8. MIKROPROZESSOR-GRUNDLAGEN

Diese Kapitel geht nun einen Abstraktionsebene höher und betrachtet Mikroprozessoren als eine Universalhardware, die durch den Austausch von Programmen flexibel für die Implementierung der verschiedensten, auch komplexeren Abläufe/Algorithmen eingesetzt werden kann. Es gibt dazu eine Einführung in die Assemblerprogrammierung, widmet sich aber auch verschiedenen grundlegenden Architekturprinzipien für die flexiblen Unterstützung von Hochsprachenprogrammen. Der Schwerpunkt liegt hierbei auf der effektiven Adressierung von Operanden und Mechanismen der Programmflusskontrolle.

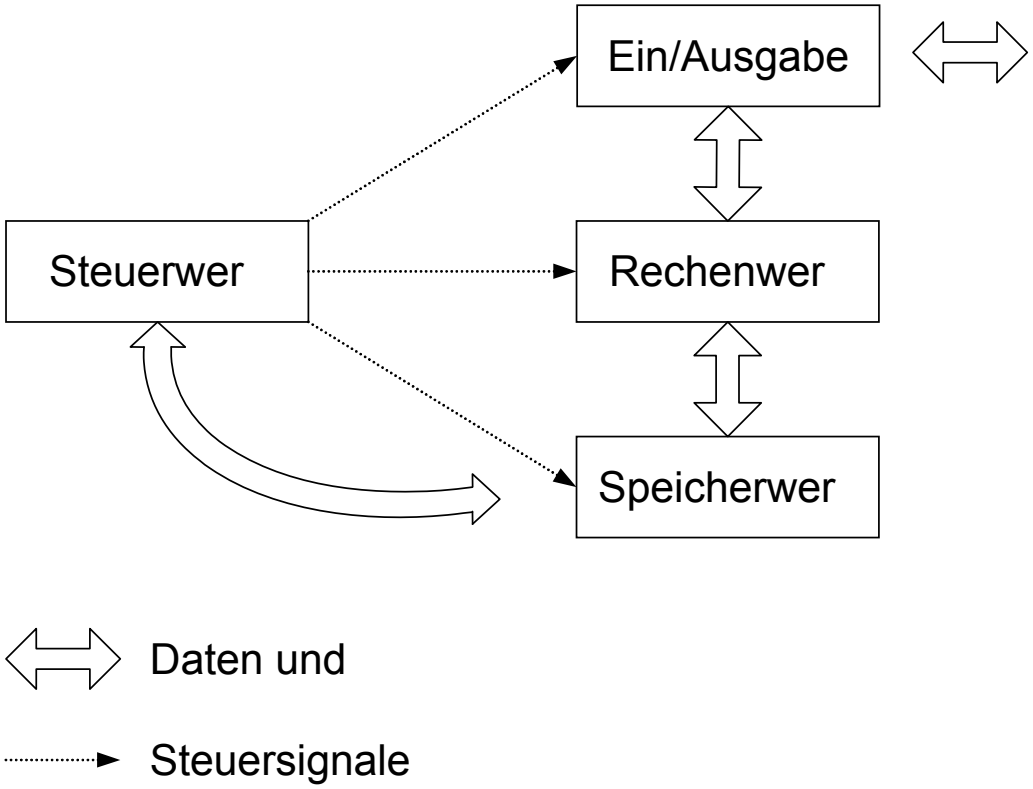
8.1 Befehlsatz-Architekturen

Mittels einer Registertransfer-Beschreibung lassen sich nun komplexere Digitalschaltungen anschaulich und wartbar entwerfen. Durch die Mikroprogrammierung sind auch Funktionsänderungen und Erweiterungen des Verhaltens leicht möglich, weil in den meisten Fällen nur das Mikroprogramm ausgetauscht werden muss, nicht aber die Hardware zu dessen Abarbeitung oder das Operationswerk geändert werden müssen.

Durch diese Methoden ist es auch vergleichsweise einfach, die Hardware so zu gestalten, dass von außen angelegte Kommandos die Funktionalität (interne Abläufe) steuern. Ein komplexes Schaltwerk (mit inneren Zuständen) kann nun so betrachtet werden, als wäre es ein Stück Hardware, das durch Kommandos (Instruktionen, Befehle, Maschinenbefehle) von einer höheren Ebene beliebig in seinem Verhalten gesteuert werden kann. Man erhält also so etwas wie eine Universalhardware.

Wird die Hardware nun so gestaltet, dass sie selbst nach der Abarbeitung eines Kommandos (Maschinenbefehl) das nächste aus einem Speicher holt, lässt sich ein (Mikro-)Prozessor, der nach dem von-Neumann-Prinzip arbeitet, recht einfach realisieren.

Von-Neumann-Architektur



Ein Computer (mehr dazu in Kap. 9) besteht also immer aus:

- Prozessor
- Speicher (Programm- und Arbeitsspeicher)
- Ein-/Ausgabeeinheit (E/A)

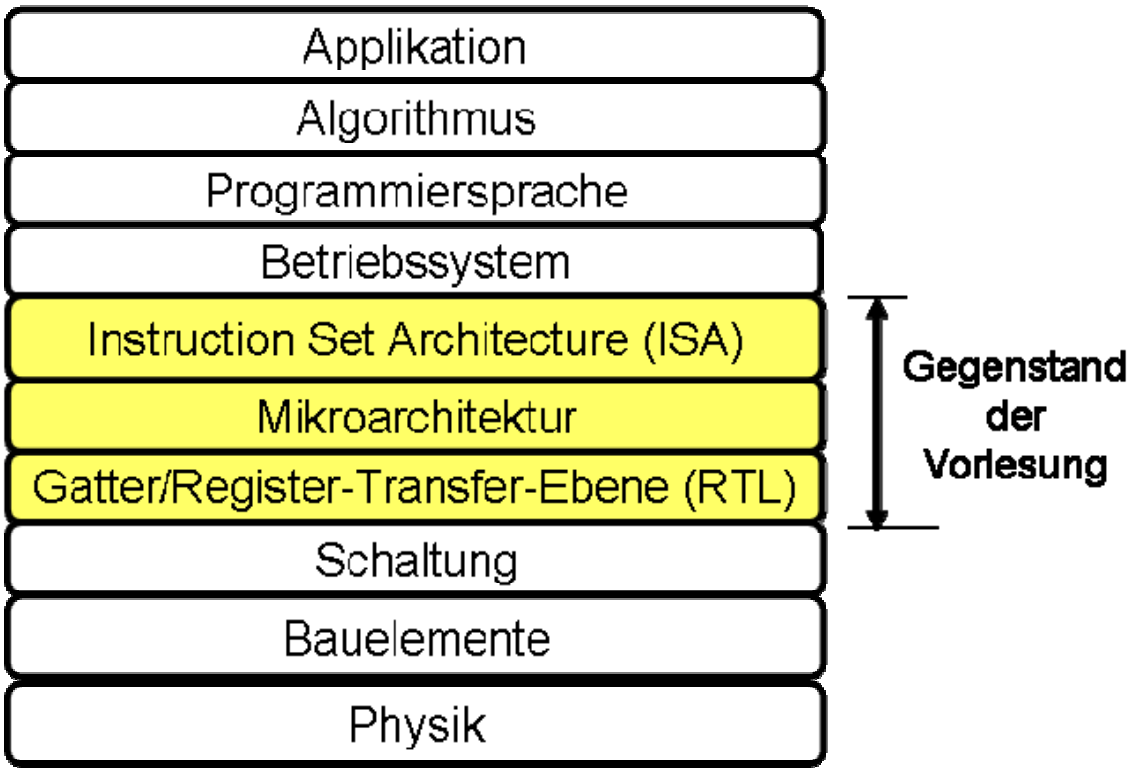
Befehlssatzarchitektur

Die (Mikro-)Prozessoren stellen einen fest vorgegebenen Befehlssatz und eine Menge von Arbeitsregistern für die Programmierung bereit. Diese Hardware-Software-Schnittstelle wird **Befehlssatzarchitektur** genannt (ISA-Architektur, Instruction Set Architecture). Sie ist so gestaltet, dass durch sie beliebige Algorithmen abgearbeitet werden können (Universalität).

Jeder Prozessortyp (-familie) hat einen eigenen Maschinenbefehlssatz. Die Programme müssen daher in dem richtigen Maschinencode (Objektcode) im Speicher liegen.

Die Maschinenbefehlssatz steht direkt in Form einer Assemblersprache für die Programmierung zur Verfügung.

Einordnung in das Schichtenmodell eines Computers



Prozessorarchitektur

Es gibt mehrere Grundprinzipien einen Prozessor mit solch einer Befehlssatz-Architektur zu realisieren.

Diese *Prozessorarchitekturen* können nach der Art der im Verarbeitungsmodell für die Ausführung der Operationen verwendeten Arbeitsregister klassifiziert werden in:

- Akkumulatormaschinen,
- Stackmaschinen,
- Registermaschinen.

Je nach dem Maschinentyp unterscheiden sich die Programmiermodelle und dadurch auch die Umsetzung von Algorithmen in Assemblersprache.

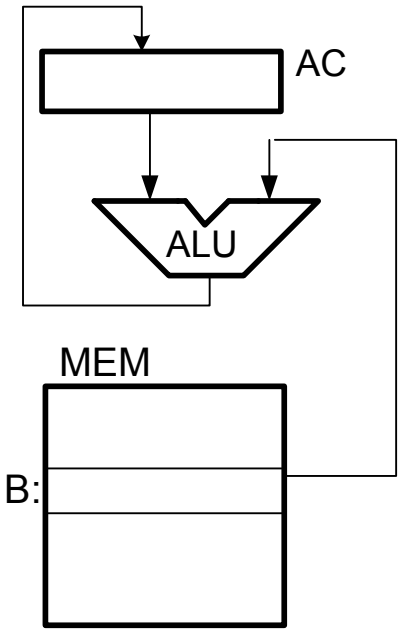
Reale Prozessoren haben (in Erweiterung zur einfachen Beispiel-CPU HAM aus Kap. 7.7) noch zusätzliche Register (wie Indexregister, Stackpointer, Statusregister) zur Unterstützung einer flexibleren und effektiveren Adressierung von Operanden bzw. bedingter Sprünge und Unterprogrammaufrufe, um (Hochsprachen-) Programme und Betriebssystemfunktionen flexibel und effizient in der Hardware zu unterstützen.

8.2 Akkumulatormaschine

8.2.1 Grundprinzip

Eine *Akkumulatormaschine* ist eine einfache Prozessorarchitektur mit einem oder zwei Arbeitsregistern *(Akkumulatoren),* die zur Durchführung aller arithmetischen und logischen Befehle sowie als Quelle und Ziel für Transferbefehle von/zum Speicher bzw. Ein-/Ausgabe dienen. D. h., die Befehle haben als impliziten Operanden den Akkumulator und ggf. als weiteren Operanden eine Speicheradresse (siehe auch Beispiel-CPU HAM aus Kap. 7.7).

Blockdiagramm einer Akkumulatormaschine



Beispiel: ADD B

AC ← AC + MEM (B)

Beliebte Architektur bei älteren Maschinen und einfachen, kostengünstigen (8-Bit-)Mikroprozessoren (insbesondere Mikrocontrollern).

Beispiele:		
	Intel 8080, 8085, 8051	(1 Akku)
	Zilog Z80	(2 Akkus)
	Motorola 6800, 68HC11*, 68HC12	(2 Akkus)
	MOS Technology 6510, 650X	(1 Akku)
	Motorola **6809**, 68HC08, 68HC11*	(1 Akku)

*: 2 8-Bit- auch als 1 16-Bit-Akkumulator nutzbar

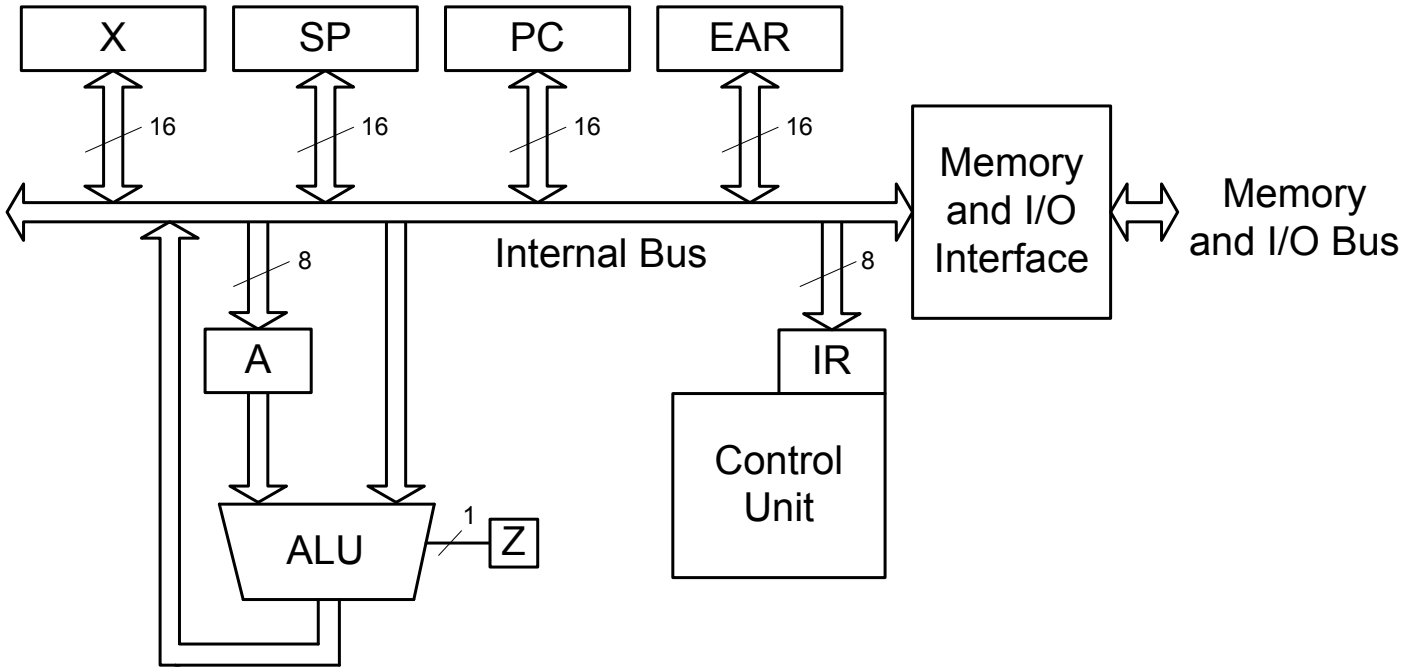
8.2.2 Organisation der hypothetischen Akkumulatormaschine H6809

Der H6809 ist eine Untermenge des Mikroprozessors *Motorola 6809*[1]. Er weist typische Kennzeichen modernen Mikroprozessoren auf.

Charakteristika:

- Wortlänge: 8 Bit (1 Byte)
- 16-Bit-Adressen, d. h. Adressraum: 2^{16} = 64 kB
- variables Befehlsformat: 1-, 2-, 3-Byte-Befehle
- Adressregister X für indirekte Adressierung
- Stapel für Unterprogramm-Rückkehradressen

Blockdiagramm des H6809



[1] nach John F. Wakerly: *Microcomputer Architecture and Programming*; John Wiley & Sons, New York, 1981

ALU *Arithmetisch-logische Einheit:*
Verknüpfung von 8-Bit-Operanden

Control Unit (Steuerwerk):
Decodierung des Befehls im IR-Register und Erzeugung der Steuersignale

Mem & I/O-Interface:
Schnittstelle zum Speicher- und E/A-Bus

IR *Instruktions-Register (Befehlsregister):*
Operationscode des auszuführenden Befehls (8 Bit)

EAR *Effektiv-Adress-Register:*
Adressteil des ausgeführten Befehls für Speicherzugriff in der Ausführungsphase (16 Bit)

PC *Program Counter (Befehlszähler):*
Zeigt auf die Adresse des nächsten auszuführenden Befehls (16 Bit)

A *Akkumulator:* Arbeitsregister für Daten (8 Bit)

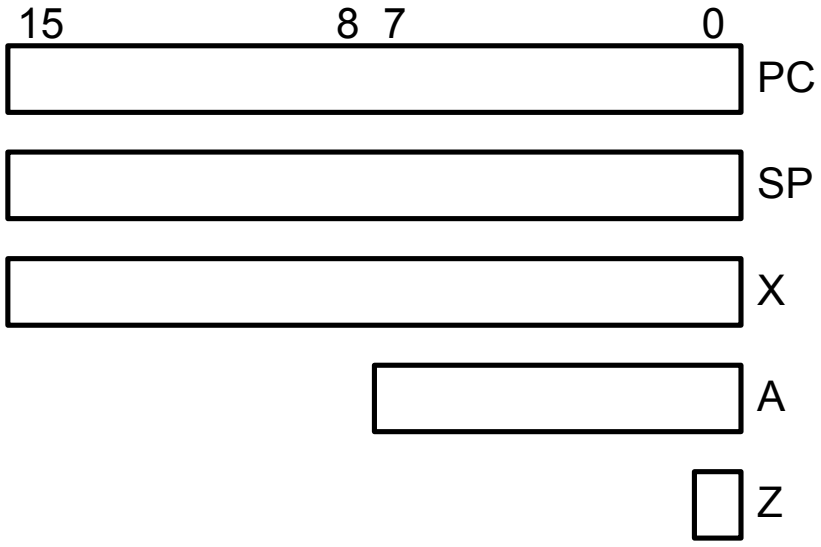
Z *Zero-Flag:*
1-Bit-Register; gesetzt, wenn Resultat = 0

