I. GRUNDLAGEN

Aufgaben der Hardware:

Ein- und Ausgabe von Daten
Verarbeiten von Daten
Speichern von Daten

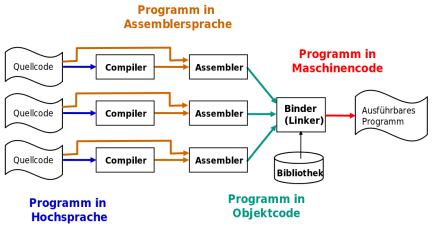
Klassische Hardwarekomponenten:

Ein- und Ausgabe
Hauptspeicher
Rechenwerk
Leitwerk

II. ANFORDERUNGEN HÖHERER PROGRAMMIERSPRACHEN

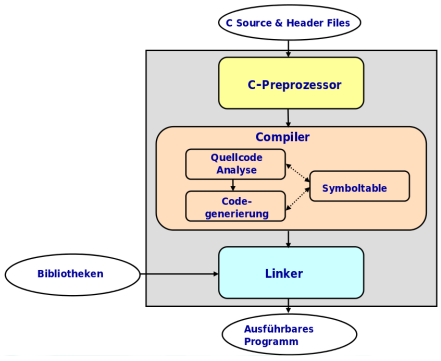
Begriffe:

Maschinensprache: Für Prozessor verständliche Anweisungsrepräsentation, z.B. 00101101001110101
Assemblersprache: Für Menschen verständliche Maaschinensprache, z.B. `add $s2, $s1, $s0`
Assembler: Übersetzt Assemblersprache eindeutig in Maschinensprache
Objektcode: Maschinenprogramm mit ungelösten externen Referenzen
Binder/Linker: Löst ungelöste Referenzen auf, verbindet alles zu einem ausführbaren Maschinenprogramm



Programmiersprache C:

Zwischenstellung zwischen Assembler und Hochsprache
hohe Portabilität trotz guter Architekturanpassung
einfache Programmierung
Datentypen: `char, int, float, double`
Kontrollstrukturen: Entscheidungen, Schleifen, Blöcke, Unterprogramme
Zeiger als Parameter möglich



C - Datentypen:

`char`: Ein Zeichen, meist 1 Byte
`int`: Integerzahl, 2 oder 4 Byte
`float`: Gleitkommazahl, meist 4 Byte
`double`: Gleitkommazahl, meist 8 Byte

C - Operatoren:

`*`: Multiplikation (`x*y`)
`/`: Division (`x/y`)
`%`: Modulo (`x%y`)
`+`: Addition (`x+y`)
`-`: Subtraktion (`x-y`)
`+` und `-` auch als Prä- und Postfix, alle auch als assign (`=` anhängen)

C - Bit-Operatoren:

`~`: Bitwise NOT (`~x`)
`<<`: links schieben (`x<<y`)
`>>`: rechts schieben (`x>>y`)
`&`: bitweise AND (`x&y`)
`^`: bitweise XOR (`x^y`)
`|`: bitweise OR (`xy|`)
alle auch als Assign (`=` anhängen)

C - Vergleichsoperatoren:

`>,<`: größer, kleiner als (`x>y, x<y`)
`>=,<=`: größergleich, kleinergleich als (`x>=y, x<=y`)
`==,!=`: gleich, ungleich (`x==y, x!=y`)

C - Spezialoperatoren:

Auswahloperator: `z = (a < b) ? a : b` (`z=a`, falls `a<b`, sonst `z=b<`)

C - Operatoren-Priorität

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

C - Kontrollstrukturen

```
if (Bedigung) { Aktionen_if } else { Aktionen_else }
switch (var) { case a: ... break; ... default: ... break; }
while (Bedigung) { ... }
for (init; Bedingung; reinit) { ... }
do { ... } while (Bedingung)
```

C - Programmaufbau

1. Präprozessor-Anweisungen:
   (a) `#include <stdio.h>` (Bibliotheken einbinden)
   (b) `#include "modul.h"` (Module einbinden)
   (c) `#define COLOR blau` (Globale Textersetzung)
2. Globale Deklarationen/Definitionen:
   (a) `int i;` (Deklaration)
   (b) `int j = 13;` (Definition)
   (c) `int fakultaet (int n);` (Funktionsprototyp)
3. Funktionen/Programmstruktur

