Technische Universität Braunschweig

Informatik für Ingenieure – VL 11

Übersicht

Letztes Mal:

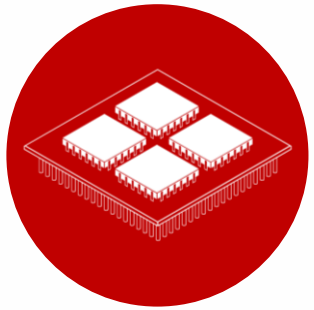
- Speicher
- Einführung Mikrorechner

Heute:

- Mikrorechner
- Adressierung
- Typische Befehle

Teile des heutigen Vortrags basiert auf der Vorlesungen von
Prof. H Michalik (TU Braunschweig)

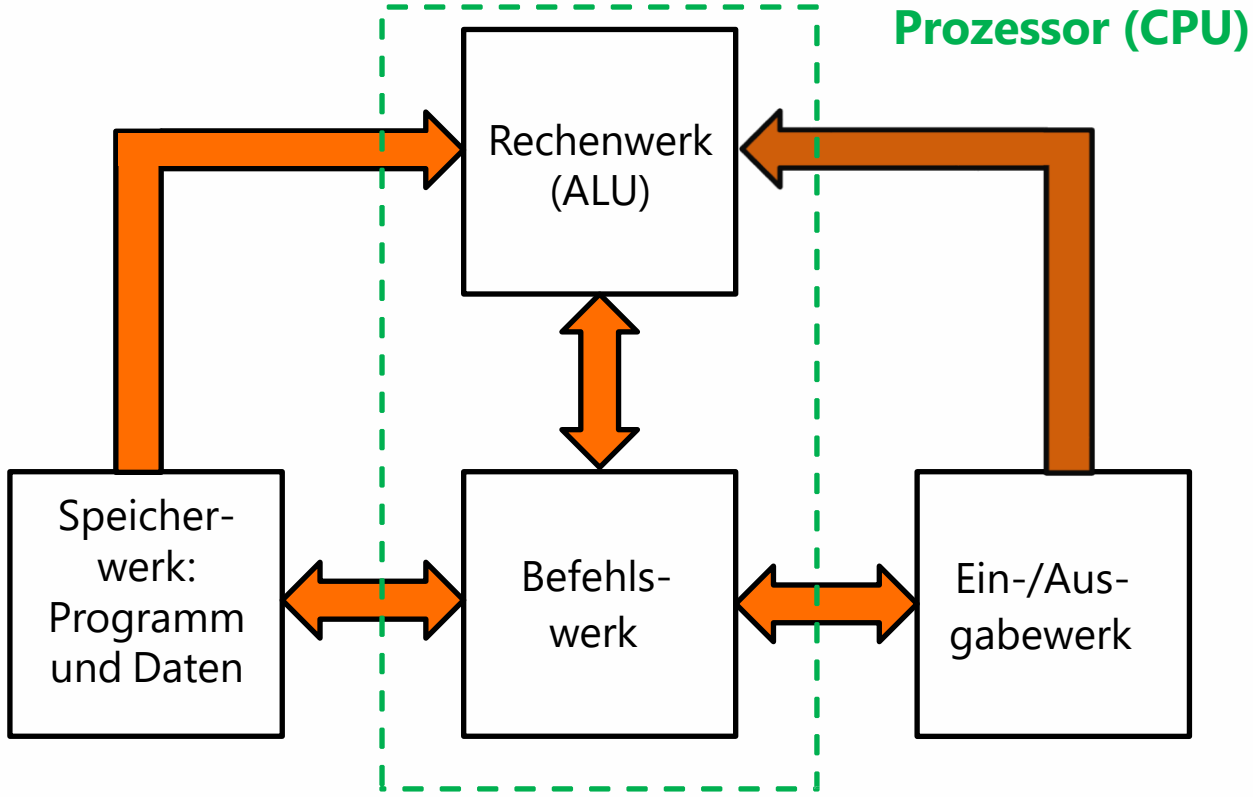
Mikrorechner



Die ‚von Neumann' Architektur

Im Jahre 1945 hat der Mathematiker **John von Neumann** eine Architektur vorgeschlagen, deren Grundstruktur die Computertechnik bis heute geprägt hat. Die von Neumann Architektur ist ein Schaltungskonzept zur Realisierung **universeller Rechner**

von Neumann'scher Universalrechenautomat



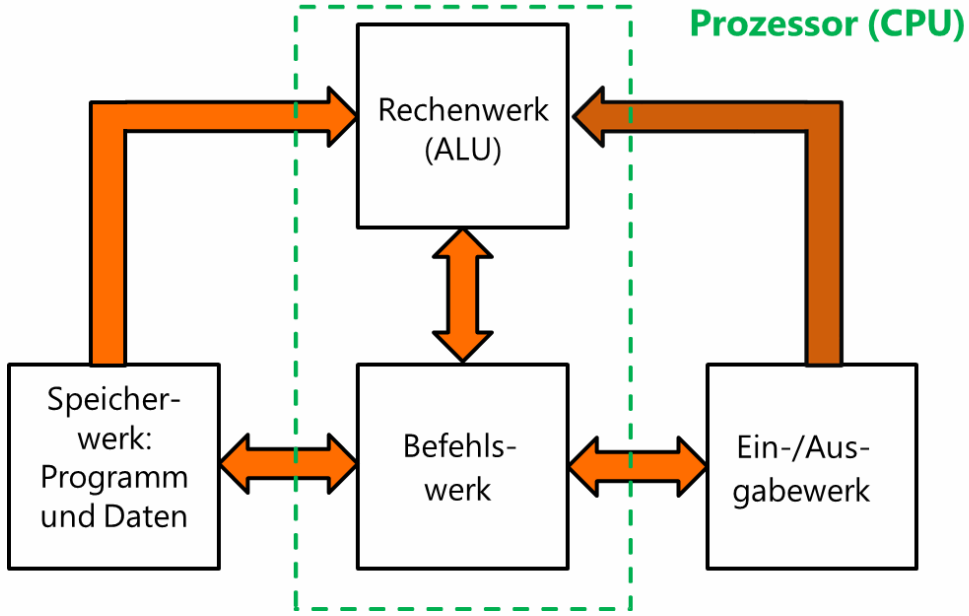
Mikrorechner- Übersicht

Die CPU besteht aus dem Befehls- und Rechenwerk.

Das **Befehlswerk** (auch Steuerwerk gennant) organisiert den Speicherzugang zu Befehlen und Daten, entschlüsselt die Befehle und löst die Fortschaltung zum nächsten Befehl und die einzelnen Operationen aus.

Das **Rechenwerk** ist eine autonome Ablaufsteuerung z.B. für die Bearbeitung arithmetischer Operationen, das mit dem Befehlsablaufnetz kommuniziert.

Beide Werke sind dem Prinzip nach Schaltwerke nach Kapitel 3, die man allgemein als Mikroprogrammsteuerwerke bezeichnet.



von Neumann vs. einfache Maschine

Wie unterscheidet sich die von Neumann'sche Computer Architektur von einer einfachen Rechenmaschine?

Die Wahl des nächsten Befehls kann von dem aktuellen Ergebnis aus der ALU abhängen:

Konditionalabfragen möglich: *if* <Wert a> grösser 0, *then* ...

Schleifen mit variabler Anzahl an Durchläufen möglich

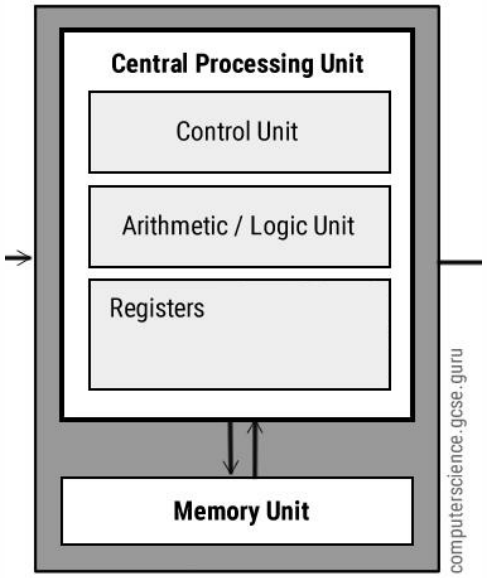
Gleichartige Speicherung von Daten und Befehlen

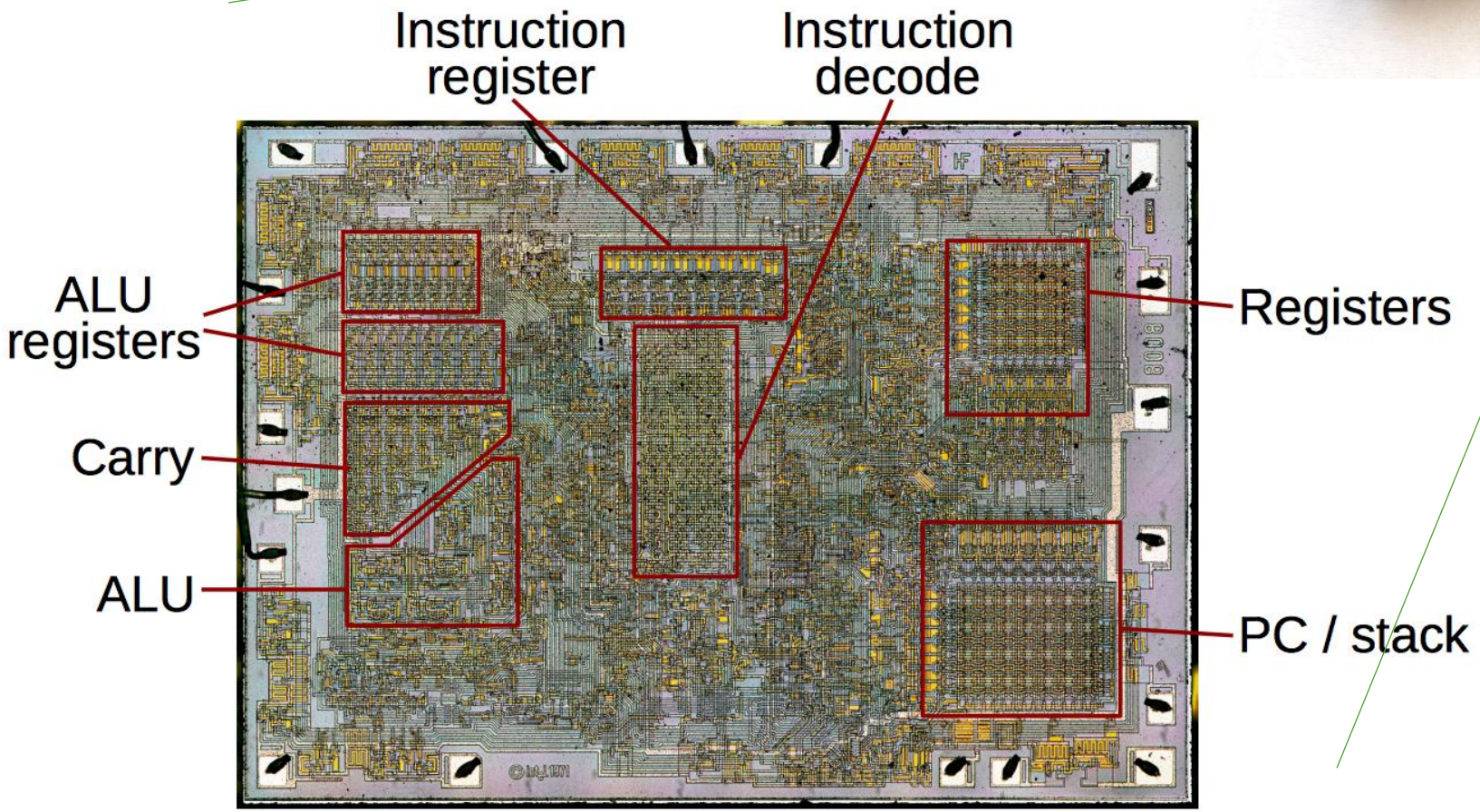
Befehlsadressen können wie Daten abgespeichert und zu einem späteren Zeitpunkt zurückgeholt werden, um den Programmablauf an einer bestimmten Stelle weiterlaufen zu lassen. → Unterprogramme (*Subroutines*) sind möglich

