Echtzeit–Videoverarbeitung

Einführung in x86–Assembler Programmierung

Marcus Barkowsky

Überblick

Prozessoraufbau

- Register
- Speicher
- Adressierung

Assembler

- x86 Befehle
- MMX−Erweiterung
- Pseudo Operationen

Praktische Programmierung

- Umgang mit dem Editor xemacs
- Aufruf des Assemblers nasm
- Arbeiten mit dem Debugger ddd

Gründe für Assemblerprogrammierung

- Geschwindigkeitssteigerung
- Projektion von Algorithmen auf real vorhandene Strukturen
- Erweiterung des Verständnisses der technischen Vorgänge
- Ausnutzung spezieller Prozessoreigenschaften
- Fehlende Hochsprachencompiler bei neuen Prozessoren
- Verringerung des Programmspeicherbedarfs

Unterschiede zwischen Assembler und Hochsprachen

- Deutlich reduzierter Funktionsumfang der Befehle
- Direkte Nachvollziehbarkeit der Berechnungsvorgänge
- Unmittelbarer Zugriff auf Ressourcen
  - Speichermanagement
  - Rechenleistung
  - Peripherie
- Keine Strukturelemente
- Unterscheidung in Speicher– und Registerzugriffe
- Bytecodierung möglich

Prozessoren der x86 Familie

Prozessor	Daten-bus	Adr.–bus	Besonderheiten
8086/8087	16	20	Adressierung in Segment/Offset
80186	16	20	Spezialanwendungen
80286/287	16	24	Protected Mode, Segmentierung
80386/387	32	32	Paging
80486	32	32	Eingebauter Coprozessor
Pentium	32	32	Power Management, Pipelining
Pentium mit MMX	32	32	MMX Technologie
Pentium II	32	32	Multitasking Support, RISC–Kern
Pentium III	32	32	ISSE

Bedeutung von Speicherinhalten

Speicher:

2	1	0	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	31	30
0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	1	0

unsigned 8Bit

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	1

181

signed 8Bit

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	1

−75

Flags

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	1

Slot für Reinigungsband
Slot für automatische Kopie
Medium vorhanden
Medium schreibgeschützt
Slotnummer

Umgang mit negativen Zahlen

Zahlendarstellung:

Berechnung des 2er Komplements: $-A = (\sim A) + 1$

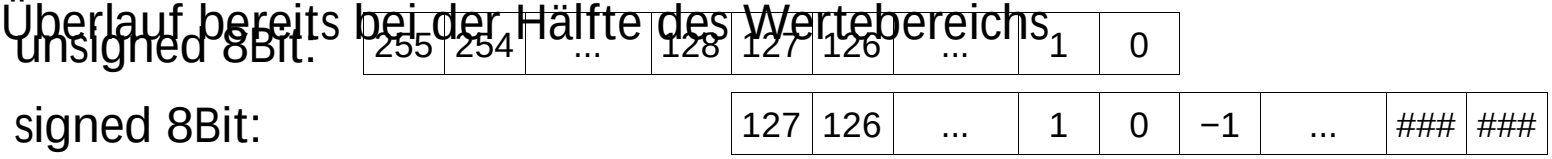
42d	=	0010 1010b				
~42d	=	1101 0101b	=	213d	=	−43d
~42d+1	=	1101 0110b	=	214d	=	−42d
−42d	=	1101 0110b	=	214d		
~(−42d)	=	0010 1001b	=	41d		
~(−42d)+1	=	0010 1010b	=	42d		

Anmerkungen:

Bei Zahlenbereichskonversion: Vorzeichenerweiterung (sign extension)

signed 8 Bit:	1101 0110b	=	−42d
signed 16 Bit:	0000 0000 1101 0110b	=	214d
	1111 1111 1101 0110b	=	−42d

Überlauf bereits bei der Hälfte des Wertebereichs



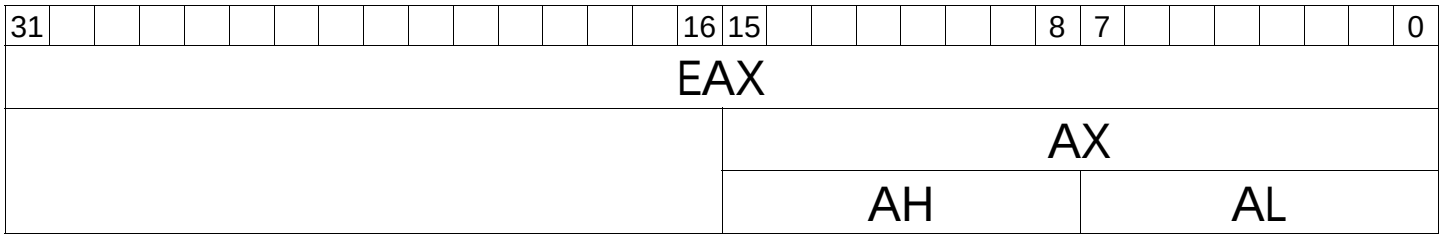
Prozessorregister

Rechenregister: 32bit

- A: Akkumulator
- B: Base
- C: Counter
- D: Data

Zugriff am Beispiel Akkumulator:

- EAX: 32Bit Register
- AX: 16Bit Register
- AH: High Byte von AX
- AL: Low Byte von AX



Prozessorregister

Segmentregister 16bit

Inhalt: Anfangsadressen auf Speicheradressen

- CS: Codesegment
- DS: Datensegment
- SS: Stacksegment
- ES: Extra Segment (frei verfügbar)
- FS: (frei verfügbar)
- GS: (frei verfügbar)

Indexregister: 32/16bit

Inhalt: Offset bei Adreßberechnungen

- (E)IP: Instruction Pointer
- (E)SP: Stack Pointer
- (E)BP: Base Pointer
- **(E)SI:** Source Index Register
- **(E)DI:** Destination Index Register

Prozessorregister

Flagregister: 32bit/16bit

Inhalt: Seiteninformationen zur letzten ausgeführten Operation, wird meist für bedingte Sprünge genutzt

- (E)F:Flags des Prozessors

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				O	**D**	I	T	**S**	**Z**		A		P		**C**

- **O:** Overflow, Überlauf bei vorzeichenbehafteten (signed) Zahlen
- **D:** Direction, bei impliziter Veränderung der Indexregister (0=inc, 1=dec)
- I: Interrupt Enable
- T: Trap, Single Step
- **S:** Sign, Vorzeichenbit des Ergebnisregisters
- **Z:** Zero, Ergebnis der Berechnung war 0
- A: Auxiliary, wird für Binary−coded−decimal verwendet
- P: Parity, Ergebnis lieferte gerade Anzahl gesetzter Bits, z.B. 0101 1100
- **C:** Carry, Überlauf bei vorzeichenlosen Zahlen

