Aufbau und Funktionsweise programmierbarer digitaler Systeme (Teil 2)

1 Assemblerprogrammierung

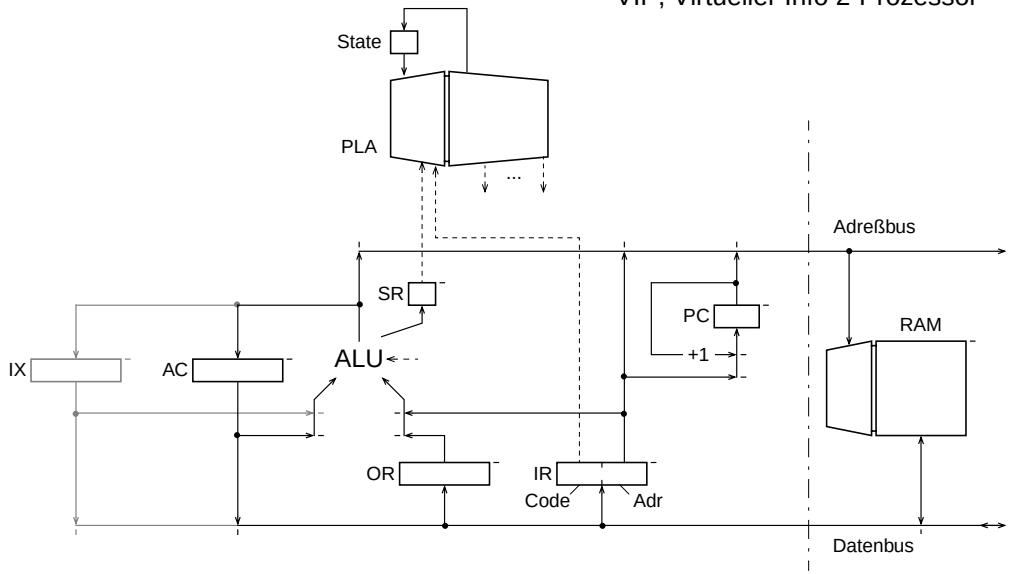
1.1 Ein einfacher Rechner (VIP)

Merkmale

- 1-Adreß-Rechner
  - → explizite Adressierung des Hauptspeichers
  - → implizite Adressierung eines Akkumulator-Registers
  - → 16-Bit-Datenformat, d.h. 16-Bit-ALU
  - → 8-Bit-Speicheradresse
- Hauptspeicher
  - → Kapazität: 2^8 = 256 16-Bit-Wörter
  - → Zugriff: wortweise

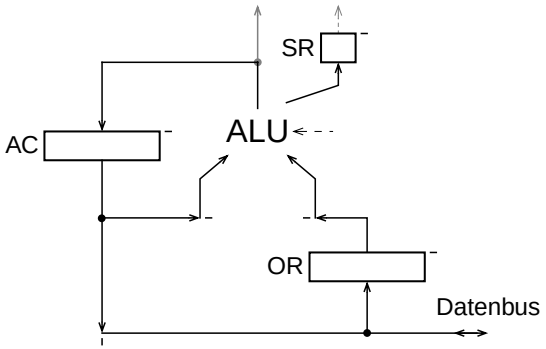
Gesamtstruktur

Einfacher 1-Adreß-Rechner:
VIP, Virtueller Info 2-Prozessor



Register und Speicher

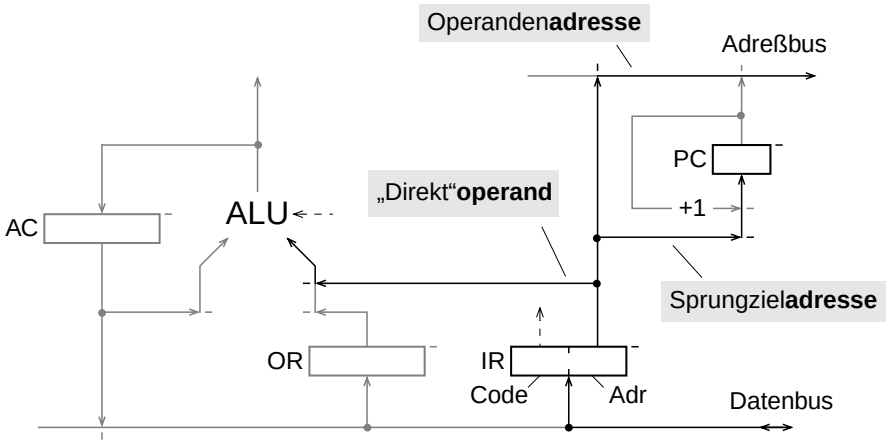
- **AC** (**ac**cumulator) → **1. Quellregister** sowie **Zielregister** der ALU, implizit per Befehl adressierbar.
- **OR** (**o**perand **r**eg.) → **2. Quellregister** der ALU. Puffer bei Lesezugriffen auf Operanden im Speicher (nicht per Befehl adressierbar).
- **SR** (**s**tatus **r**eg.) → **Prozessorstatusregister** als Zielregister für die **Bedingungsbits** z, n, c, v.



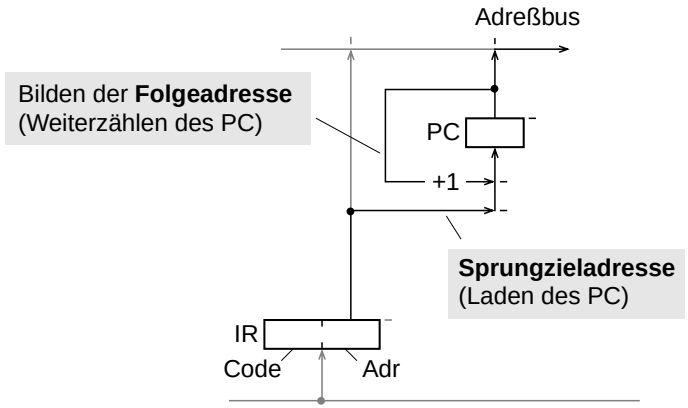
Arithmetisch-logische Einheit, ALU, mit ihren Registern.

- **IR** (**i**nstruction **r**eg.) → **Befehlsregister** zur Pufferung und Auswertung des aktuellen Befehls nach dem Lesen aus dem Speicher.

  Man beachte die unterschiedlichen Wirkungen von „Adr“!

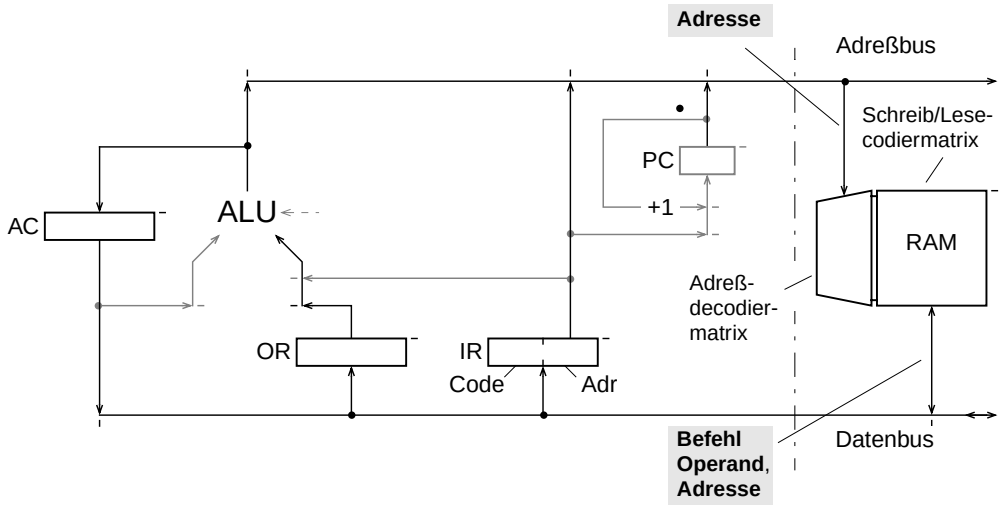


- **PC** (**p**rogram **c**ounter) → „**Befehlszähler**“register für das Speichern und Hochzählen der Adresse des nächsten aus dem Speicher zu lesenden Befehls.