Programme können andere Programme erzeugen

Compiler

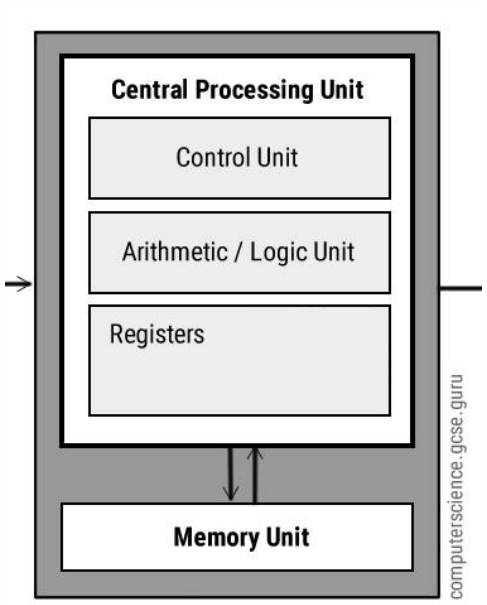
Beispiel: Intel 8008 (1972)

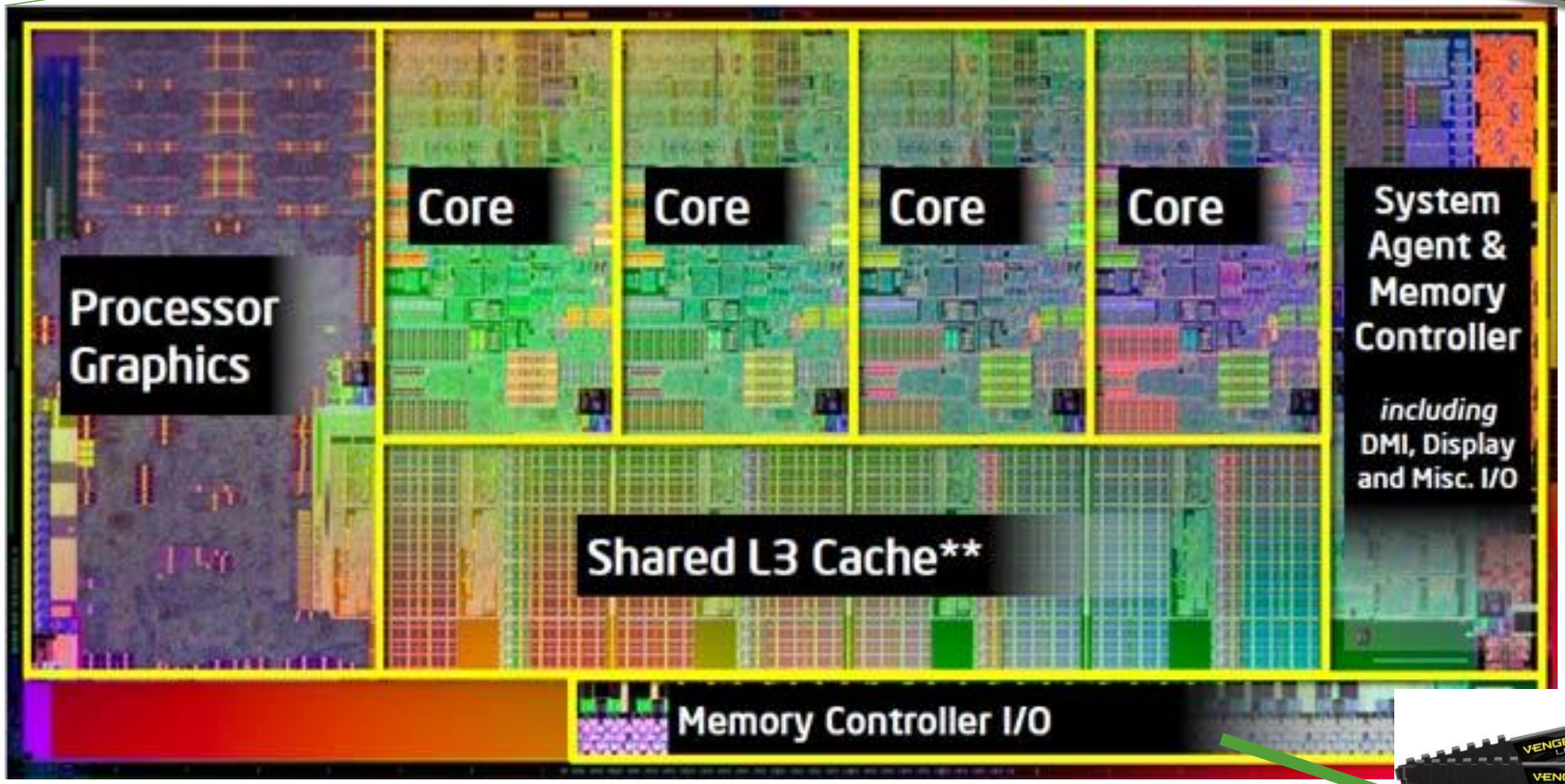


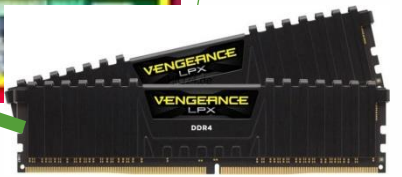


http://www.righto.com/2016/12/die-photos-and-analysis-of_24.html

Beispiel: Intel Core i7 Haswell Refresh (2014)

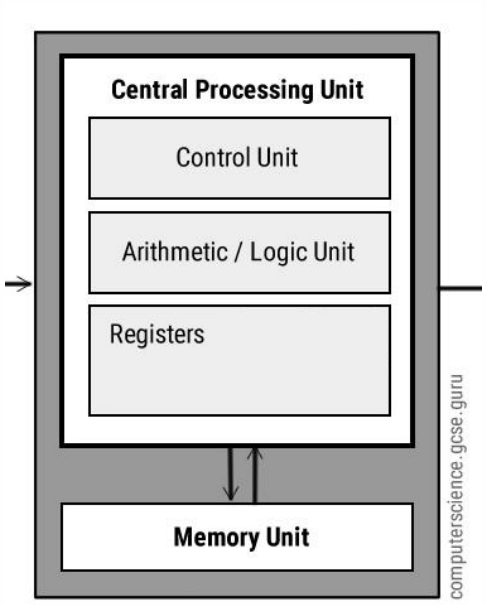


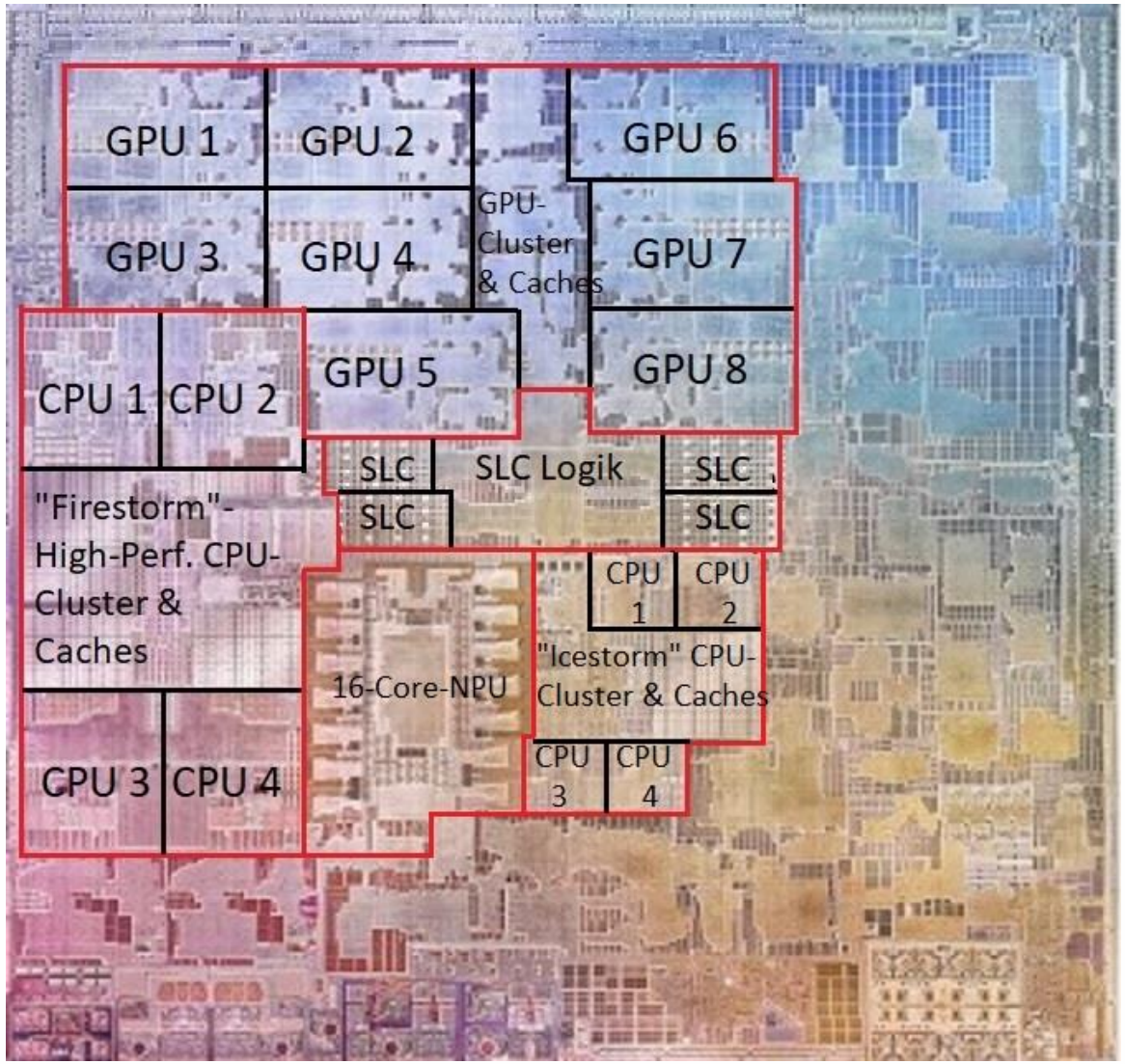




https://www.guru3d.com/articles-pages/core-i7-4790-processor-review,3.html

Beispiel: Apple M1 System-on-Chip (2021)





CPU – Central **Processing** Unit

GPU – **Graphics** Processing Unit

NPU – **Neural** (Network) Processing Unit

SLC – System-level **Cache**

Sehr einfaches Beispiel für einen Mikrorecher

Hardware-Merkmale

- 4 Speicherplätze mit einer Größe von je 4 Bit
- 2 Register mit einer Größe von je 4 Bits
- Eine ALU, die subtrahieren und addieren kann

Register	
Adresse	Inhalt
0	
1	

Datenspeicher	
Adresse	Inhalt
00	
01	
10	
11	

Ein Befehlssatz mit **5 Befehlen**

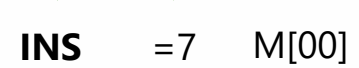
LD	Laden von Speicheradresse x (d.h., M[xx]) in Register y (d.h., R[y]):	000 xx y
ST	Speichern von R[x] in M[yy]:	001 x yy
ADD	Addiere die Registerinhalte und speichere das Ergebnis in R[x]:	010 x
SUB	Subtrahieren den Inhalt von R[0] von R[1] und speichern das Ergebnis in R[x]:	011 x
INS	Einfügen der 4-Bit-Zahl n in M[xx]	100 nnnn xx

Ein Computerprogramm für unsere Architektur

Befehlssatz

LD von M[xx] zu R[y]:	000 xx y
ST von R[x] zu M[yy]:	001 x yy
ADD Registerinhalt und Ergebnis in R[x] speichern:	010 x
SUB R[0] Inhalt von R[1] und Ergebnis in R[x] speichern	011 x
INS die 4-bit Nummer n in M[xx]	100 nnnn xx

Was bedeutet: 100 0111 00



// Programm auf Maschinenebene, das die Zahlen 7 und 9 addiert

Einfügen von 7 in M[00]: 100 0111 00
Einfügen von 9 in M[01]:
Laden von M[00] zu R[0]:
Laden von M[01] zu R[1]:
Addiere Register zu R[0]:
Speichern von R[0] in M[10]:

Register	
Adresse	Inhalt
0	
1	

Datenspeicher	
Adresse	Inhalt
00	
01	
10	
11	

Ein Computerprogramm für unsere Architektur

Befehlssatz

LD von M[xx] zu R[y]:	000 xx y
ST von R[x] zu M[yy]:	001 x yy
ADD Registerinhalt und Ergebnis in R[x] speichern:	010 x
SUB R[0] Inhalt von R[1] und Ergebnis in R[x] speichern	011 x
INS die 4-bit Nummer n in M[xx]	100 nnnn xx

Was bedeutet: 100 0111 00

INS =7 M[00]

// Programm auf Maschinenebene, das die Zahlen 7 und 9 addiert

Einfügen von 7 in M[00]:	100 0111 00
Einfügen von 9 in M[01]:	100 1001 01
Laden von M[00] zu R[0]:	000 00 0
Laden von M[01] zu R[1]:	000 01 1
Addiere Register zu R[0]:	010 0
Speichern von R[0] in M[10]:	001 0 10

