Grundlagen der Adreßrechnung

Im folgenden wollen wir elementare Adressierungsweisen und Rechengänge betrachten, mit denen effektive Adressen berechnet werden können. Manche sind von so grundsätzlicher Bedeutung, daß sie in einem modernen Universalrechner unbedingt zur Architektur gehören, bei anderen ist es hingegen eine Frage der Philosophie (z. B. der Entscheidung zwischen CISC- und RISC-Auslegung), ob sie in den Maschinenbefehlen vorgesehen werden, oder ob erwartet wird, daß sie (in der Regel über die Compiler) mit elementaren Befehlen ausprogrammiert werden.

Absolute und direkte Adressierung
Alle Adreßangaben sind Direktwerte in Befehlen (Abbildung 1.1). *Das* klassische Beispiel: Zuse Z3. Der Vorteil: Einfachheit. Die wesentlichen Nachteile:

- Datenstrukturen und Programme sind nicht verschieblich (relocatable); sie müssen vielmehr an festen Plätzen gespeichert werden,
- es ist nicht möglich, Programmschleifen auf Elemente höher aggregierter Datenstrukturen anzuwenden (z. B. auf Vektoren und Matrizen (Arrays).

Ein Trick, um die Nachteile zu überwinden:
Selbstmodifizierende Programme. Adreßrechnung modifiziert die Adreßangaben in den Zugriffs - bzw. Verzweigungsbefehlen. Selbstmodifizierende Programme sind aber - aus guten Gründen - seit längerem nicht mehr in Mode.

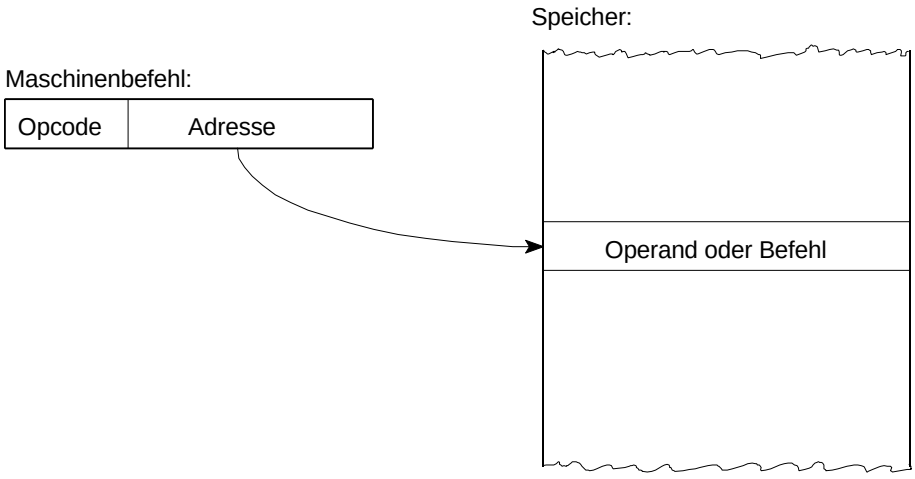


Abbildung 1.1 Absolute und direkte Adressierung

Universelle Adreßregister (Registeradressierung)
Prinzip: die betreffende Adresse wird aus einem Register entnommen (Abbildung 1.2). "Universell" bedeutet hier, daß der Registerinhalt ohne weiteres in beliebige Verarbeitungsabläufe einbezogen werden kann. Bei "echten" Universalregistern ist dies ohne weiteres gegeben; Adreßregister im eigentlichen Sinne (wie bei CDC 6600 oder Motorola 68k) müssen zumindest für Lese- und Schreibzugriffe zugänglich sein.

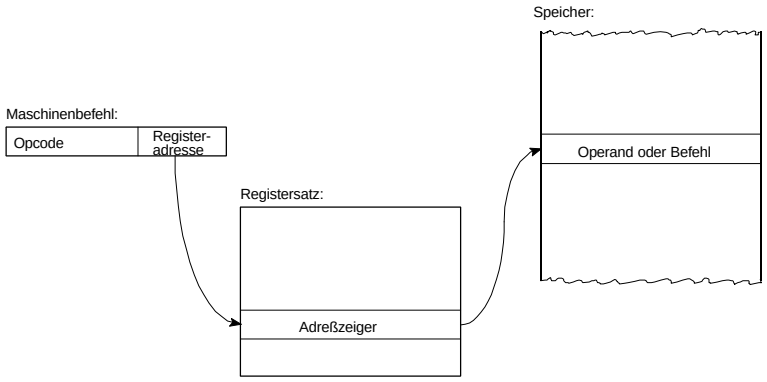


Abbildung 1.2 Registeradressierung

Indirekte Adressierung ("Adresse von Adresse")

Eine Adreßangabe ist z. B. als Direktwert im Befehl oder als Registerinhalt gegeben. Der hiermt adressierte Register- oder Speicherinhalt wird aber nicht als Datenstruktur oder als Befehl, sondern als weitere Adresse interpretiert (Abbildung 1.3). Mit dieser Adresse wird dann der eigentliche Zugriff ausgeführt. Das Prinzip entspricht an sich der Registeradressierung, nur ist man bei der Unterbringung von Adressen nicht auf den Registersatz beschränkt, sondern kann den Arbeitsspeicher dazu ausnutzen.

Mehrstufige indirekte Adressierung ("Adresse von Adresse von Adresse...")

Die ursprüngliche Adreßangabe adressiert eine Speicherposition, deren Inhalt adressiert eine weitere Speicherposition usw. Der letzte dieser Adreßzeiger verweist schließlich auf den gewünschten Inhalt. Es gibt zwei Möglichkeiten, das Ende dieser indirekten Adressierung zu kennzeichnen:

- durch eine Endekennung in den Adreßzeigern (Adreßzeiger muß länger sein als die Adreßangabe (Beispiel: PDP-1),
- durch Angabe der Stufenzahl (Levels of Indirection).

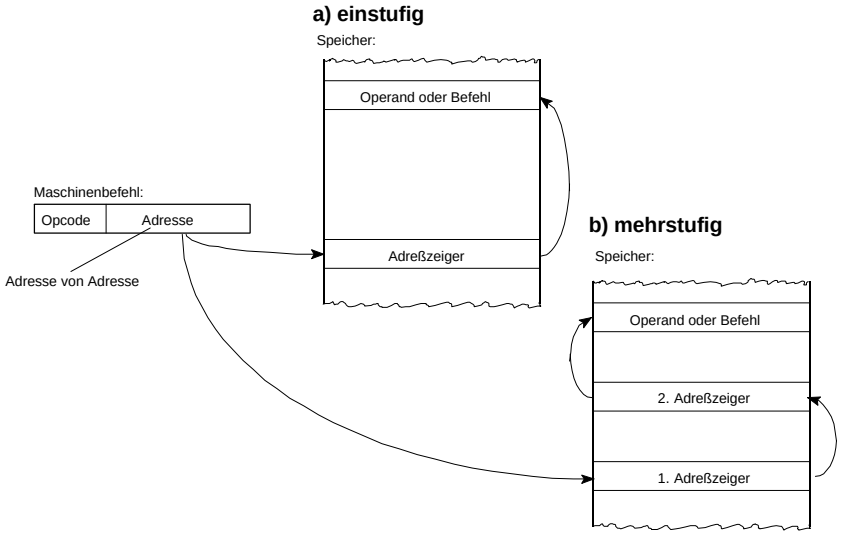


Abbildung 1.3 Indirekte Adressierung

Was sind die einfachsten wirklich universellen Adressierungsweisen?

1. Registeradressierung + Direktwert-Ladebefehle. Jedem Zugriff geht das Laden eines Adreßregisters bzw. eine entsprechende Adreßrechung voraus. Die einfachste Form: das Laden der Adresse als Direktwert.
2. einstufige indirekter Adressierung + Absolutadressierung (um die Speicherzellen erreichen zu können, die die Adreßzeiger enthalten).

Adressierung in herkömmlichen Hochleistungsrechnern
Die bisherige Erfahrung hat gezeigt, daß man mit den ganz einfachen Prinzipien nicht auskommt. Weshalb? - Zum einen verschlechtert sich die Verarbeitungsleistung bedeutend. Allein im Interesse der Verschieblichkeit von Programmen und Daten wäre für jede Verzweigung, für jeden Unterprogrammruf und für nahezu jeden Datenzugriff eine Adreßrechnung notwendig, die jeweils mehrere Befehle erfordert. Zum anderen wären viele Register durch Adressen belegt[*)]; man könnte sie nicht als Schnellspeicher für Daten verwenden. Deshalb werden elementare Rechengänge der Adreßrechnung hardwareseitig unterstützt. Im folgenden werden wir typische elementare Rechengänge am Beispiel des Protected-Modus der IA-32-Architektur erläutern.

*): herkömmliche Registersätze haben 8, 16 oder 32 Register. Bei ca. 8 Registern (x86/IA-32, Hitachi H8/300) ist eine komfortable Unterstützung der Adreßrechnung unumgänglich, bei 16 oder 32 Registern (S/360, VAX, Sparc, Mips usw.) hat es sich als zweckmäßig erwiesen, die einfachste Form der Adreßrechnung (Basis + Displacement) architekturseitig vorzusehen.

Adressierung in ganz neumodischen Hochleistungsrechnern (IA-64)
Hier kommt man tatsächlich mit Einfachem aus: IA-64 hat nur Registeradressierung. Grundsätzliche Zusammenhänge im Überblick:

- der Registersatz ist sehr groß (128 Register),
- alle Operationsbefehle betreffen Registerinhalte (Dreiadreßschema Register-Register-Register),
- der Stack Frame der aktuellen Prozedur wird typischerweise im Registeratz aufgebau,
- es geht vorwiegend darum, Daten blockweise zwischen Speicher und Registersatz auszutauschen,
- es soll der Software ermöglicht werden, vorbeugend (spekulativ) zu laden, also Daten zwecks späterem Gebrauch schon heranzuschaffen, während noch andere Verarbeitungsvorgänge ablaufen[*)],
- Adreßrechenvorgänge können mit den Verarbeitungsabläufen parallelisiert werden[*)],
- die Hardware unterstützt deshalb nur einfache Lade. und Speicherabläufe auf Grundlage der Registeradressierung, ergänzt um eine wahlweise automatische Adreßerhöhung nach dem jeweiilgen Zugriff (zum Adreßregister (Basisregister) wird entweder ein Festwert oder der Inhalt eines weiteren Registers addiert).

*): das ist hier Sache des Compilers (der die reinste Wunderwaffe sein muß, um die Architektur wirklich auszunutzen...).

Elementare Rechengänge der Adreßrechnung

1. Relativadressierung: Basis + Displacement

2. Indexadressierung: Basis + Index + Displacement

3. Skalierte Adressierung: Basis + (Index · Skalierung) + Displacement

Struktur und Unterbringung von Adreßangaben
Effektive Adressen sind *natürliche (vorzeichenlose)* Binärzahlen. Basis- und Indexadressen sind in Registern untergebracht, Displacementangaben als Direktwerte in Befehlen. Für die *Länge* von Adressen (in Bits) gilt üblicherweise, daß effektive Adresse, Basisadresse und Indexadresse gleich lang sind; die Länge entspricht der architekturseitigen Verarbeitungsbreite. Für Displacements sind teilweise auch kürzere Angaben vorgesehen. *Displacements* sind zumeist *ganze* Binärzahlen (*mit* Vorzeichen). Skalierungsangaben sind natürliche Binärzahlen und - wenn überhaupt vorgesehen - als Direktwerte in Befehlen untergebracht.

Vorsicht, Falle:

- kürzere Displacements werden vor der Adreßrechnung *vorzeichengerecht* erweitert; wir können mit einer 8-Bit-Angabe also im Bereich von -128 bis +127 Bytes "um die Basisadresse herum" adressieren; ein Offset von beispielsweise 200 Bytes erfordert dann die nächstlängere Displacement-Angabe.
- natürliche und ganze Binärzahlen können bezüglich Addition und Subtraktion mit jeweils denselben Befehlen verarbeitet werden, jedoch nicht bezüglich Multiplikation und Division. Folglich sind Skalierungswerte mit Befehlen "Multiplizieren vorzeichenlos" zu verrechnen.

Datenadressierung am Beispiel IA-32 (Protected-Modus)
Wir betrachten im folgenden die Adressierungsweisen des Protected-Modus bei 32-Bit-Adressierung. So können wir wichtige Prinzipien am konkreten Beispiel näher kennenlernen. Grundsätzlich sind in dieser Architektur alle bisher erwähnten Rechengänge vorgesehen (Abbildungen 1.4, 1.5). Der allgemeine Rechengang:

Effektive Adresse = Basis + (Index · Skalierung) + Displacement.

Durch Weglassen einzelner Anteile ergeben sich insgesamt 9 Kombinationsmöglichkeiten (Tabelle 1.1).

Adreßrechnung und Segmentierung
Eine Besonderheit der IA-32-Architektur: die Adressierung ist gleichsam auf die hardwareseitig unterstützte Segmentierung aufgesetzt - Adresse Null ist nicht die physische Speicheradresse Null, sondern die Adresse Null des ausgewählten Segments.

Rechnen wir die Segmentierungsvorkehrungen hinzu, so haben wir - im Vergleich zu anderen Architekturen - eine zusätzliche Stufe der Adreßrechnung. Das ergibt zusätzliche Möglichkeiten zum Tricksen. In der Praxis nutzt man diese aber kaum aus: die weitaus meisten Anwendungen der 32-Bit-Betriebsweise (Win32, OS/2, UNIX usw.) verwenden die Segmentierung nur, um einen flachen Speicheradreßraum bereitzustellen. Das bedeutet, es gibt praktisch nur ein einziges Segment, das mit Adresse 0 beginnt und an Adresse 4 294 947 295 endet; die Basisadresse seitens der Segmentierung ist somit immer Null. Damit entspricht IA-32 jenen Architekturen, die keine Segmentierung haben. Deshalb werden wir der folgenden Beschreibung die Segmentierung vernachlässigen (Segmentadresse 0 = effektive Adresse 0; keine Tricks).

Adressierungsweise	Adreßrechnung bei flachem (unsegmentiertem) Speichermodell
1. Nur Displacement	Displacement bezogen auf Basisadresse 0
2. Nur Basisadresse	Nur Basisadresse (aus Register)
3. Basis + Displacement	Basisadresse + Displacement
4. Index + Displacement	Index + Displacement bezogen auf Basisadresse 0
5. (Index · Skalierung) + Displacement	(Index · Skalierung) + Displacement bezogen auf Basisadresse 0
6. Basis + Index	Basisadresse + Index
7. Basis + (Index · Skalierung)	Basisadresse + (Index · Skalierung)
8. Basis + Index + Displacement	Basisadresse + Index + Displacement
9. Basis + (Index · Skalierung) + Displacement	Basisadresse + (Index · Skalierung) + Displacement

Tabelle 1.1 Adressierungsweisen der IA-32-Architektur

Bildung der effektiven Adresse

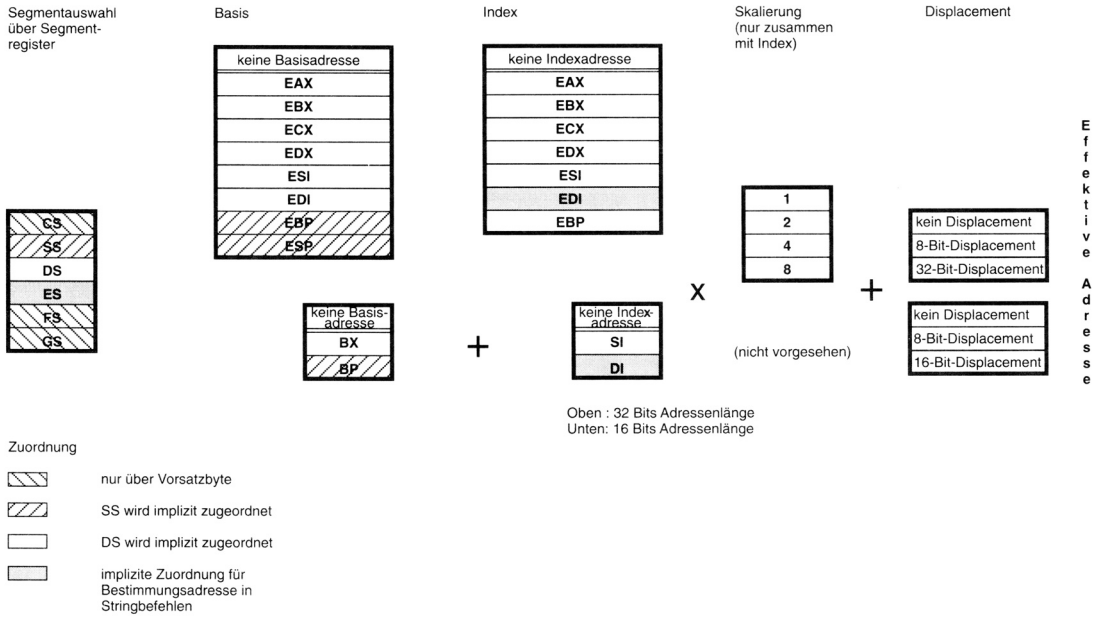
Die einzelnen Komponenten der Adreßrechnung sind aus Tabelle 1.2 genauer ersichtlich. Tabelle 1.3 zeigt, wo die entsprechenden Angaben untergebracht sind.