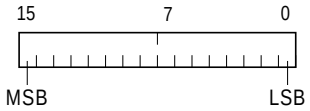
- **RAM** (**r**andom **a**ccess **m**emory) → Prozessorexterner Schreib-/Lesespeicher.

  Hauptspeicher (**Arbeitsspeicher**) für die Speicherung von Programmen und Daten.



Datenformat und Datentypen

16-Bit-Format:



MSB = most significant bit (höchstwertiges Bit)

LSB = least significant Bit (niedrigstwertiges Bit)

- **Zahlen**:

  Vorzeichenlose Dualzahlen: 0 bis $2^{16}-1$ d.h. 0 bis 65535

  2-Komplement-Zahlen: -2^{15} bis $+2^{15}-1$ d.h. –32 768 bis +32 767

- **Bitvektor**:

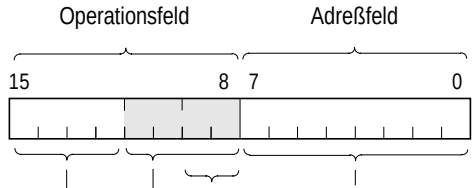
  Einzelne Bits 0 bis 15

- **Zeichen (character)**:

  ASCII-Zeichen in Bit 0 bis 6, Zero-Extension in Bit 7 bis 15

Befehlsformat

16-Bit-Format:



a) 8-Bit-**Speicheradresse** (speicherdirekt)

b) 8-Bit-Operand unmittelbar → **Direktoperand**

Adressierungsart: 1. Speicherdirekte Adressierung (00)
2. Direktoperand-Adressierung (01)
3. Speicherindirekte Adressierung (10)
4. Indizierte Adressierung (11)

4-Bit-**OpCode-Erweiterung**

4-Bit-**OpCode** (Operationscode, Befehlscode)

} „Code“

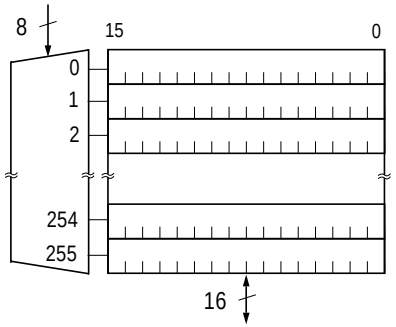
Adreßformat

8-Bit-Format:

- **Adreßraum**: 0 bis 255!

  → Es sind 256 **16-Bit-Speicherwörter** adressierbar.

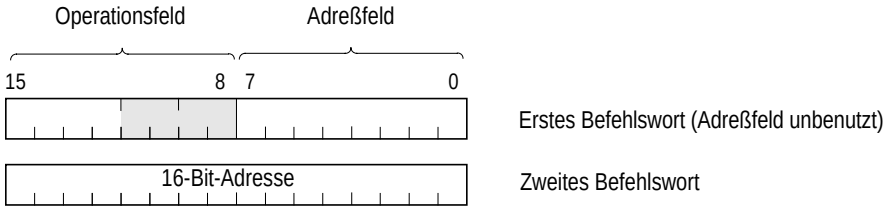
RAM:



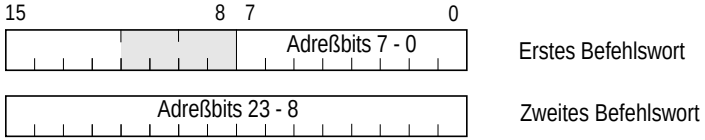
Erweitertes Adreßformat

- **Mehrwortbefehle** (typisch für CISC-Prozessoren)

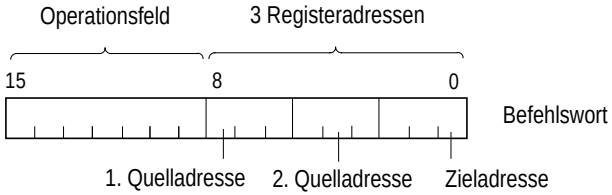
  16-Bit-Adresse → Adreßraum von **64K Adressen**



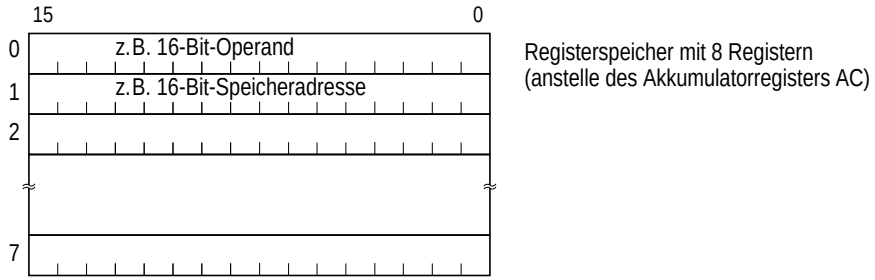
  24-Bit-Adresse → Adreßraum von **16M Adressen**



- **Einwortbefehle und registerindirekte Adressierung**
  (typisch für RISC-Prozessoren)

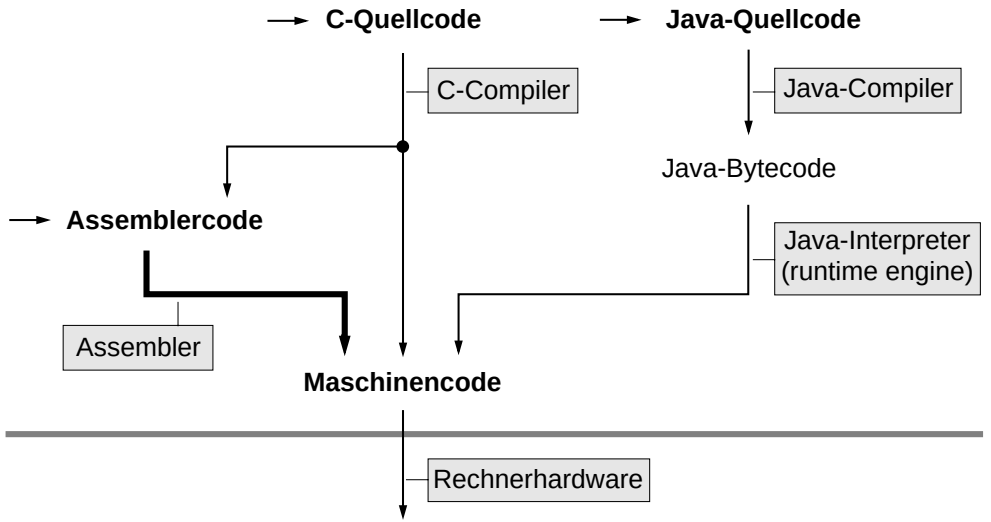


16-Bit-Adresse → Adreßraum von **64K Adressen**



1.2 Assemblersprache

Hierarchie der Sprachebenen



Assemblercode, Assemblerprogramm

→ **Symbolische Schreibweise** eines Programms.

→ Vom Assembler **1-zu-1 übersetzbar** in den Maschinencode

d.h., jede Assemblercodezeile ergibt genau eine Maschinencodezeile:

– einen **Maschinenbefehl**,

– einen Datenwert als **Konstante**,

– einen freien („leeren“) Speicherplatz für eine **Variable**.