Fertig! ☺

Register	
Adresse	Inhalt
0	~~7~~ 16
1	9

Datenpeicher	
Adresse	Inhalt
00	7
01	9
10	16
11	

Ein Computerprogramm für unsere Architektur

// Programm auf Maschinenebene, das die Zahlen 7 und 9 addiert

Einfügen von 7 in M[00]:	100 0111 00
Einfügen von 9 in M[01]:	100 1001 01
Laden von M[00] zu R[0]:	000 00 0
Laden von M[01] zu R[1]:	000 01 1
Addiere Register zu R[0]:	010 0
Speichern von R[0] in M[10]:	001 0 10

sequenzielle Ausführung

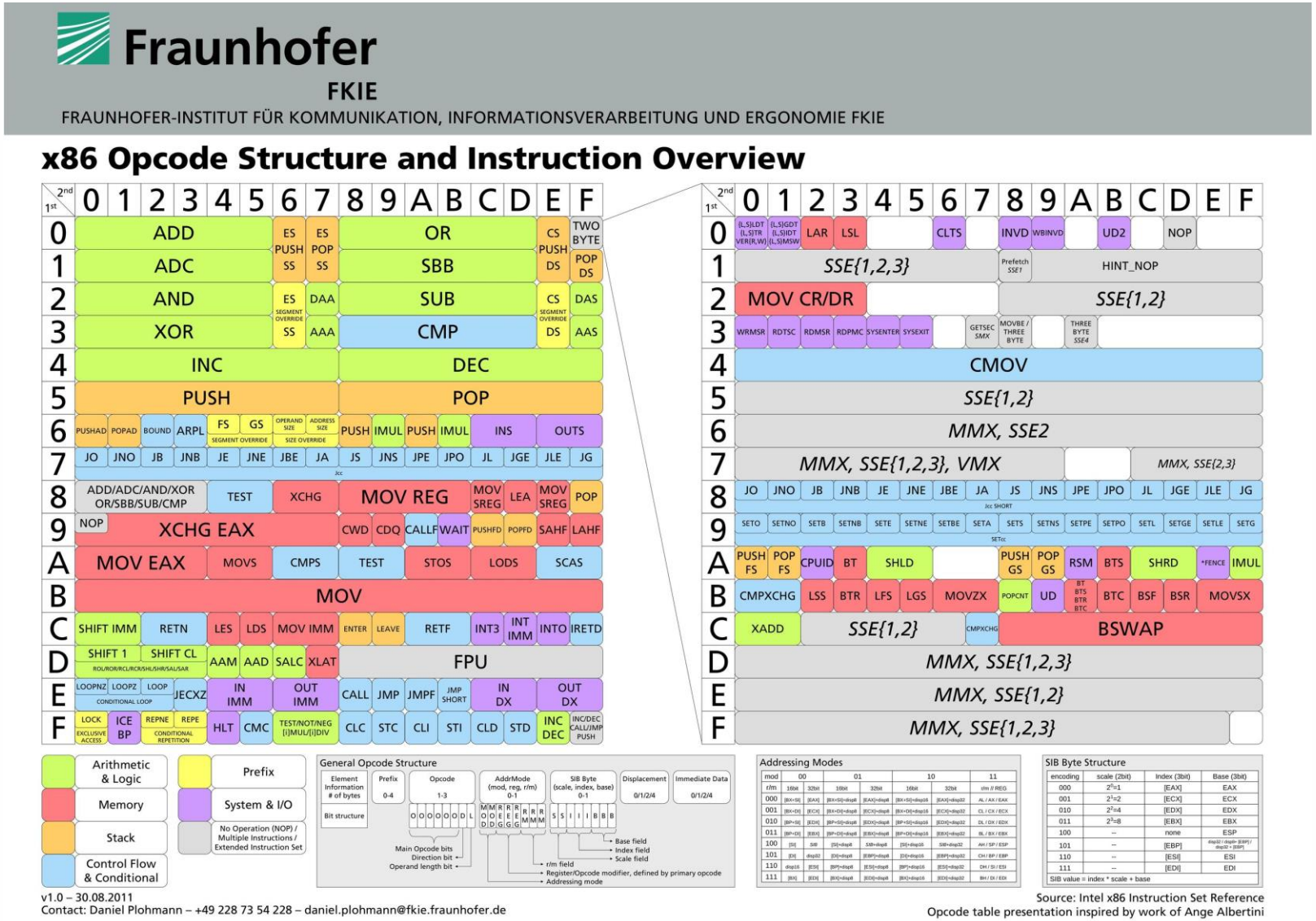
Beachten Sie, dass die Kodierung dieses Programms auf Maschinenebene eindeutig ist!
Wir können die Formatierung weglassen und das Programm so schreiben:

```
100011100 100100101 000000 000011 0100 001010
```

Ein echter Befehlssatz

Unser Befehlssatz: 5 Befehle (Load, Store, Add, Subtract, Insert)

Intel x86-64 Befehlssatz: zwischen 981 und 3683 Anweisungen (je nachdem, was als eine Anweisung gezählt wird...)

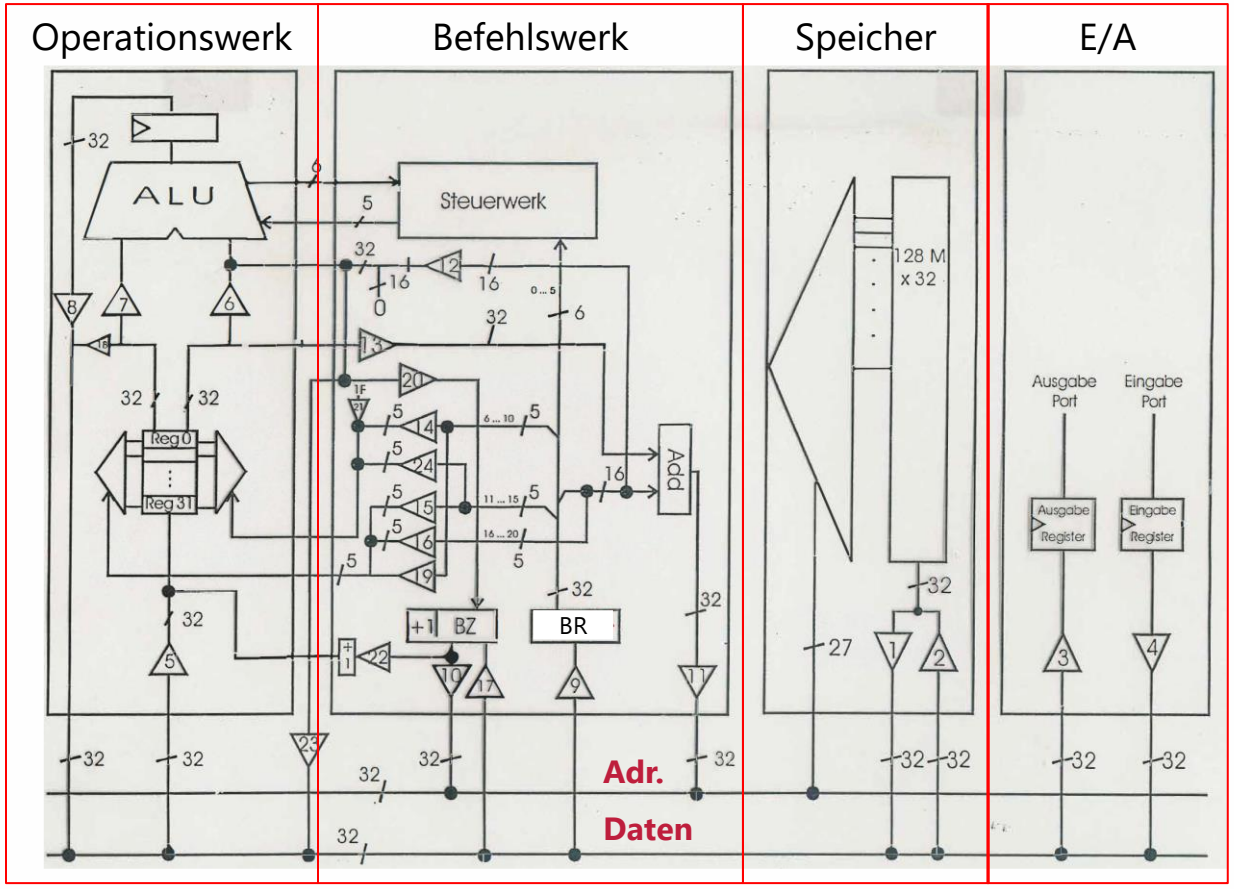


32-bit Mikrorechner Beispiel Architektur

Die Mikroprogramme der Steuerwerke liegen i. a. für eine bestimmte Rechnerarchitektur fest.

Dem überlagert ist das eigentliche Programmsteuerwerk (Befehlswerk) mit dem vom Benutzer aufgestellten Programm im Arbeitsspeicher.

Die dreieckigen Symbole kennzeichnen interne Datenwege.



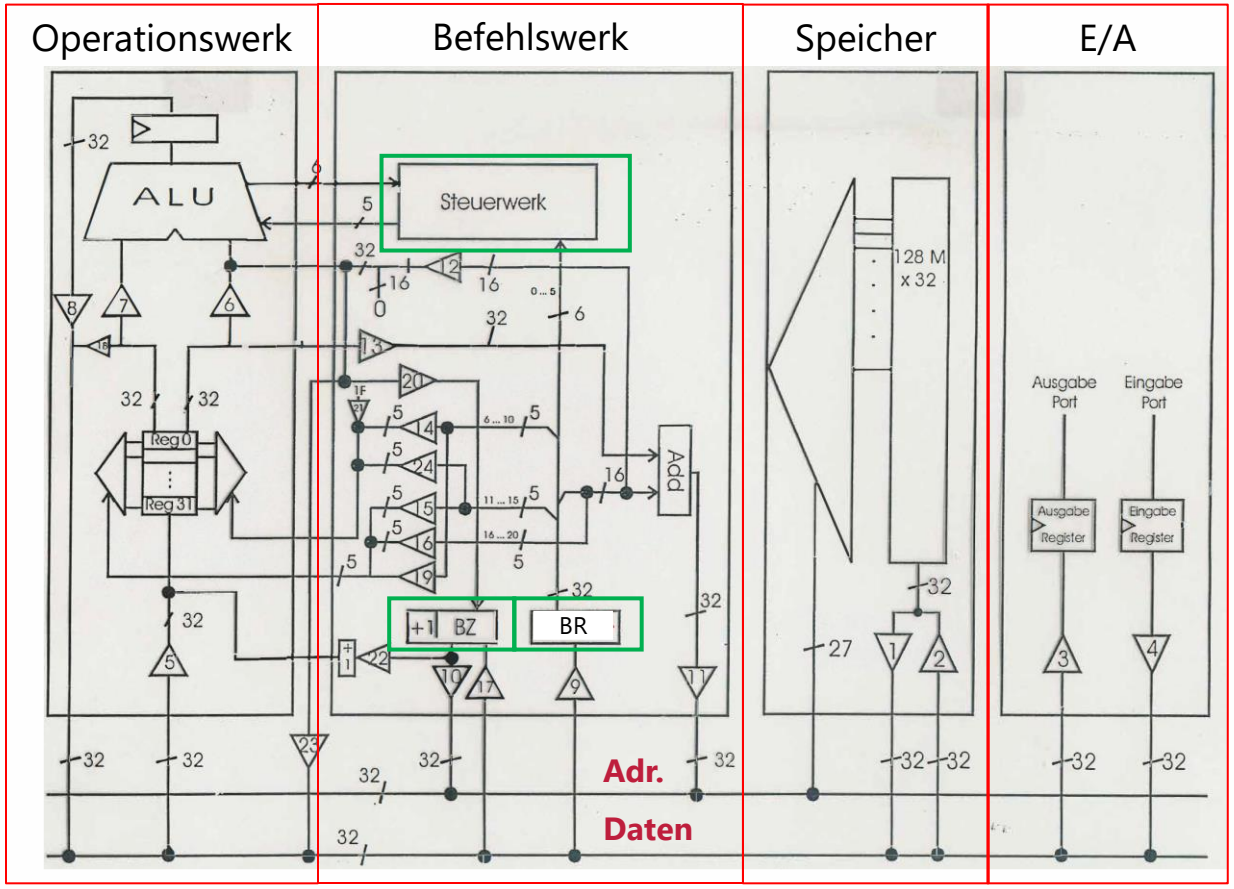
Befehlswerk

Das **Befehlswerk** Operationswerk besteht aus einem **Befehlsregister BR** und einem **Befehlszähler BZ.**

Das **BR** wird in der Regel aus dem Speicher geladen (Programmcode, 1,9)

Der BZ liefert in der Regel die Auswahlleitungen (10) für den Decoder **(=Adresse)** des Speichers, der in der Regel die Quelle für das Speicherwort ist.

