Grundlegende Aspekte der Assemblerprogrammierung

Zunächst wollen wir einmal die Notwendigkeit der Programmierung in Maschinensprache betrachten. Es gibt heute kaum noch Probleme, welche nicht auch durch eine Hochsprache (DELPHI/C/ADA) gelöst werden könnten. Trotzdem gibt es Gründe für das Programmieren in Assembler. Hauptargument ist der Wunsch nach maximaler Performance der produzierten Programme. Zur Veranschaulichung werfen wir einen Blick auf die Codeerzeugung eines Delphi-Compilers. Dieser setzt die Anweisung „X:=0;“ nach folgendem Muster um:

1. Setze ein Register des Prozessor auf den Wert 0.
2. Schreibe den Inhalt dieses Registers an die Speicherstelle der Variablen X.

Mit 2 Prozessoranweisungen kann diese Aufgabe erledigt werden, woran auch überhaupt nichts auszusetzen ist. Problematisch ist aber eine Sequenz wie folgt: „X:=0;Y:=0;Z:=0;“, für welche der Compiler 6 Befehle nach obigem Muster produziert. Davon sind jedoch 2 Befehle vom Typ 1 überflüssig, da das Register bereits den Wert 0 enthält. Delphi erkennt dieses Problem aufgrund seiner Kompilierungsregeln aber nicht. Hier ist nur ein kleines Beispiel gegeben. Komplexe Berechnungen können aufgrund fehlender Optimierung derart langsam werden, dass eine Programmversion in Maschinensprache gewaltige Geschwindigkeitsgewinne bringen kann. Neben diesem Vorteil bei der Performance gibt es aber auch einen großen Nachteil bei der Assemblerprogrammierung in Gestalt der fehlenden **Portabilität** (Übertragbarkeit der Programme auf andere Rechnerplattformen). Hochsprachenanweisungen wie „WRITELN(‘HALLO WELT!’);“ können relativ problemlos von einer Plattform (z.B. IBM-kompatibel) zu einer anderen (z.B. RISC-Systeme) übertragen werden, falls für beide ein entsprechender Compiler zur Verfügung steht. Die Anweisungen eines Programms müssen dabei häufig nicht oder nur minimal geändert werden, und der Compiler übersetzt den Quelltext, wobei nun Code für den speziellen Prozessor erzeugt wird. Bei Assemblerprogrammen müssen diese meistens komplett an die neue Plattform angepasst werden, da die entsprechenden Anweisungen häufig nicht verfügbar oder anders aufgebaut sind. Softwarefirmen erstellen ihre Programme nach Möglichkeit zu 100% mit einer Hochsprache, um sie für verschiedene Rechnerplattformen anbieten zu können.

Hex- und Binärzahlen
Der grundlegende Aufbau von Binärzahlen dürfte bekannt sein. Es gibt aber verschiedene Möglichkeiten, diese Zahlen unter Benutzung von Bitoperationen der Booleschen Algebra zu bearbeiten, wobei die grundlegenden Operationen **NOT**, **AND**, **OR** und **XOR** sind.

a	b	a AND b	a OR b	a XOR b	NOT a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bei einer Betrachtung der Wirkungen der Verknüpfungen ergeben sich verschiedene Anwendungsmöglichkeiten: Mit OR (AND) können einzelne Bits gezielt gesetzt (gelöscht) werden.

Beispiel:	ZAHL	OPEARTION	VERKNÜPFUNGSOPERAND	ERGEBNIS
	????b	AND	0011b	00??b
	????b	OR	0011b	??11b

Nach der Verknüpfung sind im Ergebnis bestimmte Bits auf jeden Fall gesetzt (oder gelöscht). Es lassen sich ein paar Regeln für das Verknüpfen von Zahlen aufstellen:

* Nach A OR B gilt: A >= B, da alle gesetzten Bits von B in A enthalten sind.
* A XOR A = 0, da zwei gleiche Bits bei der XOR-Verknüpfung 0 ergeben.
* ????b XOR 1111b = NOT ????b, da in beiden Fällen alle Bits **invertiert** (umgedreht) werden.

Bitaritmethik

Neben den Bitverknüpfungen gibt es noch Schiebeoperationen: Das Schieben nach rechts SHR (shift right), und das Schieben nach links SHL (shift left). Hierbei werden alle Bits einer Zahl um eine angegebene Anzahl von Stellen nach links/rechts geschoben, wobei an der anderen Seite Nullen eingefügt werden.

BEISPIEL: 00101111b -- SHL 3 $\Rightarrow$ 00101111000b
01011000b -- SHR 2 $\Rightarrow$ 00010110b

Nach einer Analyse der eingesetzten Zahlen und der Ergebnisse ergibt sich folgendes:

LEMMA: Das Schieben einer Zahl nach links (rechts) um x Bitstellen entspricht einer Multiplikation (Division) mit (durch) 2^x.

Hexadezimale Zahlen

Da jede Ziffer einer hexadezimalen Zahl 16 Werte haben kann, lassen sich diese mit 4 Bits darstellen: 0h-0Fh können durch 0000b-1111b angegeben werden. Eine x-stellige Hex-Zahl erfordert also maximal (x*4) Bits zur Angabe im Binärformat.
BEISPIEL: 6Ch = 01101100b, denn 6h=0110b und 0Ch=1100b

ÜBUNGEN:
1. Berechnen Sie: 50 OR 07Bh, 30 XOR 12, NOT 54, 18 AND 0Fh.
2. Wandeln Sie folgende Zahlen von Binär in Hex bzw. umgekehrt um: 0654h, 0001001010111101b, 0FFh.
3. Beweisen Sie das Lemma!