Beispiele:

`ADD M`	Addiere den Inhalt der mit der symbolischen Adresse M adressierten Speicherzelle zum Inhalt des Akkumulators AC und schreibe das Ergebnis in den Akkumulator AC: AC := AC + M (zur **Programm*lauf*zeit**).
`M DAT -2`	Stelle eine Speicherzelle mit der symbolischen Adresse M bereit und initialisiere sie mit dem Dezimalwert –2: M := –2 (zur **Programm*lade*zeit**)

Maschinencode, Maschinenprogramm

→ **Binäre Schreibweise** eines Programms → **0/1-Code**.

→ Dieses ist unmittelbar **durch die Hardware interpretierbar**!

Beispiele:

`ADD M`	→ 0100 0000 1000 0001	→ ADD hat den OpCode 0100; → die OpCode-Erweiterung (inkl.Adressierungsart) hat die Codierung 0000; → die symbolische Adresse M habe die numerische Entsprechung $1000\ 0001_2 = 129_{10}$.
`M DAT -2`	→ 1111 1111 1111 1110	→ Inhalt der Speicherzelle 129: –2.

Bestandteile der Assemblersprache

- **Maschinenbefehle**

  → Befehle an die Rechnerhardware

  ADD, **SUB**, **AND**, **OR**, **LDA**, **NOP**, **. . .**

- **Assembleranweisungen**

  → Anweisungen an den Assembler zur Programmorganisation

  ORG, **END**

  → Anweisungen an den Assembler zur Konstanten- und Variablendefinition

  EQU, **DAT**, **RES**

- **Makroanweisungen**

- Die Assemblersprache legt darüber hinaus fest:

  → Format einer **Programmzeile**

  → Syntax von **Symbolen** und **Zahlen**, inkl. **Zeichensatz**

Format einer Programmzeile

Aufteilung in

- **Namensfeld** / **Labelfeld**: Symbolische Benennung von Befehlen und Daten.
- **Operationsfeld**: Angabe eines Befehls oder Anweisung.
- **Adreßfeld**: Angabe eines Operanden oder einer Sprungzieladresse.
- **Kommentarfeld**: Zur Dokumentation; erzeugt keinen Maschinencode!

Beispiel:

```
loop    LDA   i       ; Variable i laden
        ADD   #1      ; Eins addieren
        STA   i       ; und wieder speichern
        JMP   loop    ; Zurück nach loop (endlos)
```

Symbole, Zahlen und Bitmuster

- **Symbole**
  - → Treten im **Namensfeld** und im **Adreßfeld** auf.
  - → Stehen stellvertretend für numerische **Adressen** und **Operanden**.
  - → Sind im Prinzip frei wählbar,
    unterliegen beim **VIP-Assembler** aber folgenden Vorgaben:
    - erlaubt sind **Buchstaben** (ohne Umlaute und ß) und **Ziffern** (0 bis 9)
      sowie das **Sonderzeichen** „_“ (underline);
    - das **erste Zeichen** muß ein Buchstabe sein (vereinfacht die Erkennung);
    - die ersten **256 Zeichen** sind signifikant;
    - **Groß- und Kleinschreibung** wird unterschieden.

  Beispiele:

  zulässig: loop, A_1, Var_2, EiNs, Zaehler

  nicht zulässig: 1_A, Var 2, Zähler

Der **Wert eines Symbols** wird durch sein Auftreten im **Namensfeld** bestimmt, als:

→ numerische Adresse eines Maschinenbefehls: **Sprungzieladresse**,

→ numerische Adresse eines Datenelements: **Programmkonstante** (DAT) oder **Programmvariable** (RES),

→ Wertzuweisung: **Assemblerkonstante** (EQU).

Beispiel:

```
Flag     EQU   254       ; Assemblerkonst. „Flag“
TrueFlag DAT   1         ; Programmkonst. „TrueFlag“
X        RES   1         ; Programmvariable „X“
          :
Wait     LDA   Flag      ; Sprungzieladresse „Wait“
         CMP   TrueFlag
         BNE   Wait
          :
         STA   X
```


- **Zahlen und Bitmuster**

  → **Dezimalzahl** benötigt keine Kennung: 1234, +15, -2

  → **Hexadezimalzahl** erhält „**0x**“ vorangestellt: 0x12, 0xF7, 0xff2b (ggf. Zero-Extension durch den Assembler)

  → **Dualzahl** erhält „**%**“ vorangestellt: %0000010011010010 (entspricht 1234)

  → **Bitmuster** erhält „**%**“ vorangestellt: %0000000011110111 (entspricht 0xF7)

  → **ASCII-Zeichen** wird mit „**'**“ eingerahmt: 'a' (entspricht 0x0061, d.h. Zero-Extension durch den Assembler)

  ASCII-String (ohne Null-Terminierung): 'A','B','C' oder 'ABC'

1.3 Assembleranweisungen

- Sie sind **Anweisungen an den Assembler**
- Sie werden zur **Assemblierzeit** ausgeführt.
- Sie sind **nicht Bestandteile des** zu erzeugenden **Maschinenprogramms**.

Programmanfang (origin):

ORG n Der Assembler richtet das Programm an der **numerischen Adresse n** aus.

Sie gibt den Ort des ersten im Speicher stehenden Maschinenbefehls oder Datums vor.

Programmende (end):

END Symbol Der Assembler beendet die Assemblierung.

„Symbol“ gibt die **Startadresse**, d.h. die symbolische Adresse des ersten auszuführenden Befehls an.

Sie wird „später“ in den **Befehlszähler PC** geladen.

Setze gleich (equate):

Symbol **EQU** **c**

Der Assembler setzt das Symbol gleich dem Wert der **Konstanten c** (→ **Assemblerkonstante!**).

Er führt bei Auftreten des Symbols im Programm eine entsprechende **Textersetzung** durch.

Reserviere Variablen-Speicherplatz (reserve):

[**Symbol**] **RES** **n**

Der Assembler hält n Maschinencodewörter (Speicherwörter) für **n Variablen** frei.

Steht ein Symbol im Namensfeld, so stellt dieses die **symbolische Adresse** der ersten Variablen (des ersten Speicherworts) dar.

Der Assembler ersetzt das Symbol bei seiner Verwendung im Programm durch den entsprechenden Adreßwert.

[…] kennzeichnet hier und nachfolgend eine **Option**.

Definiere Konstante (data):

[**Symbol**] **DAT** **x**[,y, …]

Der Assembler erzeugt (≥) ein Maschinencodewort (Speicherwort) mit dem Wert der **Konstanten x** (y).

Steht ein Symbol im Namensfeld, so stellt dieses die **symbolische Adresse** der (ersten) Konstanten dar.

Beispiel:

VIP-Assembler

```
          ORG   0
MwSt      EQU   16
Zahl      DAT   1234
Feld      DAT   +15,0x12
          DAT   -2
GPunkt    RES   2
Zeichen   DAT   'a'
Text      DAT   'OK',0
Start     HLT
          END   Start
```

Speicherbelegung

Zahl	0x04D2	0
Feld	0x000F	1
	0x0012	2
	0xFFFE	3
GPunkt		4
		5
Zeichen	0x0061	6
Text	0x004F	7
	0x004B	8
	0x0000	9
Start	0x0000	10

1.4 Maschinenbefehle

- Sie sind **Befehle an die Rechnerhardware**.
- Sie bilden das **Maschinenprogramm**.
- Sie werden zur **Programmlaufzeit** ausgeführt.

Die Maschinenbefehle bilden **Befehlsgruppen**, wie folgt.

Sonstige Befehle

Befehl	Wirkung		Code
HLT	halte an	Der Prozessor wird angehalten!	0000 00 00
NOP	no operation	Es wird keine Operation ausgeführt; der Befehl benötigt jedoch die reguläre Abruf- und „Ausführungszeit“.	0000 00 01

→ **HLT** dient bei unserem VIP-Rechner (Simulator) als **Programmabschluß**.