   `int fakultaet (int n) { ... }`
   jedes Programm enthält Funktion `void main(...) { ... }`
   Unterprogramm = Funktion
   Programmstart: `main` wird aufgerufen
   Rekursion ist zulässig

C - Parameterübergabe

1. Call by Value: Normalfall, Kopie des Parameters wird an Funktion übergeben, bei Änderung keine Auswirkung beim Aufrufer
2. Call by Reference: Mit Zeigern umsetzbar, selbe Speicheradresse wie Aufrufer

C - globale und lokale Variablen

Global: Sind gesamtem Programm bekannt (zu vermeiden)
Lokal: Nur in Block deklariert

C - Speicherklassen

`auto`: lokale Variablen
`register`: wird in CPU-Register gespeichert, nur für zeitkritische Variablen zu verwenden
`static`: statischer Speicherplatz
`extern`: globale Variable

C - Zeiger und Vektoren

Pointer: Enthält Adresse, die auf Daten verweist
`int* p` (`p` ist Zeiger auf `int`)
`a = 3; p = &a` (`p` enthält Adresse von `a`)
`int b = *p + 1` (=4)

```
int *p;
int *q;
int a = 3;
int b;

p = &a;
b = *p + 1;
q = (int*) 0x8010
```

	Adresse	Inhalt
p	...	0x8004
a	0x8004	3
b	...	4
q	...	0x8010
	0x8010	

III. ZAHLENDARSTELLUNG

Zahlensysteme – Stellenwertsystem

Darstellung einer Zahl durch Ziffern z_i – Stellenwert $i$te Position: $i$te Potenz der Basis b
Wert $X_b = \sum_{i=-m}^{n} z_i b^i$
Wichtige Zahlensysteme: Dual-, Oktal-, Dezimal-, Hexadezimalsystem

	Dual	Oktal	Dezimal	Sedezimal
0	0	0	0	0
1	1	1	1	1
2		2	2	2
3		3	3	3
4		4	4	4
5		5	5	5
6		6	6	6
7		7	7	7
8			8	8
9			9	9
10				A
11				B
12				C
13				D
14				E
15				F

Umwandlung von Dezimal zu Basis b

1. euklidischer Algorithmus:
   (a) Berechne p mit $b^p \leq Z < b^{p+1}$, setze $i = p$
   (b) Berechne $y_i = Z_i \text{ div } b^i$, $R_i = Z_i \bmod b^i$
   (c) Wiederhole (b)H für $i = p-1, \ldots,$ ersetze dabei Z durch R_i, bis $R_i = 0$ oder b^i klein genug ist

$$2^3 \leq 13 < 2^4$$
$$13 : 2^3 = 1 \text{ Rest } 5$$
$$5 : 2^2 = 1 \text{ Rest } 1$$
$$1 : 2^1 = 0 \text{ Rest } 1$$
$$1 : 2^0 = 1 \text{ Rest } 0$$
$$\rightsquigarrow Z = 13_{10} = 1101_2$$

2. Horner-Schema:
   (a) ganzzahliger Teil: 15741_{10} in Hexadezimal:

$$15741_{10} : 16 = 983 \text{ Rest } 13 \ (= D_{16})$$
$$983_{10} : 16 = 61 \text{ Rest } 7 \ (= 7_{16})$$
$$61_{10} : 16 = 3 \text{ Rest } 13 \ (= D_{16})$$
$$3_{10} : 16 = 0 \text{ Rest } 3 \ (= 3_{16})$$
$$\rightsquigarrow Z = 15741_{10} = 3D7D_{16}$$

   (b) Nachkommateil: $0,233_{10}$ in Hexadezimal:

$$0,233_{10} * 16 = \underline{3},728$$
$$0,728_{10} * 16 = \underline{11},648$$
$$0,648_{10} * 16 = \underline{10},368$$
$$0,368_{10} * 16 = \underline{5},888$$
$$\rightsquigarrow Z = 0,233_{10} \approx 0,3BA5_{16}$$