X *´Index´-Register:*
Adressregister für indirekte Adressierung (16 Bit) (erlaubt auch einfache Operationen auf 16 Bit-Operanden)

SP *Stack Pointer (Stapelzeiger):*
zeigt auf die erste freie Speicherstelle vor der Spitze des Stapels (TOS) im Hauptspeicher (16 Bit)

Programmiermodell

(für Programmierer sichtbare und beeinflussbare Register, deren Inhalt dem Prozessorstatus entspricht)



Elementare Befehlszyklen

- Befehlsholphase (Fetch-Zyklus):

  IR ← MEM[PC],
  PC ← PC + 1;

- Befehlsausführungsphase (Execute-Zyklus):

  Ausführung des im IR kodierten Befehls,
  z. B. für LDA *addr*

  {Adressteil *addr* ins EAR holen}
  EAR[15..8] ← MEM[PC],
  PC ← PC +1;
  EAR[7..0] ← MEM[PC],
  PC ← PC +1;

  {Operand laden}
  A ← MEM[EAR],
  If A = 0 then Z ← 1 else Z ← 0;

8.2.3 Befehlssatz des H6809

Mnem.	Operand	Z	Length (bytes)	Opcode (hex)	Description	Number of Clock Cycles
NOP			1	12	No operation	1
CLRA		*	1	4F	Clear A	1
COMA		*	1	43	One´s complement bits of A	1
NEGA		*	1	40	Negate A (two´s complement)	1
LDA	#data	*	2	86	Load A with data	2
LDA	@X	*	1	A6	Load A with MEM[X]	2
LDA	addr	*	3	B6	Load A with MEM[addr]	4
STA	@X	*	1	A7	Store A into MEM[X]	2
STA	addr	*	3	B7	Store A into MEM[addr]	4
ADDA	#data	*	2	8B	Add data to A	2
ADDA	addr	*	3	BB	Add MEM[addr] to A	3
ANDA	#data	*	2	84	Logical AND data to A	2
ANDA	addr	*	3	B4	Logical AND MEM[addr] to A	3
CMPA	#data	*	2	81	Set Z according to A-data	2
CMPA	addr	*	3	B1	Set Z according to A-MEM[addr]	3
LDX	#addr	*	3	8E	Load X with addr	3
LDX	addr	*	3	BE	Load X with MEMW[addr]	5
STX	addr	*	3	BF	Store X into MEMW[addr]	5
CMPX	#addr	*	3	8C	Set Z according to X-addr	3
CMPX	addr	*	3	BC	Set Z according to X-MEM[addr]	5
ADDX	#addr	*	3	30	Add addr to X	3
ADDX	addr	*	3	31	Add MEMW[addr] to X	5
LDS	#addr	*	3	8F	Load SP with addr	3
BNE	offset		2	26	Branch if result is nonzero (Z=0)	2
BEQ	offset		2	27	Branch if result is zero (Z=1)	2
BRA	offset		2	20	Branch unconditionally	2
JMP	addr		3	7E	Jump to addr	3
JSR	addr		3	BD	Jump to subroutine at addr	5
RTS			1	39	Return from subroutine	3

Anmerkungen:

Mnem. = mnemonic; data = 8 Bit-Datum; addr = 16 Bit-Adresse/Datum; offset = 8 Bit signed Integer, wird bei einer Verzweigung zum PC addiert

MEM[i] Speicherbyte an Adresse i; MEMW[i] Speicherwort an Adresse i, also die Konkatenation von MEM[i] und MEM[i+1]

Die mit * gekennzeichneten Befehle beeinflussen das Z-Flag.

Die Anzahl Takte wird im Wesentlichen durch die Anzahl Speicherzugriffe bestimmt (z.B. 2-Phasen-Timing).

Befehlsgruppen

- Akkumulator-Befehle:
  CLRA … CMPA

- Adressregister-Befehle: LDX … LDS
  (X-, SP-Register)

- Programmfluss-Steuerung: BNE … RTS (NOP)
  (Sprünge etc.)

Befehlsformate des H6809

Der Op-Code des H6809 hat 8 Bit. Es sind also maximal 256 Befehle codierbar.

Der mnemonische Code (*Mnemocode*) ist in Anlehnung an die Semantik des Op-Codes gewählt.

Die Adressierungsarten sind hier als Teil des Op-Codes codiert (Kennzeichnung durch # bzw. @ im Mnemocode).

Variables Befehlsformat

Op-Code

1 Byte-Befehle, z. B. CLRA

Op-Code
Datum

2 Byte-Befehle, z. B. LDA #data

Op-Code
Addr (high)
Addr (low)

3 Byte-Befehle, z. B. LDA addr

8.2.4 Adressierungsarten des H6809

Die Adressierungsarten beschreiben die Berechnung der **effektiven Adresse** (EA) im EAR-Register nach Angaben im Op-Code.

Eine mögliche Einteilung ist nach der Anzahl der Komponenten zur Adressberechnung.

0-Komponenten-Adressierung

Implizite (inhärente) Adressierung

Spezifizierung des Operanden implizit im Op-Code

z. B. CLRA $A \leftarrow 0$

Unmittelbare Adressierung (Immediate)

Der Operand ist *unmittelbar* Bestandteil des Befehls (2. Byte). Er entspricht einer **Konstanten** in Hochsprachen.

z. B. LDA #$10

Anmerkung: Operanden und Argumente werden als Dezimalzahlen interpretiert, wenn sie nicht explizit durch ein vorgestelltes „$“ als hexadezimal gekennzeichnet sind.
(Es sind auch andere Kennzeichnungen für Hex-Zahlen gebräuchlich, z.B. „…H“ oder „0X…“.)

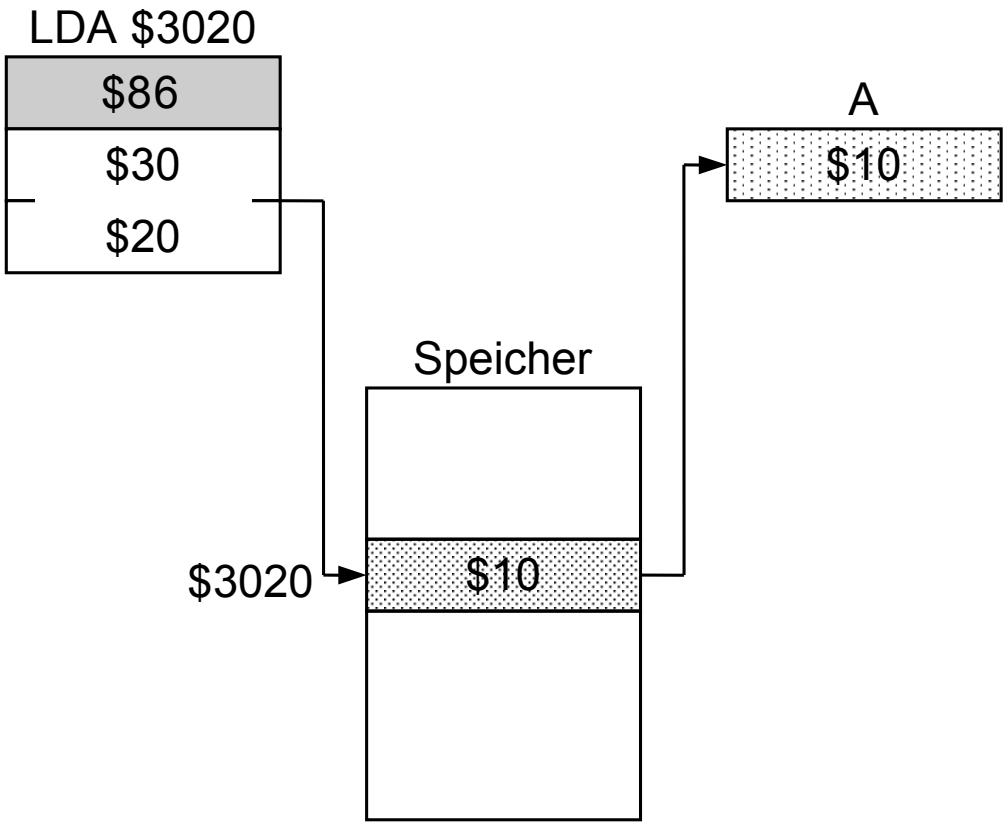
1-Komponenten-Adressierung

Ein Teil des Befehls oder ein Register bestimmen den Operanden bzw. seine Adresse.

Absolute Adressierung

Die effektive Adresse des Operanden wird direkt im Befehl angegeben (2. und 3. Byte)

z. B. LDA $3020



Dies ist die einfachste Adressierungsart für **Variablen**.

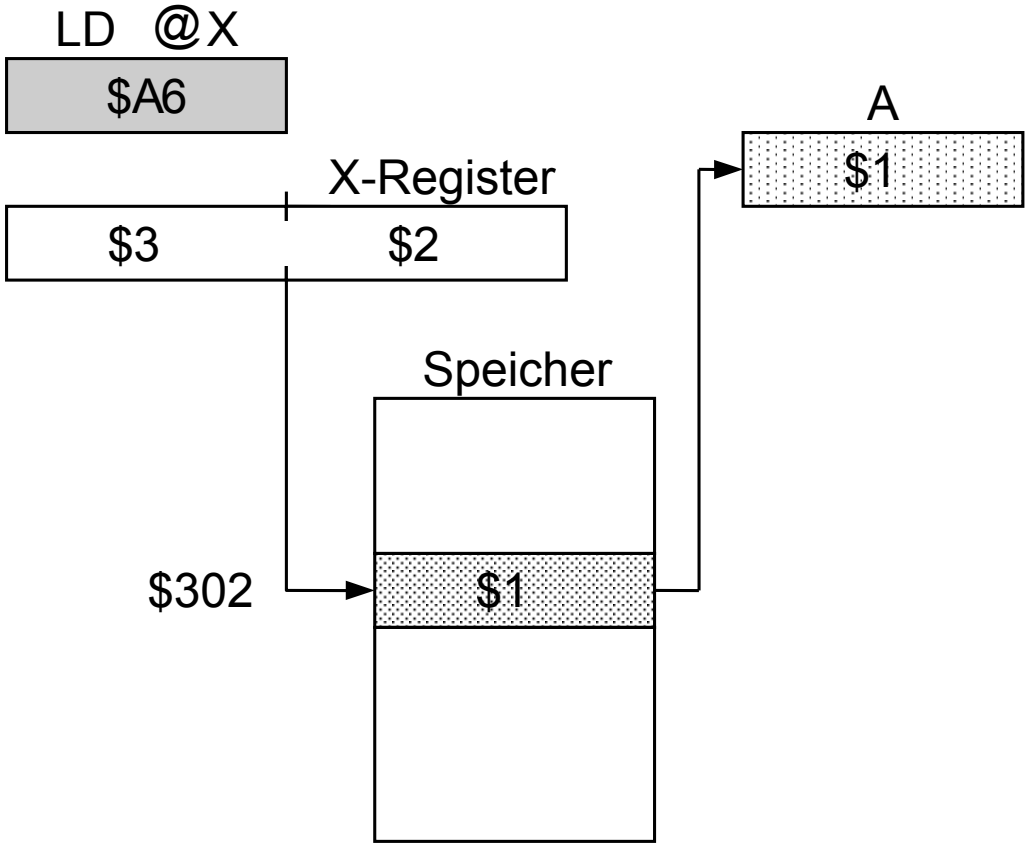
Register-indirekte Adressierung

Bei der indirekten Adressierung wird nicht die Adresse des Operanden direkt, sondern der Ort angegeben, an dem die effektive Adresse steht.
Hier steht die Adresse des Operanden im Adressregister X.

Die effektive Adresse kann also noch zur Laufzeit des Programms (hier im X-Register) geändert werden.

Das unterstützt (neben der echten indizierten Adressierung) z. B. die Implementierung komplexer Datenstrukturen wie Felder, Stapel, Schlangen …

LDA @X



Anmerkung: Das X-Register im H6809 ist kein echtes *Index*-register, weil es hier nur für die indirekte Adressierung verwendet werden kann.

Beispiel: Initialisierung eines Feldes

var Q: array[0..4] **of** Byte; …

. . .

for i := 0 **to** 4 **do** Q[i] := 0;

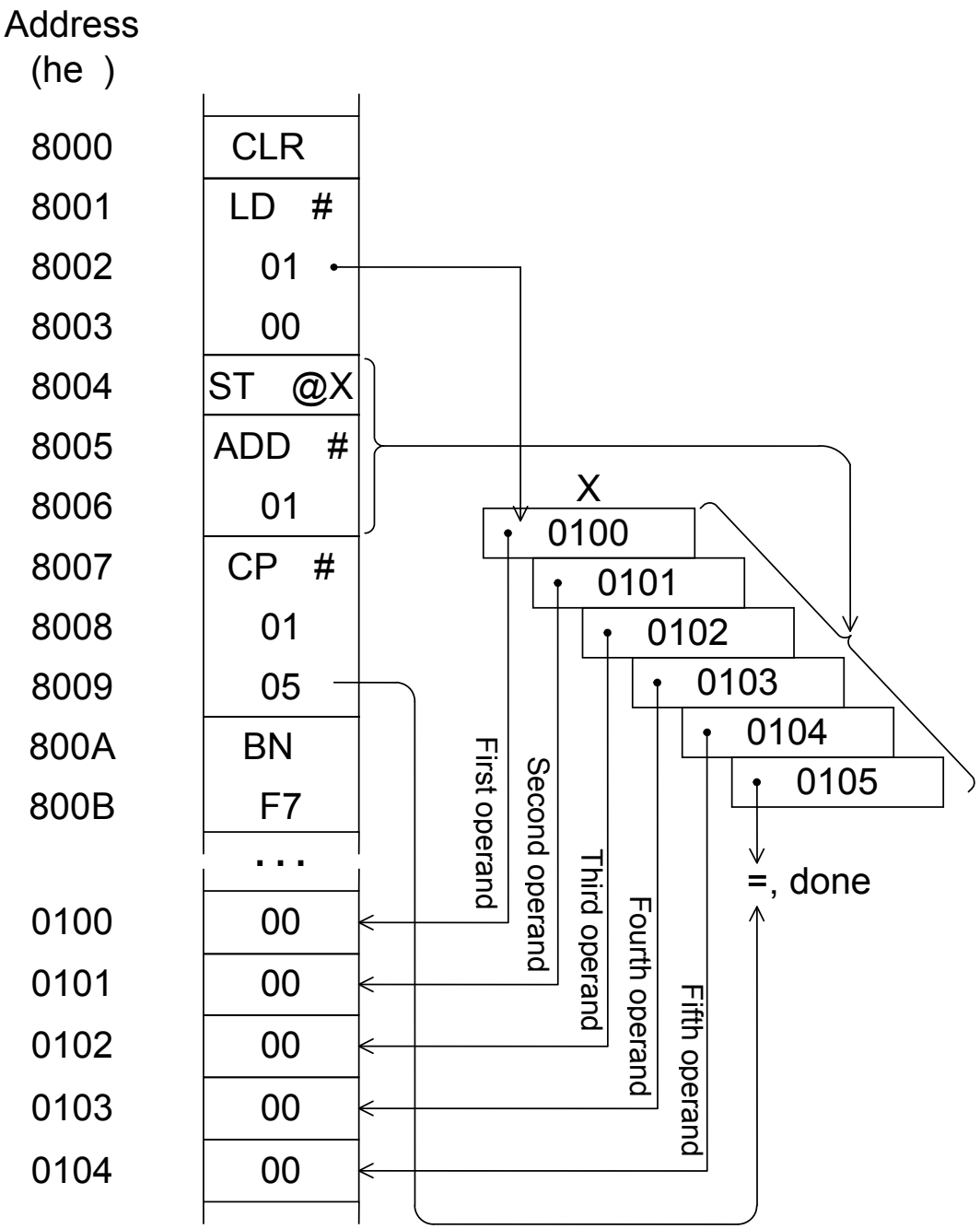
Mit absoluter Adressierung:

```
Addr  Contents  Label   Op-code  Operand   Comments
                        .ORG     $8000
8000  4F        INIT:   CLRA               ; Set components of Q to 0
8001  B7  0100          STA     Q          ; First component
8004  B7  0101          STA     Q+1        ; Second component
8007  B7  0102          STA     Q+2        ; Third component
800A  B7  0103          STA     Q+3        ; Fourth component
800D  B7  0104          STA     Q+4        ; Fifth component
8010  7E  1000          JMP     $1000      ; Return to operating syst.
...                     .ORG    $0100
0100  ??        Q       .BYTE   5          ; Reserve 5 bytes for array
...                     .EXIT   INIT
```

Mit register-indirekter Adressierung:

```
Addr  Contents  Label   Op-code  Operand   Comments
                        .ORG     $8000
8000  4F        INIT:   CLRA               ; Set components of Q to 0
8001  8E  0100          LDX     #Q         ; Address of first compon.
8004  A7        LOOP:   STA     @X         ; Set MEM[X] to 0
8005  30  0001          ADDX    #1         ; Point to next component
8008  8C  0105          CMPX    #Q+5       ; Past last component?
800B  26  F7            BNE     LOOP       ; If not, go do some more
800D  7E  1000          JMP     $1000      ; Return to operating syst.
...                     .ORG    $0100
0100  ??        Q       .BYTE   5          ; Reserve 5 bytes for array
...                     .EXIT   INIT
```

Arbeitsweise der indirekten Adressierung



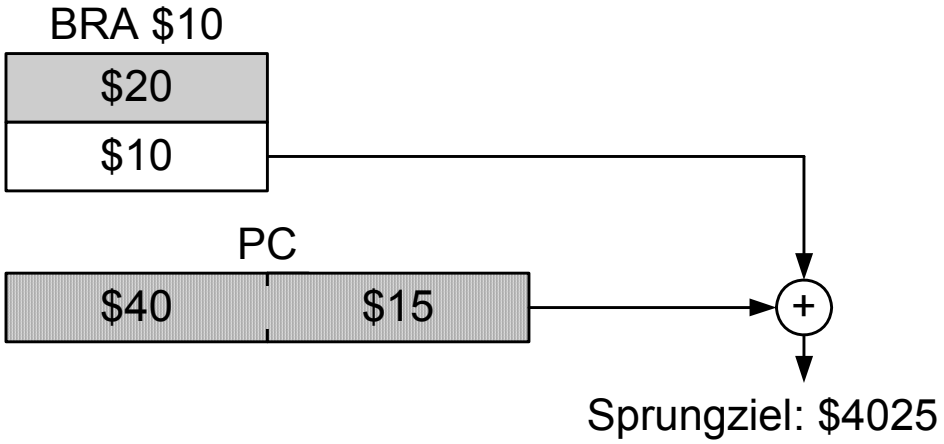
Die Anfangsadresse des Feldes ($0100) wird in das X-Register geladen und zur Laufzeit bei jeder Iteration um eins erhöht.