Stackverwaltung

- Schnelle Möglichkeit zum Zwischenspeichern von Registern
- Übergabeparameter an Unterfunktionen
- Rücksprungvektoren nach Beendigung von Unterprogrammen
- Last–In First–Out Strategie (nach unten wachsend)

Schreibzugriff (push 0x3E4A)

	5	29
	4	4A
ESP →	3	3E
	2	
	1	
	0	

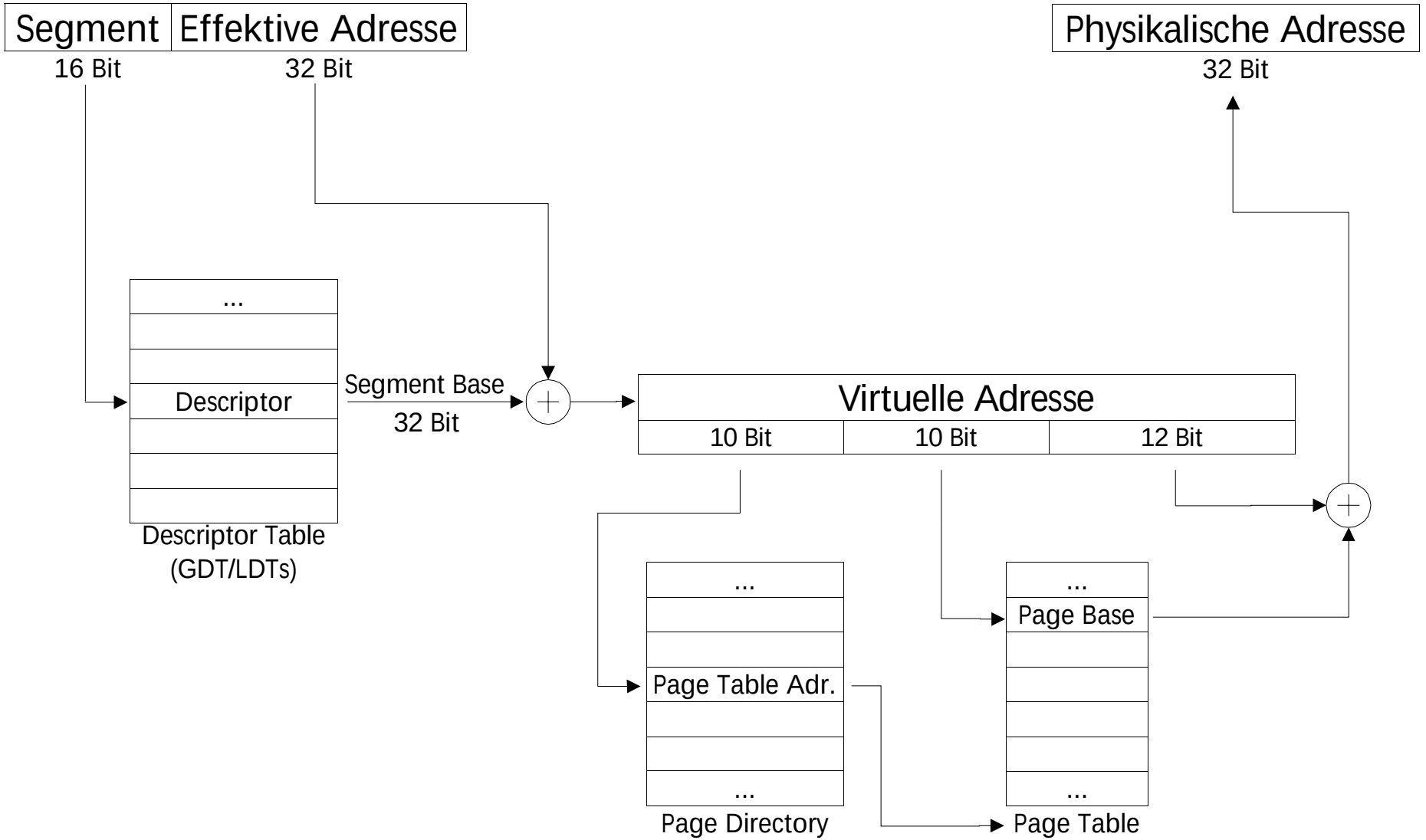
Schreibzugriff (push 0x4711)

	5	29
	4	4A
	3	3E
	2	11
ESP →	1	47
	0	

Lesezugriff (pop ax)

	5	29
	4	4A
ESP →	3	3E
	2	
	1	
	0	

Zugriff auf den Hauptspeicher



Adressierung des Hauptspeichers

Berechnung der effektiven Adresse:

$$\begin{bmatrix} EAX \\ ECX \\ EDX \\ EBX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{bmatrix} \cdot \begin{bmatrix} EAX \\ ECX \\ EDX \\ EBX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \begin{bmatrix} 8\,Bit\,Offset \\ 32\,Bit\,Offset \end{bmatrix}$$

Beispiele:

- mov ax, [ebx]
- mov cx, [esp+4]
- mov eax, [ebx + 4*esi + 8]

Assembler Syntax

4 Elemente pro Zeile:

Label: Befehl Operand(en) ; Kommentar

Anmerkungen:

- In Intel Syntax ist stets der 1.Operand das Ziel der Operation
- Bei fast allen Befehlen ist maximal 1 Speicherzugriff erlaubt
- Die Operanden können auch implizit vorgegeben sein
- Eine Zeile kann auch nur aus Label−, Befehls− oder Kommentarfeld bestehen

Beispiel:

```
            mov ecx, 0x10       ; Lade ecx Register mit 16
            mov eax, 11b        ; Lade in den Akkumulator eax 3
addition:   add eax, eax        ; Addiere eax mit eax
            loop addition       ; solange bis ecx auf 0 ist
                                ; decrementiere und springe zur Addition
```

Kurzvorstellung der x86−Befehle

Kurzschreibweise der Operanden:

- r8: 8 Bit (Byte) Register (AL, CL, DL, BL, AH, CH, DH, BH)
- r16: 16 Bit (Word) Register (AX, CX, DX, BX, SP, BP, SI, DI)
- r32: 32 Bit (Doubleword) Register (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI)
- reg: beliebig breites Register
- r/m8: Wahlweise 8 Bit Register oder 8 Bit Speicherzugriff
- r/m16: Wahlweise 16 Bit Register oder 16 Bit Speicherzugriff
- r/m32: Doppelwortzugriff auf Register oder Speicher
- mem: Speicher beliebig breit
- imm8: 8 Bit Immidiate (1 Byte Konstante)
- imm16: Wort Konstante
- imm32: Doppelwort Konstante
- imm: Konstante beliebig breit
- sreg: Segmentregister

MOV: Move

Verwendung:

Syntax	Beispiel	Takte
mov reg,reg	mov ah,bl	1
mov reg,imm	mov eax,12345h	1
mov mem,reg	mov [Wvar],ax	1
mov reg,mem	mov ebx,[Dvar]	1
mov sreg,reg	mov es,ax	2
mov sreg,mem	mov es,[Wvar]	3
mov reg,sreg	mov ax,ds	1
mov mem,sreg	mov [Wvar],fs	1

Aufgabe:

„Move“ kopiert den Inhalt des zweiten Operanden in den ersten Operanden. Quelle und Ziel müssen gleiche Datenbreite haben.

