1 Laufzeitorganisationen

1.1 Einführung und Überblick

Die Laufzeitorganisation beschäftigt sich mit der Abbildung von abstrakten Strukturen einer Hochsprache (Variablen, Prozeduren, Objekte) auf die konkreten Ressourcen der Zielmaschine (Register, Speicher, Instruktionen).

Es existiert eine **Semantic Gap** zwischen den komplexen Konstrukten der Hochsprache (Arrays, Objekte, Methoden) und den primitiven Möglichkeiten der Hardware.

Aufgaben der Laufzeitorganisation

- Datendarstellung (Primitive Typen, Records, Arrays).
- Auswertung von Ausdrücken (Stack vs. Register).
- Speicherverwaltung (Global, Lokal/Stack, Heap).
- Routinen und Aufrufkonventionen (Parameterübergabe).

1.2 Triangle Abstract Machine (TAM)

Die TAM ist eine abstrakte Zielmaschine für Lehrzwecke. Sie basiert auf einer **Harvard-Architektur**, was bedeutet, dass Befehls- und Datenspeicher getrennt sind.

1.2.1 Speicherbereiche und Register

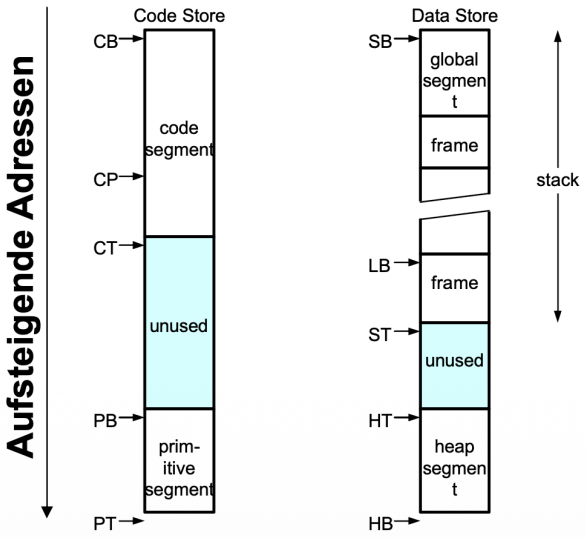
Die TAM nutzt verschiedene Register zur Adressierung der Speichersegmente.

Instruktionsspeicher (Code Store) Der Code-Speicher enthält das ausführbare Programm.

- **CB** (Code Base): Startadresse des Code-Segments (konstant).
- **CT** (Code Top): Endadresse des Code-Segments (konstant).
- **CP** (Code Pointer): Aktueller Befehlszähler (Instruction Pointer), zeigt auf den nächsten auszuführenden Befehl.
- **PB** (Primitive Base): Startadresse der Intrinsics (eingebaute Funktionen).
- **PT** (Primitive Top): Endadresse der Intrinsics.

Datenspeicher (Data Store) Der Datenspeicher ist in Stack und Heap unterteilt. In der TAM wachsen diese Bereiche aufeinander zu (siehe Speicherverwaltung).

- **SB** (Stack Base): Boden des Stacks (Start der globalen Variablen).
- **ST** (Stack Top): Aktuelles oberes Ende des Stacks.
- **HB** (Heap Base): Startadresse des Heaps (oberes Ende des Speichers).
- **HT** (Heap Top): Aktuelle Grenze des belegten Heaps.



- **LB** (Local Base): Zeiger auf den aktuellen *Stack Frame* (Beginn der lokalen Variablen der aktuellen Prozedur).

1.2.2 Instruktionen

TAM-Instruktionen sind 32-bit breit und haben folgendes Format:

$$\text{Instruktion} = \underbrace{\text{op (4 Bit)}}_{\text{Opcode}} \mid \underbrace{\text{r (4 Bit)}}_{\text{Register}} \mid \underbrace{\text{n (8 Bit)}}_{\text{Größe}} \mid \underbrace{\text{d (16 Bit)}}_{\text{Displacement}}$$

Beispiel: `LOAD (1) 3[ST]` lädt ein Wort von der Adresse $ST + 3$.

1.3 Datendarstellung (Repräsentation)

Daten müssen im Speicher so abgelegt werden, dass sie effizient zugreifbar sind.

1.3.1 Prinzipien

1. **Unverwechselbarkeit**: Unterschiedliche Werte sollten unterschiedliche Bitmuster haben.
2. **Einzigartigkeit**: Ein Wert wird immer gleich dargestellt.
3. **Konstante Größe**: Alle Werte eines Typs belegen gleich viel Platz.