In der **Realität** erfolgt der Programmabschluß durch einen sog. **Trap-Befehl**, der den Rücksprung in das Betriebssystem über eine Programmunterbrechung bewirkt!

Lade-/Speichere-Befehle

Befehl		Wirkung		z n c v	Code
LDA	M	lade	AC : = M	x x 0 0	0010 00 x x
STA	M	speichere	M : = AC	– – – –	0011 00 x x

Generell: Die **Bedingungsbits** z, n, c und v (**condition code cc**) werden von den Befehlen des Maschinenbefehlssatzes ggf. beeinflußt (x, 0) oder nicht (–).

Beispiel:

Java

```
int i = 26, j;
 j = i;
```

VIP-Assembler

```
          ORG   0x10
i         DAT   26
j         RES   1

Start     LDA   i
          STA   j
          HLT
          END   Start
```

Arithmetische und logische Befehle

Befehl		Wirkung		z n c v	Code
ADD	M	addiere	AC := AC + M	x x x x	0100 00 xx
SUB	M	subtrahiere	AC := AC – M	x x x x	0101 00 xx
CMP	M	vergleiche	AC – M wirkt nur auf cc!	x x x x	0110 00 xx
NEG		negiere	AC := 0 – AC 2-K-Zahl!	x x x x	0000 00 11
AND	M	und	AC := AC $\wedge$ M	x x 0 0	0111 00 xx
OR	M	oder	AC := AC $\vee$ M	x x 0 0	1000 00 xx
XOR	M	exkl. oder	AC := AC $\oplus$ M	x x 0 0	1001 00 xx
NOT		nicht	AC := $\neg$AC	x x 0 0	0000 00 10

→ **ADD**, **SUB** und **CMP** arbeiten sowohl mit vorzeichenlosen Dualzahlen als auch mit 2-Komplement-Zahlen. Die Auswertung der unterschiedlichen Zahlendarstellungen erfolgt über die Bedingungsbits (siehe bedingte Sprungbefehle).

→ **CMP** führt eine Subtraktion aus, erzeugt dabei aber **kein Ergebnis in AC**.

→ **AND**, **OR** und **XOR** wirken bitpaarweise, **NOT** wirkt einzelbitweise.

Beispiele für die logischen Befehle:

a) **Maskieren** (mask)

```
        ORG     0
ASCII   DAT     0x0036
Mask    DAT     0x000f
Ziffer  RES     1

        LDA     ASCII
        AND     Mask
        STA     Ziffer
```

Wirkung:

ASCII: 0000 0000 0011 0110

Mask: 0000 0000 0000 1111

AND: Ziffer: 0000 0000 0000 0110

b) **Verschmelzen** (merge)

```
        ORG     0
Ziffer  DAT     0x0006
Zone    DAT     0x0030
ASCII   RES     1

        LDA     Ziffer
        OR      Zone
        STA     ASCII
```

Wirkung:

Ziffer: 0000 0000 0000 0110

Zone: 0000 0000 0011 0000

OR: ASCII: 0000 0000 0011 0110

c) **Einzelne Bits invertieren** (toggle)

```
        ORG     0
X       DAT     0xF05A
Invert  DAT     0x0088

        LDA     X
        XOR     Invert
        STA     X
```

Wirkung:

X: 1111 0000 **0**101 **1**010

Invert: 0000 0000 **1**000 **1**000

XOR: X: 1111 0000 **1**101 **0**010

d) **Alle Bits invertieren** (one's complement)

```
        ORG     0
X       DAT     0x0001
XCompl  RES     1

        LDA     X
        NOT
        STA     XCompl
```

Wirkung:

X: 0000 0000 0000 0001

NOT: XCompl: 1111 1111 1111 1110

Weitere arithmetische Befehle (im VIP nicht realisiert!)

Befehl		Wirkung		Bitanzahl
MUL	M	multiply (signed)	AX | AC := AC $\times$ M	32 := 16 $\times$ 16
MULU	M	multiply unsigned	AX | AC := AC $\times$ M	32 := 16 $\times$ 16
DIV	M	divide (signed)	AC := AC / M, AX := AC % M	16_Q,16_R := 16 / 16
DIVU	M	divide unsigned	AC := AC / M, AX := AC % M	16_Q,16_R := 16 / 16

→ Die **Multiplikation** erzeugt ein Produkt mit doppelter Stellenanzahl. Das erfordert ein Zusatzregister AX zur Speicherung des höherwertigen Teils des Produkts.

„AX | AC" bezeichnet das „Aneinanderfügen" (**Konkatenieren**) der beiden Register AX und AC.

→ Die **Division** geht hier von einem Dividenden mit einfacher Stellenanzahl aus und erzeugt den Quotienten und den Rest mit ebenfalls einfacher Stellenanzahl.

„% M" steht für Modulo M.

→ **Multiplikation** und **Division** haben unterschiedliche Algorithmen für die vorzeichenlose (unsigned) und die vorzeichenbehaftete Zahlendarstellung (signed).

Shift-Befehle

Befehl	Wirkung		z n c v	Code
ASL	arithmetischer Shift nach links	$AC := AC \times 2^1$ (Bit 0 := 0)	x x AC_{15} x	0000 0100
ASR	arithmetischer Shift nach rechts	$AC := AC \times 2^{-1}$ (Bit 15 := Bit 15)	x x AC_0 0	0000 0101
LSL	logischer Shift nach links	$AC := AC \times 2^1$ (Bit 0 := 0)	x x AC_{15} 0	0000 0110
LSR	logischer Shift nach rechts	$AC := AC \times 2^{-1}$ (Bit 15 := 0)	x x AC_0 0	0000 0111

→ **ASL** entspricht der Multiplikation des Operanden in AC mit 2.

→ **ASR** entspricht der Division des Operanden in AC durch 2 (ohne Rest-Bildung).

Die Vorzeichendarstellung bleibt mittels Nachziehens des höchstwertigen Operandenbits in die höchstwertige Bitposition von AC erhalten: Bit 15 := Bit 15.

Sprungbefehle

a) Unbedingter Sprung (jump)

Befehl		Wirkung	Code
JMP	M	springe zur Adresse M	0001 0000

b) Bedingter Sprung (branch conditionally: B*cond*)

Befehl B*cond*		Wirkung: springe nach M, wenn Bedingung *cond* erfüllt		Code
BPL	M	plus	n = 0	0001 0001
BMI	M	minus	n = 1	0001 0010
BCC	M	carry clear	c = 0	0001 0011
BCS	M	carry set	c = 1	0001 0100
BVC	M	overflow clear	v = 0	0001 0101
BVS	M	overflow set	v = 1	0001 0110

→ Für alle Sprungbefehle ist die **speicherdirekte Adressierung** fest vorgegeben.
Die OpCode-Erweiterung wird zur Codierung der Bedingung *cond* benutzt.

b) Bedingter Sprung (branch conditionally: B*cond*) – Forts.:

Befehl B*cc*		Wirkung: springe nach M, wenn Bedingung *cond* erfüllt		Code
BEQ	M	equal =	z = 1	0001 0111
BNE	M	not equal ≠	z = 0	0001 1000
BGT	M	greater than (signed) >	z = 0 and n = v	0001 1001
BGE	M	greater or equal (signed) ≥	n = v	0001 1010
BLE	M	less or equal (signed) ≤	z = 1 or n ≠ v	0001 1011
BLT	M	less than (signed) <	n ≠ v	0001 1100
BGTU	M	greater than unsigned >	z = 0 and c = 0	0001 1101
BGEU	M	greater or equal unsigned ≥	c = 0	0001 0011
BLEU	M	less or equal unsigned ≤	z = 1 or c = 1	0001 1110
BLTU	M	less than unsigned <	c = 1	0001 0100

→ **BGEU** ist mit **BCC** und **BLTU** mit **BCS** identisch.