Umwandlung Basis b zu Dezimal

Einzelne Stellen nach Stellenwertgleichung addieren

$$101101,1101_2 = 2^{-4} + 2^{-2} + 2^{-1} + 2^0 + 2^2 + 2^3 + 2^5 = 45,8125_{10}$$

Umwandlung Basis b_1 zu Basis b_2

1. Umwandlung über Dezimalsystem
2. Ist eine Basis Potenz der anderen, so können mehrere Stellen zu einer Ziffer zusammengefasst werden

$$0110100,110101_2 = 0011\ 0100,1101\ 0100 = 34,D4_{16}$$

Darstellung negativer Zahlen

1. Betrag und Vorzeichen: Erstes Bit von Links ist Vorzeichen, Rest ist Betrag (`0001 0010 = 18, 1001 0010 = -18`)

   Vorteile: Symmetrischer Zahlenbereich
   Nachteile: Darstellungsänderung bei Bereichserweiterung, gesonderte Vorzeichenbehandlung bei Addition und Subtraktion, doppelte Darstellung der Null

2. Einerkomplement: Negative Zahl = NOT(positive Zahl)

   ```
   0000 = 0    1111 = -0
   0001 = 1    1110 = -1
   0010 = 2    1101 = -2
   0011 = 3    1100 = -3
              ...
   ```

   Vorteile: Symmetrischer Zahlenbereich, keine gesonderte Betrachtung des ersten Bits
   Nachteile: doppelte Darstellung der Null

3. Zweierkomplement: = Einerkomplement + 1

   ```
   0000 = 0    1111 = -1
   0001 = 1    1110 = -2
   0010 = 2    1101 = -3
   0011 = 3    1100 = -4
              ...
   ```

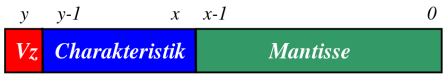
   Vorteile: Wie Einerkomplement, eindeutige Null
   Nachteile: Asymmetrischer Zahlenbereich (eine negative Zahl mehr)

4. Exzess-Darstellung: Verschiebung nach oben derart, dass kleinste negative Zahl die Darstellung `0...0` hat

Darstellung von Kommazahlen

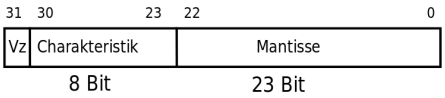
1. Festkommazahlen: Komma sitzt an einer festen Stelle
2. Gleitkommazahlen: $X = \pm\text{Mantisse} * b^{\text{Exponent}}$ (b fest)

   $X = (-1)^{\text{Vorzeichen}} * (0, \text{Mantisse}) * b^{\text{Exponent}}$
   $\text{Exponent} = \text{Charakteristik} - b^{(y-1)-x}$

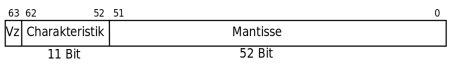


3. IEEE-Standard:

   (a) 32-Bit:



   (b) 64-Bit:



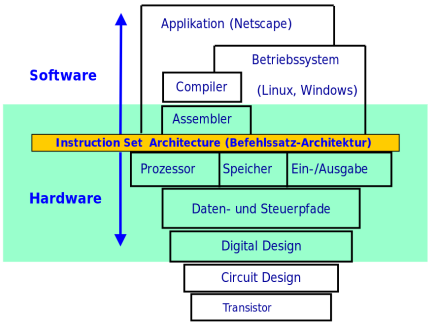
Codierungen

1. BCD: Dezimalzahl ziffernweise als Binärzahl (= *Tetrade*) codieren:

   ```
   8127 = 1000 0001 0010 0111
   ```

   Nachteil: Verbraucht viel Speicher, ungeschickt zum Rechnen
2. ASCII: 7-Bit-Codierung zur Textdarstellung
3. Unicode: Weltweit genormte Codierung aller Zeichen (wegen der vielen inkompatiblen ASCII-Derivaten)

