Organisatorisches

Die Großübung findet zweimal mit gleichen Inhalt statt:
Montag 16-18 und Mittwoch 14-16 jeweils im MA001.

Betreute Rechnerzeit:
Donnerstag 10-18 und Freitag 10-16 jeweils FR2516 **Code: 5934**
Zusätzliche Rechnerzeit: Mi 12-14 (geändert!)
Tutorenraum: FR2068

Weitere und stets aktuelle Informationen unter:
`http://pdv.cs.tu-berlin.de/Info4-SS2001/`

Abgabe der Hausaufgaben: ITM und Technomathematiker möglichst in eigenen Gruppen (max. 4 Personen pro Gruppe)!

Noch kein Bereich (login) im CS-Netz: beim Rechnerbetrieb melden (Fr5091, Mo-Fr 9:30-11:00, Mo-Do 14:00-15:00

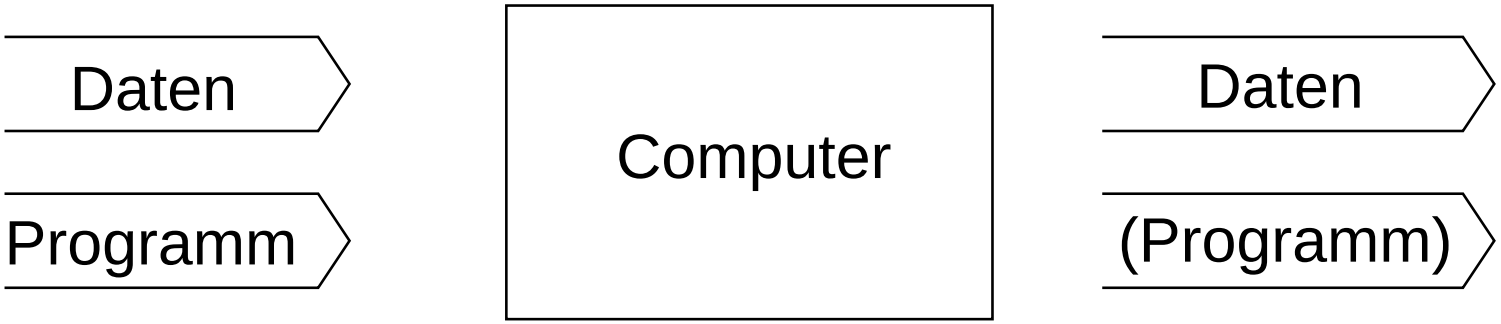
Skriptverkauf sowie sonstige Fragen nur zu den betreuten Rechnerzeiten!

1. Großübung: Assemblerprogrammierung

1. Algorithmen und Programme
2. Assemblersprache
3. Assemblerprogrammierung
4. Beispiel

1.1 Algorithmen und Programme

Programm: Algorithmus in einer vom Computer interpretierbaren Darstellung



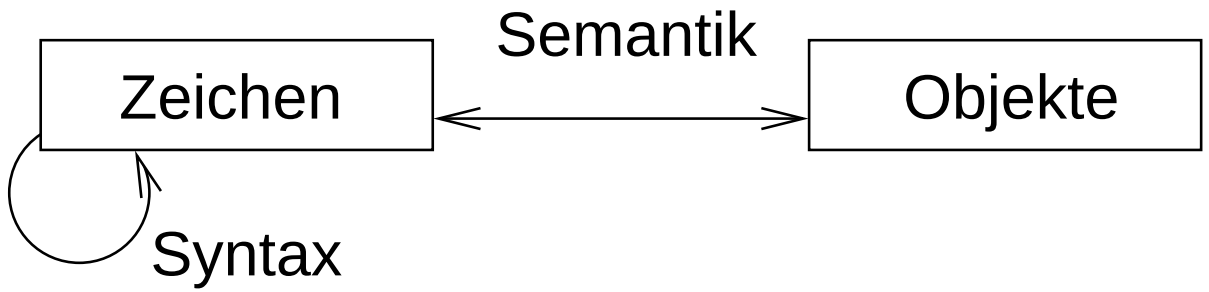
Programmiersprache: Notationssystem für die Darstellung von Programmen.

Programmiersprachen

unterscheiden sich in

Syntax: bestimmt, welche Zeichenfolge zu einer Sprache gehört

Semantik: Bedeutung, Verhältnis der Zeichen zu Objekten



Programmiersprachen (Forts.)

In der Informatik wird unterschieden zwischen

Höheren Programmiersprachen: haben eine der Problemstellung angepaßte Semantik (Java, Modula-2, C++, ...)