Das **BR** liefert u.A. den **Opcode** (6bit) für das Steuerwerk der Mikrooperationen.



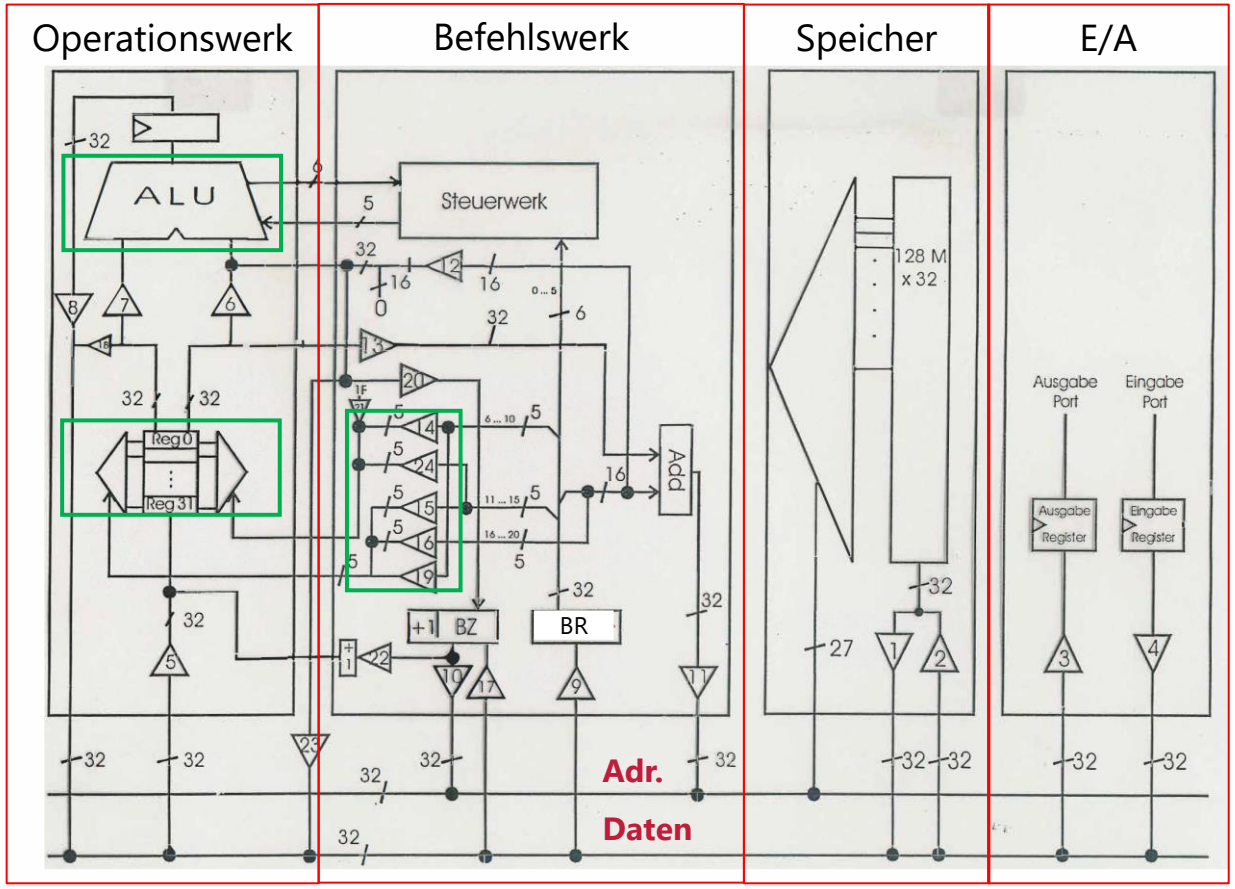
Operationswerk

Das Operationswerk besteht aus einer **ALU** sowie einem Satz von 32 Registern (**Registerbank**) als Zwischenspeicher für Operanden.

Die ALU Operanden können aus der Registerbank (6,7) geladen werden.

Die Register können aus dem Speicher, der Ein-/Ausgabe oder aus der ALU geladen werden

Die Auswahlleitungen für die Register kommen in der Regel direkt aus dem Befehlsregister **BR** (14,15,16,19,24)

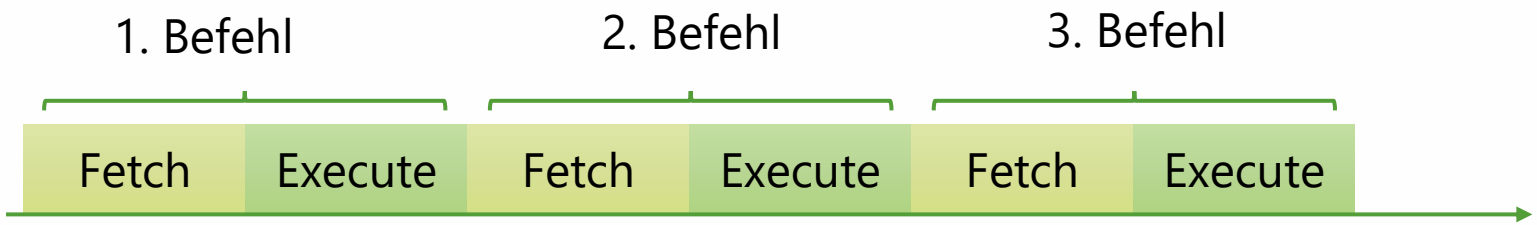


Befehl Ausführung

Der Ablauf der Befehlsabarbeitung findet in **mindestens zwei Phasen** statt:

1. Befehl holen (**Fetch Phase**)
2. Befehl interpretieren und Operation auslösen (**Execute Phase**)

Sequenzielle Ausführung



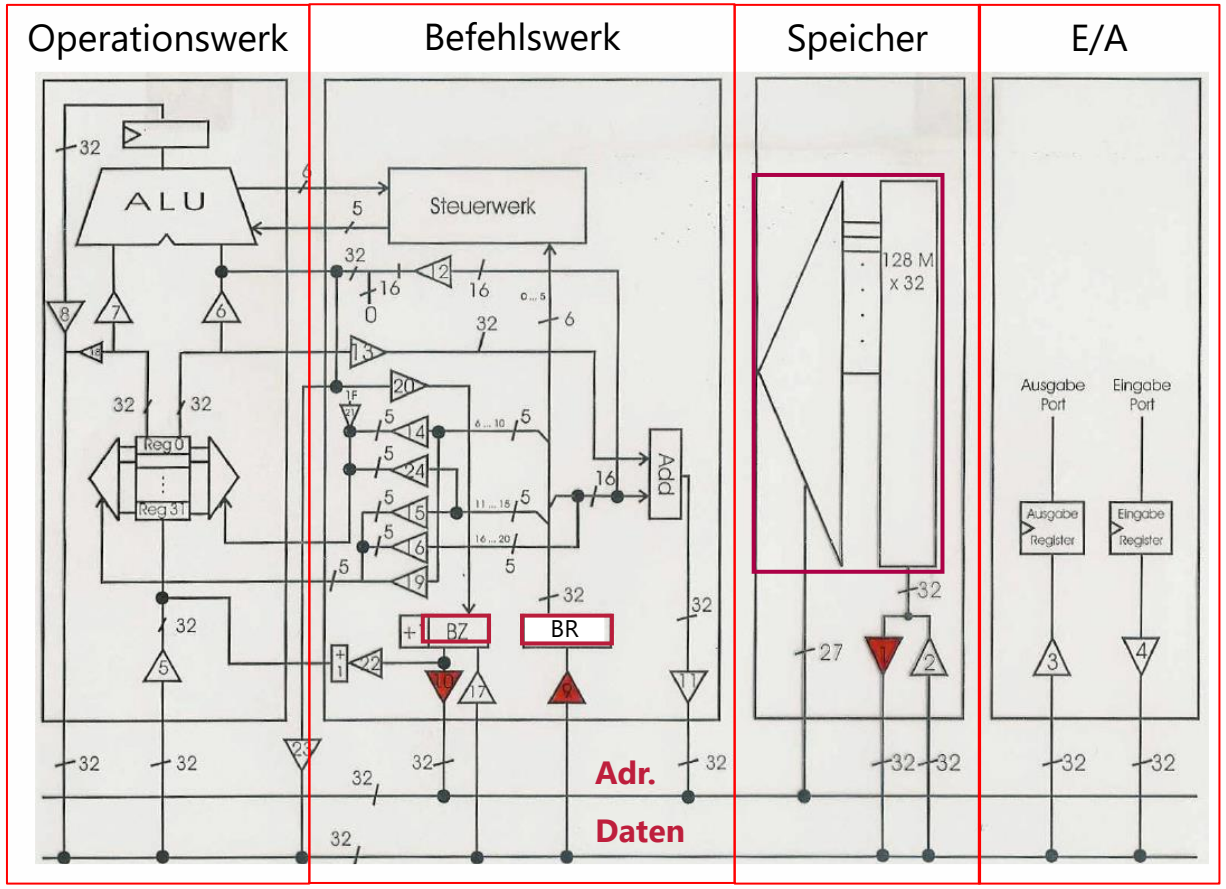
Befehlsabarbeitung I (Fetch)

In der **Fetch Phase** wird der Inhalt des BZ auf den Adressbus gelegt.

Mit der Adresse wird über den Decoder ein Wort im Speicher ausgewählt.

Danach wird das Speicherwort auf den Datenbus gelegt und in das Befehlsregister transportiert.

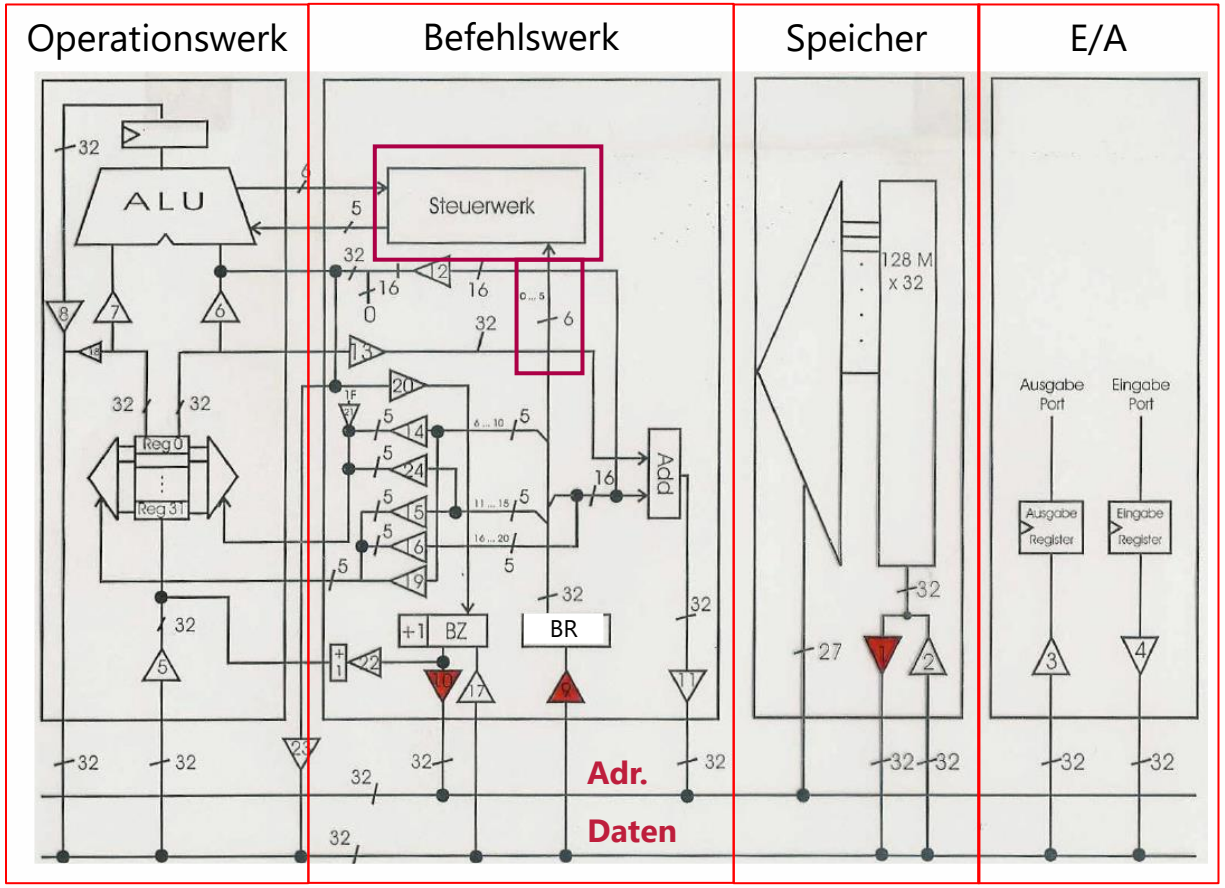
Der Befehlszähler wird um Eins hochgezählt (linearer Programmablauf).