Flags:

Keine Änderung

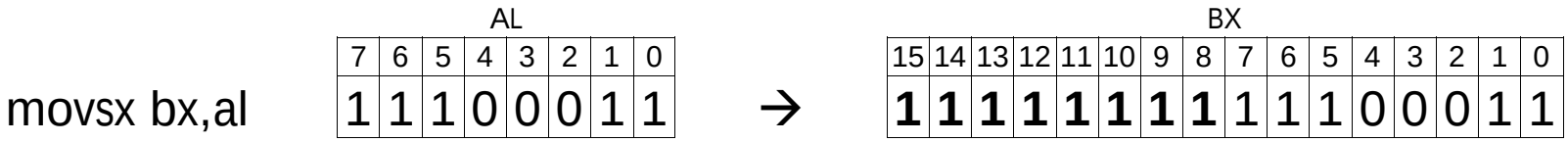
MOVSX: Move with sign extension

Verwendung:

Syntax	Beispiel	Takte
movsx r16, r/m8	movsx ax,bl	3
movsx r32,r/m8	movsx ecx,cl	3
movsx r32,r/m16	movsx esi,di	3

Aufgabe:

Arbeitet wie MOV, erweitert jedoch auf die Zielgröße. Die Quelle wird als vorzeichenbehaftete Zahl interpretiert.



Flags:

Keine Änderung

MOVZX: Move with zero extension

Verwendung:

Syntax	Beispiel	Takte
movzx r16, r/m8	movzx ax,bl	3
movzx r32,r/m8	movzx ecx,cl	3
movzx r32,r/m16	movzx esi,di	3

Aufgabe:

Arbeitet wie MOV, erweitert jedoch auf die Zielgröße. Die Quelle wird als vorzeichenlose Zahl interpretiert.

movsz bx,al

AL

7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1

→

BX

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	0	0	0	1	1

Flags:

Keine Änderung

CBW: Convert Byte to Word (with sign extension)

Verwendung:

Syntax	Beispiel	Takte
cbw	cbw	3

Aufgabe:

Kopiert das Bit 7 aus AL in Bits 8−15 von AX. Erweitert also einen signed 8Bit Wert im AL−Register zu einem Wort (signed 16Bit) in AX.

Flags:

Keine Änderung

CWDE: Convert Word to Doubleword

Verwendung:

Syntax	Beispiel	Takte
cwde	cwde	3

Aufgabe:

Kopiert das Bit 15 aus AX in Bits 16−31 von EAX. Erweitert also einen signed 16Bit Wert im AX−Register zu einem Doppelwort (signed 32Bit) in EAX.

Flags:

Keine Änderung

CDQ: Convert Doubleword to Quadword

Verwendung:

Syntax	Beispiel	Takte
cdq	cdq	2

Aufgabe:

Kopiert das Bit 31 aus EAX in Bits 0−31 von EDX. Erweitert also einen signed 32Bit Wert im EAX−Register zu einem Quadwort (signed 64Bit) in EDX:EAX.

Flags:

Keine Änderung

XCHG: Exchange register/memory with register

Verwendung:

Syntax	Beispiel	Takte
xchg r8,r/m8	xchg ah,bl	3
xchg m8,r8	xchg [Bvar],ch	3
xchg r16,r/m16	xchg ax,bx	3
xchg m16,r16	xchg [Wvar],dx	3
xchg r32,r/m32	xchg eax,[Dvar]	3
xchg m32,r32	xchg [Dvar],eax	3

Aufgabe:

Tauscht den Inhalt des ersten Operanden mit dem des zweiten Operanden

Flags:

Keine Änderung

XLAT: Table lookup translation

Verwendung:

Syntax	Beispiel	Takte
xlat	xlat	4

Aufgabe:

Kopiert ein Byte aus einer Tabelle und legt es in AL ab. Die Adresse der Tabelle steht in DS:EBX, der gewünschte Index muß in AL stehen.

Flags:

Keine Änderung

LEA: Load effective address

Verwendung:

Syntax	Beispiel	Takte
lea r16, [Bvar]	lea si, [Bvar]	1
lea r32, [Wvar]	lea edi, [eax+4*edi]	1

Aufgabe:

Kopiert den Offsetanteil der durch den zweiten Operaden adressierten Speicher–zelle in das angegebene Register. Es findet kein Zugriff auf die Speicherzelle selbst statt.

Flags:

Keine Änderung

LDS: Load far pointer to data segment

Verwendung:

Syntax	Beispiel	Takte
lds r16, [Bvar]	lds si, [Bvar]	4
lds r32, [Wvar]	lds edi, [eax+4*edi]	4

Aufgabe:

Liest an der im zweiten Operanden angegebenen Speicherstelle eine vollständige Adresse (48 Bit) aus. Der Selektoranteil wird in das Datensegment−register kopiert, die effektive Adresse in das angegebene Register.
Andere Segmentregister können mit LES, LFS, LGS und LSS geladen werden.

Flags:

Keine Änderung

PUSH: Push word or doubleword to stack

Verwendung:

Syntax	Beispiel	Takte
push r16	push ax	1
push m16	push [Wvar]	2
push i16	push 4711h	1
push r32	push eax	1
push m32	push [Dvar]	2
push i32	push 012345678h	1
push sreg	push ds	1

Aufgabe:

Push kopiert das Wort oder Doppelwort auf den Stack, wo es mit Pop wieder geladen werden kann. Der Stackpointer in SS:ESP wird dabei entsprechend verringert.

Flags:

Keine Änderung

PUSHAD: Push all general–purpose registers to stack

Verwendung:

Syntax	Beispiel	Takte
pushad	pushad	5

Aufgabe:

PUSHAD kopiert alle Register, also eax, ecx, edx, ebx, esp, ebp, esi und edi (in dieser Reihenfolge) auf den Stack. Es gibt auch die Möglichkeit, nur die 16Bit Werte der Register mit PUSHA, bzw. PUSHAW zu speichern.

