

# Assemblerprogrammierung

Matthias Jung

11. November 2008

## 1 Turing-Maschine

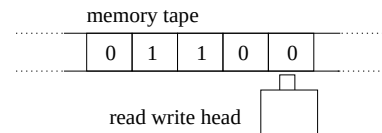
### Alan Turing (1936)

Als Gedankenexperiment für grundlegende Eigenschaften.

Besteht aus:

- Endlosen Band mit Symbol-Feldern das nach links oder rechts bewegt werden kann.
- Lese und Schreibkopf
- Speicher für ein Symbol
- Anweisungen: right, left, read, write

### Struktur:



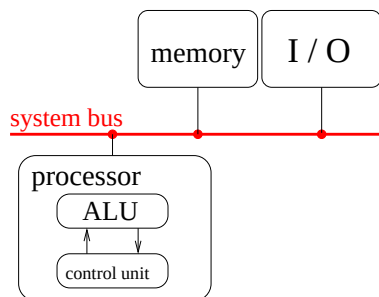
Jede Maschine die eine TM repräsentiert kann die Arbeit aller andern Maschinen die TM repräsentieren durchführen.

### Halteproblem:

Es exestiert kein Algorithmus der für alle Programme angeben kann ob die Turingmaschine anhält.

## 2 Von-Neumann-Maschine

### Struktur:



Die Von-Neumann-Maschine repräsentiert eine Turingmaschine. Daten und Programmcode stehen im gleichen Speicher.

### Befehlszyklus:

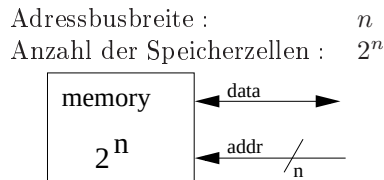
- *fetch* (holen)
- *decopde* (dekodieren)
- *execute* (bearbeiten)
- *writeback* (zurückschreiben)

### Pro und Contra:

- + Programm kann (von sich selbst) modifizieren (nicht 'fest verdrahtet')
- + Tools: *Assembler*, *Compiler* (sont Bits per Hand)
- *bus bottleneck*
- Gefahr durch Programmierung (z.B. *buffer overflow*)

## Speicher:

Der Speicher bei Von Neumann wird verwendet für Daten und für das Programm selbst (vgl Harvard-Architektur). Er besteht aus nummerierten Speicherzellen. Der Inhalt wird binär dargestellt. Die Anzahl der adressierbaren Speicherzellen wird durch die Breite des Adressbusses bestimmt. Die Maximale Speichergröße ist also abhängig von der Speicherzellgröße und der Breite des Adressbusses.



## Speicherhierarchie:

### Registerspeicher:

Der Registerspeicher liegt im Prozessor. Er ist sehr schnell und hat eine kleine Kapazität.

### Arbeitsspeicher:

Der Arbeitsspeicher ist am Systembus angeschlossen. Er ist mittelschnell und mittlgross.

### Cache:

Kleiner Zwischenspeicher (siehe unten)