Programmieraufgaben

1) Rechenprogramme für die Ganzzahltypen
2) N-Damenproblem
3) Potenzen berechnen
4) Prozeduren aus Vier Gewinnt, 3DTTT oder Gobang in Assembler umschreiben
5) Solitär in Assembler
6) Kreuzzahlenrätsellöser
7) Grafikprogrammierung Poincaré oder Chaosbilder

Der Maschinensprache - Befehlssatz des 8086 / 80386

Der **MOV**-Befehl

Allgemeine Syntax: MOV ZIEL,QUELLE

Ist die Quelle eine Konstante, so wird deren Größe automatisch an die Größe des Zieloperanden angepaßt.

```
Beispiel:   MOV AH , 23h   ; 23h ist vom Datentyp BYTE. (8 Bit)
            MOV AX , 23h   ; 23h ist vom Datentyp WORD. (16 Bit)
            MOV EAX , 23h ; 23h ist vom Datentyp DWORD. (32 Bit)
```

Natürlich kann eine Konstante niemals als Ziel eines MOV-Befehls verwendet werden. Sie kann außerdem nur in allgemeine Register geladen werden. Unmöglich ist daher ein Befehl wie „MOV ES , 0B800h“.

Ist die Quelle ein Register, so müssen beide Operanden die gleiche Größe besitzen.

```
Beispiel:   MOV AH , BL    ; AH bekommt den Inhalt von BL zugewiesen
            MOV DI , BP    ; DI := BP
```

Dementsprechend ist eine Kombination wie „MOV AX , BL“ mit dem MOV-Befehl nicht möglich, auch wenn es vom Programmierer gewünscht wird, daß die 8Bit-Zahl in BL zu einer 16Bit-Zahl in AX verwandelt wird. Ab dem 80386 gibt es entsprechende (sehr nützliche!!) Befehle hierfür (MOVSX und MOVZX),.

```
            MOVSX AX,BL ; Kopie unter Vorzeichenerweiterung
            MOVSX ECX,BX

            MOVZX DX,AL ; Kopie unter Erweiterung mit Null
```

Ist die Quelle eine Speichervariable, so müssen auch hier beide Operanden übereinstimmen.
Beispiel: Gegeben sind zwei Variablen: VAR B: BYTE; W: WORD;

```
            MOV AH , B     ; AH bekommt den Inhalt von B zugewiesen
            MOV W , CX     ; Die Variable W bekommt den Inhalt von CX zugewiesen.
```

Hierbei ist ein Mischen jedoch nicht erlaubt: „MOV AX , B“ ist illegal, da B auf eine Variable mit der Größe BYTE zeigt, und eine Typkonvertierung dieser Speicherstelle unmöglich ist.

```
Beispiel:   VAR  MEIN_ES, MEIN_CS: WORD;
            MOV MEIN_ES , ES      ; Inhalt von ES in einer Variablen sichern
            MOV MEIN_CS , CS      ; dito mit CS
```

Ein Speicher-Speicher-Transfer ist grundsätzlich nicht möglich!

ADRESSIERUNGSARTEN: Wie setzt der Compiler nun einen Befehl „MOV AX , W“ um? Der Prozessor hat keine Kenntnis von einer Variablen mit dem Namen W (sie könnte auch ganz anders heißen). Nach der Übersetzung sieht der

Befehl so aus: MOV AX , [0021h]. Die eckige Klammer kann als „der Speicherinhalt von“ übersetzt werden, und 0021h ist der Offset der Variablen W im Datensegment des Programms. Beim Speicherzugriff verwendet der Prozessor nun diesen Offset zusammen mit dem Inhalt des EDS-Registers, um die absolute Adresse zu bilden. Auch Programmierer können auf diese Art Befehle benutzen.

```
Beispiel:   MOV [1234h] , AX     ; Speichert AX an der Stelle EDS:1234h
            MOV BX , [5678h]     ; „BX := MEMW[ EDS:5678h ]“
```

Standardmäßig wird immer das EDS-Register zur Bildung der absoluten Adresse benutzt. Ein Programmierer kann aber jedes andere Segmentregister benutzen (auch ECS), er muß es nur angeben.

```
Beispiel:   MOV AX , ECS:[02h]   ; „AX := MEMW[ ECS:0002 ]“
```

Als letzte (aber sehr nützliche) Kombination gibt es auch die Möglichkeit, Register einzusetzen.

```
Beispiel:   MOV AX , [EBX]              ; AX := MEMW [ DS:EBX ]
```

In diesem Beispiel enthält EBX den Quelloffset, dessen Speicherinhalt nach AX geladen wird. Bei der Nutzung dieser komfortablen Adressierung gibt es sehr einfache Kombinationsmöglichkeiten:

```
Erlaubt sind:  - ein Basisregister      ( entweder EAX, EBX, ECX, EDX oder EBP )
               - ein Indexregister      ( entweder ESI oder EDI )
               - eine Konstante
Beispiele:  MOV AH, [EBX + EDI]
            MOV CX , [ESI]
            MOV EDX , [EBP + 234]
```

Als Offset wird jedesmal der errechnete Klammerinhalt benutzt. Ist das EBP-Register als Basisregister vorhanden (in einer Klammer), so wird standardmäßig das ESP-Register bei der Bildung der absoluten Adresse benutzt, was aber durch ein Segment-Präfix geändert werden kann.

ZUSATZOPERATIONEN: Jeder Compiler bietet die beiden Operatoren SEG und OFFSET an.

```
Beispiel:   MOV EAX , SEG W      ; EAX bekommt den Segmentwert der Adresse von W
            MOV EBX , OFFSET W   ; EBX bekommt den Offsetwert der Adresse von W
```

Auch hier wird der Ausdruck „OFFSET VARIABLEN_NAME“ wieder durch eine Konstante ersetzt. Gleiches gilt für den Segmentanteil. Eine Variable W kann demnach sehr einfach auf 5 gesetzt werden:

```
            MOV EBX , OFFSET W   ; BX bekommt den Offsetwert von W
            MOV [EBX] , 5        ; Die Speicherstelle von W bekommt den Wert 5
```

Der **XCHG**-Befehl

Allgemeine Syntax: XCHG op1 , op2 op1/2 = reg/mem

Dieser Befehl vertauscht die beiden Operanden op1 und op2. Segmentregister dürfen nicht benutzt werden. Auch ein Speicher-Speicher-Transfer ist nicht möglich. Natürlich müssen op1 und op2 die gleiche Größe haben. Auch unsinnige Kombinationen wie „XCHG CX , CX“ sind möglich.