Flags:

Keine Änderung

PUSHF: Push flag register to stack

Verwendung:

Syntax	Beispiel	Takte
pushf	pushf	4

Aufgabe:

PUSHF kopiert das Flag−Register auf den Stack, so daß eine spätere Wiederherstellung der Flags ermöglicht wird. PUSHFW ist ein Synonym für PUSHF. Darüberhinaus gibt es noch PUSHFD, welches die EFLAGS speichert.

Flags:

Keine Änderung

POP: Pop a value from stack

Verwendung:

Syntax	Beispiel	Takte
pop r16	pop ax	1
pop m16	pop [Wvar]	3
pop r32	pop eax	1
pop m32	pop [Dvar]	3
pop sreg	pop ds	3

Aufgabe:

POP holt einen auf dem Stack gespeicherten Wert in das angegebene Register, bzw. in die referenzierte Speicheradresse. Der Stackpointer in SS:ESP wird dabei entsprechend erhöht.

Flags:

Keine Änderung

POPAD: Pop all general−purpose registers from stack

Verwendung:

Syntax	Beispiel	Takte
popad	popad	5

Aufgabe:

POPAD stellt alle Register, also eax, ecx, edx, ebx, esp, ebp, esi und edi (in dieser Reihenfolge) aus dem Stack−Inhalt wieder her. Analog zu PUSHA, bzw. PUSHAW verhalten sich POPA, bzw. POPAW.

Flags:

Keine Änderung

POPF: Pop flag register from stack

Verwendung:

Syntax	Beispiel	Takte
popf	popf	6

Aufgabe:

POPF ist das Gegenstück zu PUSHF und holt den Inhalt des Flag−Registers vom Stack. Analog zu PUSHFW und PUSHFD existieren POPFW und POPFD.

Flags:

Alle Flags werden entsprechend dem auf dem Stack gespeicherten Wert geändert.

LAHF: Load status flags into AH register

Verwendung:

Syntax	Beispiel	Takte
lahf	lahf	2

Aufgabe:

Kopiert das Flag Register nach AH.

Flags:

Keine Änderung

SAHF: Store AH into flags

Verwendung:

Syntax	Beispiel	Takte
sahf	lahf	2

Aufgabe:

Kopiert den Inhalt des AH−Registers in das Flag Register

Flags:

Werden entsprechend dem Inhalt von AH geändert

JMP: Jump

Verwendung:

Syntax	Beispiel	Takte
jmp imm	jmp label	1
jmp r32	jmp eax	1
jmp mem	jmp [eax+4*esi]	2

Aufgabe:

Es wird ein unbedingter Sprung an die angegebene Adresse durchgeführt.

Flags:

Keine Änderung

CALL: Call procedure

Verwendung:

Syntax	Beispiel	Takte
call imm	call label	1
call r32	call eax	2
call mem	call [eax+4*esi]	2

Aufgabe:

Mit Hilfe des Befehls „CALL“ kann in Unterprogramme verzweigt werden. Nach Bearbeitung eines korrespondierenden „RET“ Befehls wird der Programmablauf mit dem nächsten Befehl fortgesetzt. Hierzu wird die Rücksprungadresse auf dem Stack gespeichert.

Flags:

Keine Änderung

RET: Return from procedure

Verwendung:

Syntax	Beispiel	Takte
ret	ret	2

Aufgabe:

Rücksprung aus einem Unterprogramm, welches mit „CALL“ aufgerufen wurde.

Flags:

Keine Änderung

CMP: Compare two operands

Verwendung:

Syntax	Beispiel	Takte
cmp reg,reg	cmp ah,bl	1
cmp reg,imm	cmp eax,12345h	1
cmp mem,reg	cmp [Wvar],ax	2
cmp reg,mem	cmp ebx,[Dvar]	2
cmp mem,imm	cmp [Wvar],1234h	2

Aufgabe:

Es wird der zweite Operand vom ersten subtrahiert und die Flags entsprechend gesetzt. Das Ergebnis der Subtraktion wird verworfen, d.h. die Operanden bleiben unverändert.

Flags:

Werden entsprechend dem Ergebnis der Subtraktion gesetzt.

TEST: Logical compare

Verwendung:

Syntax	Beispiel	Takte
test reg,reg	test ah,bl	1
test reg,imm	test eax,12345h	1
test mem,reg	test [Wvar],ax	2
test mem,imm	test [Wvar],1234h	2

Aufgabe:

Es wird eine AND−Verknüpfung der beiden Operanden vorgenommen und die Flags werden entprechend verändert. Das Ergebnis wird wie bei CMP verworfen, d.h. die Operanden bleiben unverändert.

Flags:

Werden entsprechend dem Ergebnis der Verundung gesetzt, Overflow und Carry−Flag werden gelöscht.

Jcc: Jump if condition is met

Verwendung:

Syntax	Beispiel	Takte
Jcc imm	jb label	1

Aufgabe:

Je nach Sprungbefehl werden verschiedene Kombinationen der Flags überprüft. Im Falle, daß die jeweile Kombination vorgefunden wird, erfolgt ein Sprung an die angegebene Adresse.
Die Befehle lassen sich in folgende Kategorien aufteilen:
(A:above, B:below, E:equal, N:not, G:greater, L:less, Z:zero)

- Vorzeichenlose Zahlen: JA, JAE, JB, JBE, JNA, JNAE, JNB, JNBE
- Vorzeichenbehaftete Zahlen: JG, JGE, JL, JLE, JNG, JNGE, JNL, JNLE
- Allgemein: JE, JNE, JNZ, JZ
- Direkte Flagauswertung: JC, JNC, JNO, JNP, JNS, JO, JP, JPE, JPO, JS
- Das (e)cx−Register betreffend: JCXZ, JECXZ

Flags:

Keine Änderung

Jcc: Jump if condition is met

Auswertung:

Zur besseren Übersicht hier die Tabelle der Bedingungen:

Name	Befehle	Bedingung	
above	JA, JNBE	op1> op2	cf=0 and zf=0
above or equal	JAE, JNB, JNC	op1>=op2	cf=0
below	JB, JNAE, JC	op1< op2	cf=1
below or equal	JBE, JNA	op1<=op2	cf=1 or zf=1
greater	JG, JNLE	op1> op2	sf=of and zf=0
greater or equal	JGE, JNL	op1>=op2	sf=of
less	JL, JNGE	op1< op2	sf!=of
less or equal	JLE, JNG	op1<=op2	sf!=of or zf=1
carry set	JC (=JB)		cf=1
carry not set	JNC (=JAE)		cf=0
zero set	JZ	op1=op2	zf=1
zero not set	JNZ	op1!=op2	zf=0
sign (not) set	JS (JNS)		sf=1(0)
overflow (not) set	JO (JNO)		of=1(0)
parity set	JP, JPE (parity even)		pf=1
parity not set	JNP, JPO (parity odd)		pf=0
(e)cx zero	J(E)CXZ	cx=0	

LOOP/LOOPcc: loop according to ecx counter

Verwendung:

Syntax	Beispiel	Takte
loop imm	loop label	5/6
loope imm	loope label	7/8
loopne imm	loopne label	7/8

Die jeweils ersten Taktwerte gelten, falls kein Sprung erfolgt.