Der Zugriff auf die Feldvariablen (Operand) erfolgt indirekt über das X-Register (STA @X).

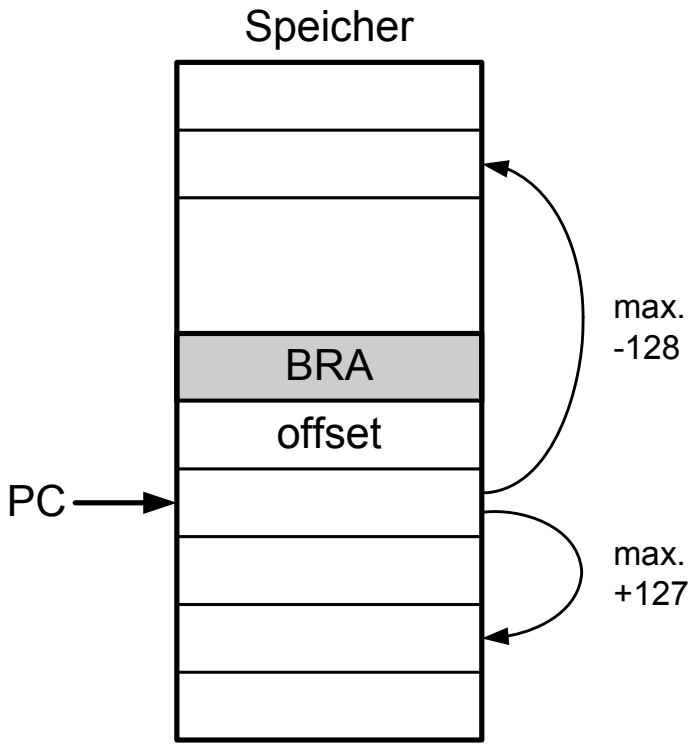
So sind auch große Felder leicht handhabbar.

PC-relative Adressierung

Die effektive Adresse (des nächsten auszuführenden Befehls) wird als Summe des aktuellen Programmzählers PC und eines Offsets (pos. oder neg. im Zweierkomplement), der Teil des Befehls ist (2. Byte), in einer eigenen Hardware für die Adressberechung gebildet.



Die PC-relative Adressierungsart wird für relative Sprünge / Verzweigungen genutzt.



Das ist günstig für verschieblichen Code, aber nur für kurze Sprungdistanzen geeignet.

8.3 Assemblerprogrammierung am Beispiel des H6809

8.3.1 Der „Assembler“

Der Assembler ist ein Programm (auf dem Entwicklungssystem), das ein Quellprogramm (Source-Code) aus der Assemblersprache mit symbolischer, mnemonischer Notation eins–zu–eins in ein binäres Maschinenprogramm übersetzt (Objektcode aus Folgen von Nullen und Einsen). D.h., eine Assembleranweisung entspricht einem Maschinenbefehl. Man spricht von einem Makroassembler, wenn der Assembler auch Makros (s.u.) unterstützt.

Moderne Assembler sind in Entwicklungsumgebungen integriert, die außerdem noch einen Editor und verschiedene Debugging-Hilfsmittel sowie einen Lader enthalten.

Der Lader lädt den Objektcode in den Speicher des Zielsystems und startet die Ausführung. Alternativ wird eine Programmierdatei z.B. für einen Flash- oder EEPROM-Speicher erzeugt (Hex-Format).

Während der Übersetzung wird für die Dokumentation und das Debugging i.d.R. auch ein List-File erzeugt, dem u.a. die Adressen der Assembleranweisungen zu entnehmen sind.

Dadurch stehen dem Entwickler von Assemblerprogrammen praktisch die gleichen Hilfsmittel wie bei der Hochsprachenentwicklung zur Verfügung. D.h., es können **und sollten** die gleichen Programmiertechniken angewendet werden:

- Strukturierung (Unterprogramme, Makros bzw. Dateien)
- aussagekräftige Label für Konstanten und Sprungziele
- Kommentare
- Debugging (z.B. **Single-Stepping, Breakpoints**)

Programmierhinweise

Bei der Programmierung in Assembler steht ein unmittelbarer Zugriff auf die Hardware(-Software-Schnittstelle) zur Verfügung. Dadurch können und müssen aber auch (im Vergleich zur Hochsprachenprogrammierung) mehr Details berücksichtigt werden.

Daher bietet sich für die Assemblerprogrammierung folgende Vorgehensweise an:

1) Programmablaufplan erstellen
2) Bestimmung der erforderlichen Konstanten und Variablen
3) Festlegung der Register- und Speicherbelegung (für Programm und Daten)
   (dabei entscheiden, ob Daten global im Speicher oder lokal beim (Unter)Programm gehalten werden)
4) Label vergeben für Daten (Variablen und Konstanten) und Programmabschnitte (mindestens Sprungziele)
5) explizites Initialisieren **aller Variablen** und des **Stack Pointers** nicht vergessen !!!
6) dann zunächst Programmfunktionalität schrittweise als Kommentar hinschreiben
7) erst danach ausprogrammieren

Tipp: Konstanten wegen Wartbarkeit explizit und möglichst global und zentral anlegen
(Bei größeren Projekten wird in der Praxis mit *Include-Dateien* gearbeitet.)

Format eines Assemblerbefehls

Label	*Op-Code*	*Operand(en)*	*Kommentar (nach ";")*
LOOP:	ADDA	addr	; Add variable at [addr] to A

Assembler-Direktiven (Pseudobefehle)

sind Anweisungen an den Assembler, die nicht in Maschinenbefehle übersetzt werden. Üblich sind Anweisungen wie:

.ORG (Programm-)Ladeadresse. Gibt an, ab welcher Adresse der nachfolgende Programmcode bzw. Datenbereich im Speicher liegen soll, nachdem das Programm geladen wurde.

.BYTE Reservierung von Speicherplatz (Anzahl Bytes) für Variablen. Die **Variable** erhält die Adresse des ersten reservierten Speicherplatzes. Der Speicherinhalt ist undefiniert.

.DB Spezifiziert 8 Bit-**Konstante**, die im Speicher abgelegt wird.

.DW Spezifiziert 16 Bit-**Konstante**, der in zwei auf einander folgenden Adressen abgelegt wird.

.DB und .DW können auch mehrere Parameter oder Ausdrücke haben (Trennung durch Komma).

.EQU Dem Bezeichner im Label-Feld wird der Wert im Operandenfeld zugewiesen. D. h. der Bezeichner kann im Programm anstelle des Wertes verwendet werden (vgl. Konstantendeklaration in Hochsprachen).

.EXIT Ende des zu assemblierenden Programmtextes, oft mit Angabe der Startadresse bzw. –marke; sonst Ende am File-Ende.

; Kommentar(zeile)

Bei manchen Assemblern:

***** in einem Ausdruck: aktuelle Adresse, die gerade assembliert wird (*Program location counter*), d. h. Festlegung zur Assemblierzeit

Beispielprogramm für den H6809

Multiplikation durch fortgesetzte Addition

Algorithmus im Pseudocode:

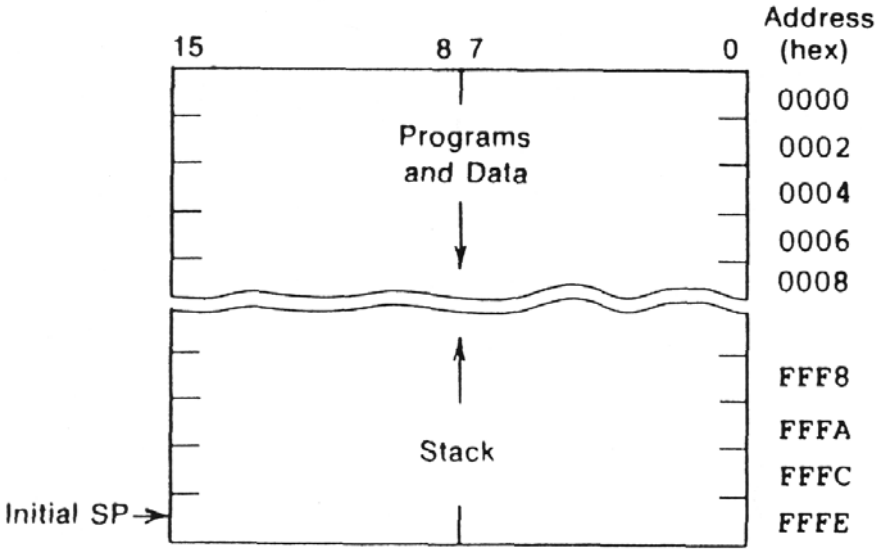
```
MCND  aus Speicherstelle holen;
MPY   aus Speicherstelle holen;
PROD  = 0;
CNT   = MPY;
WHILE CNT <> 0  DO {
    PROD = PROD +MCND;
    CNT - - ; }
```

```
Addr  Contents  Label    Op-code  Operand   Comments
                         .ORG     $2A40   ; Multiply MCND by MPY.
2A40  4F        START:   CLRA             ; Init.
2A41  B7 2C00            STA      PROD    ; Set PROD to 0
2A44  B6 2C02            LDA      MPY     ; Set CNT equal to MPY
2A47  B7 2C01            STA      CNT     ; and do loop MPY times
2A4A  B6 2C01  LOOP:     LDA      CNT     ; Done if CNT = 0
2A4D  27 10              BEQ      OUT
2A4F  8B FF              ADDA     #-1     ; Else decrement CNT
2A51  B7 2C01            STA      CNT
2A54  B6 2C00            LDA      PROD    ; Add MCND to PROD
2A57  BB 2C03            ADDA     MCND    ; only 8 bit result
2A5A  B7 2C00            STA      PROD
2A5D  20 EB              BRA      LOOP    ; Repeat the loop again
2A5F  B6 2C00  OUT:      LDA      PROD    ; Put PROD in A when done
2A62  7E 1000            JMP      $1000   ; Return to operating syst.

...                      .ORG     $2C00
2C00  ??       PROD:     .BYTE    1       ; Storage for PROD
2C01  ??       CNT:      .BYTE    1       ; Storage for CNT
2C02  05       MPY:      .DB      5       ; Multiplier value
2C03  17       MCND:     .DB      23      ; Multiplicand value
...                      .EXIT    START
```

Übliche Speicherorganisation

Aufteilung des Speichers in einen Programm- und Datenbereich, der von den niedrigeren zu den höheren Adressen wächst, sowie einen Stack (Stapel), der von einer höheren Speicheradresse zu niedrigeren Adressen hin wächst.



Im Beispiel hier: Speicher mit 16 Bit-Adressraum mit Wortorganisation von 16 Bit und Byteadressierung (*little endian*).

8.3.2 Unterprogramme (Subroutines)

Unterprogramme entsprechen PROCEDUREs, FUNCTIONs in Hochsprachen. Sie bilden eine Anweisungssequenz, die nur einmal geschrieben, aber beliebig häufig (von verschiedenen Stellen) aufgerufen werden kann. Es wird also nur einmal der entsprechende Code im Speicher abgelegt.

Probleme:

- Aufruf des Unterprogramms („*gerufenes Programm*“) erfordert Retten des **Prozessorstatus** des „*rufenden Programms*“, zumindest der Rückkehradresse
- Verlassen des Unterprogramms durch Rücksprung an gerettete Rückkehradresse im „*rufenden Programm*“
- Parameterübergaben an das Unterprogramm und zurück an das rufende Programm
- geschachtelte und ggf. rekursive/wiedereintrittsfähige Unterprogramme.

Oft ist die Schachtelungstiefe und damit auch die Anzahl an zu übergebenden Parametern zur Assemblier-(Kompilier) Zeit nicht bekannt (z.B. bei Rekursionen). Deshalb muss eine dynamische Datenstruktur verwendet werden.

Üblicherweise werden Stapel (**Stack**) für die Rückkehradresse (Return Stack) und für zu rettende Daten verwendet.

Der Stapelzeiger (**Stack Pointer**) wird vom Hauptprogramm bzw. Betriebssystem auf freien Bereich im RAM initialisiert.

Der Stack Pointer zeigt beim H6809 auf die erste freie Speicherzelle vor dem „**Top of Stack**“ (TOS). (Bei anderen Prozessoren kann er auch auf den TOS selbst zeigen.)

- Unterprogrammaufruf: **JSR *Sub***

  ***Back*:** **...** ; ab hier weiter

  Push der Adresse der nächsten Anweisung (*Back*) auf den Stapel; d.h., Schreiben des aktuellen PC auf den Stack und Dekrementieren des SP per Hardware durch die Kontrolleinheit.
  Sprung an die Unterprogramm-Einsprungadresse *Sub* durch Laden des Program Counters.

- Verlassen des Unterprogramms:

  ***Sub*:** **...** ; Unterprogr.anweisungen

  RTS

  Pop (*Back*) vom Stapel; d.h., Inkrementieren des SP und Laden des PC mit der Rückkehradresse per Hardware, dadurch Sprung an Adresse *Back*.

- Parameterübergabe an Unterprogramme:

  hier keine explizite Unterstützung (sonst meist über Register oder Stack)

Beispielprogramm mit Unterprogramm

Zählt die Anzahl Einsen in einem 16-Bit-Wort

```
Addr  Contents  Label     Opc.   Operand      Comments
                SYSRET:   .EQU   $1000      ; Operating system address
                          .ORG   $0100      ; init. small stack
0100  ??        STK:      .BYTE  7*2         ; Space for 7 return addr.
0100            STKE:     .EQU   *-1         ; Init. of address for SP

          5B29  TWORD:    .DW    $5B29       ; Test word to count 1s

                          .ORG   $2000
2000  8F  201A  MAIN:     LDS    #STKE       ; Initialize SP
2003  BE  201A            LDX    TWORD       ; Get test word
2006  BD  201C            JSR    WORDCT      ; Count number of 1s in it
2009  7E  1000            JMP    SYSRET      ; Return to operating system
201C            ; subroutine WORDCNT         ; Count the number of '1'
201C                                         ; bits in a word.
201C                                         ; Enter with word in X.
201C                                         ; Exit with count in A.
201C  BF  2032  WORDCT:   STX    CWORD       ; Save input word
201F  B6  2032            LDA    CWORD       ; Get high-order byte
2022  BD  2035            JSR    BYTECT      ; Count 1s
2025  B7  2034            STA    W1CNT       ; Save '1' count
2028  B6  2033            LDA    CWORD+1     ; Get low-order byte
202B  BD  2035            JSR    BYTECT      ; Count 1s
202E  BB  2034            ADDA   W1CNT       ; Add high-order count
2031  39                  RTS                ; Done, return
2032  ??        CWORD:    .BYTE  1*2         ; Save word being counted
2034  ??        W1CNT:    .BYTE  1           ; Save number of 1s
2035            ; subroutine BYTECNT         ; Count the number of '1'
2035                                         ; bits in a byte.
2035                                         ; Enter with byte in A.
2035                                         ; Exit with count in A.
2035  B7  2061  BYTECT:   STA    CBYTE       ; Save input byte
2038  4F                  CLRA               ; Initialize '1' count
2039  B7  2062            STA    B1CNT
203C  8E  2059            LDX    #MASKS      ; Point to 1-bit masks
203F  A6        BLOOP:    LDA    @X          ; Get next bit mask
2040  B4  2061            ANDA   CBYTE       ; Is there a '1' there?
2043  27  08              BEQ    BNO1        ; Skip if not
2045  B6  2062            LDA    B1CNT       ; Otherwise increment
2048  8B  01              ADDA   #1          ;    '1' count
204A  B7  2062            STA    B1CNT
204D  30  0001  BNO1:     ADDX   #1          ; Point to next mask
2050  8C  2061            CMPX   #MASKE      ; Past last mask?
2053  26  EA              BNE    BLOOP       ; Continue if not
2055  B6  2062            LDA    B1CNT       ; Put total count in A
2058  39                  RTS                ; Return
2059                                         ; Define 1-bit masks to test
2059                                         ; bits of byte
2059  80402010  MASKS:    .DB    $80,$40,$20,$10,$8,$4,$2,$1
205D  08040201                               ; local const. and var.
2061            MASKE:    .EQU   *           ; Address just after table
2061  ??        CBYTE:    .BYTE  1           ; Save byte being counted
2062  ??        B1CNT:    .BYTE  1           ; Save '1' count
2063                      .EXIT  MAIN
```