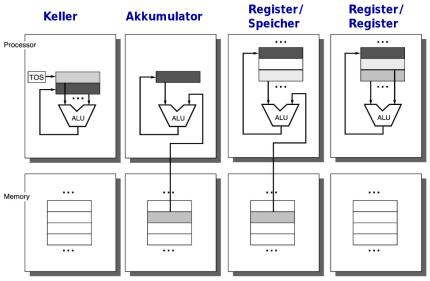
IV. BEFEHLSSATZARCHITEKTUR



ISA – Aufgaben

Wie werden Daten repräsentiert?
Wo werden Daten gespeichert?
Welche Operationen können auf den Daten ausgeführt werden?
Wie werden die Befehle codiert?
Wie wird auf die Daten zugegriffen?
⇝ abstrahiert Hardware für den Maschinenprogrammierer

Ausführungsmodelle



Ausführungsmodell – Register-Register

Alle Operanden und Ergebnis stehen in Allzweckregistern
Load/Store: Bestimmte Befehle holen Operanden aus Hauptspeicher, schreiben Inhalte von Registern in Speicher
Dreiadressformat

```
load R2,A   R2<-mem[A]
load R3,B   R3<-mem[B]
add R1,R2,R3   R1<-R2+R3
store C,R1   mem[C]<-R1
```

Vorteile: Einfaches und festes Befehlsformat, einfaches Code-Generierungsmodell, etwa gleiche Ausführungszeit der Befehle
Nachteile: Höhere Anzahl von Befehlen im Vergleich zu Architekturen mit Speicherreferenzen, längere Programme

Ausführungsmodell – Register-Speicher

Ein Operand im Speicher, ein Operand im Register, Ergebnis in Speicher oder Register
Explizite Adressierung mit/ohne Überdeckung
Zweiadressformat

```
add A,R1   mem[A]<-mem[A]+R1
add R1,A   R1<-R1+mem[A]
```

Vorteile: Zugriff auf Daten ohne vorherige Ladeoperationen, Befehlsformat-Kodierung ⇝ höhere Code-Dichte
Nachteile: Keine gleiche Operanden-Behandlung bei Überdeckungen, Taktzyklen pro Instruktion von Adressrechnung abhängig

Ausführungsmodell – Akkumulator-Register

Akkumulator: Ausgezeichnetes Register, dient als Quelle eines Operanden und als Ziel für das Resultat (zweistellige Operationen)

Implizite und überdeckte Adressierung

Spezielle Befehle ermöglichen Operanden-Transport

Einadressformat

```
 add A   acc<-acc+mem[A]
addx A   acc<-acc+mem[A+x]
add R1   acc<-acc+R1
```

Ausführungsmodell – Keller

Operanden einer zweistelligen Operation stehen auf den obersten zwei Kellerelementen

Ergebnis wird auf Keller abgelegt

Implizite Adressierung über Kellerzeiger (`tos`)

Überdeckung

Nulladressformat

```
add  tos<-tos+next
```

Ausführungsmodelle – Übersicht

```
C=A+B; D=C-B
```

Register-Register	Register-Speicher	Akkumulator	Keller

`load Reg1,A`
`load Reg2,B`
`Add Reg3,Reg1,Reg2`
`store C,Reg3`
`load Reg1,C`
`load Reg2,B`
`sub Reg3,Reg1,Reg2`
`store D,Reg3`
	`load Reg1,A`
	`add Reg1,B`
	`store C,Reg1`
	`load Reg1,C`
	`sub Reg1,B`
	`store D,Reg1`
		`load A`
		`add B`
		`store C`
		`load C`
		`sub B`
		`store D`
			`push B`
			`push A`
			`add`
			`pop C`
			`push B`
			`push C`
			`sub`
			`pop D`

Architektur – Datentypen

= Datenformat + inhaltliche Interpretation

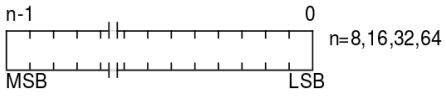
Alternative: Datentyparchitektur (Daten führen Typinformation mit sich)

Datentyp nicht von Hardware unterstützt ⇝ Programm muss Datentyp auf elementare Datentypen zurückführen

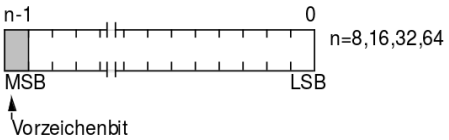
Standardformate:

1. Byte: 8 Bit
2. Halbwort: 16 Bit
3. Wort: 32 Bit
4. Doppelwort: 64 Bit

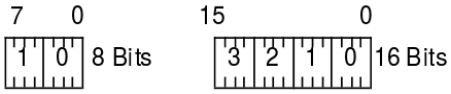
Vorzeichenlose Dualzahl:



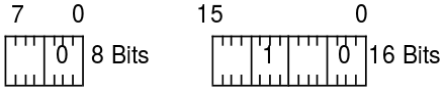
2er-Komplement (signed Integer):



BCD (gepackt): ein Halbbyte codiert eine Zahl

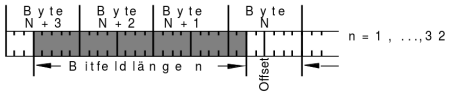


BCD (ungepackt): ein Byte codiert eine Zahl



Gleitkommazahlen: Siehe IEEE-Standard oben

Bitfeld: Darstellung/Verarbeitung von Bitvektoren, vorzeichenlosen Dualzahlen, Zweierkomplementzahlen - Darstellung durch Byte-Adresse und Bitfeld-Offset



String: Aufeinanderfolgend gespeicherte Bytes, enthalten meist ASCII-Zeichen

Speicheradressierung – Datenzugriff

1. Byte-adressierbarer Speicher: Jedes Byte ist über eine bestimmte Adresse adressierbar
2. Wort-organisierter Speicher: Zugriffsbreite = Datenbusbreite (32/64/... Bit)

Speicheradressierung – Alignment

Data Alignment: Datum (s Bytes) ist ausgerichtet abgelegt $\Leftrightarrow$ seine Adresse A ist derart, dass $A \bmod s = 0$

Data Misalignment: Daten an beliebiegen Adressen gespeichert

> **Vorteile**: Lückenlose Speichernutzung
> **Nachteile**: zusätzliche Speicherzugriffe nötig

Little Endian: Niedrigstwertigstes Byte an der niedrigsten Adresse

Big Endian: Niegrigstwertigstes Byte an der höchsten Adresse

Speicheradressierung – Adressierungsarten

1. Programmadresse: Im Programm vorliegende Adressen (Prozessor erzeugt aus Programmadressen Prozessadressen mittels Indexmodifikation/Substitution/relativer Adressierung/offener Basisadressierung)
2. Prozessadresse (effektive Adresse): Vom Prozessor verwendet (Prozessor erzeugt nach OS-Angaben aus Prozessadressen Maschinenadressen mittels verdeckter Basisadressierung/Seitenadressierung) - Grund: beliebige Lage des Programms und seiner Werte, partielle Lagerung im Speicher
3. Maschinenadresse: Vom Prozessor gegenüber Hauptspeicher verwendet

Instruction Set

legt Grundoperationen eines Prozessors fest

Befehlsarten:

1. Transport
2. Arithmetik/Logik
3. Schieben/Rotieren
4. Multimedia
5. Gleitkomma
6. Programmsteuerung
7. Systemsteuerung
8. Synchronisation

Instruction Set – Formate

Befehlsformat legt Befehlscodierung fest

Befehlscodierung: `[opcode] [parameter1] ...`

Adressformate: vier Befehlssatzklassen:

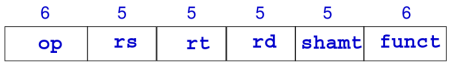
1. Dreiadressformat: `[opcode] [dest] [src1] [src2]`
2. Zweiadressformat: `[opcode] [dest/src1] [src2]`
3. Einadressformat: `[opcode] [src]`
4. Nulladressformat: `[opcode]`

Instruction Set – MIPS-Prozessor

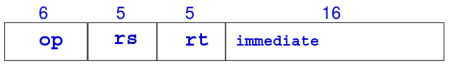
Alle Befehle 32 Bit lang

Befehlstypen:

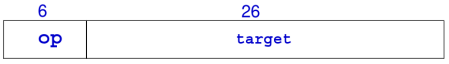
1. **Typ R**: Register-Register-Befehle



2. **Typ I**: Lade-/Speicher-Befehle



3. **Typ J**: Sprungbefehle



Abkürzungen:

`I`	Immediate (direkt)
`J`	Jump (Sprung)
`R`	Register
`op`	6 Bit, Opcode des Befehls
`rs`	5 Bit, Kodierung eines Quellenregisters/Zielregisters
`immediate`	16 Bit, unmittelbarer Wert/Adressverschiebung
`target`	26 Bit, Sprungadresse
`rd`	5 Bit, Kodierung des Zielregisters
`shamt`	5 Bit, Größe einer Verschiebung (shift amount)
`funct`	6 Bit, Codierung der Funktion (function)

Adressierung – Berechnung

Adressierungsarten: Verschiedene Möglichkeiten, die Adresse eines Operanden/Sprungziels zu berechnen

Früher: Adressen in Befehlen absolut vorgegeben ⇝ Programme lageabhängig

Heute: *dynamische Adressberechnung*:
Programmadresse ⇝ logische Adresse ⇝ physikalische Adresse

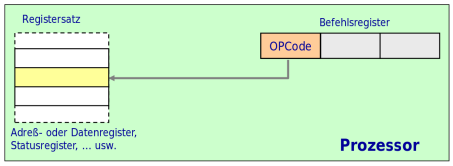
Adressierungsarten

1. Register-Adressierung
   (a) implizit: Flag
   (b) explizit
2. einstufige Speicher-Adressierung
   (a) unmittelbar
   (b) direkt: absolut, Zero-Page, Seiten(-Register)
   (c) Register-indirekt
   (d) indiziert: Speicher-relativ, Register-relativ, Register-relativ mit Index
   (e) Programmzähler-relativ
3. zweistufige Speicher-Adressierung
   (a) indirekt absolut
   (b) indirekt Register-absolut
   (c) indirekt indiziert: Speicher-relativ, Register-relativ, Register-relativ mit Index
   (d) indiziert indirekt
   (e) indirekt Programmzähler-relativ

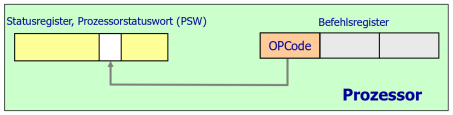
Adressierungsarten – Register-Adressierung

Operanad steht bereits im Register ⇝ kein Speicherzugriff nötig

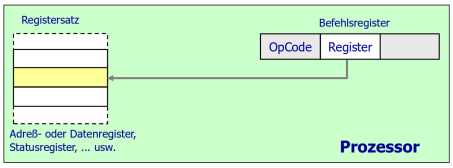
Implizite Adressierung: Nummer des Registers ist im Opcode-Feld codiert enthalten
Beispiel: `LSRA` (Verschiebe Akkumulatorinhalt eine Bitposition nach rechts)



Flag-Adressierung: Spezialfall der impliziten Adressierung: Nur ein Bit (=Flag) wird im Register angesprochen
Beispiel: `SEI/CLI` (set/clear interrupt flag)



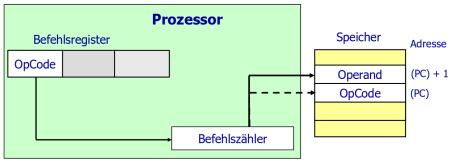
Explizite Adressierung: Registermnummer wird im Operandenfeld angegeben
Beispiel: `DEC R0` (Dekrementiere Register R0)



Adressierungsarten – Einstufige Speicher-Adressierung

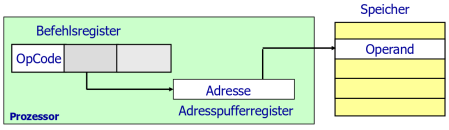
Eine Adressberechnung ist zur Ermittlung der effektiven Adresse nötig ⇝ keine mehrfachen Speicherzugriffe zur Adressermittlung

Unmittelbare Adressierung: Befehl enthält nicht Adresse des Operanden, sondern Operand selbst (Opcode und Operand stehen hintereinander im Speicher)
Beispiel: `LDA #$A3` (Lade Akkumulator mit Sedezimalwert `$A3`)