Befehlsabarbeitung II (Execute)

In der **Execute Phase** wird der Opcode im Befehl (Bit 0-5 des BR, 6Bit) im Steuerwerk ausgewertet und der Mikrocode einer Operation (Laden/Speichern, Verarbeiten, Sprung…) ausgeführt.

Je nach Komplexität der Operation kann dies mehrere Takte in Anspruch nehmen.



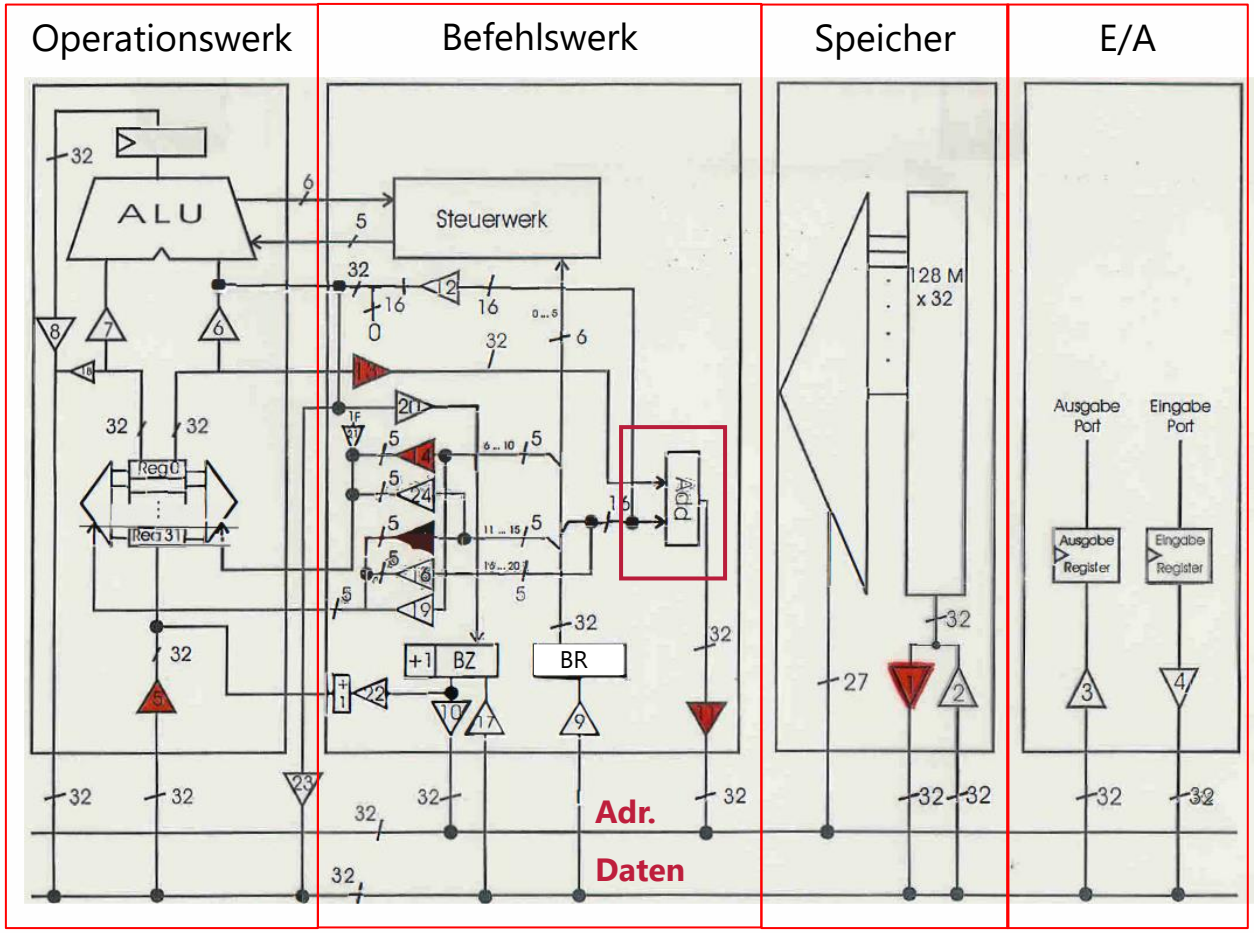
Beispiel Ladebefehl (Load Word, LW)

1. **Fetch Phase** wie vorher
2. **Execute Phase**:

   Opcode im Befehl im Steuerwerk auswerten

   Ein Register aus der Bank auswählen , dessen Inhalt zu einem Offset aus dem Befehl addiert wird, dies ergibt die Speicheradresse.

   Das Wort aus dem Speicher in das Zielregister laden.



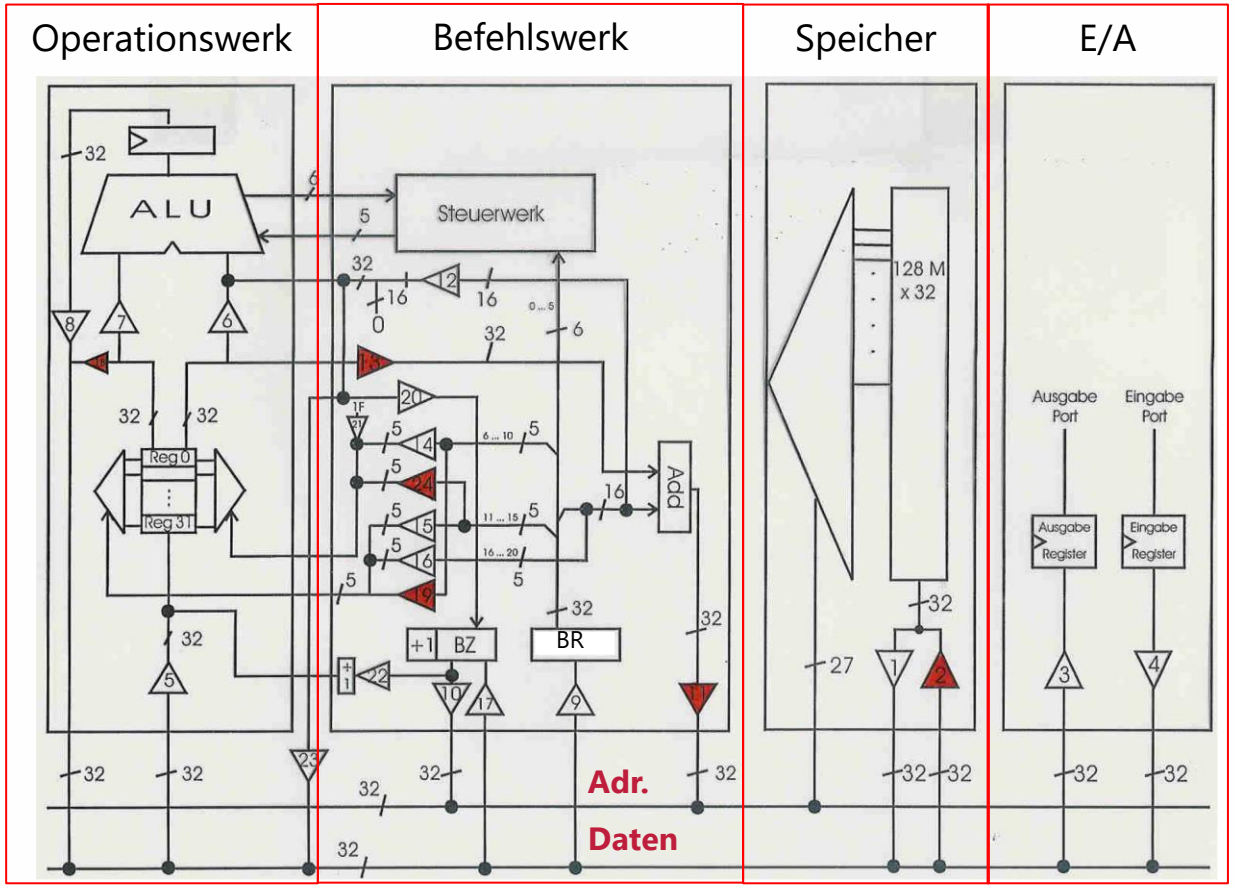
Beispiel Speicherbefehl (Store Word, SW)

1. **Fetch Phase** wie vorher
2. **Execute Phase**:

   Opcode im Befehl im Steuerwerk auswerten

   Ein Register aus der Bank auswählen, dessen Inhalt zu einem 16-Bit Offset aus dem Befehl addiert wird, dies ergibt die Speicheradresse.

   Das Wort aus Quellregister ist in den Speicher geladen.



ALU Operationen

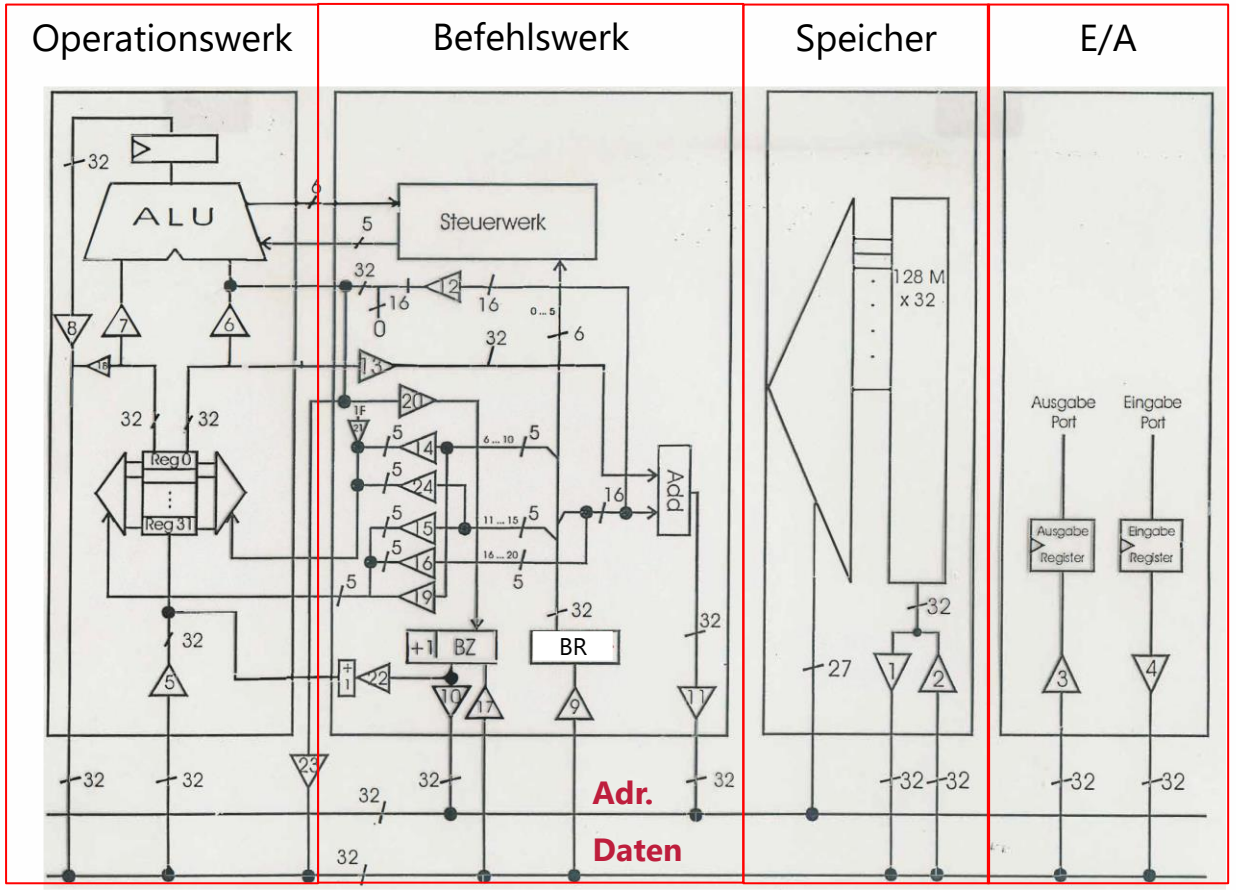
ALU Operationen werden typisch mit drei Registerbankadressen durchgeführt:

Im Befehl sind **zwei Quellregister** angegeben, die entsprechenden Registerinhalte werden zur ALU weitergegeben und dort verarbeitet.

Das Ergebnis wird in **einem Zielregister** abgelegt.