Hauptprogramm:

- initialisiert den Stapelzeiger und reserviert Platz für den Stapel (7 Worte)

- bereitet Testwort für Parameterübergabe im X-Register vor

- Sprung in Unterprogramm WORDCT

- Rückkehr ins Betriebssystem

Unterprogramm WORDCT:

- zählt die Anzahl von Einsen in dem als Parameter (Adresse im X-Register) übergebenen Wort

- ruft dazu das Unterprogramm BYTECT zweimal auf (Parameterübergabe jeweils im Akkumulator)

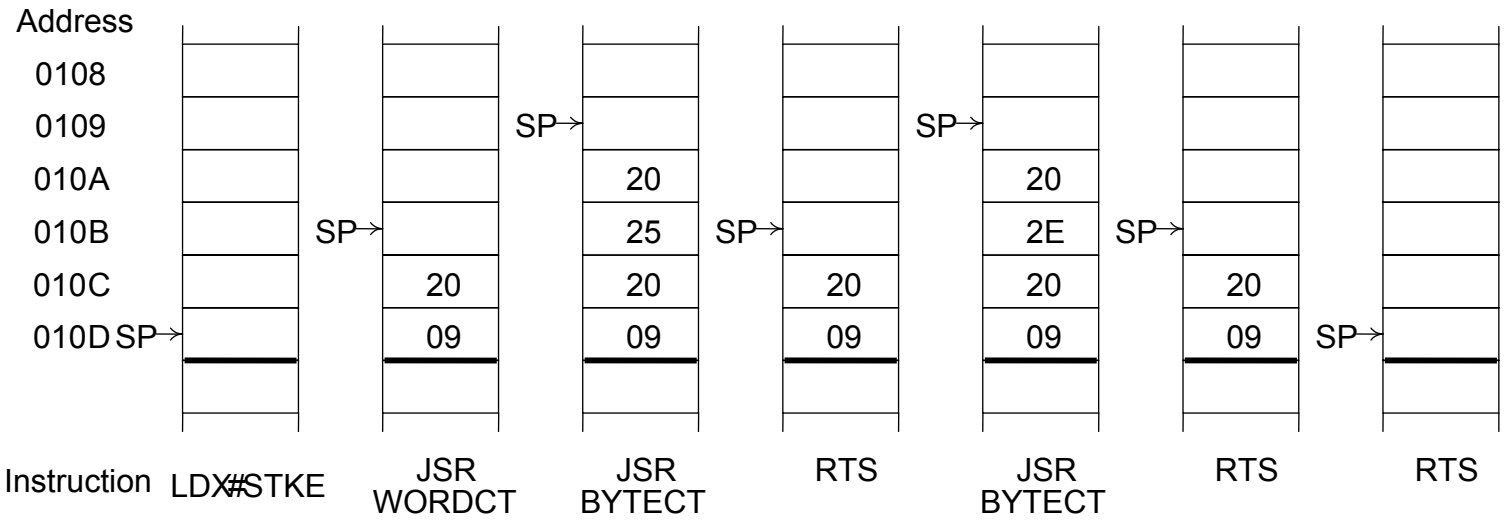
- addiert die Anzahl gezählter Einsen in beiden Bytes und kehrt ins Hauptprogramm zurück
(Parameterübergabe ebenfalls im Akkumulator)

Unterprogramm BYTECT:

- zählt die Einsen des im Akkumulator übergebenen Bytes

- gibt Ergebnis im Akkumulator an das rufende Programm zurück

Stapelinhalt im Verlauf der Programmausführung



Die Verwaltung der Unterprogramm-Rückkehradressen mittels eines Stapels als dynamische Datenstruktur ist heute Standard bei Mikroprozessoren.

Vorteile: Beliebig geschachtelte und rekursive Unterprogramme sind leicht implementierbar.

Der Stapel ist auch für andere Zwecke wie Parameterübergabe an Unterprogramme und die Auswertung arithmetischer Ausdrücke sehr gut geeignet.

Es gibt auch Prozessoren, die gar kein(e) Arbeitsregister oder Akkumulator(en) besitzen und alle Operationen nur auf dem Stapel abwickeln (Stack-Maschinen, s.u.).

8.3.3 Makros

Um Assemblerprogramme übersichtlicher zu gestalten und um für wiederkehrende Programmstücke nicht immer den gleichen Code schreiben zu müssen, werden Makros verwendet.

Format:

```
.MACRO  macroname
        …
        Anweisungsliste
        …
.ENDMACRO
```

Im Gegensatz zu Unterprogrammen wird an jeder Stelle des Makroaufrufs der entsprechende Code eingefügt. Dadurch wird der mit einem Unterprogrammaufruf verbundene (Zeit-) Aufwand eingespart, aber mehr Speicher gebraucht.

Nichtsdestotrotz können (eine beschränkte Anzahl) formale Parameter an Makros übergeben werden, um den Code für die jeweilige Verwendung zu spezialisieren.

Immer wenn der Makroname im Programm auftaucht, wird das Makro expandiert, indem an diese Stelle die Anweisungen eingetragen werden.

Die formalen Aufrufparameter sind im Makro der Reihe nach beginnend mit @0 zugreifbar.

Beim Aufruf des Makros ersetzt der Assembler (zur Assemblierzeit) die formalen Parameter durch die aktuellen Parameter.

Beispiele:

Emulation des Befehls INCA

```
.MACRO  INCA  ; increment accu
ADDA    #1    ; by simplified notation
              ; needs still two bytes of
              ; memory and a number of
              ; clock cycles
.ENDMACRO     ; end macro INCA
```

Aufruf mit: `INCA`

Emulation des Befehls SUBA #data

```
.MACRO  SUBA  ; subtract immediately
STA     temp  ; assumes a single auxiliary
              ; memory cell for all macros
LDA     #@0   ; put in actual constant
              ; parameter labeled @0
NEGA          ; build 2´s-complement
ADDA    temp  ; perform subtraction
.ENDMACRO     ; end macro SUBA
```

Aufruf z.B. mit: `SUBA $12`

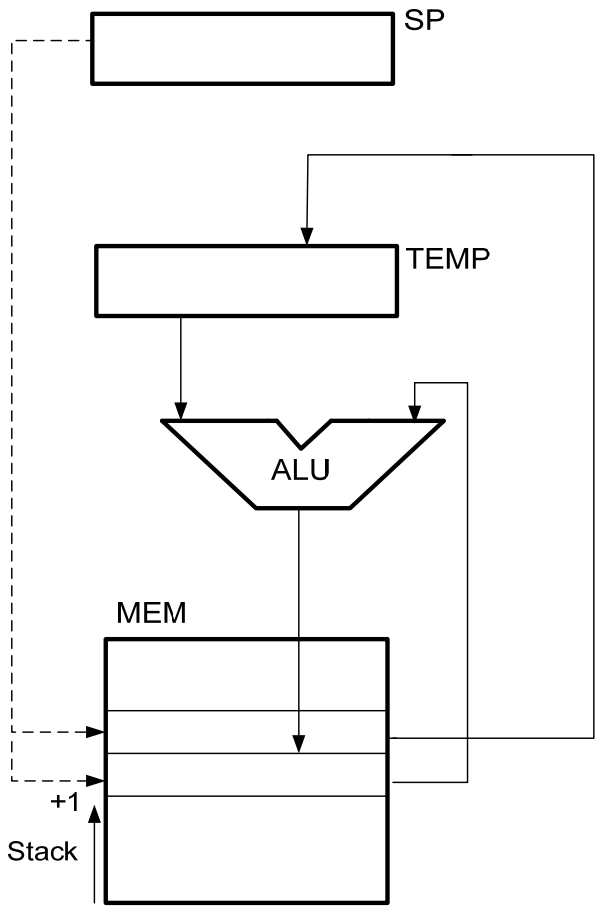
8.4 Stackmaschine

8.4.1 Grundprinzip

Die *Stackmaschine* ist eine Prozessorarchitektur, bei der alle Operationen mit einem Stapel (Stack) anstelle von Akkumulator oder Universalregistern abgewickelt werden.

Beispiele: HP300, HP3000, Bouroughs 6700, RTX 2000, Transputer

Reine Stackmaschinen sind als Prozessoren bis auf spezielle (Low Cost-)Anwendungen heute kaum noch von Bedeutung, aber Unterstützung von Stack-Mechanismen in praktisch allen modernen Prozessoren vorhanden.



Beispiel: ADD

TEMP←MEM (SP); SP←SP + 1;

MEM (SP) ←TEMP + MEM (SP);

Vergleiche:
Stapel von Büchern

Der Stapel (Stack) ist i.d.R. im Hauptspeicher als eigentlicher Arbeitsbereich angelegt. Im Prozessor sind nur Hilfsregister:

- Stackpointer SP
- temporäres Register TEMP.

Der Stack könnte im Prinzip auch alternativ im Prozessor angeordnet sein (heute nicht mehr üblich).

Evtl. enthält eine Stack-Maschine weitere Register zur Unterstützung einer flexiblen Adressierung sowie Statusregister.

Der Befehlssatz enthält die charakteristischen Operationen:

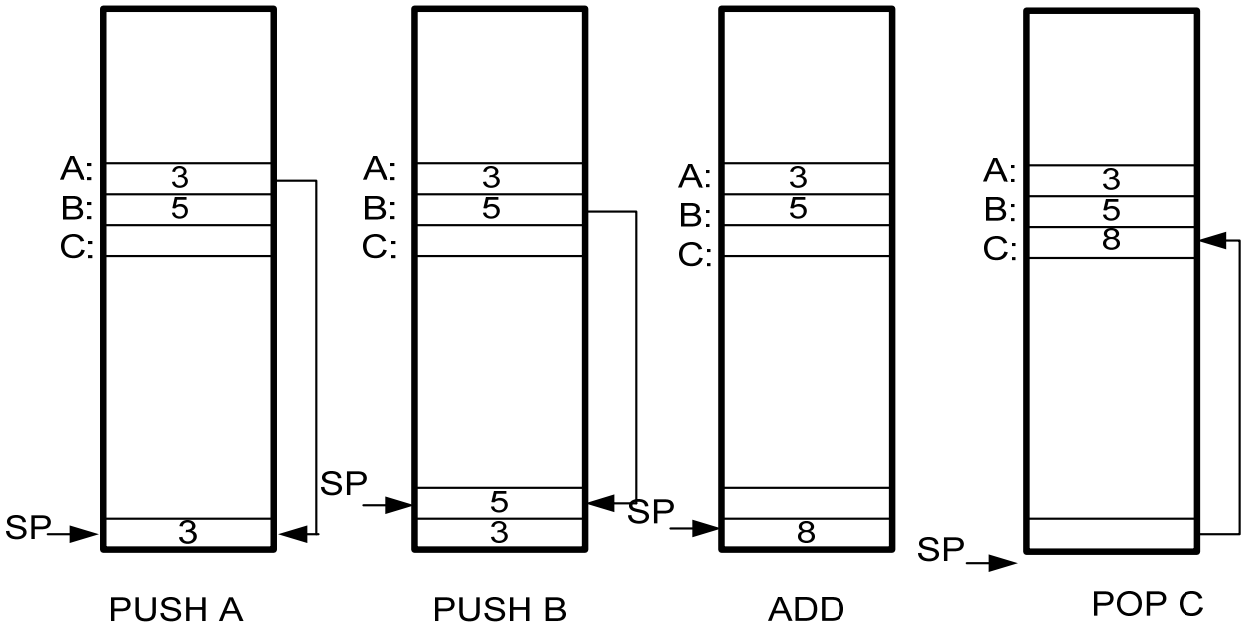
PUSH B: Legt Wort (Kopie) von Speicheradresse B auf die Spitze des Stapels.
⇒ Stapel wächst um eine Position
⇒ SP dekrementieren

POP B: „Entfernt“ Wort an Spitze des Stapels und legt es unter der Adresse B im Speicher ab
⇒ Stapel schrumpft um eine Position
⇒ SP inkrementieren

Verknüpfungsoperationen wie ADD, AND etc. spezifizieren keine Operanden explizit, sondern verknüpfen die beiden Worte an der Stapelspitze, entfernen diese und legen das Ergebnis wieder auf der Stapelspitze ab.

⇒ Stapel schrumpft um eine Position.

Beispiel: C = A + B



Der Stack wächst von höheren zu kleineren Speicheradressen), d.h. der Stackpointer wird bei PUSH erniedrigt und bei POP erhöht.

Beachte: Der Stackpointer zeigt hier auf das *oberste* ***belegte*** Element im Stack (TOS: Top of Stack).

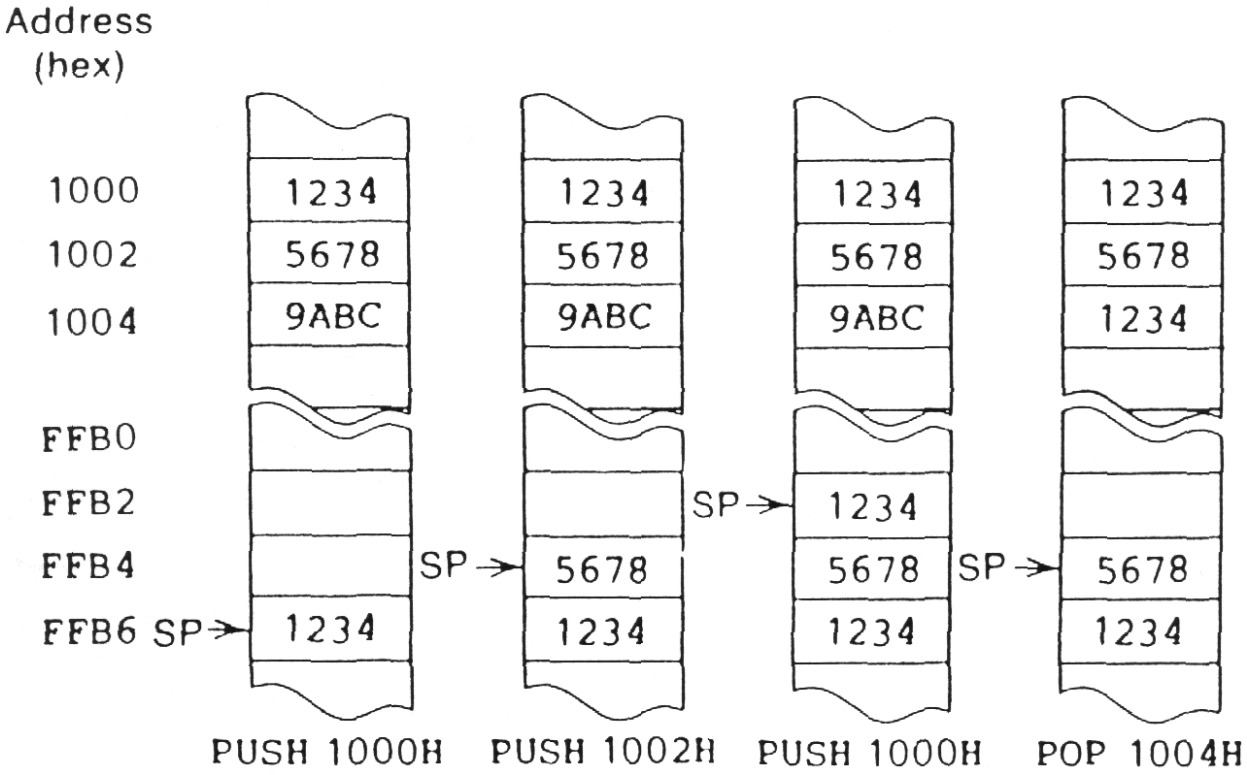
Achtung: Der Stack wird auch für Rückkehradressen bei Unterprogrammsprüngen verwendet (return address stack).

8.4.2. Befehlsverarbeitung bei Stackmaschinen[2]

PUSH/POP-Befehle

- PUSH *addr*, POP *addr*

  Das Wort an Speicheradresse *addr* kann auf den Stapel geladen (**PUSH**) bzw. vom Stapel entfernt und an der Adresse *addr* abgelegt werden (**POP**).
  Bei der hier zugrunde gelegten Wortorganisation der Operanden (16 Bit), also auch des Stacks, wird der SP immer um 2 erhöht bzw. erniedrigt. Er zeigt hier direkt auf den TOS.