Zu den bedingten Sprungbefehlen:

→ Zur Unterscheidung von **vorzeichenlosen Dualzahlen** und **2-Komplement-Zahlen** gibt es für jeden der beiden Datentypen vier bedingte Sprungbefehle, die zur Bildung der **Bedingungsoperationen** (<, ≤, ≥ oder > gleich „true“/„false“) die Bedingungsbits z und c **bzw.** z, n und v auswerten;

z.B.:

BLT und BLTU stehen beide für AC < M, jedoch für unterschiedliche Zahlendarstellungen der Operanden in AC und M.

→ Die Relationen <, ≤, ≥ und >,

z. B. AC ≤ M,

basieren auf der Subtraktion. Diese erfolgt durch einen der beiden Befehle **CMP M** oder **SUB M** als AC – M und beeinflußt mit ihrem Ergebnis die Bedingungsbits cc.

Beispiele:

a) **Programmverzeigung**: Springe bei **X > Y** (**2-Komplementzahlen**):

```
        ORG   0
X       DAT   0x8000
Y       DAT   0x7FFF
         :
        LDA   X
        CMP   Y
        BGT   Greatr
LessEq  ADD   Y
         :
         :
Greatr   :
```

Wirkung:

X: 1000 0000 0000 0000

Y: 0111 1111 1111 1111

CMP: *X–Y:* *0000 0000 0000 0001*

cc-Bits: z := 0,
n := 0,
c := 0,
v := $c_{n-1} \oplus c_n = 1$.

BGT: Branch if z = 0 and n = v

d.h., die **Sprungbedingung X > Y** ist für 2-Komplement-Zahlen **nicht erfüllt**!

Im Programm weiter mit **ADD Y**

b) **Programmverzeigung**: Springe bei **X > Y** (**vorzeichenlosen Dualzahlen**):

```
        ORG   0
X       DAT   0x8000
Y       DAT   0x7FFF
         …
        LDA   X
        CMP   Y
        BGTU  Greatr
LessEq  ADD   Y
         :
         :
Greatr   :
```

Wirkung:

X: 1000 0000 0000 0000

Y: 0111 1111 1111 1111

CMP: *X–Y:* *0000 0000 0000 0001*

cc-Bits: z := 0,
n := 0,
c := 0,
v := $c_{n-1} \oplus c_n = 1$.

BGTU: Branch if z = 0 and c = 0

d.h., die **Sprungbedingung X > Y** ist für vorzeichenlose Dualzahlen **erfüllt**!

Im Programm weiter bei **Greatr**

1.5 Steuerkonstrukte

- Sie dienen zur **Programmflußsteuerung** durch **Verzweigung** und **Schleifenbildung**.
- Ihre in **höheren Programmiersprachen** gebräuchliche Formen sind:
  - → **IF-Verzweigung** → Einfache Verzweigung.
  - → **IF-ELSE-Verzweigung** → Verzweigung mit Alternative.
  - → **WHILE-Schleife** → Wiederholung, solange Bedingung erfüllt ist.
  - → **DO-Schleife** → Es gibt wenigstens einen Durchlauf.
- Sie sind in **Assemblersprache** abzubilden.

IF-Verzweigung → Einfache Verzweigung

Java

if (<Boolesche Bedingung>)
<Anweisung>

z.B.:

```
if ( x != 0 )
  y = 1/x;
```

VIP-Assembler

```
        ORG   0
null    DAT   0
eins    DAT   1
X       RES   1
Y       RES   1
         :
        LDA   X
        CMP   null
        BEQ   contin
if      LDA   eins
        DIV   X
        STA   Y
contin   :
```

IF-ELSE-Verzweigung → Verzweigung mit Alternative

Java

if (<Boolesche Bedingung>)
<Anweisung> **else** <Anweisung>

z.B.:

```
if ( x != 0 )
  y = 1/x;
else
  y = 0;
```

VIP-Assembler

```
          ORG   0
null      DAT   0
eins      DAT   1
X         RES   1
Y         RES   1
           :
          LDA   X
          CMP   null
          BEQ   else
if        LDA   eins
          DIV   X
          STA   Y
          JMP   contin
else      LDA   null
          STA   Y
contin     :
```


While-Schleife → Wiederholung, solange Bedingung erfüllt ist

Java

while (<Boolesche Bedingung>)
<Anweisung>

z.B.:

```
int i = 0;
while ( i < 100 ) {
  ...Rumpf...; i ++;
}
```

VIP-Assembler

```
null      DAT   0
eins      DAT   1
cent      DAT   100
i         RES   1
           :
          LDA   null
          STA   i
while     LDA   i
          CMP   cent
          BGE   contin
           :
          Rumpf
           :
          LDA   i
          ADD   eins
          STA   i
          JMP   while
contin     :
```


DO-Schleife → Es gibt wenigstens einen Durchlauf

Java

do <Anweisung> **while**
(<Boolesche Bedingung>);

z.B.:

```
int i = 0;
do {
  ...Rumpf...; i ++;
} while ( i < 100 );
```

VIP-Assembler

```
null      DAT   0
eins      DAT   1
cent      DAT   100
i         RES   1
           :
          LDA   null
do        STA   i
           :
          Rumpf
           :
          LDA   i
          ADD   eins
          CMP   cent
          BLT   do
contin     :
```


1.6 Adressierungsarten

- Sie sind **Merkmale des Prozessors**, d.h. in Hardware realisiert.
- Sie bieten verschiedene Möglichkeiten des **Zugriffs auf Operanden** zur Unterstützung unterschiedlicher Programmieraufgaben.
- Gebräuchliche Adressierungsarten sind:
  - → **Speicherdirekte Adressierung**,
  - → **Direktoperand-Adressierung**,
  - → **Speicherindirekte Adressierung**,
  - → **Indizierte Adressierung**,
  - → **Registerindirekte Adressierung**.

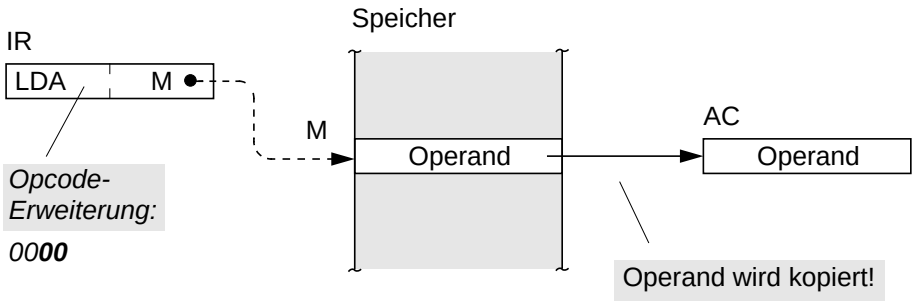
  Mit Ausnahme der registerindirekten Adressierung sind sie im **VIP** realisiert.
- Es gibt weitere Adressierungsarten, z.B.:
  - → **Basisrelative Adressierung**,
  - → **PC-relative Adressierung**.
- Es gibt **Erweiterungen** und **Kombinationen** der genannten Adressierungsarten.

Speicherdirekte Adressierung

1) Absolute Adressierung von **Operanden**:

- Die **Operandenadresse** steht im Befehl.
- Der **Operand** steht im Speicher.