Komponente	Quelle
Basisadresse	eines der acht allgemeinen Register
Indexadresse	eines der allgemeinen Register mit Ausnahme von ESP
Skalierung	Festwert im Befehl; Werte 1, 2, 4, 8
Displacement	Festwert im Befehl, 8 oder 32 Bits

Tabelle 1.2 Die Komponenten der Datenadressierung (1)

Komponente	Quelle bei 16-Bit-Adressierung	Quelle bei 32-Bit-Adressierung
Basis	Register BX oder BP	eines der acht allgemeinen Register
Index	Register SI oder DI	eines der allgemeinen Register mit Ausnahme von ESP
Skalierung	nicht vorgesehen	Festwert im Befehl; Werte 1, 2, 4, 8
Displacement	Festwert im Befehl; 8 oder 16 Bits	Festwert im Befehl, 8 oder 32 Bits

Tabelle 1.3 Die Komponenten der Datenadressierung (2)



A bbildung 1.4 Die Bildung der effektiven Adresse am Beispiel IA-32

Im folgenden werden einige Möglichkeiten der sinnfälligen Nutzung verschiedener Adressierungsweisen beschrieben.

Statische Operanden: Nur Displacement
Der Direktwert aus dem Befehl wird als Offset zu Adresse 0 addiert. Damit lassen sich vorteilhaft (es wird kein Register belegt) statische, auf festen Adressen placierte Operanden adressieren. Diese Adressierungsweise entspricht einer Absolutadressierung im ausgewählten Segment (vgl. Abbildung 1.1).

Programmseitig berechnete Adresse (Registeradressierung)

Nur Basis
Eines der allgemeinen Register enthält die Adresse des Operanden im Segment. Diese Adressierungsweise (vgl. Abbildung 1.2) ist von Vorteil, wenn die Adresse programmseitig berechnet bzw. aus gespeicherten Tabellen entnommen wird.

Zugriff auf Elemente heterogener Datenstrukturen (Records): Basis + Displacement
Eine naheliegende Nutzung (Abbildung 1.6): das Basisregister zeigt auf den Beginn der *Record*-Struktur, das Displacement gibt die Anfangsadresse des jeweiligen Feldes an.

Erklärung:
Eine heterogene bzw. *Record*-Struktur besteht aus verschiedenartigen Elementen (ganze Binärzahlen, einzelne Bits, Zeichenketten usw.). Solche Strukturen können in vielen Programmiersprachen deklariert werden. Am häufigsten werden *statische Records* verwendet. In solchen Fällen hat im compilierten Programm jede Teilstruktur eine feste Adresse (bezogen auf den Anfang des *Record*), die somit als Direktwert im Befehl angegeben werden kann.

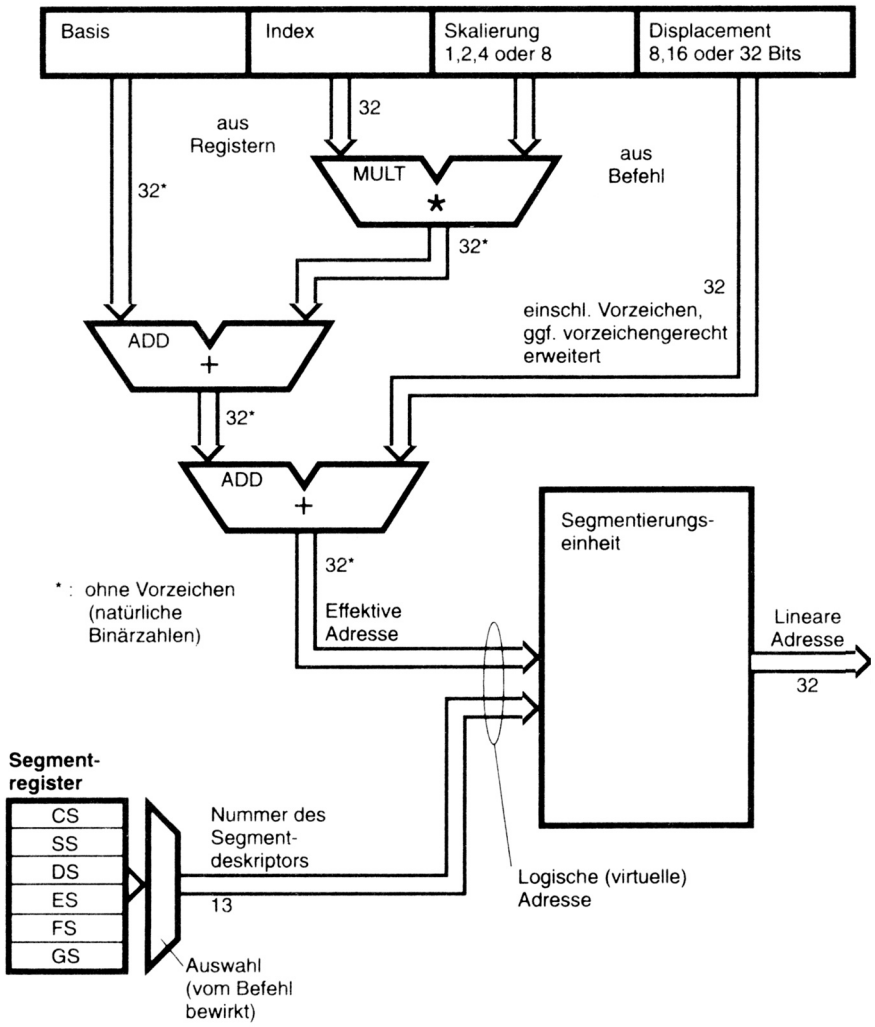


Abbildung 1.5 Zum Prinzip der Adreßrechnung

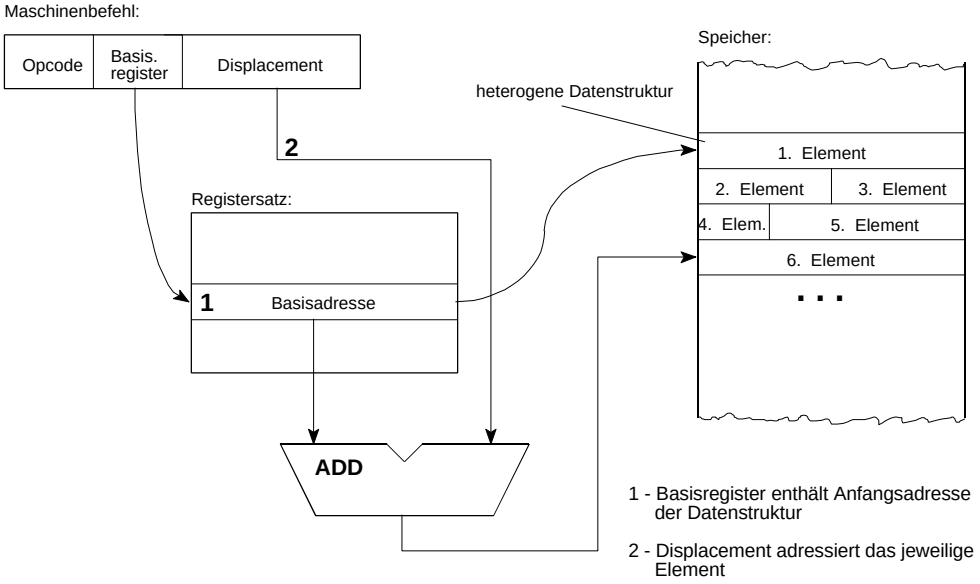


Abbildung 1.6 Basis + Displacement (1) : Zugriff auf Elemente einer heterogenen Datenstruktur

Zugriffe auf Elemente homogener Datenstrukturen (Arrays): Basis + Displacement
Eine trickreiche Nutzung, um Elemente in einem statischen, fest placierten Datenfeld zu adressieren (Abbildung 1.7): die Displacementangabe zeigt auf den Beginn des Feldes, das Basisregister enthält die Adresse des Feldelementes (die beispielsweise im Rahmen der Interpretation einer Laufanweisung (FOR-Anweisung) berechnet wurde).

Erklärung:
Eine homogene bzw. Array-Struktur besteht aus gleichartigen Elementen, auf die typischerweise in Schleifen zugegriffen wird (z. B. in FOR-Anweisungen). Jeder derartige Zugriff erfordert eine Adreßrechnung.

Die elegante Form: wir haben ein Basisregister mit der ersten Adresse des Feldes und ein weiteres Register, in das wir die jeweils berechnete Zugriffsadresse (Offset) laden.

Nun haben IA-32-Prozessoren nur wenige Register. Deshalb bringt man die Anfangsadresse des Arrays im Displacement unter und verwendet das Register, das den berechneten Offset enthält, als Basisregister. Also ein Programmiertrick (der bei IA-32 besonders gut funktioniert, weil die Displacementangabe 32 Bits lang sein darf, also den gesamten Adreßraum überstreicht - bei den typischen RISC-Maschinen mit kurzen Displacements gäbe es hier ein Problem...).

Hinweise:

1. Wenn das einzelne Element einer eindimensionalen Feldstruktur (Datentyp *Array*) aus b Bytes besteht, so errechnet sich die Offsetadresse des n-ten Elementes (n = 1,2,..), auf den Feldanfang bezogen, gemäß Offset = (n-1) · b. Für Werte n = 2, 4, 8 wird diese Rechnung implizit in der Hardware ausgeführt, wenn man die Komponente (Index · Skalierung; siehe weiter unten (Abbildung 1.7)) in die Adressierung einbezieht. Für andere Werte von n ist sie explizit zu programmieren. Dabei bietet es sich an, das Resultatregister des Rechenablaufs als Basisregister zu verwenden.

2. Die Form Basis + Displacement hat den Vorteil, daß kein weiteres Register benötigt wird. Sie hat aber den Nachteil, daß das Displacement in den Befehlen mitzuführen ist (ein Programm wird um so länger, je mehr Befehle auf die betreffende Datenstruktur zugreifen).

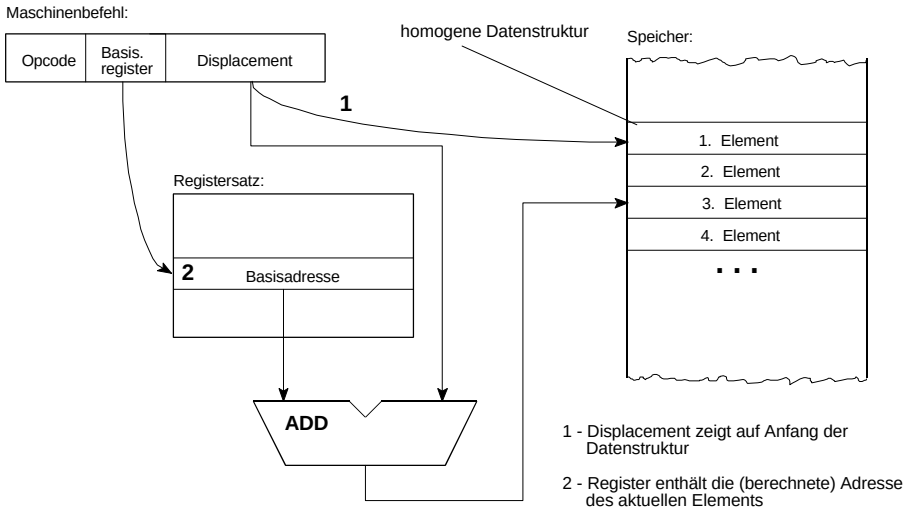


Abbildung 1.7 Basis + Displacement (2) : Zugriff auf Elemente einer homogenen Datenstruktur

Zugriff auf eindimensionale Felder: (Index @ Skalierung) + Displacement
Auf diese Weise sind statische, fest placierte Felder (*Array*-Strukturen), die aus Komponenten von 2, 4, oder 8 Bytes bestehen, vorteilhaft adressierbar (Abbildung 1.8).

Es handelt sich um ein eindimensionales Feld gleichartiger Elemente, die b Bytes lang sind (b = 2, 4, 8). Die die laufende Nummer (die Ordinalzahl) des betreffenden Elements ist gegeben. Wir berechnen zunächst aus Index und Skalierung die Offset-Adresse des n-ten Elements in Bezug auf den Feldanfang:

- Offset = (n-1) · b, falls n von 1 an läuft (1, 2, 3...),
- Offset = n · b, falls n von 0 an läuft (0, 1, 2...).

n wird in einem Register als Indexadresse bereitgestellt, b wird als Skalierung angegeben (mögliche Werte: 2, 4, 8). Die Displacementangabe zeigt (wie in Abbildung 1.7) auf den Anfang der Datenstruktur.

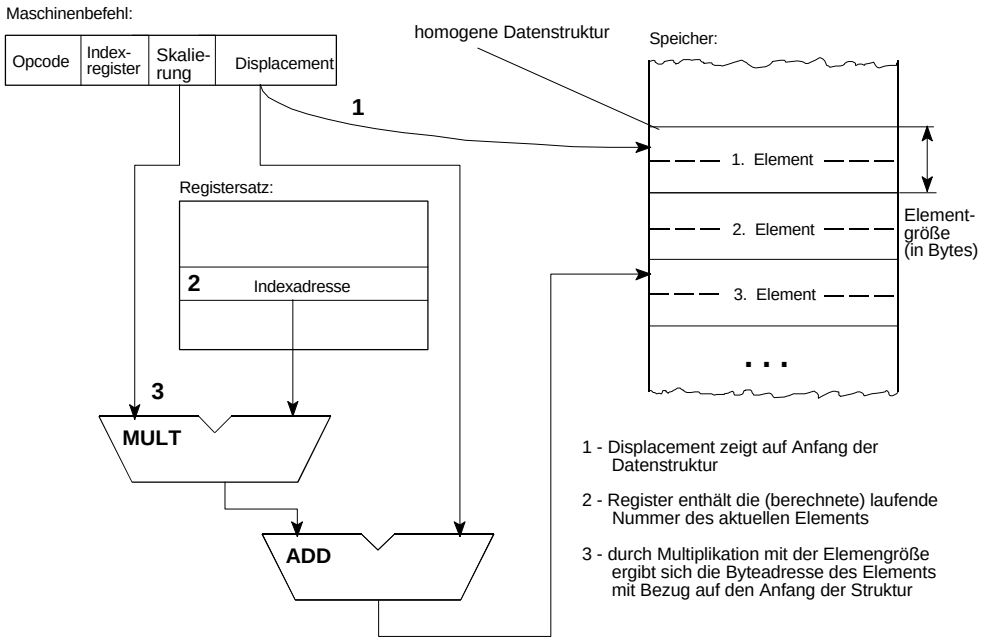


Abbildung 1.8 (Index · Skalierung)+ Displacement

Zugriff auf mehrdimensionale Feldern oder Felder, die aus Record-Strukturen bestehen: Basis + Index + Displacement
Die Register enthalten die Feldkoordinaten in Form von Ordinalzahlen, und das Displacement bezeichnet den Anfang des Feldes (das auch Teilstruktur eines *Record* sein kann). Abbildung 1.9 soll Aufbau und Adressierungsweise derartiger Datenstrukturen anhand eines einfachen Beispiels veranschaulichen.