[2] analog zur Stackmaschine H11 nach John F. Wakerly: Microcomputer Architecture and Programming; John Wiley & Sons, New York, 1981

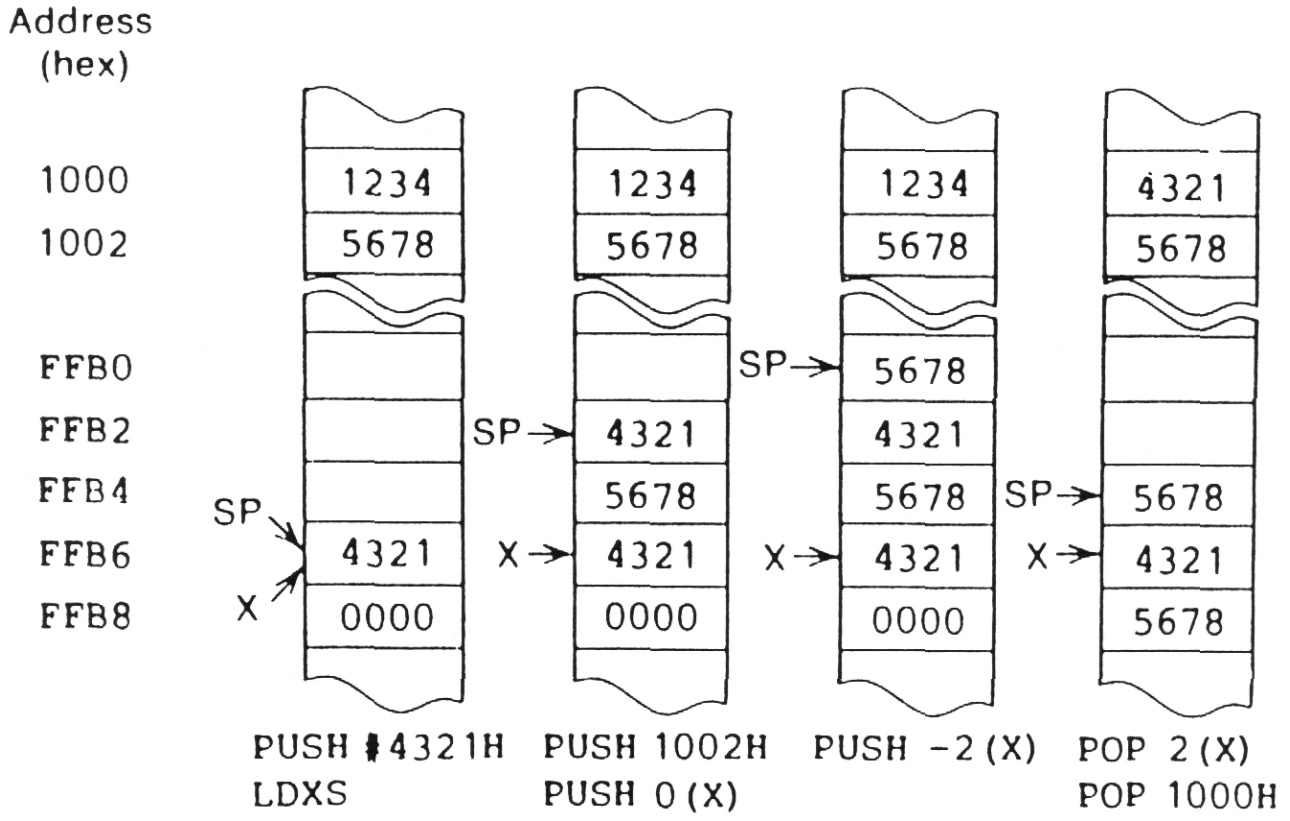
Andere PUSH/POP-Befehle

- PUSH *#data*
  Konstante *data* wird unmittelbar auf dem Stack abgelegt.

- PUSHT / PUSHS
  Spitze bzw. 2. Wort des Stacks wird auf Stack kopiert.

- PUSHX
  'Push' von X-Registerinhalt (Kopie) auf den Stack.
- PUSH offset(X), POP offset(X)

  'PUSH' von Stackwort mit Adresse in Register X plus Offset (-128 ... + 127) auf Stack, bzw. 'POP' von Wort an Stackspitze über X an die indizierte Adresse im Stack.
  Das X-Register hat hier die Funktion eines *Stack Frame Pointers.*



Typische Befehle für die indirekte Adressierung

(z. B. für Zugriff auf Felder)

- **VAL**

  Ersetzt Adresse an TOS durch zugehörigen Wert im Speicher.

  → Variablenwert holen

- **STOW**

  Speichert Wort an TOS unter Adresse im 2. Stackwort (SOS) ab und entfernt beide Worte vom Stack.

  → Variablenwert wegschrieben

SP- und X-Register-Befehle

- LD *r*, *#data* und LD r, *addr*

  Laden des Registers *r* (SP oder X) mit Konstante *data* bzw. dem Wert, der unter Adresse *addr* im Speicher steht

- LDXS

  Lädt den Registerinhalt vom SP nach X.
  (als Referenz für Zugriffe im Stackframe, z.B. für Parameterübergabe an Unterprogramme)

- ST *r*, *addr*

  Speichert Inhalt von SP- bzw. X-Register unter Adresse *addr* im Speicher ab.

- ADD *r*, *#offset*

  Addiert *offset* zum SP- bzw. X-Register.

Verzweigungs- und Sprungbefehle

Analog zur H6809.

Arithmetische und logische Befehle

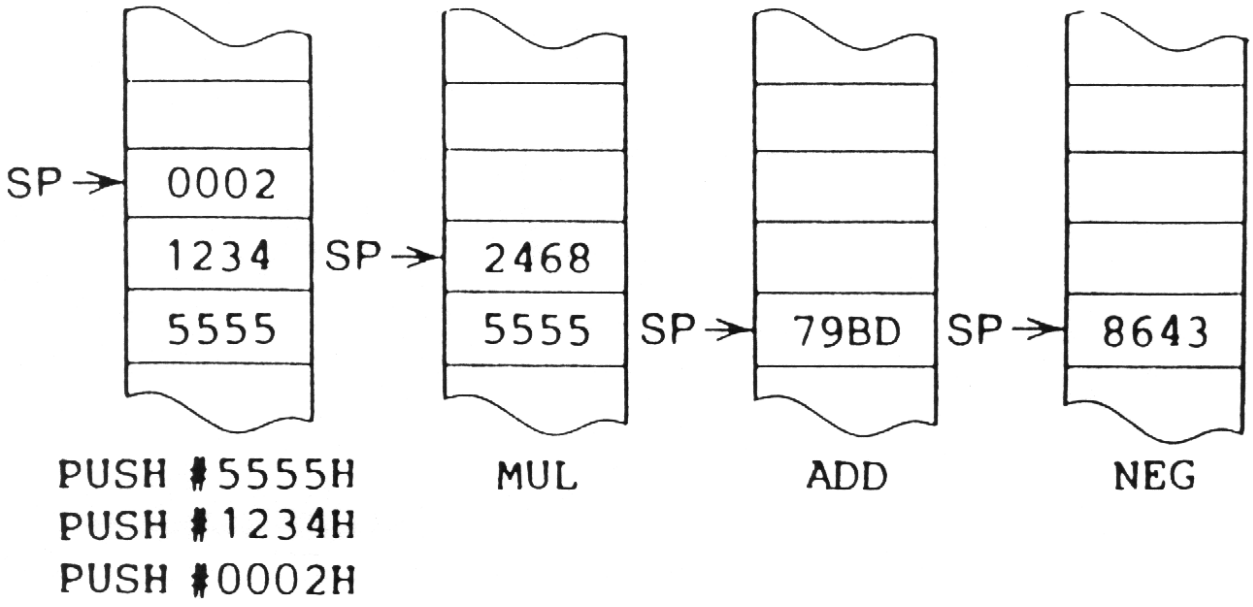
Operationen auf Daten erfolgen nur an der Stackspitze.

'Nulladress-Befehle', da keine explizite Adressangabe erforderlich.

1-Operandenbefehle:
Verändern den Wert an der Stapelspitze (TOS = Top of Stack)

2-Operandenbefehle:
Verknüpfen und entfernen die ersten beiden Elemente des Stack und legen das Resultat wieder auf dem Stack ab.

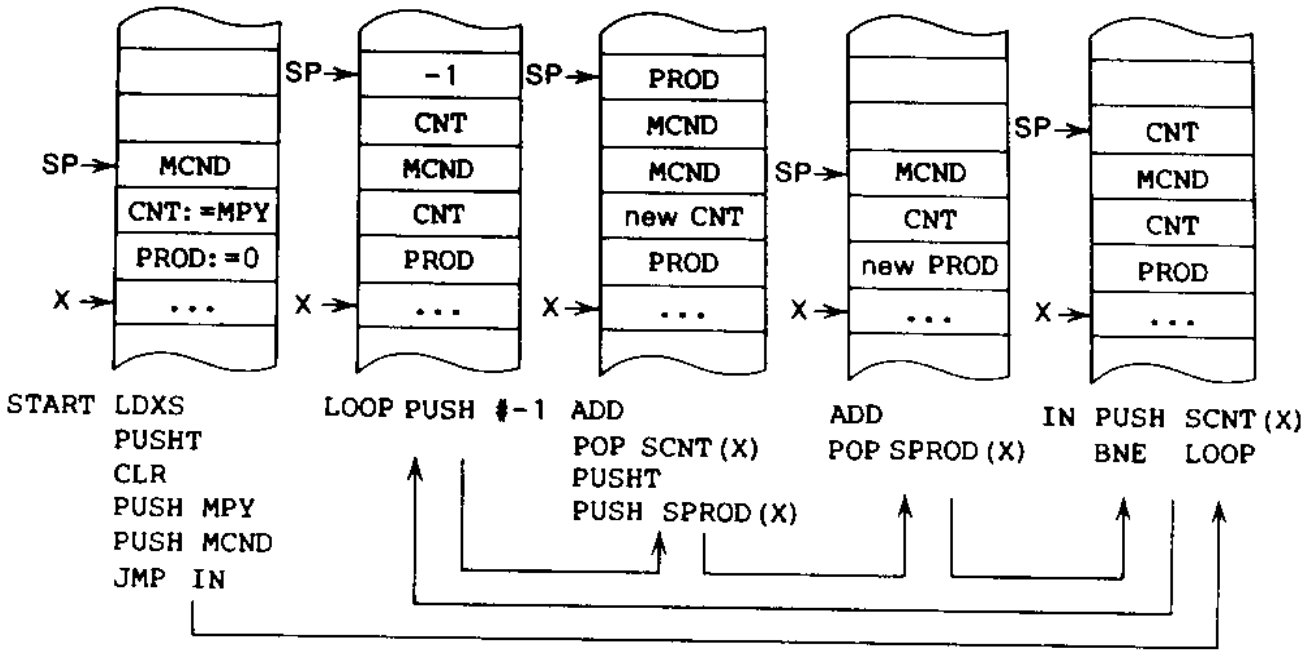
Beispiel für Arithmetik mit dem H11



8.4.3 Assemblerprogramm-Beispiel

Beispiel: Multiplikation durch wiederholte Addition (s.o.)

```
Addr  Contents  Label    Op-code  Operand     Comments
                       .ORG     $2A40       ;Multiply MCND by MPY.
2A40            SPROD:  .EQU     -2          ;Offset to PROD in stack frame
2A40            SCNT:   .EQU     -4          ;Offset to CNT
2A40            SMCND:  .EQU     -6          ;Offset to MCND
2A40            ;       Assumes SP has already been loaded on entry.
2A40  24        START:  LDXS                 ;Set up stack frame pointer.
2A41  04                PUSHT                ;Push a word onto the stack.
2A42  10                CLR                  ;Clear it (initial PROD).
2A43  01 2C00           PUSH    MPY          ;Set CNT equal to MPY.
2A46  01 2C02           PUSH    MCND         ;Make a copy of MCND in stack
2A49  50 2A58           JMP     IN           ;Do the loop MPY times.
2A4C  02 FFFF   LOOP:   PUSH    #-1          ;Decrement CNT.
2A4F  14                ADD
2A50  08 FC             POP     SCNT (X)
2A52  04                PUSHT                ;Make a copy of MCND.
2A53  03 FE             PUSH    SPROD (X)
2A55  14                ADD
2A56  08 FE             POP     SPROD (X)
2A58  03 FC     IN:     PUSH    SCNT (X)     ;Push a copy of CNT.
2A5A  40 F0             BNE     LOOP         ;Continue if not yet zero.
2A5C  29 06     DONE:   ADD     SP,#6        ;Else clean up stack.
2A5E  50 1000           JMP     $1000        ;Return to operating system,
2A61            ;                            ;PROD is at the top of stack.
2A61                    .ORG    $2C00
2C00  05        MPY:    .DW     5            ;Multiplier value.
2C02  17        MCND:   .DW     23           ;Multiplicand value.
2C04                    .EXIT   START
```



8.5 Registermaschine

8.5.1 Grundprinzip

Eine *Registermaschine* ist eine Prozessorarchitektur mit einem Satz von Universal-Registern (general purpose registers; typisch 16 - 32), die die Aufgabe von Rechen- bzw. Adressregistern übernehmen.

Vorteile:

- Mehr Daten können gleichzeitig im Prozessor gehalten werden.
  (weniger Speicherverkehr $\rightarrow$ schnellere Ausführung)
- Funktion der Register kann flexibel (zur Assemblierzeit) festgelegt werden.

Registermaschinen sind daher die Standardarchitektur bei modernen Mikroprozessoren.

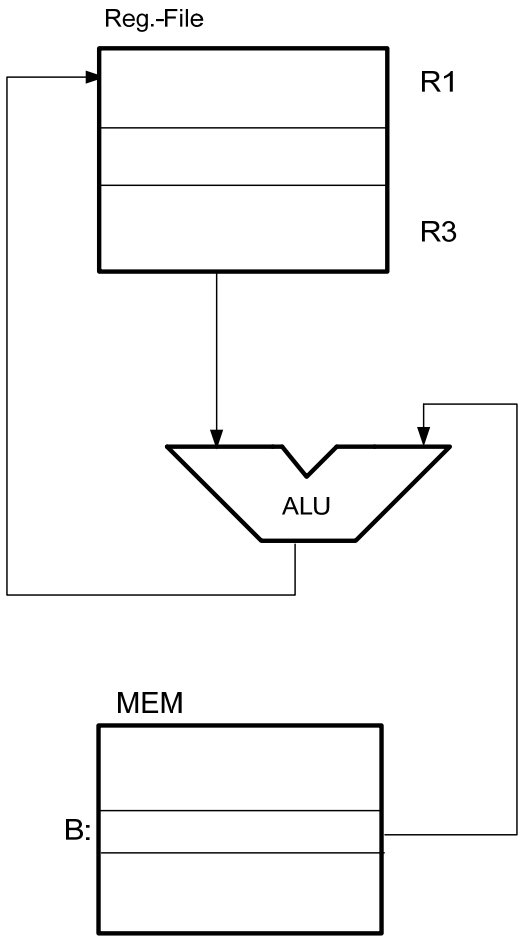
Wichtige **Varianten von Registermaschinen** sind je nach Ort der verknüpften Operanden:

reg – mem

mem – mem

reg – reg

Register-Memory-Architektur (reg-mem)



Beispiel: ADD R1, R3, B

R1←R3 + MEM (B)

Zieloperand und ein Quelloperand sind Register.

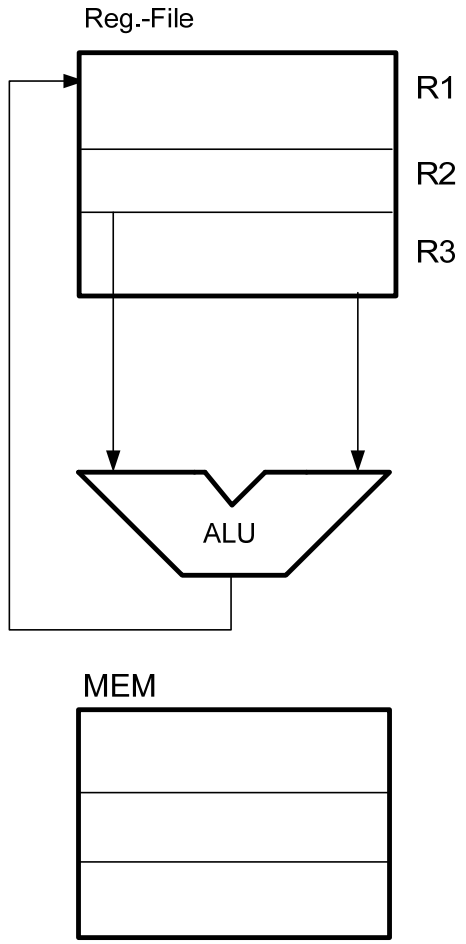
Ein zweiter Quelloperand kann ein Speicherwort sein.

Memory-Memory-Architektur (mem-mem)

Quelle und Ziel können zugleich im Speicher liegen (Speicheroperanden).

Die Memory-Memory-Architektur ist bei modernen Rechnerarchitekturen heute kaum noch gebräuchlich.

Register-Register-Architektur (Load/Store-Architektur)



Beispiel: ADD R1, R2, R3

R1 ← R2 + R3

Verknüpfungsoperationen finden *nur auf Registern* statt, also nur Registeropreationen.

Der Speicherverkehr erfolgt daher ausschließlich über LOAD/STORE-Befehle.

Vorteil: einfaches Befehlsformat fester Länge, damit einfache Code-Generierung durch Compiler und schnelle Hardware-Implementierung.

Nachteil: Mehr Befehle, d.h. längerer Code

Die LOAD/STORE-Architektur ist typisch für moderne RISC-Prozessoren (Reduced Instruction Set Computer; s. Kap. 9).

8.5.2 Organisation der hypothetischen Registermaschine HATmega16

Die hypothetische Registermaschine HATmega16[3] ist eine Untermenge des realen RISC-Mikrocontrollers Atmel ATmega16.

Sie zeigt für moderne Registermaschinen typische Grundkonzepte, konkret eine reg-reg-Maschine bzw. Load/store-Architektur.