Der **NOP**-Befehl

Dieser Befehl macht gar nichts (no operation). Er wurde von den Entwicklern bei INTEL nach dem Wunsch verschiedener Programmierer definiert, ist aber völlig identisch mit dem Befehl „XCHG AX , AX“. INTEL hat lediglich ein zusätzliches Mnemonik für einen bestehenden Opcode definiert.

Aufgabe: Schreiben Sie ein Delphiprogramm, welches den Inhalt zweier Variablen vertauscht.
VAR X, Y: DWORD;
Benutzen Sie dafür maximal **3** Assemblerbefehle.

Die Befehle **PUSH** und **POP**

Allgemeine Syntax: PUSH reg / POP reg

Mit diesen Befehlen wird der Inhalt eines Registers auf den Stapel gebracht (PUSH reg), bzw. ein Register mit dem obersten Stapelwert geladen (POP reg). Beide Befehle funktionieren mit allen allgemeinen Registern sowie den Segmentregistern.

```
Beispiel:   PUSH EAX     ; EAX auf den Stapel bringen
            POP ECX      ; ECX mit Stapelwert laden ( ECX := EAX)
```

Zusätzlich gibt es noch die Befehle **PUSHAD** und **POPAD**, welche keine Parameter erwarten, sondern jeweils mit den Allzweckregistern arbeiten. Analog arbeiten die Befehle **PUSHFD** und **POPFD** mit dem Flagregister.

Die Befehle **INC** und **DEC**

Allgemeine Syntax: INC reg/mem , DEC reg/mem

Diese Befehle verringern(DEC)/erhöhen(INC) den angegebenen Operanden um den Wert 1. Sie verändern die Flags im Flagregister: Ist nach der Ausführung von „DEC AX“ der Inhalt von AX 0, so ist das ZF (ZERO FLAG) gesetzt (=1), sonst ist es immer gelöscht (=0). Beide Befehle funktionieren auch mit Speicheroperanden.

Der **MUL**-Befehl

Allgemeine Syntax: MUL reg/mem

Mit dem MUL-Befehl können zwei Werte multipliziert werden. Je nach Größe des Operanden handelt es sich dabei um eine BYTE-, eine WORD- oder eine DWordMultiplikation. Bei der BYTE-Multiplikation wird der übergebene Faktor mit AL multipliziert, bei der WORD-Variante mit AX, bei Dword mit EAX. Das Produkt steht danach in AX bzw. als 32Bit-Zahl in DX+AX, wobei DX die oberen 16 Bit enthält; bei DWord steht das Ergebnis in EDX : EAX, wobei EDX die oberen 32 Bits enthält.

```
Beispiel:   MUL BL      ; Multipliziert BL mit AL, Ergebnis steht nun in AX
            MUL DI      ; Multipliziert DI mit AX, Ergebnis steht danach in DX : AX
                        (die oberen 16 Bits stehen in DX, die unteren 16 Bits in AX)
            MUL EBX     ; Multipliziert EBX mit EAX, Ergebnis steht danach in EDX : EAX
                        (die oberen 32 Bits stehen in EDX, die unteren 32 Bits in EAX)
```

Auch ein Quadrieren mit MUL AL ist problemlos möglich. Leider ist dieser komfortable Befehl sehr langsam. Eine Multiplikation mit Zweierpotenzen sollte daher durch eine **Schiebeoperation** ersetzt werden. Daneben gibt es noch die Variante **IMUL**, welche eine vorzeichenbehaftete Multiplikation durchführt, da **MUL** keine Vorzeichen beachtet.

Der **DIV**-Befehl

Allgemeine Syntax: DIV reg/mem

Dieser Befehl führt eine vorzeichenlose Division durch, wobei der angegebene Operand der Divisor ist. Ist er 0, so wird der INT 0 als Fault aufgerufen. Wie bei der Division gibt es zwei Versionen des Befehls, wobei die Größe des angegebenen Divisors entscheidend ist.

```
Beispiel:DIV BL     ; Entspricht „AX / BL“. Ergebnis: AL=AX DIV BL und AH=AX MOD BL
         DIV BX     ; = „DX:AX / BX“. Ergebnis: AX=(DX:AX) DIV BX und DX=(DX:AX) MOD BX
         DIV EBX    ; = „EDX:EAX / EBX“. Ergebnis: EAX=(EDX:EAX) DIV BX und EDX=(DX:AX) MOD BX
```

Eine häufige Fehlerquelle bei der Angabe eines 16(32)Bit-Divisors ist ein undefinierter Inhalt von DX(EDX). Da dieses Register mit benutzt wird (siehe Beispiel), **muß** es vorher auf **0** gesetzt werden, falls es sonst nicht gebraucht wird.
Zusätzlich gibt es noch den Befehl **IDIV**, welcher eine vorzeichenbehaftete Division durchführt.

Die Befehle **ADD** und **SUB**

Allgemeine Syntax: ADD/SUB Ziel,Quelle

Diese Befehle führen eine Addition/Subtraktion durch. Als Quelle kann eine Konstante angegeben werden

Der direkte Sprungbefehl **JMP**

Allgemeine Syntax: JMP Sprungmarke

Der JMP-Befehl sorgt für einen unmittelbaren Sprung zur angegebenen Stelle.

Der **CMP**-Befehl

Allgemeine Syntax: CMP Ziel,Quelle
Der CMP-Befehl (engl. compare) vergleicht beide Operanden miteinander, und setzt enige Bits im FLAG - Register. Meistens folgt danach ein indirekter Sprungbefehl.

```
Beispiel:   CMP AH , 5      ; Vergleicht AH mit 5.
```

Die indirekten Sprungbefehle **JCC**

Allgemeine Syntax: JCC Sprungmarke

Dieser Befehl führt einen Sprung in Abhängigkeit von bestimmten Bits im FLAG-Register aus.