ALU braucht max. 3x 5-bit Adresse und 6-bit Op-code → 21 Bits

11 Bit sind übrig (z.B. für unmittelbare Operanden und Adressen)

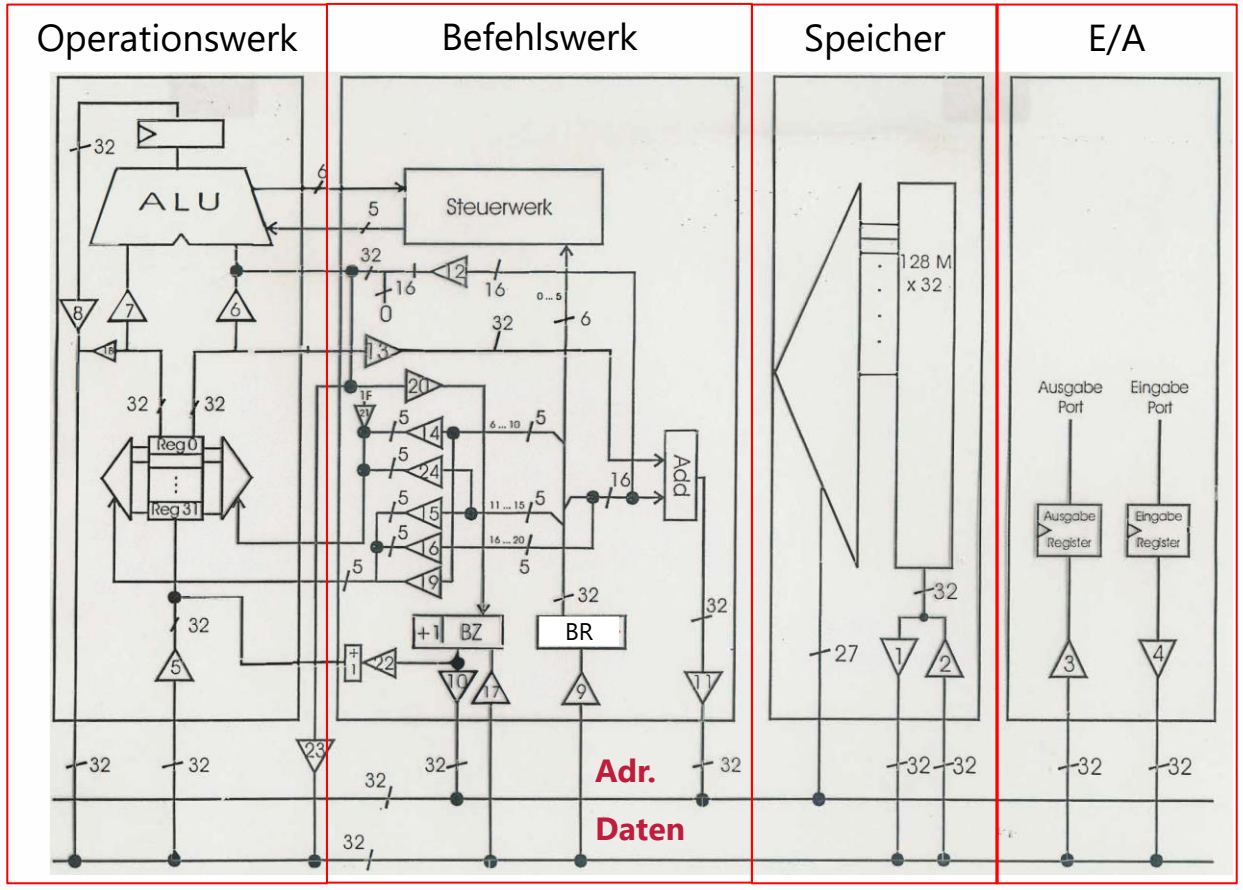


Sprungbefehle

Der Befehlszähler kann zur Abänderung eines linearen Programmablauf auch verändert werden (Sprungbefehle).

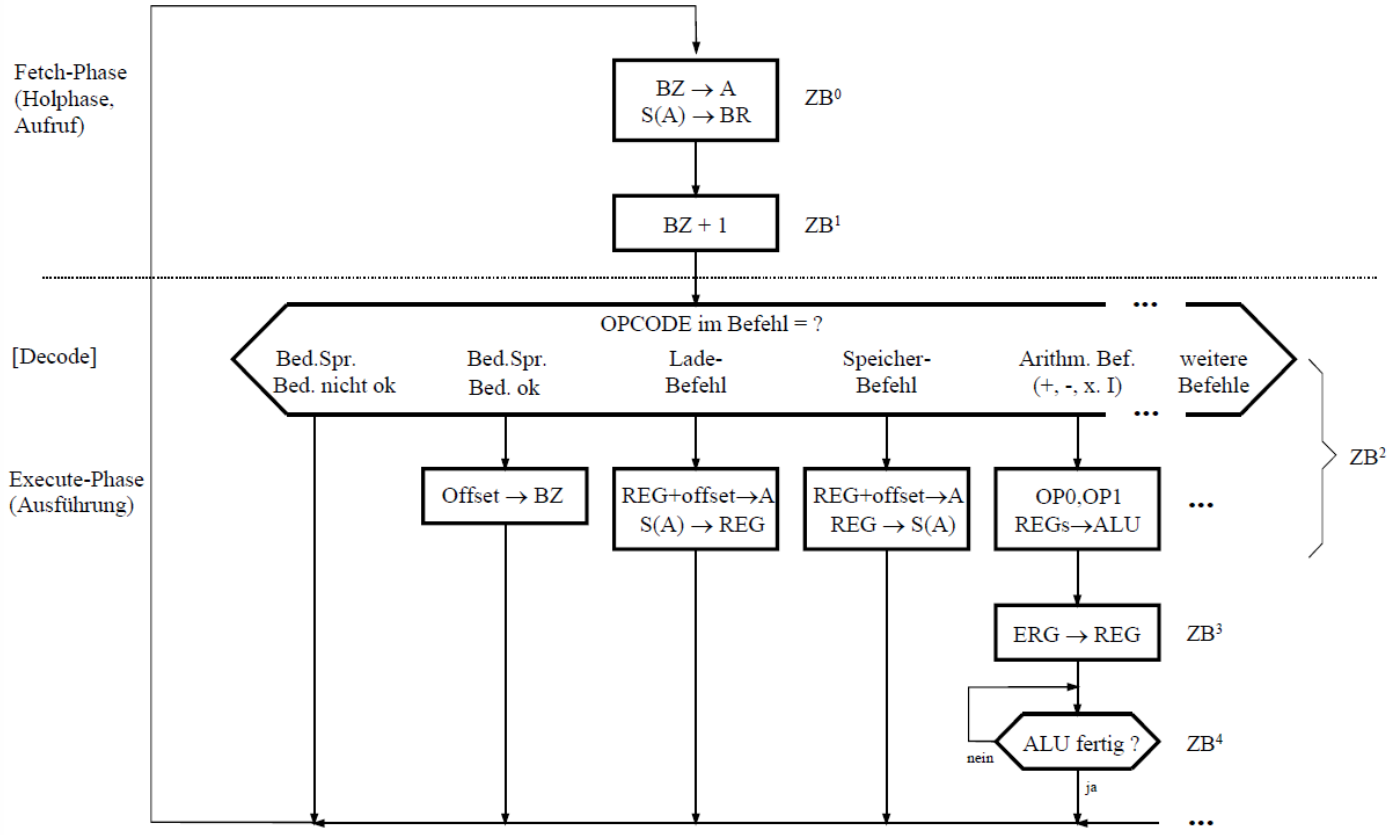
Aus einem Register kann eine neue 27-bit Befehlsadresse in den BZ geladen werden (direkter Sprung)

Oder es kann über das Befehlswort in BR ein 16 bit offset in den BZ geladen werden (bedingter Sprung).



Flussdiagramm des Befehlsablaufsteuerungs

Die Fetch Phase benötigt hier zwei Zustände, um die Erhöhung des Befehlszählers BZ von der Ausgabe der Adresse A für den Befehl zeitlich zu trennen.



Die Execute Phase benötigt mindestens einen bis mehrere zusätzliche Zustände, je nach Komplexität des Befehls.

Befehlswortstrukturen

Die Anweisungen einer Maschinensprache lassen sich grundsätzlich in vier Gruppen aufteilen:

a) Anweisungen, die eine von der Zentraleinheit auszuführende Datenmanipulation spezifizieren (**Rechenwerksbefehle**)

b) Anweisungen, die den Interpretationsablauf eines Maschinenprogramms steuern (**Sprungbefehle**)

c) Anweisungen, die den Datenfluss zwischen Hauptspeicher und Zentraleinheit bzw. innerhalb der Zentraleinheit steuern (**Transportbefehle**)

d) Anweisungen zur Steuerung der Ein-/Ausgabe (**E/A-Befehle**)

Befehlswortstrukturen

Die Struktur der Anweisungen einer konkreten Rechenanlage ist im wesentlichen geprägt durch die Struktur der Anweisungen aus Gruppe a). Deshalb sollen sie hier näher betrachtet werden.

Jede dieser Anweisungen hat im Prinzip fünf Funktionen:

1. Spezifizierung des ersten Operanden (d.h. seiner Adresse)
2. Spezifizierung des zweiten Operanden (entfällt bei Operationen auf nur einen Operanden)
3. Spezifizierung der Adresse, an die das Ergebnis gebracht werden soll
4. Spezifizierung der Operation, die auf den (beiden) Operanden ausgeführt werden soll
5. Spezifizierungen der Adresse derjenigen Anweisung, die als nächste ausgeführt werden soll

Ein Teil dieser Spezifizierungen kann implizit erfolgen. So entfällt bei fast allen Maschinensprachen die Angabe darüber, welche Anweisung als nächste ausgeführt werden soll: Man nimmt einfach einen linearen Ablauf an (BZ+1).

Maschinenbefehle

Maschinenbefehle sind in ihrer Struktur geprägt von der Angabe der Operandenadressen. Eine Adresse kennzeichnet den Platz im Speicher oder in einer Registerbank, an dem ein Operand oder ein Befehl gefunden werden kann.

Drei Adressen pro Befehl:

Für die Operation A + B = C müssten drei Operandenadressen angegeben werden:

Adresse von A	Adresse von B	Adresse von C	Op-Code (Addition)

Bei diesen **Dreiadressbefehlen** werden alle notwendigen Angaben explizit gemacht, so dass jede Anweisung selbsterklärend ist.

Zweiadressbefehlen

Bei **Zweiadressbefehlen** entfällt entweder

1. die Angabe der Adresse eines zweiten Operanden
   (wenn die Operation nur einen Operanden benötigt, z.B. ERG:= Negation von Operand A)
2. oder die Angabe der Zieladresse C.

   Da das Ergebnis im Fall 2. nach wie vor festgehalten werden muss, ist eine implizite Spezifizierung notwendig. Dazu gibt es zwei Möglichkeiten:

   Das Ergebnis wird immer in einem einmal festgelegten Register (z.B. Akkumulatorregister AK), abgelegt. Damit dessen Inhalt anderweitig verwendet werden kann, muss eine Transportanweisung vom Akkumulator zum Hauptspeicher und umgekehrt existieren.

   Die Adresse von B ist gleichzeitig Adresse des Ergebnisses. Damit stellt diese zweite Möglichkeit eine Verallgemeinerung der ersten dar.

Einadressbefehle

Unter Zuhilfenahme eines Akkumulatorregisters AK und der Auflösung eines Befehls in mehrere Befehle kann man mit einer Adresse pro Befehl auskommen.

Das Beispiel A + B = C muss dann z. B. in die folgende Sequenz von Ein-Adress-Befehlen aufgelöst werden:

1. Hole A aus dem Speicher nach AK
2. Addiere B aus dem Speicher zu AK
3. Speichere AK nach in den Speicher nach C

Diese Befehlssequenz ist von vielen Speichertransporten geprägt, die in der Regel länger dauern als der Transport innerhalb der CPU von Werten aus der Registerbank in die ALU. Daher haben moderne CPUs umfangreichere Registerbänke.

Allerdings müssen auch dann die Operanden erst in die Register transportiert werden (mit Hilfe von Transportbefehlen).

Nulladressbefehle

Bei Nulladressbefehlen müssen alle drei Adressspezifikationen implizit erfolgen.

Man legt dabei einen sogenannten Stapelspeicher (Stack), nach dem Last In/First Out (LIFO)-Prinzip) zugrunde, dessen zwei oberste Speicherplätze Akkumulatorfunktionen haben.

Solch ein Stack wird auch verwendet, um kurzzeitig Variablen aus der Registerbank zwischen zu speichern, wenn die Register für andere Operationen benötigt werden.