Invariante der Datengröße

Es muss gelten: $size[T] \geq \log_2(\#[T])$, wobei $\#[T]$ die Anzahl der unterschiedlichen Elemente in T ist.

1.3.2 Primitive Typen

- **Boolean**: 1 Wort (16b in TAM). Werte: 00..00 (false), 00..01 (true). *Hinweis: In C/x86 oft nur 8 Bit.*
- **Char**: 1 Wort (16b), Unicode/ASCII.
- **Integer**: 1 Wort (16b), Zweierkomplement.

1.3.3 Zusammengesetzte Typen

Records (Verbundtypen) Die Felder eines Records werden im Speicher nacheinander (sequenziell) abgelegt.

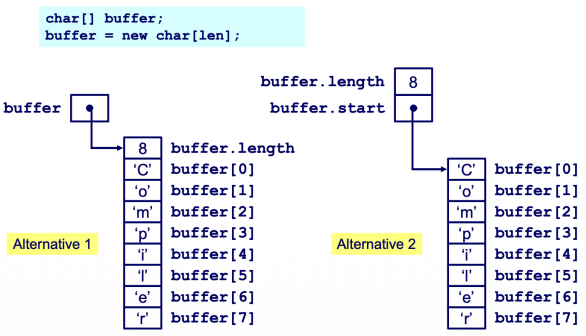
- **Adressierung**: Adresse des Records + Offset des Feldes.
- **Padding**: Viele Prozessoren verlangen eine Ausrichtung (Alignment) auf Wortgrenzen (z.B. 32-bit), was zu ungenutzten Lücken (Padding) führen kann. TAM adressiert wortweise, daher weniger Padding-Probleme, aber Platzverschwendung bei Booleans.

Arrays (Felder)

- **Statische Arrays**: Größe zur Compile-Zeit bekannt. Elemente liegen direkt hintereinander.

$$address[me[i]] = address[me] + i \times size[Element]$$

- **Dynamische Arrays**: Größe erst zur Laufzeit bekannt.
- **Repräsentation**: Indirekt über einen **Deskriptor** (Dope Vector). Dieser enthält einen Zeiger auf die Daten (im Heap) und die aktuelle Größe.



Variante Records (Disjoint Unions) Ähnlich wie Records, aber die Komponenten überlagern sich im Speicher (Union in C). Ein *Type Tag* entscheidet, welche Interpretation gerade gültig ist. Die Größe richtet sich nach der größten Komponente.

1.4 Auswertung von Ausdrücken

Wie werden mathematische Ausdrücke wie $a \times a + 2 \times a \times b$ berechnet?

1.4.1 Stack-Maschine (z.B. TAM)

Arbeitet nach dem **Post-Fix-Prinzip**. Operanden werden auf den Stack gelegt (`LOAD`), Operationen (`ADD`, `MUL`) nehmen die obersten Elemente, verrechnen sie und legen das Ergebnis zurück.

- *Vorteil*: Einfache Code-Generierung, keine Registerverwaltung nötig.
- *Nachteil*: Viele Speicherzugriffe, langsamer als Registermaschinen.

1.4.2 Register-Maschine

Berechnungen finden in schnellen CPU-Registern statt.

- *Vorteil*: Sehr schnell.
- *Nachteil*: Begrenzte Anzahl Register erfordert komplexe Zuteilungsstrategien (Register Allocation), wenn Zwischenergebnisse die Anzahl der Register übersteigen ("Spilling").

1.5 Speicherverwaltung (Stack)

Die Verwaltung des Speichers für Variablen hängt von ihrer Lebensdauer ab.

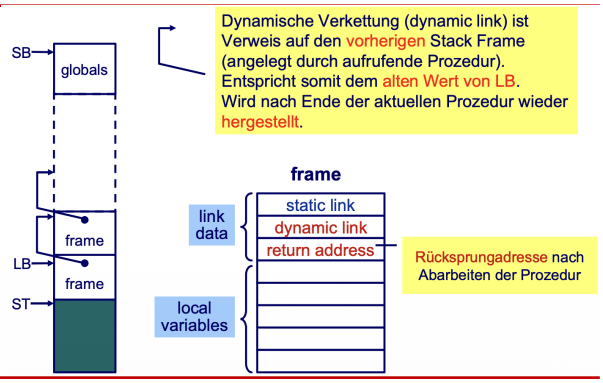
1.5.1 Arten von Variablen

1. **Globale Variablen**: Existieren über die gesamte Laufzeit. Adresse ist fest relativ zu SB.
2. **Lokale Variablen**: Existieren nur, solange der Block (Prozedur/Funktion) aktiv ist. Verwaltung über den **Stack**.
3. **Heap-Variablen**: Lebensdauer unabhängig vom Scope (siehe Abschnitt Heap).