Die allgemeine Form der Adreßrechnung:
Der Adressen-Offset (bezogen auf den Feldanfang) für ein Element eines mehrdimensionalen Feldes aus gleichartigen Elementen, die b Bytes lang sind, ist zu berechnen. Die gegebenen Feldkoordinaten seien k_1, k_2, ... ; die Anzahl der Elemente in jeder Felddimension sei C_1, C_2, ...; jeweils mit dem Wertebereich 1, 2, ... Dann gilt:

$$\text{Offset} = (k_1 - 1) \cdot b + (k_2 - 1) \cdot C_1 \cdot b + (k_3 - 1) \cdot C_1 \cdot C_2 \cdot b + ...$$

Bei einem zweidimensionalen Feld kann ein Register $(k_1 - 1) \cdot b$ enthalten und das andere $(k_2 - 1) \cdot C_1 \cdot b$.

Für zweidimensionale Felder mit Elementen von 2, 4 oder 8 Bytes läßt sich diese allgemeine Formel auf Grundlage der Adressierungsweise *Basis + (Index ▯ Skalierung) + Displacement* implementieren. Eine typische Anwendung: Matrizenrechnung.

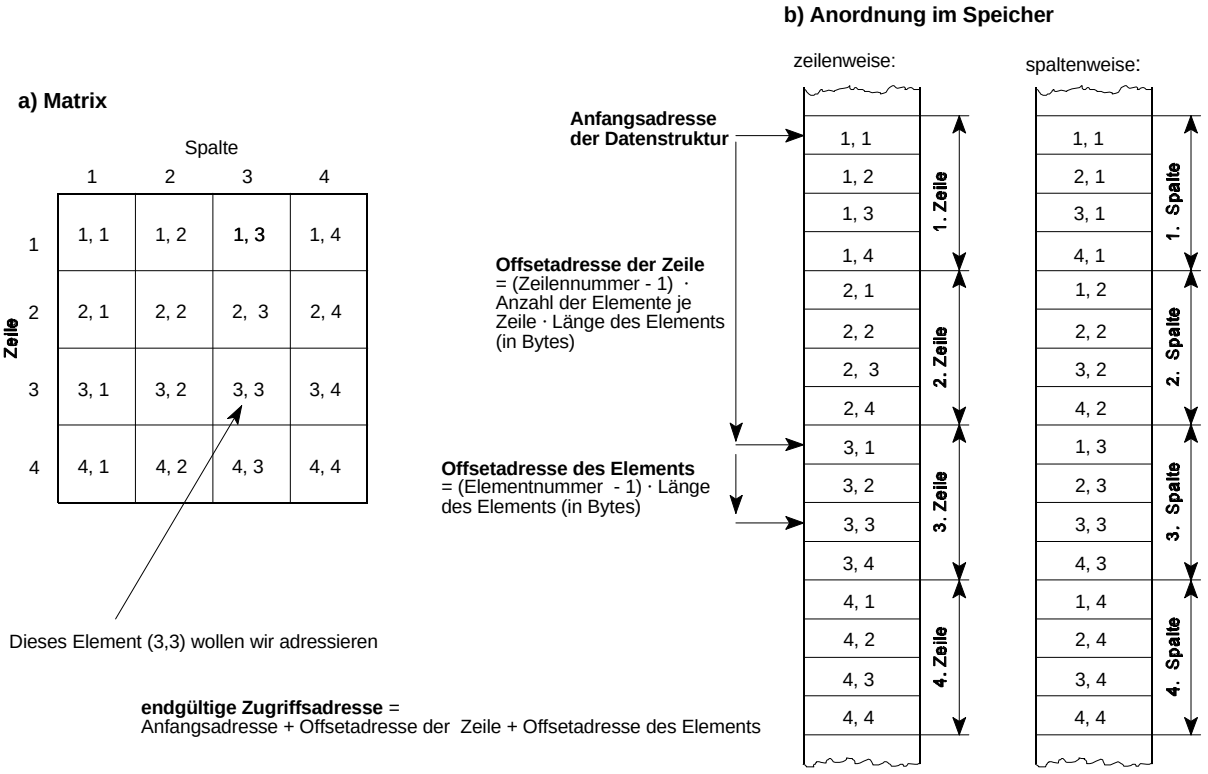


Abbildung 1.9 Eine Matrix als Beispiel einer zweidimensionalen Datenstruktur

Es gibt viele Möglichkeiten, die erforderliche Adreßrechnung mit den Vorkehrungen der IA-32-Architektur zu implementieren. Begnügen wir uns mit drei Beispielen:

1. Beispiel:

- Anfangsadresse der Datenstruktur => Displacement,
- Offsetadresse der Zeile => Basis (Rechengang wird ausprogrammiert),
- Offsetadresse des Elements => Index · Skalierung.

2. Beispiel:

- Anfangsadresse der Datenstruktur => Displacement,
- Zeilennummer · Anzahl der Elemente der Zeile (ausprogrammiert) => Index,
- Offsetadresse des Elements => Basis (Rechengang wird ausprogrammiert - liegt nahe, wenn sich Adreßrechnung im Schleifen auf Addition zurückführen läßt (z. B. bei Zugriffen auf aufeinanderfolgende Elemente in FOR-Schleifen)).

3. Beispiel:

Nur Basisadresse. Alles ausprogrammieren. Womöglich die schnellste Variante. Adreßrechnung im Befehl kostet typischerweise zusätzliche Maschinenzyklen (z. B. dann, wenn ein Indexregister angesprochen wird)[*)]. Demegegenüber ist die Wahrscheinlichkeit groß, daß (in modernen Prozessoren) mehrere elementare Befehle parallel zueinander ausgeführt werden (so daß sich Adreßrechnung und eigentliche Verarbeitung überlappen (einer der klassischen Fälle des nutzbaren inhärenten Parallelismus)).

*): die Programmoptimierungshandbücher der Hersteller enthalten einschlägige Hinweise.

Zugriff auf Stack Frames: EBP + Displacement
Um auf feste Bereiche im Stack zuzugreifen, ist EBP vorteilhaft nutzbar, da in diesem Fall automatisch das Stacksegment angesprochen wird. Das Adressierungsschema entspricht Abbildung 1.6.

Hinweis:
Die Nutzung von EBP als Zeiger auf den Stack Frame (Activation Record) der aktuellen Prozedur ist Bestandteil des IA-32-Programmiermodells und wird durch weitere Befehle (ENTER, LEAVE) unterstützt. Näheres weiter unten.

Wieviele Adressierungsweisen braucht man wirklich? - die herkömmliche RISC-Philosophie
Die Erfahrung hat gezeigt, daß die Form *Basis + Displacement* vollauf ausreichend ist - allerdings unter der Voraussetzung, daß genügend Register zur Verfügung stehen (um Adreßangaben und Daten gleichzeitig halten zu können)[*)]. Die Basisadresse befindet sich in einem allgemeinen Register, und die Displacementangabe ist als Direktwert im Befehl untergebracht. Sie wird als ganze Binärzahl interpretiert und mit Vorzeichenerweiterung verrechnet. Displacement 0 bewirkt die reine Registeradressierung. Solche Displacementangaben sind üblicherweise zwischen 13 und 16 Bits lang (diese Größenordnung hat sich in der Praxis als zumeist voll ausreichend erwiesen)[**)]. Mit schnell ablaufenden elementaren Befehlen lassen sich auf dieser Grundlage alle komplexeren Adreßrechenvorgänge ausprogrammieren.

*): mindestens 16, besser 32.

**): diese Auslegung beschränkt die typischen Zugriffe auf das Schema von Abbildung 1.6; Tricks ähnlich Abbildung 1.7 scheiden grundsätzlich aus. Mit anderen Worten: Basis + Displacement unterstützt die typischen Zugriffe auf Activation Records (Stack Frames) und auf Einträge im Stack (mit Stackpointer als Basisregister). Bei Zugriffen auf homogene Datenstrukturen ist die Adreßrechnung auszuprogrammieren.

Denksportaufgabe:
Es gibt Maschinen mit kleinen Registersätzen (IA-32, Hitachi H8/300, Z80, P/F usw.). Wie gelingt es, hier komplizierte Adreßrechenvorgänge unterzubringen?

Indem man den Stack ausnutzt. Konsequent zu Ende gedacht läuft das darauf hinaus, ein e virtuelle Stackmaschine zu bauen und die Register nicht als wirkliche Universalregister, sondern nur als Arbeitsbzw. Rechenregister auszunutzen. Ggf. auch feste Auslagerungsbereiche (Swap Areas) im Arbeitsspeicher.

Befehlsadressierung
Die Befehle, die im Prozessor ausgeführt werden sollen, werden vom Befehlszähler adressiert. Dieser schaltet automatisch von Befehl zu Befehl weiter, wobei - im Falle variabel langer Befehle - die Befehlslänge automatisch verrechnet wird. Der Befehlszähler liefert grundsätzlich von vornherein effektive Adressen, ohne weitere Adreßrechnung (wenn wir von der Segmentierung - einer IA-32-Besonderheit - absehen).

Betrachten wir zunächst den Fall, daß sich ein Programm in der Ausführung befindet. Neben der fortlaufenden Adressierung sind gelegentlich Verzweigungen auszuführen und Unterprogramme zu rufen. Wie werden die jeweiligen Befehlsadressen, zu denen verzweigt werden soll, (1) angegeben und (2) als effektive Adressen ermittelt? - Hierfür gibt es zwei Prinzipien:

1. Befehlszähler-relative Adressierung
Dieses Prinzip wird am meisten verwendet. Die Adreßangabe im Verzweigungs- oder Unterprogrammrufbefehl wird als ganze Binärzahl interpretiert und - erforderlichenfalls vorzeichengerecht erweitert - zum aktuellen Inhalt des Befehlszählers addiert (der Befehlszähler zeigt dabei anfänglich auf den Folgebefehl). Diese Displacement- bzw. Offset-Angaben sind in verschiedenen Längen vorgesehen. Bedingte Verzweigungen haben Displacements von 8...16 Bits Länge. In vielen Fällen dienen solche Verzweigungen dazu, eine Schleife aus vergleichsweise wenigen Befehlen zu schließen, so daß man mit 8 Bits (Verzweigung im Bereich von -128 bis +127 Bytes relativ zur Anfangsadresse des Folgebefehls) oft gut auskommt. Man trachtet im allgemeinen danach, in einem gegebenen Format soviele Adreßbits wie möglich unterzubringen. Wenn es in der Architektur nur eine einzige feste Befehlslänge gibt, kann man solche Displacements für Befehle anstatt für Bytes vorsehen.

So haben die bedingten Verzweigungen der SPARC- Architektur ein 22-Bit-Displacementfeld. Da der einzelne Befehl 4 Bytes lang ist, wird die Displacementangabe vor der Verrechnung mit 4 multipliziert. Damit ergibt sich ein Byte-Displacement von 24 Bits Länge, was selbst für extreme Nutzungsfälle bei weitem ausreicht (wer schreibt ein einzelnes Programm, das 16 Millionen Bytes lang ist und wo zudem laufend "wilde" Verzweigungen vorkommen?).

2. Absolutadressierung
Die effektive Adresse wird direkt geliefert und in den Befehlszähler geladen. Eine solche Adresse kann aus folgenden Quellen stammen: (1) Direktwert im Befehl, (2) Register (auch: Register + Displacement), (3) vom Befehl aus adressierte Speicherposition.

Stackorganisation und -adressierung
Das Stack- (Kellerspeicher-) Prinzip ist in der Informatik von grundsätzlicher Bedeutung, namentlich was die Programmiersprachen, die Compiler und die Systemsoftware angeht. Manche Architekturen haben Vorkehrungen, um Stacks zu unterstützen, manche nicht (dann müssen Stacks als normale Datenbereiche vorgesehen werden, deren Verwaltung mit elementaren Befehlen auszuprogrammieren ist). Die Grundprinzipien bleiben stets die gleichen. Wir werden sie im folgenden etwas näher betrachten und am Beispiel IA-32 veranschaulichen.

Grundlagen
Ein Stack ist eine Speicheranordnung, die eine gewisse Anzahl gleich langer Informationsstrukturen (*Stack-Elemente*) aufnehmen kann. Es gibt keinen wahlfreien Zugriff, sondern die Speicheranordnung wird implizit von einem Adreßzähler (*Stackpointer*) adressiert.

Stackzugriffe
Es gibt nur zwei grundlegende Zugriffsabläufe:

- ein *Push*-Ablauf legt ein Element auf den Stack,
- ein *Pop*-Ablauf entnimmt das zuletzt (vom letzten Push) auf den Stack gelegte Element (beim nächsten Pop wird dann das vom vorletzten Push abgelegte Element entnommen usw.).

Die Stack-Organisation wird deshalb gelegentlich auch als LIFO (Last In, First Out) bezeichnet.

Wachstumsrichtung
Es ist eine reine Konventionsfrage, ob bei Push-Abläufen der Inhalt des Stackpointers erhöht und bei Pop-Abläufen vermindert wird oder umgekehrt.

In vielen Architekturen *wachsen Stacks immer in Richtung niedere Adressen*, d. h. der Stackpointer zeigt anfänglich immer auf die höchstwertige Adresse. Sein Inhalt wird bei Push-Abläufen vermindert und bei Pop-Abläufen erhöht.

Zähl- und Zugriffsreihenfolge
Ebenso ist es eine reine Konventionsfrage, ob bei einem Push zunächst der Stackpointer verändert und dann das neue Element gespeichert wird oder umgekehrt.

Beispielsweise zeigt in der IA-32-Architektur ein Stackpointer *immer auf das oberste Element im Stack* (Top of Stack, TOS), nicht auf die erste freie Stackposition. Bei einem Push wird deshalb der Stackpointer-Inhalt zunächst vermindert (Ausdehnungsrichtung!); dann wird das Element gespeichert. Umgekehrt wird bei einem Pop das Element entnommen und dann der Stackpointer-Inhalt erhöht (Zugriffsprinzip: Predecrement/Postincrement).

Verfeinerungen

Stack-relative Adressierung
Es ist oft von Vorteil, wenn man zu Elementen des Stack auch wahlfrei zugreifen kann. So kann man auch untere Elemente im Stack erreichen, ohne die oberen zuvor entfernen zu müssen. Solche Zugriffe beziehen sich zweckmäßigerweise auf den Stackpointer, so daß das erste, zweite usw. Element im Stack für Lese- und Schreibzugriffe zugänglich ist, wobei der Stackpointer nicht verändert wird (explizite Stackzugriffe nach dem Prinzip Basis + Displacement mit dem Stackpointer als Basisadreßregister).

Variabel lange Stackelemente
In den meisten Architekturen, so sie überhaupt Stacks vorsehen, sind alle Elemente in einem Stack *gleich lang*. Kürzere Angaben werden zwecks Ablage auf dem Stack entsprechend erweitert.

Stack Frames
Ein Stack Frame ist ein fester Bereich im Stack. Er dient vor allem dazu, die statischen Variablen des laufenden Programms aufzunehmen.