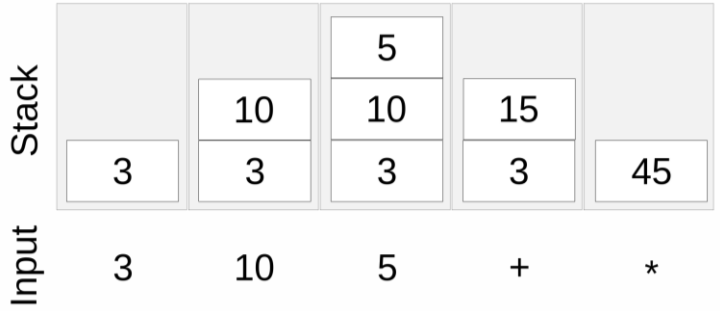
Die typischen Transportanweisungen dafür sind:

PUSH (auf den Stapel geben)

POP (vom Stapel abziehen)

Beispiel: 3*(10+5) ist Infixnotation

mit Postfixnotation, braucht man keine Klammern
3 10 5 + *



Adressierung

Adressierung von Speicherworten

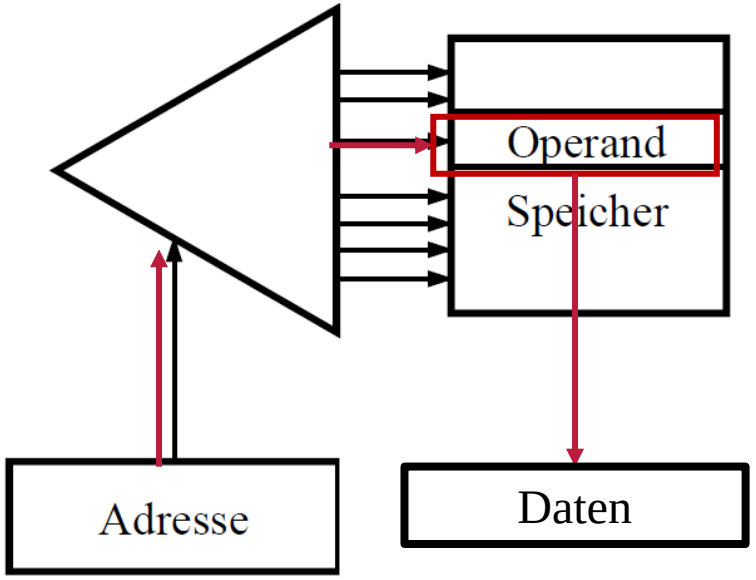
Die Adresse von Operanden kann sich auf verschiedene Speicher beziehen.

- CPU intern: hier werden Worte aus der Registerbank adressiert.
- Externer Speicher: hier wird die Adresse über Leitungen des Systembusses an den Decoder des Speichers gegeben

Quelle der Adresse können **Teile des Befehlsworts** sein, ein **Registerinhalt** oder sogar ein **Inhalt eines externen Speicherwortes**.

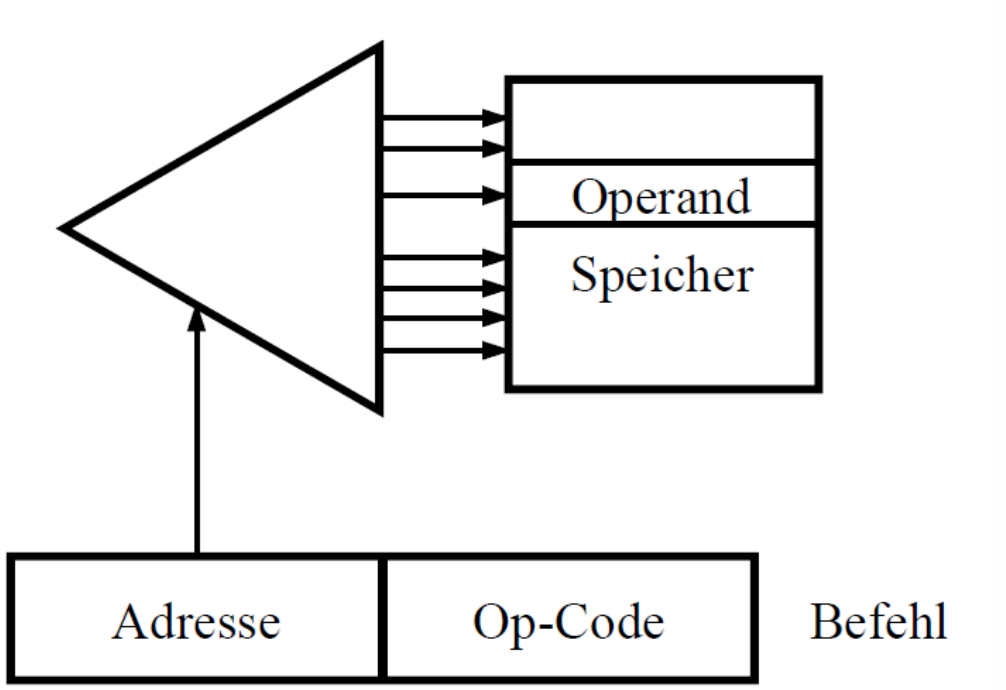
Es gibt vielfältige Möglichkeiten zur Adressierung:

- Direkte Adressierung
- Relative Adressierung



Direkte Adressierung

Bei der direkten Adressierung wird die im Befehlswort enthaltene Adresse an den Decoder des Speichers gegeben, der den zu lesenden Operanden enthält.



Relative Adressierungen

Adressenänderung I. Art: Basisadressierung

Adressenänderung II. Art: Indirekte Adressierung

Adressenänderung III. Art: Indizierte Adressierung

I. Art - Basisadressierung

Die im Befehl referierte Adresse stellt eine Adressverschiebung (displacement, auch offset genannt) dar, die additiv einer in einem Register gespeicherten Basisadresse hinzugefügt wird:

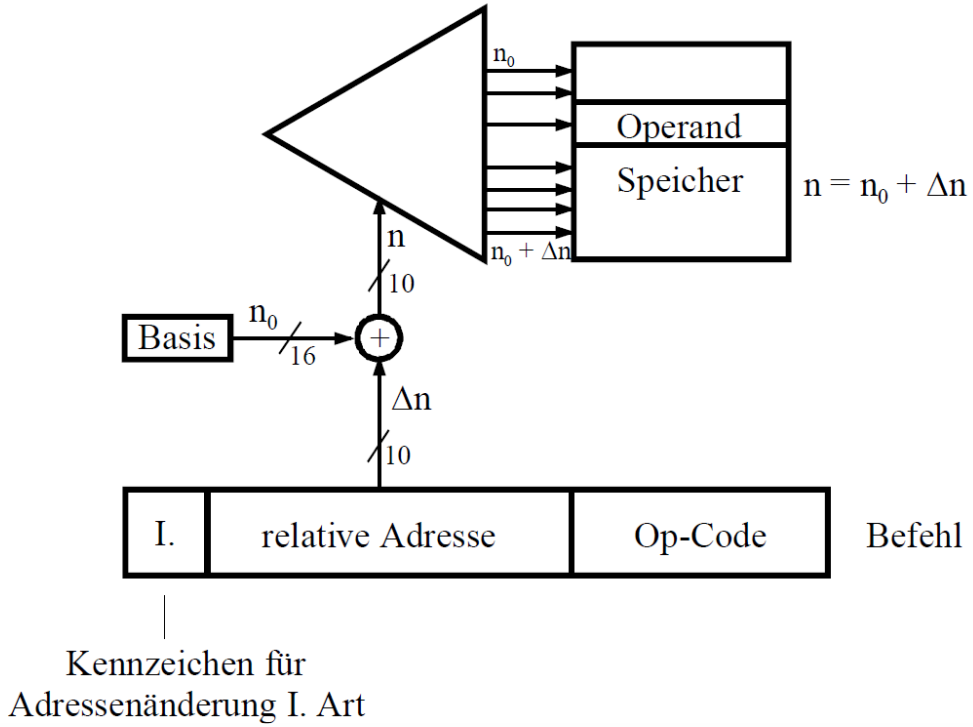
Absolute Adresse

(n) = feste Basisadresse (n_0) + relative Adresse (Δn)

Programmieren mit relativen Adressen hat folgenden Vorteil:

a) Die relativen Adressen sind kürzer als die absoluten Adressen, weil sie nur eine Untermenge abdecken.

b) Man braucht sich bei der Adressenwahl nicht um die tatsächliche Speicherbelegung zu kümmern.



I. Art - Seitenadressierung (Paging)

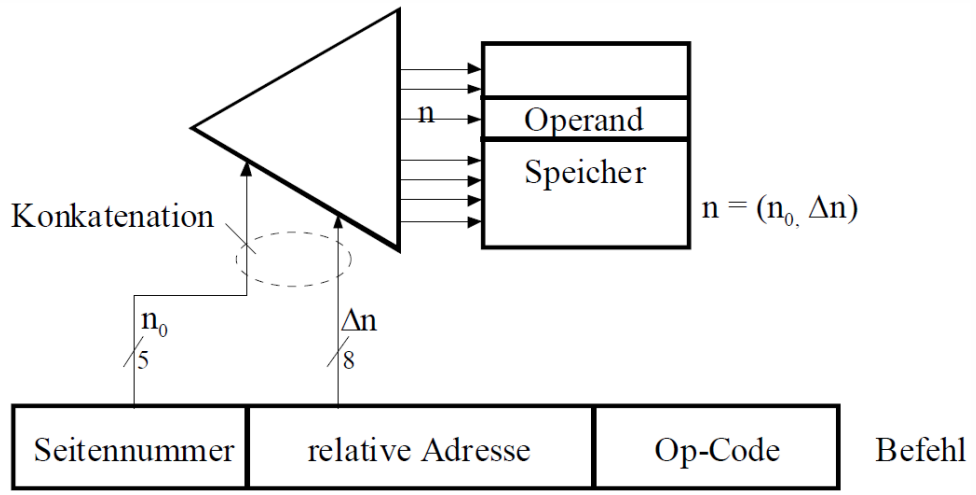
Der logische Adressraum kann in konstant große Unterräume, sogenannte Seiten, aufgeteilt werden.

Seiten ermöglichen eine relative Adressierung der zusammenhängenden Speicherplätze.

Die logische Adresse setzt sich aus Seitennummer und relativer Adresse zusammen.

Durch Konkatenation der Seitennummer und relativen Adresse entsteht die gesamte logische Adresse.

Im Beispiel werden 5 Bits für die Seitennummer und 8 Bits für die relative Adresse zu einer 13 Bit breiten logischen Adresse kombiniert, ohne dass eine Addition zur Bestimmung der absoluten Adresse erforderlich ist.



II. Art -Indirekte Adressierung

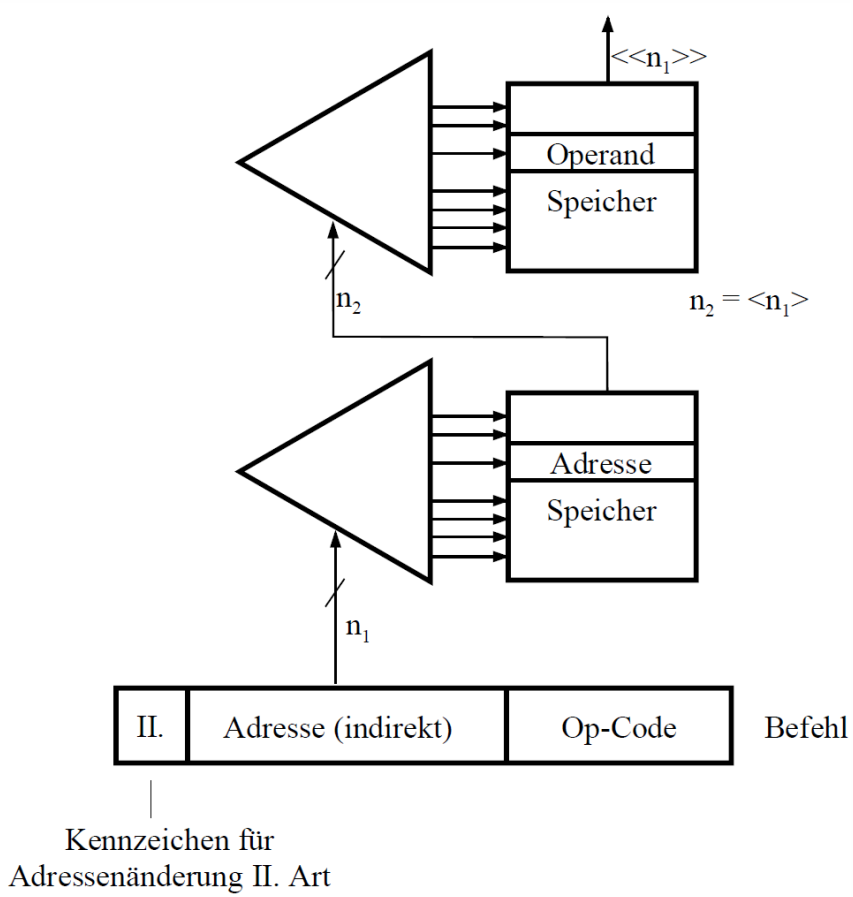
Bei der indirekten Adressierung ergibt sich die Adresse des gesuchten Wertes aus dem Inhalt des Speicherwortes der angegebenen Adresse („Adresse von Adresse“):

Gesuchtes Wort = <<Angegebene Adresse >>

Diese Adressierungsart macht Sinn, wenn die eigentliche Adresse auch erst Ergebnis einer Berechnung ist.