### Hintergrundspeicher:

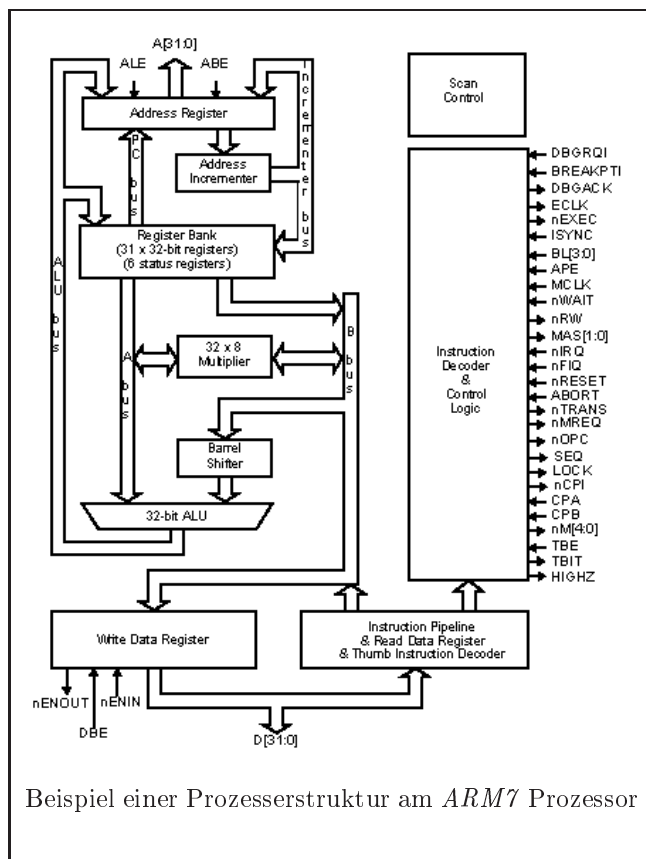
Plattenspeicher oder Wechselmedien. Er ist langsam aber die Kapazität ist quasi unbegrenzt.

## Cache:

Da der Arbeitsspeicher viel langsamer ist als die CPU und der Bus das Holen von Daten aus dem Speicher auch verlangsamt, ist es sinnvoll einen Schnellen kleinen Zwischenspeicher zu benutzen (Cache). Dies ist vor allem von Vorteil wenn man Daten oft benötigt. (while). Der Cache wird zwischen Speicher und CPU platziert (Busseitig an CPU!) Der Cache ist transparent, d.h. die CPU merkt von ihm nichts.

- + Schnellerer Zugriff
- + Entlastung BUS
- Cache voll (kostet Zeit)
- Interrupts: Cache komplett leeren und danach wieder herstellen...
- Unglückliche Folgen von Speicherzugriffen (kostet Zeit)

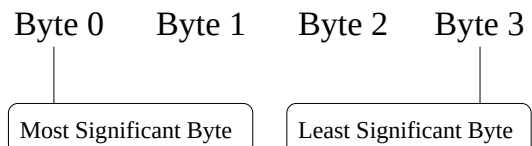
## Prozessor



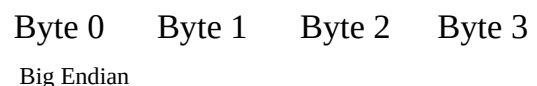
## Register:

Register dienen zum schnellen Zwischenspeichern von Werten bei der Verarbeitung. Sie sind von der Darstellung mit dem Speicher abgestimmt.

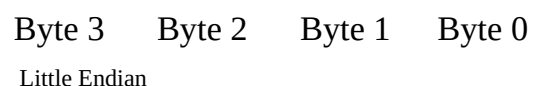
Anordnung im Register:



MSB an niedrigster Adresse



LSB an niedrigster Adresse

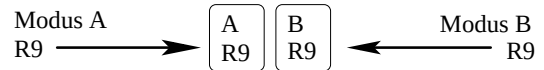


### Bearbeitungsmodi:

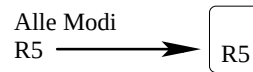
1. User-mode  
normale Bearbeitung
2. System  
privilegierte Betriebssystemfunktionen
3. Supervisor-mode  
geschützter Modus für das Betriebssystem
4. Interrupt  
Interrupt Behandlung
5. Fast interrupt  
Schnelle Interruptbehandlung
6. Abort  
Prozess und Speicherschutz
7. Undefined  
Emulation undefinierter Befehle

### Banked und Unbanked bei Registern:

Banked:



Unbanked:



Beispiel ARM7:

| User Mode        |         |            |         |           |         |         |
|------------------|---------|------------|---------|-----------|---------|---------|
| Privileged Modes |         |            |         |           |         |         |
| Exception Modes  |         |            |         |           |         |         |
| User             | System  | Supervisor | Abort   | Undefined | IRQ     | FIQ     |
| R0-R7            | R0-R7   | R0-R7      | R0-R7   | R0-R7     | R0-R7   | R0-R7   |
| R8-R12           | R8-R12  | R8-R12     | R8-R12  | R8-R12    | R8-R12  | R8-R12  |
| R13-R14          | R13-R14 | R13-R14    | R13-R14 | R13-R14   | R13-R14 | R13-R14 |
| PC               | PC      | PC         | PC      | PC        | PC      | PC      |
| CPSR             | CPSR    | CPSR       | CPSR    | CPSR      | CPSR    | CPSR    |
|                  |         | SPSR       | SPSR    | SPSR      | SPSR    | SPSR    |

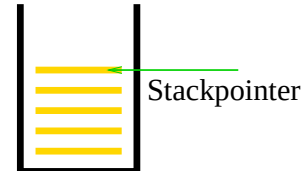
## Wichtige Register

### Programmcouter:

- Enthält die Adresse des aktuellen Befehls im Programm
  - Wird linear inkrementiert.
  - Aber kann auch gesetzt werden (Verzweigungen)
- ! Kann bei Überschreibungen zu ungewünschten Wirkungen führen

ARM7 R15

### Stackpointer:



Im Stackpointer steht die oberste Adresse für den Arbeitsspeicherbereich Stack  
ARM7 R13 individuell für Modi

### Linkregister:

Speicherplatz für *return address* bei Unterprogrammaufrufen und *interrupts*. Die Adresse muss vor einem Unterprogrammaufruf gesetzt werden.

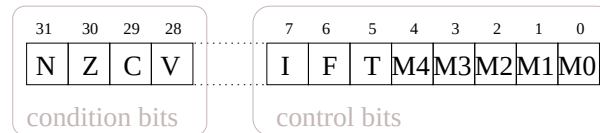
ARM7 R14 individuell für Modi

**Statusregister:** Das Statusregister enthält Informationen über den Status des Prozessors, der Prozesse und Operationen.

- Current Process Status Register (CPSR)
- Saved Process Status Register (SPSR)

## Statusregisterbits:

Beispiel ARM7:



N: *negative flag*, Vorzeichen des Ergebnisses einer Operation (1=negativ 0=positiv)

Z: *zero flag*, enthält eine 1 wenn das Ergebnis einer Operation 0 ergab. ("gleich" bei verlgleich)

C: *carry flag*, carry bei Addition, Substarktion, shifts, rotate

V: *overflow flag*, Additions-, Substarktionsoverflow

I: *interrupt* (IRQ) disable flag

F: *fast interrupt* (FIQ) disable flag

T: *thumbmode*, 0=ARM instruktion, 1=Thumb Instruktion

M: *processormode* (10000=User, 11111=System ...)

**Adressregister:** Das Adressregister hält die Adreszen für den Speicherzugriff über den Adressbus. An ihm befindet sich ein incremter der für ein schnelles Erhöhen der Adresse sorgt.

## Weitere Prozessorkomponenten

### ALU - arithmetic logical unit:

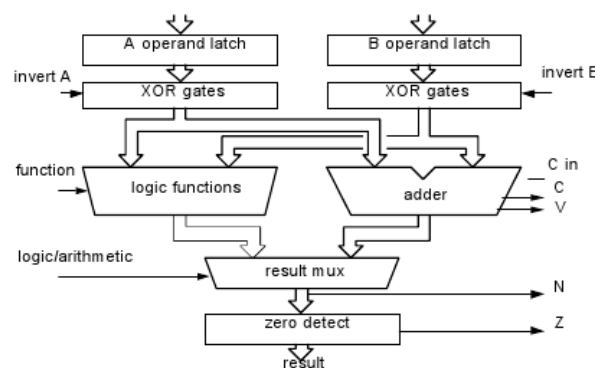
Die ALU dient zur Berechnug mit folgenden Operationen:

- arithmetic:  
add,sub,eq,mult,div ...
- logical:  
not,and,or,xor ...
- shift, rotate

### barrel shifter:

- dient zum schnellen Verschieben von Worten
- für arithmetische und Adressoperationen

### ARM6 ALU implementation



## 2.1 Adressierung

### Direct Addressing:

Beim Direct Addressing steht die Adresse direkt im Befehl. Wenn es z.B. aber  $2^{32}$  Adressen gibt kann man die Adresse in einem 32bit-Befehl nicht unterbringen. Man kann mit *Offset Addressing* oder **Indirect Addressing** trotzdem zu allen Speicherzellen gelangen.

### Offset Addressing:

- immediate: unsigned Zahl wird zu base register addiert (subtrahiert).
- register: offset in Register wird zu base register addiert (subtrahiert).
- scaled register: offset in Register + immediate wird zu base register addiert (subtrahiert).

### Register-indirect Addressing:

Die Adresse auf die Zugriffen werden soll wird in ein Register geschrieben.

Beispiel ARM7:

```
LDR r0,[r1]
STR r0,[r1]
```

[r1] dereferenziert die Adresse die in dem Register r1 steht

### Offset Addressing: *immediate*

- Pre-indexing:  
LDR r0,[r1,#4] addiert 4 zur Speicheradresse
- Post-indexing:  
LDR r0,[r1],[#4] addiert nach der Befehlsausführung 4 zur Speicheradresse. (Erspart zusätzliche Instruktionen)

### Block-copy addressing:

damit kann man 1-16 Register von bzw zum Speicher transferieren

STMIA/LDMIA: Increment After

STMIB/LDMIB: Increment Before

STMDA/LDMA: Decrement After

STMDB/LDMDB: Decrement Before

```
LDMAIA r1,{r0,r2,r5}
```

entspricht also:

```
LDR r0,[r1],#4
LDR r2,[r1],#8
LDR r5,[r1]
```

### Stackadressierung

Der Stack ist ein reservierter Arbeitsspeicherbereich. Es ist ein Stapelspeicher (LIFO). Der Stackpointer zeigt auf den Kopf des Stapels (TOS *top of stack*). Der Stack wächst zu fallenden Adressen.

#### Schreiben:

Stackpointer 2/4 erniedrigen, Speichern

#### Lesen:

Lesen, Stackpointer um 2/4 erhöhen.

#### Unterprogramm sprung:

PC wird auf Stack geschrieben.

#### Return:

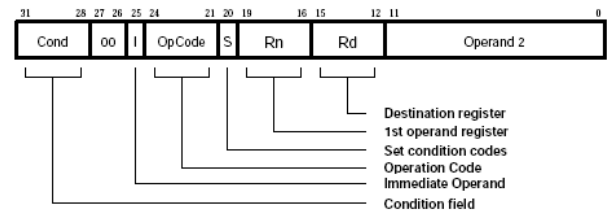
PC wird vom Stack gelesen, Der Programmierer hat dafür zu sorgen das der Stackpointer wieder an der alten Stelle steht.

## 2.2 Assembler Befehle

### Eigenschaften ARM Befehlssatz:

- Load/Store-Architektur. Hole Daten aus Speicher, bearbeite sie in Registern und gebe sie wieder in dem Speicher.
- 3-Adress Befehlsformat:
  - 1 destination - 1 register
  - 2 source - 1 source ist immer ein Register, der zweite kann ein Register oder ein Immediate Wert sein.
- Conditional execution (Bedingte Ausführung **jedes** Befehls). Condition wird immer geprüft also auch immer ein Rechenzyklus.
- Load/Store auf mehrere Register
- RISC - Reduced Instruction Set Computing (Gegenteil Complex Instruction Set Computing)

### Befehlsaufbau:



### Condition Bits:

| bits | Type         | Überprüfung |
|------|--------------|-------------|
| 0000 | EQ Equal     | Z set ?     |
| 0001 | NE Not equal | Z clear ?   |
| 0110 | VS Overflow  | V set ?     |
| 1000 | HI Higher    | N set ?     |
| 1001 | LS Lower     | N set ?     |
| 1110 | AL Always    | any         |

### Arithmetic operations:

ADD r0,r1,r2     $r0 = r1 + r2$     Addition  
 SUB r0,r1,r2     $r0 = r1 - r2$     Subtraktion  
 RSB r0,r1,r2     $r0 = r2 - r1$     Reverse substr.