1.5.2 Stack Frame (Activation Record)

Jeder Prozeduraufruf erzeugt einen neuen Stack Frame. Dieser enthält:

- **Parameter**: Vom Aufrufer abgelegt.
- **Verwaltungsdaten (Link Data)**: Static Link, Dynamic Link, Rücksprungadresse.
- **Lokale Variablen**: Innerhalb der Prozedur angelegt.
- **Zwischenergebnisse**: Für die Expression-Evaluation.



1.5.3 Verkettung (Linking)

Um auf Variablen zuzugreifen, werden zwei Arten von Links im Stack Frame gespeichert:

Dynamic Link (Dynamische Verkettung)

Zeigt auf den Stack Frame des **Aufrufers** (Caller). Entspricht dem alten Wert des LB-Registers. Dient dazu, beim Rücksprung (Return) den Stack-Kontext des Aufrufers wiederherzustellen.

Static Link (Statische Verkettung)

Zeigt auf den Stack Frame der Prozedur, die die aktuelle Prozedur im Quelltext **umschließt** (textuelle/lexikalische Hierarchie). Dient dem Zugriff auf **nicht-lokale Variablen** in verschachtelten Prozeduren.

Bestimmung des Static Link (SL) Wenn Prozedur P (auf Ebene L_P) eine Prozedur Q (auf Ebene L_Q) aufruft:

- **Aufruf einer globalen Prozedur** ($L_Q = 0$): $SL = SB$.
- **Aufruf einer eingebetteten Prozedur** ($L_Q > 0$):
  - *Q ist direkt in P definiert* ($L_Q = L_P + 1$): $SL = LB$ (aktueller Frame von P).
  - *Q ist auf gleicher Ebene oder weiter außen* ($L_Q \leq L_P$): Man muss der statischen Kette von P folgen ($k = L_P - L_Q + 1$ Schritte), um den korrekten Kontext zu finden.

Display-Register: Eine Alternative zur statischen Verkettung, bei der ein Array von Zeigern (Display) gepflegt wird, das für jede Schachtelungstiefe direkt auf den aktuellen gültigen Frame zeigt. Schnellerer Zugriff, aber aufwendigerer Prozeduraufruf.

1.6 Routinen und Protokolle

Das Zusammenspiel von Aufrufer (Caller) und Aufgerufenem (Callee) wird durch ein Protokoll (Calling Convention) geregelt.

