LODSW
Load string element

Flags **ODITSZAPC**

Operands	Bytes	Clocks	Example
	1	12	**lodsb**
	1	12	**lodsw**

Element des Originalstring wird in den Accumulator kopiert. Die Adressierung findet **ausschließlich** über **DS:SI** statt. SI wird nach dem Transfer byte- bzw. wortweise erhöht oder erniedrigt (REP-Präfix nicht sinnvoll)

STOSB STOSW

STOSB **STOSW** Store string element			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1+1	9+10*r	**rep stosw**
	1	11	**stosb**
	1	11	**stosw**

Accu-Wert wird in den Zielpuffer kopiert. Die Adressierung findet **ausschließlich** über **ES:DI** statt. DI wird nach dem Transfer byte- bzw. wortweise erhöht oder erniedrigt. Mittels REP-Präfix kann eine Blockinitialisierung erreicht werden.

SCASB SCASW

SCASB **SCASW** Scan string	Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕

Operands	Bytes	Clocks	Example
	1+1	9+15*r	**repne scasw**
	1	15	**scasb**
	1	15	**scasw**

Accu-Wert wird mit einem Element des Zielpuffers verglichen. Die Adressierung findet **ausschließlich** über **ES:DI** statt. DI wird nach dem Vergleich byte- bzw. wortweise erhöht oder erniedrigt. Mittels REPNE-Präfix kann ein bestimmtes Zeichen gesucht werden, mit REPE-Präfix wird das angegebene Zeichen 'überlesen'.

CMPSB CMPSW

CMPSB **CMPSW** Compare string elements	Flags **O D I T S Z A P C** ↕ ↕↕↕↕↕

Operands	**Bytes**	**Clocks**	**Example**
	1+1	9+22*r	**repe cmpsw**
	1	22	**cmpsb**
	1	22	**cmpsw**

Elementweiser Vergleich beider Strings. Die Adressierung findet **ausschließlich** über **DS:SI** und **ES:DI** statt. SI und DI werden nach dem Vergleich byte- bzw. wortweise erhöht oder erniedrigt. Mittels REPE-Präfix kann die Übereinstimmung zweier Zeichenketten geprüft werden.

REP

REP Repeat string operation			Flags **ODITSZAPC**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**rep movsb**

Wiederholungspräfix für MOVS und STOS. Nach jedem Transfer wird CX dekrementiert. Ist dann CX $\neq$ 0, wird der Transferbefehl wiederholt ('HW-Schleife' zum Kopieren bzw. Initialisieren von Strings, deren Länge in CX und deren Adresse in DS:SI bzw. ES:DI steht).

REPE REPNE REPZ REPNZ

REPE/REPZ **REPNE/REPNZ** Repeat string operation (cond.)			Flags **ODITSZAPC**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**repe cmpsb**

Wiederholungspräfix für CMPS und SCAS. Wie bei REP wird die Wiederholung des Befehls durch CX gesteuert. Zusätzlich wird ZF getestet und die Wiederholung **storniert**, wenn die Bedingung **nicht** mehr **erfüllt** ist (ZF$\Rightarrow$0 bei REPE/REPZ bzw. ZF$\Rightarrow$1 bei REPNE/REPNZ).

6. Prozessorsteuerung

STC

STC (no operands) Set carryflag			Flags **O D I T S Z A P C** **1**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**stc**
Explizites Setzen des Carryflag.			

CLC

CLC (no operands) Clear carryflag			Flags **O D I T S Z A P C** **0**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**clc**
Explizites Löschen des Carryflag.			

CMC

CMC (no operands) Complement carryflag			Flags **O D I T S Z A P C** ↕
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**cmc**
Umkehren des Carryflag.			

STD

STD (no operands) Set directionflag			Flags **O D I T S Z A P C** **1**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**std**
Setzen des Directionflag (Stringbefehle arbeiten rückwärts, d.h. nach fallenden Adressen).			

CLD

CLD (no operands) Clear directionflag			Flags **O D I T S Z A P C** **0**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**cld**
Löschen des Directionflag (Stringbefehle arbeiten vorwärts, d.h. nach steigenden Adressen).			

STI

STI (no operands) Set interrupt enable flag			Flags **O D I T S Z A P C** **1**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**sti**
Setzen des Interrupt Enable Flag (Zulassen von maskierbaren Interrupts).			

CLI

CLI (no operands) Clear interrupt enable flag			Flags **O D I T S Z A P C** **0**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**cli**
Löschen des Interrupt Enable Flag (Sperren von maskierbaren Interrupts).			

ESC

ESC ext-opcode[,source] Escape			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
immed6,register	2	2	**esc 6,al**
immed6,memory	2-4	8+ea	**esc 29,[bx]**
Bereitstellen eines externen Opcode (8087). Der Direktwert spezifiziert einen (von 64) Coprozessorbefehlen, der (optionale) Source-Operand stellt ihm erforderlichenfalls auf dem Datenbus Zusatzinformationen bereit. ESC wird bei Benutzung von Coprozessor-Mnemonics vom Assembler automatisch eingefügt.			

WAIT

WAIT (no operands) Wait			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	3+5*n	**wait**
Prozessor geht in den Wait-Zustand bis das Signal am TEST-Pin aktiviert wird (n = Anzahl der Bus-Zyklen im Wartezustand). Kann durch zugelassene Interrupts unterbrochen werden.			

LOCK

LOCK Lock bus			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**lock xchg [FLG], al**
Durch den Präfixbefehl LOCK wird der Bus für die Dauer des folgenden Befehls für die Benutzung durch einen anderen Prozessor (Mehrprozessorsystem - MULTIBUS^{TM}) gesperrt.			

HLT

HLT (no operands) Wait			Flags **O D I T S Z A P C**
Operands	**Bytes**	**Clocks**	**Example**
	1	2	**hlt**
Prozessor anhalten (Halt-Status). Kann nur durch einen zugelassenen Interrupt oder Reset aufgehoben werden.			