Statische und dynamische Variable
Statische Variable werden im Programmtext deklariert (jeder Variablenname wird angegeben, und es wird ihm ein Datentyp zugewiesen). Beispiel (wir verwenden der Anschaulichkeit halber eine an Pascal und Ada orientierte Syntax):

```
Artikel_Nr: Integer;                    -- 4 Bytes (ganze 32-Bit-Binärzahl)
Bezeichnung: String(64);                -- 64 Bytes (Zeichenkette)
Preis: Unpacked_BCD(16);                -- 16 Bytes (BCD-Zahl)
Länge, Breite, Höhe: Small_Integer;     -- je 2 Bytes (ganze 16-Bit-Binärzahlen)
Gewicht, Spezifisches_Gewicht: Float;   -- je 4 Bytes (32-Bit-Gleitkommazahlen)
usw.
```

Jeder diese Variablen muß der Compiler entsprechenden Speicherplatz zuweisen.

Dynamische Variable entstehen hingegen im Laufe der Verarbeitung (also ohne daß sie der Programmierer ausdrücklich deklarieren muß). Beispiel: der Programmierer schreibt hin:

```
Gewicht := Länge * Breite * Höhe * Spezifisches_Gewicht;
```

Der Compiler muß diese Formel in eine Folge von Maschinenbefehlen umsetzen (hierbei sind u. a. verschiedene Datentypen ineinander zu wandeln). Da die einzelnen Befehle nur ganz elementare Operationen ausführen können, fallen im Verlauf der Rechnung Zwischenergebnisse an. Dies sind die dynamischen Variablen, die typischerweise auf dem Stack abgelegt werden.

Sowohl statische als auch dynamische Variable werden im Stack untergebracht
Das muß nicht unbedingt so sein, hat sich aber bewährt. Und zwar vor allem deshalb, weil man gern Programme in Programme schachtelt (Unterprogrammtechnik). Dann liegt es nahe, die verfügbare Speicherkapazität im Sinne eines Stack zu verwalten und den Speicher vom oberen Ende her aufzufüllen (vgl. weiter unten Abbildung 1.11). Zuerst kommt der Stack Frame des ersten Programms. Darüber (in Richtung zu den niederen Adressen hin) werden die gerade aktuellen dynamischen Variablen auf den Stack gelegt. Wenn nun das Programm ein Unterprogramm aufruft, kommt dessen Stack Frame auf den Stack, darüber werden dessen dynamische Variable abgelegt usw.

Das Zugriffsproblem
Gemäß dem Rechenablauf wächst oder schrumpft der Stack. Andererseits sind aber immer die gleichen statischen Variablen zu adressieren. Würde man sich aber stets auf den Stackpointer (als Basisadresse) beziehen, so würden sich bei jedem Zugriff andere Displacements zu den statischen Variablen ergeben. Deshalb sieht man typischerweise ein weiteres Adreßregister vor, den sog. Frame Pointer oder Base Pointer (Abbildung 1.10).

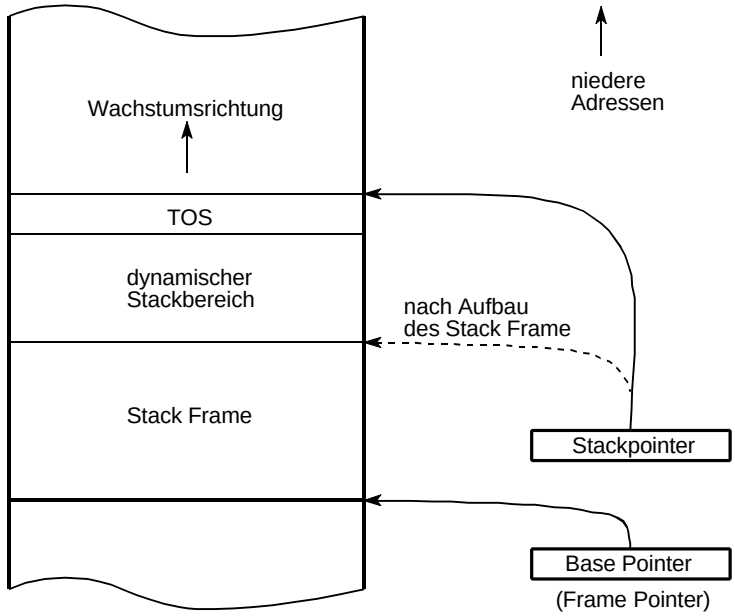


Abbildung 1.10 Stack-Organisation mit Stack Frame

Erklärung:
Der Base Pointer (Frame Pointer) zeigt stets auf den Anfang des aktuellen Stack Frame (d. h. auf das Wort an der jeweils höchsten Adresse). Alle Inhalte des aktuellen Stack Frame sind somit über negative Displacements (bezogen auf den Base Pointer) erreichbar.

Ruft das aktive Programm seinerseits ein Unterprogramm, so wird der aktuelle Inhalt des Stackpointers in den Base Pointer übernommen, und oberhalb des dynamischen Bereichs des rufenden Programms wird der Stack Frame des gerufenen aufgebaut. Spitzfindigkeiten erläuterm wir im folgenden anhand von UNIX und der Architektur IA-32.

Grundlagen der systemseitigen Speicherverwaltung
Die Speicherverwaltung hat die Aufgabe, den einzelnen Programmen im Arbeitsspeicher eine angemessene Speicherkapazität zur Verfügung zu stellen. Wieviel Speicher (z. B. in Bytes ausgedrückt) braucht aber ein Programm? - Es sind unterzubringen:

- das Programm selbst,
- die zugehörigen konstanten Daten,
- Arbeits- und Übergabebereiche,
- bedarfsweise Symbol- und Verweistabellen.

In einfachen Systemen kann man die verfügbare Speicherkapazität fest aufteilen (statische Speicheraufteilung). Moderne Hochleistungssysteme sind hingegen dadurch gekennzeichnet, daß sich die Speicherbelegung ständig ändert (dynamische Speicheraufteilung). Abbildung 1.11 veranschaulicht ein Prinzip, das häufig implementiert wird.

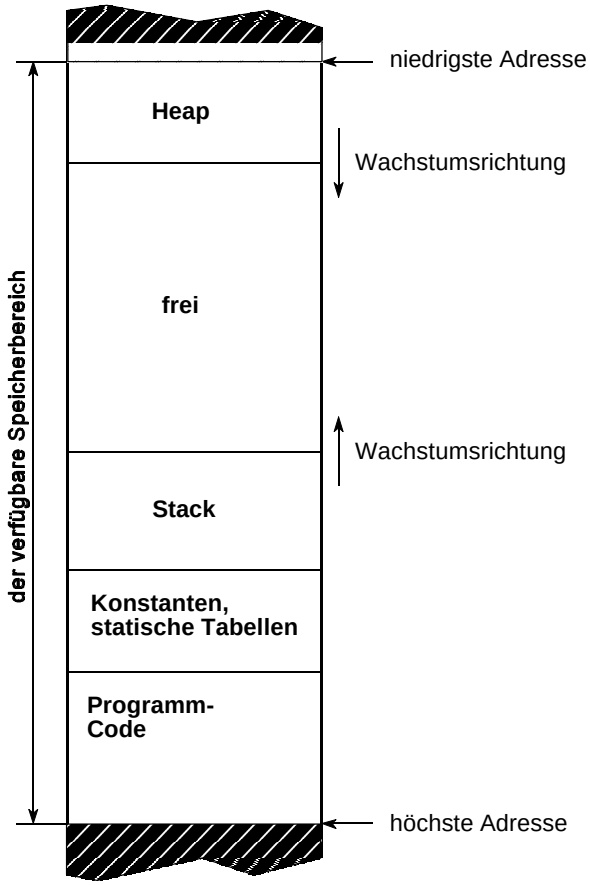


Abbildung 1.11 Zum Prinzip der Speicheraufteilung (Beispiel)

Erklärung:
Wir beginnen damit, daß ein "hinreichend" großer Speicherbereich zunächst bereitsteht. Dieser wird folgendermaßen belegt.

- der Programmcode an sich wird ganz hinten untergebracht,
- davor kommen die "statischen" - in ihrer Größe unveränderlichen Datenbereiche (Konstanten, Symboltabellen usw.),
- im Anschluß daran - zu den niederen Adressen hin - wird der Stack eingerichtet. Er nimmt dynamische Daten, Parameter, statische Variable, Zwischenergebnisse und Rückkehradressen auf. Er wächst in Richtung niederer Adressen.
- ergänzend zum Stack sieht man oft eine weitere veränderliche Struktur vor, den Heap (sprich: Hiep; wörtlich = Haufen). Der Heap wird am Anfang des Speicherbereichs angeordnet. Er wächst in Richtung höherer Adressen. Zur Verwendung von Stack und Heap siehe Tabelle 1.4.

	Stack	Heap
Nutzung (gespeichert werden...)	Rückkehradressen, lokale Daten (verschwinden bei Rückkehr aus der jeweiligen Funktion)	dynamische Daten (bleiben solange erhalten, bis sie explizit (vom Programm) wieder freigegeben werden
Belegung und Freigabe (Auf- und Abbau)	automatisch gemäß dem LIFO-Prinzip	typischerweise (vgl. Programmier-sprache C) vom Programmierer anzufordern und freizugeben
besondere Eignung	für kleinere und einfachere Datenstrukturen (zu beispielsweise 32 oder 64 Bits)	für größere und kompliziertere Datenstrukturen (z. B. von 256 Bytes an aufwärts)

Tabelle 1.4 Zur Verwendung von Stack und Heap

Sowohl Stack als auch Heap wachsen oder schrumpfen während der Ausführung des Programms. Durch die Anordnung an entgegengesetzten Enden ist stets gewährleistet, daß sich ein möglichst großer freier Bereich zwischen Stack und Heap befindet. Nur in dem - vergleichsweise unwahrscheinlichen - Fall, daß beide Strukturen wachsen und wachsen, kann es vorkommen, daß irgendwann einmal nichts mehr frei ist, daß also der Stack versucht, ein Stück des Heap zu belegen oder umgekehrt. Die Schutzvorkehrungen der Hardware bzw. das Laufzeitsystem der Software sollten dies erkennen und entsprechend reagieren (z. B. mit dem Abbruch der Programmausführung und einer entsprechenden Fehlermeldung).

Die UNIX-Stackorganisation
Für jeden Prozeß werden zwei Stacks verwaltet (Abbildung 1.12):

- der User Stack zum Aufrufen von Anwendungsprogrammen,
- der Kernel Stack zum Aufrufen der Systemfunktionen.

IA-32-Stackorganisation (1): CPU
In der zentralen Verarbeitungseinheit (CPU) wird ein Stackbereich im Speicher hardwareseitig unterstützt. Das entsprechende Segment wird über das Segmentregister SS ausgewählt. Als Stackpointer wird das Register ESP verwendet. Es zeigt auf das niedrigstwertige Byte des obersten Stack-Elementes. Stack-Elemente können 16 oder 32 Bits lang sein. Die jeweilige Länge ist programmseitig bestimmbar (unser "Normalfall": 32 Bits). Abbildung 1.13 zeigt den grundsätzlichen Aufbau eines CPU-Stacks.

Explizite Stackzugriffsbefehle
Der Stackpointer ESP kann als Quelle einer Basisadresse in Befehlen angegeben werden, zu der ein Displacement aus dem Befehl addiert werden kann. Basisadressen für Stackzugriffe können auch aus anderen Registern stammen; dann sind alle Adressierungsweisen nutzbar.

Mehrere Stacks
Für das aktuelle Programm wird in der Hardware jeweils nur ein Stack unterstützt. Es ist aber vorgesehen, daß jede Privilegebene in jeder Task einen eigenen Stack hat. Segmentselektoren und Stackpointer für die drei höheren Privilegebenen (0, 1, 2) werden in besonderen Positionen des jeweiligen Taskzustandssegments gehalten (Abbildung 1.14).

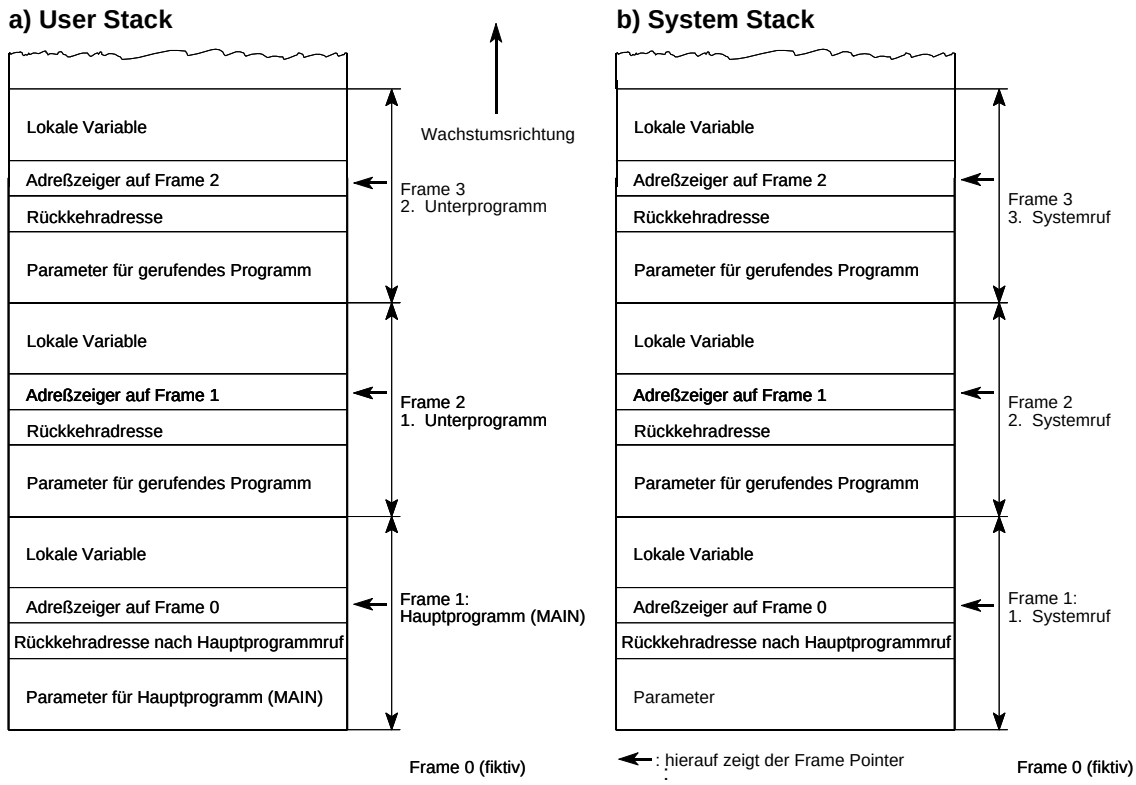


Abbildung 1.12 Die UNIX-Stackorganisation

Erklärung:
Ein UNIX-Programaufruf läuft folgendermaßen ab:

1. das rufenden Programm legt die zu übergebenden Parameter auf den Stack,
2. der Aufruf wird ausgeführt. Dabei gelangt die Rückkehradresse auf den Stack*).
3. das gerufenene Programm kopiert den bisherigen Frame Pointer auf den Stack (Adreßzeiger als Rückverweis). Typischerweise wird der aktuelle Inhalt des Stackpointers zum neuen Frame Pointer**).
4. das gerufene Programm kopiert seine lokalen Variablen in den Stack (bzw. schafft auf dem Stack soviel Platz, daß die lokalen Variablen hineinpassen),
5. der aktuelle Frame bzw. Base Pointer wird eingerichtet.

*): erste Variante: automatisch mittels CALL-Befehl (z. B. IA.32). Zweite Variante: porgrammseitig, indem der Inhalt des Adreßrettungsregisters auf den Stack gebracht wird (die typischen RISC-Maschinen (Mips, PowerPC, Alpha usw.).