Charakteristika

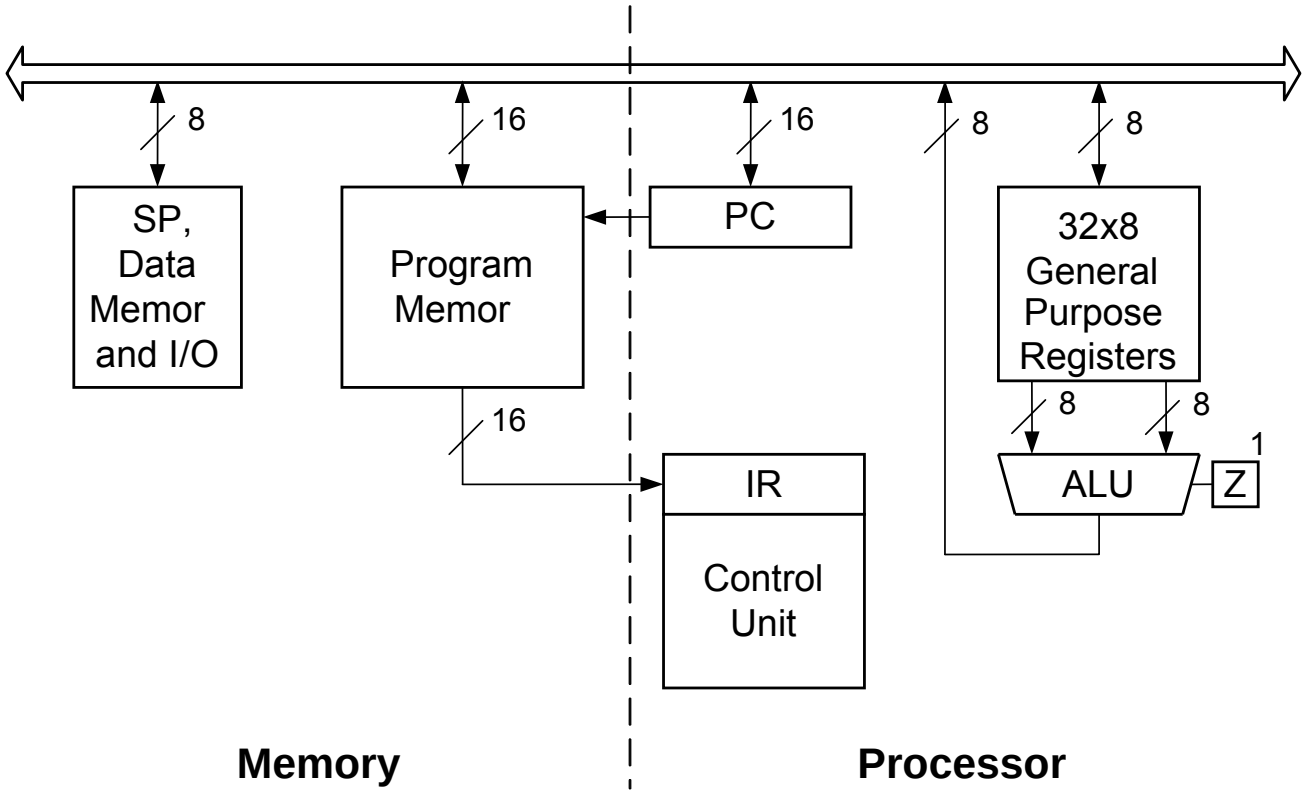
- Datenwortlänge 8 Bit
- 32 Datenregister à 8 Bit

  davon 3 Registerpaare als Adressregister à 16 Bit nutzbar
- getrennter Adressraum (und Speicher) für Befehle und Daten (*Harvard-Architektur* – eher untypisch)
- Programmspeicher: 16-Bit-Adressen, Wortlänge 16 Bit für Instruktionen
- Datenspeicher: 16-Bit-Adressen, Wortlänge 8 Bit für Daten
- RISC-Befehlssatz (Load/Store-Architektur, Befehlsformat fester Länge)
- indirekte und indizierte Adressierung
- Stapel für Unterprogramm-Rückkehradressen

Beachte: Programm- und Datenspeicher separat und mit unterschiedlichen Wortlängen

[3] nach Prof. Erik Maehle, Institut für Technische Informatik, Universität zu Lübeck

Blockdiagramm des HATmega16



32 Universalregister à 8 Bit

Spezielle Register:

PC	Befehlszähler (16 Bit)
IR	Befehlsregister (16 Bit)
Z	Zero-Flag (1 Bit)
SP	Stackpointer (16 Bit; liegt im Datenspeicher und zeigt auf erste freie Speicherstelle)

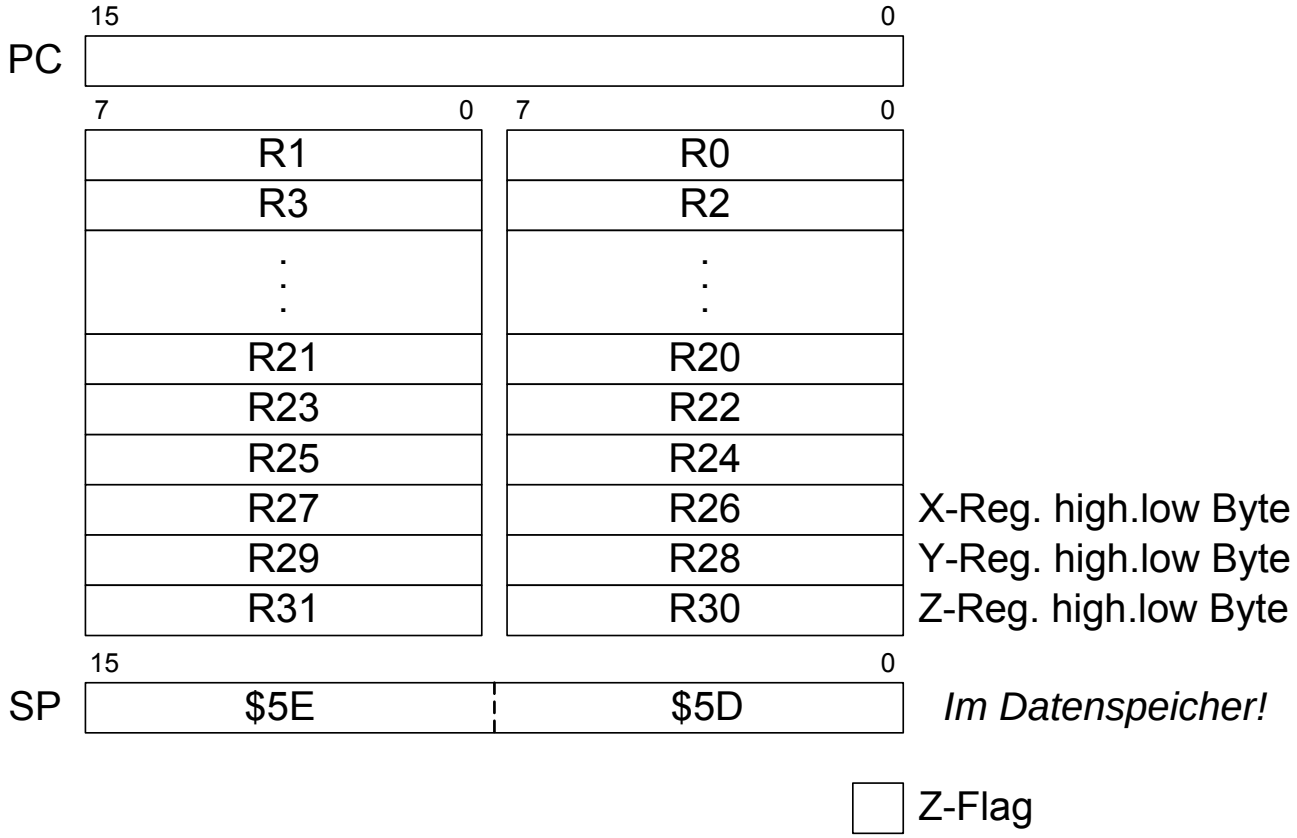
ALU (8 Bit-Rechenwerk)

Control Unit (Steuerwerk)

Program Memory (Programmspeicher)

Data Memory and I/O (Datenspeicher und Ein-/Ausgabeschnittstellen im selben Adressbereich)

Programmiermodell des HATmega16



Byte-Operationen mit allen Registern R0 bis R31

Alle Register können die Rolle eines *Akkumulators* übernehmen (Datenregister).

Sonderrolle der Register R26 bis R31:

R26.R27	X-Pointer (16 Bit)
R28.R29	Y-Pointer (16 Bit)
R30.R31	Z-Pointer (16 Bit)

Sie können auch als Doppelregister auch die Rolle von Adressregistern (Zeiger-, Indexregistern) übernehmen.

Das Z-Flag wird in Abhängigkeit vom Ergebnis der letzten Datenmanipulations-Operation gesetzt, egal mit welchem Universal-Register.

Der Stackpointer (16 Bit) ist im *Datenspeicher* untergebracht und kann auch entsprechend angesprochen werden:

$5E	SPH	Stackpointer High Byte
$5D	SPL	Stackpointer Low Byte

8.5.3 Befehlssatz des HATmega16

Mnemo.	Operand.	Description	Operation	Flags	Clock
Arithmetic and Logic Instructions					
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z	1
AND	Rd, Rr	Logical AND	Rd ← Rd ∧ Rr	Z	1
COM	Rd	One's Complement	Rd ← $FF - Rd	Z	1
NEG	Rd	Two's Complement	Rd ← $00 - Rd	Z	1
INC	Rd	Increment	Rd ← Rd + 1	Z	1
DEC	Rd	Decrement	Rd ← Rd - 1	Z	1
CP	Rd, Rr	Compare	Rd - Rr	Z	1
CPI	Rd, K	Compare with Immediate	Rd - K	Z	1
TST	Rd	Test for Zero	Rd ← Rd ∧ Rd	Z	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z	1
Branch Instructions					
RJMP	k	Relative Jump	PC ← PC + k + 1	-	2
JMP	k	Jump	PC ← k	-	3
CALL	k	Call Subroutine	PC ← k	-	4
RET		Subroutine Return	PC ← STACK	-	4
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	-	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	-	1/2
Data Transfer Instructions					
MOV	Rd, Rr	Copy Register	Rd ← Rr	-	1
LDI	Rd, K	Load Immediate	Rd ← K	-	1
LDS	Rd, k	Load Direct from Data Space	Rd ← (k)	-	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	-	2
LD	Rd, Z+	Load Indirect and Post-Increment	Rd ← (Z), Z ← Z+1	-	2
LD	Rd, -Z	Load Indirect and Pre-Decrement	Z ← Z - 1, Rd ← (Z)	-	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	-	2
STS	k, Rr	Store Direct to Data Space	(k) ← Rd	-	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	-	2
ST	Z+, Rr	Store Indirect and Post-Increment	(Z) ← Rr, Z ← Z + 1	-	2
ST	-Z, Rr	Store Indirect and Pre-Decrement	Z ← Z - 1, (Z) ← Rr	-	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	-	2
PUSH	Rr	Push Register on Stack	STACK ← Rr	-	2
POP	Rd	Pop Register from Stack	Rd ← STACK	-	2
MCU Control Instructions					
NOP		No Operation		-	1

Rd: Destination (and Source) Register in the Register File
Rr: Source Register in the Register File
K: Constant data
k: Constant address
X,Y,Z: Indirect Address Register
(X=R27:R26, Y=R29:R28 and Z=R31:R30)
q: Constant Displacement for direct addressing (6-bit)

Arithmetische und logische Befehle haben nur Registeroperanden, keine Speicheroperanden (reg-reg-Architektur).

Der Zugriff auf den Datenspeicher erfolgt allein über Load/Store-Befehle mit diversen Adressierungsarten (Load/Store-Architektur).

Keine eigenen Befehle für Adressregister X, Y, Z oder SP.

Adressierungsarten für X- und Y-Register wie für Z-Register, aber indizierte Adressierung (indirekt mit Displacement) nicht für X-Register.

Unbedingte Sprungbefehle absolut (JMP) oder relativ (RJMP), bedingte Sprungbefehle nur relativ (BREQ, BRNE).

Compare/Test-Befehle zum Setzen des Z-Flags vor bedingten Sprüngen (TST, CP, CPI).

Achtung: Nur arithm./log. Befehle und Compare-Befehle beeinflussen das Z-Flag, nicht z. B. Ladebefehle.

Unterprogrammaufruf/-rückkehr mittels Rückkehradresse auf dem Stack (CALL, RET).

Überlappung von Befehlshol- und Ausführungsphase, d.h. während ein Befehl ausgeführt wird (*execute*), wird schon der nächste Befehl geholt (*fetch*).

Befehlsausführungszeiten:

- einfache Ein-Wortbefehle: 1 Takt (z.B. ADD, CLR)
- Load/Store-Befehle: 2 Takte (z.B. LDS, STS)
- Branch-Befehle: 1 Takt bzw. 2 Takte (no/yes), (z. B. BREQ)
- Unterprogramm-Befehle: 4 Takte (CALL, RET)

Befehlstypen

- *0-Operandenbefehle*

  Beispiel: **NOP**

- *1-Operandenbefehle* mit Registeroperand

  Beispiel: **CLR R1** ; Lösche R1

- *2-Operandenbefehle*

  Operand1: Register
  Operand2: Register, Speicheroperand oder unmittelbarer Operand

  Beispiele:

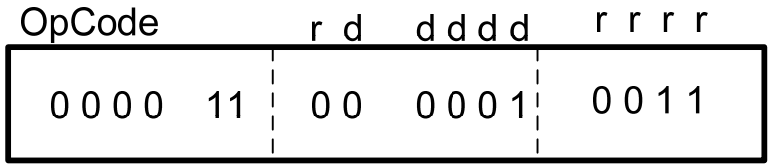
  ADD R1, R3 ; Addiere R3 zu R1
  LDS R1, addr ; Lade MEM(addr) nach R1
  LDI R1, 15 ; Lade Wert 15 in R1

  Load-Store-Befehle sind „Eineinhalbadress-Befehle“ (vgl. „Einadress-Befehle“ bei Akkumulatormaschine)

Befehlsformate (feste Länge)

- 1-Wort-Befehle (16 Bit)

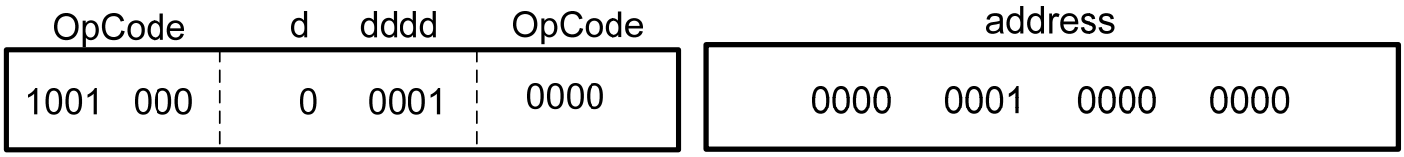
  Beispiel: **ADD R1, R3**



  Rd: Destination and Source Register
  Rr: Source Register

- 2-Wort-Befehle (32 Bit), mit Speicheradresse (für das Datum) im 2. Wort

  Beispiel: **LDS R1, $100**



8.5.4 Assemblerprogrammierung des HATmega16

Format eines Assemblerbefehls

Label	*Op-Code*	*Operand(s)*	*Comment*
LOOP:	ADD	R16, R19	; Add R16 to R19

Assembler-Direktiven (vollständiger als oben)

.DEF	Symbol = Reg	definiert einen symbolischen Namen für ein Register
.ORG	*adr*	setzt Ladeadresse für Programm- oder Datenbereich
.CSEG		leitet Codesegment ein
.DSEG		leitet Datensegment ein
.BYTE	*n*	reserviert *n* Byte (nur Datensegment)
.DW .DB	*w* *b*	definiert Wort- bzw. Byte-Konstante mit Wert *w* bzw. *b* (nur Codesegment). Auch Liste von Werten möglich.
.EQU	Symbol = expr.	setzt ein Symbol gleich einem Ausdruck. Das Symbol kann dann anstelle des Ausdrucks verwendet werden (Wert nicht änderbar)
.SET	label = expr.	weist einem Label einen Wert zu, der immer anstelle des Codes eingesetzt wird (Wert änderbar)
.MACRO	*macroname*	legt ein Makro an
.ENDMACRO	oder	
.ENDM		schließt ein Makro ab
.INCLUDE	“file“	fügt File ein (z. B. Symboldefinitionen)

Beispielprogramm: **Multiplikation durch wiederholte Addition** (vgl. oben)

Algorithmus im Pseudocode:

```
MCND = 23;
MPY  = 5;
PROD = 0;
CNT  = MPY;
WHILE CNT <> 0  DO {
  PROD = PROD +MCND;
  CNT - - ;
}
```

```
; REGISTER DEFINITIONS
.def    PROD =  R16     ;  Product
.def    CNT  =  R17     ;  Iteration Counter
.def    MPY  =  R18     ;  Multiplier
.def    MCND =  R19     ;  Multiplicand
.def    TEMP =  R20     ;  Temporary Variable

.CSEG   ;       CODE SEGMENT
.ORG $0000                      ;  Starting Adress
INIT:   LDI   MCND, 23          ;  Set Multiplicand
        LDI   MPY, 5            ;  Set Multiplier
START:  CLR   PROD              ;  Clear Product
        MOV   CNT, MPY          ;  Init Counter to MPY
        TST   CNT             ;   Counter=0?
        BREQ  END               ;  If yes, branch to END
LOOP:   ADD   PROD, MCND ; Else add MCND to PROD
        DEC   CNT               ;  Decrement Counter
        BRNE  LOOP              ;  If not zero, loop again
END:    RJMP  INIT              ;  Endless loop when done
```

8.5.5 Adressierungsarten des HATmega16

Um Operanden möglichst effektiv und flexibel adressieren zu können, verfügen moderne Prozessoren über verschiedene Adressierungsarten. Die Berechnung der effektiven Adresse erfolgt dazu zur Laufzeit im Prozessor in eigener Hardware.