Die Konditionbits werden nicht verändert

### Logic operations:

AND r0,r1,r2     $r0 = r1 \wedge r2$     Und  
 ORR r0,r1,r2     $r0 = r1 \vee r2$     Oder  
 EOR r0,r1,r2     $r0 = r2 \oplus r1$     Exklusiv oder  
 BIC r0,r1,r2     $r0 = r2 \wedge \overline{r1}$     bitclear

### Register movement:

MOV r0,r1     $r0 = r1$     Kopiere Z,Q  
 MVN r0,r1     $r0 = \overline{r1}$     Kopiere Komplement

### Memory movement:

STR r0,[r1]    Quelle, Ziel !  
 LDR r1,[r0]    Ziel, Quelle !

### Compare:

CMP r0,r1    Vergleicht auf gleichheit, größer, ...  
 CMN r0,r1    ?  
 TST r0,r1    ?  
 TEQ r0,r1    ?

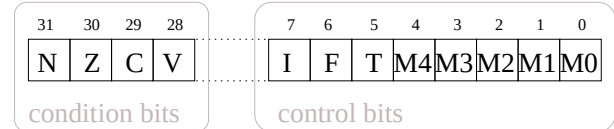
## 2.3 Interrupts

**Hardware Dispatcher:** Der Hardware Dispatcher kümmert sich um das *Interrupt handling* also entscheidet ob ein Interrupt zulässig ist oder nicht und bereitet die ISR *Interrupt Service Routine* vor.

### Ablauf:

- IRQ *interrupt request*
- Interrupt controller - Priorisierung
- Status zwischenspeichern
- ISR auswählen
- Status für Rückkehr auf Stack sichern
- ISR Vorbereiten und Starten
- ISR Beenden
- RTE *return from exception*, Status widerherstellen

### Interrupt disable bits im Statusregister: ARM7

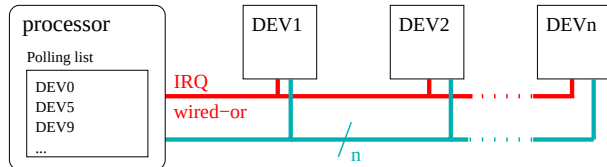


I, F -bits können Interrupts ein und ausschalten.

### Eigenschaften von Interrupts:

- ISR können ISR unterbrechen. (Je nach Priorität)
- Interrupts können Priorisiert werden. (Siehe *Polling* / *Daisychain*)
- Der Programmierer

### Polling:

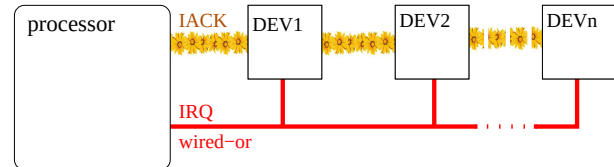


Die *Devices* haben eine gemeinsame Leitung (*wired-or*) für IRQ. Ein Device macht ein IRQ. Der Prozessor merkt dies und befragt alle Devices (nach einer Prioritätenliste) wer den IR ausgelöst hat.

### Pro und Contra:

- + Veränderbarkeit leicht möglich
- Overhead im Prozessor

### Daisychain:



Wie bei Polling teilen sich die Devices den IRQ-Eingang des Prozessors. Falls der Prozessor ein IRQ bekommt sendet er, wenn er Zeit hat, ein IACK (*interrupt acknowledge*). Weiter Anfragen werden blockiert. Die Devices reichen das IACK weiter wenn sie keinen IRQ gestellt haben. Die Priorisierung wird durch die Verdrahtung der Kette festgelegt.