Direkte Adressierung: Befehl enthält nach Opcode logische Adresse des Operanden, aber keine Vorschriften zur Manipulation

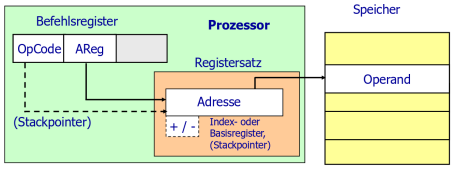
1. **Absolute Adressierung**: Speicherwort enthält vollständige (absolute) Operandenadresse
   Beispiel: `JMP $07FE` (Springe zur Adresse `$07FE`)



2. **Seitenadressierung**: Im Befehl nur Kurz-Adresse (niederwertige Teil der Adresse).
   Zero-Page: Höherwertiger Teil: `0`-Bits
   Seiten-Register: Höherwertiger Adressteil wird in Prozessorregister bereitgestellt

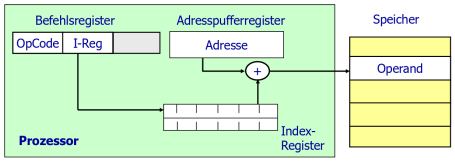
Register-indirekte Adressierung: Im Opcode angegebenes Adressregister enthält Adresse des Operanten (=*Pointer*)
Beispiel: `LD R1, (A0)` (Lade Register `R1` mit Inhalt des in `A0` angegebenen Speicherwortes)

Im Register stehende Adresse oft Anfang/Ende eines Tabellenbereiches, deswegen Hilfsmethoden:
Postinkrement: Nach Befehlsausführung Registerinhalt inkrementieren und auf nächste Speicherzelle zeigen (z.B. `INC (R0)+`: Inkrementiere Speicherwortinhalt, das von `R0` adressiert wird, danach Inhalt von `R0`)
Predekrement: Vor Befehlsausführung Registerinhalt erniedrigen und auf vorhergehende Speicherzelle zeigen (z.B. `CLR -(R0)`: Dekrementiere Inhalt von `R0`, lösche dann durch `R0` adressiertes Speicherwort)

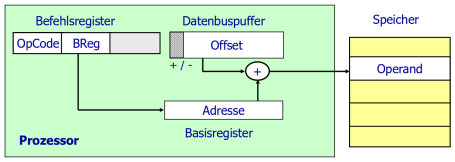


Indizierte Adressierung: Effektive Adresse wird durch Addition eines Registerinhalts zu angegebenem Basiswert berechnet

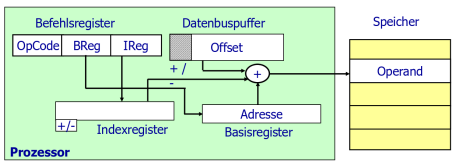
1. **Speicher-relative Adressierung**: Basiswert wird als absolute Adresse im Befehl vorgegeben
   Beispiel: `ST R1,$A704 (R0)` (Speichere Inhalt von `R1` in Speicherwort, dessen Adresse man durch Addition des Inhalts von `R0` zu Basis `$A704` erhält)



2. **Register-relative Adressierung**: Basiswert befindet sich in Basisregister, verwiesen im `BReg`-Feld des Opcodes
   Beispiel: `CLR $A7 (B0)` (Lösche Speicherwort, dass man durch Addition von `$A7` zu `B0` erhält)



3. **Register-relative Adressierung mit Index**: Basiswert in Basisregister, Addition des Indexregister-Inhalts (hat ggf. Autoinkrement/Autodekrement), ggf. Angabe eines zusätzlichen Offsets im Befehl (wird hinzuaddiert)
   Beispiel: `DEC $A7 (B0) (I0)+` (Dekrementiere Speicherwort, dessen Adresse `I0+B0+$A7` ist, inkrementiere danach den Inhalt von `I0`)



Programmzähler-relative Adressierung: Effektive Adresse = Befehlszählerstand + Offset (im Befehl angegeben) - erlaubt Programme im Hauptspeicher zu verschieben
Beispiel: `LBRA $7FFF` (verzweige *unbedingt* zu Speicherzelle, deren Adressdistanz zu Programmzähler `$7FFF` ist)