Assemblersprachen: haben eine der interpretierenden Maschine angepaßte Semantik

Maschinensprachen: vom Computer direkt interpretierbare Notation

1.2 Assemblersprache

Merkmale von Assemblersprachen sind:

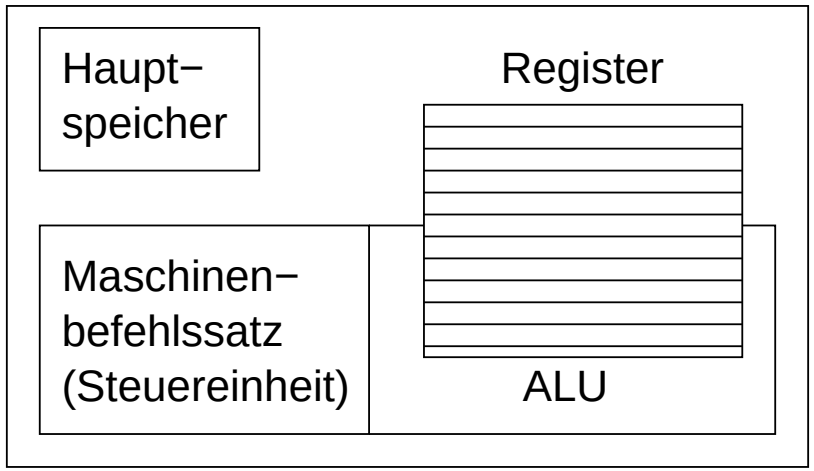
- Jede Anweisung und jede Operation entspricht genau einem Maschinenbefehl.
- Es werden Symbolische Adressen und Namen verwendet.
- Programme können nicht direkt ausgeführt werden, sondern müssen zuvor übersetzt (⇒assembler, Transformation in Objektkode) und gebunden (⇒linker) werden.
- Sie sind ein direktes Abbild der Maschinensprache und erlauben eine vollständigere Kontrolle der Hardware, als es höhere Programmiersprachen tun.
- Programme sind auf den jeweiligen Rechner zugeschnitten und **nicht portabel**!

⇒ (Fast) Jeder Rechner hat nur eine Maschinensprache, aber oft mehrere Assemblersprachen.

1.3 Assemblerprogrammierung

- Nötig sind:
  1. Kenntnis einer Assemblersprache der Maschine
  2. Kenntnis der Architektur der Maschine
- Architektur der Maschine (Hardware-System-Architecture) wird durch das Hardware-Programmiermodell beschrieben
  (auch ISA = Instruktions-Satz-Architektur).

Hardware-Programmier-Modell



Beipsiel: ein SISD-Rechner

Das Hardware-Programmier-Modell beinhaltet:

1. Registermodell
2. Maschinen-Datentypen
3. Maschinenbefehlssatz
4. Adressierungsarten
5. Ein-/Ausgabeorganisation

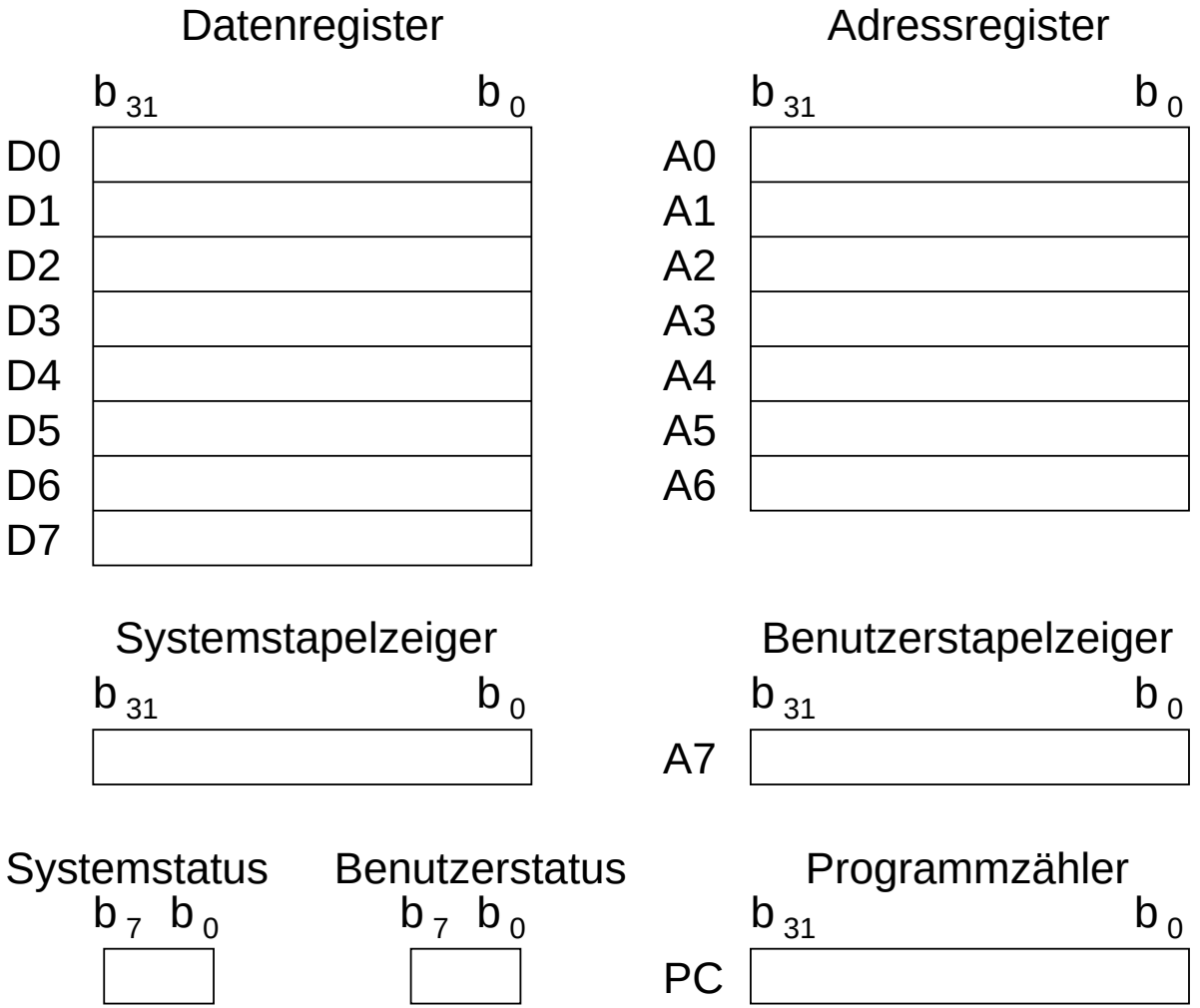
Hardware-Programmier-Modell des MC68000

Familie von Prozessoren mit aufwärtskompatiblem Programmiermodell
Eigenschaften der MC68000-ISA:

1. 32 Bit breite Datenregister, 32 Bit beite Adressregister
2. 24 Bit breiter Adressbus, 16 Bit breiter Datenbus
3. 2 Betriebsmodi (System- und User-Mode)
4. Byteorientierter Adressraum (linear) $\Rightarrow 2^{24}$ Byte $= 16$ MByte Spreicher adressierbar
5. Format beim Speicherzugriff: Höherwertiges Byte zuerst (big endian)

Adr.	b_{15} ... b_8	b_7 ... b_0
i	höherwertiges Byte	niederwertiges Byte
i+2		

Registermodell der MC68000-ISA:



Statusregister der MC68000-ISA

Die einzelnen Bits des Statusregisters werden auch ,,Flag" genannt.

Flag	Bedeutung
C (carry, Übertrag)	Bereichsüberschreitung (V für Vorzeichenlose Zahlen)
V (overflow, Überlauf)	Ergebnis einer arithmetischen Operation kann nicht dargestellt werden (Zweierkomplement)
Z (zero, Null)	Das Ergebnis einer Operation ist Null.
N (negative, Negativ)	Das Ergebnis einer Operation ist eine negative Zahl (höherwertigstes Bit ist '1').
X (extension, Erweiterung)	Wie **carry**-Flag, wird aber im Gegensatz zum carry-Flag von einigen Befehlen (Ladeoperationen, spez. Tests) nicht verändert.

Der **CMP**-Befehl arbeitet wie **SUB** ohne das Ergebnis zu speichern.

Maschinendatentypen der MC68000-ISA

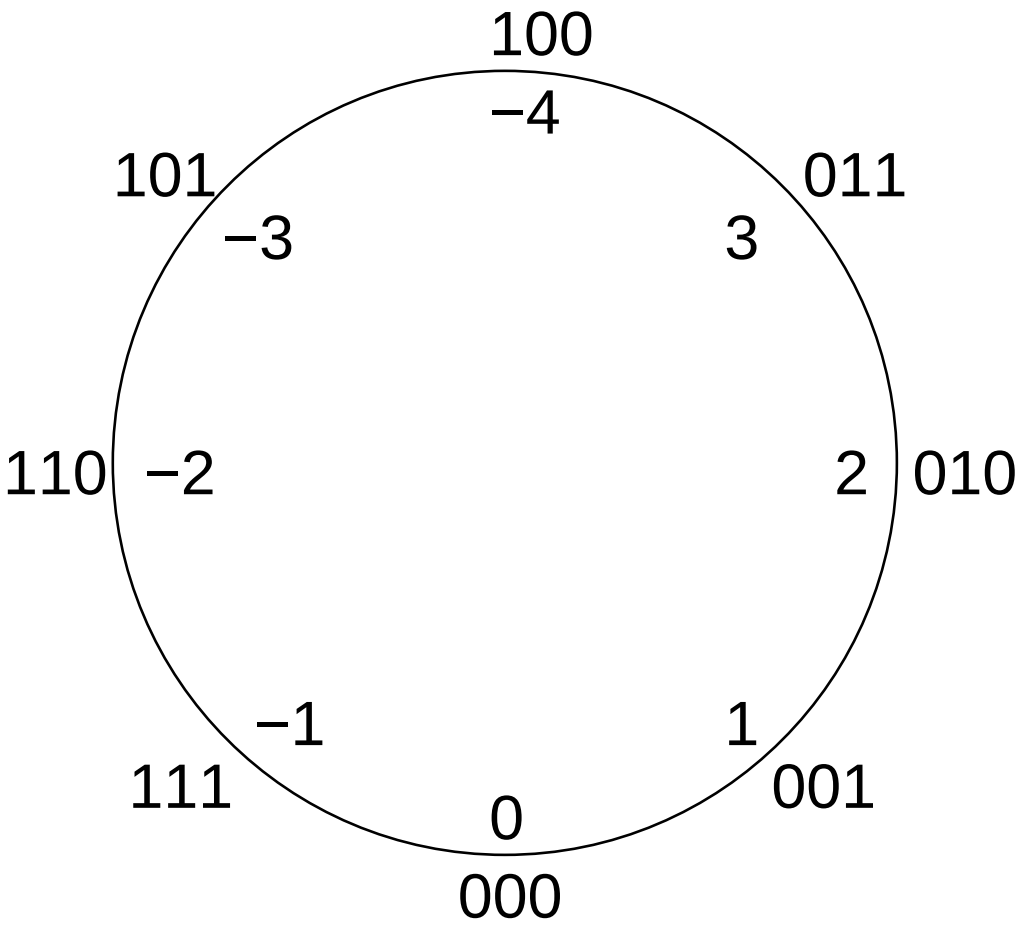
primitive Datentypen:

Typ	Wertemenge	Operationen
Bit	0,1	AND, OR, XOR, Negation, Vergleich
Byte	Bitmuster (8 Bit)	Vergleich, Schieben, bitweise Operationen
Wort	Bitmuster (16 Bit)	...
Langwort	Bitmuster (32 Bit)	...

nicht-primitive, nicht-strukturierte Datentypen:

Typ	Wertemenge	Operationen
Ganze Zahlen (Integer)	Zweierkomplement	ADD, MULT, ..., Vergleiche
Fließkommazahlen	...	FADD, FMULT, ..., Vergleiche
Zeichen (Character)	ASCII, ISO	Vergleich

Erinnerung: Zweierkomplement



3 Bit: $-2^2 <= x <= 2^2 - 1$

000	0
001	1
010	2
011	3
100	−4
101	−3
110	−2
111	−1

Vorteile: Einfache Addition, keine doppelte Null

Maschinenbefehlssatz der MC68000-ISA

Befehlsübersicht unter:

`http://pdv.cs.tu-berlin.de/Info4-SS2001/68000-Quickreference.txt:`

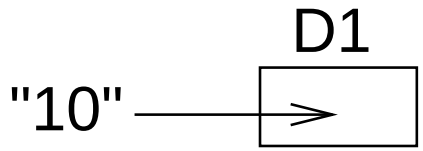
Es gibt:

- Ladeoperationen (MOVE.L, ...)
- Arithmetische Operationen, Logische Operationen (ADD.L, CMP.L, AND)
- Sprünge, bedingte Sprünge, Unterprogrammaufrufe, ... (BRA, BNE, JSR, ...)
- sonstiges

Adressierungsarten der MC68000-ISA (I)

Beschreibung der Adressierungsarten unter:

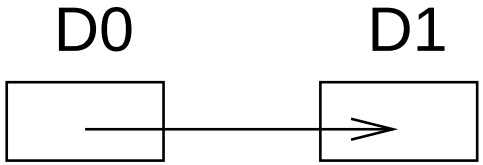
`http://pdv.cs.tu-berlin.de/Info4-SS2001/68000-Adressierungsarten.txt`



Konstanten-Adressierung (immediate Addressing):

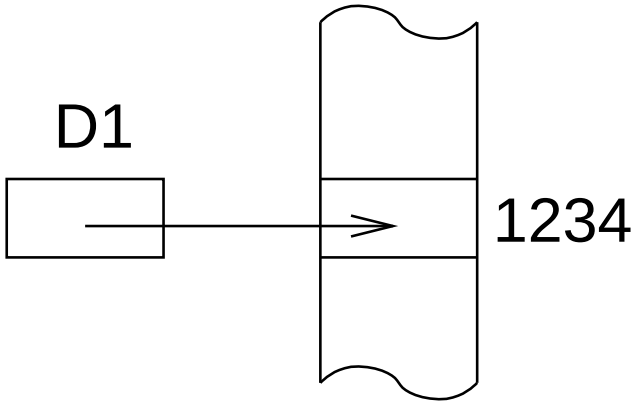
- Verwendung von Konstanten bei Zuweisungen
- Beispiel:
  `MOVE.W #10,D1 ; [D1]<-10 (dezimal)`
- Vorteil: keine Speicherzyklen
- Nachteil: u.U. zu kleine Operatoren

Adressierungsarten der MC68000-ISA (II)



Daten- oder Adreßregister direkt:

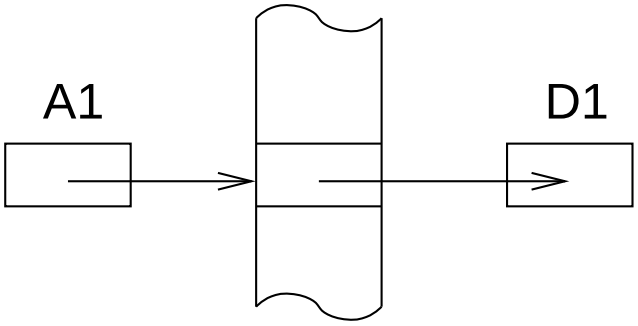
- Bewegung nur innerhalb der Register
- Beispiel:
  `MOVE.W D0,D1 ; [D1]<-[D0]`