Beispiele:
```
JA    jump if above
JB    jump if below
JC    jump if carry     (Springe wenn CF=1)
JE    jump if equal
JG    jump if greater
JP    jump if parity
JZ    jump if zero      (Springe wenn ZF=1)
JCXZ  jump if CX=0
```
Diese Befehle können auch kombiniert (z.B. JAE, JBE ...), oder durch ein eingefügtes N verneint werden (z.B. JNZ, JNGE ...). Auch existieren gleichwertige Kombinationen (z.B. JA=JNBE).

Die Befehle **AND**, **OR**, **XOR** und **NOT**

Allgemeine Syntax:
```
NOT reg/mem
AND/OR/XOR  Ziel,Quelle
```

Mit diesen Befehlen können Bit-Verknüpfungen durchgeführt werden. Der NOT-Befehl braucht nur einen Operanden, dessen sämtliche Bits dann invertiert werden. Natürlich kann man keine Konstante angeben, da diese ja nicht veränderbar ist. Die anderen Befehle akzeptieren als Quelle aber Konstanten.

Beispiel:
```
NOT AX          ; Invertiert alle Bits von AX (Bildet das Einerkomplement)
OR BX , CX
AND DL , 01     ; Verknüpfung mit einer Konstanten
```
Nach der Ausführung eines Befehls werden verschiedene Bits im Flagregister eingestellt. Speziell das ZF (Zero Flag) wird gerne von Programmierern abgefragt, da es anzeigt, ob das soeben veränderte Ziel den Wert 0 besitzt.

Beispiel:
```
AND DI,1        ; Bitverknüpfung
JZ @Sprungmarke
```
In diesem Beispiel werden durch die AND-Verknüpfung alle Bits in DI gelöscht, bis auf Bit 0. Wenn dieses gesetzt war, so ist nun DI=1, und das ZF gelöscht. Anderenfalls ist DI=0, ZF=1, und JZ springt zum angegebenen Ziel. Es wurde also eben geprüft, ob das Bit 0 in DI gesetzt war. Eine Variante des AND-Befehls ist der Befehl **TEST**, welcher wie AND arbeitet, das angegebene Ziel aber nicht verändert sondern lediglich die Bits im FLAG-Register setzt.

Der **LOOP**-Befehl

Allgemeine Syntax: LOOP Sprungmarke

Dieser Befehl ersetzt die Befehlskombination DEC ECX / JNZ Sprungmarken.

Die Befehle **CALL** und **RET**

Allgemeine Syntax: CALL Sprungmarke / RET

Der CALL-Befehl ruft eine Sprungmarke auf, indem der aktuelle IP-Wert auf den Stapel gebracht wird, und die Adresse der Sprungmarke nun nach IP geladen wird. Der RET-Befehl kann einfach mit „POP IP“ übersetzt werden, wobei es die eben genannte POP-Variante nicht gibt. Beide Befehle werden zusammen bei der Programmierung von Prozeduren verwendet.

Beispiel:
```
CALL @MEINE_PROZEDUR    ; Sprung zur angegebenen Marke
...                     ; Beliebige Anweisungen
@MEINE_PROZEDUR:        ; Sprungmarke
...                     ; Beliebige Anweisungen
RET                     ; Prozedurende: Rücksprung zur Anweisung nach CALL.
```
Ein häufiger Programmierfehler ist es, die RET-Anweisung zu vergessen. Auch kann hier sehr einfach ein Blick auf die Folgen geworfen werden, die entstehen, wenn eine Assemblerroutine innerhalb einer Delphi-Prozedur den Stapel nicht ordnungsgemäß hinterläßt: Es wird dann später einfach eine falsche Rücksprungadresse vom Stapel geholt, und der Computer stürzt ab!

Der integrierte Assembler

Der integrierte Assembler ermöglicht es, Intel-Assembler-Code direkt in Object-Pascal-Programme zu integrieren. Er implementiert eine umfangreiche Teilmenge der Syntax, die von Turbo Assembler und Macro Assembler von Microsoft unterstützt wird. Dazu gehören alle 8086/8087- und 80386/80387-Opcodes sowie einige der Ausdrucksoperatoren von Turbo Assembler. Zudem können Sie die Object-Delphi-Bezeicher in Assembler-Anweisungen verwenden.
Mit Ausnahme von DB, DW und DD (Define Byte, Word und Double Word) unterstützt der integrierte Assembler keine weiteren Direktiven von Turbo Assembler (z.B. EQU, PROC, STRUC, SEGMENT und MACRO). Operationen, die mit Turbo-Assembler-Direktiven implementiert werden, sind aber weitgehend mit äquivalenten Object-Delphi-Konstruktionen vergleichbar. Beispielsweise entsprechen die meisten EQU-Direktiven Konstanten-, Variablen- und Typdeklarationen, während die PROC-Direktive Prozedur- und Funktionsdeklarationen entspricht. Die STRUC-Direktive findet ihre Entsprechung in Datensatztypen.

Alternativ zur Verwendung des integrierten Assemblers können Sie OBJ-Datei hinzulinken, die als external deklarierte Prozeduren und Funktionen enthalten. Informationen hierzu finden Sie im Abschnitt external-Deklarationen.

Die Anweisung asm

Auf den integrierten Assembler greifen Sie über asm-Anweisungen zu, die folgende Syntax haben:

asm Anweisungsliste end

Dabei steht Anweisungsliste für eine Folge von Assembler-Anweisungen, die durch Strichpunkte, Zeilenendezeichen oder Object-Delphi-Kommentare voneinander getrennt werden.
Kommentare in einer asm-Anweisung müssen dem Object-Delphi-Stil entsprechen. Ein Strichpunkt besagt hier nicht, daß es sich beim Rest der Zeile um einen Kommentar handelt.
Das reserviert Wort inline und die Direktive assembler werden aus Gründen der Abwärtskompatibilität mitgeführt und haben keinerlei Auswirkung auf den Compiler.