LDA M



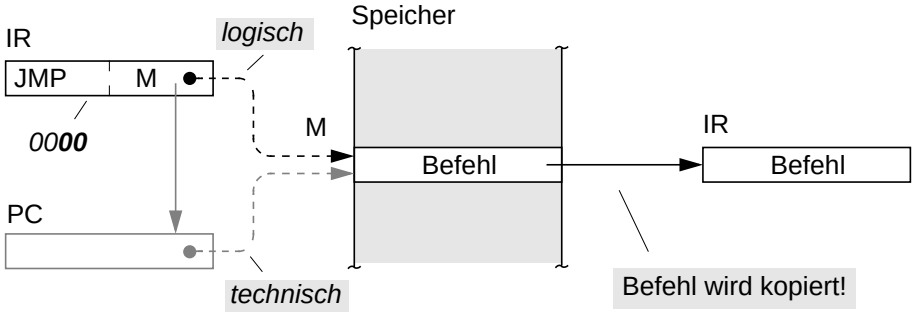
Effektive Adresse (tatsächliche Speicheradresse) des Operanden = M

Symbolik: Adressierung: ●- - -► Transport: ⟶

2) Absolute Adressierung von **Befehlen**:

- Die **Sprungzieladresse** steht im Befehl.
- Der **Befehl** steht im Speicher.

JMP M



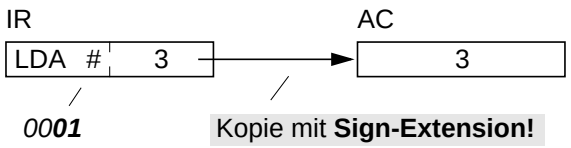
Effektive Adresse des Befehls = M

Direktoperand-Adressierung

- Der **Operand** steht als sog. **Direktoperand** unmittelbar im Befehl.
- Der Befehl kann **nur Quelle** eines Direktoperanden und **nicht Ziel** sein!
- Kennzeichnung durch **„#“-Zeichen**:

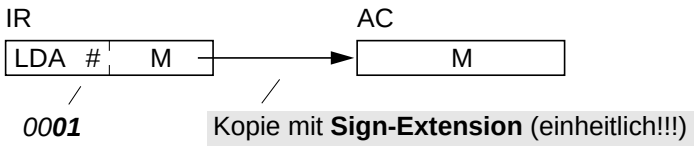
a) **Konstante** als Direktoperand, z.B. 3 oder M (mit z.B. M EQU 3):

LDA #3



b) **Adresse** als Direktoperand, z.B. M (mit z.B. M RES 1 oder M ADD i):

LDA #M



Beispiel: **Fakultät von n** (iterativ)

VIP-Assembler

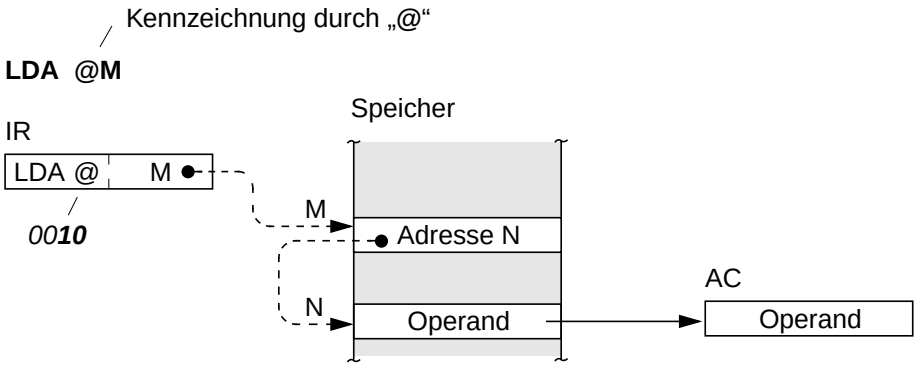
```
          ORG   0
n         DAT   5       ; Argument n
Fac       RES   1       ; Speicherplatz für n!

Start     LDA   #1
          STA   Fac     ; Fac = 1
Loop      LDA   n
          BEQ   Ende    ; while n ≠ 0 fahre fort
          MUL   Fac     ; Fac = n * Fac
          STA   Fac
          LDA   n
          SUB   #1      ; n = n - 1
          STA   n
          JMP   Loop
Ende      HLT
          END   Start
```

Speicherindirekte Adressierung für variable Adressierung

Funktion:

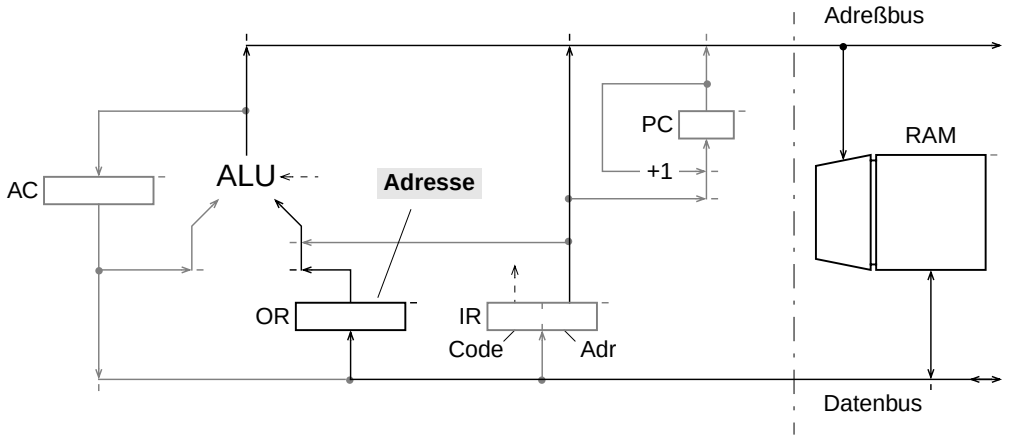
- Im Befehl steht die **Adresse der Operandenadresse**.
- Die **Operandenadresse** und der **Operand** stehen im Speicher.
- Die Operandenadresse kann jetzt verändert werden → **Pointer!**



Effektive Adresse = N

Informationsfluß:

- **OR** (**o**perand **r**egister) als **Pufferregister** für die Adreßindirektion.



Beispiel: **Polynomauswertung 1**

Ermittlung des Polynomwertes $\mathbf{p = a_0 \cdot x^3 + a_1 \cdot x^2 + a_2 \cdot x^1 + a_3}$

Dazu Festlegung folgender Konstanten und Variablen für den

VIP-Assembler:

N: Grad des Polynoms, hier als Konstante 3.

FELD: Variable Koeffizienten a_0, a_1, a_2, a_3,
gespeichert mit aufsteigenden Adressen.

PTR: Pointer als Variable zur Adressierung der Inhalte von **FELD**.

X: Argumentwert, als Variable vorgegeben.

P: Polynomwert; ist zu ermitteln.

Grundsätzlich könnte auch der Grad des Polynoms als variabel vorgegeben werden. Das Koeffizientenfeld wäre dann dennoch mit fester Anzahl möglicher Einträge vorzusehen, da die Speicherverwaltung statisch ist.

Polynomauswertung 1 mit **speicherindirekter Adressierung**

```
          ORG   0
N         DAT   3        ; Polynomgrad
FELD      RES   4        ; Koeffizientenfeld a0,a1,...
PTR       RES   1        ; Pointer (Adresszeiger)
EndAdr    RES   1        ; Endeadresse von Koeff.-Feld
X         RES   1        ; Argumentwert
P         RES   1        ; Polynomwert
Start      :             ; Koeff. und x ermittelt
          LDA   #FELD    ; Feldanfangsadresse
          STA   PTR      ; als Pointer
          ADD   N        ; Endeadresse ermittelt
          STA   EndAdr
          LDA   @PTR     ; Koeffizientenzugriff a0
          STA   P        ; p mit a0 initialisiert
```

Fortsetzung auf nächster Seite

Fortsetzung der Polynomauswertung 1

```
Loop    LDA   PTR
        CMP   EndAdr
        BEQ   Ende    ; Schleifenabbruch
        ADD   #1      ; Pointer erhoeht
        STA   PTR
        LDA   P
        MUL   X       ; erster Lauf: a0 * x
        ADD   @PTR    ;              (a0 * x) + a1
        STA   P
        JMP   Loop
Ende    HLT
        END   Start
```