Aufgabe:

Das ecx Register wird decrementiert. Anschließend wird es auf 0 verglichen. Falls es noch nicht 0 ist, wird ein Sprung ausgeführt. Mit den Befehlen LOOPE (loop equal, Synonym LOOPZ) bzw. LOOPNE (LOOPNZ) kann zusätzlich das Zero−Flag abgefragt werden. Bei LOOPE wird nicht gesprungen, d.h. die Schleife verlassen, wenn entweder ecx=0 oder zf=0.

Flags:

Keine Änderung. Die Verringerung von ecx und der Vergleich erfolgt intern.

INC: Increment by 1

Verwendung:

Syntax	Beispiel	Takte
inc reg	inc ah	1
inc mem	inc [Dvar]	3

Aufgabe:

Der Operand wird um 1 erhöht

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt, mit Ausnahme des Carry–Flags, es wird nicht verändert.

DEC: Decrement by 1

Verwendung:

Syntax	Beispiel	Takte
dec reg	dec ah	1
dec mem	dec [Dvar]	3

Aufgabe:

Der Operand wird um 1 erniedrigt

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt, mit Ausnahme des Carry–Flags, es wird nicht verändert.

NEG: Two's complement negation

Verwendung:

Syntax	Beispiel	Takte
neg reg	neg ah	1
neg mem	neg [Dvar]	3

Aufgabe:

Das Vorzeichen des Operanden wird invertiert.

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt

ADD: add

Verwendung:

Syntax	Beispiel	Takte
add reg,reg	add ah,bl	1
add reg,imm	add eax,12345h	1
add reg,mem	add ebx,[Dvar]	2
add mem,reg	add [Wvar],ax	3
add mem,imm	add [Dvar],1234h	3

Aufgabe:

Die beiden Operanden werden addiert, das Ergebnis im 1.Operand gespeichert.

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt

ADC: Add with carry

Verwendung:

Syntax	Beispiel	Takte
adc reg,reg	adc ah,bl	1
adc reg,imm	adc eax,12345h	1
adc reg,mem	adc ebx,[Dvar]	2
adc mem,reg	adc [Wvar],ax	3
adc mem,imm	adc [Dvar],1234h	3

Aufgabe:

Die beiden Operanden werden addiert, zusätzlich wird das Carry−Flag addiert. Das Ergebnis wird im 1.Operand gespeichert. Sinnvoll einsetzbar, wenn Daten−worte breiter als die Register verarbeitet werden müssen. Dann kann mit diesem Befehl der Übertrag aus der niederwertigeren Addition berücksichtigt werden.

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt

SUB: Subtract

Verwendung:

Syntax	Beispiel	Takte
sub reg,reg	sub ah,bl	1
sub reg,imm	sub eax,12345h	1
sub reg,mem	sub ebx,[Dvar]	2
sub mem,reg	sub [Wvar],ax	3
sub mem,imm	sub [Dvar],1234h	3

Aufgabe:

Der zweite Operand wird vom ersten subtrahiert.

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt

SBB: Subtract with borrow

Verwendung:

Syntax	Beispiel	Takte
sbb reg,reg	sbb ah,bl	1
sbb reg,imm	sbb eax,12345h	1
sbb reg,mem	sbb ebx,[Dvar]	2
sbb mem,reg	sbb [Wvar],ax	3
sbb mem,imm	sbb [Dvar],1234h	3

Aufgabe:

Der zweite Operand wird vom ersten subtrahiert. Zusätzlich wird das Carry–Flag subtrahiert. Ähnlich wie ADC kann auch dieser Befehl eingesetzt werden, um überlange Datenworte zu verarbeiten.

Flags:

Die Flags werden entsprechend dem Ergebnis gesetzt

MUL: Unsigned multiply

Verwendung:

Syntax	Beispiel	Takte
mul reg	mul bl	10/11
mul mem	mul [Dvar]	10/11

Bei Verarbeitung von 32 Bit Operanden werden 10 Takte benötigt.

Aufgabe:

Es wird eine Multiplikation des Akkumulators mit dem angegebenen Operand durchgeführt. Folgende Tabelle gibt an, wo das Produkt auffindbar ist:

Operand	Multiplikand	Produkt
Byte	AL	AX
Wort	AX	DX:AX
Doppelwort	EAX	EDX:EAX

Flags:

Overflow und Carry werden gesetzt, die anderen Flags sind undefiniert.

IMUL: Signed multiply

Verwendung:

Syntax	Beispiel	Takte
imul reg	imul bl	10/11
imul reg, imm	imul ax, 256h	10
imul reg, reg	imul ax, bx	10
imul reg, reg, imm	imul bx, dx, 4	10
imul reg, mem, imm	imul bx, [Wvar], 4	10

Bei Verarbeitung von 32 Bit Operanden werden 10 Takte benötigt.

Aufgabe:

Wenn nur ein Operand angegeben wird, erfolgt die Multiplikation der vorzeichenbehafteten Werte, wie in der Tabelle bei MUL dargestellt.
Wenn zwei Operanden angegeben werden, so wird der erste Operand mit dem Produkt der beiden angegebenen Operanden überschrieben.
Die Drei−Operanden Befehle multiplizieren die beiden letzten Operanden und speichern das Ergebnis im ersten Operand ab.

Flags:

Wenn nur ein Operand verwendet wird sind Overflow und Carry gesetzt, die anderen Flags sind undefiniert. Sonst zeigen sie einen Überlauf an.

AND: Logical AND

Verwendung:

Syntax	Beispiel	Takte
and reg,reg	and ah,bl	1
and reg,imm	and eax,12345h	1
and reg,mem	and ebx,[Dvar]	2
and mem,reg	and [Wvar],ax	3
and mem,imm	and [Dvar],1234h	3

Aufgabe:

„AND“ führt eine logische UND−Verknüpfung des ersten mit dem zweiten Operanden durch. Das Ergebnis wird im 1.Operanden gespeichert.