CPU - Register

Im allgemeinen sind die Regeln für die Verwendung von Registern in einer asm-Anweisung identisch mit denjenigen für eine external-Prozedur oder -Funktion. In einer asm-Anweisung **muß** der Inhalt der Register EDI, ESI, ESP, EBP und EBX **erhalten** bleiben, während die Register EAX, ECX und EDX beliebig geändert werden können. Beim Eintritt in eine asm-Anweisung zeigt BP auf den aktuellen Stackframe, SP auf den Beginn des Stack, SS enthält die Segmentadresse des Stack-Segments und DS die Segmentadresse des Datensegments. Zu Beginn der Ausführung einer asm-Anweisung ist der Registerinhalt unbekannt. Eine Ausnahme bilden die Register EDI, ESI, ESP, EBP und EBX.

Die reservierten Symbole in der folgenden Tabelle bezeichnen CPU-Register.

32-Bit-**Allzweckregister**	EAX EBX ECX EDX	32-Bit-Zeiger- oder **Indexregister**	ESP EBP ESI EDI
16-Bit-Allzweckregister	AX BX CX DX	16-Bit-Zeiger- oder Indexregister	SP BP SI DI
Untere 8-Bit-Register	AL BL CL DL	16-Bit-Segmentregister	CS DS SS ES
32-Bit-Segmentregister	FS GS		
Obere 8-Bit-Register	AH BH CH DH	Coprozessor-Registerstapel	ST

Wenn ein Operand nur aus einem Registernamen besteht, wird er als Register-Operand bezeichnet. Als Register-Operanden können alle Register verwendet werden. Einige Register lassen sich auch in einem anderen Kontext einsetzen.
Die Basisregister (BX und BP) und die Indexregister (SI und DI) werden zur Kennzeichnung der Indizierung in eckigen Klammern angegeben. Folgende Kombinationen von Basis-/Index-Registern sind erlaubt: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] und [BP+DI]. Sie können auch mit allen 32-Bit-Registern indizieren, z.B. [EAX+ECX], [ESP] oder [ESP+EAX+5].

Die Segmentregister (ES, CS, SS, DS, FS und GS) werden unterstützt, aber Segmente werden normalerweise in 32-Bit-Anwendungen nicht verwendet.
Das Symbol ST bezeichnet das oberste Register im 8087-Gleitkommaregister-Stack. Jedes der acht Gleitkommaregister kann mit ST(X) referenziert werden, wobei X eine Konstante von 0 bis 7 ist, die den Abstand vom oberen Ende des Stack angibt.

Die Syntax für Assembler-Anweisungen

Die Syntax für eine Assembler-Anweisung lautet:

Label: Präfix Opcode Operand1, Operand2

Label repräsentiert einen Label-Bezeichner, Präfix einen Assembler-Präfix-Opcode(Operationscode), Opcode einen Assembler-Anweisungscode oder eine Direktive, und Operand steht für einen Assembler-Ausdruck. Label und Präfix sind optional. Es gibt Opcodes mit nur einem Operanden, während andere überhaupt keine Operanden haben.
Kommentare sind nur zwischen, nicht aber innerhalb von Assembler-Anweisungen erlaubt:

```
MOV AX,1 {Anfangswert}    { OK }
MOV CX,100 {Zähler}       { OK }
MOV {Anfangswert} AX,1;   { Fehler! }
MOV CX, {Zähler} 100      { Fehler! }
```

Label

Label werden in Assembler auf die gleiche Weise definiert wie in Object Pascal: Vor einer Anweisung wird ein Label und ein Doppelpunkt eingefügt. Obwohl es für Label keine Längenbeschränkung gibt, sind nur die ersten 32 Zeichen signifikant. Wie in Object Pascalmüssen auch in Assembler alle Label im label-Deklarationsabschnitt des Blocks definiert werden, der die asm-Anweisung enthält. Von dieser Regel gibt es eine Ausnahme: lokale Label.
Lokale Label beginnen immer mit dem Zeichen @. Sie setzen sich aus folgenden Zeichen zusammen: dem Zeichen @, gefolgt von einem oder mehreren Buchstaben, Ziffern, Unterstrichen oder @-Zeichen. Ein lokales Label ist auf asm-Anweisungen beschränkt. Der Gültigkeitsbereich eines lokalen Label erstreckt sich vom Schlüsselwort asm bis zum Schlüsselwort end in der asm-Anweisung, in der sich das Label befindet. Ein lokales Label braucht nicht deklariert zu werden.

Operanden

Operanden im integrierten Assembler sind Ausdrücke, die aus Konstanten, Registern, Symbolen und Operatoren bestehen. Die folgenden reservierten Wörter haben bei ihrer Verwendung in Operanden eine vordefinierte Bedeutung.

AH	BX	DI	EBX	ESP	OFFSET	SP	
AL	BYTE	DL	ECX	FS	OR	SS	
AND	CH	DS	EDI	GS	PTR	ST	
AX	CL	DWORD		EDX	HIGH	QWORD	TBYTE
BH	CS	DX	EIP	LOW	SHL	TYPE	
BL	CX	EAX	ES	MOD	SHR	WORD	
BP	DH	EBP	ESI	NOT	SI	XOR	

Reservierte Wörter haben immer Vorrang vor benutzerdefinierten Bezeichnern. Im folgenden Code-Fragment wird 1 nicht in die Variable CH, sondern in das Register CH geladen:

```
var
        Ch: Char;
        ...
asm
        MOV    CH, 1
end;
```

Wenn Sie auf ein benutzerdefiniertes Symbol zugreifen wollen, das den Namen eines reservierten Wortes trägt, müssen Sie den Operator & zum Überschreiben des Bezeichners verwenden:

```
MOV    &Ch, 1
```

Benutzerdefinierte Bezeichner sollten **möglichst** nie mit den Namen reservierter Wörter belegt werden.