Erkenntnis:

- Umständlich ist das Verwalten des Zeigers (Speicherzelle PTR) mittels AC

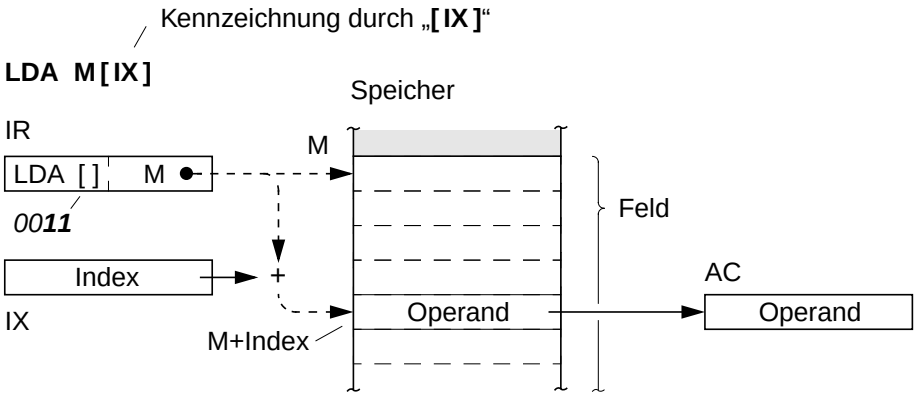
Konsequenz:

- Einbau eines speziellen Registers für die Adressierung
  → indizierte Adressierung

Indizierte Adressierung zur Adressierung von Datenfeldern

Funktion:

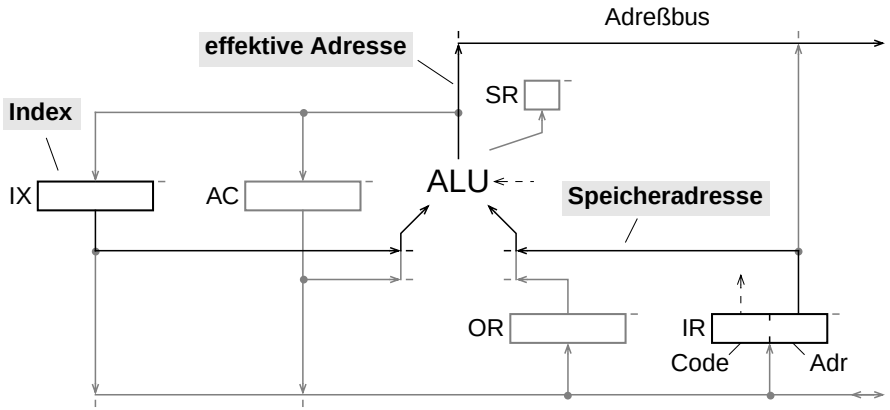
- Die **Feldanfangsadresse** steht im Befehl, eine **variable Adreßdistanz** als **Index** in einem **Indexregister** IX.
- Der **Operand** steht im Speicher (Feld).



Effektive Adresse = M + Inhalt von IX

Strukturerweiterung:

- **IX** (**i**nde**x** register) zur Speicherung einer **variablen Adreßdistanz**, bezogen auf die Anfangsadresse eines Datenfeldes.



Erweiterung der Befehlssatzes:

Befehl		Wirkung		z n c v	Code
LDX	M	lade	IX : = M	x x 0 0	0010 10 xx
STX	M	speichere	M : = IX	– – – –	0011 10 xx
ADDX	M	addiere	IX : = IX + M	x x x x	0100 10 xx
SUBX	M	subtrahiere	IX : = IX – M	x x x x	0101 10 xx
CMPX	M	vergleiche	IX – M wirkt nur auf cc!	x x x x	0110 10 xx

Anmerkung:

Das Indexregister eignet sich nicht nur zur indizierten Adressierung, sondern auch zur Schleifenzählung.

Beispiel: Polynomauswertung 2 mit **indizierter Adressierung**

```
N       DAT   3          ; Polynomgrad
FELD    RES   4          ; Koeffizienten a0,a1,...
X       RES   1          ; Argumentwert
P       RES   1          ; Polynomwert
          :
        LDX   #0         ; Index des Feldanfangs
        LDA   FELD[IX]   ; Koeffizientenzugriff a0
Loop    CMPX  N
        BEQ   Ende       ; Schleifenabbruch
        ADDX  #1
        MUL   X          ; erster Lauf: a0 * x
        ADD   FELD[IX]   ;              (a0 * x) + a1
        JMP   Loop
Ende    STA   P
```

Änderungen gegenüber der **Polynomauswertung 1**:

1. Die Speicherzelle PTR wird durch das Register IX ersetzt.
2. **Es entfallen** 4 von 10 Befehlen in der Programmschleife!!!

Zusammenfassung:

- **Speicherindirekte Adressierung**

  Vorteil: **Variable Adressen** (Pointer) sind verfügbar, d.h., das Rechnen mit Adressen ist möglich,
  (Anwendung: Listen, Parameterübergabe bei Unterprogrammen etc.)

  Nachteil: Die **Programmierung** ist **umständlich**:
  für das Rechnen mit Adressen und das „Schleifenzählen“ wird der AC benötigt.

- **Indizierte Adressierung**

  Vorteil: Die **Programmierung** ist **einfach**: a) das Indexregister IX und b) die zusätzlichen Befehle für IX entlasten den AC.

  Nachteil: Der **Index** ist als variabler Distanzwert **weniger „universell“** als eine variable Adresse.

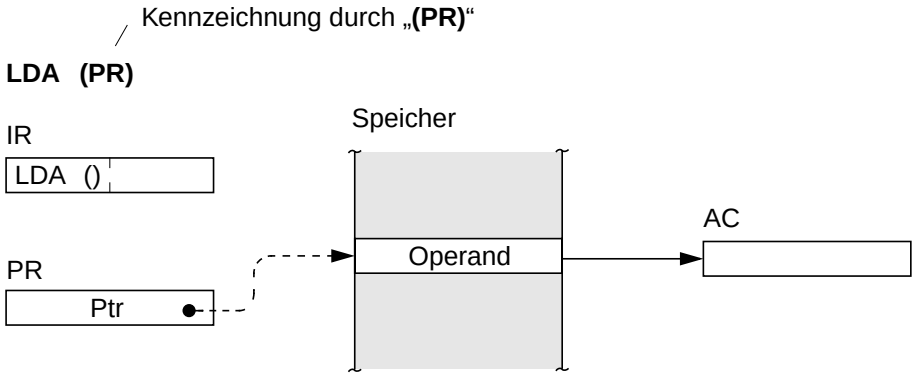
Kompromiss:

Registerindirekte Adressierung

Registerindirekte Adressierung für variable Adressierung

Funktion:

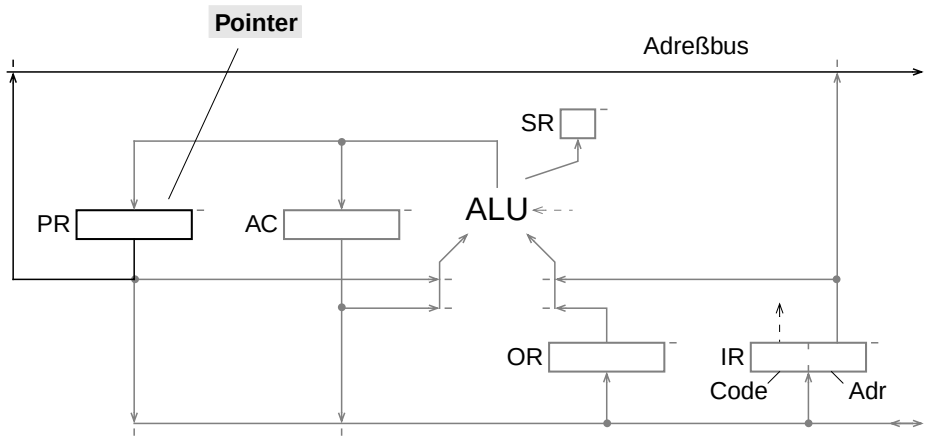
- Die **Operandenadresse** steht als **Pointer** in einem **Pointer-Register** PR.
- Der **Operand** steht im Speicher.