**): dieser Ablauf wird gelegentlich von der Hardware unterstützt (z. B. IA-32-ENTER-Befehl).

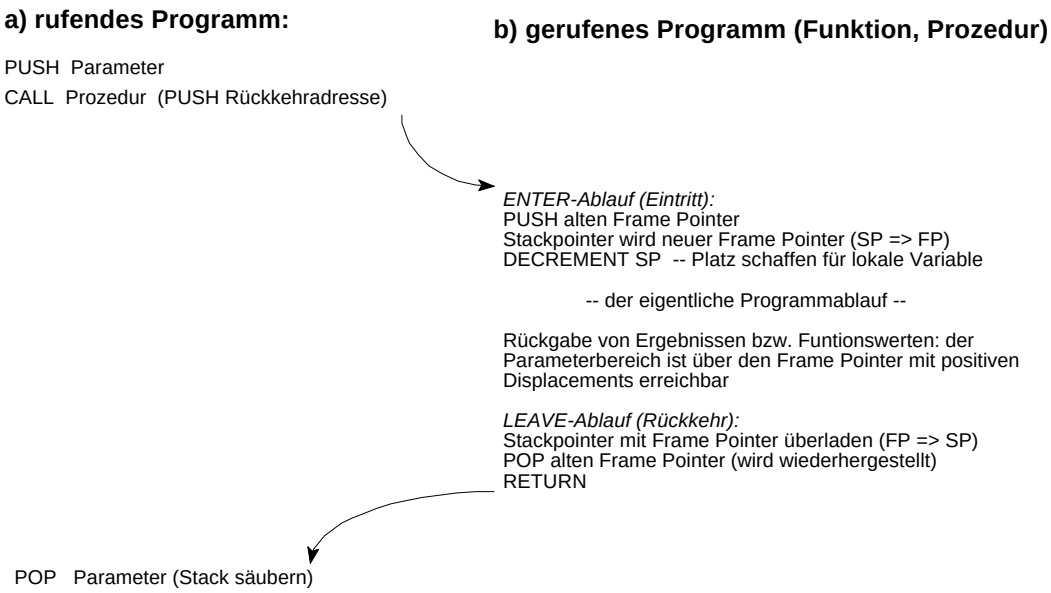


Abbildung 1.13 Unterprogrammaufruf in einer Laufzeitumgebung, die auf Stack Frames beruht

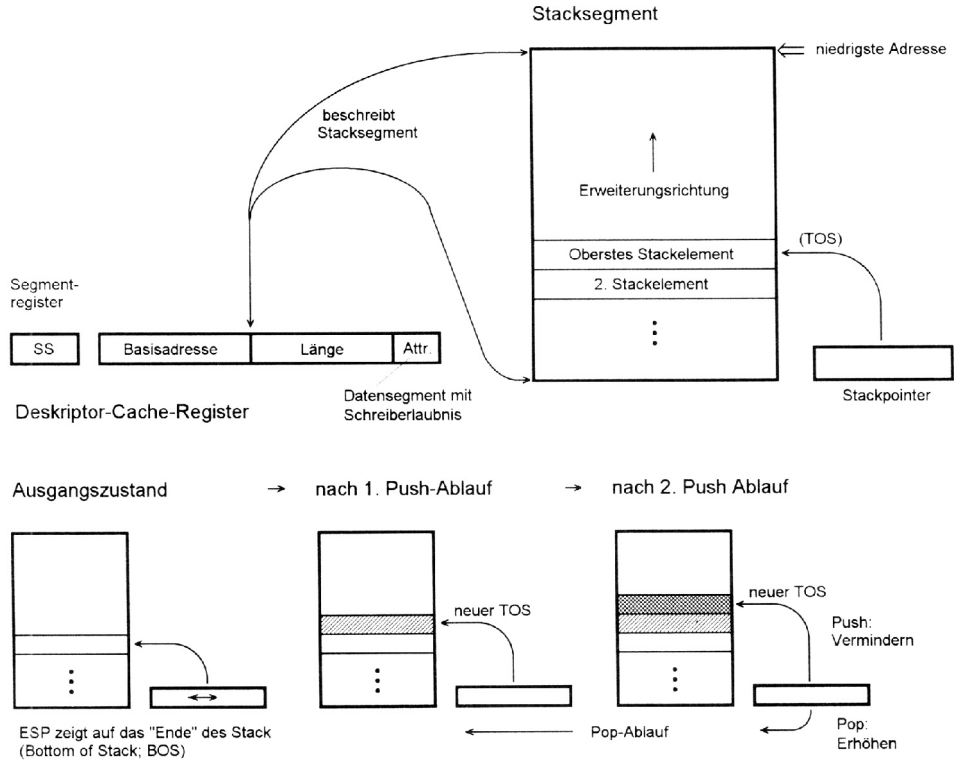


Abbildung 1.14 Aufbau eines IA-32-CPU-Stacks

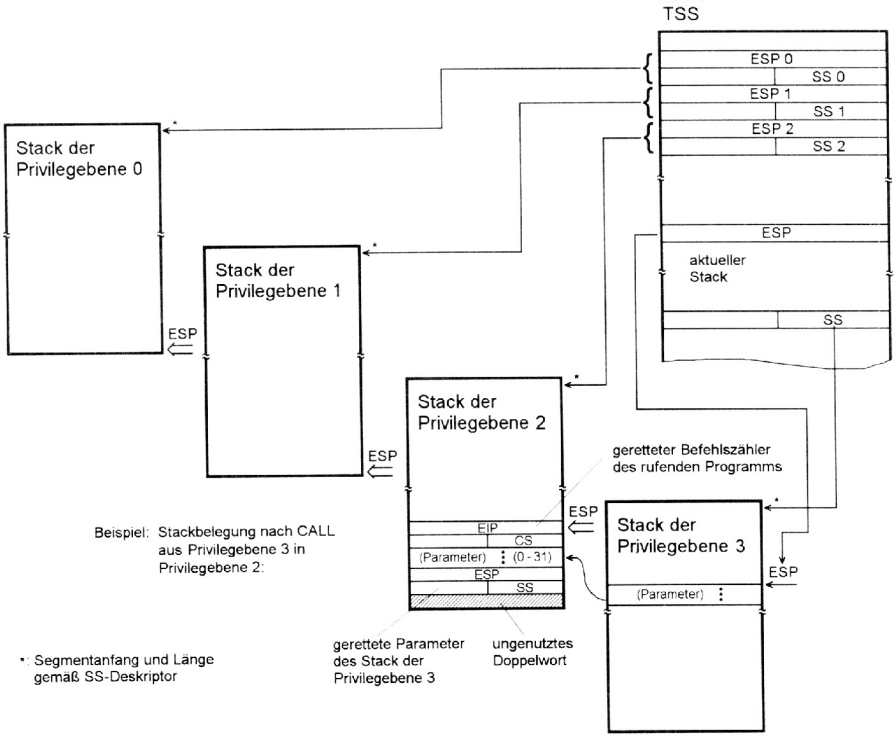


Abbildung 1.15 Stack-Organisation in einer Task

Umschalten zwischen den Stacks

1. Übergang in eine höhere Privilegebene
Das rufende Programm legt erforderlichenfalls Parameter auf *seinen* Stack und ruft ein Programm in einer höheren Privilegebene. Dann führt der Prozessor folgende Schritte aus:

1. es wird geprüft, ob auf dem neuen Stack genügend Platz für die zu rettenden Registerinhalte und ggf. für zu übergebende Parameter ist. Gegebenenfalls wird eine Stackfehler-Ausnahme wirksam. SS und ESP des alten Stacks werden in Form von zwei Doppelworten auf den neuen Stack gelegt.
2. die Parameter werden vom alten auf den neuen Stack kopiert (nur bei Ruf über Call Gate).
3. die Rückkehradresse (CS und EIP des rufenden Programms, wobei EIP auf den Befehl nach dem CALL zeigt) wird auf den neuen Stack gelegt.
4. das gerufene Programm wird mit dem neuen Stack (SS, ESP) gestartet.
5. nach der Umschaltung arbeitet der Prozessor mit dem neuen Stack, dessen aktuelle Parameter (SS, ESP) im Falle einer Taskumschaltung ins TSS gerettet werden. Die Werte ESP0...ESP2, SS0...SS2 bleiben hingegen im TSS erhalten.

Anmerkung:
Die Parameter des "alten" (verlassenen) Stacks sind im neuen gerettet. Deshalb braucht die Privilegebene 3 keinen eigenen Bereich für ihre Stack-Parameter.

Der Fernzeiger des CALL-Befehls bezeichnet das gerufene Programm direkt, wenn der Segmentselektor das Programmsegment unmittelbar auswählt. Die Adreßangabe im Fernzeiger ist dann die Startadresse im ausgewählten Programmsegment. Das automatische Kopieren von Parametern (Schritt 3) ist bei dieser Art des Aufrufs nicht möglich.

Wählt der Segmentselektor im Fernzeiger ein Call Gate aus, so ist die Adreßangabe bedeutungslos. Der Call Gate-Deskriptor enthält einen Zählwert, der die Anzahl der zu kopierenden Parameter angibt (0...31 Doppelworte).

2. Rückkehr in eine niedere Privilegebene
Sind beim Aufruf automatisch Parameter auf den neuen Stack kopiert worden, so muß deren Zahl im Rückkehrbefehl als Direktwert angegeben sein. Aus dem Stack des aktuellen Programms wird die Fortsetzungsadresse (CS, EIP) des Programms eingestellt, zu dem zurückgekehrt werden soll. Eventuell auf dem Stack liegende Parameter werden gemäß der Angabe im Rückkehrbefehl übergangen. Dann wird der Stack mit den geretteten Werten von SS und ESP wieder eingerichtet.

Hinweis:
Der Stack der höheren Privilegebene wird nicht gerettet; nach der Rückkehr befindet er sich wieder im Anfangszustand, der durch die entsprechenden SS- und ESP-Angaben im TSS gegeben ist.

Unterstützung höherer Programmiersprachen durch Stack Frames

Grundlagen
Viele verbreitete Programmiersprachen sind sogenannte blockstrukturierte Sprachen. Ihnen liegt folgendes Konzept der Programmorganisation und des Verarbeitungsablaufs zugrunde:

Verschachtelte Unterprogramme
Es gibt ein jeweils aktives Hauptprogramm, und es gibt Unterprogramme (Prozeduren), die vom Hauptprogramm aus oder auch untereinander aufgerufen werden können. Ein einmal aufgerufenes Unterprogramm kann seinerseits weitere Unterprogrammme rufen, es kann aber nur zu dem Programm zurückkehren, von dem es aufgerufen wurde. Die einzelnen Unterprogramme sind also gleichsam ineinander verschachtelt (Nested Procedures).

Lexikalische Ebenen
Die lexikalische Ebene (Lexical Level) ist eine natürliche Zahl (1, 2...n), die die Verschachtelungstiefe des jeweiligen Unterprogrammaufrufs angibt. Definitionsgemäß hat das Hauptprogramm die lexikalische Ebene 1. Unterprogramme, die unmittelbar vom Hauptprogramm gerufen wurden, haben die lexikalische Ebene 2, solche, die von Unterprogrammen der lexikalischen Ebene 2 gerufen wurden, haben die lexikalische Ebene 3 usw. (Abbildung 1.16). Dieses Schema erlaubt es, das Hauptprogramm und alle Unterprogramme allgemein als *Prozeduren* zu bezeichnen, die in verschiedenen lexikalischen Ebenen arbeiten. Eine Prozedur, die einmal Laufzeit erhalten hat, heißt *aktiv*. Die Prozedur, die gerade den Prozessor belegt, heißt *arbeitend* (Busy). Eine arbeitende Prozedur wird durch Rückkehr inaktiv.

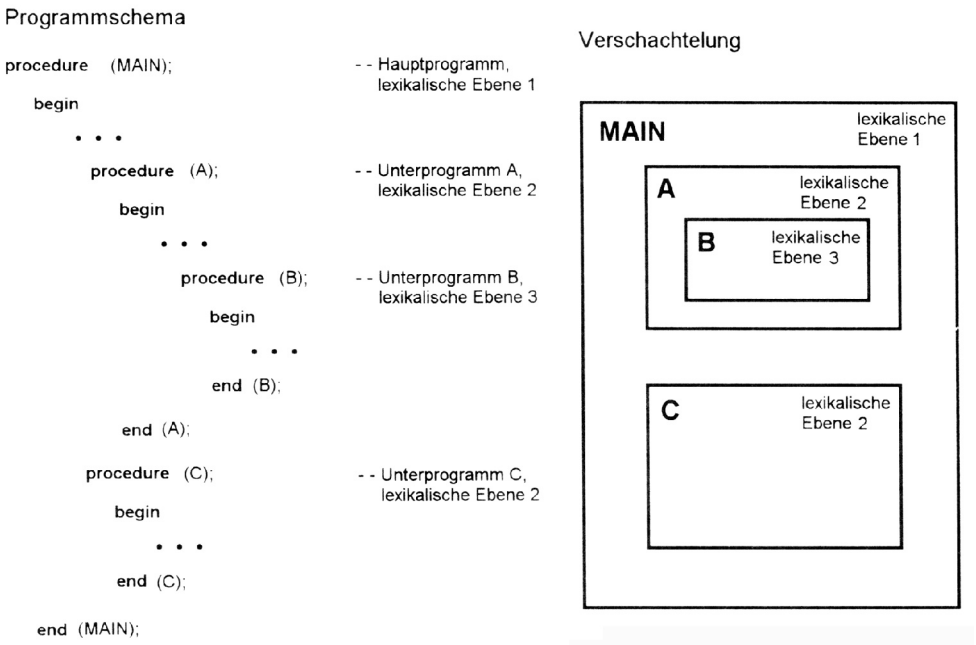


Abbildung 1.16 Verschachtelte Unterprogramme und lexikalische Ebenen

Zugriffsbereiche

Jede Prozedur kann auf bestimmte Variable zugreifen. Das sind zunächst die Variablen, die in ihr deklariert sind (lokale Variable), aber auch Variable anderer *aktiver* Prozeduren (Variable können erst dann konkrete Werte haben, wenn die jeweilige Prozedur aufgerufen wurde, die diese Werte ermittelt). Die Menge aller Variablen, zu denen eine arbeitende Prozedur Zugriff hat, wird als Zugriffsbereich (Scope) bezeichnet.

Stack Frames

Die übliche Implementierung einer solchen Aufruf- und Zugriffsorganisation beruht darauf, daß der Stack-Mechanismus ausgenutzt wird, um (1) der arbeitenden Prozedur Speicherplatz für ihre lokalen Variablen zur Verfügung zu stellen (dynamischer Speicherbereich) und um (2) ihr den Zugriff zu den jeweiligen aktuellen Variablenwerten anderer aktiver Prozeduren zu ermöglichen. Dazu wird der Stack mit Stack Frames belegt (Abbildung 1.17). Ein Stack Frame besteht aus dem dynamischen Bereich und aus einem Zeigerbereich (Display), der die Verbindung zu den Variablen anderer aktiver Prozeduren herstellt. Der Zeigerbereich enthält Zeiger (Frame Pointer) zu den Stack Frames der aktiven Prozeduren. Er bildet die Grundlage des Stack Frame. An ihn schließt sich der dynamische Speicherbereich an, und an diesen wieder der eigentliche Stackbereich (Arbeitsbereich) für die aktuelle Programmausführung.