Unterschiede zwischen Ausdrücken in Object Pascal und Assembler

Der größte Unterschied zwischen Object-Pascal-Ausdrücken und Ausdrücken des integrierten Assemblers besteht darin, daß alle Assembler-Ausdrücke einen konstantenWert ergeben müssen, d.h. einen Wert, der während der Compilierung berechnet werden kann. Beispielsweise ist für die Deklarationen

```
const
        X = 10;
        Y = 20;
var
        Z: Integer;
```

die folgende Assembler-Anweisung zulässig:

```
asm
        MOV     Z,X+Y
end;
```

Da X und Y Konstanten sind, ist der Ausdruck X + Y nur eine andere Möglichkeit zur Darstellung der Konstante 30. Die resultierende Anweisung bewirkt eine direkte Speicherung des Wertes 30 in der Word-Variablen Z. Wenn X und Y aber Variablen sind, kann der integrierte Assembler den Wert von X + Y nicht während der Compilierung berechnen:

```
var  X, Y: Integer;
```

In diesem Fall müßten Sie folgende Anweisung verwenden, um die Summe von X und Y in Z zu speichern:

```
asm
        MOV     EAX,X
        ADD     EAX,Y
        MOV     Z,EAX
end;
```

In einem Object-Pascal-Ausdruck wird eine Referenz auf eine Variable als Inhalt der Variablen interpretiert, im integrierten Assembler dagegen als Adresse der Variablen. Beispielsweise bezieht sich in Object Pascal der Ausdruck X + 4, in dem X eine Variable ist, auf den Inhalt von X + 4. Im integrierten Assembler bedeutet derselbe Ausdruck, daß sich der Inhalt des Word an einer Adresse befindet, die um vier Bytes höher ist als die Adresse von X. Dazu ein Beispiel:

```
asm
        MOV     EAX, X+4
end;
```

Obwohl dieser Code zulässig ist, würde er nicht den Wert von X + 4 in AX laden, sondern den Wert eines Word, das vier Bytes über X liegt. Um 4 zum Inhalt von X zu addieren, müssen Sie folgende Anweisung verwenden:

```
asm
        MOV     EAX,X
        ADD     EAX,4
end;
```

Konstanten

Der integrierte Assembler unterstützt zwei Konstantentypen: numerische Konstanten und String-Konstanten.

Numerische Konstanten

Numerische Konstanten müssen Integer-Zahlen sein, deren Wert im Bereich von -2147483648 bis 4294967295 liegt. Per Voreinstellung wird bei numerischen Konstanten die dezimale Notation verwendet. Der integrierte Assembler unterstützt aber auch die binäre, die oktale und die hexadezimale Notation (Basis 16). Zur Kennzeichnung der binären Notation wird der Zahl der Buchstabe B nachgestellt. In oktaler Notation steht nach der Zahl der Buchstabe O. Zur Kennzeichnung einer Hexadezimalzahl kann entweder nach der Zahl der Buchstabe H oder vor der Zahl das Zeichen $ stehen.

Numerische Konstanten müssen mit einer Ziffer von 0 bis 9 oder dem Zeichen $ beginnen. Wenn Sie eine hexadezimale Konstante mit dem Suffix H angeben und die erste signifikante Ziffer eine hexadezimale Ziffer zwischen A und F ist, müssen Sie eine zusätzliche Null an den Beginn der Zahl stellen. Beispielsweise handelt es sich bei 0BAD4H und $BAD4 um hexadezimale Konstanten, bei BAD4H aber um einen Bezeichner, weil der Ausdruck mit einem Buchstaben beginnt.

String-Konstanten

String-Konstanten müssen in halbe oder ganze Anführungszeichen eingeschlossen werden. Zwei aufeinanderfolgende Anführungszeichen, die vom selben Typ wie die umgebenden Anführungszeichen sind, werden als ein einzelnes Zeichen interpretiert. Hier einige Beispiele für String-Konstanten:

```
'Z'     'Delphi'  "Das ist alles, Leute "     '"Das war"s Leute," sagte er.'     '100'     ""
```

Symbole

Der integrierte Assembler ermöglicht den Zugriff auf nahezu alle Object-Pascal-Bezeichner in Assembler-Ausdrücken, einschließlich Konstanten, Typen, Variablen, Prozeduren und Funktionen. Außerdem ist im integrierten Assembler das spezielle Symbol @Result implementiert, das der Variable Result im Anweisungsteil einer Funktion entspricht. Die Funktion

```
function Sum(X, Y: Integer): Integer;

begin
        Result := X + Y;
end;
```

wird in Assembler folgendermaßen angegeben:

```
function Sum(X, Y: Integer): Integer; stdcall;

begin
        asm
        MOV     EAX,X
        ADD     EAX,Y
        MOV     @Result,EAX
        end;
end;
```

Die folgenden Symbole dürfen in asm-Anweisungen **nicht** verwendet werden:

Standardprozeduren und -funktionen.
Die speziellen Arrays Mem, MemW, MemL, Port und PortW.
String-, Gleitkomma- und Mengenkonstanten.
Label, die nicht im aktuellen Block deklariert sind.
Das Symbol @Result außerhalb einer Funktion.

Die folgende Tabelle faßt die Symbolarten zusammen, die in asm-Anweisungen verwendet werden können.