Absolute Adressierung:

- Zugriff auf absolute Speicherzellen
- Beispiel:
  `MOVE.W D1,1234 ; [M(1234)]<-[D1]`
- Vorteil: einfache Adressbestimmung, kein zusätzlicher Zyklus
- Nachteil: nur statische Adressen

Adressierungsarten der MC68000-ISA (III)



Adreßregister indirekt:

- Register als Zeiger
- Beispiel:

```
MOVE.W (A1),D1 ; [D1]<-[M([A1])]
```

Adreßregister indirekt mit Postinkrement/Prädekrement:

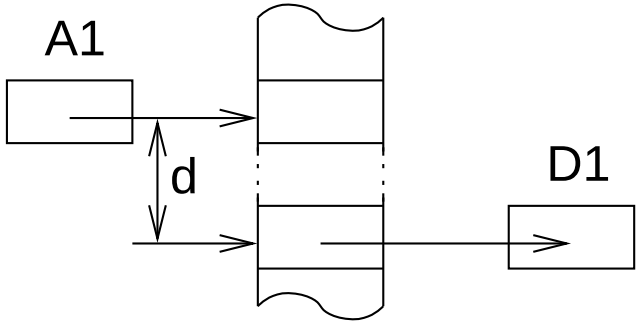
- Zusätzliche Operation auf den indirekt verwendeten Registern vor bzw. nach der Operation
- Beispiel:

```
MOVE.W (A1)+,D1 ; [D1]<-[M([A1])],
                ; [A1]<-[A1]+2
```

- z.B. sinnvoll für schnelles Kopieren

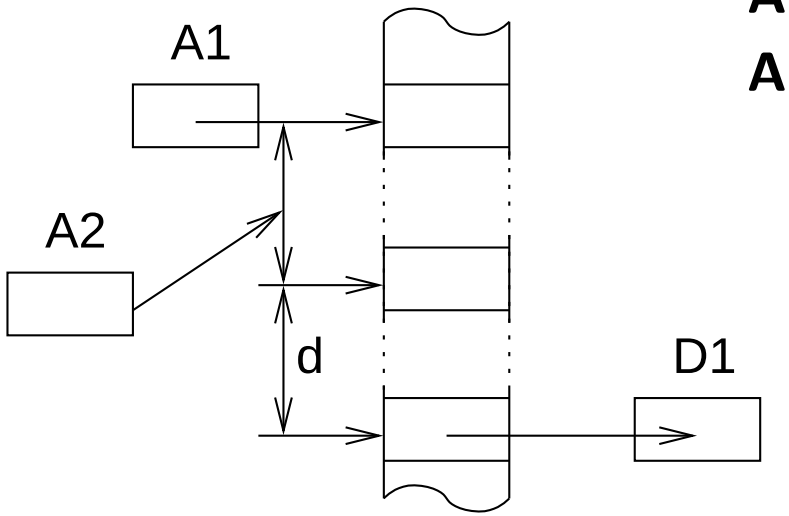
Adressierungsarten der MC68000-ISA (IV)

Adreßregister indirekt mit Adreßdistanz:



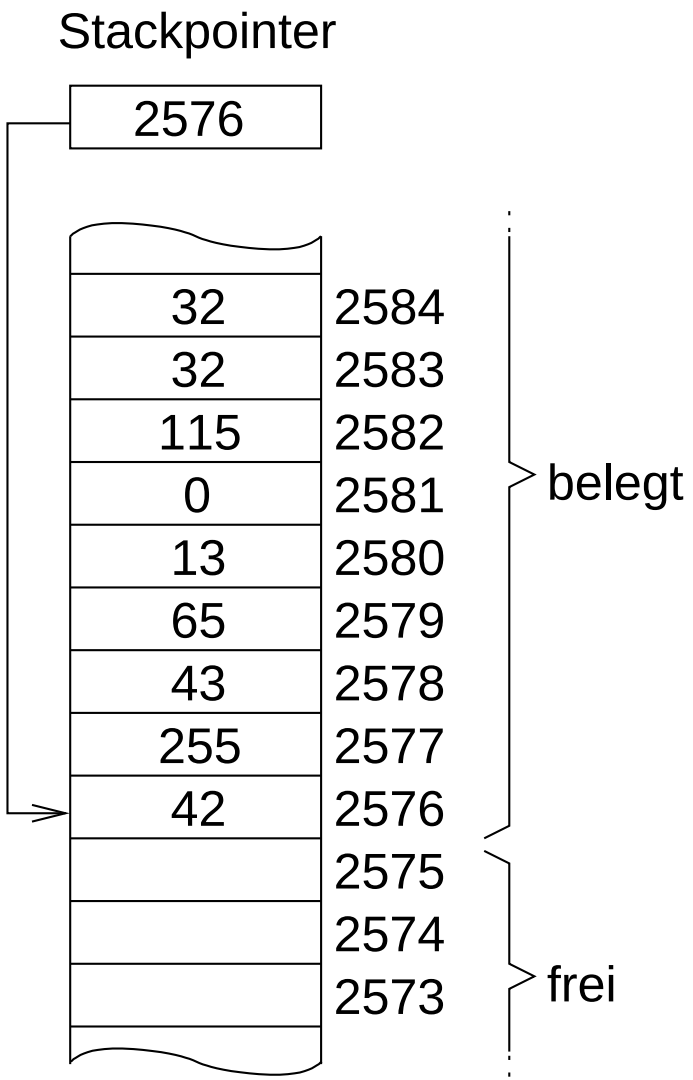
- Indirekte Adressierung mit konstanter Distanz
- Beispiel:
  `MOVE.W d(A1),D1 ; [D1]<-[M([A1]+d)]`
- Sinnvoll für den Zugriff auf Strukturen