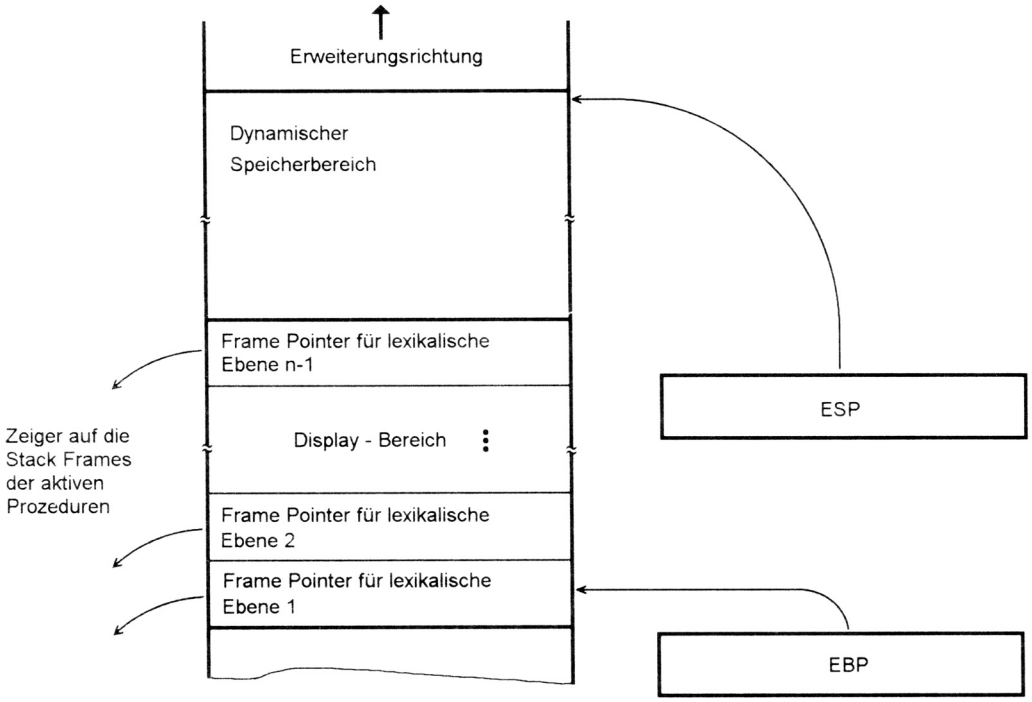


Abbildung 1.17 Aufbau eines Stack Frame

Konventionen der IA-32-Architektur
Das Register EBP dient als Frame bzw. Base Pointer. Sein Inhalt zeigt stets auf den Anfang des aktuellen Stack Frame (d. h. auf das Wort bzw. Doppelwort an der jeweils höchsten Adresse). Alle Inhalte des aktuellen Stack Frame sind somit über negative Offset-Adressen (bezogen auf EBP) erreichbar.

Der Stackpointer ESP bestimmt die aktuelle Größe des dynamischen Bereichs. Er zeigt anfänglich auf dessen Ende (d. h. auf dessen niedrigstes Wort bzw. Doppelwort im Stack).

Der Display-Bereich in einem Stack Frame enthält Zeiger (Frame Pointer) zu den Stack Frames aller aktiven Prozeduren, d. h. aller Prozeduren in niederen lexikalischen Ebenen. Diese Zeiger entsprechen den jeweiligen EBP-Inhalten.

Es sind insgesamt 32 lexikalische Ebenen vorgesehen, die von 0 bis 31 durchnumeriert werden (das ist eine erfahrungsgemäß bei weitem ausreichende Anzahl).

Ein Stack Frame wird mit einem Befehl ENTER auf- und mit einem Befehl LEAVE abgebaut.

ENTER-Befehl
ENTER hat zwei Parameter: (1) die Größe des dynamischen Speichers im neuen Stack Frame sowie (2) die lexikalische Ebene der zu rufenden Prozedur. Abbildung 1.17 zeigt die Wirkung von ENTER für verschiedene lexikalische Ebenen.

Zunächst wird der aktuelle Frame Pointer EBP auf den Stack gelegt. Der jetzt aktuelle Wert des Stackpointers ist der künftige Frame Pointer. EBP wird aber zunächst nicht überladen, sondern als Quelladresse dafür verwendet, den Display-Bereich des alten Stack Frame in den neuen zu kopieren. Der zu kopierende Bereich umfaßt so viele Doppelworte bzw. Worte, wie der zweite Parameter angibt (0...31). Nach dem Kopieren wird der neue Frame Pointer sowohl auf den Stack gelegt als auch in EBP eingestellt. Schließlich wird der Stackpointer soweit vermindert, wie dies der erste Parameter angibt. Auf diese Weise wird der dynamische Speicherbereich im Stack Frame freigelassen. ESP enthält somit die Basisadresse des dynamischen Speicherbereichs.

Über EBP sind ereichbar:

- mit positivem Displacement: die Stack Frames der aktiven Prozeduren (im besonderen die Parameter und die Rückkehradresse der rufenden Prozedur),
- mit negativem Displacement: (1) der dynamische Speicherbereich, (2) die Basisadressen der Stack Frames aller anderen aktiven Prozeduren. Man kann eine solche Adresse beispielsweise in ein allgemeines Register holen und als Basisadresse für den Zugriff auf eine beliebige Variable im betreffenden Stack Frame verwenden.

LEAVE-Befehl
LEAVE kehrt die Wirkung von ENTER um. Der Frame Pointer in EBP wird zum neuen Stackpointer. EBP wird dann aus dem Stack neu geladen (Pop-Ablauf). Damit zeigt der Stackpointer auf das oberste Element im Stack (TOS) der Prozedur, zu der zurückgekehrt werden soll (Abbildung 1.17 veranschaulicht die Wirkung von LEAVE am Beispiel der Rückkehr aus Prozedur A zu Prozedur MAIN). Ein nachfolgender RET- Befehl kann dann die Rückkehr unmittelbar veranlassen (und erforderlichenfalls zuvor Parameter vom Stack entfernen).

Formate der Stack-Belegung
Die Abbildungen 1.19 bis 1.21 zeigen verschiedene Formate von Stackbelegungen, die von Befehlen und vom Interruptsystem erzeugt werden.

Befehle PUSHA, POPA
Mit PUSHA können alle allgemeinen Register sowie der aktuelle Stackpointer (*vor* Ausführung von PUSHA) auf den Stack gelegt werden (Abbildung 1.19). POPA entfernt acht Doppelworte bzw. Worte vom Stack und lädt damit die allgemeinen Register (der (E)SP-Wert im Stack wird dabei übergangen).

Unterprogrammruf (CALL-Befehl)
Bewirkt CALL keine Taskumschaltung, so wird die Rückkehrinformation auf einem Stack abgelegt (Abbildung 1.19). Bei einem Unterprogrammruf im selben Segment (Near CALL) wird nur der Befehlszähler gerettet. Bei einem Fernunterprogrammruf (Far CALL) wird ein Fernzeiger (CS + (E)IP) für die Rückkehr ins rufende Segment gerettet, wenn das gerufene Programm in der gleichen Privilegebene arbeitet. Hat das gerufene Programm hingegen eine höhere Privilegebene, so wird dessen Stack mit den Parametern aus dem TSS eingerichtet (vgl. Abbildung 1.15). Auf diesem wird die Rückkehrinformation abgelegt, die um die aktuelle Stack-Beschreibung (SS + (E)SP) des rufenden Programms erweitert ist. Bei Ruf über ein Call Gate können zusätzlich bis zu 31 Doppelworte bzw. Worte als Parameter vom "alten" auf den neuen Stack kopiert werden.

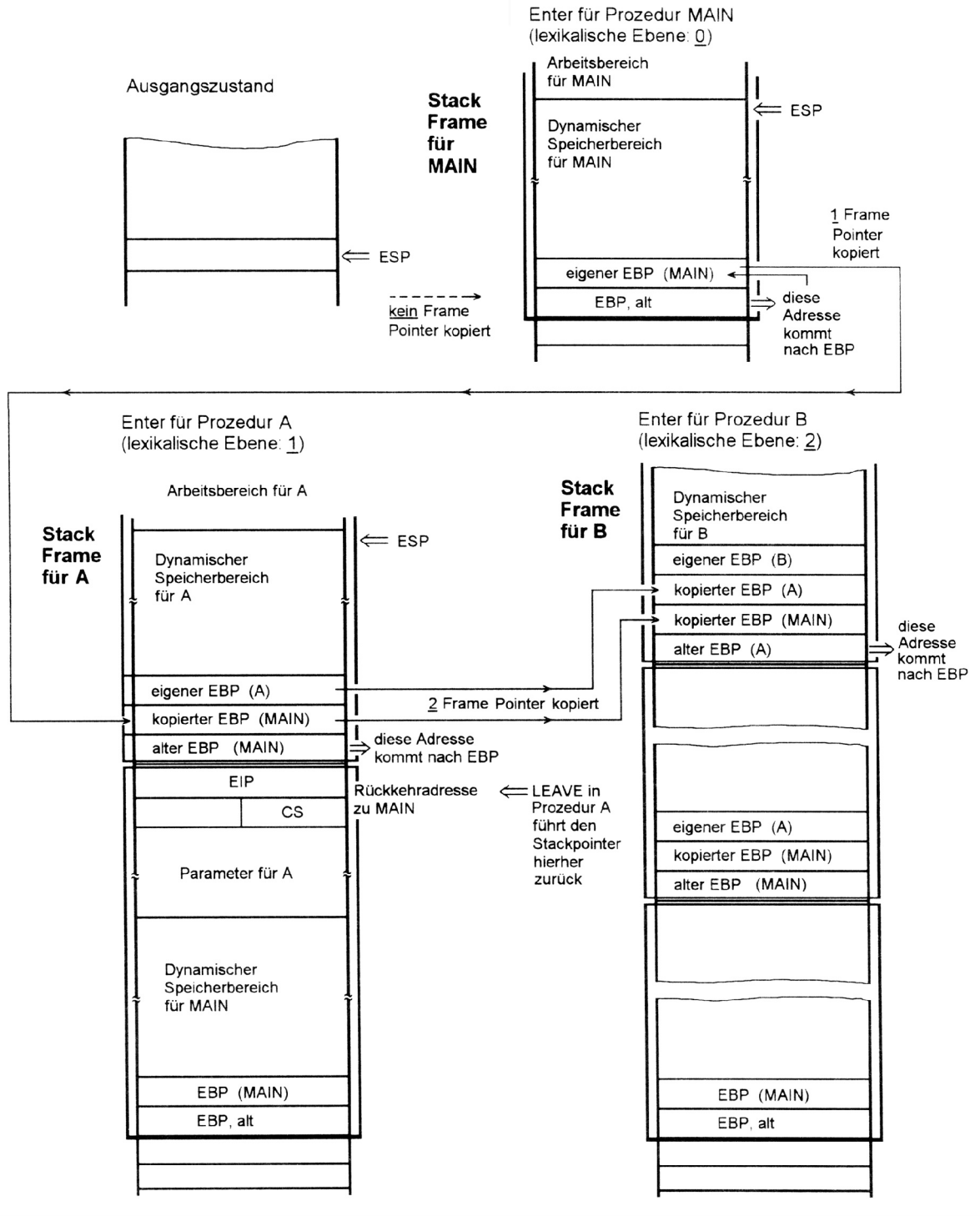


Abbildung 1.18 Zur Wirkungsweise der Befehle ENTER und LEAVE

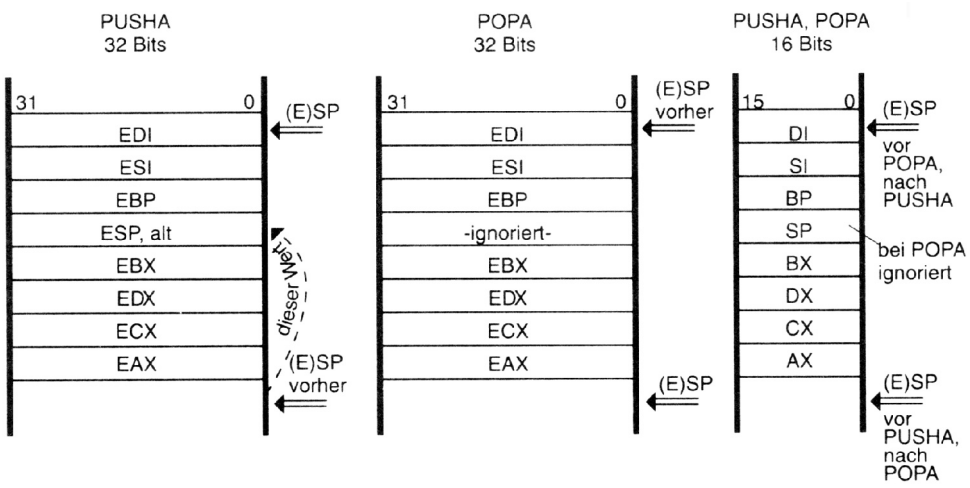


Abbildung 1.19 Zur Wirkung der Befehle PUSHA, POPA

Interrupts

Bewirkt ein Interrupt (externe Unterbrechung über INTR bzw. NMI, INT-Befehle, BOUND-Befehl, Ausnahmen) keine Taskumschaltung, so wird die Rückkehrinformation auf einem Stack abgelegt (Abbildung 1.21). Bei der Interruptbehandlung in der gleichen Privilegebene werden nur Flagregister und Rückkehr-Fernzeiger gerettet. Manche Ausnahmen legen zudem einen Fehlercode auf den Stack. Wird der Interrupt in einer höheren Privilegebene behandelt, wird dessen Stack mit den Parametern aus dem TSS eingerichtet (vgl. Abbildung 1.15). Auf diesem wird die Rückkehrinformation abgelegt, die um die aktuelle Stack-Beschreibung (SS + (E)SP) des rufenden Programms erweitert ist. Wird der Interrupt in einer V86-Task wirksam, so wird eine besondere Stackbelegung erzeugt, die um die Segmentregister SS, DS...GS erweitert ist.

Typische Konventionen des Unterprogrammrufs

Siehe Tabelle 1.5

	Pascal	C
Reihenfolge der Parameterübergabe	von links nach rechts	von rechts nach links
wer stellt bei der Rückkehr die ursprüngliche Stackbelegung wieder her (Stack Cleanup)?	das gerufene Programm	das rufende Programm
Vor- und Nachteile der Stack-Cleanup-Konvention	Cleanup-Ablauf nur einmal vorhanden (im gerufenden Programm), Funktionsaufrufe typischerweise nur mit fester Parameteranzahl	Cleanup-Ablauf in jedem rufenden Programm erforderlich nur einmal vorhanden, erster Parameter (ganz links im Funktionsaufruf) kommt stets auf TOS zu liegen (erleichtert Implementierung von Funktionsaufrufen mit variabler Parameteranzahl)

Tabelle 1.5 Typische Konventionen des Unterprogrammrufs

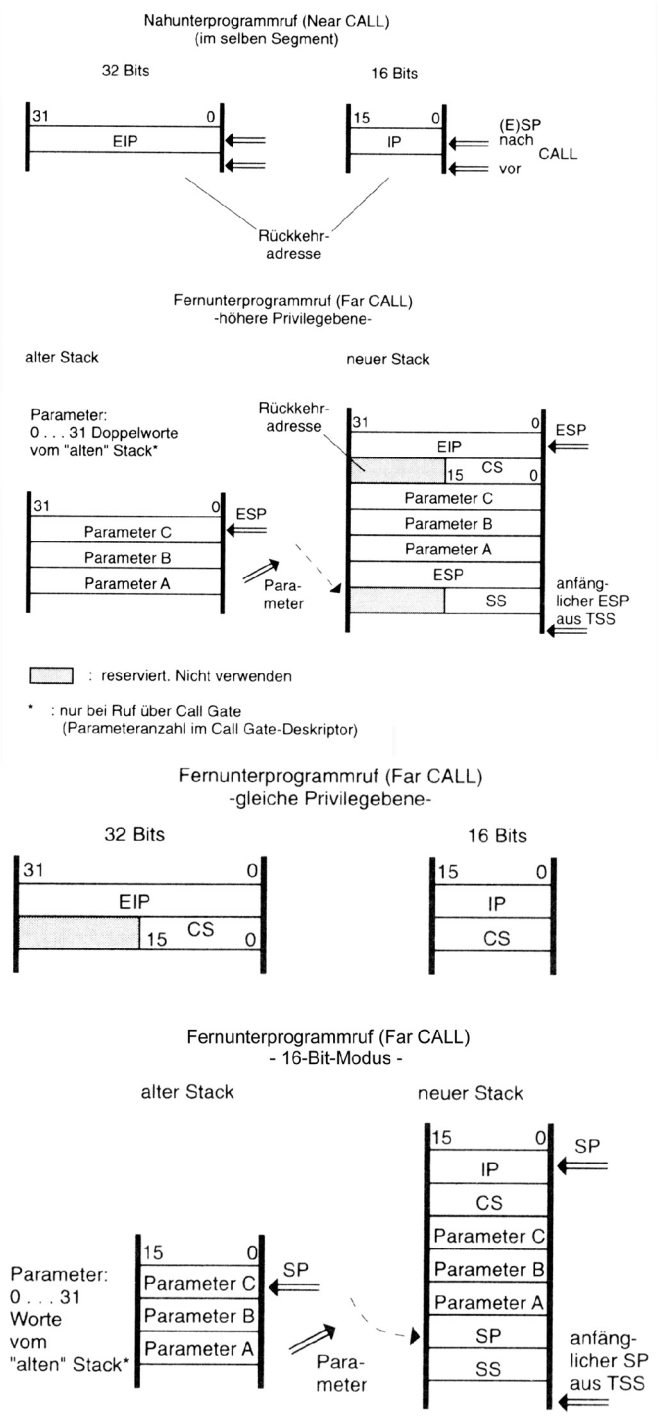


Abbildung 1.20 Stackbelegungen beim Unterprogrammruf (CALL)

Hinweis:
Bei der Rückkehr kann im RET-Befehl ein Direktwert angegeben werden, um den der Stackpointer erhöht wird. So lassen sich Parameterbereiche aus dem Stack entfernen. Die Erhöhung wird erst dann wirksam, nachdem die Rückkehradresse vom Stack entnommen bzw. der "alte" Stack wieder eingestellt wurde. Anwendung: in Programmierkonventionen, die vorsehen, daß die gerufene Prozedur bei der Rückkehr den ursprünglichen Stackzustand wiederherstellt (Beispiel: Pascal).