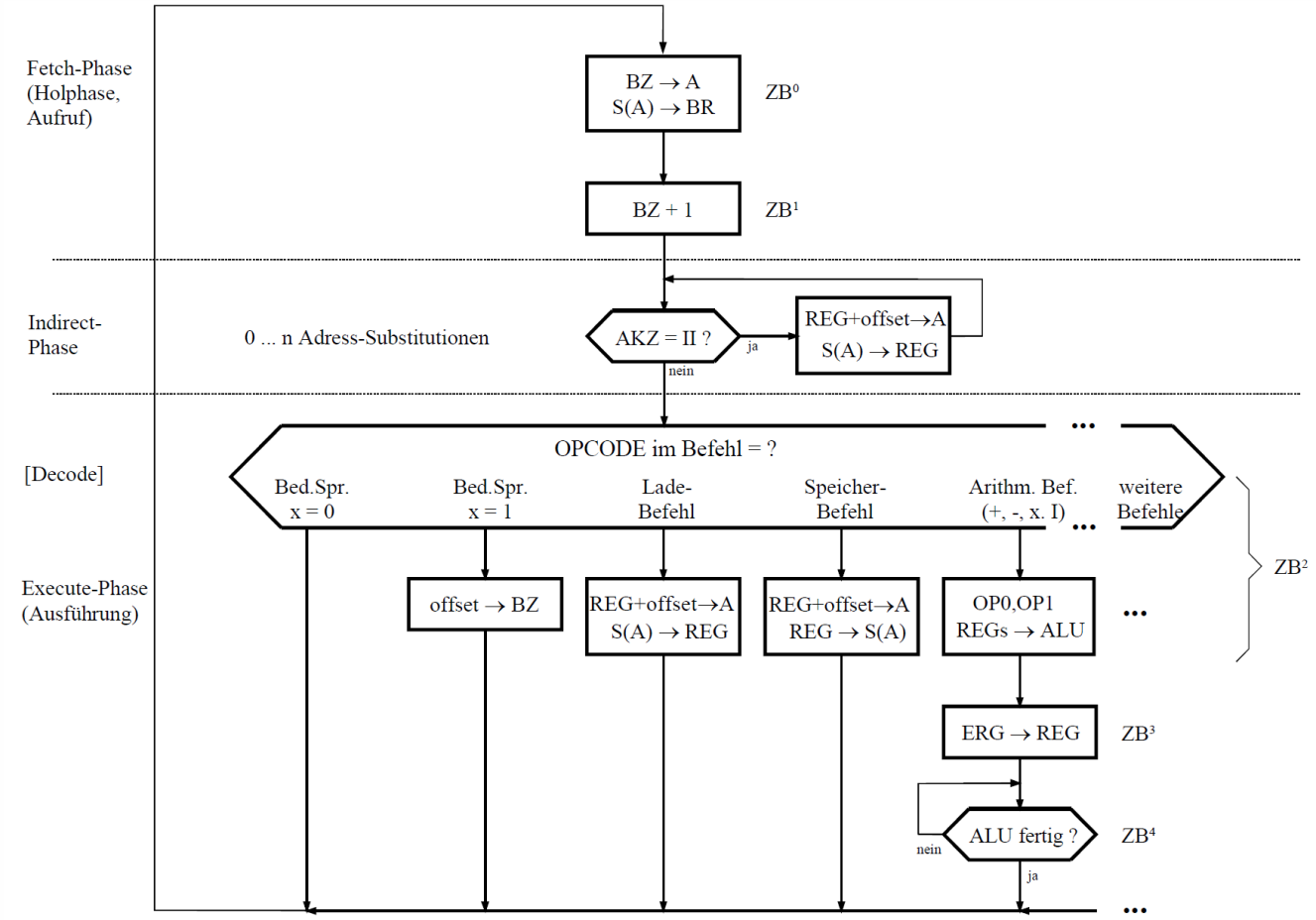
Die Wortstruktur und den Zugriffsmechanismus unter Benutzung von zwei Speichern zeigt das Bild.

Üblich ist auch, den Inhalt eines Registers aus der Registerbank als Adresse zu verwenden.



II. Art -Indirekte Adressierung

Für die Ausführung der Adressänderung ist das Befehlswerk entsprechend dem Ablaufprogramm im Bild zu modifizieren (zusätzlicher Zustand).



III. Art - Indizerte Adressierung

Adressen sind additiv mit einer im Indexregister gespeicherten Verschiebung kombiniert.

Im Gegensatz zur Adressierung mit Hilfe einer Basisadresse befindet sich dabei die Adressverschiebung (displacement) in einem im Befehl referierten Indexregister.

Adresse n:= Anfangs- oder Endadresse n0 + Inhalt eines Indexregisters <I>

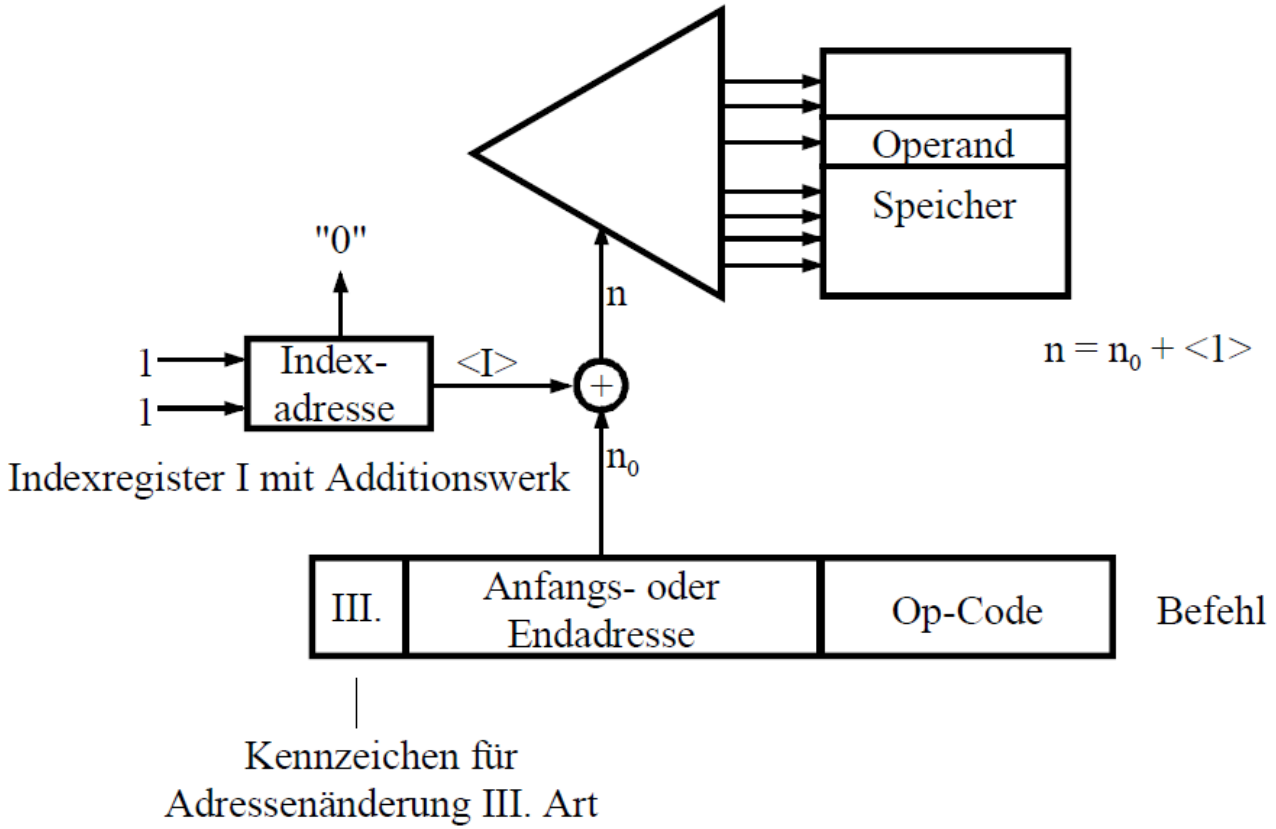
Für die Adressenrechnung werden ein Additionswerk, Indexregister, Kennzeichen für Adressenänderung III. Art und Zusatzbefehle benötigt.

Zusatzbefehle umfassen das Laden des Indexregisters mit negativer Zahl (-n1), Änderungen des Indexregisters (+1, -1) und bedingte Sprünge basierend auf dem Indexregisterwert.

Diese Art der Adressierung ähnelt der Basisadressierung, ermöglicht jedoch direkte Änderungen der Adressberechnung im Indexregister (+1, -1 usw.).

III. Art - Indizerte Adressierung

Diese Art ist immer dann sinnvoll, wenn z.B. ein Block von zusammenhängenden Daten nacheinander bearbeitet werden soll.



Registeradressierungen

Direkte Register-Adressierung: Analog zur direkten Adressierung (im Speicher) wird bei dieser Form die Registerbezeichnung R (Kurzadresse) zur Ermittlung des Operanden benutzt, d.h. der Operand ist gleich dem Inhalt des durch R bezeichneten Registers: Operand = <R>.

Entsprechend gibt es die indirekte Register-Adressierung: Bei dieser Adressierungsform gilt: Operand = <<R>>, d.h. der Operand ist gleich dem Inhalt derjenigen Speicherzelle, deren Adresse in dem durch R spezifizierten Register steht.

Beispiele für Adressierungsarten-1

Adressierungsart	Beispiel	Bedeutung	Anwendung
Register	Add R4 , R3	Regs [R4] □Regs [R4]+Regs[R3]	Wert ist im Register.
Immediate	Add R4 , #3	Regs [R4] □Regs [R4]+3	Operand ist eine Konstante
Displacement	Add R4 , 100(R1)	Regs [R4] □Regs [R4]+Mem [100+Regs[R1]]	Lokale Variable
Register deferred or indirect	Add R4 , (R1)	Regs [R4] □Regs [R4]+Mem [Regs[R1]]	Register dient als Pointer.
Direct or absolute	Add R1 , (1001)	Regs [R1] □Regs [R1]+Mem [1001]	Manchmal nützlich für Zugriff auf statische Daten
Indexed	Add R3 , (R1 + R2)	Regs [R3] □Regs [R3]+Mem [Regs[R1]+Regs[R2]]	Nützlich für array-Addressierung: R1=base of array; R2=index amount

Beispiele für Adressierungsarten-2

Adressierungsart	Beispiel	Bedeutung	Anwendung
Memory indirect	Add R1 , @(R3)	Regs [R1] □Regs [R1]+Mem[Mem[Regs[R3]]]	Wenn R3 die Adresse eines Pointers *p* enthält, dann bekommen wir **p*.
Autoincrement	Add R1 , (R2)+	Regs [R1] □Regs [R1]+ Mem[Regs[R2]]Regs [R2]	Nützlich für arrays mit Schleifen. R2 ziegt auf den array-Anfang; jeder Zugriff erhöht R2 um die Größe d eines array Elements
Autodecrement	Add R1 , -(R2)	Regs [R2] □Regs [R2]-*d* Regs [R1] □Regs [R1]+ Mem[Regs [R2]]	Genauso wie Autoincrement
Scaled	Add R1 , 100 (R2)[R3]	Regs [R1] □Regs [R1]+ Mem[100 + Regs [R2] + Regs	Indizierung von Feldern mit Datentypen der Länge d

Adressierungsarten von Mikrorechnerarchitekturen

Typ	Vorteile	Nachteile
Register-Register (0,3)	Einfaches Befehlsformat fester Länge. Einfaches Modell zur Code Generierung. Instruktionen brauchen alle etwa gleichviel Takte.	Höherer IC als die beiden anderen.
Register-Speicher (1,2)	Daten können zugegriffen werden, ohne sie erst zu laden. Instruktions-Format einfach.	Jeweils ein Operand in einer zweistelligen Operation wird zerstört. CPI variiert je nachdem von wo die Operanden geholt werden.
Speicher-Speicher (3,3)	Sehr kompakt. Braucht keine Register für Zwischenergebnisse	Hoher CPI. Viele Speicherzugriffe. Speicher-CPU-Flaschenhals.

Legende: IC = Instruction Count, Zahl der notwendigen Befehle
CPI = Clocks per Instruction, Zahl der Taktzyklen pro Befehl