Symbol	Wert	Klasse	Typ
Label	Adresse des Label	Speicherreferenz	SHORT
Konstante	Wert der Konstante	Direkter Wert	0
Typ	0	Speicherreferenz	Größe des Typs
Symbol	Wert	Klasse	Typ
Feld	Offset des Feldes	Speicher	Größe des Typs
Variable	Adresse der Variablen	Speicherreferenz	Größe des Typs
Prozedur	Adresse der Prozedur	Speicherreferenz	NEAR
Funktion	Adresse der Funktion	Speicherreferenz	NEAR
Unit	0	Direkter Wert	0
@Code	Codesegment-Adresse	Speicherreferenz	0FFF0H
@Data	Datensegment-Adresse	Speicherreferenz	0FFF0H
@Result	Ergebnisvariablen-Offset	Speicherreferenz	Größe des Typs

Bei deaktivierter Optimierung werden lokale (also in Prozeduren und Funktionen deklarierte) Variablen immer auf dem Stack abgelegt. Der Zugriff erfolgt immer relativ zu EBP. Der Wert eines lokalen Variablensymbols besteht in seinem mit Vorzeichen versehenen Offset von EBP. Der Assembler addiert zu Referenzen auf lokale Variablen automatisch [EBP] hinzu. Beispielsweise wird für die Deklaration

```
var Count: Integer;
```

in einer Funktion oder Prozedur die Anweisung

```
MOV     EAX,Count
```

in MOV EAX,[EBP–4] assembliert.
Der integrierte Assembler behandelt einen var-Parameter immer als 32-Bit-Zeiger. Die Größe eines var-Parameters beträgt immer 4 Byte. Die Syntax für den Zugriff auf einen var-Parameter unterscheidet sich von derjenigen für einen Wert-

Parameter. Für den Zugriff auf den Inhalt eines var-Parameters müssen Sie zuerst den 32-Bit-Zeiger laden und dann auf die Position zugreifen, auf die er zeigt:

```
function Sum(var X, Y: Integer): Integer; stdcall;

begin
        asm
        MOV     EAX,X
        MOV     EAX,[EAX]
        MOV     EDX,Y
        ADD     EAX,[EDX]
        MOV     @Result,AX
        end;
end;
```

Bezeichner können in asm-Anweisungen qualifiziert werden. So lassen sich für die Deklarationen

```
type
        TPoint = record
        X, Y: Integer;
        end;
        TRect = record
        A, B: TPoint;
        end;
var
        P: TPoint;
        R: TRect;
```

die folgenden Konstruktionen für den Zugriff auf Felder in einer asm-Anweisung angeben:

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

Typbezeichner können zur einfachen und schnellen Konstruktion von Variablen verwendet werden. Alle folgenden Anweisungen erzeugen denselben Maschinencode, der den Inhalt von EDX in EAX lädt:

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect(EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

Ausdrucksklassen

Der integrierte Assembler unterteilt Ausdrücke in drei Klassen: Register, Speicherreferenzen und direkte Werte. Ausdrücke, die nur aus einem Registernamen bestehen, nennt man Registerausdrücke (z.B. AX, CL, DI oder ES). Registerausdrücke, die als Operanden verwendet werden, veranlassen den Assembler zur Erzeugung von Anweisungen, die auf die CPU-Register zugreifen.

Ausdrücke, die Speicheradressen bezeichnen, nennt man Speicherreferenzen. Zu dieser Kategorie gehören Label, Variablen, typisierte Konstanten, Prozeduren und Funktionen von Object Delphi.

Ausdrücke, bei denen es sich nicht um Register handelt und die auch nicht auf Speicheradressen zeigen, werden als direkte Werte bezeichnet. Zu dieser Gruppe gehören untypisierte Konstanten und Typbezeichner von Object Delphi.
Wenn direkte Werte und Speicherreferenzen als Operanden verwendet werden, führt dies zu unterschiedlichem Code. Ein Beispiel:

```
const Start = 10;
var Count: Integer;
        ...
asm
        MOV     EAX,Start          { MOV EAX,xxxx }
```

```
        EBX,Count               { MOV EBX,[xxxx] }
        MOV     ECX,[Start]     { MOV ECX,[xxxx] }
        MOV     EDX,OFFSET Count    { MOV EDX,xxxx }
end;
```

Da Start ein direkter Wert ist, wird das erste MOV in eine Move-Immediate-Anweisung assembliert. Das zweite MOV wird in eine Move-Memory-Anweisung übersetzt, weil Count eine Speicherreferenz ist. Im dritten MOV wird Start wegen der eckigen Klammern in eine Speicherreferenz umgewandelt (in diesem Fall handelt es sich um das Word mit dem Offset 10 im Datensegment). Im vierten MOV sorgt der Operator OFFSET für die Konvertierung von Count in einen direkten Wert (mit dem Offset von Count im Datensegment).

Die eckigen Klammern und der Operator OFFSET ergänzen einander. Die folgende asm-Anweisung erzeugt denselben Maschinencode wie die ersten beiden Zeilen der obigen Anweisung:

```
asm
        MOV     EAX,OFFSET [Start]
        MOV     EBX,[OFFSET Count]
end;
```

Bei Speicherreferenzen und direkten Werten findet eine weitere Unterteilung in verschiebbare und absolute Ausdrücke statt. Unter einer Verschiebung versteht man den Vorgang, bei dem der Linker Symbolen eine absolute Adresse zuweist. Ein verschiebbarer Ausdruck bezeichnet einen Wert, für den beim Linken eine Verschiebung (Relokation) erforderlich ist. Dagegen bezeichnet ein absoluter Ausdruck einen Wert, bei dem dies nicht nötig ist. In der Regel handelt es sich bei Ausdrücken, die ein Label, eine Variable, eine Prozedur oder eine Funktion referenzieren, um verschiebbare Ausdrücke, weil die endgültige Adresse dieser Symbole zur Compilierungszeit nicht bekannt ist. Absolut sind dagegen Ausdrücke, die ausschließlich Konstanten bezeichnen.

Im integrierten Assembler kann mit absoluten Werten jede Operation ausgeführt werden. Mit verschiebbaren Ausdrücken ist dagegen nur die Addition und Subtraktion von Konstanten möglich.