1.6.1 Ablauf eines Aufrufs (TAM)

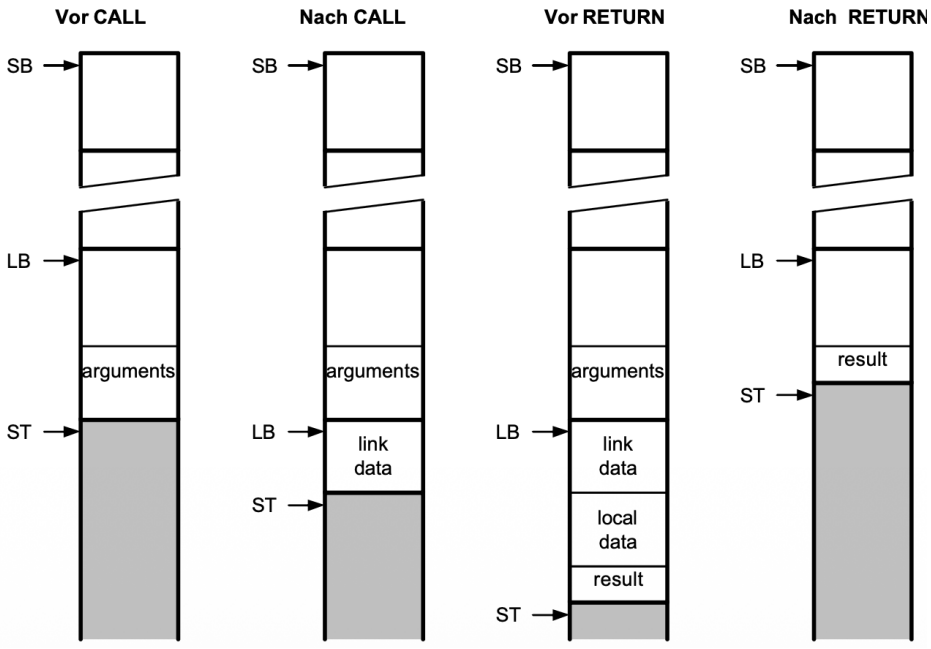


Figure 1: *Vor/Nach Call/Return im Stack*

1. **Vor CALL (Caller)**:
   - Argumente (Parameter) werden auf den Stack gepusht (in TAM: in umgekehrter Reihenfolge, damit das erste Argument oben liegt oder direkt über LB adressierbar ist).
2. **CALL (Instruktion)**:
   - Sichert *Static Link* (wird berechnet/übergeben).

- Sichert *Dynamic Link* (aktueller LB).
- Sichert *Return Address* (PC + 1).
- Setzt neuen *LB* auf den Beginn des neuen Frames.
- Sprung zur Code-Adresse der Routine.

3. **In der Routine**:
   - Reserviert Platz für lokale Variablen (Inkrementiert *ST*).
4. **RETURN (Callee)**:
   - Entfernt lokalen Speicher und Verwaltungsdaten.
   - Entfernt Argumente vom Stack.
   - Legt Rückgabewert (Result) auf den Stack.
   - Stellt alten *LB* und *ST* wieder her.
   - Springt zurück.

1.6.2 Parameterübergabe

Parameter werden relativ zu *LB* mit **negativen Offsets** adressiert (da sie vor dem Frame-Start auf den Stack gelegt wurden). Lokale Variablen haben positive Offsets.

- **Call-by-Value**: Der Wert der Variable wird kopiert. Änderungen in der Prozedur haben keinen Effekt auf den Aufrufer.
- **Call-by-Reference** (`var`): Die *Adresse* der Variable wird übergeben. Die Prozedur arbeitet via Indirektion direkt auf dem Speicherplatz des Aufrufers. Änderungen sind global sichtbar.

1.6.3 Funktionen als Parameter (Closures)

Wenn eine Funktion *F* als Parameter übergeben wird, reicht die Startadresse nicht aus, da *F* Zugriff auf ihren statischen Kontext benötigt.

- Lösung: **Closure** (Funktionsabschluss).
- Repräsentation: Paar aus `(Code-Adresse, Static Link)`.
- Aufruf: `CALLI` (Call Indirect) nutzt dieses Paar.

1.7 Heap-Speicherverwaltung

Der Heap dient für Daten, deren Lebensdauer nicht an den Block-Scope gebunden ist (z.B. verkettete Listen, Bäume).

1.7.1 Organisation

In der TAM (und vielen Systemen) wachsen Stack und Heap aufeinander zu. Wenn sie sich treffen → *Out of Memory*.

- **Allokation**: Suchen eines freien Blocks geeigneter Größe.
- **Deallokation**: Freigabe von Speicher.

1.7.2 Probleme und Strategien

- **Fragmentierung**: Durch unregelmäßiges Anlegen und Freigeben entstehen Lücken ("Löcher"), die zu klein für neue Objekte sind, obwohl in Summe genug Speicher frei wäre.
- **Freispeicherliste (Free List)**: Liste (z.B. `HF` in TAM) verkettet alle freien Blöcke.
- **Kompaktierung**: Verschieben von belegten Blöcken, um Lücken zu schließen. Erfordert Aktualisierung aller Zeiger (schwierig!) oder Nutzung von *Handles* (Zeiger auf Zeiger).

1.7.3 Garbage Collection (Automatische Speicherbereinigung)

Verfahren, um nicht mehr erreichbaren Speicher automatisch freizugeben (z.B. in Java).

Mark-and-Sweep Algorithmus

1. **Mark (Markieren)**: Gehe von allen Wurzeln (Register, Stack-Variablen) aus und verfolge alle Zeiger. Markiere jedes erreichte Objekt im Heap als "lebendig".
2. **Sweep (Fegen)**: Durchlaufe den gesamten Heap. Alle nicht markierten Objekte sind "Müll" und werden zur Freispeicherliste hinzugefügt. Markierungen werden für den nächsten Lauf zurückgesetzt.