1 Assembler-Befehle – Oder: was macht ein Compiler mit »I := 0«?

Als Hochsprachenprogrammierer – egal, ob man nun in C++, Delphi oder den anderen hoch spezialisierten oder etwas angestaubten Programmiersprachen programmiert, ja selbst unter dem interpretierenden Basic ist das so – macht man sich keine Gedanken, was eigentlich im Herz des Computers abläuft, wenn man eine so simple Zuordnung einer Konstanten, hier »0«, an eine Variable, hier »I«, programmiert. Muss man auch nicht! Wichtig ist lediglich, dass man weiß, dass irgendwo in den Katakomben des Rechners ein kleines Stückchen dotiertes Silizium reserviert ist, das man dazu benutzen kann, ihm vorübergehend einen Namen (»I«) und einen Wert (»0«) zu geben, und das bereitwillig die Information wieder abgibt, so man es unter dem eben vergebenen Namen mit dem entsprechenden Hochsprachenbefehl dazu auffordert. Wie das dann in eine Form gebracht wird, die der Prozessor dann auch verstehen und entsprechend umsetzen kann, interessiert bereits nicht mehr: Das ist Sache der Compiler – wozu hat man die sonst? Hochsprachenprogrammierer haben ihr Augenmerk auf die drei wesentlichen Kernpunkte gerichtet, die jedem Programm gemein sind: Problem – Lösungsansatz – Realisierung. Und dies spielt sich hauptsächlich auf einer Ebene ab, die Spielwiese der modernen Hochleistungscompiler von heute ist. Details stören hier nur!

Doch unter bestimmten Gesichtspunkten wird es dann notwendig, tiefer hinabzusteigen in die Tiefen der Hardware und ihrer Programmierung. Und plötzlich, als hebe sich ein Vorhang, ist der Fokus ein ganz anderer. Plötzlich sind solche Fragen wichtig wie »Bearbeite ich die Fließkommazahl *F* nun in den FPU-Registern oder besser skalar in den XMM-Registern?« oder »Kann ich diesen bedingten Sprung irgendwie vermeiden? Er mindert die Performance!«

Moderne Prozessoren von heute bestehen, betrachtet man die Situation aus einem bestimmten Blickwinkel, aus drei Einheiten: der *Central Processing Unit* (CPU), deren Aufgabe im Bereich der Verarbeitung von Integer-Daten liegt (was an dieser Stelle ganz allgemein gehalten werden soll: Auch die Befehlsinstruktionen, mit denen der Prozessor arbeitet, sind Integer-Daten, Bytes genannt!), der *Floating Point Unit*, zuständig für Fließkomma-Berechnungen, und den Komponenten, die für Multimedia-Anwendungen erforderlich sind (SIMD). Alle diese drei Bereiche können als (mehr oder weniger) unabhängige, selbstständige Einheiten betrachtet und besprochen werden.

1.1 CPU-Operationen

CPU-Datenformate

Wenn Sie von der Hochsprachenprogrammierung herkommen, vergessen Sie bitte ab jetzt alle Datendefinitionen, die Ihnen dort untergekommen sind. Der Hintergrund ist ein einfacher: Jede Hochsprache und jede neue Version einer Hochsprache definiert Daten nach Kriterien, die im Rahmen der Hochsprache und ihren Randbedingungen Sinn machen, die aber im Rahmen des Assemblers nicht immer nachvollzogen werden können.

Ein Beispiel: Unter Delphi 2.x war die gute, alte *Integer* definiert als vorzeichenbehaftete Ganzzahl, der 16 Bit zur Codierung zur Verfügung standen. Diese 16 Bit resultierten aus der Breite der damals verwendeten Prozessorregister einerseits und dem darauf aufbauenden (16-Bit-)Betriebssystem andererseits. So konnte diese Integer-Werte zwischen -32.768 und +32.767 annehmen. Mit Aufkommen der 32-Bit-Prozessoren und den entsprechenden Betriebssystemen konnten nun vorzeichenbehaftete Ganzzahlen zwischen -2^{32} und $+2^{32}$-1 verwaltet werden. Und so wurden diese neuen 32-Bit-Ganzzahlen schnell zum neuen »Standard« erklärt. Damit nun die Portierung der »alten« 16-Bit-Programme in die »neue« 32-Bit-Umgebung möglichst schnell und problemlos erfolgen konnte, wurde kurzerhand in Delphi 3.x die *Integer* als vorzeichenbehaftete 32-Bit-Ganzzahl umdefiniert und damit der *LongInt* gleichgesetzt. (Und ich bin sicher: Mit Einführung des 64-Bit-Prozessors Itanium von Intel, dem bereits avisierten 64-Bit-Betriebssystem von Microsoft – wohl auch mit Namen Windows – und somit einer neuen Runde an Software-Updates ist dann unter Delphi X.x die *Integer* eine 64-Bit-Ganzzahl.) Der Rest war einfach: Die reine Neu-Compilierung des Quelltextes führte nun (zumindest theoretisch! Die Tücke lag wie immer im Detail.) zu einem vollständig kompatiblen 32-Bit-Programm. Ohne dass eine Befehlszeile (zumindest was die Integers be-

trifft) geändert werden musste. Denn nun lud die CPU den Wert 4711 nicht mehr als 16-Bit-Integer in ein 16-Bit-Register, sondern als 32-Bit-Integer in ein 32-Bit-Register!

Diesen unter den genannten Bedingungen sicherlich sinnvollen »Modeerscheinungen« kann der Assembler nicht folgen. So kennt er noch nicht einmal die Unterscheidung zwischen vorzeichenbehafteten und vorzeichenlosen Integers: Für ihn gibt es, abgeleitet von den Daten, die die Prozessoren kennen, nur Byte-Daten (*define bytes*; DB), Word-Daten (*define words*; DW), DoubleWord-Daten (*define double words*; DD) und QuadWord-Daten (*define quad words*; DQ), die man definieren kann. Ob die vorzeichenbehaftet sind, interessiert weder den Prozessor noch den Assembler – hier ist die Interpretationsfähigkeit des Programmierers gefragt. Wir werden darauf noch zurückkommen. Um aber das Lesen dieses Buches nicht zu einer Gewalttour zu machen, werden seit langem eingeführte und probate Datenformate verwendet. Es sind im Falle von vorzeichenlosen Ganzzahlen die *Bytes* (8 Bits), *Words* (16 Bits), *DoubleWords* (32 Bits) und *QuadWords* (64 Bits) sowie im Falle der vorzeichenbehafteten Ganzzahlen die *ShortInts* (7 Bits + Vorzeichen), die *SmallInts* (15 Bits + Vorzeichen) und die *LongInts* (31 Bits + Vorzeichen). Da die zu den QuadWords analogen *QuadInts* (63 Bits + Vorzeichen) noch nicht aufgetaucht sind (die CPU-Register sind »nur« 32 Bits breit!), gibt es diese Integer zurzeit nur im Rahmen von SIMD (siehe unten). Diese Daten werden in diesem Buch als »Elementardaten« bezeichnet. Es kommen noch die einfachen und gepackten BCDs hinzu – worum es sich hier handelt, entnehmen Sie bitte genauso wie weitere Einzelheiten über die Darstellung der genannten Daten dem Kapitel »Datenformate« auf Seite 778.

Bitte beachten Sie auch, dass der Begriff »Integer« mehrfach belegt ist: So dient er als Oberbegriff für alle vorzeichenbehafteten Ganzzahlen und darüber hinaus auch für alle Ganzzahlen schlechthin. Das ist zwar bedauerlich, resultiert jedoch aus dem englischen Sprachgebrauch (der üblicherweise nicht zwischen »signed integers« und »unsigned integers« unterscheidet) und sollte eigentlich aufgrund des jeweiligen Kontextes nicht zu Problemen führen.

CPU-Basis-Register

Zur Bearbeitung der eben besprochenen Daten besitzt der Prozessorchip Strukturen, die man gemeinhin als »Register« bezeichnet. Diese Register haben die unterschiedlichsten Aufgaben: Sie können Daten mit logischen oder arithmetischen Instruktionen bearbeiten, sie können Informationen über den aktuellen Zustand des Prozessors darstellen oder

Informationen entgegennehmen, die die Aktivitäten des Prozessors steuern, oder sie können Adressen und Indices aufnehmen, die bei der Kommunikation mit der Peripherie des Prozessors eine Rolle spielen.

Gemäß ihrer Aufgabe sind die Register des Prozessors eingeteilt in die

- Allzweckregister, die die Operanden für die arithmetischen oder logischen Operationen aufnehmen oder Zeiger, die bei gewissen Befehlen eine Rolle spielen, über die die Kommunikation mit der Peripherie erfolgt. Die modernen Prozessoren der Pentium-4-Familie (und deren Klone) besitzen acht solcher Register.
- Segmentregister, die beim Datenaustausch des Prozessors mit seinem Speicher zum Tragen kommen. Die Pentium-4-Prozessoren besitzen sechs dieser Register.
- Programm-Status- und -Kontroll-Register. Sie dienen der Steuerung des Programmablaufs sowie der Darstellung des aktuellen Programmzustands. Pentium-4-Prozessoren haben ein solches Register.
- Register, die die Adresse des nächsten auszuführenden Befehls im Programmablauf beinhalten. Prozessoren der Pentium-4-Familie besitzen ein solches *instruction pointer register*.

Abbildung 1.1 zeigt Ihnen die Basisregister eines Pentium 4:

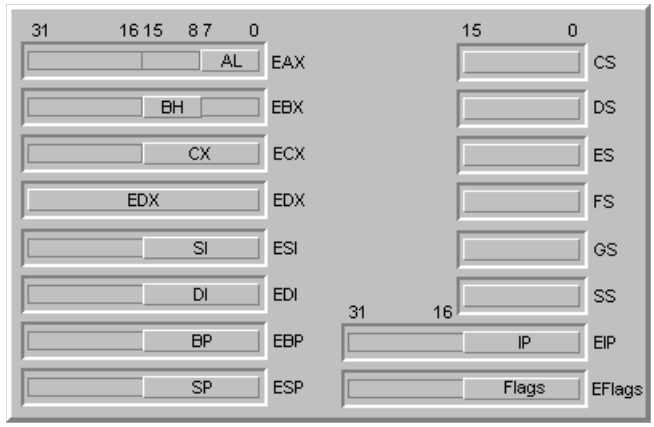

Abbildung 1.1: Die grundlegenden Register der CPU: Allzweck-, Segment-, Adressierungs- und Status-Register

Auf der linken Seite der Abbildung sind die acht 32-Bit-Allzweckregister dargestellt, die rechte zeigt die sechs 16-Bit-Segmentregister sowie das 32 Bit breite Status- und Kontrollregister EFlags und das ebenfalls 32 Bit breite »Befehlszeiger«-Register EIP.

Allzweckregister

Die Namen der Allzweckregister stammen traditionell noch aus der Zeit, in denen sie für bestimmte Aufgaben spezialisiert und lediglich 16 Bit breit waren. So ist der *Extended Accumulator* EAX aus dem *Accumulator* AX entstanden, dessen Hauptaufgabengebiet die arithmetischen Operationen waren. Das *Extended Base register* EBX entsprang dem *Base register* BX und diente als Heimat einer Basisadresse, die bei der indirekten Adressierung eine Rolle spielte (vgl. das Kapitel »Speicheradressierung«). ECX, das *Extended Counter register* diente in Form seines Vorläufers, des *Counter registers* CX, hauptsächlich der Steuerung von Programmschleifen, während das *Data register* DX, das dem *Extended Data register* EDX zugrunde liegt, zusätzliche Daten aufnahm, die entweder während verschiedener Zwischenstufen einer Berechnung entstanden oder im Rahmen verschiedener Instruktionen benötigt wurden.

Heute gibt es diese Unterscheidung nicht mehr – zumindest was die meisten Fähigkeiten betrifft. Alle acht Register, also EAX, EBX, ECX und EDX sowie ESI, EDI, EBP und ESP, sind absolut gleichberechtigt und können beliebig ausgetauscht und zu allen nur denkbaren Operationen (Arithmetik, Berechnung indirekter Adressen, Zeiger auf Speicherstellen) verwendet werden. Doch es existieren zwei Register, die mehr oder weniger als tabu gelten und für ganz bestimmte Zwecke eingesetzt werden: das *Extended Base Pointer register* EBP und das *Extended Stack Pointer register* ESP. Sie dienen der Verwaltung einer Datenstruktur, auf die der Prozessor häufig zurückgreift und ohne die gar nichts läuft: des Stacks. Was es damit auf sich hat, entnehmen Sie bitte dem Kapitel »Stack« auf Seite 385. Arbeiten Sie daher mit diesen Registern unter allen Umständen nur dann, wenn Sie genau wissen, was Sie tun!

Dennoch gibt es auch heute noch Spezialaufgaben für bestimmte Register, die andere Register nicht übernehmen können: So ist Kommunikation mit der Peripherie über Ports auch heute nur mit dem EDX- und EAX-Register möglich: EDX enthält die Adresse des Ports und EAX sendet oder empfängt das Datum. Auch können einige Befehle auf die Zusammenarbeit mit dem Akkumulator EAX hin optimiert sein und laufen mit diesem Register ggf. schneller ab als mit anderen Allzweckregistern. Und auch die letzten beiden der acht Allzweckregister, das *Extended Source Index register* ESI und das *Extended Destination Index register* EDI werden üblicherweise für bestimmte Zwecke reserviert: Sie spielen bei so genannten String-Befehlen eine entscheidende Rolle. Wir werden in diesem Kapitel noch darauf zu sprechen kommen.

Alias-Namen

Die 32-Bit-*Exx*-Register sind die physikalischen Strukturen, mit denen die Prozessoren aus der Pentium-4-Familie arbeiten. Wie bereits mehrfach geäußert, sind sie evolutionär aus den 16-Bit-Pendants der Vor-80386-Prozessoren entstanden. Nicht nur aufgrund der Abwärtskompatibilität zu diesen Prozessoren, sondern einfach auch aus praktischen Gründen gibt es jedoch die »alten« Registernamen weiterhin: Mit ihnen können eben auch Words oder Bytes in DoubleWord-Registern gezielt bearbeitet werden. Daher können Sie auch heute noch die »Register« AX, BX, CX, DX, SI, DI, BP und SP ansprechen! Allerdings stehen sie nur noch für die jeweils »unteren« sechzehn Bits 0 bis 15 der physikalisch vorhandenen Exx-Pendants, sind also nicht viel mehr als Alias-Namen bestimmter Teile des korrespondierenden 32-Bit-Registers. Analoges gilt für die 8-Bit-»Register« AH, AL, BH, BL, CH, CL sowie DH und DL, die jeweils die »oberen« (= *high*) 8 Bits der alten 16-Bit-Register repräsentieren, also die Bits 8 bis 15, oder die »unteren« (= *low*), also die Bits 0 bis 7.

Abbildung 1.1 versucht das darzustellen: Für eine Operation seien lediglich die Bits 0 bis 7 des EAX-Registers notwendig. Daher wird der Instruktion als Operand das »Register« AL (*accumulator – low byte*) übergeben. Die Operation erfolgt nun genau mit den Bits dieses »Registers«, den Bits 0 bis 7 des EAX-Registers. Alle anderen Bits bleiben »unsichtbar« und werden nicht verändert! Eine weitere Operation benötigt die Bits 8 bis 15 aus Register EBX. Der Instruktion wird daher als Operand das »Register« BH (*base register – high byte*) genannt. Auch in diesem Fall wirkt sich die Operation ausschließlich auf die Bits 8 bis 15 des EBX-Registers aus, alle anderen bleiben unverändert. Wird dagegen das niedrigerwertige Word im ECX-Register benötigt, spricht man es über CX an. Mit EDX schließlich wird dann das real existierende 32-Bit-Register EDX angesprochen.

Es gibt nur die Möglichkeit, das »untere« Word eines Registers oder die dieses Word bildenden Bytes gezielt anzusprechen! So gibt es *keine* Alias-Namen für das »obere« Word (Bits 16 bis 31) oder die dieses bildenden Bytes (Bit 16 bis 23 bzw. 24 bis 31). Auch lassen sich byteweise nur die vier Allzweckregister EAX, EBX, ECX und EDX, nicht aber alle anderen Register ansprechen. Immerhin gibt es mit »IP« bzw. »Flags« auch die Alias-Namen für das jeweils »untere« Word der Register EIP und EFlags. Analoges gilt für (E)SI, (E)DI, (E)BP und (E)SP (vgl. Abbildung 1.1).

Interpretation

Die Alias-Namen für bestimmte Registerteile der Allzweckregister erwecken den Eindruck, dass das Stichwort »Interpretation« unter Assembler eine bedeutende Rolle spielt: Das »Register« AH wird als Feld von acht bestimmten Bits des Registers EAX, den Bits 8 bis 15, *interpretiert*.

Dieser Eindruck stimmt! Ein Grund für die Flexibilität des Assembler besteht darin, dass er in Wirklichkeit nur wenige grundlegende Strukturen kennt und Annahmen macht. Den Gesamtzusammenhang im Auge zu behalten und sinnvolle Befehle auf sinnvolle Daten anzuwenden, ist Ihre Sache! Ganz besonders deutlich wird dieser Sachverhalt, wenn man einmal ein Allzweckregister genauer betrachtet, nehmen wir z.B. EAX. Wie jeder weiß, arbeitet der Prozessor ja binär, was bedeutet, dass er nur die Zustände »0« und »1« kennt. Er arbeitet also bitorientiert. Wen wundert daher, dass die Allzweckregister diesem Sachverhalt Rechnung tragen und 32 Bits realisieren, wie Abbildung 1.2 es zeigt? (Wer sich bei der binären Darstellung eines Datums noch ein wenig schwer tut, sei auf Kapitel »Datenformate« ab Seite 778 verwiesen).

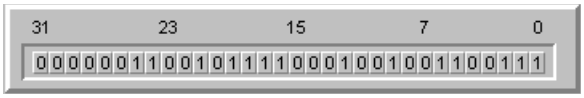

Abbildung 1.2: Binäre Darstellung eines DoubleWords mit dem dezimalen Wert 53.416.551

integer or not integer!

Doch was für ein Datum enthält EAX nun tatsächlich: Sind es 32 einzelne, von einander unabhängige Bits, die zwar gemeinsam in einer 32-Bit-Struktur gespeichert werden, die frappierend einem DoubleWord gleicht, die aber sonst wenig mit einander zu tun haben? Oder müssen diese 32 Bits im Zusammenhang gesehen werden, weil sie eine Zahl darstellen? In diesem Falle beinhaltete EAX die Ganzzahl 53.416.551.

signed or not signed!

Nächstes Problem: Ist das Datum tatsächlich eine Integer und kein Bit-Feld, erhebt sich die nächste Frage: Ist sie vorzeichenbehaftet oder nicht? Mit anderen Worten: Stellt Bit 31 das Vorzeichenbit einer Integer dar oder ist es deren höchste signifikante Stelle? Das ist, wenn man die Befehlsverarbeitung betrachtet, kein unwesentlicher Unterschied! Denn wäre in der Abbildung Bit 31 gesetzt, hätte die Zahl je nachdem, ob es ein Vorzeichen ist oder nicht, den Wert 2.200.900.199 (vorzeichenlos) bzw. -2.094.067.097 (mit Vorzeichen).

32, 16 oder 8 Bits?

Und noch ein Dilemma: Könnte es nicht sein, dass nur Teile des Registers eine Rolle spielen, da der vorangegangene Befehl einen der Alias-

Namen von oben verwendet hat? So könnte, wie Abbildung 1.3 das darstellt, der Wert »4711« (als Word) durch den vorangegangenen Befehl in das »Register« AX geschrieben worden sein und hätte damit ein anderes Datum überschrieben. Das bedeutet dann aber, dass die Bits 16 bis 31 des Registers EAX spätestens seit dem letzten Befehl Müll enthalten, der tunlichst künftig unberücksichtigt bleibt.

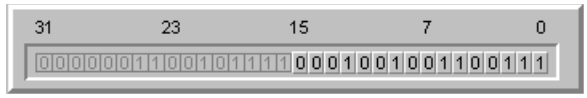

Abbildung 1.3: Binäre Darstellung eines Words mit dem dezimalen Wert 4711

Oder doch nicht? Haben diese Bits 16 bis 31 vielleicht trotz des »Überschreibens« eine Berechtigung? Denn immerhin könnten sie ja im Rahmen einer Adressberechnung durch eine Multiplikation eines Words (in den Bits 0 bis 15) mit der Konstanten 65.536 und damit ein DoubleWord als Resultat entstanden sein, zu der nun durch einfaches Überschreiben der vormals dort stehenden Nullen ein Offset addiert wird. Diese Art nicht nur der Adressenberechnung ist tatsächlich möglich, wir werden dies bei den entsprechenden Befehlen noch sehen!

nibble or not nibble!

Und schließlich: Betrachten wir einmal nur das niedrigstwertige Byte des Registers EAX, das über AL angesprochen werden kann. Zeigt der obere Teil in Abbildung 1.4 nun die Darstellung eines Bytes mit dem Wert 103, oder repräsentiert es eine *binary coded decimal*, eine BCD, mit dem Wert 7, wie es der untere Teil der Abbildung 1.4 nahe legt? (Falls Ihnen BCDs nicht geläufig sind, verweise ich auf den Abschnitt »Binary Coded Decimals« auf Seite 809.) Dann enthielten Bits 4 bis 7 wieder Müll!

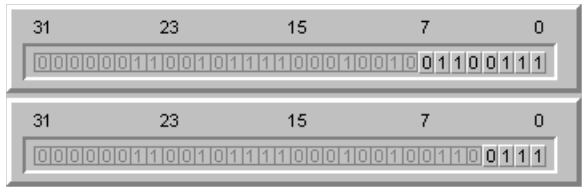

Abbildung 1.4: Binäre Darstellung eines Bytes mit dem dezimalen Wert 103 und einer BCD mit dem dezimalen Wert 7

Oder doch nicht – gibt es doch auch gepackte BDCs! Dann allerdings enthielte EAX die BCD 67 (vgl. oberer Teil der Abbildung). Wie und wodurch aber sollte die BCD 67 vom Byte 103 unterschieden werden?

Sie sehen, dass der Prozessor hier hoffnungslos überfordert wäre, müsste er diese Entscheidungen treffen. Denn mit welchen Daten Sie arbeiten – vorzeichenbehaftet oder vorzeichenlos, Ganzzahlen oder Bit-Felder, Binärdaten oder BCDs –, das weiß nur einer: Sie. Da Sie dem Prozessor diese Information aber nicht oder nur sehr eingeschränkt geben können, liegt es in Ihrer Verantwortung allein, die Ergebnisse von Berechnungen oder sonstigen Operationen korrekt zu *interpretieren*! Der Prozessor kann Ihnen hierbei nur helfen, indem er Ihnen signalisiert, was wäre, wenn eingegebenes Datum dieses oder jenes wäre. Und das tut er auch, wie wir bei der Besprechung der Flags gleich noch sehen werden. Die Entscheidungen treffen, was nun zu erfolgen hat, müssen jedoch Sie! Und dies unterscheidet Programmierung mit Assembler von Programmierung mit Hochsprachen. Denn in letzterer kann der Compiler meckern, wenn Sie versuchen, einem Byte eine Fließkommazahl zuzuordnen oder eine Routine mit einem Array als Parameter aufzurufen, die eine LongInt erwartet. Der Assembler kann das weniger stringent und lange nicht in dem Ausmaß, weil er, wie gesehen, z.B. nicht wissen kann, was in den Prozessorregistern für Daten hausen. *Assembler-Programmierung hat viel mit korrekter Interpretation dessen zu tun, was man sieht!*

Segmentregister

Die sechs Segmentregister enthalten Adressen, die beim Zugriff auf den Speicher eine wesentliche Rolle spielen. Sie sind auch nur für diesen Zweck nutzbar. Daher werden wir sie auch erst im Kapitel »Speicherverwaltung« ab Seite 394, wo es um die Speichersegmentierung geht, näher anschauen.