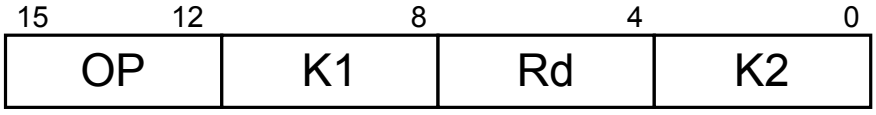
Die Klassifizierung der Adressierungsarten erfolgt je nach der Angabe der *effektiven Adresse* des Operanden nach:

- direkt, indirekt, indiziert
- Anzahl der Komponenten

0-Komponenten-Adressierung

Unmittelbare Adressierung (Immediate)

8-Bit Konstanten K = K1.K2



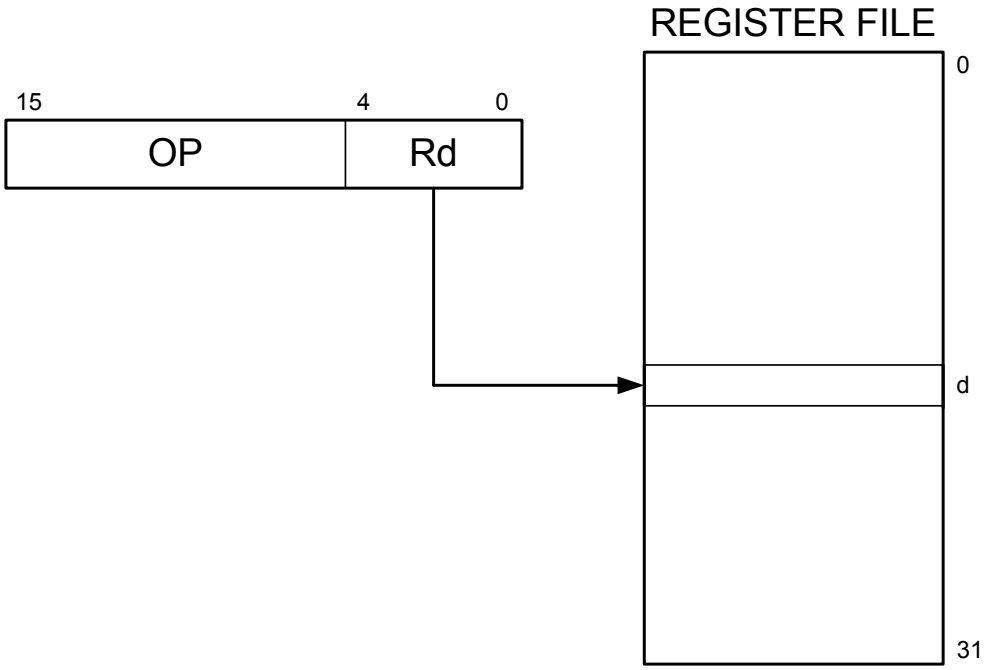
Rd ← K

LDI R20, $F6 ; R20 ← $F6

Achtung: Unmittelbare Adressierung nur für R16 bis R31 erlaubt (s. 4-Bit-Feld für Rd)!

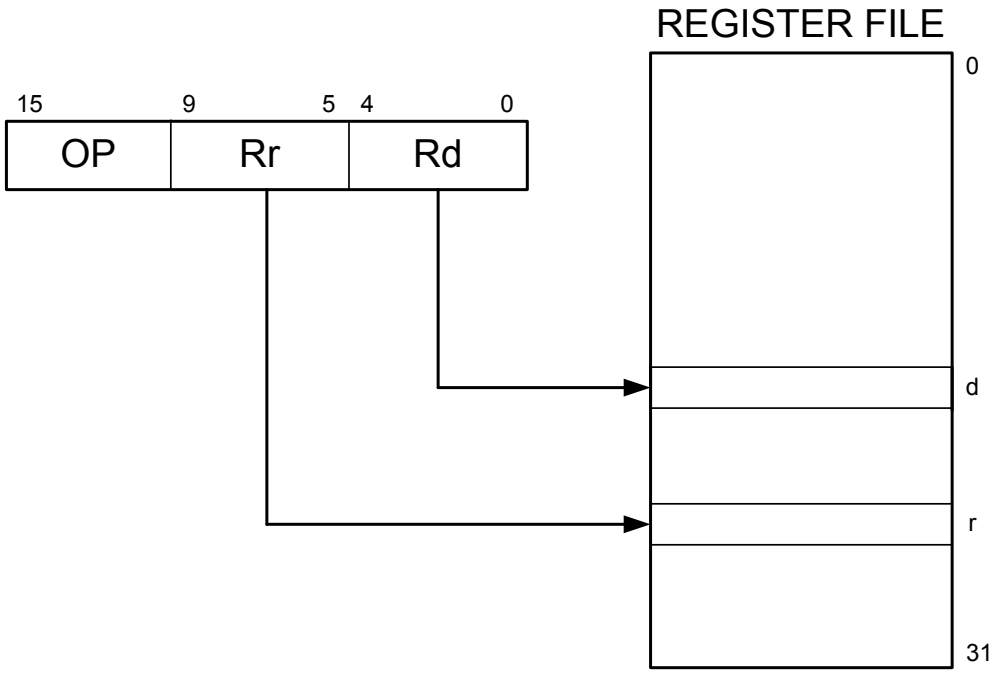
1-Komponenten-Adressierung

Direkte Einzel-Register-Adressierung



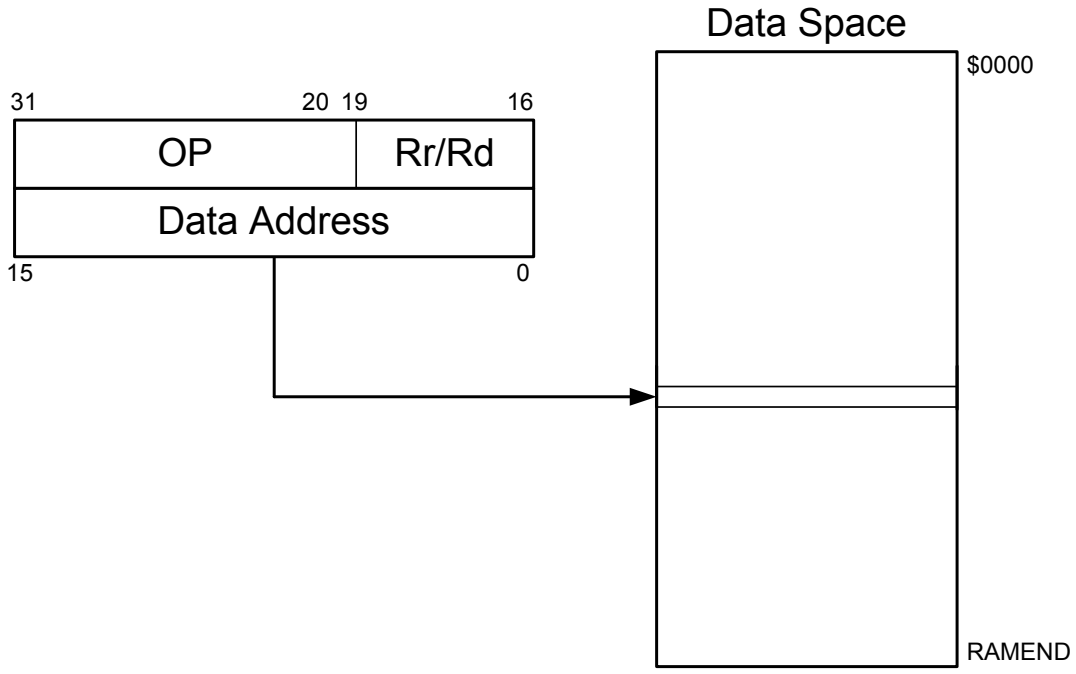
COM R12 ; R12 ← $\overline{\text{R12}}$
INC R16 ; R16 ← R16 + 1

Direkte Register-Adressierung, 2 Register



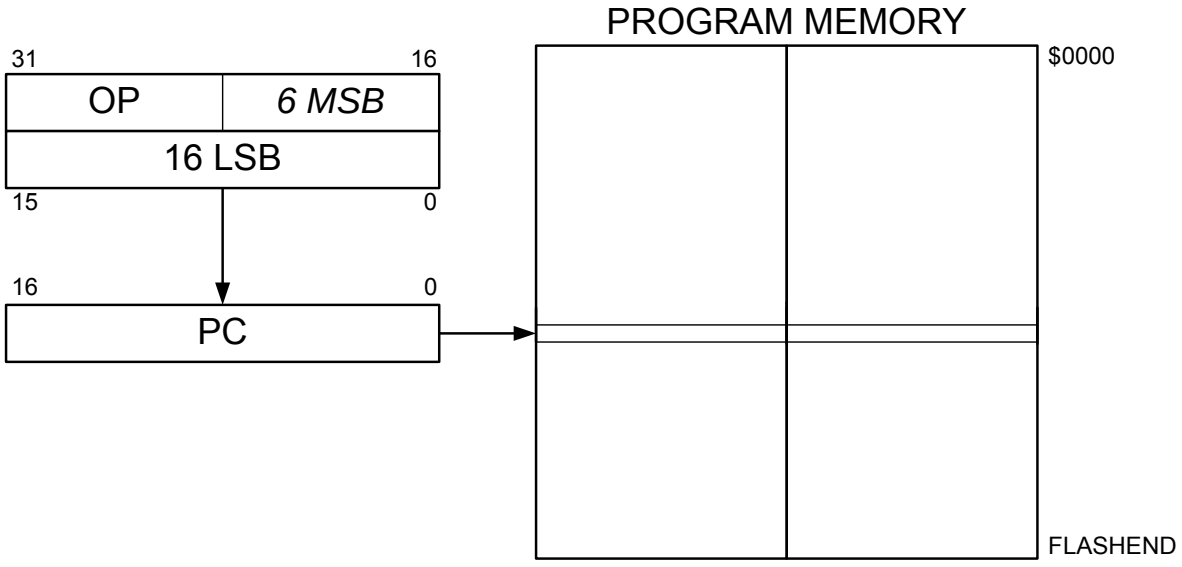
AND R16, R15 ; R16 ← R16 ∧ R15

Direkte (Absolute) Daten-Adressierung (2-Wort-Befehl!)



LDS R20, $100 ; R20 ← MEM($100)

Direkte Programmspeicher-Adressierung (2-Wort-Befehl!)

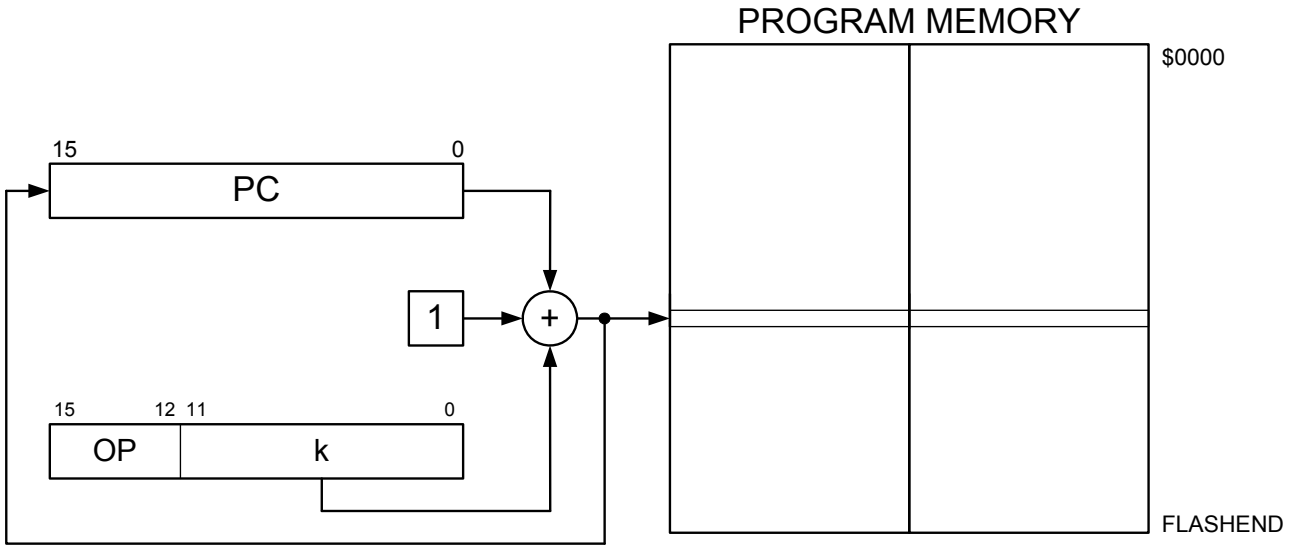


JMP $100 ; PC ← $100

CALL $500 ; STACK ← PC + 2
; SP ← SP - 2
; (2 Byte Rückkehradresse)
; PC ← $500

2-Komponenten-Adressierung

Relative Programmspeicher-Adressierung (12-Bit-Offset)

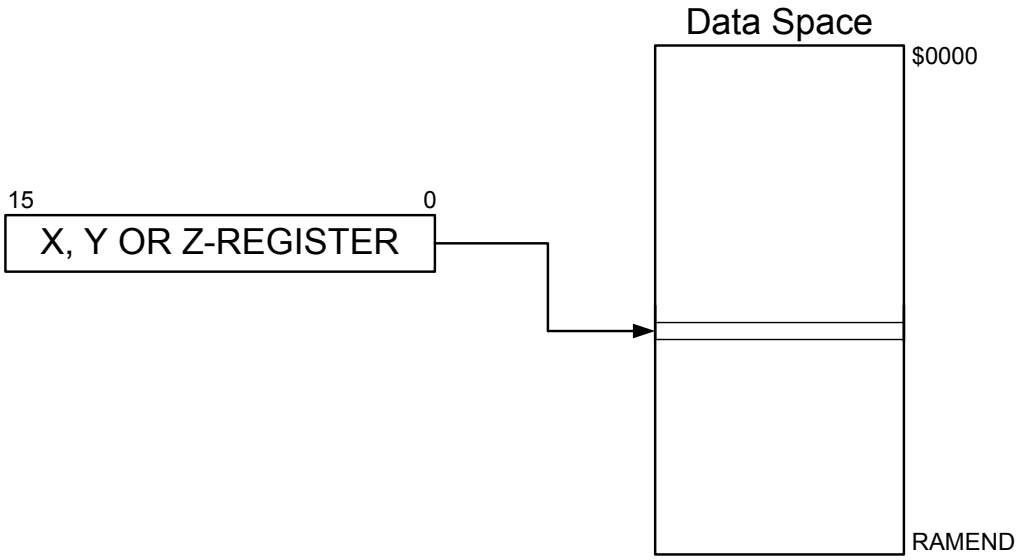


BREQ $10 ;if Z = 1 then PC ← PC + $10 + 1

Die Berechnung der effektiven Adresse erfolgt zur Laufzeit in eigener Hardware.