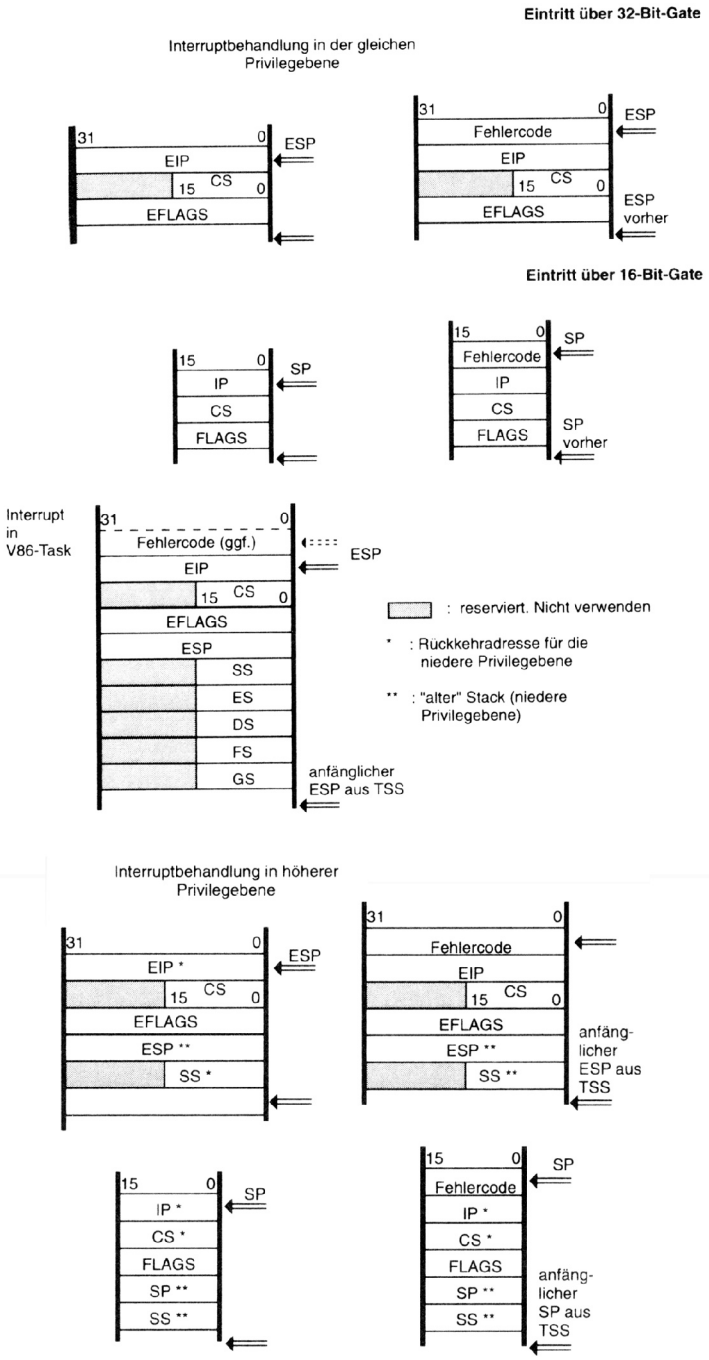


Abbildung 1.21 Stackbelegung bei Interrupt

Zum CALL-Befehl der IA-32-Prozessoren (nähere Einzelheiten)

Varianten:

- CALL rel32/16: Nahunterprogrammruf, relativ,
- CALL r/m32/16: Nahunterprogrammruf, indirekt,
- CALL ptr16:32/16:16: Fernunterprogrammruf, direkt,
- CALL m16:32/16:16: Fernunterprogrammruf, indirekt.

Die Programmabarbeitung wird gemäß den Angaben bezüglich Segment und Adresse fortgesetzt, die sich aus demCALL-Befehl ergeben. Dabei werden Adressenangaben auf den Stack bzw. in das TSS gerettet, so daß es möglich ist, zum rufenden Programm zurückzukehren. Üblicherweise bewirkt ein RET-Befehl im gerufenen Programm die Rückkehr zum nächsten Befehl nach dem CALL.

Nahunterprogrammruf (Near/Intra-Segment Call)
Mit Befehlen "Nahunterprogrammruf" wird das aktuelle Programmsegment nicht verlassen.

Der Befehl "Nahunterprogrammruf, relativ" enthält eine Adreßangabe als Direktwert, der als ganze (vorzeichenbehaftete) Binärzahl zum aktuellen Stand des Befehlszählers addiert wird. (Der Befehlszähler zeigt dabei auf das erste Byte des Folgebefehls.) Der Direktwert ist 16 bzw. 32 Bits lang, je nach Operandenlänge. Der Verzweigungsbereich umfaßt entsprechend -32 768...+ 32 767 bzw. -2^{31}...+ 2^{31} - 1 Bytes, bezogen auf die Anfangsadresse des Folgebefehls.

Der Befehl "Nahunterprogrammruf, indirekt" bewirkt, daß der Befehlszähler mit dem angesprochenen Register- bzw. Speicheroperanden geladen wird (gemäß Operandenlänge). Der Register- bzw. Speicherinhalt muß eine Befehlsadresse im aktuellen Programmsegment sein, die sich auf den Segmentanfang (Adresse 0) bezieht.

Befehlszähler: Bei einer Operandenlänge von 16 Bits werden nur die niedrigstwertigen 16 Bits von EIP zur Befehlsadressierung ausgewertet (IP).

Bei Nahunterprogrammrufen wird nur der Befehlszähler (E)IP auf den Stack gerettet, wobei der Stackpointer (E)SP um 4 bzw. 2 vermindert wird.

Fernunterprogrammruf (Far/Inter-Segment Call)
Ein Befehl "Fernunterprogrammruf" liefert einen Fernzeiger, der, je nach Operandenlänge, 48 bzw. 32 Bits lang ist.

Der Befehl "Fernunterprogrammruf, direkt" enthält den Fernzeiger als Direktwert.

Der Befehl "Fernunterprogrammruf, indirekt" adressiert den Fernzeiger als Speicheroperanden.

Die Wirkung eines Fernunterprogrammrufs wird im einzelnen vom Typ des Deskriptors bestimmt, der von der Selektorangabe des Fernzeigers ausgewählt wird.

Grundsätzlich können folgende Deskriptortypen angesprochen werden:

- Programmsegmentdeskriptoren,
- Call Gate-Deskriptoren,
- Deskriptoren für Taskzustandssegmente (TSS),
- Task Gate-Deskriptoren.

Unterprogrammruf in der aktuellen Task
Das angesprochene Programmsegment darf eine höhere Privilegebene haben als das Segment, das den Verzweigungsbefehl enthält (DPL des "neuen" Deskriptors ≤ CPL).

Durch Ansprechen eines Programmsegmentdeskriptors kann zu einem beliebigen Befehl im betreffenden Programm verzweigt werden. Die Adressenangabe des Fernzeigers enthält die jeweilige Befehlsadresse, bezogen auf den Segmentanfang Adresse 0). Sie wird in den Befehlszähler geladen. Die Selektorangabe wird in das Segmentregister CS übernommen, wobei das zugehörige Deskriptor-Cache-Register entsprechend geladen wird.

Auf den Stack werden sowohl der Befehlszähler als auch der Selektor des rufenden Programmsegments gerettet.

Bei Unterprogrammruf über ein Call Gate wird die Adreßangabe im Fernzeiger ignoriert; die eigentliche Befehlsadresse wird zusammen mit dem Selektor des eigentlichen Programmsegmentes aus dem Call Gate-Deskriptor entnommen. Bei Übergang in eine höhere Privilegebene wird ein neuer Stackbereich zur Verfügung gestellt. Segmentselektor und Stackpointer werden aus dem aktuellen TSS entnommen. Es ist möglich, bis zu 31 Parameter vom alten auf den neuen Stack zu kopieren. Die Parameteranzahl wird im Call Gate-Deskriptor angegeben. Bei einem 32-Bit-Gate ist der einzelne Parameter ein Doppelwort, bei einem 16-Bit-Gate ein Wort.

Taskumschaltung
Wird ein TSS-Deskriptor oder ein Task Gate-Deskriptor angesprochen, so wirkt der Unterprogrammruf als Taskumschaltung mit Taskverschachtelung (Nesting). Das BUSY-Bit im TSS-Deskriptor der (noch) aktuellen Task bleibt gesetzt, diese ist also weiterhin aktiv. Das NT-Bit wird gesetzt, und der Selektor ihres TSS wird in das Rückverweisfeld (Backlink) des neuen TSS eingetragen (will man nach der Rückkehr die Arbeit in dieser Task fortsetzen, muß das gerufenen Unterprogram mit einem IRET-Befehl beendet werden).

Die Adreßangabe im Fernzeiger wird ignoriert; der Selektor für das jeweilige Programmsegment und die Startadresse wird aus dem neuen TSS entnommen.

Hinweis:
Jeder Fernunterprogrammruf aus einem 32-Bit-Programmsegment in ein 16-Bit-Programmsegment sollte in den ersten 64K Bytes des rufenden Programmsegments vorgesehen sein: Da das gerufene Programm die Operandenlänge auf 16 Bits einstellt, wird nur eine 16-Bit-Offsetadresse gerettet.

Fernunterprogrammruf im Realmodus bzw. V86-Modus
In diesen Betriebarten gibt es keine Deskriptoren. Der Fernzeiger bestimmt deshalb unmittelbar das neue Programmsegment und die Startadresse. Auf den Stack werden sowohl der Befehlszähler als auch der Selektor des rufenden Programmsegmentes gerettet.

Stack-Organisation (2): FPU
Die acht Datenregister der Gleitkomma-Verarbeitungseinheit (FPU) werden als Stack betrieben (Abbildung 1.22). Der Stackpointer ist eine 3-Bit-Angabe (TOP) im FPU-Zustandsregister. (Er wird hier, um Verwechslungen mit dem Stackpointer der CPU zu vermeiden, als *Stackzeiger* bezeichnet.)

Stack-Elemente
Das einzelne Stack-Element ist 82 Bits lang (80 Daten- und 2 TAG-Bits).

Ausdehnungsrichtung
Der Stack wächst in Richtung niedere Registeradressen. Der Stackzeiger (TOP) zeigt stets auf das oberste Element im Stack (Top of Stack TOS). Bei einem Push-Ablauf wird TOP um Eins vermindert, dann wird das betreffende Register geladen. Bei einem Pop-Ablauf wird der Wert aus dem von TOP adressierten Register entnommen, dann wird TOP erhöht. Das Vermindern bzw. Erhöhen ist "modulo 8" organisiert (bei einem Push auf Register 0 wird Register 7, bei einem Pop von Register 7 wird Register 0 zum obersten Stackelement).

Belegungsanzeige
Im Gegensatz zum CPU-Stack haben die FPU-Datenregister in ihren TAG-Bits eine Belegungsanzeige. Durch Push wird ein freies Datenregister belegt, durch Pop ein belegtes frei. Ein Push auf ein belegtes bzw. ein Pop von einem freien Datenregister führt zu einer Ausnahmebedingung (Ungültige Operation; Stack-Über- bzw. Unterlauf).

Explizite (stack-relative) Zugriffe
Viele FPU-Befehle können mit relativen Adressen (0...7, bezogen auf TOP) zum Stack zugreifen. Solche Zugriffe müssen ein belegtes Datenregister betreffen, ansonsten wird eine Ausnahmebedingung (Ungültige Operation; Stack-Unterlauf) wirksam.

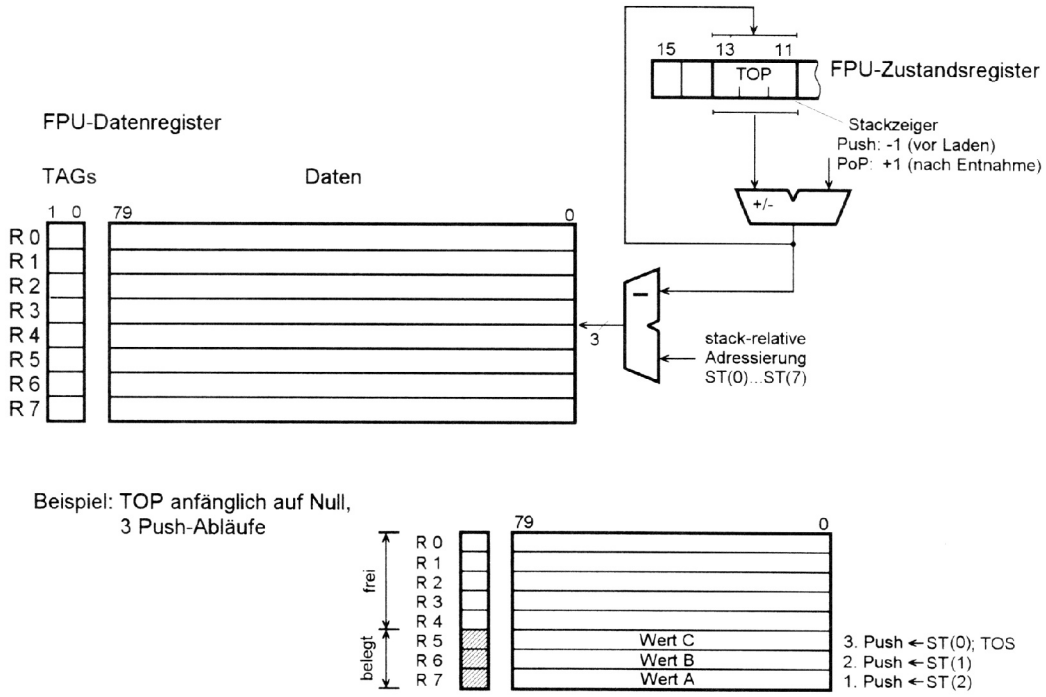


Abbildung 1.22 Der Datenregister-Stack der FPU

Der Registerstack der SPARC-Architektur

Allgemeine Register
Der einzelne Befehl sieht 32 32-Bit-Register (r0...r31). In der Hardware sind aber insgesamt 136 Register vorgesehen. Dem liegt folgendes Programmiermodell zugrunde:

- aus der Sicht des Befehls werden die ersten acht Register r0...r7 fest adressiert (r0 ist dabei kein "richtiges" Register, sondern liefert bei Lesezugriffen den Festwert Null, während Schreibzugriffe keine Wirkung haben). Diese Register heißen "globale" Register.
- die Register r8...r31 werden aus einem "Fenster" (Register Window) entnommen, das jeweils 24 Register umfaßt. Das aktuelle Fenster wird durch eine Angabe im Prozessorzustandsregister PSR ausgewählt (Fensterzeiger, Current Window Pointer CWP).