Effektive Adresse = Inhalt von PR

Strukturänderung:

- **PR** (**p**ointer **r**egister) zur Speicherung **variabler Adressen**, angebunden an die ALU und mit direktem Zugang zum Adreßbus.



Erweiterung der Befehlssatzes:

Befehl		Wirkung		z n c v	Code
LDP	M	lade	PR : = M	x x 0 0	
STP	M	speichere	M : = PR	– – – –	**im VIP nicht realisiert**
ADDP	M	addiere	PR : = PR + M	x x x x	
SUBP	M	subtrahiere	PR : = PR – M	x x x x	
CMPP	M	vergleiche	PR – M wirkt nur auf cc!	x x x x	

Vorteile:

- **Variable Adressen** (Pointer) sind verfügbar.
- Das **Pointer-Register PR** und die **Befehle** für PR entlasten den AC.

Beispiel: Polynomauswertung 3 mit **registerindirekter Adressierung**

```
N       DAT   3         ; Polynomgrad
FELD    RES   4         ; Koeffizientenfeld a0,a1,...
EndAdr  RES   1         ; Endadr. von Koeff.-Feld
X       RES   1         ; Argumentwert
P       RES   1         ; Polynomwert
         :
        LDA   #FELD
        ADD   N         ; Endeadr. ermittelt
        STA   EndAdr
        LDP   #FELD     ; Anfangsadr. als Pointer
        LDA   (PR)      ; Koeffizientenzugriff a0
Loop    CMPP  EndAdr
        BEQ   Ende      ; Schleifenabbruch
        ADDP  #1
        MUL   X         ; erster Lauf: a0 * x
        ADD   (PR)      ;              (a0 * x) + a1
        JMP   Loop
Ende    STA   P
```

→ Wie bei der indizierten Adressierung, hat die **Programmschleife nur 6 Befehle**!

1.7 Assemblierung

Erzeugung des **Maschinencodes** (grob), Programmzeile für Programmzeile:

- **Zerlegen der Programmzeile** (ASCII-String) in ihre Bestandteile (syntaktische Analyse).
- **Zuordnen von binären Operationscodes** zu den Mnemonen und Adressierungskennzeichen der Maschinenbefehle (dazu gibt es eine feste, prozessorspezifische **Zuordnungstabelle**).
- **Umwandeln von Zahlen** in ihre Interndarstellung.

  Wertermittlung für arithmetische und logische Ausdrücke in den Adreßfeldern (beim VIP-Assembler nicht implementiert).
- **Ersetzen der Symbole** in den Adreßfeldern durch ihre Werte.

  Ist ein Symbol zuvor im Namensfeld „definiert“ worden (**Rückwärtsadreßbezug**), so ist sein Wert bekannt. Wird es erst im Namensfeld einer nachfolgenden Programmzeile definiert (**Vorwärtsadreßbezug**), so muß ihm der Wert in der Maschinencodezeile rückwirkend zugewiesen werden.

  Hilfsmittel: Zwei Assemblierdurchgänge durch das Programm und Erstellen einer sog. **Symboltabelle** → 2-Phasen-Assembler.

Assemblercode für Fakultät von n (Änderungen gegenüber vorn: fett!):

```
        ORG   0
OutReg  EQU   255     ; Adr. von Ausg.-Register
n       DAT   5       ; Argument n
Fac     RES   1       ; Speicherplatz für n!

Start   LDA   #1
        STA   Fac     ; Fac = 1
Loop    LDA   n
        BEQ   Ende    ; while n ≠ 0 fahre fort
        MUL   Fac     ; Fac = n * Fac
        STA   Fac
        LDA   n
        SUB   #1      ; n = n - 1
        STA   n
        JMP   Loop
Ende    LDA   Fac
        STA   OutReg  ; Wert n! ausgeben
        HLT
        END   Start
```

Maschinencode für Fakultät von n:

```
          ORG   0            Adr.  Maschinencode
OutReg    EQU   255
n         DAT   5            0     0000 0000 00000101
Fac       RES   1            1     xxxx xxxx xxxxxxxx

Start     LDA   #1           2     0010 0001 00000001
          STA   Fac          3     0011 0000 00000001
Loop      LDA   n            4     0010 0000 00000000
          BEQ   Ende         5     0001 0111 00001100
          MUL   Fac          6     1111 0000 00000001
          STA   Fac          7     0011 0000 00000001
          LDA   n            8     0010 0000 00000000
          SUB   #1           9     0101 0001 00000001
          STA   n            10    0011 0000 00000000
          JMP   Loop         11    0001 0000 00000100
Ende      LDA   Fac          12    0010 0000 00000001
          STA   OutReg       13    0011 0000 11111111
          HLT                14    0000 0000 00000000
          END   Start
```

xx…x steht für „Inhalt unbestimmt“

Aufbau der Symboltabelle:

- Stand nach Bearbeitung der BEQ-Befehlszeile:

Symbol	verwendet	definiert	Wert
OutReg	–	x	255
n	x	x	0
Fac	x	x	1
Start	–	x	2
Loop	–	x	4
Ende	**x**	**–**	**?** ← 12

Anhang

Registertransferbeschreibung (RT-Beschreibung)

Festlegungen für die Beschreibung der funktionellen Abläufe im VIP (ohne Anspruch auf Vollständigkeit und Geschlossenheit im Sinne einer programmiersprachlichen Beschreibung):

1. Bezeichner für Speicherelemente:
   PC, IR, OR, AC, SR, IX (für die Register; ggf. erweiterbar, z.B. um PR, SP)
   RAM (für den Hauptspeicher)
2. Erweiterung der Bezeichner zur Unterteilung von Speicherelementen:
   RAM[i] (für die Anwahl der Speicherzelle mit der Adresse i)
   IR.Code, IR.Adr (für die Unterteilung von IR)
   AC_0, AC_{15} (als Beispiele für einzelne Registerbits, hier für LSB und MSB von AC)
   $IR_{15:8}$ oder $IR_{8:15}$ (als Beispiel für ein Bitfeld, hier für IR.Code)
   AC<0>, AC<15>, IR<15:8> (als Alternativschreibweisen zu AC_0, AC_{15}, $IR_{15:8}$)

3. Operatoren, realisiert durch Schaltnetze:
   $+, -, \wedge, \vee, \oplus, \times 2, \times 2^{-1}$, … (alle arithmetischen und logischen Operationen der ALU)
   $=, \neq, <, \leq, \geq, >$ (für den Vergleich)
4. Zuweisung als taktsynchroner Transport:
   := (zur Wertübernahme in ein Speicherelement, links des Symbols, z.B. AC := …)
5. Registertransferoperation:
   Speicherelement := Ergebnis der Verknüpfung von Speicherelementinhalten,
   z.B. AC := AC + OR als Transport des Ergebnisses einer ALU-Verknüpfung nach AC
6. Parallelität und Sequentialität von Registertransferoperationen:
   , (für taktparallele Vorgänge, z.B. IR := RAM[IR.Adr], PC := PC + 1)
   ; (für taktsequentielle Vorgänge, z.B. OR:=RAM[IR.Adr]; OR:=RAM[OR]; AC:=OR)
7. Steuerkonstrukte:
   if, else, while, do, goto, … (für Verzweigungen und Schleifen),
   z.B. PC := IR.Adr if SR.z = 0 als Wirkung des bedingten Sprungbefehls BNE