Adreßregister indirekt mit Index und Adreßdistanz:



- Wie zuvor nur mit zusätzl. Register als Index
- Beispiel: `MOVE.W d(A1,A2.W),D1 ;`
  `[D1]<-[M([A1]+[A2]+d)]`
- Vorteil: flexibel
- Nachteil: zusätzliche Speicherzyklen

MC68000-Assembler: Stackaufbau



Stack:

- Liegt im ,,normalen" Adressraum
- Wächst zu kleineren Adressen hin
- ein Benutzer- und ein System-Keller
- Kellerzeiger (Stackpointer) zeigt auf das oberste Kellerelement (niedrigste belegte Adresse)
- Kellerzeiger steht in Register A7 (A7=SP)
- Wichtig zum Ablegen temporärer Daten

Arbeit mit dem Stack

- Register sind die Variablen in Assemblerprogrammen.
- Normalerweise reichen diese Register nicht gleichzeitig für alle Programme.
- Deshalb können Register auf dem Stack zwischengespeichert werden:
  Ablegen von Daten auf dem Stack: `MOVEM.L D0-D7/A0-A6, -(SP)`
  Zurückholen dieser Daten vom Stack: `MOVEM.L (SP)+, D0-D7/A0-A6`
  Achtung: Am Ende muß der Stack so aussehen, wie am Anfang!!!
- Beispiel: Speichern des Systemzustands vor der Unterbrechungsbehandlung
- Weitere Funktion: Parameterübergabe beim Unterprogrammaufruf

MC68000-Assembler: Beispiel

Aufgabe: In einer Tabelle, die in einem vorgegebenen Speicherbereich abgelegt ist und die nachstehende Form hat, soll mittels einer Matrikelnmmer die zugehörige Note gesucht werden.

	Note	Matrikel–nummer	Name (Länge variabel, aber gerade)	Null–Byte
1. Datensatz	'1'	190000	Müller, Hans	0
2. DS	'3'	190001	Meier, Uwe	0
letzter DS	'2'	190255	Schmidt, Kalle	0
"EOF"	0			

Der Prozeduraufruf hat das Format:

```
void suche_note(int MatNr, char *Note)
```

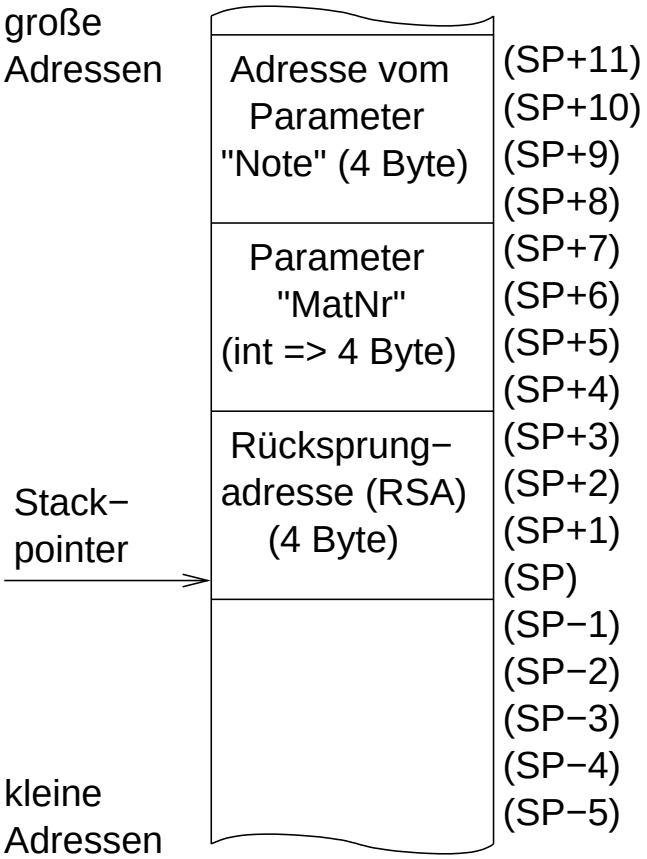
Gegeben sind des weiteren:

`TabAnfang`, `TabEnde`, eine Zahl vom Typ `int` belegt 4 Byte

Gerade Namenslänge, weil Zugriff auf ,,Longword" nur an geraden Adressen

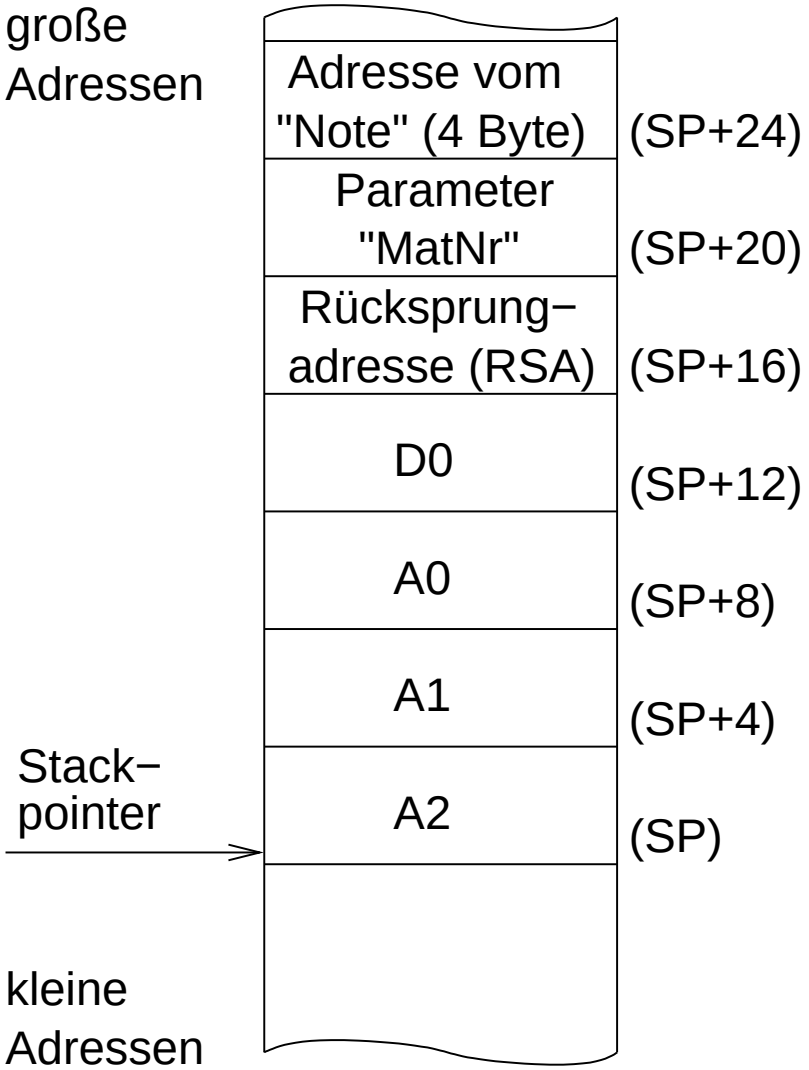
MC68000-Assembler: Unterprogrammaufruf I

gegeben: Aufgerufen wurde „`void suche_note(int MatNr, char *Note)`" mit `JSR`, wobei `int` 4 Byte belegt und das Unterprogramm die Register D0, A0, A1 und A2 benutzt und deshalb sichert und zurücksichert.



- Abbildung zeigt den Stack direkt nach dem Funktionsaufruf
- Das Unterprogramm sichert D0, A0, A1 und A2 auf dem Stack.
- D0, A0, A1 und A2 **müssen** vor dem Rücksprung mit `RTS` wieder vom Stack entfernt werden!
- Erinnerung: „Big endian", d.h. höherwertigeres Byte an der höheren Adresse

MC68000-Assembler: Beispiel (Forts.)



Bemerkungen:

- Abbildung zeigt den Stack nach dem Aufruf von `suche_note` und dem Sichern der Register D0, A0, A1 und A2 auf dem Stack
- Die Parameter der Funktion können über das Adressregister SP indirekt mit Adressdistanz $+20$ (Matrikelnummer) und $+24$ (Adresse an dem das Ergebnis gespeichert werden soll) adressiert werden.
- D0, A0, A1 und A2 **müssen** vor dem Rücksprung mit `RTS` wieder vom Stack entfernt werden!