### Pro und Contra:

- + Bearbeitung der "Prioritätenliste" ausserhalb des Prozessors
- Fest verdrahtet und schwer veränderbar
- Prozessor sieht nur nächstes Device in Kette

## 2.4 BUSSE

### Systembus:

Der Systembus besteht aus:

- Adressbus - Dient zur Übertragung von Speicheradressen
- Datenbus - Übertragung von Daten zwischen Rechnerkomponenten
- Steuerbus - Bussteuerung, Systemsteuerung
- Stromversorgung
- Takt

(Bei manchen Systemen ist der Adress- und der Datenbus gemultiplext.)

### Synchronbus:

- Festes Zeitraster für Datenübertragung. (CLOCK-Leitung)
  - Master fordert vom Slave
- + Einfaches Busprotokoll
- Langsamstes Device bestimmt die Geschwindigkeit für alle.

### Asynchronbus:

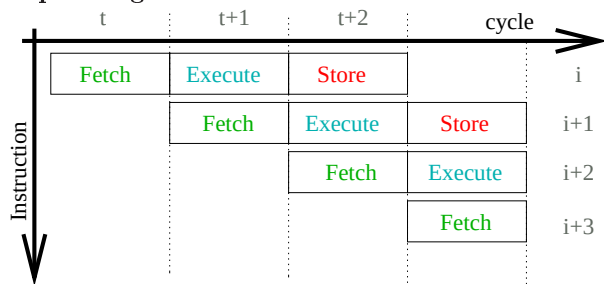
- Kein festes Zeitraster, Handshake betrieb!
  - Master wartet auf Bereitschaft
- + Individuelle Transfergeschwindigkeiten
- Komplexes Busprotokoll

### Arbiter:

Der Arbiter (elektrisch, digital, Software) regelt die Zugriffskontrolle (auch nach Prioritäten) auf den Bus. Er wird häufig bei Bussystemen mit mehreren Busmastern verwendet, um zu entscheiden welcher Busmaster Zugriff bekommt.

## 3 Moderne Architekturen

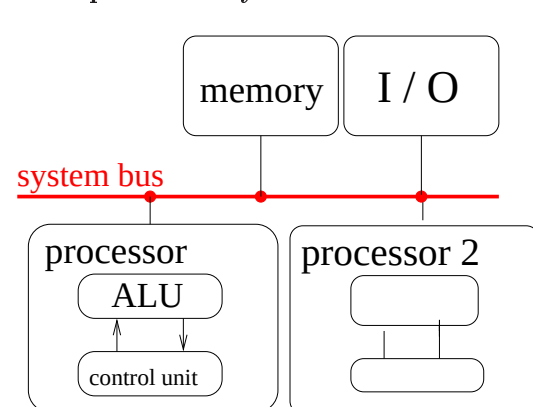
### Pipelining:



Beim Pipelining werden in einem Cycle mehrere Dinge bearbeitet. Beispiel.: Während die ALU etwas berechnet, wird schon der nächste Operand geladen. (Vgl Ausgabe Mensa)

- + Beschleunigung
- + Effektivere Ausnutzung der Prozessorkomponenten.
- Weniger Überblick was in der CPU passiert.
- Wenn ein Interrupt auftritt war alles Vorarbeiten umsonst!

### Multiprocessor Systems:



Bei einem Multiprocessorsystem gibt es mehr als eine CPU und somit ist eine "Echt-Parallele" Verarbeitung möglich.

### Floating Point Unit:

Für die Normale CPU ist eine Numerische Berechnung sehr aufwendig. Deshalb setzt man einen Coprozessor ein, der spezielle Numerische Aufgaben lösen kann.