Ausdruckstypen

Jedem Assembler-Ausdruck ist ein bestimmter Typ (genauer gesagt eine bestimmte Größe) zugeordnet, weil der Assembler den Typ eines Ausdrucks einfach aus der Größe seiner Speicherposition abliest. Beispielsweise hat eine Integer-Variable den Typ vier, weil sie vier Byte Speicherplatz belegt. Der integrierte Assembler führt, wenn möglich, immer eine Typenprüfung durch. Es gibt Fälle, in denen eine Speicherreferenz untypisiert ist. Ein Beispiel hierfür ist ein direkter Wert, der in eckige Klammern gesetzt ist:

```
MOV     AL,[100H]       MOV     BX,[100H]
```

Der integrierte Assembler läßt beide Anweisungen zu, da der Anweisung [100H] kein Typ zugeordnet ist (sie bezeichnet einfach den Inhalt der Adresse 100H im Datensegment) und der Typ anhand des ersten Operanden feststellbar ist (Byte für AL, Word für BX). Falls sich der Typ nicht über einen anderen Operanden ermitteln läßt, verlangt der integrierte Assembler eine explizite Typumwandlung:

```
INC     BYTE PTR [100H]         IMUL    WORD PTR [100H]
```

Die folgende Tabelle enthält die vordefinierten Typensymbole, die der integrierte Assembler zusätzlich zu den aktuell deklarierten Object-Delphi-Typen bereitstellt.

Symbol	Typ
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Assembler-Prozeduren und –Funktionen

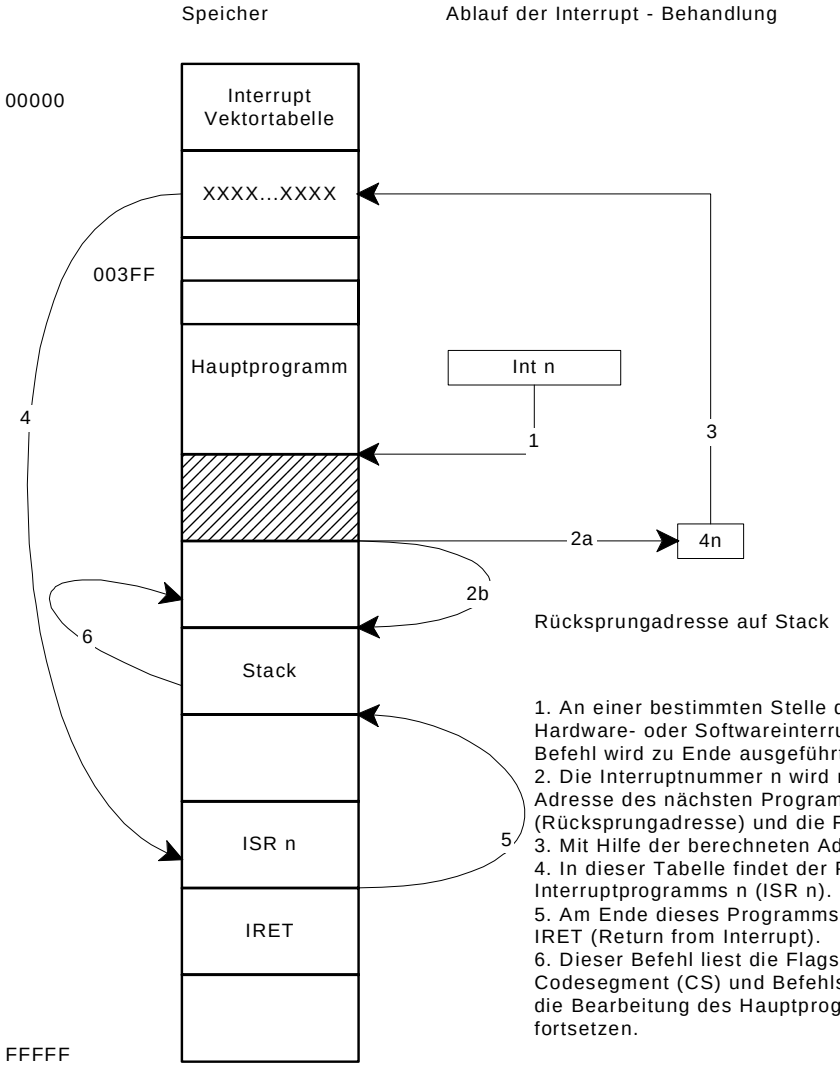
Mit dem integrierten Assembler können Sie komplette Prozeduren und Funktionen schreiben, für die keine begin...end -Anweisung erforderlich ist:

```
function LongMul(X, Y: Integer): Integer;

asm
        MOV    EAX,X
        IMUL   Y
end;
```

Der Compiler führt für diese Routinen verschiedene Optimierungen durch:

Der Compiler erstellt keinen Code zum Kopieren von Wert-Parametern in lokale Variablen. Dies betrifft alle Wert-Parameter vom Typ String und alle anderen Wert-Parameter, deren Größe nicht ein, zwei oder vier Byte beträgt. Innerhalb der Routine müssen derartige Parameter als var-Parameter behandelt werden.
Der Compiler weist keine Funktionsergebnis-Variable zu, und eine Referenz auf das Symbol @Result ist ein Fehler. Eine Ausnahme bilden Funktionen, die eine Referenz auf einen String, eine Variante oder eine Schnittstelle zurückliefern. Die aufrufende Routine weist diesen Typen immer einen @Result-Zeiger zu.



1. An einer bestimmten Stelle des Hauptprogramms tritt ein Hardware- oder Softwareinterrupt auf. Der gerade bearbeitete Befehl wird zu Ende ausgeführt.
2. Die Interruptnummer n wird mit 4 multipliziert und die Adresse des nächsten Programmbefehls des Hauptprogramms (Rücksprungadresse) und die Flags werden auf den Stack gebracht.
3. Mit Hilfe der berechneten Adresse wird auf die IVT zugegriffen.
4. In dieser Tabelle findet der Prozessor die Einsprungadresse des Interruptprogramms n (ISR n).
5. Am Ende dieses Programms befindet sich ein Interruptrückkehrbefehl IRET (Return from Interrupt).
6. Dieser Befehl liest die Flags und die Rückkehradresse vom Stapel in Codesegment (CS) und Befehlszeiger (IP). Damit kann der Prozessor die Bearbeitung des Hauptprogramms hinter der Unterbrechungsstelle fortsetzen.