Das Segmentregister DS besitzt unter den Segmentregistern eine Sonderrolle, dient es doch bei Adressberechnungen zum Zugriff auf Daten als Standard-Bezugsregister. Die Nutzung der Register ES, FS oder GS zu diesem Zweck ist zwar möglich, verlangt aber einen so genannten *segment override prefix*, der ein zusätzliches Byte in der Instruktion darstellt und die Befehlsverarbeitung in den Pipelines entsprechend verzögert. Auch eine Sonderstellung nehmen die Segmentregister CS und SS ein, die für das Codesegment und den Stack reserviert sind.

Befehlszeiger

Der *instruction pointer* EIP bzw. sein 16-Bit-Alias IP werden lediglich der Vollständigkeit halber erwähnt. Ihnen als Programmierer ist ein Zugriff auf dieses Register vollständig verwehrt. Das Register untersteht ausschließlich der Kontrolle des Prozessors: Hier speichert er die Adresse

des nächsten auszuführenden Befehls im Programm. Der Inhalt des Registers wird vom Prozessor bei jeder Ausführung eines Befehls aktualisiert. So wird der Zeiger während der Befehlsdekodierung um die Anzahl Bytes erhöht, die der augenblicklich dekodierte Befehl zur Codierung benötigt. Oder es wird in ihn das Ziel eines Sprunges oder eines Unterprogrammaufrufs eingetragen.

Direkter Zugriff auf EIP (IP) ist auch nicht notwendig. Denn ein Schreiben in das EIP hätte ja zur Folge, dass der Prozessor an der eben eingeschriebenen Adresse mit der Programmausführung fortfahren soll. Das aber können Sie einfacher und komfortabler über die Sprung- oder Unterprogrammaufrufbefehle (JMP, Jcc, CALL) erreichen, die Ihnen die lästige und nicht einfache Adressberechnung abnehmen. Und sollten Sie wirklich einmal genötigt sein, die Position des nächsten Befehls im Programm zu erfahren – nichts anderes würden Sie ja durch das Auslesen von EIP erreichen –, so gibt es hierfür andere Möglichkeiten, z.B. über die Abfrage der aktuellen Position mittels eines vordefinierten Symbols des Assemblers.

EFlags-Register

Bleibt noch das EFlags-Register zu erklären. Dieses Register, entstanden aus dem 16-Bit-Flag-Register, ist (direkt) nur schwer zugänglich: Es gibt nur sehr wenige Befehle, die das EFlags-Register als Quelle oder Ziel einer Operation akzeptieren. Das Datum in EFlags wird in eindeutiger Weise interpretiert: als Feld von 32 Bits, wie Abbildung 1.5 zeigt:

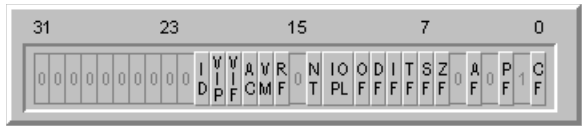

Abbildung 1.5: Speicherabbild des EFlag-Registers

Diese Bits sind vollständig unabhängig voneinander und beeinflussen sich gegenseitig nicht. Sie dienen drei Zwecken:

- Darstellung des derzeitigen Programmstatus (Condition Code),
- Steuerung gewisser Programmabläufe (Kontrollflags) und
- Darstellung bestimmter Systemparameter (Systemflags), die einen Einfluss auf die Funktion des Prozessors und das Betriebssystem haben.

Da diese Bits bestimmte Sachverhalte (entweder dem Programmierer oder dem Prozessor) signalisieren sollen, nennt man sie (der Schifffahrt entliehen) auch (Signal-)Flaggen oder »Flags«. Gemäß der drei genannten Aufgaben teilt man sie in Condition Code, Kontrollflags und Systemflags ein.

Wie Sie sehen können, sind nicht alle Flags definiert oder besser: dem Programmierer zugänglich. Die den grau dargestellten Bits 1, 3, 5, 15 und 22 bis 31 zugeordneten Flags gelten als reserviert und sollten tunlichst nicht angetastet werden. Das bedeutet, sie sollten nicht mit anderen als den jeweils aktuellen Werten belegt werden, wollen Sie unschöne Exceptions der Form »Allgemeiner Zugriffsfehler« vermeiden. Die oben gezeigten Nullen und Einsen sind die Standardwerte beim Pentium 4, andere Prozessoren können hier andere Werte haben.

Falls Sie also einmal Änderungen am Inhalt des EFlags-Register vornehmen müssen, die Sie nicht anders realisieren können – wir werden darauf zurückkommen –, so sollten Sie es zunächst auslesen, die Änderungen vornehmen und den geänderten Inhalt wieder zurückschreiben. Auf diese Weise stellen Sie sicher, dass die nicht zu verändernden Flags Prozessor-unabhängig den korrekten Standardwert enthalten.

Die wichtigsten und am häufigsten benutzten Flags sind die Statusflags (Abbildung 1.6, oben). Sie werden durch viele Instruktionen verändert oder dienen einigen Instruktionen als Input und signalisieren den Prozessorzustand nach einer arithmetischen Operation. Schon erheblich weniger häufig verwendet wird das einzige Kontrollflag (Abbildung 1.6, Mitte). Es hat lediglich bei den Stringbefehlen Wirkung und wird daher zusammen mit diesen besprochen. Mit den Systemflags (Abbildung 1.6, unten) werden Sie vermutlich selten in Berührung kommen. Sie spielen eine wesentliche Rolle bei der Verwaltung von sog. »Tasks« (NT, IOPL), in bestimmten Betriebsmodi des Prozessors (»virtual 8086 mode«; VIP, VIF, VM) sowie in speziellen Programmen (z.B. Debugger; TF, RF) oder zur Steuerung bestimmter Systemdienste (IF, AC) – Summa: Sie sind Sache des Betriebssystems oder sonstiger Spezialprogramme, die uns im Rahmen dieses Buches nicht interessieren. Daher werden wir sie an anderer Stelle besprechen.

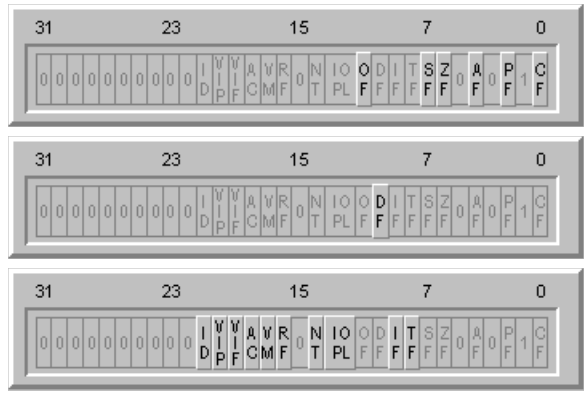

Abbildung 1.6: Status-, Kontroll- und Systemflags der CPU

ID-Flag Lediglich Bit 21, das System-Flag ID oder »*identification flag*«, könnte Sie interessieren: So können Sie anhand des Zustandes dieses Flags feststellen, ob der Prozessor über den äußerst wichtigen CPUID-Befehl verfügt. Interessant wird das aber nur bei Prozessoren vor dem Pentium (ja, die gibt's noch: mein alter 486 dient mir noch als Druckerserver!), da seither jedem Prozessor der Befehl CPUID implementiert wurde und künftig wohl auch wird.

Statusflags Die Statusflags sind genau die Hilfe, die Ihnen der Prozessor bei der Interpretation von Registerinhalten zur Verfügung stellt. Sie werden gebildet vom

- »carry flag« (CF). Dieses Flag ist ein sehr häufig und zu den verschiedensten Zwecken benutztes Flag. Seine eigentliche und Hauptaufgabe ist allerdings, einen Über- oder Unterlauf nach arithmetischen Operationen mit vorzeichenlosen Integers anzuzeigen. Es wird daher während einer Operation gesetzt, wenn z.B. die Addition zweier Daten den Wertebereich der verwendeten Daten überschreiten würde (z.B. bei Words: Überlauf von Bit 15 in das bei Words nicht vorhandene Bit 16) oder eine Subtraktion zweier Daten das untere Limit »0« unterschreiten würde (Unterlauf mit Borgen aus dem z.B. bei DoubleWords nicht vorhandenen Bit 32). Das carry flag nimmt sozusagen die Position des jeweils »fehlenden« Bits ein: Bit 32 bei DoubleWords, Bit 16 bei Words und Bit 8 bei Bytes.
- »parity flag« (PF). Dieses Flag wird immer dann gesetzt, wenn das niedrigstwertige Byte des Datums eine gerade Anzahl von gesetzten Bits hat, sonst wird es gelöscht. Bedeutung hat dieses Flag im Zusammenhang mit der Kommunikation über serielle Schnittstellen,

da ja Übertragungsprotokolle ebenfalls solche *parity bits* senden (können) und auf diese Weise recht schnell festgestellt werden kann, ob das empfangene Byte korrekt empfangen wurde (PF und gesendetes *parity bit* stimmen überein) oder nicht.

- »adjust flag«, auch »auxiliary carry flag« oder kurz »auxiliary flag« (AF). Dieses Flag kommt bei der BCD-Arithmetik zum Einsatz, da es wie das carry flag einen Über- oder Unterlauf anzeigt. Da BCDs einzelne Nibble (oder »half bytes«) und damit kleiner als die kleinste definierte Einheit (Byte) sind, kann das carry flag hier nicht die »Retterrolle« spielen; dies erfolgt durch das adjust flag: Es ist das bei BCDs nicht vorhandene »Bit 4«, in das oder aus dem ein Über-/Unterlauf erfolgt.
- »zero flag« (ZF). Es wird immer dann gesetzt, wenn das Ergebnis der Operation null ist, also kein Bit gesetzt ist. Andernfalls ist es gelöscht.
- »sign flag« (SF). Dieses Flag enthält, wie der Name schon vermuten lässt, fast immer das Vorzeichen des Ergebnisses einer Operation (Ausnahme im übernächsten Absatz!). Je nach eingesetztem Datum (ShortInt, SmallInt, LongInt) ist es eine Kopie des Bits 7, 15 oder 31 des Ergebnisses, das das Vorzeichen repräsentiert. Ist das sign flag gesetzt, signalisiert es ein negatives Vorzeichen, andernfalls ist das Datum positiv.
- »overflow flag« (OF). Dieses Flag ist das CF-Pendant für vorzeichenbehaftete Zahlen. Sobald das Ergebnis einer Operation nicht mehr im verwendeten Format (ShortInt, SmallInt oder LongInt) darstellbar ist, wird OF gesetzt, andernfalls gelöscht.

Achtung Falle! Das overflow flag signalisiert einen Übertrag in das/aus dem MSB, dem *most significant bit*. Bei LongInts handelt es sich hierbei wie bei DoubleWords um das Bit 31, bei SmallInts/Words um Bit 15 und bei ShortInts/Bytes um Bit 7. Während jedoch bei vorzeichenlosen Zahlen dieses MSB Teil der zur Zahlendarstellung verfügbaren Bits ist, repräsentiert es bei vorzeichenbehafteten Zahlen das Vorzeichen und besitzt somit im sign flag eine Kopie. Und dies führt zu Interpretationsproblemen, wenn der Wertebereich einer vorzeichenbehafteten Zahl über- oder unterschritten wird. Zur Illustration diene Abbildung 1.7:

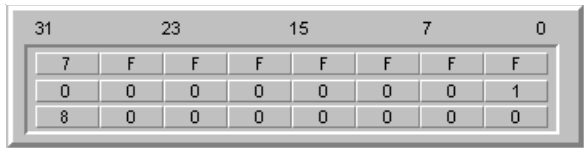

Abbildung 1.7: Darstellung eines Überlaufs nach Addition zweier vorzeichenbehafteter Zahlen

In der oberen Zeile ist (am Beispiel eines 32-Bit-DoubleWords) die größte positive, vorzeichenbehaftete Zahl dargestellt: $7FFF_FFFF. Addiert man zu dieser Zahl eine »1«, sieht man das Dilemma: Da $7FFF_FFFF, als vorzeichen*lose* Zahl interpretiert, noch nicht der Weisheit letzter Schluss ist, addiert der Prozessor dies brav zu $8000_0000. Denn schließlich ist das ja, vorzeichenlos interpretiert, auch korrekt. Damit ist aber Bit 31 und im Gefolge auch das sign flag gesetzt, was, vorzeichenbehaftet interpretiert, ein negatives Vorzeichen bedeutet. Damit steht im Register nun die kleinste negativ darstellbare Integer (*cave*: 2er-Komplement!). Das bedeutet: Was vorzeichenlos interpretiert absolut korrekt ist, ist vorzeichenbehaftet interpretiert falsch – die Überschreitung des positiven Wertebereichs führt zu einer negativen Zahl. Da der Prozessor nun nicht wissen kann, ob $7FFF_FFFF nun +2.147.483.647 (vorzeichenbehaftet) oder 2.147.483.647 (vorzeichenlos) ist, führt er die Addition so aus, als würden vorzeichenlose Zahlen verwendet. Um aber zu signalisieren, dass im Falle vorzeichenbehafteter Zahlen ein Überlauf stattgefunden hat (Übertrag von Bit 30 in das Vorzeichen-Bit 31!), setzt er OF. Das bedeutet: Ist OF gesetzt und gleichzeitig auch SF, so wurde, vorzeichenbehaftet interpretiert, durch die Operation der positive Wertebereich überschritten und SF zeigt das falsche, »entgegengesetzte«, hier also negative Vorzeichen an. Ist OF dagegen gelöscht, gibt SF das korrekte, hier positive Vorzeichen an.

Die gleiche Überlegung rückwärts zeigt auch den Sachverhalt an, wenn der negative Wertebereich unterschritten wird. Auch hier kann Abbildung 1.7 als Illustration herhalten: In der untersten Zeile steht die kleinste negative Zahl. Subtrahiert man von ihr »1«, so stellt sich aufgrund der für vorzeichenlose Zahlen korrekt durchgeführten Operation das in der obersten Zeile dargestellte Ergebnis ein. Dies ist analog der eben durchgeführten Betrachtung aber die größte positive Zahl. Somit spiegelt auch hier die Stellung des sign flag einen falschen Sachverhalt wider: Nach Subtraktion einer Zahl von der kleinsten negativen Zahl wird das Vorzeichenbit gelöscht, was »positiv« heißen würde. OF ist auch in diesem Fall gesetzt, da ein Borgen aus Bit 31 in Bit 30 notwen-

dig wurde. Das aber bedeutet: Ist OF gesetzt und SF gelöscht, so wurde durch die Operation der negative Wertebereich unterschritten und SF zeigt das falsche, »entgegengesetzte« Vorzeichen. Ist OF dagegen gelöscht, so gibt SF wiederum das Vorzeichen korrekt an.

Anhand der Definition der Flags können Sie schon erkennen, dass ihre Funktion untrennbar mit den verschiedenen einsetzbaren Daten verknüpft ist: Das carry flag unterstützt die Interpretation vorzeichenloser Zahlen, sign und overflow flag die der vorzeichenbehafteten und adjust flag die der BCDs. Das zero flag kann für alle Zahlenarten verwendet werden, während das parity flag in diesem Zusammenhang keine Funktion hat.

Da diese Entscheidungshilfen von so großer Bedeutung sind, wurden Mnemonics (zur Definition des Begriffs siehe Kapitel »Mnemonics, Befehlssequenzen, Opcodes und Microcode« auf Seite 768) geschaffen, die Teil von bestimmten Befehls-Mnemonics sind und jeweils für eine ganz bestimmte Kombination von Statusflags gelten. Tabelle 1.1 zeigt die 30 Mnemonics, die aufgrund bestimmter Redundanzen und Beziehungen untereinander durch nur 16 unterschiedliche Prüfungen realisiert werden.

Mnemonics						Prüfung
Bedingung	**Negierung**		**Identität zu**			
vorzeichenlos:						
A	above	NBE	not below or equal			CF = 0 und ZF = 0
AE	above or equal	NB	not below	NC	no carry	CF = 0
B	below	NAE	not above or equal	C	carry	CF = 1
BE	below or equal	NA	not above			CF = 1 oder ZF = 1
vorzeichenneutral:						
E	equal			Z	zero	ZF = 1
NE	not equal			NZ	not zero	ZF = 0
vorzeichenbehaftet:		NLE	not less or equal			OF = SF und ZF = 0
G	greater	NL	not less			OF = SF
GE	greater or equal	NGE	not greater or equal			OF ≠ SF
L	less	NG	not greater			OF ≠ SF oder ZF = 1
LE	less or equal					
allgemein:				PO	parity odd	
NO	no overflow					OF = 0
NP	no parity					PF = 0
NS	no sign					SF = 0
O	overflow					OF = 1
P	parity			PE	parity even	PF = 1
S	sign					SF = 1

Tabelle 1.1: Mnemonics für die Kombination bestimmter Statusflags nach vergleichenden Befehlen

So ist, wie eben gesehen, eine vorzeichenbehaftete Zahl dann größer als eine andere, wenn nach Bildung der Differenz das zero flag gelöscht und sign und overflow flag gleich sind. Diese Bedingung und die dahinter stehende Prüfung besitzt das Mnemonic »G« (»*greater*«), das Teil von so genannten bedingten Befehlen ist (z.B. JG, jump *if greater*). Diese Befehle werden wir weiter unten kennen lernen.

Die Statusflags sind, bis auf eine Ausnahme, nicht direkt veränderbar. Das heißt: Es gibt keine Befehle, die sie direkt einzeln und gezielt verändern! Wie gesagt: Bis auf eine Ausnahme, und die betrifft das carry flag. Seiner Bedeutung nicht nur als Statusflag nach arithmetischen Operationen entsprechend können Sie es mit bestimmten Befehlen setzen, löschen oder »umdrehen«. Die hierzu notwendigen Befehle werden wir im Kapitel »Instruktionen zur gezielten Veränderung des Flagregisters« auf Seite 127 besprechen.

Es gibt aber sehr wohl eine Möglichkeit, die Statusflags indirekt gezielt zu verändern. Dazu muss aber das gesamte EFlags-Register ausgelesen und in ein Allzweckregister transportiert werden. Hier lässt/lassen sich dann das/die zu verändernden Flag(s) mit logischen oder Bit-orientierten Instruktionen verändern und wieder in das EFlags-Register zurücktransferieren. Diese Methode werden wir in Teil 2 des Buches kennen lernen.

CPU-Befehle

Soweit im Folgenden nicht ausdrücklich anders vermerkt, lassen sich alle CPU-Befehle mit DoubleWords (32 Bits), Words (16 Bits) oder Bytes (8 Bits) bzw. ihren vorzeichenbehafteten Pendants (LongInt, SmallInt, ShortInt) durchführen. Die Unterscheidung erfolgt ausschließlich durch die Angabe des entsprechenden »Register«-Namen (z.B. ADD AL, 3 - Byte; SUB BH, BL - Byte; INC CX, 1 - Word; DEC EDX, EAX - DoubleWord), soweit (mindestens) ein Register involviert ist. Ist dagegen kein Register beteiligt, kommt eine Speicherstelle zum Einsatz. Diese muss daher vorab in geeigneter Weise definiert worden sein, damit der Assembler weiß, mit welchen Datenbreiten er arbeiten muss. Wir werden darauf in Teil 2 des Buches zurückkommen.

Da in den folgenden Betrachtungen mit verschiedenen Beispielen gearbeitet wird, empfiehlt es sich, zunächst zum besseren Verständnis das Kapitel »Datenformate« ab Seite 778 zu konsultieren, in dem wichtige Aspekte der prozessorinternen Darstellung verschiedener Daten be-

schrieben werden, von deren Kenntnis im Folgenden Gebrauch gemacht wird.

Der CPU-Befehlssatz umfasst Befehle zu

- arithmetischem Manipulieren von Daten
- logischem Manipulieren von Daten
- Datenvergleich
- bitorientierten Operationen
- Datenaustausch
- Datenkonversion
- Sprungbefehlen
- Flagmanipulationen
- Stringoperationen
- Verwaltungsoperationen
- speziellen Operationen

Die meisten der folgenden CPU-Befehle haben Operanden, also Parameter, die ihnen übergeben werden. Diese Operanden müssen in einer speziellen Art und Weise angegeben werden, um korrekt zu arbeiten. Sollten Sie mit der Angabe dieser Operanden (Befehlssemantik) nicht vertraut sein, konsultieren Sie bitte das Kapitel »Befehlssemantik« auf Seite 763, bevor Sie weiterlesen.

1.1.1 Arithmetische Operationen

Die CPU ist Integer-orientiert. Das bedeutet, dass sie nur mit Ganzzahlen arbeiten kann. Es verwundert daher nicht sonderlich, dass sich die Arithmetik der CPU auf die Grundrechenarten und wenig mehr beschränkt, dafür aber mit einigen Variationen, die unterschiedliche Bedingungen berücksichtigen.

ADD
SUB

Beginnen wir mit den grundlegendsten Berechnungen. Natürlich kann die CPU Integers addieren (ADD) und subtrahieren (SUB). Das Ergebnis der Operation kann – abgesehen vom Wert, zu dessen Berechnung wohl nicht viel zu sagen ist – mit Hilfe der Statusflags gemäß der eingesetzten Daten interpretiert werden.

Statusflags

So signalisiert das zero flag, wenn gesetzt, dass das Ergebnis »Null« ist, unabhängig davon, ob die betrachteten Zahlen vorzeichenlos oder vorzeichenbehaftet sind. (Es gibt also hier keine »negative Null« wie bei Fließkommazahlen, wie wir noch sehen werden!) Bei vorzeichenlosen Zahlen signalisiert das carry flag darüber hinaus, ob ein Über- oder Unterlauf stattgefunden hat, der gültige Wertebereich somit über- oder unterschritten wurde. Ob es ein Über- oder Unterlauf war, entscheidet die Operation: Eine Unterschreitung des Wertebereichs mit ADD ist bei vorzeichenlosen (und somit immer positiven) Zahlen genauso wenig möglich wie eine Überschreitung mittels SUB.

Dies kann nur bei vorzeichenbehafteten Zahlen erfolgen. Hier übernimmt daher das overflow flag die Funktion des carry flag. Ist OF gelöscht, hat durch die Operation kein Überlauf in das oder Borgen aus dem Vorzeichenbit (sign flag oder MSB, most significant bit; Bit 31 bei LongInts, Bit 15 bei SmallInts, Bit 7 bei ShortInts) stattgefunden. Im gesetzten Zustand wurde das sign flag aufgrund des Über- bzw. Unterlaufs verändert. Wie das overflow flag in Verbindung mit dem sign flag zu interpretieren ist, wurde bereits weiter oben geschildert (Seite 41 ff.).