Dies hat den Zweck, beim Unterprogrammaufruf Parameter übergeben zu können, ohne Registerinhalte transportieren bzw. retten zu müssen (Prinzip des Stack Frame). 8 der 24 Register sind vorgesehen, um Parameter vom rufenden Programm zu übernehmen (In-Register), weitere 8, um lokale Parameter zu halten (lokale Register), und die verbleibenden 8, um Parameter an ein weiteres zu rufendes Unterprogramm zu übergeben (Out-Register).

Das Wirkprinzip im Überblick:

- das aktuelle Programm will ein Unterprogramm rufen. Es hinterlegt dazu die Parameter in seinen Out-Registern.
- beim Unterprogrammruf schaltet man den Fensterzeiger CWP um 16 weiter. Damit werden die Out-Register des rufenden Programms zu In-Registern des gerufenen.
- das Unterprogramm findet so seine Parameter vor, ohne daß dafür irgendwelche Transportabläufe notwendig sind. Des weiteren stehen ihm 16 "eigene" Register zur Verfügung (8 lokale und 8 Out-Register). Es ist also auch nicht erforderlich, Registerinhalte auf einen Stack zu retten, nur um die Register freizubekommen.
- Sinngemäß können Ergebnisse in den In-Registern an das rufende Programm zurückgegeben werden. Dazu schaltet man den Fensterzeiger CWP um 16 zurück, so daß das rufende Programm die Ergebnisse in seinen Out-Registern vorfindet, wiederum ohne daß irgendeine Transportoperation erforderlich ist (Abbildung 1.23).

Fenstermaskenregister WIM
Das Register WIM (Window Invalid Mask) enthält für jedes Register-Fenster ein Bit. Eine Eins kennzeichnet das betreffende Fenster als ungültig. Wird es in die Fensterumschaltung einbezogen, so wird eine Ausnahmebedingung wirksam.

Der Unterprogrammruf

1. Mit CALL-Befehl (Umschaltung des Registerfensters)
Ein Unterprogrammruf mit Umschaltung des Registerfensters besteht aus einem CALL- Befehl mit nachfolgendem SAVE-Befehl. Die Adresse des CALL-Befehls wird in Register r15 gerettet, einem Out-Register des aktuellen Fensters. Der folgende (noch vor der Verzweigung ausgeführte) SAVE-Befehl schaltet auf das nächste Fenster um, so daß die gerettete Adresse vom Unterprogramm in dessen In-Register r31 vorgefunden wird (Abbildung 1.23).

2. Mit JMPL-Befehl
JMPL hat das Format eines Operationsbefehls: 3 Registerangaben oder 2 Registerangaben und ein Displacement. Die Befehlsadresse wird in das im Befehl angegebene Bestimmungsregister gerettet; dann wird aus zwei Registerinhalten oder aus Registerinhalt + Displacement die Verzweigungsadresse bestimmt. Auch hier wird der nachfolgende Befehl noch ausgeführt. Das Registerfenster wird nicht umgeschaltet.

Rückkehr
Es wird grundsätzlich ein JMPL-Befehl verwendet, und zwar mit r0 als Bestimmungsregister (bedeutet: “Befehlsadresse verwerfen” - also: nicht retten). Die andere Registerangabe bezieht sich dann auf das Rettungsregister. Die Displacement-Angabe ist üblicherweise "+8", um aus der geretteten Adresse die erforderliche Rückkehradresse zu errechnen[*)]. Der dem JMPL nachfolgende Befehl wird auch noch ausgeführt. Ist ein Zurückschalten des Register-Fensters notwendig, so ist dies zweckmäßigerweise ein RESTORE-Befehl.

*): die Adreßrettung beim Aufruf betrifft hier nicht die Adresse des Folgebefehls, sondern die des Aufrufbefehls. Da der nächste Befehl nach dem Aufruf noch ausgeführt wird, ist die Adresse also um 2 Befehlslängen = 8 Bytes zu erhöhen.

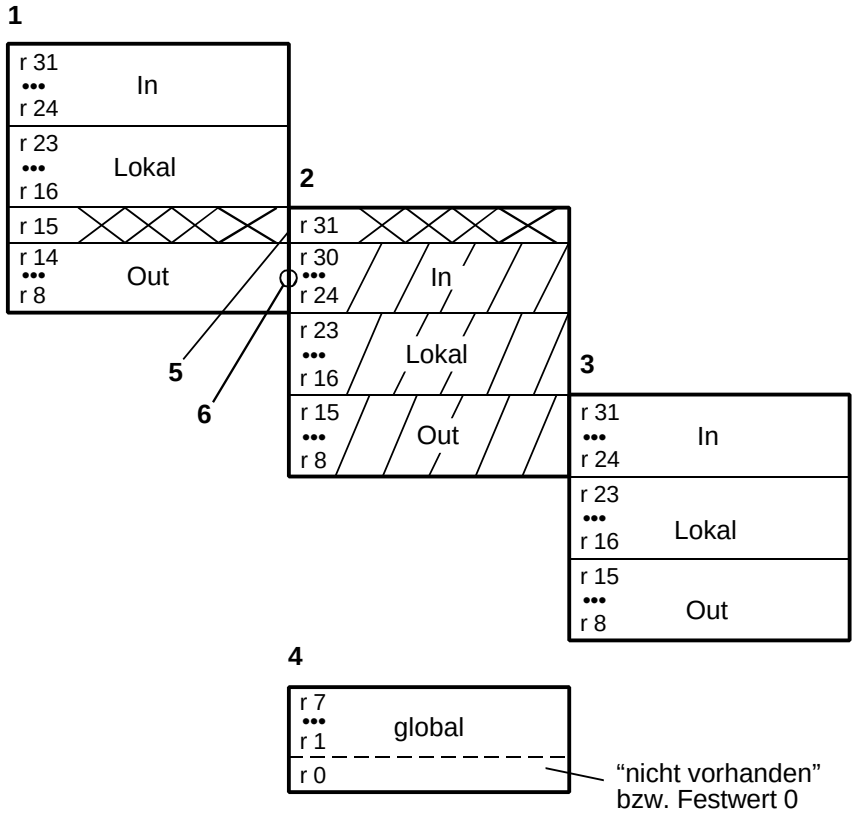


Abbildung 1.23 Registerfenster beim Unterprogrammruf

Erklärung:
1 - Registerfenster vor dem Aufruf; 2 - Registerfenster des gerufenen Unterprogramms; 3 - Registerfenster eines Unterprogramms, das im gerufenden Unterprogramm gerufen wird; 4 - globale Register; 5 - Rettung der Befehlsadresse; 6 - Parameter- und Ergebnisübergabe.

Zum Ablauf:

- das rufende Programm übergibt die Parameter an das Unterprogramm in seinen Registern r14...r8,
- der CALL-Befehl bewirkt die Rettung der Befehlsadresse in Register 15 des Registerfensters 1. Der nachfolgend noch ausgeführte Befehl ist ein SAVE-Befehl, der auf das Registerfenster 2 umschaltet.
- das Unterprogramm findet seine Parameter in den Registern r30...r24 seines Registerfensters 2 vor. Falls es seinerseits ein Unterprogramm aufruft, so stellt es dessen Parameter in seinen Registern r14...r8 bereit usw.
- das Unterprogramm gibt seine Ergebnisse in den Registern r30...r24[*)] an das rufende Programm zurück.

- der Rückkehrbefehl (JMPL) entnimmt die Verzweigungsadresse aus dem Register r31 und erhöht sie um 8. Der nachfolgend noch ausgeführte Befehl ist ein RESTORE-Befehl, der auf das Registerfenster 1 zurückschaltet.
- das rufende Programm wird nun fortgesetzt. Es findet die vom Unterprogramm errechneten Ergebnisse in seinen Registern r14...r8 vor.

*): Register r31 (das ehemalige r15) enthält die gerettete Rückkehradresse.

Der IA-64-Registerstack
Die Architektur sieht 128 allgemeine Register zu 64 Bits vor. Die ersten 32 Register (31...0) werden fest adressiert. Die verbleibenden 96 Register werden im Sinne eines Registerstacks verwaltet. Das Prinzip ähnelt jenem der SPARC-Architektur. Die wesentlichen Unterschiede:

- die Stack Frames haben eine variable, softwareseitig einstellbare Größe,
- Stack Frames, mit denen zur Zeit nicht gearbeitet wird, können automatisch (ohne Eingriff der Software) in den Arbeitsspeicher geschafft und von dort wieder zurückgebracht werden (Register Stack Engine RSE).

Der aktuelle Stack Frame
Er wird mittels des CFM-Registers beschrieben (CFM = Current Frame Marker; Abbildung 1.24).

37 32	31 25	24 18	17 14	13 7	6 0
rrb.pr	**rrb.fr**	**rrb.gr**	**sor**	**sol**	**sof**
Umbenennungsbasis für Prädikatregister (6 Bits)	Umbenennungsbasis für Gleitkommaregister (7 Bits)	Umbenennungsbasis für allgemeine Register (7 Bits)	Größe des rotierenden Teils des Stack Frame (= 8 · sor)	Größe des lokalen Teils des Stack Frame (7 Bits)	Größe des Stack Frame (7 Bits)

Abbildung 1.24 Das CFM-Register

Wir beschränken uns im folgenden auf die allgemeinen Register (Abbildung 1.25).

Registerumbenennung
Das aktuelle Programm sieht das erste Register seines Stack Frame stets unter Registeradresse 32. Das Feld rrb.gr im CFM gibt an, welches physische Register dieser Adresse zugeordnet wird.

Der vorhergehende Stack Frame
Beim Ruf eines Unterprogramms wird der CFM des rufenden Programms in das PFM-Feld des PFS-Registers gerettet. PFM = Previous Frame Marker, PFS = Previous Function State Register (ein IA-64-Steuerregister).

Festlegung eines Stack Frame
Mittels *alloc*-Befehl (Allocate Stack Frame). *alloc* hat 5 Parameter:

- ri: Adresse eines allgemeinen Registers. Die aktuelle PFS-Belegung wird in dieses Register transportiert (hiermit wird u. a. die Beschreibung des vorhergehenden Stack Frames gerettet). Die Registeradresse bezieht sich auf den neu definierten Stack Frame.
- i: Anzahl der Eingangsregister (Input),
- l: Anzahl der lokalen Register (Local),
- o: Anzahl der Ausgangsregister (Output),
- r: Anzahl der rotierenden Register (Rotating).

Größe des Stack Frames: sof = i + l + o

Größe des lokalen Teils: sol = i + l.

Im Gegensatz zu SPARC wird an sich nicht zwischen Eingangs- und lokalen Registern unterschieden. Die Trennung in zwei Angaben (i, l) dient nur dazu, dem Assembler Information über die Anzahl der lokalen Register zu übergeben (zu eventuellen Optimierungszwecken).

Unterprogrammruf (br.call)):*

- die Angaben zum bisherigen Stack Frame werden gerettet: CFM => PFM,
- es wird ein neuer Stack Frame erzeugt, der die Ausgangsregister des bisherigen Stack Frame umfaßt: $sof_{neu} := sof_{alt} - sol_{alt}$,
- die Umbenennungsbasis des aktuellen (neuen) Stack Frames wird auf das erste Ausgangsregister des alten Stack Frames gestellt (dieses Ausgangsregister wird zu Register 32),
- die Größe des lokalen Teils wird auf Null gestellt (sol := 0),
- die Rückkehradresse wird in das im Befehl angegebene Verzweigungsregister (Branch Register) gerettet.

Rückkehr (br.ret)):*

- der vorhergehende Stack Frame wird wieder hergestellt (PFM => CFM),
- die Verzweigung wird ausgeführt (typischerweise mit der im angegebenen Verzweigungsregister geretteten Rückkehradresse).

*): unter Vernachlässigung weiterer Einzelheiten.

Der alloc-*Befehl*
Das aufgerufene Unterprogram führt typischerweise einen *alloc*-Befehl aus, um seinen Stack Frame entsprechend zu vergrößern. Die Registerumbenennung wird dadurch nicht beeinflußt. Sind nicht genügend Register frei, so werden vorhandene Stack Frames in den Arbeitsspeicher ausgelagert.

Geschachtelte Unterprogrammrufe
Was hierbei nicht automatisch gerettet wird: die Angaben zum jeweils vorhergehenden Stack Frame (PFM). Unterprogramme, die weitere Unterprogramme rufen, müssen den PFM entsprechend retten und wiederherstellen.

Hinweis:
Der *alloc*-Befehl unterstützt das Retten des PFM.

Weitere Befehle in Zusammenhang mit dem Aus- und Einlagern von Stack Frames:

- *flushrs* lagert alle vorhergehenden Stack Frames in den Arbeitsspeicher aus,
- *cover* erzeugt einen neuen Stack Frame der Größe Null (sof = sol = 0). Dabei wird der vorhergehende Stack Frame automatisch in den Arbeitsspeicher gerettet..
- *loadrs* tranportiert eine angegebene Anzahl an Bytes aus dem Rettungsbereich im Arbeitsspeicher in den Registerstack.

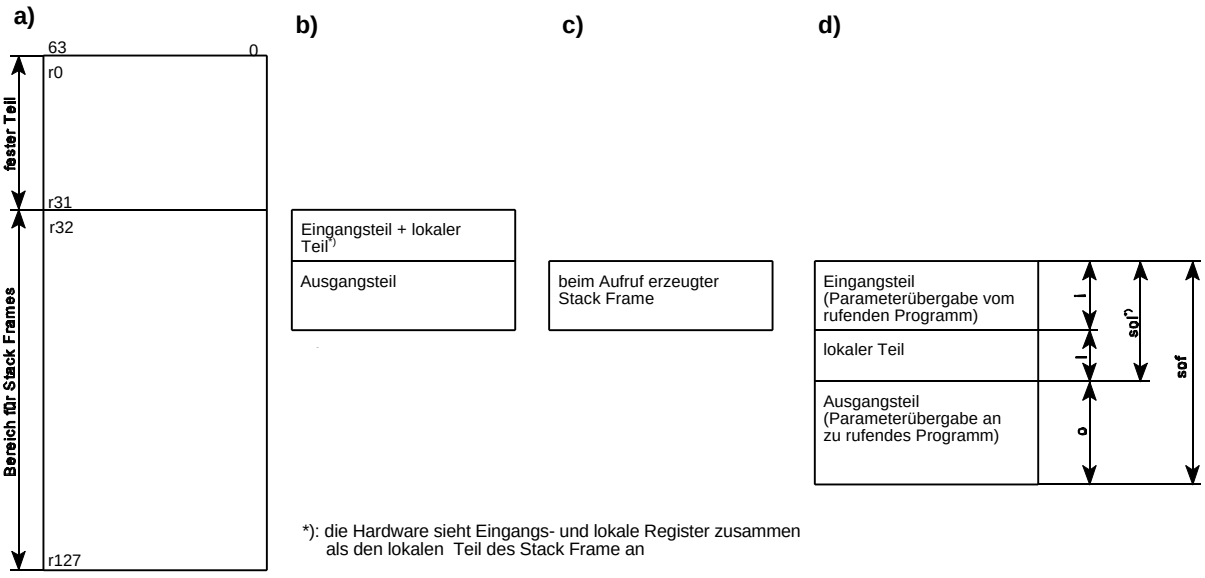


Abbildung 1.25 Der IA-64-Registerstack im Überblick

Erklärung:

a) die 128 allgemeinen Register,
b) ein Stack Frame,
c) ein Unterprogrammruf (Befehl br.call oder brl.call) hat einen neuen Stack Frame erzeugt,
d) mittels *alloc*-Befehl hat das gerufende Unterprogramm seinen Stack Frame entsprechend erweitert.