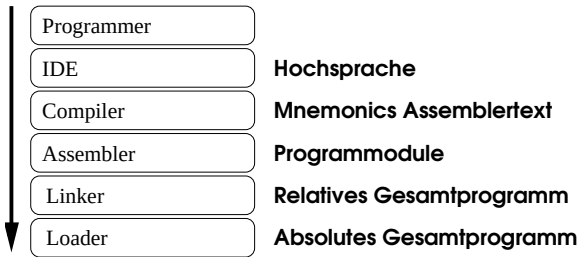
### Shared Memory TODO

### Multicore TODO

### Hyperthreading TODO

## 4 Vom Algorithmus zu den Befehlen

### Der Weg von der Idee zum Programm:

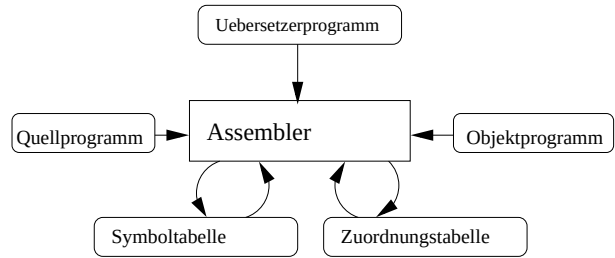


Ist der Entwicklungssystemprozessor gleich dem Zielsystemprozessor so spricht man von *Assembler/Compiler*. Ist der Entwicklungssystemprozessor ungleich dem Zielsystemprozessor so spricht man von *Cross-Assembler / Cross-Compiler*.

#### Linker und Loader:

Der *linker* erzeugt aus zwei oder mehreren relativen Programmodulen ein relatives Gesamtprogramm. Relativ bedeutet das innerhalb des Programms die Adressbezüge absolut sind aber insgesamt noch verschiebbar im Adreßraum des Speichers. Der *loader* rechnet die relativen Adressen (mit vorgegebener Anfangsadresse) in absolute Adressen um.

### Was macht der Assembler?:



|                    |                                                            |
|--------------------|------------------------------------------------------------|
| Quellprogramm:     | Programm in <i>mnemocode</i>                               |
| Symboltabelle:     | Werte für Gerätereisteradr., Adreßwerte von Labels...      |
| Zuordnungstabelle: | Binärcodes für Operationen und Register                    |
| Assembler:         | Enthält Zuordnungszähler zur Berechnung von Adreßdistanzen |
| Objektprogramm:    | binäre Form des Programms in Maschinensprache              |

Es gibt eine Zwei-Phasen-Assemblierung:  
Phase 1: Adresszuordnungen und Aufbau der Symboltabelle  
Phase 2: Zuordnung der Binärcodes und erzeugung eines Protokolls

### Makros

Makros sind sind Anbeschlossene Assemblerprogramme mit symbolischen Namen, die an beliebiger Stelle eines anderen Assemblerprogramms wie ein Befehl aufgerufen werden können. Es können parameter übergeben werden.

- + Erweiterung des Prozessorbefehlssatz.
- + Makrobibliothek für "rapid Prototyping".
- Hoher Speicherplatzbedarf bei häufiger Verwendung (Siehe Unterprogramme).

### Unterprogramme

- + Häufig vorkommende Programmteile nur einmal vorhanden.
- + Modularer Aufbau (Änderbarkeit, Wartbarkeit...)
- Anspruchsvollere Programmierarbeit.

*Call by value, call by reference.*

### Rekursive Unterprogramme:

Rekursive Programme sind Programme die sich selbst aufrufen. Dafür wird eine aufwendige Rettung von Register- und Stackinhalten und deren Restaurierung benötigt.

### Reentrante Unterprogramme:

Ein Programm kann mehrmals von einem äusseren Programm gestartet werden. Damit sich die Einzelprogramme nicht in die Quere kommen dürfen sie nur eigene Speicherbereiche verwenden. Sprich: Keine globalen Variablen!