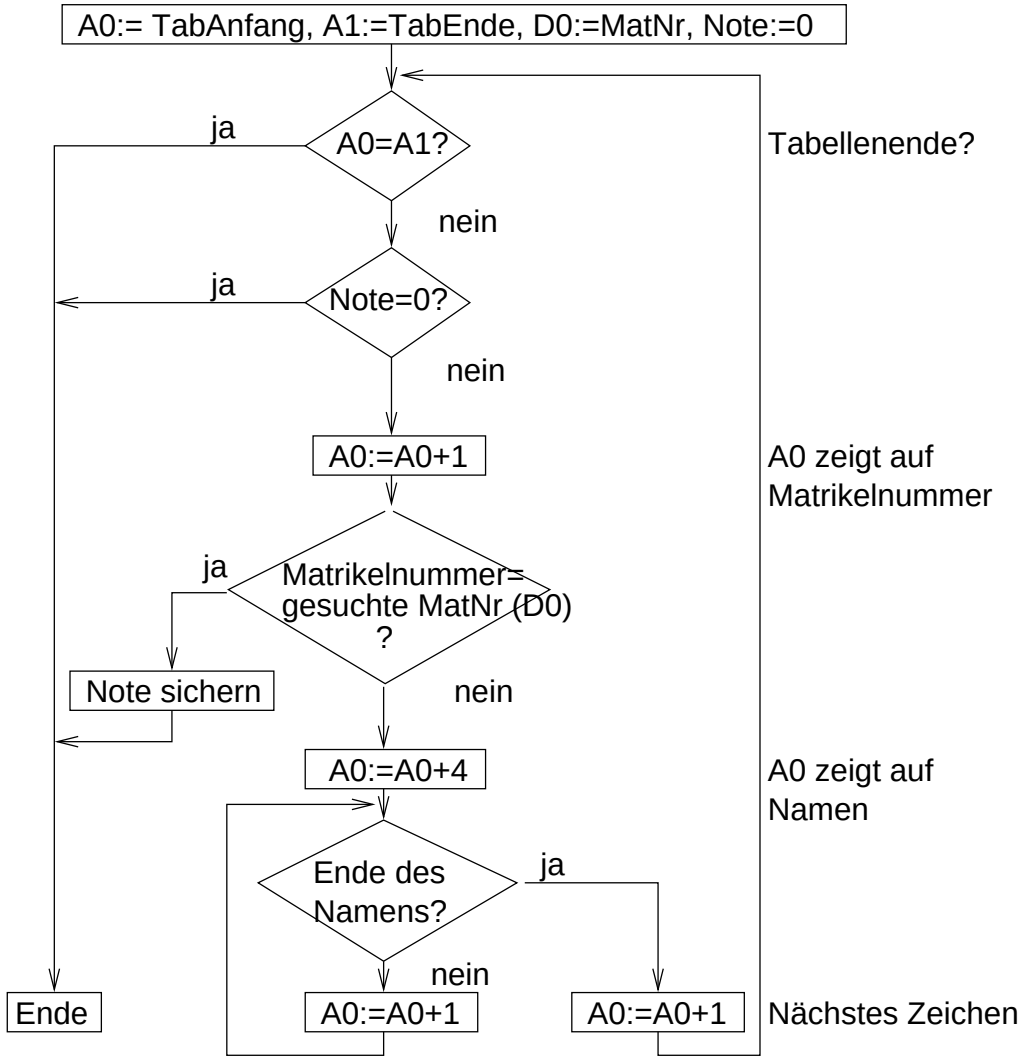
MC68000-Assembler: Beispiel (Forts.)

```
| A0 : Tabellenanfang
| A1 : Tabellenende
| A2 : Adresse des Ergebnisses (Note)
| D0 : Gesuchte Matrikelnummer
#define aMatNr 20(SP)
#define aNote  24(SP)
.EVEN
.GLOBAL _suche_note
_suche_note:  MOVEM.L D0/A0-A2, -(SP)   | Inhalt der benutzten Register sichern
              LEA _TabAnfang, A0        | globale Variable in Register kopieren
              LEA _TabEnde, A1          |
              MOVE.L aMatNr, D0         |
              MOVEA.L aNote, A2         |
              MOVE.B #0,(A2)            | "Default"-Ergebnis 0 = nicht gefunden

              ...                       | hier steht der "eigentliche" Code

       Ende:  MOVEM.L (SP)+, D0/A0-A2   | gesicherte Register zurueckholen
              RTS
```

MC68000-Assembler: Beispiel (Ablaufplan)



MC68000-Assembler: Beispiel (Programm)

Dieser Code ist einzufügen in den vorigen Rahmen:

```
Schleife1: CMP.L      A0,A1          | Ende der Tabelle erreicht?
           BEQ        Ende           | Wenn ja: Ende

           CMPI.B     #0,(A0)+       | Note=0 ("EOF")?
           BEQ        Ende           | Wenn ja: Ende

           CMP.L      (A0)+,D0       | gesuchte Matrikelnummer gefunden?
           BEQ        Erfolg         | Wenn ja: Note kopieren

Schleife2: CMPI.B     #0,(A0)+       | Ende des Namens?
           BEQ        Schleife1      | Wenn ja: naechster Datensatz

           BRA        Schleife2      | Naechstes Zeichen

Erfolg:    MOVE.B     -5(A0),(A2)    | Note uebertragen
```

Bemerkungen zur Hausaufgabe 1:

Die Aufgaben sind zu finden unter:
`http://pdv.cs.tu-berlin.de/Info4-SS2001/Aufgaben.html`

Eine Anleitung, wie die Assemblerumgebung funktioniert steht unter:
`http://pdv.cs.tu-berlin.de/Info4-SS2001/AssEnt.html`

Unbedingt `'gmake'` verwenden und Umgebungsvariable richtig setzen (Unterlagen zur Assemblerumgebung lesen, besonders Kapitel 3)!

Notwendig zum Scheinerhalt: Abgabe (Vorführen!) der gelösten Aufgabe 1 bis einschließlich Freitag 11.5. (16:00 ist Schluß)!

Viel Spaß und Erfolg!