Flags:

Werden entsprechend dem Ergebnis der Verundung gesetzt, Overflow und Carry−Flag werden gelöscht.

OR: Logical inclusive OR

Verwendung:

Syntax	Beispiel	Takte
or reg,reg	or ah,bl	1
or reg,imm	or eax,12345h	1
or reg,mem	or ebx,[Dvar]	2
or mem,reg	or [Wvar],ax	3
or mem,imm	or [Dvar],1234h	3

Aufgabe:

„OR“ führt eine logische ODER−Verknüpfung des ersten mit dem zweiten Operanden durch. Das Ergebnis wird im 1.Operanden gespeichert.

Flags:

Werden entsprechend dem Ergebnis der Verundung gesetzt, Overflow und Carry−Flag werden gelöscht.

XOR: Logical exclusive OR

Verwendung:

Syntax	Beispiel	Takte
xor reg,reg	xor ah,bl	1
xor reg,imm	xor eax,12345h	1
xor reg,mem	xor ebx,[Dvar]	2
xor mem,reg	xor [Wvar],ax	3
xor mem,imm	xor [Dvar],1234h	3

Aufgabe:

„XOR“ führt eine logische Exklusiv−ODER−Verknüpfung des ersten mit dem zweiten Operanden durch. Das Ergebnis wird im 1.Operanden gespeichert.

Flags:

Werden entsprechend dem Ergebnis der Verundung gesetzt, Overflow und Carry−Flag werden gelöscht.

NOT: One’s complement negation

Verwendung:

Syntax	Beispiel	Takte
not reg	not ah	1
not mem	not [Dvar]	3

Aufgabe:

Jedes Bit des angegebenen Operanden wird invertiert.

Flags:

Keine Änderung

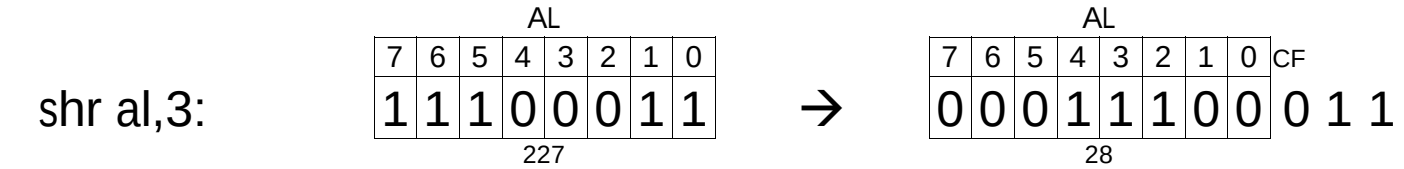
SHR: Shift logical right

Verwendung:

Syntax	Beispiel	Takte
shr reg,i8	shr eax,5	1
shr mem,i8	shr [edx],3	3
shr reg,CL	shr eax,cl	4
shr mem,CL	shr [edx],cl	4

Aufgabe:

Shifted den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach rechts. Bei vorzeichenlosen Zahlen ist dies identisch mit der Division durch die entsprechende Zweierpotenz.
Für vorzeichenbehaftete Zahlen sollte SAR verwendet werden.



Flags:

Die Flags werden entprechend gesetzt, das Carry−Flag enthält das zuletzt nach rechts rausgeschobene Bit

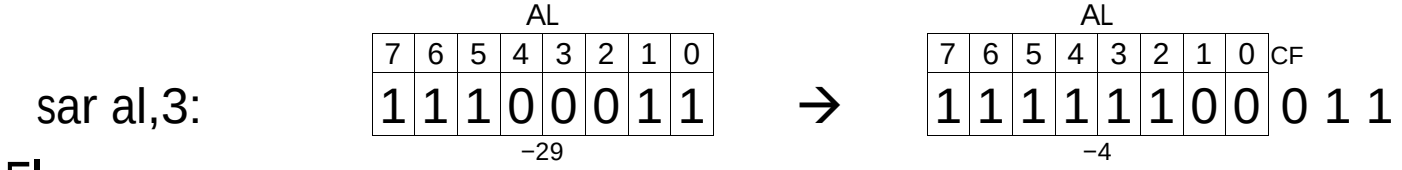
SAR: Shift arithmetic right

Verwendung:

Syntax	Beispiel	Takte
sar reg,i8	sar eax,5	1
sar mem,i8	sar [edx],3	3
sar reg,CL	sar eax,cl	4
sar mem,CL	sar [edx],cl	4

Aufgabe:

Shifted den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach rechts. Da die hinzugekommenen Bits gleich dem Vorzeichenbit des Ausgangswertes sind (sign extenstion), entspricht dies der Division durch die entsprechende Zweierpotenz.
Für vorzeichenlose Zahlen sollte SHR verwendet werden.



Flags:

Die Flags werden entprechend gesetzt, das Carry−Flag enthält das zuletzt nach rechts rausgeschobene Bit

SHL: Shift logical left

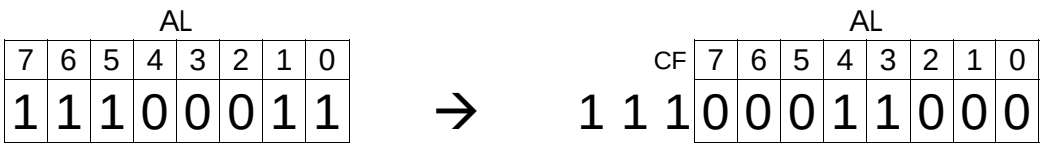
Verwendung:

Syntax	Beispiel	Takte
shl reg,i8	shl eax,5	1
shl mem,i8	shl [edx],3	3
shl reg,CL	shl eax,cl	4
shl mem,CL	shl [edx],cl	4

Aufgabe:

Shifted den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach links. Bei vorzeichenlosen Zahlen ist dies identisch mit der Multi−plikation mit der entsprechenden Zweierpotenz.
SAL und SHL sind identische Befehle.

shl al,3:



Flags:

Die Flags werden entprechend gesetzt, das Carry−Flag enthält das zuletzt nach links rausgeschobene Bit

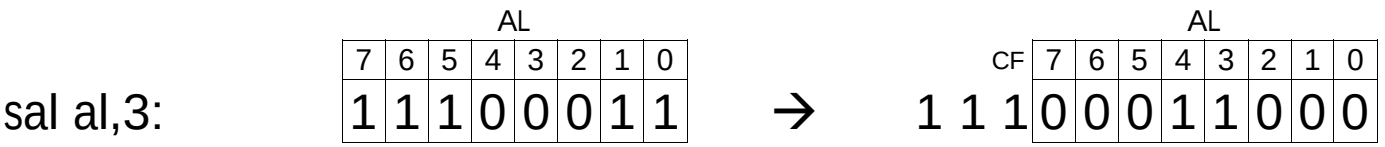
SAL: Shift arithmetic left

Verwendung:

Syntax	Beispiel	Takte
sal reg,i8	sal eax,5	1
sal mem,i8	sal [edx],3	3
sal reg,CL	sal eax,cl	4
sal mem,CL	sal [edx],cl	4

Aufgabe:

Shifted den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach links. Bei vorzeichenlosen Zahlen ist dies identisch mit der Multi−plikation mit der entsprechenden Zweierpotenz.
SAL und SHL sind identische Befehle.