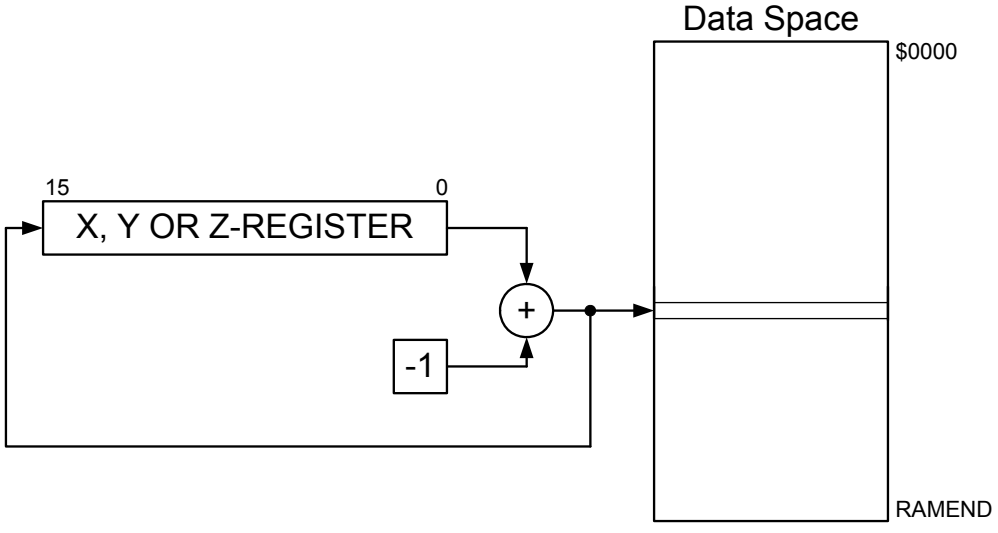
Indirekte und indizierte Adressierung

Register-indirekte Adressierung



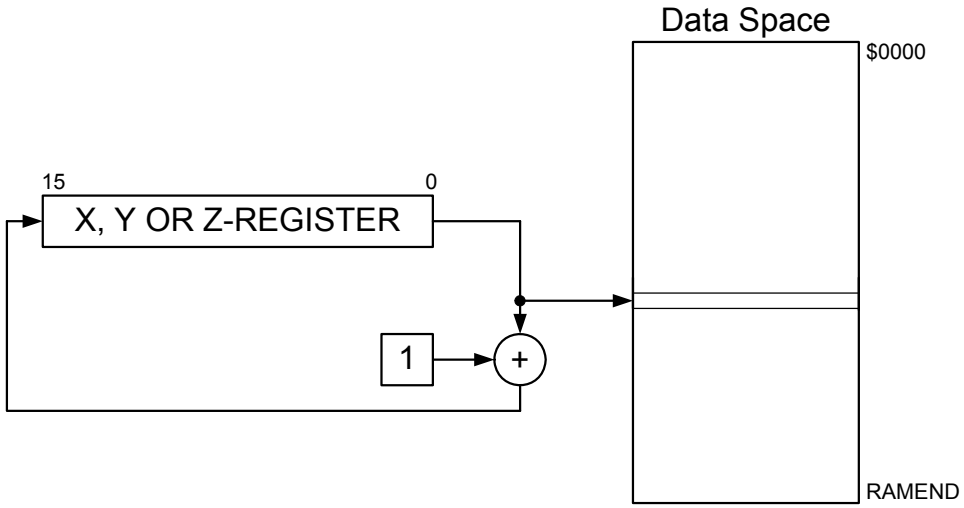
ST Z, R20 ; MEM (Z) ← R20

Register-indirekte Adressierung mit Prädekrement



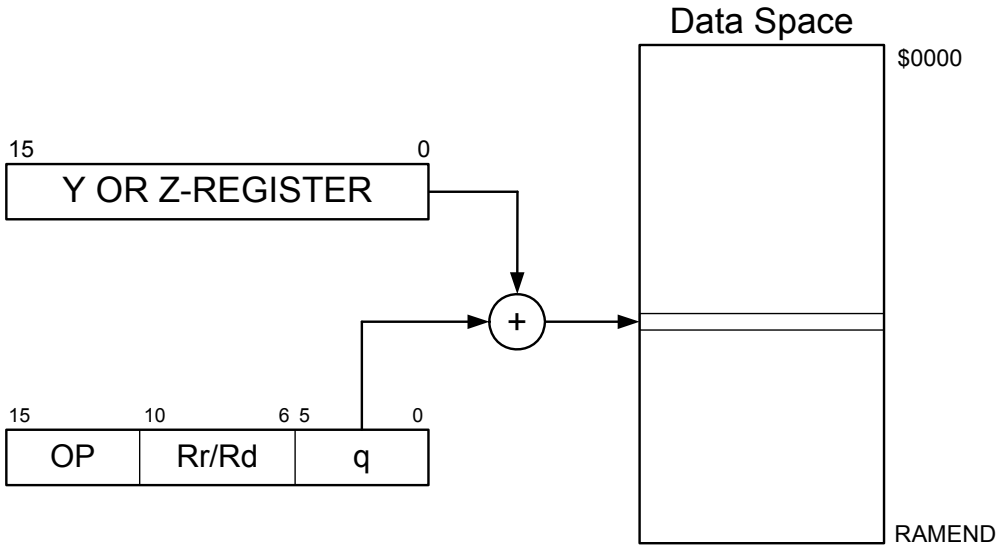
ST -Z, R20 ; Z ← Z - 1, MEM (Z) ← R20

Register-indirekte Adressierung mit Postinkrement



LD R20, Z+ ; R20 ← MEM (Z), Z ← Z + 1

Indizierte Adressierung (Register-indirekt mit Displacement)



q: 6-Bit-offset

LD R20, Z + 5 ; R20 ← MEM (Z + 5)

Achtung: Nur für Y- und Z-Register möglich, nicht für X-Register!

Nützlich z. B. für die Adressierung von Feldern oder Tabellen

Indirekte Adressierung

Unterstützt z.B. die Implementierung komplexer Datenstrukturen wie Felder, Stapel, Schlangen, Listen.

<u>Beispiel:</u> Initialisierung eines Feldes (vgl. oben)

```
byte [] Q = new byte [5]; …
for (i = 0; i <5, i ++)
   Q [i] = 0;
```

Beispiel mit absoluter Adressierung

```
;REGISTER DEFINITIONS
.def    TEMP    = R16          ; Temporary Variable

.DSEG                          ; DATA SEGMENT
.ORG    $0100                  ; Starting Adr in RAM
Q:      .BYTE   5              ; Reserve 5 bytes

.CSEG                          ; CODE SEGMENT
.ORG    $0000                  ; Starting Adr. in ROM
START:  CLR     TEMP           ; Clear TEMP
        STSQ, TEMP      ; Set Q[0] to zero
        STS     Q+1, TEMP  ; Set Q[1] to zero
        STS     Q+2, TEMP  ; Set Q[2] to zero
        STS     Q+3, TEMP  ; Set Q[3] to zero
        STS     Q+4, TEMP  ; Set Q[4] to zero
        RJMP    START          ; done, endless loop
```

<u>Problem:</u> Nicht für große Felder und dynamische Datenstrukturen geeignet!

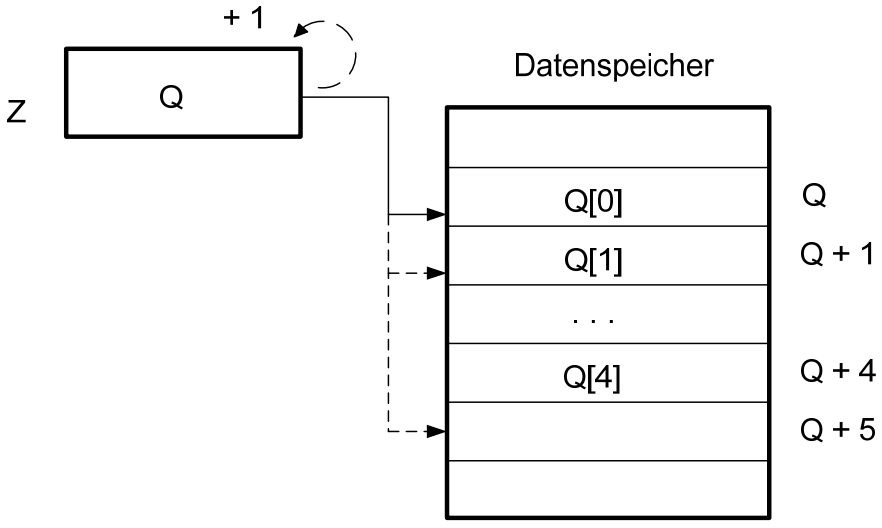
Beispiel mit indirekter Adressierung

```
; REGISTER DEFINITIONS
.def    TEMP    =   R16         ;  Temporary Variable
.def    ZL      =   R30         ;  Z-Reg low byte
.def    ZH      =   R31         ;  Z-Reg high byte

.DSEG   ;       DATA SEGMENT
.ORG    $0100                   ;  Starting Adr in RAM
Q:      .BYTE   5               ;  Reserve 5 bytes

.CSEG   ;       CODE SEGMENT
.ORG    $0000                   ;  Starting Adr. in ROM
INIT:   LDI     ZL, low(Q)      ;  Init Z-Reg with
        LDI     ZH, high(Q)     ;  Adr of Q[0]
        CLR     TEMP            ;  Clear Temp Var
ILOOP:  ST      Z+, TEMP        ;  Store in Q, postinc Z
        CPI     ZL, low(Q+5)    ;  End of Q reached?
        BRNE    ILOOP           ;  If not, then loop
        RJMP    INIT            ;  Else done, endless loop
```

Das Z-Register wird mit Adresse Q des Anfangselements Q[0] initialisiert und *indirekt* über dieses Register auf das Feld zugegriffen.



Nach Ausführung des Befehls ST Z+, TEMP wird das Z-Register automatisch um eins erhöht (Postinkrement) und zeigt auf das nächste Feldelement.

Die Ausführung erfolgt in einer Schleife, bis Z den Wert Q+5 erreicht, d.h. alle Elemente abgearbeitet sind.

8.5.6 Unterprogramme (Subroutines)

Beispiel: Byte-Multiplik. P = A·B·C durch wiederholte Addition mit Unterprog.

```
;REGISTER DEFINITIONS
.def    PROD      =     R16             ; Product
.def    MPY       =     R17             ; Multiplier
.def    MCND      =     R18             ; Multiplicand
.def    TEMP      =     R20             ; Temporary Variable
;MEMORY and I/O ADDRESSES
.equ    RAMEND    =     $45F            ; highest RAM addr
.equ    SPH       =     $5E             ; stackpointer addr.
.equ    SPL       =     $5D             ; high and low byte
; CONSTANTS
.equ    A         =     8
.equ    B         =     5
.equ    C         =     3
.DSEG                                   ; DATA SEGMENT
.ORG          $0100
P:            .BYTE     1

.CSEG                                   ; CODE SEGMENT
.ORG          $0000                     ; Starting Adress

;MAIN PROGRAM
;Computes P = A * B * C
INIT:         LDI       TEMP, high(RAMEND)
              STS       SPH, TEMP       : Init Stackpointer
              LDI       TEMP, low(RAMEND)
              STS       SPL, TEMP       ; to end of RAM
START:        LDI       MCND, A         ; Set Multiplicand
              LDI       MPY, B          ; Set Multiplier
              CALL      MULTI           ; First Multiply
              MOV       MCND, PROD      ; Set Mcnd to Prod
              LDI       MPY, C          ; Set Mpy once more
              CALL      MULTI           ; Second Multiply
              STS       P, PROD         ; Store Prod in RAM
              RJMP      INIT            ; Done, Endless loop

;SUBROUTINE MULTI
;Computes PROD = MCND * MPY
.def          CNT   =   R21             ; Iteration Counter
MULTI:        CLR       PROD            ; Clear Product
              MOV       CNT, MPY        ; Init Counter to MPY
              TST       CNT             ; Counter=0?
              BREQ      END             ; If yes, branch to END
LOOP:         ADD       PROD, MCND      ; add MCND to PROD
              DEC       CNT             ; Decrement Counter
              BRNE      LOOP            ; If not zero, loop again
END:          RET                       ; Return when done
```

Hauptprogramm MAIN

- Initialisiert den Stackpointer auf die höchste Adresse des Datenspeichers (RAMEND).

- Bereitet A und B als Eingabeparameter zur Übergabe in R18 und R17 vor.

- Ruft Unterprogramm MULTI auf.

- Setzt Ausgabeparameter PROD in R16 als Eingabeparameter in R18 sowie C in R 17.

- Speichert Ausgabeparameter PROD unter P im Datenspeicher.

Unterprogramm MULTI

- Eingabeparameter: MCND und MPY in R18 und R17.

- Berechnet Multiplikation MCND * MPY durch fortgesetzte Addition.

- Ausgabeparameter: PROD in R16 enthält Produkt.

- Kehrt ins Hauptprogramm zurück (RET-Befehl).

- Temporäres Register: R21 für CNT.

Achtung: R21 darf hier nicht im Hauptprogramm verwendet werden, da es im Unterprogramm modifiziert wird (**Seiteneffekt !**).

Vermeidung von Seiteneffekten

- Eingabeparameter (nach außen!) nicht verändern!

- Temporär genutzte Register im Unterprogramm auf Stack retten (PUSH) und vor Rückkehr ins Hauptprogramm restaurieren (POP), d.h. die Registerinhalte bleiben aus Sicht des rufenden Programms unverändert!

Beispiel: Unterprogramm MULTI ohne Seiteneffekte

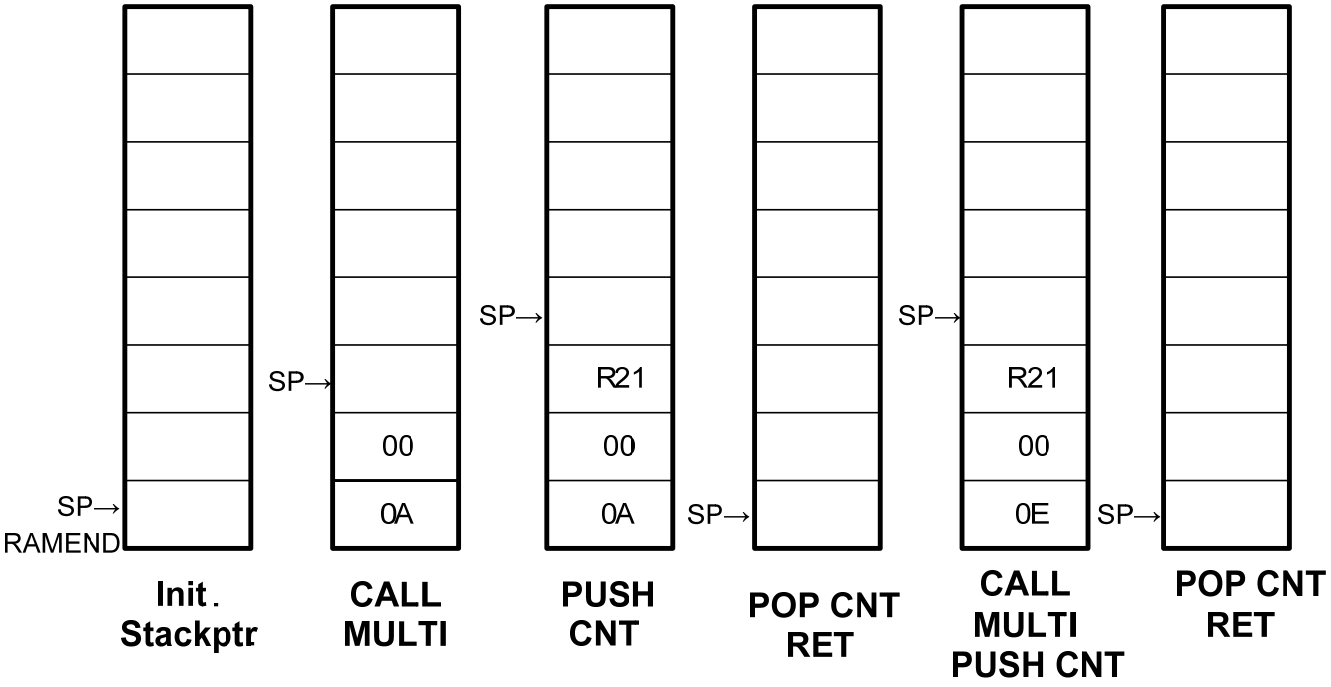
```
; SUBROUTINE MULTI
; Computes PROD = MCND * MPY
.def    CNT    = R21          ; Iteration Counter

MULTI:  PUSH   CNT            ; Save R21 on Stack
        CLR    PROD           ; Clear Product
        MOV    CNT, MPY       ; Init Counter to MPY
        TST    CNT            ; Counter=0?
        BREQ   END            ; If yes, branch to END
LOOP:   ADD    PROD, MCND     ; Else add MCND to PROD
        DEC    CNT            ; Decrement Counter
        BRNE   LOOP           ; If not zero, loop again
END:    POP    CNT            ; Restore R21 from Stack
        RET                   ; Return when done
```

Register R21 wird temporär auf dem Stack zwischengespeichert.

Für zuverlässige Programmierung unerlässlich!

Stapelinhalt im Verlauf der Programmausführung



Beachte: Der Stackpointer zeigt beim HATmega16 immer auf das **erste freie** Element des Stacks.

Die Verwaltung von Unterprogramm-Rückkehradressen mittels Stapel ist heute bei Mikroprozessoren Standard.

Vorteile:

- Geschachtelte und rekursive Unterprogramme sind leicht implementierbar.
- Stapel sind auch für andere Zwecke wie Parameterübergabe an Unterprogramme, Zwischenspeichern von Registerinhalten und lokalen Daten sowie Auswertung arithmetischer Ausdrücke sehr gut geeignet.

8.6 Vergleich der Befehlssatz-Architekturen

Beobachtungen:

Stacks sind für viele Anwendungen nützlich und können bei entsprechenden Adressierungsarten als zusätzlicher Mechanismus bei Register- oder Akku-Maschine leicht implementiert werden.

Der Zugriff auf einen externen (Programm-/Daten-)Speicher dauert wesentlich länger als auf prozessorinterne Register. Deshalb bestimmt bei modernen Prozessoren die Zahl der Operandenzugriffe auf den Speicher ganz wesentlich die Geschwindigkeit („Speicherflaschenhals“).

- *Akku-Maschine* benötigt viele temporäre Variablen im Speicher. Das führt zu langen Programmen und einer hohen Anzahl Speicherzugriffe.

  $\Rightarrow$ langsam bei langsamen Hauptspeicher (gilt für moderne Mikroprozessoren mit schnellem On-chip-Speicher (Cache) nicht unbedingt!)

  Akkumaschinen sind aber aus Kostengründen immer noch weit verbreitet (einfache Mikrocontroller).

- *Stackmaschine:* elegante Programmierung z. B. für die Abarbeitung von arithmetischen Ausdrücken, aber hohe Speicher(transfer)last

  $\Rightarrow$ Kaum noch reine *Stackmaschinen* in Hardware.

- *Registermaschine:* kurze Programme und geringe Speicherlast durch prozessorinterne Speicherung von Daten und Zwischenergebnissen.

- Registermaschinen mit Stack-Mechanismen stellen die *flexiblere und leistungsfähigere* Prozessorarchitektur dar.

- *Load/Store- bzw. reg-reg-Architektur:* durch Befehle fester Länge (1 Wort = 1 Befehl) einfache und schnelle Befehlsdekodierung

  $\Rightarrow$ geringe Kosten und hohe Taktrate

Moderne Prozessoren erlauben Überlappung von Befehlshol- und Befehlsausführungsphase (Phasen-Pipelining, s.u.), so dass eine größere Befehlslänge nicht so sehr ins Gewicht fällt.

RISC-orientierte Befehlssätze erlauben kurze Befehle auch bei Registermaschinen.

Fazit:

Registermaschinen mit Load/Store-Architektur sind ein sehr guter Kompromiss aus Hardwareaufwand (Kosten) und Geschwindigkeit.

$\Rightarrow$ Fast alle modernen Prozessoren verwenden eine Load/Store-Architektur