Handelte es sich dagegen weder um vorzeichenlose noch um vorzeichenbehaftete Zahlen, sondern um BCDs, hat das adjust flag seinen Auftritt. Es zeigt analog zum carry flag an, dass ein Überlauf bei der Addition zweier ungepackter BCDs stattgefunden hat, hier allerdings von Bit 3 in Bit 4, da BCDs ja 4-Bit-Integers sind. Bitte beachten Sie, dass nach Addition zweier ungepackter BCDs der Korrekturbefehl AAA und nach Subtraktion zweier BCDs der Korrekturbefehl AAS aufgerufen werden muss, um ein korrektes Ergebnis zu erhalten.

Auch das carry flag kann bei BCDs eine Rolle spielen. Neben den ungepackten BCDs, die mit AAA und AAS korrigiert werden können, können auch gepackte BCDs addiert und subtrahiert werden. In diesem Fall fungiert CF als AF des zweiten Nibbles, also der zweiten BCD. Nach Addition/Subtraktion von gepackten BCDs muss die Korrektur DAA bzw. DAS aufgerufen werden. Einzelheiten zu diesen Korrekturbefehlen finden Sie weiter unten.

Bleibt noch das parity flag. Es signalisiert wiederum die Parität des niedrigstwertigen Byte des Ergebnisses, also seiner Bits 7 bis 0: Liegt eine gerade Anzahl von gesetzten Bits vor (»gerade Parität«), so ist PF gesetzt, andernfalls gelöscht.

ADD und SUB erlauben verschiedene Arten von Operanden (XXX dient im Folgenden als Platzhalter für ADD bzw. SUB): *Operanden*

- Addition/Subtraktion einer Konstanten zum/vom Akkumulatorinhalt
  `XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Registerinhalt
  `XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Speicheroperand
  `XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32`
- Addition/Subtraktion einer Byte-Konstanten zu/von einem Registerinhalt mit Vorzeichenerweiterung
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Addition/Subtraktion einer Byte-Konstanten zu/von einem Speicheroperand mit Vorzeichenerweiterung
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Addition/Subtraktion eines Registerinhaltes zu/von einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Addition/Subtraktion eines Speicheroperanden zu/von einem Registerinhalt
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Addition/Subtraktion eines Registerinhalts zu/von einem Speicheroperanden
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

Sie sehen, die grundlegenden arithmetischen Befehle sind so grundlegend, dass mit ihnen praktisch jede Datenquelle (Konstante, Register, Speicher) und praktisch jedes Ziel (Register, Speicher) verwendet werden kann.

Beachten Sie bitte, dass der Akkumulator (also das EAX-Register bzw. seine AX- bzw. AL-Form) auch bei den modernen Prozessoren mit gleichberechtigten Allzweckregistern immer noch in der Form eine Sonderrolle spielt, dass es sich bei der Addition/Subtraktion von Konstanten zu/vom Akkumulator um Ein-Byte-Befehle handelt, während alle anderen Versionen mindestens zwei Bytes umfassen.

MUL
IMUL

Ganz so einfach wie bei Addition und Subtraktion ist die Sache bei der Multiplikation zweier Zahlen nicht. Das fängt mit der Feststellung an, ob ein Vorzeichen existiert oder nicht. Wie jeder mit Papier und Bleistift nachvollziehen kann, ist die Frage, ob das höchstwertige Bit des Datums in die Berechnung einbezogen werden muss (kein Vorzeichenbit) oder nicht (Vorzeichenbit), also ob die Zahlen durch 31 oder 30 Bits (DoubleWords/LongInts) dargestellt werden, ganz entscheidend für das Ergebnis. (Analoges gilt natürlich für Words/SmallInts und Bytes/ShortInts.) Man kann also in diesem Fall nicht einfach anhand von Flagstellungen und *nachträglicher* Interpretation des Wertes zu einem korrekten Ergebnis kommen: Die durchgeführten Operationen sind unterschiedlich! Daher existieren für die Multiplikation jeweils zwei Befehle, die entweder vorzeichenlose Ganzzahlen verarbeiten (MUL) oder vorzeichenbehaftete Integers im engeren Sinne (*integer multiplication* IMUL).

Statusflags

Da für jeden Fall ein eigenständiger Befehl existiert, spielen die Flagstellungen bei MUL/IMUL eine untergeordnete Rolle, wenn überhaupt. Nach MUL und IMUL haben nur CF und OF eine Bedeutung. Sie zeigen an, ob das Ergebnis der Multiplikation den Wertebereich der Operanden überschritten hat oder nicht.

Was heißt das? Bei MUL ist das einfach zu erklären. Wenn beispielsweise zwei Words mit einander multipliziert werden, so kann das Ergebnis Werte im Bereich eines DoubleWords annehmen (z.B. \$1000 · \$00FF = \$000F_F000). Muss aber nicht: Es kann auch im Wertebereich eines Words bleiben (z.B. \$0100 · \$00FF = \$0000_FF00). Und genau dieser Sachverhalt wird durch CF und OF signalisiert: Wird durch die Multiplikation der Wertebereich der Operanden (hier Words) überschritten, so werden CF und OF gesetzt. In diesem Fall ist das höherwertige Word des Ergebnis-DoubleWords nicht »0«. Bleibt dagegen das Ergebnis im Wertebereich Word, so ist das höherwertige Word des Ergebnisses »0« und CF und OF werden gelöscht.

Bei IMUL ist das zwar grundsätzlich gleich. Doch nachdem IMUL mit vorzeichenbehafteten Integers arbeitet, kann das Ergebnis auch negativ sein. In diesem Falle ist der höherwertige Anteil der resultierenden LongInt von Null verschieden, selbst wenn das Ergebnis vom absoluten Betrag her in ein Word passen würde. (Stichwort: »sign extension«. Im höherwertigen Teil steht dann der Wert \$FFFF, der aus der Vorzeichenerweiterung einer SmallInt in eine LongInt resultiert.) Daher wird bei IMUL dann CF und OF gelöscht, wenn der höherwertige Anteil des Ergebnisses entweder »0« ist (positive Zahl) oder »\$FFFF« (negative

Zahl). Andernfalls sind beide Flags gesetzt. (Es versteht sich, glaube ich, von selbst, dass der hier an der Multiplikation von SmallInts dargestellte Sachverhalt analog mit den anderen Daten – ShortInts und LongInts – funktioniert!)

Operanden

Als Operatoren für die Befehle kommen lange nicht so viele Möglichkeiten wie bei der Addition/Subtraktion in Betracht. Hinzu kommt, dass die Befehle den ersten Operanden, der Ziel- und ersten Quelloperanden angibt, schlichtweg implizieren. Insofern gibt es nur zwei Möglichkeiten (XXX steht für MUL/IMUL):

- Expliziter (zweiter) Quelloperand ist ein Register
  `XXX Reg8; XXX Reg16; XXX Reg32`
- Expliziter (zweiter) Quelloperand ist eine Speicherstelle
  `XXX Mem8; XXX Mem16; XXX Mem32`

In allen Fällen ist der erste Quelloperand (= Multiplikand) und damit auch das Ziel (= Produkt) vorgegeben: der Akkumulator. (Wieder eine »Verletzung« des Gleichheitsprinzips für Allzweckregister!) Je nach Größe des explizit angegebenen Operanden (Quelloperand 2 = Multiplikator!) ist damit die implizierte Quelle (Quelloperand 1) und das ebenfalls implizierte Ziel vorgegeben, wie Tabelle 1.2 zeigt:

expliziter Operand (Source-Operand #2)	durch expliziten Operanden festgelegte Datengröße	impliziter Operand (Source-Operand #1)	impliziter Zieloperand
Reg8 / Mem8	Byte	AL	AX
Reg16 / Mem16	Word	AX	DX:AX
Reg32 / Mem32	DoubleWord	EAX	EDX:EAX

Tabelle 1.2: Explizite und implizite Operanden des MUL-/IMUL-Befehls

Beachten Sie bitte, dass bei der Verwendung von Words als Operanden das resultierende DoubleWord auch bei 32-Bit-Prozessoren *nicht* in EAX abgelegt wird, sondern in das höherwertige Word in DX und das niedrigerwertige Word in AX aufgeteilt wird: *DX := HiWord(AX * Mem16/Reg16), AX := LoWord(AX * Mem16/Reg16).* Dies ist in der Abwärtskompatibilität zu den 16-Bit-Prozessoren begründet. Leider gibt es keine MUL-Version, die ein DoubleWord-Ergebnis in EAX ablegt.

Bei IMUL dagegen sieht das (scheinbar) ein wenig erfreulicher aus. Der IMUL-Befehl existiert in drei Formen: der Ein-Operanden-Form, der Zwei-Operanden-Form und sogar in einer Drei-Operanden-Form. Durch die Erweiterungen können auch erster Quell- und Zieloperand

explizit vorgegeben werden. Doch erkauft man sich dies mit einem Nachteil: Das Multiplikationsergebnis kann eventuell nicht korrekt sein, wie wir gleich sehen werden.

In der Ein-Operanden-Form verhält sich der IMUL-Befehl analog zu MUL, mit der Ausnahme, dass er vorzeichenbehaftete Integers verwendet. Es gilt also auch hier die Tabelle und die Aufteilung eines Ergebnis-DoubleWords in zwei Word-Register selbst bei 32-Bit-Prozessoren.

In der Zwei-Operanden-Form sind folgende Operanden erlaubt:

- Multiplikation eines Registerinhaltes mit einer vorzeichenerweiterten Konstante
  `IMUL Reg16, Const8; IMUL Reg32, Const8`
- Multiplikation eines Registerinhaltes mit einer Konstanten
  `IMUL Reg16, Const16; IMUL Reg32, Const32`
- Multiplikation eines Registerinhaltes mit einem Registerinhalt
  `IMUL Reg16, Reg16; IMUL Reg32, Reg32`
- Multiplikation eines Registerinhaltes mit einem Speicheroperanden
  `IMUL Reg16, Mem16; IMUL Reg32, Mem32`

Bitte beachten Sie, dass bei den Zwei-Operanden-Sequenzen das Ziel (und damit auch die erste Quelle) immer ein Register sein muss. Dessen Inhalt kann entweder (unter Vorzeichenerweiterung) mit einer Byte-Konstanten oder einer Word- bzw. DoubleWord-Konstanten multipliziert werden, mit einem anderen (passenden) Registerinhalt oder dem Inhalt einer Speicherstelle.

Bei Multiplikationen der Zwei-Operanden-Form müssen Quell- und Zieloperanden die gleiche Größe besitzen. Das bedeutet, dass das Ergebnis einer Multiplikation ggf. nicht korrekt ist – dann nämlich, wenn es den Wertebereich der eingesetzten Operanden überschreitet. In diesem Falle wird im Ziel lediglich der niedrigerwertige Anteil des Ergebnisses abgelegt, der höherwertige Teil schlichtweg verworfen und CF und OF gesetzt. Passte dagegen das Multiplikationsergebnis in das Ziel, werden CF und OF gelöscht.

In der Drei-Operanden-Form bezeichnet der erste Operand das Ziel (= Produkt), das immer ein Register sein muss. Allerdings dient dieser Operand *nicht* als Quelloperand. Vielmehr wird das Produkt aus den beiden folgenden Operanden gebildet: Operand 1 := Operand 2 · Operand 3, wobei Operand 2 (= Multiplikand) ein Register oder eine Spei-

cherstelle sein kann und Operand 3 (= Multiplikator) grundsätzlich eine Konstante ist. Diese kann entweder ein Byte sein, was dann vorzeichenerweitert verwendet wird, oder eine Konstante mit der gleichen Größe wie die beiden anderen Operatoren (Word bzw. DoubleWord). Es sind folgende Operationen definiert:

- Multiplikation eines Speicheroperanden mit einer vorzeichenerweiterten Konstanten und Ablage in einem Register
  `IMUL Reg16, Mem16, Const8; IMUL Reg32, Mem32, Const8`
- Multiplikation eines Registers mit einer vorzeichenerweiterten Konstanten und Ablage in einem anderen Register
  `IMUL Reg16, Reg16, Const8; IMUL Reg32, Reg32, Const8`
- Multiplikation eines Speicheroperanden mit einer Konstanten und Ablage in einem Register
  `IMUL Reg16, Mem16, Const16, IMUL Reg32, Mem32, Const32`
- Multiplikation eines Registers mit einer Konstanten und Ablage in einem anderen Register
  `IMUL Reg16, Reg16, Const16; IMUL Reg32, Reg32, Const32`

Auch in diesem Fall gilt, dass das Ergebnis ggf. um den höherwertigen Anteil »beschnitten« wird, da Zieloperand und alle Quelloperanden (Vorzeichenerweiterung!) die gleiche Größe haben. Daher signalisieren CF und OF, ob das Resultat in das Zielregister passte (CF = OF = 0) oder nicht.

DIV
IDIV

Ähnlich wie beim Paar MUL/IMUL verhält es sich bei der Integerdivision. Auch hier gibt es zwei Befehle, die entweder auf vorzeichenlose (DIV) oder vorzeichenbehaftete Integers (IDIV) angewendet werden.

Statusflags

Bei diesen beiden Befehlen spielen die Statusflags überhaupt keine Rolle, gelten also als undefiniert und sollten tunlichst nicht ausgewertet werden.

Bitte beachten Sie, dass sowohl DIV als auch IDIV trotz der verwirrenden Namensgebungen so genannte »Integerdivisionen« sind, also Divisionen, die keinen Nachkommateil erzeugen (sonst wäre das Resultat ja eine Realzahl!)! Das bedeutet, dass 3 DIV 2 = 3 IDIV 2 = 1, genauso wie 2 DIV 2 und 2 IDIV 2, und dass -3 IDV 2 = -1 ergibt, genauso wie -2 IDIV 2. Lediglich die ebenfalls berechneten Reste unterscheiden sich entsprechend.

Operanden

Auch bei den Divisionen kommen analog zu den Multiplikationen nur vergleichsweise wenige Möglichkeiten der Operatorenwahl in Betracht (XXX steht für DIV/IDIV):

- Expliziter (zweiter) Quelloperand ist ein Register
  `XXX Reg8, XXX Reg16, XXX Reg32`
- Expliziter (zweiter) Quelloperand ist eine Speicherstelle
  `XXX Mem8, XXX Mem16, XXX Mem32`

Auch hier ist der Dividend (= erste Quelloperand) und damit auch das Ziel (= Quotient) vorgegeben: der Akkumulator. Und auch hier sind je nach Größe des explizit angegebenen Divisors (Quelloperand 2!) damit die Quelle und das Ziel vorgegeben, wie die folgende Tabelle 1.3 zeigt:

expliziter Operand (Source-Operand #2)	durch expliziten Operanden festgelegte Datengröße	impliziter Operand (Source-Operand #1)	impliziter Zieloperand
Reg8 / Mem8	Byte	AX	AL; AH
Reg16 / Mem16	Word	DX:AX	AX; DX
Reg32 / Mem32	DoubleWord	EDX:EAX	EAX; EDX

Tabelle 1.3: Explizite und implizite Operanden des DIV-/IDIV-Befehls

Die Division eines (implizit in AX übergebenen) Dividenden durch den explizit über ein Byte-Register bzw. eine Byte-Speicherstelle übergebenen Divisor resultiert in einem ganzzahligen Divisionsergebnis, das im Byte-Akkumulator (AL) übergeben wird. AH enthält den Divisionsrest, ebenfalls in Byte-Form. Analog führt die Division des implizit in DX/EDX (höherwertiges Word/DoubleWord) und AX/EAX (niedrigerwertiges Word/DoubleWord) übergebene Dividenden durch den explizit übergebenen Word/DoubleWord-Divisor zur einem Word/DoubleWord-Divisionsergebnis in AX/EAX und einem Divisionsrest in DX/EDX.

Falls der Divisor bei DIV/IDIV »0« ist oder das Ergebnis der Division nicht in das Zielregister passt (z.B. $FFFF / $04 = $3FFF > $FF!), wird eine *divide error exception* (#DE) ausgelöst! Es werden also *nicht* wie bei den korrespondierenden Multiplikationen CF und OF verändert!

Das Vorzeichen des Divisionsrestes ist immer das gleiche wie das des Dividenden, es sei denn der Divisionsrest ist »0«, da bei der Integerdarstellung eine »-0« nicht existiert. Dies ist auch logisch, da ja die Umkehrrechnung (Quotient · Divisor + Rest = Dividend) gelten muss und

die Quotientenbildung durch »Runden in Richtung 0« erfolgt, was praktisch einem Abschneiden des imaginären Nachkommateils der Division entspricht. Konsequenterweise führt auch die Division von 3 durch -2 mittels IDIV zu -1 Rest 1 ($-1 \cdot -2 + 1 = 3$).

Der IDIV-Befehl hat anders als der IMUL-Befehl im Laufe der Evolution der Intel-Prozessoren keine wie auch immer geartete Erweiterung erfahren. Insbesondere können nicht Ziel- und erster Quelloperand explizit angegeben werden und es gibt auch keine Zwei- oder Drei-Operanden-Formen.

BCD-Korrekturen

Nachdem auch BCDs mit einander multipliziert und dividiert werden können, gibt es analog der Korrekturbefehle bei Addition und Multiplikation auch für diese Operationen Korrekturbefehle. Sie heißen AAM und AAD und werden etwas weiter unten besprochen.

ADC
SBB

Häufiger kommt es vor, dass man den Wertebereich von Integers gerne über die zur Verfügung stehenden Grenzen hinaus erweitern möchte, zumindest was Additionen und Multiplikationen betrifft. So gab es zu Zeiten der 16-Bit-Prozessoren den Wunsch, auch DoubleWords mit 32 Bits addieren oder subtrahieren zu können, seit den 32-Bit-Prozessoren sollten es 64-Bit-QuadWords sein. Hardwareseitig war das nicht sehr schwer, ließen sich doch 32-Bit-DoubleWords in zwei 16-Bit-Words aufteilen, die in zwei Allzweckregister passten. Und nachdem die Prozessoren vier solcher Allzweckregister hatten, konnte man auf diese Weise tatsächlich zwei DoubleWords bearbeiten. Gleiches gilt natürlich heute bei QuadWords und 32-Bit-Registern.

Will man nun zwei Zahlen auf diese Weise mit einander addieren, so muss zunächst der niedrigerwertige Teil beider Zahlen addiert werden (also z.B. das jeweilige niedrigerwertige DoubleWord der QuadWords). Hierbei kann ein Überlauf stattfinden. Dieser Überlauf muss bei der Addition der beiden höherwertigen Anteile berücksichtigt werden. Hierzu dient der Befehl ADC, *add with carry*. Nachdem der Überlauf nach der Addition zweier vorzeichenloser Zahlen mit dem carry flag signalisiert wird, ist dieses Flag der richtige Partner für ADC: Ist carry gesetzt, wird einfach zur Summe der beiden Operanden eine »1« addiert, andernfalls nicht.

Analoges gilt für die Subtraktion: Ergibt sich bei der Subtraktion der beiden niedrigerwertigen Anteile der QuadWords ein Unterlauf, wird

er im carry flag signalisiert. SBB, *subtract with borrow*, subtrahiert dann von der Differenz der beiden Operanden eine »1«. Andernfalls nicht.

Eine Addition und Subtraktion zweier QuadWords (z.B. in EDX:EAX und ECX:EBX) ist also ganz einfach zu erreichen:

```
ADD EAX, EBX  ; niedrigerwertige Anteile
ADC EDX, ECX  ; höherwertige Anteile mit Überlauf

SUB EAX, EBX  ; niedrigerwertige Anteile
SBB EDX, ECX  ; höherwertigen Anteile mit Unterlauf
```

Mit der jeweils ersten Instruktion werden die niedrigerwertigen DoubleWords addiert/subtrahiert, in der jeweils zweiten Zeile dann die höherwertigen, wobei ein Über-/ Unterlauf aus der ersten Operation im carry flag signalisiert und vom zweiten Befehl berücksichtigt wird.

Diese Kombinationen funktionieren sowohl bei vorzeichenlosen wie auch bei vorzeichenbehafteten Zahlen. Da nämlich im jeweils ersten Schritt die niedrigerwertigen Anteile der Daten addiert bzw. subtrahiert werden, spielen die Vorzeichen keine Rolle: Die niedrigerwertigen Anteile haben als vorzeichenlos interpretiert zu werden, weshalb das carry flag zur Erkennung eines Über-/Unterlaufs das richtige Flag ist. ADC und SBB arbeiten vollständig analog zu ADD und SUB mit der einzigen Ausnahme, dass das CF quasi ein impliziter dritter zu berücksichtigender Operand ist. Somit kann mit Hilfe der nach ADC/SBB gesetzten Flags die korrekte Interpretation erfolgen, je nachdem ob die QuadWords vorzeichenbehaftet waren oder nicht. (Zu den Flags nach ADC/SBB siehe ADD/SUB.)

Statusflags Die Statusflags werden durch ADC und SBB genauso behandelt wie durch die Zwillingsbefehle ADD und SUB.

Operanden ADC und SBB erlauben die gleichen Arten von Operatoren wie die korrespondierenden Befehle ADD und SUB (XXX dient im Folgenden als Platzhalter für ADC bzw. SBB):

- Addition/Subtraktion einer Konstanten zum/vom Akkumulatorinhalt
  `XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32`
- Addition/Subtraktion einer Konstanten zu/von einem Registerinhalt
  `XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32`

- Addition/Subtraktion einer Konstanten zu/von einem Speicheroperand
  `XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32`
- Addition/Subtraktion einer vorzeichenerweiterten Byte-Konstanten zu/von einem Registerinhalt
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Addition/Subtraktion einer vorzeichenerweiterten Byte-Konstanten zu/von einem Speicheroperand
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Addition/Subtraktion eines Registerinhaltes zu/von einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Addition/Subtraktion eines Speicheroperanden zu/von einem Registerinhalt
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Addition/Subtraktion eines Registerinhalts zu/von einem Speicheroperanden
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

INC
DEC

INC, *increment*, und DEC, *decrement*, sind im Prinzip als »einfache« Additionen und Subtraktionen aufzufassen, die eine Zahl um 1 erhöhen oder erniedrigen. Insofern sind sie von den Aktionen her nichts Besonderes. Doch vom Ergebnis her unterscheiden sie sich ein wenig von ADD bzw. SUB.

Aus den Hochsprachen werden Sie die gleichnamigen Befehle kennen. Dort gibt es allerdings die Möglichkeit, zu dem zu inkrementierenden/dekrementierenden Wert eine beliebige Konstante, nicht notwendigerweise »1«, addieren/subtrahieren zu können. Dies ist beim Assembler nicht der Fall. INC und DEC können nur mit der (implizierten) Konstanten »1« arbeiten!

Statusflags

Das carry flag (CF) wird von beiden Befehlen nicht verändert, alle anderen Statusflags (OF, SF, AF und PF) werden anhand des Ergebnisses gesetzt: ZF, wenn das Ergebnis »0« ist, PF, wenn in den Bits 7 bis 0 eine gerade Anzahl gesetzter Bits vorliegt, und OF, wenn ein Über-/Unterlauf in/aus Bit 31 (DoubleWords), 15 (Words) oder 7 (Bytes) erfolgte, da dieses Bit ja das Vorzeichen repräsentiert. Das Vorzeichenbit wird in SF kopiert.