Flags:

Die Flags werden entprechend gesetzt, das Carry−Flag enthält das zuletzt nach links rausgeschobene Bit

ROL: Rotate left

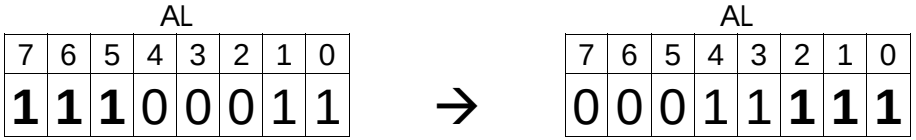
Verwendung:

Syntax	Beispiel	Takte
rol reg,i8	rol eax,5	1
rol mem,i8	rol [edx],3	3
rol reg,CL	rol eax,cl	4
rol mem,CL	rol [edx],cl	4

Aufgabe:

Rotiert den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach links.

rol al,3:



Flags:

Keine Änderung

ROR: Rotate right

Verwendung:

Syntax	Beispiel	Takte
ror reg,i8	ror eax,5	1
ror mem,i8	ror [edx],3	3
ror reg,CL	ror eax,cl	4
ror mem,CL	ror [edx],cl	4

Aufgabe:

Rotiert den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach rechts.

ror al,3:

AL

7	6	5	4	3	2	1	0
1	1	1	0	0	**0**	**1**	**1**

→

AL

7	6	5	4	3	2	1	0
0	**1**	**1**	1	1	1	0	0

Flags:

Keine Änderung

RCL: Rotate through carry left

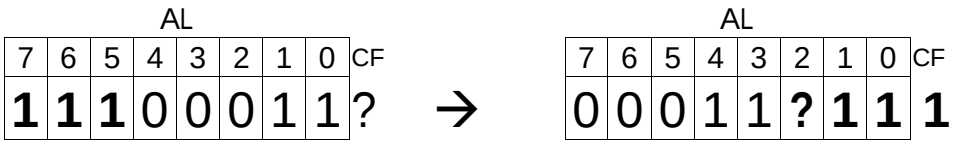
Verwendung:

Syntax	Beispiel	Takte
rcl reg,i8	rcl eax,5	8−25
rcl mem,i8	rcl [edx],3	10−27
rcl reg,CL	rcl eax,cl	7−24
rcl mem,CL	rcl [edx],cl	9−26

Aufgabe:

Rotiert den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach links, wobei ein zusätzliches Bit jeweils im Carry Flag abgespeichert wird.

rcl al,3:



Flags:

Das Carry−Flag enthält das zuletzt nach links rausgeschobene Bit, alle anderen Flags bleiben unverändert.

RCR: Rotate through carry right

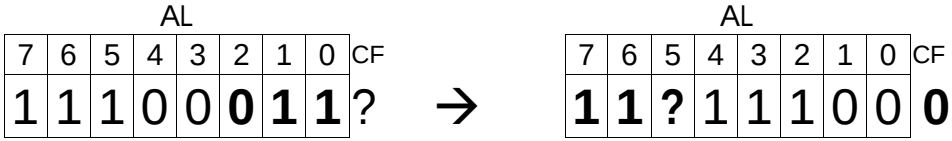
Verwendung:

Syntax	Beispiel	Takte
rcr reg,i8	rcr eax,5	8−25
rcr mem,i8	rcr [edx],3	10−27
rcr reg,CL	rcr eax,cl	7−24
rcr mem,CL	rcr [edx],cl	9−26

Aufgabe:

Rotiert den 1.Operanden um die im zweiten Operanden angegebene Anzahl binär nach rechts, wobei ein zusätzliches Bit jeweils im Carry Flag abgespeichert wird.

rcr al,3:



Flags:

Das Carry−Flag enthält das zuletzt nach rechts rausgeschobene Bit, alle anderen Flags bleiben unverändert.

STC: Set carry flag

Verwendung:

Syntax	Beispiel	Takte
stc	stc	2

Aufgabe:

Setzt das Carry Flag auf 1

Flags:

CF=1

CLC: Clear carry flag

Verwendung:

Syntax	Beispiel	Takte
clc	clc	2

Aufgabe:

Setzt das Carry Flag auf 0

Flags:

CF=0

CMC: Complement carry flag

Verwendung:

Syntax	Beispiel	Takte
cmc	cmc	2

Aufgabe:

Invertiert das Carry Flag

Flags:

CF=~CF

STD: Set direction flag

Verwendung:

Syntax	Beispiel	Takte
std	std	2

Aufgabe:

Setzt das Direction Flag auf 1, damit werden die Indexregister bei den entsprechenden String−Befehlen automatisch decrementiert

Flags:

DF=1

CLD: Clear direction flag

Verwendung:

Syntax	Beispiel	Takte
cld	cld	2

Aufgabe:

Setzt das Direction Flag auf 0, damit werden die Indexregister bei den entsprechenden String−Befehlen automatisch incrementiert

Flags:

DF=0

LODSB/LODSW/LODSD: Load string

Verwendung:

Syntax	Beispiel	Takte
lodsb	lodsb	2
lodsw	lodsw	2
lodsd	lodsd	2

Aufgabe:

Lädt dasjenige Byte/Wort/Doppelwort aus dem Speicher in AL/AX/EAX, welches durch DS:ESI referenziert wird. Das ESI Register wird anschließend um 1/2/4 erhöht (DF=0) oder erniedrigt (DF=1).

Flags:

Keine Änderung

STOSB/STOSW/STOSD: Store string

Verwendung:

Syntax	Beispiel	Takte
stosb	stosb	3
stosw	stosw	3
stosd	stosd	3

Aufgabe:

Speichert AL/AX/EAX an die durch ES:EDI angegebene Speicherzelle. Das EDI Register wird anschließend um 1/2/4 erhöht (DF=0) oder erniedrigt (DF=1).

Flags:

Keine Änderung

SCASB/SCASW/SCASD: Scan string

Verwendung:

Syntax	Beispiel	Takte
scasb	scasb	4
scasw	scasw	4
scasd	scasd	4

Aufgabe:

Vergleicht AL/AX/EAX mit der durch ES:EDI angegebenen Speicherzelle.
Der Vergleich entspricht einer virtuellen Subtraktion Akkumulator−[ES:EDI].
Das EDI Register wird anschließend um 1/2/4 erhöht (DF=0) oder erniedrigt (DF=1).

Flags:

Die Flags werden entsprechend dem Vergleich geändert.

MOVSB/MOVSW/MOVSD: Move from string to string

Verwendung:

Syntax	Beispiel	Takte
movsb	movsb	4
movsw	movsw	4
movsd	movsd	4

Aufgabe:

Kopiert das durch [DS:ESI] referenziert Byte/Wort/Doppelwort nach [ES:EDI]. Sowohl das EDI, als auch das ESI Register wird anschließend um 1/2/4 erhöht (DF=0) oder erniedrigt (DF=1).
Dies ist einer der wenigen x86−Befehle, die 2 Speicherzugriffe erlauben.

Flags:

Keine Änderung

CMPSB/CMPSW/CMPSD: Scan string

Verwendung:

Syntax	Beispiel	Takte
cmpsb	cmpsb	5
cmpsw	cmpsw	5
cmpsd	cmpsd	5

Aufgabe:

Vergleicht das durch [DS:ESI] referenziert Byte/Wort/Doppelwort mit demjenigen an der Stelle [ES:EDI]. Es findet eine virtuelle Subtraktion von [DS:ESI]−[ES:EDI] statt. Sowohl das EDI, als auch das ESI Register wird anschließend um 1/2/4 erhöht (DF=0) oder erniedrigt (DF=1).
Dies ist einer der wenigen x86−Befehle, die 2 Speicherzugriffe erlauben.

Flags:

Die Flags werden entsprechend dem Vergleich geändert.

REP/REPcc: Repeat string operation prefix

Verwendung:

Syntax	Beispiel	Takte
rep movs?	rep movsd	0+13*n
rep stos?	rep stosb	0+9*n
rep lods?	rep lodsb	7+3*n
repe scas?	repe scasb	9+4*n
repe cmps?	repe cmpsb	9+4*n
repne scas?	repne scasb	9+4*n
repne cmps?	repne cmpsw	9+4*n

n: Anzahl der Wiederholungen des Stringbefehls

Aufgabe:

„REP“ kann nur in Verbindung mit String−Operationen eingesetzt werden und ermöglicht eine schnelle Schleifenprogrammierung. Der Stringbefehl wird so lange wiederholt, bis ECX=0 ist. ECX wird nach jedem Durchgang decrementiert. Darüberhinaus kann ein Abbruch erfolgen, wenn das Zero−Flag durch den Stringbefehl gesetzt (repe/repz) bzw. gelöscht (repne/repnz) wurde.

Flags:

Das Zero−Flag wird möglicherweise durch den Stringbefehl verändert.

Programmbeispiel zu MOV, LEA, CALL, PUSH, POP

```
; Prototyp: long asm_function(long par);
; Aufruf:   fprintf(stderr,"asm_function(10) returns: %ld\n", asm_function(10));

    global asm_function

asm_function:
    push ebp              ; Put Basepointer to stack
    mov ebp, esp          ; Get Own Basepointer from current stack pointer
    push ebx
    push esi
    push edi

    mov ebx,27            ; ebx=27
    lea edx,[ebp+8]       ; Get address of first parameter
    mov eax,[edx]         ; Get first parameter
    call add_two_values   ; Call subroutine

    pop edi               ; Get values back from stack
    pop esi
    pop ebx
    mov esp,ebp           ; reinstall old stack pointer
    pop ebp
    ret                   ; return to calling function eax is return parameter

add_two_values:
    add eax,ebx
    ret
```

Beispiel zu CMP

```
    mov eax, -27
    mov ebx, -10
cmp_test:
    sub eax, ebx     ; eax=-17        ; eax=-7
    cmp eax, ebx     ; -17-(-10)=-7   ; -7-(-10)=3
    jl cmp_test      ; jmp cmp_test   ; nothing
```

Jump less: SF!=OF

cmp 30,40 (30<40)
30−40=−10 (SF=1, OF=0)

cmp −120,20 (−120<20)
−120−20=−140=116 (SF=0, OF=1)

cmp 40,30 (40<30)
40−30=10 (SF=0, OF=0)

cmp 20,−120 (20<−120)
20−(−120)=140=−116(SF=1, OF=1)

cmp 20,20 (20<20)
20−20=0 (SF=0, OF=0)

Beispiel zu ADC

```
void big_add(long *value1, long *value2);

int main(int argc, char**argv)
{
  long value1[4], value2[4];

  value1[0]=0x99999999; value1[1]=0x10;  value1[2]=0x10;       value1[3]=0xfffffffe;
  value2[0]=0x20;       value2[1]=0x20;  value2[2]=0xffffffff; value2[3]=0x20;

  big_add(value1, value2);
  return(0);
}
```

```
%include "asm_c.mac"

        global big_add
big_add:
        START_ASM_CODE
        mov esi,[ebp+12]      ; get source pointer
        mov edi,[ebp+8]       ; get destination pointer

        mov ecx,3             ; initialize loop counter
        clc                   ; first adc: add zero
.loop:  mov eax,[esi+4*ecx]   ; get source data
        adc [edi+4*ecx],eax   ; add to destination data
        loop .loop            ; loop until cx=0
        mov eax,[esi+4*ecx]   ; last value
        adc [edi+4*ecx],eax

        END_ASM_CODE
        ret                     ; return to calling function
```

Kopieren von Speicher

Ausgangswerte:
ecx= Anzahl zu kopierender Bytes, esi=Quell−Adresse, edi= Ziel−Adresse

Kopieren einzelner Bytes

```
.loop:   mov al, [esi]
         mov [edi], al
         inc esi
         inc edi
         dec ecx
         jnz .loop
```

Verwendung des loop−Befehls

```
.loop:  mov al, [esi]
        mov [edi], al
        inc esi
        inc edi
        loop .loop
```

Doppelwortweises Kopieren

```
.loop:   mov eax, [esi]
         mov [edi]. eax
         add esi,4
         add edi,4
         sub ecx,4
         jnz .loop
```

Indexorientiertes Kopieren

```
        shr ecx,2
.loop:  mov eax, [esi+4*ecx]
        mov [edi+4*ecx], eax
        dec ecx
        jg .loop
```

Verwendung des String−Befehls

```
         shr ecx,2
         cld
         rep movsd
```