Der Grund dafür, dass das carry flag unangetastet bleibt, ist, dass auf diese Weise Schleifenzähler realisiert werden können, ohne den Zustand des carry flag zu verändern. Auf diese Weise können z.B. weitere Abbruchbedingungen der Schleife realisiert werden, die abhängig von einer anderen Prüfung in der Schleife sind:

```
Start: :
       :
       CMP  AL, BL ; setzt carry flag, wenn AL < BL
       :
       :
       DEC  CL     ; setzt zero flag, wenn CL = 0
       JNBE Start  ; springt zurück, solange AL > BL (CF =
       :           ; 0) und CL > 0 (ZF = 0)
       :
```

Das Haupteinsatzgebiet von INC/DEC ist daher auch die Realisierung eines Schleifenzählers.

Da INC/DEC das carry flag nicht verändern, müssen Sie ADD/SUB mit einem Operanden »1« verwenden, wenn Sie nicht-vorzeichenbehaftete Zahlen um 1 inkrementieren oder dekrementieren und das carry flag zwecks Feststellung eines Über-/Unterlaufs auswerten wollen.

Operanden

INC und DEC erlauben das Inkrementieren und Dekrementieren von Registerinhalten oder Speicherstellen (XXX dient im Folgenden als Platzhalter für INC bzw. DEC):

- Inkrementieren/Dekrementieren eines Registerinhaltes
  `XXX Reg8; XXX Reg16; XXX Reg32`
- Inkrementieren/Dekrementieren einer Speicherstelle
  `XXX Mem8; XXX Mem16; XXX Mem32`

Beachten Sie bitte, dass es für die Codierung der Registervarianten zwei Opcodes gibt, wenn ein 16-Bit- oder ein 32-Bit-Register inkrementiert/dekrementiert wird: Ein-Byte-Opcodes und Zwei-Byte-Opcodes. In der Regel wird aber der Assembler den optimalen Code erzeugen. ACHTUNG: Die Existenz von Zwei-Byte-Codes ist leider nicht analog den Befehlen AAD und AAM zu sehen, indem das zweite Byte die Inkrementationskonstante codiert (s.u.). Das zweite Byte ist tatsächlich als Teil des Opcodes für die Codierung »Inkrementieren/Dekrementieren um 1« notwendig.

NEG

NEG, *negate*, ist ein sehr einfacher arithmetischer Befehl: Er bildet den »arithmetisch negierten« Wert – das »2er-Komplement« –, also quasi das Ergebnis der Operation (0 – Wert) und hat daher nur dann einen Sinn, wenn die Daten als vorzeichenbehaftet interpretiert werden.

Statusflags

Daher hat das carry flag hier auch eine leicht andere Bedeutung: Es ist nur dann gelöscht, wenn der Operand den Wert »0« hat; somit hat es den entgegengesetzten Wert zum zero flag. Dies ist auch sinnvoll, da ja außer bei 0, wo kein »Borgen« notwendig wird, bei jedem anderen Wert ein Unterlauf erfolgen *muss*, den das carry flag logischerweise signalisiert. Alle anderen Statusflags werden anhand des Ergebnisses wie gewohnt gesetzt: zero flag, wenn der Inhalt des Operanden nach der Negierung 0 ist (was nur dann der Fall sein kann, wenn der Operand vorher ebenfalls den Wert 0 hatte), sign flag, wenn das MSB gesetzt ist. Das overflow flag wird immer gelöscht, da es niemals eine Über- oder Unterschreitung des Wertebereichs geben kann. Das parity flag wird gesetzt, wenn die Anzahl gesetzter Bits in niedrigstwertigen Byte des Operanden gerade ist, und das adjust flag ist ebenso wie das carry flag immer dann gesetzt, sobald der zu negierende Operand einen Wert größer als 0 besitzt, da es dann grundsätzlich einen Unterlauf aus Bit 4 in Bit 3 des Operanden geben *muss*.

Operanden

Als Operanden kommen bei NEG nur Registerinhalte oder Speicherstellen in Frage:

- Negieren eines Registerinhaltes
  `NEG Reg8; NEG Reg16; NEG Reg32`
- Negieren einer Speicherstelle
  `NEG Mem8; NEG Mem16; NEG Mem32`

Weitere Operanden machten auch keinen Sinn!

AAA
AAS
AAM
AAD

AAA, *ASCII adjust after addition*, AAS, *ASCII adjust after subtraction*, AAM, *ASCII adjust after multiplication*, und AAD, *ASCII adjust before division*, sind eigentlich keine »echten« arithmetischen Befehle, sondern eher »Korrekturbefehle«. Sie dienen dazu, die Fehler zu korrigieren, die entstehen, wenn man binär ausgerichtete Operationen auf »dezimale« Daten anwendet (die ja eigentlich nicht wirklich dezimal, sondern nur dezimal codiert und ansonsten binär sind). Da sie aber im Kontext zu arithmetischen Befehlen zu sehen sind, werden sie auch an dieser Stelle behandelt. AAA, AAS und AAM sind Operationen, die eine *vorangegangene* arithmetische Operation korrigieren, während AAD eine Division *vorbereitet*, damit das Ergebnis eine korrekte BCD ist. (Falls Sie In-

formationen zu BCDs benötigen, konsultieren Sie bitte das Kapitel »Datenformate« auf Seite 778.)

Die Namen der Korrekturbefehle sind nicht gerade selbsterklärend! Sie resultieren aus einer der Hauptanwendungen nicht gepackter BDCs. So sind solche BCDs recht einfach in die ASCII-Codes der korrespondierenden Zeichen überführbar, indem man zu der in den Bits 3 bis 0 eines Bytes codierten BCD-Ziffer den Wert $30 (und damit einen Inhalt der Bits 4 bis 7 des Bytes) addiert: Das ASCII-Zeichen »0« hat den Code $30, das ASCII-Zeichen »9« den Code $39. AAA, AAS, AAM und AAD wurden (und werden heute noch manchmal) daher eingesetzt, um ASCII-Ziffern zu erzeugen.

Operanden Alle Befehle haben keinen expliziten Operanden. Vielmehr implizieren sie den Akkumulator als Quell- und Zieloperanden. Sie gehen von folgenden Voraussetzungen aus:

- Es wurden/werden *ungepackte* BCDs (eine Ziffer pro Byte) manipuliert.
- Das Ergebnis bzw. der Dividend der mathematischen Operation steht in AL (eine Ziffer) bzw. AH und AL (zwei Ziffern), wobei in AH die höherwertige und in AL die niedrigerwertige Ziffer steht.
- Das adjust flag AF wurde an die Situation angepasst (nicht bei AAD und AAM).

Die einzelnen Korrekturbefehle führen nun folgende Operationen aus:

```
AAA:
if (AL > 9) or AF = 1
   then (AL := AL + 6) mod 16; AH := AH + 1; AF := 1; CF := 1;
   else AF := 0; CF := 0;

AAS:
if (AL > 9) or AF = 1
   then (AL := AL – 6) mod 16; AH := AH – 1; AF := 1; CF := 1;
   else AF := 0; CF := 0;

AAM:
AH := AL div 10; AL := AL mod 10;

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

Nach einer Addition zweier ungepackter BCD-Ziffern können drei Fälle auftreten:

- Das Ergebnis ist eine BCD-Ziffer im Bereich 0 bis 9. Dann ist alles OK, weshalb AAA lediglich AF und CF löscht, um diesen Sachverhalt zu signalisieren.
- Das Ergebnis liegt zwischen 10 und 15. Dann ist AF zwar nicht gesetzt, da kein Überlauf in Bit 4 stattgefunden hat, aber AL ist größer als 9. In diesem Fall wird 6 (binär) addiert und das Ergebnis modulo 16 genommen. Es liegt somit zwischen 0 ($0A + 6 = $10 = 16; 16 mod 16 = 0) und 5 ($0F + 6 = $15 = 21; 21 mod 16 = 5). Die zweite Ziffer (immer eine 1!) wird zum Inhalt in AH addiert und AF und CF gesetzt, um den Überlauf in AH zu signalisieren.
- AF ist gesetzt, da die Addition einen Überlauf in Bit 4 erzeugte. Dies ist der Fall, wenn das Ergebnis zwischen 16 und 18 liegt (18 ist der maximal mögliche Wert, wenn zwei BCD-Ziffern mit maximalem Wert 9 addiert wurden!). In diesem Fall erfolgt der gleiche Vorgang wie zuvor.

Analoges gilt für die Korrektur nach Subtraktion, nur mit umgekehrtem Vorzeichen (im wahrsten Sinne des Wortes!):

- Liegt das Ergebnis zwischen 9 und 0, werden lediglich die Flags AF und CF gelöscht, da eine gültige BCD-Ziffer vorliegt.
- Wenn AF gesetzt ist, da binäre Werte zwischen $FF (= 0 – 1) und $F7 (= 0 – 9), die nach der Subtraktion entstehen können und korrigiert werden müssten, nur durch Borgen aus Bit 4 entstehen können und damit einen Unterlauf hervorrufen, der durch ein gesetztes AF angezeigt wird, oder/und das Ergebnis > 9 ist, wird der Wert 6 vom Ergebnis abgezogen und dieses modulo 16 genommen, was zu Werten zwischen 9 ($FF – 6 = $F9; $F9 mod 16 = 9) und 1 ($F7 – 6 = $F1; $F1 mod 16 = 1) führt. Dann wird 1 von AH abgezogen (Borgen!) und AF und CF zum Signalisieren des Unterlaufs gesetzt.

Die Korrektur nach Multiplikationen ist relativ klar: Da die eigentliche Multiplikation mittels MUL erfolgte (IMUL darf nicht verwendet werden, da CPU-BCDs per definitionem kein Vorzeichen besitzen), ist das Ergebnis eine binär codierte Zahl im Bereich 0 bis 81 (= $00 bis $51), die in zwei Dezimalteile aufgeteilt werden muss. Dies erfolgt durch Division mit 10, wobei das Divisionsergebnis (höherwertige Ziffer) in AH abgelegt wird, der Divisionsrest (niedrigerwertige Ziffer) in AL.

Auch die Vorbereitung einer Division ist klar, sie erfolgt umgekehrt zur Multiplikation: Die in AH und AL abgelegten Ziffern einer zweistelligen, ungepackten BCD werden durch Multiplikation der höherwertigen Ziffer (AH) mit 10 und Addition der niedrigerwertigen Ziffer (AL) in eine Binärzahl umgeformt, die dann als Dividend der Division fungieren kann.

Statusflags Das overflow flag, das sign flag sowie die Flags zero und parity sind nach AAA und AAS undefiniert. Das carry flag sowie das adjust flag wurden gesetzt, wenn durch die Korrektur ein dezimaler Über-/Unterlauf in das/vom AH-Register erfolgte, andernfalls sind sie gelöscht. Nach AAM und AAD sind das carry flag, das adjust flag sowie das overflow flag undefiniert, zero flag, sign flag und parity flag werden aufgrund des binären Resultates der Operation entsprechend gesetzt.

Alle hier vorgestellten Korrekturbefehle gehen davon aus, dass die Randbedingungen für das korrekte Arbeiten von AAA, AAS, AAM und AAD gegeben und vom Programmierer sichergestellt sind. Aus diesem Grunde erfolgt auch keine weitergehende Korrektur oder Kontrolle der eingesetzten Operanden oder resultierenden Ergebnisse. So erwartet AAA, dass zwei einziffrige BCDs addiert wurden. Zwar ist aufgrund der Tatsache, dass die Korrektur durch Inkrementieren des AH-Registers erfolgt, auch die Addition einer Ziffer zu einer zweiziffrigen (ungepackten!) BCD möglich, doch darf dann das Resultat den Wert $0909 (ungepackte BCD-Ziffern befinden sich in AH und AL!) nicht überschreiten: So führt z.B. die Addition von 1 zu 99 mit BCD-Ziffern zunächst zum vorläufigen binären Ergebnis $090A ($99_{BCD} + 1_{BCD}$= $0909 + $01 = $090A), das durch AAA in $0A00 und damit ein falsches Ergebnis »korrigiert« wird: Die niedrigerwertige Ziffer stimmt, aber der Überlauf wird lediglich ungeprüft und unkorrigiert zu AH addiert. Analog geht AAS davon aus, dass in AH eine von »0« verschiedene Ziffer existiert, wenn eine größere von einer kleineren Ziffer abgezogen wird. Denn AAS verarbeitet den entstehenden Überlauf durch unkorrigiertes und ungeprüftes Dekrementieren von AH. Steht dort 0, resultiert der falsche Wert $FF! Schließlich geht auch AAM davon aus, dass maximal zwei einziffrige BCD-Zahlen miteinander multipliziert werden. Ist dies nicht gewährleistet, kann das zu vollkommen unerwarteten oder gar falschen Ergebnissen führen.

Auch die vorbereitende Erzeugung eines divisionsfähigen Dividenden durch AAD erfordert die Einhaltung der Randbedingungen. Dazu ein Beispiel: Die Division der nicht gepackten BCD 98 (bei der die Ziffer 9

in AH und die Ziffer 8 in AL steht!) durch die BCD 2 führt zunächst durch das vorbereitende AAD zu der binären Zahl \$62 (98_{BCD} = \$0908 = 9 · 10 + 8 = 98 = \$62) in AL. Diese Zahl durch 2 dividiert ergibt \$31 = 31_{BCD} = \$0301, was weit weg ist vom korrekten Ergebnis 49_{BCD} = \$0409. Grund: Das Ergebnis erfordert mehr als eine Ziffer, ist also ohne Korrektur nicht darstellbar. Dies ist aber nicht im Sinne der BCD-Arithmetik-Philosophie! Nur dann, wenn die BCD-Division als Umkehrung der maximal möglichen Multiplikation mit zwei einzelnen BCD-Ziffern angesehen wird, wobei der maximal erlaubte Wert des Dividenden durch den Divisor vorgegeben wird, kann ein korrektes Ergebnis herauskommen: 9 · 9 = 81; daher transformiert AAD 81_{BCD} in \$51 und die Division von \$51 durch \$09 ergibt \$09 oder 9_{BCD}. Somit sind bei einem Divisor von 9 alle Dividenden oberhalb 81 = 9 · 9 verboten. (Anderes Beispiel: 9 · 5 = 45, daher 45_{BCD} → \$2D; \$2D / \$05 = \$09 = 9_{BCD}; somit sind bei einem Divisor von 5 alle Dividenden > 9 · 5 = 45 verboten.) Die korrekten Randbedingungen für die Rechnung mit BCDs sicherzustellen und einzuhalten, liegt in der Verantwortung des Programmierers.

AAM und AAD sind Zwei-Byte-Instruktionen, was bedeutet, dass ihr Opcode im Gegensatz zu den Ein-Byte-Instruktionen AAA und AAS aus zwei Bytes besteht. Sie werden das nur dann feststellen, wenn Sie das Assemblat im Debugger betrachten. Dann nämlich werden Sie sehen, dass AAM mit \$D40A und AAD mit \$D50A übersetzt wird. \$D4 steht hierbei für AAM und \$D5 für AAD. \$0A ist in beiden Fällen eine Konstante (mit dem Wert \$0A = 10), die der Assembler automatisch als expliziten (zweiten!) Operanden einfügt. Wer nun ein bisschen nachdenkt, wird schnell feststellen, dass die Konstante 10 gerade der Wert ist, der bei der Multiplikation/Division, die die beiden Befehle durchführen, verwendet wird. Damit erhebt sich die Frage, ob auch andere Werte als \$0A verwendet werden können.

Ja, sie können. Es kann anstelle von \$0A jeder beliebige Byte-Wert als Operand übergeben werden, der dann für die Multiplikation/Division herangezogen wird. Das bedeutet, dass die verallgemeinerten Versionen von AAM und AAD eine einfache Art der Umrechnung einer Ziffer der Basis 2 (binäre Zahlen) in eine Ziffer der Basis A und umgekehrt ermöglichen. Mit verschiedenen Kombinationen der erweiterten AAM-/AAD-Instruktionen sind sogar Umrechnungen von Zahlen der Basis A in Zahlen der Basis B möglich oder das Packen oder Entpacken von BCDs.

Allerdings gibt es hierzu kein Mnemonic, sodass der Assembler dies nicht unterstützt: Die Befehlssequenz muss »von Hand« erzeugt werden. Das ist aber sehr einfach, indem man die DB- oder DW-Instruktionen des Assemblers verwendet. Ein Beispiel dafür finden Sie auf der beiliegenden CD-ROM.

DAA
DAS

Was für ungepackte BCDs gilt, sollte, zumindest teilweise, auch für gepackte BCDs gelten. Daher gibt es mit DAA, *decimal adjust after addition*, und DAS, *decimal adjust after subtraction*, zwei Befehle, die AAA und AAS sehr ähnlich sind. Einziger Unterschied: Sie erwarten zwei BCD-Ziffern pro Byte.

Im Einzelnen machen die Befehle Folgendes:

```
DAA:
if (AL[3..0] > 9) or AF = 1
   then AL := AL + 6; AF := 1; CF := CF or AF;
   else AF := 0;
if (AL[7..4] > 9) or CF = 1
  then AH := AL + $60; CF := 1;
   else CF := 0;

DAS:
if (AL[3..0] > 9) or AF = 1
   then AL := AL – 6; AF := 1; CF := CF or AF;
   else AF := 0;
if (AL[7..4] > 9) or CF = 1
   then AL := AL - $60; CF := 1;
   else CF := 0;
```

Im Prinzip ist DAA ein zweimal hintereinander ausgeführtes AAA, wobei einmal die niedrigerwertige Ziffer in den Bits 3 bis 0 von AL korrigiert wird und danach die höherwertige Ziffer in den Bits 7 bis 4 von AL. Da die BCDs hier aber gepackt sind, muss keine Restbildung (*mod*) erfolgen mit Übertrag vom/ins AH-Register. Vielmehr wird der Korrekturfaktor 6 addiert, sobald die Ziffer in AL[3..0] größer als 9 oder AF gesetzt ist. Ein eventuell stattfindender Überlauf erfolgt genau da, wo er hin soll: in die Bits 7..4 der zweiten Ziffer. Dieser Überlauf wird durch Setzen des AF signalisiert. Gleichzeitig aber wird auch CF gesetzt, sodass CF nun immer dann gesetzt ist, wenn entweder die Addition vor DAA einen Überlauf erzeugte (der ja in der zweiten Ziffer der gepackten BCD korrigiert werden muss!) oder die Korrektur der niedrigerwertigen Ziffer einen Überlauf (AF!) erzeugte.

In einem zweiten Schritt wird nun die in AL[7..4] stehende, zweite BCD korrigiert. Und zwar nur dann, wenn CF gesetzt ist und damit einen additions- oder korrekturbedingten Überlauf signalisiert oder der Wert größer als 9 ist. Dann wird der Korrekturfaktor $60 addiert und CF gesetzt. Ein Überlauf in ein anderes Register, wie bei AAA, erfolgt nicht! Ein solcher Überlauf muss via CF behandelt werden.

Das gleiche Prinzip liegt auch bei DAS vor, das als zweifaches Ausführen von AAS mit den beiden gepackten BCD-Ziffern aufgefasst werden kann.

Statusflags

Das overflow flag ist nach DAA und DAS undefiniert. Ein gesetztes adjust flag signalisiert eine erfolgte Korrektur des niedrigerwertigen Nibbles (Bit 3 bis 0), carry flag eine Korrektur des höherwertigen Nibbles (Bit 7 bis 4) der gepackten BCD. Das sign flag ist gesetzt, wenn Bit 7 auch gesetzt ist, spiegelt hier aber *nicht* die Stellung eines Vorzeichens wider, da CPU-BCDs definitionsgemäß vorzeichenlos sind! Das parity flag wird anhand der Gegebenheiten ebenso gesetzt wie das zero flag.

Korrekturbefehle für Multiplikation und Division gepackter BCDs gibt es nicht, da eine Multiplikation/Division mit gepackten BCDs via MUL/DIV nicht möglich ist! Hierzu müssten zunächst die gepackten BCDs entpackt und in eine binäre Zahl konvertiert werden, bevor die Multiplikation/Division erfolgen könnte. Das Ergebnis müsste dann zunächst wieder in die Form entpackter BCDs gebracht werden, die dann gepackt werden müssten. Der Aufwand hierzu rechtfertigt aber aufgrund der beschränkten Anwendungsgebiete für gepackte BCDs nicht die Entwicklung entsprechender Befehle.

1.1.2 Logische Operationen

Wie in der Einleitung zum Kapitel der CPU-Operationen bereits gesagt, können die Daten, mit denen die CPU arbeitet, sehr unterschiedlich interpretiert werden. Alle arithmetischen Operationen betrachten die Bits der Operanden als nicht voneinander unabhängig: Sie codieren eine wie auch immer geartete Zahl. Veränderungen an einzelnen Bits (z.B. die Addition zweier gesetzter Bits 0 zweier Zahlen) hat bei solchen Instruktionen immer Auswirkungen auf die anderen Bits (weil z.B. Überträge berücksichtigt werden).

Die »logischen« Operationen dieses Abschnitts dagegen fassen die Bits der Operanden als eigenständige, logische Zustände auf, die nur die Werte »0« und »1« für die Inhalte »falsch« und »wahr« annehmen können. Überträge gibt es nicht! Dementsprechend operieren die folgenden Operationen bitweise und der Inhalt der Operanden wird als Bitfeld von unabhängigen Bits interpretiert.

AND
OR
XOR
NOT

Die CPU kennt mit AND, OR, XOR und NOT die vier grundlegenden Operationen aus dem Bereich der »Logik«. AND führt eine bitweise AND-Verknüpfung durch, bei der ein Ergebnisbit dann und nur dann gesetzt ist, wenn beide korrespondierenden Ausgangsbits ebenfalls gesetzt waren. Andernfalls wird es gelöscht. Bei der ODER-Verknüpfung ist das Ergebnisbit dann gesetzt, wenn eines oder beide Ausgangsbits ebenfalls gesetzt waren. Die »Ausschließende Oder«-Verknüpfung (*eXclusive OR*) liefert ein gesetztes Ergebnisbit, wenn eines der beiden Ausgangsbits gesetzt war, nicht aber beide. Und die logische Negierung NOT bildet das 1er-Komplement, bei dem das Ergebnisbit gesetzt wird, wenn das Ausgangsbit gelöscht war und umgekehrt. AND, OR und XOR verknüpfen damit zwei Bits miteinander, während NOT lediglich den Zustand eines Bits »umdreht«: Aus »1« wird »0« bzw. aus »0« wird »1«. Die Ergebnisse lassen sich in Tabelle 1.4 zusammenfassen:

Bit 1: \ Bit 2:	AND 0	AND 1	OR 0	OR 1	XOR 0	XOR 1	NOT
0	0	0	0	1	0	1	1
1	0	1	1	1	1	0	0

Tabelle 1.4: Darstellung der Bitstellungen nach den logischen Operationen AND, OR, XOR und NOT

Operanden

AND, OR und XOR erlauben verschiedene Arten von Operanden (XXX dient im Folgenden als Platzhalter für AND, OR und XOR):

- Logische Verknüpfung des Akkumulatorinhalts mit einer Konstanten
  ```
  XXX AL, Const8; XXX AX, Const16; XXX EAX, Const32
  ```
- Logische Verknüpfung eines Registerinhalts mit einer Konstanten
  ```
  XXX Reg8, Const8; XXX Reg16, Const16; XXX Reg32, Const32
  ```
- Logische Verknüpfung eines Speicheroperands mit einer Konstanten
  ```
  XXX Mem8, Const8; XXX Mem16, Const16; XXX Mem32, Const32
  ```

- Logische Verknüpfung eines Registerinhalts mit einer vorzeichenerweiterten Byte-Konstanten
  `XXX Reg16, Const8; XXX Reg32, Const8`
- Logische Verknüpfung eines Speicheroperands mit einer vorzeichenerweiterten Byte-Konstanten
  `XXX Mem16, Const8; XXX Mem32, Const8`
- Logische Verknüpfung eines Registerinhaltes mit einem Registerinhalt
  `XXX Reg8, Reg8; XXX Reg16, Reg16; XXX Reg32, Reg32`
- Logische Verknüpfung eines Registerinhalts mit einem Speicheroperanden
  `XXX, Reg8, Mem8; XXX Reg16, Mem16; XXX, Reg32, Mem32`
- Logische Verknüpfung eines Speicheroperanden mit einem Registerinhalt
  `XXX, Mem8, Reg8; XXX Mem16, Reg16; XXX Mem32, Reg32`

Naturgemäß sind die Möglichkeiten bei der logischen Verneinung NOT eingeschränkt, da durch diesen Befehl keine Verknüpfung erfolgt sondern eine Veränderung bestehender Bits, somit nur ein Quelloperand in Frage kommt, der gleichzeitig auch Zieloperand ist:

- Logische Verneinung eines Registerinhalts
  `NOT, Reg8; NOT Reg16; NOT, Reg32`
- Logische Verneinung eines Speicheroperanden
  `NOT, Mem8; NOT Mem16; NOT Mem32`

Beachten Sie bitte, dass die genannten Bitverknüpfungen immer zwischen den korrespondierenden Bits der beiden Operanden der Befehle AND, OR und XOR erfolgen, nie aber »innerhalb« eines Operanden! Die einzelnen Bits eines Operanden sind und bleiben voneinander unabhängig:

```
AND:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] and Source2[I]

OR:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] or Source2[I]

XOR:
for I := 0 to Length(Destination – 1) do
  Destination[I] := Source1[I] XOR Source2[I]
```

```
NOT:
for I := 0 to Length(Destination – 1) do
  Destination[I] := not Source1[I]
```

Statusflags Durch AND, OR und XOR werden das carry und overflow flag explizit gelöscht. Dies signalisiert richtigerweise, dass es nach logischen Verknüpfungen weder einen vorzeichenlosen (carry flag) noch vorzeichenbehafteten (overflow flag) Über- bzw. Unterlauf geben kann: Die Operanden der Befehle sind einzelne, von einander unabhängige Bits, keine Zahlen. Aus diesem Grunde spielt auch das adjust flag keine Rolle: Es ist nach den logischen Verknüpfungen undefiniert.

Zero flag, sign flag und parity flag dagegen werden entsprechend dem Ergebnis gesetzt: Sind alle Bits gelöscht, so ist zero flag gesetzt, andernfalls gelöscht. Das sign flag ist eine Kopie des Bits 31, 15 oder 7 – je nach Größe des eingesetzten Operanden. Somit erhebt es die genannten Bits in einen Sonderstatus, da sie durch die Verknüpfung mit dem sign flag »ein wenig gleicher« sind als die anderen Bits des Bitfeldes, die ja eigentlich alle untereinander gleich sind! Und das parity flag zeigt im gesetzten Zustand eine gerade Parität an, die sich in einer geraden Anzahl gesetzter Bits äußert.

NOT dagegen verändert keine Flags – warum auch?

Die logischen Verknüpfungen wirken, wie gesagt, auf einzelne Bits, weshalb die Inhalte der Operanden auch als Felder voneinander unabhängiger Bits und nicht als Zahl interpretiert werden (müssen). Allerdings gibt es eine Ausnahme: Wenn man eine Integer (im Beispiel ein DoubleWord) darstellt als Summe fallender Potenzen von 2:

$$I = Bit_{31} \cdot 2^{31} + Bit_{30} \cdot 2^{30} + ... + Bit_1 \cdot 2^1 + Bit_0 \cdot 2^0,$$

so lassen sich zwar nicht die Komponenten dieser Reihe als unabhängig voneinander betrachten (weil sie addiert werden) und verändern, wohl aber die Koeffizienten (Bit_x) der jeweiligen 2er-Potenzen. Auf diese Weise lassen sich auch zwei »Zahlen« logisch verknüpfen – mit teilweise frappierendem Ergebnis. So können einzelne Bits dieser Zahlen gezielt verändert werden. Zum Beispiel lassen sich die letzten acht Bits durch AND-Verknüpfung mit einer Zahl, bei der die letzten acht Bits gelöscht sind, löschen. Zu beachten ist dabei jedoch, dass die restlichen 24 Bits gesetzt sein müssen, sollen sie unverändert bleiben. Dies führt dazu, dass die Zahl, mit der verknüpft werden soll, die Konstante $FFFF_FF00 ist (Bits 31 bis 8 gesetzt, Bits 7 bis 0 gelöscht). Verknüpft man obige Integer mit dieser »Maske«, wie man solche Bit-Konstanten

nennt, mit einer AND-Verknüpfung, so sind die Bits 31 bis 8 des Ergebnisses je nach der Stellung in I gesetzt oder gelöscht und die Bits 7 bis 0 auf jeden Fall gelöscht. Betrachtet man das Resultat wiederum als Zahl, so wurde praktisch die Integer zunächst durch 256 dividiert und der resultierende Quotient der Integerdivision mit 256 multipliziert. I wurde also um den Rest einer Division durch $0000_0100 vermindert:

(I and $FFFF_FF00) ≡ I := 256 · (I div 256) = I – (I mod 256)

Erzeugt man dagegen eine Maske, in der alle Bits außer den letzten 8 Bits gelöscht sind ($0000_00FF), und führt damit eine AND-Verknüpfung durch, so sind im Ergebnis nur diejenigen der letzten acht Bits gesetzt, die auch in I gesetzt waren. Arithmetisch betrachtet handelt es sich also um eine Restbildung nach Division mit $0000_0100, auch Modulo-Bildung genannt:

(I and $0000_00FF) ≡ I := I – (256 · (I div 256)) = I mod 256

Beachten Sie bitte, dass nicht jede logische Verknüpfung und nicht jede »Maske« Sinn machen. So ergibt die Modulo-Bildung nur dann korrekte Ergebnisse, wenn als Maske »Zahlen« verwendet werden, die mit Bit 0 beginnend lückenlos bis zu dem gewünschten Divisor gesetzt sind. Damit kommen (bei Byte-Betrachtung!) nur die Zahlen 1 (Bit 0 gesetzt), 3 ($03: Bit 0 und 1 gesetzt), 7 ($07: Bit 0, 1 und 2 gesetzt), 15 ($0F), 31 ($1F), 63 ($3F), 127 (7F) und 255 ($FF) in Frage. Sie entsprechen der Modulo-Bildung mit (Maske + 1), also 2 ($01 + 1 = $02), 4 ($04), 8 ($08), 16 ($10), 32 ($20), 64 ($40), 128 ($80), 256 ($100). Der Versuch, eine Zahl mit der Maske $00F5 UND-zuverknüpfen und anzunehmen, das Ergebnis wäre der Modulus 246 ($F5 + 1 = 245 + 1) einer Zahl, scheitert! Vielmehr ist das Ergebnis eine Zahl, bei der die Bits 3 und 1 explizit gelöscht sind, was nicht zwingend nur beim Modulus 246 der Fall ist. Der Grund dafür ist, dass die Bits, aus denen die Zahlen zusammengesetzt sind, eben alles andere als unabhängig voneinander sind.

Auch die Verwendung der XOR-Verknüpfung von »echten« Zahlen miteinander oder mit Masken wird in den seltensten Fällen Sinn machen. So kann man mittels XOR eine einfache Form der Datenverschlüsselung durchführen. Die Operation C = D XOR Maske verschlüsselt das DoubleWord D mit Maske zur Chiffre C, die ihrerseits mit Hilfe von Maske zum ursprünglichen DoubleWord D entschlüsselt werden kann: D = C XOR Maske. Wie gesagt: ein einfaches Verschlüsselungsverfahren, das sicherlich nicht mit PGP und anderen professionellen Verschlüsselungsprogrammen konkurrieren kann, jedoch für manche Fälle

durchaus brauchbar ist. Eine andere Anwendung von XOR auf Zahlen ist die Bildung von Prüfsummen zum Zwecke der Verifizierung von Datenströmen, bei der dann die Daten nicht mit Masken, sondern mit folgenden Daten geXORt werden.

Und auch OR macht nur dann Sinn, wenn im Ergebnis bestimmte Bits auf jeden Fall gesetzt sein sollen. So kann die Umrechnung einer BCD in das korrespondierende ASCII-Zeichen entweder durch Addition der Konstanten $30 erfolgen oder (was gleichbedeutend ist!) durch ODER-Verknüpfung mit $30.

Allerdings gibt es noch je einen Anwendungsfall für OR und XOR, der in Assembler-Quelltexten häufig zu finden ist. So führt die OR-Verknüpfung eines Operanden mit sich selbst etwa der Form OR EAX, EAX zu einem Ergebnis, das zunächst nicht sehr sinnvoll erscheint: Es hat sich am Inhalt nichts verändert. Doch darf nicht vergessen werden, dass OR auch Flags setzt. Und das kann immer dann eine wertvolle Hilfe sein, wenn (mit Hilfe der Flags) eine Programmverzweigung aufgrund des Inhaltes des Operanden erfolgen soll, die vorangegangene Operation aber keine Flags gesetzt hat:

```
      MOV EAX, [Mem32] ; MOV verändert keine Flags
      OR  EAX, EAX     ; Flags setzen anhand des Wertes
      JZ  Zero         ; verzweigt, wenn EAX = 0.
      :
      :
Zero: :
      :
```

Zwar könnte man dies auch mittels eines arithmetischen Vergleiches mit CMP erreichen und hätte dann sogar die Möglichkeit, andere Bedingungen zu prüfen (größer, kleiner etc.). Doch ist OR kürzer, schneller, effektiver und häufig absolut ausreichend.

Einen ähnlichen Trick kann man mit XOR machen. Verknüpft man analog den Operanden mit sich selbst, wie in XOR EAX, EAX, so kommt als Resultat 0 heraus. Denn die XOR-Verknüpfung von 0 mit 0 und 1 mit 1 ergibt jeweils 0. Dies ist eine schnelle, einfache und effiziente Art, einen Operanden zu löschen. Andernfalls müsste man den ineffektiveren Weg über MOV EAX, 0 nehmen ...

1.1.3 Operationen zum Datenvergleich

Datenvergleich ist sowohl bei »arithmetischen« Daten, also Integers, wichtig wie auch bei logischen Werten, also Feldern aus »Wahrheitswerten«. Dementsprechend gibt es zwei Befehle, die genau das ermöglichen: CMP und TEST.

CMP

CMP ist der »arithmetische« Datenvergleich. Mit Hilfe dieses Befehles lassen sich zwei Zahlen mit einander vergleichen. Als Ergebnis werden die Statusflags gesetzt, die dann ein Auswerten des Ergebnisses in Form von Programmverzweigungen ermöglichen.

Der CMP-Befehl ist eigentlich ein verkappter SUB-Befehl, da er tatsächlich den zweiten Quelloperanden vom ersten abzieht. Der Unterschied zu SUB besteht nun allein darin, dass dieses Ergebnis nicht in ein Ziel transferiert wird; vielmehr werden analog SUB lediglich die Flags gesetzt und das Ergebnis danach verworfen. CMP ist somit einer der wenigen Befehle, die zwar Quelloperanden besitzen, aber keine Zieloperanden – noch nicht einmal implizit! CMP verändert die Inhalte der Operanden nicht.

Operanden

Nachdem CMP eigentlich ein SUB-Befehl ist, verfügt er auch über die analogen Möglichkeiten der Operandennutzung:

- Vergleich des Akkumulators mit einer Konstanten
  `CMP AL, Const8; CMP AX, Const16; CMP EAX, Const32`
- Vergleich eines Registerinhalts mit einer Konstanten
  `CMP Reg8, Const8; CMP Reg16, Const16; CMP Reg32, Const32`
- Vergleich eines Speicheroperands mit einer Konstanten
  `CMP Mem8, Const8; CMP Mem16, Const16; CMP Mem32, Const32`
- Vergleich eines Registerinhalts mit einer vorzeichenerweiterten Byte-Konstanten
  `CMP Reg16, Const8; CMP Reg32, Const8`
- Vergleich eines Speicheroperanden mit einer vorzeichenerweiterten Byte-Konstanten
  `CMP Mem16, Const8; CMP Mem32, Const8`
- Vergleich eines Registerinhaltes mit einem Registerinhalt
  `CMP Reg8, Reg8; CMP Reg16, Reg16; CMP Reg32, Reg32`
- Vergleich eines Registerinhalts mit einem Speicheroperanden
  `CMP, Reg8, Mem8; CMP Reg16, Mem16; CMP, Reg32, Mem32`
- Vergleich eines Speicheroperanden mit einem Registerinhalt
  `CMP, Mem8, Reg8; CMP Mem16, Reg16; CMP Mem32, Reg32`

Statusflags Wie kann man die Flags heranziehen, um festzustellen, welcher Operand nun größer war, wenn überhaupt? Dazu müssen zwei Fälle unterschieden werden, je nachdem, ob vorzeichenlose oder vorzeichenbehaftete Integers betrachtet werden:

vorzeichenlose Integer Fall 1: Die Operanden sind vorzeichenlose Zahlen. Dann richten wir unser Augenmerk auf das zero und carry flag, da ja das carry flag einen möglichen Überlauf vorzeichenloser Zahlen signalisiert. Ein solcher Überlauf müsste berücksichtigt werden. In diesem Fall gibt es drei Möglichkeiten:

- Operand1 > Operand2. Dann ist (Operand1 – Operand2) > 0, weshalb weder zero noch carry flag gesetzt sind.
- Operand1 = Operand2. Dann ist (Operand1 – Operand2) = 0, weshalb zero flag gesetzt ist, carry flag gelöscht.
- Operand1 < Operand2. Dann ist (Operand1 – Operand2) < 0, weshalb carry, nicht aber zero flag gesetzt ist. Das gesetzte carry flag signalisiert das »Borgen« aus dem nicht vorhandenen, dem MSB (most significant bit) der Zahl folgenden Bit.

Das bedeutet, dass bei vorzeichenlosen Integers immer dann Operand1 größer als Operand2 ist, wenn das carry flag gelöscht ist. Ist es gesetzt, ist Operand2 größer als Operand1. Sind beide Operanden gleich groß, so ist das zero flag gesetzt.

vorzeichenbehaftete Integer Fall 2: Die Operanden sind vorzeichenbehaftete Zahlen. Dann spielen neben dem zero flag auch noch das overflow und das sign flag eine Rolle. Die Lage ist damit ein wenig komplizierter. Hier unterscheiden wir die Fälle:

- Operand1 > Operand2. Dann ist ZF = 0 sowie SF = OF, wie die weitere, folgende Fallunterscheidung zeigt, die aufgrund der unterschiedlichen Vorzeichenkombinationen erforderlich ist:
  - Operand1 > 0; Operand2 ≥ 0. Dann ist (Operand1 – Operand2) > 0 und es hat kein Unterlauf stattgefunden. Damit ist SF = 0 und OF = 0 und somit SF = OF.
  - Operand1 > 0; Operand2 ≤ 0. Dann ist zwar (Operand1 – Operand2) > 0. Je nach absoluter Größe von Operand1 und Operand2 können aber zwei Situationen auftreten: Das Ergebnis der Addition (Subtraktion eines negativen Wertes ist identisch mit Addition des Absolutwertes!) passt in den Wertebereich. Dann ist SF = 0 und OF = 0. Überschreitet es dagegen den Werte-

bereich, so ist OF = 1 und SF = 1). In jedem Falle ist wiederum SF = OF.

  - Operand1 ≤ 0; Operand2 < 0. Dann ist (Operand1 – Operand2) > 0 (da ja Operand1 > Operand2, s.o.) und es hat kein Überlauf stattgefunden, da der maximal mögliche positive Wert durch die Subtraktion eines kleineren negativen Wertes von einem größeren negativen Wert nicht überschritten werden kann. Damit ist SF = 0 und OF = 0 und ebenfalls SF = OF.

- Operand1 = Operand2. Dann ist (Operand1 – Operand2) = 0 und somit ZF = 1 und SF und OF = 0.
- Operand1 < Operand2. Hier ist wiederum ZF = 0, jedoch ist SF ≠ OF, wie die analoge weitere Fallunterscheidung zeigt:
  - Operand1 ≥ 0; Operand2 > 0. Dann ist (Operand1 – Operand2) < 0 und es hat kein Unterlauf stattgefunden, da das Ergebnis negativ ist, niemals aber den maximalen negativen Wert überschreiten kann. Damit ist SF = 1 und OF = 0 und somit SF ≠ OF.
  - Operand1 < 0; Operand2 ≥ 0. Dann ist (Operand1 – Operand2) < 0, und es muss wiederum unterschieden werden, ob das Ergebnis in den Wertebereich passt. Tut es das, ist SF = 1 und OF = 0 und damit SF ≠ OF. Andernfalls ist OF = 1 und SF = 0 und wiederum SF ≠ OF.
  - Operand1 < 0; Operand2 < 0. Dann ist (Operand1 – Operand2) > 0, ohne dass ein Überlauf stattfinden kann, da, Absolutwerte betrachtet, der negative Operand2 immer um den Absolutbetrag des Operand1 vermindert wird. Somit ist SF = 1, OF = 0 und SF ≠ OF.

Bei vorzeichenbehafteten Integers ist also immer dann Operand1 größer als Operand2, wenn das overflow flag und das sign flag den gleichen Wert haben. Haben sie dagegen entgegengesetzte Werte, ist Operand1 kleiner als Operand2. Auch bei diesen Zahlen zeigt ein gesetztes zero flag an, dass beide Operanden gleich groß sind.

Wer die Fallunterscheidungen von eben mitverfolgt hat, wird eventuell auf eine »Ungereimtheit« gestoßen sein, die mich auch eine Weile überlegen ließ, weil ich das Problem zu sehr von der mathematischen Seite betrachtet habe. Sie betrifft die Fälle bei vorzeichenbehafteten Zahlen,

in denen die Subtraktion des Operand2 vom Operand1 zu Wertüberschreitungen geführt hat, also wenn

1. entweder (Operand1 > 0) > (Operand2 < 0)
2. oder (Operand1 < 0) < (Operand2 > 0) ist.

Die Frage ist, warum in 1. das sign flag gesetzt wird, obwohl doch das temporäre Ergebnis positiv ist (die Subtraktion einer negativen Zahl von einer positiven ist immer positiv!) bzw. in 2. gelöscht wird, obwohl doch das temporäre Ergebnis negativ ist (die Subtraktion einer positiven Zahl von einer negativen ist immer negativ!). Wie gesagt: Diese Ungereimtheit liegt an der allzu mathematischen Betrachtung. Rekapitulieren Sie bitte, was ich bei der Besprechung des EFlags-Registers geschrieben habe: Das sign flag enthält schlicht den Inhalt des MSB, das most significant bit, des Datums. Das bedeutet, dass ein Übertrag in das MSB erfolgt und erfolgen muss, da es ja auch sein könnte, dass die vorliegenden Zahlen vorzeichenlose Integers sind! Ein gesetztes overflow flag zeigt nun das »negierte« Vorzeichen an: Wenn also kein Über-/Unterlauf stattgefunden hat, ist OF immer gelöscht und SF zeigt das Vorzeichen des temporären Ergebnisses korrekt an. Ist es daher positiv (wie in 1.), so ist Operand1 > Operand2, ist es negativ (wie in 2.), so ist Operand1 < Operand2. Ist dagegen OF gesetzt, so hat ein Über-/Unterlauf stattgefunden und das sign flag signalisiert *nicht* den Zustand des Vorzeichens, sondern sein Gegenteil, weil der Übertrag in das MSB erfolgte (MSB = 1: gelöschtes sign flag wurde gesetzt, weil positiver Wertebereich überschritten wurde) oder aus dem MSB erfolgen musste (MSB = 0: gesetztes sign flag wurde gelöscht und damit der negative Wertebereich unterschritten). Somit ist in diesem Fall Operand1 > Operand2, wenn SF gesetzt ist (1.), andernfalls ist Operand1 < Operand2 (2.)

TEST TEST ist das »logische Pendant« zu CMP. Es prüft, ob zwei Bitfelder sich von einander unterscheiden. Auch bei TEST werden als Ergebnis die Statusflags gesetzt, sodass mit Programmverzweigungen auf die bestehende Situation reagiert werden kann. Allerdings ist die Flag-Auswertung lange nicht so kompliziert wie bei CMP.

Auch TEST ist eigentlich eine Verknüpfung, bei der das Resultat verworfen wird. Der Vergleich nutzt eine logische AND-Verknüpfung der beiden Operanden und setzt anhand des temporären Ergebnisses die Statusflags analog zum AND-Befehl. Dann wird das Ergebnis verwor-

fen, ohne in ein Ziel transferiert zu werden. TEST ist damit wie CMP einer der wenigen Befehle ohne Zieloperand.

Operanden

Mit einer Ausnahme sind somit bei TEST alle Operandenkombinationen erlaubt, die auch AND gestattet:

- Vergleich des Akkumulatorinhalts mit einer Maske
  `TEST AL, Const8; TEST AX, Const16; TEST EAX, Const32`
- Vergleich eines Registerinhalts mit einer Maske
  `TEST Reg8, Const8; TEST Reg16, Const16; TEST Reg32, Const32`
- Vergleich eines Speicheroperands mit einer Maske
  `TEST Mem8, Const8; TEST Mem16, Const16; TEST Mem32, Const32`
- Vergleich eines Registerinhalts mit einer vorzeichenerweiterten Byte-Maske
  `TEST Reg16, Const8; TEST Reg32, Const8`
- Vergleich eines Speicheroperands mit einer vorzeichenerweiterten Byte-Maske
  `TEST Mem16, Const8; TEST Mem32, Const8`
- Vergleich eines Registerinhaltes mit einem Registerinhalt
  `TEST Reg8, Reg8; TEST Reg16, Reg16; TEST Reg32, Reg32`
- Vergleich eines Speicheroperanden mit einem Registerinhalt bzw. Vergleich eines Registerinhalts mit einem Speicheroperanden
  `TEST Mem8, Reg8; TEST Mem16, Reg16; TEST Mem32, Reg32`

Die Ausnahme ist *TEST Reg, Mem*. Diese Register-Speicheroperand-Kombination ist unter TEST nicht möglich, was bei genauerem Hinsehen auch gerechtfertigt ist: Vom Ergebnis her ist *TEST Reg, Mem* das Gleiche wie *TEST Mem, Reg*, da die AND-Verknüpfung in beiden Fällen die gleichen Bitstellungen erzeugt (AND und somit auch TEST sind von der logischen Operation her kommutativ!). Die beiden »AND«-Fälle würden sich also lediglich darin unterscheiden, in welches Ziel das Ergebnis gespeichert wird: *Reg* bzw. *Mem*. Da aber, anders als bei AND, das Ergebnis bei TEST nach dem Setzen der Statusflags verworfen wird, sind beide Fälle gleich und daher einer redundant, weshalb auf *TEST Reg, Mem* verzichtet wird.

Statusflags

Analog AND werden bei TEST die Flags gesetzt: OF und CF sind explizit gelöscht, da es weder einen vorzeichenlosen noch einen vorzeichenbehafteten Über- oder Unterlauf geben kann. AF gilt als undefiniert, lässt also keine Auswertung zu. SF dient hier nicht als Signal für die Stellung eines Vorzeichens, sondern ist wie bei logischen Operationen

gewohnt je nach Stellung des Bits 31 (DoubleWords), 15 (Words) oder 7 (Bytes) gesetzt; und ZF ist gesetzt, wenn alle Bits gelöscht sind, das Bitfeld somit unbesetzt. PF ist nach TEST wie üblich gesetzt, wenn in Bits 7 bis 0 eine gerade Anzahl gesetzter Bits vorgefunden wird.

1.1.4 Bitorientierte Operationen

Neben den »logischen« Operationen, die auf Bitfelder wirken, gibt es noch weitere bitorientierte Prozessorbefehle. Sie dienen zum einen dazu, einzelne Bits in den Bitfeldern gezielt anzusprechen (BT, BTS, BTR, BTC) oder zu suchen (BSF, BSR), zum anderen dazu, die Reihenfolge der Bits in den Bitfeldern zu ändern (SHL, SHR, SAL, SAR, SHLD, SHRD, ROL, ROR, RCL, RCR).

Wie bereits im Abschnitt über die Logischen Operationen geäußert, können auch Zahlen als Bitfelder aufgefasst werden. So lässt sich die *Zahl* $4711_d = 1267_h$ binär darstellen als

$1 \cdot 2^{12} + 0 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$.

In dieser Summe fallender 2er-Potenzen lassen sich die Koeffizienten der 2er-Potenzen als eigenständige, von einander unabhängige Ziffern interpretieren, die einzeln und von einander unabhängig manipuliert werden können. Sie bilden somit ein Bitfeld. Dass diese Sichtweise durchaus sinnvoll sein kann, haben wir beim »Missbrauch« des AND- bzw. OR-Befehls bereits gesehen. Auch in diesem Kapitel werden wir Befehle kennen lernen, die für *Zahlen* »missbraucht« werden können, ja die sogar eigens dafür geschaffen wurden.

Zu den grundlegenden bitorientierten Befehlen gehören die »Verschiebe«-Befehle. Dies sind Befehle, bei denen die Bits um eine gewisse Anzahl von Stellen innerhalb des Bitfeldes nach links oder rechts verschoben werden. Je nachdem, wie das Schicksal der am einen Ende das Bitfeld verlassenden Bits aussieht, kennt man verschiedene Arten von Verschiebebefehlen: die »Shift«-Befehle, bei denen die »aus dem Bitfeld herausgeschobenen« Bits verworfen werden, und die »Rotationsbefehle«, bei denen die am einen Ende herausgeschobenen Bits das Bitfeld über das andere Ende wieder betreten, also im Bitfeld »rotieren«.

Die folgenden, erläuternden Abbildungen gehen von einer Situation aus, die in Abbildung 1.8 dargestellt ist.

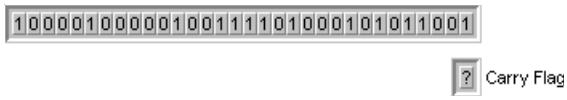

Abbildung 1.8: Speicherabbild eines Bitfeldes als Ausgangssituation vor einem Bit-Schiebebefehl

Im ersten Quelloperanden der Befehle liege ein Bitfeld vor, in dem die Bits 31, 26, 20, 17 bis 14, 12, 8, 6, 4, 3 und 0 gesetzt sind. Der Inhalt des carry flags sei unbestimmt.

SHL
SHR

Shift logical left, SHL, und *shift logical right*, SHR, sind zwei Vertreter der ersten Kategorie. Bei SHL erfolgt das Verschieben der Bits um eine anzugebende Anzahl von Positionen nach links, sodass am linken Ende die gleiche Anzahl Bits das Bitfeld verlassen und verworfen werden. Bei SHR verlassen diese Bits das Bitfeld rechts und werden ebenfalls verworfen.

Was aber passiert mit den frei gewordenen Positionen rechts (bei SHL) und links (bei SHR)? In der Grundversion der Shift-Befehle, SHL und SHR, werden die freien Plätze mit Nullen aufgefüllt. Basta! Abbildung 1.9 zeigt das Ergebnis nach einem Shift um 5 Positionen nach rechts (oben) bzw. um 3 Positionen nach links (unten). Die grau dargestellten Ziffern repräsentieren die von der »Nullhalde« aufgefüllten Ziffern.

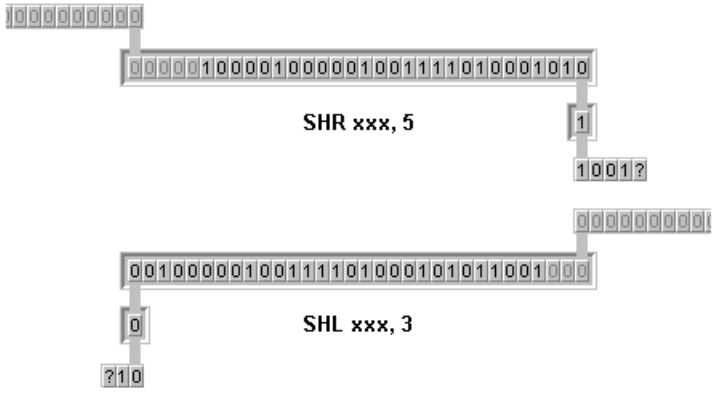

Abbildung 1.9: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Verschieben nach rechts (SHR; oben) bzw. links (SHL; unten)

SHLD
SHRD

Bei SHLD, *double precision shift left*, und SHRD, *double precision shift right*, gibt es eine Quelle, aus der die nachrückenden Bits rekrutiert werden. Diese Quelle ist ein weiteres Bitfeld, das als zweiter Operand übergeben wird. Das bedeutet, die am einen Ende des ersten Bitfeldes auftretenden Lücken werden mit Bits aus dem anderen Ende des zweiten Bitfeldes gefüllt. Abbildung 1.10 demonstriert das.

In der Abbildung herrscht jeweils die gleiche Situation wie in Abbildung 1.9, jedoch werden hier die Bits aus einem weiteren Bitfeld (»yyy«) gewonnen, in dem in diesem Beispiel vor der Operation jedes gerade Bit gesetzt war. Wie man erkennt, »saugen« die frei werdenden Positionen im Zieloperanden die Bits aus dem zweiten Quelloperanden, was dazu führt, dass auch dort die Bits verschoben werden. Dies erfolgt allerdings nur formal, da sich der Inhalt des zweiten Quelloperanden nicht ändert.

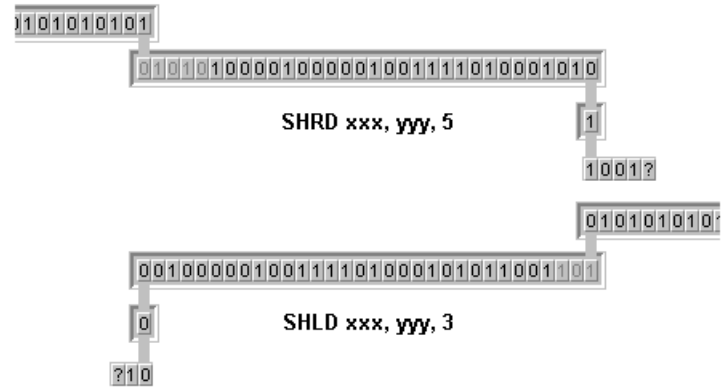

Abbildung 1.10: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »doppelt präzisem« Verschieben nach rechts (SHDR; oben) bzw. links (SHDL; unten)

SAL
SAR

Bei SAR, *shift arithmetic right*, dagegen ist Quelle das most significant bit, also Bit 7 bei Bytes, Bit 15 bei Words und Bit 31 bei DoubleWords. Das bedeutet, die links frei werdenden Stellen werden alle mit dem Bit aufgefüllt, das vor der Verschieberei einmal das MSB (Vorzeichen!) war. Nützlich und Grund für seine Existenz ist bei SAR daher, dass eine automatische sign extension durchgeführt wird, wenn vorzeichenbehaftete Zahlen nach rechts verschoben werden. Hier haben wir einen solchen »arithmetischen« Bit-orientierten Befehl. Abbildung 1.11 zeigt ein Beispiel.

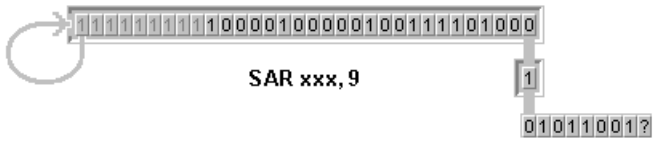

Abbildung 1.11: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »arithmetischem« Verschieben nach rechts (SAR)

Und im Falle von Verschiebungen nach links? Bleibt das Vorzeichen (MSB) der Zahlen bei SAL, *shift arithmetic left*, ebenfalls erhalten? Leider nein! SAL ist nur ein Alias für SHL und wird durch die Assembler in die gleiche Befehlssequenz übersetzt.

Alle sechs Shift-Befehle, also SHL, SHR, SAL, SAR, SHLD und SHRD, involvieren das carry flag. So wird bei allen Befehlen das letzte Bit, das aus dem Bitfeld geschoben wird, in das CF kopiert.

ROL
ROR

Lässt man dagegen die am einen Ende austretenden Bits am anderen Ende wieder eintreten, also die Bits »nur rotieren«, so hat man mit den Befehlen *rotate left*, ROL bzw. *rotate right*, ROR, die zweite Kategorie an Verschiebebefehlen. ROL und ROR involvieren das CF wie die Shift-Befehle, indem sie eine Kopie des zuletzt aus dem Bitfeld rotierten Bits in CF ablegen. (Bei ROL ist somit CF eine Kopie des LSB, da das LSB das MSB vor dem letzten Rotationszyklus war, somit das zuletzt nach links herausgeschobene und rechts aufgenommene Bit ist. Analog ist bei ROR das CF eine Kopie des MSB.)

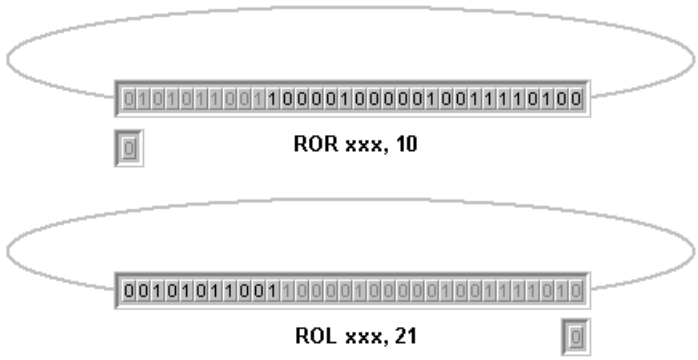

Abbildung 1.12: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach »einfachem« Rotieren nach rechts (ROR) bzw. links (ROL)

Abbildung 1.12 zeigt die beiden Rotationsbefehle. Im oberen Teil erfolgt eine Rotation nach rechts um 10 Positionen, im unteren eine nach links um 21 Positionen. Die jeweils grau dargestellten Bits sind die an einem Ende ausgetretenen und am anderen wieder eingetretenen Bits.

RCR
RCL

Diese »Rotationsbefehle« gibt es auch in einer Version, die das carry flag direkt in die Rotation mit einbezieht und nicht nur als Kopie des zuletzt rotierten Bits auffasst: RCL, *rotate left with carry,* und RCR, *rotate right with carry,* rotieren »über« das CF (siehe Abbildung 1.13). In diesem Fall muss man sich das carry flag als imaginäres Bit 32 (RCR) bzw. -1 (RCL) vorstellen (hier werden DoubleWords zugrunde gelegt, Analoges gilt natürlich für Words und Bytes!). Das bedeutet: Das erste in eine frei werdende Position hineingeschobene Bit stammt aus dem carry flag, das letzte herausgeschobene Bit landet im carry flag.

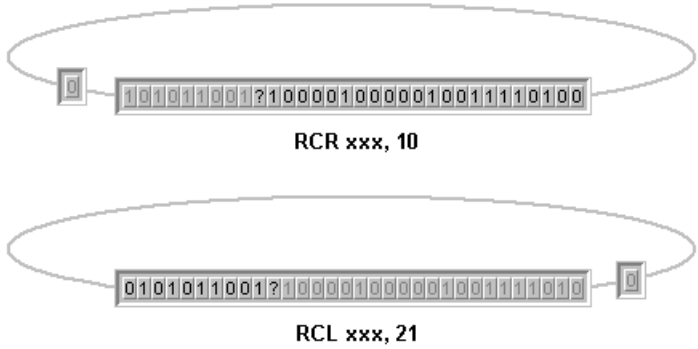

Abbildung 1.13: Speicherabbild des Bitfeldes aus Abbildung 1.8 nach Verschieben »über carry« nach rechts (RCL) bzw. links (RCL)

Das Fragezeichen zeigt die Position, an der der Inhalt des unbestimmten carry flag »eingereiht« wurde. Es ist jeweils die erste »aufgefüllte« Position. Das CF selbst beinhaltet jeweils das letzte aus dem Bitfeld geschobene Bit.

Da bei den Befehlen RCR und RCL das carry flag das erste nachrückende Bit enthält, ist es notwendig, ihm vor dem Aufruf von RCR oder RCL einen definierten Inhalt zu geben (in den Abbildungen somit eine »0« oder eine »1« zuzuordnen). Hierzu gibt es Befehle, die es explizit setzen, löschen oder gezielt verändern können. Diese Befehle werden wir im Abschnitt über »Instruktionen zur gezielten Veränderung des Flagregisters« weiter unten kennen lernen.

Die Shift-Befehle werden häufig für einfache und schnelle Multiplikationen bzw. Divisionen mit Multiplikatoren bzw. Divisoren verwendet, die eine Potenz zur Basis 2 sind. So ist die Verschiebung des Bitfeldes um eine Position nach rechts de facto eine Division durch $2^1 = 2$, während die Verschiebung um 4 Positionen nach links eine Multiplikation mit $2^4 = 16$ ist. Mit SAR lässt sich so eine vorzeichenbehaftete Division vom Typ IDIV erreichen, während SHR eine DIV-ähnliche Division durchführt. SAL und SHL entsprechen dem Befehl MUL. Ein Pendant für IMUL gibt es nicht.

Eine Division mittels SAR führt nicht zum gleichen Ergebnis wie IDIV! Treten bei der Division mittels IDIV Divisionsreste auf, so werden sie (zumindest was das Ergebnis als Integer betrifft) verworfen, es erfolgt somit eine Rundung »in Richtung Null«. Eine Division durch Bitverschiebung mittels SAR dagegen rundet »in Richtung negative Unendlichkeit«. Beispiel: -9 IDIV 4 ergibt -2 (-2.25 Richtung 0 gerundet). -9 SAR 2 ergibt -3 (-2.25 Richtung -∞ gerundet)! (Wer's nicht glaubt – cave: 2er-Komplement: $-9_d = F7_h = 11110111_b$; zwei Bits nach rechts mit sign extension: $11111101_b = FD_h = -3_d$). Diese Unterschiede treten jedoch nur bei IDIV und SAR mit negativen Zahlen auf. Für positive Zahlen oder bei DIV und SHR sind die Ergebnisse gleich, da in diesem Fall bei DIV eine Rundung »in Richtung 0« und bei SHR eine Rundung »in Richtung -∞« und somit jeweils in die gleiche Richtung erfolgt!

Operanden

Die Shift-Befehle SHL, SHR, SAL und SAR sowie die Rotationsbefehle ROL, ROR, RCL und RCR verwenden die gleichen Operandenstrukturen. Die Bitfelder werden immer als erster Operand übergeben, sind somit erster Quell- und Zieloperand. Sie können sowohl in Registern als auch an Speicherstellen stehen und 8, 16 oder 32 Bits umfassen. Als zweiten Quelloperanden erwarten die Befehle eine Zahl, die die Anzahl der zu verschiebenden Positionen angibt. Dies kann eine Konstante sein, allerdings kann sie auch über ein Register angegeben werden. Dieses Register ist immer und muss immer das 8-Bit-Register CL sein. Somit ergeben sich folgende möglichen Befehlsfolgen (XXX steht für SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR):

- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das Bitfeld in einem Register vorliegt:
  `XXX Reg8, Const8; XXX Reg16, Const8; XXX Reg32, Const8`
- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das Bitfeld in einer Speicherstelle vorliegt:
  `XXX Mem8, Const8; XXX Mem16, Const8; XXX Mem32, Const8`

- Indirekte Angabe der Anzahl zu verschiebender Positionen via CL, wobei das Bitfeld in einem Register vorliegt:
  `XXX Reg8, CL; XXX Reg16, CL; XXX Reg32, CL`
- Indirekte Angabe der Anzahl zu verschiebender Positionen via CL, wobei das Bitfeld in einer Speicherstelle vorliegt:
  `XXX Mem8, CL; XXX Mem16, CL; XXX Mem32, CL`

Die Befehle SHLD und SHRD benötigen, wie bereits angedeutet, drei Operanden. Der erste Quell- und damit auch Zieloperand ist wiederum das Bitfeld, das verändert werden soll. Hier kommen wiederum Register oder Speicherstellen als Ort für das Bitfeld in Frage, allerdings sind Byte-Operanden nicht möglich. Der zweite Quelloperand ist immer ein Register. In ihm ist das Bitfeld angegeben, das zur »Auffüllung« dienen soll – weshalb er immer die gleiche Größe wie der erste Quelloperand haben muss. Der Inhalt dieses Registers wird nicht verändert! Dritter Quelloperand ist eine Konstante wie bei den übrigen Verschiebebefehlen, die die Anzahl der zu shiftenden Positionen angibt. Somit sind folgende Befehlsfolgen möglich (XXX steht für SHLD oder SHRD):

- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld in einem Register vorliegt:
  `XXX Reg16, Reg16, Const8; XXX Reg32, Reg32, Const8`
- Direkte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld an einer Speicherstelle vorliegt:
  `XXX Mem16, Reg16, Const8; XXX Mem32, Reg32, Const8`
- Indirekte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld in einem Register vorliegt:
  `XXX Reg16, Reg16, CL; XXX Reg32, Reg32, CL`
- Indirekte Angabe der Anzahl zu verschiebender Positionen, wobei das zu verändernde Bitfeld an einer Speicherstelle vorliegt:
  `XXX Mem16, Reg16, CL; XXX Mem32, Reg32, CL`

Der Wert, der zur Bestimmung der zu verschiebenden Positionen benutzt wird, ist immer ein 8-Bit-Wert. Er wird mit der Maske $1F UND-verknüpft, um nur die fünf niedrigerwertigen Bits zuzulassen. Dies beschränkt den Wertebereich des »Positionszählers« auf [0, 31]. Damit ist gewährleistet, dass maximal um 31 Positionen verschoben werden kann. Diese Reduktion wird im Falle der Rotationsbefehle ggf. noch weiter getrieben: Werden 16-Bit-Operanden verwendet, so wird der auf den Bereich [0,31] skalierte Wert nochmals modulo 1/ und bei 8-Bit-Operanden modulo 9 genommen. Dies reduziert den Wertebereich auf

die jeweilige Operandengröße ([0,16] bei Word-Operanden, [0,8] bei Byte-Operanden) und verhindert, dass mehrere unnötige Rotationszyklen mit identischem Ergebnis durchgeführt werden. ACHTUNG: Bei Word- und Byte-Operanden ist es möglich, bei den Befehlen RCL und RCR um eine Position mehr zu rotieren, als der Operand Bits hat! Grund: Das carry flag hat hier die Funktion eines »zusätzlichen« Bits, sodass formal Words hier 9 Bits breit sind und Bytes 8 – zumindest, was die Rotation angeht.

Ungeachtet der Reduktion der Anzahl zu verschiebender Bits auf den maximalen Wertebereich [0,31] bei den Shift-Befehlen kann es (nur bei diesen!) vorkommen, dass er dennoch zu groß ist: Wenn 16- oder 8-Bit-Operanden zum Einsatz kommen. In diesem Falle sind sowohl der Inhalt des Zieloperanden wie auch alle Statusflags undefiniert.

Statusflags

Ist die im zweiten (bzw. bei SHLD und SHRD: dritten) Operanden übergebene Zahl zu verschiebender Positionen Null, so werden keine Flags verändert.

Bei allen Verschiebebefehlen nach links (ROL, RCL, SHL, SAL und SHLD) um eine Position zeigt das overflow flag einen »Vorzeichenwechsel« durch die Verschiebung an. Dies erfolgt, indem vor der Verschiebung das MSB und das benachbarte, niedrigerwertige Bit XOR-verknüpft werden und das Ergebnis im OF abgelegt wird. Ein Beispiel: In einem 32-Bit-Feld ist das Bit 31 das »Vorzeichen«, wenn man die Bits als Koeffizienten einer LongInt auffasst. Durch die Verschiebung nach links um eine Position (Multiplikation mit 2) wird dieses MSB herausgeschoben und Bit 30, das ursprünglich benachbarte Bit, avanciert zum »neuen« Vorzeichenbit. Haben somit Bit 31 und Bit 30 des unveränderten Bitfeldes den gleichen Zustand, ändert sich das »Vorzeichen« durch die Verschiebung nicht, andernfalls sehr wohl. Die XOR-Verknüpfung von Bit 31 und Bit 30 erzeugt das korrekte Ergebnis: OF = Bit_{31} XOR Bit_{30}.

(In manchen Dokumentationen ist zu lesen, dass für das OF das MSB und das CF geXORt werden! Das ist zweifellos richtig, wenn man den Zustand nach der Operation zur Definition heranzieht. Da das CF in diesem Fall das ursprüngliche MSB – und somit das ursprüngliche Bit 31 im Beispiel – enthält und das neue MSB das ursprünglich dem MSB benachbarte Bit – Bit 30 im Beispiel –, sind beide Aussagen identisch. Ich selbst bevorzuge die Definition anhand des Ausgangszustands, da

mir auf diese Weise der Effekt, »Vorzeichenwechsel«, deutlicher nachvollziehbar erscheint, als wenn man ein MSB mit einem CF verknüpft!)

Auch bei den Verschiebebefehlen um eine Position nach rechts (ROR, RCR, SHR, SAR und SHRD) zeigt das OF nach der Operation einen »Vorzeichenwechsel« an, indem das MSB und das niedrigerwertige, benachbarte Bit XOR-verknüpft werden (also z.B. Bit 31 und 30 in einem 32-Bit-Feld; ACHTUNG: hier wird tatsächlich der Zustand *nach* der Operation betrachtet, also wenn das »neue Vorzeichen« bereits im MSB vorliegt und das »alte« rechts daneben; hier würde die Erklärung mit dem Zustand *vor* der Operation weniger anschaulich sein!). Dieser »Vorzeichenwechsel« tritt bei SAR *niemals* ein, da ja das »alte« MSB durch die Operation in das »neue« MSB kopiert wird, die XOR-Verknüpfung also »0« ergibt. Dies ist absolut korrekt, da ja SAR eine »Vorzeichenerweiterung nach Division« durchführt, das Vorzeichen also gleich bleibt. Bei SHR dagegen hat OF immer den Wert des »alten Vorzeichens (=MSB)«, da immer eine »0« nachgeschoben wird und eine XOR-Verknüpfung eines beliebigen Zustandes mit »0« immer den Zustand selbst ergibt. (0 XOR 0 = 0 ➞ OF: »positiv« bleibt »positiv«, kein Wechsel; 1 XOR 0 = 1 ➞ OF: »negativ« wird »positiv«, Wechsel!).

Zusammengefasst heißt das: Das overflow flag zeigt nach den Verschiebebefehlen um eine Position einen ggf. aufgetretenen »Vorzeichenwechsel« (bei SAR also nie) an, bei Verschiebungen um mehr als eine Position ist OF undefiniert.

Die Flags SF, ZF, AF und PF bleiben bei allen Verschiebebefehlen außer SHLD und SHRD unverändert. Bei SHRD und SHLD werden SF, ZF und PF anhand des Ergebnisses gesetzt, AF ist undefiniert.

Das carry flag enthält grundsätzlich eine Kopie des zuletzt aus dem Bitfeld geschobenen Bits (im Falle von RCR und RCR enthält es *das* zuletzt aus dem Bitfeld geschobene Bit).

BT
BTS
BTR
BTC

Die BTx-Familie ist eine Gruppe von Bit-orientierten Befehlen, die den Zustand eines bestimmten Bits in einem Bitfeld prüfen. Der Zustand des überprüften Bits wird im carry flag gespeichert, sodass nach dem BTx-Befehl unmittelbar eine Programmverzweigung mittels bedingtem Sprung oder anderen bedingten Operationen (SETcc) erfolgen kann. Anschließend wird entweder gar nichts mehr unternommen (BT, *bit test*), das eben geprüfte Bit gesetzt (BTS, *bit test and set*), gelöscht (BTR, *bit test and reset*) oder »umgedreht« (BTC, *bit test and complement*).

Das Bitfeld kann entweder Word- bzw. DoubleWord-Größe besitzen und damit vollständig in einem Register abbildbar sein. Es ist jedoch auch möglich, »Bitstrings« zu verwenden, also Strukturen, die erheblich größer als eine Word-/DoubleWord-Variable sind. In diesem Fall müssen diese Strukturen jedoch eine Größe aufweisen, die ein ganzzahliges Vielfaches von Words bzw. DoubleWords ist.

Operanden

Alle BTx-Befehle erwarten als ersten Quelloperanden (und damit auch Zieloperanden) das Bitfeld. Dies kann entweder direkt in einem Register oder indirekt in einem Speicheroperanden enthalten sein. Der zweite Quelloperand gibt dann an, welches Bit in diesem Feld verwendet werden soll. Das kann wiederum entweder direkt durch Angabe einer Konstanten erfolgen oder indirekt über ein Register. Somit sind folgende Operandenkombinationen möglich:

- Direkte Angabe des zu prüfenden Bits durch eine Konstante, das Bitfeld liegt direkt in einem Register vor:
  `BTx Reg16, Const8; BTx Reg32, Const8`
- Direkte Angabe des zu prüfenden Bits durch eine Konstante, das Bitfeld liegt indirekt in einem Speicheroperanden vor:
  `BTx Mem16, Const8; BTx Mem32, Const8`
- Indirekte Angabe des zu prüfenden Bits durch eine Register-Konstante, das Bitfeld liegt direkt in einem Register vor:
  `BTx Reg16, Reg16; BTx Reg32, Reg32`
- Indirekte Angabe des zu prüfenden Bits durch eine Register-Konstante, das Bitfeld liegt indirekt in einem Speicheroperanden vor:
  `BTx Mem16, Reg16; BTx Mem32, Reg32`

Wird ein Register als erster Quelloperand verwendet, so enthält dieses Register bereits das gesamte Bitfeld. Daher können in diesem Fall nur die Bits 0 bis 15 (bei Word-Registern) bzw. 0 bis 31 (bei DoubleWord-Registern) angesprochen werden. Die BTx-Befehle würdigen diese Tatsache, indem sie als zu prüfende Bitposition den Modulus 16 (Word-Register) bzw. Modulus 32 (DoubleWord-Register) des im zweiten Quelloperanden (Const8, Reg16, Reg32) übergebenen Wertes als gewünschte Bitposition nehmen.

Wird dagegen als erster Quelloperand eine Speicherstelle verwendet, so wird dem Befehl die Adresse auf ein Word bzw. DoubleWord übergeben. Diese Adresse kann jedoch nicht nur als Zeiger auf ein Word/DoubleWord aufgefasst werden, sonder als Adresse auf das erste Word/DoubleWord eines Feldes aus Words/DoubleWords, in dem wei-

tere Bits verzeichnet sind. Daher wird sie als »Bitbasis« (*bit base*) bezeichnet. Das aber bedeutet, dass erheblich mehr als 16 bzw. 32 Bits angesprochen werden können. Die im zweiten Quelloperanden übergebene Bitposition wird daher nicht mehr auf den Wertebereich eines Words bzw. DoubleWords beschränkt. Die bedeutet aber, dass sie nun umgerechnet werden muss in einen als Offset (»Bitoffset«, *bit offset*) bezeichneten Wert, der angibt, im wievielten Word/DoubleWord beginnend mit der bit base das zu prüfende Bit steht und an welcher Position in diesem Word/DoubleWord es sich befindet.

Mit dem im zweiten Operanden über ein Register übergebenen Wert lassen sich 2^{16} = 65.536 Bits (Word-Register) und 2^{32} = 4.294.967.296 Bits (DoubleWord-Register) adressieren. Da die BTx-Befehle den Inhalt dieser Register als vorzeichenbehaftete Integer interpretieren, lassen sich somit Bitpositionen von -32.768 bis +32.767 (Word-Register) bzw. -2.147.483.648 bis +2.147.483.647 angeben (was bedeutet, dass Bits »vor« und »hinter« der bit base angesprochen werden können). Die bereits angesprochene Berechnung des Offset erfolgt nun bei Word-Operanden durch

$$\text{Offset} := 2 \cdot (\text{Operand div } 16); \text{ Position} := \text{Operand mod } 16.$$

Die Integer-Division des im zweiten Operanden übergebenen Wertes durch 16 gibt an, in welchem Word das gewünschte Bit verzeichnet ist (Bits 0 bis 15 in Word 0, Bits 16 bis 31 in Word 1 etc.). Dieser Wert mit 2 multipliziert gibt an, wie viele Bytes zur bit base addiert werden müssen, um das Word zu identifizieren, das das gewünschte Bit enthält. Der Modulus 16 des gewünschten Bits, also quasi der Rest, der bei der Offset-Berechnung mittels DIV übrig bleibt, gibt dann die Position des Bits in diesem Word an. Bitte beachten Sie, dass der Offset je nach übergebenem Wert positiv oder negativ sein kann, die Position jedoch nicht!

Analog erfolgt die Berechnung bei DoubleWord-Operanden:

$$\text{Offset} := 4 \cdot (\text{Operand div } 32); \text{ Position} := \text{Operand mod } 32$$

Nach dieser Berechnung addieren die BTx-Befehle nun den Offset zur bit base und laden das so identifizierte Word/DoubleWord in die internen Arbeitsregister. Dann wird das Bit an der Stelle *Position* geprüft und ggf. verändert, bevor das Resultat ggf. wieder zurückgeschrieben wird.

Zweierlei gibt es noch zu berücksichtigen. Erstens: Bei Verwendung der Byte-Konstanten als zweitem Operator erfolgt diese Adressberechnung nicht! Vielmehr wird das an der im ersten Operanden übergebenen Stel-

le stehende Word/DoubleWord geladen und der Modulus 16 bzw. 32 der Konstante als Bitposition verwendet.

Zweitens: Diese Art der Bit-Identifizierung ist gefährlich! Nachdem bei Verwendung von DoubleWord-Operanden 4.294.967.296 Bits angesprochen werden können, sind Bitfeld-Strukturen bis 536.870.912 Byte = 512 MByte möglich (8 Bits pro Byte)! Das bedeutet, dass es sehr leicht zu Schutzverletzungen kommen kann, falls man nicht erheblich aufpasst! Diese können darin begründet sein, dass z.B. die Datensegmente nicht groß genug sind, negative Offsets verwendet werden, ohne die bit base entsprechend anzupassen, oder sog. »memory-mapped I/O register« angesprochen werden: Schnell ist ein falscher Wert in das Register geschrieben, vor allem, wenn er berechnet wird! Intel empfiehlt daher, die Adressberechnung nach den obigen Formeln selbst vorzunehmen, das entsprechende Word/DoubleWord mittels MOV in ein Register zu laden und dann die »Register-Version« der BTx-Befehle zu nutzen.

Manche Assembler erlauben, dass bei der indirekten Bitprüfung (Speicherstellen) mittels direkter Angabe des Prüfbits (Const8) auch Bitpositionen größer 15 (Word-Speicherstellen) bzw. größer 31 (DoubleWord-Speicherstellen) angegeben werden können. Dazu berechnet der Assembler anhand der oben angeführten Formeln ein Offset und eine Position aus der Konstanten. Die Position ist dann wieder im Bereich 0 bis 15 bzw. 0 bis 32 und wird zur »neuen« Const8. Der Offset wird zur bit base addiert. Dies ist gleichbedeutend mit einer automatischen Verschiebung der bit base um den Offset; die neue bit base ist dann die neue Adresse des Speicheroperanden. Das alles ist für den Programmierer vollkommen transparent. Man kann dieses Verhalten des Assemblers nur anhand der Tatsache erkennen, dass sich jeweils Speicheradresse und Wert der Const8 im Assembler-Quelltext und im Assemblat unterscheiden.

Statusflags

Das carry flag erhält den Zustand des geprüften Bits. Alle anderen Flags sind undefiniert, sollten also nicht ausgewertet werden. Nach Setzen des carry flag löscht BTR das geprüfte Bit, BTS setzt es und BTC negiert es.

BSF
BSR

BSF und BSR sind zwei Operationen, die in einem Bitfeld das erste gesetzt Bit suchen. BSF, *bit scan forward*, beginnt hierbei am LSB, dem *least significant bit*, an Position 0 im Bitfeld und sucht in Richtung MSB, dem *most significant bit* an Position 15 (Word-Operanden) bzw. 31 (DoubleWord-Operanden). BSR, *bit scan reverse*, sucht umgekehrt beginnend

mit dem MSB in Richtung LSB. Die Suche bricht ab, wenn entweder ein gesetztes Bit gefunden oder das gesamte Bitfeld durchsucht wurde. Wurde ein gesetztes Bit gefunden, wird dessen Position in den Zieloperanden eingetragen. Andernfalls gilt der Inhalt des Zieloperands als undefiniert.

Operanden Die Befehle sind zwei der Ausnahmen, in denen Operand #1 nicht gleichzeitig Quell- und Zieloperand sind, sondern nur Zieloperand. Hierbei kann es sich nur um ein Register handeln. Er nimmt die Position auf, an der das erste gesetzte Bit gefunden wurde. Der erste und einzige Quelloperand ist somit Operand #2. Über ihn kann das zu prüfende Bitfeld entweder direkt via Allzweckregister oder indirekt über eine Speicherstelle übergeben werden. Somit sind folgende Instruktionen möglich:

- Direkte Angabe des Bitfeldes über ein Register
  `BSx Reg16, Reg16; BSx Reg32, Reg32`
- Indirekte Angabe des Bitfeldes über einen Speicheroperanden
  `BSx Reg16, Mem16; BSx Reg32, Mem32`

Statusflags Wenn der Wert des Bitfeldes »0«, d.h. kein Bit gesetzt ist, wird das zero flag gesetzt, andernfalls gelöscht. Alle anderen Statusflags gelten als undefiniert, können also nicht ausgewertet werden.

1.1.5 Operationen zum Datenaustausch

Da in diesem Abschnitt die Kommunikation mit dem Speicher beschrieben wird, empfiehlt es sich, die Kapitel »Speicherverwaltung« auf Seite 394 und »Adress- und Operandengrößen« auf Seite 765 sowie »Stack« auf Seite 385 und »Ports« auf Seite 827 durchgelesen zu haben.

MOV Einer der wohl wichtigsten Befehle des Befehlssatzes überhaupt ist der MOV-Befehl. Er ist dafür verantwortlich, ein Datum von einer Stelle zu einer anderen zu bewegen. Das Mnemonic MOV, *move data*, ist allerdings etwas missverständlich: Bewegt wird nur eine Kopie des Datums, das Datums selbst bleibt an seinem Ursprungsort unverändert erhalten.

Operanden Gemäß seiner Bedeutung akzeptiert der Befehl sehr viele Operandentypen und -kombinationen.

- Kopieren einer Konstanten in ein Allzweckregister
  `MOV Reg8, Const8; MOV Reg16, Const16; MOV Reg32, Const32`

- Kopieren einer Konstanten an eine Speicherstelle
  `MOV Mem8, Const8; MOV Mem16, Const16; MOV Mem32, Const32`
- Kopieren eines Allzweckregisterinhaltes in ein Allzweckregister
  `MOV Reg8, Reg8; MOV Reg16, Reg16; MOV Reg32, Reg32`
- Kopieren des Inhalts einer Speicherstelle in ein Allzweckregister
  `MOV Reg8, Mem8; MOV Reg16, Mem16; MOV Reg32, Mem32`
- Kopieren eines Allzweckregisterinhaltes an eine Speicherstelle
  `MOV Mem8, Reg8; MOV Mem16, Reg16; MOV Mem32, Reg32`
- Kopieren eines Speicherinhalts in den Akkumulator
  `MOV AL, Mem8; MOV AX, Mem16; MOV EAX, Mem32`
- Kopieren eines Speicherinhalts in den Akkumulator
  `MOV Mem8, AL; MOV Mem16, AX; MOV Mem32, EAX`
- Kopieren eines Allzweckregisterinhaltes in ein Segmentregister
  `MOV SReg, Reg16;`
- Kopieren eines Speicherinhaltes in ein Segmentregister
  `MOV SReg, Mem16;`
- Kopieren eines Segmentregisterinhaltes in ein Allzweckregister
  `MOV Reg16, SReg;`
- Kopieren eines Segmentregisterinhaltes an eine Speicherstelle
  `MOV Mem16, SReg;`

Es gibt noch Erweiterungen des MOV-Befehls, die den Zugriff auf die Kontroll- und Debug-Register des Prozessors ermöglichen. Dieser Zugriff ist jedoch nur unter Privilegstufe 0 möglich, sodass die entsprechenden Erweiterungen als privilegierte Befehle gelten und im Abschnitt »Verwaltungsbefehle« besprochen werden.

Der MOV-Befehl kann *nicht* dazu benutzt werden, Daten in das CS-Segmentregister zu schreiben. Der Versuch, dies zu tun, erzeugt eine invalid opcode exception (#UD). Das CS-Register kann nur im Rahmen von JMP, CALL, RET und IRET-Befehlen von außen verändert werden.

Falls mit dem MOV-Befehl ein Selektor in ein Segment-Register geladen werden soll, muss dieser valide sein. Im protected mode heißt das, dass er auf einen gültigen Eintrag in der global descriptor table (GDT) oder der aktuellen local descriptor table (LDT) zeigen muss. Der Prozessor kopiert in diesem Fall die Daten aus dem Deskriptor in den nicht zugänglichen Cache des Segmentregisters und führt die erforderlichen

Validierungen durch. Nullselektoren dürfen in Segmentregister geladen werden. Das bedeutet, es wird keine Exception ausgelöst, wenn das Segmentregister beschrieben wird. Der nächste Zugriff auf das so selektierte »Nullsegment« jedoch erzeugt eine general protection exception (#GP).

Gemäß der im Anhang unter »Standard-Adress- und Operandengrößen« genannten Bedingungen erzeugen Assembler in 32-Bit-Umgebungen Befehlssequenzen mit einem in diesem Fall überflüssigen operand size override prefix, wenn Daten zwischen einem Segmentregister und einem Allzweckregister ausgetauscht werden, da hier Nicht-Standard-Daten (16-Bit-Word in 32-Bit-Umgebung) Verwendung finden. Dies ist zwar kein Problem und der Prozessor arbeitet absolut korrekt; dennoch stellt das Präfix hier ein cycle penalty dar, das immerhin einen Takt kostet. Man kann dies verhindern, indem man als Allzweckregister ein 32-Bit-Register angibt. Die meisten Assembler gehen dann von Standarddaten aus und kopieren die unteren 16 Bits aus dem Allzweckregister in das Segmentregister oder umgekehrt. Prozessoren vor dem Pentium Pro lassen bei dem Kopieren des Segmentregister-Inhalts in das niedrigerwertige Word des Allzweckregisters den höherwertigen Anteil unangetastet, Prozessoren ab dem Pentium Pro dagegen setzen ihn auf Null.

Falls mit dem MOV-Befehl das SS-Register verändert werden soll, werden bis nach der Ausführung des folgenden Befehls alle Interrupts unterbunden. Dies erfolgt, um nach einem Neuladen des SS-Registers auch das dazugehörige ESP-Register neu laden zu können, ohne von Interrupts, die ja vom Stack Gebrauch machen, gestört zu werden. Die Nutzung eines »alten« Stack-Pointers in einem »neuen« Stacksegment würde mit sehr hoher Wahrscheinlichkeit zu Problemen führen.

Statusflags Die Statusflags werden durch MOV nicht verändert.

MOVSX
MOVZX Die Befehle *move with sign extension*, MOVSX, und *move with zero extension*, MOVZX, sind Abarten des MOV-Befehls, die im Rahmen des Kopierens den Wertebereich des Datums vorzeichenerweitert (MOVSZ) bzw. vorzeichenlos (MOVZX) auf den des nächst»höheren« Datums erweitern (ShortInt zu SmallInt, SmallInt zu LongInt bzw. Byte zu Word, Word zu DoubleWord).

Operanden

Als Zieloperanden kommen Allzweckregister in Frage, als Quelloperand ein Allzweckregister oder Speicherstellen:

- Kopieren eines Allzweckregisterinhaltes in ein Allzweckregister unter Erweiterung des Wertebereiches
  `MOV Reg16, Reg8; MOV Reg32, Reg8; MOV Reg32, Reg16`
- Kopieren des Inhaltes einer Speicherstelle in ein Allzweckregister unter Erweiterung des Wertebereiches
  `MOV Reg16, Mem8; MOV Reg32, Mem8; MOV Reg32, Mem16`

Statusflags

Statusflags werden nicht verändert.

XCHG

MOV hat einen Nachteil: Es überschreibt den Inhalt des Zieloperanden mit dem Inhalt des Quelloperanden. Sollen dagegen die Inhalte zweier Operanden ausgetauscht werden, ist ein Platz (= Register oder Speicherstelle) nötig, an dem temporär der Inhalt des Zieloperanden zwischengespeichert wird, bevor MOV in Aktion tritt. Mit dem Nachteil, dass dieser temporäre Bereich auch überschrieben wird.

Aus diesem Dilemma hilft der Befehl *exchange*, XCHG. Er tauscht die Inhalte der beiden Operanden aus, indem er einen prozessorinternen temporären Bereich nutzt. Da bei diesem Befehl erster und zweiter Operand sowohl Ziel als auch Quelle sind, spricht man bei XCHG nicht von Ziel- und Quelloperanden, sondern von erstem und zweitem Operanden.

Operanden

Beide Operanden können Register oder Speicherstellen sein. Allerdings ist eine Einschränkung, dass einer der beiden Operanden ein Register sein muss: Der direkte Austausch von Daten zwischen zwei Speicherstellen ist mit XCHG nicht möglich. Falls ein Austausch zwischen einer Speicherstelle und dem Akkumulator erfolgen soll, gibt es eine Ein-Byte-Version, die besonders effektiv ist:

- Austausch der Inhalte zweier Allzweckregister
  `XCHG Reg8, Reg8; XCHG Reg16, Reg16; XCHG Reg32, Reg32`
- Austausch der Inhalte eines Allzweckregisters mit einer Speicherstelle
  `XCHG Reg8, Mem8; XCHG Reg16, Mem16; XCHG Reg32, Mem32`
- Austausch der Inhalte des Akkumulators mit einer Speicherstelle
  `XCHG AX, Mem16; XCHG EAX, Mem32`

Formell möglich, aber in der Wirkung redundant, ist das Vertauschen der Operanden in der Operandenliste, wenn eine Speicherstelle involviert ist:

- Austausch der Inhalte eines Allzweckregisters mit einer Speicherstelle
  `XCHG Mem8, Reg8; XCHG Mem16, Reg16; XCHG Mem32, Reg32`
- Austausch der Inhalte des Akkumulators mit einer Speicherstelle
  `XCHG Mem16, AX; XCHG Mem32, EAX`

Statusflags XCHG verändert keine Statusflags.

XCHG führt einen automatischen LOCK-Befehl aus, falls ein Speicherzugriff im Rahmen von XCHG erfolgt, unabhängig davon, ob der LOCK-Präfix verwendet wird oder nicht! Auf diese Weise ist in jedem Fall sichergestellt, dass während XCHG nur der Prozessor Zugriff auf den Datenbus hat, der den Befehl gerade ausführt.

BSWAP XCHG kann auch dazu benutzt werden, Daten »innerhalb eines Registers« auszutauschen, z.B. `XCHG AH, AL`. Leider ist die Wirksamkeit dieses Befehls auf 16-Bit-Daten beschränkt und dazu noch auf das niedrigerwertige Word der vier Allzweckregister EAX, EBX, ECX und EDX, da nur sie über Alias verfügen, die als Operanden von XCHG akzeptiert werden.

Aus dieser Bedrängnis hilft der Befehl *byte swap*, BSWAP. Er vertauscht byteweise den Inhalt eines 32-Bit-Registers »von vorne nach hinten«:

```
Temp := Reg[07..00];
Reg[07..00] := Reg[31..24]
Reg[31..24] := Reg[07..00]
Temp := Reg[15..08]
Reg[15..08] := Reg[23..16]
Reg[23..16] := Temp
```

BSWAP ist damit der geeignete Befehl, Daten aus dem »Intel-Format« in das »Motorola-Format« zu überführen und umgekehrt (vgl.: »Little-Endian«- und »Big-Endian«-Format« auf Seite 781).

Operanden BSWAP akzeptiert als Operand nur ein 32-Bit-Register:

```
BSWAP Reg32
```

Falls BSWAP ein 16-Bit-Register übergeben wird (Nutzung des operand size override prefix), ist das Ergebnis unbestimmt!

BSWAP verändert keine Statusflags.

Statusflags

XLAT
XLATB

XLAT und XLATB sind zwei Befehle, eine »table look-up translation« durchführen. Hierunter versteht Intel, ein Byte aus einer Tabelle anhand seines Indexes auszulesen.

Operanden

Table-look-up-Befehle haben keine expliziten Operanden, da die logische Adresse der auszulesenden Tabelle über eine Registerkombination angegeben und der Index über den Akkumulator übergeben wird. Auch das Ziel ist klar: der Akkumulator. Dennoch gibt es neben der »echten«, parameterlosen Form (XLATB) auch eine »parametrische« Form, XLAT.

Die parameterlose Form des XLAT-Befehls erwartet in der Registerkombination DS:(E)BX die Adresse der Byte-Tabelle, aus der das interessierende Byte ausgelesen werden soll:

```
AL := Byte[DS:(E)BX + AL]
```

Der in AL stehende Index wird als vorzeichenloser Offset zur Tabellenbasis interpretiert, somit vorzeichenlos auf 16 bzw. 32 Bit erweitert und zu der in DS:(E)BX stehenden Adresse der Tabelle addiert. Das dadurch im Speicher lokalisierte Byte wird in AL kopiert.

Die »parametrische« Form der table look-up translation erwartet neben einer korrekt belegten Registerkombination DS:(E)BX und dem Index in AL einen explizit angegebenen Operanden. Ihr muss auf Assemblerebene *formal* ein Speicheroperand übergeben werden, *der keinerlei Funktion hat*. Der Assembler übersetzt dann den »parametrischen« Befehl XLAT automatisch in den »parameterfreien« Befehl XLATB. Die parametrische Form wird wie folgt dargestellt:

```
XLAT Mem8
```

Beachten Sie hierbei bitte, dass Mem8 lediglich ein Dummy ist. *Er spielt absolut keine Rolle!* Tatsächlich herangezogen wird XLATB und als Adresse der Tabelle die Adresse, die vorher in die Registerkombination DS:(E)BX eingetragen wurde.

Warum gibt es dann die »parametrische« Form überhaupt? Weiß ich nicht! Und Intel selbst auch nicht: »*This explicit-operand form is provided to allow documentation*«. Aber: »*However, note that the documentation provided by this form can be misleading.*«. Wozu eine Möglichkeit zur Dokumentation, wenn die zu Missverständnissen führen kann? Dann doch lieber gar keine!

Daher mein Tipp: Vergessen Sie einfach die parametrische Form von XLAT, Sie vermeiden dadurch schwer aufzufindende Programmierfehler, die daraus resultieren, dass bei oberflächlicher Betrachtung eine korrekte Nutzung eines übergebenen Operanden vorgegaukelt und vergessen wird, die Registerkombination DS:(E)BX korrekt zu beladen! Und exakt dokumentieren kann man auch anders!

Statusflags Statusflags werden durch XLAT bzw. XLATB nicht verändert.

XADD XADD, *exchange and add*, vertauscht die Inhalte des ersten und zweiten Operanden, addiert sie und schreibt das Ergebnis in den Zieloperanden zurück:

```
Temp := Destination
Destination := Destination + Source
Source := Temp
```

Operanden XADD erwartet als Quelloperanden (zweiter Operand!) ein Register, das Ziel kann entweder ein Register oder eine Speicherstelle sein:

- Austausch und Addition der Inhalte zweier Allzweckregister
  `XADD Reg8, Reg8; XADD Reg16, Reg16; XADD Reg32, Reg32`
- Austausch und Addition der Inhalte eines Allzweckregisters und einer Speicherstelle
  `XADD Mem8, Reg8; XADD Mem16, Reg16; XADD Mem32, Reg32`

Statusflags Die Statusflags werden nach XADD wie nach dem Additionsbefehl ADD gesetzt und spiegeln somit den Zustand der Addition wider.

CMPXCHG
CMPXCHG8B CMPXCHG, *compare and exchange*, und CMPXCHG8B, *compare and exchange 8 bytes*, sind zwei Befehle, die einen »bedingten Austausch« zweier Daten ermöglichen. Allerdings ist dieser bedingte Austausch nicht ganz mit den anderen bedingten Befehlen des Prozessors vergleichbar:

- Es werden keine Statusflags zur Entscheidungsfindung herangezogen, ob die Bedingung erfüllt ist oder nicht.

- Die Prüfung auf Erfüllung der Bedingung (»CMP«-Teil) und die Reaktion auf das Ergebnis (»XCHG«-Teil) erfolgen innerhalb einer Aktion.
- Die Prüfung ist auf Gleichheit der geprüften Daten beschränkt.

CMPXCHG und CMPXCHG8B sind die gleichen Befehle, auch wenn es, an ihren Operanden gemessen, nicht so zu sein scheint! Sie führen folgende Operation durch:

```
if TestValue = DestinationValue
then DestinationValue := SourceValue
else TestValue := DestinationValue
```

Das bedeutet: Sind Testwert und zu testendes Datum gleich, wird in das Ziel der Inhalt der Quelle kopiert. Sind sie es nicht, wird der Testwert mit dem zu prüfenden Datum überschrieben. CMPXCHG verwendet hierzu 8-Bit-, 16-Bit- oder 32-Bit-Daten, also Bytes, Words oder DoubleWords, CMPXCHG8B macht das Gleiche mit 64-Bit-Daten, also QuadWords.

Operanden

Wie das Ablaufschema zeigt, benötigen beide Befehle drei Datenquellen: einen Testwert, einen getesteten Wert und einen Wert, der ggf. zum Verändern des getesteten Wertes benötigt wird (»Korrekturwert«). Der getestete Wert kann dabei entweder in einem Register (bzw., bei CMPXCHG8B, einer Registerkombination) oder an einer Speicherstelle stehen. Da Intel-Prozessor-Befehle nicht mit zwei Speicheroperanden arbeiten können, muss somit sowohl der Testwert als auch der »Korrekturwert« in einem Register stehen. Für den Testwert wurde der Akkumulator reserviert, sodass dieser nicht explizit angegeben werden muss. CMPXCHG hat somit zwei explizite und mit dem Akkumulator einen impliziten Operanden:

- Bedingtes Tauschen mit Allzweckregistern als Operanden, der Testwert befindet sich in AL, AX bzw. EAX:
  `CMPXCHG Reg8, Reg8; CMPXCHG Reg16, Reg16; CMPXCHG Reg32, Reg32`
- Bedingtes Tauschen mit einer Speicherstelle als Operanden, der Testwert befindet sich in AL, AX bzw. EAX:
  `CMPXCHG Mem8, Reg8; CMPXCHG Mem16, Reg16; CMPXCHG Mem32, Reg32`

Am Beispiel von DoubleWords gezeigt führt der Befehl somit folgende Operation durch:

```
if [EAX] = [Mem32/DestReg32]
then [Mem32/DestReg32] := [SourceReg32]; ZF := 1
else [EAX] := [Mem32/DestReg32]; ZF := 0
```

Im Falle von CMPXCHG8B muss der Testwert aufgrund seiner Größe (64 Bit) in einer Registerkombination stehen: EDX:EAX. Damit bleibt für den Ziel- und Quelloperanden nur noch eine Allzweckregister-Kombination (ECX:EBX) übrig. Folglich muss einer der Operanden (der Zieloperand) eine Speicherstelle sein und explizit angegeben werden, während der andere ebenfalls implizit festgelegt ist:

- Bedingtes Tauschen, der Testwert befindet sich in EDX:EAX, der »Korrekturwert« (Quelloperand) in ECX:EBX
  `CMPXCHG8B Mem64`

CMPXCHG8B realisiert somit die folgende Aktion:

```
if [EDX:EAX] = [Mem64]
then [Mem64] := [EXC:EBX]; ZF := 1
else [EDX:EAX] := [Mem64]; ZF := 0
```

Statusflags Falls die Prüfung eine Gleichheit von Testwert und zu testendem Datum zeigt, wird das zero flag gesetzt, andernfalls gelöscht. Bei CMPXCHG werden die anderen Statusflags wie nach CMP gesetzt, bei CMPXCHG8B bleiben sie unverändert.

PUSH POP Der MOV-Befehl als der zentralste und wichtigste Befehl zum Datenaustausch mit dem Speicher adressiert immer das Standard-Datensegment (DS:), sobald ein Speicheroperand involviert ist. Mit Hilfe der segment override prefixes können auch andere Datensegmente benutzt werden, unter anderem auch das Stacksegment (SS:).

Das Stacksegment ist jedoch ein Datensegment, das sich nicht unerheblich von anderen Datensegmenten unterscheidet. Am auffälligsten ist, dass es »von oben nach unten« wächst wie die Stalagtiten in einer Tropfsteinhöhle, während andere Datensegmente »von unten nach oben« wachsen. Aber ein anderer Aspekt hebt es noch in entscheidenderem Maße von »normalen« Datensegmenten ab: Es ist der Notizzettel des Prozessors. Hier legt er wichtige Daten ab, wie z.B. Rücksprungadressen, wenn Unterprogramme aufgerufen werden, oder Parameter, die an Unterprogramme übergeben werden sollen.

Es verwundert daher nicht, dass es »MOV«-Befehle gibt, die dieser Sonderfunktion Rechnung tragen und auf die spezielle Funktion des Stacks eingehen. Zwei dieser Befehle sind PUSH und POP, die ein Datum »auf den Stack PUSHen«, sprich schreiben, oder »vom Stack POPpen«, sprich entfernen.

Der Stack wird dabei genauso behandelt wie das Gebilde, nach dem er benannt ist: wie ein Stapel. Das bedeutet: Man kann mit PUSH und POP nur ein Datum auf den Stapel legen oder das oberste Datum von ihm entfernen!

Bitte beachten Sie, dass das nur für die speziellen »Stack-Befehle« PUSH und POP gilt. Natürlich kann das Stacksegment auch wie jedes andere Datensegment über eine logische Adresse (SS:Offset), z.B. mit dem MOV-Befehl, angesprochen werden.

Da mit PUSH und POP immer das Datum auf der Spitze des Stapels angesprochen wird, braucht die Adresse nicht explizit angegeben zu werden! Das ist auch der entscheidende Vorteil gegenüber der MOV-Version, die ja jeweils die Adresse benötigt. Der Prozessor besitzt zwei Register, die die aktuelle Stackspitze verwalten: Das Segmentregister SS: und das Stackpointer-Register (E)SP.

Die Funktion der Befehle ist einfach: PUSH dekrementiert (!) zunächst den Inhalt von (E)SP, sodass SS:(E)SP nun auf eine freie Stelle auf dem Stack zeigt, der Stack also gewachsen ist. An diese Position wird nun der Inhalt des Operanden von PUSH kopiert. POP geht den umgekehrten Weg: Zunächst wird der Inhalt von SS:(E)SP in den Operanden von POP kopiert und dann (E)SP inkrementiert (!). Der Stack ist geschrumpft.

Bitte denken Sie an die Stalagtiten, wenn Sie im Kopf die Stackspitze verschieben! Der Stack wächst zu niedrigeren Adressen, weshalb die Adresse in (E)SP dekrementiert werden muss und er schrumpft zu höheren Adressen, was ein Inkrementieren bewirkt.

Wer dekrementiert/inkrementiert? Und mit welchem Wert? Der Prozessor! Wie jedes andere Segment auch hat das Stacksegment ein Flag, das die »Standard-Datengröße« bestimmt. Im Codesegment ist es das D-Bit, bei Daten- und Stacksegmenten das B-Bit im jeweiligen Deskriptor. Ist es im Stacksegment gesetzt, so benutzt der Prozessor das 32-Bit-ESP-Register als Stackpointer, der Stack ist dann »32-bittig«. Ist es gelöscht, repräsentiert den Stackpointer das 16-Bit-SP-Register, der stack ist dann 16-bittig.

Die Anzahl zu addierender Bytes liest er aus dem D-Flag des Code- bzw. dem B-Flag des Datensegmentes – je nachdem, woher der Operand kommt (und welches Segment damit betroffen ist) und welches

Datum betroffen ist (Adressen oder »echte« Daten). Damit spielen auch etwaige operand size bzw. address size prefixes eine Rolle. Sind die jeweiligen Flags gesetzt oder zwingen die override prefixes dazu, addiert/subtrahiert er vier Bytes zum Stackpointer, da die Segmente dann 32-bittig ausgelegt sind. Sind sie gelöscht, werden nur zwei Bytes verwendet (16-bittig).

Das kann zu Problemen führen, wenn Daten- oder Codesegment 16-bittig ausgelegt ist, das Stacksegment jedoch 32-bittig. Bei einem PUSH von Daten oder Adressen werden dann nur zwei Bytes auf den Stack geschoben. Die neue Adresse in ESP liegt dann aber nicht an Double-Word-, sondern nur an Word-Grenzen. Diesen Sachverhalt nennt man »*misalignment*« des Stacks (»falsche Ausrichtung«). Falls der sog. alignment check eingeschaltet ist, führt das zu einer alignment check exception (#AC).

Operanden Beide Befehle können Konstanten, Inhalte von Registern oder Speicherstellen als Operanden akzeptieren (XXX steht für PUSH bzw. POP):

- PUSHen/POPpen einer Konstanten
  `XXX Const8; XXX Const16; XXX Const32`
- PUSHen/POPpen eines Registerinhalts
  `XXX Reg16; XXX Reg32`
- PUSHen/POPpen des Inhalts einer Speicherstelle
  `XXX Mem16; XXX Mem32`
- PUSHen/POPpen des Inhalts eines Segmentregisters
  `XXX CS; XXX DS; XXX ES; XXX FS; XXX GS; XXX SS`

Nachdem auch die Inhalte von Registern auf den Stack geschoben werden können, kann man auch das (E)SP-Register verwenden. Damit hat man ein Problem: (E)SP enthält den Stackpointer; dieser wird vor dem Kopieren auf den Stack dekrementiert. Wird nun der »alte« Inhalt – vor dem Dekrementieren – an die neue Stackspitze geschrieben oder der »neue« – nach dem Dekrementieren? Antwort: der alte! Dies gilt übrigens auch für Werte, bei denen zunächst die Adresse berechnet werden muss (»indirekte Adressierung«) und bei denen hierbei das (E)SP-Register involviert ist. Regel: Zuerst wird die Adresse berechnet und dann dekrementiert (und somit (E)SP verändert). Analoges gilt für POP, nur umgekehrt: Zuerst wird inkrementiert und der an der neuen Stackposition stehende Wert für die Adressberechung verwendet.

Nachdem auch Segmentregister Operand für die Befehle sein können, kann mittels POP auch ein Segmentregister geladen werden. Der hierbei verwendete Selektor muss gültig sein und auf ein gültiges Segment zeigen, da jedes Beschreiben eines Segmentregisters den Inhalt des durch den Selektor spezifizierten Deskriptors in den verborgenen Teil des Segmentregisters schreibt. Damit aber ist die Validierung des Segments verbunden inklusive der Prüfung der Privilegien. Es kann zwar, ohne eine exception auszulösen, ein Null-Selektor in ein Segmentregister gePOPpt werden. Jeder folgende Zugriff auf dieses Register führt dann jedoch zu einer general protection exception (#GP).

Das CS-Register kann durch POP nicht neu geladen werden! Um einen neuen Wert in das CS-Register zu schreiben, ist der RET-Befehl erforderlich.

Statusflags

Statusflags werden durch PUSH und POP nicht verändert.

PUSHA
POPA
PUSHAD
POPAD

PUSHA, *push all general-purpose registers*, und POPA, *pop all general-purpose registers*, sind zwei Befehle, die die Register aller Allzweckregister auf den stack schieben oder von dort holen. Sie erfüllen somit folgende Aufgabe »in einem Rutsch«:

```
PUSHA: Temp := (E)SP
       PUSH (E)AX
       PUSH (E)CX
       PUSH (E)DX
       PUSH (E)BX
       PUSH Temp
       PUSH (E)BP
       PUSH (E)SI
       PUSH (E)DI

POPA:  POP (E)DI
       POP (E)SI
       POP (E)BP
       ADD (E)SP, (4)2       ; (E)SP-Inhalt vor PUSHA; daher ADD!
       POP (E)BX
       POP (E)DX
       POP (E)CX
       POP (E)AX
```

Ob durch PUSHA/POPA die 16- oder 32-Bit-Register verwendet werden, entscheidet die Umgebung, in der die Befehle aufgerufen werden.

PUSHAD, *push all general-purpose registers as doublewords*, und POPAD, *pop all general-purpose registers as double words*, sind Alias von PUSHA und POPA. Manche Assembler erzwingen bei Verwendung von PUSHA/POPA Befehlssequenzen für 16-Bit-Register, bei PUSHAD/POPAD für 32-Bit-Register. In der Regel aber werden die Assembler die korrekten Befehlssequenzen anhand der aktuellen Umgebung setzen.

Operanden PUSHA, POPA, PUSHAD und POPAD haben keine Operanden.

Statusflags PUSHA, POPA, PUSHAD und POPAD verändern die Statusflags nicht.

IN
OUT

Die bislang betrachteten Befehle ermöglichten einen Datenaustausch mit dem Speicher, sofern nicht prozessorintern Daten in einzelnen Registern ausgetauscht wurden. Doch neben der Kommunikation mit dem Speicher beherrscht der Prozessor natürlich auch die Kommunikation mit externen »Geräten« wie Druckern, Modems etc. Hierzu bedient er sich der Ports. (Zur Beschreibung von Ports vgl. »Ports« auf Seite 827.) Und was beim Speicher der MOV-Befehl ist, ist bei Ports das Befehlspaar IN – OUT. Hierbei übernimmt IN das Lesen eines Datums »aus dem Port«, während OUT ein Datum »über ein Port ausgibt«.

Die Kommunikation mit der Peripherie ist bei den modernen Betriebssystemen Sache des Betriebssystems! Es allein hat und muss die Kontrolle über den Zugriff haben. Daher können Sie in der Regel im protected mode Ports direkt nicht mehr ansprechen, sondern müssen Betriebssystemfunktionen benutzen. Dies ist Teil der Schutzkonzepte im protected mode.

Die Port-Adresse ist bei beiden Befehlen vorgegeben: Sie wird im DX-Register abgelegt. Der Datenaustausch erfolgt über den Akkumulator.

Beachten Sie hierbei, dass unabhängig von der Umgebung (16-Bit- bzw. 32-Bit-Umgebungen) die Port-Adresse immer 16-bittig ist, da die IA-32-Architektur von Intel »nur« 65.536 Ports zulässt. Somit reichen zu einer Adressierung der Ports Words und damit ein Word-Register aus. Allerdings haben die Ports 0 bis 255 eine herausragende Bedeutung, sodass auch eine Byte-Konstante als Portadresse übergeben werden kann.

Dagegen bestimmt der Akkumulator die Datengröße des zu übertragenden Datums: Wird AL benutzt, werden Bytes ausgetauscht, bei AX Words und bei EAX DoubleWords.

IN und OUT können somit folgende Operanden annehmen: *Operanden*

- Adressierung des Ports durch eine Konstante
```
IN  AL, Const8; IN  AX, Const8; IN  EAX, Const8
OUT Const8, AL; OUT Const8, AX; OUT Const8, EAX
```
- Adressierung des Ports durch das DX-Register
```
IN  AL, DX; IN  AX, DX; IN  EAX, DX
OUT DX, AL; OUT DX, AX; OUT DX, EAX
```

Die Statusflags werden durch IN und OUT nicht beeinflusst. *Statusflags*

1.1.6 Operationen zur Datenkonvertierung

Unter Datenkonvertierung versteht die CPU die Überführung einer Integer in eine andere, konkret das Erweitern des Wertebereiches einer Integer. Somit ist der umgekehrte Weg nicht realisiert.

Lassen Sie sich durch die Begriffe »Byte«, »Word«, »DoubleWord« und »QuadWord« in den Mnemonics der folgenden Befehle nicht verwirren! Die Befehle verarbeiten vorzeichenbehaftete Zahlen, berücksichtigen also ein Vorzeichen. Daher können sie ShortInts in SmallInts, SmallInts in LongInts und LongInts in QuadInts konvertieren. Die Konversion führt nur dann mit vorzeichenlosen Integers (Bytes, Words, DoubleWords) zu korrekten Ergebnissen, wenn deren MSB nicht gesetzt ist! Der Grund dafür ist, dass es die einzige und eigentliche Aufgabe aller Konvertierungsbefehle ist, eine Vorzeichenerweiterung (»sign extension«) durchführen – und sonst nichts!

CBW
CWD
CDQ

CBW, *convert byte to word*, CWD, *convert word to double word*, und CDQ, *convert double word to quad word* führen genau diese Vorzeichenerweiterung durch. Das Datum muss dazu im Akkumulator stehen. CBW kopiert nun das MSB (= Vorzeichenbit 7) der ShortInt in AL achtmal in die Bitpositionen 8 bis 15, CWD das MSB (= Vorzeichenbit 15) der SmallInt aus AX 16-mal in DX (!) und CDQ das MSB (Vorzeichenbit 31) der LongInt aus EAX 32-mal in EDX. Das Ergebnis sind eine SmallInt in AX, eine LongInt in DX:AX und eine QuadInt in EDX:EAX mit korrektem Vorzeichen. (Vergleiche hierzu den Abschnitt »Codierung von Integers« auf Seite 801.)

CWDE

CWD wurde noch zu einer Zeit realisiert, als die Prozessoren noch nicht über 32-Bit-Register verfügten und daher als zwei 16-Bit-Teile behandelt werden mussten. Daher legt CDW die LongInt in der Registerkom-

bination DX:AX ab. Mit dem Aufkommen der 32-Bit-Prozessoren wurde daher ein Zwilling für CWD geschaffen, der die LongInt in ein 32-Bit-Register ablegt: CWDE, convert word to DoubleWord in extended register. Dieser Befehl entnimmt dem MSB der SmallInt in AX das Vorzeichen und kopiert es 16-mal in die Bitpositionen 16 bis 31 des EAX-Registers.

Operanden Die Befehle haben keine expliziten Operanden. Sie verwenden implizite Quell-/Ziel-Operanden: das AL-/AX-Register (CBW), das AX-/DX:AX-Register (CWD), das AX-/EAX-Register (CWDE) bzw. das EAX/EDX:EAX-Register (CDQ).

Statusflags Statusflags werden nicht verändert.

CBW und CWDE besitzen den gleichen Opcode ($98), sind also identisch. Das mag zunächst verwundern, ist aber dennoch logisch. Mit CBW soll eine ShortInt in eine SmallInt konvertiert werden, den vorzeichenbehafteten Standardwert von 16-Bit-Prozessoren. CWDE konvertiert eine SmallInt in eine LongInt, den vorzeichenbehafteten Standardwert von 32-Bit-Prozessoren. Das bedeutet, beide Befehle konvertieren jeweils in einen Standardwert in einer bestimmten Umgebung. In 32-Bit-Umgebungen (32-Bit-Prozessoren und 32-Bit-Betriebssystem) ist die Standard-Datengröße 32 Bits, in 16-Bit-Umgebungen (16-/32-Bit-Prozessoren und 16-Bit-Betriebssystem) ist sie 16 Bits. Daher wird in 32-Bit-Umgebungen der Opcode $98 durch das Mnemonic CWDE, in 16-Bit-Umgebungen durch CBW repräsentiert. Nutzt man nun in 16-Bit-Umgebungen das Mnemonic CWDE, so wird dem Opcode der operand size override prefix vorangestellt. Analog wird er verwendet, wenn in 32-Bit-Umgebungen das Mnemonic CBW benutzt wird. In der Regel erfolgt das für den Assemblerprogrammierer transparent durch den Assembler.

Analoges erfolgt übrigens mit CWD und CDQ – beide haben den Opcode $99. In 16-Bit-Umgebungen wird diesem Opcode der operand size override prefix vorangestellt, wenn CDQ verwendet wird, in 32-Bit-Umgebungen, wenn CWD genutzt wird.